COMS 311
Problem 2-4

2.

Worst Case Time Complexity:

This happens when every node must be accessed which is every case, so the time complexity is O(n).

```
public Node[] getMinimumDifference(Node T, int k) {
   int min = Integer.MAX_VALUE;
   Node prev = null;
   Node a, b;
   Stack<Node> stack = new Stack<Node>();
   Node[] result = new Node[2];

   while (T != null || (!stack.isEmpty())) {
      if (T != null) {
         stack.push(T);
         T = T.left;
      } else {
         T = stack.pop();
         if (prev != null) {
            if(Math.abs((T.data + prev.data) - k) < min) {
                           min = Math.abs((T.data+prev.data) - k);
                           a = T;
                             b = prev;
                             result[0] = T;
                             result[1] = prev;
            }

         }
         prev = T;
         T = root.right;
      }
   }

   return result;
}
```

Justify Correctness:

The program starts at the root, and first adds every value in the left subtree to a stack. Then it pops off the top of the stack, which will be th left most child, and attemps to visit its right subtree. If it is not empty, it sets T equal to the right child, and then again recurses through the entire left subtree, adding all nodes to the stack. It steps like this through the whole tree, all left children first, then the right child of the left-most child, then all left children of that right child... etc. It is a depth first search.

3.

ANSWER:

We will used an augmented, balanced BST. Each node will contain the following values:

```
Node parent;
int frequency = (1 for 1 node of this value; 2 for 2 of same values, 3 for 3 of same values... etc)
int numRight =  (Number of values in right subtree" AKA values greater than given node)
```

By doing this, the time-complexity for getting the frequency() or order() of any given element will be the same as searching for that element in a BST. Will update the values of frequency and numRight as we construct the tree

ALGORITHM:

This algorithm adds an element in log(n) time, than performs a log(n) execution to update the weights of parent nodes.

```
Add an element to the tree.
balance tree.
Recursively crawl back through the parent's of the node just added
If this node just added > parent, then add 1 to parent.numRight
```

FUNCTIONS:

//Search uses a helper function called exists, which is like search, except it returns the entire Node if found, else returns null

```
search(x):
        Boolean search(Node root, int x)
                Node result = exists(root, x);
                        if(result != null) return true;
                return false;
```

```
                    Node exists(Node root, int x)
                        if (root==null || root.data==x)
                            return root;
                        if (root.data > x)
                            return exists(root.left, x);
                        return exists(root.right, x);
```

add(x):

```
            void addMe(int x){
                     Node parent = NULL;
                     Node newNode = add(root, x, parent);
                     updateWeights(newNode);
            }
               Node add(Node root, int x, Node parent)
                   if (root == null)
                       root = new Node(x);
                       root.parent = parent;
                       return root;
                   if (x < root.data)
                       root.left = insertRec(root.left, x, root);
                   else if (x > root.data)
                       root.right = insertRec(root.right, x, root);
                   else if(x == root.data)
                        root.frequency += 1;
                   return root;
```

frequency(x):

```
            public int frequency(int x)
                          Node node = exists(root, x);
                          if(node != null) return node.frequency;
                          return 0;
```

order(x):

```
            int order(int x)
                          Node node = exists(root, x);
                          if(node != null) return node.numRight;
                          return 0;
```

updateWeights():

```
                   void updateWeights(Node root) {
```

```
Node parent = root.parent;
int justAdded = root.data;
while(parent != null)
        if(justAdded > parent.data)
                parent.numRight +=1;
```

TIME COMPLEXITIES:

search(x):

At each recursion, the size of the tree being searched is cut in half. So it follows the formular

n + n/2 + n/4 + n/8 ...... for log(n) times

Thus this is O(log(n).

frequency(x):

This uses the search function, and then executs a constant time operation so:

c + O(log(n)) = O(log(n))

add(x) :

At each recursion the size of the tree being recursed is cut in half. So we have :

n + n/2 + n/4 + n/8 ... for log(n) times

Thus this is O(log(n)).

Also, on each added node, we much recurse back through the sub-tree and update weights, which also takes O(log(n)), so total time is:

O(log(n)) + O(log(n)) = O(log(n))

order(x):

This uses the search function, and then executs a constant time operation so:

c + O(log(n)) = O(log(n))

udateWeights():

This starts at a leaf, and moves through the parent nodes until it reaches the root, takes O(log(n)) time.

4.

```
void getLeafNodes(T, root)
      if (root.left==null && root.right==null)
              print (root.data + " " + findOrder(T, root.data));
      if (root.left != null)
            getLeafNodes(T, root.left);
      if (root.right !=null)
        getLeafNodes(T, root.right);


  int findOrder(T, n)
      if(T == null)
          return 0;
      if(T.data == n)
          return 1 + findOrder(T.left, n);
      if(T.data < n)
          return 1 + findOrder(T.left, n) + findOrder(T.right, n);

      return findOrder(T.left, n);
```

Worst Case Runtimes:

The getLeafNodes(T, root) algorithm must always traverse every element in the tree until it eventually reaches a leaf node. This makes its runtime O(n) in all situations. Once the leaf node is found, the findOrder(T, n) algorithm executes. This algorithm will execute for every leaf node in the tree, so total execution time of this step will be :

O(findOrder) * numOfLeafNodes.

The worst case runtime of the findOrder algorithm happens if the element being searched for is the rightmost leaf node (Meaning the largest value in the tree). In this case, the algorithm will always have to recursively check the left and right subtrees. This takes O(n) time also, so the total runtime of both functions is

O(n) + (numOfLeafNodes) * O(n)

Proof of Correctness:

A.) First we will prove the correctness of getLeafNodes(T, root)

1.) We want to prove that for all Binary Search trees of distinct integers rooted at T, this algorithm will print all leaf nodes.

We will prove with induction.

Base Case:

Consider a BST made of only one node, the root (AKA leaf). In this case, getLeafNoes will immediately enter the conditional(root.left==null && root.right==null), and print the value of root.data.

Inductive Hypothesis:

Now consider a root T, with BST's L and R being made of simply one root node as its left and right subtrees respectively. In this case L and R are the leaf nodes of the BST rooted at T.

Induction Step:

We will show that getLeafNodes(T, T.root) will return L and R as its leaf nodes.

In the first iteration of getLeafNodes(T, T.root), both conditionals (root.left != null) and (root.right != null) are entered. On the recursive iteration of the left subtree, we know from our hypothesis that L.left == null and L.right == null, therefore the conditional (root.left == null && root.right == null) will be entered. In this conditional, the value of L.data will be printed. The same can be said for the recursive iteration of the right subtree.

B.) Now we will prove correctness of findOrder(T, n)

1.) We want to prove for all Binary Search Trees of distinct integers rooted at T, findOrder(T, n) returns all numbers <= n.

Base Case:

Again consider a BST made of only one node, the root. We call the root T. In this case findOrder(T, n) will immediately enter the conditional (T.data == n) and return 1 + findOrder(T.left, n). Because this BST is made only of one root, it means its left and right subtress are both null, so the recursion through the left subtree will enter the first conditional of

(T == null) and return 0. Thus the end return statement will be return 1 + 0 = 1.

Inductive Hypothesis:

Now consider a root T, with BST's L and R being made of simply one root node as its left and right subtrees respectively. In this case L and R are the leaf nodes of the BST rooted at T. This presents two new cases to consider. The first case will be recursively looking for all elements <= the right-most leaf, and the second case will be recursively looking for all elements <= the left-most leaf. We know from the definition of a BST that since L is the left sub-tree of T, L.data < T.data, and similarly R.data > T.data. This means that while recursively looking to find the order of the left-most leaf, the value will be equal to 1, and the for the right-most leaf, the value will be equal to the number of nodes in the tree, which is 3.

Inductive Step:

We will show that the reucursion of tree T to find the order of the left-most leaf returns 1, and the recursion of T to find the order of the right-most tree returns the number of nodes in the tree, which is 3.

1.) T.data < n

In this case, the value of n is the right-most leaf of the tree. On the first iteration the program returns:

1 + findOrder(T.left, n) + findOrder(T.right, n);

First Consider the call findOrder(T.left, n):

In the recursion of the left subtree, the value of n will be equal to the left-most leaf and enter the condtional (T.data < n) again where it will return:

1 + findOrder(T.left, n) + findOrder(T.right, n) once again

Since we know the left subtree of L is null, the call findOrder(T.left, n) will return 0, So the above recursion now looks like:

1 + 0 + findOrder(T.right, n).

Again, we know the right subtree of L is also null, so the return statement becomes

1 + 0 + 0 = 1

Thus 1 is returned to the call findOrder(T.left, n) in the first iteration of the function.

Now Consider the call findOrder(T.right, n):

In this case, T.right is the subtree R, and it will be equal to the right-most leaf, which contains the value n. On this recursion, the conditional (T.data == n) will be entered, and return :

1 + findOrder(T.left, n);

Since we know the left subtree of R is null, the call findOrder(T.left, n) will return 0, thus the return statement becomes:

1 + 0 = 1;

Now 1 is returned to the call findOrder(T.right, n) in the first iteration of the function

The return statement returned by the first iteration of this function is now:

1 + 1 + 1 = 3

This is correct, since we were looking for the amount of elements <= the right most leaf, which is equal to the number of nodes in the tree.

2.) T.data > n

In this case, the value of n is the left-most leaf of the tree. After the first iteration, the program returns:

findOrder(T.left, n)

Consider the call to findOrder(T.left, n)

Essentially this cuts the size of the tree in half. Now we will recurse through sub-tree L to find the node which contains value n. On the first recursion through the left subtree, T.data will contain the value of n, and enter the conditional (T.data == n) where it returns:

return 1 + findOrder(T.left, n);

From the Inductive Hypothesis we know the value of L's left subtree is null, so the call findOrder(T.left, n) will return 0, thus making the return statement:

1 + 0 = 1

Now 1 is returned to the call findOrder(T.left, n) in the first iteration of the function, and the final value returned is also 1.