

13
.1
0

Serialize and Deserialize Binary Tree

HARD

■ What You Will Learn

- ✓ What serialization means: tree $\rightarrow$ string
- ✓ Preorder DFS with null markers to encode tree structure
- ✓ Deserialization: string $\rightarrow$ tree using a Queue
- ✓ Why this is a HARD problem and how to think through it

■ Intuition First

Serialization = converting a tree into a string that can be stored or transmitted. Deserialization = reconstructing the original tree from that string.

The challenge: we need to encode the STRUCTURE, not just the values. The key: include 'null' markers in the serialized string wherever a node is absent. Use Preorder (Root $\rightarrow$ Left $\rightarrow$ Right) so when deserializing, you always build the root first.

■ The Strategy

- Serialize: Preorder DFS. Append val for real nodes, 'N' for nulls
- Separator: comma between values in the string
- Deserialize: split string by comma $\rightarrow$ queue of tokens
- Rebuild: poll token. If 'N' $\rightarrow$ return null. Else $\rightarrow$ create node, recurse left then right

■ Visual — Serialize

Input tree

```
1
 / \
2 3
 / \
4 5
```

Preorder serialize:

Visit 1 $\rightarrow$ '1'

Go left to 2 $\rightarrow$ '1,2'

```

2.left = null → '1,2,N'
2.right = null → '1,2,N,N'
Go right to 3 → '1,2,N,N,3'
3.left = 4 → '1,2,N,N,3,4'
4.left=N,4.right=N → '1,2,N,N,3,4,N,N'
3.right = 5,5.left=N,5.right=N → '1,2,N,N,3,4,N,N,5,N,N'

```

Serialized: '1,2,N,N,3,4,N,N,5,N,N'

■ Visual — Deserialize

Tokens queue: [1, 2, N, N, 3, 4, N, N, 5, N, N]

Step 1	Poll '1' → create node(1). Recurse left.
Step 2	Poll '2' → create node(2). Recurse left.
Step 3	Poll 'N' → return null. node(2).left = null.
Step 4	Recurse right for node(2). Poll 'N' → return null. node(2).right = null.
Step 5	node(1).left = node(2). Recurse right for node(1).
Step 6	Poll '3' → create node(3). Recurse left → '4' → create node(4)...
Step 7	Eventually: full tree reconstructed ✓

■ Java Code — Serialize

```

public String serialize(TreeNode root) {
    StringBuilder sb = new StringBuilder();
    serializeHelper(root, sb);
    return sb.toString();
}

private void serializeHelper(TreeNode node, StringBuilder sb) {
    if (node == null) {
        sb.append("N,"); // Null marker
        return;
    }
    sb.append(node.val).append(","); // Append value
    serializeHelper(node.left, sb); // Recurse left (Preorder)
    serializeHelper(node.right, sb); // Recurse right
}

```

■ Java Code — Deserialize

```

public TreeNode deserialize(String data) {
    Queue<String> tokens = new LinkedList<>(Arrays.asList(data.split(",")));
    return deserializeHelper(tokens);
}

```

```
}

private TreeNode deserializeHelper(Queue<String> tokens) {
    String val = tokens.poll();
    if (val.equals("N")) return null; // Null marker → no node
    TreeNode node = new TreeNode(Integer.parseInt(val));
    node.left = deserializeHelper(tokens); // Rebuild left subtree
    node.right = deserializeHelper(tokens); // Rebuild right subtree
    return node;
}
```

■ ■ Complexity

Case	Time	Space	What it means
Serialize	O(n)	O(n)	Visit each node once. String built incrementally
Deserialize	O(n)	O(n)	Process each token once. Queue + recursion stack O(h)

■ Why Null Markers Are Essential

- Without null markers: '1,2,3' could represent many different trees
- With null markers: '1,2,N,N,3,N,N' uniquely defines one tree
- Preorder + null markers → complete structural encoding
- Inorder alone can't uniquely reconstruct (needs Preorder or Postorder too)

■ ■ Common Mistakes

- Using inorder for serialization — inorder alone isn't unique without nulls
- Forgetting to encode null nodes — without them structure is lost
- Using index-based deserialization with array — Queue is cleaner and avoids off-by-one errors
- Not handling negative values in split (e.g. '-1' with wrong delimiter)

■ Pro Tips

- This is exactly how expression trees are stored in databases and compilers
- Level order serialization is also valid — used in LeetCode's own tree format: '1,2,3,null,null,4,5'
- Space optimization: use bit encoding for null markers in production

■ Visual Thinking

Serialization is like writing a recipe for a specific tree.

Anyone who follows the recipe (deserializes) gets the exact same tree.

The null markers are like 'stop here' instructions — they tell you where branches end.

Preorder ensures you always have the parent before you need to attach children.

Animation: tree nodes pop up one by one as tokens are consumed from the queue.

Null markers cause 'phantom' leaf positions to be skipped over.

■ Practice Problems

[Hard] Serialize using level order (BFS approach)

[Hard] Serialize BST — can you do it without null markers? Why?

[Hard] Encode N-ary tree (multiple children per node)

[Hard] Verify if a serialized string represents a valid binary tree

■ Quick Revision Box

- Serialize: Preorder DFS, append val or 'N' for nulls, comma-separated
- Deserialize: split string → Queue of tokens
- Poll token: 'N' → return null, else → create node, recurse left then right
- $O(n)$ time and space for both operations
- Null markers are the KEY — they encode tree structure
- Preorder + nulls uniquely identifies any binary tree