Assignment 7: Binary Trees

CS3305/W01 Data Structures

Casey Hampson

October 14, 2024

Program Output

```
Tree 1:
Binary Search Tree (BST)? Yes!
Depth/Height of Root: 1
Max: 3.0000
Sum: 6.0000
Average: 2.0000
Balanced? Yes!

Tree 2:
Binary Search Tree (BST)? No.
Depth/Height of Root: 3
Max: 12.0000
Sum: 78.0000
Average: 6.5000
Balanced? Yes!

Tree 3:
Binary Search Tree (BST)? No.
Depth/Height of Root: 10
Max: 0.4625
Sum: 12.9191
Average: 0.2267
Balanced? Yes!
```

```
Tree 4:
Binary Search Tree (BST)? No.
Depth/Height of Root: 11
Max: 0.4875
Sum: 16.6020
Average: 0.2478
Balanced? No.

Tree 5:
Binary Search Tree (BST)? No.
Depth/Height of Root: 10
Max: 99.6742
Sum: 11018.3681
Average: 47.4930
Balanced? Yes!

Tree 6:
Binary Search Tree (BST)? No.
Depth/Height of Root: 12
Max: 99.9515
Sum: 29775.0429
Average: 48.8917
Balanced? Yes!
```

Source Code

```java
// Name: Casey Hampson
// Class: CS 3305/W01
// Term: Fall 2024
// Instructor: Sharon Perry
// Assignment: 7-Binary Trees

import java.util.Arrays;




class Node<E extends Number> {
    public E elem;
    public Node<E> left;
    public Node<E> right;
    public int height;

    Node(E elem) {
        this.elem = elem;
        this.left = null;
        this.right = null;
        this.height = 0;
    }
}

class BinaryTree<E extends Number> {
    public Node<E> root;
    public int size;
    public int depth;

    public BinaryTree() { this.root = null; }



    // determines if the tree is a BST
    // we just check each node for the condition for a BST
    // NOTE: there is no way to do mathematical operations on any generic
    // even if it extends `Number`; i convert everything to doubles to circumvent this
    private boolean __isbinarytree(Node<E> root) {
        if (root == null) return true;
        boolean check_current =
                (root.left == null ? true : root.left.elem.doubleValue() < root.elem.doubleValue())
            && (root.right == null ? true : root.right.elem.doubleValue() > root.elem.doubleValue());
        return check_current && __isbinarytree(root.left) && __isbinarytree(root.right);
    }
    public boolean IsBinaryTree() {
        return __isbinarytree(this.root);
    }
```

```
// grabs the height of a node
private int __getheight(Node<E> node) {
    if (node == null) return 0;

    if (node.left == null && node.right == null) return 0;
    else if (node.left == null) return 1 + __getheight(node.right);
    else if (node.right == null) return 1 + __getheight(node.left);
    else return 1 + Math.max(__getheight(node.left), __getheight(node.right));
}
// goes through and updates the height values for each node
// needed for determining whether the tree is balanced
private void __updateheights(Node<E> node) {
    if (node == null) return;

    node.height = __getheight(node);
    node.left.height = __getheight(node.left);
    node.right.height = __getheight(node.right);
}

// updates all the heights and grabs the height of the root (i.e. the depth of the tree)
public int GetHeight() {
    __updateheights(this.root);
    return this.root.height;
}


// just as with before, generics cannot do math stuff,
// so i convert everything to doubles
private double __findmax(Node<E> node) {
    if (node.left == null && node.right == null) return node.elem.doubleValue();
    else if (node.left == null) return Math.max(__findmax(node.right), node.elem.doubleValue());
    else if (node.right == null) return Math.max(__findmax(node.left), node.elem.doubleValue());
    else return Math.max(__findmax(node.left), __findmax(node.right));
}
public double FindMax() {
    return __findmax(this.root);
}



// this one should be pretty straightforward
private double __findsum(Node<E> node) {
    if (node.left == null && node.right == null) return node.elem.doubleValue();
    else if (node.left == null) return node.elem.doubleValue() + __findsum(node.right);
    else if (node.right == null) return node.elem.doubleValue() + __findsum(node.left);
    else return node.elem.doubleValue() + __findsum(node.left) + __findsum(node.right);
}
public double FindSum() {
    return __findsum(this.root);
}


// as should this one
public double FindAverage() {
```

```
        return this.FindSum() / (double)this.size;
    }


    // if any node is not balanced, we return false
    // otherwise everything falls through and true is returned
    private int __balancefactor(Node<E> node) {
        if (node.right == null && node.left == null) return 0;
        else if (node.left == null) return +node.height;
        else if (node.right == null) return -node.height;
        else return node.right.height - node.left.height;
    }
    private boolean __isbalanced(Node<E> node) {
        if (Math.abs(this.__balancefactor(node)) > 2) return false;
        if (node.left != null) __isbalanced(node.left);
        if (node.right != null) __isbalanced(node.right);
        return true;
    }
    public boolean IsBalanced() {
        return this.__isbalanced(this.root);
    }




}
```

```
public class A7 {


    // helper function to find the index of an element in an array
    // need to do it this way due to doubles
    public static <E extends Number> int IndexOf(E[] arr, E elem) {
        for (int i=0; i< arr.length; i++) {
            if (Math.abs(arr[i].doubleValue() - elem.doubleValue()) < 0.0001) return i;
        }
        // in case it's not found!
        return -1;
    }

    public static <E extends Number> Node<E> __BSTCreate(E[] preorder, E[] inorder) {
        // the basic idea is that in preorder notation, the first element is the root
        // in inorder notation the root is right in the middle of the two subtrees
        // so with the root from the preorder tree, we split the inorder list
        // into the left and right subtrees
        // we also split the preorder array by doing the same splitting as
        // with the inorder array but incrementing the starting
        // index by 1
```

```
    // if then length of the preorder array is 1, then its the last one
    // and we can just return that node
    if (preorder.length == 1) return new Node<>(preorder[0]);
    // if it is null then there is nothing here so return null
    if (preorder.length == 0) return null;

    // grab the index of the root in the inorder array
    // and place it into a node
    E root_elem = preorder[0];
    Node<E> root = new Node<>(root_elem);
    int root_inorder_idx = IndexOf(inorder, root_elem);
    if (root_inorder_idx == -1) return null; // make sure element was found


    // split inorder array into the two arrays
    E[] left_arr_inorder =  Arrays.copyOfRange(inorder, 0, root_inorder_idx);
    E[] right_arr_inorder = Arrays.copyOfRange(inorder, root_inorder_idx+1, inorder.length);

    // then split the preorder array
    E[] left_arr_preorder =  Arrays.copyOfRange(preorder, 1, root_inorder_idx+1);
    E[] right_arr_preorder = Arrays.copyOfRange(preorder, root_inorder_idx+1, preorder.length);

    // recursive step
    root.left = __BSTCreate(left_arr_preorder, left_arr_inorder);
    root.right = __BSTCreate(right_arr_preorder, right_arr_inorder);

    // return the final root
    // the other method turns it into a binary tree
    return root;
}

// gets the root Node from the previous method
// and turns it into a binary tree
public static <E extends Number> BinaryTree<E> BSTCreate(E[] preorder, E[] inorder, int size) {
    Node<E> root = __BSTCreate(preorder, inorder);
    if (root == null) return null;

    BinaryTree<E> bt = new BinaryTree<>();
    bt.root = root;
    bt.size = size;

    return bt;
}


// prints out all of the required stuff for the assignment
public static <E extends Number> void GetBTDetails(BinaryTree<E> tree, String tree_name) {
    System.out.printf("%s:\n", tree_name);
    System.out.printf("Binary Search Tree (BST)? %s\n", tree.IsBinaryTree() ? "Yes!" : "No.");
    System.out.printf("Depth/Height of Root: %d\n", tree.GetHeight());
    System.out.printf("Max: %.4f\n", tree.FindMax());
    System.out.printf("Sum: %.4f\n", tree.FindSum());
    System.out.printf("Average: %.4f\n", tree.FindAverage());
    System.out.printf("Balanced? %s\n\n", tree.IsBalanced() ? "Yes!" : "No.");
```

```
    }


    public static void main(String[] args) {
        final int tree1_size = 3;
        Integer[] pre1 = {2, 1, 3};
        Integer[] in1  = {1, 2, 3};
        BinaryTree<Integer> bt1 = BSTCreate(pre1, in1, tree1_size);
        if (bt1 == null) { System.out.printf("Error creating tree 1!\n"); return; }
        GetBTDetails(bt1, "Tree 1");

        final int tree2_size = 12;
        Integer[] pre2 =  { 1, 2, 4, 8, 5, 9, 3, 6, 10, 11, 7, 12};
        Integer[] in2 = {8, 4, 2, 9, 5, 1, 10, 6, 11, 3, 7, 12};
        BinaryTree<Integer> bt2 = BSTCreate(pre2, in2, tree2_size);
        if (bt2 == null) { System.out.printf("Error creating tree 2!\n"); return; }
        GetBTDetails(bt2, "Tree 2");

        final int tree3_size = 57;
        Double[] pre3 = {0.185038, 0.394342, 0.291092, 0.289448, 0.101709, 0.0611125, 0.193208, 0.13517
        Double[] in3 = {0.193208, 0.13517, 0.0611125, 0.101709, 0.289448, 0.410769, 0.0198583, 0.151873
        BinaryTree<Double> bt3 = BSTCreate(pre3, in3, tree3_size);
        if (bt3 == null) { System.out.printf("Error creating tree 3!\n"); return; }
        GetBTDetails(bt3, "Tree 3");

        final int tree4_size = 67;
        Double[] pre4 = {0.0548774, 0.0219107, 0.399846, 0.11906, 0.276626, 0.0025399, 0.0871276, 0.414
        Double[] in4 = {0.276626, 0.0871276, 0.0025399, 0.414861, 0.245851, 0.11906, 0.326767, 0.399846
        BinaryTree<Double> bt4 = BSTCreate(pre4, in4, tree4_size);
        if (bt4 == null) { System.out.printf("Error creating tree 4!\n"); return; }
        GetBTDetails(bt4, "Tree 4");

        final int tree5_size = 232;
        Double[] pre5 = {73.4985, 46.8521, 14.3886, 12.9232, 85.4859, 46.2981, 80.6752, 88.638, 4.35911
        Double[] in5 = {80.6752, 88.638, 46.2981, 58.6892, 4.35911, 84.1117, 40.5937, 85.4859, 18.5848,
        BinaryTree<Double> bt5 = BSTCreate(pre5, in5, tree5_size);
        if (bt5 == null) { System.out.printf("Error creating tree 5!\n"); return; }
        GetBTDetails(bt5, "Tree 5");

        final int tree6_size = 609;
        Double[] pre6 = {4.3306, 3.11497, 52.1211, 19.2026, 9.15248, 72.7208, 63.34, 46.2895, 64.0222, (
        Double[] in6 = {64.0222, 46.2895, 63.34, 89.066, 63.4586, 6.59548, 77.8256, 72.7208, 56.1533, 58
        BinaryTree<Double> bt6 = BSTCreate(pre6, in6, tree6_size);
        if (bt6 == null) { System.out.printf("Error creating tree 6!\n"); return; }
        GetBTDetails(bt6, "Tree 6");
    }
}
```