Closest Leaf in a Binary Tree

Given a binary tree **where every node has a unique value**, and a target key `k`, find the value of the closest leaf node to target `k` in the tree.

Here, *closest* to a leaf means the least number of edges travelled on the binary tree to reach any leaf of the tree. Also, a node is called a *leaf* if it has no children.

In the following examples, the input tree is represented in flattened form row by row. The actual `root` tree given will be a TreeNode object.

Example 1:

```
Input:
root = [1, 3, 2], k = 1
Diagram of binary tree:
          1
         / \
        3   2

Output: 2 (or 3)

Explanation: Either 2 or 3 is the closest leaf node to the target of 1.
```

Example 2:

```
Input:
root = [1], k = 1
Output: 1

Explanation: The closest leaf node is the root node itself.
```

Example 3:

```
Input:
root = [1,2,3,4,null,null,null,5,null,6], k = 2
Diagram of binary tree:
             1
            / \
           2   3
          /
         4
        /
       5
      /
     6

Output: 3
Explanation: The leaf node with value 3 (and not the leaf node with value 6) is close
st to the node with value 2.
```

Note:

1. `root` represents a binary tree with at least `1` node and at most `1000` nodes.
2. Every node has a unique `node.val` in range `[1, 1000]`.
3. There exists some node in the given binary tree for which `node.val == k`.

Solution 1

Should be immediately banned from submitting questions. What a horrible description!!! The word "closest" could be interpreted in so many ways. The fact that this person did not even consider that others will get confused by his/her choice of words says a lot about their communication style.

written by galster original link here

Solution 2

Treat the tree as an undirected graph and perform BFS from the target node to find the first leaf node.

We first run `traverse` function to convert the tree into an undirected graph and keep track of the leaf nodes in a set.

For a tree that looks like this:

```
          1
         / \
        2   3
       /
      4
     /
    5
   /
  6
```

the `graph` dict and `leaves` set look like:

```
# graph
{
  1: [2, 3],
  2: [1, 4],
  3: [1],
  4: [2, 5],
  5: [4, 6],
  6: [5],
}

# leaves
{3, 6}
```

Note that a node can never have more than three neighbors as per the Binary Tree property (two children, one parent).

Then we perform a BFS on the graph from node K, terminating as soon as we find a leaf node.

- Yangshun

```python
class Solution(object):
    def findClosestLeaf(self, root, k):
        # Time: O(n)
        # Space: O(n)
        from collections import defaultdict
        graph, leaves = defaultdict(list), set()
        # Graph construction
        def traverse(node):
            if not node:
                return
            if not node.left and not node.right:
                leaves.add(node.val)
                return
            if node.left:
                graph[node.val].append(node.left.val)
                graph[node.left.val].append(node.val)
                traverse(node.left)
            if node.right:
                graph[node.val].append(node.right.val)
                graph[node.right.val].append(node.val)
                traverse(node.right)
        traverse(root)
        # Graph traversal - BFS
        queue = [k]
        while len(queue):
            next_queue = []
            for node in queue:
                if node in leaves:
                    return node
                next_queue += graph.pop(node, [])
            queue = next_queue
```

Thanks to @ManuelP for shortening my solution yet again 🅰🅰

written by yangshun original link here

Solution 3

My solution is simple:

1. First, preform DFS on root in order to find the node whose val = k, at the meantime use `HashMap` to keep record of all back edges from child to parent;
2. Then perform BFS on this node to find the closest leaf node.

```java
class Solution {

    public int findClosestLeaf(TreeNode root, int k) {
        Map<TreeNode, TreeNode> backMap = new HashMap<>();   // store all edges that trace node back to its parent
        Queue<TreeNode> queue = new LinkedList<>();          // the queue used in BFS
        Set<TreeNode> visited = new HashSet<>();             // store all visited nodes

        // DFS: search for node whoes val == k
        TreeNode kNode = DFS(root, k, backMap);
        queue.add(kNode);
        visited.add(kNode);

        // BFS: find the shortest path
        while(!queue.isEmpty()) {
            TreeNode curr = queue.poll();
            if(curr.left == null && curr.right == null) {
                return curr.val;
            }
            if(curr.left != null && visited.add(curr.left)) {
                queue.add(curr.left);
            }
            if(curr.right != null && visited.add(curr.right)) {
                queue.add(curr.right);
            }
            if(backMap.containsKey(curr) && visited.add(backMap.get(curr))) {  // go alone the back edge
                queue.add(backMap.get(curr));
            }
        }
        return -1; // never hit
    }

    private TreeNode DFS(TreeNode root, int k, Map<TreeNode, TreeNode> backMap) {
        if(root.val == k) {
            return root;
        }
        if(root.left != null) {
            backMap.put(root.left, root);        // add back edge
            TreeNode left = DFS(root.left, k, backMap);
            if(left != null) return left;
        }
        if(root.right != null) {
            backMap.put(root.right, root);       // add back edge
            TreeNode right = DFS(root.right, k, backMap);
            if(right != null) return right;
        }
        return null;
    }

}
```

written by cwleoo original link here

From Leetcoder.