1028. Recover a Tree From Preorder Traversal

We run a preorder depth-first search (DFS) on the `root` of a binary tree.

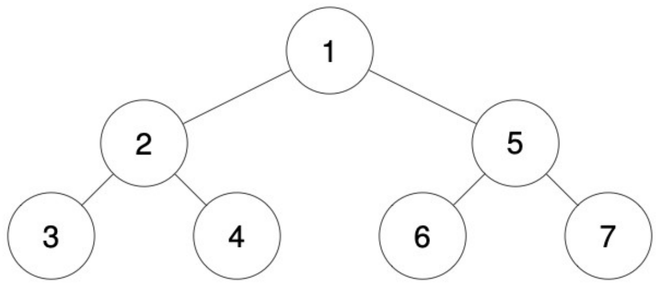
At each node in this traversal, we output `D` dashes (where `D` is the depth of this node), then we output the value of this node. If the depth of a node is `D`, the depth of its immediate child is `D + 1`. The depth of the `root` node is `0`.

If a node has only one child, that child is guaranteed to be **the left child**.

Given the output `traversal` of this traversal, recover the tree and return *its* `root`.

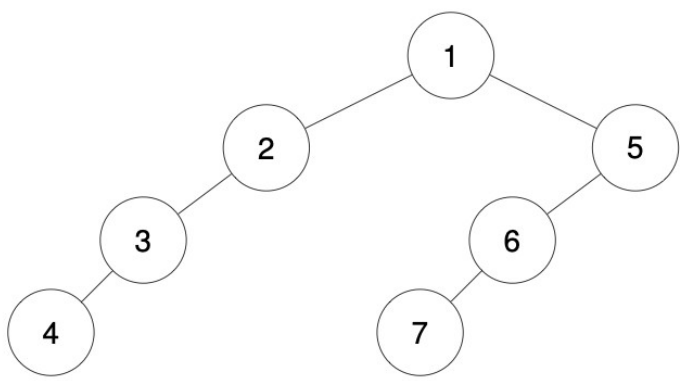
Example 1:

```
Input: traversal = "1-2--3--4-5--6--7" Output: [1,2,5,3,4,6,7]
```

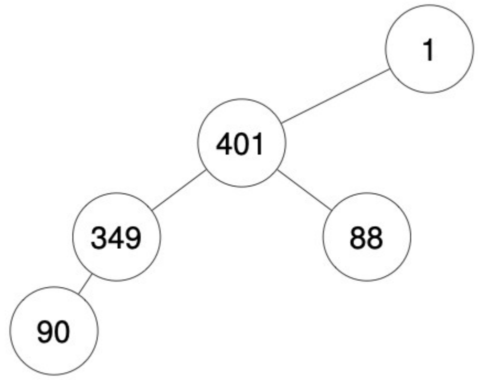


Example 2:

```
Input: traversal = "1-2--3---4-5--6---7"   Output: [1,2,5,3,null,6,null,4,null,7]
```



Input: `traversal = "1-401--349---90--88"` **Output:** `[1,401,null,349,88,90]`



Constraints:

- The number of nodes in the original tree is in the range `[1, 1000]`.
- `1 <= Node.val <= 109`

Overview

We are given a string representation of a preorder traversal of a binary tree, where each node is represented as D dashes followed by its value. The number of dashes D indicates the depth of the node in the tree, with the root having depth 0. Each node may have one or two children, and if a node has only one child, it is always the left child. Our task is to reconstruct the original binary tree from this traversal string.

Since preorder traversal follows the **root** → **left** → **right** order, we process the nodes in sequence and assign them to their correct positions.

For example, given `traversal = "1-2--3--4-5--6--7"`, we can break it down as follows:

1 (Root)
|- 2 (Depth 1, Left child of 1)
| |- 3 (Depth 2, Left child of 2)
| |- 4 (Depth 2, Right child of 2)
|- 5 (Depth 1, Right child of 1)
|- 6 (Depth 2, Left child of 5)
|- 7 (Depth 2, Right child of 5)

This means the tree structure is:

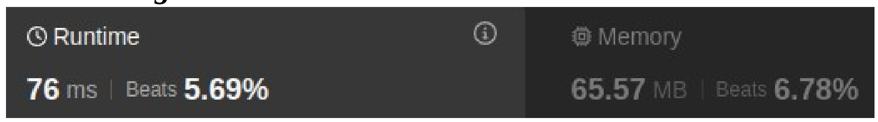
The output should be: `[1, 2, 5, 3, 4, 6, 7]`.

Regardless of the approach, the core idea is the same: When we encounter a new node, we determine its depth. We find the next node at `depth + 1` and attach current subtree root node to the new node. Then we ensure that the first child assigned to a parent is the left child, and the second (if present) is the right child.

```
/**
 * 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) {}
 * };
 */
```

1028. Recover a Tree From Preorder Traversal

```
/*
   Preorder: With processing the traversal string
   Time complexity:  O(2n)
   Space complexity:  O(3n)
   n: #nodes
*/
```

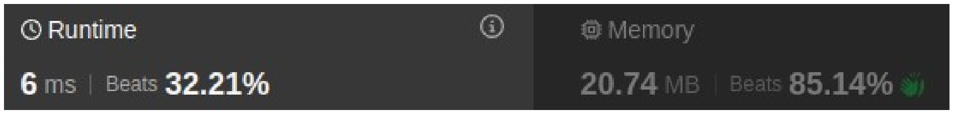



```
typedef std::vector<int> vi;
class Solution {
   public:
      // Process on the travsersal string
      // We get an array values contains node values, and depths, where depths[i]
      // contains the depths of values[i]
      // traversal="1-401--349---90--88"
      // values=  {   88, 90, 349,    401,    1}
      // depths   {    2,  3,   2,      1,    0}
      void process_input(std::string& traversal,vi& values,vi& depths){
         traversal="-"+traversal;
         int m=traversal.size();
         int s=m-1,e=m-1;
         while(s>0){
            while(s>0 && traversal[s]!='-') s--;

            values.push_back(std::stoi(traversal.substr(s+1,e)));

            int depth=0;
            while(s>0 && traversal[s]=='-'){
               s--;
               depth++;
            }

            depths.push_back(depth);

            e=s;
         }
      }
```

```
    TreeNode* recoverFromPreorder(std::string traversal){
    // process node values and depths each in separate arrays
    vi values,depths;
    process_input(traversal,values,depths);

    int n=values.size();
    int i=n-1; // Root is at the end of the values array

    // Recursive function to recover array from values and depths array in preoder traversal
    // Because nodes appear before their children in preorder,
    // we can sequentially assign them to their parents without needing to look ahead or
    // backtrack significantly.
    auto recover_tree=[&](TreeNode*& cur_node, int cur_depth,auto& self)->void{
        // i<0: If we are at the end of the values or
        // depths[i]!=cur_depth: The node at i does not belong to the current subtree
         // rooted by cur_node.
        // This ensures that we only create nodes that belong to the expected depth.
        if(i<0 || depths[i]!=cur_depth) return;

        // If the node belongs to the current subtree

        cur_node=new TreeNode(values[i]); // Create the current node

        i--; // Move to the next node in preorder sequence

        // Start by creating the left subtree if the next node in preorder sequence has a depth
        // greater than the current node by 1
        if(i>=0 && depths[i]==cur_depth+1) self(cur_node->left,cur_depth+1,self);

        // Otherwise, Create the right subtree
        // or keep moving UP until finding the correct depth, and create the right subtree
        self(cur_node->right,cur_depth+1,self);

    };

    TreeNode* root=nullptr;
    recover_tree(root,0,recover_tree);
    return root;
  }
};
```

1028. Recover a Tree From Preorder Traversal

```
/*
   Preorder: Without processing the traversal string
   Time compelxity: O(n)
   Space compelxity: O(n)
   n: #nodes
*/
```




```
typedef std::vector<int> vi;
class Solution {
   public:
      TreeNode* recoverFromPreorder(std::string traversal){
         int m=traversal.size();
         int i=0;

         // Recusive function to recover array from values and depths array in prepder traversal
         auto recover_tree=[&](TreeNode*& cur_node, int cur_depth,auto& self)->void{
            if(i>=m) return ;

            // Extract the node depth
            int depth=0;
            while(i+depth<m && traversal[i+depth]=='-') depth++;

            // depth od the node!=cur_depth: The node does not belong to the current
            //subtree rooted by cur_node.
            // This ensures that we only create nodes that belong to the expected depth.
            if(depth!=cur_depth) return;

            // Move index past the dashes
            i+=depth;

            // Extract the node value
            int value=0;
            while(i<m && traversal[i]!='-'){
               value=value*10+(traversal[i]-'0');
               i++;
            }

            cur_node=new TreeNode(value); // Create the current node

            // Start by creating the left subtree
            self(cur_node->left,cur_depth+1,self);

            // Otherwise, keep moving UP until finding the correct depth, and create the right subtree
            self(cur_node->right,cur_depth+1,self);
         };

         TreeNode* root=nullptr;
         recover_tree(root,0,recover_tree);
         return root;
      }
};
```