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

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

Leetcode Link

set to NULL.

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.

4. Inside this loop, we initialize a variable p to None, to keep track of the previous node as we process. 5. We then process nodes level by level:

systematic manner.

current level.

1 while q:

10

11

as levels):

5 4 5 6 7

p = None

if p:

Example Walkthrough

p = node

if node.left:

if node.right:

for _ in range(len(q)):

node = q.popleft()

p.next = node

q.append(node.left)

q.append(node.right)

children to the queue.

- 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

3. We start a loop that will continue until the queue is empty, which means there are no more levels to process.

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

The implementation of the solution is primarily based on the Breadth-First Search (BFS) algorithm, which is a common way to

Solution Approach

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

traverse trees level by level. To make this possible, we employ a queue data structure, which will hold the nodes of the tree according to the BFS traversal order. The Python deque from the collections module is used for the queue because it allows for fast

1. We first check if the root is None. If it is, we return None as there is nothing to connect.

We use q.popleft() to remove and obtain the first node from the queue.

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:

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

2. We initialize a queue q with the root node of the tree. 3. We enter a while loop that will run as long as there are elements in the queue.

6. We then use a for loop to iterate over all nodes at the current level:

• 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.

7. Finally, once the while loop finishes executing, all nodes' next pointers have been properly set, and we return the root of the

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.

5. We determine the number of nodes at the current level of the tree by examining the length of the queue using len(q).

- 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
- traversing the tree. Here's a snippet of the key part of the implementation that demonstrates the core process:

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

In this snippet, the loop structures and the assignments are critical in understanding how the connections (based on next pointers) are made between nodes at the same level.

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

1. Create an empty queue (will use a deque for efficiency) and enqueue the root node (node 1). 2. Process the nodes until the queue is empty.

Enqueue its children, queue becomes deque([2, 3]). • Level 1: Nodes 2 and 3

2 -> 3 -> NULL

Python Solution

self.val = val

self.left = left

self.next = next

self.right = right

:rtype: Optional[Node]

return root

while queue:

class Node:

class Solution:

10

13

14

21

23

25

26

27

28

29

30

31

32

33

34

35

36

39

40

41

42

43

44

45

46

47

48

49

50

51

52 53

54

55

56

57

58

Node previousNode = null;

// Process all nodes in the current level.

// 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.

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

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

for (int i = queue.size(); i > 0; --i) {

Node currentNode = queue.poll();

if (previousNode != null) {

previousNode = currentNode;

if (currentNode.left != null) {

if (currentNode.right != null) {

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

• Level 2: Nodes 4, 5, 6, and 7

updating p to the latest node.

o p = None

Now let's walkthrough each level:

• Level 0: Tree root (Node 1)

Initialize queue as deque([1]).

• p is updated to node 4. o Continue with nodes 5, 6, and 7 in the same way, dequeuing each node, setting its next pointer to the subsequent node, and

• p = None, we then dequeue node 4 and since p is None, next is not set.

Node 3's children (6, 7) are enqueued, queue becomes deque([4, 5, 6, 7]).

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

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

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

At the end of this process, here's how the next pointers would be set: 1 -> NULL

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

∘ 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.

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

def __init__(self, val: int = 0, left: 'Node' = None, right: 'Node' = None, next: 'Node' = None):

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

and every parent has two children. The next pointer of the last node in 16 17 each level is set to None. 18 19 :param root: The root of the binary tree. 20 :type root: Optional[Node]

previous_node.next = node

""" Definition for a binary tree node with a next pointer. """

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

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

previous_node = None # Initialize previous node as None

level_size = len(queue) # Current level's size in the binary tree

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

for _ in range(level_size): # Traverse all nodes at the current level

previous_node = node # Update previous node to the current node

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

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

```
37
38
                    # Add left and right children to the queue if they exist
39
                    if node.left:
                        queue.append(node.left)
40
                    if node.right:
41
42
                        queue.append(node.right)
43
44
               # At the end of the level, previous_node is the last node and its next should already be None
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.
 2 class Node {
       public int val;
       public Node left;
       public Node right;
       public Node next;
       public Node() {}
 8
       public Node(int _val) {
10
11
           val = _val;
12
13
       public Node(int _val, Node _left, Node _right, Node _next) {
14
           val = _val;
15
            left = _left;
16
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<>();
33
           // Start with the root node.
           queue.offer(root);
34
35
36
           // Traverse the binary tree level by level.
37
           while (!queue.isEmpty()) {
38
               // Previous node on the current level.
```

26 27 28 29 30

```
59
60
61
62
           // Return the root of the modified tree.
63
64
           return root;
65
66 }
67
C++ Solution
1 class Solution {
   public:
       // Connects all nodes at the same level in a binary tree using 'next' pointers.
       Node* connect(Node* root) {
           // If the tree is empty, return the root directly.
           if (!root) {
               return root;
           // Initialize the queue with the root of the tree.
10
           queue<Node*> nodesQueue;
11
           nodesQueue.push(root);
12
           // Process nodes level by level.
14
           while (!nodesQueue.empty()) {
15
               // The previous node at the current level, initialized as nullptr.
16
17
               Node* prevNode = nullptr;
18
               // 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.
                   Node* currentNode = nodesQueue.front();
25
                   nodesQueue.pop();
                   // Connect the previous node with the current node.
                    if (prevNode) {
                        prevNode->next = currentNode;
31
                   // Update the previous node to the current node.
33
                    prevNode = currentNode;
34
                   // Add the left child to the queue if it exists.
35
                   if (currentNode->left) {
36
                        nodesQueue.push(currentNode->left);
37
                   // Add the right child to the queue if it exists.
39
                    if (currentNode->right) {
                        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
```

return root; 18 19 20 21

respect to the number of nodes.

of the tree, making it O(N) in the worst case.

interface INode {

6

8

13

15

22

23

24

25

26

27

28

29

30

31

32

33

35

*/

val: number;

left: INode | null;

right: INode | null;

next: INode | null;

// TypeScript definition of the Node class structure

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

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

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

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

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

```
// Connect the left child's next pointer to the right child.
       root.left.next = root.right;
       // 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) {
           root.right.next = root.next.left;
       // Recursively connect the left and right subtrees.
       connect(root.left);
       connect(root.right);
       // Return the root node after completing the next pointer connections.
       return root;
34 }
Time and Space Complexity
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 breadth-
first 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
```

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

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