# CS310 Data Structures

## Midterm 2

**Total Points: 100** **Number of Questions: 5**

**Name: Stephen Giang**

**REDID: 823184070**

***Please read the questions thoroughly before answering.***

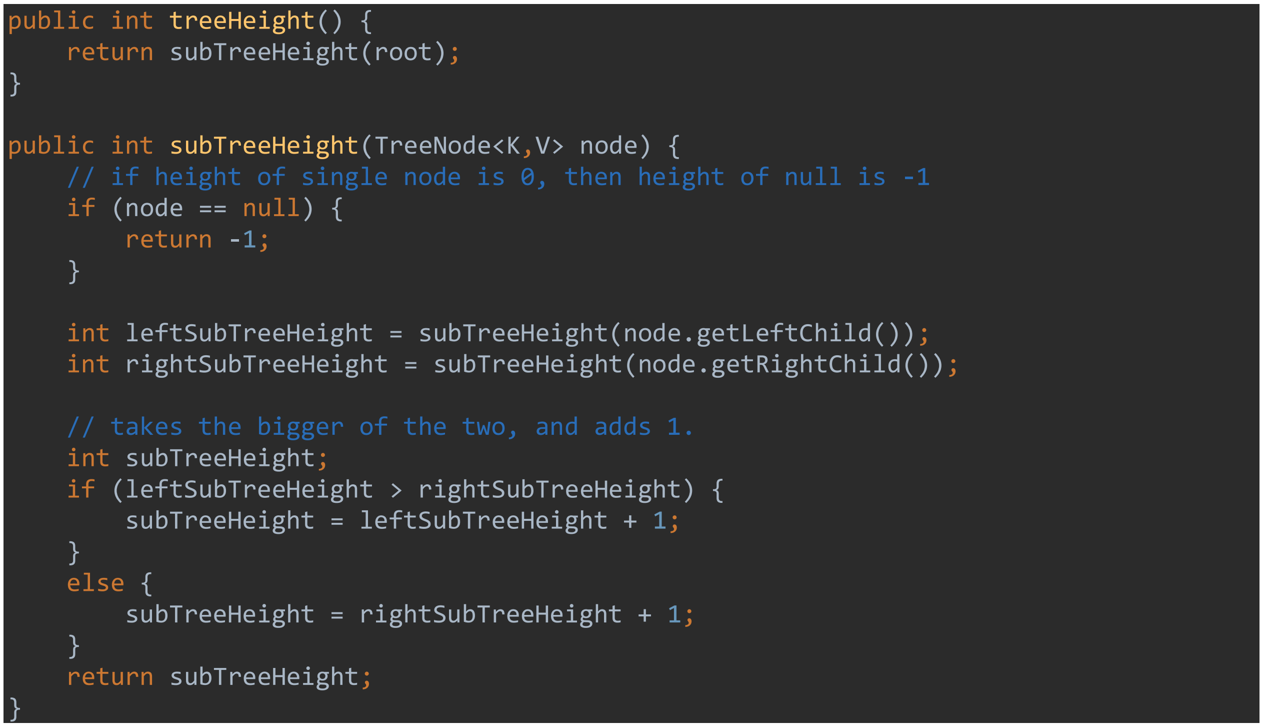
***Submit the exam as a PDF (written & scanned / filled in word and exported)***

1. (20 points total)
   The following code for a binary search tree is given:

```
public class TreeNode<K extends Comparable<K>,V>{
     private K key;
     private V value;
     private TreeNode<K,V> leftChild;
     private TreeNode<K,V> rightChild;
}
public class BinarySearchTree<K extends Comparable<K>,V>{
     private TreeNode<K,V> root;
     public void insert(K key, V value) { ... }
     public void delete(K key) { ... }
     public SomeType find(int key) { ... }
}
```

   Assume that the method bodies are complete and work as intended.
   Write **JAVA CODE** for the following:
   (a)(8 Points) Add a method **public int treeHeight()** to the BinarySearchTree class that computes the height of the tree.
   (b)(8 Points) Add method **public void deleteMax()** to the BinarySearchTree class that deletes the maximum element in the tree (or does nothing if the tree has no elements). You may want a helper method.
   (c)(4 Points) For your answers in part (a) and part (b), give a worst-case big-Oh running time in terms of n, where n is the number of nodes in the tree. Do not assume the tree is balanced.

**<<SPACE FOR WRITING CODE>>**



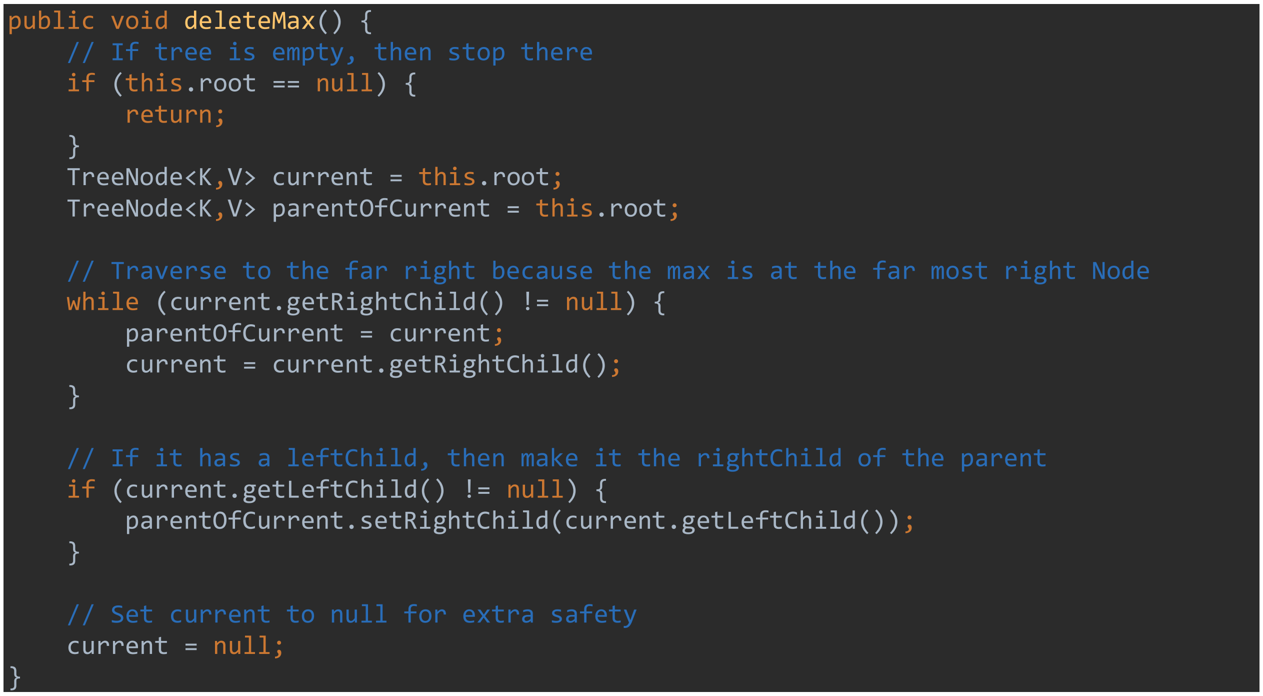
```java
public int treeHeight() {
    return subTreeHeight(root);
}

public int subTreeHeight(TreeNode<K,V> node) {
    // if height of single node is 0, then height of null is -1
    if (node == null) {
        return -1;
    }

    int leftSubTreeHeight = subTreeHeight(node.getLeftChild());
    int rightSubTreeHeight = subTreeHeight(node.getRightChild());

    // takes the bigger of the two, and adds 1.
    int subTreeHeight;
    if (leftSubTreeHeight > rightSubTreeHeight) {
        subTreeHeight = leftSubTreeHeight + 1;
    }
    else {
        subTreeHeight = rightSubTreeHeight + 1;
    }
    return subTreeHeight;
}
```

**Worst-Case Running Time – O(N)**



```java
public void deleteMax() {
    // If tree is empty, then stop there
    if (this.root == null) {
        return;
    }
    TreeNode<K,V> current = this.root;
    TreeNode<K,V> parentOfCurrent = this.root;

    // Traverse to the far right because the max is at the far most right Node
    while (current.getRightChild() != null) {
        parentOfCurrent = current;
        current = current.getRightChild();
    }

    // If it has a leftChild, then make it the rightChild of the parent
    if (current.getLeftChild() != null) {
        parentOfCurrent.setRightChild(current.getLeftChild());
    }

    // Set current to null for extra safety
    current = null;
}
```

**Worst-Case Running Time – O(N)**

2. (20 points total)
   a) (4 pts) What is the minimum and maximum number of nodes in a balanced AVL tree of height 5? (Hint: the height of a tree consisting of a single node is 0)
   Give an exact number (with no exponents) for your answers
   Minimum = **20**

   Maximum = **63**

   b) (4 pts) What is the minimum and maximum number of nodes in a full tree of height 6?
   Give an exact number (with no exponents) for your answers – not a formula.

   Minimum = **127**

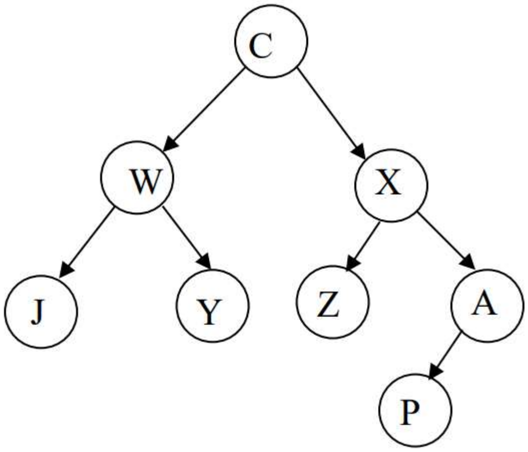
   Maximum = **127**

   c) (4 pts) What is the minimum and maximum number of non-leaf nodes in a complete tree of height 5?
   Give an exact number (with no exponents) for your answers.

   Minimum = **31**

   Maximum = **31**

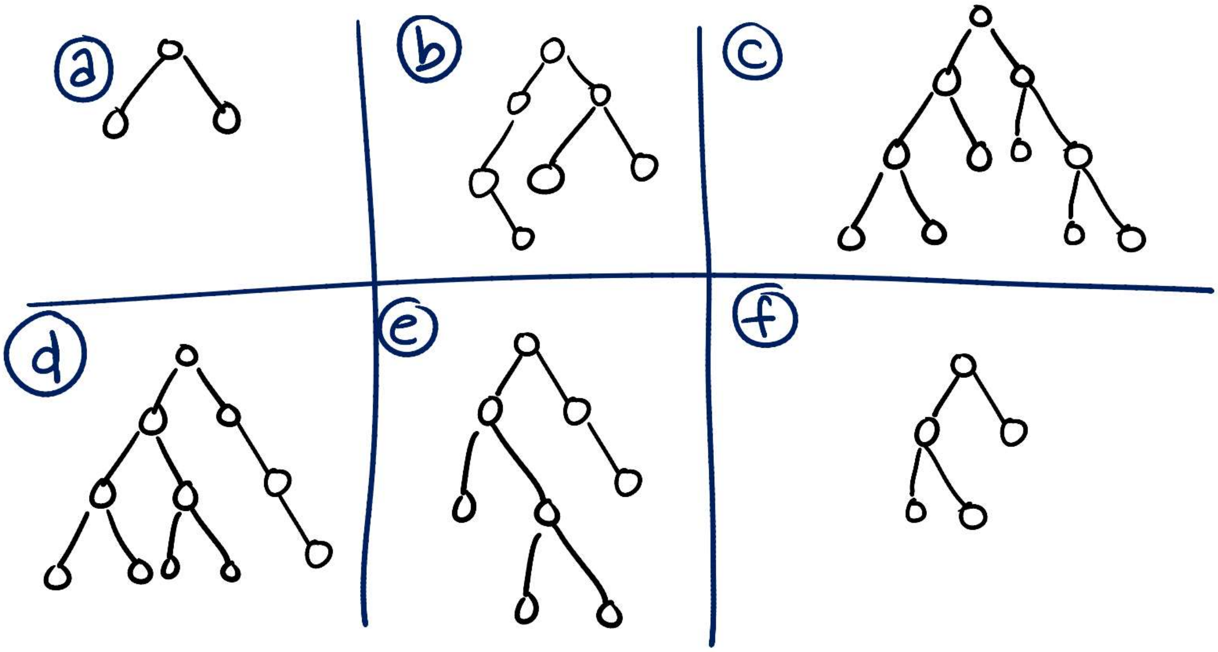
   d) (1 pt) What is the depth of node **P** in the tree shown below: **3**



   e) (1 pt) Give a **Pre-Order** traversal of the tree shown above in (d):

   **C, W, J, Y, X, Z, A, P**

f) (6 pts) Given the following six trees a through f:



List the letters of all of the trees that have the following properties: (Note: It is possible that none of the trees above have the given property, it is also possible that some trees have more than one of the following properties.)

**Full:** **a**

**Complete:** **a, f**

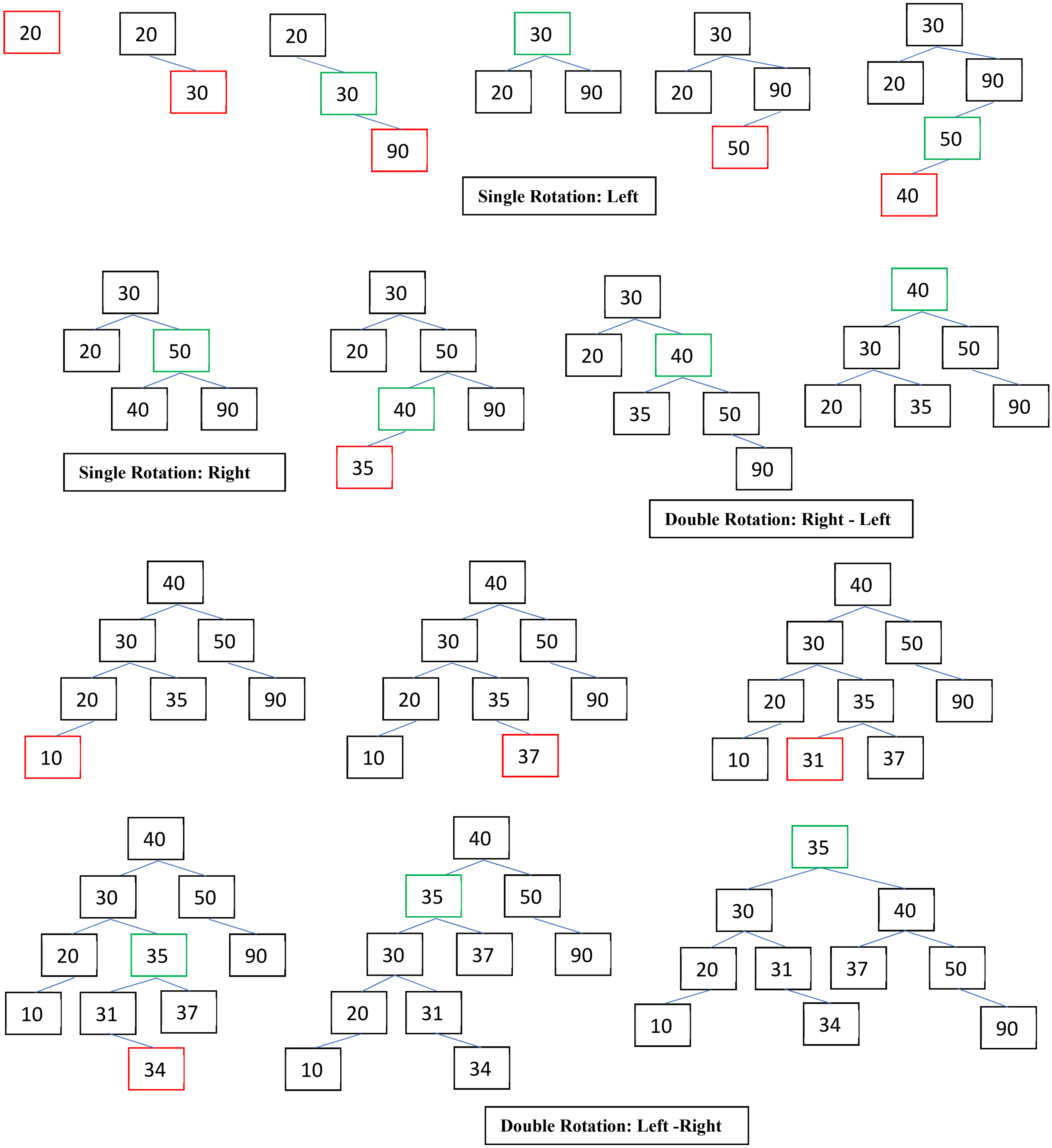
**AVL balanced:** **a, c, e, f**

3. a) (11 points) AVL Trees Draw the AVL tree that results from inserting the keys:

**20, 30, 90, 50, 40, 35, 10, 37, 31, 34**

in that order into an initially empty AVL tree.

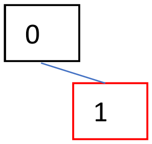
Draw all intermediate trees and label all the rotations that happen.



3 b) (9 Points) Indicate for each of the following statements if it is true or false. You must justify your answers to get credit.

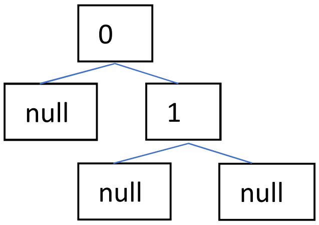
(i) The subtree of the root of a red-black tree is always itself a red-black tree. (Here, the definition of red-black tree is as I have given in class and as described in the textbook.)

**False, the subtree of the root of a red-black tree can have a red root, thus not making it a red-black tree. Ex. [0,1]. [1] is a red root, so it can not be a red-black tree.**



(ii) The sibling of a null child reference in a red-black tree is either another null child reference or a red node.

**True, if a black node was a sibling of a null child, then it would break the rule of all paths to the leaves have the same number of black nodes. Ex: [0, 1]. From the left it has 2 blacks, and the right has 3 blacks. So the sibling must be red or a null child for it to be a red-black tree.**

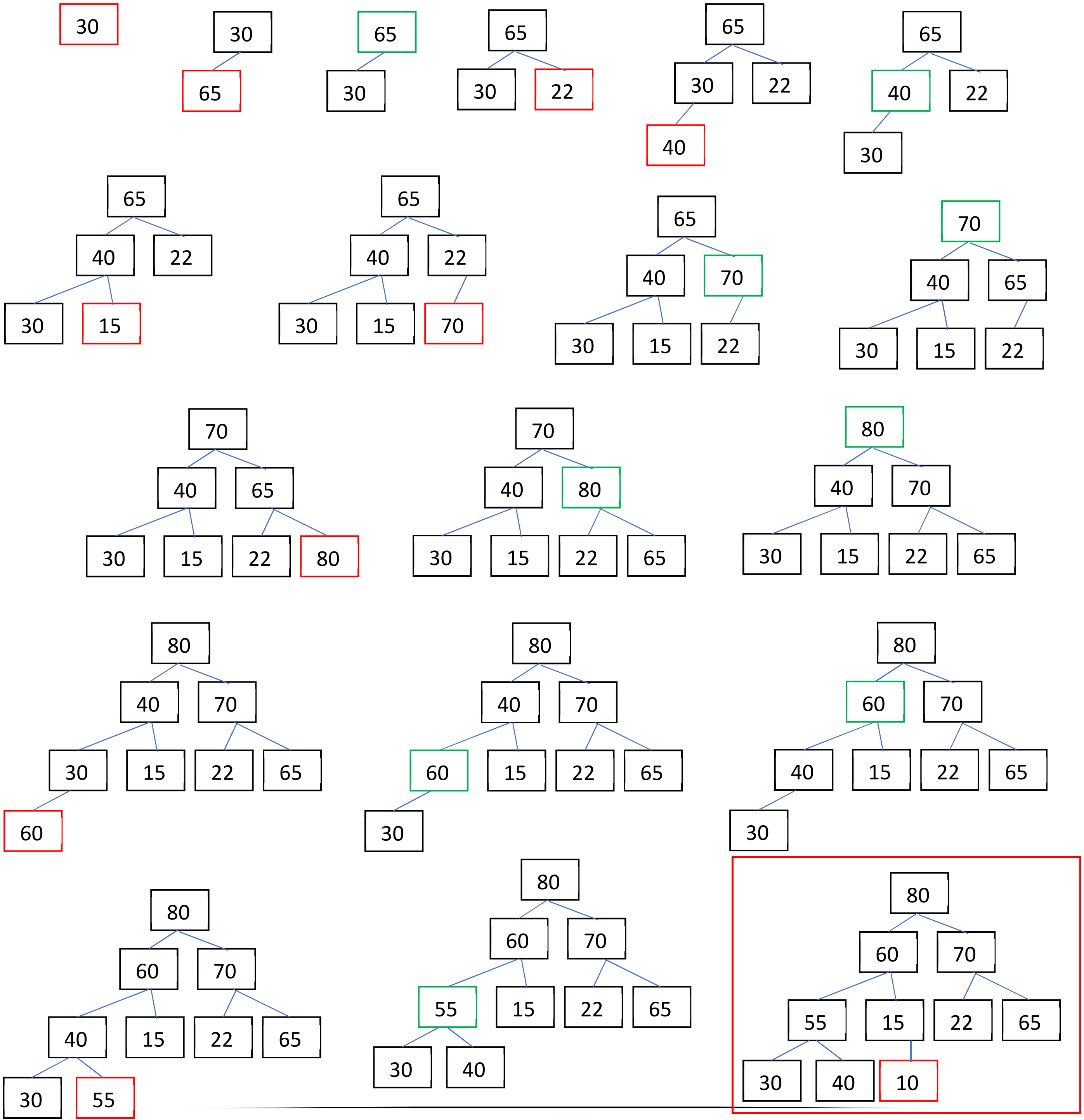


(iii) The worst case time complexity of the insert operation into an AVL tree is O(log n), where n is the number of nodes in the tree.

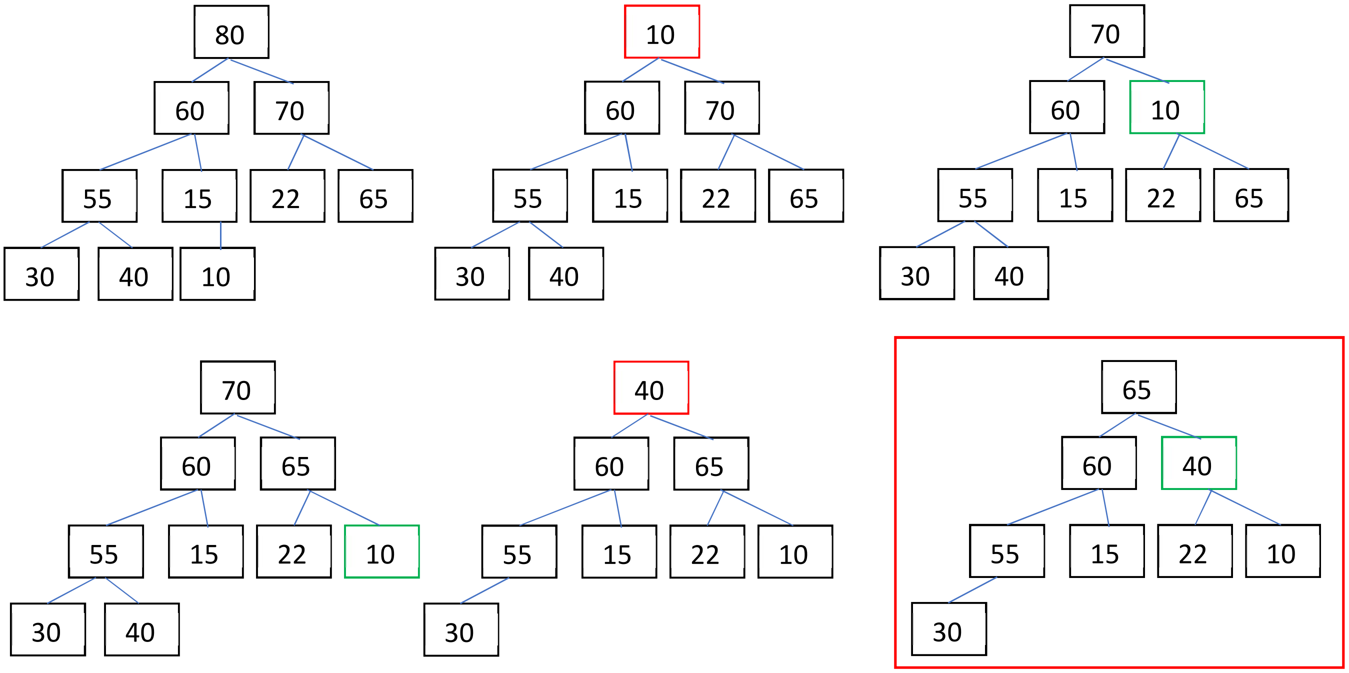
**True, all operations on AVL trees take worse case: O(logN). The Worst Case traverses through the tree which takes O(logN).**

4. (20 Points) Given the following integer elements:
30, 65, 22, 40, 15, 70, 80, 60, 55, 10
For a and b Show all intermediate steps for full credit. You do not need to show the array representation of the heap.

    a. Draw the tree representation of the heap that results when all of the above elements are added (in the given order) to an initially empty maximum binary heap. Circle the final tree that results from performing the additions.



b. After adding all the elements, perform 2 removes on the heap. Circle the tree that results after the two elements are removed.



5. (20 Points) This problem implements a minimum priority queue using a queue. The queue is already implemented with a linked list in the LinkedListQueue class. Elements are added to the priority queue by being put at the end of the queue. The add method of the queue-based priority queue is implemented below. Finish the implementation of the class IntQueuePQ, a priority queue implemented using a queue, by **writing the remove method (Write JAVA CODE)**
**Method description:**
   i. The remove method finds, removes, and returns the minimum element from the priority queue.
   ii. If the priority queue contains no elements and the remove method is called an **IllegalStateException** is thrown.

   In order to write remove you may call any of the methods declared by the **IntQueue interface** on the queue instance variable found in **IntQueuePQ**. However, **you may not use any additional data structures or abstract data types** other than queue declared in the IntQueuePQ class in order to solve this problem.

```
public interface IntPriorityQueue {

      public void add(int value);

      public int remove();

}

public interface IntQueue {

      public boolean isEmpty();

      public void enqueue(int i);

      public int dequeue();

      public int size();

}

public class IntQueuePQ implements IntPriorityQueue {

      private IntQueue queue;

      public IntQueuePQ() {

            queue = new LinkedListQueue();

      }

      public void add(int value) {

            queue.enqueue(value);

      }

       // THE REMOVE METHOD WOULD GO HERE

}
```

**<<WRITE CODE HERE>>**

```
public int remove () {
    // If empty, throw IllegalStateException
    try{
        if (queue.size() == 0) {
            throw new IllegalStateException();
        }
    }catch (IllegalStateException e) {
        System.out.println("Queue is Empty");
        return 0;
    }

    // Used to initialize the min as the top of the queue
    boolean minInitialized = false;
    // Temporary initializer
    int min = 0;
    // Gets the size of the queue before adding and deleting
    int sizeOfQueue = queue.size();

    // Traverse through entire queue to find min
    for (int i = 0; i < sizeOfQueue; i++) {
        int dequeueValue = queue.dequeue();

        // Initializes min to the top of the queue
        if (minInitialized == false) {
            min = dequeueValue;
            minInitialized = true;
        }

        if (dequeueValue < min) {
            min = dequeueValue;
        }
        queue.enqueue(dequeueValue);
    }

    // Removes min
    for (int i = 0; i < sizeOfQueue; i++) {
        int dequeueValue = queue.dequeue();
        if (dequeueValue != min) {
            queue.enqueue(dequeueValue);
        }
    }
    return min;
}
```