Topic Introduction

Today's PrepLetter dives into **Binary Tree Traversals**: the "routes" we take to systematically visit every node in a binary tree. If you've worked with trees before, you know that order matters — and in coding interviews, understanding traversal techniques is key.

What is a Tree Traversal?
A traversal is a well-defined method of visiting every node in a tree exactly once. In binary trees (where each node has up to two children), traversals are fundamental for searching, copying, printing, or evaluating tree-based data. The three classic depth-first traversals are:

- **Preorder**: Visit the root, then the left subtree, then the right subtree.
- **Inorder**: Visit the left subtree, then the root, then the right subtree.
- **Postorder**: Visit the left subtree, then the right subtree, then the root.

Why do these matter in interviews?
Interviewers love these problems because they test your understanding of recursion, stacks, and problem decomposition. Traversals also underpin tree serialization, expression evaluation, and more.

Simple Example (not using our three problems):
Suppose you have this tiny tree:

```
   2
  / \
 1   3
```

- **Inorder**: 1, 2, 3
- **Preorder**: 2, 1, 3
- **Postorder**: 1, 3, 2

Notice how the "visit order" defines what you see — and can be achieved recursively or with an explicit stack (iterative).

Why group these three problems?

Today's three problems are the bedrock of tree questions:

- **Binary Tree Inorder Traversal**: Visit nodes left-root-right.
- **Binary Tree Preorder Traversal**: Visit nodes root-left-right.
- **Binary Tree Postorder Traversal**: Visit nodes left-right-root.

They all require you to systematically process every node, typically using recursion or an explicit stack. Mastering these will make any tree-based problem feel much less intimidating!

Problem 1: Binary Tree Inorder Traversal

Problem Statement (Rephrased):

Given the root of a binary tree, return a list containing the values of its nodes as you traverse the tree in *inorder* (left, root, right) order.

Leetcode Link

Example Input and Output:

Given this tree:

```
    1
     \
      2
     /
    3
```

Expected output: `[1, 3, 2]`

How to Think About It:

- Inorder traversal means: For every node, visit its left subtree first, then the node itself, then its right subtree.
- If you draw out the tree and trace this rule, you'll see the correct sequence.
- Try it on paper with the above tree!

Another Test Case:

Tree:

```
    2
   / \
  1   3
```

Expected output: `[1, 2, 3]`

Brute-Force Approach:

- You could collect all nodes in any order and sort them, but that won't work for non-BSTs and loses the traversal intent.
- Time complexity: O(n log n) (because of sorting).
- This is not correct for general trees—order matters!

Optimal Approach:

- Use **recursion** or an **explicit stack** for iterative traversal.
- Recursion leverages the call stack to "remember" where we are.
- The key pattern:
  - Traverse left subtree (recursive call)
  - Visit node (add value to result)
  - Traverse right subtree (recursive call)

Python Solution:

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
```

```
        self.right = right

def inorderTraversal(root):
    result = []

    def inorder(node):
        if node is None:
            return
        # Traverse left subtree
        inorder(node.left)
        # Visit node
        result.append(node.val)
        # Traverse right subtree
        inorder(node.right)

    inorder(root)
    return result
```

Time Complexity: O(n) (every node visited once)

Space Complexity: O(n) (result list + call stack in worst case)

Code Walk-Through:

- `inorderTraversal` is the main function.
- `result` will hold our answer.
- The inner `inorder` function is our recursive helper:
    - If the node is `None`, do nothing (base case).
    - Recursively traverse the left child.
    - Append the current node's value.
    - Recursively traverse the right child.

Trace with Example:

For the tree:

```
    1
     \
      2
     /
    3
```

- Start at 1: left is None.
- Visit 1 -> result: [1]
- Move to 2 (right child of 1)
    - Left child is 3
        - 3's left is None.
        - Visit 3 -> result: [1, 3]
        - 3's right is None.

- Visit 2 -> result: [1, 3, 2]
- 2’s right is None.

Try This Tree Yourself:

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

Expected output: `[1, 2, 3, 4, 5]`

Give it a try on your own before checking the code!

Reflection Prompt:
Did you know you can also do inorder traversal iteratively using a stack? Try rewriting this solution with an explicit stack after you’re comfortable with the recursive approach!

Problem 2: Binary Tree Preorder Traversal

Problem Statement (Rephrased):
Return the preorder traversal (root, left, right) of a given binary tree’s nodes as a list.
[Leetcode Link]

How is this different?

- The only change: Visit the node BEFORE traversing left and right children.
- It’s the same concept and technique, just a different order.

Example Input and Output:
Given:

```
    1
     \
      2
     /
    3
```

Expected output: `[1, 2, 3]`

Another Test Case:

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

Expected output: `[4, 2, 1, 3, 5]`

Brute-Force Approach:

- Like before, collecting nodes randomly or in any order won't work. The traversal order must be preserved.

Optimal Approach:

- Use recursion:
    - Visit node (add value to result)
    - Traverse left subtree (recursive call)
    - Traverse right subtree (recursive call)

Pseudocode:

```
function preorder(node):
    if node is null:
        return
    add node.val to result
    preorder(node.left)
    preorder(node.right)
```

Step-by-Step Example:

With the sample tree:

- At 1: add 1 -> [1]
- Go right to 2: add 2 -> [1, 2]
- Go left to 3: add 3 -> [1, 2, 3]

Try This Test Case:

```
    5
   / \
  3   7
```

Expected output: `[5, 3, 7]`

Time Complexity: O(n)

Space Complexity: O(n) (result + call stack)

Walk-through:

- The approach is nearly identical to inorder, but the "visit" step comes first.
- The only code change: move the `append` line before the recursive calls.

Try implementing this in code yourself — it's a great way to reinforce the pattern!

Problem 3: Binary Tree Postorder Traversal

Problem Statement (Rephrased):

Return the postorder traversal (left, right, root) of a given binary tree's nodes as a list.

Leetcode Link

What's different here?

- Now you visit the node LAST: left subtree, then right subtree, then the node itself.
- This is trickier for some, especially for iterative solutions.

Example Input and Output:

Given:

```
    1
     \
      2
     /
    3
```

Expected output: `[3, 2, 1]`

Another Test Case:

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

Expected output: `[1, 3, 2, 5, 4]`

Optimal Approach (Pseudocode):

```
function postorder(node):
    if node is null:
        return
    postorder(node.left)
    postorder(node.right)
    add node.val to result
```

- Same recursion as before, but now the "visit" step is last.

Try This Test Case:

```
    5
   / \
  3   7
```

Expected output: `[3, 7, 5]`

Time Complexity: O(n)

Space Complexity: O(n) (result + call stack)

Implement this in your editor or on paper. Notice how the recursion "unwinds" and adds values at the last step!

Subtle Nudge:

Can you implement postorder traversal using an explicit stack? It's a bit trickier than inorder or preorder, but great for strengthening your iterative skills.

Summary and Next Steps

Today, you tackled the three core binary tree traversals: **inorder, preorder, and postorder**.

- They all use the same underlying pattern — recursion (or an explicit stack) — but differ in the order nodes are visited.
- Remember:
  - **Inorder**: left, root, right
  - **Preorder**: root, left, right
  - **Postorder**: left, right, root
- Practicing these helps you master recursion, decomposition, and stack-based thinking.

Common Traps:

- Mixing up the order of operations (write it on paper or comment it in code!)
- Forgetting to handle the null (base) case.
- Forgetting to collect the result in the right place (before, between, or after recursive calls).

Action List:

- [ ] Implement all three traversals on your own, even the one with code here.
- [ ] Try each traversal using both recursion and an explicit stack (iterative).
- [ ] Dry-run your code on several trees, including edge cases (empty tree, single node, skewed trees).
- [ ] Compare your solutions with others and check for off-by-one or order mistakes.
- [ ] Explore "Morris Traversal" and iterative postorder as stretch goals.
- [ ] Don't worry if you get stuck — the key is to practice, reflect, and try again!

Happy coding — and may the recursion be with you!