Trees :

- There are three types of DFS traversals in tree → pre, in and post
- and only one type of BFS → level order traversal

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if (root == nullptr) {
            return ans;
        }

        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            // Get the size of the current level
            int size = q.size();
            vector<int> level;

            for (int i = 0; i < size; i++) {
                TreeNode* node = q.front();
                q.pop();
                level.push_back(node->val);

                // Enqueue the child nodes if they exist
                if (node->left != nullptr) {
                    q.push(node->left);
                }
                if (node->right != nullptr) {
                    q.push(node->right);
                }
            }
            ans.push_back(level);
        }

        return ans;
    }
};
```

Height / Max Depth of BT

```
int maxDepth(TreeNode* root) {
    if (root == nullptr) return 0;
```

```
    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);
    return max(leftDepth, rightDepth) + 1;
}
```

Check if tree is balanced or not

```
class Solution {
public:
    bool isBalanced(Node* root) {
        return dfsHeight(root) != -1;
    }

    int dfsHeight(Node* root) {
        if (root == NULL) return 0;

        int leftHeight = dfsHeight(root->left);
        if (leftHeight == -1)
            return -1;

        int rightHeight = dfsHeight(root->right);
        if (rightHeight == -1)
            return -1;

        if (abs(leftHeight - rightHeight) > 1)
            return -1;
        return max(leftHeight, rightHeight) + 1;
    }
};
```

Diameter of a tree

```
class Solution {
public:
    int diameterOfBinaryTree(Node* root) {
        int diameter = 0;
        height(root, diameter);

        return diameter;
    }

private:
    int height(Node* node, int& diameter) {
        if (!node) {
            return 0;
        }
```

```
        int lh = height(node->left, diameter);
        int rh = height(node->right, diameter);

        diameter = max(diameter, lh + rh);

        return 1 + max(lh, rh);
    }
};
```

Max path sum

```
class Solution {
public:
    int findMaxPathSum(Node* root, int &maxi) {
        if (root == nullptr) {
            return 0;
        }

        int leftMaxPath = max(0, findMaxPathSum(root->left, maxi));
        // to make sure we don't take any -ve sum ahead
        int rightMaxPath = max(0, findMaxPathSum(root->right, maxi));

        maxi = max(maxi, leftMaxPath + rightMaxPath + root->data);

        return max(leftMaxPath, rightMaxPath) + root->data;
    }

    int maxPathSum(Node* root) {
        int maxi = INT_MIN;
        findMaxPathSum(root, maxi);

        return maxi;
    }
};
```

Check if two trees are identical

```
class Solution {
public:
    bool isIdentical(Node* node1, Node* node2){

        if( node1== NULL || node2==NULL){
        // this ensure same structure
            return node1 == node2;
        }

        return ((node1->data == node2->data)
```

```
        // this ensures same values in them
            && isIdentical(node1->left, node2->left)
                && isIdentical(node1->right, node2->right));
    }
};
```

Zig-Zag / Spiral Traversal

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (root == nullptr) return result;

    queue<TreeNode*> q;
    q.push(root);
    bool leftToRight = true;

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> level;

        for (int i = 0; i < levelSize; ++i) {
            TreeNode* node = q.front();
            q.pop();
            level.push_back(node->val);

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        // Reverse the level if we're going right to left
        if (!leftToRight) {
            reverse(level.begin(), level.end());
        }

        result.push_back(level);
        leftToRight = !leftToRight;  // Toggle direction
    }

    return result;
}
```

Boundary Traversal :

```
// Utility function to check if a node is a leaf.
bool isLeaf(Node* root) {
    return !root->left && !root->right;
}
```

```
// Function to add the left boundary nodes (excluding leaf nodes).
void addLeftBoundary(Node* root, vector<int>& res) {
    Node* cur = root->left;
    while (cur) {
        if (!isLeaf(cur)) res.push_back(cur->data);
        if (cur->left) cur = cur->left;
        else cur = cur->right;
    }
}

// Function to add the right boundary nodes (excluding leaf nodes).
void addRightBoundary(Node* root, vector<int>& res) {
    Node* cur = root->right;
    vector<int> tmp; // Temporary vector to store nodes in reverse order.
    while (cur) {
        if (!isLeaf(cur)) tmp.push_back(cur->data);
        if (cur->right) cur = cur->right;
        else cur = cur->left;
    }
    // Add the right boundary nodes in reverse order.
    for (int i = tmp.size() - 1; i >= 0; --i) {
        res.push_back(tmp[i]);
    }
}

// Function to add all the leaf nodes.
void addLeaves(Node* root, vector<int>& res) {
// any traversal is fine because we are printing
// left subtree before right subtree
    if (isLeaf(root)) {
        res.push_back(root->data);
        return;
    }
    if (root->left) addLeaves(root->left, res);
    if (root->right) addLeaves(root->right, res);
}

// Main function to perform boundary traversal.
vector<int> printBoundary(Node* root) {
    vector<int> res;
    if (!root) return res; // Return empty vector if the tree is empty.

    // Add the root if it’s not a leaf node.
    if (!isLeaf(root)) res.push_back(root->data);

    // Add the left boundary.
    addLeftBoundary(root, res);
```

```
    // Add all the leaf nodes.
    addLeaves(root, res);

    // Add the right boundary.
    addRightBoundary(root, res);

    return res;
}
```

Vertical Order Traversal

```
class Solution {
public:
    // Function to perform vertical order traversal
    // and return a 2D vector of node values
    // if collision occurs return them in a sorted order,
    // considering duplicacy is allowed
    vector<vector<int>> findVertical(Node* root){

        // vertical lvl, horizontal lvl , node value
        map<int, map<int, multiset<int>>> nodes;

        // Queue for BFS traversal, each
        // element is a pair containing node
        // and its vertical and level information
        queue<pair<Node*, pair<int, int>>> todo;

        // Push the root node with initial vertical
        // and level values (0, 0)
        todo.push({root, {0, 0}});

        while(!todo.empty()){
            auto p = todo.front();
            todo.pop();
            Node* temp = p.first;

            // Extract the vertical and level information
            // x -> vertical
            int x = p.second.first;
            // y -> level
            int y = p.second.second;

            // Insert the node value into the
            // corresponding vertical and level
            // in the map
            nodes[x][y].insert(temp->data);
```

```
            // Process left child
            if(temp->left){
                todo.push({
                    temp->left,
                    {
                        // Move left in
                        // terms of vertical
                        x-1,
                        // Move down in
                        // terms of level
                        y+1
                    }
                });
            }

            // Process right child
            if(temp->right){
                todo.push({
                    temp->right,
                    {
                        // Move right in
                        // terms of vertical
                        x+1,
                        // Move down in
                        // terms of level
                        y+1
                    }
                });
            }
        }

        // Prepare the final result vector
        // by combining values from the map
        vector<vector<int>> ans;
        for(auto p: nodes){
            vector<int> col;
            for(auto q: p.second){
                // Insert node values
                // into the column vector
                col.insert(col.end(), q.second.begin(), q.second.end());
            }
            // Add the column vector
            // to the final result
            ans.push_back(col);
        }
        return ans;
```

```
    }
};
```

Top view

```
void topView(Node* root) {
    if (root == nullptr) return;

    // Map to store the first node at each horizontal distance.
    map<int, int> topNodes; // no duplicates allowed first come first served

    // It stores pairs of (node, horizontal distance).
    queue<pair<Node*, int>> q;
    q.push({root, 0});

    while (!q.empty()) {
        auto it = q.front();
        q.pop();

        Node* node = it.first;
        int horizontalDist = it.second;

        // If the horizontal distance is encountered for
        // the first time, add the node.
        if (topNodes.find(horizontalDist) == topNodes.end()) {
            topNodes[horizontalDist] = node->data;
        }

        // Move to the left and right children with
        // updated horizontal distances.
        if (node->left) {
            q.push({node->left, horizontalDist - 1});
        }
        if (node->right) {
            q.push({node->right, horizontalDist + 1});
        }
    }

    // Print the top view.
    for (auto it : topNodes) {
        cout << it.second << " ";
    }
    cout << endl;
}
```

Bottom view : If collision occurs take the rightmost guy

- same as top view but here we need to keep overwriting and what about collision, no extra handling as left is traversed first and then right

```
vector<int> bottomView(Node* root) {
    vector<int> ans;
    if (root == nullptr) return ans;

    // Map to store the last node at each horizontal distance (line).
    map<int, int> mpp; // why not unordered_map ?
    // => we need sorting, left most level first and then right

    // It stores pairs of (node, horizontal distance).
    queue<pair<Node*, int>> q;
    q.push({root, 0});

    while (!q.empty()) {
        auto it = q.front();
        q.pop();

        Node* node = it.first;
        int line = it.second;

        // Update the map with the current node for the given horizontal distance.
        mpp[line] = node->data;

        // Move to the left child with horizontal distance - 1.
        if (node->left != nullptr) {
            q.push({node->left, line - 1});
        }
        // Move to the right child with horizontal distance + 1.
        if (node->right != nullptr) {
            q.push({node->right, line + 1});
        }
    }

    // Store the final bottom view in the result vector.
    for (auto it : mpp) {
        ans.push_back(it.second);
    }

    return ans;
}
```

- Given a Binary Tree, return the left view of it

```
void recursionLeft(Node* root, int level, vector<int>& res){
        if(root == NULL){
```

```
            return;
        }

        // Check if the size of the result vector
        // is equal to the current level as each level
        // can have only one node visible
        if(res.size() == level){
            res.push_back(root->data);
        }

        recursionLeft(root->left, level + 1, res);
        // First give priority to left if we want right view
        // then give it second priority
        recursionLeft(root->right, level + 1, res);
    }

vector<int> leftsideView(Node* root)
        vector<int> res;
        recursionLeft(root, 0, res);

        return res;
    }
```

Is symmetric ?

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        // If the tree is empty, it's symmetric.
        return root == NULL || isSymmetricHelp(root->left, root->right);
    }

private:
    bool isSymmetricHelp(TreeNode* left, TreeNode* right) {
        // If both subtrees are null, they are symmetric.
        if (left == NULL || right == NULL) return left == right; //**

        // If the values of the nodes don't match, the tree is not symmetric.
        if (left->val != right->val) return false;

        // Recursively check the left subtree of the left child
        // with the right subtree of the right child,
        // and the right subtree of the left child with the
        // left subtree of the right child.
        return isSymmetricHelp(left->left, right->right) &&
               isSymmetricHelp(left->right, right->left);
```

```
    }
};
```

Print Path from root to Node

```
class Solution {
public:
    bool getPath(TreeNode* root, vector<int> &arr, int x) {
        // Base case: if the node is NULL, return false
        if (!root) return false;

        // Push the current node's value to the path
        arr.push_back(root->val);

        // If the current node's value is the target value, return true
        if (root->val == x) return true;

        // Recur for the left or right subtree
        if (getPath(root->left, arr, x) ||
        getPath(root->right, arr, x)) return true;

        // If the target value is not found in either subtree, backtrack
        arr.pop_back();
        return false;
    }

    vector<int> solve(TreeNode* A, int B) {
        vector<int> arr;
        // If the tree is empty, return an empty vector
        if (A == NULL) return arr;

        // Get the path from the root to the node with value B
        getPath(A, arr, B);
        return arr;
    }
};
```

Lowest Common Ancestor

Method 1 : we find paths of both roots, then compare both and find the last similar guy

Method 2 :

- Case 1 : If we find a node return it else return NULL
- Case 2 : If a node receives a node and NULL return node
- Case 3 : If a node receives two non-nulls return the current node

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // Base case: if root is NULL or root is either p or q, return root
        if (root == NULL || root == p || root == q) {
            return root;
        }

        // Recursive calls for left and right subtrees
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        // If one subtree is NULL, return the non-NULL subtree.
        // Here we are not checking root but the return values of children
        if (left == NULL) {
            return right;
        } else if (right == NULL) {
            return left;
        } else {
            // If both left and right are non-NULL, root is the LCA
            return root;
        }
    }
};
```

Given a root, target and distance k. return the root values which are at a distance of k from target

```
class Solution {

    void markParents(TreeNode *root, unordered_map<TreeNode*,
        TreeNode*> &parent_track, TreeNode* target) {

        queue<TreeNode*> queue;
        queue.push(root);

        while (!queue.empty()) {
            TreeNode* current = queue.front();
            queue.pop();

            if (current->left) {
                parent_track[current->left] = current;
                queue.push(current->left);
            }

            if (current->right) {
```

```
                parent_track[current->right] = current;
                queue.push(current->right);
            }
        }
    }

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

        unordered_map<TreeNode*, TreeNode*> parent_track; //**
        markParents(root, parent_track, target);
        // Keeps track of visited nodes
        unordered_map<TreeNode*, bool> visited;
        queue<TreeNode*> queue;
        queue.push(target);
        // Tracks the current level
        // while traversing the tree
        int curr_level = 0;

        while (!queue.empty()) {
            int size = queue.size(); // simultaneous multiple BFS
            if (curr_level++ == k) {
                // Break if the current level
                // matches the required distance (k)
                break;
            }

            for (int i = 0; i < size; i++) {
                TreeNode* current = queue.front();
                queue.pop();

                if (current->left && !visited[current->left]) {
                    queue.push(current->left);
                    visited[current->left] = true;
                }

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

                if (parent_track[current] && !visited[parent_track[current]]) {
                    queue.push(parent_track[current]);
                    visited[parent_track[current]] = true;
                }
            }
        }
```

```
        // Stores nodes at distance k from the target
        vector<int> result;
        while (!queue.empty()) {
            // Extract nodes at distance k from the queue
            TreeNode* current = queue.front();
            queue.pop();
            // Store node values in the result vector
            result.push_back(current->val);
        }

        return result;
    }
};
```

Given a BinaryTree {has data, leftchild, rightchild}. One of its node is set on fire, find the minimum time taken for the entire tree to burn down.

- A simple BFS from the target would be sufficient to solve this problem
- But we can’t backtrack to parents
- So, we will keep a track of parent node of each node

```
class Solution {
  private:
    void mapParents(Node* root, unordered_map<Node*, Node*> &parentMap,
                    Node* &targetNode, int target)
    {
        queue<Node*> q;
        q.push(root);

        while(!q.empty()) {
            Node* current = q.front();
            q.pop();

            if(current->data == target) { // Sub-Task
                targetNode = current;
            }

            if(current->left) {
                parentMap[current->left] = current;
                q.push(current->left);
            }

            if(current->right) {
                parentMap[current->right] = current;
                q.push(current->right);
            }
        }
```

```
    }
  public:
    int minTime(Node* root, int target)
    {
        if(!root) return 0;

        unordered_map<Node*, Node*> parentMap;
        Node* targetNode = NULL;

        mapParents(root, parentMap, targetNode, target);

        queue<Node*> q;
        set<Node*> visited;

        q.push(targetNode);
        visited.insert(targetNode);
        int time = 0;


        while(!q.empty()) {
            int size = q.size();
            bool burned = false;

            for(int i = 0; i < size; i++) {
            // Why are we doing this ?
            // {SIMULTANEOUS BFS} other acc to this logic it would take more time
                Node* current = q.front();
                q.pop();

                // Burn left child
                if (current->left &&
                visited.find(current->left) == visited.end()) {
                    burned = true;
                    visited.insert(current->left);
                    q.push(current->left);
                }

                // Burn right child
                if (current->right &&
                visited.find(current->right) == visited.end()) {
                    burned = true;
                    visited.insert(current->right);
                    q.push(current->right);
                }

                // Burn parent
                if (parentMap[current] &&
                visited.find(parentMap[current]) == visited.end()) {
```

```
                // For the root, map will return NULL {0 if it were integer type}
                    burned = true;
                    visited.insert(parentMap[current]);
                    q.push(parentMap[current]);
                }
            }

            // Increment time if any node was burned in this step
            if (burned) {
                time++;
            }
        }

        return time;
    }
};
```

Count total Nodes in a COMPLETE Binary Tree

```
class Solution {
public:
    int countNodes(TreeNode* root) {
        if (root == NULL) {
            return 0;
        }

        // Find the left height and
        // right height of the tree
        int lh = findHeightLeft(root);
        int rh = findHeightRight(root);

        // Check if the last level
        // is completely filled
        if (lh == rh) {
            // If so, use the formula for
            // total nodes in a perfect
            // binary tree ie. 2^h - 1
            return (1 << lh) - 1;
        }

        // If the last level is not completely
        // filled, recursively count nodes in
        // left and right subtrees
        return 1 + countNodes(root->left) + countNodes(root->right);
    }

    int findHeightLeft(TreeNode* node) {
```

```
        int height = 0;
        while (node) {
            height++;
            node = node->left;
        }
        return height;
    }

    int findHeightRight(TreeNode* node) {
        int height = 0;
        while (node) {
            height++;
            node = node->right;
        }
        return height;
    }
};
```

Given a Binary Tree convert it into a string. Now delete the Tree. Reconstruct the tree using the string and return the new root.

```
class Solution {
public:
    // Encodes the tree into a single string
    string serialize(TreeNode* root) {
        if (!root) {
            return "";
        }

        string s = "";
        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            TreeNode* curNode = q.front();
            q.pop();

            // Check if the current node is
            // null and append "#" to the string
            if (curNode == nullptr) {
                s += "#,";
            } else {
                // Append the value of the
                // current node to the string
                s += to_string(curNode->val) + ",";
                // Push the left and right children
                // to the queue for further traversal
```

```
                q.push(curNode->left);
                q.push(curNode->right);
            }
        }

        return s;
    }

    // Decode the encoded
    // data to a tree
    TreeNode* deserialize(string data) {
        if (data.empty()) {
            return nullptr;
        }

        // Use a stringstream to
        // tokenize the serialized data
        stringstream s(data);
        string str;
        // Read the root value
        // from the serialized data
        getline(s, str, ',');
        TreeNode* root = new TreeNode(stoi(str));

        queue<TreeNode*> q;
        q.push(root);

        // Perform level-order traversal
        // to reconstruct the tree
        while (!q.empty()) {
            TreeNode* node = q.front();
            q.pop();

            // Read the value of the left
            // child from the serialized data
            getline(s, str, ',');
            // If the value is not "#", create a new
            // left child and push it to the queue
            if (str != "#") {
                TreeNode* leftNode = new TreeNode(stoi(str));
                node->left = leftNode;
                q.push(leftNode);
            }

            // Read the value of the right child
            // from the serialized data
            getline(s, str, ',');
            // If the value is not "#", create a
```

```
            // new right child and push it to the queue
            if (str != "#") {
                TreeNode* rightNode = new TreeNode(stoi(str));
                node->right = rightNode;
                q.push(rightNode);
            }
        }

        return root;
    }
};
```

Morris Traversal is a method to traverse a binary tree without using additional space like a stack or recursion. It is a space-efficient algorithm because it uses no extra space apart from a few pointers. The idea is to temporarily modify the tree structure during the traversal and restore it afterward.

1. **If the left child is NULL:**
2. If the current node does not have a left child, print the current node's data and move to the right child.
   Explanation: There’s no left subtree to explore, so you simply move to the right child.
3. **If the left child is not NULL:**
4. Find the inorder predecessor of the current node in the left subtree. The inorder predecessor is the rightmost node in the left subtree.
5. Subcase 1: If the right child of the predecessor is NULL:
   Make the current node the right child of its inorder predecessor (temporary link) and move the current node to its left child.
   Explanation: This temporary link allows you to return to the current node after exploring the left subtree.
6. Subcase 2: If the right child of the predecessor is not NULL:
   This means the left subtree has already been visited, so remove the temporary link (restore the original tree), print the current node's data, and move to the right child.
   Explanation: You’ve finished visiting the left subtree, so you can now process the current node and move to the right subtree.

```
void morrisTraversal(Node* root) {
    Node* current = root;

    while (current != NULL) {
        if (current->left == NULL) { // case 1
            cout << current->data << " ";
            current = current->right;
        } else { // case 2
            Node* predecessor = current->left;
```

```
            while (predecessor->right != NULL && predecessor->right != current) {
                predecessor = predecessor->right;
            }

            if (predecessor->right == NULL) { // sub-case 1
                predecessor->right = current;
                current = current->left;
            } else { // subcase 2
                predecessor->right = NULL;
                cout << current->data << " ";
                current = current->right;
            }
        }
    }
}
```

In a BST, apart from the tree itself, the left and right subtrees are BST’s aswell.

```
// search in BST
struct Node* search(struct Node* root, int key) {
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->data == key) // Takin care of NULL is imp
        return root;

    // Key is greater than root's data
    if (root->data < key)
        return search(root->right, key);

    // Key is smaller than root's data
    return search(root->left, key);
}
```

Find ceil in BST

```
int findCeil(Node* root, int input) {
    int ans = -1;

    while (root != NULL) {
        // If the current node's value is equal to the input, that's the ceil.
        if (root->data == input) {
            return root->data; // no need to further traverse
        }

        // If the current node's value is greater than the input, it could be the cei
        if (root->data > input) {
            ans = root->data; // Update the answer
            root = root->left;
```

```
            // Move to the left subtree to find a smaller candidate
        } else {
            // If the current node's value is less than the input,
            // move to the right subtree.
            root = root->right;
        }
    }

    return ans;
}
```

Insertion in a BST

```
TreeNode<int>* insertionInBST(TreeNode<int>* root, int val) {
    // If the tree is empty, return a new node
    if (!root) return new TreeNode<int>(val);

    TreeNode<int>* curr = root;
    while (true) {
        if (val <= curr->data) {
            if (curr->left) {
                curr = curr->left;
            } else {
                curr->left = new TreeNode<int>(val);
                break;
            }
        } else {
            if (curr->right) {
                curr = curr->right;
            } else {
                curr->right = new TreeNode<int>(val);
                break;
            }
        }
    }

    return root;
}
```

Given a BST and a key, if the key exists in the binary tree delete it else do nothing. return the root of the tree.

```
class Solution {
private:
    // Helper function to find the minimum value in the right subtree.
    int find_min(TreeNode* node) {
        while (node->left) {
```

```
            node = node->left;
        }
        return node->val;
    }

public:
    TreeNode* deleteNode(TreeNode* root, int key) {

        // Only reached when either tree is empty or
        // the key doesn't exists in the tree
        if (!root) return root;

        // Navigate the tree based on the key value
        if (key < root->val) {
            root->left = deleteNode(root->left, key);
        }
        else if (key > root->val) {
            root->right = deleteNode(root->right, key);
        }
        else {
            // Node found and now we handle the three cases:

            // Case 1: Node has no left child
            if (!root->left) {
                TreeNode* temp = root->right;
                delete root;
                return temp;
            }
            // Case 2: Node has no right child
            else if (!root->right) {
                TreeNode* temp = root->left;
                delete root;
                return temp;
            }
            // Case 3: Node has two children
            else {
                // Get the in-order successor (smallest in the right subtree)
                int minValue = find_min(root->right);
                root->val = minValue;
                // we are not swapping we have just replaced the node value
                // with inorder successor

                // Delete the in-order successor
                root->right = deleteNode(root->right, minValue);
            }
        }

        return root;
```

```
    }
};
```

Kth smallest element in BST

- inorder traversal of a BST is always sorted

```
class Solution {
private:
    int count = 0;  // Counter to track the number of nodes visited
    int result = -1; // To store the K-th smallest element

    // Helper function to perform in-order traversal
    void inorder(TreeNode* root, int k) {
        if (!root) return;

        // Traverse the left subtree
        inorder(root->left, k);

        // Increment the counter when visiting a node
        count++;

        // If the counter equals k, we have found the K-th smallest element
        if (count == k) {
            result = root->val;
            return;
        }

        // Traverse the right subtree
        inorder(root->right, k);
    }

public:
    int kthSmallest(TreeNode* root, int k) {
        inorder(root, k);
        return result;
    }
};
```

- If we want the kth largest element then modify k = n - k;

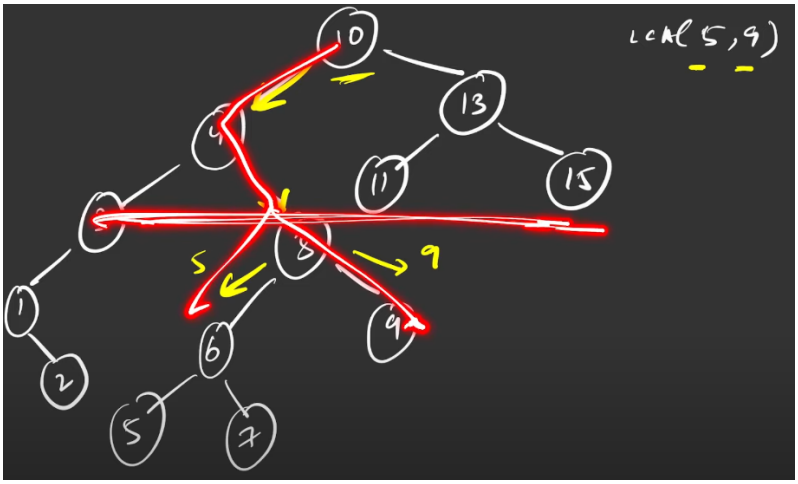
Given a BST validate if it is a valid BST

```
class Solution {
private:
    bool isIt(TreeNode* root, long long left_bound, long long right_bound) {
        if (!root) return true;
        if (root->val <= left_bound || root->val >= right_bound) {
```

```
            return false;
        }
        return isIt(root->right, root->val, right_bound) &&
               isIt(root->left, left_bound, root->val);
    }
public:
    bool isValidBST(TreeNode* root) {
        return isIt(root, LLONG_MIN, LLONG_MAX);
    }
};
```

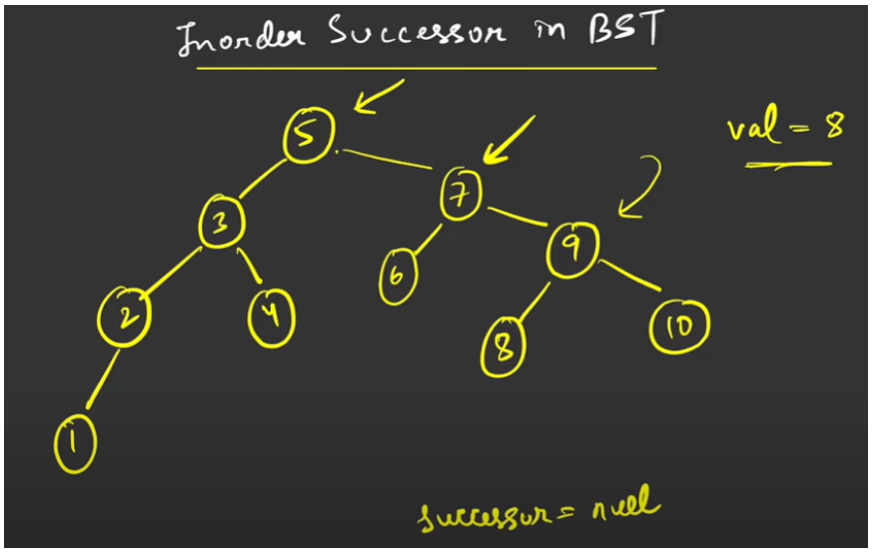
- Why are we taking long long ? if INT_MAX or INT_MIN then when we check ≤ or ≥ case it is not a duplicacy!

LCA in BST



- If both of them are on the right or left we can simply move.
- The instant they we are not able to say both of them don’t lie in left or right we got out LCA!

- It is not always the case that inorder successor or predecessor are children of the node.

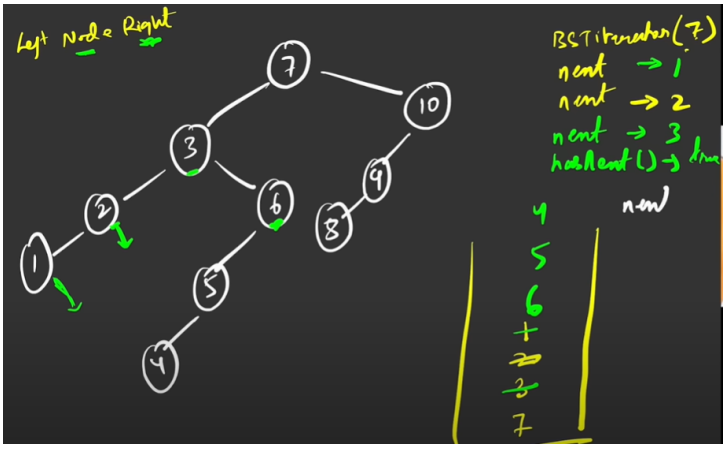


- For 8 here, IS = 9 (It's parent)

```
TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
    TreeNode* successor = NULL;

    while (root != NULL) {

        if (p->val >= root->val) {
            root = root->right;
        } else {
            successor = root;
            root = root->left;
        }
    }

    return successor;
}
```

BST iterator

```
class BSTIterator {
    stack<TreeNode *> myStack;
public:
    BSTIterator(TreeNode *root) {
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !myStack.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *tmpNode = myStack.top();
        myStack.pop();
        pushAll(tmpNode->right);
        return tmpNode->val;
    }

private:
    void pushAll(TreeNode *node) {
        for (; node != NULL; myStack.push(node), node = node->left);
    }
};
```

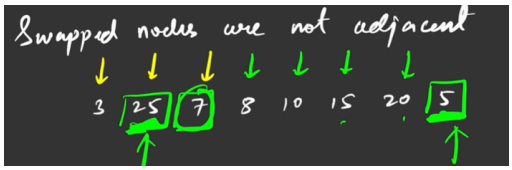


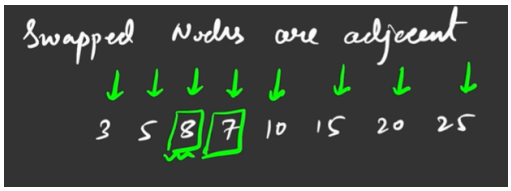
- So basically instead of storing entire inorder, jitna chahiye utna we are maintaining.

- two sum in BST. Two pointer approach on inorder ⇒ BST iterator next and before.
- Instead of making a complete different class, make changes in class itself

```
class BSTIterator {
    stack<TreeNode *> myStack;
    // reverse -> true -> before
    // reverse -> false -> next
    bool reverse = true;
public:
    BSTIterator(TreeNode *root, bool isReverse) {
        reverse = isReverse;
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !myStack.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *tmpNode = myStack.top();
        myStack.pop();
        if(!reverse) pushAll(tmpNode->right);
        else pushAll(tmpNode->left);
        return tmpNode->val;
    }

private:
    void pushAll(TreeNode *node) {
        for(;node != NULL; ) {
            myStack.push(node);
            if(reverse == true) {
                node = node->right;
            } else {
                node = node->left;
            }
        }
    }
};
```

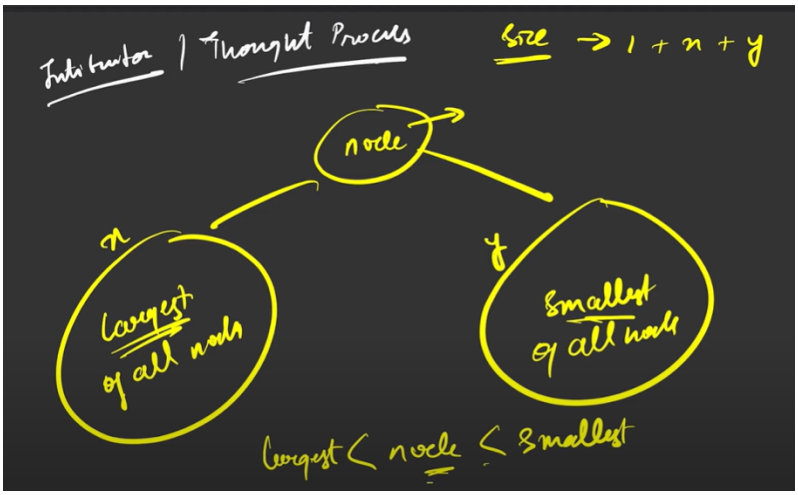
If two elements in a BST are swapped, recover it





```cpp
class Solution {
private:
    TreeNode* first;
    TreeNode* prev;
    TreeNode* middle;
    TreeNode* last;
private:
    void inorder(TreeNode* root) {
        if(root == NULL) return;

        inorder(root->left);

        if (prev != NULL && (root->val < prev->val))
        {

            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( first == NULL )
            {
                first = prev;
                middle = root;
            }

            // If this is second violation, mark this node as last
            else
                last = root;
        }

        // Mark this node as previous
        prev = root;
        inorder(root->right);
    }
public:
    void recoverTree(TreeNode* root) {
        first = middle = last = NULL;
        prev = new TreeNode(INT_MIN);
        inorder(root);
        if(first && last) swap(first->val, last->val);
        else if(first && middle) swap(first->val, middle->val);
    }
};
```

Given a BT return the size of largest BST in it



```cpp
class NodeValue {
public:
    int maxNode, minNode, maxSize;

    NodeValue(int minNode, int maxNode, int maxSize) {
        this->maxNode = maxNode;
        this->minNode = minNode;
        this->maxSize = maxSize;
    }
};

class Solution {
private:
    NodeValue largestBSTSubtreeHelper(TreeNode* root) {
        // An empty tree is a BST of size 0.
        if (!root) {
            return NodeValue(INT_MAX, INT_MIN, 0);
        }

        // Get values from left and right subtree of current tree.
        auto left = largestBSTSubtreeHelper(root->left);
        auto right = largestBSTSubtreeHelper(root->right);

        // Current node is greater than max in left AND smaller than min in right, it
        if (left.maxNode < root->val && root->val < right.minNode) {
            // It is a BST.
            return NodeValue(min(root->val, left.minNode), max(root->val, right.maxNod
                             left.maxSize + right.maxSize + 1);
        }

        // Otherwise, return [-inf, inf] so that parent can't be valid BST
        return NodeValue(INT_MIN, INT_MAX, max(left.maxSize, right.maxSize));
    }
    public:
    int largestBSTSubtree(TreeNode* root) {
        return largestBSTSubtreeHelper(root).maxSize;
    }
};
```