Penn Engineering
ONLINE LEARNING

CIT 594 Module 6 Programming Assignment

In this module, we learned that a Binary Search Tree (BST) must remain balanced in order to guarantee $O(\log_2 n)$ operations. This assignment asks you to use and modify a Binary Search Tree implementation in order to determine whether the tree is balanced.

Learning Objectives

In completing this assignment, you will:

- Apply what you have learned about how Binary Search Trees represent and store data
- Implement tree traversal algorithms for determining the structure of a tree
- Modify an existing Binary Search Tree implementation

Getting Started

Start by downloading **BinarySearchTree.java**, which implements a BST similar to the one we saw in this module, using Java Generics.

The implementation that we have provided ensures that all elements in the tree are distinct, i.e. have different values.

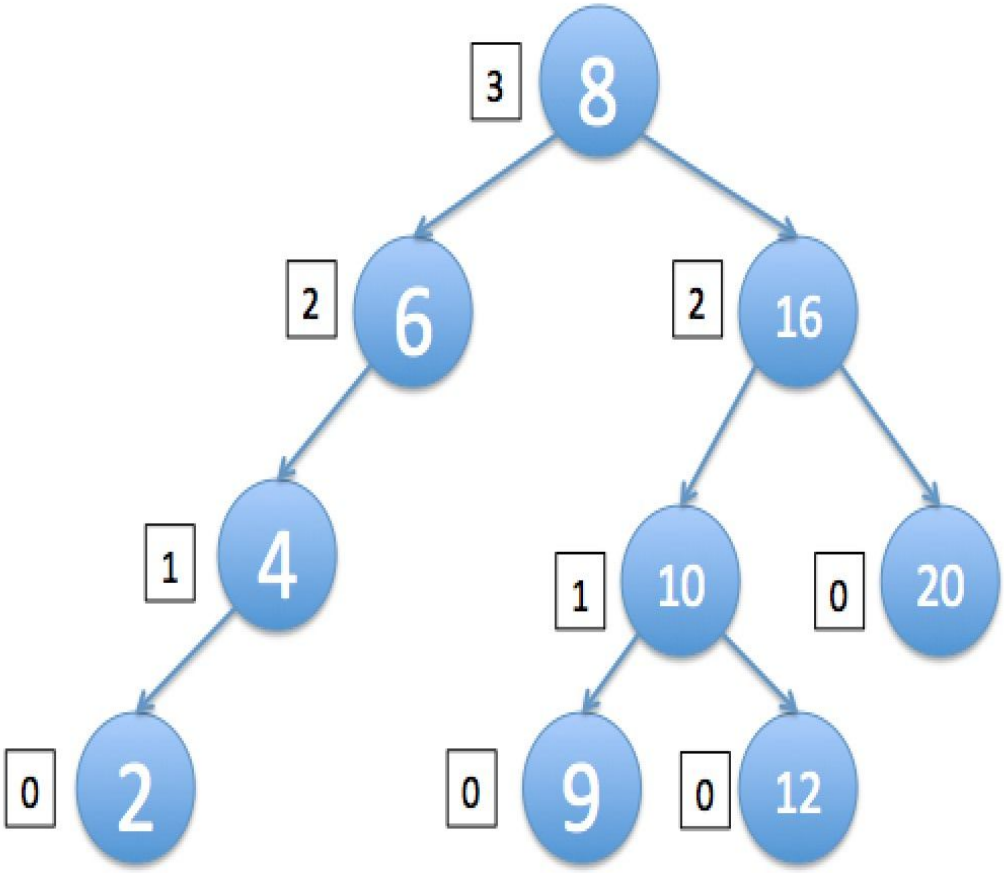
Activity

First, implement the **findNode** method in **BinarySearchTree.java**. Given a value that is stored in the BST, it returns the corresponding Node that holds it. If the value is null or does not exist in this BST, this method should return null.

Then, implement the **depth** method. Given a value, this method should return the “depth” of its Node, which is the number of ancestors between that node and the root, including the root but not the node itself. The depth of the root is defined to be 0; the depth of its two children (if any) is defined to be 1; the depth of the root’s grandchildren (if any) is defined to be 2; and so on. If the value is null or does not exist in this BST, this method should return -1.

Next, implement the **height** method. Given a value, this method should return the “height” of its Node, which is the greatest number of nodes between that node and any descendant node that is a leaf, including the leaf but not the node itself. The height of a leaf node (i.e., one which has no children) is defined to be 0. If the input value is null or does not exist in this BST, this method should return -1.

The image below shows a binary search tree with the height of each node indicated to its left:



In the diagram above, the height of the node labeled 6 is 2, because there are two other nodes (labeled 4 and 2) from that node to a leaf, which is labeled 2. The height of the node labeled 16 is 2, because the maximum number of nodes between it and a leaf is 2; note that we don't consider the leaf labeled 20 because we're looking for the maximum number of nodes to a leaf.

Next, implement the **isBalanced(Node)** method. Given a Node, return true if the absolute value of the difference in heights of its left and right children is 0 or 1, and return false otherwise, as in the AVL Tree implementation we saw in this module. If the Node is null or does not exist in this BST, this method should return false.

As an example, in the diagram above, the node labeled 16 should be considered balanced, since the height of its left child (labeled 10) is 1, and the height of its right child (labeled 20) is 0, and |1 - 0| <= 1.

Note that if a Node's child is null, then the height of that child should be considered as -1. In the diagram above, the node labeled 6 should not be considered balanced, because its left child (labeled 4) has a height of 1, and its right child is null, meaning its height is -1. Since the difference is |1 - (-1)| = 2 and that is greater than 1, this node should be considered unbalanced.

Finally, implement **isBalanced()** so that it returns true if *isBalanced*(*Node*) returns true for *all* Nodes in the tree. This method then allows the user of the BST to determine whether the BST is balanced, using the methods you've implemented above. Note that the root being balanced does *not* imply that the entire tree is balanced (see hint below).

Please do not change the signatures of these five methods, and do not create any additional .java files for your solution. You can, of course, add to or modify the *BinarySearchTree* class and the inner *Node* class if you'd like, but if you need additional classes, please define them in *BinarySearchTree.java.* Also, please make sure your *BinarySearchTree* class is in the default package, i.e. there is no "package" declaration at the top of the source code.

Helpful Hints

For *findNode*, keep in mind that binary search trees are necessarily stored such that, for any node, smaller values are in its left subtree and larger ones are in its right subtree; this is key to writing an efficient method to locate a node by its value, and you can then use it in other methods.

For *depth* and *height*, consider the tree traversal techniques discussed in the lesson and think about how you can use them for navigating a node's ancestors and successors. If you have trouble understanding the difference between "depth" and "height," there is a good explanation at https://stackoverflow.com/questions/2603692/what-is-the-difference-between-tree-depth-and-height

As mentioned above, *isBalanced( )* should not simply return true just because *isBalanced(root)* is true. That is, even if the root is balanced, the tree may not be, as shown in the diagram above: in this case, the heights of the root's left and right children are equal; however, the node labeled 6 is not balanced, therefore the tree is not balanced.

Before You Submit

Please be sure that:

- your *BinarySearchTree* class is in the default package, i.e. there is no "package" declaration at the top of the source code
- your *BinarySearchTree* class compiles and you have not changed the signatures of the *findNode*, *depth*, *height*, and two *isBalanced* methods
- you have not created any additional .java files

Penn Engineering
ONLINE LEARNING

How to Submit

After you have finished implementing the *BinarySearchTree* class, go to the "Module 6 Programming Assignment Submission" item and click the "Open Tool" button to go to the Codio platform.

Once you are logged into Codio, read the submission instructions in the README file. Be sure you upload your code to the "submit" folder.

To test your code before submitting, click the "Run Test Cases" button in the Codio toolbar.

As in the previous assignment, **this will run some but not all of the tests that are used to grade this assignment.** That is, there **are** "hidden tests" on this assignment!

The test cases we provide here are "sanity check" tests to make sure that you have the basic functionality working correctly, but **it is up to you to ensure that your code satisfies all of the requirements described in this document.** Just because your code passes all the tests when you click "Run Test Cases" doesn't mean you'd get 100% if you submit the code for grading!

When you click "Run Test Cases," you'll see quite a bit of output, even if all tests pass, but at the bottom of the output you will see the number of successful test cases and the number of failed test cases.

You can see the name and error messages of any failing test cases by scrolling up a little to the "Failures" section.

Assessment

This assignment is scored out of a total of 72 points.

The **findNode** method is worth a total of 14 points, based on whether it returns the correct Node and handles errors.

The **depth** method is worth a total of 13 points, based on whether it correctly calculates the height of the node and handles errors.

The **height** method is worth a total of 10 points, based on whether it correctly calculates the height of the node and handles errors.

The **isBalanced(Node)** method is worth a total of 17 points, based on whether it correctly determines whether the node's subtrees are balanced and correctly handles errors.

The **isBalanced()** method is worth a total of 18 points, based on whether it correctly determines whether the tree is balanced.