In this article, we will provide the recursive solution for the three traversal methods we have mentioned. And talk about the implementation of the iterative solution. Finally, we will discuss the difference between them.

*Pre-order Traversal - Recursive Solution*

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  private void preorderTraversal(TreeNode root, List<Integer> answer) {  if (root == null) {  return;  }  answer.add(root.val); // visit the root  preorderTraversal(root.left, answer); // traverse left subtree  preorderTraversal(root.right, answer); // traverse right subtree  }  public List<Integer> preorderTraversal(TreeNode root) {  List<Integer> answer = new ArrayList<>();  preorderTraversal(root, answer);  return answer;  }  } |

*In-order Traversal - Recursive Solution*

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  private void inorderTraversal(TreeNode root, List<Integer> answer) {  if (root == null) {  return;  }  inorderTraversal(root.left, answer); // traverse left subtree  answer.add(root.val); // visit the root  inorderTraversal(root.right, answer); // traverse right subtree  }  public List<Integer> inorderTraversal(TreeNode root) {  List<Integer> answer = new ArrayList<>();  inorderTraversal(root, answer);  return answer;  }  } |

*Post-order Traversal - Recursive Solution*

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  private void postorderTraversal(TreeNode root, List<Integer> answer) {  if (root == null) {  return;  }  postorderTraversal(root.left, answer); // traverse left subtree  postorderTraversal(root.right, answer); // traverse right subtree  answer.add(root.val); // visit the root  }  public List<Integer> postorderTraversal(TreeNode root) {  List<Integer> answer = new ArrayList<>();  postorderTraversal(root, answer);  return answer;  }  } |

*Iterative Solution*

There are several iterative solutions for tree traversal. One of the solutions is to use a stack to simulate the recursion process.

Taking pre-order traversal as an example, in each iteration, we pop one node from the stack and visit this node. Then if this node has a right child, push its right child into the stack. If this node has a left child, push its left child into the stack. It is noteworthy that we push the right child first so that we can visit the left child first since the nature of the stack is LIFO(last in first out). After that, we can continue to the next iteration until the stack is empty.

Here are java and c++ codes for your reference.

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  public List<Integer> preorderTraversal(TreeNode root) {  List<Integer> answer = new ArrayList<>();  Stack<TreeNode> s = new Stack<TreeNode>();  if (root != null) {  s.push(root);  }  TreeNode cur;  while (!s.empty()) {  cur = s.pop();  answer.add(cur.val); // visit the root  if (cur.right != null) {  s.push(cur.right); // push right child to stack if it is not null  }  if (cur.left != null) {  s.push(cur.left); // push left child to stack if it is not null  }  }  return answer;  }  } |

*Complexity Analysis*

As we mentioned before, we can traverse a tree recursively to retrieve all the data in pre-order, in-order or post-order. The time complexity is O(N) because we visit each node exactly once. And the depth of the tree might be N in the worst case. That is to say, the level of recursion might be at most N in the worst case. Therefore, taking system stack into consideration, the space complexity is O(N) as well.

To be cautious, the complexity might be different due to a different implementation. It is comparatively easy to do traversal recursively but when the depth of the tree is too large, we might suffer from stack overflow problem. That's one of the main reasons why we want to solve this problem iteratively sometimes.

For the iterative solution, the time complexity is apparently the same with recursion solution which is O(N). The space complexity is also O(N) since in the worst case, we will have all the nodes in the stack. There are some other solutions for iterative traversal which can reduce the space complexity to O(1).