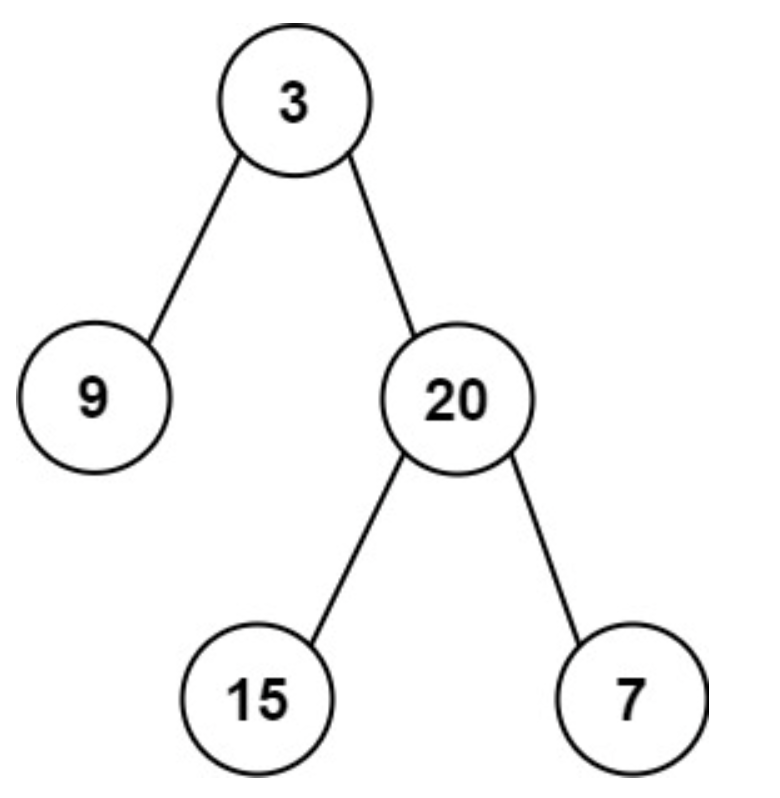
# 题目

给定两个整数数组preorder和inorder ，其中preorder是二叉树的先序遍历，inorder是同一棵树的中序遍历，请构造二叉树并返回其根节点。

示例 1:



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

输出: [3,9,20,null,null,15,7]

示例 2:

输入: preorder = [-1], inorder = [-1]

输出: [-1]

提示:

1 <= preorder.length <= 3000

inorder.length == preorder.length

-3000 <= preorder[i], inorder[i] <= 3000

preorder和inorder均无重复元素

inorder 均出现在preorder

preorder 保证为二叉树的前序遍历序列

inorder 保证为二叉树的中序遍历序列

# 分析

## 方法一：递归法

要解决根据先序遍历和中序遍历构造二叉树的问题，我们可以利用两种遍历的特性，通过递归的方式逐步构建树的每个节点。

解题思路：

1、先序遍历与中序遍历的特性：

先序遍历的顺序是“根节点 -> 左子树 -> 右子树”，因此先序遍历的第一个元素必然是整棵树的根节点。

中序遍历的顺序是“左子树 -> 根节点 -> 右子树”，因此在中序遍历中，根节点左侧的元素都属于左子树，右侧的元素都属于右子树。

2、递归构建逻辑：

从先序遍历中取第一个元素作为当前子树的根节点。

在中序遍历中找到该根节点的位置，从而划分出左子树和右子树的中序遍历范围。

根据左子树的节点数量，在先序遍历中划分出左子树和右子树的先序遍历范围。

递归构建左子树和右子树，并将它们分别连接到当前根节点的左、右指针上。

3、优化查找效率：

为了快速在中序遍历中找到根节点的位置，可以使用哈希表（unordered\_map）存储中序遍历中元素与索引的映射关系，将查找时间从O(n)优化为O(1)。

代码：

/\*\*

\* 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>& preorder, vector<int>& inorder) {

// 构建中序遍历元素到索引的映射，用于快速查找根节点位置

unordered\_map<int, int> inorder\_map;

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

inorder\_map[inorder[i]] = i;

}

// 递归构建二叉树

return build(preorder, 0, preorder.size() - 1,

inorder, 0, inorder.size() - 1,

inorder\_map);

}

private:

TreeNode\* build(vector<int>& preorder, int pre\_start, int pre\_end,

vector<int>& inorder, int in\_start, int in\_end,

unordered\_map<int, int>& inorder\_map) {

// 递归终止条件：当前子树无节点

if (pre\_start > pre\_end || in\_start > in\_end) {

return nullptr;

}

// 先序遍历的第一个元素是当前子树的根节点

int root\_val = preorder[pre\_start];

TreeNode\* root = new TreeNode(root\_val);

// 找到根节点在中序遍历中的位置

int root\_idx = inorder\_map[root\_val];

// 计算左子树的节点数量

int left\_size = root\_idx - in\_start;

// 递归构建左子树：

// 先序遍历范围：[pre\_start+1, pre\_start+left\_size]

// 中序遍历范围：[in\_start, root\_idx-1]

root->left = build(preorder, pre\_start + 1, pre\_start + left\_size,

inorder, in\_start, root\_idx - 1,

inorder\_map);

// 递归构建右子树：

// 先序遍历范围：[pre\_start+left\_size+1, pre\_end]

// 中序遍历范围：[root\_idx+1, in\_end]

root->right = build(preorder, pre\_start + left\_size + 1, pre\_end,

inorder, root\_idx + 1, in\_end,

inorder\_map);

return root;

}

};

解释

1、哈希表映射：inorder\_map存储中序遍历中每个元素对应的索引，使得查找根节点位置的操作可以在`O(1)`时间内完成，避免了每次递归都线性遍历中序数组的低效性。

2、递归函数参数：

- pre\_start和pre\_end：当前子树在先序遍历中的起始和结束索引。

- in\_start和in\_end：当前子树在中序遍历中的起始和结束索引。

- 函数返回当前子树的根节点。

3、左子树与右子树的划分：

根节点在中序遍历中的索引为root\_idx，因此左子树的节点数量为left\_size = root\_idx - in\_start。

左子树的先序遍历范围是[pre\_start+1, pre\_start+left\_size]（紧跟根节点的left\_size个元素），中序遍历范围是[in\_start, root\_idx-1]（根节点左侧的元素）。

右子树的先序遍历范围是[pre\_start+left\_size+1, pre\_end]（剩余元素），中序遍历范围是[root\_idx+1, in\_end]（根节点右侧的元素）。

4、时间复杂度：O(n)，其中n是节点数量。每个节点被构建一次，哈希表的插入和查找操作均为O(1)。

5、空间复杂度：O(n)，主要用于存储哈希表和递归栈（最坏情况下树为链状，递归深度为n）。

## 方法二：迭代法