Problem Set 7, Part I

Problem 1: Working with stacks and queues

1.1)

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

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


}
```

1.2)

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

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


}
```

Problem 2: Using queues to implement a stack

Assume K is the length of the ADT.

Push(item):

Time Complexity: Since we have to traverse through the whole length,
This should give us a time complexity of O(K).

// we enqueue (add) item to Q1
// denqueue (remove) each item of Q2
// and enqueue (add) that to Q2
// lastly, we assign each of Q's to each other

Pop():

Time Complexity: O(1) because we are only need access of the start/ front of the ADT. We never have to traverse through full length (K) of the ADT.

// if Q1 , Q2 empty
// return null
// else
// remove from start / front (access @ front) of Q1/Q2
// decrement size

Peek():

Time Complexity: O(1) because we are only peeking from start/ front. We never have to traverse through size of the ADT.

// Note: Peek is similar to Pop except, we dont have to remove !

// if Q1 , Q2 empty
// return null
// else
// return top/ front/start element of ADT

Problem 3: Binary tree basics

1) **Height:** 3

2) **Leaf Nodes (Nodes without Kids):** 4

   Interior Nodes (Nodes from root to leaf): 4

3) **PreORDER Traversal:** Root, Left , Right
   21,18,7,25,19,27,30,26,35

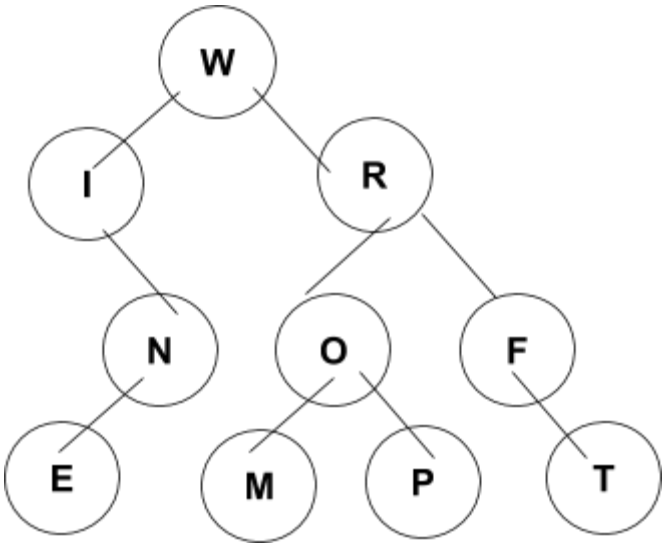
4) **PostOrder Traversal:** Left , Right, Root
   19,7,25,18,26,35,30,27,21

5) **Level Order Traversal:** Top to Bottom, Left to Right
   21, 18,27,7,25,30,19,26,35

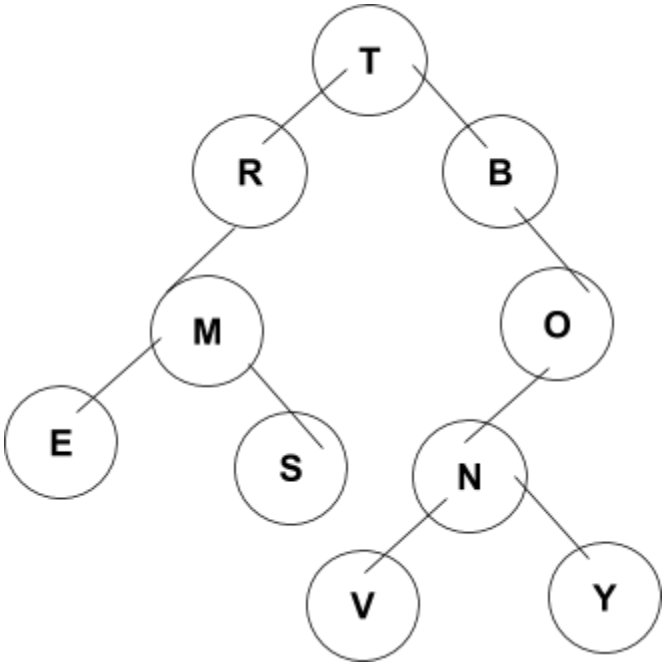
6) **Is this Tree a SEARCH Tree?**
   No, this is not a search tree because 25 is in the Left SubTree of root 21. Where 25 is larger than 21, hence the inOrder Traversal would NOT be in ascending and thus, this is not a Search tree.
7) **Is this a BALANCED Tree?**
   Yes, this is a balanced tree because the nodes subtree differ by atmost 1.

Problem 4: Tree traversal puzzles

4-1)

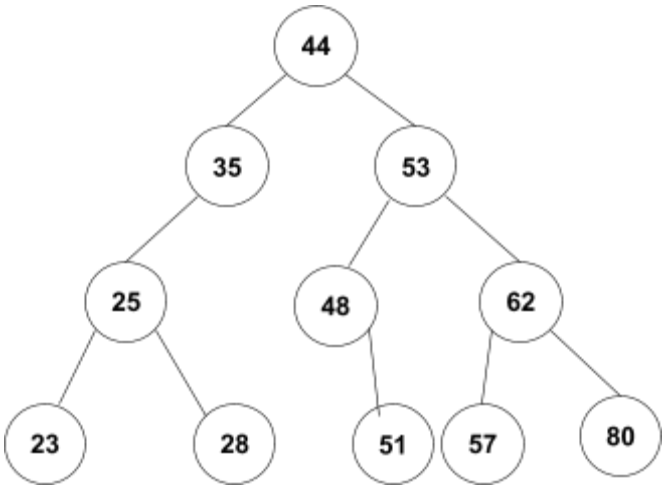


4-2)

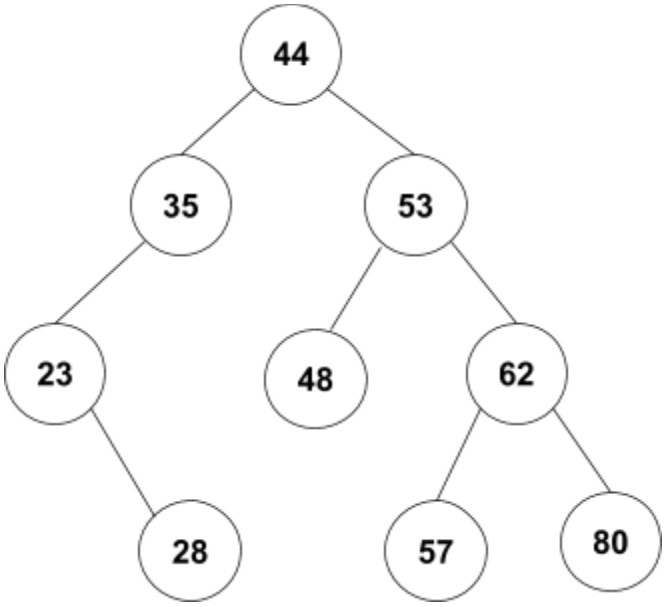


Problem 5: Binary search trees

5-1)



5-2)



Problem 6: Determining the depth of a node

6.1)

Binary Tree-

Best Case: The time complexity in the best case would be **O(1)**, because we find the key in the root.

Worse Case:

Balanced: The time complexity in the worse case for a balanced tree would be O(height) = **O(logn)** because we search through half of tree due to the recursive implemntation. The nodes subtrees have the same height / or have heights that differ by atmost 1.

Not Balanced: The time complexity in the best case for a not balanced tree would be O(height) = **O(n)** because the tree is now equivalent to a LinkedList and height = n-1.

6.2)

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

6.3)

Binary Search Tree- revised algorithm

Best Case: The time complexity in the best case would be **O(1)**, because we find the key in the root.

Worse Case:

Balanced: The time complexity in the worse case for a balanced tree would be O(height) = **O(logn)** because we search through half of tree due to the recursive implemntation. The nodes subtrees have the same height / or have heights that differ by atmost 1.

Not Balanced: The time complexity in the best case for a not balanced tree would be O(height) = **O(n)** because the tree is now equivalent to a LinkedList and depth = n-1.

Problem 7: 2-3 Trees and B-trees

7-1)