199. Binary Tree Right Side View Medium Tree **Binary Tree Depth-First Search Breadth-First Search** 

# **Problem Description**

means that we want the last node's value in each level of the tree, proceeding from top to bottom. A binary tree is a data structure where each node has at most two children referred to as the left child and the right child.

This problem asks us to determine the values of the nodes visible when we look at a binary tree from the right side. This essentially

Intuition

level (which will be the last one we encounter in the queue for that level) and record that value.

To solve this problem, we can use a level-order traversal strategy. The level-order traversal involves traveling through the tree one

level at a time, starting at the root. This is typically done using a queue data structure where we enqueue all nodes at the current level before we proceed to the next level.

While performing a level-order traversal, we can keep track of the nodes at the current level by counting how many nodes are in the

queue before we start processing the level. Then, as we enqueue their children, we can observe the rightmost node's value of each

visible from that perspective. **Solution Approach** 

This approach allows us to see the <u>tree</u> as if we were standing on its right side and collect the values of the nodes that would be

The solution uses a level-order traversal approach, utilizing a queue to track the nodes at each level. Here's a step-by-step walkthrough of the implementation:

2. We check if the root is None (i.e., the tree is empty) and if so, return the empty list ans. There's nothing to traverse if the tree is

## empty.

while loop exits.

side of the <u>tree</u>.

**Example Walkthrough** 

3. A queue q is initialized with the root node as its only element. This queue will help in the level-order traversal, keeping track of nodes that need to be processed.

4. We begin a while loop which continues as long as there are nodes in the queue q. Each iteration of this loop represents

1. We initialize an empty list ans which will store the values of the rightmost nodes at each level of the binary tree.

traversing one level of the binary tree.

nodes in the current level by the current length of the queue len(q).

does, we append the left child to the queue.

- 5. At the beginning of each level traversal, we append the value of the last node in the queue (the rightmost node of the current level) to the ans list. We use q[-1].val to fetch this value as we are using a double-ended queue deque from Python's collections
- module. 6. We then enter another loop to process all nodes at the current level, which are already in the queue. We find the number of

7. Inside this inner loop, we pop the leftmost node from the queue using q.popleft() and check if this node has a left child. If it

8. We also check if the node has a right child, and if so, we append the right child to the queue. 9. After this inner loop finishes, all the nodes of the next level have been added to the queue, and we proceed to the next iteration of the while loop to process the next level.

10. When there are no more nodes in the queue, it indicates that we have completed traversing the binary tree. At this point, the

Through this algorithm, we effectively conduct a breadth-first search (BFS) while taking a snapshot of the last node at each level, resulting in the view from the right side of the binary tree.

11. Finally, we return the ans list, which now contains the values of the rightmost nodes of each level, as seen from the right-hand

Let's take a small example to illustrate the solution approach. Consider the following binary tree:

2. Since the root is not None, we don't return an empty list but proceed with the next steps.

8. Now, the queue has the next level of nodes, so we loop back to the outer while loop.

14. Queue state: [5, 4]. The rightmost node is 4. Append 4 to ans, so ans = [1, 3, 4].

15. This level has two nodes. We enter the inner loop to process both nodes.

16. Pop 5 from the queue. 5 has no children, so we move on.

17. Pop 4 from the queue. 4 has no children as well.

9. Queue state: [2, 3]. The rightmost node now is 3. We add value 3 to ans, so ans = [1, 3].

4. Enter the while loop, as our queue has one element at this point.

Now, let us walk through the solution using the presented level-order traversal approach. 1. Start by initializing the answer list ans = [] which will hold the values of the rightmost nodes.

3. Initialize the queue q with the root node of the binary tree, which in our case is the node with the value 1. So, q = deque([1]).

7. Pop 1 from the queue using q.popleft(). Node 1 has two children: 2 (left child) and 3 (right child). Enqueue these children into q,

5. Queue state: [1]. The rightmost node is the last element of the queue, which is 1. Append 1 to ans, so ans = [1].

### 6. The number of nodes at the current level (root level) is 1. We proceed to process this level.

resulting in q = deque([2, 3]).

10. There are two nodes at this level. We begin the inner loop to process both.

11. Pop 2 from the queue using q.popleft(). Node 2 has no children, so we do not add anything to the queue.

13. All nodes of the current level are processed, loop back to the outer loop.

12. Pop 3 from the queue. It has two children: 5 (left child) and 4 (right child). Enqueue these children, making q = deque([5, 4]).

- 18. The queue is empty, so the outer while loop exits. 19. We have traversed the entire tree and recorded the rightmost node at each level.
- Through the level-order traversal, we select the last node's value at each level, aligning with the view one would see if they looked at the tree from its right side.

# Import the deque class from collections for queue implementation

# Use deque as a queue to hold the nodes at each level

# The rightmost element at the current level is visible from the right side

def \_\_init\_\_(self, val=0, left=None, right=None):

class Solution: 12 def rightSideView(self, root: Optional[TreeNode]) -> List[int]: 13 # Initialize the array to hold the right side view elements 14 right\_side\_view = [] 15 # If the tree is empty, return the empty list 16

20. The final answer list ans now contains [1, 3, 4], which are the values of the nodes visible from the right-hand side of the binary

### # If right child exists, add it to the queue 37 38 if current\_node.right: 39 40 # Return the list containing the right side view of the tree 41

Java Solution

1 import java.util.ArrayDeque;

2 import java.util.ArrayList;

import java.util.Deque;

import java.util.List;

TreeNode left;

#include <vector>

struct TreeNode {

int val;

14 class Solution {

8

9

10

11

13

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

10

11 }

**}**;

12 };

15 public:

TreeNode \*left;

TreeNode \*right;

if (!root) {

nodesQueue.push(root);

while (!nodesQueue.empty()) {

// Traverse the current level

if (currentNode->left) {

if (currentNode->right) {

nodesQueue.pop();

// Definition for a binary tree node.

TreeNode(): val(0), left(nullptr), right(nullptr) {}

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

// Function to get the right side view of a binary tree.

vector<int> rightView; // Stores the right side view elements

rightView.emplace\_back(nodesQueue.back()->val);

TreeNode\* currentNode = nodesQueue.front();

nodesQueue.push(currentNode->left);

nodesQueue.push(currentNode->right);

constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {

return rightView; // If the root is null, return an empty vector

queue<TreeNode\*> nodesQueue; // Queue to perform level order traversal

// Add the rightmost element of current level to the right view

for (int levelSize = nodesQueue.size(); levelSize > 0; --levelSize) {

// If left child exists, add it to the queue for next level

// If right child exists, add it to the queue for next level

return rightView; // Return the final right side view of the binary tree

vector<int> rightSideView(TreeNode\* root) {

TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

2 #include <queue>

TreeNode right;

class TreeNode {

int val;

tree.

Python Solution

class TreeNode:

6

8

9

10

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

42

43

9

11

from collections import deque

self.val = val

self.left = left

if root is None:

while queue:

queue = deque([root])

return right\_side\_view

// Definition for a binary tree node.

return right\_side\_view

# Continue until the queue is empty

for \_ in range(len(queue)):

if current\_node.left:

right\_side\_view.append(queue[-1].val)

current\_node = queue.popleft()

# Iterate over nodes at the current level

# Pop the node from the left side of the queue

# If left child exists, add it to the queue

queue.append(current\_node.left)

queue.append(current\_node.right)

self.right = right

# Definition for a binary tree node.

```
12
       TreeNode() {}
13
       TreeNode(int val) {
14
            this.val = val;
15
16
17
       // Constructor to initialize binary tree nodes with values and its children reference
18
       TreeNode(int val, TreeNode left, TreeNode right) {
19
20
           this.val = val;
           this.left = left;
            this.right = right;
23
24
25
   class Solution {
27
       // Function to get a list of integers representing the right side view of the binary tree
28
29
       public List<Integer> rightSideView(TreeNode root) {
30
           // Initialize an answer list to store the right side view
31
           List<Integer> answer = new ArrayList<>();
32
33
           // Return empty list if the root is null
           if (root == null) {
34
35
                return answer;
36
37
38
           // Initialize a dequeue to perform level order traversal
39
            Deque<TreeNode> queue = new ArrayDeque<>();
40
           // Add the root to the queue as the start of traversal
41
42
           queue.offer(root);
43
           // Perform a level order traversal to capture the rightmost element at each level
44
45
           while (!queue.isEmpty()) {
                // Get the rightmost element of the current level and add to the answer list
46
                answer.add(queue.peekLast().val);
48
49
               // Iterate through nodes at current level
                for (int n = queue.size(); n > 0; --n) {
50
51
                    // Poll the node from the front of the queue
52
                    TreeNode node = queue.poll();
53
54
                    // If left child exists, add it to the queue
                    if (node.left != null) {
55
56
                        queue.offer(node.left);
57
58
                    // If right child exists, add it to the queue
59
60
                    if (node.right != null) {
                        queue.offer(node.right);
61
62
63
64
65
           // Return the list containing the right side view of the tree
66
67
            return answer;
68
69 }
70
C++ Solution
```

### // Definition for a binary tree node. class TreeNode { val: number; left: TreeNode | null;

Typescript Solution

right: TreeNode | null;

this.val = val === undefined ? 0 : val;

this.left = left === undefined ? null : left;

this.right = right === undefined ? null : right;

```
12
13
   /**
    * Gets the values of the rightmost nodes at every level of the binary tree.
    * @param root - The root of the binary tree.
    * @returns An array of numbers representing the rightmost values for each level.
17
    */
   function rightSideView(root: TreeNode | null): number[] {
     // Initialize an array to hold the answer.
     const rightMostValues = [];
20
21
22
     // If the root is null, return the empty array.
23
     if (!root) {
24
        return rightMostValues;
26
     // Initialize a queue and enqueue the root node.
28
     const queue: TreeNode[] = [root];
29
30
     // Iterate as long as there are nodes in the queue.
31
     while (queue.length) {
       const levelSize = queue.length;
32
33
34
       // Get the last element of the queue (rightmost of this level)
35
       rightMostValues.push(queue[levelSize - 1].val);
36
37
       // Traverse the nodes of the current level.
       for (let i = 0; i < levelSize; i++) {</pre>
38
39
         // Remove the first element from the queue.
40
         const node = queue.shift();
41
         // If the node has a left child, add it to the queue.
42
         if (node.left) {
43
44
           queue.push(node.left);
45
46
47
         // If the node has a right child, add it to the queue.
         if (node.right) {
48
           queue.push(node.right);
49
50
51
52
53
54
     // Return the array of rightmost values.
55
     return rightMostValues;
56
57
Time and Space Complexity
Time Complexity
The given Python function rightSideView() performs a level-order traversal on a binary tree using a queue. In the worst case, it will
visit all nodes of the tree once. Therefore, if there are N nodes in the tree, the time complexity is:
0(N)
Space Complexity
```

# The space complexity is determined by the maximum number of nodes that can be held in the queue at any given time, which would

0(N)In a less balanced scenario, the space complexity will fluctuate, but it cannot exceed the total number of nodes in the tree, hence

be the maximum width of the tree. This width corresponds to the maximum number of nodes on any level of the tree. In the worst

case for a completely balanced binary tree, this would be N/2 (for the last level). Therefore, the space complexity is:

still O(N).