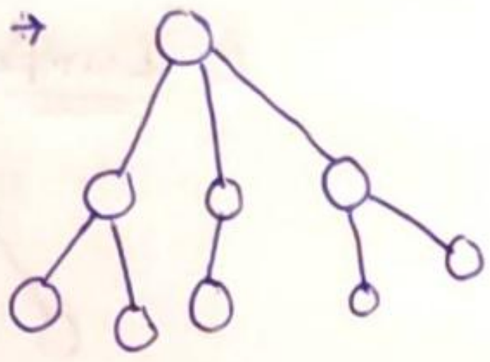
⊙ Binary Trees →

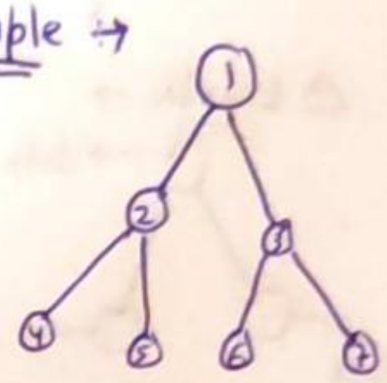
⊡ Tree → ⊙ A tree is a non-linear datastructure, where one node can get connected to multiple other nodes.

⊙ Example →



⊙ Note → ⊡ A Binary tree is a tree where child of any node can be (<=2), in number.

⊡ Example →



⊙ Important terms related to Peree →

(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).

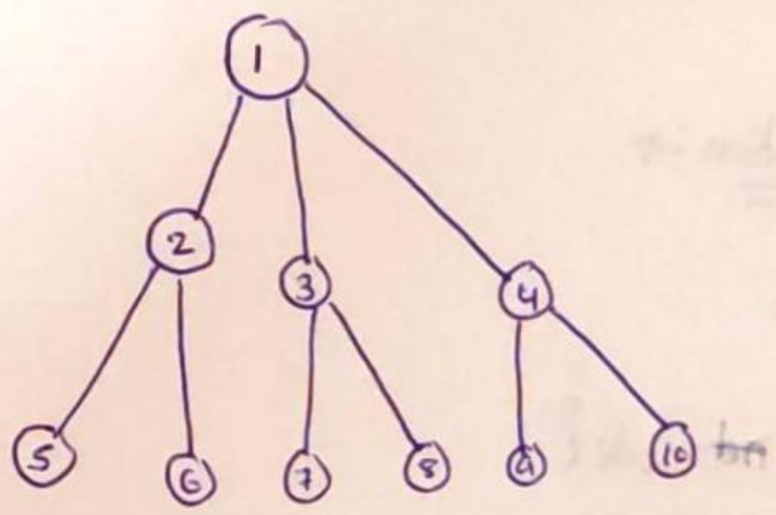


Fig 1.1.

(iii) Parent → The nodes having subnodes is termed as parent.
Example → (1) is parent of (2), (3) (4).

(iv) Children → As node (1) is parent of nodes (2), (3), (4). So, nodes (2), (3), (4) are children of node (1).

(v) Sibblings → Two or more nodes haveing same parent are termed as sibblings.
Example → (2), (3), (4) nodes are sibblings of each other.

(vi) Ancestors → An ancestor is any other node on path from that node to the root.
Example → (1), (3) is ancestor of (7)

(vii) Descendants → It is just opposite of ancestors.
Example → Descendant of (2), is (5) and (6).

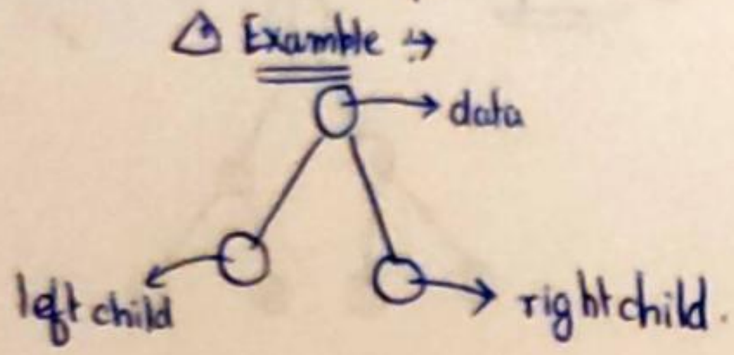
(viii) Leaf → Leaf node are those nodes which do not have children.
Example - 5

⊙ For Binary trees node structure :→

△ Snippet :→

```
node {
    int data;
    int * left ; → For left child
    int * right; → For right child.
}
```

△ Example :→



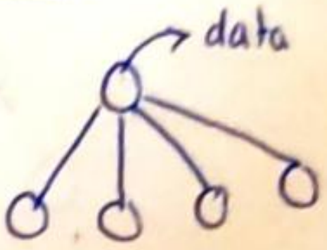
⊙ Notes :→ ⊡ Here child can be node or, even NULL.

⊙ For n-ary tree node Structure :→

△ Snippet →

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

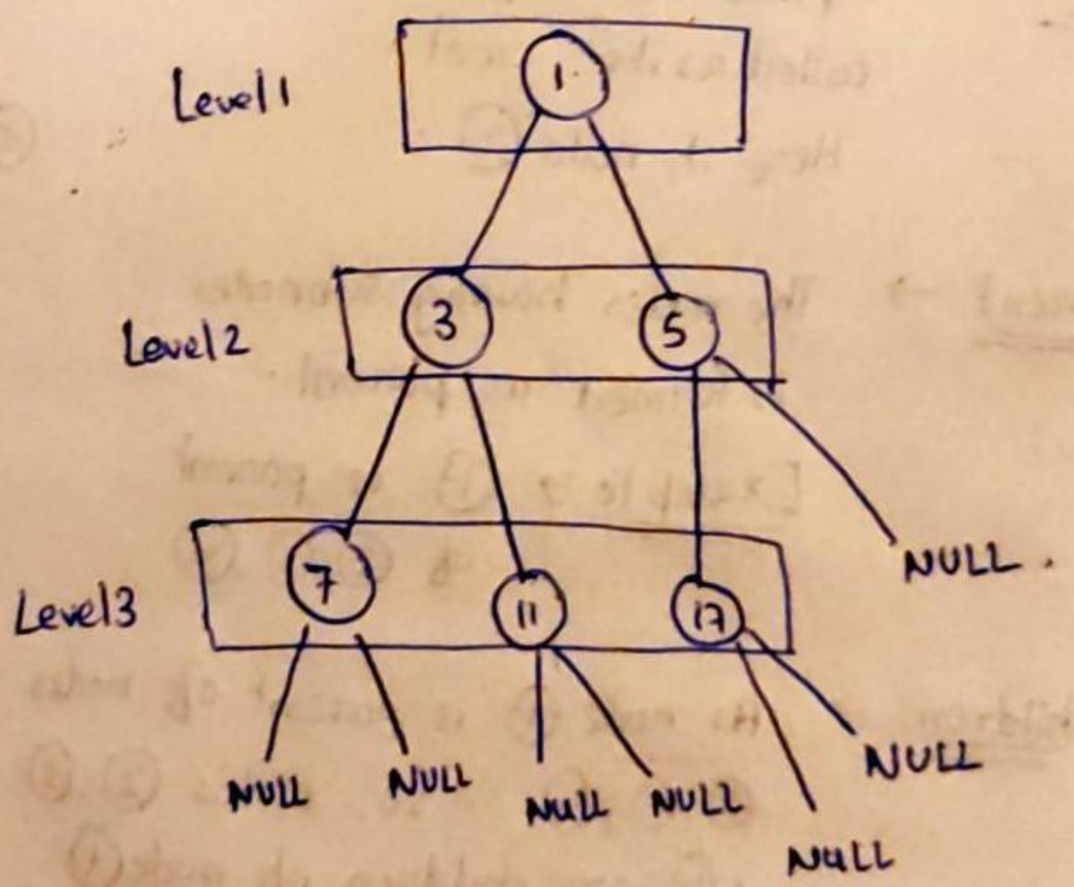
△ Example :→



⊙ 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;
    }
```



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

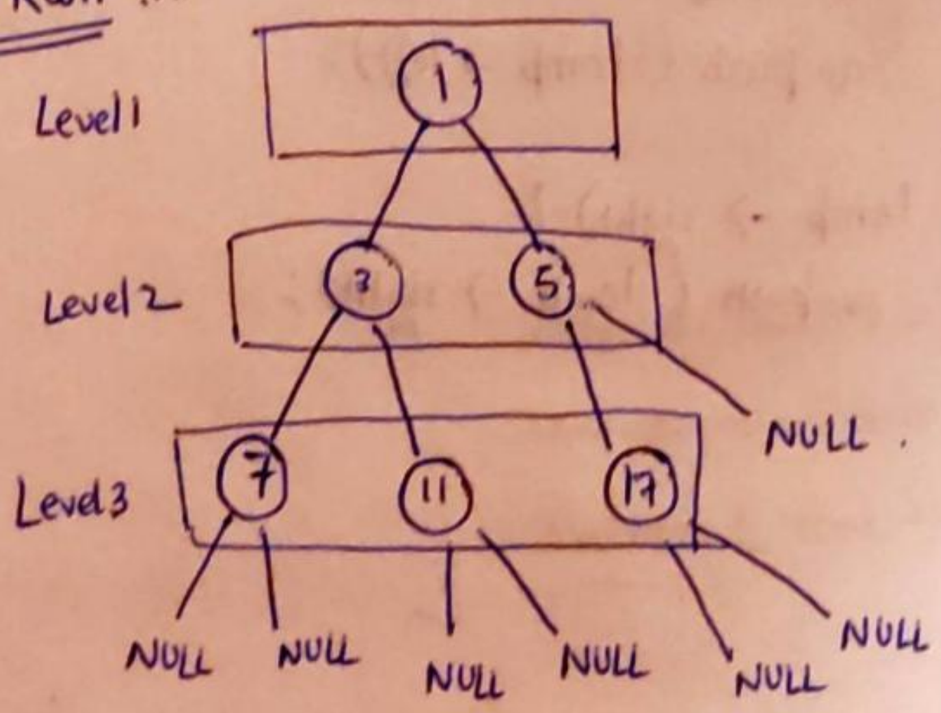
⇓

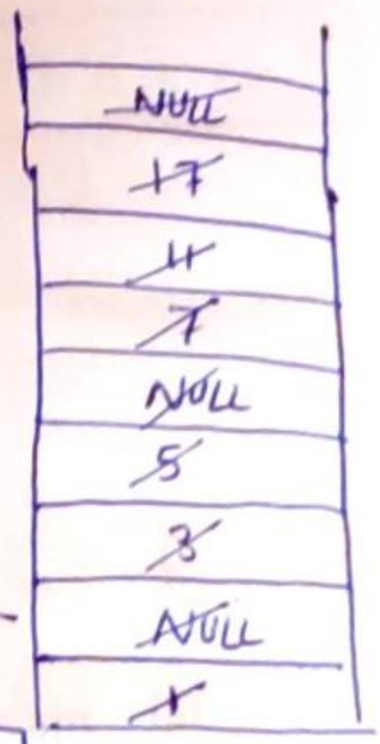
Here -1 denotes NULL, and according to input our recursion works.

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

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

        return root;
}

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

⊙ Level Order Traversal :→

⊡ Dry Run :→



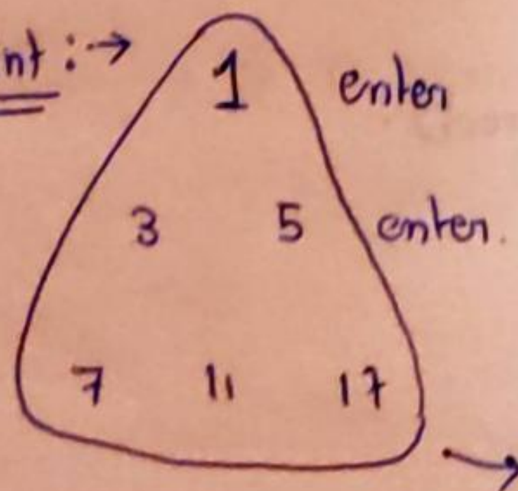


⊡ When our level ends we push NULL.

⊡ For NULL we do enter instead of printing anything.

Queue

Print :→



→ Ans.

Code →

```
void Level_Order_Traversal (node * root) {
        queue <node*> q;
        q.push(root);
        q.push(NULL);

        while (!q.empty()) {
            node * temp = q.front();
            q.pop();
            if (temp == NULL) {
                cout<< endl;
                if(!q.empty()) {
                    q.push(NULL);
                }
            }
            else {
                cout << temp-> data << " ";
                if (temp -> left) {
                    q.push (temp -> left);
                }
                if (temp -> right) {
                    q.push (temp -> right);
                }
            }
        }
}

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

3.

⊙ Types of Tree Traversals →

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

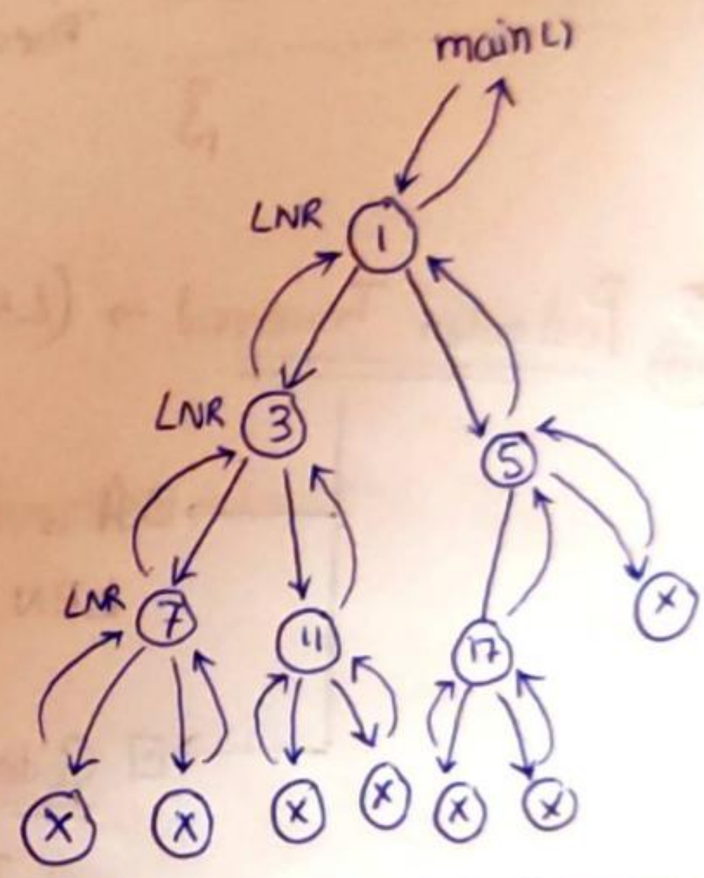
(v) Zig zag Traversal
(vi) Boundary Traversal.
(vii) Vertical order Traversal.
(viii) Morris Traversal.

⊙ Note →

- All traversal techniques ~~tha~~ take O(N) time and O(N) space complexity
- Only morrish traversal takes O(N) time and O(1) space.

(i) Inorder Traversal → (LNR)

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



7 3 11 1 17 5

↓

Inorder traversal

- Code →

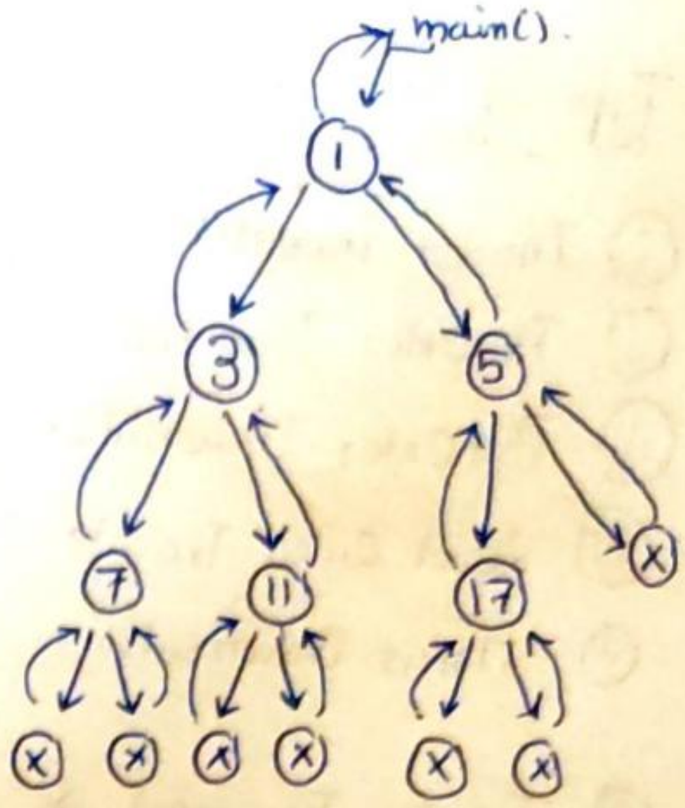
```
void Inorder (node* root){
    if (root == NULL){
        return;
    }
    Inorder (root → left);
    cout << root → data << " ";
    Inorder (root → right);
}
```


(ii) 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

↓

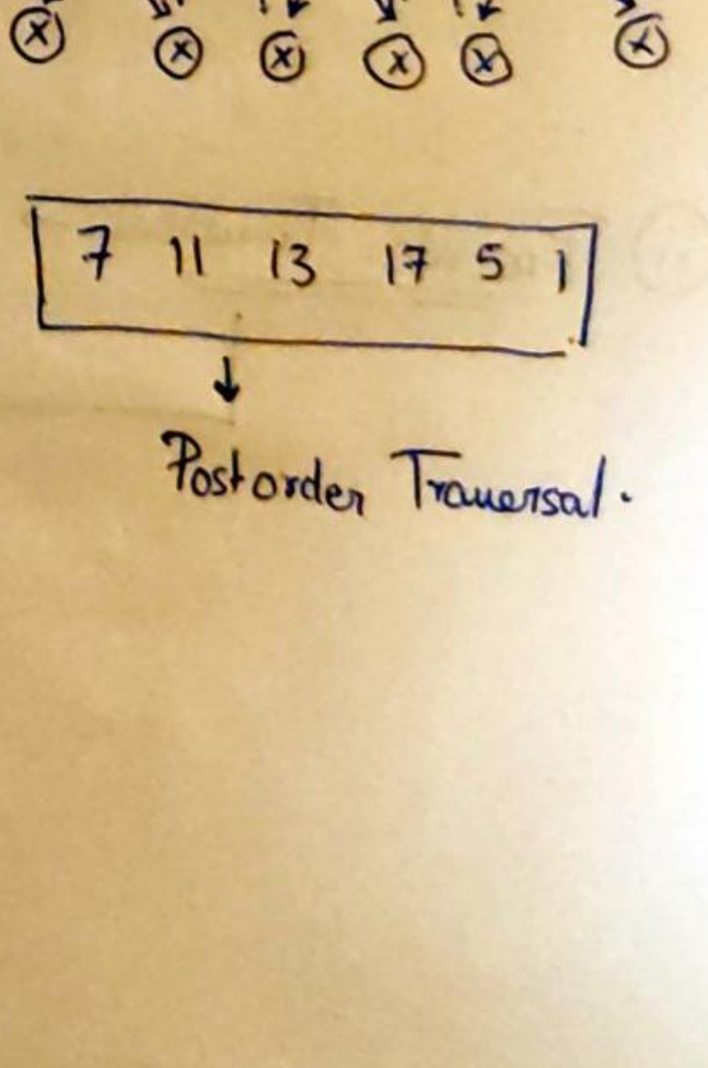
Preorder Traversal.

(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.

⊙ 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);
}
}
}
```

Step1: Create a root node and push it into the queue.

Step2: until our queue is empty, we will store it top variable node data and pop it out.

Step3: We will take two datas 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.

⊙ Height of a tree → ⊡ It is the longest path between the root node and a leaf node.

⊡ Time and Space complexity

Time complexity = O(height).

Space complexity = O(N).

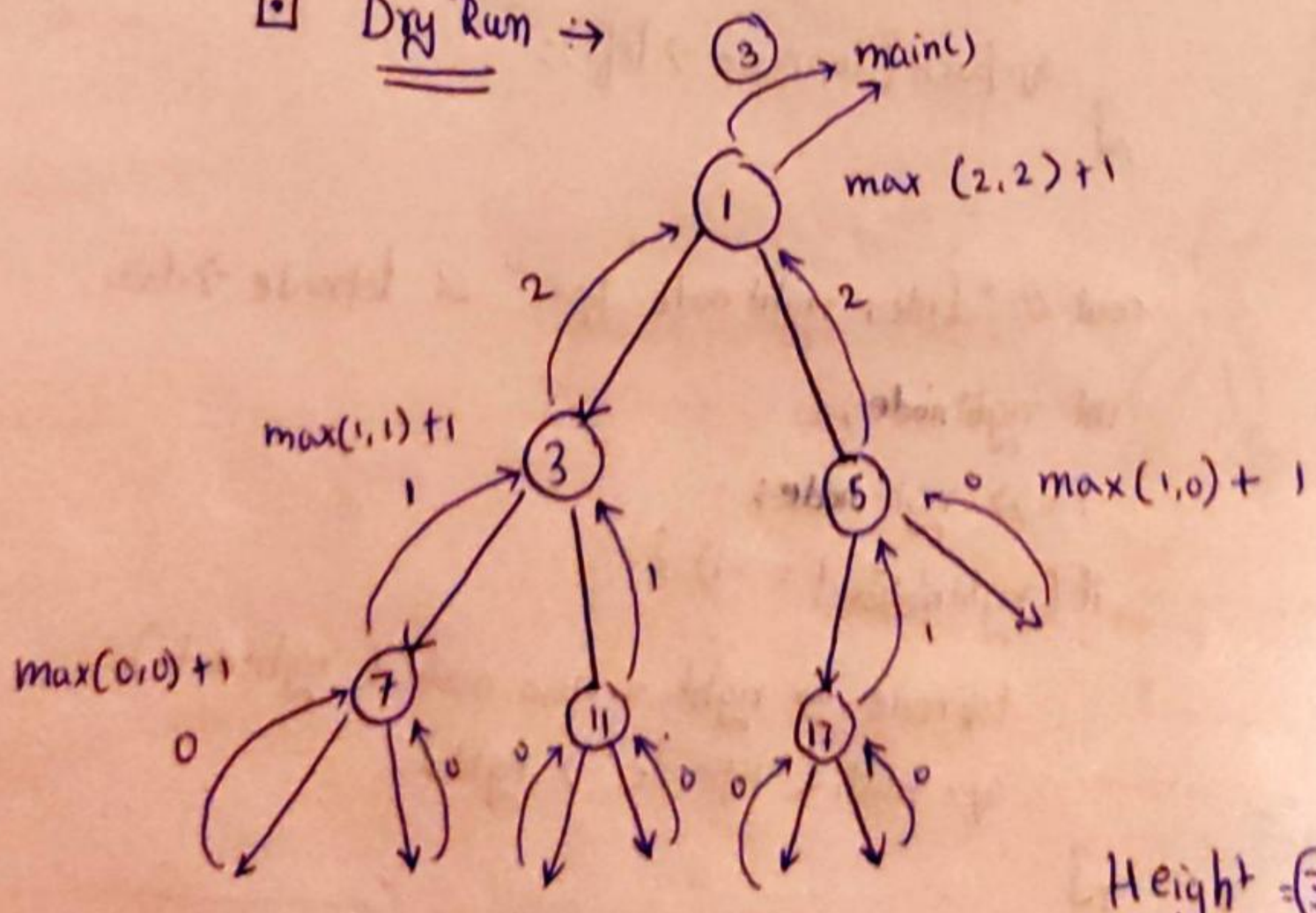
⊙ Note :→

For a skew tree i.e, where only left or only right nodes are present.
∴ T.C = O(N)

⊡ Code →

```
void height ( node* root) {
    if (root == NULL) {
        return 0;
    }
    int left = height( root → left);
    int right = height ( root → right);
    return max ( left, right) + 1;
}
```

⊡ Dry Run →



Height = 3

⊙ Counting the number of leaf present ina 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 :→

Leaf = ③

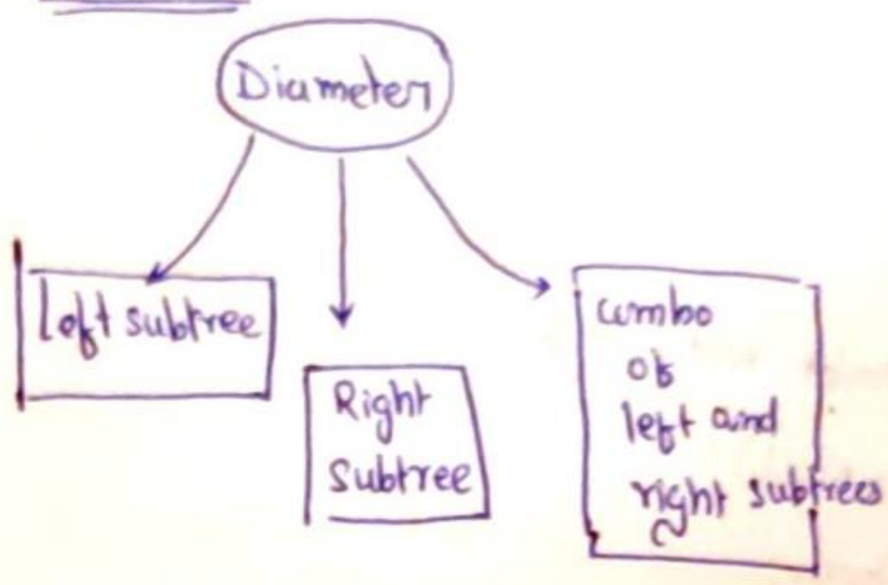
⊙ Diameter of Binary Tree

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

⇓

Maximum no. of nodes between two end nodes.

Approach 1.



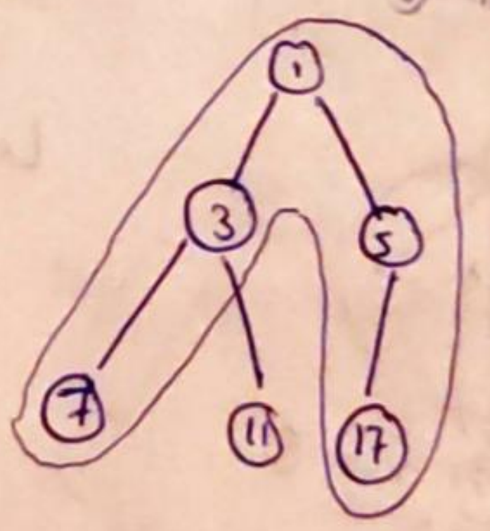
Code :→

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

    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 <int, int>.

Dry run :→



Diameter = 5.

- space complexity → O(h)

⊙ 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 height(left subtrees) - height(right subtrees) <=1 ;

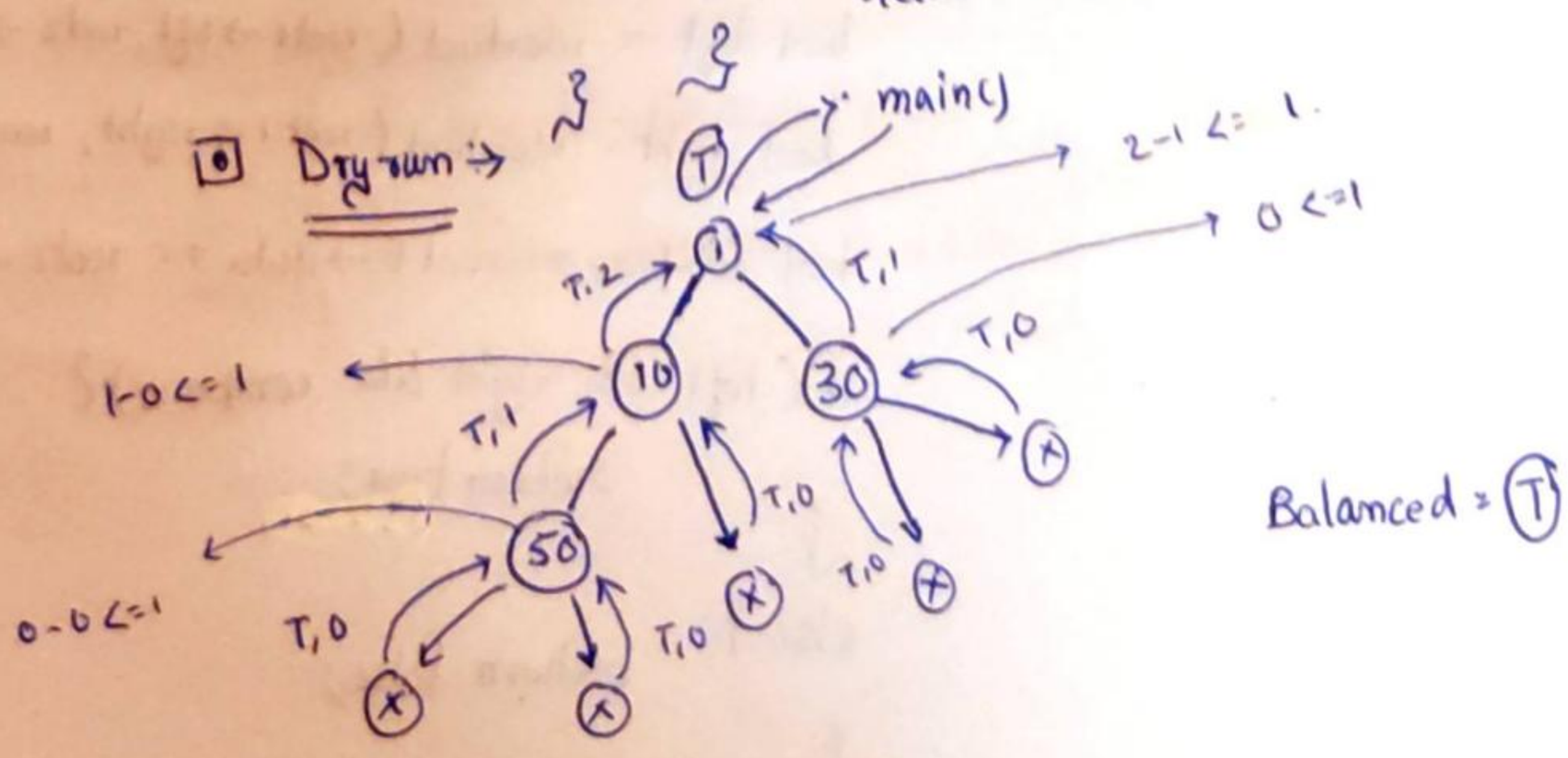
⊡ Code ⇢

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

⊡ Time complexity ⇢
O(n²) ⇢ But can be optimised using pair<int,int>

⊡ Space complexity ⇢ O(h).

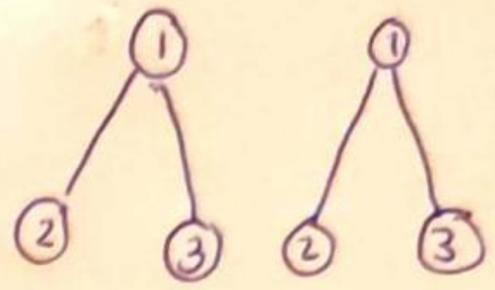
⊡ Dry run ⇢



⊙ Check whether the trees are identical or not.

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

Ex →



→ ⊡ Approach →

△ We will replicate same traversing movements ~~way~~ 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.

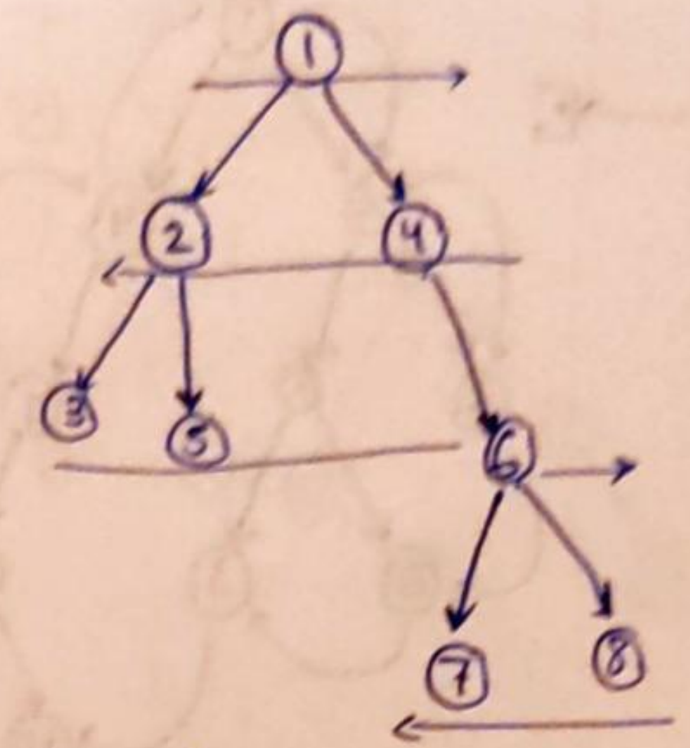
Note →

- Time complexity - O(N)
- Space complexity - O(h)

→ ⊡ Code →

```
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;
}
}
```

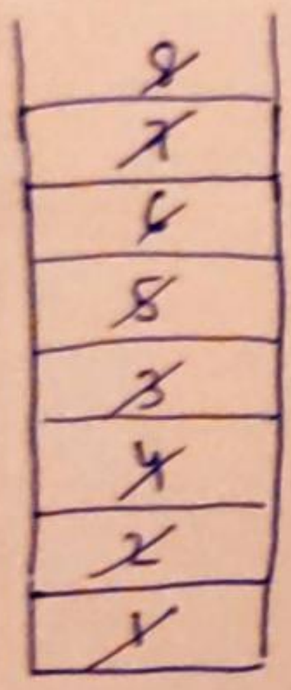
⊙ ZigZag Traversals :→ (Spiral)



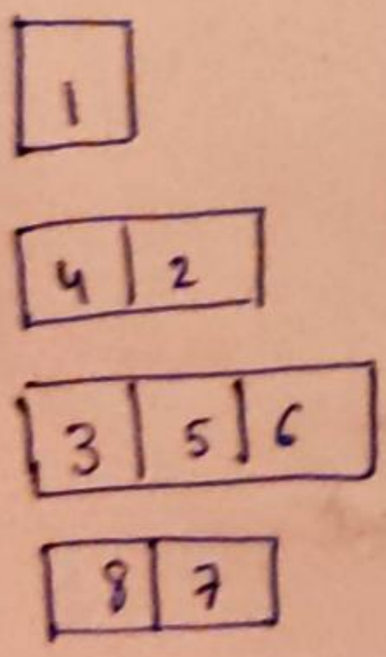
1 4 2 3 5 6 8 7

⊡ Approach :→

Queue



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



ans vector at each level.

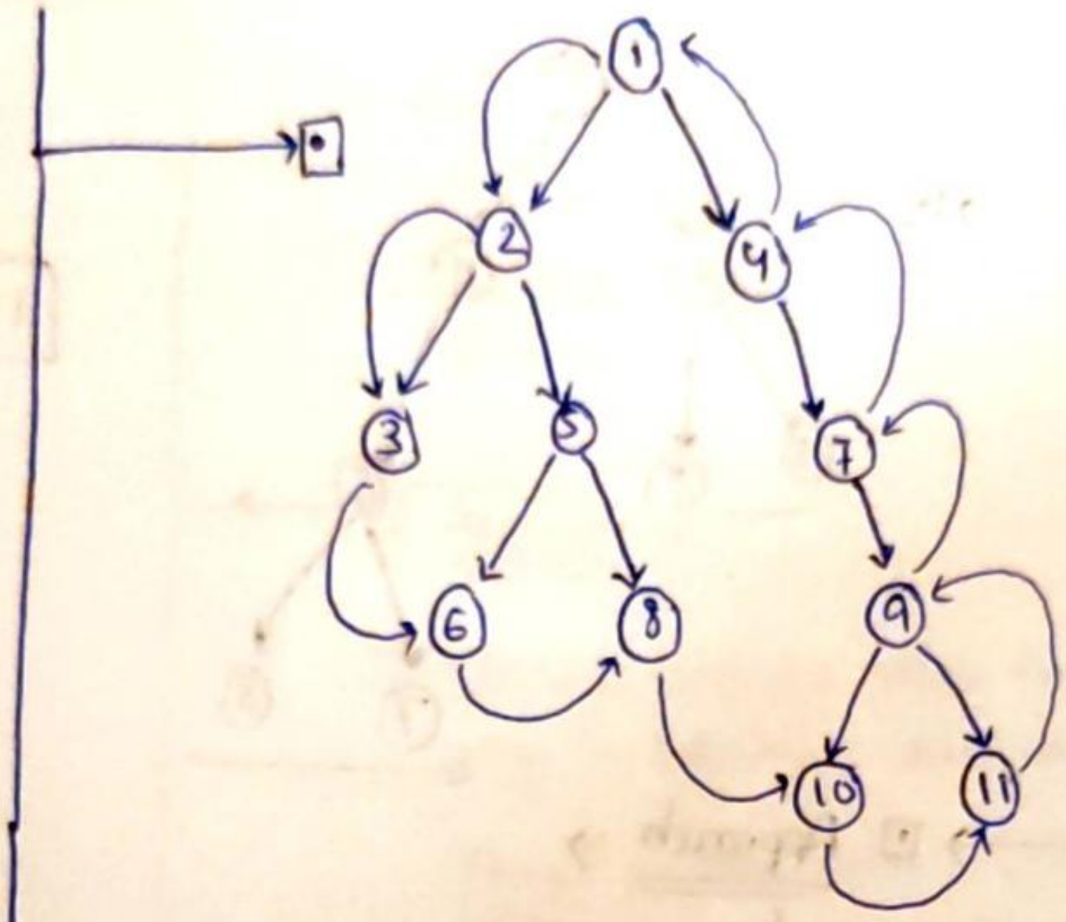
(i) We will use level order traversal Here in which we will use a queue. Now, we will simply do a level order traversal using a flag variable whose value updates after each level either 0 or 1. If value is 1 we do level order traversal from left else from right.

(ii) lastly we will store this ~~answer~~ node according to the sequence in a vector at each level and finally display it.

⊙ Time Complexity → O(N)

⊙ Space Complexity → O(N)

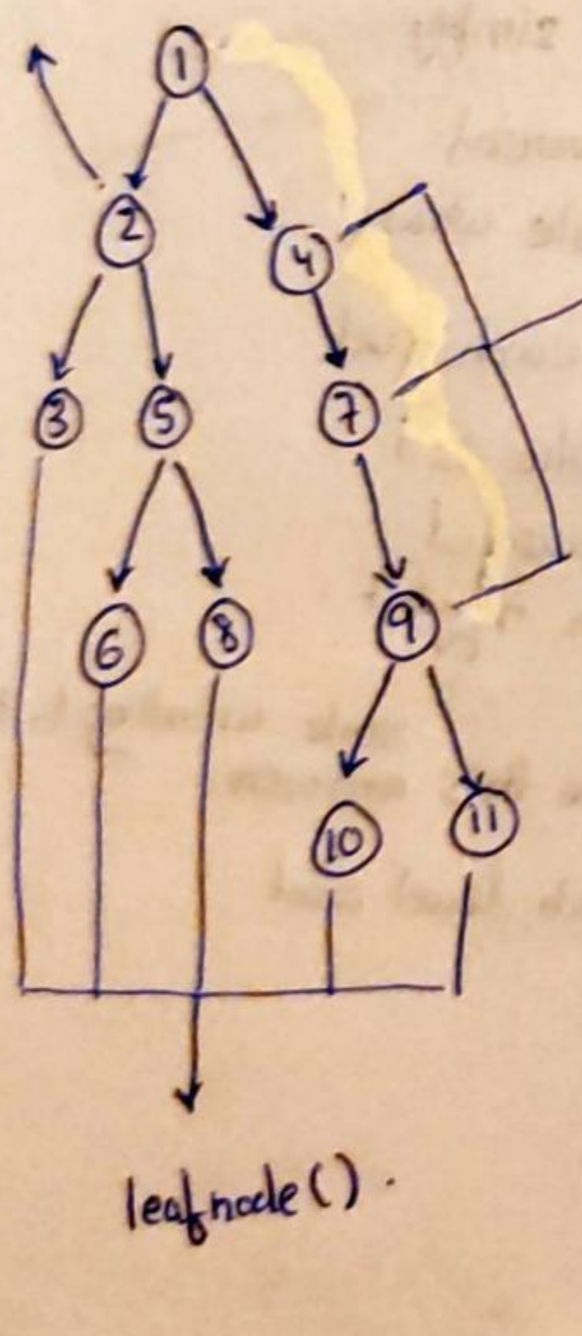
⊙ Boundary Traversal :→



1 2 3 6 8 10 11 9 7 4

Approach

leftpart()



leafnode().

rev rightpart()

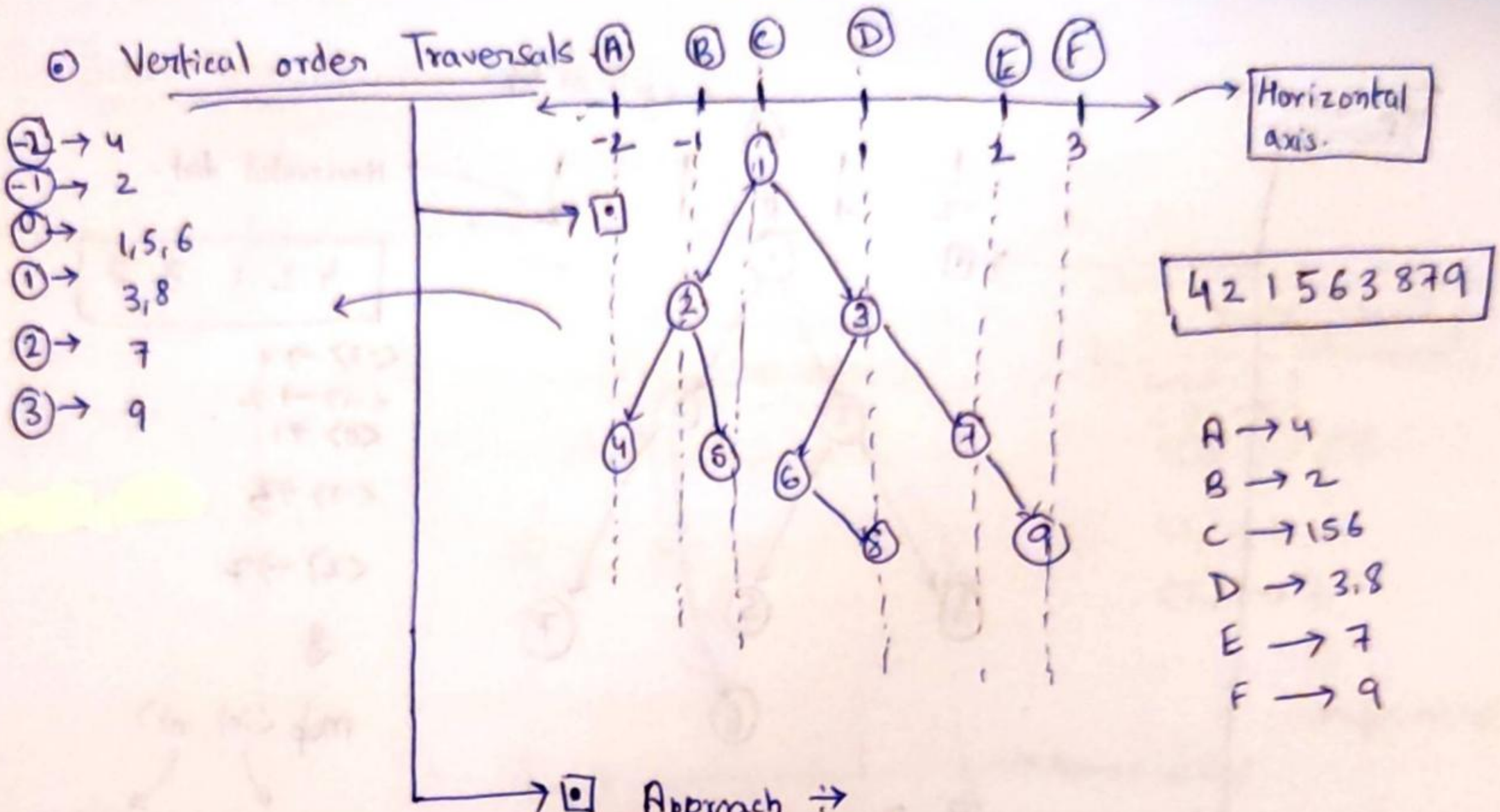
→ Divide the question in 3 function parts. One left part printing left node from root to the nodes before leaf.
Keep in mind in leftpart() if leftnode exist then only move in left else you can move in right.

→ leafnode() print all the leaf nodes. should be called for both left and right part.

→ reverse rightpart() prints all the nodes present in right parts if it exist or else it can also move to leftpart.

- Time Complexity → O(N)
- Space Complexity → O(h)

⊙ Vertical order Traversals



⊡ Approach :→

(i) We will heve use the level order traversal on a queue storing (Horizontal, level, node) and the we gonna do simple level order traversal.

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

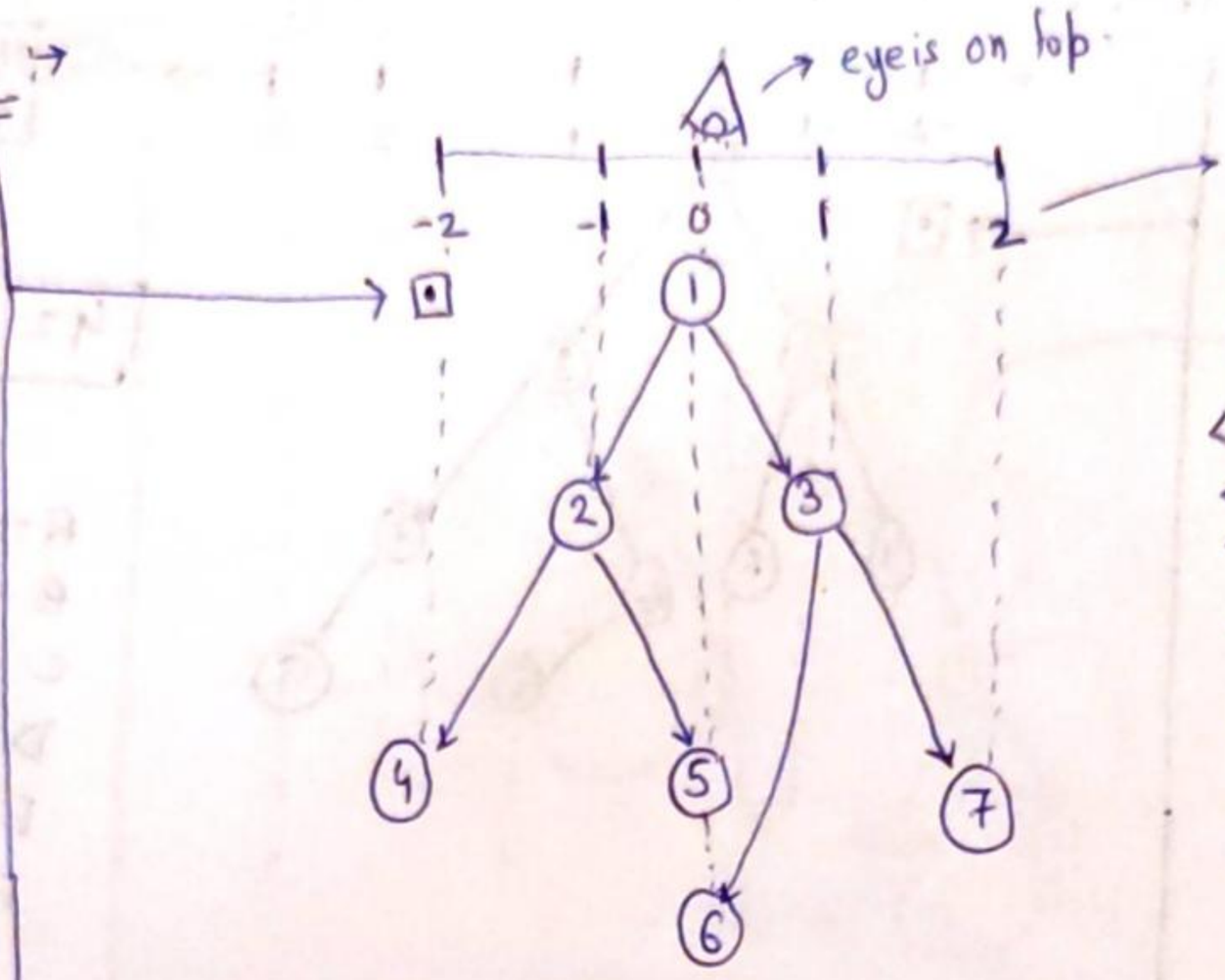
map < int, pair <int, vector<int>>> .

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

⊡ Time Complexity → O(N)

⊡ Space Complexity → O(N).

⊙ Top View :→



4 2 1 3 7

<-2> → 4
<-1> → 2
<0> → 1
<1> → 3
<2> → 7

⇓

map <int, int>

Horizontal dist

node→data

⊡ Approach :→

① → 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.

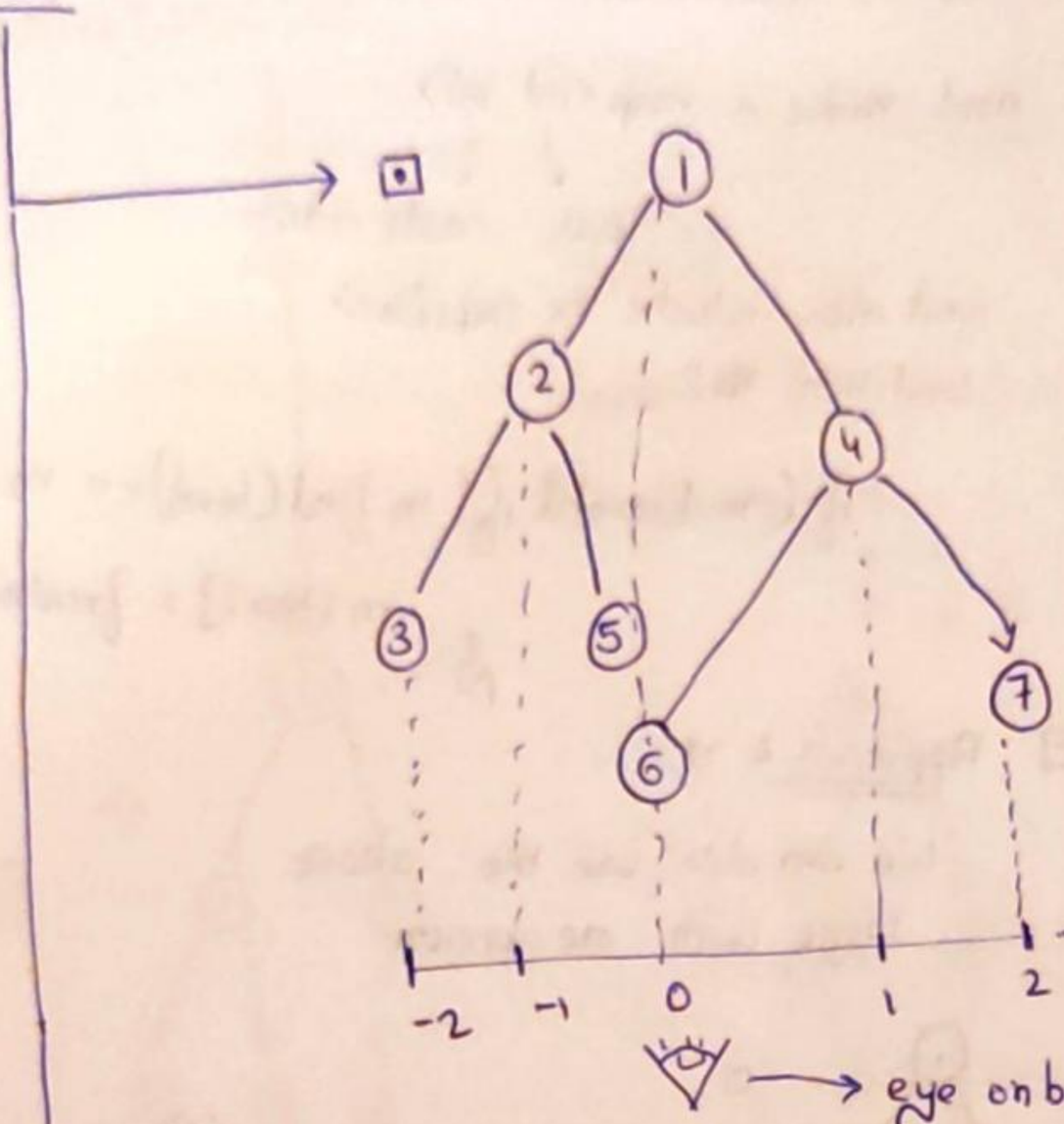
⚠ ⊙ Important condition :

```
if( m.find(horizontal) == m.end()){
    m[horizontal] = frontnode->data;
}
```

⊙ Time complexity :→ O(N)

⊙ Space complexity :→ O(N)

⊙ Bottom View →



3 2 6 4 7

<-2> → 3
<-1> → 2
<0> → X 6
<1> → 4
<2> → 7

⇓

map<int, int>.

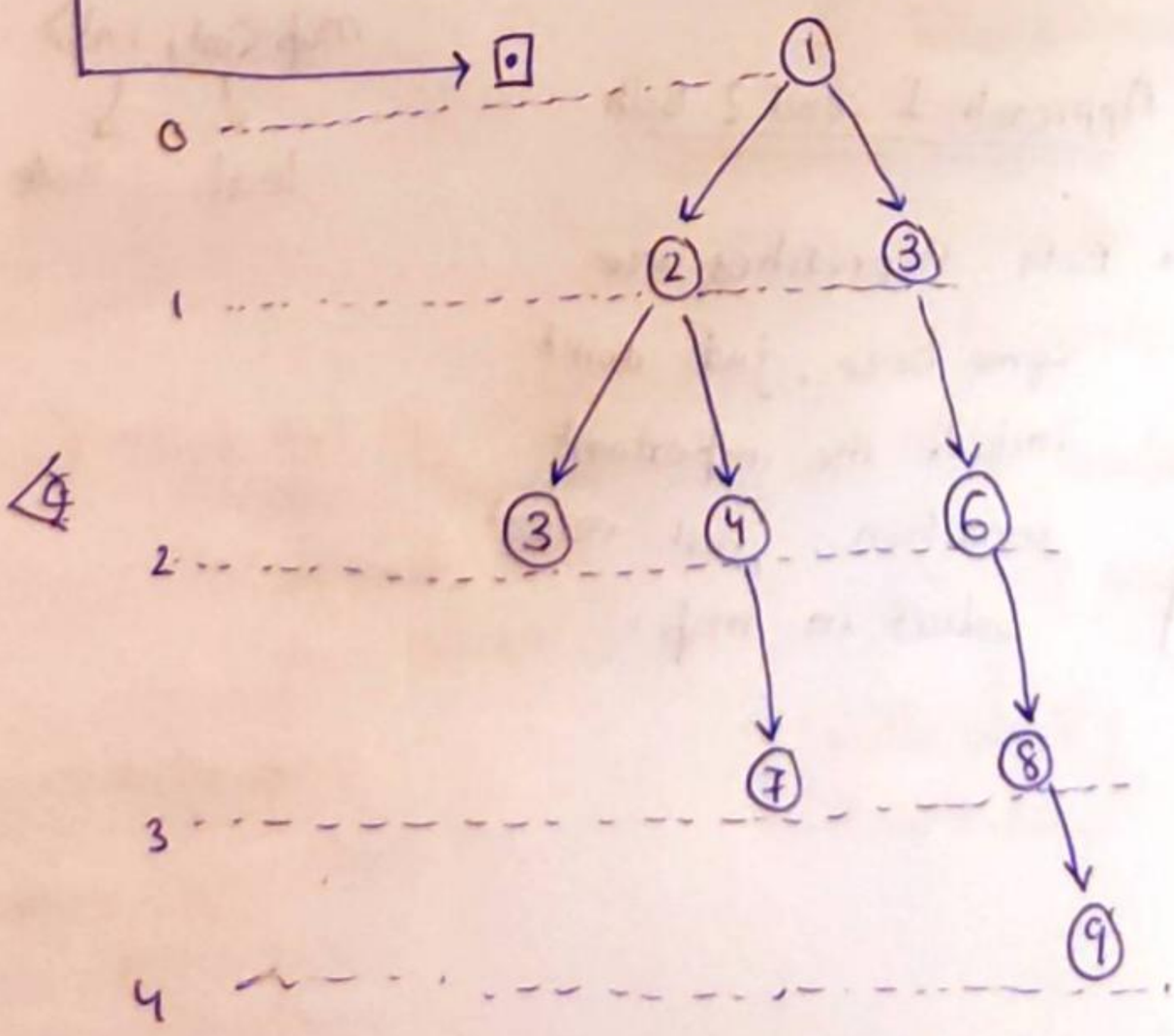
⊡ Approach →

(i) All would be same only exclud important condition and do the insertion inside the map.

⊙ Time complexity → O(N)

⊙ Space complexity → O(N)

⊙ Left view →



<0> → 1
<1> → 2
<2> → 3
<3> → 7
<4> → 9

1 2 3 7 9

map <int, int> :

level → node → data.

Approach 1 :→

- Just do level order traversal

and make a map<int, int>

level node → data.

and also include the important condition :

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

m [level] = frontnode → data;

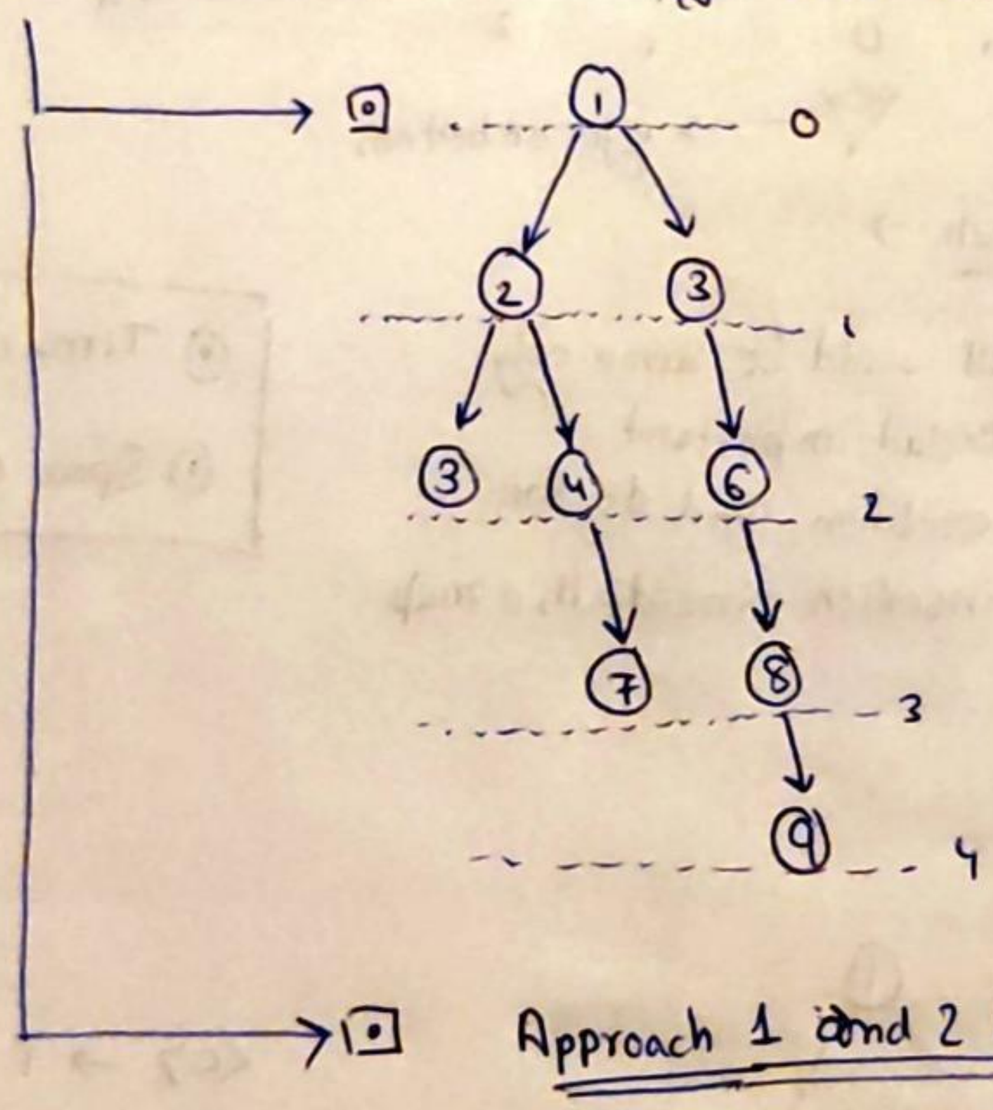
}

- Time Complexity :→ O(N)
- Space Complexity :→ O(N)

- Approach 2 :→

We can also use the above logic with recursion

- Right view :→



<0> → 1

<1> → ~~2~~ 3

<2> → ~~5~~ ~~4~~ 6

<3> → ~~7~~ 8

<4> → 9

1 3 6 8 9

map<int, int>

level node → 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 :→ O(N)
- Space Complexity :→ O(N)

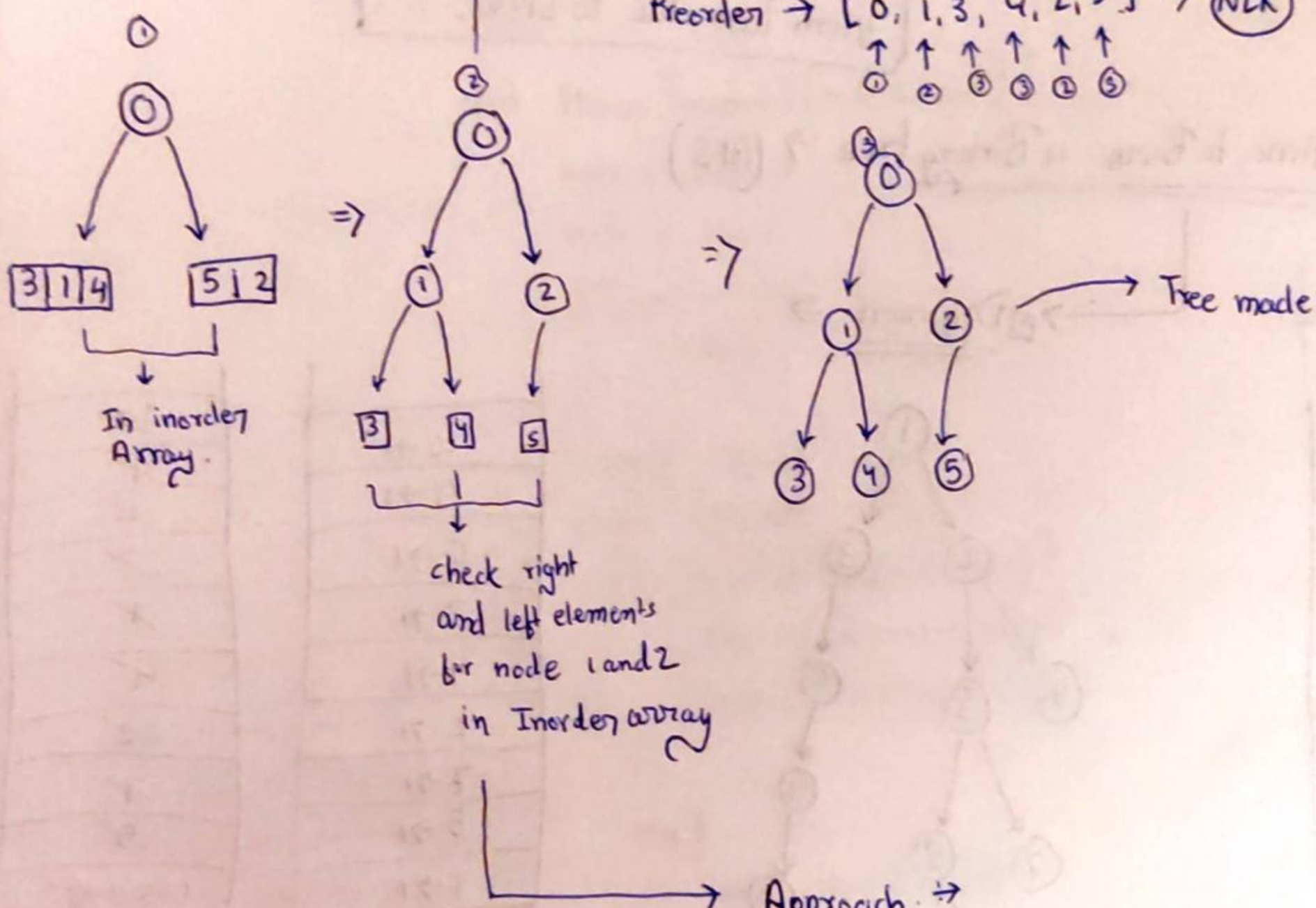
⊙ Construct a Binary Tree from Inorder and Preorder.

Remember in Preorder we traverse the array from beginning as NLR

→ Dry Run :→

Inorder → [3, 1, 4, 0, 5, 2] → (LNR)

Preorder → [0, 1, 3, 4, 2, 5] → (NLR)



→ Approach :→

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

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

inorder start → pos-1	pos+1 → inorder end.

⊙ Time complexity →

O(nlogn) → (if map is used for storing inorder elements with their indexes.)

⊙ Space complexity →

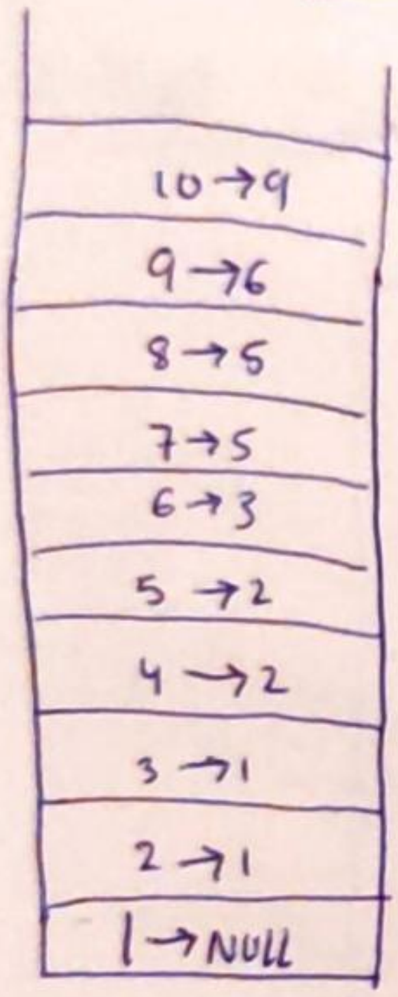
O(N)

⊙ 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.

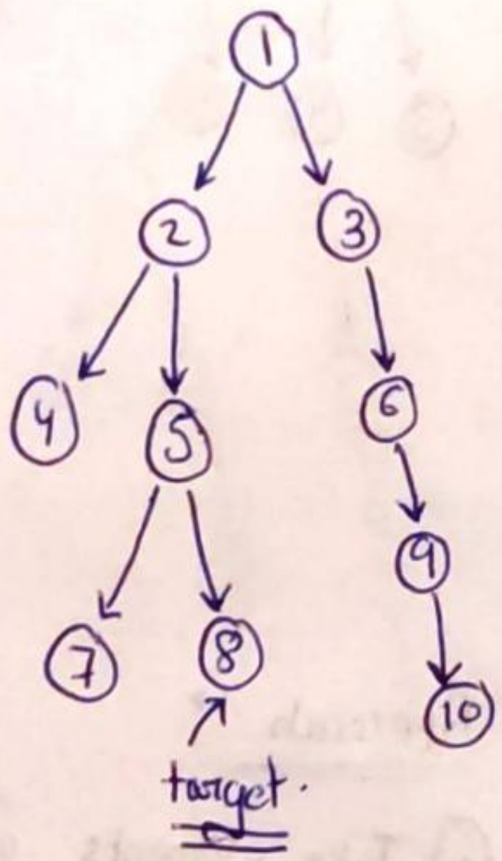
⊙ Minimum time to Burn a Binary tree → (BFS)

↳ ⊡ Dry run →



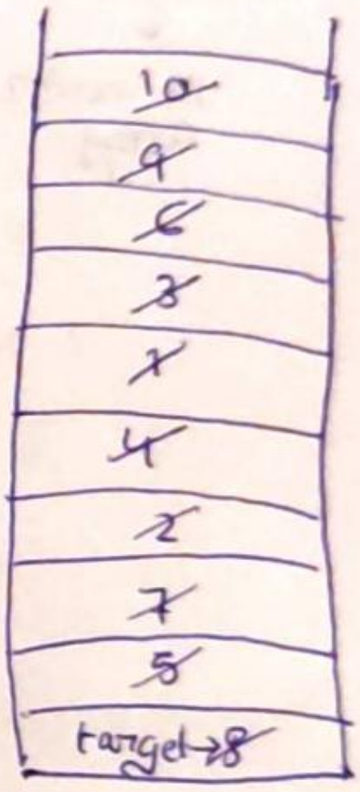
map <Node, parent>

STEP 1



visit map (Node, bool)

STEP 3



queue

STEP → 2

time = 0 1 2 3 4 5 6 7

∴ Ans time = 7

⊡ Approach →

(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.

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

⊡ Time Complexity → O(N)

⊡ Space Complexity → O(N)

→ if insert operation of of Node, parent map is O(1) else O(NlogN)

⊙ Morris Traversal →

- ⊡ Time Complexity = O(n)
  Space Complexity = O(1) → All other traversal techniques use O(n) space.
- ⊡ Code →

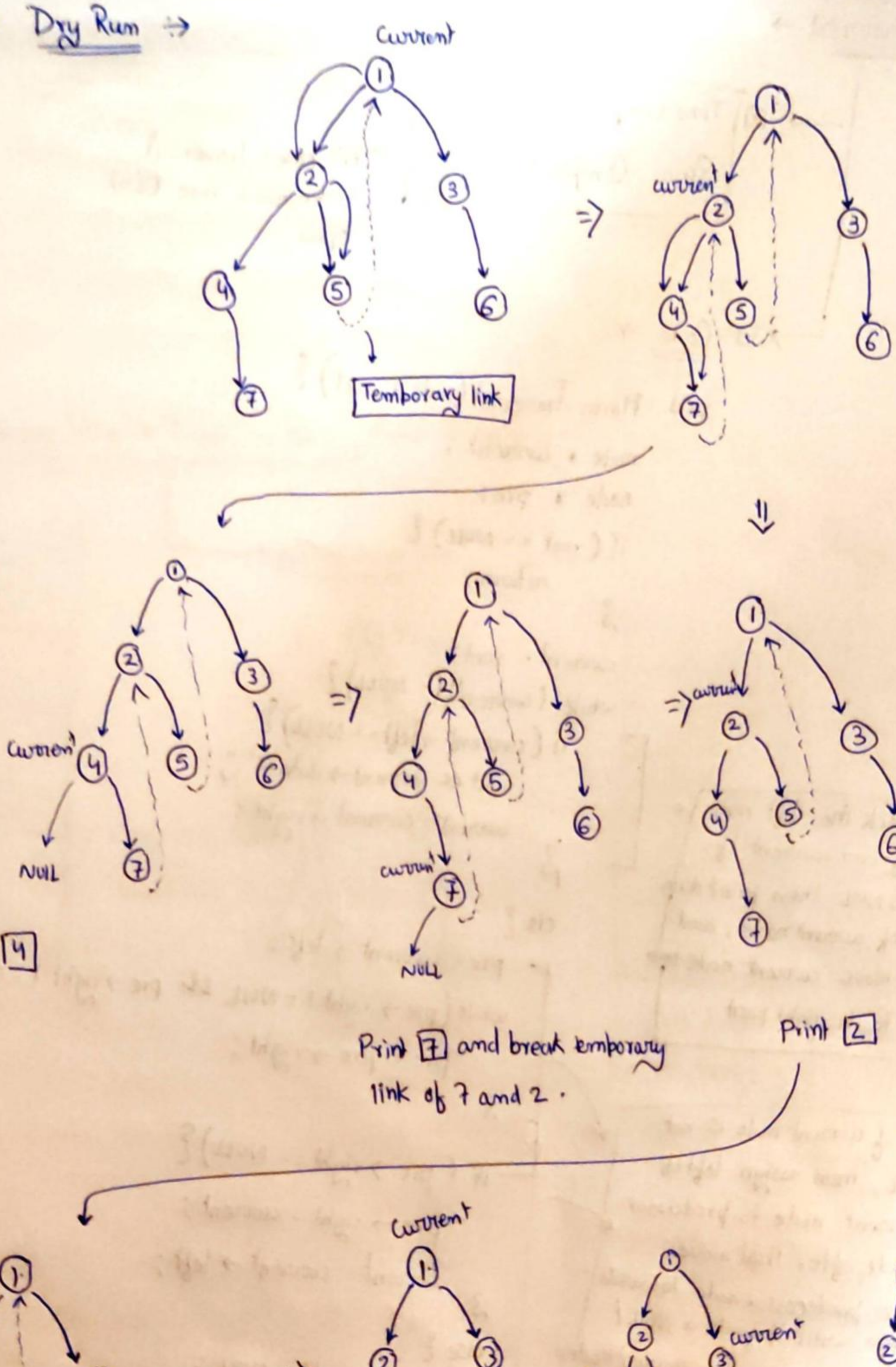
```
void Moriss Traversal (node * root) {
node * current;
node * pre;
if (root == NULL) {
    return;
}
current = root;
while (current != NULL) {
    if (current → left == NULL) {
        cout << current → data << " ";
        current = current → right;
    }
    else {
        pre = current → left;
        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 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 → right = current node.

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

Dry Run ⇒



⊙ Binary Trees →

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

- (i) Node
- (ii) Root
- (iii) Parent
- (iv) children
- (v) Sibblings.
- (vi) Anchestors
- (vii) Descendants.
- (viii) Leaf.

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

③ Write code to create a Binary tree using Reursion and level order Traversal.

④ Dissuss Approach, Dury 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 disscuss 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 Whatis Balanced Tree ? Discuss Approach, time, Space complexity. dryRun and Code.

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

(9.) Important Interview questions, discuss time and Space, dryrun, abraach, Code...

(i) Zizag Traversal.
(ii) Boundary Traversal.
(iii) Vertical order Traversal.
(iv) Top view and Bottom view.
(v) Left view and Right view.
(vi) Construct a Binary Tree from Inorder and Preorder Array.
(vii) Construct a Binary Tree from Inorder and Postorder Array.
(viii) Calculate minimum time required to burn a Binary Tree. using BFS.
(ix) Sum of Longest Bloodline.
(x.) LCA of Binary Tree.
(xi.) K sum paths.
(xii) K^{th} Ancestor.
(xiii) Maximum Sum of Adjacent Nodes.