TREES

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Invert Binary Tree**			
	Given the root of a binary tree, invert the tree, and return its root.	- Main Idea: **Recursion** - At each node, store the current node's left pointer in a temporary variable - Recurlively call the function on the left child: `root->left = F(root->right)` - Recursively call the function on the right child: `root->right = F(temp)`	O(N) Worst case: Skewed Tree	O(N) Worst case: Skewed Tree
2	**Maximum Depth of Binary Tree**			
	Given the root of a binary tree, return its maximum depth. (N: number of nodes in the tree)	- Main Idea: **Recursion** - At every node --> **max_depth = max(l_depth, r_depth) + 1** - l_depth = max depth of its left children - r_depth = max depth of its right children	O(N) Worst case: Skewed Tree	O(N) Worst case: Skewed Tree
3	**Diameter of a Binary Tree**			
	Given root of binary tree, return length of diameter of tree (longest path b/w any 2 nodes)	- **Recursive** 1) Recursively calculate the depth of the left subtree (l_depth) and the depth of the right subtree (r_depth). - The depth of the tree rooted at the current node root is `1 + max(l_depth, r_depth)` 2) The diameter of the tree passing through the current node is calculated as `l_depth + r_depth`	O(N) Worst case: Skewed Tree	O(N) Worst case: Skewed Tree
4	**Balanced Binary Tree**			
	Given binary tree, determine if it's height-balanced (all left & right subtrees height diff <= 1)	**Approach_1** - Go over each node and call height() function of left and right child - Check whether `abs(l_depth - r_depth) <= 1`	O(N^2)	O(N)
		Approach_2 (Each node is visited only once) - Instead of calling height() function explicitly for each node, return the height of the current node in recursion. - When the subtree rooted at current node is balanced, the function height() returns a non-negative value as the height. - Otherwise it returns -1.	O(N)	O(h) -- O(N) Worst Case
5	**Same Tree**			
	Given roots of 2 binary trees, check if they're the same or not (same structure & values)	- Base case: 1) Both p and q are NULL, indicating that the current nodes in both trees are empty. --> `Return True` 2) If only one of p and q is NULL while the other is not, it means that the trees are not identical. --> `Return False` - If both p and q are not NULL, compare their values. If the values are different, the trees are not identical -> `Return False` - Recursively check the same for left subtrees `(p->left and q->left)` and the right subtrees `(p->right and q->right)`	O(N)	O(h) -- O(N) Worst Case
6	**Subtree of Another Tree**			
	Given the roots of 2 binary trees, return true if a tree has a subtree of the other tree N: no. of nodes in the main tree M: no. of nodes in the subtree	- For each node 'x' in the tree rooted at 'root' - `if(x->val == subRoot->val)` --> Check if subtree rooted at 'x' is same as the tree rooted at 'subRoot'	O(N*M)	O(max(M, N))
7	**Lowest Common Ancestor of a Binary Search Tree**			
	Given a binary search tree (BST), find the lowest common ancestor of 2 given nodes in the BST	- Use **BST property** 1) If both nodes p and q have values **greater** than the root --> LCA is located in the **right** subtree of the root. 2) If both nodes p and q have values **less** than the root --> LCA is located in the **left** subtree of the root. 3) If one of them is gretaer than root and the other is less than the root (i.e p and q lies on opposite side of the root) OR If one of them is equal to the root --> LCA is root itself.	O(h) -- O(N) Worst Case	O(h) -- O(N) Worst Case
8	**Binary Tree (Level Order Traversal)**			
	Given root of binary tree, return level order traversal of its nodes	- Use **Queue** - At each level, push left & right nodes if they exist to queue	O(N)	O(N)

TREES

No.	Problem Statement	Solution	Time complexity	Space complexity
9	**Binary Tree (Right Side View)**			
	Given root of binary tree, return values that can only be seen from the right side	- **Level order traversal** - Push only last node of the current level	O(N)	O(N)
10	**Count Good Nodes in Binary Tree**			
	Given binary tree, a node X in the tree is named "good" if in the path from root to X there are no nodes with a value greater than X. Return # of "good" nodes	- Maintain **maximum value seen so far** on a path from root to 'X' - If current node >= max_so_far Then X is "good" node *(Note: just comparing the current value in the path with previous value will not work* *Example: [3, 6, 7, 4, Nil, Nil, 8, 5] --> Above approach will count '5' as good node which is not the case)*	O(N)	O(N)
11	**Validate Binary Search Tree**			

	Given the root of a binary tree, determine if it is a valid binary search tree (BST).	- Approach_1: Keep track of **min & max on current path** for each node
		- Idea: **node->val > curr_path_min && node->val < curr_path_max**
		`if(root->val > curr_path_min && root->val < curr_path_max){`
		`if(helper(root->left, curr_path_min, root->val) && helper(root->right, root->val, curr_path_max)) return true;`
		`}`
			O(N)	O(h) -- O(N)
				Worst Case

		- Approach_2: **In-order Traversal** produces a sequence of values in sorted manner. - Check whether each node's value is greater than the previously visited node's value to determine if it's a valid BST.	O(N)	O(h) -- O(N) Worst Case
12	**Kth Smallest Element in a BST**			
	Given root of BST & int K, return Kth smallest value (1-indexed) in the tree	- **Inorder Traversal** produces a sequence of values in sorted manner. - First traverse the left subtree - **After the left subtree traversal, k is decremented by 1.** - If k becomes 0 after decrementing, it indicates that the current node's value is the kth smallest - If k is not yet 0 continue traversing the right subtree	O(N) If asked to find the greatest element	O(h) -- O(N) Worst Case
13	**Construct Binary Tree from Preorder & Inorder Traversal**			
	Given 2 integer arrays preorder & inorder, construct & return the root of the binary tree	- Approach_1: Recursive: **Preorder for values & Inorder for positions** - 1) *index* keeps track of root node in current subtree i.e preorder[index] - 2) Find the corresponding root in the inorder list - 3) Split the inorder list into two parts: nodes before this root (left subtree) and nodes after this root (right subtree)	O(N^2) Worst case Skewed Tree	O(N)
		- Approach_2: Optimize step 2 of Approach_1 using **unordered_map** - map: {node->value, index of the node in the inorder vector} - Allows constant time access	O(N)	O(N)
14	**Binary Tree Maximum Path Sum**			
	Given the root of a binary tree, return the maximum path sum of any non-empty path.	- For **each** node X in the binary tree, recursively calculate the maximum path sum that **goes through** X - l_sum = Maximum path sum that goes through the left child of the current node - r_sum = Maximum path sum that goes through the right child of the current node - Update ans with maximum path sum going through the current node `ans = max(ans, l_sum+r_sum+curr->val)` - For each node return the maximum path sum till the current node (either from left child or right child)	O(N)	O(N)
15	**Serialize and Deserialize Binary Tree**			
	Given the root of a binary tree, Serialize the tree into a string and Deserialize this string into the original tree structure. *(Serialization: Process of converting a data structure into a sequence of bits so that it can be stored in a file or memory buffer)*	- Use **ostringstream & istringstream** to concisely handle negatives, nulls, etc. + **Preorder Traversal**	O(N)	O(N)