// Thread\_BinaryTree\_OJ\_Homework.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

//

/\* 线索二叉树的结点定义 \*/

#include<iostream>

#include<sstream>

#include<cstring>

using namespace std;

enum Tags { Link = 0, Thread = 1 };//Link用于表示对应的结点的某一边有孩子，而Thread表示对应结点的某一边没有孩子

template<class ElemType>

struct Thread\_BinaryTreeNode {

Thread\_BinaryTreeNode<ElemType>\* LChild, \* RChild;//当lTag==Link时，LChild指向自己的左孩子结点，当LChild==

//Thraed时，LChild指向前驱结点，同理：当rTag==Link时，RChild指向自己的右孩子结点，当rTag==Thread时，RChild

//指向后继

ElemType data;

Tags lTag, rTag;//是否有无孩子结点的标志位（等于Link时有孩子结点，等于Thraed时没有）

Thread\_BinaryTreeNode() : LChild(NULL), RChild(NULL), lTag(Link), rTag(Link) {} //构造函数1，用于构造根结点

//注意，默认条件下构造根结点时都认为有孩子结点

Thread\_BinaryTreeNode(const ElemType& item, Thread\_BinaryTreeNode<ElemType>\* Lptr = NULL, Thread\_BinaryTreeNode<ElemType>\* Rptr = NULL, Tags leftTag = 0, Tags rightTag = 0) //构造函数2，用于构造其他结点

//函数参数表中的形参允许有默认值，但是带默认值的参数需要放后面

{//注：用于构造其他结点的该构造函数不默认有孩子结点

LChild = Lptr;

RChild = Rptr;

data = item;

lTag = leftTag;

rTag = rightTag;

}

ElemType getData() { return data; } //取得结点中的数据

void SetLChild(Thread\_BinaryTreeNode<ElemType>\* link) { LChild = link; } //修改结点的左孩子域

void SetRChild(Thread\_BinaryTreeNode<ElemType>\* link) { RChild = link; } //修改结点的右孩子域

void SetData(const ElemType& value) { data = value; } //修改结点的data域

Thread\_BinaryTreeNode<ElemType>\* GetLChild() const { return LChild; } //获取左孩子结点

Thread\_BinaryTreeNode<ElemType>\* GetRChild() const { return RChild; } //获取左孩子结点

void SetLTag(Tags tag) { lTag = tag; }//更改左指针指向类型（是指向孩子还是指向前驱）的标志位

void SetRTag(Tags tag) { rTag = tag; }//更改右指针指向类型（是指向孩子还是指向后继）的标志位

//结点做右指针赋值函数

void AssignLeft(Thread\_BinaryTreeNode<ElemType>\* pre) { LChild = pre; }

void AssignRight(Thread\_BinaryTreeNode<ElemType>\* p) { RChild = p; }

//判断对应的标志位的情况进而判断结点左右指针指向的类型

bool node\_RTG() { return (rTag == Thread) ? Thread : Link; }

bool node\_LTG() { return (lTag == Thread) ? Thread : Link; }

};

//中序线索二叉树

template<class ElemType>

class Thread\_BinaryTree {

private:

Thread\_BinaryTreeNode<ElemType>\* root; // 根结点指针

void Thread\_BinaryTreeDestroy\_Cursive(Thread\_BinaryTreeNode<ElemType>\*& T); //销毁树（递归部分，private）

void InThreading(Thread\_BinaryTreeNode<ElemType>\* root, Thread\_BinaryTreeNode<ElemType>\*& pre); //中序线索化二叉树（递归, private）

//注意：这里虽然线索化后的二叉树有一个头结点，但是如果把这个头结点加到这个类中的话就会显得比较多余，而且

//相关的构造函数处理起来比较麻烦，所以这里我们采用的做法是把这个头结点放到外面去，即在调用完二叉树线索化

//函数后会返回一个指向头结点的指针（头结点本身），好多类内的函数在发挥其作用的时候需要这个头结点

public:

//无参数的构造函数

Thread\_BinaryTree() :root(NULL) {}

//带参数的构造函数

Thread\_BinaryTree(const ElemType& item) { root = new Thread\_BinaryTreeNode<ElemType>(item); }

//生成树（暂时认为没有什么用）

//void makeBinaryTree( const ElemType &item, BinaryTree &left, BinaryTree &right);

//析构函数

~Thread\_BinaryTree() {

bool b = Thread\_BinaryTreeDestroy();

/\*if(b)

cout << "调用了析构函数" << endl; \*/

}

//销毁线索二叉树

bool Thread\_BinaryTreeDestroy();

//判断线索二叉树是否为空

bool Thread\_BinaryTreeisEmpty() const { return root == NULL; }

//获取根结点元素值

ElemType getRootData() const { return root->data; }

//设置根结点

void setRoot(Thread\_BinaryTreeNode<ElemType>\* p) { root = p; }

//返回指向根结点的指针

Thread\_BinaryTreeNode<ElemType>\* getRoot() const { return root; }

//查找值为x的结点的位置

Thread\_BinaryTreeNode<ElemType>\* get\_x\_location(const ElemType& x)const;

//查找值为x的结点的位置 （递归）

Thread\_BinaryTreeNode<ElemType>\* location(Thread\_BinaryTreeNode<ElemType>\* root, const ElemType& x)const;

//在中序线索二叉树上查找值为x的结点的中序前驱结点

Thread\_BinaryTreeNode<ElemType>\* findPrior\_InorderTree(const ElemType& x)const;

//在中序线索二叉树上查找值为x的结点的中序后继结点

Thread\_BinaryTreeNode<ElemType>\* findNext\_InorderTree(const ElemType& x)const;

//二叉树的中序线索化

Thread\_BinaryTreeNode<ElemType>\* inOrderThreading(Thread\_BinaryTreeNode<ElemType>\* root);

//中序线索二叉树的遍历

void inThreading\_Traverse(Thread\_BinaryTreeNode<ElemType>\* root/\*, bool (\*visit)(Thread\_BinaryTreeNode<ElemType>\* root)\*/);

//建立二叉树的存储结构 (递归）

Thread\_BinaryTreeNode<ElemType>\* CreateThreadBinaryTree(ElemType x[], const ElemType& empty, int& n);

};

//销毁树（递归部分，private）

template<class ElemType>

void Thread\_BinaryTree<ElemType>::Thread\_BinaryTreeDestroy\_Cursive(Thread\_BinaryTreeNode<ElemType>\*& T) {

if (T == NULL) {

delete T;

return;

}

if (T->lTag == Thread && T->rTag == Thread) {

T = NULL;

delete T;//root若无子树，直接删除

}

else {//运用后序遍历，先左右再根部

if (T->lTag == Link) {

Thread\_BinaryTreeDestroy\_Cursive(T->LChild);

}

if (T->rTag == Link) {

Thread\_BinaryTreeDestroy\_Cursive(T->RChild);

}

T = NULL;

delete T;

}

}

//销毁线索二叉树

template<class ElemType>

bool Thread\_BinaryTree<ElemType>::Thread\_BinaryTreeDestroy() {

Thread\_BinaryTreeDestroy\_Cursive(root); //销毁树（递归部分，private）

if (root)

return false;

else return true;

}//经过测试没有什么问题

// 查找值为x的结点的位置

template<class ElemType>

Thread\_BinaryTreeNode<ElemType>\* Thread\_BinaryTree<ElemType>::get\_x\_location(const ElemType& x)const {

Thread\_BinaryTreeNode<ElemType>\* r = location(root, x);

return r;

}

//查找值为x的结点的位置 （递归）

template<class ElemType>

Thread\_BinaryTreeNode<ElemType>\* Thread\_BinaryTree<ElemType>::location(Thread\_BinaryTreeNode<ElemType>\* T, const ElemType& x)const {

if (!T) {

return NULL;

}

else {//采用中序遍历找到的第一个等于x的结点的指针

Thread\_BinaryTreeNode<ElemType>\* re = NULL;

if (T->lTag == Link) {//有左孩子

re = location(T->LChild, x);

}

if (re == NULL) {

if (T->data == x) {

return T;

}

else if (T->rTag == Link) {//有右孩子

re = location(T->RChild, x);

}

}

return re;

}

}//暂时经过测试没有问题

//在中序线索二叉树上查找值为x的结点的中序前驱结点

template<class ElemType>

Thread\_BinaryTreeNode<ElemType>\* Thread\_BinaryTree<ElemType>::findPrior\_InorderTree(const ElemType& x)const {

Thread\_BinaryTreeNode<ElemType>\* pointer = get\_x\_location(x);

if (!pointer) {//一定要考虑所给结点可能在二叉树中找不到，那么也就没有必要找找到其前驱和后继，这个时候我们让

//函数返回NULL

return NULL;

}

if (pointer->lTag == Thread) {//此时表明该结点的做指针指向的应该是其中序遍历的前驱

return pointer->LChild;

}

else {//表明其有左子树，那么其前驱应该是左子树最靠右的一个结点，即左子树按中序遍历的最后一个结点

Thread\_BinaryTreeNode<ElemType>\* p = pointer->LChild;

while (p->rTag == Link) {//如果这个结点有右子树，就一直往右子树的方向找，因为根据中序遍历的特点，子树的

//最后一个结点必然位于右子树上，如果没有，那么必然就是这个结点本身，这里如果

p = p->RChild;

}

return p;

}

}

//在中序线索二叉树上查找值为x的结点的中序后继结点

template<class ElemType>

Thread\_BinaryTreeNode<ElemType>\* Thread\_BinaryTree<ElemType>::findNext\_InorderTree(const ElemType& x)const {

Thread\_BinaryTreeNode<ElemType>\* pointer = get\_x\_location(x);

if (!pointer) {//没有找到指向x指针

return NULL;

}

if (pointer->rTag == Thread) {//说明结点的RChild指向的类型是后继结点

return pointer->RChild;

}

else {//说明结点的RChild指向的类型是其右子树

Thread\_BinaryTreeNode<ElemType>\* p = pointer->RChild;

while (p->lTag == Link) {

p = p->LChild;

}

return p;

}

}//找前驱和后继结点函数暂时没有发现什么问题

template<class ElemType>

Thread\_BinaryTreeNode<ElemType>\* Thread\_BinaryTree<ElemType>::CreateThreadBinaryTree(ElemType x[], const ElemType& empty, int& n) {

ElemType element = x[n];

n++;

if (element == empty) {

return NULL;

}

else {

Thread\_BinaryTreeNode<ElemType>\* Node = new Thread\_BinaryTreeNode<ElemType>;

Node->data = element;

Node->LChild = CreateThreadBinaryTree(x, empty, n);

if (Node->LChild == NULL) {

Node->lTag = Thread;

}

else Node->lTag = Link;

Node->RChild = CreateThreadBinaryTree(x, empty, n);

if (Node->RChild == NULL) {

Node->rTag = Thread;

}

else Node->rTag = Link;

return Node;

}

}

//建立二叉树的存储结构 （外壳部分，用户函数）

//注意这个不是类内的函数

template<class ElemType>

void CreateTreadBinaryTree(Thread\_BinaryTree<ElemType>& T, const string& str, ElemType& empty) {

ElemType tmp, t[100];

int num = 0;

istringstream input\_T(str);

while (input\_T >> tmp) {//这里是把str字符串中的内容以空格分开并转化为tmp类型依次读到tmp中

t[num] = tmp;

num++;

}

Thread\_BinaryTreeNode<ElemType>\* root;

num = 0;//num

root = T.CreateThreadBinaryTree(t, empty, num);

T.setRoot(root);

}

template<class ElemType>

void Thread\_BinaryTree<ElemType>::InThreading(Thread\_BinaryTreeNode<ElemType>\* T,

Thread\_BinaryTreeNode<ElemType>\*& pre) { //中序线索化二叉树（递归, private）

if (T) {//T必须不能为空

InThreading(T->LChild, pre);

if (!T->LChild) {

T->lTag = Thread;

T->LChild = pre;

}

if (!pre->RChild) {

pre->rTag = Thread;

pre->RChild = T;

}

pre = T;

InThreading(T->RChild, pre);

}

}

//二叉树的中序线索化

template<class ElemType>

Thread\_BinaryTreeNode<ElemType>\* Thread\_BinaryTree<ElemType>::inOrderThreading(Thread\_BinaryTreeNode<ElemType>\* T) {

Thread\_BinaryTreeNode<ElemType>\* head = new Thread\_BinaryTreeNode<ElemType>;//创建头结点并分配空间

if (head == NULL) {

return NULL;//未成功分配到空间，线索化失败

}

head->RChild = head;//暂时回指，待中序遍历结束后就指向中序遍历的最后一个结点

head->lTag = Link;//头结点的左指针类型相当于指向一个孩子结点

head->rTag = Thread;//头指针的右指针类型相当于是指向一个后继结点

Thread\_BinaryTreeNode<ElemType>\* pre;//永远指向当前访问的前驱结点

if (!T) {//二叉树为空

head->LChild = head;//为空则左指针指向自己

}

else {

head->LChild = T;

pre = head;

InThreading(T, pre);

pre->RChild = head;

pre->rTag = Thread;//这里注意，二叉树线索化的时候，我们把中序遍历的最后一个结点的rTag设置为了Thread，即

//最后一个结点的右指针类型相当于指向的是后继

head->RChild = pre;

}

return head;

}

//中序线索二叉树的遍历

template<class ElemType>

void Thread\_BinaryTree<ElemType>::inThreading\_Traverse(Thread\_BinaryTreeNode<ElemType>\* head

/\*, bool (\*visit)(Thread\_BinaryTreeNode<ElemType>\* root)\*/) {//传进来的root是指向根结点的指针，在该函数内我们

//采用和普通而二叉树中序遍历一样的方法进行操作

Thread\_BinaryTreeNode<ElemType>\* p = head->LChild;

while (p != head) {//如果该结点没有左孩子，才说明终于找到了第一个对象

while (p->lTag == Link) {//说明此时p结点有左孩子，继续往下找

p = p->LChild;

}

cout << p->data<<' ' /\*<<'(' << p->lTag << ',' << p->rTag << ')' << ' '\*/;

while (p->rTag == Thread && p->RChild != head) {//没有右孩子且下一个结点不是头结点

p = p->RChild;

cout << p->data<< ' ' /\*<< '(' << p->lTag << ',' << p->rTag << ')' << ' '\*/;

}

p = p->RChild;

}

}

int main(){

Thread\_BinaryTree<char>T;

char empty;

string str;

cin >> empty;

cin.ignore();

getline(cin, str);

CreateTreadBinaryTree(T, str, empty);

Thread\_BinaryTreeNode<char>\* head=T.inOrderThreading(T.getRoot());

char x;

cin >> x;

Thread\_BinaryTreeNode<char>\* p = T.get\_x\_location(x);

T.inThreading\_Traverse(head);

cout << endl;

if (p) {

cout << p->data<<endl;

Thread\_BinaryTreeNode<char>\* p1 = T.findPrior\_InorderTree(x);

Thread\_BinaryTreeNode<char>\* p2 = T.findNext\_InorderTree(x);

if (p1 != head) {

cout << p1->data << endl;

}

else {

cout << "NULL";

}

if (p2 != head) {

cout << p2->data;

}

else {

cout << "NULL";

}

}

else {

cout << "NULL";

}

return 0;

}