

WEEK8

## **Contents**

- 1. Longest Repeating Character Replacement - Medium**
- 2. Longest Substring Without Repeating Characters - Medium**
- 3. Kth Smallest Element in a BST - Medium**
- 4. K Closest Points to Origin - Medium**
- 5. Top K Frequent Elements - Medium**
- 6. Sort Characters By Frequency - Medium**
- 7. Kth Largest Element in an Array - Medium**
- 8. Reorganize String - Medium**
- 9. Course Schedule - Medium**
- 10. Course Schedule II - Medium**
- 11. Minimum Height Trees - Medium**
- 12. Binary Tree Level Order Traversal II - Medium**
- 13. Binary Tree Level Order Traversal - Medium**
- 14. Binary Tree Zigzag Level Order Traversal - Medium**

## Longest Repeating Character Replacement

You are given a string  $s$  and an integer  $k$ . You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most  $k$  times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

### Example 1:

**Input:**  $s = \text{"ABAB"}, k = 2$  **Output:** 4 **Explanation:** Replace the two 'A's with two 'B's or vice versa.

### Example 2:

**Input:**  $s = \text{"AABABBA"}, k = 1$  **Output:** 4 **Explanation:** Replace the one 'A' in the middle with 'B' and form "AABBBBA". The substring "BBBB" has the longest repeating letters, which is 4. There may exists other ways to achieve this answer too.

### Constraints:

- $1 \leq s.length \leq 10^5$
- $s$  consists of only uppercase English letters.
- $0 \leq k \leq s.length$

```
class Solution { public: int characterReplacement(string s, int k) { } };
```

## Longest Substring Without Repeating Characters

Given a string `s`, find the length of the **longest substring** without repeating characters.

### Example 1:

**Input:** `s = "abcabcbb"` **Output:** 3 **Explanation:** The answer is "abc", with the length of 3.

### Example 2:

**Input:** `s = "bbbbbb"` **Output:** 1 **Explanation:** The answer is "b", with the length of 1.

### Example 3:

**Input:** `s = "pwwkew"` **Output:** 3 **Explanation:** The answer is "wke", with the length of 3. Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

### Constraints:

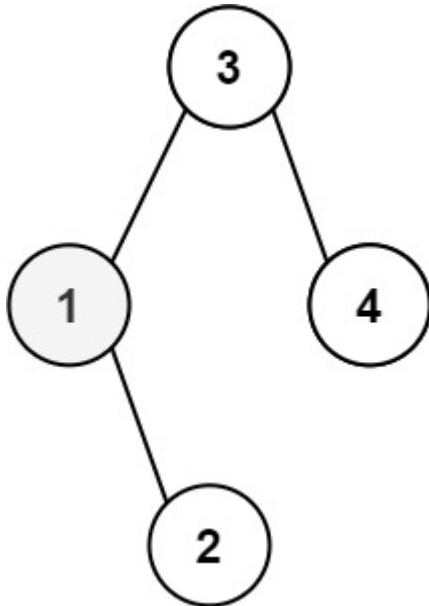
- $0 \leq s.length \leq 5 \times 10^4$
- `s` consists of English letters, digits, symbols and spaces.

```
class Solution { public: int lengthOfLongestSubstring(string s) {} };
```

## Kth Smallest Element in a BST

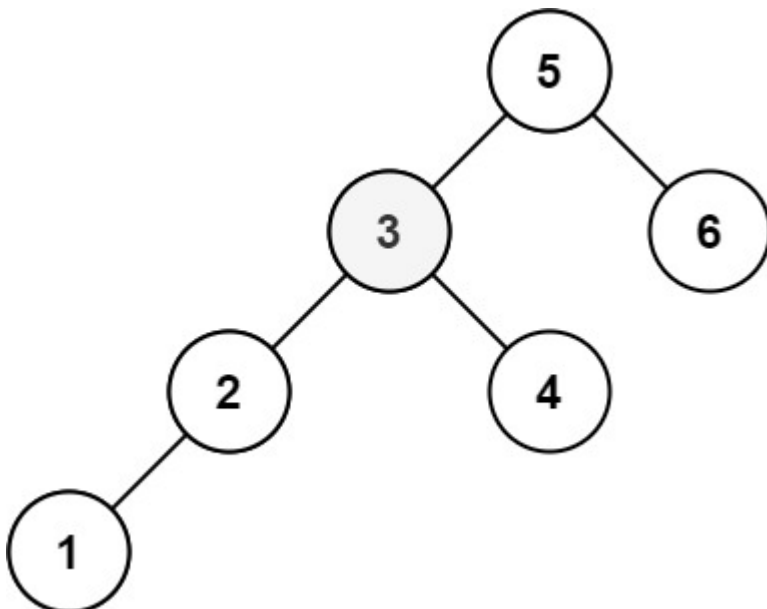
Given the `root` of a binary search tree, and an integer `k`, return *the  $k^{\text{th}}$  smallest value (1-indexed) of all the values of the nodes in the tree.*

Example 1:



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

Example 2:



**Input:** `root = [5,3,6,2,4,null,null,1]`, `k = 3` **Output:** 3

**Constraints:**

- The number of nodes in the tree is `n`.

- $1 \leq k \leq n \leq 10^4$
- $0 \leq \text{Node.val} \leq 10^4$

**Follow up:** If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

```
/** * 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 kthSmallest(TreeNode* root, int k) { } };
```

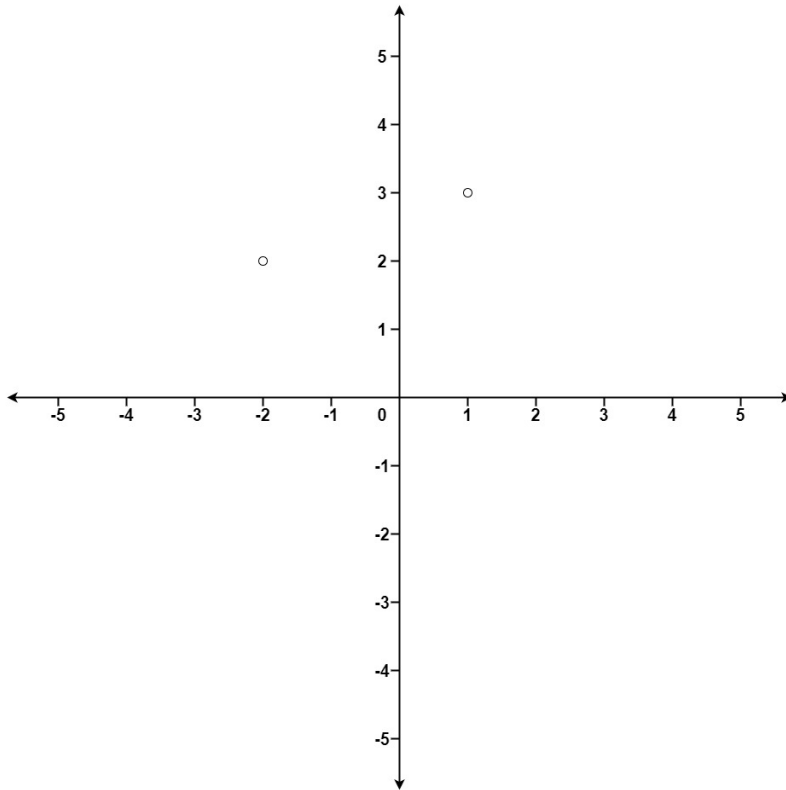
## K Closest Points to Origin

Given an array of `points` where `points[i] = [xi, yi]` represents a point on the **X-Y** plane and an integer `k`, return the `k` closest points to the origin `(0, 0)`.

The distance between two points on the **X-Y** plane is the Euclidean distance (i.e.,  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ ).

You may return the answer in **any order**. The answer is **guaranteed** to be **unique** (except for the order that it is in).

### Example 1:



**Input:** `points = [[1,3],[-2,2]]`, `k = 1` **Output:** `[[-2,2]]` **Explanation:** The distance between `(1, 3)` and the origin is `sqrt(10)`. The distance between `(-2, 2)` and the origin is `sqrt(8)`. Since `sqrt(8) < sqrt(10)`, `(-2, 2)` is closer to the origin. We only want the closest `k = 1` points from the origin, so the answer is just `[[-2,2]]`.

### Example 2:

**Input:** `points = [[3,3],[5,-1],[-2,4]]`, `k = 2` **Output:** `[[3,3],[-2,4]]` **Explanation:** The answer `[[-2,4],[3,3]]` would also be accepted.

### Constraints:

- `1 <= k <= points.length <= 104`
- `-104 <= xi, yi <= 104`

```
class Solution { public: vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {}};
```

## Top K Frequent Elements

Given an integer array `nums` and an integer `k`, return *the k most frequent elements*. You may return the answer in **any order**.

### Example 1:

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

### Example 2:

**Input:** `nums = [1]`, `k = 1` **Output:** `[1]`

### Constraints:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`
- `k` is in the range `[1, the number of unique elements in the array]`.
- It is **guaranteed** that the answer is **unique**.

**Follow up:** Your algorithm's time complexity must be better than  $O(n \log n)$ , where `n` is the array's size.

```
class Solution { public: vector topKFrequent(vector& nums, int k) { } };
```



## Sort Characters By Frequency

Given a string `s`, sort it in **decreasing order** based on the **frequency** of the characters. The **frequency** of a character is the number of times it appears in the string.

Return *the sorted string*. If there are multiple answers, return *any of them*.

### Example 1:

**Input:** `s = "tree"` **Output:** `"eert"` **Explanation:** 'e' appears twice while 'r' and 't' both appear once. So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

### Example 2:

**Input:** `s = "cccaaa"` **Output:** `"aaaccc"` **Explanation:** Both 'c' and 'a' appear three times, so both "cccaaa" and "aaaccc" are valid answers. Note that "cacaca" is incorrect, as the same characters must be together.

### Example 3:

**Input:** `s = "Aabb"` **Output:** `"bbAa"` **Explanation:** "bbaA" is also a valid answer, but "Aabb" is incorrect. Note that 'A' and 'a' are treated as two different characters.

### Constraints:

- $1 \leq s.length \leq 5 * 10^5$
- `s` consists of uppercase and lowercase English letters and digits.

```
class Solution { public: string frequencySort(string s) {} };
```

## Kth Largest Element in an Array

Given an integer array `nums` and an integer `k`, return *the  $k^{\text{th}}$  largest element in the array*.

Note that it is the  $k^{\text{th}}$  largest element in the sorted order, not the  $k^{\text{th}}$  distinct element.

Can you solve it without sorting?

### Example 1:

**Input:** `nums = [3,2,1,5,6,4]`, `k = 2` **Output:** 5

### Example 2:

**Input:** `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4` **Output:** 4

### Constraints:

- $1 \leq k \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

```
class Solution { public: int findKthLargest(vector& nums, int k) { } };
```

## Reorganize String

Given a string  $s$ , rearrange the characters of  $s$  so that any two adjacent characters are not the same.

Return *any possible rearrangement of  $s$*  or return "" if not possible.

### Example 1:

**Input:**  $s = \text{"aab"}$  **Output:**  $\text{"aba"}$

### Example 2:

**Input:**  $s = \text{"aaab"}$  **Output:**  $\text{" "}$

### Constraints:

- $1 \leq s.length \leq 500$
- $s$  consists of lowercase English letters.

```
class Solution { public: string reorganizeString(string s) { } };
```

## Course Schedule

There are a total of `numCourses` courses you have to take, labeled from 0 to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you **must** take course `bi` first if you want to take course `ai`.

- For example, the pair `[0, 1]`, indicates that to take course 0 you have to first take course 1.

Return `true` if you can finish all courses. Otherwise, return `false`.

### Example 1:

**Input:** `numCourses = 2, prerequisites = [[1,0]]` **Output:** `true` **Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

### Example 2:

**Input:** `numCourses = 2, prerequisites = [[1,0],[0,1]]` **Output:** `false` **Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

### Constraints:

- `1 <= numCourses <= 2000`
- `0 <= prerequisites.length <= 5000`
- `prerequisites[i].length == 2`
- `0 <= ai, bi < numCourses`
- All the pairs `prerequisites[i]` are **unique**.

```
class Solution { public: bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {} };
```

## Course Schedule II

There are a total of `numCourses` courses you have to take, labeled from 0 to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you **must** take course `bi` first if you want to take course `ai`.

- For example, the pair `[0, 1]`, indicates that to take course 0 you have to first take course 1.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

### Example 1:

**Input:** `numCourses = 2, prerequisites = [[1,0]]` **Output:** `[0,1]` **Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is `[0,1]`.

### Example 2:

**Input:** `numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]` **Output:** `[0,2,1,3]` **Explanation:** There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is `[0,1,2,3]`. Another correct ordering is `[0,2,1,3]`.

### Example 3:

**Input:** `numCourses = 1, prerequisites = []` **Output:** `[0]`

### Constraints:

- `1 <= numCourses <= 2000`
  - `0 <= prerequisites.length <= numCourses * (numCourses - 1)`
  - `prerequisites[i].length == 2`
  - `0 <= ai, bi < numCourses`
  - `ai != bi`
  - All the pairs `[ai, bi]` are **distinct**.
- ```
class Solution { public: vector findOrder(int numCourses, vector<vector<int>>& prerequisites) {} };
```

## Minimum Height Trees

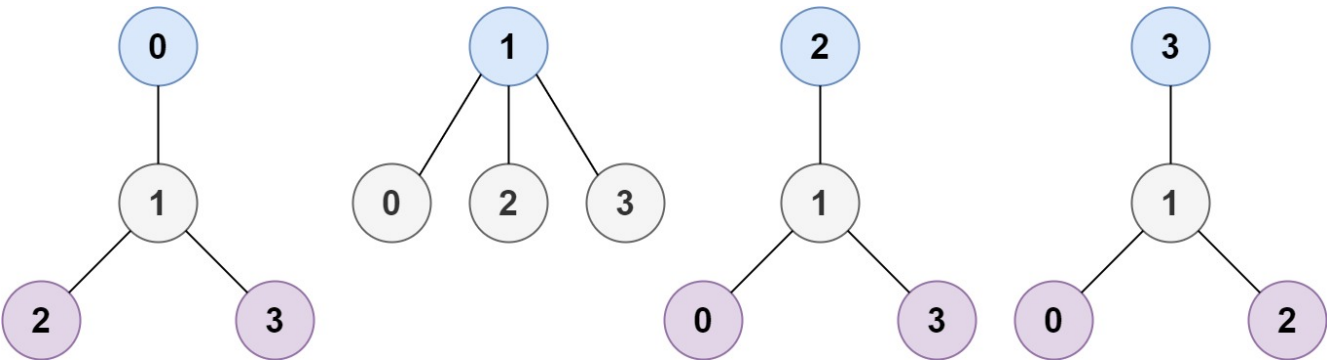
A tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of  $n$  nodes labelled from  $0$  to  $n - 1$ , and an array of  $n - 1$  edges where  $\text{edges}[i] = [a_i, b_i]$  indicates that there is an undirected edge between the two nodes  $a_i$  and  $b_i$  in the tree, you can choose any node of the tree as the root. When you select a node  $x$  as the root, the result tree has height  $h$ . Among all possible rooted trees, those with minimum height (i.e.  $\min(h)$ ) are called **minimum height trees** (MHTs).

Return a list of all **MHTs'** root labels. You can return the answer in **any order**.

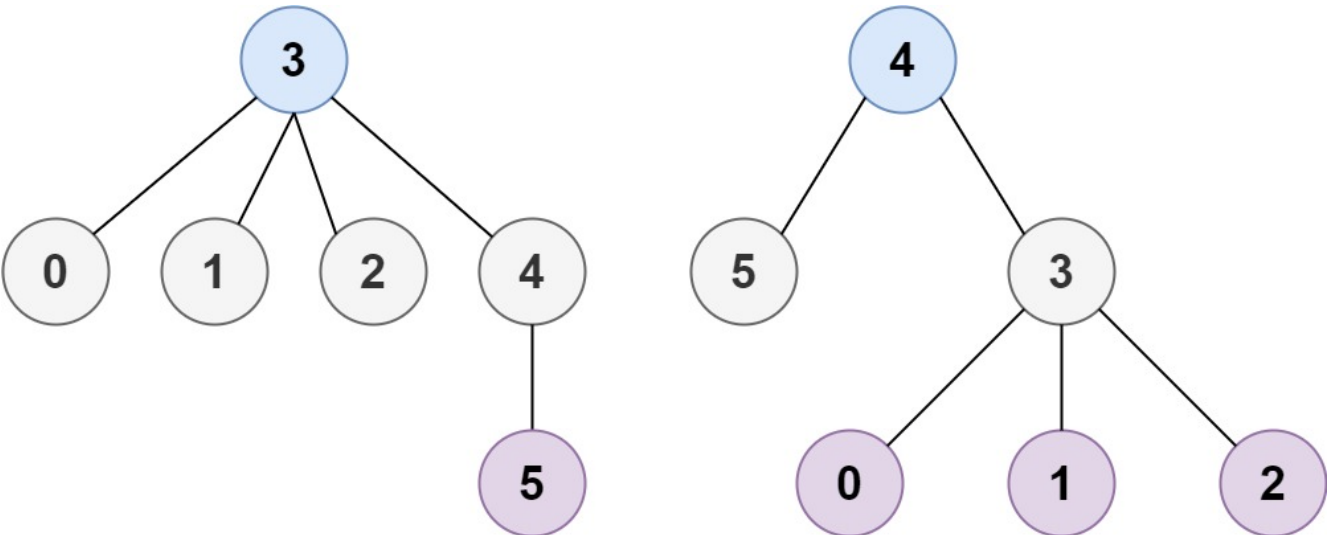
The **height** of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

**Example 1:**



**Input:**  $n = 4$ ,  $\text{edges} = [[1,0],[1,2],[1,3]]$  **Output:**  $[1]$  **Explanation:** As shown, the height of the tree is 1 when the root is the node with label 1 which is the only MHT.

**Example 2:**



**Input:**  $n = 6$ ,  $\text{edges} = [[3,0],[3,1],[3,2],[3,4],[5,4]]$  **Output:**  $[3,4]$

**Constraints:**

- $1 \leq n \leq 2 \cdot 10^4$

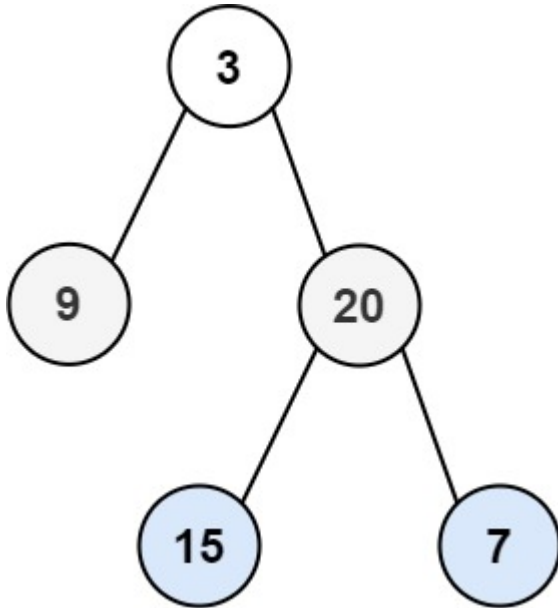
- `edges.length == n - 1`
- $0 \leq a_i, b_i < n$
- $a_i \neq b_i$
- All the pairs  $(a_i, b_i)$  are distinct.
- The given input is **guaranteed** to be a tree and there will be **no repeated** edges.

```
class Solution { public: vector findMinHeightTrees(int n, vector<vector<int>>& edges) { } };
```

## Binary Tree Level Order Traversal II

Given the `root` of a binary tree, return *the bottom-up level order traversal of its nodes' values*. (i.e., from left to right, level by level from leaf to root).

**Example 1:**



**Input:** `root = [3,9,20,null,null,15,7]` **Output:** `[[15,7],[9,20],[3]]`

**Example 2:**

**Input:** `root = [1]` **Output:** `[[1]]`

**Example 3:**

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

### Constraints:

- The number of nodes in the tree is in the range `[0, 2000]`.
- `-1000 <= Node.val <= 1000`

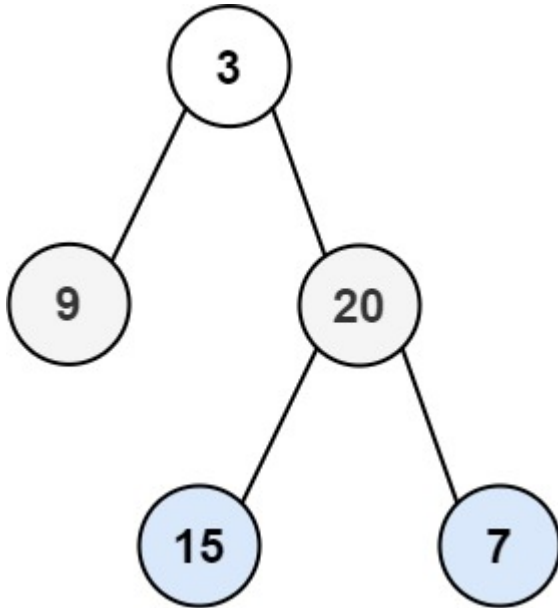
```
/** * Definition for a binary tree node. * struct TreeNode { * int val; * TreeNode *left; * TreeNode *right; * TreeNode() : val(0), left(nullptr), right(nullptr) {} * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),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



## Binary Tree Level Order Traversal

Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

**Example 1:**



**Input:** `root = [3,9,20,null,null,15,7]` **Output:** `[[3],[9,20],[15,7]]`

**Example 2:**

**Input:** `root = [1]` **Output:** `[[1]]`

**Example 3:**

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

### Constraints:

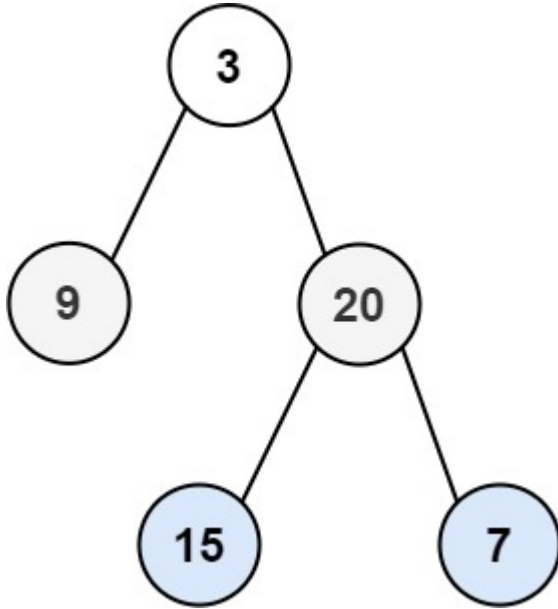
- The number of nodes in the tree is in the range `[0, 2000]`.
- `-1000 <= Node.val <= 1000`

```
/** * 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<vector<int>> levelOrder(TreeNode* root) { }}
```

## Binary Tree Zigzag Level Order Traversal

Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**



**Input:** `root = [3,9,20,null,null,15,7]` **Output:** `[[3],[20,9],[15,7]]`

**Example 2:**

**Input:** `root = [1]` **Output:** `[[1]]`

**Example 3:**

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

### Constraints:

- The number of nodes in the tree is in the range `[0, 2000]`.
- `-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<vector<int>> zigzagLevelOrder(TreeNode* root) { }
```