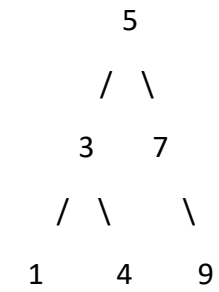
PostOrder Traversal of Binary Tree LeetCode

Post-order traversal: Left subtree, right subtree, current node

Example:



Output: **[1, 4, 3, 9, 7, 5]**

Approach 1: Perform an post-order traversal of the binary tree using recursion

- Define a recursive **solve** function to traverse the binary tree in the following order:
    - Recursively visit the left subtree.
    - Recursively visit the right subtree.
    - Push the value of the current node.
- In the **postOrderTraversalRecursively** function, call the **solve** function and store the results in a vector.
- **Time Complexity: O(N) as it visits each node exactly once.**
- **Space Complexity: O(N) for the function call stack and the vector.**

Approach 2: Perform an post-order traversal of the binary tree using an iterative approach

- Initialize an empty vector **ans** to store the traversal result and a stack **st** to help traverse the tree iteratively.
- Use two pointers: **currNode** for the current node and **lastVisited** for the last visited node.
- While **currNode** is not null or the stack is not empty:
    - Inside the first while loop:
        - Push the current node and move to its left child.
    - In the second while loop, if the current node exists:

        - If the current node has a right child and it has not been visited yet, move to its right child.
        - If there is no right child or it has already been visited, process the current node (push its value to **ans**), pop it from the stack, and update **lastVisited**.
- In the **postOrderTraversalIteratively** function, return the **ans** vector.
- **Time Complexity: O(N) as it visits each node exactly once.**
- **Space Complexity: O(H), where H is the height of the binary tree. In the worst case, where the tree is skewed, H could be N, making the space complexity O(N). In a balanced tree, it is O(log N).**

Approach 3: Morris Traversal Algorithm to perform an iterative Postorder traversal of a binary tree

- Create an empty vector **ans** to store the traversal result.
- Start from the root node as **currNode**.
- While **currNode** is not null:
    - If the current node has no right child, visit it and move to its left child.
    - If the current node has a right child, find its in-order predecessor:
        - Initialize **predecessor** to the right child.
        - Traverse to the leftmost node of the right subtree if not visited already.
        - If the predecessor's left child is not assigned, assign it to the current node, visit the current node, and then move to the right child.
        - If the predecessor's left child is already assigned, reset it to nullptr and move to the left child of the current node.
- Reverse the result vector to get the post-order traversal order (since Morris traversal generates a reversed post-order sequence).
- Return the reversed **ans** vector.
- **Time Complexity: O(N) as it visits each node exactly once.**
- **Space Complexity: O(1) as it doesn't use additional data structures except for the ans vector.**

Conclusion:

- All three approaches successfully perform a post-order traversal of the binary tree and return the results in the same order.
- The recursive, iterative, and Morris traversal methods all yield the expected traversal sequence.
- The Morris traversal approach offers the advantage of a space complexity of O(1), making it a memory-efficient option for post-order tree traversal.