## 863. All Nodes Distance K in Binary Tree Medium Tree **Binary Tree Depth-First Search Breadth-First Search**

to find nodes only in the subtree of the target node but from the entire tree, so we must consider not only the children of the target node but also its ancestors and siblings. Intuition

The problem presents a binary tree and asks us to find all nodes that are a certain distance k from a given target node. It's important

to note that distance here refers to the number of edges we must traverse to reach one node from another. This problem doesn't ask

**Problem Description** 

case with tree traversals) but also upwards, from any node to its parent. This is not usually possible in a binary tree because nodes don't have a reference to their parent. Therefore, we create a map that keeps track of each node's parent.

2. Performing the Search: Once we have the ability to move both up and down the tree, we perform a depth-first search (DFS)

To solve this problem, we need to find all nodes that are at the specified distance k from the target node, regardless of whether

they're ancestors, descendants, or neither (siblings or cousins). The approach to finding this solution is divided into several parts:

1. Mapping Parents: We first need a way to traverse the tree not only downwards from the root to the leaves (as is normally the

- starting from the target node. As we explore the tree, we keep track of the current distance from the target. When this distance equals **k**, we add the current node's value to our answer. 3. Avoiding Revisits: To ensure we don't count any nodes twice or enter a loop, we keep a set of visited nodes. Whenever we visit
- a node, we add it to the set. If we encounter a node that's already in our set, we skip it. With this approach, we can explore all possible paths - down to the children, up to the parent, and across to siblings and other relatives - to find all nodes that are exactly k distance away from the target.
- **Solution Approach** The solution is implemented using a depth-first search (DFS) algorithm along with additional data structures to track parent nodes

and visited nodes. Here's a step-by-step breakdown of how the code accomplishes the task: 1. Parent Mapping: The first function, parents, is a recursive function that populates a dictionary p where keys are nodes of the

# search, starting from the target node. The function takes a node and the distance k as arguments.

then right child) and assigns the current node's parent to it in the dictionary. 2. Depth-First Search (DFS): The second function, dfs, is the crucial part of the implementation where we perform a depth-first

tree and values are their respective parent nodes. It goes through all the nodes in a pre-order traversal (parent, then left child,

3. Visited Set: Inside the DFS function, we first ensure that the current node is not None and hasn't been visited yet, as indicated by

4. Base Case for Distance: If the distance k is 0, it means we have found a node at the exact distance from the target we are looking for, so we append the value of the current node to the ans list, which will eventually be our output. 5. Recursive Case for Traversal: If k is not 0, we recursively call DFS on the left child, the right child, and the parent of the current

node, each time with k - 1 to account for the decrease in distance as we move one step away from the target.

checking the vis set. If we have seen the node already, we return early to avoid revisiting it.

and we want to find all the nodes that are a distance k = 2 away from the target node with value 5.

target node. At the end of the function, we simply return ans.

Step 1: Parent Mapping First, we construct a map of parent pointers:

- 6. Execution: Finally, we initialize our data structures: p for parents, ans for answers, and vis for visited nodes. We then populate p using the parents function and execute dfs starting with the target node and the initial distance k. 7. Result: After DFS completes the traversal of the tree, and will contain all the node values that are k distance away from the
- possible paths from the target node to find nodes that are exactly k steps away while avoiding loops and repetitions. Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above. Suppose we have the following binary tree

In summary, the algorithm traverses the entire binary tree to build a parent-child relationship map, then it uses DFS to search all

 The parent of node 4 is 2. The parent of node 5 is 2, and so on.

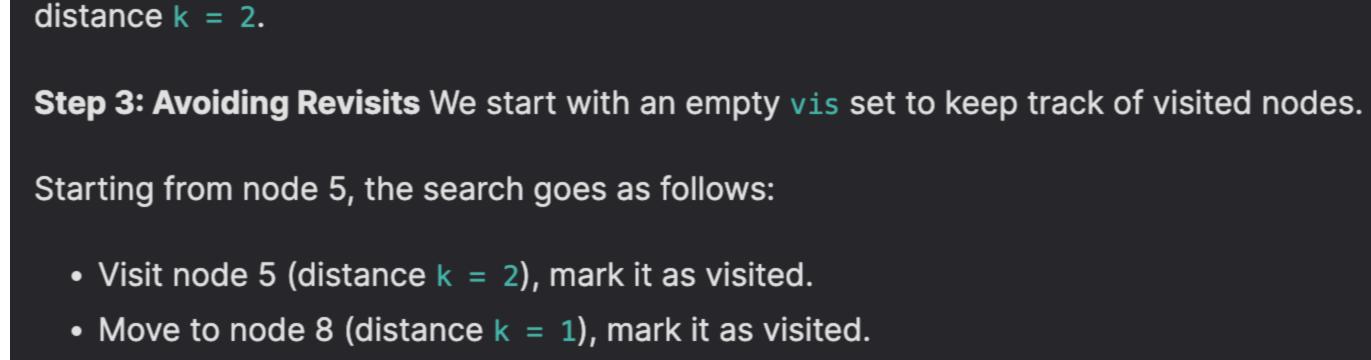
Step 2: Performing the Search We then perform a DFS starting from the target node 5. We are looking for nodes that are at a

Move to node 4 (distance k = 0), since the distance is 0 and node 4 hasn't been visited, add it to the answer.

Since node 9 has no children, return to node 5 and move up to its parent node 2 (distance k = 1), mark it as visited.

• Since node 2 is visited for the first time with distance k = 1, move to node 4 (already visited) and node 3 (distance k = 0 via

By following the steps in the provided solution approach, we have successfully found all nodes at a specific distance from a given



target in a binary tree.

Python Solution

2 class TreeNode:

class Solution:

10

11

12

13

14

15

16

17

18

19

20

21

28

35

36

37

38

39

40

41

42

43

21

22

23

24

25

26

27

28

29

30

31

32

33

34

41

42

43

44

45

46

47

48

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

6

8

9

10

11

13

16

19

22

31

33

36

37

38

39

41

42 }

32 }

12 }

**}**;

1 # Definition for a binary tree node.

if node:

return

visited = set()

return result

map\_parents(root, None)

The parent of node 2 is 1.

The parent of node 3 is 1.

Step 4: Summary of the Results At the end of the DFS, we find that the nodes at distance 2 from the target node 5 are nodes 3 and 4. Thus, the answer is [3, 4].

def \_\_init\_\_(self, val=0, left=None, right=None):

parent\_map[node] = parent

map\_parents(node.left, node)

if not node or node.val in visited:

# Map parents for all nodes in the tree

// Associates each node with its parent.

findNodesAtDistanceK(target, k);

// Maps each node to its parent recursively.

// Perform DFS to find nodes at distance K.

private void mapParents(TreeNode node, TreeNode parent) {

private void findNodesAtDistanceK(TreeNode node, int k) {

if (node == null || visited.contains(node.val)) {

// Helper method to relate each node to its parent in the tree

// The DFS method that traverses the tree to find nodes at a distance k

// If the desired distance is reached, add the node's value to the results

// If the node is null or already visited, stop the search

void storeParents(TreeNode\* node, TreeNode\* parent) {

if (!node) return;

parentMap[node] = parent;

storeParents(node->left, node);

storeParents(node->right, node);

void depthFirstSearch(TreeNode\* node, int k) {

// Mark the current node as visited

result.push\_back(node->val);

depthFirstSearch(node->left, k - 1);

depthFirstSearch(node->right, k - 1);

visited.insert(node->val);

**if** (k == 0) {

Typescript Solution

val: number;

2 class TreeNode {

1 // Definition for a binary tree node.

left: TreeNode | null;

right: TreeNode | null;

this.val = val;

14 // Maps each node to its parent

17 // Keeps track of visited nodes

21 const result: number[] = [];

this.left = null;

this.right = null;

18 const visited: Set<number> = new Set();

// Find and store all parents of nodes

constructor(val: number) {

return;

if (!node || visited.count(node->val)) return;

// Continue the search in left and right subtree

// Also consider the parent node in the search

depthFirstSearch(parentMap[node], k - 1);

15 const parentMap: Map<TreeNode, TreeNode | null> = new Map();

// Function that returns all nodes at distance K from the target node

// Helper function to relate each node to its parent in the tree

function storeParents(node: TreeNode | null, parent: TreeNode | null) {

function distanceK(root: TreeNode | null, target: TreeNode, k: number): number[] {

// DFS to find and add all nodes that are K distance from the given node.

// Base case: if the node is null or already visited, stop the search.

mapParents(root, null);

return answer;

if (node == null) {

return;

return;

# Start DFS from the target node

find\_nodes\_at\_distance\_k(target, k)

map\_parents(node.right, node)

def map\_parents(node, parent):

def distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:

# Helper function to perform Depth-First Search (DFS) and find nodes at distance k

find\_nodes\_at\_distance\_k(node.right, remaining\_distance - 1)

# Helper function to store parent pointers for each node

def find\_nodes\_at\_distance\_k(node, remaining\_distance):

parent 1). Again, as the distance is 0, add node 3 to the answer.

Similarly, move to node 9 (distance k = 1), mark it as visited.

self.val = val self.left = left self.right = right

visited.add(node.val) 23 if remaining\_distance == 0: 24 result.append(node.val) 25 else: 26 # Visit children and parent find\_nodes\_at\_distance\_k(node.left, remaining\_distance - 1) 27

```
find_nodes_at_distance_k(parent_map[node], remaining_distance - 1)
29
30
31
            # Initialize the parent map and result list
32
            parent_map = {}
33
            result = []
```

```
Java Solution
1 import java.util.*;
   // Definition for a binary tree node.
   class TreeNode {
       int val;
       TreeNode left;
       TreeNode right;
       TreeNode(int x) { val = x; }
9 }
10
  class Solution {
       private Map<TreeNode, TreeNode> parentMap;
12
       private Set<Integer> visited;
13
       private List<Integer> answer;
14
15
16
       // This method finds all nodes at distance K from the target node.
17
       public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {
18
           parentMap = new HashMap<>(); // Used to store parent-child relationships.
           visited = new HashSet<>(); // Tracks visited nodes to prevent cycles.
19
           answer = new ArrayList<>(); // Stores the results.
20
```

### 35 36 37 parentMap.put(node, parent); mapParents(node.left, node); // Parent for left child is 'node' 39 mapParents(node.right, node); // Parent for right child is 'node' 40

```
visited.add(node.val); // Mark this node as visited.
49
50
           if (k == 0) {
51
52
               // If the distance is zero, the current node is K distance from the target.
53
               answer.add(node.val);
54
               return; // Node added to the answer, end this path here.
55
56
57
           // Continue searching in the left subtree.
58
           findNodesAtDistanceK(node.left, k - 1);
59
60
           // Continue searching in the right subtree.
           findNodesAtDistanceK(node.right, k - 1);
61
           // Also search the sub-tree above using the parent references.
64
           findNodesAtDistanceK(parentMap.get(node), k - 1);
65
66 }
67
C++ Solution
  1 /**
     * Definition for a binary tree node.
     */
    struct TreeNode {
         int val;
        TreeNode *left;
        TreeNode *right;
         TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    };
  9
 10
 11 class Solution {
 12 public:
         // Maps each node to its parent
 13
 14
         unordered_map<TreeNode*, TreeNode*> parentMap;
 15
 16
         // Keeps track of visited nodes
 17
         unordered_set<int> visited;
 18
 19
         // Stores the results
         vector<int> result;
 20
 21
 22
         // Function that returns all nodes at distance K from the target node
 23
         vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
             // Find and store all parents of nodes
 24
 25
             storeParents(root, nullptr);
 26
 27
             // Perform search to find all nodes at distance k
 28
             depthFirstSearch(target, k);
 29
 30
             return result;
```

### 26 storeParents(root, null); 27 28 // Perform search to find all nodes at distance k 29 depthFirstSearch(target, k); 30

return result;

if (!node) return;

parentMap.set(node, parent);

storeParents(node.left, node);

storeParents(node.right, node);

20 // Stores the results

43 // The DFS method that traverses the tree to find nodes at a distance k function depthFirstSearch(node: TreeNode | null, k: number) { 46 // If the node is null or already visited, stop the search if (!node || visited.has(node.val)) return; 47 48 49 // Mark the current node as visited 50 visited.add(node.val); 51 52 // If the desired distance is reached, add the node's value to the results 53 **if** (k === 0) { 54 result.push(node.val); 55 return; 56 57 58 // Continue the search in the left and right subtree depthFirstSearch(node.left, k - 1); 59 60 depthFirstSearch(node.right, k - 1); 61 62 // Also consider the parent node in the search 63 const parentNode = parentMap.get(node) || null; 64 depthFirstSearch(parentNode, k - 1); 65 } 66 Time and Space Complexity The provided Python code defines a function distanceK that finds all nodes at distance k from a target node in a binary tree. **Time Complexity:** To analyze the time complexity, let's break down the procedure into its main steps:

1. The function parents traverses the entire tree to build a dictionary p mapping each node to its parent. This is a depth-first

2. The function dfs is a recursive depth-first search. In the worst case, it can visit each node once when k equals the height of the

search (DFS) with a single visit to each node, hence its complexity is O(N), where N is the number of nodes in the tree.

Given these steps, the overall time complexity is O(N) since both parents and dfs functions operate linearly with respect to the

2. The ans list could hold up to N nodes (in the case that the tree is a straight line and k equals N-1), so its space is 0(N).

tree. Additionally, it might visit each node's parent, resulting in O(2N) operations, which simplifies to O(N).

Now, let's consider the space complexity: 1. The p dictionary holds a pair for each node in the tree, hence O(N) space.

number of nodes in the tree.

**Space Complexity:** 

3. The call stack for the DFS function can go up to the height of the tree in the case of a skewed tree, which in the worst case is O(N). 4. The vis set contains nodes that are already visited, which in the worst case can be O(N) as well when every node is visited.

- The space complexity is determined by adding the space required for the p dictionary, the ans list, the call stack, and the vis set, which all contribute linearly to O(N) space.
- Overall, taking into account both the worst-case scenarios for the recursive calls stack and the space needed for auxiliary data structures, the space complexity is O(N).