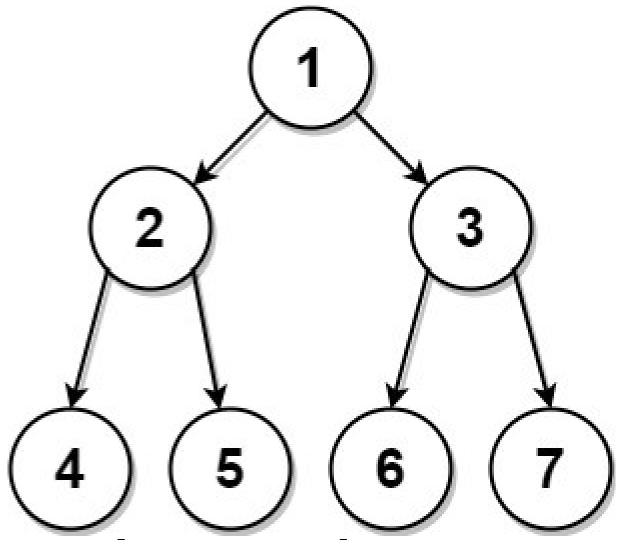
# 889. Construct Binary Tree from Preorder and Postorder Traversal

Given two integer arrays, `preorder` and `postorder` where `preorder` is the preorder traversal of a binary tree of **distinct** values and `postorder` is the postorder traversal of the same tree, reconstruct and return *the binary tree*.

If there exist multiple answers, you can **return any** of them.

**Example 1:**

```
Input: preorder = [1,2,4,5,3,6,7],
postorder = [4,5,2,6,7,3,1]
```




```
Output: [1,2,3,4,5,6,7]
```

**Example 2:**

```
Input: preorder = [1], postorder = [1]
Output: [1]
```

**Constraints:**

- 1 <= preorder.length <= 30
- 1 <= preorder[i] <= preorder.length
- All the values of preorder are **unique**.
- postorder.length == preorder.length
- 1 <= postorder[i] <= postorder.length
- All the values of postorder are **unique**.
- It is guaranteed that preorder and postorder are the preorder traversal and postorder traversal of the same binary tree.

## 889. Construct Binary Tree from Preorder and Postorder Traversal

```
/**
 * 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) {}
 * };
 */
/*
    Recursion: Divide and Conquer
    Time complexity: O(n)
    Space complexity: O(2n)
*/
class Solution {
    public:
        TreeNode* constructFromPrePost(std::vector<int>& preorder, std::vector<int>& postorder){
            int n=preorder.size();

            // nodes's values are distincts, we can map the postorder values a conctant time lookup
            // 1 <= preorder.length <= 30
            // 1 <= preorder[i] <= preorder.length
            std::vector<int> post_index(31);
            for(int i=0;i<n;++i){
                post_index[postorder[i]]=i;
            }
```

```
        // Recursive function to recover the binary tree from a preorder and a postorder DFS
        // traversal
        auto recover=[&](int pre_start,int pre_end,int post_start,auto& self)->TreeNode*{
            // Base case: If there are no nodes to process, return NULL
            if(pre_start>pre_end) return nullptr;

            // Base case: If only one node is left, return that node
            if(pre_start==pre_end) return new TreeNode(preorder[pre_start]);

            // Obtain the root of the left subtree from the preorder traversal
            int left_subtree_root=preorder[pre_start+1];

            // Obtain the left subtree size from the postorder traversal
            int left_subtree_size=post_index[left_subtree_root]-post_start+1;

            // Build the root and its left and right subtree
            // Root is obtained from preorder travsersal
            // Based on the above data, we can determine the elements (ranges) of both left and right
            // subtrees
            TreeNode* root=new TreeNode(preorder[pre_start],
                self(pre_start+1,pre_start+left_subtree_size,post_start,self),
                self(pre_start+left_subtree_size+1,pre_end,post_start+left_subtree_size,self));

            return root;
        };

        return recover(0,n-1,0,recover);

    }
};
```