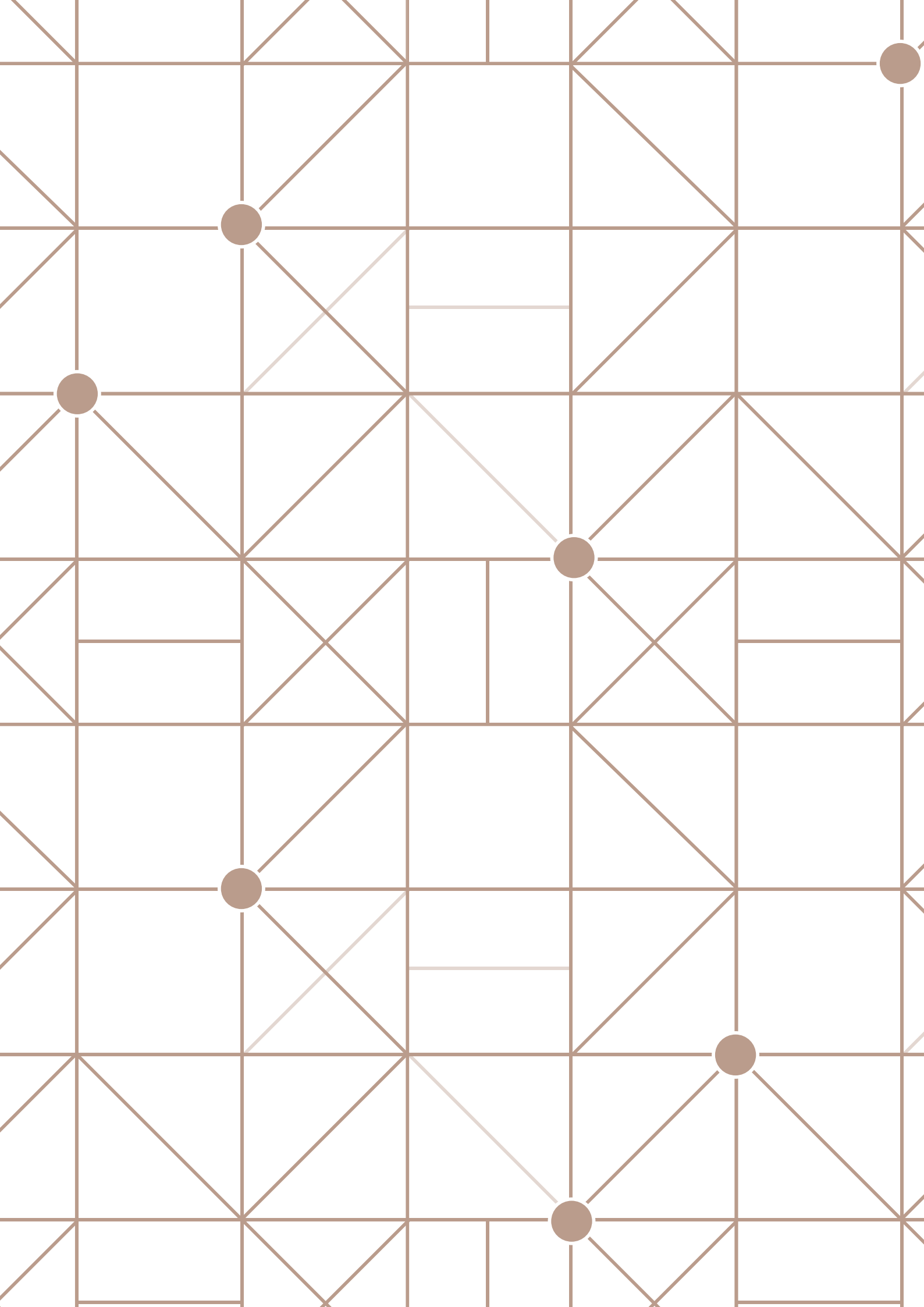
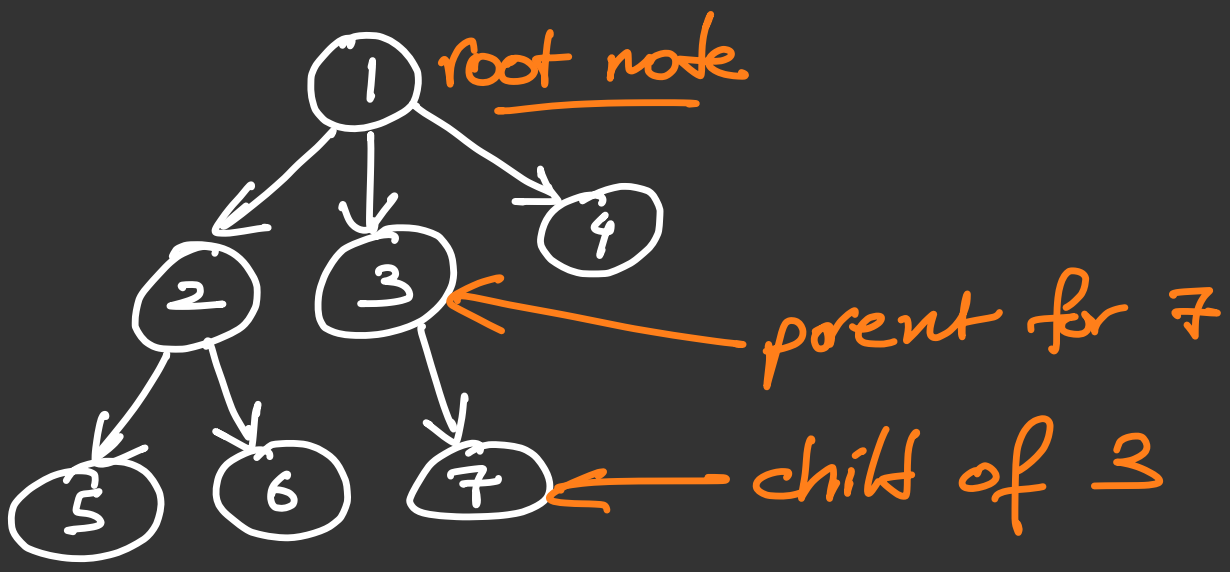


Binary Tree

Lecture 62

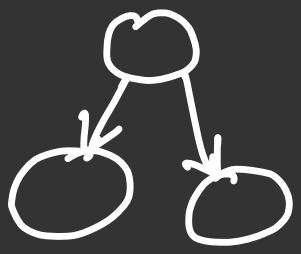
Tree

↳ non-linear Data Structure



node — element of a tree

```
node
{
    int data;
    node *left;
    node *right;
};
```



siblings — same parent (2, 3)

ancestor — all the nodes above a particular node

decendant — all the nodes below a particular node

leaf node — node which has no child (4, 5, 6, 7)

N-Ary tree

↳ multiple child

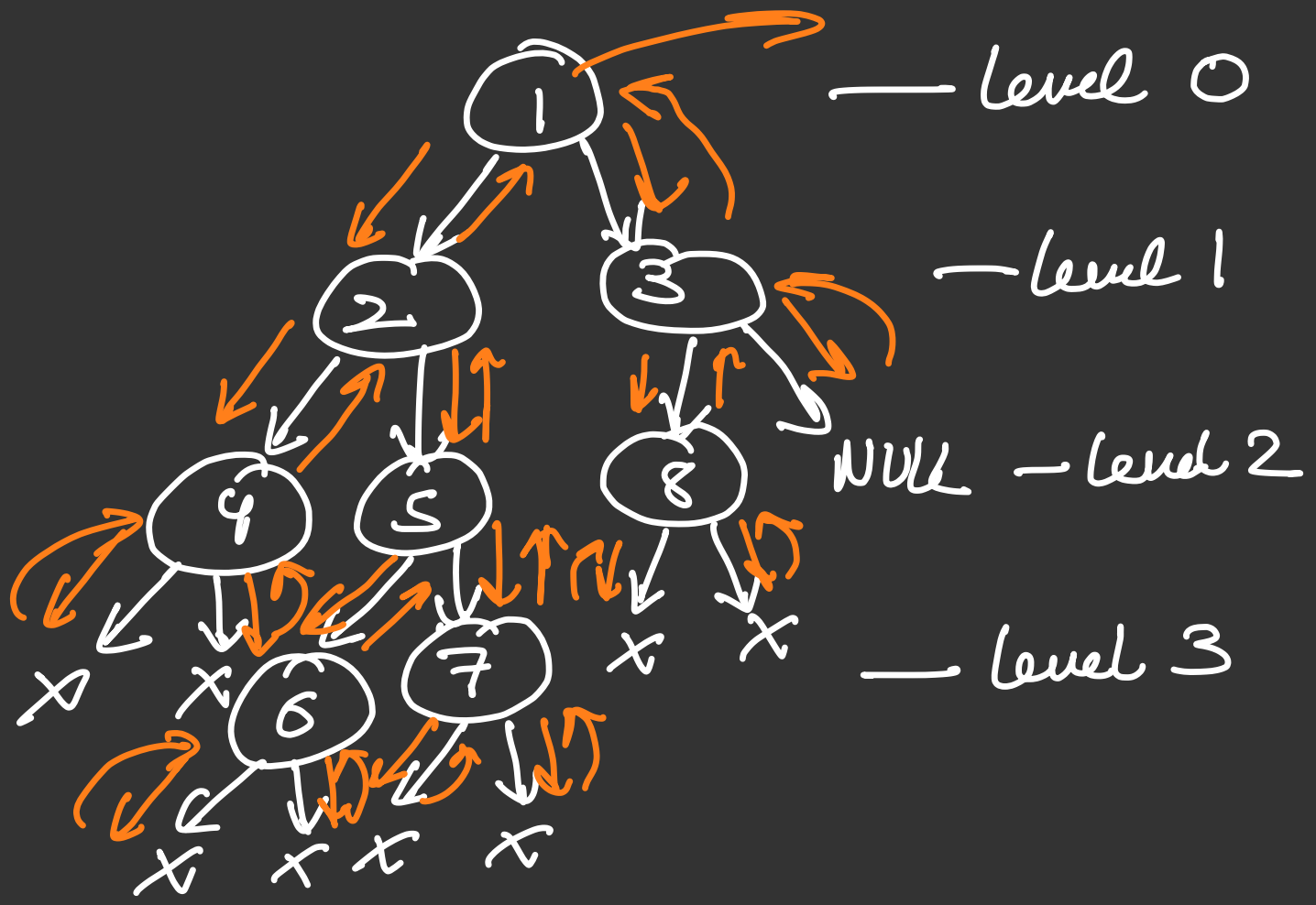
```
node
{
    int data;
    list<node*> child;
}
```

☆ But binary tree → 2 children

Creation / Build Binary Tree

→ We do this recursively

1. For the main node (root node)
2. Then recursively for left and right node
3. If entered `data = -1`, considered as NULL.



Level Order Traversal

o/p →

1
2 3
4 5 8
6 7

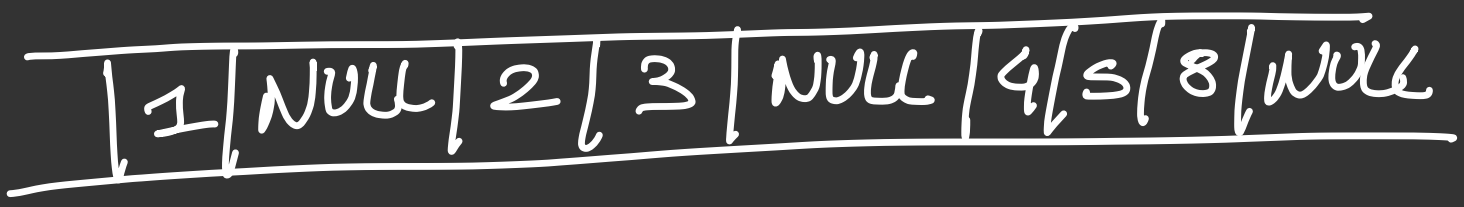
→ Separators as NULL for endl;

1. Use Queue
2. Push root and NULL
3. While (Queue is not empty)
   {
   temp = q.front();
   → if NULL
   cout << endl;
   and now if queue is

not empty
push NULL

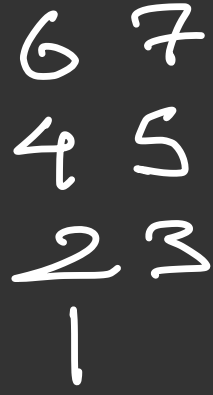
→ cout << temp->data
Push left and
right if they
are not NULL

3



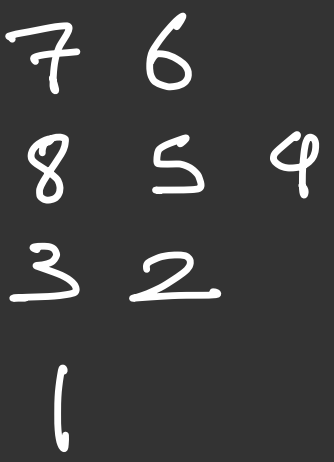
→ binaryTree.cpp

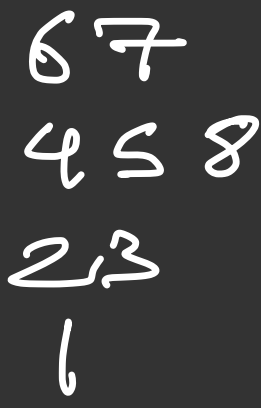
Reverse Level Order Traversal



→ We store queue in stack and print from stack

☆ Also to fix





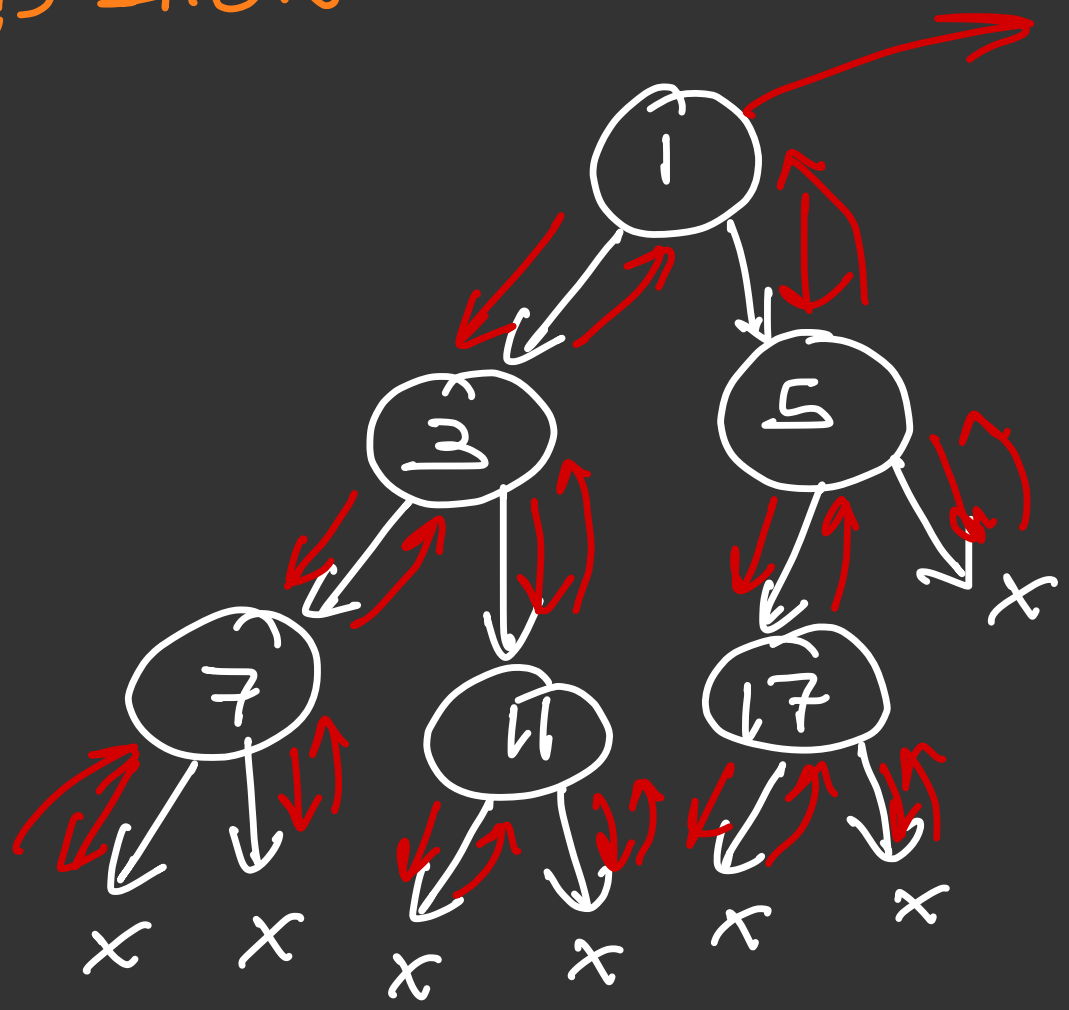
→ we first push right and then left.

→ binaryTree.cpp

InOrder / PreOrder / PostOrder

LNR NLR LRN

(i) InOrder



7, 3, 11, 1, 17, 5

⟶ inOrder.cpp

⟶ preOrder.cpp

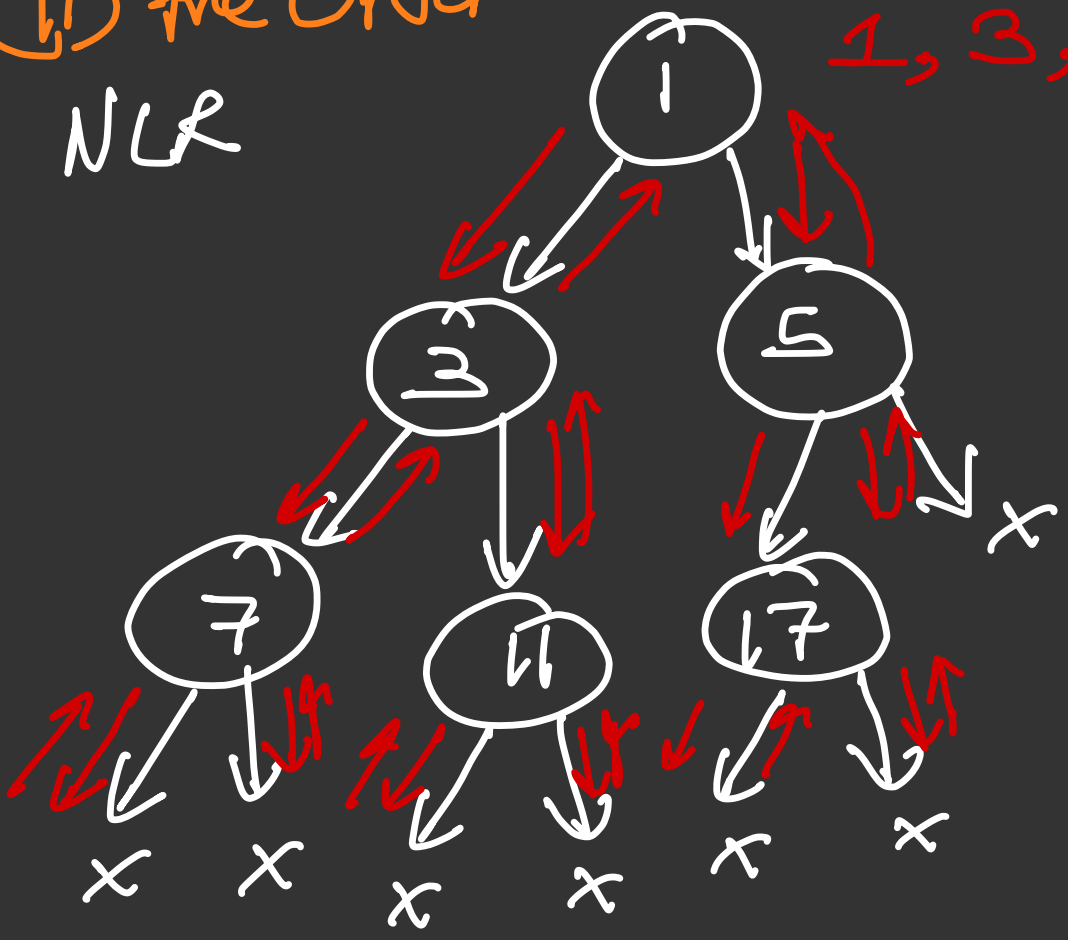
⟶ postOrder.cpp

TC = O(n)

(ii) Pre Order

NLR

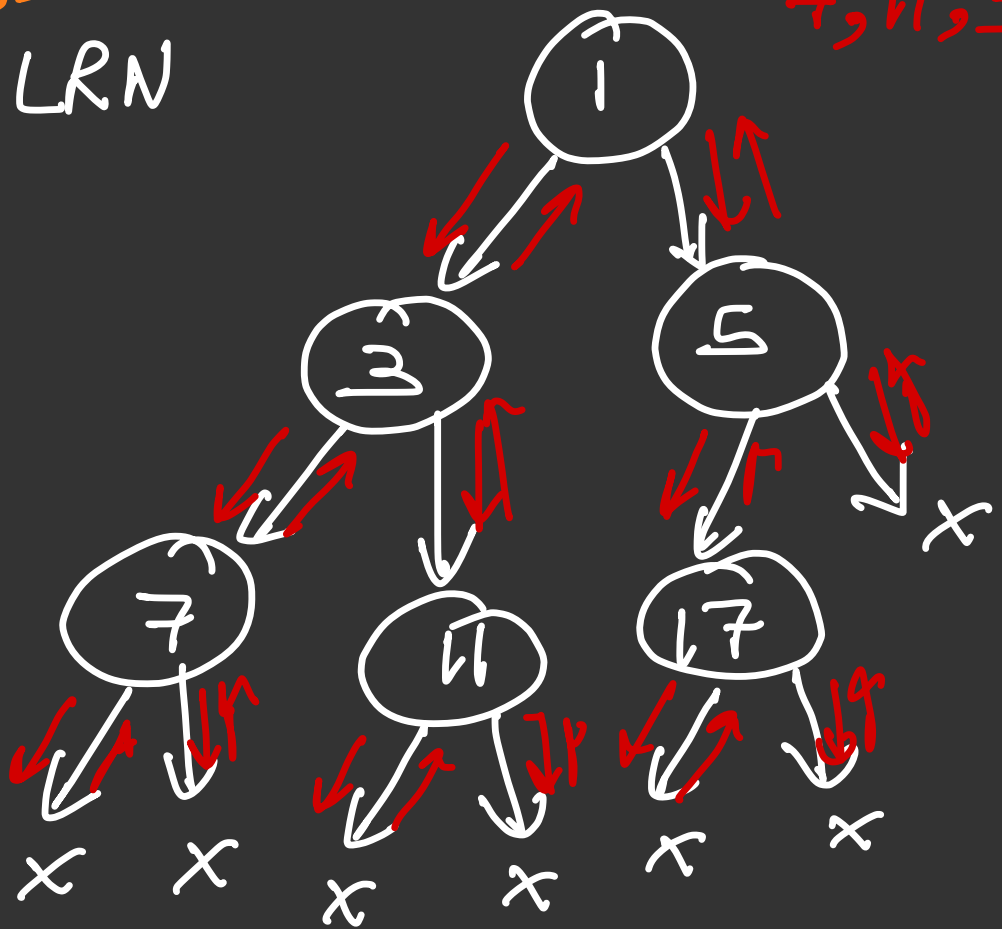
1, 3, 7, 11, 5, 17



(iii) Post Order

LRN

7, 11, 3, 17, 5, 1



Building in Order Level

☆ Everytime it is concerning Order Level → Queue.

① Store data in root then push it into queue

② While Queue is not empty
{
→ front in temp. then, POP

→ For left input
data
if data != -1
store in temp → left

→ Same for temp → right
}

✡ Make sure to pop from the queue after storing it in temp.

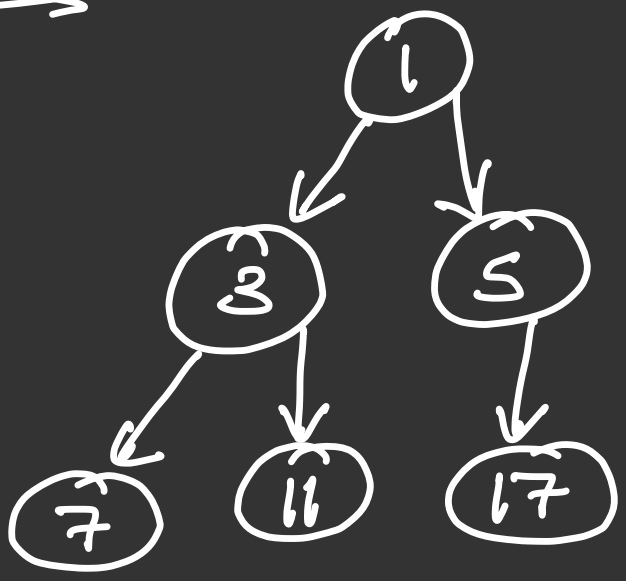
Thus, one by one elements come in queue and left and right are pushed in queue., forming Order.

⟶ buildLevelOrder.cpp

Lecture 63

Height / Depth of the Binary Tree

i/p →



h = 3

Height

↳ longest path b/w root node and leaf node.

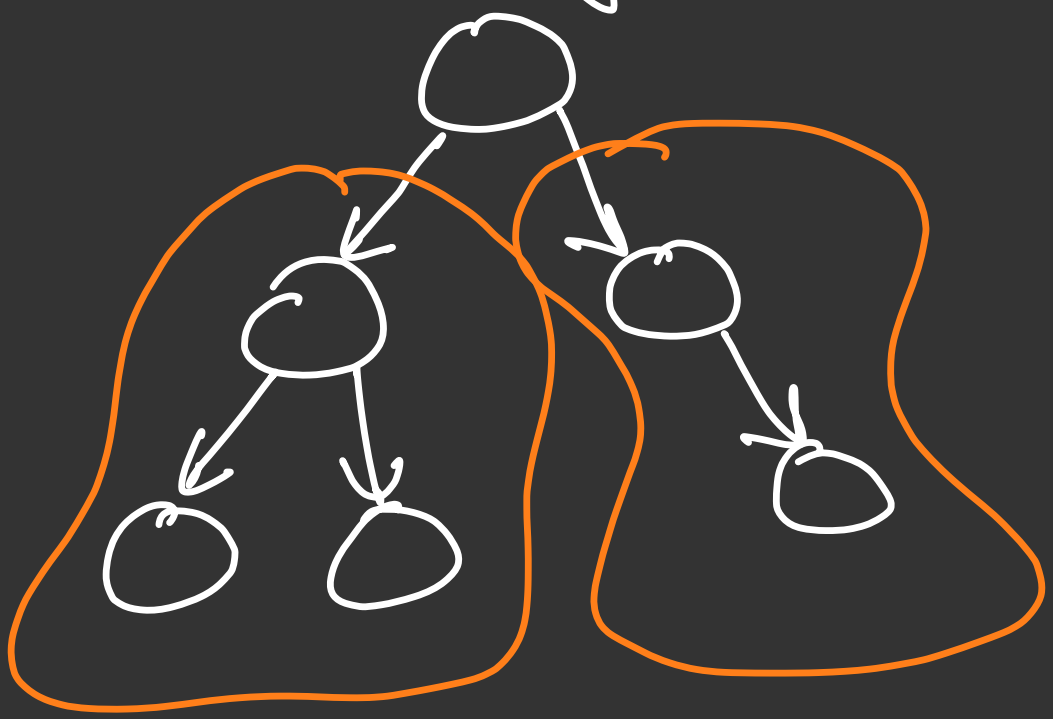
Approach 1 →

What I was thinking like in Ordered printing of nodes whenever NULL comes in queue we add +1 to a variable

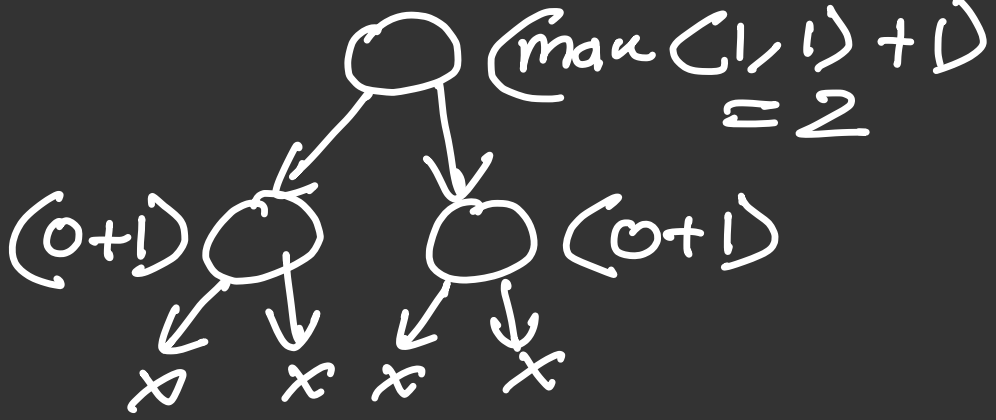
and in the end is the answer.

Approach 2 ⟶

Using recursion

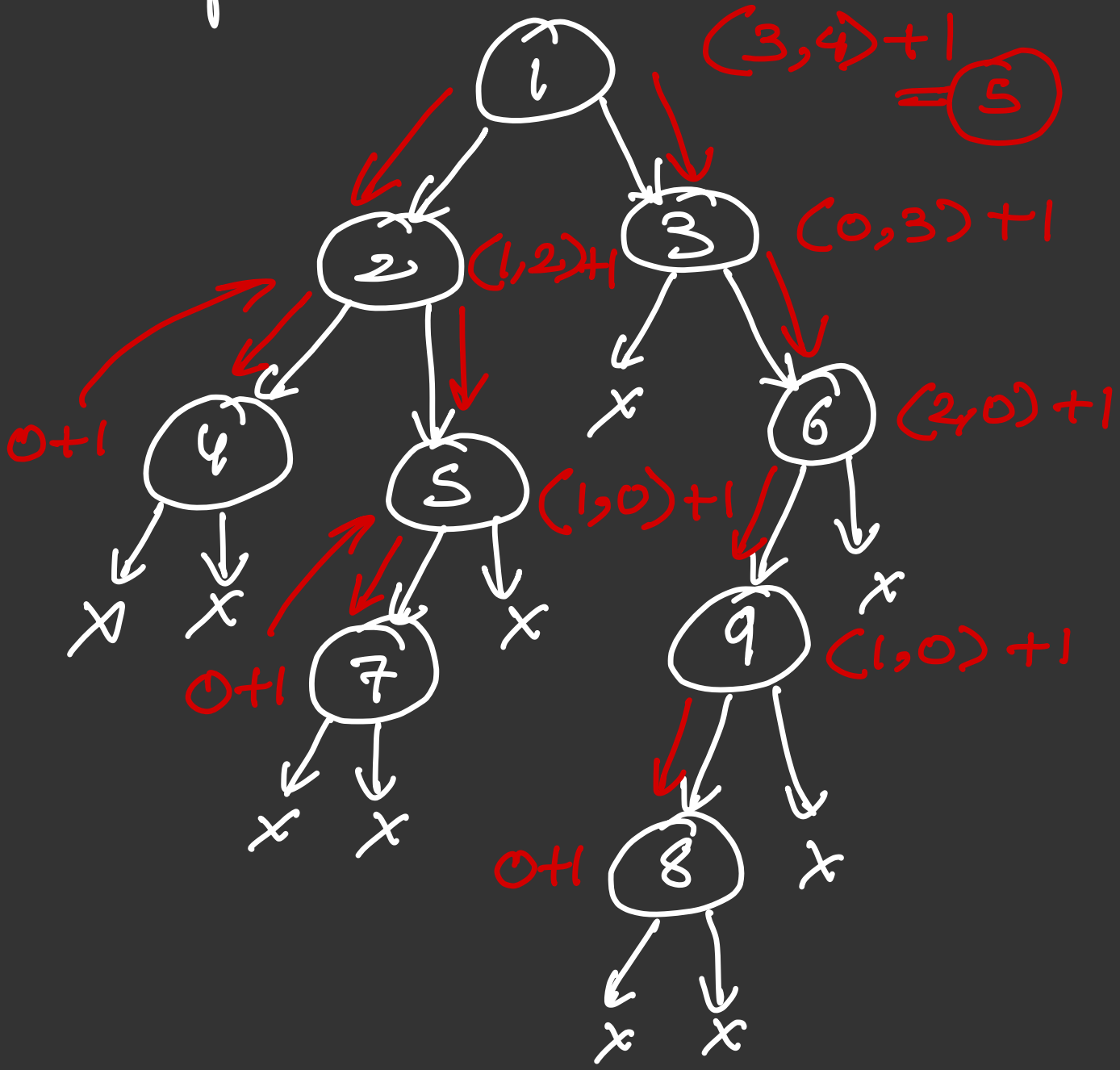


height = max (left subtree, right subtree) + 1



⟶ height_of_Tree.cpp

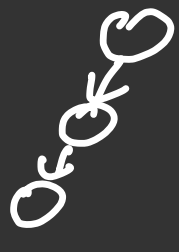
example:



⟶ Every node is visited single time

TC = O(n)

SC = O(height)

* But for skewed tree  SC = O(n)

Diameter of the tree

↳ longest path b/w two nodes

Approach 1 →

Three possible diameter can be there

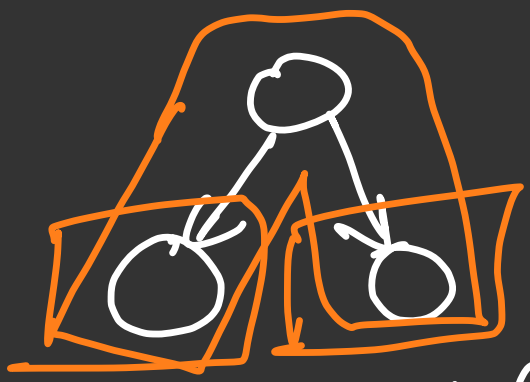
↳ in the left subtree
↳ in the right subtree
↳ else height (left subtree) + height (right subtree) + 1

But, because of height our TC = $O(n^2)$

→ diameter_of_Tree.cpp

Approach 2 ⟶

In Approach 1, for each node,



we are calculating left subtree, right subtree and the whole.

So, the diameter of whole = (left dia) + (right) + 1

Thus, calculating height also simultaneously

Using

```
pair <int, int> ans;
```

- first → height
- second → diameter

Check for Balanced Tree

↳ Difference b/w heights of subtrees should not be more than 1.
FOR ALL NODES.

Approach 1 →

We go to each node and find height of node → left and node → right then check, but calculating height for each node

TC = $O(n^2)$

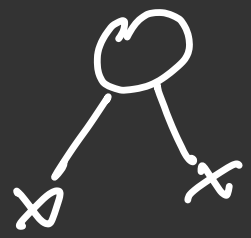
☆ Starts at leaf node, then move upwards, and for a node to be balance, left and right also needs to balanced
↳ recursion.

Approach 2 ⟶

Use that pair method to calculate height and the ans, for each node at the same time.

$$TC = O(n)$$

① Imagine leaf node



⟶ thus base condition

```
if (root == NULL)
    return <0, true>;
```

② Now calculate balance for left and right nodes

⟶ recursive calls.

(3) What ever you get find the height of the current node

⟶ max(left, right) + 1;

(4) And, check if any of the left and right are unbalance return false in bool

Else check the condition abs(-) <= 1

Return in pair.

⟶ balanced_Tree.cpp

☆ Thus wherever, it is about all NODES ⟶ pair

Identical tree or not ?

Approach 1 →

☆ Keeping in mind we start at the leaf node.

(i) if both NULL
↳ true

(ii) if both not NULL
→ both data equal
↳ then we check for left and right

→ else
↳ return false

(iii) Only one is NULL
which means return false;

TC = O(n)

→ identical_Tree.cpp

Sum Tree

Approach ⟶

* Since for all Nodes
  ↳ we use pair <int, bool>

1. If leaf node
   ↳ return node→data
2. If any one subtree is NULL
   Thus, if node == NULL
   return 0.
3. Both subtree bool has to be true and match the condition

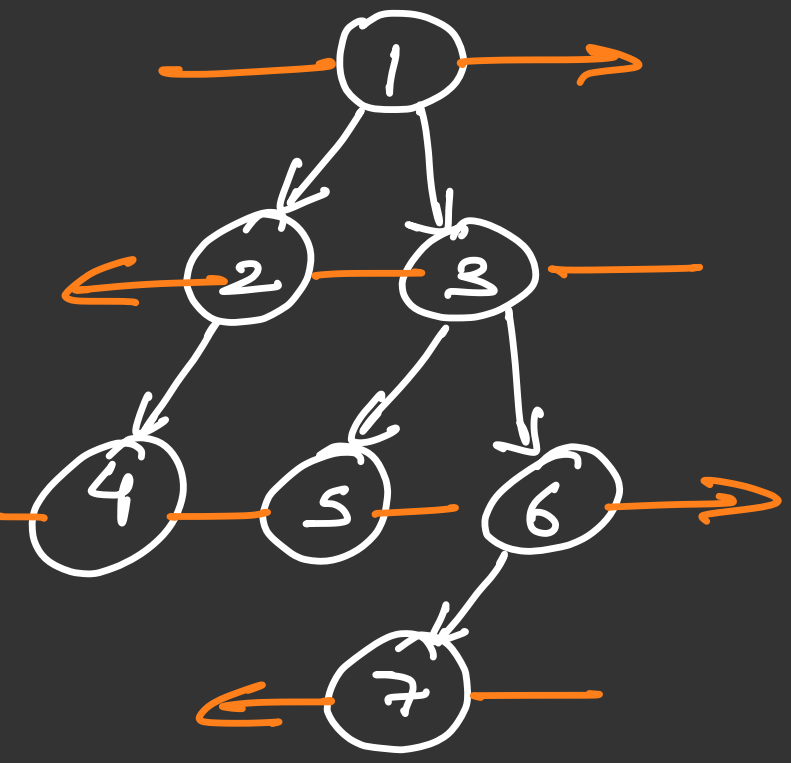
root→data == left.first + right.first

⟶ sumTree.cpp

TC = O(N), SC = O(height).

Lecture 64

Zig-Zag (or) Spiral Traversal



Approach 1 & 2 → My approach

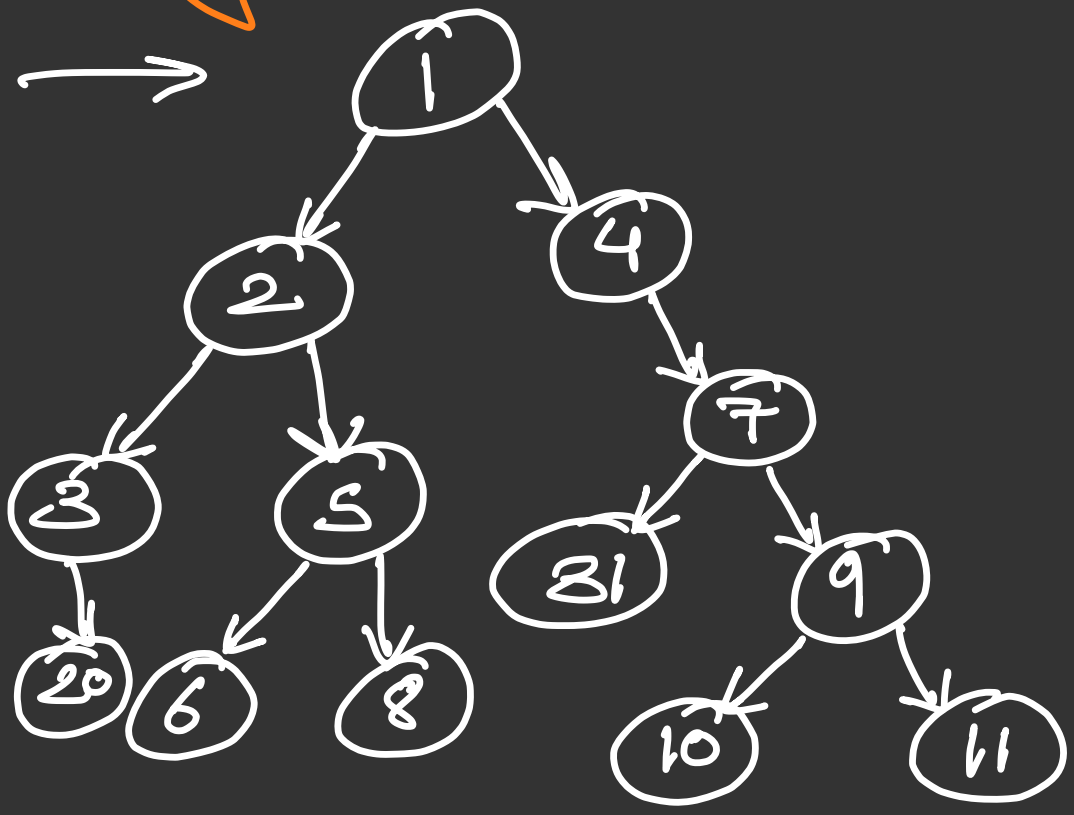
Approach 2

↳ A better way of writing, and comparitively a little less TC.

→ zigzag_Traversal.cpp

Boundary Traverse

i/p →



o/p → 1, 2, 3, 6, 8, 10, 11, 9, 7, 4

Approach 1 →

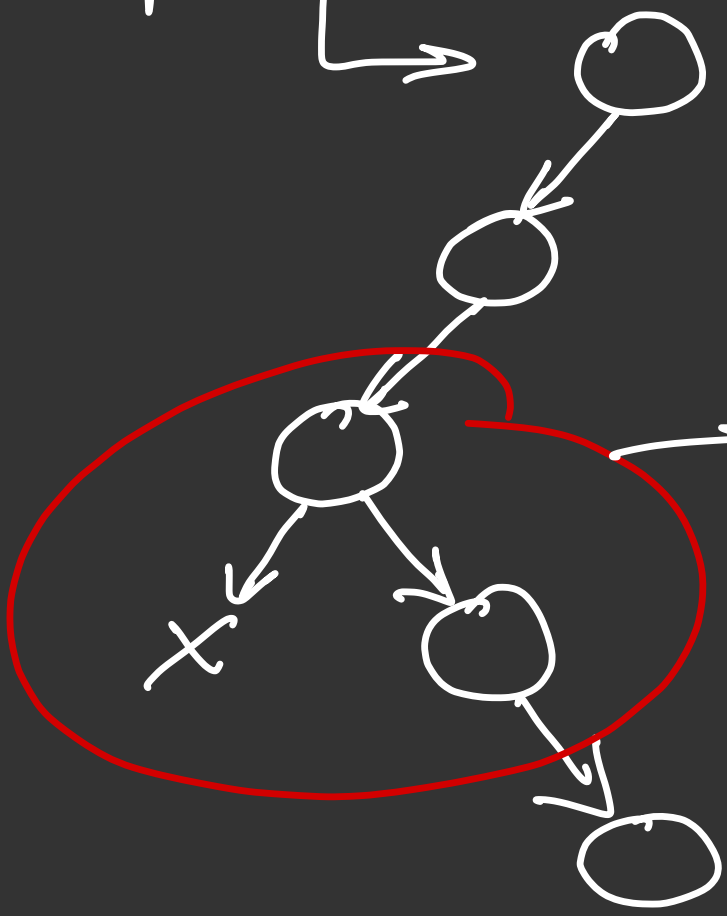
Divide in three parts

(i) Print all the left parts except leaf nodes

(ii) All leaf nodes

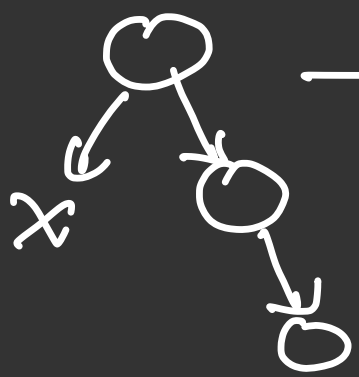
(iii) Print all right parts except leaf node in reverse order

☆ Special case



→ thus even while going only for left if root → left is NULL go, for right

Thus, we cannot include the root node in this



→ will cause an error.

Seperate for functions for leaf nodes also.

→ treat root node differently

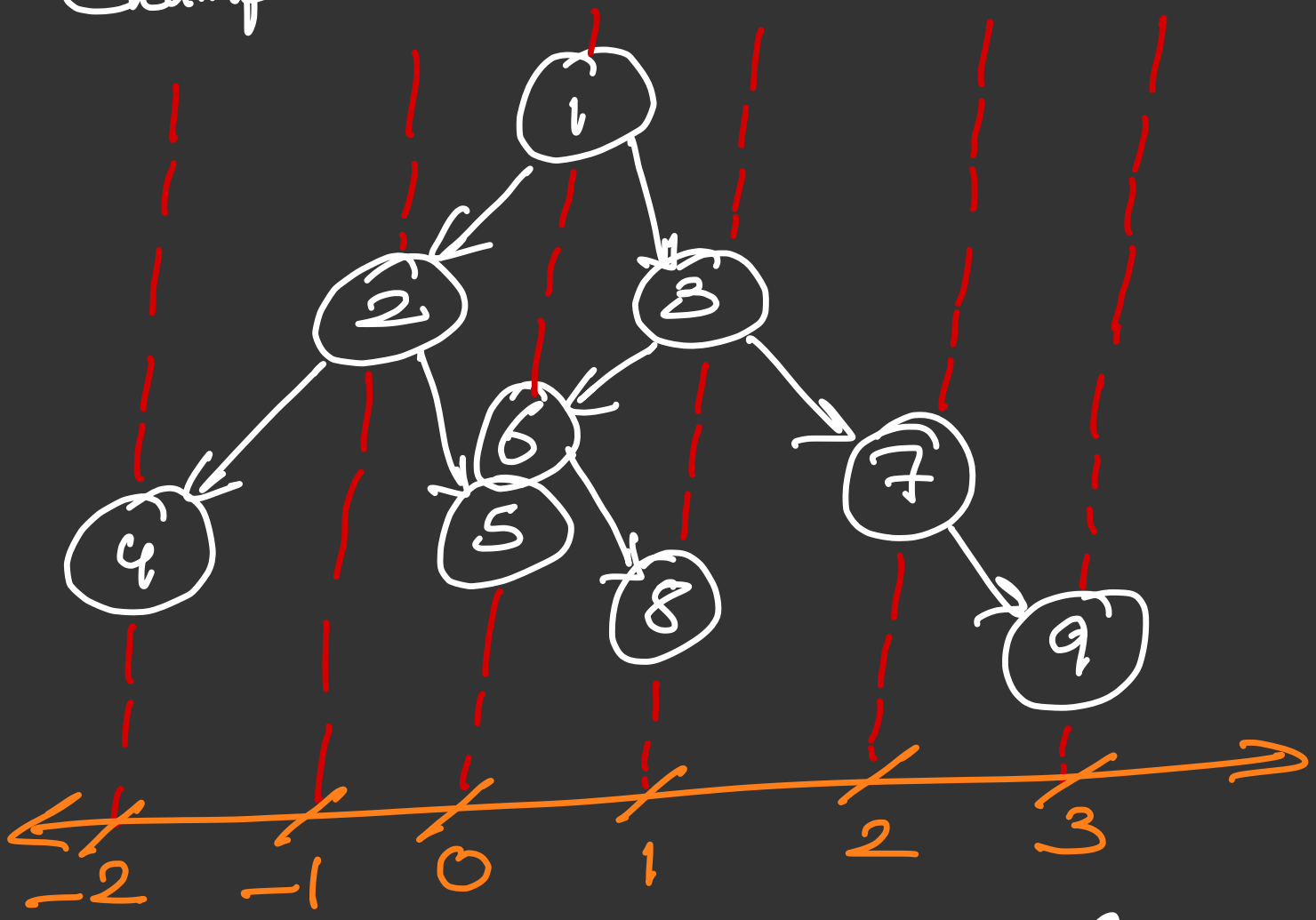
→ left subtree differently

→ right subtree differently

→ border_Traversal.cpp

Vertical Order traversal

example →



o/p → 4 2 1 5 6 3 8 7 9

* in case of 5 and 6, left to right

-2 - 4
-1 - 2
0 - 1, 5, 6
1 - 3, 8
2 - 7
3 - 9

→ A number line with root node at 0.

→ For the mapping, we need two things.

1. Position in number line
2. At what order the node is there (level)

Mapping →

horizontal distance → level wise nodes

↓

mapping

↳ level → nodes

```
map<int, map<int, vector<int>>>
    nodes;
```

☆ Keys of maps are arranged in ascending order.

for level order traversal

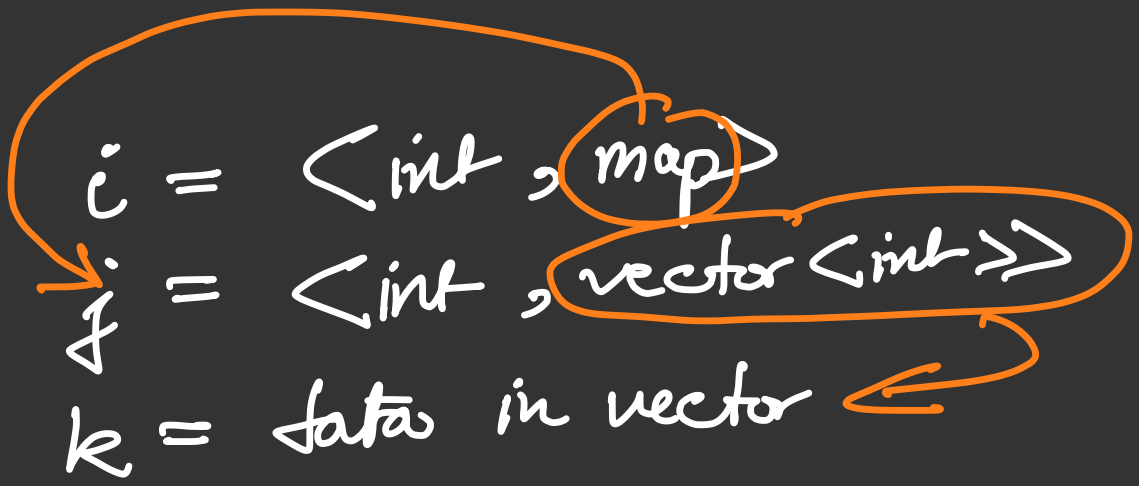
↳ queue<pair<Node*, pair<int, int>>>

level ↑

position ↓

☆ Level / Order wise

↳ so have to use queue.

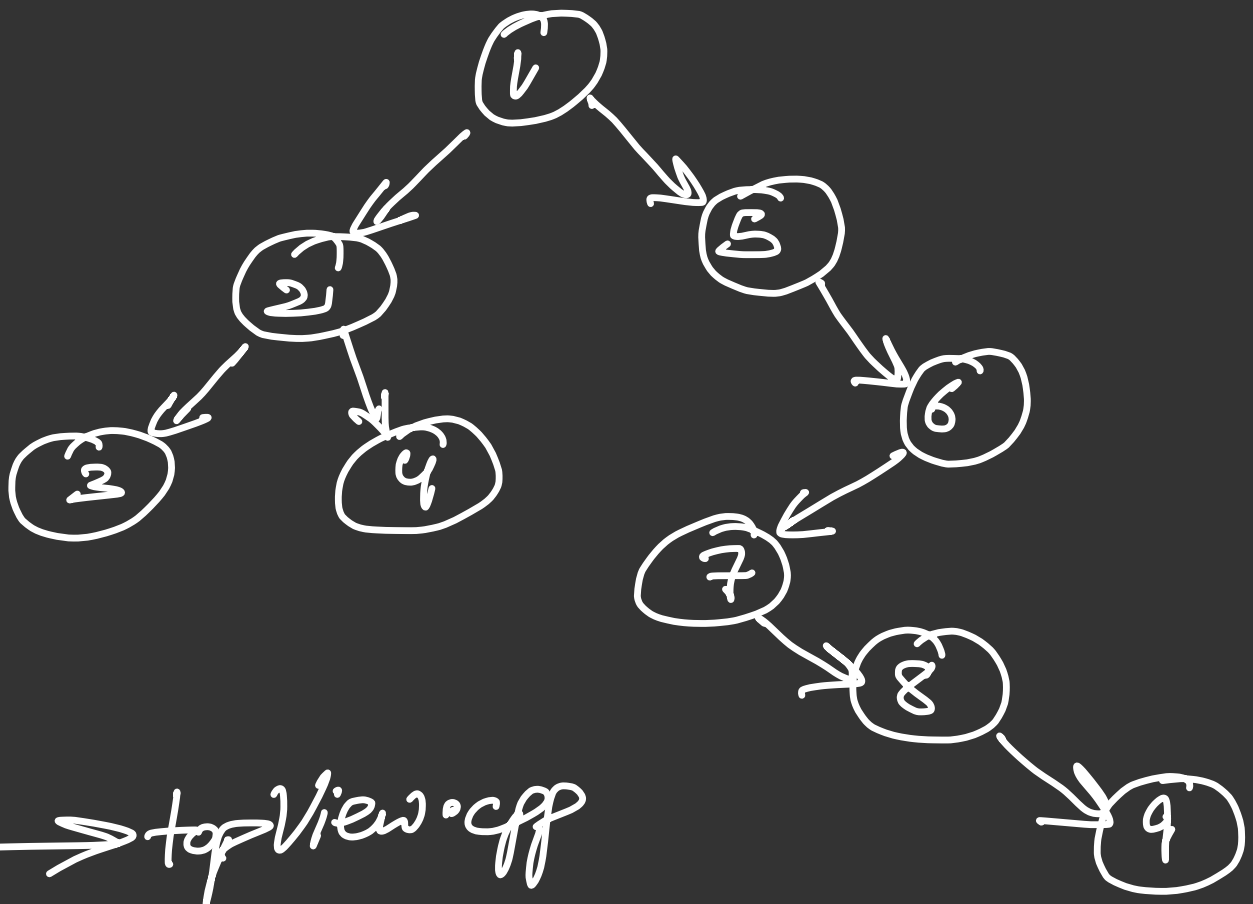


☆ We are going in level order to every node and storing it in a map. (hd, level)

⟶ vertical_Traversal.cpp

Top View of Binary

exception case →



→ topView.cpp

o/p → 3, 2, 1, 5, 6, 9 → this is also included

→ thus logic of last question should be used

↳ Vertical Order traversal

☆ No NEED of level in this, just horizontal distance in the number line.

☆ for checking if map already a value for a key

```
if(mapl.find(a) == mapl.end())
```

Bottom View

⟶ just remove the above condition and let the value change for key of map everytime

⟶ bottomView.cpp

IMPORTANT

☆ In front and bottom view, only position or the number line was important.

But, in left and right view, only level is important

In Vertical traversal both were important.

☆ Using map effects TC

→ Need to read ON this.

Left View

Better Approach ⟶

Using Recursion (as approach 2)

```
function (root, level, vector<int>& ans)
{
    if(root == NULL)
        return;

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

    if(root->left != NULL)
        level+1

    (root->right)

}
```

level wise but RECURSION

⟶ leftView.cpp

Right View

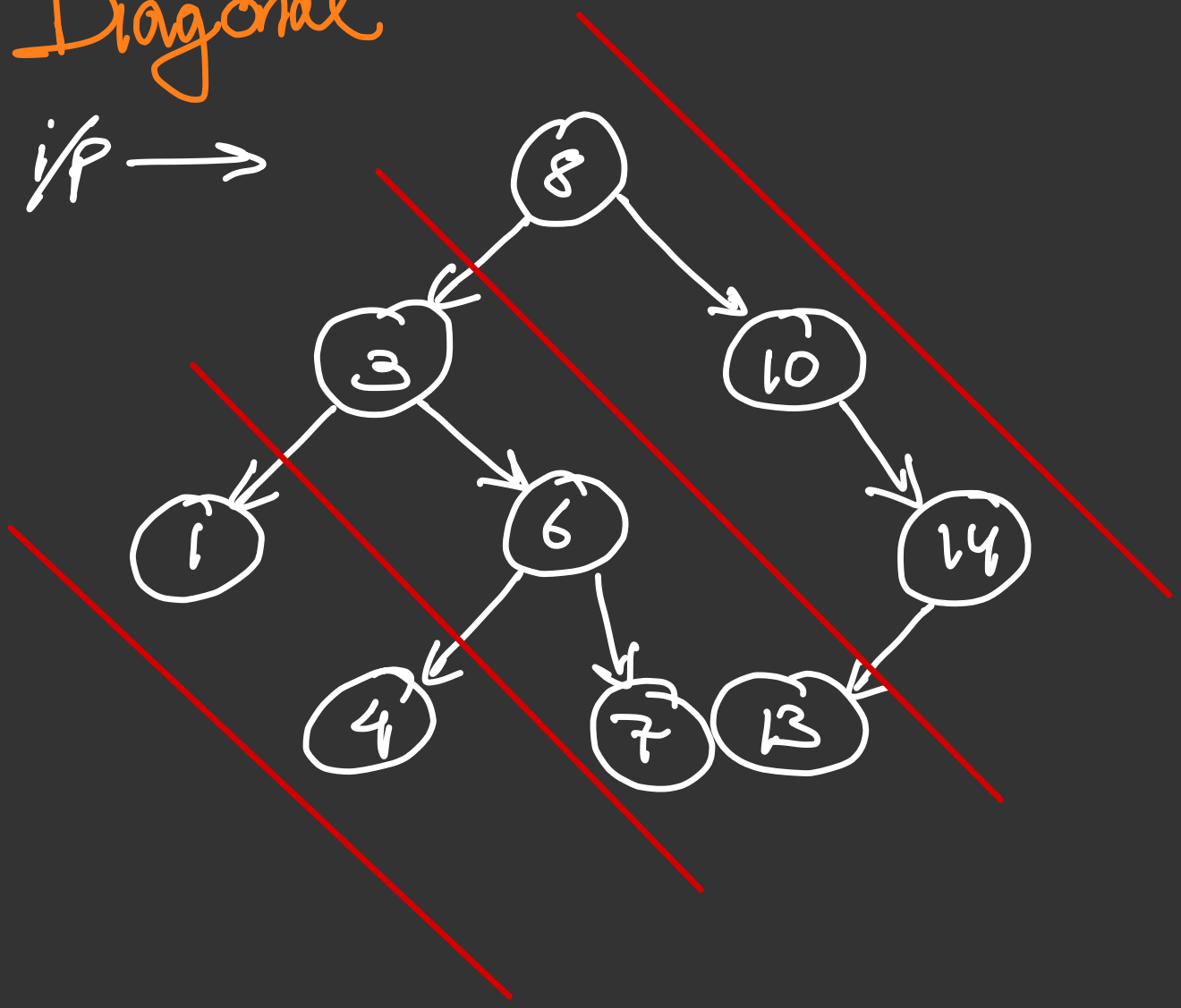
→ put root → right first
rest all same.

```
function (root, level, vector<int>& ans)
{
    if(root == NULL)
        return;

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

    (root->right) ——————→
                  level+1

    if(root->left != NULL)
    ←——————
                  level+1
}
```

→ rightView.cpp

Diagonal

i/p →



o/p → 8 10 14 3 6 7 13 1 4

→ diagonalTraversal.cpp

☆ Didn't have to be level wise, so used Recursion

map<int, vector<int>>

↳ positions.

Whenever going left

↳ hd − 1

going right

↳ same hd

And keep on adding values in the vector.

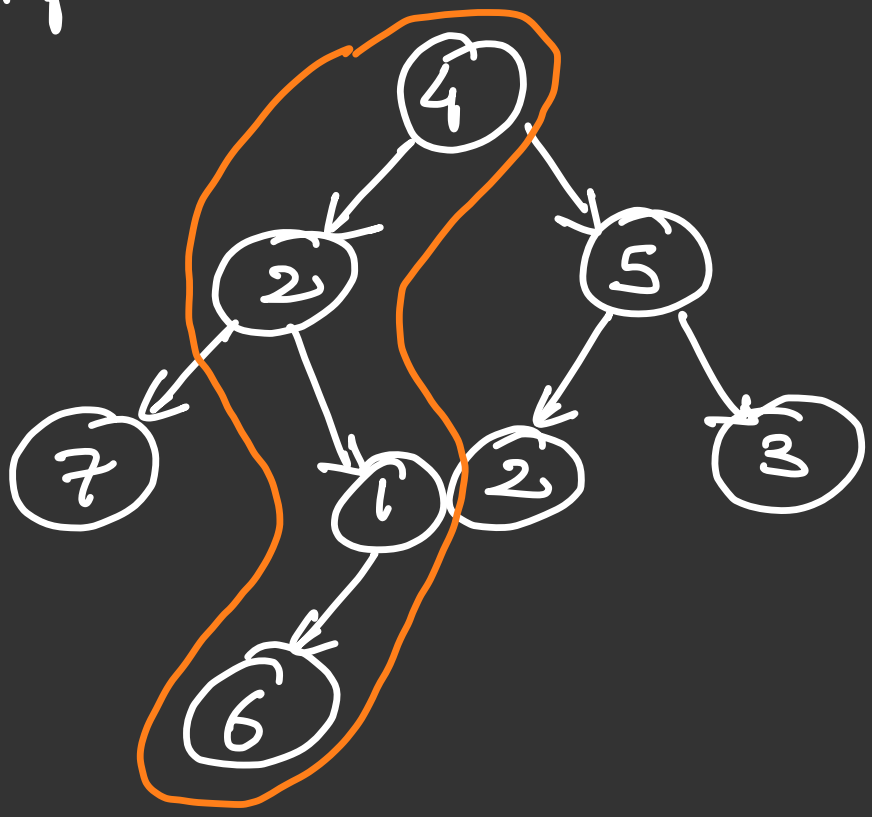
At last ⟶

```
for(auto it=node.rbegin(); it!=node.rend(); it++)
{
    for(auto j: it->second)
    {
        ans.push_back(j);
    }
}
```

Lecture 65

Sum of nodes in Longest Path

example:



We need

- Sum
- Length

} at each node from root node.

Think logically what you need, no queue (or) map is required.

Just two variables
↳ sum
↳ length } pair

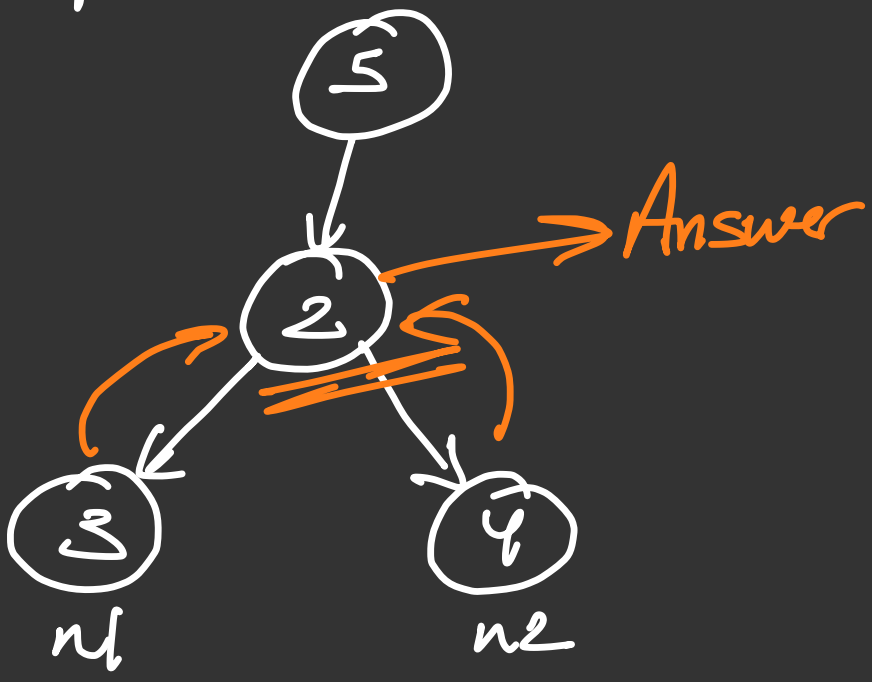
And keep on checking the conditions for them and update their values using Recursion.

⟶ sumOfNodes_longestPath.cpp

TC = O(height)

Lowest Common Ancestor in a Binary Tree

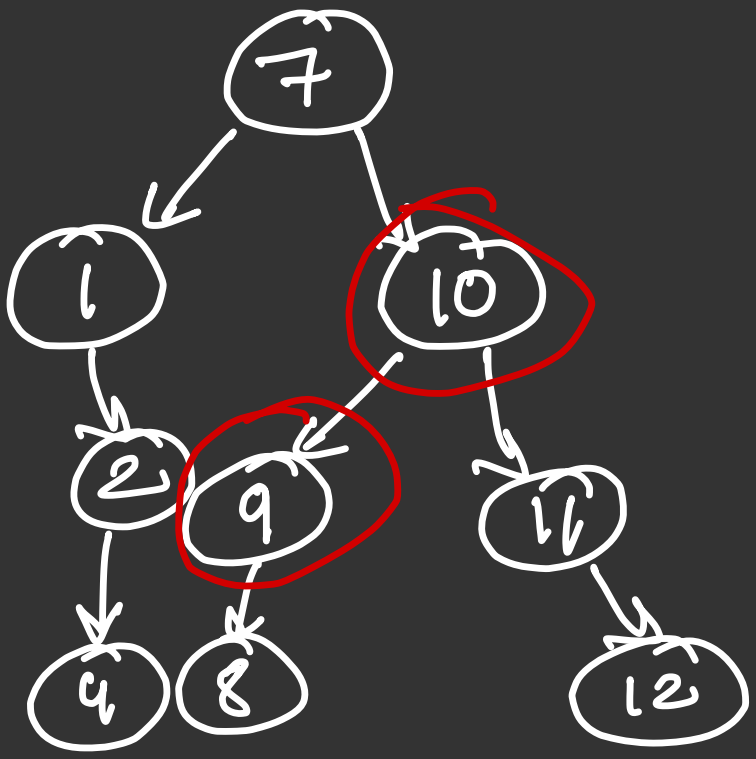
Example:



DID IT !!!

→ lowestCommonAncestor.cpp

* Even in if conditions condition of 2 before 1 should be there.



→ In this case, we store 10 in ans and return 1.

Two conditions

↳ if they both are in same branch, the first node of the element we find is the answer

→ If they both are in different branch this ans will change eventually whenever b1==1 && b2==1 comes for the first time.

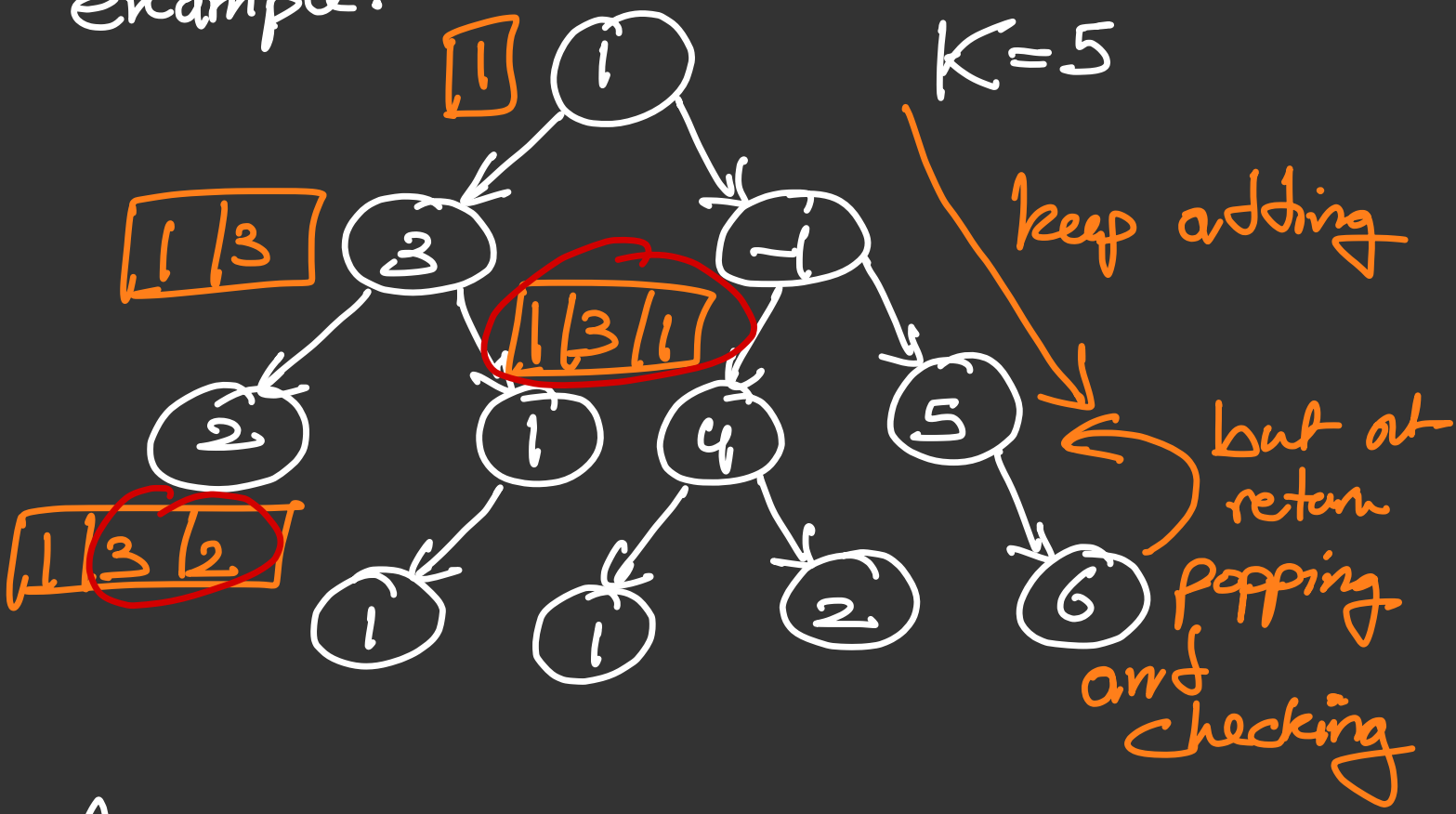
(b1==1) || (b2==1)

↳ if any one of the element is there in one of the branches but not in others, so just return 1.

TC = O(height)

K-Sum Paths

example:



Approach →

keep adding elements of the node to the vector. And when returning check if sum from the back == K.

✗ Showed TLE, maybe need to use MAPS.

Kth Ancestor in a Tree

Approach 1 ⟶

Store the path in a vector, once the node is found just go back to the kth position in the vector.

Approach 2 ⟶

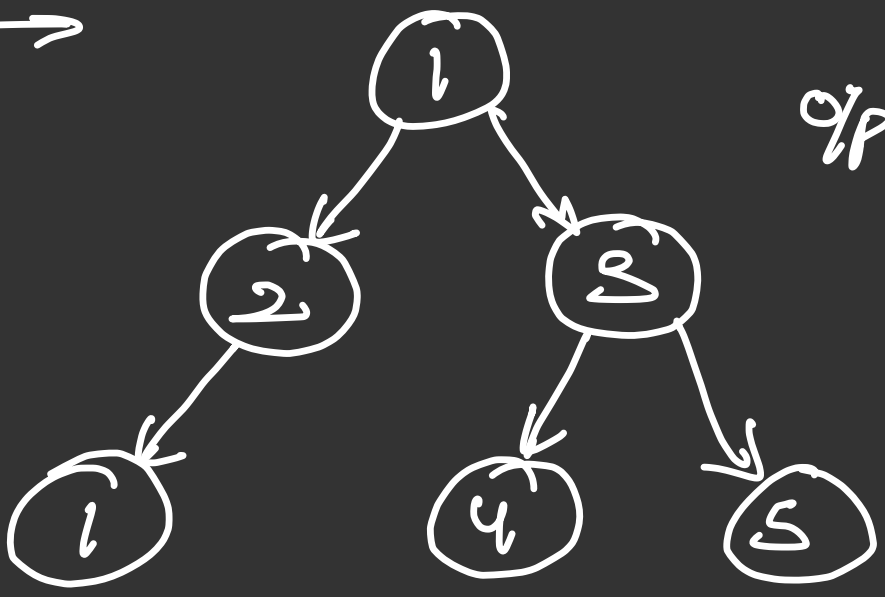
PRACTICE This.

(previous question too)

⟶ Kth_Ancestor.cpp

Maximum sum of Non-Adjacent Nodes

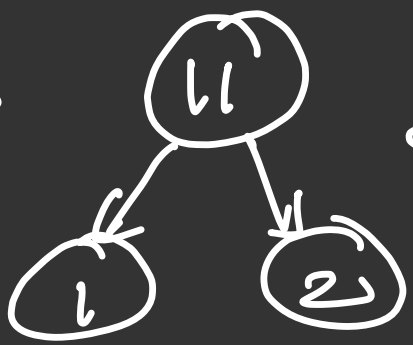
i/p →



o/p → 11

☆ So basically if we have a node, we cannot consider its child (or) parent nodes considered

Also,



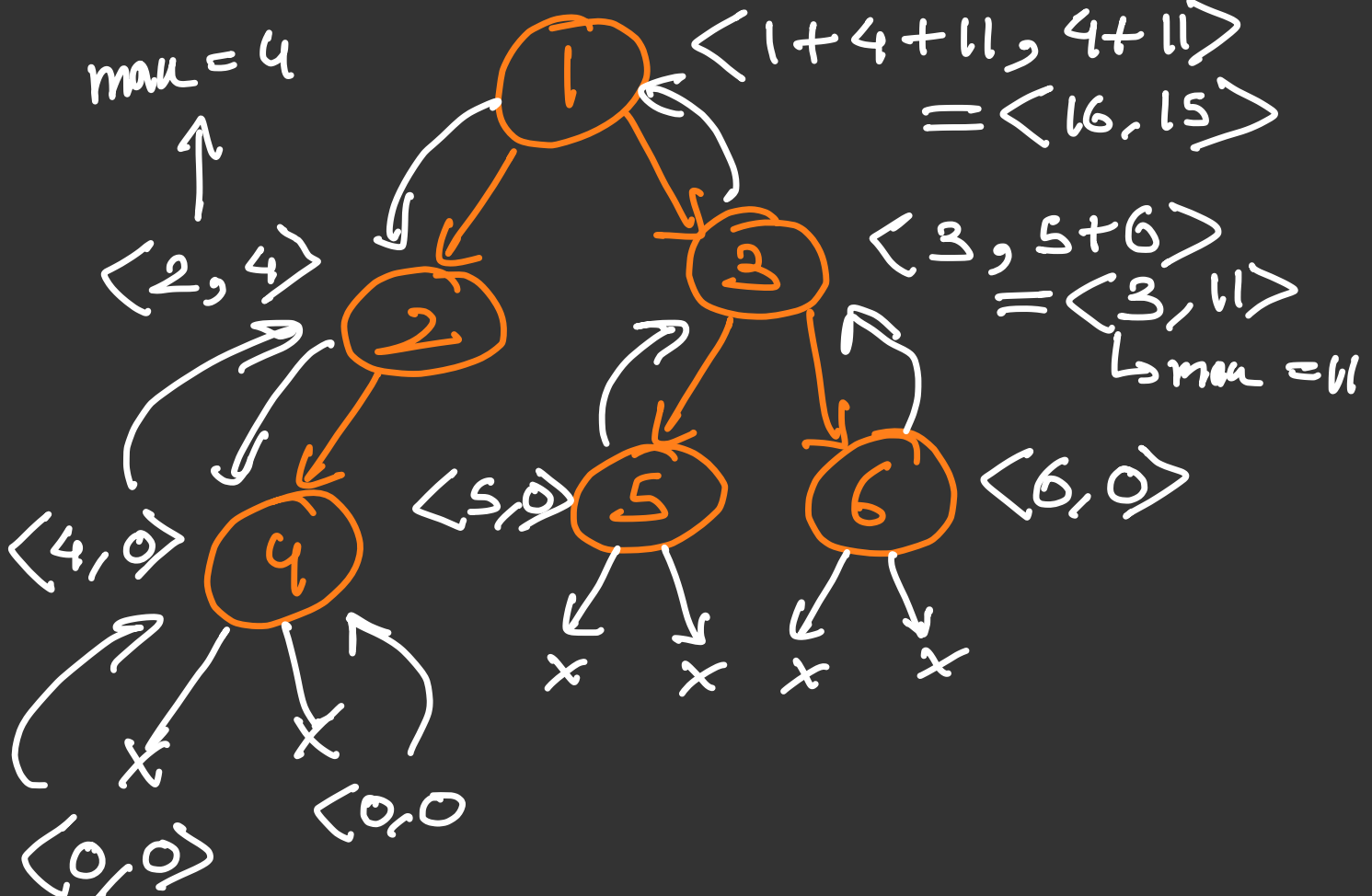
→ in this only root node ans = 11

Approach ⟶

pair < int a , int b >

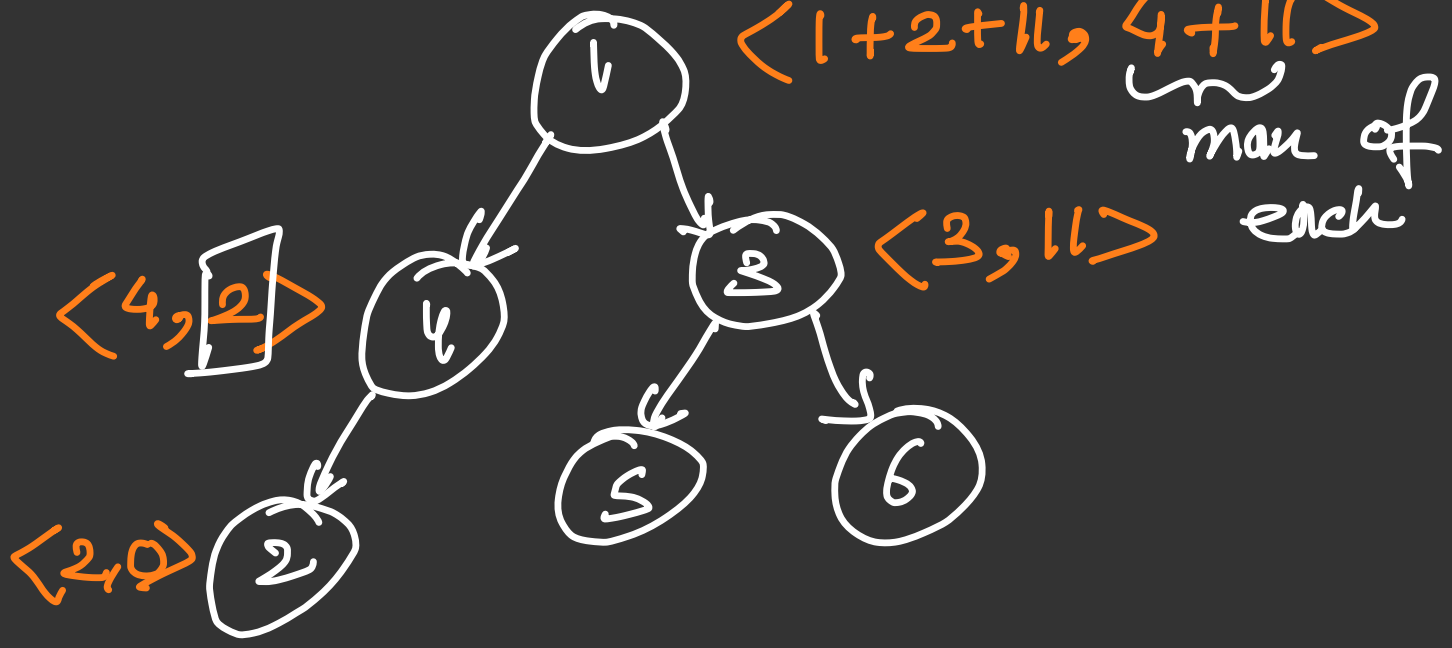
a ⟶ including self and excluding of child nodes

b ⟶ excluding self and max of (a @ b) of child nodes



⟶ maxSum_AdjacentNodes.cpp

The max of both a & b of child nodes is because it says adjacent nodes, so it doesn't has to be level wise nodes only



(4+5+6) > (1+2+5+6)

↳ and these are non adjacent

* The logic of max fits well.

Lecture 66

Construct a Binary Tree when In/Pre Order is given

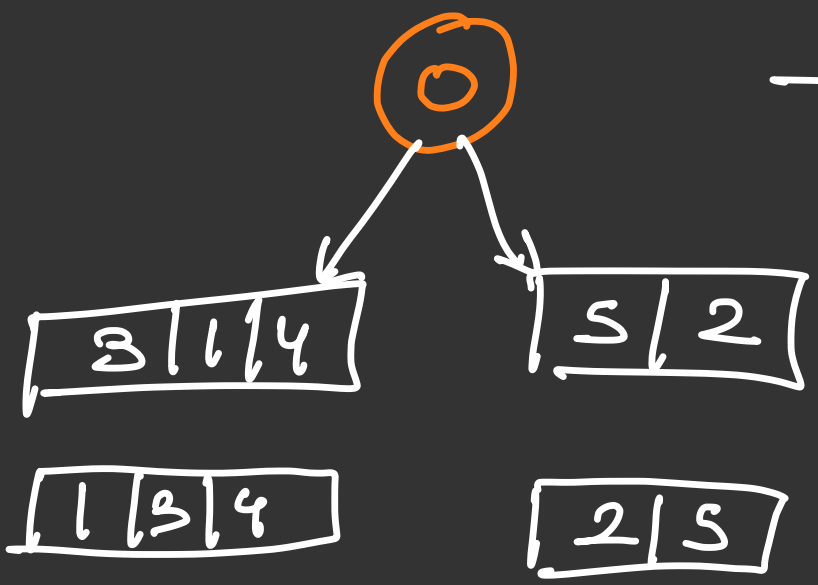
example:

Inorder (LNR)

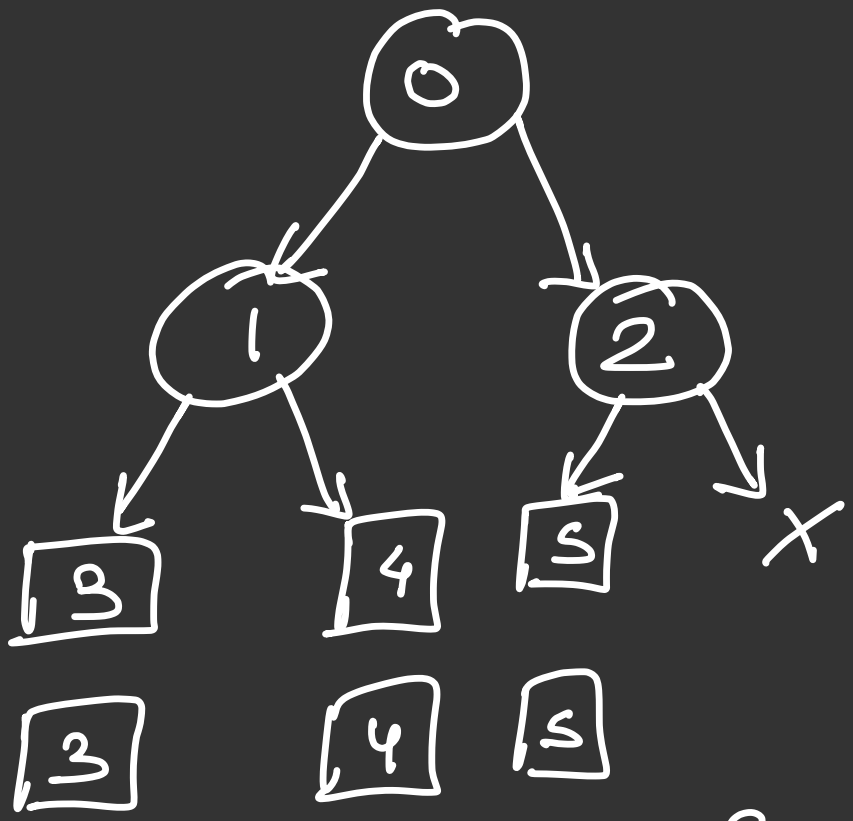
↳ [3 1 4 0 5 2]

PreOrder (NLR)

↳ [0 1 3 4 2 5]



→ give left IN and +1 PRE



→ constructTree_from_InOrderAndPreOrder.cpp

① if (n == 1)
return inorder[0]; (or) pre first element

② Now, for a new node to be added, element will always be the first element of Pre-Order
↳ stored in ele

(3) Now look for this element in InOrder as j.

☆ i ⟶ represents number of elements in both the arrays.

(4) For left,

if in preorder the first element only was ele, thus return NULL in left.

else.

prtl,

in remains first number of elements = j in i

(5) For right
if we found our elmt
in the end of Inordr
the return NULL

else.

* Since in PreOrder left comes first and I mark pr as 4 in parameter, just for all left it kept on increasing and now only right elements of current node are left in preordr.

NLR

→ once more pr+1

→ inordr will be sent right of what element found and i-j-1 as i

Tree from PostOrder and InOrder

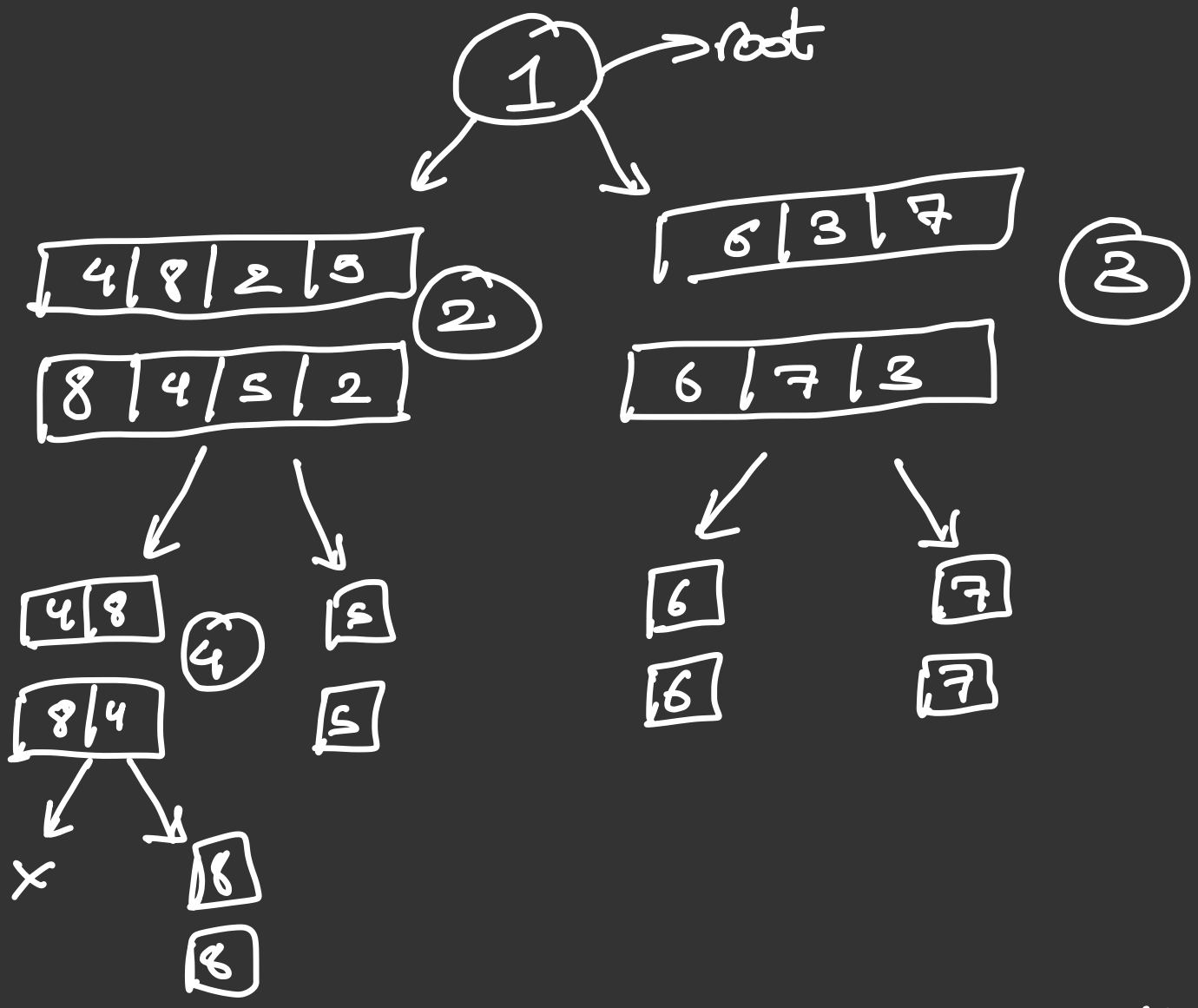
example:

InOrder (LNR)

→ [4 8 2 5 1 6 3 7]

PostOrder [LR N]

→ [8 4 5 2 6 7 3 1]



→ ConstructTree_from_InOrderAndPostOrder.cpp

TC = $O(n^2)$

↳ for both of them.

* But we can crete a mapping of all InOrder elements with their positions and then directly access position from there and creating of mapping will be treated as $O(n)$.

Lecture 67

Burning Tree

→ a target is given and we have to calculate the minimum time.

* We need connection to parent node too, thus mapping of each node with its parent node.

→ burningTree.cpp

1st step
↳ mapping
<Node*, Node*>

2nd step
↳ find the target node

3rd step

↳ Start burning

⟶ Need two data structures

① To track if the node is visited or not.

map <Node *, bool>

② Queue

initially it has target node

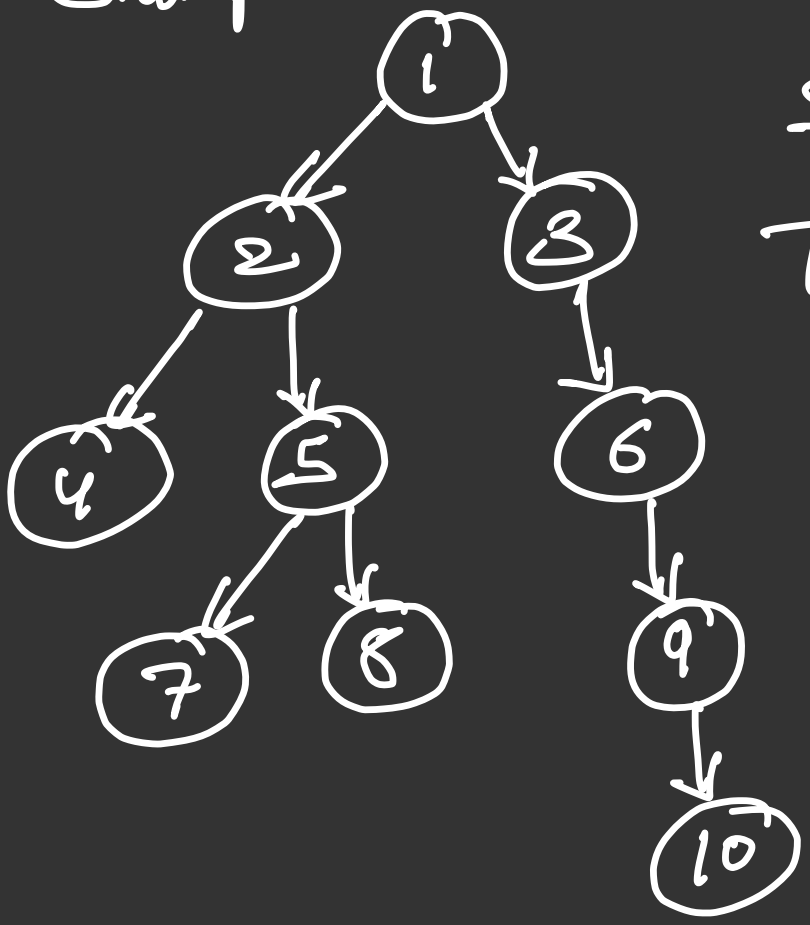
⟶ time = 0

* time is increased only when something is added in queue.

* Also, Order wise traversal is used everywhere rather than recursion

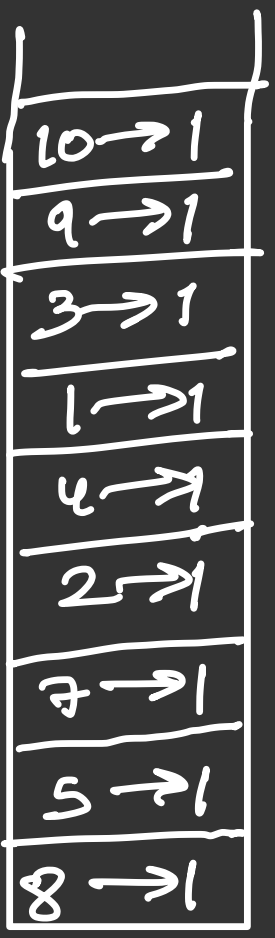
↳ where we going not in a defined manner

Example:



SC = O(N)

TC = O(N) + O(N)
= O(N)



visited

queue

☆ if there was an addition in queue time++;

else if visited[i] → 1 ignore.

Lecture 68

Morris Traversal

✡ All other traversals

↳ TC = O(n)
↳ SC = O(n)

In Morris Traversal

↳ TC = O(n)
↳ SC = O(1)

Algorithm ⟶

1. current = root
2. While (root != NULL)

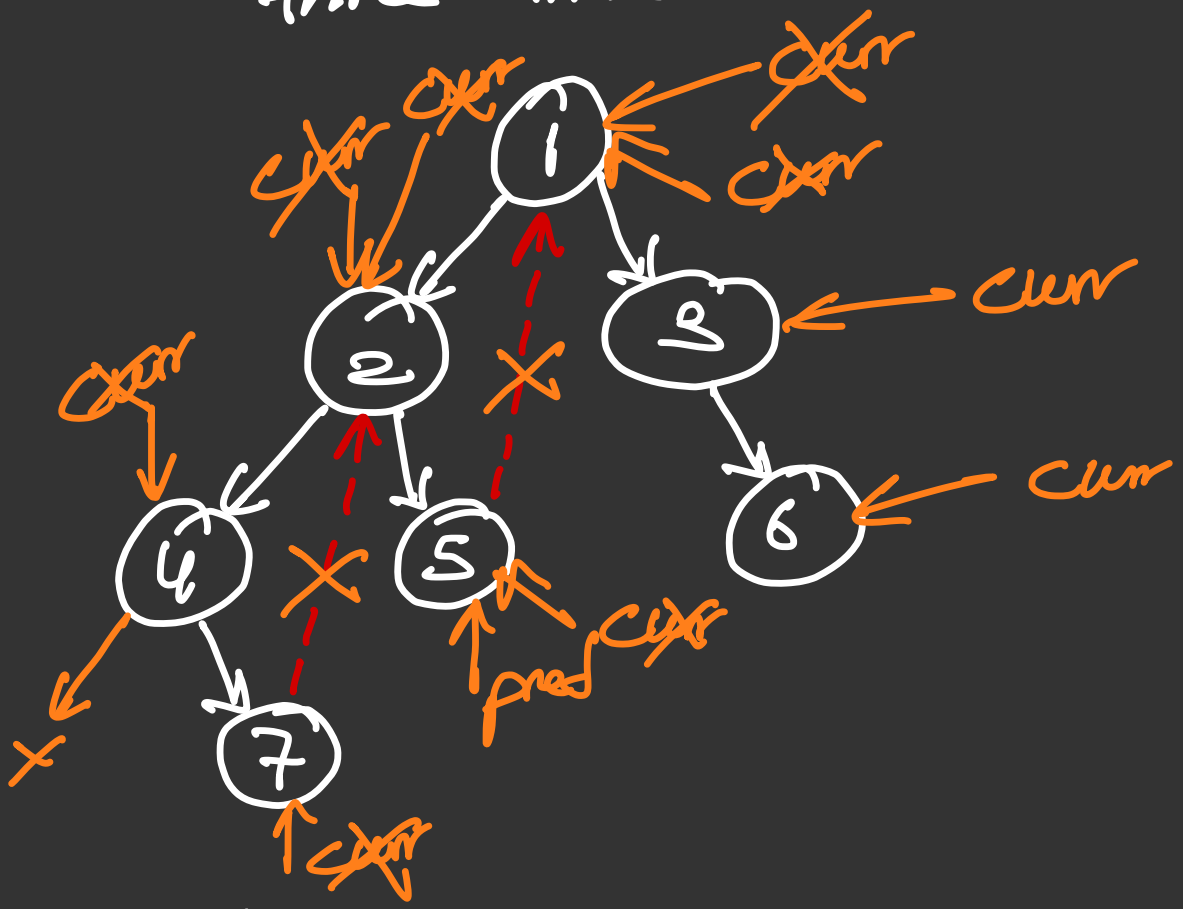
```
{
    if (root -> left == NULL)
    {
        visit (current);
        current = current -> right;
    }
}
```

```
else
{
    predecessor = find(error)
    if (pred->right == NULL)
    {
        pred->right = current
        current = current->left
    }
    else
    {
        pred->right = NULL.
        visit (current)
        current = current->right
    }
}
```

Precessor →

```
pred = curr->left
while (pred->right != NULL
    && pred->right != curr)
{
    pred = pred->right
}
```

☆ Every node is visited almost three times.



InOrder → 4 7 2 5 1 3 6

4 7 2 5 1 3 6

While (cur != NULL)

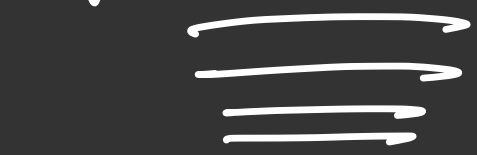
if (current → left == NULL)

else {

PRED = curr → left

while (pred → right)

if(PRED → right == NULL)



else

}

⟶ morrisTraversal.cpp

↳ GFG Version

While I was practicing

↳ morrispractice.cpp

☆ for current keep in check of cur → left

for pre keep a check of pre → right.

Flatten a tree into Linked List

→ for each node left should be null

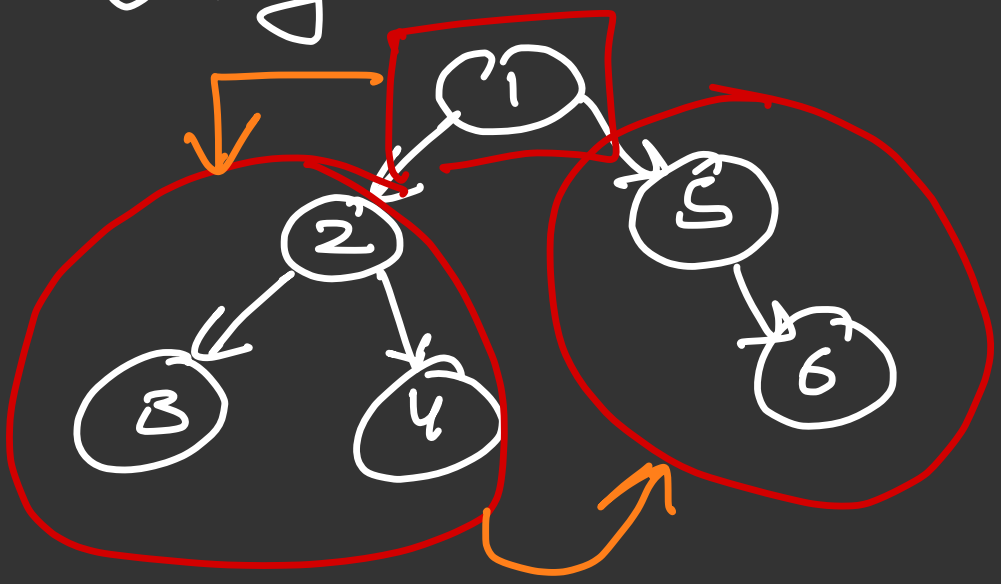
→ and right should point to the next node in PreOrder

✡ In question given SC needs to be O(1)

and for all traversals other than Morris, SC = O(N)

Approach 1 →

Using Recursion

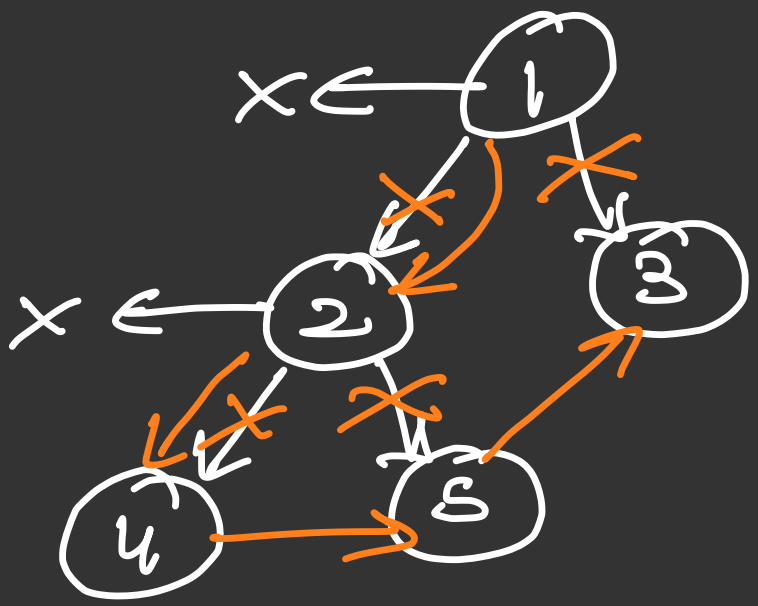


but in recursion
SC = O(n)

Approach 2 →
Using Morris Traversal

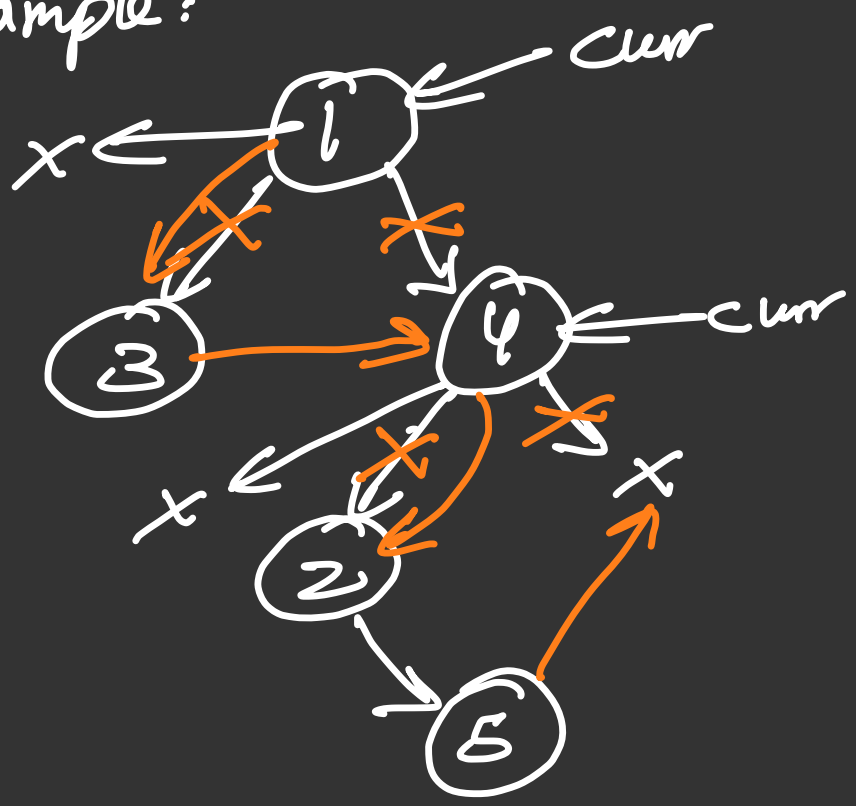
☆ We use Inorder Predecessor only but we point it to curr → right

& then
curr → right = curr → left
curr → left = NULL



if (curr → left == NULL)
↳ curr = curr → right.

example:



Solved this with recursion as well as Morris Traversal

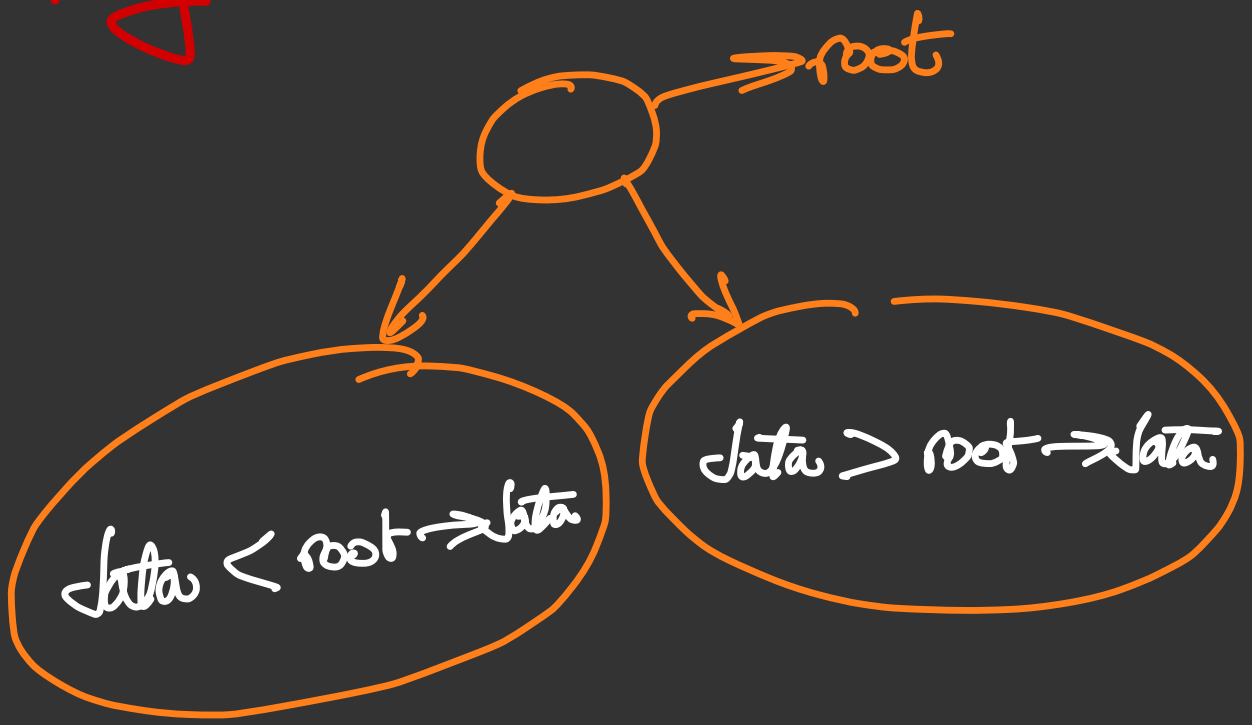
→ flattenTree_usingMorrisTraversal.cpp

→ flattenTree_usingRecursion.cpp

TC = O(n)
SC = O(1)

Lecture 69

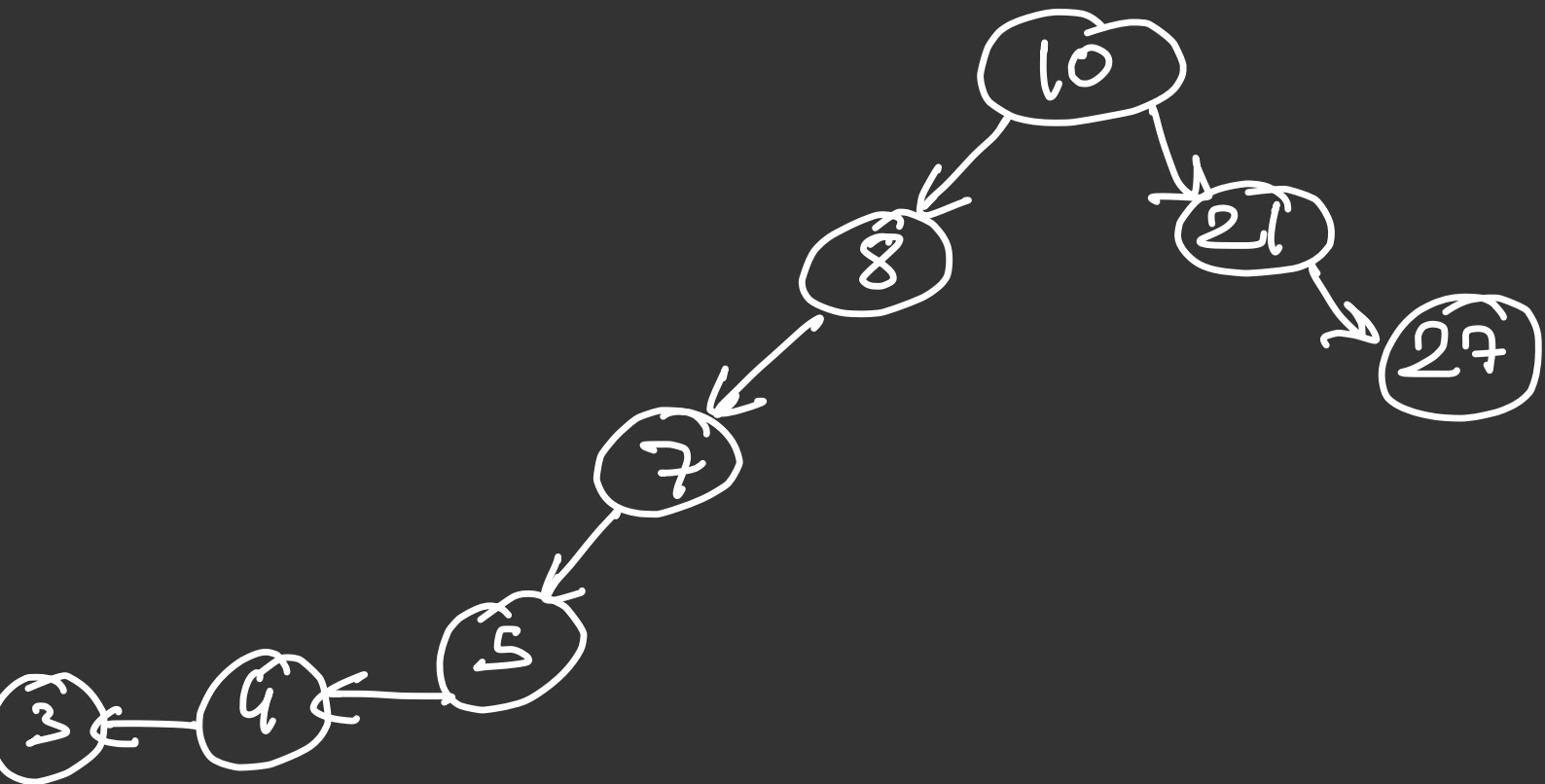
Binary Search Tree



→ for all subtree

Input Stream

→ 10 8 21 7 27 5 4 3



Algo: → < n >

root → data

n < root → data

n > root → data

✡ Code →

```
if (root == NULL)
  {  root = new Node(d);
     return root;
  }

if (d < root -> data)
   root -> left = solve(root -> left, d);
else
   root -> right = solve(root -> right, d);

return root;
```

→ insertIntoBST.cpp

* Insertion Complexity
  → O(log n)

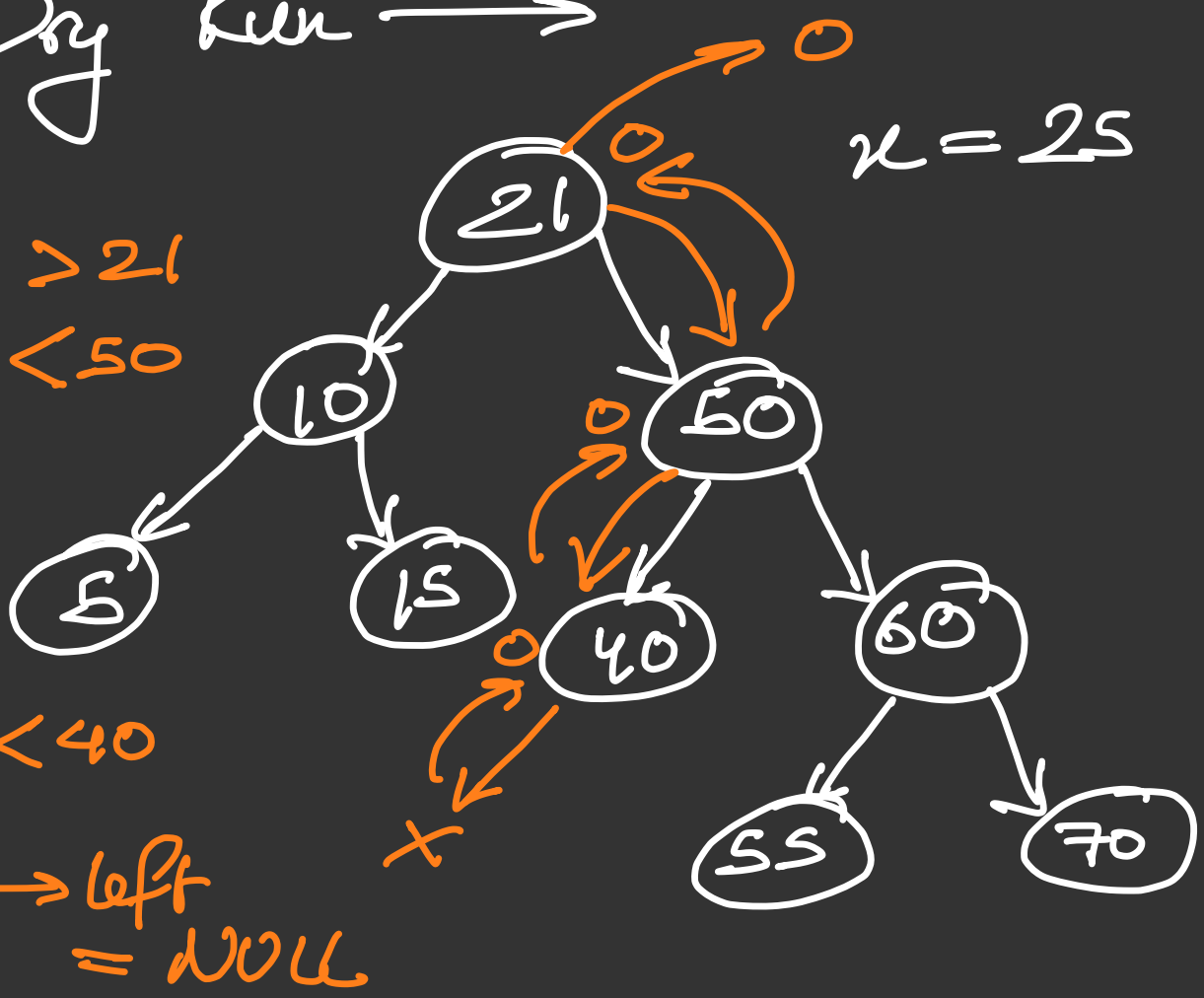
Search In BST

* Please don't Overcomplicate

```
if(root == NULL)
    return false;

if(root -> data == x)
    return true;
else if (x > root -> data)
    return solve(root -> right, x);
else
    return solve(root -> left, x);
```

→ searchInBST.cpp

Dry Run →



★ InOrder of BST is
↳ Sorted

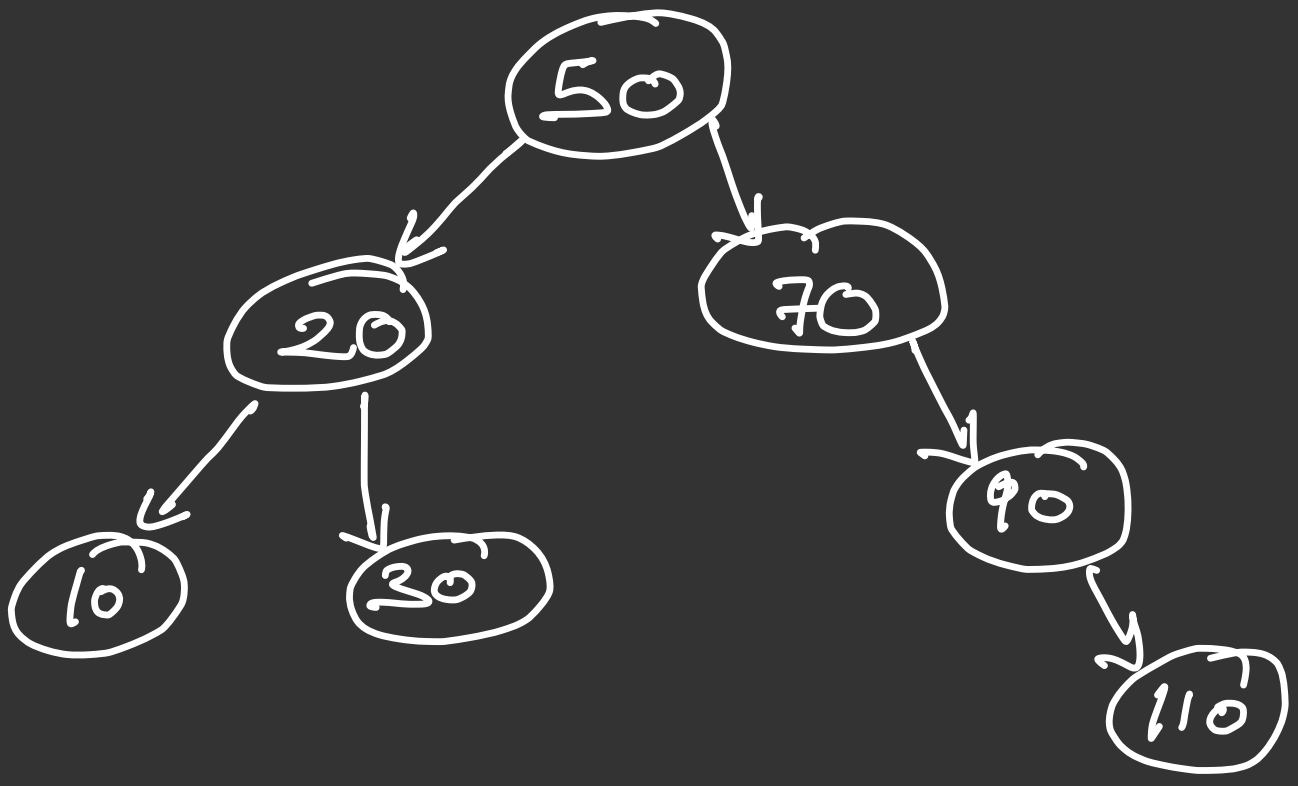
Minimum and Maximum element in BST

minimum ⟶ extreme left

maximum ⟶ extreme right

⟶ min_max_ele_BTS.cpp

Inorder Predecessor / Successor ?



* Inorder is always in SORTED ORDER

For Successor

Approach 1 →

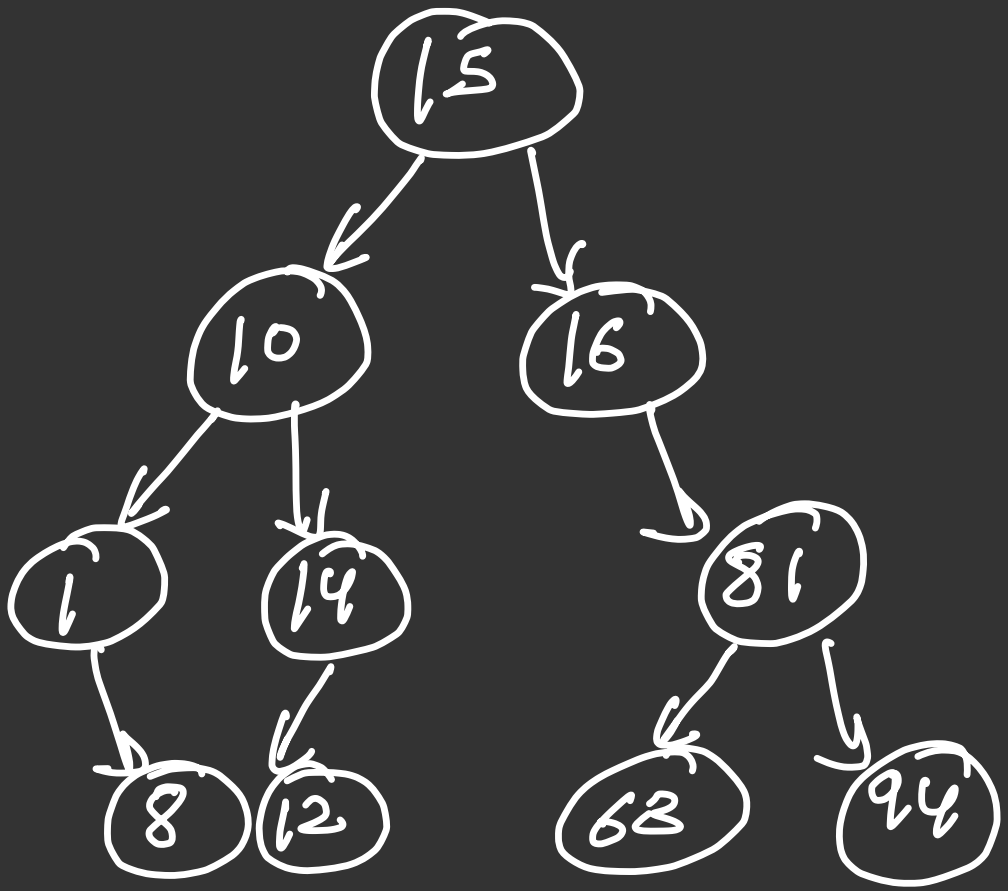
We do a InOrder Traversal, and as soon as we get a node greater than target node, that is our answer.

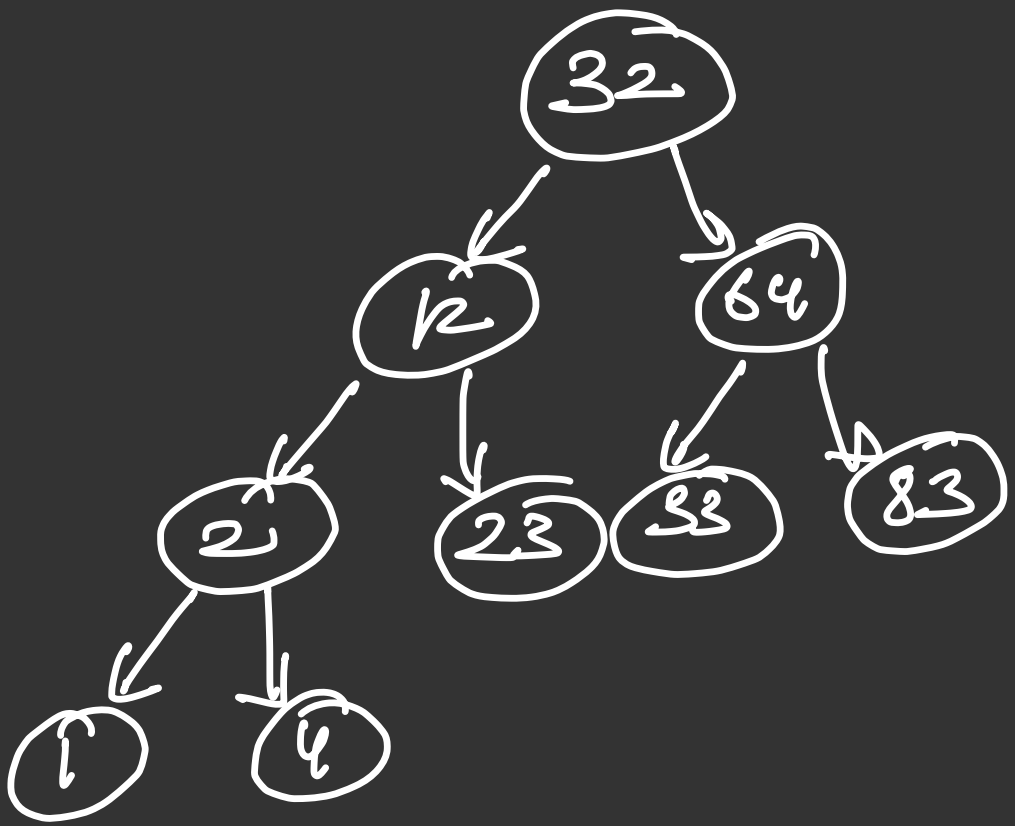
TC = O(n)
SC = O(1) excluding recursion stack

Approach 2 →

→ We need a value greater than target

→ inOrder_Sucessor.cpp


15
10
16
1
14
81
8
12
68
94


32
12
64
2
23
33
83
1
4

(1) First check

if (root → data <= x → data)

→ while less we go to root = root → right

→ Now two options it might be greater than or equal to x → data.

(i) If equal just go to the root → right

→ if root → right == NULL return NULL

→ else after going to root → right, continue to root → left until equals to NULL

(ii) root → data can be greater now also or in the starting itself so, we recursive call the function for (root → left) but if returns NULL then the root of root → left is the ans as it was greater than x → data.

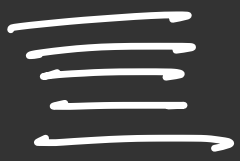
✡ Whole logic based on

⟶ In InOrder of BTS it is in sorted order

⟶ left subtree of root of BTS contains smaller elements and right one has larger elements.

Deletion of node in BST

Algo →

1. Reach node
2. if (root → data == x)
   {
   
   }
3. else if (root → data > x)
   { left part }
4. else if (root → data < x)
   { right part }

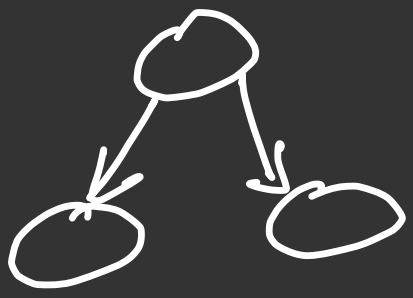
☆ Need more practice on this

Node that has to be deleted

→ has zero childs

→ has single child

→ has two child



(i) from left bring the max element and recursively delete that node as well

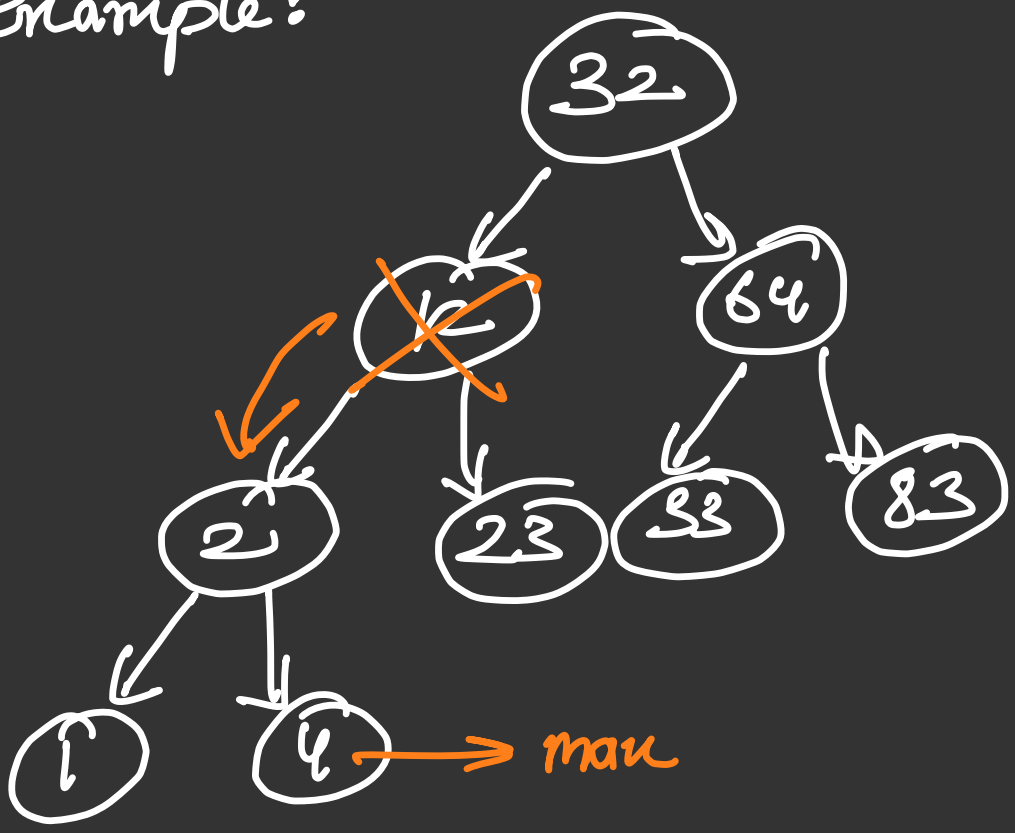
(or)

from right bring the smallest value and delete that.

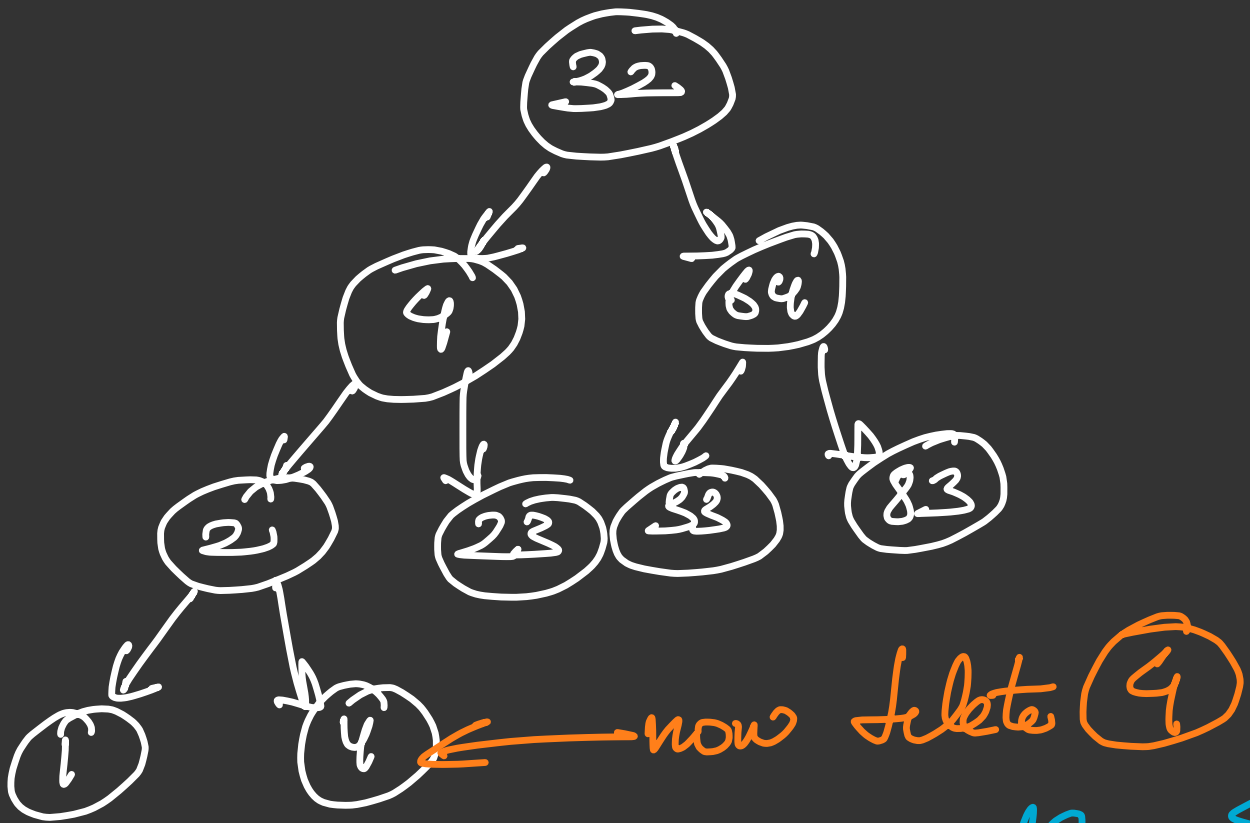
→ root is only NULL.

TC = O(height or n)

Example:



If we delete 12



via recursive call ☆ with root as 2 and value 4.

Binary Search tree

→ We work with a particular value

Thus, traversal happens

if (root → data < val)
    recursive call (root → right)

else if (root → data > val)
    recursive call (root → left)

Lecture 70

Validate BST

Properties

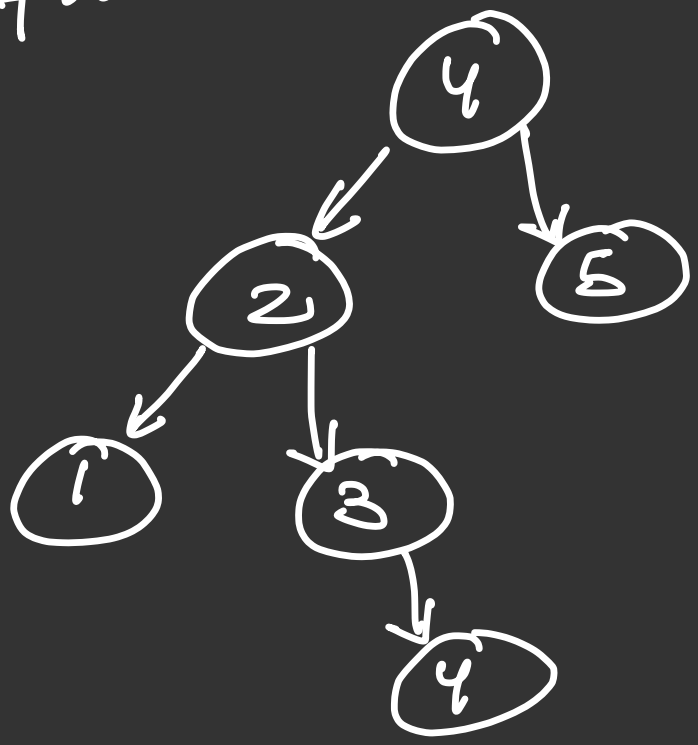
- for every node, left subtree has elements less that the node element, greater for right subtree
- InOrder Traversal is always in sorted order

Approach 1 →

Store the Inorder traversal and check if it is in sorted order or not.

Approach 2 →

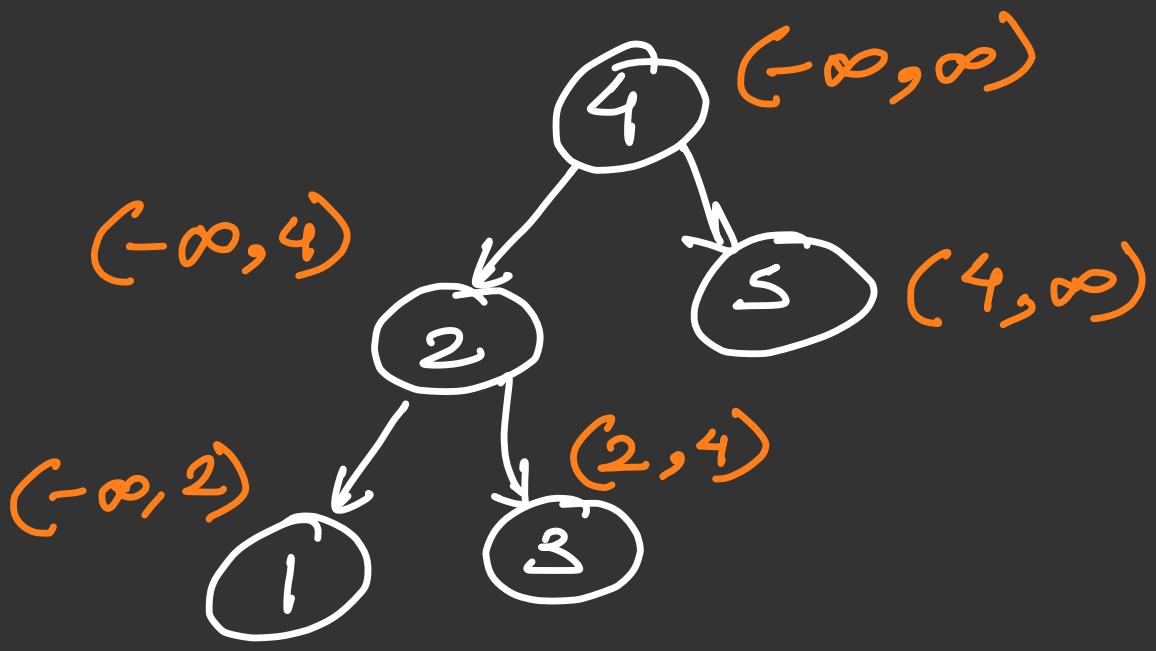
My way had a flaw, if we just keep on checking root → left and root → right values with the node, a flaw is there in this



* According to my thinking this is a BST but it is wrong as the left subtree of 4 should only have elements less than 4.

Thus, another way →

→ we will have a min and a max value for each node.



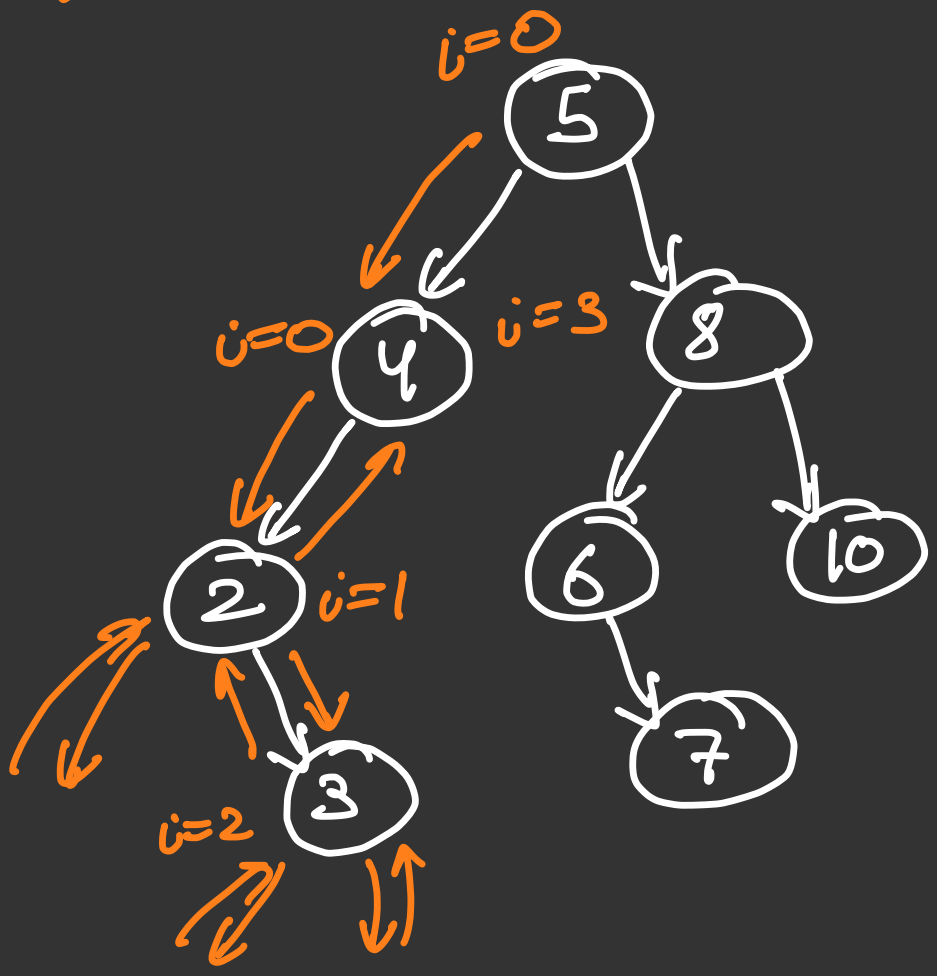
Thus, there is range of every node in BST

→ validateBST.cpp

TC = O(n)

SC = O(height)

Kth Smallest element in BST



4 is the answer

```
Node* solve(Node* root, int k, int& i)
{
    if(root==NULL)
        return NULL;

Left {
    Node* a = solve(root->left, k, i);
    if(a!=NULL)
        return a;
```

Node { i++;
if (i == k)
return root;

right {return solve(root->right, k, i);

↑ there is a possibility that
3 the ans might be in
right subtree also,
else it will keep on
returning NULL.

⟶ When playing with
Inorder traversal as well
as data.

⟶ Kth_Smallest_BST.cpp

TC = O(n)
SC = O(height)

* Do this using Morris Traversal.

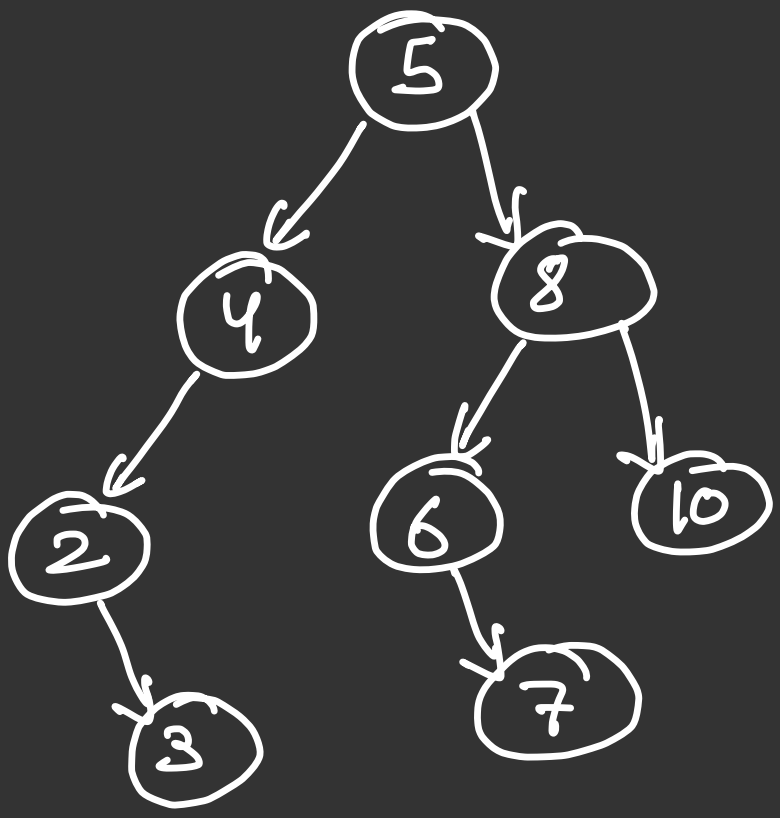
☆ for k^{th} largest
$= (n-k)+1^{th}$ smallest element

Calculating no. of nodes — 'n'
$= O(n)$

Calculating using k^{th} smallest
$= O(n)$

Thus, total $O(n)$

Predecessor and Successor in BST



(1) Find node

But while reaching to the node also we are keep a check of the parent nodes based on they are larger or smaller.

ex: 2 is our node

but 2 → left is empty.

thus parent of two is going to be the successor.

For Predecessor ⟶

After finding the node, we look for node→left, and if node→left == NULL then parent of node is the ans.

And if node→left is not NULL, then while

```
temp = node->left;
while (temp != NULL)
{
    ans = temp->data;
    temp = temp->right
}
```

Similarly for Successor

⟶ predecessor_successor_BST.cpp

TC = O(n)

LCA in a BST

↳ Lowest Common Ancestor

(i) root < a && root < b

↳ go right

(ii) root > a && root > b

↳ go left

(iii) root < a && root > b

(a)

root > a && root < b

↳ root is the ans.

Property Used ⟶ left of every node contains elements less than root data and greater than for right.

⟶ LCA_in_BST.cpp

Lecture 71

Two Sum in BST

Approach ⟶

InOrder of BST is in sorted order, so store the inOrder.

Now, using two pointer approach we check if the sum = target,

if sum > target ⟶ j--;

else sum < target ⟶ i++;

⟶ twoSum_BST.cpp

☆ Also could have done using mapping.

Flatten BST to a Sorted List

→ Store all the Nodes in a vector in InOrder

→ Now start pointing right to the next node and left to NULL

☆ Recursion is not always the solution, even while using vector and storing elements we can reach to optimal solution.

→ flattenBST_into_LL.cpp

Normal BST to Balanced BST

Balanced BST → height of two subtrees of every node differs no more than 1.

Approach →

1. Store inorder in a vector
2. The start with the middle element as the new root and make subtrees similarly using recursion.

⚠ When passing a vector or array and no direct changes are made to them, do it by reference, helps in TC

→ normalToBalanced_BST.cpp

BST from PreOrder

Approach 1 ⟶

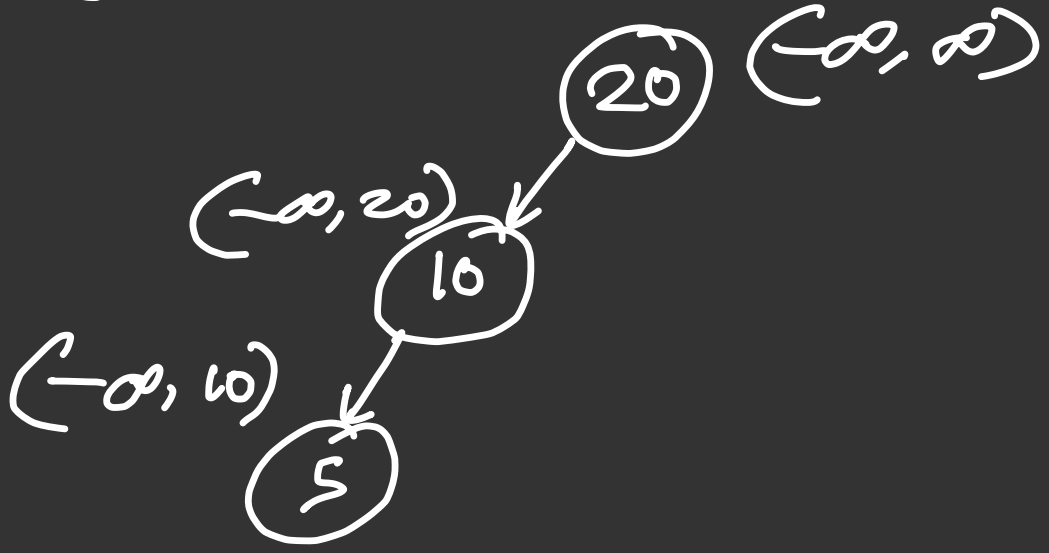
If I sort PreOrder,
I will get InOrder.

Now, using InOrder and PreOrder,
I can construct the
Tree.

$TC = O(N \log N)$

Approach 2 ⟶

{ 20, 10, 5, 15, 13, 35, 30, 42 }



(1) Every node has a range min and max
But if the value in preorder vector at that moment is out of range then return NULL

(2) Else that node value is the preorder value
Do it;

(3) and then recursive call for left and right. with their appropriate range

→ BST_from_PreOrder.cpp.

Lecture 72

Merge 2 BST

Approach 1 ⟶

1. In Order of both BST as arrays
2. Merge two sorted arrays
3. Make BST from this new InOrder.

like done in InOrder to BST.

$TC = O(m+n)$
$SC = O(m+n)$

Approach 2 ⟶

for $SC = O(h_1 + h_2)$

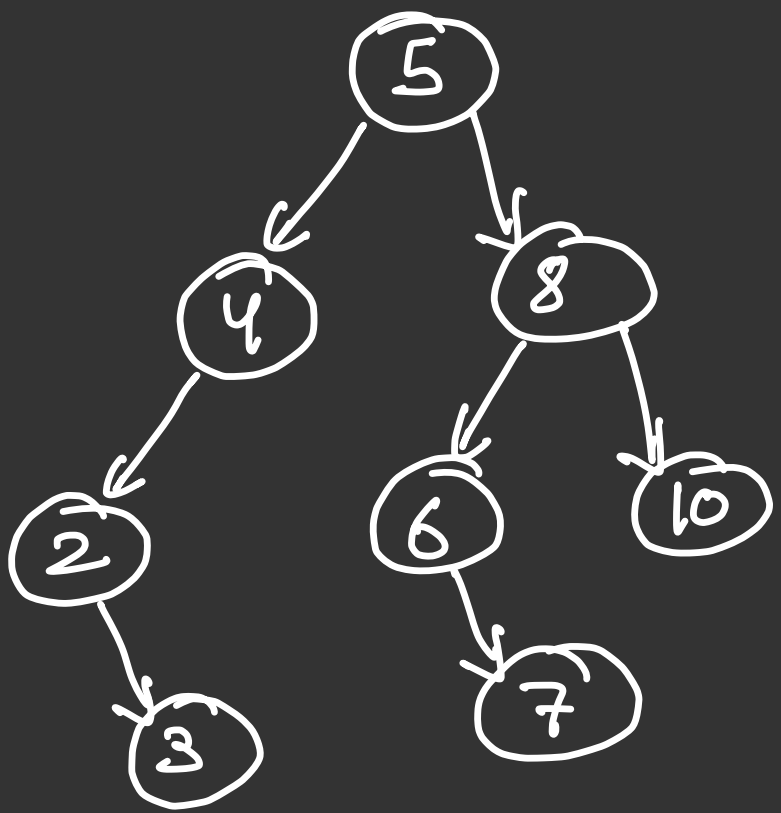
⟶ We convert both the tree into Linked List (in place)

→ And then merge 2 sorted Linked List

Step 1: Convert trees into LL

→ We cannot use vector as SC will be O(n) we want O(h)

→ Converting BST into doubly linked list



☆ We need a HEAD too !!

```
void convert (Node *root, Node *& head)
{
    if (root == NULL)
        return;

    convert (root->right, head);

    root -> right = head;
    if (head != NULL)
        head -> left = root;

    head = root;

    convert (root -> left , head);
}
```

* We need head because the in the left the least element will be first element of the Linked List and not the root.

Step 2: Merge Both the LL in orted order

Step 3: Make BST from the merged sorted LL *

⟶ treat it like inorder

(i) left subtree from first n/2 nodes

(ii) (n/2 + 1)th node is the root node

(iii) all the remaining nodes for right subtree.

$(n - \frac{n}{2} - 1)$

* Write all those into statements and thats the code.

head will be passed by reference while making BST from merged LL.

⟶ merge2BST.cpp

SC = O(height)

* Dry run the last part.

⟶ It was combination of 6 questions.

Lecture 73

Largest BST

Approach 1 ⟶

Go to every node and check if BST is possible, if yes store the number of nodes involved and check with max size stored, if yes update max size.

⟶ Using ValidBst

TC = $O(n^2)$

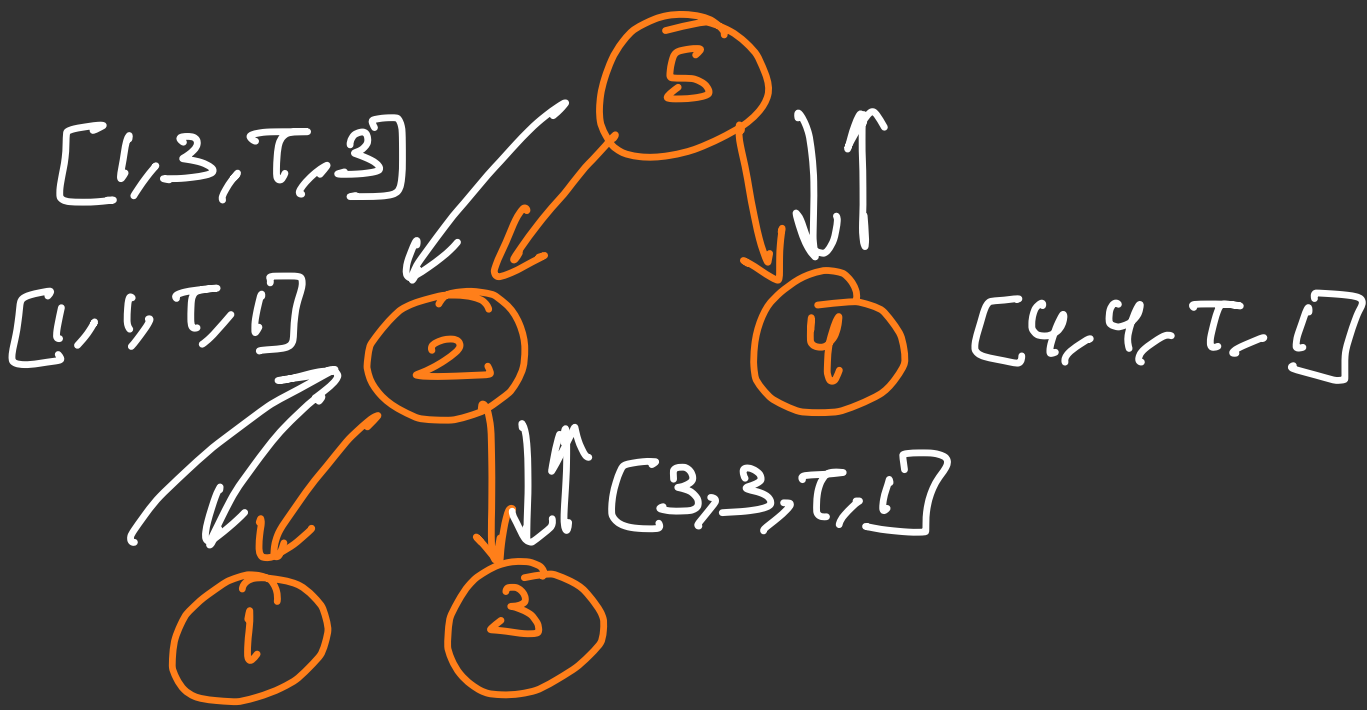
Approach 2 →

We go to each node and check

- left subtree is valid BST
- right subtree
- left subtree max is less root → data, and root → data is less than min of right subtree.

We need { maxi, mini, is valid BST } for each subtree size.

```
class info
{
  public:
    int maxi;
    int mini;
    bool isBST;
    int size;
};
```



$1 < 2 < 3 \rightarrow T$

$mini_2 = mini(1,2) = 1$

$maxi_2 = max(3,2) = 3$

size = left.size + right.size + 1

$3 < 5 < 4$ ✗

✗ for 5 both left and right subtree are T, but this is not a valid BST.

✗ 5 should be greater than left subtree maximum and 5 should be less than right subtree minimum.

——> largestBST_SubTree.cpp.

TC = O(n)