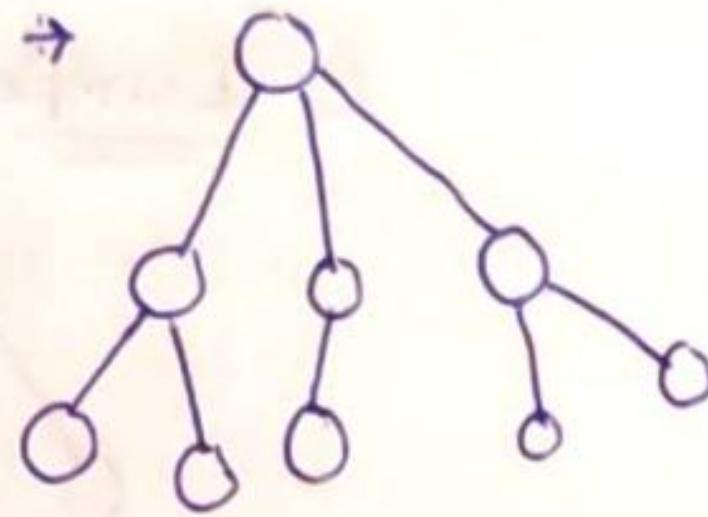


① Binary Trees →

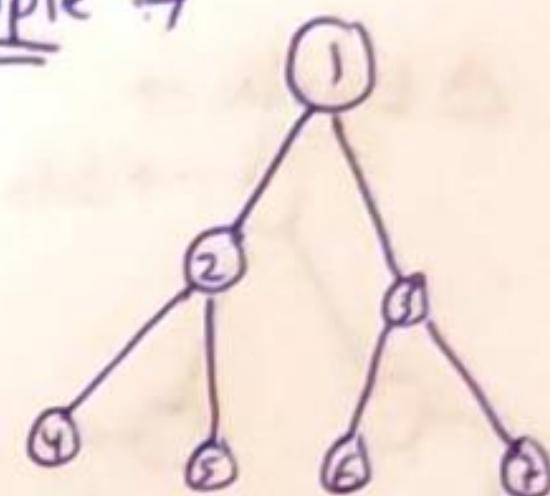
- ◻ Tree → A tree is a non-linear datastructure, where one node can get connected to multiple other nodes.

◻ Example →



① Note → A Binary tree is a tree where child of any node can be (≤ 2), in number.

◻ Example →



② Important terms related to Tree →

i) Node → It is an element of our tree which is responsible for storing data.

ii) Root → Topmost node of tree is called as its root.
Here it node ①.

iii) Parent → The nodes having subnodes is termed as parent.

Example → ① is parent of ②, ③, ④.

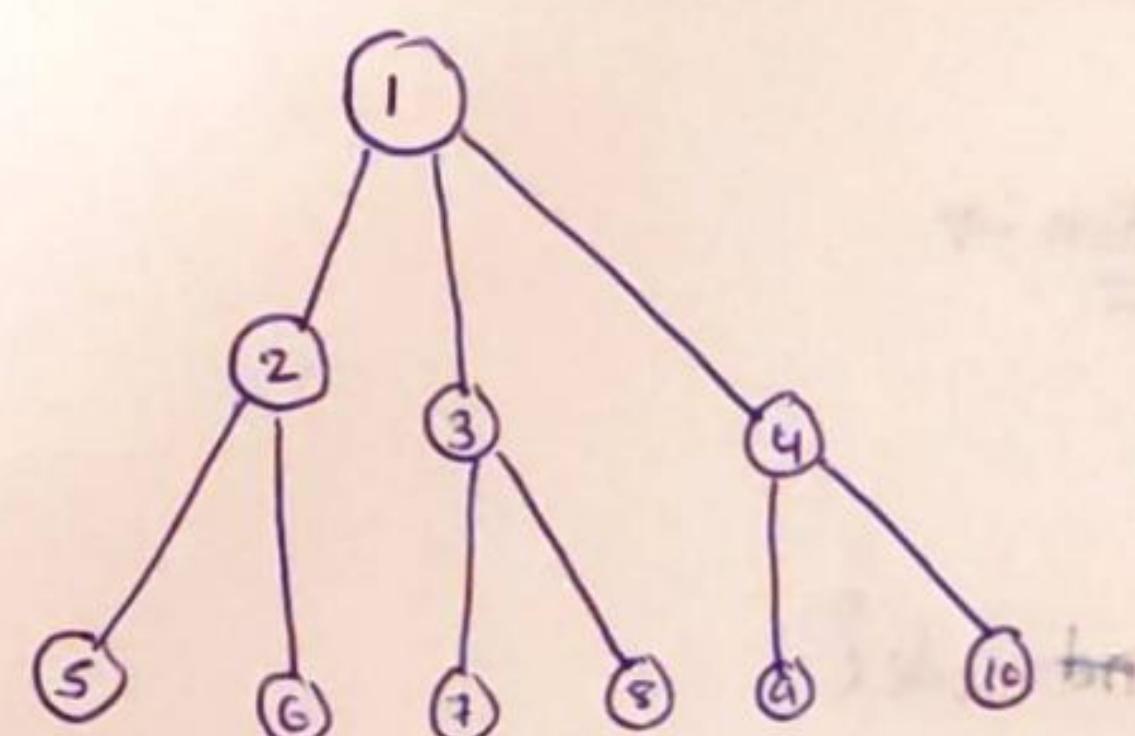


Fig 1.1.

vii) Descendants → It is just opposite of ancestors.

Example → Descendant of ②, is ⑤ and ⑥.

v) Siblings → Two or more nodes having same parent are termed as siblings.

Example → ②, ③, ④ nodes are siblings of each other.

viii) Leaf → Leaf node are those nodes which do not have children.

Example - 5

vi) Ach

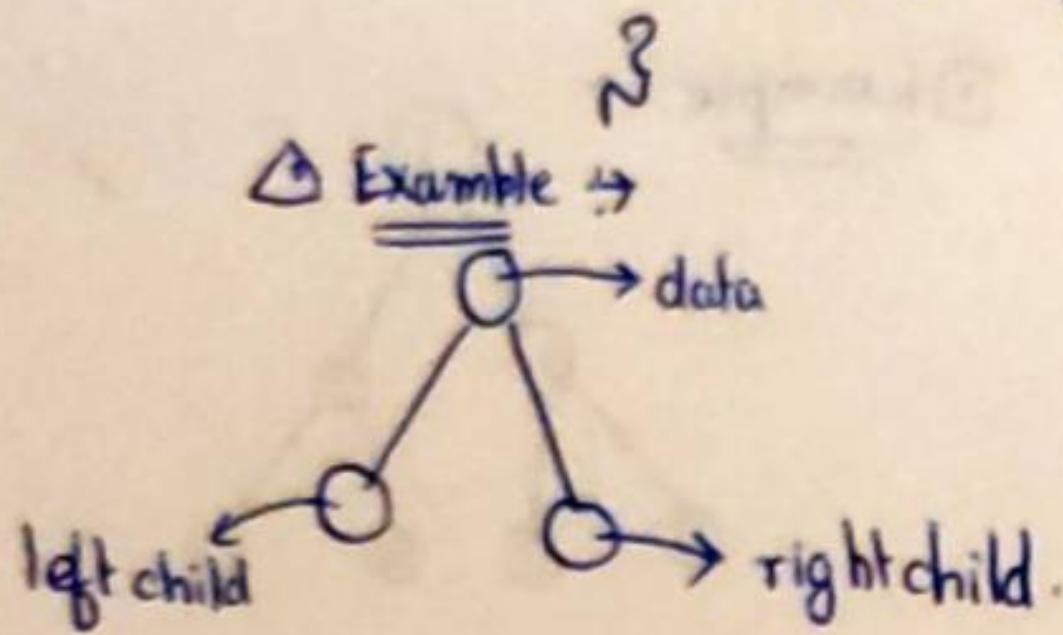
Ancestors → An ancestor is any other node on path from that node to the root.

Example → ①, ③ is ancestor of ⑦

① For Binary trees node structure:

△ Snippet → node :

```
int data;
int *left; → For left child
int *right; → For right child.
```



① Notes: → □ Here child can be node
or, even NULL.

② Tree Creation:

Code →

```
class node {
public:
    int data;
    node *left;
    node *right;

    node (int d) {
        this->data = d;
        this->left = NULL;
        this->right = NULL;
    }
};
```

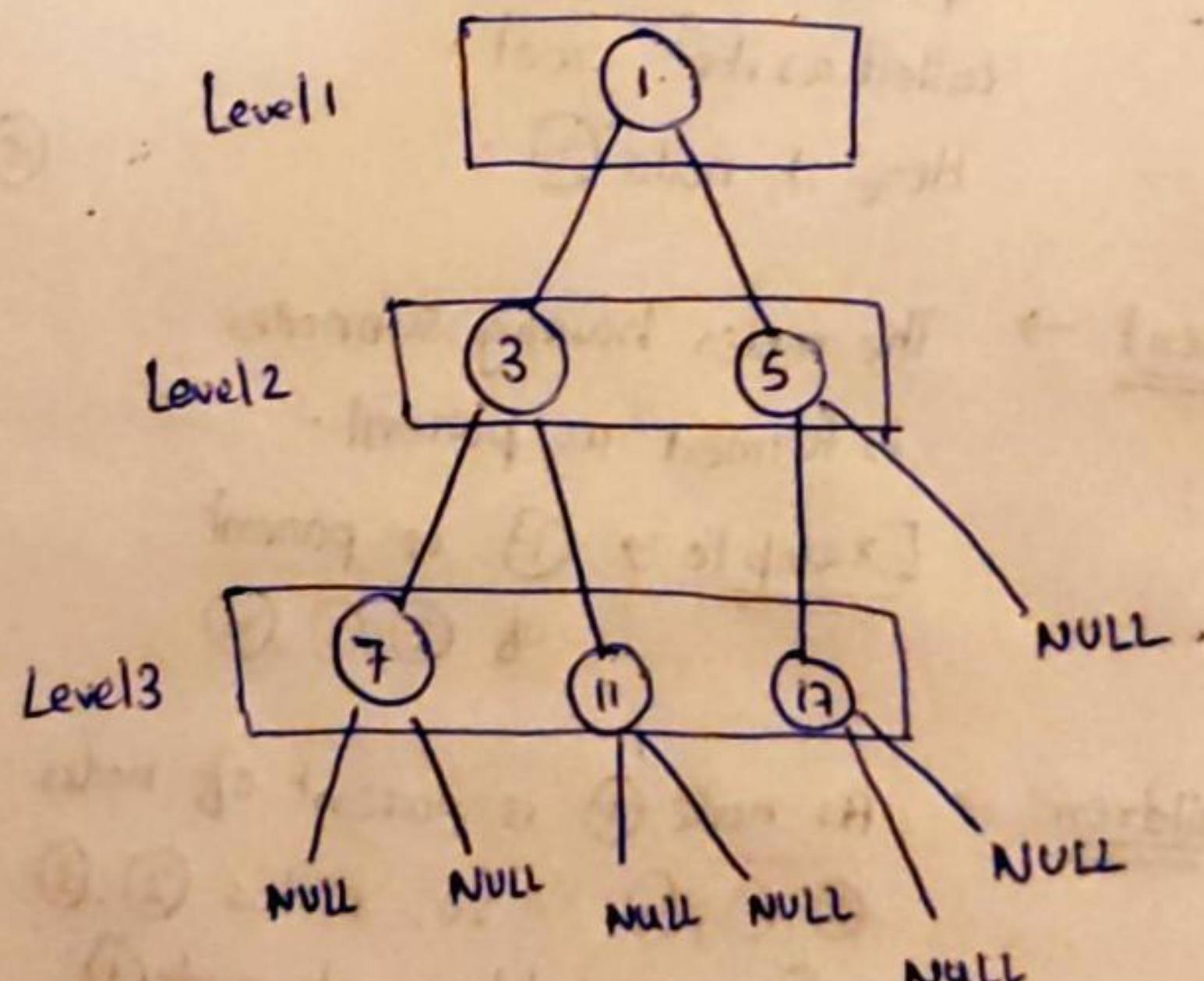
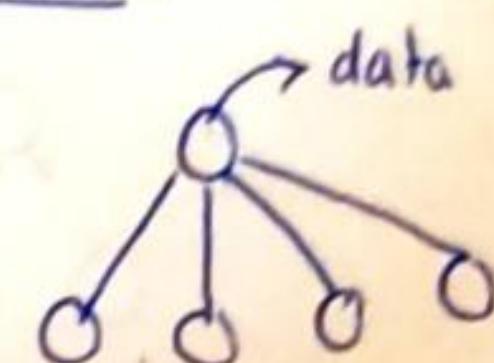
```
node * buildTree (node * root) {
    cout << "Enter data: ";
    int data;
    cin >> data;
    root = new node(data);
    if (data == -1) {
        return NULL;
    }
```

① For n-ary tree node Structure:

△ Snippet → node :

```
int data;
list<node*> child;
```

△ Example →



input :→ 1, 3, 7, -1, -1, 11, -1, -1, 5, 17, -1, -1, -1



Here -1 denotes NULL, and according to input our recursion works.

```

cout << "Enter data for inserting in left " << data;
cout << endl;
root -> left = buildTree(xroot -> left);

```

```

cout << "Enter data for inserting in right " << data;
cout << endl;
root -> right = buildTree(xroot -> right);
return root;
}

```

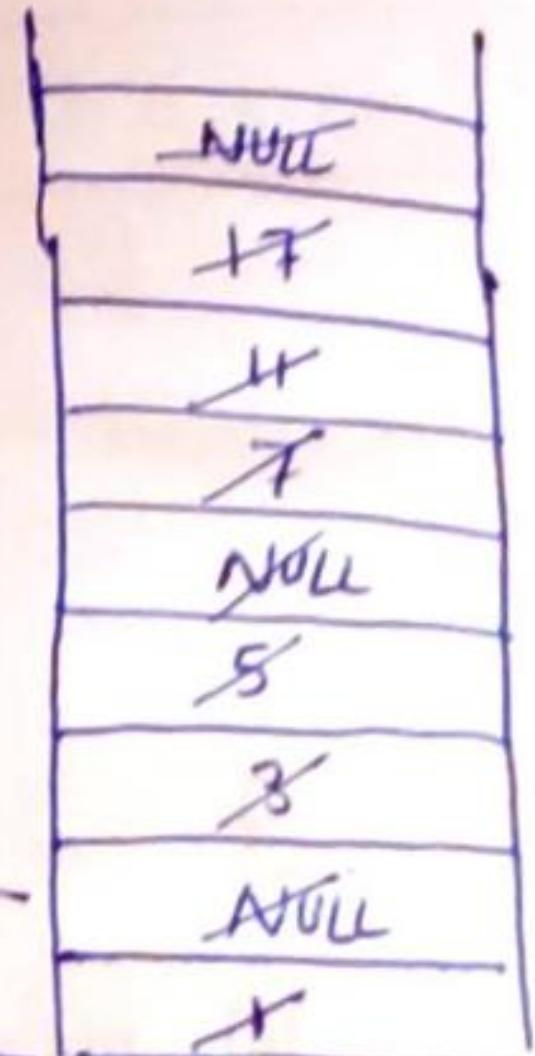
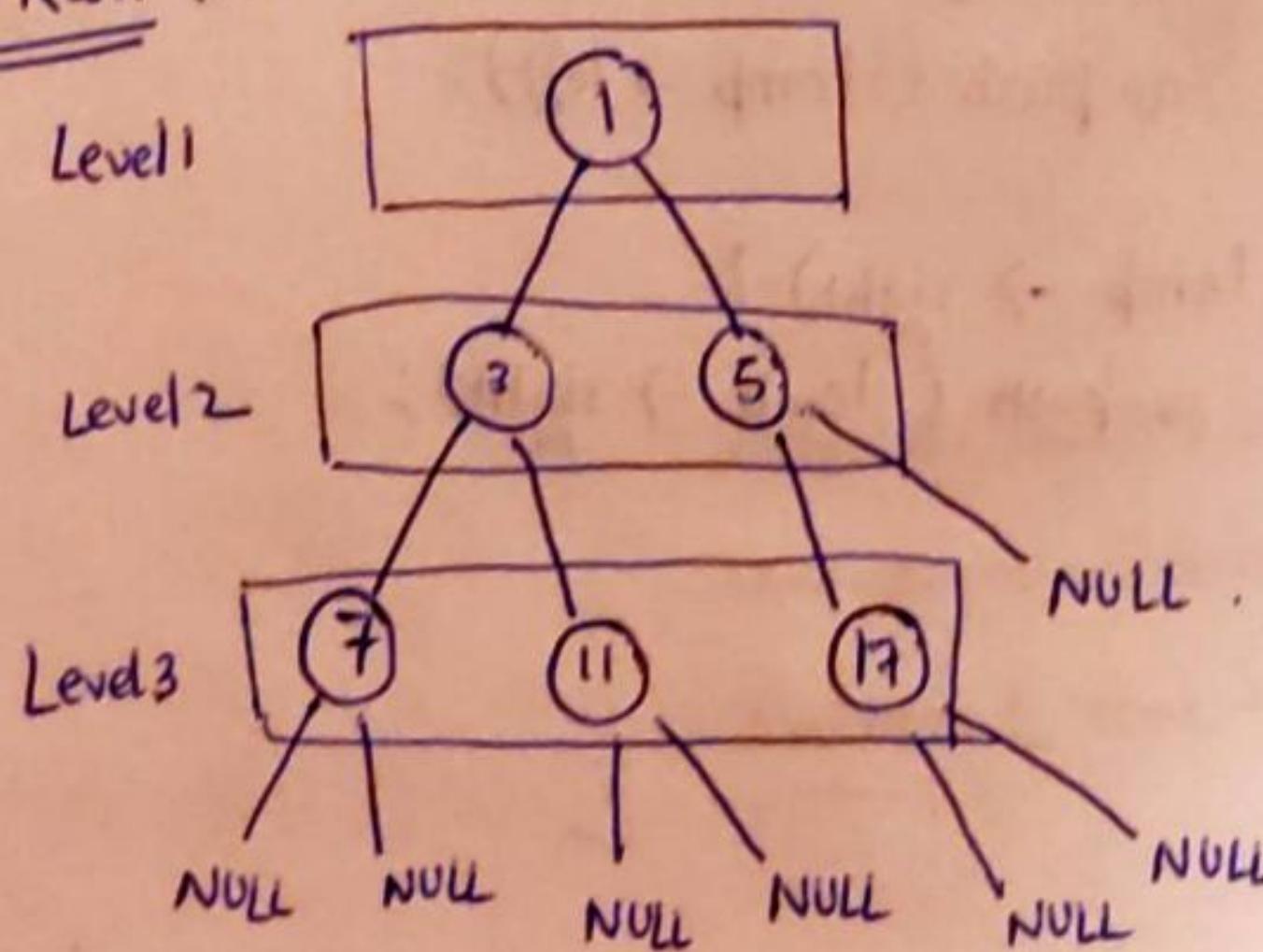
```

int main()
{
    node* root = NULL;
    root = buildTree(root);
    return 0;
}

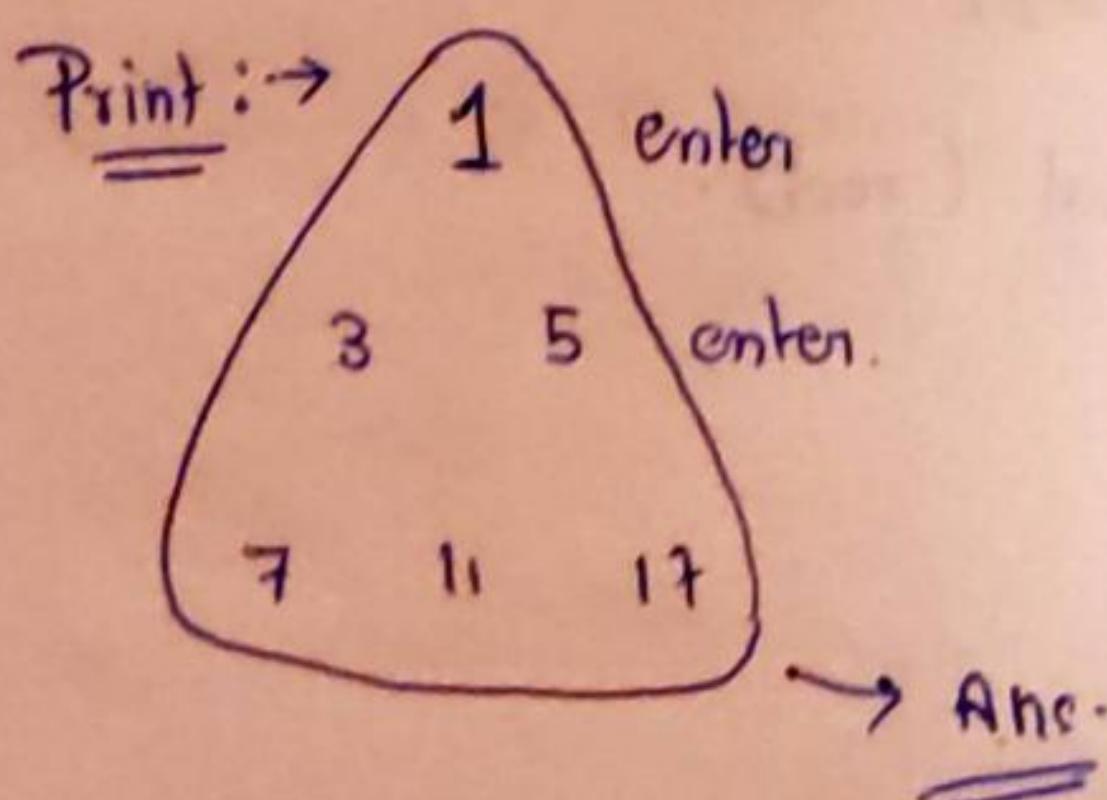
```

① Level Order Traversal \Rightarrow

Dry Run \Rightarrow



When our level ends we push NULL.
 For NULL we do enter instead of printing anything.



□ Code →

```
void level_Order_Traversal (node * root) {
    queue<node*> qv;
    qv.push(root);
    qv.push(NULL);

    while (!qv.empty()) {
        node * temp = qv.front();
        qv.pop();

        if (temp == NULL) {
            cout << endl;
            if (!qv.empty()) {
                qv.push(NULL);
            }
        } else {
            cout << temp->data << " ";
            if (temp->left) {
                qv.push(temp->left);
            }
            if (temp->right) {
                qv.push(temp->right);
            }
        }
    }

    int main() {
        node * root = NULL;
        root = buildTree (root);
        Level_order_Traversal (root);
    }
}
```

① Types of Tree Traversals →

- i) Inorder Traversals.
- ii) PreOrder Traversals.
- iii) PostOrder Traversals.
- iv) Level Order Travels.
- v) Morris Traversals.

- vi) Zig zag Traversal
- vii) Boundary Traversal.
- viii) Vertical Order Traversal.
- viii) Morris Traversal.

① Note →

- All traversal techniques take $O(N)$ time and $O(N)$ space complexity.
- Only morris traversal takes $O(N)$ time and $O(1)$ space.

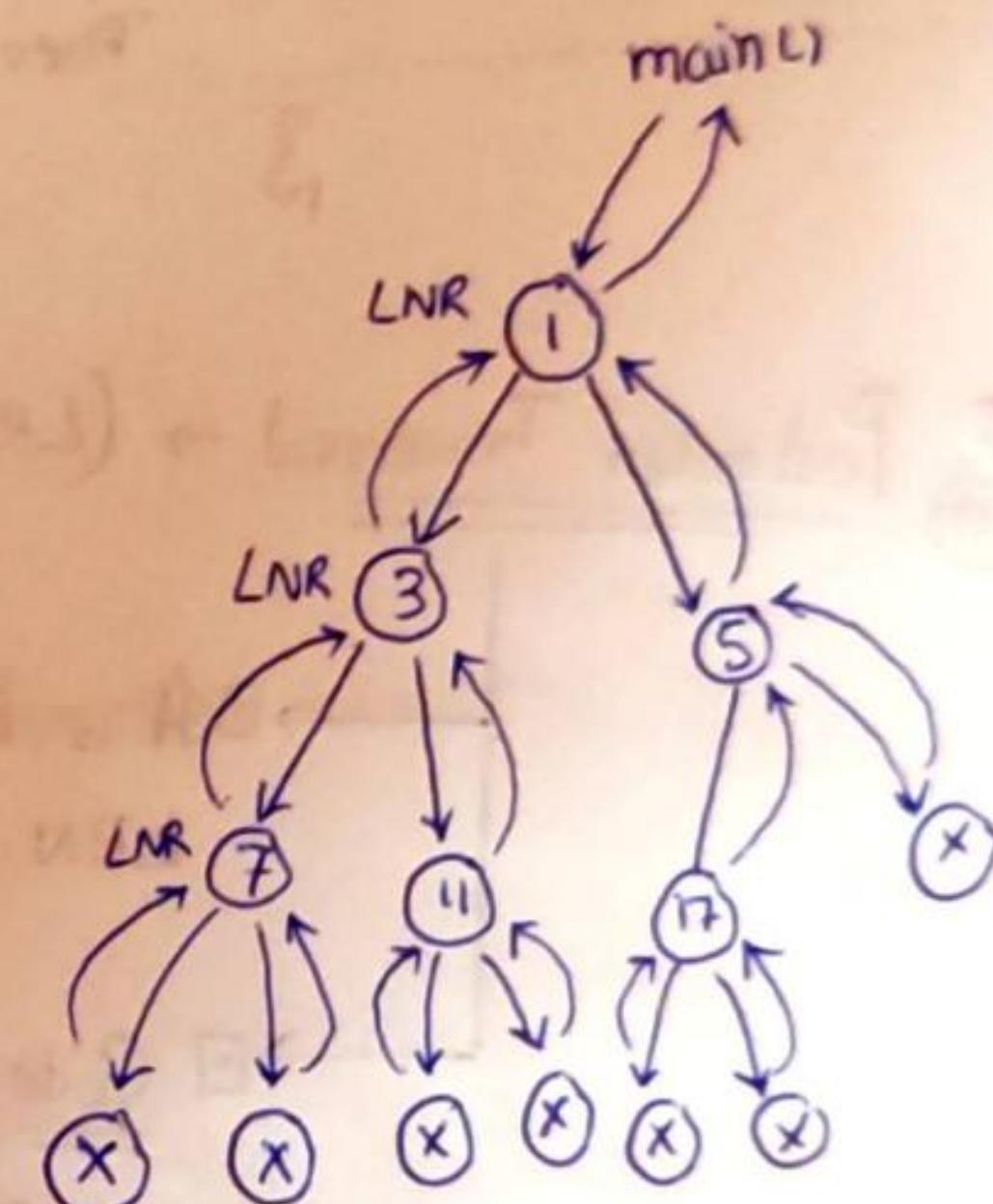
① Inorder Traversal → (LNR)

- □ It is based on logic LNR ---
- ① L: Left node traverse.
- ② N: Printing the node.
- ③ R: right node traverse.

□ Code →

```
void Inorder(node* root){  
    if (root == NULL){  
        return;  
    }  
    Inorder (root->left);  
    cout << root->data << " ";  
    Inorder (root->right);  
}
```

23



7 3 11 1 17 5

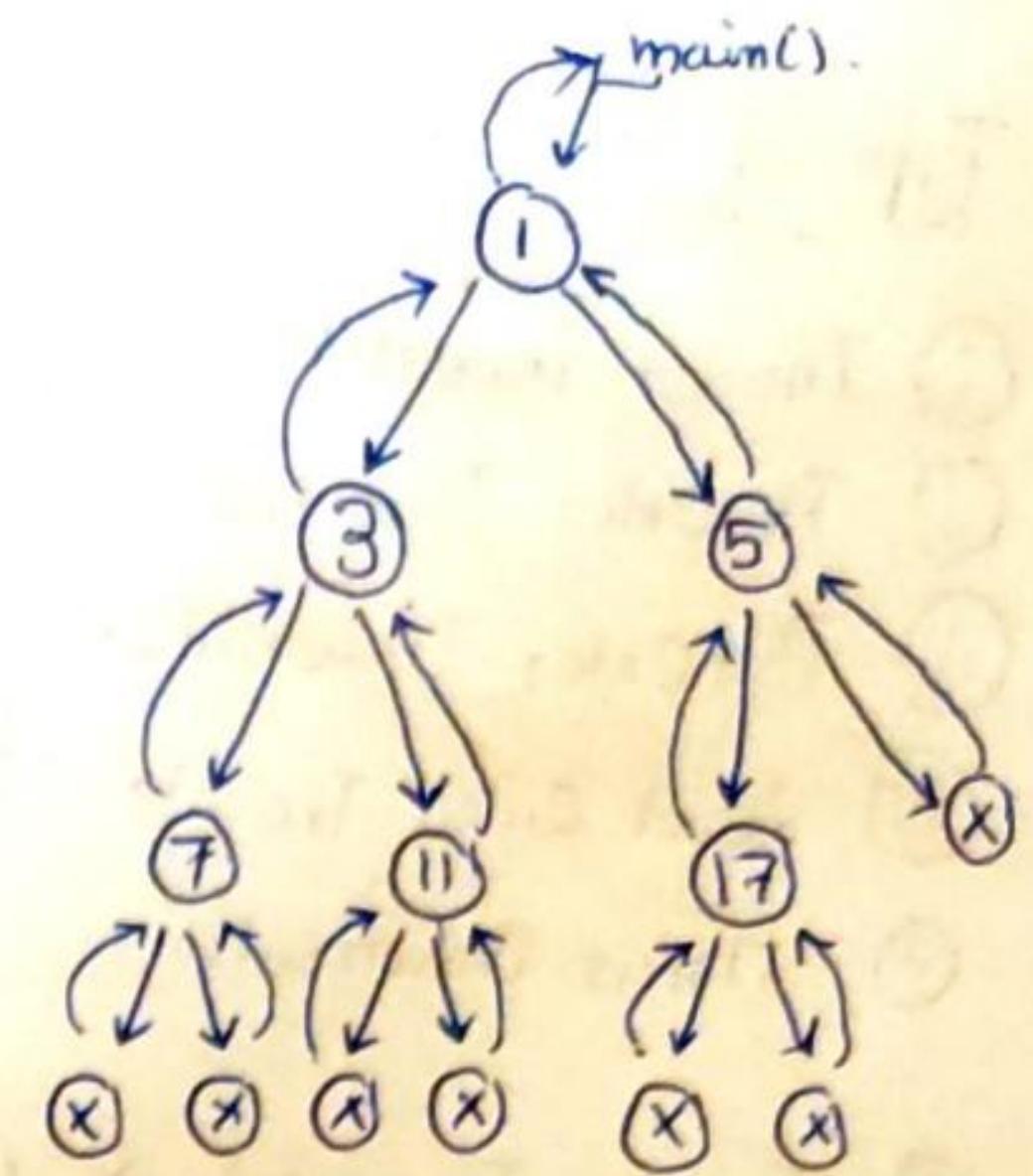
Inorder traversal

② Preorder Traversal → (NLR)

- □ It basically follows the logic of NLR.

→ Code →

```
void Preorder (node* root) {
    if (root == NULL) {
        return;
    }
    cout << root->data << " ";
    Preorder (root->left);
    Preorder (root->right);
}
```



1	3	7	11	5	17
---	---	---	----	---	----

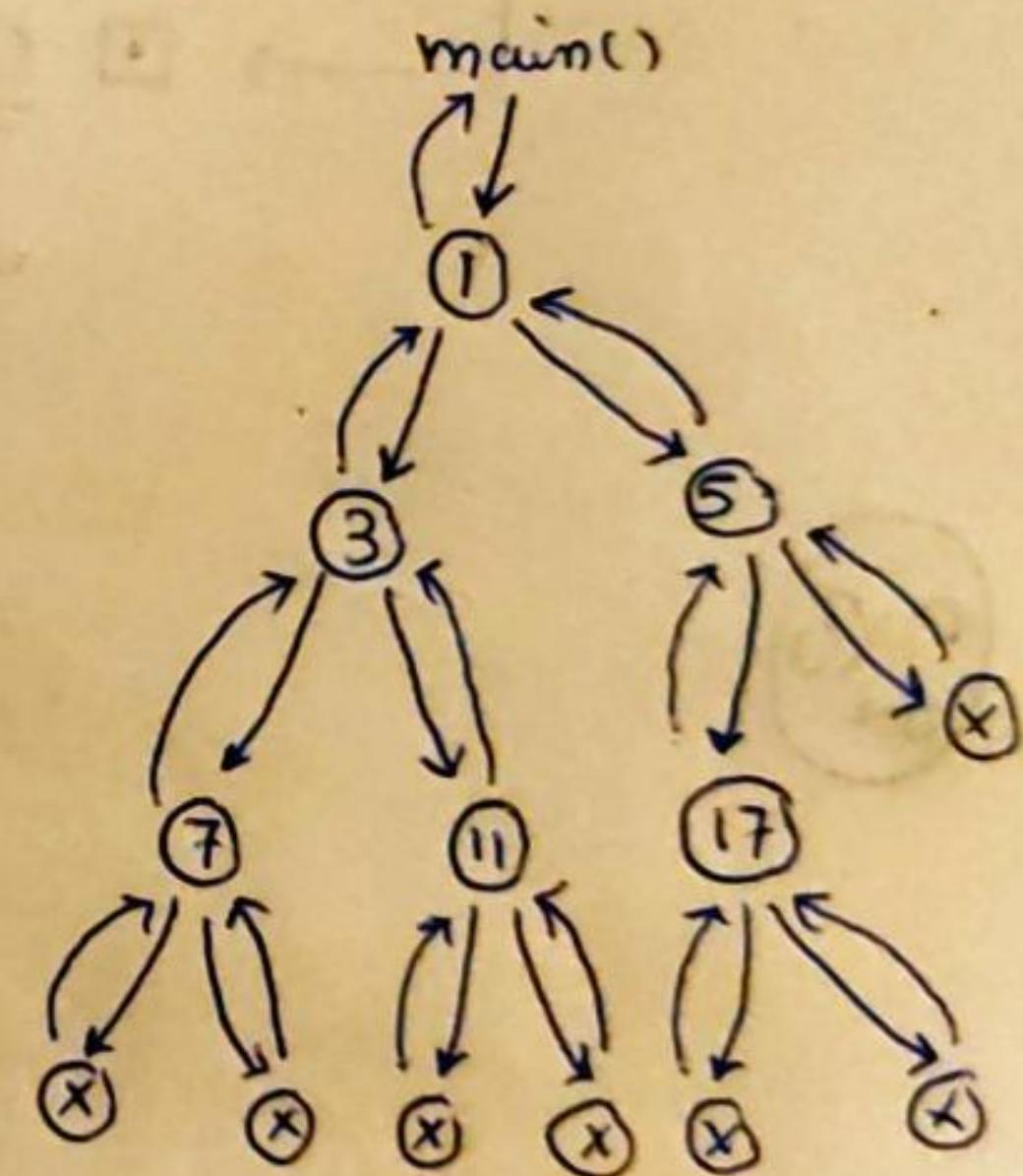
↓
Preorder traversal.

(iii) Postorder Traversal → (LRN)

→ It is based on the logic of LRN.

→ Code →

```
void Postorder (node* root) {
    if (root == NULL) {
        return;
    }
    Postorder (root->left);
    Postorder (root->right);
    cout << root->data << " ";
}
```



7	11	13	17	5	1
---	----	----	----	---	---

↓
Postorder traversal.

① Tree Creation Using Level Order Traversal →

◻ Code →

```
void level_order_create (node* &root) {
    queue<node*> q;
    int data;
    cin >> data;
    root = new node(data);
    q.push(root);

    while (!q.empty()) {
        node* topnode = q.front();
        q.pop();

        cout << "Enter left node for " << topnode->data;
        int leftdata;
        cin >> leftdata;
        if (leftdata != -1) {
            topnode->left = new node(leftdata);
            q.push(topnode->left);
        }

        cout << "Enter right node for " << topnode->data;
        int rightdata;
        cin >> rightdata;
        if (rightdata != -1) {
            topnode->right = new node(rightdata);
            q.push(topnode->right);
        }
    }
}
```

Step 1: Create a root node and push it into the queue.

Step 2: Until our queue is empty, we will store it top variable node data and pop it out.

Step 3: We will take two status one for left and other right node of our top node. If data is not -1, then using it we will create our left and right nodes and also push them into the queue.

⑥ Height of a tree \rightarrow It is the longest path between the root node and a leaf node.

■ Time and Space complexity

Time complexity = $O(\text{height})$.

Space complexity = $O(N)$.

■ Code \rightarrow

```
void height ( node* root ) {
```

```
    if (root == NULL) {
```

```
        return 0;
```

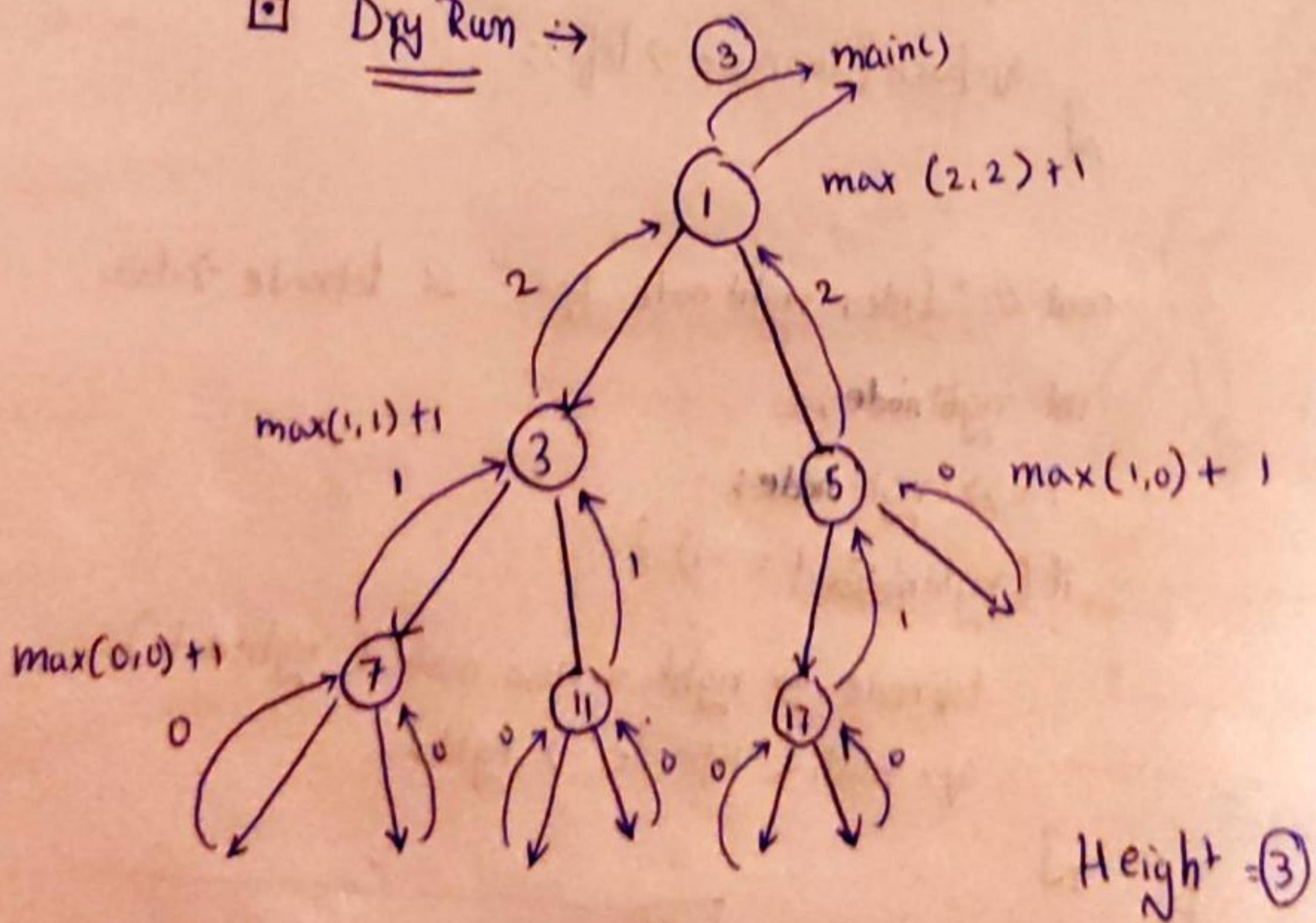
```
    int left = height ( root -> left );
```

```
    int right = height ( root -> right );
```

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

\leftarrow Answer

■ Dry Run \rightarrow



⑦ Note: \rightarrow

For a skew tree

i.e., where only left or
only right nodes are
present.

$$\therefore T.C = O(N)$$

- Counting the number of leaf present in a tree ↗

□ Code ↗

```
void count_leaf (node* root, int & count) {
```

```
    if (root == NULL) {
```

```
        return;
```

```
}
```

```
    count_leaf (root->left, count);
```

```
    count_leaf (root->right, count);
```

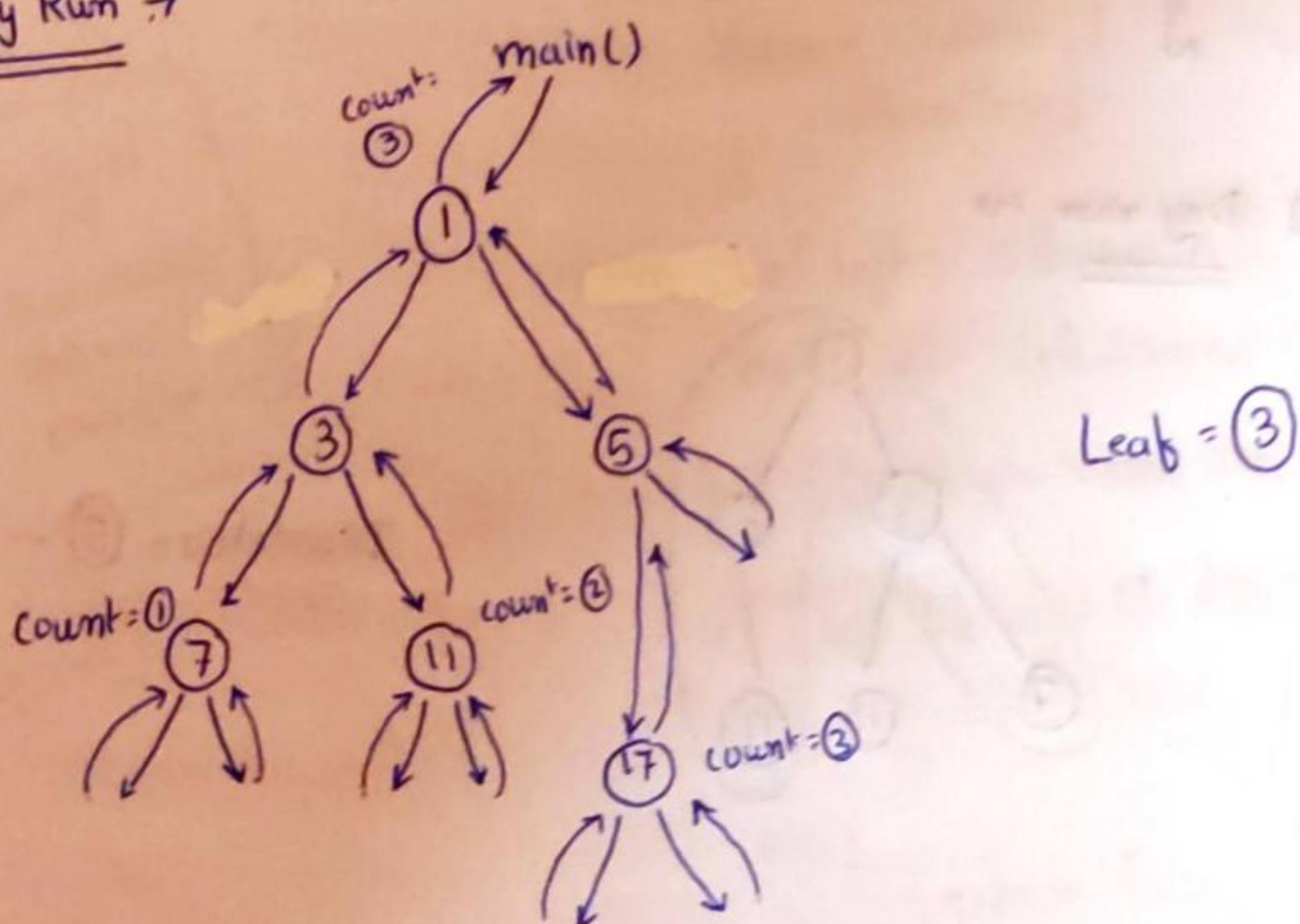
```
    if (root->right == NULL && root->left == NULL) {
```

```
        count++;
```

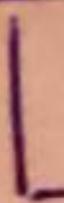
```
}
```

```
}
```

□ Dry Run ↗



- Diameter of Binary Tree

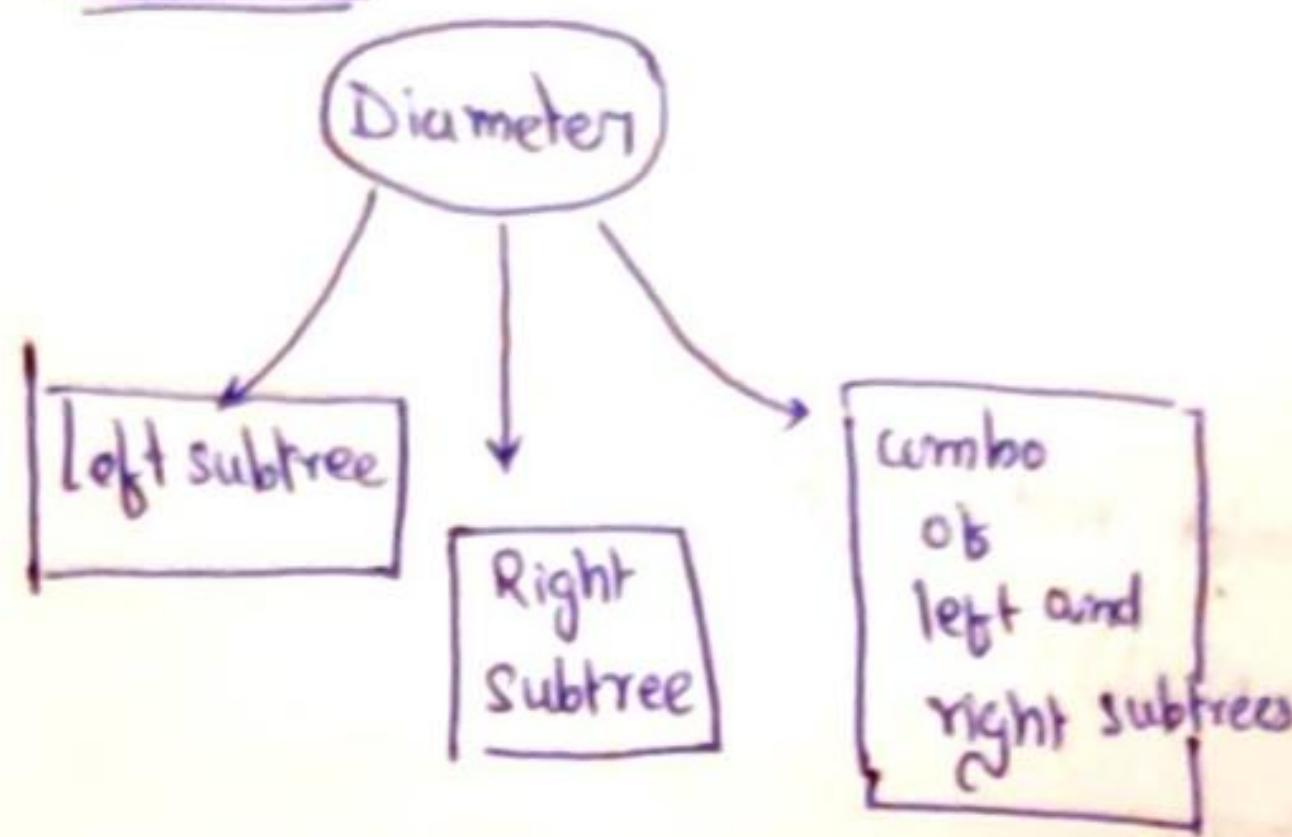


□ It basically longest path between any 2 nodes. (No. of nodes) .



Maximum no. of nodes between two end nodes.

Approach 1.



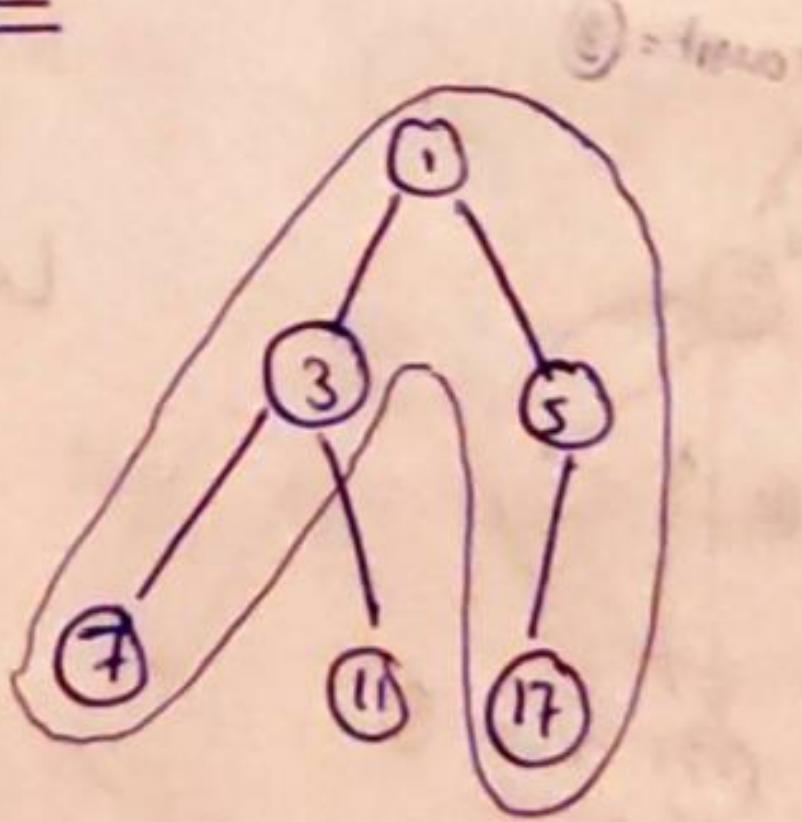
Code :

```
int diameter (node* root){  
    if (root == NULL){  
        return 0;  
    }  
    left ← int op1 = diameter (root → left);  
    right ← int op2 = diameter (root → right);  
    Both. ← int op3 = height (root → left) + height (root → right) + 1;  
    return max (op1, max (op2, op3));  
}
```

Note :

- The time complexity over here is $O(n^2)$ as we are using different function to calculate height.
- We can optimise it by calculating height along with diameter in same function by use of pair $\langle \text{int}, \text{int} \rangle$.
- Space complexity $> O(h)$

Day run :



Diameter = 5.

① Balanced Tree check →

→ Height of a tree is balanced when the difference between left and right subtrees is not more than 1 for all nodes of tree.

→ Approach →

If this all conditions are true, then we return true else false.

- i) check left subtrees
- ii) check right subtrees.
- iii) check $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree}) \leq 1$;

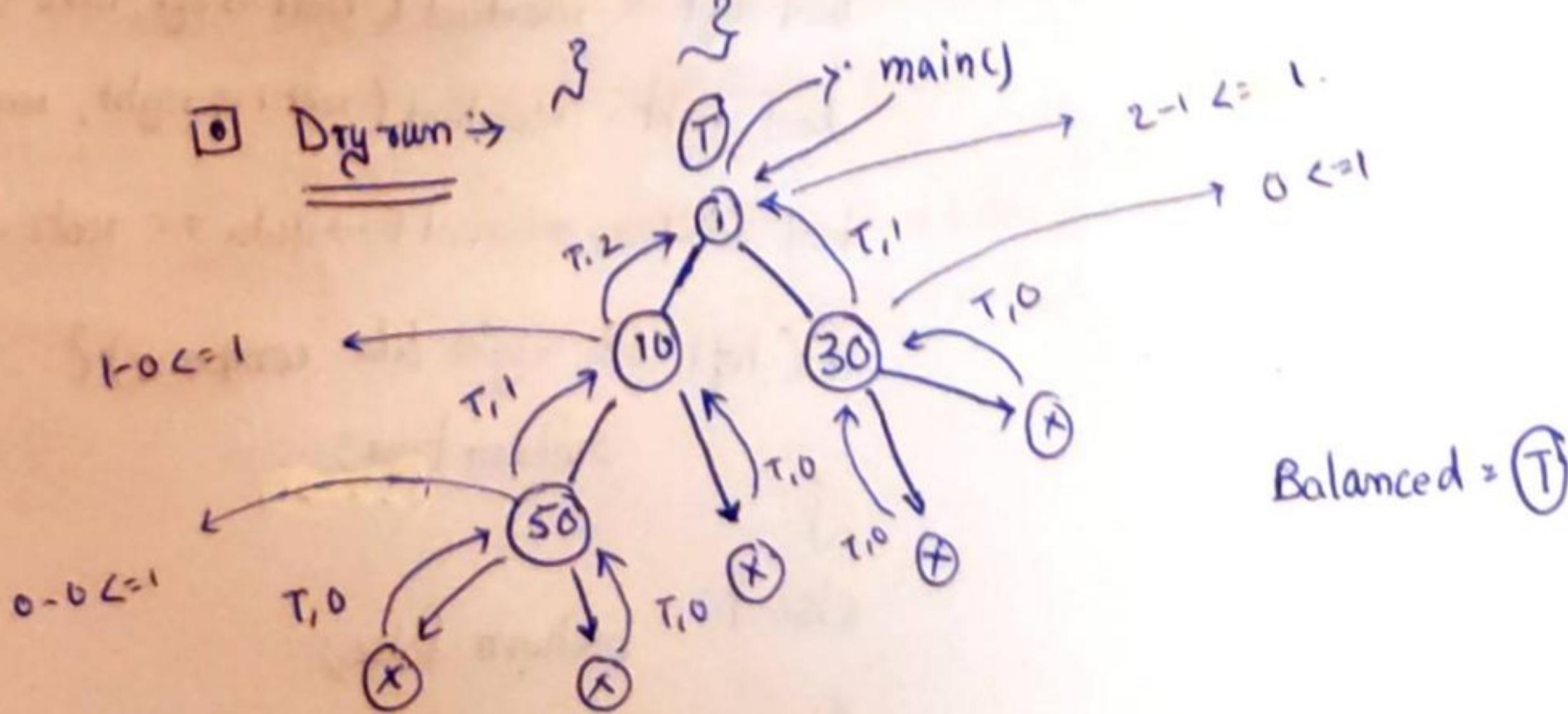
→ Code →

```
bool isBalanced (node * root) {
    if (root == NULL) {
        return true;
    }
    bool left = isBalanced (root -> left);
    bool right = isBalanced (root -> right);
    abs
    bool diff =  $(\text{height}(\text{root} \rightarrow \text{left}) - \text{height}(\text{root} \rightarrow \text{right})) \leq 1$ 
    if (left && right && diff) {
        return true;
    }
    else {
        return false;
    }
}
```

Time complexity → $O(n^2)$ → But can be optimised using pair<int,int>

Space complexity → $O(n)$.

② Dry run →

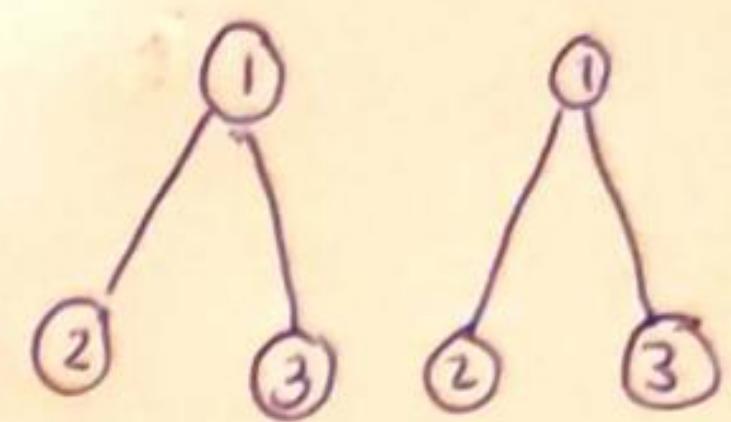


Balanced = T

① Check whether the trees are identical or not.

→ Two tree are identical when at same position the data inside the node is same.

EX



→ Approach →

△ We will replicate same traversing movements in both trees and compare the data stored at their nodes, if it is same we return true, else return false.

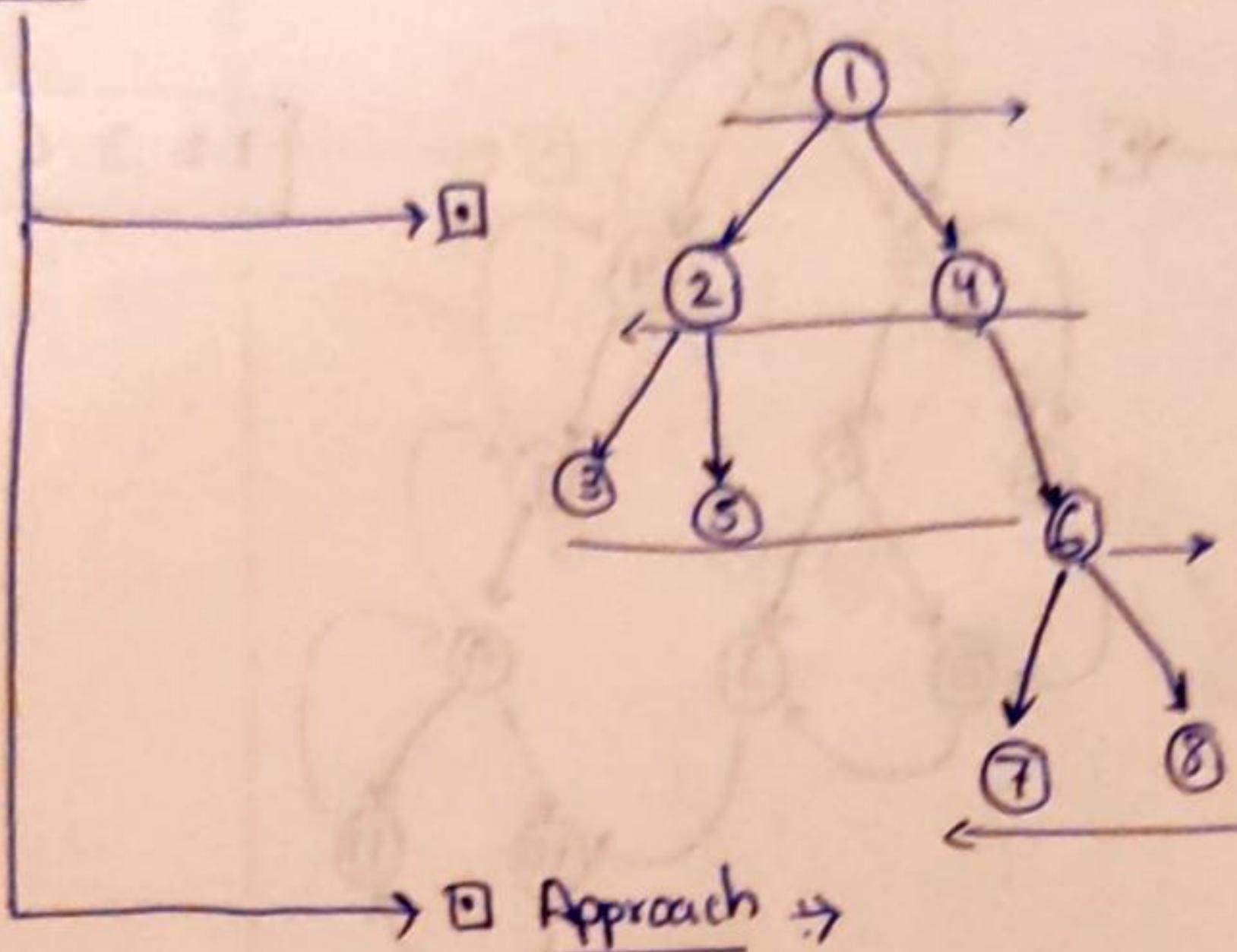
△ Left and right movements are handled with recursion.

→ Code →

Note →
• Time complexity - $O(N)$
• Space complexity - $O(h)$

```
bool identical(node* root1, node* root2){  
    if(root1 == NULL && root2 == NULL){  
        return true;  
    }  
    if(root1 != NULL && root2 != NULL){  
        return false;  
    }  
    if(root1 == NULL && root2 != NULL){  
        return false;  
    }  
  
    bool left = identical(root1->left, root2->left);  
    bool right = identical(root1->right, root2->right);  
    bool compare = root1->data == root2->data;  
  
    if(left && right && compare){  
        return true;  
    }  
    else{  
        return false;  
    }  
}
```

④ ZigZag Traversals \Rightarrow (Spiral)



1 4 2 3 5 6 8 7

8
7
6
5
4
3
2
1

Queue

flag = 0
flag = 1
flag = 0
flag = 1

1
4 2
3 5 6
8 7

{ ans vector
at each level.

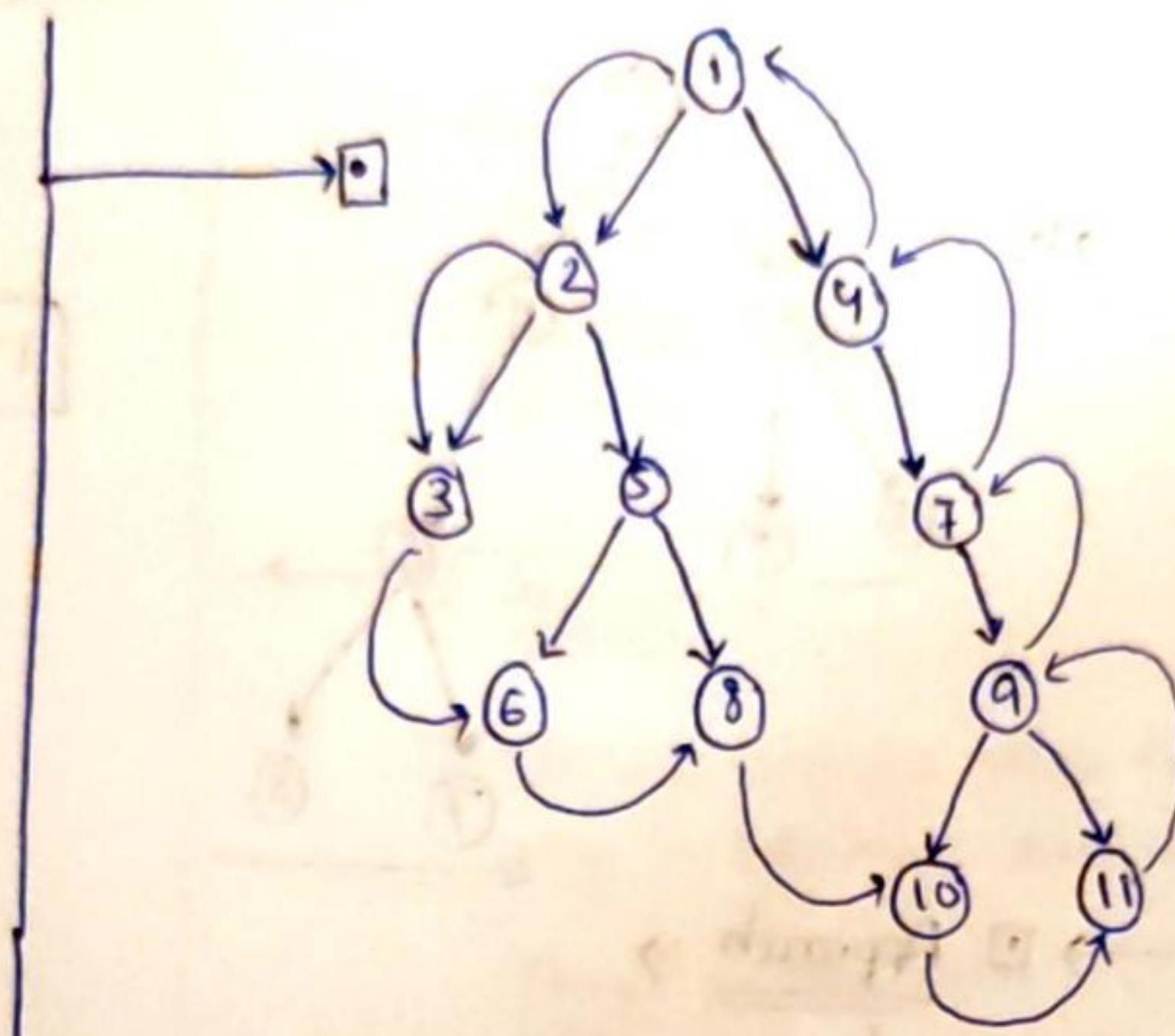
① We will use level order traversal here in which we will use a queue. Now, we will simply do a level order traversal using a flag variable whose value updates after each level either 0 or 1. If value is 1 we do level order traversal from left else from right.

② Lastly we will store this ^{node according to} sequence in a vector at each level and finally display it.

③ Time Complexity $\Rightarrow O(N)$

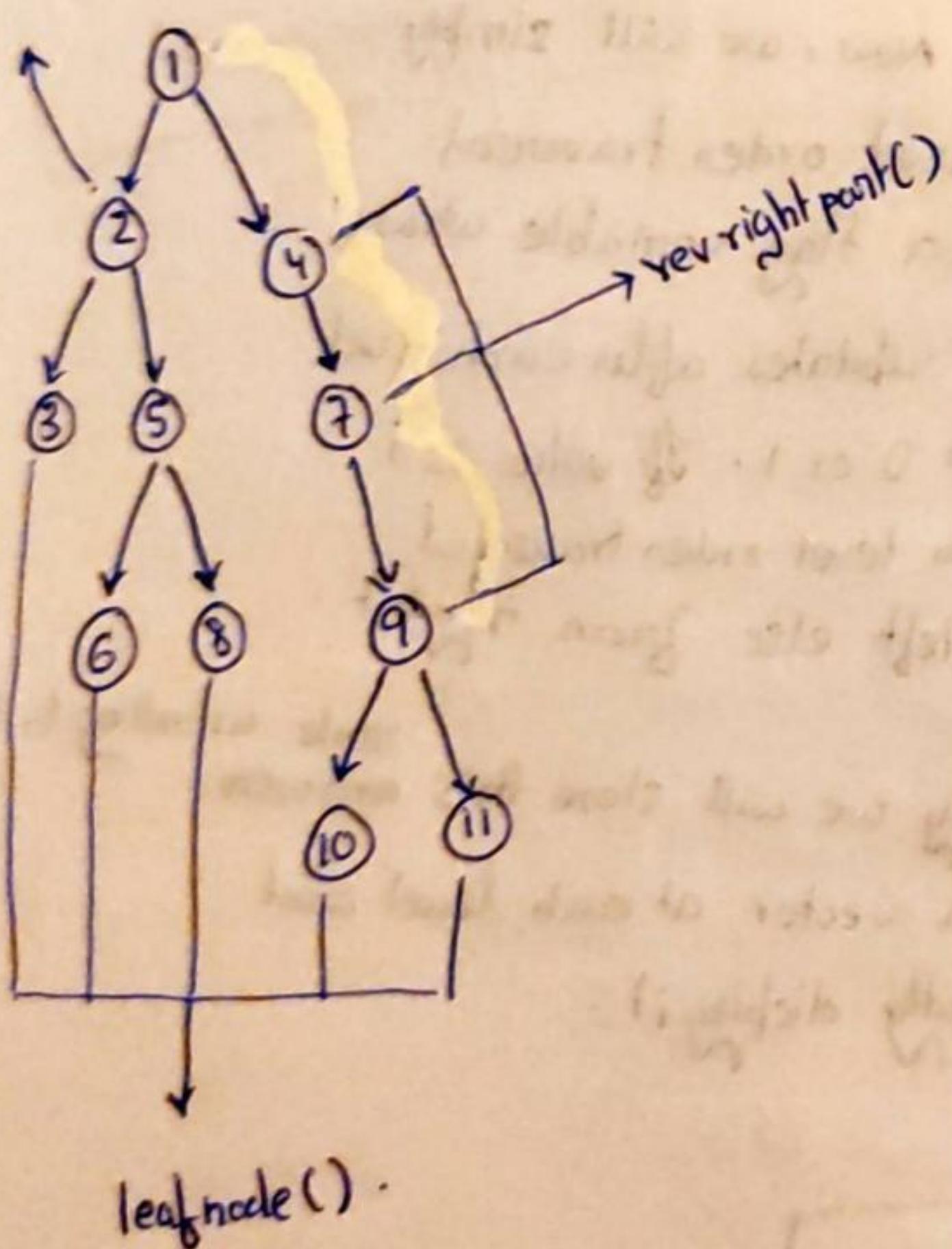
④ Space Complexity $\Rightarrow O(N)$

① Boundary Traversal ↗



1 2 3 6 8 10 11 9 7 4

leaf part()



leaf node()

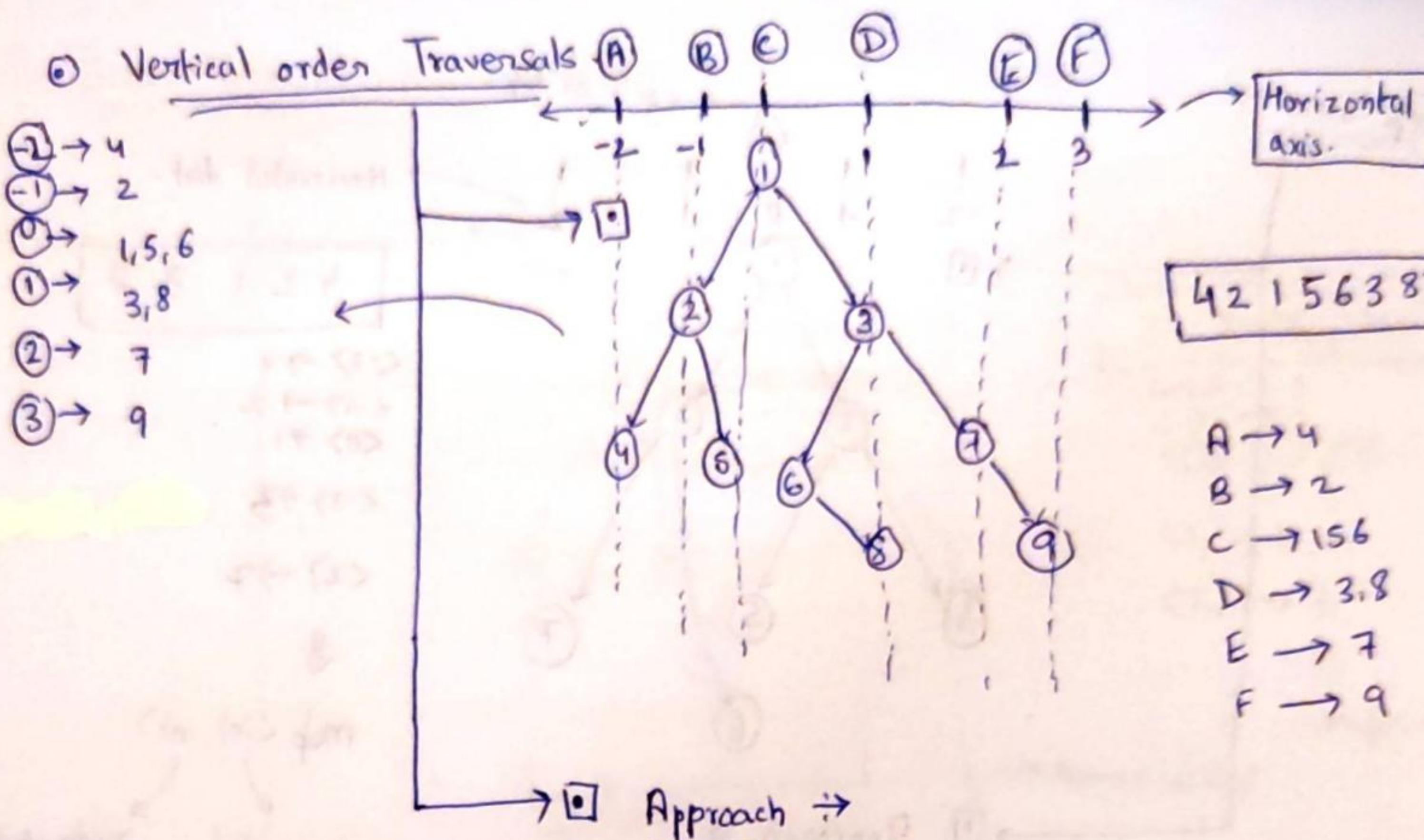
Approach

Divide the question in 3 function parts. One left part printing left node from root to the nodes before leaf. Keep in mind in left part() if left node exist then only move in left else you can move in right.

leaf node() print all the leaf nodes. Should be called for both left and right part.

reverse right part() prints all the nodes present in right parts if it exists or else it can also move to left part.

Time Complexity $\rightarrow O(N)$
Space Complexity $\rightarrow O(n)$



i) We will have use the level order traversal on a queue node, storing (Horizontal, level, node)

and then we gonna do simple level order traversal.

ii) Remember the data structure which will store data here would be:

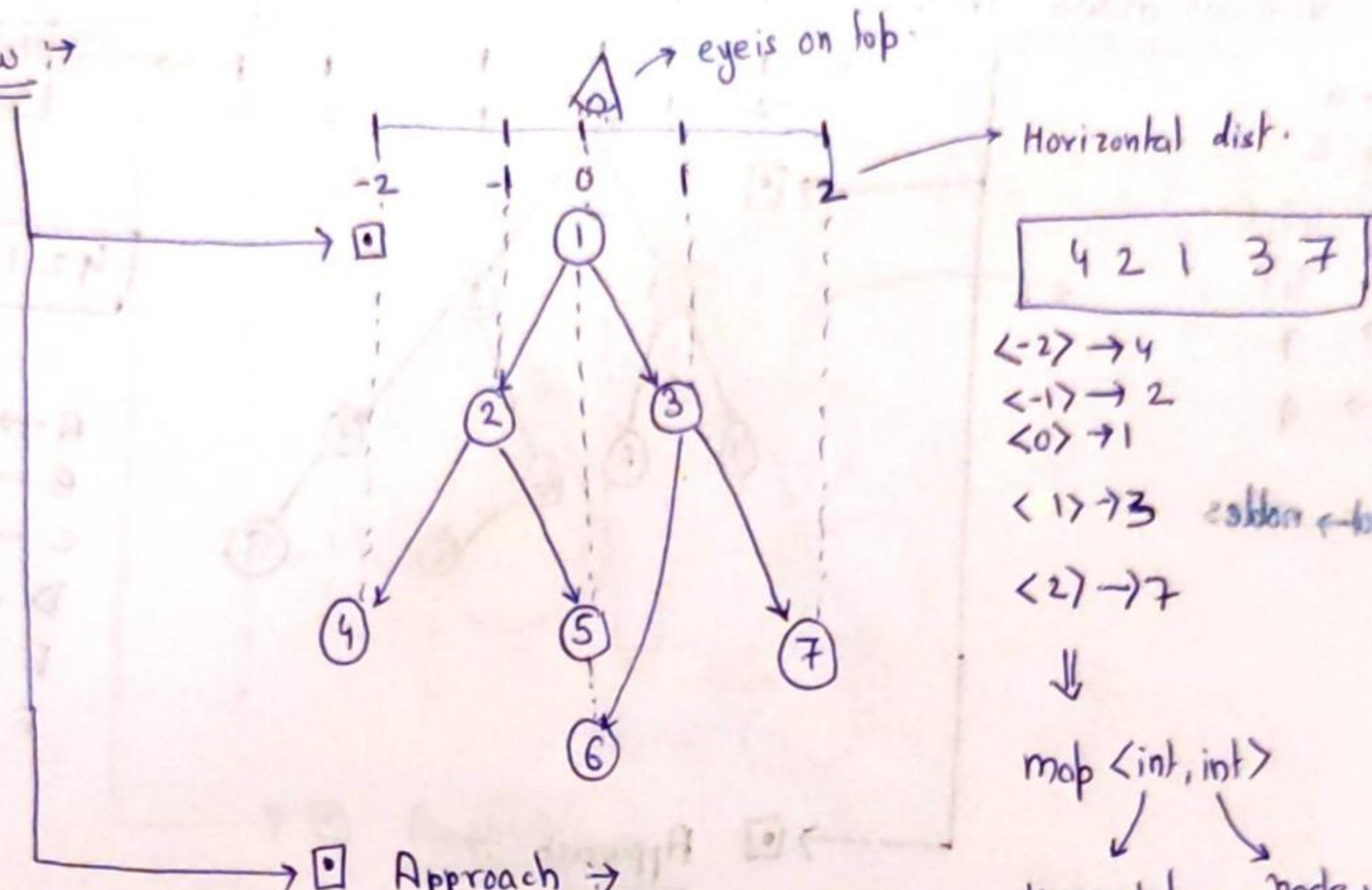
`map < int, pair < int, vector<int>> .`

`(map → {Horizontal, level, vector of nodes}) .`

Time Complexity $\rightarrow O(N)$

Space Complexity $\rightarrow O(N)$.

① Top View →



→ Approach →

i) Levelorder Traversal

map <int, int>

↓
Horizontal → node → data.

ii) Here mapping would be
one to one. i.e.,

if $m[0] = 1$

$m[0]$ cannot be equal to 5 or 6.

△ Important condition:

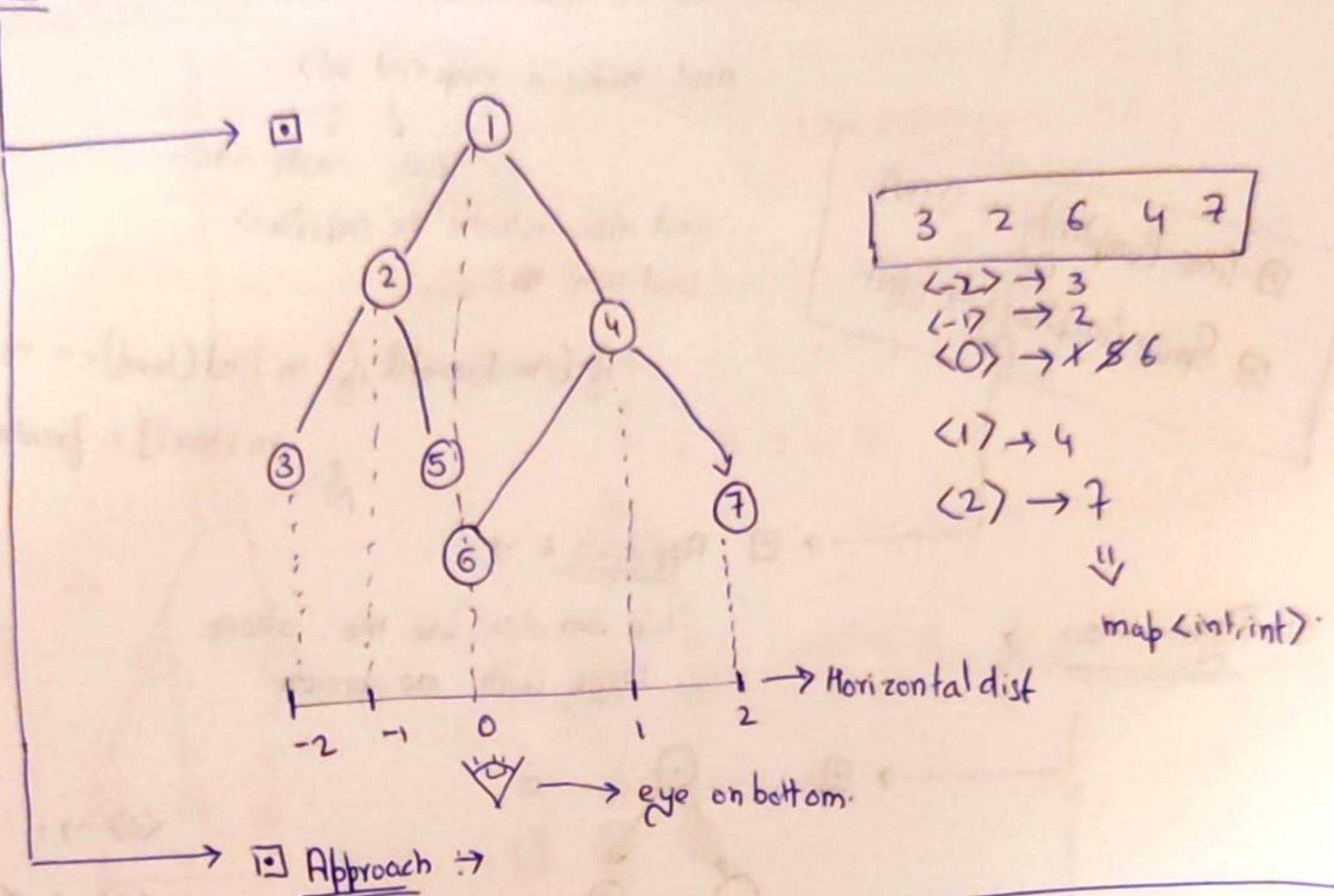
if($m.\text{find}(\text{horizontal}) == m.\text{end}()$) {

$m[\text{horizontal}] = \text{frontnode} \rightarrow \text{data};$

① Time complexity $\rightarrow O(N)$

② Space complexity $\rightarrow O(N)$

① Bottom View \Rightarrow

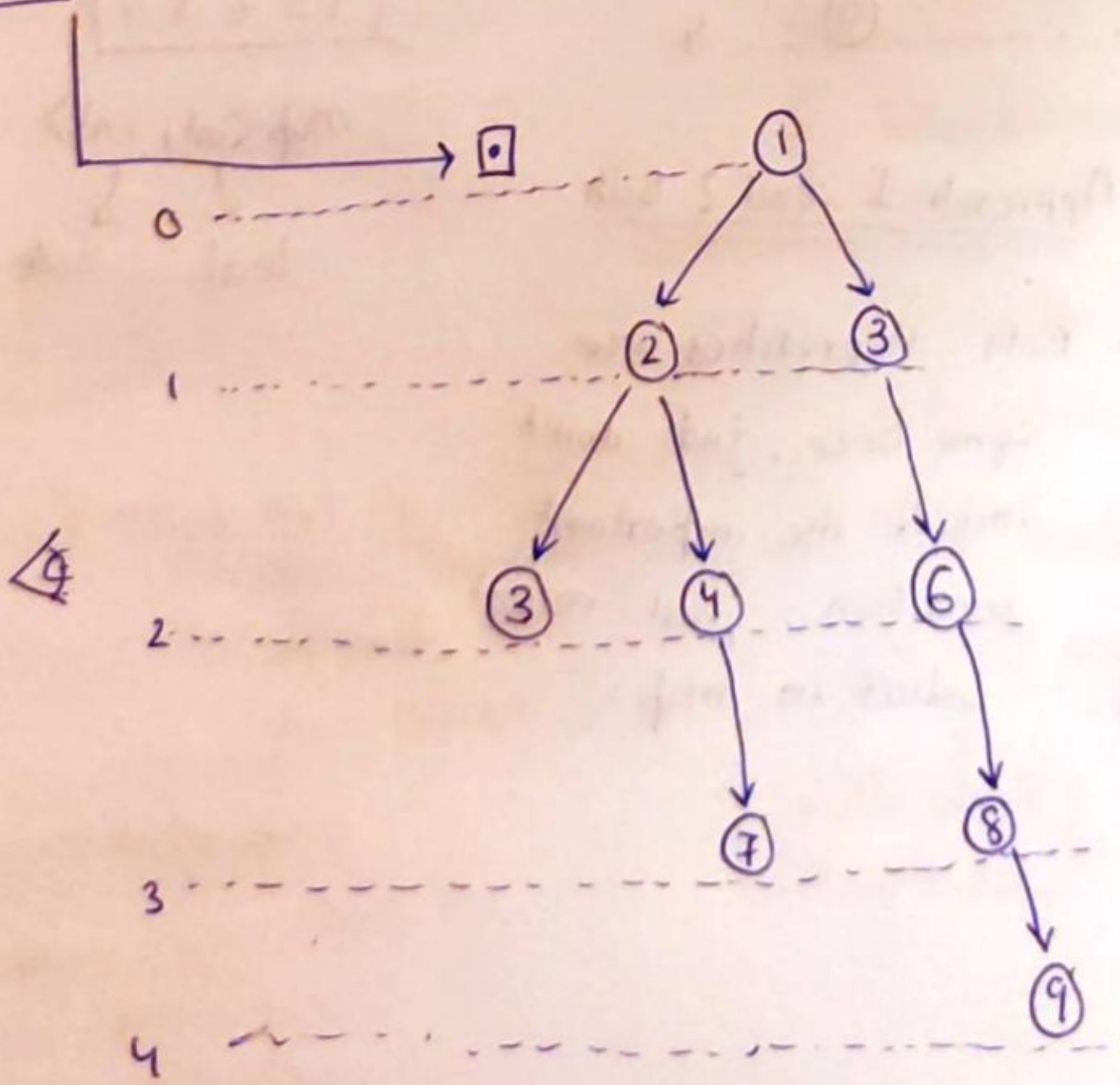


② Approach \Rightarrow

- i All would be same only exclude important condition and do the insertion inside the map.

① Time complexity $\rightarrow O(N)$
② Space complexity $\rightarrow O(N)$

① Left view \Rightarrow



$\langle 0 \rangle \rightarrow 1$
 $\langle 1 \rangle \rightarrow 2$
 $\langle 2 \rangle \rightarrow 3$
 $\langle 3 \rangle \rightarrow 7$
 $\langle 4 \rangle \rightarrow 9$

1 2 3 7 9

map<int, int>:
level
node \rightarrow data

Approach 1 :

Just do level order traversal

and make a map $\langle \text{int}, \text{int} \rangle$

level node \rightarrow data.

and also include the important condition ~~if~~:

~~if (m[Level]) if (m.find(level) == m.end())?~~

~~m[level] = frontnode->data;~~

~~q.~~

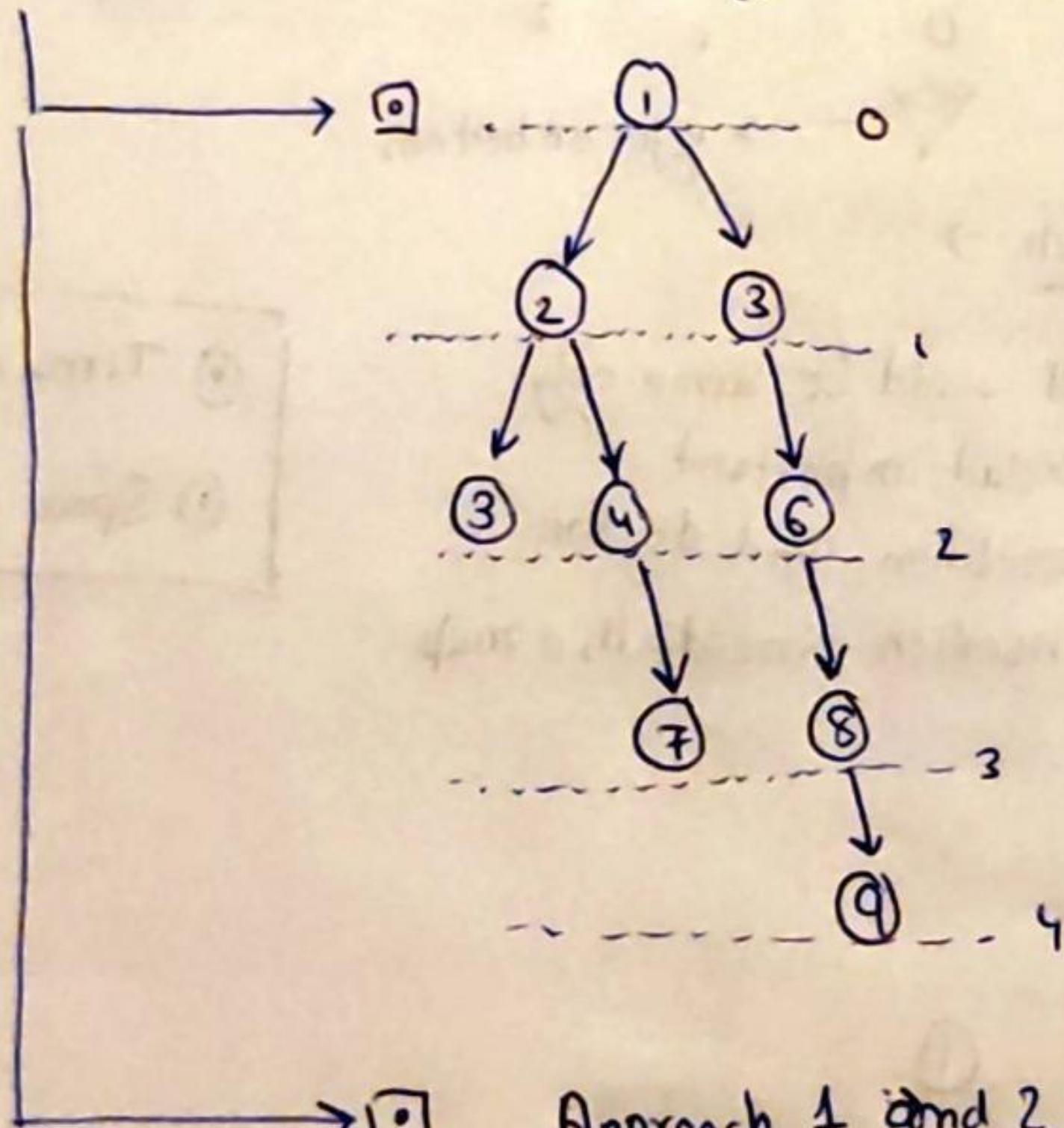
Time Complexity $\rightarrow O(N)$

Space Complexity $\rightarrow O(N)$

Approach 2 :

Right view \rightarrow

We can also use the above logic with recursion



$\langle 1 \rangle \rightarrow 1$

$\langle 2 \rangle \rightarrow 2 \ 3$

$\langle 3 \rangle \rightarrow 6 \ 8$

$\langle 4 \rangle \rightarrow 9$

1 3 6 8 9

map $\langle \text{int}, \text{int} \rangle$

level node \rightarrow data.

Approach 1 and 2 Both

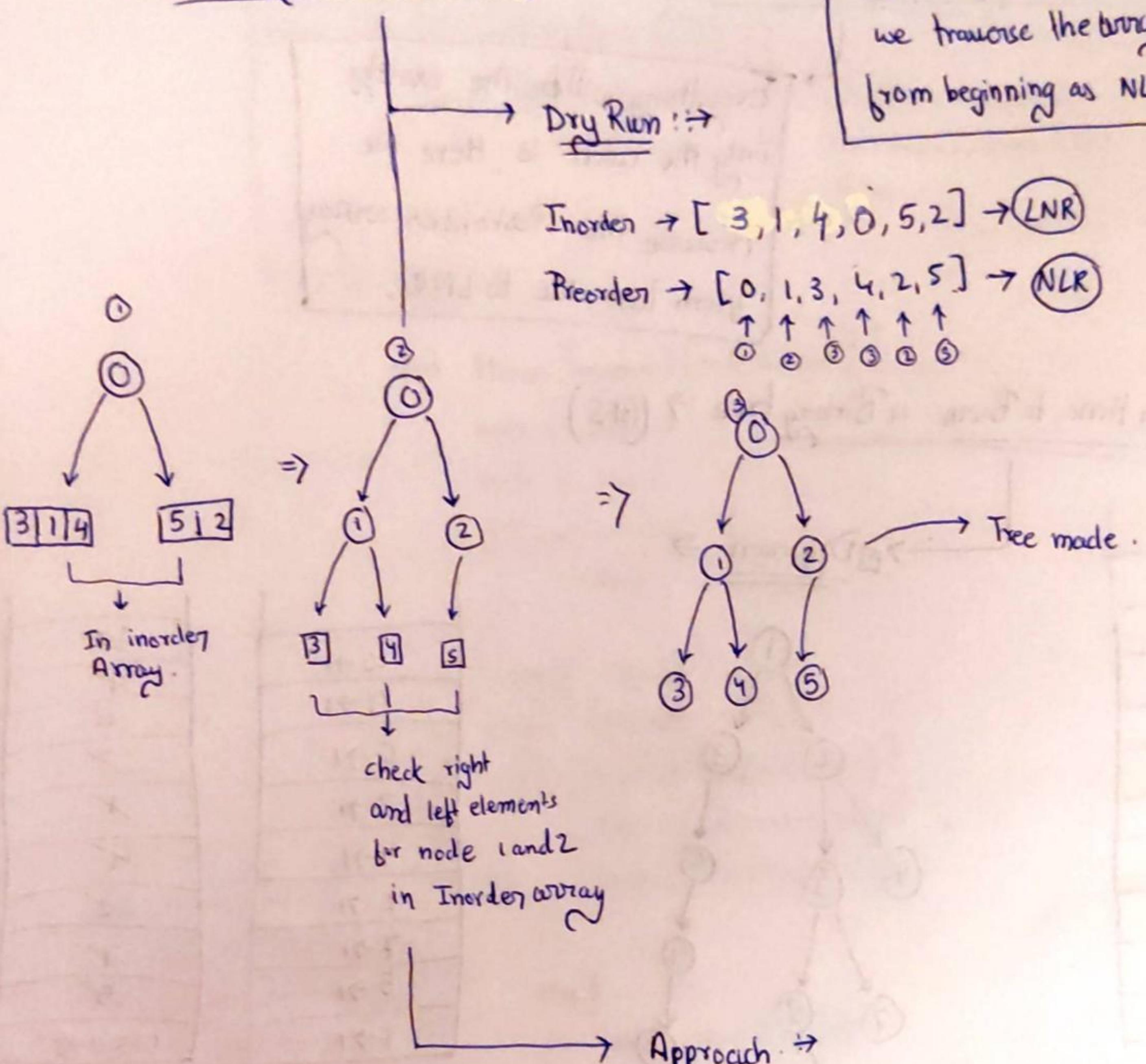
- Both Approaches are same here, just don't include the important condition, just insert values in map.

Time Complexity $\rightarrow O(N)$

Space Complexity $\rightarrow O(N)$

① Construct a Binary Tree from Inorder and Preorder.

Remember in Preorder
we traverse the binary
from beginning as NLR



Time complexity :→

$O(n \log n)$ → (if map is used for storing inorder elements with their indexes.)

Space complexity :→

$O(N)$

i) Take elements of Preorder Array as root elements and find the root elements position (pos) in inorder array.

ii) Then make recursive calls for left and right nodes of tree.

inorder start
↳ pos-1

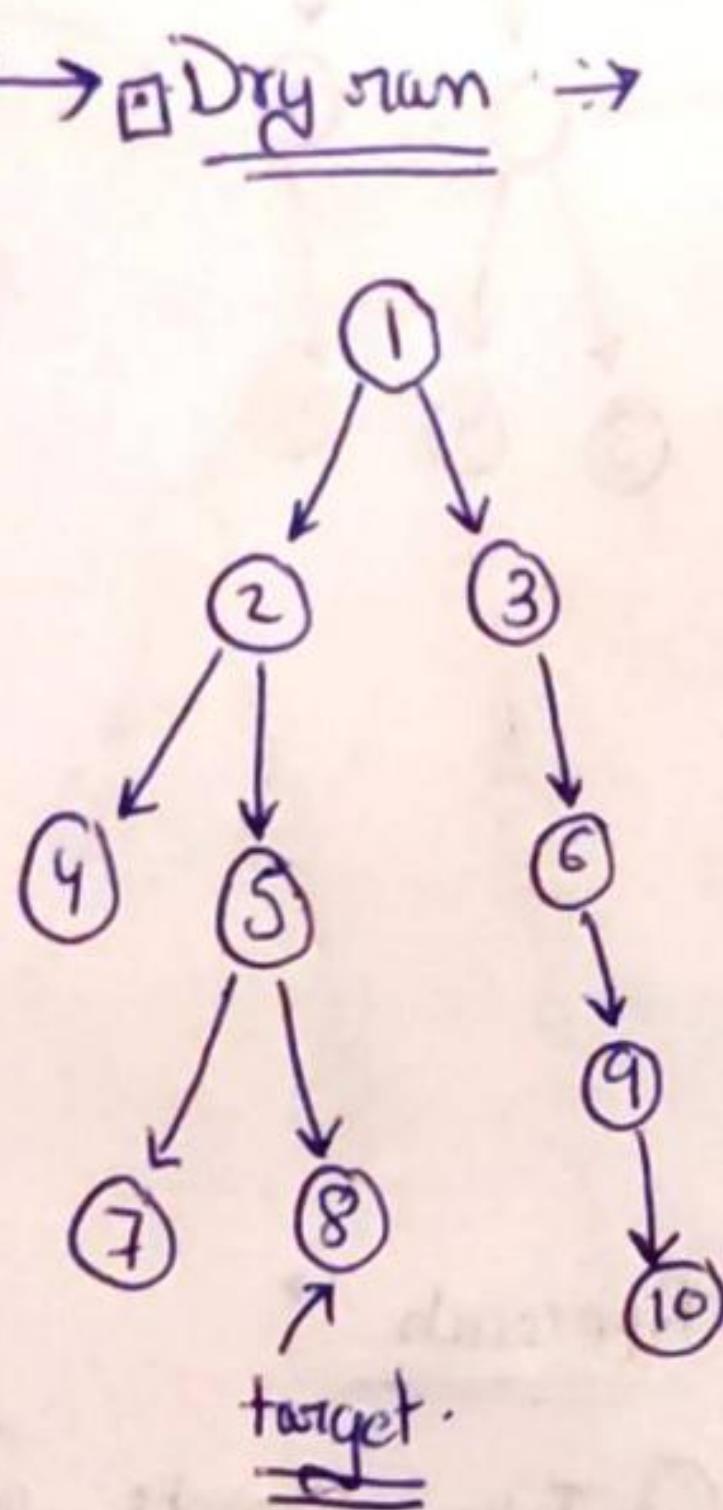
post1 → inorder end.

① Construct a Binary Tree from inorder and Postorder.

Everything will be the exactly
only the catch is, Here we
traverse the Postorder array
from last due to LNR.

② Minimum time to Bwin a Binary tree \Rightarrow (BFS)

10 \rightarrow 9
9 \rightarrow 6
8 \rightarrow 5
7 \rightarrow 5
6 \rightarrow 3
5 \rightarrow 2
4 \rightarrow 2
3 \rightarrow 1
2 \rightarrow 1
1 \rightarrow NULL



map <Node, parent>

STEP 1

$\therefore \text{Ans time} = 7$

(iii) Also remember to increment
time if right, left childs and
parent any of the one are
found unvisited.

Time Complexity $\Rightarrow O(N)$

Space Complexity $\Rightarrow O(N)$

if insert
operation of
of Node, parent
map is $O(1)$
else $O(N \log N)$

Dry run \Rightarrow

10 \rightarrow 1
9 \rightarrow 1
8 \rightarrow 1
7 \rightarrow 1
6 \rightarrow 1
5 \rightarrow 1
4 \rightarrow 1
2 \rightarrow 1
7 \rightarrow 1
5 \rightarrow 1
8 \rightarrow 1

visit map

(Node*, bool*)

STEP 3

10
9
8
7
6
5
4
2
7
5
8

queue

STEP 2

Time = 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7

Approach \Rightarrow

i) Do node to parent mapping

map <Node*, Node*> by
level order traversal.

ii) Find target node and push
it queue, then check for its child
right, left and parent; if they
are not visited in visited map
then push them in queue and mark them
visited and then pop the current node.

① Morris Traversal ↗

Time Complexity = $O(n)$
Space Complexity = $O(1)$

All other traversal techniques use $O(n)$ space.

Code ↗

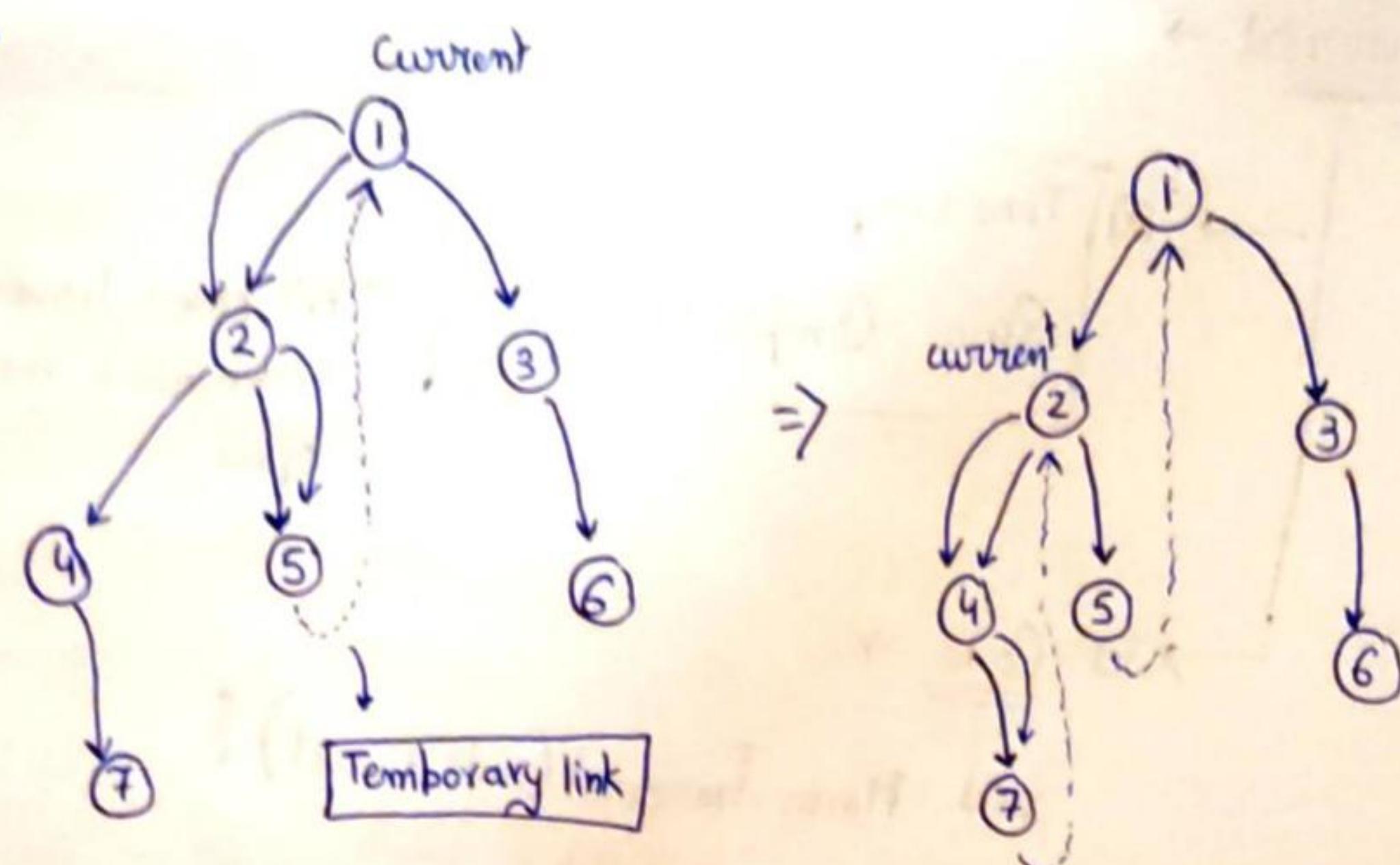
```
void MorrisTraversal(node *root) {
    node *current;
    node *pre;
    if (root == NULL) {
        return;
    }
    current = root;
    while (current != NULL) {
        if (current->left == NULL) {
            cout << current->data << " ";
            current = current->right;
        } else {
            pre = current->left;
            while (pre->right != NULL && pre->right != current) {
                pre = pre->right;
            }
            if (pre->right == NULL) {
                pre->right = current;
                current = current->left;
            } else {
                pre->right = NULL;
                cout << current->data << " ";
                current = current->right;
            }
        }
    }
}
```

check the left node of our current if it is NULL then print data of current node, and move current node two to its right part.

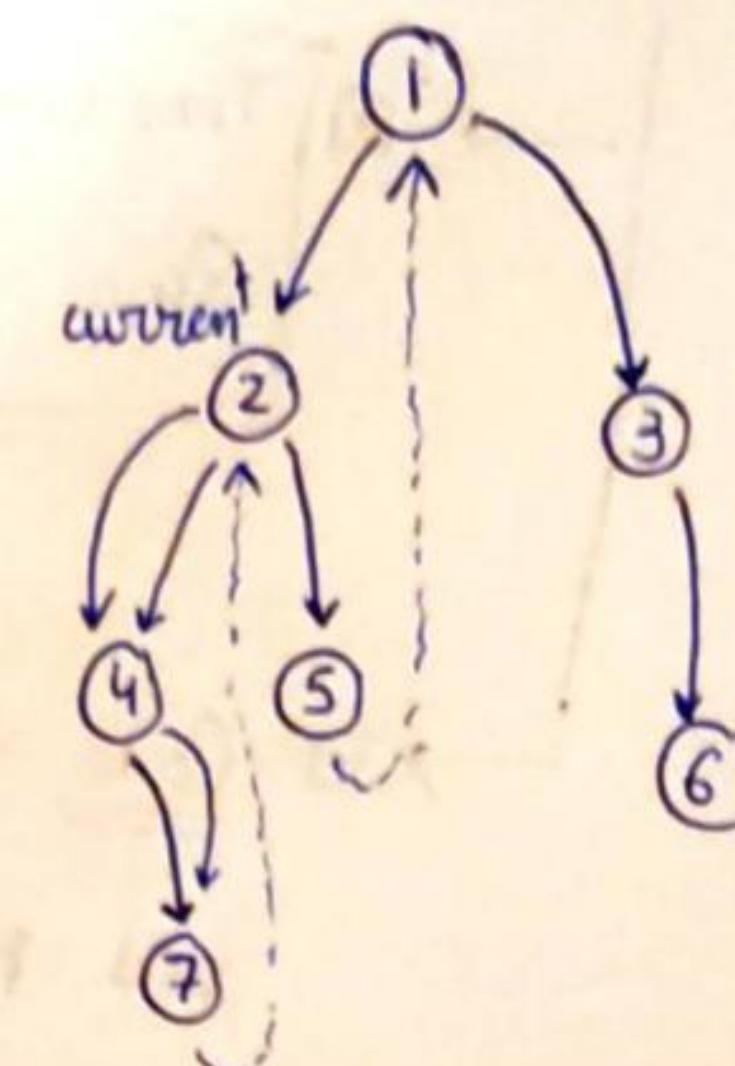
Now if current node is not NULL, then assign left of current node to predecessor node, after that move the predecessor node towards right until it finds a NULL predecessor \rightarrow $right = current$.

Now, if $pre \rightarrow right == NULL$ we do temporary linking of $pre \rightarrow right$ to current node, else we remove that temporary link.

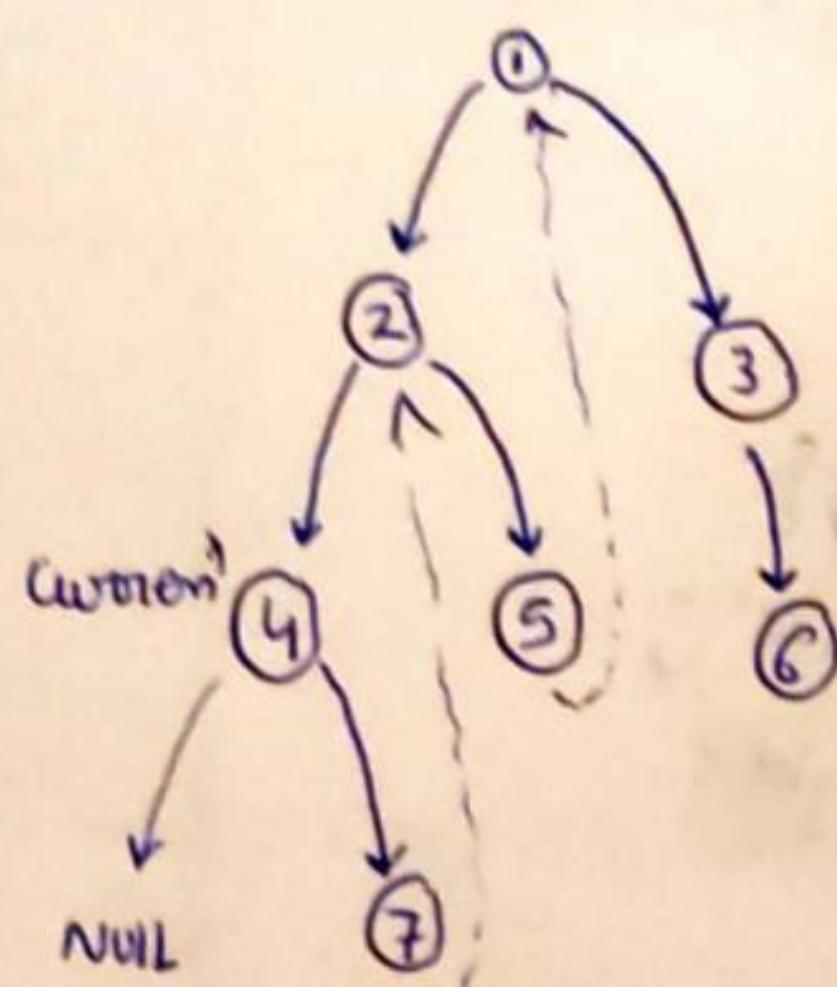
② Dry Run →



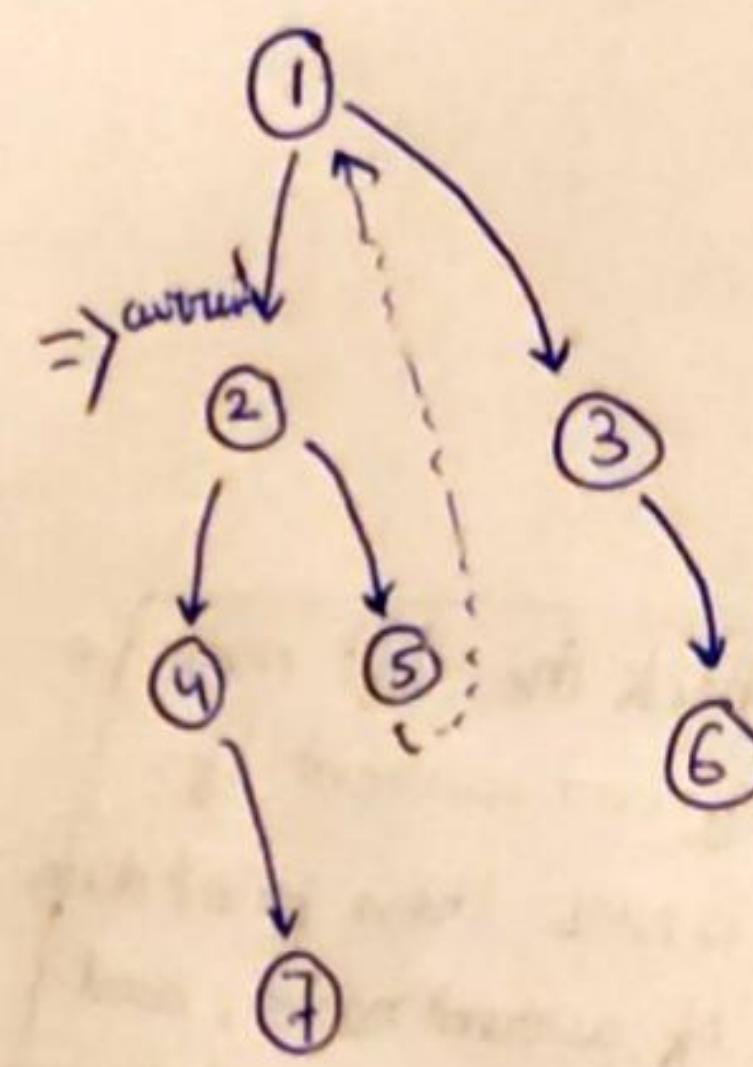
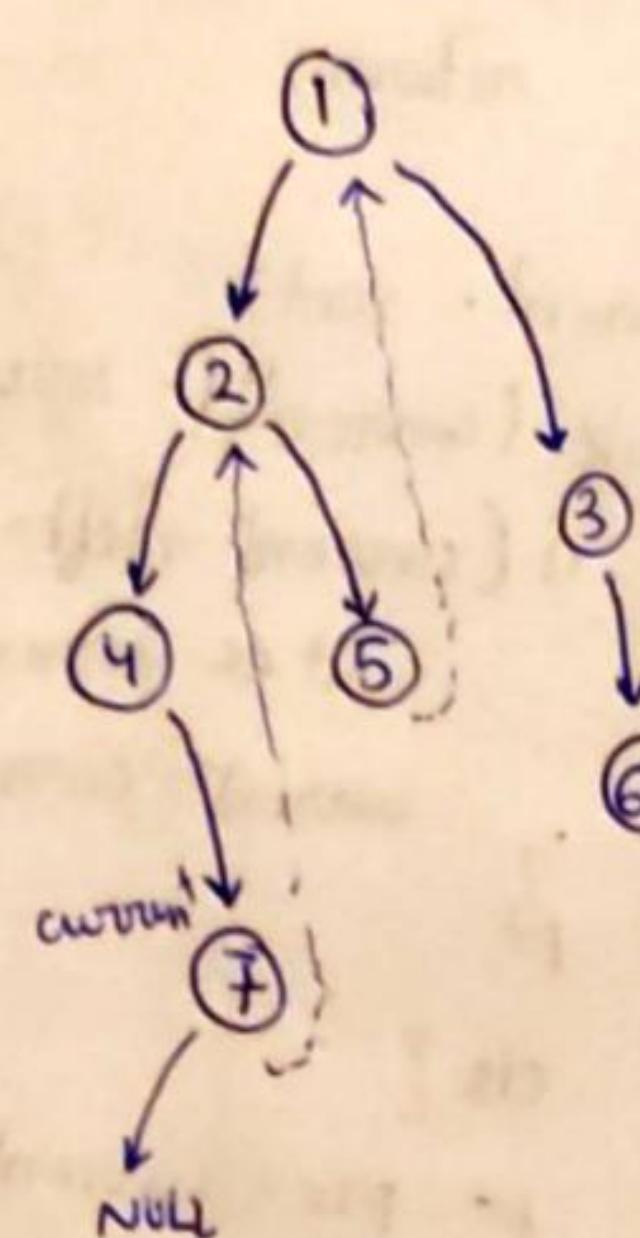
⇒



↓



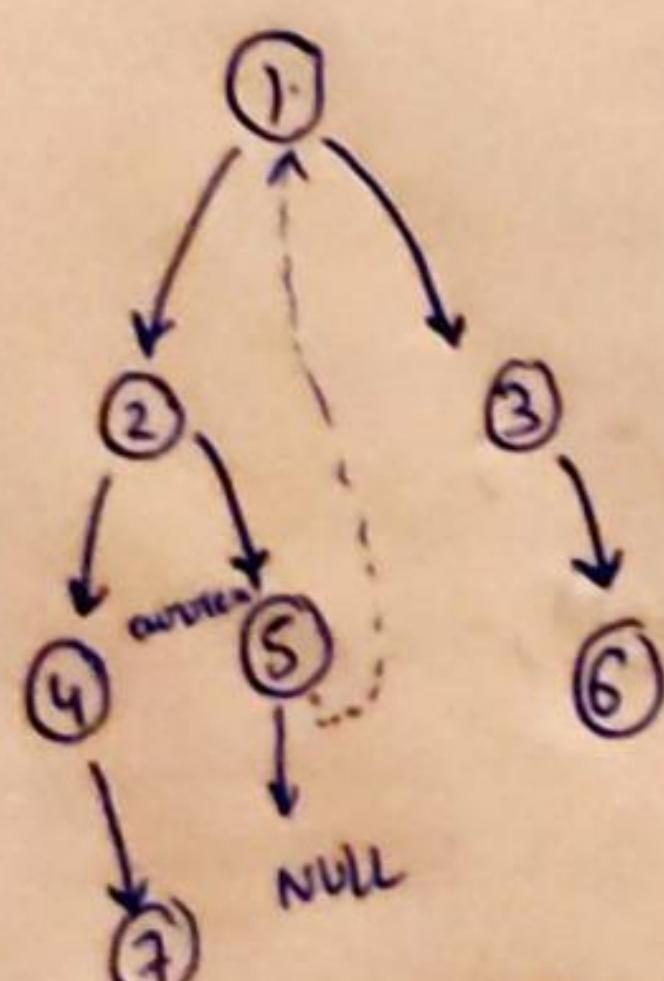
⇒



Print 4

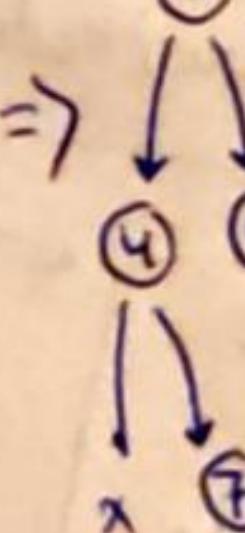
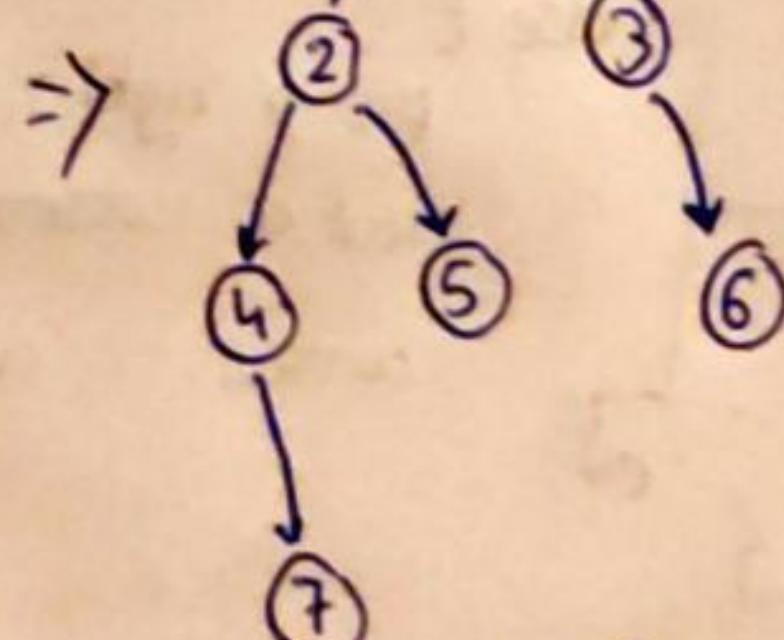
Print 7 and break temporary link of 7 and 2.

Print 2

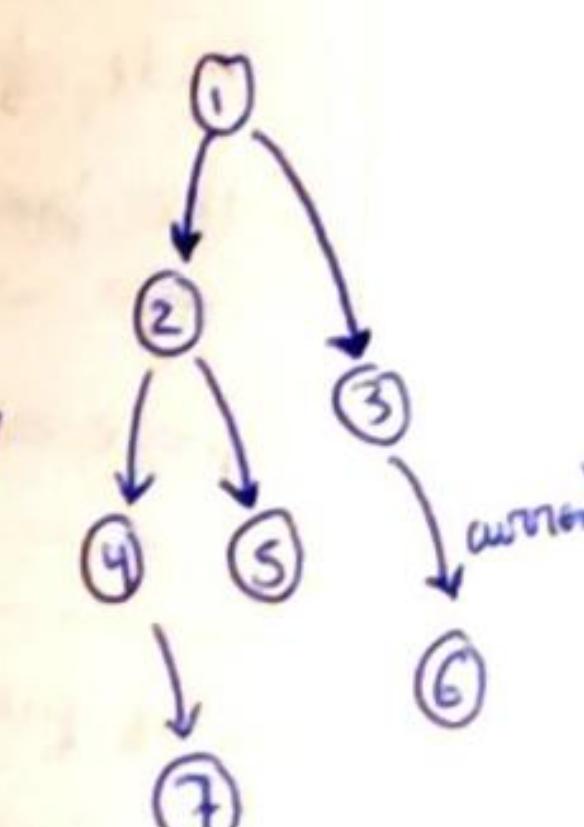


Print 5 and break link of 5 and 1

⇒



⇒



output: 4, 7, 2, 5, 1, 3, 6

① Binary Trees →

① What is a Tree? What is a Binary Tree. Explain terms listed below

- i) Node
- v) Siblings
- ii) Root
- vi) Ancestors
- iii) Parent
- vii) Descendants
- iv) Children
- viii) Leaf

② Structure of node for Binary Trees and N-way trees. A node can have child as node and NULL.

③ Write code to create a Binary tree using Recursion and Level order Traversal.

④ Discuss Approach, Dry Run, Time and Space Complexity of following Traversals

- i) Level Order Traversal
- ii) Inorder Order Traversal
- iii) Preorder Order Traversal
- iv) Postorder Order Traversal
- v) Morris Traversal, How morris Traversal is different from other Traversals

⑤ What is Height? Write code to find it? What is its time and Space complexity

Give Dry Run and also discuss what is a skew tree? Count no. of leaf node in a binary tree, give dry run and code.

⑥ What is Diameter of Binary tree? Discuss Approach, time and space complexity, Dry Run, code. How to optimise its time complexity from $O(n^2) \rightarrow O(n)$.

⑦ What is Balanced Tree? Discuss Approach, time, Space complexity, dryRun and Code.

⑧ When two trees are said to be identical? Discuss approach, time and Space, dry run and Code.

- ⑨ Important Interview questions , discuss time and Space, dry run, approach, Code
- ① Zigzag Traversal.
 - ② Boundary Traversal.
 - ③ Vertical order Traversal.
 - ④ Top view and Bottom view.
 - ⑤ Left view and Right view.
 - ⑥ Construct a Binary Tree from Inorder and Preorder Array.
 - ⑦ Construct a Binary Tree from Inorder and Postorder Array.
 - ⑧ Calculate minimum time required to burn a Binary Tree.
using BFS.
 - ⑨ Sum of Longest Bloodline.
 - ⑩ LCA of Binary Tree.
 - ⑪ k sum paths.
 - ⑫ Kth Ancestor.
 - ⑬ Maximum Sum of Adjacent Nodes.