PAGE No
DATE

FREE KA TREE SERIES

Notes:

BY: Rohit Bindal

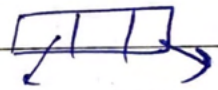
take U forward

Trees

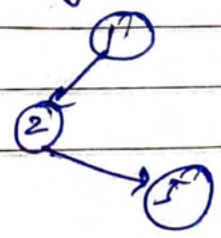
- Binary Tree :- Hierarchical data structure in which each node can have atmost two childrens

1. Full Binary Tree :- either 0 or 2 childrens
2. Complete Binary Tree :- all levels are completely filled except the last level which has all nodes in left only.
3. Perfect BT: all leaf nodes are at same level
4. Balanced BT: height of tree can be atmost logn.
5. Degenerated Tree :- skewed trees (linked list)

- Representation in C++ :-



```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
}

main() {
    struct Node *root = new Node(1);
    root -> left = new Node(2);
    root -> left -> right = new Node(5);
}
```



PAGE No
DATE

Traversals

DFS

→ Inorder (left, root, right)
→ preorder (root, left, right)
→ post order (left, right, root)

BFS (level wise)

① void preorder (node) {
if (node == NULL) return;

cout << node → data;
preorder (node → left);
preorder (node → right);
}

② void bfs (node * root) {
queue <node *> q;
q.push (root);
while (! q.empty())
{
node *f = q.front();
cout << f → data;
if (f → left) q.push (f → left);
if (f → right) q.push (f → right);
q.pop();
}
}

PAGE No
DATE

③ Iterative Preorder:-

→ 1) push root on stack
2) while stack is not empty:
   a) pop the top & print it
   b) push its RIGHT in stack
   c) push its LEFT in stack

Code:
```
stack < node * > st;
st.push(root);
while(!st.empty()) {
    root = st.top();
    st.pop();
    cout << root -> data << " ";
    if(root -> right)
        st.push(root -> right);
    if(root -> left)
        st.push(root -> left);
}
```

④ Iterative Inorder:-

→ 1) Keep going left & pushing into stack until you get null
2) as null comes, print & pop top & then go right & push on the stack
3) repeat

Code:
```
Stack < Node * > st;
Node * node = root;
while(true) {
    if(node != NULL) {
        st.push(node);
        node = node -> left;
    }
```

```
else {
        if(st.empty()) break;
        node = st.top();
        st.pop();
        cout << node -> data << " ";
        node = node -> right;
    }
}
```

(c) Iterative Postorder using 2 stacks:-

1) push root in stack 1
2) pop top of the stack 1, push it in stack 2 and then push its left & right in stack 1.
3) repeat the step 2 until stack 1 is not empty
4) postorder will be available in stack 2

Code:

```
stack<Node *> st1, st2;
st1.push(root);
while(!st1.empty()) {
    root = st1.top();
    st1.pop();
    st2.push(root);
    if(root -> left) st1.push(root -> left);
    if(root -> right) st1.push(root -> right);
    }
while(!st2.empty()) {
        cout << st2.top() -> data << " ";
        st2.pop();
    }
```

PAGE No
DATE

(6) Iterative Postorder using 1 stack:

1) go left until null
2) go one time right
3) go again left untill null

} recursive idea

Code:

```
curr = root;   stack<Node *> st;
while(curr != NULL || !st.empty())
{
    if (curr != NULL) {
        st.push(curr);
        curr = curr->left;
    }
    else {
        temp = st.top() -> right;
        if (temp == NULL) {
            temp = st.top();
            st.pop();
            cout << temp -> data << " ";
            while(!st.empty() && temp == st.top()->right)
            {
                temp = st.top();
                st.pop();
                cout << temp -> data << " ";
            }
        }
        else
            curr = temp;
    }
}
```

PAGE No
DATE

(7) Pre, In & Post order Traversals in one traversal:-

Idea: Maintain three vectors pre, In & post.
push a pair {root, 1} onto the stack
number

i) if num = 1:
a) push root's value in pre vector & pop in st
b) increment num, i.e. push {root, 2} in stack.
c) if root's left exist, push {root→left, 1} in stack

ii) if num = 2:
a) push in in vector & pop from stack
b) increment num & push
c) push right

iii) if num = 3;
a) push in post vector
b) pop from stack

Code:

```
stack<pair<Node*, int>> st;
st.push({root, 1});
vector<int> pre, in, post;
if(root == NULL) return;
while(!st.empty()) {
    auto it = st.top();
    st.pop();

    if(it.second == 1) {
        pre.push_back(it.first->val);
        it.second++;
```

```
st.push(it);
if (it.first -> left != NULL) {
        st.push({it.first -> left, 1});
    }
}
else if (it.second == 2) {
        in.push_back(it.first -> val);
        it.second++;
        st.push(it);

        if (it.first -> right != NULL) {
            st.push({it.first -> right, 1});
        }
    }
else {
        post.push_back(it.first -> val);
    }
}
```

(8) Maximum Depth (Height) of Binary Tree :-

Code:

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

    int left = Height(root -> left);
    int right = Height(root -> right);

    return 1 + max(left, right);
}
```

PAGE No
DATE

① Check for Balanced Binary Tree:-

for every node: height(left) - height(right) <= 1

→ Naive approach: (Time = $O(n^2)$)

```
Bool check(node) {
    if(node == NULL) return true;

    int left = Height(node->left);
    int right = Height(node->right);

    if (abs(left-right) > 1) return false;

    bool left = check(node->left);
    bool right = check(node->right);

    if(!left || !right) return false;

    return true;
}
```

→ $O(N)$ approach: (same as finding height)

Code:

```
int check(node) {
    if(node==NULL) return 0;

    int left = check(node->left);
    int right = check(node->right);
    if (left==-1 or right==-1) return -1;
    if (abs(left-right) > 1) return -1;

    return max(left, right) + 1;
}
```

PAGE No
DATE

(5) Diameter of B.T. :- (longest path b/w 2 nodes)

→ (1) $O(N^2)$:

```
int diameter(node * root) {
    if (root == NULL) return 0;

    int h1 = height(root -> left);
    int h2 = height(root -> right);

    int op1 = h1 + h2; //when passes through root
    int op2 = diameter(root -> left);
    int op3 = diameter(root -> right);
    return max({op1, op2, op3});
}
```

(2) $O(N)$:

```
class Pair {
    public:
    int height;
    int diameter;
}

Pair diameter(node * root) {
    Pair p;
    if (root == NULL) {
        p.height = 0;
        p.diameter = 0;
        return p; }

    Pair left = diameter(root -> left);
    Pair right = diameter(root -> right);
    p.height = 1 + max(left.height, right.height);
```

```
p.diameter = max({left.height + right.height,
                  left.diameter,
                  right.diameter});
return p;
}
```

(11) Maximum path sum in B.T. :-
find max sum of a path between any two nodes

code:
```
int maxPathSum(node * root) {
    int maxi = INT_MIN;
    maxSum(root, maxi);
    return maxi;
}

int maxSum(node * root, int & maxi) {
    if(root == NULL) return 0;

    int left = Max(0, maxSum(root->left, maxi));
                 to ignore negative path sums
    int right = max(0, maxSum(root->right, maxi));
    maxi = max(maxi, left + right + root->val);
    return max(left, right) + root->val;
}
```

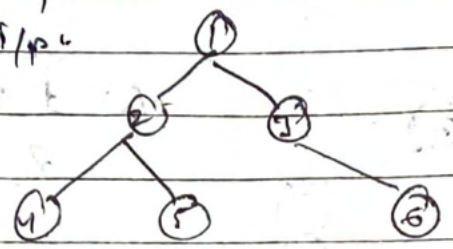
(12) Check if two trees are identical

code
```
bool isSame(node * p, node * q) {
    if (p == NULL || q == NULL) {
        return (p == q); }

    return (p->val == q->val) &&
           isSame(p->left, q->left) &&
           isSame(p->right, q->right); }
```

PAGE No
DATE

③ Zig-Zag / Spiral Traversal in a B.T. :-

eg. i/p:



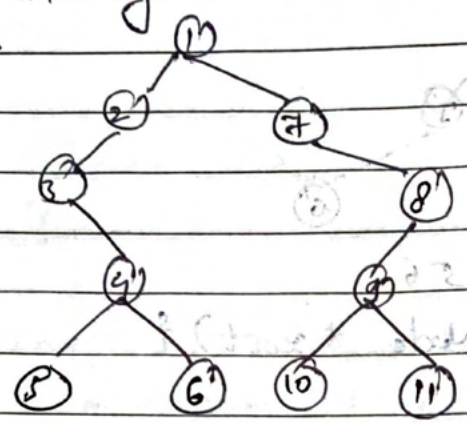
o/p: 1 3 2 4 5 6

Code:
```
vector<int> findSpiral(Node *root) {
    vector<int> ans;
    queue<Node*> q;
    if (root == NULL) return ans;
    q.push(root);
    bool rightToLeft = true;
    while(!q.empty()) {
        int size = q.size();
        vector<int> temp(size);
        for(int i=0; i<size; i++) {
            Node *f = q.front();
            q.pop();
            if(rightToLeft) temp[size-i-1] = f->data;
            else temp[i] = f->data;

            if(f->left) q.push(f->left);
            if(f->right) q.push(f->right);
        }
        rightToLeft = !rightToLeft;
        for(int i=0; i<size; i++)
            ans.push_back(temp[i]);
        }
    return ans;
}
```

PAGE No
DATE

(14) Boundary Traversal in B.T. :- (anticlockwise)

i/p:



o/p: 1 2 3 4 5 6 10 11 9
8 7

→ Left Boundary (excluding leaves) + leaf nodes + right boundary in reverse (excluding leaves)

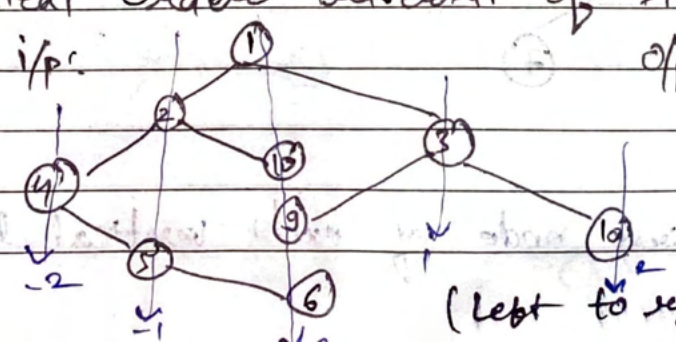
Code:
```
bool isLeaf(Node *root)
    return (root->left==NULL && root->right==NULL);

void left(Node *root, vector<int> &ans) {
    root = root->left;
    if(!root) return;
    while(!isLeaf(root)) {
        ans.push_back(root->data);
        if(root->left) root = root->left;
        else root = root->right;
    }
}

void leaf(Node* root, vector<int> &ans) {
    if(root==NULL) return;
    if(isLeaf(root)) ans.push_back(root->data);
    leaf(root->left, ans);
    leaf(root->right, ans);
}
```

PAGE No
DATE

```
void right (Node *root, vector<int> &ans) {
        vector<int> temp;
        root = root -> right;
        if(!root) return;
        while(!isLeaf(root)) {
            temp.push_back(root->data);
            if(root->right) root = root->right;
            else root = root -> left;
        }

    for(int i = temp.size()-1; i>=0; i--)
        ans.push_back(temp[i]);
    }

vector<int> boundary(Node *root) {
        vector<int> ans;
        if(!root) return ans;
        ans.push_back(root->data);
        if(isLeaf(root)) return ans;
        left(root, ans);
        leaf(root, ans);
        right(root, ans);
        return ans;
    }
```

(15) Vertical Order Traversal of Binary Tree :-

i/p:



o/p: 4 2 5 1 ~~10~~ 9 10 6 3 10

In case of overlap, write smaller value first.

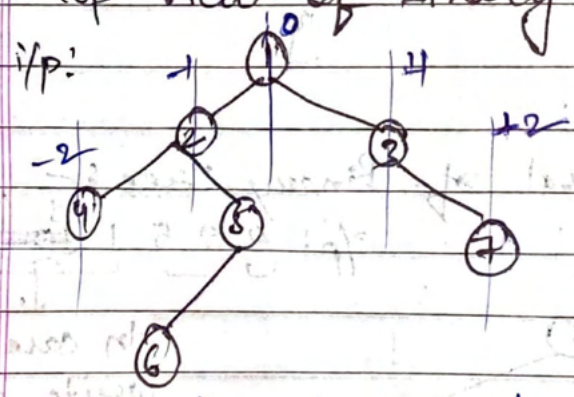
(Left to right + Top to Bottom)

PAGE No
DATE

Code:
```
vector<int> verticalOrder(Node *root) {
    vector<int> ans;
    Map <int, vector<int>> mp; // {verticalLevel, nodes}
    queue< pair< Node*, int>> q; // {root, verticalLevel}
    q.push({root, 0});

    While( !q.empty()) {
        Node *f = q.front().first;
        int vl = q.front().second;
        q.pop();
        mp[vl].push_back(f->data);
        if(f->left) q.push({f->left, vl-1});
        if(f->right) q.push({f->right, vl+1});
    }

    for(auto it: mp) {
        for(auto i: it.second)
            ans.push_back(i);
    }

    return ans;
}
```

(16) Top view of Binary Tree :-

i/p:



o/p: 4 2 1 3 7

→ print the first node of each vertical line

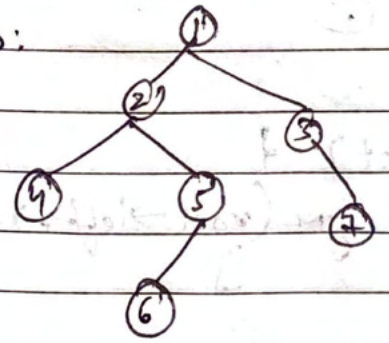
PAGE No
DATE

Code:
```
vector<int> topView (Node *root) {
    vector<int> ans;
    queue< pair<Node*, int>> q;
    Map<int, int> mp;
    if (!root) return ans;
    q.push({root, 0});
    while(!q.empty()) {
        Node *f = q.front().first;
        int vl = q.front().second;
        q.pop();
        if (mp.find(vl) == Mp.end())
            Mp[vl] = f->data;
        if (f->left) q.push({f->left, vl-1});
        if (f->right) q.push({f->right, vl+1});
    }
    for(auto it: mp)
        ans.push_back(it.second);
    return ans;
}
```

By removing this line, you will get bottom view of B.T.

(17) Right view of Binary Tree :-

i/p:



o/p: 1 3 7 6

→ print last node of every level

Generally: Space in Iterative (level order) O(n) > Recursive O(Height)

PAGE No
DATE

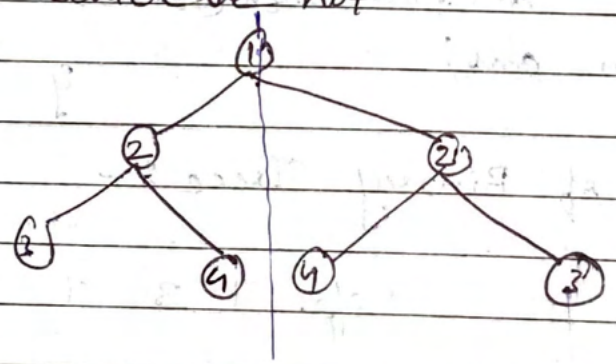
→ if you traverse each level from right to left then first node of each level will be answer
→ use recursive traversal as: root right left, and print when you visit a level for first time

Code
```
void pre(Node * root, int level, vector<int> &ans){
    if(root == NULL) return;

    if(level == ans.size())
        ans.push_back(root->data);

    pre(root->right, level+1, ans);
    pre(root->left, level+1, ans);
}
```

swap these to get the left view.

(18) Check for Symmetrical Binary Trees:-
whether it forms a mirror image of itself around the center or not

i/p:



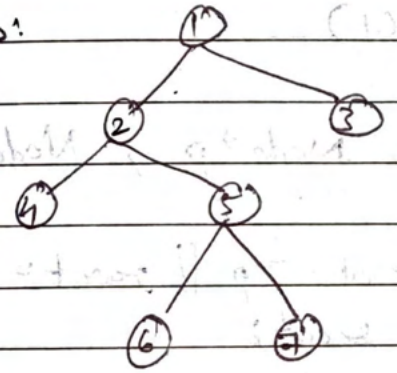
o/p: True

Code
```
bool isSymmetric(Node* root) {
    return root == NULL || sym(root->left, root->right);
}

bool sym(Node* left, Node* right) {
```

```
if(left == NULL || right == NULL)
        return left == right;

if(left->data != right->data) return false;

return sym(left->left, right->right)
        &&
        sym(left->right, right->left);
}
```

(19) Print Root to node path:-

i/p:



node = 7

o/p: {1, 2, 5, 7}

Code:

```
bool getPath(Node * root, vector<int> &arr, int x){
        if(!root) return false;

        arr.push_back(root->val);

        if(root->val == x) return true;

        if(getPath(root->left, arr, x) || getPath(root->right,
                                                arr, x))
                return true;
```

```
    ... pop_back();
    return false;
}
```

(20) Lowest Common Ancestor (LCA) in BT :-

m-1 find path from root to first node & root to second node, store in two vectors
→ now find the last common element in both vectors.

time = space = O(n)

m-2 Time = O(n), Space = O(1)

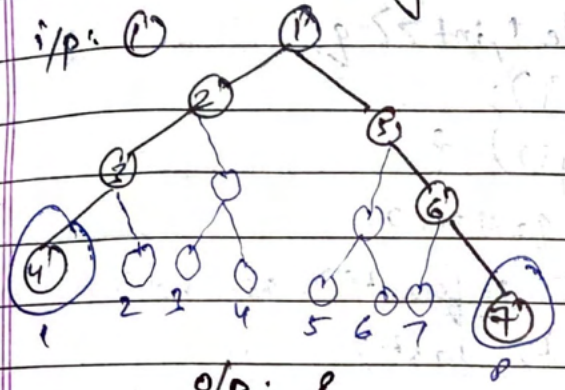
code
```
Node* LCA(Node* root, Node* p, Node* q) {
    //base case
    if (root==NULL || root==p || root==q)
        return root;

    Node* left = LCA(root->left, p, q);
    Node* right = LCA(root->right, p, q);

    //result
    if (left==NULL) return right;
    else if (right==NULL) return left;
    else {
        // we found our result
        return root;
    }
}
```

PAGE No
DATE

(25) Maximum width of Binary Tree :-

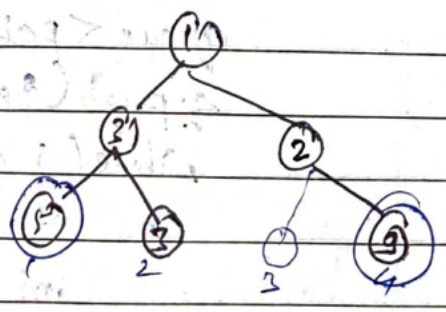
width: No. of nodes in a level between any 2 nodes including those 2 nodes

i/p: (1)



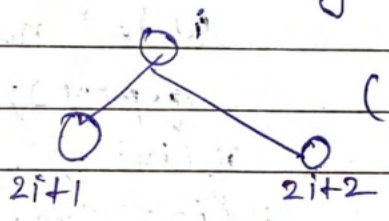
o/p: 8

(2)



o/p: 4

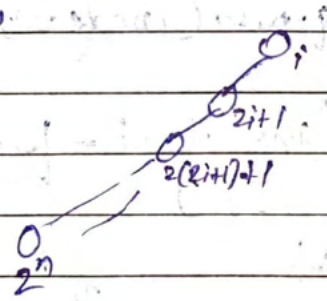
Idea: we can give index to every node



(0 Based indexing)

and then width of a level = last index − first index + 1

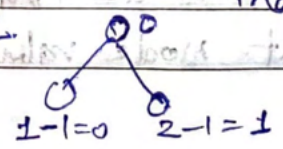
but if tree is skewed then we might get overflow



So, we will try that first node of each level will start with index 0

so, original index = current index − minimum index (i.e. index of first node)

eg.



PAGE No
DATE

code

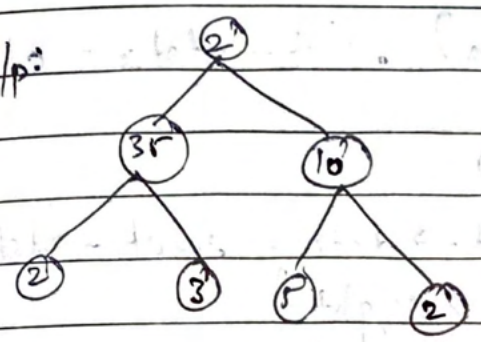
```
int width(Node * root) {
    if(!root) return 0;
    int ans = 0;
    queue<pair<Node *, int>> q;
    q.push({root, 0});
    while(!q.empty()) {
        int size = q.size();
        int mini = q.front().second;
        int first, last;
        for(int i=0; i<size; i++) {
            int curr_id = q.front().second - mini;
            Node * node = q.front().first;
            q.pop();
            if(i==0) first = curr_id;
            if(i==size-1) last = curr_id;
            if(node->left)
                q.push({node->left, curr_id*2+1});
            if(node->left)
                q.push({node->right, curr_id*2+2});
        }
        ans = Max(ans, last - first + 1);
    }
    return ans;
}
```

(22) Children Sum property

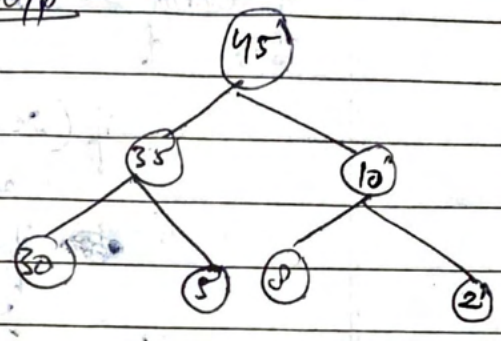
each node value should be equal to sum of its left & right node values

→ you can increement node value (you cant decrease)

i/p:

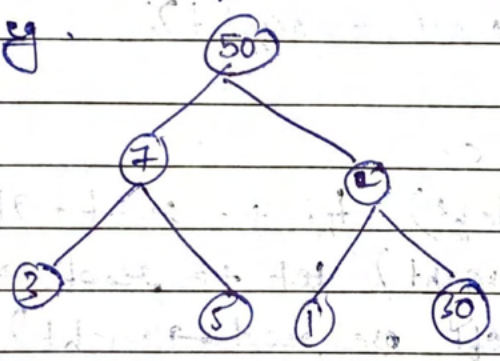


one of o/p:

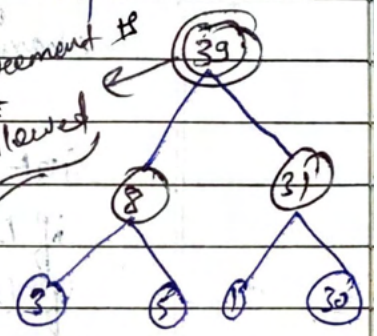


→ We cant replace simply the addition of left & right child from bottom to up.

eg.



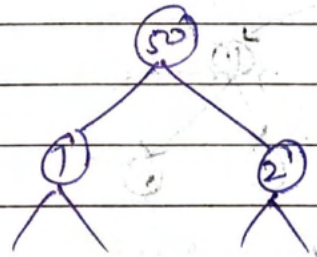
Decrement is not allowed



shortage of (50 − 39) = 11

→ So going from top to down, we will change the values in such a way that when we again go from bottom to up then there will not be any shortage.

eg.



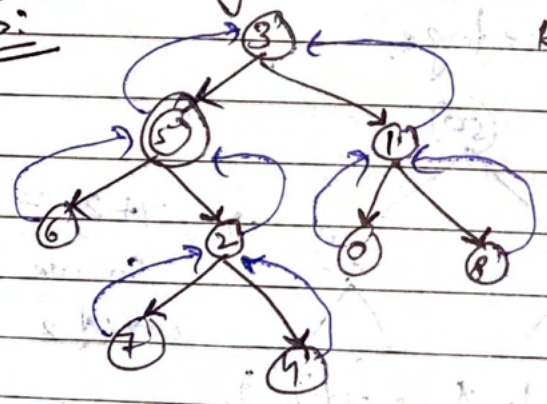
→ 7+2 = 9 < 50
So we will make 7 & 2 as 50 only.

Code:
```
void changeTree(Node * root) {
    if (root == NULL) return;
    int child = 0;
    if (root → left) child += root → left → data;
    if (root → right) child += root → right → data;
```

```
if (child >= root->data) root->data = child;
else {
        if (root->left)
            root->left->data = root->data;
        if (root->right)
            root->right->data = root->data;
    }
changeTree(root->left);
changeTree(root->right);

int tot = 0;
if (root->left) tot += root->left->data;
if (root->right) tot += root->right->data;
if (root->left or root->right) root->data = tot;
}
```

(23) Print all the nodes at distance of k from the given target node

i/p:



k = 2, target = 5 (add. given)

o/p: 7, 4, 1.

→ as we cant move backward so first store all the parents using BFS.

→ start BFS from target node & go to left, right & parent if they are not visited already &

increment the distance by 1, whenever distance = k, print the nodes

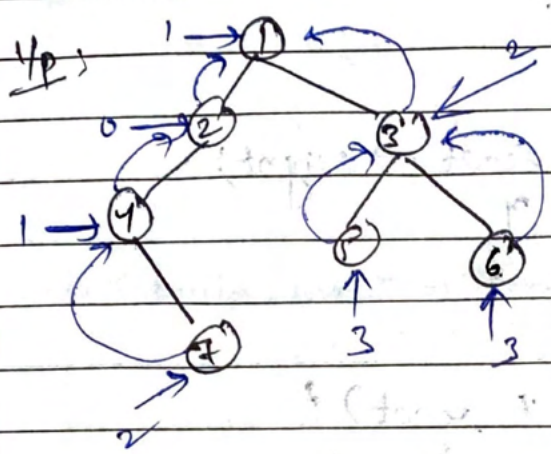
code:
```
void markParents(Node *root, unordered_map<Node*,Node*> &parents)
{
    queue<Node*> q;
    q.push(root);
    while(!q.empty()) {
        Node* f = q.front();
        q.pop();
        if(f->left) {
            parent[f->left] = f;
            q.push(f->left);
        }
        if(f->right) {
            parent[f->right] = f;
            q.push(f->right);
        }
}
vector<int> distanceK(Node* root, Node* target, int k) {
    unordered_map<Node*, Node*> parent;
    markParents(root, parent);

    unordered_map<Node*, bool> visited;
    queue<Node*> q;
    q.push(target);
    visited[target] = true;
    int curr_level = 0;
    while(!q.empty()) {
```

```
int size = q.size();
if (curr_level == k) break;
for(int i=0; i<size; i++) {
    Node* current = q.front();
    q.pop();
    if (current -> left && !visited[current->left])
    {
        q.push(current->left);
        visited[current->left] = true;
    }
    if (current -> right && !visited[current->right])
    {
        q.push(current -> right);
        visited[current -> right] = true;
    }
    if (parent[current] && !visited[parent[current]])
    {
        q.push(parent[current]);
        visited[parent[current]] = true;
    }
}
}
vector<int> result;
while(!q.empty()) {
    Node* curr = q.front();
    q.pop();
    result.push_back(curr->val);
}
return result;
}
```

PAGE No
DATE

(24) Minimum time to burn a B.T. from a node

I/p:



node = 2 (node value is given)

O/p: 3

→ steps: (1) Mark parents & also get add. of target node
(2) apply BFS. (same as previous)

(25) Count total nodes in a complete Binary Tree in O(Logn)

→ if height of CBT = h, then
#nodes = $2^h - 1$.

→ at a particular node, just find height of its left most & right most node, if both are equal then just return $2^h - 1$ otherwise 1 + left + right.

code:
```
int leftMost(Node* root) {
    int h=0;
    while(root) {
        h++;
        root = root->left;
    }
    return h;
}

int rightMost(Node* root) {
```

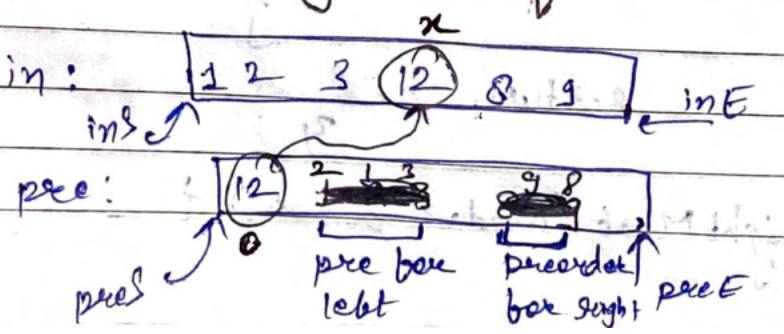
```
int h = 0;
while (root) {
        h++;
        root = root -> right;
        }
    return h;
}
int countNodes(Node * root) {
    if (root == NULL) return 0;

    int l = leftMost(root);
    int r = rightMost(root);

    if (l == r) return (1<<l) - 1;

    return 1 + countNodes(root -> left) +
                countNodes(root -> right);
}
```

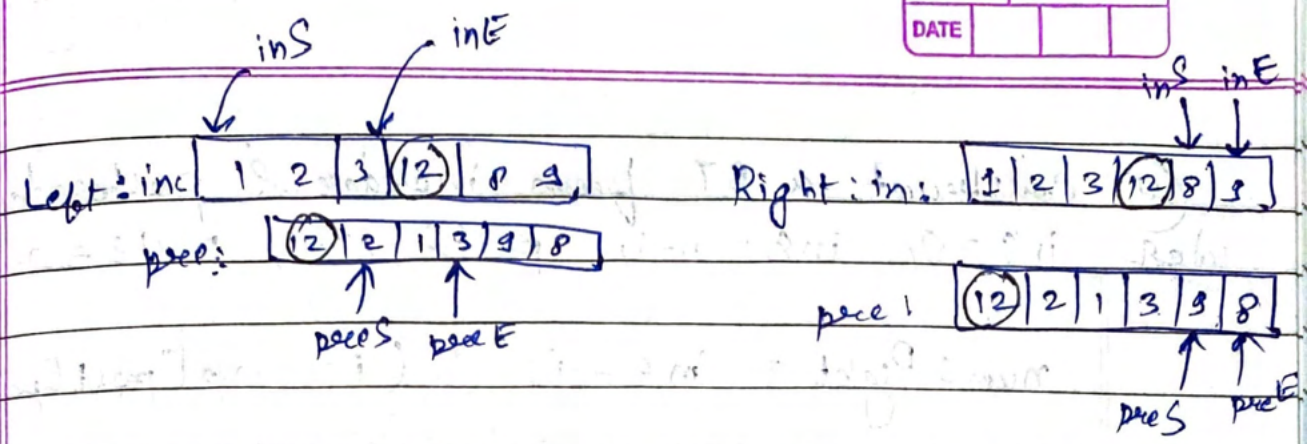
NOTE: 1) Given Preorder & postorder, many binary trees can be constructed.

2) To construct a unique binary tree, inorder must be given.

(26) Construct a binary Tree from Preorder & Inorder

Step (1)






```
Node* build(int *in, int *pre, int preS, int preE,
            int inS, int inE, map<int, int> &mp)
{
    if (preS > preE or inS > inE) return NULL;

    int i = mp[pre[preS]];
    int numsLeft = i - inS;

    Node * root = new Node(pre[preS]);

    root->left = build(in, pre, preS+1, preS + numsLeft,
                       inS, i-1, mp);
    root->right = build(in, pre, preS + numsLeft + 1,
                        preE, i+1, inE, mp);

    return root;
}

Node * buildTree(int in[], int pre[], int n) {
    map<int, int> mp;
    for(int i=0; i<n; i++) mp[in[i]] = i;

    return build(in, pre, 0, n-1, 0, n-1, mp);
}
```

PAGE No
DATE

(27) Construct a BT from inorder & postorder :-

idea: $inS = 0$, $inE = n-1$, $postS = n-1$, $postE = 0$.

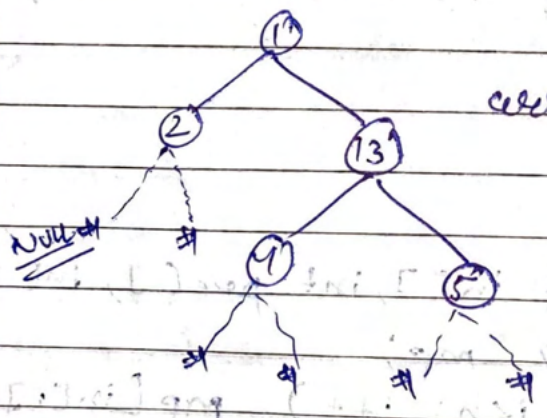
$numsRight = inE - i$; $(i = mp[post[postS]])$

root → left = build(in, post, postS − numsRight − 1, postE, inS, i − 1, mp);
root → right = build(in, post, postS − 1, postS − numsRight, i+1, inE, mp);

(28) Serialize and Deserialize a Binary Tree :-
Serialization is to store tree in a file so that it can be restored later. The structure of tree must be maintained. Deserialization is reading tree back from file.

→ Task is to complete the function "Serialize" which stores the tree into an array A[ ] and "deSerialize" which deserializes the array to tree and returns it.

eg



store it level wise :
arr[ ] = 1, 2, 13, #, #, 9, 5, #, #, #, #

code:
```
vector<int> serialize(Node * root) {
    vector<int> v;
    if (root == NULL) return v;
```

```
queue<Node *> q;
q.push(root);

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

        if(f==NULL) v.push_back(0);
        else {
                v.push_back(f->data);
                q.push(f->left);
                q.push(f->right);
        }
    }    return v;
}

Node *deSerialize(vector<int> &v) {
    if(v.size()==0) return NULL;
    Node *root = new Node(A[0]);
    queue<Node*> q;
    q.push(root);
    int i=0;
    while(!q.empty()) {
        Node *f = q.front();
        q.pop();

        i++;
        if(v[i]!=0) {
            f->left = new Node(v[i]);
```

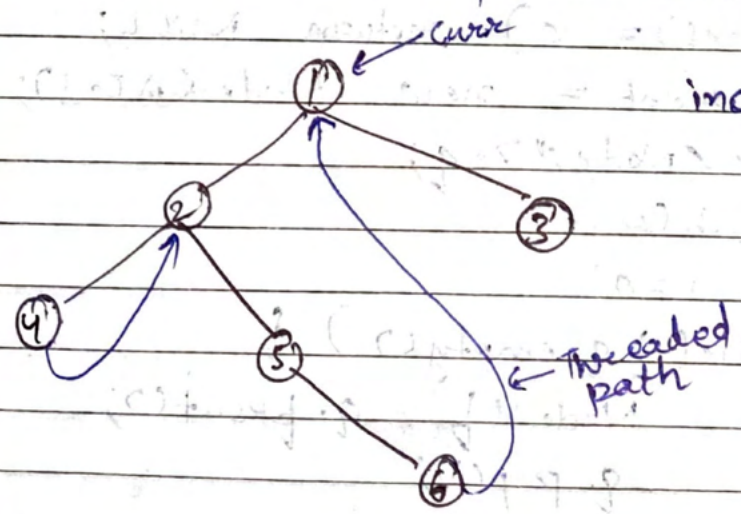
```
            q.push(f->left); }
        else {
                f->left = NULL;
            }
        i++;
        if (A[i]!=0) {
            f->right = new Node(A[i]);
            q.push(f->right);
            }
        else f->right = NULL;
    }
    return root;
}
```

(29) Morris Traversal (Inorder):

→ uses threaded binary tree

→ time = $O(n)$, Space = $O(1)$. (No stack/Queue)

eg



inorder : left Root Right

→ When curr is moved to left, then we will make a threaded path from the right most node of left subtree to the root, because after

PAGE No
DATE

completing left subtree, we need to go to root again

→ if left of curr is null then that means curr is root so just print it & go to right.

→ if you move to the right most node in left subtree & if there is already a thread then that means left part is completed so remove the thread & move curr to right & print the root too.

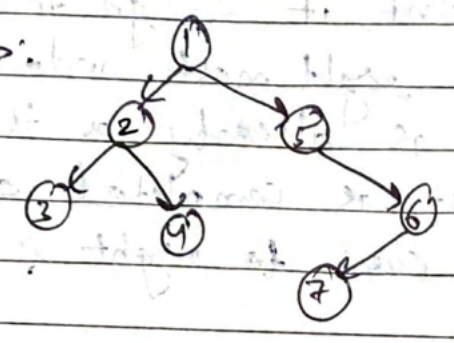
code:
```
vector<int> getInorder(Node * root) {
    vector<int> inorder;
    Node *curr = root;
    while(curr != NULL) {
        if (curr->left == NULL) {
            inorder.push_back(curr->val);
            curr = curr->right;
        }
        else {
            Node *prev = curr->left;
            While(prev->right && prev->right != curr)
                prev = prev->right;

            if (prev->right == NULL) {
//Create Thread// prev->right = curr;
                curr = curr->left; }
            else {
                prev->right = NULL; //Remove Thread
                inorder.push_back(curr->val);
                curr = curr->right;
            }
        }
    }
```

If you shift this line then it will become pre order

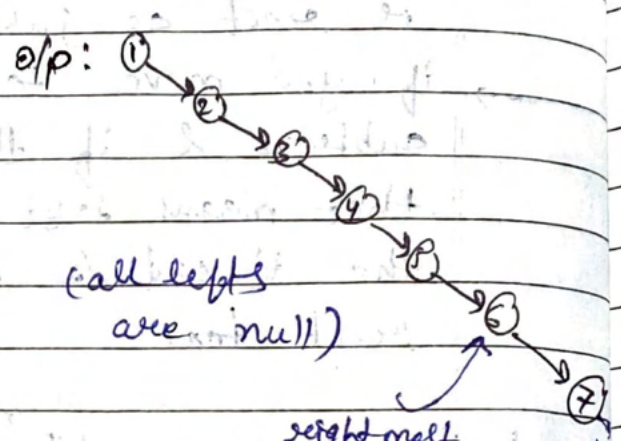
(30) Flatten a Binary Tree to Linked List :- (Preorder)

M-1 : Recursive :

i/p :



o/p :



(all lefts are null)

right most nodes → right = its left

So reversal suitable is :
right left root

Code :

```
Node * prev = NULL;
void flatten(Node * root) {
    if (root == NULL) return;

    flatten(root -> right);
    flatten(root -> left);

    root -> right = prev;
    root -> left = NULL;
    prev = root;
}
```

M-2 : Iterative :

1) push root into stack
2) curr = top of stack

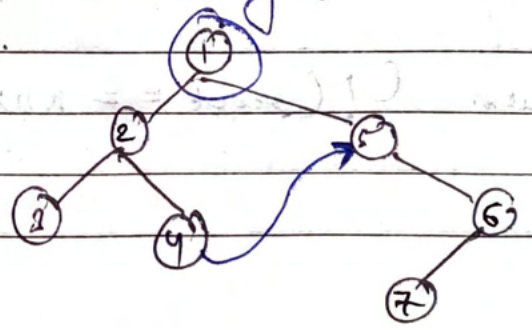
3) push curr's left & right in stack (order) (right → left)
4) curr's right will be top of stack
& curr's left = null
5) curr = pop()
6) repeat 3, 4, 5 until stack is not empty.

code:
```
void flatten(Node * root) {
    stack<Node*> st;
    st.push(root);
    Node* curr;
    while(!st.empty()) {
        curr = st.top(); st.pop();
        if(curr → right) st.push(curr → right);
        if(curr → left) st.push(curr → left);

        if(!st.empty()) curr → right = st.top();

        curr → left = NULL;
    }
}
```

M-3 : space = O(1) (similar to Morris Traversal)

Step (1) Connect the rightmost node in left subtree of the root to root's right.

eg.



PAGE No
DATE

code:
```
void flatten(Node *root){
    Node *curr = root;
    while(curr != NULL){
        if(curr->left != NULL){
            Node *prev = curr->left;
            while(prev->right != NULL)
                prev = prev->right;
            prev->right = curr->right;
            curr->right = curr->left;
            curr->left = NULL;
        }
        curr = curr->right;
    }
}
```

- Binary Search Tree :-

→ left < root, right > root ∀ roots
left subtree = BST
right subtree = BST

① Search a node in BST :-

code:
```
bool search(Node *root, int x){
    while(root and root->data != x){
        if(root->data > x) root = root->left;
        else root = root->right;
    }
    return (!(root == NULL));
}
```

PAGE No
DATE

② Ceil in a BST

find ~~ceil~~ lowest value >= key.

code

```
int findCeil (Node *root, int key) {
    int ceil = -1;
    while (root) {
        if (root -> data == key) {
            ceil = root -> data;
            return ceil;
        }
        if (key > root -> data) {
            root = root -> right;
        }
        else {
            ceil = root -> data;
            root = root -> left;
        }
    }
    return ceil;
}
```

③ Floor in a BST

find greatest value <= key

code

```
int findFloor (Node *root, int key) {
    int floor = -1;
    while (root) {
        if (root -> val == key) return key;
        if (key > root -> val) {
            floor = root -> val;
            root = root -> right; }
        else { root = root -> left; } }
    return floor;
}
```

PAGE No
DATE

④ Insert a given node in BST :-

Code

```
Node* insert(Node* root, int val) {
    if (root == NULL) return new Node(val);
    Node* curr = root;
    while(true) {
        if (curr->val <= val) {
            if (curr->right != NULL) curr = curr->right;
            else {
                curr->right = new Node(val);
                break;
            }
        }
        else {
            if (curr->left != NULL)
                curr = curr->left;
            else {
                curr->left = new Node(val);
                break;
            }
        }
    }
    return root;
}
```

⑤ Delete a Node in BST :-

code:

```
Node *deleteNode(Node* root, int key) {
    if (root == NULL) return NULL;

    if (root->data < key) {
```

```
root -> right = deleteNode(root -> right, key);
    return root;  }
else if (root -> data > key) {
    root -> left = deleteNode (root -> left, key);
    return root;  }
else
    {
        if (root -> left == NULL && root -> right == NULL)
            {
                delete (root);
                return NULL;
            }
        else if (root -> left == NULL && root -> right != NULL)
            {
                Node * temp = root -> right;
                delete root;
                return temp;
            }
        else if (root -> right == NULL && root -> left != NULL)
            {
                Node * temp = root -> left;
                delete root;
                return temp;
            }
        else {
                Node *prev = root -> left;
                while (prev -> right != NULL)
                        prev = prev -> right;
                root -> data = prev -> data;
```

```
root -> left = deleteNode(root -> left, prev -> data);
return root;
    }
  }
}
```

⑥ k^{th} smallest element in BST :-

Steps ① count = 0

② do inorder traversal (as it will be sorted)

③ instead of printing root, do count++.

④ stop when count = k.

⑦ Validate a BST :-

→ Give each node a range & check if the node's value lies in the range or not. (Pass in function)

initial range = $[-\infty, \infty]$

going left = $[-\infty, root]$

going right = $[root, \infty]$

⑧ LCA in BST :- $(O(\log n)) \approx O(Height)$

→ to find LCA of u & v:

① if at node n, u lies left & v lies in right then n is LCA

② if both lies left ⇒ go left

if both lies right ⇒ go right

③ if n is u or v then also n is LCA

⑨ Construct a BST from a preorder traversal :-

M-1 1) get the inorder by sorting preorder

2) construct unique BT.

Time = $O(n \log n)$

M-2 Maintain upper bound at every node

→ & acc to upper bound, go to either left or right (i.e. use root left right traversal, if upcoming value is less than upper bound then place {if otherwise go back}

Code:
```
Node * build (vector<int> &pre, int &i, int bound) {
    if (i == pre.size() || pre[i] > bound)
        return NULL;

    Node * root = new Node(pre[i++]);
    root -> left = build (pre, i, root -> val);
    root -> right = build(pre, i, bound);
    return root;
}

Node * bstFromPreorder (vector<int> &pre) {
    int i=0;
    return build(pre, i, INT_MAX);
}
```

(10) Inorder Successor in BST :- O(log(H))

→ inorder successor of x is the first value which is greater than x

Code
```
Node * Succ (Node * root, Node * p) {
    Node * successor = NULL;
    while (root != NULL) {
        if (p -> val >= root -> val) root = root -> right;
        else { successor = root;
            root = root -> left; }
    }
    return successor; }
```

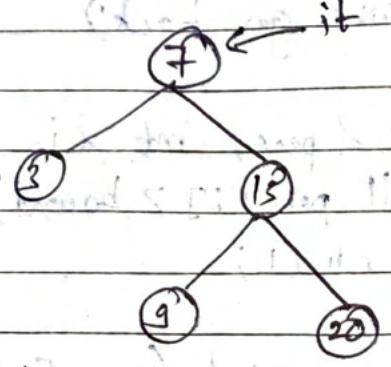
(11) BST Iterator

perform Given steps.

next : inorder successor of current iterator's node

hasNext : True if inorder successor is present.

i/p :



BSTIterator(7)

Steps :

next → 3

next → 7

hasNext → True

next → 9

hasNext → True

next → 15

next → 20

hasNext → False

| 3 | 7 | 9 | 15 | 20 |

it

Space = O(H)

Time ≈ O(1)

⇒ use a stack & follow inorder.

i) push all lefts in stack

ii) next : return top of stack & pop it, also if it have a right then push it along with all its lefts

iii) hasNext : return true if stack is non empty

code :

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

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

int next() {
        Node * temp = myStack.top();
        myStack.pop();
        pushAll(temp -> right);
        return temp -> val;
    }

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

before() can be find out by using opposite logic (i.e. go left & push all rights)

(12) Two Sum in BST :-

→ Given k, find two nodes whose sum is k.

Idea: M-1: store inorder & then solve it using concept of 2 sum problem. (using 2 pointers)

M-2: use BSTIterator

i = next() j = before()

if i+j < k ⇒ i = next()

else ⇒ j = before()

PAGE No
DATE

(13) Recover BST:-

→ In a BST, 2 nodes are swapped now correct the BST.

Idea: as inorder traversal should be sorted, but if ~~two~~ two numbers are swapped then there can be two cases:

(1) Swapped nodes are not adjacent

eg. 3, 25, 7, 8, 10, 15, 20, 5 ← last

first 1st violation 2nd violation

(2) swapped nodes are adjacent

eg. 3, 5, 8, 7, 10, 15, 20, 25

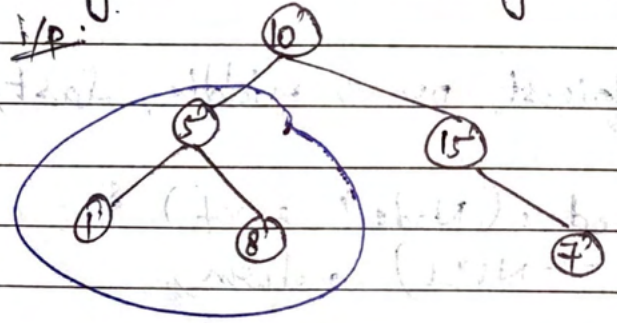
only one violation

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

            inorder(root->left);
            if(prev != NULL && (root->val < prev->val))
            {
                // if its first violation
                if (first == NULL) {
                    first = prev;
                    middle = root;
                }
```

```
        else  last = root;
        }
    prev = root;
    inorder(root -> right);
    }
public:
    void recoverTree(Node* root) {
        first = Middle = last = NULL;
        prev = new Node(INT_MIN);
        inorder(root);
        if (first && last) swap(first->val, last->val);
        else if (first && Middle)
            swap(first->val, middle->val);
            }
    };
```

(14) largest BST in given BT :-

i/p :



o/p : 3

M-1 : at each node, see if its valid BST, if its valid, then count no. of nodes => $O(n^2)$

(2) if a particular node is :
i) greater than the greatest element of its left subtree

ii) smaller than the smallest element in right subtree.

then its a BST

→ to get the size of BST, follow Bottom up approach

→ at each node, maintain {size, smallest, largest}

code: 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(Node * root){

//An empty tree is a BST of size 0

if (!root) return NodeValue(INT_MAX, INT_MIN, 0);

}

auto left = largestBSTSubtreeHelper(root → left);

auto right = largestBSTSubtreeHelper(root → right);

if (left.maxNode < root → val && root → val < right.minNode)

{

return NodeVal(min(root → val, left.minNode),

max(root → val, right.maxNode),

left.maxSize + right.maxSize + 1); }

```
return NodeVal(INT_MIN, INT_MAX,
        Max(left.maxSize, right.maxSize));
    }

public:
    int largestBSTSubtree(Node *root) {
        return largestBSTSubtreeHelper(root).maxSize;
    }
};
```

THANK YOU :)

Keep Learning