## Defn of an AVL tree :-

- An empty binary tree B is an AVL tree.
- If B is a non-empty binary tree with $B_L$ and $B_R$ as its left and right subtrees then B is an AVL tree, if and only if

  a) $B_L$ and $B_R$ are AVL trees and

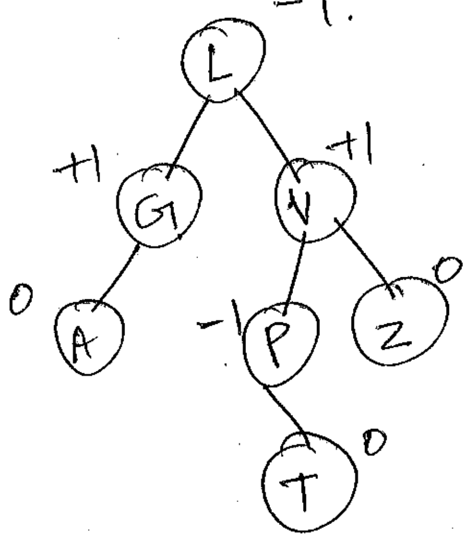
  b) $|h_L - h_R| \leq 1$.

  where $h_L$ and $h_R$ are the heights of $B_L$ and $B_R$ respectively.

## Balance Factor :-

- Each node has a balance factor = height of its LST − height of its RST.
- balance factors of nodes in a balanced tree are −1, 0 or 1.

Ex :




- If Balance factor of any node is not -1, 0 or +1, then it is not an AVL tree.

## Representation of an AVL tree :-

Each node :

| | | | |
|---|---|---|---|
| Left | info | BF | Right |

## Searching in an AVL tree :-

||lar to search in a BST.

## Insertion in an AVL tree

- insert after finding an appropriate position as in BST.
- needs height balancing - rotations

1) if insertion is into an initially empty AVL tree,

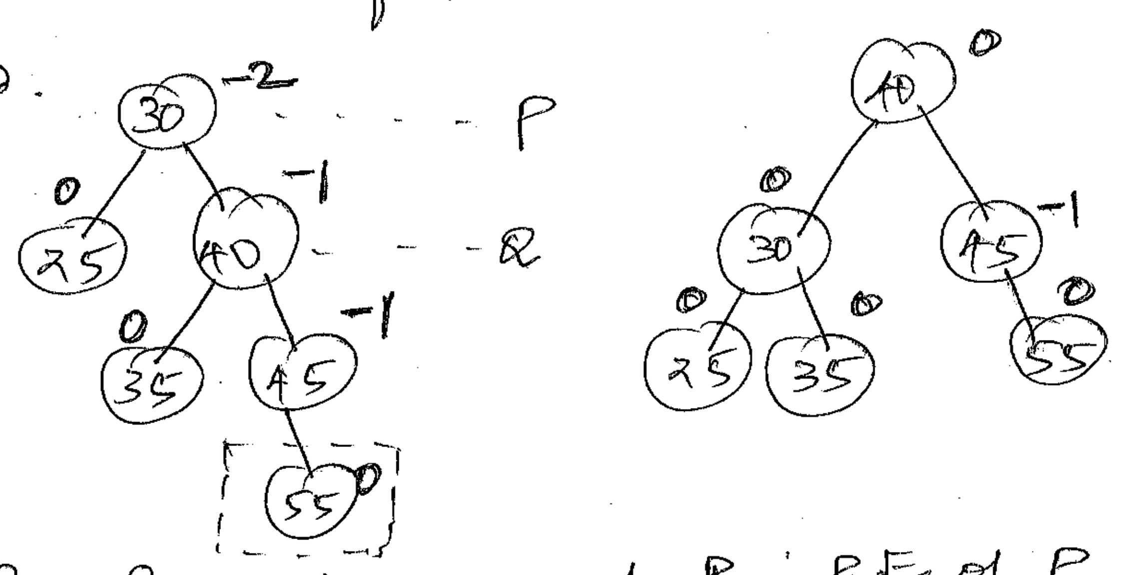
- inserted node → root node
- tree is height balanced.

2) if tree has only root node before insertion, new node may be inserted as left / right child of root depending on its value. Tree is height balanced.

3) Needs height balancing

Inserting a node with key 'K' increases the height of the RST of the root:-

a) height of the right subtree of the right subtree of the root node is increased.




Rotate Q about its parent P, ∴ BFs of P

and α becomes zero.

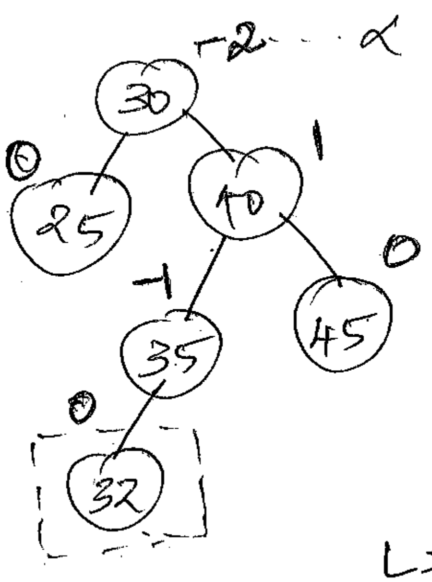
- If α is the node to be height balanced, ~~inserting~~ inserted a node is in the right subtree of rightchild of α.

RST( rightchild of α)

- RR rotation - single rotation

b) inserting a node in the ~~RST( leftchild of root)~~ LST( rightchild of root)

Ex:



LST( rightchild of α)

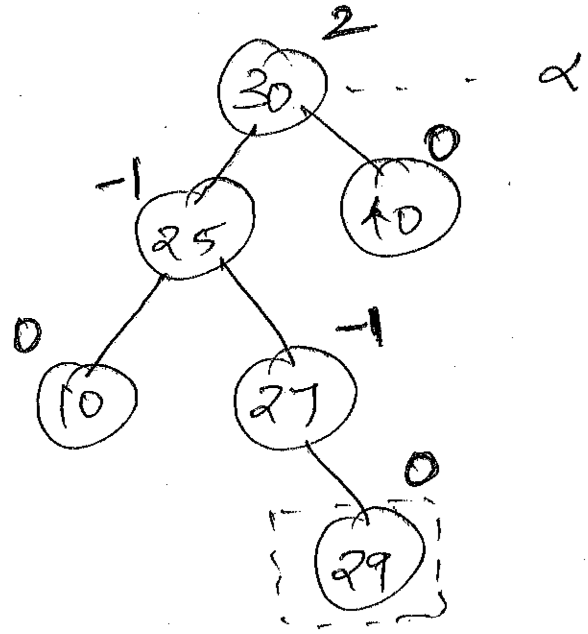
RL rotation

= double rotation

4) inserting a node with key 'K',
increases the ht of the LST of root.

a) ht of the RST (lchild of root) is
increased"

Ex:



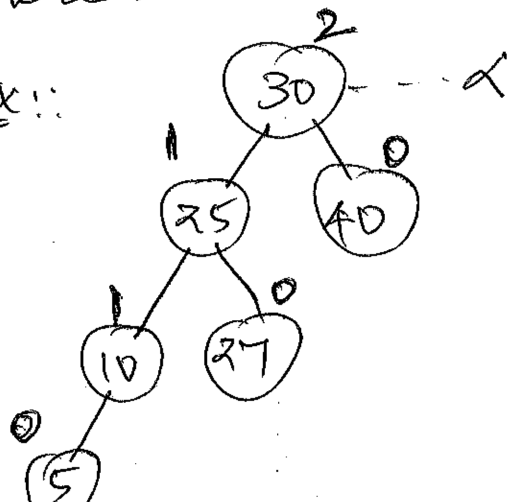
= RST( lchild of α)

= needs "LR
rotation

= double rotation

b) ht of the LST( lchild of root) is
increased.

Ex::



LST( lchild of α)

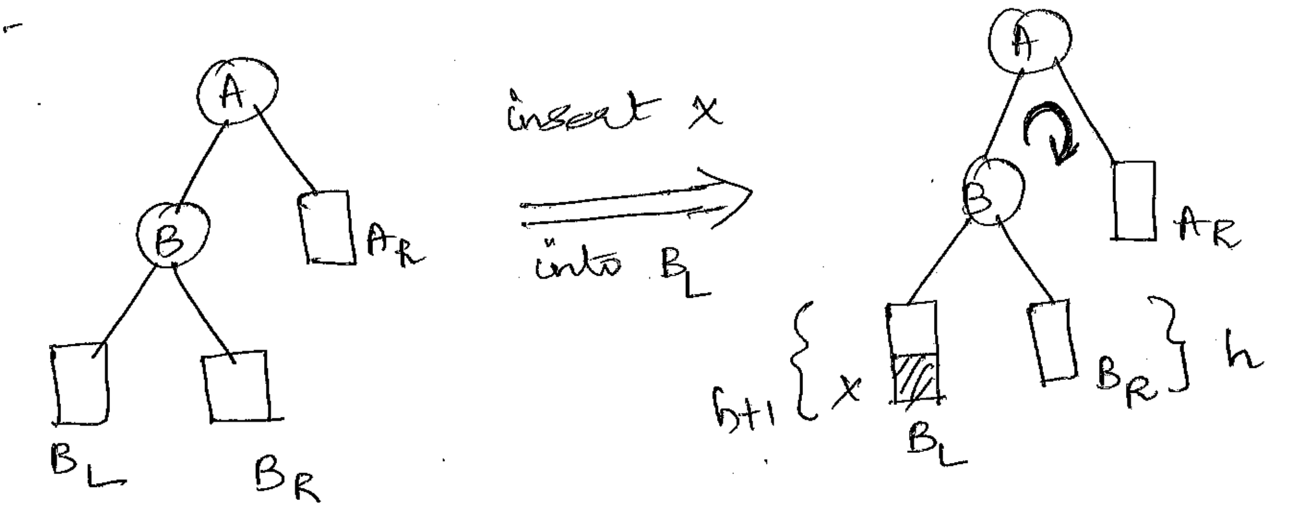
= needs single LL
rotation

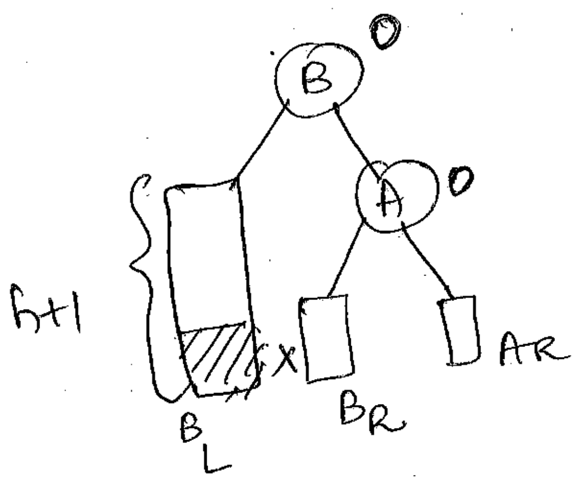
## Single and double rotations

a) **LL rotation:-**

new node is inserted into the LST (lchild of node A), whose balance BF becomes +2 after insertion.

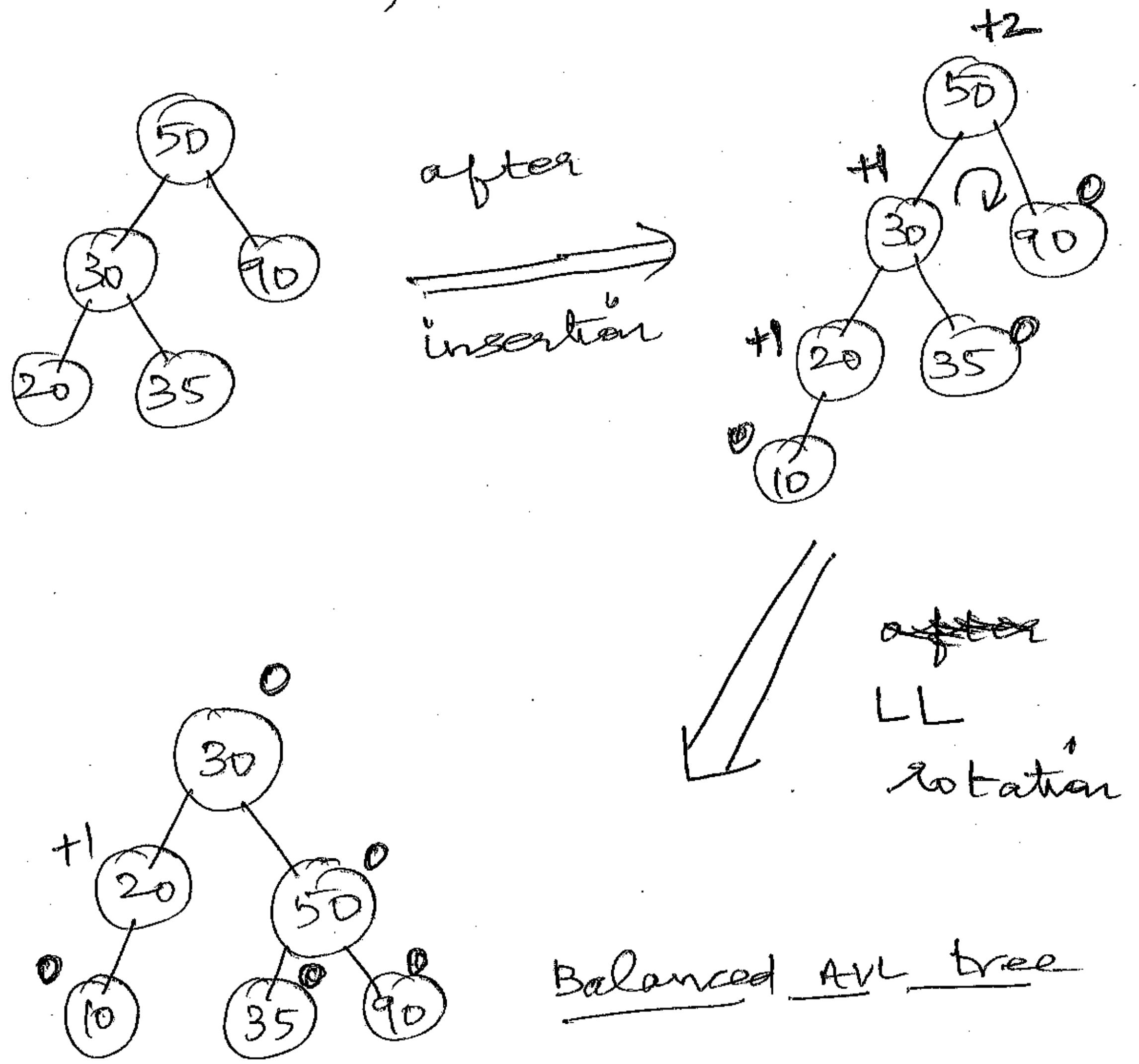
BI:-




To rebalance:-
1) Make B the root
2) Left(A) ← Right(B)
3) Right(B) ← A

Ex: insert node 10, in



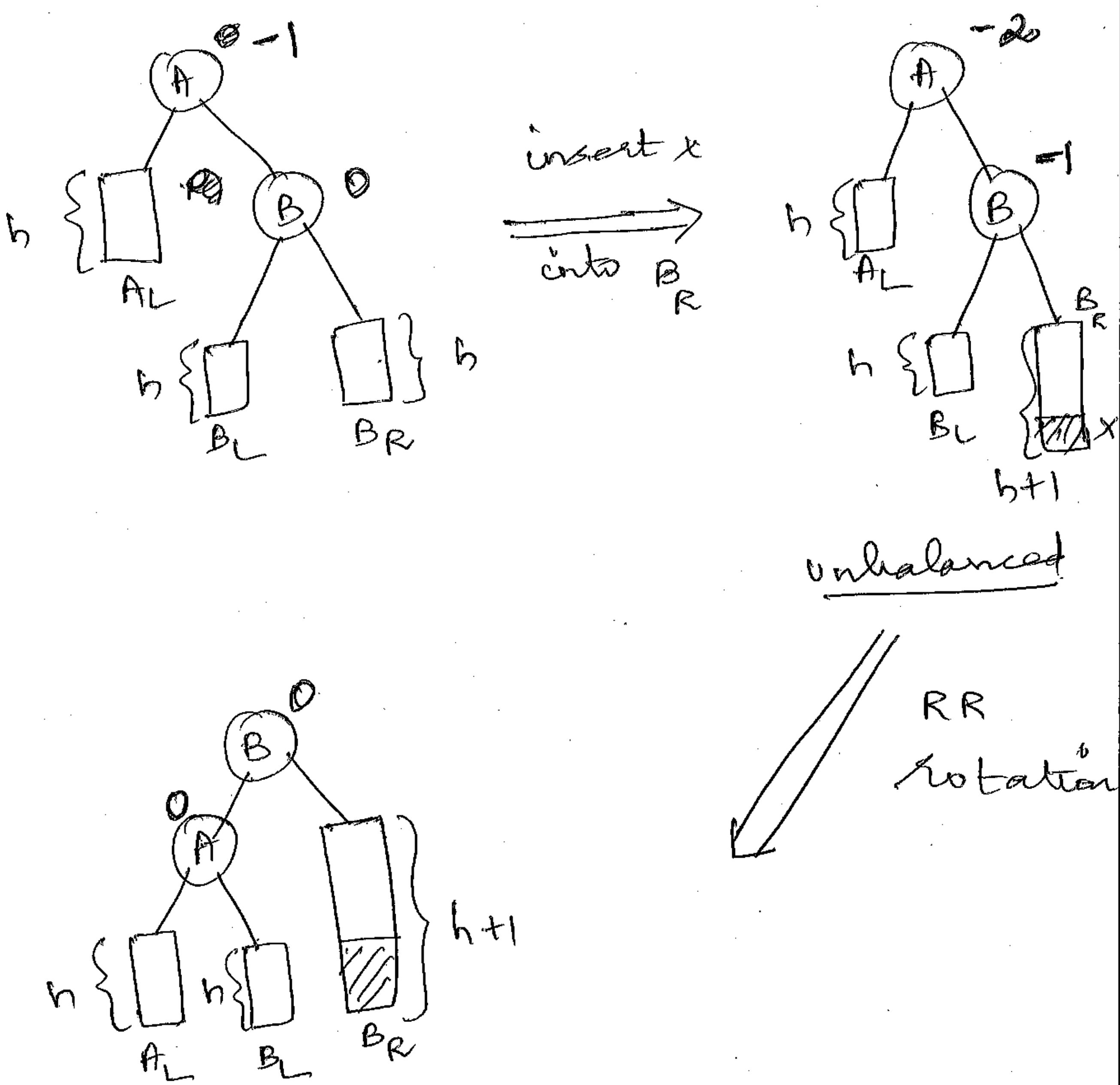

b) RR rotation :-

new node is inserted in the RST of right child of A — which needs to be re-balanced.

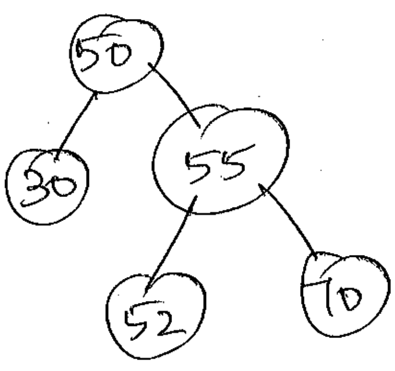
To rebalance :-

1) Make B as the root node
2) Right(A) ← left(B)
3) left(B) ← A

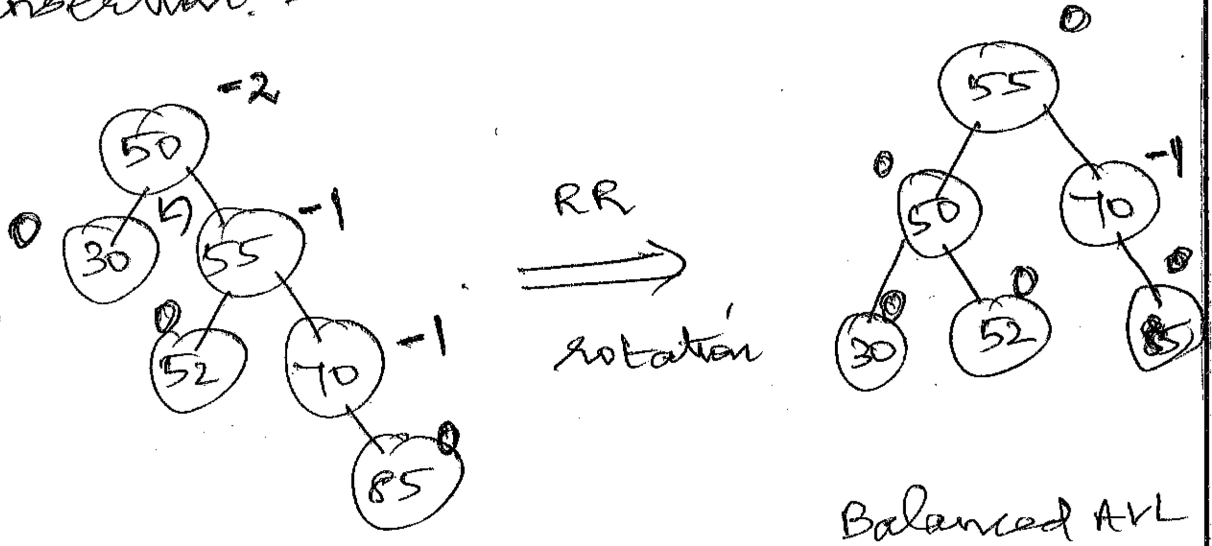
Eg:




Ex: insert node 85 in:



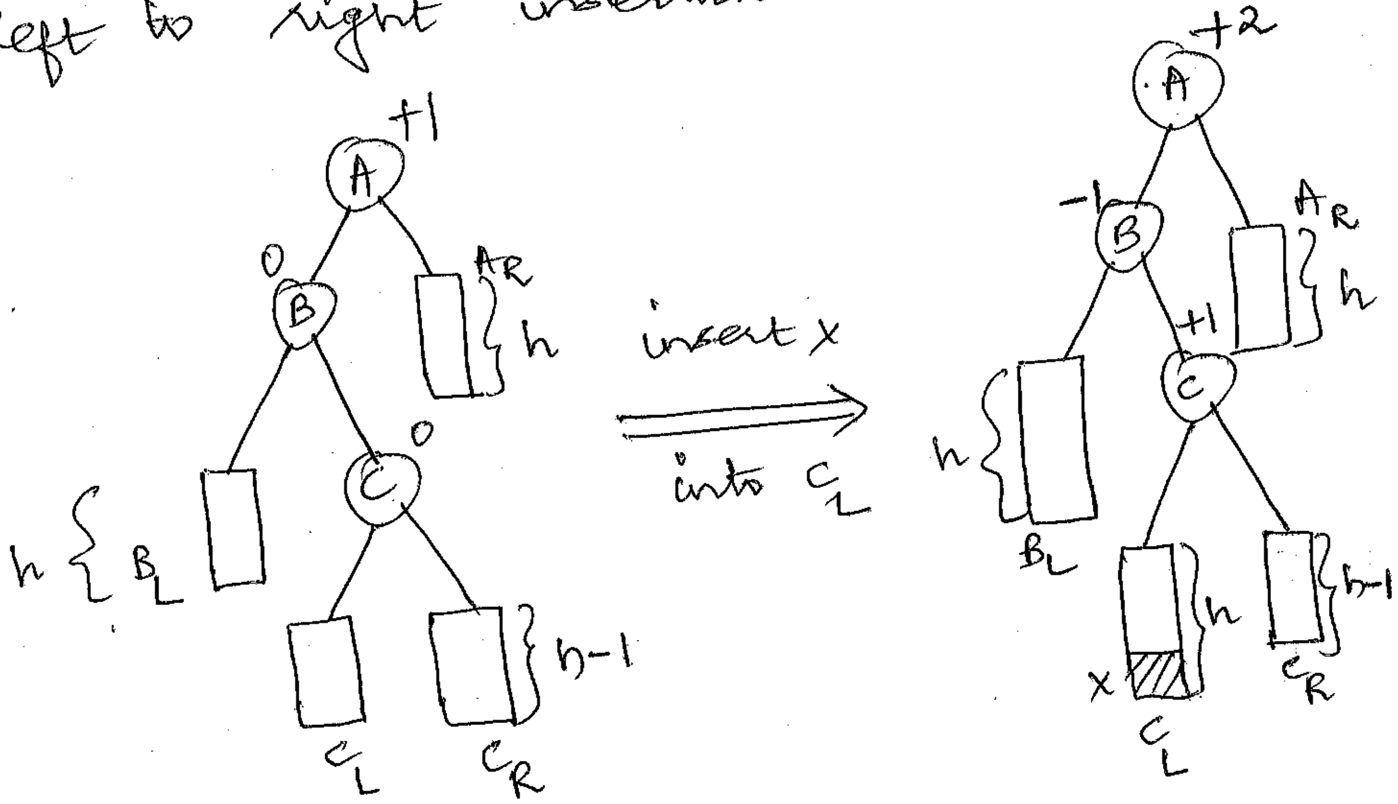
After insertion:-




Balanced AVL tree

c) <u>LR rotation</u>:-

unbalance due to insertion is the RST of left child of the root, node — left to right insertion.

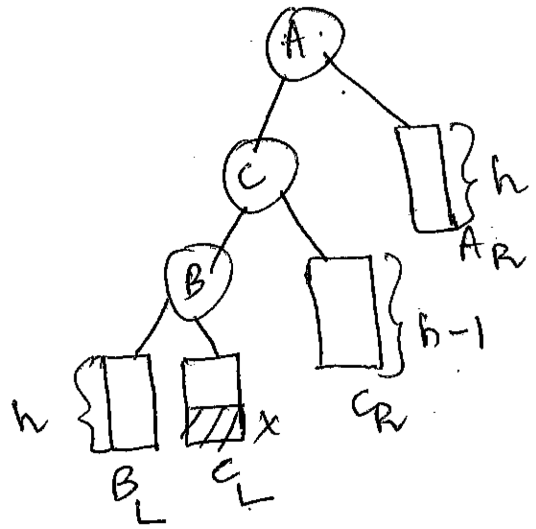


This needs two rotations to manipulate pointers.

<u>Rotation 1</u>:- The left subtree $(C_L)$ of the right child (C) of the left child (B) of the pivot/unbalanced node (A) becomes the right subtree of the left child (B).

The left child (B) of the pivot node (A) becomes the left child of C ie,

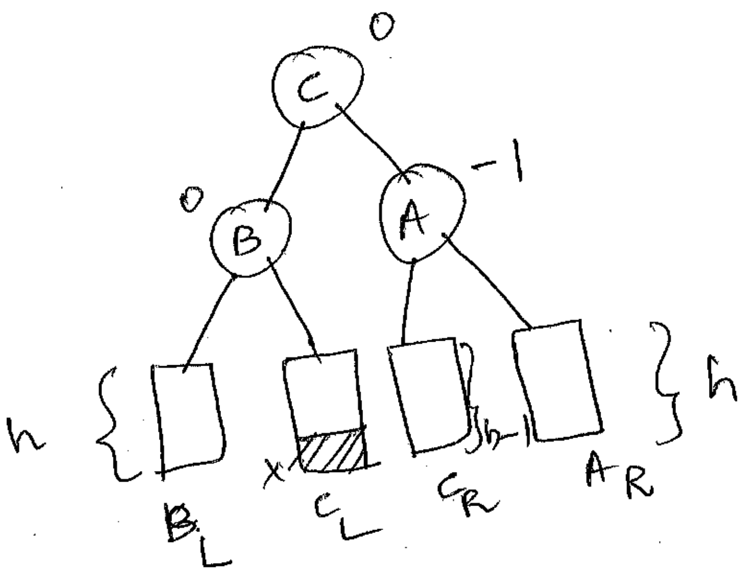
<u>RR</u> <u>rotation</u>.




<u>Rotation 2</u>:- The right subtree $(C_R)$ of the left child (C) of the left child (B) of the pivot node (A) becomes the left subtree of A.

A becomes the right child of C.

This is LL rotation.



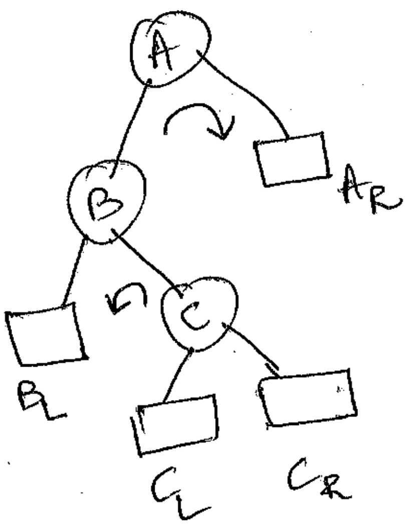

Pointer movements are :-

Right(B) ← left(C)

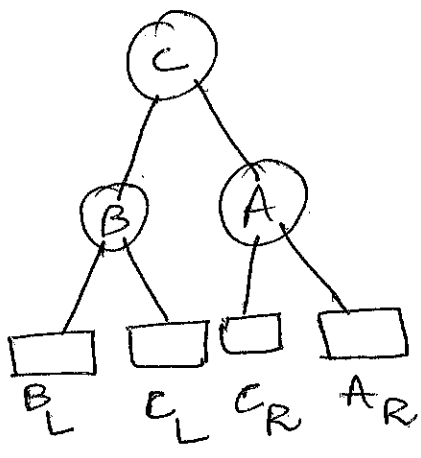
left(A) ← right(C)

left(C) ← B

right(C) ← A.

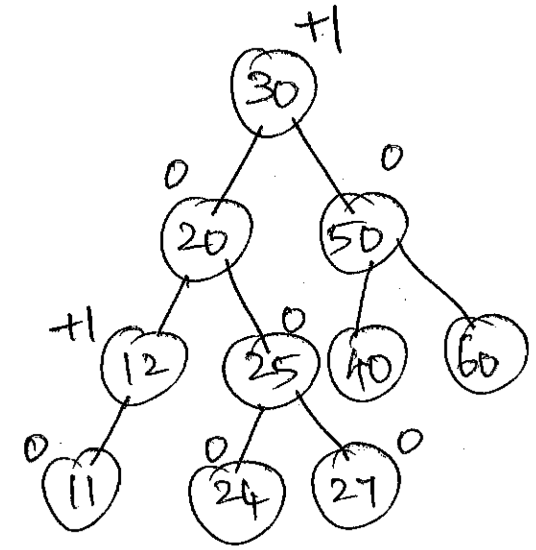



LR
rotation



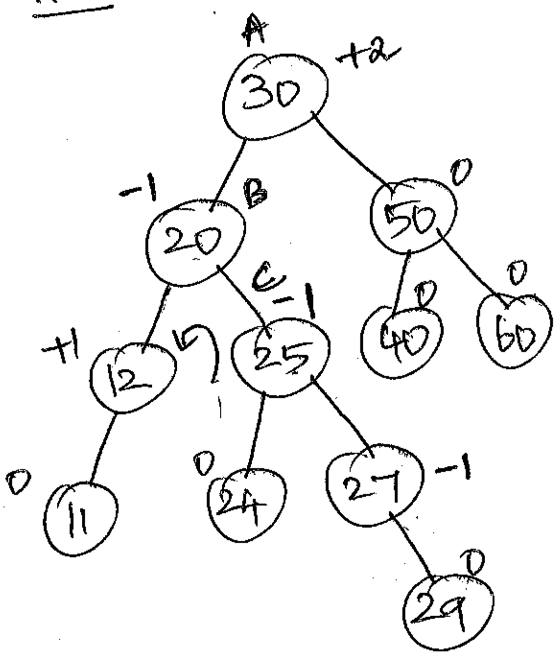

insert( RST( lchild of root))

Ex:

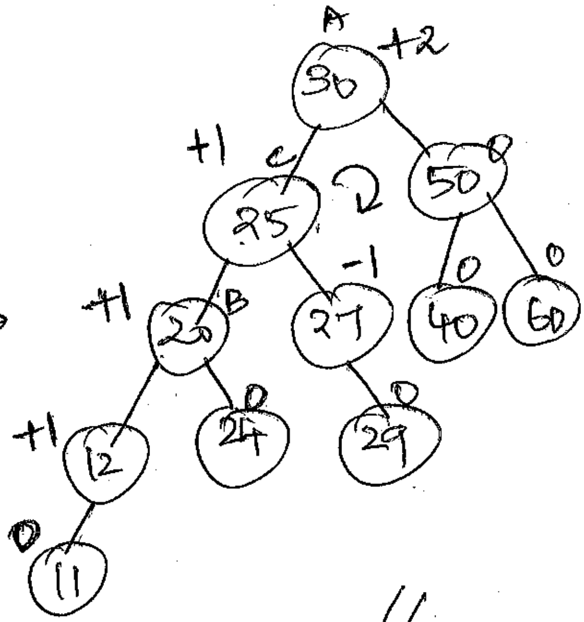


insert node 29

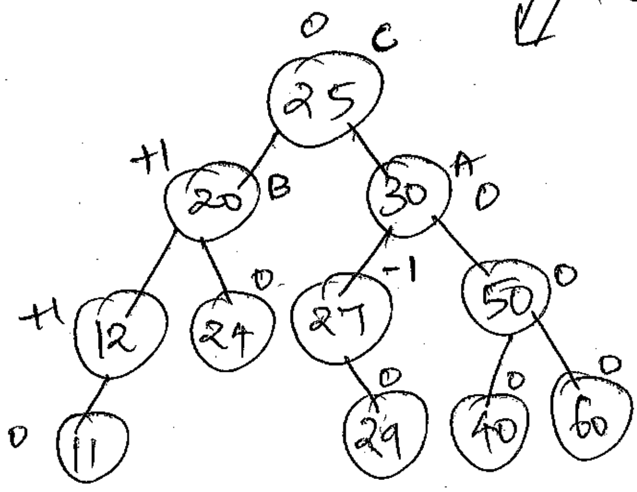
A I:



RR rotn



LL rotn



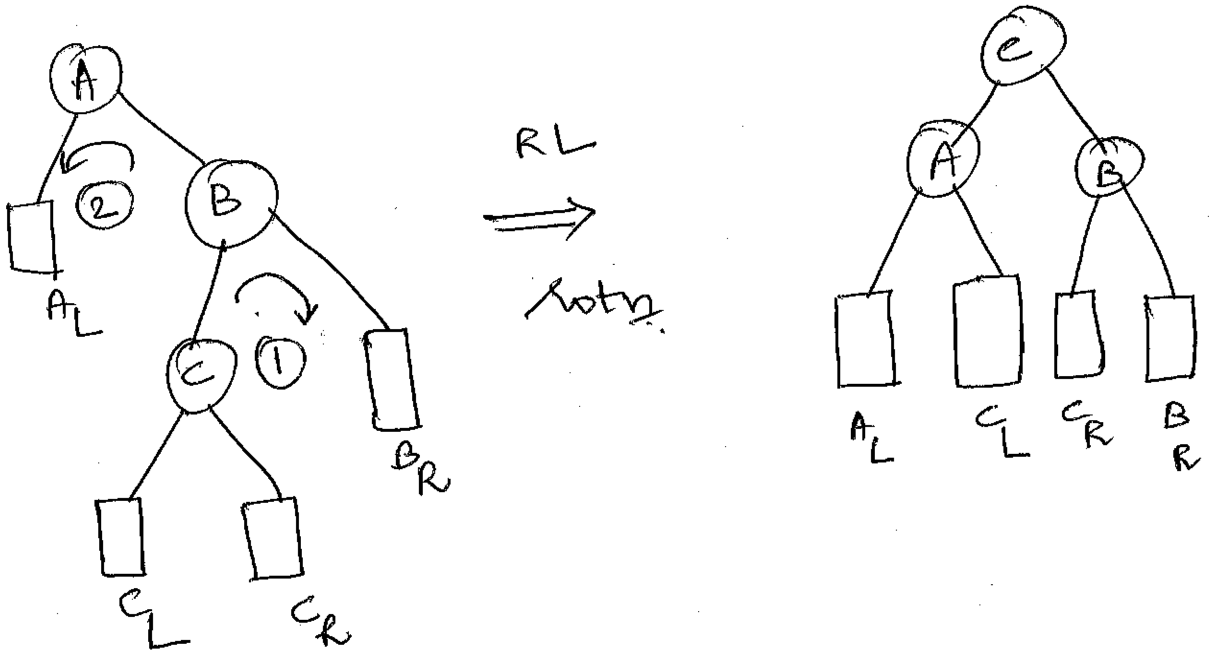
$\therefore$ LR = RR + LL

d) RL Rotation :-

unbalanced - due to insertion in the LST of the right child of the root (pivot) node.

- This is known as right to left insertion.

- RL rotn. is the mirror image of LR-rotn.




Rotn 1: The right subtree ($C_R$) of the left child (C) of the right child (B) of root node A becomes the left subtree of B and the right child (B) of the root node becomes the right child of C.

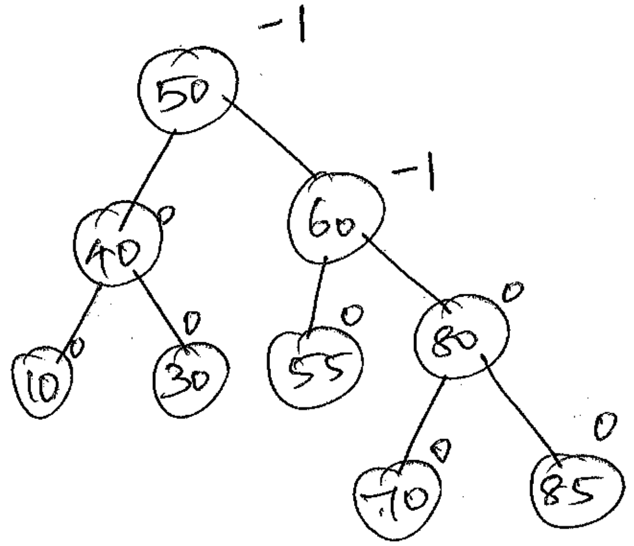
This is LL rotn.

Rotn 2: The left subtree ($C_L$) of the left child (C) of the right child (B) of the root/pivot node becomes the right subtree of A.

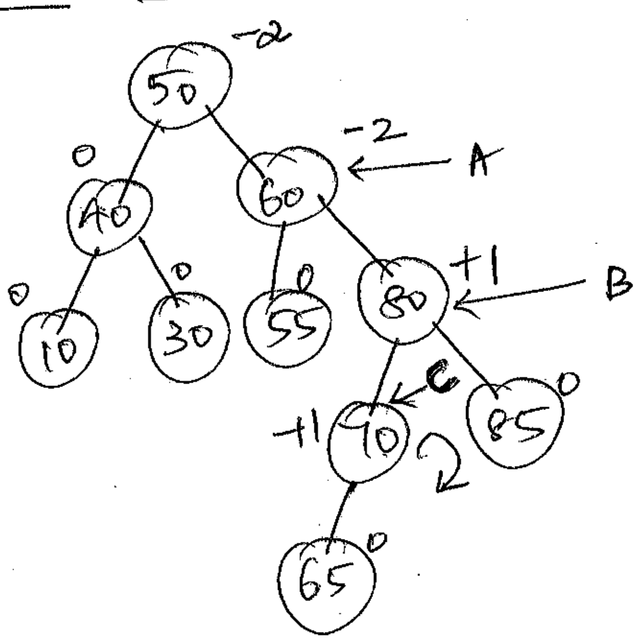
This is RR rotn.

RL = LL + RR

Ex:

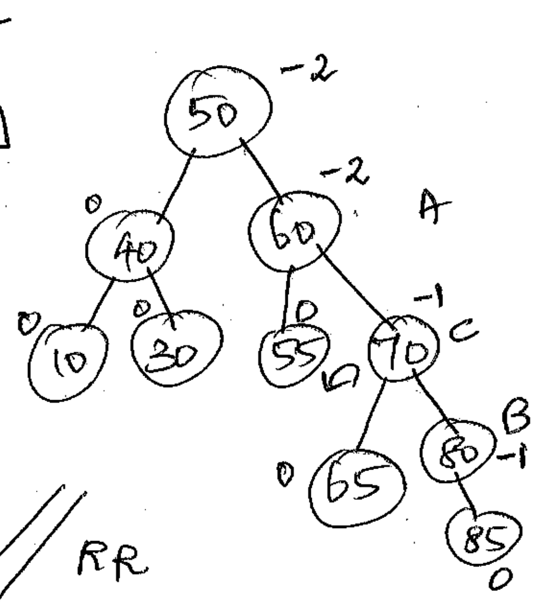


Insert 65 :-

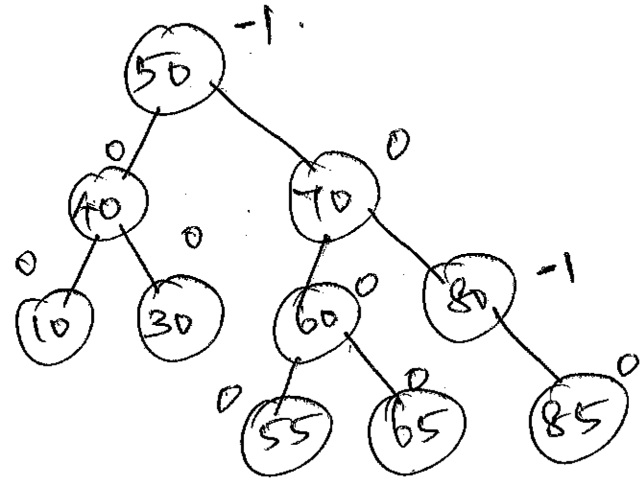


LST (R child of A) = RL rotn

LL



RR



Summary :-

∴ if pivot is the node to be height balanced, $\alpha$

insertion is in LST (lchild of $\alpha$) — LL rotn

" " " RST (rchild of $\alpha$) — RR rotn

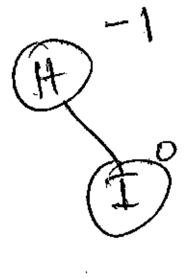
" " " LST (rchild of $\alpha$) — RL rotn

" " " RST (lchild of $\alpha$) — LR rotn

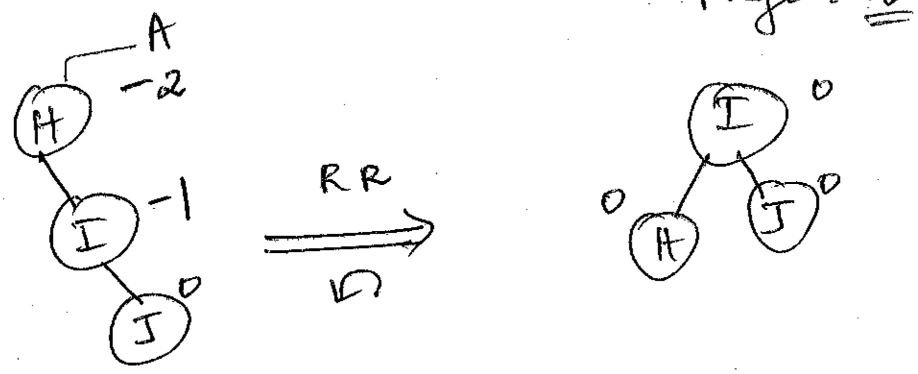
Ex: Create an AVL search tree from the given set of values :-

H, I, J, B, A, E, C, F, D.

Insert H  No rebalancing

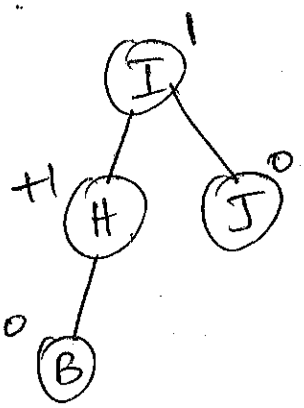
Insert I  No rebalancing

Insert J:



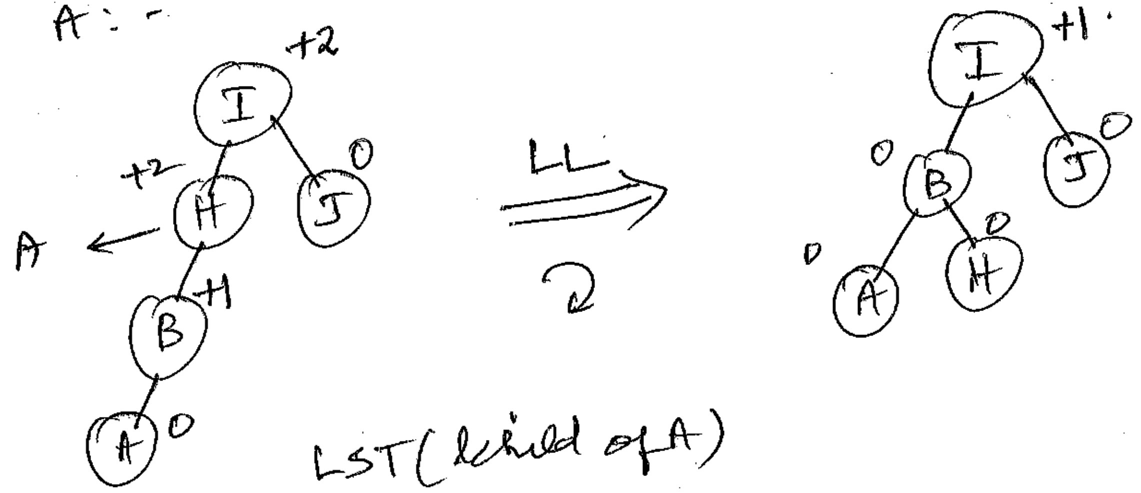

RST (Rchild of A) : RR

Insert B:-



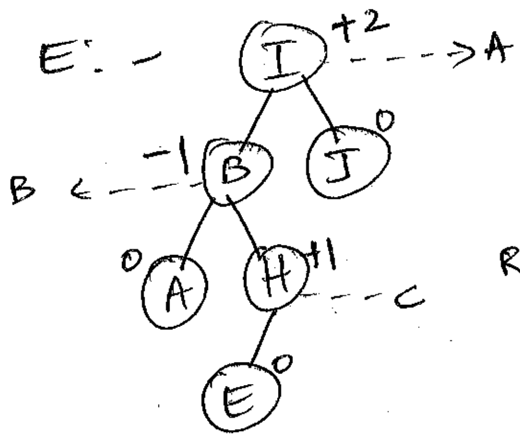

No rebalancing

Insert A:-




LST (Lchild of A)

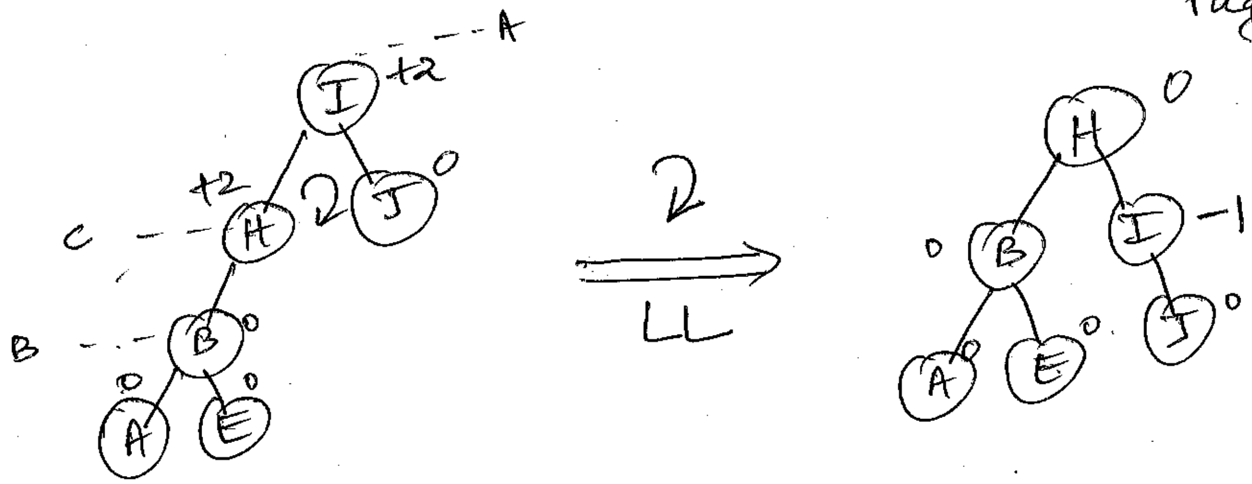
Insert E:-



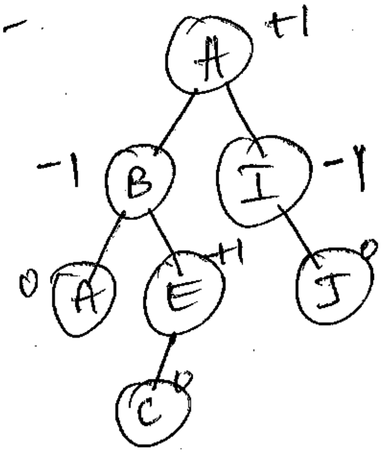

RST (Lchild of A)

LR

RR + LL

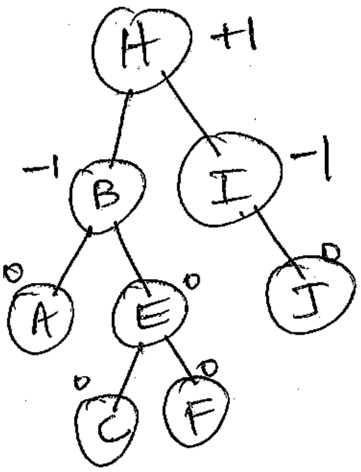



Insert C :-




No rebalancing

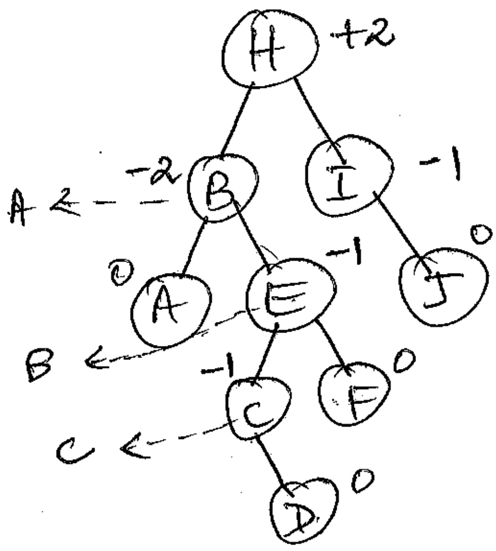
Insert F :-




No rebalancing

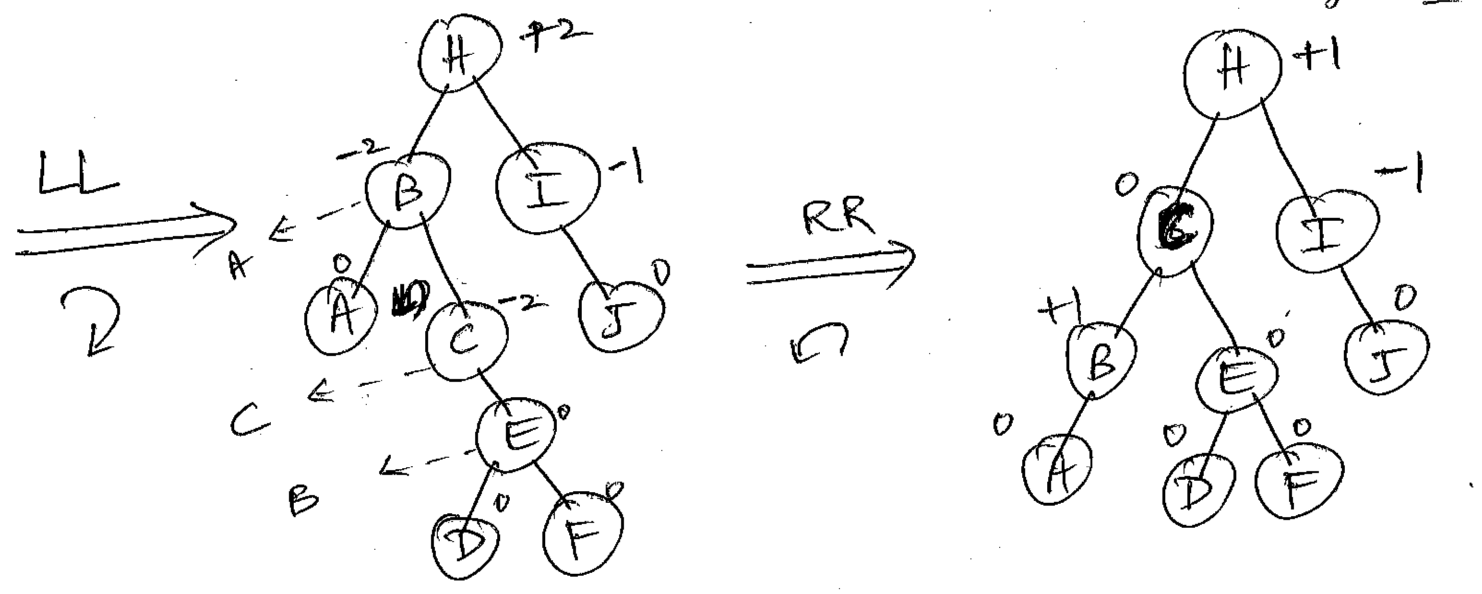
Insert D :-

LST (rchild of A)




RL rotn

LL + RR

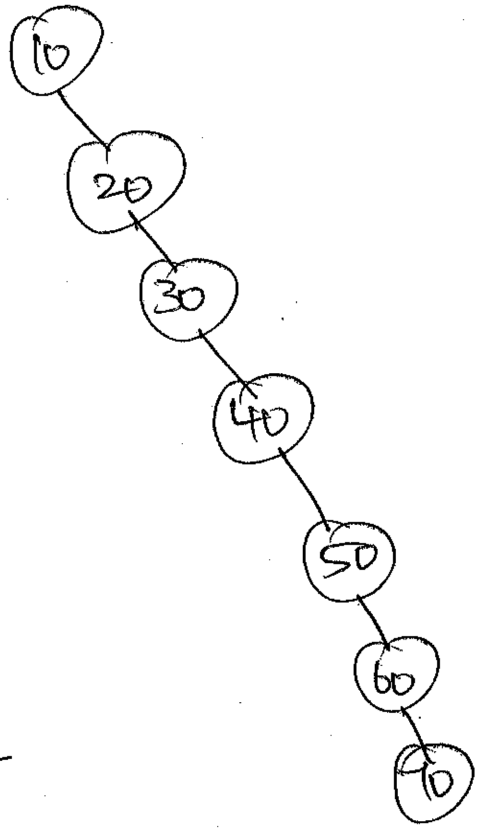



∴ Balanced.

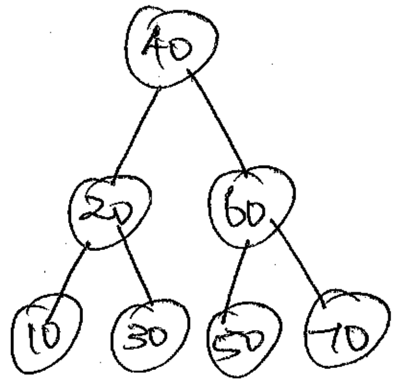
## Advantages of an AVL tree :-

∵ AVL trees are height balanced, opns like insertion and deletions have low time complexity.

Ex: If we have the following keys, 10, 20, 30, 40, 50, 60, 70, then a binary tree & an AVL tree would be



BST



AVL tree

To insert a node with key 'K' in the

→ BST — needs 7 comparisons - worst case

— AVL — needs only 3 comparisons - worst case

which is less than half of the BST.

∴ AVL trees ↑se the efficiency of the pgms.

x———x.