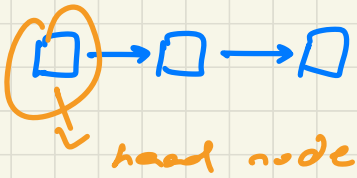
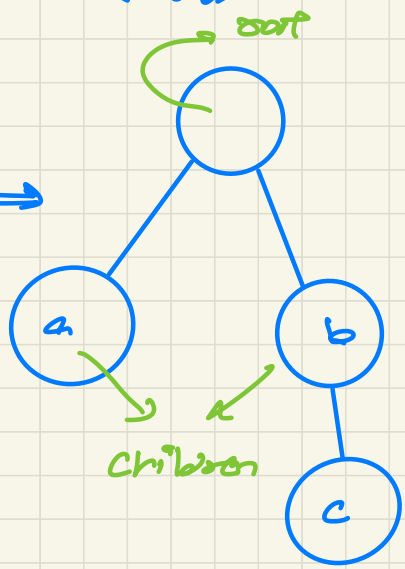

TREES

Trees

1. LinkedList :



(Right)



→ in trees, we've root node in place of head node

2. ① root node : have access to children

② a, b → children

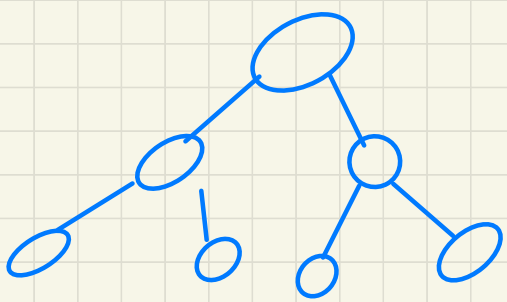
③ c → leaf node
↳ that has no further children

3. Generic tree node

↳ node class contain two attributes

① node data

② second attribute will be a list of children nodes



* List.add() function to add nodes to generic trees.

TreeNode<Integer> n1 = TreeNode<>(5);
TreeNode<Integer> n2 = TreeNode<>(2);

n1.children.add(n2) → adding the n2 in the children class of n1

4. Print a generic tree (recursively)

① if root = null, then the tree is empty, return null.

Code :
```
public void printTree(TreeNode root){
    if(root == null){
        return;
    }
    System.out.println(root.data);
    for(TreeNode child : root.children)
        printTree(child);
}
```

will call this line again & again till there are no more child nodes.

5. Height of a tree → length of the path from the tree's root node to any of its leaf node

Code : Case-1

```
public int maxDepth(TreeNode temp) {
    if (temp == null) {
        return -1;
    }
    int lDepth = maxDepth(temp.left);
    int rDepth = maxDepth(temp.right);
    if (lDepth > rDepth) {
        return (lDepth + 1);
    } else {
        return (rDepth + 1);
    }
}
```

will go to the left most node of the tree

will go to the right most node of the tree

+1 to include root node to the count.

Case-2 : if left & right nodes as none, not given, instead children node is present.

Node class where we don't have left or right node instead children list

```java
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
  public int maxDepth(Node root) {
    if (root == null)
      return 0;

    int ans = 0;

    for (Node child : root.children)
      ans = Math.max(ans, maxDepth(child));

    return 1 + ans; // ans+1 to add root node to the count
  }
}
```

at every iteration, maxchild(child) will update its value (increase by 1)

ternary operation (for loop)

initially depth = 0

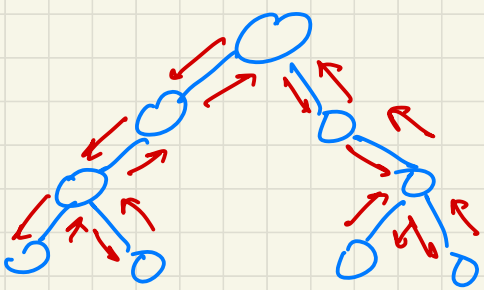
max of ans in maxdepth will be the output.

if height = 0, then still point max is 0.

6. Size of a tree → no. of elements present in the tree.

7. Depth first search

it will go the depth till either node.left or node.right = null.



first the deepest node is visited & then backtracking to its parent node if no sibling of that node exists.

* depth of node : dist from root
height of node : dist. from leaf

e.g. following is a code for checking whether the two trees are leaf similar or not, using dfs.

making two ArrayList where we'll store the values of the trees

Comparing the two lists on the basis of all the no. present in it.

node.left will be stored after node.right.

```
/**
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
  public boolean leafSimilar(TreeNode root1, TreeNode root2) {
    List<Integer> leaves1 = new ArrayList<>();
    List<Integer> leaves2 = new ArrayList<>();
    dfs(root1, leaves1);
    dfs(root2, leaves2);
    return leaves1.equals(leaves2); //returns true/false
  }

  public void dfs(TreeNode node, List<Integer> leaves) {
    if (node == null)
      return;
    if (node.left == null && node.right == null) {
      leaves.add(node.val);
      return;
    }

    dfs(node.right, leaves);
    dfs(node.left, leaves);
  }
}
```

only the leaf nodes will be there in the list.

if the left or right node are null, then add that node to the list.

node.right will be stored first.

8. Breadth first search (2) ⟷ level order traversal

: traversing algorithm where we start traversing from a selected node

(1) first node horizontally, & visit all the nodes of current layer.

(2) move to the next layer.

Code through the question on leetcode.

* (1) T.C of BFS : $O(V+E)$

V: vertices
E: edges

(when adjacency list is used)

: $O(V^2)$ (when adjacency matrix is used)

→ Adjancy list : graph as an array of linked list.

→ Adjancy matrix : way of representing a graph as a matrix of booleans

will be studied in the Graph topic

② TC of DFS : $O(V+E)$ { in case of adjancy list }

: $O(V^2)$ { in case of adjancy matrix }

BFS code using maxDepth example

making a queue where all the node data will be stored

```java
public int maxDepthTree(Node root) {
    if (root == null) return 0;

    int depth = 0;
    Queue<Node> queue = new LinkedList<Node>();
    queue.offer(root);
    queue.poll();

    while (!queue.isEmpty()) {
        int size = queue.size();

        for (int i = 0; i < size; i++) {
            Node currentnode = queue.poll();
            for (Node child : currentnode.children)
                queue.offer(child);
        }
        depth++;
    }
    return depth++;
}
}
```

.offer()

retrieves & removes head of the queue

will remove root & will show that

.poll()

retrieves but doesn't remove the head of the queue.

will make currentnode as the head of the queue after every addition to the queue.

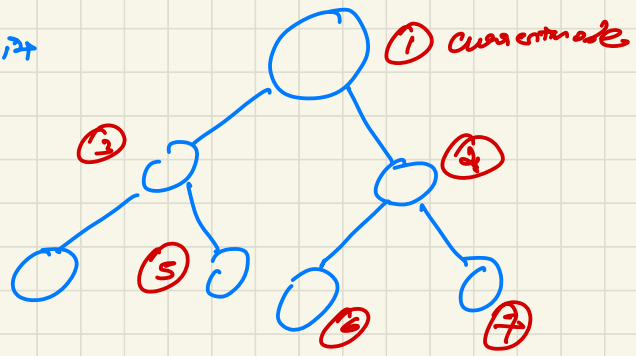
will show the child node till there are child nodes in the tree (level by level)

```
            for (int i = 0; i < size; i++) {
                Node currentnode = queue.poll();
                for (Node child : currentnode.children)
                    queue.offer(child);
            }
            depth++;
        }
        return depth++;
    }
}
```

the loop will run till the end of the queue is

Same process repeated till node has children

when (1) → currentnode, it will go to its children rather than going into the depth first, then (2) & (3) will be showed as child & depth will be increased by one.



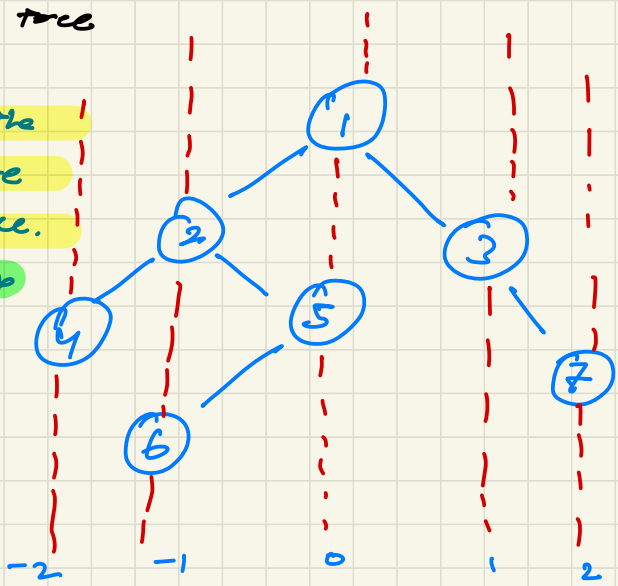
9. Top view of a binary tree (3)

the first node cutting the vertical line will be the top view of the binary tree.

requirement : queue + map

m (line, node)

q

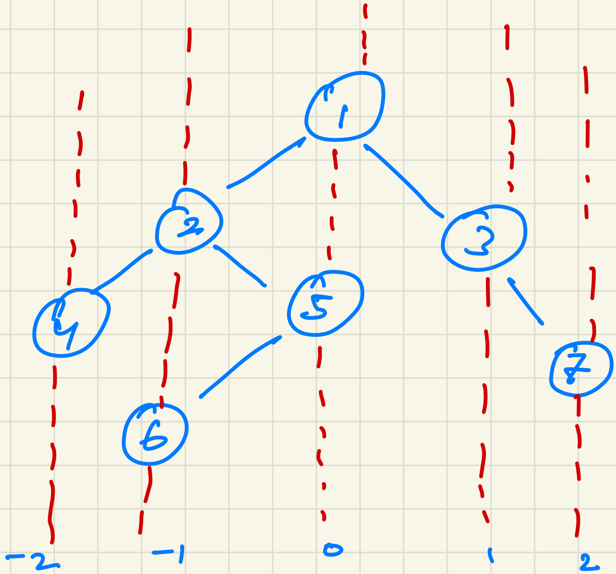


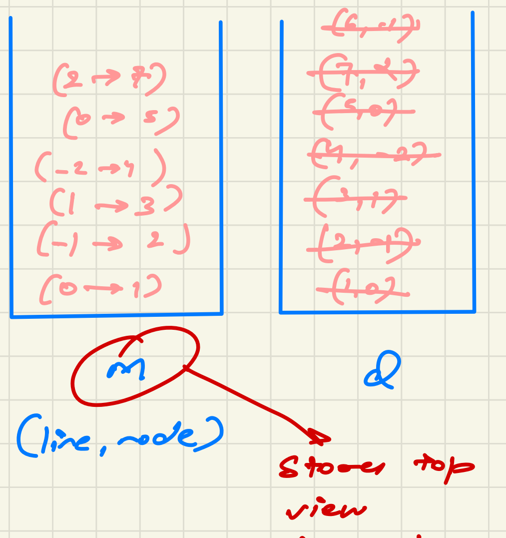
Q : to store initial

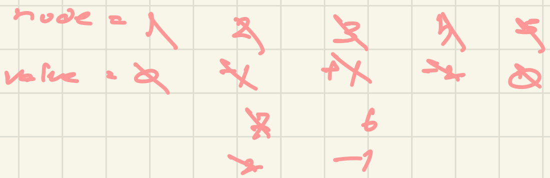
M : nodes on the vertical line.

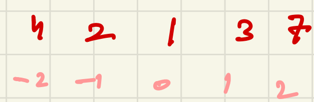
* Map data structure

1. abstract data type
2. stores key-value pairs
3. there can't be duplicate keys
4. the key is used to decide where to store the object in the structure.










```java
private void TopView(TreeNode root)
{
    class QueueObj {
        6 usages
        TreeNode node;
        5 usages
        int hd;

        3 usages
        QueueObj(TreeNode node, int hd)
        {
            this.node = node;
            this.hd = hd;
        }
    }
    Queue<QueueObj> q = new LinkedList<QueueObj>();
    Map<Integer, TreeNode> topViewMap
            = new TreeMap<Integer, TreeNode>();
    if (root == null) {
        return;
    }
    else {
        q.add(new QueueObj(root, hd: 0));
    }
    System.out.println("The top view of the tree is : ");
    while (!q.isEmpty()) {
        QueueObj tmpNode = q.poll();
        if (!topViewMap.containsKey(tmpNode.hd)) {
            topViewMap.put(tmpNode.hd, tmpNode.node);
        }
        if (tmpNode.node.left != null) {
            q.add(new QueueObj(tmpNode.node.left, hd: tmpNode.hd - 1));
        }
        if (tmpNode.node.right != null) {
            q.add(new QueueObj(tmpNode.node.right, hd: tmpNode.hd + 1));
        }
```

Node class

Queue to store the values

map structure to store the node as the vertical line number

if map contains the key that is equivalent to hd, then it will remove that & will show

if node.left or node.right != null, then it will add that to the q with the corresponding hd.

Also the map dt will not allow duplicate items, ∴ the node with same hd will not be added

* Map terms in the code :

① .entry() : → interface
→ returns the key corresponding to this entry

② .entrySet() : a map entry (key-value pair)
∴ entry may be unmodifiable
∴ entry may be independent of any map.

returns the set view of the mapping contained in the map.

{a collection that contains no duplicate elements.}

③ .containsKey() : return true if this map maps one or more keys to the specified value.
∴ returns true if & only if the map contains at least one mapping to a value V.

④ .put() : if the map contains a mapping from key K to value V, then that mapping is removed.

(5) `.getValue()` : replaces the value corresponding to this entry with the specified value.

```
        for (Map.Entry<Integer, TreeNode> entry :
                topViewMap.entrySet()) {
            System.out.print(entry.getValue().val + " ");
        }
    }
}
```

will perform entrySet operation & as it's a set, no duplicate values will be allowed & ∴

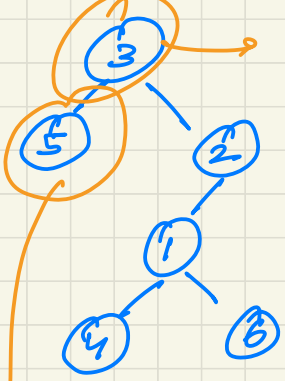
10. Binary tree to DLL : Constraints

(4)

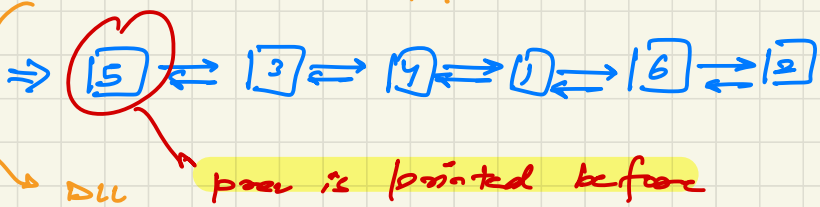
(1) inorder form of BT
(2) left node : prev
right node : next

Rough code :

```
Node prev = null ; head = null;
void convertToDLL (Node root){
    if (root == null) return null;
    convertToDLL ( root.left) ;
    if ( prev == null) head = root ;
    else {
      root.left = prev ;
      prev.right = root ;
    }
    prev = root ;
    convertToDLL ( root.right ) ;
}
```



prev means that root is also the left node to some node & if prev = null, head = root → BT



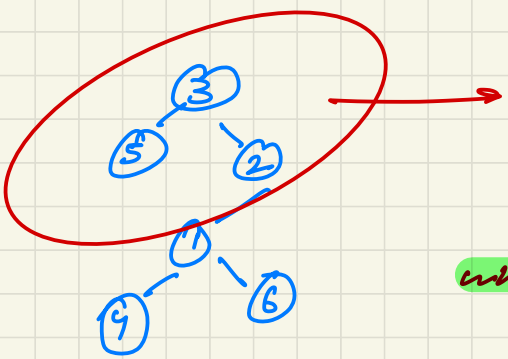
DLL

prev is pointed before root (head)

root.left = prev

&

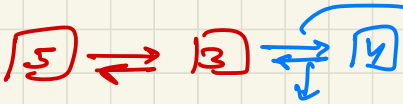
prev will be pointed first in the DLL as we are following the inorder traversal.



if prev != null, i.e. when head = 5, then prev = 3 & prev.right = root i.e. root = 2 then

when (2) → root.prev ≠ null,

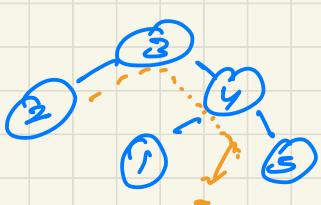
∴ root.left = (1) & then again the same function.



root.left = prev

prev.right = root

11. Diameter of a binary tree

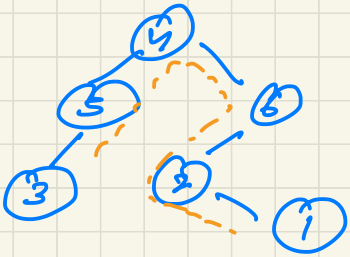


Diameter of BT

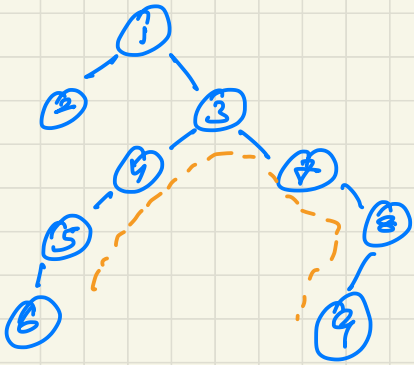
↓

no. of nodes in the largest path b/w two leaf nodes

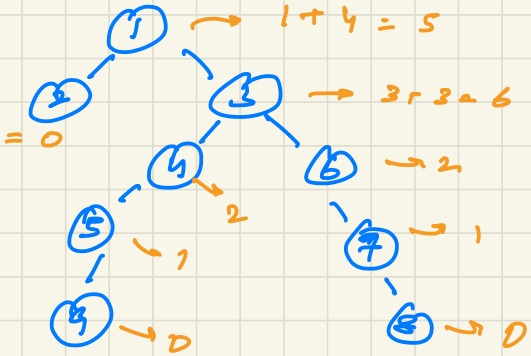
lh + rh = diameter



: Diameter = 5



Diameter : 6



```java
public int diameterOfBinaryTree(TreeNode root) {
    maxDepth(root);
    return ans;
}

3 usages
private int ans = 0;

2 usages
int maxDepth2(TreeNode root) {
    if (root == null)
        return 0;

    final int l = maxDepth2(root.left);
    final int r = maxDepth2(root.right);
    ans = Math.max(ans, l + r);
    return 1 + Math.max(l, r);
}
}
```

this function will return ans that will give diameter through max(ans, l+r) in maxdepth2 function

current depth = 0 in the starting

will go to root.left till null object

same as left going

comparing l+r and diameter as it get updated everytime

with give the tree height

maxdepth2 will return the max b/w and & l+r2 diameter.

Step process : ① Use maxdepth function previously used in calculating the height of the function

② make changes in the height function by changing to l+r

③ make a function diameter seperately that will return ans we get in the maxDepth function & then make the diameter function return ans.

Binary Search Tree

1. Binary search tree is a node-based binary tree which has following structure :

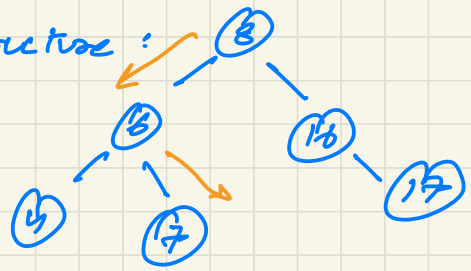
① Right node's value is bigger than root node's value

② left node's value is smaller than root node's value

rough code :

```
class BST {
    boolean search(Node root, int x) {
        if(root == null) return false;
        if(root.data == x) return true;
        if(root.data > x) {
            return search(root.left, x);
        }
        return search(root.right, x);
    }
}
```

Structure :



to search 7, as value is less than root, then will go to the left side.
Now, root = 6 & 7 is higher than 6 so going to right

T.C. : $\log_2 n$

Advantages : (1) searching the element in the binary tree is easy

(2) insertion & deletion are easier in BST.

2. Insertion in a BST (6)

```
public TreeNode(int val) {
    this.val = val;
```

this function because the function in TreeNode that takes only one input is this one

```
package Trees;

public class BinaryTree extends TreeNode {

    public BinaryTree(int val) {
        super(val);
    }

    public BinaryTree(int val, TreeNode left, TreeNode right) {
        super(val, left, right);
    }

    2 usages
    public TreeNode search(TreeNode root, int val) {
        if(root==null || root.val==val) return root;
        if(root.val>val){
            return search(root.left,val);
        }
        return search(root.right,val);
    }
}
```

this super will call

if root.val > n, that means it will be put at the left of the root node

if root.val < n, then putting n at the right of the root

```
public TreeNode(int val, TreeNode left, TreeNode right) {
    this.val = val;
    this.left = left;
    this.right = right;
}
```

taking three value input.

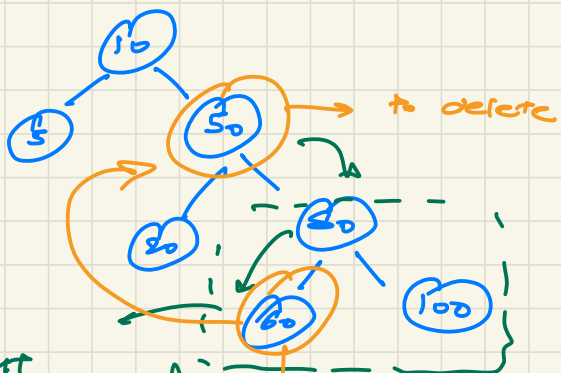
3. Note : (1) if we follow the inorder traversal of the final BST, we will get the sorted array.

(2) T.C of insertion for each node : O(Log H) — height of the tree

(3) for inserting N nodes, complexibility will be O(N * log(n))

4. Deletion in BST

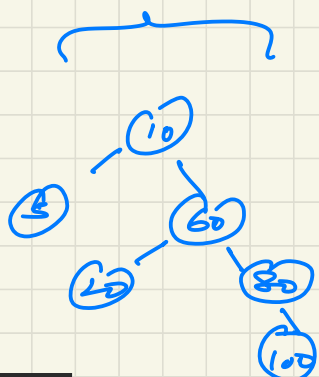
(7)



Smallest element of right subtree

left of 60 should be null for 60 to be the smallest.

Satisfy the BST condition



going to left is deleting the left node - going to the left node recursively

if left to the root node is null, then return the right node

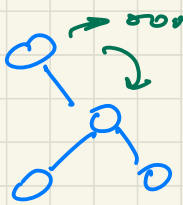
```
public TreeNode deleteNode(TreeNode root, int val) {
    if (root == null) return root;
    if (val < root.val) {
        root.left = deleteNode(root.left, val);
    } else if (val > root.val) {
        root.right = deleteNode(root.right, val);
    } else {
        if (root.left == null) return root.right;
        else if (root.right == null) return root.left;
        root.val = minValue(root.right);
        root.right = deleteNode(root.right, val);
    }
    return root;
}
```

deleting the inorder successor

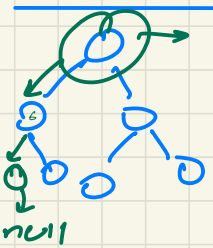
Same as for the left recursive call.

changing the root if the root to be deleted further has a tree

root.right = deleteNode(root.right, val);



; deleteNode will find the smallest value in the right subtree & will delete it.




```java
public int minValue(TreeNode root){
    int minV = root.val;
    while(root.left != null){
        minV = root.left.val;
        root = root.left;
    }
    return minV;
}
}
```

root = root.left

will make 6 as the root node

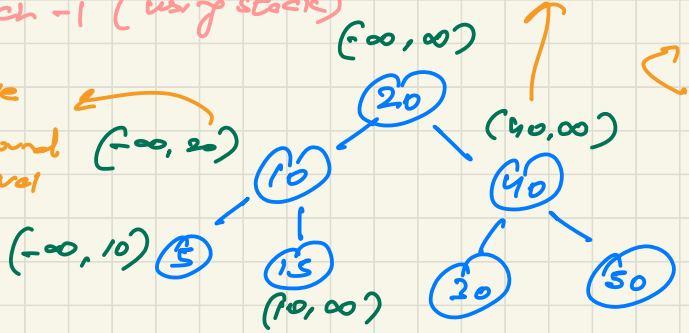
minValue function will return smallest element in the tree

Step process

5. Check for BST (Leetcode 98)

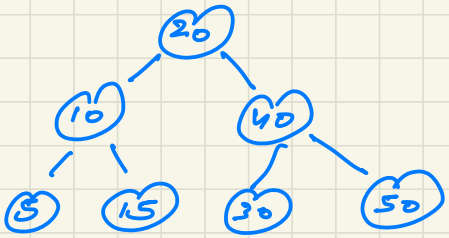
8

* Approach - 1 (using stack)



worst case : TC → O(n)

* Inorder traversal of a BST is a sorted array

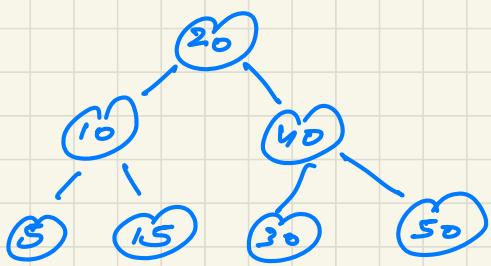


⇒ 5 10 15 20 30 40 50

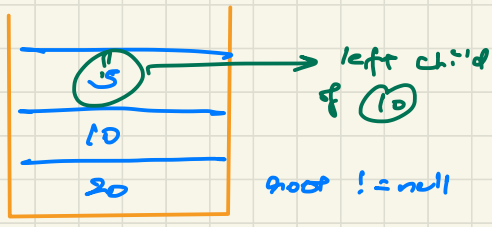
↳ sorted array

T.C : O(n) & S.P : O(n)

→ To complete the function in O(1), we can instead of storing them in the array, we can check if the next value is greater than the poev val.




```
while(root != null){
    stack.push(root);
    root = root.left
}
```



pop

root.left to stack.push

∴ left child has to be smaller

5 10 20

root.val <= left child ?

↳ false if no

null

```
root = stack.pop();
if ( root.val <= left_child_val ) return false;
left_child_val = root.val;
root = root.right;
```

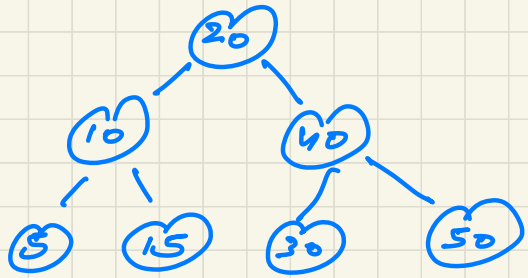
right traversing

20 will be pop

left_child_val = 20

root.val (after root.right) = 40

20 → 40 → 50



(Stack · LIFO)

creating a stack to store the values of BST

creating a variable that has the max value

```java
public boolean isValidBST(TreeNode root) {
    Stack<TreeNode> stack = new Stack<TreeNode>();
    double left_child_value = -Double.MAX_VALUE;
    while(!stack.isEmpty() || root!=null){
        while(root!=null){
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        if(root.val<=left_child_value) return false;
        left_child_value = root.val;
        root = root.right;
    }
    return true;
}
```

if root != null, keep adding in the stack

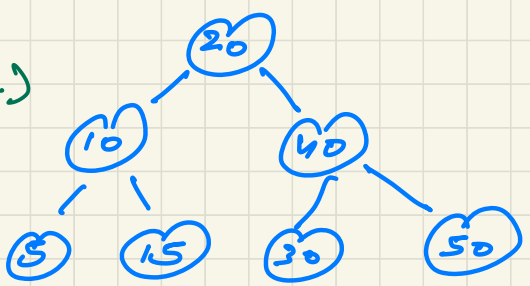
left side traversal

right side traversal to check if BST or not.

if (root.val <= left_child_value)
    return false;

20 < root.left.val ?

false to check the BST condition.



* Approach -2 (recursive)

T.C : O(n)

S.C : O(h)

```java
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, null, null);
    }

    private boolean isValidBST(TreeNode root, TreeNode minNode, TreeNode maxNode) {
        if (root == null)
            return true;
        if (minNode != null && root.val <= minNode.val)
            return false;
        if (maxNode != null && root.val >= maxNode.val)
            return false;

        return isValidBST(root.left, minNode, root) &&
               isValidBST(root.right, root, maxNode);
    }
}
```

a function with arguments like minNode & maxNode.

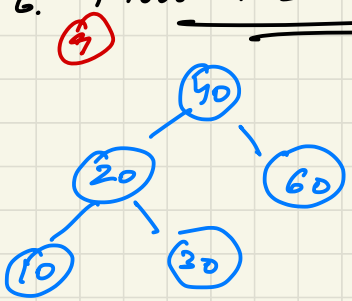
for left side, maxNode is root & minNode will be updated by the root.left value everytime

Similarly with maxNode

null

null node is also a BST

6. Floor & ceil value in a BST



val : 25

sbse bdi toh jo val as root.

ceil value : 30

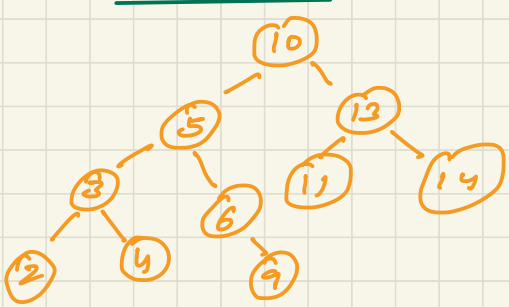
floor value : 20

just choti

{ to prevent the usage of space, we'll go iteratively instead of recursively. }

1) Ceil value : Val ≥ Key

Key = 8



{ Ceil : Smallest number that is greater than the key }

e.g. 9 if Key = 8

iterating only half of the code every time

∴ T.C → O(log₂N)

Code :

making a ceil variable to store -1

```java
public int Ceil(TreeNode root, int val) {
    int ceil = -1;
    while(root != null) {
        if(root.val==val){
            ceil = root.val;
            return ceil;
        }
        if(val>root.val){
            root = root.right;
        }
        else{
            ceil = root.val;
            root=root.left;
        }
    }
    return ceil;
}
```

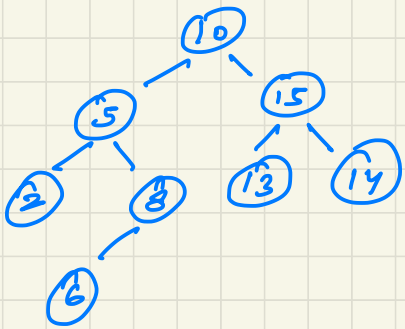
if val = root.val it will return the ceil.

iterating to right & changing root as long as root.val < val

iterating on the left side

returning the ceil.

(2) Floor value → key = 7 ∴ floor value : 6



val < key → just choti value

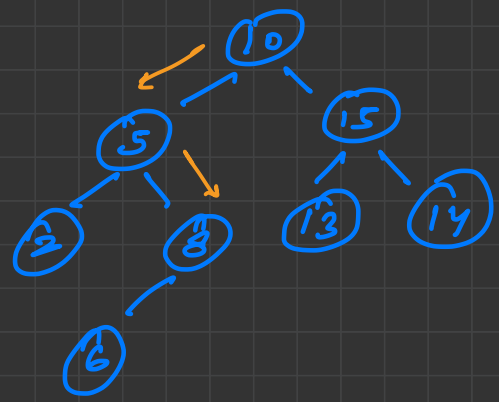
key = 9 floor value :

↑ val ≤ key

* moving left because we want just small value.

Code & explanation

```java
public int Floor(TreeNode root, int val){
    int floor = -1;
    while(root != null) {
        if (root.val == val) {
            floor = root.val;
            return floor;
        }
        if(val>root.val){
            floor = root.val;
            root = root.right;
        }
        else{
            root = root.left;
        }
    }
    return floor;
  }
}
```



okay run : val : 9

root. val = 10

val < root.val

∴ else condition

[1]

Now, root = 5

root. val = 5 &

val = 9

val > root. val

∴ root. right

[2] Now, root = 8 ∵ root. val = 8, val = 9 } val > root.val

but root. right = null ∴ it will return ∴ right condition floor i.e.

floor = root. val = 8

{ ∵ for val = 9, floor val = 8 }

7. Two sum → { Given an array of integers & the integer target, return the indices of the no. such that they add up to target. }

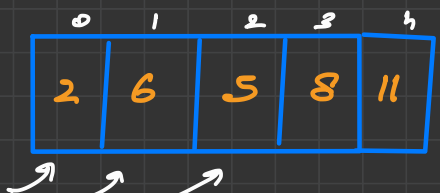
target = 14

i → 14 - 2 = 12 exist ?

(i) → 14 - 6 = 8 exist ?

} Answer : [1,3]

∴ T.C → O(n²) & S.C → O(1)



14 - 2 = 12

14 - 6 = 8

14 - 5 = 9

(5,2)
(6,1)
(2,0)

14 - 8 = 6

6 exist in the hash map at index = 1

{ we are checking if these values are in the hash table } Hash table

→ we get 3 from the pointer i & 1 from the hash table. ∴ (1,3) is the answer { (3,1) is also the accepted answer }

T.C → O(N) for searching & insertion

S.C → O(N)

Code & explanation → result : contains the index of

the set that needs to be returned at the end

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int[] result = new int[2];
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < numbers.length; i++) {
            if (map.containsKey(target - numbers[i])) {
                result[1] = i;
                result[0] = map.get(target - numbers[i]);
                return result;
            }
            map.put(numbers[i], i);
        }
        return result;
    }
}
```

if there exist the key then i will be the result[1]

Searches for key & returns the key for desired parameter.

i : index of value in number array

8. Delete BST → if we delete the root node, the children nodes of root become eligible for garbage collection

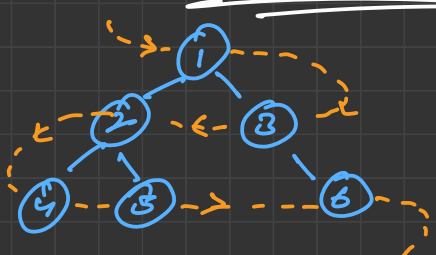
(10)

Code : public void deleteBST(){
root = null;
System.out.println("BST has been deleted");
}

T.C → O(1) & S.C → O(1)

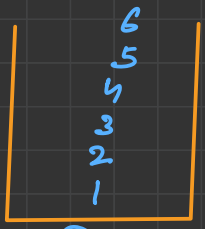
Questions

11. Zig-Zag / spiral traversal in binary tree



Zig-Zag : 1, 3, 2, 4, 5, 6

a variable flag is required

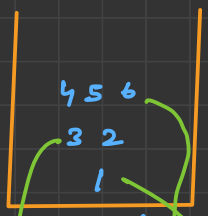


1 → once one level is cleared, increase the value of flag

flag = 0 in the beginning

flag → = 0 : L → R
→ = 1 : R → L

2 → when 2 is done, flag = +1 & move to 3.



flag = 0 → Adding to stack from L → R.
flag = 1 → adding to stack from R → L

Vector / ds

flag = 0 (L → R)

flag = 0 again (L → R)

flag = 1 (R → L)

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        if (root == null)
            return new ArrayList<>();

        List<List<Integer>> ans = new ArrayList<>();
        Deque<TreeNode> q = new ArrayDeque<>(Arrays.asList(root));
        boolean isLeftToRight = true;

        while (!q.isEmpty()) {
            List<Integer> currLevel = new ArrayList<>();
            for (int sz = q.size(); sz > 0; --sz) {
                TreeNode node = q.poll();
                if (isLeftToRight)
                    currLevel.add(node.val);
                else
                    currLevel.add(0, node.val);
                if (node.left != null)
                    q.offer(node.left);
                if (node.right != null)
                    q.offer(node.right);
            }
            ans.add(currLevel);
            isLeftToRight = !isLeftToRight;
        }

        return ans;
    }
}
```

Main queue

child queue

Starting from the deque end

returning the queue elements from the top

if true then add node.val

else, add from the 0th index

after every iteration isLeftToRight's value changes (i.e. true → false)

① After every iteration, isleftToRight variable changes its value.

For true, it adds node.val & for else, it adds at the 0th index because everytime, we perform

```
TreeNode node = q.poll()
```

this line makes the queue empty everytime starting from the last element to first.

→ Same can be done in case of Stacks. Code will be like

```
while (stack.size > 0){
    node = stack.pop();
    System.out.println(node.val + " ");
}
```

② So in the beginning, isleftToRight has been set to true & if true, we've set it to point L→R. If isleftToRight = false → will point R→L.

& after every iteration isleftToRight changes value.

* Difference in push() & add()

push() returns the object you are pushing

add() always returns true if added successfully.

③ .poll(): retrieve & remove the first element of the queue or the element pointed as head.

Terms : ① . asList : returns a fin-size list backed by the specified array

the returned list is serialisable & implements randomAccess

* Serialisable : a) interface present in java.io package

b) It's a marker interface that doesn't have any methods & fields.

mechanism of converting the state of an object into a byte stream.

* RandomAccess : the primary purpose of this interface is to allow generic algorithms to alter their behaviour to provide good performance when applied to random or sequential access list.

② Deque < TreeNode > q = new ArrayDeque <> (Arrays.asList(root));

Doubly ended queue

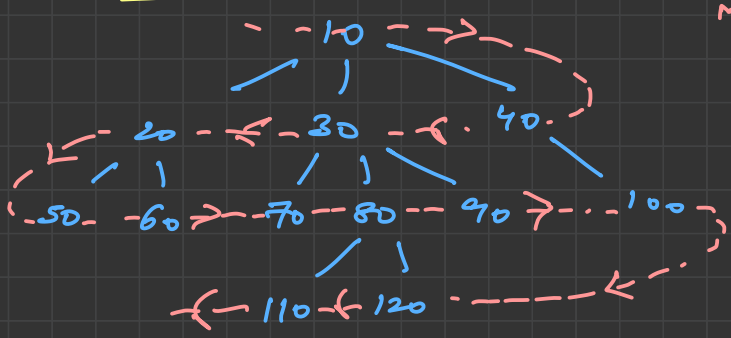
DS ArrayDeque of size Arrays.asList(root)

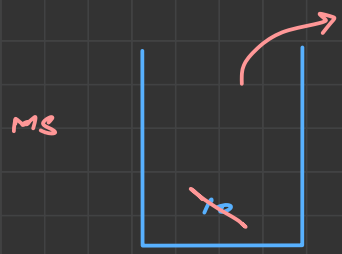
* Arrays.asList() : this list is just a wrapper that makes the array available as a list. No data is copied or created.

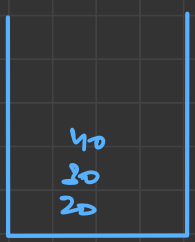
Modification we make to the single items of the list will be reflected in our original array.

(3) . offer : retrieves & removes the head of the queue.
↳ equivalent to `removeFirst()`

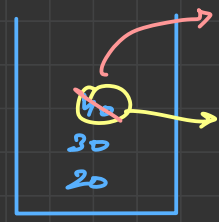
Zig - Zag explanation ↓





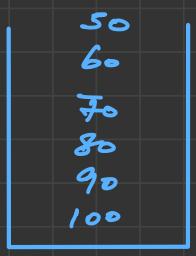


after every level, the child stack is the main stack

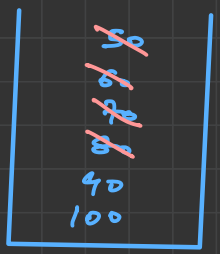


first, children of 40 will be stored, then 30 & then 20

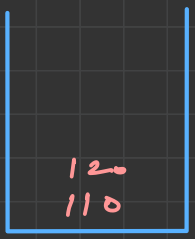
10
40
30
20



Now, child stack will be the main stack as the level will be changed.



MS



CS

If no child of nodes of the main class elements, has children, instead of transferring to CS, it will directly be printed.

10
40
30
20
50

10
20
30
40
50
60
70
80
90
100

After level = 3, the remaining child nodes will be returned that are stored in CS.

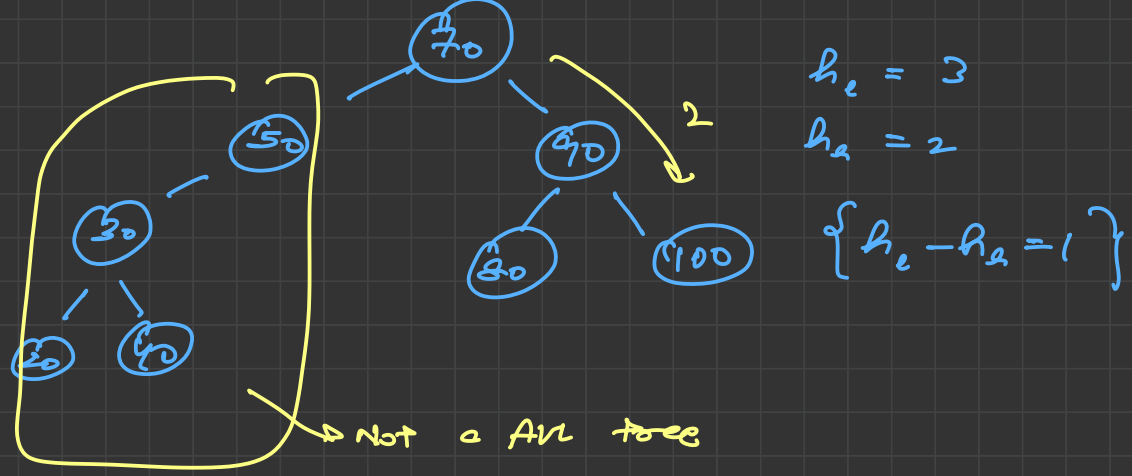
10 20 30 40 50 60 70 80 90 100 110 120

→ the children should be added in the sequence of addition of parent node.

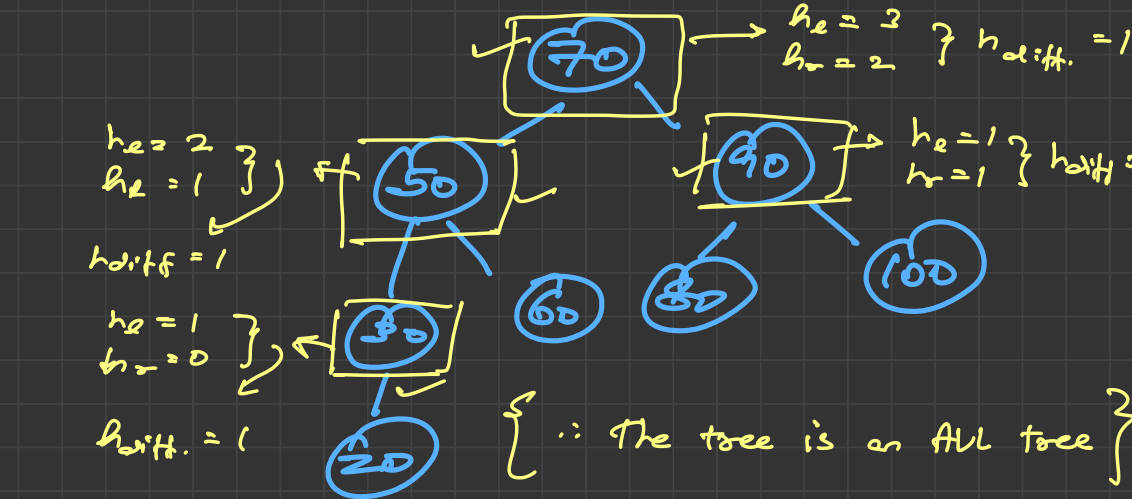
If the parent nodes are being added from L → R, then children nodes should be added L → R

AVL TREE

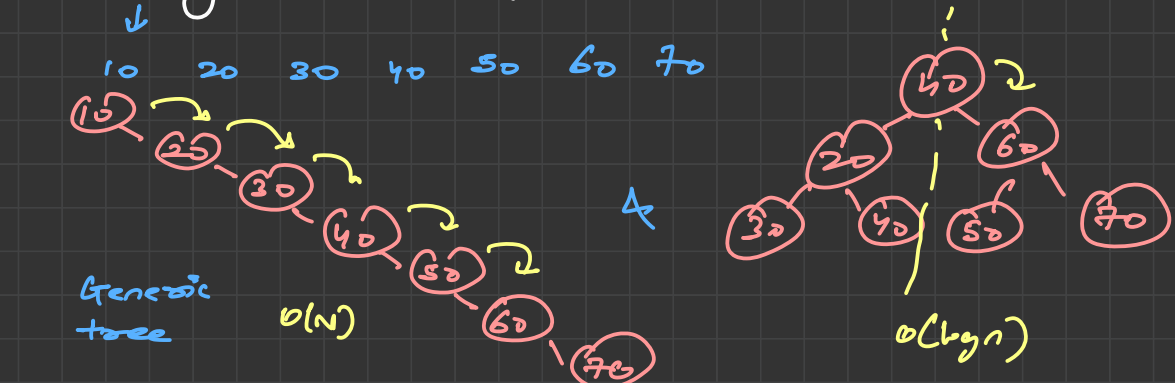
1. An AVL tree is a self-balancing BST where the difference between the heights of left & right subtrees cant be more than one for all nodes. i.e. $|h_L - h_R| \leq 1$



2. to make it an AVL tree, we've to rebalance the tree to restore the AVL tree, the process is called rotation.



3. Why do we need AVL tree?



In case of an AVL tree, we traverse only at one side of the tree that saves time.

3. Creation, insertion & deletion is similar to BST.

{ TC → O(1)
SC → O(1) }

4. All the traversals are same. ⇒ T.C → O(n)
SC → O(n)

5. Code for AVL tree :

```
class Node {
    8 usages
    int key, height;
    17 usages
    Node left, right;

    1 usage
    Node(int d) {
        key = d;
        height = 1;
    }
}
```

→ at start, height is set to 1

{ key = val of node }

6. Insertion in an AVL tree :

```
Node insert(Node node, int key) {
    if (node == null)
        return (new Node(key));

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else // Duplicate keys not allowed
        return node;

    node.height = 1 + max(height(node.left),
            height(node.right));
    int balance = getBalance(node);

    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);
```

→ O : if only node with no value, then return node.val

→ Same traversal as did in BST

→ updating height of the ancestor node

→ balance > 1 : $h_l > h_r$ & node.val < node.left.val
∴ right rotate

{ $h_r > h_l$
∴ leftRotate }

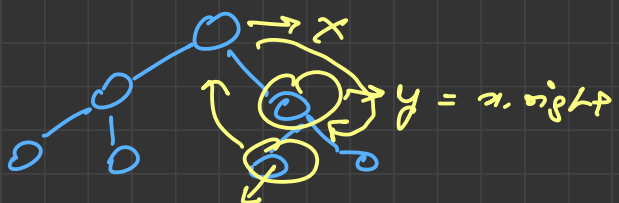
getBalance() function :

```
// Get Balance factor of node N
1 usage
int getBalance(Node N) {
    if (N == null)
        return 0;

    return height(N.left) - height(N.right);
}
```

As the defination of the AVL tree, $h = h_l - h_r$

Left rotate function

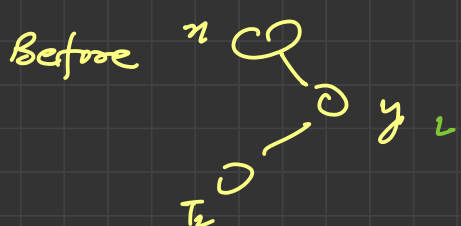
```
Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = max(height(x.left), height(x.right)) + 1;
    y.height = max(height(y.left), height(y.right)) + 1;

    // Return new root
    return y;
}
```



$T_2 = y.left$

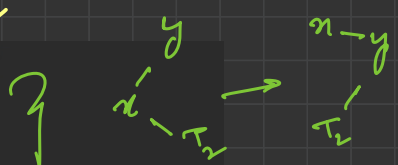
$x.right = T_2$

Before



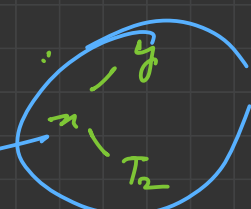
Right rotate function

```
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(height(x.left), height(x.right)) + 1;

    // Return new root
    return x;
}
```

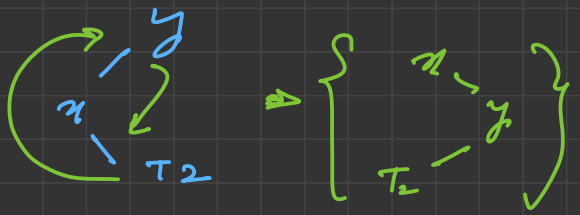


$y.left = x$:

$x.right = T_2$:

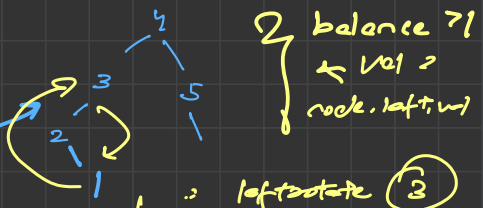


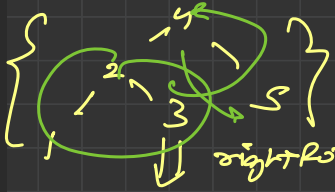
After left Rotate

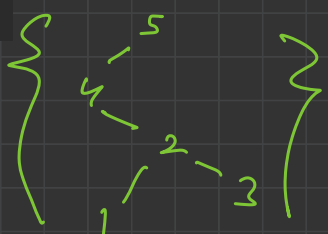


: Right rotate

```
// Left Right Case
if (balance > 1 && key > node.left.key) {
    node.left = leftRotate(node.left);
    return rightRotate(node);
}

// Right Left Case
if (balance < -1 && key < node.right.key) {
    node.right = rightRotate(node.right);
    return leftRotate(node);
}
```







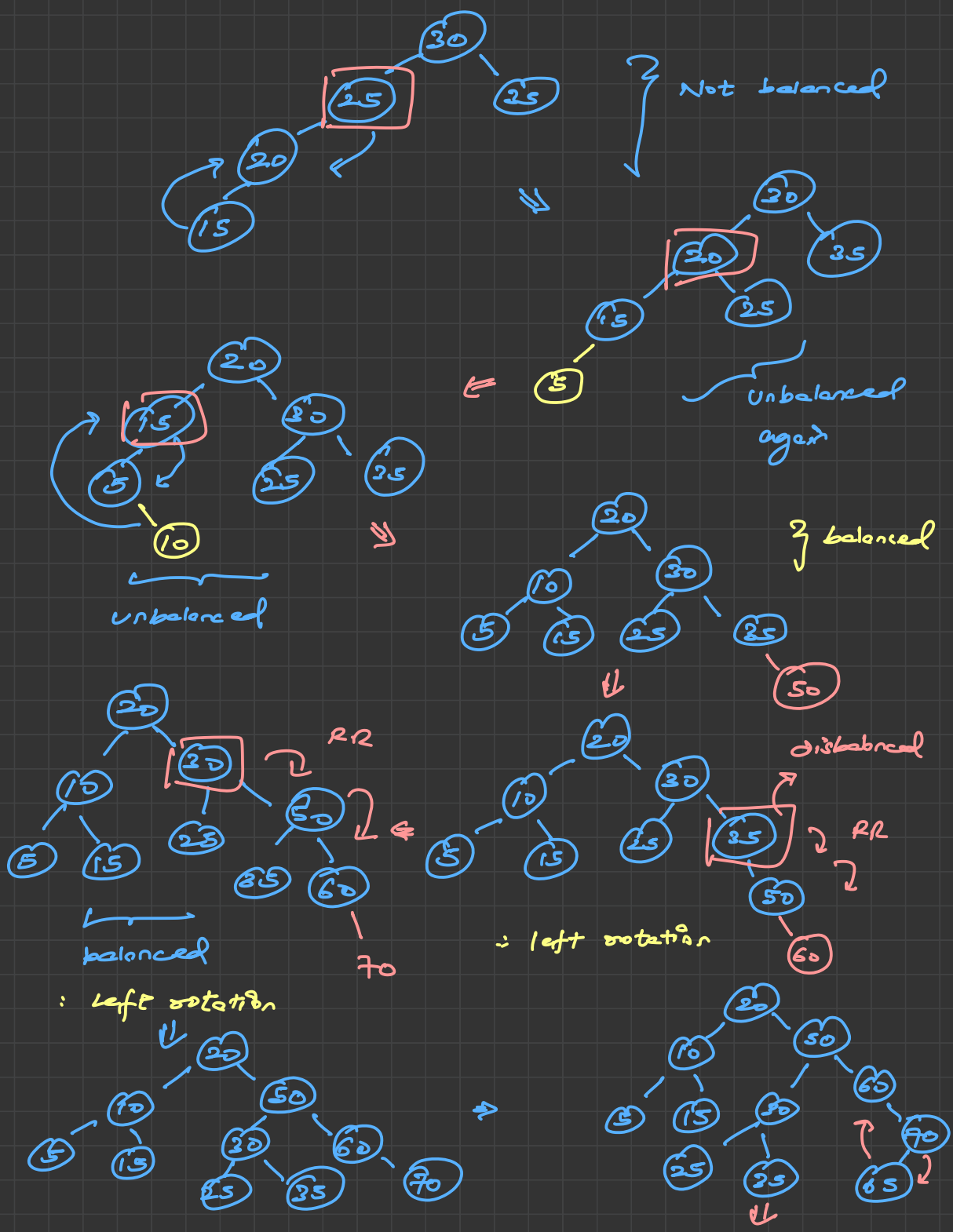
* Insert a node in AVL → Case-1 : rotation is not required

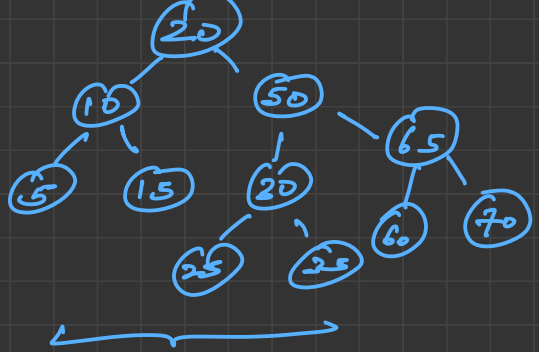
→ Case-2 : rotation is required

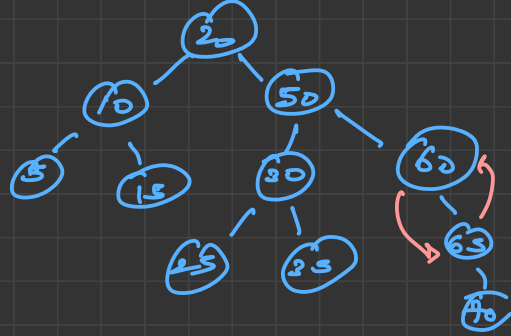
30, 25, 35, 20, 15, 5, 10, 50, 60, 70, 65

Soln → 30 → as root node in the beginning

→ After every addition, we've to check the balance of AVL tree.


30
25
25
20
15
Not balanced
20
20
35
15
25
5
Unbalanced again
20
15
30
5
25
25
10
unbalanced
20
10
30
5
15
25
35
50
balanced
20
10
20
50
5
15
25
35
60
70
RR
balanced
: left rotation
20
10
30
5
15
25
35
50
60
disbalanced
RR
: left rotation
20
10
50
5
15
20
60
15
35
70
20
10
50
5
15
50
60
25
35
70
65





7. Delete a node from AVL tree

1. Case 1 : Rotation is not required
   Case 2 : Rotation is required (LL, LR, RL, RR)

2. Left - Left condition : Right rotation

   Left - right condition : Left rotation & then right rotation

   Right - right condition : Left rotation

   Right - left condition : Right rotation & then left rotation.

8. Delete an AVL tree

∴ to delete an AVL tree, delete the root node.

→ if rootNode = null, the left & right subtree become eligible for garbage collection.

→ TC → O(1) & SC → O(1)

Q. Time complexity of AVL

	T.C	S.P
Create AVL	$O(1)$	$O(1)$
Insert node AVL	$O(\log N)$	$O(\log N)$
Traversal AVL	$O(N)$	$O(N)$
Search for Node AVL	$O(\log N)$	$O(\log N)$
Delete node from AVL	$O(\log N)$	$O(\log N)$
Delete entire AVL	$O(1)$	$O(1)$