CS 314 Final Review — Binary Search Trees — **Solution**

Verify a Binary Search Tree

Suppose we were given an implementation of a BinarySearchTree but were doubtful of its correctness. Let's write an instance method for a BinarySearchTree class which will verify if `this` is a valid BinarySearchTree.

You will need to ensure that all nodes in `this` follows the BinarySearchTree rule that, for any given node, all nodes' values in its left subtree are less than the node's value and all nodes' values in its right subtree is greater than the node's value. Also, make sure that the `size` instance variable correctly stores the number of nodes in the tree.

Complete the following method.

```
// pre: none
// post: returns true iff the BST is valid
//       this object is unaltered as a result of this call
public boolean verifyBST() {
```

You may use the following `BinarySearchTree` implementation

```
public class BinarySearchTree<E extends Comparable<? super E>>{
  private BSTNode<E> root;
  private int size;

  private static class BSTNode<E extends Comparable<? super E>>{
    private E data;
    private BSTNode left, right;
  }
}
```

You may create a single int array of size 1.
Do not create any other data structures or use any other Java classes or methods.

```
// pre: none
// post: returns true iff the BST is valid
//       this object is unaltered as a result of this call
public boolean verifyBST(){
    if(root == null)
        return size == 0;
    int[] nodeCount = new int[1];
    boolean valid = validHelper(root, nodeCount, getMin(), getMax());
    return valid && nodeCount[0] == size;
}

public boolean validHelper(BSTNode<E> n, int[] nodeCount, E min, E max){
    if(n == null)
        return true;
    nodeCount[0]++;

    //Already found too many nodes, return early
    if(nodeCount[0] > size)
      return false;

    //This doesn't check for ties because the absolute mins/maxes in the
    //tree will be equal to the min/max parameter
    if(n.data.compareTo(min) < 0 || n.data.compareTo(max) > 0)
        return false;

    //We have to check for ties here instead, where min and max will
    //be leaf nodes
    if(n.left != null && (n.data.compareTo(n.left.data) <= 0)
        return false;
    if(n.right != null && n.data.compareTo(n.right.data) >= 0)
        return false;

    return validHelper(n.left, nodeCount, min, n.data) &&
        validHelper(n.right, nodeCount, n.data, max);
}

//Get the maximum value in the tree
private E getMax(){
    BSTNode<E> n;
    for(n = root; n.right != null; n = n.right);
    return n.data;
}

//Get the minimum value in the tree
private E getMin(){
```

```
        BSTNode<E> n;
        for(n = root; n.left != null; n = n.left);
        return n.data;
    }
```

This problem requires you to really understand the structure and properties of BinarySearchTrees. For instance, the problem stated that a node's value must be greater than *all the values in its left subtree*. However, we don't have to go check the entire left subtree right away. Instead, as we descend down the tree, we shrink the range of possible values the nodes can hold. When we go down the left subtree, we restrict the maxium value lower nodes can hold; when we go down the right subtree, we restrict the minimum value the lower nodes can hold. We initialize the minimum and maximum values in the kickoff method by finding the absolute minium and maximum values of the tree. Also, this solution uses a single pass to check both properties of the tree. It accomplishes this by passing a reference to an int array of size 1 which holds the number of nodes seen so far. If we encounter more nodes than what we expect based on `size` then we can return early from our recursive calls.