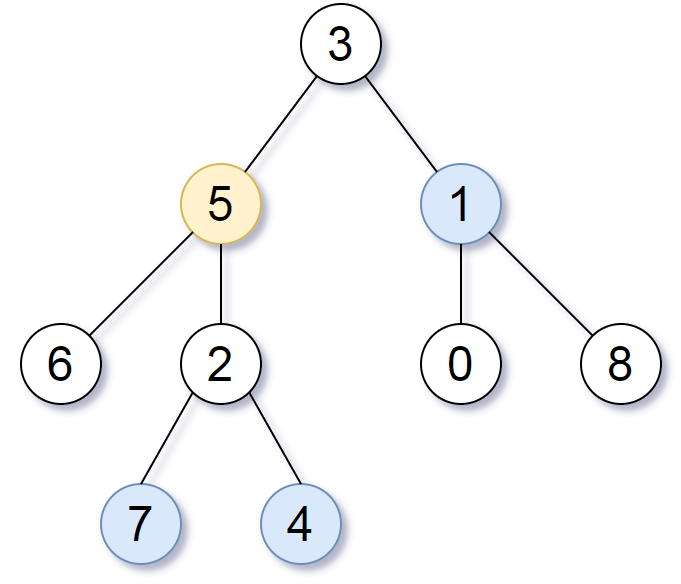
<https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/description/>

Given the root of a binary tree, the value of a target node target, and an integer k, return *an array of the values of all nodes that have a distance*k*from the target node.*

You can return the answer in **any order**.

**Example 1:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2

**Output:** [7,4,1]Explanation: The nodes that are a distance 2 from the target node (with value 5) have values 7, 4, and 1.

**Example 2:**

**Input:** root = [1], target = 1, k = 3

**Output:** []

**Constraints:**

The number of nodes in the tree is in the range [1, 500].

0 <= Node.val <= 500

All the values Node.val are **unique**.

target is the value of one of the nodes in the tree.

0 <= k <= 1000

**Attempt 1: 2025-08-07**

**Solution 1: DFS + Hash Table (60 min)**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) { val = x; }

 \* }

 \*/

class Solution {

    public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {

        List<Integer> result = new ArrayList<>();

        Map<TreeNode, TreeNode> parentMap = new HashMap<>();

        buildParentMap(root, null, parentMap);

        Set<TreeNode> visited = new HashSet<>();

        helper(target, k, visited, parentMap, result);

        return result;

    }

    private void buildParentMap(TreeNode node, TreeNode parent,

        Map<TreeNode, TreeNode> parentMap) {

        if(node == null) {

            return;

        }

        parentMap.put(node, parent);

        buildParentMap(node.left, node, parentMap);

        buildParentMap(node.right, node, parentMap);

    }

    private void helper(TreeNode node, int k, Set<TreeNode> visited,

        Map<TreeNode, TreeNode> parentMap, List<Integer> result) {

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

                return;

            }

            visited.add(node);

            if(k == 0) {

                result.add(node.val);

                return;

            }

            // Recurse on left child

            helper(node.left, k - 1, visited, parentMap, result);

            // Recurse on right child

            helper(node.right, k - 1, visited, parentMap, result);

            // Recurse on parent

            helper(parentMap.get(node), k - 1, visited, parentMap, result);

        }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 2: BFS + Hash Table (60 min)**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) { val = x; }

 \* }

 \*/

class Solution {

    public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {

        List<Integer> result = new ArrayList<>();

        Map<TreeNode, TreeNode> parentMap = new HashMap<>();

        buildParentMap(root, null, parentMap);

        Set<TreeNode> visited = new HashSet<>();

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(target);

        visited.add(target);

        int distance = 0;

        while(!queue.isEmpty() && distance < k) {

            int size = queue.size();

            for(int i = 0; i < size; i++) {

                TreeNode node = queue.poll();

                // Add left child

                if(node.left != null && !visited.contains(node.left)) {

                    queue.offer(node.left);

                    visited.add(node.left);

                }

                // Add right child

                if(node.right != null && !visited.contains(node.right)) {

                    queue.offer(node.right);

                    visited.add(node.right);

                }

                // Add parent

                TreeNode parent = parentMap.get(node);

                if(parent != null && !visited.contains(parent)) {

                    queue.offer(parent);

                    visited.add(parent);

                }

            }

            distance++;

        }

        while(!queue.isEmpty()) {

            result.add(queue.poll().val);

        }

        return result;

    }

    private void buildParentMap(TreeNode node, TreeNode parent,

        Map<TreeNode, TreeNode> parentMap) {

        if(node == null) {

            return;

        }

        parentMap.put(node, parent);

        buildParentMap(node.left, node, parentMap);

        buildParentMap(node.right, node, parentMap);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

Here's a Depth-First Search (DFS) approach to solve this problem, which provides an alternative to the BFS solution.

**DFS Approach**

**Build Parent Map**: First create a mapping from each node to its parent (same as BFS approach)

**DFS Traversal**: Perform a DFS starting from the target node, tracking the current distance from target

**Collect Results**: When reaching distance K, collect the nodes

**Solution Code**

class Solution {

public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {

Map<TreeNode, TreeNode> parentMap = new HashMap<>();

buildParentMap(root, null, parentMap);

List<Integer> result = new ArrayList<>();

Set<TreeNode> visited = new HashSet<>();

dfs(target, k, parentMap, visited, result);

return result;

}

private void dfs(TreeNode node, int k, Map<TreeNode, TreeNode> parentMap,

Set<TreeNode> visited, List<Integer> result) {

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

return;

}

visited.add(node);

if (k == 0) {

result.add(node.val);

return;

}

// Recurse on left child

dfs(node.left, k - 1, parentMap, visited, result);

// Recurse on right child

dfs(node.right, k - 1, parentMap, visited, result);

// Recurse on parent

TreeNode parent = parentMap.get(node);

dfs(parent, k - 1, parentMap, visited, result);

}

private void buildParentMap(TreeNode node, TreeNode parent, Map<TreeNode, TreeNode> parentMap) {

if (node == null) return;

parentMap.put(node, parent);

buildParentMap(node.left, node, parentMap);

buildParentMap(node.right, node, parentMap);

}

}

**Explanation**

**Parent Mapping**: The buildParentMap function recursively creates a parent-child relationship map, same as in the BFS solution.

**DFS Traversal**:

Start DFS from the target node

Track visited nodes to avoid cycles

Decrement K at each level of recursion

When K reaches 0, add the current node to results

**Three-Way Recursion**:

Explore left child

Explore right child

Explore parent (using the parent map)

Key differences from BFS:

Uses recursion stack instead of a queue

Explores each path fully before backtracking

Still maintains O(n) time complexity

Call stack depth could be O(n) in worst case (unbalanced tree)

This DFS solution provides the same result as BFS but with a different traversal order. Both are valid approaches with similar time/space complexity.

**Can we have a BFS solution ?**

**Approach**

To find all nodes at distance K from a target node in a binary tree:

**Build Parent Map**: Create a mapping from each node to its parent to enable upward traversal.

**BFS Traversal**: Perform a breadth-first search starting from the target node, expanding outward to nodes at distance K.

**Track Visited Nodes**: Avoid revisiting nodes by maintaining a visited set.

**Solution Code**

class Solution {

public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {

Map<TreeNode, TreeNode> parentMap = new HashMap<>();

buildParentMap(root, null, parentMap);

Queue<TreeNode> queue = new LinkedList<>();

Set<TreeNode> visited = new HashSet<>();

queue.offer(target);

visited.add(target);

int distance = 0;

while (!queue.isEmpty() && distance < k) {

int size = queue.size();

for (int i = 0; i < size; i++) {

TreeNode current = queue.poll();

// Add left child

if (current.left != null && !visited.contains(current.left)) {

queue.offer(current.left);

visited.add(current.left);

}

// Add right child

if (current.right != null && !visited.contains(current.right)) {

queue.offer(current.right);

visited.add(current.right);

}

// Add parent

TreeNode parent = parentMap.get(current);

if (parent != null && !visited.contains(parent)) {

queue.offer(parent);

visited.add(parent);

}

}

distance++;

}

List<Integer> result = new ArrayList<>();

while (!queue.isEmpty()) {

result.add(queue.poll().val);

}

return result;

}

private void buildParentMap(TreeNode node, TreeNode parent, Map<TreeNode, TreeNode> parentMap) {

if (node == null) return;

parentMap.put(node, parent);

buildParentMap(node.left, node, parentMap);

buildParentMap(node.right, node, parentMap);

}

}

**Explanation**

**Parent Mapping**: The buildParentMap function recursively creates a mapping of each node to its parent, enabling upward traversal from any node.

**BFS Initialization**: Start BFS from the target node, marking it as visited.

**Level-wise Expansion**: For each level (distance from target):

Process all nodes at the current distance.

Add unvisited left child, right child, and parent to the queue for the next level.

**Result Collection**: After reaching distance K, collect all remaining nodes in the queue as the result.

This approach efficiently explores the tree in all directions (children and parent) while maintaining the correct distance from the target node. The time complexity is O(n) where n is the number of nodes, and space complexity is O(n) for the parent map and queue.

**Refer to**

[L2385.Amount of Time for Binary Tree to Be Infected (Ref.L863)](note://WEBbedc4c89d9307474c33cd65c4a942605)