Trường Đại Học Quốc Tế - ĐHQG TP.HCM

LAB REPORT

Course: Algorithms & Data Structures LAB 7

Full Name: Trần Minh Phúc

Student's ID: ITCSIU24070

1 Add a method that counts the elements in a binary tree into the Tree Class. Specifically, the method takes no parameters and returns an integer equal to the number of elements in the tree.

```java
public static int countNode(Node root) {
    if(root == null) return 0;
    return 1 + countNode(root.leftChild) + countNode (root.rightChild);
}
```

2 Add a method that computes the height of a binary tree into the Tree Class. Specifically, this method has no parameters and returns an integer equal to the height of the tree.

```java
public static int treeHeight(Node root) {
    if(root == null) return 0;
    int leftHeight = treeHeight (root.leftChild);
    int rightHeight = treeHeight (root.rightChild);
    return 1 + Math.max(leftHeight, rightHeight);
}
```

3 Add a method that counts a binary tree's leaves tree into the Tree Class. Specifically, this method has no parameters and returns an integer equal to the number of leaves in the tree.

```java
public static int countLeaves(Node root) {
    if(root == null) return 0;
    if(root.leftChild == null && root.rightChild == null) return 1;
    return countLeaves(root.leftChild) + countLeaves (root.rightChild);
}
```

4 Add a method that determines whether a binary tree is fully balanced. This method takes no parameters and returns a Boolean value: true if the tree is fully balanced and false if not.

```java
public static boolean isBalanced(Node root) {
    if(root == null) return true;
    int leftHeight = treeHeight (root.leftChild);
    int rightHeight = treeHeight (root.rightChild);
    if(Math.abs(leftHeight - rightHeight) > 1) return false;
    return isBalanced(root.leftChild) && isBalanced(root.rightChild);
}
```

5 Define two binary trees to be identical if both are empty or their roots are equal, their left subtrees are identical, and their right subtrees are identical. Design a method that determines whether two binary trees are identical (this method takes a second binary tree as its only parameter and returns a Boolean value: true if the tree receiving the message is identical to the parameter, and false otherwise).

```java
public static boolean areIdentical (Node t1, Node t2){
 if (t1 == null && t2 == null)
     return true;

 if (t1 == null || t2 == null)
     return false;
 boolean t1_is_leaf = (t1.leftChild == null && t1.rightChild == null);
 boolean t2_is_leaf = (t2.leftChild == null && t2.rightChild == null);
 if (t1_is_leaf && t2_is_leaf) {
     return (t1.value == t2.value);
 }
 else if (!t1_is_leaf && !t2_is_leaf) {
     // So sánh phép toán và đệ quy kiểm tra các cây con
     return (t1.operation == t2.operation) &&
            areIdentical(t1.leftChild, t2.leftChild) &&
            areIdentical(t1.rightChild, t2.rightChild);
 }
 else {
     return false;
 }

```

6 Huffman coding

```java
public class HuffmanTree {
    public static void main(String[] args) {
        Tree huffmanTree = new Tree();
        String s = "I am a student at International University. My name is TRAN MINH PHUC. I am working on a DSA lab.";
        int[] freqArray = new int[256];
        for (char c : s.toCharArray()) {
            freqArray[c]++;
        }
        PriorityQueue<Node> pq = new PriorityQueue<>(new NodeComparator());

        for (int i = 0; i < 256; i++) {
            if (freqArray[i] > 0) {
                Node newNode = new Node();
                newNode.cData = (char) i;
                newNode.freq = freqArray[i];
                pq.add(newNode);
            }
        }
        for (int i = 0; i < 256; i++) {
            if (freqArray[i] > 0) {
                System.out.println("'" + (char) i + "': " + freqArray[i]);
            }
        }
        for (Node item : pq) {
            System.out.print(item.cData + " ");
        }
        System.out.println();
        for (Node item : pq) {
            System.out.print(item.freq + " ");
        }
        System.out.println();
        while (pq.size() > 1) {
            Node left = pq.poll();
            Node right = pq.poll();
            Node parent = new Node();
            parent.cData = '*';
            parent.freq = left.freq + right.freq;
            parent.leftChild = left;
            parent.rightChild = right;
            pq.add(parent);
        }

        huffmanTree.root = pq.poll();
        huffmanTree.showTree(0, huffmanTree.root);
    }
}
class NodeComparator implements java.util.Comparator<Node> {
    @Override
    public int compare(Node x, Node y) {
        return x.freq - y.freq;
    }
}
```

7 TreeApp.java

- During the implementation, counters were added to measure the number of comparisons performed by each operation.

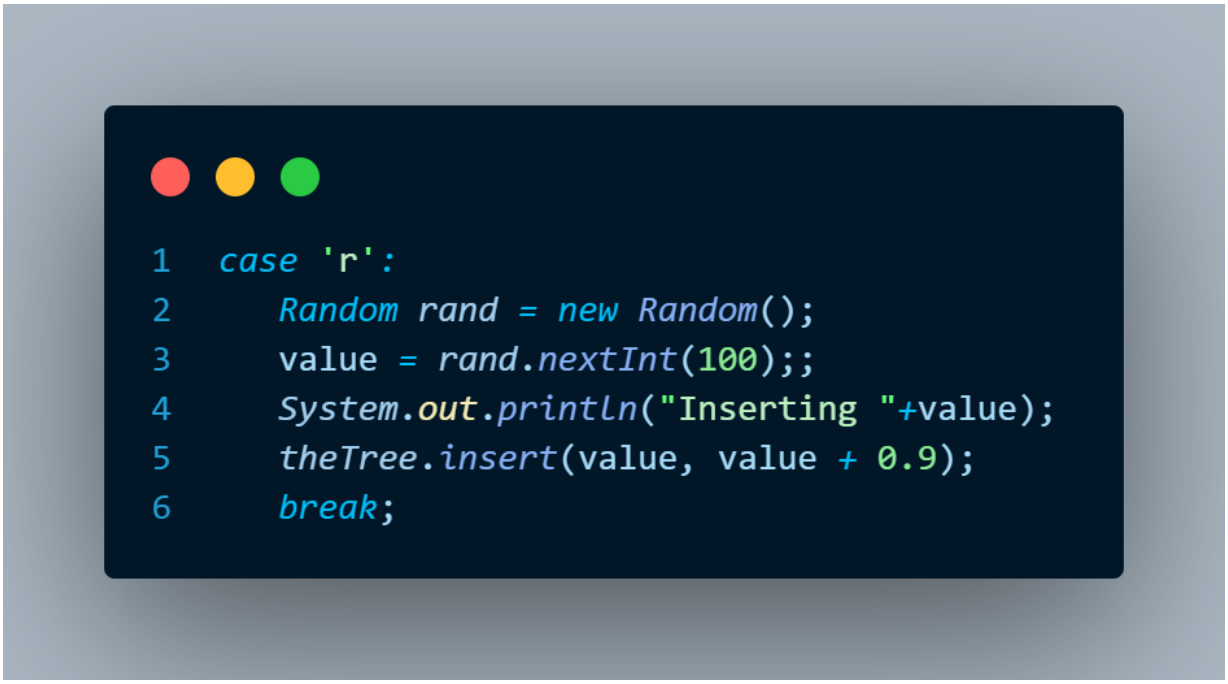
  find(): The number of comparisons grows proportionally to the height of the tree. In a well-balanced BST, this is O(log n), but in an unbalanced tree it can degrade to O(n).

  insert(): Similar to find(), insertion compares keys while traversing down the tree. Efficiency is also O(h), where h is the height.

  delete(): Deletion requires locating the node and then adjusting the structure (handling 0, 1, or 2 children). Its comparisons remain O(h), but are typically slightly higher than find() and insert() due to structural adjustments.

- 

```
case 'c':
    theTree.root = null;
    theTree = new Tree();
    System.out.println("Tree cleared");
    break;
```

- 

```
case 'r':
    Random rand = new Random();
    value = rand.nextInt(100);;
    System.out.println("Inserting "+value);
    theTree.insert(value, value + 0.9);
    break;
```

- 

```
public int minItem() {
   Node current = root;
   if(current == null) return -1; // tree is empty
   while(current.leftChild != null) {
      current = current.leftChild;
   }
   return current.iData;
}
public int maxItem() {
   Node current = root;
   if(current == null) return -1; // tree is empty
   while(current.rightChild != null) {
      current = current.rightChild;
   }
   return current.iData;
}
```

- Different traversals create significantly different BST shapes, proving that traversal sequences do not

preserve structural information. Only the set of values is preserved, not the tree's shape.

This is the end of the report