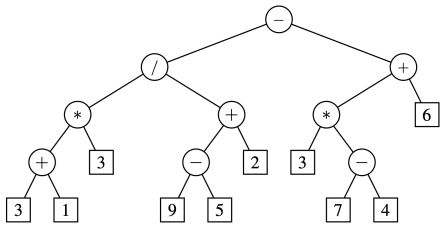
Data Structures: 5th Recitation Questions (with solutions)

1. Tree Traversals

   Write the pre-order, in-order, and post-order traversal of the following tree. Include the tree diagram below:



   Solution. See the quiz solution for an example

2. Tree Construction

   Draw a binary tree T that simultaneously satisfies the following conditions:

   - Each internal node of T stores a single character.
   - A pre-order traversal of T yields EXAMFUN.
   - An in-order traversal of T yields MAFXUEN.

   Solution. (a) **Identify the Root:** The first element in the preorder traversal is the root of the binary tree (or subtree).

   (b) **Split the Tree:** Utilize the root found in the preorder sequence to divide the inorder sequence into two parts: the left and the right subtrees. Elements to the left of the root in the inorder sequence are part of the left subtree, and those to the right are part of the right subtree.

   (c) **Recurse for Subtrees:**

   - For the left subtree, use the next element in the preorder sequence as the root, and split the left part of the inorder sequence based on this new root.
   - Repeat the process for the right subtree with the remaining elements.

(d) **Terminate the Recursion:** If a subtree contains no elements, it is a leaf node, and the recursive process stops.

3. Traversal Operations

Design algorithms for the following operations for a binary tree T:

a) `preorderNext(`p`)`: Return the position visited after p in a preorder traversal of T (or null if p is the last node visited).
b) `inorderNext(`p`)`: Return the position visited after p in an inorder traversal of T (or null if p is the last node visited).
c) `postorderNext(`p`)`: Return the position visited after p in a postorder traversal of T (or null if p is the last node visited).

What are the worst-case running times of your algorithms?

Solution. a) `preorderNext(`p`)`: To find the node visited after p in a preorder traversal (*Root, Left, Right*), check if p has a left child; if so, return it. Otherwise, find the nearest ancestor of p that has an unvisited right child and return this right child. If no such ancestor exists, return null. **Worst-case running time:** $O(h)$, where h is the height of the tree.

b) `inorderNext(`p`)`: In an inorder traversal (*Left, Root, Right*), if p has a right child, return the leftmost node of p's right subtree. If p has no right child, ascend ancestors until finding a node that is the left child of its parent; return this parent. If such a node doesn't exist, p is the last node, and return null. **Worst-case running time:** $O(h)$.

c) `postorderNext(`p`)`: For postorder traversal (*Left, Right, Root*), if p is the right child of its parent or a left child with no sibling, return its parent. If p is a left child with a right sibling, return the leftmost node of its sibling's subtree. If p is the root, return null. **Worst-case running time:** $O(h)$.

The worst-case running times for these algorithms are all $O(h)$, as they may require traversing up to the height of the tree in the case of deeply nested structures.

4. Subtree Height Computation

Give an efficient algorithm that computes and prints, for every position p of a tree T, the element of p followed by the height of p's subtree.

Solution. See the code below

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}

public class SubtreeHeightComputation {
    public static void printSubtreeHeights(TreeNode root)
{
        computeHeight(root);
    }
    private static int computeHeight(TreeNode node) {
        if (node == null) {
            return -1; // Height of empty tree is -1
        }
        // Compute the height of left and right subtrees
        int leftHeight = computeHeight(node.left);
        int rightHeight = computeHeight(node.right);
        // Height of the current node's subtree
        int subtreeHeight = 1 + Math.max(leftHeight,
rightHeight);
        // Print the current node's value and its subtree
height
        System.out.println("Node: " + node.val + ",
Subtree Height: " + subtreeHeight);
        return subtreeHeight;
    }
}

```

Listing 1: Subtree Height Computation Algorithm

The algorithm recursively computes the height of each subtree rooted at a given node and prints the node's value along with its subtree height. The time complexity of this algorithm is $O(n)$, where n is the number of nodes in the tree, since each node is visited exactly once during the postorder traversal.

5. Lowest Common Ancestor*

   Let T be a tree with n positions. Define the *lowest common ancestor* (LCA) between two positions p and q as the lowest position in T that has both p and q as descendants (where we allow a position to be a descendant of itself). Given two positions p and q, describe an efficient algorithm for finding the LCA of p and q. What is the running time of your algorithm?

   Solution. To find the LCA efficiently, we can use a recursive approach that traverses the tree starting from the root. The algorithm works as follows:

```
public class LowestCommonAncestor {

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) {
            return root;
        }

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);

        // If we got non-null responses from both the left and right children,
        // then p and q are in different subtrees, and root is their LCA
        if (left != null && right != null) {
            return root;
        }

        // Otherwise, if one of the children returned a non-null value,
        // that means both p and q are located in the same subtree.
        // Return whichever is non-null.
        return left != null ? left : right;
    }

}


```

Listing 2: Lowest Common Ancestor Algorithm