Problem Set 7, Part I

Problem 1: Working with stacks and queues

```
public static void remAllStack(Stack<Object> stack, Object item){

ListIterator<Object> trav = stack.iterator();
while(trav.hasNext()){
      Object itemAt =trav.next();
      if(itemAt.equals(item)){
            stack.pop():
      }
}

}
```

```
public static void remAllQueue(Queue<Object> queue, Object item){

ListIterator<Object> trav = queue.iterator();
while(x.hasNext()){
      Object itemAt = trav.next();
      if (itemAt.equals(item)){
            queue.remove();
      }
}

}
```

Problem 2: Using queues to implement a stack

Push(Object):

Add Object to Q1
Remove each Object from Q2
And add that to Q2
Assign each of the items in each queue to each other

Big O: This has a time complexity of O(n) since we have to traverse the entire length of the queue.

Pop():

If Q1 or Q2 is empty, then return null
Else remove Object from the front of Q1/Q2
Decrease size

Big O: Since we only have to look at the front of the queue, this has a O(1)

Peek():

If Q1 or Q2 is empty, then return null
Else return the front element of the ADT

Big O: Since we only have to look at the front of the queue, this has a O(1)

Problem 3: Binary tree basics

1) 3

2) 4 leaves and 5 interior nodes

3) 21, 18, 7, 25, 19, 27, 30, 26, 35

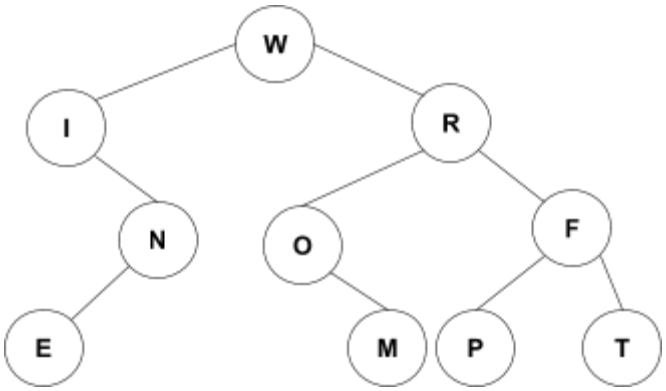
4) 19, 7, 25, 18, 26, 35, 30, 27, 21

5) 21, 18, 27, 7, 25, 30, 19, 26, 35

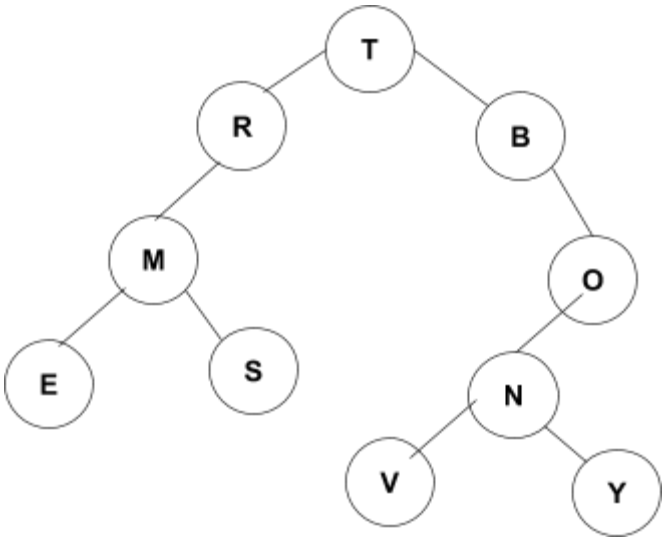
6)No, for this to be a Search Tree all elements to the left of the root need to be less than the root. However, 25 is more than 21. Therefore, this is not a Search Tree.

7) Yes, this is a Balanced Tree because every side of the tree differs by at most 1 node in depth

Problem 4: Tree traversal puzzles
4-1)

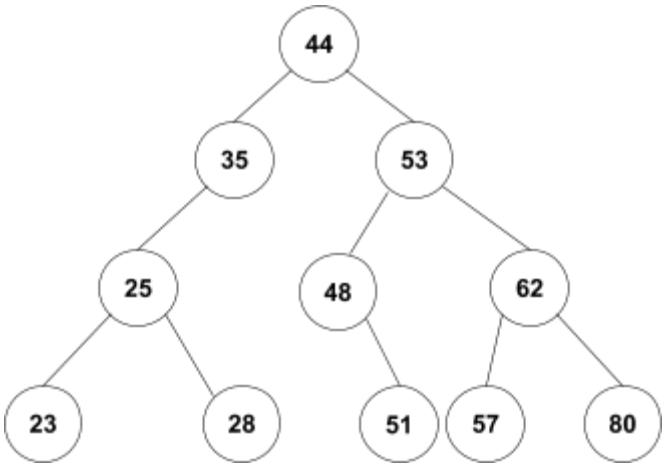


4-2)

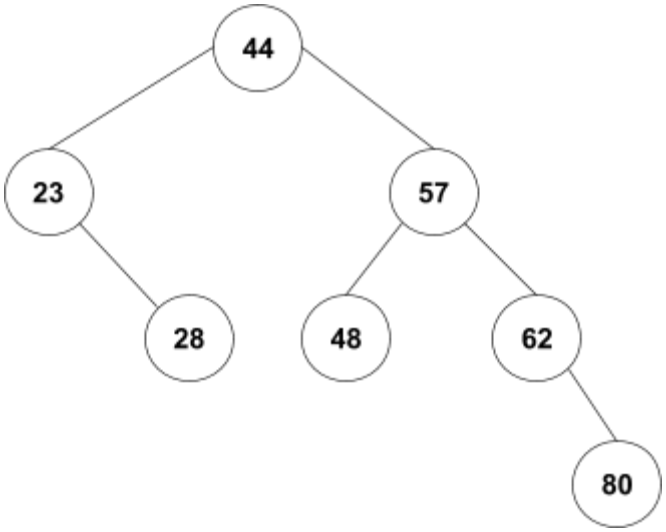


Problem 5: Binary search trees

5-1)



5-2)



Problem 6: Determining the depth of a node

6.1)
Best Case: The best case would be O(1) for when the input node is the root node of the tree.

Worst Case (Balanced & Unbalanced): Regardless of whether the tree is balanced or not, the run time for the worst case would be O(n) since you would have to search every node to find the depth of the one you are looking for (if it even is on the tree). Just because the tree is balanced does not mean you can rule out any nodes. Hence, why the expression is the same for balanced and unbalanced.

6.2)

```
private static int depthInTree(int key, Node root){
        if (key == root.key){
                return 0;
        }
        if (root.left != null && key < root.key){
                int leftDepth = depthInTree(key, root.left);
                if (leftDepth != -1){
                        return leftDepth + 1;
                }
        }
        if (root.right != null && key > root.key){
                int rightDepth = depthInTree(key, root.right);
                if (rightDepth != -1){
                        return rightDepth + 1;
                }
        }
        return  -1;
}
```

6.3)
Best Case: Again, the best case time complexity would be O(1) for when the input node is the root node of the tree.

Worst Case:

Balanced: The worst case time complexity for a balance tree would be O(logn) since we don't have to go through the whole tree to find the node we're looking for and its depth. We only search through log(n) nodes since in a balanced search tree, the height of the tree is log(n).

Unbalanced: The worst case time complexity would be O(n) since the whole tree could just be a big linked list where every node just points to one more node after it. Therefore, you'd need to traverse all n nodes to find the key you're looking for.

Problem 7: 2-3 Trees and B-trees

7-1)