Questions

1. `updateHeight` takes the maximum of the left and right children heights and adds 1 to give the height of the current node using the `height` instance variable.

2. `getBalance` takes the difference of the left and right children heights to give the balance factor of the current node using the `height` instance variable. Returning a positive value means the left subtree is taller, 0 means the tree is balanced, and a negative value means the right subtree is taller.

3. `rebalance` checks the balance factor with `getBalance` (helper) and performs the rotations if it unbalanced based on `getBalance` where we rotate for 4 cases: left-left, left-right, right-right, and right-left. To rotate the tree we use helper functions `leftRotate` and `rightRotate`.

4. `rightRotate` rotates the tree (clockwise) by moving the root node to the right child and making the left child the new root node. The `updateHeight` helper function recalculates the height of the new root node, and `setLeft` and `setRight` helper functions fix the pointers from the rotation operation.

5. In the insertion and removal function we have to update the height of the rood node then check if the tree is unbalanced and rebalance it if necessary. The height update happens after the insertion or removal operation, and before the rebalancing operation.

6. If `rootKey <` v we go to the right subtree and look again for the least element in the set $>= v$. This is because all the elements in the left subtree will be `<` v so we can ignore them.

7. If `rootKey >=` v we might have found the answer (the root), but we still need to check the left subtree for a potentially least element in the set $>= v$. This is because the left subtree might have a smaller element than the root but still greater than v.

8. PSEUDOCODE

```
        // little helper function
        public T firstAfter(T v) {
            return firstAfterHelper(v, root, null);
        }

        // recursive part
        private T firstAfterHelper(T v, TreeNode<T> root, T bestSoFar) {
            if (root == null) {
                return bestSoFar;
            }

            int comparison = v.compareTo(root.value);

            if (comparison > 0) {
                // node < v ==== look in right subtree
                return firstAfterHelper(v, root.right, bestSoFar);
            } else {
                // node >= v ==== might be ans, but look left subtree
                return firstAfterHelper(v, root.left, root.value);
            }
        }

```