Problem Set 7, Part I

Problem 1: Working with stacks and queues

1.

```
public static void remAllStack(Stack<Object> stack, Object item) {
    Stack<Object> temp = new LLStack<Object>();
    while (!stack.isEmpty()) {
        if (!stack.peek().equals(item)) {
            temp.push(stack.peek());
        }
        stack.pop();
    }
    while (!temp.isEmpty()) {
        stack.push(temp.peek());
        temp.pop();
    }
}
```

2.

```
public static void remAllQueue(Queue<Object> queue, Object item) {
    Queue<Object> temp = new LLQueue<Object>();
    while (!queue.isEmpty()) {
        if (!queue.peek().equals(item)) {
            temp.insert(queue.peek());
        }
        queue.remove();
    }
    while (!temp.isEmpty()) {
        queue.insert(temp.peek());
        temp.remove();
    }
}
```

Problem 2: Using queues to implement a stack

```
/*
 * TwoQueuesStack.java
 *
 * Computer Science 112, Boston University
 * Andy Vo
 */

 /*
 * A generic class that implements the Stack interface using two
queues.
 * Note: Stack is "last in, first out; Queue is "first in, first out".
 */
public class TwoQueuesStack<T> implements Stack<T> {
    private LLQueue<T> q1 = new LLQueue<T>(); //queue 1
    private LLQueue<T> q2 = new LLQueue<T>(); //queue 2
    private int top;                        //index of the top item

    /**
     * Constructor
     */
    public TwoQueuesStack() {
        top = -1;
    }

    /**
     * push - adds the specified item to the top of the stack.
     * Always returns true, because the linked list is never full.
     *
     * Time complexity: O(n). We are inserting each new element to
     * q2 to keep it as the top element. We remove all elements in
     * q1 one at a time if there are any and insert them into q2
     * so that the new element is always positioned at the front of
     * q2. We then swap q1 with q2. O(n) because we iterate through
     * q1.
     *
     * Note: push can be O(1) in the event where the new element is
     * always added to the rear of q1 and kept as the top stack
     * element
     * if we change "top" to a reference to the top rather than an
     * index.
     */
```

```
public boolean push(T item) {
    if (item == null) {
        throw new IllegalArgumentException();
    }
    q2.insert(item);
    top++;
    while (!q1.isEmpty()) {
        q2.insert(q1.remove());
    }
    LLQueue<T> temp = q1;
    q1 = q2;
    q2 = temp;
    return true;
}

/**
 * pop - removes the item at the top of the stack and returns a
 * reference to the removed object.  Returns null if the stack is
 * empty.
 *
 * Time complexity: O(1). Queues and stacks share a common trait
 * in that both remove from the top of the data structure.
 * Since queues and stacks always have a reference to the
 * top-most object, it will remove it in constant time.
 *
 * Note: pop can be O(n) if push is O(1). Essentially, we would
 * need to move all items up to the top item in q1 to q2, remove
 * the top item, and then swap q1 with q2.
 */
public T pop() {
    if (top == -1) {
        return null;
    }
    top--;
    return q1.remove();
}

/**
 * peek - returns a reference to the item at the top of the stack
 * without removing it. Returns null if the stack is empty.
 *
 * Time complexity: O(1). Using LLQueue, peek has a reference to
 * the front item in constant time. Thus, it is O(1) time. We are
```

```
     * simply returning the reference to the front.
     */
    public T peek() {
        if (top == -1) {
            return null;
        }
        return q1.peek();
    }
}
```

Problem 3: Binary tree basics

1. The height of the tree is 3.

2. There are four leaf nodes and five interior nodes.

3. Preorder traversal: 21 18 7 25 19 27 30 26 35

4. Postorder traversal: 7 19 25 18 26 35 30 27 21

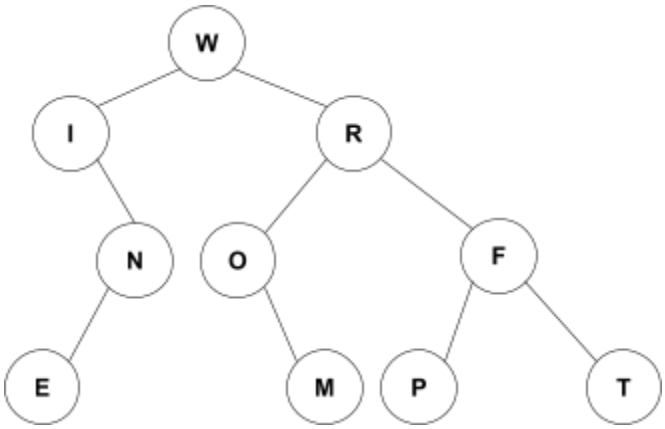
5. Level-order traversal: 21 18 27 7 25 30 19 26 35

6. The tree is not a search tree. In the left subtree, although the value 25 is greater than 18 (so that it belongs on the right subtree of 18), 25 is greater than 21. All values in the left subtree of 21 should be less than 21.

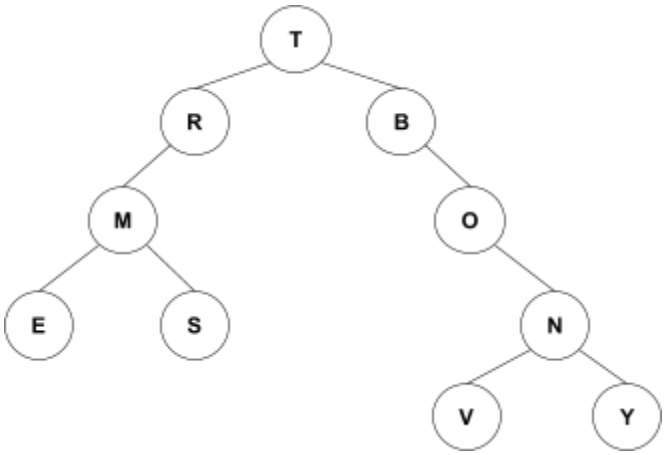
7. To be a balanced tree, two conditions must be met. One, a tree's left and right subtrees must differ in height by at most one. This condition is satisfied because the left subtree's height is 2 (excluding the root node and counting 18 as the left subtree's root node) and the right subtree's height is 2. Secondly, because of the recursive nature of a tree, the left and right "sub"-subtrees of any subtree must also be balanced. On the right subtree of the root node, "sub" root 27's right subtree has a height of 2 and a left subtree of height 0. The difference is greater than one, and thus, this tree is not balanced.

Problem 4: Tree traversal puzzles

4-1)

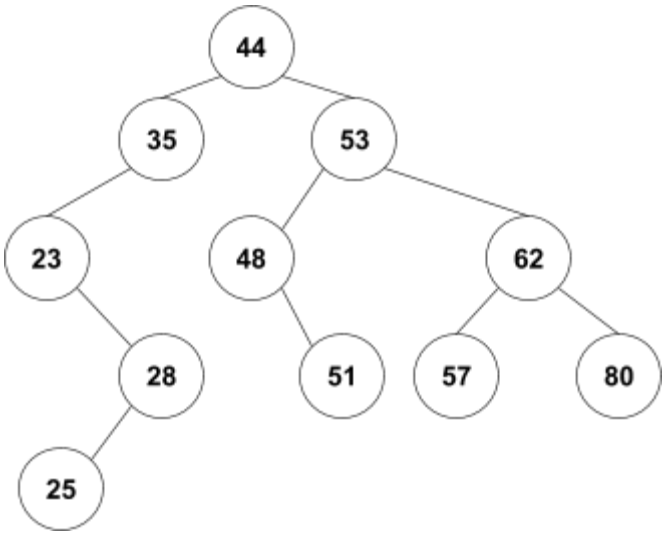


4-2)

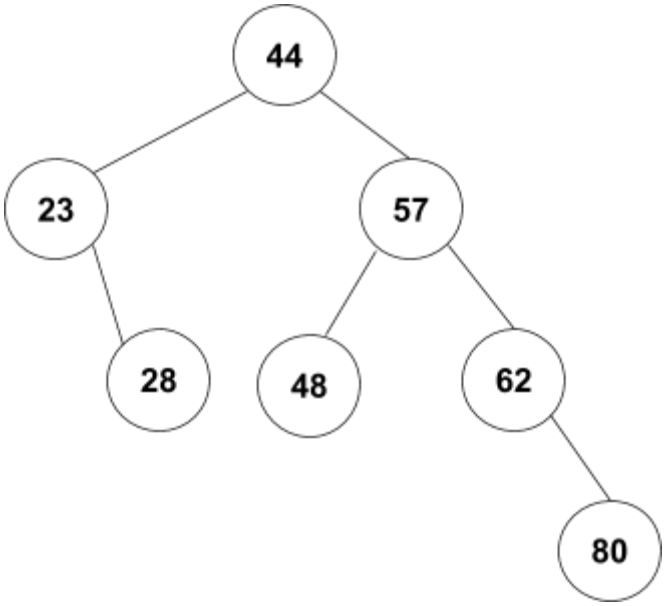


Problem 5: Binary search trees

5-1)



5-2)



Problem 6: Determining the depth of a node

1. The time complexity of this algorithm in its best case would simply be O(1). This would happen if the depth of the node is 0 where the key is the root node. The algorithm makes a comparison in the first if statement and does not need to go through any recursive calls. When the tree is balanced, the worst case is O(n) because in this algorithm, the recursive algorithm will continuously be called as long as the left and/or right child is not null. The same is said for unbalanced trees: O(n) time complexity. Essentially, the algorithm will search for the key throughout the entire tree regardless of it being balanced or unbalanced.

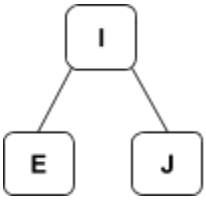
2.

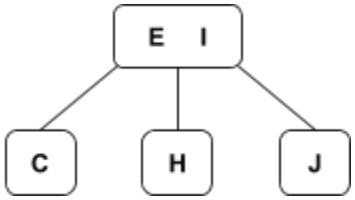
```
private static int depthInTree(int key, Node root) {
    if (key == root.key) {
        return 0;
    }

    if (key < root.key && root.left != null) {
        int depthInLeft = depthInTree(key, root.left);
        if (depthInLeft != -1) {
            return depthInLeft + 1;
        }
    }

    if (key > root.key && root.right != null) {
        int depthInRight = depthInTree(key, root.right);
        if (depthInRight != -1) {
            return depthInRight + 1;
        }
    }

    return -1;
}
```

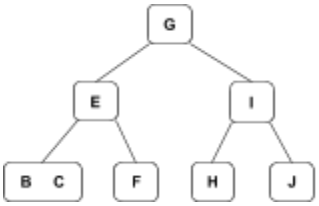
3. With this new algorithm, the best case remains the same at O(1), where the key is in the root node. For the worst case, when the tree is balanced, the time complexity is O(log n) because we divide the searches by half of possible cases for each depth given that the tree is a balanced binary search tree with only a left and right subtree. For an unbalanced tree, the worst case time complexity is O(n), where n can represent the height of the tree. For example, consider a tree with only right subtrees and the key is the leaf node.

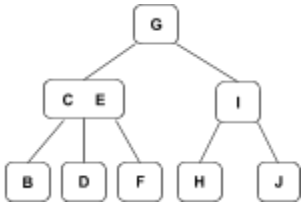
Problem 7: 2-3 Trees and B-trees

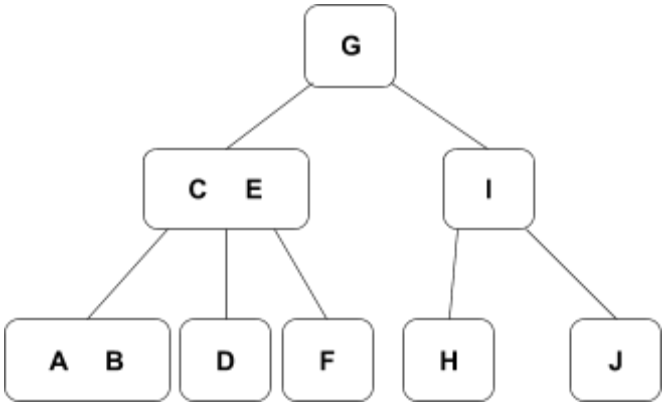
7-1)











7-2)