WEEK9

Contents

Populating Next Right Pointers in Each Node

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

`struct Node { int val; Node *left; Node *right; Node *next; }`

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Example 1:

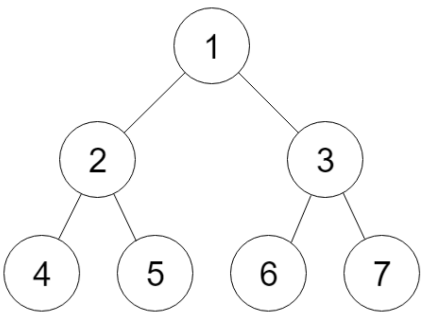


Figure A

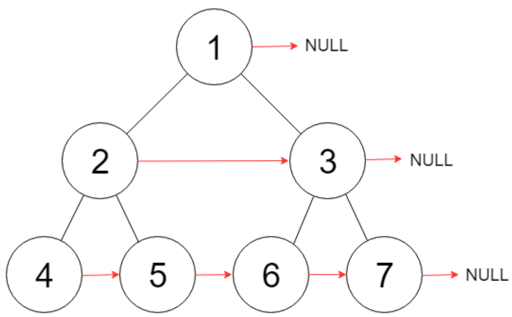


Figure B

`Input:` `root = [1,2,3,4,5,6,7]` **`Output:`** `[1,#,2,3,#,4,5,6,7,#]` **`Explanation:`** `Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.`

Example 2:

`Input:` `root = []` **`Output:`** `[]`

Constraints:

- The number of nodes in the tree is in the range $[0, 2^{12} - 1]$.
- `-1000 <= Node.val <= 1000`

Follow-up:

- You may only use constant extra space.
- The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

/* // Definition for a Node. class Node { public: int val; Node* left; Node* right; Node* next; Node() : val(0), left(NULL), right(NULL), next(NULL) {} Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {} Node(int _val, Node* _left, Node* _right, Node* _next) : val(_val), left(_left), right(_right), next(_next) {} }; */ class Solution { public: Node* connect(Node* root) { } };

Populating Next Right Pointers in Each Node II

Given a binary tree

`struct Node { int val; Node *left; Node *right; Node *next; }`

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Example 1:

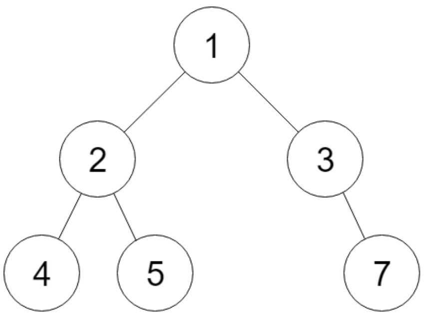


Figure A

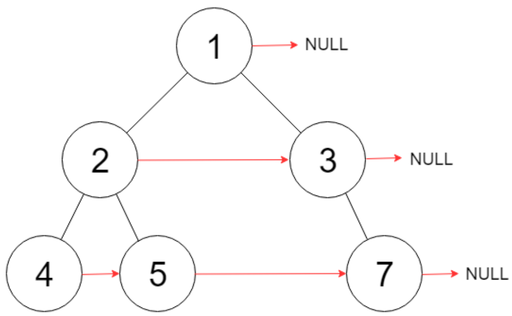


Figure B

Input: `root = [1,2,3,4,5,null,7]` **Output:** `[1,#,2,3,#,4,5,7,#]` **Explanation:** `Given the above binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.`

Example 2:

Input: `root = []` **Output:** `[]`

Constraints:

- The number of nodes in the tree is in the range `[0, 6000]`.
- `-100 <= Node.val <= 100`

Follow-up:

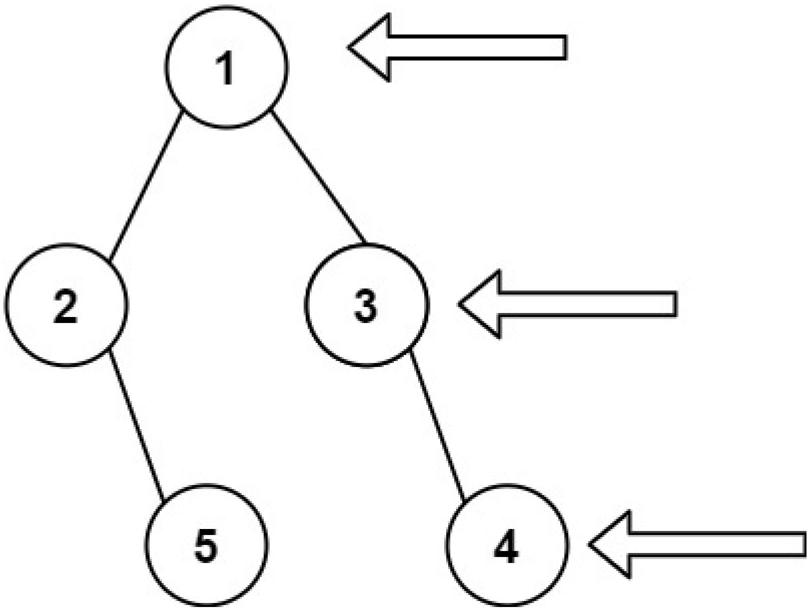
- You may only use constant extra space.
- The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

/* // Definition for a Node. class Node { public: int val; Node* left; Node* right; Node* next; Node() : val(0), left(NULL), right(NULL), next(NULL) {} Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {} Node(int _val, Node* _left, Node* _right, Node* _next) : val(_val), left(_left), right(_right), next(_next) {} }; */ class Solution { public: Node* connect(Node* root) { } };

Binary Tree Right Side View

Given the `root` of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

Example 1:



Input: `root = [1,2,3,null,5,null,4]` **Output:** `[1,3,4]`

Example 2:

Input: `root = [1,null,3]` **Output:** `[1,3]`

Example 3:

Input: `root = []` **Output:** `[]`

Constraints:

- The number of nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

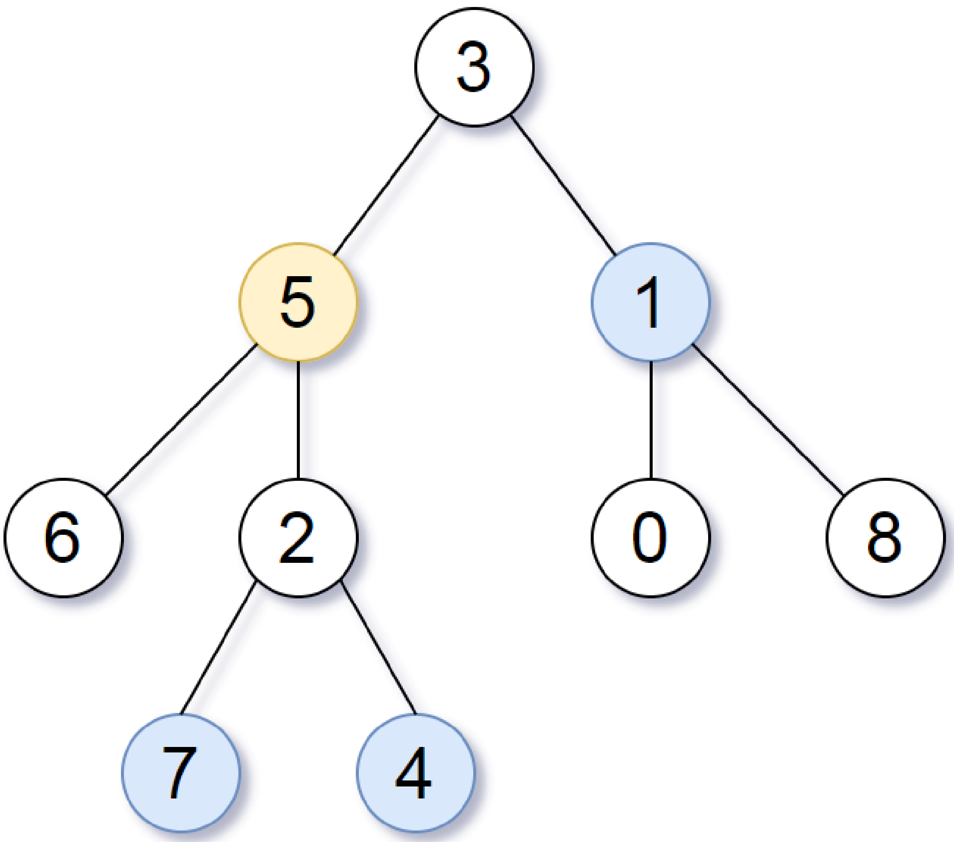
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {} * }; */ class Solution { public: vector rightSideView(TreeNode* root) { } };

All Nodes Distance K in Binary Tree

Given the `root` of a binary tree, the value of a target node `target`, and an integer `k`, return *an array of the values of all nodes that have a distance* `k` *from the target node.*

You can return the answer in **any order**.

Example 1:



Input: `root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2` **Output:** `[7,4,1]` Explanation: The nodes that are a distance 2 from the target node (with value 5) have values 7, 4, and 1.

Example 2:

Input: `root = [1], target = 1, k = 3` **Output:** `[]`

Constraints:

- The number of nodes in the tree is in the range `[1, 500]`.
- `0 <= Node.val <= 500`
- All the values `Node.val` are **unique**.
- `target` is the value of one of the nodes in the tree.
- `0 <= k <= 1000`

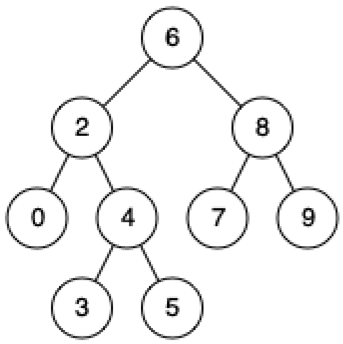
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode(int x) : val(x), left(NULL), right(NULL) {} * }; */ class Solution { public: vector distanceK(TreeNode* root, TreeNode* target, int k) { } };

Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

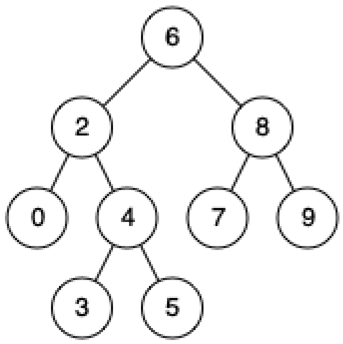
According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:



Input: `root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8` **Output:** `6` **Explanation:** `The LCA of nodes 2 and 8 is 6.`

Example 2:



Input: `root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4` **Output:** `2` **Explanation:** `The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.`

Example 3:

Input: `root = [2,1], p = 2, q = 1` **Output:** `2`

Constraints:

- The number of nodes in the tree is in the range $[2, 10^5]$.
- $-10^9 <= Node.val <= 10^9$
- All `Node.val` are **unique**.
- `p != q`
- p and q will exist in the BST.

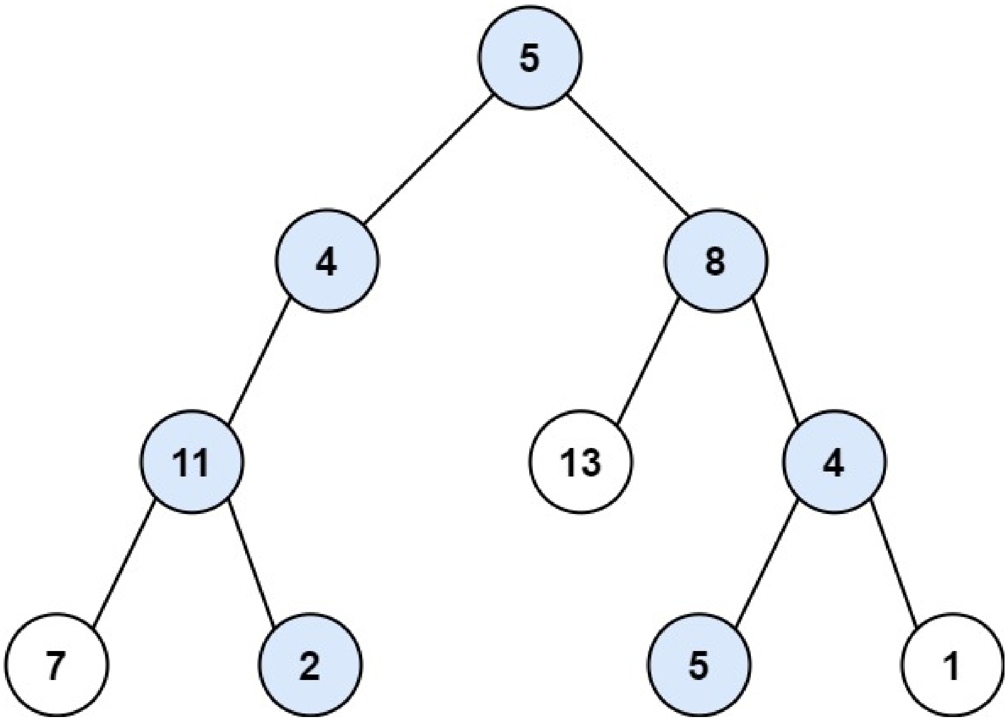
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode(int x) : val(x), left(NULL), right(NULL) {} * }; */ class Solution { public: TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) { } };

Path Sum II

Given the `root` of a binary tree and an integer `targetSum`, return *all **root-to-leaf** paths where the sum of the node values in the path equals* `targetSum`*. Each path should be returned as a list of the node **values**, not node references*.

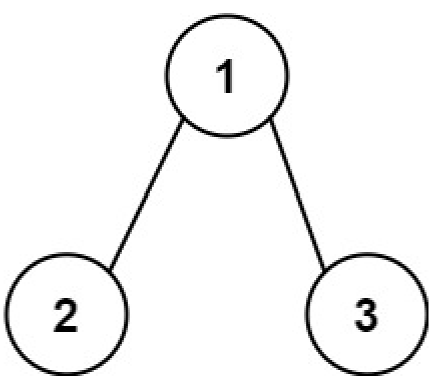
A **root-to-leaf** path is a path starting from the root and ending at any leaf node. A **leaf** is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22 **Output:** [[5,4,11,2],[5,8,4,5]]
Explanation: There are two paths whose sum equals targetSum: 5 + 4 + 11 + 2 = 22 5 + 8 + 4 + 5 = 22

Example 2:



Input: root = [1,2,3], targetSum = 5 **Output:** []

Example 3:

Input: root = [1,2], targetSum = 0 **Output:** []

Constraints:

- The number of nodes in the tree is in the range `[0, 5000]`.
- `-1000 <= Node.val <= 1000`
- `-1000 <= targetSum <= 1000`

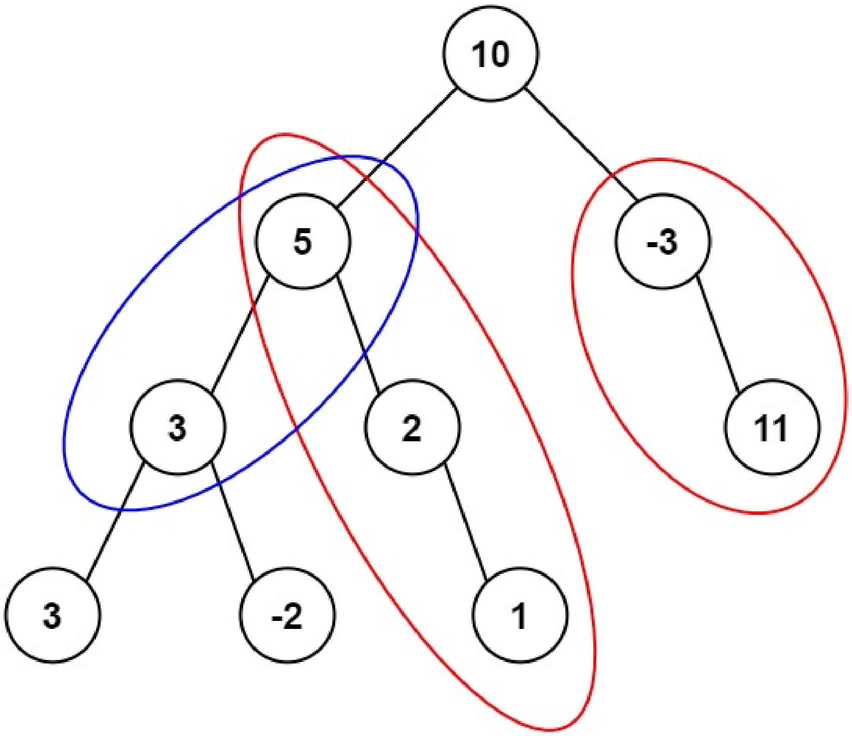
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {} * }; */ class Solution { public: vector> pathSum(TreeNode* root, int targetSum) { } };

Path Sum III

Given the `root` of a binary tree and an integer `targetSum`, return *the number of paths where the sum of the values along the path equals* `targetSum`.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

Example 1:



Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8 **Output:** 3 **Explanation:** The paths that sum to 8 are shown.

Example 2:

Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22 **Output:** 3

Constraints:

- The number of nodes in the tree is in the range `[0, 1000]`.
- -10^9 <= Node.val <= 10^9
- `-1000 <= targetSum <= 1000`

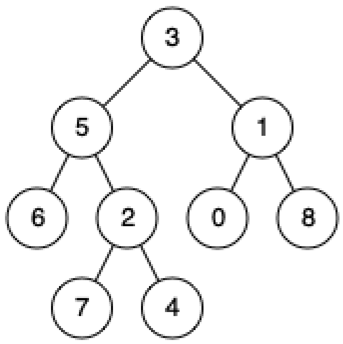
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {} * }; */ class Solution { public: int pathSum(TreeNode* root, int targetSum) { } };

Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

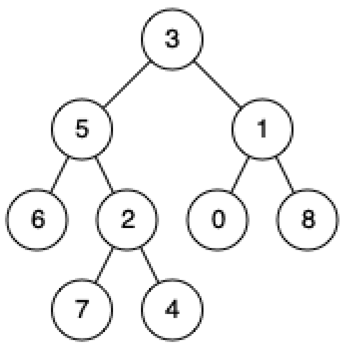
According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in `T` that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Example 1:



Input: `root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1` **Output:** `3` **Explanation:** `The LCA of nodes 5 and 1 is 3.`

Example 2:



Input: `root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4` **Output:** `5` **Explanation:** `The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.`

Example 3:

Input: `root = [1,2], p = 1, q = 2` **Output:** `1`

Constraints:

- The number of nodes in the tree is in the range $[2, 10^5]$.
- -10^9 <= `Node.val` <= 10^9
- All `Node.val` are **unique**.
- `p != q`
- p and q will exist in the tree.

/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode(int x) : val(x), left(NULL), right(NULL) {} * }; */ class Solution { public: TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) { } };

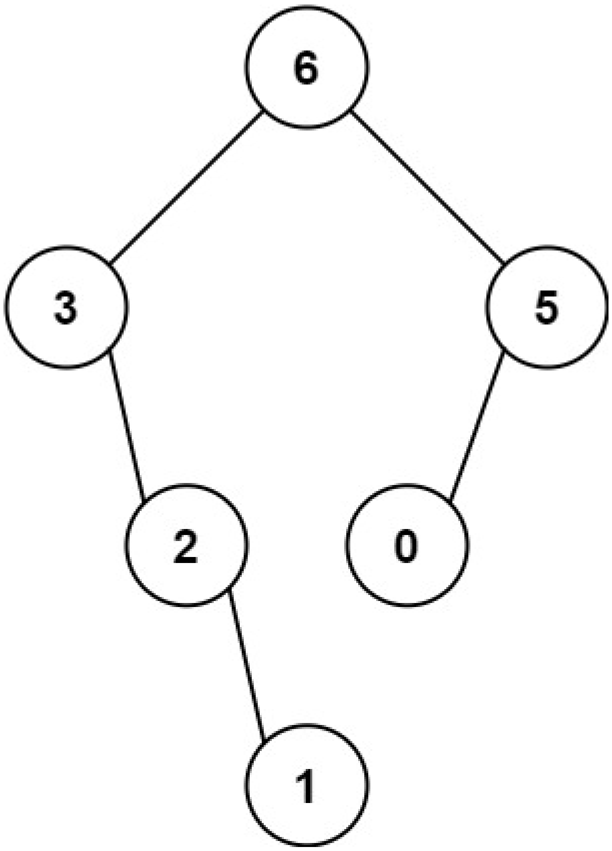
Maximum Binary Tree

You are given an integer array `nums` with no duplicates. A **maximum binary tree** can be built recursively from `nums` using the following algorithm:

1. Create a root node whose value is the maximum value in `nums`.
2. Recursively build the left subtree on the **subarray prefix** to the **left** of the maximum value.
3. Recursively build the right subtree on the **subarray suffix** to the **right** of the maximum value.

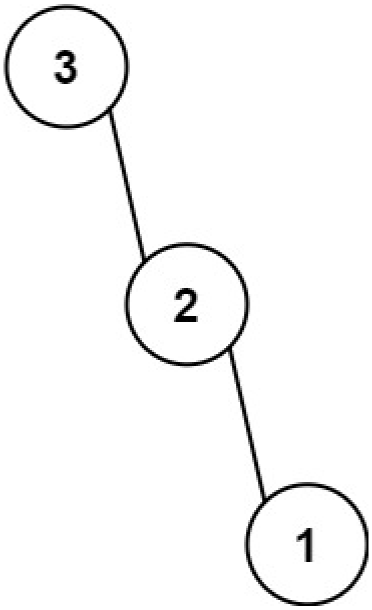
Return *the **maximum binary tree** built from* `nums`.

Example 1:




```
Input: nums = [3,2,1,6,0,5] Output: [6,3,5,null,2,0,null,null,1] Explanation: The recursive calls are as follow: - The largest value in [3,2,1,6,0,5] is 6. Left prefix is [3,2,1] and right suffix is [0,5]. - The largest value in [3,2,1] is 3. Left prefix is [] and right suffix is [2,1]. - Empty array, so no child. - The largest value in [2,1] is 2. Left prefix is [] and right suffix is [1]. - Empty array, so no child. - Only one element, so child is a node with value 1. - The largest value in [0,5] is 5. Left prefix is [0] and right suffix is []. - Only one element, so child is a node with value 0. - Empty array, so no child.
```

Example 2:



`Input:` `nums = [3,2,1]` **`Output:`** `[3,null,2,null,1]`

Constraints:

- `1 <= nums.length <= 1000`
- `0 <= nums[i] <= 1000`
- All integers in `nums` are **unique**.

/** * 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* constructMaximumBinaryTree(vector& nums) { } };

Maximum Width of Binary Tree

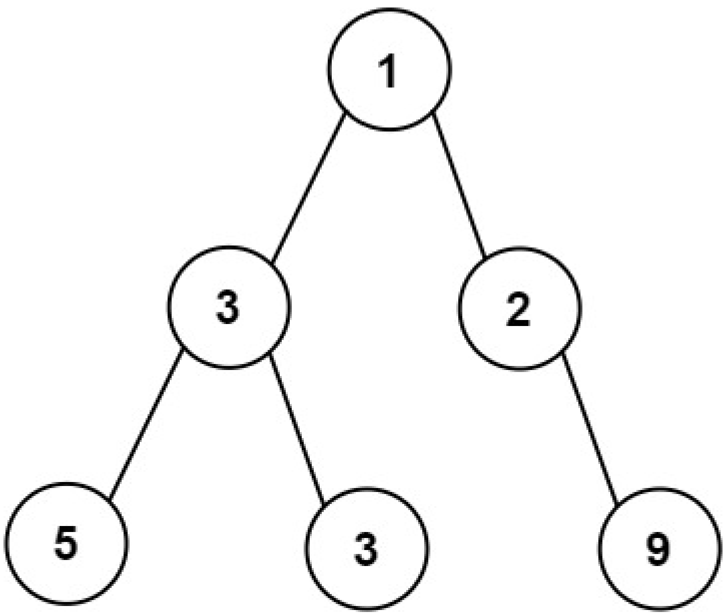
Given the `root` of a binary tree, return *the **maximum width** of the given tree*.

The **maximum width** of a tree is the maximum **width** among all levels.

The **width** of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

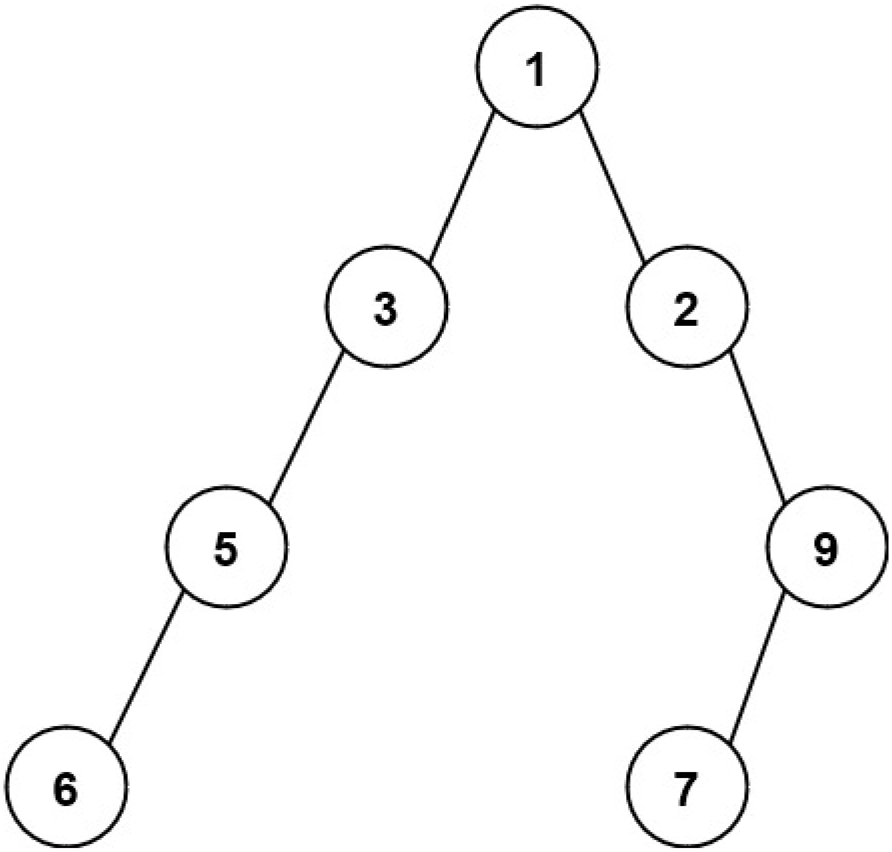
It is **guaranteed** that the answer will in the range of a **32-bit** signed integer.

Example 1:



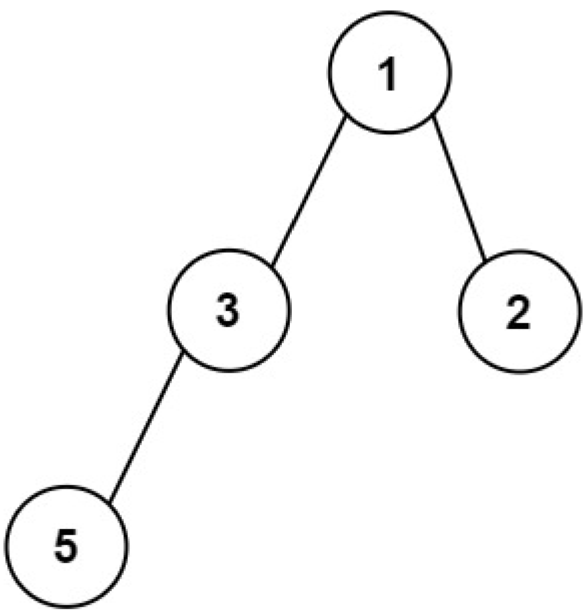
Input: `root = [1,3,2,5,3,null,9]` **Output:** `4` **Explanation:** `The maximum width exists in the third level with length 4 (5,3,null,9).`

Example 2:



Input: root = [1,3,2,5,null,null,9,6,null,7] **Output:** 7 **Explanation:** The maximum width exists in the fourth level with length 7 (6,null,null,null,null,null,7).

Example 3:



Input: root = [1,3,2,5] **Output:** 2 **Explanation:** The maximum width exists in the second level with length 2 (3,2).

Constraints:

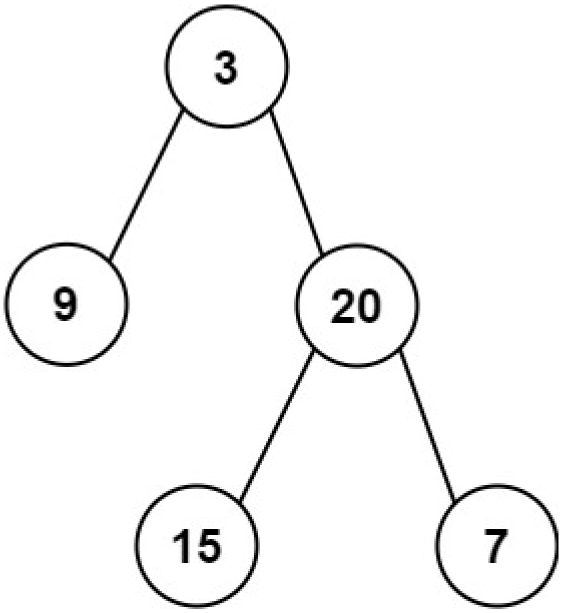
- The number of nodes in the tree is in the range `[1, 3000]`.
- `-100 <= Node.val <= 100`

/** * 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: int widthOfBinaryTree(TreeNode* root) { } };

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.

/** * 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& preorder, vector& inorder) { } };

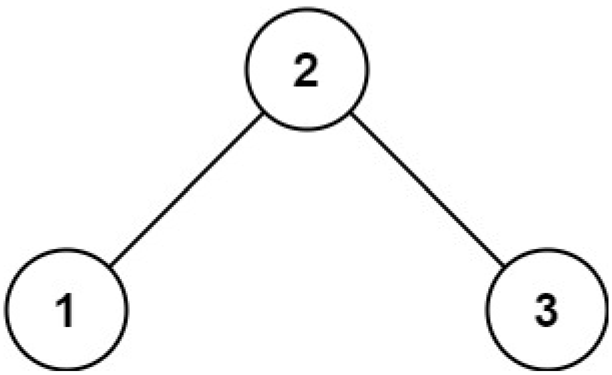
Validate Binary Search Tree

Given the `root` of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

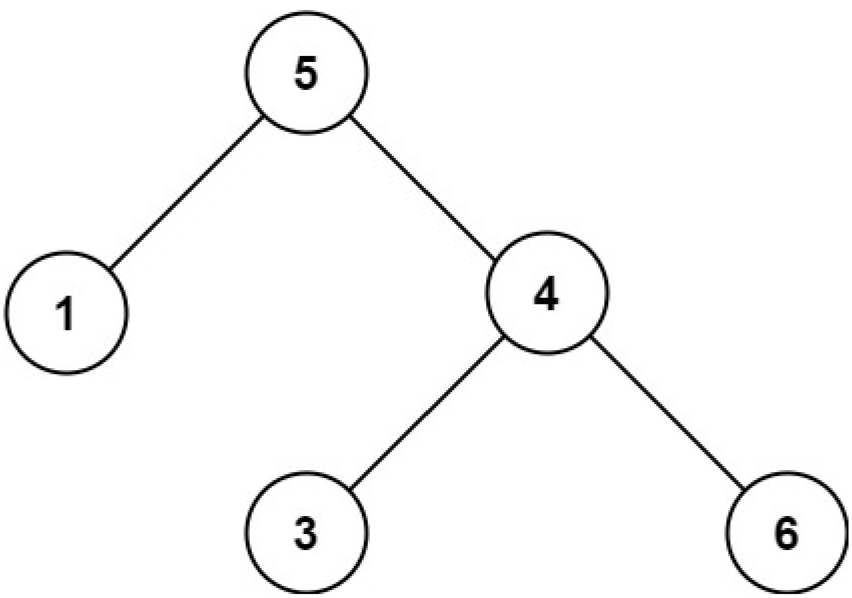
- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



Input: `root = [2,1,3]` **Output:** `true`

Example 2:



Input: `root = [5,1,4,null,null,3,6]` **Output:** `false` **Explanation:** `The root node's value is 5 but its right child's value is 4.`

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- $-2^{31} <= Node.val <= 2^{31} - 1$

/** * 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: bool isValidBST(TreeNode* root) { } };

Implement Trie (Prefix Tree)

A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

- `Trie()` Initializes the trie object.
- `void insert(String word)` Inserts the string `word` into the trie.
- `boolean search(String word)` Returns `true` if the string `word` is in the trie (i.e., was inserted before), and `false` otherwise.
- `boolean startsWith(String prefix)` Returns `true` if there is a previously inserted string `word` that has the prefix `prefix`, and `false` otherwise.

Example 1:

```
Input ["Trie", "insert", "search", "search", "startsWith", "insert", "search"] [[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]] Output [null, null, true, false, true, null, true] Explanation Trie trie = new Trie(); trie.insert("apple"); trie.search("apple"); // return True trie.search("app"); // return False trie.startsWith("app"); // return True trie.insert("app"); trie.search("app"); // return True
```

Constraints:

- `1 <= word.length, prefix.length <= 2000`
- `word` and `prefix` consist only of lowercase English letters.
- At most $3 * 10^4$ calls **in total** will be made to `insert`, `search`, and `startsWith`.

class Trie { public: Trie() { } void insert(string word) { } bool search(string word) { } bool startsWith(string prefix) { } }; /** * Your Trie object will be instantiated and called as such: * Trie* obj = new Trie(); * obj->insert(word); * bool param_2 = obj->search(word); * bool param_3 = obj->startsWith(prefix); */

3Sum

Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

Input: nums = [-1,0,1,2,-1,-4] **Output:** [[-1,-1,2],[-1,0,1]] **Explanation:** nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0. nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0. nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0. The distinct triplets are [-1,0,1] and [-1,-1,2]. Notice that the order of the output and the order of the triplets does not matter.

Example 2:

Input: nums = [0,1,1] **Output:** [] **Explanation:** The only possible triplet does not sum up to 0.

Example 3:

Input: nums = [0,0,0] **Output:** [[0,0,0]] **Explanation:** The only possible triplet sums up to 0.

Constraints:

- `3 <= nums.length <= 3000`
- -10^5 `<= nums[i] <=` 10^5

class Solution { public: vector> threeSum(vector& nums) { } };

3Sum Closest

Given an integer array `nums` of length n and an integer `target`, find three integers in `nums` such that the sum is closest to `target`.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

Example 1:

Input: nums = [-1,2,1,-4], target = 1 **Output:** 2 **Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Example 2:

Input: nums = [0,0,0], target = 1 **Output:** 0 **Explanation:** The sum that is closest to the target is 0. (0 + 0 + 0 = 0).

Constraints:

- `3 <= nums.length <= 500`
- `-1000 <= nums[i] <= 1000`
- -10^4 `<= target <=` 10^4

class Solution { public: int threeSumClosest(vector& nums, int target) { } };