Topic Introduction

Today, we dive into the world of **serialization and encoding** — a foundational concept that shines in system design and coding interviews alike.

What is Serialization and Encoding?
Serialization is the process of converting data structures or objects into a format that can be easily stored, transmitted, or reconstructed later. Common serialization formats include strings, arrays, or even binary blobs.

Encoding is a broader term — it means transforming data from one form into another, usually to make it fit constraints (like shortening a URL), or to make it processable by another system.

Why does this matter?
In interviews, you're often asked to design systems that:

- Save and load complex data (like a binary tree) to/from disk or network.
- Convert data for efficient storage or communication, e.g., shortening URLs.
- Manage resources efficiently, like assigning/releasing phone numbers.

Example (but not from our problems):
Suppose you need to save a list of integers to disk. You might serialize it as a comma-separated string: `[1, 2, 3]` becomes `"1,2,3"`. Later, you parse that string back into a list. This pattern generalizes to much more complex objects!

Serialization and encoding are everywhere — from saving game progress to sending messages across the internet.

Why These 3 Problems?

Today's trio all revolve around **transforming data between different forms**:

- Serialize and Deserialize Binary Tree: Turn a tree into a string and back.
- Encode and Decode TinyURL: Shrink a long URL into a short one and retrieve it.
- Design Phone Directory: Efficiently assign and recycle numbers (encoding which numbers are in use).

Each problem highlights a different facet of serialization/encoding: data structure flattening, data shortening, and resource tracking. Let's explore each, starting with the most structural — the binary tree.

Problem 1: Serialize and Deserialize Binary Tree

[LeetCode Link](https://leetcode.com/problems/serialize-and-deserialize-binary-tree/)

Problem Statement (Rephrased)

You are given the root of a binary tree. Design an algorithm to:

- Serialize the tree into a string, so it can be saved or sent over a network.

- Deserialize that string back into the original binary tree.

You must be able to reconstruct the *exact* tree structure, including null (missing) children.

Example

Input Tree:

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

Serialized: `"1,2,#,#,3,4,#,#,5,#,#"`

Deserialized: Rebuild the same tree structure from the string.

Thought Process

- Serialization must encode both values and "empty" spots — otherwise, different trees could map to the same string!
- We need a way to mark missing nodes. Let's use `#` as a placeholder.
- A common approach is **preorder traversal** (root, left, right):
  - Write the value.
  - If a node is missing, write `#`.

Try sketching out a small example on paper. Can you see how the tree's shape is preserved?

Extra Test Case:

Tree:

```
    1
     \
      2
     /
    3
```

Serialized: ?

Brute Force

You could record only values, but then you can't tell where left/right children are missing. Not enough information!

- **Complexity:** O(N) time and space (since you must process each node).

Optimal Approach

Pattern: *Recursive serialization with null markers (preorder traversal).*

Step-by-Step:

- **Serialize:**
  - For each node:
    - If null, write #.
    - Otherwise:
      - Write node's value.
      - Serialize left child.
      - Serialize right child.
  - Join parts with commas.
- **Deserialize:**
  - Read values split by comma.
  - For each value:
    - If #, return None.
    - Otherwise, create a node, then recursively build left and right children.

Code (Python)

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string."""
        def helper(node):
            if node is None:
                vals.append("#")
                return
            vals.append(str(node.val))
            helper(node.left)
            helper(node.right)
        vals = []
        helper(root)
        return ",".join(vals)

    def deserialize(self, data):
        """Decodes your encoded data to tree."""
        def helper():
```

```
            val = next(values)
            if val == "#":
                return None
            node = TreeNode(int(val))
            node.left = helper()
            node.right = helper()
            return node
        values = iter(data.split(","))
        return helper()
```

Time Complexity: O(N)

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

How the Code Works

- The `serialize` function does a preorder traversal. For each node, it adds the value or `#` for None. All parts are joined by commas.
- The `deserialize` function reads the string, splits by comma, and reconstructs the tree by recursively assigning children.
- `values = iter(data.split(","))` creates an iterator so each recursive call consumes the next value.

Trace Example

For the tree:

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

Serialization steps:

- Write `1`
- Left child: `2`
    - Left: `#`
    - Right: `#`
- Right child: `3`
    - Left: `4`
        - Left: `#`
        - Right: `#`
    - Right: `5`
        - Left: `#`
        - Right: `#`

Final: `"1,2,#,#,3,4,#,#,5,#,#"`

Try this:

Tree:

```
  1
   \
    2
   /
  3
```

What does the serialized string look like? Draw it out and test your understanding!

Take a moment to try this problem on your own before reviewing the code above.

Problem 2: Encode and Decode TinyURL

[LeetCode Link](https://leetcode.com/problems/encode-and-decode-tinyurl/)

Problem Statement (Rephrased)

Design a service to encode a long URL into a short URL, and decode it back. Each encode/decode operation should be fast, and the mapping must be one-to-one.

Why it's Similar

Like the previous problem, we're converting data between forms — but now, it's about mapping a potentially huge input space (all URLs) into a smaller, fixed-length code space.

Example

Input:
Long URL: `"https://leetcode.com/problems/design-tinyurl"`
Output:
Short URL: `"http://tinyurl.com/abc123"`
Decoding `"http://tinyurl.com/abc123"` gives the original long URL.

Test case to try:
Encode: `"https://google.com/"`
Decode your result—do you get the original?

Brute Force

You could store every long URL with an incrementing ID, then map IDs to URLs. However, this can leak information (sequential IDs), and may not be compact.

- **Complexity:** O(1) per operation with good hashing/data structures.

Optimal Approach

Pattern: *Generate a unique short code for each long URL and store the mapping.*

Step-by-Step

- **Encode:**
  - Generate a unique short code (e.g., 6 random characters).
  - Store a mapping from code to long URL.
  - Return the base URL plus the code.
- **Decode:**
  - Extract the code from the short URL.
  - Look up the code in the mapping to get the long URL.

Collision avoidance: If a random code is already used, re-generate.

Pseudocode

```
initialize: code_to_url = empty hash map

encode(longUrl):
    repeat:
        code = generate_random_6_char_string()
    until code not in code_to_url
    code_to_url[code] = longUrl
    return "http://tinyurl.com/" + code

decode(shortUrl):
    code = last 6 characters of shortUrl
    return code_to_url[code]
```

Example Walkthrough

- Encode "https://leetcode.com":
  - Generate "abc123"
  - Store: code_to_url["abc123"] = "https://leetcode.com"
  - Return "http://tinyurl.com/abc123"
- Decode "http://tinyurl.com/abc123":
  - Extract "abc123"
  - Look up: returns "https://leetcode.com"

Test case for you:

Encode `"https://openai.com/"`
What short URL do you get? Decode it — does it match the original?

Complexity:

- Time: O(1) per encode/decode (assuming hash map lookups and code generation are fast)
- Space: O(N) for N URLs stored

Problem 3: Design Phone Directory

[LeetCode Link](https://leetcode.com/problems/design-phone-directory/)

Problem Statement (Rephrased)

Design a phone directory that supports:

- `get()`: Provide an available number.
- `check(number)`: Is this number available?
- `release(number)`: Recycle a number, making it available again.

All numbers are in a fixed range [0, maxNumbers - 1]. Operations must be efficient.

How It's Similar and Different

Here, "encoding" is about managing and tracking available resources (numbers), and mapping between their "in-use" and "available" states. We're serializing the pool of numbers into a data structure that enables fast allocation and checks.

Brute Force

You could scan the entire array of numbers to find an available one, but that would be O(N) per operation.

Optimal Approach

Pattern: *Use a set or queue to track available numbers, and a set for assigned numbers.*

Step-by-Step

- **Initialize:**
  - Store all numbers in a queue (or set) of available numbers.
  - Use a set to track assigned numbers.
- **get():**
  - If available numbers exist, pop one and add to assigned set.
  - Return it. If none, return -1.

- **check(number):**
    - Return True if number is not in assigned set.
- **release(number):**
    - If number is in assigned set, remove and add back to available.

Pseudocode

```
initialize:
    available = queue of [0, 1, ..., maxNumbers-1]
    assigned = empty set

get():
    if available is empty:
        return -1
    number = available.pop()
    assigned.add(number)
    return number

check(number):
    return number not in assigned

release(number):
    if number in assigned:
        assigned.remove(number)
        available.add(number)
```

Example

Suppose `maxNumbers = 3`

- Start: available = [0, 1, 2], assigned = {}
- `get()` -> returns 0, assigned = {0}
- `get()` -> returns 1, assigned = {0,1}
- `check(2)` -> True
- `get()` -> returns 2, assigned = {0,1,2}
- `get()` -> returns -1
- `release(1)`, assigned = {0,2}, available = [1]
- `get()` -> returns 1

Try this test case:

- Start with maxNumbers = 2
- get(), get(), check(1), release(0), get()

What should the results be?

Complexity:

- Time: O(1) per operation (with sets and queue)
- Space: O(N)

Summary and Next Steps

Today, you explored **serialization and encoding** from three perspectives:

- Flattening and reconstructing complex structures (trees)
- Mapping big objects to small representations (URLs)
- Managing resource state with efficient encoding (phone numbers)

Key Patterns to Remember:

- Use null markers to preserve structure in tree serialization.
- Store mappings explicitly for unique object encodings (like URLs).
- For resource pools, sets and queues make O(1) allocation and checking possible.

Common Pitfalls:

- Forgetting to handle nulls or missing children in trees.
- Not guarding against code collisions in URL encoding.
- Leaking information with sequential assignment (e.g., URLs).
- Double-adding or leaking released numbers in resource pools.

Action List

- Try implementing all three problems yourself (especially the last two).
- For TinyURL, experiment with deterministic vs. random code assignment.
- For Phone Directory, can you do it with just an array and a pointer?
- Explore other serialization/encoding problems, like serializing graphs or linked lists.
- Dry-run your code with tricky edge cases (empty trees, all numbers assigned, very long URLs).
- Compare your code to others for style and edge-case handling.
- If you get stuck, review the explanation and break the problem into smaller parts.

Keep practicing! Serialization and encoding pop up everywhere — the more comfortable you become, the more prepared you'll be for any interview curveball.