Pre-order Traversal Summary

Pre order traversal in short

Pre-order traversal is a tree traversal method where the nodes are visited in the following order:

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

This is typically implemented using recursion or a stack. In a binary tree, for example:

```scss
preOrder(node):
    if node is not null:
        visit(node)
        preOrder(node.left)
        preOrder(node.right)
```

In order traversal in short

In-order traversal is a tree traversal method where the nodes are visited in the following order:

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

For a binary tree, the implementation would look like this:

```scss
inOrder(node):
    if node is not null:
        inOrder(node.left)
        visit(node)
        inOrder(node.right)
```

can u make the code more opti,ized and readable

Sure! Here's an optimized and more readable version of the in-order traversal using iteration instead of recursion, which avoids the overhead of function calls and uses a stack:

java

```java
import java.util.Stack;

public class BinaryTree {
    static class TreeNode {
        int val;
        TreeNode left, right;
        TreeNode(int x) { val = x; }
    }

    public void inOrderTraversal(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;

        while (current != null || !stack.isEmpty()) {
            // Reach the leftmost node of the current subtree
            while (current != null) {
                stack.push(current);
                current = current.left;
            }

            // Current must be null at this point
            current = stack.pop();
            System.out.print(current.val + " ");

            // Visit the right subtree
            current = current.right;
        }
    }
}
```

Key improvements:

1. **Iterative Approach**: Uses a stack to avoid recursion, making it more space-efficient for large trees.

2. **Readability**: The code is broken down into clear steps: move to the leftmost node, visit the node, and then process the right subtree.

This approach ensures O(n) time complexity and O(h) space complexity, where `n` is the number of nodes and `h` is the height of the tree.