AVL (ADELSON-VELSKII AND LANDIS)

Def^n:

→ An empty binary tree is an AVL tree.

→ An non empty binary tree T is an AVL tree iff given T^L & T^R to be the left & right subtree of T and $h(T^L)$ and $h(T^R)$ to be the heights of subtree T^L & T^R resp; T^L & T^R are AVL trees & $|h(T^L) - h(T^R)| \leq 1$

→ $|h(T^L) - h(T^R)$ is known as balance factor.

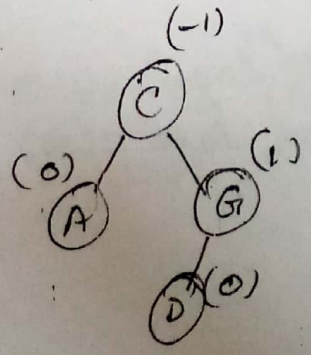
→ For AVL tree the balance factor of a node can be neither 0, 1 or -1

→ An AVL search tree is a binary search tree & which is an AVL tree.

Representation of an AVL tree

→ It is represented using link list representation.

→ The number against each node represents its balance factor.



INSERTION IN AN AVL SEARCH TREE

→ Insertion in AVL tree is similar to inserting an element into Binary search tree.

→ After insertion balance factor is checked

→ Rotation technique is used to restore balance of the search tree.

We only we need to rotate that's balance factor is changed.

Rebalancing of AVL

(1) LL Rotation

(2) RR "

(3) RL Rotation → ~~RR + LL~~ LL + RR

(4) LR Rotation → RR + LL

ADVANTAGE OF AVL

→ Insertion of element in Binary search tree may turn tree into right skewed or left skewed ~~tree~~ binary search tree. The skewed Tree take time O(n). ∴ it is require to maintain binary search tree to be of balanced height, whose time complexity is O(log n)

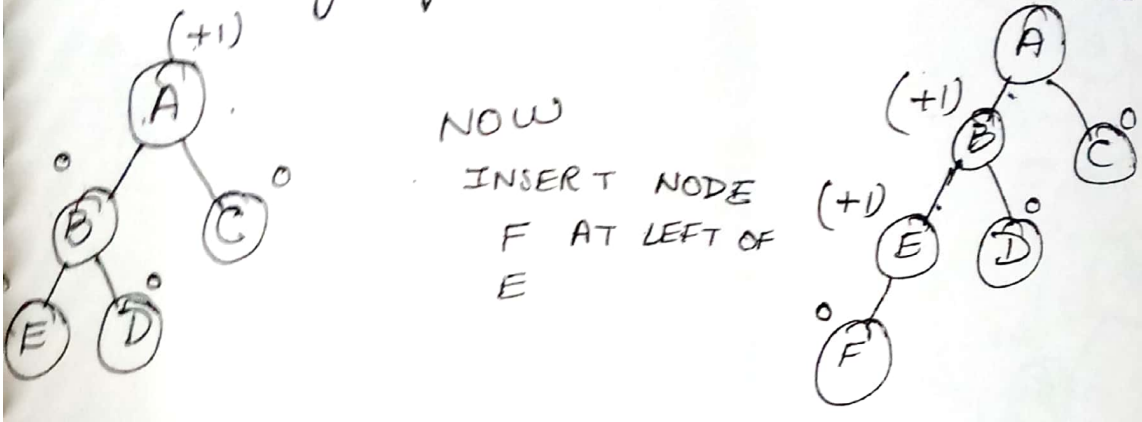
→ It means AVL tree take O(log n) where as left skewed & right skewed take O(n)

Rebalancing rotations are classified as follows based on insertion of node:-

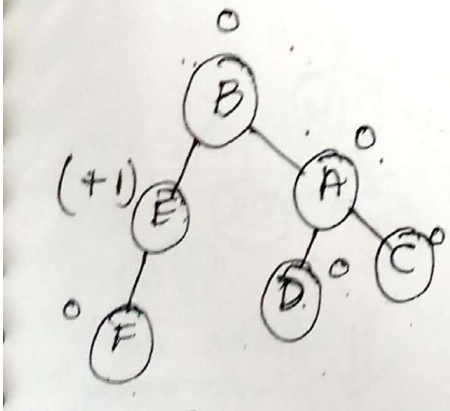
1) LL ROTATION:

→ Inserted node is in the left subtree of left subtree of node A.

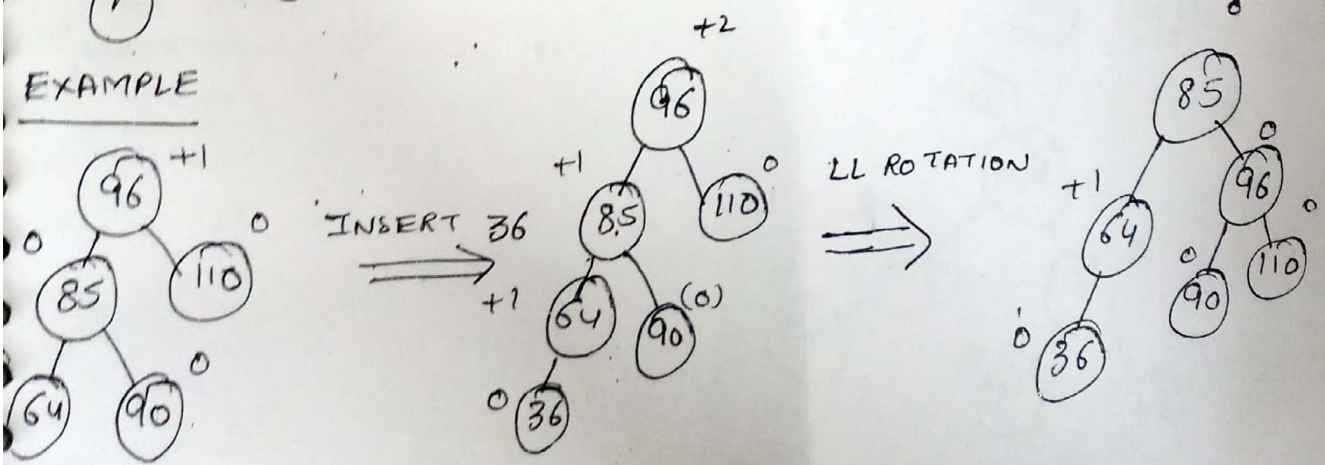
→ Balancing of LL Rotation is as follows:



after LL Rotation



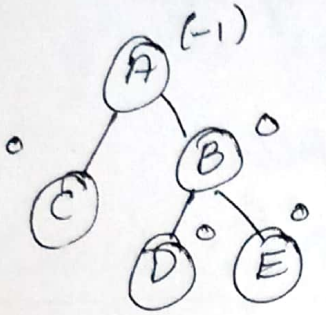
EXAMPLE

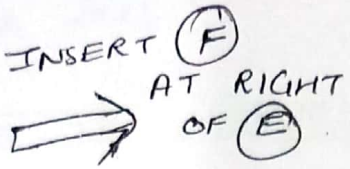


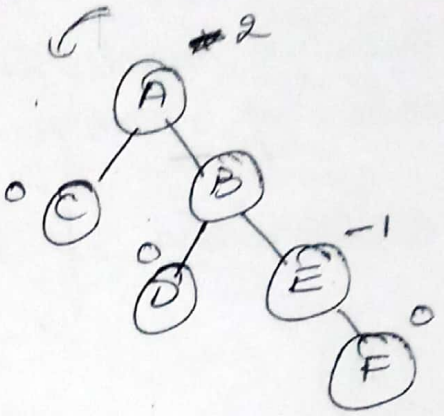
(2) (RR ROTATION)

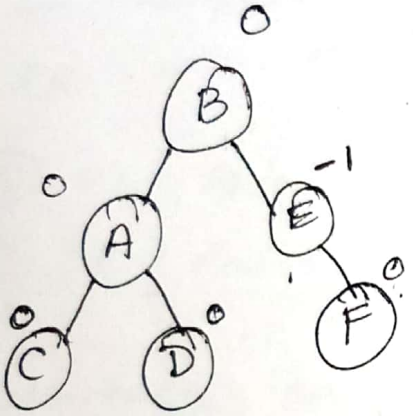
→ Inserted node is in the right subtree of right subtree of node A.

→ It is done in following way:

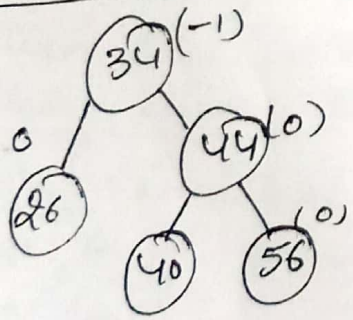


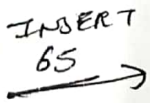


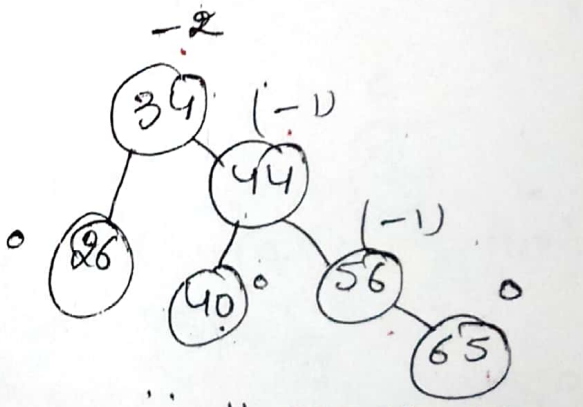




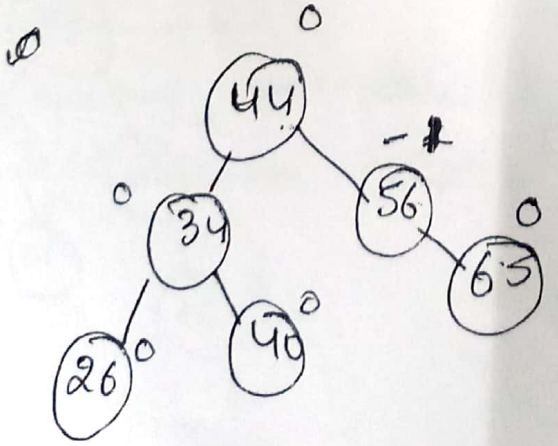
EXAMPLE :







AFTER RR ROTATION.



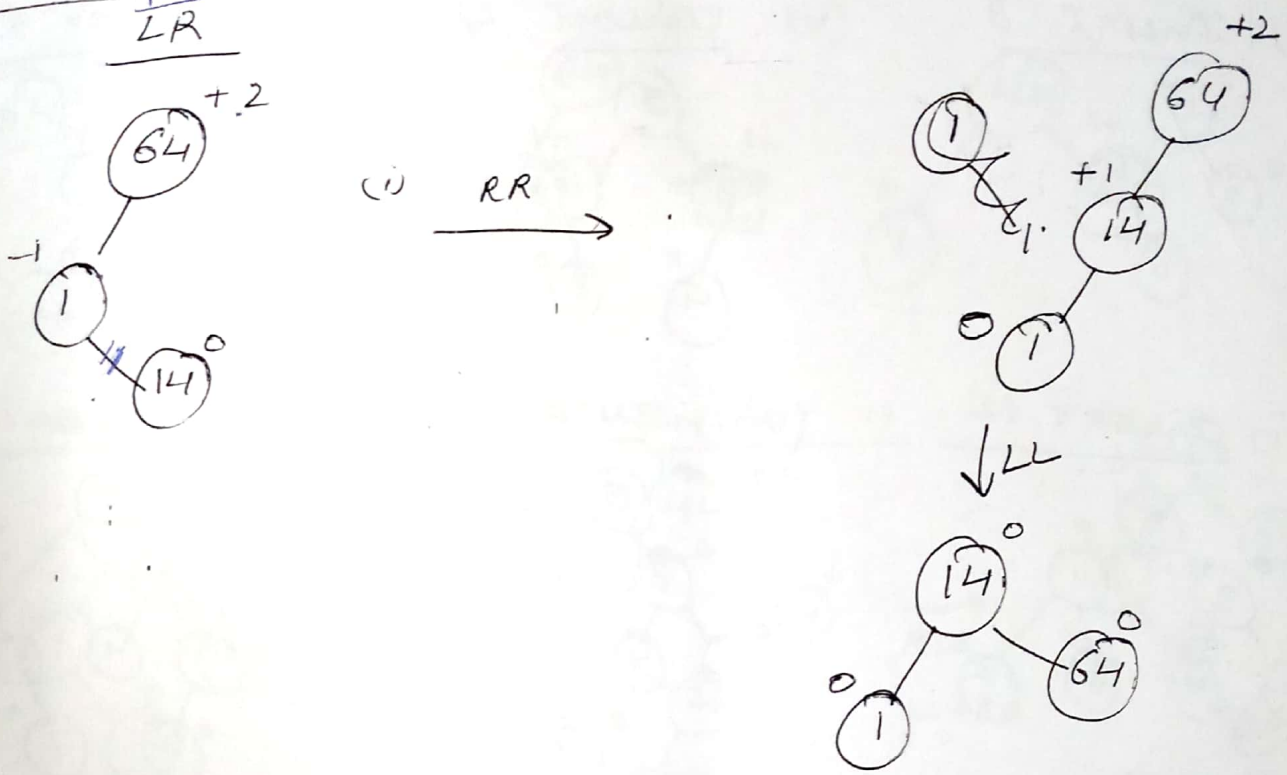
(3) LR Rotation

→ Inserted node is in the right subtree of left subtree of node A

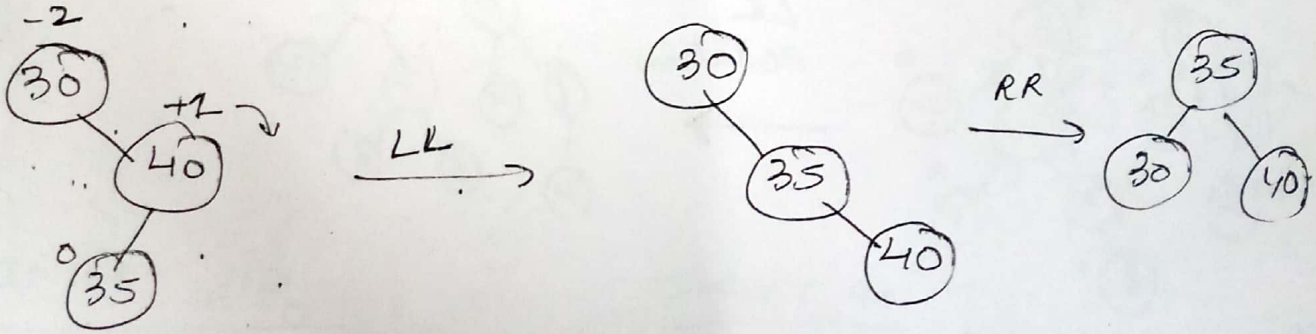
→ It is double rotation i.e. RR followed by LL

EXAMPLE

LR



(4) RL Rotation

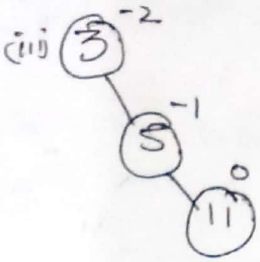


Ans:

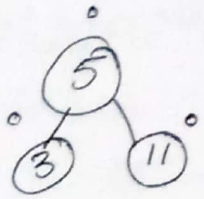
3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10

i) 

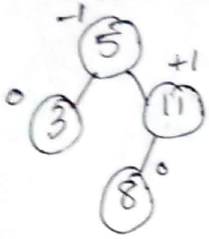
ii) 

iii) 

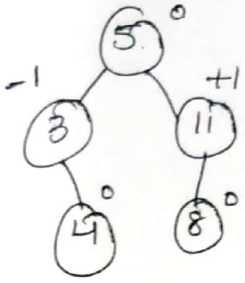
Apply RR Rotation



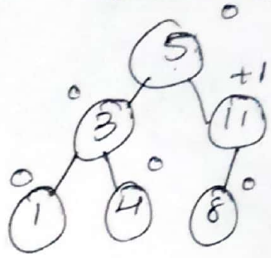
iv) Insert 8



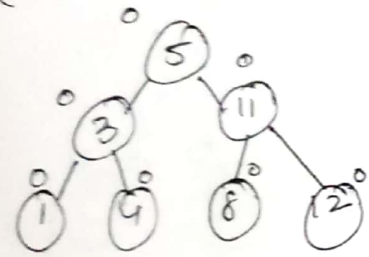
(v) Insert 4



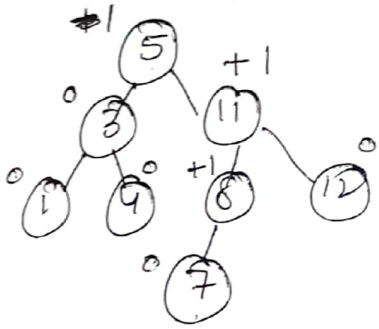
(vi) INSERT 1



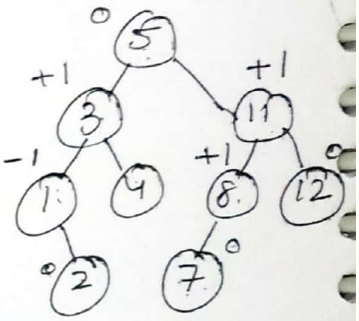
(vii) INSERT 12



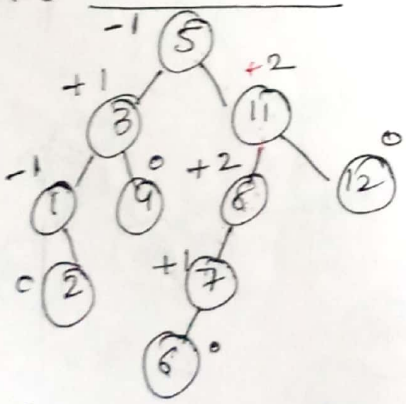
(viii) INSERT 7



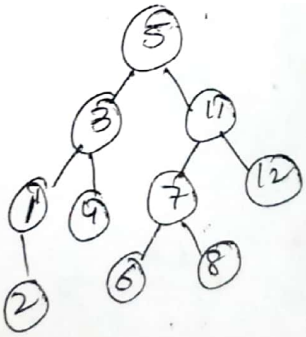
(ix) INSERT 2



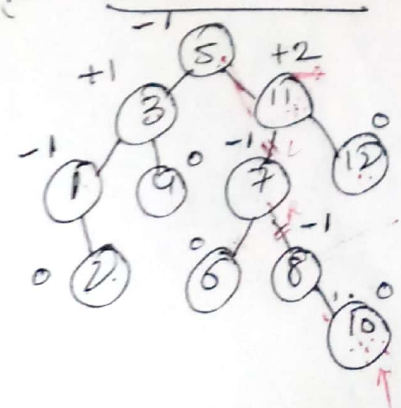
(x) INSERT 6



LL Rotation

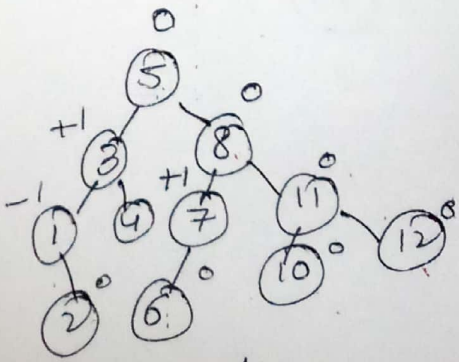


(xi) INSERT 10



LR Rotation

[Apply ~~LL + RR~~ RR + LL (i) (ii)]



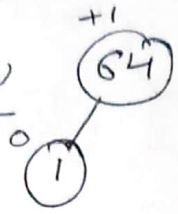
Ans

Question

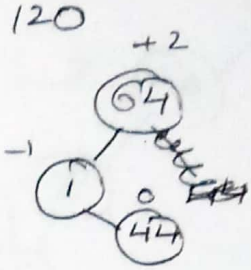
64, 1, 44, 26, 13, 110, 98, 85, 52, 120

(i)

(ii)

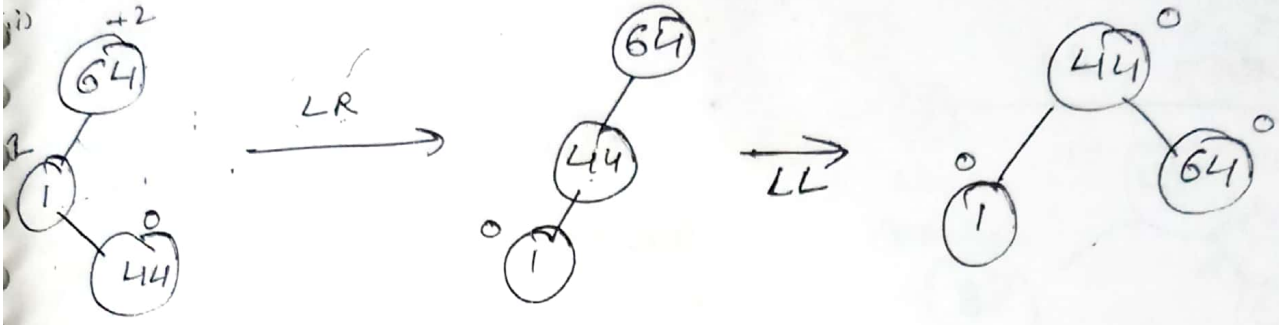


(iii)



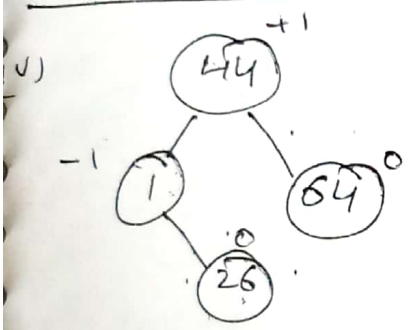
INSERT 44

(iv)

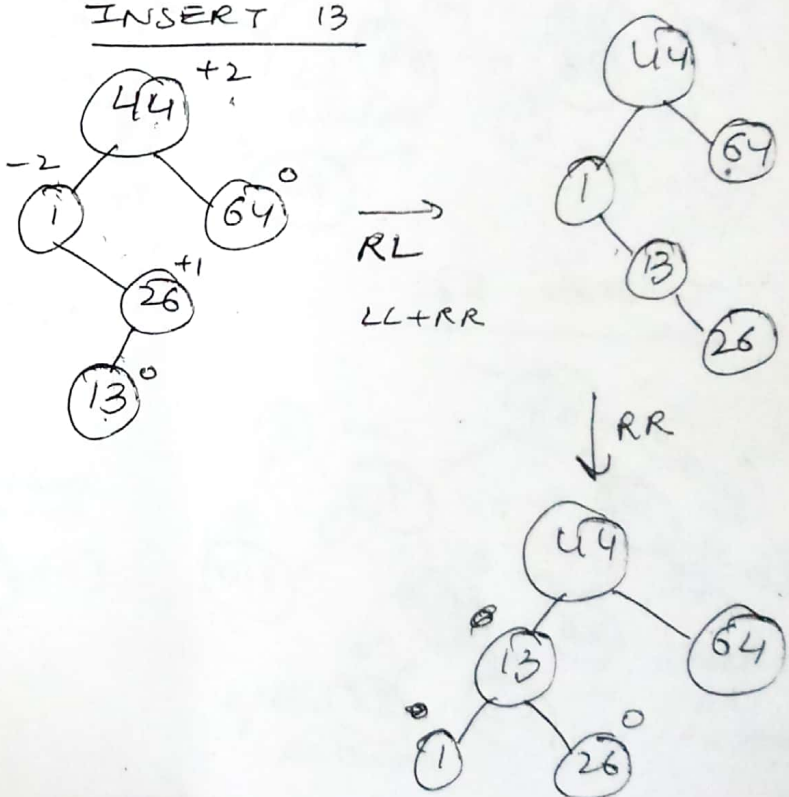


INSERT 26

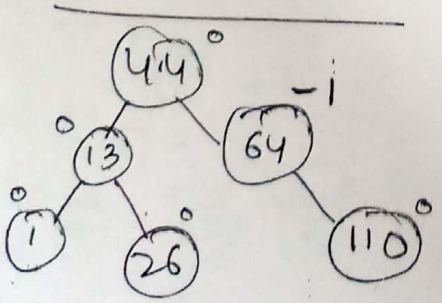
(v)



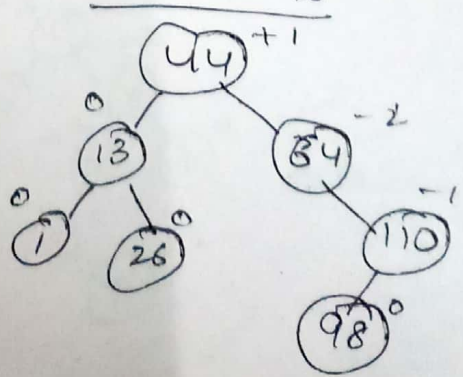
(v) INSERT 13



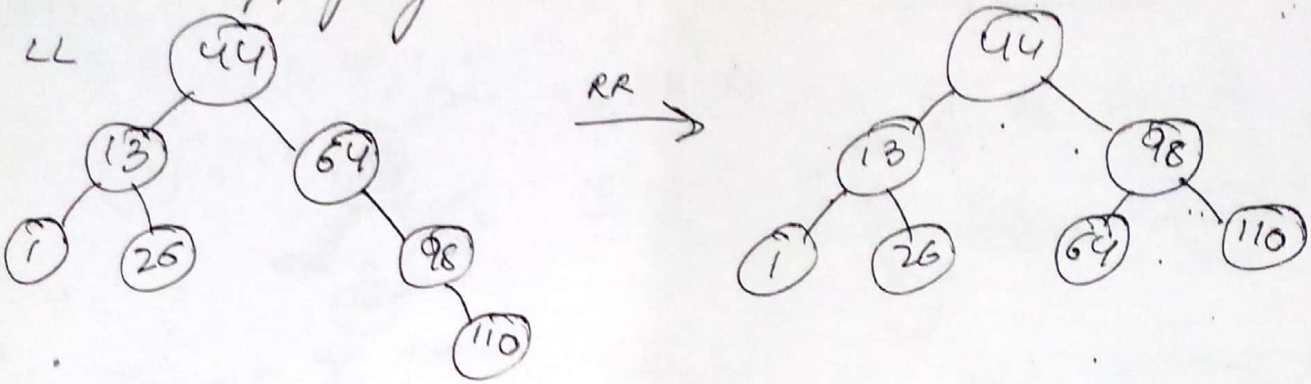
INSERT 110



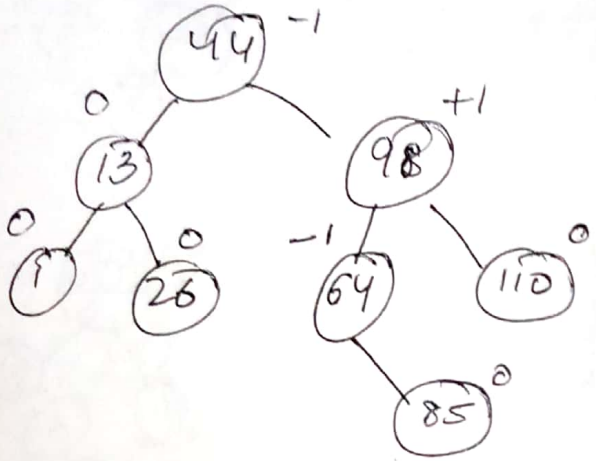
INSERT 98



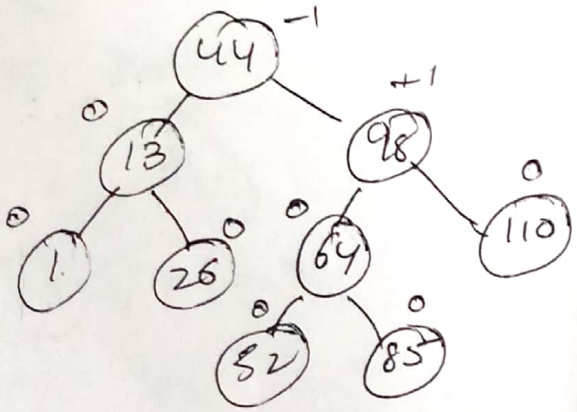
Now applying RL rotation



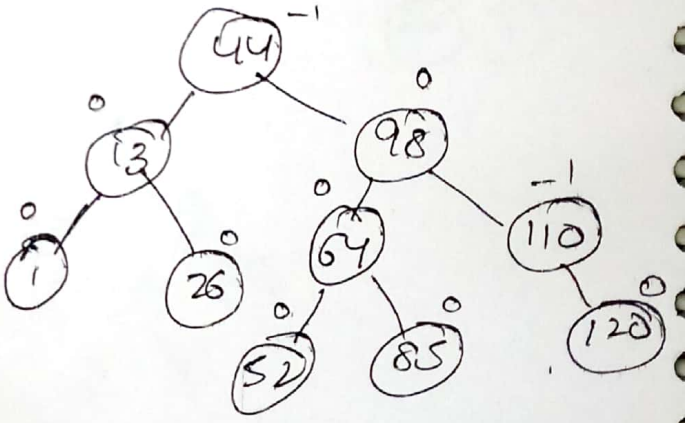
INSERT 85



INSERT 52



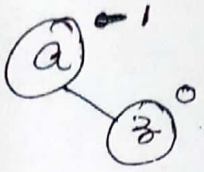
INSERT 120



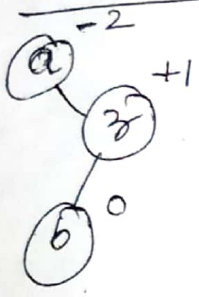
Q.11: a, z, b, y, c, x, d, w, e, v, f

Solution (i) INSERT a

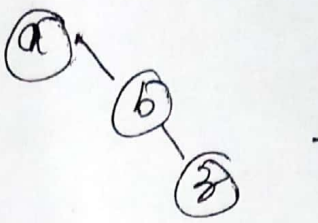
(ii) INSERT z



(iii) INSERT b

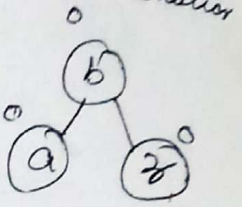


Apply RL Rotation [i.e. LL + RR (i) (ii)]



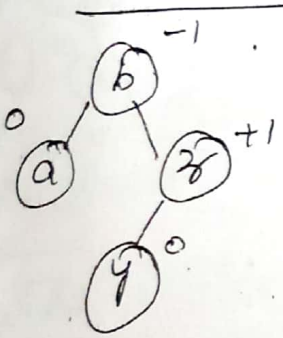
LL Rotation

RR Rotation

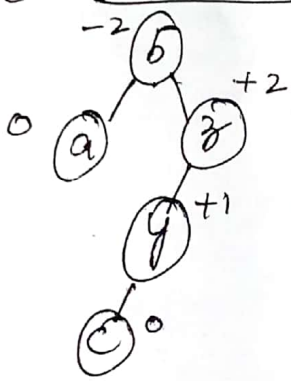


[Tree after apply RL rotation]

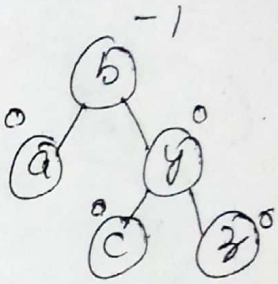
(iv) INSERT y



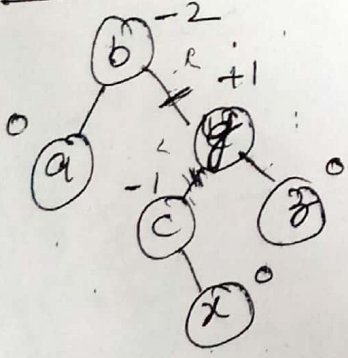
(v) INSERT C



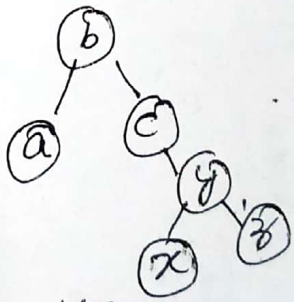
LL Rotation



(vi) INSERT x

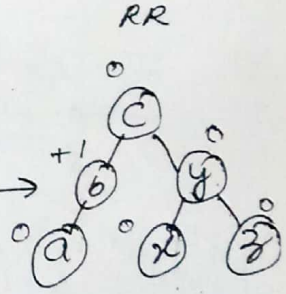


RL rotation [LL + RR]



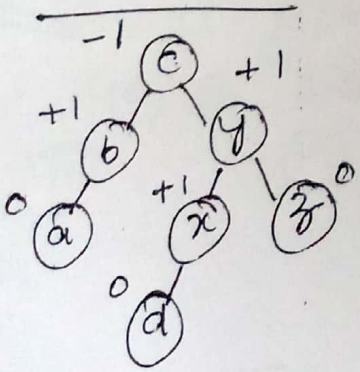
after LL Rotation

RR

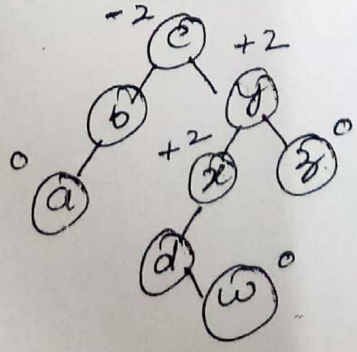


[tree after RL rotation]

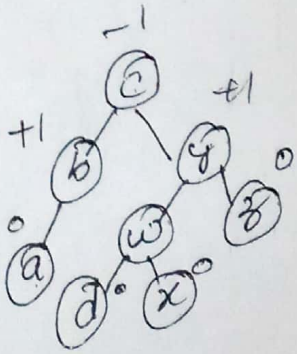
(vi) INSERT d



(vii) INSERT w

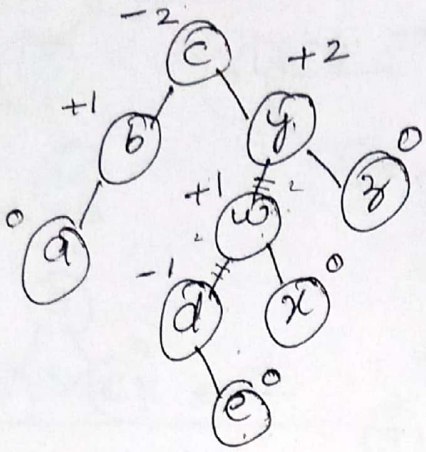


LR Rotation

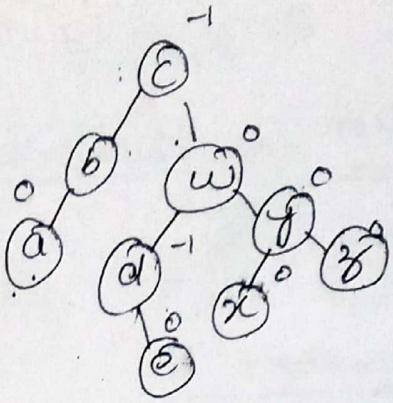


~~INSERT~~

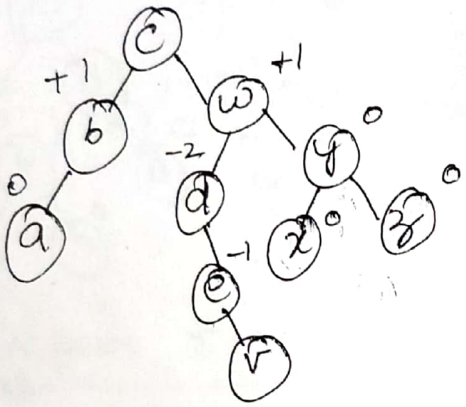
(viii) INSERT e



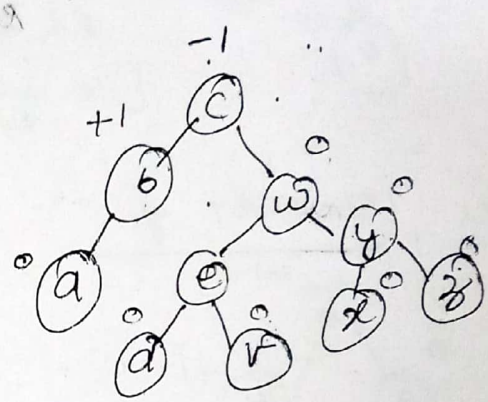
LL rotation →



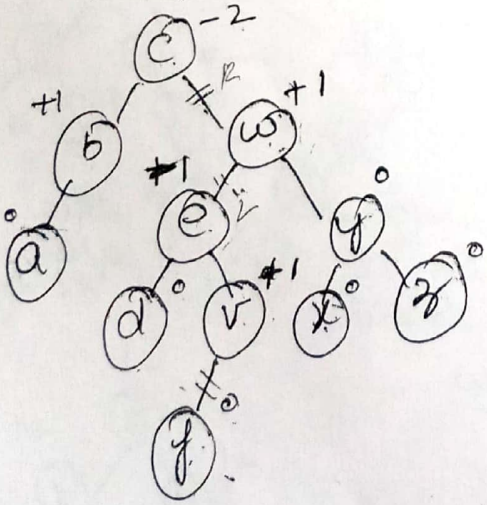
(ix) INSERT v



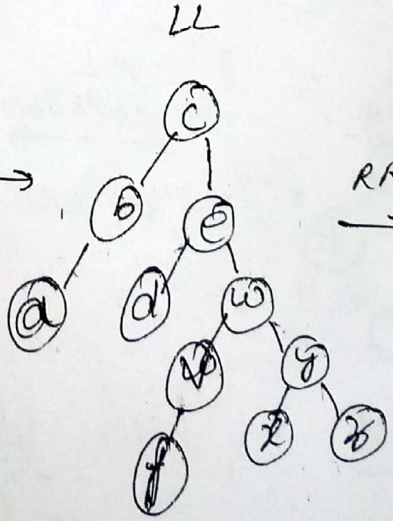
RR rotation →



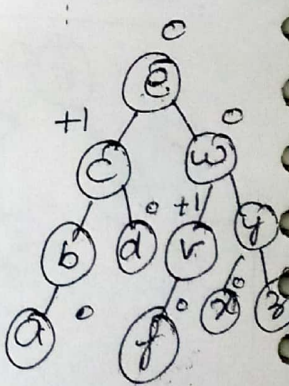
X) INSERT f



RL →



RR →



Binary Search tree

→ Average running time $f(n) = O(\log n)$
→ In this easy to delete & insert item.
→ BST ~~have advanta~~ is prefered over foll:

(a) Sorted linear array:
In this searching requires $f(n) = O(\log_2 n)$ time but it is expensive to insert & delete item.

(b) Linked list → It is easy to insert & delete item, but expensive to search as it take $f(n) = O(n)$ time.

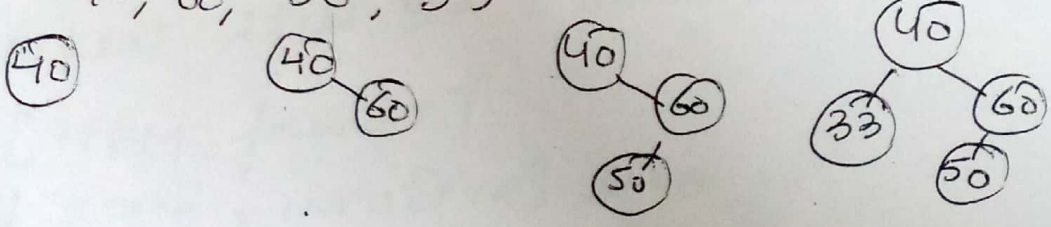
BST which is either empty or satisfies the foll. rule:

(i) The value of the key in the left child or left subtree is less than the value of root.

(ii) The value of the key in the right child or right subtree is more than or equal to the value of the root.

(iii) All the sub-trees of the left & right childrens observe the two rules.

Ques: Insert foll elements in BST
40, 60, 50, 33



Hugo

Find location of item.

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

Three cases are:

i) LOC = NULL & PAR = NULL → Tree empty

ii) LOC ≠ NULL & PAR = NULL → item is at root of T

iii) LOC = NULL & PAR ≠ NULL → item not in T & can be added in T as a child.

(1) LOC ≠ NULL & PAR ≠ NULL

(2) [Tree empty] ✓

If ROOT = NULL
then
set LOC = NULL & PAR = NULL
return

(3) [item at root] ROOT → INFO

If ITEM = INFO[ROOT] ROOT → INFO
Set LOC = ROOT & PAR = NULL
return

(4) [Initialize pointer PTR & SAVE]

If ITEM < INFO[ROOT] then:
Set PTR = LEFT[ROOT] & SAVE = ROOT
else:
Set PTR = RIGHT[ROOT] & SAVE = ROOT
[end of if structure]

(5) Repeat steps 5 & 6 while PTR ≠ NULL.

(6) [ITEM found?]

If ITEM = INFO[PTR] then
Set LOC = PTR & PAR = SAVE
Return.

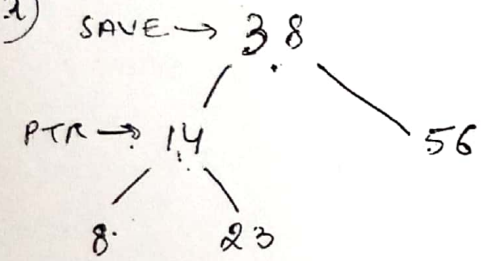
6) If ITEM < INFO[PTR] then
Set SAVE = PTR & PTR = LEFT[PTR]
else:
Set SAVE = PTR & PTR = RIGHT[PTR]
[end of if]
[end of step 4 loop]

7) [Search unsuccessful] ✓
Set LOC = NULL & PAR = SAVE ✓

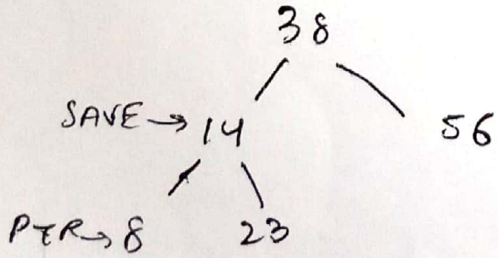
8) Exit.

EXAMPLE: ITEM = 8

a)



b) ITEM < INFO[PTR]



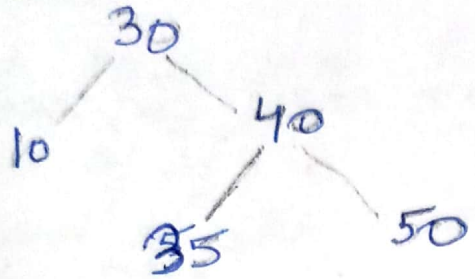
Algo

Insertion algorithm:

1) Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2) If LOC ≠ NULL, Then Exit.
3) [Copy ITEM into new node in AVAIL list]
4) If AVAIL = NULL, Then write: OVERFLOW & Exit
5) Set NEW := AVAIL, AVAIL = LEFT[AVAIL] &
   INFO[NEW] = ITEM
6) Set LOC = NEW, LEFT[NEW] = NULL &
   RIGHT[NEW] = NULL
7) [Add ITEM to tree.]
   If PAR = NULL then:
   set ROOT = NEW
   else if ITEM < INFO[PAR] Then:
   Set LEFT[PAR] = NEW
   else:
   set RIGHT[PAR] = NEW
   [end of if structure]
8) Exit.

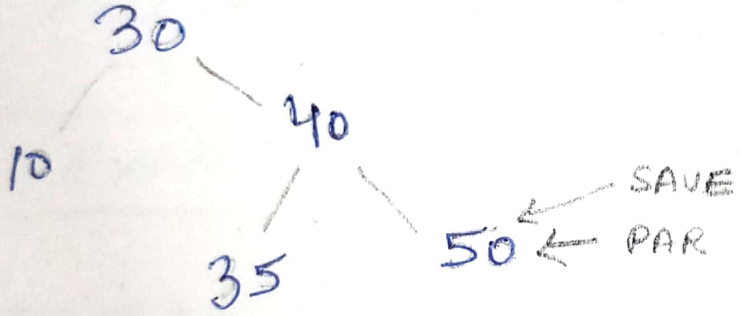
EXAMPLE :

INSERT – 55



→ call find algo

→



Deletion

Case 1: N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer.

Case 2: N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

Case 3: N has two children. Let S(N) denote the inorder successor of N. (RIGHT) The N is deleted from T by first deleting S(N) from T (by using case 1 & case 2) & then replacing node N in T by the node S(N).

InOrder Successor

BE
↺

ALGO OF DELETION

→ When there is no child of a node.
→ " " " single " " "

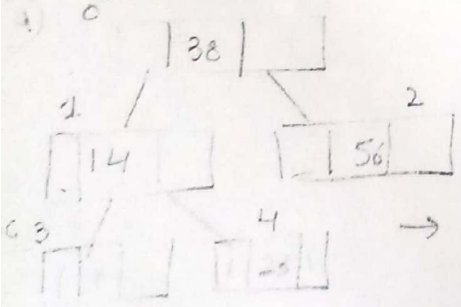
CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1) [Initialize Child]
If LEFT[LOC] = NULL & RIGHT[LOC] = NULL then:
  set CHILD = NULL
else if LEFT[LOC] ≠ NULL then:
  set CHILD = LEFT[LOC]
else
  set CHILD = RIGHT[LOC]
[end of if structure]

2) If PAR ≠ NULL then
  If LOC = LEFT[PAR] then
  set LEFT[PAR] = CHILD
else:
  set RIGHT[PAR] = CHILD
[end of if]
else
set ROOT = CHILD

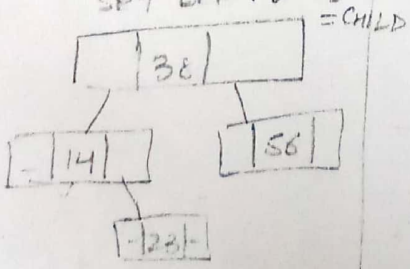
3) Return.

EXAMPLE:

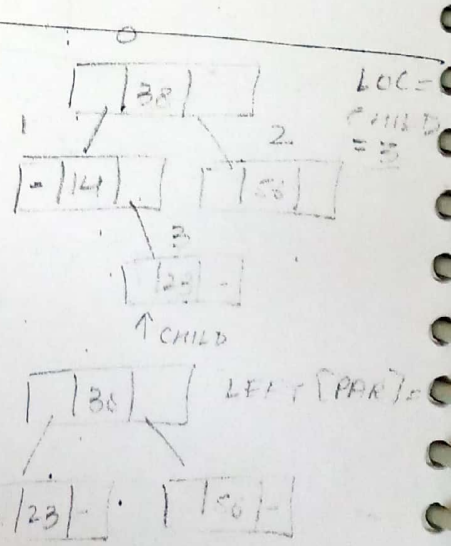


POINTERS → CHILD
→ PAR
LOC = 3, ITEM = 8
CHILD = NULL

SET LEFT[PAR] = CHILD



(b)



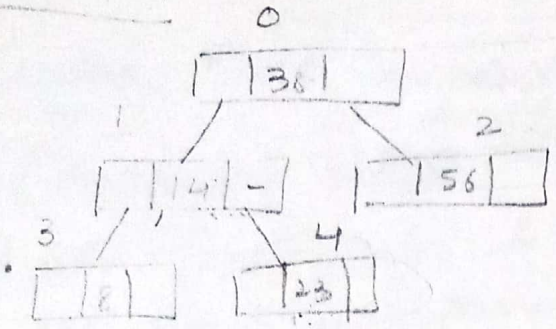
ALGO OF DELETION

(2) when node have ~~two~~ two childeren

Case B (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1. [find SUC & PARSUC]
2. Set PTR = RIGHT[LOC] & SAVE = LOC
3. Repeat while LEFT[PTR] ≠ NULL
   Set SAVE = PTR & PTR = LEFT[PTR]
   [end of LOOP]
4. Set SUC = PTR & PARSUC = SAVE
5. [Delete inorder successor]
   Call CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
6. [Replace node N by its inorder successor]
7. If PAR ≠ NULL then:
   If LOC = LEFT[PAR] Then:
   set LEFT[PAR] = SUC
   else
   SET RIGHT[PAR] = SUC
   [end of if structure]
   else
   set ROOT = SUC
   [end of if structure]
8. Set LEFT[SUC] = LEFT[LOC] &
   RIGHT[SUC] = RIGHT[LOC]
9. Return

EXAMPLE.



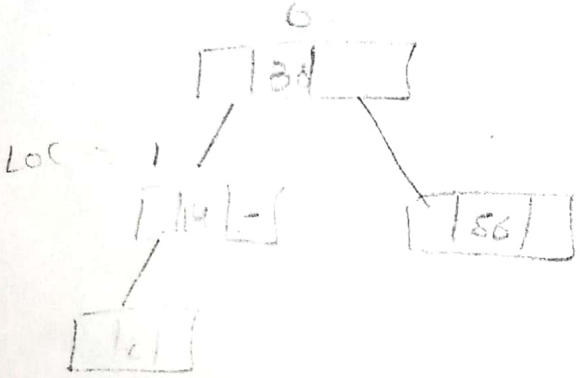
suppose delete item = 14, LOC = 1.

(i) PTR = 4, SAVE = 1.

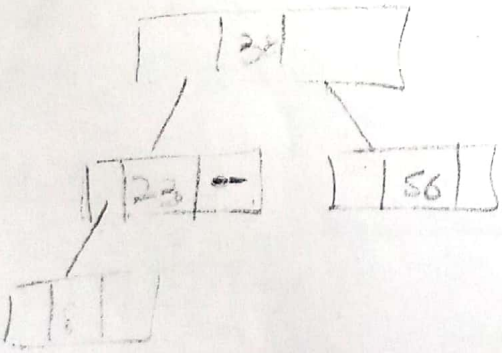
(ii) SUC = 4 & PARSUC = 1

CASE A: (14, 3, 4, 0, 4, 1) ⇒ LOC = 4, PAR = 1.

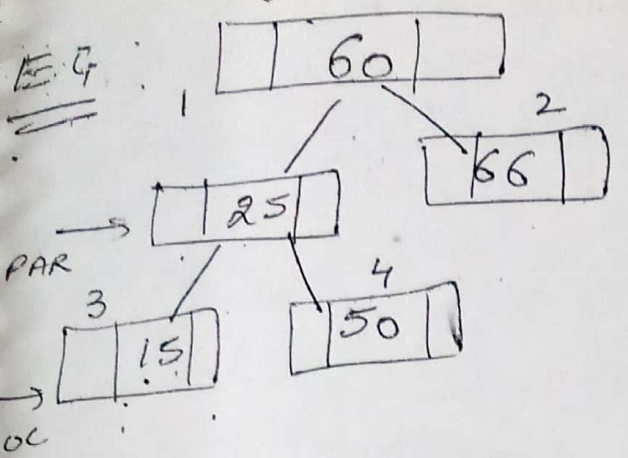
CHILD = NULL,



LEFT(PAR) = SUC



E.G :



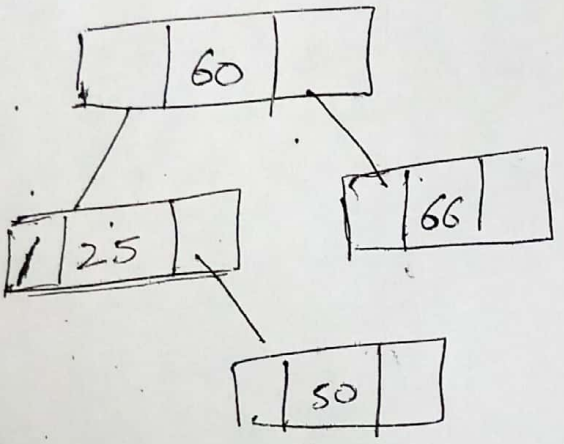
DELETE – 15.

LOC = 3

i) In this left of LOC = NULL & Right [LOC] = NULL

∴ CHILD = NULL

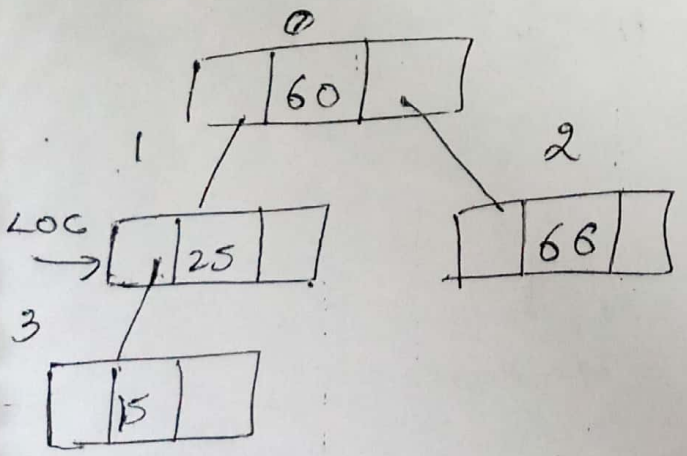
ii) LOC = LEFT[PAR], so, set Left[PAR] = CHILD



E.G when one child

Delete – 25

LOC = 1

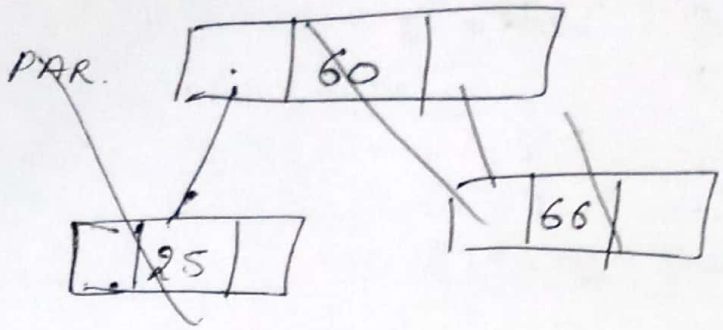


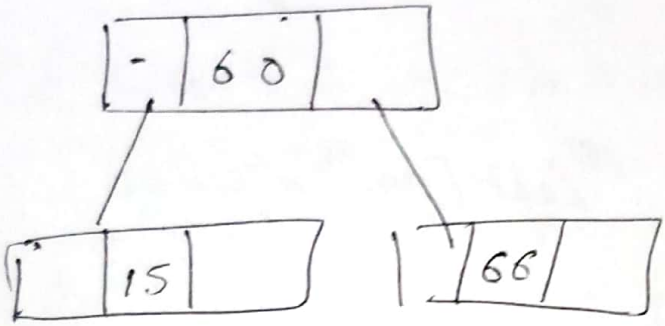
→ In this Left[Loc] ≠ NULL

∴ set child = Left[LOC]

child = 3.

→ set Left[PAR] = child

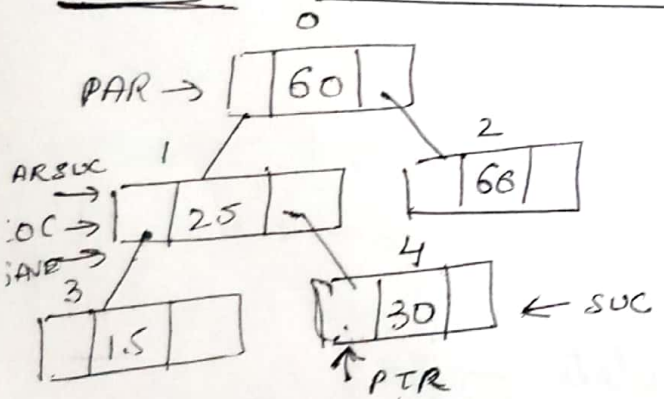




Case B Example when 2 children.

Delete → ~~25~~

P.



(a) PTR = Right[LOC], PTR = 4, SAVE = 1, LOC = 1

(b)

(c) SUC = 4, PARSUC = 1

(d) Call caseA(25, 3, 4, 0, 4, 1)

Info, = 25, left = 3, right = 4, Root = 0,
LOC = 4, PAR = 1

→ child = Null

→ ~~if~~ LOC = ~~Left~~ Right[PAR]

set Right[PAR] = child