《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 21计科 | 姓名 | 方浩楠 | 学号 | 2127405048 |
| 实验布置日期 | | 2022.11.1 | | 提交  日期 | 2022.11.20 | | 成绩 |  |

课程实践实验7：二叉树的实现及测试

## 问题描述及要求

假设二叉树的结点值为单个字符。要求能演示二叉树的基本操作。基本操作至少包括：构造二叉树，按先序、中序、后序、层序遍历这棵二叉树，计算二叉树的深度、叶子结点数目。

二叉树采用链式存储结构。

需完成的二叉树类方法参考：

（1）构造空二叉树

（2）判别二叉树是否为空

（3）二叉树先序遍历、中序遍历、后序遍历递归实现

（4）对二叉树进行层次遍历

（5）二叉树先序遍历、中序遍历非递归实现

（6）求二叉树的结点数、叶子结点数

（7）清空已有二叉树

（8）求二叉树的高度

（9）在二叉树上插入一个结点

（10）拷贝构造函数

（11）赋值重载运算

（12）析构函数

（13）从两个序列创建二叉树

选做：

对二叉查找树做上述工作，且增加以下操作：插入、删除给定键的元素、查找目标键。

## 概要设计

### 内容理解

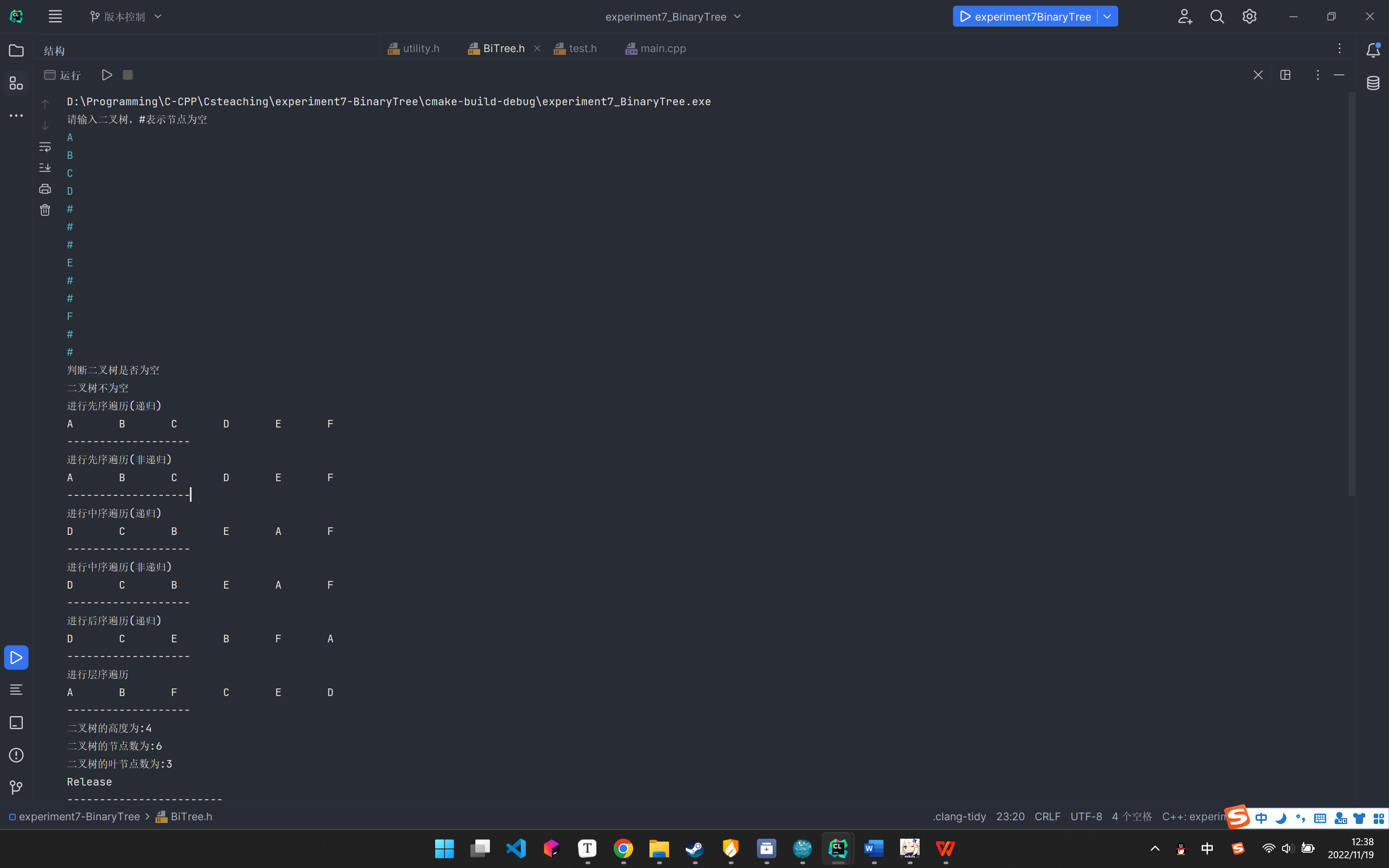
实验要求我们实现一棵二叉树，二叉树采用链式结构存储.

### 功能列表

1. 构造函数
2. 析构函数
3. 拷贝构造函数
4. 判断二叉树是否为空
5. 先序遍历，中序遍历和后续遍历的递归以及迭代的函数
6. 获取二叉树高度，节点数，叶节点数
7. 在二叉树中插入元素

### 程序运行的界面设计

输入二叉树，



电脑萤幕的截图

描述已自动生成

### 总体设计思路

程序需要对二叉树进行多种操作，因此将多种操作放在了class BiTree中。二叉树的节点为struct BiNode中。

#### Struct BiNode

template<typename DataType>

struct BiNode

{

DataType data; //节点的值

BiNode<DataType>\* left\_child; //左孩子

BiNode<DataType>\* right\_child; //右孩子

};

#### Class BiTree



## 三、详细设计

### （1）构造函数

需要用到private中的Create()。其中Create()的作用是不断输入节点，并且以递归的方式来构造一棵二叉树，并且最终返回二叉树的根节点。

### （2）析构函数

需要用到private中的Release()。其中Release()的作用是后续遍历二叉树，并且每次都删除当前的节点。选用后续遍历的原因是为了防止将根节点销毁后找不到其左右子树。

### （3）拷贝构造函数

需要用到private中的copy()。其中copy()的作用是通过递归的方式，遍历原有的二叉树的节点，然后进行深拷贝进行拷贝构造.

### （4）bool empty()

获取根节点，判断根节点是否为nullptr

### (5)RecursionPreOrder(),RecursionInOrder(),RecursionPostOrder()

均使用递归的方式，分别对二叉树实现了前序遍历，中序遍历和后续遍历。

### （6）PreOrder(),InOrder(),PostOrder()

使用迭代方式对二叉树进行遍历。其中都用到了std::stack()来储存节点.

### （7）LevelOrder()

对二叉树进行层序遍历，其中用到了std::queue来储存二叉树每一层的节点，对二叉树每层的节点不断的push()和pop()，从而实现层序遍历.

### （8）GetHeight()

需要用到private中的GetHeight(BiNode\* bt)。int GetHeight(BiNode<DataType>\*bt)是用递归的方式来遍历二叉树，并且求出二叉树的高度。

### （9）GetTreeSize(BiNode<DataType>\* node)

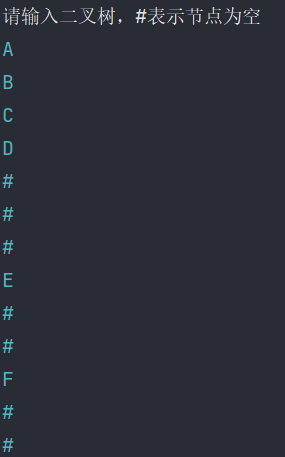
通过递归的方式遍历二叉树，从而获得二叉树的大小。

### （10）GetLeafSize(BiNode<DataType>\* node)

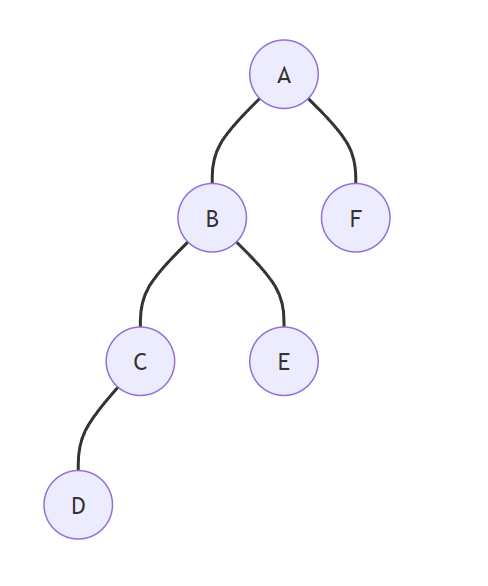
使用递归来遍历二叉树，若某节点左右孩子均为空，则说明该节点为叶节点，从而来计算二叉树的叶节点树。

## 四、实验结果

### （1）测试输入



输入的二叉树:



### （2）测试目的:测试二叉树构建是否正确，以及各函数是否正确.

### （3）正确输出:

前序遍历:A B C D E F

中序遍历:D C B E A F

后序遍历::D C E B F A

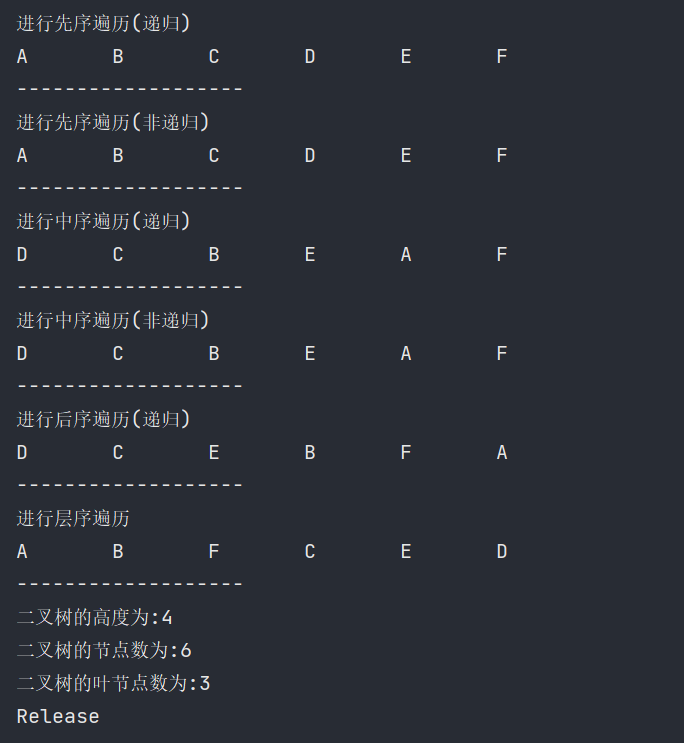
层序遍历:A B F C E D

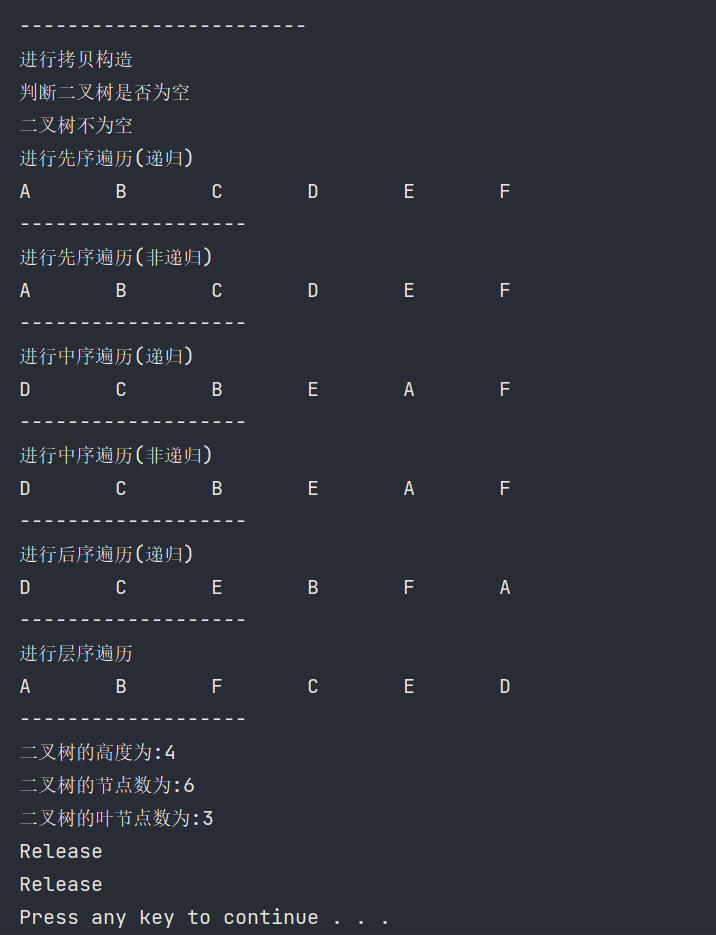
二叉树高度: 4

二叉树节点数: 6

二叉树叶节点数: 3

### （4）实际输出





### （5）测试结果:**测试通过**

## 五、实验分析与探讨

### （1）测试结果分析:

二叉树的许多需要涉及到遍历的操作的时间复杂度一般均为，如前序，中序以及后序遍历，求二叉树的高度，求二叉树的节点数等操作。而二叉树的插入操作时间复杂度一般为，其中为树的高度，并且。

二叉树的大多数操作都有递归和迭代两个解决方式，比如遍历操作，求树高操作，求节点数操作。其中递归代码一般都是利于理解，但是由于递归会用到堆栈区，且可能会出现尾递归现象，因此递归的性能一般较差。而迭代的代码一般较为复杂，但是迭代的运行效率一般都大于递归.

### （2）实验中的一些问题

对于二叉树的每一个节点，其左孩子的值，右孩子的值与节点自己的值之间没有关系，因此普通的二叉树的插入，查找，删除操作变得极为繁琐且没有意义（因为此时对二叉树的插入，删除以及查找需要的时间复杂度均为，与普通的单链表相比没有优势）因此本程序未实现对于二叉树的插入，删除以及查找操作，而是将这三个操作留在了二叉查找树（Binary Search Tree，简称BST）中实现。

## 六、小结

到此，二叉树的实验已经完成，但是二叉树的插入，删除和查询操作并未实现，本人将这三个操作放在BST中实现。

补充说明:

1.执行程序时打开main.exe来运行，或者打开main.cpp来运行。

## 附录：源代码

### \*（1）实验环境:



编译器:mingw

gcc version 8.1.0 (x86\_64-posix-seh-rev0，Built by NinGW-w64 project)

C++版本:C++ 17

### （2）源代码:

#### 1. utility.h

#include<iostream>  
#include<queue>  
#include<stack>  
using namespace std;

#### 2.BiTree.h

#include "utility.h"  
  
template<typename DataType>  
struct BiNode  
{  
 DataType data;  
 BiNode<DataType>\* left\_child;  
 BiNode<DataType>\* right\_child;  
};  
  
template<typename DataType>  
struct element  
{  
 BiNode<DataType>\* ptr;  
 int flag;  
};  
  
template<typename DataType>  
class BiTree  
{  
public:  
 BiTree()  
 {  
 cout<<"请输入二叉树，#表示节点为空"<<endl;  
 root = Creat();  
 }  
  
 ~BiTree()  
 {  
 cout<<"Release"<<endl;  
 Release(root);  
 }  
  
 BiTree(BiTree<DataType> const &T);  
  
 BiNode<DataType>\* GetRoot()  
 {return this->root;}  
  
 bool Empty()  
 { return !(root == nullptr) ;}  
  
 void RecursionPreOrder()  
 {RecursionPreOrder(root);}  
  
 void RecursionInOrder()  
 {RecursionInOrder(root);}  
  
 void RecursionPostOrder()  
 {RecursionPostOrder(root);}  
  
 void LevelOrder();  
  
 void PreOrder()  
 { PreOrder(root);}  
  
 void InOrder()  
 {InOrder(root);}  
  
 void PostOrder()  
 {PostOrder();}  
  
 int GetHeight()  
 {return GetHeight(root);}  
  
 int GetTreeSize(BiNode<DataType>\* node);  
  
 int GetLeafSize(BiNode<DataType>\* node);  
  
 void Insert(BiNode<DataType>\* insert\_node);  
  
private:  
 BiNode<DataType> \* Creat();  
  
 void Release(BiNode<DataType>\* bt);  
  
 void RecursionPreOrder(BiNode<DataType>\* bt);  
  
 void RecursionInOrder(BiNode<DataType>\* bt);  
  
 void RecursionPostOrder(BiNode<DataType>\* bt);  
  
 BiNode<DataType>\* root;  
  
 void PreOrder(BiNode<DataType>\* bt);  
  
 void InOrder(BiNode<DataType>\* bt);  
  
 void PostOrder(BiNode<DataType>\*bt);  
  
 int GetHeight(BiNode<DataType>\*bt);  
  
 BiNode<DataType> \* copy(BiNode<DataType>\* node);  
};  
  
template<typename DataType>  
BiNode<DataType> \* BiTree<DataType>::Creat()  
{  
 BiNode<DataType>\* bt;  
 char ch;  
 cin>>ch;  
 if(ch =='#')  
 {bt = nullptr;}  
 else  
 {  
 bt = new BiNode<DataType>;  
 bt->data = ch;  
 bt->left\_child = Creat();  
 bt->right\_child = Creat();  
 }  
 return bt;  
}  
  
template<typename DataType>  
void BiTree<DataType>::Release(BiNode<DataType> \*bt) *//销毁二叉树采用后序遍历，其原因是为了防止将根节点销毁后找不到其左右子树。*{  
 if(bt == nullptr)  
 {return;}  
 else  
 {  
 Release(bt->left\_child);  
 Release(bt->right\_child);  
 delete bt;  
 }  
}  
  
template<typename DataType>  
void BiTree<DataType>::RecursionPreOrder(BiNode<DataType> \*bt)  
{  
 if(bt == nullptr)  
 {return;}  
 else  
 {  
 cout<<bt->data<<"\t";  
 RecursionPreOrder(bt->left\_child);  
 RecursionPreOrder(bt->right\_child);  
 }  
}  
  
template<typename DataType>  
void BiTree<DataType>::RecursionInOrder(BiNode<DataType> \*bt)  
{  
 if(bt == nullptr)  
 { return;}  
 else  
 {  
 RecursionInOrder(bt->left\_child);  
 cout<<bt->data<<"\t";  
 RecursionInOrder(bt->right\_child);  
 }  
}  
  
template<typename DataType>  
void BiTree<DataType>::RecursionPostOrder(BiNode<DataType> \*bt)  
{  
 if(bt == nullptr)  
 { return;}  
 else  
 {  
 RecursionPostOrder(bt->left\_child);  
 RecursionPostOrder(bt->right\_child);  
 cout<<bt->data<<"\t";  
 }  
}  
  
template<typename DataType>  
void BiTree<DataType>::LevelOrder()  
{  
 queue<BiNode<DataType>\*>q;  
 if(root!= nullptr)  
 {  
 q.push(root);  
 }  
 while (q.empty() == false)  
 {  
 auto temp = q.front();  
 if(temp->left\_child != nullptr)  
 {  
 q.push(temp->left\_child);  
 }  
 if(temp->right\_child != nullptr)  
 {  
 q.push(temp->right\_child);  
 }  
 cout<<temp->data<<"\t";  
 q.pop();  
 }  
}  
  
template<typename DataType>  
void BiTree<DataType>::PreOrder(BiNode<DataType> \*bt)  
{  
 stack<BiNode<DataType>\*> s;  
 while(bt!= nullptr || s.empty()== false)  
 {  
 while(bt!= nullptr)  
 {  
 cout<<bt->data<<"\t";  
 s.push(bt);  
 bt = bt->left\_child;  
 }  
 if(s.empty() == false)  
 {  
 bt = s.top();  
 s.pop();  
 bt = bt->right\_child;  
 }  
 }  
}  
  
template<typename DataType>  
void BiTree<DataType>::InOrder(BiNode<DataType> \*bt)  
{  
 stack<BiNode<DataType>\*> s;  
 while(bt!= nullptr || s.empty()== false)  
 {  
 while(bt!= nullptr)  
 {  
 s.push(bt);  
 bt = bt->left\_child;  
 }  
 if(s.empty() == false)  
 {  
 bt = s.top();  
 cout<<bt->data<<"\t";  
 s.pop();  
 bt = bt->right\_child;  
 }  
 }  
}  
  
template<typename DataType>  
void BiTree<DataType>::PostOrder(BiNode<DataType> \*bt)  
{  
 element<DataType> S[100]; *//顺序栈，最多100个元素* int top = -1; *//顺序栈初始化* while (bt != nullptr || top != -1) *//两个条件都不成立才退出循环* {  
 while (bt != nullptr)  
 {  
 top++;  
 S[top].ptr = bt; S[top].flag = 1; *//bt连同标志flag入栈* bt = bt->left\_child;  
 }  
 while (top != -1 && S[top].flag == 2)  
 {  
 bt = S[top--].ptr;  
 cout << bt->data;  
 }  
 if (top != -1) {  
 S[top].flag = 2;  
 bt = S[top].ptr->right\_child;  
 }  
 else  
 bt = nullptr;  
 }  
}  
  
template<typename DataType>  
int BiTree<DataType>::GetHeight(BiNode<DataType>\*bt)  
{  
 int left\_height;  
 int right\_height;  
 if(bt== nullptr)  
 {  
 return 0;  
 }  
 else  
 {  
 left\_height = GetHeight(bt->left\_child);  
 right\_height = GetHeight(bt->right\_child);  
 return left\_height>right\_height?++left\_height:++right\_height;  
 }  
}  
  
template<typename DataType>  
int BiTree<DataType>::GetTreeSize(BiNode<DataType> \*node)  
{  
 if(node == nullptr){return 0;}  
 else  
 {  
 if(node->left\_child == nullptr && node->right\_child == nullptr)  
 {  
 return 1;  
 }  
 }  
 return 1 + GetTreeSize(node->left\_child) + GetTreeSize(node->right\_child);  
}  
  
template<typename DataType>  
int BiTree<DataType>::GetLeafSize(BiNode<DataType> \*node)  
{  
 if(node == nullptr)  
 {  
 return 0;  
 }  
 else  
 {  
 if(node->left\_child == **NULL** && node->right\_child == **NULL**)  
 {  
 return 1;  
 }  
 }  
 return GetLeafSize(node->left\_child) + GetLeafSize(node->right\_child);  
}  
  
template<typename DataType>  
void BiTree<DataType>::Insert(BiNode<DataType> \*insert\_node)  
{  
 auto p = root;  
 if(p == nullptr)  
 {  
 p = insert\_node;  
 p->left\_child = nullptr;  
 p->right\_child = nullptr;  
 }  
 else  
 {  
 while(p!= nullptr)  
 {  
 p = p->left\_child;  
 }  
 p = insert\_node;  
 p->left\_child = nullptr;  
 p->right\_child = nullptr;  
 }  
}  
  
template<typename DataType>  
BiTree<DataType>::BiTree(BiTree<DataType> const &T)  
{  
 this->root = copy(T.root);  
}  
  
template<typename DataType>  
BiNode<DataType> \* BiTree<DataType>::copy(BiNode<DataType>\* node)  
{  
 BiNode<DataType>\* new\_root;  
 if(node == nullptr)  
 {  
 return nullptr;  
 }  
 else  
 {  
 auto left\_node = copy(node->left\_child);  
 auto right\_node = copy(node->right\_child);  
 new\_root = new BiNode<DataType>;  
 new\_root->left\_child = left\_node;  
 new\_root->right\_child = right\_node;  
 new\_root->data = node->data;  
 return new\_root;  
 }  
}

#### 3.test.h

#include "utility.h"  
#include "BiTree.h"  
  
template<typename DataType>  
void Order(BiTree<DataType> tree)  
{  
 cout<<"判断二叉树是否为空"<<endl;  
 if(tree.Empty() == 0){cout<<"二叉树为空"<<endl;}  
 else  
 {  
 cout<<"二叉树不为空"<<endl;  
  
 cout<<"进行先序遍历(递归)"<<endl;  
 tree.RecursionPreOrder();  
 cout<<""<<endl;  
 cout<<"-------------------"<<endl;  
  
 cout<<"进行先序遍历(非递归)"<<endl;  
 tree.PreOrder();  
 cout<<endl;  
 cout<<"-------------------"<<endl;  
  
 cout<<"进行中序遍历(递归)"<<endl;  
 tree.RecursionInOrder();  
 cout<<endl;  
 cout<<"-------------------"<<endl;  
  
 cout<<"进行中序遍历(非递归)"<<endl;  
 tree.InOrder();  
 cout<<endl;  
 cout<<"-------------------"<<endl;  
  
 cout<<"进行后序遍历(递归)"<<endl;  
 tree.RecursionPostOrder();  
 cout<<endl;  
 cout<<"-------------------"<<endl;  
  
 cout<<"进行层序遍历"<<endl;  
 tree.LevelOrder();  
 cout<<endl;  
 cout<<"-------------------"<<endl;  
  
 cout<<"二叉树的高度为:"<<tree.GetHeight()<<endl;  
 cout<<"二叉树的节点数为:"<<tree.GetTreeSize(tree.GetRoot())<<endl;  
 cout<<"二叉树的叶节点数为:"<<tree.GetLeafSize(tree.GetRoot())<<endl;  
  
 }  
}  
  
void test()  
{  
 BiTree<char>T1{ };  
 Order(T1);  
 cout<<"------------------------"<<endl;  
 cout<<"------------------------"<<endl;  
 cout<<"进行拷贝构造"<<endl;  
 const BiTree<char>&T2(T1);  
 Order(T2);  
}

#### 4 main.cpp

#include "test.h"  
  
int main()  
{  
 test(); *//测试函数* system("pause");  
 return 0;  
}