Lab 10 - Binary Search Trees
Due (via Bitbucket and hard copy) Wednesday, 27 April 2016
50 points

Lab Goals

- Review construction techniques for trees
- Implement a Binary Search Tree
- Answer a few questions about trees

Assignment Details

For the past week, we have been implementing a generic Binary Tree during the lectures. This generic Binary Tree allowed us to store data inside of the tree in whatever order we wanted, specifying the location to insert a new node based on which parent node we wanted, and whether the new node should be a left child or a right child.

While our generic Binary Tree is a useful data structure, traversing the tree to find an item is on the same complexity level as traversing a list – searches take $O(n)$ time, which in turn mean that insertion and deletion, which both depend on search, take $O(n)$ time. Surely there must be a more efficient system!

And there is indeed. A Binary Search Tree is a special kind of Binary Tree, in which information about the location of a Node is encoded into every other Node that you visit starting at the root of the tree. We compare the value we are inserting (or seeking or removing) to the current Node we are at. If the current Node is larger than our data, we will proceed to the left child; if the current Node is smaller than our data, we will proceed to the right child.

In this lab, you will implement a Binary Search Tree. You can use the BinaryTree.java code that we have created in class as a starting point, but the functions will need to be modified to reflect the new information propagated through the tree. Alternatively, you can use framework code provided in the lab9/src directory, and just fill in the functions that are empty. The details of each of the functions to will implement are specified in the next few subsections.

Part 1: insert() Method (15 points)

In our generic Binary Tree implementation, we needed to pass three parameters into our add() function – the value of the Node we wanted to insert, the value of the parent Node we wanted to insert below, and an option for inserting the new Node as a left or right child. In a Binary Search
Tree, we only need to specify a single parameter – the value of the new \texttt{Node}. All of the other information can be gained simply by examining the nodes in the tree.

For example, consider the Binary Search Tree example in Figure 1. Say we wanted to insert a 35 value into the tree. We call \texttt{insert(35)}, which will begin traversing the tree looking for where to insert this new node, beginning at the root. We compare the 35 value to the 37 in the root. Because the root is greater than 35, we will follow its left child. Now we are at node 21. We compare the 21 to the 35 that we plan to insert and find that the 21 is smaller, so we follow the right child. Now we are at node 29. Again, we compare the 29 to the 35 that we plan to insert, and find that the 29 is smaller, so we follow the right child. Now we are at a node that does not exist. Because 29 was a leaf node in our Binary Search Tree, and because 35 is greater than 29, we will then insert a new \texttt{Node} with a value of 35 into the right child of 29. The new tree after this insertion is shown in Figure 2.

Figure 1: A sample Binary Search Tree

Figure 2: The same Binary Search Tree after inserting the “35” \texttt{Node}
In this section, you will need to write an `insert()` function which takes only the integer value of the `Node` you want to insert as a parameter. This `insert()` function will call a recursive second `insert()` function with an additional parameter – the current `Node` location in the tree. This recursive function will traverse the tree, similar to the generic Binary Tree that we wrote in class, that will decide whether to follow the left or right child paths by comparing the value in the current location to the value we wish to insert. When a leaf `Node` is reached, the function will insert a new `Node` appropriately as the left or right child of that leaf.

**Part 2: get() Method (10 points)**

After implementing the `insert()` function, the `get()` function should be trivial. Using a similar approach, we wish to determine whether or not a `Node` with a given value exists in the Binary Search Tree. For example, consider our updated tree from Figure 2. We want to know if a “9” `Node` exists in the tree. We begin with a “setup” `get()` function which takes as a parameter only the value that we are looking for, and calls a recursive `get()` function that starts at the root node. We compare the 9 we are searching for to the 37 in the root, note that the 9 is smaller, and take the left child edge. We then compare the 9 we are searching for to the 21 stored in our current node, note that the 9 is smaller, and take the left child edge. Finally, we compare the 9 we are searching for to the 9 stored in the current node, note that they are the same, and can return the fact that we found the 9 up the recursive call stack.

On the other hand, if we were searching for a “123” `Node` in the tree, we would compare the 123 value to the 37 at the root and go right, then compare the 123 to the 71 in the right child and go right, and then end up at a null `Node`. Because we hit this null node, we know that a 123 value does not exist in the Binary Search Tree, and can return that fact by returning the null value.

In this section, you will write both the setup and recursive `get()` functions. The functionality should be nearly identical to the functionality of the `insert()` functions, but without actually inserting any new data to the Binary Search Tree.

**Part 3: remove() Method (15 points)**

The `remove()` method is by far the trickiest of the three methods that you are asked to implement in this lab, and in fact we will not consider all of the possible nodes for removal. Instead, we will consider two cases:

1. We are trying to remove a `Node` with no children.
2. We are trying to remove a `Node` with a single child.

Let’s start by considering the first case, removing a `Node` with no children. In this case, we know that the `Node` we plan to remove is a leaf node. We can traverse the tree as we did in the `insert()` and `get()` functions, and when we find the `Node` we are looking for, we can simply travel back to the parent and set the appropriate child link to `null`.

It gets a little bit more difficult if we are trying to remove a node with a child. Similar to the doubly linked list, we will need to patch the tree in both directions. Considering again the Binary
Search Tree in Figure 2, if we want to delete Node 29, we will need to set Node 21’s right child to be 35, and we will need to set Node 35’s parent to 21. This will effectively delete the 29 Node from the tree, since it will be unreachable. Again, you can search through the tree until you find the Node that you are looking to remove, then patch the tree at that location. Remember that the child that gets reattached could either be a left or right child.

Part 4: Additional Questions (10 points)

Please answer the following questions thoroughly:

1. Given a set of \( n \) values to insert, what is the worst-case height of a binary tree? Draw a tree that demonstrates this case.

2. Given a set of \( n \) values to insert, what is the best-case height of a general (not necessarily binary) tree? Draw a tree that demonstrates this case.

3. Rather than storing the overall size of the tree in a \textit{size} variable in the BinaryTree class, we could make \textit{size} a property of each Node. In this case, the size is a Node is equal to itself, plus the size of its left subtree, plus the size of its right subtree. Again considering the tree in Figure 2, the Nodes 9, 35, and 71 each would have a size of 1, Node 29 would have a size of 2, Node 21 would have a size of 4, and Node 37 would have a size of 6. Explain how you could traverse a given tree to calculate the \textit{size} variable for each Node. (Pseudocode, actual code, and English descriptions are all acceptable.)

Part 5: Extra Credit (5 points)

The one case that we did not consider for deletion is the case in which the Node we want to remove has two children. For example, consider the tree in Figure 3 on the next page. We want to delete the Node 21, which certainly looks like it will be a difficult prospect, especially if we want to maintain the “left = smaller, right = bigger” relationship between all of the nodes in the tree. For up to 10 points of extra credit, you will find the next largest Node in the tree that is bigger than 21. We call this node the “successor.”

Luckily, the successor is not particularly difficult to find, thanks to the aforementioned “left = smaller, right = bigger” relationship. To find the next largest node, we will go to the right child of 21, guaranteeing that we will find something larger, then follow all of the left child links until we can no longer go left, guaranteeing that we have found the smallest item in the right subtree. In this case, we only have one left child link to follow before we find the successor, which turns out to be the 25 Node. Our goal is to replace the 21 Node with the 25 Node.

Unfortunately, the 25 Node also has a child, but because we know that it has no left child, we know that it can only have one. This is good, because we can use our earlier remove() function to temporarily pull the 25 Node out of the tree, and patch the tree so that the 27 Node is now the left child of the 29 Node.

Now, we want the 25 Node to take the place of the 21 Node. This means that 37’s left child will be 25 and 25’s parent will be 37, 25’s left child will be 9 and 9’s parent will be 25, and 25’s right
child will be 29 and 29's parent will be 25. In other words, every relationship that once existed between the 21 node and the 9, 29, and 37 nodes will now exist between the 25 node and the 9, 29, and 37 nodes. The final state of this tree is shown in Figure 4.

An alternative solution is to simply find the successor node, remove it and patch the tree at that point, and then simply take the value from the 25 node and place it into the 21 node, overwriting the 21. Because this solution take a somewhat easier approach to tree patching, it will be awarded up to 3 extra credit points.
Submission Details

For this assignment, your submission (to both your BitBucket repository and by hardcopy) should include the following:

1. (Upload) The source code for your Binary Search Tree, including the methods that you wrote to insert, retrieve, and delete information from the tree.

2. (Print and Upload) Proof that your tree works as intended with the provided `TreeTester.java` code.

3. (Print and Upload) The answers to the questions from Part 4.

4. (Print and Upload) An Assignment Information Sheet for your code submission.

Before you turn in this assignment, you also must ensure that the course instructor has read access to your BitBucket repository that is named according to the convention `cs112s2016-<your user name>`.