CSC 151 Assignment #9

1. **Honor Code**

A. For individual assignments: Jane Doe and John Doe will be replaced by your full name(s)
I affirm that I have carried out my academic endeavors with full academic honesty.
[Signed, Manav Bilakhia]
B. Resources/References

Geeksforgeeks for syntax

2. **Java files and outputs**

A. Java files
Class: TreeGenericArrayList.Java

```
/*
 * I affirm that I have carried out my academic endeavors with full academic honesty.
 * Manav Bilakhia MB
 */
package assignment;
import java.util.*;
/**
 * This class generates a binary complete tree from an ArrayList. The tree can
 * be constructed by preserving the heap property if selected. The tree can be
 * traversed by inOrder, preOrder or postOrder methods The tree can be formatted
 * as indented.
 *
 * @author Zeynep Orhan
 *
 * @param <T> type of the data to be stored in the tree nodes. <T> should have a
 * compareTo method
 */
public class TreeGenericArrayList<T extends Comparable<? super T>> {
    private Node root; // Root Node
    // Private inner class Node for a binary tree
    private class Node {
        private T data;
        private Node left, right;
        private Node(T data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }
    public T getRootData() {
        return root.data;
    }
    public void setRootData(T data) {
        root.data = data;
    }
    // getRoot, setRoot, getRootData and setRootData methods
    public Node getRoot() {
        return root;
    }

    public void setRoot(Node root) {
        this.root = root;
    }

    /**
```

```
     * heapifyArray: This method reorders the items in an ArrayList arr to preserve the
heap property
     *
     * @param arr ArrayList<T> that keeps the items to be stored
     */
    public void heapifyArray(ArrayList<T> arr)
    {
        int swapCount = 0;
        for(int i = arr.size()-1; i>0;i--)
        {
//            System.out.println("index"+ i);
            if(i % 2 == 0)
            {
                int indexToGet = (i-2)/2;

                if(arr.get(i).compareTo(arr.get(indexToGet)) >= 0)
                {
                    T swap = arr.get(i);
                    arr.set(i, arr.get(indexToGet));
                    arr.set(indexToGet,swap);
                    swapCount++;
                }
            }
            else
            {
                int indexToGet = (i-1)/2;

                if(arr.get(i).compareTo(arr.get(indexToGet)) > 0)
                {
                    T swap = arr.get(i);
                    arr.set(i, arr.get(indexToGet));
                    arr.set(indexToGet,swap);
                    swapCount++;
                }
            }

        }
        if (swapCount !=0)
        {
            heapifyArray(arr);
            swapCount--;
        }
    }
    /**
     * insertLevelOrderHeap: Inserts the items in the arr and creates a tree whose root
is root
     * If heap is true call heapifyArray first
     *
     * @param arr ArrayList<T> that keeps the items to be stored
     * @param root root of the tree of type Node
     * @param i the int index of the ArrayList item to be inserted
     * @param heap boolean the ArrayList will be heapified if true, otherwise it will
be inserted as is
     * @return root of the tree
     */
    public Node insertLevelOrderHeap(ArrayList<T> arr, Node root, int i, boolean heap)
    {
        if (heap==true)
        {
            heapifyArray(arr);
            return insertLevelOrder(arr,root,i);
        }
        else
```

```
            return insertLevelOrder(arr,root,i);
    }
    /**
     * insertLevelOrder: Inserts the items in the arr and creates a tree whose root is
root
     *
     * @param arr ArrayList<T> that keeps the items to be stored
     * @param root root of the tree of type Node
     * @param i the int index of the ArrayList item to be inserted
     * @return root of the tree
     */
    public Node insertLevelOrder(ArrayList<T> arr, Node root, int i)
    {
        if (i < arr.size()) {
            Node temp = new Node(arr.get(i));
            root = temp;
            root.left = insertLevelOrder(arr, root.left,
                    2 * i + 1);
            root.right = insertLevelOrder(arr, root.right,
                    2 * i + 2);
        }
        return root;
    }
    /**
     * inOrder: String representation of the inorder traversal
     *
     * @param root root of the tree
     * @return a String
     */
    public String inOrder(Node root) {
        String nodeToReturn = "";
        if (root != null) {
            nodeToReturn = nodeToReturn +
                    this.inOrder(root.left);
            nodeToReturn = nodeToReturn + root.data + " ";
            nodeToReturn = nodeToReturn +
                    this.inOrder(root.right);
        }
        return nodeToReturn;
    }

    /**
     * preOrder: String representation of the preorder traversal
     *
     * @param root root of the tree
     * @return a String
     */
    public String preOrder(Node root)
    {
        String nodeToReturn = "";
        if (root != null) {
            nodeToReturn = nodeToReturn + root.data + " ";
            nodeToReturn = nodeToReturn +
                    this.preOrder(root.left);
            nodeToReturn = nodeToReturn +
                    this.preOrder(root.right);
        }
        return nodeToReturn;
    }
    /**
     * postOrder: String representation of the postorder traversal
     *
     * @param root root of the tree
```

```
     * @return a String
     */
    public String postOrder(Node root)
    {
        String nodeToReturn = "";
        if (root != null) {
            nodeToReturn = nodeToReturn + this.postOrder(root.left);
            nodeToReturn = nodeToReturn + this.postOrder(root.right);
            nodeToReturn = nodeToReturn + root.data + " ";
        }
        return nodeToReturn;
    }
    /**
     * height: Calculates the height of the tree
     * @param root of the tree
     * @return height of the tree as int
     */
    public int height(Node root)
    {
            if (root == null)
                return 0;
            else
            {
                int lDepth = height(root.left);
                int rDepth = height(root.right);
                if (lDepth > rDepth)
                    return (lDepth + 1);
                else
                    return (rDepth + 1);
            }
    }
    /**
     * display: Display the tree in a formatted way (vertical)
     * If the items are { 6, 0, 1, 3, 6, 5, 4, 7, 9, 2, 12, 15, 28, 32, 48 }
     * The tree will be displayed as
     * 6
     * |_0
     * |_|_3
     * |_|_|_7
     * |_|_|_9
     * |_|_6
     * |_|_|_2
     * |_|_|_12
     * |_1
     * |_|_5
     * |_|_|_15
     * |_|_|_28
     * |_|_4
     * |_|_|_32
     * |_|_|_48
     *
     * @param root root of the tree
     * @return
     */
    public String display(Node root)
    {
        return helpDisplay(root,0);
    }
    private String helpDisplay(Node root, int depth) {
        if (root != null) {
            if (depth == 0) {
                return root.data + helpDisplay(root.left, depth + 1) +
helpDisplay(root.right, depth + 1);
```

```
            } else {
                String extraStuff = "";
                for (int i = 0; i < depth; i++) {
                    extraStuff += "|_";
                }
                return "\n" + extraStuff + root.data + helpDisplay(root.left, depth +
1) + helpDisplay(root.right, depth + 1);
            }
        }
        return "";
    }


    public static void main(String args[]) {
        TreeGenericArrayList<Integer> t1 = new TreeGenericArrayList<>();
        Integer arr[] = { 6, 0, 1, 3, 6, 5, 4, 7, 9, 2, 12, 15, 28, 32, 48 };
        ArrayList<Integer> arr1 = new ArrayList<>();
        Collections.addAll(arr1, arr);
        t1.setRoot(t1.insertLevelOrderHeap(arr1, t1.getRoot(), 0, true));
        System.out.println();
        System.out.println("\ninorder with heap");
        System.out.println(t1.inOrder(t1.root));
        System.out.println();
        System.out.println("\npreorder with heap");
        System.out.println(t1.preOrder(t1.root));
        System.out.println();
        System.out.println("\npostorder with heap");
        System.out.println(t1.postOrder(t1.root));
        System.out.println();
        System.out.println("\nDisplay as a tree with heap");
        System.out.println(t1.display(t1.root));
        TreeGenericArrayList<Integer> t2 = new TreeGenericArrayList<>();
        ArrayList<Integer> arr2 = new ArrayList<>();
        Collections.addAll(arr2, arr);
        t2.setRoot(t2.insertLevelOrderHeap(arr2, t2.getRoot(), 0, false));
        System.out.println();
        System.out.println("\nIn order without heap");
        System.out.println(t2.inOrder(t2.root));
        System.out.println();
        System.out.println("\npreorder without heap");
        System.out.println(t2.preOrder(t2.root));
        System.out.println();
        System.out.println("\npostorder without heap");
        System.out.println(t2.postOrder(t2.root));
        System.out.println();
        System.out.println("\nDisplay as a tree without heap");
        System.out.println(t2.display(t2.root));
    }
}
```

B. Sample output 1

I. Describe your test 1: see If display method works as desired

II. Text output 1:

6

|_0

```
|_|_3
|_|_|_7
|_|_|_9
|_|_6
|_|_|_2
|_|_|_12
|_1
|_|_5
|_|_|_15
|_|_|_28
|_|_4
|_|_|_32
|_|_|_48
```

III. Screenshot 1:

```
6
|_0
|_|_3
|_|_|_7
|_|_|_9
|_|_6
|_|_|_2
|_|_|_12
|_1
|_|_5
|_|_|_15
|_|_|_28
|_|_4
|_|_|_32
|_|_|_48
```

C. Sample output 2

I. Describe your test 2: checking preorder with heap

II. Text output 2:
48 12 9 7 3 6 2 0 32 28 15 5 6 4 1

III. Screenshot 2:

```
48 12 9 7 3 6 2 0 32 28 15 5 6 4 1
```

D. Sample output 3

I. Describe your test 3: checking preorder without heap

II. Text output 3:
6 0 3 7 9 6 2 12 1 5 15 28 4 32 48

III. Screenshot 3:

```
6 0 3 7 9 6 2 12 1 5 15 28 4 32 48
```