**《数据结构》课程实践报告**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软工 | 姓名 | 王天予 | 学号 | 2362401031 |
| 实验布置日期 | | 2024.11.19 | | 提交  日期 |  | | 成绩 |  |

课程实践实验5：二叉树操作

## 一、问题描述及要求

### 1. 问题描述和要求

本实验的目标是设计并实现一个支持多种操作的二叉树程序。二叉树是树形结构中最基础的数据结构之一，广泛应用于搜索、排序以及表达式解析等领域。实验内容要求实现二叉树的基本操作，包括构造、遍历、节点插入、树的特性计算等。此外，还需扩展实现两个二叉树的镜像判断，以及拷贝构造与赋值重载的功能。

本实验需要实现以下功能：

- 构造空二叉树。

- 根据前序和中序序列构建二叉树。

- 判断二叉树是否为空。

- 支持递归实现的先序、中序、后序遍历。

- 实现层次遍历。

- 清空二叉树。

- 求二叉树的高度、宽度，以及节点总数和叶子节点数。

- 在二叉树上较矮的子树上插入一个节点（若两侧高度相等，则插入左子树）。

- 通过前序和中序序列创建另一棵二叉树，并判断两棵二叉树是否互为镜像。

- 添加拷贝构造函数，实现另一棵二叉树的深拷贝。

- 实现赋值运算符 = 的重载。

### 2. 示例输入与输出

示例：

- 输入前序和中序序列：ABDECFG 和 DBEAFCG，构建二叉树后支持遍历输出。

- 输入插入的节点：H，程序会根据规则插入到较矮的子树。

- 输入另一棵树的前序和中序序列：GFCABED 和 CFGAEBD，判断两棵树是否互为镜像。

## 二、概要设计

### 1. 系统功能列表

二叉树类：

- createEmptyTree(): 构造空二叉树。

- createFromPreIn(string preorder, string inorder): 根据前序和中序序列构建二叉树。

- isEmpty(): 判断二叉树是否为空。

- traversePreOrder(): 递归实现先序遍历。

- traverseInOrder(): 递归实现中序遍历。

- traversePostOrder(): 递归实现后序遍历。

- levelOrderTraversal(): 层次遍历。

- getHeight(): 求二叉树的高度。

- getWidth(): 求二叉树的宽度。

- getNodeCount(): 求二叉树节点总数。

- getLeafCount(): 求叶子节点数。

- insert(char key): 在较矮子树上插入节点。

- isMirrorOf(const BinaryTree& other): 判断两棵树是否互为镜像。

- 深拷贝与赋值：

- BinaryTree(const BinaryTree& other): 拷贝构造函数。

- operator=(const BinaryTree& other): 赋值运算符重载。

### 2. 二叉树的数据对象与数据关系

#### 数据对象

一个二叉树的数据对象可以表示为：

}

其中， 是节点值的集合，每个节点都有唯一的键值 ，且可以为空。  
节点的键值是二叉树结构的基础，它用于唯一标识节点并支持操作。二叉树的键值可扩展为复杂类型，以适应不同应用场景，例如排序和表达式求值。通过这种定义，我们可以精确描述二叉树中每个节点的数据内容。

#### 数据关系

二叉树的数据关系表示为：

}

即，父节点与子节点之间的关系由有向边定义。  
二叉树中的关系是有序的，每个父节点最多只能有两个直接子节点。通过这种有向关系，可以确定每个节点在树中的唯一位置，并对树进行操作。数据关系也决定了遍历方法的多样性，例如先序遍历需要优先访问根节点，而中序遍历则优先访问左子树。

#### 二叉树的约束条件

二叉树满足以下约束条件：

1. 每个节点最多有两个子节点：左子节点和右子节点。
2. 设 为二叉树的高度，树中最多有 个节点。

3. 根节点无父节点，所有其他节点有且仅有一个父节点。

这些约束确保了二叉树的结构稳定性和层次性。通过限制节点的子节点数量，可以高效地执行插入、删除和遍历操作。高度和节点数量的关系使得二叉树适用于需要快速访问和动态调整的场景，例如搜索树和堆。

## 三、详细设计

### 二叉树类 ADT

#### ADT BinaryTree

**数据对象**：以链式存储实现的二叉树。

**基本操作**： 1. **构造和销毁** - createEmptyTree(BinaryTree &T): 构造空树。 - clear(BinaryTree &T): 清空树，释放所有节点内存。

1. **遍历与操作**
   * traversePreOrder(BinaryTree T): 先序遍历。
   * traverseInOrder(BinaryTree T): 中序遍历。
   * traversePostOrder(BinaryTree T): 后序遍历。
   * levelOrderTraversal(BinaryTree T): 层次遍历。
2. **查询与计算**
   * getHeight(BinaryTree T): 获取树高。
   * getWidth(BinaryTree T): 获取树宽。
   * getNodeCount(BinaryTree T): 获取节点总数。
   * getLeafCount(BinaryTree T): 获取叶子节点数。
   * isEmpty(BinaryTree T): 判断树是否为空。
3. **扩展功能**
   * insert(BinaryTree &T, char key): 插入节点。
   * isMirrorOf(BinaryTree T1, BinaryTree T2): 判断两棵树是否互为镜像。
4. **深拷贝与赋值**
   * BinaryTree(const BinaryTree& other): 拷贝构造。
   * operator=(const BinaryTree& other): 赋值重载。

## 四、实验结果

### 测试数据及结果

1. **测试用例 1**：构建二叉树
   * 输入：前序 ABDECFG，中序 DBEAFCG
   * 预期输出：通过遍历验证结构，所有遍历结果正确。
2. **测试用例 2**：插入节点
   * 输入：H
   * 预期输出：节点 H 插入到较矮子树，树结构更新正确。
3. **测试用例 3**：判断两棵树是否互为镜像
   * 树 1 前序、中序：ABDECFG、DBEAFCG
   * 树 2 前序、中序：GCFEDBA、FCGBEDA
   * 预期输出：两棵树互为镜像。
4. **测试用例 4**：清空树并插入新节点
   * 输入：clearTree() 和 insert('Z')
   * 预期输出：树清空后插入节点成功，Z 为根节点。
5. **测试用例 5**：求高度、宽度、节点数和叶子数
   * 树：ABDECFG
   * 高度：3
   * 宽度：4
   * 节点数：7
   * 叶子数：3
   * 预期输出：所有属性值正确。

## 五、实验分析与探讨

### 1. 功能验证

程序成功实现了基本和扩展功能，所有测试用例均通过。通过深拷贝和赋值重载的实现，程序对多树操作的支持更加完善。

### 2. 设计思路和问题解决

1. **深拷贝和赋值重载**  
   深拷贝通过递归复制节点，确保多棵树操作时数据独立。赋值重载避免内存泄漏，设计时需注意避免自赋值。
2. **插入节点规则**  
   使用递归计算左右子树高度，确保节点插入在较矮子树。
3. **镜像判断**  
   通过递归对比左子树与右子树，判断两棵树的镜像关系。

## 六、总结与反思

1. **二叉树的链式存储**  
   链式存储实现操作灵活，但需注意内存释放。通过递归清空树的设计，确保程序运行后无内存泄漏。
2. **递归算法的优化**  
   在实现过程中，递归函数较多，需谨慎处理终止条件，防止栈溢出。
3. **扩展功能的挑战**  
   插入节点的逻辑和镜像判断的递归实现需确保效率和正确性，设计过程中考虑了多种边界情况。

**附录：源代码**

1. **实验环境**：

编译环境：TDM-GCC 4.9.2 64bit release

命令行参数：-static-libgcc -std=c++11

2、**目录结构**

（1）Solution.cpp

#include <iostream>

#include <queue>

#include <string>

using namespace std;

struct TreeNode {

char key;

TreeNode\* left;

TreeNode\* right;

TreeNode(char k) : key(k), left(nullptr), right(nullptr) {}

};

class BinaryTree {

private:

TreeNode\* root;

void clear(TreeNode\*& node) {

if (node) {

clear(node->left);

clear(node->right);

delete node;

node = nullptr;

}

}

int getHeight(TreeNode\* node) const {

if (!node) return 0;

return 1 + max(getHeight(node->left), getHeight(node->right));

}

int getWidth(TreeNode\* node) const {

if (!node) return 0;

queue<TreeNode\*> q;

q.push(node);

int maxWidth = 0;

while (!q.empty()) {

int count = q.size();

maxWidth = max(maxWidth, count);

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

TreeNode\* temp = q.front();

q.pop();

if (temp->left) q.push(temp->left);

if (temp->right) q.push(temp->right);

}

}

return maxWidth;

}

int getNodeCount(TreeNode\* node) const {

if (!node) return 0;

return 1 + getNodeCount(node->left) + getNodeCount(node->right);

}

int getLeafCount(TreeNode\* node) const {

if (!node) return 0;

if (!node->left && !node->right) return 1;

return getLeafCount(node->left) + getLeafCount(node->right);

}

void preorderTraversal(TreeNode\* node) const {

if (node) {

cout << node->key << " ";

preorderTraversal(node->left);

preorderTraversal(node->right);

}

}

void inorderTraversal(TreeNode\* node) const {

if (node) {

inorderTraversal(node->left);

cout << node->key << " ";

inorderTraversal(node->right);

}

}

void postorderTraversal(TreeNode\* node) const {

if (node) {

postorderTraversal(node->left);

postorderTraversal(node->right);

cout << node->key << " ";

}

}

TreeNode\* buildFromPreIn(string& preorder, string& inorder) {

if (preorder.empty() || inorder.empty()) return nullptr;

char rootKey = preorder[0];

auto rootIndex = inorder.find(rootKey);

if (rootIndex == string::npos) return nullptr;

TreeNode\* newNode = new TreeNode(rootKey);

string leftIn = inorder.substr(0, rootIndex);

string rightIn = inorder.substr(rootIndex + 1);

preorder = preorder.substr(1);

newNode->left = buildFromPreIn(preorder, leftIn);

newNode->right = buildFromPreIn(preorder, rightIn);

return newNode;

}

void insertNode(TreeNode\*& node, char key) {

if (!node) {

node = new TreeNode(key);

} else {

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

if (leftHeight <= rightHeight) {

insertNode(node->left, key);

} else {

insertNode(node->right, key);

}

}

}

bool areMirrors(TreeNode\* a, TreeNode\* b) {

if (!a && !b) return true;

if (!a || !b || a->key != b->key) return false;

return areMirrors(a->left, b->right) && areMirrors(a->right, b->left);

}

TreeNode\* copyTree(TreeNode\* node) {

if (!node) return nullptr;

TreeNode\* newNode = new TreeNode(node->key);

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

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

return newNode;

}

public:

BinaryTree() : root(nullptr) {}

BinaryTree(const BinaryTree& other) {

root = copyTree(other.root);

}

~BinaryTree() {

clear(root);

}

BinaryTree& operator=(const BinaryTree& other) {

if (this != &other) {

clear(root);

root = copyTree(other.root);

}

return \*this;

}

void createEmptyTree() {

clear(root);

}

void createFromPreIn(string preorder, string inorder) {

clear(root);

root = buildFromPreIn(preorder, inorder);

}

void insert(char key) {

insertNode(root, key);

}

bool isEmpty() const {

return root == nullptr;

}

void traversePreOrder() const {

preorderTraversal(root);

cout << endl;

}

void traverseInOrder() const {

inorderTraversal(root);

cout << endl;

}

void traversePostOrder() const {

postorderTraversal(root);

cout << endl;

}

void levelOrderTraversal() const {

if (!root) {

cout << "二叉树为空。" << endl;

return;

}

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode\* node = q.front();

q.pop();

cout << node->key << " ";

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

cout << endl;

}

void clearTree() {

clear(root);

}

int getHeight() const {

return getHeight(root);

}

int getWidth() const {

return getWidth(root);

}

int getNodeCount() const {

return getNodeCount(root);

}

int getLeafCount() const {

return getLeafCount(root);

}

bool isMirrorOf(const BinaryTree& other) {

return areMirrors(root, other.root);

}

};

void displayMenu() {

cout << "二叉树操作菜单：" << endl;

cout << "1. 构造空二叉树" << endl;

cout << "2. 根据前序和中序序列构建二叉树" << endl;

cout << "3. 判别二叉树是否为空" << endl;

cout << "4. 递归先序遍历" << endl;

cout << "5. 递归中序遍历" << endl;

cout << "6. 递归后序遍历" << endl;

cout << "7. 层次遍历" << endl;

cout << "8. 清空二叉树" << endl;

cout << "9. 求二叉树高度" << endl;

cout << "10. 求二叉树结点数和叶子结点数" << endl;

cout << "11. 求二叉树宽度" << endl;

cout << "12. 插入节点" << endl;

cout << "13. 判断两棵树是否互为镜像" << endl;

cout << "0. 退出" << endl;

}

int main() {

BinaryTree tree, otherTree;

int choice;

while (true) {

displayMenu();

cin >> choice;

if (choice == 1) {

tree.createEmptyTree();

} else if (choice == 2) {

string preorder, inorder;

cin >> preorder >> inorder;

tree.createFromPreIn(preorder, inorder);

} else if (choice == 3) {

cout << (tree.isEmpty() ? "二叉树为空。" : "二叉树不为空。") << endl;

} else if (choice == 4) {

tree.traversePreOrder();

} else if (choice == 5) {

tree.traverseInOrder();

} else if (choice == 6) {

tree.traversePostOrder();

} else if (choice == 7) {

tree.levelOrderTraversal();

} else if (choice == 8) {

tree.clearTree();

} else if (choice == 9) {

cout << "高度: " << tree.getHeight() << endl;

} else if (choice == 10) {

cout << "节点数: " << tree.getNodeCount() << ", 叶子结点数: " << tree.getLeafCount() << endl;

} else if (choice == 11) {

cout << "宽度: " << tree.getWidth() << endl;

} else if (choice == 12) {

char key;

cin >> key;

tree.insert(key);

} else if (choice == 13) {

string preorder, inorder;

cin >> preorder >> inorder;

otherTree.createFromPreIn(preorder, inorder);

cout << (tree.isMirrorOf(otherTree) ? "互为镜像" : "不互为镜像") << endl;

} else if (choice == 0) {

break;

}

}

return 0;

}