1. Binary Search Tree Modification:

In order to support computing kth order statistic in $\Theta(h)$ time, I add a size field in class Node. For an arbitrary node x, the size field of x stores the number of nodes in the subtree rooted at x (including x itself). As a result, when I am at the root of tree, I can know the size of left subtree of root. If k = leftTreeSize+1, it means the root is the kth smallest element that we are looking for. If k < leftTreeSize+1, it means the element we are looking for is in the left subtree of root and we can skip the right subtree of root. If k > leftTreeSize+1, it means the element we are looking for is in the right subtree of root and we can skip the left subtree of root. The worst case is to go through the height of tree. Thus, computing kth order statistic can be done in $\Theta(h)$ time.

2. **InsertAndUpdateSize** Description:

This method first creates a new node with the input key value. Then it locates the parent node of the new node we insert which is in O(h) time where h is the height of tree. if the tree is empty before we insert the new node, then the new node becomes the root of the tree. Otherwise, the method compares the new node's key with its parent's key and determines which child should it be. This process takes constant time. Finally the method fixes the size fields of the nodes who have the wrong size after the new node is inserted. This process also takes O(h) time. Thus, the whole process of **InsertAndUpdateSize** method takes O(h) time.

InsertAndUpdateSize PSEUDO-CODE

```
function InsertAndUpdateSize(T, z)
y ← NIL;
x ← T.root;
while (x ≠ NIL) do
y ← x;
if (z.key < x.key) then x ← x.left;
else x ← x.right;
end
z.parent ← y;
if (y = NIL) then T.root ← z;
else if (z.key < y.key) then y.left ← z;
else y.right ← z;
end
while (y ≠ NIL) do
    y.size ← y.size + 1;
    y ← y.parent;
end
```

3. **kthSmallest** Description:

The flag represents if or not the kth smallest integer is found. If the kth smallest integer is found, it becomes true. Otherwise, it remains false.

First, we set our reference node to refer the root of tree. Because each node has a size field, I can compare the size of left subtree of reference node with k. If k = leftTreeSize+1, it means the root is the kth smallest element that we are looking for. Then we assign the key of root to our result and set flag to be true which let us get out of the while loop.

If k < leftTreeSize+1, it means the element we are looking for is in the left subtree of root and we can skip the right subtree of root. So we set the left child of root as our new reference node.

If k>leftTreeSize+1, it means the element we are looking for is in the right subtree of root and we can skip the left subtree of the root. So we set the right child of root as our new reference node and subtract (leftTreeSize+1) from k.

The loop body will continue to execute until the kth smallest is found. The worst case is to go through the height of tree, which is in O(h) time.

kthSmallest PSEUDO-CODE

```
function kthSmallest(root, k)
result ←0;
reference ←root;
flag ←false;
while(!flag) do
    leftTreeSize ←0;
    if (reference.left ≠ NIL) then
        leftTreeSize ←reference.left.size;
    if (k == leftTreeSize+1) then
        result ←reference. key;
        flag ←true;
    else if (k > leftTreeSize) then
        k ←k-(leftTreesize+1);
        reference ←reference.right;
    else
        reference ←reference.left;
    end
end
```