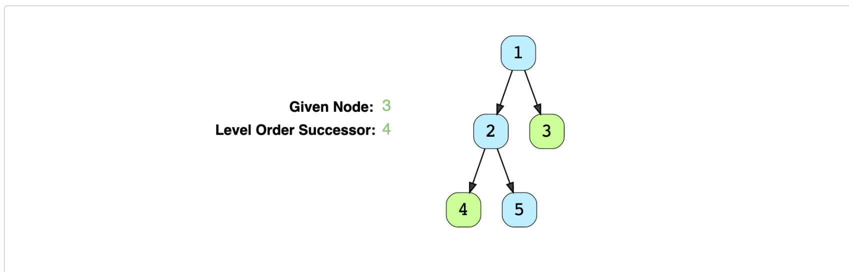
Level Order Successor (easy)

We'll cover the following

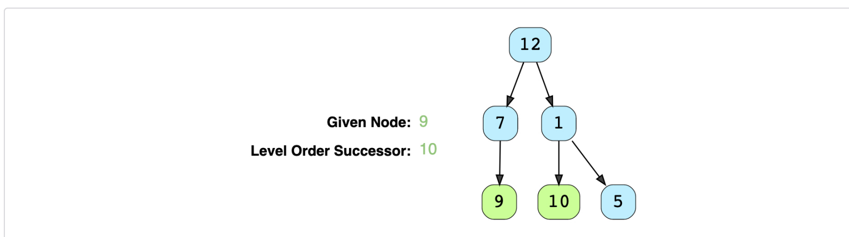
Problem Statement

Given a binary tree and a node, find the level order successor of the given node in the tree. The level order successor is the node that appears right after the given node in the level order traversal.

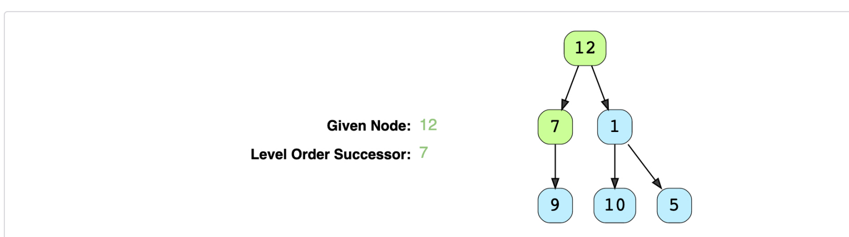
Example 1:



Example 2:



Example 3:



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
  constructor(val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
};


const find_successor = function(root, key) {
  // TODO: Write your code here
  return null;
};


var root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
```

```
root.left.left = new TreeNode(9)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
result = find_successor(root, 12)
if (result != null)
  console.log(result.val)
result = find_successor(root, 9)
if (result != null)
  console.log(result.val)
```

RUN SAVE RESET

Close

✓ Succeeded

Solution

This problem follows the Binary Tree Level Order Traversal pattern. We can follow the same **BFS** approach. The only difference will be that we will not keep track of all the levels. Instead we will keep inserting child nodes to the queue. As soon as we find the given node, we will return the next node from the queue as the level order successor.

Code

Here is what our algorithm will look like; most of the changes are in the highlighted lines:

Java | Python3 | C++ | JS

```
const Deque = require('./collections/deque'); //http://www.collectionsjs.com


class TreeNode {
  constructor(val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}

function find_successor(root, key) {
  if (root === null) {
    return null;
  }

  const queue = new Deque();
  queue.push(root);
  while (queue.length > 0) {
    currentNode = queue.shift();
    // insert the children of current node in the queue
    if (currentNode.left !== null) {
      queue.push(currentNode.left);
    }
    if (currentNode.right !== null) {
      queue.push(currentNode.right);
    }
    // break if we have found the key
```

RUN SAVE RESET

Close

Output 9.197s

```
7
10
```

Time complexity

The time complexity of the above algorithm is $O(N)$, where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

Space complexity

The space complexity of the above algorithm will be $O(N)$ which is required for the queue. Since we can have a maximum of $N/2$ nodes at any level (this could happen only at the lowest level), therefore we will need $O(N)$ space to store them in the queue.

+
Create