Binary Tree Assignment (Day 1)

Binary Tree Assignment (Day 1)

Introduction

In this assignment, you will explore the Binary Tree data structure. A binary tree is a hierarchical data structure where each node has at most two children, referred to as the left child and the right child. Binary trees are fundamental in computer science and are used in various applications including searching, sorting, and representing hierarchical data.

This assignment focuses on understanding a basic implementation of a Binary Search Tree (BST), which is a special type of binary tree where for each node: - All elements in the left subtree are less than the node's value - All elements in the right subtree are greater than or equal to the node's value

The key learning objective is to understand how recursion is used to solve tree-related problems.

Binary Tree Class Overview

You are designing h a `BinaryTree` class that contains the following components:

1. An inner `Node` class that represents each node in the tree
2. Methods for creating and manipulating the tree
3. Methods for traversing and analyzing the tree structure

Class Structure

```
public class BinaryTree {
    protected class Node {
        int data;
        Node left;
        Node right;

        public Node(int data) {
            this.data = data;
        }
    }

    protected Node root;

    // Constructors and methods...
}
```

Method Descriptions

Constructors

`public BinaryTree()`

- Creates an empty binary tree with a null root.

`public BinaryTree(int data)`

- Creates a binary tree with a root node containing the specified data.

Insertion

`public void insert(int data)`

- Public API
- calls `insert_recursive(Node root, int data)`

`private Node insert_recursive(Node node, int data)`

- Recursively insert `data` into (sub)tree rooted at `node`.
- Maintains BST order (left tree is all $<$ root, right tree is $>=$)
- Modifies the tree rooted at `node`
- private, called only by `insert`

Traversal

`public String in_order_traversal()`

- Public API
- Calls `in_order_traversal(Node root, String s)`

`private String in_order_traversal(Node node, String s)`

- Performs an in-order traversal of the (sub)tree rooted at `node`.
- Returns a string representation of the traversal.
- private, called only by `in_order_traversal`
- note this method uses overloading instead of using a new name!

Analysis

`public int count_nodes()`

- public API
- calls `count_nodes_recursive(Node root)`

`private int count_nodes_recursive(Node node)`

- Counts the total number of nodes in the (sub)tree rooted at `node`.
- Uses recursion to traverse all nodes.

`public int depth()`

- public API
- calls `depth_recursive(Node node)`

`private int depth_recursive(Node node)`

- Calculates the depth (or height) of the (sub)tree rooted at `node`.
- The depth is the length of the longest path from the root to any leaf.
- We will say a one node tree has depth 1, an empty tree has depth 0.

Assignment Tasks

1. Implement the `BinaryTree` class.
2. Test its execution using your own driver class (see example below)
3. Submit to javadrop.io by the end of class.

Sample Output

Here's an example of creating a binary tree and using its methods:

```java
public class BinaryTreeDemo {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Insert values
        tree.insert(50);
        tree.insert(30);
        tree.insert(70);
        tree.insert(20);
        tree.insert(40);
        tree.insert(60);
        tree.insert(80);

        // Display in-order traversal
        System.out.println("In-order traversal:");
        tree.in_order_traversal();

        // Count nodes
        System.out.println("\nNumber of nodes: " + tree.count_nodes());

        // Calculate depth
        System.out.println("Tree depth: " + tree.depth());
    }
}
```

Expected output:

```
In-order traversal:
20 30 40 50 60 70 80
Number of nodes: 7
Tree depth: 3
```

Thinking About Recursion

For each recursive method in the `BinaryTree` class, consider: 1. What is the base case? (When does the recursion stop?) 2. How does the method break down the problem into smaller subproblems? 3. How does the method combine the results of the subproblems to solve the original problem?

Understanding these aspects of recursion is crucial for working with tree structures.