**N-ary Tree**

Introduction

In the previous article, we focused more on binary tree. This card extends the concept you have learned in binary tree to n-ary tree.

By completing this card, you will:

1. Understand the definition of an n-ary tree.
2. Know the different traversal of n-ary trees.
3. Have basic knowledge on how to approach n-ary tree problems.

Traversal

 Traversal of N-ary Tree

 N-ary Tree Preorder Traversal

 N-ary Tree Preorder Traversal

 N-ary Tree Postorder Traversal

 N-ary Tree Postorder Traversal

 N-ary Tree Level Order Traversal

Recursion

 Classical Recursion Solution of N-ary Tree

 Maximum Depth of N-ary Tree

 Encode N-ary Tree to Binary Tree

 Solution: Encode N-ary Tree to Binary Tree

Conclusion

 Serialize and Deserialize N-ary Tree

**Traversal**

N-ary Tree Definition

A binary tree is a rooted tree in which each node has no more than 2 children. Let's extend this definition to N-ary tree. If a tree is a rooted tree in which each node has no more than N children, it is called N-ary tree.

Here is an example of 3-ary tree:

Application, background pattern, icon

Description automatically generated

Trie is one of the most frequently used N-ary trees.

Also, a binary tree is a special form of a N-ary tree. In the following sections, we will extend what we have learned about binary trees to N-ary trees.

**Traversal of N-ary Tree**

### **Tree Traversal**

A binary tree can be traversed in preorder, inorder, postorder or level-order. Among these traversal methods, preorder, postorder and level-order traversal are suitable to be extended to an N-ary tree.

Retrospect - Traverse a Binary Tree

1. Preorder Traversal: Visit the root node, then traverse the left subtree and finally traverse the right subtree.
2. Inorder Traversal: Traverse the left subtree, then visit the root node and finally traverse the right subtree.
3. Postorder Traversal: Traverse the left subtree, then traverse the right subtree and finally visit the root node.
4. Level-order Traversal: Traverse the tree level by level.

Note that here is no standard definition for in-order traversal in n-ary trees. It probably only makes sense for binary trees. While there are several different possible ways that one could define in-order traversal for n-ary trees, each of those feels a bit odd and unnatural and probably not terribly useful in practice.

To generalize the above to n-ary trees, you simply replace the steps:

Traverse the left subtree.... Traverse the right subtree....

in the above by:

For each child:  
      Traverse the subtree rooted at that child by recursively calling the traversal function

We assume that the for-loop will iterate through the children in the order they are found in the data-structure: typically, in left-to-right order, for a diagram such as below.

### **N-ary Tree Traversal Examples**

We will use the following 3-ary tree as example:

Application, background pattern, icon

Description automatically generated

#### **1. Preorder Traversal**

In an N-ary tree, preorder means visit the root node first and then traverse the subtree rooted at its children one by one. For instance, the preorder of the 3-ary tree above is: A->B->C->E->F->D->G.

#### **2. Postorder Traversal**

In an N-ary tree, postorder means traverse the subtree rooted at its children first and then visit the root node itself. For instance, the postorder of the 3-ary tree above is: B->E->F->C->G->D->A.

#### **3. Level-order Traversal**

Level-order traversal in an N-ary tree is the same with a binary tree. Typically, when we do breadth-first search in a tree, we will traverse the tree in level order. For instance, the level-order of the 3-ary tree above is: A->B->C->D->E->F->G.

### **Exercises**

Up next, we have few exercises for you to practice N-ary tree traversals.

**N-ary Tree Preorder Traversal**

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:**

Shape, arrow

Description automatically generated

**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 O(*N*), where *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 O(*N*).

**N-ary Tree Preorder Traversal**

## 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 O(*N*), where *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 O(*N*).

**N-ary Tree Postorder Traversal**

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:**

Shape, arrow

Description automatically generated

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

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

**Example 2:**

Shape

Description automatically generated

**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)

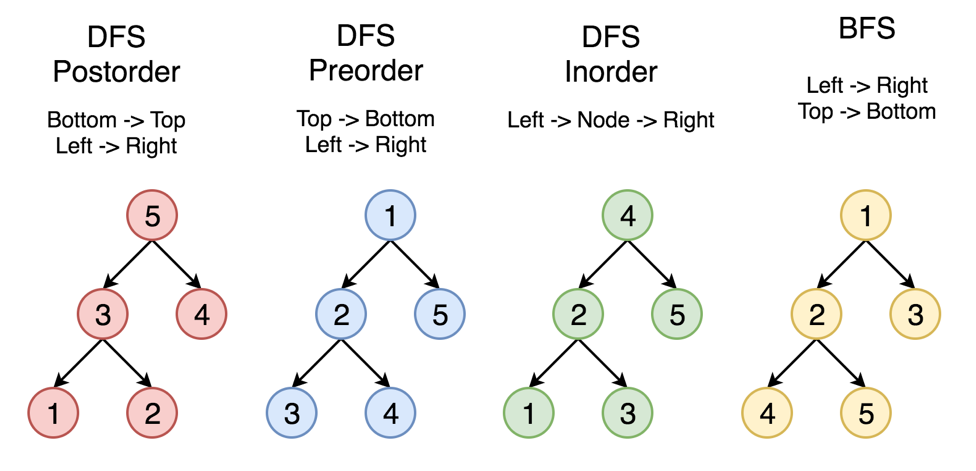
We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

* 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 O(*N*), where *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 O(*N*).

**N-ary Tree Postorder Traversal**

## 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)

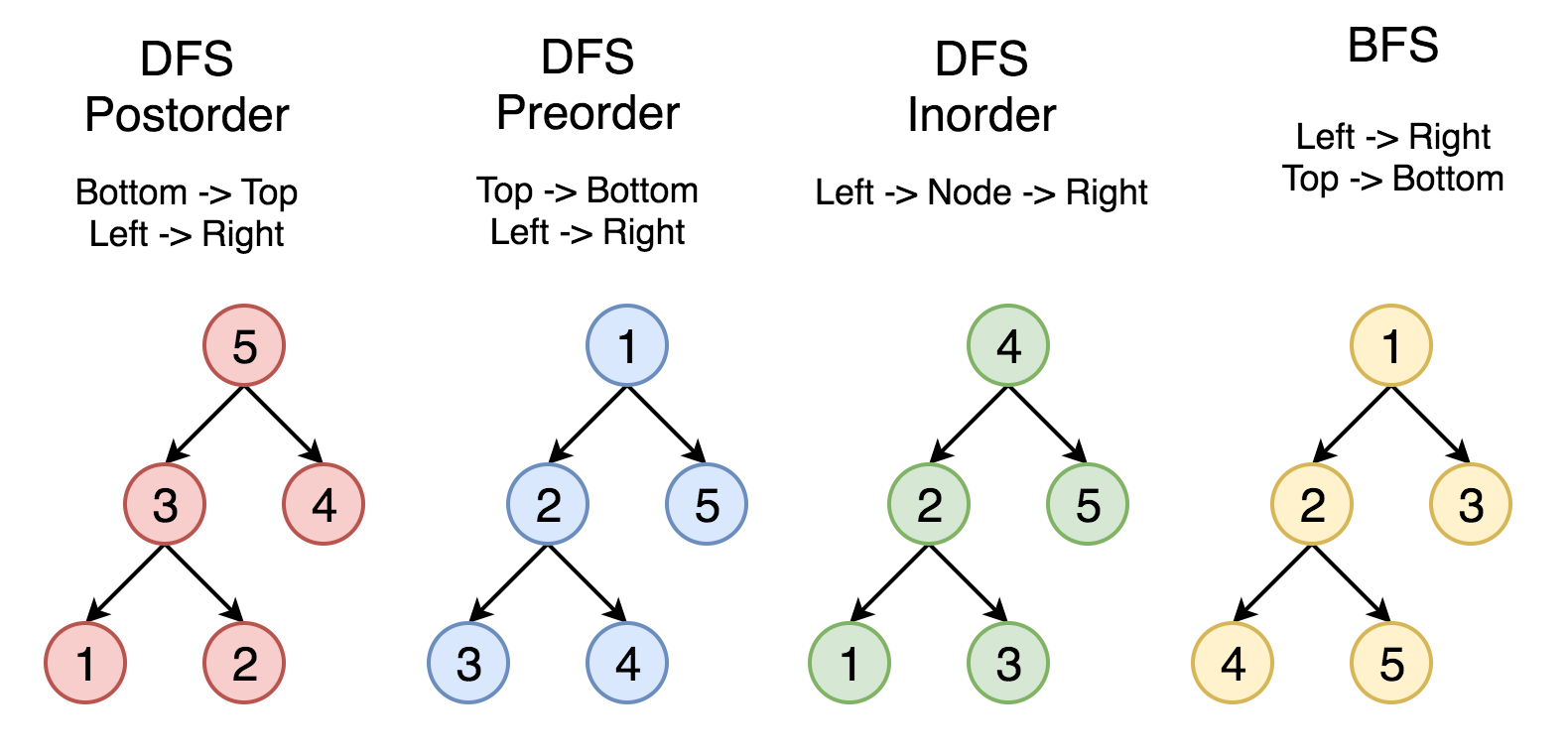
We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

* 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**

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 O(*N*), where *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 O(*N*).

**N-ary Tree Level Order Traversal**

Given an n-ary tree, return the *level order* 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).

**Example 1:**

Shape, arrow

Description automatically generated

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

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

**Example 2:**

Shape

Description automatically generated

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

**Constraints:**

* The height of the n-ary tree is less than or equal to 1000
* The total number of nodes is between [0, 104]

## Solution

#### **Approach 1: Breadth-first Search using a Queue**

**Intuition**

We want to make a list of sub-lists, where each sub-list is the values from one row in the tree. The rows should be in the order they appear from top to bottom.

Because we're traversing the tree, starting with the nodes nearest to the root and then working our way down to the nodes furthest from the root, this is a type of **breadth-first search**. To do a breadth-first search, we use a **queue**. Recall that a queue is a data structure that we put items in one end, and take them out of the other. We call it First In, First Out (FIFO) because the first items that go in should be the first to come out. It works the same way as the queue you have to wait in to get into a busy stadium.

A **stack** would be the wrong data structure to use here. Stacks are used for **depth-first search** (there is a convoluted way you could do it, but it's not sensible).

Let's start by using the most basic of queue-based traversal algorithms on the tree to see what it does. This is a fundamental algorithm you should be aiming to memorize.

List<Integer> values = new ArrayList<>();

Queue<Node> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

Node nextNode = queue.remove();

values.add(nextNode.val);

for (Node child : nextNode.children) {

queue.add(child);

}

}

Make a list to put integers in, and a queue to put nodes on. Put the root node onto the queue, and then while the queue is not empty, take a node off the queue, add its value to the list, and add each of its children onto the queue. Note that we are putting **Node** objects onto the queue, not integers. If we were to only put the integer values, we'd have no way of getting the child nodes out of them.

Let's see what we get when we use this algorithm to traverse the tree (don't worry about the inner lists yet, we're ensuring we have a way of getting the nodes from left to right and then top to bottom).

Diagram

Description automatically generated

Diagram

Description automatically generated with medium confidence

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated with medium confidence

Bubble chart

Description automatically generated with medium confidence

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated with medium confidence

Diagram

Description automatically generated with medium confidence

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated with low confidence

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Chart, diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Indeed, it does return the nodes from left to right, then top to bottom. Next we'll be looking at how we can take this basic algorithm and modify it so that we have each level in its own sub-list. Have a go at modifying the above algorithm on your own first.

**Algorithm**

The basic breadth-first search algorithm above got us part of the way, but we still need to do those sub-lists, and also make sure our code works if the root is null (a tree with no data).

We need to create a new sub-list each time we're starting a new layer, and we need to insert all nodes from the layer into that sub-list. A good way we can do this is by checking the current size of the queue at the start of the while loop body. Then, we can have another loop that processes that number of nodes. This way, we are guaranteed to be processing *one layer* for *each* iteration of the while loop so can put all nodes within the same iteration into the same sub-list. On the first iteration of the while loop, we only have **1** node: the root node. So we'll loop around the inner loop once, removing the root node, and put all of its children onto the queue. Then in the second iteration, we'll remove all the children from the queue (as that's the number of times we'll loop around the inner loop) and put all the grandchildren onto the queue. And so forth.

It's very important to use a **Queue** type for nodesToExplore, and *not* a Vector, List, or Array. Removing items off the front of those other data structures is an *O*(*n*) operation because all the remaining elements are moved along to fill the gap. A Queue is designed so that it is *O*(1).

|  |
| --- |
| // This code is a modified version of the code posted by  // #zzzliu on the discussion forums.  class Solution {  public List<List<Integer>> levelOrder(Node root) {  List<List<Integer>> result = new ArrayList<>();  if (root == null) return result;  Queue<Node> queue = new LinkedList<>();  queue.add(root);  while (!queue.isEmpty()) {  List<Integer> level = new ArrayList<>();  int size = queue.size();  for (int i = 0; i < size; i++) {  Node node = queue.poll();  level.add(node.val);  queue.addAll(node.children);  }  result.add(level);  }  return result;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*), where *n* is the number of nodes.

Each node is getting added to the queue, removed from the queue, and added to the result exactly once.

* Space complexity : *O*(*n*).

We are using a queue to keep track of nodes we still need to visit the children of. At most, the queue will have 2 layers of the tree on it at any given time. In the worst case, this is all of the nodes. In the best case, it is just 1 node (if we have a tree that is equivalent to a linked list). The average case is difficult to calculate without knowing something of the trees we can expect to see, but in balanced trees, half or more of the nodes are often in the lowest 2 layers. So we should go with the worst case of *O*(*n*), and know that the average case is probably similar.

#### **Approach 2: Simplified Breadth-first Search**

**Intuition**

A variant of the above approach is to make a new list on each iteration instead of using a single queue. This makes the code slightly simpler because we lose the size variable and the counting loop, which are a potential source of off-by-one errors.

**Algorithm**

|  |
| --- |
| // This code is a modified version of the code posted by  // #zzzliu on the discussion forums.  class Solution {  public List<List<Integer>> levelOrder(Node root) {  List<List<Integer>> result = new ArrayList<>();  if (root == null) return result;  List<Node> previousLayer = Arrays.asList(root);  while (!previousLayer.isEmpty()) {  List<Node> currentLayer = new ArrayList<>();  List<Integer> previousVals = new ArrayList<>();  for (Node node : previousLayer) {  previousVals.add(node.val);  currentLayer.addAll(node.children);  }  result.add(previousVals);  previousLayer = currentLayer;  }  return result;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*), where *n* is the number of nodes.
* Space complexity : *O*(*n*). Same as above, we always have lists containing levels of nodes.

#### **Approach 3: Recursion**

**Intuition**

We can use recursion for this problem. Often we can't use recursion for a breadth-first search (which is what a level-order traversal is). That is because breadth-first search is based on using a **queue**, whereas recursion is using the runtime **stack** and so is suited to depth-first search. In this case, however, we are putting all the values into a list before returning it. This means it's okay for us to get them in a different order to what they'll appear in the final list. As long as we know what level each node is from, and ensure they are in the correct order within each level, it will work.

**Algorithm**

|  |
| --- |
| class Solution {  private List<List<Integer>> result = new ArrayList<>();  public List<List<Integer>> levelOrder(Node root) {  if (root != null) traverseNode(root, 0);  return result;  }  private void traverseNode(Node node, int level) {  if (result.size() <= level) {  result.add(new ArrayList<>());  }  result.get(level).add(node.val);  for (Node child : node.children) {  traverseNode(child, level + 1);  }  }  } |

The iterative approach traversed the nodes in level order, whereas the recursive approach traversed them from left to right. While it was still easy to put them into the correct order using the recursive approach for this particular question, it could be problematic in practice. Often when we do a level-order traversal (or a breadth-first search), we are using the **Iterator** pattern and instead of storing the values in a list like we did here, the nodes are obtained one-by-one and processed. The iterator approach getting the nodes in the correct order will be much more useful for this use case. This is especially true with huge trees (e.g. links on a web page that you need to crawl and index).

Stack-based iterative approaches using a similar strategy to the recursion are possible too, however, they are not a good approach because this is supposed to be a breadth-first search problem (which uses queues) and they don't have the elegance of the recursive approach. If you used a stack for this question in an interview, you might leave your interviewer wondering why you felt the need to make the problem more difficult than it needed to be! For this reason, I've chosen not to include them in this article.

**Complexity Analysis**

* Time complexity : *O*(*n*), where *n* is the number of nodes.
* Space complexity : *O*(log*n*) average case and O(n) worst case. The space used is on the runtime stack.

## Recursion

I met a traveler from an antique land who said: "I met a traveler from an antique land, who said: "I met a traveler from an antique land, who said:" I met...

**Classical Recursion Solution of N-ary Tree**

Classical Recursion Solution

We talked about how to solve tree related problems recursively in previous article. In that article, we focused on binary trees but the idea can also be extended to a N-ary tree.

Make sure that you have read ([Solve Tree Problems Recursively](https://leetcode.com/explore/learn/card/data-structure-tree/17/solve-problems-recursively/534/)) before reading the following contents.

1. "Top-down" Solution

"Top-down" means that in each recursion level, we will visit the node first to come up with some values, and pass these values to its children when calling the function recursively.

A typical "top-down" recursion function top\_down(root, params) works like this:

1. return specific value for null node

2. update the answer if needed // answer <-- params

3. for each child node root.children[k]:

4. ans[k] = top\_down(root.children[k], new\_params[k]) // new\_params <-- root.val, params

5. return the answer if needed // answer <-- all ans[k]

2. "Bottom-up" Solution

"Bottom-up" means that in each recursion level, we will firstly call the functions recursively for all the children nodes and then come up with the answer according to the return values and the value of the root node itself.

A typical "bottom-up" recursion function bottom\_up(root) works like this:

1. return specific value for null node

2. for each child node root.children[k]:

3. ans[k] = bottom\_up(root.children[k]) // call function recursively for all children

4. return answer // answer <- root.val, all ans[k]

**Maximum Depth of N-ary Tree**

Given a n-ary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

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

**Example 1:**

Shape, arrow

Description automatically generated

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

**Output:** 3

**Example 2:**

Shape

Description automatically generated

**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:** 5

**Constraints:**

* The depth of the n-ary tree is less than or equal to 1000.
* The total number of nodes is between [0, 104].

## Solution

**Tree definition**

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

Here is the definition of the TreeNode which we would use.

|  |
| --- |
| // 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;  }  }; |

#### **Approach 1: Recursion**

The intuitive approach is to solve the problem by recursion. Here we demonstrate an example with the DFS (Depth First Search) strategy.

|  |
| --- |
| class Solution {  public int maxDepth(Node root) {  if (root == null) {  return 0;  } else if (root.children.isEmpty()) {  return 1;  } else {  List<Integer> heights = new LinkedList<>();  for (Node item : root.children) {  heights.add(maxDepth(item));  }  return Collections.max(heights) + 1;  }  }  } |

**Complexity analysis**

* Time complexity : we visit each node exactly once, thus the time complexity is O(*N*), where *N* is the number of nodes.
* Space complexity : in the worst case, the tree is completely unbalanced, e.g. each node has only one child node, the recursion call would occur *N* times (the height of the tree), therefore the storage to keep the call stack would be O(*N*). But in the best case (the tree is completely balanced), the height of the tree would be log(*N*). Therefore, the space complexity in this case would be O(log(*N*)).

#### **Approach 2: Iteration**

We could also convert the above recursion into iteration, with the help of stack.

The idea is to visit each node with the DFS strategy, while updating the max depth at each visit.

So we start from a stack which contains the root node and the corresponding depth which is 1. Then we proceed to the iterations: pop the current node out of the stack and push the child nodes. The depth is updated at each step.

|  |
| --- |
| class Solution {  public int maxDepth(Node root) {  Queue<Pair<Node, Integer>> stack = new LinkedList<>();  if (root != null) {  stack.add(new Pair(root, 1));  }  int depth = 0;  while (!stack.isEmpty()) {  Pair<Node, Integer> current = stack.poll();  root = current.getKey();  int current\_depth = current.getValue();  if (root != null) {  depth = Math.max(depth, current\_depth);  for (Node c : root.children) {  stack.add(new Pair(c, current\_depth + 1));  }  }  }  return depth;  }  } |

**Complexity analysis**

* Time complexity : O(*N*).
* Space complexity :  O(*N*).

**Encode N-ary Tree to Binary Tree**

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:

A picture containing watch

Description automatically generated

**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:**

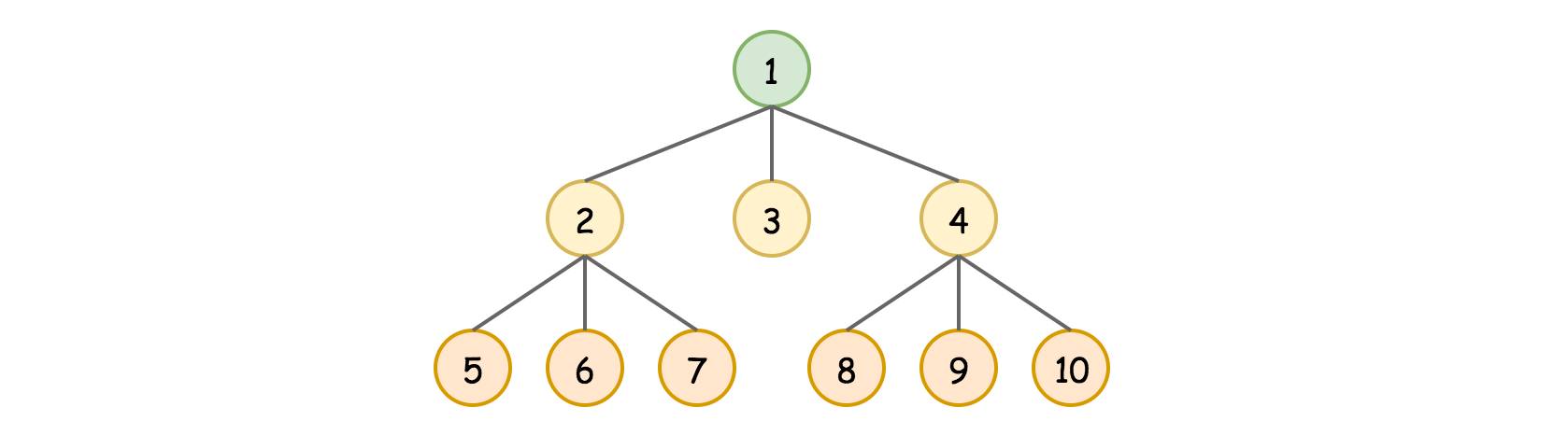
* The height of the n-ary tree is less than or equal to 1000
* The total number of nodes is between [0, 10^4]
* Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

## 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.

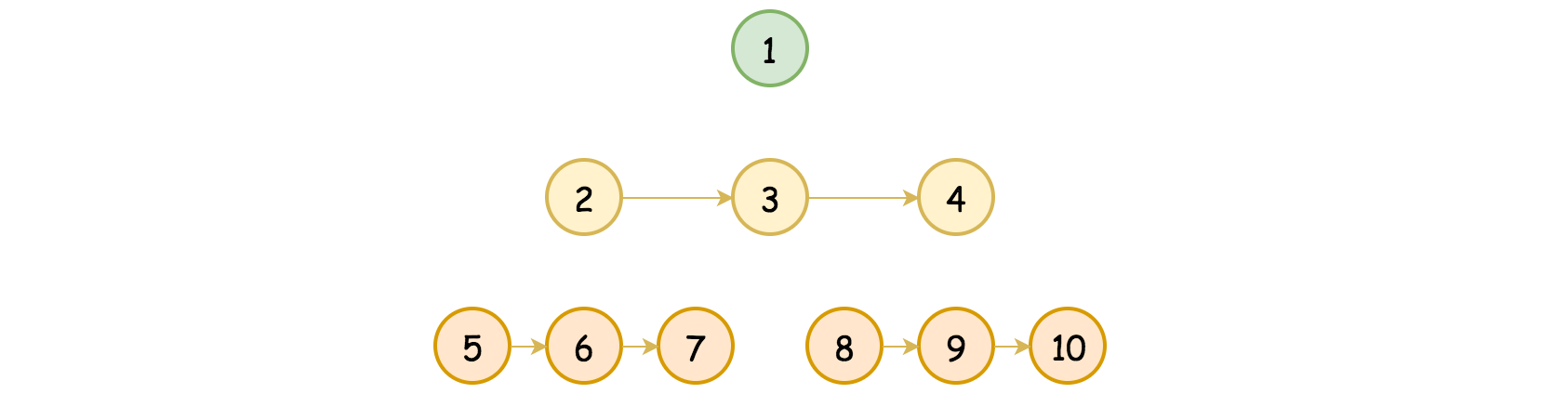


To put it simple, the algorithm can be summarized in two steps. We use the above N-ary tree as an example for demonstration.

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.

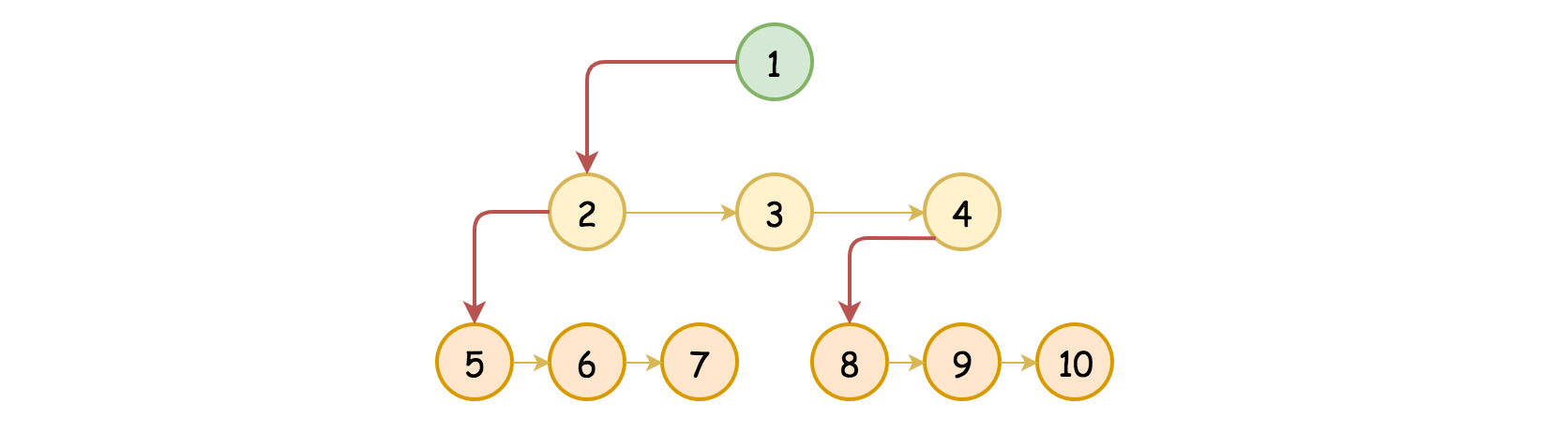
In the first step, we first chain up all the sibling nodes together, i.e. nodes that share the same parent. By chaining up, we would link the nodes via either left or right child pointers of the binary tree node. Here we choose to do with the right pointer.



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

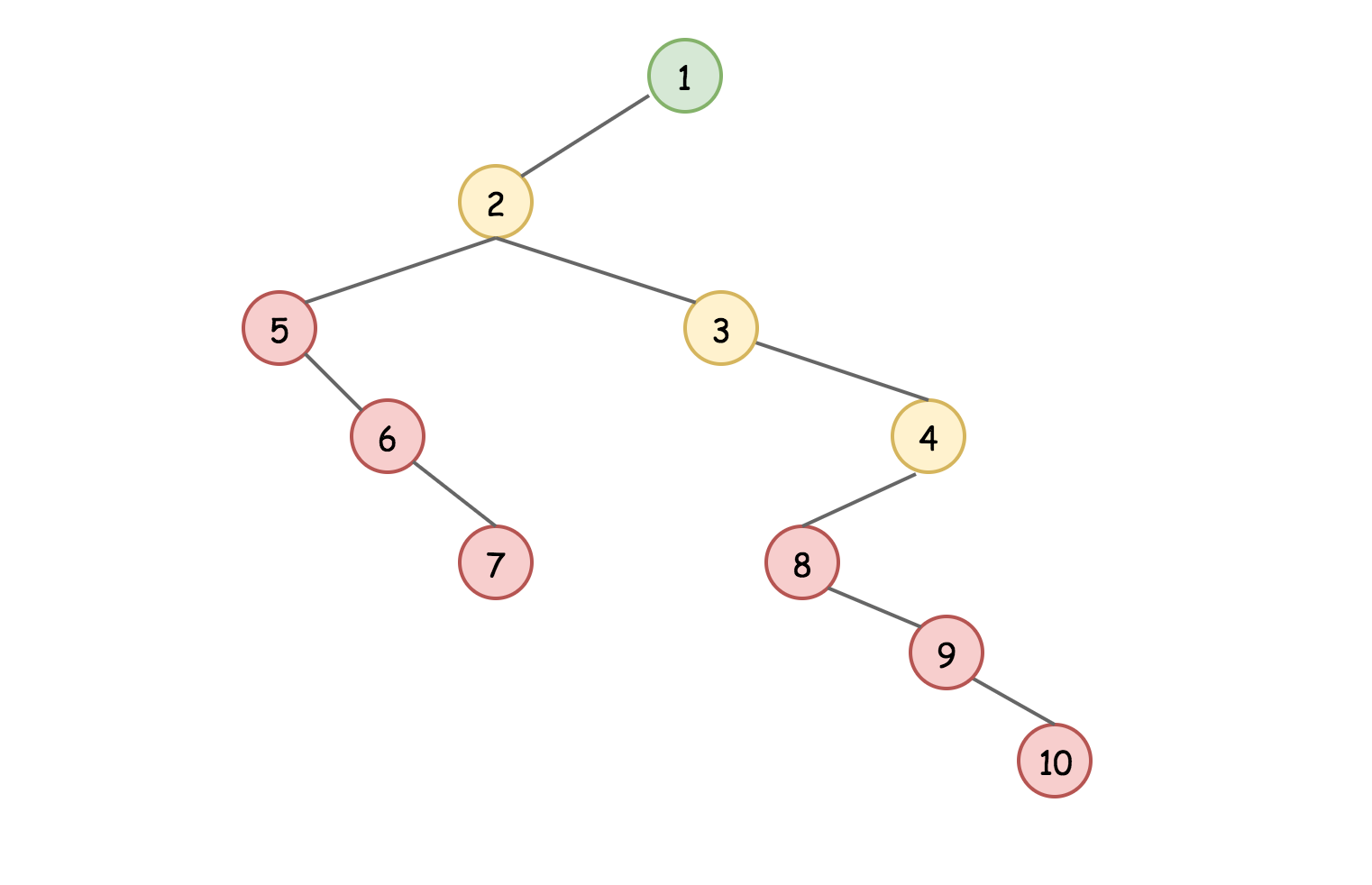
Now that the siblings are chained up, we just need to link this sibling list with their parent node.

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.

A picture containing diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated with medium confidence

Diagram

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing chart

Description automatically generated

|  |
| --- |
| /\*  // 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: O(*N*) where *N* is the number of nodes in the N-ary tree. We traverse each node in the tree once and only once.
* Space Complexity: O(*L*) where L*L* is the maximum number of nodes that reside at the same level. Since *L* is proportional to *N* in the worst case, we could further generalize the time complexity to 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*, the size of the queue would be less than 2*L* at any time.
  + Therefore, the space complexity of both encode() and decode() functions is 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.

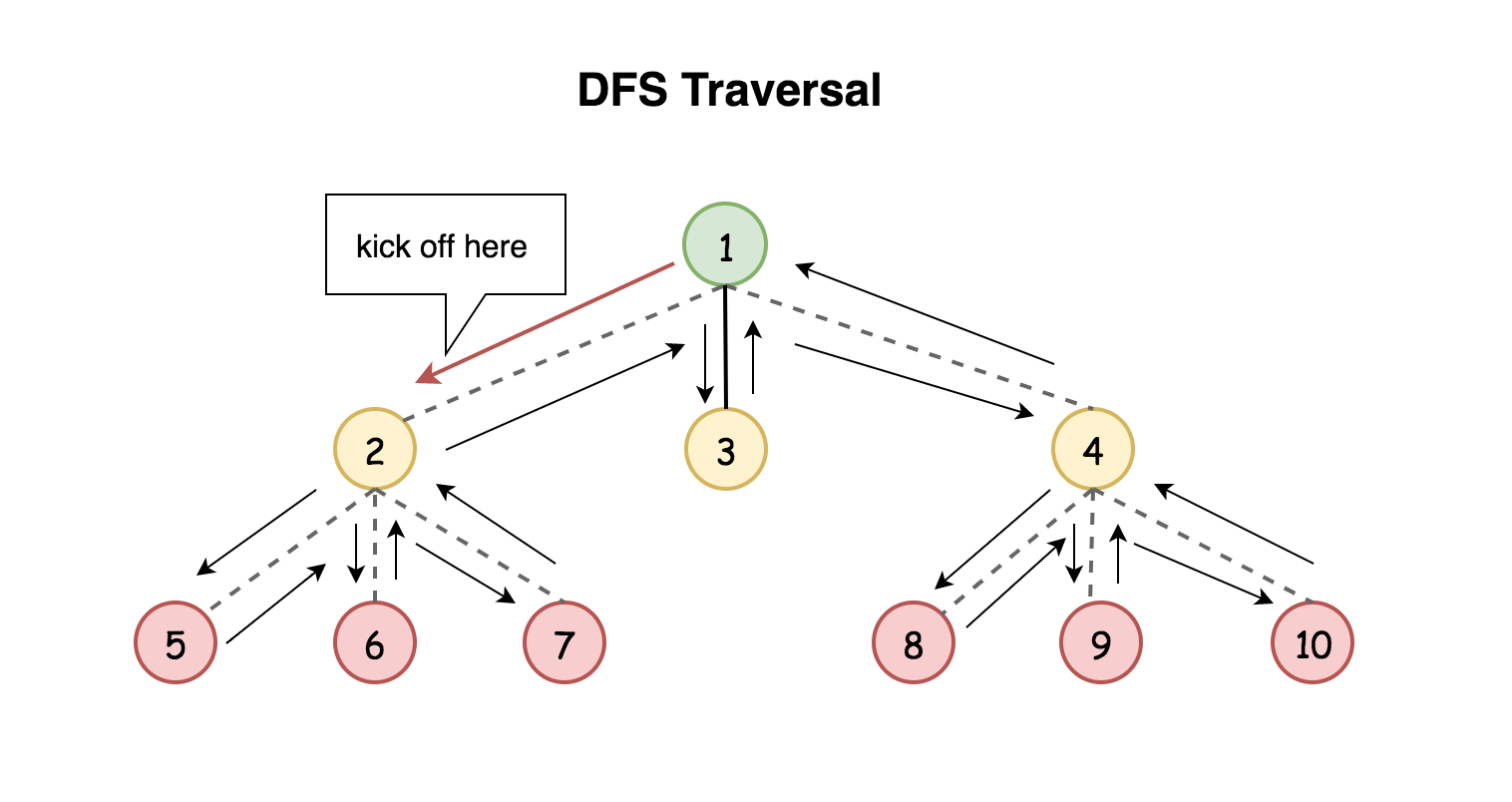
The idea is that while we traverse the N-ary tree node by node in the DFS manner, we **weave** the nodes together into a Binary tree, following the same intuition of encoding in the previous approach.

**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: O(*N*) where *N* is the number of nodes in the N-ary tree. We traverse each node in the tree once and only once.
* Space Complexity: O(*D*) where *D* is the depth of the N-ary tree. Since *D* is proportional to *N* in the worst case, we could further generalize the time complexity to 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.

**Solution: Encode N-ary Tree to Binary Tree**

There are a lot of solutions to convert a N-ary tree to a binary tree. We only introduce one classical solution.

The strategy follows two rules:

1. The left child of each node in the binary tree is the next sibling of the node in the N-ary tree.
2. The right child of each node in the binary tree is the first child of the node in the N-ary tree.

Here is an example to help you understand this strategy:

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Using this strategy, you can simply convert a N-ary tree to a binary tree recursively. Also, you can easily recover the N-ary tree from the binary tree you converted.

The recursion recovery strategy for each node is:

1. Deal with its children recursively.
2. Add its left child as the next child of its parent if it has a left child.
3. Add its right child as the first child of the node itself if it has a right child.

Here is an example to help you understand this strategy:

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Note that there are a lot of different solutions for this problem. Try experimenting in your own way!

## Conclusion

The goal of this card is to introduce the basic idea of an N-ary tree. Actually, a binary tree is just a special form of an N-ary tree and the solution for a problem related with an N-ary tree is quite similar with what we have done with a binary tree. Therefore, we can extend what we learned about a binary tree to an N-ary tree.

We also provide some classic N-ary Tree exercises for you to further understand N-ary trees in this chapter.

**Serialize and Deserialize N-ary Tree**

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize an N-ary tree. An N-ary tree is a rooted tree in which each node has no more than N children. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that an N-ary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following 3-ary tree

Shape, arrow

Description automatically generated

as [1 [3[5 6] 2 4]]. Note that this is just an example, you do not necessarily need to follow this format.

Or you can follow LeetCode's level order traversal serialization format, where each group of children is separated by the null value.

Shape

Description automatically generated

For example, the above tree may be serialized as [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].

You do not necessarily need to follow the above suggested formats, there are many more different formats that work so please be creative and come up with different approaches yourself.

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* 0 <= Node.val <= 104
* The height of the n-ary tree is less than or equal to 1000
* Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

## Solution

**Template**

This is one of the most interesting problems on the leetcode platform simply because there are a lot of different ways of solving this problem. There is no incorrect approach here. Some approaches are much easier to code-up and are more efficient as opposed to others. However, the variations for serialization and deserialization are endless. The following article is a collection of my own approaches for solving this problem and the solutions, learnings I got from some amazing posts in the discussion section.

Before we get on with the solutions themselves, we need to look at a basic template that shall be followed throughout this article. After all, the serialization method produces a string as an output and the deserialization method takes that string as input and reconstructs the tree. So, the general code template that we will be following is as follows:

|  |
| --- |
| class Codec {  // A wrapper class to pass the index in the data  // string by reference since the problem statement  // says that we are not allowed to use any globals or  // member variables to store the states. It should be  // stateless. Primitives are pass by value, so we create  // a wrapper object.  class WrappableInt {  private int value;  public WrappableInt(int x) {  this.value = x;  }  public int getValue() {  return this.value;  }  public void increment() {  this.value++;  }  }    // Encodes a tree to a single string.  public String serialize(Node root) {    StringBuilder sb = new StringBuilder();  this.\_serializeHelper(root, sb);  return sb.toString();  }    private void \_serializeHelper(Node root, StringBuilder sb) {  // To be written for every approach  }  // Decodes your encoded data to tree.  public Node deserialize(String data) {  if(data.isEmpty())  return null;    Node rootNode = new Node(data.charAt(0) - '0', new ArrayList<Node>());  WrappableInt index = new WrappableInt(1);  this.\_deserializeHelper(data, rootNode, index);  return rootNode;  }    private void \_deserializeHelper(String data, Node node, WrappableInt index) {    // To be written for every approach.  }  } |

There are some important things in this template which need to be addressed before moving on with the algorithms and their implementations.

**Wrappable Int**

First of all, we need to look at the nested class WrappableInt. If you read the problem constrains carefully, you'll see that we are asked to make our functions stateless.

Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

Some implementations out there actually split the input string and convert it into a queue. That seems rather unnecessary for this problem and is more time consuming. Instead, we can simply use an index to iterate over the data in the input string during deserialization and that would be much faster. So, the nested class is simply to provide us with an iterator during our recursive calls. Note that if your approach doesn't involve any kind of recursion, then we don't need any such custom object because we can simply iterate over the input string one character at a time in a for loop.

This custom class is used in the deserialization method. We need to process one character at a time and in case we are following a recursive approach, we need a variable that preserves the updates across function calls. As we all know, primitives are pass by value and in no circumstance will they maintain their values. So, we create a custom class which is a thin wrapper around an integer. Basically, for objects, the function calls are pass by value, but the object itself is not copied. Instead, a new reference of the object is created and passed around in function calls. As long as we don't assign this new reference to some other object, we should be good and the changes made within recursive calls shall be preserved. Again, there may be more elegant ways of achieving what is being done here, but this is an implementation detail and this is how I decided to implement the solutions.

**String Builder**

There are many ways of producing the serialized string from the data given to us. Not in terms of the algorithmic approach we take. But in terms of the way we choose to implement the same. Sure, we can go about the normal, easiest of operations which is string concatenation

serializedStr += data;

If you don't already know, then it's time to read more on this topic. Even though, the + method for string concatenation is one of the fastest out there (performance wise!), it does consume a shit ton of memory especially if there are a lot of such concatenations. This is mostly because parts of the resulting string are copied multiple times. Indeed, on every + operator, String class allocates a new block in memory and copies everything it has into it; plus a suffix being concatenated.

Hence, to save up on precious memory, we will be avoiding this approach. The next interesting approach is to use a dynamic list of strings (it's actually characters but we'll get to that in a bit) and finally, use string join method to stitch all the data together. Even though, these days, a + operation and a join operation are both super optimized by the Java compiler internally, still, we may not have similar optimizations in other languages.

In any case, building a list of strings and then joining them would consume much less memory than the string concatenation using the + operator. The final method, which is only applicable for Java is by using a StringBuilder. It is said that if we have an array of strings already built, then a join operation may be faster than a StringBuilder. However, for our implementation, we can use the StringBuilder on the fly and that turns out giving us the best performance according to leetcode stats. Unfortunately, we be using the join method in Python since we don't have an equivalent of StringBuilder there.

**You mentioned a list of characters? How so?**

Yeah, so this is a trick I learned in one of the fastest of Java solutions. Unfortunately, I don't have a profile to tag it with for credit here. The data provided to us in the nodes of the tree are integers. Sure, we can represent each integer as a string of digits. However, if we do that, then we would need some sort of a delimiter to separate the numbers themselves. After all 1234 could be 12 and 34 or it could be 1 and 234. Without a delimiter to separate them, the deserializer won't know.

If we do add a delimiter, it would add to the length of the overall string, which is fine. However, in the deserializer then, we would have to use the split operation and form a list of strings (more like a queue since that is how we will process them) and that is a relatively costly operation in terms of time and not to mention the extra space that the list would use.

Instead, we can use this neat trick which is to represent each number as a unicode character.

Of course there are limitations to this approach:

* Won't work on negative numbers
* Won't work if the numbers are > 65536

So, it's not really something that we can rely on 100% for correctness. It just so happens that for the test cases in this problem, this trick works perfectly fine and it is something to remember for solving other programming problems as well.

Essentially, we can represent each number as a unicode character. In Java, an integer is essentially the same as a unicode character and all we need is an explicit typecast and we'll be good to go. In Python we have a special function called unichr() which does this job. Starting Python 3.6, the more commonly known function chr() does the task. And for converting the character back to an integer, we can use ord() in Python and a simple implicit typecast does the trick in Java. This implementation has many advantages as you can think of:

* We don't need to use any special delimiters just for separating numbers.
* We don't need costly split operations and instead, we can iterate on the input string one character at a time and form our tree.
* It's blazingly fast! On one of the solutions, I saw the run time come down to 2ms down from a whooping 10ms in Java. That's a 5X jump and definitely worthy of note.

Now that we have the basics out of the way, let's finally get on with our algorithms themselves for serializing and deserializing an n-ary tree. It goes without saying that there may be a lot of different approaches out of there which are surely not explored in this article and if you feel you have a great new take on this problem, do let us know in the comments section and we'd love to enhance the article! That being said, let's get on.

#### **Approach 1: Parent Child relationships**

**Intuition**

The intuition for this approach is pretty straightforward. The serialized string would contain the parent child information for each of the nodes in the tree and we will use that to reconstruct the tree. Given a serialized string we will construct a hash map with a node being the key and the value being it's parent. Since we will have all the parent child relationships, we can just keep creating nodes as required and update the children array. Let's see how the serialized string would look like for a given tree and the corresponding hash map that would be created using the serialized string.

Diagram

Description automatically generated

As mentioned in the image, with this simple serialization, we will run into problems since there can be nodes with duplicate data and we can't rely on just the values for deserialization. We need a way of differentiating different nodes. For that, we will be using a unique identifier for each node. Again, the WrappableInt data structure will come in handy here in the implementation. Let's see what the serialized string looks like with these unique Ids.

Diagram, schematic

Description automatically generated

Note that the Ids assigned in the above example make sense if we do a level order traversal in the code. If we do a depth first traversal, then the order of processing the nodes would change and so would the Ids and the final serialized string. Note that even though we can process the serialized string in any order and recreate the original tree, we have to stick to the inherent ordering defined in the string itself.

The important thing is that we need to maintain not only the correct children's list but also the correct ordering of the children. So, for the above example, the children's list for the root node cannot be [5, 3, 3, 7]. It has to be [3, 5, 3, 7].

Now let's look at the hash map as formed using the serialized string containing the unique Ids.

Diagram

Description automatically generated

**Algorithm**

Serialization

1. We'll do a simple depth first traversal of the tree starting from the root node and the StringBuilder (list in case of Python).
2. The helper function would take one WrappableInt as an input in addition to the node itself. The custom integer would represent the unique Id of the current node. As for the parent node, we pass a simple Integer object since we don't want retention for parentIds across recursion.
3. For every node, we will add 3 values to the serialized string. The first would be the unique Id of the current node. Next we add the actual value of the node and finally, we add the unique Id of the parent node.
4. Remember to use the unicode character trick discussed in the introduction section of this article. We will be using it heavily to keep down the overall length of the serialized string.
5. For the root node, we will be using a special dummy value N. We can use a negative value as well since the test cases don't have any negative value. However, for achieving as much generalization as possible, let's use a dummy character.

Deserialization

1. For deserialization, we are given the string as an input. We will always be processing the input in triplets since 3 characters represent the information for one node.
2. We will initialize a HashMap that will contain the data from the string. It's the hash map from the figures before.
3. For every triplet in the input string (a, b, c), we will create a new entry in the hash map with a being the key and a pair of b, c being the value. Remember, a represents the unique Id for the node, b represents its actual value and c represents the Id of the parent node. Also, in addition to the 2 values b, c, we will also be adding new TreeNode or Node data structures to the dictionary. This is because we will be re-using this dictionary to fill up the children lists for each node. So the actual entry in the hash map would be

a -> (b, c, Node(a, []))

1. Once we are done constructing the dictionary, we have to construct the original tree. We have already constructed all the nodes of the tree. All that remains is establishing the right connections in the right order. Remember when we mentioned about the ordering of the children nodes, we have to ensure we don't mess that up here.
2. We can't process nodes in any random order. So, we use the original string itself and use every third entry as the node to process.

|  |
| --- |
| class Codec {  class WrappableInt {  private Integer value;  public WrappableInt(Integer x) {  this.value = x;  }  public Integer getValue() {  return this.value;  }  public void increment() {  this.value++;  }  }    // Was searching for typedef alternatives in Java and came across fake classes  // Mostly considered an anti-pattern but it definitely makes our code much more  // readable!  class DeserializedObject extends HashMap<Integer, Pair<Integer, Pair<Integer, Node>>> {}      // Encodes a tree to a single string.  public String serialize(Node root) {    StringBuilder sb = new StringBuilder();  this.\_serializeHelper(root, sb, new WrappableInt(1), null);  return sb.toString();  }    private void \_serializeHelper(Node root, StringBuilder sb, WrappableInt identity, Integer parentId) {    if (root == null) {  return;  }    // Own identity  sb.append((char) (identity.getValue() + '0'));    // Actual value  sb.append((char) (root.val + '0'));    // Parent's identity  sb.append((char) (parentId == null ? 'N' : parentId + '0'));    parentId = identity.getValue();  for (Node child : root.children) {  identity.increment();  this.\_serializeHelper(child, sb, identity, parentId);  }  }  // Decodes your encoded data to tree.  public Node deserialize(String data) {  if(data.isEmpty())  return null;    return this.\_deserializeHelper(data);  }    private Node \_deserializeHelper(String data) {    // HashMap explained in the algorithm  DeserializedObject nodesAndParents = new DeserializedObject();    // Constructing the hashmap using the input string  for (int i = 0; i < data.length(); i+=3) {  int id = data.charAt(i) - '0';  int orgValue = data.charAt(i + 1) - '0';  int parentId = data.charAt(i + 2) - '0';  Pair<Integer, Pair<Integer, Node>> node = new Pair<Integer, Pair<Integer, Node>>(orgValue,  new Pair<Integer, Node>(parentId,  new Node(orgValue, new ArrayList<Node>())));  nodesAndParents.put(id, node);  }    // A second pass for tying up the proper child connections  for (int i = 3; i < data.length(); i+=3) {    // Current node  int id = data.charAt(i) - '0';  Node node = nodesAndParents.get(id).getValue().getValue();    // Parent node  int parentId = data.charAt(i + 2) - '0';  Node parentNode = nodesAndParents.get(parentId).getValue().getValue();    // Attach!  parentNode.children.add(node);  }    // Return the root node.  return nodesAndParents.get(data.charAt(0) - '0').getValue().getValue();  }  } |

**Complexity Analysis**

Time Complexity

* Serialization: O(N)*O*(*N*) where N*N* are the number of nodes in the tree. For every node, we add 3 different values to the final string and every node is processed exactly once.
* Deserialization: Well technically, it is 3N3*N* for the first for loop and N*N* for the second one. However, constants are ignored in asymptotic complexity analysis. So, the overall time complexity for deserialization is O(N)*O*(*N*).

Space Complexity

* Serialization: The space occupied by the serialization helper function is through recursion stack and the final string that is produced. Usually, we don't take into consideration the space of the output. However, in this case, the output is something which is not fixed. For all we know, someone might be able to generate a string of size N/2. We don't know! So, the size of the final string is a part of the space complexity here. Overall, the space is 4N4*N* = O(N)*O*(*N*).
* Deserialization: The space occupied by the deserialization helper function is through the hash map. For each entry, we have 3 values. Thus, we can say the space is 3N3*N*. But again, the constants don't really matter in asymptotic complexity. So, the overall space is O(N)*O*(*N*).

Note that for this particular problem, the asymptotic time and space will remain the same across all the approaches. The only thing that will change are the constants and that does impact the runtime in a major way. So, we will be focusing on the constants rather than the final complexity in all these approaches.

#### **Approach 2: Depth First Search with Children Sizes!**

**Intuition**

That previous approach works well, however, the problem is that we end up generating a serialized string which is three times the size of the tree. The reason for that is, we need unique identities for every node since we have no way of differentiating them just on the basis of values. So in this approach we will be incorporating two things into the serialized string per node - it's value, and the number of children it has. Let's quickly look at what that looks like on a sample tree.

Diagram

Description automatically generated

The next part is, how do we use this information and rebuild the correct tree during deserialization. In this approach, the deserialization simply tries to run the same recursion that we did during serialization. Except, now instead of tree nodes, we simply have information from the input string. Let's look at the pseudocode of how this approach would work.

|  |
| --- |
| func deserialize(data, index)  {  if (index == data.length)  {  return null  }    node = Construct new node using value at data[index]  for i in range 0...data[index+1]  {  node.add(deserialize(data, index+2))  }  } |

In addition to this pseudocode, let's also look at a figure explaining this on the serialized string from above. It goes without saying that the index above is actually a WrappableInt since we need to to maintain where it has reached in the input string exactly, across recursions.

Diagram

Description automatically generated

**Algorithm**

Serialization

1. Like before, we will initialize a StringBuilder (or a list in Python). We don't need our custom integer wrapper here since we don't need any unique identities.
2. We will be doing a depth first traversal on the input tree.
3. For every node, we will add it's value and also the number of children it has, to the string.
4. Again, we will be using the unicode character trick as explained in the introduction of the article. That will come in handy as we try to bound the size of the output string by some constant factor of the number of nodes in the tree.

Deserialization

1. The deserialization is simple as well. We simply need to rebuild the tree using the same recursion as we used in the serialization function.
2. We will need our WrappableInt index here since we need to keep track of what characters have already been processed in the string.
3. For a given index, i, we will create a new Node using data[i] as the value where data represents the input string.
4. Next, we will have a loop equal to the number of children this node has. That is given by data[i+1]. Remember, for every node in the tree, we added two pieces of information in the string. One is its original value and the other is the number of children it has.
5. Within this loop, we will make further recursive calls, one for each child of the current node. The deserialization helper function will return a node which will be the root of a fully constructed tree.

|  |
| --- |
| class Codec {  class WrappableInt {  private int value;  public WrappableInt(int x) {  this.value = x;  }  public int getValue() {  return this.value;  }  public void increment() {  this.value++;  }  }    // Encodes a tree to a single string.  public String serialize(Node root) {    StringBuilder sb = new StringBuilder();  this.\_serializeHelper(root, sb);  return sb.toString();  }    private void \_serializeHelper(Node root, StringBuilder sb) {    if (root == null) {  return;  }    // Add the value of the node  sb.append((char) (root.val + '0'));    // Add the number of children  sb.append((char) (root.children.size() + '0'));    // Recurse on the subtrees and build the  // string accordingly  for (Node child : root.children) {  this.\_serializeHelper(child, sb);  }  }  // Decodes your encoded data to tree.  public Node deserialize(String data) {  if(data.isEmpty())  return null;    return this.\_deserializeHelper(data, new WrappableInt(0));  }    private Node \_deserializeHelper(String data, WrappableInt index) {    if (index.getValue() == data.length()) {  return null;  }    // The invariant here is that the "index" always  // points to a node and the value next to it  // represents the number of children it has.  Node node = new Node(data.charAt(index.getValue()) - '0', new ArrayList<Node>());  index.increment();  int numChildren = data.charAt(index.getValue()) - '0';  for (int i = 0; i < numChildren; i++) {  index.increment();  node.children.add(this.\_deserializeHelper(data, index));  }    return node;  }  } |

**Complexity Analysis**

Time Complexity

* Serialization: *O*(*N*) where *N* are the number of nodes in the tree. For every node, we add 2 different values to the final string and every node is processed exactly once.
* Deserialization: For deserialization, we process the entire string, one character at a time and also construct the tree along the way. So, the overall time complexity for deserialization is 2*N* = *O*(*N*)

Space Complexity

* Serialization: The space occupied by the serialization helper function is through recursion stack and the final string that is produced. We know the size of the final string to be 2*N*. So, that is one part of the space complexity. The other part is the one occupied by the recursion stack which is *O*(*N*). Overall, the space is *O*(*N*).
* Deserialization: For deserialization, the space occupied is by the recursion stack only. We don't use any other intermediate data structures like we did in the previous approach and simply rely on the information in the string and recursion to work it's magic. So, the space complexity would be *O*(*N*) since this is not a balanced tree of any sort. It's not even binary.

This is one of the simplest algorithms for solving this problem. The serialization and deserialization have a very similar format and the overall space and time complexity are also very low. Also, what's nice is that it's easy to code up quickly in an interview!

#### **Approach 3: Depth First Search with a Sentinel**

**Intuition**

This approach is very similar to the previous approach. The only difference is that instead of adding the number of children a node has, to the serialized string, we add a sentinel value when all the children have been added to the final string. Let's look at the serialized string for the sample tree we've been looking at throughout the article.

Diagram

Description automatically generated

The next part is, how do we use this information and rebuild the correct tree during deserialization. In this approach, the deserialization simply tries to run the same recursion that we did during serialization. Except that now, instead of tree nodes, we simply have information from the input string. Let's look at the pseudocode of how this approach would work.

|  |
| --- |
| func deserialize(data, index)  {  if (index == data.length)  {  return null  }    node = Construct new node using value at data[index]  while data[index] != '#'  {  node.add(deserialize(data, index+1))  }    index++  } |

Here we need to move along the input string in accordance with the recursion from serialization before. Also, once we encounter the corresponding sentinel, we discard it. Just like during serialization we added a sentinel value after all the child nodes had been processed, similarly, we will encounter and discard the sentinel once all the child subtrees have been built completely. This is ensured by the recursion we write.

|  |
| --- |
| class Codec {  class WrappableInt {  private int value;  public WrappableInt(int x) {  this.value = x;  }  public int getValue() {  return this.value;  }  public void increment() {  this.value++;  }  }    // Encodes a tree to a single string.  public String serialize(Node root) {    StringBuilder sb = new StringBuilder();  this.\_serializeHelper(root, sb);  return sb.toString();  }    private void \_serializeHelper(Node root, StringBuilder sb) {    if (root == null) {  return;  }    // Add the value of the node  sb.append((char) (root.val + '0'));    // Recurse on the subtrees and build the  // string accordingly  for (Node child : root.children) {  this.\_serializeHelper(child, sb);  }    // Add the sentinel to indicate that all the children  // for the current node have been processed  sb.append('#');  }  // Decodes your encoded data to tree.  public Node deserialize(String data) {  if(data.isEmpty())  return null;    return this.\_deserializeHelper(data, new WrappableInt(0));  }    private Node \_deserializeHelper(String data, WrappableInt index) {    if (index.getValue() == data.length()) {  return null;  }    Node node = new Node(data.charAt(index.getValue()) - '0', new ArrayList<Node>());  index.increment();  while (data.charAt(index.getValue()) != '#') {  node.children.add(this.\_deserializeHelper(data, index));  }    // Discard the sentinel. Note that this also moves us  // forward in the input string. So, we don't have the index  // progressing inside the above while loop!  index.increment();    return node;  }  } |

**Complexity Analysis**

Time Complexity

* Serialization: *O*(*N*) where *N* are the number of nodes in the tree. For every node, we add 2 different values to the final string and every node is processed exactly once.
* Deserialization: For deserialization, we process the entire string, one character at a time and also construct the tree along the way. So, the overall time complexity for deserialization is 2*N* = *O*(*N*)

Space Complexity

* Serialization: The space occupied by the serialization helper function is through recursion stack and the final string that is produced. We know the size of the final string to be 2*N*. So, that is one part of the space complexity. The other part is the one occupied by the recursion stack which is *O*(*N*). Overall, the space is *O*(*N*).
* Deserialization: For deserialization, the space occupied is by the recursion stack only. We don't use any other intermediate data structures like we did in the previous approach and simply rely on the information in the string and recursion to work it's magic. So, the overall space complexity would be *O*(*N*).

#### **Approach 4: Level order traversal**

**Intuition**

This approach is based on the suggestion given by the problem statement itself. It's very similar to the strategy used by leetcode for serializing and deserializing a tree structure in problem statements. Essentially, we use level order traversal for serializing the tree and when deserializing, we construct one level at a time. The two main pieces of information that have to be infused somehow in the serialized string are:

* Which node has what children since a level can contain a lot nodes and we need to know the parent of each one of them.
* Second, and the more common information is the switch from one level to another. We need to add this information somehow in the string which helps the deserializer know that a level has finished and a new one has begun.

For the first piece, we will be using a sentinel value of $ and whenever we start adding children of a different node, we add this sentinel value to the string and then start adding the children. For the next piece of information, we add another sentinel \## to the string. Before we switch to the next level of the tree during serialization, we add this sentinel value to the string so that the deserializer knows that one level has ended and a new one has started. Let's look at what the serialized string looks like for the sample tree.

Diagram

Description automatically generated

We can get rid of the extra, unwanted string in the end. However, instead of doing another iteration and performing a substring operation or using some other tricky logic to not add that to the final string, we decided to handle it in the deserializer itself. Sure, if we can get rid of that extra portion, the string length would reduce.

**Algorithm**

Serialization

1. Since this is a level-order traversal, we will be making use of a queue here for traversing the tree one level at a time.
2. For Java, since our queue is composed of nodes, we cannot add the sentinel characters as is to the queue. So we create two dummy nodes and call them childNode and endNode. By using object comparison we know if a node is a sentinel node or a normal tree node.
3. We perform a normal level order traversal. The only change is that we add the childNode to the queue whenever we are done adding the children of a particular node to the queue.
4. Also, when a particular level ends, we add an endNode to the queue.
5. Only when we pop a sentinel node from the queue do we add the corresponding characters to the final serialized string.
6. As for the nodes in the tree, we use the unicode character trick we've been following all along.

Deserialization

1. For deserialization, we will go one level at a time for reconstructing the tree.
2. For this purpose, we maintain two lists currentLevel and prevLevel. The prevLevel contains the nodes from the previous level while we add the nodes on the current level to the corresponding list. Once we have these two lists figured out, we establish the corresponding connections.
3. The sentinel values come in handy since whenever we encounter a $ , the child switch sentinel, we pop a new parent node from prevLevel and any children encountered from this point to the next $ belong to this parent node.
4. Similarly, whenever we encounter the level end sentinel \##, we assign prevLevel to currentLevel since the nodes in the current level now become parents for the next level.

|  |
| --- |
| class Codec {  // Encodes a tree to a single string.  public String serialize(Node root) {    if (root == null) {  return "";  }    StringBuilder sb = new StringBuilder();  this.\_serializeHelper(root, sb);  return sb.toString();  }    private void \_serializeHelper(Node root, StringBuilder sb) {    // Queue to perform a level order traversal of the tree  Queue<Node> q = new LinkedList<Node>();    // Two dummy nodes that will help us in serialization string formation.  // We insert the "endNode" whenever a level ends and the "childNode"  // whenever a node's children are added to the queue and we are about  // to switch over to the next node.  Node endNode = new Node();  Node childNode = new Node();  q.add(root);  q.add(endNode);    while (!q.isEmpty()) {    // Pop a node  Node node = q.poll();    // If this is an "endNode", we need to add another one  // to mark the end of the current level unless this  // was the last level.  if (node == endNode) {    // We add a sentinal value of "#" here  sb.append('#');  if (!q.isEmpty()) {  q.add(endNode);  }  } else if (node == childNode) {    // Add a sentinal value of "$" here to mark the switch to a  // different parent.  sb.append('$');  } else {    // Add value of the current node and add all of it's  // children nodes to the queue. Note how we convert  // the integers to their corresponding ASCII counterparts.  sb.append((char) (node.val + '0'));  for (Node child : node.children) {  q.add(child);  }    // If this not is NOT the last one on the current level,  // add a childNode as well since we move on to processing  // the next node.  if (q.peek() != endNode) {  q.add(childNode);  }  }  }  }  // Decodes your encoded data to tree.  public Node deserialize(String data) {  if (data.isEmpty()) {  return null;  }    Node rootNode = new Node(data.charAt(0) - '0', new ArrayList<Node>());  this.\_deserializeHelper(data, rootNode);  return rootNode;  }    private void \_deserializeHelper(String data, Node rootNode) {    // We move one level at a time and at every level, we need access  // to the nodes on the previous level as well so that we can form  // the children arrays properly. Hence two arrays.  LinkedList<Node> currentLevel = new LinkedList<Node>();  LinkedList<Node> prevLevel = new LinkedList<Node>();  currentLevel.add(rootNode);  Node parentNode = rootNode;    // Process the characters in the string one at a time.  for (int i = 1; i < data.length(); i++) {  char d = data.charAt(i);  if (d == '#') {  // Special processing for end of level. We need to swap the  // array lists. Here, we simply re-initialize the "currentLevel"  // arraylist rather than clearing it.  prevLevel = currentLevel;  currentLevel = new LinkedList<Node>();    // Since we move one level down, we take the parent as the first  // node on the current level.  parentNode = prevLevel.poll();  } else {  if (d == '$') {    // Special handling for change in parent on the same level  parentNode = prevLevel.poll();  } else {  Node childNode = new Node(d - '0', new ArrayList<Node>());  currentLevel.add(childNode);  parentNode.children.add(childNode);  }  }  }  }  } |

**Complexity Analysis**

Time Complexity

* Serialization: *O*(*N*) where *N* are the number of nodes in the tree. For every node, we add 2 different values to the final string and every node is processed exactly once. We add the value of the node itself and we also add the child switch sentinel. Also, for the nodes that end a particular level, we add the level end sentinel.
* Deserialization: For deserialization, we process the entire string, one character at a time and also construct the tree along the way. So, the overall time complexity for deserialization is 2*N* = *O*(*N*)

Space Complexity

* Serialization: The space occupied by the serialization helper function is through the queue and the final string that is produced. We know the size of the final string to be 2*N*. So that is one part of the space complexity. The other part is the one occupied by the queue which is *O*(*N*). Overall, the space is *O*(*N*).
* Deserialization: For deserialization, the space is mostly occupied by the two lists that we use. The space complexity there is *O*(*N*). Note that when we re-initialize a list, the memory that was allocated earlier is deallocated by the garbage collector and it's essentially equal to a single list of size *O*(*N*).