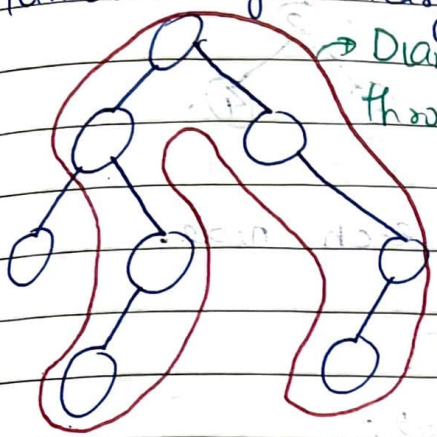
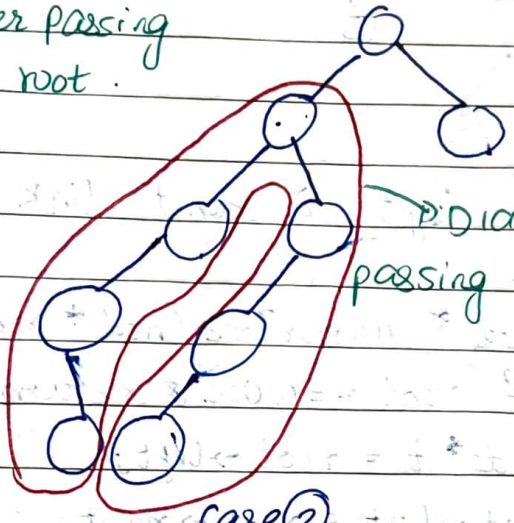
BINARY TREES

Q 3] Diameter of Binary Tree.



→ Diameter passing through root.



→ Diameter not passing through root

case ①

case ②

```
int height (Node* root) {
  if (root == NULL) return 0;
  int l = height (root->left);
  int r = height (root->right);
  int h = 1 + max (l, r);
  return h;
}
```

} Function to calculate height from given Node.

```
int Diameter (Node* root) {
   if (root == NULL) return 0;  → Base condition
   int ans1 = Diameter (root->left);
   int ans2 = Diameter (root->right);
   int d = 1 + height(root->left) + height (root->right);
   return  max ( d, max(ans1, ans2));
}
```

} Hypothesis

} Induction

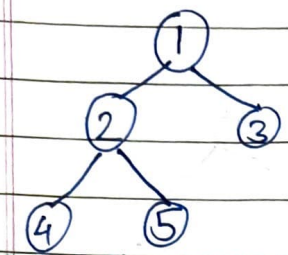
if diameter pass through root node.

if max diameter is present in left or present in right [without counting root Node].

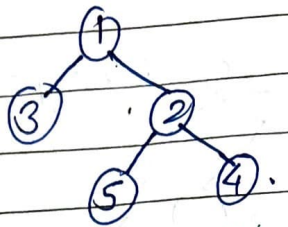
TIME : O(N)

Space : O(log N) → Height of Tree.

Q4] Mirror the given Binary Tree.



INORDER → 4 2 5 1 3



INORDER: 3 1 5 2 4

→ Swap left & right link of each node.

```
treenode* mirrorTree(node* root){.
BASE ← if(root==NULL) return root;
INDUCTION { node* t = root->left;
            root->left = root->right;
            root->right = t;
          } swapping links of left & right nodes.
HYPOTHESIS { if(root->left) mirrorTree(root->left);
             if(root->right) mirrorTree(root->right);
             return root.
           }
```

TIME: O(n) (swapping n nodes)

Q INORDER TRAVERSAL.

- Recursive → VS code
- Iterative → Using Stack → VS code. → TIME: O(n) SPACE: O(1)
  - Without using Stack. (Morris Traversal)
    - TIME: O(n)
    - SPACE: O(1)

PREORDER TRAVERSAL.

- Recursive
- Iterative → Using Stack.
  - Without using Stack (Morris Traversal)

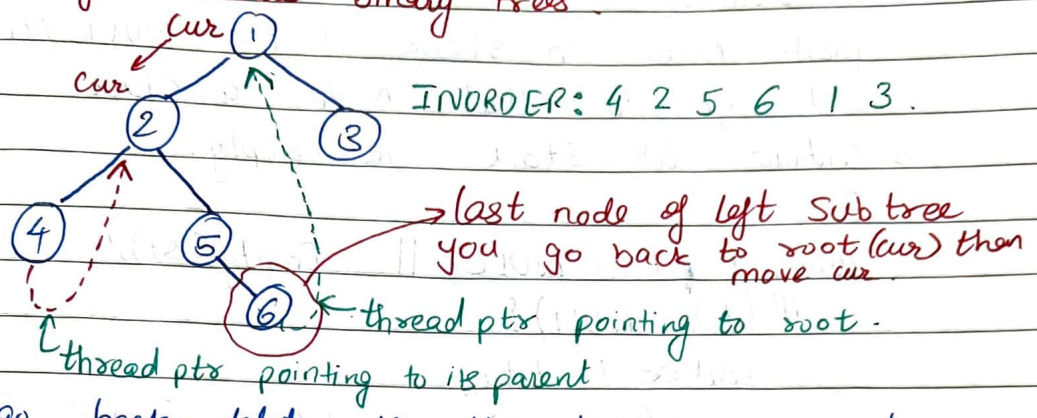
POSTORDER TRAVERSAL.

- Recursive
- Iterative → using 2 stack
  - using 1 stack
  - Morris

Page No.
Date

Q] Iterative Inorder traversal Without Stacks. [Morris Traversal]

Ans] Uses concept of threaded binary trees.



Once you go back, delete the thread ptrs you made.

① Set root as curr

② Go to rightmost node of left subtree & point to cur (root)

③ Move root to left & repeat this process. Untill you reach left node.

④ Make cur back to its root.

⑤ Cut off that thread link. & move cur to right.

Code:

```
vector<int> ans; Node* cur = root;
while (cur != NULL){
    if (cur->left == NULL){                    // if no left subtree present; print root & move to right.
        ans.push_back(cur->value);
        cur = cur->right;
    } else {
        Node* prev = cur->left;                // go to rightmost node of left subtree
        while (prev->right != NULL && prev->right != cur){
            prev = prev->right;
        }
        if (prev->right == NULL){              // create thread to cur & move cur to left
            prev->right = cur;
            cur = cur->left;
        } else {                               // if thread already present remove thread, push cur in ans & cur right
            prev->right = NULL;
            ans.push(cur->value);
            cur = cur->right;
        }
    }
}
return ans;
```

TIME: O(n)

SPACE: O(1).

In preorder ans.push(cur->value) ↑ ✓ add this

✗ remove this

Morris-Postorder
↳ Same as Inorder but Complete opposite [see vs code]

Q6] Iterative Preorder Traversal & Inorder Traversal
↳ Stack.

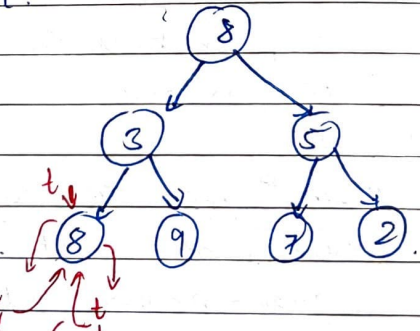
→ push Nodes in stack; if t == NULL; pop from stack and go to that Node & check left side.
→ continue till Stack is empty

Code:

```
while ( t != NULL || isEmpty(st)){
  if (t != NULL) {
    cout << t->data;
    st.push(t);          TIME: O(n)
    t = t->left;
  }
  else {
    t = st.pop;          If iterative Inorder
    t = t->right;        cout<< t->data;
  }
}
```

Q7] Iterative Post Order Traversal (see 29 abdul bari if not understood)

```
while (t != NULL || st.size != 0){
  if (t != NULL){
    st.push (t);
    t = t->left;
  } else {
    temp = t st.top();
    st.pop();
    if (temp > 0) {          → push -temp
      st.push (-temp);
      t = temp->right
    }
    else {
      cout << temp->data
      t = NULL;
}}}
```



To go back to 8 after right child; we need to push it again.
if pushed is +ve & t is NULL; we know we have got null on left child
if pushed is -ve & t is NULL; we know we have to go to right child.

Q8] Left View
Q9] → Create a
→ keep i
→ after e
again
NULL
after
→ mak
left or
Recursiv
→ Maint
if c
node
→ updat
→ recu
void

Page No.
Date

Q8] Left View of Binary Tree / Right View.

→ Create queue, Insert NULL.
→ keep inserting nodes like level order
→ after each level; pop q.front; store front in ans and again push NULL in queue
NULL acts as an indicator that the first element after null will be in left/right view.
→ make root on left & right child. first acc to left or right view.

Recursive

→ Maintain curr height & max height
if curh will be more than maxh; then that node will be appearing first time in that view.
→ update maxh as curh.
→ recursively call solve(root→left, maxh, curh+1, v) (next level)

```
void solve(Node* root, int &maxh, int curh, vector<int> &v) {
    if (root == NULL) return;
    if (curh > maxh) {
        v.push_back(root->val);
        maxh = curh;
    }
    solve(root->left, maxh, curh+1, v);
    solve(root->right, maxh, curh+1, v);
}
```

solve(root→right, " , " , ") ← in right view call this first

Page No.
Date

Q10] Top View of Tree.

→ Make a queue <pair<Node*, int>> q

→ Make a Map<int, int> mp

→ Check if already that level's data is p

→ create iterator for q.front l==2

pop from queue

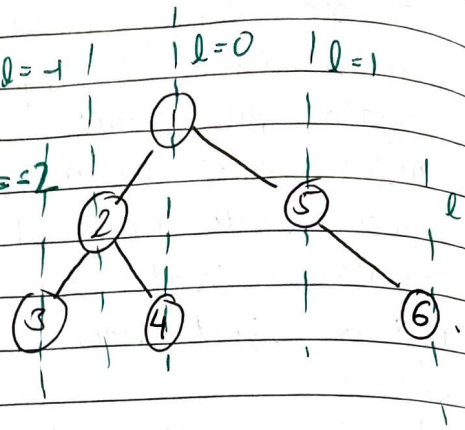
find if that node's line is present in map; if no add it to the map.

mp[line] = t->data.

→ if left is present; push into q (t->left, line-1)

→ if right is present; push into q (t->right, line+1)

→ create iterator for map and pushback into ans vector



Q11] Bottom view of tree.

→ Same as top view but here you need to print all nodes which come last in that level. so if you get that line in map; update with the new node we get in the same line.

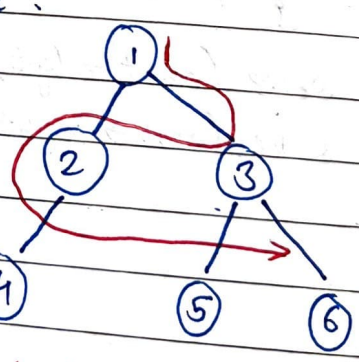
Q12] Zig Zag traversal of binary tree.

→ Solve using deque <Node*> dq.

create flag = 0.

push root & after that NULL in dq

NULL acts as indicator that level is completed. so push in dq from other side.



1 3 2 4 5 6.

if NULL found at dq.front()

v.push_back(), dq.push_back(t->left), dq.push_back(dq->right)

dq.popfront()

if NULL found at dq.back()

v.push_back(), dq.push_front(t->right), dq.push_front(t->left),

dq.pop_back()

Change flag everytime NULL is found in front or back.

Q13] Balan

→ write

if

if lh

else

bool

Q14] D

→ N

→ t

→ C

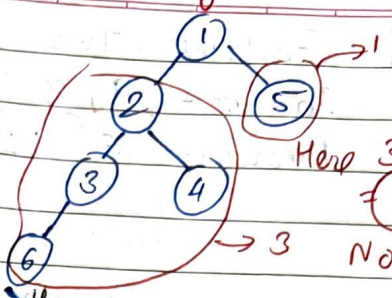
Q15

Page No.
Date

[13] Balanced Binary Tree.

→ write height funct^n for calc height.
if abs(lh - rh) >= 1 return -1
if lh || rh == -1 return -1.
else return height.

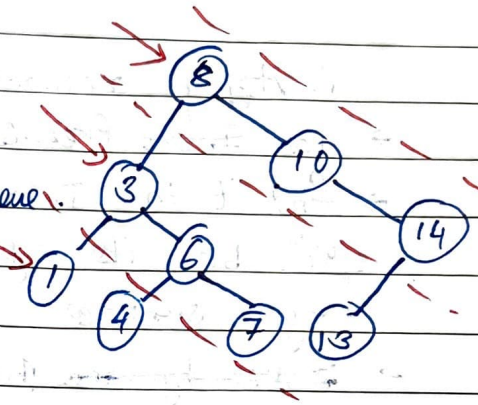
bool balanced (Node * root) {
return height(root) != -1;



Height of left Subtree
- " " right "
should be <= 1

[14] Diagonal Traversal.

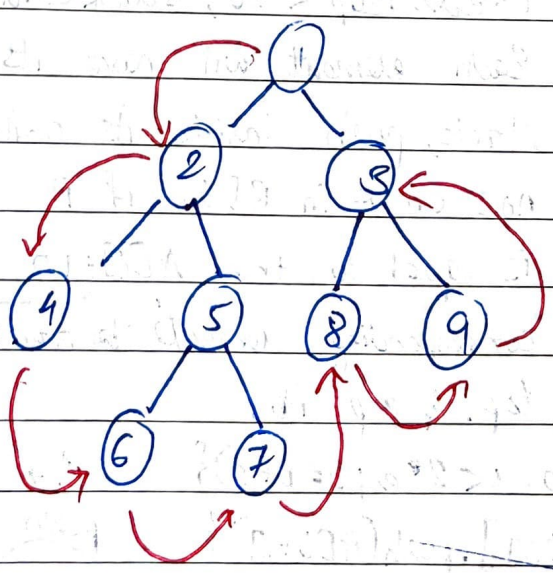
→ Make queue, push root. ~~q.push~~
→ Keep pushing left nodes if exist in queue.
→ Observe; Right childs are printed
while (!q.empty) { q.push(root).
Node * t = q.front()
q.pop();
while (t) {
if (t -> left) {
q.push(t -> left)
}
v.push_back(t -> data)
t = t -> right)
}



8 10 14 3 6 7 13 1 4.

[15] Boundary Traversal.

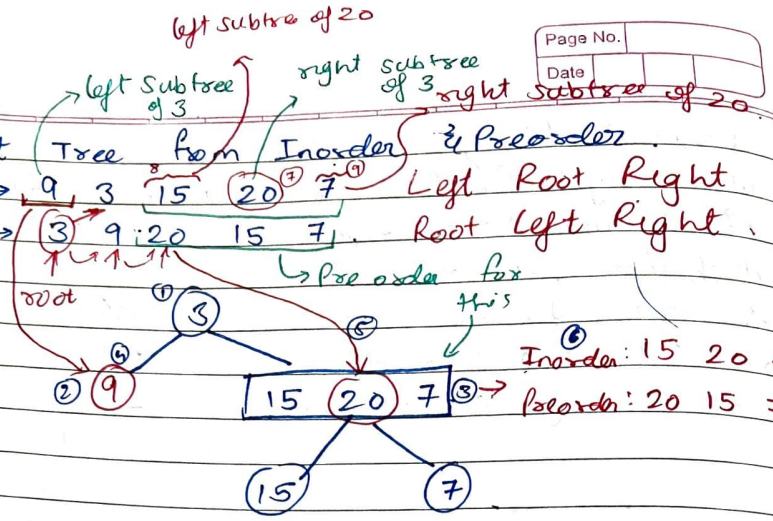
So code (understand hoga).
① Traverse left boundry.
② Traverse leaf nodes by Inorder.
③ Traverse Right boundary without leaf nodes & ~~rev~~ push reverse ans in ans vector.



Q19] Construct Tree from Inorder & Preorder.

Inorder → 9 3 15 20 7 . Left Root Right.

Preorder → 3 9 20 15 7 . Root Left Right.



∴ Apply recursion on left & right subtree.

→ in ~~Inorder~~ Preorder, first element you get root.

→ find it in Inorder (using maps). on left of that will be its left subtree & on right of that will be right subtree.

→ ~~move by one~~ The pre order of those subtrees will be present in number of nodes present after the current root. (see above 20 15 7 is preorder as in preorder list ~~after~~ we have to see 3 nodes after 20 (including 20).

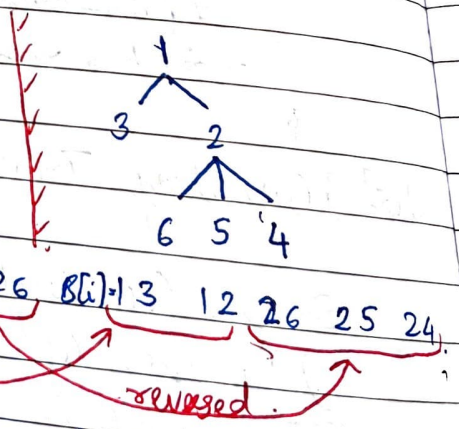
→ Repeat recursively for left & right subtrees.

Q24] Check Mirror in N-ary tree.

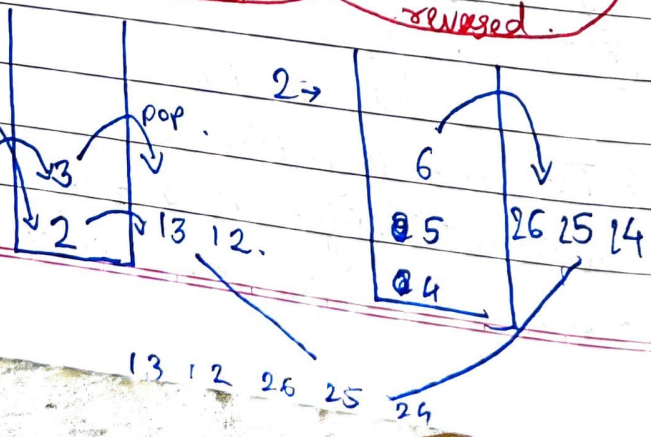
→ Create Map < int, stack<int>> mp.

Each element will have its stack, push inside its children

now check in B[i] if it is equal to the corresponding elements stack top. & pop it.




```
for(int i=0, i<2*e; i=i+2){
    mp[A[i]].push(A[i+1])}
for(
    ){
if(mp[B[i]].top() != B[i+1]){
    return 0;
} mp[B[i]].pop();} return 1;
```



Q23] Duplicate

→ Create a key will and int string.

→ if root ==

Eg

∴ We

∴ if

→ if we

→ first then

→ Incre

→ if

Q25] Su

→ Cre fir

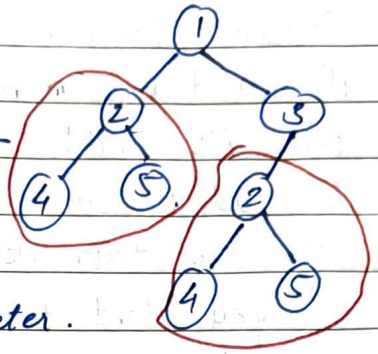
Se if

if

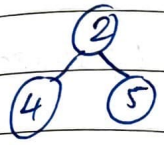
Page No.
Date

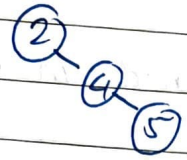
[923] Duplicate subtree of size more than 2.

→ Create a map <string, int>;
key will be string of 2 4 5 ←
and int will keep count of the string.



→ if root == NULL; return "$" as delimeter.



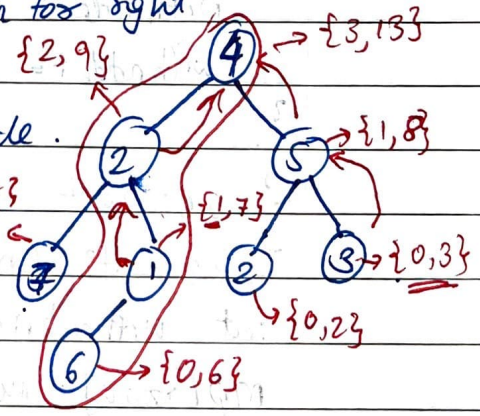


; String would be 2 4 5 in both and it would consider as duplicate; but it is not.
∴ We add delimeter if child does not exist
∴ String will become 2 4 5 and 2 $ 4 $ 5 ∴ different

→ if we get leaf node we add it to string.
→ first add root's data in string, then recursive call on left then recursive call on right.
→ Increase the count for that string in map.
→ if any value in map is >= 2; duplicate exist.

[925] Sum of nodes on longest path from root to leaf.

→ Create 2 vectors 1 for left subtree other for right
first index of vector will store height;
second index will store sum up to that node.




```
if [a[0] > b[0])
    return { a[0]+1, a[1] + root->data };
if (b[0] > a[0])
    return { b[0]+1, b[1] + root->data };
else {
    return { a[0]+1, max(a[1], b[1]) + root->data };
```

→ if height equal from both left & right choose one with max sum and increase height by 1.

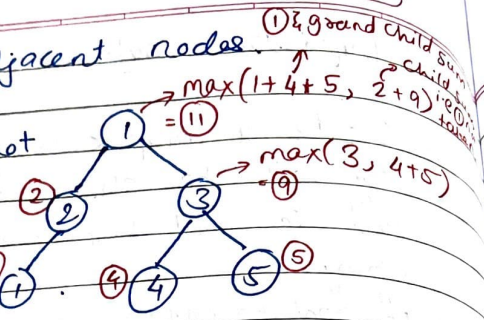
Page No.
Date

Q28] Maximum Sum of Non-adjacent nodes.

MP

Dynamic Programming.

→ if you take a node; you cannot take its childs or parent to add to sum;
you can take its grandchilds.



→ if you dont take node you can tak its childrens

→ Store the sum of each node in map → Optimization.

```
map<Node*, int> mp;
if(!root) return 0;
if(mp[root]) return mp[root]; → if we ghed already stored that data in map; no need to call recursive again on that node :. [optimised].
```

If we take that node
Take its grandchilderen.

```
int withnode = root->data;
if(root->left){
    withnode += getSum(root->left->left);
    withnode += getSum(root->left->right);
}
if(root->right){
    withnode += getSum(root->~~left~~->~~right~~);   (right : left)
    withnode += getSum(root->right->right);
}
```

TIME: O(n)
SPACE: O(n)

If we dont take that node
Take its children.

```
int withoutnode = getSum(root->left) + getSum(root->right);
mp[root] = max(withnode, withoutnode); → Store max of with or without in map.
return mp[root];
```

Q29] Print

→ Print
node
→ Make
the
→ Add
→ Start
and
→ if
prin
→ pop
if

[29] Print all 'k' Sum paths in Binary Tree.

→ Print all paths downwards from that node whose sum = k.

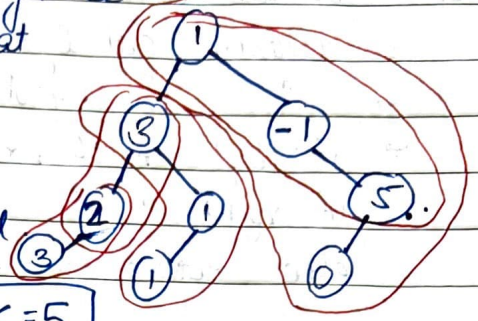
→ Make a vector path to push all the nodes.

→ Add nodes in vector acc to preorder traversal

→ Start from behind of vector and keep adding sum

→ if at any pt sum = k, print path from that point to vector end

→ pop from vector.



k = 5.

1 3 2 | 3 1 1 | 1 -1 5 | 1 -1 5 0 | 2 3

```
if (!root) return;
v.push_back(root->data);
Solve(root->left, v, k);
Solve(root->right, v, k);
int sum = 0;
for(int j = v.size(); j >= 0; j--){
    sum = sum + v[j];
    if (sum == k){
        for(int m = j; m <= v.size(); m++){
            cout << v[m] << " ";
        }
        cout << endl;
    }
}
v.pop();
}
```

recursive on all nodes — 2 for loops for vector

TIME: $O(n * h * h)$

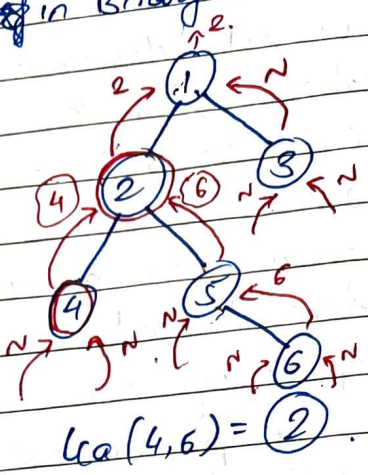
as max size of v can be h (height)

SPACE: $O(h)$

Max stored in v is h (height)

Q 30] Lowest Common Ancestor of in Binary Tree

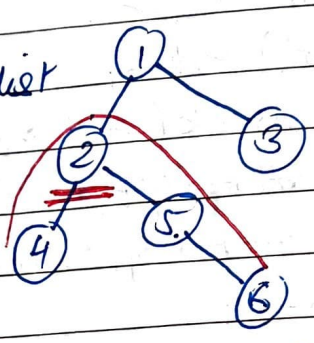
→ recursive call left
→ " " right.
→ if left == NULL, return right
if right " " left
else return root.



lca(4,6) = 2.

Q31] Min dist bet^n 2 nodes.

→ We find LCA as minimum dist will always be through LCA.
→ Make another funct^n to find dist bet^n 2 nodes.
↳ find^n dist (LCA, a)
y = dist(LCA, b)
return x + y (-2) → see eg: 4 nodes - 2 = 2.



dist(4,6) = 2.