# Question

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



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.



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:

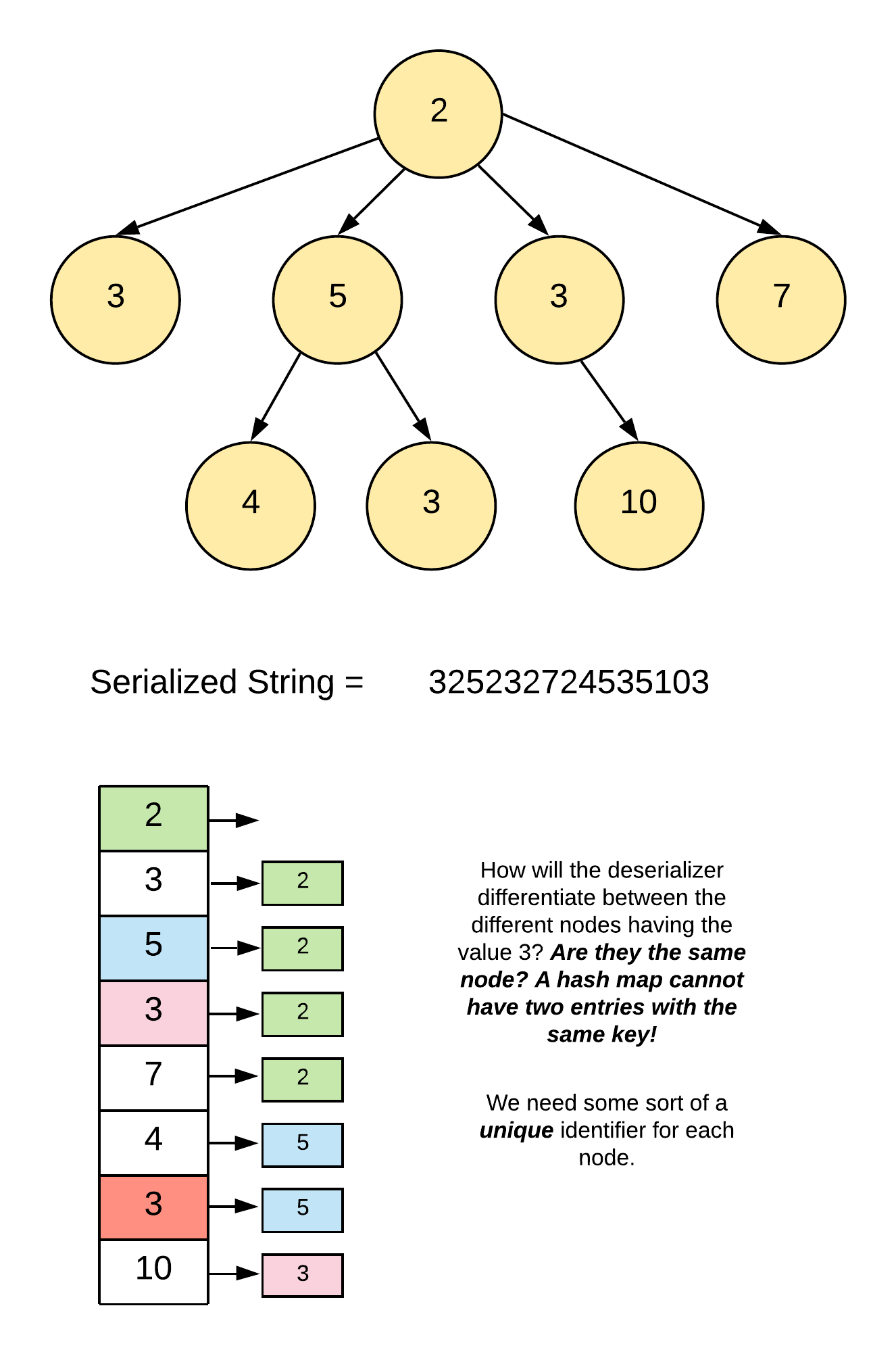
* We don't need to use any special delimiters just for separating numbers.
* We don't need costly split operations and instead, we can iterate on the input string one character at a time and form our tree.
* It's blazingly fast! On one of the solutions, I saw the run time come down to 2ms down from a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

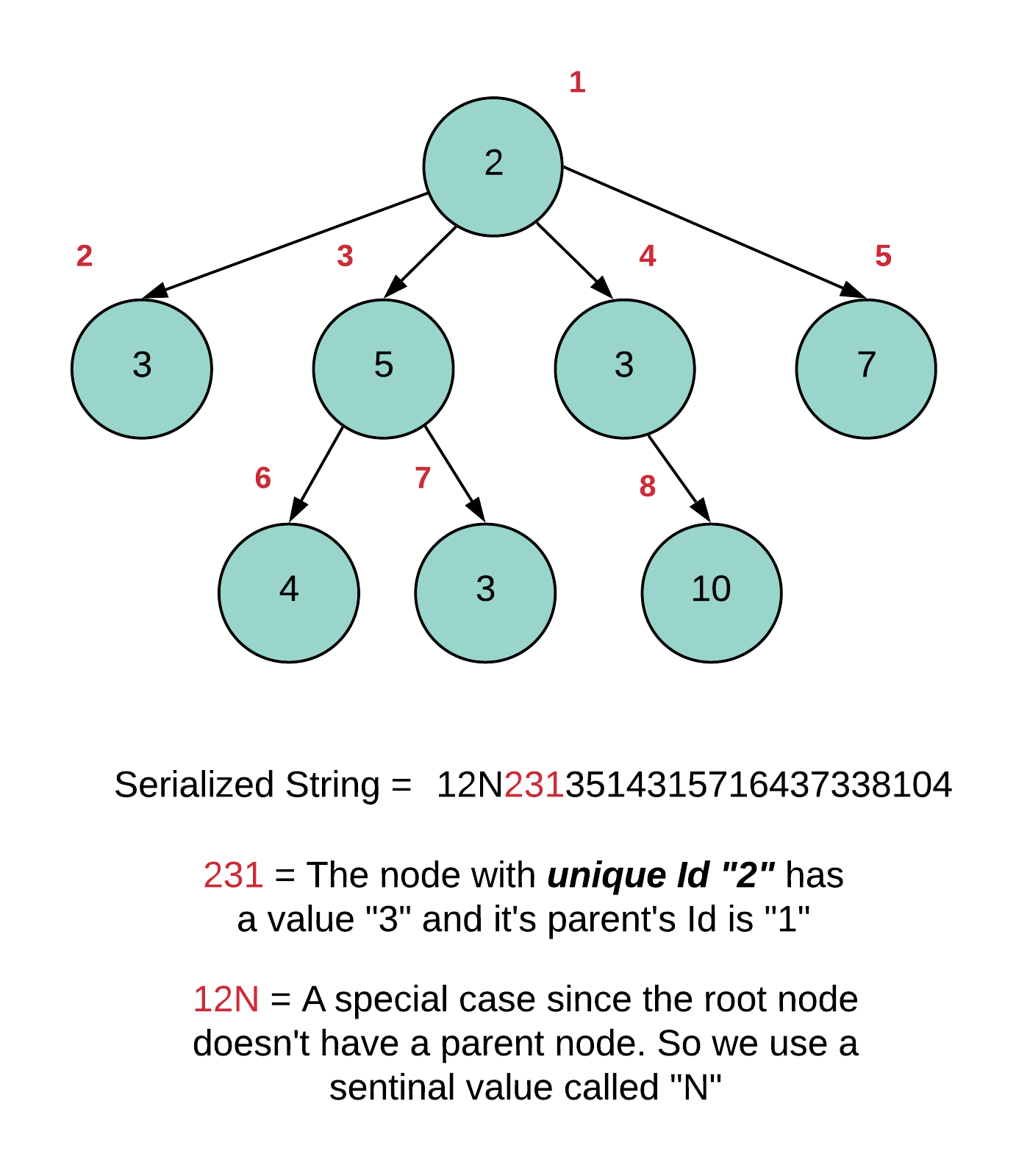
#### **Approach 1: Parent Child relationships**

**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.



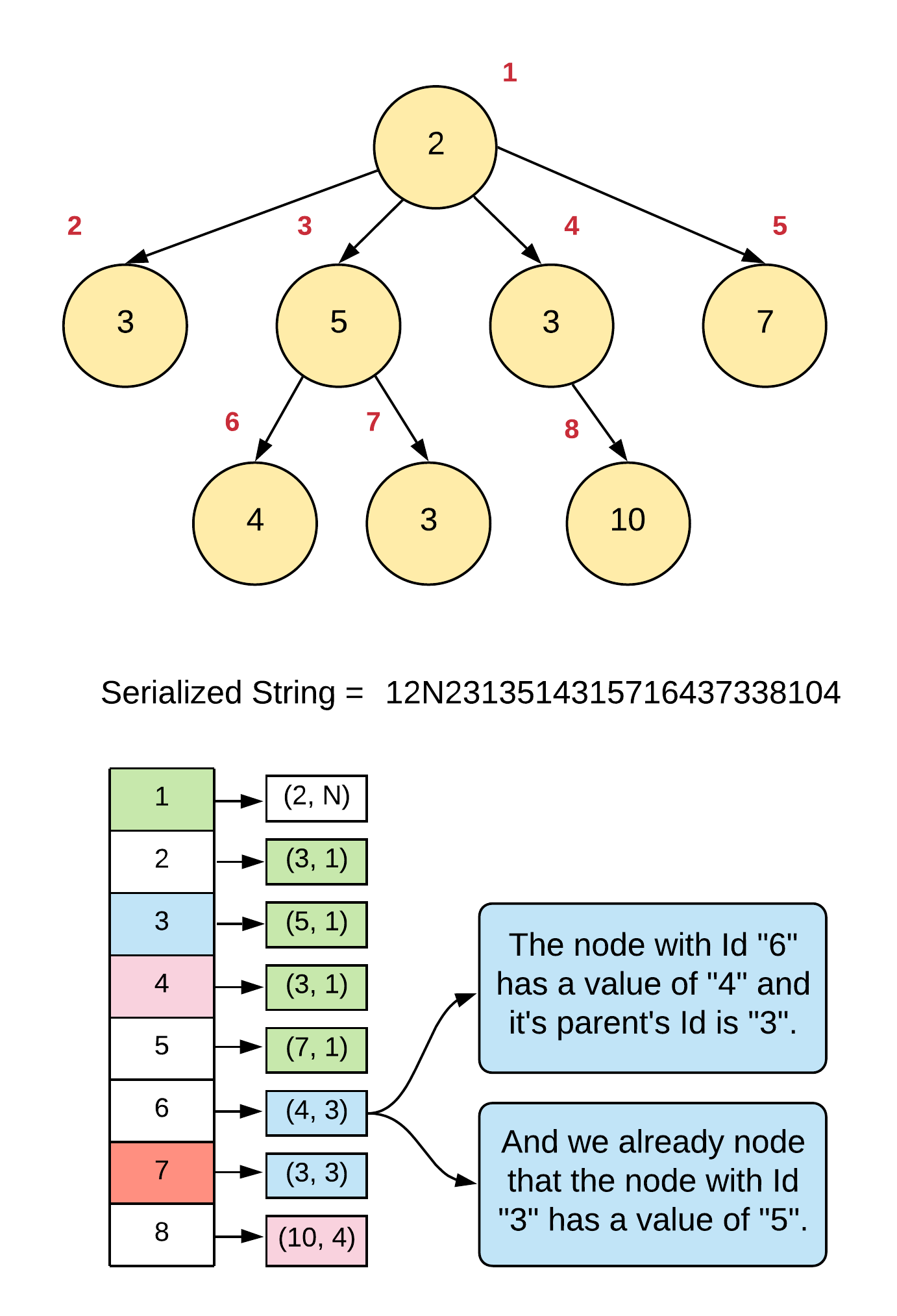
As mentioned in the image, with this simple serialization, we will run into problems since there can be nodes with duplicate data and we can't rely on just the values for deserialization. We need a way of differentiating different nodes. For that, we will be using a unique identifier for each node. Again, 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.



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.



**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

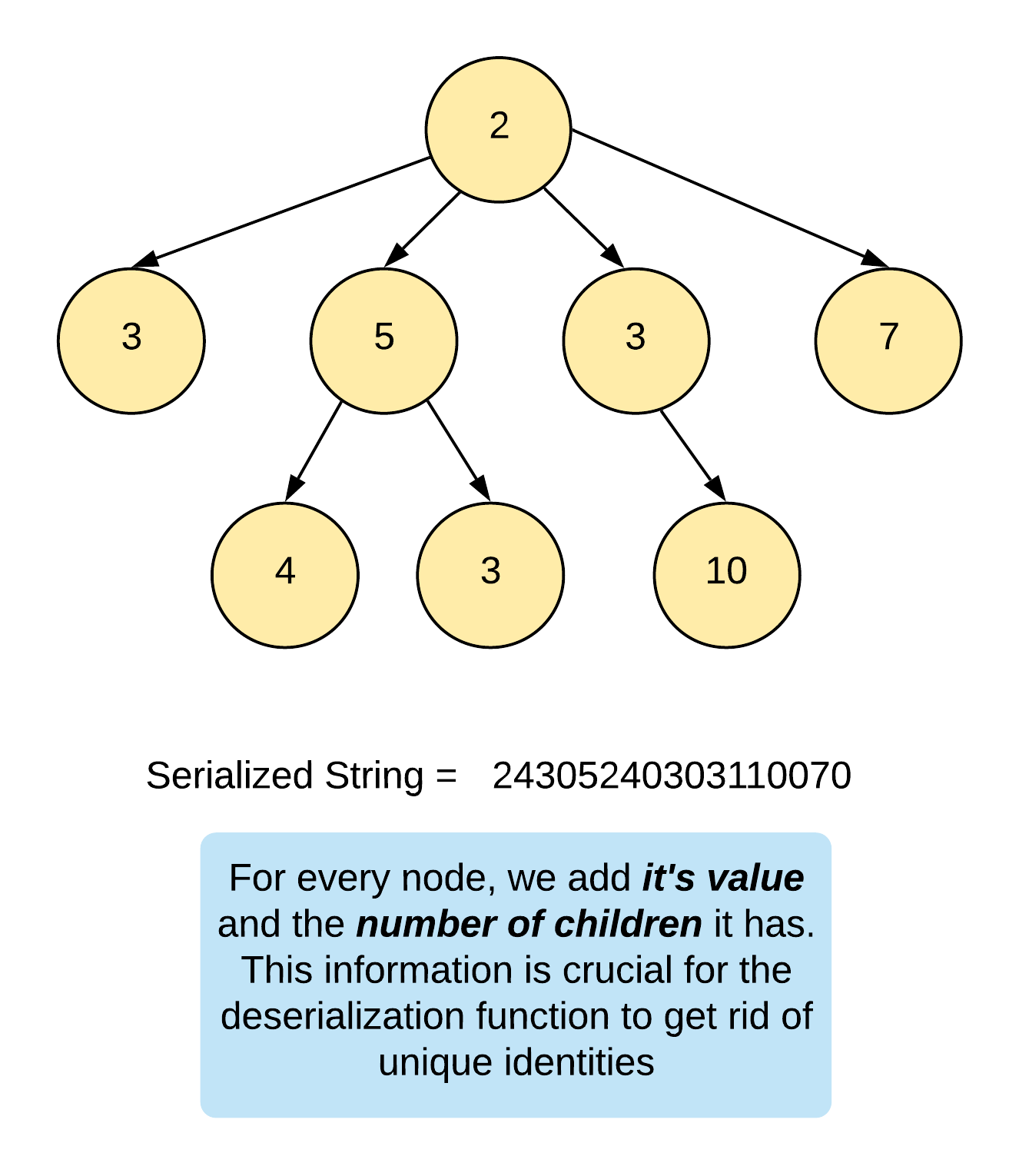
* Serialization: The space occupied by the serialization helper function is through recursion stack and the final string that is produced. Usually, we don't take into consideration the space of the output. However, in this case, the output is something which is not fixed. For all we know, someone might 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.



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]

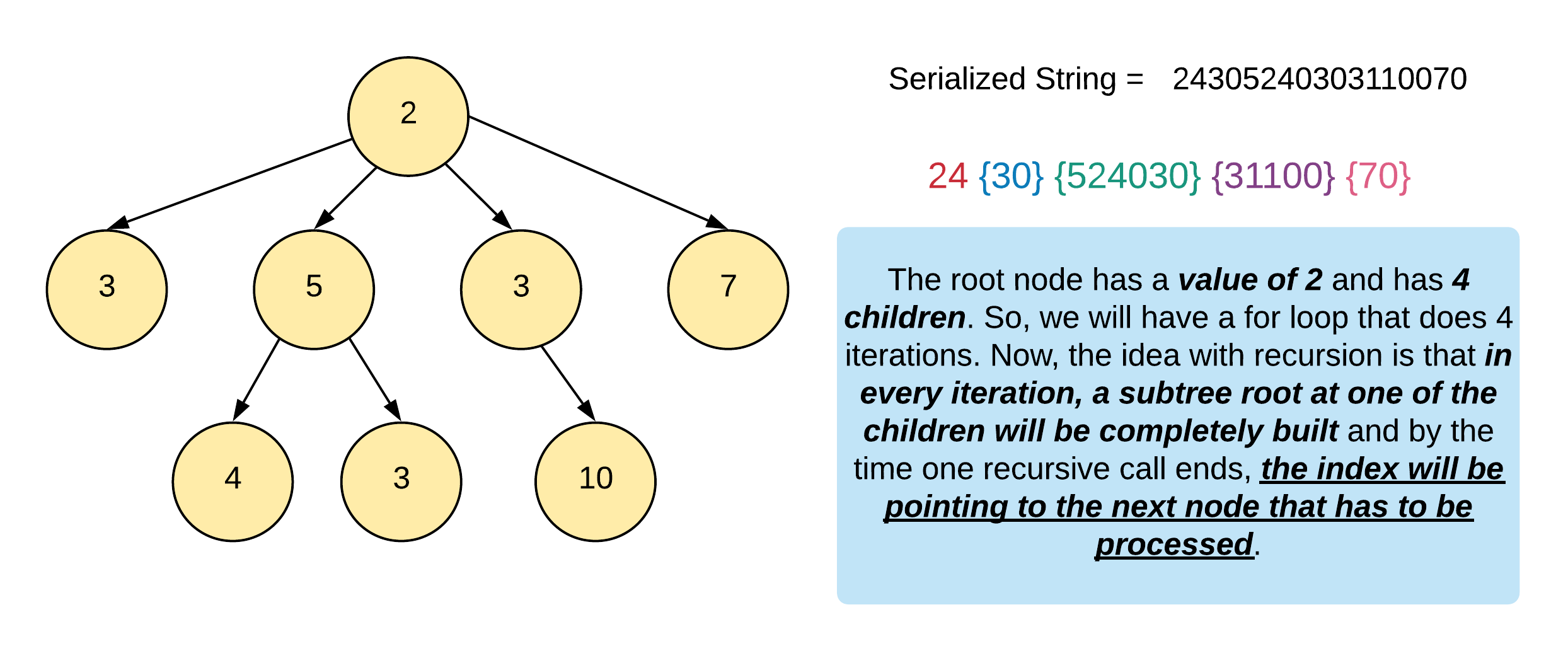
{

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.



**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)*O*(*N*) where N*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 2N2*N* = O(N)*O*(*N*)

Space Complexity

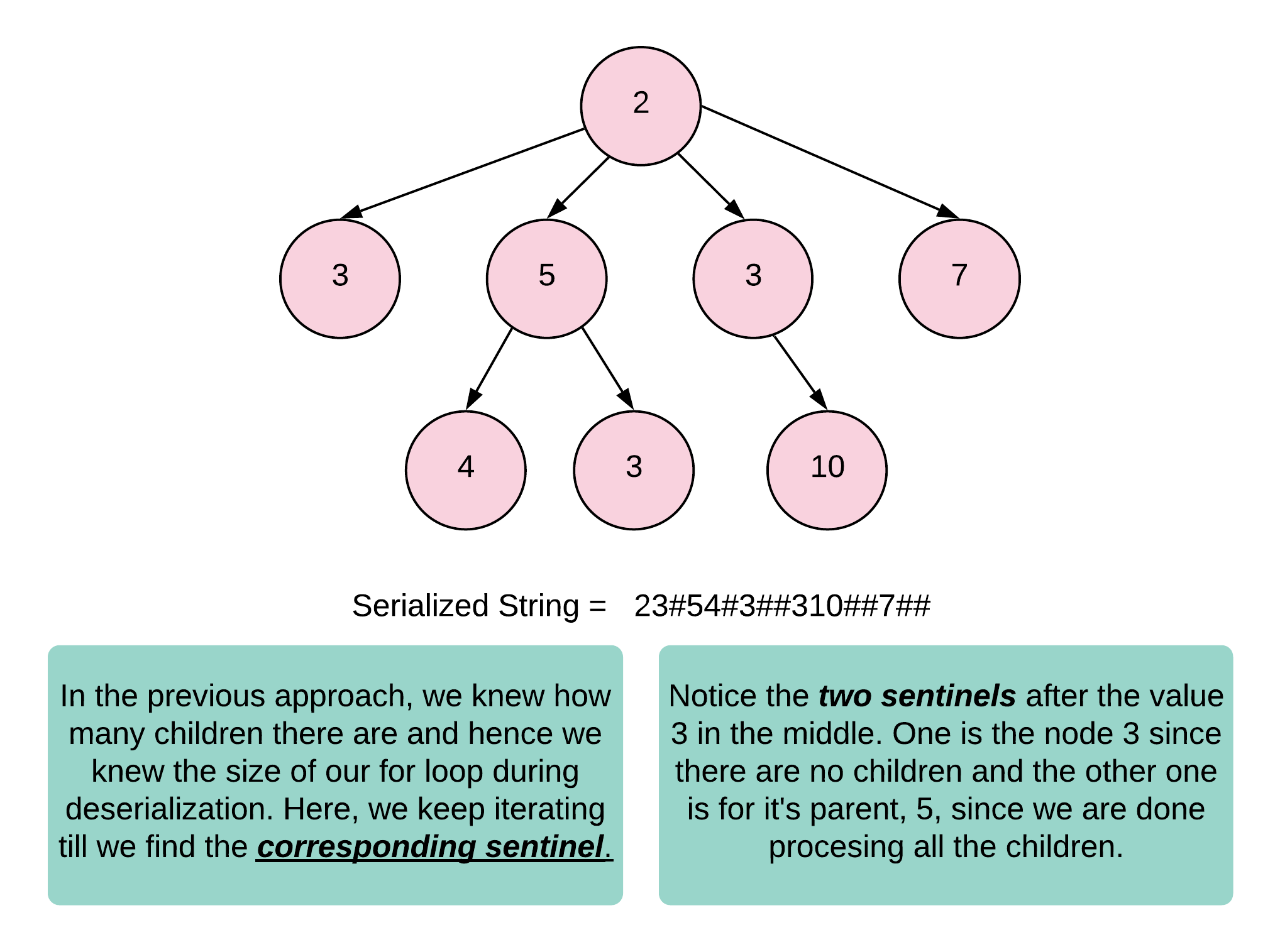
* Serialization: The space occupied by the serialization helper function is through recursion stack and the final string that is produced. We know the size of the final string to be 2N2*N*. So, that is one part of the space complexity. The other part is the one occupied by the recursion stack which is O(N)*O*(*N*). Overall, the space is O(N)*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)*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.



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)*O*(*N*) where N*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 2N2*N* = 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. We know the size of the final string to be 2N2*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)*O*(*N*). Overall, the space is O(N)*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)*O*(*N*).

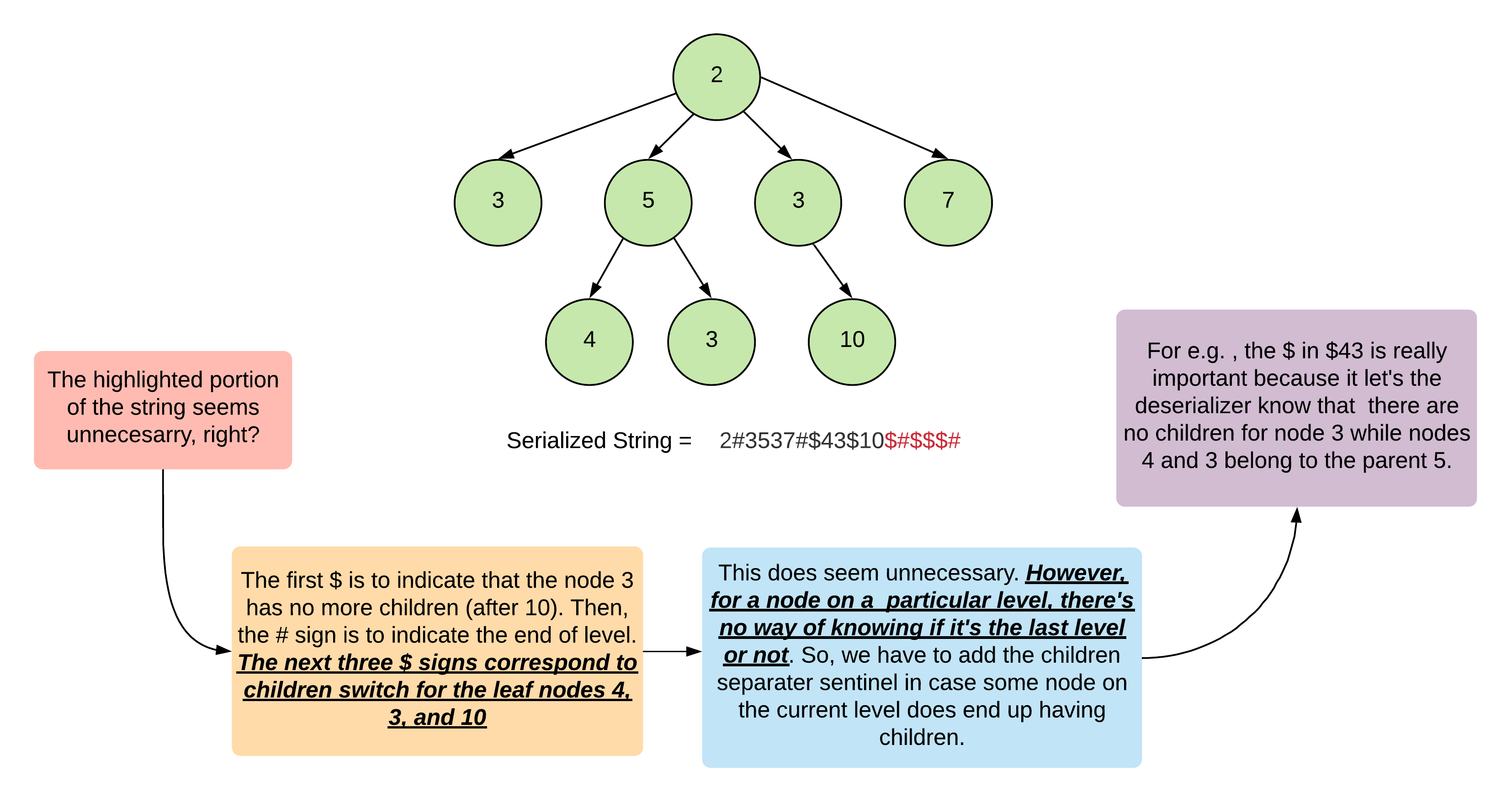
#### **Approach 4: Level order traversal**

**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.



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)*O*(*N*) where N*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 2N2*N* = O(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 2N2*N*. So that is one part of the space complexity. The other part is the one occupied by the queue which is O(N)*O*(*N*). Overall, the space is O(N)*O*(*N*).
* Deserialization: For deserialization, the space is mostly occupied by the two lists that we use. The space complexity there is O(N)*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)*O*(*N*).