

**课 程 实 验 报 告**

**课程名称： 数 据 结 构 实 验**

**专业班级： 计科1409班**

**学 号： U201414800**

**姓 名： 刘 一 龙**

**指导教师： 李 剑 军**

**报告日期： 2015年 12 月 10 日**

**计算机科学与技术学院**

目录

[1基于顺序存储结构实现线性表的基本运算 1](#_Toc440028098)

[1.1 问题描述 1](#_Toc440028099)

[1.2 顺序表演示系统设计 1](#_Toc440028100)

[1.2.1 系统总体设计 1](#_Toc440028101)

[1.2.2 有关常量和类型定义 2](#_Toc440028102)

[1.2.3 算法设计 2](#_Toc440028103)

[1.3 顺序表演示系统实现与测试 4](#_Toc440028104)

[1.3.1 系统实现 4](#_Toc440028105)

[1.3.2 系统测试 29](#_Toc440028106)

[1.4 实验小结 31](#_Toc440028107)

[2基于链式存储结构实现线性表的基本运算 33](#_Toc440028108)

[2.1 问题描述 33](#_Toc440028109)

[2.2.1 系统总体设计 33](#_Toc440028110)

[2.2.2 有关常量和类型定义 34](#_Toc440028111)

[2.2.3 算法设计 34](#_Toc440028112)

[2.3.1 系统实现 37](#_Toc440028113)

[2.3.2 系统测试 64](#_Toc440028114)

[2.4 实验小结 66](#_Toc440028115)

[3基于顺序存储结构实现栈的基本运算 68](#_Toc440028116)

[3.1 顺序栈实验 68](#_Toc440028117)

[3.1.1 问题描述 68](#_Toc440028118)

[3.1.1.1 实验任务 68](#_Toc440028119)

[3.1.1.2 实验目的 68](#_Toc440028120)

[3.1.1.3 实验算法 68](#_Toc440028121)

[3.1.2 顺序栈设计 68](#_Toc440028122)

[3.1.2.1 系统总体设计 68](#_Toc440028123)

[3.1.2.3 算法设计 69](#_Toc440028124)

[3.1.3 顺序栈实现与测试 70](#_Toc440028125)

[3.1.3.1 实现部分 70](#_Toc440028126)

[3.1.3.2 测试部分 95](#_Toc440028127)

[3.2 表达式求值实验 98](#_Toc440028128)

[3.2.1 问题描述 98](#_Toc440028129)

[3.2.1.1 实验目的 98](#_Toc440028130)

[3.2.1.2 实验任务 98](#_Toc440028131)

[3.2.2算法设计部分 98](#_Toc440028132)

[3.2.2.1 有关常量和类型定义 98](#_Toc440028133)

[3.2.2.2 实现算法 98](#_Toc440028134)

[3.2.3实现与测试部分 100](#_Toc440028135)

[3.2.3.1 实现部分 100](#_Toc440028136)

[3.2.3.2 测试部分 101](#_Toc440028137)

[3.3 实验小结 101](#_Toc440028138)

[4基于循环队列存储结构实现队列的基本运算 104](#_Toc440028139)

[4.1 问题描述 104](#_Toc440028140)

[4.1.1 实验任务 104](#_Toc440028141)

[4.1.2 实验目的 104](#_Toc440028142)

[4.1.3 实验算法 104](#_Toc440028143)

[4.2 循环队列设计 104](#_Toc440028144)

[4.2.1 系统总体设计 104](#_Toc440028145)

[4.2.2 有关常量和类型定义 104](#_Toc440028146)

[4.2.3 算法设计 105](#_Toc440028147)

[4.3 顺序栈实现与测试 106](#_Toc440028148)

[4.3.1 实现部分 106](#_Toc440028149)

[4.3.2 测试部分 120](#_Toc440028150)

[4.4 实验小结 122](#_Toc440028151)

[5基于二叉链表实现二叉树的基本运算 124](#_Toc440028152)

[5.1 问题描述 124](#_Toc440028153)

[5.1.1 实验任务 124](#_Toc440028154)

[5.1.2 实验目的 124](#_Toc440028155)

[5.1.3 实现算法 124](#_Toc440028156)

[5.2 二叉树设计 124](#_Toc440028157)

[5.2.1 系统总体设计 124](#_Toc440028158)

[5.2.2 有关常量和类型定义 125](#_Toc440028159)

[5.2.3 算法设计 126](#_Toc440028160)

[5.3 二叉树实现与测试 132](#_Toc440028161)

[5.3.1 实现部分 132](#_Toc440028162)

[5.3.2 测试部分 177](#_Toc440028163)

[5.4 实验小结 197](#_Toc440028164)

[6基于邻接表实现图的基本运算 199](#_Toc440028165)

[6.1 问题描述 199](#_Toc440028166)

[6.1.1 实验任务 199](#_Toc440028167)

[6.1.2 实验目的 199](#_Toc440028168)

[6.1.3 实现算法 199](#_Toc440028169)

[6.2 基于邻接表的图设计 199](#_Toc440028170)

[6.2.1 系统总体设计 199](#_Toc440028171)

[6.2.2 有关常量和类型定义 200](#_Toc440028172)

[6.2.3 算法设计 201](#_Toc440028173)

[6.3 基于邻接表的图实现与测试 207](#_Toc440028174)

[6.3.1 实现部分 207](#_Toc440028175)

[6.3.2 测试部分 237](#_Toc440028176)

[6.4 实验小结 242](#_Toc440028177)

[参考文献 244](#_Toc440028178)

# 1基于顺序存储结构实现线性表的基本运算

1.1 问题描述

一、基于顺序存储，实现线性表的基本操作。线性表的物理结构如图1-1所示。



图1-1 顺序表物理结构示意图

二、实验目的

（1）加深对线性表的概念、基本运算的理解；

（2）熟练掌握线性表的逻辑结构与物理结构的关系；

（3）物理结构采用顺序表,熟练掌握线性表的基本运算的实现。

三、实现算法

线性表的初始化、销毁、重置；

线性表的空表查询、长度查询、元素值查询、位序值查询；

线性表的前驱/后继元素查询、插入、删除、遍历。

1.2 顺序表演示系统设计

1.2.1 系统总体设计

本系统为简易顺序存储线性表使用系统。本系统可自动初始化线性表与自动销毁线性表。本系统可自动将线性表保存在文件output.txt中。本系统具有以下几大功能：

1. 查询功能

查询空表状态、查询表长、按元素值查询位序、按位序查询元素值、

查询前驱元素值、查询后继元素值

1. 修改功能

插入新元素、删除旧元素

1. 打印功能

遍历输出完整线性表

1.2.2 有关常量和类型定义

一、所有实验公有部分：见程序清单DataStructure.h

二、本实验常量和类型定义

1.线性表存储空间默认初始分配量

#define LIST\_INIT\_SIZE 100

2.线性表存储空间一次的分配增量

#define LISTINCREMENT 10

3．定义顺序存储的线性表数据结构

typedef struct

{

ElemType \* elem; //存储空间基址

int length; //当前线性表长度

int listsize; //当前分配存储总容量(以sizeof(ElemType)为单位)

} Sqlist;

1.2.3 算法设计

一、InitaList(&L) {

L.elem = malloc; //为基址分配定长连续内存单元

L.length = 0; //表长置为0

L.listsize = LIST\_INIT\_SIZE;//表容量置为正确值

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

二、DestroyList(&L) {

free(L.elem); //释放定长连续内存单元

L.elem = NULL; //基址置为空

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

三、ClearList (&L) {

L.length = 0; //表长置为0

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

四、ListEmpty(L) {

//表长为0时返回TRUE，表长不为0时返回FALSE

return L.length == 0 ? TRUE : FALSE;

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

五、ListLength(L) {

return L.length; //返回表长

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

六、GetElem(L, i,&e) {

return L.elem[i-1]; //返回第i-1个元素

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

七、LocateElem(L, e, compare()) {

//遍历线性表，直至表尾或查找到目标元素

for (i = 1; i <= ListLength (L) && cmp (e, L.elem[i - 1]) == FALSE; i++);

//i不大于表长时说明查找成功，返回位序；否则说明查找失败返回0

return (i <= ListLength (L) ? i : 0);

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

八、PriorElem(L, cur\_e, &pre\_e) {

cur\_pos = LocateElem (L, cur\_e, cmp); //定位当前元素位序

pre\_e = L.elem[cur\_pos - 2]; //返回前驱元素

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

九、NextElem(L, cur\_e, &next\_e) {

cur\_pos = LocateElem (L, cur\_e, cmp); //定位当前元素位序

next\_e = L.elem[cur\_pos]; //返回后继元素

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

十、ListInsert(&L,i,e) {

//插入位置及以后的元素统统从后往前向右移动

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

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

L.elem[i - 1] = e; //插入元素

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

十一、ListDelete(&L,i,&e) {

//删除位置之后的元素从前往后向左移动

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

{

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

}

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

十二、ListTraverse(L, visit()) {

//遍历线性表

for (i = 1; i <= ListLength (L); i++)

{

visitor (L.elem[i - 1]);

}

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

1.3 顺序表演示系统实现与测试

1.3.1 系统实现

一、编程环境

编辑器：Sublime Text 3/GVim

编译器：MinGW-gcc 4.9.1(C99标准)

调试器：MinGW-gdb 7.8

二、文件调用关系



三、函数调用关系



四、程序清单

1.DataStructure.h：所有实验公有头文件

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验一：顺序存储线性表

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef DATASTRUCTURE\_H\_INCLUDED

#define DATASTRUCTURE\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

//自定义宏

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

//Status是函数的类型,其值是函数结果状态代码

typedef int Status;

//将ElemType定义为int型

typedef int ElemType;

#endif /\*DATASTRUCTURE\_H\_INCLUDED\*/

2.SqList.h：抽象数据类型定义

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验一：顺序存储线性表抽象数据类型定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef SQLIST\_H\_INCLUDED

#define SQLIST\_H\_INCLUDED

#include "DataStructure.h"

//线性表存储空间默认初始分配量

#define LIST\_INIT\_SIZE 100

//线性表存储空间一次的分配增量

#define LISTINCREMENT 10

//定义顺序存储的线性表数据结构

typedef struct

{

ElemType \* elem; //存储空间基址

int length; //当前线性表长度

int listsize; //当前分配存储总容量(以sizeof(ElemType)为单位)

} Sqlist;

/\*\*

\* 构造一个空的线性表

\* @param L 线性表地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitList (Sqlist \* L);

/\*\*

\* 销毁线性表L

\* 初始条件:线性表L已存在

\* @param L 线性表地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DestroyList (Sqlist \* L);

/\*\*

\* 将L重置为空表

\* 初始条件:线性表已存在

\* @param L 线性表地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ClearList (Sqlist \* L);

/\*\*

\* 查询L是否为空表

\* 初始条件:线性表L已存在

\* @param L 线性表L

\* @return 若L为空表,则返回TRUE,否则返回FALSE

\*/

Status ListEmpty (Sqlist L);

/\*\*

\* 查询L的当前长度

\* 初始条件:线性表L已存在

\* @param L 线性表L

\* @return L中数据元素的个数

\*/

int ListLength (Sqlist L);

/\*\*

\* 查询L中第i个数据元素的值

\* 初始条件：线性表已存在，1 ≤ i ≤ ListLength(L)

\* @param L 线性表L

\* @param i 数据元素的位序

\* @param e 用于存放数据元素的值

\* @return 操作结果状态Status OK/ERROR

\*/

Status GetElem (Sqlist L, int i, ElemType \* e);

/\*\*

\* 查询满足比较规则的线性表数据元素的位序

\* 初始条件：线性表已存在

\* @param L 线性表L

\* @param e 目标数据元素值

\* @param compare 自定义数据元素比较规则

\* @return L中第1个与e满足关系compare()关系的数据元素的位序(不存在为0)

\*/

int LocateElem (Sqlist L, ElemType e, Status (\*cmp) (ElemType Elem, ElemType AnotherElem));

/\*\*

\* 定义L中元素的比较规则

\* @param Elem L中某个元素

\* @param AnotherElem L中另一元素

\* @return 比较结果 TRUE/FALSE

\*/

Status compare (ElemType Elem, ElemType AnotherElem);

/\*\*

\* 查询目标元素的前驱元素的值

\* 初始条件:线性表已存在

\* @param L 线性表L

\* @param cur\_e 目标数据元素值

\* @param pre\_e 用于存放前驱元素的值

\* @return 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

\*/

Status PriorElem (Sqlist L, ElemType cur\_e, ElemType \* pre\_e);

/\*\*

\* 查询目标元素的后继元素的值

\* 初始条件:线性表已存在

\* 调用函数:LocateElem

\* @param L 线性表L

\* @param cur\_e 目标数据元素值

\* @param next\_e 用于存放后继元素的值

\* @return 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

\*/

Status NextElem (Sqlist L, ElemType cur\_e, ElemType \* next\_e);

/\*\*

\* 在L的第i个位置之前插入新的数据元素e，L的长度加1

\* 初始条件:线性表L已存在且非空，1 ≤ i ≤ ListLength(L)+1

\* 调用函数:LocateElem

\* @param L 线性表地址值

\* @param i 目标插入位序

\* @param e 插入数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ListInsert (Sqlist \* L, int i, ElemType e);

/\*\*

\* 删除L的第i个数据元素，用e返回其值，L的长度减1

\* 初始条件：线性表L已存在，1 ≤ i ≤ ListLength(L)

\* @param L 线性表地址值

\* @param i 目标删除位序

\* @param e 用于保留删除数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ListDelete (Sqlist \* L, int i, ElemType \* e);

/\*\*

\* 遍历线性表L,并对每一元素都执行visit操作

\* 初始条件:线性表L已存在

\* @param L 线性表L

\* @param visit 对线性表L中每一元素所作操作

\* @return 操作结果状态Status OK/ERROR

\*/

Status ListTraverse (Sqlist L, Status (\*visitor) (ElemType Elem));

/\*\*

\* 打印数据元素Elem的值

\* @param Elem 线性表L中的某一元素

\* @return 操作结果状态Status OK/ERROR

\*/

Status visit (ElemType Elem);

/\*\*

\* 从output.dat读取二进制数据

\* @param L 待读取数据的线性表

\* @return 操作结果状态Status OK/ERROR

\*/

Status LoadData (Sqlist \*L);

/\*\*

\* 二进制输出数据元素Elem的值至output.dat

\* @param L 当前线性表

\* @return 操作结果状态Status OK/ERROR

\*/

Status SavaData (Sqlist L);

#endif /\*SQLIST\_H\_INCLUDED\*/

3.SqList.c：基本操作实现

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验一：顺序存储线性表基本操作实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "SqList.h"

Status InitList (Sqlist \* L)

{

//构造一个空的线性表L

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

if (! (\*L).elem)

{

exit (OVERFLOW); //存储分配失败

}

(\*L).length = 0; //空表长度为0

(\*L).listsize = LIST\_INIT\_SIZE; //初始存储容量

return OK;

}

Status DestroyList (Sqlist \* L)

{

//线性表不存在,函数执行失败

if (! (\*L).elem)

{

return ERROR;

}

//线性表存在,函数执行正确

free ( (\*L).elem);

(\*L).elem = NULL;

return OK;

}

Status ClearList (Sqlist \* L)

{

//线性表不存在,函数执行失败

if (! (\*L).elem)

{

return ERROR;

}

//线性表存在,函数执行正确

(\*L).length = 0;

return OK;

}

Status ListEmpty (Sqlist L)

{

//线性表不存在,函数执行失败

if (! L.elem)

{

return ERROR;

}

//线性表存在,函数执行正确

return L.length == 0 ? TRUE : FALSE;

}

int ListLength (Sqlist L)

{

//线性表不存在,函数执行失败

if (! L.elem)

{

return ERROR;

}

//线性表存在,函数执行正确

return L.length;

}

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

{

//线性表不存在,函数执行失败

if (! L.elem)

{

return ERROR;

}

//i超出范围,函数执行失败

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

{

return ERROR;

}

//函数执行正确

(\*e) = L.elem[i - 1]; //将目标元素值存入e指向内存单元

return OK;

}

int LocateElem (Sqlist L, ElemType e,

Status (\*cmp) (ElemType Elem, ElemType AnotherElem))

{

int i; //循环变量

//线性表不存在,函数执行失败

if (! L.elem)

{

return ERROR;

}

//初始化i为1

//遍历线性表,直至表尾或找到目标元素

for (i = 1; i <= ListLength (L) && cmp (e, L.elem[i - 1]) == FALSE; i++);

//当i>L.length时，说明表中无目标元素，返回0

//当i<L.length时，说明成功找到目标元素，返回其位序

return (i <= ListLength (L) ? i : 0);

}

Status compare (ElemType Elem, ElemType AnotherElem)

{

return Elem == AnotherElem ? TRUE : FALSE;

}

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

{

int cur\_pos = 0; //当前数据元素位序

if (! L.elem)

{

return ERROR; //线性表不存在,函数执行失败

}

//为调用LocateElem函数，定义函数指针compare

Status (\*cmp) (ElemType Elem, ElemType AnotherElem) = compare;

//利用LocateElem函数定位数据元素cur\_e;

cur\_pos = LocateElem (L, cur\_e, cmp);

//目标元素不存在或为首元素，执行失败

//否则将前驱元素值存入e所指内存单元

if (cur\_pos == 0 || cur\_pos == 1)

{

return ERROR;

}

(\*pre\_e) = L.elem[cur\_pos - 2];

return OK;

}

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

{

int cur\_pos = 0; //当前数据元素位序

if (! L.elem)

{

return ERROR; //线性表不存在,函数执行失败

}

//为调用LocateElem函数，定义函数指针compare

Status (\*cmp) (ElemType Elem, ElemType AnotherElem) = compare;

//利用LocateElem函数定位数据元素cur\_e;

cur\_pos = LocateElem (L, cur\_e, cmp);

//目标元素不存在或为尾元素，执行失败

//否则将前驱元素值存入e所指内存单元

if (cur\_pos == 0 || cur\_pos == L.length)

{

return ERROR;

}

(\*next\_e) = L.elem[cur\_pos];

return OK;

}

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

{

int j; //循环变量

//为扩容而分配的新基址newBase

ElemType \* newBase;

//线性表不存在,函数执行失败

if (! (\*L).elem)

{

return ERROR;

}

//i超出范围,函数执行失败

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

{

return ERROR;

}

if ( (\*L).length >= (\*L).listsize) //当前存储空间已满，增加存储总容量

{

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

if (! newBase)

{

exit (OVERFLOW); //存储分配失败

}

(\*L).elem = newBase; //新基址

(\*L).listsize += LISTINCREMENT; //增加存储容量

}

//插入位置及之后的元素右移

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

{

(\*L).elem[j] = (\*L).elem[j - 1];

}

//插入e

(\*L).elem[i - 1] = e;

//表长增1

(\*L).length++;

return OK;

}

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

{

int j;

//线性表不存在,函数执行失败

if (! (\*L).elem)

{

return ERROR;

}

//i超出范围,函数执行失败

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

{

return ERROR;

}

//被删除元素的值

(\*e) = (\*L).elem[i - 1];

//删除位置之后的元素左移

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

{

(\*L).elem[j - 1] = (\*L).elem[j];

}

//表长减1

(\*L).length--;

return OK;

}

Status ListTraverse (Sqlist L, Status (\*visitor) (ElemType Elem))

{

int i;

//线性表不存在,函数执行失败

if (! L.elem)

{

return ERROR;

}

//遍历线性表

for (i = 1; i <= ListLength (L); i++)

{

visitor (L.elem[i - 1]);

}

return OK;

}

Status visit (ElemType Elem)

{

printf ("%d\n", Elem);

return OK;

}

Status LoadData (Sqlist \*L)

{

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","r")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//读取二进制数据

while( fread(&(L->elem[L->length]), sizeof(ElemType), 1, fp) )

{

L->length++;

}

fclose(fp);

return OK;

}

Status SavaData (Sqlist L)

{

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","w")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//保存二进制数据

fwrite(L.elem, sizeof(ElemType), L.length, fp);

fclose(fp);

return OK;

}

4.assert\_test.c：断言测试

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验一：顺序存储线性表断言测试

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 测试原则 :

\* - 动态操作后检查线性表状态和长度

\* - 静态操作后不检查线性表状态和长度

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <assert.h>

#include "SqList.h"

#undef NDEBUG //开启断言测试

#define TEST\_LENGTH 10 //定义线性表测试长度为10

int main (void)

{

//循环变量

int i;

//断言测试用变量

Sqlist test\_sqlist;

test\_sqlist.elem = NULL;

ElemType test\_elem;

Status (\*test\_compare) (ElemType, ElemType) = compare;

Status (\*test\_visit) (ElemType) = visit;

//遍历线性表(失败)

assert (ListTraverse (test\_sqlist, test\_visit) == ERROR);

//初始化线性表

assert (InitList (&test\_sqlist) == OK);

//遍历线性表

assert (ListTraverse (test\_sqlist, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_sqlist) == 0);

assert (ListEmpty (test\_sqlist) == TRUE);

//销毁线性表

assert (DestroyList (&test\_sqlist) == OK);

//再次初始化线性表

assert (InitList (&test\_sqlist) == OK);

//遍历线性表

assert (ListTraverse (test\_sqlist, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_sqlist) == 0);

assert (ListEmpty (test\_sqlist) == TRUE);

//插入新元素

for ( i = 1; i <= TEST\_LENGTH; i++)

{

assert (ListInsert (&test\_sqlist, i, i) == OK);

printf ("InsertElement : %d\n", i);

}

//遍历线性表

assert (ListTraverse (test\_sqlist, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_sqlist) == TEST\_LENGTH);

assert (ListEmpty (test\_sqlist) == FALSE);

//删除前5个元素

for ( i = 1; i <= TEST\_LENGTH / 2; i++)

{

assert (ListDelete (&test\_sqlist, 1, &test\_elem) == OK);

printf ("DeleteElement : %d\n", test\_elem);

}

//遍历线性表

assert (ListTraverse (test\_sqlist, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_sqlist) == TEST\_LENGTH / 2);

assert (ListEmpty (test\_sqlist) == FALSE);

//查找元素测试

assert (GetElem (test\_sqlist, 0, &test\_elem) == ERROR);

for ( i = 1; i <= ListLength (test\_sqlist); i++)

{

assert (GetElem (test\_sqlist, i, &test\_elem) == OK);

printf ("GetElement : %d\n", test\_elem);

}

//定位元素测试

assert (LocateElem (test\_sqlist, 0, test\_compare) == 0);

for ( i = 1; i <= ListLength (test\_sqlist); i++)

{

assert (LocateElem (test\_sqlist, i + 5, test\_compare) == i);

printf ("LocateElement : %d\n", i);

}

//前驱元素测试

assert (PriorElem (test\_sqlist, 0, &test\_elem) == ERROR);

assert (PriorElem (test\_sqlist, 1, &test\_elem) == ERROR);

for ( i = 1; i < ListLength (test\_sqlist); i++)

{

assert (PriorElem (test\_sqlist, i + 6, &test\_elem) == OK);

printf ("PriorElement : %d\n", test\_elem);

}

//后继元素测试

assert (NextElem (test\_sqlist, 0, &test\_elem) == ERROR);

assert (NextElem (test\_sqlist, ListLength (test\_sqlist), &test\_elem) == ERROR);

for ( i = 1; i < ListLength (test\_sqlist); i++)

{

assert (NextElem (test\_sqlist, i + 5, &test\_elem) == OK);

printf ("NextElement : %d\n", test\_elem);

}

//重置线性表

assert (ClearList (&test\_sqlist) == OK);

//遍历线性表

assert (ListTraverse (test\_sqlist, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_sqlist) == 0);

assert (ListEmpty (test\_sqlist) == TRUE);

system("pause");

}

5.main.c：系统主程序

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验一：顺序存储线性表主程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "SqList.h"

int main (void)

{

Sqlist sqlist; //线性表结构

sqlist.elem = NULL;

ElemType input; //用户输入变量

ElemType container;//用于存放各个功能函数反馈的数据元素值

Status (\*test\_compare) (ElemType, ElemType) = compare;

Status (\*test\_visit) (ElemType) = visit;

//初始化线性表

InitList (&sqlist);

//读取数据

if (LoadData(&sqlist) == ERROR )

{

exit(ERROR);

}

//打印系统菜单界面

printf ("\t\t欢迎使用顺序线性表\n\n");

while (TRUE)

{

//打印系统菜单界面

printf ("1:重置线性表\n");

printf ("2:查询是否为空表\n");

printf ("3:查询表长\n");

printf ("4:查询元素\n");

printf ("5:定位元素\n");

printf ("6:查询前驱元素\n");

printf ("7:查询后继元素\n");

printf ("8:插入元素\n");

printf ("9:删除元素\n");

printf ("10:遍历线性表\n");

printf ("11:退出系统\n\n");

//提示用户选择功能

printf ("请输入所选功能选项:");

//用户选择事件

scanf ("%d", &input);

//处理用户选择事件

switch (input)

{

case 1:

ClearList (&sqlist);

printf ("已重置\n");

break;

case 2:

if (ListEmpty (sqlist) == TRUE)

{

printf ("当前表为空表\n");

}

else

{

printf ("当前表不为空表\n");

}

break;

case 3:

container = ListLength (sqlist);

printf ("当前表长为：%d\n", container);

break;

case 4:

printf ("请输入目标元素位序:");

scanf ("%d", &input);

if (GetElem (sqlist, input, &container) == ERROR)

{

printf ("查询失败\n");

}

else

{

printf ("目标元素值为:%d\n", container);

}

break;

case 5:

printf ("请输入目标元素值:");

scanf ("%d", &input);

if ( (container = LocateElem (sqlist, input, test\_compare)) == 0)

{

printf ("定位失败，不存在此元素\n");

}

else

{

printf ("目标元素位序为:%d\n", container);

}

break;

case 6:

printf ("请输入目标元素值:");

scanf ("%d", &input);

if (PriorElem (sqlist, input, &container) == ERROR)

{

printf ("查询失败，此元素不存在前驱元素\n");

}

else

{

printf ("目标前驱元素为:%d\n", container);

}

break;

case 7:

printf ("请输入目标元素值:");

scanf ("%d", &input);

if (NextElem (sqlist, input, &container) == ERROR)

{

printf ("查询失败，此元素不存在后继元素\n");

}

else

{

printf ("目标后继元素为:%d\n", container);

}

break;

case 8:

printf ("请输入插入元素位序:");

scanf ("%d", &input);

printf ("请输入插入元素值:");

scanf ("%d", &container);

if (ListInsert (&sqlist, input, container) == ERROR)

{

printf ("插入失败\n");

}

else

{

printf ("插入 %d 成功\n", container);

}

break;

case 9:

printf ("请输入删除元素位序:");

scanf ("%d", &input);

if (ListDelete (&sqlist, input, &container) == ERROR)

{

printf ("删除失败\n");

}

else

{

printf ("删除 %d 成功\n", container);

}

break;

case 10:

ListTraverse (sqlist, test\_visit);

break;

case 11:

//保存数据

if (SavaData(sqlist) == ERROR)

{

exit(ERROR);

}

//销毁线性表

DestroyList (&sqlist);

exit (OK);

break;

default:

printf ("所选功能不存在,请重新选择功能");

break;

}

printf ("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

}

6.Makefile：编译链接

DT\_LAB\_01.exe : SqList.o main.o

gcc –Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -o DT\_LAB\_01.exe SqList.o main.o

SqList.o : SqList.c SqList.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -c SqList.c

main.o : main.c SqList.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -c main.c

1.3.2 系统测试

本次测试采用断言测试，测试主程序及源代码见附件AssertTest.exe及程序清单assert\_test.c。

本次断言测试采用两种方式，一为普通断言，二为循环断言。普通断言即为单独调用一次目标函数，循环断言即为循环调用目标函数多次，并同时进行断言测试。普通断言测试函数：InitList、DestroyList、ClearList、ListLength、ListEmpty、ListTraverse；循环断言测试函数：ListInsert、ListDelete、GetElem、LocateElem、PriorElem、NextElem。

每次调用静态函数后不对ListTraverse、ListLength、ListEmpty进行断言测试，每次调用动态函数后都对ListTraverse、ListLength、ListEmpty进行断言测试。

测试流程如下：

ListTraverse == ERROR

InitList == OK

ListTraverse == OK

ListLength == 0

ListEmpty == TRUE

DestroyList == OK

InitList == OK

ListTraverse == OK

ListLength == 0

ListEmpty == TRUE

for (TEST\_LENGTH times) {

ListInsert == OK

}

ListTraverse == OK

ListLength == TEST\_LENGTH

ListEmpty == FALSE

for (TEST\_LENGTH / 2 times) {

ListDelete == OK

}

ListTraverse == OK

ListLength == TEST\_LENGTH / 2

ListEmpty == FALSE

for (ListLength times) {

GetElem == OK

}

for (ListLength times) {

LocateElem == i

}

PriorElem (0) == ERROR

PriorElem (1) == ERROR

for (ListLength times) {

PriorElem == OK

}

NextElem (0) == ERROR

NextElem (ListLength (test\_sqlist)) == ERROR

for (ListLength times) {

NextElem == OK

}

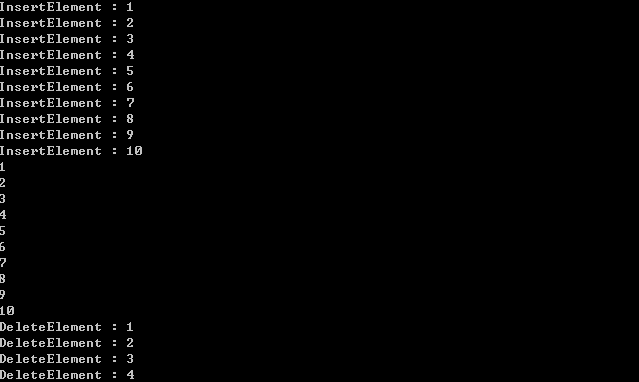
ClearList (&test\_sqlist) == OK

ListTraverse == OK

ListLength == 0

ListEmpty == TRUE

测试结果如图1-2所示，所有断言成功，程序正常退出，结果符合预期。



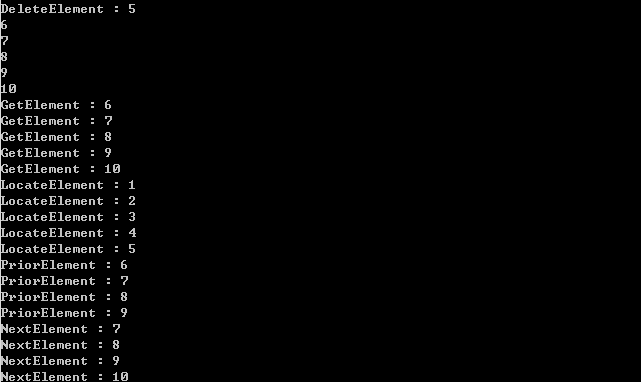


图1-2 断言测试结果图

1.4 实验小结

一、借助断言测试，极大地缩短了Debug时长与难度，也省去了人工输入测试用例的麻烦。在进行完断言测试并将相关函数完成后，再进行main系统主程序的编写，编写主程序的过程异常顺利。同时，断言测试也缩短了编写系统主程序的时长与难度。通过此次实验的断言测试实践，深刻体会到了C标准库撰写人员的伟大智慧；

二、借助直接使用命令行调用gcc编译工具与gdb调试工具。通过不借助IDE的自动化流程成功编译出可执行文件，加深了对C程序编译、链接具体过程的理解；通过不借助IDE的可视化调试器而使用字符界面调试工具，加强了自身Debug能力；

三、通过实现顺序存储线性表的基本操作，对线性表这一简单的数据结构有了良好的掌握，希望能为以后的数据结构学习打下良好基础。

**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  (40分) | 程序规范  (10分) | 报告内容  (40分) | 报告规范  (10分) | 逾 期  扣 分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |

# 2基于链式存储结构实现线性表的基本运算

2.1 问题描述

一、实验任务

采用带表头结点的单链表作为线性表的物理结构，实现2.2描述的基本操作。整体系统采用顺序存储多个首结点，链式存储不同单链表的方式，实现对多个单链表实现操作，链式存储多线性表物理结构如图2-1所示。

图2-1 多线性表的物理结构示意图

二、实验目的

（1）加深对线性表的概念、基本运算的理解；

（2）熟练掌握线性表的逻辑结构与物理结构的关系；

（3）物理结构采用带表头结点的单链表,熟练掌握线性表基本运算的实现。

三、实现算法

（1）线性表的初始化、销毁、重置；

（2）线性表的空表查询、长度查询、元素值查询、位序值查询；

（3）线性表的前驱/后继元素查询、插入、删除、遍历；

（4）线性表的切换，根据ID号切换操作不同链表。

2.2 单链表演示系统设计

2.2.1 系统总体设计

本系统为链式存储多线性表使用系统。本系统可自动初始化线性表与自动销毁线性表。本系统可自动将线性表保存在文件output.txt中。本系统具有以下几大功能：

一、切换功能

允许用户通过键入目标链表的ID号，切换至不同链表，以实现对多链表的操作。当前系统允许操作的链表数为100，ID号为1-100。

1. 查询功能

查询空表状态、查询表长、按元素值查询位序、按位序查询元素值、

查询前驱元素值、查询后继元素值

1. 修改功能

插入新元素、删除旧元素

1. 打印功能

遍历输出完整线性表

2.2.2 有关常量和类型定义

一、所有实验公有部分：见程序清单DataStructure.h

二、本实验常量和类型定义

1.可操作线性表的个数（当前系统可操作链表数为100）

#define LIST\_INIT\_SIZE 100

2．定义链式存储的线性表数据结构

typedef struct LNode

{

ElemType data;

struct LNode \* next;

} LNode, \*LinkList;

2.2.3 算法设计

一、InitaList(&L) {

//所有链表的首结点均在main函数中

//分配在一片定长连续内存单元

//故在此函数中无需为链表首结点另行分配内存

L->next = NULL; //构造一个空表

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

二、DestroyList(&L) {

//遍历链表，释放已分配给链表每个结点的内存单元

for (current = prior->next;prior->next != NULL;

prior = current, current = current->next)

{

free (prior);

}

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

三、ClearList (&L) {

//遍历链表，释放已分配给链表每个结点的内存单元

for (current = prior->next;prior->next != NULL;

prior = current, current = current->next)

{

free (prior);

}

//将链表置为空

L->next = NULL

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

四、ListEmpty(L) {

//表头结点的后继结点为NULL时说明为空表

return L->next == NULL ? TRUE : FALSE;

}

时间及空间复杂度：T(n) = O(1) , S(n) = O(1)

五、ListLength(L) {

//遍历链表,查找链表长度

for (ptr = L, length = 0; ptr->next != NULL; ptr = ptr->next, length++)

{

;

}

return L.length; //返回表长

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

六、GetElem(L, i,&e) {

//遍历链表,查找目标结点

for (ptr = L, j = 0; j < i; ptr = ptr->next, j++)

{

;

}

//利用e保存目标元素

e = ptr->data;

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

七、LocateElem(L, e, compare()) {

//遍历线性表,直至表尾或找到目标元素

for (ptr = L->next, i = 1; i <= ListLength (L) && cmp (e, ptr) == FALSE;

ptr = ptr->next, i++)

{

;

}

//i不大于表长时说明查找成功，返回位序；否则说明查找失败返回0

return (i <= ListLength (L) ? i : 0);

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

八、PriorElem(L, cur\_e, &pre\_e) {

cur\_pos = LocateElem (L, cur\_e, cmp); //定位当前元素位序

//遍历线性表,直至找到目标前驱结点

for (ptr = L, i = 0; i < cur\_pos - 1; ptr = ptr->next, i++);

//利用pre\_e保存目标元素

(\*pre\_e) = ptr->data;

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

九、NextElem(L, cur\_e, &next\_e) {

cur\_pos = LocateElem (L, cur\_e, cmp); //定位当前元素位序

//遍历线性表,直至找到目标后继结点

for (ptr = L, i = 0; i < cur\_pos + 1; ptr = ptr->next, i++);

//利用next\_e保存目标元素

next\_e = ptr->data;

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

十、ListInsert(&L,i,e) {

//寻找待插入位置前驱结点

for (ptr = L, j = 0; j < i - 1; ptr = ptr->next, j++)

{

;

}

//改变指针指向，插入新结点

insNode->next = ptr->next;

ptr->next = insNode;

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

十一、ListDelete(&L,i,&e) {

//寻找待删除结点的前驱结点

for (ptr = L, j = 0; j < i - 1; ptr = ptr->next, j++)

{

;

}

//改变指针指向，删除目标结点

delNode = ptr->next;

ptr->next = delNode->next;

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

十二、ListTraverse(L, visit()) {

//遍历线性表

for (ptr = L->next; ptr != NULL ; ptr = ptr->next)

{

visitor (ptr);

}

}

时间及空间复杂度：T(n) = O(n) , S(n) = O(1)

2.3 单链表演示系统实现与测试

2.3.1 系统实现

一、编程环境

编辑器：Sublime Text 3

编译器：MinGW-gcc 4.9.1(C99标准)

调试器：MinGW-gdb 7.8

二、文件调用关系



三、函数调用关系



四、程序清单

1.DataStructure.h：公有常量、类型定义

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验二：链式存储线性表

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef DATASTRUCTURE\_H\_INCLUDED

#define DATASTRUCTURE\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

//自定义宏

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

//Status是函数的类型,其值是函数结果状态代码

typedef int Status;

//将ElemType定义为int型

typedef int ElemType;

#endif /\*DATASTRUCTURE\_H\_INCLUDED\*/

2.LinkList.h：链式表抽象数据定义

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验二：链式存储线性表抽象数据类型定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef LINKLIST\_H\_INCLUDED

#define LINKLIST\_H\_INCLUDED

#include "DataStructure.h"

#define LIST\_INIT\_SIZE 100

//定义链式存储的线性表数据结构

typedef struct LNode

{

ElemType data;

struct LNode \* next;

} LNode, \*LinkList;

/\*\*

\* 构造一个空的线性表

\* @param L 线性表地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitList (LNode \* L);

/\*\*

\* 销毁线性表L

\* 初始条件:线性表L已存在

\* @param L 线性表地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DestroyList (LNode \* L);

/\*\*

\* 将L重置为空表

\* 初始条件:线性表已存在

\* @param L 线性表地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ClearList (LNode \* L);

/\*\*

\* 查询L是否为空表

\* 初始条件:线性表L已存在

\* @param L 线性表L

\* @return 若L为空表,则返回TRUE,否则返回FALSE

\*/

Status ListEmpty (LNode \* L);

/\*\*

\* 查询L的当前长度

\* 初始条件:线性表L已存在

\* @param L 线性表L

\* @return L中数据元素的个数

\*/

int ListLength (LNode \* L);

/\*\*

\* 查询L中第i个数据元素的值

\* 初始条件：线性表已存在，1 ≤ i ≤ ListLength(L)

\* @param L 线性表L

\* @param i 数据元素的位序

\* @param e 用于存放数据元素的值

\* @return 操作结果状态Status OK/ERROR

\*/

Status GetElem (LNode \* L, int i, ElemType \* e);

/\*\*

\* 查询满足比较规则的线性表数据元素的位序

\* 初始条件：线性表已存在

\* @param L 线性表L

\* @param e 目标数据元素值

\* @param compare 自定义数据元素比较规则

\* @return L中第1个与e满足关系compare()关系的数据元素的位序(不存在为0)

\*/

int LocateElem (LNode \* L, ElemType e, Status (\*cmp) (ElemType Elem, LNode \* AnotherElem));

/\*\*

\* 定义L中元素的比较规则

\* @param Elem L中某个元素

\* @param AnotherElem L中另一元素

\* @return 比较结果 TRUE/FALSE

\*/

Status compare (ElemType Elem, LNode \* AnotherElem);

/\*\*

\* 查询目标元素的前驱元素的值

\* 初始条件:线性表已存在

\* @param L 线性表L

\* @param cur\_e 目标数据元素值

\* @param pre\_e 用于存放前驱元素的值

\* @return 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

\*/

Status PriorElem (LNode \* L, ElemType cur\_e, ElemType \* pre\_e);

/\*\*

\* 查询目标元素的后继元素的值

\* 初始条件:线性表已存在

\* 调用函数:LocateElem

\* @param L 线性表L

\* @param cur\_e 目标数据元素值

\* @param next\_e 用于存放后继元素的值

\* @return 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

\*/

Status NextElem (LNode \* L, ElemType cur\_e, ElemType \* next\_e);

/\*\*

\* 在L的第i个位置之前插入新的数据元素e，L的长度加1

\* 初始条件:线性表L已存在且非空，1 ≤ i ≤ ListLength(L)+1

\* 调用函数:LocateElem

\* @param L 线性表地址值

\* @param i 目标插入位序

\* @param e 插入数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ListInsert (LNode \* L, int i, ElemType e);

/\*\*

\* 删除L的第i个数据元素，用e返回其值，L的长度减1

\* 初始条件：线性表L已存在，1 ≤ i ≤ ListLength(L)

\* @param L 线性表地址值

\* @param i 目标删除位序

\* @param e 用于保留删除数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ListDelete (LNode \* L, int i, ElemType \* e);

/\*\*

\* 遍历线性表L,并对每一元素都执行visit操作

\* 初始条件:线性表L已存在

\* @param L 线性表L

\* @param visit 对线性表L中每一元素所作操作

\* @return 操作结果状态Status OK/ERROR

\*/

Status ListTraverse (LNode \* L, Status (\*visitor) (LNode \* Elem));

/\*\*

\* 打印数据元素Elem的值

\* @param Elem 线性表L中的某一元素

\* @return 操作结果状态Status OK/ERROR

\*/

Status visit (LNode \* Elem);

#endif /\*LINKLIST\_H\_INCLUDED\*/

3.LinkList.c：链表基本操作实现

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验二：链式存储线性表基本操作实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "LinkList.h"

Status InitList (LNode \* L)

{

//构造一个空的线性表L

L->next = NULL;

return OK;

}

Status DestroyList (LNode \* L)

{

//遍历指针

LNode \* prior, \*current;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

prior = L->next;

//空表

if (!prior)

{

current = NULL;

}

//遍历释放各个结点所占内存单元

else

{

for (current = prior->next;prior->next != NULL;

prior = current, current = current->next)

{

free (prior);

}

}

return OK;

}

Status ClearList (LNode \* L)

{

//遍历指针

LNode \* prior, \* current;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

prior = L->next;

//空表

if (!prior)

{

current = NULL;

}

//遍历释放各个结点所占内存单元

else

{

for (current = prior->next; prior->next != NULL;

prior = current, current = current->next)

{

free (prior);

}

}

//线性表存在,函数执行正确

L->next = NULL;

return OK;

}

Status ListEmpty (LNode \* L)

{

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

//线性表存在,函数执行正确

return L->next == NULL;

}

int ListLength (LNode \* L)

{

//链表长度

int length;

//链表遍历指针

LNode \* ptr;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

//线性表存在,函数执行正确

//遍历链表,查找链表长度

for (ptr = L, length = 0; ptr->next != NULL; ptr = ptr->next, length++)

{

;

}

return length;

}

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

{

//循环变量

int j;

//链表遍历指针

LNode \* ptr;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

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

{

return ERROR;

}

//函数执行正确

//遍历链表,查找目标元素

for (ptr = L, j = 0; j < i; ptr = ptr->next, j++)

{

;

}

(\*e) = ptr->data;//将目标元素值存入e指向内存单元

return OK;

}

int LocateElem (LNode \* L, ElemType e,

Status (\*cmp) (ElemType Elem, LNode \* AnotherElem))

{

//循环变量

int i;

//链表遍历指针

LNode \* ptr;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

//初始化i为1

//遍历线性表,直至表尾或找到目标元素

for (ptr = L->next, i = 1; i <= ListLength (L) && cmp (e, ptr) == FALSE; ptr = ptr->next, i++)

{

;

}

//当i>L.length时，说明表中无目标元素，返回0

//当i<L.length时，说明成功找到目标元素，返回其位序

return (i <= ListLength (L) ? i : 0);

}

Status compare (ElemType Elem, LNode \* AnotherElem)

{

return Elem == AnotherElem->data ? TRUE : FALSE;

}

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

{

{

//循环变量

int i;

//链表遍历指针

LNode \* ptr;

int cur\_pos = 0; //当前数据元素位序

if (! L)

{

return ERROR; //线性表不存在,函数执行失败

}

//为调用LocateElem函数，定义函数指针compare

Status (\*cmp) (ElemType Elem, LNode \* AnotherElem) = compare;

//利用LocateElem函数定位数据元素cur\_e;

cur\_pos = LocateElem (L, cur\_e, cmp);

//目标元素不存在或为首元素，执行失败

//否则将前驱元素值存入e所指内存单元

if (cur\_pos == 0 || cur\_pos == 1)

{

return ERROR;

}

//遍历线性表,直至找到目标元素

for (ptr = L, i = 0; i < cur\_pos - 1; ptr = ptr->next, i++);

(\*pre\_e) = ptr->data;

return OK;

}

}

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

{

//循环变量

int i;

//链表遍历指针

LNode \* ptr;

int cur\_pos = 0;

//当前数据元素位序

if (! L)

{

return ERROR; //线性表不存在,函数执行失败

}

//为调用LocateElem函数，定义函数指针compare

Status (\*cmp) (ElemType Elem, LNode \* AnotherElem) = compare;

//利用LocateElem函数定位数据元素cur\_e;

cur\_pos = LocateElem (L, cur\_e, cmp);

//目标元素不存在或为尾元素，执行失败

//否则将前驱元素值存入e所指内存单元

if (cur\_pos == 0 || cur\_pos == ListLength (L))

{

return ERROR;

}

//遍历线性表,直至找到目标元素

for (ptr = L, i = 0; i < cur\_pos + 1; ptr = ptr->next, i++);

(\*next\_e) = ptr->data;

return OK;

}

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

{

int j; //循环变量

//链表遍历指针

LNode \* ptr;

//插入结点

LNode \* insNode;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

//i超出范围,函数执行失败

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

{

return ERROR;

}

//构造插入结点

insNode = (LNode \*) malloc (sizeof (LNode));

insNode->data = e;

//插入操作

//寻找前驱结点

for (ptr = L, j = 0; j < i - 1; ptr = ptr->next, j++)

{

;

}

//插入新结点

insNode->next = ptr->next;

ptr->next = insNode;

return OK;

}

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

{

int j; //循环变量

//链表遍历指针

LNode \* ptr;

//保存被删除结点

LNode \* delNode;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

//i超出范围,函数执行失败

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

{

return ERROR;

}

//删除操作

//寻找前驱结点

for (ptr = L, j = 0; j < i - 1; ptr = ptr->next, j++)

{

;

}

//保存删除结点值

(\*e) = ptr->next->data;

//删除目标结点

delNode = ptr->next;

ptr->next = delNode->next;

free (delNode);

return OK;

}

Status ListTraverse (LNode \* L, Status (\*visitor) (LNode \* Elem))

{

//遍历指针

LNode \* ptr;

//线性表不存在,函数执行失败

if (! L)

{

return ERROR;

}

//空表

if (!L->next)

{

return ERROR;

}

//遍历线性表

for (ptr = L->next; ptr != NULL ; ptr = ptr->next)

{

visitor (ptr);

}

return OK;

}

Status visit (LNode \* Elem)

{

printf ("%d\n", Elem->data);

return OK;

}

4.assert\_test.c：断言测试

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验二：链式存储线性表断言测试

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 测试原则 :

\* - 动态操作后检查线性表状态和长度

\* - 静态操作后不检查线性表状态和长度

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <assert.h>

#include "LinkList.h"

#undef NDEBUG //开启断言测试

#define TEST\_LENGTH 10 //定义线性表测试长度为10

int main (void)

{

//断言测试用变量

LNode \* test\_list = (LNode \*)malloc(sizeof(LNode));

ElemType test\_elem;

Status (\*test\_compare) (ElemType, LNode \*) = compare;

Status (\*test\_visit) (LNode \*) = visit;

//初始化线性表

assert (InitList(test\_list) == OK);

//遍历线性表

assert (ListTraverse(test\_list, test\_visit) == OK);

//查询长度及空表

assert (ListLength(test\_list) == 0);

assert (ListEmpty(test\_list) == TRUE);

//销毁线性表

assert (DestroyList(test\_list) == OK);

//再次初始化线性表

test\_list = (LNode \*)malloc(sizeof(LNode));

assert (InitList(test\_list) == OK);

//遍历线性表

assert (ListTraverse(test\_list, test\_visit) == OK);

//查询长度及空表

assert (ListLength(test\_list) == 0);

assert (ListEmpty(test\_list) == TRUE);

//插入新元素

for (int i = 1; i <= TEST\_LENGTH; i++)

{

assert (ListInsert (test\_list, i, i) == OK);

printf ("InsertElement : %d\n", i);

}

//遍历线性表

assert (ListTraverse (test\_list, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_list) == TEST\_LENGTH);

assert (ListEmpty (test\_list) == FALSE);

//删除前5个元素

for (int i = 1; i <= TEST\_LENGTH / 2; i++)

{

assert (ListDelete (test\_list, 1, &test\_elem) == OK);

printf ("DeleteElement : %d\n", test\_elem);

}

//遍历线性表

assert (ListTraverse (test\_list, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_list) == TEST\_LENGTH / 2);

assert (ListEmpty (test\_list) == FALSE);

//查找元素测试

assert (GetElem (test\_list, 0, &test\_elem) == ERROR);

for (int i = 1; i <= ListLength (test\_list); i++)

{

assert (GetElem (test\_list, i, &test\_elem) == OK);

printf ("GetElement : %d\n", test\_elem);

}

//定位元素测试

assert (LocateElem (test\_list, 0, test\_compare) == 0);

for (int i = 1; i <= ListLength (test\_list); i++)

{

assert (LocateElem (test\_list, i + 5, test\_compare) == i);

printf ("LocateElement : %d\n", i);

}

//前驱元素测试

assert (PriorElem (test\_list, 0, &test\_elem) == ERROR);

assert (PriorElem (test\_list, 1, &test\_elem) == ERROR);

for (int i = 1; i < ListLength (test\_list); i++)

{

assert (PriorElem (test\_list, i + 6, &test\_elem) == OK);

printf ("PriorElement : %d\n", test\_elem);

}

//后继元素测试

assert (NextElem (test\_list, 0, &test\_elem) == ERROR);

assert (NextElem (test\_list, ListLength (test\_list), &test\_elem) == ERROR);

for (int i = 1; i < ListLength (test\_list); i++)

{

assert (NextElem (test\_list, i + 5, &test\_elem) == OK);

printf ("NextElement : %d\n", test\_elem);

}

//重置线性表

assert (ClearList(test\_list) == OK);

//遍历线性表

assert (ListTraverse (test\_list, test\_visit) == OK);

//查询长度及空表

assert (ListLength (test\_list) == 0);

assert (ListEmpty (test\_list) == TRUE);

system("pause");

}

5.main.c：系统主程序

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验二：链式存储线性表主程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "LinkList.h"

int main (void)

{

//为LIST\_INIT\_SIZE个链表首结点分配合适内存单元

LNode \* multi\_list = (LNode \*)malloc(LIST\_INIT\_SIZE \* sizeof(LNode));

if (! multi\_list) exit(OVERFLOW); //内存空间不足，退出系统

int list\_id = 1; //链表ID号,初始化为1，表示默认操作第一个链表

int max\_list\_id = LIST\_INIT\_SIZE; //最大ID号

ElemType input; //用户输入变量

ElemType container; //用于存放各个功能函数反馈的数据元素值

Status (\*test\_compare) (ElemType, LNode \*) = compare;

Status (\*test\_visit) (LNode \*) = visit;

//初始化所有链表

for (list\_id = 1;list\_id <= LIST\_INIT\_SIZE;list\_id++)

{

InitList(multi\_list + list\_id - 1);

}

//当前操作链表重置为第一个链表

list\_id = 1;

//打印系统菜单界面

printf ("\t\tWelcome to MultiList System\n\n");

while (TRUE)

{

//打印系统菜单界面

printf ("1:Clear List\n");

printf ("2:Require List Empty or Not\n");

printf ("3:Require List Length\n");

printf ("4:Require List Element by Inputing Position\n");

printf ("5:Require List Element by Inputing Value\n");

printf ("6:Require Prior Element\n");

printf ("7:Require Next Element\n");

printf ("8:Insert Element\n");

printf ("9:Delete Element\n");

printf ("10:Traverse List\n");

printf ("11:Change List to be Manipulated(ID:1-%d)\n", max\_list\_id);

printf ("12:Exit System\n\n");

//提示用户选择功能

printf ("Please Choose Function:");

//用户选择事件

scanf ("%d", &input);

//处理用户选择事件

switch (input)

{

case 1:

ClearList (multi\_list + list\_id - 1);

printf ("Clear Success\n");

break;

case 2:

if (ListEmpty (multi\_list + list\_id - 1) == TRUE)

{

printf ("No.%d List is Empty\n", list\_id);

}

else

{

printf ("No.%d List is not Empty\n", list\_id);

}

break;

case 3:

container = ListLength (multi\_list + list\_id - 1);

printf ("No.%d List Length:%d\n", list\_id, container);

break;

case 4:

printf ("Please input target position:");

scanf ("%d", &input);

if (GetElem (multi\_list + list\_id - 1, input, &container) == ERROR)

{

printf ("Require Failed\n");

}

else

{

printf ("No.%d Element Value:%d\n", input, container);

}

break;

case 5:

printf ("Please input target value:");

scanf ("%d", &input);

if ( (container = LocateElem (multi\_list + list\_id - 1, input, test\_compare)) == 0)

{

printf ("Locate Failed\n");

}

else

{

printf ("The position of %d :%d\n", input, container);

}

break;

case 6:

printf ("Please input target value:");

scanf ("%d", &input);

if (PriorElem (multi\_list + list\_id - 1, input, &container) == ERROR)

{

printf ("No Prior Element\n");

}

else

{

printf ("Prior Element:%d\n", container);

}

break;

case 7:

printf ("Please input target value:");

scanf ("%d", &input);

if (NextElem (multi\_list + list\_id - 1, input, &container) == ERROR)

{

printf ("No Next Element\n");

}

else

{

printf ("Next Element:%d\n", container);

}

break;

case 8:

printf ("Please input position to be inserted:");

scanf ("%d", &input);

printf ("Please input value to be inserted:");

scanf ("%d", &container);

if (ListInsert (multi\_list + list\_id - 1, input, container) == ERROR)

{

printf ("Insert Failed\n");

}

else

{

printf ("Insert %d Success\n", container);

}

break;

case 9:

printf ("Please input the position of element to be deleted:");

scanf ("%d", &input);

if (ListDelete (multi\_list + list\_id - 1, input, &container) == ERROR)

{

printf ("Delete Failed\n");

}

else

{

printf ("Delete %d Success\n", container);

}

break;

case 10:

if (ListTraverse (multi\_list + list\_id - 1, test\_visit) == ERROR)

{

printf("Traverse Failed\n");

}

break;

case 11:

//此循环用于限定用户输入

do

{

printf("Please input target ID:");

//用户键入目标链表ID

scanf("%d", &list\_id);

}

while(list\_id < 1 || list\_id > LIST\_INIT\_SIZE);

break;

case 12:

//将线性表存入文件

//将标准输出重定向至输出文件output

freopen("output.txt", "w", stdout);

//调用ListTraverse函数,将链表打印至output

for(list\_id = 1;list\_id <= LIST\_INIT\_SIZE;list\_id++)

{

ListTraverse(multi\_list + list\_id - 1, test\_visit);

}

//销毁多线性表

for(list\_id = 1;list\_id <= LIST\_INIT\_SIZE;list\_id++)

{

DestroyList(multi\_list + list\_id - 1);

}

//销毁所有首结点

free(multi\_list);

exit (OK);

break;

default:

printf ("No Such Function");

break;

}

printf ("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

}

6.Makefile：编译链接

DT\_LAB\_02.exe : LinkList.o main.o

gcc -Wall -std=c99 -g -o DT\_LAB\_02.exe LinkList.o main.o

LinkList.o : LinkList.c LinkList.h

gcc -Wall -std=c99 -g -c LinkList.c

main.o : main.c LinkList.h

gcc -Wall -std=c99 -g -c main.c

2.3.2 系统测试

本次测试采用断言测试，测试主程序及源代码见附件AssertTest.exe及程序清单assert\_test.c。

本次断言测试采用两种方式，一为普通断言，二为循环断言。普通断言即为单独调用一次目标函数，循环断言即为循环调用目标函数多次，并同时进行断言测试。普通断言测试函数：InitList、DestroyList、ClearList、ListLength、ListEmpty、ListTraverse；循环断言测试函数：ListInsert、ListDelete、GetElem、LocateElem、PriorElem、NextElem。

每次调用静态函数后不对ListTraverse、ListLength、ListEmpty进行断言测试，每次调用动态函数后都对ListTraverse、ListLength、ListEmpty进行断言测试。

测试流程如下：（测试流程与上次实验略有差异）

InitList == OK

ListTraverse == OK

ListLength == 0

ListEmpty == TRUE

DestroyList == OK

InitList == OK

ListTraverse == OK

ListLength == 0

ListEmpty == TRUE

for (TEST\_LENGTH times) {

ListInsert == OK

}

ListTraverse == OK

ListLength == TEST\_LENGTH

ListEmpty == FALSE

for (TEST\_LENGTH / 2 times) {

ListDelete == OK

}

ListTraverse == OK

ListLength == TEST\_LENGTH / 2

ListEmpty == FALSE

GetElem (0) == ERROR

for (ListLength times) {

GetElem == OK

}

LocateElem (0) == 0

for (ListLength times) {

LocateElem == i

}

PriorElem (0) == ERROR

PriorElem (1) == ERROR

for (ListLength times) {

PriorElem == OK

}

NextElem (0) == ERROR

NextElem (ListLength (test\_sqlist)) == ERROR

for (ListLength times) {

NextElem == OK

}

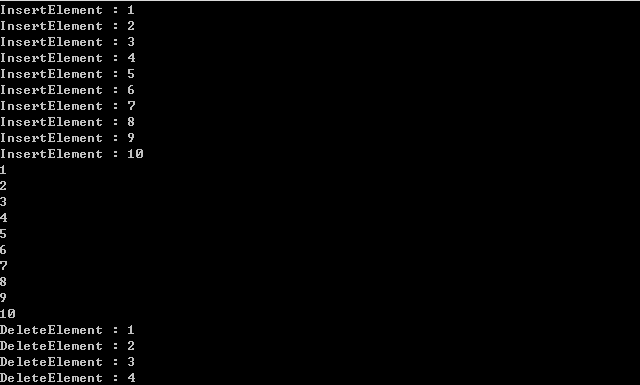
ClearList == OK

ListTraverse == OK

ListLength == 0

ListEmpty == TRUE

测试结果如图2-2所示，所有断言成功，程序正常退出，结果符合预期。



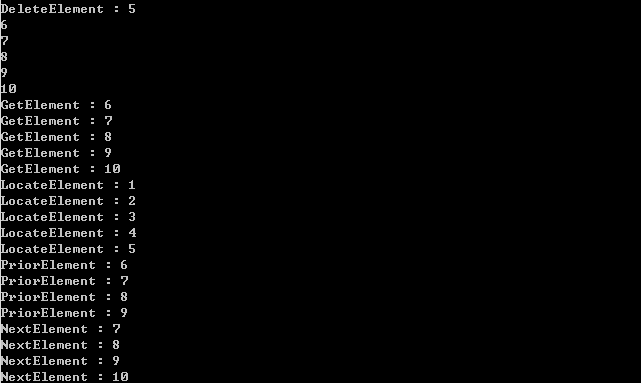


图2-2 断言测试结果图

2.4 实验小结

一、借助断言测试，极大地缩短了Debug时长与难度，也省去了人工输入测试用例的麻烦。在进行完断言测试并将相关函数完成后，再进行main系统主程序的编写，编写主程序的过程异常顺利。同时，断言测试也缩短了编写系统主程序的时长与难度。

二、继上次实验使用gcc、gdb这两个工具后，本次实验能够更好地运用这两个工具。由于此次实验继涉及顺序存储，有涉及到链式存储，所以在调试过程中经常遇到SIGSEGV错误信息，借助新学习了gdb更多的常用指令(l、b、s、n、p)后，将遇到的内存引用错误一一化解。在这个过程中，加深了对指针的理解，同时也加强了自身对gdb的使用。

三、通过实现链式存储线性表的基本操作，并结合顺序存储实现多链表系统，巩固了上次实验所掌握内容，并加强了对指针的理解和使用，希望能为以后的数据结构学习带来帮助。

四、本次实验遇到最大的挑战便是保存文件和读取文件模块。经过苦思冥想，终于想出了如何正确的保存和读取多链表。对于每个单链表的保存，除了保存它的结点数据域外，还保存了ID号与链表长度。通过这两个数据，便可在读取时精准定位并还原每个单链表的位置与长度。

**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  (40分) | 程序规范  (10分) | 报告内容  (40分) | 报告规范  (10分) | 逾 期  扣 分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |

# 3基于顺序存储结构实现栈的基本运算

3.1 顺序栈实验

3.1.1 问题描述

3.1.1.1 实验任务

采用顺序栈作为栈的物理结构，实现3.2所列栈的全部操作；

3.1.1.2 实验目的

（1）加深对栈的概念、基本运算的理解；

（2）熟练掌握栈的逻辑结构与物理结构的关系；

（3）熟练掌握顺序栈的基本运算的实现；

（4）通过栈的应用体会其用途。

3.1.1.3 实验算法

（1）顺序栈的初始化、销毁、重置；

（2）顺序栈的空表查询、长度查询、栈顶素值查询；

（3）顺序栈的压栈、退栈、遍历。

3.1.2 顺序栈设计

3.1.2.1 系统总体设计

本系统为顺序存储栈使用系统。本系统可自动初始化栈与自动销毁栈。本系统可自动将栈保存在文件output.txt中。本系统具有以下几大功能：

1.查询功能

查询空栈状态、查询栈长、查询栈顶元素

2.修改功能

压栈、退栈

3.打印功能

遍历输出完整栈

3.1.2.2 有关常量和类型定义

1.所有实验公有部分：见程序清单DataStructure.h

2.本实验常量和类型定义

//栈存储空间默认初始分配量

#define STACK\_INIT\_SIZE 100

//栈存储空间一次的分配增量

#define STACKINCREMENT 10

//定义顺序存储栈数据结构

typedef struct

{

ElemType \* base; //在栈构造之前和销毁之后，base的值为NULL

ElemType \* top; //栈顶指针

int stacksize; //当前分配存储总容量(以sizeof(ElemType)为单位)

} Stack;

3.1.2.3 算法设计

1.InitStack (Stack S)

//构造栈底

S->base = (ElemType \*)malloc(STACK\_INIT\_SIZE \* sizeof(ElemType));

//构造栈顶

S->top = S->base;

//初始化栈长

S->stacksize = 0;

T(n) = O(1) , S(n) = O(1)

2.DestroyStack (Stack S)

//销毁栈底指针所指内存单元

free(S->base);

//栈底指针重置为NULL,栈顶指针重置为栈底指针，栈长重置为0

S->base = NULL;

S->top = S->base;

S->stacksize = 0;

T(n) = O(1) , S(n) = O(1)

3.ClearStack (Stack S)

//栈顶指针重置为栈底指针，栈长重置为0

S->top = S->base;

S->stacksize = 0;

T(n) = O(1) , S(n) = O(1)

4.StackEmpty (Stack S)

return S.base == S.top ? TRUE : FALSE;

T(n) = O(1) , S(n) = O(1)

5.StackLength (Stack S)

return S.top - S.base;

T(n) = O(1) , S(n) = O(1)

6.GetTop (Stack S, ElemType e)

//利用e返回栈顶元素值

E = S.top - 1;

T(n) = O(1) , S(n) = O(1)

7.Push (Stack S, ElemType e)

(S->top)++ = e;

T(n) = O(1) , S(n) = O(1)

8.Pop (Stack S, ElemType e)

e = --S->top;

T(n) = O(1) , S(n) = O(1)

9.StackTraverse (Stack S, visitor)

//遍历输出栈中元素

ptr = S.base;

while (S.top != ptr)

{

Visit(ptr++);

}

T(n) = O(n) , S(n) = O(1)

10.LoadData (Stack S)

//从指定文件读取数据，并进行压栈操作

while(fread(S->top, sizeof(ElemType), 1, fp))

{

(S->top)++;

}

T(n) = O(n) , S(n) = O(1)

11.SaveData (Stack S)

//保存数据至指定文件

fwrite(S.base, sizeof(ElemType), StackLength(S), fp);

T(n) = O(1) , S(n) = O(1)

3.1.3 顺序栈实现与测试

3.1.3.1 实现部分

1.编程环境

编辑器：Sublime Text 3

编译器：MinGW-gcc 4.9.1(C99标准)

调试器：MinGW-gdb 7.8

2.文件调用关系



3.函数调用关系



4.程序清单

(1)DataStructure.h：共有类型定义

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验三：顺序存储栈

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef DATASTRUCTURE\_H\_INCLUDED

#define DATASTRUCTURE\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

//自定义宏

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

//Status是函数的类型,其值是函数结果状态代码

typedef int Status;

//将ElemType定义为char型

typedef char ElemType;

#endif /\*DATASTRUCTURE\_H\_INCLUDED\*/

(2)Stack.h：栈抽象数据定义

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验三：顺序存储栈抽象数据类型定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef STACK\_H\_INCLUDED

#define STACK\_H\_INCLUDED

#include "DataStructure.h"

//栈存储空间默认初始分配量

#define STACK\_INIT\_SIZE 100

//栈存储空间一次的分配增量

#define STACKINCREMENT 10

//定义顺序存储栈数据结构

typedef struct

{

ElemType \* base; //在栈构造之前和销毁之后，base的值为NULL

ElemType \* top; //栈顶指针

int stacksize; //当前分配存储总容量(以sizeof(ElemType)为单位)

} Stack;

/\*\*

\* 构造一个空栈S

\* @param S 栈地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitStack (Stack \* S);

/\*\*

\* 销毁栈S

\* 初始条件:栈S已存在

\* @param S 栈地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DestroyStack (Stack \* S);

/\*\*

\* 将S重置为空栈

\* 初始条件:栈已存在

\* @param S 栈地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ClearStack (Stack \* S);

/\*\*

\* 查询S是否为空栈

\* 初始条件:栈S已存在

\* @param S 栈S

\* @return 若S为空栈,则返回TRUE,否则返回FALSE

\*/

Status StackEmpty (Stack S);

/\*\*

\* 查询S的当前长度

\* 初始条件:栈S已存在

\* @param S 栈S

\* @return S中数据元素的个数

\*/

int StackLength (Stack S);

/\*\*

\* 查询S栈顶元素

\* 初始条件：栈已存在且非空栈

\* @param S 栈S

\* @param e 用于存放栈顶元素的值

\* @return 操作结果状态Status OK/ERROR

\*/

Status GetTop (Stack S, ElemType \* e);

/\*\*

\* 插入元素e为新的栈顶元素

\* 初始条件:栈S已存在

\* @param S 栈地址值

\* @param e 插入数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status Push (Stack \* S, ElemType e);

/\*\*

\* 删除栈S的栈顶元素，并送入e

\* 初始条件：栈S已存且非空

\* @param S 栈地址值

\* @param e 用于保留删除数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status Pop (Stack \* S, ElemType \* e);

/\*\*

\* 从栈底到栈顶依次对栈S中的元素使用函数visit进行访问

\* 初始条件:栈S已存在

\* @param S 栈S

\* @param visit 对栈S中每一元素所作访问操作

\* @return 操作结果状态Status OK/ERROR

\*/

Status StackTraverse (Stack S, Status (\*visitor) (ElemType \* Elem));

/\*\*

\* 打印数据元素Elem的值

\* @param Elem 栈S中的某一元素

\* @return 操作结果状态Status OK/ERROR

\*/

Status visit (ElemType \* Elem);

/\*\*

\* 从output.txt读取数据

\* @param S 待读取数据的栈

\* @return 操作结果状态Status OK/ERROR

\*/

Status LoadData (Stack \* S);

/\*\*

\* 保存数据元素Elem的值至output.txt

\* @param S 当前栈

\* @return 操作结果状态Status OK/ERROR

\*/

Status SaveData (Stack S);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 以下所有定义用于实现表达式求值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//定义操作数类型为int类型

typedef int OperandType;

//定义运算符优先级表

static char PriorTable[7][7] =

{

/\* '+' '-' '\*' '/' '(' ')' '#' \*/

/\*'+'\*/ {'>','>','<','<','<','>','>'},

/\*'-'\*/ {'>','>','<','<','<','>','>'},

/\*'\*'\*/ {'>','>','>','>','<','>','>'},

/\*'/'\*/ {'>','>','>','>','<','>','>'},

/\*'('\*/ {'<','<','<','<','<','=',' '},

/\*')'\*/ {'>','>','>','>',' ','>','>'},

/\*'#'\*/ {'<','<','<','<','<',' ','='}

};

//定义运算符常量表

enum {PLUS, REDUCE, RIDE, SUB, LEFTPARE, RIGHTPARE, SHARP};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 以下定义用于建立Operand类型的栈及其操作

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//定义顺序存储栈数据结构

typedef struct

{

OperandType \* base; //在栈构造之前和销毁之后，base的值为NULL

OperandType \* top; //栈顶指针

int stacksize; //当前分配存储总容量(以sizeof(ElemType)为单位)

} OPND;

/\*\*

\* 构造一个空栈S

\* @param S 栈地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitOPND (OPND \* S);

/\*\*

\* 插入元素e为新的栈顶元素

\* 初始条件:栈S已存在

\* @param S 栈地址值

\* @param e 插入数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status PushOPND (OPND \* S, OperandType e);

/\*\*

\* 删除栈S的栈顶元素，并送入e

\* 初始条件：栈S已存且非空

\* @param S 栈地址值

\* @param e 用于保留删除数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status PopOPND (OPND \* S, OperandType \* e);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 以下定义用于实现表达式求值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* 根据运算符计算结果

\* @param a 操作数a

\* @param theta 运算符

\* @param b 操作数b

\* @return 运算结果

\*/

OperandType Operate(OperandType a, char theta, OperandType b);

/\*\*

\* 计算两个运算符的优先级

\* @param stackOperator 退栈运算符

\* @param otherOperator 用户输入运算符

\* @return 运算符优先级'>','=','<'

\*/

char Precede(char stackOperator, char otherOperator);

/\*\* 计算表达式的值 \*/

OperandType EvaluateExpression(void);

#endif /\*STACK\_H\_INCLUDED\*/

(3)Stack.c：栈基本操作实现

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验三：顺序存储栈基本操作实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Stack.h"

Status InitStack (Stack \* S)

{

//构造栈底

S->base = (ElemType \*)malloc(STACK\_INIT\_SIZE \* sizeof(ElemType));

if (! S->base ) exit(OVERFLOW);

//构造栈顶

S->top = S->base;

//初始化栈长

S->stacksize = 0;

return OK;

}

Status DestroyStack (Stack \* S)

{

//栈S不存在，函数错误执行

if (! S->base) return ERROR;

//销毁栈底指针所指内存单元

free(S->base);

//栈底指针重置为NULL,栈顶指针重置为栈底指针，栈长重置为0

S->base = NULL;

S->top = S->base;

S->stacksize = 0;

return OK;

}

Status ClearStack (Stack \* S)

{

//栈S不存在，函数错误执行

if (! S->base) return ERROR;

//栈顶指针重置为栈底指针，栈长重置为0

S->top = S->base;

S->stacksize = 0;

return OK;

}

Status StackEmpty (Stack S)

{

//栈S不存在，函数错误执行

if (! S.base) return ERROR;

return S.base == S.top ? TRUE : FALSE;

}

int StackLength (Stack S)

{

//栈S不存在，函数错误执行

if (! S.base) return ERROR;

return S.top - S.base;

}

Status GetTop (Stack S, ElemType \* e)

{

//栈为空，函数错误执行

if (S.top == S.base) return ERROR;

//利用e返回栈顶元素值

(\*e) = \*(S.top - 1);

return OK;

}

Status Push (Stack \* S, ElemType e)

{

//栈S不存在，函数错误执行

if (! S->base) return ERROR;

//插入元素e为新的栈顶元素

if (S->top - S->base >= S->stacksize)

{ //满栈，追加空间

S->base = (ElemType \*) realloc(S->base, (S->stacksize + STACKINCREMENT) \* sizeof(ElemType));

if (! S->base) exit(OVERFLOW);

S->top = S->base + S->stacksize;

S->stacksize += STACKINCREMENT;

}

\*(S->top)++ = e;

return OK;

}

Status Pop (Stack \* S, ElemType \* e)

{

//栈为空，函数错误执行

if (S->top == S->base) return ERROR;

\*e = \*(--S->top);

return OK;

}

Status StackTraverse (Stack S, Status (\*visitor) (ElemType \* Elem))

{

//循环变量

ElemType \*ptr;

//栈为空，函数错误执行

if (S.base == S.top) return ERROR;

//遍历输出栈中元素

ptr = S.base;

while (S.top != ptr)

{

visitor(ptr++);

}

return OK;

}

Status visit (ElemType \* Elem)

{

printf("%c\n", \*Elem);

return OK;

}

Status LoadData (Stack \* S)

{

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","r")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//读取数据

while(fread(S->top, sizeof(ElemType), 1, fp))

{

(S->top)++;

}

fclose(fp);

return OK;

}

Status SaveData (Stack S)

{

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","w")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//保存数据

fwrite(S.base, sizeof(ElemType), StackLength(S), fp);

fclose(fp);

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InitOPND (OPND \* S)

{

//构造栈底

S->base = (OperandType \*)malloc(STACK\_INIT\_SIZE \* sizeof(OperandType));

if (! S->base ) exit(OVERFLOW);

//构造栈顶

S->top = S->base;

//初始化栈长

S->stacksize = 0;

return OK;

}

Status PushOPND (OPND \* S, OperandType e)

{

//栈S不存在，函数错误执行

if (! S->base) return ERROR;

//插入元素e为新的栈顶元素

if (S->top - S->base >= S->stacksize)

{ //满栈，追加空间

S->base = (OperandType \*) realloc(S->base, (S->stacksize + STACKINCREMENT) \* sizeof(OperandType));

if (! S->base) exit(OVERFLOW);

S->top = S->base + S->stacksize;

S->stacksize += STACKINCREMENT;

}

\*(S->top)++ = e;

return OK;

}

Status PopOPND (OPND \* S, OperandType \* e)

{

//栈为空，函数错误执行

if (S->top == S->base) return ERROR;

\*e = \*(--S->top);

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

char Precede(char stackOperator, char otherOperator)

{

int row, col; //运算符优先级表的行列号

//根据第一个运算符确定行号

switch(stackOperator)

{

case '+':

row = PLUS;

break;

case '-':

row = REDUCE;

break;

case '\*':

row = RIDE;

break;

case '/':

row = SUB;

break;

case '(':

row = LEFTPARE;

break;

case ')':

row = RIGHTPARE;

break;

case '#':

row = SHARP;

break;

default:

//返回$符号，表示函数执行错误

return '$';

}

//根据第二个运算符确定列号

switch(otherOperator)

{

case '+':

col = PLUS;

break;

case '-':

col = REDUCE;

break;

case '\*':

col = RIDE;

break;

case '/':

col = SUB;

break;

case '(':

col = LEFTPARE;

break;

case ')':

col = RIGHTPARE;

break;

case '#':

col = SHARP;

break;

default:

//返回$符号，表示函数执行错误

return '$';

}

//根据行列号返回目标运算符优先级

return PriorTable[row][col];

}

OperandType Operate(OperandType a, char theta, OperandType b)

{

//根据运算符返回计算结果

switch(theta)

{

case '+':

return a + b;

case '-':

return a - b;

case '\*':

return a \* b;

case '/':

return a / b;

default :

return 0;

}

}

OperandType EvaluateExpression(void)

{

Stack OPTR; //OPTR:运算符栈

OPND OPND; //OPND:操作数栈

char input; //用户输入表达式字符

char theta; //退栈运算符

OperandType a, b; //Operand型退栈操作数

OperandType f; //Operand型计算结果

//初始化两个栈，并将'#'压入运算符栈

InitStack(&OPTR);

Push(&OPTR, '#');

InitOPND(&OPND);

//读入表达式

input = getchar();

while (input != '#' || (GetTop(OPTR, &theta), theta != '#') )

{ //不是运算符则进栈

if(input >= '0' && input <= '9')

{

f = (int) (input - '0');

PushOPND(&OPND, f);

input = getchar();

}

else if (input == ' ' || input == '\n')

{ //过滤控制字符和空格符

input = getchar();

}

else

{

GetTop(OPTR, &theta);

switch (Precede(theta, input))

{

case '<': //栈顶元素优先级低

Push(&OPTR, input);

input = getchar();

break;

case '=': //脱括号并接收下一字符

Pop(&OPTR, &input);

input = getchar();

break;

case '>': //退栈并将运算结果入栈

//将待计算的操作数和运算符退栈取出

Pop(&OPTR, &theta);

PopOPND(&OPND, &b);

PopOPND(&OPND, &a);

//将计算结果压入操作数栈

f = Operate(a, theta, b);

PushOPND(&OPND, f);

break;

case '$': //读入不明字符，计算失败

scanf("%\*[^\n]"); //清空输入流

return INT\_MAX;

break;

}

}

}

//返回运算结果

PopOPND(&OPND, &f);

return f;

}

(4)main.c：系统主程序

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验三：顺序存储栈主程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Stack.h"

int main (void)

{

Stack stack; //栈结构

int choice; //用户所选功能编号

ElemType container;//用于存放各个功能函数反馈的数据元素值

Status (\*test\_visit)(ElemType \*) = visit;

//初始化栈

InitStack(&stack);

//读取文本数据

LoadData(&stack);

//打印系统菜单界面

printf ("\t\tWelcome to Stack System\n\n");

while (TRUE)

{

//打印系统菜单界面

printf ("1:Clear Stack\n");

printf ("2:Require whether Empty or not\n");

printf ("3:Require Stack Length\n");

printf ("4:Require Top Element\n");

printf ("5:Push Element\n");

printf ("6:Pop Element\n");

printf ("7:Traverse Stack\n");

printf ("8:Evaluate Expression\n");

printf ("9:Exit\n\n");

//提示用户选择功能

printf ("Please input function number:");

//用户选择事件

scanf ("%d", &choice);

//处理用户选择事件

switch (choice)

{

case 1:

ClearStack (&stack);

printf ("Clear Success\n");

break;

case 2:

if (StackEmpty (stack) == TRUE)

{

printf ("Stack is Empty\n");

}

else

{

printf ("Stack is not Empty\n");

}

break;

case 3:

container = StackLength (stack);

printf ("Stack Length:%d\n", container);

break;

case 4:

if (GetTop (stack, &container) == ERROR)

{

printf ("Require Failed\n");

}

else

{

printf ("Top Element:%c\n", container);

}

break;

case 5:

printf ("Please input the value of Element what to be pushed:");

scanf ("%1s", &container);

if (Push (&stack, container) == ERROR)

{

printf ("Push Failed\n");

}

else

{

printf ("Push %c Success\n", container);

}

break;

case 6:

if (Pop (&stack, &container) == ERROR)

{

printf ("Pop Failed\n");

}

else

{

printf ("Pop %c Success\n", container);

}

break;

case 7:

if(StackTraverse (stack, test\_visit) == ERROR)

{

printf("Traverse Failed\n");

}

break;

case 8:

printf("Please input espression end with '#':\n");

container = EvaluateExpression();

//根据运算结果反馈用户

if (container == INT\_MAX)

{ //计算失败

printf("\*\*\* Evaluate Failed.Please check your expression \*\*\*\n");

}

else if (container == INFEASIBLE)

{ //未以'#'结尾

printf("\*\*\* Evalueate Failed.Please end with '#'\*\*\*\n");

}

else

{ //计算成功

printf(": = %d", container);

}

break;

case 9:

//保存数据

SaveData(stack);

//销毁栈

DestroyStack (&stack);

exit (OK);

break;

default:

printf ("No Such Function");

break;

}

printf ("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

return OK;

}

(5)assert\_test.c：断言测试主程序

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验三：顺序存储栈断言测试

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 测试原则 :

\* - 动态操作后检查栈状态和长度

\* - 静态操作后不检查栈状态和长度

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <assert.h>

#include "Stack.h"

#undef NDEBUG //开启断言测试

#define TEST\_LENGTH 10 //定义栈测试长度为10

#define TEST\_START\_CHAR 49 //定义起始测试字符为'1'

int main (void)

{

//断言测试用变量

Stack test\_stack;

test\_stack.base = NULL;

test\_stack.top = test\_stack.base;

ElemType test\_elem;

int i;

Status (\*test\_visit) (ElemType \*) = visit;

//遍历栈(失败)

assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//初始化栈

assert (InitStack (&test\_stack) == OK);

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//查询长度及空栈

assert (StackLength (test\_stack) == 0);

assert (StackEmpty (test\_stack) == TRUE);

//销毁栈

assert (DestroyStack (&test\_stack) == OK);

//再次初始化栈

assert (InitStack (&test\_stack) == OK);

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//查询长度及空栈

assert (StackLength (test\_stack) == 0);

assert (StackEmpty (test\_stack) == TRUE);

//压栈

for (i = TEST\_START\_CHAR; i < TEST\_LENGTH + TEST\_START\_CHAR; i++)

{

assert (Push (&test\_stack, i) == OK);

printf ("PushElement : %c\n", i);

}

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == OK);

//查询长度及空栈

assert (StackLength (test\_stack) == TEST\_LENGTH);

assert (StackEmpty (test\_stack) == FALSE);

//退栈，并查询当前栈顶元素

for (i = 1; i <= TEST\_LENGTH / 2; i++)

{

assert (Pop (&test\_stack, &test\_elem) == OK);

printf ("PopElement : %c\n", test\_elem);

assert (GetTop (test\_stack, &test\_elem) == OK);

printf ("TopElement : %c\n", test\_elem);

}

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == OK);

//查询长度及空栈

assert (StackLength (test\_stack) == TEST\_LENGTH / 2);

assert (StackEmpty (test\_stack) == FALSE);

//重置栈

assert (ClearStack (&test\_stack) == OK);

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//查询长度及空栈

assert (StackLength (test\_stack) == 0);

assert (StackEmpty (test\_stack) == TRUE);

system("pause");

return OK;

}

(6)Makefile：编译链接文件

DT\_LAB\_03.exe : Stack.o main.o

gcc -Wall -std=c99 -g -o DT\_LAB\_03.exe Stack.o main.o

Stack.o : Stack.c Stack.h

gcc -Wall -std=c99 -g -c Stack.c

main.o : main.c Stack.h

gcc -Wall -std=c99 -g -c main.c

3.1.3.2 测试部分

本次测试采用断言测试，测试主程序及源代码见附件AssertTest.exe及程序清单assert\_test.c。

本次断言测试采用两种方式，一为普通断言，二为循环断言。普通断言即为单独调用一次目标函数，循环断言即为循环调用目标函数多次，并同时进行断言测试。普通断言测试函数：InitStack、DestroyStack、ClearStack、StackLength、StackEmpty、StackTraverse；循环断言测试函数：GetTop、Push、Pop。

每次调用静态函数后不对StackTraverse、StackLength、StackEmpty进行断言测试，每次调用动态函数后都对StackTraverse、StackLength、StackEmpty进行断言测试。

测试流程如下：（测试流程与上次实验略有差异）

//遍历栈(失败)

assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//初始化栈

assert (InitStack (&test\_stack) == OK);

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//查询长度及空栈

assert (StackLength (test\_stack) == 0);

assert (StackEmpty (test\_stack) == TRUE);

//销毁栈

assert (DestroyStack (&test\_stack) == OK);

//再次初始化栈

assert (InitStack (&test\_stack) == OK);

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//查询长度及空栈

assert (StackLength (test\_stack) == 0);

assert (StackEmpty (test\_stack) == TRUE);

//压栈

for (i = TEST\_START\_CHAR; i < TEST\_LENGTH + TEST\_START\_CHAR; i++)

{

assert (Push (&test\_stack, i) == OK);

}

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == OK);

//查询长度及空栈

assert (StackLength (test\_stack) == TEST\_LENGTH);

assert (StackEmpty (test\_stack) == FALSE);

//退栈，并查询当前栈顶元素

for (i = 1; i <= TEST\_LENGTH / 2; i++)

{

assert (Pop (&test\_stack, &test\_elem) == OK);

assert (GetTop (test\_stack, &test\_elem) == OK);

}

//遍历栈

assert (StackTraverse (test\_stack, test\_visit) == OK);

//查询长度及空栈

assert (StackLength (test\_stack) == TEST\_LENGTH / 2);

assert (StackEmpty (test\_stack) == FALSE);

//重置栈

assert (ClearStack (&test\_stack) == OK);

//遍历栈

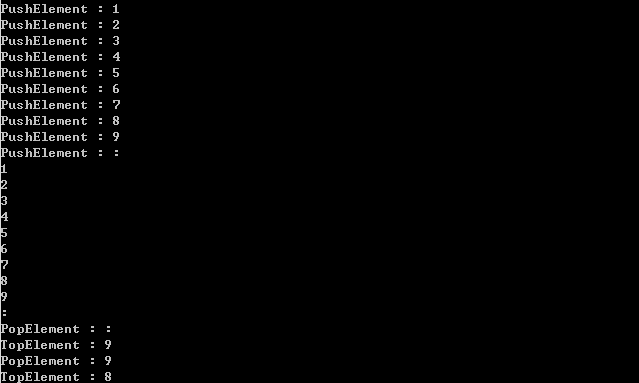
assert (StackTraverse (test\_stack, test\_visit) == ERROR);

//查询长度及空栈

assert (StackLength (test\_stack) == 0);

assert (StackEmpty (test\_stack) == TRUE);

测试结果如图3-1所示，所有断言成功，程序正常退出，结果符合预期。



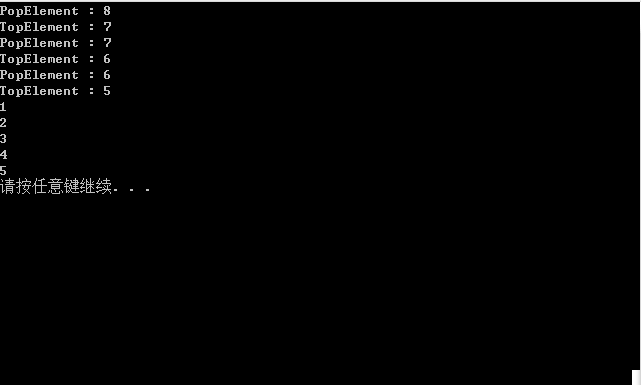


图3-1 断言测试结果图

3.2 表达式求值实验

3.2.1 问题描述

3.2.1.1 实验目的

利用栈，实现表达式的求值，对含运算符+、-、\*、/和1位数的操作数的表达式求值。

3.2.1.2 实验任务

利用顺序栈实现简单表达式的求值算法。

3.2.2算法设计部分

3.2.2.1 有关常量和类型定义

//定义操作数类型为int类型

typedef int OperandType;

//定义运算符优先级表

static char PriorTable[7][7] =

{

/\* '+' '-' '\*' '/' '(' ')' '#' \*/

/\*'+'\*/ {'>','>','<','<','<','>','>'},

/\*'-'\*/ {'>','>','<','<','<','>','>'},

/\*'\*'\*/ {'>','>','>','>','<','>','>'},

/\*'/'\*/ {'>','>','>','>','<','>','>'},

/\*'('\*/ {'<','<','<','<','<','=',' '},

/\*')'\*/ {'>','>','>','>',' ','>','>'},

/\*'#'\*/ {'<','<','<','<','<',' ','='}

};

//定义运算符常量表

enum {PLUS, REDUCE, RIDE, SUB, LEFTPARE, RIGHTPARE, SHARP};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 以下定义用于建立Operand类型的栈及其操作

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//定义顺序存储栈数据结构

typedef struct

{

OperandType \* base; //在栈构造之前和销毁之后，base的值为NULL

OperandType \* top; //栈顶指针

int stacksize; //当前分配存储总容量(以sizeof(ElemType)为单位)

} OPND;

3.2.2.2 实现算法

1.InitOPND、 PushOPND、 PopOPND

实现算法与实验3.1无异，仅将ElemType改为OperandType

时间空间复杂度分析见实验3.1

2.char Precede(char Operator, char otherOperator)——计算运算符优先级

{

//switch语句：根据Operator确定运算符优先级表行号

case(+/-/\*///(/)/#) row = 0/1/2/3/4/5/6/7;

//switch语句：根据otherOperator确定运算符优先级表列号

column(+/-/\*///(/)/#) row = 0/1/2/3/4/5/6/7;

//根据行列号返回目标运算符优先级

return PriorTable[row][col];

}

T(n) = O(1) ; S(n) = O(1)

3. OperandType Operate(OperandType a, char theta, OperandType b)

{

//根据运算符返回计算结果

switch(theta)

{ case '+': return a + b;

case '-': return a - b;

case '\*': return a \* b;

case '/': return a / b;

default : return 0;

}

}

T(n) = O(1) ; S(n) = O(1)

4. OperandType EvaluateExpression(void)

 T(n) = O(n) ; S(n) = O(1)

3.2.3实现与测试部分

3.2.3.1 实现部分

1.编程环境

编辑器：Sublime Text 3

编译器：MinGW-gcc 4.9.1(C99标准)

调试器：MinGW-gdb 7.8

2.函数调用关系

 3.程序清单——见实验3.1程序清单

已将将本实验实现表达式求值功能作为一个功能项(**功能8**)整合进实验3.1中所实现顺序栈操作系统，程序清单一并在实验3.1贴出。

3.2.3.2 测试部分

表3-1 EvaluateExpression算法测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| 用例1 | 3\*(7-2)# | 15 | C:\Users\gzsxkj171\AppData\Roaming\Tencent\Users\2426543832\QQ\WinTemp\RichOle\6KU0$UQJDU09BHZF`L$CZF9.png |
| 用例2 | 3\*7-2# | 19 | C:\Users\gzsxkj171\AppData\Roaming\Tencent\Users\2426543832\QQ\WinTemp\RichOle\M[MHG(4G%_28(J`X_1}J~BY.png |
| 用例3 | 3/3-2\*(5-3)# | -3 | C:\Users\gzsxkj171\AppData\Roaming\Tencent\Users\2426543832\QQ\WinTemp\RichOle\R0%)[EHLH_U4SEVXV(I85RW.png |
| 用例4 | 3/3-2\*(5-3) Ctrl+Z | 中断 | C:\Users\gzsxkj171\AppData\Roaming\Tencent\Users\2426543832\QQ\WinTemp\RichOle\6Y0PBIEU9`79ARY$2TBZYRL.png |
| 用例5 | 3$3-3\*4 | 中断 | C:\Users\gzsxkj171\AppData\Roaming\Tencent\Users\2426543832\QQ\WinTemp\RichOle\}F9JG6AW[$JN]U(R_9~%I1R.png |
| 用例6 | 3$3-3\*4# | 中断 | C:\Users\gzsxkj171\AppData\Roaming\Tencent\Users\2426543832\QQ\WinTemp\RichOle\XY_P{OKKII%0MVCV7Q)EH_W.png |

根据理论结果与运行结果的分析，测试符合预期，算法无误。

3.3 实验小结

一、借助断言测试，极大地缩短了Debug时长与难度，也省去了人工输入测试用例的麻烦。在进行完断言测试并将相关函数完成后，再进行main系统主程序的编写，编写主程序的过程异常顺利。同时，断言测试也缩短了编写系统主程序的时长与难度。

二、通过实现顺序存储栈的基本操作，加强了对栈的理解和使用，希望能为以后的数据结构学习带来帮助。

三、本次实验遇到最大的挑战便是表达式求值算法的实现。通过“自顶向下”的设计原则，将该算法一步步拆分，逐渐地实现了表达式求值的功能。

**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  (40分) | 程序规范  (10分) | 报告内容  (40分) | 报告规范  (10分) | 逾 期  扣 分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |

# 4基于循环队列存储结构实现队列的基本运算

4.1 问题描述

4.1.1 实验任务

采用顺序存储作为队列的物理结构，循环队列作为队列的逻辑结构，实现所列循环队列的全部操作。

4.1.2 实验目的

（1）加深对队列的概念、基本运算的理解；

（2）熟练掌握队列的逻辑结构与物理结构的关系；

（3）熟练掌握循环队列的算法实现。

4.1.3 实验算法

（1）循环队列的初始化、销毁、重置；

（2）循环队列的空队列查询、长度查询、队首元素值查询；

（3）循环队列的入列、出列、遍历。

4.2 循环队列设计

4.2.1 系统总体设计

本系统为顺序存储循环队列使用系统。本系统可自动初始化队列与自动销毁队列。本系统可自动将队列保存在文件output.txt中。本系统具有以下几大功能：

1.查询功能

查询空队列状态、查询队列长度、查询队首元素

2.修改功能

入列、出列

3.打印功能

遍历输出完整队列

4.2.2 有关常量和类型定义

1.所有实验公有部分：见程序清单DataStructure.h

2.本实验常量和类型定义

//最大队列长度

#define MAXQSIZE 100

//定义顺序存储循环队列数据结构

typedef struct Queue

{

ElemType \* base; //初始化的动态分配存储空间

int front; //头指针，若队列不空，指向队首元素

int rear; //尾指针，若队列不空，指向队尾元素的下一位置

} Queue;

4.2.3 算法设计

1.InitQueue (Queue Q)

//分配定长字节内存单元，并进行成功性检查

Q->base = (ElemType \*)malloc(MAXQSIZE \* sizeof(ElemType));

//初始化队首与队尾值

Q->rear = Q->front = 0;

T(n) = O(1) , S(n) = O(1)

2.DestroyQueue (Queue Q)

//释放内存空间

free(Q->base);

//将base重置为null，front与rear重置为0

Q->base = NULL;

Q->rear = Q->front = 0;

T(n) = O(1) , S(n) = O(1)

3.ClearQueue (Queue Q)

//将front与rear重置为0

Q->rear = Q->front = 0;

T(n) = O(1) , S(n) = O(1)

4.QueueEmpty (Queue Q)

return Q.rear == Q.front ? TRUE : FALSE;

T(n) = O(1) , S(n) = O(1)

5.QueueLength (Queue Q)

return ( (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE );

T(n) = O(1) , S(n) = O(1)

6.GetHead (Queue Q, ElemType e)

//利用e返回栈顶元素值

e = Q.base[Q.front];

T(n) = O(1) , S(n) = O(1)

7.EnQueue (Queue Q, ElemType e)

Q->base[Q->rear] = e;

//队列尾下标自增

Q->rear = (Q->rear + 1) % MAXQSIZE;

T(n) = O(1) , S(n) = O(1)

8.DeQueue (Queue Q, ElemType e)

e = Q->base[Q->front];

Q->front = (Q->front + 1) % MAXQSIZE;

T(n) = O(1) , S(n) = O(1)

9.QueueTraverse (Queue S, visitor)

//遍历输出元素

for (pos = Q.front, count = 0;count < QueueLength(Q);count++, pos = (pos + 1) % MAXQSIZE)

{

//访问目标元素

Visit(Q.base + pos);

}

T(n) = O(n) , S(n) = O(1)

10.LoadData (Queue S)

//将保存的第一个数据赋给Q->front

fread(&(Q->front), sizeof(int), 1, fp);

//初始化队尾rear值

Q->rear = Q->front;

//从指定文件读取数据，并进行入列操作

while(fread(Q->base + Q->rear, sizeof(ElemType), 1, fp))

{

Q->rear = (Q->rear + 1) % MAXQSIZE;

}

T(n) = O(n) , S(n) = O(1)

11.SaveData (Queue S)

//第一个数据用于保存Q.front值

fwrite(&(Q.front), sizeof(int), 1, fp);

fwrite(Q.base + Q.front, sizeof(ElemType), QueueLength(Q), fp);

T(n) = O(1) , S(n) = O(1)

4.3 顺序栈实现与测试

4.3.1 实现部分

1.编程环境

编辑器：Sublime Text 3

编译器：MinGW-gcc 4.9.1(C99标准)

调试器：MinGW-gdb 7.8

2.文件调用关系



3.函数调用关系



4.程序清单

(1)DataStructure.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验四：顺序存储循环队列

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef DATASTRUCTURE\_H\_INCLUDED

#define DATASTRUCTURE\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

//自定义宏

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

//Status是函数的类型,其值是函数结果状态代码

typedef int Status;

//将ElemType定义为int型

typedef int ElemType;

#endif /\*DATASTRUCTURE\_H\_INCLUDED\*/

(2)Queue.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验四：顺序存储循环队列抽象数据类型定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef QUEUE\_H\_INCLUDED

#define QUEUE\_H\_INCLUDED

#include "DataStructure.h"

//最大队列长度

#define MAXQSIZE 100

//定义顺序存储循环队列数据结构

typedef struct Queue

{

ElemType \* base; //初始化的动态分配存储空间，在队列构造之前和销毁之后，base的值为NULL

int front; //头指针，若队列不空，指向队列首元素

int rear; //尾指针，若队列不空，指向队列尾元素的下一个位置

} Queue;

/\*\*

\* 构造一个空队列Q

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitQueue (Queue \* Q);

/\*\*

\* 销毁队列Q

\* 初始条件:队列Q已存在

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DestroyQueue (Queue \* Q);

/\*\*

\* 将Q重置为空队列

\* 初始条件:队列已存在

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status ClearQueue (Queue \* Q);

/\*\*

\* 查询Q是否为空队列

\* 初始条件:队列Q已存在

\* @param Q 队列Q

\* @return 若Q为空队列,则返回TRUE,否则返回FALSE

\*/

Status QueueEmpty (Queue Q);

/\*\*

\* 查询Q的当前长度

\* 初始条件:队列Q已存在

\* @param Q 队列Q

\* @return Q中数据元素的个数

\*/

int QueueLength (Queue Q);

/\*\*

\* 查询Q队列首元素

\* 初始条件：队列已存在且非空队列

\* @param Q 队列Q

\* @param e 用于存放队列首元素的值

\* @return 操作结果状态Status OK/ERROR

\*/

Status GetHead (Queue Q, int \* e);

/\*\*

\* 插入元素e为新的队列尾元素

\* 初始条件:队列Q已存在

\* @param Q 队列地址值

\* @param e 插入数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status EnQueue (Queue \* Q, int e);

/\*\*

\* 删除队列Q的队列首元素，并送入e

\* 初始条件：队列Q已存且非空

\* @param Q 队列地址值

\* @param e 用于保留删除数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DeQueue (Queue \* Q, int \* e);

/\*\*

\* 从队首到队尾依次对队列Q中的元素使用函数visit进行访问

\* 初始条件:队列Q已存在

\* @param Q 队列Q

\* @param visit 对队列Q中每一元素所作访问操作

\* @return 操作结果状态Status OK/ERROR

\*/

Status QueueTraverse (Queue Q, Status (\*visitor) (ElemType \* Elem));

/\*\*

\* 打印数据元素Elem的值

\* @param Elem 队列Q中的某一元素

\* @return 操作结果状态Status OK/ERROR

\*/

Status visit (ElemType \* Elem);

/\*\*

\* 从output.txt读取数据

\* @param Q 待读取数据的队列

\* @return 操作结果状态Status OK/ERROR

\*/

Status LoadData (Queue \* Q);

/\*\*

\* 保存数据元素Elem的值至output.txt

\* @param Q 当前队列

\* @return 操作结果状态Status OK/ERROR

\*/

Status SaveData (Queue Q);

#endif /\*QUEUE\_H\_INCLUDED\*/

(3)Queue.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验四：顺序存储循环队列基本操作实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Queue.h"

Status InitQueue (Queue \* Q)

{

//分配定长字节内存单元，并进行成功性检查

Q->base = (ElemType \*)malloc(MAXQSIZE \* sizeof(ElemType));

if (! Q->base) return OVERFLOW;

//初始化

Q->rear = Q->front = 0;

return OK;

}

Status DestroyQueue (Queue \* Q)

{

//队列不存在

if (! Q->base) return ERROR;

//队列存在，函数正确执行

free(Q->base);

//将base重置为null，front与rear重置为0

Q->base = NULL;

Q->rear = Q->front = 0;

return OK;

}

Status ClearQueue (Queue \* Q)

{

//队列不存在

if (! Q->base) return ERROR;

//队列存在，函数正确执行

//将front与rear重置为0

Q->rear = Q->front = 0;

return OK;

}

Status QueueEmpty (Queue Q)

{

//队列不存在

if (! Q.base) return ERROR;

//队列存在，函数正确执行

return Q.rear == Q.front ? TRUE : FALSE;

}

int QueueLength (Queue Q)

{

//队列不存在

if (! Q.base) return ERROR;

//队列存在，函数正确执行

return ( (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE );

}

Status GetHead (Queue Q, ElemType \* e)

{

//队列不存在

if (! Q.base) return ERROR;

//队列为空

if (QueueEmpty(Q)) return ERROR;

//队列存在且非空

(\*e) = Q.base[Q.front];

return OK;

}

Status EnQueue (Queue \* Q, ElemType e)

{

//队列不存在

if (! Q->base) return ERROR;

//队列满

if ( (Q->rear + 1) % MAXQSIZE == Q->front ) return ERROR;

//队列存在且未满

Q->base[Q->rear] = e;

//队列尾下标自增

Q->rear = (Q->rear + 1) % MAXQSIZE;

return OK;

}

Status DeQueue (Queue \* Q, ElemType \* e)

{

//队列不存在

if (! Q->base) return ERROR;

//队列为空

if (QueueEmpty(\*Q)) return ERROR;

//队列存在且非空

(\*e) = Q->base[Q->front];

Q->front = (Q->front + 1) % MAXQSIZE;

return OK;

}

Status QueueTraverse (Queue Q, Status (\*visitor) (ElemType \* Elem))

{

int pos; //循环变量，用于标明当前元素下标

int count; //循环变量，用于记录当前循环次数(此变量值应不大于队列长度),判断何时终止遍历

//队列不存在

if (! Q.base) return ERROR;

//队列为空

if (QueueEmpty(Q)) return ERROR;

//队列存在且非空

for (pos = Q.front, count = 0;count < QueueLength(Q);count++, pos = (pos + 1) % MAXQSIZE)

{

//访问目标元素

visitor(Q.base + pos);

}

printf("\n");

return OK;

}

Status visit (ElemType \* Elem)

{

printf("%d\t", \*Elem);

return OK;

}

Status LoadData (Queue \* Q)

{

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","r")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//读取数据

//将保存的第一个数据赋给Q->front

fread(&(Q->front), sizeof(int), 1, fp);

//初始化队尾rear值

Q->rear = Q->front;

while(fread(Q->base + Q->rear, sizeof(ElemType), 1, fp))

{

Q->rear = (Q->rear + 1) % MAXQSIZE;

}

fclose(fp);

return OK;

}

Status SaveData (Queue Q)

{

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","w")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//保存数据

//第一个数据用于保存Q.front值

fwrite(&(Q.front), sizeof(int), 1, fp);

fwrite(Q.base + Q.front, sizeof(ElemType), QueueLength(Q), fp);

fclose(fp);

return OK;

}

(4)main.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验四：顺序存储循环队列主程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Queue.h"

int main (void)

{

Queue queue; //队列结构

queue.base = NULL;

ElemType input; //用户输入变量

ElemType container;//用于存放各个功能函数反馈的数据元素 1

Status (\*test\_visit) (ElemType \*) = visit;

//初始化队列

InitQueue (&queue);

//读取文本数据

LoadData(&queue);

//打印系统菜单界面

printf ("\t\t欢迎使用顺序队列\n\n");

while (TRUE)

{

//打印系统菜单界面

printf ("1:重置队列\n");

printf ("2:查询是否为空队列\n");

printf ("3:查询队列长\n");

printf ("4:查询队列顶元素\n");

printf ("5:插入队尾元素\n");

printf ("6:删除队首元素\n");

printf ("7:遍历队列\n");

printf ("8:退出系统\n\n");

//提示用户选择功能

printf ("请输入所选功能选项:");

//用户选择事件

scanf ("%d", &input);

//处理用户选择事件

switch (input)

{

case 1:

ClearQueue (&queue);

printf ("已重置\n");

break;

case 2:

if (QueueEmpty (queue) == TRUE)

{

printf ("当前队列为空队列\n");

}

else

{

printf ("当前队列不为空队列\n");

}

break;

case 3:

container = QueueLength (queue);

printf ("当前队列长为：%d\n", container);

break;

case 4:

if (GetHead (queue, &container) == ERROR)

{

printf ("查询失败\n");

}

else

{

printf ("队列头元素值为:%d\n", container);

}

break;

case 5:

printf ("请输入插入元素值:");

scanf ("%d", &container);

if (EnQueue (&queue, container) == ERROR)

{

printf ("插入失败\n");

}

else

{

printf ("插入 %d 成功\n", container);

}

break;

case 6:

if (DeQueue (&queue, &container) == ERROR)

{

printf ("删除失败\n");

}

else

{

printf ("删除 %d 成功\n", container);

}

break;

case 7:

if (QueueTraverse (queue, test\_visit) == ERROR)

{

printf("遍历失败\n");

}

break;

case 8:

//销毁队列

SaveData(queue);

DestroyQueue (&queue);

exit (INFEASIBLE);

break;

default:

printf ("所选功能不存在,请重新选择功能");

break;

}

printf ("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

}

(5)assert\_test.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验四：顺序存储循环队列断言测试

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 测试原则 :

\* - 动态操作后检查队列状态和长度

\* - 静态操作后不检查队列状态和长度

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <assert.h>

#include "Queue.h"

#undef NDEBUG //开启断言测试

#define TEST\_LENGTH 10 //定义队列测试长度为10

//将队列最大长度重设为10，便于测试队列的循环性

#undef MAXQSIZE

#define MAXQSIZE 10

int main (void)

{

//断言测试用变量

Queue test\_queue;

test\_queue.base = NULL;

ElemType test\_elem;

Status (\*test\_visit) (ElemType \*) = visit;

//遍历不存在队列

assert (QueueTraverse (test\_queue, test\_visit) == ERROR);

//初始化队列

assert (InitQueue (&test\_queue) == OK);

//遍历空队列

assert (QueueTraverse (test\_queue, test\_visit) == ERROR);

//查询长度及空队列

assert (QueueLength (test\_queue) == 0);

assert (QueueEmpty (test\_queue) == TRUE);

//销毁队列

assert (DestroyQueue (&test\_queue) == OK);

//再次初始化队列

assert (InitQueue (&test\_queue) == OK);

//查询长度及空队列

assert (QueueLength (test\_queue) == 0);

assert (QueueEmpty (test\_queue) == TRUE);

//压队列10次

for (int i = 1; i <= TEST\_LENGTH; i++)

{

assert (EnQueue (&test\_queue, i) == OK);

printf ("InsertElement : %d\n", i);

}

//遍历队列

assert (QueueTraverse (test\_queue, test\_visit) == OK);

//查询长度及空队列

assert (QueueLength (test\_queue) == TEST\_LENGTH);

assert (QueueEmpty (test\_queue) == FALSE);

//退队列5次，并查询当前队列顶元素

for (int i = 1; i <= TEST\_LENGTH / 2; i++)

{

assert (DeQueue (&test\_queue, &test\_elem) == OK);

printf ("DeleteElement : %d\n", test\_elem);

assert (GetHead (test\_queue, &test\_elem) == OK);

printf ("TopElement\t: %d\n", test\_elem);

}

//遍历队列

assert (QueueTraverse (test\_queue, test\_visit) == OK);

//查询长度及空队列

assert (QueueLength (test\_queue) == TEST\_LENGTH / 2);

assert (QueueEmpty (test\_queue) == FALSE);

//重置队列

assert (ClearQueue (&test\_queue) == OK);

//遍历空队列

assert (QueueTraverse (test\_queue, test\_visit) == ERROR);

//查询长度及空队列

assert (QueueLength (test\_queue) == 0);

assert (QueueEmpty (test\_queue) == TRUE);

printf("\n断言测试成功!\n");

system("pause");

return OK;

}

(6)Makefile

DT\_LAB\_04.exe : Queue.o main.o

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -o DT\_LAB\_04.exe Queue.o main.o

Queue.o : Queue.c Queue.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -c Queue.c

main.o : main.c Queue.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -c main.c

4.3.2 测试部分

本次测试采用断言测试，测试主程序及源代码见附件AssertTest.exe及程序清单assert\_test.c。

本次断言测试采用两种方式，一为普通断言，二为循环断言。普通断言即为单独调用一次目标函数，循环断言即为循环调用目标函数多次，并同时进行断言测试。普通断言测试函数：InitQueue、DestroyQueue、ClearQueue、QueueLength、QueueEmpty、QueueTraverse；循环断言测试函数：GetHead、EnQueue、DeQueue。

每次调用静态函数后不对QueueTraverse、QueueLength、QueueEmpty进行断言测试，每次调用动态函数后都对QueueTraverse、QueueLength、QueueEmpty进行断言测试。

测试流程如下：（测试流程与上次实验略有差异）

//遍历不存在队列

assert (QueueTraverse (test\_queue, test\_visit) == ERROR);

//初始化队列

assert (InitQueue (&test\_queue) == OK);

//遍历空队列

assert (QueueTraverse (test\_queue, test\_visit) == ERROR);

//查询长度及空队列

assert (QueueLength (test\_queue) == 0);

assert (QueueEmpty (test\_queue) == TRUE);

//销毁队列

assert (DestroyQueue (&test\_queue) == OK);

//再次初始化队列

assert (InitQueue (&test\_queue) == OK);

//查询长度及空队列

assert (QueueLength (test\_queue) == 0);

assert (QueueEmpty (test\_queue) == TRUE);

//压队列10次

for (int i = 1; i <= TEST\_LENGTH; i++)

{

assert (EnQueue (&test\_queue, i) == OK);

printf ("InsertElement : %d\n", i);

}

//遍历队列

assert (QueueTraverse (test\_queue, test\_visit) == OK);

//查询长度及空队列

assert (QueueLength (test\_queue) == TEST\_LENGTH);

assert (QueueEmpty (test\_queue) == FALSE);

//退队列5次，并查询当前队列顶元素

for (int i = 1; i <= TEST\_LENGTH / 2; i++)

{

assert (DeQueue (&test\_queue, &test\_elem) == OK);

printf ("DeleteElement : %d\n", test\_elem);

assert (GetHead (test\_queue, &test\_elem) == OK);

printf ("TopElement\t: %d\n", test\_elem);

}

//遍历队列

assert (QueueTraverse (test\_queue, test\_visit) == OK);

//查询长度及空队列

assert (QueueLength (test\_queue) == TEST\_LENGTH / 2);

assert (QueueEmpty (test\_queue) == FALSE);

//重置队列

assert (ClearQueue (&test\_queue) == OK);

//遍历空队列

assert (QueueTraverse (test\_queue, test\_visit) == ERROR);

//查询长度及空队列

assert (QueueLength (test\_queue) == 0);

assert (QueueEmpty (test\_queue) == TRUE);

测试结果如图4-1所示，所有断言成功，程序正常退出，结果符合预期。

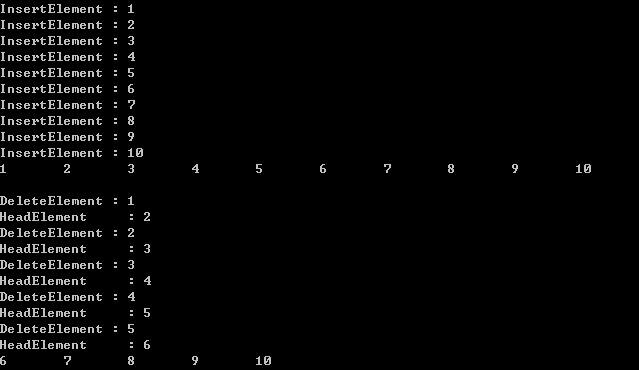


图4-1 断言测试结果图

4.4 实验小结

一、借助断言测试，极大地缩短了Debug时长与难度，也省去了人工输入测试用例的麻烦。在进行完断言测试并将相关函数完成后，再进行main系统主程序的编写，编写主程序的过程异常顺利。同时，断言测试也缩短了编写系统主程序的时长与难度。

二、通过实现顺序存储循环队列的基本操作，加强了对队列的理解和使用，希望能为以后的数据结构学习带来帮助。

**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  (40分) | 程序规范  (10分) | 报告内容  (40分) | 报告规范  (10分) | 逾 期  扣 分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |

# 5基于二叉链表实现二叉树的基本运算

5.1 问题描述

5.1.1 实验任务

实现对多个结点元素类型相同的二叉树的管理。其中每一个二叉树采用二叉链表作为物理结构，每一个二叉树的属性至少包含有二叉树名称。采用顺序表的方式管理多个线性表。其物理结构如图5-1所示。



图5-1 多二叉树管理的物理结构示意图

构造成具有功能菜单的系统，能对每棵二叉树实现二叉树的基本操作；多个二叉树采用1个或多个文件保存。

5.1.2 实验目的

（1）加深对二叉树的概念、基本运算的理解；

（2）熟练掌握二叉树的逻辑结构与物理结构的关系；

（3）以二叉链表作为物理结构，熟练掌握二叉树基本运算的实现。

5.1.3 实现算法

（1）基础模块

初始化、销毁、创建、清空二叉树。

（2）查询模块

空状态、深度、根结点、Data域查询；

左孩子、右孩子、左兄弟、右兄弟、双亲查询；

前序遍历、中序遍历、后序遍历、层序遍历；

凹入表形式打印、广义表形式打印。

（3）编辑模块

Data域赋值、插入子树、删除子树。

（4）数据模块

自动读取、自动保存。

5.2 二叉树设计

5.2.1 系统总体设计

本系统为二叉链存储多二叉树使用系统。本系统可自动初始化所有二叉树与自动销毁所有二叉树。本系统可自动保存与读取数据。

本系统具有以下几大功能：

一、切换功能

允许用户通过键入目标二叉树的ID号，切换至不同二叉树，以实现对多二叉树的操作。当前系统允许操作的二叉树总数为100，ID号为1-100。

二、基础功能

初始化、销毁、创建、清空二叉树；

自动加载与保存数据。

三、查询功能

空状态、深度、根结点、Data域查询；

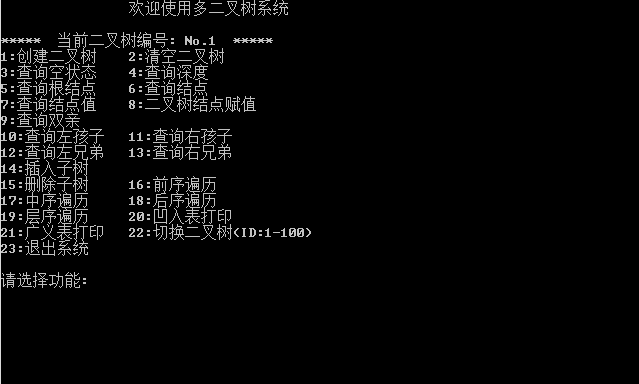
左孩子、右孩子、左兄弟、右兄弟、双亲查询；

前序遍历、中序遍历、后序遍历、层序遍历；

凹入表形式打印、广义表形式打印。

四、编辑功能

Data域赋值、插入子树、删除子树。



5.2.2 有关常量和类型定义

1.所有实验公有部分：见程序清单DataStructure.h

2.本实验常量和类型定义

//系统可操作二叉树的个数

#define TREE\_INIT\_SIZE 100

//定义二叉树的二叉链表存储

typedef struct BiTNode

{

ElemType data;

struct BiTNode \* lchild, \* rchild; //左右孩子指针

} BiTNode, \* BiTree;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 说明：利用队列,实现二叉树中部分函数(如层序遍历算法)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//最大队列长度

#define MAXQSIZE 100

//定义顺序存储循环队列数据结构

typedef struct Queue

{

BiTNode \* \* base; //初始化的动态分配存储空间，在队列构造之前和销毁之后，base的值为NULL

int front; //头指针，若队列不空，指向队列首元素

int rear; //尾指针，若队列不空，指向队列尾元素的下一个位置

} Queue;

5.2.3 算法设计

一、用于实现层序遍历的队列

1.InitQueue (Queue Q)

//分配定长字节内存单元，并进行成功性检查

Q->base = (ElemType \*)malloc(MAXQSIZE \* sizeof(ElemType));

//初始化队首与队尾值

Q->rear = Q->front = 0;

T(n) = O(1) , S(n) = O(1)

2.DestroyQueue (Queue Q)

//释放内存空间

free(Q->base);

//将base重置为null，front与rear重置为0

Q->base = NULL;

Q->rear = Q->front = 0;

T(n) = O(1) , S(n) = O(1)

3.QueueEmpty (Queue Q)

return Q.rear == Q.front ? TRUE : FALSE;

T(n) = O(1) , S(n) = O(1)

4.EnQueue (Queue Q, ElemType e)

Q->base[Q->rear] = e;

//队列尾下标自增

Q->rear = (Q->rear + 1) % MAXQSIZE;

T(n) = O(1) , S(n) = O(1)

5.DeQueue (Queue Q, ElemType e)

e = Q->base[Q->front];

Q->front = (Q->front + 1) % MAXQSIZE;

T(n) = O(1) , S(n) = O(1)

二、二叉树关键算法——基础模块

1. Status InitBiTree (BiTree \* T)

//构造根结点

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

//根结点data域置为空

(\*T)->data = '\0';

//左右子树置为空

(\*T)->lchild = (\*T)->rchild = NULL;

T(n) = O(1) , S(n) = O(1)

2. Status DestroyBiTree (BiTree T)

DestroyBiTree(T->lchild); //销毁左子树

DestroyBiTree(T->rchild); //销毁右子树

free(T); //销毁根结点

T(n) = O(n2) , S(n) = O(1)

3. Status CreateBiTree(BiTree \* T, char \* presequence,char \* insequence)

//序列为空，不构造二叉树

if(strlen(presequence) == 0 || strlen(insequence) == 0) \*T = NULL;

//截取左子树与右子树的前序序列与后序序列

lchild\_insequence = insequence(0, root);

rchild\_insequence = insequence(root, end);

lchild\_presequence = presequence(1, lchild\_insequence.length);

rchild\_presequence = presequence(lchild\_insequence.length, end);

//递归创建二叉树

(\*T) = (BiTree)malloc(sizeof(BiTNode)); //构造根结点

(\*T)->data = rootNode; //赋值根结点data域

CreateBiTree(&((\*T)->lchild), lchild\_presequence, lchild\_insequence); //创建左子树

CreateBiTree(&((\*T)->rchild), rchild\_presequence, rchild\_insequence); //创建右子树

T(n) = O(n2) , S(n) = O(n)

4. Status ClearBiTree (BiTree T)

//清空左右子树，data域置为空

T->lchild = T->rchild = NULL;

T->data = '\0';

T(n) = O(1) , S(n) = O(1)

5. Status BiTreeEmpty (BiTree T)

//根结点的左右子树都为空,且data域为空，二叉树为空

return (T->lchild == NULL && T->rchild == NULL && T->data == '\0') ? TRUE : FALSE;

T(n) = O(1) , S(n) = O(1)

6. int BiTreeDepth(BiTree T)

//二叉树为空,深度为0

if (BiTreeEmpty(T) == TRUE) depth = 0;

//二叉树的根结点为子叶结点,深度为1

else if(T->lchild == NULL && T->rchild == NULL) depth = 1;

//二叉树的根结点为内点

else if(T->lchild != NULL || T->rchild != NULL)

{ //二叉树的深度为子树最大深度+1

ldepth = BiTreeDepth(T->lchild); //左子树深度

rdepth = BiTreeDepth(T->rchild); //右子树深度

subdepth = ldepth > rdepth ? ldepth : rdepth;//子树最大深度

depth = subdepth + 1;

}

T(n) = O(n2) , S(n) = O(n)

7. BiTNode \* Root(BiTree T)

//二叉树存在,返回根结点

return T;

8. BiTNode \* Locate(BiTree T, BiTNode e)

//二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，判断根结点是否为e

else if (T->data == e.data) dest = T;

//否则，在左子树中寻找目标结点e

else if((dest = Locate(T->lchild, e)) != NULL) ;

//否则，在右子树中寻找目标结点e

else if((dest = Locate(T->rchild, e)) == NULL) dest = NULL;

T(n) = O(n2) , S(n) = O(n)

9. ElemType Value(BiTree T, BiTNode e)

//定位结点e，并返回其data域

if((pdest = Locate(T, e)) != NULL) dest = pdest->data;

else dest = '\0';

T(n) = O(n2) , S(n) = O(n)

10. Status Assign(BiTree T, BiTNode e, ElemType value)

//定位结点e,并修改data域为value

if((dest = Locate(T, e)) != NULL) dest->data = value;

else return ERROR;

T(n) = O(n2) , S(n) = O(n)

11. BiTNode \* Parent(BiTree T, BiTNode e)

/二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，判断根结点是否为e的双亲

else if (T->lchild)

{ //根结点有左孩子

if(T->lchild->data == e.data) dest = T;

}

else if (T->rchild)

{ //根结点有右孩子

if(T->rchild->data == e.data) dest = T;

//当前根结点不是目标结点的双亲,搜索左/右子树

if (dest == NULL)

{

//在左子树中寻找目标结点e的双亲

if((dest = Parent(T->lchild, e)) != NULL) ;

//否则，在右子树中寻找目标结点e的双亲

else if((dest = Parent(T->rchild, e)) == NULL) dest = NULL;

}

}

T(n) = O(n2) , S(n) = O(n)

12. BiTNode \* LeftChild(BiTree T, BiTNode e)

//定位结点e,返回左孩子指针

if((dest = Locate(T, e)) != NULL) dest = dest->lchild;

else dest = NULL;

T(n) = O(n2) , S(n) = O(n)

13. BiTNode \* RightChild(BiTree T, BiTNode e)

//定位结点e,返回右孩子指针

if((dest = Locate(T, e)) != NULL) dest = dest->rchild;

else dest = NULL;

T(n) = O(n2) , S(n) = O(n)

14. BiTNode \* LeftSibling(BiTree T, BiTNode e)

//定位结点e，返回左兄弟指针

if((dest = Locate(T, e)) != NULL)

{

//定位结点e的双亲结点

parent = Parent(T, e);

if(parent) //具有双亲结点

{ //结点e为左孩子，不存在左兄弟

if(dest == parent->lchild) dest = NULL;

//否则，返回结点e双亲的左孩子，即为e的左兄弟

else dest = parent->lchild;

}

}T(n) = O(n2) , S(n) = O(n)

15. BiTNode \* RightSibling(BiTree T, BiTNode e)

//定位结点e,返回右兄弟指针

if((dest = Locate(T, e)) != NULL)

{

//定位结点e的双亲结点

parent = Parent(T, e);

if(parent) //具有双亲结点

{ //结点e为右孩子，不存在右兄弟

if(dest == parent->rchild) dest = NULL;

//否则，返回结点e双亲的右孩子，即为e的右兄弟

else dest = parent->rchild;

}

}

T(n) = O(n2) , S(n) = O(n)

16.Status InsertChild(BiTree T, BiTNode p, int LR, BiTree c)

//定位结点p

if((dest = Locate(T, p)) != NULL)

{

switch (LR)

{ //将c插入为p的左子树

case 0:

//p的左子树作为c的右子树

c->rchild = dest->lchild;

//插入c

dest->lchild = c;

return OK;

//将c插入为p的右子树

case 1:

//p的右子树作为c的右子树

c->rchild = dest->rchild;

//插入c

dest->rchild = c;

return OK;

}

T(n) = O(n2) , S(n) = O(n)

17. Status DeleteChild(BiTree T, BiTNode p, int LR)

//定位结点p

if((dest = Locate(T, p)) != NULL)

{ switch (LR)

{

case 0:

//删除左子树

dest->lchild = NULL;

return OK;

case 1:

//删除右子树

dest->rchild = NULL;

return OK;

default:

return ERROR;

}

}

T(n) = O(n2) , S(n) = O(n)

18. Status PreOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

//二叉树不存在/空,遍历结束

if (T == NULL || BiTreeEmpty(T) == TRUE) return OK;

//递归遍历

if (visit(T))

if(PreOrderTraverse(T->lchild, visit))

if(PreOrderTraverse(T->rchild, visit))

return OK;

T(n) = O(n2) , S(n) = O(n)

19. Status InOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

//二叉树不存在/空,遍历结束

if (T == NULL || BiTreeEmpty(T) == TRUE) return OK;

//递归遍历

if(InOrderTraverse(T->lchild, visit))

if (visit(T))

if(InOrderTraverse(T->rchild, visit))

return OK;

T(n) = O(n2) , S(n) = O(n)

20. Status PostOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

//二叉树不存在/空,遍历结束

if (T == NULL || BiTreeEmpty(T) == TRUE) return OK;

//递归遍历

if(PostOrderTraverse(T->lchild, visit))

if(PostOrderTraverse(T->rchild, visit))

if (visit(T))

return OK;

T(n) = O(n2) , S(n) = O(n)

21. Status LevelOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

//利用队列实现广度优先搜索算法

//根结点入列

EnQueue(&queue,T);

//利用队列数据结构,实现广度搜索算法

while(!QueueEmpty(queue))

{ //队尾结点出列

DeQueue(&queue, &node);

//访问队尾结点

visit(node);

//将队尾结点的左/右孩子入列

if(node->lchild) EnQueue(&queue,node->lchild);

if(node->rchild) EnQueue(&queue,node->rchild);

}

T(n) = O(n2) , S(n) = O(n)

5.3 二叉树实现与测试

5.3.1 实现部分

1.编程环境

编辑器：Sublime Text 3

编译器：MinGW-gcc 4.9.1(C99标准)

调试器：MinGW-gdb 7.8

2.文件调用关系



3.函数调用关系



4.程序清单

(1)DataStructure.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验五：二叉链存储二叉树

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef DATASTRUCTURE\_H\_INCLUDED

#define DATASTRUCTURE\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

//自定义宏

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

//Status是函数的类型,其值是函数结果状态代码

typedef int Status;

//将ElemType定义为char型

typedef char ElemType;

#endif /\*DATASTRUCTURE\_H\_INCLUDED\*/

(2)Tree.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验五：二叉链存储二叉树抽象数据类型定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef TREE\_H\_INCLUDED

#define TREE\_H\_INCLUDED

#include "DataStructure.h" //标准库;公有类型与变量定义

//系统可操作二叉树的个数

#define TREE\_INIT\_SIZE 100

//定义二叉树的二叉链表存储

typedef struct BiTNode

{

ElemType data;

struct BiTNode \* lchild, \* rchild; //左右孩子指针

} BiTNode, \* BiTree;

/\*\*

\* 构造一个空二叉树T

\* @param T 二叉树

\* @call BiTreeEmpty

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitBiTree (BiTree \* T);

/\*\*

\* 销毁二叉树T

\* 初始条件:二叉树T已存在

\* @param T 二叉树

\* @return 操作结果状态Status OK/ERROR

\*/

Status DestroyBiTree (BiTree T);

/\*\*

\* 按前序和中序序列构造二叉树

\* @param T 二叉树

\* @param presequence 前序序列

\* @param insequence 中序序列

\* @call strlen

\* @call myStrncpy

\* @return 操作结果状态Status OK/ERROR

\*/

Status CreateBiTree(BiTree \* T, char \* presequence,char \* insequence);

/\*\*

\* 将T重置为空二叉树

\* 初始条件:二叉树已存在

\* @param T 二叉树

\* @return 操作结果状态Status OK/ERROR

\*/

Status ClearBiTree (BiTree T);

/\*\*

\* 查询T是否为空二叉树

\* 初始条件:二叉树T已存在

\* @param T 二叉树T

\* @return 若T为空二叉树,则返回TRUE,否则返回FALSE

\*/

Status BiTreeEmpty (BiTree T);

/\*\*

\* 查询T的深度

\* 初始条件:二叉树T已存在

\* @param T 二叉树T

\* @return T的层数

\*/

int BiTreeDepth(BiTree T);

/\*\*

\* 查找二叉树的根结点

\* 初始条件:二叉树T已存在

\* @param T 二叉树

\* @call BiTreeEmpty

\* @return T的根结点

\*/

BiTNode \* Root(BiTree T);

/\*\*

\* 定位结点e在树T中的位置

\* 初始条件:二叉树T已存在且不为空

\* @param T 二叉树

\* @param e 目标结点

\* @call BiTreeEmpty

\* @return e不在树中返回NULL;e在树中返回e的地址

\*/

BiTNode \* Locate(BiTree T, BiTNode e);

/\*\*

\* 查询结点e的值

\* 初始条件:二叉树T已存在，e是T中结点

\* @param T 二叉树

\* @param e T中结点

\* @call BiTreeEmpty

\* @call Locate

\* @return e的值

\*/

ElemType Value(BiTree T, BiTNode e);

/\*\*

\* 给结点e赋值value

\*

\* @param T 二叉树

\* @param e 目标结点

\* @param value 所赋值

\* @call BiTreeEmpty

\* @call Locate

\* @return 操作结果状态 OK/ERROR

\*/

Status Assign(BiTree T, BiTNode e, ElemType value);

/\*\*

\* 查找目标结点的双亲结点

\* 初始条件:二叉树T已存在,e是T中结点

\* @param T 二叉树

\* @param e 目标结点

\* @call BiTreeEmpty

\* @return 非根结点返回双亲结点指针，否则返回NULL

\*/

BiTNode \* Parent(BiTree T, BiTNode e);

/\*\*

\* 查找目标结点的左孩子结点

\* 初始条件:二叉树T已存在,e是T中结点

\* @param T 二叉树

\* @param e 目标结点

\* @call BiTreeEmpty

\* @call Locate

\* @return 返回左孩子结点指针/NULL

\*/

BiTNode \* LeftChild(BiTree T, BiTNode e);

/\*\*

\* 查找目标结点的右孩子结点

\* 初始条件:二叉树T已存在,e是T中结点

\* @param T 二叉树

\* @param e 目标结点

\* @call BiTreeEmpty

\* @call Locate

\* @return 返回右孩子结点指针/NULL

\*/

BiTNode \* RightChild(BiTree T, BiTNode e);

/\*\*

\* 查找目标结点的左兄弟结点

\* 初始条件:二叉树T已存在,e是T中结点

\* @param T 二叉树

\* @param e 目标结点

\* @call BiTreeEmpty

\* @call Locate

\* @call Parent

\* @return 返回左兄弟结点指针/NULL

\*/

BiTNode \* LeftSibling(BiTree T, BiTNode e);

/\*\*

\* 查找目标结点的右兄弟结点

\* 初始条件:二叉树T已存在,e是T中结点

\* @param T 二叉树

\* @param e 目标结点

\* @call BiTreeEmpty

\* @call Locate

\* @call Parent

\* @return 返回右兄弟结点指针/NULL

\*/

BiTNode \* RightSibling(BiTree T, BiTNode e);

/\*\*

\* 将c插入T中，c作为p的左子树/右子树

\* 初始条件: 二叉树T存在，p指向T中的某个结点，LR为0或1，,非空二

叉树c与T不相交且右子树为空

\* @param T 二叉树

\* @param p T中目标结点

\* @param LR 左右子树选择符

\* @param c 新结点(插入结点)

\* @call BiTreeEmpty

\* @call Locate

\* @return 操作结果状态 OK/ERROR

\*/

Status InsertChild(BiTree T, BiTNode p, int LR, BiTree c);

/\*\*

\* 删除T中的结点p

\* 初始条件: 二叉树T存在，p指向T中的某个结点，LR为0或1

\* @param T 二叉树

\* @param p T中目标结点

\* @param LR 左右子树选择符

\* @call BiTreeEmpty

\* @call Locate

\* @return 操作结果状态 OK/ERROR

\*/

Status DeleteChild(BiTree T, BiTNode p, int LR);

/\*\*

\* 先序遍历二叉树T

\* 初始条件:二叉树T已存在

\* @param T 二叉树

\* @param visit 访问操作函数

\* @call BiTreeEmpty

\* @call Visit

\* @return 操作结果状态 OK/ERROR

\*/

Status PreOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e));

/\*\*

\* 中序遍历二叉树T

\* 初始条件:二叉树T已存在

\* @param T 二叉树

\* @param visit 访问操作函数

\* @call BiTreeEmpty

\* @call Visit

\* @return 操作结果状态 OK/ERROR

\*/

Status InOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e));

/\*\*

\* 后序遍历二叉树T

\* 初始条件:二叉树T已存在

\* @param T 二叉树

\* @param visit 访问操作函数

\* @call BiTreeEmpty

\* @call Visit

\* @return 操作结果状态 OK/ERROR

\*/

Status PostOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e));

/\*\*

\* 层序遍历二叉树T

\* 初始条件:二叉树T已存在

\* @param T 二叉树

\* @param visit 访问操作函数

\* @call BiTreeEmpty

\* @call Visit

\* @call InitQueue/DestoryQueue/EnQueue/DeQueue

\* @return 操作结果状态 OK/ERROR

\*/

Status LevelOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e));

/\*\*

\* 输出结点e的值

\* @param e 二叉树T中结点

\* @return 操作结果状态Status OK/ERROR

\*/

Status Visit (BiTNode \* e);

/\*\*

\* 以凹入表形式打印二叉树T

\* 根结点位于中央，右子树位于上方，左子树位于下方

\* 初始条件:二叉树T已存在且非空

\* @param T 二叉树

\* @param instance 凹入表间距

\* @call BiTreeEmpty

\* @return 操作结果状态 OK/ERROR

\*/

Status RTPrint(BiTree T, int instance);

/\*\*

\* 以广义表形式打印二叉树T

\* 初始条件:二叉树T已存在且非空

\* @param T 二叉树

\* @call BiTreeEmpty

\* @return 操作结果状态 OK/ERROR

\*/

Status GLPrint(BiTree T);

/\*\*

\* 从output.txt读取数据

\* @param T 多二叉树头指针

\* @call CreateBiTree

\* @return 操作结果状态Status OK/ERROR

\*/

Status LoadData (BiTree \* T);

/\*\*

\* 保存数据至output.txt

\* @param T 多二叉树头指针

\* @param visit 访问操作函数指针(重定向标准输出流)

\* @call BiTreeEmpty

\* @call NodeNum

\* @call PreOrderTraverse

\* @call InOrderTraverse

\* @return 操作结果状态Status OK/ERROR

\*/

Status SaveData (BiTree \*T, Status (\*visit)(BiTNode \* e));

/\*\*

\* 改进strncpy库函数，使之自动在dest为添加'\0'

\* @param dest 目标字符串

\* @param src 元字符串

\* @param n 拷贝字符个数

\* @return 拷贝成功:dest,拷贝失败:NULL

\*/

char \*myStrncpy(char \*dest, const char \*src, size\_t n);

/\*\*

\* 计算二叉树结点总数,用于存储至output.txt

\* @param T 二叉树

\* @param num 结点总数地址值

\* @return 操作结果状态 OK/ERROR

\*/

Status NodeNum(BiTree T, int \*num);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 说明：利用队列,实现二叉树中部分函数(如层序遍历算法)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//最大队列长度

#define MAXQSIZE 100

//定义顺序存储循环队列数据结构

typedef struct Queue

{

BiTNode \* \* base; //初始化的动态分配存储空间，在队列构造之前和销毁之后，base的值为NULL

int front; //头指针，若队列不空，指向队列首元素

int rear; //尾指针，若队列不空，指向队列尾元素的下一个位置

} Queue;

/\*\*

\* 构造一个空队列Q

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitQueue (Queue \* Q);

/\*\*

\* 销毁队列Q

\* 初始条件:队列Q已存在

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DestroyQueue (Queue \* Q);

/\*\*

\* 查询Q是否为空队列

\* 初始条件:队列Q已存在

\* @param Q 队列Q

\* @return 若Q为空队列,则返回TRUE,否则返回FALSE

\*/

Status QueueEmpty (Queue Q);

/\*\*

\* 插入元素e为新的队列尾元素

\* 初始条件:队列Q已存在

\* @param Q 队列地址值

\* @param e 插入数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status EnQueue (Queue \* Q, BiTNode \* e);

/\*\*

\* 删除队列Q的队列首元素，并送入e

\* 初始条件：队列Q已存且非空

\* @param Q 队列地址值

\* @param e 用于保留删除数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DeQueue (Queue \* Q, BiTNode \* \* e);

#endif /\*TREE\_H\_INCLUDED\*/

(3)Tree.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验五：二叉链存储二叉树基本操作实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Tree.h"

Status InitBiTree (BiTree \* T)

{ //构造根结点

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

//根结点data域置为空

(\*T)->data = '\0';

//左右子树置为空

(\*T)->lchild = (\*T)->rchild = NULL;

//检查所构造树是否为空树

if (BiTreeEmpty(\*T) == TRUE) return OK;

else return ERROR;

}

Status DestroyBiTree (BiTree T)

{ //二叉树不存在,ERROR

if(T == NULL) return ERROR;

//销毁二叉树

DestroyBiTree(T->lchild); //销毁左子树

DestroyBiTree(T->rchild); //销毁右子树

free(T); //销毁根结点

return OK;

}

Status CreateBiTree(BiTree \* T, char \* presequence,char \* insequence)

{ //前/中序序列为空，构造的二叉树为空

if(strlen(presequence) == 0 || strlen(insequence) == 0)

{

\*T = NULL;

return ERROR;

}

char rootNode = presequence[0]; //根结点

int rootIndex = strchr(insequence, rootNode) - insequence; //根在中序序列中的位置

//截取左/右子树的中序序列

char \* lchild\_insequence = (char \*)malloc(strlen(insequence) \* sizeof(char)); //左子树的中序序列

char \* rchild\_insequence = (char \*)malloc(strlen(insequence) \* sizeof(char)); //右子树的中序序列

myStrncpy(lchild\_insequence, insequence, rootIndex ); //左子树的中序序列

myStrncpy(rchild\_insequence, insequence + rootIndex + 1, strlen(insequence) - rootIndex - 1);//右子树的中序序列

int lchild\_length = strlen(lchild\_insequence); //左子树的长度

int rchild\_length = strlen(rchild\_insequence); //右子树的长度

//截取左/右子树的前序序列

char \* lchild\_presequence = (char \*)malloc(strlen(presequence) \* sizeof(char)); //左子树的前序序列

char \* rchild\_presequence = (char \*)malloc(strlen(presequence) \* sizeof(char)); //右子树的前序序列

myStrncpy(lchild\_presequence, presequence + 1, lchild\_length); //左子树的前序序列

myStrncpy(rchild\_presequence, presequence + 1 + lchild\_length, rchild\_length); //右子树的前序序列

//构造根结点

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

//分配内存失败

if (\*T == NULL) return OVERFLOW;

//赋值根结点data域

(\*T)->data = rootNode;

CreateBiTree(&((\*T)->lchild), lchild\_presequence, lchild\_insequence); //创建左子树

CreateBiTree(&((\*T)->rchild), rchild\_presequence, rchild\_insequence); //创建右子树

return OK;

}

Status ClearBiTree (BiTree T)

{ //二叉树不存在，返回ERROR

if (T == NULL) return ERROR;

//清空左右子树，data域置为空

T->lchild = T->rchild = NULL;

T->data = '\0';

return OK;

}

Status BiTreeEmpty (BiTree T)

{ //二叉树不存在，返回ERROR

if (T == NULL) return ERROR;

//根结点的左右子树都为空,且data域为空，二叉树为空

return (T->lchild == NULL && T->rchild == NULL && T->data == '\0') ? TRUE : FALSE;

}

int BiTreeDepth(BiTree T)

{

int depth = 0; //二叉树的深度

int ldepth; //左子树深度

int rdepth; //右子树深度

int subdepth; //子树最大深度

//二叉树不存在,ERROR

if (T == NULL) return ERROR;

//二叉树为空,深度为0

if (BiTreeEmpty(T) == TRUE) depth = 0;

//二叉树的根结点为子叶结点,深度为1

else if(T->lchild == NULL && T->rchild == NULL) depth = 1;

//二叉树的根结点为内点

else if(T->lchild != NULL || T->rchild != NULL)

{ //二叉树的深度为子树最大深度+1

ldepth = BiTreeDepth(T->lchild); //左子树深度

rdepth = BiTreeDepth(T->rchild); //右子树深度

subdepth = ldepth > rdepth ? ldepth : rdepth;//子树最大深度

depth = subdepth + 1;

}

return depth;

}

BiTNode \* Root(BiTree T)

{ //二叉树不存在,NULL

if (T == NULL) return NULL;

//二叉树存在,返回根结点

return T;

}

BiTNode \* Locate(BiTree T, BiTNode e)

{

BiTNode \* dest = NULL; //目标结点

//二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，判断根结点是否为e

else if (T->data == e.data) dest = T;

//否则，在左子树中寻找目标结点e

else if((dest = Locate(T->lchild, e)) != NULL) ;

//否则，在右子树中寻找目标结点e

else if((dest = Locate(T->rchild, e)) == NULL) dest = NULL;

return dest;

}

ElemType Value(BiTree T, BiTNode e)

{

ElemType dest; //目标元素值

BiTNode \* pdest; //目标结点

//二叉树不存在/空,dest为ERROR

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = '\0';

//否则，定位结点e，返回data域

if((pdest = Locate(T, e)) != NULL) dest = pdest->data;

else dest = '\0';

return dest;

}

Status Assign(BiTree T, BiTNode e, ElemType value)

{

BiTNode \* dest; //目标结点

//二叉树不存在/空,ERROR

if (T == NULL || BiTreeEmpty(T) == TRUE) return ERROR;

//否则，定位结点e,并修改data域为value

if((dest = Locate(T, e)) != NULL) dest->data = value;

else return ERROR;

return OK;

}

BiTNode \* Parent(BiTree T, BiTNode e)

{

BiTNode \* dest = NULL; //目标结点

//二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，判断根结点是否为e的双亲

else if (T->lchild || T->rchild )

//以下多个if语句目的:防止Segmentation Fault

{ //根结点有左孩子

if (T->lchild)

{

if(T->lchild->data == e.data) dest = T;

}

//根结点有右孩子

if (T->rchild)

{

if(T->rchild->data == e.data) dest = T;

}

//当前根结点不是目标结点的双亲,搜索左/右子树

if (dest == NULL)

{

//在左子树中寻找目标结点e的双亲

if((dest = Parent(T->lchild, e)) != NULL) ;

//否则，在右子树中寻找目标结点e的双亲

else if((dest = Parent(T->rchild, e)) == NULL) dest = NULL;

} //end of if(dest==NULL):必须放于else if (T->lchild || T->rchild )内部

//否则,将引起Segmentation Fault

}

return dest;

}

BiTNode \* LeftChild(BiTree T, BiTNode e)

{

BiTNode \* dest = NULL; //目标结点

//二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，定位结点e,返回左孩子指针

if((dest = Locate(T, e)) != NULL) dest = dest->lchild;

else dest = NULL;

return dest;

}

BiTNode \* RightChild(BiTree T, BiTNode e)

{

BiTNode \* dest = NULL; //目标结点

//二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，定位结点e,返回右孩子指针

if((dest = Locate(T, e)) != NULL) dest = dest->rchild;

else dest = NULL;

return dest;

}

BiTNode \* LeftSibling(BiTree T, BiTNode e)

{

BiTNode \* dest = NULL; //目标结点

BiTNode \* parent = NULL; //目标结点的双亲结点

//二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，定位结点e，返回左兄弟指针

if((dest = Locate(T, e)) != NULL)

{

//定位结点e的双亲结点

parent = Parent(T, e);

if(parent) //具有双亲结点

{ //结点e为左孩子，不存在左兄弟

if(dest == parent->lchild) dest = NULL;

//否则，返回结点e双亲的左孩子，即为e的左兄弟

else dest = parent->lchild;

}

}

else dest = NULL;

return dest;

}

BiTNode \* RightSibling(BiTree T, BiTNode e)

{

BiTNode \* dest = NULL; //目标结点

BiTNode \* parent = NULL; //目标结点的双亲结点

//二叉树不存在/空,dest为NULL

if (T == NULL || BiTreeEmpty(T) == TRUE) dest = NULL;

//否则，定位结点e,返回右兄弟指针

if((dest = Locate(T, e)) != NULL)

{

//定位结点e的双亲结点

parent = Parent(T, e);

if(parent) //具有双亲结点

{ //结点e为右孩子，不存在右兄弟

if(dest == parent->rchild) dest = NULL;

//否则，返回结点e双亲的右孩子，即为e的右兄弟

else dest = parent->rchild;

}

}

else dest = NULL;

return dest;

}

Status InsertChild(BiTree T, BiTNode p, int LR, BiTree c)

{

BiTNode \* dest = NULL; //目标结点

//二叉树不存在/空,ERROR

if (T == NULL || BiTreeEmpty(T) == TRUE) return ERROR;

//二叉树c不存在/空/右子树不为空,ERROR

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

//否则，定位结点p

if((dest = Locate(T, p)) != NULL)

{

switch (LR)

{ //将c插入为p的左子树

case 0:

//p的左子树作为c的右子树

c->rchild = dest->lchild;

//插入c

dest->lchild = c;

return OK;

//将c插入为p的右子树

case 1:

//p的右子树作为c的右子树

c->rchild = dest->rchild;

//插入c

dest->rchild = c;

return OK;

default:

return ERROR;

}

}

else return ERROR;

}

Status DeleteChild(BiTree T, BiTNode p, int LR)

{

BiTNode \* dest = NULL; //目标结点

//二叉树不存在/空,ERROR

if (T == NULL || BiTreeEmpty(T) == TRUE) return ERROR;

//否则，定位结点p

if((dest = Locate(T, p)) != NULL)

{

switch (LR)

{

case 0:

//删除左子树

dest->lchild = NULL;

return OK;

case 1:

//删除右子树

dest->rchild = NULL;

return OK;

default:

return ERROR;

}

}

else return ERROR;

}

Status PreOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

{

//二叉树不存在/空,遍历结束

if (T == NULL || BiTreeEmpty(T) == TRUE) return OK;

//遍历成功

if (visit(T))

if(PreOrderTraverse(T->lchild, visit))

if(PreOrderTraverse(T->rchild, visit))

return OK;

//遍历失败

return ERROR;

}

Status InOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

{

//二叉树不存在/空,遍历结束

if (T == NULL || BiTreeEmpty(T) == TRUE) return OK;

//遍历成功

if(InOrderTraverse(T->lchild, visit))

if (visit(T))

if(InOrderTraverse(T->rchild, visit))

return OK;

//遍历失败

return ERROR;

}

Status PostOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

{

//二叉树不存在/空,遍历结束

if (T == NULL || BiTreeEmpty(T) == TRUE) return OK;

//遍历成功

if(PostOrderTraverse(T->lchild, visit))

if(PostOrderTraverse(T->rchild, visit))

if (visit(T))

return OK;

//遍历失败

return ERROR;

}

Status LevelOrderTraverse(BiTree T, Status (\*visit)(BiTNode \* e))

{

BiTNode \* node;

Queue queue;

//二叉树不存在,ERROR

if (T == NULL) return ERROR;

//二叉树为空,遍历结束

if (BiTreeEmpty(T) == TRUE) return OK;

//初始化队列

InitQueue(&queue);

//根结点入列

EnQueue(&queue,T);

//利用队列数据结构,实现广度搜索算法

while(!QueueEmpty(queue))

{ //队尾结点出列

DeQueue(&queue, &node);

//访问队尾结点

visit(node);

//将队尾结点的左/右孩子入列

if(node->lchild) EnQueue(&queue,node->lchild);

if(node->rchild) EnQueue(&queue,node->rchild);

}

//销毁队列

DestroyQueue(&queue);

return OK;

}

Status Visit (BiTNode \* e)

{ //结点不存在,ERROR

if (e == NULL) return ERROR;

printf("%c", e->data);

return OK;

}

Status RTPrint(BiTree T, int instance)

{

//二叉树不存在,ERROR

if (T == NULL) return ERROR;

//二叉树为空,ERROR

if (BiTreeEmpty(T) == TRUE) return ERROR;

//打印右子树

RTPrint(T->rchild,instance + 5);

//打印根结点

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

{

printf(" ");

}

printf("%5c\n",T->data);

//打印左子树

RTPrint(T->lchild,instance + 5);

return OK;

}

Status GLPrint(BiTree T)

{

//二叉树不存在,ERROR

if (T == NULL) return ERROR;

//二叉树为空,ERROR

if (BiTreeEmpty(T) == TRUE) return ERROR;

//打印根结点

printf("%c",T->data);

if(T->lchild || T->rchild)

{

printf("(");

//打印左子树

GLPrint(T->lchild);

if(T->rchild)

printf(",");

//打印右子树

GLPrint(T->rchild);

printf(")");

}

return OK;

}

Status LoadData (BiTree \* T)

{

int bitree\_id; //读取二叉树编号

int \* bitree\_num = (int \*)malloc(sizeof(int));//读取二叉树长度

char \* presequence; //读取的二叉树先序序列

char \* insequence; //读取的二叉树中序序列

char end[] = {'\0'}; //串尾结束符

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","r")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//读取数据

while (TRUE)

{

//将每个二叉树的第一个数据(编号)读取至bitree\_id变量

fread(&bitree\_id, sizeof(int), 1, fp);

//将每个二叉树的第二个数据(二叉树结点总数)读取至bitree\_length变量

fread(bitree\_num, sizeof(int), 1, fp);

//构造二叉树的前序序列与中序序列

presequence = (char \*)malloc( (\*bitree\_num + 1) \* sizeof(char));

insequence = (char \*)malloc( (\*bitree\_num + 1) \* sizeof(char));

//从文件中读取前序序列与中序序列

if (!fread(presequence, sizeof(char), \*bitree\_num, fp)) break; //若读取完毕，退出循环

if (!fread(insequence, sizeof(char), \*bitree\_num, fp)) break; //若读取完毕，退出循环

//添加'\0'，形成序列字符串

strcpy(presequence + \*bitree\_num, end), strcpy(insequence + \*bitree\_num, end);

//根据前序序列与中序序列创建二叉树

CreateBiTree(T + bitree\_id - 1, presequence, insequence);

}

fclose(fp);

return OK;

}

Status SaveData (BiTree \* T, Status (\*visit)(BiTNode \* e))

{

int bitree\_id; //用于保存二叉树编号

int \* bitree\_num = (int \*)malloc(sizeof(int)); //用于保存二叉树结点总数

BiTree ptr; //用于遍历所有二叉树

//以覆盖原有数据的形式，重定向标准输出流至文件"output.txt"

if (freopen("output.txt","w+", stdout) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//保存数据

//遍历所有二叉树并保存其前序序列与中序序列

for (bitree\_id = 1, ptr = \*T;bitree\_id <= TREE\_INIT\_SIZE; bitree\_id++, ptr = (\*(T + bitree\_id - 1)))

{

//当前二叉树存在且空，则保存当前二叉树

if (ptr != NULL && BiTreeEmpty(ptr) == FALSE)

{

//每个二叉树的第一个数据用于保存二叉树ID号

fwrite(&bitree\_id, sizeof(int), 1, stdout);

//每个二叉树的第二个数据用于保存二叉树结点总数

\*bitree\_num = 0;

NodeNum(ptr, bitree\_num);

fwrite(bitree\_num, sizeof(int), 1, stdout);

//前序遍历二叉树(标准输出流已重定向)

PreOrderTraverse(ptr, visit);

//中序遍历二叉树(标准输出流已重定向)

InOrderTraverse(ptr, visit);

}

}

return OK;

}

char \*myStrncpy(char \*dest, const char \*src, size\_t n)

{

int size = sizeof(char)\*(n+1);

char \*tmp = (char\*)malloc(size); // 开辟大小为n+1的临时内存tmp

if(tmp)

{

memset(tmp, '\0', size); // 将内存初始化为空字符

memcpy(tmp, src, size-1); // 将src的前n个字节拷贝到tmp

memcpy(dest, tmp, size); // 将临时空间tmp的内容拷贝到dest

free(tmp); // 释放内存

return dest;

}

else

{

return NULL;

}

}

Status NodeNum(BiTree T, int \*num)

{

//二叉树不存在,ERROR

if (T == NULL) return ERROR;

//二叉树为空,ERROR

if (BiTreeEmpty(T) == TRUE) return ERROR;

//递归求结点总数

(\*num)++ ;

NodeNum(T->lchild, num) ;

NodeNum(T->rchild, num) ;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 说明：利用队列,实现二叉树中部分函数(如层序遍历算法)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InitQueue (Queue \* Q)

{

//分配定长字节内存单元，并进行成功性检查

Q->base = (BiTNode \*\*)malloc(MAXQSIZE \* sizeof(BiTNode \*));

if (! Q->base) return OVERFLOW;

//初始化

Q->rear = Q->front = 0;

return OK;

}

Status DestroyQueue (Queue \* Q)

{

//队列不存在

if (! Q->base) return ERROR;

//队列存在，函数正确执行

free(Q->base);

//将base重置为null，front与rear重置为0

Q->base = NULL;

Q->rear = Q->front = 0;

return OK;

}

Status QueueEmpty (Queue Q)

{

//队列不存在

if (! Q.base) return ERROR;

//队列存在，函数正确执行

return Q.rear == Q.front ? TRUE : FALSE;

}

Status EnQueue (Queue \* Q, BiTNode \* e)

{

//队列不存在

if (! Q->base) return ERROR;

//队列满

if ( (Q->rear + 1) % MAXQSIZE == Q->front ) return ERROR;

//队列存在且未满

Q->base[Q->rear] = e;

//队列尾下标自增

Q->rear = (Q->rear + 1) % MAXQSIZE;

return OK;

}

Status DeQueue (Queue \* Q, BiTNode \* \* e)

{

//队列不存在

if (! Q->base) return ERROR;

//队列为空

if (QueueEmpty(\*Q)) return ERROR;

//队列存在且非空

(\*e) = Q->base[Q->front];

Q->front = (Q->front + 1) % MAXQSIZE;

return OK;

}

(4)main.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验五：二叉链存储二叉树主程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Tree.h"

int main (void)

{

int bitree\_id = 1; //二叉树ID号,初始化为1，表示默认操作第一个二叉树

int max\_bitree\_id = TREE\_INIT\_SIZE; //最大ID号

ElemType input = '\0'; //用户输入变量

int LR; //插入/删除子树的方向 以及 选择功能编号

BiTNode bitree\_node; //用于存放各个功能函数输入参数以及反馈的数据元素

BiTNode \* bitree\_pnode; //用于存放各个功能函数输入参数以及反馈的数据元素

char presequence[TREE\_INIT\_SIZE]; //用于存放用户输入的前序序列

char insequence[TREE\_INIT\_SIZE]; //用于存放用户输入的中序序列

Status (\*visit) (BiTNode \*) = Visit;//访问操作函数指针

//为TREE\_INIT\_SIZE个二叉树根结点分配合适内存单元

BiTree \* multi\_bitree = (BiTree \*)malloc(TREE\_INIT\_SIZE \* sizeof(BiTree));

if (! multi\_bitree) exit(OVERFLOW); //内存空间不足，退出系统

//初始化所有二叉树

for (bitree\_id = 1;bitree\_id <= TREE\_INIT\_SIZE;bitree\_id++)

{

InitBiTree(multi\_bitree + bitree\_id - 1);

}

//读取文本数据

LoadData(multi\_bitree);

//当前操作二叉树重置为第一个二叉树

bitree\_id = 1;

//打印系统菜单界面

printf ("\t\t欢迎使用多二叉树系统\n\n");

while (TRUE)

{

//打印系统菜单界面

printf("\*\*\*\*\* 当前二叉树编号: No.%d \*\*\*\*\*\n", bitree\_id);

printf ("1:创建二叉树\t");

printf ("2:清空二叉树\n");

printf ("3:查询空状态\t");

printf ("4:查询深度\n");

printf ("5:查询根结点\t");

printf ("6:查询结点\n");

printf ("7:查询结点值\t");

printf ("8:二叉树结点赋值\n");

printf ("9:查询双亲\n");

printf ("10:查询左孩子\t");

printf ("11:查询右孩子\n");

printf ("12:查询左兄弟\t");

printf ("13:查询右兄弟\n");

printf ("14:插入子树\n");

printf ("15:删除子树\t");

printf ("16:前序遍历\n");

printf ("17:中序遍历\t");

printf ("18:后序遍历\n");

printf ("19:层序遍历\t");

printf ("20:凹入表打印\n");

printf ("21:广义表打印\t");

printf ("22:切换二叉树(ID:1-%d)\n", max\_bitree\_id);

printf ("23:退出系统\n\n");

//提示用户选择功能

printf ("请选择功能:");

//用户选择事件

scanf ("%d", &LR);

//处理用户选择事件

switch (LR)

{

case 1: //创建二叉树

printf("请输入前序序列:\n");

scanf("%99s", presequence);

printf("请输入中序序列:\n");

scanf("%99s", insequence);

if (CreateBiTree(multi\_bitree + bitree\_id - 1, presequence, insequence) == OK)

printf("创建 No.%d 成功\n", bitree\_id);

else printf("创建 No.%d 失败\n", bitree\_id);

printf("\n");

system("pause");

system("CLS");

break;

case 2: //清空二叉树

if (ClearBiTree (\*(multi\_bitree + bitree\_id - 1)) == OK)

printf("清空 No.%d 成功\n", bitree\_id);

else printf("清空 No.%d 失败\n", bitree\_id);

printf("\n");

system("pause");

system("CLS");

break;

case 3: //查询空状态

if (BiTreeEmpty (\*(multi\_bitree + bitree\_id - 1)) == TRUE)

printf("No.%d 为空树\n", bitree\_id);

else printf("No.%d 不为空树\n", bitree\_id);

printf("\n");

system("pause");

system("CLS");

break;

case 4: //查询深度

if ((input = BiTreeDepth(\*(multi\_bitree + bitree\_id - 1))) != ERROR)

printf("No.%d 深度:%d\n", bitree\_id, input);

else printf("No.%d 为空树\n", bitree\_id);

printf("\n");

system("pause");

system("CLS");

break;

case 5: //查询根结点

if ((bitree\_pnode = Root(\*(multi\_bitree + bitree\_id - 1))) != NULL)

printf("No.%d 根结点:%c\n", bitree\_id, bitree\_pnode->data);

else printf("No.%d 不存在\n", bitree\_id);

printf("\n");

system("pause");

system("CLS");

break;

case 6: //查询结点

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

if ((bitree\_pnode = Locate(\*(multi\_bitree + bitree\_id - 1), bitree\_node)) != NULL)

printf("No.%d 存在结点:%c\n", bitree\_id, bitree\_pnode->data);

else printf("No.%d 不存在结点:%c\n", bitree\_id, bitree\_node.data);

printf("\n");

system("pause");

system("CLS");

break;

case 7: //查询结点值

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

if ((input = Value(\*(multi\_bitree + bitree\_id - 1), bitree\_node)) != '\0')

printf("结点 %c 值:%c\n", bitree\_node.data, input);

else printf("不存在结点:%c\n", bitree\_node.data);

printf("\n");

system("pause");

system("CLS");

break;

case 8: //二叉树结点赋值

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

getchar();

printf("请输入修改新值:");

scanf("%c", &input);

if ((Assign(\*(multi\_bitree + bitree\_id - 1), bitree\_node, input)) == OK)

printf("结点 %c 值:%c\n", bitree\_node.data, input);

else printf("不存在结点:%c\n", bitree\_node.data);

printf("\n");

system("pause");

system("CLS");

break;

case 9: //查询双亲

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

if ((bitree\_pnode = Parent(\*(multi\_bitree + bitree\_id - 1), bitree\_node)) != NULL)

printf("结点 %c 双亲:%c\n", bitree\_node.data, bitree\_pnode->data);

else printf("不存在结点:%c\n", bitree\_node.data);

printf("\n");

system("pause");

system("CLS");

break;

case 10: //查询左孩子

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

if ((bitree\_pnode = LeftChild(\*(multi\_bitree + bitree\_id - 1), bitree\_node)) != NULL)

printf("结点 %c 左孩子:%c\n", bitree\_node.data, bitree\_pnode->data);

else printf("查询失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 11: //查询右孩子

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

if ((bitree\_pnode = RightChild(\*(multi\_bitree + bitree\_id - 1), bitree\_node)) != NULL)

printf("结点 %c 右孩子:%c\n", bitree\_node.data, bitree\_pnode->data);

else printf("查询失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 12: //查询左兄弟

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

if ((bitree\_pnode = LeftSibling(\*(multi\_bitree + bitree\_id - 1), bitree\_node)) != NULL)

printf("结点 %c 左兄弟:%c\n", bitree\_node.data, bitree\_pnode->data);

else printf("查询失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 13: //查询右兄弟

getchar();

printf("请输入目标结点:");

scanf("%c", &(bitree\_node.data));

if ((bitree\_pnode = RightSibling(\*(multi\_bitree + bitree\_id - 1), bitree\_node)) != NULL)

printf("结点 %c 右兄弟:%c\n", bitree\_node.data, bitree\_pnode->data);

else printf("查询失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 14: //插入子树

getchar();

printf("请输入目标结点");

scanf("%c", &(bitree\_node.data));

do

{

printf("请选择插入方向(0-左子树,1-右子树)");

scanf("%d", &LR);

}

while(LR != 0 && LR != 1);

printf("请输入新二叉树的前序序列( 无右子树 ):\n");

scanf("%99s", presequence);

printf("请输入新二叉树的中序序列( 无右子树 ):\n");

scanf("%99s", insequence);

if (CreateBiTree(&bitree\_pnode, presequence, insequence) == OK)

{

if ((InsertChild(\*(multi\_bitree + bitree\_id - 1), bitree\_node, LR, bitree\_pnode)) == OK)

printf("插入成功\n");

else printf("插入失败\n");

}

else printf("插入失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 15: //删除子树

getchar();

printf("请输入目标结点");

scanf("%c", &(bitree\_node.data));

do

{

printf("请选择删除方向(0-左子树,1-右子树)");

scanf("%d", &LR);

}

while(LR != 0 && LR != 1);

if ((DeleteChild(\*(multi\_bitree + bitree\_id - 1), bitree\_node, LR)) == OK)

printf("删除成功\n");

else printf("删除失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 16: //前序遍历

if ((PreOrderTraverse(\*(multi\_bitree + bitree\_id - 1), visit)) == OK)

printf("\n");

else printf("遍历失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 17: //中序遍历

if ((InOrderTraverse(\*(multi\_bitree + bitree\_id - 1), visit)) == OK)

printf("\n");

else printf("遍历失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 18: //后序遍历

if ((PostOrderTraverse(\*(multi\_bitree + bitree\_id - 1), visit)) == OK)

printf("\n");

else printf("遍历失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 19: //层序遍历

if ((LevelOrderTraverse(\*(multi\_bitree + bitree\_id - 1), visit)) == OK)

printf("\n");

else printf("遍历失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 20: //凹入表打印

if ((RTPrint(\*(multi\_bitree + bitree\_id - 1), 0)) == OK)

printf("\n");

else printf("打印失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 21: //广义表打印

if ((GLPrint(\*(multi\_bitree + bitree\_id - 1))) == OK)

printf("\n");

else printf("打印失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 22: //切换二叉树

//循环用于限定用户输入为1-TREE\_INIT\_SIZE

do

{

printf("请输入切换目标二叉树ID号:");

//户键入目标二叉树ID

scanf("%d", &bitree\_id);

}

while(bitree\_id < 1 || bitree\_id > TREE\_INIT\_SIZE);

printf("已切换至 No.%d 号二叉树", bitree\_id);

printf("\n");

system("pause");

system("CLS");

break;

case 23: //退出系统

//保存多线二叉树

SaveData(multi\_bitree, visit);

//销毁多二叉树

for(bitree\_id = 1;bitree\_id <= TREE\_INIT\_SIZE;bitree\_id++)

{

DestroyBiTree(\*(multi\_bitree + bitree\_id - 1));

}

//销毁所有根结点

free(multi\_bitree);

exit (OK);

printf("\n");

system("pause");

system("CLS");

break;

default:

printf ("选择功能错误\n");

printf("\n");

system("pause");

system("CLS");

break;

}

printf ("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

}

(5)assert\_test.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验五：二叉链存储二叉树断言测试

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <assert.h>

#include "Tree.h"

#undef NDEBUG //开启断言测试

#define TEST\_LENGTH 50 //定义二叉树测试最大长度为50

int main (void)

{

BiTree test\_bitree = NULL; //测试用二叉树

BiTNode test\_node; //测试用结点

BiTNode \* test\_pnode, \*test\_pnode2; //测试用结点指针

ElemType test\_data; //测试用结点数据域

Status (\*visit)(BiTNode \*) = Visit; //访问操作函数指针

char \* test\_presequence, \*test\_insequence; //遍历序列

test\_presequence = (char \*)malloc( TEST\_LENGTH \* sizeof(char));//前序序列

test\_insequence = (char \*)malloc( TEST\_LENGTH \* sizeof(char)); //中序序列

//初始化前/中序序列

strcpy(test\_presequence, "ABDGLKMUVNWCEHOPIQFJRSXYZT");

strcpy(test\_insequence, "LGUMVKWNDBAOHPEQICFJXSZYRT");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 断言测试 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 不存在二叉树 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\* 基础模块 \*\*\*\*\*\*\*\*\*\*/

//查询不存在的二叉树深度

assert(BiTreeDepth(test\_bitree) == ERROR);

//查询不存在的二叉树的根结点

assert(Root(test\_bitree) == NULL);

/\*\*\*\*\*\*\*\*\*\* 编辑与查询模块 \*\*\*\*\*\*\*\*\*\*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 不存在二叉树 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

//定位A-Z

for (char i = 'A'; i <= 'C';i++)

{ //构造Locate函数的第二个结点参数

test\_node.data = i;

//取得Locate函数返回值

test\_pnode = Locate(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("Locate %c Failed\n", i);

}

//求A-G值

for (char i = 'A'; i <= 'C';i++)

{ //构造Value函数的第二个结点参数

test\_node.data = i;

//取得Value函数返回值

test\_data = Value(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_data == ERROR);

printf("Get %c Value Failed\n", i);

}

//将A-G的值全部修改为a-z

for (char i = 'A', j = 'a'; j <= 'c';i++, j++)

{ //构造Assign函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(Assign(test\_bitree, test\_node, j) == ERROR);

printf("%c Assign Failed\n", test\_node.data);

}

//查询双亲结点

for (char i = 'B'; i <= 'C';i++)

{ //构造Parent函数的第二个结点参数

test\_node.data = i;

//取得Parent函数返回值

//断言声明及信息提示

assert(Parent(test\_bitree, test\_node) == NULL);

printf("No %c Parent\n", i);

}

//查询孩子结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftChild与RightChild函数的第二个结点参数

test\_node.data = i;

//取得LeftChild函数返回值

test\_pnode = LeftChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftChild\n", i);

//取得RightChild函数返回值

test\_pnode = RightChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightChild\n", i);

}

//查询兄弟结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftSibling与RightSibling函数的第二个结点参数

test\_node.data = i;

//取得LeftSibling函数返回值

test\_pnode = LeftSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftSibling\n", i);

//取得RightSibling函数返回值

test\_pnode = RightSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightSibling\n", i);

}

system("pause");

/\*\*\*\*\*\*\*\*\*\* 遍历与打印模块 \*\*\*\*\*\*\*\*\*\*/

//先序遍历不存在的二叉树

assert(PreOrderTraverse(test\_bitree, visit) == OK);

//中序遍历不存在的二叉树

assert(InOrderTraverse(test\_bitree, visit) == OK);

//后序遍历不存在的二叉树

assert(PostOrderTraverse(test\_bitree, visit) == OK);

//层序遍历不存在的二叉树

assert(LevelOrderTraverse(test\_bitree, visit) == ERROR);

//以凹入表形式打印不存在的二叉树

assert(RTPrint(test\_bitree, 0) == ERROR);

//以广义表形式打印不存在的二叉树

assert(GLPrint(test\_bitree) == ERROR);

//销毁不存在的二叉树

assert(DestroyBiTree(test\_bitree) == ERROR);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 空二叉树 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\* 基础模块 \*\*\*\*\*\*\*\*\*\*/

//构造空二叉树

assert(InitBiTree(&test\_bitree) == OK);

//空二叉树检查

assert(BiTreeEmpty(test\_bitree) == TRUE);

//查询空二叉树深度

assert(BiTreeDepth(test\_bitree) == 0);

//查询空二叉树的根结点

assert(Root(test\_bitree) == test\_bitree);

/\*\*\*\*\*\*\*\*\*\* 编辑与查询模块 \*\*\*\*\*\*\*\*\*\*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 空二叉树 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

//定位A-Z

for (char i = 'A'; i <= 'C';i++)

{ //构造Locate函数的第二个结点参数

test\_node.data = i;

//取得Locate函数返回值

test\_pnode = Locate(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("Locate %c Failed\n", i);

}

//求A-G值

for (char i = 'A'; i <= 'C';i++)

{ //构造Value函数的第二个结点参数

test\_node.data = i;

//取得Value函数返回值

test\_data = Value(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_data == ERROR);

printf("Get %c Value Failed\n", i);

}

//将A-G的值全部修改为a-z

for (char i = 'A', j = 'a'; j <= 'c';i++, j++)

{ //构造Assign函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(Assign(test\_bitree, test\_node, j) == ERROR);

printf("%c Assign Failed\n", test\_node.data);

}

//查询双亲结点

for (char i = 'B'; i <= 'C';i++)

{ //构造Parent函数的第二个结点参数

test\_node.data = i;

//取得Parent函数返回值

//断言声明及信息提示

assert(Parent(test\_bitree, test\_node) == NULL);

printf("No %c Parent\n", i);

}

//查询孩子结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftChild与RightChild函数的第二个结点参数

test\_node.data = i;

//取得LeftChild函数返回值

test\_pnode = LeftChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftChild\n", i);

//取得RightChild函数返回值

test\_pnode = RightChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightChild\n", i);

}

//查询兄弟结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftSibling与RightSibling函数的第二个结点参数

test\_node.data = i;

//取得LeftSibling函数返回值

test\_pnode = LeftSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftSibling\n", i);

//取得RightSibling函数返回值

test\_pnode = RightSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightSibling\n", i);

}

system("pause");

/\*\*\*\*\*\*\*\*\*\* 遍历与打印模块 \*\*\*\*\*\*\*\*\*\*/

//先序遍历空二叉树

assert(PreOrderTraverse(test\_bitree, visit) == OK);

//中序遍历空二叉树

assert(InOrderTraverse(test\_bitree, visit) == OK);

//后序遍历空二叉树

assert(PostOrderTraverse(test\_bitree, visit) == OK);

//层序遍历空二叉树

assert(LevelOrderTraverse(test\_bitree, visit) == OK);

//以凹入表形式打印空二叉树

assert(RTPrint(test\_bitree, 0) == ERROR);

//以广义表形式打印空二叉树

assert(GLPrint(test\_bitree) == ERROR);

//销毁空二叉树

assert(DestroyBiTree(test\_bitree) == OK);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 字母二叉树 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\* 基础模块 \*\*\*\*\*\*\*\*\*\*/

//根据前序序列和中序序列创建二叉树

assert(CreateBiTree(&test\_bitree, test\_presequence, test\_insequence) == OK);

//查询二叉树深度

assert(BiTreeDepth(test\_bitree) == 8);

//查询二叉树的根结点

assert(Root(test\_bitree) == test\_bitree);

assert(Root(test\_bitree->lchild) == test\_bitree->lchild);

assert(Root(test\_bitree->rchild) == test\_bitree->rchild);

/\*\*\*\*\*\*\*\*\*\* 编辑与查询模块 \*\*\*\*\*\*\*\*\*\*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 字母二叉树 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

//定位A-Z

for (char i = 'A'; i <= 'Z';i++)

{ //构造Locate函数的第二个结点参数

test\_node.data = i;

//取得Locate函数返回值

test\_pnode = Locate(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode->data == i);

printf("Locate %c Success\n", i);

}

system("pause");

//求A-G值

for (char i = 'A'; i <= 'Z';i++)

{ //构造Value函数的第二个结点参数

test\_node.data = i;

//取得Value函数返回值

test\_data = Value(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_data == i);

printf("%c Value: %c\n", i, test\_data);

}

system("pause");

//查询双亲结点

for (char i = 'B'; i <= 'Z';i++)

{ //构造Parent函数的第二个结点参数

test\_node.data = i;

//取得Parent函数返回值

test\_pnode = Parent(test\_bitree, test\_node);

//断言声明及信息提示

//以下多个if语句目的:防止Segmentation Fault

if (test\_pnode->lchild != NULL && test\_pnode->rchild == NULL)

{ //双亲结点只存在左孩子

assert(test\_pnode->lchild->data == i);

}

else if (test\_pnode->lchild == NULL && test\_pnode->rchild != NULL)

{ //双亲结点只存在右孩子

assert(test\_pnode->rchild->data == i);

}

else if (test\_pnode->lchild != NULL && test\_pnode->rchild != NULL)

{ //双亲结点存在左与右孩子

assert(test\_pnode->lchild->data == i || test\_pnode->rchild->data == i);

}

printf("%c Parent: %c\n", i, test\_pnode->data);

}

system("pause");

//查询孩子结点

for (char i = 'A'; i <= 'Z';i++)

{ //构造LeftChild与RightChild函数的第二个结点参数

test\_node.data = i;

//定位test\_node

test\_pnode2 = Locate(test\_bitree, test\_node);

//取得LeftChild函数返回值

test\_pnode = LeftChild(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node左孩子data域等于test\_pnode的data域

if (test\_pnode && test\_pnode2 && test\_pnode2->lchild)

{

assert(test\_pnode->data == test\_pnode2->lchild->data);

printf("%c LeftChild: %c\n", i, test\_pnode->data);

}

//取得RightChild函数返回值

test\_pnode = RightChild(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node右孩子data域等于test\_pnode的data域

if (test\_pnode && test\_pnode2 && test\_pnode2->rchild)

{

assert(test\_pnode->data == test\_pnode2->rchild->data);

printf("%c RightChild: %c\n", i, test\_pnode->data);

}

}

system("pause");

//查询兄弟结点

for (char i = 'A'; i <= 'Z';i++)

{ //构造LeftSibling与RightSibling函数的第二个结点参数

test\_node.data = i;

//定位test\_node的双亲结点

test\_pnode2 = Parent(test\_bitree, test\_node);

//取得LeftSibling函数返回值

test\_pnode = LeftSibling(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node左兄弟test\_pnode的data域等于test\_pnode2的左孩子data域

if (test\_pnode && test\_pnode2 && test\_pnode2->lchild)

{

assert(test\_pnode->data == test\_pnode2->lchild->data);

printf("%c LeftSibling: %c\n", i, test\_pnode->data);

}

//取得RightSibling函数返回值

test\_pnode = RightSibling(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node右兄弟test\_pnode的data域等于test\_pnode2的右孩子data域

if (test\_pnode && test\_pnode2 && test\_pnode2->rchild)

{

assert(test\_pnode->data == test\_pnode2->rchild->data);

printf("%c RightSibling: %c\n", i, test\_pnode->data);

}

}

system("pause");

/\*\*\*\*\*\*\*\*\*\* 遍历与打印 \*\*\*\*\*\*\*\*\*\*/

//先序遍历二叉树

printf("PreOrderTraverse\n");

assert(PreOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//中序遍历二叉树

printf("InOrderTraverse\n");

assert(InOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//后序遍历二叉树

printf("PostOrderTraverse\n");

assert(PostOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//层序遍历二叉树

printf("LevelOrderTraverse\n");

assert(LevelOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//以凹入表形式打印二叉树

printf("Recessed Table Print\n");

assert(RTPrint(test\_bitree, 0) == OK);

printf("\n");

//以广义表形式打印二叉树

printf("Generalized List Print\n");

assert(GLPrint(test\_bitree) == OK);

printf("\n\n");

system("pause");

/\*\*\*\*\*\*\*\*\*\* 赋值函数Assign \*\*\*\*\*\*\*\*\*\*/

//将A-G的值全部修改为a-z

for (char i = 'A', j = 'a'; j <= 'z';i++, j++)

{ //构造Assign函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(Assign(test\_bitree, test\_node, j) == OK);

printf("%c Assign: %c\n", test\_node.data, j);

}

//以凹入表形式打印修改后的二叉树

printf("Recessed Table Print\n");

assert(RTPrint(test\_bitree, 0) == OK);

printf("\n");

/\*\*\*\*\*\*\*\*\*\* 插入与删除函数InsertChild与DeleteChild \*\*\*\*\*\*\*\*\*\*/

//再次初始化前/中序序列，测试插入与删除模块

strcpy(test\_presequence, "12345");

strcpy(test\_insequence, "24531");

//根据前序序列与中序序列创建另外两个无右子树的二叉树12345

CreateBiTree(&test\_pnode, test\_presequence, test\_insequence);

CreateBiTree(&test\_pnode2, test\_presequence, test\_insequence);

//测试设计:为二叉树中每个结点都进行0与1(左右子树)的插入

// 同时也为二叉树中每个结点都进行0与1(左右子树)的删除

//插入模块:自根结点向子叶结点插入

for (char i = 'a'; i <= 'z';i++)

{ //构造InsertChild函数的第二个结点参数

test\_node.data = i;

//根据前序序列与中序序列创建另外两个无右子树的二叉树12345

CreateBiTree(&test\_pnode, test\_presequence, test\_insequence);

CreateBiTree(&test\_pnode2, test\_presequence, test\_insequence);

//断言声明及信息提示

assert(InsertChild(test\_bitree, test\_node, 0, test\_pnode) == OK);

assert(InsertChild(test\_bitree, test\_node, 1, test\_pnode2) == OK);

//打印修改后的二叉树前序序列

printf("Insert %c's Child Success : InOrderTraverse\n", i);

assert(InOrderTraverse(test\_bitree, visit) == OK);

printf("\n\n");

}

system("pause");

printf("\n\n");

//删除模块:自子叶结点向根结点删除

for (char i = 'z'; i >= 'a';i--)

{ //构造DeleteChild函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(DeleteChild(test\_bitree, test\_node, 0) == OK);

assert(DeleteChild(test\_bitree, test\_node, 1) == OK);

//打印修改后的二叉树前序序列

printf("Delete %c's Child Success : InOrderTraverse\n", i);

assert(InOrderTraverse(test\_bitree, visit) == OK);

printf("\n\n");

}

//销毁二叉树

assert(DestroyBiTree(test\_bitree) == OK);

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 断言测试成功! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

system("pause");

return OK;

}

(6)Makefile

DT\_LAB\_05.exe : Tree.o main.o

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -o DT\_LAB\_05.exe Tree.o main.o

Tree.o : Tree.c Tree.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -c Tree.c

main.o : main.c Tree.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -g -c main.c

5.3.2 测试部分

本次测试采用断言测试，测试主程序及源代码见附件AssertTest.exe及程序清单assert\_test.c。

一、测试用例为一个含有26个结点，深度为8的字母二叉树，

前序序列为"ABDGLKMUVNWCEHOPIQFJRSXYZT"，

中序序列为"LGUMVKWNDBAOHPEQICFJXSZYRT"，

结构如下图所示：



图5-2 测试用例：字母二叉树

二、插入模块测试用例为一个结点总数为5 ，深度为5的数字字符二叉树，

前序序列为"12345"，

中序序列为 "24531"，

结构如下图所示：



图5-3 测试用例：待插入数字字符二叉树

三、测试流程如下：

1.不存在二叉树

**/\*\*\*\*\*\*\*\*\*\* 基础模块 \*\*\*\*\*\*\*\*\*\*/**

//查询不存在的二叉树深度

assert(BiTreeDepth(test\_bitree) == ERROR);

//查询不存在的二叉树的根结点

assert(Root(test\_bitree) == NULL);

**/\*\*\*\*\*\*\*\*\*\* 编辑与查询模块 \*\*\*\*\*\*\*\*\*\*/**

//定位A-Z

for (char i = 'A'; i <= 'C';i++)

{ //构造Locate函数的第二个结点参数

test\_node.data = i;

//取得Locate函数返回值

test\_pnode = Locate(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("Locate %c Failed\n", i);

}

//求A-G值

for (char i = 'A'; i <= 'C';i++)

{ //构造Value函数的第二个结点参数

test\_node.data = i;

//取得Value函数返回值

test\_data = Value(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_data == ERROR);

printf("Get %c Value Failed\n", i);

}

//将A-G的值全部修改为a-z

for (char i = 'A', j = 'a'; j <= 'c';i++, j++)

{ //构造Assign函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(Assign(test\_bitree, test\_node, j) == ERROR);

printf("%c Assign Failed\n", test\_node.data);

}

//查询双亲结点

for (char i = 'B'; i <= 'C';i++)

{ //构造Parent函数的第二个结点参数

test\_node.data = i;

//取得Parent函数返回值

//断言声明及信息提示

assert(Parent(test\_bitree, test\_node) == NULL);

printf("No %c Parent\n", i);

}

//查询孩子结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftChild与RightChild函数的第二个结点参数

test\_node.data = i;

//取得LeftChild函数返回值

test\_pnode = LeftChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftChild\n", i);

//取得RightChild函数返回值

test\_pnode = RightChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightChild\n", i);

}

//查询兄弟结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftSibling与RightSibling函数的第二个结点参数

test\_node.data = i;

//取得LeftSibling函数返回值

test\_pnode = LeftSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftSibling\n", i);

//取得RightSibling函数返回值

test\_pnode = RightSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightSibling\n", i);

}

system("pause");

**/\*\*\*\*\*\*\*\*\*\* 遍历与打印模块 \*\*\*\*\*\*\*\*\*\*/**

//先序遍历不存在的二叉树

assert(PreOrderTraverse(test\_bitree, visit) == OK);

//中序遍历不存在的二叉树

assert(InOrderTraverse(test\_bitree, visit) == OK);

//后序遍历不存在的二叉树

assert(PostOrderTraverse(test\_bitree, visit) == OK);

//层序遍历不存在的二叉树

assert(LevelOrderTraverse(test\_bitree, visit) == ERROR);

//以凹入表形式打印不存在的二叉树

assert(RTPrint(test\_bitree, 0) == ERROR);

//以广义表形式打印不存在的二叉树

assert(GLPrint(test\_bitree) == ERROR);

//销毁不存在的二叉树

assert(DestroyBiTree(test\_bitree) == ERROR);

2.空二叉树

**/\*\*\*\*\*\*\*\*\*\* 基础模块 \*\*\*\*\*\*\*\*\*\*/**

//构造空二叉树

assert(InitBiTree(&test\_bitree) == OK);

//空二叉树检查

assert(BiTreeEmpty(test\_bitree) == TRUE);

//查询空二叉树深度

assert(BiTreeDepth(test\_bitree) == 0);

//查询空二叉树的根结点

assert(Root(test\_bitree) == test\_bitree);

**/\*\*\*\*\*\*\*\*\*\* 编辑与查询模块 \*\*\*\*\*\*\*\*\*\*/**

//定位A-Z

for (char i = 'A'; i <= 'C';i++)

{ //构造Locate函数的第二个结点参数

test\_node.data = i;

//取得Locate函数返回值

test\_pnode = Locate(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("Locate %c Failed\n", i);

}

//求A-G值

for (char i = 'A'; i <= 'C';i++)

{ //构造Value函数的第二个结点参数

test\_node.data = i;

//取得Value函数返回值

test\_data = Value(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_data == ERROR);

printf("Get %c Value Failed\n", i);

}

//将A-G的值全部修改为a-z

for (char i = 'A', j = 'a'; j <= 'c';i++, j++)

{ //构造Assign函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(Assign(test\_bitree, test\_node, j) == ERROR);

printf("%c Assign Failed\n", test\_node.data);

}

//查询双亲结点

for (char i = 'B'; i <= 'C';i++)

{ //构造Parent函数的第二个结点参数

test\_node.data = i;

//取得Parent函数返回值

//断言声明及信息提示

assert(Parent(test\_bitree, test\_node) == NULL);

printf("No %c Parent\n", i);

}

//查询孩子结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftChild与RightChild函数的第二个结点参数

test\_node.data = i;

//取得LeftChild函数返回值

test\_pnode = LeftChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftChild\n", i);

//取得RightChild函数返回值

test\_pnode = RightChild(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightChild\n", i);

}

//查询兄弟结点

for (char i = 'A'; i <= 'C';i++)

{ //构造LeftSibling与RightSibling函数的第二个结点参数

test\_node.data = i;

//取得LeftSibling函数返回值

test\_pnode = LeftSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c LeftSibling\n", i);

//取得RightSibling函数返回值

test\_pnode = RightSibling(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode == NULL);

printf("No %c RightSibling\n", i);

}

system("pause");

**/\*\*\*\*\*\*\*\*\*\* 遍历与打印模块 \*\*\*\*\*\*\*\*\*\*/**

//先序遍历空二叉树

assert(PreOrderTraverse(test\_bitree, visit) == OK);

//中序遍历空二叉树

assert(InOrderTraverse(test\_bitree, visit) == OK);

//后序遍历空二叉树

assert(PostOrderTraverse(test\_bitree, visit) == OK);

//层序遍历空二叉树

assert(LevelOrderTraverse(test\_bitree, visit) == OK);

//以凹入表形式打印空二叉树

assert(RTPrint(test\_bitree, 0) == ERROR);

//以广义表形式打印空二叉树

assert(GLPrint(test\_bitree) == ERROR);

//销毁空二叉树

assert(DestroyBiTree(test\_bitree) == OK);

3.字母二叉树(中途插入数字字符二叉树)

**/\*\*\*\*\*\*\*\*\*\* 基础模块 \*\*\*\*\*\*\*\*\*\*/**

//根据前序序列和中序序列创建二叉树

assert(CreateBiTree(&test\_bitree, test\_presequence, test\_insequence) == OK);

//查询二叉树深度

assert(BiTreeDepth(test\_bitree) == 8);

//查询二叉树的根结点

assert(Root(test\_bitree) == test\_bitree);

assert(Root(test\_bitree->lchild) == test\_bitree->lchild);

assert(Root(test\_bitree->rchild) == test\_bitree->rchild);

**/\*\*\*\*\*\*\*\*\*\* 编辑与查询模块 \*\*\*\*\*\*\*\*\*\*/**

//定位A-Z

for (char i = 'A'; i <= 'Z';i++)

{ //构造Locate函数的第二个结点参数

test\_node.data = i;

//取得Locate函数返回值

test\_pnode = Locate(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_pnode->data == i);

printf("Locate %c Success\n", i);

}

system("pause");

//求A-G值

for (char i = 'A'; i <= 'Z';i++)

{ //构造Value函数的第二个结点参数

test\_node.data = i;

//取得Value函数返回值

test\_data = Value(test\_bitree, test\_node);

//断言声明及信息提示

assert(test\_data == i);

printf("%c Value: %c\n", i, test\_data);

}

system("pause");

//查询双亲结点

for (char i = 'B'; i <= 'Z';i++)

{ //构造Parent函数的第二个结点参数

test\_node.data = i;

//取得Parent函数返回值

test\_pnode = Parent(test\_bitree, test\_node);

//断言声明及信息提示

//以下多个if语句目的:防止Segmentation Fault

if (test\_pnode->lchild != NULL && test\_pnode->rchild == NULL)

{ //双亲结点只存在左孩子

assert(test\_pnode->lchild->data == i);

}

else if (test\_pnode->lchild == NULL && test\_pnode->rchild != NULL)

{ //双亲结点只存在右孩子

assert(test\_pnode->rchild->data == i);

}

else if (test\_pnode->lchild != NULL && test\_pnode->rchild != NULL)

{ //双亲结点存在左与右孩子

assert(test\_pnode->lchild->data == i || test\_pnode->rchild->data == i);

}

printf("%c Parent: %c\n", i, test\_pnode->data);

}

system("pause");

//查询孩子结点

for (char i = 'A'; i <= 'Z';i++)

{ //构造LeftChild与RightChild函数的第二个结点参数

test\_node.data = i;

//定位test\_node

test\_pnode2 = Locate(test\_bitree, test\_node);

//取得LeftChild函数返回值

test\_pnode = LeftChild(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node左孩子data域等于test\_pnode的data域

if (test\_pnode && test\_pnode2 && test\_pnode2->lchild)

{

assert(test\_pnode->data == test\_pnode2->lchild->data);

printf("%c LeftChild: %c\n", i, test\_pnode->data);

}

//取得RightChild函数返回值

test\_pnode = RightChild(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node右孩子data域等于test\_pnode的data域

if (test\_pnode && test\_pnode2 && test\_pnode2->rchild)

{

assert(test\_pnode->data == test\_pnode2->rchild->data);

printf("%c RightChild: %c\n", i, test\_pnode->data);

}

}

system("pause");

//查询兄弟结点

for (char i = 'A'; i <= 'Z';i++)

{ //构造LeftSibling与RightSibling函数的第二个结点参数

test\_node.data = i;

//定位test\_node的双亲结点

test\_pnode2 = Parent(test\_bitree, test\_node);

//取得LeftSibling函数返回值

test\_pnode = LeftSibling(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node左兄弟test\_pnode的data域等于test\_pnode2的左孩子data域

if (test\_pnode && test\_pnode2 && test\_pnode2->lchild)

{

assert(test\_pnode->data == test\_pnode2->lchild->data);

printf("%c LeftSibling: %c\n", i, test\_pnode->data);

}

//取得RightSibling函数返回值

test\_pnode = RightSibling(test\_bitree, test\_node);

//断言声明及信息提示

//test\_node右兄弟test\_pnode的data域等于test\_pnode2的右孩子data域

if (test\_pnode && test\_pnode2 && test\_pnode2->rchild)

{

assert(test\_pnode->data == test\_pnode2->rchild->data);

printf("%c RightSibling: %c\n", i, test\_pnode->data);

}

}

system("pause");

**/\*\*\*\*\*\*\*\*\*\* 遍历与打印 \*\*\*\*\*\*\*\*\*\*/**

//先序遍历二叉树

printf("PreOrderTraverse\n");

assert(PreOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//中序遍历二叉树

printf("InOrderTraverse\n");

assert(InOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//后序遍历二叉树

printf("PostOrderTraverse\n");

assert(PostOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//层序遍历二叉树

printf("LevelOrderTraverse\n");

assert(LevelOrderTraverse(test\_bitree, visit) == OK);

printf("\n");

//以凹入表形式打印二叉树

printf("Recessed Table Print\n");

assert(RTPrint(test\_bitree, 0) == OK);

printf("\n");

//以广义表形式打印二叉树

printf("Generalized List Print\n");

assert(GLPrint(test\_bitree) == OK);

printf("\n\n");

system("pause");

**/\*\*\*\*\*\*\*\*\*\* 赋值函数Assign \*\*\*\*\*\*\*\*\*\*/**

//将A-G的值全部修改为a-z

for (char i = 'A', j = 'a'; j <= 'z';i++, j++)

{ //构造Assign函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(Assign(test\_bitree, test\_node, j) == OK);

printf("%c Assign: %c\n", test\_node.data, j);

}

//以凹入表形式打印修改后的二叉树

printf("Recessed Table Print\n");

assert(RTPrint(test\_bitree, 0) == OK);

printf("\n");

**/\*\*\*\*\*\*\*\*\*\* 插入与删除函数InsertChild与DeleteChild \*\*\*\*\*\*\*\*\*\*/**

//再次初始化前/中序序列，测试插入与删除模块

strcpy(test\_presequence, "12345");

strcpy(test\_insequence, "24531");

//根据前序序列与中序序列创建另外两个无右子树的二叉树12345

CreateBiTree(&test\_pnode, test\_presequence, test\_insequence);

CreateBiTree(&test\_pnode2, test\_presequence, test\_insequence);

//测试设计:为二叉树中每个结点都进行0与1(左右子树)的插入

// 同时也为二叉树中每个结点都进行0与1(左右子树)的删除

//插入模块:自根结点向子叶结点插入

for (char i = 'a'; i <= 'z';i++)

{ //构造InsertChild函数的第二个结点参数

test\_node.data = i;

//根据前序序列与中序序列创建另外两个无右子树的二叉树12345

CreateBiTree(&test\_pnode, test\_presequence, test\_insequence);

CreateBiTree(&test\_pnode2, test\_presequence, test\_insequence);

//断言声明及信息提示

assert(InsertChild(test\_bitree, test\_node, 0, test\_pnode) == OK);

assert(InsertChild(test\_bitree, test\_node, 1, test\_pnode2) == OK);

//打印修改后的二叉树前序序列

printf("Insert %c's Child Success : InOrderTraverse\n", i);

assert(InOrderTraverse(test\_bitree, visit) == OK);

printf("\n\n");

}

system("pause");

printf("\n\n");

//删除模块:自子叶结点向根结点删除

for (char i = 'z'; i >= 'a';i--)

{ //构造DeleteChild函数的第二个结点参数

test\_node.data = i;

//断言声明及信息提示

assert(DeleteChild(test\_bitree, test\_node, 0) == OK);

assert(DeleteChild(test\_bitree, test\_node, 1) == OK);

//打印修改后的二叉树前序序列

printf("Delete %c's Child Success : InOrderTraverse\n", i);

assert(InOrderTraverse(test\_bitree, visit) == OK);

printf("\n\n");

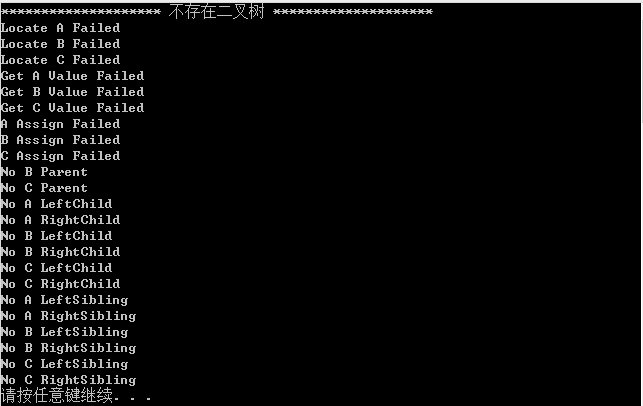
}

//销毁二叉树

assert(DestroyBiTree(test\_bitree) == OK);

四、测试结果如以下测试图所示，所有断言成功，程序正常退出，结果符合预期。

1.不存在二叉树

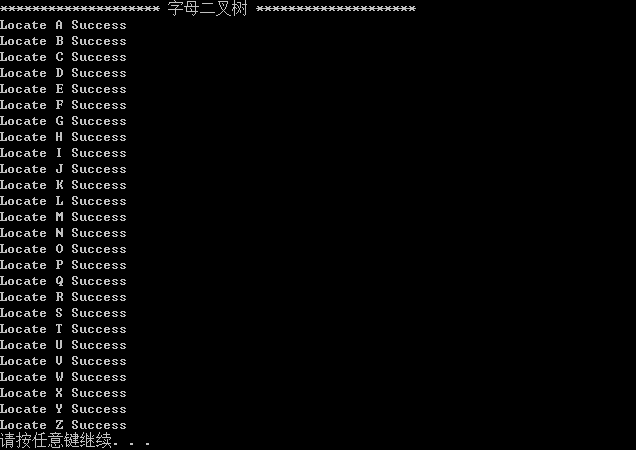


2.空二叉树



3.字母二叉树

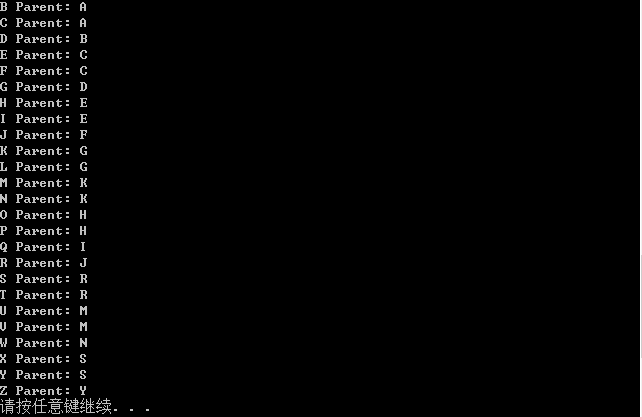
定位函数Locate



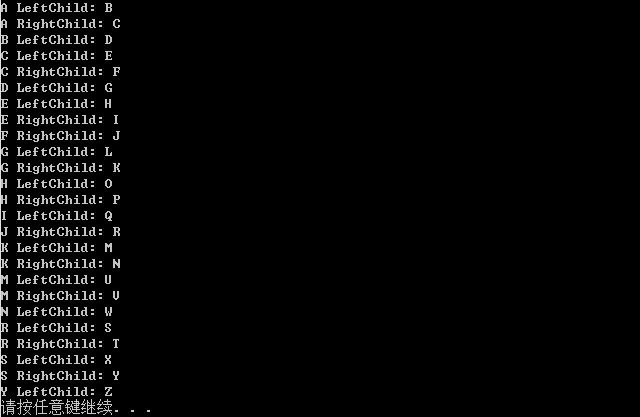
查询Data域函数Value



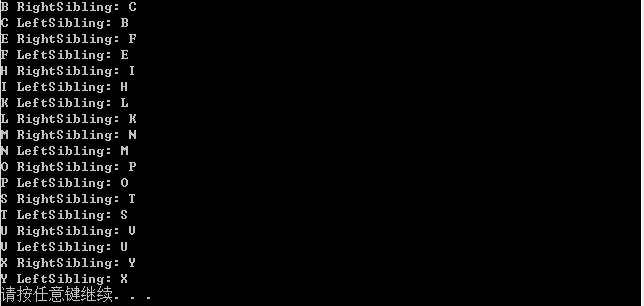
查询双亲函数Parent



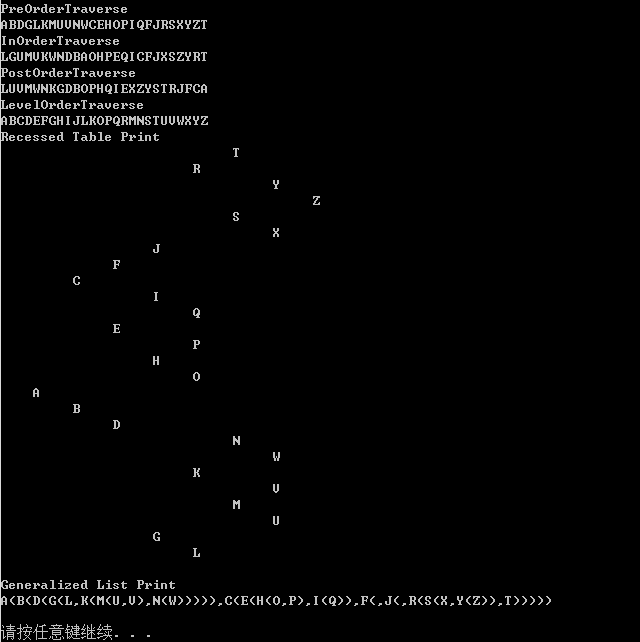
查询孩子函数LeftChild与RightChild



查询兄弟函数LeftSibling与RightSibling



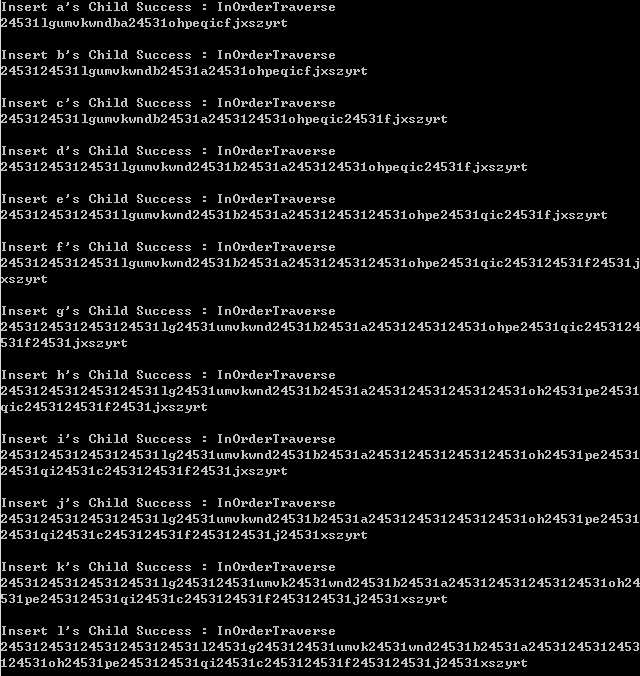
前/中/后序遍历，凹入表/广义表打印



赋值函数Assign



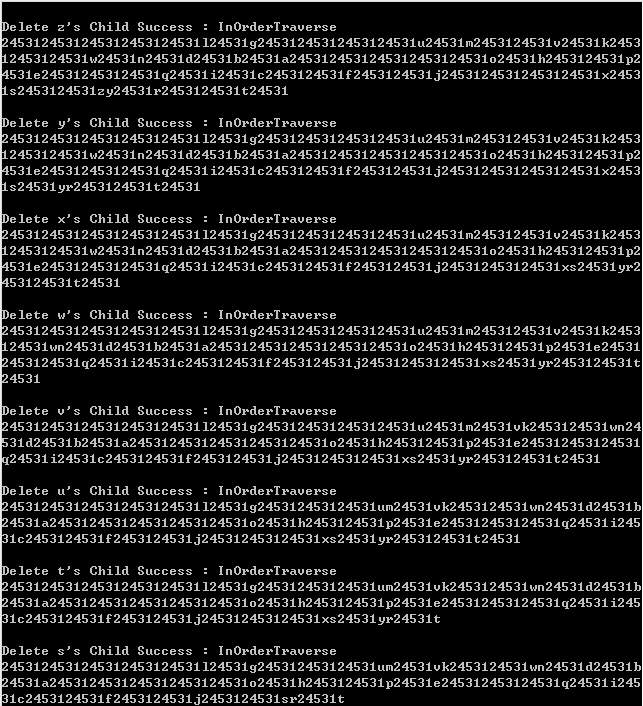
插入函数InsertChild



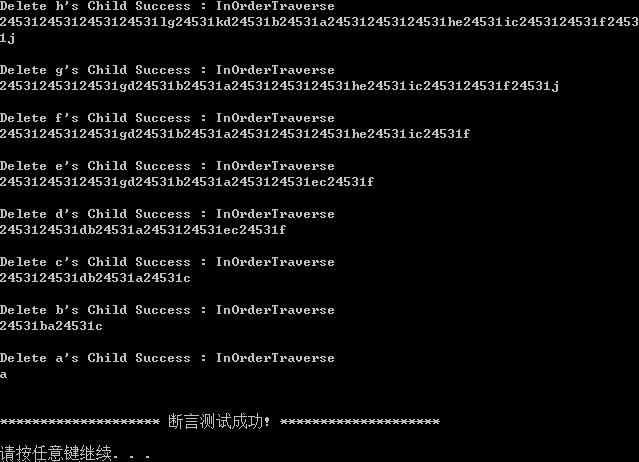




删除函数DeleteChild







五、测试过程中未解决重大缺陷

当创建链表时，若用户输入的前序序列与中序序列不匹配，则会导致程序直接退出。

5.4 实验小结

一、此次实验的断言测试代码达422行，看似繁琐而无用，但这些代码帮助我排除了许多Segmentation Fault，极大地加快了Debug指针方面Bug的速度。测试用例为含A-Z 26个字母结点的8层二叉树，极具代表性，且进行了 不存在二叉树 与 空二叉树的模块检查，使得最终的系统具有较好的健壮性和容错性。这一切都是编写断言测试代码带来的良好结果。先写好测试代码，再写功能代码，未尝不是一种有效的编程选择；

二、在寻找如何监视变量时，通过大致浏览一些GDB手册，发现GDB有太多神奇的用法，是集成开发环境中的套壳调试器(尽管也是GDB)无法实现的，再一次感受到了GDB的强大魅力，是个值得细细学习的强大工具；

三、通过实现二叉链存储二叉树的基本操作，加强了对二叉树的理解和使用，希望能为以后的数据结构学习带来帮助；

四、通过大量使用递归操作，对递归算法有了更深层次的理解。理解了通过函数调用栈、队列实现深度优先搜索算法(如前序遍历算法)与广度优先搜索算法(如层序遍历算法)。

**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  (40分) | 程序规范  (10分) | 报告内容  (40分) | 报告规范  (10分) | 逾 期  扣 分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |

# 6基于邻接表实现图的基本运算

6.1 问题描述

6.1.1 实验任务

采用邻接表表示法（Representation），实现无向图的基本运算。构造成具有功能菜单的系统，采用文件形式保存和读取无向图。实现创建图、销毁图、定位顶点、查找第一邻接顶点、查找下一邻接顶点、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、输出等基本功能。

6.1.2 实验目的

（1）加深对图的概念、基本运算的理解；

（2）熟练掌握图的逻辑结构与物理结构的关系；

（3）熟练掌握图的基本运算在不同的物理结构下的实现。

6.1.3 实现算法

（1）基础模块

创建无向图、销毁无向图。

（2）查询模块

定位顶点、查找第一邻接顶点、查找下一邻接顶点；

深度优先遍历、广度优先遍历、输出打印。

（3）编辑模块

插入顶点、删除顶点、插入弧、删除弧。

（4）数据模块

自动读取、自动保存。

6.2 基于邻接表的图设计

6.2.1 系统总体设计

本系统为邻接表存储无向图使用系统。本系统可自动保存与读取数据。

本系统具有以下几大功能：

一、基础功能

创建无向图、销毁无向图；

自动读取、自动保存。

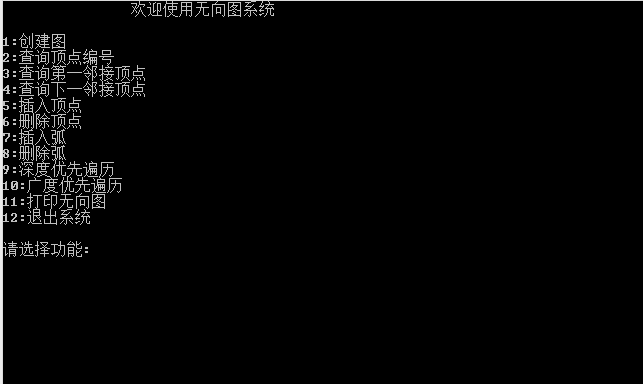
二、查询功能

定位顶点、查找第一邻接顶点、查找下一邻接顶点；

深度优先遍历、广度优先遍历、输出打印。

三、编辑功能

插入顶点、删除顶点、插入弧、删除弧。



6.2.2 有关常量和类型定义

1.所有实验公有部分：见程序清单DataStructure.h

2.本实验常量和类型定义

//系统可操作图的顶点最大值

#define MAX\_VERTEX\_NUM 20

//访问标志数组

Boolean visited[MAX\_VERTEX\_NUM];

//弧的相关信息

typedef char InfoType;

//顶点的相关信息

typedef char VertexType; //顶点名称

typedef struct ArcNode {

int adjvex; //该弧所指向的顶点的位置

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

InfoType info[6]; //该弧的相关信息

} ArcNode;

typedef struct VNode {

VertexType data; //顶点相关信息

ArcNode \* firstarc; //指向第一条依附该顶点的弧的指针

} VNode, AdjList[MAX\_VERTEX\_NUM];

typedef struct {

AdjList vertices;

int vexnum, arcnum; //图的当前顶点数和弧数

} ALGraph;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 说明：利用队列, 实现广度优先遍历算法

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//最大队列长度

#define MAXQSIZE 100

//定义顺序存储循环队列数据结构

typedef struct Queue

{

VNode \* \* base; //初始化的动态分配存储空间, 在队列构造之前和销毁之后, base的值为NULL

int front; //头指针, 若队列不空, 指向队列首元素

int rear; //尾指针, 若队列不空, 指向队列尾元素的下一个位置

} Queue;

6.2.3 算法设计

1.CreateCraph(&G,V,VR)

/\*\*

\* 建立邻接存储的无向图

\* @param G 无向图

\* @call InsertVex

\* @call InsertArc

\* @return OK/ERROR

\*/

//建立顶点表

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

{

//插入顶点

InsertVex(G, j);

}

//向顶点表中插入弧结点,建立完整的邻接表

for (k = 0;k < arcnum;k++)

{

//插入弧

InsertArc(G, i, j);

}

//处理顶点表中空位置

for(k = G->vexnum;k < MAX\_VERTEX\_NUM;k++)

{

G->vertices[k].data = '\0';

G->vertices[k].firstarc = NULL;

}

T(n) = O(3n) , S(n) = O(n)

2.DestroyCraph(&G)

/\*\*

\* 销魂图G

\* 初始条件:图G存在

\* @param G 无向图

\* @return OK/ERROR

\*/

//循环遍历并释放邻接表的内存空间

for(i = 0;i < G->vexnum;i++)

{

//当前顶点邻接链表为空，无需遍历

if (G->vertices[i].firstarc == NULL) continue;

//否则遍历邻接链表

for (priorArc = G->vertices[i].firstarc, currentArc = priorArc->nextarc;

priorArc != NULL;

priorArc = currentArc, currentArc = currentArc->nextarc)

{

free (priorArc);

if (currentArc == NULL) break; //遍历结束

}

}

//释放顶点表以及图

free(G);

T(n) = O(n2) , S(n) = O(n)

3.LocateVex(G,u)

/\*\*

\* 定位目标顶点的位置，返回位置信息

\* 初始条件:图G存在

\* @param G 无向图

\* @param u 目标顶点的信息

\* @return u的位置信息(编号)/-1

\*/

//遍历邻接表，定位目标顶点

for(i = 0;i <= G->vexnum;i++)

{

if(G->vertices[i].data == u) return i;

}

T(n) = O(n) , S(n) = O(n)

4.FirstAdjVex(&G, v)

/\*\*

\* 返回顶点v的第一个邻接顶点

\* 初始条件:图G存在

\* @param G 无向图

\* @param v 目标顶点

\* @call LocateVex

\* @return v的邻接顶点地址/NULL

\*/

//,定位目标顶点v

pos = LocateVex(G, v))

//定位v成功，寻找其邻接顶点

//将邻接顶点的位置序号存入pos

pos = G->vertices[pos].firstarc->adjvex;

//根据pos返回邻接顶点地址

return G->vertices + pos;

T(n) = O(n) , S(n) = O(n)

5.NextAdjVex(&G, v, w)

/\*\*

\* 返回v的(相对于w)下一个邻接顶点

\* 初始条件 图G存在;w是v的邻接顶点

\* @param G 无向图

\* @param v 目标顶点

\* @param w 目标顶点的邻接顶点

\* @call LocateVex

\* @call FirstAdjVex

\* @return v的下一邻接顶点地址/NULL

\*/

//定位目标顶点v

pos = LocateVex(G, v);

//遍历顶点v的邻接链表，查找v相对于w的下一邻接顶点

for(arc = G->vertices[pos].firstarc;arc != NULL;arc = arc->nextarc)

{

if(arc->adjvex == LocateVex(G, w) && arc->nextarc != NULL)

{ //定位顶点w，且v相对于w的下一邻接顶点存在

pos = arc->nextarc->adjvex; //定位v相对于w的下一邻接顶点

return G->vertices + pos; //返回下一邻接顶点的地址

}

}

T(n) = O(2n) , S(n) = O(n)

6.InsertVex(&G,v)

/\*\*

\* 插入新顶点v

\* 初始条件:图G存在

\* @param G 无向图

\* @param v 插入顶点

\* @return OK/ERROR

\*/

//建立顶点

//读入顶点名称

G->vertices[G->vexnum].data = v;

//将依附于顶点的第一条弧置为NULL

G->vertices[G->vexnum].firstarc = NULL;

//顶点数加1

G->vexnum ++;

T(n) = O(1) , S(n) = O(1)

7.DeleteVex(&G,v)

/\*\*

\* 删除顶点v

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param v 删除顶点

\* @call LocateVex

\* @return OK/ERROR

\*/

//定位目标顶点v

pos = LocateVex(G, v))

//从头遍历邻接链表

for (arc = G->vertices[pos].firstarc;arc != NULL;arc = arc->nextarc)

{ //删除弧<w,v>

priorArc->nextarc = currentArc->nextarc;

free(currentArc);

//删除弧<v, w>

//弧数自减

G->arcnum--;

}

T(n) = O(2n) , S(n) = O(n)

8.InsertArc(&G,v,w)

/\*\*

\* 插入弧<v, w>与<w, v>

\* 初始条件:图G存在

\* @param G 无向图

\* @param v 弧的顶点

\* @param w 弧的顶点

\* @call LocateVex

\* @return OK/ERROR

\*/

//插入弧<v, w>

//将新弧所指顶点位置置为wpos

arc->adjvex = wpos;

//插入操作

arc->nextarc = G->vertices[vpos].firstarc;

G->vertices[vpos].firstarc = arc;

//该图为无向图，故同时插入弧<w, v>

//弧<w, v>指向顶点编号vpos

arc->adjvex = vpos;

//插入操作

arc->nextarc = G->vertices[wpos].firstarc;

G->vertices[wpos].firstarc = arc;

//弧数自加

G->arcnum++;

T(n) = O(1) , S(n) = O(1)

9.DeleteArc(&G,v,w)

/\*\*

\* 删除弧<v, w>与<w, v>

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param v 弧的顶点

\* @param w 弧的顶点

\* @call LocateVex

\* @return OK/ERROR

\*/

//在v的邻接链表中删除除弧<v, w>

priorArc->nextarc = currentArc->nextarc;

//在w的邻接链表中删除弧<w, v>

priorArc->nextarc = currentArc->nextarc;

T(n) = O(2n) , S(n) = O(n)

10.DFSTraverse(G,visit())

/\*\*

\* 深度遍历无向图G

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param visit 访问操作函数指针

\* @call LocateVex

\* @call FirstAdjVex

\* @call NextAdjVex

\* @call Visit

\* @return OK/ERROR

\*/

//对尚未访问的顶点调用DFS

for(vpos = 0;vpos < G->vexnum;vpos++)

{

if(!visited[vpos]) DFS(G, G->vertices + vpos, visit);

}

T(n) = O(n3) , S(n) = O(n2)

11.BFSTraverse(G,visit())

/\*\*

\* 广度遍历无向图G

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param visit 访问操作函数指针

\* @call LocateVex

\* @call FirstAdjVex

\* @call NextAdjVex

\* @call Visit

\* @call Queue相关函数

\* @return OK/ERROR

\*/

for(vpos = 0;vpos < G->vexnum;vpos++)

{

//尚未访问v

if(!visited[vpos])

{

visited[vpos] = TRUE;

v = G->vertices + vpos;

visit(v);

//v入队列

EnQueue(&Q, v);

while(!QueueEmpty(Q))

{

//队首元素出队并置为u

DeQueue(&Q, &u);

//访问u的所有邻接顶点并将它们送入队列尾部

for (w = FirstAdjVex(G, u->data);w != NULL;w = NextAdjVex(G, u->data, w->data))

{

//w为u的尚未访问的邻接顶点

if(!visited[wpos])

{

visited[wpos] = TRUE;

visit(w);

EnQueue(&Q, w);

} //if

} //for

} //while

} //if

}

T(n) = O(n2) , S(n) = O(n)

6.3 基于邻接表的图实现与测试

6.3.1 实现部分

1.编程环境

编辑器：Sublime Text 3

编译器：MinGW-gcc 4.9.1(C99标准)

调试器：MinGW-gdb 7.8

2.文件调用关系



3.函数调用关系

 4.程序清单

1. DataStructure.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验六：邻接表存储图

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef DATASTRUCTURE\_H\_INCLUDED

#define DATASTRUCTURE\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

//自定义宏

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

//Status是函数的类型,其值是函数结果状态代码

typedef int Status;

//将Boolean定义为int型

typedef int Boolean;

//将ElemType定义为char型

typedef int ElemType;

#endif /\*DATASTRUCTURE\_H\_INCLUDED\*/

1. Graph.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验六：邻接表存储图抽象数据类型定义

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef GRAPH\_H\_INCLUDED

#define GRAPH\_H\_INCLUDED

#include "DataStructure.h" //标准库;公有类型与变量定义

//系统可操作图的顶点最大值

#define MAX\_VERTEX\_NUM 20

//访问标志数组

Boolean visited[MAX\_VERTEX\_NUM];

//弧的相关信息

typedef char InfoType;

//顶点的相关信息

typedef char VertexType; //顶点名称

typedef struct ArcNode {

int adjvex; //该弧所指向的顶点的位置

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

InfoType info[6]; //该弧的相关信息

} ArcNode;

typedef struct VNode {

VertexType data; //顶点相关信息

ArcNode \* firstarc; //指向第一条依附该顶点的弧的指针

} VNode, AdjList[MAX\_VERTEX\_NUM];

typedef struct {

AdjList vertices;

int vexnum, arcnum; //图的当前顶点数和弧数

} ALGraph;

/\*\*

\* 建立邻接存储的无向图

\* @param G 无向图

\* @call InsertVex

\* @call InsertArc

\* @return OK/ERROR

\*/

Status CreateGraph(ALGraph \* G);

/\*\*

\* 销魂图G

\* 初始条件:图G存在

\* @param G 无向图

\* @return OK/ERROR

\*/

Status DestroyGraph(ALGraph \* G);

/\*\*

\* 定位目标顶点的位置，返回位置信息

\* 初始条件:图G存在

\* @param G 无向图

\* @param u 目标顶点的信息

\* @return u的位置信息(编号)/-1

\*/

int LocateVex(ALGraph \* G, VertexType u);

/\*\*

\* 返回顶点v的第一个邻接顶点

\* 初始条件:图G存在

\* @param G 无向图

\* @param v 目标顶点

\* @call LocateVex

\* @return v的邻接顶点地址/NULL

\*/

VNode \* FirstAdjVex(ALGraph \* G, VertexType v);

/\*\*

\* 返回v的(相对于w)下一个邻接顶点

\* 初始条件 图G存在;w是v的邻接顶点

\* @param G 无向图

\* @param v 目标顶点

\* @param w 目标顶点的邻接顶点

\* @call LocateVex

\* @call FirstAdjVex

\* @return v的下一邻接顶点地址/NULL

\*/

VNode \* NextAdjVex(ALGraph \* G, VertexType v, VertexType w);

/\*\*

\* 插入新顶点v

\* 初始条件:图G存在

\* @param G 无向图

\* @param v 插入顶点

\* @return OK/ERROR

\*/

Status InsertVex(ALGraph \* G, VertexType v);

/\*\*

\* 删除顶点v

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param v 删除顶点

\* @call LocateVex

\* @return OK/ERROR

\*/

Status DeleteVex(ALGraph \* G, VertexType v);

/\*\*

\* 插入弧<v, w>与<w, v>

\* 初始条件:图G存在

\* @param G 无向图

\* @param v 弧的顶点

\* @param w 弧的顶点

\* @call LocateVex

\* @return OK/ERROR

\*/

Status InsertArc(ALGraph \* G, VertexType v, VertexType w);

/\*\*

\* 删除弧<v, w>与<w, v>

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param v 弧的顶点

\* @param w 弧的顶点

\* @call LocateVex

\* @return OK/ERROR

\*/

Status DeleteArc(ALGraph \* G,VertexType v,VertexType w);

/\*\*

\* 深度遍历无向图G

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param visit 访问操作函数指针

\* @call LocateVex

\* @call FirstAdjVex

\* @call NextAdjVex

\* @call Visit

\* @return OK/ERROR

\*/

Status DFSTraverse(ALGraph \* G, Status (\*visit)(VNode \* v));

Status DFS(ALGraph \*G, VNode \* v, Status (\*visit)(VNode \* v));

/\*\*

\* 广度遍历无向图G

\* 初始条件:图G存在且不为空

\* @param G 无向图

\* @param visit 访问操作函数指针

\* @call LocateVex

\* @call FirstAdjVex

\* @call NextAdjVex

\* @call Visit

\* @call Queue相关函数

\* @return OK/ERROR

\*/

Status BFSTraverse(ALGraph \* G, Status (\*visit)(VNode \* v));

/\*\*

\* 输出顶点v的信息

\* @param v 图G中顶点

\* @return 操作结果状态Status OK/ERROR

\*/

Status Visit (VNode \* v);

/\*\*

\* 输出顶点v的邻接弧的信息

\* @param v 图G中顶点

\* @return 操作结果状态Status OK/ERROR

\*/

Status Print (VNode \* v);

/\*\*

\* 从output.txt读取数据

\* @param G 无向图

\* @call LocateVex

\* @call InsertVex

\* @call InsertArc

\* @return 操作结果状态Status OK/ERROR

\*/

Status LoadData (ALGraph \* G);

/\*\*

\* 保存数据至output.txt

\* @param G 无向图

\* @return 操作结果状态Status OK/ERROR

\*/

Status SaveData (ALGraph \* G);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 说明：利用队列, 实现广度优先遍历算法

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//最大队列长度

#define MAXQSIZE 100

//定义顺序存储循环队列数据结构

typedef struct Queue

{

VNode \* \* base; //初始化的动态分配存储空间, 在队列构造之前和销毁之后, base的值为NULL

int front; //头指针, 若队列不空, 指向队列首元素

int rear; //尾指针, 若队列不空, 指向队列尾元素的下一个位置

} Queue;

/\*\*

\* 构造一个空队列Q

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status InitQueue (Queue \* Q);

/\*\*

\* 销毁队列Q

\* 初始条件:队列Q已存在

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DestroyQueue (Queue \* Q);

/\*\*

\* 查询Q是否为空队列

\* 初始条件:队列Q已存在

\* @param Q 队列Q

\* @return 若Q为空队列, 则返回TRUE, 否则返回FALSE

\*/

Status QueueEmpty (Queue Q);

/\*\*

\* 插入元素v为新的队列尾元素

\* 初始条件:队列Q已存在

\* @param Q 队列地址值

\* @param v 插入数据元素值

\* @return 操作结果状态Status OK/ERROR

\*/

Status EnQueue (Queue \* Q, VNode \* v);

/\*\*

\* 删除队列Q的队列首元素

\* 初始条件：队列Q已存且非空

\* @param Q 队列地址值

\* @return 操作结果状态Status OK/ERROR

\*/

Status DeQueue (Queue \* Q, VNode \* \* v);

#endif /\*GRAPH\_H\_INCLUDED\*/

1. Graph.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验六：邻接表存储图基本操作实现

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Graph.h"

Status CreateGraph(ALGraph \* G)

{

int i,j; //顶点位置(编号)

int k; //循环变量

int vexnum, arcnum;//顶点总数和弧总数

printf("请输入\"顶点数 边数\":");

scanf("%d %d", &vexnum, &arcnum);

//初始化图的顶点数和边数

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

//建立顶点表

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

{

//输入顶点名称

printf("请输入 No.%d 顶点名称(单字符):", i);

scanf("%1s", &j );

//插入顶点

InsertVex(G, j);

}

//向顶点表中插入弧结点,建立完整的邻接表

for (k = 0;k < arcnum;k++)

{

//输入弧结点信息

printf("请输入顶点对(形式:v w):");

scanf("%1s %1s", &i, &j);

//插入弧

InsertArc(G, i, j);

}

//处理顶点表中空位置

for(k = G->vexnum;k < MAX\_VERTEX\_NUM;k++)

{

G->vertices[k].data = '\0';

G->vertices[k].firstarc = NULL;

}

return OK;

}

Status DestroyGraph(ALGraph \* G)

{

int i; //遍历顶点表的循环变量

ArcNode \* priorArc, \* currentArc;//遍历每个顶点的邻接链表的循环变量

//图不存在,ERROR

if (G == NULL) return ERROR;

//循环遍历并释放邻接表的内存空间

for(i = 0;i < G->vexnum;i++)

{

//当前顶点邻接链表为空，无需遍历

if (G->vertices[i].firstarc == NULL) continue;

//否则遍历邻接链表

for (priorArc = G->vertices[i].firstarc, currentArc = priorArc->nextarc;

priorArc != NULL;

priorArc = currentArc, currentArc = currentArc->nextarc)

{

free (priorArc);

if (currentArc == NULL) break; //遍历结束

}

}

//释放顶点表以及图

free(G);

return OK;

}

int LocateVex(ALGraph \*G, VertexType u)

{

int i; //遍历顶点表的循环变量

//图不存在或顶点不存在，返回-1编号

if(G == NULL) return -1;

//遍历邻接表，定位目标顶点

for(i = 0;i <= G->vexnum;i++)

{

if(G->vertices[i].data == u) return i;

}

//定位失败，返回-1编号

return -1;

}

VNode \* FirstAdjVex(ALGraph \* G, VertexType v)

{

int pos; //目标顶点在图中的位置序号

//图不存在，返回NULL

if(G == NULL) return NULL;

//否则,定位目标顶点v

if((pos = LocateVex(G, v)) != -1)

{ //定位v成功，寻找其邻接顶点

if(G->vertices[pos].firstarc != NULL)

{ //顶点v拥有邻接顶点

//将邻接顶点的位置序号存入pos

pos = G->vertices[pos].firstarc->adjvex;

//根据pos返回邻接顶点地址

return G->vertices + pos;

}

else

{ //顶点v没有邻接顶点

return NULL;

}

}

else

{ //定位失败

return NULL;

}

}

VNode \* NextAdjVex(ALGraph \* G, VertexType v, VertexType w)

{

int pos; //目标顶点在图中的位置序号

VNode \* vex; //存放目标顶点v的第一个邻接顶点

ArcNode \* arc; //遍历每个顶点的邻接链表的循环变量

//图不存在或，返回NULL

if(G == NULL) return NULL;

//否则,定位目标顶点v的第一个邻接顶点

if( (vex = FirstAdjVex(G, v)) != NULL)

{ //定位第一个邻接顶点成功

pos = LocateVex(G, v); //定位顶点v

//遍历顶点v的邻接链表，查找v相对于w的下一邻接顶点

for(arc = G->vertices[pos].firstarc;arc != NULL;arc = arc->nextarc)

{

if(arc->adjvex == LocateVex(G, w) && arc->nextarc != NULL)

{ //定位顶点w，且v相对于w的下一邻接顶点存在

pos = arc->nextarc->adjvex; //定位v相对于w的下一邻接顶点

return G->vertices + pos; //返回下一邻接顶点的地址

}

}

}

return NULL;

}

Status InsertVex(ALGraph \* G, VertexType v)

{

//图不存在，返回NULL

if(G == NULL) return ERROR;

//建立顶点

//读入顶点名称

G->vertices[G->vexnum].data = v;

//将依附于顶点的第一条弧置为NULL

G->vertices[G->vexnum].firstarc = NULL;

//顶点数加1

G->vexnum ++;

return OK;

}

Status DeleteVex(ALGraph \* G,VertexType v)

{

int pos1, pos2; //目标顶点及其邻接顶点的位序

ArcNode \* arc, \* priorArc, \*currentArc;//用以遍历邻接链表的指针

//图不存在或为空，ERROR

if(G == NULL || G->vexnum == 0) return ERROR;

//定位目标顶点v

if ((pos1 = LocateVex(G, v)) != -1)

{

//从头遍历邻接链表

for (arc = G->vertices[pos1].firstarc;arc != NULL;arc = arc->nextarc)

{

for(//第一个邻接结点位序 //删除邻接顶点的邻接链表中对应弧

pos2 = arc->adjvex , priorArc = G->vertices[pos2].firstarc,currentArc = priorArc;

currentArc != NULL;priorArc = currentArc,currentArc = currentArc->nextarc)

{

if (currentArc->adjvex == pos1)

{ //删除目标弧结点是否是邻接表首结点

if(currentArc == G->vertices[pos2].firstarc)

{

G->vertices[pos2].firstarc = G->vertices[pos2].firstarc->nextarc;

}

else

{

priorArc->nextarc = currentArc->nextarc;

free(currentArc);

}

//弧数自减

G->arcnum--;

break;

} //if

} //for

} //for

//从顶点表中删除顶点v

for(pos2 = pos1;pos2 < MAX\_VERTEX\_NUM - 1;pos2++)

{

G->vertices[pos2] = G->vertices[pos2 + 1];

}

G->vertices[MAX\_VERTEX\_NUM - 1].data = '\0';

G->vertices[MAX\_VERTEX\_NUM - 1].firstarc = NULL;

//顶点数自减

G->vexnum--;

return OK;

} //if

return ERROR;

}

Status InsertArc(ALGraph \* G,VertexType v,VertexType w)

{

ArcNode \* arc;

//弧端点在顶点表中的位序

int vpos, wpos;

if((vpos = LocateVex(G, v)) == -1) return ERROR;

if((wpos = LocateVex(G, w)) == -1) return ERROR;

//图不存在，ERROR

if(G == NULL) return ERROR;

//插入弧<v, w>

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

if (arc == NULL) return OVERFLOW;

//将新弧所指顶点位置置为wpos

arc->adjvex = wpos;

//建立该弧相关信息

arc->info[0] = '<';

arc->info[1] = G->vertices[vpos].data;

arc->info[2] = ',';

arc->info[3] = G->vertices[wpos].data;

arc->info[4] = '>';

arc->info[5] = '\0';

//插入操作

arc->nextarc = G->vertices[vpos].firstarc;

G->vertices[vpos].firstarc = arc;

//该图为无向图，故同时插入弧<w, v>

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

if (arc == NULL) return OVERFLOW;

//弧<w, v>指向顶点编号vpos

arc->adjvex = vpos;

//建立该弧相关信息

arc->info[0] = '<';

arc->info[1] = G->vertices[wpos].data;

arc->info[2] = ',';

arc->info[3] = G->vertices[vpos].data;

arc->info[4] = '>';

arc->info[5] = '\0';

//插入操作

arc->nextarc = G->vertices[wpos].firstarc;

G->vertices[wpos].firstarc = arc;

//弧数自加

G->arcnum++;

return OK;

}

Status DeleteArc(ALGraph \* G,VertexType v,VertexType w)

{

//用以遍历邻接链表的指针

ArcNode \* priorArc, \* currentArc;

//弧端点在顶点表中的位序

int vpos, wpos;

if((vpos = LocateVex(G, v)) == -1) return ERROR;

if((wpos = LocateVex(G, w)) == -1) return ERROR;

//图不存在或为空，ERROR

if(G == NULL || G->vexnum == 0) return ERROR;

//删除弧<v, w>

for(priorArc = G->vertices[vpos].firstarc, currentArc = priorArc;

currentArc != NULL;priorArc = currentArc, currentArc = currentArc->nextarc)

{

if (currentArc->adjvex == wpos)

{ //目标弧结点为邻接表是否为首结点

if(currentArc == G->vertices[vpos].firstarc)

{

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

}

else

{

priorArc->nextarc = currentArc->nextarc;

}

//释放内存空间

free(currentArc);

break;

}

}

//删除弧<w, v>

for(priorArc = G->vertices[wpos].firstarc, currentArc = priorArc;

currentArc != NULL;priorArc = currentArc, currentArc = currentArc->nextarc)

{

if (currentArc->adjvex == vpos)

{ //目标弧结点为邻接表是否为首结点

if(currentArc == G->vertices[wpos].firstarc)

{

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

}

else

{

priorArc->nextarc = currentArc->nextarc;

}

//释放内存空间

free(currentArc);

break;

}

}

//弧数自减

G->arcnum--;

return OK;

}

Status DFSTraverse(ALGraph \* G, Status (\*visit)(VNode \* v))

{

int vpos = 0; //从第0个顶点开始遍历

//图不存在或为空，ERROR

if(G == NULL || G->vexnum == 0) return ERROR;

//访问标志数组初始化

for(vpos = 0;vpos < G->vexnum;vpos++) visited[vpos] = FALSE;

//对尚未访问的顶点调用DFS

for(vpos = 0;vpos < G->vexnum;vpos++)

{

if(!visited[vpos]) DFS(G, G->vertices + vpos, visit);

}

return OK;

}

Status DFS(ALGraph \*G, VNode \* v, Status (\*visit)(VNode \* v))

{

int vpos, wpos; //顶点位序

VNode \* w; //邻接顶点

//图不存在或为空，ERROR

if(G == NULL || G->vexnum == 0) return ERROR;

//从第v个顶点出发，递归地深度优先遍历图G

if ((vpos = LocateVex(G, v->data)) == -1) return ERROR;

//标记已访问该顶点

visited[vpos] = TRUE;

//访问第v个顶点

visit(v);

//访问邻接顶点

for(w = FirstAdjVex(G, v->data); w != NULL; w = NextAdjVex(G, v->data, w->data))

{

if ((wpos = LocateVex(G, w->data)) == -1) return ERROR;

//对v的尚未访问的邻接顶点w递归调用DFS

if(!visited[wpos]) DFS(G, w, visit);

}

return OK;

}

Status BFSTraverse(ALGraph \* G, Status (\*visit)(VNode \* v))

{

//顶点位序

int vpos, wpos;

//邻接顶点

VNode \* u;

VNode \* v;

VNode \* w;

//辅助用队列

Queue Q;

//图不存在或为空，ERROR

if(G == NULL || G->vexnum == 0) return ERROR;

//访问标志数组初始化

for(vpos = 0;vpos < G->vexnum;vpos++) visited[vpos] = FALSE;

//置空的辅助队列Q

InitQueue(&Q);

for(vpos = 0;vpos < G->vexnum;vpos++)

{

//尚未访问v

if(!visited[vpos])

{

visited[vpos] = TRUE;

v = G->vertices + vpos;

visit(v);

//v入队列

EnQueue(&Q, v);

while(!QueueEmpty(Q))

{

//队首元素出队并置为u

DeQueue(&Q, &u);

for (w = FirstAdjVex(G, u->data);w != NULL;w = NextAdjVex(G, u->data, w->data))

{

if ((wpos = LocateVex(G, w->data)) == -1) return ERROR;

//w为u的尚未访问的邻接顶点

if(!visited[wpos])

{

visited[wpos] = TRUE;

visit(w);

EnQueue(&Q, w);

} //if

} //for

} //while

} //if

}

DestroyQueue(&Q);

return OK;

}

Status Visit (VNode \* v)

{

//顶点不存在,ERROR

if (v == NULL || v->firstarc == NULL) return ERROR;

//否则，访问该顶点，输出顶点信息

printf("%c\n", v->data);

return OK;

}

Status Print (VNode \* v)

{

ArcNode \* arc; //遍历目标顶点的邻接链表的循环变量

//顶点不存在,ERROR

if (v == NULL || v->data == '\0') return ERROR;

//遍历输出目标顶点的邻接链表

for(arc = v->firstarc;arc != NULL;arc = arc->nextarc)

{

//邻接顶点未被访问，则访问邻接顶点，并输出弧信息

if(visited[arc->adjvex] == FALSE)

{

//打印邻接弧的信息

printf("%s\n",arc->info);

}

}

return OK;

}

Status LoadData (ALGraph \* G)

{

int i, j; //循环变量

VertexType vex\_data; //某一顶点的名称

VertexType adjvex\_data; //某一顶点的邻接顶点名称

int vexnum; //图的顶点总数

int nextarc\_num; //某一顶点的邻接边数

FILE \*fp; //文件指针

if ((fp = fopen("output.txt","r")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//将G的顶点总数和边总数初始化为0

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

//图不存在,ERROR

if(G == NULL) return ERROR;

//否则，读取数据

//读取顶点表信息

if (!fread(&vexnum, sizeof(int), 1, fp)) return ERROR;

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

{

if(!fread(&(vex\_data), sizeof(char), 1, fp)) return ERROR;

InsertVex(G, vex\_data);

}

//读取邻接表信息

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

{

//读取当前顶点的邻接顶点数

if(!fread(&nextarc\_num, sizeof(int), 1, fp)) return ERROR;

//读取当前顶点的邻接顶点名称，并插入连接两点的弧

for(j = 0;j < nextarc\_num;j++)

{

if(!fread(&adjvex\_data, sizeof(char), 1, fp)) break;

if(LocateVex(G, adjvex\_data) > i)

{

InsertArc(G, G->vertices[i].data, adjvex\_data);

}

}

}

fclose(fp);

return OK;

}

Status SaveData (ALGraph \* G)

{

int nextarc\_num; //某一顶点的邻接边数

int pos; //位序循环变量

ArcNode \* arc; //弧顶点循环变量

FILE \*fp; //文件指针

//以覆盖原有数据的形式,打开output.txt

if ((fp = fopen("output.txt","w+")) == NULL)

{

printf("File OPEN ERROR\n ");

return ERROR;

}

//图不存在或为空，ERROR

if(G == NULL || G->vexnum == 0) return ERROR;

//否则,保存数据

//保存顶点表信息

//保存顶点总数

fwrite(&(G->vexnum), sizeof(int), 1, fp);

//保存顶点信息

for(pos = 0;pos < G->vexnum;pos++)

{

fwrite(&(G->vertices[pos].data), sizeof(char), 1, fp);

}

//保存各顶点邻接链表信息

for (pos = 0;pos < G->vexnum;pos++)

{

//每个邻接链表的第一个数据用于保存当前顶点的邻接顶点数

//初始化当前顶点的邻接边数

nextarc\_num = 0;

//遍历邻接链表计算邻接顶点数

for(arc = G->vertices[pos].firstarc;arc != NULL;arc = arc->nextarc)

{

nextarc\_num++;

}

//保存邻接顶点数

fwrite(&nextarc\_num, sizeof(int), 1, fp);

//每个邻接链表的其余数据保存当前顶点的邻接顶点名称

for(arc = G->vertices[pos].firstarc;arc != NULL;arc =arc->nextarc)

{

fwrite(&(G->vertices[arc->adjvex].data), sizeof(char), 1, fp);

}

}

fclose(fp);

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 说明：利用队列,实现广度优先遍历算法

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InitQueue (Queue \* Q)

{

//分配定长字节内存单元，并进行成功性检查

Q->base = (VNode \*\*)malloc(MAXQSIZE \* sizeof(VNode \*));

if (! Q->base) return OVERFLOW;

//初始化

Q->rear = Q->front = 0;

return OK;

}

Status DestroyQueue (Queue \* Q)

{

//队列不存在

if (! Q->base) return ERROR;

//队列存在，函数正确执行

free(Q->base);

//将base重置为null，front与rear重置为0

Q->base = NULL;

Q->rear = Q->front = 0;

return OK;

}

Status QueueEmpty (Queue Q)

{

//队列不存在

if (! Q.base) return ERROR;

//队列存在，函数正确执行

return Q.rear == Q.front ? TRUE : FALSE;

}

Status EnQueue (Queue \* Q, VNode \* v)

{

//队列不存在

if (! Q->base) return ERROR;

//队列满

if ( (Q->rear + 1) % MAXQSIZE == Q->front ) return ERROR;

//队列存在且未满

Q->base[Q->rear] = v;

//队列尾下标自增

Q->rear = (Q->rear + 1) % MAXQSIZE;

return OK;

}

Status DeQueue (Queue \* Q, VNode \* \* v)

{

//队列不存在

if (! Q->base) return ERROR;

//队列为空

if (QueueEmpty(\*Q)) return ERROR;

//队列存在且非空

(\*v) = Q->base[Q->front];

Q->front = (Q->front + 1) % MAXQSIZE;

return OK;

}

1. main.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验六：邻接表存储图主程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Graph.h"

int main (void)

{

int choice = '\0'; //用户功能选择变量

VertexType input; //用户输入变量

ALGraph \* graph = NULL;

VNode \* vex; //用于存放各个功能函数输入参数以及反馈的数据元素

Status (\*visit) (VNode \*) = Visit;//访问操作函数指针

Status (\*print) (VNode \*) = Print;//打印操作函数指针

//为图指针分配内存空间

graph = (ALGraph \*)malloc(sizeof(ALGraph));

if (graph == NULL) return OVERFLOW;

//读取文本数据

LoadData(graph);

//打印系统菜单界面

printf ("\t\t欢迎使用无向图系统\n\n");

while (TRUE)

{

//打印系统菜单界面

printf ("1:创建图\n");

printf ("2:查询顶点编号\n");

printf ("3:查询第一邻接顶点\n");

printf ("4:查询下一邻接顶点\n");

printf ("5:插入顶点\n");

printf ("6:删除顶点\n");

printf ("7:插入弧\n");

printf ("8:删除弧\n");

printf ("9:深度优先遍历\n");

printf ("10:广度优先遍历\n");

printf ("11:打印无向图\n");

printf ("12:退出系统\n\n");

//提示用户选择功能

printf ("请选择功能:");

//用户选择事件

scanf ("%d", &choice);

//处理用户选择事件

switch (choice)

{

case 1: //创建图

if (CreateGraph(graph) == OK)

printf("创建成功\n");

else printf("创建失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 2: //查询顶点编号

printf("请输入目标结点名称:");

scanf("%1s", &input);

if ((choice = LocateVex(graph, input)) != -1)

printf("%c 位序:%d\n", input, choice);

else printf("定位 %c 失败\n", input);

printf("\n");

system("pause");

system("CLS");

break;

case 3: //查询第一邻接顶点

printf("请输入目标结点名称:");

scanf("%1s", &input);

if ((vex = FirstAdjVex(graph, input)) != NULL)

printf("%c 第一个邻接顶点:%c\n", input, vex->data);

else printf("查询失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 4: //查询下一邻接顶点

printf("请输入目标结点名称:");

scanf("%1s", &input);

printf("请输入%c的一个邻接顶点:", input);

scanf("%1s", &choice);

if ((vex = NextAdjVex(graph, input, choice)) != NULL)

printf("%c 相对于 %c 的下一个邻接顶点:%c\n", input, choice, vex->data);

else printf("查询失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 5: //插入顶点

printf("请输入插入结点名称:");

scanf("%1s", &input);

if (InsertVex(graph, input) == OK)

printf("插入 %c 成功\n", input);

else printf("插入 %c 失败\n", input);

printf("\n");

system("pause");

system("CLS");

break;

case 6: //删除顶点

printf("请输入删除结点名称:");

scanf("%1s", &input);

if (DeleteVex(graph, input) == OK)

printf("删除 %c 成功\n", input);

else printf("删除 %c 失败\n", input);

printf("\n");

system("pause");

system("CLS");

break;

case 7: //插入弧

printf("请输入插入弧端点(v w):");

scanf("%1s %1s", &input, &choice);

if (InsertArc(graph, input, choice) == OK)

printf("插入 <%c,%c> 成功\n", input, choice);

else printf("插入 <%c,%c> 失败\n", input, choice);

printf("\n");

system("pause");

system("CLS");

break;

case 8: //删除弧

printf("请输入删除弧端点(v w):");

scanf("%1s %1s", &input, &choice);

if (DeleteArc(graph, input, choice) == OK)

printf("删除 <%c,%c> 成功\n", input, choice);

else printf("删除 <%c,%c> 失败\n", input, choice);

printf("\n");

system("pause");

system("CLS");

break;

case 9: //深度优先遍历

if (DFSTraverse(graph, visit) == OK)

;

else printf("深度优先遍历失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 10: //广度优先遍历

if (BFSTraverse(graph, visit) == OK)

;

else printf("深度优先遍历失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 11: //打印无向图

if (DFSTraverse(graph, print) == OK)

;

else printf("打印失败\n");

printf("\n");

system("pause");

system("CLS");

break;

case 12: //退出系统

//保存图

SaveData(graph);

//销毁图

DestroyGraph(graph);

printf("\n");

system("pause");

system("CLS");

exit (OK);

break;

default:

printf ("选择功能错误\n");

printf("\n");

system("pause");

system("CLS");

break;

}

printf ("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

}

1. assert\_test.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作者：计科1409班 U201414800 刘一龙

\* 说明：数据结构上机实验六：邻接表存储图断言测试

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <assert.h>

#include "Graph.h"

#undef NDEBUG //开启断言测试

int main (void)

{

ALGraph \* G = NULL; //测试用图

int i; //测试用循环变量，用于给出顶点位置

VertexType data; //测试用循环变量，用于给出顶点名称

VNode \* vex = NULL; //测试用顶点指针，用于存储查询顶点函数的返回值

Status (\*visit)(VNode\*) = Visit; //访问操作函数指针

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 断言测试 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 不存在图 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printf("不存在图测试:检验程序的健壮性\n");

/\*\*\*\*\*\*\*\*\*\* 查询模块 \*\*\*\*\*\*\*\*\*\*/

//定位失败测试

assert(LocateVex(G, 'A') == -1);

printf("Locate Failed\n");

//查询邻接顶点失败测试

assert(FirstAdjVex(G, 'A') == NULL);

printf("Require First AdjVex Failed\n");

//查询下一邻接顶点失败测试

assert(NextAdjVex(G, 'A', 'B') == NULL);

printf("Require Next AdjVex Failed\n");

/\*\*\*\*\*\*\*\*\*\* 编辑模块 \*\*\*\*\*\*\*\*\*\*/

//插入顶点失败测试

assert(InsertVex(G, 'A') == ERROR);

printf("InsertVex Failed\n");

//删除顶点失败测试

assert(DeleteVex(G, 'A') == ERROR);

printf("DeleteVex Failed\n");

//插入弧失败测试

assert(InsertArc(G, 'A', 'B') == ERROR);

printf("InsertArc Failed\n");

//删除弧失败测试

assert(DeleteArc(G, 'A', 'B') == ERROR);

printf("DeleteArc Failed\n");

/\*\*\*\*\*\*\*\*\*\* 遍历模块 \*\*\*\*\*\*\*\*\*\*/

assert(DFSTraverse(G, visit) == ERROR);

assert(BFSTraverse(G, visit) == ERROR);

/\*\*\*\*\*\*\*\*\*\* 销毁模块 \*\*\*\*\*\*\*\*\*\*/

//不存在图销毁测试

assert(DestroyGraph(G) == ERROR);

printf("Destroy Failed\n");

system("pause");

printf("\n");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 空图 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printf("空图测试:检验程序的健壮性\n");

//为图指针分配内存空间

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

if (G == NULL) return OVERFLOW;

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

/\*\*\*\*\*\*\*\*\*\* 查询模块 \*\*\*\*\*\*\*\*\*\*/

//定位失败测试

assert(LocateVex(G, 'A') == -1);

printf("Locate Failed\n");

//查询邻接顶点失败测试

assert(FirstAdjVex(G, 'A') == NULL);

printf("Require First AdjVex Failed\n");

//查询下一邻接顶点失败测试

assert(NextAdjVex(G, 'A', 'B') == NULL);

printf("Require Next AdjVex Failed\n");

/\*\*\*\*\*\*\*\*\*\* 编辑模块 \*\*\*\*\*\*\*\*\*\*/

//插入顶点测试

assert(InsertVex(G, 'A') == OK);

assert(InsertVex(G, 'B') == OK);

assert(InsertVex(G, 'C') == OK);

assert(InsertVex(G, 'D') == OK);

printf("InsertVex 'A''B''C''D' Success\n");

//删除顶点测试

assert(DeleteVex(G, 'C') == OK);

printf("DeleteVex 'C' Success\n");

assert(DeleteVex(G, 'E') == ERROR);

printf("DeleteVex 'E' Failed\n");

//插入弧测试

assert(InsertArc(G, 'A', 'B') == OK);

printf("InsertArc <A,B> Success\n");

assert(InsertArc(G, 'A', 'D') == OK);

printf("InsertArc <A,D> Success\n");

assert(InsertArc(G, 'A', 'C') == ERROR);

printf("InsertArc <A,C> Failed\n");

//删除弧测试

assert(DeleteArc(G, 'A', 'B') == OK);

printf("DeleteArc <A,B> Success\n");

assert(DeleteArc(G, 'A', 'C') == ERROR);

printf("DeleteArc <A,B> Failed\n");

//通过以上四个操作构造一张字母图

printf("通过以上4个插入、删除操作构造一张字母图如下:\n");

assert(InsertVex(G, 'C') == OK);

assert(InsertVex(G, 'E') == OK);

assert(InsertArc(G, 'A', 'B') == OK);

assert(InsertArc(G, 'B', 'C') == OK);

assert(InsertArc(G, 'C', 'D') == OK);

assert(InsertArc(G, 'B', 'E') == OK);

assert(InsertArc(G, 'C', 'E') == OK);

/\*\*\*\*\*\*\*\*\*\* 遍历模块 \*\*\*\*\*\*\*\*\*\*/

printf("深度优先遍历:\n");

assert(DFSTraverse(G, visit) == OK);

printf("广度优先遍历:\n");

assert(BFSTraverse(G, visit) == OK);

/\*\*\*\*\*\*\*\*\*\* 销毁模块 \*\*\*\*\*\*\*\*\*\*/

//空图销毁测试

assert(DestroyGraph(G) == OK);

printf("Destroy Success\n");

system("pause");

printf("\n");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 字母图 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\* 创建模块 \*\*\*\*\*\*\*\*\*\*/

//为图指针分配内存空间

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

if (G == NULL) return OVERFLOW;

//创建无向图测试

assert(CreateGraph(G) == OK);

printf("Create Success\n");

/\*\*\*\*\*\*\*\*\*\* 查询模块 \*\*\*\*\*\*\*\*\*\*/

//定位函数测试

for(i = 0, data = 'A';i < G->vexnum;i++, data++)

{

assert(LocateVex(G, data) == i);

printf("No.%d : %c\n", i, data);

}

//查询邻接顶点测试

for(i = 0, data = 'A';i < G->vexnum;i++, data++)

{

assert((vex = FirstAdjVex(G, data)) != NULL);

printf("No.%d(%c) First AdjVex:%c \n", i, data, vex->data);

}

//查询下一邻接顶点测试

for(i = 0, data = 'A';i < G->vexnum;i++, data++)

{

//定位顶点data的第一个邻接顶点

vex = FirstAdjVex(G, data);

//查询vex第二个邻接顶点

assert((vex = NextAdjVex(G, data, vex->data)) != NULL);

printf("No.%d(%c) Next AdjVex:%c \n", i, data, vex->data);

}

/\*\*\*\*\*\*\*\*\*\* 遍历模块 \*\*\*\*\*\*\*\*\*\*/

printf("深度优先遍历:\n");

assert(DFSTraverse(G, visit) == OK);

printf("广度优先遍历:\n");

assert(BFSTraverse(G, visit) == OK);

/\*\*\*\*\*\*\*\*\*\* 销毁模块 \*\*\*\*\*\*\*\*\*\*/

//销毁无向图测试

assert(DestroyGraph(G) == OK);

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 断言测试成功! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

system("pause");

return OK;

}

1. Makefile

DT\_LAB\_06.exe : Graph.o main.o

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -static -g -o DT\_LAB\_06.exe Graph.o main.o

Graph.o : Graph.c Graph.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -static -g -c Graph.c

main.o : main.c Graph.h

gcc -Wall -std=c99 -finput-charset=UTF-8 -fexec-charset=GBK -static -g -c main.c

6.3.2 测试部分

本次测试采用断言测试，测试主程序及源代码见附件AssertTest.exe及程序清单assert\_test.c。

一、测试用例为通过InsertVex/DeleteVex/InsertArc/DeleteArc四个函数复合生成，其边集为 (A-B, A-D, B-C, B-E, C-D, C-E)，结构如下图所示：



图6-1 测试用例：由空图生成的无向图

二、创建模块测试用例为一个顶点总数为7,边数为11的数字字符二叉树，结构如下图所示：



图6-2 测试用例：由用户输入创建的无向图

三、测试流程如下：

1.不存在图

**/\*\*\*\*\*\*\*\*\*\* 查询模块 \*\*\*\*\*\*\*\*\*\*/**

//定位失败测试

assert(LocateVex(G, 'A') == -1);

//查询邻接顶点失败测试

assert(FirstAdjVex(G, 'A') == NULL);

//查询下一邻接顶点失败测试

assert(NextAdjVex(G, 'A', 'B') == NULL);

**/\*\*\*\*\*\*\*\*\*\* 编辑模块 \*\*\*\*\*\*\*\*\*\*/**

//插入顶点失败测试

assert(InsertVex(G, 'A') == ERROR);

//删除顶点失败测试

assert(DeleteVex(G, 'A') == ERROR);

//插入弧失败测试

assert(InsertArc(G, 'A', 'B') == ERROR);

//删除弧失败测试

assert(DeleteArc(G, 'A', 'B') == ERROR);

**/\*\*\*\*\*\*\*\*\*\* 遍历模块 \*\*\*\*\*\*\*\*\*\*/**

assert(DFSTraverse(G, visit) == ERROR);

assert(BFSTraverse(G, visit) == ERROR);

**/\*\*\*\*\*\*\*\*\*\* 销毁模块 \*\*\*\*\*\*\*\*\*\*/**

//不存在图销毁测试

assert(DestroyGraph(G) == ERROR);

2.空图

//为图指针分配内存空间

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

if (G == NULL) return OVERFLOW;

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

**/\*\*\*\*\*\*\*\*\*\* 查询模块 \*\*\*\*\*\*\*\*\*\*/**

//定位失败测试

assert(LocateVex(G, 'A') == -1);

//查询邻接顶点失败测试

assert(FirstAdjVex(G, 'A') == NULL);

//查询下一邻接顶点失败测试

assert(NextAdjVex(G, 'A', 'B') == NULL);

**/\*\*\*\*\*\*\*\*\*\* 编辑模块 \*\*\*\*\*\*\*\*\*\*/**

//插入顶点测试

assert(InsertVex(G, 'A') == OK);

assert(InsertVex(G, 'B') == OK);

assert(InsertVex(G, 'C') == OK);

assert(InsertVex(G, 'D') == OK);

//删除顶点测试

assert(DeleteVex(G, 'C') == OK);

assert(DeleteVex(G, 'E') == ERROR);

//插入弧测试

assert(InsertArc(G, 'A', 'B') == OK);

assert(InsertArc(G, 'A', 'D') == OK);

assert(InsertArc(G, 'A', 'C') == ERROR);

//删除弧测试

assert(DeleteArc(G, 'A', 'B') == OK);

assert(DeleteArc(G, 'A', 'C') == ERROR);

//通过以上四个操作构造一张字母图

assert(InsertVex(G, 'C') == OK);

assert(InsertVex(G, 'E') == OK);

assert(InsertArc(G, 'A', 'B') == OK);

assert(InsertArc(G, 'B', 'C') == OK);

assert(InsertArc(G, 'C', 'D') == OK);

assert(InsertArc(G, 'B', 'E') == OK);

assert(InsertArc(G, 'C', 'E') == OK);

**/\*\*\*\*\*\*\*\*\*\* 遍历模块 \*\*\*\*\*\*\*\*\*\*/**

assert(DFSTraverse(G, visit) == OK);

assert(BFSTraverse(G, visit) == OK);

**/\*\*\*\*\*\*\*\*\*\* 销毁模块 \*\*\*\*\*\*\*\*\*\*/**

//空图销毁测试

assert(DestroyGraph(G) == OK);

3.字母图

**/\*\*\*\*\*\*\*\*\*\* 创建模块 \*\*\*\*\*\*\*\*\*\*/**

//为图指针分配内存空间

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

if (G == NULL) return OVERFLOW;

//创建无向图测试

assert(CreateGraph(G) == OK);

**/\*\*\*\*\*\*\*\*\*\* 查询模块 \*\*\*\*\*\*\*\*\*\*/**

//定位函数测试

for(i = 0, data = 'A';i < G->vexnum;i++, data++)

{

assert(LocateVex(G, data) == i);

}

//查询邻接顶点测试

for(i = 0, data = 'A';i < G->vexnum;i++, data++)

{

assert((vex = FirstAdjVex(G, data)) != NULL);

}

//查询下一邻接顶点测试

for(i = 0, data = 'A';i < G->vexnum;i++, data++)

{

//定位顶点data的第一个邻接顶点

vex = FirstAdjVex(G, data);

//查询vex第二个邻接顶点

assert((vex = NextAdjVex(G, data, vex->data)) != NULL);

}

**/\*\*\*\*\*\*\*\*\*\* 遍历模块 \*\*\*\*\*\*\*\*\*\*/**

assert(DFSTraverse(G, visit) == OK);

assert(BFSTraverse(G, visit) == OK);

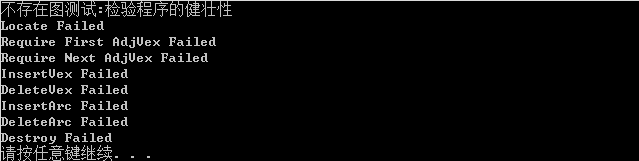
**/\*\*\*\*\*\*\*\*\*\* 销毁模块 \*\*\*\*\*\*\*\*\*\*/**

//销毁无向图测试

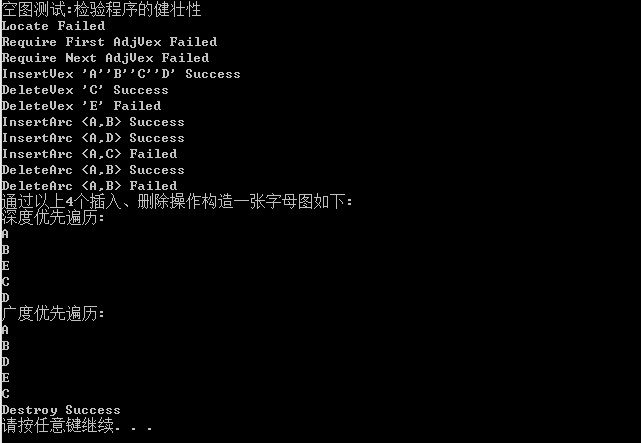
assert(DestroyGraph(G) == OK);

四、测试结果如以下测试图所示，所有断言成功，程序正常退出，结果符合预期。

1.不存在图

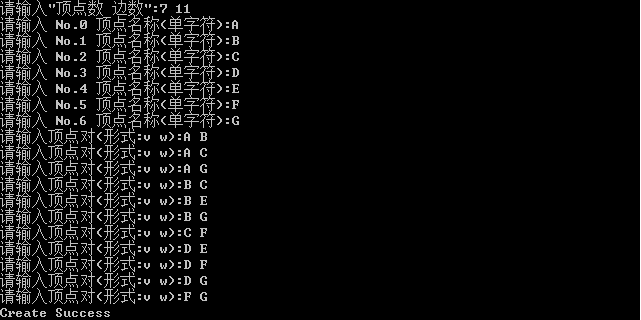


2.空图(测试用例1)



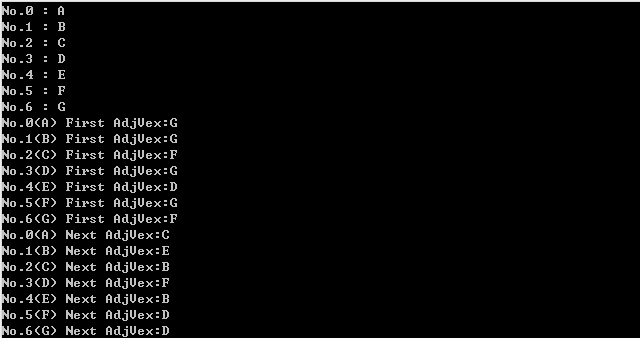
3.字母图(测试用例2)

a.利用CreateGraph函数创建无向图

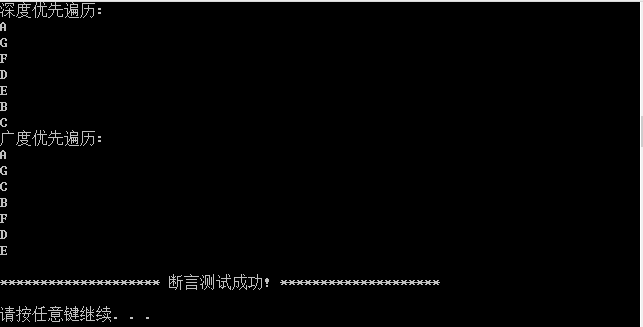


b.断言测试结果图

查询模块



遍历模块



6.4 实验小结

一、此次实验的断言测试代码减少至159行，但基本实现了测试功能(不存在图、空图、字母图、边界检查)，极大地加快了Debug指针方面Bug的速度，使得最终的系统具有较好的健壮性和容错性；

二、通过实现二叉链存储二叉树的基本操作，加强了对二叉树的理解和使用，希望能为以后的更加高级灵活的数据结构的学习带来帮助。

**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  (40分) | 程序规范  (10分) | 报告内容  (40分) | 报告规范  (10分) | 逾 期  扣 分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |

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