Binary Tree Traversals — Preorder, Inorder, Postorder

Binary tree traversals are fundamental operations that define the order in which all nodes are visited. Each node may be processed before, between, or after visiting its children. The three core types are: **Preorder**, **Inorder**, and **Postorder**.

1■■ Preorder Traversal (Root → Left → Right)

Logic: Visit the root first, then traverse the left subtree, followed by the right subtree.
Use cases: Tree serialization, prefix notation, or copying trees.

```
def preorder(root):
    if not root:
        return []
    return [root.val] + preorder(root.left) + preorder(root.right)

def preorder_iterative(root):
    if not root:
        return []
    stack, res = [root], []
    while stack:
        node = stack.pop()
        res.append(node.val)
        if node.right: stack.append(node.right)
        if node.left:  stack.append(node.left)
    return res
```

2■■ Inorder Traversal (Left → Root → Right)

Logic: Traverse left subtree first, then visit the root, then traverse right subtree.
Use cases: For Binary Search Trees (BST), it gives sorted order of values.

```
def inorder(root):
    if not root:
        return []
    return inorder(root.left) + [root.val] + inorder(root.right)

def inorder_iterative(root):
    res, stack = [], []
    node = root
    while stack or node:
        while node:
            stack.append(node)
            node = node.left
        node = stack.pop()
        res.append(node.val)
        node = node.right
    return res
```

3■■ Postorder Traversal (Left → Right → Root)

Logic: Traverse left and right subtrees first, then visit the root.
Use cases: Deleting trees, evaluating postfix expressions.

```
def postorder(root):
    if not root:
        return []
```

```
    return postorder(root.left) + postorder(root.right) + [root.val]

def postorder_iterative(root):
    if not root:
        return []
    stack, res = [root], []
    while stack:
        node = stack.pop()
        res.append(node.val)
        if node.left:  stack.append(node.left)
        if node.right: stack.append(node.right)
    return res[::-1]
```

■ Traversal Comparison Summary

Traversal	Visit Order	Typical Use Case
Preorder	Root → Left → Right	Tree copy, serialization
Inorder	Left → Root → Right	BST → Sorted order of elements
Postorder	Left → Right → Root	Delete tree, evaluate postfix

Example Combined Implementation

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

# Example Tree
#       1
#      / \
#     2   3
#    / \
#   4   5

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("Preorder:", preorder(root))
print("Inorder:", inorder(root))
print("Postorder:", postorder(root))
```

Output:

Preorder: [1, 2, 4, 5, 3]
Inorder: [4, 2, 5, 1, 3]
Postorder: [4, 5, 2, 3, 1]

■ Time & Space Complexity

Traversal Type	Time Complexity	Space Complexity (Recursion Stack)
Preorder	O(n)	O(h) (height of tree)

| Inorder | O(n) | O(h) |
| Postorder | O(n) | O(h) |