Daily Coding Problem #112

Problem

This problem was asked by Twitter.

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. Assume that each node in the tree also has a pointer to its parent.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)."

Solution

When given a problem statement, make sure to clarify the question, as well as its inputs and outputs. In this problem, we are given a binary tree, and not necessarily a binary search tree. We will be given the root of a binary tree as input, as well as two nodes `a` and `b`. We are to return the node that satisfies the LCA definition of the two nodes, or `null` if no such nodes exist.

It's also important to identify edge cases, and provide examples of these cases. One important edge case is where both `a` and `b` point to the same node. Another case that may be worth mentioning is if nodes `a` and `b` do not share any ancestors. In this case, `a` and `b` must not be in the same tree. You can clarify with your interviewer whether or not you can assume `a` and `b` are in the tree.

The key to this problem is that we are given parent pointers. Although this problem can be solved without parent pointers, the iterative solution is simplified when given these. The interviewer may also try to direct your solution to use the pointers.

In order to find the LCA, we have to identify the first node in which the paths from `a` and `b` intersect while traversing upwards to the root. One way we can do this is by using two loops. For each node in `a`'s path, check whether there is a node in `b`'s path that is the same. This solution would have an overall time complexity of `O(depth(a) * depth(b))`, which could be `O(N^2)` in the worst-case, where `a` and `b` are leaf nodes, and the tree is unbalanced (close to a linked list). The space complexity is `O(1)`.

We can reduce the time complexity by trading off space. We first store the nodes seen in the path from a in a hash set. Then, while iterating from b to the root, we check whether the current node is found in the set. We return the first node that meets this condition. Then, we have a solution which runs in linear (or `O(depth(a) + depth(b))`) time, and linear space.

However, we can improve the space complexity on this solution by observing that both paths must converge and end at the root, as long as both nodes are in the same tree. Then, we can reduce this problem to finding the first node of two linked lists' intersection. First, we get the lengths of both nodes' paths (their depth in the tree). Then, we move the longer path's pointer forward so both pointers are the same distance from the root. Finally, we move both pointers in lock-step and return when we find the first node that is the same from both paths.

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

def lca(root, a, b):
    def depth(node):
        count = 0
        while node:
            count += 1
            node = node.parent

        return count

    depth_a, depth_b = depth(a), depth(b)
    if depth_a < depth_b:
        while depth_a < depth_b:
            b = b.parent
            depth_b -= 1
    elif depth_a > depth_b:
        while depth_a > depth_b:
            a = a.parent
            depth_a -= 1

    while a and b and (a is not b):
        a = a.parent
        b = b.parent

    return a if (a is b) else None
```

Now, our solution has a linear time complexity, and uses constant space.