《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软件工程 | 姓名 | 梅子羽 | 学号 | 2327406107 |
| 实验布置日期 | | 2024.11.5 | | 提交  日期 | 2024.11.19 | | 成绩 |  |

课程实践实验5：二叉树操作

## 一、问题描述及要求

题目：用一个菜单驱动的演示程序说明二叉树的使用

[问题描述]

二叉树的元素由单个键组成，键为单个字符。

程序运行时，输出帮助信息，信息描述输入某个数字代表某个菜单操作，比如，输入数字1，代表进行建树操作；输入2代表进行某个遍历操作等。如果没有进行构建就进行遍历操作，可以给用户先构建某个二叉树后再进行后续的操作

注：二叉树要采用链式存储结构。

二叉树的操作至少包括：

（1）构造空二叉树

（2）输入前序、中序序列后创建二叉树

（3）判别二叉树是否为空

（4）先序遍历、中序遍历、后序遍历递归实现

（5）层次遍历

（6）先序遍历、中序遍历、后序遍历非递归实现（选做）

（7）求二叉树的结点数、叶子结点数

（8）清空已有二叉树

（9）求二叉树的高度、宽度

（10）在二叉树上较矮的子树上插入一个结点，如果子树高度相同，则插在左子树

（11）判断两棵二叉树是否互为镜像

（12）拷贝构造函数

（13）赋值重载运算

（14）析构函数

## 二、概要设计

#### （1）对实验内容的理解

本实验的内容是设计并实现一个二叉树类BinaryTree，实现基本的二叉树操作（如插入节点、遍历、删除树、查找树的节点个数、叶子节点个数、树的高度、宽度等），并支持树的创建、复制以及比较两个树是否是镜像等功能。此外，实验还涉及到树的前序、中序、后序遍历以及层次遍历的实现。

二叉树是一个基础的树形数据结构，在许多算法和数据处理中都得到了广泛的应用。本实验旨在帮助学生理解和掌握树形结构的基础概念、二叉树的操作以及递归和非递归算法。

#### （2）系统的功能列表

系统的功能可以按照模块化的方式进行设计，功能清单如下：

1. **树的创建**
   * 通过给定的前序遍历和中序遍历序列创建二叉树。
2. **树的遍历**
   * 前序遍历
   * 中序遍历
   * 后序遍历
   * 层次遍历
3. **树的基本操作**
   * 判断树是否为空
   * 统计树的节点个数
   * 统计树的叶子节点个数
   * 统计树的高度
   * 统计树的宽度
4. **树的修改操作**
   * 插入一个新节点
   * 删除整棵树
5. **树的比较**
   * 判断两棵树是否是镜像树
6. **树的复制与赋值**
   * 复制一棵树
   * 使用赋值运算符进行树的赋值

#### （3）程序运行的界面设计

界面设计是用户与程序交互的方式，在本实验中，程序运行是基于命令行界面的，用户通过输入不同的数字来选择不同的操作。具体的交互界面如下：

==== Binary Tree Operations ====

1. Build tree using preorder and inorder sequences

2. Preorder traversal (recursive)

3. Inorder traversal (recursive)

4. Postorder traversal (recursive)

5. Level order traversal

6. Count nodes

7. Count leaves

8. Count height

9. Count width

10. Insert a new node in the tree

11. Check if two trees are mirror images

12. Copy binary tree

13. Assign operator (=) for binary tree

14. Clear the tree

0. Exit

=================================

用户通过输入选项（数字）来选择对应的功能，例如选择1可以输入前序和中序序列来构建树，选择2可以进行前序遍历。

#### （4）总体设计思路及数据结构选择

##### 设计思路：

本系统采用面向对象的编程方法，通过设计二叉树类BinaryTree来封装所有二叉树操作。二叉树的节点结构使用链式存储方式，每个节点包含一个值（val）、左子节点指针（left）和右子节点指针（right）。主要操作如树的遍历、节点插入、树的高度计算等都通过递归算法实现，而层次遍历则采用队列实现。

##### 数据结构：

* **ListNode**：每个节点结构，包含节点的值及其左右子节点的指针。
* **BinaryTree**：二叉树类，管理树的根节点，并提供对树的各种操作。

##### 类设计：

1. **ListNode<DataType>**：
   * 成员变量：
     + val：存储节点的数据。
     + left、right：指向左右子节点的指针。
   * 构造函数：
     + 默认构造函数：将节点的值设为默认值，左右指针为空。
     + 参数化构造函数：根据给定值初始化节点。
2. **BinaryTree<DataType>**：
   * 成员变量：
     + root：指向二叉树根节点的指针。
   * 成员函数：
     + createByPreInOrder：根据给定的前序和中序序列构建二叉树。
     + 各种遍历函数：preOrderTraversal、inOrderTraversal、postOrderTraversal、levelTraversal。
     + 树的操作函数：countNodes、countLeaves、countHeight、countWidth。
     + insertNode：插入新节点。
     + isMirrorTree：判断两棵树是否是镜像树。
     + copyBinaryTree：复制一棵树。

##### 递归与非递归算法：

* **递归算法**：主要用于树的遍历（前序、中序、后序遍历）和一些递归操作（如计算树的高度、删除树等）。
* **非递归算法**：用于层次遍历，采用队列数据结构进行广度优先遍历。

#### （5）程序结构设计

* **文件结构**：
  + BinaryTree.h：定义二叉树类的头文件，包含成员函数声明和数据成员声明。
  + ListNode.h：定义二叉树节点的结构体。
  + BinaryTree.cpp：实现二叉树类的成员函数。
  + main.cpp：包含程序的主函数，提供用户接口并调用二叉树的操作。
* **各部分关系描述**：
  + BinaryTree 类通过指向 ListNode 对象的指针来管理树的节点。二叉树的遍历和其他操作主要通过递归调用来实现。
  + ListNode 用于存储树的每个节点的数据及其左右子树的信息。

##### 数据结构和算法的选择理由：

1. **链式存储结构**：使用链式结构存储二叉树的节点，使得插入、删除节点操作高效且灵活。
2. **递归遍历算法**：二叉树的遍历自然适合递归处理，能够简洁地表达每种遍历方式。
3. **队列（层次遍历）**：层次遍历采用队列存储节点，确保按层次顺序访问所有节点。
4. **镜像树判断**：通过递归算法，递归地比较两棵树的左右子树是否对称，实现镜像树的判断。

通过上述设计，我们能够高效地实现二叉树的基本操作，并且保证系统的可扩展性，能够根据实际需求添加新的树操作。

## 三、详细设计

在本部分中，我们将重点分析程序的主函数、关键算法的设计，并对程序的难点部分进行详细的分析。

#### 1. ****主函数设计****

主函数作为程序的入口，负责控制程序的整体流程。其主要功能是展示操作菜单，接收用户的输入，并根据用户选择调用相应的功能。

* **主函数的基本结构：**
  + **菜单展示**：每次循环都会显示操作菜单，提供给用户选择各种功能。
  + **选择输入**：通过cin读取用户的输入，选择对应的操作。
  + **功能调用**：根据用户的选择调用相应的二叉树操作，例如创建树、遍历树、插入节点等。
  + **循环操作**：程序会持续运行，直到用户选择退出选项（选项0）。
  + **空树检测**：每次进行树的操作时（如遍历、节点计数等），都会先检查树是否为空，若为空则提示用户构建树。

主函数的流程图可以描述为：

1. 显示菜单。
2. 获取用户输入。
3. 根据输入执行相应的操作（如创建树、遍历、插入节点等）。
4. 如果用户选择退出，程序结束；否则，重复上述步骤。

#### 2. ****关键算法设计****

在程序中，最核心的操作是二叉树的创建、遍历、插入和其他基本操作。我们将依次分析以下关键算法的设计和实现：

##### 2.1 ****二叉树的创建（****createByPreInOrder****）****

该算法通过前序和中序遍历序列来创建二叉树。利用前序遍历的第一个元素作为根节点，并根据中序遍历的信息来确定左右子树的分割。

* 该操作会递归进行，首先通过前序序列的第一个元素确定根节点，再通过中序序列找到该根节点的位置，进而将树分成左右两部分，分别构建左子树和右子树。

##### 2.2 ****二叉树遍历（前序、中序、后序）****

对于不同的遍历操作，分别设计了递归和非递归的方法：

* **前序遍历**：先访问根节点，然后递归遍历左子树和右子树。
* **中序遍历**：先递归遍历左子树，然后访问根节点，最后递归遍历右子树。
* **后序遍历**：先递归遍历左子树和右子树，然后访问根节点。

这些遍历的实现方式为递归，简单且直观。

##### 2.3 ****层次遍历（****levelTraversal****）****

层次遍历使用队列来实现。队列存储每一层的节点，先访问根节点，再访问根节点的子节点。每当访问完一层的所有节点时，就处理下一层的节点。

##### 2.4 ****节点计数、叶子计数、树高、树宽****

* **节点计数**：递归地遍历树的每个节点，统计节点总数。
* **叶子计数**：叶子节点是没有子节点的节点，递归遍历时检查每个节点是否有左右子节点，若都没有，则为叶子节点。
* **树的高度**：树的高度是根节点到最深叶子节点的路径长度。通过递归计算左右子树的高度，返回更大值加1。
* **树的宽度**：树的宽度是某一层节点的最大数目。层次遍历时统计每一层的节点数，并更新最大值。

##### 2.5 ****插入新节点（****insertNode****）****

插入新节点操作通常会在树的某个位置进行。在此程序中，插入节点是根据一定的规则来完成的。例如，可以将新节点插入到最底层的某个空位置。

##### 2.6 ****镜像树判断（****isMirrorTree****）****

判断两棵树是否为镜像树的操作是比较复杂的。它递归地检查两棵树的结构是否对称：

* 对于树1的左子树和树2的右子树，树1的右子树和树2的左子树进行递归判断，直到所有节点的左右结构一致。

##### 2.7 ****树的复制（****copyBinaryTree****）****

树的复制通常是通过递归实现的，对于每一个节点，都创建一个新的节点，并将其子树递归复制到新节点上。

#### 3. ****难点分析****

* **递归实现的正确性**：大部分的操作（如树的遍历、树的高度计算、树的复制）都使用了递归方法。递归需要注意终止条件的设置，特别是在处理空树或叶子节点时，防止程序出现死循环或栈溢出。
* **镜像树判断**：判断两棵树是否镜像的算法是一个典型的树形递归问题。需要确保树的结构完全对称，检查每对节点的左右子树是否递归对称。
* **树的复制**：树的复制是一个较为复杂的操作，需要正确地为每个节点分配内存，并且递归地复制所有子节点。
* **层次遍历**：虽然层次遍历可以用队列来实现，但处理时需要小心队列的使用，确保每层的节点被正确地访问和统计。

#### 4. ****类设计****

在本程序中，主要有一个类 BinaryTree 用来实现二叉树的各种操作。该类的主要成员函数包括：

* createByPreInOrder: 通过前序和中序序列创建树。
* preOrderTraversal, inOrderTraversal, postOrderTraversal: 分别实现前序、中序、后序的递归遍历。
* levelTraversal: 实现层次遍历。
* countNodes, countLeaves, countHeight, countWidth: 分别实现计算节点数、叶子节点数、树的高度和树的宽度。
* insertNode: 插入新节点。
* isMirrorTree: 判断两棵树是否是镜像树。
* copyBinaryTree: 复制树。

每个函数的设计都围绕着树的基本操作进行，所有操作都通过递归实现，简洁且高效。

#### 5. ****程序结构设计****

该程序采用了模块化的结构，主要分为以下几个文件：

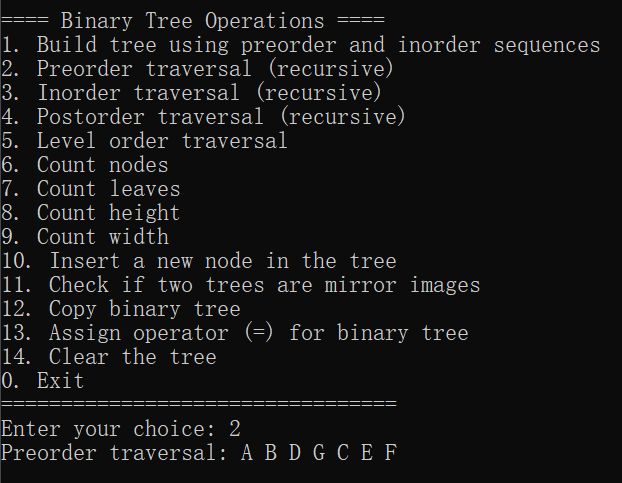
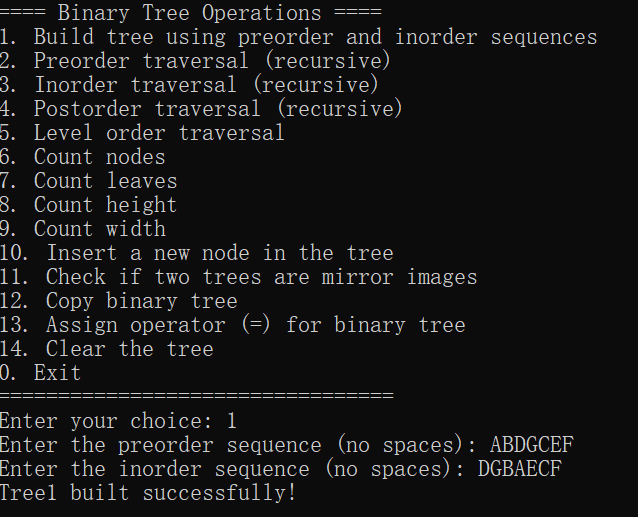
* BinaryTree.h：定义二叉树类及其成员函数。
* BinaryTree.cpp：实现二叉树的具体功能。
* main.cpp：包含主函数和用户界面，负责菜单展示和与用户的交互。

程序结构是清晰的，代码的可维护性较强，通过合理的函数划分，确保每个功能模块独立，便于扩展和修改。

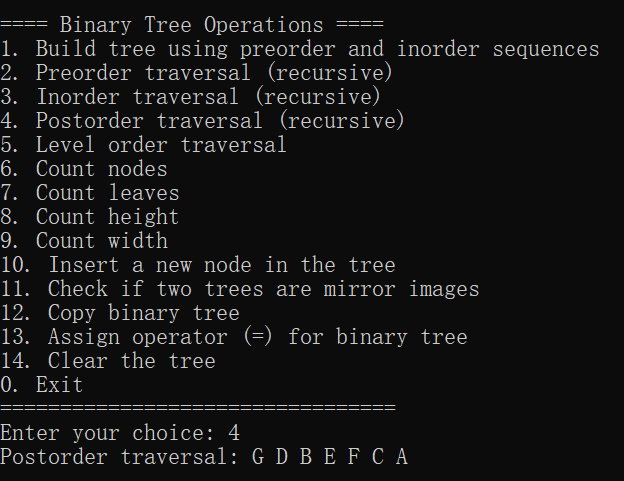
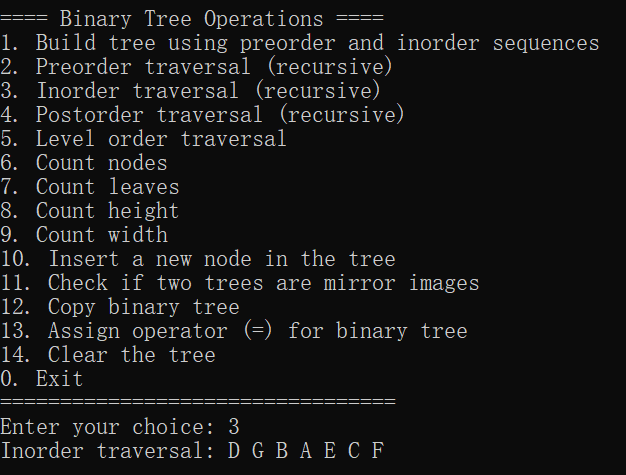
## 四、实验分析与探讨

测试样例一：tree1：前序：ABDGCEF 中序：DGBAECF，覆盖功能1-10，检查程序完成度

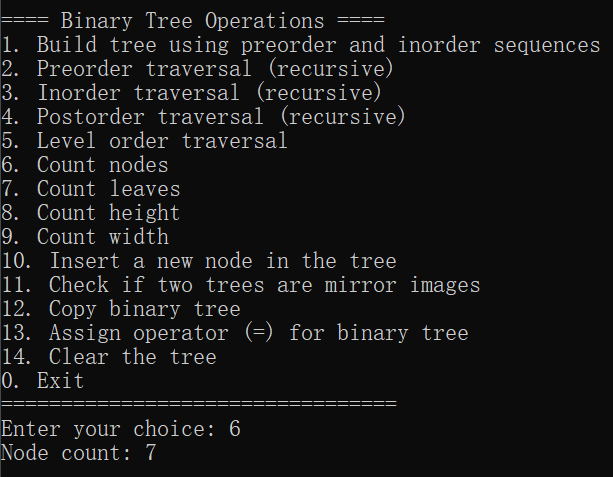
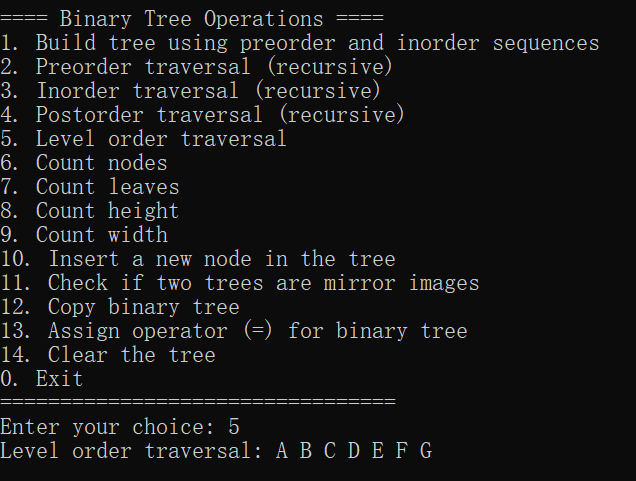
功能1,2：根据前序和中序序列构造二叉树，构造成功；实现前序遍历二叉树，遍历正确



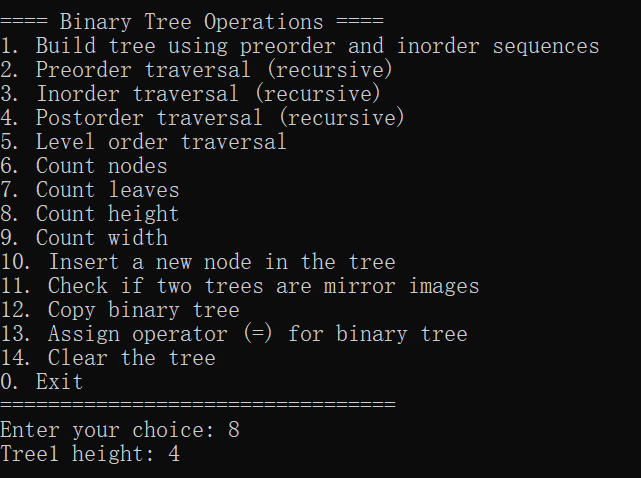
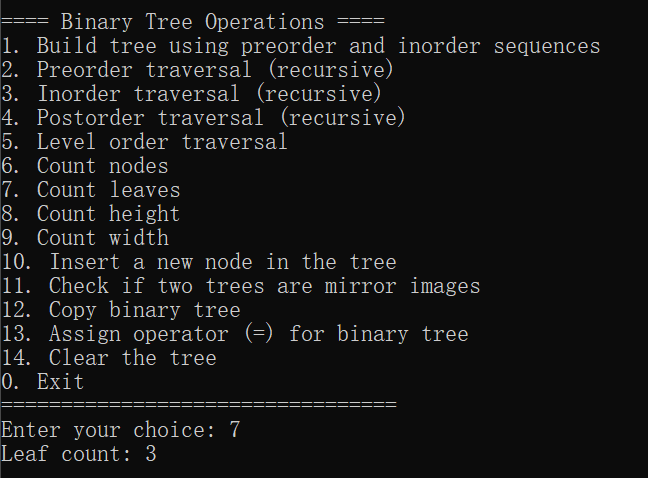
功能3,4：实现中序，后序遍历二叉树，遍历正确



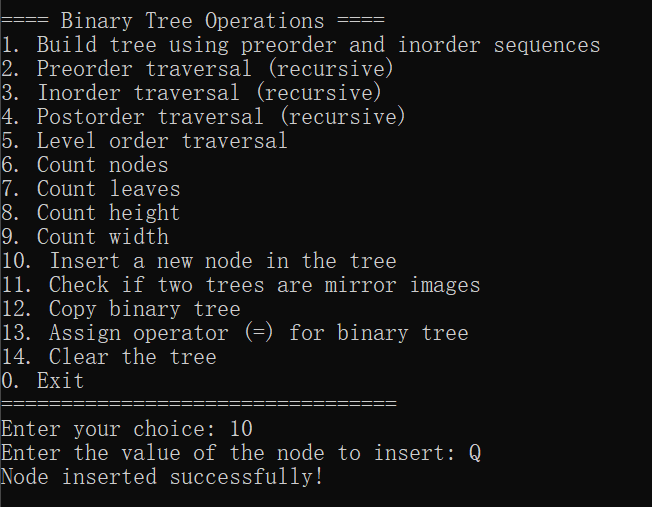
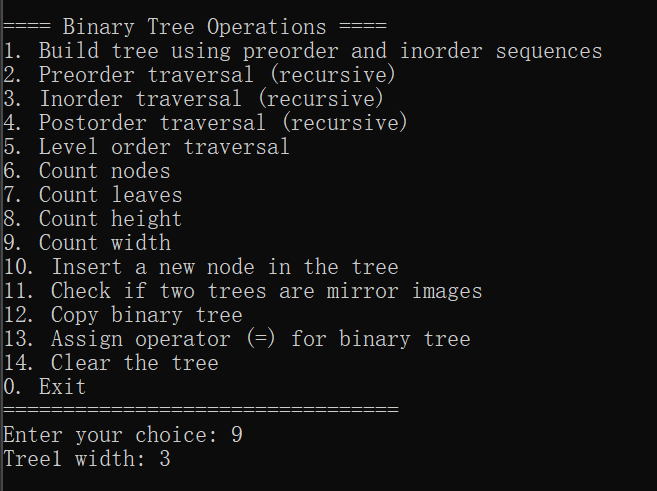
功能5,6：实现层序遍历二叉树，遍历正确；计算树节点，结果正确



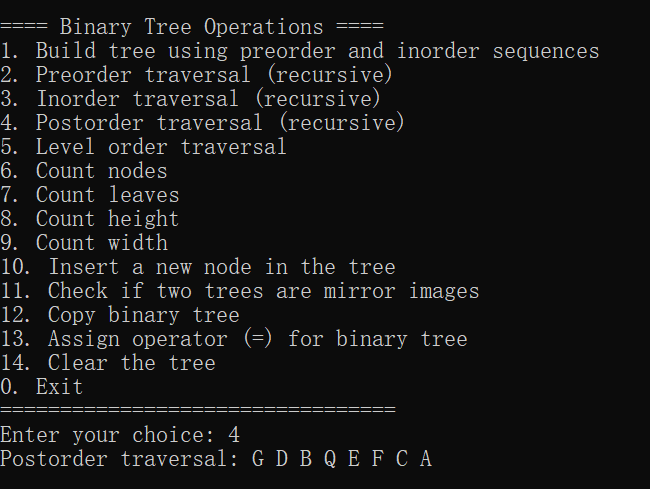
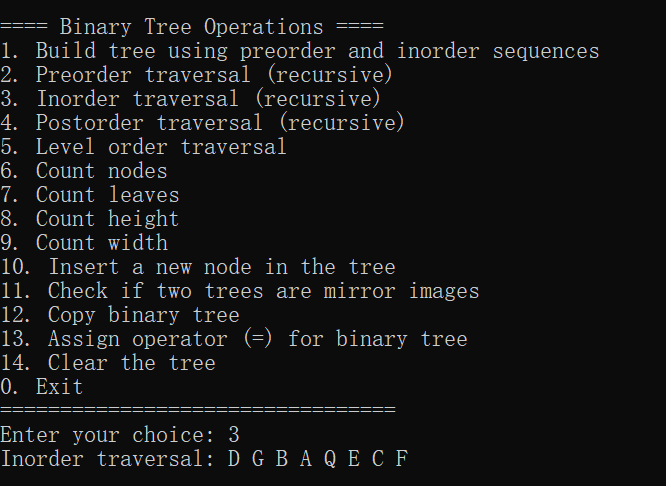
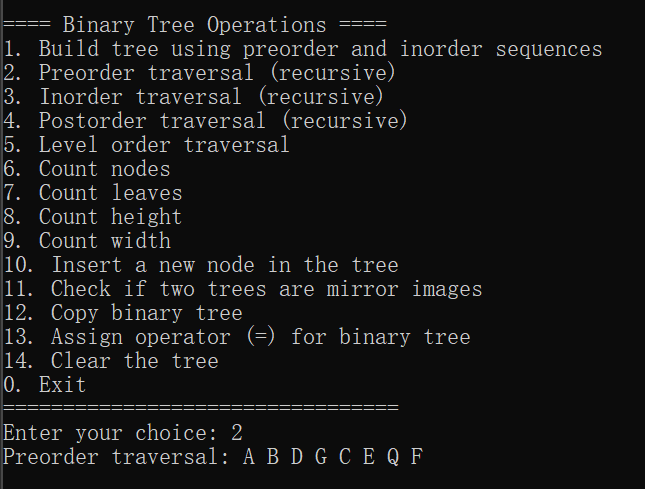
功能7,8：计算叶子节点，树高度，结果正确



功能9,10：计算树宽度，结果正确；实现在二叉树上较矮的子树上插入一个结点，如果子树高度相同，则插在左子树，插入正确

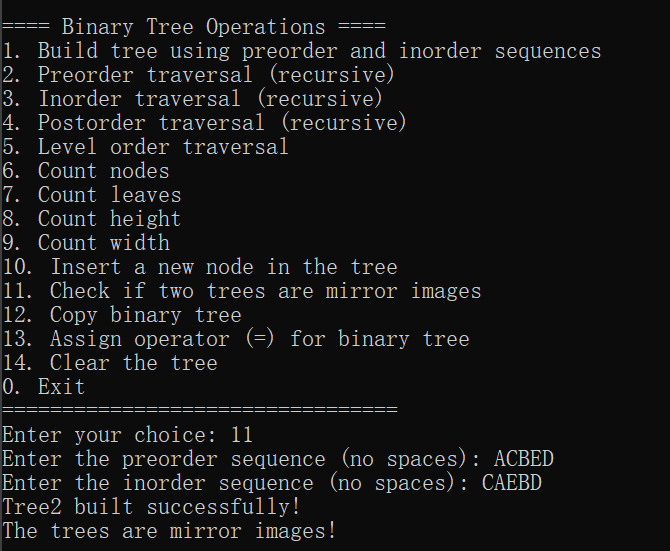
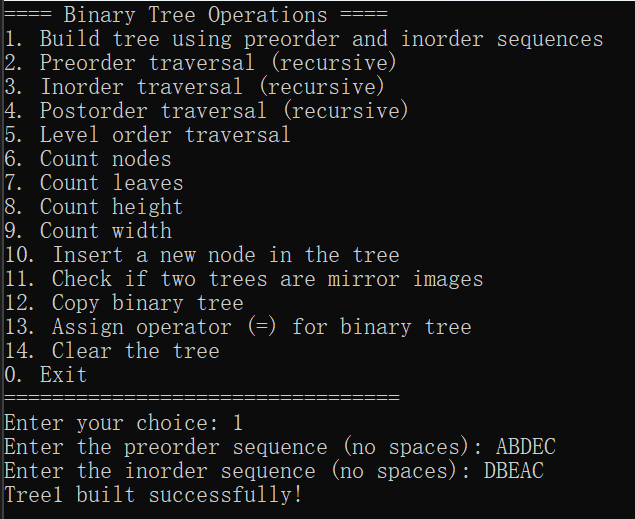


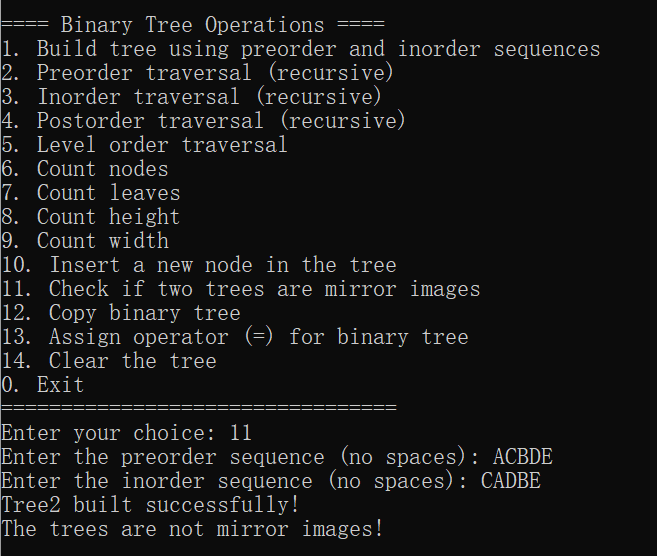
检验接点插入正确，再重复2,3,4，后面功能也测试了但图太多就不放置了，插入成功



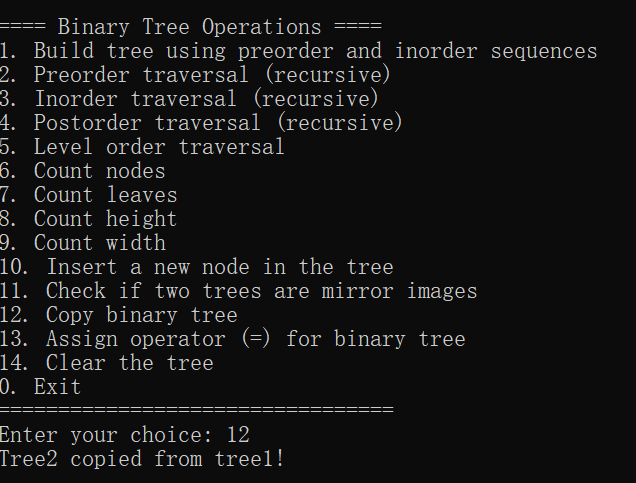
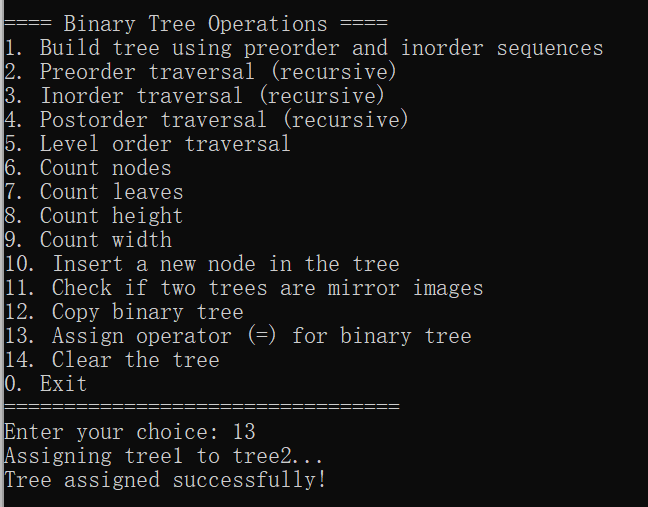
测试样例二：tree1：前序：ABDEC 中序：DBEAC，覆盖功能11-14，检查程序完成度

功能11：判断是否为镜像树，构造镜像树tree2：前序：ACBED 中序：CAEBD，输出正确

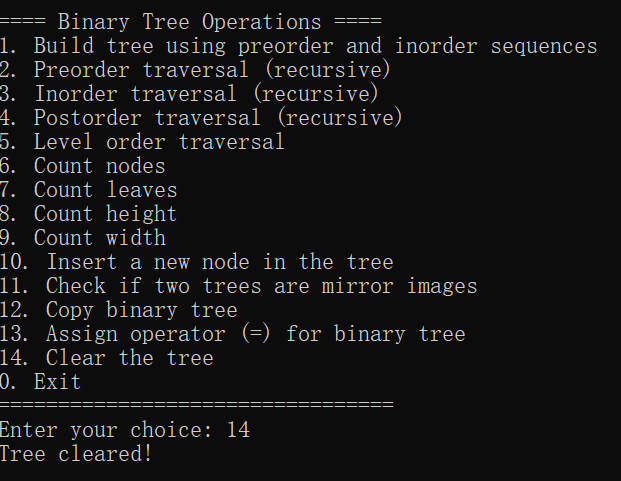
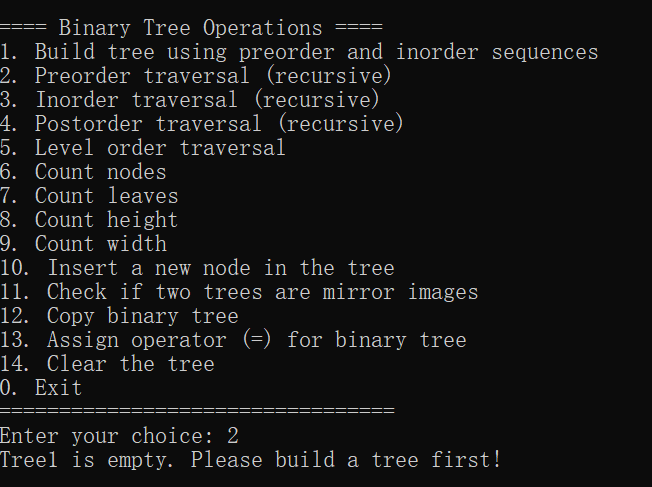
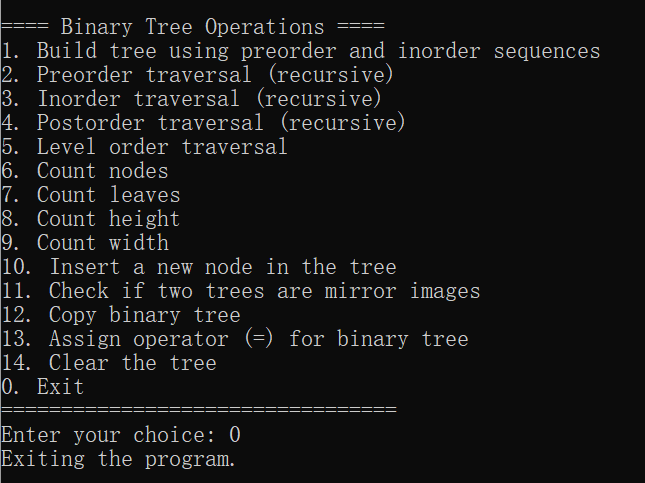


 再构造非镜像树tree2：前序：ACBDE 中序：CADBE，输出正确

功能12，13：深拷贝复制数，浅拷贝赋值树，都是将tree1复制到tree2，操作成功

功能14,0：执行析构函数，删除tree1并退出程序，期间用功能2检验是否删除，均正确

实验分析：此次实验针对二叉树类实现了各种各样的功能，核心算法思想是递归，二叉树的存储结构是之前学习的链表，不难发现时间复杂度主要取决于各类函数中递归算法的设计，空间复杂度主要来源于递归栈和链表指针的结构性开销。所以当明确分析对象后，本次实验的复杂度分析主要还是取决于依据前序中序序列构造二叉树这一函数功能，**时间复杂度**: 最坏情况下（完全二叉树），每次递归查找 O(n)，递归深度为 n，总复杂度为 O(n^2)。**空间复杂度**: 递归调用栈深度为树的高度，最坏情况下为 O(n)（退化为链表）。而平均下来的话，**时间复杂度**: 各操作的时间复杂度大多为O(n)（完全二叉树时有部分为O(logn)。**空间复杂度**: 递归操作的空间复杂度依赖于树的高度，最坏情况下为 O(n)（退化为链表），最好为 O(logn)（完全二叉树）。

## 五、小结

#### 1. ****设计与实现过程总结****

在本次二叉树的实现过程中，程序的设计与开发围绕着二叉树的基本操作展开，包括树的创建、遍历、节点计数、插入、复制、镜像判断等功能。通过设计BinaryTree类及其相关的成员函数，我们实现了二叉树的完整功能，能够满足大部分常见的二叉树操作需求。

* **已完成的工作：**
  + **树的创建**：通过前序和中序序列创建二叉树，构造了一个可以根据用户输入序列自动构建树的功能。
  + **遍历操作**：实现了前序、中序、后序和层次遍历，且支持递归与非递归两种方式。
  + **节点计数、叶子计数、树高、树宽**：实现了二叉树节点、叶子节点、树高和树宽的计算，提供了基本的树形结构信息。
  + **节点插入**：设计了插入新节点的功能，支持在树中动态插入新节点。
  + **镜像树判断与树复制**：完成了镜像树判断功能，并实现了树的深度复制。
  + **界面交互**：通过菜单提供了与用户交互的界面，用户可以选择各种树的操作进行实验。
* **未完成的工作：**
  + **高级优化与性能改进**：虽然程序功能实现较为完整，但在性能上尚有进一步优化的空间。例如，树的高度、宽度等计算可以在遍历过程中优化，避免重复遍历相同的节点。
  + **内存管理**：在树的复制与销毁方面，尽管程序能够正常执行，但可能在处理复杂树结构时存在一定的内存管理问题，需要通过智能指针等机制进一步优化内存管理。
* **选做部分的完成情况：**
  + 程序中涉及到的选做部分（如树的复制、镜像判断等）已经完成，且运行正常。

#### 2. ****程序局限性****

尽管本程序实现了二叉树的多种操作，但仍存在一定的局限性：

* **树的插入逻辑较为简单**：当前插入节点的操作是根据某种简单的规则进行的，实际应用中可能需要更复杂的插入策略（例如平衡二叉树的插入）。
* **内存管理**：程序未使用智能指针或其他现代C++技术来管理内存，在处理大规模树结构时，可能会遇到内存泄漏问题。
* **界面交互**：用户界面较为简单，操作完全通过命令行进行，未来可以进一步优化用户体验，加入图形界面或更复杂的输入输出方式。
* **树的深度较大时的性能问题**：当树的深度非常大时，递归算法可能会面临栈溢出的风险，未来可以考虑使用非递归的遍历方法来解决这一问题。

#### 3. ****展望与进一步学习方向****

* **学习与优化算法**：在未来可以进一步学习树的平衡算法，如AVL树、红黑树等，这些可以提高树的操作效率，尤其是在大量插入和删除操作时。
* **图形界面设计**：目前程序的交互界面较为简单，可以考虑实现一个图形用户界面（GUI）来提升用户体验，让用户更加直观地操作树结构。
* **性能优化**：程序在处理非常大的树时可能会遇到性能瓶颈。可以学习如何进行性能分析和优化，尤其是树的深度计算和节点的动态插入等操作。
* **内存管理**：进一步学习C++中的内存管理技巧，避免出现内存泄漏问题，并优化程序的内存使用。

#### 4. ****输入文件格式与说明****

在程序中，用户输入的树的遍历序列是通过键盘输入的，要求用户按顺序输入没有空格的字符串，例如：

* **前序序列**：ABCD
* **中序序列**：DCBA

输入时，用户需要注意：

* 输入的前序和中序序列必须符合二叉树的结构关系，即前序序列中的根节点必须在中序序列中存在，并且中序序列可以根据根节点的定位将树分为左右子树。
* 由于程序目前仅支持单一的数据格式输入，未来可以考虑扩展支持从文件中读取树的遍历序列。

#### 5. ****感想与收获****

通过本次实验，我不仅加深了对二叉树基本操作的理解，而且提升了自己在数据结构与算法方面的实际编程能力。实现过程中，我遇到了一些递归与内存管理方面的问题，但通过调试和查阅资料，逐渐解决了这些问题。此外，本次实验也让我意识到程序设计中可能会遇到的各种挑战，例如性能问题、用户交互的设计等，未来可以通过进一步学习和实践来提升解决问题的能力。

#### 6. ****建议****

* **代码优化**：可以通过智能指针或其他现代C++技术来改进内存管理，减少内存泄漏的风险。
* **扩展功能**：在二叉树的基本操作基础上，可以加入更多高级的树结构操作，如平衡树的插入、删除等操作，进一步提升程序的功能性与实用性。
* **优化用户界面**：目前的界面仅支持命令行操作，未来可以通过GUI技术提升用户体验。

总体而言，程序已经基本完成预定的功能，但仍有许多方面可以进行优化和扩展，期待在以后的学习中进一步完善该项目。

## 附录：源代码

**1、实验环境：Dev-C++ 5.11 C++11标准**

2、

（1）//ListNode.h

#ifndef LISTNODE\_H

#define LISTNODE\_H

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

template <typename DataType>

struct ListNode

{

DataType val;

ListNode<DataType>\* left;

ListNode<DataType>\* right;

ListNode() : val(), left(nullptr), right(nullptr) {}

ListNode(DataType x) : val(x), left(nullptr), right(nullptr) {}

};

#endif

（2）//BinaryTree.h

#ifndef BINARY\_TREE\_H

#define BINARY\_TREE\_H

#include "ListNode.h"

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

using namespace std;

template <typename DataType>

class BinaryTree

{

private:

ListNode<DataType> \*root;

public:

BinaryTree();

~BinaryTree();

void createByPreInOrder(vector<DataType> &, vector<DataType> &);

void preOrderTraversal()const;

void inOrderTraversal()const;

void postOrderTraversal()const;

void levelTraversal()const;

bool isEmpty()const;

int countNodes()const;

int countLeaves()const;

void delTree();

int countHeight();

int countWidth();

void insertNode(DataType &);

bool isMirrorTree(BinaryTree &);

void copyBinaryTree(BinaryTree<DataType> &);

BinaryTree& operator=(BinaryTree<DataType> &);

};

#endif

（3）//BinaryTree.cpp

#include "BinaryTree.h"

#include "ListNode.h"

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

#include <queue>

using namespace std;

template <typename DataType>

BinaryTree<DataType>::BinaryTree() : root(nullptr) {}

template <typename DataType>

BinaryTree<DataType>::~BinaryTree()

{

delTree();

}

template <typename DataType>

ListNode<DataType>\* createByPreInOrderHelper(

vector<DataType> &preOrder, int preStart, int preEnd,

vector<DataType> &inOrder, int inStart, int inEnd,

map<DataType, int> &mymap

)

{

if(preStart > preEnd || inStart > inEnd)

return nullptr;

DataType rootValue = preOrder[preStart];

int rootIndex = mymap[rootValue];

int leftSize = rootIndex - inStart;

ListNode<DataType>\* root = new ListNode<DataType>(rootValue);

root->left = createByPreInOrderHelper(

preOrder, preStart + 1, preStart + leftSize,

inOrder, inStart, rootIndex - 1,

mymap

);

root->right = createByPreInOrderHelper(

preOrder, preStart + 1 + leftSize, preEnd,

inOrder, rootIndex + 1, inEnd,

mymap

);

return root;

}

template <typename DataType>

void BinaryTree<DataType>::createByPreInOrder(vector<DataType> &preOrder, vector<DataType> &inOrder)

{

if(preOrder.size() == 0)

{

root = nullptr;

return;

}

map<DataType, int> mymap;

for(int i = 0; i < inOrder.size(); i ++)

{

mymap[inOrder[i]] = i;

}

root = createByPreInOrderHelper(

preOrder, 0, preOrder.size() - 1,

inOrder, 0, inOrder.size() - 1,

mymap

);

}

template <typename DataType>

void preOrderTraversalHelper(ListNode<DataType>\* root)

{

if(root == nullptr)

return;

cout << root->val << " ";

preOrderTraversalHelper(root->left);

preOrderTraversalHelper(root->right);

}

template <typename DataType>

void BinaryTree<DataType>::preOrderTraversal()const

{

preOrderTraversalHelper(root);

}

template <typename DataType>

void inOrderTraversalHelper(ListNode<DataType>\* root)

{

if(root == nullptr)

return;

inOrderTraversalHelper(root->left);

cout << root->val << " ";

inOrderTraversalHelper(root->right);

}

template <typename DataType>

void BinaryTree<DataType>::inOrderTraversal()const

{

inOrderTraversalHelper(root);

}

template <typename DataType>

void postOrderTraversalHelper(ListNode<DataType>\* root)

{

if(root == nullptr)

return;

postOrderTraversalHelper(root->left);

postOrderTraversalHelper(root->right);

cout << root->val << " ";

}

template <typename DataType>

void BinaryTree<DataType>::postOrderTraversal()const

{

postOrderTraversalHelper(root);

}

template <typename DataType>

void BinaryTree<DataType>::levelTraversal()const

{

queue<ListNode<DataType>\*> myque;

if(root != nullptr)

{

myque.push(root);

cout << root->val << " ";

}

ListNode<DataType>\* p;

while(!myque.empty())

{

p = myque.front();

myque.pop();

if(p->left != nullptr)

{

cout << p->left->val << " ";

myque.push(p->left);

}

if(p->right != nullptr)

{

cout << p->right->val << " ";

myque.push(p->right);

}

}

}

template <typename DataType>

bool BinaryTree<DataType>::isEmpty()const

{

return (root == nullptr);

}

template <typename DataType>

int BinaryTree<DataType>::countNodes()const

{

queue<ListNode<DataType>\*> myque;

int ans = 0;

if(root != nullptr)

{

myque.push(root);

ans ++;

}

ListNode<DataType>\* p;

while(!myque.empty())

{

p = myque.front();

myque.pop();

if(p->left != nullptr)

{

ans ++;

myque.push(p->left);

}

if(p->right != nullptr)

{

ans ++;

myque.push(p->right);

}

}

return ans;

}

template <typename DataType>

void countLeavesHelper(ListNode<DataType>\* root, int &ans)

{

if(root != nullptr)

{

if(root->right == nullptr && root->left == nullptr)

ans ++;

countLeavesHelper(root->left, ans);

countLeavesHelper(root->right, ans);

}

}

template <typename DataType>

int BinaryTree<DataType>::countLeaves()const

{

int ans = 0;

countLeavesHelper(root, ans);

return ans;

}

template <typename DataType>

void delTreeHelper(ListNode<DataType>\* root)

{

if(root == nullptr)

return;

delTreeHelper(root->left);

delTreeHelper(root->right);

delete root;

}

template <typename DataType>

void BinaryTree<DataType>::delTree()

{

delTreeHelper(root);

root = nullptr;

}

template <typename DataType>

int countHeightHelper(ListNode<DataType>\* root)

{

if (!root)

return 0;

return 1 + max(countHeightHelper(root->left), countHeightHelper(root->right));

}

template <typename DataType>

int BinaryTree<DataType>::countHeight()

{

return countHeightHelper(root);

}

template <typename DataType>

int BinaryTree<DataType>::countWidth()

{

if(root == nullptr)

return 0;

queue<ListNode<DataType>\*> myque;

myque.push(root);

int maxWidth = 0;

while(!myque.empty())

{

int levelWidth = myque.size();

maxWidth = max(maxWidth, levelWidth);

for(int i = 0; i < levelWidth; i ++)

{

ListNode<DataType>\* temp = myque.front();

myque.pop();

if (temp->left != nullptr)

myque.push(temp->left);

if (temp->right != nullptr)

myque.push(temp->right);

}

}

return maxWidth;

}

template <typename DataType>

void insertNodeHelper(ListNode<DataType>\* &root, DataType &node)

{

if(root == nullptr)

{

root = new ListNode<DataType>(node);

return;

}

int leftHeight = countHeightHelper(root->left);

int rightHeight = countHeightHelper(root->right);

if(leftHeight <= rightHeight)

insertNodeHelper(root->left, node);

else

insertNodeHelper(root->right, node);

}

template <typename DataType>

void BinaryTree<DataType>::insertNode(DataType &node)

{

insertNodeHelper(root, node);

}

template <typename DataType>

bool isMirrorTreeHelper(ListNode<DataType>\* root1, ListNode<DataType>\* root2)

{

if(!root1 && !root2)

return true;

if(!root1 || !root2)

return false;

return (root1->val == root2->val) &&

isMirrorTreeHelper(root1->left, root2->right) &&

isMirrorTreeHelper(root1->right, root2->left);

}

template <typename DataType>

bool BinaryTree<DataType>::isMirrorTree(BinaryTree &other)

{

return isMirrorTreeHelper(root, other.root);

}

template <typename DataType>

ListNode<DataType>\* copyTree(ListNode<DataType>\* node)

{

if(node == nullptr)

return nullptr;

ListNode<DataType>\* newNode = new ListNode<DataType>(node->val);

newNode->left = copyTree(node->left);

newNode->right = copyTree(node->right);

return newNode;

}

template <typename DataType>

void BinaryTree<DataType>::copyBinaryTree(BinaryTree<DataType> &other)

{

root = copyTree(other.root);

}

template <typename DataType>

BinaryTree<DataType>& BinaryTree<DataType>::operator=(BinaryTree<DataType> &other)

{

if (this == &other)

return \*this;

root = other.root;

return \*this;

}

template class BinaryTree<char>;

（4）//main.cpp

#include "BinaryTree.h"

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

#include <queue>

using namespace std;

void printMenu()

{

cout << "==== Binary Tree Operations ====" << endl;

cout << "1. Build tree using preorder and inorder sequences" << endl;

cout << "2. Preorder traversal (recursive)" << endl;

cout << "3. Inorder traversal (recursive)" << endl;

cout << "4. Postorder traversal (recursive)" << endl;

cout << "5. Level order traversal" << endl;

cout << "6. Count nodes" << endl;

cout << "7. Count leaves" << endl;

cout << "8. Count height" << endl;

cout << "9. Count width" << endl;

cout << "10. Insert a new node in the tree" << endl;

cout << "11. Check if two trees are mirror images" << endl;

cout << "12. Copy binary tree" << endl;

cout << "13. Assign operator (=) for binary tree" << endl;

cout << "14. Clear the tree" << endl;

cout << "0. Exit" << endl;

cout << "=================================" << endl;

}

int main()

{

BinaryTree<char> tree1, tree2; // Create two trees

vector<char> preorder, inorder; // To hold the sequences

int choice;

char nodeValue;

while (true)

{

printMenu();

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

{

cout << "Enter the preorder sequence (no spaces): ";

string preOrderStr;

cin >> preOrderStr;

preorder.clear();

for (char ch : preOrderStr)

{

preorder.push\_back(ch);

}

cout << "Enter the inorder sequence (no spaces): ";

string inOrderStr;

cin >> inOrderStr;

inorder.clear();

for (char ch : inOrderStr)

{

inorder.push\_back(ch);

}

tree1.createByPreInOrder(preorder, inorder);

cout << "Tree1 built successfully!" << endl;

}

break;

case 2:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Preorder traversal: ";

tree1.preOrderTraversal();

cout << endl;

}

}

break;

case 3:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Inorder traversal: ";

tree1.inOrderTraversal();

cout << endl;

}

}

break;

case 4:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Postorder traversal: ";

tree1.postOrderTraversal();

cout << endl;

}

}

break;

case 5:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Level order traversal: ";

tree1.levelTraversal();

cout << endl;

}

}

break;

case 6:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Node count: " << tree1.countNodes() << endl;

}

}

break;

case 7:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Leaf count: " << tree1.countLeaves() << endl;

}

}

break;

case 8:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Tree1 height: " << tree1.countHeight() << endl;

}

}

break;

case 9:

{

if (tree1.isEmpty())

{

cout << "Tree1 is empty. Please build a tree first!" << endl;

}

else

{

cout << "Tree1 width: " << tree1.countWidth() << endl;

}

}

break;

case 10:

{

cout << "Enter the value of the node to insert: ";

cin >> nodeValue;

tree1.insertNode(nodeValue);

cout << "Node inserted successfully!" << endl;

}

break;

case 11:

{

cout << "Enter the preorder sequence (no spaces): ";

string preOrderStr1;

cin >> preOrderStr1;

preorder.clear();

for (char ch : preOrderStr1)

{

preorder.push\_back(ch);

}

cout << "Enter the inorder sequence (no spaces): ";

string inOrderStr1;

cin >> inOrderStr1;

inorder.clear();

for (char ch : inOrderStr1)

{

inorder.push\_back(ch);

}

tree2.createByPreInOrder(preorder, inorder);

cout << "Tree2 built successfully!" << endl;

if (tree1.isMirrorTree(tree2))

{

cout << "The trees are mirror images!" << endl;

}

else

{

cout << "The trees are not mirror images!" << endl;

}

}

break;

case 12:

{

tree2.copyBinaryTree(tree1); // Using overloaded assignment operator

cout << "Tree2 copied from tree1!" << endl;

}

break;

case 13:

{

cout << "Assigning tree1 to tree2..." << endl;

tree2 = tree1;

cout << "Tree assigned successfully!" << endl;

}

break;

case 14:

{

tree1.delTree();

cout << "Tree cleared!" << endl;

}

break;

case 0:

{

cout << "Exiting the program." << endl;

return 0;

}

break;

default:

{

cout << "Invalid choice. Please try again." << endl;

}

break;

}

cout << endl;

}

return 0;

}