EFFICIENT BINARY SEARCH TREES

BST is a binary tree where the left child has a value lesser than the root node and the right child has a value greater than or equal to the parent node.

A BST using which we can retrieve data efficiently (fast) is a efficient binary search tree.

Types of efficient binary search trees :
- → Optimal binary search tree
- → AVL trees
- → Red black trees

OPTIMAL BINARY SEARCH TREE

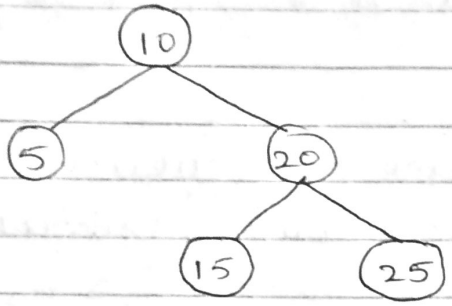
Given a binary search tree with fixed number of elements ~~in a tree~~ and the probabilities of each key, the average expected cost of accessing the elements in a tree can be computed.

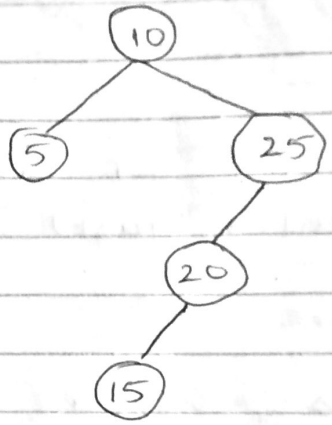
An optimal BST or OBST is a BST which has minimal or least expected cost.

This BST is well suited for fixed set of elements

NOTE : We can construct OBST only for a fixed set of element represented as BST. For such a tree, we make no additions or deletions from the set. Only searches are performed.

Ex: Find the expected cost (average number of comparisons) for the following trees with the priorities/probabilities given





Keys :	5	10	15	20	25
A> Probability:	1/5	1/5	1/5	1/5	1/5
B> Probability:	0.3	0.3	0.05	0.05	0.3

A> Cost of first tree

Item	5	10	15	20	25
Probability	1/5	1/5	1/5	1/5	1/5
Height of Node (or) No. of comparisons	2	1	3	2	3
cost = probability * height	1/5 * 2	1/5 * 1	1/5 * 3	1/5 * 2	1/5 * 3

Total cost = (1/5 * 2) + (1/5 * 1) + (1/5 * 3) + (1/5 * 2) + (1/5 * 3)

= 2.2

A> cost of second tree

Item	5	10	15	20	25
Priority	Y5	Y5	Y5	Y5	Y5
Height(or) No. of comparisons	2	1	4	3	2
cost = priority * height	Y5 * 2	Y5 * 1	Y5 * 4	Y5 * 3	Y5 * 2

Total cost = 2.4

Note: By comparing the cost of the first tree and second, the first tree has a better average behaviour

B> cost of first tree

Item	5	10	15	20	25
Priority	0.3	0.3	0.05	0.05	0.3
Height (or) No. of comparisons	2	1	3	2	3
cost = priority * height	0.3 * 2	0.3 * 1	0.05 * 3	0.05 * 2	0.3 * 3

Total cost = 2.05

B> Cost of second tree.

Item	5	10	15	20	25
Priority	0.3	0.3	0.05	0.05	0.3
Height (or) No. of comparisons	2	1	4	3	2
cost = priority * height	0.3*2	0.3*1	0.05*4	0.05*3	0.3*2

Total cost = 1.85

By comparing the cost of the first tree and second tree, the second tree has a better average behavior.

AVL tree

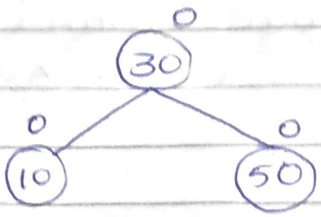
Adelson-Velsky and Landis

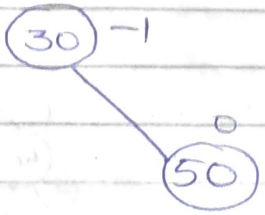
Definition: An AVL tree is a binary search tree in which the heights of the two subtrees of every node differ maximum by 1.

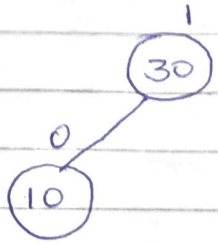
i.e. (height of left subtree) − (height of right subtree) can be 0, 1 or −1

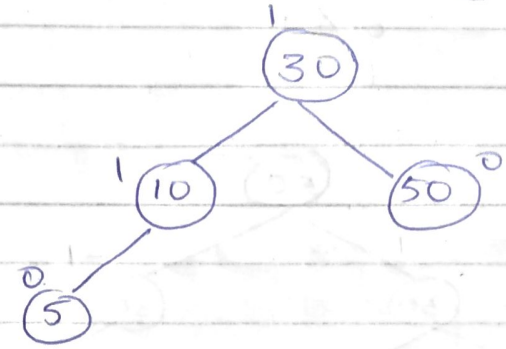
Balance factor = Height of left subtree − Height of right subtree

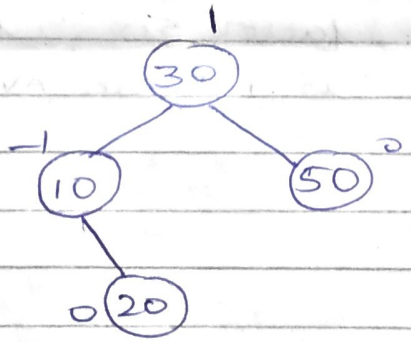
BALANCE FACTOR OF EACH NODE

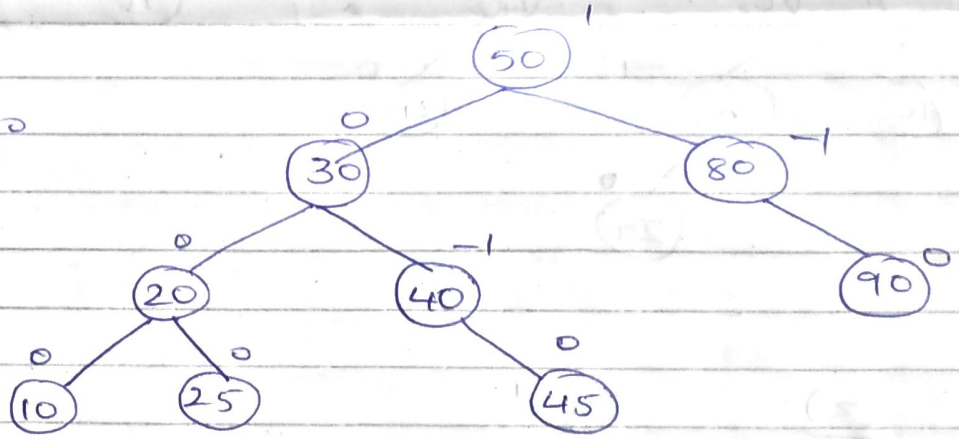






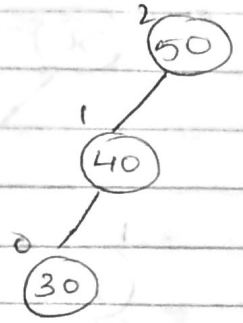




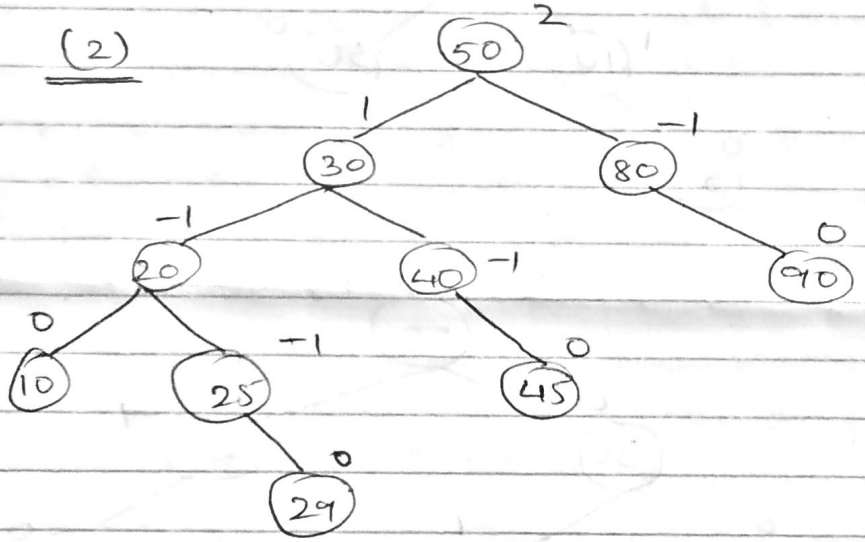


NOTE : All the above trees are BST and each node has a balance factor of 0, 1 or -1. ∴ All the above trees are AVL trees.

NOTE: (1) The following is a BST, but balance factor of 50 is 2. Hence it is not an AVL tree.

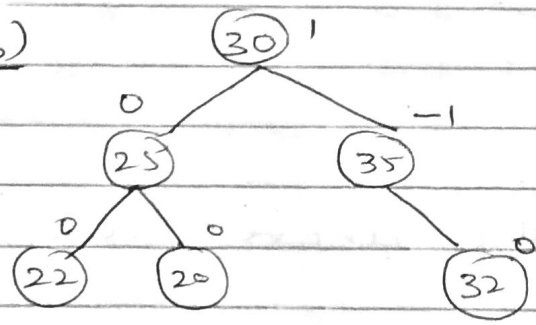


(2)



The tree is a BST but node 50 has balance factor 50 ∴ it is not an AVL tree

(3)



20 is inserted towards right of 25. Hence it is not a BST and hence not AVL tree.

PROPERTIES OF AVL TREE :-

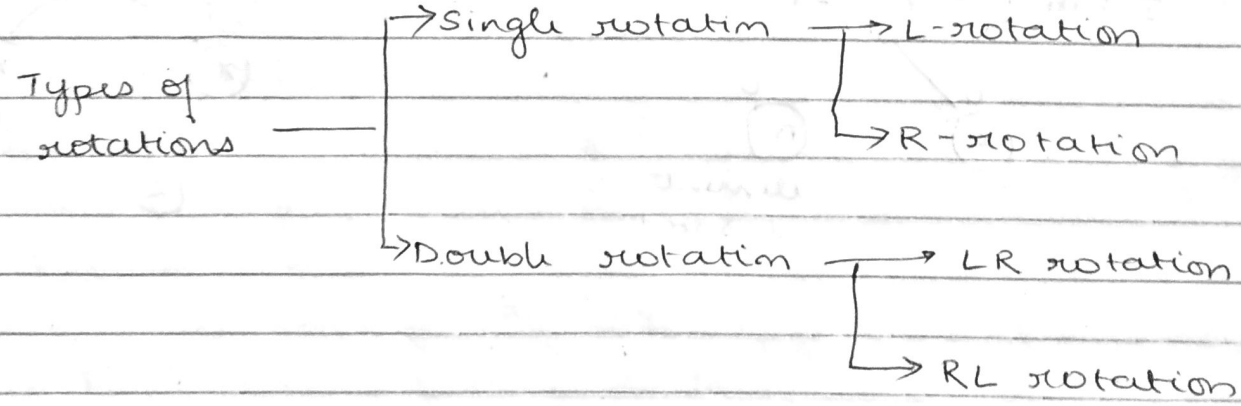
→ AVL trees are binary search trees.

→ Each node in an AVL tree has a balance factor of 0, 1 or −1.

Balance factor = Height of left subtree − height of right subtree.

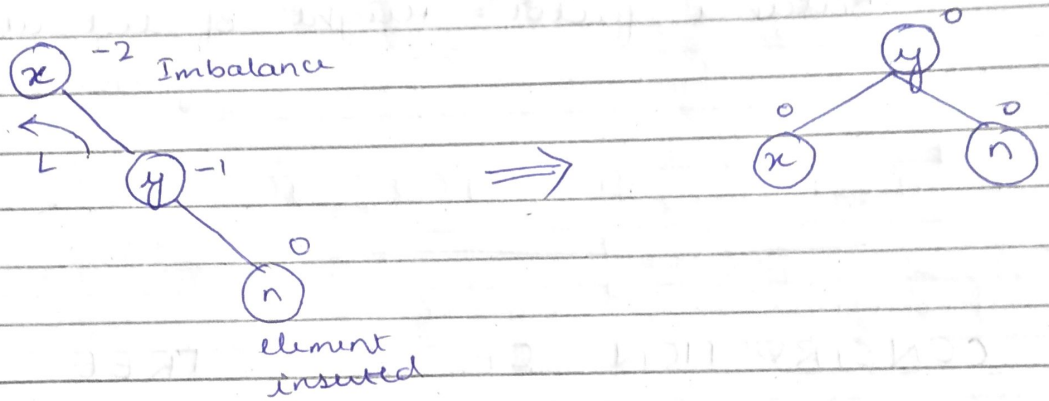
CONSTRUCTION OF AVL TREE

Once an element is inserted into the tree, the tree may be unbalanced. In such case, it is necessary to balance the tree. Balancing of AVL tree is done using rotations.

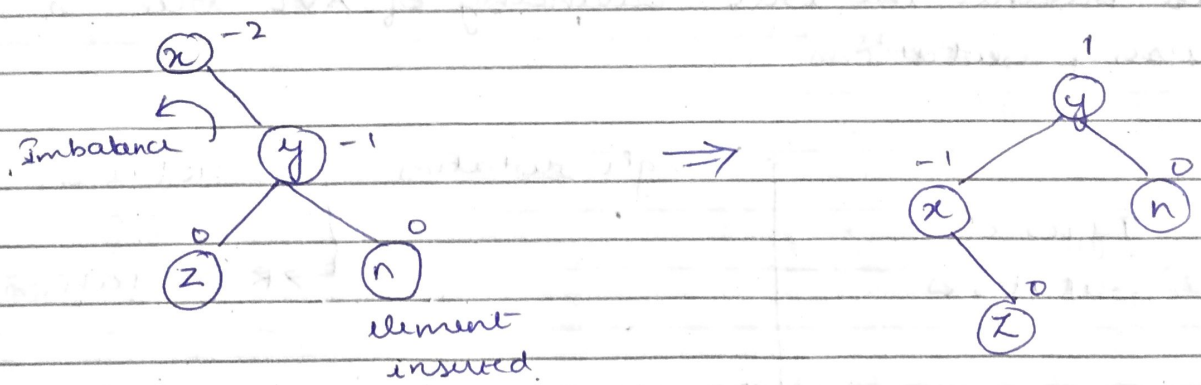


LEFT ROTATION

case 1: No left child for y

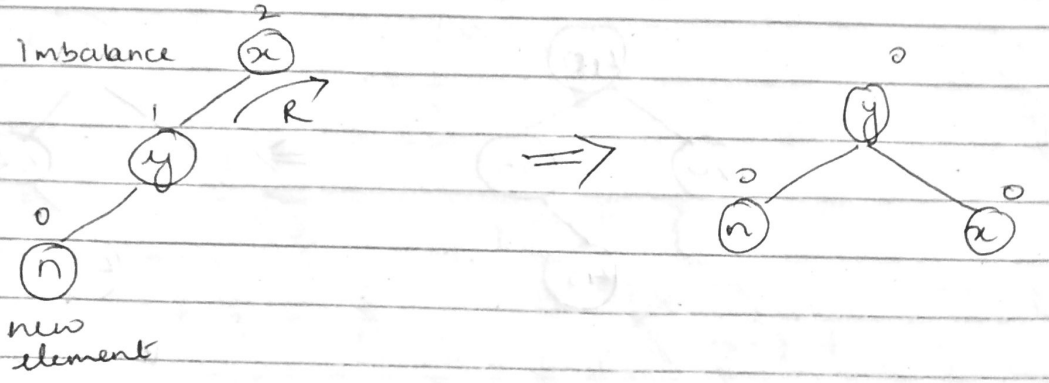


case 2: With a left child for y

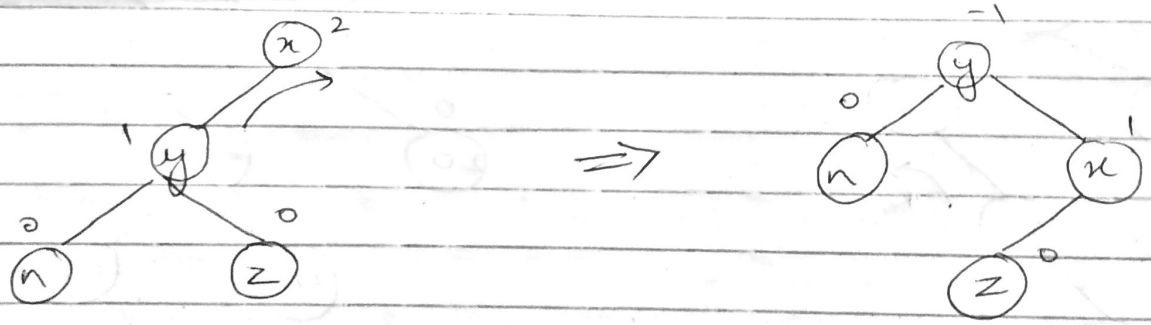


RIGHT ROTATION

case 1: No right child for y.

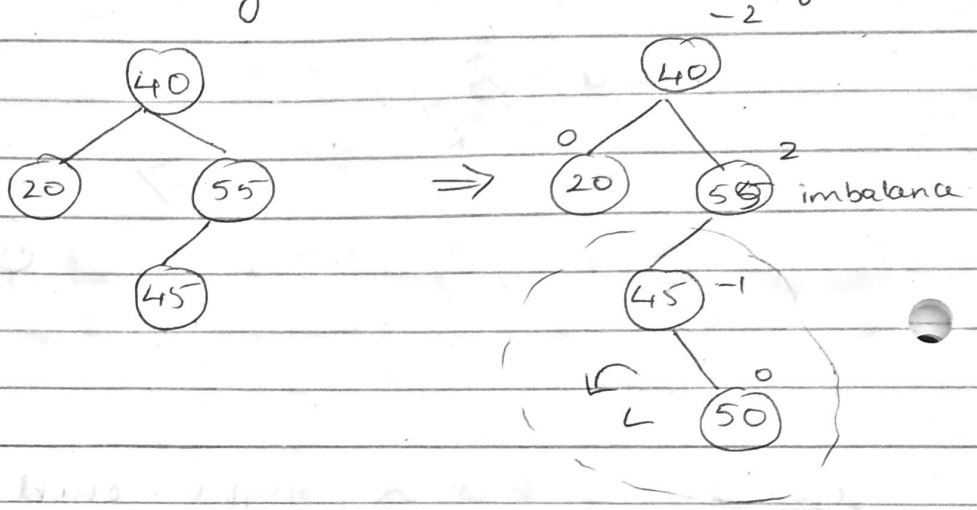


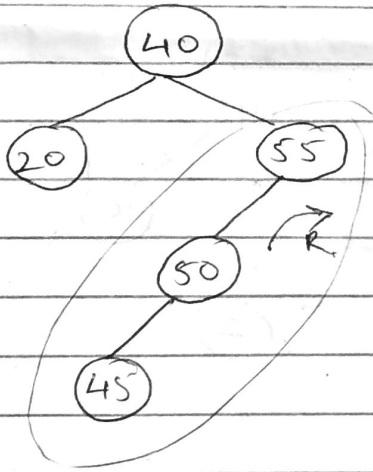
case 2: y has a right child.

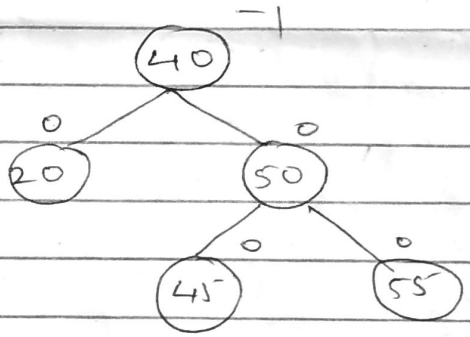


L-R Rotation

consider inserting 50 to the following AVL tree

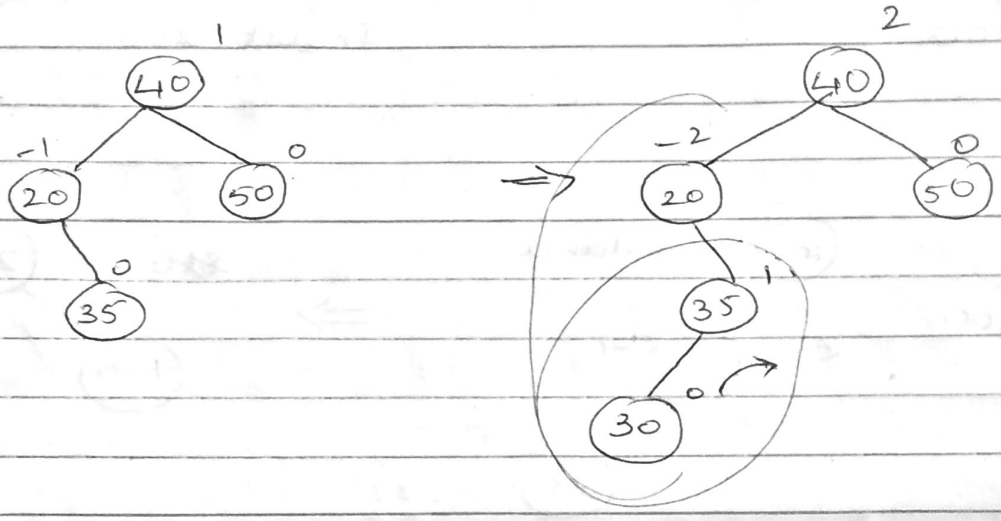


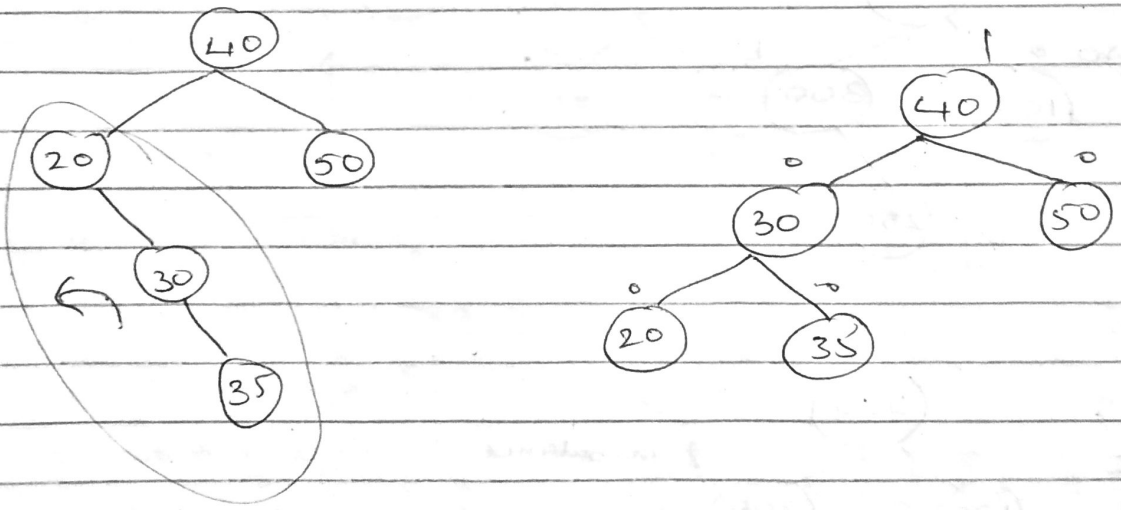




R-L ROTATION

consider adding 30 to the following AVL tree

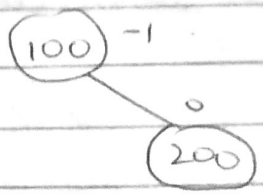




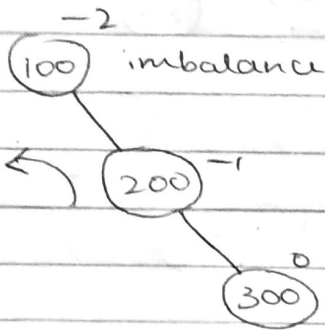
Construct an AVL tree for the following elements
100, 200, 300, 250, 270, 70, 40.

Step 1
Insert 100

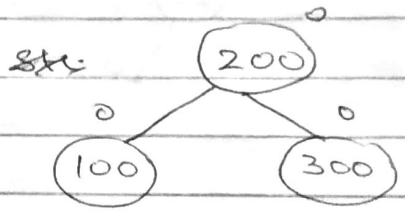
Step 2
Insert 200



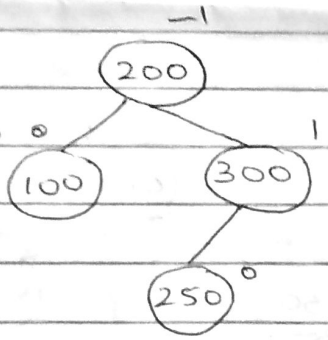
Step 3:
Insert 300



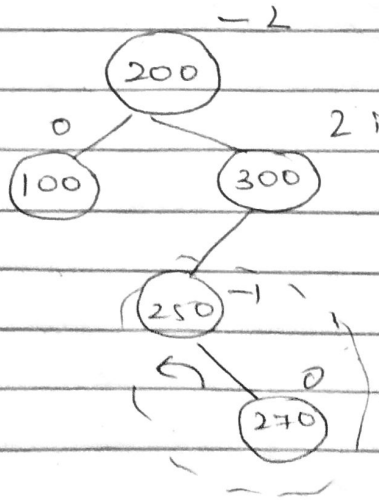
⇒

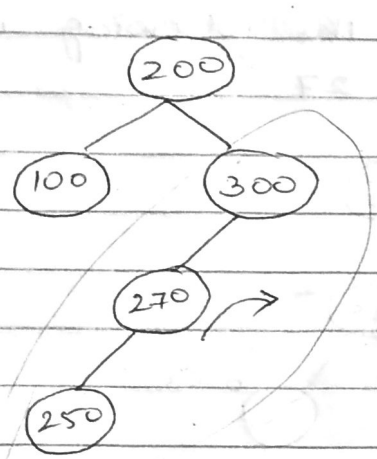


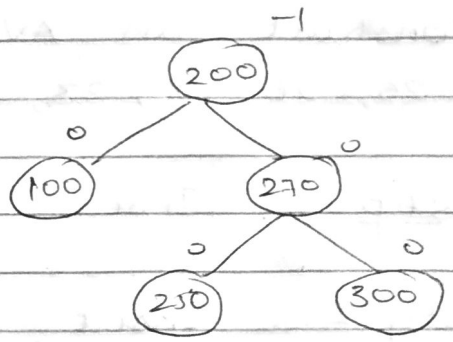
Step 4 :
Insert 250



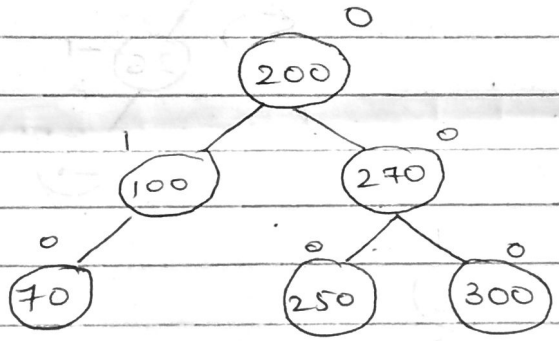
Step 5:
Insert 270



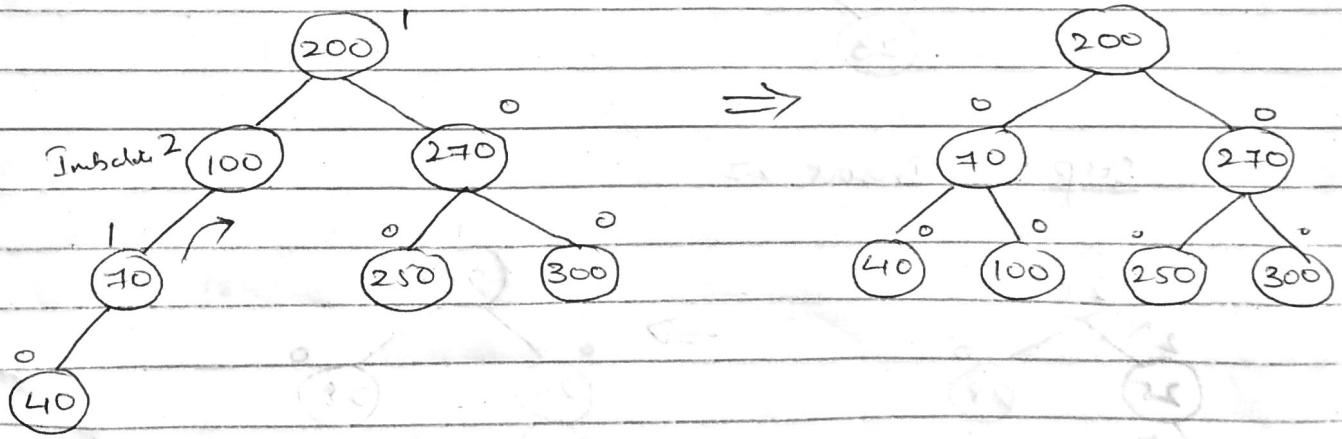




Step 6 : Insert 70



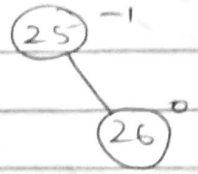
Step 7 Insert 40



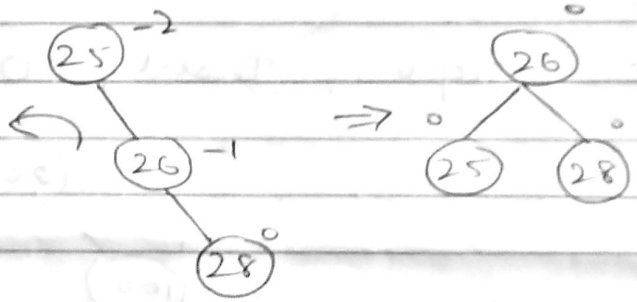
Construct an AVL tree for the following integers 25, 26, 28, 23, 22, 24 and 27.

Step 1 : Insert 25

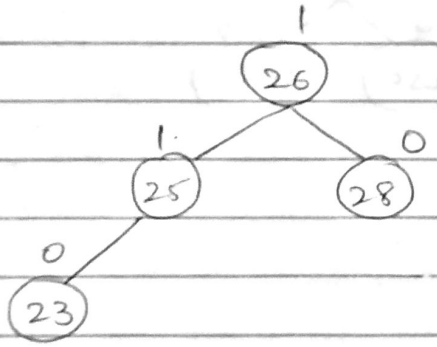
Step 2 : Insert 26



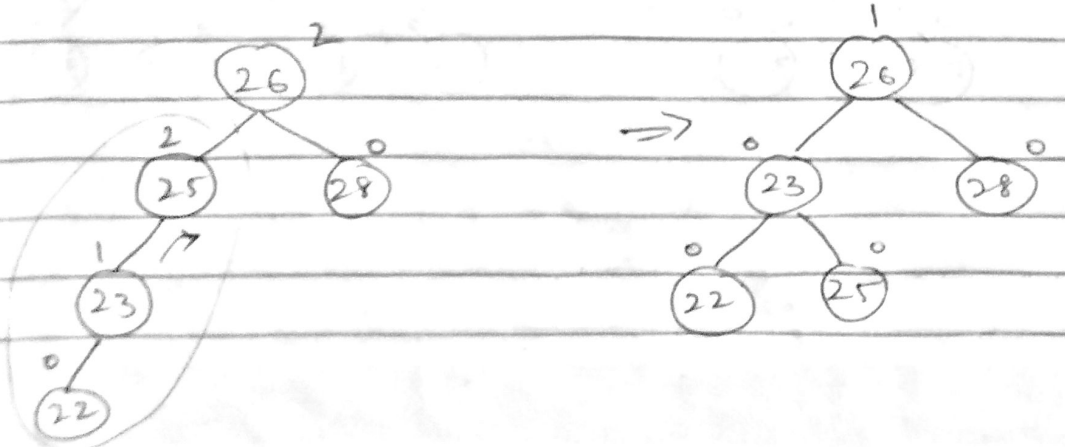
Step 3 : Insert 28



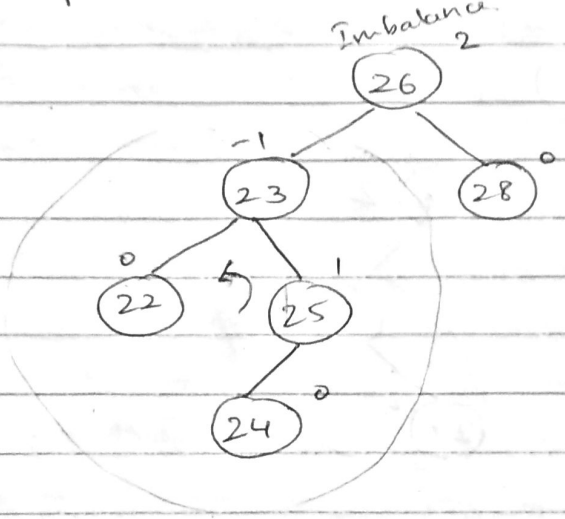
Step 4:
Insert 23



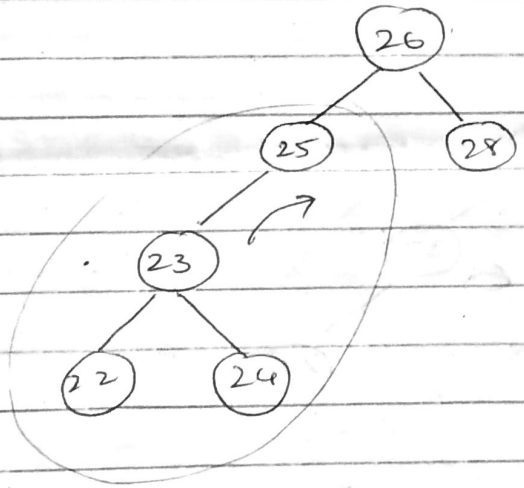
Step 5 Insert 22



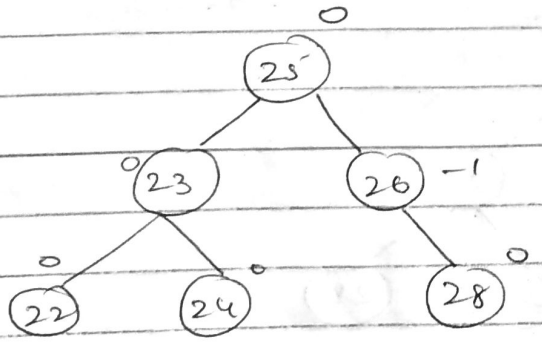
Step 6: Insert 24



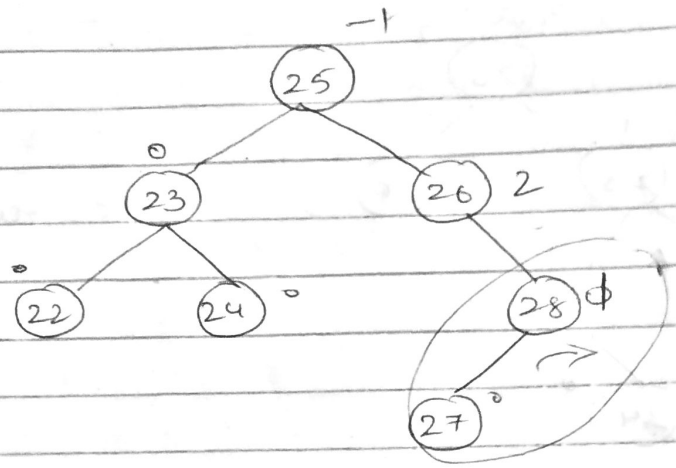
Left rotate at 23

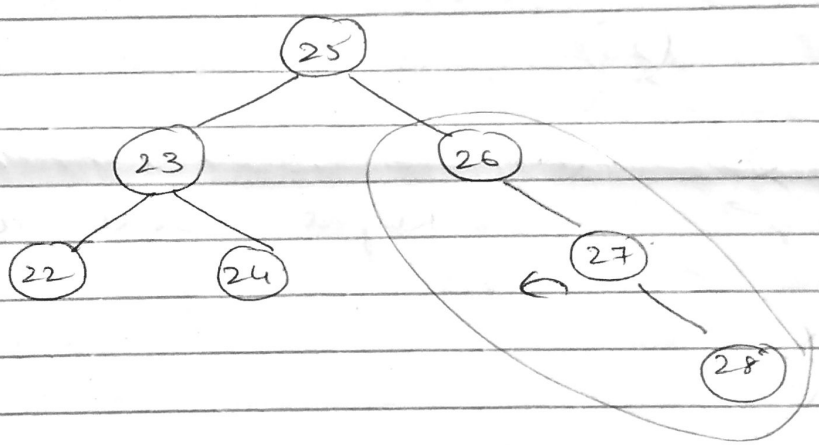


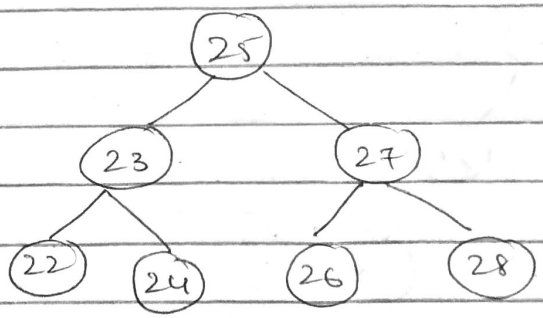
Right rotate at 26.



Step 7 : Insert 27.



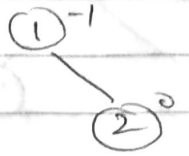




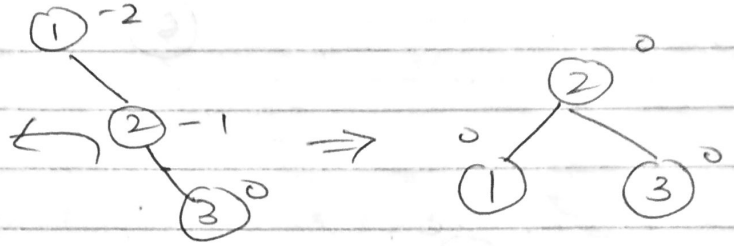
construct an AVL tree for the following elements
1, 2, 3, 4, 5, 6

Step 1: Insert 1.

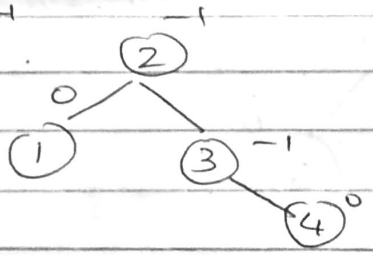
Step 2: Insert 2



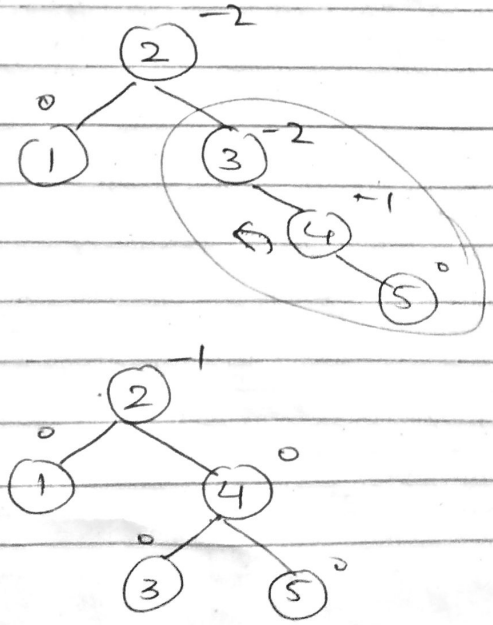
Step 3: Insert 3



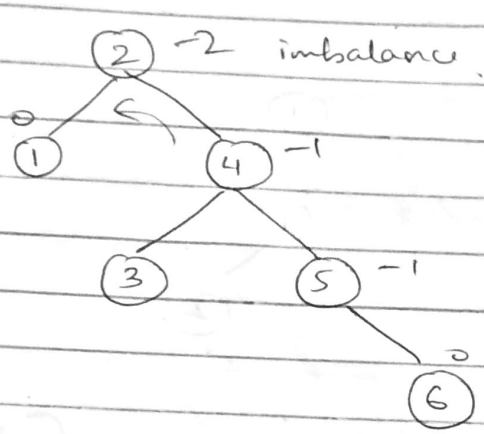
Step 4: Insert 4

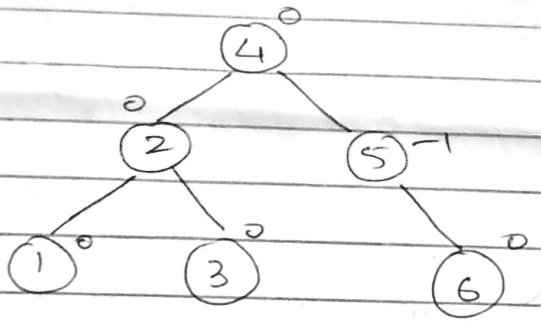


Step 5: Insert 5



Step 6 : Insert 6





RED - BLACK TREES

Red - black trees is an efficient binary search tree with each node coloured red or black and also satisfies the following properties.

- Root is always black
- There are no Red - Red conflict in a parent child relation
- Number of black nodes in all the paths are equal.

Insertion of a node to Red black trees :-

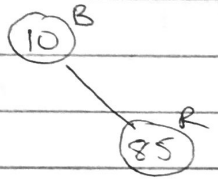
1. If tree is empty create black root node
2. Insert new leaf node as Red.
   (a) If parent is black then done
   (b) If parent is red
   → if parents' sibling is black or absent rotate and recolour
   → if parents' sibling is red the recolour and check again.

construct a red-black tree for the following elements.

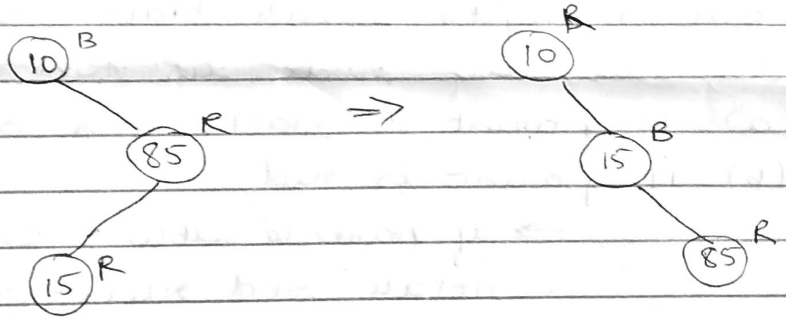
10, 85, 15, 70, 20, 60, 30, 50

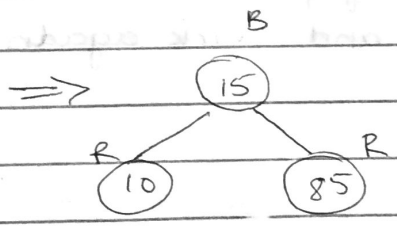
Step 1:

Step 2:

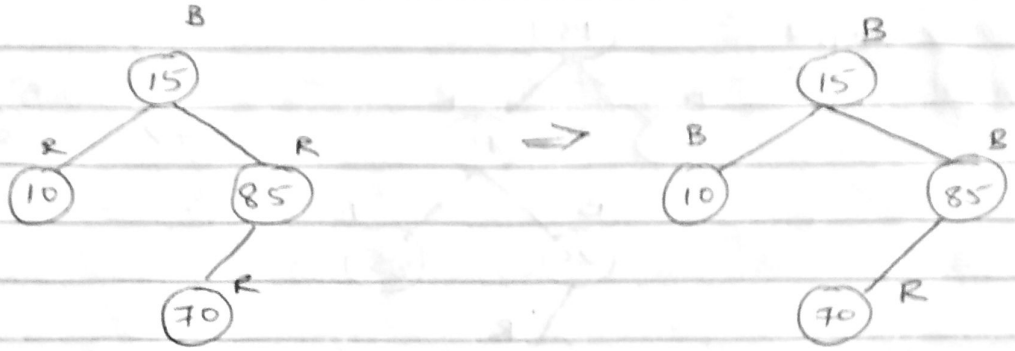


Step 3:

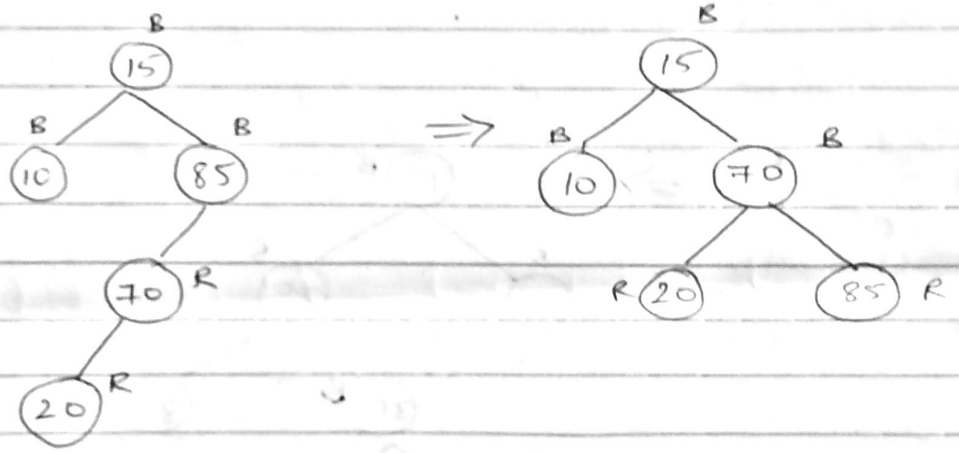




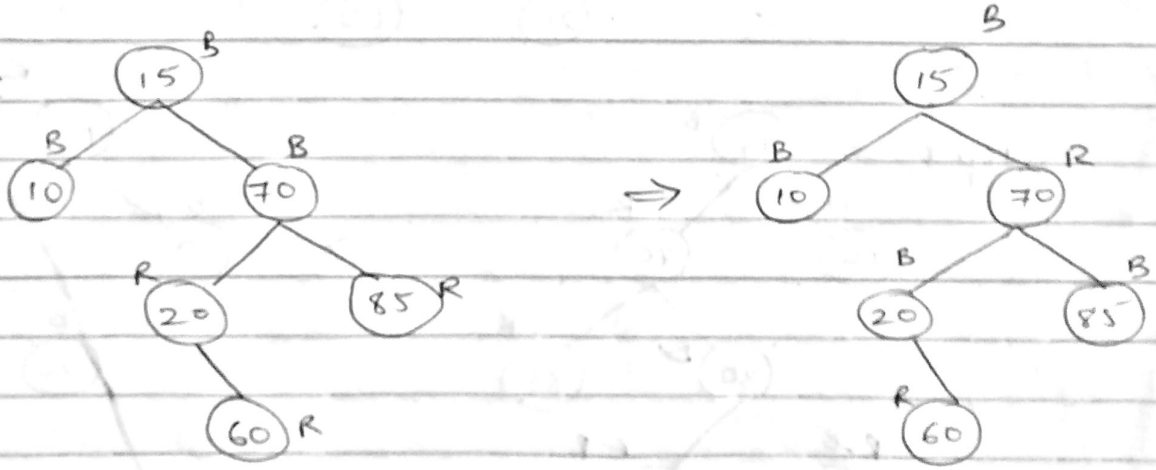
Step 4



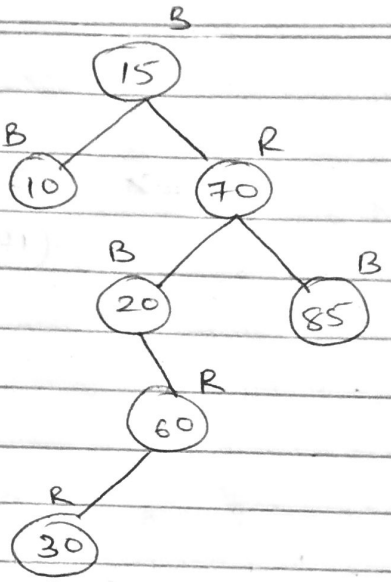
Step 5:

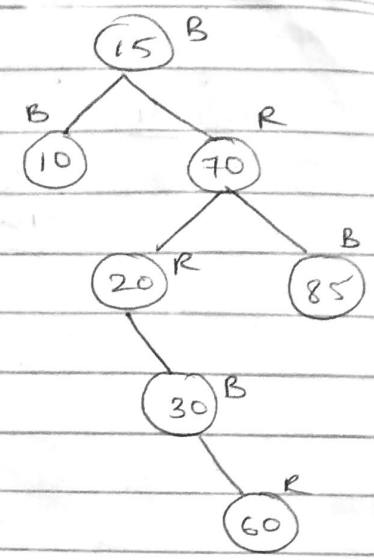


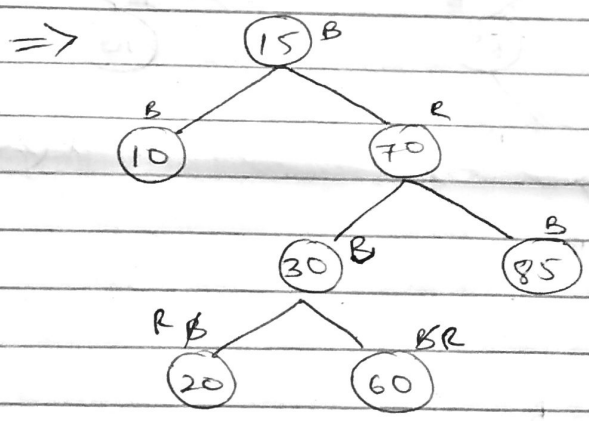
Step 6:



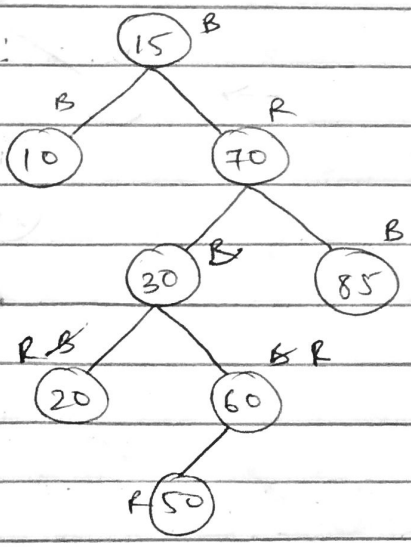
Step 7:

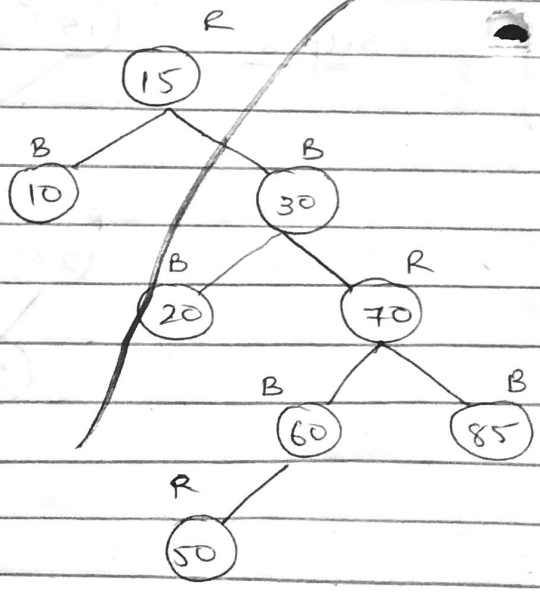

B
15
B
10
R
70
B
20
B
85
R
60
R
30

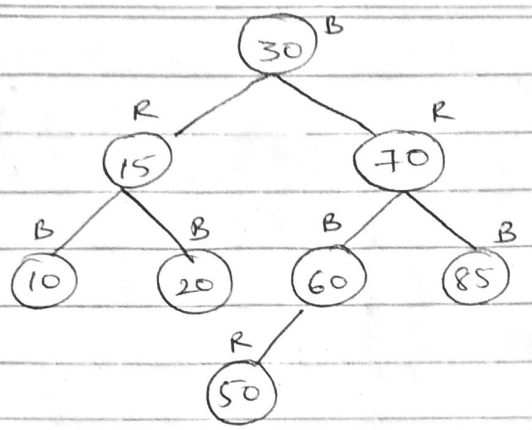

15
B
B
10
R
70
20
R
B
85
B
30
R
60


15
B
B
10
R
70
B
30
B
85
R
20
B
B
R
60

Step 8:


15
B
B
10
R
70
B
30
B
85
R
20
B
B
R
60
R
50


R
15
B
B
10
B
30
B
20
R
70
B
B
60
85
R
50


30
B
R
R
15
70
B
B
B
B
10
20
60
85
R
50