Exercise 6: Simple Binary Tree

Related Topics

Dynamic Memory Allocation, Deep Copy, Operator Overloading

Background

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. The topmost node in the tree is called the root. The following is an example of a binary tree:

Recall the `tree_t` type from Project 2, it is actually an abstracted binary tree with basic functionalities. In this exercise, we will reimplement binary trees through a `BinaryTree` class with the basic functionalities like tree construction, copy, equality check, and basic tree operations.

Task Overview

In this exercise, each tree node will store an integer value and is already defined as follows:

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
};
```

You need to implement the `BinaryTree` class with the following member functions or operators:

- **Constructors and Destructor**: Default constructor, parameterized constructors, copy constructor and destructor.
- **Overloaded Operators**: Assignment operator, equality operator, inequality operator, and output operator.

- **Tree Operations**: Check if the tree is empty, calculate the sum of all values, calculate the depth of the tree, and perform an in-order traversal (overloaded output operator).
- **Other Private Helper Functions (If Needed)**: Copy nodes from another tree, remove all nodes from the tree, and check equality between two trees, etc.

Header File

You may refer to the comments in `ex6.h` for the detailed requirements.

```
#ifndef EX6_H
#define EX6_H

#include <ostream>
/**
* @struct TreeNode
* @brief Represents a node in a binary tree.
*/
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
};
    /**
    * @class BinaryTree
    * @brief Implements a simple binary tree data structure.
    */
class BinaryTree {
    public:
    /**
    * @brief Default constructor that initializes an empty binary tree.
    * The root node is set to nullptr.
    */
    BinaryTree();
    /**
    * @brief Parameterized constructor to create a binary tree with a given value.
    * The left and right subtrees are empty.
    * @param val The value for the root node of the binary tree.
    */
    BinaryTree(int val);
    /**
    * @brief Parameterized constructor to create a binary tree with given value and sub
    * @param val The value for the root node of the binary tree.
    * @param left The left subtree.
    * @param right The right subtree.
    */
    BinaryTree(int val, const BinaryTree &left, const BinaryTree &right);
    /**
    * @brief Copy constructor that creates a deep copy of another binary tree.
    * @param other The binary tree to copy from.
    */
    BinaryTree(const BinaryTree &other);
    /**
```

```
* @brief Assignment operator that copies the content of another binary tree.
* @param other The binary tree to assign from.
* @return A reference to the modified binary tree.
*/
BinaryTree &operator=(const BinaryTree &other);
/**
* @brief Destructor that cleans up the allocated resources of the binary tree.
*/
~BinaryTree();

/**
* @brief Checks equality between this binary tree and another binary tree.
* @param other The binary tree to compare with.
* @return True if the trees are equal, false otherwise.
*/
bool operator==(const BinaryTree &other) const;
/**
* @brief Checks inequality between this binary tree and another binary tree.
* @param other The binary tree to compare with.
* @return True if the trees are not equal, false otherwise.
*/
bool operator!=(const BinaryTree &other) const;
/**
* @brief Checks if the binary tree is empty.
* @return True if the tree is empty, false otherwise.
*/
bool isEmpty() const;
/**
* @brief Calculates the sum of all values in the binary tree.
* @return The sum of all values. If the tree is empty, return 0.
*/
int sum() const;

/**
* @brief Calculates the depth of the binary tree.
* @return The depth of the tree. If the tree is empty, return 0.
*/
int depth() const;
/**
* @brief Overloaded output operator to print the binary tree.
* Performs an in-order traversal of the binary tree and prints each value.
* In-order traversal visits the left subtree, then the root, and finally the right
*
```

```
    * For example, consider the following binary tree:
    *
    *     4
    *    / \
    *   2  5
    *  / \
    * 1  3
    *
    * The expected output of the overloaded operator<< for this tree would be:
    * "1 2 3 4 5"
    *
    * @param os The output stream to print the tree.
    * @param tree The binary tree to print.
    * @return The output stream.
    */
    friend std::ostream &operator<<(std::ostream &os, const BinaryTree &tree);
private:
    TreeNode *root;
    /**
    * @brief Recursively copies all nodes from another tree.
    * @param root The root node of the tree to copy.
    * @return The root node of the copied tree.
    */
    TreeNode *copy(TreeNode *root);
    /**

    * @brief Recursively removes all nodes from the tree, deallocating memory.
    * @param root The root node of the tree to remove.

    */
    void removeAll(TreeNode *root);
    /**
    * @brief Recursively checks equality between two trees.
    * @param root The root node of the first tree.
    * @param other The root node of the second tree.
    * @return True if the trees are equal, false otherwise.
    */
    bool equal(TreeNode *root, TreeNode *other) const;
    // You can add additional private helper methods here.
};
#endif // EX6_H
```

Testing

In this exercise, we won't turn on the memory leak detection for the pretest cases. This means that you need to make sure there are no memory leaks in your implementation and check it by yourself. You can use tools like `valgrind` to check for memory leaks:

```
valgrind --leak-check=full ./your_program
```

Implementation Details

- You should implement the `BinaryTree` class in `ex6.h` and `ex6.cpp` . You're welcome to write your own main function to test your implementation, but only these two files should be submitted.
- You may add additional private helper functions if needed, but **the public interface should remain the same.**
- Make sure your objects are **deep copied** in relevant functions.
- You may consider **utilizing private helper functions** when implementing the public member functions.
- In the overloaded `operator<<` , you shouldn't add a new line character at the end of the output.
- Make sure you **don't have any memory leaks** in your implementation. Since this exercise is designed to test your understanding of dynamic memory allocation, any memory leak detected in correct cases will lose **all of the score** for that case.
- **Any typos or format errors will be treated as failed test cases.**

Submission

Submit your `ex6.h` and `ex6.cpp` files to JOJ. **The due date is 12.22 23:59.**