

**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1607**

**学 号： U201614700**

**姓 名： 王亚宁**

**指导教师： 李剑军**

**报告日期： 2017年 12月 1 日**

**计算机科学与技术学院**

目 录

[1 基于顺序存储结构的线性表实现 2](#_Toc503363027)

[1.1 问题描述 2](#_Toc503363028)

[1.2 系统设计 2](#_Toc503363029)

[1.3 系统实现 3](#_Toc503363030)

[1.4 实验小结 17](#_Toc503363031)

[2 基于链式存储结构的线性表实现 18](#_Toc503363032)

[2.1 问题描述 18](#_Toc503363033)

[2.2 系统设计 18](#_Toc503363034)

[2.3 系统实现 19](#_Toc503363035)

[2.4 实验小结 28](#_Toc503363036)

[3 基于二叉链表的二叉树实现 30](#_Toc503363037)

[3.1 问题描述 30](#_Toc503363038)

[3.2 系统设计 30](#_Toc503363039)

[3.3 系统实现 32](#_Toc503363040)

[3.4 实验小结 57](#_Toc503363041)

[4 基于邻接表的图实现 58](#_Toc503363042)

[4.1 问题描述 58](#_Toc503363043)

[4.2 系统设计 58](#_Toc503363044)

[4.3 系统实现 61](#_Toc503363045)

[4.4 实验小结 77](#_Toc503363046)

[参考文献 78](#_Toc503363047)

[附录A 基于顺序存储结构线性表实现的源程序 81](#_Toc503363048)

[文件linearList.h内容 81](#_Toc503363049)

[文件testLinearList.h内容： 88](#_Toc503363050)

[附录B 基于链式存储结构线性表实现的源程序 93](#_Toc503363051)

[文件linkedList.h内容： 93](#_Toc503363052)

[文件testLinkedList.h内容： 102](#_Toc503363053)

[附录C 基于二叉链表二叉树实现的源程序 108](#_Toc503363054)

[文件binaryTree.h内容： 108](#_Toc503363055)

[文件testBinaryTree.h内容： 120](#_Toc503363056)

[附录D 基于邻接表图实现的源程序 128](#_Toc503363057)

[文件graph.h内容 128](#_Toc503363058)

[文件testGraph.txt 143](#_Toc503363059)

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

基于内存中的连续空间实现线性表，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。

## 1.2 系统设计

实现所需要的几种功能：初始化表、销毁表、清空表、判断空表、求表长、根据位置获取元素、查找元素、获得前驱、获得后驱、插入元素、删除元素和遍历表，根据实际需求我还自己添加了从文件初始化和将表保存到文件的功能。下面是具体的每个函数声明。

由于我使用的是C++，所以函数在使用时不需要传入表参数，直接使用表名进行调用即可，在构造函数中我将表的参数初始化，并没有为其分配空间，实际的函数如下。

1. 初始化表：status InitaList(),使用对象名调用，不需参数。当表已经存在时不进行初始化，输出错误信息并返回，其中我添加了从文件初始化的选项。返回值为初始化结果。
2. 销毁表：status DestroyList()，使用对象名调用，不需参数，当表未初始化或者已经销毁过的时候返回错误。返回值为是否销毁成功。
3. 清空表：函数名称是ClearList()；初始条件是线性表已存在；操作结果是将表重置为空表。
4. 判定空表：函数名称是ListEmpty()；初始条件是线性表已存在；操作结果是若表为空表则返回true,否则返回false。
5. 求表长：函数名称是ListLength()；初始条件是线性表已存在；操作结果是返回表中数据元素的个数。
6. 获得元素：函数名称是GetElem(i,e)；初始条件是线性表已存在，1≤i≤ListLength()；操作结果是用e返回表中第i个数据元素的值。
7. 查找元素：函数名称是LocateElem(e, comp)；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comp（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。
8. 获得前驱：函数名称是PriorElem(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。
9. 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。
10. 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。
11. 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。
12. 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()。

## 1.3 系统实现

1.3.1代码实现

代码实现请见附带文件。

1.3.2程序测试

对程序的测试如下：

首先是表的初始化，使用文件初始化，首先读入元素的个数，然后将元素一次读入表中，见图1-1：

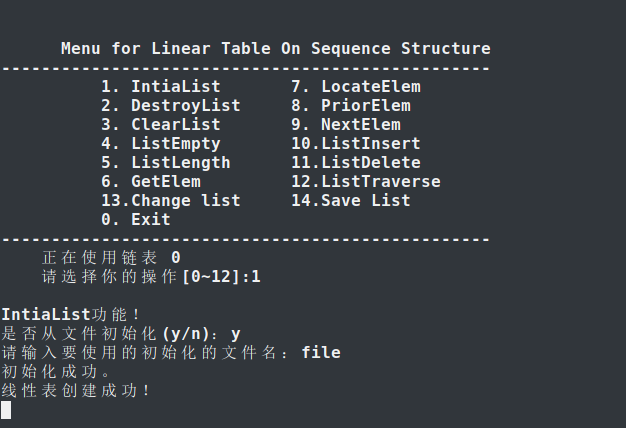


图1-1

初始化完成后元素如图1-2：



图1-2

检测一下链表现在是否为空，见图1-3：

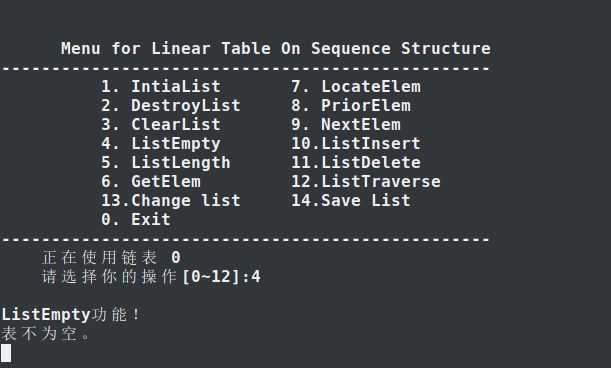


图1-3

得到表的长度，见图1-4：

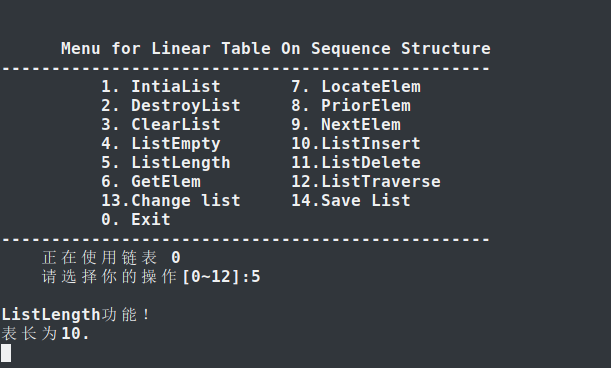


图1-4

使用GetElem函数获取元素，见图1-5,1-6,1-7：

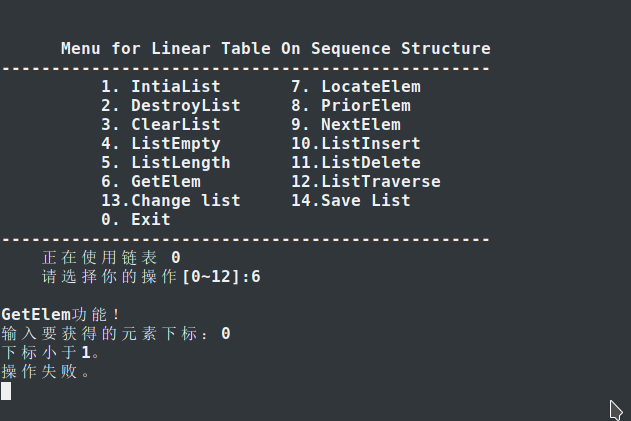


图1-5

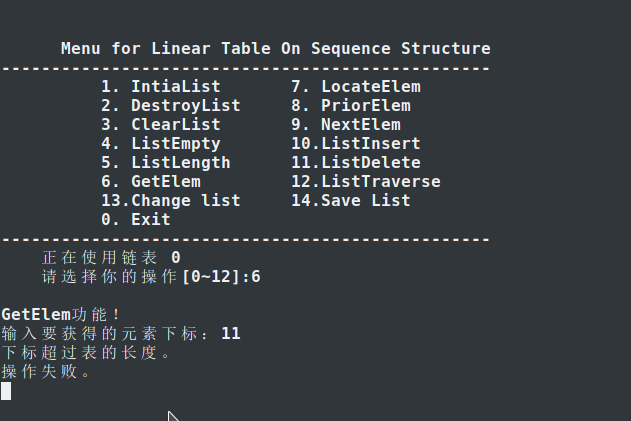


图1-6

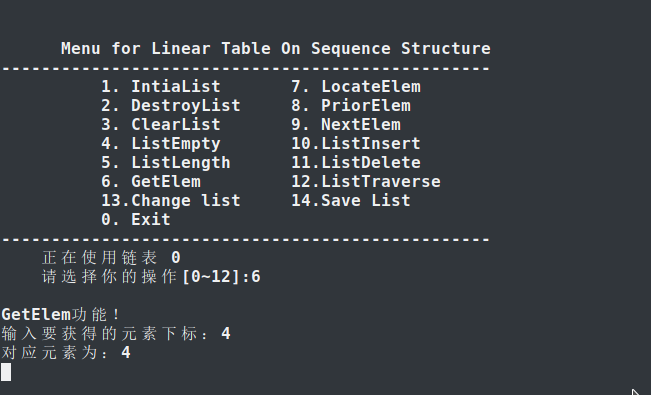


图1-7

定位元素的操作见图1-8,1-9,1-10：

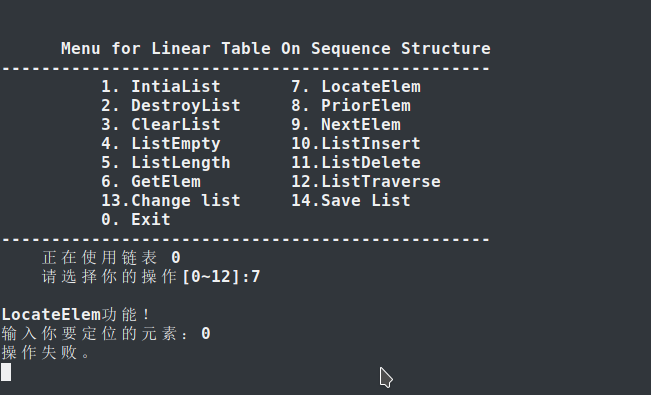


图1-8

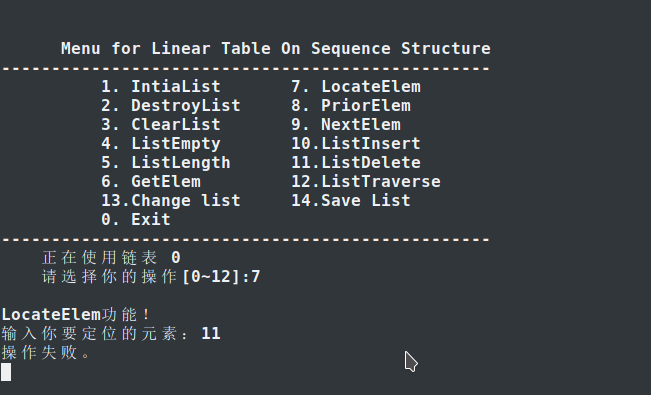


图1-9

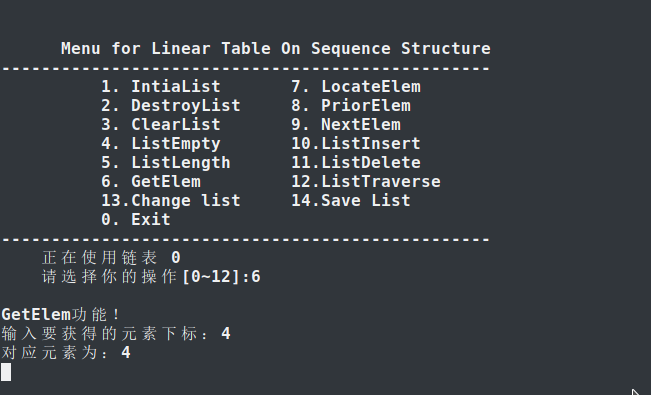


图1-10

获得前驱的操作见图1-11,1-12：

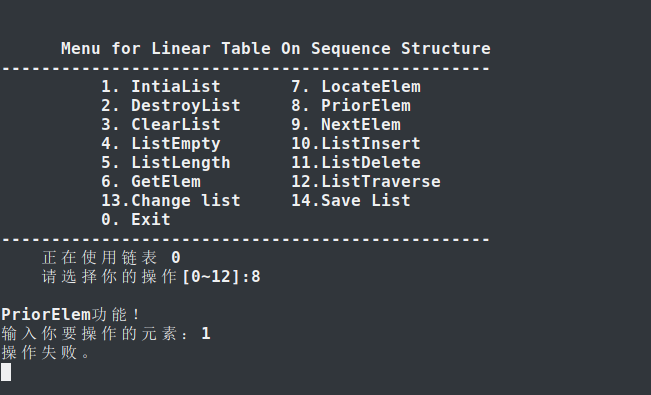


图1-11

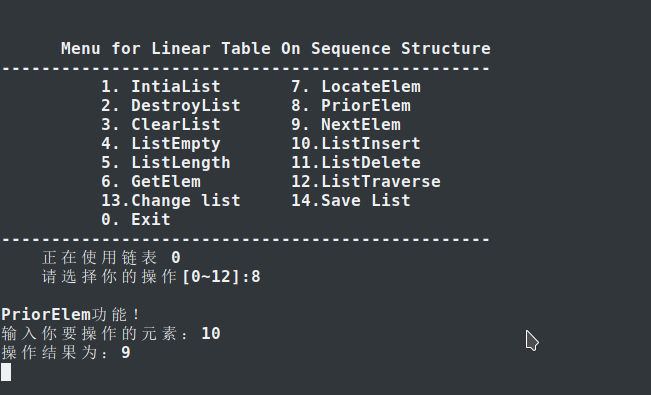


图1-12

获得后驱的函数测试1-13,1-14：

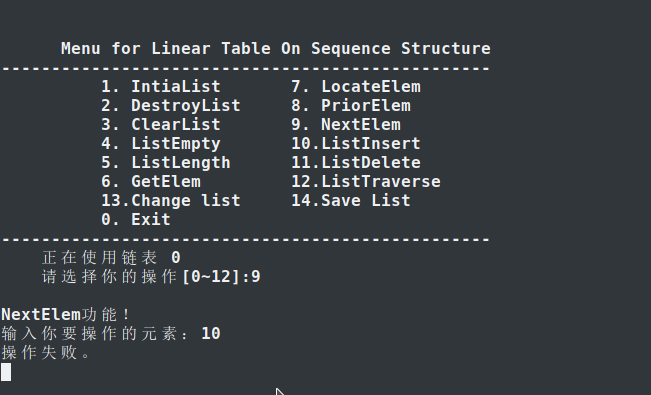


图1-13

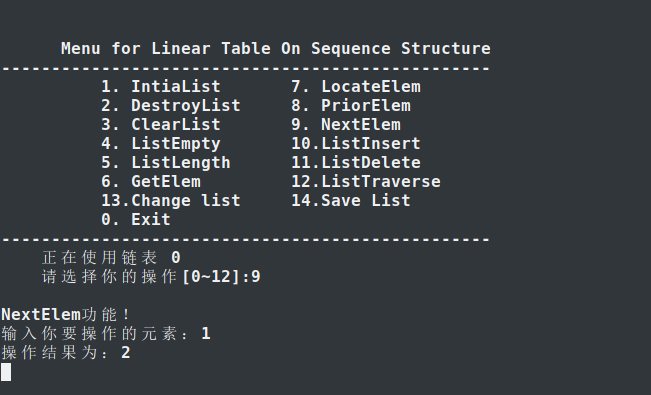


图1-14

元素插入测试见图1-15,1-16，1-17：

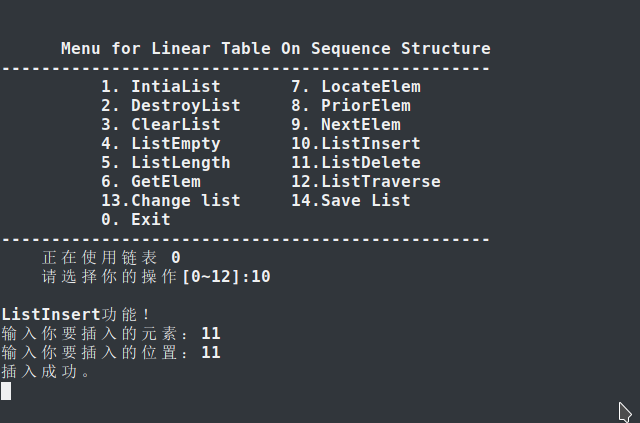


图1-15



图1-16

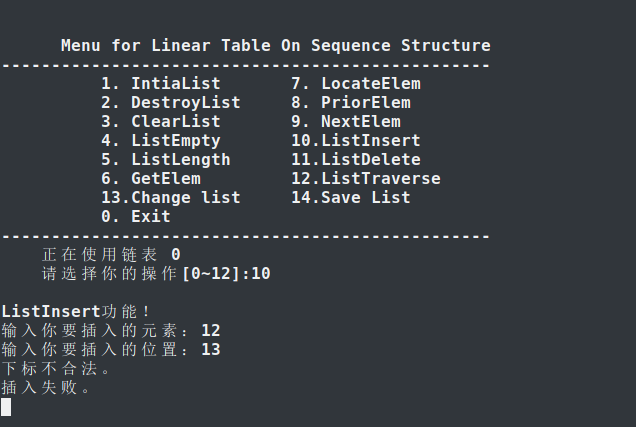


图1-17

元素删除测试如下图1-18,1-19,1-20：

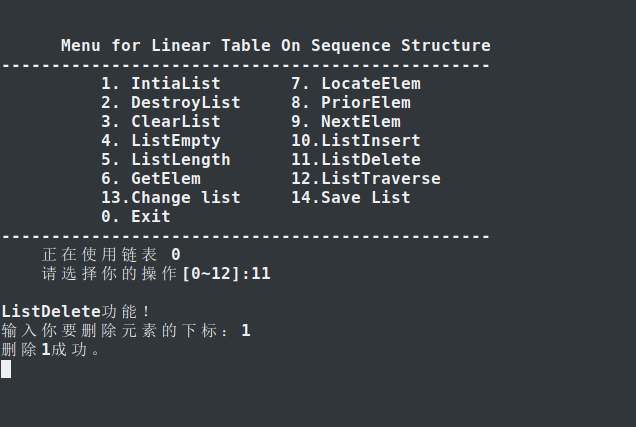


图1-18



图1-19

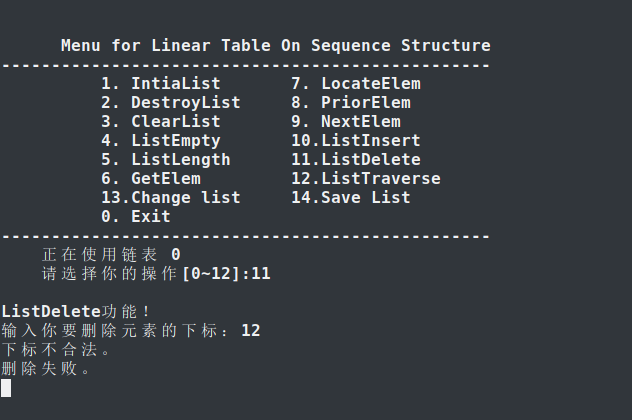


图1-20

保存链表测试见图1-21,1-22：

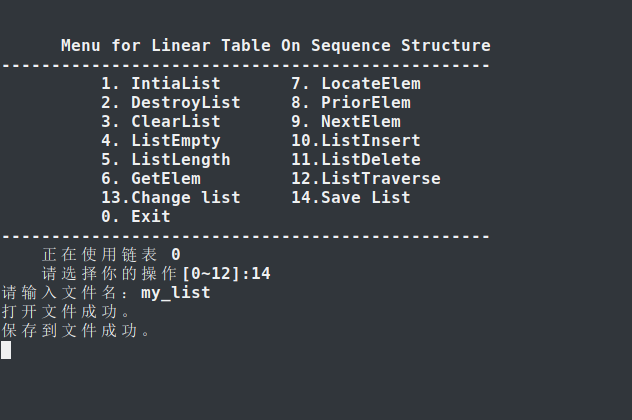


图1-21

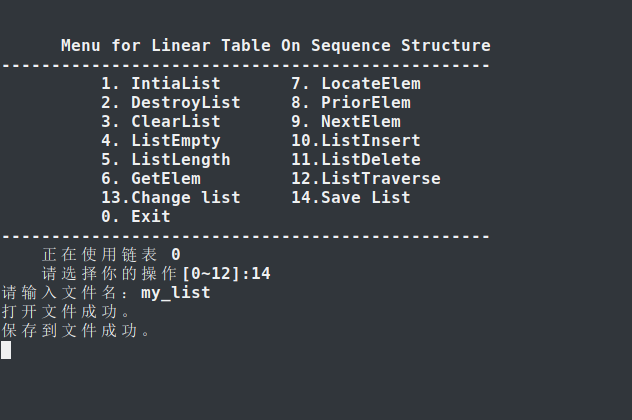


图1-22

切换链表操作见图1-23,1-24,1-25：

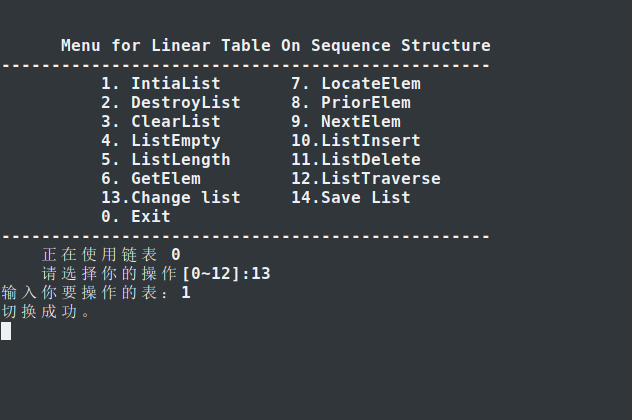


图1-23

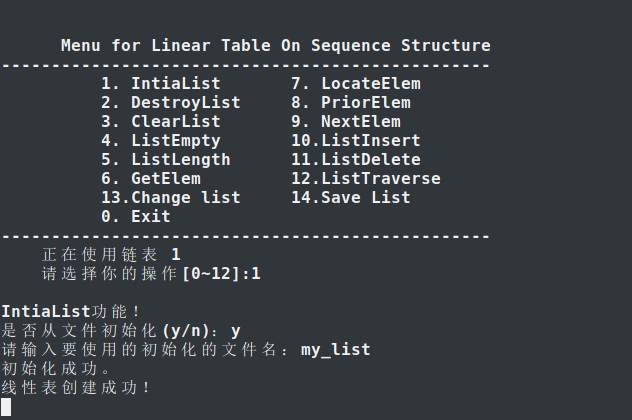


图1-24



图1-24

## 1.4 实验小结

通过自己实现线性表，我发现了一些在程序设计中比较重要的问题，在设计循环的时候要特别注意边缘检测，良好的测试数据和测试习惯也很重要。对自己的程序设计能力有很大的提升。

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

本次实验主要是使用链表来实现一个线性表，主要涉及初始化、插入、删除等操作，操作和上次的基于顺序表的线性表的操作是一样的。

## 2.2 系统设计

使用链表实现线性表，主要就是使用链表和相应的函数完成对和线性表一样的操作，首先需要设计一个结构体作为结点类型：

首先是初始化操作，初始化操作首先判断此表是否已经进行了初始化，如果已经进行了就返回错误，否则进行空间开辟和变量初始化等操作。

摧毁表，首先判断表是否未初始化（由于判断初始化操作太普遍，以下操作说明时省略此步判断，代码中会书写），未初始化返回错误，不能进行摧毁，否则进行摧毁操作释放空间并设置变量值，初始化状态更改为false。

清空表，清空表就是摧毁表后将初始化状态设置为true，对于链表不需要为后续元素预留空间。

判定空表，空表就是长度为0的表，返回长度是否为0即可，表未初始化也返回true。

求表长操作，未初始化的表长度默认为0，已经初始化的表长度即为表长变量的值，返回变量即可。

获得元素操作，需要根据给定的下标来返回对应的元素，首先应该判断给定的下表是否合法，不合法表示元素不可能存在，返回错误标志，若下表合理就对从表头指针向后移动适当距离即可得到需要的元素所在的结构指针，返回元素即可。

获得前驱元素，判断空表，然后将表头指针向后移动，并判断指针的后继所指的结构体的元素是否为和给定的元素相等，相等就返回当前指针的元素，即为所求前驱。

获得后继，操作大部分与获得前驱相同，区别在于判断的是当前元素是否与给定相同，相同返回next指针指向的元素。

插入操作，首先判断下标合法性，需要注意插入到头指针的时head指针需要改变，普通下标插入后移适当位置到达其前驱结点处即可，再进行插入，完成后长度加一即可。

删除操所，和插入操作大部分类似，删除后长度减一。

遍历表，使用while或者for循环即可。

## 2.3 系统实现

2.3.1代码实现

系统的代码实现可见附带文件。

2.3.2程序测试

进行测试：首先测试初始化功能，使用自带文件初始化，如图2-1：

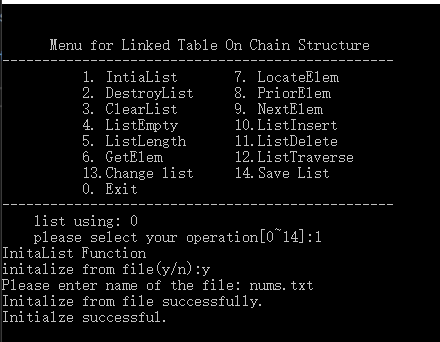


图2-1

初始化成功，接下来检测表是否为空，如图2-1：

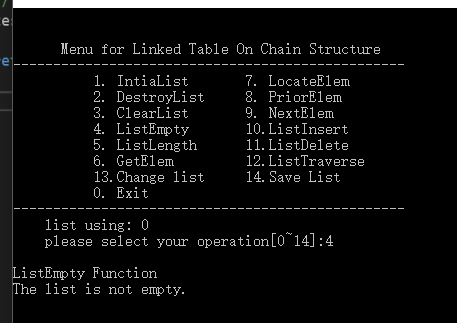


图2-1

获取表长操作见图2-3：

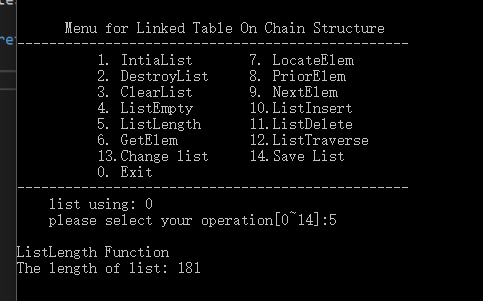


图2-3

获取指定下标的元素，如图2-4，2-5：

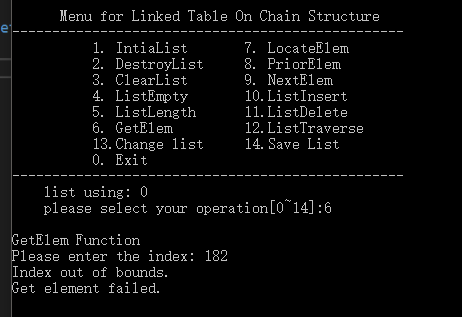


图2-4

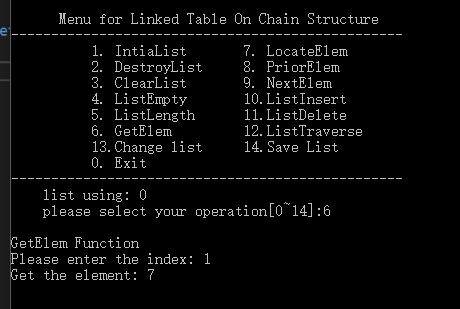


图2-5

定位元素，见图2-6、2-7：

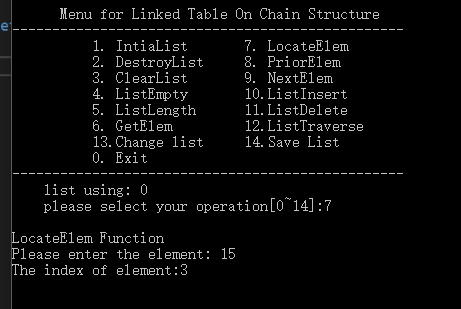


图2-6

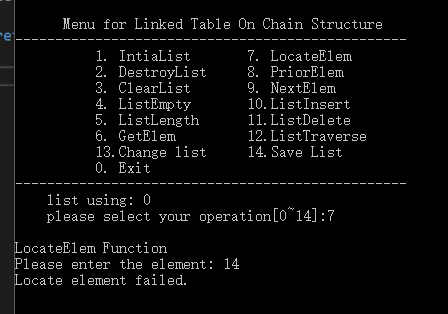


图2-7

测试获取前驱，如图2-8、2-9：

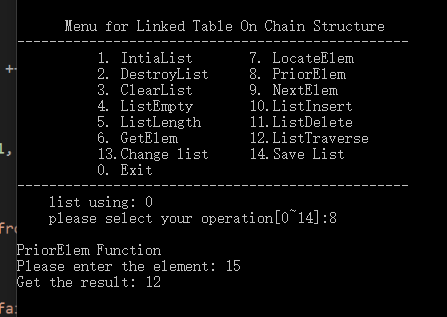


图2-8

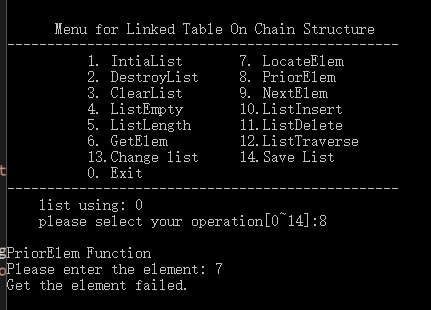


图2-9

获得后继函数测试，见图2-10、2-11：

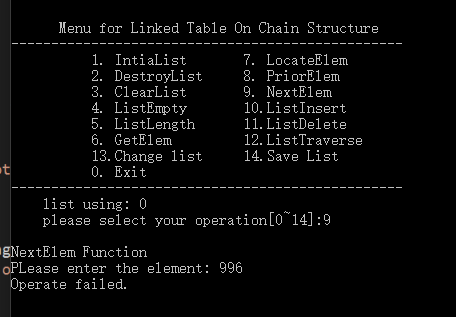


图2-10

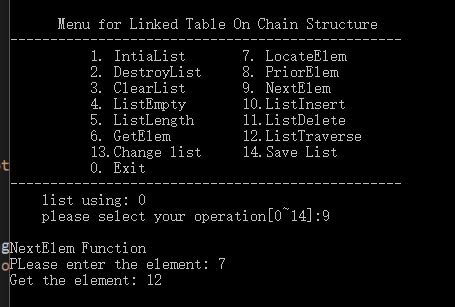


图2-11

插入函数测试，见图2-12，图2-13，图2-14：

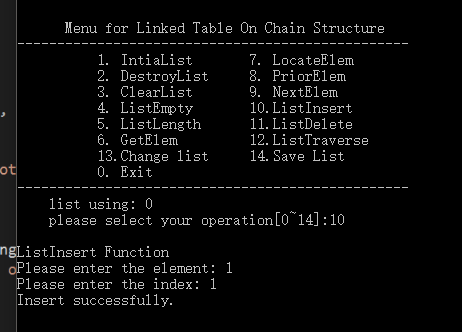


图2-12

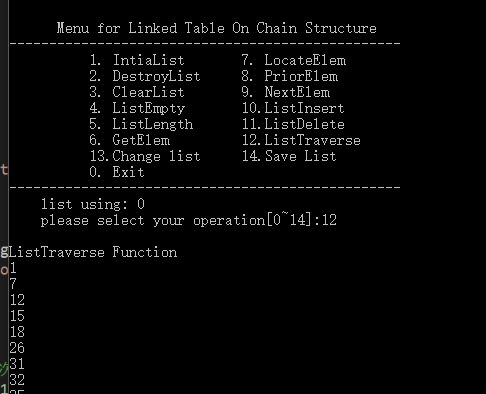


图2-13

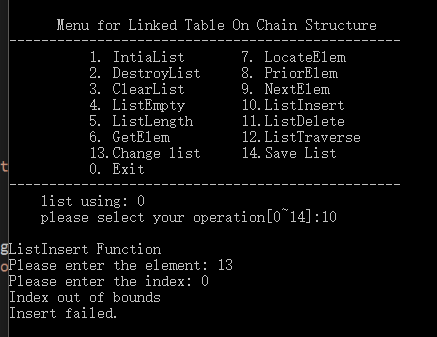


图2-14

对链表进行删除元素如图2-15、图2-16：

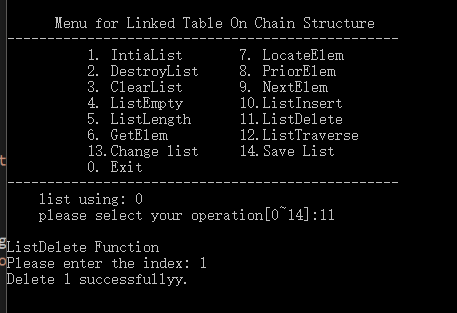


图2-15

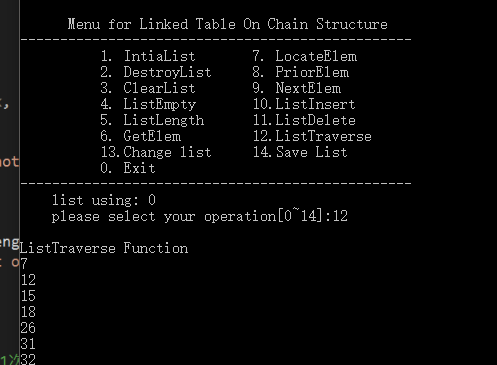


图2-16

链表遍历如图2-17、2-18：

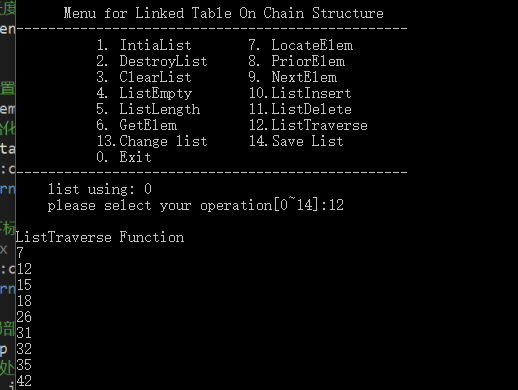


图2-17

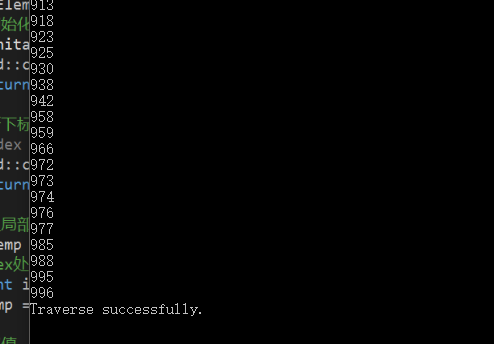


图2-18

将表保存到文件，如图2-19：

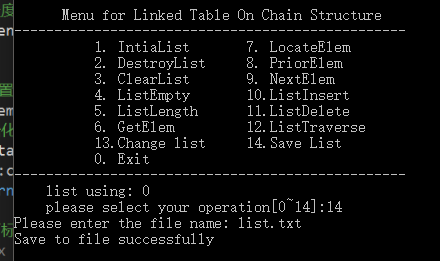


图2-19

通过ChangeList可以切换链表。

## 2.4 实验小结

# 3 基于二叉链表的二叉树实现

## 3.1 问题描述

本次实验的目的是让我们熟悉二叉树的一些操作，实验要求我们使用二叉链表实现二叉树，这个和二叉树的逻辑层面是比较接近的，难度较小，实验内容主要就是完成二叉树的构建、查询、赋值、插入、删除等操作。

## 3.2 系统设计

这次实验需要实现的函数有一点多，难度也比前两次的大上不少，还是比较难的。首先我要对二叉树这个数据结构进行抽象数据结构的定义，二叉树主要由结点和它们之间的指针，还有对树的操作组成，首先要定义二叉树这个类型，我使用c++的模板类对其进行定义，模板参数就是其结点存储的数据类型，这样便于代码重用。内部的结点使用二叉链表实现，结点定义如下：

*// 结点定义*

**template**<**typename** ValueType>

**struct** Node {

*// 键域，方便二叉树使用*

int key;

*// 值域*

ValueType value;

*// 左孩子指针*

Node<ValueType> \*left = **nullptr**;

*// 右孩子指针*

Node<ValueType> \*right = **nullptr**;

*// 父节点指针*

Node<ValueType> \*parent;

*// 禁止默认构造函数，防止父指针未被初始化*

Node() = **delete**;

*// 唯一的构造函数*

Node(**const** int key, **const** ValueType &value,

Node<ValueType> \*&parent)

:value(value), parent(parent), key(key) {}

};

实验中需要实现的函数如下：

初始化函数：由于我是用c++的模板类进行实现，所以这个初始化函数势没有什么必要的，主要作用就是将树的初始化状态设置为true，表示此树可以进行插入初始化等操作。

销毁二叉树函数：主要进行的行为是将二叉树的空间进行释放，防止内存泄漏，并将树的状态标记为未初始化。

创建二叉树函数：此函数需要使用操作人员输入的值序列来进行树的初始化，输入的是先序遍历的结果（带空结点的），递归对其进行创建，最终完成树的初始化。

清空二叉树：函数和销毁二叉树一样，同样是需要释放空间，不过清空后可以直接进行进行二叉树的创建，不需要再去调用初始化函数。

判空二叉树：判断二叉树是否为空，首先判断树是否进行了初始化，未初始化的树默认判定为空，也比较符合逻辑，如果已经进行了初始化，再对树的高度判断看高度是否为0，显然0表示为空，大于0的表示不为空。

求二叉树深度的函数：直接返回此函数的重载版本对根节点深度的计算，重载版本参数为二叉树结点指针，进行的操作主要是先判断此结点是否为空，为空返回深度为0，不为空递归获取左右子树的深度，返回其中最大值加一即为结点指针下的深度。

获得根节点函数：由于使用了模板类，在二叉树已进行初始化后直接返回根节点即可。

获得结点函数：考虑到直接输入结点指针的难度，我选择先打印出树中所有结点的键和值，让使用者输入结点对应的key来进行操作，使用类的内置函数获取key所对应的结点的指针，然后返回指针即可。

结点赋值：获取结点的操作和上面的一样，输入key得到值，在检查指针有效性后对其值域赋值即可。

获得双亲结点：获取结点，返回父节点即可。

获取左孩子或者右孩子：获取结点：返回对应的结点指针即可。

获取左兄弟或者右兄弟：首先获取当前结点指针，判断非空后获取父节点指针，判断非空，再判断此节点是否就是对应结点，是的话报错，返回空指针不是就返回对应指针。

插入子树：通过函数参数获取结点指针，//todo

删除子树：通过函数参数得到对应的结点指针，根据给出的删除参数释放左子树或者右子树。

前/中/后序遍历：函数按照顺序递归访问结点即可。

层次遍历：使用STL提供的队列即可，先将根节点压入队列，当队列不为空时，现对取出的结点的值进行访问，再判断左右结点是否为空，将非空结点压入队列，将访问过的结点弹出队列，访问完毕即可。

## 3.3 系统实现

3.3.1代码实现

具体代码请见附录或者附带文件。

3.3.2程序测试

下面使用实例对代码进行测试。

程序运行界面如图3-1所示：

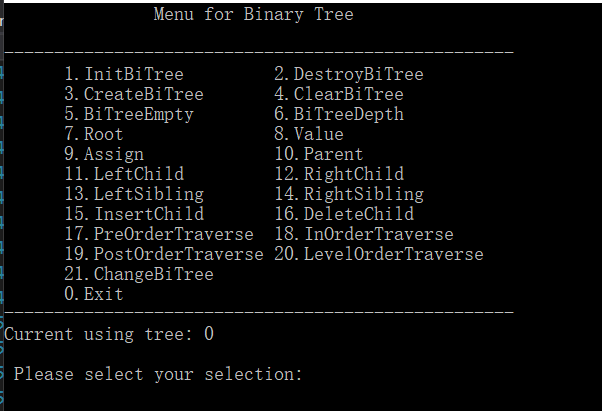


图3-1

进行初始化操作，结果如图3-2：

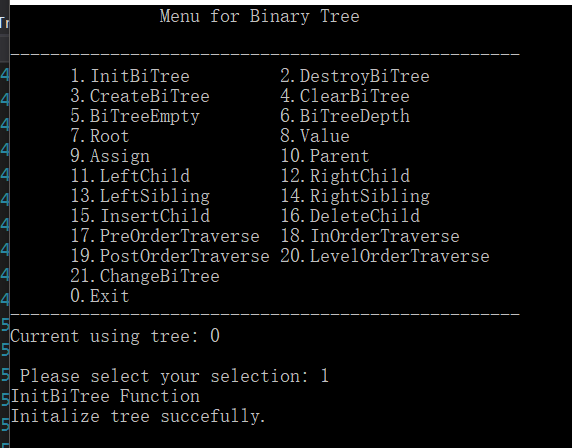


图3-2

接下来使用下面的序列进行建树，见图3-3，3-4：

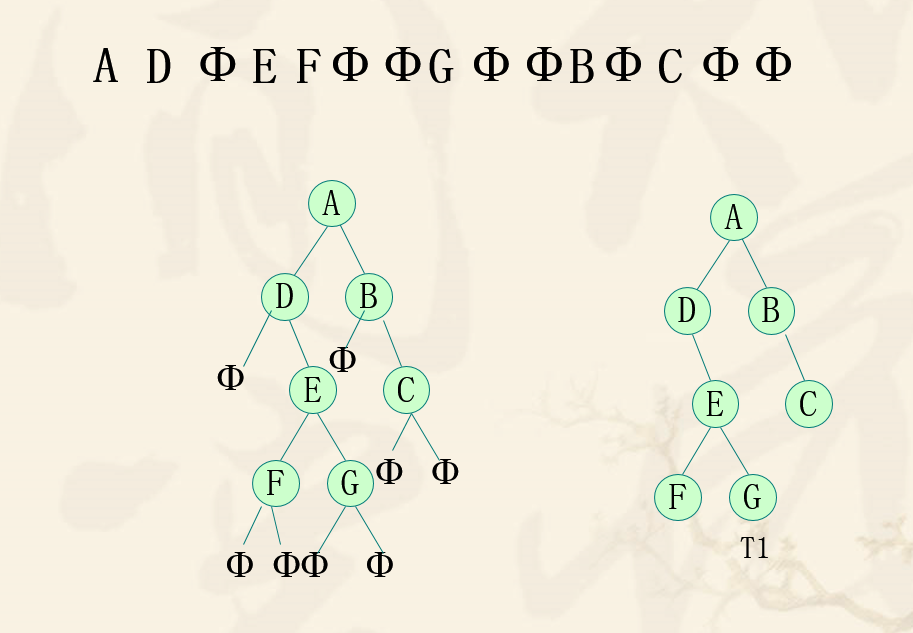


图3-3

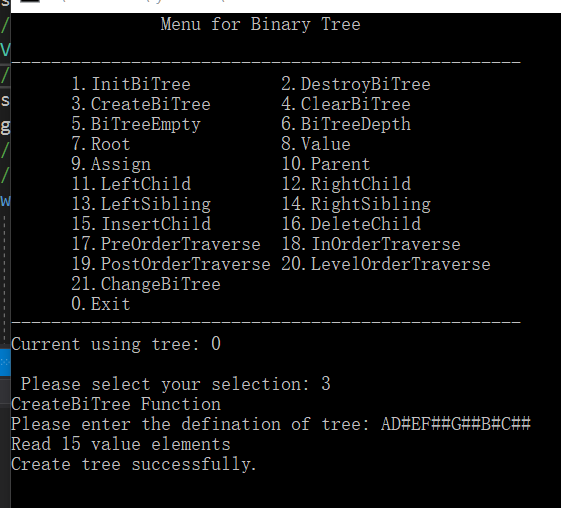


图3-4

建立完毕使用先序遍历检测一下，如图3-5：

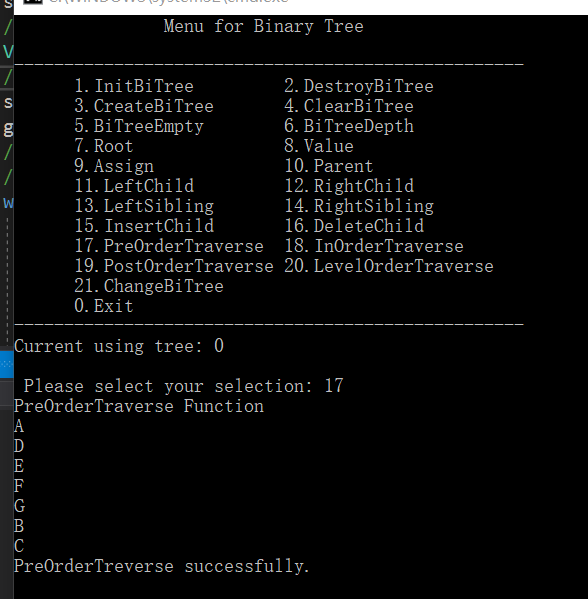


图3-5

和图中所示的顺序符合，说明创建二叉树成功。

检测二叉树是否为空，如图3-6所示：

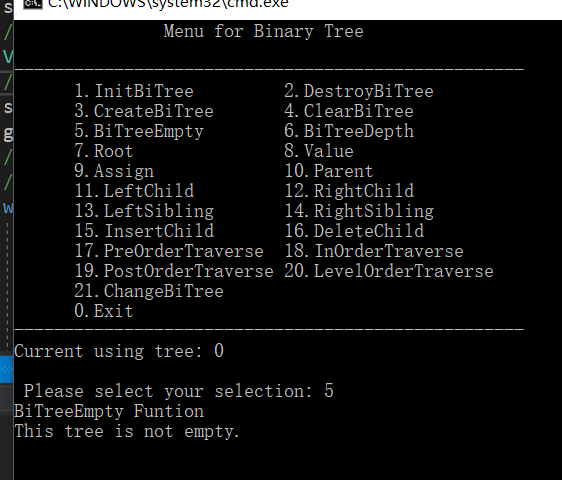


图3-6

获取树的高度或者深度，如图3-7所示：

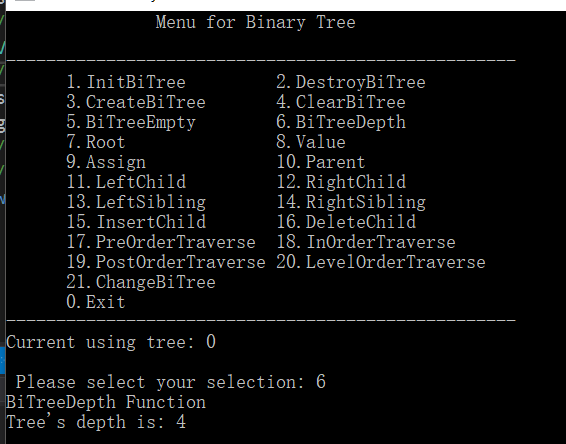


图3-7

根据使用的二叉树可知高度计算正确。

运行Root函数，结果如图3-8所示：

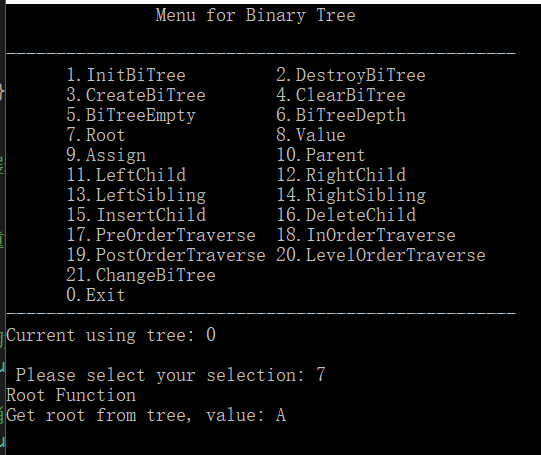


图3-8

运行Value函数，结果如图3-9所示：

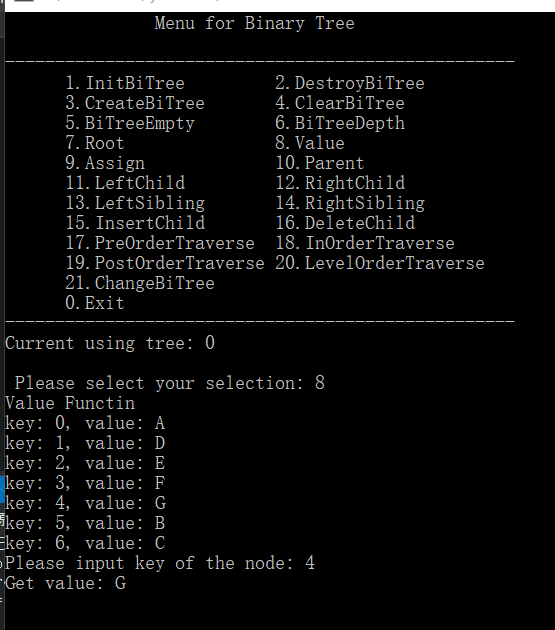


图3-9

更改结点值的Assign函数，测试结果如图3-10所示：

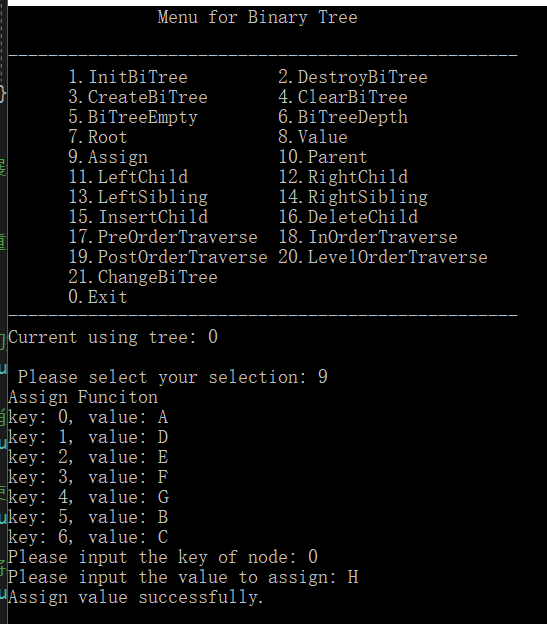


图3-10

检测一下更改是否生效，如图3-11所示：

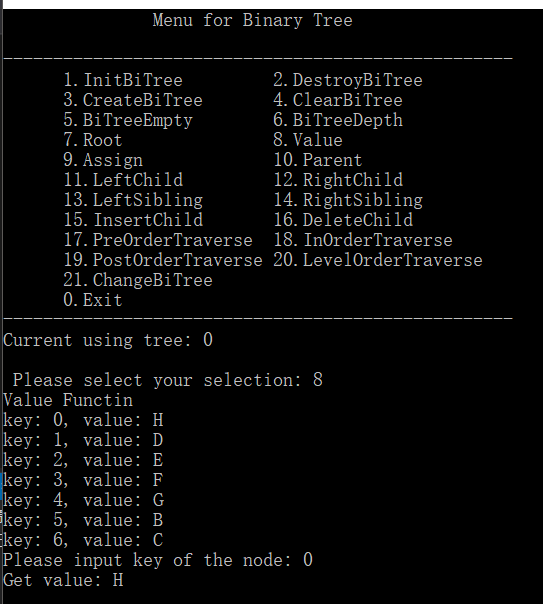


图3-11

由结果可知运行成功，更改生效。

运行Parent函数获取某个结点的父节点，测试获取C结点的父节点，由图3-3可知应为B，结果如图3-12所示：

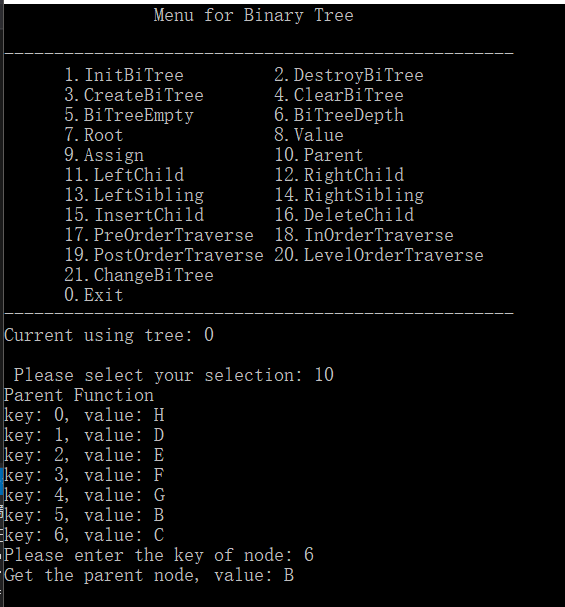


图3-12

接下来尝试获取H结点的左右子结点的值，根据图3-3可知应该为D和B，如图3-13，3-14所示：

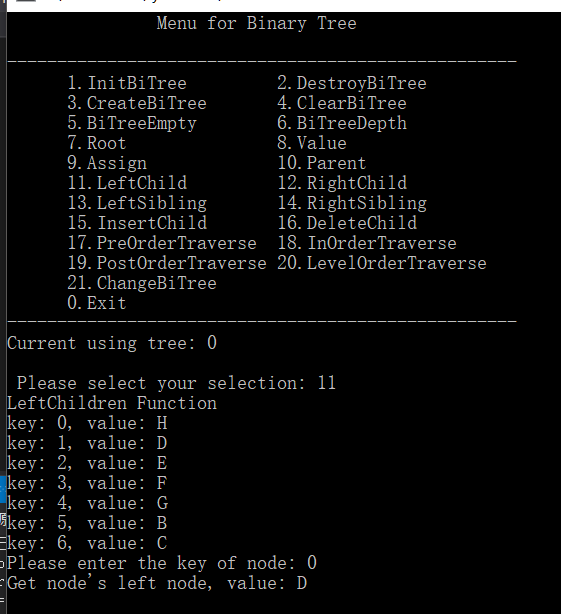


图3-13

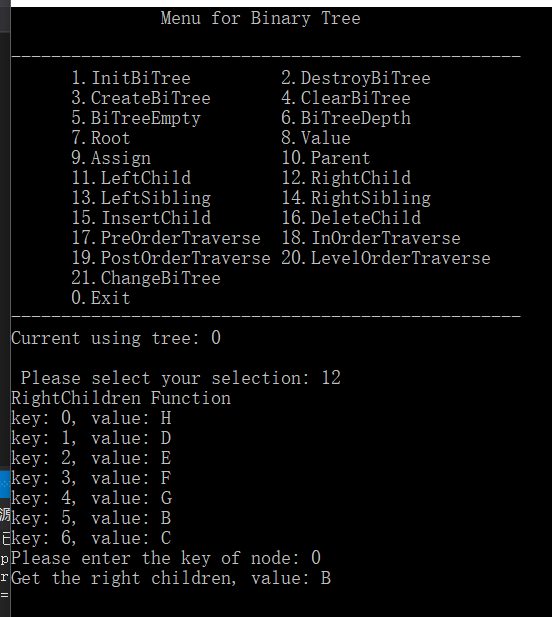


图3-14

测试兄弟结点函数，测试BD结点，两者互为左右结点，结果如图3-15，3-16所示：

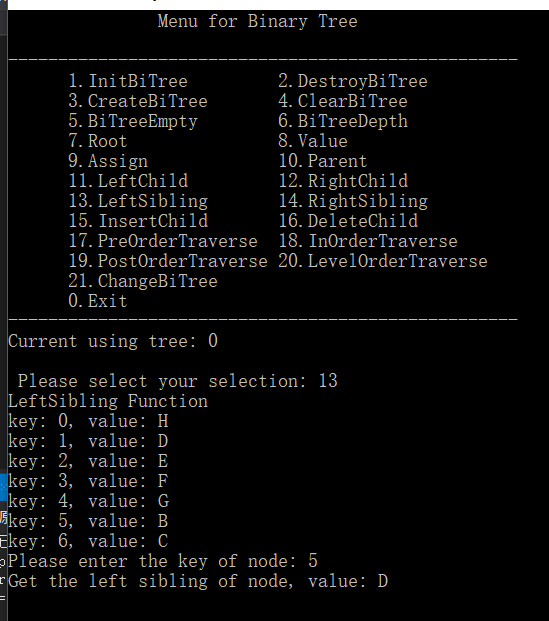


图3-15

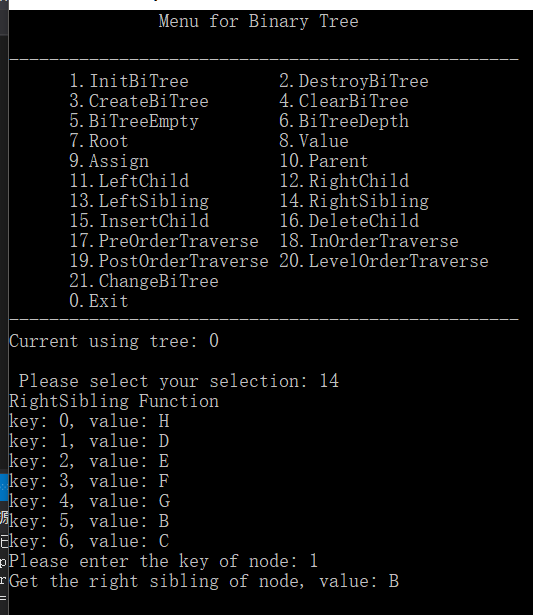


图3-16

结果正确，可知运行没有问题。

接下来进行四种遍历，由图3-3可知遍历结果应如表3-1所示：

|  |  |
| --- | --- |
| **遍历方式** | **期望结果** |
| **前序遍历** | HDEFGBC |
| **中序遍历** | DFEGHBC |
| **后序遍历** | FGEDCBH |
| **层次遍历** | HDBECFG |

表3-1：四种遍历的期望结果

测试结果如图3-17，3-18，3-19，3-20所示：

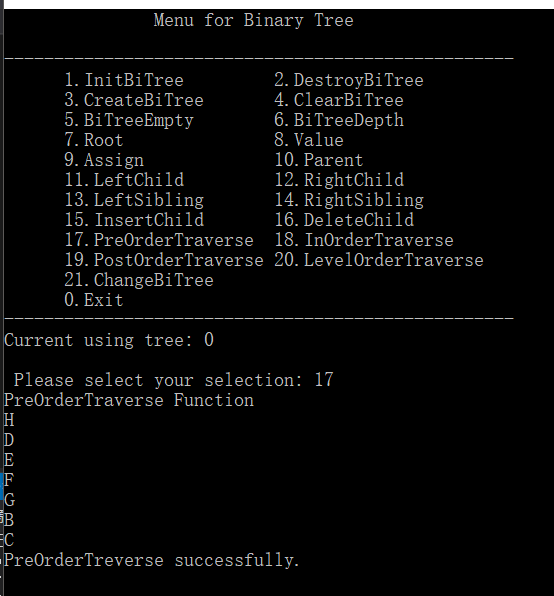


图3-17

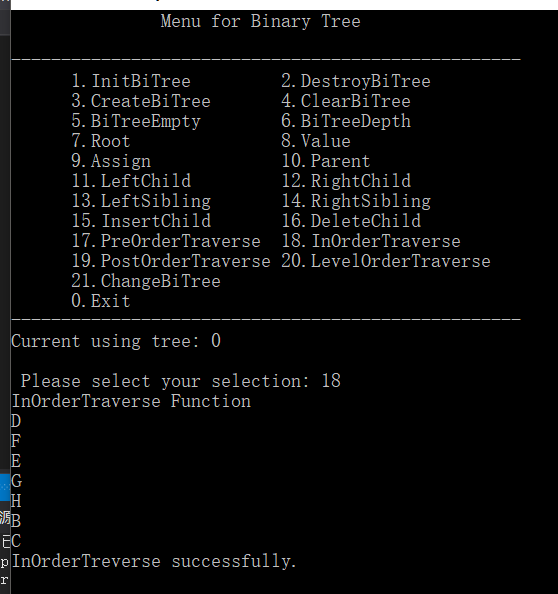


图3-18

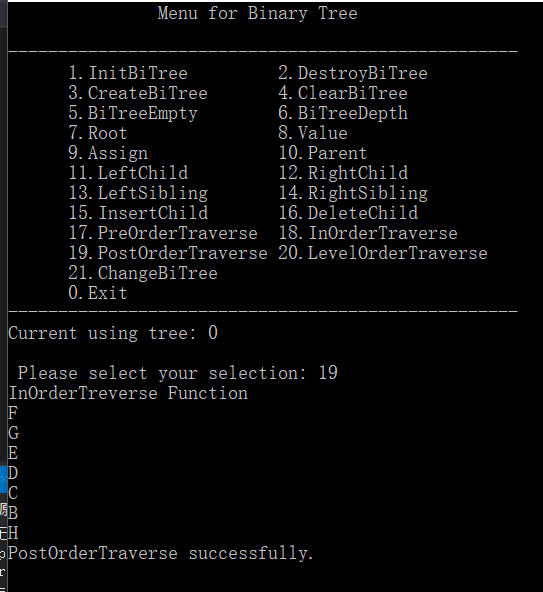


图3-19

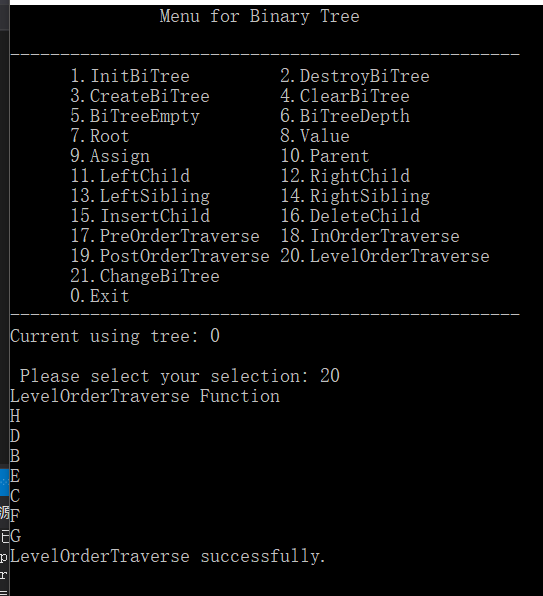


图3-20

下面测试插入子树和删除子树操作，测试将层次遍历序列为IJK##L###的子树插入到B结点左子树，

将此树插入后的前序和中序遍历结果应该变为如表3-2所示：

|  |  |
| --- | --- |
| **遍历方式** | **期望结果** |
| **前序遍历** | HDEFGBIJKLC |
| **中序遍历** | DFEGHKJLIBC |

表3-2：前序和中序遍历的期望结果

插入测试，如图3-21所示：

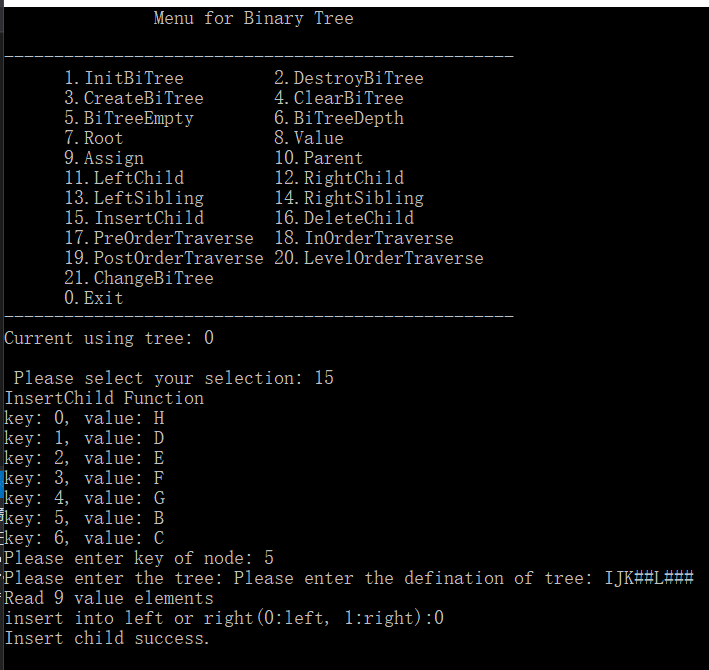


图3-21

对齐进行先序和中序遍历结果如图3-22，3-23所示：

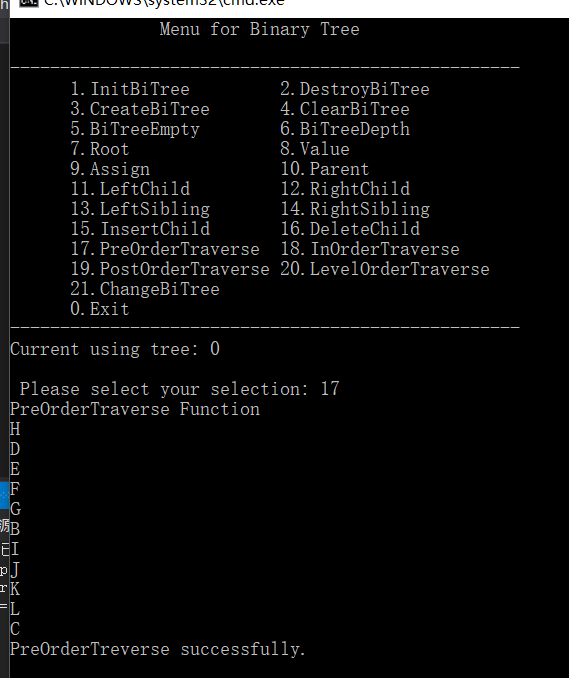


图3-22

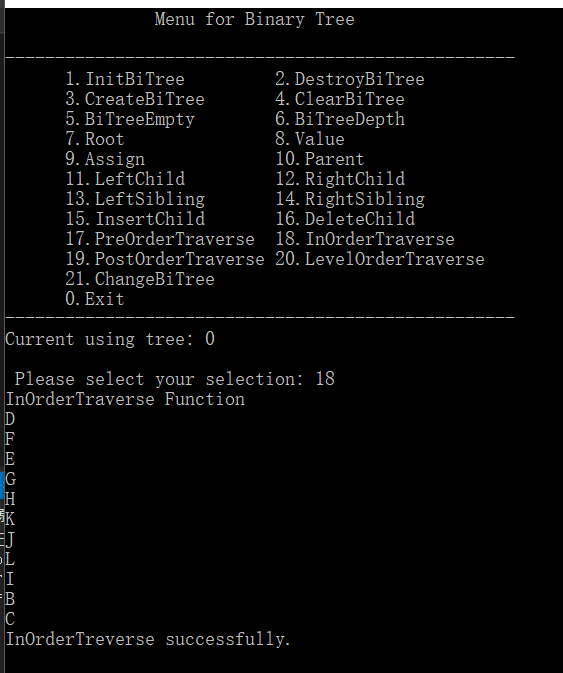


图3-23

由于前序遍历加上中序遍历可以唯一确定一棵二叉树，可知插入结果和预期结果符合，插入成功。

删除操作，测试删除D结点的右子树，删除后树的前序遍历和中序遍历期望结果如表3-3所示：

|  |  |
| --- | --- |
| **遍历方式** | **期望结果** |
| **前序遍历** | HDBIJKLC |
| **中序遍历** | DHKJLIBC |

表3-3：前序和中序遍历的期望结果

删除结果，如图3-24所示：

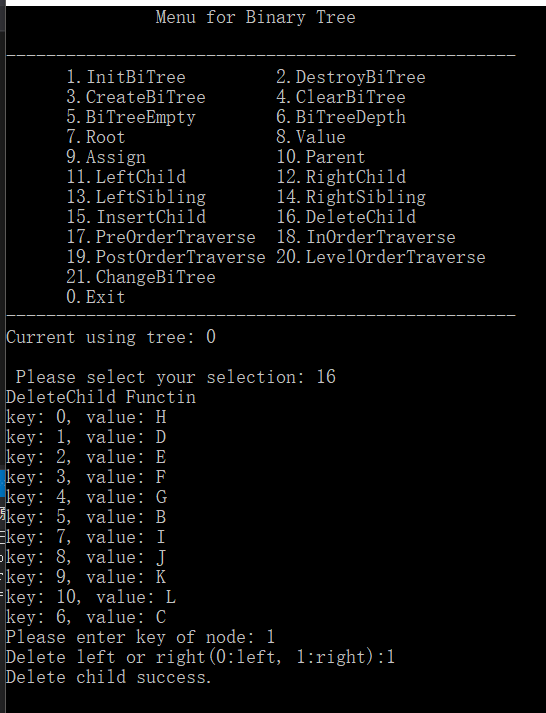


图3-24

进行前序和中序遍历结果如图3-25和3-26所示：

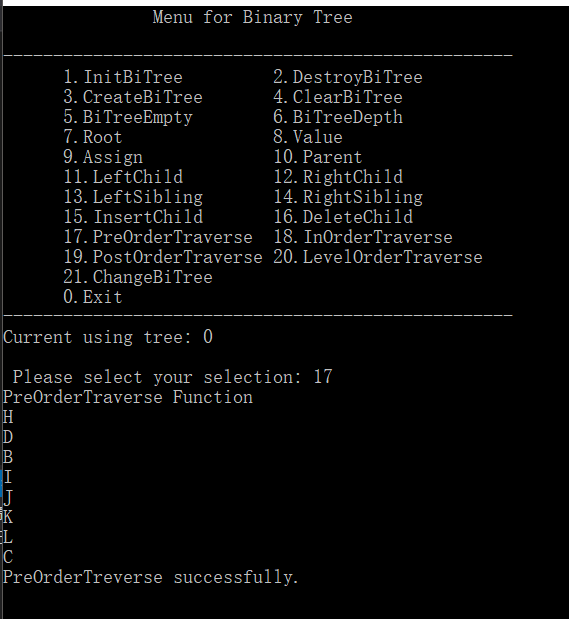


图3-25

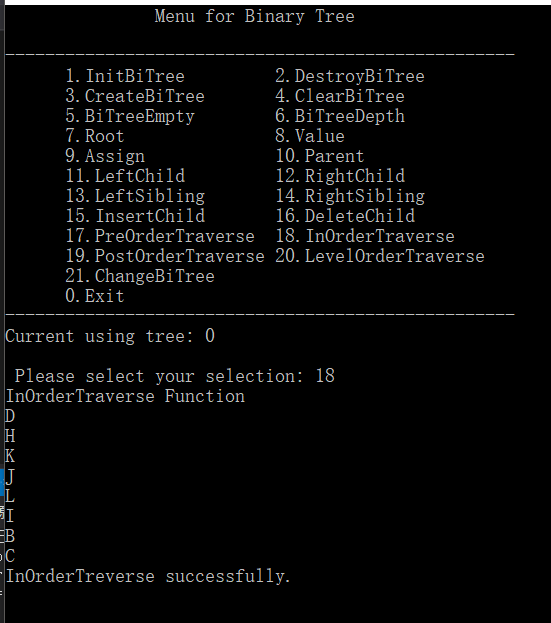


图3-26

可见运行结果和预期结果相同，程序运行正确。

## 3.4 实验小结

本次实验的二叉树和我平时写的平衡二叉树不太一样，平衡二叉树只需要提供插入删除查询接口即可，而本次的二叉树有很多意料之外的函数，比如获取根节点，插入子树，删除子树等，在实际使用中遇到的概率还是比较小的。虽然本次实验的要求比较多，但是经过两晚的努力还是圆满完成了任务，看着自己的程序能够正确运行并且没有产生Bug还是很高兴的。

# 4 基于邻接表的图实现

## 4.1 问题描述

本实验要求使用自己熟悉的一个实现方式实现图这个数据结构，并且实现几种图的基本操作。

## 4.2 系统设计

图这个数据结构由于涉及到删除结点的操作，因此不便于使用过于固定的存储方式，因为当删除节点的时候可能需要再次分配内存，导致程序效率降低，我选择使用邻接表来存储这个数据结构，实现起来还是比较有难度的，需求部分说明我的理解不太清楚，导致有几个函数都不知道在干什么，不过经过老师指点还是顺利完成了。图的数据结构里由于我需要使用邻接表实现，我使用的结点定义如下：

邻接表结点：

**template**<**typename** U>

**struct** Vertex {

*// 键：标识一个顶点*

int key;

*// 值域*

U value;

*// 和本结点相连的点的链表*

Node<U> \*next\_node = **nullptr**;

*// 下一个图结点的指针*

Vertex<U> \*next\_vertex = **nullptr**;

*// 构造函数*

Vertex<U>(**const** U &value, **const** int &key)

: value(value), key(key) {}

*// 添加结点*

bool add\_node(**const** U &value) {

**if** (next\_node == **nullptr**) {

next\_node = **new** Node<U>(value);

**return** true;

}

**else** {

**auto** temp\_n = next\_node;

**while** (temp\_n->next != **nullptr**) {

**if** (temp\_n->next->value == value) {

**return** false;

}

temp\_n = temp\_n->next;

}

temp\_n->next = **new** Node<U>(value);

**return** true;

}

**return** false;

}

*// 删除结点*

bool remove\_node(**const** U &value) {

**if** (next\_node != **nullptr**) {

**if** (next\_node->value == value) {

**auto** t = next\_node;

next\_node = next\_node->next;

**delete** (t);

**return** true;

}

**else** {

**auto** temp\_n = next\_node;

**while** (temp\_n->next != **nullptr**) {

**if** (temp\_n->next->value == value) {

**auto** t = temp\_n->next;

temp\_n->next = t->next;

**delete** (t);

**return** true;

}

temp\_n = temp\_n->next;

}

}

}

**return** false;

}

};

然后是结点所相连的所有邻接点的结点定义：

**template**<**typename** U>

**struct** Node {

*// 结点值定义*

U value;

*// 指向下个边结点的指针*

Node<U> \*next = **nullptr**;

*// 构造函数*

**explicit** Node<U>(**const** U &value) : value(value) {}

};

图里主要需要实现的操作如下：

创建图：这个函数主要是使用传参进来的结点集合和边集合构造一个图，我的思路是先使用结点集合创建结点的链表，再遍历边的集合将结点添加到对应的链表中。

销毁图：由于我是使用邻接表实现的图，所以只需要遍历所有的结点，依次释放空间即可。

查找顶点：查找结点函数，根据给出的结点，判断结点是否属于此图，如果属于此图就返回这个结点的位置信息，否则返回节点的其他信息，只需要遍历图的结点列表就可以了。

获得顶点值：给定顶点的相关信息，返回结点中存储的值，同样遍历顶点集合即可。

顶点赋值：首先需要判断顶点是否属于当前图，如果不属于的话需要向控制台输出错误信息，然后遍历顶点集合将值等于给定结点值顶点的值全部更改为给定的值。

获得第一邻接点：由于我是用邻接表存储的结点，只需先得到顶点所在结点，再获取其邻接表里第一个元素即可。

获取下一个邻接点：遍历相应顶点的邻接表即可。

插入顶点：在顶点的邻接表后接上一个顶点即可。

删除顶点：先删除顶点邻接表中的顶点结点，然后遍历各个顶点的邻接表，将给定顶点删除。

插入弧和删除弧：遍历顶点的邻接表，添加或删除对应的顶点即可。

深度优先遍历：使用map记录结点是否被遍历，在遍历一个结点后，使用函数递归遍历其邻接表内结点。

广度优先遍历：使用map记录结点是否被遍历，使用队列记录未被遍历的结点，在遍历一个节点后，将其邻接表内未便利的结点指针放入队列，直到队列为空。

## 4.3 系统实现

4.3.1代码实现

代码见附录或者附录文件

4.3.2程序测试

程序测试：

为了防止数据输入错误我选择了使用文件初始化图，图的结构选自教学ppt上的一个图，如图4-1：

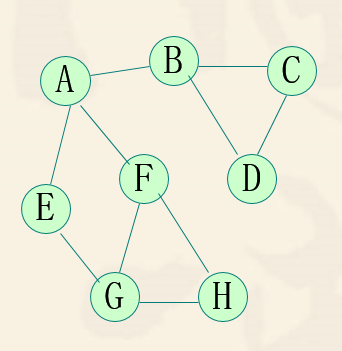


图4-1

使用数据输入的初始化结果如图4-2：

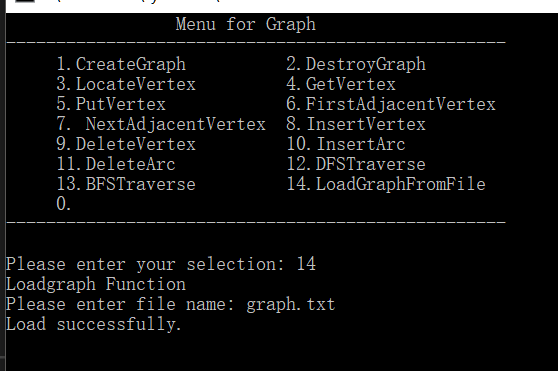


图4-2

可见初始化图成功，接下来测试其中的函数，首先测试定位顶点函数测试结果如图4-3所示：

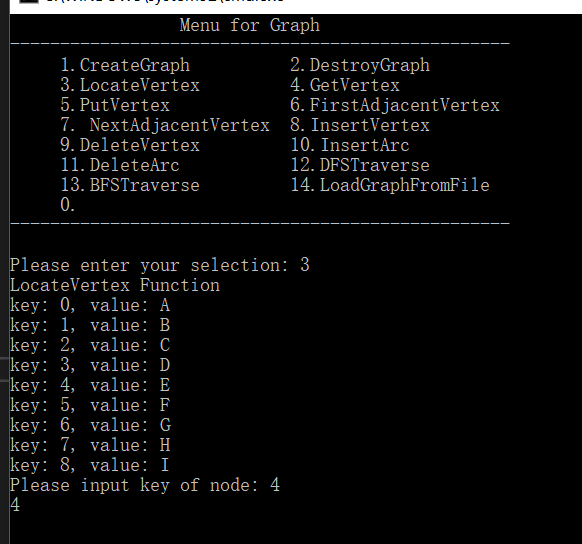


图4-3

测试获取结点值的函数如图4-4所示：

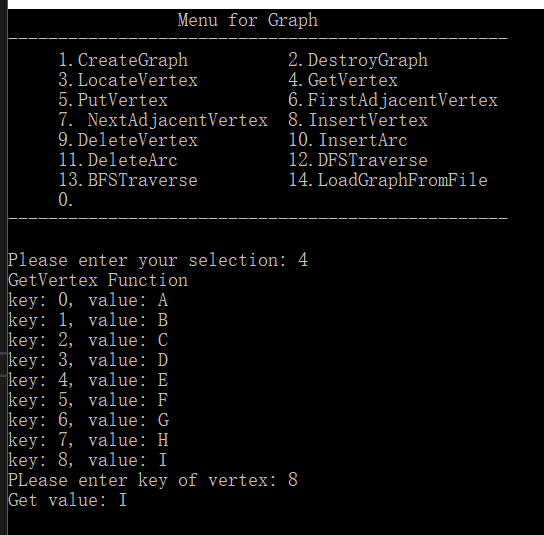


图4-4

测试结点赋值函数，将A所在结点的值设置为J，结果如图4-5和图4-6所示：

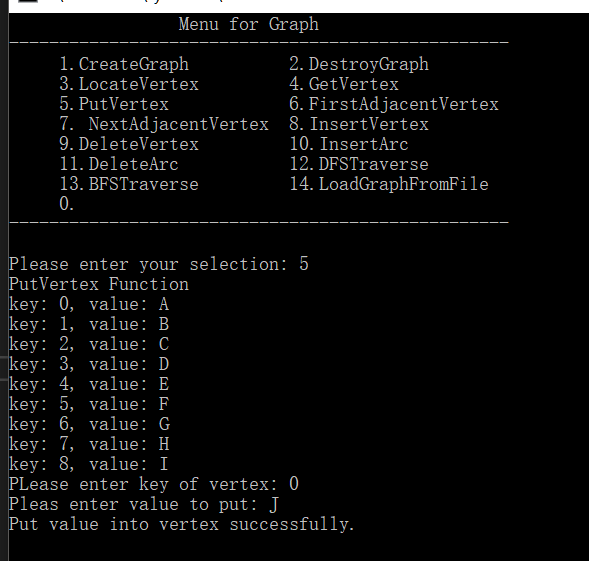


图4-5

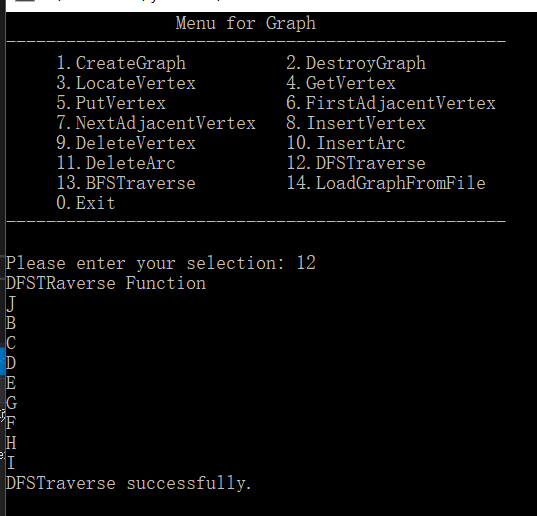


图4-6

根据遍历结果认为赋值没有问题。接下来测试获取第一个邻接点的函数，先显示一下当前的邻接表的结点分布，如图4-7所示：

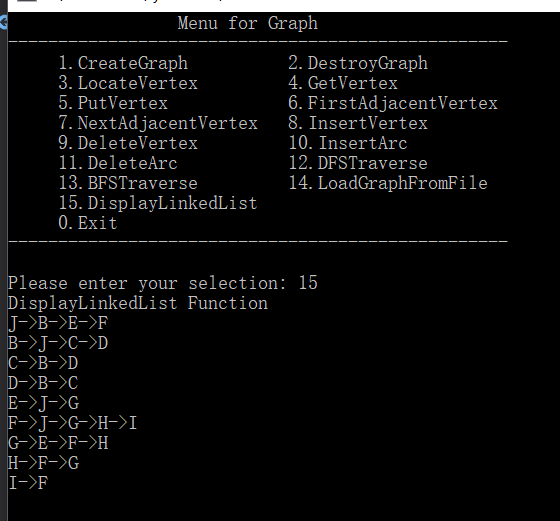


图4-7

获取结点G的第一邻接点，测试如图4-8所示：

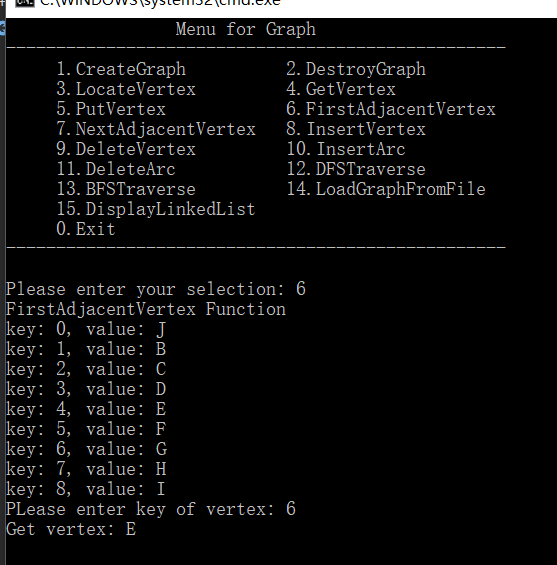


图4-8

获取下一邻接点测试如图4-9所示，测试获取顶点G的F后的下一邻接点，根据邻接表的展示结果推断函数结果应该是H：

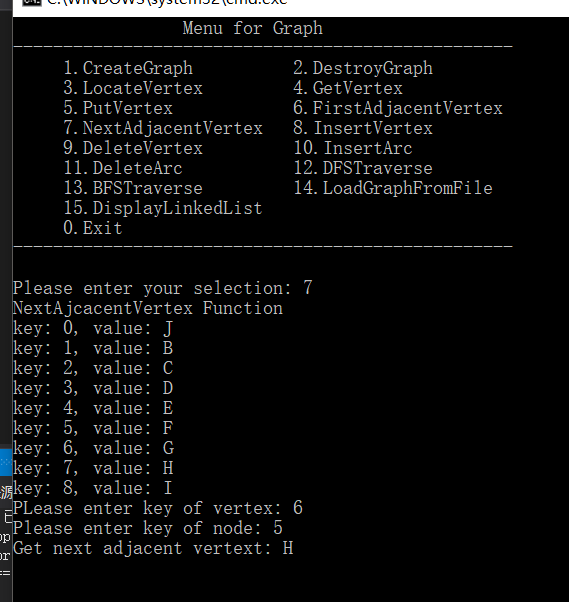


图4-9

可知获取下一邻接点成功。测试插入结点函数，插入K，测试结果如图4-10所示：

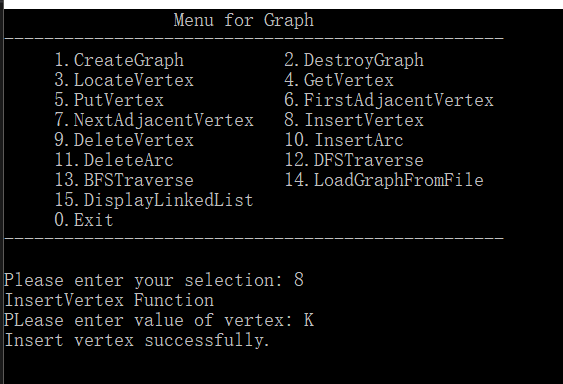


图4-10

遍历邻接表如图4-11所示：

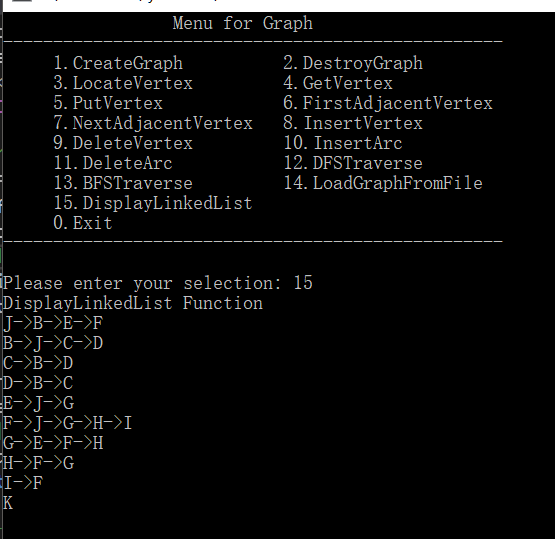


图4-11

可知插入成功，接下来删除结点测试，删除结点E测试，测试前后图的邻接表如图4-12，4-13，4-14所示：

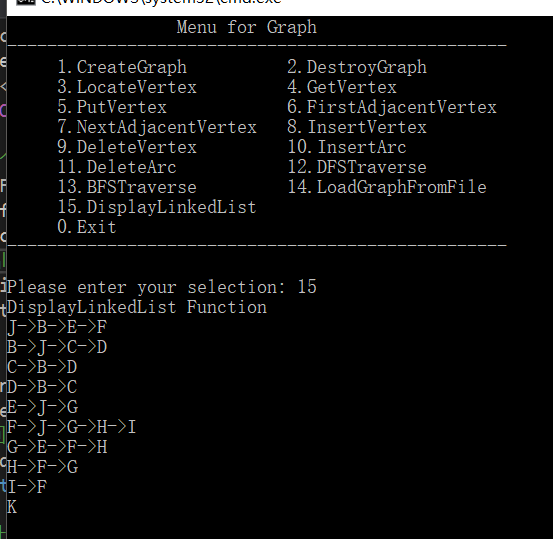


图4-12

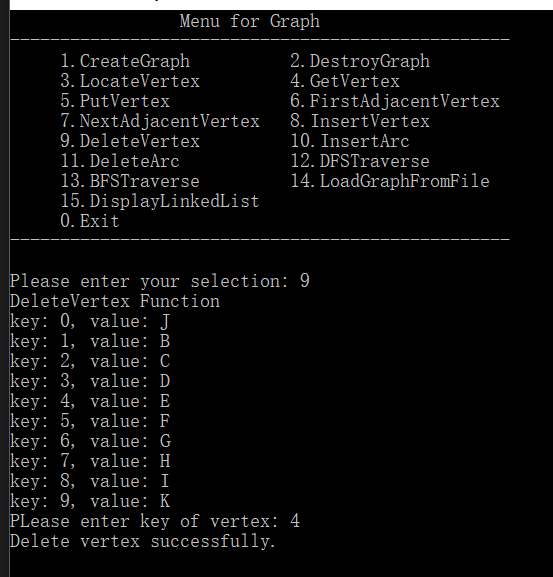


图4-14

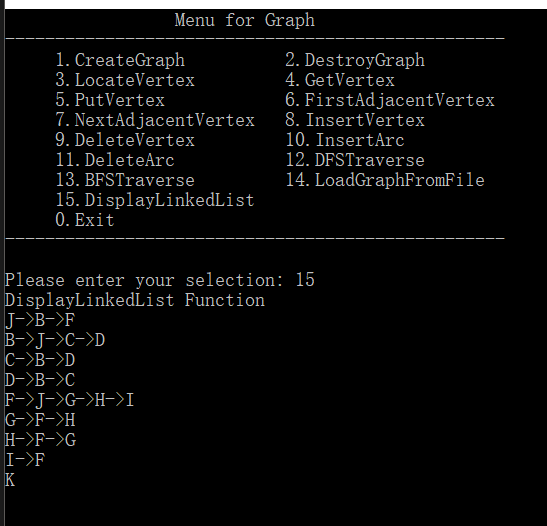


图4-15

可知删除成功。接下来测试插入弧的测试，插入B到K的弧，将K接入图，测试结果如图4-16，4-17：

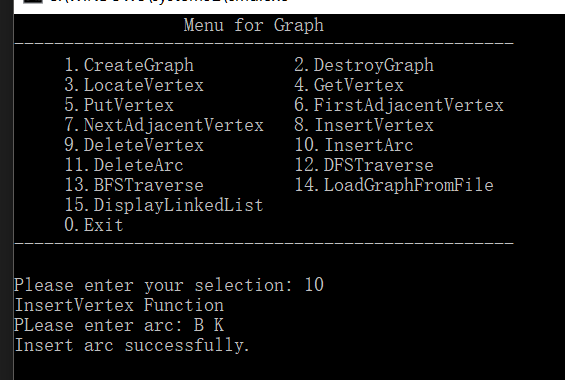


图4-16

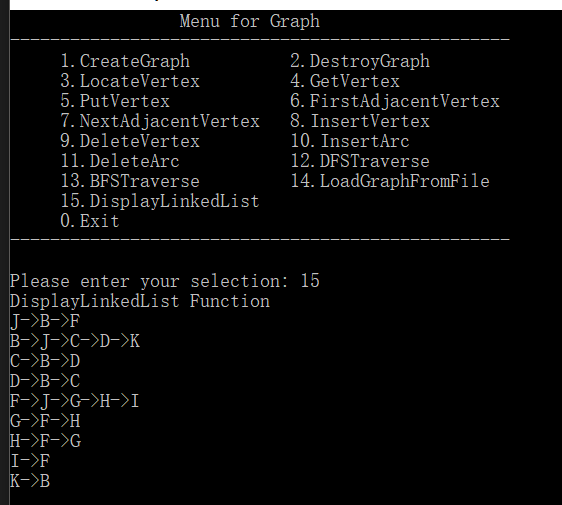


图4-17

可见B与K之间已经建立联系，插入成功。测试删除弧，删除弧HG，测试结果如图4-18，图4-19：

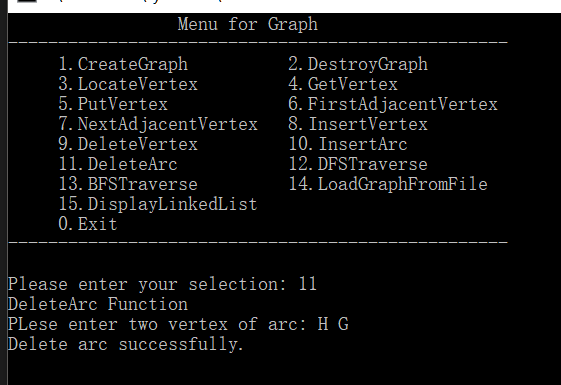


图4-18

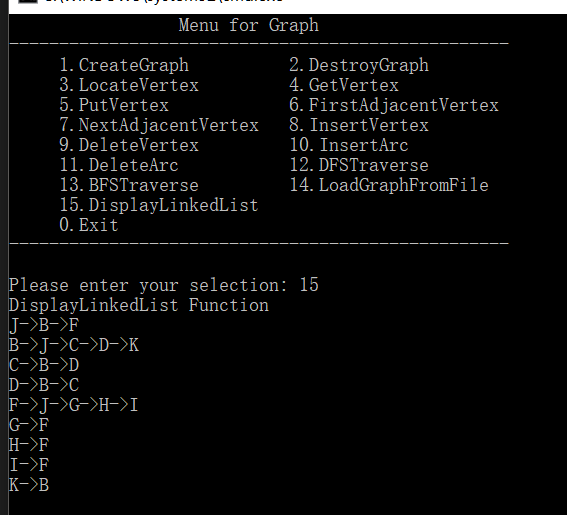


图4-19

可知删除成功，接下来测试深度遍历和广度遍历，由于图已经被修改太多，我选择使用最初的图进行遍历，这样比较容易判断是否正确，结果如图4-20和图4-21：

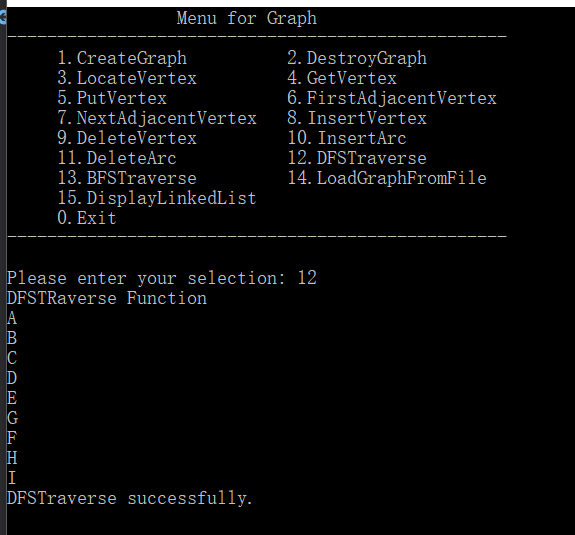


图4-20

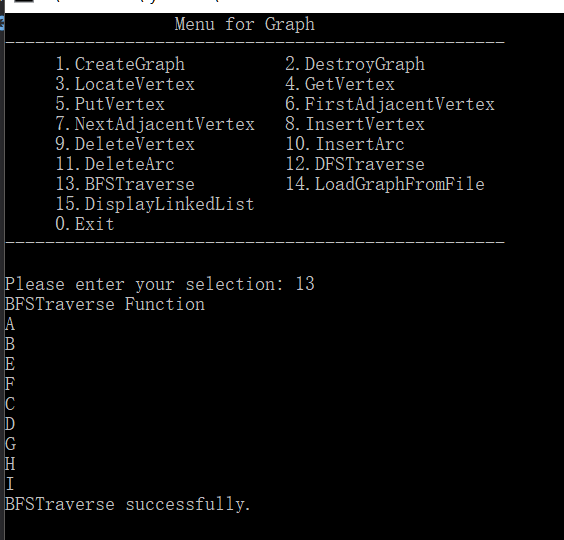


图4-21

对比ppt上的遍历结果可知遍历正确。

## 4.4 实验小结

本次实验还是比较难的，由于我是使用邻接表存储的图，所以也是相当于复习了一下链表的内容，在实验中犯了一些使用常见的错误，幸好在运行时崩溃了才发现，就及时更正了，其中主要的几个错误主要有：

1.链表使用时，忘记将指针向后移动到下一个指针，造成程序无响应，无限循环。

2.深度遍历忘记对所有的结点使用DFS和BFS，没有考虑到图有可能不为连通图的情况，造成了遍历不完全的情况。

# 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

## 文件linearList.h内容

#ifndef \_LINEARLIST\_H

#define \_LINEARLIST\_H

#include *"status.h"*

*// 初始长度*

#define LIST\_INIT\_SIZE 100

*// 增长速率*

#define LISTINCREMENT 10

*// 定义模板参数*

**template**<**typename** T>

*// 类定义*

**class** **LinearList** {

*// 私有变量及函数部分*

**private**:

*// 数组的指针，初始化为nullptr*

T \* elem = **nullptr**;

*// 表的长度，初始化为0*

int length = 0;

*// 表的容量，初始化为0*

int listSize = 0;

*// 扩大链表的函数，只允许内部调用，设置访问权限为private*

void largerList() {

T \*temp = **nullptr**;

**try** {

*// 尝试开辟空间*

temp = **new** T[listSize + LISTINCREMENT];

}*// 失败抛出异常，异常处理*

**catch** (**const** std::bad\_alloc &e) {

*// 输出错误信息*

std::cerr << "ERROR: " << e.what() << std::endl;

*// 程序系统面临崩溃，及时退出*

exit(EXIT\_FAILURE);

}

*// 否则表示空间分配完成，将原来的数据复制到新的地方*

**for** (int i = 0; i < length; ++i) {

temp[i] = std::move(elem[i]);

}

*// 释放原来的空间*

**delete**[] elem;

*// 移交指针*

elem = temp;

*// 容量增长*

listSize += LISTINCREMENT;

}

**public**:

*// 公开函数部分*

*// 初始化函数*

status InitaList() {

*// 已经进行了初始化*

**if** (listSize != 0) {

std::cerr << "线性表已存在，初始化失败，请先进行销毁操作。**\n**" << std::endl;

**return** ERROR;

}

*// 初始化需要分配空间*

**try** {

*// new空间*

elem = **new** T[LIST\_INIT\_SIZE];

}

**catch** (**const** std::bad\_alloc& e) {

*// 分配失败*

std::cerr << "Error: " << e.what() << std::endl;

*// 返回错误*

**return** ERROR;

}

*// 表的容量即为LIST\_INIT\_SIZE*

listSize = LIST\_INIT\_SIZE;

*// 长度设为0*

length = 0;

char ch;

*// 询问是否从文件初始化*

std::cout << "if initalize from file(y/n): ";

std::cin >> ch;

getchar();

**if** (ch == 'y') {

*// 从文件初始化需要输入文件名*

std::cout << "Please enter file name: ";

std::string file\_name;

*// 得到文件名*

std::getline(std::cin, file\_name);

*// 打开输入流*

std::fstream fin(file\_name, std::ios\_base::in);

*// 判断是否打开完成*

**if** (!fin.is\_open()) {

std::cerr << "Open file failed.**\n**";

**return** ERROR;

}

**else** {

int cnt = 0;

*// 输入文件中的元素数量*

fin >> cnt;

*// 局部元素*

T ele;

*// 进行插入操作*

**for** (int i = 0; i < cnt; ++i) {

fin >> ele;

ListInsert(length + 1, ele);

}

std::cout << "Initalize from file successfully.**\n**";

}

*// 关闭文件流*

fin.close();

}

**return** OK;

}

*// 销毁线性表*

status DestroyList() {

*// 如果未进行初始化*

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 清理空间*

**delete**[] elem;

*// 置为空指针*

elem = **nullptr**;

*// 长度和容量置为0*

length = 0;

listSize = 0;

**return** OK;

}

*// 清空线性表*

status ClearList() {

*// 未初始化*

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 清空只需重置表长即可*

length = 0;

**return** OK;

}

*// 判断表是否为空*

bool ListEmpty() {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** true;

}

*// 判断表长是否为0即可*

**return** length == 0;

}

*// 获取表长*

int ListLength() {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** -1;

}

*// 返回表长*

**return** length;

}

*// 获取下表为index处的元素*

status GetElem(**const** int &index, T &e) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 判断下表合法性*

**if** (index > length || index < 1) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 直接获取*

e = elem[index - 1];

**return** OK;

}

*// 定位元素*

status LocateElem(**const** T &e, int &index, Compare<T> comp) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 下标初始化*

index = 0;

*// 遍历*

**for** (int i = 0; i < length; ++i) {

**if** (comp(e, elem[i])) {

index = i + 1;

**return** OK;

}

}

**return** ERROR;

}

*// 获取元素前驱*

status PriorElem(**const** T &cur, T &pre\_e, Compare<T> comp) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 遍历即可，注意范围*

**for** (int i = 0; i < length - 1; ++i) {

**if** (comp(cur, elem[i + 1])) {

pre\_e = elem[i];

**return** OK;

}

}

**return** ERROR;

}

*// 获取元素后驱*

status NextElem(**const** T &cur, T &next, Compare<T> comp) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 和前驱部分相同，遍历即可*

**for** (int i = 1; i < length; ++i) {

**if** (comp(cur, elem[i - 1])) {

next = elem[i];

**return** OK;

}

}

**return** ERROR;

}

*// 插入元素*

status ListInsert(**const** int &index, **const** T &e) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 判断插入位置合法性*

**if** (index < 1 || index > length + 1) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 如果表已经满了，则需要扩容*

**if** (length >= listSize) {

*// 进行扩容*

largerList();

}

*// 将对应元素依次向后移动*

**for** (int i = length; i >= index; --i) {

elem[i] = elem[i - 1];

}

*// 插入元素*

elem[index - 1] = e;

*// 长度要加1*

++length;

**return** OK;

}

*// 删除元素*

status ListDelete(**const** int &index, T &e) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 判断下标合法性*

**if** (index < 1 || index > length) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 直接返回即可*

e = elem[index - 1];

*// 将后面的函数向前移*

**for** (int i = index - 1; i < length - 1; ++i) {

elem[i] = elem[i + 1];

}

*// 长度减1*

--length;

**return** OK;

}

*// 遍历线性表*

status ListTraverse(Visit<T> visit) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 依次遍历即可*

**for** (int i = 0; i < length; ++i) {

visit(elem[i]);

}

**return** OK;

}

*// 保存到文件*

status ListSave() {

**if** (elem == **nullptr**) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

getchar();

*// 保存文件需要输入文件名*

std::cout << "Please enter file name: ";

std::string file\_name;

*// 输入文件名*

std::getline(std::cin, file\_name);

*// 打开文件流*

std::fstream fout(file\_name, std::ios\_base::out);

*// 判断打开是否成功*

**if** (fout.is\_open()) {

std::cout << "Open file successfully.**\n**";

*// 先将长度写入到文件中*

fout << length << "**\n**";

*// 将元素写入*

**for** (int i = 0; i < length; ++i) {

fout << elem[i] << "**\n**";

}

*// 关闭文件流*

fout.close();

std::cout << "Save successfully.**\n**";

**return** OK;

}

**else** {

std::cerr << "Open file failed.**\n**";

**return** ERROR;

}

}

*// 析构函数，释放空间即可*

~LinearList() {

**delete** elem;

}

};

#endif *//\_LINEARLIST\_H*

## 文件testLinearList.h内容：

#ifndef DATASTRUCTURE\_TESTLINEARLIST\_H

#define DATASTRUCTURE\_TESTLINEARLIST\_H

#include *"linearList.h"*

#include *"status.h"*

#define LEN 100

void testLinearList() {

int temp = 0;

long op = 1;

int cur = 0, next = 0, pre = 0, elem = 0;

int list\_index = 0;

int temp\_index = 0;

int index;

LinearList<int> lists[LEN];

Compare<int> comp = [](**const** int &a, **const** int &b) -> bool { **return** a == b; };

Visit<int> visit = [](**const** int &a) { std::cout << a << std::endl; };

**while** (op != 0) {

cls();

std::cout << "**\n\n**";

std::cout << " Menu for Linked Table On Chain Structure **\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " 1. IntiaList 7. LocateElem**\n**";

std::cout << " 2. DestroyList 8. PriorElem**\n**";

std::cout << " 3. ClearList 9. NextElem **\n**";

std::cout << " 4. ListEmpty 10.ListInsert**\n**";

std::cout << " 5. ListLength 11.ListDelete**\n**";

std::cout << " 6. GetElem 12.ListTraverse**\n**";

std::cout << " 13.Change list 14.Save List**\n**";

std::cout << " 0. Exit**\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " list using: " << list\_index << "**\n**";

std::cout << " please select your operation[0~14]:";

std::cin >> op;

**auto** &L = lists[list\_index];

**switch** (op) {

**case** 1:

std::cout << "InitaList Function**\n**";

**if** (L.InitaList() == OK) std::cout << "Initialze successful.**\n**";

**else** std::cout << "Initailze failed.**\n**";

getchar();

getchar();

**break**;

**case** 2:

std::cout << "**\n**DestroyList Function**\n**";

**if** (L.DestroyList() == OK) std::cout << "Destroy successfully.**\n**";

**else** std::cout << "Destroy failed.**\n**";

getchar();

getchar();

**break**;

**case** 3:

std::cout << "**\n**ClearList Function**\n**";

**if** (L.ClearList() == OK) std::cout << "Clear successfully**\n**";

**else** std::cout << "CLear failed.**\n**";

getchar();

getchar();

**break**;

**case** 4:

std::cout << "**\n**ListEmpty Function**\n**";

**if** (L.ListEmpty()) {

std::cout << "The list is empty.**\n**";

}

**else** {

std::cout << "The list is not empty.**\n**";

}

getchar();

getchar();

**break**;

**case** 5:

std::cout << "**\n**ListLength Function**\n**";

std::cout << "The length of list: " << L.ListLength() << std::endl;

getchar();

getchar();

**break**;

**case** 6:

std::cout << "**\n**GetElem Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.GetElem(index, elem) == OK) {

std::cout << "Get the element: " << elem << "**\n**";

}

**else** std::cout << "Get element failed.**\n**";

getchar();

getchar();

**break**;

**case** 7:

std::cout << "**\n**LocateElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

**if** (L.LocateElem(elem, index, comp) == OK) {

std::cout << "The index of element:" << index << std::endl;

}

**else** {

std::cout << "Locate element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 8:

std::cout << "**\n**PriorElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> cur;

**if** (L.PriorElem(cur, pre, comp) == OK) {

std::cout << "Get the result: " << pre << "**\n**";

}

**else** {

std::cout << "Get the element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 9:

std::cout << "**\n**NextElem Function**\n**";

std::cout << "PLease enter the element: ";

std::cin >> cur;

**if** (L.NextElem(cur, next, comp) == OK) {

std::cout << "Get the element: " << next << "**\n**";

}

**else** {

std::cout << "Operate failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 10:

std::cout << "**\n**ListInsert Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

std::cout << "Please enter the index: ";

scanf\_s("%d", &index);

**if** (L.ListInsert(index, elem) == OK) {

std::cout << "Insert successfully.**\n**";

}

**else** {

std::cout << "Insert failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 11:

std::cout << "**\n**ListDelete Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.ListDelete(index, elem) == OK) {

std::cout << "Delete " << elem << " successfullyy.**\n**";

}

**else** {

std::cout << "Delete failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 12:

std::cout << "**\n**ListTraverse Function**\n**";

**if** (L.ListTraverse(visit)) {

std::cout << "Traverse successfully.**\n**";

}

**else** {

std::cout << "Traverse failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 13:

std::cout << "Enter the index of list:[0-99] ";

std::cin >> temp\_index;

**if** (temp\_index >= LEN || temp\_index < 0) {

std::cout << "Change failed**\n**";

}

**else** {

list\_index = temp\_index;

std::cout << "Change successfully**\n**";

}

getchar();

getchar();

**break**;

**case** 14:

**if** (L.ListSave() == OK) {

std::cout << "Save to file successfully**\n**";

}

**else** {

std::cout << "Save to file failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 0:

**break**;

**default**:

**break**;

}*//end of switch*

}*//end of while*

std::cout << "Good bye.**\n**";

}

#endif *//DATASTRUCTURE\_TESTLINEARLIST\_H*

# 附录B 基于链式存储结构线性表实现的源程序

## 文件linkedList.h内容：

#ifndef DATASTRUCTURE\_LINKEDLIST\_H

#define DATASTRUCTURE\_LINKEDLIST\_H

*// 包含所需头文件*

#include *"status.h"*

*// 使用模板*

**template**<**typename** T = int>

*// 定义类*

**class** **LinkedList** {

**private**:

*// 结点类型*

**struct** Node {

*// 值*

T value;

*// 指向下一个文件的指针*

Node \*next;

*// 删除默认构造函数*

Node() = **delete**;

*// 定义构造函数*

Node(**const** T &value) : value(value), next(**nullptr**) {}

};

*// 定义头指针*

Node \*head = **nullptr**;

*// 链表长度*

int length = 0;

*// 是否进行了初始化*

bool initaled = false;

**public**:

*// 初始化函数*

status InitaList() {

*// 已经初始化过的返回ERROR*

**if** (initaled) {

*// 输出错误信息*

std::cerr << "The list had been initalized.**\n**";

**return** ERROR;

}

*// 长度初始化为0*

length = 0;

*// 询问是否从文件初始化*

std::cout << "initalize from file(y/n):";

char ch;

*// 输入选择*

std::cin >> ch;

**if** (ch == 'y') {

getchar();

*// 得到文件名*

std::cout << "Please enter name of the file: ";

std::string file\_name;

std::getline(std::cin, file\_name);

*// 打开文件流*

std::fstream fin(file\_name, std::ios\_base::in);

*// 查看打开是否成功*

**if** (fin.is\_open()) {

*// 进行输入数据*

initaled = true;

int cnt = 0;

*// 得到数据个数*

fin >> cnt;

*// 定义局部变量*

T temp;

**for** (int i = 0; i < cnt; ++i) {

*// 输入变量*

fin >> temp;

*// 插入到链表后面*

ListInsert(length + 1, temp);

}

*// 关闭文件流*

fin.close();

std::cout << "Initalize from file successfully.**\n**";

}

**else** {

*// 否则输出错误*

std::cerr << "Open file failed.**\n**";

**return** ERROR;

}

}

initaled = true;

**return** OK;

}

*// 销毁链表*

status DestroyList() {

*// 如果未初始化返回错误*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

Node \*temp = head;

Node \*pre;

*// 清理所有链表空间*

**while** (temp != **nullptr**) {

pre = temp->next;

**delete** temp;

temp = pre;

}

*// 头指针置为空*

head = **nullptr**;

*// 长度置零*

length = 0;

*// 初始化置为false*

initaled = false;

**return** OK;

}

*// 清空链表*

status ClearList() {

*// 未初始化不能清空*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 唯一区别就是摧毁后的初始化状态为true*

DestroyList();

initaled = true;

**return** OK;

}

*// 检测是否为空表*

bool ListEmpty() **const** {

*// 未初始化的默认是空表*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** true;

}

*// 判断长度是否为0*

**return** length == 0;

}

*// 得到链表长度*

int ListLength() **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** -1;

}

*// 返回长度*

**return** length;

}

*// 获取特定位置的元素*

status GetElem(**const** int &index, T &e) **const** {

*// 未初始化报错*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 判断下标是否合法*

**if** (index < 0 || index > length) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 定义局部变量*

**auto** temp = head;

*// index处元素向后移动index-1次即可*

**for** (int i = 0; i < index - 1; ++i) {

temp = temp->next;

}

*// 获取值*

e = temp->value;

**return** OK;

}

*// 定位元素*

status LocateElem(**const** T &e, int &index, Compare<T> comp) **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 局部头指针*

**auto** temp\_ptr = head;

*// 遍历元素，看是否有comp为真的*

**for** (int i = 0; i < length; i++) {

**if** (comp(e, temp\_ptr->value)) {

*// 存在就返回下标*

index = i + 1;

**return** OK;

}

*// 指针后移*

temp\_ptr = temp\_ptr->next;

}

*// 否则就是没有*

index = 0;

**return** ERROR;

}

*// 获取前驱元素*

status PriorElem(**const** T &cur, T &pre\_e, Compare<T> comp) **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 局部头指针*

**auto** temp\_ptr = head;

*// 遍历元素*

**while** (temp\_ptr != **nullptr** && temp\_ptr->next != **nullptr**) {

*// 如果此处元素的后驱是给出的元素，返回此元素*

**if** (comp(temp\_ptr->next->value, cur)) {

pre\_e = temp\_ptr->value;

**return** OK;

}

temp\_ptr = temp\_ptr->next;

}

**return** ERROR;

}

*// 获得后驱*

status NextElem(**const** T &cur, T &next, Compare<T> comp) **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

**auto** temp\_ptr = head;

*// 如果此元素满足comp为真，下一个即为所求元素*

**while** (temp\_ptr != **nullptr** && temp\_ptr->next != **nullptr**) {

**if** (comp(temp\_ptr->value, cur)) {

next = temp\_ptr->next->value;

**return** OK;

}

temp\_ptr = temp\_ptr->next;

}

**return** ERROR;

}

*// 插入元素*

status ListInsert(**const** int &index, **const** T &e) {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 判断下标是否合法*

**if** (index < 1 || index > length + 1) {

std::cerr << "Index out of bounds**\n**";

**return** ERROR;

}

*// 使用初始值新建结点*

**auto** t = **new** Node(e);

*// 插入到首元素位置特殊处理*

**if** (index == 1) {

t->next = head;

head = t;

}

**else** {

*// 向后移动到适当位置*

**auto** pre = head;

**for** (int i = 0; i < index - 2; ++i) {

pre = pre->next;

}

*// 将t插入*

t->next = pre->next;

pre->next = t;

}

*// 长度加1*

++length;

**return** OK;

}

*// 删除元素*

status ListDelete(**const** int &index, T &e) {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 判断下标合法性*

**if** (index < 1 || index > length) {

std::cerr << "Index out of bounds**\n**";

**return** ERROR;

}

*// 删除首元素特殊处理*

**if** (index == 1) {

Node \*temp = head;

head = head->next;

e = temp->value;

**delete** temp;

}

**else** {

**auto** pre = head;

*// 后移适当位置，找到要删除元素的前驱*

**for** (int i = 0; i < index - 2; ++i) {

pre = pre->next;

}

*// 删除元素*

**auto** to\_del = pre->next;

pre->next = to\_del->next;

e = to\_del->value;

**delete** to\_del;

}

*// 长度减 1*

--length;

**return** OK;

}

*// 遍历元素*

status ListTraverse(Visit<T> visit) {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 直接遍历即可*

**auto** temp\_ptr = head;

**for** (int i = 0; i < length; ++i) {

*// 使用visit指针*

visit(temp\_ptr->value);

*// 指针后移*

temp\_ptr = temp\_ptr->next;

}

**return** OK;

}

*// 将链表保存到文件中*

status ListSave() {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

getchar();

*// 得到文件名*

std::cout << "Please enter the file name: ";

std::string file\_name;

std::cin >> file\_name;

*// 打开输出文件流*

std::fstream fout(file\_name, std::ios\_base::out);

*// 先写入元素个数*

fout << length << "**\n**";

*// 遍历写入*

**auto** temp\_ptr = head;

**for** (int i = 0; i < length; ++i) {

fout << temp\_ptr->value << "**\n**";

temp\_ptr = temp\_ptr->next;

}

*// 关闭文件流保存文件*

fout.close();

**return** OK;

}

*// 析构函数*

~LinkedList() {

*// 主要就是释放空间了*

**auto** temp = head;

**while** (temp != **nullptr**) {

**auto** next = temp->next;

**delete** temp;

temp = next;

}

head = **nullptr**;

}

};

#endif *//DATASTRUCTURE\_LINKEDLIST\_H*

## 文件testLinkedList.h内容：

#ifndef DATASTRUCTURE\_TESTLINKEDLIST\_H

#define DATASTRUCTURE\_TESTLINKEDLIST\_H

#define LEN 100

#include *"linkedList.h"*

void testLinkedList() {

int list\_index = 0;

long op = 1;

int cur = 0, next = 0, pre = 0, elem = 0;

int l\_index = 0;

int index;

LinkedList<int> lists[LEN];

Compare<int> comp = [](**const** int &a, **const** int &b) -> bool { **return** a == b; };

Visit<int> visit = [](**const** int &a) { std::cout << a << std::endl; };

**while** (op != 0) {

cls();

std::cout << "**\n\n**";

std::cout << " Menu for Linked Table On Chain Structure **\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " 1. IntiaList 7. LocateElem**\n**";

std::cout << " 2. DestroyList 8. PriorElem**\n**";

std::cout << " 3. ClearList 9. NextElem **\n**";

std::cout << " 4. ListEmpty 10.ListInsert**\n**";

std::cout << " 5. ListLength 11.ListDelete**\n**";

std::cout << " 6. GetElem 12.ListTraverse**\n**";

std::cout << " 13.Change list 14.Save List**\n**";

std::cout << " 0. Exit**\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " list using: " << l\_index << "**\n**";

std::cout << " please select your operation[0~14]:";

std::cin >> op;

**auto** &L = lists[l\_index];

**switch** (op) {

**case** 1:

std::cout << "InitaList Function**\n**";

**if** (L.InitaList() == OK) std::cout << "Initialze successful.**\n**";

**else** std::cout << "Initailze failed.**\n**";

getchar();

getchar();

**break**;

**case** 2:

std::cout << "**\n**DestroyList Function**\n**";

**if** (L.DestroyList() == OK) std::cout << "Destroy successfully.**\n**";

**else** std::cout << "Destroy failed.**\n**";

getchar();

getchar();

**break**;

**case** 3:

std::cout << "**\n**ClearList Function**\n**";

**if** (L.ClearList() == OK) std::cout << "Clear successfully**\n**";

**else** std::cout << "CLear failed.**\n**";

getchar();

getchar();

**break**;

**case** 4:

std::cout << "**\n**ListEmpty Function**\n**";

**if** (L.ListEmpty()) {

std::cout << "The list is empty.**\n**";

}

**else** {

std::cout << "The list is not empty.**\n**";

}

getchar();

getchar();

**break**;

**case** 5:

std::cout << "**\n**ListLength Function**\n**";

std::cout << "The length of list: " << L.ListLength() << std::endl;

getchar();

getchar();

**break**;

**case** 6:

std::cout << "**\n**GetElem Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.GetElem(index, elem) == OK) {

std::cout << "Get the element: " << elem << "**\n**";

}

**else** std::cout << "Get element failed.**\n**";

getchar();

getchar();

**break**;

**case** 7:

std::cout << "**\n**LocateElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

**if** (L.LocateElem(elem, index, comp) == OK) {

std::cout << "The index of element:" << index << std::endl;

}

**else** {

std::cout << "Locate element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 8:

std::cout << "**\n**PriorElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> cur;

**if** (L.PriorElem(cur, pre, comp) == OK) {

std::cout << "Get the result: " << pre << "**\n**";

}

**else** {

std::cout << "Get the element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 9:

std::cout << "**\n**NextElem Function**\n**";

std::cout << "PLease enter the element: ";

std::cin >> cur;

**if** (L.NextElem(cur, next, comp) == OK) {

std::cout << "Get the element: " << next << "**\n**";

}

**else** {

std::cout << "Operate failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 10:

std::cout << "**\n**ListInsert Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

std::cout << "Please enter the index: ";

scanf\_s("%d", &index);

**if** (L.ListInsert(index, elem) == OK) {

std::cout << "Insert successfully.**\n**";

}

**else** {

std::cout << "Insert failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 11:

std::cout << "**\n**ListDelete Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.ListDelete(index, elem) == OK) {

std::cout << "Delete " << elem << " successfullyy.**\n**";

}

**else** {

std::cout << "Delete failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 12:

std::cout << "**\n**ListTraverse Function**\n**";

**if** (L.ListTraverse(visit)) {

std::cout << "Traverse successfully.**\n**";

}

**else** {

std::cout << "Traverse failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 13:

std::cout << "Enter the index of list:[0-99] ";

std::cin >> list\_index;

**if** (list\_index >= LEN || list\_index < 0) {

std::cout << "Change failed**\n**";

}

**else** {

l\_index = list\_index;

std::cout << "Change successfully**\n**";

}

getchar();

getchar();

**break**;

**case** 14:

**if** (L.ListSave() == OK) {

std::cout << "Save to file successfully**\n**";

}

**else** {

std::cout << "Save to file failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 0:

**break**;

**default**:

**break**;

}*//end of switch*

}*//end of while*

std::cout << "Good bye.**\n**";

}

#endif *//DATASTRUCTURE\_TESTLINKEDLIST\_H*

# 附录C 基于二叉链表二叉树实现的源程序

## 文件binaryTree.h内容：

#ifndef DATASTRUCTURE\_BINARY\_TREE\_H

#define DATASTRUCTURE\_BINARY\_TREE\_H

#include *"status.h"*

*// 结点定义*

**template**<**typename** ValueType>

**struct** Node {

*// 键域，方便二叉树使用*

int key;

*// 值域*

ValueType value;

*// 左孩子指针*

Node<ValueType> \*left = **nullptr**;

*// 右孩子指针*

Node<ValueType> \*right = **nullptr**;

*// 父节点指针*

Node<ValueType> \*parent;

*// 禁止默认构造函数，防止父指针未被初始化*

Node() = **delete**;

*// 唯一的构造函数*

Node(**const** int key, **const** ValueType &value,

Node<ValueType> \*&parent)

:value(value), parent(parent), key(key) {}

};

*// 二叉树定义，需要传入存储的值类型和空值的表示*

**template**<**typename** ValueType, **const** ValueType NULL\_VALUE>

**class** **BinaryTree** {

**private**:

**using** NodeType = Node<ValueType>;

*// 根结点定义*

NodeType \*head = **nullptr**;

*// 标记树是否被初始化*

bool initalized = false;

*// key是键，内部为结点唯一分配，相当于ID*

int key = 0;

*// 释放二叉树空间*

void freeTree(NodeType \*&node) {

**if** (node != **nullptr**) {

*// 先递归调用释放子树*

freeTree(node->left);

freeTree(node->right);

*// 释放当前结点*

**delete** node;

*// 置为null*

node = **nullptr**;

}

}

*// 重载函数，获取树的深度*

int BiTreeDepth(**const** NodeType \*node) {

*// 当前结点不为空才继续*

**if** (node != **nullptr**) {

*// 获取左右结点最深的，加一返回*

int left = BiTreeDepth(node->left);

int right = BiTreeDepth(node->right);

**return** std::max(left, right) + 1;

}

*// 递归终止条件*

*// 为空结束返回0*

**return** 0;

}

*// 重载前序遍历*

void PreOrderTraverse(**const** NodeType \*node,

**const** Visit<ValueType> &visit) {

*// 结点为空为递归终止条件*

**if** (node != **nullptr**) {

*// 前序遍历先遍历当前结点*

visit(node->value);

*// 依次左右结点*

PreOrderTraverse(node->left, visit);

PreOrderTraverse(node->right, visit);

}

}

*// 重载中序遍历*

void InOrderTraverse(**const** NodeType \*node,

**const** Visit<ValueType> &visit) {

*// 递归终止同前序遍历*

**if** (node != **nullptr**) {

*// 左中右顺序遍历*

InOrderTraverse(node->left, visit);

visit(node->value);

InOrderTraverse(node->right, visit);

}

}

*// 重载后序遍历，详细内容同上*

void PostOrderTraverse(**const** NodeType \*node,

**const** Visit<ValueType> &visit) {

**if** (node != **nullptr**) {

*// 顺序左右中*

PostOrderTraverse(node->left, visit);

PostOrderTraverse(node->right, visit);

visit(node->value);

}

}

*// 根据key寻找结点，返回值为找到的结点指针或者nullptr*

NodeType \*find\_by\_key(NodeType \*node, **const** int key) {

*// 同样递归终止条件为指针参数为空*

**if** (node != **nullptr**) {

*// 找到就返回指针*

**if** (node->key == key) {

**return** node;

}

*// 向左边寻找*

**auto** left = find\_by\_key(node->left, key);

**if** (left != **nullptr**) {

**return** left;

}

*// 向右边寻找*

**auto** right = find\_by\_key(node->right, key);

**if** (right != **nullptr**) {

**return** right;

}

}

*// 默认返回nullptr*

**return** **nullptr**;

}

*// 对建树进行重载的函数*

void CreateBiTree(NodeType \*&node, NodeType \*parent,

std::vector<ValueType> values, size\_t &index) {

*// 当前下标参数未超出vector范围并且此处值不为空*

**if** (index < values.size() && values[index] != NULL\_VALUE) {

*// 将node赋值为新开辟的结点，key进行自增*

node = **new** NodeType(key++, values[index], parent);

*// 递归进行左右构建*

CreateBiTree(node->left, node, values, ++index);

CreateBiTree(node->right, node, values, ++index);

}

}

*// 展示现有树的结点，输出key及对应的值*

void display(**const** NodeType \*node) {

**if** (node != **nullptr**) {

std::cout << "key: " << node->key << ", value: " << node->value << "**\n**";

*// 递归对左右进行调用*

display(node->left);

display(node->right);

}

}

*// 重载插入子树操作*

void InsertChildren(NodeType \*&node, **const** NodeType \*tree\_node, **const** LR &lr) {

*// 此操作将给出的子树进行整棵复制而不是直接接入，防止外部释放子树后出现问题*

*// 参数为要进行赋值的结点引用*

**if** (tree\_node != **nullptr**) {

*// 获取插入位置是左还是右*

**auto** &chi\_node = lr == LR::L ? node->**left** : node->right;

*// 对获取到的结点赋值*

chi\_node = **new** NodeType(key++, tree\_node->value, node);

*// 递归复制*

InsertChildren(chi\_node, tree\_node->left, LR::L);

InsertChildren(chi\_node, tree\_node->right, LR::R);

}

}

**public**:

*// 初始化树的函数*

status InitBiTree() {

*// 首先判断树是否已被初始化*

**if** (initalized) {

std::cerr << "The binary tree has been initalized.**\n**";

**return** ERROR;

}

*// 将初始化标志设置为true*

initalized = true;

**return** OK;

}

*// 销毁树的函数*

status DestroyBiTree() {

*// 常规检测*

**if** (initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** ERROR;

}

*// 释放空间*

freeTree(head);

*// 更改初始化状态*

initalized = false;

*// key归0*

key = 0;

**return** OK;

}

*// 使用值序列建树的函数*

status CreateBiTree() {

*// 检测是否被初始化*

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** ERROR;

}

*// 提示输入树的*

std::cout << "Please enter the defination of tree: ";

*// 管它有木有默认构造函数。。。*

ValueType value;

*// 设置一个值类型的vector*

std::vector<ValueType> values;

getchar();

*// 尝试得到值序列*

**while** (std::cin >> value) {

*// 没按回车就继续*

**if** (value != '\n') {

values.push\_back(value);

**continue**;

}

**break**;

}

*// 输出读入的元素个数*

std::cout << "Read " << values.size() << " value elements**\n**";

size\_t index = 0;

*// 使用序列建树*

CreateBiTree(head, **nullptr**, values, index);

**return** OK;

}

*// 将树清空的函数*

status ClearBiTree() {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** ERROR;

}

*// 清理空间*

freeTree(head);

*// key归0*

key = 0;

**return** OK;

}

*// 判断树是否为空的函数*

status BiTreeEmpty() {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** true;

}

*// 判断深度是否为0即可*

**return** BiTreeDepth() == 0;

}

*// 得到树的深度，或者说高度*

int BiTreeDepth() {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** 0;

}

*// 直接对根节点使用重载的函数*

**return** BiTreeDepth(head);

}

*// 获取根节点*

NodeType \*Root() **const** {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** **nullptr**;

}

*// 可以直接返回根结点*

**return** head;

}

*// 获取key对应的结点的值*

ValueType Value(**const** int key) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** NULL\_VALUE;

}

*// 先寻找结点指针*

**auto** node = find\_by\_key(head, key);

*// 为空说明不存在*

**if** (node == **nullptr**) {

std::cout << "The tree has no node with key: " << key << "!**\n**";

**return** NULL\_VALUE;

}

*// 返回结点的值*

**return** node->value;

}

*// 对结点进行赋值操作*

status Assign(**const** int key, **const** ValueType value) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** ERROR;

}

*// 先寻找对应结点指针*

**auto** node = find\_by\_key(head, key);

**if** (node == **nullptr**) {

std::cerr << "The tree has no node with key: " << key << "!**\n**";

**return** ERROR;

}

*// 赋值*

node->value = value;

**return** OK;

}

*// 寻找结点的父结点*

NodeType \*Parent(**const** int key) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** **nullptr**;

}

*// 先寻找一下key对应的结点*

**auto** node = find\_by\_key(head, key);

**if** (node == **nullptr**) {

std::cout << "The tree has no node with key: " << key << "!**\n**";

**return** **nullptr**;

}

*// node不为空就存在父节点*

**return** node->parent;

}

*// 返回对应结点的左孩子指针*

NodeType \*LeftChildren(**const** int key) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** **nullptr**;

}

*// 先找到前结点*

**auto** node = find\_by\_key(head, key);

**if** (node == **nullptr**) {

std::cerr << "The tree has no node with key: " << key << "!**\n**";

**return** **nullptr**;

}

*// 返回左孩子*

**return** node->left;

}

*// 逻辑同上*

NodeType \*RightChildren(**const** int key) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** **nullptr**;

}

**auto** node = find\_by\_key(head, key);

**if** (node == **nullptr**) {

std::cout << "The tree has no node with key: " << key << "!**\n**";

**return** **nullptr**;

}

**return** node->right;

}

*// 获取左兄弟结点*

NodeType \*LeftSibling(**const** int key) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** **nullptr**;

}

*// 先找到当前结点*

**auto** node = find\_by\_key(head, key);

*// 判断是否为空*

**if** (node == **nullptr**) {

std::cout << "The tree has no node with key: " << key << "!**\n**";

**return** **nullptr**;

}

*// 判断父结点是否为空*

**if** (node->parent == **nullptr**) {

std::cerr << "Node is head pointer! no sibling.**\n**";

**return** **nullptr**;

}

*// 判断此节点是否就是父节点的左孩子*

**if** (node->parent->left == node) {

std::cout << "This node has no left sibling.**\n**";

**return** **nullptr**;

}

*// 返回左兄弟结点*

**return** node->parent->left;

}

*// 基本思路同上*

NodeType \*RightSibling(**const** int key) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** **nullptr**;

}

**auto** node = find\_by\_key(head, key);

**if** (node == **nullptr**) {

std::cout << "The tree has no node with key: " << key << "!**\n**";

**return** **nullptr**;

}

**if** (node->parent == **nullptr**) {

std::cerr << "Node is head pointer! no sibling.**\n**";

**return** **nullptr**;

}

**if** (node->parent->right == node) {

std::cout << "This node has no right sibling.**\n**";

**return** **nullptr**;

}

**return** node->parent->right;

}

*// 插入子树函数*

status InsertChildren(**const** int key, **const** LR &lr,

**const** BinaryTree<ValueType, NULL\_VALUE> &tree) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** ERROR;

}

*// 获取key对应结点指针*

NodeType \*node = find\_by\_key(head, key);

**if** (node == **nullptr**) {

std::cout << "The tree has no node with key: " << key << "!**\n**";

**return** ERROR;

}

*// 判断是否满足插入条件：插入左边时node左孩子指针为空*

**if** (lr == LR::L && node->left != **nullptr** ||

lr == LR::R && node->right != **nullptr**) {

std::cout << "The child node is not empty!**\n**";

**return** ERROR;

}

*// 满足条件插入即可*

InsertChildren(node, tree.Root(), lr);

**return** OK;

}

*// 删除子树，操作类似于插入*

status DeleteChild(**const** int key, **const** LR &lr) {

**if** (!initalized) {

std::cerr << "The binary tree has not been initalized.**\n**";

**return** ERROR;

}

**auto** node = find\_by\_key(head, key);

**if** (node == **nullptr**) {

std::cout << "The tree has no node with key: " << key << "!**\n**";

**return** ERROR;

}

freeTree(lr == LR::L ? node->**left** : node->right);

**return** OK;

}

*// 前序遍历*

status PreOrderTraverse(**const** Visit<ValueType> &visit) {

**if** (!initalized) {

std::cerr << "This binary tree has not been initalized.**\n**";

**return** ERROR;

}

PreOrderTraverse(head, visit);

**return** OK;

}

*// 中序遍历*

status InOrderTraverse(**const** Visit<ValueType> &visit) {

**if** (!initalized) {

std::cerr << "This binary tree has not been initalized.**\n**";

**return** ERROR;

}

InOrderTraverse(head, visit);

**return** OK;

}

*// 后序遍历*

status PostOrderTraverse(**const** Visit<ValueType> &visit) {

**if** (!initalized) {

std::cerr << "This binary tree has not been initalized.**\n**";

**return** ERROR;

}

PostOrderTraverse(head, visit);

**return** OK;

}

*// 层序遍历*

status LevelOrderTraverse(**const** Visit<ValueType> &visit) {

**if** (!initalized) {

std::cerr << "This binary tree has not been initalized.**\n**";

**return** ERROR;

}

*// 需要借助队列*

std::queue<NodeType\*> nodes;

*// 将头节点压入nodes*

nodes.push(head);

**while** (!nodes.empty()) {

*// 得到最后的压入的结点*

**const** NodeType\* node = nodes.front();

*// 将其弹出队列*

nodes.pop();

*// 访问值*

visit(node->value);

*// 压入左右非空结点*

**if** (node->left != **nullptr**) {

nodes.push(node->left);

}

**if** (node->right != **nullptr**) {

nodes.push(node->right);

}

}

**return** OK;

}

*// 重载输入运算符*

**friend** std::istream &**operator**>>(std::istream &is,

BinaryTree<ValueType, NULL\_VALUE> &tree) {

**if** (!tree.initalized) {

tree.InitBiTree();

tree.CreateBiTree();

}

**else** {

tree.ClearBiTree();

tree.CreateBiTree();

}

**return** is;

}

void displsy() {

display(head);

}

};

#endif *// !DATASTRUCTURE\_BINARY\_TREE\_H*

## 文件testBinaryTree.h内容：

#ifndef DATASTRUCTURE\_TESTBINARYTREE\_H

#define DATASTRUCTURE\_TESTBINARYTREE\_H

*// 包含要测试的数据类型头文件*

#include *"binaryTree.h"*

*// 设置值类型为char型*

**using** ValueType = char;

*// 空字符设置为'#'*

**const** ValueType NULL\_VALUE = '#';

*// 测试主函数*

void testBinaryTree() {

*// 选择的操作*

int operation = -1;

*// 一个辅助变量*

ValueType value = 0;

*// 结点类型指针*

Node<ValueType> \*temp\_node = **nullptr**;

*// 定义一棵二叉树*

BinaryTree<ValueType, NULL\_VALUE> tree;

*// visit函数*

Visit<ValueType> visit = [](**const** ValueType &t) {std::cout << t << std::endl; };

*// 默认键为-1*

int key = -1;

*// 左右选择*

LR lr = LR::L;

*// 插入子树时的临时树*

BinaryTree<ValueType, NULL\_VALUE> temp\_tree;

*// 用户选择的选项*

std::string choice;

**while** (operation != 0) {

*// 清屏*

cls();

*// 打印菜单*

std::cout << "";

std::cout << " Menu for Binary Tree**\n\n**";

std::cout << "---------------------------------------------------**\n**";

std::cout << " 1.InitBiTree 2.DestroyBiTree**\n**";

std::cout << " 3.CreateBiTree 4.ClearBiTree**\n**";

std::cout << " 5.BiTreeEmpty 6.BiTreeDepth**\n**";

std::cout << " 7.Root 8.Value**\n**";

std::cout << " 9.Assign 10.Parent**\n**";

std::cout << " 11.LeftChild 12.RightChild**\n**";

std::cout << " 13.LeftSibling 14.RightSibling**\n**";

std::cout << " 15.InsertChild 16.DeleteChild**\n**";

std::cout << " 17.PreOrderTraverse 18.InOrderTraverse**\n**";

std::cout << " 19.PostOrderTraverse 20.LevelOrderTraverse**\n**";

std::cout << " 0.Exit**\n**";

std::cout << "---------------------------------------------------**\n**";

std::cout << " Please select your selection: ";

std::cin >> choice;

*// 将字符串转为数字*

operation = std::stoi(choice);

*// 根据选择进行操作*

**switch** (operation) {

**case** 1:

**if** (tree.InitBiTree() == OK) {

std::cout << "Initalize tree succefully.**\n**";

}

**else** {

std::cout << "Initalize tree failed.**\n**";

}

wait();

**break**;

**case** 2:

**if** (tree.DestroyBiTree() == OK) {

std::cout << "Destroy tree successfully.**\n**";

}

**else** {

std::cout << "Destroy tree failed.**\n**";

}

wait();

**break**;

**case** 3:

**if** (tree.CreateBiTree() == OK) {

std::cout << "Create tree successfully.**\n**";

}

**else** {

std::cout << "Create tree failed.**\n**";

}

wait();

**break**;

**case** 4:

**if** (tree.ClearBiTree() == OK) {

std::cout << "Clear tree successfully.**\n**";

}

**else** {

std::cout << "Clear tree failed.**\n**";

}

wait();

**break**;

**case** 5:

**if** (tree.BiTreeEmpty()) {

std::cout << "This tree is empty.**\n**";

}

**else** {

std::cout << "This tree is not empty.**\n**";

}

wait();

**break**;

**case** 6:

std::cout << "Tree's depth is: "

<< tree.BiTreeDepth() << "**\n**";

wait();

**break**;

**case** 7:

temp\_node = tree.Root();

**if** (temp\_node != **nullptr**) {

std::cout << "Get root from tree, value: "

<< temp\_node->value << "**\n**";

}

**else** {

std::cout << "Get root from tree failed.**\n**";

}

wait();

**break**;

**case** 8:

tree.displsy();

std::cout << "Please input key of the node: ";

std::cin >> key;

value = tree.Value(key);

std::cout << "Get value: " << value << "**\n**";

wait();

**break**;

**case** 9:

tree.displsy();

std::cout << "Please input the key of node: ";

std::cin >> key;

std::cout << "Please input the value to assign: ";

std::cin >> value;

**if** (tree.Assign(key, value) == OK) {

std::cout << "Assign value successfully.**\n**";

}

**else** {

std::cout << "Assign value failed.**\n**";

}

wait();

**break**;

**case** 10:

tree.displsy();

std::cout << "Please enter the key of node: ";

std::cin >> key;

temp\_node = tree.Parent(key);

**if** (temp\_node == **nullptr**) {

std::cout << "Get the parent node failed.**\n**";

}

**else** {

std::cout << "Get the parent node, value: "

<< temp\_node->value << std::endl;

}

wait();

**break**;

**case** 11:

tree.displsy();

std::cout << "Please enter the key of node: ";

std::cin >> key;

temp\_node = tree.LeftChildren(key);

**if** (temp\_node != **nullptr**) {

std::cout << "Get node's left node, value: "

<< temp\_node->value << std::endl;

}

**else** {

std::cout << "Get left node failed.**\n**";

}

wait();

**break**;

**case** 12:

tree.displsy();

std::cout << "Please enter the key of node: ";

std::cin >> key;

temp\_node = tree.RightChildren(key);

**if** (temp\_node != **nullptr**) {

std::cout << "Get the right children, value: "

<< temp\_node->value << std::endl;

}

wait();

**break**;

**case** 13:

tree.displsy();

std::cout << "Please enter the key of node: ";

std::cin >> key;

temp\_node = tree.LeftSibling(key);

**if** (temp\_node != **nullptr**) {

std::cout << "Get the left sibling of node, value: "

<< temp\_node->value << "**\n**";

}

**else** {

std::cout << "Get the left sibling filed.**\n**";

}

wait();

**break**;

**case** 14:

tree.displsy();

std::cout << "Please enter the key of node: ";

std::cin >> key;

temp\_node = tree.RightSibling(key);

**if** (temp\_node != **nullptr**) {

std::cout << "Get the right sibling of node, value: "

<< temp\_node->value << "**\n**";

}

**else** {

std::cout << "Get right siblling failed.**\n**";

}

wait();

**break**;

**case** 15:

tree.displsy();

std::cout << "Please enter key of node: ";

std::cin >> key;

std::cout << "Please enter the tree: ";

std::cin >> temp\_tree;

std::cout << "insert into left or right(0:left, 1:right):";

std::cin >> temp\_tree\_index;

lr = temp\_tree\_index == 0 ? LR::**L** : LR::R;

**if** (tree.InsertChildren(key, lr, temp\_tree) == OK) {

std::cout << "Insert childern success.**\n**";

}

**else** {

std::cout << "Insert children failed.**\n**";

}

wait();

**break**;

**case** 16:

tree.displsy();

std::cout << "Please enter key of node: ";

std::cin >> key;

std::cout << "Delete left or right(0:left, 1:right):";

std::cin >> temp\_tree\_index;

lr = temp\_tree\_index == 0 ? LR::**L** : LR::R;

**if** (tree.DeleteChild(key, lr) == OK) {

std::cout << "Delete child success.**\n**";

}

**else** {

std::cout << "Delete child failed.**\n**";

}

wait();

**break**;

**case** 17:

std::cout << "PreOrderTraverse Function**\n**";

**if** (tree.PreOrderTraverse(visit) == OK) {

std::cout << "PreOrderTreverse successfully.**\n**";

}

**else** {

std::cout << "PreOrderTreverse failed.**\n**";

}

wait();

**break**;

**case** 18:

std::cout << "InOrderTraverse Function**\n**";

**if** (tree.InOrderTraverse(visit) == OK) {

std::cout << "InOrderTreverse successfully.**\n**";

}

**else** {

std::cout << "InOrderTreverse failed.**\n**";

}

wait();

**break**;

**case** 19:

std::cout << "InOrderTreverse Function**\n**";

**if** (tree.PostOrderTraverse(visit) == OK) {

std::cout << "PostOrderTraverse successfully.**\n**";

}

**else** {

std::cout << "PostOrderTraverse failed.**\n**";

}

wait();

**break**;

**case** 20:

std::cout << "LevelOrderTraverse Function**\n**";

**if** (tree.LevelOrderTraverse(visit) == OK) {

std::cout << "LevelOrderTraverse successfully.**\n**";

}

**else** {

std::cout << "LevelOrderTraverse failed.**\n**";

}

wait();

**break**;

**case** 0:

**break**;

**default**:

**break**;

}

}

}

#endif *// DATASTRUCTURE\_TESTBINARYTREE\_H*

# 附录D 基于邻接表图实现的源程序

## 文件graph.h内容

#ifndef DATASTRUCTURE\_GRAPH\_H

#define DATASTRUCTURE\_GRAPH\_H

#include *"status.h"*

*// 图的模板定义*

**template**<**typename** T>

**class** **Graph** {

**private**:

*// 结点定义*

**template**<**typename** U>

**struct** Node {

*// 结点值定义*

U value;

*// 指向下个边结点的指针*

Node<U> \*next = **nullptr**;

*// 构造函数*

**explicit** Node<U>(**const** U &value) : value(value) {}

};

*// 顶点定义*

**template**<**typename** U>

**struct** Vertex {

*// 键：标识一个顶点*

int key;

*// 值域*

U value;

*// 和本结点相连的点的链表*

Node<U> \*next\_node = **nullptr**;

*// 下一个图结点的指针*

Vertex<U> \*next\_vertex = **nullptr**;

*// 构造函数*

Vertex<U>(**const** U &value, **const** int &key)

: value(value), key(key) {}

*// 添加结点*

bool add\_node(**const** U &value) {

**if** (next\_node == **nullptr**) {

next\_node = **new** Node<U>(value);

**return** true;

}

**else** {

**auto** temp\_n = next\_node;

**while** (temp\_n->next != **nullptr**) {

**if** (temp\_n->next->value == value) {

**return** false;

}

temp\_n = temp\_n->next;

}

temp\_n->next = **new** Node<U>(value);

**return** true;

}

**return** false;

}

*// 删除结点*

bool remove\_node(**const** U &value) {

**if** (next\_node != **nullptr**) {

**if** (next\_node->value == value) {

**auto** t = next\_node;

next\_node = next\_node->next;

**delete** (t);

**return** true;

}

**else** {

**auto** temp\_n = next\_node;

**while** (temp\_n->next != **nullptr**) {

**if** (temp\_n->next->value == value) {

**auto** t = temp\_n->next;

temp\_n->next = t->next;

**delete** (t);

**return** true;

}

temp\_n = temp\_n->next;

}

}

}

**return** false;

}

};

*//using VertexType = Vertex<T>;*

int key = 0;

*// 建立从key到值的映射*

std::map<int, T> map;

*// 头顶点指针*

Vertex<T> \*head = **nullptr**;

*// 是否初始化过*

bool initialized = false;

*// 释放图所占空间*

bool free\_graph() {

**auto** temp\_v = head, next\_v = temp\_v;

**while** (temp\_v != **nullptr**) {

next\_v = temp\_v->next\_vertex;

**auto** temp\_n = temp\_v->next\_node, next\_n = temp\_n;

**while** (temp\_n != **nullptr**) {

next\_n = temp\_n->next;

**delete** (temp\_n);

temp\_n = next\_n;

}

**delete** (temp\_v);

temp\_v = next\_v;

}

head = **nullptr**;

**return** true;

}

*// 根据结点值查找结点指针，找不到就返回空指针*

Vertex<T> \*find\_vertex\_by\_value(**const** T &value) **const** {

**auto** temp\_v = head;

*// 遍历查找*

**while** (temp\_v != **nullptr**) {

**if** (temp\_v->value == value) {

**return** temp\_v;

}

*// 向后移动指针避免死循环*

temp\_v = temp\_v->next\_vertex;

}

*// 查找失败返回空指针*

**return** **nullptr**;

}

*// 释放结点空间*

void freeVertex(**const** Vertex<T> \*vertex) {

**auto** temp\_n = vertex->next\_node, next\_n = temp\_n;

*// 遍历结点邻接表，释放空间*

**while** (temp\_n != **nullptr**) {

next\_n = temp\_n->next;

*// delete空间*

**delete** (temp\_n);

temp\_n = next\_n;

}

*// 释放当前结点空间*

**delete** (vertex);

}

*// 深度遍历函数*

status DFSTraverse(**const** Vertex<T> \*vertex,

std::map<T, bool> &visited,

**const** Visit<T> &visit) **const** {

*// 先访问当前节点值*

visit(vertex->value);

*// 获取当前顶点的结点邻接表*

**auto** temp\_n = vertex->next\_node;

**while** (temp\_n != **nullptr**) {

*// 结点值未遍历就递归进行DFS*

**if** (!visited[temp\_n->value]) {

visited[temp\_n->value] = true;

DFSTraverse(find\_vertex\_by\_value(temp\_n->value),

visited, visit);

}

*// 结点指针后移，防止死循环*

temp\_n = temp\_n->next;

}

**return** OK;

}

*// 广度遍历*

status BFSTraverse(**const** Vertex<T> \*vertex,

std::map<T, bool> &visited,

**const** Visit<T> &visit) **const** {

*// 使用队列遍历存储未遍历的结点的指针*

std::queue<**const** Vertex<T>\*> vertexes;

*// 结点入队列*

vertexes.push(vertex);

visited[vertex->value] = true;

*// 队列非空继续遍历*

**while** (!vertexes.empty()) {

**const** **auto** v = vertexes.front();

visit(v->value);

**auto** temp\_n = v->next\_node;

*// 将当前结点非空的结点全部加入*

**while** (temp\_n != **nullptr**) {

**if** (!visited[temp\_n->value]) {

vertexes.push(find\_vertex\_by\_value(temp\_n->value));

visited[temp\_n->value] = true;

}

temp\_n = temp\_n->next;

}

*// 已经访问过的结点pop掉*

vertexes.pop();

}

**return** OK;

}

**public**:

*// 必须在这里设置下别名才能在别的文件访问到*

**using** VertexType = Vertex<T>;

**using** NodeType = Node<T>;

*// 建立图的函数*

status CreateGraph(**const** std::vector<T> &nodes,

**const** std::vector<std::pair<T, T>> &edges) {

**if** (initialized) {

std::cout << "Your graph had been initialized,\

please destroy it before create.**\n**";

**return** ERROR;

}

*// 先将顶点处理了*

**decltype**(head) last\_node = **nullptr**;

*// 键值映射先清空*

map.clear();

**for** (size\_t i = 0; i < nodes.size(); i++) {

*// 首个结点将head设置为第一个vertex的地址*

**if** (i == 0) {

last\_node = head = **new** Vertex<T>(nodes[i], key++);

}

**else** {

*// 正常添加*

last\_node = last\_node->next\_vertex =

**new** Vertex<T>(nodes[i], key++);

}

*// map及时添加key对应的value*

map[last\_node->key] = last\_node->value;

}

*// 使用邻接表建图*

**for** (**const** **auto** &**edge** : edges) {

**auto** v1 = edge.first, v2 = edge.second;

VertexType \*temp\_v = head;

*// 遍历顶点的集合，添加结点*

**while** (temp\_v != **nullptr**) {

**if** (temp\_v->value == v1) {

temp\_v->add\_node(v2);

}

**else** **if** (temp\_v->value == v2) {

temp\_v->add\_node(v1);

}

*// 指针后移*

temp\_v = temp\_v->next\_vertex;

}

}

*// 初始化完成*

initialized = true;

**return** OK;

}

*// 销毁图的函数*

status DestroyGraph() {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 释放图*

**if** (free\_graph()) {

key = 0;

initialized = false;

map.clear();

**return** OK;

}

**else** {

**return** ERROR;

}

}

*// 查找顶点的位置信息*

int LocateVertex(**const** int &ver\_key) {

**if** (!initialized) {

std::cerr << "This graph has not been initaialized.**\n**";

**return** -1;

}

*// 未找到表明顶点不在此图中*

**auto** it = map.find(ver\_key);

**if** (it != map.cend()) {

*// 找到返回结点的key*

**return** ver\_key;

}

**return** -1;

}

*// 通过key寻找结点*

status GetVertex(T &value, **const** int &ver\_key) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 寻找结点值*

**auto** it = map.find(ver\_key);

**if** (it != map.cend()) {

value = it->second;

**return** OK;

}

**return** ERROR;

}

*// 为结点赋值*

status PutVertex(**const** T &value, **const** int &ver\_key) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

T find\_value;

*// 得到迭代器*

**auto** it = map.find(ver\_key);

*// 如果找到了需要将所有这个结点的值更新一遍*

**if** (it != map.cend()) {

find\_value = it->second;

*// 找到了就需要遍历了*

*// 先把map更新一下*

map[ver\_key] = value;

**auto** temp\_v = head;

**while** (temp\_v != **nullptr**) {

*// 顶点也要更新*

**if** (temp\_v->value == find\_value) {

temp\_v->value = value;

}

**auto** temp\_n = temp\_v->next\_node;

**while** (temp\_n != **nullptr**) {

*// 找到了对应的node将值更新*

**if** (temp\_n->value == find\_value) {

temp\_n->value = value;

**break**;

}

*// 后移*

temp\_n = temp\_n->next;

}

*// 向后移动*

temp\_v = temp\_v->next\_vertex;

}

}

**else** {

*// 没有找到，说明顶点集里没有参数给的*

**return** ERROR;

}

**return** OK;

}

*// 寻找第一邻接点*

NodeType \*FirstAdjacentVertex(**const** int &ver\_key) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 遍历寻找顶点*

**auto** \*vertex = head;

**while** (vertex != **nullptr**) {

**if** (vertex->key == ver\_key) {

*// 找到后直接返回其邻接表中第一个结点的指针即可*

**return** vertex->next\_node;

}

vertex = vertex->next\_vertex;

}

*// 未找到返回空指针*

**return** **nullptr**;

}

*// 寻找下一邻接点*

NodeType \*NextAdjacentVertex(**const** int &ver\_key, **const** int &node\_key) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 直接寻找第一个邻接点，重用代码*

**auto** \*nodes = FirstAdjacentVertex(ver\_key);

**auto** it = map.find(ver\_key);

**if** (it != map.cend()) {

*// 寻找给出临界点的值*

**auto** node\_iter = map.find(node\_key);

**if** (node\_iter == map.cend()) {

**return** **nullptr**;

}

T value = node\_iter->second;

**while** (nodes != **nullptr**) {

*// 返回下一邻接点*

**if** (nodes->value == value) {

**return** nodes->next;

}

*// 指针后移*

nodes = nodes->next;

}

}

*// 查找失败返回空指针*

**return** **nullptr**;

}

*// 插入顶点*

status InsertVertex(**const** T &value) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 图为空的情况添加新结点为头指针*

**if** (head == **nullptr**) {

head = **new** Vertex<T>(value, key++);

*// map更新值*

map[head->key] = value;

**return** OK;

}

**auto** temp\_v = head;

*// 寻找最后的顶点，在其后接上加入的顶点*

**while** (temp\_v->next\_vertex != **nullptr**) {

temp\_v = temp\_v->next\_vertex;

}

temp\_v->next\_vertex = **new** Vertex<T>(value, key++);

*// 更新map*

map[temp\_v->next\_vertex->key] = value;

**return** OK;

}

*// 删除顶点*

status DeleteVertex(**const** int &ver\_key) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 寻找顶点是否存在于当前图*

**auto** it = map.find(ver\_key);

**if** (it == map.cend()) {

**return** ERROR;

}

*// 删除map中顶点的值*

T value = it->second;

map.erase(it);

**auto** temp\_v = head;

*// 删除结点有点麻烦，需要删除边和链表中的结点*

*// 要删除的结点为首节点*

**if** (temp\_v->key == ver\_key) {

head = head->next\_vertex;

freeVertex(temp\_v);

}

**else** {

*// 要删除的顶点不是头结点*

**while** (temp\_v->next\_vertex != **nullptr**) {

**if** (temp\_v->next\_vertex->value == value) {

**auto** t = temp\_v->next\_vertex;

temp\_v->next\_vertex = t->next\_vertex;

freeVertex(t);

}

temp\_v = temp\_v->next\_vertex;

}

}

*// 删除图里的点*

temp\_v = head;

**while** (temp\_v != **nullptr**) {

temp\_v->remove\_node(value);

temp\_v = temp\_v->next\_vertex;

}

**return** OK;

}

*// 插入弧*

status InsertArc(**const** std::pair<T, T> &edge) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 得到边的两个顶点*

**auto** v1 = edge.first, v2 = edge.second;

**auto** temp\_v = head;

*// 遍历删除*

**while** (temp\_v != **nullptr**) {

**if** (temp\_v->value == v1) {

temp\_v->add\_node(v2);

}

**else** **if** (temp\_v->value == v2) {

temp\_v->add\_node(v1);

}

*// 指针后移*

temp\_v = temp\_v->next\_vertex;

}

**return** OK;

}

*// 删除弧*

status DeleteArc(**const** T &v1, **const** T &v2) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

*// 标记删除是否成功*

bool del = false;

**auto** temp\_v = head;

**while** (temp\_v != **nullptr**) {

**if** (temp\_v->value == v1) {

*// 配对删除*

del = temp\_v->remove\_node(v2);

}

**else** **if** (temp\_v->value == v2) {

del = temp\_v->remove\_node(v1);

}

temp\_v = temp\_v->next\_vertex;

}

*// 删除成功返回OK*

**return** del ? **OK** : ERROR;

}

*// 深度遍历*

status DFSTraverse(**const** Visit<T> &visit) **const** {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

std::map<T, bool> visited;

**auto** temp\_v = head;

**if** (head != **nullptr**) {

*// 建立map*

**while** (temp\_v != **nullptr**) {

visited[temp\_v->value] = false;

temp\_v = temp\_v->next\_vertex;

}

temp\_v = head;

*// 循环调用DFS，因为图不一定相连*

**while** (temp\_v != **nullptr**) {

**if** (!visited[temp\_v->value]) {

visited[temp\_v->value] = true;

DFSTraverse(temp\_v, visited, visit);

}

temp\_v = temp\_v->next\_vertex;

}

}

**return** OK;

}

*// 广度遍历*

status BFSTraverse(**const** Visit<T> &visit) {

**if** (!initialized) {

std::cerr << "This graph has not been initialized.**\n**";

**return** ERROR;

}

std::map<T, bool> visited;

**auto** temp\_v = head;

**if** (head != **nullptr**) {

*// 建立map*

**while** (temp\_v != **nullptr**) {

visited[temp\_v->value] = false;

temp\_v = temp\_v->next\_vertex;

}

*// temp\_v设置为head*

temp\_v = head;

**while** (temp\_v != **nullptr**) {

*// 循环使用BFS，避免因为图不连通导致无法遍历完全*

**if** (!visited[temp\_v->value]) {

BFSTraverse(head, visited, visit);

}

temp\_v = temp\_v->next\_vertex;

}

}

**return** OK;

}

*// 显示结点的key和value*

void display\_vertex() {

**auto** temp\_v = head;

**while** (temp\_v != **nullptr**) {

std::cout << "key: " << temp\_v->key

<< ", value: " << temp\_v->value << "**\n**";

temp\_v = temp\_v->next\_vertex;

}

}

*// 显示邻接表形式的图遍历*

void display\_linked\_list() {

**auto** temp\_v = head;

**while** (temp\_v != **nullptr**) {

**auto** temp\_n = temp\_v->next\_node;

std::cout << temp\_v->value;

**while** (temp\_n != **nullptr**) {

std::cout << "->" << temp\_n->value;

temp\_n = temp\_n->next;

}

std::cout << "**\n**";

temp\_v = temp\_v->next\_vertex;

}

}

*// 从文件加载图*

bool LoadGraphFromFile(**const** std::string &file\_name) {

std::fstream in(file\_name, std::ios\_base::in);

**if** (in.is\_open()) {

size\_t count = 0;

T v1, v2;

initialized = true;

DestroyGraph();

initialized = true;

in >> count;

**for** (size\_t i = 0; i < count; i++) {

in >> v1;

InsertVertex(v1);

}

in >> count;

**for** (size\_t i = 0; i < count; i++) {

in >> v1 >> v2;

InsertArc(std::make\_pair(v1, v2));

}

in.close();

std::cout << "Load successfully.**\n**";

**return** true;

}

**else** {

**return** false;

}

}

};

*// 输入结点集合*

**template**<**typename** T>

void input\_vertexes(std::vector<T> &vertexes) {

std::cout << "PLease enter count of vertexes: ";

size\_t count = 0;

std::cin >> count;

T value;

vertexes.clear();

vertexes.reserve(count);

std::cout << "PLease enter the value of vertexes: ";

**for** (size\_t i = 0; i < count; i++) {

std::cin >> value;

vertexes.push\_back(value);

}

}

*// 输入边集合*

**template**<**typename** T>

void input\_edges(std::vector<std::pair<T, T>> &edges) {

std::cout << "Please enter count of edges: ";

size\_t count = 0;

std::cin >> count;

T v1, v2;

edges.clear();

edges.reserve(count);

std::cout << "Please enter these edges: ";

**for** (size\_t i = 0; i < count; i++) {

std::cin >> v1 >> v2;

edges.push\_back(std::make\_pair(v1, v2));

}

}

#endif *// !DATASTRUCTURE\_GRAPH\_H*

## 文件testGraph.txt

#ifndef DATASTRUCTURE\_TESTGRAPH\_H

#define DATASTRUCTURE\_TESTGRAPH\_H

#include *"graph.h"*

*// 值类型定义*

**using** ValueType = char;

void testGraph() {

*// 图*

Graph<ValueType> graph;

*// 边集合*

std::vector<std::pair<ValueType, ValueType>> edges;

*// 一条边*

std::pair<ValueType, ValueType> edge;

*// 顶点集合*

std::vector<ValueType> vertexes;

*// 单个顶点*

ValueType v1, v2;

*// 单个边顶点*

Graph<ValueType>::NodeType \*node = **nullptr**;

*// 单个图顶点*

Graph<ValueType>::VertexType \*vertex = **nullptr**;

*// 顶点值暂时变量*

ValueType value = '#';

*// 键值暂存变量*

int key = 0, node\_key = 0;

*// 选择的操作*

int operation = -1;

*// 访问函数*

Visit<ValueType> visit = [](**const** **auto** &value) { std::cout << value << "**\n**"; };

*// 文件名*

std::string file\_name;

*// 主循环*

**while** (operation != 0) {

cls();

std::cout << " Menu for Graph**\n**";

std::cout << "--------------------------------------------------**\n**";

std::cout << " 1.CreateGraph 2.DestroyGraph**\n**";

std::cout << " 3.LocateVertex 4.GetVertex**\n**";

std::cout << " 5.PutVertex 6.FirstAdjacentVertex**\n**";

std::cout << " 7.NextAdjacentVertex 8.InsertVertex**\n**";

std::cout << " 9.DeleteVertex 10.InsertArc**\n**";

std::cout << " 11.DeleteArc 12.DFSTraverse**\n**";

std::cout << " 13.BFSTraverse 14.LoadGraphFromFile**\n**";

std::cout << " 15.DisplayLinkedList**\n**";

std::cout << " 0.Exit**\n**";

std::cout << "--------------------------------------------------**\n\n**";

std::cout << "Please enter your selection: ";

std::cin >> operation;

**switch** (operation) {

**case** 1:

std::cout << "CreateGraph Function**\n**";

input\_vertexes(vertexes);

input\_edges(edges);

std::cout << "End input.**\n**";

**if** (graph.CreateGraph(vertexes, edges) == OK) {

std::cout << "Create graph successfully.**\n**";

}

**else** {

std::cout << "Create graph failed.**\n**";

}

wait();

**break**;

**case** 2:

std::cout << "DeatroyGraph Function**\n**";

**if** (graph.DestroyGraph() == OK) {

std::cout << "Deatroy graph successfully.**\n**";

}

**else** {

std::cout << "Destroy graph failed.**\n**";

}

wait();

**break**;

**case** 3:

std::cout << "LocateVertex Function**\n**";

graph.display\_vertex();

std::cout << "Please input key of node: ";

std::cin >> key;

std::cout << graph.LocateVertex(key) << "**\n**";

wait();

**break**;

**case** 4:

std::cout << "GetVertex Function**\n**";

graph.display\_vertex();

std::cout << "PLease enter key of vertex: ";

std::cin >> key;

**if** (graph.GetVertex(value, key) == OK) {

std::cout << "Get value: " << value << "**\n**";

}

**else** {

std::cout << "Get value failed.**\n**";

}

wait();

**break**;

**case** 5:

std::cout << "PutVertex Function**\n**";

graph.display\_vertex();

std::cout << "PLease enter key of vertex: ";

std::cin >> key;

std::cout << "Pleas enter value to put: ";

std::cin >> value;

**if** (graph.PutVertex(value, key) == OK) {

std::cout << "Put value into vertex successfully.**\n**";

}

**else** {

std::cout << "Put value into vertex failed.**\n**";

}

wait();

**break**;

**case** 6:

std::cout << "FirstAdjacentVertex Function**\n**";

graph.display\_vertex();

std::cout << "PLease enter key of vertex: ";

std::cin >> key;

node = graph.FirstAdjacentVertex(key);

**if** (node != **nullptr**) {

std::cout << "Get vertex: " << node->value << "**\n**";

}

**else** {

std::cout << "Get vertex failed.**\n**";

}

wait();

**break**;

**case** 7:

std::cout << "NextAjcacentVertex Function**\n**";

graph.display\_vertex();

std::cout << "PLease enter key of vertex: ";

std::cin >> key;

std::cout << "Please enter key of node: ";

std::cin >> node\_key;

node = graph.NextAdjacentVertex(key, node\_key);

**if** (node != **nullptr**) {

std::cout << "Get next adjacent vertext: "

<< node->value << "**\n**";

}

**else** {

std::cout << "Get next adjacent vertex failed.**\n**";

}

wait();

**break**;

**case** 8:

std::cout << "InsertVertex Function**\n**";

std::cout << "PLease enter value of vertex: ";

std::cin >> value;

**if** (graph.InsertVertex(value) == OK) {

std::cout << "Insert vertex successfully.**\n**";

}

**else** {

std::cout << "Insert vertex failed.**\n**";

}

wait();

**break**;

**case** 9:

std::cout << "DeleteVertex Function**\n**";

graph.display\_vertex();

std::cout << "PLease enter key of vertex: ";

std::cin >> key;

**if** (graph.DeleteVertex(key) == OK) {

std::cout << "Delete vertex successfully.**\n**";

}

**else** {

std::cout << "Delete vertex failed.**\n**";

}

wait();

**break**;

**case** 10:

std::cout << "InsertVertex Function**\n**";

std::cout << "PLease enter arc: ";

std::cin >> edge.first >> edge.second;

**if** (graph.InsertArc(edge) == OK) {

std::cout << "Insert arc successfully.**\n**";

}

**else** {

std::cout << "Insert arc failed.**\n**";

}

wait();

**break**;

**case** 11:

std::cout << "DeleteArc Function**\n**";

std::cout << "PLese enter two vertex of arc: ";

std::cin >> v1 >> v2;

**if** (graph.DeleteArc(v1, v2) == OK) {

std::cout << "Delete arc successfully.**\n**";

}

**else** {

std::cout << "Delete arc failed.**\n**";

}

wait();

**break**;

**case** 12:

std::cout << "DFSTRaverse Function**\n**";

**if** (graph.DFSTraverse(visit) == OK) {

std::cout << "DFSTraverse successfully.**\n**";

}

**else** {

std::cout << "DFSTraverse failed.**\n**";

}

wait();

**break**;

**case** 13:

std::cout << "BFSTraverse Function**\n**";

**if** (graph.BFSTraverse(visit) == OK) {

std::cout << "BFSTraverse successfully.**\n**";

}

**else** {

std::cout << "BFSTraverse failed.**\n**";

}

wait();

**break**;

**case** 14:

std::cout << "Loadgraph Function**\n**";

std::cout << "Please enter file name: ";

std::cin >> file\_name;

graph.LoadGraphFromFile(file\_name);

wait();

**break**;

**case** 15:

std::cout << "DisplayLinkedList Function**\n**";

graph.display\_linked\_list();

wait();

**break**;

**case** 0:

**break**;

**default**:

**break**;

}

}

std::cout << "Good Bye.**\n**";

}

#endif *// !DATASTRUCTURE\_TESTGRAPH\_H*