Serialization and Deserialization of a BST - Complete Documentation

1. Problem Statement

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 to be reconstructed later in the same or another computer environment.

The problem requires designing an algorithm to serialize and deserialize a Binary Search Tree (BST).

- Serialization: Convert a BST into a string representation.
- Deserialization: Convert the serialized string back into the original BST structure.

Constraints

- The number of nodes in the tree is in the range [0, 10^4].
- $0 \leq \text{Node.val} \leq 10^4$.
- The input tree is guaranteed to be a Binary Search Tree (BST).
- The serialized string should be as compact as possible.

2. Intuition

Since the input tree is a Binary Search Tree (BST), we can take advantage of its properties:

- The left child contains values smaller than the root.
- The right child contains values greater than the root.
- The in-order traversal results in a sorted sequence.

A simple way to serialize and deserialize a BST is to use level-order traversal (BFS) or pre-order traversal (DFS).

3. Key Observations

i. **Serialization Format:**
   a. We traverse the BST using level-order (BFS) and store values as a comma-separated string.
   b. null represents missing children.

ii. **Deserialization Logic:**
   a. Read the values from the serialized string.
   b. Reconstruct the tree level by level using a queue.

iii. **Compact Representation:**
   a. Using null values ensures that the tree structure remains intact.

4. Approach

Serialization (Level-Order Traversal - BFS)

i. Use a queue for level-order traversal.
ii. Append node values to a list.
iii. Use "null" for missing nodes.
iv. Convert the list to a comma-separated string.

Deserialization (Level-Order Reconstruction - BFS)

i. Convert the string back to a list.
ii. Create the root node from the first value.
iii. Use a queue to assign children level by level.

5. Edge Cases

- Empty Tree → "" (empty string should deserialize to None).
- Single Node Tree → [1].
- Left-skewed Tree → [3,2,null,1,null].
- Right-skewed Tree → [1,null,2,null,3].
- Balanced Tree → [4,2,6,1,3,5,7].
- Large Tree (e.g., 10^4 nodes).

6. **Complexity Analysis**

Time Complexity

- Serialization: O(N), where N is the number of nodes (traverse each node once).
- Deserialization: O(N), as we traverse each node and reconstruct the tree.

Space Complexity

- Serialization: O(N), as we store all node values in a list.
- Deserialization: O(N), due to the queue storing tree nodes.

7. **Alternative Approaches**

Preorder Traversal (DFS) Approach

- Instead of BFS, we can use preorder traversal for serialization.
- The order is root → left → right.
- During deserialization, we recursively reconstruct the tree.
- Pros: More compact representation (no null values).
- Cons: Requires additional logic to reconstruct BST correctly.

8. **Test Cases**

```
def test_codec():
    ser = Codec()
    deser = Codec()

    # Test 1: Simple BST
    root = TreeNode(2)
    root.left = TreeNode(1)
    root.right = TreeNode(3)
    assert ser.serialize(root) == "2,1,3,null,null,null,null"
    assert deser.serialize(deser.deserialize(ser.serialize(root))) == "2,1,3,null,null,null,null"
    # Test 2: Empty Tree
    assert ser.serialize(None) == ""
    assert deser.deserialize("") == None

    # Test 3: Skewed Tree (Left)
    root = TreeNode(3)
    root.left = TreeNode(2)
    root.left.left = TreeNode(1)
    assert ser.serialize(root) == "3,2,null,1,null,null,null"

    # Test 4: Skewed Tree (Right)
    root = TreeNode(1)
    root.right = TreeNode(2)
    root.right.right = TreeNode(3)
    assert ser.serialize(root) == "1,null,2,null,3,null,null"

    print("All tests passed!")

# Run the tests
test_codec()
```

9. Final Thoughts

- This solution effectively serializes and deserializes a BST using level-order traversal (BFS).
- The approach ensures a compact representation while maintaining the tree's structure.
- Alternative DFS (preorder) solutions can be considered for further optimization.
- The implemented test cases cover all edge cases, ensuring robustness.