# Question

Given inorder and postorder traversal of a tree, construct the binary tree.

**Note:**  
You may assume that duplicates do not exist in the tree.

For example, given

inorder = [9,3,15,20,7]

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

Return the following binary tree:

3

/ \

9 20

/ \

15 7

# Solution

#### **How to traverse the tree**

There are two general strategies to traverse a tree:

* Depth First Search (DFS)

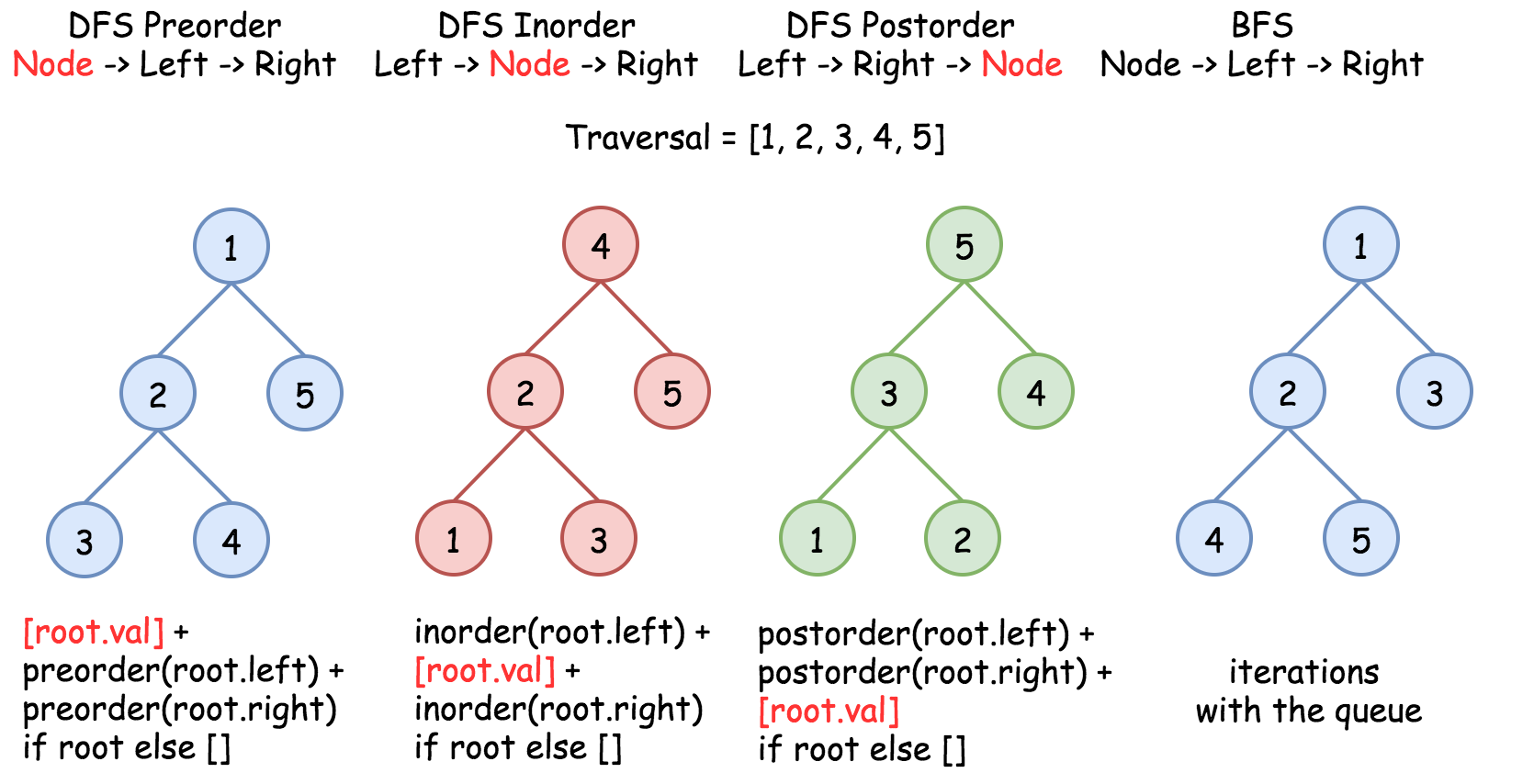
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node and right node.

* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are enumerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



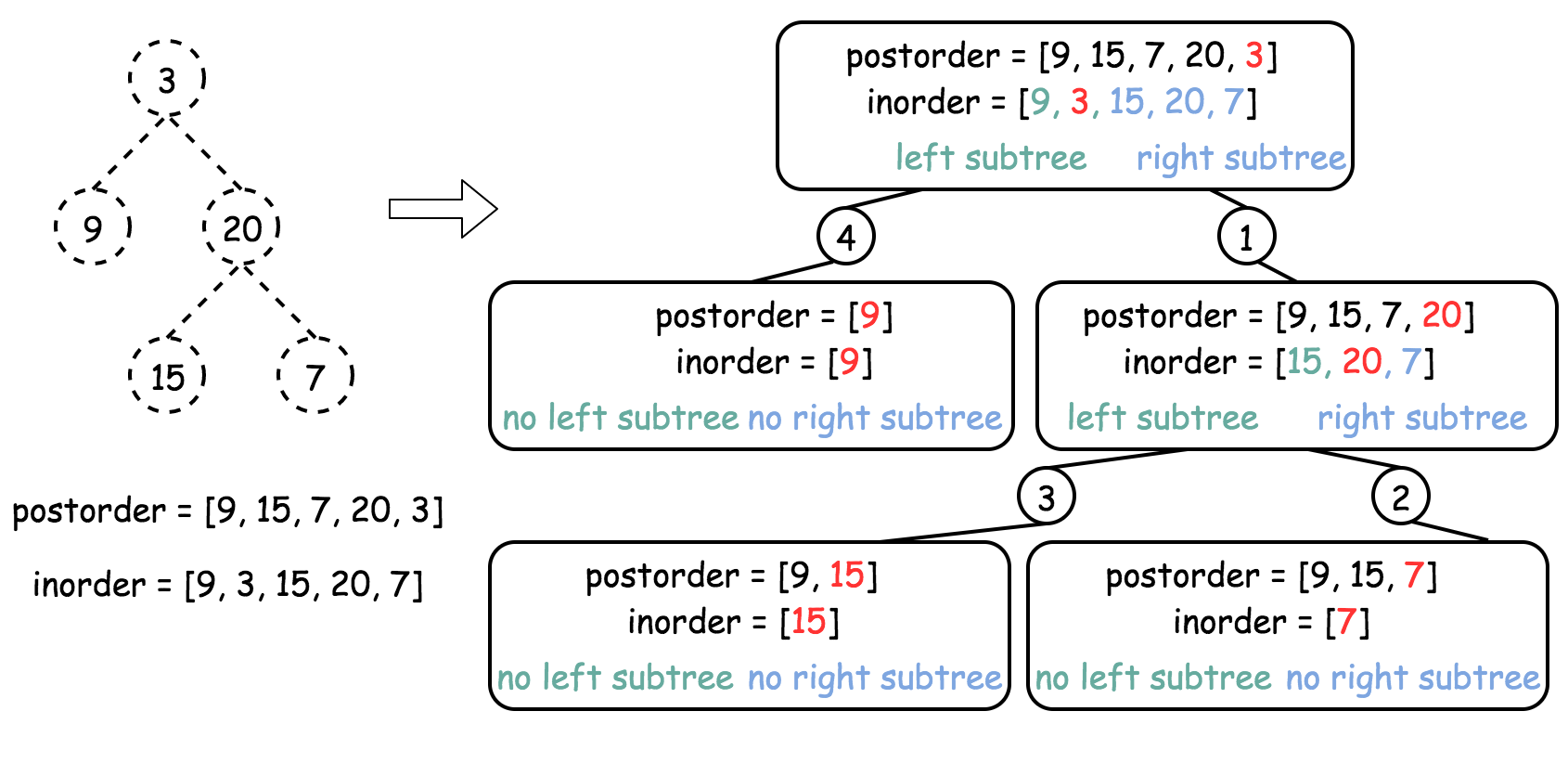
In this problem one deals with inorder and postorder traversals.

#### **Approach 1: Recursion**

**How to construct the tree from two traversals: inorder and preorder/postorder/etc**

Problems like this one are often at Facebook interviews, and could be solved in \mathcal{O}(N)O(*N*) time:

* Start from not inorder traversal, usually it's preorder or postorder one, and use the traversal picture above to define the strategy to pick the nodes. For example, for preorder traversal the first value is a root, then its left child, then its right child, etc. For postorder traversal the last value is a root, then its right child, then its left child, etc.
* The value picked from preorder/postorder traversal splits the inorder traversal into left and right subtrees. The only information one needs from inorder - if the current subtree is empty (= return None) or not (= continue to construct the subtree).



**Algorithm**

* Build hashmap value -> its index for inorder traversal.
* Return helper function which takes as the arguments the left and right boundaries for the current subtree in the inorder traversal. These boundaries are used only to check if the subtree is empty or not. Here is how it works helper(in\_left = 0, in\_right = n - 1):
  + If in\_left > in\_right, the subtree is empty, return None.
  + Pick the last element in postorder traversal as a root.
  + Root value has index index in the inorder traversal, elements from in\_left to index - 1 belong to the left subtree, and elements from index + 1 to in\_right belong to the right subtree.
  + Following the postorder logic, proceed recursively first to construct the right subtree helper(index + 1, in\_right) and then to construct the left subtree helper(in\_left, index - 1).
  + Return root.

**Implementation**

1 / 7

|  |
| --- |
| **class Solution {**  **int post\_idx;**  **int[] postorder;**  **int[] inorder;**  **HashMap<Integer, Integer> idx\_map = new HashMap<Integer, Integer>();**  **public TreeNode helper(int in\_left, int in\_right) {**  **// if there is no elements to construct subtrees**  **if (in\_left > in\_right)**  **return null;**  **// pick up post\_idx element as a root**  **int root\_val = postorder[post\_idx];**  **TreeNode root = new TreeNode(root\_val);**  **// root splits inorder list**  **// into left and right subtrees**  **int index = idx\_map.get(root\_val);**  **// recursion**  **post\_idx--;**  **// build right subtree**  **root.right = helper(index + 1, in\_right);**  **// build left subtree**  **root.left = helper(in\_left, index - 1);**  **return root;**  **}**  **public TreeNode buildTree(int[] inorder, int[] postorder) {**  **this.postorder = postorder;**  **this.inorder = inorder;**  **// start from the last postorder element**  **post\_idx = postorder.length - 1;**  **// build a hashmap value -> its index**  **int idx = 0;**  **for (Integer val : inorder)**  **idx\_map.put(val, idx++);**  **return helper(0, inorder.length - 1);**  **}**  **}** |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*). Let's compute the solution with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) T(N) = aT\left(\frac{b}{N}\right) + \Theta(N^d)*T*(*N*)=*aT*(*Nb*​)+Θ(*Nd*). The equation represents dividing the problem up into a*a* subproblems of size \frac{N}{b}*bN*​ in \Theta(N^d)Θ(*Nd*) time. Here one divides the problem in two subproblemes a = 2, the size of each subproblem (to compute left and right subtree) is a half of initial problem b = 2, and all this happens in a constant time d = 0. That means that \log\_b(a) > dlog*b*​(*a*)>*d* and hence we're dealing with [case 1](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_1_example) that means \mathcal{O}(N^{\log\_b(a)}) = \mathcal{O}(N)O(*N*log*b*​(*a*))=O(*N*) time complexity.
* Space complexity : \mathcal{O}(N)O(*N*), since we store the entire tree.