Problem Set 7, Part I

Problem 1: Working with stacks and queues

1-1)

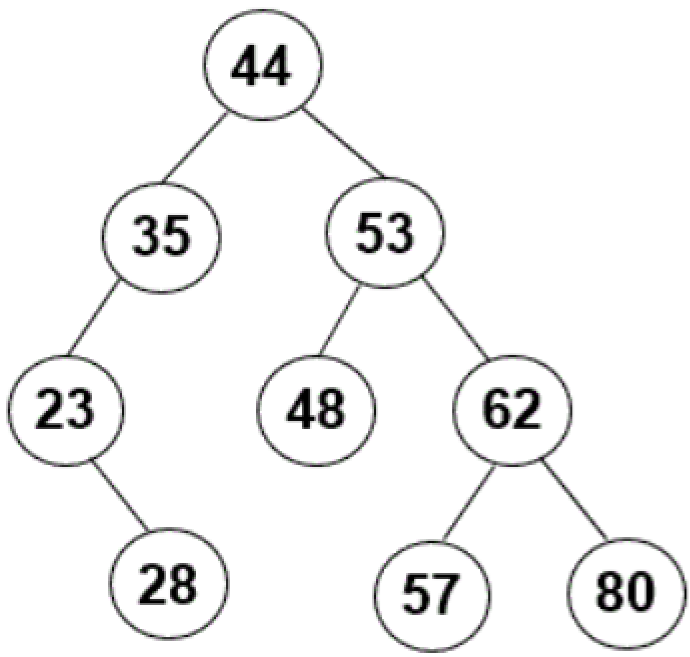
```
`public static void doubleAllStack(Stack<Object> stack, Object item)
{
        if(item==null){
            throw new NullPointerException();
        }
        Stack<Object> stack1 = new Stack<Object>();
        while(!stack.isEmpty()){
            Object top=stack.pop();
            if(item.equals(top)){
                stack1.push(top);
            }
            stack1.push(top);
        }
        while(!stack1.isEmpty()){
            stack.push(stack1.pop());
        }
    }
```

1-2)

```
   public static void doubleAllQueue(Queue<Object> stack, Object item)
{
        if(item==null){
            throw new NullPointerException();
        }
        Queue<Object> q= new Queue<Object>();
        while(!stack.isEmpty()){
            Object top=stack.remove();
            if(item.equals(top)){
                q.insert(top);
            }
            q.insert(top);
        }
        while(!q.isEmpty()){
            stack.insert(q.remove());
        }
    }
```

Problem 2: Using a queue to search a stack

```
boolean found=false;
//checks if item I is in the stack by removing all of the items in the
//stack and placing them in the queue Q
while(!S.isEmpty()){
	Object item = S.pop();
	//if the item is found, the boolean variable found is set to true
	if(I.equals(item)){
		found=true;
	}
	Q.insert(item);
}
//adds the items from the queue back into the stack
while(!Q.isEmpty()){
	S.push(Q.remove());
}

//the items are in backwards order so we add the items from the stack
//back into the queue
while(!S.isEmpty()){
	Q.insert(S.pop());
}

//finally, we add the items from the queue back into the stack and
//they are in the correct order
while(!Q.isEmpty()){
	S.push(Q.remove());
}

//returns the boolean variable found, indicating that item I was found
return found;
```



Problem 3: Binary tree basics

3-1) Height: 3

3-2) Leaf Nodes: 4
Interior Nodes: 5

3-3) 44 35 23 28 53 48 62 57 80

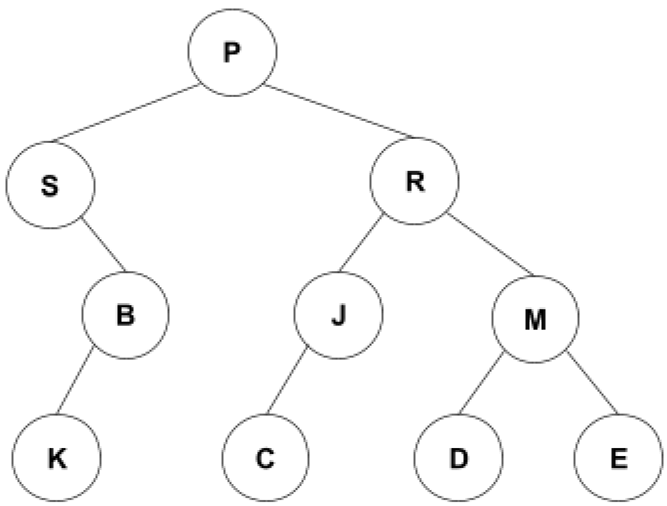
3-4) 28 23 35 48 57 80 62 53 44

3-5) 44 35 53 23 48 62 28 57 80

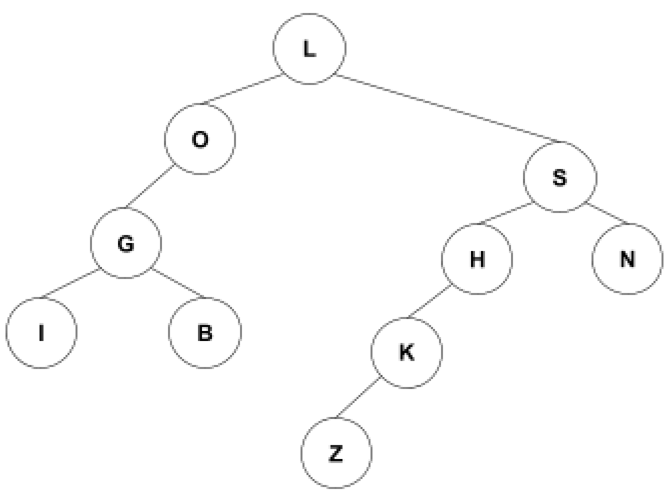
3-6) Yes, this is a search tree as for each node the left subtree is less than the node and the right subtree is greater than the node

3-7) Yes, this is a balanced search tree as for each node its subtree has either the same height or its height - 1 .

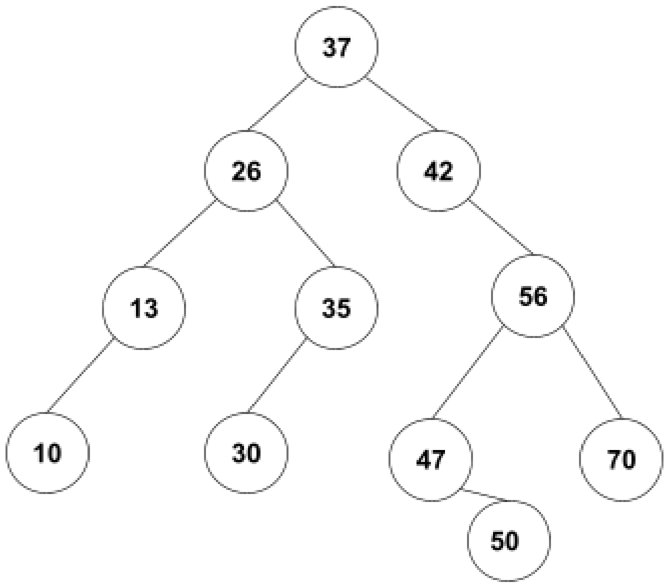
Problem 4: Tree traversal puzzles
4-1)



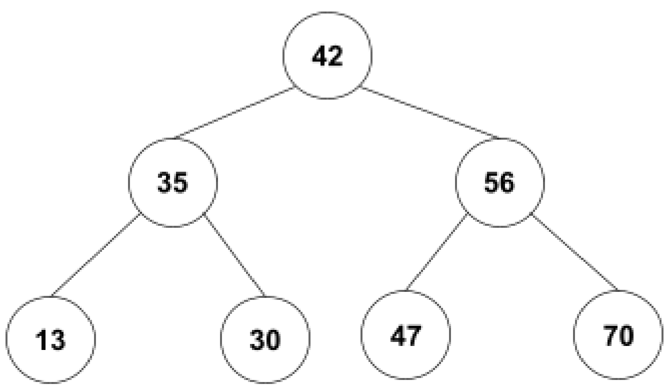
4-2)



Problem 5: Binary search trees
5-1)



5-2)



Problem 6: Counting keys below a threshold

6-1)

Best case: O(1) The best case occurs when the tree only consists of a root node, in this case the root key will only be compared to t and therefore no recursive calls will be made. There would only be a call to the numSmallerInTree method that would check if the root key is greater than or less than t. This does not traverse the tree therefore in the best case it would be O(1).

Worst case: O(n) In the worst case the entire tree has to be searched and it is unbalanced. If it is unbalanced this gives a worst case time complexity of O(n) to search it, as the entire tree has to be searched. The Tree contains n items and you must process all n of the nodes, performing O(1) operations on each, which gives O(n) time complexity.

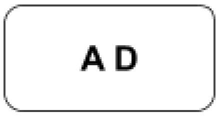
6-2)

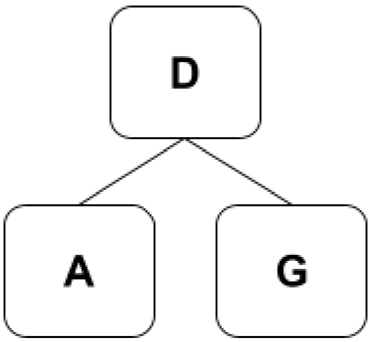
```
private static int numSmallerInTree(Node root, int t) {
      int count=0;
      if ( root.left != null && root.key < t) {
            count += numSmallerInTree(root.left, t);
        }
       if (root.right!= null && root.key< t) {
            count += numSmallerInTree(root.right, t);
        }
       if(root.left!=null && root.key>t){
            count += numSmallerInTree(root.left, t);
        }

       if (root.key < t) {
            return 1 + count;
       } else {
            return count;
       }
}
```

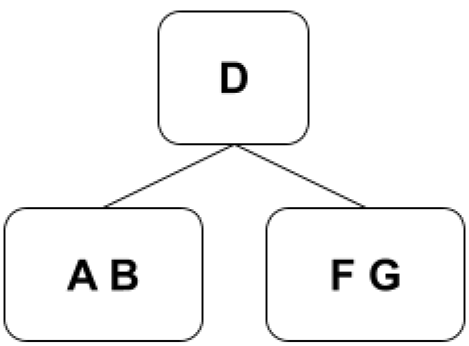
6-3) Best Case: O(1), if the tree only consists of a root node, it does not have to traverse the tree therefore it has a time efficiency of O(1)

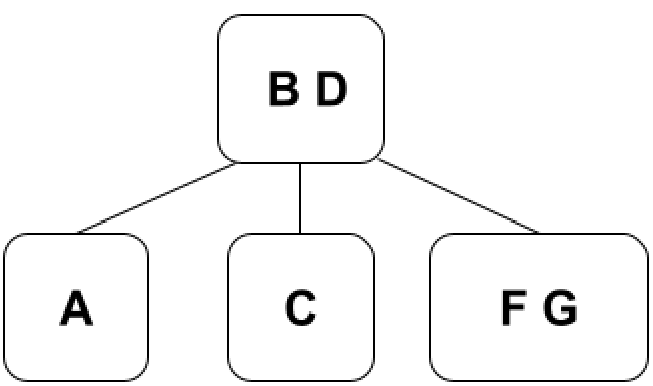
Worst Case: O(n), This method has an O(n) time efficiency as if every node was less than t it would search the whole tree giving an O(n) efficiency.

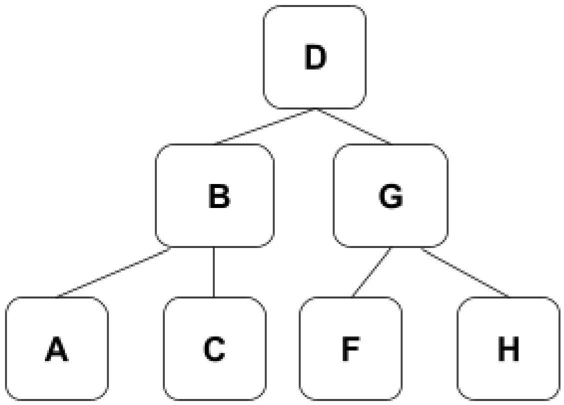
Problem 7: Balanced search trees

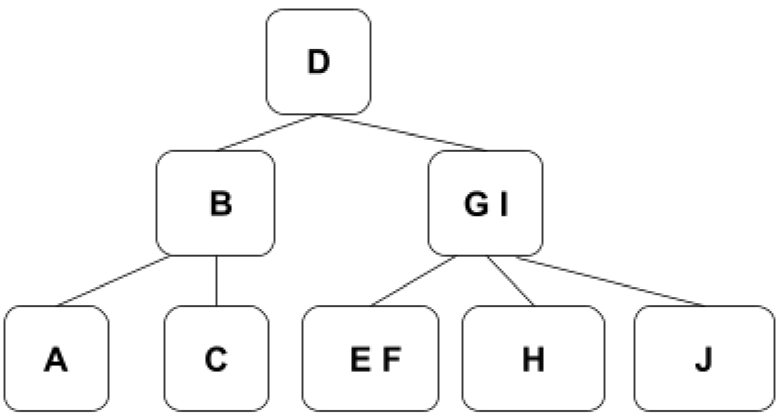









D
B
G
A
C
F
H


D
B
G I
A
C
E F
H
J