educative

# Level Order Successor (easy)

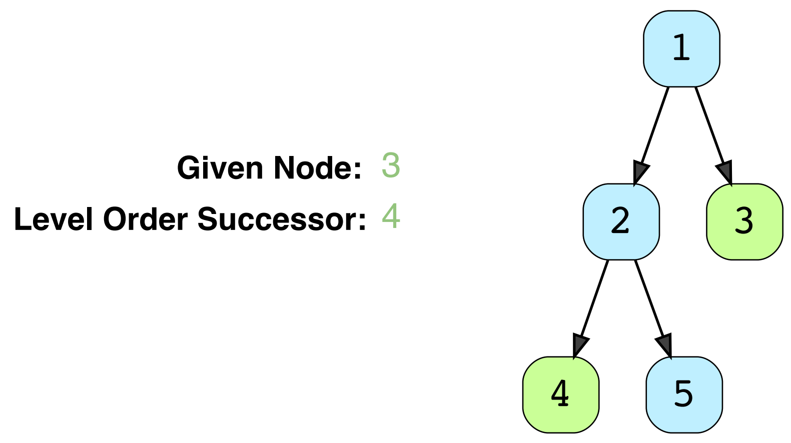
**We'll cover the following**

- Problem Statement
- Try it yourself
- Solution
- Code
  - Time complexity
  - Space complexity

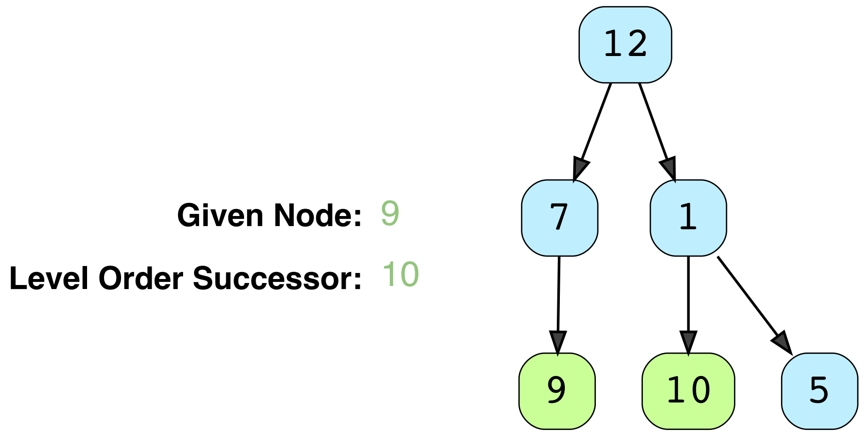
## Problem Statement #

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.

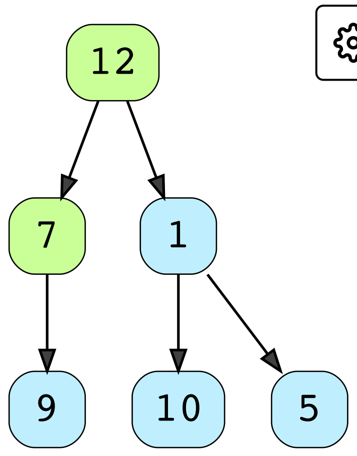
**Example 1:**



## Example 2:



## Example 3:



**Given Node:** 12

**Level Order Successor:** 7

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
from collections import deque


class TreeNode:
  def __init__(self, val):
    self.val = val
    self.left, self.right = None, None


def find_successor(root, key):
  # TODO: Write your code here
  return None


def main():
  root = TreeNode(12)
  root.left = TreeNode(7)
  root.right = TreeNode(1)
  root.left.left = TreeNode(9)
  root.right.left = TreeNode(10)
  root.right.right = TreeNode(5)
  result = find_successor(root, 12)
  if result:
    print(result.val)
  result = find_successor(root, 9)
  if result:
    print(result.val)

```

## Solution #

This problem follows the Binary Tree Level Order Traversal (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5726607939469312/) 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

```


def find_successor(root, key):
  if root is None:
    return None

  queue = deque()
  queue.append(root)
  while queue:
    currentNode = queue.popleft()
    # insert the children of current node in the
    if currentNode.left:
      queue.append(currentNode.left)
    if currentNode.right:
      queue.append(currentNode.right)

    # break if we have found the key
    if currentNode.val == key:
      break

  return queue[0] if queue else None


def main():
  root = TreeNode(12)
  root.left = TreeNode(7)
  root.right = TreeNode(1)
  root.left.left = TreeNode(9)
```

## 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.