449. Serialize and Deserialize BST

May 28, 2019 | 33.6K views

Average Rating: 4.22 (36 votes)

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

Design an algorithm to serialize and deserialize a **binary search tree**. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary search tree can be serialized to a string and this string can be deserialized to the original tree structure.

The encoded string should be as compact as possible.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

Solution

How to make the encoded string as compact as possible

This question is similar to the Google interview question discussed last week.

To serialize a binary tree means to:

- Encode tree structure.
- Encode node values.
- Choose delimiters to separate the values in the encoded string.

Hence there are three axes of optimisation here.

Approach 1: Postorder traversal to optimise space for the tree structure.

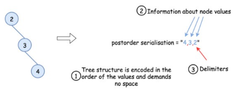
Intuition

Let's use here the fact that BST could be constructed from preorder or postorder traversal only. Please check this article for the detailed discussion. In brief, it's a consequence of two facts:

- Binary tree could be constructed from preorder/postorder and inorder traversal.
- Inorder traversal of BST is an array sorted in the ascending order: `inorder = sorted(preorder)`.

That means that BST structure is already encoded in the preorder or postorder traversal and hence they are both suitable for the compact serialization.

Serialization could be easily implemented with both strategies, but for optimal deserialization better to choose the postorder traversal because member/global/static variables are not allowed here.



Implementation

Java | Python — Copy

```
class Codec:
    def serialize(self, root):
        """
        Encodes a tree to a single string.
        """
        def postorder(root):
            return postorder(root.left) + postorder(root.right) + [root.val] if root else []
        return ' '.join(map(str, postorder(root)))

    def deserialize(self, data):
        """
        Decodes your encoded data to tree.
        """
        def helper(lower = float('-inf'), upper = float('inf')):
            if not data or data[-1] < lower or data[-1] > upper:
                return None

            val = data.pop()
            root = TreeNode(val)
            root.right = helper(val, upper)
            root.left = helper(lower, val)
            return root

        data = [int(x) for x in data.split(' ') if x]
        return helper()
```

Complexity Analysis

- Time complexity : $O(N)$ both for serialization and deserialization. Let's compute the solution with the help of master theorem $T(N) = aT\left(\frac{N}{b}\right) + \Theta(N^d)$. The equation represents dividing the problem up into a subproblems of size $\frac{N}{b}$ in $\Theta(N^d)$ time. Here one divides the problem in two subproblems $a = 2$, the size of each subproblem (to compute left and right subtree) is a half of initial problem $b = 2$, and all this happens in a constant time $d = 0$. That means that $\log_b(a) > d$ and hence we're dealing with case 1 that means $O(N^{\log_b(a)}) = O(N)$ time complexity.
- Space complexity : $O(N)$, since we store the entire tree. Encoded string: one needs to store $(N-1)$ delimiters, and N node values in the encoded string. Tree structure is encoded in the order of values and uses no space.

Approach 2: Convert int to 4-bytes string to optimise space for node values.

Intuition

Approach 1 works fine with the small node values but starts to consume more and more space in the case of large ones.

For example, the tree `[2,null,3,null,4]` is encoded as a string `"4 3 2"` which uses `5` bytes to store the values and delimiters, `1` byte per value + delimiter. So far everything is fine.

Let's consider now the tree `[12345,null,12346,null,12347]` which is encoded as `"12347 12346 12345"` and consumes `17` bytes to store 3 integers and 2 delimiters. `15` bytes for node values only. At the same time it's known that `4` bytes is enough to store an int value, i.e. `12` bytes should be enough for 3 integers. `15 > 12` and hence the storage of values could be optimised.

> How to do it? Convert each integer into 4-bytes string.

Implementation

Python — Copy

```
class Codec:
    def postorder(self, root):
        return self.postorder(root.left) + self.postorder(root.right) + [root.val] if root else []

    def int_to_str(self, x):
        """
        Encodes integer to bytes string.
        """
        bytes = [(x>>s * 8) & 0xff for s in range(4)]
        bytes.reverse()
        bytes_str = ''.join(bytes)
        return bytes_str

    def serialize(self, root):
        """
        Encodes a tree to a single string.
        """
        int = self.postorder(root)
        int = [self.int_to_str(x) for x in int]
        return ''.join(int)

    def str_to_int(self, bytes_str):
        """
        Decodes bytes string to integer.
        """
        result = 0
        for ch in bytes_str:
```

Complexity Analysis

- Time complexity : $O(N)$ both for serialization and deserialization.
- Space complexity : $O(N)$, since we store the entire tree. Encoded string: one needs $2(N-1)$ bytes for the delimiters, and $4N$ bytes for the node values in the encoded string. Tree structure is encoded in the order of node values and uses no space.

Approach 3: Get rid of delimiters.

Intuition

Approach 2 works well except for delimiter usage.

Since all node values are now encoded as 4-bytes strings, one could just split the encoded string into 4-bytes chunks, convert each chunk back to the integer and proceed further.

Implementation

Java | Python — Copy

```
class Codec:
    def postorder(self, root):
        return self.postorder(root.left) + self.postorder(root.right) + [root.val] if root else []

    def int_to_str(self, x):
        """
        Encodes integer to bytes string
        """
        bytes = [(x>>s * 8) & 0xff for s in range(4)]
        bytes.reverse()
        bytes_str = ''.join(bytes)
        return bytes_str

    def serialize(self, root):
        """
        Encodes a tree to a single string.
        """
        int = [self.int_to_str(x) for x in self.postorder(root)]
        return ''.join(int)

    def str_to_int(self, bytes_str):
        """
        Decodes bytes string to integer.
        """
        result = 0
        for ch in bytes_str:
            result = result * 256 + ord(ch)
```

Complexity Analysis

- Time complexity : $O(N)$ both for serialization and deserialization.
- Space complexity : $O(N)$, since we store the entire tree. Encoded string: no delimiters, no additional space for the tree structure, just $4N$ bytes for the node values in the encoded string.

Rate this article: