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

Problem Description In this LeetCode problem, we are given a binary tree where each node has an additional pointer called next. This tree structure is

different from the standard binary tree structure, which typically only includes left and right pointers to represent the node's children. The next pointer in each node should be used to point to its immediate right neighbor on the same level. If no such neighbor exists, the next pointer should be set to NULL.

To clarify, if we imagine the nodes being aligned at each level from left to right, the next pointer of a node should reference the adjacent node to the right within the same level. For the rightmost nodes of each level, since there is no node to their right, their next pointer will remain NULL as initialized. Initially, all the next pointers of the nodes in the tree are set to NULL. Our task is to modify the tree such that all next pointers are

Intuition The solution involves a level-order traversal (BFS-like approach) of the binary tree, where we iterate through nodes level by level.

Since we need to connect nodes on the same level from left to right, we keep track of the first node on the new level that we visit while still iterating on the current level. This way, we have a reference when we need to move down a level. To solve this, we need variables to keep track of:

the next variable to point to the first node at the next lower level when moving down the tree.

correctly assigned according to the problem's rules.

- 3. The current node that we're attaching to the next pointer of the previous node (curr). A helper function modify takes care of the linking process, linking the current node to the previous one if it exists, and also updating
- We use a while loop to navigate through the levels, and inside the loop, we traverse the current level using the next pointers (which are already pointing to the right or are NULL for the rightmost nodes). As we go, we use the modify helper function to link the children of the current node properly. We then transition down to the next level of the tree using the next variable, which we've already set to

1. The previous node on the current level (prev), so we can set its next pointer to the current node.

2. The first node on the next level (next), which is where we will move after finishing the current level.

The traversal repeats until there are no more levels to process (i.e., when we've processed the rightmost node at the lowest level of the tree), ensuring that all next pointers are appropriately linked.

Solution Approach The implementation for this problem involves a clever usage of pointers to traverse the binary tree level by level without using additional data structures like queues, which are commonly used for level-order traversal. Here's a step-by-step walkthrough of the process using the given Python code:

1. Initialize Pointers: We start by creating a node pointer that points to the root of the tree. The loop that follows will process one level at a time. Two additional pointers, prev and next, are used to track the previous node and the first node on the next level, respectively.

2. Outer While Loop: The outer loop continues until node is None. This loop ensures that we visit all the levels of the binary tree.

on the current level, and next will be updated to point to the first node of the next level as soon as we encounter it.

3. Set Level Pointers: At the beginning of each iteration, we set prev and next to None. Here, prev will hold the last node we visited

updating node = node.next.

Example Walkthrough

the first node on that level.

4. Inner While Loop: The inner loop processes each level. It continues while there is a node to process on the current level. 5. Helper Function - modify: Each iteration within the inner while loop calls the modify function with the node. left and node. right children. This function does three things:

o If there's a previous of link its next pointer to the current child, curr. This connects the previous child on the same level to the current one.

Assigns curr to prev because for the next child, curr will be the previous node.

which now has all its next pointers correctly set, is returned.

the next pointers that we're setting along the way.

The inner while loop processes the top level.

next to node 2 (the first child of this level).

• prev is then updated to 2.

The first non-None child encountered will be the starting node for the next level, and we update next to point to this child.

• Checks if the current child curr is None. If it is, we don't have anything to modify or link, so we return immediately.

7. Move to Next Level: Once the inner loop is done, we have processed all nodes on the current level, and next points to the first node of the next level. We set node = next to move down the <u>tree</u> and begin processing the next level.

8. Return Modified Tree: The process continues until all nodes have been processed. The outer loop ends, and the original root,

Throughout the process, we are effectively doing a BFS traversal without an auxiliary queue, using next pointers to navigate through

the tree instead. The solution leverages the fact that we can link the next level's nodes while we are still on the current level, using

6. Traverse Current Level: After the children of the current node are processed, we move to the next node on the same level by

- To illustrate the solution approach, consider a binary tree with the following structure:
- We start by initiating a variable node to point to the root of the tree, in this case, the node with value 1. The variables prev and next are initialized to None. The outer while loop starts, indicating we're beginning with the top level of the tree.

• The modify helper function is called first with node.left (2) and then with node.right (3). Since prev is None, we just update

• Since node 3 is the rightmost node at this level, its next pointer is left as None. • Finally, the inner loop concludes, as there are no more nodes on the top level.

· At the second level:

node.right of 3.

1 -> NULL

2 -> 3 -> NULL

4 ->5 -> 7 -> NULL

Python Solution

class Node:

class Solution:

1111111

13

14

15

38

39

40

41

42

10

11

12

13

16

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

46

47

48

49

50

52

53

54

55

56

57

59

58 }

2 # Definition for a Node.

self.val = val

self.left = left

self.next = next

self.right = right

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

current_node = next_start_node

Node currentLevelNode = root;

nextLevelStart = null;

previous = null;

while (currentLevelNode != null) {

// Outer loop: Traverse levels of the tree

while (currentLevelNode != null) {

// Proceed to the next level

// Return the modified tree's root

if (currentNode == null) {

// to the current node

if (previous != null) {

previous = currentNode;

private void modifyPointer(Node currentNode) {

nextLevelStart = currentNode;

previous.next = currentNode;

// Update the previous pointer to the current node

// If current node is null, do nothing

currentLevelNode = nextLevelStart;

// Modify the left child pointer

// Modify the right child pointer

modifyPointer(currentLevelNode.left);

modifyPointer(currentLevelNode.right);

// Helper function to connect child nodes at the current level

// If a previous node was found, link the previous node's next pointer

Return the root with updated next pointers.

def connect_nodes(curr: 'Node'):

nodes to their right.

At the top level:

 We start with node 2. Again, the inner loop processes the level. • The modify function is called first with node.left (4) and then with node.right (5). next is updated to 4, and prev to 4.

• We connect 2 to 3 by setting 2.next to 3 through the modify function with node.right.

We switch levels by setting node to next, which is node 2, and prev and next to None.

o prev (4) is connected to 5 via 5's next pointer because 5 is node. right of 2.

After the outer while loop exits, we have successfully connected all the next pointers:

• After finishing node 3, the inner loop concludes, as there are no more nodes at this level with a non-NULL next pointer. • We again switch levels by setting node to next, which is node 4. However, nodes 4, 5, and 7 have no children, so subsequent iterations of the outer loop will not modify any next pointers.

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

The tree's next pointers have been set up to link nodes together across each level from left to right, with the rightmost nodes pointing to NULL, as required by the problem. All the intended connections have been made while respecting the binary tree structure and the rule stating that only immediate right neighbors should be linked. The solution also ensures that no additional data structures are used by leveraging the next pointers and processing the tree level by level.

o On to node 3, its children are None and 7. So, modify skips the None child, and connects 5 (the prev) to 7, since 7 is

• The rightmost nodes (5 and 7) have their next pointers already as None, and they stay that way since there are no more

if curr is None: 16 17 return 18 # Initialize the start of the next level if it hasn't been set yet. next_start_node = next_start_node or curr 19 # If there is a previous node on the same level, link it to the current node.

Helper function to connect nodes at the same level.

nonlocal previous_node, next_start_node

20 if previous_node: 22 previous_node.next = curr 23 # Update the previous_node to the current one for the next iteration. 24 previous_node = curr 26 current_node = root # Iterate through the levels of the tree. while current_node: 29 # Reset previous_node and next_start_node for the next level. 30 previous_node = next_start_node = None 31 # Iterate nodes in the current level and connect child nodes. 32 while current_node: 33 connect_nodes(current_node.left) 34 connect_nodes(current_node.right) 35 # Move to the next node in the current level. 36 current_node = current_node.next 37 # Proceed to the next level.

return root

Java Solution class Solution { // Class-level variables to hold the previous node and // the next level's start node private Node previous; private Node nextLevelStart; public Node connect(Node root) {

// Inner loop: Traverse nodes within the current level

// Move to the next node in the current level

currentLevelNode = currentLevelNode.next;

// Reset previous and nextLevelStart pointers when moving to a new level

40 return; 41 42 43 // If this is the first node of the next level, // update the nextLevelStart pointer 44 if (nextLevelStart == null) { 45

return root;

C++ Solution 1 class Solution { 2 public: Node* connect(Node* root) { Node* currentNode = root; // Start with the root of the tree Node* previousNode = nullptr; // This will keep track of the previous node on the current level Node* nextLevelStart = nullptr; // This will point to the first node of the next level // Lambda function to connect nodes at the same level auto connectNodes = [&](Node* childNode) { if (!childNode) { 10 return; // If the child node is null, no connection can be made 11 12 13 if (!nextLevelStart) { nextLevelStart = childNode; // Set the start of the next level, if it hasn't been set 14 15 if (previousNode) { 16 17 previousNode->next = childNode; // Connect the previous node to the current child node 18 previousNode = childNode; // Current child node becomes the previous node for the next iteration 19 **}**; 20 21 22 // Traverse the tree by level 23 while (currentNode) { 24 previousNode = nextLevelStart = nullptr; // Reset pointers for each level 25 26 // Connect children nodes within the current level while (currentNode) { 27 28 connectNodes(currentNode->left); // Connect left child if it exists 29 connectNodes(currentNode->right); // Connect right child if it exists currentNode = currentNode->next; // Move to the next node at the same level 30 31 32 currentNode = nextLevelStart; // Move down to the start of the next level 33 34 35 36 return root; // Return the modified tree with next pointers connected 37 38 }; 39 Typescript Solution 1 // Definition for a Node. 2 class Node { val: number; left: Node | null;

constructor(val?: number, left?: Node, right?: Node, next?: Node) {

16 // This function connects each node on the same level with the following node

17 // (to its right) in a binary tree and returns the root of the tree. It uses a

18 // level order traversal method utilizing a queue to process nodes level by level.

// If the root is null, the tree is empty; thus return the root as is.

// Variable to keep track of the previous node to set the next pointer.

// If the current node has a right child, add it to the queue.

// If there was a previous node in this level, connect its next to the current node.

// Retrieve and remove the first node from the queue.

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

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

// Initialize the queue with the root of the tree.

// Number of nodes at the current level.

let previousNode: Node | null = null;

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

if (previousNode !== null) {

if (currentNode.right) {

// Loop through each node on the current level.

previousNode.next = currentNode;

queue.push(currentNode.right);

// After connecting all levels, return the root node of the tree.

const currentNode = queue.shift()!;

const levelSize = queue.length;

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

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

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

```
45
               // The current node will be the previous one when the for loop processes the next node.
               previousNode = currentNode;
46
47
               // If the current node has a left child, add it to the queue.
48
49
               if (currentNode.left) {
50
                    queue.push(currentNode.left);
```

return root;

Time and Space Complexity

right: Node | null;

if (root === null) {

return root;

const queue: Node[] = [root];

while (queue.length > 0) {

next: Node | null;

6

8

9

10

11

12

13

15

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

51

52

53

54

55

56

57

58

59

60

61

62

63

65

64 }

14 }

Time Complexity:

node exactly once. During each visit, it only performs constant time operations such as setting the next pointer.

The time complexity of the code is O(N) where N is the number of nodes in the binary tree. This is because the algorithm visits each

Each level of the tree is examined using a while loop which iterates through the nodes that are horizontally connected via the next

pointer. For each node, it calls the modify function twice—once for the left child and once for the right child. However, these calls do

The given code is designed to connect each node to its next right node in a binary tree, making use of level order traversal.

// After the level is processed, the last node should point to null, which is already the default.

not result in recursive function calls that would amplify the runtime since they only alter pointers if the children are not null.

Space Complexity: The space complexity of the code is 0(1) assuming that function call stack space isn't considered for the space complexity (as is common for such analysis). This is because the algorithm uses only a constant amount of extra space: a few pointers (curr, prev, next) to keep track of the current node and to link the next level nodes. It does not allocate any additional data structures that grow

with the size of the input tree. However, if we do consider recursive call stack then the space complexity would be O(H) where H is the height of the tree. This accounts for the call stack during the execution of the function when called recursively for each level of the tree. For a balanced binary tree, this would be O(log N), and for a completely unbalanced tree, it could be O(N) in the worst case. The provided code does not seem to use recursion, so the space complexity remains 0(1).