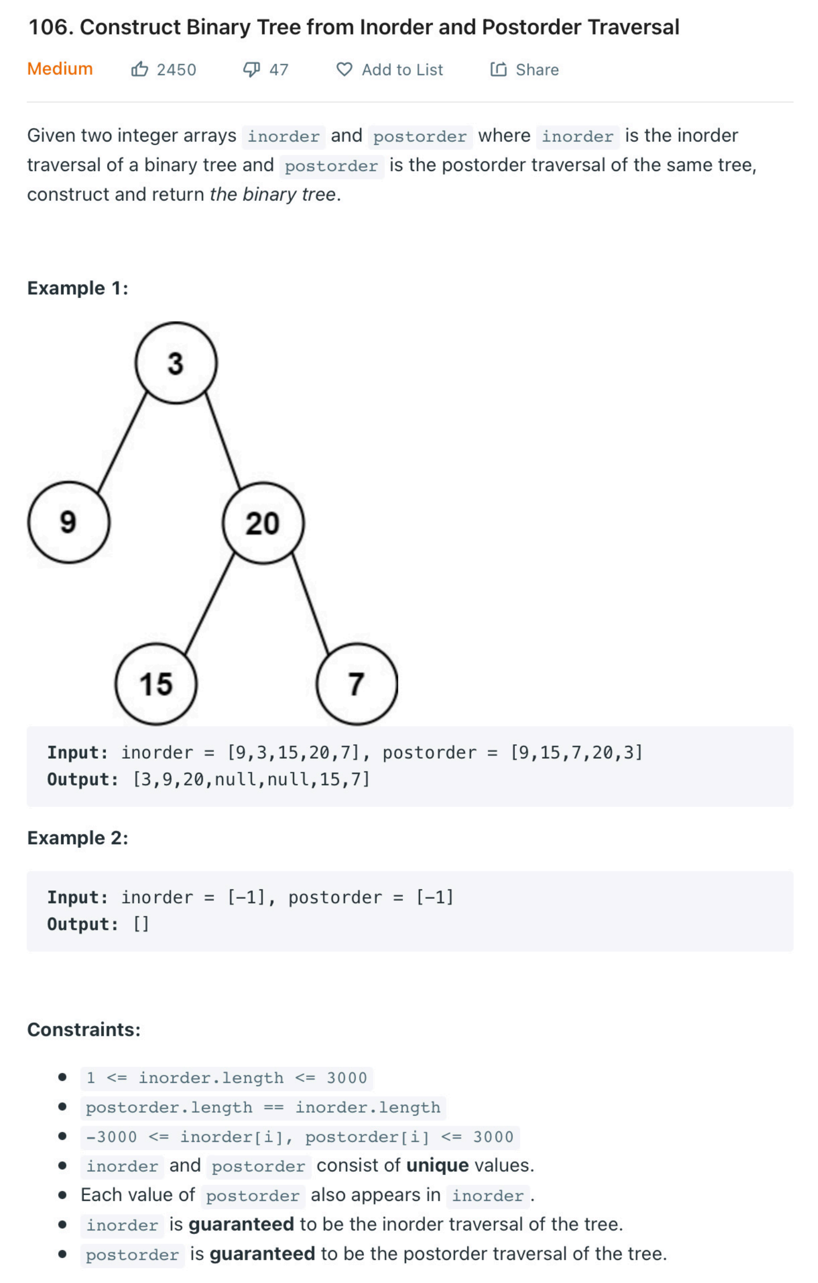


106. Construct Binary Tree from Inorder and Postorder Traversal

Given two integer arrays `inorder` and `postorder` where `inorder` is the inorder traversal of a binary tree and `postorder` is the postorder traversal of the same tree, construct and return *the binary tree*.

Example 1:

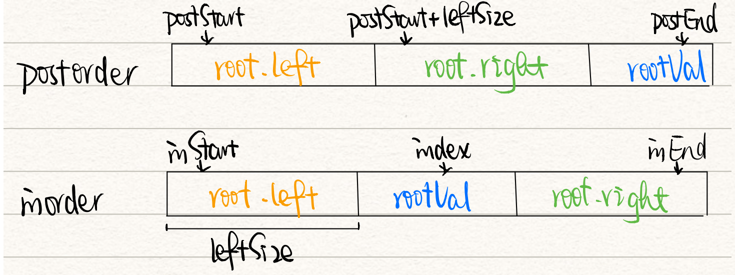
```
Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]
Output: [3,9,20,null,null,15,7]
```

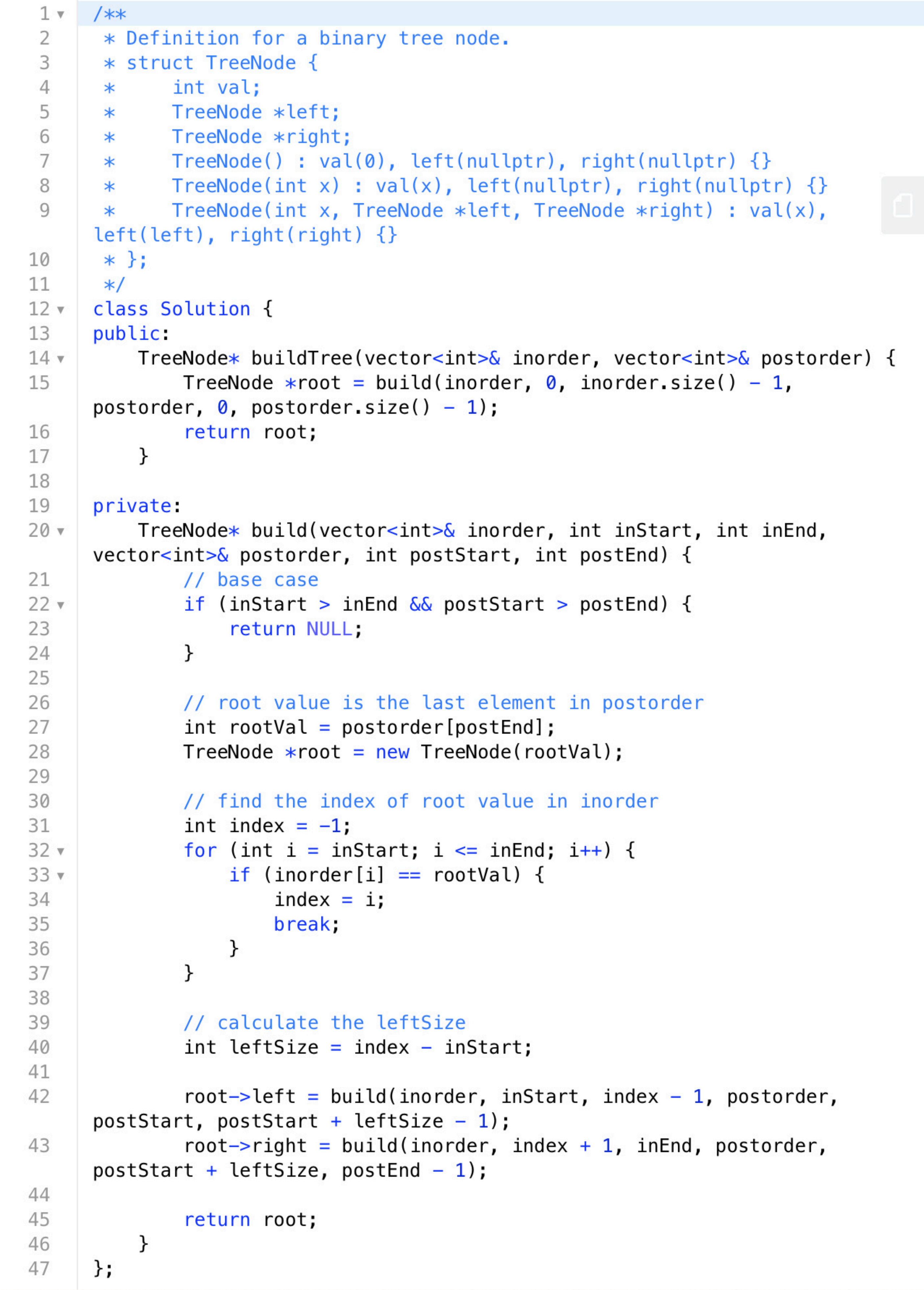
Example 2:

```
Input: inorder = [-1], postorder = [-1]
Output: []
```

Constraints:

- `1 <= inorder.length <= 3000`
- `postorder.length == inorder.length`
- `-3000 <= inorder[i], postorder[i] <= 3000`
- `inorder` and `postorder` consist of **unique** values.
- Each value of `postorder` also appears in `inorder`.
- `inorder` is **guaranteed** to be the inorder traversal of the tree.
- `postorder` is **guaranteed** to be the postorder traversal of the tree.





```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        TreeNode *root = build(inorder, 0, inorder.size() - 1, postorder, 0, postorder.size() - 1);
        return root;
    }

private:
    TreeNode* build(vector<int>& inorder, int inStart, int inEnd, vector<int>& postorder, int postStart, int postEnd) {
        // base case
        if (inStart > inEnd && postStart > postEnd) {
            return NULL;
        }

        // root value is the last element in postorder
        int rootVal = postorder[postEnd];
        TreeNode *root = new TreeNode(rootVal);

        // find the index of root value in inorder
        int index = -1;
        for (int i = inStart; i <= inEnd; i++) {
            if (inorder[i] == rootVal) {
                index = i;
                break;
            }
        }

        // calculate the leftSize
        int leftSize = index - inStart;

        root->left = build(inorder, inStart, index - 1, postorder, postStart, postStart + leftSize - 1);
        root->right = build(inorder, index + 1, inEnd, postorder, postStart + leftSize, postEnd - 1);

        return root;
    }
};
```