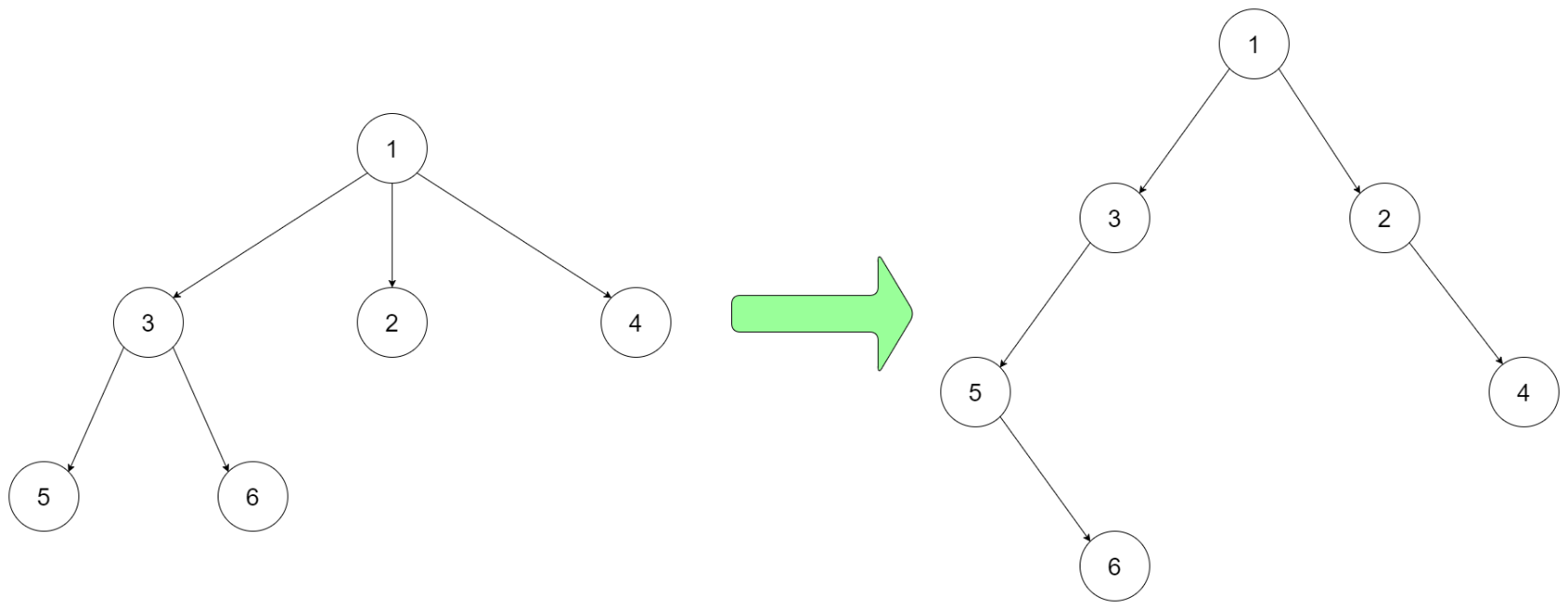
# Question

Design an algorithm to encode an N-ary tree into a binary tree and decode the binary tree to get the original N-ary tree. An N-ary tree is a rooted tree in which each node has no more than N children. Similarly, a binary tree is a rooted tree in which each node has no more than 2 children. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that an N-ary tree can be encoded to a binary tree and this binary tree can be decoded to the original N-nary tree structure.

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

For example, you may encode the following 3-ary tree to a binary tree in this way:



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

Note that the above is just an example which *might or might not* work. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Constraints:**

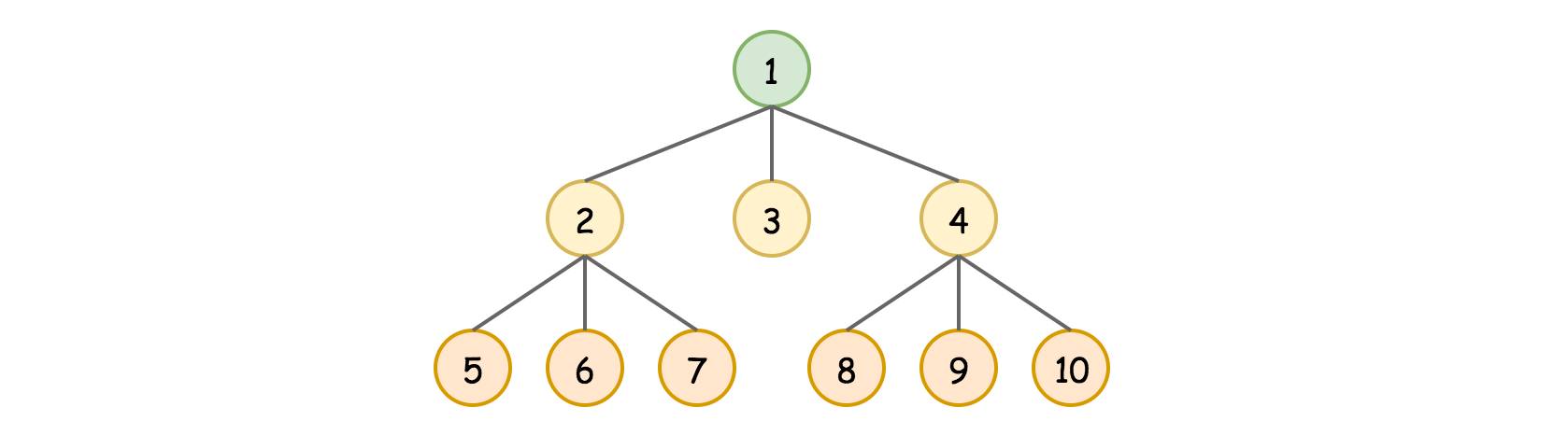
* The height of the n-ary tree is less than or equal to 1000
* The total number of nodes is between [0, 10^4]
* Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

# Solution

#### **Intuition**

There are several ways to encode the N-ary tree to a binary tree. However, a majority of the algorithms could all be traced back to the one that is documented on the [Wikipedia](https://en.wikipedia.org/wiki/M-ary_tree#Convert_a_m-ary_tree_to_binary_tree).

Here we would like to illustrate intuitively the idea first, which we would implement in different manners in the following sections.

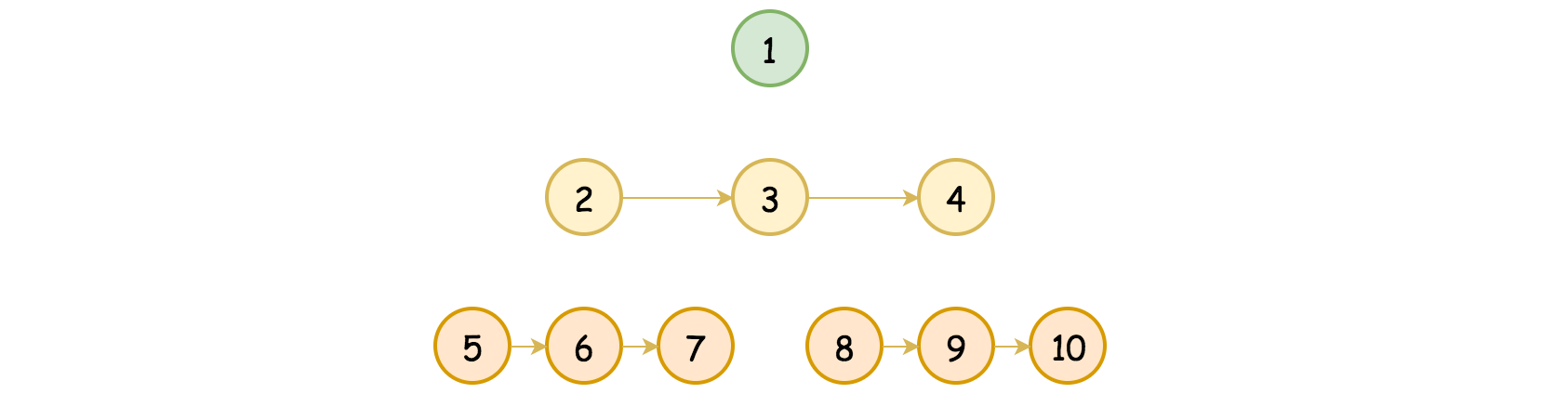


To put it simple, the algorithm can be summarized in two steps. We use the above N-ary tree as an example for demonstration.

Step 1). Link all **siblings** together, like a singly-linked list.

Each node in the original N-ary tree would correspond uniquely to a node in the resulting binary tree.

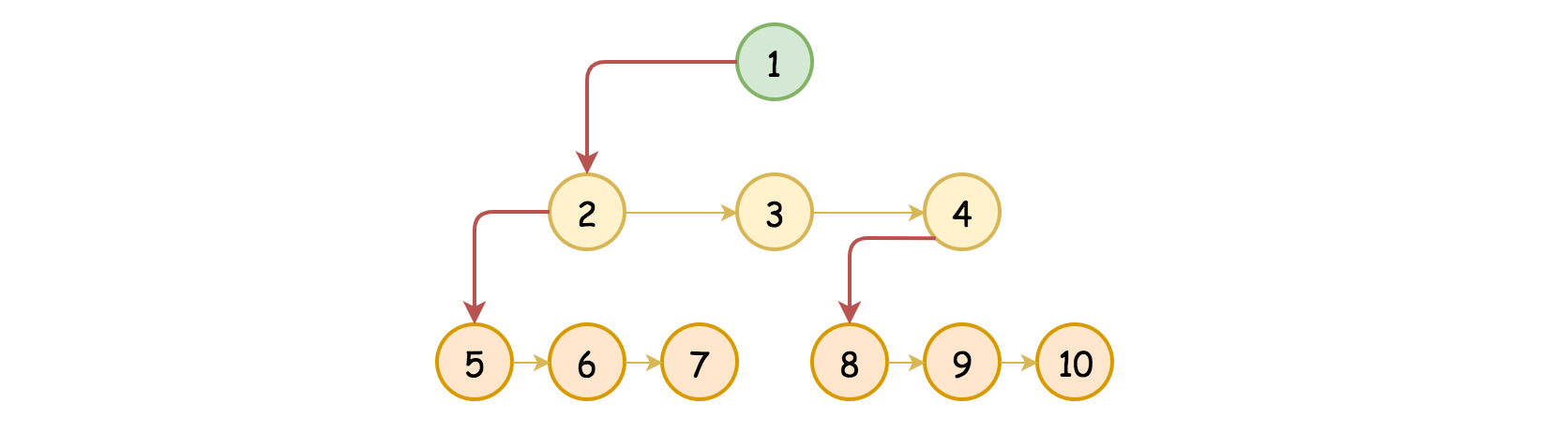
In the first step, we first chain up all the sibling nodes together, i.e. nodes that share the same parent. By chaining up, we would link the nodes via either left or right child pointers of the binary tree node. Here we choose to do with the right pointer.



Step 2). Link the **head** of the obtained list of siblings with its **parent** node.

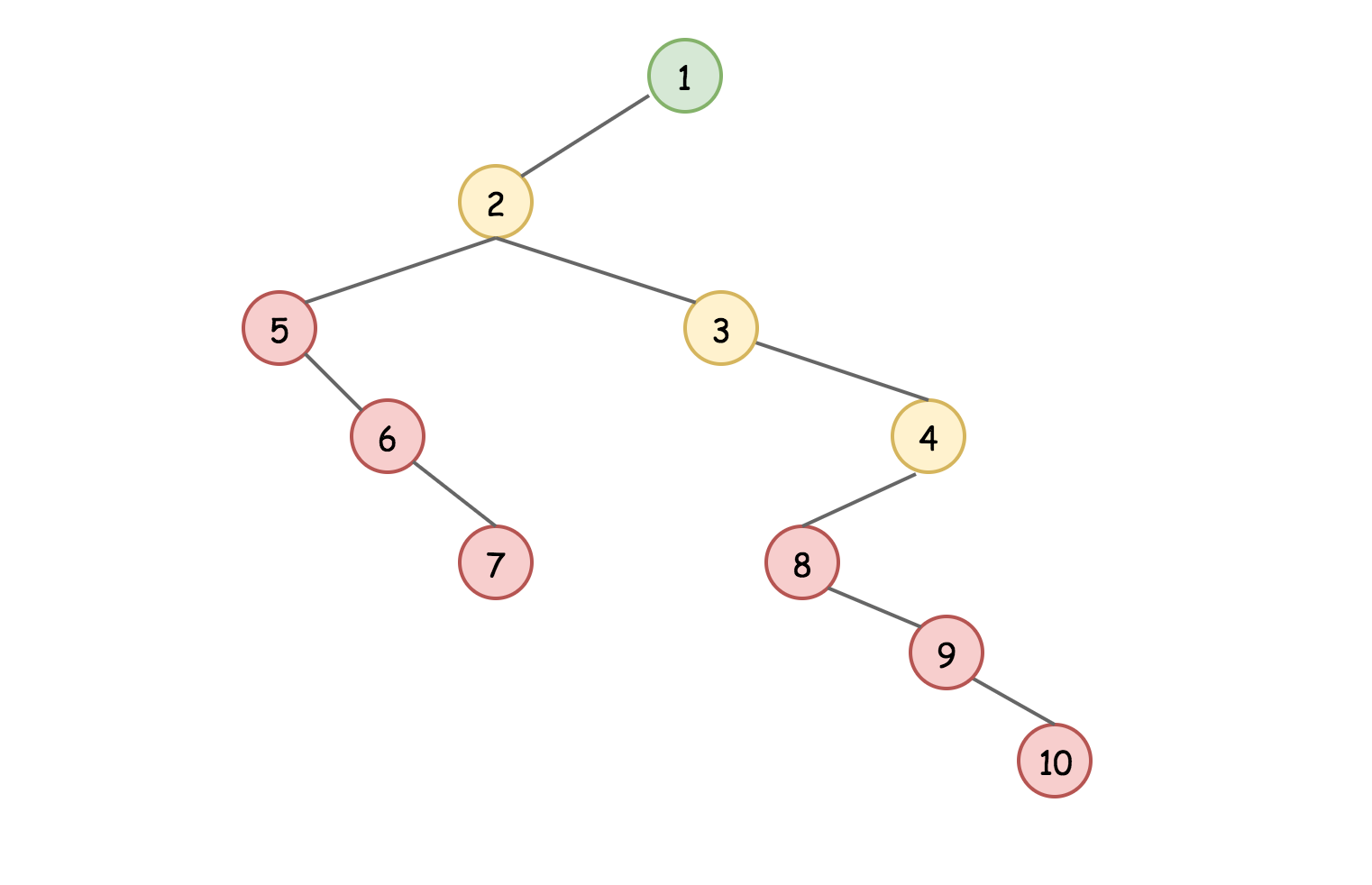
Now that the siblings are chained up, we just need to link this sibling list with their parent node.

As one can see, we don't have to link each one of the siblings to its parent, and we cannot do so either, since we only got two pointers for a node in binary tree. It suffices to pick one of the siblings. Naturally, we could link the head of the list with its parent node.

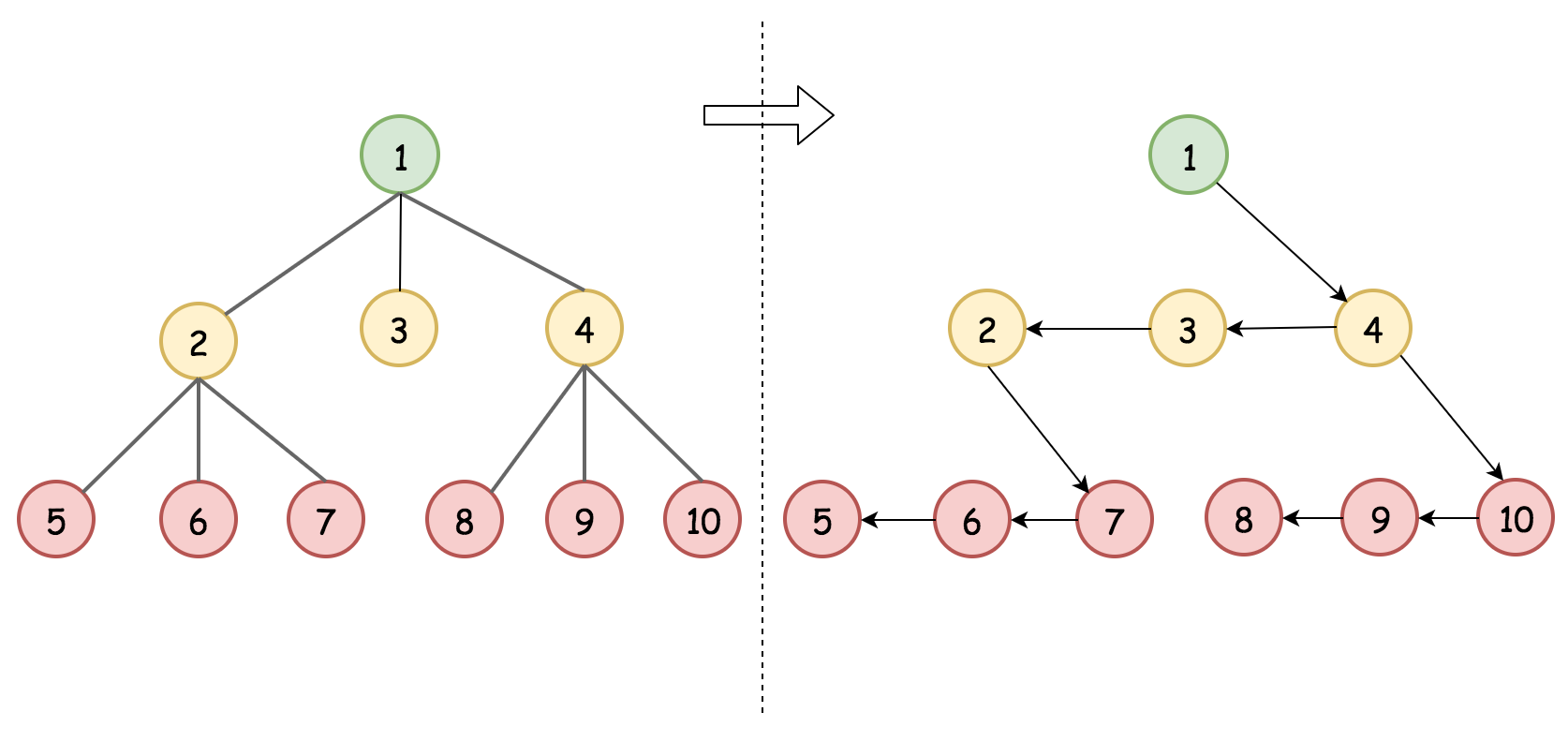


Before one notices, after the above two steps, we have already converted the N-ary tree to a binary tree !

It might not be evident from the above graph. But if one turns the graph 45 degrees clockwise, a binary tree would appear.



As one can imagine, based on the above idea, one can create some variants. For instance, instead of linking the child nodes with the right pointers, we could use the left pointers. And accordingly, we could start from the last child node to chain up the siblings. Here is the variant.



#### **Approach 1: BFS (Breadth-First Search) Traversal**

**Intuition**

There are generally two strategies to traverse the tree data structure: BFS (Breadth-First Search) and DFS (Depth-First Search).

Based on the intuition in the above section, one might find it fit well with the BFS strategy, since we are traversing the nodes level by level, i.e. we chain up the sibling nodes which reside in the same level of the tree. Indeed, we could implement the algorithm with the BFS strategy. But actually, as we would demonstrate later, we could also implement it via the DFS strategy.

**Algorithm**

Let us start with the BFS on the encode(root) function:

* Speaking about BFS, one shall recall that it is essentially implemented via the **queue** data structure. Indeed, first of all, all the sibling nodes would be pushed into the queue in sequence. And the one at the head of the queue would be processed first, which follows the principle of the queue data structure, **FIFO** (First In, First Out).
* The main body of the algorithm consists of a ***loop*** that iterates through the queue until it becomes empty. At each step of the loop, we pop out a node from the head of the queue, and process it.
* For the popped out node, we then run another **loop** over its children nodes. As one notices, this is a nested loop inside the previous loop over the queue. At each step of this nested loop, for each child node, we do two things:
  + First, we chain this child node with its previous neighbor sibling node.
  + Second, we append this child node into the queue, in order that it would have its turn to be processed as a parent node to encode its own children nodes.
* Voila. That is it. An important note is that we do the traversing of the N-ary tree in parallel with the construction of the desired Binary Tree. As a result, we keep each entry in the queue as a **tandem**, i.e. pair(n-ary\_tree\_node, binary\_tree\_node).
* To render the algorithm more robust, we could handle the case where the input N-ary tree is empty at the beginning of the function.

1 / 9

|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public List<Node> children;  public Node() {}  public Node(int \_val) {  val = \_val;  }  public Node(int \_val, List<Node> \_children) {  val = \_val;  children = \_children;  }  };  \*/  /\*  // Definition for a binary tree node.  public class TreeNode {  int val;  TreeNode left;  TreeNode right;  TreeNode(int x) { val = x; }  }  \*/  class Pair<U, V> {  public U first;  public V second;  public Pair(U first, V second) {  this.first = first;  this.second = second;  }  }  class Codec {  // Encodes an n-ary tree to a binary tree.  public TreeNode encode(Node root) {  if (root == null) {  return null;  }  TreeNode newRoot = new TreeNode(root.val);  Pair<TreeNode, Node> head = new Pair<TreeNode, Node>(newRoot, root);  // Add the first element to kickoff the loop  Queue<Pair<TreeNode, Node>> queue = new ArrayDeque<Pair<TreeNode, Node>>();  queue.add(head);  while (queue.size() > 0) {  Pair<TreeNode, Node> pair = queue.remove();  TreeNode bNode = pair.first;  Node nNode = pair.second;  // Encoding the children nodes into a list of TreeNode.  TreeNode prevBNode = null, headBNode = null;  for (Node nChild : nNode.children) {  TreeNode newBNode = new TreeNode(nChild.val);  if (prevBNode == null) {  headBNode = newBNode;  } else {  prevBNode.right = newBNode;  }  prevBNode = newBNode;  Pair<TreeNode, Node> nextEntry = new Pair<TreeNode, Node>(newBNode, nChild);  queue.add(nextEntry);  }  // Attach the list of children to the left node.  bNode.left = headBNode;  }  return newRoot;  }  // Decodes your binary tree to an n-ary tree.  public Node decode(TreeNode root) {  if (root == null) {  return null;  }  Node newRoot = new Node(root.val, new ArrayList<Node>());  // adding the first element to kickoff the loop  Queue<Pair<Node, TreeNode>> queue = new ArrayDeque<Pair<Node, TreeNode>>();  Pair<Node, TreeNode> head = new Pair<Node, TreeNode>(newRoot, root);  queue.add(head);  while (queue.size() > 0) {  Pair<Node, TreeNode> entry = queue.remove();  Node nNode = entry.first;  TreeNode bNode = entry.second;  // Decoding the children list  TreeNode firstChild = bNode.left;  TreeNode sibling = firstChild;  while (sibling != null) {  Node nChild = new Node(sibling.val, new ArrayList<Node>());  nNode.children.add(nChild);  // prepare the decoding the children of the child, by standing in the queue.  Pair<Node, TreeNode> nextEntry = new Pair<Node, TreeNode>(nChild, sibling);  queue.add(nextEntry);  sibling = sibling.right;  }  }  return newRoot;  }  }  // Your Codec object will be instantiated and called as such:  // Codec codec = new Codec();  // codec.decode(codec.encode(root)); |

As to the decode(node) function, similarly with our encoding function, we could implement in the BFS manner.

* Again, the main algorithm is organized as a loop around a queue data structure.
* We start from the root node of the encoded binary tree by pushing it into the queue.
* At each step of the iteration, we pop out a binary node from the tree, we then take the left child node of the node as its corresponding first child node of the original N-ary node.
* We then recover the rest of the child nodes by following the right pointer of the binary nodes.

**Complexity Analysis**

* Time Complexity: \mathcal{O}(N)O(*N*) where N*N* is the number of nodes in the N-ary tree. We traverse each node in the tree once and only once.
* Space Complexity: \mathcal{O}(L)O(*L*) where L*L* is the maximum number of nodes that reside at the same level. Since L*L* is proportional to N*N* in the worst case, we could further generalize the time complexity to \mathcal{O}(N)O(*N*).
  + We use a queue data structure to do BFS traversal, i.e. visiting the nodes level by level.
  + **At any given moment, the queue contains nodes that are at most spread into two levels**. As a result, assuming the maximum number of nodes at one level is L*L*, the size of the queue would be less than 2L2*L* at any time.
  + Therefore, the space complexity of both encode() and decode() functions is \mathcal{O}(L)O(*L*).

#### **Approach 2: DFS (Depth-First Search) Traversal**

**Intuition**

As it turned out, we could also implement the idea at the beginning of the article through DFS (Depth-First Search) traversal strategy.

Often the case, we implement the DFS algorithm with the technique of **recursion** which could greatly simplify the logic. Instead of ironing out all iterative steps, we could implement the function with the help of the function itself.

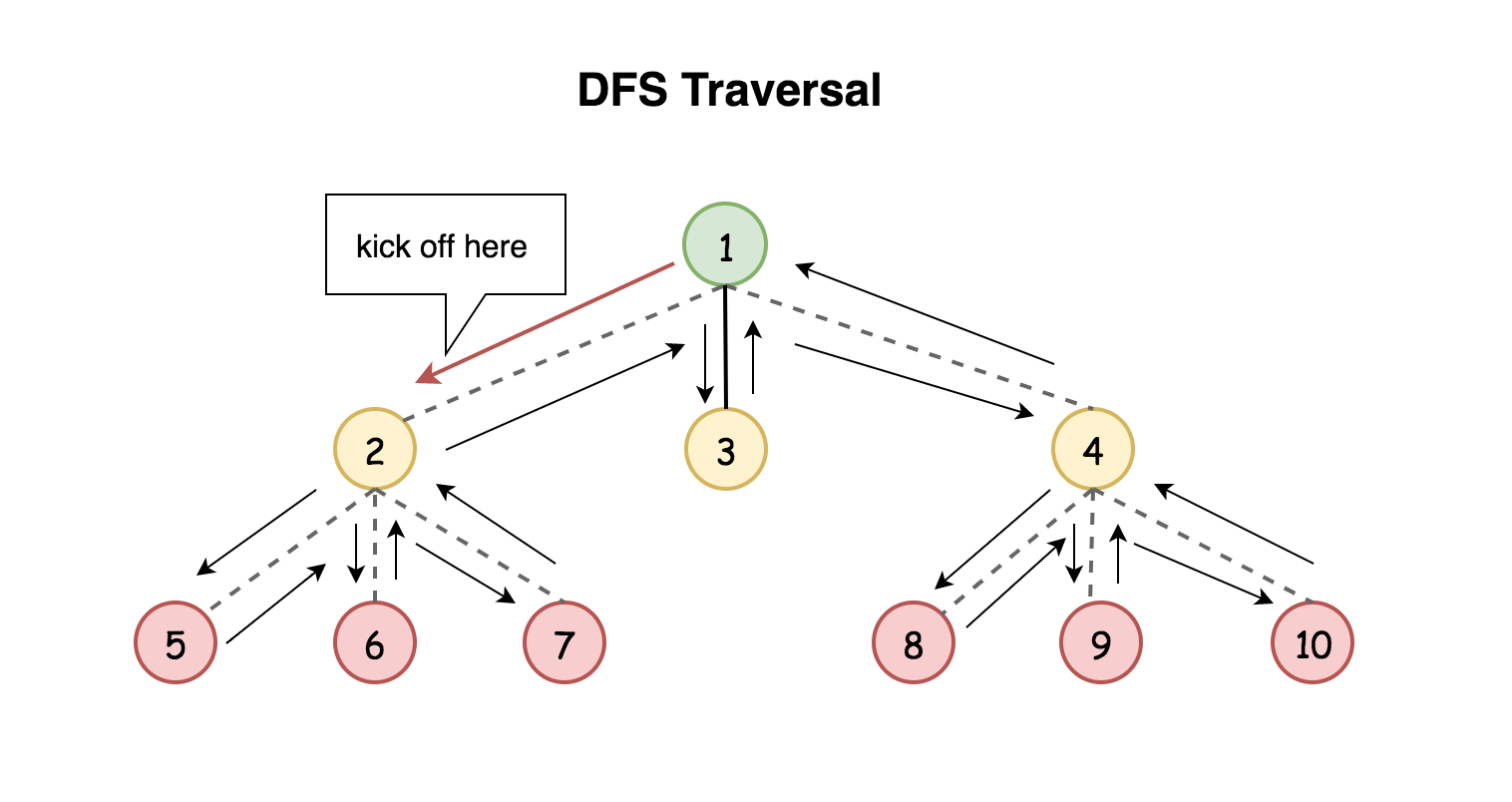
The idea is that while we traverse the N-ary tree node by node in the DFS manner, we **weave** the nodes together into a Binary tree, following the same intuition of encoding in the previous approach.

**Algorithm**

Again, let's demonstrate the encode(node) function as an example.

The main idea of the algorithm is that for each node, we only take care the encoding of the node itself, and we invoke the function itself to encode each of its child node, i.e. encode(node.children[i]).

* At the beginning of the encode(node) function, we create a binary tree node to contain the value of the current node.
* Then we put the first child of the N-ary tree node as the left node of the newly-created binary tree node. We call the encoding function recursively to encode the first child node as well.
* For the rest of the children nodes of the N-ary tree node, we chain them up with the right pointer of the binary tree node. And again, we call recursively the encoding function to encode each of the child node.



Note: the following implementation is inspired from the post by [wangzi6147](https://leetcode.com/problems/encode-n-ary-tree-to-binary-tree/discuss/153061/Java-Solution-(Next-Level-greater-left-Same-Level-greater-right)) in the discussion forum.

|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public List<Node> children;  public Node() {}  public Node(int \_val) {  val = \_val;  }  public Node(int \_val, List<Node> \_children) {  val = \_val;  children = \_children;  }  };  \*/  /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Codec {  // Encodes an n-ary tree to a binary tree.  public TreeNode encode(Node root) {  if (root == null) {  return null;  }  TreeNode newRoot = new TreeNode(root.val);  // Encode the first child of n-ary node to the left node of binary tree.  if (root.children.size() > 0) {  Node firstChild = root.children.get(0);  newRoot.left = this.encode(firstChild);  }  // Encoding the rest of the sibling nodes.  TreeNode sibling = newRoot.left;  for (int i = 1; i < root.children.size(); ++i) {  sibling.right = this.encode(root.children.get(i));  sibling = sibling.right;  }  return newRoot;  }  // Decodes your binary tree to an n-ary tree.  public Node decode(TreeNode root) {  if (root == null) {  return null;  }  Node newRoot = new Node(root.val, new ArrayList<Node>());  // Decoding all the children nodes  TreeNode sibling = root.left;  while (sibling != null) {  newRoot.children.add(this.decode(sibling));  sibling = sibling.right;  }  return newRoot;  }  }  // Your Codec object will be instantiated and called as such:  // Codec codec = new Codec();  // codec.decode(codec.encode(root)); |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(N)O(*N*) where N*N* is the number of nodes in the N-ary tree. We traverse each node in the tree once and only once.
* Space Complexity: \mathcal{O}(D)O(*D*) where D*D* is the depth of the N-ary tree. Since D*D* is proportional to N*N* in the worst case, we could further generalize the time complexity to \mathcal{O}(N)O(*N*).
  + Unlike the BFS algorithm, we don't use the queue data structure in the DFS algorithm. However, implicitly the algorithm would consume more space in the function call stack due to the recursive function calls.
  + And this consumption of call stack space is the main space complexity for our DFS algorithm. As we can see, the size of the call stack at any moment is exactly the number of ***level*** where the currently visited node resides, e.g. for the root node (level 0), the recursive call stack is empty.