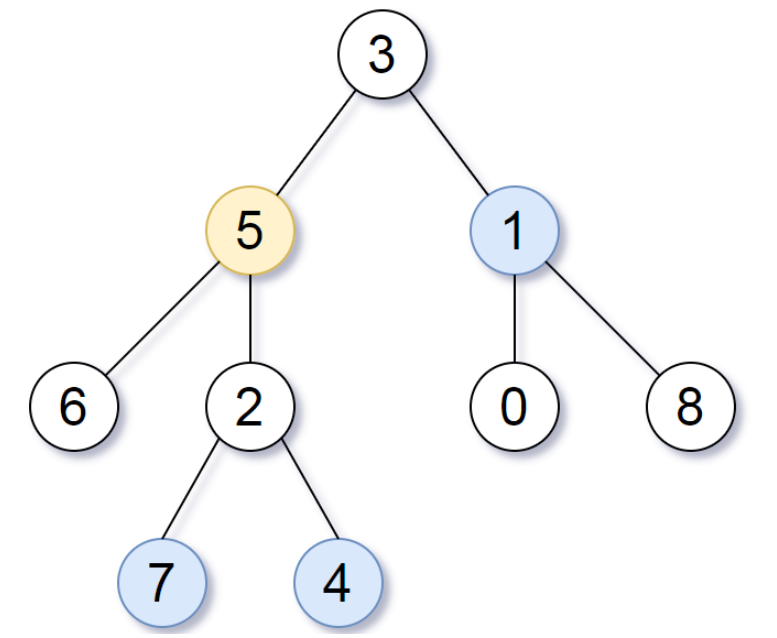
# 描述

给定一个二叉树（具有根结点root），一个目标结点target，和一个整数值K。

返回到目标结点target距离为K的所有结点的值的列表。答案可以以**任何顺序**返回。

**示例 1：**



输入：root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

输出：[7,4,1]

**解释：**

所求结点为与目标结点（值为5）距离为2的结点，值分别为7，4，以及1

注意，输入的 "root" 和 "target" 实际上是树上的结点。

上面的输入仅仅是对这些对象进行了序列化描述。

示例 2:

输入: root = [1], target = 1, k = 3

输出: []

**提示：**

节点数在 [1, 500] 范围内

0 <= Node.val <= 500

Node.val 中所有值 不同

目标结点 target 是树上的结点。

0 <= k <= 1000

# 代码

## 方法一：深度优先搜索 + 哈希表

**思路：**

若将target当作树的根结点，我们就能从target出发，使用深度优先搜索去寻找与target距离为k的所有结点，即深度为k的所有结点。

由于输入的二叉树没有记录父结点，为此，我们从根结点root出发，使用深度优先搜索遍历整棵树，同时用一个哈希表记录每个结点的父结点。

然后从target出发，使用深度优先搜索遍历整棵树，除了搜索左右儿子外，还可以顺着父结点向上搜索。

代码实现时，由于每个结点值都是唯一的，哈希表的键可以用结点值代替。此外，为避免在深度优先搜索时重复访问结点，递归时额外传入来源结点from，在递归前比较目标结点是否与来源结点相同，不同的情况下才进行递归。

**代码：**

class Solution {

private:

unordered\_map<int, TreeNode\*> parents; // 需要记录父节点

vector<int> ans;

public:

void findParents(TreeNode\* node) {

if (node->left != nullptr) {

parents[node->left->val] = node;

findParents(node->left);

}

if (node->right != nullptr) {

parents[node->right->val] = node;

findParents(node->right);

}

}

void findAns(TreeNode\* node, TreeNode\* from, int depth, int k) {

if (node == nullptr) {

return;

}

if (depth == k) {

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

return;

}

if (node->left != from) {

findAns(node->left, node, depth + 1, k);

}

if (node->right != from) {

findAns(node->right, node, depth + 1, k);

}

if (parents[node->val] != from) {

findAns(parents[node->val], node, depth + 1, k);

}

}

vector<int> distanceK(TreeNode\* root, TreeNode\* target, int k) {

// 从 root 出发 DFS，记录每个结点的父结点

findParents(root);

// 从 target 出发 DFS，寻找所有深度为 k 的结点

findAns(target, nullptr, 0, k);

return ans;

}

};

**复杂度分析：**

时间复杂度：O(n)，其中n是二叉树的结点个数。需要执行两次深度优先搜索，每次的时间复杂度均为O(n)。

空间复杂度： O(n)。记录父节点需要O(n)的空间，深度优先搜索需要O(n)的栈空间。