### 102. Binary Tree Level Order Traversal Breadth-First Search Binary Tree Medium Tree

pop and append operations) with the root node as the starting point.

structure to traverse the tree. The code involves the following steps:

We dequeue the node from q using popleft.

We add the node's value to the temporary list t.

If the node has a left child, we add it to the queue.

Similarly, if the node has a right child, we add it to the queue as well.

## **Problem Description** The given problem is about performing a level order traversal on a binary tree. In a level order traversal, we visit all the nodes of a

tree level by level, starting from the root. We start at the root level, then move to the nodes on the next level, and continue this process until all levels are covered. The goal is to return the values of the nodes in an array of arrays, where each inner array represents a level in the tree and contains the values of the nodes at that level, ordered from left to right. Intuition

size of our queue q.

next level.

neighbor nodes before moving to the next level. This characteristic of BFS makes it perfectly suited for level order traversal. To implement BFS, we use a queue data structure. The queue helps us process nodes in the order they were added, ensuring that we visit nodes level by level. Here's the step-by-step intuition behind the solution:

1. If the root is None, the binary tree is empty, and we have nothing to traverse. Therefore, we return an empty list.

To solve this problem, we use an algorithm known as <u>Breadth-First Search</u> (BFS). BFS is a traversal technique that explores the

3. We want to process each level of the tree one at a time. We do this by iterating while there are still nodes in our queue q. 4. For each level, we initialize a temporary list t to store the values of the nodes at this level.

5. Before we start processing the nodes in the current level, we note how many nodes there are in this level which is the current

2. Initialize an empty list ans to store the level order traversal result, and a queue q (implemented as a deque to allow for efficient

- 6. We then dequeue each node in this level from q, one by one, and add their values to our temporary list t. For each node we
- process, we check if they have a left or right child. If they do, we enqueue those children to q. This prepares our queue for the
- 7. Once a level is processed, we append our temporary list t to ans. 8. Continue the process until there are no more nodes left in the queue, meaning we have visited all levels.
- of the tree. The breadth-first nature of the queue ensures that we visit all nodes at a given depth before moving on to nodes at the next depth, fitting the needs of a level order traversal perfectly.

9. Return ans, which contains the level order traversal result. Each inner list within ans represents node values at a respective level

1. Initialization: We first define a class Solution with the method levelOrder, which takes the root of the binary tree as an input. We initialize an empty list ans to hold our results. If the root is None, we immediately return the empty list as there are no nodes

2. Queue Setup: We initialize a queue q (typically implemented as a double-ended queue deque in Python for efficient addition and

removal of elements). We add the root to the queue to serve as our starting point for the level order traversal.

The implementation of the solution primarily involves utilizing the Breadth-First Search (BFS) algorithm with the help of a queue data

# 3. Traversal Loop: We then enter a while loop, which will run until the queue is empty. This loop is responsible for processing all

queue. For each node:

problem.

Consider the following binary tree:

to traverse.

Solution Approach

levels in the tree. 4. Level Processing: Inside the loop, we start by creating a temporary list t to store node values for the current level. Since queue q currently holds all nodes at the current level, we use a for loop to process the number of nodes equal to the current size of the

5. Appending to Result: Once we finish processing all nodes at one level, we append temporary list t, which contains the values of these nodes, to our main result list ans.

6. Completion: After the while loop exits (the queue is now empty since all nodes have been visited), we return ans. The list ans

now holds the node values for each level of the binary tree, arranged from top to bottom and left to right, as required by the

The BFS approach ensures we process all nodes at one level before moving onto the next, aligning with the level order traversal

solution effectively combines data structure utilization with traversal strategy to solve the problem efficiently.

3. Traversal Loop: The queue is not empty (it contains the root), so we start the while loop.

We dequeue the root node (value 1) from the queue and add it to list t (which is now [1]).

• The root node has two children, nodes with values 2 and 3, which we add to the queue.

5. Appending to Result: We've finished processing the first level and append t to ans (which is now [[1]]).

• Dequeue node with value 3, add it to t, and enqueue its children (values 5 and 6 to queue).

definition. By taking advantage of the queue to keep track of nodes to visit and maintaining a list for each level's node values, the

After we process all nodes at the current level, their children are in the queue, ready to be processed for the next level.

**Example Walkthrough** Let's illustrate the solution approach with a small binary tree example.

1. Initialization: We create an instance of Solution and call the method levelorder with the root of the tree (node with value 1). We also initialize an empty list ans to store the result of the traversal. 2. Queue Setup: We initialize a queue q and add the root node with value 1 to it.

## Create a temporary list t to store node values for this level. • The current queue size is 1, so the loop runs for one iteration.

4. Level Processing - First Iteration

6. Level Processing - Second Iteration

 Since the queue has two elements (values 2 and 3), we process two nodes this round. Dequeue node with value 2, add it to temporary list t, and enqueue its child (value 4 to queue).

represents the level order traversal of the tree.

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

• This time queue size is 3, and we process nodes 4, 5, and 6 similarly.

We end up with t having [4, 5, 6], and the queue is now empty.

Here's how the level order traversal would process this tree:

 At the end of this iteration, t has [2, 3], and the queue has nodes 4, 5, and 6. 7. Appending to Result: We've finished processing the second level and append t to ans (which is now [[1], [2, 3]]). 8. Level Processing - Third Iteration

10. Completion: The queue is empty, so the loop ends and we return ans. The final result is [[1], [2, 3], [4, 5, 6]], which correctly

The traversal has visited all nodes level by level and has constructed a list of node values for each level, fitting the problem

Python Solution

9. Appending to Result: The third level is processed, and we append t to ans (which is now [[1], [2, 3], [4, 5, 6]]).

24 # Temporary list to store the values of nodes at the current level level\_values = [] 27 # Iterate over nodes at the current level for \_ in range(len(queue)):

# Use a queue to keep track of nodes to visit, starting with the root node

# If the node has a left child, add it to the queue for the next level

# Pop the node from the left side of the queue

# Append the node's value to the temporary list

current\_node = queue.popleft()

if current\_node.left:

level\_values.append(current\_node.val)

queue.append(current\_node.left)

self.val = val self.left = left self.right = right class Solution: def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]: # Initialize a list to hold the level order traversal result 12 result = [] 13 14 # If the tree is empty, return the empty list 15 if root is None: 16

17

18

19

20

23

29

30

31

32

33

34

35

36

28

29

30

31

32

33

34

35

36

37

38

39

40

42

43

44

45

46

47

48

49

50

51

52

53

54

56

57

return result;

queue.offer(root);

while (!queue.isEmpty()) {

// Create a queue to hold nodes at each level.

// Start the level order traversal from the root.

List<Integer> level = new ArrayList<>();

for (int i = 0; i < levelLength; ++i) {</pre>

level.add(currentNode.val);

if (currentNode.left != null) {

if (currentNode.right != null) {

queue.offer(currentNode.left);

// Process all nodes at the current level.

TreeNode currentNode = queue.poll();

// Retrieve and remove the head of the queue.

// Add the node's value to the temporary list.

// Temporary list to store the values of nodes at the current level.

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

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

Deque<TreeNode> queue = new ArrayDeque<>();

// While there are nodes to process

int levelLength = queue.size();

requirements perfectly.

class TreeNode:

from collections import deque

# Definition for a binary tree node.

return result

queue = deque([root])

while queue:

# Loop until the queue is empty

```
37
                   # If the node has a right child, add it to the queue for the next level
38
                   if current_node.right:
39
                       queue.append(current_node.right)
40
               # Append the list of values for this level to the result list
41
               result.append(level_values)
42
43
           # Return the result list containing the level order traversal
44
45
           return result
Java Solution
1 import java.util.List;
2 import java.util.ArrayList;
  import java.util.Deque;
   import java.util.ArrayDeque;
   // Definition for a binary tree node.
   class TreeNode {
       int val;
       TreeNode left;
9
       TreeNode right;
       TreeNode() {}
11
12
       TreeNode(int val) { this.val = val; }
       TreeNode(int val, TreeNode left, TreeNode right) {
         this.val = val;
           this.left = left;
           this.right = right;
16
17
18 }
19
   class Solution {
21
       // Method to perform a level order traversal of a binary tree.
       public List<List<Integer>> levelOrder(TreeNode root) {
           // Create a list to hold the result.
24
           List<List<Integer>> result = new ArrayList<>();
25
26
           // Return an empty list if the tree is empty.
27
           if (root == null) {
```

#### 66 67 68 69 } 70

```
queue.offer(currentNode.right);
58
59
60
61
62
               // Add the temporary list to the result list.
63
               result.add(level);
64
65
           // Return the list of levels.
           return result;
C++ Solution
  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) {}
  9
         TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 10
 11 };
 12
    class Solution {
    public:
 14
 15
         /**
          * Performs a level order traversal of a binary tree and returns the values of nodes at each level
 16
          * as a 2D vector.
          * @param root The root node of the binary tree.
 19
          * @return A 2D vector where each inner vector contains the values of nodes at the corresponding level.
 20
 21
 22
         vector<vector<int>> levelOrder(TreeNode* root) {
 23
             vector<vector<int>> levels; // Stores the values of nodes at each level
             if (!root) return levels; // If the tree is empty, return an empty vector
 24
 25
 26
             // Initialize a queue to assist in BFS traversal
 27
             queue<TreeNode*> nodesQueue;
             nodesQueue.push(root);
 28
 29
             while (!nodesQueue.empty()) {
 30
 31
                 int levelSize = nodesQueue.size(); // Number of nodes at the current level
 32
                                                    // Vector to store values of node at current level
                 vector<int> currentLevel;
 33
 34
                 while (levelSize > 0) {
 35
                     TreeNode* currentNode = nodesQueue.front(); // Take the front node in the queue
 36
                     nodesQueue.pop();
                                                                 // Remove that node from the queue
 37
 38
                     // Add current node's value to current level vector
 39
                     currentLevel.push_back(currentNode->val);
 40
                     // If the current node has a left child, add it to the queue
 41
                     if (currentNode->left) {
 42
                         nodesQueue.push(currentNode->left);
 43
 44
 45
                     // If the current node has a right child, add it to the queue
                     if (currentNode->right) {
 46
                         nodesQueue.push(currentNode->right);
 47
 48
 49
 50
                     levelSize--; // Decrement the counter for nodes remaining at current level
 51
 52
 53
                 // After processing all nodes at the current level, add the current level vector to the results
                 levels.push_back(currentLevel);
 54
 55
 56
 57
             return levels; // Return the 2D vector with all the levels' values
```

#### 25 // Proceed if the currentNode is not null. 26 if (currentNode) { 27 // Add the value to the current level's result. currentLevelValues.push(currentNode.val); 28 29

return result;

58

59

60

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

30

31

32

33

34

37

38

39

42 }

43

**}**;

Typescript Solution

if (root === null) {

return result;

function levelOrder(root: TreeNode | null): number[][] {

// Initialize a queue to hold nodes at each level.

// Process all nodes at the current level.

const currentNode = queue.shift();

const currentLevelValues: number[] = [];

for (let i = 0; i < levelLength; i++) {</pre>

// Get the number of nodes at the current level.

// Shift the first node from the queue.

// Add left and right children if they exist.

// Add the current level's values to the overall result array.

// Return the result array containing the level order traversal.

if (currentNode.left) queue.push(currentNode.left);

if (currentNode.right) queue.push(currentNode.right);

// Initialize the result array.

const queue: TreeNode[] = [root];

while (queue.length !== 0) {

// Iterate until the queue is empty.

const levelLength = queue.length;

result.push(currentLevelValues);

the tree is visited exactly once during the traversal.

const result: number[][] = [];

Time and Space Complexity The given Python code implements a level order traversal of a binary tree, where root is the root of the binary tree.

// Function to perform level order traversal on a binary tree and return the values in level—wise order.

// If the root is null, the tree is empty, and we return the empty result.

# **Space Complexity:**

**Time Complexity:** 

The space complexity of the algorithm can be analyzed in terms of two components: the space used by the data structure q (which holds the nodes at the current level), and the space for the ans list.

The time complexity of this algorithm is O(n), where n is the total number of nodes in the binary tree. This is because each node in

a complete binary tree or has the most nodes at one level. Space is needed to store all nodes of the widest level in the data structure q at one time. Therefore, the overall space complexity of the algorithm is O(w) if we only consider the space taken by the queue. If we add the

space taken by the output list ans, the total space complexity remains O(n) since ans will eventually contain all nodes of the tree.

The worst-case space complexity is O(w), where w is the maximum width of the tree. This worst case occurs when the binary tree is