Assignment 8 - Tree Applications, Huffman Encoding and Sorting

Q1:
For implementing Right In-Thread Binary Tree using an array, I defined a tree node type having 4 properties: data, leftIdx, rightIdx, rThread. With use of the node, the tree class can be implemented with a pointer (index) for the root and an arraylist of the treeNode representing the tree data.

To make tree, initialize a root node with a data, and left, right index of -1, which means the pointer points to empty, and setting rThread as false. Then add this treeNode to an array list.

For setting left, the method requires a target node to add the left child: nodeIdx, and the data of the child. With the nodeIdx, the method can access the parent node, create a new child setting rightIdx as nodeIdx and rThread as true, then change the parent's left pointer index.

For setRight method, similar approach is taken, and the unique part is accessing the parent node's rightIdx that is going to be the new child's in-order successor.

```java
import java.util.*;
public class RightInThreadBT {
   public int rootPointer;
   public ArrayList<TreeNode> treeArray;

   RightInThreadBT(){
       this.rootPointer = 0;
       this.treeArray = new ArrayList<TreeNode>();
   }

   public void makeTree(String data){
       this.treeArray.add(new TreeNode(data, -1, -1, false));
   }

   public void setLeft(String data, int nodeIdx){
       if (nodeIdx >= this.treeArray.size()) return;
       TreeNode pNode = this.treeArray.get(nodeIdx);
       if (pNode.leftIdx != -1) return;

       TreeNode lNode = new TreeNode(data, -1, nodeIdx, true);
       int newIdx = this.treeArray.size();
       this.treeArray.add(lNode);
       pNode.leftIdx = newIdx;
   }

   public void setRight(String data, int nodeIdx){
       if (nodeIdx >= this.treeArray.size()) return;
       TreeNode pNode = this.treeArray.get(nodeIdx);
       if (pNode.rightIdx != -1 && !pNode.rThread) return;

       int sucIdx = pNode.rightIdx;
       int newIdx = this.treeArray.size();

       TreeNode rNode = new TreeNode(data, -1, sucIdx, true);
       this.treeArray.add(rNode);
       pNode.rightIdx = newIdx;
       pNode.rThread = false;
   }



class TreeNode {
   public String data;
   public int leftIdx;
   public int rightIdx;
   public boolean rThread;

   TreeNode(String data, int leftIdx, int rightIdx, boolean rThread){
       this.data = data;
       this.leftIdx = leftIdx;
       this.rightIdx = rightIdx;
       this.rThread = rThread;
   }
}
```

Q2:
Traversing can be done by checking the pointer's value and rThread boolean starting from the root node pointer.

```
   public void traverse(){
       TreeNode node = this.treeArray.get(this.rootPointer);
       Stack<String> visited = new Stack<String>();
```

```
    while (node.rightIdx != -1 || !node.rThread){
        // if the node is not rThread, check the node is visited or not
        // if not visited, traverse left child
        // if visited, visit the current node and traverse right child
        if (!node.rThread){
            if (visited.isEmpty() || visited.peek() != node.data){
                visited.push(node.data);
                node = this.treeArray.get(node.leftIdx);
            } else {
                System.out.print(visited.pop());
                node = this.treeArray.get(node.rightIdx);
            }

        // if the node is rThread, the node is a leaf
        // so access the data and traverse to the rightIdx(in-order successor)
        } else {
            System.out.print(node.data);
            node = this.treeArray.get(node.rightIdx);
        }
    }

    // access last element
    System.out.println(node.data);

}
public static void main(String[] args) {
    RightInThreadBT myRightInThreadBT = new RightInThreadBT();

    myRightInThreadBT.makeTree("A");
    myRightInThreadBT.setLeft("B",0);
    myRightInThreadBT.setRight("C",0);
    myRightInThreadBT.setLeft("D",1);
    myRightInThreadBT.setRight("E",1);
    myRightInThreadBT.setLeft("F",2);
    myRightInThreadBT.setRight("G",2);

    myRightInThreadBT.traverse();
}
```

Q3:

```java
public class FibonacciTree{
   public Node fibTree(int n){
       if (n == 0 || n == 1){
           return new Node(n);
       }
       Node head = new Node(n);
       head.left = fibTree(n-1);
       head.right = fibTree(n-2);
       return head;
   }


}


class Node{
   int data;
   Node left;
   Node right;

   Node(int data){
       this.data = data;
       this.left = null;
       this.right = null;
   }
}
```

Q4:

a) Yes. Since the subtrees for n =2, 3, 4 are binary trees and the Fibonacci tree sets the left and right pointers to these subtrees. So every node in a Fibonacci tree has 0 or 2 children.

b) If n =0 or 1, the number of leaves is 1
If n =2, the left subtree (n=1) and the right subtree(n=0) each has 1 leaf, so the total is 2:
If n=3, the left subtree(n=2) has 2 leaves and the right subtree(n=1) has 1 leaf, the total is 3
If n =4, the left subtree (n=3) has 3 leaves and the right subtree (n=2) has 2 leaves, the total is 2+3 =5.

So the number of leaves for any n is a fibonacci number of order n.

c) For n = 0, or 1, the depth is 0.
For n = 2, the depth is 1 + max(depth(1), depth(0)) = 1;
For n = 3, 1 + max(depth(2), depth(1)) = 2
For n = 4, 1 + max(depth(3), depth(2)) = 3
…

Since the left subtree has 1 more depth than the right subtree, and for each increment of n, the depth increases by 1. Therefore for a Fibonacci tree of order n, the depth would be n-1.

Q5:
Assume there's no duplicate number or element in array A, the output array size is the same as A.
If there's any duplicate element, for i = index of the element, Output[count[i]] is accessing the same location, so some field would have empty field

```java
public class MySort {
   public static void main(String[] args) {
       int[] arr = {4,7,8,5,9,2,3,1};
       int[] rev = mySort(arr);

       for (int i=0; i< rev.length; i++){
           System.out.print(rev[i]+ " ");
       }

       System.out.println();

   }

   public static int[] mySort(int[] arr){
       int len = arr.length;
       int[] output = new int[len];

       for (int i = 0; i< len; i++){
           int num = arr[i];
           int cnt = 0;
           for (int j = 0; j< len; j++){
               if (arr[j]< num) cnt++;
           }
           output[cnt] = arr[i];
```

```
        }

        return output;

    }

}
```

Q6:

The time complexity is O(n^2).

Since the function runs a nested loop, one number A[i] is accessed then compared with numbers from the entire array. So the number of the comparison would be n^2. Also assigning a new value to an output array would take C*n time, so the total cost would be O(n^2 + C*n) = O(n^2).

Even if the input array is random, ordered or reverse order, total comparison would not change. Therefore, there’s no impact on the time and cost complexity.