Iterative Preorder Traversal DSA

Give DSA notes along with optimized readable code for Iterative Preorder traversal

DSA Notes: Iterative Preorder Traversal

Preorder Traversal Overview

Preorder traversal of a binary tree follows the order:

1. **Visit the root node**
2. **Traverse the left subtree**
3. **Traverse the right subtree**

Unlike recursive traversal, the iterative approach avoids function call stack overhead by using an **explicit stack**.

Optimized Approach: Using Stack

We use a **stack** to mimic the function call behavior:

- Push the root node into the stack.
- While the stack is not empty:
  - Pop the top node.
  - Process it (store the value).
  - Push the **right child first** (so the left child is processed first).
  - Push the **left child**.

Optimized Readable Code in Java

```java
import java.util.*;
```

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

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

public class PreorderTraversal {
    public List<Integer> iterativePreorder(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);

        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            result.add(node.val); // Process the node

            if (node.right != null) stack.push(node.right); // Push right child
first
            if (node.left != null) stack.push(node.left);   // Push left child next
        }
        return result;
    }

    // Driver code to test the traversal
    public static void main(String[] args) {
        PreorderTraversal tree = new PreorderTraversal();
        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);

        List<Integer> preorder = tree.iterativePreorder(root);
        System.out.println(preorder); // Output: [1, 2, 4, 5, 3]
    }
}
```

Time & Space Complexity

- **Time Complexity:** $O(N)$, as each node is processed once.
- **Space Complexity:** $O(H)$, where H is the height of the tree.
  - **Worst case (skewed tree):** $O(N)$
  - **Best case (balanced tree):** $O(\log N)$

This approach ensures readability, efficiency, and avoids recursive stack overhead. 🚀 Let me know if you need further optimizations!

Go with Dry run

Dry Run of Iterative Preorder Traversal

Let's dry run the given code on the following **binary tree**:

```markdown
      1
    /  \
   2    3
  / \
 4   5
```

Step-by-Step Execution

We use a **stack** to simulate recursion and process nodes in preorder (**Root → Left → Right**).

Initial State:

- Stack: `[1]`
- Result: `[]`

Step 1: Pop `1`

- **Processing Node:** `1`
- **Stack after pushing right (`3`) and left (`2`):** `[3, 2]`
- **Result:** `[1]`

Step 2: Pop `2`

- **Processing Node:** `2`
- **Stack after pushing right (`5`) and left (`4`):** `[3, 5, 4]`
- **Result:** `[1, 2]`

Step 3: Pop `4`

- **Processing Node:** `4`
- **Stack remains unchanged** (No children)
- **Result:** `[1, 2, 4]`

Step 4: Pop `5`

- **Processing Node:** `5`
- **Stack remains unchanged** (No children)
- **Result:** `[1, 2, 4, 5]`

Step 5: Pop `3`

- **Processing Node:** `3`
- **Stack remains unchanged** (No children)

- **Result:** `[1, 2, 4, 5, 3]`

Final Output

📌 **Preorder Traversal:** `[1, 2, 4, 5, 3]` ✅

This dry run confirms that our iterative approach correctly follows **preorder traversal (Root → Left → Right)** while efficiently using a stack. 🚀