Topic Introduction

Today, let's dive into the wonderful world of **serialization** — a technique that lets us convert complex data structures, like trees and lists, into a simple string format, and then back again. Serialization is like packing your favorite LEGO castle into a box so you can rebuild it exactly the same way later (deserialization).

Serialization means converting a data structure into a string (or sequence of bytes) so it can be stored, transmitted, or reconstructed. **Deserialization** is the reverse: rebuilding the original structure from the string.

This concept is crucial in interviews because:

- It appears in distributed systems, file storage, and network communication questions.
- It tests your ability to traverse and encode structures efficiently.
- It checks your attention to detail for edge cases.

When is serialization useful?

- Saving/loading data (think: saving a game state).
- Sending data over a network.
- Comparing, logging, or duplicating structures.

A Simple Example (not from today's problems):
Suppose you have a linked list: `1 -> 2 -> 3`.
You could serialize this as the string `"1,2,3"`.
To deserialize, split the string by commas and rebuild the list node by node.

But what if the data structure is more complex, like a tree, or contains variable-length strings, or even null values? That's where today's problems come in!

Why These Problems Are Grouped Together

The three problems below all ask you to **serialize and deserialize data structures**, but each with a twist:

- **Serialize and Deserialize Binary Tree:** Any binary tree, including null children, using a traversal (e.g., preorder) and placeholders for nulls.
- **Serialize and Deserialize BST:** A Binary Search Tree, which lets us optimize since BSTs are more structured.
- **Encode and Decode Strings:** An array of strings, where you must handle strings of any length (including those with commas or special characters).

Each one builds your skill for encoding/decoding structures robustly, and shows different serialization tricks.

Problem 1: Serialize and Deserialize Binary Tree

LeetCode Link: Serialize and Deserialize Binary Tree

Problem Statement (In Our Words)

You are given the root of a binary tree.

Write two functions:

- `serialize(root)`: Convert the tree into a string.
- `deserialize(data)`: Rebuild the tree from that string.

You must encode all structure, including null pointers, so you can recover the *exact same* tree.

Example

Input tree:

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

Serialized string: `"1,2,null,null,3,4,null,null,5,null,null"`

How?

- Preorder traversal (root-left-right).
- Use `'null'` for empty children.

Thought Process

This problem is about faithfully recording both values and tree structure.
A simple preorder traversal (visit node, then left, then right) works — but we must record *null* children so that the tree can be rebuilt identically.

Pen-and-paper tip:
Draw the tree and write down the order in which you'd visit nodes, including where the nulls would be hit.

Another test case to try:

- Input tree:

`

1
 \
  2

`

- Expected serialization: `"1,null,2,null,null"`

Brute-force Approach

You could try to serialize by just listing node values in preorder, but you'll lose information about null children and won't be able to rebuild the tree uniquely.

Complexity:

- Time: O(N) to visit all nodes.
- Space: O(N) for output string and recursion stack.

Optimal Approach: Preorder with Nulls

Pattern:

- Use preorder traversal.
- For each node:
    - If node is not null: add its value.
    - If node is null: add `'null'`.
- Join values with commas.

Step-by-step:

- Write a helper that does preorder traversal, appending values or `'null'` for missing children.
- To deserialize, split the string by commas, and recursively rebuild the tree:
    - If value is `'null'`, return None.
    - Otherwise, create a node with that value, and recurse for left and right.

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Codec:
    # Serialize: tree -> string
    def serialize(self, root):
        def helper(node):
            if not node:
                vals.append('null')
                return
            vals.append(str(node.val))
            helper(node.left)
            helper(node.right)
        vals = []
        helper(root)
        return ','.join(vals)

    # Deserialize: string -> tree
    def deserialize(self, data):
        def helper():
            val = next(vals)
            if val == 'null':
                return None
            node = TreeNode(int(val))
            node.left = helper()
            node.right = helper()
            return node
        vals = iter(data.split(','))
        return helper()
```

Time Complexity: O(N), N = number of nodes

Space Complexity: O(N) for the output string and recursion stack

Code Explanation

- `serialize` uses a helper function to do a preorder traversal, adding values or `'null'` (for None) to a list, then .join()s them into a string.
- `deserialize` uses an iterator over the split string, and recursively rebuilds the tree. For each value:
    - If `'null'`, return None.
    - Else, create a node, and recursively set its left and right children.

Trace (using example tree):

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

Traversal order: 1, 2, null, null, 3, 4, null, null, 5, null, null

- serialize: builds "1,2,null,null,3,4,null,null,5,null,null"
- deserialize: splits by comma, and for each value, builds the nodes and left/right recursively.

Try this test case manually:

Input:

```
    1
   /
  2
```

What should the serialized string look like?

Try to dry-run the code on this example!

Take a moment to solve this yourself before reading the code!

Problem 2: Serialize and Deserialize BST

LeetCode Link: Serialize and Deserialize BST

Problem Statement (Paraphrased)

Given the root of a Binary Search Tree (BST), serialize it to a string, and deserialize it back to a BST.

Key difference:
Because it's a BST, you do **not** need to encode null children explicitly. You can reconstruct the tree using preorder traversal and BST properties.

How is this different from Problem 1?

- **Binary Tree:** Need to record all nulls.
- **BST:** Structure can be inferred from values (using value order and BST rules).

Example

Input tree:

```
    2
   / \
  1   3
```

Serialized: `"2,1,3"`

Deserialize: reconstructs the original BST.

Try this:
Input: `[5, 3, 7, 2, 4, 6, 8]`
Preorder: `5,3,2,4,7,6,8`

Brute-force

If you did the same as Problem 1 (recording nulls), it's correct, but wasteful. For BSTs, you can save space by not storing nulls.

Optimal Approach: Preorder Traversal and Bounds

Pattern:
- Serialize with preorder traversal (just values, comma-separated).
- To deserialize, use the preorder sequence and BST bounds:
    - At each step, if the next value fits the current bounds, create a node and recurse.
    - Otherwise, return None.

Step-by-Step:

- Serialize: Preorder traversal, join values by comma.
- Deserialize: Use recursion with upper/lower bounds to rebuild left and right subtrees.
    - For each value, if it fits the bound, create node, recurse for children with updated bounds.

Pseudocode:

```
function serialize(root):
    vals = []
    preorder(node):
        if node is null: return
        vals.append(node.val)
        preorder(node.left)
        preorder(node.right)
    preorder(root)
    return join(vals, ',')

function deserialize(data):
    vals = iterator(split(data, ','))
    function build(lower, upper):
        if no more vals: return null
        if vals[0] < lower or vals[0] > upper: return null
        val = next(vals)
        node = new TreeNode(val)
        node.left = build(lower, val)
        node.right = build(val, upper)
        return node
    return build(-infinity, infinity)
```

Example Input/Output:

Input BST:

```
    5
   / \
  3   7
```

Serialize: `"5,3,7"`

Deserialize: reconstructs the same BST.

Test case to try:

Input: `[2,1,3]`

Serialized: `"2,1,3"`

Code Trace (for "5,3,2,4,7,6,8"):

- Take 5: root
- 3 < 5: left child
- 2 < 3: left child

- (no value < 2): left null
- next is 4 > 3: right child of 3
- 7 > 5: right child
- 6 < 7: left child
- 8 > 7: right child

Complexity:

- Time: O(N)
- Space: O(N)

Problem 3: Encode and Decode Strings

LeetCode Link: Encode and Decode Strings

Problem Statement (In Our Words)

Given a list of strings, write two functions:

- `encode`: Convert the list into a single string.
- `decode`: Convert the string back into the original list.

Challenge:
Strings may contain any character, including your chosen delimiter!

How is this different or more challenging?

- No tree structure — but you must handle arbitrary strings (even ones containing commas or special markers).
- You can't just join with a delimiter (since the delimiter might appear in the actual string).

Example

Input: `["hello", "world"]`
Possible encoding: `"5#hello5#world"`

Each string is prefixed by its length and a special separator (like `#`).

Brute-force

Joining with a simple delimiter (e.g., `",".join(strings)`) breaks if the strings themselves contain commas.

Optimal Approach: Length-Prefix Encoding

Pattern:

- For each string, write its length, a separator (e.g., `#`), then the string itself.
- To decode, read the length, the separator, then the string of that length, and repeat.

Pseudocode:

```
function encode(strs):
    result = ""
    for s in strs:
        result += len(s) + "#" + s
    return result

function decode(s):
    i = 0
    result = []
    while i < len(s):
        j = i
        while s[j] != '#': j += 1
        length = int(s[i:j])
        result.append(s[j+1 : j+1+length])
        i = j+1+length
    return result
```

Example Input/Output:

Input: `["leet", "code", "interview"]`

Encoding: `"4#leet4#code9#interview"`

Decoding: parses each length and string to recover the original array.

Test case to try:

Input: `["", "#", "12#abc"]`

Encoding: `"0#1##5#12#abc"`

Complexity:

- Time: O(N), where N = total length of all strings.
- Space: O(N)

Summary and Next Steps

Today, you tackled **serialization and deserialization** — the art of turning complex data structures into strings and back again. You saw:

- How to record tree structure with nulls (general binary tree).
- How BST properties can help you serialize more compactly.
- How to handle arbitrary strings, even when they contain your chosen delimiters.

Key patterns:

- Preorder traversal for trees.
- Using placeholders for nulls when structure can't be inferred.
- Encoding string length to handle arbitrary character data.

Common mistakes:

- Forgetting to encode nulls when necessary.
- Picking a delimiter that might appear in the data.
- Failing to use BST constraints for more efficient serialization.

Action List

- Try solving all three problems yourself — even if a solution is shown, writing it out is how you build understanding!
- For Problem 2 and 3, challenge yourself: can you implement them using a different approach? For example, can you serialize a BST using inorder and postorder?
- Explore other serialization problems: try serializing n-ary trees, graphs, or nested lists.
- Compare your solutions with the official ones and discuss with peers — especially for tricky cases like empty strings or all-null trees.
- If you get stuck, that's normal! Step back, sketch out some small examples, and keep practicing.

Keep up the great work — serialization is a superpower, and you’re well on your way to mastering it!