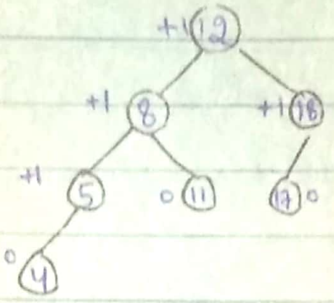
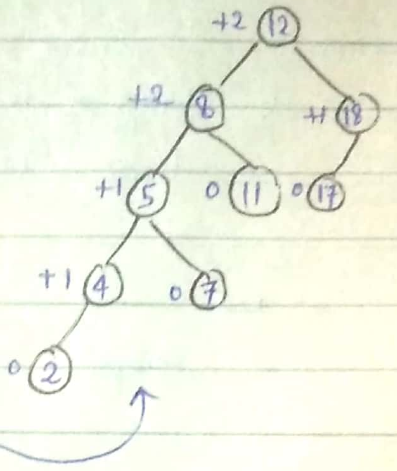
* AVL TREES ( self-balancing binary Search trees)

AVL trees are special types of Binary Search trees which are balanced i.e. in which balance factor of each node lies in range {-1, 0, +1} where balance factor is calculated as :

⇒ Balance factor of node = height of left subtree - height(right ST)

eg of AVL tree :





eg of BST which is not Binary AVL :

{ see node 8 and 12 has B.F = +2 ∴ Not AVL }.

• Why to use AVL trees : ?

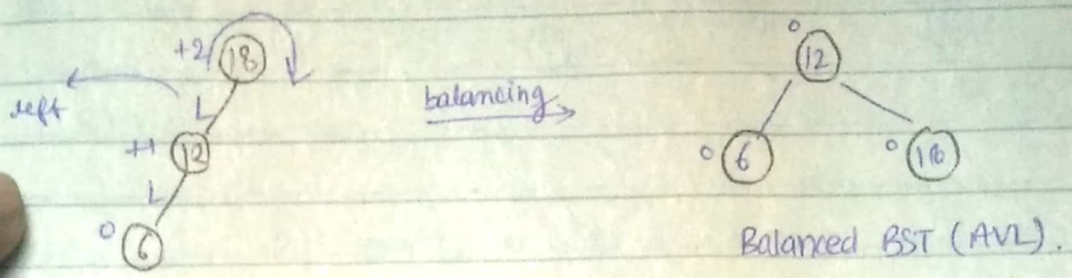
Binary Search Trees (BST) are used to store data which involves quick and frequent searching, insertion & deletion operations. Complexity of all these operations is O(h) where h is height of Binary Search tree. If BST is a skewed BST, then complexity will become O(n) which is quite large when compared to quantity of data we store in BST. where n = no. of nodes in a BST. So, to reduce O(h) to nearly O(logn), tree should be balanced or nearly balanced. That's why we use AVL trees to ensure that insertion, deletion & searching operations takes O(logn) time.

NOTE : All leaf nodes has balance factor zero.

How to balance a BST to form AVL tree ?

Any unbalanced BST can be balanced using four types of rotations :

i) left-left rotation.



Procedure for doing left-left rotation :

a) Notice the leaf node . (6)

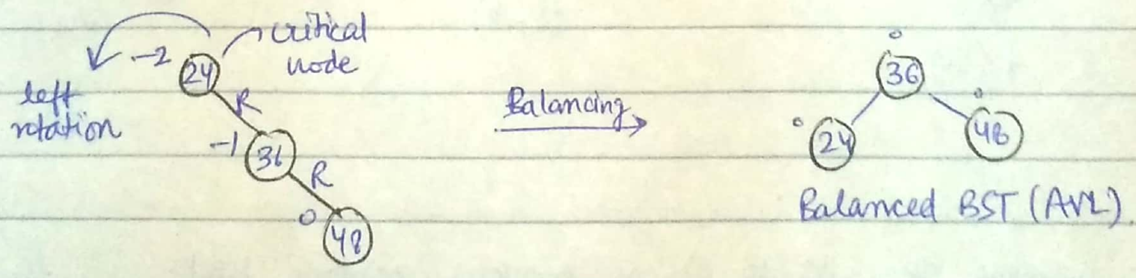
b) using a bottom-up-approach , move from leaf node to root-node ( 6 to 18) and find the first node whose B.F does not lie in desired range. (18).

c) Now , notice the first two route through which you can reach leaf node from critical node. (A critical node is a node which is unbalanced).

d) From 18 , we have to reach 6 , so movement will be left & then again left. That's why , it's a left-left problem. (LL)

e) Since , its a LL-problem , so, do right rotation about critical node to make tree balance.

ii) Right -Right Case :



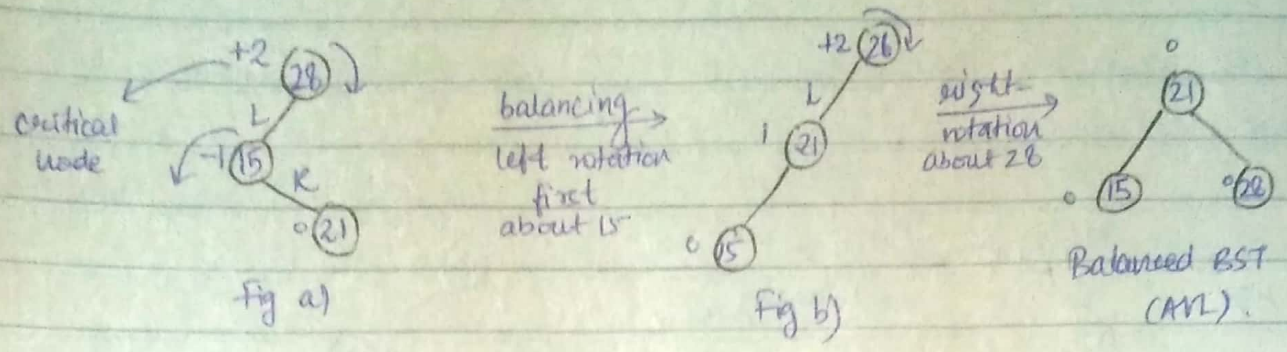
Procedure for doing right-right problem solving :

a) Follow points a) → c) of above left-left case. These are general steps to be followed to find the type of problem.

b) from 24 , we have to reach 48 which can be done using right-right movement

c) So, its RR problem . So, do left rotation about critical node to balance tree.

c) Left-Right Case :



Procedure :

a) follow general steps a) to c) of previous problems first.

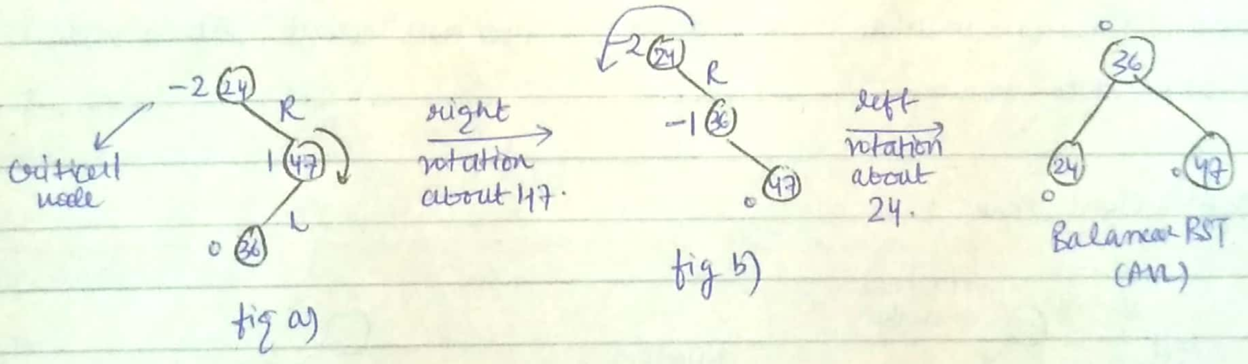
b) Now, we can reach 21 from 28 by moving first left then right, so, its a left-right case.

c) This problem is solved using two rotations.

d) Since, in fig a), node 15 suffers from R ∴ do left-rotation about 15 to get fig b).

e) Since, in fig b), node 28 suffers from L ∴ do right-rotation about 28 to get Balanced tree (AVL).

d) Right-left Case :



Procedure :

a) Follow general steps a) to c) of previous problems first.

b) Now, we have to reach 36 from 24 by moving Right & left. So, its a right-left case.

c) Since, in fig a), node 47 suffers from L, do right-rotation about 47 to get fig b).

d) Since, in fig b), node 24 suffers from R, do left-rotation about 24 to get AVL.

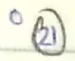
* OPERATIONS ON AVL

i) Searching
Searching operation is same as done in case of BST.

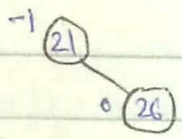
ii) Insertion
We will first insert the node in the same way as done in case of BST. Since, after insertion, tree may get some nodes having B.F out of desired range, so, we will balance it using above four discussed techniques.

Ques → Insert nodes : 21 26 30 9 4 14 28 18 15 10 2 3 7 in AVL tree.
{In insertion case, notice inserted node instead of leaf node}.

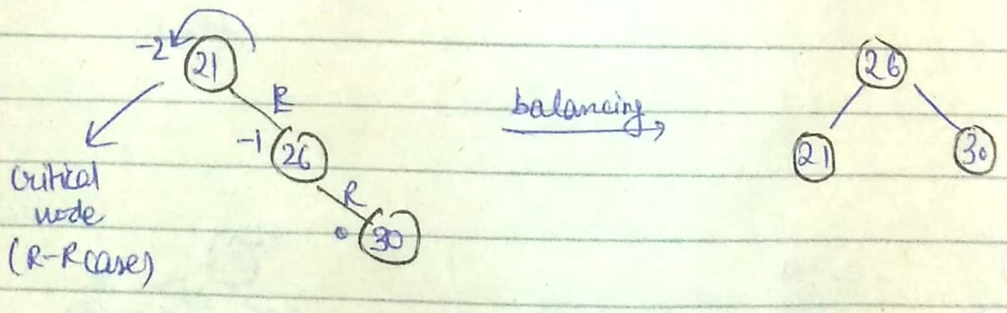
a) insert (21)



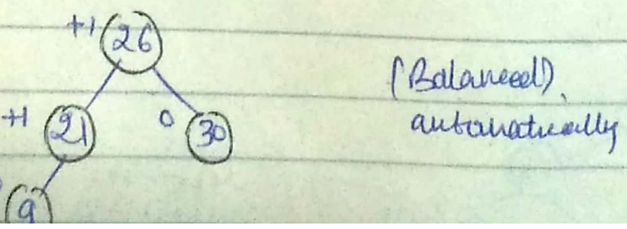
b) insert (26)



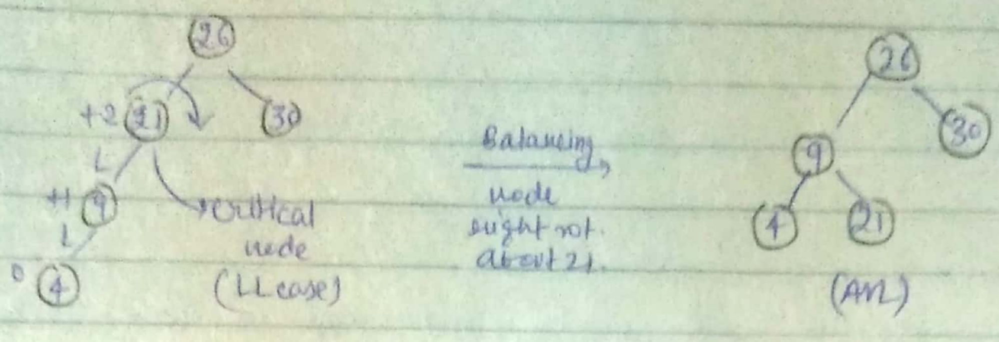
c) insert (30)



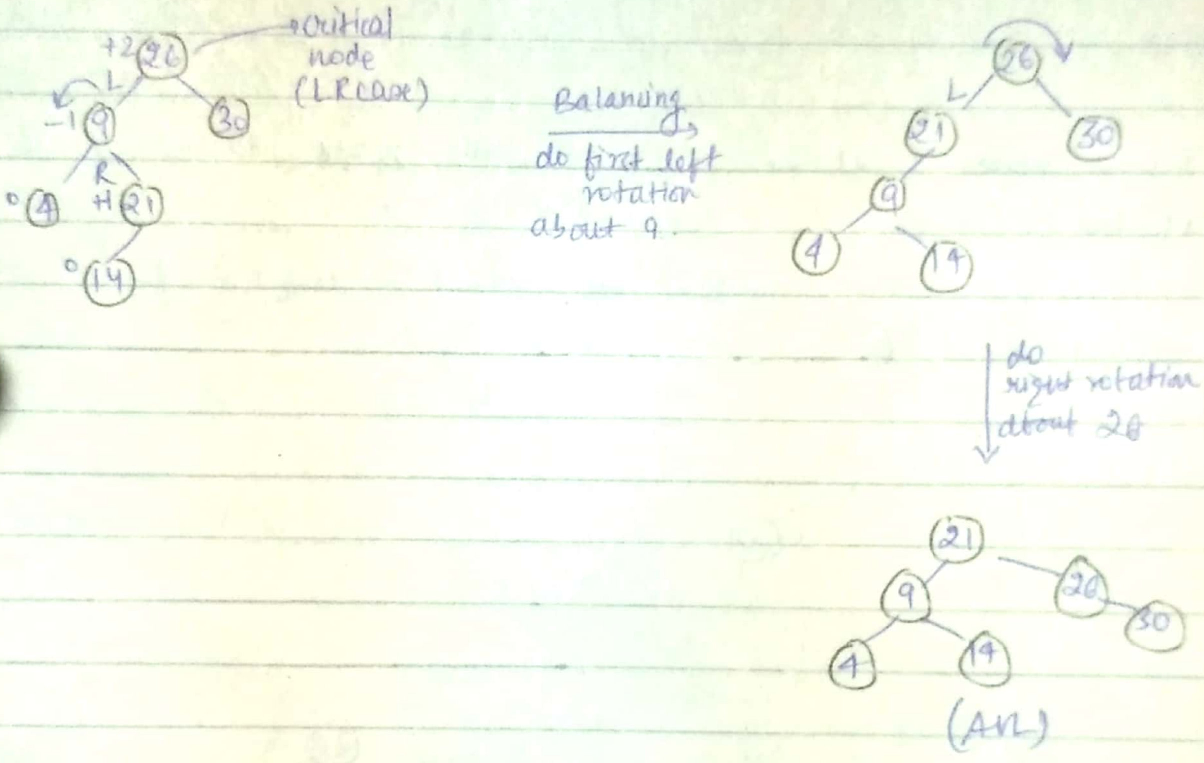
d) Insert (9)



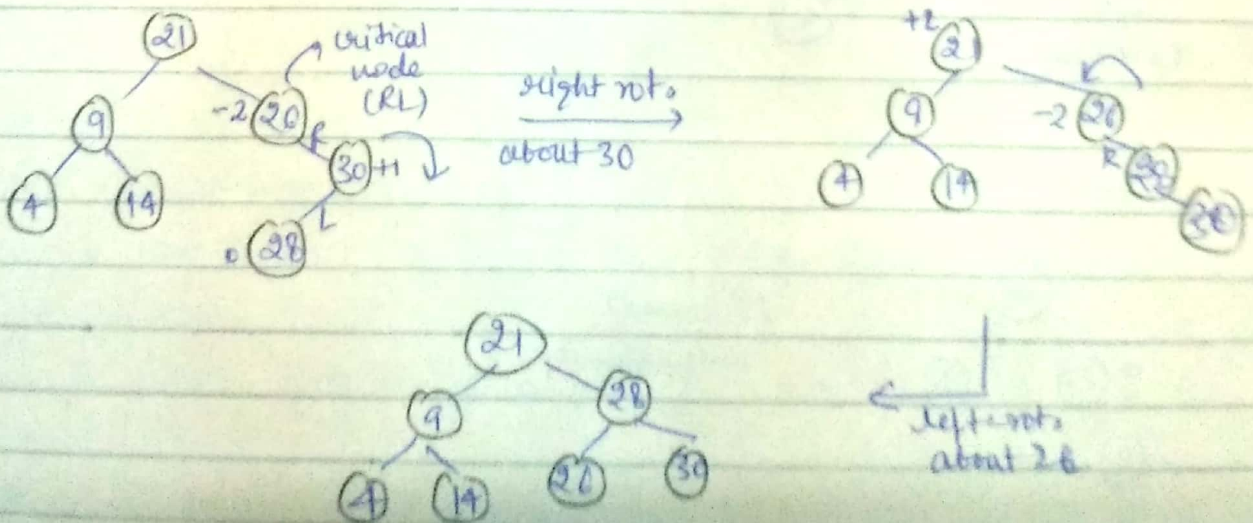
e) insert(4)



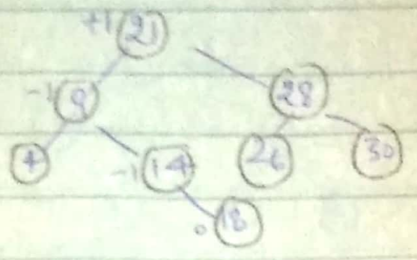
f) insert(14)



g) insert(28)

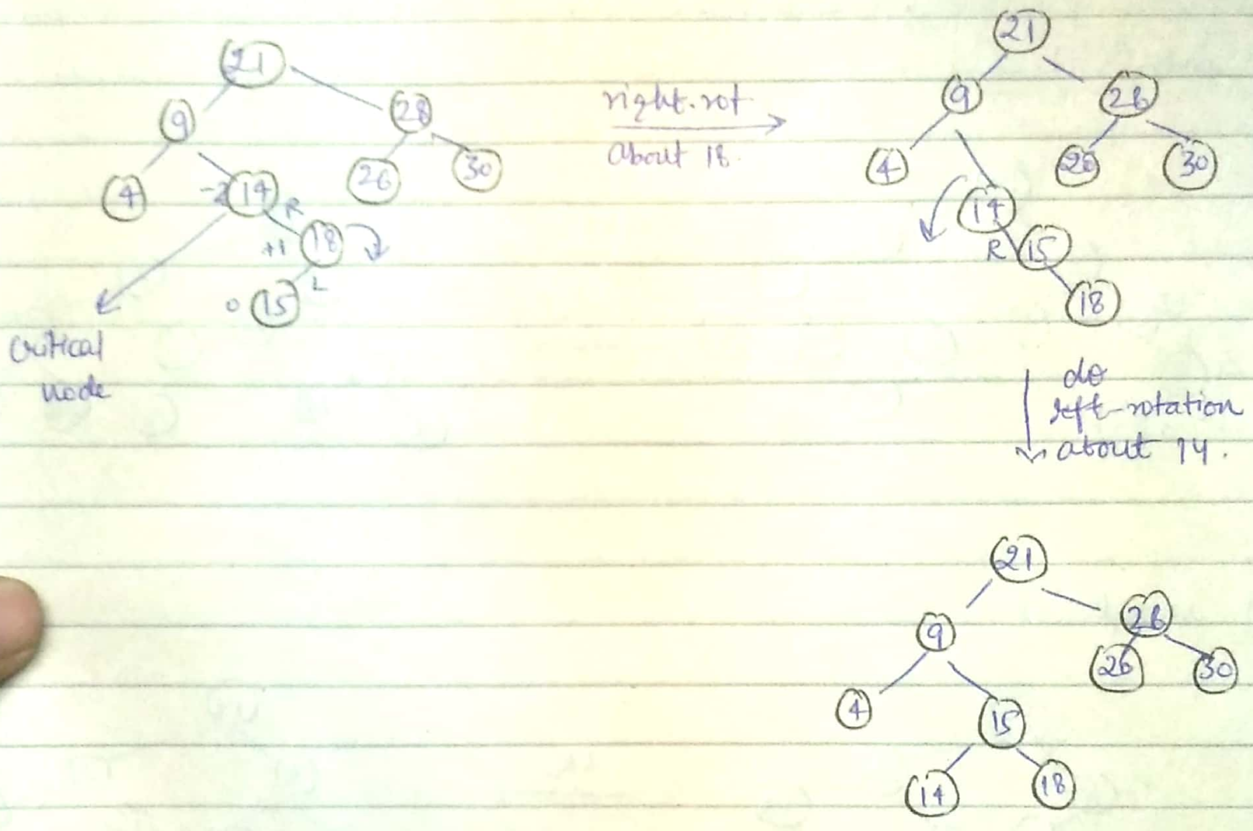


h) insert (18)

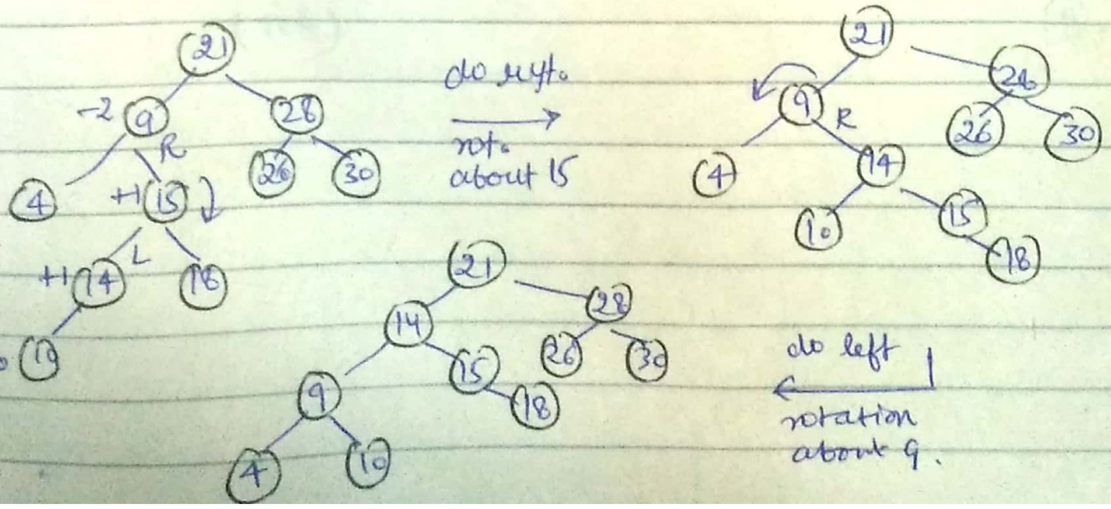


(balanced)

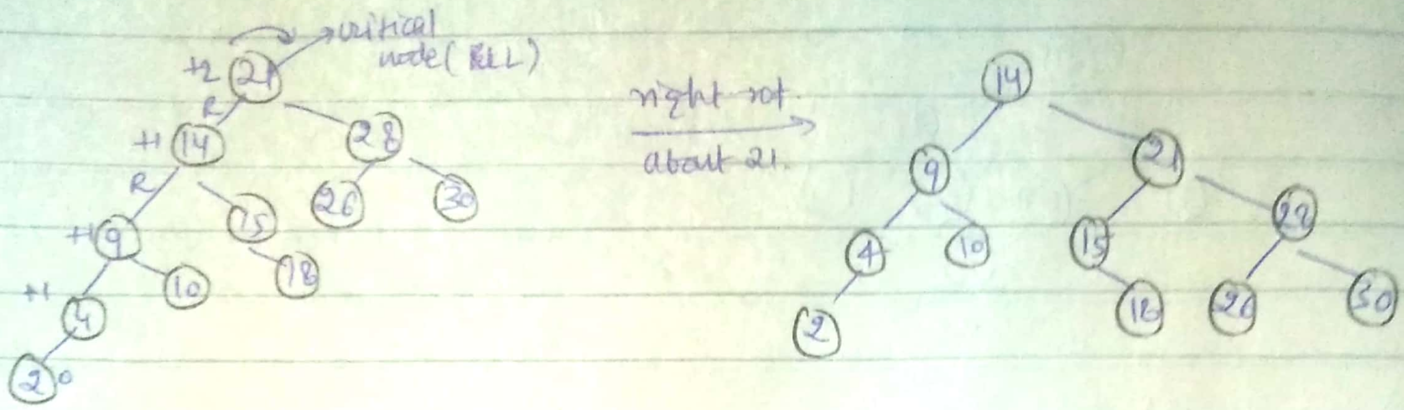
i) insert (15)



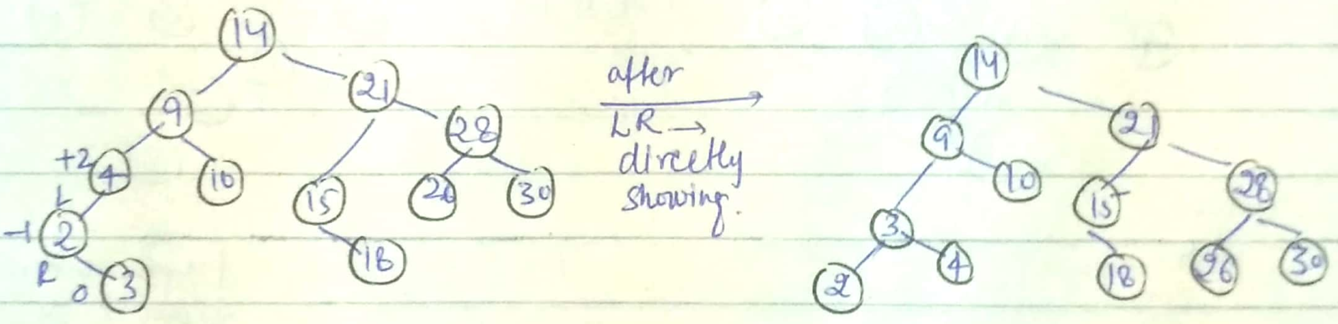
j) insert(10)



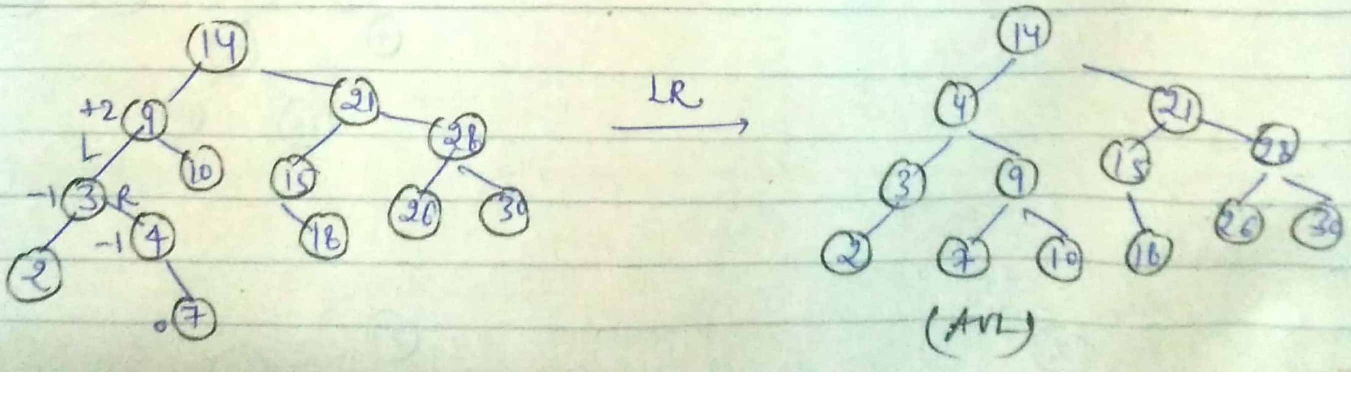
k) insert(2)



l) insert(3).



m) insert(7).


+2
L
-1
R
-1
0
LR
(AVL)
14
9
21
3
4
10
15
28
2
7
18
26
30