**LAB 7 REPORT**

Tree.java

First of all, to do the tree, we need to know specific nodes we add and it legitimately, so we are going to create a class names **TreeNode.java** at the beginning:

A screen shot of a computer program

Description automatically generated

This class represents a node in a binary tree.

Each node contains:

* Value
* Left child and right child

After that, we will move on the the Tree class (**Tree.java**), which contain various methods to work with problems

Problem 1: Counting the elements

Can understand that we make the method to count the number of Nodes that added to the tree:

A computer screen with text and images

Description automatically generated

This method will

* Returns 0 if the node is null.
* Otherwise, it recursively counts the nodes in the left and right subtrees, then adds 1 for the current node.

Problem 2: Compute the height of the tree

A computer screen shot of code

Description automatically generated

This method will

* Returns -1 if the node is null (base case).
* Recursively calculates the height of the left and right subtrees.
* Returns the maximum height of the two subtrees plus 1.

Problem 3: Counts the number of leaves in the tree

**A computer screen shot of code

Description automatically generated**

The method countLeaves() will

* Returns 0 if the node is null.
* Returns 1 if the node has no children (left == null and right == null).
* Otherwise, recursively counts leaves in the left and right subtrees.

Problem 4: Checks if the tree is fully balanced: if height difference between the left and right subtrees of every node is at most 1

A computer code on a black background

Description automatically generated

The method isBalanced() will

* Returns true if the node is null (base case).
* Recursively checks if the left and right subtrees are fully balanced.
* Checks if the height difference between the left and right subtrees is at most 1.

We also add method to add the input value to the tree:

A computer screen shot of code

Description automatically generated

add() method:

* Starts the insertion process from the root using the addRecursive helper.

addRecursive() method:

* If the current node is null, it creates a new node with the value.
* If the value is smaller, it moves to the left subtree.
* If the value is larger, it moves to the right subtree.

Output:

A screen shot of a computer

Description automatically generated

TreeApp.java

Problem 5: Compare the roots, left subtrees, and right subtrees of two trees.

A computer screen shot of code

Description automatically generated

Base Case 1: If both nodes being compared are null, they are identical, so return true.

Base Case 2: If one node is null and the other is not, they are not identical, so return false.

Recursive Case:

* Check if the values of the current nodes are the same.
* Recursively check the left subtrees and the right subtrees.

If all checks pass, the trees are identical; otherwise, they are not.

Output:

A screenshot of a computer program

Description automatically generated

Problem 5:

The binary search tree maintains the binary search property: values less than a node are on the left, values greater than a node are on the right.

The insert() method places nodes in the correct position based on comparisons.

A screen shot of a computer code

Description automatically generated

The delete() method handles three cases for node deletion: node with no children, node with one child, and node with two children (in this case, the smallest value from the right subtree is used to replace the node).

A screen shot of a computer program

Description automatically generated

FindMin() and FindMax() allow to get the smallest and largest values in the tree

A computer screen shot of code

Description automatically generated

The in-order traversal stores the tree values in sorted order into an array.

A computer screen shot of a program code

Description automatically generated

The comparison count helps track the number of comparisons made during operations, which can be used to evaluate the performance and efficiency of the tree. Also can clean the tree if we want using **clear()** method

A computer screen with text on it

Description automatically generated

Output:

A screenshot of a computer program

Description automatically generated

Results from Different Traversals:

* In-Order Traversal: The tree will be reinserted with values in sorted order, so the tree remains a balanced binary search tree.
* Pre-order Traversal: If we use pre-order to save and reinsert values, the tree may be skewed depending on the order.
* Post-order Traversal: Similar to pre-order, this will likely cause an unbalanced tree when reinserting values.
* In-order traversal keeps the tree balanced when reinserting values because the elements are inserted in sorted order.
* Pre-order and post-order traversals may create skewed trees since the insertion order does not guarantee balance.

Huffman coding

Text:” I am a student at International University. My name is Ta Thi Phuong Thao. I am working on a DSA lab”

A screenshot of a computer

Description automatically generated

