Day 17 / 100 :

Topic - Binary Tree

Given the root of a binary tree, the value of a target node target, and an 1 Problem statement: **All node distance K in binary tree** (Medium)
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: []

Solutions :

Approach 1 - BFS

If we just need to find the nodes at distance K from a target node in child nodes, then it is pretty easy. But here we need to find all the nodes at distance K which means that we need to search even in the parent nodes.

1. We take the help of hash maps to traverse the whole tree and store the child->parent reln.
2. This will help us in knowing the parent of every child. This is how we can traverse upward.
3. We use BFS to traverse the graph from target node to downwards and upward for a distance of K nodes. We can now easily traverse upwards (thanks to the hash above)We keep a visited map as well so that we dont traverse a node more than once.

Time Complexity: O(n)
Space Complexity: O(n)

```
class Solution {
public:

     // Recursive Function to get the "parents hash" where
child->parent relationship is kept
    void fill_parents(unordered_map <TreeNode*, TreeNode*>& parents,
TreeNode* node){
        if (node == NULL)
            return;
        // store the child->parent relationship and iterate for the
left child
        if(node->left != NULL){
            parents[node->left] = node;
            fill_parents(parents, node->left);
        }
        // store the child->parent relationship and iterate for the
right child
        if(node->right != NULL){
            parents[node->right] = node;
            fill_parents(parents, node->right);
        }
    }

    // Function to get all nodes which are a distance K from target
node.
    vector<int> distanceK(TreeNode* root, TreeNode* target, int K) {
        if (root == NULL)
            return {};

        unordered_map <TreeNode*, TreeNode*> parents;
        // Get the child->parent relationship in the "parents" map.
        fill_parents(parents, root);

        // Hash to store the nodes which are already visited so we do
not fall in an infinite loop
        unordered_map<TreeNode*, bool> visited;
        // Queue for BFS
        queue<TreeNode*> q;

        int currentLevel = 0;
        q.push(target);
        visited[target] = true;

        // Till the queue is empty
        while(!q.empty()){

           // If we are already at level 'K', then we break since
queue will have our distance K nodes
           // otherwise increase the level
           if (currentLevel == K){
                break;
            }
            currentLevel++;

           // Get the size of the queue.
            int size = q.size();

            // For the size number of elements in the queue, keep
trying to visit them and push in the queue
            // This is needed because this will tell us the number of
nodes which are at the same level.
            // This is really important to understand that when the
control comes here the number of elements
            // in the queue are all at the same level. While the loop
runs, it may increase but the 'size'
            // variable is still the same and it will stop. The new
nodes which are added are all at the same level.
            // NOTE:  Putting NULL marker after the level ends will
not work here.
            for(int i=0; i<size; i++) {
                TreeNode* front = q.front();
                visited[front] = true;
                q.pop();
                if (front->left && !visited[front->left]){
                    q.push(front->left);
                    visited[front->left] = true;
                }

                if (front->right && !visited[front->right]){
                    q.push(front->right);
                    visited[front->right] = true;
                }

                if (parents[front] && !visited[parents[front]]){
                    q.push(parents[front]);
                    visited[parents[front]] = true;
                }
            }
        }
        vector<int> res;
        while(!q.empty()){
            TreeNode *f = q.front();
            q.pop();
            res.push_back(f->val);
        }
        return res;
    }
};
```

2 Problem statement: Integer to Roman (Medium)

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

I can be placed before V (5) and X (10) to make 4 and 9.
X can be placed before L (50) and C (100) to make 40 and 90.
C can be placed before D (500) and M (1000) to make 400 and 900.
Given an integer, convert it to a roman numeral.

Example 1:
Input: num = 3
Output: "III"
Explanation: 3 is represented as 3 ones.

Example 2:
Input: num = 58
Output: "LVIII"
Explanation: L = 50, V = 5, III = 3.

Example 3:
Input: num = 1994
Output: "MCMXCIV"
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Solutions :

Approach 1 - Hashmap + Maths

Intuition:
We can solve this problem using strings + Hash Table + Math.

Approach:
We can easily understand the approach by seeing the code, which is easy to understand with comments. Roman numeral
Complexity:
Time complexity:

Time Complexity : O(1), The maximum length of the string(s) can be 15 (as per the Constraint), therefore, the worst case time complexity can be O (15) or O(1).

Space complexity:
Space Complexity : O(1), We are using unordered_map(map) to store the Roman symbols and their corresponding integer values, but there are only 7 symbols hence the worst case space complexity can be O(7) which is equivalent to O(1).

```cpp
class Solution {
public:
    int romanToInt(string s) {
        unordered_map<char, int> map;

        map['I'] = 1;
        map['V'] = 5;
        map['X'] = 10;
        map['L'] = 50;
        map['C'] = 100;
        map['D'] = 500;
        map['M'] = 1000;

        int ans = 0;

        for(int i=0; i<s.length(); i++){
            if(map[s[i]] < map[s[i+1]]){
                ans -= map[s[i]];
            }
            else{
                ans += map[s[i]];
            }
        }
        return ans;
    }
};
```