2458. Height of Binary Tree After Subtree Removal Queries

You are given the `root` of a **binary tree** with `n` nodes. Each node is assigned a unique value from `1` to `n`. You are also given an array `queries` of size `m`.

You have to perform `m` **independent** queries on the tree where in the `ith` query you do the following:

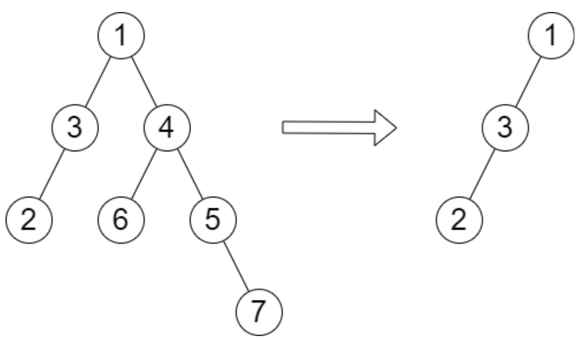
- **Remove** the subtree rooted at the node with the value `queries[i]` from the tree. It is **guaranteed** that `queries[i]` will **not** be equal to the value of the root.

Return *an array* `answer` *of size* `m` *where* `answer[i]` *is the height of the tree after performing the* `ith` *query*.

Note:

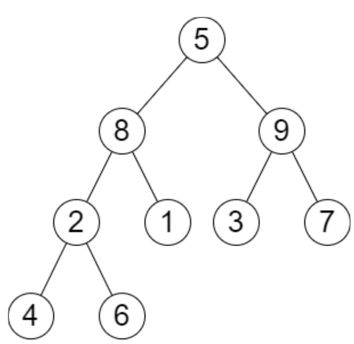
- The queries are independent, so the tree returns to its **initial** state after each query.
- The height of a tree is the **number of edges in the longest simple path** from the root to some node in the tree.

Example 1:




```
Input: root = [1,3,4,2,null,6,5,null,null,null,null,null,7], queries = [4]
Output: [2]
Explanation: The diagram above shows the tree after removing the subtree rooted at node with value 4.
The height of the tree is 2 (The path 1 -> 3 -> 2).
```

Example 2:




```
Input: root = [5,8,9,2,1,3,7,4,6], queries = [3,2,4,8]
Output: [3,2,3,2]
Explanation: We have the following queries:
- Removing the subtree rooted at node with value 3. The height of the tree becomes 3 (The path 5 -> 8 -> 2 -> 4).
- Removing the subtree rooted at node with value 2. The height of the tree becomes 2 (The path 5 -> 8 -> 1).
- Removing the subtree rooted at node with value 4. The height of the tree becomes 3 (The path 5 -> 8 -> 2 -> 6).
- Removing the subtree rooted at node with value 8. The height of the tree becomes 2 (The path 5 -> 9 -> 3).
```

Constraints:

- The number of nodes in the tree is `n`.
- `2 <= n <= 10^5`
- `1 <= Node.val <= n`
- All the values in the tree are **unique**.
- `m == queries.length`
- `1 <= m <= min(n, 10^4)`
- `1 <= queries[i] <= n`
- `queries[i] != root.val`

2458. Height of Binary Tree After Subtree Removal Queries

```
/*
   DFS+prefix max+suffix max
   Time complexity: O(n+q)
   Space complexity: O(5n)
   n: number of nodes in the tree
*/

typedef std::vector<int> vi;
class Solution {
   public:
      vi treeQueries(TreeNode* root, vi& queries){
         vi heights;
         std::unordered_map<int,int> node_index,sub_tree_size;
         int index=0;

         /*
            preorder traversal to map each node to its index(0 to n-1),
            and to compute the height of each node.

            postorder traversal to compute each node size

            Time complexity: O(n)
            Space complexity: O(n)
            n: number of nodes in the tree
         */
         auto dfs=[&](TreeNode* node,int height,auto& self)->int{
            if(!node) return 0;

            node_index[node->val]=index++;
            heights.push_back(height);

            int left=self(node->left,height+1,self);
            int right=self(node->right,height+1,self);

            int total=left+right+1;
            sub_tree_size[node->val]=total;

            return total;
         };

         dfs(root,0,dfs);

         int n=heights.size();
```

```
    /*
       prefix_max_height[i] stores the maximum height from the root up to the i-th node in the DFS traversal order.
       This array allows us to quickly find the maximum height among nodes visited before a specified node in the DFS traversal.

       suffix_max_height[i] stores the maximum height from the i-th node in the DFS traversal up to the last node.
       This array lets us find the maximum height among nodes visited after a specified node in the DFS traversal.

    */
    vi prefix_max_height(n),suffix_max_height(n);
    prefix_max_height[0]=heights[0];
    suffix_max_height[n-1]=heights[n-1];
    for(int i=1;i<n;++i){
       prefix_max_height[i]=std::max(prefix_max_height[i-1],heights[i]);
    }
    for(int i=n-2;i>=0;--i){
       suffix_max_height[i]=std::max(suffix_max_height[i+1],heights[i]);
    }

    /*
       By precomputing these arrays, for each query we can exclude the subtree rooted
        at the node in question and quickly look up the maximum heights to the left and right of this subtree.
    */
    vi ans;
    for(auto& node: queries){
       int i=node_index[node]-1;
       int j=i+sub_tree_size[node]+1;
       int max_height=prefix_max_height[i];
       if(j<n) max_height=std::max(max_height,suffix_max_height[j]);
       ans.push_back(max_height);
    }

    return ans;
  }
};
```