# 742. Closest Leaf in Binary Tree

Dec. 10, 2017 | 13.3K views

Average Rating: 3.88 (8 votes)

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

Here, *nearest* 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 nearest leaf node to the target of 1.
```

**Example 2:**

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

Explanation: The nearest 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 neares
```

**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`.

## Approach #1: Convert to Graph [Accepted]

**Intuition**

Instead of a binary tree, if we converted the tree to a general graph, we could find the shortest path to a leaf using breadth-first search.

**Algorithm**

We use a depth-first search to record in our graph each edge travelled from parent to node.

After, we use a breadth-first search on nodes that started with a value of `k`, so that we are visiting nodes in order of their distance to `k`. When the node is a leaf (it has one outgoing edge, where the `root` has a "ghost" edge to `null`), it must be the answer.

Java Python Copy

```python
class Solution(object):
    def findClosestLeaf(self, root, k):
        graph = collections.defaultdict(list)
        def dfs(node, par = None):
            if node:
                graph[node].append(par)
                graph[par].append(node)
                dfs(node.left, node)
                dfs(node.right, node)

        dfs(root)
        queue = collections.deque(node for node in graph
                                  if node and node.val == k)
        seen = set(queue)

        while queue:
            node = queue.popleft()
            if node:
                if len(graph[node]) <= 1:
                    return node.val
                for nei in graph[node]:
                    if nei not in seen:
                        seen.add(nei)
                        queue.append(nei)
```

**Complexity Analysis**

- Time Complexity: $O(N)$ where $N$ is the number of nodes in the given input tree. We visit every node a constant number of times.
- Space Complexity: $O(N)$, the size of the graph.

## Approach #2: Annotate Closest Leaf [Accepted]

**Intuition and Algorithm**

Say from each node, we already knew where the closest leaf in it's subtree is. Using any kind of traversal plus memoization, we can remember this information.

Then the closest leaf to the target (in general, not just subtree) has to have a lowest common ancestor with the `target` that is on the path from the `root` to the `target`. We can find the path from `root` to `target` via any kind of traversal, and look at our annotation for each node on this path to determine all leaf candidates, choosing the best one.

Java Python Copy

```python
class Solution(object):
    def findClosestLeaf(self, root, k):
        annotation = {}
        def closest_leaf(root):
            if root not in annotation:
                if not root:
                    ans = float('inf'), None
                elif not root.left and not root.right:
                    ans = 0, root
                else:
                    d1, leaf1 = closest_leaf(root.left)
                    d2, leaf2 = closest_leaf(root.right)
                    ans = min(d1, d2) + 1, leaf1 if d1 < d2 else leaf2
                annotation[root] = ans
            return annotation[root]

        #Search for node.val == k
        path = []
        def dfs(node):
            if not node:
                return
            if node.val == k:
                path.append(node)
                return True
            path.append(node)
            ans1 = dfs(node.left)
            if ans1: return True
            ans2 = dfs(node.right)
```

**Complexity Analysis**

- Time and Space Complexity: $O(N)$. The analysis is the same as in *Approach #1*.

Analysis written by: @awice.

Rate this article:



Comments: 8

Sort By

Type comment here... (Markdown is supported)

Preview Post

coder001 ★ 18 November 15, 2018 11:36 PM

In solution 1, condition `if len(graph[node]) <= 1:` can be simplified to `if len(graph[node]) == 1:` as all leaf nodes have exactly only one neighbor.

1 Share Reply

SHOW 2 REPLIES

rajatmittal18 ★ 225 June 28, 2019 8:30 AM Report

In solution 2, we actually don't need to call closest_leaf function more than once, just call it once, form the annotation dictionary and use it while traversing the path.

0 Share Reply

kanerodriguezpro ★ 1 May 19, 2019 8:28 PM

Simple tree traversal beats **98% time** | **67% space**

```
class Solution:
    def findClosestLeaf(self, root: TreeNode, k: int) -> int:
        def helper(node: TreeNode, k: int) -> (int,int,int):
```

Read More

0 Share Reply

calvinchankf ★ 2522 April 25, 2019 8:06 AM

A bit more detail on approach 1, since the description mentions that all nodes have diff integer values and they are ranged `from 1 to 1000`, we can use the integer values as keys and use -1 as null parent or children. i think its easier to understand and implement in an interview

Convert the tree to a graph + bfs + hashtable

Read More

0 Share Reply

sunnyrain013 ★ 3 March 24, 2019 1:00 AM

In solution 2, why d0 += len(path) - 1 - i ? Thanks!

0 Share Reply

xivaxy ★ 0 December 12, 2017 11:39 PM

In Python Approach#1:

```
for node in queue:
    if node:
        ....
```

Read More

0 Share Reply

SHOW 1 REPLY

ImranChoudhury ★ 6 September 11, 2019 10:26 AM Report

For Solution 1, Since the question already mentioned that every node has unique values,I don't think we need the seen = set(queue) here. We can simply use queue for the BFS.

```
graph = collections.defaultdict(list)
def dfs(node, par = None):
```

Read More

0 Share Reply

SHOW 1 REPLY

iit_coders ★ 2 August 10, 2019 9:27 AM

Any idea what's wrong with this C++ solution equivalent to the first solution (DFS+BFS) in Java ?

```
typedef map<TreeNode*,list<TreeNode*>> Graph;
```

Read More

0 Share Reply