Iterative Inorder Traversal DSA

Give DSA notes along with optimized readable code & dry run for Iterative Inorder traversal

Iterative Inorder Traversal (DSA Notes)

1. Understanding Inorder Traversal

Inorder traversal of a binary tree visits nodes in the following order:
Left → Root → Right

2. Why Use an Iterative Approach?

- The recursive approach uses extra stack space due to function calls.
- The iterative approach avoids recursion and explicitly manages the traversal stack, making it more memory-efficient in some cases.

Algorithm: Iterative Inorder Traversal

We use a **stack** to simulate the recursion manually.

1. Initialize an empty stack.
2. Start from the root node and push all left children onto the stack.
3. When a node has no left child, pop the stack, process the node, and move to its right child.
4. Repeat until all nodes are visited.

Optimized Readable Java Code

```java
import java.util.*;

class TreeNode {
```

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

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

public class IterativeInorderTraversal {
    public static List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode curr = root;

        while (curr != null || !stack.isEmpty()) {
            // Push all left nodes onto the stack
            while (curr != null) {
                stack.push(curr);
                curr = curr.left;
            }

            // Process the node
            curr = stack.pop();
            result.add(curr.val);

            // Move to the right subtree
            curr = curr.right;
        }

        return result;
    }

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

        System.out.println(inorderTraversal(root)); // Output: [1, 3, 2]
    }
}
```

Key Optimizations in the Code

- Avoids unnecessary recursion.
- Uses a **while loop** to ensure efficient traversal.
- Maintains a **stack** to store nodes that need processing.

Dry Run (Step-by-Step Execution)

Example Tree

```markdown
    1
     \
      2
     /
    3
```

Execution Steps

Stack	Current Node	Output
`[ ]`	`1`	`[]`
`[1]`	`null`	`[]`
`[ ]`	`1 (pop)`	`[1]`
`[ ]`	`2`	`[1]`
`[2]`	`3`	`[1]`
`[2, 3]`	`null`	`[1]`
`[2]`	`3 (pop)`	`[1, 3]`
`[2]`	`null`	`[1, 3]`
`[ ]`	`2 (pop)`	`[1, 3, 2]`

Final Output:

`[1, 3, 2]`

Time & Space Complexity

- **Time Complexity:** $O(N) \rightarrow$ Each node is pushed and popped once.
- **Space Complexity:** $O(H) \rightarrow$ Stack stores nodes up to the height H (worst case $O(N)$ for skewed trees, $O(\log N)$ for balanced trees).

Let me know if you need any modifications! 🚀