Binary Trees :

(1) Recursion
(2) Object Oriented Programming } prerequisites

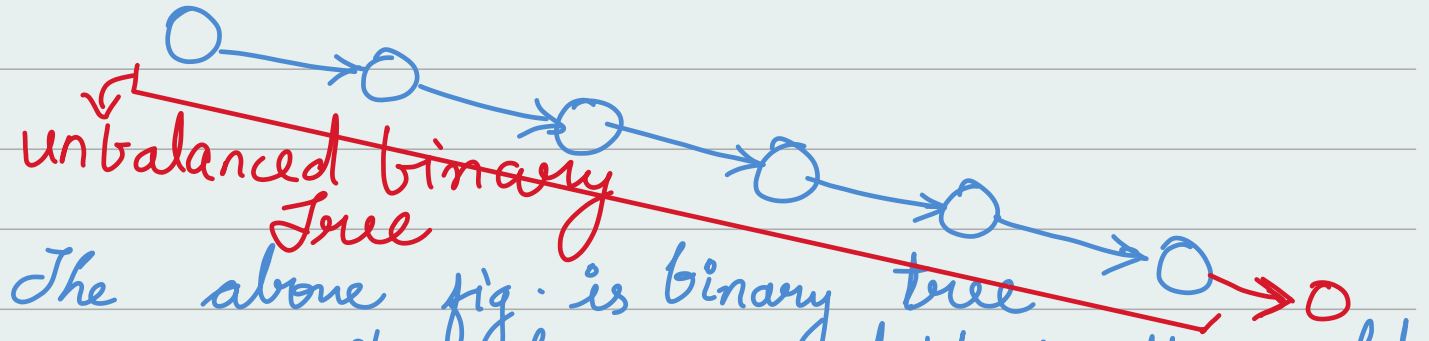
why?

* Binary trees are useful because it can add, find or remove in log(N) time $O(\log N)$

* Ordered storage.

* Cost efficient

* Binary Tree has 0, 1 or 2 children



The above fig. is binary tree cause it has one child to the right side and the L.H.S is null.

→ If a new node is added in the RHS of B.T then it would take O(N) times T.C. So, that is why the fig. is Unbalanced B.T.

→ How to make sure Tree is remained balance all the time?

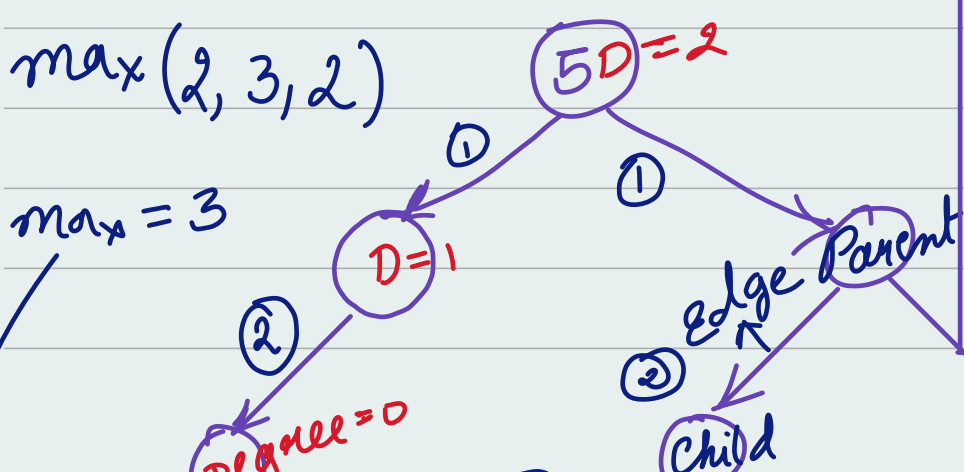
Ans: It can be solved by Self balancing Binary Tree and it will be done in next chapter.

→ Where is it used?

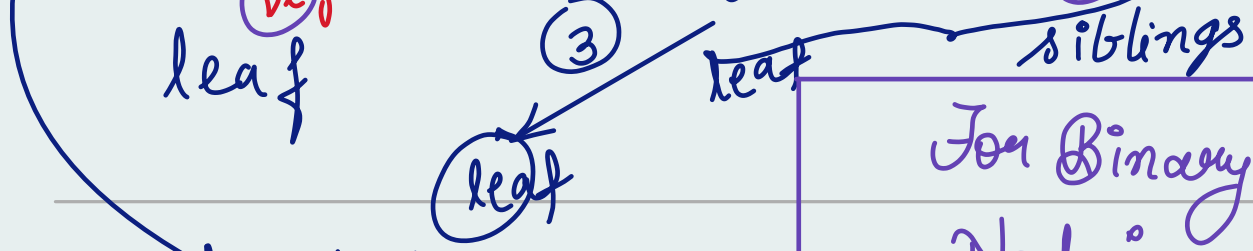
i) File systems
ii) Database
iii) Networking / Algorithms
iv) Mathematics
v) Decision Trees ⟶ Machine Learning
vi) Compression of files
vii) Future data structures

max(2, 3, 2)

max = 3



for linked list
Node:
int value;
Node next;



Height is 3 for the above Binary Tree

For Binary Trees

Node:
int value
Node left;
Node right

* Properties.

① Size = Total number of nodes

② Siblings ③ Edge ④ Child & Parent

⑤ Height:- Maximum number of edges from that node to the leaf node

⑥ leaf node

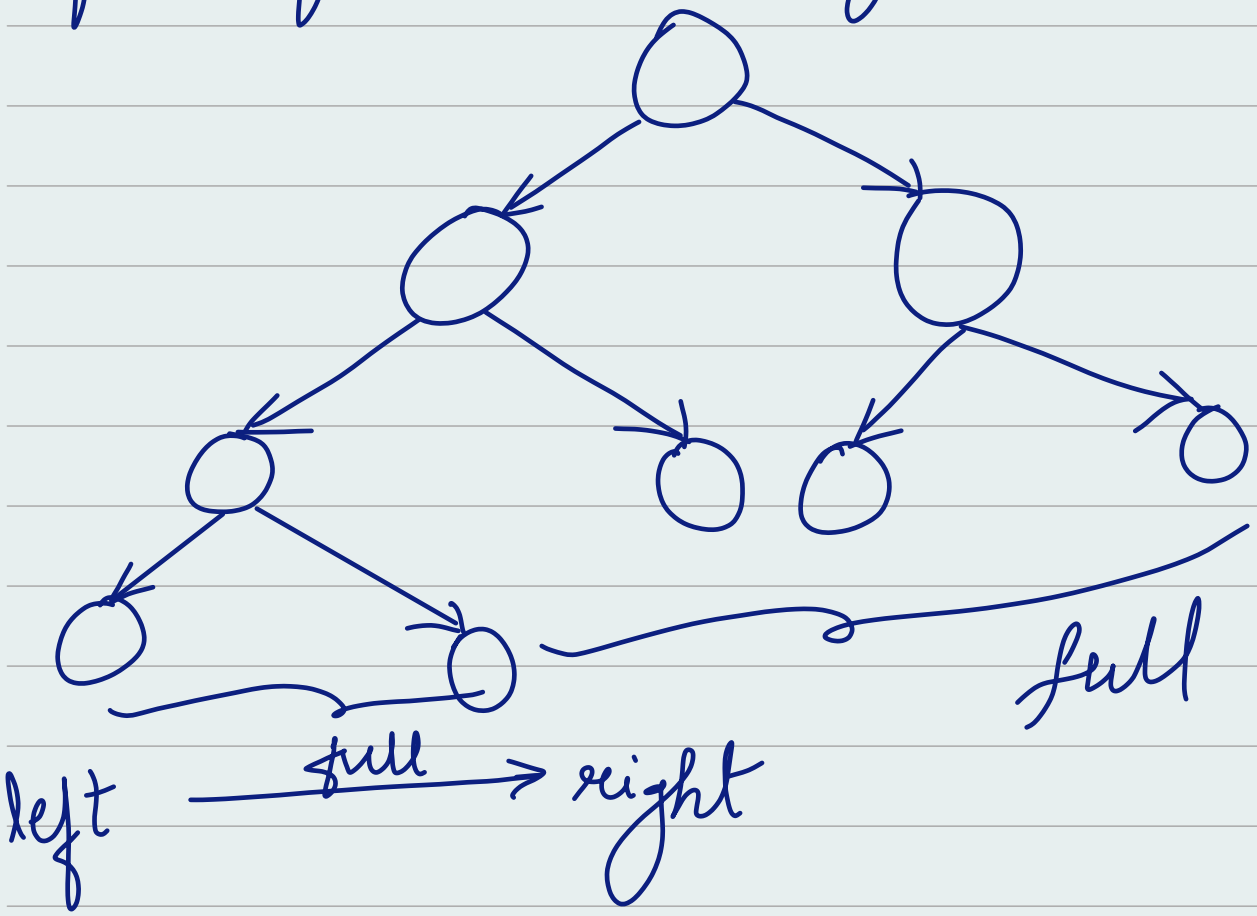
⑦ level → subtract height of root - height of node.
root level = 0

⑧ Ancestor and Descendent.

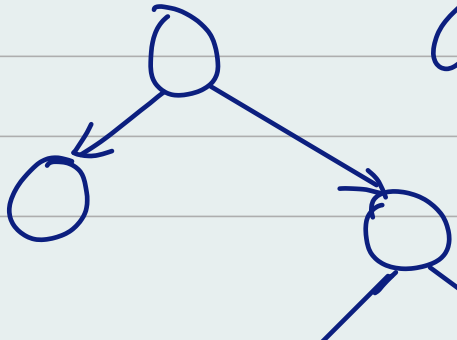
⑨ Degree = (0, 1, 2)

Types of Binary Tree :-

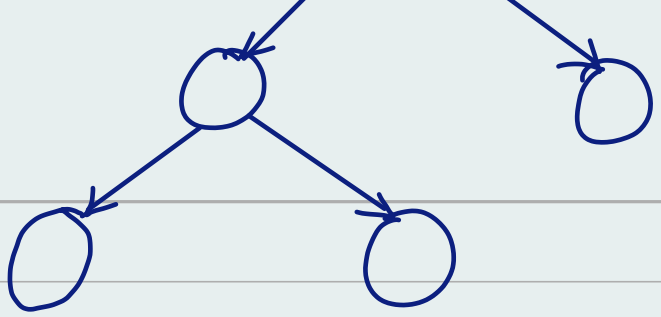
① Complete Binary Tree :- All the levels are full apart from the last level, but the last level is full from left to right.



② Full Binary Tree / Strict Binary Tree
Here each node must have zero children or two children - No single child.

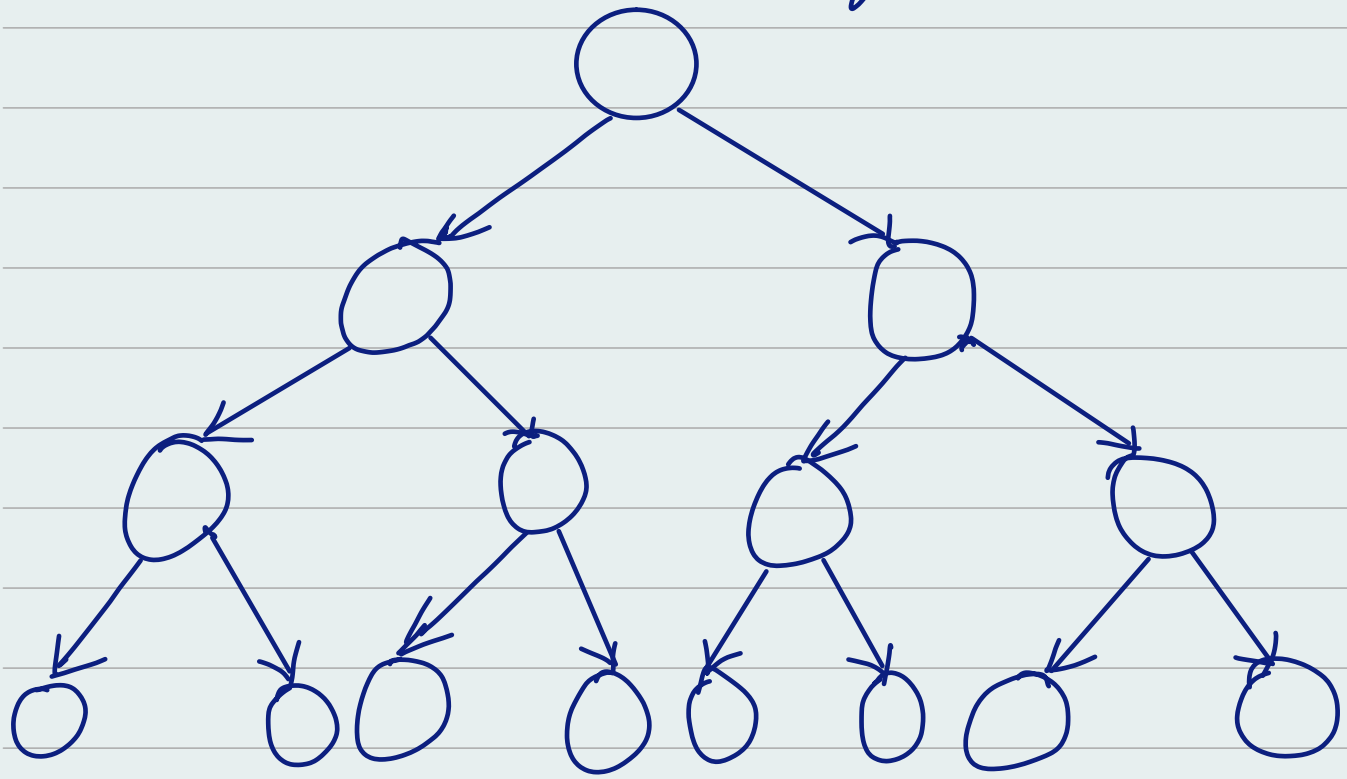


Use Case:
Compression/
segment trees



(3) Perfect Binary Tree

→ All levels are full

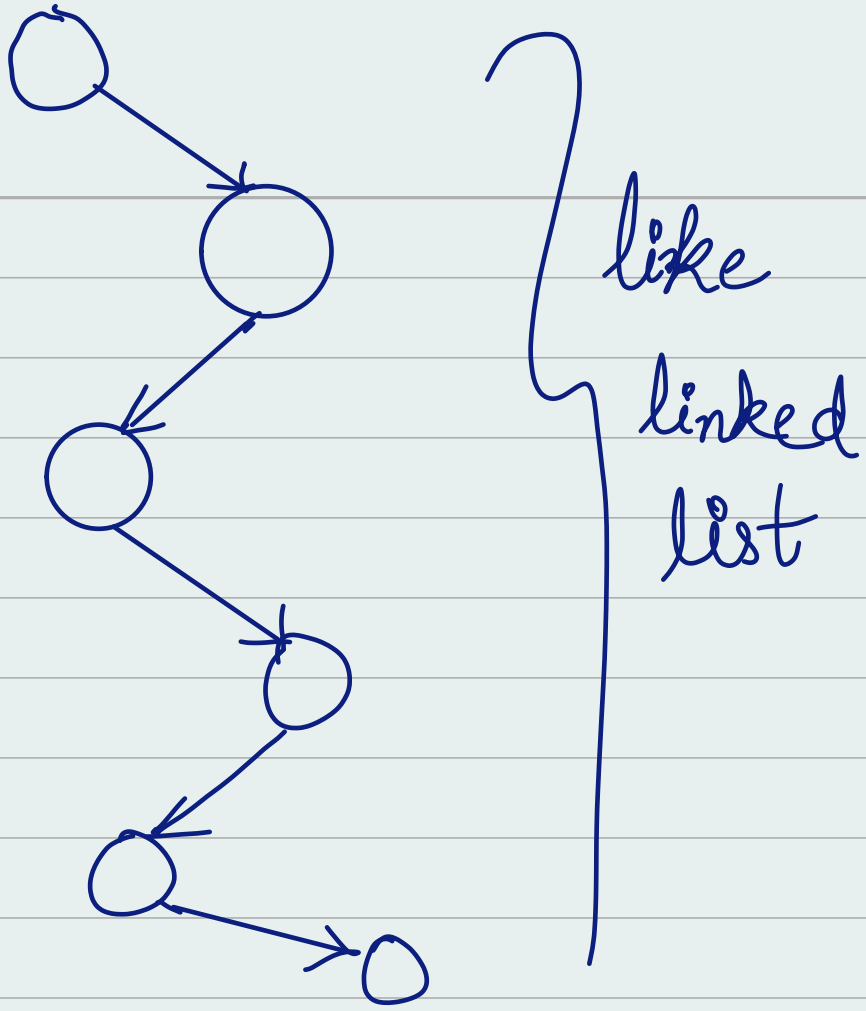


(4) Height balanced Binary Tree

↳ Average Height $O(\log n)$

(5) Skewed Binary Tree ∴ Every node has one child. Average height

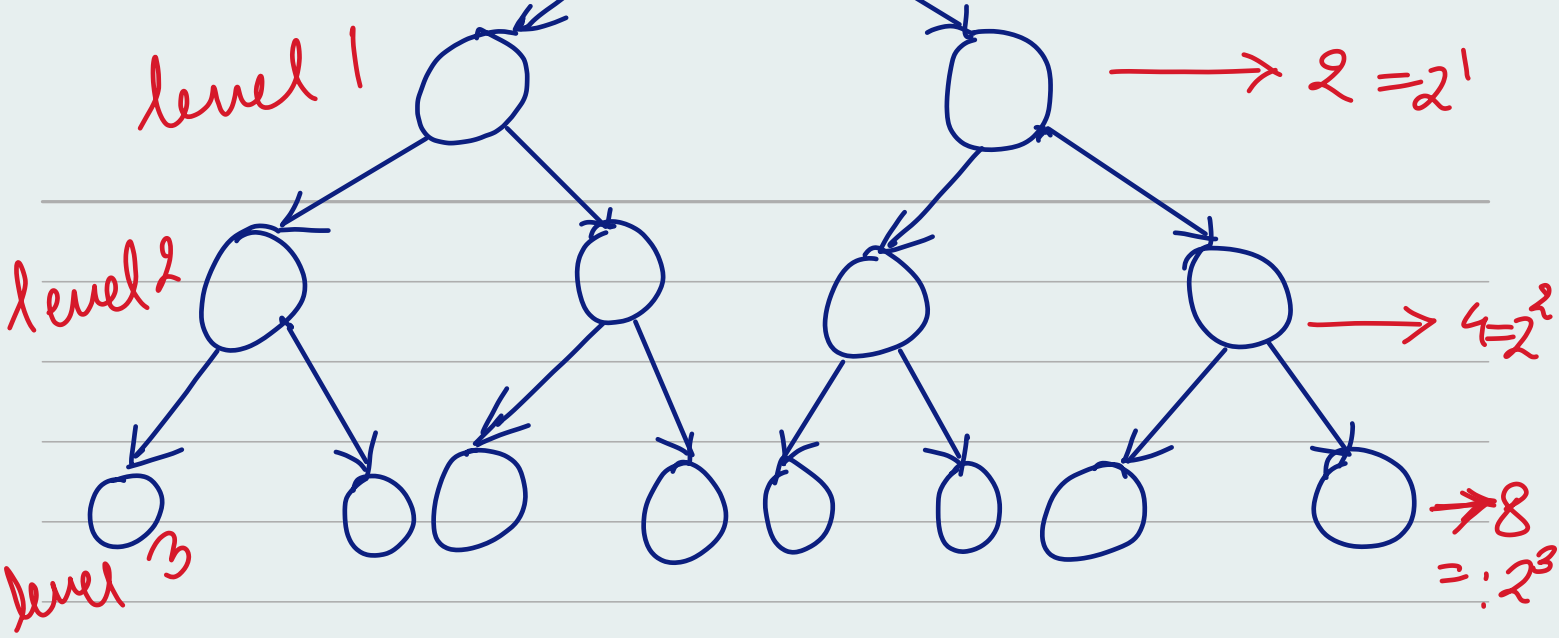
O (N) .



(6) Ordered Binary Tree :- Every node has some property or some conditions example :- Binary Search Tree.

Properties that will help us in some questions :-

i) Total number of nodes in a perfect B.T of height h is equal to $2^{(h+1)} - 1$



$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots + 2^h$ = Geometric Progression

$$GP = \left[2^{(n+1)} - 1\right]$$

(2) Leaf nodes in perfect B.T $= 2^{\text{height}}$

$$2^{h+1} - 1 - 2^h = 2^h - 1$$

(3) N = no of leaves

$\log N + 1$ levels at least

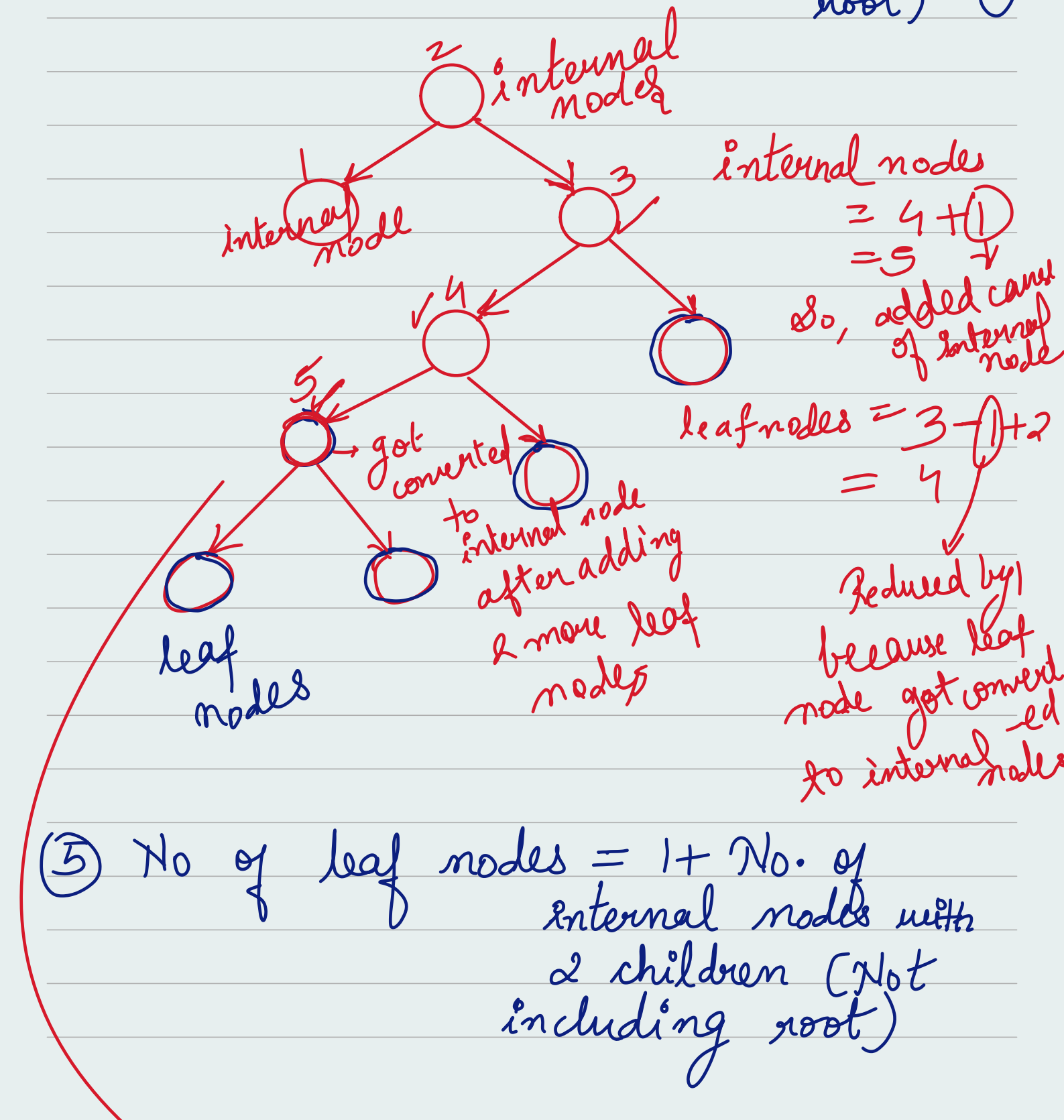
N nodes → $\log(N+1)$ min levels.

(4) Strict Binary Tree, N → leaf nodes

$(N-1)$ = internal nodes.

No. of leaf nodes = No of internal nodes + 1.

for strict BT (not including root)



(5) No of leaf nodes = 1 + No. of internal nodes with 2 children (Not including root)

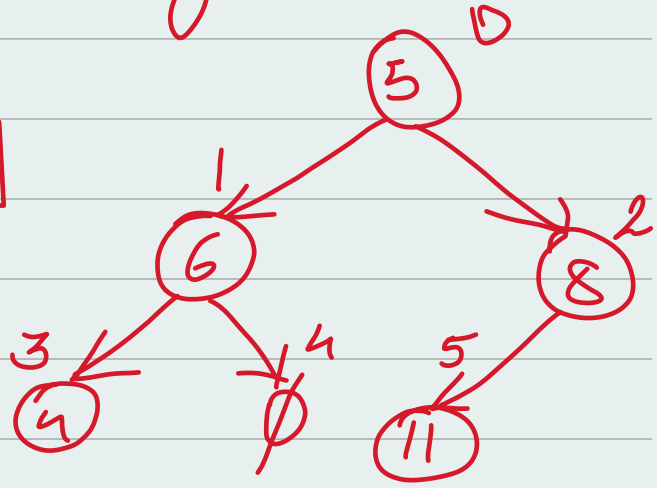
from the above figure

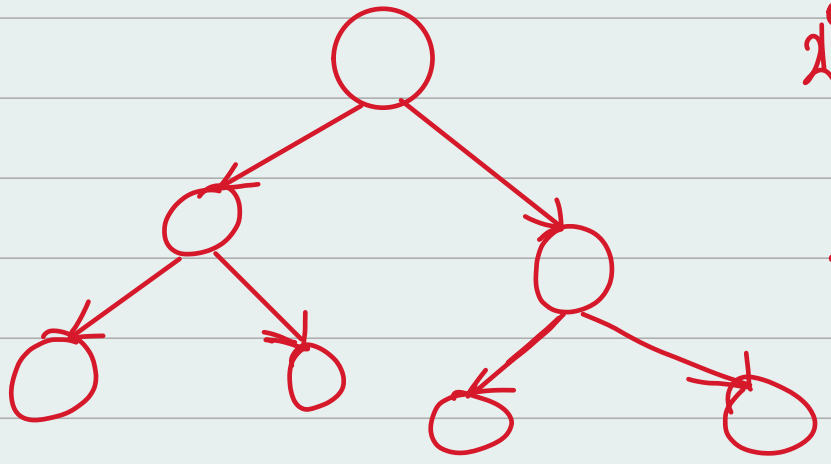
$4 = 1 + 3$

* Implementation :-

① linked representation

② Sequential → using array





2^0

2^1

2^2

2^3

.

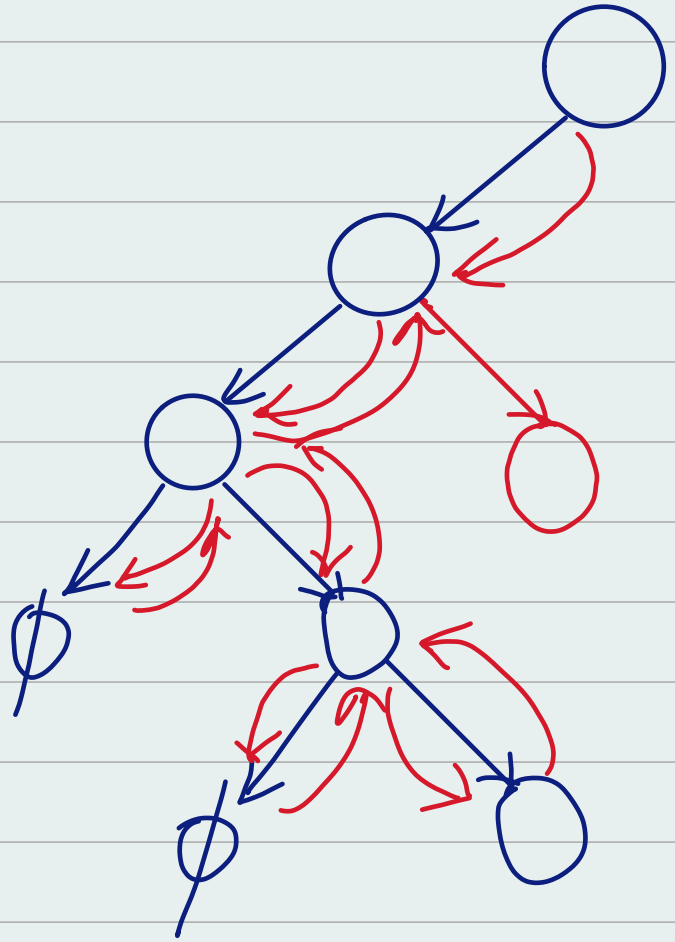
2^h

$N = 2^{h+1} = 2^h \cdot 2$

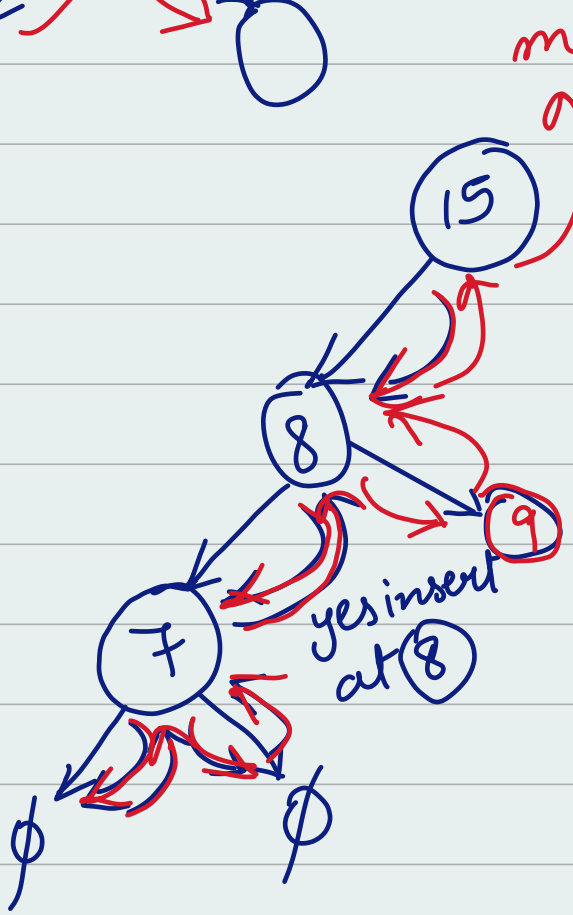
$\log N = h \log 2$

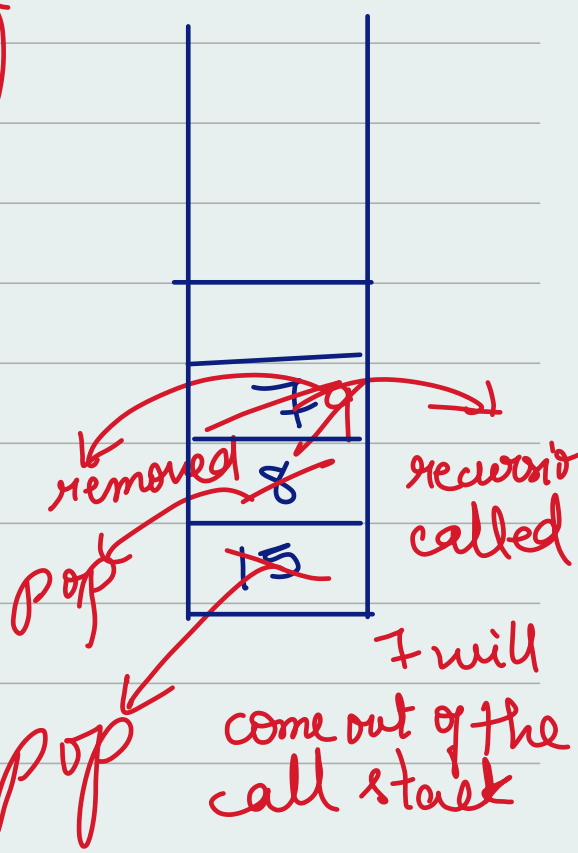
$h = O(\log(N))$

* Recursion



insert(
insert(left)
insert(right)





* Code'

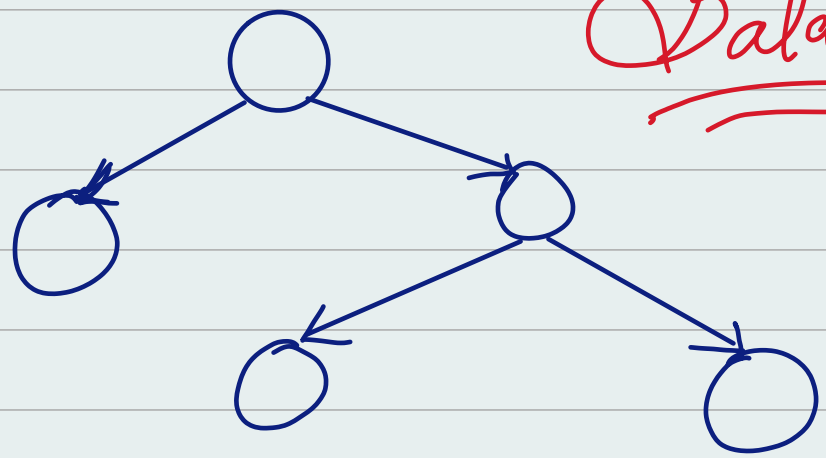
```
package BT;
import java.util.Scanner;
public class binaryTree {
private static class Node{
int value;
Node left;
Node right;
// when you create a new node, initialize the value
public Node(int value){
this.value = value;
}
}
private Node root; // insert elements
public void populate(Scanner scanner){
System.out.println("Enter the root node value: ");
int value = scanner.nextInt();
root = new Node(value); // call another recursion function from the root node
that will fill the entire tree as desire. populate(scanner, root);
}
private void populate(Scanner scanner, Node node) {
System.out.println("Do you want to enter left of " + node.value);
boolean left = scanner.nextBoolean();
if (left) { System.out.println("Enter the value of the left of " + node.value);
int value = scanner.nextInt();
node.left = new Node(value); populate(scanner, node.left);
}
System.out.println("Do you want to enter right of " + node.value);
boolean right = scanner.nextBoolean();
if (right) {
System.out.println("Enter the value of the right of " + node.value);
int value = scanner.nextInt();
node.right = new Node(value); populate(scanner, node.right);
}
}
public void display(){
display(root, "");
}
private void display(Node node, String indent){
if(node == null){
return;
}
System.out.println(indent + node.value); display(node.left, indent + "\t");
display(node.right, indent +ring indent){
if(node == null){
```

```
return;
}
System.out.println(indent + node.value); display(node.left, indent + "\t");
display(node.right, indent +"\t");
}
public void prettyDisplay(){
prettyDisplay(root, 0);
}
private void prettyDisplay(Node node, in"\t");
}
public void prettyDisplay(){
prettyDisplay(root, 0);
}
private void prettyDisplay(Node node, int level){
if(node == null){
return;
}
prettyDisplay(node.right, level + 1);
if(level != 0){
// means the node is one or two level down. for that add that many spaces.
for (int i = 0; i < level - 1; i++) {
System.out.println("|\t\t");
}
System.out.println("|------>" + node.value);
 }else{
 System.out.println(node.value);
}
prettyDisplay(node.left, level + 1);
}
 public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
 binaryTree tree = new binaryTree();
tree.populate(scanner);
tree.prettyDisplay();
}
}
```

Binary Search Tree

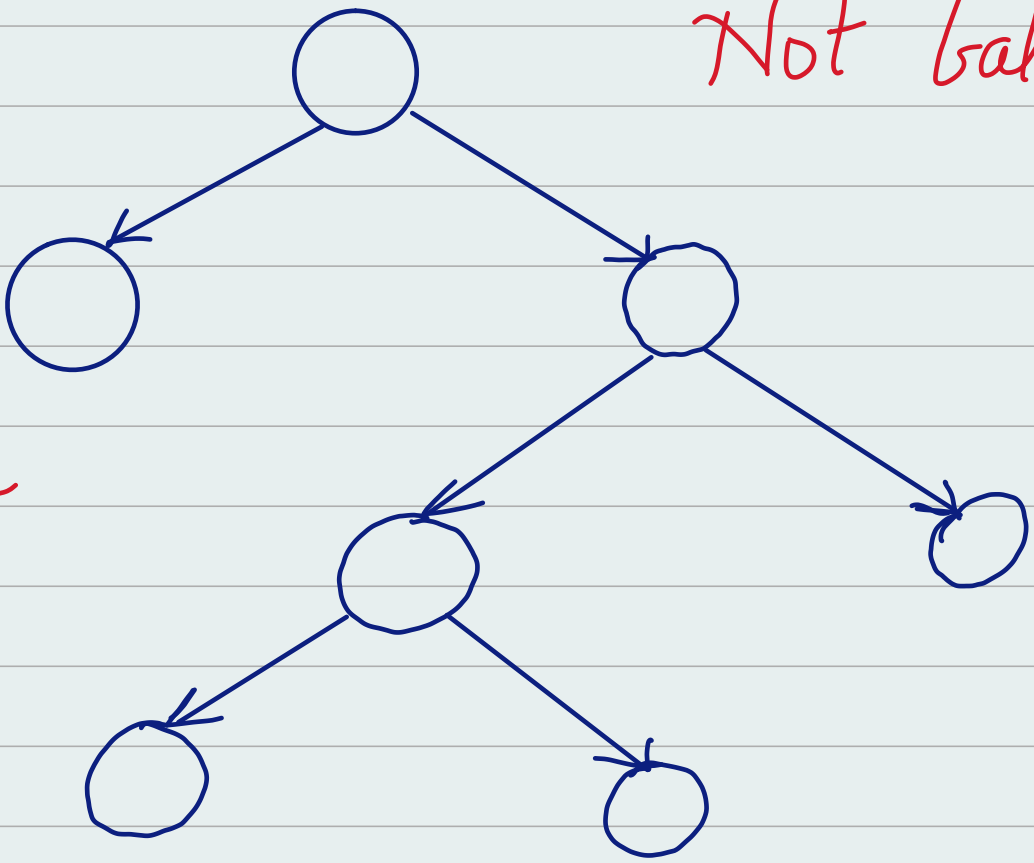
For Binary Search tree height difference should always be less than or equal to 1.

Balanced

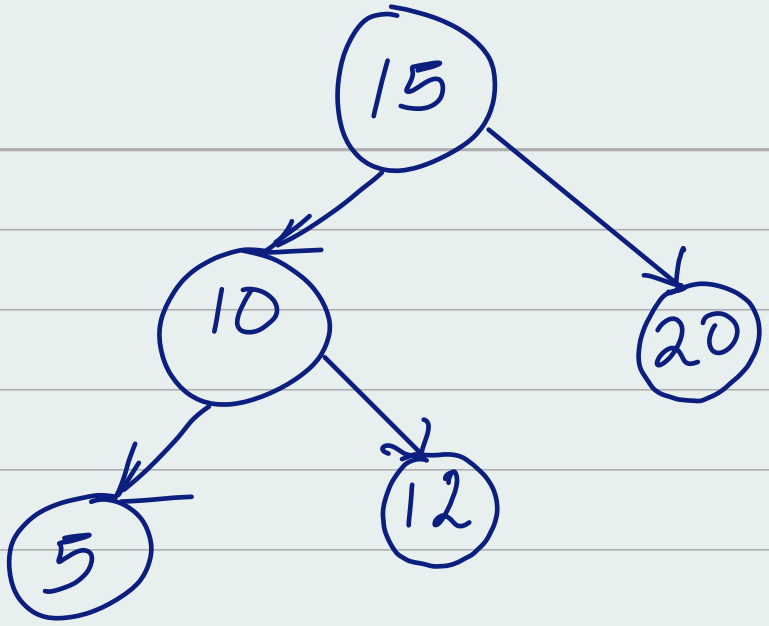


height difference is 1

Not balanced



height difference is 2.



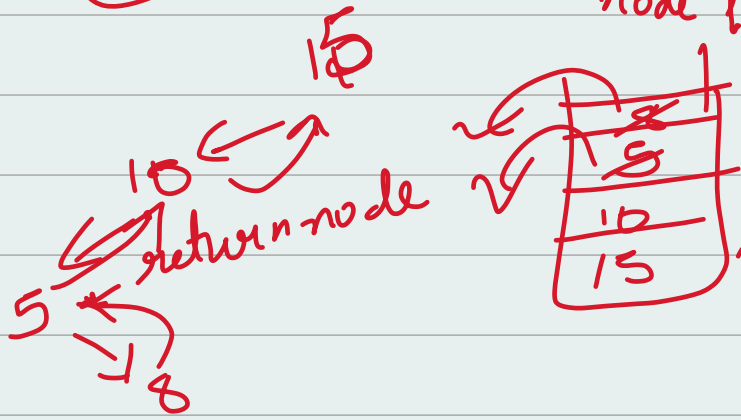

```
Node {
int value;
Node left;
Node right;
int height;
}
```

Time complexity O(log(N))

8 → Insert

If value < node value
node.left = insert(value, node.left)

If value > node.value
node.right = insert(value, node.right);



5.right = 8 ✓ then return the node

5 → right ⇒ null

create a new node;

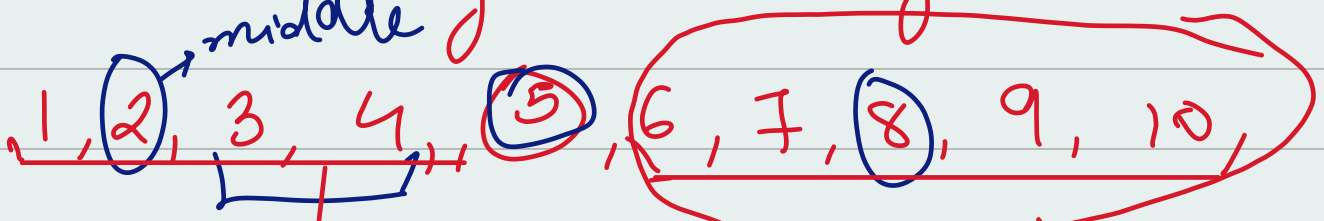
```
if (node == null) {
    node = new Node(value);
```

So when a node is added we have to increase the height of the node.

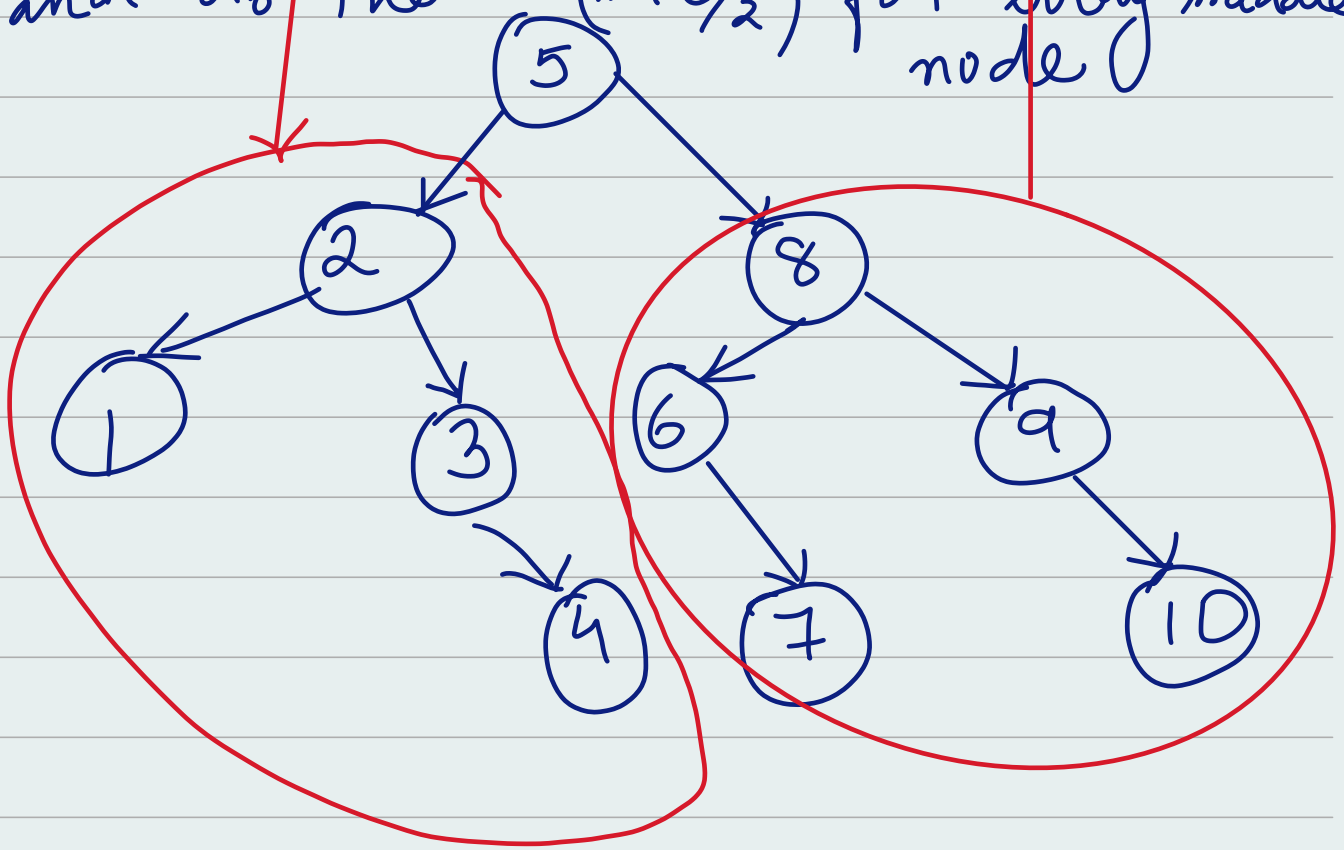
node.height = Math.max(height(node.left), height(node.right)) + 1;

& then return node

In Sorted array we can use recursion for building a binary tree.



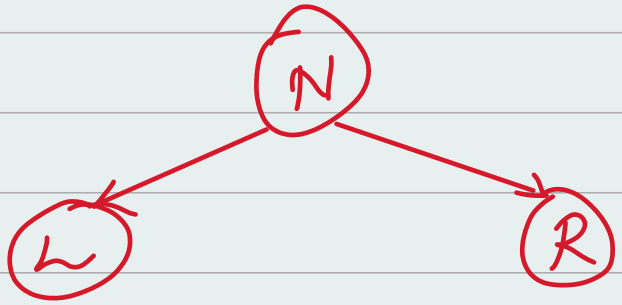
i)→ Take the middle element as a root. and do the $(s+e/2)$ for every middle node

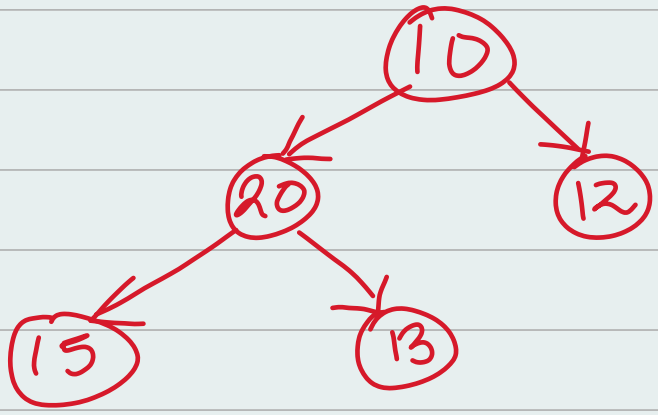


(i) * Traversal Methods :-

(1) Pre-order

N - L - R
(displayed)





10, 20, 15, 13, 12

Node — Left — Right

Code:

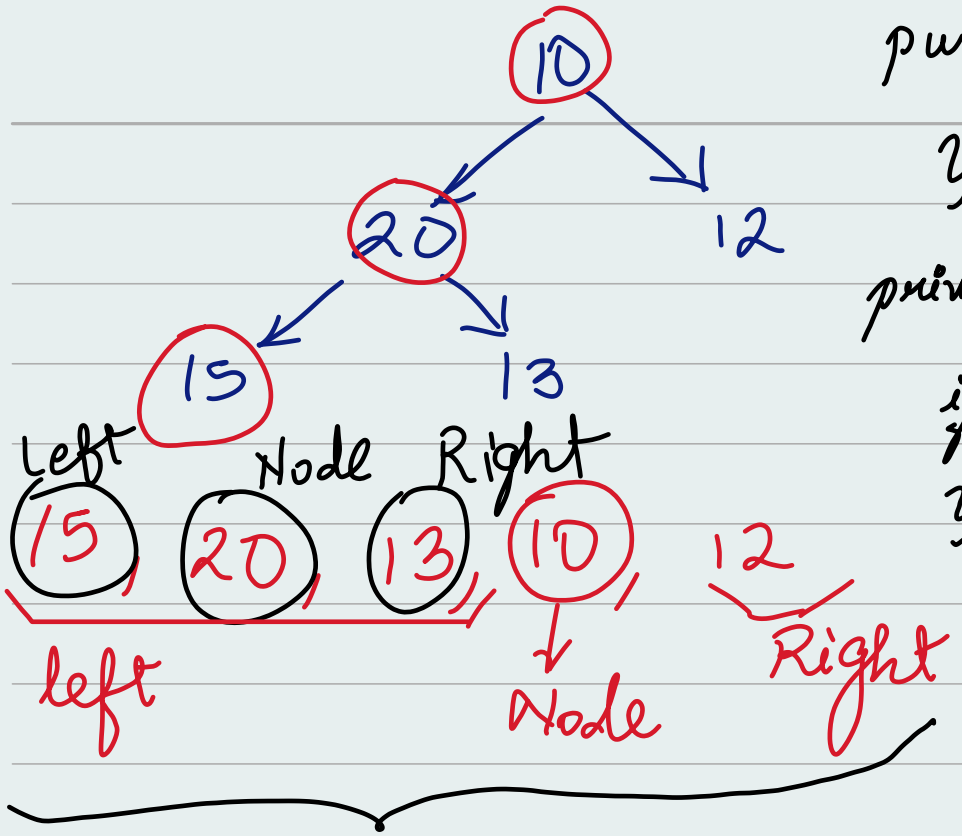
```
public void preOrder(){
    preOrder(root);
}

private void preOrder(Node node){
    if(node == null){
        return;
    }
    System.out.println(
        node.value + " ");
    preOrder(node.left);
    preOrder(node.right);
}
```

* Used for evaluating Math expressions or Making a copy.

* Serialization from String / Array.

(ii) * InOrder Traversals-

L → N → R

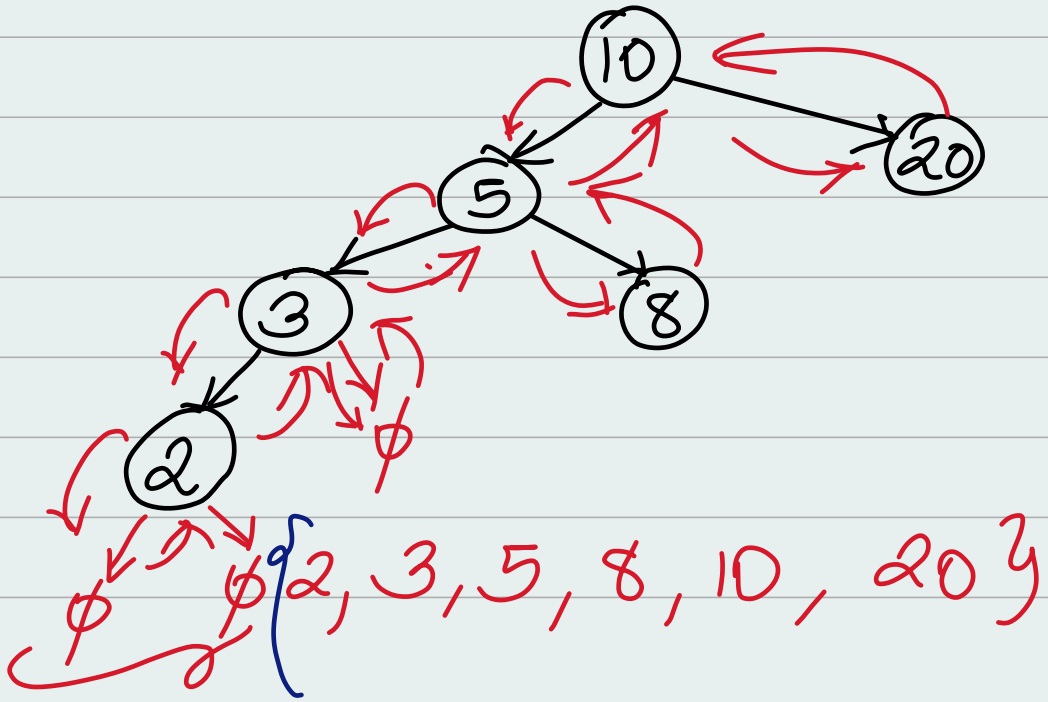


Code:

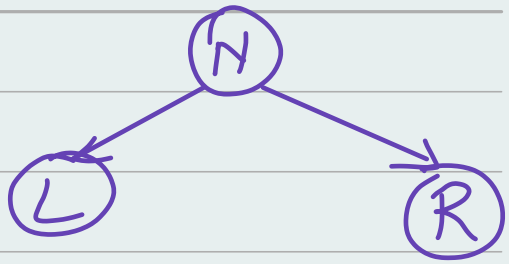
```
public void inOrder(){
    inOrder(root);
}

private void inOrder(Node node){
    if(node == null){
        return;
    }
    inOrder(node.left);
    System.out.println(node.value + " ");
    inOrder(node.right);
}
```

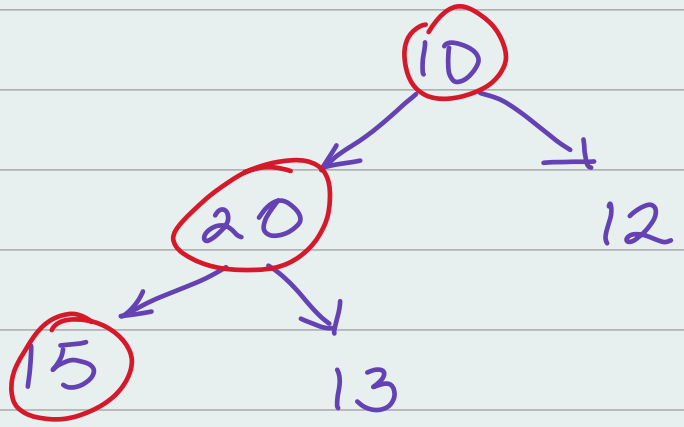
Again Recursion.

In BST, using inorder traversal we can visit the node in sorted manner.



(iii) * Post Order :





// print
{ node.left
node.right
node }

{ 15, 13, 20, 12, 10 }

★ Delete a binary tree

★ Bottom-up Calculation

Next topics are Breadth First Search & Depth First Search or Breadth First Traversals or Depth First Traversals.

Code for post Order :-

```
public void postOrder (){
    postOrder (root);
}
private void postOrder (Node node){
    if (node == null){
        return;
    }
    postOrder (node.left);
    postOrder (node.right);
    System.out.println (node
        .value + " ");
}
```