LeetCode 117

https://leetcode.com/problems/populating-next-right-pointers-in-each-node/description/

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

Yifeng Zeng

Description

116. Populating Next Right Pointers in Each Node

117. Populating Next Right Pointers in Each Node II

Given a binary tree

```
struct TreeLinkNode {
  TreeLinkNode *left;
  TreeLinkNode *right;
  TreeLinkNode *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.

Note:

You may only use constant extra space.

You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,
Given the following perfect binary tree,

```
         1
       /  \
      2    3
     / \  / \
    4  5  6  7
```

After calling your function, the tree should look like:

```
         1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \  / \
    4->5->6->7 -> NULL
```

Idea Report

My primitive idea is just do a level order traversal of the binary tree and link all the nodes from left to rigth in the same level. We can use a queue to do the level order traversal, but this takes O(n) extra space, so we need to optimize it later.

Code:

```
class Solution {
  public void connect(TreeLinkNode root) {
      if (root == null) {
          return;
      }

      Deque<TreeLinkNode> q = new LinkedList<>();
      q.offer(root);

      while (!q.isEmpty()) {
          int size = q.size();
          for (int i = 0; i < size; i++) {
              TreeLinkNode cur = q.poll();
              if (i < size - 1) {
                  cur.next = q.peek();
              }
              if (cur.left != null) {
                  q.offer(cur.left);
```

```
                }
                if (cur.right != null) {
                    q.offer(cur.right);
                }
            }
        }
    }
}
```

We are able to assume that it is a perfect binary tree, so for any node, if it has a left child, it must have a right child, so we can just connect then directly using root.left.next = root.right. And if a node has next node, then we can connect its right child to its next node's left child using root.right.next = root.next.left. We can connect the children first and then since children are connected, so we can traverse children to connect childern's children, in this way, we can do the level order traversal without using extra space.

Code:

```
public class Solution {

    public void connect(TreeLinkNode root) {
        while (root != null) {
            TreeLinkNode left = root.left;
            while (root != null && root.left != null) {
                root.left.next = root.right;
                if (root.next != null) {
                    root.right.next = root.next.left;
                }
                root = root.next;
            }
            root = left;
        }
    }
}
```

Follow up LC117

Follow up for problem "Populating Next Right Pointers in Each Node".

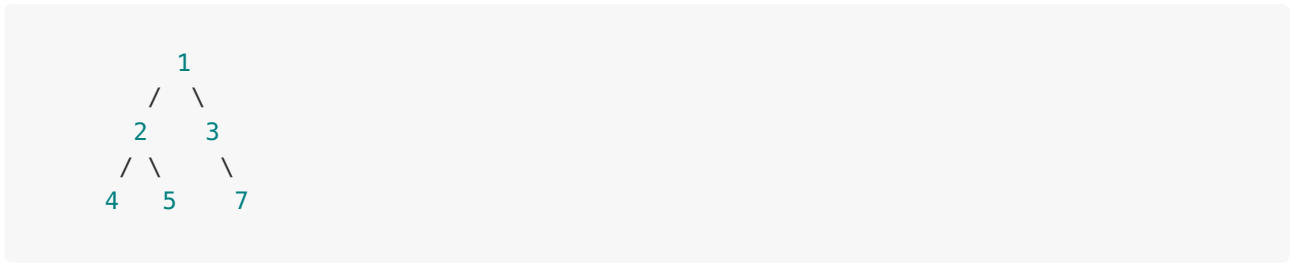
What if the given tree could be any binary tree? Would your previous solution still work?

Note:

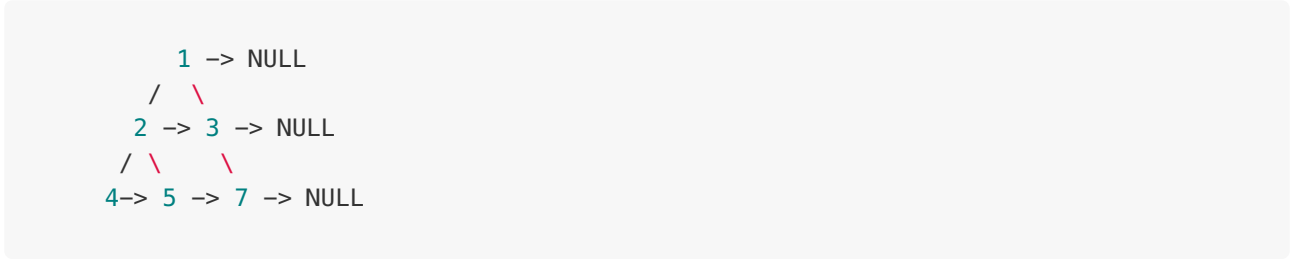
You may only use constant extra space.

For example,

Given the following binary tree,



After calling your function, the tree should look like:



Idea Report

The primitive idea of LC116 and the first piece of code can actually solve this follow up directly. So how can we optimize it and just use constant space? For the LC116, we assumed it is a perfect tree, so we can just use the left child of the first node as the begining of the next level. In this follow up, there may not be a left node of the first node at all, so we can just use a dummy node as a beginning of the next level, and connect the next level children together.

Code:

```
public class Solution {
    public void connect(TreeLinkNode root) {
        TreeLinkNode dummy = new TreeLinkNode(0);
        TreeLinkNode head = dummy;
        while (root != null) {
```

```
            while (root != null) {
                if (root.left != null) {
                    head.next = root.left;
                    head = head.next;
                }
                if (root.right != null) {
                    head.next = root.right;
                    head = head.next;
                }
                root = root.next;
            }
            root = dummy.next;
            dummy.next = null;
            head = dummy;
        }
    }
}
```

Summary

- Consider what to do at each node.
- Use dummy node to save temporary result.