Daily Coding Problem #204

Problem

This problem was asked by Amazon.

Given a complete binary tree, count the number of nodes in faster than O(n) time. Recall that a complete binary tree has every level filled except the last, and the nodes in the last level are filled starting from the left.

Solution

If there were no time restrictions, we could perform an in-order traversal of the tree and count up the elements. Instead, we must get a little creative.

Often with tree-based problems it is helpful to think recursively. In this case, we can define the number of nodes as 1 + (number of nodes in the left subtree) + (number of nodes in the right subtree).

By itself this is not terribly useful. But note that for a full tree, the depth to the leftmost node is the same as the depth to the rightmost node, and the number of nodes is $2^{depth+1} - 1$. In other words, in this case, using only two O(h) operations, we can calculate the total number of nodes. As a result, whenever we find a complete subtree, we can calculate its node count directly and stop recursing. And because this is a complete tree, it must be the case that either the left or right subtree (or both) is full, at each level. In this way, we will only need to analyze half the tree at each depth.

Let's look at an example.

Starting from the root, we first check whether the tree is full. Since the depth to the leftmost node is 2 and the depth to the rightmost node is 1, this is not the case. So we must compute `1 +`

`count_nodes(Node(5)) + count_nodes(Node(9))`. For each of these subtrees, we first check if it is full. When we find the left one is full, we know that it must contain $2^{depth+1} - 1$, where `depth = 1`.

We continue recursing on Node(9). Since its left depth is 2 and its right depth is 1, we compute `1 + count_nodes(Node(8)) + count_nodes(None)`. Since `Node(8)` has no children, it is a full tree of depth 1, so its total number of nodes is $2^1 - 1$. To handle the right child, we specify a base case that returns 0 whenever the node passed in does not exist.

```
def find_left_height(root):
    height = 0
    while root.left:
        root = root.left
        height += 1

    return height

def find_right_height(root):
    height = 0
    while root.right:
        root = root.right
        height += 1
    return height

def count_nodes(root):
    if not root:
        return 0
    left = find_left_height(root)
    right = find_right_height(root)

    if left == right:
        return (2 << left) - 1
    else:
        return count_nodes(root.left) + count_nodes(root.right) + 1
```

At each level of the tree we make a constant number of `O(h)` calls, where `h` is the depth at that level. So our time complexity is $O(h + (h - 1) + ... + 0) = O(h^2) = O((\log N)^2)$.