CIT 594 Module 6 Programming Assignment

Binary Search Trees

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 constitutes an ordered set of values. Since uniqueness of values is maintained by calling each value's `compareTo` method, the implementation will reject any attempt at adding a null value to the set.

Activity

Implement the following specifications in **BinarySearchTree.java**. Do not change the signatures of these five methods, and do not create any additional .java files for your solution. You may add to the *BinarySearchTree* class and the inner *Node* class if desired, but if you need additional classes, define them in **BinarySearchTree.java**. Also, ensure your *BinarySearchTree* class is in the default package, i.e. that there is no "package" declaration at the top of the source code.

`Node findNode(E)`

Given a value that is stored in this BST, this method returns the corresponding Node that holds it, or null if the value is null or does not exist in this BST.

`int depth(E)`

Given a value, this method returns the *depth* of its Node, or -1 if the value is null or does not exist in this BST. The depth is defined as the number of edges from that node to the root.

The depth of the root is 0, and its immediate children (if any) would have a depth of 1.

`static int height(BinarySearchTree<?>.Node)`

For the subtree rooted at the given Node, this method returns the *height* of the subtree. The height of a subtree is defined as the maximum number of edges between the root of the subtree and any descendant leaf. The height of a leaf is 0. The height of an empty subtree, represented as a null Node, is defined to be -1.

`static boolean isBalancedNode(BinarySearchTree<?>.Node)`

Given null, returns true. Given an instance of Node, returns true only if the absolute value of the difference in heights of its left and right children is less than 2.

`boolean isBalanced()`

This method returns false if and only if there exists any node in the tree for which `isBalancedNode`(n) returns false; otherwise it returns true.

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*, *isBalancedNode*, or *isBalanced* methods
- you did not overload any exposed methods
- you have not created any additional .java files
- you have filled out the required academic integrity signature in the comment block at the top of your submission file

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.

You must manually submit when you are done. Your code will not be automatically submitted at the deadline.

Assessment

This assignment is scored out of a total of 71 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 9 points, based on whether it correctly calculates the height of the node and handles errors.

The **isBalancedNode** 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.

Optional Challenges

Extra tasks for your understanding, discussion, practice and a little fun. These **will not be graded**.

Add Big-O runtimes

Add a comment to the javadoc of each method you implement that includes the asymptotic runtime complexity (Big-O) of your implementation and whether or not you think it's optimal.

Implement `boolean isBalancedFast()`

This method produces the same result as `isBalanced()` but does so in $O(n)$ time.

Note there are no extra points for doing this task, so write the graded `isBalanced()` first and as cleanly as you can without worrying about performance. Only implement the alternative method if you are comfortable with the rest of your submission.

Frequently Asked Questions

What is the difference between a Node and a subtree?

Recall from m2pa that we referred to linked lists as "recursive data structures", where a (raw) list was either empty (represented by the value `null`) or non-empty (represented as a non-null Node consisting of a value and a pointer to another linked list). In particular, recall the point we made that every node in a linked list can be *thought of* as being the head of its own linked list.

Same thing here, but now our non-null nodes contain *two* pointers: to a left "BST" and a right "BST", and every node can be thought of as the "root" of its own little binary search tree. In this sense, values of type Node can be used to represent *subtrees* rooted at that node, where `null` child nodes encode empty subtrees.

So it's a conceptual difference. When we speak of subtrees, we implicitly include the node at the root of that subtree along with all of its descendant nodes, while when we speak only of nodes, we mean just the concrete data structure containing a value and two pointers.

How do I implement `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. You can then use it in other methods.

Not very subtle hint: this method could have been called "binary search". Make sure you get this right (refer heavily to the lecture material as needed) since this algorithm is critical in this assignment and ubiquitous in computing generally.

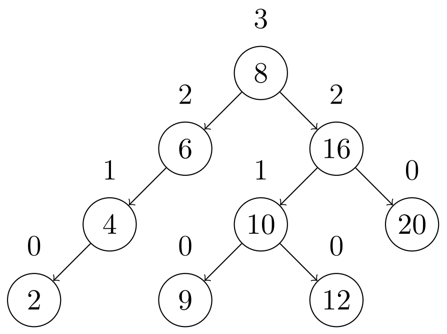
What is the difference between depth and height?

There is a good explanation at StackOverflow.

For implementation, 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.

I don't understand the definition of *height*.

The image below shows a binary search tree with the height of each node indicated on top:



The height of node 6 is 2, because there are two edges from node 6 to node 2, which is its deepest leaf. The height of node 16 is 2, because the maximum number of edges 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 edges to a leaf, and nodes 9 and 12 are the deepest descendant leaves from node 16.

When is a node considered balanced?

As an example, in the diagram shown in the previous question, node 16 should be considered balanced, since the height of its left child (node 10) is 1, and the height of its right child (node 20) is 0, and $|1 - 0| < 2$.

Recall in the definition of *height* that an empty subtree is defined to have a height of -1. In the diagram above, node 6 should not be considered balanced, because its left child (node 4) has a height of 1, and its right child is empty, so the right child has height -1. Since the absolute value of the difference of the heights is $|1 - (-1)| = 2$, this node should be considered unbalanced.

Is an empty tree balanced?

Yes, because an empty tree has no unbalanced nodes.

What is the difference between `isBalancedNode(BinarySearchTree<?>.Node)` and `isBalanced()`?

Notice that `isBalancedNode` is static and takes a subtree of any BST (not necessarily *this* BST) as a parameter, while `isBalanced` takes no explicit parameters, but it's non-static (so it actually has `this` as an implicit parameter).

So `isBalancedNode` takes any Node from any BST and reports whether that single Node is balanced, but it doesn't say whether the entire subtree starting from that Node is balanced. On the other hand, `isBalanced` reports whether an entire tree is balanced, meaning that every single Node in the tree is balanced.

Referring to the diagram shown 3 questions ago, calling `isBalancedNode` on the root node 8 should return true, but calling `isBalanced` on the whole tree should return false, because even though the root alone is balanced, `isBalancedNode` returns false on node 6. Put simply, a tree is only balanced if *all* of its subtrees are balanced.

How do I construct BSTs and call these methods from my test files?

First, do not attempt to write `import BinarySearchTree.Node`. You can only import classes inside named packages. You can't import from the default package, since it has no name. This is just a quirk of Java.

Second, pay attention to which methods are static and which are not. Static methods aren't called on an instance of a *BinarySearchTree*, they're called on the class itself and can take the Node of any BST as parameter.

For your own JUnit tests, see the below examples for how you could call each method:

```java
BinarySearchTree<Integer> bst = new BinarySearchTree<>();
bst.add(5);
int d5 = bst.depth(5);
BinarySearchTree.Node n5 = bst.findNode(5);
int h5 = BinarySearchTree.height(n5);
boolean b5 = BinarySearchTree.isBalancedNode(n5);
boolean b = bst.isBalanced();
```

If you wanted to directly create nodes yourself instead of getting them back from `findNode`, you could write this:

```java
BinarySearchTree<Integer> bst = new BinarySearchTree<>();
BinarySearchTree<Integer>.Node node = bst.new Node(5);
```

Just be aware that the above variable `node` would not actually be part of the binary search tree named `bst`, as it hasn't been added to it. It would just be a disconnected object of type Node that has a *reference* back to `bst`'s instance (and therefore shares its type binding for E, etc.). A production-ready implementation of a BST would likely make the inner Node class private to prevent this; however, we need to expose Node for our test cases to be able to access it.

As another example, this expression should return true, even though the constructed Node is detached from its enclosing BST:

```java
BinarySearchTree.isBalancedNode(new BinarySearchTree<String>().new Node("hi"))
```

Would it be better to use recursion or iteration to write these methods?

We recommend writing functions recursively first, as this follows the natural structure of binary search trees and is a good learning exercise. An equally good exercise is to then translate your recursive functions to iterative (loop-based) functions.

A comment on performance: Java is not an ideal programming language for recursion. Languages designed for recursion use an optimization technique that results in tail-recursive functions being compiled down to machine instructions that make them indistinguishable from loops. *Java doesn't do this.* In Java, C, and Python, loops are strictly more performant and have fewer limitations than recursion. Since these languages don't perform tail-call optimization, you'll not only see the high overhead of making a function call, but you also risk blowing the stack if you make too many recursive calls. So if you want to write industrial-strength Java, you'll typically avoid recursion.

All that said, none of this is a problem for this assignment. The test cases are not even remotely large enough to produce a stack overflow (unless you accidentally write an infinite recursion). You typically need 1000 or more recursive calls to trigger a stack overflow.

As an aside, even if you did choose to use recursion to implement a BST in C, Python, or Java, stack overflow wouldn't be a problem if you maintain the *balance property* of a balanced binary search tree. Think about why this would be the case.

But if iteration is better in Java, why do you use so much recursion in the starter code?

Yes, `add` and `remove` use recursion heavily and in a way that you wouldn't want to use in industrial strength Java. Our purpose here is to optimize the code for demonstrating recursive thinking – not at all for machine-level efficiency. In particular, we are attempting to make it easy to match it the `remove` method against the high-level description of the node deletion algorithm given in *Data Structures and Algorithms in Java* (Section 11.1.2, page 464).

May I add helper methods to the class?

Yes, in fact you'll need helper methods if you want to use recursion.

However, we strongly recommend that you either give your helper methods a different name from the methods in the starter code, or else give them `private` visibility so that they can't accidentally be triggered by our test cases. Here's why: suppose there are two methods with the same name that are visible to the test cases: `myFunction(A a)` and `myFunction(B b)`. If one tries to call `myFunction(null)`, the compiler has no way of knowing whether this call is referring to `myFunction(A)` or `myFunction(B)`, since null can take on any type.

(Side note for the curious: there are ways around the above problem. You can actually coerce null to a type like so: `myFunction((A) null)`, which tells the compiler that you want to call `myFunction(A)` with null. But, you generally can't expect those who call your methods to know that they need to do this.)

Besides the assigned method implementations, may I modify the starter code?

Do not modify existing methods or fields except for the methods that you are supposed to implement. You may add additional fields, methods and inner classes as needed, but you must use different names for any added methods (do not overload).

Much of the starter code is provided as examples of how to work with trees; you are welcome to copy and adapt as much of it as you like into the methods you implement.