Iterative Postorder Traversal DSA

Give DSA notes for iterative post order traversal, with clean, modular & readable code. Include both approaches. Use nested while loop for approach 2, add comments on required places on code with better understanding, Also mention why approach 1 working and why approach 2 working

5/5

Here are detailed notes on **Iterative Postorder Traversal** of a binary tree, along with **both approaches**.

Iterative Postorder Traversal

In **postorder traversal**, we visit nodes in the following order:

1. Left Subtree
2. Right Subtree
3. Root

Since postorder is **Left → Right → Root**, but **stack follows LIFO**, we need a **reversal mechanism** when using stacks.

Approach 1: Using Two Stacks

How It Works?

- We use **two stacks**:
    1. The first stack (`stack1`) is used for normal traversal.
    2. The second stack (`stack2`) stores nodes in **Root → Right → Left** order, which is the **reverse** of postorder traversal.

- We **pop nodes from stack1**, push them into stack2, and push their left and right children into stack1.
- Finally, popping nodes from stack2 gives us postorder traversal.

Code (Approach 1 - Two Stacks)

java

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}

public class PostorderTraversal {

    public static List<Integer> postorderUsingTwoStacks(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Stack<TreeNode> stack1 = new Stack<>();
        Stack<TreeNode> stack2 = new Stack<>();

        stack1.push(root);

        // Step 1: Traverse and store nodes in stack2 in reversed order
        while (!stack1.isEmpty()) {
            TreeNode node = stack1.pop();
            stack2.push(node);

            // Push left first, then right to ensure right is processed first in
stack2
            if (node.left != null) stack1.push(node.left);
            if (node.right != null) stack1.push(node.right);
        }

        // Step 2: Pop from stack2 to get postorder sequence
```

```
        while (!stack2.isEmpty()) {
            result.add(stack2.pop().val);
        }
        return result;
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);
        root.right.left = new TreeNode(6);
        root.right.right = new TreeNode(7);

        System.out.println(postorderUsingTwoStacks(root)); // [4, 5, 2, 6, 7, 3, 1]
    }
}
```

Approach 2: Using One Stack (Nested While Loop)

How It Works?

- This method **simulates recursion using a single stack**.
- It follows a **nested while loop** approach:
  1. **Traverse to the leftmost node**, pushing nodes into the stack.
  2. **Check right subtree**:
     - If the right child exists and is unprocessed, process it first.
     - Otherwise, process the node itself.
  3. Maintain a pointer (`lastVisited`) to track the last processed node.

Code (Approach 2 - One Stack, Nested While Loop)

java

```java
public static List<Integer> postorderUsingOneStack(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    Stack<TreeNode> stack = new Stack<>();
    TreeNode curr = root, lastVisited = null;

    while (curr != null || !stack.isEmpty()) {
        // Step 1: Reach the leftmost node
        while (curr != null) {
            stack.push(curr);
            curr = curr.left;
        }

        // Step 2: Peek the node (do not pop yet)
        TreeNode peekNode = stack.peek();

        // Step 3: If right subtree exists and is unprocessed, process it first
        if (peekNode.right != null && peekNode.right != lastVisited) {
            curr = peekNode.right; // Move to right subtree
        } else {
            // Step 4: Process the current node
            result.add(peekNode.val);
            lastVisited = stack.pop(); // Mark as visited
        }
    }
    return result;
}
```

Why These Approaches Work?

Approach 1 (Two Stacks)

- The first stack stores nodes in **Root → Right → Left** order.
- The second stack **reverses** this order to give **Left → Right → Root** (postorder).
- This approach works because the **first stack pushes right before left**, ensuring that left is processed first when popping from the second stack.

Approach 2 (One Stack)

- This method **simulates recursion** by using a **nested while loop**:
  - The **first while loop** pushes nodes to the leftmost.
  - The **second condition** ensures the right subtree is processed before the node itself.
- The `lastVisited` variable prevents **revisiting right subtrees**, ensuring correct postorder traversal.

Complexity Analysis

Approach	Time Complexity	Space Complexity
Two Stacks	**O(N)**	**O(N)** (extra stack)
One Stack	**O(N)**	**O(N)** (worst case skewed tree)

Final Notes

- **Approach 1 (Two Stacks) is simpler** but uses extra space.
- **Approach 2 (One Stack) is more efficient** but trickier to implement.
- If **space is a constraint**, use **Approach 2**.

Let me know if you need further clarifications! 🚀