Previous Next

116. Populating Next Right Pointers in Each Node

Dec. 1, 2019 | 31.6K views

Average Rating: 4.83 (47 votes)

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

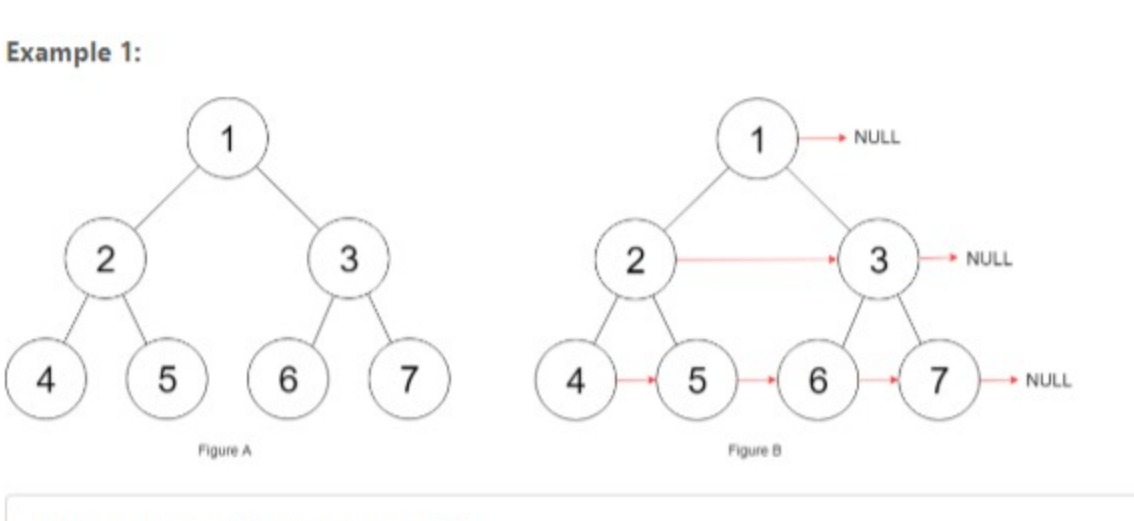
Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Follow up:

- You may only use constant extra space.
- Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

Example 1:




```
Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should popu
```

Constraints:

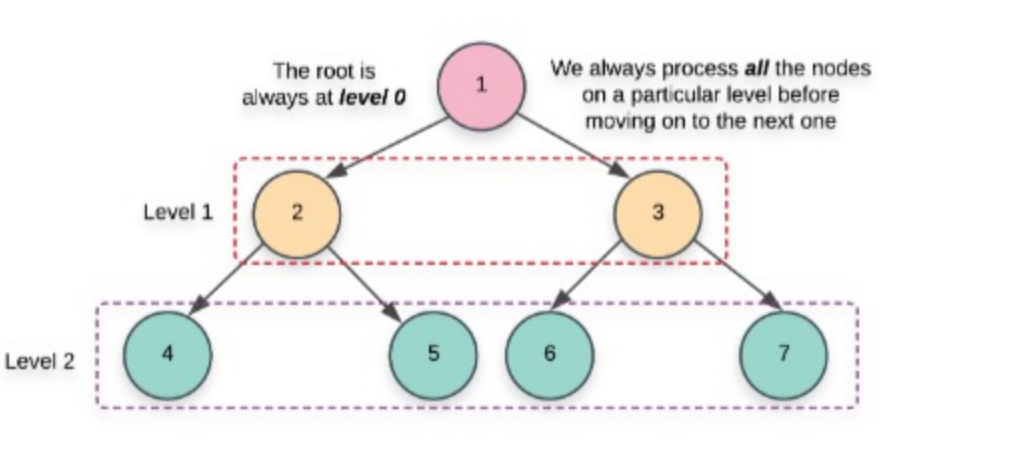
- The number of nodes in the given tree is less than `4096`.
- `-1000 <= node.val <= 1000`

Solution

Approach 1: Level Order Traversal

Intuition

There are two basic kinds of traversals on a tree or a graph. One is where we explore the tree in a depth first manner i.e. one branch at a time. The other one is where we traverse the tree breadth-wise i.e. we explore one level of the tree before moving on to the next one. For trees, we have further classifications of the depth first traversal approach called `preorder`, `inorder`, and the `postorder` traversals. Breadth first approach to exploring a tree is based on the concept of the `level` of a node. The `level` of a node is its depth or distance from the root node. We process all the nodes on one level before moving on to the next one.

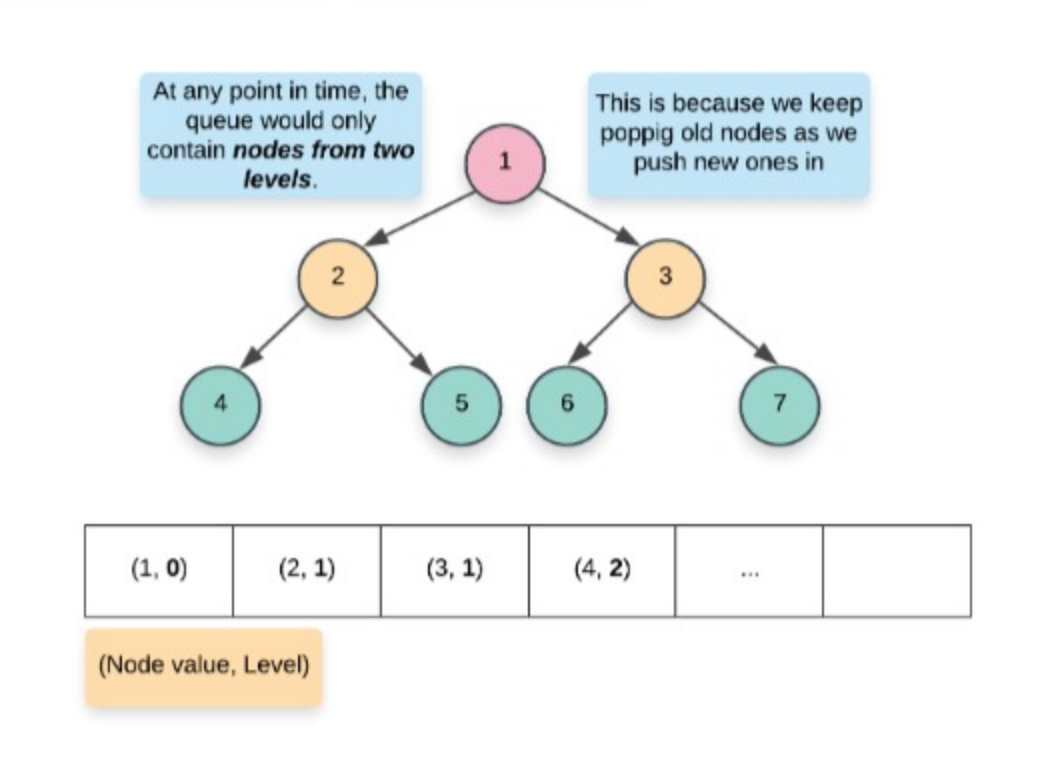


Now that we have the basics out of the way, it's pretty evident that the problem statement strongly hints at a breadth first kind of a solution. We need to link all the nodes together which lie on the `same level` and the level order or the breadth first traversal gives us access to all such nodes.

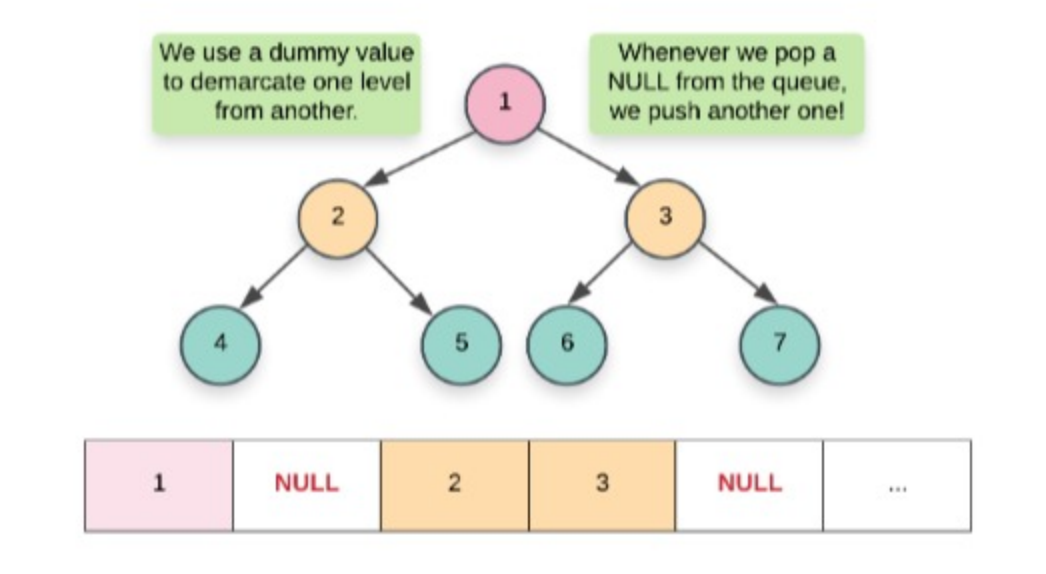
Algorithm

1. Initialize a queue, `Q` which we will be making use of during our traversal. There are multiple ways to implement the level order traversal especially when it comes to identifying the level of a particular node.

   1. We can add a pair of $(node, level)$ to the queue and whenever we add the children of a node, we add $(node.left,\ parent\_level + 1)$ and $(node.right,\ parent\_level + 1)$. This approach wouldn't be very efficient for our algorithm since we need *all* the nodes on the same level and we would need another data structure just for that.



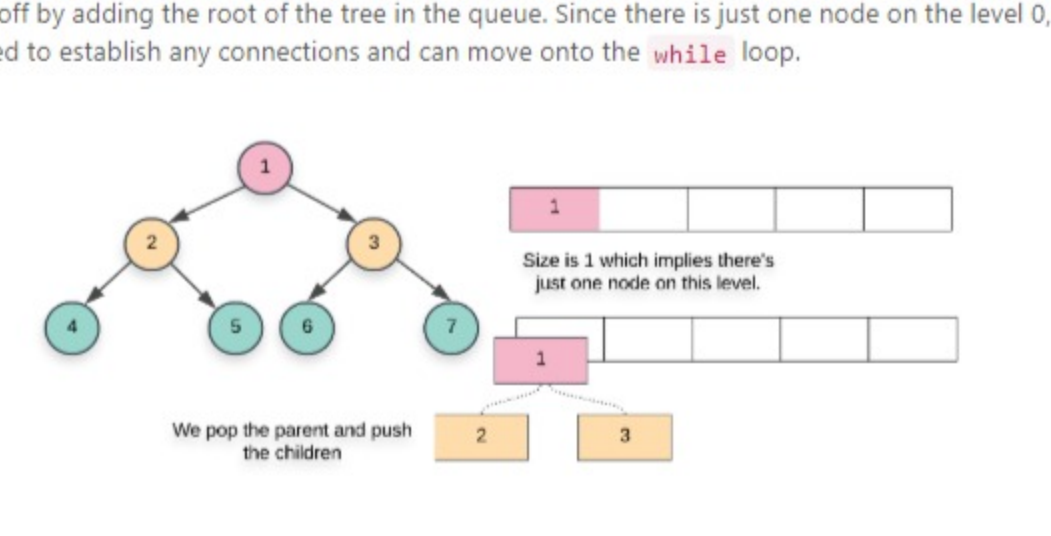
   2. A more memory efficient way of segregating the same level nodes is to use some demarcation between the levels. Usually, we insert a `NULL` entry in the queue which marks the end of the previous level and the start of the next level. This is a great approach but again, it would still consume some memory proportional to the number of levels in the tree.



   3. The approach we will be using here would have a nested loop structure to get around the requirement of a `NULL` pointer. Essentially, at each step, we record the size of the queue and that always corresponds to ***all*** the nodes on a particular level. Once we have this size, we only process these many elements and no more. By the time we are done processing `size` number of elements, the queue would contain ***all*** the nodes on the next level. Here's a pseudocode for the same:

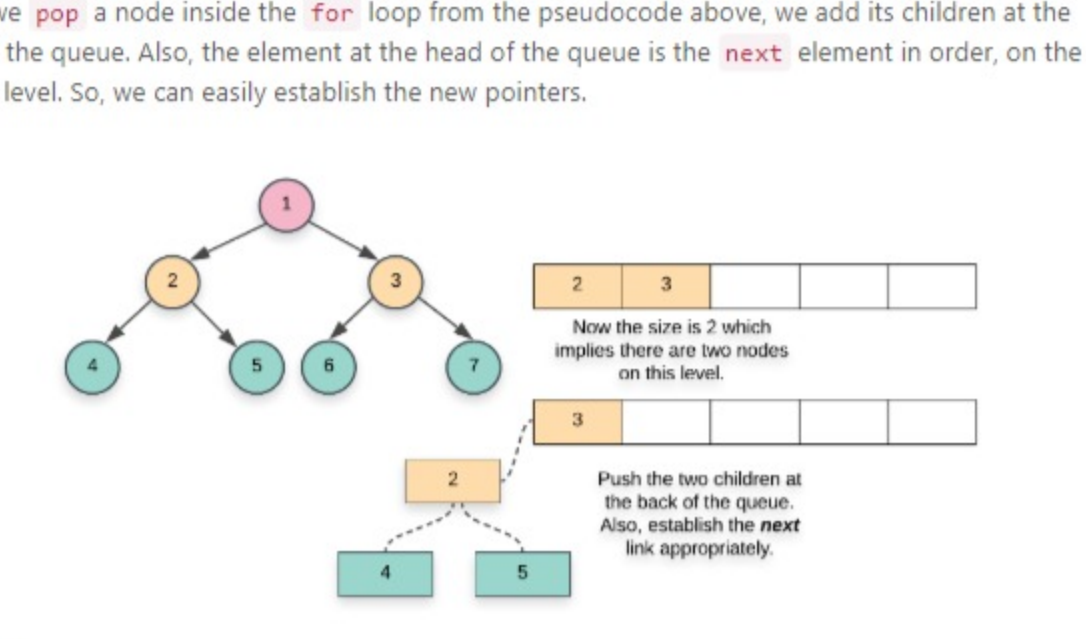
```
while (!Q.empty())
{
    size = Q.size()
    for i in range 0..size
    {
        node = Q.pop()
        Q.push(node.left)
        Q.push(node.right)
    }
}
```

2. We start off by adding the root of the tree in the queue. Since there is just one node on the level 0, we don't need to establish any connections and can move onto the `while` loop.



3. The first `while` loop from the pseudocode above essentially iterates over each level one by one and the inner for loop iterates over all the nodes on the particular level. Since we have access to all the nodes on the same level, we can establish the next pointers easily.

4. When we `pop` a node inside the `for` loop from the pseudocode above, we add its children at the back of the queue. Also, the element at the head of the queue is the `next` element in order, on the current level. So, we can easily establish the new pointers.



Java Python Copy

```
import collections

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

        if not root:
            return root

        # Initialize a queue data structure which contains
        # just the root of the tree
        Q = collections.deque([root])

        # Outer while loop which iterates over
        # each level
        while Q:

            # Note the size of the queue
            size = len(Q)

            # Iterate over all the nodes on the current level
            for i in range(size):

                # Pop a node from the front of the queue
                node = Q.popleft()

                # This check is important. We don't want to
                # establish any wrong connections. The queue will
```

Complexity Analysis

- Time Complexity: $O(N)$ since we process each node exactly once. Note that processing a node in this context means popping the node from the queue and then establishing the next pointers.
- Space Complexity: $O(N)$. This is a perfect binary tree which means the last level contains $N/2$ nodes. The space complexity for breadth first traversal is the space occupied by the queue which is dependent upon the maximum number of nodes in particular level. So, in this case, the space complexity would be $O(N)$.

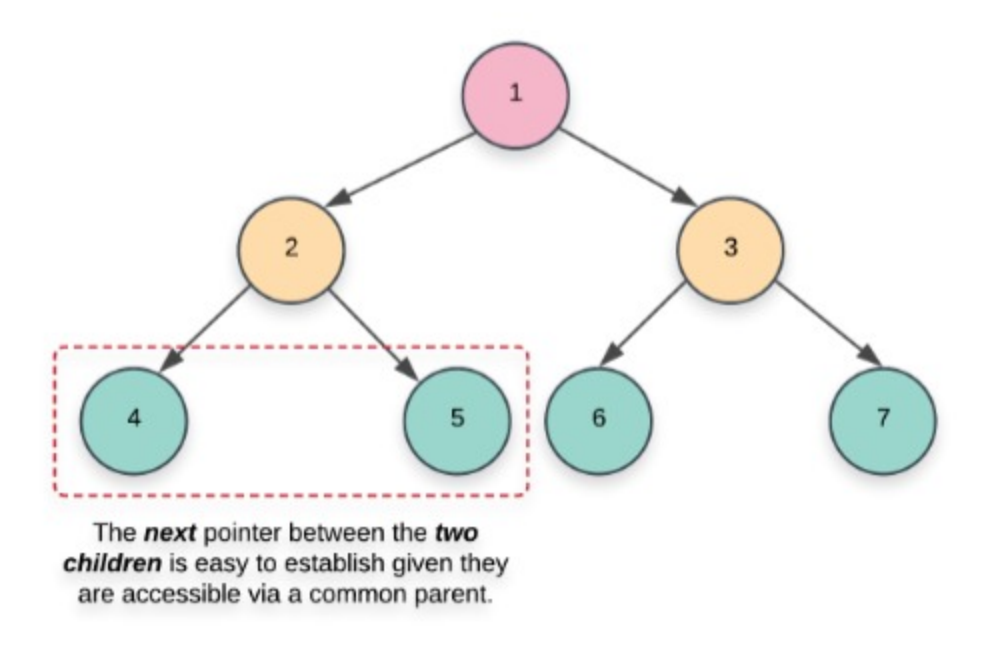
Approach 2: Using previously established next pointers

Intuition

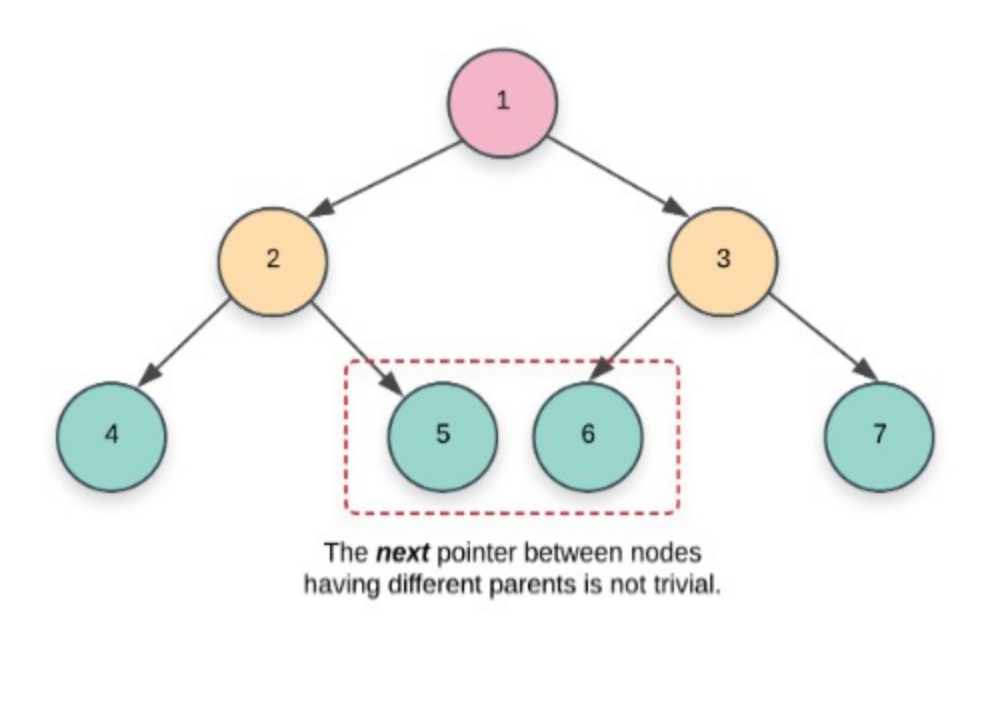
Let's look at the two types of `next` pointer connections we need to establish for a given tree.

1. This first case is the one where we establish the next pointers between the two children of a given node. This is the easier of the two cases since both the children are accessible via the same node. We can simply do the following to establish this connection.

```
node.left.next = node.right
```



2. This next case is not too straightforward to handle. In addition to establishing the next pointers between the nodes having a common parent, we also need to set-up the correct pointers between nodes which have a different parent.



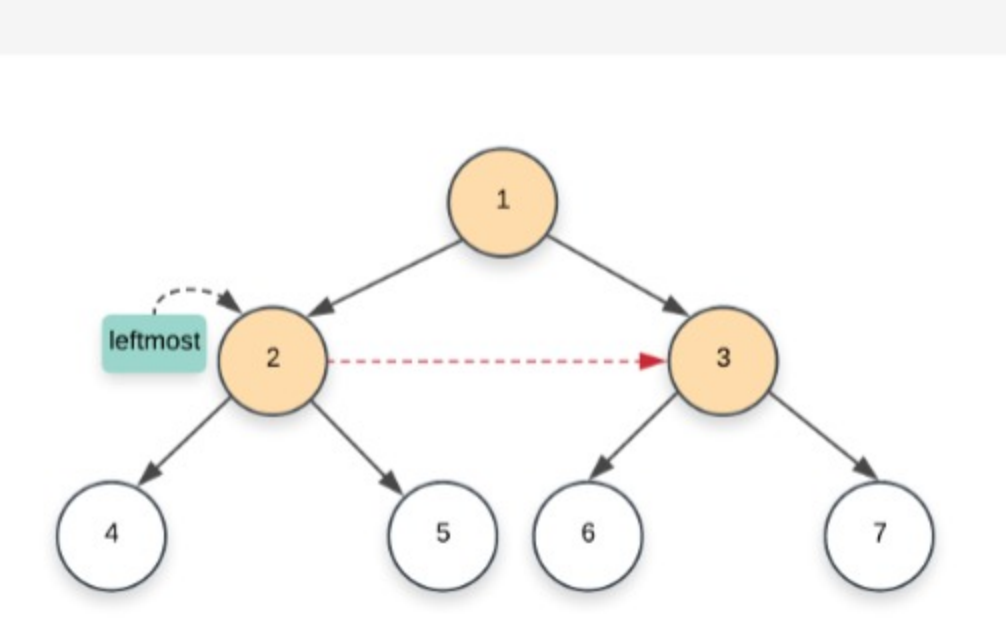
If we simply had the parent pointers available with each node, this problem would have been trivial to solve. However, we don't have any such pointers available. The basic idea for this approach is based on the fact that:

> We only move on to the level N+1 when we are done establishing the next pointers for the level N. Since we have access to all the nodes on a particular level via the next pointers, we can use these next pointers to establish the connections for the next level or the level containing their children.

Algorithm

1. We start at the root node. Since there are no more nodes to process on the first level or level `0`, we can establish the next pointers on the next level i.e. level 1. An important thing to remember in this algorithm is that we establish the next pointers for a level N while we are still on level $N-1$ and once we are done establishing these new connections, we move on to N and do the same thing for $N+1$.
2. As we just said, when we go over the nodes of a particular level, their next pointers are already established. This is what helps get rid of the queue data structure from the previous approach and helps save space. To start on a particular level, we just need the `leftmost` node. From there on out, its just a linked list traversal.
3. Based on these ideas, our algorithm will have the following pseudocode:

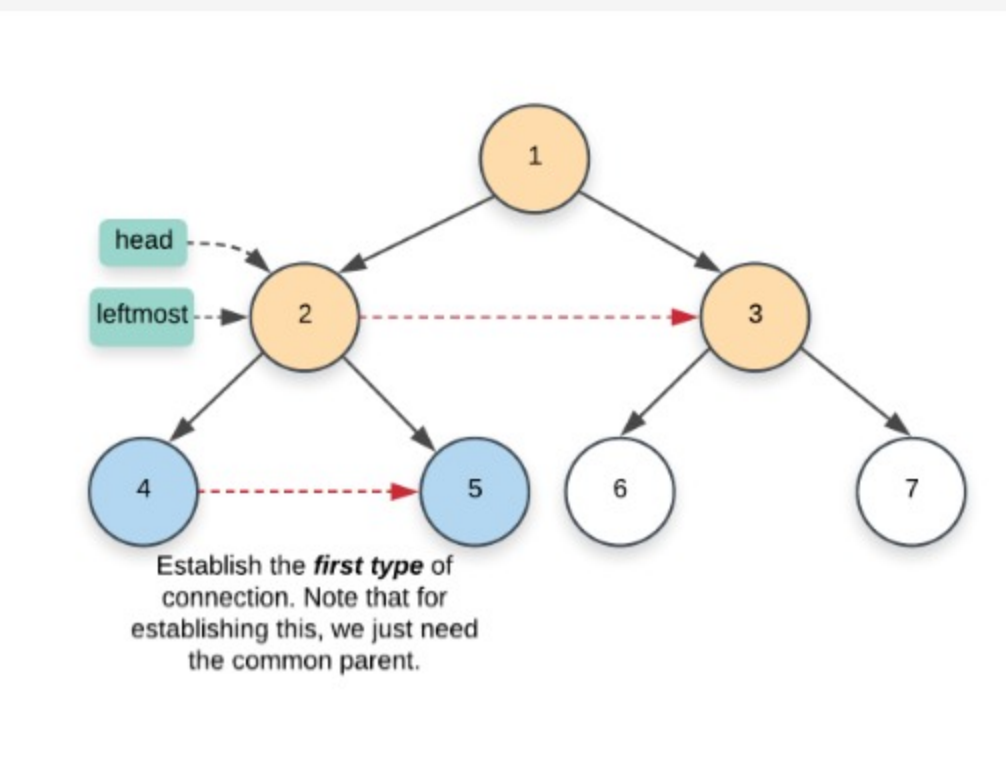
```
leftmost = root
while (leftmost.left != null)
{
    head = leftmost
    while (head.next != null)
    {
        1) Establish Connection 1
        2) Establish Connection 2 using next pointers
        head = head.next
    }
    leftmost = leftmost.left
}
```



4. The `Connection 1` and `Connection 2` mentioned above correspond to the two kinds of connections we looked at earlier on in the intuition section of this approach.

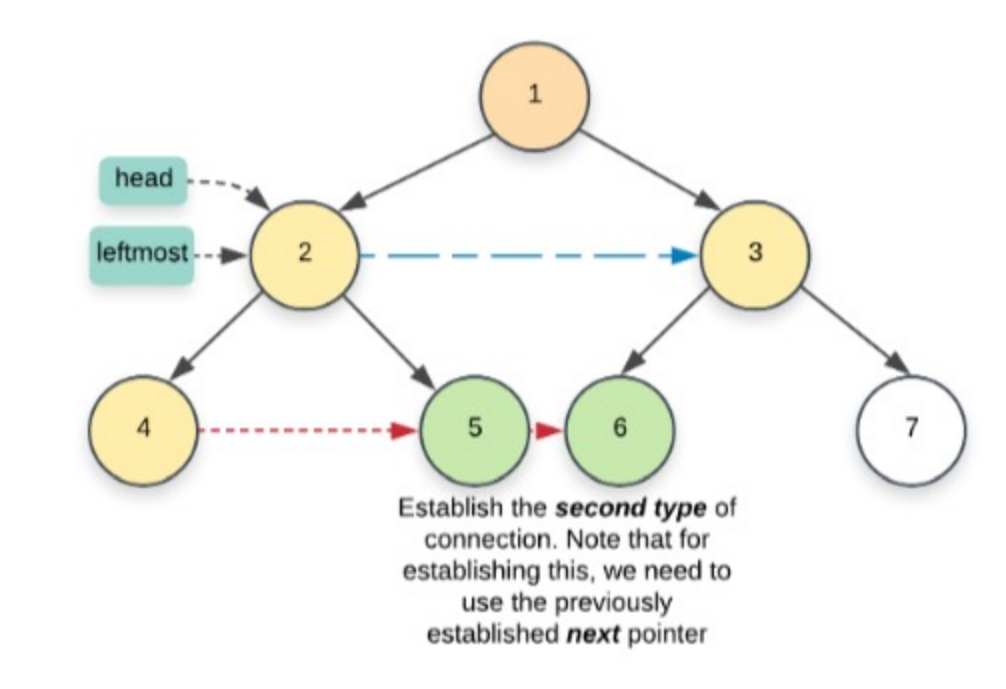
   1. The first one is fairly simple to establish given that it's between the two nodes having a common parent. So, we could simply do something like this to link the two children:

```
node.left.next = node.right
```

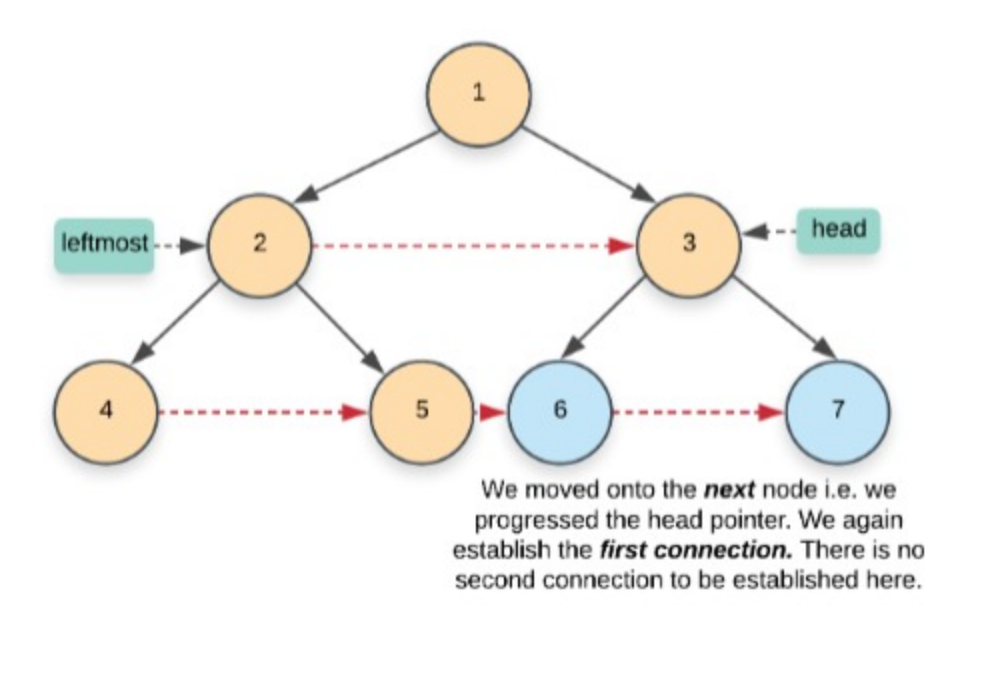


   2. For the second type of connection, we have to make use of the next pointers on the current level. Remember that this second type of connection is between nodes that have different parents. More specifically, it's the link between the right child of a node and the left child of the `next` node. Since we already have the next pointers set up on the current level, we use this to set up the correct pointers on the next level.

```
node.right.next = node.next.left
```



5. Once we are done with the current level, we move on to the next one. One last thing that's left here is to update the `leftmost` node. We need that node to start traversal on a particular level. Think of it as the head of the linked list. Since this is a `perfect binary tree`, the leftmost node will always be the left child of the current leftmost node. The only nodes which don't have any children are the ones on the final level and these would be the leaves of the tree.



Java Python Copy

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

        if not root:
            return root

        # Start with the root node. There are no next pointers
        # that need to be set up on the first level
        leftmost = root

        # Once we reach the final level, we are done
        while leftmost.left:

            # Iterate the "linked list" starting from the head
            # node and using the next pointers, establish the
            # corresponding links for the next level
            head = leftmost
            while head:

                # CONNECTION 1
                head.left.next = head.right

                # CONNECTION 2
                if head.next:
                    head.right.next = head.next.left

                # Progress along the list (nodes on the current level)
```

Complexity Analysis

- Time Complexity: $O(N)$ since we process each node exactly once.
- Space Complexity: $O(1)$ since we don't make use of any additional data structure for traversing nodes on a particular level like the previous approach does.

Rate this article:

Previous Next

Comments: 20 Sort By

Type comment here... (Markdown is supported)

Preview Post

theseungjin ★ 163 January 20, 2020 4:59 AM
Really great solution article. Thank you. I'm glad you covered the 1st solution despite there being a constraint for constant space use.
30 Share Reply

silentcoder9 ★ 59 December 29, 2019 8:37 AM
Recursive solution exists here

```
class Solution {
    public void connectNext(Node root){
        if(root == null){
```

Read More
7 Share Reply
SHOW 2 REPLIES

AlgorithmImplementer ★ 571 April 9, 2020 1:10 AM Report
@sachinmalhotra1993 You shall write more articles to gain back the trust on the quality of leetcode articles. Kudos to you for such a lucid and detailed explanation!
3 Share Reply

CodePilgrim ★ 6 January 25, 2020 5:49 PM
👏👏👏
3 Share Reply

lsheng_mel ★ 167 May 20, 2020 6:06 PM
Wow, the second approach is brilliant. Trying to adapt to a similar idea using recursion, but definitely not as efficient as this.
2 Share Reply

bonsaT ★ 1 June 30, 2020 8:45 AM
Simple and easy to understand JS solution

```
var connect = function(root) {
    if(!root) return root
```

Read More
1 Share Reply

sabirockster ★ 4 February 20, 2020 7:22 AM Report
Similar recursive solution

```
def connect(self, root: 'Node', n = None) -> 'Node':
        if not root: return
        root.next = n
```

Read More
1 Share Reply

ShahidKalam ★ 1 June 25, 2020 10:53 PM
An excellent article, try to read from the top even if you have done approach 1 it helps to build intuition for approach 2
0 Share Reply

rgbuv ★ 3 June 20, 2020 9:31 PM Report
And here is my iterative solution with O(1) space and O(N) compute.

```
class Solution {
    public Node connect(Node root) {
        if(root == null || root.left == null || root.right == null){
```

Read More
0 Share Reply

rgbuv ★ 3 June 20, 2020 8:40 PM
Algorithm #2 is elegant and simple. Since I implemented my solution before I saw this, in my algorithm I used Depth-First search. I traverse to right first and used the right most node at every level to store the next candidate for Node.next assignment at that level. Code below.

```
class Solution {
```

Read More
0 Share Reply

‹ 1 2 ›