Project 4 Instructions

Due Date: Sunday, December 10th @ Midnight

Tasks:

1. Import the project from GitHub. You will be emailed a link.
2. Implement the add, contains, and remove methods.
3. Complete the block comments for contains and remove methods.
4. Complete the header comment.

Recursive definition of Binary Search Tree:

A set of nodes T is a binary search tree if either of the following are true:

- T is empty
- If T is not empty, its root node has two sub-trees, T-left and T-right, such that T-left and T-right are binary search trees and **the value in the root node of T is greater than all values in T-left and is less than all values in T-right.**
- The nodes of the tree are a set (no duplicate values).

Why use a Binary Search Tree?

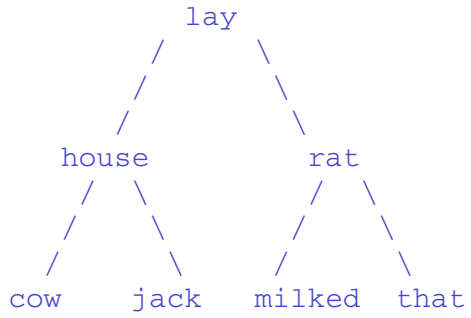
The reasoning for storing values in a search tree is to minimize the number of steps required to find the item. In a worst case scenario, we would have to take a number of steps equal to the height of the tree. Ideally the tree is Balanced, meaning both left and right subtrees have about the same height and most of the nodes up until the last level are full. A Complete binary tree is a good example of a Balanced tree. By observing the fact that the number of nodes at each level of the tree double when the tree is Balanced, then the size (n) of the tree is n = 2 ^ height. If we solve for the height of the tree, we get the worst case scenario for searching a balanced tree: O(log n).

Recursive algorithm for searching a Binary Search Tree:

1. If the root is null, return false
2. If the target is equal to root.data, return true
3. Else if the target is less than root.data, return the result of searching the left subtree
4. Else return the result of searching the right subtree.

Keep in mind that the term "root" does not necessarily refer to the root field of the BinarySearchTree.java class. It refers instead to the recursive definition of a binary search tree where every sub-tree is a binary tree with its own root.

Example for searching a Binary Search Tree:



If we wish to see if the tree contains "jack", first compare "jack" with "lay". Because "jack" is less than "lay", we continue the search with the left subtree and compare "jack" with "house". Because "jack" is greater than "house", we continue with the right subtree and compare "jack" with "jack". Since they are equal, we

return true. However, if we were searching for "jill", we would have continued down the same path, and seeing that "jill" is greater than "jack", have tried to search the empty subtree on the right and thus concluded "jill" is not in the tree.

Recursive algorithm for insertion into a Binary Search Tree:
1. If the root is null, replace empty tree with a new tree with the item at the root and return true
2. Else if the item is equal to the data at the root, then the item is already in the tree; return false
3. Else if the item is less than root.data, return the result of inserting the item in the left subtree
4. Else return the result of inserting the item in the right subtree

This is a generic algorithm. It does not take into account our particular implementation of the BinarySearchTree class. Implementation options:
-**Option A:** Check in advance if the child is null. If it is, assign it a new Node with the given item and return true.
-**Option B:** Do not check in advance if the child is null. Instead, reassign the child to be the result of calling a private add method on the child. Since the public add method returns a boolean, but this strategy returns a Node, you will need to create a global boolean flag which will be used to indicate if the item has been inserted (true) or is a duplicate (false).

For example:
```
private Node<E> add(Node<E> node, E item){
      // If node is null, set the flag to true, return a new node with the item
      // If the item is already in node, set the flag to false, return node
      // Otherwise, call add(child) on the appropriate subtree of node and then
      // return the node. i.e. node.left = add(node.left, item); return node;
}
```

Recursive algorithm for removal from a Binary Search Tree:
1. If the root is null, return false
2. Else if the item is less than the root, return the result of removing from the left subtree
3. Else if the item is greater than the root, return the result of removing from the right subtree
4. Else, the item must equal the root, and the root needs to be removed.
4a. If the root has no children, the root becomes null.
4b. If the root has a single child, the root becomes that child.
4c. If the root has two children, the root becomes the largest value in its left subtree. (Hint: It's called the *inorder predecessor* and it's always located in the same place.) The node containing that value must also be removed. Return true.
As with the add method, **Options A and B** apply here as well.

Tips:
- Figure out your base cases (i.e. when you don't recurse).
- If you have a base case where the given node is null, check that base case first to avoid null pointer exceptions.
- Use pencil and paper to come up with examples to use for crafting your algorithm.
- If you hit a bug, use the Debugger to find the problem, then refine your algorithm.
- Attempt this during Thanksgiving break.
- Come to lab after Thanksgiving break with your questions. I will help you work on the project.

Submission:

Zip and submit the BinarySearchTree.java file to ECampus.
Due to time constraints on grading, late submissions will not be accepted.