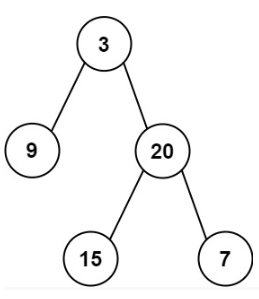
<https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/>

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

**Example 1:**



Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

Output: [3,9,20,null,null,15,7]

**Example 2:**

Input: preorder = [-1], inorder = [-1]

Output: [-1]

**Constraints:**

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

**Attempt 1: 2022-10-26**

**Solution 1:  Recursive traversal / Divide and Conquer (720min, too long to figure out 'rootIndexInPreorder' is necessary and also need 'rootIndexInInorder' to find left and right subtree size)**

/\*\*

\* 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[] preorder, int[] inorder) {

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

}

// 'rootIndexInPreorder' is necessary, not able to write helper function with only

// with 'inorderStart' and 'inorderEnd', since preorder's natural property is each

// section's first element is root, then use inorder's natural property as root elment

// will divide left and right subtree, then search that preorder's each section's

// first element value on inorder to get actual root index, the purpose is get

// left and right subtree size, and use size on preorder to properly cut array

// into sections that correspond to inorder, recursively till no range

private TreeNode helper(int[] preorder, int[] inorder, int rootIndexInPreorder, int inorderStart, int inorderEnd) {

if(inorderStart > inorderEnd) {

return null;

}

int rootVal = preorder[rootIndexInPreorder];

TreeNode root = new TreeNode(rootVal);

int rootIndexInInorder = 0;

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

if(inorder[i] == rootVal) {

rootIndexInInorder = i;

break;

}

}

// e.g

// 8

// / \

// 6 12

// / / \

// 4 10 13

// / \ \ \

// 2 5 11 15

// preorder = [8,6,4,2,5,12,10,11,13,15]

// inorder = [2,4,5,6,8,10,11,12,13,15]

// Round 1:

// helper(preorder, inorder, 0, 0, 9);

// rootVal = preorder[rootIndexInPreorder] = preorder[0] = 8

// rootIndexInInorder = 4

// leftSubtreeSize = 4 - 0 = 4

// preorder = [8 | 6,4,2,5 | 12,10,11,13,15]

// inorder = [2,4,5,6 | 8 | 10,11,12,13,15]

// root.left = helper(preorder, inorder, 0+1[=6 in preorder], 0[=2 in inorder], 4-1[=6 in inorder])

// root.right = helper(preorder, inorder, 0+4+1[=12 in preorder], 4+1[=10 in inorder], 9[=15 in inorder])

// ------------------------------------------------------------------------------------

// Round 2:

// helper(preorder, inorder, 0+1[=6 in preorder], 0[=2 in inorder], 4-1[=6 in inorder])

// = helper(preorder, inorder, 1, 0, 3)

// rootVal = preorder[rootIndexInPreorder] = preorder[1] = 6

// rootIndexInInorder = 3

// leftSubtreeSize = 3 - 0 = 3

// preorder = [8 | [6 | 4,2,5 | null] | 12,10,11,13,15]

// inorder = [[2,4,5 | 6 | null] | 8 | 10,11,12,13,15]

// root.left = helper(preorder, inorder, 1+1[=4 in preorder], 0[=2 in inorder], 3-1[=5 in inorder])

// root.right = helper(preorder, inorder, 1+3+1, 3+1, 3) => since inorderStart > inorderEnd => null

// ------------------------------------------------------------------------------------

// .... etc

int leftSubtreeSize = rootIndexInInorder - inorderStart;

root.left = helper(preorder, inorder, rootIndexInPreorder + 1, inorderStart, rootIndexInInorder - 1);

root.right = helper(preorder, inorder, rootIndexInPreorder + leftSubtreeSize + 1, rootIndexInInorder + 1, inorderEnd);

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[] preorder, int[] inorder) {

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

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

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

}

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

}

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

if(inorderStart > inorderEnd) {

return null;

}

int rootVal = preorder[rootIndexInPreorder];

TreeNode root = new TreeNode(rootVal);

int rootIndexInInorder = map.get(rootVal);

int leftSubtreeSize = rootIndexInInorder - inorderStart;

root.left = helper(map, preorder, inorder, rootIndexInPreorder + 1, inorderStart, rootIndexInInorder - 1);

root.right = helper(map, preorder, inorder, rootIndexInPreorder + leftSubtreeSize + 1, rootIndexInInorder + 1, inorderEnd);

return root;

}

}

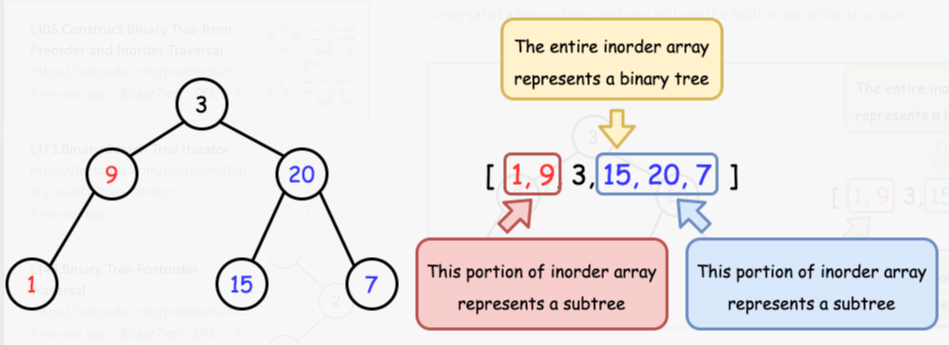
**Refer to**

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

**Overview**

This problem examines your understanding of preorder and inorder binary tree traversals. If you are not familiar with them, feel free to visit our [Explore Cards](https://leetcode.com/explore/learn/card/data-structure-tree/) where you will see all the ways to traverse a binary tree including preorder, inorder, postorder, and level-order traversals :)

A tree has a recursive structure because it has subtrees which are trees themselves. Let's take a look at the inorder traversal of a binary tree, and you will see the built-in recursive structure.



Henceforth, we will leverage this property and find a way to recursively construct the tree.

**Approach 1: Recursion**

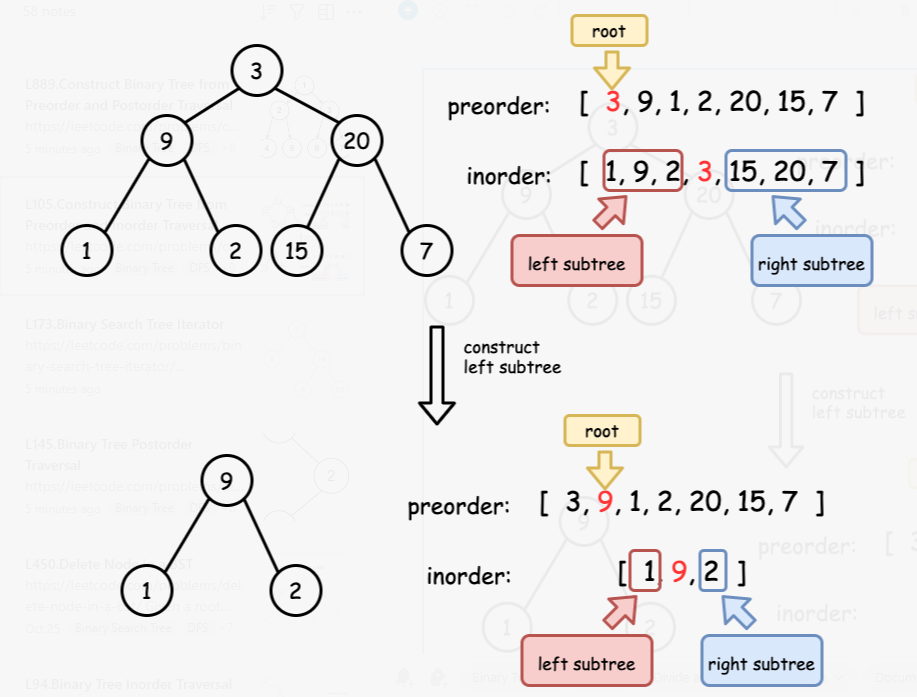
**Intuition**

The two key observations are:

1. Preorder traversal follows Root -> Left -> Right, therefore, given the preorder array preorder, we have easy access to the root which is preorder[0].
2. Inorder traversal follows Left -> Root -> Right, therefore if we know the position of Root, we can recursively split the entire array into two subtrees.

Now the idea should be clear enough. We will design a recursion function: it will set the first element of preorder as the root, and then construct the entire tree. To find the left and right subtrees, it will look for the root in inorder, so that everything on the left should be the left subtree, and everything on the right should be the right subtree. Both subtrees can be constructed by making another recursion call.

It is worth noting that, while we recursively construct the subtrees, we should choose the next element in preorder to initialize as the new roots. This is because the current one has already been initialized to a parent node for the subtrees.



*Figure 2. Always use the next element in preorder to initialize a root.*

**Algorithm**

* Build a hashmap to record the relation of value -> index for inorder, so that we can find the position of root in constant time.
* Initialize an integer variable preorderIndex to keep track of the element that will be used to construct the root.
* Implement the recursion function arrayToTree which takes a range of inorder and returns the constructed binary tree:
  + if the range is empty, return null;
  + initialize the root with preorder[preorderIndex] and then increment preorderIndex;
  + recursively use the left and right portions of inorder to construct the left and right subtrees.
* Simply call the recursion function with the entire range of inorder.

class Solution {

int preorderIndex;

Map<Integer, Integer> inorderIndexMap;

public TreeNode buildTree(int[] preorder, int[] inorder) {

preorderIndex = 0;

// build a hashmap to store value -> its index relations

inorderIndexMap = new HashMap<>();

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

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

}

return arrayToTree(preorder, 0, preorder.length - 1);

}

private TreeNode arrayToTree(int[] preorder, int left, int right) {

// if there are no elements to construct the tree

if (left > right) return null;

// select the preorder\_index element as the root and increment it

int rootValue = preorder[preorderIndex++];

TreeNode root = new TreeNode(rootValue);

// build left and right subtree

// excluding inorderIndexMap[rootValue] element because it's the root

root.left = arrayToTree(preorder, left, inorderIndexMap.get(rootValue) - 1);

root.right = arrayToTree(preorder, inorderIndexMap.get(rootValue) + 1, right);

return root;

}

}

**Complexity analysis**

Let Nbe the length of the input arrays.

* Time complexity : O(N).

Building the hashmap takes O(N)time, as there are Nnodes to add, and adding items to a hashmap has a cost of O(1), so we get N \* O(1) = O(N).

Building the tree also takes O(N)time. The recursive helper method has a cost of O(1)for each call (it has no loops), and it is called *once* for each of the Nnodes, giving a total of O(N).

Taking both into consideration, the time complexity is O(N).

* Space complexity : O(N).

Building the hashmap and storing the entire tree each requires O(N)memory. The size of the implicit system stack used by recursion calls depends on the height of the tree, which is O(N)in the worst case and O(log N)on average. Taking both into consideration, the space complexity is O(N).

**Solution 3: Iterative traversal with Monotonic Stack (60 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[] preorder, int[] inorder) {

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 = preorder[0];

TreeNode root = new TreeNode(rootVal);

stack.push(root);

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

int nodeVal = preorder[i];

TreeNode node = new TreeNode(nodeVal);

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

stack.peek().left = node;

} else {

TreeNode parent = null;

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

parent = stack.pop();

}

parent.right = node;

}

stack.push(node);

}

return root;

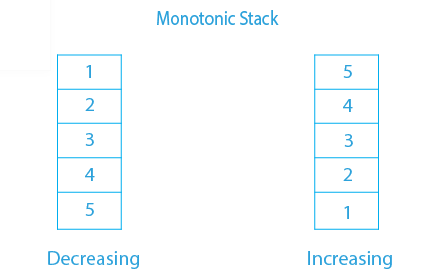
}

}

**Monotonic Stack (nodes on stack strictly obtain decreasing 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.



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

Stack status: maintain monotonic increasing from top(stack peek) to bottom based on node value's index in preorder

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

Note: Since we are constructing the Binary Tree instead of Binary Search Tree(BST), it strictly the comparison between node value's index instead of node value itself

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

For BST we can directly compare between the node value itself to determine the tree structure.

Example below:

e.g

8

/ \

6 12

/ / \

4 10 13

/ \ \ \

2 5 11 15

preorder = [8,6,4,2,5,12,10,11,13,15]

inorder = [2,4,5,6,8,10,11,12,13,15]

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

=== ===

2 push 2 2 pop 2

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

4 push 4 4 4 pop 4 5 push 5

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

6 push 6 6 6 6 6

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

push root -> 8 -> check 6 -> 8 -> check 4 -> 8 -> check 2 -> 8 -> check 5 -> 8 -----> 8

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

(last popped out 4 as most recent partent for 5)

8 8 8 8 8

/ / / /

6 6 6 6

/ / /

4 4 4

/ / \

2 2 5

===

5 pop 5

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

6 pop 6 10 push 10 10 pop 10 11 push 11

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

-> check 12 -> 8 pop 8 --> 12 ---- check 10 ----> 12 -> check 11 -> 12 ------> 12

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

(last popped out 8 as most recent partent for 12) (last popped out 10 as most recent partent for 11)

8 8 8

/ \ / \ / \

6 12 6 12 6 12

/ / / / /

4 4 10 4 10

/ \ / \ / \ \

2 5 2 5 2 5 11

====

11 pop 11

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

-> check 13 -> 12 pop 12 --> 13 --- check 15 --> 13 pop 13 -> 15 push 15

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

(last popped out 12 as most recent partent for 13) (last popped out 13 as most recent partent for 15)

8 8

/ \ / \

6 12 6 12

/ / \ / / \

4 10 13 4 10 13

/ \ \ / \ \ \

2 5 11 2 5 11 15

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

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

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

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

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

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

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

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

===

9 push 9

=== ---

push root -> 3 -> check inorder index of 9(=0) -> 3 -> check inorder index of 20(=3)

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

of 3(=1), its smaller, push of 9(=0), 3(=1), its larger,

first pop then push

3 3

/

9

=== ====

9 pop 9 15 push 15

--- ==== ----

-> 3 pop 3 -> 20 push 20 -> check inorder index of 15(=2) -> 20 -> check inorder index of 7(=4)

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

of 20(=3), its smaller, push of 15(=2), 20(=3), its larger,

first pop then push

(last popped out 3 as most recent partent for 20)

3 3

/ \ / \

9 20 9 20

/

15

====

15 pop 15

---- ===

-> 20 pop 20 -> 7 push 7

==== ===

(last popped out 20 as most recent partent for 7)

3

/ \

9 20

/ \

15 7

**Refer to**

<https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/discuss/34555/The-iterative-solution-is-easier-than-you-think!/117721>

class Solution {

public TreeNode buildTree(int[] preorder, int[] inorder) {

// deal with edge case(s)

if (preorder.length == 0) {

return null;

}

// build a map of the indices of the values as they appear in the inorder array

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

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

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

}

// initialize the stack of tree nodes

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

int value = preorder[0];

TreeNode root = new TreeNode(value);

stack.push(root);

// for all remaining values...

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

// create a node

value = preorder[i];

TreeNode node = new TreeNode(value);

if (map.get(value) < map.get(stack.peek().val)) {

// the new node is on the left of the last node,

// so it must be its left child (that's the way preorder works)

stack.peek().left = node;

} else {

// the new node is on the right of the last node,

// so it must be the right child of either the last node

// or one of the last node's ancestors.

// pop the stack until we either run out of ancestors

// or the node at the top of the stack is to the right of the new node

TreeNode parent = null;

while(!stack.isEmpty() && map.get(value) > map.get(stack.peek().val)) {

parent = stack.pop();

}

parent.right = node;

}

stack.push(node);

}

return root;

}

}