# Question

Given an n-ary tree, return the *preorder* 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:** [1,3,5,6,2,4]

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

**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

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

**Algorithm**

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

First of all, here is the definition of the tree Node 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 output list following the order Top->Bottom and Left->Right, that naturally reproduces preorder traversal.

|  |
| --- |
| class Solution {  public List<Integer> preorder(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.add(node.val);  Collections.reverse(node.children);  for (Node item : node.children) {  stack.add(item);  }  }  return output;  }  } |

**Complexity Analysis**

* Time complexity : we visit each node exactly once, and for each visit, the complexity of the operation (i.e. appending the child nodes) is proportional to the number of child nodes n (n-ary tree). Therefore the overall 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*).