## 116. Populating Next Right Pointers in Each Node **Linked List Binary Tree** Medium **Depth-First Search Breadth-First Search**

**Problem Description** 

two children. Each node in the tree has an additional pointer called next in addition to the usual left and right child pointers. Initially, all next pointers are set to NULL. The goal is to modify the tree such that each node's next pointer points to the following node on the same level. For the nodes at the far right of each level, since there is no next right node, their next pointer should remain set to NULL.

In this problem, we are dealing with a perfect binary tree where all leaves are at the same level, and every parent node has exactly

Intuition The solution approach is based on a level-order traversal (also known as a breadth-first search) of the tree. Since we're dealing with

a perfect binary tree, every level is fully filled, and therefore the problem simplifies to connecting all nodes at each level in a left-to-

right manner.

Here is the intuition step by step: 1. We initialize a queue that will store nodes at the current level we're processing. 2. We begin traversal with the root node by adding it into the queue.

3. We start a loop that will continue until the queue is empty, which means there are no more levels to process. 4. Inside this loop, we initialize a variable p to None, to keep track of the previous node as we process.

that level.

traversing the tree.

11

as levels):

5 4 5 6 7

node = q.popleft()

p.next = node

q.append(node.left)

q.append(node.right)

Process the nodes until the queue is empty.

Enqueue its children, queue becomes deque([2, 3]).

if p:

p = node

if node.left:

if node.right:

- 5. We then process nodes level by level:
- At the start of a level, we record the number of nodes at that level (the queue's length) to ensure we only process those nodes before moving to the next level.
- We pop nodes off of the queue left to right, and for each node: a. If p is not NULL (this is not the first node of the level), we set p's next to the current node. b. We then update p to the current node. c. We add the current node's left and right children to the queue.
- The given solution ensures that each node at a given level points to its adjacent node on the right before descending to the next level. It takes advantage of the queue data structure for keeping track of nodes level by level and updates the next pointers in a systematic manner.

6. By the end of the loop, all next pointers should accurately point to the next right node, or remain NULL if it's the far-right node of

- **Solution Approach**
- The implementation of the solution is primarily based on the <u>Breadth-First Search</u> (BFS) algorithm, which is a common way to

3. We enter a while loop that will run as long as there are elements in the queue.

7. Finally, we return the modified tree starting with the root node.

and efficient appending and popping of elements from both ends. Here is a breakdown of the implementation steps, referring to the data structures and patterns used:

according to the BFS traversal order. The Python deque from the collections module is used for the queue because it allows for fast

traverse trees level by level. To make this possible, we employ a queue data structure, which will hold the nodes of the tree

# 1. We first check if the root is None. If it is, we return None as there is nothing to connect. 2. We initialize a queue q with the root node of the tree.

4. In each iteration of the while loop, we first initialize a variable p to None, which will serve as a pointer to the previous node at the current level. 5. We determine the number of nodes at the current level of the tree by examining the length of the queue using len(q). 6. We then use a for loop to iterate over all nodes at the current level: We use q.popleft() to remove and obtain the first node from the queue.

• We check if p is not None. This implies that this is not the first node at the current level, so we set the next of p to the current

node. We set p to the current node after updating its next.

- If the current node has a left child, we append it to the queue, and similarly, if it has a right child, we append it to the queue. 7. Finally, once the while loop finishes executing, all nodes' next pointers have been properly set, and we return the root of the
- updated tree. This solution utilizes the properties of a perfect binary tree and a level-order traversal algorithm to connect the nodes at each level

efficiently. With the help of the queue, we are able to process nodes in the correct order and update their next pointers while

Here's a snippet of the key part of the implementation that demonstrates the core process: 1 while q: p = Nonefor \_ in range(len(q)):

are made between nodes at the same level. **Example Walkthrough** 

Let's consider a small example of a perfect binary tree to illustrate the solution approach. Our example tree is as follows (represented

In this snippet, the loop structures and the assignments are critical in understanding how the connections (based on next pointers)

Initialize queue as deque([1]).

op = None

• Level 1: Nodes 2 and 3

• p is updated to node 4.

1 -> NULL

2 -> 3 -> NULL

Python Solution

class Solution:

13

14

16

17

18

19

20

25

26

27

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52 53

54

55

56

57

58

59

60

61

62

63

64

65

67

66 }

self.left = left

self.next = next

self.right = right

each level is set to None.

:type root: Optional[Node]

if node.left:

if node.right:

:rtype: Optional[Node]

return root

:param root: The root of the binary tree.

5 4->5->6->7 -> NULL

updating p to the latest node.

Now let's walkthrough each level:

• Level 0: Tree root (Node 1)

 Node 3's children (6, 7) are enqueued, queue becomes deque([4, 5, 6, 7]). • Level 2: Nodes 4, 5, 6, and 7

• p = None, we then dequeue node 2 and since p is None, next is not set.

• There is only one node at this level, so we don't have to set any next pointers.

To begin, here is an outline of the steps we'll follow to execute the algorithm described:

1. Create an empty queue (will use a deque for efficiency) and enqueue the root node (node 1).

- o p = None, we then dequeue node 4 and since p is None, next is not set.
- At the end of this process, here's how the next pointers would be set:

After node 7, the queue is empty, and the next pointer is not updated as it is the last node at this level.

Continue with nodes 5, 6, and 7 in the same way, dequeuing each node, setting its next pointer to the subsequent node, and

All the next pointers at the same level have been connected, and the next pointers of the last nodes at each level remain pointing to

∘ p is updated to node 2. Node 2's children (4, 5) are enqueued, queue becomes deque([3, 4, 5]).

• We dequeue node 3, and set next of node 2 to node 3 (p.next = node), p is updated to node 3.

class Node: """ Definition for a binary tree node with a next pointer. """ def \_\_init\_\_(self, val: int = 0, left: 'Node' = None, right: 'Node' = None, next: 'Node' = None): self.val = val

from collections import deque # Importing deque from collections module for queue functionality

def connect(self, root: 'Optional[Node]') -> 'Optional[Node]':

Connects each node to its next right node in a binary tree.

:return: The root of the binary tree with next pointers set.

queue = deque([root]) # Initialize queue with the root node

previous\_node.next = node

queue.append(node.left)

queue.append(node.right)

// Get the next node from the queue.

previousNode.next = currentNode;

queue.offer(currentNode.left);

queue.offer(currentNode.right);

// Link the previous node (if any) to the current one.

// Connects all nodes at the same level in a binary tree using 'next' pointers.

// Current node becomes the previous node for the next iteration.

// Add the children of current node to the queue for next level processing.

Node currentNode = queue.poll();

if (previousNode != null) {

previousNode = currentNode;

// Return the root of the modified tree.

return root;

Node\* connect(Node\* root) {

return root;

if (!root) {

if (currentNode.left != null) {

if (currentNode.right != null) {

// If the tree is empty, return the root directly.

if root is None: # If the tree is empty, return the empty root

It is perfect binary tree which means all leaves are on the same level,

and every parent has two children. The next pointer of the last node in

node = queue.popleft() # Remove the node from the queue

# Add left and right children to the queue if they exist

previous\_node = node # Update previous node to the current node

NULL, signifying the end of the level. This completes the walk through of the tree using the given solution approach.

28 while queue: 29 previous\_node = None # Initialize previous node as None 30 level\_size = len(queue) # Current level's size in the binary tree 31 32 for \_ in range(level\_size): # Traverse all nodes at the current level

```
43
               # At the end of the level, previous_node is the last node and its next should already be None
44
45
46
           return root # Return the modified tree root with next pointers connected
47
Java Solution
  // Definition for a binary tree node with a next pointer.
  class Node {
       public int val;
       public Node left;
       public Node right;
       public Node next;
 6
       public Node() {}
8
       public Node(int _val) {
10
11
           val = _val;
12
13
       public Node(int _val, Node _left, Node _right, Node _next) {
14
15
           val = _val;
16
           left = _left;
17
           right = _right;
18
           next = _next;
19
20 }
21
   class Solution {
23
       // Method connects each node to its next right node in the same level.
24
       // If there is no next right node, the next pointer should be set to NULL.
25
       public Node connect(Node root) {
26
           // Handle the base case when the tree is empty.
           if (root == null) {
28
               return root;
29
30
31
           // Queue to hold the nodes to be processed.
32
           Deque<Node> queue = new ArrayDeque<>();
           // Start with the root node.
33
           queue.offer(root);
34
35
36
           // Traverse the binary tree level by level.
37
           while (!queue.isEmpty()) {
38
               // Previous node on the current level.
39
               Node previousNode = null;
40
               // Process all nodes in the current level.
41
42
               for (int i = queue.size(); i > 0; --i) {
```

if previous\_node: # If there is a previous node, set its next pointer to the current node

## 15 16 17 18

C++ Solution

1 class Solution {

2 public:

```
// Initialize the queue with the root of the tree.
10
           queue<Node*> nodesQueue;
11
           nodesQueue.push(root);
12
13
           // Process nodes level by level.
14
           while (!nodesQueue.empty()) {
               // The previous node at the current level, initialized as nullptr.
               Node* prevNode = nullptr;
                // Number of nodes at the current level.
19
20
                int levelNodeCount = nodesQueue.size();
21
22
               // Iterate over all nodes in the current level.
23
                for (int i = 0; i < levelNodeCount; ++i) {</pre>
24
                   // Retrieve and remove the node from the front of the queue.
25
                    Node* currentNode = nodesQueue.front();
                    nodesQueue.pop();
26
27
28
                    // Connect the previous node with the current node.
                    if (prevNode) {
29
                        prevNode->next = currentNode;
30
31
                    // Update the previous node to the current node.
                    prevNode = currentNode;
33
34
35
                    // Add the left child to the queue if it exists.
                    if (currentNode->left) {
36
                        nodesQueue.push(currentNode->left);
37
38
                    // Add the right child to the queue if it exists.
39
                    if (currentNode->right) {
40
                        nodesQueue.push(currentNode->right);
43
               // The last node in the level points to nullptr, which is already the case.
45
46
           // Return the updated tree root with all 'next' pointers set.
48
           return root;
49
50 };
51
Typescript Solution
   // TypeScript definition of the Node class structure
   interface INode {
       val: number;
       left: INode | null;
       right: INode | null;
       next: INode | null;
 6
 8
    * Connects each node to its next right node in a perfect binary tree.
```

// If the tree is empty or if it's a single node tree, return the root since there are no next pointers to set.

// If next pointer is available for the current node, connect the current's right child to the next's left child.

### if (root.next !== null) { 24 25 root.right.next = root.next.left; 26 27 28 // Recursively connect the left and right subtrees. connect(root.left); 29

return root;

return root;

connect(root.right);

root.left.next = root.right;

<u>Time and Space Complexity</u>

13

15

17

18

19

20

21

22

23

30

31

32

33

35

34 }

\*/

\* @param root - The root node of the binary tree.

function connect(root: INode | null): INode | null {

if (root === null || root.left === null) {

\* @returns The root node of the modified tree with connected next pointers.

// Return the root node after completing the next pointer connections.

// Connect the left child's next pointer to the right child.

**Time Complexity** The time complexity of the code is O(N), where N is the total number of nodes in the binary tree. The algorithm employs a breadthfirst search approach, where each node is visited exactly once. Because we visit each node once and perform a constant amount of work for each node (like setting the next pointers and pushing children to the queue), the overall time complexity is linear with respect to the number of nodes.

# **Space Complexity**

The space complexity of the code is O(N). The space is mainly occupied by the queue q, which in the worst-case scenario can contain all nodes at the deepest level of the binary tree. In a perfect binary tree, the last level could contain approximately N/2 nodes, where N is the total number of nodes. Thus, the space complexity is proportional to the number of nodes in the bottom level of the tree, making it O(N) in the worst case.