1. private void updateHeight(TreeNode$< T >$ root);
   The updateHeight method works by going through four cases to finding the height of the parameter TreeNode$< T >$ root, which is also an instance variable. The four cases are when the root has no children, a right child, a left child, or both children. Each case follows the formula of $height = max(rightchild, leftchild) + 1$. And if one child is null the height will just be the other child's height plus 1. No helper methods are called.

2. private int getBalance(TreeNode$< T >$ root);
   The getBalance methid returns the balance of the instance variable/ parameter root. In this method there are four cases. One is if the root has no children, in which case the balance is 0. The second and third are if the root has either the left or right child. Lastly, if the root has both children. All cases follow the formula, left child's height minus the right child's height. If the child is null it's value was -1. No helper methods were involved.

3. private TreeNode$< T >$ rebalance(TreeNode$< T >$ root);
   The reBalance method returns the root of the subtree after it rebalances. It uses the instance variable root. Here the method checks to see if the tree is left-left leaning, left-right leaning, right-left leaning or right right leaning by calling on the getBalance method as a helper method. When this is known, the reBalance method can then use other helper methods (rightRotate and leftRotate) as needed to balance the tree.

4. private TreeNode$< T >$ rightRotate(TreeNode$< T >$ root);
   The rightRotate Method, rotates the the tree right at the root specified. It does this by saving the root's left right child as a dummy variable then making the root the right child of the roots left child and the original root has the left right child added as its left child. No helper methods were used in this method.

5. Briefly describe the modifications you made to the insertion and removal functions to maintain height and keep the tree balanced. Where did you initialize the height of a new node?//

   The modification done in the insertion and removal functions to maintain height and balance were right before the return statement. There I updated the height of the tree since now it had an extra node, which will be needed in evaluating the rebalancing part.. Then in the return statement I called rebalance to maintain the balance of the tree. In that method is will then change the tree if needed and again update the height of the tree to make it correct.

6. Suppose we implement firstAfter() as a top-down tree walk, similar to find(). What should we do if the root of the tree has a key < v? Justify why the behavior you specify is correct.

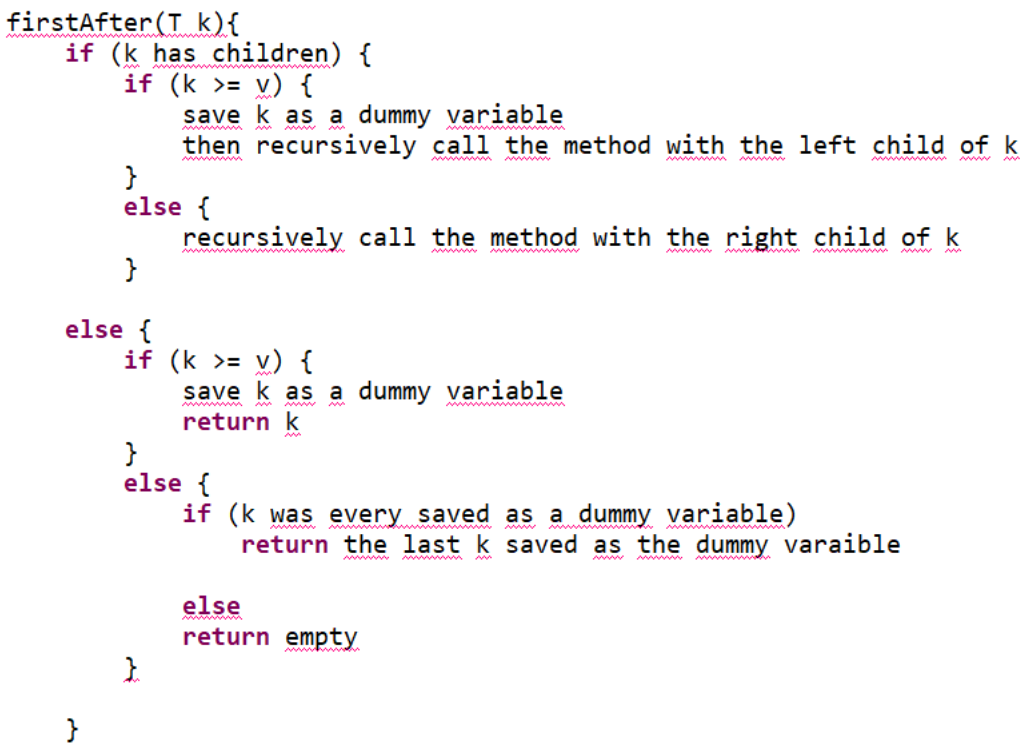
   If there is a key less than v, then the method should continue searching, by checking the right child of the key. If the key is a leaf node then stop and say the least element found. In the AVL Tree v, might be larger or smaller than the root. So you need to compare the two. The first action taken can be justified because if the root was less than v and has a right child then the search can continue for a key that would be greater than or equal to v. The second one can be justified since there is no other place to look after a leaf node, the method should stop searching.

7. If the root has a key k v, what should we do to determine whether k is the least key v? Justify why the behavior you specify is correct.

   The Key k should be saved in a dummy variable and the search should continue down the to the left child of the root. The value v should then be compared again and if the child had a key k that was

$\geq$ v but less than the root value, we now save this value as the dummy variable and again continue down the tree until it compares its value to a tree node.

8. Based on your answers above, give pseudocode for an implementation of firstAfter() that runs on a BST in time proportional to its height.



```
firstAfter(T k){
    if (k has children) {
        if (k >= v) {
            save k as a dummy variable
            then recursively call the method with the left child of k
        }
        else {
            recursively call the method with the right child of k
        }

    else {
        if (k >= v) {
            save k as a dummy variable
            return k
        }
        else {
            if (k was every saved as a dummy variable)
                return the last k saved as the dummy varaible

            else
            return empty
        }

    }
```