

**课 程 实 验 报 告**

**课程名称： 数据结构实验**

**专业班级： 计算机科学与技术**

**学 号：**

**姓 名：**

**指导教师： 袁凌**

**报告日期： 2017年 11月 16 日**

**计算机科学与技术学院**

目 录

[1 基于顺序存储结构的线性表实现 3](#_Toc531532121)

[1.1 问题描述 3](#_Toc531532122)

[1.1.1 实验要完成的顺序表函数 3](#_Toc531532123)

[1.1.2 实验目标 4](#_Toc531532124)

[1.2 系统设计 4](#_Toc531532125)

[1.2.1 系统总体设计 4](#_Toc531532126)

[1.2.2 有关常量和类型定义 5](#_Toc531532127)

[1.2.3 函数设计 5](#_Toc531532128)

[1.3 系统实现 7](#_Toc531532129)

[1.3.1 系统测试 7](#_Toc531532130)

[1.4 实验小结 16](#_Toc531532131)

[2 基于顺序存储结构的线性表实现 17](#_Toc531532132)

[2.1 问题描述 17](#_Toc531532133)

[2.1.1 实验要完成的顺序表函数 17](#_Toc531532134)

[2.1.2 实验目标 18](#_Toc531532135)

[2.2 系统设计 18](#_Toc531532136)

[2.2.1 系统总体设计 18](#_Toc531532137)

[1.2.2 有关常量和类型定义 19](#_Toc531532138)

[2.2.3 函数设计 19](#_Toc531532139)

[2.3 系统实现 21](#_Toc531532140)

[2.3.1 系统测试 21](#_Toc531532141)

[2.4 实验小结 30](#_Toc531532142)

[3 基于二叉链表的二叉树实现 33](#_Toc531532143)

[3.1 问题描述 33](#_Toc531532144)

[3.2 系统设计 33](#_Toc531532145)

[3.3 系统实现 33](#_Toc531532146)

[3.4 实验小结 33](#_Toc531532147)

[4 基于邻接表的图实现 34](#_Toc531532148)

[4.1 问题描述 34](#_Toc531532149)

[4.2 系统设计 34](#_Toc531532150)

[4.3 系统实现 34](#_Toc531532151)

[4.4 实验小结 34](#_Toc531532152)

[参考文献 35](#_Toc531532153)

[附录A 基于顺序存储结构线性表实现的源程序 37](#_Toc531532154)

[附录B 基于链式存储结构线性表实现的源程序 59](#_Toc531532155)

[附录C 基于二叉链表二叉树实现的源程序 72](#_Toc531532156)

[附录D 基于邻接表图实现的源程序 74](#_Toc531532157)

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

线性表在物理内存中可以以顺序表的方式实现，即物理上存储位置相邻的两个元素是线性表中的相邻元素，且数据元素的前后关系不变。需要完成下述函数。

## 1.1.1 实验要完成的顺序表函数

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

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

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

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

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

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

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

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

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

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

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

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

## 1.1.2 实验目标

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 系统设计

1.2.1 系统总体设计

本系统采用顺序表作为线性表的物理结构，实现线性表的基本运算。

系统具有一个功能菜单。在主程序中完成函数调用所需实参值的准备和函数执行结果的现实，并给出适当的操作提示显示。

本系统通过维护一个指向SqList指针的指针数组ListTracker，和一个用于指向当前操作表的指针的L，来实现多个线性表的管理。每当要对线性表进行操作的时候，就利用指针L来实现，当选择切换表时则利用ListTracker修改L的值，实现对多个线性表的管理。

系统开始运行的时候默认不使用文件中的数据，但是用户随时可以将文件中的数据导入到内存中，同时提供数据保存的功能

该演示系统提供的操作有：表的初始化、销毁、清空、判空，求表长、获取数据元素、查找数据元素、获得前驱、获得后继、插入数据元素、删除数据元素、表的遍历、显示当前表的基本信息(为了调试而给出的选项)、定位元素、表的选择、数据保存、导入数据。

在程序中实现消息处理和操作提示，包括数据的输入和输出，错误操作提示、程序的退出。

1.2.2 有关常量和类型定义

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR -1 //为了与FALSE区分开来，改为了-1

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

#define MAX\_LIST\_NUM 10 //可创建的线性表最大数量

#define bool int //使用bool表意更为明确

#define true 1

#define false 0

typedef int status;

typedef int ElemType; //数据元素类型定义

typedef struct{ //线性表（顺序结构）的定义

ElemType \* elem;

int length;

int listsize;

}SqList;

SqList \* ListTracker[MAX\_LIST\_NUM]; //储存所有数据表的地址

1.2.3 函数设计

（1）InitList(SqList \* L)

设计：为当前选择的表分配存储空间，并初始化表长为0，表容量为LIST\_INIT\_SIZE。

操作结果：初始化当前选择的表为一个空的线性表。

（2）DestroyList(SqList \* L)

设计：释放当前表的储存空间，但并不将指向当前表的L直接指向到存在的表，需要

用户重新选择表

操作结果：销毁当前线性表L。

（3）ClearList(SqList \* L)

设计：当前线性表L的长度赋值为0

操作结果：将L重置为空表。

（4）ListEmpty(SqList L)

设计：根据当前表表长判断表是否为空

操作结果：若L为空表，则返回TRUE,否则返回FALSE。

（5）ListLength(SqList L)

设计：返回当前表表长

操作结果：返回L中数据元素的个数。

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

设计：根据位序找到当前表中第i个元素的地址并将其值赋值给指针e指向的元素

操作结果：用指针e指向的元素返回L中第i个数据元素的值。

（7）LocateElem(SqList L, ElemType e, compare)

设计：遍历当前线性表找到第一个与元素e满足compare关系的的元素(即函数compare返回true)

操作结果：返回L中第1个与e相等的的数据元素的位序，若这样的数据元素不存在，则返回值为0。

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

设计：遍历当前线性表找到第一个和元素cur的相等的元素，如果其有前驱，用pre\_e返回，函数返回TRUE；否则函数返回FALSE，pre\_e无意义，其中，若找到的元素为列表的第一个元素，则返回ERROR，并提示找到的元素为第一个元素

操作结果：若cur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）

设计：遍历线性表找到第一个和元素cur的相等的元素，如果其有后继，用next\_e返回，函数返回TRUE；否则函数返回FALSE，next\_e无意义，其中，若找到的元素为列表的最后一个元素，则返回ERROR，并提示找到的元素为最后一个元素

操作结果：若cur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(SqList \* L, int i, ElemType e)

设计：如果当前线性表已满，重新分配存储空间。将线性表指针L指向的线性表第i个元素之后的元素都右移一个位序，之后将e插入第i个位序

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

设计：将当前表中第i个位序的值赋给指针e指向的变量，之后第i个位序之后的元素全部左移一个位序

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(SqList L, visit)

设计：遍历表L中元素，对每一个元素调用visit函数，由于设计中的visit函数为打印元素的值，所以最后效果为打印每一个元素的值

操作结果：对每一个元素调用visit函数，在该程序的具体背景下，最终效果为打印所有元素

（13）DescribeList(void)

设计：为了调试而给出的操作，实际上并不以函数的方式给出。给出当前表的基本信息

操作结果：打印表长，当前分配给表的空间，以及当前表的序号

（14）ChangeList(void)

设计：为了实现多表管理而给出的操作，实际上并不以函数的方式给出。根据用户输入的数字将表切换到对应的表。

操作结果：切换表

（15）SaveData(ListTracker)

设计：将ListTracker所管理的所有表按照一定格式以文件的形式保存下来

操作结果：保存所有表的数据

（16）LoadData(ListTracker)

设计：尝试将同文件下的文件中的数据加载到ListTracker中，若当前已存在表，则会覆盖

操作结果：导入表的数据，覆盖当前数据

## 1.3 系统实现

编译环境：Windows下使用mingw-gcc编译，使用codeblocks自带工具格式化。程序清单如附录A所示

1.3.1 系统测试

测试数据：

表1：

100 120 200

表2

1 2 3

表4：

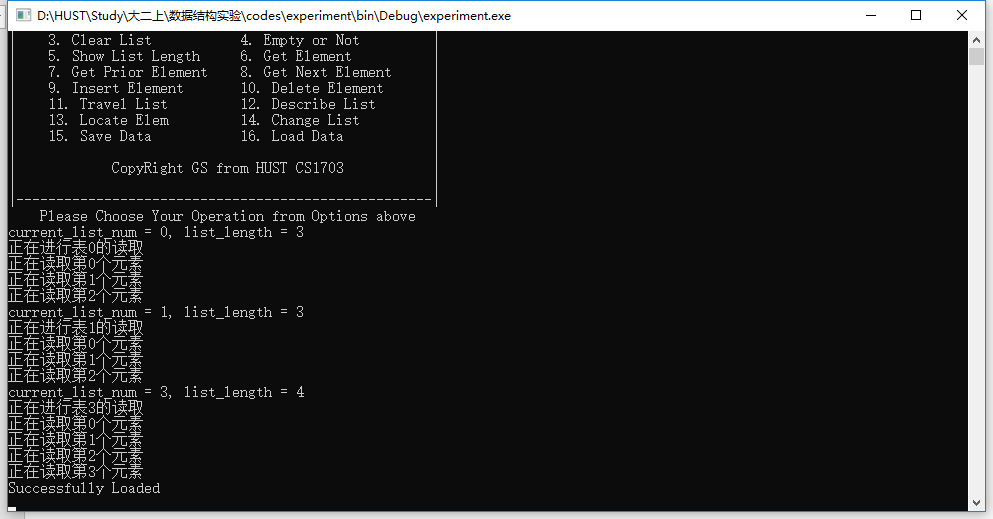
100 56 20 20

测试用例及其结果如下（各函数测试为独立测试，测试初始数据相同，不受上个函数测试影响）：

1. LoadData

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入16进入函数 | 输出各个线性表导入时的log以及每次导入的表的基本信息 | 如图1-1 所示 |

**表1-1LoadData函数测试**



**图1-1 LoadData 函数测试**

1. InitaList

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入1 进入函数 | 输出提示：List successfully initialized | 如图1-2所示 |

**表1-2InitList函数测试**



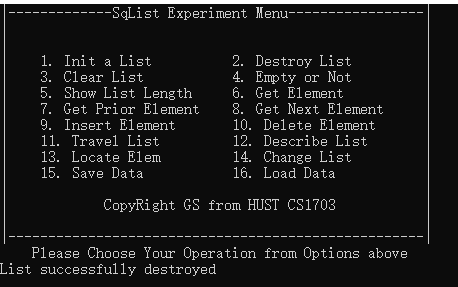
**图1-2 InitList 函数测试**

1. DestroyList

主界面导入数据后选择第一个表，进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 输入2 进入函数 | 提示List successfully destroyed | 如图1-3 所示 |

**表1-3DestroyList函数测试**



**图1-3 Destroy函数测试**

1. ListEmpty

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入1初始化表 2. 主界面输入4进入函数 | 输出提示：List is empty |  |
| 2 | 1. 主界面输入16 导入数据 2. 主界面输入4 进入函数 | 输出提示：List is not empty |  |

**表1-4 ListEmpty函数测试**

1. ListLength

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入5进入函数 | 输出：The Length of the List is 3 |  |

**表1-5 ListLength函数测试**

1. GetElem

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入6进入函数 | 输出：Please enter the position you want to query:(between 1 to 3) | 输出：如图1-6所示 |
| 2 | 输入2 表示查找第二个位置的数据 | 输出：Element of position 2 is 120 | 输出：如图1-6所示 |

**表1-6 GetElem函数测试**



**图1-6 GetElem函数测试**

1. PriorElem

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入7进入函数 | 输出：Please enter the elem you want to query: | 输出：如图1-7-a 所示 |
| 2 | 输入100 表示查找值为100的数据的前驱 | 输出：The target element is the first element | 输出：如图1-7-a 所示 |
| 3 | 1. 主界面输入7进入函数 2. 输入120 表示查找值为120的数据的前驱 | 输出：The prior element of 120 is 100 | 输出：如图1-7-b 所示 |
| 4 | 1. 主界面出入7进入函数 2. 输入0 表示查找值为0的元素的前驱 | 输出：Can't find Target Element | 输出：如图1-7-c 所示 |

**表1-7 PriorElem函数测试**



**图1-7-a PriorElem函数测试**



**图1-7-b PriorElem函数测试**



**图1-7-c PriorElem 函数测试**

1. NextElem

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入7进入函数 | 输出：Please enter the elem you want to query: | 输出：如图1-8-a 所示 |
| 2 | 输入200 表示查找值为100的数据的后继 | 输出：The target element is the last element | 输出：如图1-8-a 所示 |
| 3 | 1. 主界面输入7进入函数 2. 输入120 表示查找值为120的数据的后继 | 输出：The prior element of 120 is 200 | 输出：如图1-8-b 所示 |
| 4 | 1. 主界面出入7进入函数 2. 输入0 表示查找值为0的元素的前驱 | 输出：Can't find Target Element | 输出：如图1-7-c 所示 |

**表1-8 NextElem函数测试**



**图1-8-a NextElem测试**



**图1-8-b NextElem测试**



**图1-8-c NextElem测试**

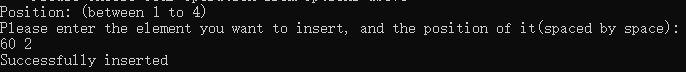
1. ListInsert

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入11打印所有元素 | 输出：100 120 200  successfully traveled all elements | 如图1-9-a所示 |
| 2 | 输入9 进入函数 | 输出：Position: (between 1 to 4)  Please enter the element you want to insert, and the position of it(spaced by space): | 如图1-9-b所示 |
| 3 | 输入60 2表示在第二个位置插入数据60 | 提示：Successfully inserted |
| 4 | 输入11 遍历表 | 输出100 60 120 200  successfully traveled all elements | 100 60 120 200  successfully traveled all elements |

**表1-9 ListInsert函数测试**



**图1-9-a ListInsert函数测试**



**图1-9-b ListInsert函数测试**



**图1-9-c ListInsert 函数测试**

1. ListDelete

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入11打印所有元素 | 输出：100 120 200  successfully traveled all elements | 输出：如图1-10-a所示 |
| 2 | 输入10进入函数 | 输出：Please enter the position of the element you want to delete(between 1 to 3): | 输出：如图1-10-b所示 |
| 3 | 输入2 表示删除第二个位置的数据 | 提示：Successfully deleted 120 in position 2 | 输出：如图1-10-b所示 |
| 4 | 输入11 遍历表 | 输出100 200  successfully traveled all elements | 输出：如图1-10-c所示 |

**表1-10 ListDelete函数测试**



**图1-10-a ListDelete测试**



**图1-10-b ListDelete测试**



**图1-10-c ListDelete测试**

1. ListTrabverse

导入数据后进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入11进入函数 | 输出：100 120 200  successfully traveled all elements | 输出：如图1-11所示 |

**表1-1ListTrabverse函数测试**



**图1-11 ListTrabverse测试**

1. LocateElem

导入数据后进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入13进入函数 2. 输入100 | 输出：Please enter the elem you want to locate with function compare  The position of 100 is 1 | 输出：如图1-12所示 |

**表1-12 LocateElem 函数测试**



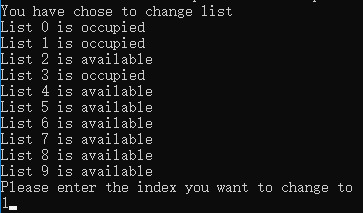
**图1-12 LocateElem 函数测试**

1. ChangeList 功能测试

导入数据后进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入14进入函数 2. 输入1 | 输出：You have chose to change list  List 0 is occupied  List 1 is occupied  List 2 is available  List 3 is occupied  List 4 is available  List 5 is available  List 6 is available  List 7 is available  List 8 is available  List 9 is available  Please enter the index you want to change to  Successfully changed | 输出：如图1-13-a所示 |
| 2 | 主界面输入11遍历表 | 输出：1 2 3  successfully traveled all elements | 输出：如图1-13-b所示 |

**表1-13 ChangeList功能测试**



**图1-13-a ChangeList功能测试**



**图1-13-b ChangeList功能测试**

1. SaveData函数测试

导入数据后进行操作

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入9插入 2. 输入82 3表示在第三个位置插入数据82 | 输出：Position: (between 1 to 4)  Please enter the element you want to insert, and the position of it(spaced by space): | 如图1-14-a所示 |
| 2 | 主界面输入16保存数据 | 提示：Successfully Saved，并在文件中可以看到修改 | 结果如图1-14-b所示 |

**表1-14 SaveData函数测试**



**图1-14-a SaveDate 函数测试**

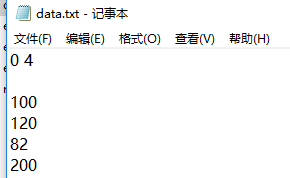


图1-14-b SaveData函数测试

## 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和物理结构之间的关系。以下纪录实现过程中遇到的问题以及解决方案

1. Destroy后直接遍历表发生错误

问题描述：

在任意过程中，选择Destroy当前表后直接遍历，会发生错误。

问题解决：

Destroy函数值负责将ListTracker中纪录的当前表摧毁，但是我却忘记了将标识当前表的指针L置为NULL，导致检查函数checkList 没有检测到异常状态，进而出错

1. InitList函数可以重复执行，导致之前输入的数据被覆盖

问题描述：同上

问题解决：在Init函数之前再添加一个判断条件，如果当前指针L不为NULL，则提示不能重复初始化同一个表

# 2 基于顺序存储结构的线性表实现

## 问题描述

线性表在物理内存中可以以链表表的方式实现，即物理上存储位置不一定相邻的两个元素通过指针连结，成为线性表中的相邻元素，且数据元素的前后关系不变。

## 实验要完成的顺序表函数

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

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

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

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

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

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

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

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

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

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

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

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

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

## 2.1.2 实验目标

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. 物理结构采用单链表,熟练掌握线性表的基本运算的实现。

## 2.2 系统设计

2.2.1 系统总体设计

本系统采用链表作为线性表的物理结构，实现线性表的基本运算。

系统具有一个功能菜单。在主程序中完成函数调用所需实参值的准备和函数执行结果的现实，并给出适当的操作提示显示。

本系统通过维护一个List结构数组ListTracker，用于存放头节点，和一个用于指向当前操作表的指针的L，来实现多个线性表的管理。每当要对线性表进行操作的时候，就通过修改指针L来实现，当选择切换表时则利用ListTracker修改L的值，实现对多个线性表的管理。

系统开始运行的时候默认不使用文件中的数据，但是用户随时可以将文件中的数据导入到内存中，同时提供数据保存的功能

系统开始运行时调用函数读取文件中的数据，并提供数据保存功能以实现线性表的文件形式保存。

该演示系统提供的操作有：表的初始化、销毁、清空、判空，求表长、获取数据元素、查找数据元素、获得前驱、获得后继、插入数据元素、删除数据元素、表的遍历、显示当前表的基本信息(为了调试而给出的选项)、定位元素、表的选择、数据保存、导入数据。

在程序中实现消息处理和操作提示，包括数据的输入和输出，错误操作提示、程序的退出。

2.2.2 有关常量和类型定义

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR -1 //为了与FALSE区分开来，改为了-1

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

#define MAX\_LIST\_NUM 10 //可创建的线性表最大数量

#define bool int //使用bool表意更为明确

#define true 1

#define false 0

typedef int status;

typedef int ElemType; //数据元素类型定义

typedef struct{ //线性表（顺序结构）的定义

ElemType \* elem;

int length;

int listsize;

}SqList;

SqList \* ListTracker[MAX\_LIST\_NUM]; //储存所有数据表的地址

2.2.3 函数设计

（1）InitList(SqList \* L)

设计：为当前选择的表分配存储空间，并初始化表长为0，表容量为LIST\_INIT\_SIZE。

操作结果：初始化当前选择的表为一个空的线性表。

（2）DestroyList(SqList \* L)

设计：释放当前表的储存空间，但并不将指向当前表的L直接指向到存在的表，需要

用户重新选择表

操作结果：销毁当前线性表L。

（3）ClearList(SqList \* L)

设计：当前线性表L的长度赋值为0

操作结果：将L重置为空表。

（4）ListEmpty(SqList L)

设计：根据当前表表长判断表是否为空

操作结果：若L为空表，则返回TRUE,否则返回FALSE。

（5）ListLength(SqList L)

设计：返回当前表表长

操作结果：返回L中数据元素的个数。

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

设计：根据位序找到当前表中第i个元素的地址并将其值赋值给指针e指向的元素

操作结果：用指针e指向的元素返回L中第i个数据元素的值。

（7）LocateElem(SqList L, ElemType e, compare)

设计：遍历当前线性表找到第一个与元素e满足compare关系的的元素(即函数compare返回true)

操作结果：返回L中第1个与e相等的的数据元素的位序，若这样的数据元素不存在，则返回值为0。

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

设计：遍历当前线性表找到第一个和元素cur的相等的元素，如果其有前驱，用pre\_e返回，函数返回TRUE；否则函数返回FALSE，pre\_e无意义，其中，若找到的元素为列表的第一个元素，则返回ERROR，并提示找到的元素为第一个元素

操作结果：若cur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）

设计：遍历线性表找到第一个和元素cur的相等的元素，如果其有后继，用next\_e返回，函数返回TRUE；否则函数返回FALSE，next\_e无意义，其中，若找到的元素为列表的最后一个元素，则返回ERROR，并提示找到的元素为最后一个元素

操作结果：若cur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(SqList \* L, int i, ElemType e)

设计：如果当前线性表已满，重新分配存储空间。将线性表指针L指向的线性表第i个元素之后的元素都右移一个位序，之后将e插入第i个位序

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

设计：将当前表中第i个位序的值赋给指针e指向的变量，之后第i个位序之后的元素全部左移一个位序

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(SqList L, visit)

设计：遍历表L中元素，对每一个元素调用visit函数，由于设计中的visit函数为打印元素的值，所以最后效果为打印每一个元素的值

操作结果：对每一个元素调用visit函数，在该程序的具体背景下，最终效果为打印所有元素

（13）DescribeList(void)

设计：为了调试而给出的操作，实际上并不以函数的方式给出。给出当前表的基本信息

操作结果：打印表长，当前分配给表的空间，以及当前表的序号

（14）ChangeList(void)

设计：为了实现多表管理而给出的操作，实际上并不以函数的方式给出。根据用户输入的数字将表切换到对应的表。

操作结果：切换表

（15）SaveData(ListTracker)

设计：将ListTracker所管理的所有表按照一定格式以文件的形式保存下来

操作结果：保存所有表的数据

（16）LoadData(ListTracker)

设计：尝试将同文件下的文件中的数据加载到ListTracker中，若当前已存在表，则会覆盖

操作结果：导入表的数据，覆盖当前数据

## 2.3 系统实现

编译环境：Windows下使用mingw-gcc编译，使用codeblocks自带工具格式化。程序清单如附录A所示

* + 1. 系统测试

测试数据：

表1：

100 120 200

表2

1 2 3

表4：

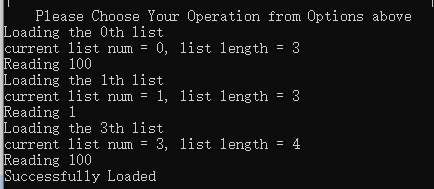
100 56 20 20

测试用例及其结果如下（各函数测试为独立测试，测试初始数据相同，不受上个函数测试影响）：

1. LoadData

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入16进入函数 | 输出各个线性表导入时的log以及每次导入的表的基本信息 | 如图2-1 所示 |

**表2-1LoadData函数测试**



**图2-1 LoadData 函数测试**

1. InitaList

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入1 进入函数 | 输出提示：List successfully initialized | 如图2-2所示 |

**表2-2InitList函数测试**



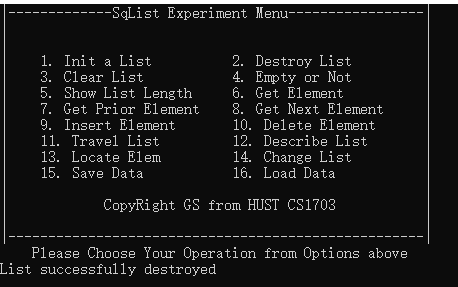
**图2-2 InitList 函数测试**

1. DestroyList

主界面导入数据后选择第一个表，进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 输入2 进入函数 | 提示List successfully destroyed | 如图2-3 所示 |

**表2-3DestroyList函数测试**



**图2-3 Destroy函数测试**

1. ListEmpty

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入1初始化表 2. 主界面输入4进入函数 | 输出提示：List is empty |  |
| 2 | 1. 主界面输入16 导入数据 2. 主界面输入4 进入函数 | 输出提示：List is not empty |  |

**表2-4 ListEmpty函数测试**

1. ListLength

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入5进入函数 | 输出：The Length of the List is 3 |  |

**表2-5 ListLength函数测试**

1. GetElem

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入6进入函数 | 输出：Please enter the position you want to query:(between 1 to 3) | 输出：如图2-6所示 |
| 2 | 输入2 表示查找第二个位置的数据 | 输出：Element of position 2 is 120 | 输出：如图2-6所示 |

**表2-6 GetElem函数测试**



**图2-6 GetElem函数测试**

1. PriorElem

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入7进入函数 | 输出：Please enter the elem you want to query: | 输出：如图2-7-a 所示 |
| 2 | 输入100 表示查找值为100的数据的前驱 | 输出：The target element is the first element |
| 3 | 1. 主界面输入7进入函数 2. 输入120 表示查找值为120的数据的前驱 | 输出：The prior element of 120 is 100 | 输出：如图2-7-b 所示 |
| 4 | 1. 主界面出入7进入函数 2. 输入0 表示查找值为0的元素的前驱 | 输出：Can't find Target Element | 输出：如图2-7-c 所示 |

**表2-7 PriorElem函数测试**



**图2-7-a PriorElem函数测试** **图2-7-b PriorElem函数测试**



**图2-7-c PriorElem 函数测试**

1. NextElem

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入7进入函数 | 输出：Please enter the elem you want to query: | 输出：如图2-8-a 所示 |
| 2 | 输入200 表示查找值为100的数据的后继 | 输出：The target element is the last element | 输出：如图2-8-a 所示 |
| 3 | 1. 主界面输入7进入函数 2. 输入120 表示查找值为120的数据的后继 | 输出：The prior element of 120 is 200 | 输出：如图2-8-b 所示 |
| 4 | 1. 主界面出入7进入函数 2. 输入0 表示查找值为0的元素的前驱 | 输出：Can't find Target Element | 输出：如图2-7-c 所示 |

**表2-8 NextElem函数测试**



**图2-8-a NextElem测试**  **图2-8-b NextElem测试**



**图2-8-c NextElem测试**

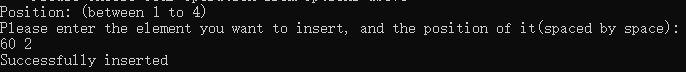
1. ListInsert

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入11打印所有元素 | 输出：100 120 200  successfully traveled all elements | 输出：如图2-9-a所示 |
| 2 | 输入9 进入函数 | 输出：Position: (between 1 to 4)  Please enter the element you want to insert, and the position of it(spaced by space): | 输出，如图2-9-b所示 |
| 3 | 输入60 2表示在第二个位置插入数据60 | 提示：Successfully inserted |
| 4 | 输入11 遍历表 | 输出100 60 120 200  successfully traveled all elements | 100 60 120 200  successfully traveled all elements |

**表2-9 ListInsert函数测试**



**图2-9-a ListInsert函数测试**



**图2-9-b ListInsert函数测试**



**图2-9-c ListInsert 函数测试**

1. ListDelete

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入16 导入数据 2. 主界面输入11打印所有元素 | 输出：100 120 200  successfully traveled all elements | 输出：如图2-10-a所示 |
| 2 | 输入10进入函数 | 输出：Please enter the position of the element you want to delete(between 1 to 3): | 输出：如图2-10-b所示 |
| 3 | 输入2 表示删除第二个位置的数据 | 提示：Successfully deleted 120 in position 2 |
| 4 | 输入11 遍历表 | 输出100 200  successfully traveled all elements | 输出：如图2-10-c所示 |

**表2-10 ListDelete函数测试**



**图2-10-a ListDelete测试**



**图2-10-b ListDelete测试**



**图2-10-c ListDelete测试**

1. ListTrabverse

导入数据后进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入11进入函数 | 输出：100 120 200  successfully traveled all elements | 输出：如图2-11所示 |

**表2-11ListTrabverse函数测试**



**图2-11 ListTrabverse测试**

1. LocateElem

导入数据后进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入13进入函数 2. 输入100 | 输出：Please enter the elem you want to locate with function compare  The position of 100 is 1 | 输出：如图2-12所示 |

**表2-12 LocateElem 函数测试**



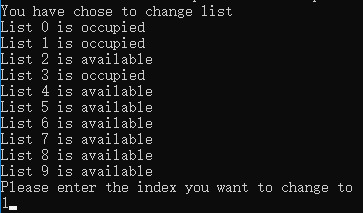
**图2-12 LocateElem 函数测试**

1. ChangeList 功能测试

导入数据后进行测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入14进入函数 2. 输入1 | 输出：You have chose to change list  List 0 is occupied  List 1 is occupied  List 2 is available  List 3 is occupied  List 4 is available  List 5 is available  List 6 is available  List 7 is available  List 8 is available  List 9 is available  Please enter the index you want to change to  Successfully changed | 输出：如图2-13-a所示 |
| 1 | 主界面输入11遍历表 | 输出：1 2 3  successfully traveled all elements | 输出：如图2-13-b所示 |

**表1-13 ChangeList功能测试**



**图2-13-a ChangeList功能测试**



**图2-13-b ChangeList功能测试**

1. SaveData函数测试

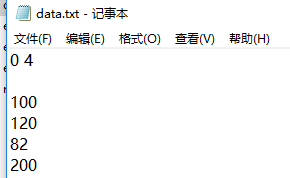
导入数据后进行操作

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 1. 主界面输入9插入 2. 输入82 3表示在第三个位置插入数据82 | 输出：Position: (between 1 to 4)  Please enter the element you want to insert, and the position of it(spaced by space): | 如图2-14-a所示 |
| 2 | 1. 主界面输入16保存数据 | 提示：Successfully Saved，并在文件中可以看到修改 | 结果如图2-14-b所示 |

**表2-14 SaveData函数测试**



**图2-14-a SaveDate 函数测试**



**图2-14-b SaveData函数测试**

## 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和物理结构之间的关系。以下纪录实现过程中遇到的问题以及解决方案

1. 打开文件后没有关闭导致无法正确写入

问题描述：

在导入数据以及写入数据的时候使用fopen之后没有fclose就会出现没有正确导入的情况

问题解决：

发现报错是有关于权限的，发现没有fclose，在加入fclose语句之后问题解决

1. 没有加入头节点

问题描述：最初写的时候没有加入头节点，只有一个头指针，导致无法判断表是否初始化，一些函数无法正常工作

问题解决：

重写代码，加入头节点并且改写所有相关函数的定义

# 3 基于二叉链表的二叉树实现

## 3.1问题描述

通过实验达到⑴加深对二叉树的概念、基本运算的理解；⑵熟练掌握二叉树的逻辑结构与物理结构的关系；⑶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。

## 实验要完成的二叉树函数

依据最小完备性和常用性相结合的原则，以函数形式定义了二叉树的初始化二叉树、销毁二叉树、创建二叉树、清空二叉树、判定空二叉树和求二叉树深度等20种基本运算，具体运算功能定义如下。

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

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

⑶创建二叉树：函数名称是CreateBiTree(T,definition)；初始条件是definition给出二叉树T的定义；操作结果是按definition构造二叉树T。

⑷清空二叉树：函数名称是ClearBiTree (T)；初始条件是二叉树T存在； 操作结果是将二叉树T清空。

⑸判定空二叉树：函数名称是BiTreeEmpty(T)；初始条件是二叉树T存在；操作结果是若T为空二叉树则返回TRUE，否则返回FALSE。

⑹求二叉树深度：函数名称是BiTreeDepth(T)；初始条件是二叉树T存在；操作结果是返回T的深度。

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

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

⑼结点赋值：函数名称是Assign(T,&e,value)；初始条件是二叉树T已存在，e是T中的某个结点；操作结果是结点e赋值为value。

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

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

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

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

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

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

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

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

⒅中序遍历：函数名称是InOrderTraverse(T,Visit))；初始条件是二叉树T存在，Visit是对结点操作的应用函数；操作结果是中序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

⒆后序遍历：函数名称是PostOrderTraverse(T,Visit))；初始条件是二叉树T存在，Visit是对结点操作的应用函数；操作结果是后序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

⒇按层遍历：函数名称是LevelOrderTraverse(T,Visit))；初始条件是二叉树T存在，Visit是对结点操作的应用函数；操作结果是层序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

## 3.2系统设计

3.2.1系统总体设计

本系统采用二叉链表作为二叉树的物理结构，实现二叉树的基本运算。

系统具有一个功能菜单。在主程序中完成函数调用所需实参值的准备和函数执行结果的现实，并给出适当的操作提示显示。

本系统通过维护一个指向root节点的指针数组TreeTracker，和一个用于指向当前操作二叉树头节点的的指针的currentHead，以及一个标识当前树编号的全局变量currentIndex，来实现多个线性表的管理。每当要对线性表进行操作的时候，就通过修改指针currentHead以及currentIndex来实现，当选择切换表时则利用TreeTracker修改currentHead以及currentIndex的值，实现对多个多棵树的管理。

系统开始运行的时候默认不使用文件中的数据，但是用户随时可以将文件中的数据导入到内存中，同时提供数据保存的功能。用户可以通过输入提示表中的数字来进行实验要求当中的相关操作。系统也提供了简单的横向打印二叉树的功能，可以帮助直观的显示二叉树的结构。

在程序中也实现了对队列这种数据结构的简单包装，用于层遍历以及选择元素时使用

在程序中实现消息处理和操作提示，包括数据的输入和输出，错误操作提示、程序的退出。

3.2.2有关常量和类型定义

#define MAX\_TREE\_NUM 10

#define OK 1

#define ERROR -1

#define true 1

#define false 0

#define bool int

typedef int status;

typedef int ElemType;

using namespace std;

struct TreeNode

{

ElemType elem;

TreeNode \* LChild;

TreeNode \* RChild;

};

struct Quene

{

TreeNode \* elem;

Quene \* next;

};

struct HeadNode

{

TreeNode \* Tree;

};

HeadNode \* currentHead; //用于存放当指向当前树的头节点的指针

HeadNode \* TreeTracker[MAX\_TREE\_NUM]; //头指针数组

int currentIndex; //指示当前树的序号

myQuene quene;

myQuene quene\_for\_select;

int getElemConst;

int getElemTarget;

TreeNode \* tempTarget;

//以下为简单包装的一个队列类，可以使用简单的push、pop等操作

class myQuene {

public:

Quene \* ListHead = nullptr;

int listlength = 0;

void push(TreeNode \* tarelem) {

if (ListHead) {

Quene \* temp = ListHead;

while (temp->next)

temp = temp->next;

temp->next = (Quene \*)malloc(sizeof(Quene));

temp->next->elem = tarelem;

temp->next->next = NULL;

}

else {

ListHead = (Quene \*)malloc(sizeof(Quene));

ListHead->elem = tarelem;

ListHead->next = nullptr;

}

listlength++;

}

TreeNode \* pop() {

Quene \* temp = ListHead;

TreeNode \* tempValue = temp->elem;

ListHead = temp->next;

free(temp);

listlength--;

return tempValue;

}

void clearAll() {

ListHead = nullptr;

}

};

## 3.3系统实现

* + 1. 系统测试

测试数据：(用# 表示空，用前序表示)

树一： 1 2 # # 3 4 # # #



树二： 1 2 # # #



测试用例及其结果如下（各函数测试为独立测试，测试初始数据相同，不受上个函数测试影响）：

1. LoadData();

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23进入函数 | 提示导入成功 |  |
| 2 | 主界面输入21切换树  输入0表示切换到0号树  输入17进行前序遍历 | 输出1 2 3 4并提示遍历成功 |  |

**表3-1 LoadData函数测试**

1. InitBiTree()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入1进入函数 | 提示初始化成功 |  |

**表3-2 InitBiTree函数测试**

1. DestroyBiTree()

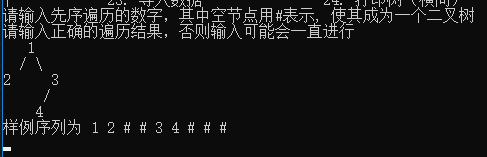
|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入2销毁当前（0号）树 | 提示销毁成功 |  |

**表3-3 DestroyBiTree函数测试**

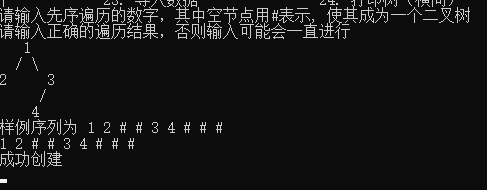
1. CreateBiTree()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入1初始化当前树  输入5进入函数 | 提示初始化成功并给出创建树的相关提示 | 如图3-4-1所示 |
| 2 | 输入1 2 # # 3 4 # # # | 提示初始化成功 | 如图3-4-2所示 |

**表3-4 CreateBiTree函数测试**



**图3-4-1**



**图3-4-2**

1. ClearBiTree()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入3清空当前（0号）树 | 提示清空成功 |  |

**表3-5 ClearBiTree函数测试**

1. BiTreeEmpty()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入4判断是否为空 | 提示当前树不为空 |  |
| 3 | 输入3清空当前树(0号)  输入4判断是否为空 | 提示清空后提示当前树为空 |  |

**表3-6 BiTreeEmpty函数测试**

1. depth()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入8获取当前树的深度 | 输出当前树深度为3 |  |
| 3 | 输入21切换到1号树  输入8或缺当前树的深度 | 输出当前树的深度为2 |  |

**表3-7 depth函数测试**

1. Root()

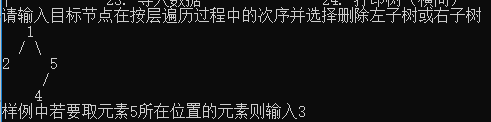
|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入6获取当前树的根节点 | 输出当前树的根节点为1 |  |

**表3-8 Root函数测试**

1. Value()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入7进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入3表示获取按层遍历过程中对应的元素 | 输出3 |  |

**表3-9 Value函数测试**



**图3-9-1**

1. Assign()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入9进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入2表示获取按层遍历过程中对应的元素 | 提示选中的元素为2，并要求用户输入新的值 |  |
| 4 | 输入5表示将其改为5 | 提示成功修改 |  |

**表3-10 Assign函数测试**

1. Parent()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入10进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入3表示获取按层遍历过程中对应的元素 | 输出节点的值以及父节点（如果存在的话） |  |

**表3-11 Parent函数测试**

1. LeftChild()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入11进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入3表示获取按层遍历过程中对应的元素 | 输出节点的值以及左子节点（如果存在的话） |  |

**表3-12 LeftChild函数测试**

1. RightChild()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入12进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入1表示获取按层遍历过程中对应的元素 | 输出节点的值以及右子节点（如果存在的话） |  |

**表3-13 RightChild测试**

1. LeftSibling()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入13进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入3表示获取按层遍历过程中对应的元素 | 输出节点的值以及左邻节点（如果存在的话） |  |

**表3-14 LeftSibling函数测试**

1. RightSibling()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入14进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入2表示获取按层遍历过程中对应的元素 | 输出节点的值以及右邻节点（如果存在的话） |  |

**表3-15 RightSibling函数测试**

1. InsertChild()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入15进入函数 | 输出所有树的可用信息，供用户选择 |  |
| 3 | 输入1表示将1号树插入当前树 | 提示用户继续选择插入的位置 |  |
| 4 | 输入3表示插在三号元素之后  输入1表示作为右子树插入 | 提示插入成功 |  |

**表3-16 InsertChild函数测试**

1. DeleteChild()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入16进入函数 | 输出接受的输入规范 | 如图3-9-1所示 |
| 3 | 输入1 1表示删除第一个节点的右子树 | 提示删除成功 |  |

**表3-17 DeleteChild函数测试**

1. PreOrderTraverse()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入17进入函数 | 输出前序遍历结果 |  |

**表3-18 PreOrderTraverse函数测试**

1. InOrderTraverse()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入18进入函数 | 输出中序遍历结果 |  |

**表3-19 InsertTraverse函数测试**

1. PostOrderTraverse()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入19进入函数 | 输出后序遍历结果 |  |

**表3-20 PostOrderTraverse函数测试**

1. LevelOrderTraverse()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入23导入数据 | 导入成功 |  |
| 2 | 主界面输入21切换到0号树  输入20进入函数 | 输出后序遍历结果 |  |

**表3-21 LevelOrderTraverse函数测试**

## 3.4实验小结

本次实验较前两次实验来讲难度有了比较大的提高。主要在两点上花费了很多时间，并且最后由于时间关系也没有很好的解决。

1. 无法以一种很好的方式把二叉树纵向打印出来，而无法把二叉树打印出来的话，在调试其他函数的时候就会很麻烦，最后由于时间关系选择了横向打印二叉树
2. 一开始总是想让用户以按层遍历这种比较舒服的方式输入二叉树的初始数据，但是由于使用了C++的reference，还不是很熟练，就频繁出现segment fault错误，最终放弃了这个方案，而使用前序遍历的方式输入，直接用递归解决问题

# 4 基于邻接表的图实现

## 问题描述

通过实验达到⑴加深对图的概念、基本运算的理解；⑵熟练掌握图的逻辑结构与物理结构的关系；⑶以邻接表作为物理结构，熟练掌握图基本运算的实现。

4.1.1实验要完成的图函数

依据最小完备性和常用性相结合的原则，以函数形式定义了创建图、销毁图、查找顶点、获得顶点值和顶点赋值等13种基本运算，具体运算功能定义如下。

⑴创建图：函数名称是CreateCraph(&G,V,VR)；初始条件是V是图的顶点集，VR是图的关系集；操作结果是按V和VR的定义构造图G。

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

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

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

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

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

⑺获得下一邻接点：函数名称是NextAdjVex(&G, v, w)；初始条件是图G存在，v是G的一个顶点,w是v的邻接顶点；操作结果是返回v的（相对于w）下一个邻接顶点，如果w是最后一个邻接顶点，返回“空”。

⑻插入顶点：函数名称是InsertVex(&G,v)；初始条件是图G存在，v和G中的顶点具有相同特征；操作结果是在图G中增加新顶点v。

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

⑽插入弧：函数名称是InsertArc(&G,v,w)；初始条件是图G存在，v、w是G的顶点；操作结果是在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>。

⑾删除弧：函数名称是DeleteArc(&G,v,w)；初始条件是图G存在，v、w是G的顶点；操作结果是在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>。

⑿深度优先搜索遍历：函数名称是DFSTraverse(G,visit())；初始条件是图G存在；操作结果是图G进行深度优先搜索遍历，依次对图中的每一个顶点使用函数visit访问一次，且仅访问一次。

⒀广深度优先搜索遍历：函数名称是BFSTraverse(G,visit())；初始条件是图G存在；操作结果是图G进行广度优先搜索遍历，依次对图中的每一个顶点使用函数visit访问一次，且仅访问一次。

## 系统设计

4.2.1系统总体设计

本实验采用邻接表作为图的物理结构，实现了上述实验要求中的所有基本运算。ElemType采用最简单的Int型。系统具有一个功能菜单。在主程序中完成函数调用所需实参值的准备和函数执行结果的现实，并给出适当的操作提示显示。

本实验尝试将面向对象编写，Vex节点，Graph图均为对象，主体操作函数定义在Index对象中。每一个图和每一个点都有一个string类型的名字以及一个int类型的值。Index中以vector的形式保存了所有的图对象，Graph中同样以vector的形式保存了所有的本图中的所有顶点对象，顶点中利用链表的结构储存相邻节点，其为Graph中节点的副本。Vector由于其易拓展和顺序插入的特性更适合于存放此类数据。由于队列的实现并不是本次实验的重点，并且在实验三中已经尝试包装了一个队列对象，所以在广度优先遍历中使用了c++ STL中的queue容器。

本实验实现了多表操作以及数据的文件形式存储和读取。用户可以随时保存数据或是导入数据。在程序中实现消息处理和操作提示，包括数据的输入和输出，错误操作提示、程序的退出。

4.2.2有关常量和类型定义

#define OK 1

#define ERROR -1

typedef int status;

typedef int ElemType;

//此处只写出了对象的基本成员以及构造函数和实验要求的功能函数，一些辅助函数在此省略

//Vex 对象表示一个点

class Vex {

public:

ElemType data;

string name;

bool visited;

Vex \* nextVex;

Vex(string name, ElemType data) {

this->data = data;

this->name = name;

this->visited = false;

this->nextVex = nullptr;

}

};

//Graph对象，表示一个图

class Graph {

public:

vector <Vex> List;

string name;

Graph(string graphname) {

name = graphname;

}

}

//Index对象表示一个操作体，所有操作都在其上进行，包括多图的管理和文件的存储

class Index {

public:

int currentGraphIndex;

vector <Graph> List;

status CreateGraph(string graphname);

status DestroyGraph(string graphname);

Vex & LocateVex(string graphname, string vexname);

ElemType GetVex(string graphname, string vexname);

status PutVex(string graphname, string vexname, ElemType value);

Vex \* FirstAdjVex(string graphname, string vexname);

Vex \* NextAdjVex(string graphname, string vexname, string vexname2);

status InsertVex(string graphname, string vexname, ElemType data);

status DeleteVex(string graphname, string vexname);

status InsertArc(string graphname, string vexname1, string vexname2);

status DeleteArc(string graphname, string vexname1, string vexname2);

status DFSTraverse(string graphname, void (\*visit)(Vex));

status BFSTraverse(string graphname, void (\*visit)(Vex));

void SaveData();

void LoadData();

void PrintTable();

private:

void DFS(Graph & G, int position, void(\*visit)(Vex), string graphname);

}

## 系统实现

4.3.1系统测试

测试数据：如图，其中字母为键，数字为值。

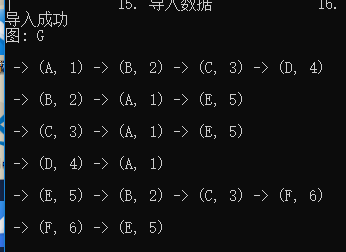


测试用例及其结果如下（各函数测试为独立测试，测试初始数据相同，不受上个函数测试影响）：

1. LoadData();

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入15进入函数 | 提示导入成功并以邻接表的形式打印出图 | 如图4-1所示 |

**表4-1 LoadData函数测试**



**图4-1 LoadData函数测试**

1. CreateGraph():

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入1进入函数  依次输入  G2  3  V1 10  V2 20  V3 30  2  V1 V2  V1 V3  分别表示新建的图的名字，图中含有的节点的个数，每个节点的名字以及值，图中含有边的个数，每条边的liangg 顶点分别为什么。 | 提示创建成功 |  |
| 2 | 输入16打印邻接表 | 输入邻接表形式的图 |  |

**表4-2CreateGraph函数测试**

1. DestroyGraph()

在1，2的基础上进行测试，先导入了图，然后又创建了名为G2的另一个图

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入2进入函数，并输入要销毁的图的名字  G2 | 提示销毁成功 |  |

**表4-3DestroyGraph函数测试**

1. LocateVex():

在1的基础上进行(以下测试若无特殊说明则均在1的基础上进行)

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入3进入函数并依次输入图的名字以及点的名字G A，表示图G中名为A的点 | 显示G中点A的值1 |  |
| 2 | 主界面输入3进入函数并依次输入图的名字以及点的名字G Q，表示图G中名为Q  的点(此为异常测试，当输入的点或者图不存在时，后续函数不再测试) | 提示输入的节点不存在 |  |

**表4-4 LocateVex函数测试**

1. GetVex()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入4进入函数并依次输入图的名字以及点的名字G A，表示图G中名为A的点 | 显示G中点A的值1 |  |

**表4-5 GetVex函数测试**

1. PutVex()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入5进入函数并依次输入图的名字以及点的名字G A，表示图G中名为A的点 | 显示G中点A的值1,并提示输入新值 |  |
| 2 | 输入50，表示将其改为50 | 提示修改成功 |

**表4-6 PutVex函数测试**

1. FirstAdjVex():

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入6进入函数并依次输入图的名字以及点的名字G A，表示图G中名为A的点 | 提示：位于图G中节点A的第一邻接点为名为B且值为2的节点 |  |

**表4-7 FirstAdjVex函数测试**

1. NextAdjVex()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入7进入函数并依次输入图的名字以及点的名字G A B，表示图G中名为A的点相对B的第一邻接点 | 提示：位于图G中节点A相对于B的下一邻接节点为名为C且值为3的节点 |  |

**表4-8 NextAdjVex函数测试**

1. InsertVex()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入8进入函数并依次输入图的名字以及点的名字和值G G 10，表示向图G中插入名为G且值为10 的节点 | 提示插入成功 |  |
| 2 | 输入16打印邻接表 | 输出结果 |  |

**表4-9 InsertVex函数测试**

1. DeleteVex()

在9的基础上进行

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入9进入函数并依次输入图的名字以及点的名字G A，表示删除图G中名为A的节点 | 提示删除成功 |  |
| 2 | 输入16打印邻接表 | 输出结果 |  |

**表4-10 DeleteVex函数测试**

1. InsertArc()

在10的基础上进行

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入10进入函数并依次输入图的名字以及点的名字和值G B G，表示插入弧BG到图G中 | 提示插入成功 |  |
| 2 | 输入16打印邻接表 | 输出结果 |  |

**表4-11 InsertArc函数测试**

1. DeleteArc()

在11的基础上进行

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入11进入函数并依次输入图的名字以及点的名字和值G E F，表示删除图G中连接E F的弧 | 提示删除成功 |  |
| 2 | 输入16打印邻接表 | 输出结果 |  |

**表4-12 DeleteArc函数测试**

1. DFSTraverse()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入12进入函数并输入图的名字G | 输出遍历结果(由于邻接表建立的顺序不同，可能有多种答案) |  |

**表4-13 DFSTraverse函数测试**

1. BFSTraverse()

|  |  |  |  |
| --- | --- | --- | --- |
| 测试步骤 | 测试输入 | 理论结果 | 运行结果 |
| 1 | 主界面输入13进入函数并输入图的名字G | 输出遍历结果(由于邻接表建立的顺序不同，可能有多种答案) |  |

**表4-14 BFSTraverse函数测试**

## 实验小结

在本次实验期间对C++有了一些浅薄的理解，因此这次实验就尽量使用学到的一些C++的语法特性所写。实验中用到了vector这种可以类似于数组的容器作为存放节点以及图的方法，好处就在于可以方便的添加，删减，遍历元素。再者使用了string类型，相比于C中的char \* 还是要方便许多。由于这次实验选择的是无向图，所以实验难度其实比二叉树还要低，实验中遇到的最大困难也就是对于各种异常输入的处理，主体部分基本上都很顺利，最重要的收获就是使用了C++吧。

**参考文献**

[1] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

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

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

[4] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

指导教师评定意见

一、对实验报告的评语

|  |
| --- |
|  |

二、对实验报告评分

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

# 附录A 基于顺序存储结构线性表实现的源程序

#include <stdio.h>

#include <stdlib.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR -1 //为了与FALSE区分开来，改为了-1

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

#define MAX\_LIST\_NUM 10

#define bool int //使用bool表意更为明确

#define true 1

#define false 0

typedef int status;

typedef int ElemType; //此处为了简化难度而直接使用了int而不是结构体

//定义结构体

typedef struct

{

ElemType \* elem;

int length;

int listsize;

} SqList;

status IntiaList(SqList \* L);

status DestroyList(SqList \* ListTracker[MAX\_LIST\_NUM], int currentIndex);

status ClearList(SqList \* L);

bool ListEmpty(SqList L);

int ListLength(SqList L);

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

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

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 ListTrabverse(SqList L, void (\* visite)(ElemType));

status LoadData(SqList \* ListTracker[MAX\_LIST\_NUM]);

//用于遍历时的测试函数

void visit(ElemType item)

{

printf("%d ", item);

}

//用于locate时的测试函数

bool compare(ElemType elem1, ElemType elem2)

{

if (elem1 == elem2)

return true;

else

return false;

}

//用于在每次基本操作时检查当前表是否符合操作所需要求

bool checkList(SqList \* L)

{

if (!L)

{

printf("List doesn't exist\n");

return false;

}

else if (!L->elem)

{

printf("You need to initialize first\n");

return false;

}

else

return true;

}

//用于管理多表，返回所有表的个数

int countListNum(SqList \* ListTracker[MAX\_LIST\_NUM])

{

int count = 0;

int i = 0;

for (; i < MAX\_LIST\_NUM; i++)

if (ListTracker[i])

count++;

return count;

}

//用于管理多表，将每个指针先初始化为NULL

void clearAllList(SqList \* ListTracker[MAX\_LIST\_NUM])

{

int i = 0;

for (; i < MAX\_LIST\_NUM; i ++)

ListTracker[i] = NULL;

}

//用于管理多表时给出信息，供用户参考选择

void printListInfo(SqList \* ListTracker[MAX\_LIST\_NUM])

{

int i = 0;

for (; i < MAX\_LIST\_NUM; i++)

{

if (ListTracker[i])

printf("List %d is occupied\n", i);

else

printf("List %d is available\n", i);

}

}

//用于打印菜单

void printMenu()

{

printf("|-------------SqList Experiment Menu-----------------|\n");

printf("| |\n");

printf("| |\n");

printf("| 1. Init a List 2. Destroy List |\n");

printf("| 3. Clear List 4. Empty or Not |\n");

printf("| 5. Show List Length 6. Get Element |\n");

printf("| 7. Get Prior Element 8. Get Next Element |\n");

printf("| 9. Insert Element 10. Delete Element |\n");

printf("| 11. Travel List 12. Describe List |\n");

printf("| 13. Locate Elem 14. Change List |\n");

printf("| 15. Save Data 16. Load Data |\n");

printf("| |\n");

printf("| CopyRight GS from HUST CS1703 |\n");

printf("| |\n");

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

printf(" Please Choose Your Operation from Options above \n");

}

//主函数

int main(void)

{

SqList \* ListTracker[MAX\_LIST\_NUM];

clearAllList(ListTracker);

//首次使用时默认使用位置0处的表

ListTracker[0] = (SqList \*)malloc(sizeof(SqList));

ListTracker[0]->elem = NULL;

//指定当前表

SqList \* L = ListTracker[0];

//用于指示当前表的位置

int currentIndex = 0;

printMenu();

int op = 1;

while (op)

{

scanf("%d", &op);

//Windows下起到清屏作用

system("cls");

printMenu();

switch(op)

{//以下基本上所有操作都需要利用checkList()先进行检查检查

case 1:

//初始话表的时候需要检查是否处于刚刚执行过destroy操作

if (L == NULL)

{ //如果是，则需要用户冲洗选择一个表

printf("You need to chose a List first\n");

}

else

{ //如果不是，则再进行检查，当前表是否已经初始化过

if (L->elem)//如果是，则提示不不能重复初始化

printf("You can't repeatedly initialize the same List\n");

else

{ //否则则进行正常初始化操作

if (InitaList(L) == OK)

printf("List successfully initialized\n");

else

printf("Failed to create a List\n");

}

}

getchar();

break;

case 2:

if (checkList(L) && DestroyList(ListTracker, currentIndex) == OK)

{

L = NULL;

printf("List successfully destroyed\n");

}

else

printf("Failed to destroy List\n");

getchar();

break;

case 3:

if (checkList(L)){

if (ClearList(L) == OK)

printf("List successfully cleared\n");

else if (InitaList(L) == ERROR)

printf("Failed to clear List\n");

}

getchar();

break;

case 4:

if (checkList(L))

{

if (ListEmpty(\*L))

printf("List is empty\n");

else

printf("List is not empty\n");

}

getchar();

break;

case 5:

if (checkList(L))

{

printf("The length of the list is %d\n", ListLength(\*L));

}

getchar();

break;

case 6:

if (checkList(L))

printf("Please enter the position you want to query:(between 1 to %d)\n", ListLength(\*L));

int queryPosition;

ElemType queryResult;

//用户输入位置

scanf("%d", &queryPosition);

if (GetElem(\*L, queryPosition, &queryResult) != ERROR)

printf("Element of position %d is %d\n", queryPosition, queryResult);

getchar();

break;

case 7:

if (checkList(L))

{

printf("Please enter the elem you want to query:\n");

ElemType queryElem\_prior;

ElemType queryResult\_prior;

//用户输入元素

scanf("%d", &queryElem\_prior);

if (PriorElem(\*L, queryElem\_prior, & queryResult\_prior) != ERROR)

printf("The prior element of %d is %d\n", queryElem\_prior, queryResult\_prior);

else

printf("failed to find\n");

}

getchar();

break;

case 8:

if (checkList(L))

{

printf("Please enter the elem you want to query:\n");

ElemType queryElem\_next;

ElemType queryResult\_next;

//用户输入元素

scanf("%d", &queryElem\_next);

if (NextElem(\*L, queryElem\_next, & queryResult\_next) != ERROR)

printf("The next element of %d is %d\n", queryElem\_next, queryResult\_next);

else

printf("failed to find\n");

}

getchar();

break;

case 9:

if (checkList(L))

{

printf("Position: (between 1 to %d)\n", ListLength(\*L)+1);

printf("Please enter the element you want to insert, and the position of it(spaced by space):\n");

ElemType insertElem;

int insertPosition;

//用户输入

scanf("%d %d", &insertElem, &insertPosition);

if (checkList(L) && ListInsert(L, insertPosition, insertElem) != ERROR)

printf("Successfully inserted\n");

else

printf("failed to insert\n");

}

getchar();

break;

case 10:

if (checkList(L))

printf("Please enter the position of the element you want to delete(between 1 to %d):\n", ListLength(\*L));

ElemType deleteElem;

int deletePosition;

//用户输入

scanf("%d", &deletePosition);

if (checkList(L) && ListDelete(L, deletePosition, &deleteElem) != ERROR)

printf("Successfully deleted %d in position %d\n", deleteElem, deletePosition);

else

printf("failed to deleted\n");

getchar();

break;

case 11:

if (checkList(L))

{

if (ListTrabverse(\*L, visit) == OK)

printf("successfully traveled all elements\n");

else

printf("failed to travel all element\n");

}

getchar();

break;

case 12:

if (checkList(L))

printf("List length = %d, ListSize = %d, ListNum = %d\n", L->length, L->listsize, currentIndex);

else

printf("ListNUM = %d\n", currentIndex);

break;

case 13:

if (checkList(L))

{

printf("Please enter the elem you want to locate with function compare\n");

ElemType queryElem\_locate;

int queryResult\_locate;

//用户输入

scanf("%d", &queryElem\_locate);

if (queryResult\_locate = LocateElem(\*L, queryElem\_locate, compare))

{

printf("The position of %d is %d\n", queryElem\_locate, queryResult\_locate);

}

}

getchar();

break;

case 14:

printf("You have chose to change list\n");

//打印出每个位置的占用信息，供用户选择

printListInfo(ListTracker);

printf("Please enter the index you want to change to\n");

int tempIndex;

scanf("%d", &tempIndex);

if (tempIndex >= 0 && tempIndex < MAX\_LIST\_NUM)

{ //如果用户的选择再规定范围之中

currentIndex = tempIndex;

if (!ListTracker[currentIndex])

{ //如果用户所选的位置还没有分配空间。则分配

ListTracker[currentIndex] = (SqList \*)malloc(sizeof(SqList));

ListTracker[currentIndex]->elem = NULL;

}//否则则正常切换

L = ListTracker[currentIndex];

printf("Successfully changed\n");

}

else

printf("Number illegal\n");

break;

case 15:

SaveData(ListTracker);

printf("Successfully Saved\n");

getchar();

break;

case 16:

LoadData(ListTracker);

printf("Successfully Loaded\n");

L = ListTracker[0];

currentIndex = 0;

getchar();

break;

default://当输入其他字符的时候默认退出

op = 0;

break;

}

}

return 0;

}

//由于在所有操作之前均进行了检查表的操作，所以以下操作均假设输入为正常输入，不含有异常处理

status InitaList(SqList \* L)

{

//为表中的elem分配区域

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

if(!L->elem)

exit(OVERFLOW);

//同时初始化表厂为0

L->length=0;

L->listsize=LIST\_INIT\_SIZE;

return OK;

}

status DestroyList(SqList \* ListTracker[MAX\_LIST\_NUM], int currentIndex)

{

//首先free掉elem所指向的空间

free(ListTracker[currentIndex]->elem);

//再令其为NULL

ListTracker[currentIndex]->elem = NULL;

//同时free掉整个表结构

free(ListTracker[currentIndex]);

//然后将ListTracker中对应的表赋值为NULL，以表明状态

ListTracker[currentIndex] = NULL;

return OK;

}

status ClearList(SqList \* L)

{

//直接令表长为0，表明其元素个数为0，亦即元素可以被覆盖

L->length = 0;

return OK;

}

bool ListEmpty(SqList L)

{

if (L.length)

return false;

else

return true;

}

int ListLength(SqList L)

{

return L.length;

}

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

{

if (ListEmpty(L))

{

printf("The List is still empty\n");

return ERROR;

}

else if (i <= 0 || i > L.length)

{

printf("Target position doesn't exist\n");

return ERROR;

}

else

{

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

return OK;

}

}

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

{

if (!ListEmpty(L))

{

int literate\_time = 0;//迭代次数

for (; literate\_time < L.length; literate\_time ++)

{

if (compare(L.elem[literate\_time], e))

return literate\_time + 1;

}

printf("Can't find the target element\n");

return 0;

}

else

{

printf("List is empty\n");

return ERROR;

}

}

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

{

if (L.length)

{

int literate\_time = 0;//迭代次数

for (; literate\_time < L.length; literate\_time++)

{

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

{

if (literate\_time)

{ //如果找到的元素不是第一个

\* pre\_e = L.elem[literate\_time - 1];

//就将找到的元素赋值给pre\_e

return OK;

}

else//否则提示找到的元素为第一个元素，没有前驱

printf("The target element is the first element\n");

return ERROR;

}

}

printf("Can't find Target Element\n");

return ERROR;

}

else

{

printf("List is empty\n");

return ERROR;

}

}

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

{

if (L.length)

{

int literate\_time = 0;

for (; literate\_time < L.length; literate\_time++)

{

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

{

if (literate\_time != L.length - 1)

{ //如果找到的元素为最后一个元素

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

//则将找打的元素赋值给next\_e

return OK;

}

else//否则提示找到的是最后一元素，没有后继

printf("The target element is the last element\n");

return ERROR;

}

}

printf("Can't find Target Element\n");

return ERROR;

}

else

{

printf("List is empty\n");

return ERROR;

}

}

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

{

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

{

printf("Target Position illegal\n");

return ERROR;

}

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

{ //如果储存空间不够，则尝试增加空间分配

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

if (!newbase)

exit(OVERFLOW);

L->elem = newbase;

L->listsize += LISTINCREMENT;

}

ElemType \* p, \* q;

q = &(L->elem[i-1]);

//如果空间足够，就先将元素向后挪出位置

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

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

//然后将元素插入

\* q = e;

//同时更新表长

L->length++;

return OK;

}

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

{

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

return ERROR;

ElemType \* p, \* q;

p = &(L->elem[i-1]);

\*e = \*p;

q = L->elem + L->length - 1;

//直接将元素前移

for (++p; p <= q; ++p)

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

//同时修改更新表长

--L->length;

return OK;

}

status ListTrabverse(SqList L, void (\* visit)(ElemType))

{

if (L.length)

{ //迭代次数

int literate\_time = 0;

for (; literate\_time < L.length; literate\_time++)

{ //对每一个元素执行visit函数，此处visit函数的作用是打印元素

visit(L.elem[literate\_time]);

}

printf("\n");

return OK;

}

else

{

printf("List length = 0, failed to travel\n");

return ERROR;

}

}

status LoadData(SqList \* ListTracker[MAX\_LIST\_NUM])

{ //尝试打开文件

FILE \* fp = fopen("data.txt", "r");

if (fp == NULL)

{ //如果文件不存在

printf("File doesn't exist\n");

return ERROR;

}

int literate\_time = 0;

int current\_list\_num;

ElemType current\_elem;

int list\_length;

//不断读取直到文件尾，即EOF

while(literate\_time < MAX\_LIST\_NUM && fscanf(fp, "%d %d\n\n", &current\_list\_num, &list\_length) != EOF)

{ //打印log

printf("current\_list\_num = %d, list\_length = %d\n", current\_list\_num, list\_length);

printf("正在进行表%d的读取\n", current\_list\_num);

int i = 0;

//free(ListTracker[current\_list\_num]);

ListTracker[current\_list\_num] = (SqList \*)malloc(sizeof(SqList));

ListTracker[current\_list\_num]->length = list\_length;

ListTracker[current\_list\_num]->listsize = LIST\_INIT\_SIZE;

ListTracker[current\_list\_num]->elem = (ElemType \*)malloc(sizeof(ElemType) \* LIST\_INIT\_SIZE);

for (; i < list\_length; i++)

{

fscanf(fp, "%d\n", &current\_elem);

printf("正在读取第%d个元素\n", i);

(ListTracker[current\_list\_num]->elem)[i] = current\_elem;

}

literate\_time++;

}

return OK;

}

status SaveData(SqList \* ListTracker[MAX\_LIST\_NUM])

{

FILE \* fp = fopen("data.txt", "w");

//尝试打开，如果文件不存在，则创建文件

if (fp == NULL)

fp = fopen("data.txt", "wb");

int literate\_time = 0;

for (; literate\_time < MAX\_LIST\_NUM; literate\_time++)

{

if (ListTracker[literate\_time] && ListTracker[literate\_time]->length)

{ //按照一定格式将数据保存到文件中

fprintf(fp, "%d %d\n\n", literate\_time, ListTracker[literate\_time]->length);

int i = 0;

for(; i < ListTracker[literate\_time]->length; i++)

fprintf(fp, "%d\n", ListTracker[literate\_time]->elem[i]);

}

}

fclose(fp);

return OK;

}

# 附录B 基于链式存储结构线性表实现的源程序

#include <stdio.h>

#include <stdlib.h>

#define MAX\_LIST\_NUM 10

#define OK 1

#define ERROR -1 //为了与false区分开来，使用-1

#define true 1

#define false 0

#define bool int

typedef int status;

typedef int ElemType;

typedef struct List

{

ElemType elem;

struct List \* next;

}List;

status InitaList();

status DestroyList(List \* L);

status ClearList(List \*L);

bool ListEmpty(List \* L);

int ListLength(List \* L);

status GetElem(List \*L, int i, ElemType \* e);

int LocateElem(List \*L, ElemType e, bool compare(ElemType, ElemType));

status PriorElem(List \* L, ElemType cur\_e, ElemType \* pre\_e);

status NextElem(List \* L, ElemType cur\_e, ElemType \* next\_e);

status ListInsert(List \* L, int i, ElemType e);

status ListDelete(List \* L, int i, ElemType \* e);

status ListTraverse(List \* L, void visit(ElemType));

status SaveData();

status LoadData();

List ListTracker[MAX\_LIST\_NUM]; //用于存放头指针

List \* currentList = &ListTracker[0]; //用于指向当前操作的表

int currentIndex = 0; //用于标明当前表的序号

//在初始时，将所有头节点中的next指针置为NULL

void clearAllList() {

int i = 0;

for (; i < MAX\_LIST\_NUM; i++) {

ListTracker[i].next = NULL;

}

}

//为了后面的测试写的compare函数

bool compare(ElemType a , ElemType b) {

if (a == b)

return true;

else

return false;

}

//为了后面的测试写的visit函数

void visit(ElemType e) {

printf("%d ", e);

}

//打印菜单

void printMenu()

{

printf("|-------------List Experiment Menu-------------------|\n");

printf("| |\n");

printf("| |\n");

printf("| 1. Init a List 2. Destroy List |\n");

printf("| 3. Clear List 4. Empty or Not |\n");

printf("| 5. Show List Length 6. Get Element |\n");

printf("| 7. Get Prior Element 8. Get Next Element |\n");

printf("| 9. Insert Element 10. Delete Element |\n");

printf("| 11. Travel List 12. Describe List |\n");

printf("| 13. Locate Elem 14. Change List |\n");

printf("| 15. Save Data 16. Load Data |\n");

printf("| |\n");

printf("| CopyRight GS from HUST CS1703 |\n");

printf("| |\n");

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

printf(" Please Choose Your Operation from Options above \n");

}

//打印当前表的信，用于在切换表的时候供用户选择

void printListInfo() {

int i = 0;

for (; i < MAX\_LIST\_NUM; i++) {

if (ListTracker[i].next)

printf("List %d is occupied\n", i);

else

printf("List %d is available\n", i);

}

}

//用于切换表

status changeList(int target) {

if (target >= 0 && target < MAX\_LIST\_NUM) {

currentIndex = target;

if (ListTracker[currentIndex].next)

currentList = &ListTracker[currentIndex];

else

currentList = NULL;

return OK;

}

else {

printf("length illegal\n");

return ERROR;

}

}

//以下输入操作均默认输入正确，所以没有做输入的检查

int main(void) {

//先将所有表置为初始状态

clearAllList();

printMenu();

int op = 1;

while (op)

{

scanf("%d", &op);

system("cls");

printMenu();

switch (op)

{

case 1:

if (InitaList() == OK)

printf("Successfully initialized\n");

else

printf("Failed to initialize\n");

getchar();

break;

case 2://如果当前表为NULL

if (currentList) {

if (DestroyList(currentList) == OK) {

printf("Successfully destroyed\n");

}

}//提示用户需要先初始化，以下操作都需要先检查

else {

printf("You need to inite a list first\n");

}

getchar();

break;

case 3:

if (currentList) {

if (ClearList(currentList) == OK) {

printf("Successfully cleared list\n");

}

else {

printf("Failed to clear");

}

if (currentList == NULL) {

printf("fuck\n");

}

}

else {

printf("You need to inite a list first\n");

}

getchar();

break;

case 4:

if (currentList) {

if (ListEmpty(currentList))

printf("List is empty\n");

else

printf("List is not empty\n");

}

else {

printf("You need to inite list first\n");

}

getchar();

break;

case 5:

if (currentList) {

printf("List length is %d\n", ListLength(currentList));

}

else {

printf("You need to inite a list first\n");

}

getchar();

break;

case 6:

if (currentList) {

if (ListEmpty(currentList))

printf("List is still empty\n");

else {

ElemType elem\_get;

int position\_get;

printf("Please enter the position you want to get (from 1 to %d)", ListLength(currentList));

scanf("%d", &position\_get);

if (GetElem(currentList, position\_get, &elem\_get) == OK)

printf("Element of position %d is %d", position\_get, elem\_get);

else

printf("Failed to get Element\n");

}

}

else {

printf("You need to inite a list first\n");

}

getchar();

break;

case 7:

if (currentList) {

printf("Please enter the elem you want to query:\n");

ElemType elem\_prior;

ElemType input\_prior;

//用户输入元素

scanf("%d", &input\_prior);

if (PriorElem(currentList, input\_prior, &elem\_prior) != ERROR)

printf("The prior element of %d is %d\n", input\_prior, elem\_prior);

else

printf("failed to find\n");

}

else {

printf("You need to inite a list first\n");

}

getchar();

break;

case 8:

if (currentList) {

printf("Please enter the elem you want to query:\n");

ElemType elem\_next;

ElemType input\_next;

scanf("%d", &input\_next);

if (NextElem(currentList, input\_next, &elem\_next) != ERROR)

printf("The next element of %d is %d\n", input\_next, elem\_next);

else

printf("failed to find\n");

}

else {

printf("You need to inite a list first\n");

}

getchar();

break;

case 9:

if (currentList) {

printf("Position: (between 1 to %d)\n", ListLength(currentList) + 1);

printf("Please enter the element you want to insert, and the position of it(spaced by space):\n");

ElemType insertElem;

int insertPosition;

//用户输入

scanf("%d %d", &insertElem, &insertPosition);

if (ListInsert(currentList, insertPosition, insertElem) != ERROR)

printf("Successfully inserted\n");

else

printf("failed to insert\n");

}

else

printf("You need to init a list first\n");

getchar();

break;

case 10:

if (currentList) {

printf("Please enter the position of the element you want to delete(between 1 to %d):\n", ListLength(currentList));

ElemType deleteElem;

int deletePosition;

scanf("%d", &deletePosition);

if (ListDelete(currentList, deletePosition, &deleteElem) != ERROR)

printf("Successfully deleted %d in position %d\n", deleteElem, deletePosition);

else

printf("failed to deleted\n");

}

getchar();

break;

case 11:

if (currentList)

{

if (ListTraverse(currentList, visit) == OK)

printf("\nsuccessfully traveled all elements\n");

else

printf("\nList is still empty, failed to travel all element\n");

}

else {

printf("You need to init a list first\n");

}

getchar();

break;

case 12:

if (currentList) {

printf("List length = %d, currentIndex = %d\n", ListLength(currentList), currentIndex);

}

else {

printf("current index = %d, not inite\n", currentIndex);

}

getchar();

break;

case 13:

if (currentList)

{

printf("Please enter the elem you want to locate with function compare\n");

ElemType queryElem\_locate;

int queryResult\_locate;

scanf("%d", &queryElem\_locate);

if (queryResult\_locate = LocateElem(currentList, queryElem\_locate, compare))

{

printf("The position of %d is %d\n", queryElem\_locate, queryResult\_locate);

}

else

printf("Failed to find\n");

}

else {

printf("You need to inite a list first\n");

}

getchar();

break;

case 14:

printf("You have chose to change list\n");

printListInfo();

printf("Please enter the index you want to change to\n");

int tempIndex;

scanf("%d", &tempIndex);

if (changeList(tempIndex) == OK)

printf("Successfully changed, current index is %d\n", currentIndex);

else

printf("Failed to change list");

getchar();

break;

case 15:

SaveData();

printf("Successfully Saved\n");

getchar();

break;

case 16:

LoadData();

printf("Successfully Loaded\n");

getchar();

break;

default://当输入其他字符的时候默认退出

op = 0;

break;

}

}

return 0;

}

status InitaList() {

if (currentList) {

printf("Can't repeatedly initialize the same list\n");

return ERROR;

}

else {

currentList = &ListTracker[currentIndex];

return OK;

}

}

//destroy 之后需要手动将currentList 置空

status DestroyList(List \* L) {

ClearList(L);

currentList = NULL;

return OK;

}

status ClearList(List \*L) {

if (L->next == NULL) {

return OK;

}

else {

ClearList(L->next);

free(&L->next->elem);

L->next = NULL;

return OK;

}

}

bool ListEmpty(List \* L) {

if (L->next)

return false;

else

return true;

}

int ListLength(List \* L) {

int length = 0;

while (L->next) {

length++;

L = L->next;

}

return length;

}

status GetElem(List \*L, int i, ElemType \* e) {

if (ListLength(L) >= i && i > 0) {

while (i) {

L = L->next;

i--;

}

\*e = L->elem;

return OK;

}

else {

printf("Length illigal\n");

return ERROR;

}

}

int LocateElem(List \*L, ElemType e, bool compare(ElemType, ElemType)) {

int position = 0;

L = L->next;

while (L) {

position += 1;

if (compare(e, L->elem)) {

return position;

}

L = L->next;

}

return 0;

}

status PriorElem(List \* L, ElemType cur\_e, ElemType \* pre\_e) {

//如果查询到的是第一个数据

if (L->next->elem == cur\_e) {

//提示

printf("The target element is the first element\n");

return ERROR;

}

while (L->next) {

if (L->next->elem == cur\_e) {

\*pre\_e = L->elem;

return OK;

}

L = L->next;

}

return ERROR;

}

status NextElem(List \* L, ElemType cur\_e, ElemType \* next\_e) {

while (L->next) {

if (L->elem == cur\_e) {

\*next\_e = L->next->elem;

return OK;

}

L = L->next;

}

//如果查询到的是最后一个数据

if (L->elem == cur\_e)//提示

printf("The target element is the last element\n");

return ERROR;

}

status ListInsert(List \* L, int i, ElemType e) {

if (i <= ListLength(L) + 1 && i >= 1) {

if (i == 1) {

List \* temp = L->next;

ListTracker[currentIndex].next = (List \*)malloc(sizeof(List));

currentList = &ListTracker[currentIndex];

currentList->next->next = temp;

currentList->next->elem = e;

}

else {

List \* TempPointer = L;

while (i-1) {

TempPointer = TempPointer->next;

i--;

}

List \* temp = TempPointer->next;

TempPointer->next = (List \*)malloc(sizeof(List));

TempPointer->next->elem = e;

TempPointer->next->next = temp;

}

return OK;

}

else {

printf("Position illigal\n");

return ERROR;

}

}

status ListDelete(List \* L, int i, ElemType \* e) {

if (ListLength(L) >= i && i >= 1) {

if (i == 1) {

List \*temp = L->next;

\*e = L->next->elem;

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

free(temp);

}

else {

List \* TempPointer = L;

while (i-1) {

TempPointer = TempPointer->next;

i--;

}

List \* temp = TempPointer->next;

\*e = TempPointer->next->elem;

TempPointer->next = TempPointer->next->next;

free(temp);

}

return OK;

}

else {

printf("Position illigal\n");

return ERROR;

}

}

status ListTraverse(List \* L, void visit(ElemType)) {

if (!L->next)

return ERROR;

while (L->next) {

visit(L->next->elem);

L = L->next;

}

return OK;

}

status SaveData() {

FILE \* fp = fopen("data.txt", "w");

if (fp == NULL) {

fp = fopen("data.txt", "wb");

}

int literateTime = 0;

for (; literateTime < MAX\_LIST\_NUM; literateTime++) {

List \* temp = &ListTracker[literateTime];

if (!ListEmpty(temp)) {

printf("List should write %d %d", literateTime, ListLength(temp));

//储存表的序号以及长度

fprintf(fp, "%d %d\n\n", literateTime, ListLength(temp));

while (temp->next) {

//储存具体的数据

printf("%d List is writing %d\n", literateTime, temp->next->elem);

fprintf(fp, "%d\n", temp->next->elem);

temp = temp->next;

}

}

}

fclose(fp);

return OK;

}

status LoadData() {

FILE \* fp = fopen("data.txt", "r");

if (fp == NULL) {

printf("File doesn't exist\n");

return ERROR;

}

else {

int current\_list\_num;

int list\_length;

ElemType current\_elem;

while (fscanf(fp, "%d %d\n\n", &current\_list\_num, &list\_length) != EOF) {

printf("Loading the %dth list\n", current\_list\_num);

printf("current list num = %d, list length = %d\n", current\_list\_num, list\_length);

fscanf(fp, "%d\n", &current\_elem);

ListTracker[current\_list\_num].next = (List \*)malloc(sizeof(List));

List \* temp = &ListTracker[current\_list\_num];

temp->next->elem = current\_elem;

temp->next->next = NULL;

temp = temp->next;

while (list\_length-1) {

temp->next = (List \*)malloc(sizeof(List));

temp = temp->next;

fscanf(fp, "%d\n", &current\_elem);

temp->elem = current\_elem;

temp->next = NULL;

list\_length--;

}

}

}

fclose(fp);

return OK;

}

# 附录C 基于二叉链表二叉树实现的源程序

#include <stdio.h>

#include <iostream>

#include <stdlib.h>

#define MAX\_TREE\_NUM 10

#define OK 1

#define ERROR -1

#define true 1

#define false 0

#define bool int

typedef int status;

typedef int ElemType;

using namespace std;

struct TreeNode

{

ElemType elem;

TreeNode \* LChild;

TreeNode \* RChild;

};

struct Quene

{

TreeNode \* elem;

Quene \* next;

};

struct HeadNode

{

TreeNode \* Tree;

};

class myQuene {

public:

Quene \* ListHead = nullptr;

int listlength = 0;

void push(TreeNode \* tarelem) {

if (ListHead) {

Quene \* temp = ListHead;

while (temp->next)

temp = temp->next;

temp->next = (Quene \*)malloc(sizeof(Quene));

temp->next->elem = tarelem;

temp->next->next = NULL;

}

else {

ListHead = (Quene \*)malloc(sizeof(Quene));

ListHead->elem = tarelem;

ListHead->next = nullptr;

}

listlength++;

}

TreeNode \* pop() {

Quene \* temp = ListHead;

TreeNode \* tempValue = temp->elem;

ListHead = temp->next;

free(temp);

listlength--;

return tempValue;

}

void clearAll() {

ListHead = nullptr;

}

};

status InitBiTree();

status ClearBiTree(HeadNode \* currentHead);

void ClearFromRoot(TreeNode \* root);

bool BiTreeEmpty(HeadNode \* currentHead);

status CreateBiTree(TreeNode \* & currentPtr);

status DestroyBiTree(HeadNode \* currentHead);

TreeNode \* Root();

TreeNode \* getElem();

TreeNode \* Parent(TreeNode \* root, TreeNode \* e);

TreeNode \* LeftChild(TreeNode \* e);

TreeNode \* RightChild(TreeNode \* e);

int depth(TreeNode \* currentHead);

status PreOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType));

status InOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType));

status PostOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType));

status LevelOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType));

status DeleteChild(TreeNode \* p, int LR);

TreeNode \* LeftSibling(TreeNode \* root, TreeNode \* e);

TreeNode \* RightSibling(TreeNode \* root, TreeNode \* e);

status InsertChild(TreeNode \* p, int LR, TreeNode \* targetTree);

void iterSave(FILE \* fp, TreeNode \* ptr);

void SaveData();

status iterLoad(FILE \* fp, TreeNode \* & currentPtr);

status LoadData();

void printTree(TreeNode \* currentTree, int h);

HeadNode \* currentHead; //用于存放当指向当前树的头节点的指针

HeadNode \* TreeTracker[MAX\_TREE\_NUM]; //头指针数组

int currentIndex; //指示当前树的序号

myQuene quene;

myQuene quene\_for\_select;

int getElemConst;

int getElemTarget;

TreeNode \* tempTarget;

void clearTracker() {

int i = 0;

for (; i < MAX\_TREE\_NUM; i++)

TreeTracker[i] = NULL;

}

void Visit(ElemType elem) {

cout << elem << " ";

}

void printListInfo() {

int i = 0;

for (; i < MAX\_TREE\_NUM; i++) {

if (TreeTracker[i])

cout << "树 " << i << " 已被占用" << endl;

else

cout << "树 " << i << "可用" << endl;

}

}

void printMenu(void) {

cout << "|--------------------B Tree Experiment Menu--------------------|" << endl;

cout << "| 1. 初始化当前树 2. 销毁当前树 |" << endl;

cout << "| 3. 清空当前树 4. 判断当前树是否为空 |" << endl;

cout << "| 5. 利用前序创建一颗树 6. 获取根节点 |" << endl;

cout << "| 7. 获取一个节点 8. 查询当前树深度 |" << endl;

cout << "| 9. 赋值 10. 获取父节点 |" << endl;

cout << "| 11. 获取左孩子 12. 获取右孩子 |" << endl;

cout << "| 13. 获取左邻元素 14. 获取右邻节点 |" << endl;

cout << "| 15. 插入一颗树 16. 删除子树 |" << endl;

cout << "| 17. 前序遍历 18. 中序遍历 |" << endl;

cout << "| 19. 后序遍历 20. 按层遍历 |" << endl;

cout << "| 21. 切换树 22. 保存数据 |" << endl;

cout << "| 23. 导入数据 24. 打印树（横向） |" << endl;

}

void main(void) {

//先清空所有空间

clearTracker();

printMenu();

int op = 1;

while (op) {

cin >> op;

system("cls");

printMenu();

switch (op) {

case 1:

if (InitBiTree() == OK) {

cout << "初始化成功" << endl;

}

else {

cout << "初始化失败" << endl;

}

break;

case 2:

if (DestroyBiTree(currentHead) == OK) {

cout << "成功销毁" << endl;

}

else {

cout << "销毁失败" << endl;

}

break;

case 3:

if (ClearBiTree(currentHead) == OK) {

cout << "清空成功" << endl;

}

else {

cout << "清空失败" << endl;

}

break;

case 4:

if (BiTreeEmpty(currentHead) == ERROR) {

cout << "判断失败" << endl;

}

else {

if (BiTreeEmpty(currentHead))

cout << "当前树为空" << endl;

else

cout << "当前树不为空" << endl;

}

break;

case 5:

if (currentHead) {

cout << "请输入先序遍历的数字，其中空节点用#表示, 使其成为一个二叉树" << endl;

cout << "请输入正确的遍历结果，否则输入可能会一直进行" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 3" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例序列为 1 2 # # 3 4 # # #" << endl;

if (CreateBiTree(currentHead->Tree) == OK) {

cout << "成功创建" << endl;

}

else

cout << "创建失败" << endl;

}

else {

cout << "请先初始化" << endl;

}

break;

case 6:

if (currentHead && currentHead->Tree) {

if (Root()) {

cout << "根节点的值为 " << Root()->elem << endl;

}

else

cout << "获取失败" << endl;

}

else {

cout << "请先选择一棵树或者初始化当前树" << endl;

}

break;

case 7:

cout << "请输入目标节点在按层遍历过程中的次序并选择删除左子树或右子树" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

break;

case 8:

if (currentHead)

cout << "当前树的深度为 " << depth(currentHead->Tree) << endl;

else if (!currentHead)

cout << "当前树已被销毁，若要继续操作请重新选择一颗树" << endl;

break;

case 9:

cout << "请输入目标节点在按层遍历过程中的次序" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

cout << "请输入要为其赋的新值" << endl;

if (cin >> tempTarget->elem) {

cout << "成功将其赋值为" << tempTarget->elem << endl;

}

else

cout << "赋值失败" << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

break;

case 10:

if (currentHead->Tree) {

cout << "请输入目标节点在按层遍历过程中的次序" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

if (Parent(currentHead->Tree, tempTarget)) {

cout << "其父元素为" << Parent(currentHead->Tree, tempTarget)->elem << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

}

else

cout << "请先初始化" << endl;

break;

case 11:

if (currentHead->Tree) {

cout << "请输入目标节点在按层遍历过程中的次序" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

if (LeftChild(tempTarget)) {

cout << "其左子元素为" << LeftChild(tempTarget)->elem << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

}

else

cout << "请先初始化" << endl;

break;

case 12:

if (currentHead->Tree) {

cout << "请输入目标节点在按层遍历过程中的次序" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

if (RightChild(tempTarget)) {

cout << "其右子元素为" << RightChild(tempTarget)->elem << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

}

else

cout << "请先初始化" << endl;

break;

case 13:

if (currentHead->Tree) {

cout << "请输入目标节点在按层遍历过程中的次序" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

if (LeftSibling(currentHead->Tree, tempTarget)) {

cout << "其左临元素为" << LeftSibling(currentHead->Tree, tempTarget)->elem << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

}

else

cout << "请先初始化" << endl;

break;

case 14:

if (currentHead->Tree) {

cout << "请输入目标节点在按层遍历过程中的次序" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

if (RightSibling(currentHead->Tree, tempTarget)) {

cout << "其右临元素为" << RightSibling(currentHead->Tree, tempTarget)->elem << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

}

else

cout << "请先初始化" << endl;

break;

case 15:

cout << "请选择要插入当前树的的目标:" << endl;

printListInfo();

int op\_index;

cin >> op\_index;

getchar();

if ((op\_index < 0) || (op\_index > MAX\_TREE\_NUM) || (TreeTracker[op\_index] == NULL))

{

cout << "编号" << op\_index << "不能使用" << endl;

break;

}

cout << "请输入插入的目标节点在当前树中按层遍历过程中的次序" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

cout << "你选择的元素为" << tempTarget->elem << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

cout << "请选择想要插在左边还是右边：" << endl;

int choice;

cin >> choice;

getchar();

if (InsertChild(tempTarget, choice, TreeTracker[op\_index]->Tree) == OK)

cout << "插入成功" << endl;

break;

case 16:

cout << "请输入目标节点在按层遍历过程中的次序并选择删除左子树或右子树" << endl;

cout << " 1 " << endl;

cout << " / \\ " << endl;

cout << "2 5" << endl;

cout << " /" << endl;

cout << " 4" << endl;

cout << "样例中若要取元素5所在位置的元素则输入3" << endl;

getElemConst = 0;

quene\_for\_select.push(currentHead->Tree);

if (cin >> getElemTarget) {

if (tempTarget = getElem()) {

int choice;

cin >> choice;

if (DeleteChild(tempTarget, choice) != ERROR)

cout << "删除成功" << endl;

else

cout << "删除失败" << endl;

}

else

cout << "查找失败" << endl;

}

else

cout << "请输入正确的元素位置" << endl;

quene\_for\_select.clearAll();

break;

case 17:

if (currentHead) {

if (PreOrderTraverse(currentHead->Tree, Visit) == ERROR)

cout << "遍历失败" << endl;

else

cout << "\n遍历成功" << endl;

}

else {

cout << "当前树已被销毁，请重新选择一棵树" << endl;

}

break;

case 18:

if (currentHead) {

if (InOrderTraverse(currentHead->Tree, Visit) == ERROR)

cout << "遍历失败" << endl;

else

cout << "\n遍历成功" << endl;

}

else {

cout << "当前树已被销毁，请重新选择一棵树" << endl;

}

break;

case 19:

if (currentHead) {

if (PostOrderTraverse(currentHead->Tree, Visit) == ERROR)

cout << "遍历失败" << endl;

else

cout << "\n遍历成功" << endl;

}

else {

cout << "当前树已被销毁，请重新选择一棵树" << endl;

}

break;

case 20:

if (currentHead) {

if (LevelOrderTraverse(currentHead->Tree, Visit) == ERROR)

cout << "遍历失败" << endl;

else

cout << "\n遍历成功" << endl;

}

else {

cout << "当前树已被销毁，请重新选择一棵树" << endl;

}

break;

case 21:

printListInfo();

cout << "请输入目标树的序号" << endl;

cin >> currentIndex;

currentHead = TreeTracker[currentIndex];

cout << "切换成功，当前树的序号为" << currentIndex << endl;

break;

case 22:

SaveData();

break;

case 23:

LoadData();

cout << "导入成功" << endl;

break;

case 24:

printTree(currentHead->Tree, depth(currentHead->Tree));

break;

}

}

}

status InitBiTree() {

if (TreeTracker[currentIndex]) {

//如果当前树已经被初始化，那么提示错误

cout << "不能重复初始化同一棵树" << endl;

return ERROR;

}

else {

//否则初始化头节点，并处理头节点中的相关数据

TreeTracker[currentIndex] = (HeadNode \*)malloc(sizeof(HeadNode));

TreeTracker[currentIndex]->Tree = NULL;

currentHead = TreeTracker[currentIndex];

return OK;

}

}

status DestroyBiTree(HeadNode \* currentHead) {

if (!currentHead) {

cout << "请先初始化" << endl;

return ERROR;

}

if (ClearBiTree(currentHead) == ERROR){

return ERROR;

}

free(currentHead);

TreeTracker[currentIndex] = NULL;

currentHead = NULL;

return OK;

}

status ClearBiTree(HeadNode \* currentHead) {

if (!currentHead) {

cout << "请先初始化" << endl;

return ERROR;

}

else if (BiTreeEmpty(currentHead)) {

cout << "树已空" << endl;

return ERROR;

}

else {

ClearFromRoot(currentHead->Tree);

free(currentHead->Tree);

currentHead->Tree = NULL;

return OK;

}

}

void ClearFromRoot(TreeNode \* root) {

if (root->LChild) {

ClearFromRoot(root->LChild);

}

if (root->RChild) {

ClearFromRoot(root->RChild);

}

free(root->RChild);

free(root->LChild);

root->RChild = NULL;

root->LChild = NULL;

}

status CreateBiTree(TreeNode \* & currentPtr) {

int temp;

int returnValue = scanf("%d", &temp);

if (returnValue > 0) {

currentPtr = (TreeNode \*)malloc(sizeof(TreeNode));

currentPtr->elem = temp;

if (CreateBiTree(currentPtr->LChild) == ERROR)

return ERROR;

if (CreateBiTree(currentPtr->RChild) == ERROR)

return ERROR;

return OK;

}

else if (returnValue == 0){

currentPtr = nullptr;

getchar();

return OK;

}

else if (returnValue == -1) {

currentPtr = nullptr;

return OK;

}

}

bool BiTreeEmpty(HeadNode \* currentHead) {

if (!currentHead) {

cout << "请先初始化" << endl;

return ERROR;

}

else {

if (depth(currentHead->Tree))

return false;

else

return true;

}

}

int depth(TreeNode \* currentHead) {

if (currentHead == NULL)

return 0;

else

{

int LeftDepth = depth(currentHead->LChild);

int RightDepth = depth(currentHead->RChild);

if (LeftDepth > RightDepth)

return LeftDepth + 1;

else

return RightDepth + 1;

}

}

TreeNode \* Root() {

return currentHead->Tree;

}

ElemType Value(TreeNode \* e) {

return e->elem;

}

status Assign(TreeNode \* e, ElemType value) {

e->elem = value;

return OK;

}

//如果找到了就返回对应节点的父节点，如果没有找到，返回NULL

TreeNode \* Parent(TreeNode \* root, TreeNode \* e) {

if (root->LChild == e)

return root;

else if (root->RChild == e)

return root;

else {

if (root->LChild) {

TreeNode \* temp = Parent(root->LChild, e);

//如果在左子树没有找到，就在右子树查找

if (temp) {

return temp;

}

else {

//不论temp是否为NULL，都进行返回

temp = Parent(root->RChild, e);

return temp;

}

}

}

}

TreeNode \* LeftChild(TreeNode \* e) {

return e->LChild;

}

TreeNode \* RightChild(TreeNode \* e) {

return e->RChild;

}

TreeNode \* LeftSibling(TreeNode \* root, TreeNode \* e) {

TreeNode \* temp = Parent(root, e);

if (temp && temp->RChild == e)

return temp->LChild;

return NULL;

}

TreeNode \* RightSibling(TreeNode \* root, TreeNode \* e) {

TreeNode \* temp = Parent(root, e);

if (temp && temp->LChild == e) {

return temp->RChild;

}

return NULL;

}

status InsertChild(TreeNode \* p, int LR, TreeNode \* targetTree) {

if (targetTree->RChild) {

cout << "The right subtree is not empty" << endl;

return ERROR;

}

else {

if (LR) {

targetTree->RChild = p->RChild;

p->RChild = targetTree;

}

else {

targetTree->RChild = p->LChild;

p->LChild = targetTree;

}

return OK;

}

}

status DeleteChild(TreeNode \* p, int LR) {

if (p) {

if (LR) {

if (p->RChild) {

ClearFromRoot(p->RChild);

free(p->RChild);

p->RChild = nullptr;

}

}

else {

if (p->LChild) {

ClearFromRoot(p->LChild);

free(p->LChild);

p->LChild = nullptr;

}

}

}

else {

return ERROR;

}

return OK;

}

status PreOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType)) {

if (root == NULL)

return ERROR;

Visit(root->elem);

PreOrderTraverse(root->LChild, Visit);

PreOrderTraverse(root->RChild, Visit);

return OK;

}

status InOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType)) {

if (root == NULL)

return ERROR;

InOrderTraverse(root->LChild, Visit);

Visit(root->elem);

InOrderTraverse(root->RChild, Visit);

return OK;

}

status PostOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType)) {

if (root == NULL)

return ERROR;

PostOrderTraverse(root->LChild, Visit);

PostOrderTraverse(root->RChild, Visit);

Visit(root->elem);

return OK;

}

status LevelOrderTraverse(TreeNode \* root, void(\*Visit)(ElemType)) {

quene.push(root);

TreeNode \* cur;

while (quene.listlength > 0) {

cur = quene.pop();

Visit(cur->elem);

if (cur->LChild)

quene.push(cur->LChild);

if (cur->RChild)

quene.push(cur->RChild);

}

return OK;

}

void printTree(TreeNode \* currentTree, int h) {

int i;

if (currentTree != NULL) {

printTree(currentTree->RChild, h+1);

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

putchar('/t');

cout << currentTree->elem << endl;

putchar('\n');

printTree(currentTree->LChild, h+1);

}

}

TreeNode \* getElem() {

while (quene\_for\_select.listlength) {

if (getElemConst == getElemTarget - 1) {

return quene\_for\_select.pop();

}

TreeNode \* temp = quene\_for\_select.pop();

if (!temp) {

return nullptr;

}

else {

if (temp->LChild) {

quene\_for\_select.push(temp->LChild);

}

if (temp->RChild) {

quene\_for\_select.push(temp->RChild);

}

}

getElemConst++;

}

}

void SaveData() {

FILE \* fp = fopen("data.txt", "w");

if (fp == NULL) {

fp = fopen("data.txt", "wb");

}

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

if (TreeTracker[i] && TreeTracker[i]->Tree) {

fprintf(fp, "%d\n", i);

iterSave(fp, TreeTracker[i]->Tree);

}

fprintf(fp, "/n");

}

fclose(fp);

}

void iterSave(FILE \* fp, TreeNode \* ptr) {

if (ptr) {

fprintf(fp, "%d ", ptr->elem);

iterSave(fp, ptr->LChild);

iterSave(fp, ptr->RChild);

}

else {

fprintf(fp, "# ");

}

}

status LoadData() {

FILE \* fp = fopen("data.txt", "r");

if (fp == nullptr) {

cout << "文件不存在" << endl;

return ERROR;

}

else {

int temp;

while (fscanf(fp, "%d\n", &temp) != EOF) {

TreeTracker[temp] = (HeadNode \*)malloc(sizeof(HeadNode));

iterLoad(fp, TreeTracker[temp]->Tree);

}

}

fclose(fp);

}

status iterLoad(FILE \* fp, TreeNode \* & currentPtr) {

ElemType temp;

int returnValue = fscanf(fp, "%d ", &temp);

if (returnValue > 0) {

currentPtr = (TreeNode \*)malloc(sizeof(TreeNode));

currentPtr->elem = temp;

if (iterLoad(fp, currentPtr->LChild) == ERROR)

return ERROR;

if (iterLoad(fp, currentPtr->RChild) == ERROR)

return ERROR;

return OK;

}

else if (returnValue == 0) {

currentPtr = nullptr;

fgetc(fp);

return OK;

}

else if (returnValue == -1) {

currentPtr = nullptr;

return OK;

}

}

# 附录D 基于邻接表图实现的源程序

//无向图

//C++

#include<queue>

#include <iostream>

#include <string>

#include <vector>

#include <fstream>

#define OK 1

#define ERROR -1

using namespace std;

typedef int status;

typedef int ElemType;

//创建的模板，用于查找vector中目标元素的位置

template <typename T>

int findElem(T target, string name) {

for (int i = target.size() - 1; i >= 0; i--) {

if (target[i].name == name) {

return i;

}

}

return -1;

}

class Vex {

public:

ElemType data;

string name;

bool visited;

Vex \* nextVex;

//构造函数

Vex(string name, ElemType data) {

this->data = data;

this->name = name;

this->visited = false;

this->nextVex = nullptr;

}

void DestroyVex(Vex \* v) {

if (v != nullptr) {

DestroyVex((\*v).nextVex);

delete v;

}

}

void Destroy() {

DestroyVex(nextVex);

}

};

class Graph {

public:

vector <Vex> List;

string name;

Graph(string graphname) {

name = graphname;

}

/\*New Destroy with built-in iterator for vector\*/

void Destroyself() {

for (auto v : List) {

v.Destroy();

}

List.clear();

}

void AddVex(string vexname, ElemType vexvalue) {

Vex v(vexname, vexvalue);

List.push\_back(v);

}

};

class Index {

public:

int currentGraphIndex;

vector <Graph> List;

status CreateGraph(string graphname);

status DestroyGraph(string graphname);

Vex & LocateVex(string graphname, string vexname);

ElemType GetVex(string graphname, string vexname);

status PutVex(string graphname, string vexname, ElemType value);

Vex \* FirstAdjVex(string graphname, string vexname);

Vex \* NextAdjVex(string graphname, string vexname, string vexname2);

status InsertVex(string graphname, string vexname, ElemType data);

status DeleteVex(string graphname, string vexname);

status InsertArc(string graphname, string vexname1, string vexname2);

status DeleteArc(string graphname, string vexname1, string vexname2);

status DFSTraverse(string graphname, void (\*visit)(Vex));

status BFSTraverse(string graphname, void (\*visit)(Vex));

void SaveData();

void LoadData();

void PrintTable();

int GraphExist(string graphname) {

int gposition = findElem(List, graphname);

if (gposition >= 0) {

return gposition;

}

else {

return -1;

}

}

bool VexExist(string graphname, string vexname) {

int gposition = GraphExist(graphname);

if (gposition >= 0) {

int vposition = findElem(List[gposition].List, vexname);

if (vposition >= 0)

return true;

else

return false;

}

else

return false;

}

void Clearvisited(string graphname) {

int gposition = findElem(List, graphname);

/\*new clear with built-in iterator with vector\*/

for (auto i : List[gposition].List) {

i.visited = false;

}

}

private:

void DFS(Graph & G, int position, void(\*visit)(Vex), string graphname);

};

status Index::CreateGraph(string graphname) {

Graph G(graphname);

List.push\_back(G);

cout << "请输入将要输入的节点数目" << endl;

//一些临时变量

int vexnum;

int arcnum;

string vexname;

string vexname2;

int vexvalue;

cin >> vexnum;

cout << "请依次输入" << vexnum << "个节点的名字以及每个节点的值" << endl;

for (int i = 0; i < vexnum; i++) {

cin >> vexname;

cin >> vexvalue;

InsertVex(graphname, vexname, vexvalue);

}

cout << "请输入要输入边的数目" << endl;

cin >> arcnum;

cout << "请依次输入" << arcnum << "组边的两个节点名字" << endl;

for (int i = 0; i < arcnum; i++) {

cin >> vexname;

cin >> vexname2;

InsertArc(graphname, vexname, vexname2);

}

return OK;

}

status Index::DestroyGraph(string graphname) {

int position = findElem(List, graphname);

if (position >= 0) {

List[position].Destroyself();

vector<Graph>::iterator iter = List.begin() + position;

List.erase(iter);

return OK;

}

else {

return ERROR;

}

}

Vex & Index::LocateVex(string graphname, string vexname) {

int gposition = findElem(List, graphname);

int vposition = findElem(List[gposition].List, vexname);

return List[gposition].List[vposition];

}

ElemType Index::GetVex(string graphname, string vexname) {

return LocateVex(graphname, vexname).data;

}

status Index::PutVex(string graphname, string vexname, ElemType value) {

Vex &target = LocateVex(graphname, vexname);

Vex \* temp = &target;

target.data = value;

while (temp->nextVex) {

//找到邻接节点

Vex tempV = LocateVex(graphname, temp->nextVex->name);

while (tempV.nextVex->name != target.name)

tempV = \*tempV.nextVex;

//找到后，更新值

tempV.nextVex->data = value;

temp = temp->nextVex;

}

return OK;

}

Vex \* Index::FirstAdjVex(string graphname, string vexname) {

int gposition = findElem(List, graphname);

int vposition = findElem(List[gposition].List, vexname);

Vex temp = List[gposition].List[vposition];

if (temp.nextVex) {

return temp.nextVex;

}

else {

return nullptr;

}

}

Vex \* Index::NextAdjVex(string graphname, string vexname, string vexname2) {

int gposition = findElem(List, graphname);

int vposition = findElem(List[gposition].List, vexname);

Vex temp = List[gposition].List[vposition];

while (temp.nextVex) {

if (temp.nextVex->name == vexname2 && temp.nextVex->nextVex) {

return temp.nextVex->nextVex;

}

temp = \*temp.nextVex;

}

return nullptr;

}

status Index::InsertVex(string graphname, string vexname, ElemType data) {

int position = findElem(List, graphname);

List[position].AddVex(vexname, data);

return OK;

}

status Index::DeleteVex(string graphname, string vexname) {

int gposition = findElem(List, graphname);

int vposition = findElem(List[gposition].List, vexname);

Vex & target = LocateVex(graphname, vexname);

Vex \* temp = &target;

while (temp->nextVex) {

//找到邻接节点

Vex \*tempV = &(LocateVex(graphname, temp->nextVex->name));

while (tempV->nextVex->name != target.name)

tempV = tempV->nextVex;

//找到后，创建临时指针用于删除点

Vex \* deleteTemp = tempV->nextVex;

//在邻接点的邻接链表中去掉目标节点

tempV->nextVex = tempV->nextVex->nextVex;

//删除点

delete deleteTemp;

temp = temp->nextVex;

}

target.Destroy();

vector<Vex>::iterator iter = List[gposition].List.begin() + vposition;

List[gposition].List.erase(iter);

return OK;

}

status Index::InsertArc(string graphname, string vexname1, string vexname2) {

Vex &v1 = LocateVex(graphname, vexname1);

Vex &v2 = LocateVex(graphname, vexname2);

Vex \* tempV1 = &v1;

Vex \* tempV2 = &v2;

while (tempV1->nextVex) {

if (tempV1->nextVex->name == v2.name)

return ERROR;

tempV1 = tempV1->nextVex;

}

while (tempV2->nextVex) {

if (tempV2->nextVex->name == v1.name)

return ERROR;

tempV2 = tempV2->nextVex;

}

tempV1->nextVex = new Vex(v2.name, v2.data);

tempV2->nextVex = new Vex(v1.name, v1.data);

return OK;

}

status Index::DeleteArc(string graphname, string vexname1, string vexname2) {

Vex &v1 = LocateVex(graphname, vexname1);

Vex &v2 = LocateVex(graphname, vexname2);

Vex \* tempV1 = &v1;

Vex \* tempV2 = &v2;

while (tempV1) {

if (tempV1->nextVex->name == v2.name)

break;

else

tempV1 = tempV1->nextVex;

}

while (tempV2) {

if (tempV2->nextVex->name == v1.name)

break;

else

tempV2 = tempV2->nextVex;

}

if (!(tempV1 && tempV2)) {

return ERROR;

}

else {

Vex \* deleteTemp1 = tempV1->nextVex;

Vex \* deleteTemp2 = tempV2->nextVex;

tempV1->nextVex = tempV1->nextVex->nextVex;

tempV2->nextVex = tempV2->nextVex->nextVex;

delete deleteTemp1;

delete deleteTemp2;

return OK;

}

}

void Index::DFS(Graph & G, int position, void (\*visit)(Vex), string graphname) {

G.List[position].visited = true;

visit(G.List[position]);

for (Vex \* w = G.List[position].nextVex; w != nullptr; w = w->nextVex) {

if (!LocateVex(graphname, w->name).visited) {

DFS(G, findElem(G.List, w->name), visit, graphname);

}

}

}

status Index::DFSTraverse(string graphname, void (\*visit)(Vex)) {

int gposition = findElem(List, graphname);

Graph & G = List[gposition];

for (int i = 0; i < G.List.size(); i++) {

if (!G.List[i].visited)

DFS(G, i, visit, G.name);

}

return OK;

}

status Index::BFSTraverse(string graphname, void(\*visit)(Vex)) {

queue <string> Queue;

int gposition = findElem(List, graphname);

Graph & G = List[gposition];

for (auto i : G.List) {

if (!i.visited) {

Queue.push(i.name);

while (Queue.size() != 0) {

string vname = Queue.front();

if (!G.List[findElem(G.List, vname)].visited) {

visit(LocateVex(graphname, vname));

G.List[findElem(G.List, vname)].visited = true;

}

Queue.pop();

for (Vex \* w = LocateVex(graphname, vname).nextVex; w != nullptr; w = w->nextVex) {

if (!LocateVex(graphname, w->name).visited)

Queue.push(w->name);

}

}

}

}

return OK;

}

//用于打印邻接表

void Index::PrintTable() {

for (auto i : List) {

cout << "图: " << i.name << endl;

cout << endl;

for (auto j : i.List) {

for (Vex \* temp = &j; temp != nullptr; temp = temp->nextVex) {

cout << " -> " << "(" << temp->name << ", " << temp->data << ")";

}

cout << endl;

cout << endl;

}

}

}

void Index::SaveData() {

ofstream outfile;

outfile.open("data.txt", ios::out);

//图的数量

outfile << List.size() << endl;

int arcnum = 0;

for (auto i : List) {

//图的名字

outfile << i.name << endl;

//图中节点的个数

outfile << " " << i.List.size() << endl;

for (auto j : i.List) {

//节点的名字以及节点的值

outfile << j.name << " " << j.data << endl;

}

for (auto j : i.List) {

//边的信息

for (Vex \* w = j.nextVex; w != nullptr; w = w->nextVex) {

//outfile << j.name << " " << w->name << endl;

arcnum++;

}

}

outfile << arcnum << endl;

for (auto j : i.List) {

//边的信息

for (Vex \* w = j.nextVex; w != nullptr; w = w->nextVex) {

outfile << j.name << " " << w->name << endl;

}

}

}

outfile.close();

}

void Index::LoadData() {

ifstream infile;

infile.open("data.txt", ios::in);

int graphnum;

int vexnum;

int arcnum;

string graphname;

string vexname1;

string vexname2;

ElemType vexvalue;

infile >> graphnum;

for (int i = 0; i < graphnum; i++) {

infile >> graphname;

Graph G(graphname);

List.push\_back(G);

infile >> vexnum;

for (int j = 0; j < vexnum; j++) {

infile >> vexname1;

infile >> vexvalue;

InsertVex(graphname, vexname1, vexvalue);

}

infile >> arcnum;

for (int m = 0; m < arcnum; m++) {

infile >> vexname1;

infile >> vexname2;

InsertArc(graphname, vexname1, vexname2);

}

}

}

void PrintMenu() {

cout << "|-------------------Graph Experiment Menu-----------------|" << endl;

cout << "| 1. 创建图 2. 销毁图 |" << endl;

cout << "| 3. 查找顶点 4. 获得顶点值 |" << endl;

cout << "| 5. 顶点赋值 6. 获得第一邻节点 |" << endl;

cout << "| 7. 获得下一邻节点 8. 插入节点 |" << endl;

cout << "| 9. 删除节点 10. 插入弧 |" << endl;

cout << "| 11. 删除弧 12. 深度优先 |" << endl;

cout << "| 13. 广度优先 14. 保存数据 |" << endl;

cout << "| 15. 导入数据 16. 打印邻接表 |" << endl;

}

void visit(Vex v) {

cout << "(" << v.name << ", " <<v.data << ") ";

}

int main(void) {

int op = 1;

string graphname;

string vexname;

string vexname2;

ElemType vexvalue;

Index index;

PrintMenu();

while (op) {

cin >> op;

system("cls");

PrintMenu();

switch (op)

{

case 1:

cout << "请输入要创建的图名" << endl;

cin >> graphname;

if (index.GraphExist(graphname) >= 0) {

cout << "该图名已被使用" << endl;

}

else {

index.CreateGraph(graphname);

cout << "创建成功" << endl;

}

break;

case 2:

cout << "请输入要销毁的图名" << endl;

cin >> graphname;

if (index.GraphExist(graphname) >= 0) {

index.DestroyGraph(graphname);

cout << "销毁成功" << endl;

}

else {

cout << "输入的图名不存在" << endl;

}

break;

case 3:

cout << "请输入要查找的节点所在图的名字以及节点本身的名字" << endl;

cin >> graphname;

cin >> vexname;

if (index.VexExist(graphname, vexname)) {

Vex v = index.LocateVex(graphname, vexname);

cout << "位于图" << graphname << "的节点" << vexname << "的值为" << v.data;

}

else {

cout << "输入的节点不存在" << endl;

}

break;

case 4:

cout << "请输入要获取的节点所在图的名字以及节点本身的名字" << endl;

cin >> graphname;

cin >> vexname;

if (index.VexExist(graphname, vexname)) {

int value = index.GetVex(graphname, vexname);

cout << "位于图" << graphname << "的节点" << vexname << "的值为" << value;

}

else {

cout << "输入的节点不存在" << endl;

}

break;

case 5:

cout << "请输入要赋值的节点所在图的名字以及节点本身的名字" << endl;

cin >> graphname;

cin >> vexname;

if (index.VexExist(graphname, vexname)) {

cout << "位于图" << graphname << "的节点" << vexname << "的值为" << index.GetVex(graphname, vexname) << endl;

cout << "请输入新值" << endl;

cin >> vexvalue;

index.PutVex(graphname, vexname, vexvalue);

cout << "位于图" << graphname << "的节点" << vexname << "的值已修改为" << vexvalue << endl;

}

else {

cout << "输入的节点不存在" << endl;

}

break;

case 6:

cout << "请输入要查找的节点所在图的名字以及节点本身的名字" << endl;

cin >> graphname;

cin >> vexname;

if (index.VexExist(graphname, vexname)) {

Vex \* v = index.FirstAdjVex(graphname, vexname);

if (v) {

cout << "位于图" << graphname << "的节点" << vexname << "的第一邻接点为";

cout << "名为" << v->name << "且值为" << v->data << "的节点" << endl;

}

else {

cout << "位于图" << graphname << "的节点" << vexname << "不存在第一邻接点";

}

}

else {

cout << "输入的节点不存在" << endl;

}

break;

case 7:

cout << "请输入两个节点所在图的名字以及两个节点的名字" << endl;

cin >> graphname;

cin >> vexname;

cin >> vexname2;

if (index.VexExist(graphname, vexname) && index.VexExist(graphname, vexname2)) {

Vex \* v = index.NextAdjVex(graphname, vexname, vexname2);

if (v) {

cout << "位于图" << graphname << "的" << vexname << "相对于" << vexname2 << "的下一邻接节点为";

cout << "名为" << v->name << "且值为" << v->data << "的节点" << endl;

}

else {

cout << "位于图" << graphname << "的" << vexname << "相对于" << vexname2 << "的下一邻接节点不存在";

}

}

else {

cout << "输入的节点或图不存在" << endl;

}

break;

case 8:

cout << "请输入目标图的名称和要插入的节点的名称以及值" << endl;

cin >> graphname;

cin >> vexname;

cin >> vexvalue;

if (index.GraphExist(graphname) >= 0) {

if (index.VexExist(graphname, vexname)) {

cout << "目标图" << graphname << "中已存在名为" << vexname << "的节点" << endl;

}

else {

index.InsertVex(graphname, vexname, vexvalue);

cout << "插入成功" << endl;

}

}

else {

cout << "目标图不存在" << endl;

}

break;

case 9:

cout << "请输入目标图的名称和要删除的节点的名称" << endl;

cin >> graphname;

cin >> vexname;

if (index.GraphExist(graphname) >= 0) {

if (index.VexExist(graphname, vexname)) {

if (index.DeleteVex(graphname, vexname) == OK)

cout << "删除成功" << endl;

else

cout << "删除失败" << endl;

}

else {

cout << "目标图" << graphname << "中已不存在名为" << vexname << "的节点" << endl;

}

}

else {

cout << "目标图不存在" << endl;

}

break;

case 10:

cout << "请输入要插入的图的名称以及两个顶点的名称" << endl;

cin >> graphname;

cin >> vexname;

cin >> vexname2;

if (index.GraphExist(graphname) >= 0) {

if (vexname == vexname2) {

cout << "无法在一个节点上创建弧" << endl;

}

else if (index.VexExist(graphname, vexname) && index.VexExist(graphname, vexname2)) {

if (index.InsertArc(graphname, vexname, vexname2) == OK) {

cout << "成功插入" << endl;

}

else

cout << "插入失败" << endl;

}

else

cout << "目标图中不存在顶点" << vexname << "或" << vexname2 << endl;

}

else {

cout << "目标图不存在" << endl;

}

break;

case 11:

cout << "请输入要删除的弧所在图的名称以及两个顶点的名称" << endl;

cin >> graphname;

cin >> vexname;

cin >> vexname2;

if (vexname != vexname2) {

if (index.VexExist(graphname, vexname) && index.VexExist(graphname, vexname2)) {

if (index.DeleteArc(graphname, vexname, vexname2) == OK) {

cout << "成功删除" << endl;

}

else

cout << "删除失败" << endl;

}

else

cout << "目标图中不存在顶点" << vexname << "或" << vexname2 << endl;

}

else {

cout << "节点相同 删除失败" << endl;

}

break;

case 12:

cout << "请输入要进行深度优先遍历的图的名称" << endl;

cin >> graphname;

cout << "遍历结果为：" << endl;

if (index.GraphExist(graphname) >= 0) {

index.Clearvisited(graphname);

index.DFSTraverse(graphname, visit);

cout << endl;

}

else

cout << "目标图不存在" << endl;

break;

case 13:

cout << "请输入要进行广度优先遍历的图的名称" << endl;

cin >> graphname;

cout << "遍历结果为：" << endl;

if (index.GraphExist(graphname) >= 0) {

index.Clearvisited(graphname);

index.BFSTraverse(graphname, visit);

cout << endl;

}

else

cout << "目标图不存在" << endl;

break;

case 14:

index.SaveData();

cout << "保存成功" << endl;

break;

case 15:

index.LoadData();

cout << "导入成功" << endl;

case 16:

index.PrintTable();

break;

default:

op = 0;

break;

}

}

return 0;

}