# Question

Given an n-ary tree, return the *postorder* traversal of its nodes' values.

*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).*

**Follow up:**

Recursive solution is trivial, could you do it iteratively?

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

**Output:** [5,6,3,2,4,1]

**Example 2:**



**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** [2,6,14,11,7,3,12,8,4,13,9,10,5,1]

**Constraints:**

* The height of the n-ary tree is less than or equal to 1000
* The total number of nodes is between [0, 10^4]

# Solution

#### **How to traverse the tree**

First of all, please refer to [this article](https://leetcode.com/articles/binary-tree-postorder-transversal/) for the solution in case of binary tree. This article offers the same ideas with a bit of generalisation.

There are two general strategies to traverse a tree:

* Breadth First Search (BFS)

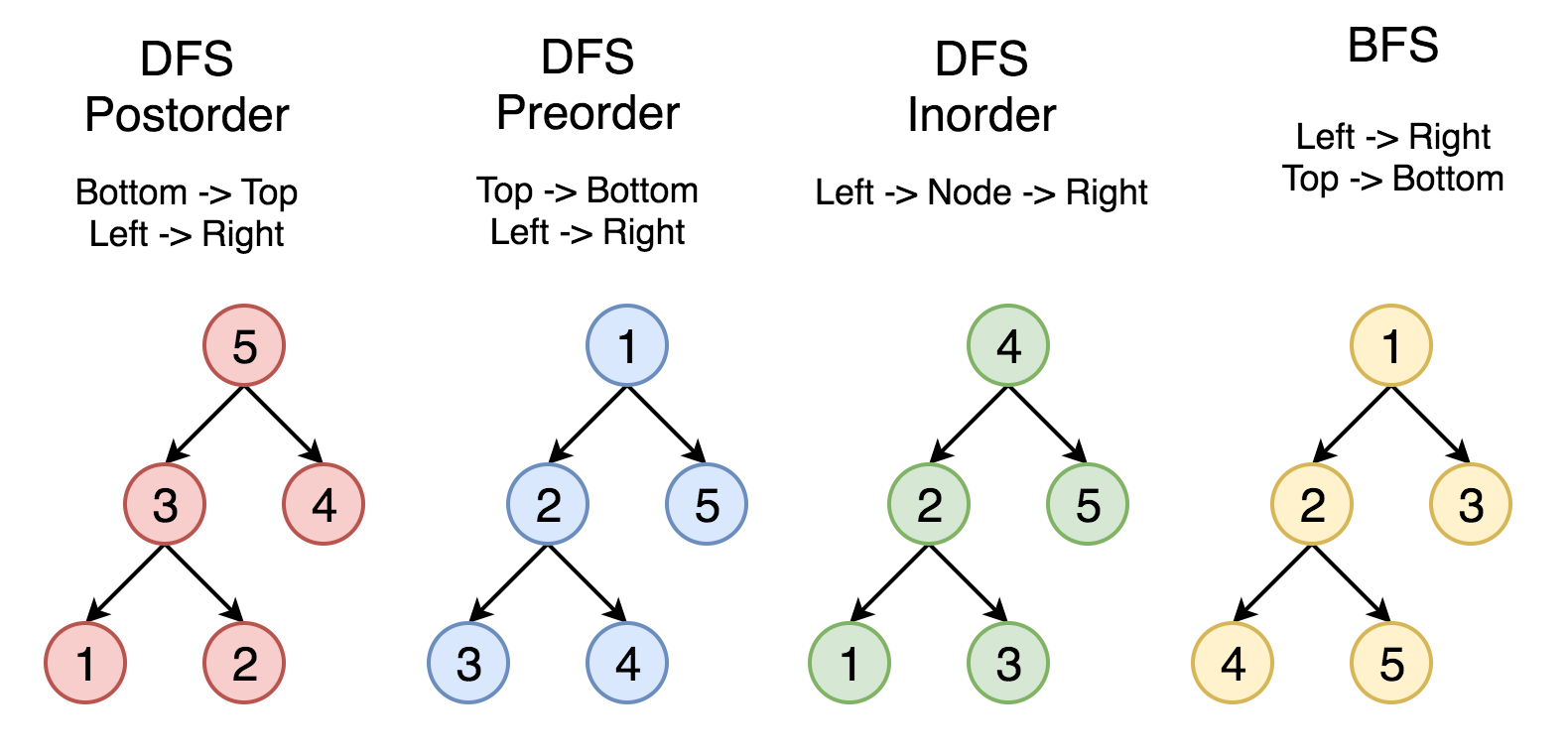
We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

* Depth First Search (DFS)

In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node and right node.

On the following figure the nodes are numerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



Here the problem is to implement postorder traversal using iterations.

#### **Approach 1: Iterations**

**Algorithm**

First of all, here is the definition of the TreeNode which we would use in the following implementation.

|  |
| --- |
| // Definition for a Node.  class Node {  public int val;  public List<Node> children;  public Node() {}  public Node(int \_val, List<Node> \_children) {  val = \_val;  children = \_children;  }  }; |

Let's start from the root and then at each iteration pop the current node out of the stack and push its child nodes. In the implemented strategy we push nodes into stack following the order Top->Bottom and Left->Right. Since DFS postorder traversal is Bottom->Top and Left->Right the output list should be reverted after the end of loop.

|  |
| --- |
| class Solution {  public List<Integer> postorder(Node root) {  LinkedList<Node> stack = new LinkedList<>();  LinkedList<Integer> output = new LinkedList<>();  if (root == null) {  return output;  }  stack.add(root);  while (!stack.isEmpty()) {  Node node = stack.pollLast();  output.addFirst(node.val);  for (Node item : node.children) {  if (item != null) {  stack.add(item);  }  }  }  return output;  }  } |

**Complexity Analysis**

* Time complexity : we visit each node exactly once, thus the time complexity is \mathcal{O}(N)O(*N*), where N*N* is the number of nodes, i.e. the size of tree.
* Space complexity : depending on the tree structure, we could keep up to the entire tree, therefore, the space complexity is \mathcal{O}(N)O(*N*).