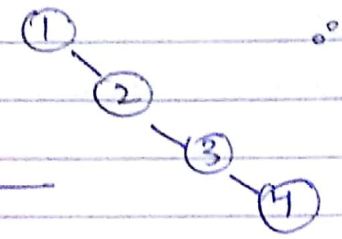
AVL Trees & Balanced trees (guaranteed)

↳ O(logN) is guaranteed → motivation
same is the case for R and B trees

eg: Const a BST from a sorted array
[1, 2, 3, 4]

unbalanced ←



∴, if we const a BST from a sorted array, we'll end up with a linked list

ye to L.L bangayi!

O(logN) reduced to O(N) ↳ takes way more time

↳ disadv. of BST

- In an AVL tree, the heights of the 2 child subtrees of any node differ by at most one.
- AVL trees are faster than R-B trees coz they are more rigidly balanced but needs more work.
- OS's relies heavily on these DS's !!
- on every insertion, we're to checks whether the tree is unbalanced or not!
- baki sab same as BST oper^n (del, max, min etc)

Balanced trees :

	Avg. Case	Worst case
Space	O(n)	O(n)
Insert	O(logn)	O(logn)
delete	O(logn)	O(logn)
search	O(logn)	O(logn)

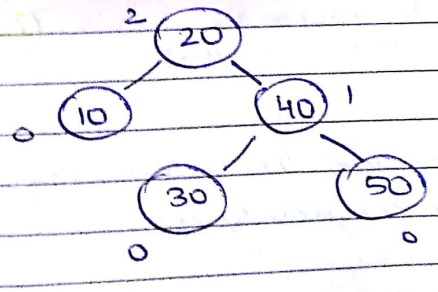
↓ guaranteed O(logn) (very predictable too)

Height :

Height of a node : length of the longest path from it to a leaf

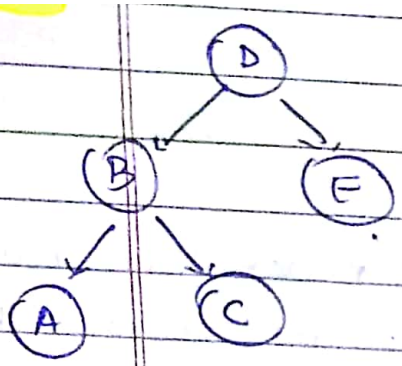
We can use recursion to calc it :

height = max (left.child.height(), right.child.height()) + 1



The leaf nodes have NULL children : we consider the height to be -1 for NULLS !!

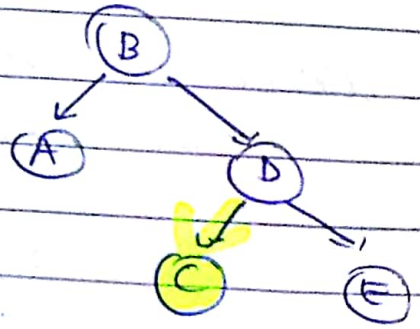
" AVL algo uses heights of nodes, we want the heights as small as possible : we store the height parameter if it gets high, we fix it



rightRotate(D)

leftRotate(B)

↳ rotation to make it balanced.



↳ ye 'C', 'D' ka left child

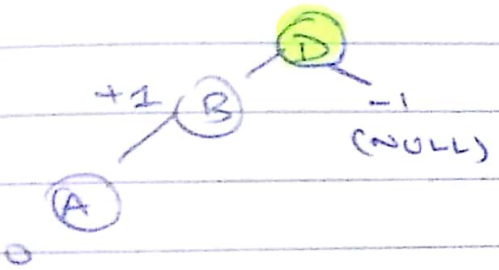
We just have to update the references which can be done in O(1) time complexity !!!

The in-order traversal is the same!

A - B - C - D - E → alphabetical ordering obtained

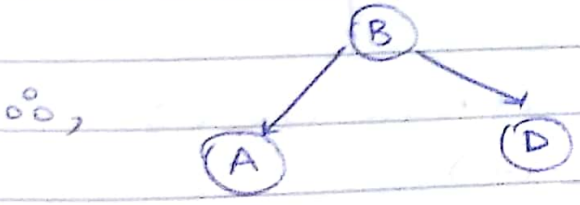
↓

implementation page no. 2 (eg used: 'c' is stored in 't')



The diff of height param is more than 1 (i.e, -> 2
1-(-1) = 2)

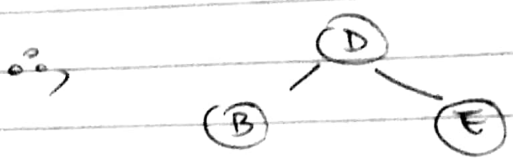
∴, rotation to the right!



∴, The new root node is the B, which was the left child of D before the rotation.

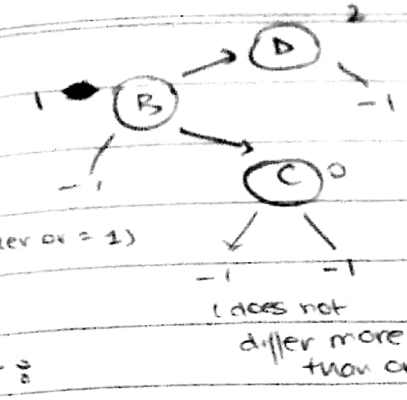
BEGIN rotateRight(Node node)

Node tempLeftNode = node.getLeftNode()
Node t = tempLeftNode.getRightNode()



BEGIN rotateLeft (Node node)

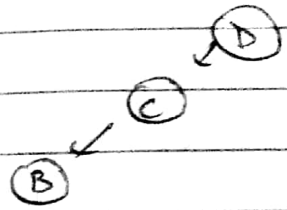
(excepts one (1))



these nodes may have left and right children but it does not matter // we do not modify the pointers for them !!

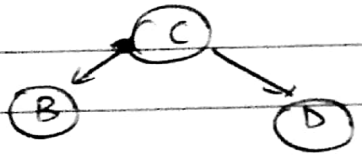
zing AVL :
op

We're to make a left rotation on node B

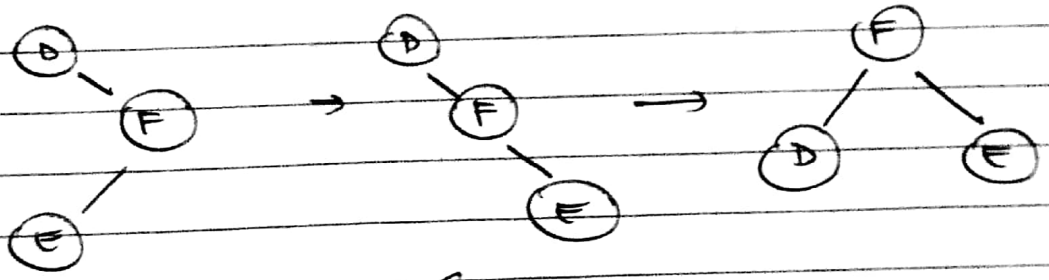


(case 1 bangaya vapas)

then right on D :

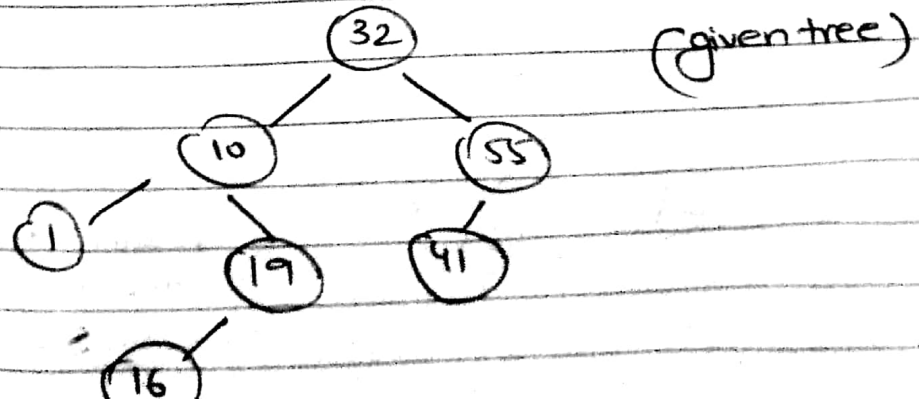


→ obtained



(similar to case '3')

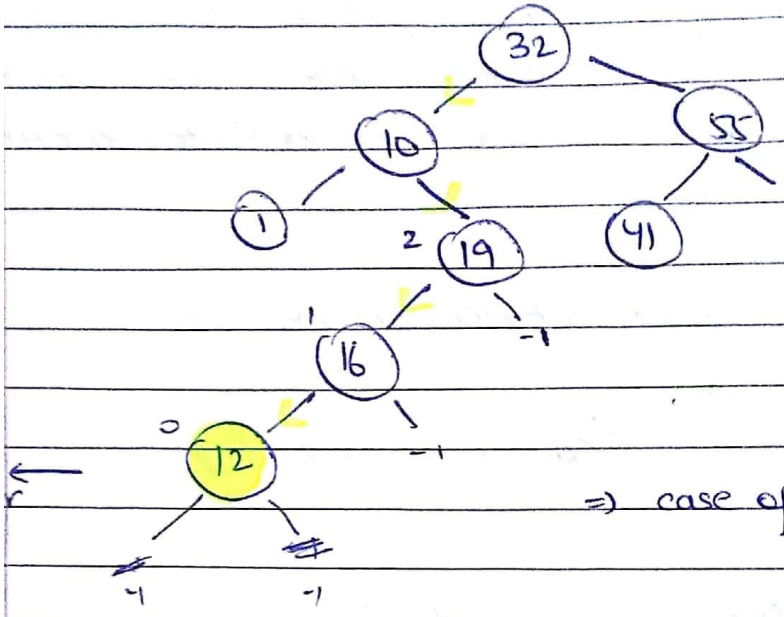
(case(2) obt.)



(given tree)

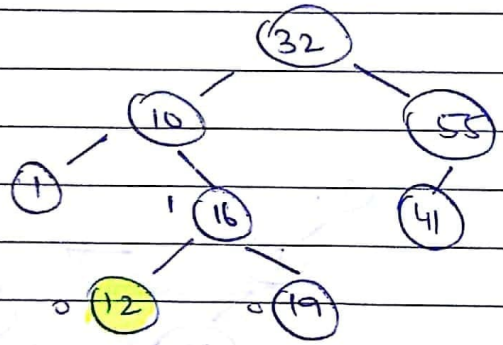
Check for unbalancy at each iteration in insertion!

balanceTree.insert(12):



⇒ case of doubly-left heavy (case 1)

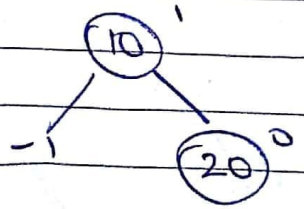
⇒



no need for rotn now!

balanceTree.insert(10)
(from a sorted array [10,20,30,40,60]

insert(10)

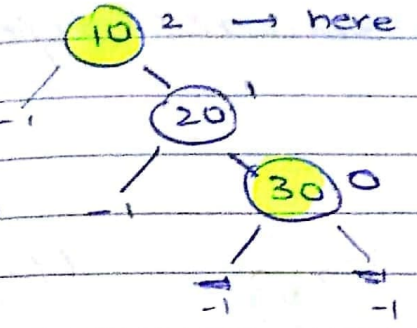


insert(20)

⇒ balance tree till now (diff ≤ 1)

1-(-1) = 2

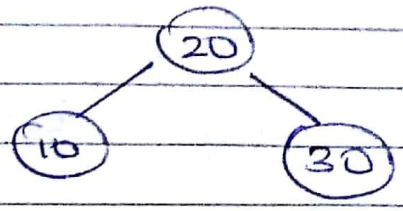
now,



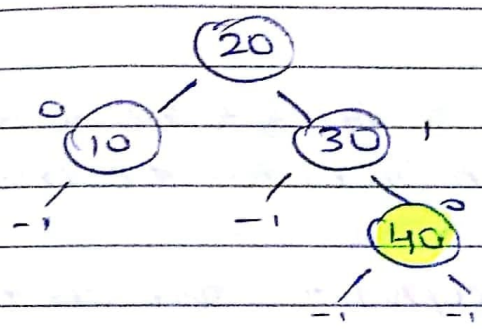
→ here diff ≥ 1 (i.e. → 2)

∴, (rotations needed) → doubly heavy right

insert(30)



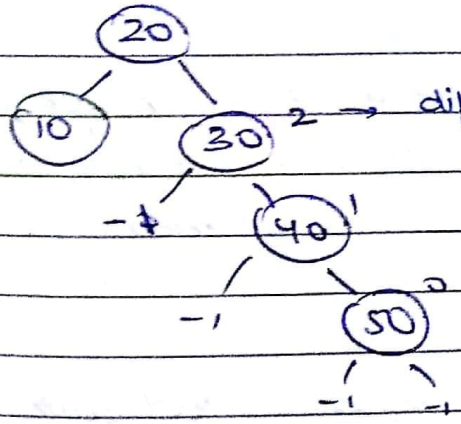
now, insert(40) :



diff ≤ 1

(no rotation) req.

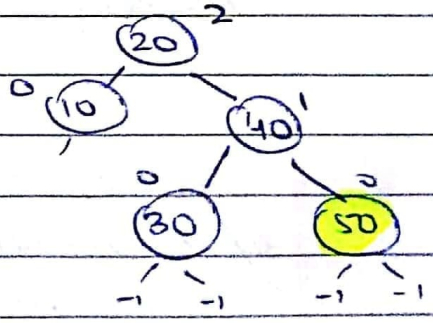
now, insert(50) :



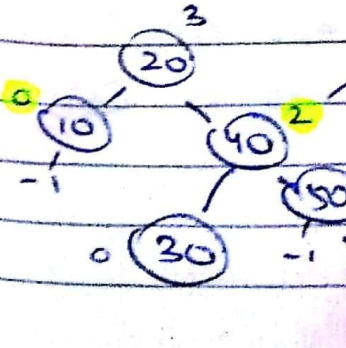
→ diff ≥ 1 (rotations needed)

case II

∴,



now insert(60) :



for root node(20)

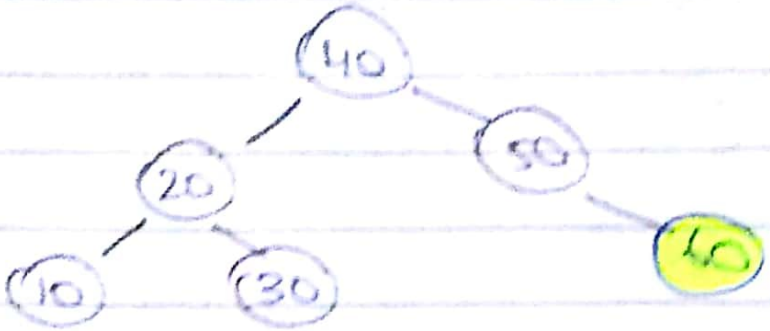
diff (>1) i.e, 2

rotations needed

→

no rotr needed (diff ≤ 1)

2., root node rotated towards left



(final tree obtained)

Applications :

AVL sort : → insert 'N' items to be sorted
→ make an 'in-order' traversal

T.C : Insertion $O(N \log N)$ + In order trav. $O(N)$

= $O(N * \log N)$ → overall

other sorts like quicksort, heap, outperforms AVL sort because q.s doesn't use any additional memory, whereas AVL uses $O(N)$ memory.

implementation :

```
class Node (object)

    def __init__ (self, data):
        self.data = data
        self.height = 0          ---> one thing which is to added in node class (height parameter)
        self.leftChild = None        node
        self.rightChild = None       longest path from it to a leaf.

class AVL (object):

    def __init__ (self):
        self.root = None

    def calcHeight (self, node):      ---> helper methods (height, balance, left right) ---> used for insertion
        if not node:                  (assign -1 to leaf)
            return -1

        return node.height

# if it means value > 1, it means it is a left heavy tree      right rotation
    def calcBalance (self, node):                                 if < -1, left rotation

        if not node:
            return 0

        return self.calcHeight(node.leftChild)  -  self.calcHeight(node.rightChild)
                                                   subtract sign
```

def rotateLeft (self, node)

(same 'left' ko 'right' karde bas)
pichle 'def' se and vise-versa

def insertNode ( ) method to be written often
def __init__ ( )

```
def insertNode (self, data, node) :

    if not node :
        return Node(data)

    if data < node.data :
        node.leftChild = self.insertNode(data, node.leftChild)

    else :
        node.rightChild = - - - - - - - - - (..., ..rightChild)

    node.height = max(self.calcHeight(node.leftChild),
                      self. - - - (rightChild)) + 1

    return self.settleViolation(data, node)


def settleViolation (self, data, node) :

    balance = self.calcBalance(node) :
    # case 1 -> doubly left
    if balance > 1 and data < node.leftChild.data :
        print("Left Left heavy situation")
        return self.rotateRight(node)

    # case2 -> doubly right
    if balance < -1 and data > node.rightChild.data :
        print("Right Right heavy . . . . .")
        return self.rotateLeft(node)

    if balance > and - - - -> left - - -
        node.leftChild = self.rotateLeft(node.leftChild)
        return self.rotate(node)
```

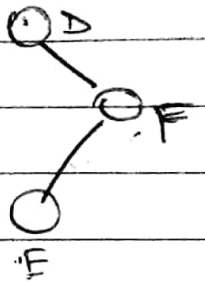
if <-1 - - - - - < right- :

print ("Right-Right - - -)

node.rightChild = self.rotateRight(node.right

return self.rotateLeft()

↳ eg:



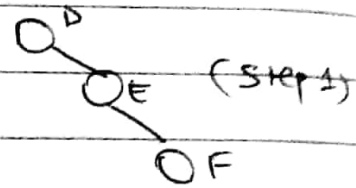
2 rotations req →

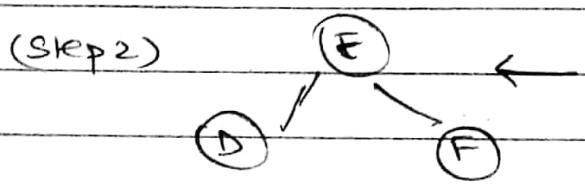
① → right on F ∴, R-R obtain

② → left on D

&

∴,





↳ return node

---

we forgot to implement traverse() :

↳ write after calc ~~height~~ balance()

```
def traverse(self):
    if self.root:
        self.traverseInOrder(self.root)

def traverseInOrder(self, node):
    if node.leftChild:
        self.traverseInorder(node.leftChild)

print("%s" % node.data)
```

Testing:

```
avl = AVL()
avl.insert(5)
avl.insert(4)
avl.insert(3)

avl.traverse()
```

Output: Left Left heavy situation...
Rotating to the right on node 5

```
3
4
5
```

eg: Input: (5)
(7)
(6)

Output: Right Left heavy sit"
Rotating to the right on 7
left on 5

```
5
6
7
```

- Check the removal method from video / codes
  - similar to BST (don't avoid)