Solution: Serialize and Deserialize Binary Tree

Let's solve the Serialize and Deserialize Binary Tree problem using the Tree Depth-first Search pattern.

We'll cover the following

Statement

Serialize a given binary tree to a file and deserialize it back to a tree. Make sure that the original and the deserialized trees are identical.

- **Serialize:** Write the tree to a file.
- **Deserialize:** Read from a file and reconstruct the tree in memory.

Serialize the tree into a list of integers, and then, deserialize it back from the list to a tree. For simplicity's sake, there's no need to write the list to the files.

Constraints:

- The number of nodes in the tree is in the range $[0, 10^4]$.
- $-1000 \leq$ `Node.value` ≤ 1000

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

An initial idea may be to store one of the traversals of the tree into a file when serializing a tree and read that traversal back to create a tree when deserializing. However, any one of the traversals is not unique. That is, two different trees can have the same in-order traversal. The same goes for pre-order or post-order traversal as well. As a simple example, consider a right-slanting degenerate tree and a left-slanting degenerate tree. Both of these trees have the same in-order, pre-order as well as post-order traversals, but the are different trees. However, two traversals are sufficient to uniquely represent and reconstruct a binary tree.

For serialization, this approach will store both the inorder and preorder traversal and place a delimiter to separate them.

For deserialization, pick each node from the preorder traversal, make it root, and find its index in the inorder traversal. The nodes to the left of that index will be the part of the left subtree, and the nodes to the right of that index will be the part of the right subtree.

We got the required solution but at what cost? Can we avoid using twice the space? Can we avoid using tree traversal twice?

Optimized approach using tree depth-first search

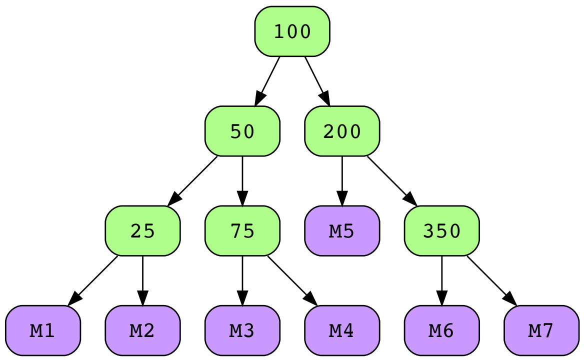
We can save the extra space by storing the preorder traversal of the tree and a marker for NULL pointers. The marker is used to identify the NULL nodes of the tree, which is helpful in deserializing the binary tree. We use the preorder (root, left, right) traversal to serialize the whole tree. While deserializing the binary tree, it's easy to reconstruct the entire tree using preorder traversal. We start by creating the root, and then its left and right children, respectively. The preorder traversal comes under depth-first search.

Here is how we'll implement this algorithm:

We'll serialize the tree as follows:

- We perform a **depth-first** traversal and serialize individual nodes to the stream.
- We use a **preorder** (root, left, right) traversal here.
- We serialize the marker to represent a NULL pointer, which helps deserialize the tree.

Consider the binary tree below as an example:



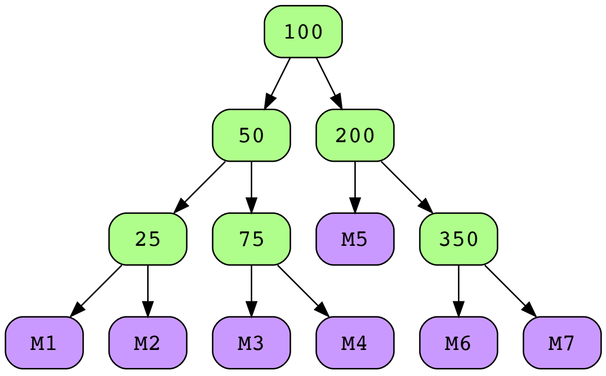
The markers $(M*)$ are added to this tree to represent NULL nodes. The number with each marker, such as 1 in $M1$ or 2 in $M2$, represents only the relative position of a marker in the stream.

The serialized tree (preorder traversal) from the example above looks like the list below:

100	50	25	M1	M2	75	M3	M4	200	M5	350	M6	M7

To deserialize the tree, we again use the preorder traversal and create a new node for every non-marker node. Encountering a marker indicates a NULL node.

Let's run this approach on the above tree:



Preorder traversal:

1 of 28

Let’s look at the code for this solution below:

Java

main.java

BinaryTree.java

TreeNode.java

```java
class SerializeDeserialize {
    // Initializing our marker as the max possible int value
    private static final String MARKER = "M";
    private static int m = 1;

    private static void serializeRec(TreeNode<Integer> node, List<String> st
        // Adding marker to stream if the node is null
        if (node == null) {
            String s = Integer.toString(m);
            stream.add(MARKER+s);
            m = m + 1;
            return;
        }

        // Adding node to stream
        stream.add(String.valueOf(node.data));

        // Doing a pre-order tree traversal for serialization
        serializeRec(node.left, stream);
        serializeRec(node.right, stream);
    }

    // Function to serialize tree into a list.
    public static List<String> serialize(TreeNode<Integer> root) {
        List<String> stream = new ArrayList<>();
        serializeRec(root, stream);
        return stream;
```

Solution summary

To recap, the solution to this problem can be divided into the following two parts:

- Perform a depth-first traversal and serialize individual nodes to the stream.
    - Serialize the marker to represent a NULL pointer as it will help to deserialize the tree.
- Deserialize the tree using preorder traversal. During traversal, create a new node for every non-marker node.

Time complexity

The time complexity of the serialization function is $O(n)$, and the time complexity of the deserialization function is also $O(n)$, where n is the number of nodes in the binary tree. Therefore, the overall time complexity of this solution is $O(n)$.

Space complexity

The space complexity of this solution is $O(h)$, where h is the height of the tree. This is because our recursive algorithm uses space on the call stack, which can grow to the height (h) of the binary tree. The complexity will be $O(\log n)$ for a balanced tree and $O(n)$ for a degenerate tree.

Additional thoughts

Furthermore, if we know that our tree is almost a complete binary tree, we can serialize it similarly to how heap data structures are stored in an array. We can use the two rules below to serialize and deserialize data:

- The children of a node at index i are located at indices $2 * i$ and $(2 * i) + 1$.
- The parent of a node at index i is located at index $i/2$.

Back
Serialize and Deseriali...

Next
Invert Binary Tree

Mark as