<https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/>

Given a binary tree

struct Node {

int val;

Node \*left;

Node \*right;

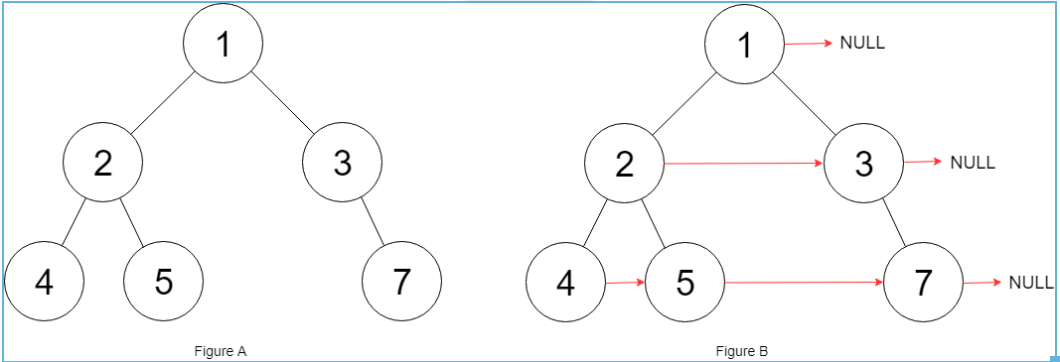
Node \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**Example 1:**



Input: root = [1,2,3,4,5,null,7]

Output: [1,#,2,3,#,4,5,7,#]

Explanation: Given the above binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

**Example 2:**

Input: root = []

Output: []

**Constraints:**

* The number of nodes in the tree is in the range [0, 6000].
* -100 <= Node.val <= 100

**Follow-up:**

* You may only use constant extra space.
* The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

**Attempt 1: 2023-08-19**

**Solution 1: BFS (10min)**

/\*

// Definition for a Node.

class Node {

public int val;

public Node left;

public Node right;

public Node next;

public Node() {}

public Node(int \_val) {

val = \_val;

}

public Node(int \_val, Node \_left, Node \_right, Node \_next) {

val = \_val;

left = \_left;

right = \_right;

next = \_next;

}

};

\*/

class Solution {

public Node connect(Node root) {

if(root == null) {

return null;

}

Queue<Node> q = new LinkedList<Node>();

q.offer(root);

while(!q.isEmpty()) {

int size = q.size();

Node prev = null;

for(int i = 0; i < size; i++) {

Node cur = q.poll();

if(i > 0) {

prev.next = cur;

}

prev = cur;

// The only difference than L116, check both

// left and right child if exist

if(cur.left != null) {

q.offer(cur.left);

}

if(cur.right != null) {

q.offer(cur.right);

}

}

}

return root;

}

}

**Refer to**

<https://leetcode.wang/leetcode-117-Populating-Next-Right-Pointers-in-Each-NodeII.html>

给定一个二叉树，然后每个节点有一个 next 指针，将它指向它右边的节点。和 [116 题](https://leetcode.wang/leetcode-116-Populating-Next-Right-Pointers-in-Each-Node.html) 基本一样，区别在于之前是满二叉树。

# **解法一 BFS**

直接把 [116 题](https://leetcode.wang/leetcode-116-Populating-Next-Right-Pointers-in-Each-Node.html) 题的代码复制过来就好，一句也不用改。

利用一个栈将下一层的节点保存。通过pre指针把栈里的元素一个一个接起来。

public Node connect(Node root) {

if (root == null) {

return root;

}

Queue<Node> queue = new LinkedList<Node>();

queue.offer(root);

while (!queue.isEmpty()) {

int size = queue.size();

Node pre = null;

for (int i = 0; i < size; i++) {

Node cur = queue.poll();

if (i > 0) {

pre.next = cur;

}

pre = cur;

if (cur.left != null) {

queue.offer(cur.left);

}

if (cur.right != null) {

queue.offer(cur.right);

}

}

}

return root;

}

**Solution 2: DFS (120min, must traverse on right subtree first)**

/\*

// Definition for a Node.

class Node {

public int val;

public Node left;

public Node right;

public Node next;

public Node() {}

public Node(int \_val) {

val = \_val;

}

public Node(int \_val, Node \_left, Node \_right, Node \_next) {

val = \_val;

left = \_left;

right = \_right;

next = \_next;

}

};

\*/

class Solution {

public Node connect(Node root) {

return helper(root);

}

private Node helper(Node node) {

if(node == null) {

return null;

}

if(node.left != null) {

if(node.right != null) {

node.left.next = node.right;

} else {

node.left.next = findNext(node.next);

}

}

if(node.right != null) {

node.right.next = findNext(node.next);

}

// Refer to

// https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/solutions/37979/o-1-concise-fast-what-s-so-hard/comments/291663

// Q: In the recursive call what's the reason behind traversing

// right node first and then the left node. If i do it the other

// way around, it is wrong

// e.g

// Input

// [2,1,3,0,7,9,1,2,null,1,0,null,null,8,8,null,null,null,null,7]

/\*\*

2 ---> null [level 1]

/ \

1 ---> 3 ---> null [level 2]

/ \ / \

0 ---> 7 ---> 9 ---> 1 ---> null [level 3]

/ / \ / \

2 ---> 1 -> 0 ---> 8 -> 8 --> null [level 4]

/

7 ---> null [level 5]

\*/

// Output

// [2,#,1,3,#,0,7,9,1,#,2,1,0,#,7,#]

// Expected

// [2,#,1,3,#,0,7,9,1,#,2,1,0,8,8,#,7,#]

// A: findNext() looks for a valid node that can be linked to the

// next of current child. We repeat for findNext(root->next).

// So next of any node needs to be solved before we process that

// node. This will only happen if we solve right before left.

// From the above example, when try to find level 4's 0's next,

// when hit logic "node.right.next = findNext(node.next)", the

// 'node' is level 3's 7, 'node.next' is level 3's 9, go through

// findNext(node) logic now, 9's left & right child both null, then

// hit findNext(node.next) logic, but instead of correctly getting

// next node of level 3's 9 as 1, and continue findNext(1) to correctly

// get left child level 4's 8, here, the 'node.next' is null only,

// in another word, next node of level 3's 9 not existing yet.

// To resolve this, must recursive on right before left

helper(node.right);

helper(node.left);

//helper(node.right);

return node;

}

private Node findNext(Node node) {

if(node == null) {

return null;

}

if(node.left != null) {

return node.left;

}

if(node.right != null) {

return node.right;

}

return findNext(node.next);

}

}

**Test by below example**

import java.util.\*;

public class TreeSolution {

private class TreeNode {

public int val;

public TreeNode left, right;

public TreeNode(int val) {

this.val = val;

this.left = this.right = null;

}

}

private class Node {

public int val;

public Node left, right, next;

public Node(int val) {

this.val = val;

this.left = this.right = this.next = null;

}

}

public Node connect(Node root) {

return helper(root);

}

private Node helper(Node node) {

if(node == null) {

return null;

}

if(node.left != null) {

if(node.right != null) {

node.left.next = node.right;

} else {

node.left.next = findNext(node.next);

}

}

if(node.right != null) {

node.right.next = findNext(node.next);

}

// Refer to

// https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/solutions/37979/o-1-concise-fast-what-s-so-hard/comments/291663

// Q: In the recursive call what's the reason behind traversing

// right node first and then the left node. If i do it the other

// way around, it is wrong

// e.g

// Input

// [2,1,3,0,7,9,1,2,null,1,0,null,null,8,8,null,null,null,null,7]

/\*\*

2 ---> null [level 1]

/ \

1 ---> 3 ---> null [level 2]

/ \ / \

0 ---> 7 ---> 9 ---> 1 ---> null [level 3]

/ / \ / \

2 ---> 1 -> 0 ---> 8 -> 8 --> null [level 4]

/

7 ---> null [level 5]

\*/

// Output

// [2,#,1,3,#,0,7,9,1,#,2,1,0,#,7,#]

// Expected

// [2,#,1,3,#,0,7,9,1,#,2,1,0,8,8,#,7,#]

// A: findNext() looks for a valid node that can be linked to the

// next of current child. We repeat for findNext(root->next).

// So next of any node needs to be solved before we process that

// node. This will only happen if we solve right before left.

// From the above example, when try to find level 4's 0's next,

// when hit logic "node.right.next = findNext(node.next)", the

// 'node' is level 3's 7, 'node.next' is level 3's 9, go through

// findNext(node) logic now, 9's left & right child both null, then

// hit findNext(node.next) logic, but instead of correctly getting

// next node of level 3's 9 as 1, and continue findNext(1) to correctly

// get left child level 4's 8, here, the 'node.next' is null only,

// in another word, next node of level 3's 9 not existing yet.

// To resolve this, must recursive on right before left

helper(node.right);

helper(node.left);

//helper(node.right);

return node;

}

private Node findNext(Node node) {

if(node == null) {

return null;

}

if(node.left != null) {

return node.left;

}

if(node.right != null) {

return node.right;

}

return findNext(node.next);

}

public static void main(String[] args) {

TreeSolution s = new TreeSolution();

Node zero\_1 = s.new Node(0);

Node zero\_2 = s.new Node(0);

Node one\_1 = s.new Node(1);

Node one\_2 = s.new Node(1);

Node one\_3 = s.new Node(1);

Node two\_1 = s.new Node(2);

Node two\_2 = s.new Node(2);

Node three = s.new Node(3);

Node seven\_1 = s.new Node(7);

Node seven\_2 = s.new Node(7);

Node eight\_1 = s.new Node(8);

Node eight\_2 = s.new Node(8);

Node nine = s.new Node(9);

two\_1.left = one\_1;

two\_1.right = three;

one\_1.left = zero\_1;

one\_1.right = seven\_1;

three.left = nine;

three.right = one\_2;

zero\_1.left = two\_2;

seven\_1.left = one\_3;

seven\_1.right = zero\_2;

one\_2.left = eight\_1;

one\_2.right = eight\_2;

zero\_2.left = seven\_2;

Node result = s.connect(two\_1);

System.out.println(result);

}

}

## **O(1) Space O(n) Time Recursive Solution**

**Refer to**

<https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/solutions/37979/o-1-concise-fast-what-s-so-hard/comments/163808>

Q: In the recursive call what's the reason behind traversing right node first and then the left node. If i do it the other way around, i get a Wrong Answer

A: findNext looks for a valid node that can be linked to the next of current child.

We repeat for findNext(root->next). So next of any node needs to be solved before we process that node. This will only happen if we solve right before left.

void connect(TreeLinkNode \*root)

{

if(root == NULL)

return;

if(root->left != NULL)

root->left->next = root->right!=NULL ? root->right : findNext(root->next);

if(root->right != NULL)

root->right->next = findNext(root->next);

connect(root->right);

connect(root->left);

}

TreeLinkNode\* findNext(TreeLinkNode\* curr)

{

if(!curr) return NULL;

if(curr->left!=NULL) return curr->left;

if(curr->right!=NULL) return curr->right;

return findNext(curr->next);

}

**Refer to**

<https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/solutions/172861/mostly-recursive-solution-o-n-time-beats-99-32-and-o-1-space-without-considering-stack/comments/260137>

Lets first be clear on what findNext means. findNext is essentially "recursively find the next left most child". Since we need to find "next" left most, we should call right first to make sure the nexts(on the right side) are already well connected, otherwise there would be gap in calling fnext. For example this is what would happen for this input if we call left first:

2

/ \

1-> 3

/ \ / \

0->7->9 1

/ / \ / \

2-> 1->"0"8 8

When the recursion comes to the "0" in the lowest level, the next step is to connect 0 with 8. However since 9 and 1 on the upper level haven't been connected yet (because we execute left part before we do right), findNext is not able to reach the left most child of 1, due to the gap between 9 and 1.

**Solution 3: BFS - Space-Optimized (120min)**

/\*

// Definition for a Node.

class Node {

public int val;

public Node left;

public Node right;

public Node next;

public Node() {}

public Node(int \_val) {

val = \_val;

}

public Node(int \_val, Node \_left, Node \_right, Node \_next) {

val = \_val;

left = \_left;

right = \_right;

next = \_next;

}

};

\*/

// 本题完全无法使用L116中的技术, 因为L117中无论root.left还是root.right都可能是null,

// 以下的逻辑基础不存在, 比如cur.left可以等于null, 逻辑甚至不会运行

/\*\*

// 提前把下一层的next构造完成，到了下一层的时候就可以遍历了

if(cur.left != null) {

cur.left.next = cur.right;

if(cur.next != null) {

cur.right.next = cur.next.left;

}

} else {

break;

}

\*/

class Solution {

public Node connect(Node root) {

Node cur = root;

while(cur != null) {

// dummy用于定位下一层的开头, dummy的更新来自tail的移动

Node dummy = new Node(-1);

// tail用于在cur遍历当前层的同时建立下一层

Node tail = dummy;

// 遍历cur的当前层

while(cur != null) {

if(cur.left != null) {

// 由于之前设定了tail = dummy，相当于通过对tail.next

// 赋值为cur.left也同时让dummy移动到了cur的下一行,

// 用cur遍历当前层的同时也通过tail = tail.next构建了

// 下一层

tail.next = cur.left;

tail = tail.next;

}

if(cur.right != null) {

// 由于之前设定了tail = dummy，相当于通过对tail.next

// 赋值为cur.right也同时让dummy移动到了cur的下一行,

// 用cur遍历当前层的同时也通过tail = tail.next构建了

// 下一层

tail.next = cur.right;

tail = tail.next;

}

// 用cur = cur.next遍历当前层

cur = cur.next;

}

// 更新cur到下一层

cur = dummy.next;

}

return root;

}

}

**Refer to**

<https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/solutions/37979/o-1-concise-fast-what-s-so-hard/>

This is definitely medium, not hard. Took me about 5 minutes, and some medium problems took me a few hours! Since you have to walk the tree in BFS order and you're given extra references to help you do just that, it's intuitive, it's simple and it's fast. The first level (root) is connected already, so you connect the next level and then you just walk through the linked list you've just created and so on.

public void connect(TreeLinkNode root) {

for (TreeLinkNode head = root; head != null; ) {

TreeLinkNode nextHead = new TreeLinkNode(0), nextTail = nextHead;

for (TreeLinkNode node = head; node != null; node = node.next) {

if (node.left != null) {

nextTail.next = node.left;

nextTail = node.left;

}

if (node.right != null) {

nextTail.next = node.right;

nextTail = node.right;

}

}

head = nextHead.next;

}

}

P. S. As it was mentioned several times in the comments, I decided to show the improved version as well. There is no need to allocate the sentinel node every time:

public void connect(TreeLinkNode root) {

for (TreeLinkNode head = root, nextHead = new TreeLinkNode(0), nextTail; head != null; ) {

(nextTail = nextHead).next = null;

for (; head != null; head = head.next) {

if (head.left != null) {

nextTail.next = head.left;

nextTail = head.left;

}

if (head.right != null) {

nextTail.next = head.right;

nextTail = head.right;

}

}

head = nextHead.next;

}

}

It doesn't show much in the submission, but I put it to a simple test: create a 10 million node tall tree with only one node per level, then connect it the previous way and the improved way. Initial tree creation took about 550–570 MB of RAM on my PC (running under Java 1.8.0\_172 x64). Then, the first version takes about 100 MB more and runs for about 2 seconds. The improved version runs for about 0.1 seconds and takes no additional RAM (no big surprise here). Looks I overestimated GC and memory allocation optimization. I was almost sure it would just reuse the same memory again and again.

**Refer to**

<https://leetcode.wang/leetcode-117-Populating-Next-Right-Pointers-in-Each-NodeII.html>

参考 [这里](https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/discuss/37979/O(1).-Concise.-Fast.-What's-so-hard>)。

利用解法一的思想，我们利用 pre 指针，然后一个一个取节点，把它连起来。解法一为什么没有像解法二那样考虑当前节点为 null 呢？因为我们没有添加为 null 的节点，就是下边的代码的作用。

if (cur.left != null) {

queue.offer(cur.left);

}

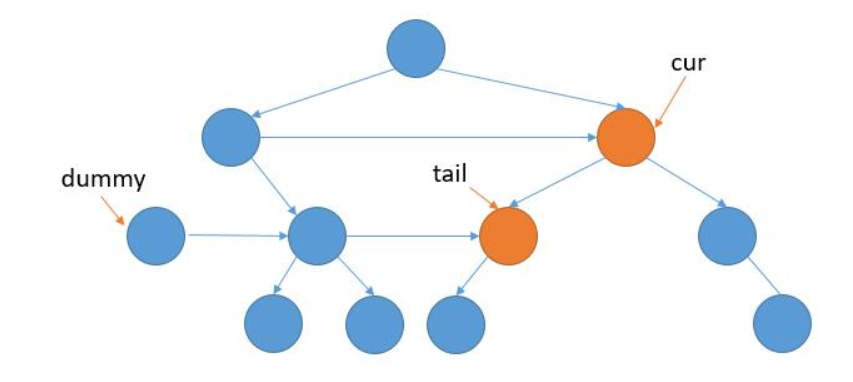
if (cur.right != null) {

queue.offer(cur.right);

}

所以这里是一样的，如果当前节点为null不处理就可以了。

第二个问题，怎么得到每次的开头的节点呢？我们用一个dummy指针，当连接第一个节点的时候，就将dummy指针指向他。此外，之前用的pre指针，把它当成tail指针可能会更好理解。如下图所示：



cur 指针利用 next 不停的遍历当前层。

如果 cur 的孩子不为 null 就将它接到 tail 后边，然后更新tail。

当 cur 为 null 的时候，再利用 dummy 指针得到新的一层的开始节点。

dummy 指针在链表中经常用到，他只是为了处理头结点的情况，它并不属于当前链表。

代码就异常的简单了。

Node connect(Node root) {

Node cur = root;

while (cur != null) {

Node dummy = new Node();

Node tail = dummy;

//遍历 cur 的当前层

while (cur != null) {

if (cur.left != null) {

tail.next = cur.left;

tail = tail.next;

}

if (cur.right != null) {

tail.next = cur.right;

tail = tail.next;

}

cur = cur.next;

}

//更新 cur 到下一层

cur = dummy.next;

}

return root;

}

# **总**

本来为了图方便，在 [116 题](https://leetcode.wang/leetcode-116-Populating-Next-Right-Pointers-in-Each-Node.html) 的基础上把解法二改了出来，还搞了蛮久，因为为 null 的情况太多了，不停的报空指针异常，最后终于理清了思路。但和解法三比起来实在是相形见绌了，解法三太优雅了，但其实这才是正常的思路，从解法一的做法产生灵感，利用 tail 指针将它们连起来。