**Serialize and Deserialize Binary 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 a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

**Clarification:** The input/output format is the same as [how LeetCode serializes a binary tree](https://leetcode.com/faq/#binary-tree). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Example 1:**

Diagram

Description automatically generated

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

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

**Example 2:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -1000 <= Node.val <= 1000

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* Encodes a tree to a single string.

\*

\* @param {TreeNode} root

\* @return {string}

\*/

var serialize = function(root) {

};

/\*\*

\* Decodes your encoded data to tree.

\*

\* @param {string} data

\* @return {TreeNode}

\*/

var deserialize = function(data) {

};

/\*\*

\* Your functions will be called as such:

\* deserialize(serialize(root));

\*/

Solution

Approach 1: Depth First Search (DFS)

**Intuition**

A screenshot of a computer

Description automatically generated with low confidence

The **serialization** of a Binary Search Tree is essentially to encode its values and more importantly its structure. One can traverse the tree to accomplish the above task. And it is well know that we have two general strategies to do so:

* *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.

In this task, however, the DFS strategy is more adapted for our needs, since the linkage among the adjacent nodes is naturally encoded in the order, which is rather helpful for the later task of **deserialization**.

Therefore, in this solution, we demonstrate an example with the preorder DFS strategy. One can check out more tutorial about Binary Search Tree on the [LeetCode Explore](https://leetcode.com/explore/learn/card/introduction-to-data-structure-binary-search-tree/).

**Algorithm**

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

/\* Definition for a binary tree node. \*/

public class TreeNode {

int val;

We start from the root, node 1, the serialization string is 1,. Then we jump to its left subtree with the root node 2, and the serialization string becomes 1,2,. Now starting from node 2, we visit its left node 3 (1,2,3,None,None,) and right node 4 (1,2,3,None,None,4,None,None) sequentially. Note that None,None, appears for each leaf to mark the absence of left and right child node, this is how we save the tree structure during the serialization. And finally, we get back to the root node 1 and visit its right subtree which happens to be a leaf node 5. Finally, the serialization string is done as 1,2,3,None,None,4,None,None,5,None,None,.

TreeNode right;

TreeNode(int x) {

val = x;

}

}

The preorder DFS traverse follows *recursively* the order of  
root -> left subtree -> right subtree.

As an example, let's serialize the following tree. Note that serialization contains information about the node values as well as the information about the tree structure.

// Serialization

public class Codec {

public String rserialize(TreeNode root, String str) {

// Recursive serialization.

if (root == null) {

str += "null,";

} else {

str += str.valueOf(root.val) + ",";

str = rserialize(root.left, str);

str = rserialize(root.right, str);

}

return str;

}

// Encodes a tree to a single string.

public String serialize(TreeNode root) {

return rserialize(root, "");

}

};

Now let's deserialize the serialization string constructed above 1,2,3,None,None,4,None,None,5,None,None,. It goes along the string, initiate the node value and then calls itself to construct its left and right child nodes.

public class Codec {

public TreeNode rdeserialize(List<String> l) {

// Recursive deserialization.

if (l.get(0).equals("null")) {

l.remove(0);

return null;

}

TreeNode root = new TreeNode(Integer.valueOf(l.get(0)));

l.remove(0);

root.left = rdeserialize(l);

root.right = rdeserialize(l);

return root;

}

// Decodes your encoded data to tree.

public TreeNode deserialize(String data) {

String[] data\_array = data.split(",");

List<String> data\_list = new LinkedList<String>(Arrays.asList(data\_array));

return rdeserialize(data\_list);

}

};

**Complexity Analysis**

* Time complexity : in both serialization and deserialization functions, we visit each node exactly once, thus the time complexity is O(N)*O*(*N*), where N*N* is the number of nodes, i.e. the size of tree.
* Space complexity : in both serialization and deserialization functions, we keep the entire tree, either at the beginning or at the end, therefore, the space complexity is O(N)*O*(*N*).

The solutions with BFS or other DFS strategies normally will have the same time and space complexity.

**Further Space Optimization**

In the above solution, we store the node value and the references to None child nodes, which means N \cdot V + 2N*N*⋅*V*+2*N* complexity, where V*V* is the size of value. That is called natural serialization, and has was implemented above.

The N \cdot V*N*⋅*V* component here is the encoding of values, can't be optimized further, but there is a way to reduce 2N2*N* part which is the encoding of the tree structure.

The number of unique binary tree structures that can be constructed using n nodes is C(n)*C*(*n*), where C(n)*C*(*n*) is the nth Catalan number. Please refer to [this article](https://leetcode.com/articles/unique-binary-search-trees/) for more information.

There are C(n)*C*(*n*) possible structural configurations of a binary tree with n nodes, so the largest index value that we might need to store is C(n) - 1*C*(*n*)−1. That means storing the index value could require up to 1 bit for n \leq 2*n*≤2, or \lceil log\_2(C(n) - 1) \rceil⌈*log*2​(*C*(*n*)−1)⌉ bits for n > 2*n*>2.

In this way one could reduce the encoding of the tree structure by \log Nlog*N*. More precisely, the [Catalan numbers](https://en.wikipedia.org/wiki/Catalan_number) grow as C(n) \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}*C*(*n*)∼*n*3/2*π*​4*n*​ and hence the theoretical minimum of storage for the tree structure that could be achieved is log(C(n)) \sim 2n - \frac{3}{2}\log(n) - \frac{1}{2}\log(\pi)*log*(*C*(*n*))∼2*n*−23​log(*n*)−21​log(*π*)