Populate Next Right Pointers Tree

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.

Example 1:

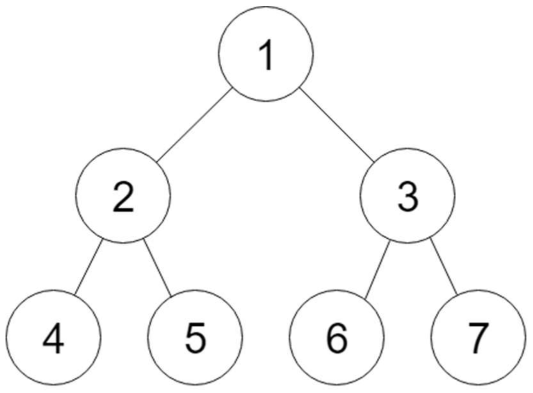


Figure A

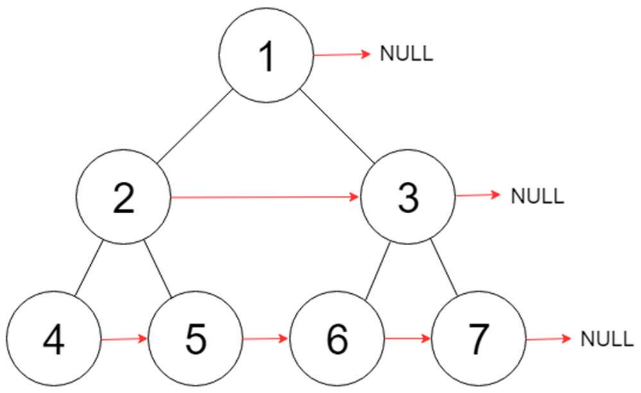


Figure B

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 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, 2^{12} - 1].
- -1000 <= Node.val <= 1000

Follow-up:

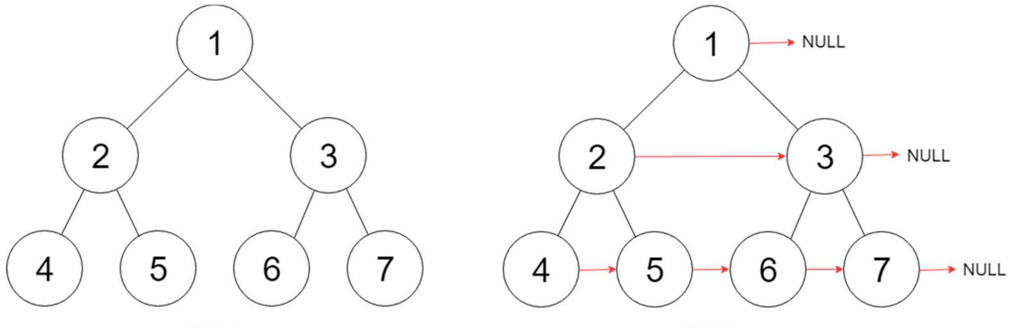
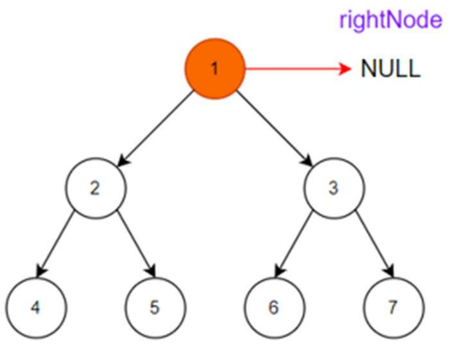
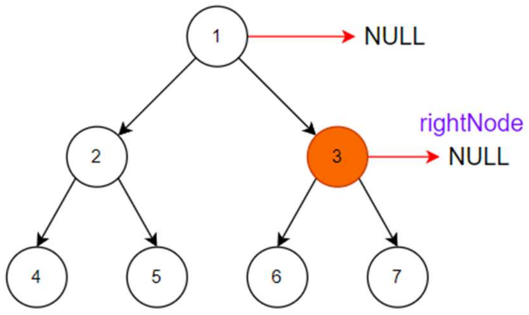
- You may only use constant extra space.
- The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.
- ***Solution - I (BFS - Right to Left)***
- It's important to see that the given tree is a **perfect binary tree**. This means that each node will always have both children and only the last level of nodes will have no children.
  - 

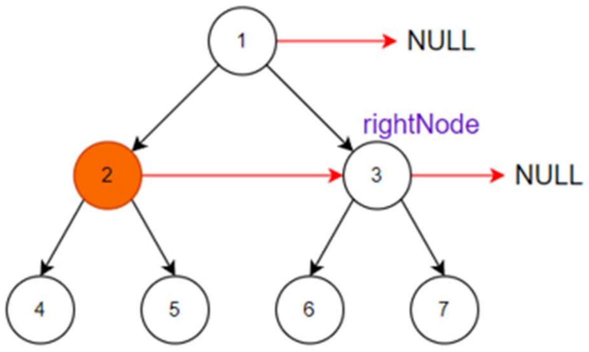
Figure A Figure B
- Now, we need to populate next pointers of each node with nodes that occur to its immediate right on the same level. This can easily be done with BFS. Since for each node, we require the right node on the same level, we will perform a **right-to-left BFS** instead of the standard left-to-right BFS.
- Before starting the traversal of each level, we would initialize a `rightNode` variable set to NULL. Then, since we are performing right-to-left BFS, we would be starting at rightmost node of each level. We set the next node of `cur` as `rightNode` and update `rightNode = cur`. This would ensure that each node would be assigned its `rightNode` properly while traversing from right to left. Also, if `cur` has a child, we would first push its right child and only then its left child (since we are doing right-to-left BFS). Once BFS is completed (after queue becomes empty), all next node would be populated and we can finally return `root`.
- The process is illustrated below -



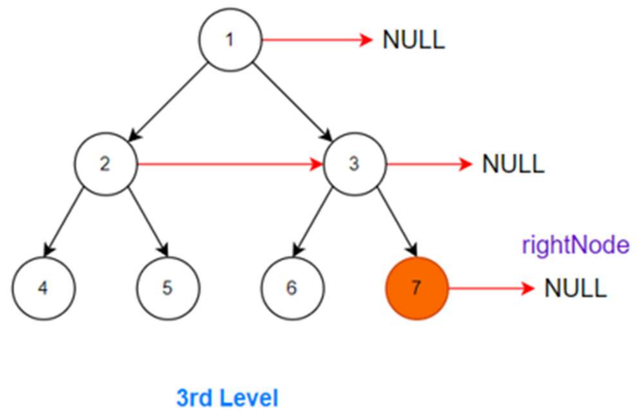
1st Level

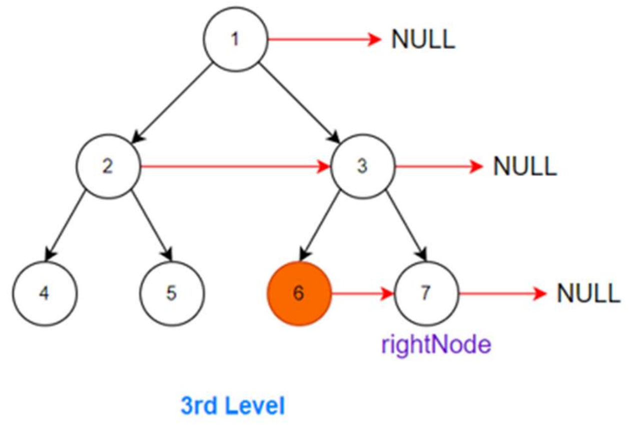


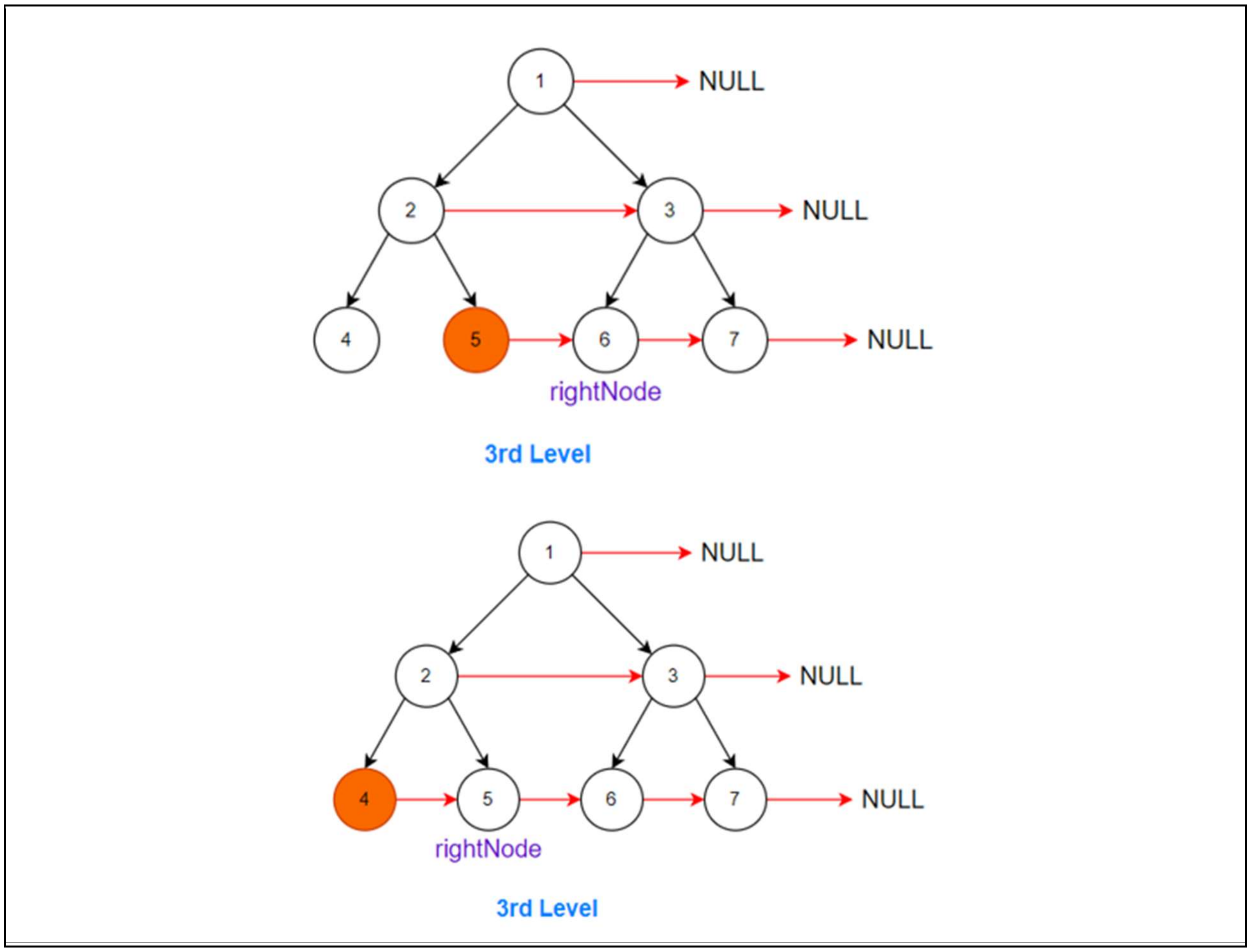
2nd Level



2nd Level


1
NULL
2
3
NULL
rightNode
4
5
6
7
NULL
3rd Level


1
NULL
2
3
NULL
4
5
6
7
NULL
rightNode
3rd Level



C++

```
class Solution {
public:
    Node* connect(Node* root) {
        if(!root) return nullptr;
        queue<Node*> q;
        q.push(root);
        while(size(q)) {
            Node* rightNode = nullptr;                    // set rightNode to null initially
            for(int i = size(q); i; i--) {                // traversing each level
                auto cur = q.front(); q.pop();            // pop a node from current level and,
                cur -> next = rightNode;                  // set its next pointer to rightNode
                rightNode = cur;                          // update rightNode as cur for next iteration
                if(cur -> right)                          // if a child exists
                    q.push(cur -> right),                 // IMP: push right first to do right-to-left BFS
                    q.push(cur -> left);                  // then push left
            }
        }
        return root;
    }
};
```

Solution - II (DFS)

We can also populate the next pointers recursively using DFS. This is slightly different logic than above but relies on the fact that the given tree is a perfect binary tree.

In the above solution, we had access to right nodes since we traversed in level-order. But in DFS, once we go to the next level, we cant get access to right node. So, we must update next pointers of the child of each node from the its parent's level itself. Thus at each recursive call -

- If child node exists:
  - assign next of left child node as right child node: `root -> left -> next = root -> right`. Note that, if once child exists, the other exists as well.
  - assign next of right child node as left child of root's next (if root's next exists): `root -> right -> next = root -> next -> left`.
    How? We need right immediate node of right child. This wont exist if current root's next node doesnt exists. If next node of current root is present (the next pointer of root would already be populated in above level) , the right immediate node of root's right child must be root's next's left child because if child of root exists, then the child of root's next must also exist.
- If child node doesn't exist, we have reached the last level, we can directly return since there's no child nodes to populate their next pointers

The process is very similar to the one illustrated in the image below with just the difference that we are traversing with DFS instead of BFS shown below.

```
class Solution {
public:
    Node* connect(Node* root) {
        if(!root) return nullptr;
        auto L = root -> left, R = root -> right, N = root -> next;
        if(L) {
            L -> next = R;                                // next of root's left is assigned as root's right
            if(N) R -> next = N -> left;                  // next of root's right is assigned as root's next's left (if root's
next exist)
            connect(L);                                   // recurse left  - simple DFS
            connect(R);                                   // recurse right
        }
        return root;
    }
};
```