Previous Next

106. Construct Binary Tree from Inorder and Postorder Traversal

May 24, 2019 | 31.1K views

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

For example, given

```
inorder = [9,3,15,20,7]
postorder = [9,15,7,20,3]
```

Return the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

Solution

How to traverse the tree

There are two general strategies to traverse a tree:

- *Depth First Search* (`DFS`)

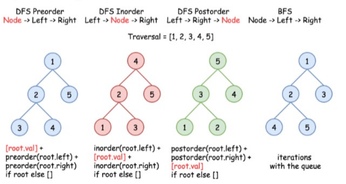
  In this strategy, we adopt the `depth` as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

  The DFS strategy can further be distinguished as `preorder`, `inorder`, and `postorder` depending on the relative order among the root node, left node and right node.

- *Breadth First Search* (`BFS`)

  We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are enumerated in the order you visit them, please follow `1-2-3-4-5` to compare different strategies.



> In this problem one deals with inorder and postorder traversals.

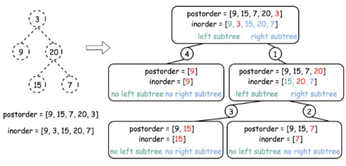
Approach 1: Recursion

How to construct the tree from two traversals: inorder and preorder/postorder/etc

Problems like this one are often at Facebook interviews, and could be solved in $O(N)$ time:

- Start from not inorder traversal, usually it's preorder or postorder one, and use the traversal picture above to define the strategy to pick the nodes. For example, for preorder traversal the *first* value is a root, then its left child, then its right child, etc. For postorder traversal the *last* value is a root, then its right child, then its left child, etc.

- The value picked from preorder/postorder traversal splits the inorder traversal into left and right subtrees. The only information one needs from inorder - if the current subtree is empty (= return `None`) or not (= continue to construct the subtree).



Algorithm

- Build hashmap `value -> its index` for inorder traversal.

- Return `helper` function which takes as the arguments the left and right boundaries for the current subtree in the inorder traversal. These boundaries are used only to check if the subtree is empty or not. Here is how it works `helper(in_left = 0, in_right = n - 1)`:

  - If `in_left > in_right`, the subtree is empty, return `None`.

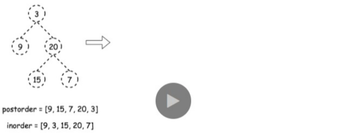
  - Pick the last element in postorder traversal as a root.

  - Root value has index `index` in the inorder traversal, elements from `in_left` to `index - 1` belong to the left subtree, and elements from `index + 1` to `in_right` belong to the right subtree.

  - Following the postorder logic, proceed recursively first to construct the right subtree `helper(index + 1, in_right)` and then to construct the left subtree `helper(in_left, index - 1)`.

  - Return `root`.

Implementation



Java | Python — Copy

```python
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:
        def helper(in_left, in_right):
            # if there is no elements to construct subtrees
            if in_left > in_right:
                return None

            # pick up the last element as a root
            val = postorder.pop()
            root = TreeNode(val)

            # root splits inorder list
            # into left and right subtrees
            index = idx_map[val]

            # build right subtree
            root.right = helper(index + 1, in_right)
            # build left subtree
            root.left = helper(in_left, index - 1)
            return root

        # build a hashmap value -> its index
        idx_map = {val:idx for idx, val in enumerate(inorder)}
        return helper(0, len(inorder) - 1)
```

Complexity Analysis

- Time complexity : $O(N)$. 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 subproblemes `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.

Rate this article:

Previous Next

Comments: 6 — Sort By

Type comment here... (Markdown is supported)

Preview Post

SoumavaBan ★ 72 September 2, 2019 2:22 AM
What is the reason that we have to construct the right sub-tree first and then the left sub-tree?
16 Share Reply
SHOW 5 REPLIES

lenchen1112 ★ 973 February 2, 2020 8:16 PM Report
Refer to the greate answer of @StefanPochmann in this post.
It's possible to reach O(N) time without the index map.

```
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:
```
Read More
4 Share Reply
SHOW 1 REPLY

porikant ★ 50 January 24, 2020 1:10 AM
Since we are popping nodes from the end from post-order list, we come across nodes in the order: node, node.right and then node.left
3 Share Reply

antolerin ★ 1 April 2, 2020 4:09 PM Report
My js solution

```
var splitArray = (arr, splitter) => {
    const index = arr.indexOf(splitter);
    const left = arr.slice(0, index);
```
Read More
1 Share Reply

piergiacomi ★ 0 a day ago
Why is the size of each subproblem half the size of the initial problem? If you have a very unbalanced graph you can have two subproblems one of size 999999 (N-1) and the other with size 1, in that case isn't b very close to 1 and not 2?
0 Share Reply

Aprifel ★ 0 June 8, 2020 10:08 PM
Just have one question (Python), is building a hashmap more performant than using the list `index()` function? If so, could you give me some hints about why?
0 Share Reply
SHOW 1 REPLY