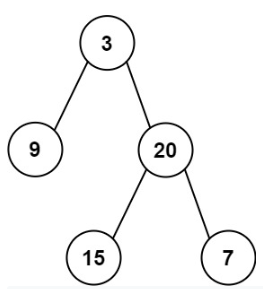
<https://leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/>

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: [-1]

**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.

**Attempt 1: 2022-10-27**

**Solution 1:  Recursive traversal / Divide and Conquer (10min)**

/\*\*

\* 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 buildTree(int[] inorder, int[] postorder) {

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

}

private TreeNode helper(int[] inorder, int[] postorder, int rootIndexInPostorder, int inorderStart, int inorderEnd) {

if(inorderStart > inorderEnd) {

return null;

}

int rootVal = postorder[rootIndexInPostorder];

TreeNode root = new TreeNode(rootVal);

int rootIndexInInorder = 0;

for(int i = inorderStart; i <= inorderEnd; i++) {

if(inorder[i] == rootVal) {

rootIndexInInorder = i;

break;

}

}

// Compare with L105.Construct Binary Tree from Preorder and Inorder Traversal

// Since we have postorder instead of preorder in L106, since root position stored at the end

// of each section in postorder recursively,correspondingly we need to find right subtree size

// instead of left subtree size to locate root index in each recursion

//int leftSubtreeSize = rootIndexInInorder - inorderStart;

int rightSubtreeSize = inorderEnd - rootIndexInInorder;

// Based on postorder nature, similar to above find right subtree size instead of left one,

// we will firstly build right subtree rather than left subtree

root.right = helper(inorder, postorder, rootIndexInPostorder - 1, rootIndexInInorder + 1, inorderEnd);

root.left = helper(inorder, postorder, rootIndexInPostorder - rightSubtreeSize - 1, inorderStart, rootIndexInInorder - 1);

return root;

}

}

**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 buildTree(int[] inorder, int[] postorder) {

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

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

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

}

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

}

private TreeNode helper(Map<Integer, Integer> map, int[] inorder, int[] postorder, int rootIndexInPostorder, int inorderStart, int inorderEnd) {

if(inorderStart > inorderEnd) {

return null;

}

int rootVal = postorder[rootIndexInPostorder];

TreeNode root = new TreeNode(rootVal);

int rootIndexInInorder = map.get(rootVal);

int rightSubtreeSize = inorderEnd - rootIndexInInorder;

root.right = helper(map, inorder, postorder, rootIndexInPostorder - 1, rootIndexInInorder + 1, inorderEnd);

root.left = helper(map, inorder, postorder, rootIndexInPostorder - rightSubtreeSize - 1, inorderStart, rootIndexInInorder - 1);

return root;

}

}

**Solution 3: Iterative traversal with Monotonic Stack (10 min)**

/\*\*

\* 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 buildTree(int[] inorder, int[] postorder) {

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

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

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

}

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

int rootVal = postorder[postorder.length - 1];

TreeNode root = new TreeNode(rootVal);

stack.push(root);

for(int i = postorder.length - 2; i >= 0; i--) {

int nodeVal = postorder[i];

TreeNode node = new TreeNode(nodeVal);

if(map.get(nodeVal) > map.get(stack.peek().val)) {

stack.peek().right = node;

} else {

TreeNode parent = null;

while(!stack.isEmpty() && map.get(nodeVal) < map.get(stack.peek().val)) {

parent = stack.pop();

}

parent.left = node;

}

stack.push(node);

}

return root;

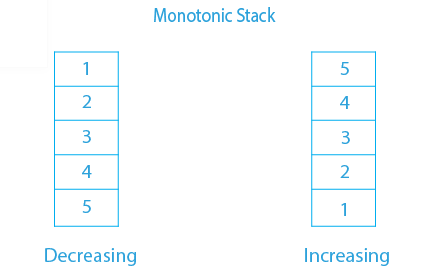
}

}

**Monotonic Stack (nodes on stack strictly obtain increasing index in inorder array) status update step by step**

**What is Monotonic Stack ?**

A monotonic stack is a stack whose elements are monotonically increasing or decreasing. If the top elements of the stack are less than bottom elements , then it is called decreasing stack , else If the top elements of the stack is greater than the bottom elements , then it is called increasing stack.



For Binary Tree we have to compare between the node value's index in inorder to determine the tree structure.

e.g

3

/ \

9 20

/ \

15 7

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

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

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

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

Stack status: Make sure all nodes stored on stack have strict increasing on node value's index in inorder array

====

20 push 20

=== ----

push root -> 3 -> check inorder index of 20(=3) -> 3 -> check inorder index of 7(=4)

=== compare with inorder index ==== compare with inorder index

of 3(=1), its larger, push of 20(=3), 3(=1), its larger, push

3 3

\

20

==== ====

7 push 7 7 pop 7

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

20 20 pop 20 15 push 15

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

-> 3 ------> check inorder index of 15(=2) ------> 3 --------> 3 ---> check inorder index of 9(=0)

==== compare with inorder index ==== ==== compare with inorder index

of 7(=4), 20(=3), its smaller, of 15(=2), its smaller,

first pop then push first pop then push

(last popped out 20 as most recent partent for 15)

3 3

\ \

20 20

\ / \

7 15 7

====

15 pop 15

---- ===

-> 3 pop 3 -> 9 push 9

==== ===

(last popped out 3 as most recent partent for 9)

3

/ \

9 20

/ \

15 7