TREES

Many trees questions are solved using recursion. So first check if recursion can be used in the problem

1. Identify how to solve the problem -> **preorder** OR **inorder** OR **postorder** OR **level order**
2. Identify what strategy to use
    1. Using **states**
    2. recursion
    3. Node -> Root path method (when some operation/calculation needs to be performed on each node of node -> root path
    4. Diameter of btree approach (whan ans my be present in left subtree OR right subtree OR left- > root -> right
        Many **path problems** can be solved using this approach
    5. If there is any queston related to BST, first check if it can be solved using In order traversal

```cpp
Node* createTree(vector<int> vec){
    Node *root = new Node(vec[0]);

    pair<Node*, int> p = {root, 1};
    stack<pair<Node*, int>> s;
    s.push(p);

    for(int i=1; s.size()>0;){
        pair<Node*, int> temp = s.top();
        if(temp.second == 1){   //New node to left
            s.top().second++;
            if(vec[i] != NULL){
                temp.first->left = new Node(vec[i]);
                s.push({temp.first->left, 1});
            }
            i++;
        }
        else if(temp.second == 2){ //New node to right
            s.top().second++;
            if(vec[i] != NULL){
                temp.first->right = new Node(vec[i]);
                s.push({temp.first->right, 1});
            }
            i++;
        }
        else{
            s.pop();
        }
    }
    return root;
}
```

btree creation from preorder

```cpp
void preOrderTraversal(Node *root){
    stack<Node*> s;
    while(1){
        while(root){
            //Print root
            cout<<root->data<<" ";
            s.push(root);
            //Left sub-tree (If left subtree exists, add to stack)
            root = root->left;
        }
        if(s.empty())
            break;
        root = s.top();
        s.pop();
        //Right sub-tree(left subtree completed, Now move to right subtree)
        root = root->right;
    }
}
```

iterative preorder

```cpp
void inOrderTraversal(Node *root){
    stack<Node*> s;
    while(1){
        while(root){
            s.push(root);
            //Left sub-tree(Get left subtree and keep on adding to stack)
            root = root->left;
        }
        if(s.empty())
            break;
        root = s.top();
        s.pop();
        //Print root
        cout<<root->data<<" ";
        //Right sub-tree
        root = root->right;
    }
}
```

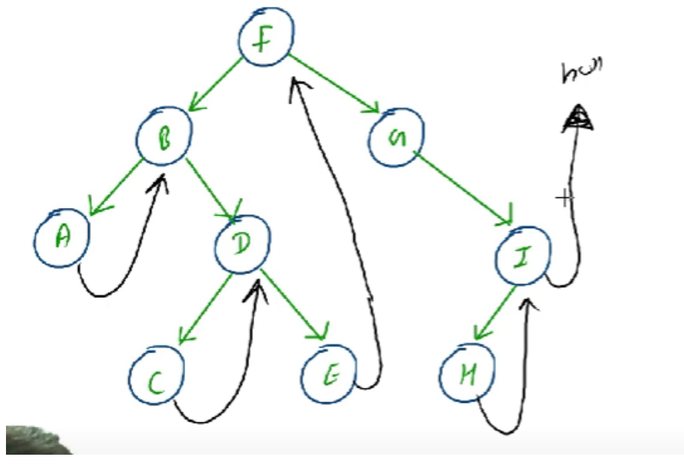
iterative inorder

```cpp
void postOrderTraversal2(Node *root){  //s : O(N)
    stack<Node*> s;
    Node *prev = NULL;
    do{
        while(root){
            //Left sub-tree
            s.push(root);
            root = root->left;
        }
        while(root == NULL && !s.empty()){
            root = s.top();
            //If right tree doesnot exist OR we are traversing up -> Print node and pop stack
            if(root->right == NULL || root->right == prev){
                //root->right wud be equal to prev only if the right child is
                //already visited that means we are traversing up the b-tree

                //Printing root
                cout<<root->data<<" ";
                prev = root;
                s.pop();
                root = NULL;
            }
            //If right sub tree exists, go to right sub tree
            //Right sub-tree
            else{
                root = root->right;
            }
        }
    }while(!s.empty());
}
```

iterative postorder

```cpp
void PreInPost(Node *root){
    // 1 => pre, state++, left
    // 2 => in , state++, right
    // 3 => post, pop
    string pre = "", in = "", post = "";
    stack < pair<Node*, int> > s;
    s.push({root, 1});
    while(!s.empty()){
        pair <Node*, int> temp = s.top();
        if(temp.second == 1){
            pre += to_string(temp.first->data) + " ";
            s.top().second++;
            if(temp.first->left != NULL)
                s.push({temp.first->left, 1});
        }
        else if(temp.second == 2){
            in += to_string(temp.first->data) + " ";
            s.top().second++;
            if(temp.first->right != NULL)
                s.push({temp.first->right, 1});
        }
        else{
            post += to_string(temp.first->data) + " ";
            s.pop();
        }
    }
    cout<<"Pre : "<<pre<<endl;
    cout<<"In : "<<in<<endl;
    cout<<"Post : "<<post<<endl;
}
```

pre in post




```cpp
void levelOrderTraversal(Node *root){
    if(!root)
        return;
    queue<Node*> q;
    q.push(root);
    while(!q.empty()){
        int count = q.size();
        for(int i=0; i<count; i++){
            Node *temp = q.front();
            q.pop();
            //print cvurrent child
            cout<<temp->data<<" ";
            //At level 'l' add nodes of 'l+1'th level to queue
            if(temp->left != NULL)
                q.push(temp->left);
            if(temp->right != NULL)
                q.push(temp->right);
        }
        cout<<endl;
    }
}
```

level order

```cpp
static vector<int> path;
bool findPath(Node *root, int target){
    if(root == NULL)
        return false;
    if(root->data == target){
        path.push_back(root->data);
        return true;
    }
    bool checkLeft = findPath(root->left, target);
    if(checkLeft){
        path.push_back(root->data);
        return true;
    }
    bool checkRight = findPath(root->right, target);
    if(checkRight){
        path.push_back(root->data);
        return true;
    }
    return false;
}
```

node to root path

```cpp
vector<int> morrisInTraversal(TreeNode* root) {
    if(root == nullptr){
        return vector<int>{};
    }
    TreeNode *curr = root;
    vector<int> ans;
    while(curr){
        TreeNode *ln = curr->left; // Left node
        //Left tree doesnot exist
        if(ln == nullptr){
            ans.push_back(curr->val);
            curr = curr->right;
        }
        else{
            TreeNode *rightmost = ln;
            //Get rightmost element
            while(rightmost->right && rightmost->right != curr){
                rightmost = rightmost->right;
            }
            //Create thread
            if(rightmost->right == nullptr){
                rightmost->right = curr;
                curr = curr->left;
            }
            //Left sub tree of curr node already traversed
            else{ // (ln->right == curr) -> This will be the case here
                ans.push_back(curr->val);
                curr = curr->right;
                //Removing thread
                rightmost->right = nullptr;
            }
        }
    }
    return ans;
}
```

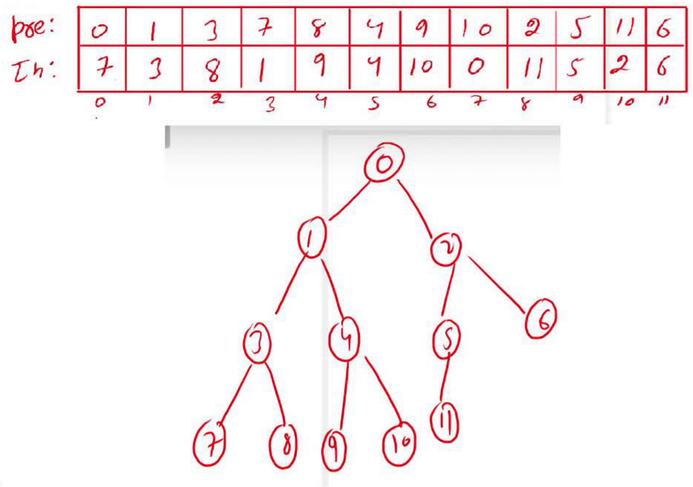
In threaded b-tree we print when
1. left child doesnot exist ie node->left == NULL
2. When we destroy a thread (left tree traversed)

```cpp
vector<int> morrisPreTraversal(TreeNode* root) {
    if(root == nullptr){
        vector<int> vec;
        return vec;
    }
    TreeNode *curr = root;
    vector<int> ans;
    while(curr){
        TreeNode *ln = curr->left; // Left node
        //Left tree doesnot exist
        if(ln == nullptr){
            ans.push_back(curr->val);
            curr = curr->right;
        }
        else{
            TreeNode *rightmost = ln;
            //Get rightmost element
            while(rightmost->right && rightmost->right != curr){
                rightmost = rightmost->right;
            }
            //Create thread
            if(rightmost->right == nullptr){
                ans.push_back(curr->val);
                rightmost->right = curr;
                curr = curr->left;
            }
            //Left sub tree of curr node already traversed
            else{ // (rightmost->right == curr) -> This will be the case here
                curr = curr->right;
                //Removing thread
                rightmost->right = nullptr;
            }
        }
    }
```

In threaded b-tree we print when
1. left child doesnot exist ie node->left == NULL
2. When we create a thread (left tree being traversed)

```
preorder[idx] == inorder[idx] ==> That is a root element of some subtree
All elements on the left  of idx in inorder -> Left subtree of root
All elements on the right of idx in inorder -> Right subtree of root
*/
//F - Build a Btree using pre and inorder
//F - Left and right child already know how to build tree using pre and inorder
//E-F - Pass left childs pre and inorder to left child and right childs pre and inorder to right child and curr node to left and right
child
TreeNode *buildTree(vector<int> &preorder, int psi, int pei, vector<int> &inorder, int isi, int iei){
    if (isi > iei) return nullptr;

    TreeNode *node = new TreeNode(preorder[psi]);

    if(isi == iei) return node;
    int idx = isi;
    while(inorder[idx] != preorder[psi]){
        idx++;
    }
    int countL = idx - isi; //count of left sub tree elements

    node->left = buildTree(preorder, psi+1, psi+countL, inorder, isi, idx-1);
    node->right = buildTree(preorder, psi+countL+1, pei, inorder, idx+1, iei);

    return node;
}
```

btree from pre and in

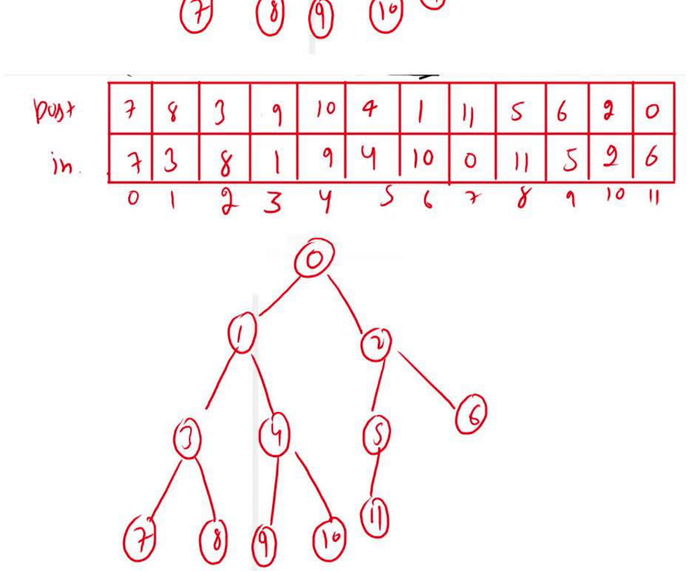



```
// Is tree BST = Left sub tree is bst  AND right sub tree is bst AND current node satisfies conditon of bst
//Current node condition -> if root->data >= leftMax  then root->data is greater than the whole left subtree
//                          if root->data <= RightMin then root->data is smaller than the whole right subtree
bstPair isBST(TreeNode *root){
    if(root == nullptr){
        bstPair base;
        base.isBst = true;
        base.mx = INT_MIN;
        base.mn = INT_MAX;
        return base;
    }
    bstPair left = isBST(root->left);
    bstPair right = isBST(root->right);

    bstPair curr;
    curr.isBst = left.isBst && right.isBst && root->val > left.mx && root->val < right.mn;
    curr.mx = max(root->val, max(left.mx, right.mx));
    curr.mn = min(root->val, min(left.mn, right.mn));

    return curr;
}

bool isValidBST(TreeNode *root){
    bstPair check = isBST(root);
    return check.isBst;
}
```

is bst -> using post prder

```
postorder[x] == inorder[y] ==> That is a root element of some subtree
All elements on the left  of idx in inorder -> Left subtree of root
All elements on the right of idx in inorder -> Right subtree of root

All elements on left of idx in postorder -> Complete Sub tree of node
node x -> is root of the tree
*/
//E - Build a Btree using post and inorder
//F - Left and right child already know how to build tree using post and inorder
//E-F - Pass left childs post and inorder to left child and right childs post and inorder to right child and link curr node to left and
right child
TreeNode *buildTree(vector<int> &postorder, int psi, int pei, vector<int> &inorder, int isi, int iei){
    if(isi > iei)
        return nullptr;

    TreeNode *node = new TreeNode(postorder[pei]);

    if(isi == iei)
        return node;
    int idx = isi;
    while(inorder[idx] != postorder[pei]){
        idx++;
    }
    int countL = idx - isi;

    node->left = buildTree(postorder, psi, psi+countL-1, inorder, isi, idx-1);
    node->right = buildTree(postorder, psi+countL, pei-1, inorder, idx+1, iei);

    return node;
}
```

btree from post and in

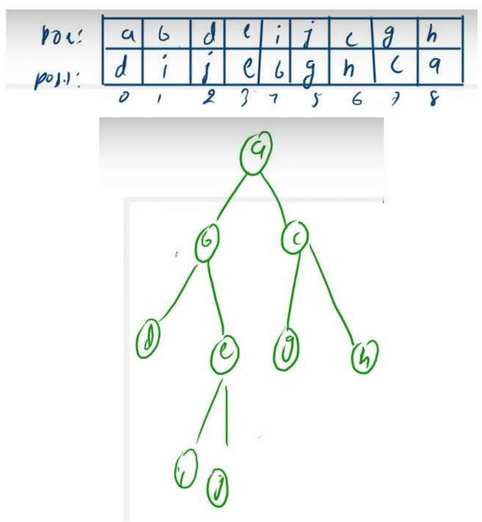



```
WKT the inorder traversal of a bst is sorted in acscending order
Hence if we run inorder traersal and keep track of prev element in the traversal
if prev->val >= curr->val then the elemets are not in ascending order => NOT BST
*/
bool validate(TreeNode* node, TreeNode* &prev) {
    if (node == NULL) return true;

    if (!validate(node->left, prev)) return false;

    if (prev != NULL && prev->val >= node->val) return false;
    prev = node;

    return validate(node->right, prev);
}
```

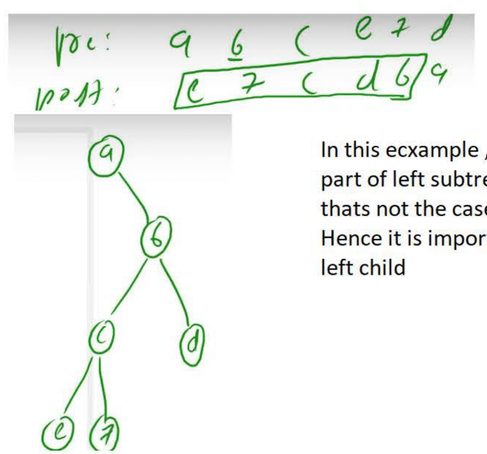
isbst -> Using inorder

```
//E - Build a Btree using pre and post order
//F - Left and right child already know how to build tree using pre and post order
//E-F - Pass left childs pre and post order to left child and right childs pre and post order to right child and curr node to left and
right child
TreeNode *buildTree(vector<int> &preorder, int psi, int pei, vector<int> &postorder, int PTsi, int PTei){
    if(PTsi > PTei) return nullptr;

    TreeNode *node = new TreeNode(preorder[psi]);

    if(PTsi == PTei) return node;

    int idx = PTsi;
    while(postorder[idx] != preorder[psi+1]){
        idx++;
    }
    int countL = idx-PTsi+1;

    node->left = buildTree(preorder, psi+1, psi+countL, postorder, PTsi, idx);
    node->right = buildTree(preorder, psi+countL+1, pei, postorder, idx+1, PTei-1);

    return node;
}
```

btree from pre and post



To generate a tree from pre and post order it is must that the tree should be FULL BINARY TREE OR COMPLETE BTREE



In this ecxample , nodes inside the box must be a part of left subtree according to out program but thats not the case here.
Hence it is important that each parent node has left child

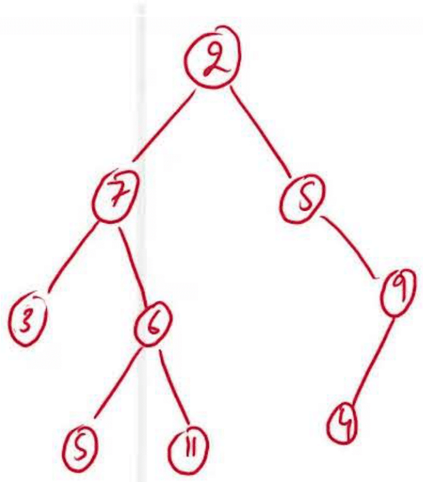
```cpp
TreeNode *buildTree(vector<int> &inorder, int isi, int iei, vector<int> &levelOrder){
    if(isi > iei)
        return nullptr;

    TreeNode *node = new TreeNode(levelOrder[0]);

    if(levelOrder.size() == 1)
        return node;

    int idx = isi;
    while(inorder[idx] != levelOrder[0]){
        idx++;
    }

    //Creating level order for left and right sub tree
    unordered_set<int> s;
    for(int i = isi; i < idx; i++){
        s.insert(inorder[i]);
    }
    vector <int> levelOrderL;
    vector <int> levelOrderR;
    for(int i = 0; i < levelOrder.size(); i++){
        int ele = levelOrder[i];
        if(!s.empty() && s.find(ele) != s.end()){
            levelOrderL.push_back(ele);
            s.erase(ele);
        }
        else
            levelOrderR.push_back(ele);
    }

    node->left = buildTree(inorder, isi, idx-1, levelOrderL);
    node->right = buildTree(inorder, idx+1, iei, levelOrderR);

    return node;
}
```

btree from level order and inorder

LO: 2 7 5 3 6 9 8 11 4

IO: 3 7 5 6 11 2 8 4 9

0 1 2 3 4 5 6 7 8




```cpp
TreeNode* buildBst(vector <int> &levelOrder){
    queue <MyPair> q;
    TreeNode *root;
    q.push(MyPair(nullptr, INT_MIN, INT_MAX));
    int idx = 0;

    while(!q.empty()){
        MyPair p = q.front();
        q.pop();
        int element = levelOrder[idx];

        if (element < p.l || element > p.h) continue;

        TreeNode *node = new TreeNode(element);
        idx++;

        if(p.node == nullptr) root = node;
        else{
            TreeNode *parent = p.node;
            if(element < parent->val) parent->left = node;
            else parent->right = node;
        }

        q.push(MyPair(node, p.l, element)); //Left range
        q.push(MyPair(node, element, p.h)); //Right right range

    }
    return root;
}
```

bst from level order

```cpp
TreeNode* buildBst(vector<int> &inorder, int si, int ei){
    if(si > ei) return nullptr;

    int mid = (si + ei)/2;
    TreeNode *node = new TreeNode(inorder[mid]);
    node->left = buildBst(inorder, si, mid-1);
    node->right = buildBst(inorder, mid+1, ei);

    return node;
};
```

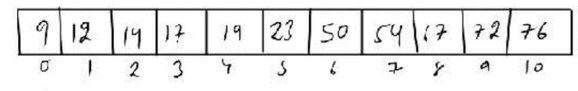
bst from inorder

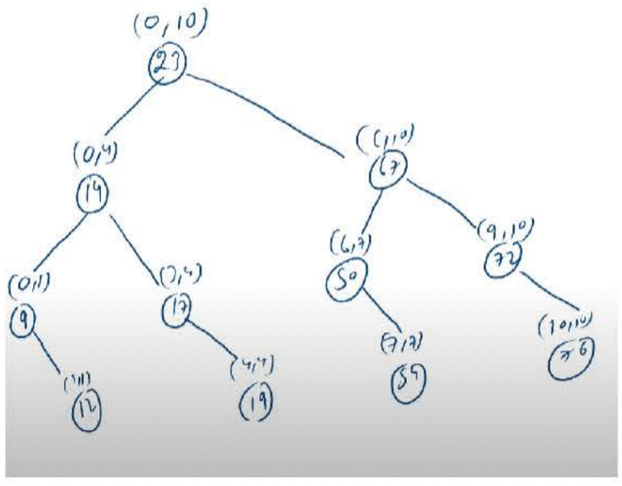
```cpp
static int idx = 0;
TreeNode* buildBst(vector <int> &preorder, int l, int h){
    if(idx >= preorder.size() || preorder[idx] > h || preorder[idx] < l)
        return nullptr;

    TreeNode *node = new TreeNode(preorder[idx++]);
    node->left = buildBst(preorder, l, node->val);
    node->right = buildBst(preorder, node->val, h);

    return node;
}
```

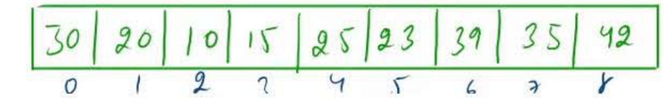
bst from pre order

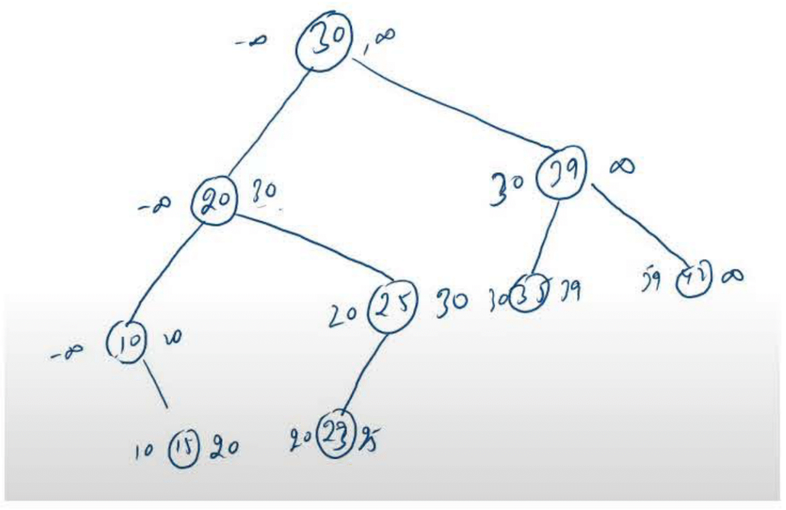
```cpp
TreeNode* buildBst(vector <int> &postorder, int l, int h){
    static int idx = postorder.size()-1;
    if(idx >= postorder.size() || postorder[idx] > h || postorder[idx] < l)
        return nullptr;

    TreeNode *node = new TreeNode(postorder[idx--]);
    node->right = buildBst(postorder, node->val, h);
    node->left = buildBst(postorder, l, node->val);

    return node;
}
```

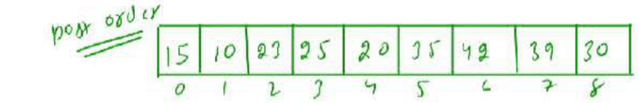
bst from post order

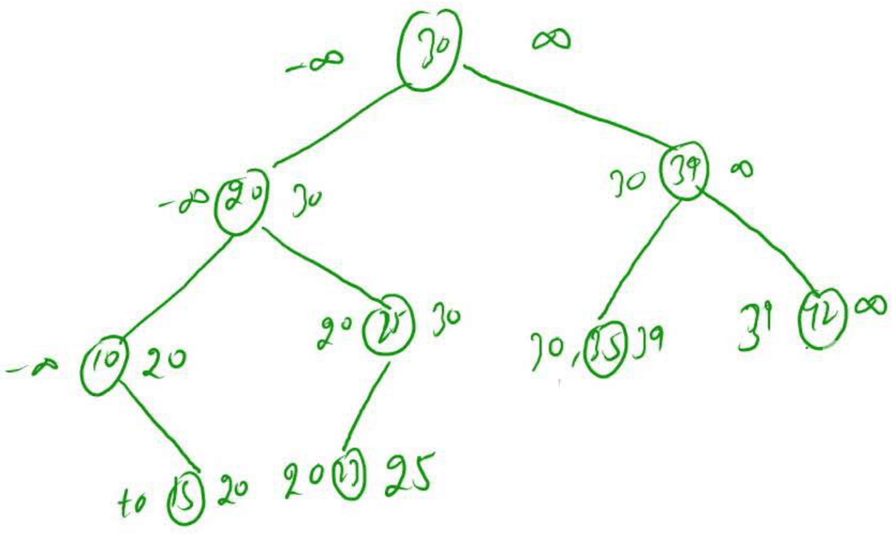




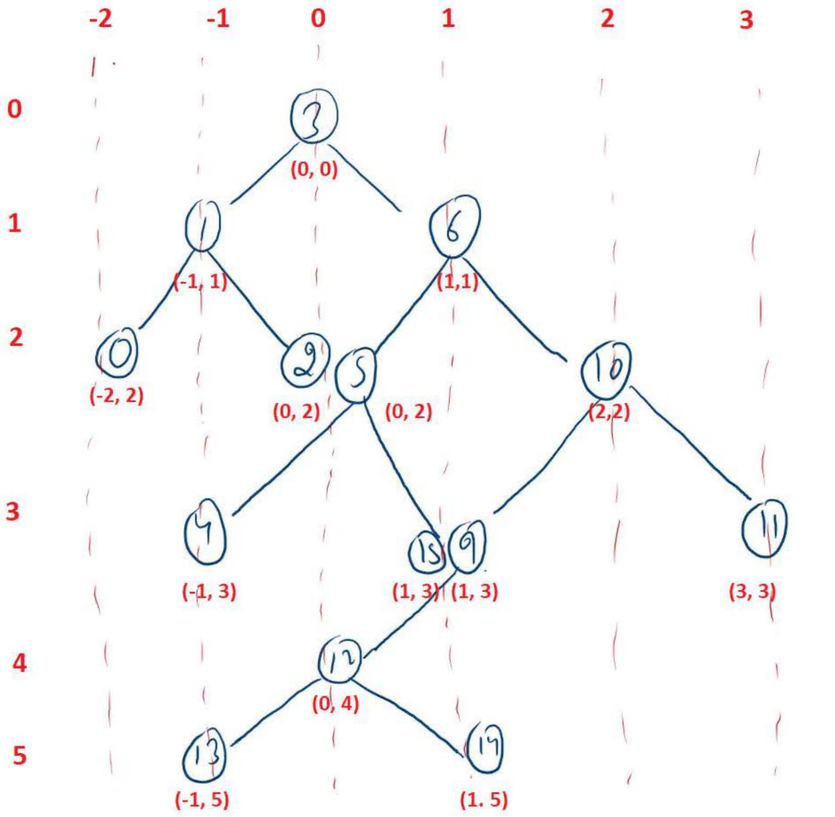


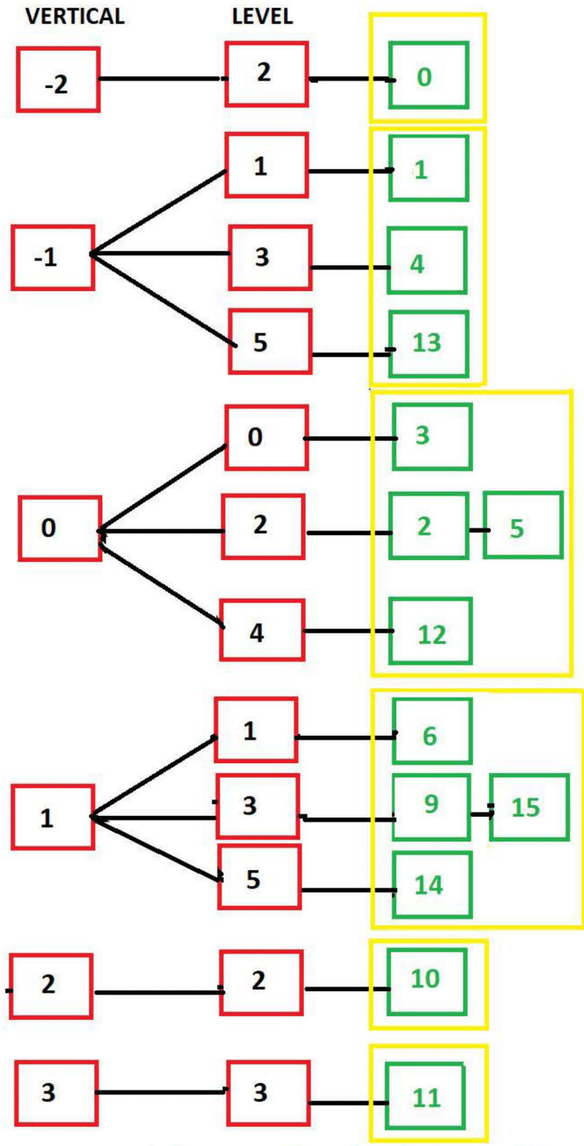






VERTICAL ORDER TRAVERSAL





map < vertical , map < level , multiset < value > > >

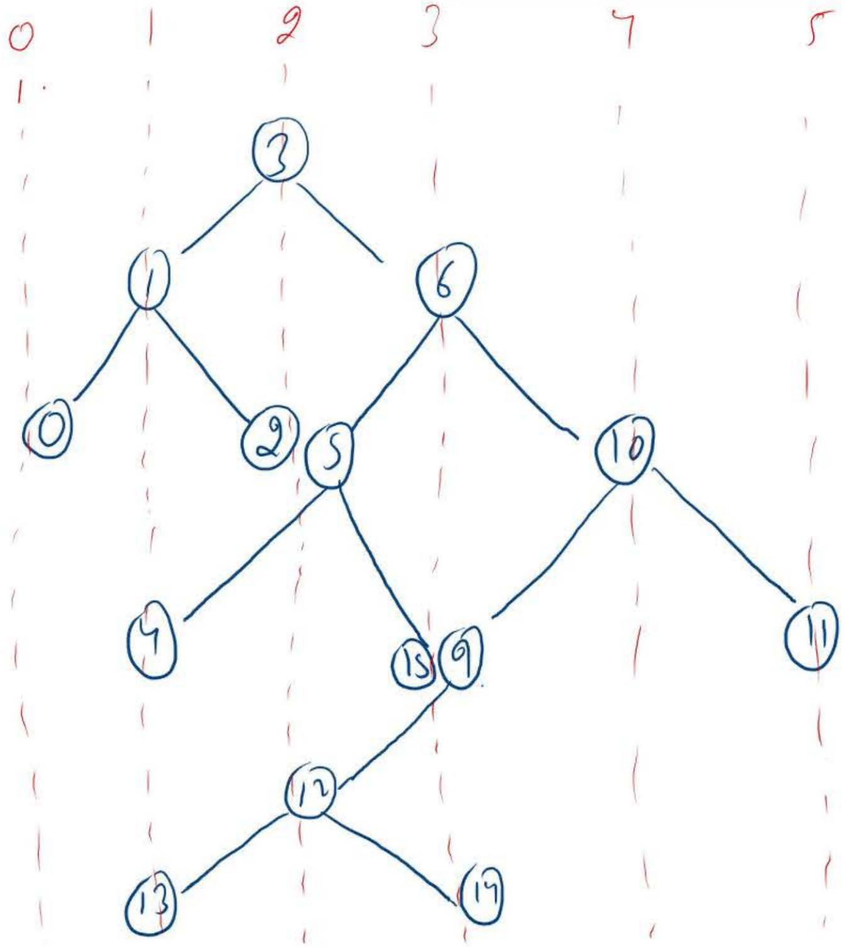
Time complexity - O(N log N)
N (DFS) * log N (insertion into set)

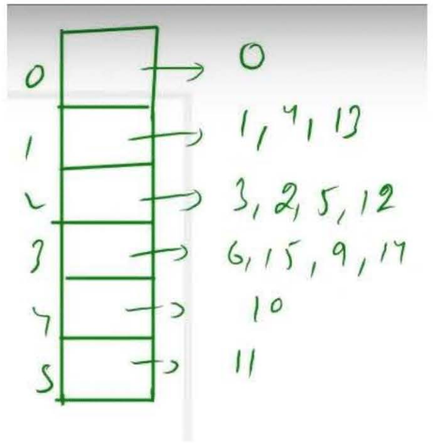
```cpp
//Using DFS
void traversal(TreeNode *node, int x, int y, map<int, map<int, multiset<int>>> &mp){
    if (node == nullptr) return;

    //x -> vertical no(..-2 -1 0 1 2..) , y -> level
    mp[x][y].insert(node->val);
    traversal(node->left, x-1, y+1, mp);
    traversal(node->right, x+1, y+1, mp);
}

vector<vector<int>> verticalTraversal(TreeNode* root) {
    map<int, map<int, multiset<int>>> mp;
    traversal(root, 0, 0, mp);
    vector<vector<int>> ans;
    for (auto p : mp) {
        vector<int> col;
        for (auto q : p.second) {
            col.insert(col.end(), q.second.begin(), q.second.end());
        }
        ans.push_back(col);
    }
    return ans;
}
```

```cpp
//Using BFS or Level Order
vector<vector<int>> verticalTraversal(TreeNode* root) {
    map<int, map<int, multiset<int>>> nodes;
    // queue(node, vertical, level)
    queue < pair <TreeNode*, pair<int, int>> > q;
    q.push({root, {0, 0}});
    while (!q.empty()) {
        auto p = q.front();
        q.pop();
        TreeNode* node = p.first;
        int x = p.second.first, y = p.second.second;

        nodes[x][y].insert(node -> val);
        if (node->left) q.push({node -> left, {x - 1, y + 1}});
        if (node->right) q.push({node -> right, {x + 1, y + 1}});
    }
    vector<vector<int>> ans;
    for (auto p : nodes) {
        vector<int> col;
        for (auto q : p.second) {
            col.insert(col.end(), q.second.begin(), q.second.end());
        }
        ans.push_back(col);
    }
    return ans;
}
```

In this approach if two nodes of a level belong to same vertical, they are inserted into vector in SORTED ORDER(depending on node val)
Eg: 2 -> (0, 2) and 5 (0, 2) -> 2 will apear first and then 2 in vertical order

VERTICAL ORDER TRAVERSAL





Time complexity = O(N)

```cpp
//Using DFS
void traversal(TreeNode *node, int v, map<int, vector<int>> &mp){
    if (node == nullptr) return;

    //v -> vertical no(..-2 -1 0 1 2..) , y -> level
    mp[v].push_back(node->val);
    traversal(node->left, v-1, mp);
    traversal(node->right, v+1, mp);
}

vector<vector<int>> verticalTraversal(TreeNode* root) {
    map<int, vector<int>> mp;
    traversal(root, 0, mp);
    vector<vector<int>> ans;
    for(auto p: mp){
        vector<int> col;
        col.insert(col.end(), p.second.begin(), p.second.end());
        ans.push_back(col);
    }
    return ans;
}
```

using DFS

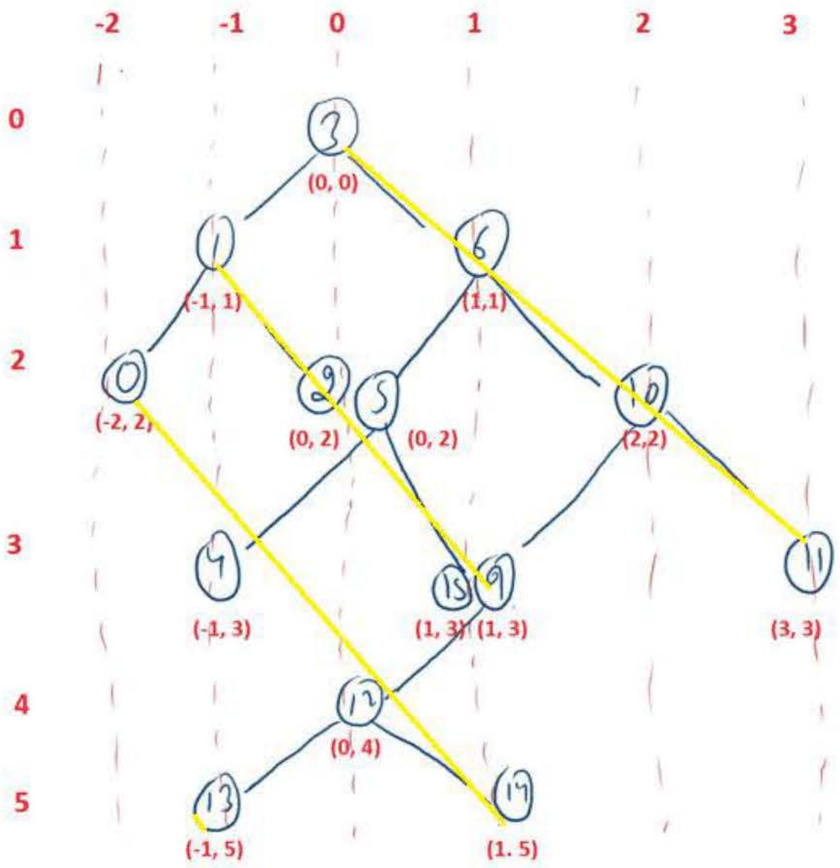
Right view of BT : Last node of each level
Left view of BT : First node of each level
Bottom view of BT : Last node of each vertical in vertical order traversal
Top view fo BT : First node of each vertical in vertical order traversal
Width of BT : Max vertical - Min Vertical + 1

In this approack if two nodes of a level belong to same vertical, they are inserted into vector in order in which it is visited(NOT SORTED)

```cpp
int mn = INT_MAX;
int mx = INT_MIN;
void width(TreeNode *node, int v){
    if(node == nullptr) return;

    mn = min(mn, v);
    mx = max(mx, v);
    width(node->left, v-1);
    width(node->right, v+1);
}

int widthOfBinaryTree(TreeNode* root) {
    width(root, 0);
    return mx-mn+1;
}

vector<vector<int>> verticalTraversal(TreeNode* root) {
    int w = widthOfBinaryTree(root);
    vector <vector<int>> vec(w);
    queue<pair<TreeNode*, int>> q;
    q.push({root, 0});
    while (!q.empty()) {
        auto p = q.front();
        q.pop();
        TreeNode* node = p.first;
        int v = p.second;

        vec[v-mn].push_back(node->val);
        if (node->left) q.push({node -> left, v-1});
        if (node->right) q.push({node -> right, v+1});
    }
    return vec;
}
```

using BFS

DIAGONAL ORDER TRAVERSAL




```cpp
vector<vector<int>> DiagonalTraversal(TreeNode* root) {
    map<int, map<int, multiset<int>>> nodes;
    // queue(node, vertical, level)
    queue<pair<TreeNode*, pair<int, int>>> q;
    q.push({root, {0, 0}});
    while (!q.empty()) {
        int size = q.size();
        while(size-- > 0){
            auto p = q.front();
            q.pop();

            TreeNode* node = p.first;
            int x = p.second.first, y = p.second.second;
            nodes[(y-x)/2][y].insert(node -> val);

            if (node->left) q.push({node -> left, {x - 1, y + 1}});
            if (node->right) q.push({node -> right, {x + 1, y + 1}});
        }
    }
    vector<vector<int>> ans;
    for (auto p : nodes) {
        vector<int> col;
        for (auto q : p.second) {
            col.insert(col.end(), q.second.begin(), q.second.end());
        }
        ans.push_back(col);
    }
    return ans;
}
```

Nodes of same level are in sorted order

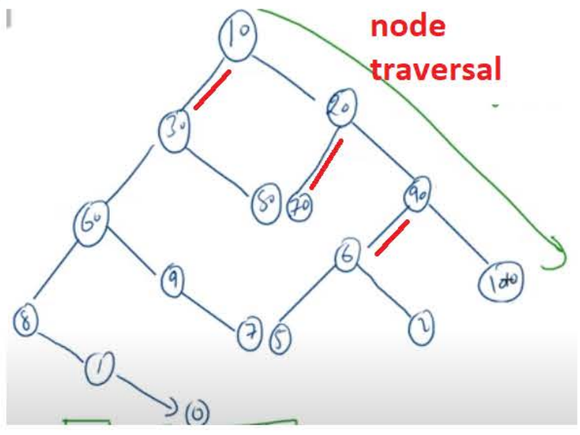
In this approach, our main goal is to traverse diagonally ie current node -> right most node. But if we only do that, we wil loose reference to other nodes. So here we do the following

Before moving to the right, we check if there is a node to left of current node. The left of current node is a part of the next diagonal of the tree. If yes, we push on to the queue.

By doing so, we dont loose any reference

```cpp
while(node){
    diagonal.push_back(node->val);
    if (node->left) q.push(node->left);
    node = node->right;
}
```

Here queue is used to store the next diagonal elements



queue: ~~10~~ | 30 | 70 | 6

```cpp
vector<vector<int>> DiagonalTraversal(TreeNode* root) {
    queue <TreeNode*> q;
    q.push(root);

    vector <vector<int>> ans;
    while(!q.empty()){
        int size = q.size();
        vector<int> diagonal;
        while(size-- > 0){
            TreeNode *node = q.front();
            q.pop();

            while(node){
                diagonal.push_back(node->val);
                if (node->left) q.push(node->left);
                node = node->right;
            }
        }
        ans.push_back(diagonal);
    }
    return ans;
}
```

Nodes of same lvl are in order in which they were visted

BST Iterator

This is an itrator which returns the next element of inoder traversal of BST when asked

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

    void addAllLeft(TreeNode* root){
        while(root != nullptr){
            s.push(root);
            root = root->left;
        }
    }

    int next() {
        TreeNode* top = s.top();
        s.pop();
        addAllLeft(top->right);
        return top->val;
    }

    bool hasNext() {
        return !s.empty();
    }
};
```

Space - O(log N)

Using Morris inorder traversal, we reduce Space to O(1). The code is same as that of traversal, but here when we are in inorder ie when left subtree doesnot exist or left is already traversed, we assign the node value to the variable val and **break**.

```
//Using Morris inorder traversal
class BSTIterator {
private:
    TreeNode *curr = nullptr;
public:
    BSTIterator(TreeNode* root) {
        curr = root;
    }

    int nextElement(){
        int val = -1;
        while(curr){
            TreeNode *ln = curr->left; // Left node
            //Left tree doesnot exist
            if(ln == nullptr){
                val = curr->val;
                curr = curr->right;
                break;
            }
            else{
                //Get rightmost element
                while(ln->right && ln->right != curr){
                    ln = ln->right;
                }
                //Create thread
                if(ln->right == nullptr){
                    ln->right = curr;
                    curr = curr->left;
                }
                //Left sub tree of curr node already traversed
                else{ // (ln->right == curr) -> This will be the case here
                    val = curr->val;
                    curr = curr->right;
                    //Removing thread
                    ln->right = nullptr;
                    break;
                }
            }
        }
        return val;
    }
    int next() {
        return nextElement();
    }

    bool hasNext() {
        return curr != nullptr;
    }
};
```

curr is playing the role of s.top here

K nodes away

```
void printKthlevel(Node *root, int K, Node *blocker){
    if(root == NULL || K<0 || root == blocker)
        return;
    if(K == 0)  //K is ZERo only when we have reached Kth level
        cout<<root->data<<" ";
    printKthlevel(root->left, K-1, blocker);
    printKthlevel(root->right, K-1, blocker);
}

static vector<Node*> path;
bool findPath(Node *root, int target){
    if(root == NULL)
        return false;
    if(root->data == target){
        path.push_back(root);
        return true;
    }
    bool checkLeft = findPath(root->left, target);
    if(checkLeft){
        path.push_back(root);
        return true;
    }
    bool checkRight = findPath(root->right, target);
    if(checkRight){
        path.push_back(root);
        return true;
    }
    return false;
}

void printKnodesFar(Node *root, int target, int k){
    findPath(root, target);
    for(int i=0; i<path.size(); i++){
        printKthlevel(path[i], k-i, i == 0 ? NULL : path[i-1]);
    }
}
```

```
vector<TreeNode*> ans;
void printKlevelsDown(TreeNode *node, int k, TreeNode *blocker){
    if(node == nullptr || k < 0 || node == blocker) return;
    if(k == 0){
        ans.push_back(node);
        return;
    }
    printKlevelsDown(node->left, k-1, blocker);
    printKlevelsDown(node->right, k-1, blocker);
}

int nodeToRootPath(TreeNode *node, int target,int k){
    if(node == nullptr) return -1;

    if(node->val == target){
        printKlevelsDown(node, k, nullptr);
        return 1;
    }

    int ld = nodeToRootPath(node->left, target, k); //left distance
    if(ld != -1){
        printKlevelsDown(node, k-ld, node->left);
        return ld+1;
    }
    int rd = nodeToRootPath(node->right, target, k);
    if(rd != -1){
        printKlevelsDown(node, k-rd, node->right);
        return rd+1;
    }
    return -1;
}
```

Instead of first fetching the node=>root path and then iterating over it, we can print k levels down while finding node->root path itself.
We will return the dist/heignt of target node from curr node (which acts as **i** in the other approach)

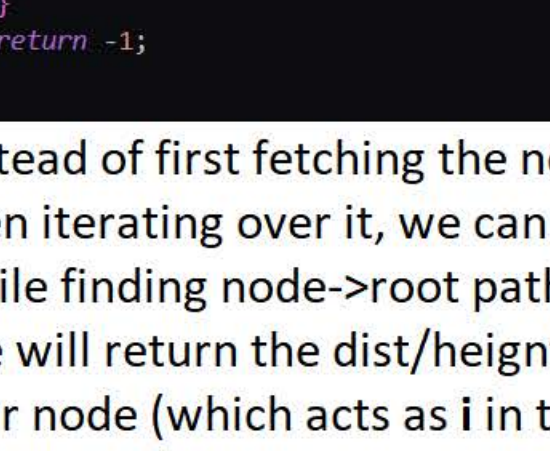
```
//We use the concept of array representation of tree
// Parent = i
// left child = 2*i+1
// right child = 2*i+2
int widthOfBinaryTree(TreeNode* root){
    queue < pair<TreeNode*, int> > q;
    q.push({root, 0});

    int mx = 0;
    while(!q.empty()){
        int size = q.size();
        int l = q.front().second;
        int r = q.front().second;
        while(size-- > 0){
            TreeNode *node = q.front().first;
            int idx = q.front().second;
            q.pop();

            r = idx;
            if(node->left) q.push({node->left, 2*idx + 1});
            if(node->right) q.push({node->right, 2*idx + 2});
        }
        mx = max(mx, r-l+1);
    }
    return mx;
}
```

width of BT

This width is no same as that of vertical order traversal.

The **maximum width** of a tree is the maximum **width** among all levels.

The **width** of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.



Input: root = [1,3,2,5,3,null,9]
Output: 4
Explanation: The maximum width exists in the third level with length 4 (5,3,null,9).

```cpp
//func already knows to construct fullBT for left and right sub tree and return root of left/right subtree
vector<TreeNode*> allPossibleFBT(int n) {
    if(n == 1) return vector<TreeNode*> {new TreeNode(0)};

    vector<TreeNode*> ans;
    for(int i = 1; i < n; i+=2){ // -> This is the most inportant statement because the main property we are using in this
    questio is that a FULL BT can have either 0 or 2 children => node 0, 2, 4... ie with even idx can never become root

        vector<TreeNode*> left = allPossibleFBT(i);          //Returns root of left fullBT(left subtree)
        vector<TreeNode*> right = allPossibleFBT(n-i-1);     //Returns root of right fullBT(right subtree)
        for(TreeNode *l : left){
            for(TreeNode *r : right){
                TreeNode *root = new TreeNode(0);
                root->left = l;
                root->right = r;
                ans.push_back(root);
            }
        }
    }
    return ans;
}
```

All possile full btrees

```cpp
int height(Node *root){
    if(root == NULL)
        return -1;
    int lh = height(root->left);        //Gives max height on left side
    int rh = height(root->right);       //Gives max height on right side
    int h = max(lh, rh) + 1;
    return h;
}

int diameter1(Node *root){
    if(root == NULL)
        return 0;
    //max distance between two nodes on lhs (When both the two deepest nodes are present on LHS)
    int ld = diameter1(root->left);
    //max distance between two nodes on rhs (When both the two deepest nodes are present on RHS)
    int rd = diameter1(root->right);
    //max distance btw left deepest node and right deepest node (When one deepest node is peresent on RHS and other on LHS)
    int f = height(root->left) + height(root->right) + 2;
    int diameter = max(f, max(ld, rd));
    return diameter;
}
```

Diameter - Bruteforce

```cpp
// pair<diameter, height>
pair<int, int> diameter2(Node *root){ //--> Finding height and diameter at the same time
    if(root == NULL)
        return {0, -1};

    //max distance between two nodes on lhs and height of lhs
    pair<int, int> lp = diameter2(root->left);
    //max distance between two nodes on rhs and height of rhs
    pair<int, int> rp = diameter2(root->right);
    //max distance btw left deepest node and right deepest node
    int f = lp.second + rp.second + 2;
    //Finding height
    pair<int, int> myPair;
    myPair.second = max(lp.second, rp.second) + 1;
    //Finding diameter
    myPair.first = max(f, max(lp.first, rp.first));
    return myPair;
}
```

Diameter - Efficient