

**课 程 实 验 报 告**

**课程名称： 数 据 结 构 实 验**

**专业班级：**

**学 号：**

**姓 名：**

**指导教师：**

**报告日期： 2015年11月20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](#_Toc440364266)

[**1.1 问题描述** 1](#_Toc440364267)

[**1.2 顺序表演示系统设计** 3](#_Toc440364268)

[**1.2.1 系统总体设计** 3](#_Toc440364269)

[**1.2.2 有关常量和类型定义** 3](#_Toc440364270)

[**1.2.3 算法设计** 3](#_Toc440364271)

[**1.3 顺序表演示系统实现与测试** 8](#_Toc440364272)

[**1.3.1 系统实现** 8](#_Toc440364273)

[**1.3.2 系统测试** 18](#_Toc440364274)

[**1.4 实验小结** 20](#_Toc440364275)

[**2基于链式存储结构实现线性表的基本运算** 21](#_Toc440364276)

[**2.1 问题描述** 21](#_Toc440364277)

[**2.2 单链表演示系统设计** 21](#_Toc440364278)

[**2.2.1 系统总体设计** 21](#_Toc440364279)

[**2.2.3 算法设计** 22](#_Toc440364280)

[**2.3 单链表演示系统实现与测试** 25](#_Toc440364281)

[**2.3.1 系统实现** 25](#_Toc440364282)

[**2.3.2 系统测试** 36](#_Toc440364283)

[**2.4 实验小结** 37](#_Toc440364284)

[**3基于顺序存储结构实现栈的基本运算** 38](#_Toc440364285)

[**3.1 顺序栈实验** 38](#_Toc440364286)

[**3.1.1 问题描述** 38](#_Toc440364287)

[**3.1.2 顺序栈设计** 38](#_Toc440364288)

[**3.1.2 顺序栈实现与测试** 44](#_Toc440364289)

[**3.2 表达式求值实验** 55](#_Toc440364290)

[**3.2.1 问题描述** 55](#_Toc440364291)

[**3.2.2算法设计部分** 55](#_Toc440364292)

[**3.2.3实现与测试部分** 56](#_Toc440364293)

[**3.3 实验小结** 57](#_Toc440364294)

[**4基于循环队列存储结构实现队列的基本运算** 58](#_Toc440364295)

[4.1 问题描述 58](#_Toc440364296)

[4.2 队列基本操作算法设计： 58](#_Toc440364297)

[4.3 实验小结 66](#_Toc440364298)

[**5基于二叉链表实现二叉树的基本运算** 67](#_Toc440364299)

[5.1实验内容与要求 67](#_Toc440364300)

[5.2程序概要设计 67](#_Toc440364301)

[5.3数据结构与算法设计 68](#_Toc440364302)

[5.4程序清单与测试 70](#_Toc440364303)

[5.5实验总结与评价 83](#_Toc440364304)

[**6基于邻接表实现图的基本运算** 84](#_Toc440364305)

[6.1实验内容与要求 84](#_Toc440364306)

[6.2程序概要设计 84](#_Toc440364307)

[6.3数据结构与算法设计 85](#_Toc440364308)

[6.4程序清单与测试 87](#_Toc440364309)

[6.5实验总结与评价 109](#_Toc440364310)

[**参考文献** 109](#_Toc440364311)

**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**

叙述实验中线性表的物理结构形式，如何用物理结构表示数据元素间的逻辑关系，可用图的方式直观表示物理结构，如图1-1所示。



图1-1 顺序表物理结构示意图

实验要完成的顺序表算法：

（1）InitaList(&L)

操作结果：构造一个空的线性表。

（2）DestroyList(&L)

初始条件：线性表L已存在。

操作结果：销毁线性表L。

（3）ClearList (&L)

初始条件：线性表L已存在。

操作结果：将L重置为空表。

（4）ListEmpty(L)

初始条件：线性表L已存在。

操作结果：若L为空表，则返回TRUE,否则返回FALSE。

（5）ListLength(L)

初始条件：线性表已存在。

操作结果：返回L中数据元素的个数。

（6）GetElem(L, i,&e)

初始条件：线性表已存在，1≤i≤ListLength(L)。

操作结果：用e返回L中第i个数据元素的值。

（7）LocateElem(L, e，compare（）)

初始条件：线性表已存在。

操作结果：返回L中第1个与e满足关系compare（）关系的数据元素的

位序，若这样的数据元素不存在，则返回值为0。

（8）PriorElem（L，cur\_e，&pre\_e）

初始条件：线性表L已存在。

操作结果：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。

（9）NextElem（L，cur\_e，&next\_e）

初始条件：线性表L已存在。

操作结果：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。

（10）ListInsert(&L,i,e)

初始条件：线性表L已存在且非空，1≤i≤ListLength(L)+1。

操作结果：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

（11）ListDelete(&L,i,&e)

初始条件：线性表L已存在且非空，1≤i≤ListLength(L)。

操作结果：删除L的第i个数据元素，用e返回其值，L的长度减1.

（12）ListTraverse(L，visit（）)

初始条件：线性表L已存在。

操作结果：依次对L的每个数据元素调用函数visit()。一旦调用失败，则操

作失败。

（13）ReadList(&L)

操作结果：读取线性表

（14）SaveList(L)

操作结果：保存线性表

实验目标：

构造成具有功能菜单的系统完成线性表基本操作。通过实验达到：（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。

**1.2 顺序表演示系统设计**

**1.2.1 系统总体设计**

系统的总体架构：界面上采用简易菜单，通过switch函数进行功能的选择，进入相关功能函数执行相关操作。

**1.2.2 有关常量和类型定义**

typedef int ElemType; //数据元素类型定义

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

typedef struct{ //顺序表（顺序结构）的定义

ElemType \* elem;

int length;

int listsize;

}SqList;

**1.2.3 算法设计**

（1）InitaList(&L)

设计：分配存储空间，并将length值设为0，listsize值设为预定义的初始存储容量。

复杂度：时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

1. DestroyList(&L)

设计：释放内存，并让ele指向NULL，表长及存储容量置为0。

复杂度：时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（3）ClearList (&L)

设计：将表长设为0。

复杂度：时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（4）ListEmpty(L)

设计：读取表长length的值，为0则返回TRUE，否则FALSE。

复杂度：时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（5）ListLength(L)

设计：返回L.length的值。

复杂度：时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（6）GetElem(L, i,&e)

设计：将L.elem[i - 1]的值赋给e，并返回OK

复杂度：时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（7）LocateElem(L, e，compare（）)

设计：遍历顺序表，将与给定元素e满足关系compare的元素的位序返回。

复杂度：时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

1. PriorElem（L，cur\_e，&pre\_e）

设计：从头遍历，若当前节点后继值为cur\_e，则将当前结点的元素赋给pre\_e，return OK。若未找到，则返回FALSE。流程图如下所示：



复杂度：时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

1. NextElem（L，cur\_e，&next\_e）

设计：从头遍历，若当前节点值为cur\_e且非表尾，则将后继结点的元素赋给pre\_e，return OK。若未找到，则返回FALSE。流程图如下：



复杂度：时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

1. ListInsert(&L,i,e)

设计：先判断i值是否符合要求，不符返回ERROR。i值合法，则检查空间大小，若空间不够，增加存储容量。然后将插入位置及之后的元素右移，最后插入e，表长加一。流程图如下所示：



复杂度：时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

1. ListDelete(&L,i,&e)

设计：先判断i值是否符合要求，不符返回ERROR。i值合法，则将被删除元素的值赋给e，然后将被删除元素之后的元素左移 ，最后表长减一。

流程图如下所示：



复杂度：时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

1. ListTraverse(L，visit（）)

设计：遍历顺序表，对每个元素调用函数指针visit所指向的函数，若调用失败则提示操作失败。

复杂度：时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

1. ReadList(&L)

设计：若原L已存在，销毁，新建，再要求用户输入读取文件名，然后使用fread函数进行文件的读取。

复杂度：时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

1. SaveList(L)

设计：让用户输入所要保存到的文件名，然后使用fwrite函数将顺序表保存入文件中。

复杂度：时间复杂度T(n) = O(1)

空间复杂度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**

编程环境：WIN 10下使用Dev—C++进行编译调试，语言选择C语言。

各函数间调用关系：主函数通过switch调用各功能函数，各函数间除ReadList调用DestroyList和InitList外，各自相互独立。

程序源代码：

#include <stdio.h>

#include <stdlib.h>

//函数结果状态代码

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -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;

//数据元素类型定义

typedef int ElemType;

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

typedef struct{ //顺序表（顺序结构）的定义

ElemType \* elem;

int length;

int listsize;

}SqList;

Status InitList(SqList \* L);

Status DestroyList(SqList \* L);

Status ClearList(SqList \* L);

Status ListEmpty(SqList L);

int ListLength(SqList L);

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

int LocateElem(SqList L,ElemType e, Status (\*compare)(ElemType a, ElemType b));

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

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

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

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

Status ListTraverse(SqList L, Status (\*visit)(ElemType a));

Status ReadList(SqList \* L);

Status SaveList(SqList L);

Status Compare(ElemType a, ElemType b);

Status Visit(ElemType a);

/\*

\*函数名称：IntiaList

\*输入参数：线性表L地址

\*返回值：Status

\*函数功能：构造一个空的线性表。

\*/

Status InitList(SqList \* L)

{

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

if(!L->elem)

{

exit(OVERFLOW);

}

L->length=0;

L->listsize=LIST\_INIT\_SIZE;

return OK;

}

/\*

\*函数名称：DestroyList

\*输入参数：线性表L地址

\*返回值：Status

\*函数功能：构造一个空的线性表。

\*/

Status DestroyList(SqList \* L)

{

free(L->elem);

L->elem = NULL;

L->length=0;

L->listsize=0;

return OK;

}

/\*

\*函数名称：ClearList

\*输入参数：线性表L地址

\*返回值：Status

\*函数功能：将L重置为空表。

\*/

Status ClearList(SqList \* L)

{

L->length = 0;

return OK;

}

/\*

\*函数名称：ListEmpty

\*输入参数：线性表L

\*返回值：Status

\*函数功能：若L为空表，则返回TRUE,否则返回FALSE。

\*/

Status ListEmpty(SqList L)

{

if (L.length == 0)

{

return TRUE;

}

else

{

return FALSE;

}

}

/\*

\*函数名称：ListLength

\*输入参数：线性表L

\*返回值：Status

\*函数功能：返回L中数据元素的个数。

\*/

int ListLength(SqList L)

{

return L.length;

}

/\*

\*函数名称：GetElem

\*输入参数：线性表L,结点位置i，用于返回的e的地址

\*返回值：Status

\*函数功能：用e返回L中第i个数据元素的值。

\*/

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

{

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

{

return ERROR;

}

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

return OK;

}

/\*

\*函数名称：LocateElem

\*输入参数：线性表L,用于比较的元素e，指向用于比较的函数的函数指针compare

\*返回值：int

\*函数功能：返回L中第1个与e满足关系compare（）关系的数据元素的

\* 位序，若这样的数据元素不存在，则返回值为0。

\*/

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

{

int i;

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

{

if ((\*compare)(L.elem[i-1], e))

{

return(i);

}

}

return 0;

}

/\*

\*函数名称：PriorElem

\*输入参数：线性表L,找的元素cur\_e，用于返回其前驱的元素pre\_e的地址

\*返回值：Status

\*函数功能：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)

{

int i;

for (i = 0; i < (L.length-1); i++)

{

if (L.elem[i+1] == cur\_e)

{

\*pre\_e = L.elem[i];

return OK;

}

}

return ERROR;

}

/\*

\*函数名称：NextElem

\*输入参数：线性表L,查找的元素cur\_e，用于返回其后继的元素next\_e的地址

\*返回值：Status

\*函数功能：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)

{

int i;

for (i = 0; i < (L.length-1); i++)

{

if (L.elem[i] == cur\_e)

{

\*next\_e = L.elem[i+1];

return OK;

}

}

return ERROR;

}

/\*

\*函数名称：ListInsert

\*输入参数：线性表L地址, 插入位置i，插入元素e

\*返回值：Status

\*函数功能：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

\*/

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

{

if (i < 1 || i > L->length+1) //i值不合法

{

return ERROR;

}

if (L->length >= L->listsize) //当前存储空间已满，增加分配

{

ElemType \* newbase;

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

}

ElemType \* p, \* q;

q = &(L->elem[i-1]); //q为插入位置

for (p = &(L->elem[L->length-1]); p >= q; --p)

{

\*(p+1) = \*p; //插入位置及之后的元素右移

}

\*q = e; //插入e

++L->length; //表长增加1

return OK;

}

/\*

\*函数名称：ListDelete

\*输入参数：线性表L地址,删除位置i，用于返回其值的元素e的地址

\*返回值：Status

\*函数功能：删除L的第i个数据元素，用e返回其值，L的长度减1.

\*/

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

{

ElemType \* p, \* q;

if (i < 1 || i > L->length) //i值不合法

{

return ERROR;

}

p = &(L->elem[i-1]); //p为被删元素位置

\*e = \*p; //被删除元素的值赋给e

q = L->elem + L->length - 1; //表尾元素的位置

for (++p; p <= q; ++p) //被删除元素之后的元素左移

{

\*(p-1) = \*p;

}

--L->length; //表长减1

return OK;

}

/\*

\*函数名称：ListTraverse

\*输入参数： 线性表L, 指向调用函数的函数指针visit

\*返回值： Status

\*函数功能： 依次对L的每个数据元素调用函数visit()。

\*/

Status ListTraverse(SqList L, Status(\*visit)(ElemType a))

{

int i;

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

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

{

if (!(\*visit)(L.elem[i]))

{

printf("操作失败！");

}

}

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

return L.length;

}

/\*

\*函数名称：ReadList

\*输入参数： 线性表L

\*返回值： Status

\*函数功能： 读取线性表。

\*/

Status ReadList(SqList \* L)

{

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

{

DestroyList(L);

}

InitList(L);

FILE \*fp;

char filename[30];

printf("请输入读取文件名: ");

scanf("%s", &filename);

if ((fp = fopen(filename,"r")) == NULL)

{

printf("文件打开失败！\n ");

return ERROR;

}

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

L->length++;

fclose(fp);

return OK;

}

/\*

\*函数名称：SaveList

\*输入参数： 线性表L

\*返回值： Status

\*函数功能： 保存线性表。

\*/

Status SaveList(SqList L)

{

FILE \*fp;

char filename[30];

printf("请输入保存文件名： ");

scanf("%s",&filename);

//写文件的方法

if ((fp = fopen(filename, "w")) == NULL)

{

printf("文件打开失败\n ");

return ERROR;

}

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

fclose(fp);

return OK;

}

/\*

\*函数名称： Compare

\*输入参数： ElemType a, ElemType b

\*返回值： Status

\*函数功能： 判断a，b是否满足关系

\*/

Status Compare(ElemType a, ElemType b)

{

if (a == b)

{

return TRUE;

}

else

{

return FALSE;

}

}

/\*

\*函数名称： Visit

\*输入参数： ElemType a

\*返回值： Status

\*函数功能： 输出a

\*/

Status Visit(ElemType a)

{

printf("%d ",a);

return OK;

}

**1.3.2 系统测试**

使用预先保存的测试用例挑选LocateElem，NextElem，ListInsert，ListDelete、ListTraverse进行测试。

使用的测试用例：Test1(表含5,12,0,44,94) ，Test2(空表)。

（1）LocateElem测试：

表1.3.2-1　LocateElem算法测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| Test1 | 1.主界面选择7进入函数  2.按提示输入所要比较的函数（本程序比较函数设为两数相等返回ＴＲＵＥ，故提示语为输入要搜寻的函数），输入12 | 输出“第二个元素符合条件”。 | 输出“第二个元素符合条件”，符合预期。 |
| 1.主界面选择7进入函数  2.输入100 | 提示“未找到符合条件的元素” | 输出“未找到符合条件的元素”，符合预期。 |
| Test2 | 1.主界面选择7进入函数  2.输入100 | 提示“未找到符合条件的元素” | 输出“未找到符合条件的元素”，符合预期。 |
| 无 | 1.主界面选择7进入函数 | 提示“线性表不存在！请先创建！” | 输出“线性表不存在！请先创建！”，符合。 |

（2）NextElem测试：

表1.3.2-2　NextElem算法测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| Test1 | 1.主界面选择9进入函数  2.输入12 | 输出“该元素后继为0”。 | 输出“该元素后继为0”，符合预期。 |
| 1.主界面选择9进入函数  2.输入100 | 提示“未找到” | 输出“未找到”，符合预期。 |
| 1.主界面选择9进入函数  2.输入94 | 提示“未找到” | 输出“未找到”，符合预期。 |
| Test2 | 1.主界面选择9进入函数  2.输入12 | 提示“未找到” | 输出“未找到”，符合预期。 |
| 无 | 1.主界面选择9进入函数 | 提示“线性表不存在！请先创建！” | 输出“线性表不存在！请先创建！”，符合。 |

（3）ListInsert测试：

表1.3.2-3　ListInsert算法测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| Test1 | 1.主界面选择10进入函数  2.根据提示输入位置，输入1  3.根据提示输入新元素，输入10 | 提示插入成功  使用Traverse函数，得到表10,5,12,0,44,94 | 输出“插入成功！”  使用Traverse函数，输出“10 5 12 0 44 94”，符合预期。 |
| 1.主界面选择10进入函数  2.输入6  3.输入10 | 提示插入成功  使用Traverse函数，得到表5,12,0,44,94,10 | 输出“插入成功！”  使用Traverse函数，输出“5 12 0 44 94 10”，符合预期。 |
| 1.主界面选择10进入函数  2.输入7  3.输入10 | i值不符要求，提示 插入失败 | 输出“插入失败！请检查i值”，符合预期。 |
| 1.主界面选择10进入函数  2.输入0  3.输入10 | i值不符要求，提示 插入失败 | 输出“插入失败！请检查i值”，符合预期。 |
| Test2 | 1.主界面选择10进入函数  2.输入1  3.输入10 | 提示插入成功  使用Traverse函数，得到表10 | 输出“插入成功！”  使用Traverse函数，输出“10”，符合预期。 |
| 1.主界面选择10进入函数  2.输入2  3.输入10 | i值不符要求，提示 插入失败 | 输出“插入失败！请检查i值”，符合预期。 |
| 1.主界面选择10进入函数  2.输入0  3.输入10 | i值不符要求，提示 插入失败 | 输出“插入失败！请检查i值”，符合预期。 |
| 无 | 1.主界面选择10进入函数 | 提示“线性表不存在！请先创建！” | 输出“线性表不存在！请先创建！”，符合。 |

（4）ListDelete测试：(与上类似，故简化)

表1.3.2-4　ListDelete算法测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| Test1 | 1.主界面选择11进入函数  2.根据提示输入位置，输入1 | 提示插入成功  使用Traverse函数，得到表12,0,44,94 | 输出“插入成功！”  使用Traverse函数，输出“12 0 44 94”，符合预期。 |
| 1.主界面选择10进入函数  2.输入0 | i值不符要求，提示 插入失败 | 输出“插入失败！请检查i值”，符合预期。 |

（5）ListTraverse测试：

表1.3.2-5　ListTraverse算法测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| Test1 | 主界面选择12进入函数 | 得到表5,12,0,44,94 | 输出“5 12 0 44 94”，符合预期。 |
| Test2 | 主界面选择12进入函数 | 提示线性表为空 | 输出“线性表是空表”，符合预期。 |
| 无 | 主界面选择12进入函数 | 提示“线性表不存在！请先创建！” | 输出“线性表不存在！请先创建！”，符合。 |

**1.4 实验小结**

本次实验加深了对线性表的概念、基本运算的理解，掌握了线性表的基本运算的实现。熟练了线性表的逻辑结构与物理结构的关系。今后的学习过程中应当多从数据结构的角度分析如何进行数据的处理、存储以方便问题的解决，并要勤加练习达到熟能生巧的地步。

**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**

掌握基于链式存储结构，实现线性表的基本的、常见的运算

**2.2 单链表演示系统设计**

**2.2.1 系统总体设计**

一共有17个基本函数，具体功能如下

（1）InitSqList(SqList \* L)

操作结果：创建链式表组。

（2）InitList(LinkList \* L)

操作结果：构造一个空的链式表

（3）DestroyList(LinkList \* L)

初始条件：链式表L已存在。

操作结果：销毁链式表L。

（4）ClearList (LinkList \* L)

初始条件：链式表L已存在。

操作结果：将L重置为空链式表。

（5）ListEmpty(LinkList \* L)

初始条件：链式表L已存在。

操作结果：若L为空链式表，则返回TRUE,否则返回FALSE.

（6）ListLength(LinkList \* L)

初始条件：链式表已存在。

操作结果：返回L中数据元素的个数。

（7）GetElem(LinkList L,int i,ElemType \* e)

初始条件：单链表已存在，1≤i≤ListLength(L)。

操作结果：用e返回L中第i个结点的数据元素值。

（8）LocateElem(LinkList L,ElemType e, Status (\*compare)(ElemType a, ElemType b))

初始条件：链式表已存在。

操作结果：用e返回L中第i个数据元素的值。

（9）PriorElem（LinkList L,ElemType cur\_e,ElemType \* pre\_e）

初始条件：单链表L已存在。

操作结果：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。

（10）NextElem（LinkList L,ElemType cur\_e,ElemType \* next\_e）

初始条件：单链表L已存在。

操作结果：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。

（11）ListInsert(LinkList \* L,int i,ElemType e)

初始条件：链式表L已存在且非空，1≤i≤ListLength(L)+1。

操作结果：在L的第i个结点之前插入新数据元素e的结点。

（12）ListDelete(LinkList \* L,int i,ElemType \* e)

初始条件：链式表L已存在且非空，1≤i≤ListLength(L)。

操作结果：删除L第i个数据元素的结点，用e返回其结点数据元素的值。

（13）ListTraverse(LinkList L, Status (\*visit)(ElemType a)

初始条件：链式表L已存在。

操作结果：依次对L的每个数据元素调用函数visit()。一旦调用失败，则操

作失败。

（14）ChooseList(LinkList \* L, char s[], SqList L0)

初始条件：链式表L已存在

操作结果：选取已有链式表组中的一组链式表

（15）DeleteList(LinkList \* L, char s[], SqList L0)

初始条件：链式表L存在

操作结果：删除链式表L

（16）ReadList(SqList \* L)

初始条件：文件夹中保存有链式表

操作结果：成功读取链式表L

（17）SaveList(SqList L)

初始条件：已创建链式表L

操作结果：将链式表L保存在创建的文件夹中

**2.2.3 算法设计**

（1）InitSqList(SqList \* L)

创建链表组，参数为各个链式表的表头指针地址，从而实现多个链式表的共同管理。

复杂度：时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（2）InitList(LinkList \* L)

创建空链式表L分三步：1.分配空间 2.将数据域初始化为零 3.将指针域初始化为0。

时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（3）DestroyList(LinkList \* L)

销毁链式表分2步：1.将上一个链式表与下一个链式表连接 2.释放链式表

时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（4）ClearList (LinkList \* L)

将链式表置空，先释放链式表的存储空间，再将上一个与下一个链式表连接，最后将自身指针域置空处理，由此达到重置为空表的目的。

时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（5）ListEmpty(LinkList \* L)

判断链式表是否为空。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（6）ListLength(LinkList \* L)

遍历链表，利用计数器i统计链表元素个数。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（7）GetElem(LinkList L,int i,ElemType \* e)

利用结点p遍历链式表L，再利用计数器j来判断是否到达所需位置i，如果到达，则返回该处元素，达到函数目的。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（8）LocateElem(LinkList L,ElemType e, Status (\*compare)(ElemType a, ElemType b))

利用p指针遍历链表，再运用compare函数进行比较，如果在遍历过程中找到元素e则返回此时计数器i，就是该元素在链表中的位置，由此实现函数功能。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（9）PriorElem（LinkList L,ElemType cur\_e,ElemType \* pre\_e）

参数为链表头指针L, 查找的元素cur\_e，用于返回其前驱的元素pre\_e的地址，Status 成功则返回OK，未找到返回ERROR，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。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（10）NextElem（LinkList L,ElemType cur\_e,ElemType \* next\_e）

参数为链表头指针L，Status 成功则返回OK，未找到返回ERROR，若cur\_e是L的数据元素，且不是第一个，则用next\_e返回它的后驱，否则操作失败，next\_e无定义。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（11）ListInsert(LinkList \* L,int i,ElemType e)

输入参数： 链表头指针地址&L, 插入位置i，插入元素e，在L的第i个结点之前插入新数据元素e的结点。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（12）ListDelete(LinkList \* L,int i,ElemType \* e)

链表头指针L, 删除位置i，用于返回其值的元素e的地址，删除L第i个数据元素的结点，用e返回其结点数据元素的值。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（13）ListTraverse(LinkList L, Status (\*visit)(ElemType a)

链表头指针L, 指向调用函数的函数指针visit，依次对L的每个数据元素调用函数visit()。一旦调用失败，则操作失败。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（14）ChooseList(LinkList \* L, char s[], SqList L0)

单链表头指针地址, 选择的单链表名s,用于管理单链表的顺序表L0，选择指定单链表。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（15）DeleteList(LinkList \* L, char s[], SqList L0)

单链表头指针地址, 选择的单链表名s，用于管理单链表的顺序表L0，删除指定单链表。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（16）ReadList(SqList \* L)

读取链式表。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（17）SaveList(SqList L)

保存链式表

时间复杂度T(n) = O(n)

空间复杂度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**

程序源代码：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

typedef int Status; //status是函数的类型，其值是函数结果状态代码

typedef int ElemType; //数据元素类型定义

#define LIST\_INIT\_SIZE 100 //线性表存储空间的初始分配量

#define LISTINCREMENT 10 //线性表存储空间的分配增量

typedef struct Lnode{ //线性表的单链表存储结构

ElemType data; //单链表的数据域

struct Lnode \* next; //单链表的指针域

}LNode, \*LinkList;

typedef struct

{

char name[20];

LinkList TopLnode;

}Top;

typedef struct{

Top \* elem;

int length;

int listsize;

}SqList;

Status InitSqList(SqList \* L); //创建链式表组

Status InitList(LinkList \* L); //构造一个空的链式表

Status DestroyList(LinkList \* L); //销毁线性表L

Status ClearList(LinkList \* L); //将L重置为空表

Status ListEmpty(LinkList L); //若L为空表，则返回1,否则返回0

int ListLength(LinkList L); //返回L中数据元素的个数

Status GetElem(LinkList L,int i,ElemType \* e); //用e返回L中第i个数据元素的值

int LocateElem(LinkList L,ElemType e, Status (\*compare)(ElemType a, ElemType b));

Status PriorElem(LinkList L,ElemType cur\_e,ElemType \* pre\_e); /\*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 NextElem(LinkList L,ElemType cur\_e,ElemType \* next\_e); /\*若cur\_e是L的

数据元素，且不是最后一个，则用next\_e返回的后继，否则操作失败，next\_e无定义\*/

Status ListInsert(LinkList \* L,int i,ElemType 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\*/

Status ListDelete(LinkList \* 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\*/

Status ListTraverse(LinkList L, Status (\*visit)(ElemType a)); /\*依次对L的每个

数据元素调用函数visit()。一旦调用失败，则操作失败\*/

Status ChooseList(LinkList \* L, char s[], SqList L0); //选取已有链表中一个链式表

Status DeleteList(LinkList \* L, char s[], SqList L0); //删除一个链式表

Status ReadList(SqList \* L); //读取链式表

Status SaveList(SqList L); //保存链式表

Status Visit(ElemType a);

Status Compare(ElemType a, ElemType b);

Status InitSqList(SqList \* L) //参数： 管理各单链表的顺序表头指针地址

{

L->elem = (Top \*)malloc(LIST\_INIT\_SIZE \* sizeof(Top));

if(!L->elem)

{

exit(OVERFLOW);

}

L->length=0; //空表长度为0

L->listsize=LIST\_INIT\_SIZE; //初始存储容量

return OK;

}

Status InitList(LinkList \* L) //构造一个空的链式表

{

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

if(\*L == NULL)

{

exit(OVERFLOW);

}

(\*L)->data = 0; //空表长度为0

(\*L)->next = NULL; //将指针域初始化为0

return OK;

}

Status DestroyList(LinkList \* L)

{

LinkList p,q;

p= \*L; // p指向头结点

while(p) //遍历没到表尾

{

q=p->next; //将上一个链式表与下一个链式表连接

free(p);

p=q;

}

\*L = NULL;

return OK;

}

Status ClearList(LinkList \* L)

{

LinkList p,q;

p=(\*L)->next; //p指向第一个结点

while(p) //遍历没到表尾

{

q=p->next; //将上一个与下一个链式表连接

free(p);

p=q;

}

(\*L)->next=NULL; //头结点指针域为空

return OK;

}

Status ListEmpty(LinkList L) //判断链式表是否为空

{

if(L->next)

{

return FALSE;

}

else

{

return TRUE;

}

}

int ListLength(LinkList L)

{

int i=0;

LinkList p=L->next; //p指向第一个结点

while(p) //当p没到表尾

{

i++;

p=p->next;

}

return i;

}

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

{

int j = 1; //j为计数器

LinkList p; //声明一结点p

p = L->next; //让p指向链表L的第一个结点

while (p && j<i) //p不为空或者计数器j还没有等于i时，循环继续

{

p = p->next; //让p指向下一个结点

++j;

}

if ( !p || j>i )

{

return ERROR; //第i个元素不存在

}

\*e = p->data; //用e取第i个元素

return OK;

}

int LocateElem(LinkList L,ElemType e, Status (\*compare)(ElemType a, ElemType b))

{

int i=0;

LinkList p=L->next; //p初始为头指针

while(p)

{

i++;

if((\*compare)(p->data, e)) //找到这样的数据元素

{

return i;

}

p=p->next;

}

return 0;

}

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

{

LinkList p = L->next;

while(p->next != NULL && p->next->data != cur\_e)

{

p = p->next;

}

if (p->next == NULL) //未找到对应元素的前驱

{

return ERROR;

}

\*pre\_e = p->data;

return OK;

}

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

{

LinkList p = L->next;

while(p->next != NULL && p->data != cur\_e)

{

p = p->next;

}

if (p->next == NULL) //到最后一个仍未找到

{

return ERROR;

}

\*next\_e = p->next->data;

return OK;

}

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

{

int j = 1;

LinkList p,new;

p = \*L;

while (p && j < i) //寻找第i个结点

{

p = p->next;

++j;

}

if (!p || j > i) //第i个元素不存在

{

return ERROR;

}

new = (LinkList)malloc(sizeof(LNode)); //生成新结点

if (new == NULL)

{

exit(OVERFLOW);

}

new->data = e;

new->next = p->next; //将p的后继结点赋值给new的后继

p->next = new; //将new赋值给p的后继

return OK;

}

Status Compare(ElemType a, ElemType b)

{

if (a == b)

{

return TRUE;

}

else

{

return FALSE;

}

}

Status Visit(ElemType a)

{

printf("%d ",a);

return OK;

}

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

{

int j = 1;

LinkList p,q;

p = \*L;

while (p->next && j < i) //寻找第i个元素

{

p = p->next;

++j;

}

if (!(p->next) || j > i) //第i个元素不存在

{

return ERROR;

}

q = p->next;

p->next = q->next; //将q的后继赋值给p的后继

\*e = q->data; //将q结点中的数据给e

free(q); //释放内存

return OK;

}

Status ListTraverse(LinkList L, Status (\*visit)(ElemType a))

{

LinkList p = L->next;

if (!p)

{

return ERROR;

}

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

while(p)

{

if (!(\*visit)(p->data))

{

printf("操作失败！");

}

p=p->next;

}

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

return OK;

}

Status ChooseList(LinkList \* L, char s[], SqList L0)

{

int i;

for (i = 0; i < L0.length; i++)

{

if (strcmp(L0.elem[i].name, s) == 0)

{

\*L = L0.elem[i].TopLnode;

return OK;

}

}

return ERROR;

}

Status DeleteList(LinkList \* L, char s[], SqList L0)

{

int i, j;

for (i = 0; i < L0.length; i++)

{

if (strcmp(L0.elem[i].name, s) == 0)

{

//若当前单链表为删除的链表，将单链表置空

if (\*L = L0.elem[i].TopLnode)

{

ClearList(L);

}

//删除位置之后的单链表前移

for (j = i; j < L0.length - 1; j++)

{

L0.elem[j].TopLnode = L0.elem[j+1].TopLnode;

strcpy(L0.elem[j].name,L0.elem[j+1].name);

}

L0.elem[j].TopLnode = NULL;

strcpy(L0.elem[j].name, "无");

L0.length--;

return OK;

}

}

return ERROR;

}

Status ReadList(SqList \* L)

{

int i, num;

InitSqList(L);

FILE \*fp;

char filename[30];

printf("请输入读取文件名: ");

scanf("%s", &filename);

if ((fp = fopen(filename,"r")) == NULL)

{

printf("文件打开失败！\n ");

return ERROR;

}

while(fread(L->elem[L->length].name, sizeof(char), 20, fp))

{

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

L->elem[L->length].TopLnode = l;

fread(&num, sizeof(int), 1, fp);

for (i = 0; i <= num; i++)

{

fread(&l->data, sizeof(ElemType), 1, fp);

l->next = (i != num)?(LinkList)malloc(sizeof(LNode)):NULL;

l = l->next;

}

L->length++;

}

fclose(fp);

return OK;

}

Status SaveList(SqList L) //保存线性表

{

int i;

FILE \*fp;

char filename[30];

printf("请输入保存文件名： ");

scanf("%s",&filename);

//写文件的方法

if ((fp = fopen(filename, "w")) == NULL)

{

printf("文件打开失败\n ");

return ERROR;

}

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

{

int num;

fwrite(L.elem[i].name, sizeof(char), 20, fp);

LinkList l = L.elem[i].TopLnode;

num = ListLength(l);

fwrite(&num, sizeof(int), 1, fp);

int j;

for (j = 0; j <= num; j++)

{

fwrite(l, sizeof(ElemType), 1, fp);

l = l->next;

}

}

fclose(fp);

return OK;

}

**2.3.2 系统测试**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| 1.文件名 | M文件和S文件 | 当退出系统时会将数据储存在建立的文件中 | 文件夹中出现M文件和S文件 |
| 2.创建链式表L1、L2、L3 | L1、L2储存在M文件中，L3储存在S文件中 | 当执行13,：读取文件夹M时，再选取其中链式表L1即可成功选取 | 成功选取链式表L1 |
| 3.向链式表L1输入数据 | 操作序号10，依次输入位置及元素：1,3,5,7 | 输出L1中元素时成功输出1,3,5,7 | 成功输出1,3,5,7 |
| 4.返回链式表L1中第三个元素 | 操作序号6，再输入位置3 | 屏幕显示输出5 | 成功输出5 |
| 5.判断链式表L1是否为空 | 操作序号4 | L1不为空 | 测试成功 |
| 6.返回链式表L1中数据8的序号 | 操作序号7，再输入元素5 | 8在链式表L1中序号为3 | 测试成功 |
| 7.求链式表L1中元素8的前一个元素 | 操作序号8，  再输入元素5 | 返回元素5的前一个元素3 | 测试成功 |
| 8. 求链式表L1中元素8的后一个元素 | 操作序号9，  再输入元素5 | 返回元素5的后一个元素7 | 测试成功 |
| 9.:链式表L1中插入第5个位置64 | 操作序号10，  再输入位置5以及元素64 | 插入成功，以及输出链式表L1的元素时也会出现64 | 测试成功 |
| 10.删除线链式表L  中第5个元素64 | 操作序号11，再输入位置5 | 删除成功，以及输出链式表L1的元素时64消失 | 测试成功 |
| 11.选取链式表L2进行操作 | 操作序号15，再输入链式表名称L2 | 后续操作即可为链式表L2进行操作 | 测试成功 |

**2.4 实验小结**

相对第一次上机感觉这次的内容要复杂一些，对链表的操作要比对数组的操作更具难度，刚开始不知道怎么入手，问过同学后才知道该怎么对多个链表同时进行操作，遇到很多不懂的又翻看课本，最终都找到解决办法，总之这次上机还是获益匪浅的，学会用链式结构对线性表进行运算等操作，但自己的编程能力确实还有待加强，学得还不够深入，使得每次上机都变得困难重重。以后要努力改进，争取做得更好。

**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**

完成栈的创建等基本操作，通过栈进行简单的个位数的四则运算。

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。

(5) 通过栈实现简单的个位数的四则运算

栈的简介：栈是一种数据结构，是一种只能在一端进行插入和删除操作的特殊线性表。它按照后进先出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。栈具有记忆作用，对栈的插入与删除操作中，不需要改变栈底指针。

栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶(top)，另一端为栈底(base)；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈（PUSH），删除则称为退栈（POP）。栈也称为后进先出表。

**3.1.2 顺序栈设计**

（1）InitStack(&S)

操作结果：构造一个空栈S。

函数代码：

status InitStack(SqStack \*S){

//构造一个空栈S

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

if(!S->base) {printf("初始化栈失败");exit(OVERFLOW);}//存储分配失败

S->top=S->base;

S->stacksize=STACK\_INIT\_SIZE;

return OK;

}//InitStack



（2）DestroyStack(&S)

初始条件：栈S存在。

操作结果：栈S被销毁，不在存在。

函数代码：

status DestroyStack(SqStack \*S){

//销毁栈S，S不再存在

free(S->base);//释放栈底指针

S->base=NULL;//栈底指针指向空。代表栈不存在

S->top=NULL;//栈顶指针也指向空

S->stacksize=0;//销毁栈时把栈的长度改为0

return OK;

}



（3）ClearStack(&S)

初始条件：栈S存在。

操作结果：将S清成空栈。

函数代码：

status ClearStack(SqStack \*S){

//把S置为空栈

S->top=S->base;

return OK;

}



（4）StackEmpty(S)

初始条件：栈S存在。

操作结果：若S为空栈,则返回值为TRUE；否则为FALSE。

函数代码：

status StackEmpty(SqStack \*S){

//若栈S为空栈，则返回TURE，否则返回FALSE

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

else return FALSE;

}



（5）StackLength(S)

初始条件：栈S存在。

操作结果：返回栈S的元素个数。

函数代码：

status StackLength(SqStack \*S){

//返回S的元素个数，即栈的长度

return S->top-S->base;

}



（6）GetTop(S,&e)

初始条件：栈S存在并且非空。

操作结果：将栈顶元素拷贝到e。

若S为空栈，则结束拷贝。

函数代码：

status GetTop(SqStack S,SElemType \*e){

//若栈不为空，则用e返回S的栈顶元素，并返回OK，否则返回ERROR

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

e=\*(S.top-1);//选取栈顶元素e

return OK;

}



（7）Push(&S,e)

初始条件：栈S存在。若S已满，则发生溢出。若不能解决溢出，重新分配空间失败，则插入失败。

操作结果：插入元素e为新的栈顶元素。

函数代码：

status Push(SqStack \*S,SElemType e){

//插入元素e为新的栈顶元素

if(S->top-S->base>=S->stacksize){//栈满，追加存储空间

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

if(!S->base) exit(OVERFLOW);//存储分配失败

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

}

\*S->top++=e;

return OK;

}//Push



（8）Pop(&S,&e)

初始条件：栈S存在并且非空。

操作结果：删除栈S的栈顶元素，并送入e。

若S为空栈，发生“下溢”（underflow）；

为空栈时，表示某项任务已完成。

函数代码：

status Pop(SqStack \*S){

//若栈不空，则删除S的栈顶元素，返回其值，否则返回ERROR

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

return (\*--S->top);

}//Pop



（9）StackTravserse(S,visit())

初始条件：栈S存在。

操作结果：从栈底到栈顶依次对栈S中的元素使用函数visit进行访问。

函数代码：

int StackTravserse(SqStack S,int(\*visit)(SElemType ))

{//从栈底到栈顶依次对栈中的每个元素调用函数visit()。一旦visit()失败，则操作失败

while(S.top>S.base)

visit(\*S.base++);

return OK;

}



**3.1.2 顺序栈实现与测试**

1. 实现部分

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\*函数结果状态代码\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

typedef int Status; //status是函数的类型，其值是函数结果状态代码

typedef int SElemType; //数据元素类型定义

#define STACK\_INIT\_SIZE 100 //存储空间的初始分配量

#define STACKINCREMENT 10 //存储空间的分配增量

typedef struct {

SElemType \* base;

SElemType \* top;

int stacksize;

}SqStack;

Status InitStack (SqStack \*S);

//构造一个空栈S

Status DestroyStack (SqStack \*S);

//销毁栈S,S不再存在

Status ClearStack (SqStack \*S);

//把S置为空栈

Status StackEmpty (SqStack S);

//若栈S为空栈，则返回TRUE，否则返回error

int StackLength (SqStack S);

//返回S的元素个数，即栈的长度

Status GetTop (SqStack S,SElemType \*e);

//若栈不空，则用e返回S的栈顶元素，并返回OK，否则返回error

Status Push (SqStack \*S,SElemType e);

//插入元素e为新的栈顶元素

Status Pop (SqStack \*S,SElemType \*e);

//若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK，否则返回error

Status StackTraverse (SqStack S,Status (\* visit)());

//从栈底到栈顶依次对栈中每个元素调用函数visit()。一旦visit（）失败，则操作失败

Status ReadStack (SqStack \* S);

//读取栈操作

Status SaveStack (SqStack S);

//保存栈操作

Status Visit (SElemType a);

//显示a

Status In (char c,char op[]);

//判断输入的某个字符是否是运算符

char Precede (char a, char b);

//比较两个运算符的优先级

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

int EvaluateExpression();

Status InitStack (SqStack \*S) {

//构造一个空栈

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

if(!S->base) exit (OVERFLOW); //存储分配失败

S->top = S->base;

S->stacksize = STACK\_INIT\_SIZE;

return OK;

} //InitStack

Status DestroyStack (SqStack \*S) {

S->top = NULL;

S->stacksize = 0;

free (S->base);

return OK;

} //DestroyStack

Status ClearStack (SqStack \*S) {

S->top = S->base;

return OK;

}

Status StackEmpty (SqStack S) {

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

else return ERROR;

}

int StackLength (SqStack S) {

return (S.top - S.base);

}

Status GetTop (SqStack S,SElemType \*e) {

//若栈不空，则用e返回S的栈顶元素，并返回OK，否则返回error

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

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

return OK;

} //GetTop

Status Push (SqStack \*S,SElemType e) {

//插入e为新的栈顶元素

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

//栈满，追加存储空间

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

if (!S->base) exit (OVERFLOW); //存储分配失败

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

S->stacksize += STACKINCREMENT;

}

\*S->top++ = e;

return OK;

} //Push

Status Pop (SqStack \*S,SElemType \*e) {

//若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK,否则返回ERROR

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

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

return OK;

} //Pop

Status StackTraverse (SqStack S,Status (\* visit)(SElemType a)){

while (S.top > S.base) {

visit (\*S.base++);

}

return OK;

}

Status ReadStack (SqStack \* S){

int i = 0;

if ((\*S).base != NULL) DestroyStack(S);

InitStack(S);

FILE \*fp;

char filename[30];

printf("请输入文件名：");

scanf ("%s",&filename);

if ((fp = fopen(filename,"r")) == NULL){

printf("文件打开失败！\n");

return ERROR;

}

while (fread(&(S->base[i]),sizeof(SElemType),1,fp)) i++;

S->top = &(S->base[i]);

fclose(fp);

return OK;

}

Status SaveStack(SqStack S){

FILE \*fp;

char filename[30];

printf("请输入保存文件名：");

scanf("%s",&filename);

if ((fp = fopen(filename,"w")) == NULL){

printf("文件打开失败！\n");

return ERROR;

}

fwrite(S.base,sizeof(SElemType),S.top - S.base,fp);

fclose(fp);

return OK;

}

Status Visit(SElemType a){

printf("%d",a);

return OK;

}

Status In(char c,char op[]){

char \*p;

p = op;

while (\*p != '\0'){

if(c == \*p)return OK;

p++;

}

return ERROR;

}

char Precede(char a,char b){

int i,j;

char pre[][7]={

/\*运算符之间的优先级制作成一张表格\*/

{'>','>','<','<','<','>','>'},

{'>','>','<','<','<','>','>'},

{'>','>','>','>','<','>','>'},

{'>','>','>','>','<','>','>'},

{'<','<','<','<','<','=','0'},

{'>','>','>','>','0','>','>'},

{'<','<','<','<','<','0','='}

};

switch(a)

{

case '+': i=0; break;

case '-': i=1; break;

case '\*': i=2; break;

case '/': i=3; break;

case '(': i=4; break;

case ')': i=5; break;

case '#': i=6; break;

}

switch(b)

{

case '+': j=0; break;

case '-': j=1; break;

case '\*': j=2; break;

case '/': j=3; break;

case '(': j=4; break;

case ')': j=5; break;

case '#': j=6; break;

}

return pre[i][j];

}

int Operate(int a,char theta,int b){

int i,result;

switch(theta){

case '+': result = a + b; break;

case '-': result = a - b; break;

case '\*': result = a \* b; break;

case '/': result = a / b; break;

}

return result;

}

int EvaluateExpression()

{

int x, theta;

char c;

int a, b,final;

char OP[]="+-\*/()#"; //运算符集合

SqStack OPTR; //运算符栈

SqStack OPND; //运算数栈

InitStack(&OPTR);

Push(&OPTR,'#'-'0');

InitStack(&OPND);

c = getchar();

while(GetTop(OPTR, &x), c != '#' || x != '#'-'0')

{

if (!In(c, OP))

{

Push(&OPND, c-'0');

c = getchar();

}

else

{

GetTop(OPTR, &x);

switch(Precede(x+'0', c))

{

case '<': //栈顶元素优先级低

Push(&OPTR,c-'0');

c = getchar();

break;

case '=': //脱括号并接受下一字符

Pop(&OPTR,&x);

c = getchar();

break;

case '>': //退栈并将运算结果入栈

Pop(&OPTR, &theta);

Pop(&OPND,&b);

Pop(&OPND,&a);

Push(&OPND, Operate(a, theta+'0', b));

break;

}

}

}

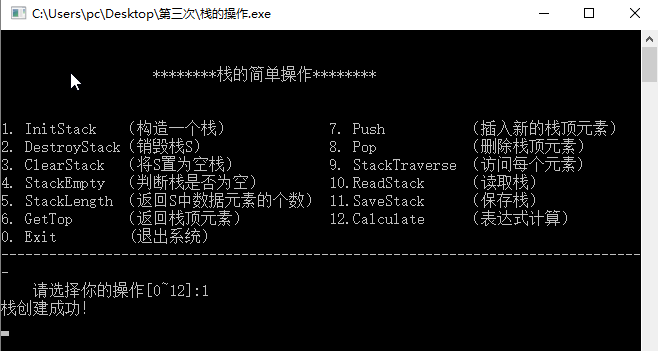
GetTop(OPND, &final);

return (final);

}

2. 测试部分

构造一个栈：

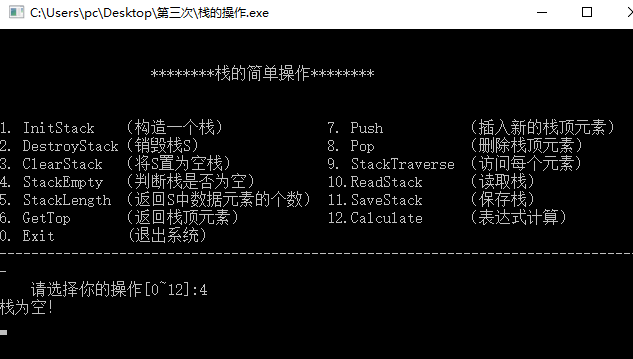


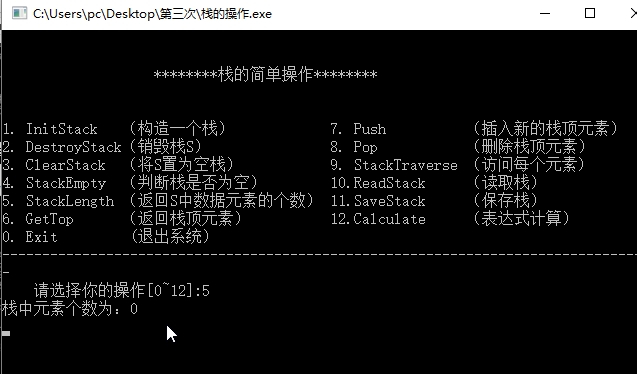
销毁栈：



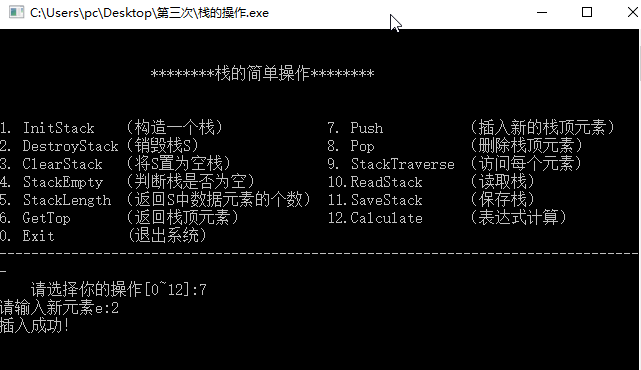
置为空栈：

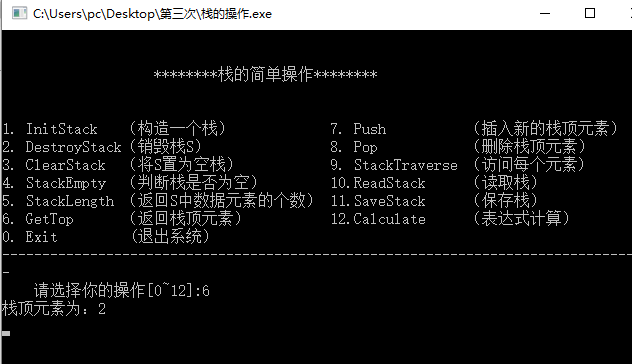


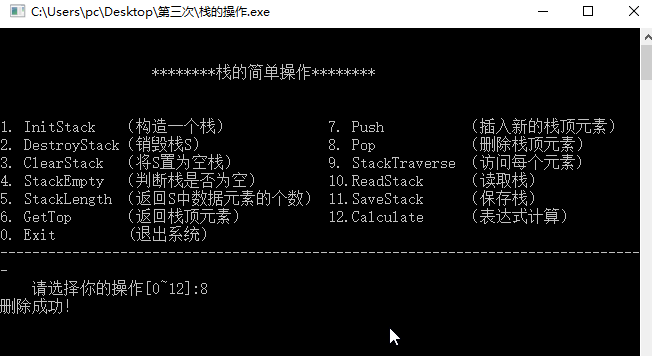
判断是否为空：  


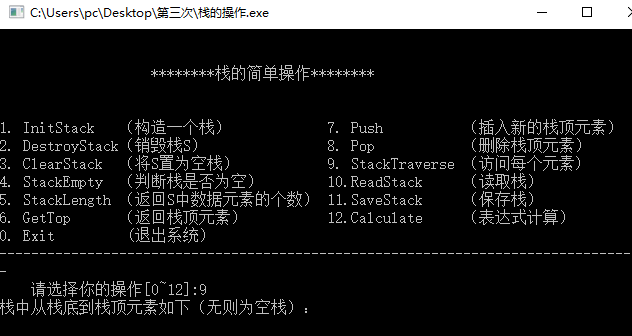
返回栈中元素个数：  


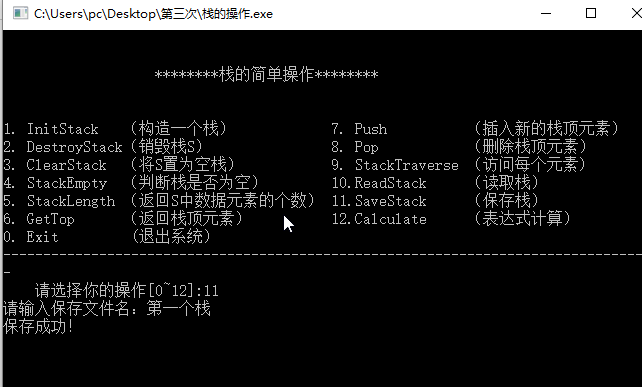
插入栈顶元素：插入2

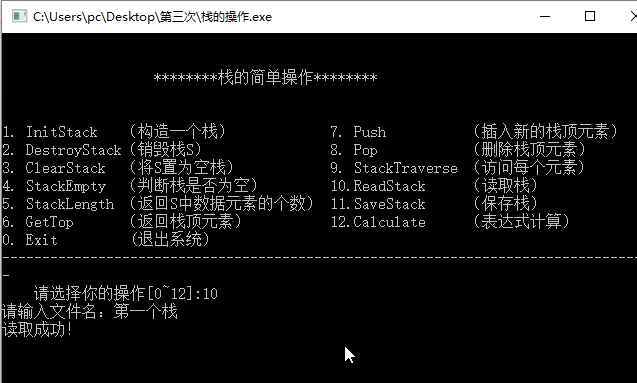


返回栈顶元素：  


删除栈顶元素：  


访问每个元素：无元素  


保存栈：文件名：第一个栈  


读取栈：  


**3.2 表达式求值实验**

**3.2.1 问题描述**

利用两个栈实现一位数的简单四则运算。

**3.2.2算法设计部分**

为了实现算符优先算法。可以使用两个工作栈。一个称为OPTR，用以寄存运算符，另一个称做OPND，用以寄存操作数或运算结果。

1.首先置操作数栈为空栈，表达式起始符”#”为运算符栈的栈底元素，而#也表示着表达式求值开始；

2.依次读入表达式，若是操作符即进OPND栈，若是运算符则和OPTR栈的栈顶运算符比较优先权后作相应的操作，具体分为三种情况：

（1）栈顶运算符优先级高时，先将OPTR栈中弹出栈顶运算符，同时OPND栈中弹出两个操作数，并将运算结果存入OPND栈中。

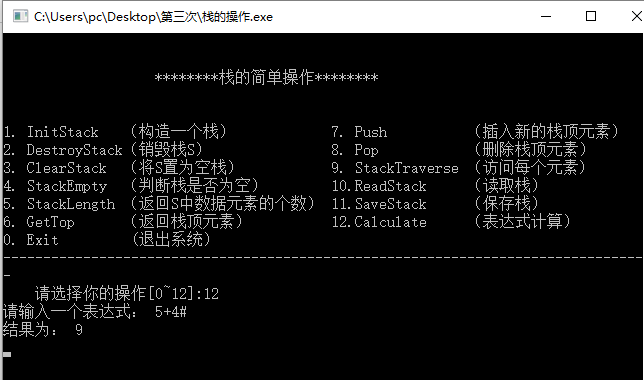
（2）优先关系相等。此时仅有两种可能，栈顶为左括号‘（’或者‘#’。若为左括号‘（’，运算时直接做去除括号‘（）‘处理；若为‘#’，则代表表达式求值结束。

（3）栈顶运算符优先级低时，将运算符压入栈OPTR中。

**3.2.3实现与测试部分**

1. 测试部分

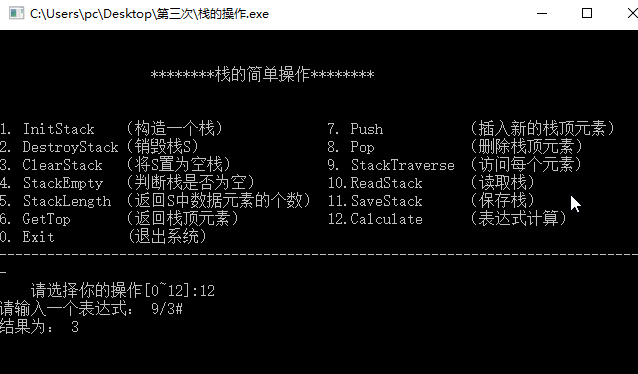
5+4=9：



5-4=1：  


5\*4=20：



9/3=3：  


**3.3 实验小结**

总体来说这次实验除了后半部分表达式求值比较困难之外，还是比较简单的，前半部分顺序储存结构实现栈的操作较之前两次实验中顺序存储结构和链式存储结构的创建删除等操作并无太大区别，以及文件的读取和保存操作都差不多相同，但后半部分难度确实存在，开始不知道怎么写优先级比较的函数，百度后明白一些，又问了同学，才搞清楚，这次实验收获还是蛮大的，学会了如何实现栈的简单基本操作，并且学会用两个栈实现简单的一位数的四则运算，受益匪浅，但还是要多加练习，达到熟能生巧的地步，后面还有很长的路要走。

**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

用循环队列储存结构实现队列的基本运算，实验要求完成的算法：

（1）InitQueue(&Q)

操作结果：构造一个空队列Q。

（2） DestroyQueue(&Q)

初始条件：队列Q存在。

操作结果：将队列Q销毁，不再存在。

（3）ClearQueue(&Q)

初始条件：队列Q存在。

操作结果：将队列Q清为空队列。

（4）QueueEmpty(Q)

初始条件：队列Q存在。

操作结果：若队列Q为空队列，返回TRUE，否则返回FALSE。

（5）QueueLength(Q)

初始条件：队列Q存在。

操作结果：返回队列元素个数。

（6）GetHead(Q,&e)

初始条件：队列Q存在并且非空。

操作结果：读取队列Q的首元素,送e返回其值。

（7）EnQueue(&Q,e)

初始条件：队列Q存在。

操作结果：插入元素e到队列Q中作为尾元素。

（8）DeQueue(&Q,&e)

初始条件：队列Q存在并且非空。

操作结果：删除队列Q的首元素,并且用e返回其值。

（9）QueueTravserse(Q,visit())

初始条件：队列Q存在并且非空。

操作结果：从队首到队尾依次对队列中的元素使用函数visit进行访问。

实验目标：（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。

4.2 队列基本操作算法设计：

1. InitQueue(SqQueue \* Q)

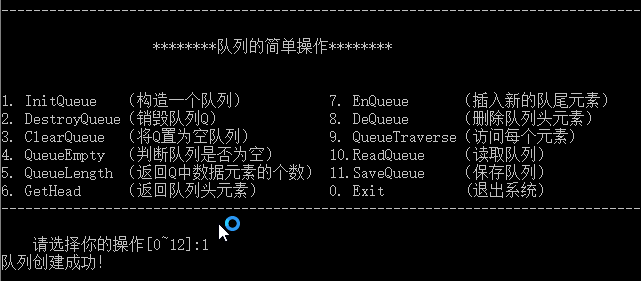
函数名称： InitQueue

输入参数： SqQueue Q的地址

返回值： Status 成功则返回OK ，分配不成功exit(OVERFLOW)，

函数功能： 构造一个空队列

测试结果：



2. DestroyQueue(SqQueue \* Q)

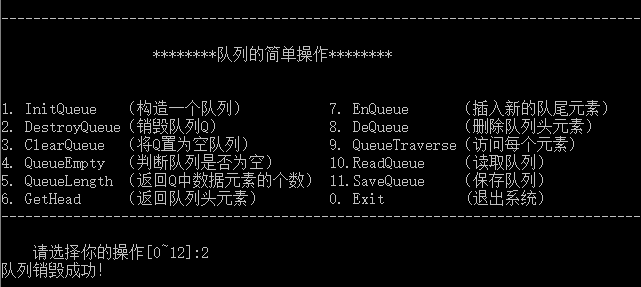
函数名称： DestroyQueue

输入参数： SqQueue Q的地址

返回值： Status 成功则返回OK

函数功能： 将队列Q销毁

测试结果：



3. ClearQueue(SqQueue \* Q)

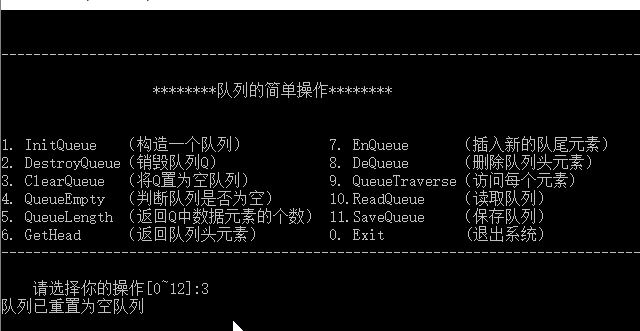
函数名称： ClearQueue

输入参数： SqQueue Q的地址

返回值： Status 成功则返回OK

函数功能： 将队列Q清为空队列

测试结果：



4. QueueEmpty(SqQueue Q)

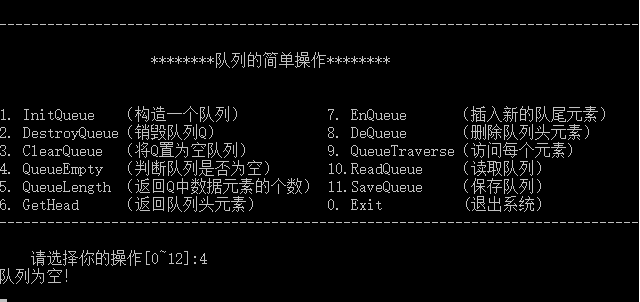
函数名称： QueueEmpty

输入参数： 循环队列Q

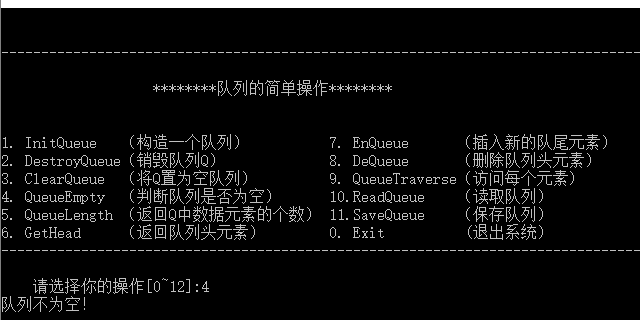
返回值： Status 若队列Q为空队列，返回TRUE，否则返回FALSE

函数功能： 判断队列Q是否空队列

测试结果：



插入元素2后：



5. QueueLength(SqQueue Q);

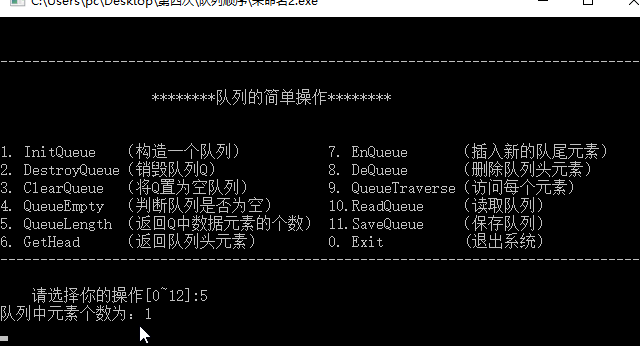
函数名称： QueueLength

输入参数： 循环队列Q

返回值： int 队列元素个数

函数功能： 返回队列元素个数

测试结果：



6. GetHead(SqQueue Q, QElemType \* e)

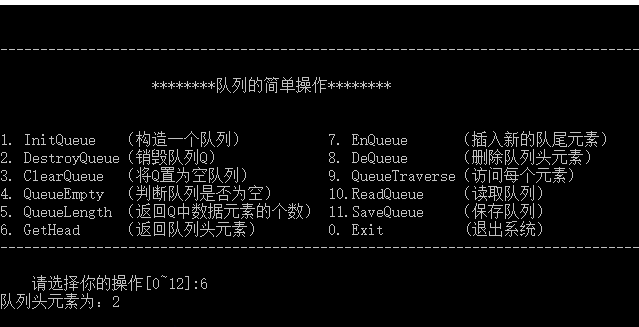
函数名称： GetHead

输入参数： 循环队列Q ，用于返回首元素的元素e的地址

返回值： Status 成功则返回OK

函数功能： 读取队列Q的首元素,用e返回其值

测试结果：



7. EnQueue(SqQueue \* Q, QElemType e)

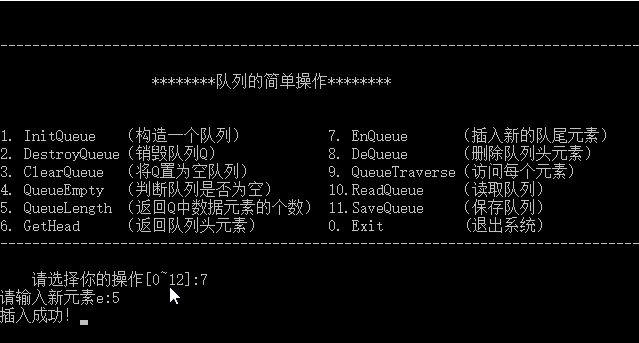
函数名称： EnQueue

输入参数： 循环队列Q地址，新增元素e

返回值： Status 成功则返回OK

函数功能： 插入元素e为Q的新队尾元素

测试结果（插入元素5）：



8. DeQueue(SqQueue \* Q, QElemType \* e)

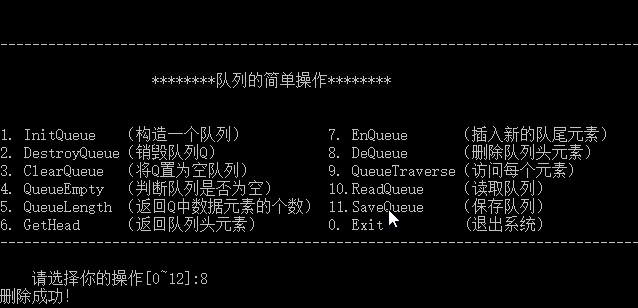
函数名称： DeQueue

输入参数： 循环队列Q地址，用于返回队头元素的元素e的地址

返回值： Status

函数功能： 若队列不空，则删除Q的队头元素，用e返回其值，并返回OK；否则返回ERROR

测试结果：



9. QueueTraverse(SqQueue Q, Status (\*visit)(QElemType a))

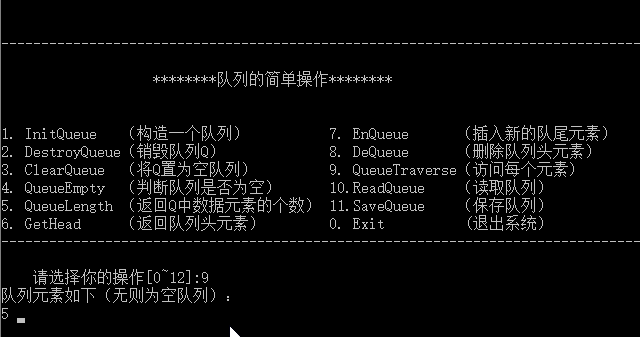
函数名称：QueueTraverse

输入参数： 循环队列Q, 指向调用函数的函数指针visit

返回值： Status

函数功能： 从队首到队尾依次对队列中的元素使用函数visit进行访问

测试结果：



10. Visit(QElemType a)

函数名称： Visit

输入参数： QElemType a

返回值： Status

函数功能： 输出a

11. ReadQueue(SqQueue \* Q)

函数名称：ReadQueue

输入参数： 队列Q的地址

返回值： Status

函数功能： 读取队列

12. SaveQueue(SqQueue Q)

函数名称：SaveList

输入参数：队列Q

返回值： Status

函数功能： 保存队列

程序清单：

#include <stdio.h>

#include <string.h>

/\*函数结果状态代码\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

typedef int Status; //status是函数的类型，其值是函数结果状态代码

typedef int QElemType; //数据元素类型定义

typedef struct {

QElemType \*base; //初始化的动态分配存储空间

int front; //头指针，若队列不空，指向队列头元素

int rear; //尾指针，若队列不空，指向队列列尾元素的下一个位置

}SqQueue;

#define MAXQSIZE 100 //最大队列长度

Status InitQueue (SqQueue \*Q);

//构造一个空队列Q

Status DestroyQueue (SqQueue \*Q);

//销毁队列Q,Q不再存在

Status ClearQueue (SqQueue \*Q);

//把Q清为空队列

Status QueueEmpty (SqQueue Q);

//若队列Q为空队列，则返回TRUE，否则返回error

int QueueLength (SqQueue Q);

//返回Q的元素个数，即队列的长度

Status GetHead (SqQueue Q,QElemType \*e);

//若队列不空，则用e返回Q的队头元素，并返回OK，否则返回error

Status EnQueue (SqQueue \*Q,QElemType e);

//插入元素e为新的队尾元素

Status DeQueue (SqQueue \*Q,QElemType \*e);

//若队列不空，则删除Q的队头元素，用e返回其值，并返回OK，否则返回error

Status QueueTraverse (SqQueue Q,Status (\* visit)());

//从队头到队尾依次对队列Q中每个元素调用函数visit()。一旦visit（）失败，则操作失败

Status ReadQueue (SqQueue \*Q);

//读取队列操作

Status SaveQueue (SqQueue Q);

//保存队列操作

Status Visit (QElemType a);

//显示a

Status InitQueue (SqQueue \*Q) {

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

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

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

return OK;

}

Status DestroyQueue (SqQueue \*Q) {

free(Q->base);

Q->base = NULL;

Q->front = 0;

Q->rear = 0;

return OK;

}

Status ClearQueue (SqQueue \*Q) {

Q->front = 0;

Q->rear = 0;

return OK;

}

Status QueueEmpty (SqQueue Q) {

if ((Q.front == 0)&&(Q.rear == 0)) return OK;

else return ERROR;

}

int QueueLength (SqQueue Q) {

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

}

Status GetHead (SqQueue Q,QElemType \*e) {

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

return OK;

}

Status EnQueue (SqQueue \*Q,QElemType e) {

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

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

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

return OK;

}

Status DeQueue (SqQueue \*Q,QElemType \*e) {

if (Q->front == Q->rear)return ERROR;

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

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

return OK;

}

Status QueueTraverse (SqQueue Q,Status (\* visit)()) {

int i = Q.front;

while (i != Q.rear)

{

if (!(\*visit)(Q.base[i]))

{

printf("操作失败！");

}

i = (i + 1) % MAXQSIZE;

}

return (Q.front - Q.rear);

}

Status ReadQueue (SqQueue \*Q) {

if ((\*Q).base != NULL)

{

DestroyQueue(Q);

}

InitQueue(Q);

FILE \*fp;

char filename[30];

printf("请输入读取文件名: ");

scanf("%s", &filename);

if ((fp = fopen(filename,"r")) == NULL)

{

printf("文件打开失败！\n ");

return ERROR;

}

while(fread(&(Q->base[Q->rear]), sizeof(QElemType), 1, fp))

Q->rear++;

fclose(fp);

return OK;

}

Status SaveQueue (SqQueue Q) {

FILE \*fp;

int i, j;

char filename[30];

printf("请输入保存文件名： ");

scanf("%s",&filename);

//写文件的方法

if ((fp = fopen(filename, "w")) == NULL)

{

printf("文件打开失败\n ");

return ERROR;

}

j = Q.front;

for (i = 1; i <= (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE; i++)

{

fwrite(&(Q.base[j]), sizeof(QElemType), 1, fp);

j = (j + 1) % MAXQSIZE;

}

fclose(fp);

return OK;

}

Status Visit (QElemType a) {

printf("%d ",a);

return OK;

}

4.3 实验小结

总体来说这次实验比较简单，课上就已经能够完成，这次实验收获还是蛮大的，学会了如何实现队列这种先进先出的储存结构的简单基本操作，使用循环队列储存结构实现队列的基本运算，受益匪浅，但还是要多加练习，达到熟能生巧的地步，任重道远，只有多加练习才能更上一层楼！

**5基于二叉链表实现二叉树的基本运算**

## 5.1实验内容与要求

基于二叉链表，通过实验加深对二叉树的概念、基本运算的理解；熟练掌握二叉树的逻辑结构与物理结构的关系；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。

## 5.2程序概要设计

1.明确基本功能

程序需要实现的20个基本功能分别是：InitBiTree(构造空二叉树)，DestroyBiTree(销毁二叉树)，CreateBiTree(先序创建二叉树)，ClearBiTree(清空二叉树)，BiTreeEmpty(判断二叉树是否为空)，BiTreeDepth (求二叉树的深度)，Root(返回二叉树的根)，Value(返回结点的值)，Assign(给结点赋值)，Parent(求双亲结点)，LeftChild(求结点左孩子)，RightChild(求结点右孩子)，LeftSibling(求结点左兄弟)，RightSiling(求结点右兄弟)，InsertChild(插入子树)，DeleteChild(删除子树)，PreOrderTraverse(先序遍历)，InOrderTrabverse(中序遍历)，PostOrderTrabverse(后序遍历)，LevelOrderTrabverse(层序遍历)。

2.确定各功能实现的函数参数

Status InitBiTree(BiTree \* T); //构造空二叉树T

Status DestroyBiTree(BiTree \* T); //二叉树T已存在,销毁二叉树T

void CreateBiTree(BiTree \* T); //先序构造二叉树T

Status ClearBiTree(BiTree \* T); //清空二叉树

Status BiTreeEmpty(BiTree T);

//初始条件：二叉树T存在，若T为空二叉树，则返回TRUE,否则返回FALSE.

Status BiTreeDepth(BiTree T); //求二叉树深度

BiTNode Root(BiTree T); //返回二叉树的根

Status Value(BiTree T, BiTree e); //返回二叉树结点的值

Status Assign(BiTree T, BiTree e, ElemType value); //给二叉树结点赋值

ElemType Parent(BiTree T,ElemType e); //求双亲结点

ElemType LeftChild(BiTree T,ElemType e); //求左孩子结点

ElemType RightChild(BiTree T,ElemType e); //求右孩子结点

ElemType LeftSibling(BiTree T,ElemType e); //求左兄弟结点

ElemType RightSibling(BiTree T,ElemType e); //求右兄弟结点

Status InsertChild(BiTree T,BiTree p,int LR,BiTree c);//插入子树

Status DeleteChild(BiTree T,BiTree p,int LR); //删除子树

Status PreOrderTraverse(BiTree T, Status (\* Visit)(ElemType e)); //先序遍历二叉树

Status InOrderTrabverse(BiTree T,Status (\* Visit)(ElemType e)); //中序遍历二叉树

Status PostOrderTrabverse(BiTree T,Status (\* Visit)(ElemType e)); //后序遍历二叉树

Status LevelOrderTraverse(BiTree T,Status (\* Visit)(ElemType e)); //层序遍历二叉树

void level(BiTree T, int h, Status (\* Visit)(ElemType e));

Status Print(ElemType e);

## 5.3数据结构与算法设计

为了满足概述中的功能，结合线性表结构，给出以下结构的定义：

/\*status是函数的类型，其值是函数结果状态代码\*/

typedef int Status;

//数据元素类型定义

typedef char ElemType;

typedef struct BiTNode{

ElemType date;

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

}BiTNode, \*BiTree;

typedef struct QNode

//队列节点，包括一个数据域和一个指针域，数据域为二叉树的指针类型

{

BiTree data;

struct QNode \*next;

}QNode,\*QueuePtr;

typedef struct

{

QueuePtr front,rear; //队头、队尾指针

}LinkQueue;

**算法设计：**

1. CreateBiTree：传入根节点指针，使用递归算法，由用户自行输入每个根节点的信息
2. PreOrderTraverse：采用递归算法，先访问根节点，即对于任一结点，其可看做是根结点，因此可以直接访问，访问完之后，若其左孩子不为空，按相同规则访问它的左子树；当访问其左子树时，再访问它的右子树。

关键代码： if(T)//若二叉树不为空

{

if(Visit(T->date))

if(PreOrderTraverse(T->lchild, Visit))//先序遍历左子树

if(PreOrderTraverse(T->rchild, Visit)) //先序遍历右子树

return OK;

return 0;

}

else return OK;

3.InOrderTraverse：采用递归算法，先访问左子树，即对于任一结点，其可看做是根结点，因此可以直接访问，访问完之后，若其左孩子为空，访问根节点，再访问它的右子树。

关键代码：

if(T)

{

if(InOrderTrabverse(T->lchild,Visit))

if(Visit(T->date))

if(InOrderTrabverse(T->rchild,Visit))

return 1;

return 0;

}else return 1;

4.PostOrderTraverse：递归算法与前序中序相似，先访问左子树然后右子树最后根节点。

关键代码：

if(T)

{

if(PostOrderTrabverse(T->lchild,Visit))

if(PostOrderTrabverse(T->rchild,Visit))

if(Visit(T->date))

return 1;

return 0;

}else return 1;

5.LevelOrderTraverse：构建一个队列，按层将每一次的节点输入队列，最后先进先出。

关键代码：

if(T)

{

if(h==1)

Visit(T->date);

else{

level(T->lchild,h-1, Visit);

level(T->rchild,h-1, Visit);

}

}

6. BiTreeDepth：关键代码：

int dept = 0;

if(T)

{

int lchilddepth = BiTreeDepth(T->lchild);

int rchilddepth = BiTreeDepth(T->rchild);

dept = lchilddepth >= rchilddepth ? (lchilddepth + 1) : (rchilddepth + 1);

}

return dept;

从根节点出发，树的深度等于左右子树深度较大值加1，若左右子树均为空，则该树深度为1；若根节点为空，则深度为0

## 5.4程序清单与测试

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <conio.h>

/\*结果状态代码\*/

#define OVERFLOW -2

#define OK 1

#define TRUE 1

#define FALSE 0

#define ERROR 0

int depth, LR;

/\*status是函数的类型，其值是函数结果状态代码\*/

typedef int Status;

//数据元素类型定义

typedef char ElemType;

typedef struct BiTNode{

ElemType date;

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

}BiTNode, \*BiTree;

typedef struct QNode //队列节点，包括一个数据域和一个指针域，数据域为二叉树的指针类型

{

BiTree data;

struct QNode \*next;

}QNode,\*QueuePtr;

typedef struct

{

QueuePtr front,rear; //队头、队尾指针

}LinkQueue;

ElemType e, value, E;

BiTree e1, e2, e3, p;

Status InitQueue(LinkQueue \*Q); //构造一个空队列

Status EnQueue(LinkQueue \*Q,BiTree e); //销毁队列Q，Q不再存在

Status QueueEmpty(LinkQueue Q); //若栈Q为空队列，则返回TURE，否则返回FALSE

Status DeQueue(LinkQueue \*Q,BiTree \*e); //删除队列Q的首元素,并且用e返回其值并返回OK；否则返回ERROR

BiTree Point(BiTree T,ElemType s);

Status InitBiTree(BiTree \* T); //构造空二叉树T

Status DestroyBiTree(BiTree \* T); //二叉树T已存在,销毁二叉树T

void CreateBiTree(BiTree \* T); //先序构造二叉树T

Status ClearBiTree(BiTree \* T); //清空二叉树

Status BiTreeEmpty(BiTree T); //初始条件：二叉树T存在，若T为空二叉树，则返回TRUE,否则返回FALSE.

Status BiTreeDepth(BiTree T); //求二叉树深度

BiTNode Root(BiTree T); //返回二叉树的根

Status Value(BiTree T, BiTree e); //返回二叉树结点的值

Status Assign(BiTree T, BiTree e, ElemType value); //给二叉树结点赋值

ElemType Parent(BiTree T,ElemType e); //求双亲结点

ElemType LeftChild(BiTree T,ElemType e); //求左孩子结点

ElemType RightChild(BiTree T,ElemType e); //求右孩子结点

ElemType LeftSibling(BiTree T,ElemType e); //求左兄弟结点

ElemType RightSibling(BiTree T,ElemType e); //求右兄弟结点

Status InsertChild(BiTree T,BiTree p,int LR,BiTree c);//插入子树

Status DeleteChild(BiTree T,BiTree p,int LR); //删除子树

Status PreOrderTraverse(BiTree T, Status (\* Visit)(ElemType e)); //先序遍历二叉树

Status InOrderTrabverse(BiTree T,Status (\* Visit)(ElemType e)); //中序遍历二叉树

Status PostOrderTrabverse(BiTree T,Status (\* Visit)(ElemType e)); //后序遍历二叉树

Status LevelOrderTraverse(BiTree T,Status (\* Visit)(ElemType e)); //层序遍历二叉树

void level(BiTree T, int h, Status (\* Visit)(ElemType e));

Status Print(ElemType e);

Status InitBiTree(BiTree \*T)

{

\*T = NULL;//根节点初始化为空

return 0;

}

Status DestroyBiTree(BiTree \*T)

{

free(T);//释放根节点

return 0;

}

void CreateBiTree(BiTree \*T)

{

//按先序次序输入二叉树中结点的值(一个字符)，空格字符表示空树

char ch;

ch = getchar();

if(ch == '#')

{

\*T = NULL;

}

else

{

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

(\*T)->date = ch; //生成根节点

CreateBiTree(&((\*T)->lchild));//构造左子树

CreateBiTree(&((\*T)->rchild));//构造右子树

}

}

Status ClearBiTree(BiTree \*T)

{

if((\*T)==NULL)

return OK;

else

{

ClearBiTree(&(\*T)->rchild);

ClearBiTree(&(\*T)->lchild);

free(T);

return OK;

}

}

Status BiTreeEmpty(BiTree T)

{

if(T == NULL)

{

return 1;

}

else

{

return 0;

}

}

Status BiTreeDepth(BiTree T)

{

int dept = 0;

if(T)

{

int lchilddepth = BiTreeDepth(T->lchild);

int rchilddepth = BiTreeDepth(T->rchild);

dept = lchilddepth >= rchilddepth ? (lchilddepth + 1) : (rchilddepth + 1);

}

return dept;

}

BiTNode Root(BiTree T)

{

return \*T;

}

Status Value(BiTree T, BiTree e)

{

return e->date;

}

Status Assign(BiTree T, BiTree e, ElemType value)

{

e->date = value;

return OK;

}

ElemType Parent(BiTree T,ElemType e)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点的元素 \*/

/\* 操作结果: 若e是T的非根结点,则返回它的双亲,否则返回＂空＂ \*/

LinkQueue q;

BiTree a;

if(T) /\* 非空树 \*/

{

InitQueue(&q); /\* 初始化队列 \*/

EnQueue(&q,T); /\* 树根入队 \*/

while(!QueueEmpty(q)) /\* 队不空 \*/

{

DeQueue(&q,&a); /\* 出队,队列元素赋给a \*/

if((a->lchild && a->lchild->date==e) || (a->rchild && a->rchild->date==e))

/\* 找到e(是其左或右孩子) \*/

return a->date; /\* 返回e的双亲的值 \*/

else /\* 没找到e,则入队其左右孩子指针(如果非空) \*/

{

if(a->lchild)

EnQueue(&q,a->lchild);

if(a->rchild)

EnQueue(&q,a->rchild);

}

}

}

return ERROR; /\* 树空或没找到e \*/

}

ElemType LeftChild(BiTree T,ElemType e)

{

BiTree a;

if(T)

{

a=Point(T,e);

if(a&&a->lchild)

return a->lchild->date;

}

return ERROR;

}

ElemType RightChild(BiTree T,ElemType e)

{

BiTree a;

if(T)

{

a=Point(T,e);

if(a&&a->rchild)

return a->rchild->date;

}

return ERROR;

}

ElemType LeftSibling(BiTree T,ElemType e)

{

ElemType a;

BiTree p;

if(T)

{

a=Parent(T,e);

p=Point(T,a);

if(p->lchild&&p->rchild&&p->rchild->date==e)

return p->lchild->date;

}

return ERROR;

}

ElemType RightSibling(BiTree T,ElemType e)

{

ElemType a;

BiTree p;

if(T)

{

a=Parent(T,e);

p=Point(T,a);

if(p->lchild&&p->rchild&&p->lchild->date==e)

return p->rchild->date;

}

return ERROR;

}

Status InsertChild(BiTree T,BiTree p,int LR,BiTree c)

{

if(!c&&!T&&(LR==1||LR==0)) exit(OVERFLOW);

if(LR==0)

{

p->rchild = c->rchild;

p->lchild = NULL;

c->rchild = p;

}

else

{

p->lchild = c->lchild;

p->rchild = NULL;

c->lchild = p;

}

return OK;

}

Status DeleteChild(BiTree T,BiTree p,int LR)

{

if(T)

{

if(LR)

{

p->rchild=NULL;

}

else

{

p->lchild=NULL;

}

return OK;

}

return ERROR;

}

Status PreOrderTraverse(BiTree T, Status (\* Visit)(ElemType e))

{//先序遍历

if(T)//若二叉树不为空

{

if(Visit(T->date))

if(PreOrderTraverse(T->lchild, Visit))//先序遍历左子树

if(PreOrderTraverse(T->rchild, Visit)) //先序遍历右子树

return OK;

return 0;

}

else return OK;

}

Status InOrderTrabverse(BiTree T,Status (\* Visit)(ElemType e))

{

if(T)

{

if(InOrderTrabverse(T->lchild,Visit))

if(Visit(T->date))

if(InOrderTrabverse(T->rchild,Visit))

return 1;

return 0;

}else return 1;

}

Status PostOrderTrabverse(BiTree T,Status (\* Visit)(ElemType e))

{

if(T)

{

if(PostOrderTrabverse(T->lchild,Visit))

if(PostOrderTrabverse(T->rchild,Visit))

if(Visit(T->date))

return 1;

return 0;

}else return 1;

}

void level(BiTree T, int h, Status (\* Visit)(ElemType e))

{//

if(T)

{

if(h==1)

Visit(T->date);

else{

level(T->lchild,h-1, Visit);

level(T->rchild,h-1, Visit);

}

}

}

Status LevelOrderTraverse(BiTree T,Status (\* Visit)(ElemType e))

{

if(T)

{

int h=BiTreeDepth(T);

int i;

for(i=1;i<=h;i++)

{

level(T,i, Visit);

}return OK;

}else return OK;

}

Status Print(ElemType e)

{

printf("%c", e);

return OK;

}

Status InitQueue(LinkQueue \*Q)

{ /\* 构造一个空队列Q \*/

(\*Q).front=(\*Q).rear=(QueuePtr)malloc(sizeof(QNode));

if(!(\*Q).front)

exit(OVERFLOW);

(\*Q).front->next=NULL;

return OK;

}

Status EnQueue(LinkQueue \*Q,BiTree e)

{ /\* 插入元素e为Q的新的队尾元素 \*/

QueuePtr p=(QueuePtr)malloc(sizeof(QNode));

if(!p) /\* 存储分配失败 \*/

exit(OVERFLOW);

p->data=e;

p->next=NULL;

(\*Q).rear->next=p;

(\*Q).rear=p;

return OK;

}

Status QueueEmpty(LinkQueue Q)

{ /\* 若Q为空队列,则返回TRUE,否则返回FALSE \*/

if(Q.front==Q.rear)

return TRUE;

else

return FALSE;

}

Status DeQueue(LinkQueue \*Q,BiTree \*e)

{ /\* 若队列不空,删除Q的队头元素,用e返回其值,并返回OK,否则返回ERROR \*/

QueuePtr p;

if((\*Q).front==(\*Q).rear)

return ERROR;

p=(\*Q).front->next;

\*e=p->data;

(\*Q).front->next=p->next;

if((\*Q).rear==p)

(\*Q).rear=(\*Q).front;

free(p);

return OK;

}

BiTree Point(BiTree T,ElemType s)

{ /\* 返回二叉树T中指向元素值为s的结点的指针。\*/

LinkQueue q;

BiTree a;

if(T) /\* 非空树 \*/

{

InitQueue(&q); /\* 初始化队列 \*/

EnQueue(&q,T); /\* 根结点入队 \*/

while(!QueueEmpty(q)) /\* 队不空 \*/

{

DeQueue(&q,&a); /\* 出队,队列元素赋给a \*/

if(a->date==s)

return a;

if(a->lchild) /\* 有左孩子 \*/

EnQueue(&q,a->lchild); /\* 入队左孩子 \*/

if(a->rchild) /\* 有右孩子 \*/

EnQueue(&q,a->rchild); /\* 入队右孩子 \*/

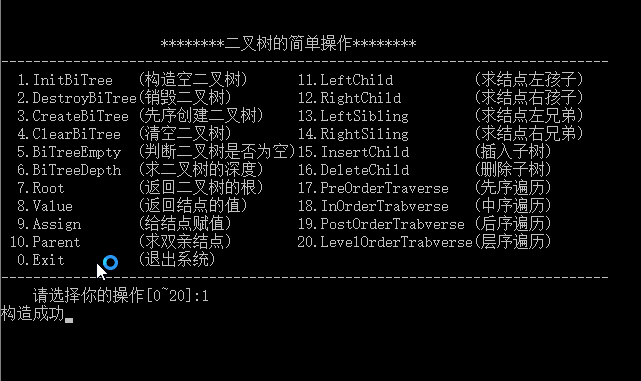
}

}

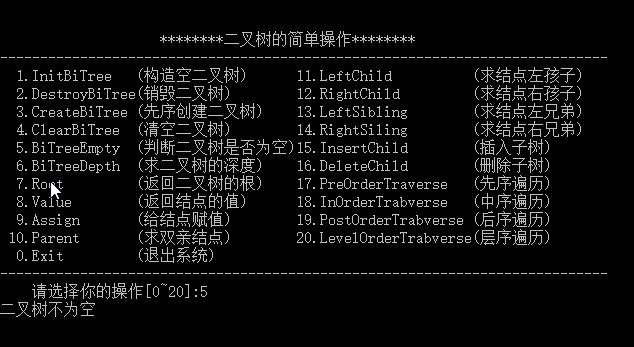
return NULL;

}

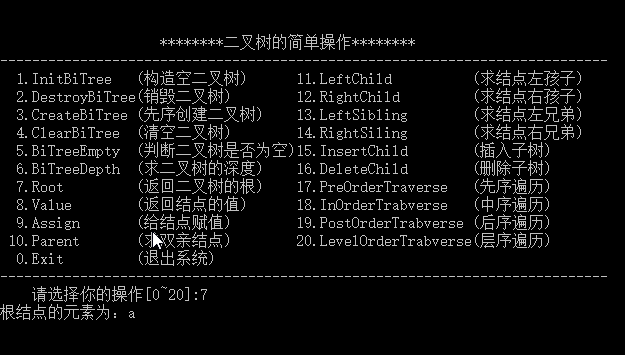
**测试：**创建二叉树、输入abc##d##ef###、判断是否为空、求深度、返回根、给结点d赋值p、求e的双亲结点、求a左孩子、求a右孩子、求p左兄弟、求p右兄弟、先序遍历、中序遍历、后序遍历、层序遍历、清空二叉树、销毁二叉树、退出



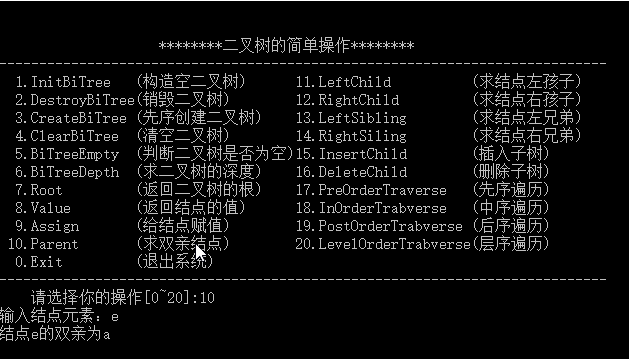


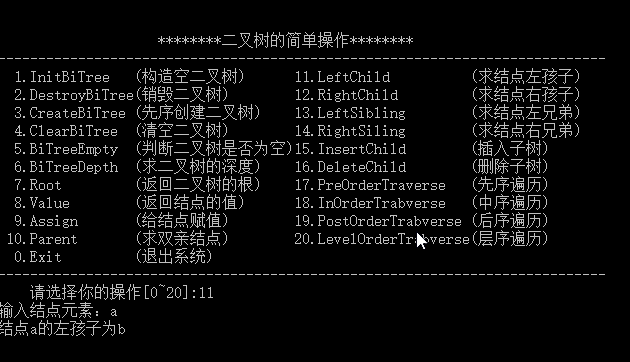


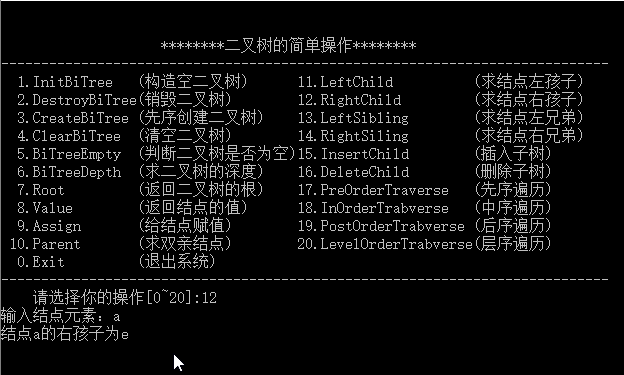


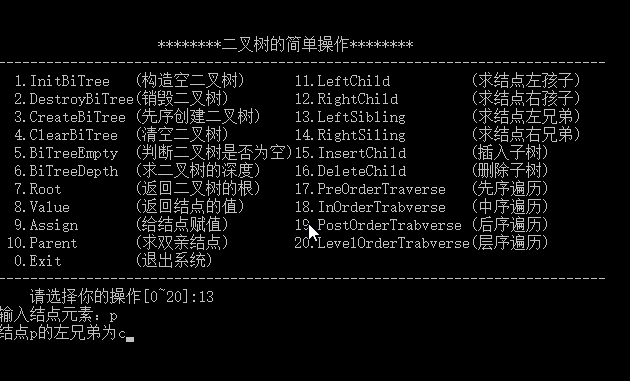


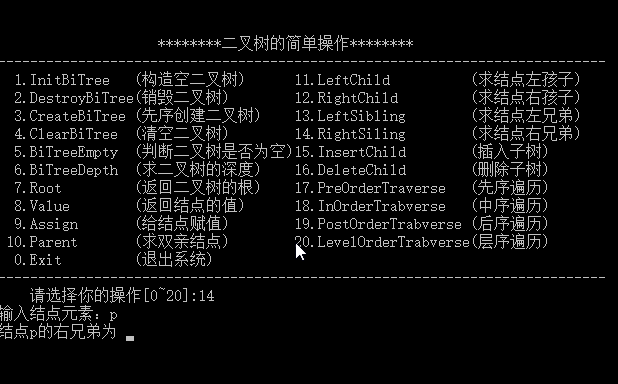


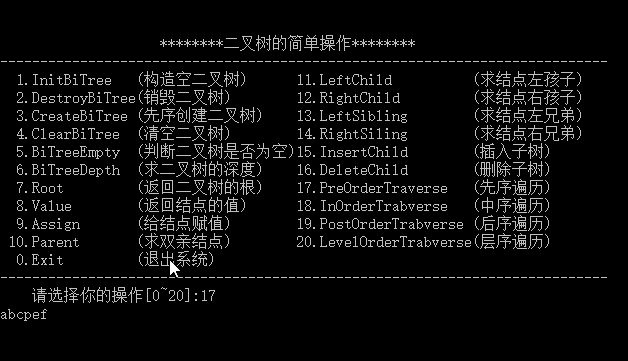


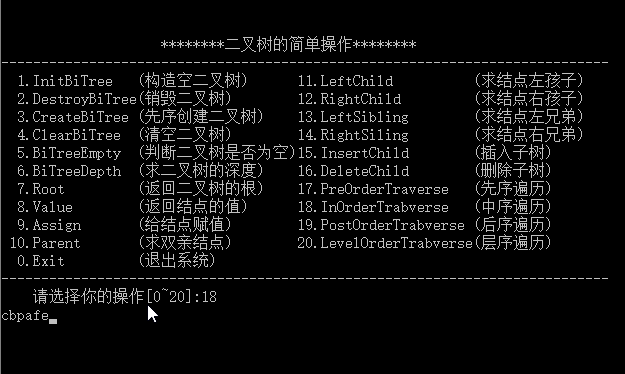


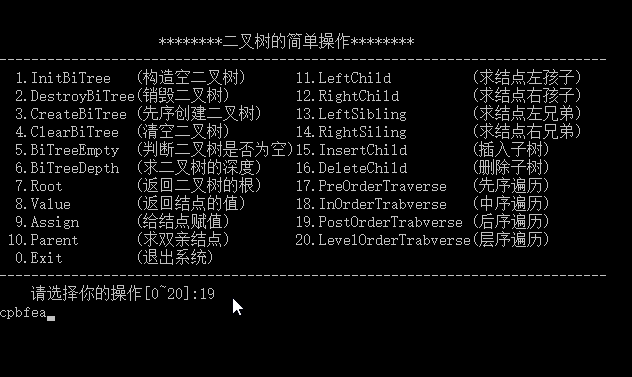


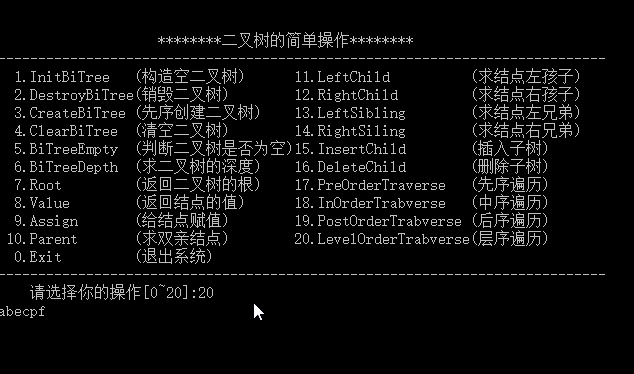














## 5.5实验总结与评价

这次实验感觉较之前几次的都要难一些，所以花的时间也比较多，中间写先序、中序、后序遍历算法比较纠结，问过同学后发现用递归算法很简单，之后很快就搞定了。这次实验学会了如何创建二叉树以及对二叉树的各类简单操作，对递归的思想更加熟悉，对指针的运用更加熟练，收获很大，来日方长，在编程的路上还会遇到很多困难，都要努力的寻找办法去解决，这是成为一个好的程序员的必经之路！

**6基于邻接表实现图的基本运算**

## 6.1实验内容与要求

基于邻接表，通过实验加深对图的概念、基本运算的理解；熟练掌握图的逻辑结构与物理结构的关系；以邻接表作为物理结构，熟练掌握图基本运算的实现。

## 6.2程序概要设计

1.明确基本功能

程序需要实现的11个基本功能分别是：CreateCraph (构造空图)，DestroyCraph (销毁图)，LocateVex (返回顶点位置信息)，FirstAdjVex（返回v的第一个邻接顶点），NextAdjVex（返回v的下一个邻接顶点），InsertVex（增加新顶点v），DeleteVex（删除顶点v），InsertArc（增加弧），DeleteArc（删除弧），DFSTraverse（深度优先搜索遍历），BFSTraverse（广度优先搜索遍历）。

2.确定各功能实现的函数参数

Status CreateGraph(ALGraph \* G);

//按V和VR的定义构造图G

Status DestroyGraph(ALGraph \* G);

//销毁图G

int LocateVex(ALGraph G, VertexType u);

//若u在图G中存在，返回顶点u的位置序号，否则返回-1

int FirstAdjVex(ALGraph G, VertexType v);

//返回v的第一个邻接顶点，如果v没有邻接顶点，返回空

int NextAdjVex(ALGraph G, VertexType v, VertexType w);

//返回v的（相对于w）下一个邻接顶点，如果w是最后一个邻接顶点，返回空。

Status InsertVex(ALGraph \* G, VertexType v);

//在图G中增加新顶点v

Status DeleteVex(ALGraph \* G, VertexType v);

//在图G中删除顶点v和与v相关的弧

Status InsertArc(ALGraph \* G, VertexType v, VertexType w);

//在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>

Status DeleteArc(ALGraph \* G, VertexType v, VertexType w);

//在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>

Status DFSTraverse(ALGraph G, Status (\*visit)(VertexType a));

//对图G进行深度优先搜索遍历

Status BFSTraverse(ALGraph G, Status (\*visit)(VertexType a));

//对图G进行广度优先搜索遍历

int OutDegree(AdjList \* i);

//计算图G第i个定点的出度

Status SaveGraph(ALGraph G);

//保存

Status LoadGraph(ALGraph \* G);

//读取

Status Visit(VertexType a);

//输出a

Boolean visited[MAX\_VERTEX\_NUM];

//访问标志数组

Status (\*VisitFunc)(VertexType a);

//函数变量

## 6.3数据结构与算法设计

为了满足概述中的功能，结合图邻接表结构，给出以下结构的定义：

//Status是函数的类型，其值是函数结果状态代码

typedef int Status;

队列相关：

typedef int QElemType;

typedef struct QNode{

QElemType data;

struct QNode \*next;

}QNode,\*QueuePtr;

typedef struct{

QueuePtr front,rear; //队头、队尾指针

}LinkQueue;

图相关：

//数据元素类型定义

typedef char VertexType[MAX\_VERNAME\_LEN];

typedef int InfoType;

typedef struct ArcNode{

int adjvex; //该弧所指向的顶点位置

struct ArcNode \* nextarc; //指向下一条弧的指针

InfoType \*info; //该弧相关信息的指针

}ArcNode;

typedef struct VNode{

ArcNode \* firstarc; //指向第一条依附该顶点的弧的指针

VertexType data; //顶点信息

}VNode, AdjList[MAX\_VERTEX\_NUM];

typedef struct{

AdjList vertices;

int vexnum, arcnum; //图的当前顶点数和弧数

int kind; //图的种类标志

}ALGraph;

**算法设计：**

1. CreateCraph(&G,V,VR)

函数名称：CreateGraph

输入参数： 图G地址

返回值： Status 成功则返回OK

函数功能： 构造图G。

（2）DestroyCraph(&G)

函数名称：DestroyGraph

输入参数： 图G地址

返回值： Status 成功则返回OK

函数功能： 销毁图G。

（3）LocateVex(G,u)

函数名称：LocateVex

输入参数： 图G,所要查找的顶点的名称或值u

返回值： int

函数功能： 若u在图G中存在，返回顶点u的位置序号，否则返回-1。

（4）FirstAdjVex(&G, v)

函数名称：FirstAdjVex

输入参数： 图G, G中某个顶点v

返回值： int

函数功能： 返回v的第一个邻接顶点的序号。若顶点在G中没有邻接顶点,则返回-1。

（5）NextAdjVex(&G, v, w)

函数名称：NextAdjVex

输入参数： 图G, G中某个顶点v, v的邻接顶点w

返回值： int

函数功能： 返回v的(相对于w的)下一个邻接顶点的序号，若w是v的最后一个邻接点,则返回-1。

（6）InsertVex(&G,v)

函数名称：InsertVex

输入参数： 图G地址, 新顶点v

返回值： Status

函数功能： 在图G中增加新顶点v。

（7）DeleteVex(&G,v)

函数名称：DeleteVex

输入参数： 图G地址, v是G中某个顶点

返回值： Status 成功返回OK

函数功能： 删除G中顶点v及其相关的弧。

（8）InsertArc(&G,v,w)

函数名称：InsertArc

输入参数： 图G地址, G中两个顶点v、w

返回值： Status 成功返回OK；

函数功能：在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>。

（9）DeleteArc(&G,v,w)

函数名称： DeleteArc

输入参数： 图G地址, G中两个顶点v、w

返回值： Status 成功返回OK；

函数功能： 在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>。

（10）DFSTraverse(G,visit())

函数名称： DFSTraverse

输入参数： 图G, 调用函数visit

返回值： Status 成功返回OK；

函数功能： 对图G进行深度优先搜索遍历，依次对图中的每一个顶点使用函数visit访问一次，且仅访问一次。

（11）BFSTraverse(G,visit())

函数名称： BFSTraverse

输入参数： 图G, 调用函数visit

返回值： Status 成功返回OK；

函数功能： 按广度优先非递归遍历图G。

## 6.4程序清单与测试

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

//函数结果状态代码

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -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;

typedef int QElemType;

typedef struct QNode{

QElemType data;

struct QNode \*next;

}QNode,\*QueuePtr;

typedef struct{

QueuePtr front,rear; //队头、队尾指针

}LinkQueue;

Status InitQueue(LinkQueue \*Q)

{

//构造一个空队列Q

(\*Q).front = (\*Q).rear = (QueuePtr)malloc(sizeof(QNode));

if(!(\*Q).front) exit(OVERFLOW);

(\*Q).front->next = NULL;

return OK;

}

Status QueueEmpty(LinkQueue Q)

{

//若Q为空队列,则返回TRUE,否则返回FALSE

if(Q.front == Q.rear) return TRUE;

else return FALSE;

}

Status EnQueue(LinkQueue \*Q, QElemType e)

{

//插入元素e为Q的新的队尾元素

QueuePtr p = (QueuePtr)malloc(sizeof(QNode));

if(!p) //存储分配失败

exit(OVERFLOW);

p->data = e;

p->next = NULL;

(\*Q).rear->next = p;

(\*Q).rear = p;

return OK;

}

Status DeQueue(LinkQueue \*Q, QElemType \*e)

{

//若队列不空,删除Q的队头元素,用e返回其值,并返回OK,否则返回ERROR

QueuePtr p;

if((\*Q).front == (\*Q).rear)

return ERROR;

p = (\*Q).front->next;

\*e = p->data;

(\*Q).front->next = p->next;

if((\*Q).rear == p)

(\*Q).rear = (\*Q).front;

free(p);

return OK;

}

typedef int Boolean;

#define MAX\_VERTEX\_NUM 20 //最大顶点数

#define MAX\_VERNAME\_LEN 10 //顶点名称最大长度

//数据元素类型定义

typedef char VertexType[MAX\_VERNAME\_LEN];

typedef int InfoType;

typedef struct ArcNode{

int adjvex; //该弧所指向的顶点位置

struct ArcNode \* nextarc; //指向下一条弧的指针

InfoType \*info; //该弧相关信息的指针

}ArcNode;

typedef struct VNode{

ArcNode \* firstarc; //指向第一条依附该顶点的弧的指针

VertexType data; //顶点信息

}VNode, AdjList[MAX\_VERTEX\_NUM];

typedef struct{

AdjList vertices;

int vexnum, arcnum; //图的当前顶点数和弧数

int kind; //图的种类标志

}ALGraph;

Status CreateGraph(ALGraph \* G);

//按V和VR的定义构造图G

Status DestroyGraph(ALGraph \* G);

//销毁图G

int LocateVex(ALGraph G, VertexType u);

//若u在图G中存在，返回顶点u的位置序号，否则返回-1

int FirstAdjVex(ALGraph G, VertexType v);

//返回v的第一个邻接顶点，如果v没有邻接顶点，返回空

int NextAdjVex(ALGraph G, VertexType v, VertexType w);

//返回v的（相对于w）下一个邻接顶点，如果w是最后一个邻接顶点，返回空。

Status InsertVex(ALGraph \* G, VertexType v);

//在图G中增加新顶点v

Status DeleteVex(ALGraph \* G, VertexType v);

//在图G中删除顶点v和与v相关的弧

Status InsertArc(ALGraph \* G, VertexType v, VertexType w);

//在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>

Status DeleteArc(ALGraph \* G, VertexType v, VertexType w);

//在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>

Status DFSTraverse(ALGraph G, Status (\*visit)(VertexType a));

//对图G进行深度优先搜索遍历

Status BFSTraverse(ALGraph G, Status (\*visit)(VertexType a));

//对图G进行广度优先搜索遍历

int OutDegree(AdjList \* i);

//计算图G第i个定点的出度

Status SaveGraph(ALGraph G);

//保存

Status LoadGraph(ALGraph \* G);

//读取

Status Visit(VertexType a);

//输出a

Boolean visited[MAX\_VERTEX\_NUM];

//访问标志数组

Status (\*VisitFunc)(VertexType a);

//函数变量

/\*函数名称：CreateGraph

\*输入参数： 图G地址

\*返回值： Status 成功则返回OK

\*函数功能： 构造图G

\*/

Status CreateGraph(ALGraph \* G)

{

int i,j,k;

int w; //权值W

VertexType va,vb;//弧尾Va，弧头Vb

ArcNode \*p;

printf("请输入图的3项基本信息：\n类型(0-有向图, 1-有向网, 2-无向图, 3-无向网): ");

//输图类型

scanf("%d",&(\*G).kind);

while ((\*G).kind < 0 || (\*G).kind > 3)

{

printf("输入值非法！请重输: ");

scanf("%d",&(\*G).kind);

}

//输顶点数

printf("顶点数（%d以内）: ", MAX\_VERTEX\_NUM);

scanf("%d", &(\*G).vexnum);

while ((\*G).vexnum < 0 || (\*G).vexnum > MAX\_VERTEX\_NUM)

{

printf("输入值非法！请重输: ");

scanf("%d",&(\*G).vexnum);

}

//输边数

printf("弧(边)数: ");

scanf("%d", &(\*G).arcnum);

printf("请输入%d个顶点的名称或值(%d个字符以内,空格分开):\n", (\*G).vexnum, MAX\_VERNAME\_LEN);

for (i = 0; i < (\*G).vexnum; i++) //输入各顶点信息

{

getchar();

scanf("%s",(\*G).vertices[i].data);

(\*G).vertices[i].firstarc = NULL;

}

if ((\*G).kind % 2) //1、3为网

{

printf("请按序输入弧(边)权值W、弧尾Va和弧头Vb(空格分开):\n W Va Vb\n");

}

else //0、2为图

{

printf("请按序输入弧(边)的弧尾Va和弧头Vb(空格分开):\n Va Vb\n");

}

for (k = 0; k < (\*G).arcnum; k++) //构造邻接表

{

printf("%d: ", k+1);

if ((\*G).kind % 2) //网

{

scanf("%d%s%s", &w, va, vb);

}

else //图

{

scanf("%s%s", va, vb);

}

i = LocateVex(\*G, va); //弧尾

j = LocateVex(\*G, vb); //弧头

if (i == -1 || j == -1)

{//顶点不存在

printf("顶点不存在！请重输！\n");

k--;

continue;

}

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

p->adjvex = j;

if ((\*G).kind % 2)//网

{

p->info = (int \*)malloc(sizeof(int));

\*(p->info) = w;

}

else //图

{

p->info = NULL;

}

p->nextarc = (\*G).vertices[i].firstarc; //插在表头

(\*G).vertices[i].firstarc = p;

//无向图或无向网还需逆向的

if((\*G).kind >= 2)

{

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

p->adjvex = i;

if ((\*G).kind == 3) //无向网

{

p->info = (int\*)malloc(sizeof(int));

\*(p->info) = w;

}

else //无向图

{

p->info = NULL;

}

p->nextarc = (\*G).vertices[j].firstarc; //插在表头

(\*G).vertices[j].firstarc = p;

}

}

return OK;

}

/\*函数名称：DestroyGraph

\*输入参数： 图G地址

\*返回值： Status 成功则返回OK

\*函数功能： 销毁图G

\*/

Status DestroyGraph(ALGraph \*G)

{

int i;

ArcNode \* p, \* q;

for(i = 0; i < (\*G).vexnum; i++)

{

p=(\*G).vertices[i].firstarc;

while(p)

{

q = p->nextarc;

if((\*G).kind % 2) //网

{

free(p->info);

}

free(p);

p = q;

}

}

(\*G).vexnum = 0;

(\*G).arcnum = 0;

(\*G).kind = -1;//以-1代表图不存在

return OK;

}

/\*函数名称：LocateVex

\*输入参数： 图G,所要查找的顶点的名称或值u

\*返回值： int

\*函数功能： 若u在图G中存在，返回顶点u的位置序号，否则返回-1

\*/

int LocateVex(ALGraph G, VertexType u)

{

int i;

for(i = 0; i < G.vexnum; i++)

{

if(strcmp(u,G.vertices[i].data) == 0)

{

return i;

}

}

return -1;//未找到则返回-1

}

/\*函数名称：FirstAdjVex

\*输入参数： 图G, G中某个顶点v

\*返回值： int

\*函数功能： 返回v的第一个邻接顶点的序号。若顶点在G中没有邻接顶点,则返回-1

\*/

int FirstAdjVex(ALGraph G, VertexType v)

{

ArcNode \*p;

int v1;

v1 = LocateVex(G,v);//获取v在图G中的序号v1

if (v1 == -1)

{

return -2;

}

p = G.vertices[v1].firstarc;

if (p)

{

return p->adjvex;

}

else

{

return -1;

}

}

/\*函数名称：NextAdjVex

\*输入参数： 图G, G中某个顶点v, v的邻接顶点w

\*返回值： int

\*函数功能： 返回v的(相对于w的)下一个邻接顶点的序号，

\* 若w是v的最后一个邻接点,则返回-1

\*/

int NextAdjVex(ALGraph G,VertexType v,VertexType w)

{

ArcNode \*p;

int v1, w1;

v1 = LocateVex(G,v); //获取v在图G中的序号v1

w1 = LocateVex(G,w); //获取w在图G中的序号w1

if (v1 == -1)

{//顶点不存在

return(-2);

}

if (w1 == -1)

{//邻接点不存在

return(-3);

}

p = G.vertices[v1].firstarc;

while (p && p->adjvex != w1)

{//指针p不空且所指表结点不是w

p = p->nextarc;

}

//没找到w或w是最后一个邻接点

if(!p || !p->nextarc)

{

return -1;

}

else

{//返回v的(相对于w的)下一个邻接顶点的序号

return p->nextarc->adjvex;

}

}

/\*函数名称：InsertVex

\*输入参数： 图G地址, 新顶点v

\*返回值： Status

\*函数功能： 在图G中增加新顶点v

\*/

Status InsertVex(ALGraph \*G, VertexType v)

{

if((\*G).vexnum == MAX\_VERTEX\_NUM)//顶点数达最大值

{

return ERROR;

}

//构造新顶点

strcpy((\*G).vertices[(\*G).vexnum].data, v);

(\*G).vertices[(\*G).vexnum].firstarc=NULL;

(\*G).vexnum++; //图G的顶点数加1

return OK;

}

/\*函数名称：DeleteVex

\*输入参数： 图G地址, v是G中某个顶点

\*返回值： Status 成功返回OK

\*函数功能： 删除G中顶点v及其相关的弧

\*/

Status DeleteVex(ALGraph \*G, VertexType v)

{

int i, j;

ArcNode \*p, \*q;

j = LocateVex(\*G, v); //获取顶点v的序号

if(j == -1) //v不是图G的顶点

{

return ERROR;

}

//删除以v为出度的弧或边

p = (\*G).vertices[j].firstarc;

while(p)

{

q = p;

p = p->nextarc;

if((\*G).kind%2) //网

{

free(q->info);

}

free(q);

(\*G).arcnum--; //弧或边数减1

}

//重新排列顶点

(\*G).vexnum--; //顶点数减1

for(i = j; i < (\*G).vexnum; i++) //顶点v后面的顶点前移

{

(\*G).vertices[i] = (\*G).vertices[i+1];

}

//删除以v为入度的弧或边且修改弧所指顶点位置值

for(i = 0; i < (\*G).vexnum; i++)

{

p = (\*G).vertices[i].firstarc; //指向第1条弧或边

while(p) //有弧或边

{

if(p->adjvex == j)//该弧指向被删结点

{

if(p == (\*G).vertices[i].firstarc) //待删结点是第1个结点

{

(\*G).vertices[i].firstarc = p->nextarc;

if((\*G).kind%2) //网

{

free(p->info);

}

free(p);

p = (\*G).vertices[i].firstarc;

if((\*G).kind < 2)

{//有向图需再减一次

(\*G).arcnum--; //弧或边数减1

}

}

else

{

q->nextarc = p->nextarc;

if((\*G).kind % 2) //网

{

free(p->info);

}

free(p);

p = q->nextarc;

if((\*G).kind < 2)

{//有向需再减一次

(\*G).arcnum--; //弧或边数减1

}

}

}

else//该弧不指向被删结点

{

if(p->adjvex > j)//该弧所指元素位置序号在删除顶点之后

{//更改弧所指顶点位置序号

p->adjvex--;

}

q = p;

p = p->nextarc;//分析下一条弧

}

}

}

return OK;

}

/\*函数名称：InsertArc

\*输入参数： 图G地址, G中两个顶点v、w

\*返回值： Status 成功返回OK；

\*函数功能：在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>

\*/

Status InsertArc(ALGraph \*G,VertexType v,VertexType w)

{

ArcNode \*p;

int w1, i, j;

i = LocateVex(\*G, v); //获取v位置序号

j = LocateVex(\*G, w); //获取w位置序号

if(i == -1 || j == -1)

{//未找到相应顶点

return ERROR;

}

//建立弧v-w

(\*G).arcnum++;

if((\*G).kind % 2) //网要输权值

{

printf("请输入弧(边)%s->%s的权值: ", v, w);

scanf("%d", &w1);

}

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

p->adjvex = j;

if((\*G).kind % 2) //网

{

p->info = (int \*)malloc(sizeof(int));

\*(p->info) = w1;

}

else

{

p->info=NULL;

}

p->nextarc = (\*G).vertices[i].firstarc;

(\*G).vertices[i].firstarc = p;

//无向图需要再建立弧w-v

if((\*G).kind >= 2)

{

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

p->adjvex = i;

if((\*G).kind == 3) //无向网

{

p->info = (int\*)malloc(sizeof(int));

\*(p->info) = w1;

}

else

{

p->info = NULL;

}

p->nextarc = (\*G).vertices[j].firstarc;

(\*G).vertices[j].firstarc = p;

}

return OK;

}

/\*函数名称： DeleteArc

\*输入参数： 图G地址, G中两个顶点v、w

\*返回值： Status 成功返回OK；

\*函数功能： 在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>

\*/

Status DeleteArc(ALGraph \* G,VertexType v,VertexType w)

{

ArcNode \*p, \*q;

int i, j;

i = LocateVex(\*G, v); //顶点v序号

j = LocateVex(\*G, w); //顶点w序号

if(i < 0 || j < 0)

{

return ERROR;

}

//删除弧<v,w>

p = (\*G).vertices[i].firstarc;

while(p && p->adjvex != j) //p不空且所指之弧不是待删除弧<v,w>

{ //p指向下一条弧

q = p;

p = p->nextarc;

}

if(p && p->adjvex == j) //找到弧<v,w>

{

if(p == (\*G).vertices[i].firstarc) //p所指是第1条弧

{//指向下一条弧

(\*G).vertices[i].firstarc = p->nextarc;

}

else

{//指向下一条弧

q->nextarc = p->nextarc;

}

if((\*G).kind % 2)

{//网要多释放权值

free(p->info);

}

free(p); //释放此结点

(\*G).arcnum--; //弧或边数减1

}

//无向图要删除对称弧<w,v>

if((\*G).kind >= 2)

{

p = (\*G).vertices[j].firstarc;

while(p && p->adjvex != i) //p不空且所指之弧不是待删除弧<w,v>

{ //p指向下一条弧 \*/

q = p;

p = p->nextarc;

}

if(p && p->adjvex == i) //找到弧<w,v>

{

if(p == (\*G).vertices[j].firstarc) //p所指是第1条弧

{//指向下一条弧

(\*G).vertices[j].firstarc=p->nextarc;

}

else

{//指向下一条弧

q->nextarc=p->nextarc;

}

if((\*G).kind % 2)

{//网要多释放权值

free(p->info);

}

free(p); //释放此结点

}

}

return OK;

}

/\*函数名称： DFS

\*输入参数： 图G, 顶点序号v

\*返回值： 无

\*函数功能： 从第v个顶点出发递归地深度优先遍历图G。

\*/

void DFS(ALGraph G,int v)

{

int w;

VertexType v1, w1;

strcpy(v1, G.vertices[v].data);

visited[v] = TRUE; //设置访问标志为TRUE(已访问)

VisitFunc(G.vertices[v].data); /\* 访问第v个顶点 \*/

for(w = FirstAdjVex(G,v1); w >= 0; w = NextAdjVex(G, v1, strcpy(w1, G.vertices[w].data)))

{

if(!visited[w])

{

DFS(G,w); //对v的尚未访问的邻接点w递归调用DFS

}

}

}

/\*函数名称： DFSTraverse

\*输入参数： 图G, 调用函数visit

\*返回值： Status 成功返回OK；

\*函数功能： 对图G进行深度优先搜索遍历，依次对图中的每一个顶点使用函

\* 数visit访问一次，且仅访问一次

\*/

Status DFSTraverse(ALGraph G, Status (\*Visit)(VertexType a))

{

int v;

VisitFunc = Visit; //使用全局变量VisitFunc,使DFS不必设函数指针参数

for(v = 0; v < G.vexnum; v++)

{

visited[v]=FALSE; //访问标志数组初始化

}

for(v = 0; v < G.vexnum; v++)

{

if(!visited[v])

{

DFS(G,v); //对尚未访问的顶点调用DFS

}

}

return OK;

}

/\*函数名称： BFSTraverse

\*输入参数： 图G, 调用函数visit

\*返回值： Status 成功返回OK；

\*函数功能： 按广度优先非递归遍历图G。

\*/

Status BFSTraverse(ALGraph G, Status (\*Visit)(VertexType a))

{

int v, u, w;

VertexType u1, w1;

LinkQueue Q;

for(v = 0; v < G.vexnum; ++v) visited[v]=FALSE;

InitQueue(&Q); //置空的辅助队列Q

for(v = 0; v < G.vexnum; ++v)

if(!visited[v]) //v尚未访问

{

visited[v] = TRUE;

Visit(G.vertices[v].data);

EnQueue(&Q, v); //v入队列

while(!QueueEmpty(Q))

{

DeQueue(&Q, &u); //队头元素出队并置为u

strcpy(u1, G.vertices[u].data);

for(w = FirstAdjVex(G,u1); w >= 0; w = NextAdjVex(G, u1, strcpy(w1, G.vertices[w].data)))

if(!visited[w]) //w为u的尚未访问的邻接顶点

{

visited[w] = TRUE;

Visit(G.vertices[w].data);

EnQueue(&Q,w); //w入队

}

}

}

return OK;

}

/\*函数名称：OutDegree

\*输入参数： 图G的第i个顶点地址

\*返回值： int 图G的第i个顶点出度

\*函数功能： 计算图G的第i个顶点出度

\*/

int OutDegree(AdjList \* i)

{

int num = 0;

ArcNode \* p;

p = (\*i)->firstarc;

while (p != NULL)

{

num++;

p = p->nextarc;

}

return num;

}

/\*函数名称：LoadGraph

\*输入参数： 图G地址

\*返回值： Status

\*函数功能： 读取线性表。

\*/

Status LoadGraph(ALGraph \* G)

{

int i, k = 0, num;

FILE \*fp;

ArcNode \*p;

char filename[30];

printf("请输入读取文件名: ");

scanf("%s", &filename);

if ((fp = fopen(filename,"r")) == NULL)

{

printf("文件打开失败！\n ");

return ERROR;

}

//读取图基本信息

fread(&G->kind, sizeof(int), 1, fp);

fread(&G->vexnum, sizeof(int), 1, fp);

fread(&G->arcnum, sizeof(int), 1, fp);

//读取邻接表

while(fread(G->vertices[k].data, sizeof(char), MAX\_VERTEX\_NUM, fp))//有顶点

{

fread(&num, sizeof(int), 1, fp);

if (num == 0)

{

G->vertices[k].firstarc = NULL;

}

else

{

G->vertices[k].firstarc = NULL;

for (i = 0; i < num; i++)

{

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

fread(&(p->adjvex), sizeof(int), 1, fp);

if ((\*G).kind % 2)//网

{

p->info = (int \*)malloc(sizeof(int));

fread(p->info, sizeof(InfoType), 1, fp);

}

else //图

{

p->info = NULL;

}

p->nextarc = (\*G).vertices[k].firstarc; //插在表头

(\*G).vertices[k].firstarc = p;

}

}

k++;

}

fclose(fp);

return OK;

}

/\*函数名称：SaveGraph

\*输入参数： 线性表G

\*返回值： Status

\*函数功能： 保存线性表。

\*/

Status SaveGraph(ALGraph G)

{

int i, num;

FILE \*fp;

char filename[30];

ArcNode \*p;

printf("请输入保存文件名： ");

scanf("%s",&filename);

//写文件的方法

if ((fp = fopen(filename, "w")) == NULL)

{

printf("文件打开失败\n ");

return ERROR;

}

//保存图基本信息

fwrite(&G.kind, sizeof(int), 1, fp);

fwrite(&G.vexnum, sizeof(int), 1, fp);

fwrite(&G.arcnum, sizeof(int), 1, fp);

//保存邻接表

for (i = 0; i < G.vexnum; i++)

{

fwrite(G.vertices[i].data, sizeof(char), MAX\_VERTEX\_NUM, fp);

num = OutDegree(&(G.vertices[i]));

fwrite(&num, sizeof(int), 1, fp);

if ((p = G.vertices[i].firstarc) != NULL)

{

fwrite(&p->adjvex, sizeof(int), 1, fp);

if (G.kind % 2)//网

{

fwrite(p->info, sizeof(int), 1, fp);

}

while ((p = p->nextarc) != NULL)

{

fwrite(&p->adjvex, sizeof(int), 1, fp);

if (G.kind % 2)//网

{

fwrite(p->info, sizeof(int), 1, fp);

}

}

}

}

fclose(fp);

return OK;

}

/\*

\*函数名称： Visit

\*输入参数： ElemType a

\*返回值： Status 返回OK

\*函数功能： 输出a

\*/

Status Visit(VertexType a)

{

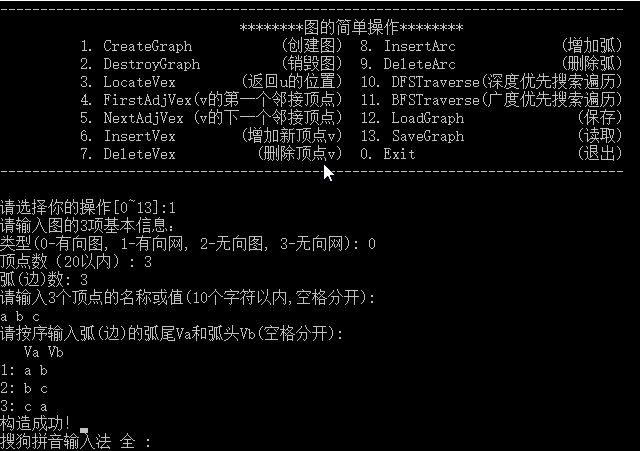
printf("%s ",a);

return OK;

}

**测试（有向图为例）：**

创建3个顶点abc的图

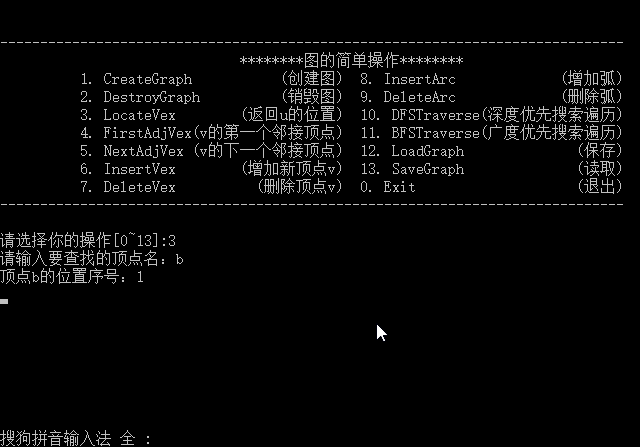


c

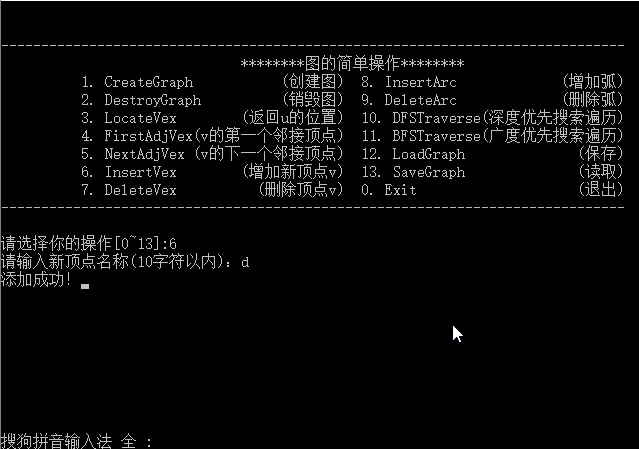
a

b

返回b的位置1



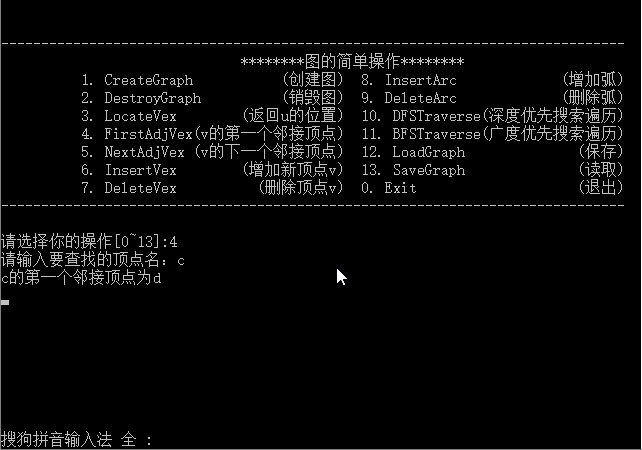
增加新顶点d



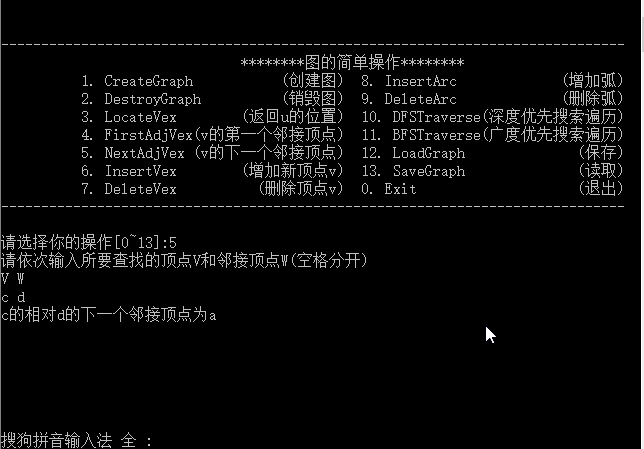
增加弧c d



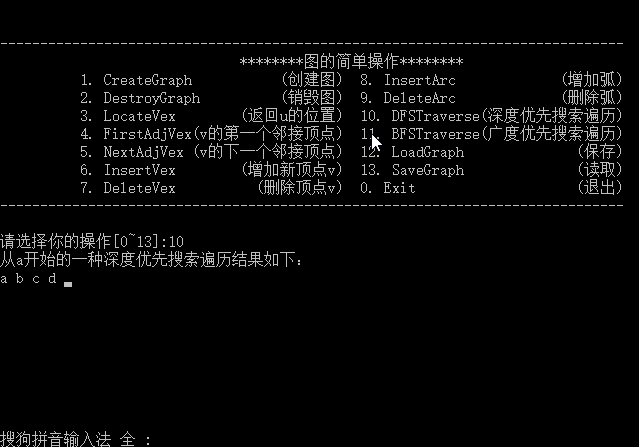
返回c的第一个邻接顶点d



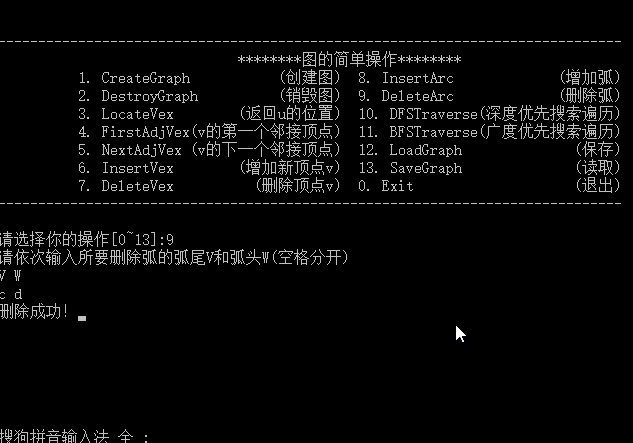
返回c相对d下一个邻接顶点a



遍历



删除弧c d



删除顶点d



## 6.5实验总结与评价

这次实验基本能够熟练掌握关于图的各项基本操作，个人认为这次实验核心在于用邻接表储存图的结构，以及深度、广度优先遍历图的方法的掌握。实验过程中遇到些小问题，都通过问同学一一解决了。数据结构的实验到此差不多快要结束了，希望自己能通过寒假的课设更加系统的增强自己在编程这方面的能力，将数据结构课上所学加以实践，编程之路还很漫长，还需更加努力！

**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