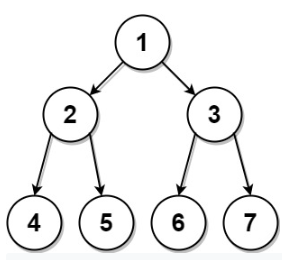
<https://leetcode.com/problems/construct-binary-tree-from-preorder-and-postorder-traversal/>

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.

**Attempt 1: 2022-10-28**

**Solution 1:  Recursive traversal / Divide and Conquer (120min, too long to figure out we can use left subtree root value (very next to whole tree root value on preorder array) which get from preorder to get same value's index mapping on postorder array,  the range between start node and index of subtree root index on postorder array is the size of left subtree)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {

return helper(preorder, postorder, 0, preorder.length - 1, 0, postorder.length - 1);

}

private TreeNode helper(int[] preorder, int[] postorder, int preorderStart, int preorderEnd, int postorderStart, int postorderEnd) {

if(preorderStart > preorderEnd || postorderStart > postorderEnd) {

return null;

}

// Base case: when range limit to one element, we have to create a new node

if(preorderStart == preorderEnd) {

return new TreeNode(preorder[preorderStart]);

}

int rootVal = preorder[preorderStart];

TreeNode root = new TreeNode(rootVal);

// left subtree root is very next to whole tree root on preorder array

int leftSubtreeStartInPreorder = preorderStart + 1;

int leftSubtreeRootVal = preorder[leftSubtreeStartInPreorder];

int leftSubtreeRootValIndexInPostorder = 0;

for(int i = postorderStart; i <= postorderEnd; i++) {

if(postorder[i] == leftSubtreeRootVal) {

leftSubtreeRootValIndexInPostorder = i;

break;

}

}

int leftSubtreeEndInPreorder = leftSubtreeStartInPreorder + (leftSubtreeRootValIndexInPostorder - postorderStart);

root.left = helper(preorder, postorder, leftSubtreeStartInPreorder, leftSubtreeEndInPreorder, postorderStart, leftSubtreeRootValIndexInPostorder);

root.right = helper(preorder, postorder, leftSubtreeEndInPreorder + 1, preorderEnd, leftSubtreeRootValIndexInPostorder + 1, postorderEnd - 1);

return root;

}

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/construct-binary-tree-from-preorder-and-postorder-traversal/discuss/161268/C++JavaPython-One-Pass-Real-O(N)/311443>

When we meet a node value pre[i] equals to the current post[j], it means we have completed building the subtree of pre[i]. So we should not continue to add child nodes to that subtree. Instead, we should pop that subtree and continues to the path where we can add child nodes.

**Refer to**

<https://leetcode.com/problems/construct-binary-tree-from-preorder-and-postorder-traversal/discuss/161268/C++JavaPython-One-Pass-Real-O(N)/186220>

Explaination:

pre : 1(245)(367)

post : (452)(673)1

pre: 2 (4)(5) 3(6)(7)

post:(4)(5)2 (6)(7)3

So each time the first node of pre array 1 and the last node of post array 1 if they are equal means we just found a root, then the next node 2 in pre array will be the root of the sub tree, then since we use HashMap to store all the nodes' index, it will be very easy to find the index of 2 in post array, then we cut the post array into half by this index, then do the recursion, until we have only one node left which is the base case for the recursion..

the key that made my feel stuck is I did not realize that 2 will the root of sub tree at the very start...

**Refer to**

<https://leetcode.com/problems/construct-binary-tree-from-preorder-and-postorder-traversal/discuss/161372/Java-Python-Divide-and-Conquer-with-Explanation>

The problem is easier to solve if we decrease it into sub problems using **Divide and Conquer**.

e.g. Given preorder : 1 2 4 5 3 6; postorder: 4 5 2 6 3 1.

We see it as preorder : 1 (2 4 5) (3 6); postorder: (4 5 2) (6 3) 1 [to be explained afterwards]

That can be decreased to subproblems A, B, C:

A. preorder : 1; postorder: 1 =>

1

B. preorder : (2 4 5); postorder: (4 5 2) =>

2

/ \

4 5

C. preorder : (3 6); postorder: (6 3) =>

3

/

6 or

3

\

6

\* Then we conquer the subproblems => A.left = B; A.right = C;

1

/ \

2 3

/ \ /

4 5 6

If we observe parameters in each recursion above:

preStart: 0, preEnd: 5, postStart: 0, postEnd: 5

preStart: 1, preEnd: 3, postStart: 0, postEnd: 2

preStart: 4, preEnd: 5, postStart: 3, postEnd: 4

For the commented, [to be explained afterwards], how do we decrease a problem?

That is, 1 is root and 2 is its left child. Since 2 is the root of the left subtree, all elements in front of 2 in post[] must be in the left subtree also.

We recursively follow the above approach.

Please note that pre[preStart + 1] may also be the root of the right subtree if there is no left subtree at all in the orginal tree. Since we are asked to generate one possible original tree, I assume pre[preStart + 1] to be the left subtree root always.

**Code**

public TreeNode constructFromPrePost(int[] pre, int[] post) {

return constructFromPrePost(pre, 0, pre.length - 1, post, 0, pre.length - 1);

}

private TreeNode constructFromPrePost(int[] pre, int preStart, int preEnd, int[] post, int postStart, int postEnd) {

// Base cases.

if (preStart > preEnd) {

return null;

}

if (preStart == preEnd) {

return new TreeNode(pre[preStart]);

}

// Build root.

TreeNode root = new TreeNode(pre[preStart]);

// Locate left subtree.

int leftSubRootInPre = preStart + 1;

int leftSubRootInPost = findLeftSubRootInPost(pre[leftSubRootInPre], post, postStart, postEnd);

int leftSubEndInPre = leftSubRootInPre + (leftSubRootInPost - postStart);

// Divide.

TreeNode leftSubRoot = constructFromPrePost(pre, leftSubRootInPre, leftSubEndInPre,

post, postStart, leftSubRootInPost);

TreeNode rightSubRoot = constructFromPrePost(pre, leftSubEndInPre + 1, preEnd,

post, leftSubRootInPost + 1, postEnd - 1);

// Conquer.

root.left = leftSubRoot;

root.right = rightSubRoot;

return root;

}

private int findLeftSubRootInPost(int leftSubRootVal, int[] post, int postStart, int postEnd) {

for (int i = postStart; i <= postEnd; i++) {

if (post[i] == leftSubRootVal) {

return i;

}

}

throw new IllegalArgumentException();

}

**Solution 2: Recursive traversal / Divide and Conquer (10 min, promote with map to find root index in O(1))**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

for(int i = 0; i < postorder.length; i++) {

map.put(postorder[i], i);

}

return helper(map, preorder, postorder, 0, preorder.length - 1, 0, postorder.length - 1);

}

private TreeNode helper(Map<Integer, Integer> map, int[] preorder, int[] postorder, int preorderStart, int preorderEnd, int postorderStart, int postorderEnd) {

if(preorderStart > preorderEnd || postorderStart > postorderEnd) {

return null;

}

// Base case: when range limit to one element, we have to create a new node

if(preorderStart == preorderEnd) {

return new TreeNode(preorder[preorderStart]);

}

int rootVal = preorder[preorderStart];

TreeNode root = new TreeNode(rootVal);

// left subtree root is very next to whole tree root on preorder array

int leftSubtreeStartInPreorder = preorderStart + 1;

int leftSubtreeRootVal = preorder[leftSubtreeStartInPreorder];

int leftSubtreeRootValIndexInPostorder = map.get(leftSubtreeRootVal);

int leftSubtreeEndInPreorder = leftSubtreeStartInPreorder + (leftSubtreeRootValIndexInPostorder - postorderStart);

root.left = helper(map, preorder, postorder, leftSubtreeStartInPreorder, leftSubtreeEndInPreorder, postorderStart, leftSubtreeRootValIndexInPostorder);

root.right = helper(map, preorder, postorder, leftSubtreeEndInPreorder + 1, preorderEnd, leftSubtreeRootValIndexInPostorder + 1, postorderEnd - 1);

return root;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 3: Iterative traversal with Stack (60 min, not able to come up to the idea)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {

Stack<TreeNode> stack = new Stack<TreeNode>();

TreeNode root = new TreeNode(preorder[0]);

stack.push(root);

int j = 0;

for(int i = 1; i < preorder.length; i++) {

TreeNode node = new TreeNode(preorder[i]);

while(stack.peek().val == postorder[j]) {

stack.pop();

j++;

}

if(stack.peek().left == null) {

stack.peek().left = node;

} else {

stack.peek().right = node;

}

stack.push(node);

}

return root;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Stack status update step by step**

The backend logic is when peek.val == post[j] means we find all nodes for current subtree, current subtree's root node is at the peek, to construct next subtree, we have to remove current subtree's root node by pop it out

e.g

3

/ \

9 20

/ \

15 7

preorder = [3,9,20,15,7]

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

-------------------------------------------------------------------------------------------------

Iterate on 'preorder' because we can obtain 'root' first

Stack status: The backend logic is when peek.val == post[j] means we find all nodes for current subtree, current subtree's root node is at the peek, to construct next subtree, we have to remove current subtree's root node by pop it out

=== === ====

9 push 9 9 pop 9 20 push 20

=== --- --- ----

push root -> 3 ---> check 9 ---> 3 ---> check 20 ---> 3 ---------------------> 3

=== i=1,j=0 === i=2,j=0 === ====

peek.val=3 peek.val=9

3 post[j]=9 3 post[j]=9 3

3.left=null / peek.val==post[j] / \

3.left=9 9 finish subtree build on subtree root=9 9 20

remove current subtree root for next

subtree build, pop 9,j++=1

3.left!=null

3.right=20

==== ==== ====

15 push 15 15 pop 15 7 push 7

---- ---- ----

20 20 20

---- ---- ----

---> check 15 ---> 3 ---> check 7 --------------------------------> 3 -------> 3

i=3,j=1 ==== i=4,j=1 ==== ====

peek.val=20 peek.val=15

post[j]=15 3 post[j]=15 3

20.left=null / \ peek.val=post[j] / \

20.left=15 9 20 finish subtree build on subtree root=15 9 20

/ remove current subtree root for next / \

15 subtree build, pop 15,j++=2 15 7

20.left!=null

20.right=7

**Refer to**

<https://leetcode.com/problems/construct-binary-tree-from-preorder-and-postorder-traversal/discuss/161268/C++JavaPython-One-Pass-Real-O(N)/209048>

**Iterative Solution**

We will **preorder** generate TreeNodes, push them to stack and **postorder** pop them out.

1. Iterate on pre array and construct node one by one.
2. stack save the current path of tree.
3. node = new TreeNode(pre[i]), if not left child, add node to the left. otherwise add it to the right.
4. **If we meet a same value in the pre and post, it means we complete the construction for current subtree. We pop it from stack.**

public TreeNode constructFromPrePost(int[] pre, int[] post) {

Stack<TreeNode> stack = new Stack<>();

TreeNode root = new TreeNode(pre[0]);

stack.push(root);

for (int i = 1, j = 0; i < pre.length; ++i) {

TreeNode node = new TreeNode(pre[i]);

while (stack.peek().val == post[j]) {

stack.pop();

j++;

}

if (stack.peek().left == null) {

stack.peek().left = node;

} else {

stack.peek().right = node;

}

stack.push(node);

}

return root;

}