Efficient BST:

Types of BST:
- Optimal BST
- AVL trees
- Red-Black trees
- Splay trees

Optimal BST:

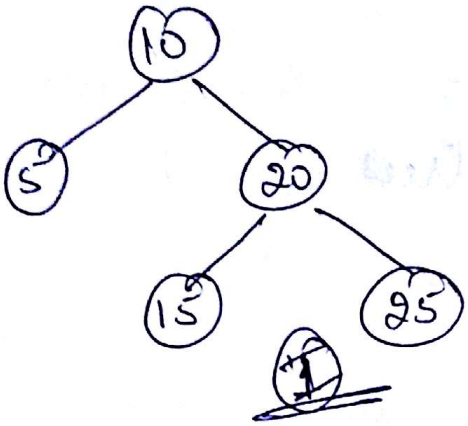
Defn: for a BST 'c' fixed no of elements & the probability of each key, the avg expected cost of accessing the elements in a tree can be computed.

An O-BST is a BST 'w' has minimal or least expected cost (less comparisons, less height, complete BT)

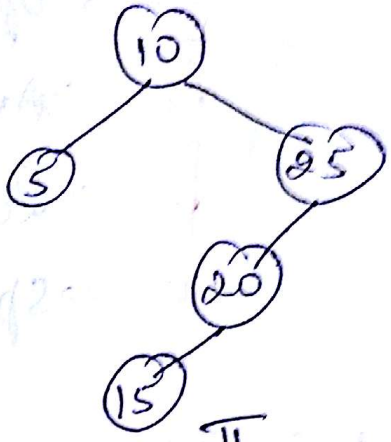
This BST is well suited for fixed set of elements.

Eg: Find expected cost (avg. no of comparisons) for the following trees 'c'

Eg: (1) Find the avg no of comparisons (expected cost) for the following trees



I



II

Keys:	5	10	15	20	25
Prob:	1/5	1/5	1/5	1/5	1/5
Priority:	0.3	0.3	0.05	0.05	~~0.25~~ 0.3

Soln

Part A: Estimated cost is:

Case 1: Consider Tree I

Item	5	10	15	20	25	
Probability	1/5	1/5	1/5	1/5	1/5	
Level of item / No of comparisons	2	1	3	2	3	
Cost = Prob × comparisons	$\frac{1}{5}\times 2$	$\frac{1}{5}\times 1$	$\frac{1}{5}\times 3$	$\frac{1}{5}\times 2$	$\frac{1}{5}\times 3$	

$$\text{Total} = \frac{2}{5}+\frac{1}{5}+\frac{3}{5}+\frac{2}{5}+\frac{3}{5}$$

$$= \frac{11}{5} = 2.2$$

∴ Cost of first tree = 2.2

Better avg. behaviour

Case 2: Consider second tree

Item	5	10	15	20	25
Prob.	1/5	1/5	1/5	1/5	1/5
Level/comparisms	2	1	4	3	2
Cost	$\frac{1}{5}\times 2$	$\frac{1}{5}\times 1$	$\frac{1}{5}\times 4$	$\frac{1}{5}\times 3$	$\frac{1}{5}\times 2$

$$\text{Total} = \frac{2}{5}+\frac{1}{5}+\frac{4}{5}+\frac{3}{5}+\frac{2}{5}$$

$$= \frac{12}{5} = 2.4$$

∴ Cost of second tree = 2.4

Part B: Consider the first tree

Item	5	10	15	20	25
Priority	0.3	0.3	0.05	0.05	0.3
Level/comp.	2	1	3	2	3
Cost = Priority × Level	0.6	0.3	0.15	0.10	0.9

10
+ 5
60
30
90
2.05

Total = 2.05

∴ Cost of first tree = 2.05

Consider second tree:

Item	5	10	15	20	25
Priority	0.3	0.3	0.05	0.05	0.3
Level / comp	2	1	4	3	2
Cost	0.6	0.3	0.2	0.15	0.6

Total = 1.85

∴ Cost of second tree = 1.85

Better avg. behaviour

Note: There may be another tree consisting of same set of elements which can have better avg. behaviour when compared to both.

Even though both trees have same elements, the tree structures are different & the avg. cost of accessing the element is different.

AVL T

Defn: An

subtrees

height o

This c

binary

It

In

balanc

If

Eg:

AVL Tree — Russian G.M Adelson-Velsky & E.M-Landis

Defn: An AVL tree is a BST in which heights of the 2 subtrees of every node differ max. by 1 i.e

height of left subtree − height of right subtree can be 0, 1 or −1

This condition should be satisfied by each node in the binary tree — then AVL tree.

It is a height balanced BST

In AVL tree, each node is associated with a balance factor.

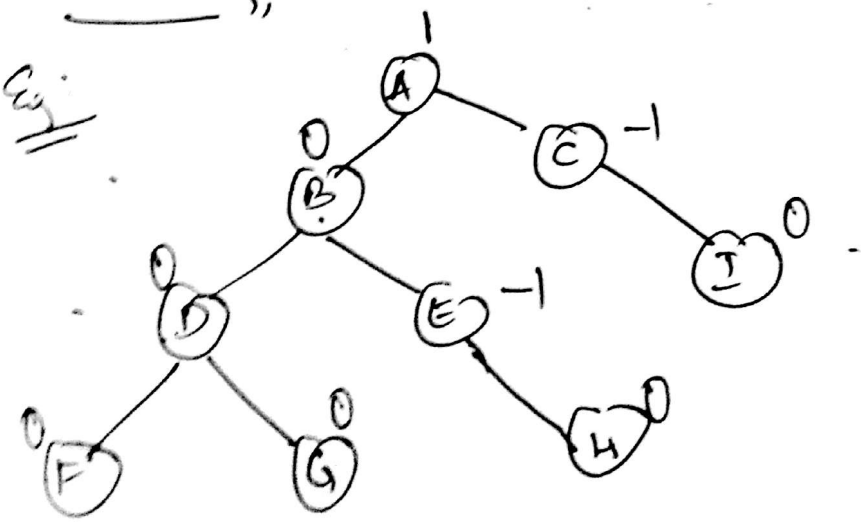
> Balance Factor = Height (left subtree) − Height (right subtree)

If same, bf = 0

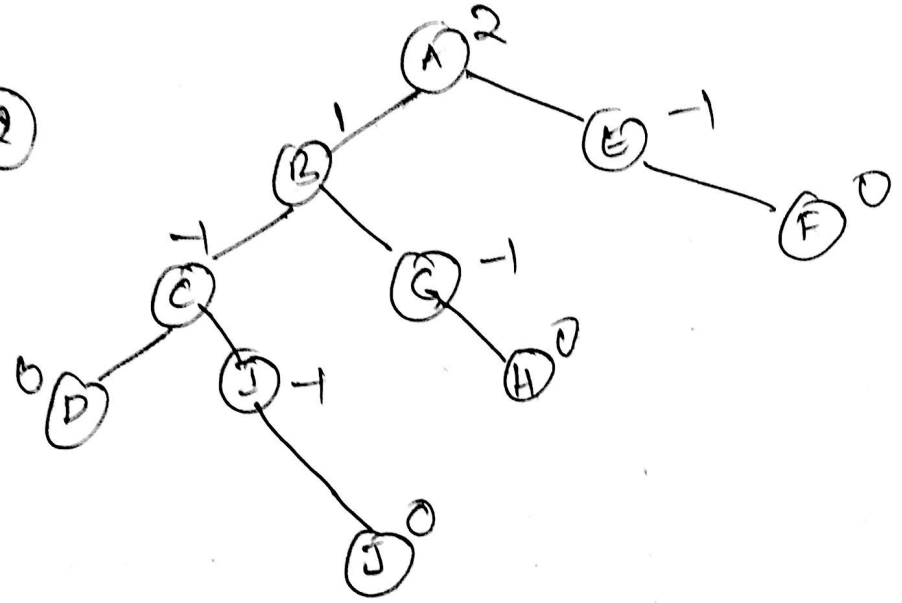
If ht (ls) is 1 more than ht(rs) ⇒ bf = 1

" " −1 less " " , bf = −1

Eg:

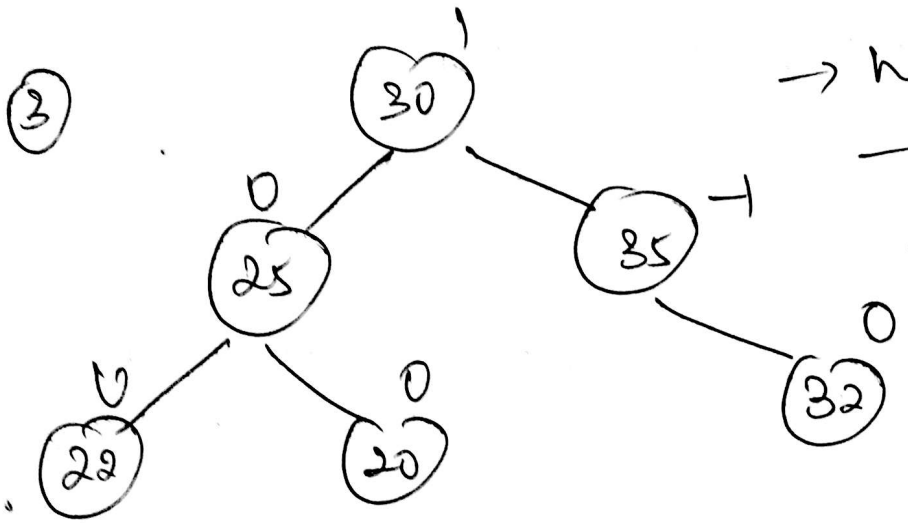


(2)



→ not AVL

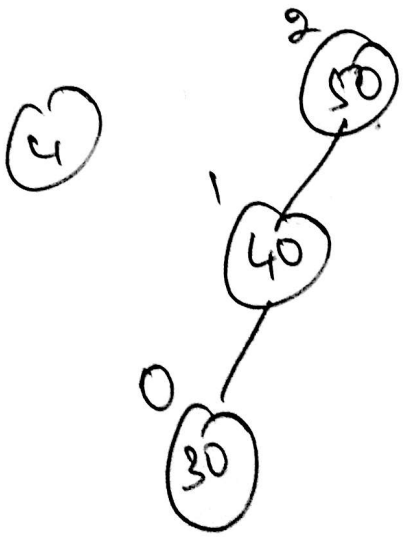
(3)



→ not BST

→ ∴ not AVL

(4)



→ not BST

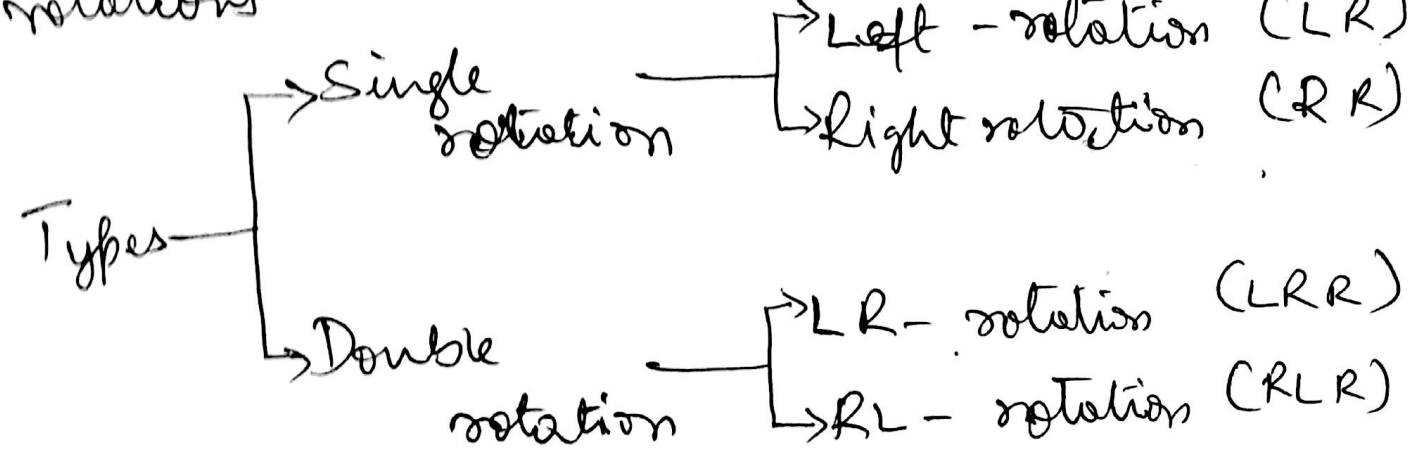
→ not AVL

Types of Rotations :

If each node has a balance factor of 0, 1 or -1, then such a tree is height-balanced & it becomes an AVL tree.

To make it happen, we have to rearrange the nodes.

Once an item is inserted, the tree may be unbalanced. Balancing of AVL trees is done using rotations.



When is rotation required?

After inserting a new node, trace backwards towards root & compute bf, if a node has bf = other than 1, -1 or 0, then rotation is required.

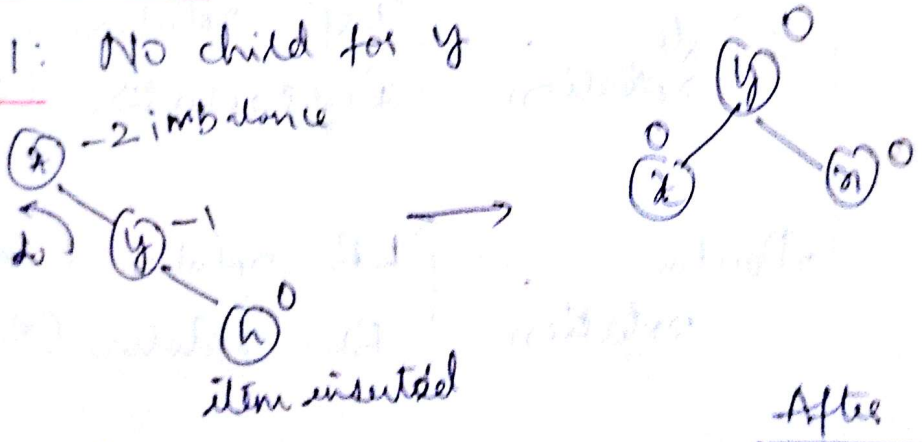
Procedure to identify LR or RL

Step 1: Identify the first node where balance factor is not 0, 1, -1 in the path from node inserted towards root.

Step 2: Identify 2 other nodes below the node where imbalance occurs in the path.

Left Rotation

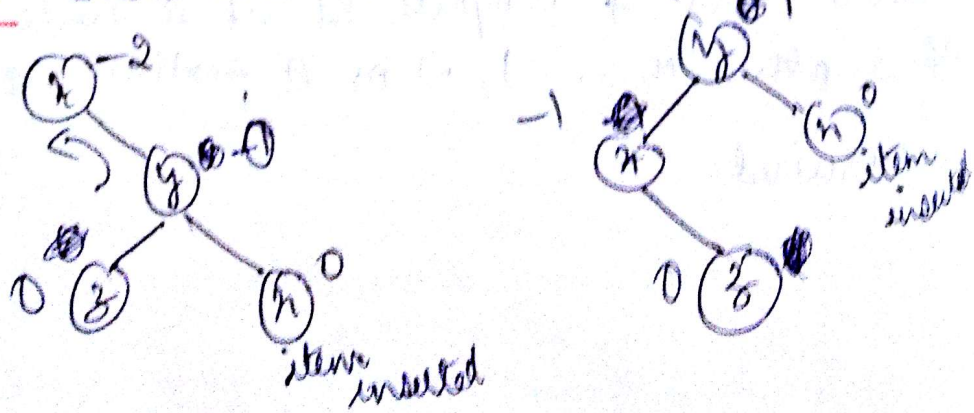
Case 1: No child for y



Before

After

Case 2: There is a left child for y.



Right Rotation

Case 1: N

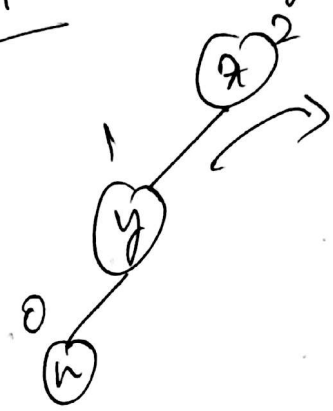
Case 2:

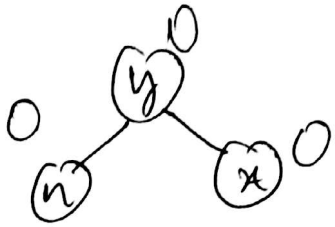
LR

Scanned by CamScanner

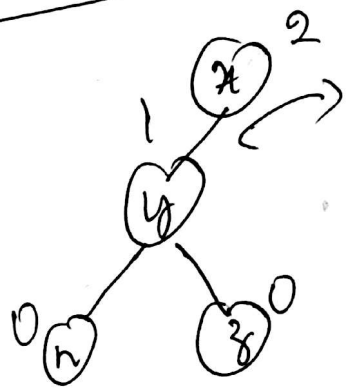
Right Rotation

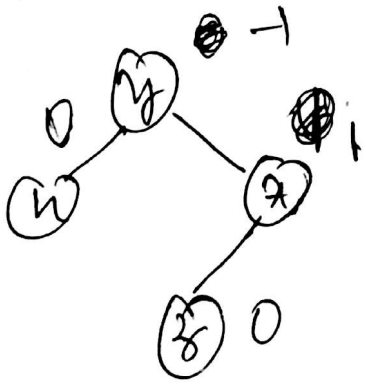
Case 1: No right child for y.





Case 2: Right child for y.





LR Rotation

Combination of 2 single rotations:

(1) Assume x is a node where inbalance occurs & y is its child in the path.

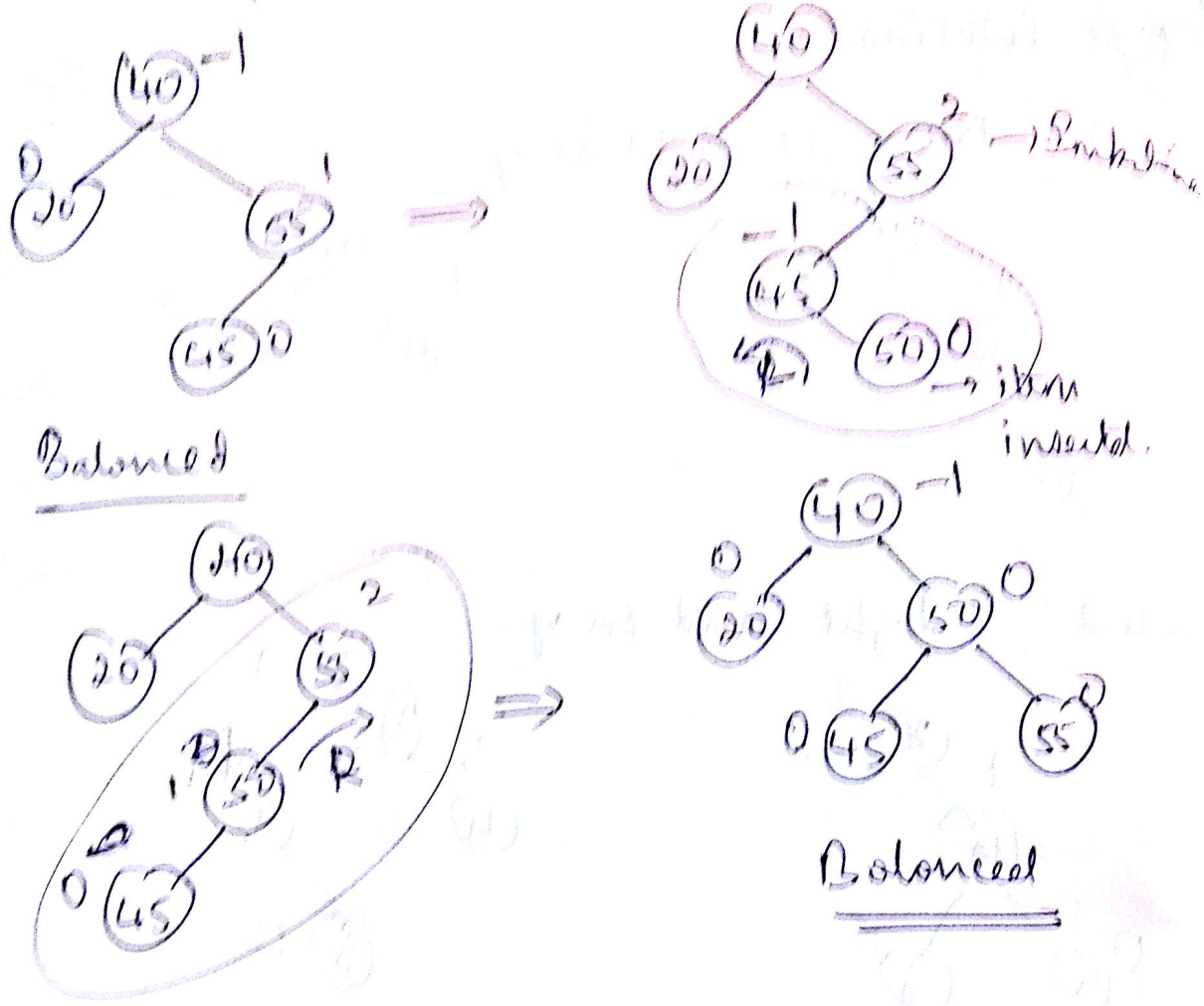
If bf of y = −1 ⟶ right heavy.

∴ Left Rotate y.

Attach the parent of resulting subtree to x.

(2) 2nd rotation @ x.

∴ Right rotation.



R L Rotation — Combination

(1) Assume x is a node where imbalance happens.

y is its child.

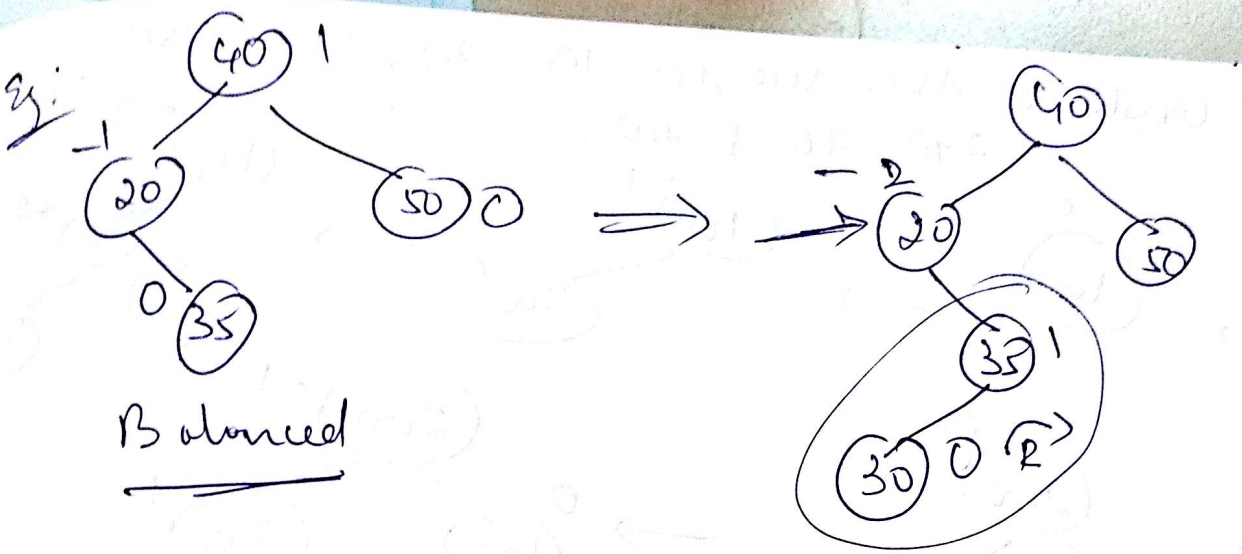
if bf(y) = 1 → left heavy.

∴ Right rotate y.

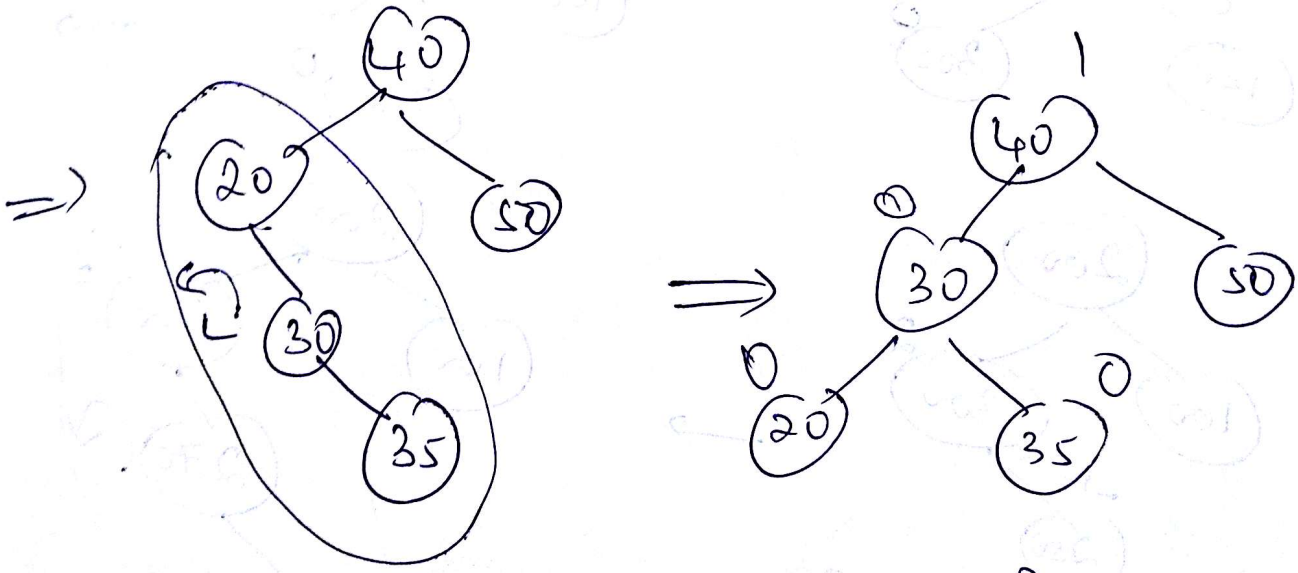
Attach parent of subtree to x.

(2) and rotation on x.

Eg:

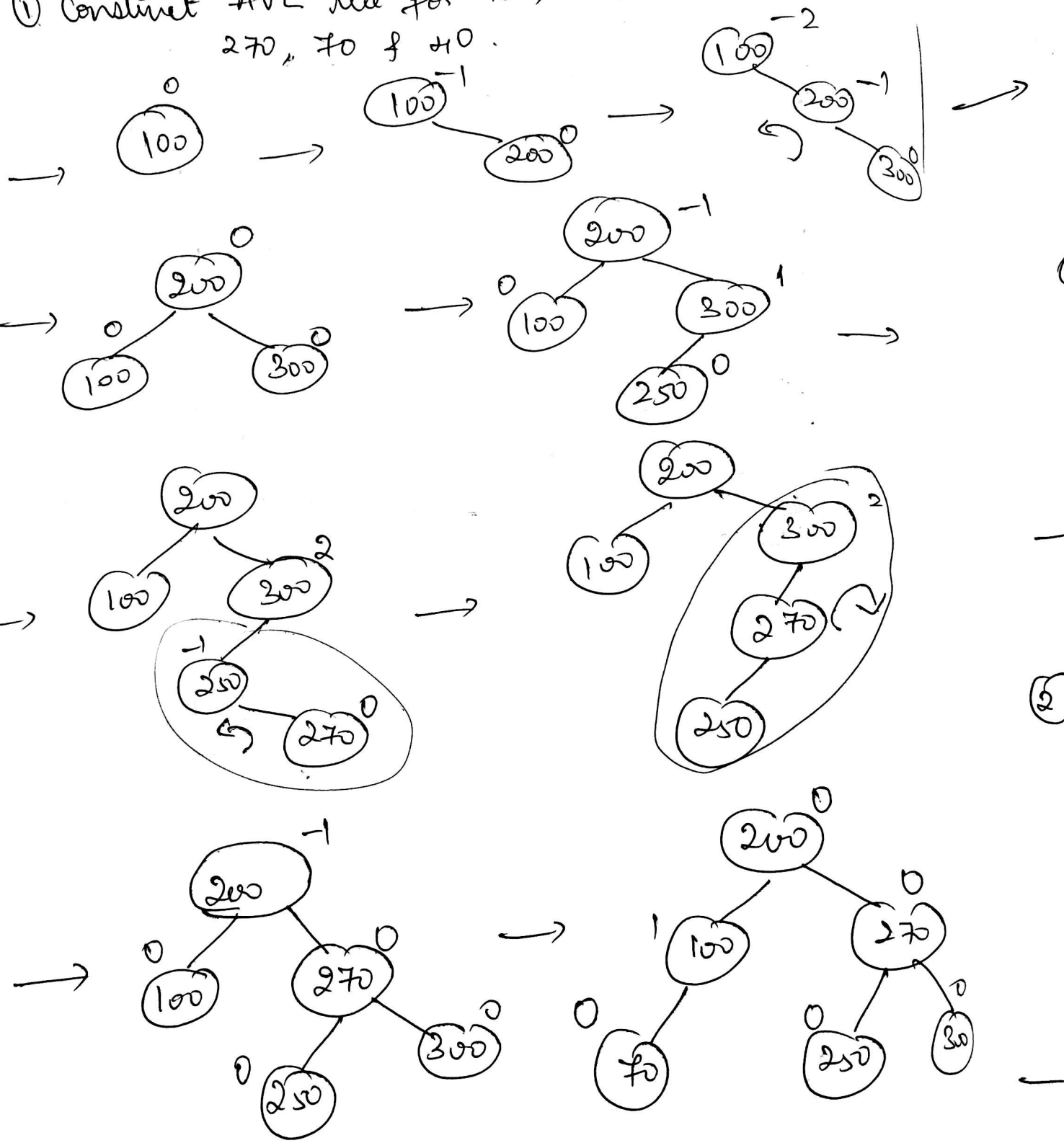


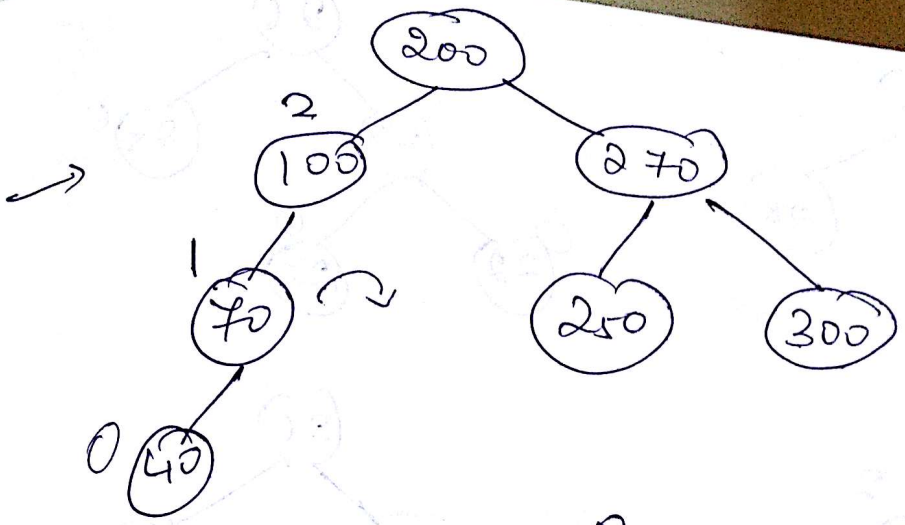
Balanced

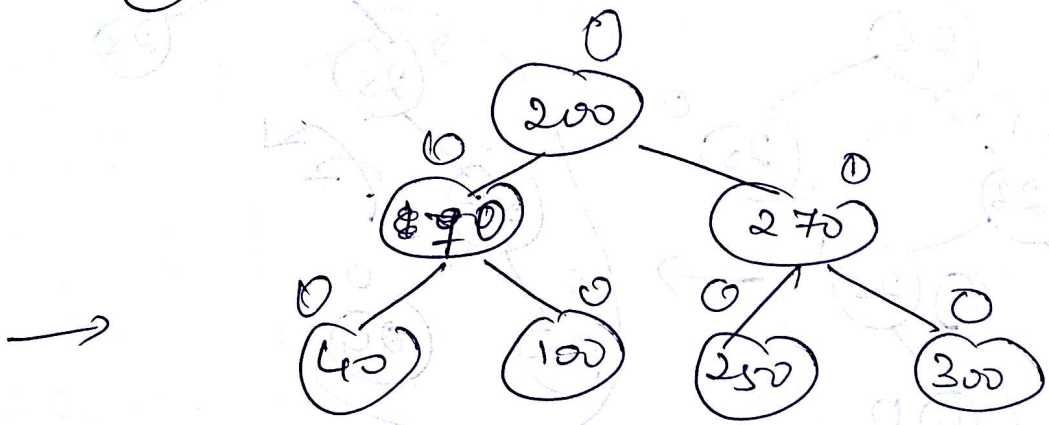


Balanced

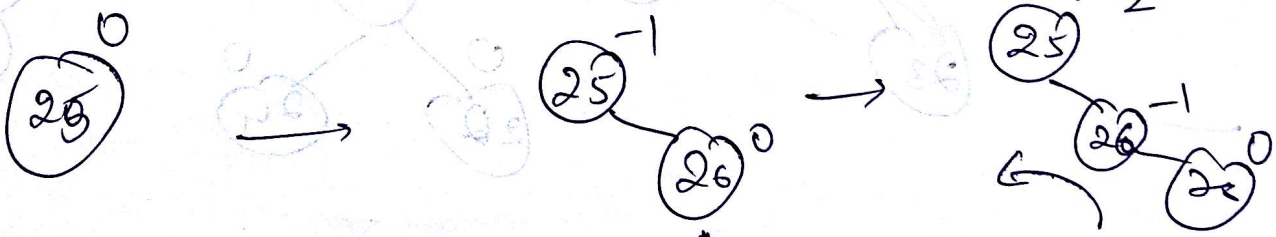
① Construct AVL tree for 100, 200, 300, 250, 270, 70 & 40.

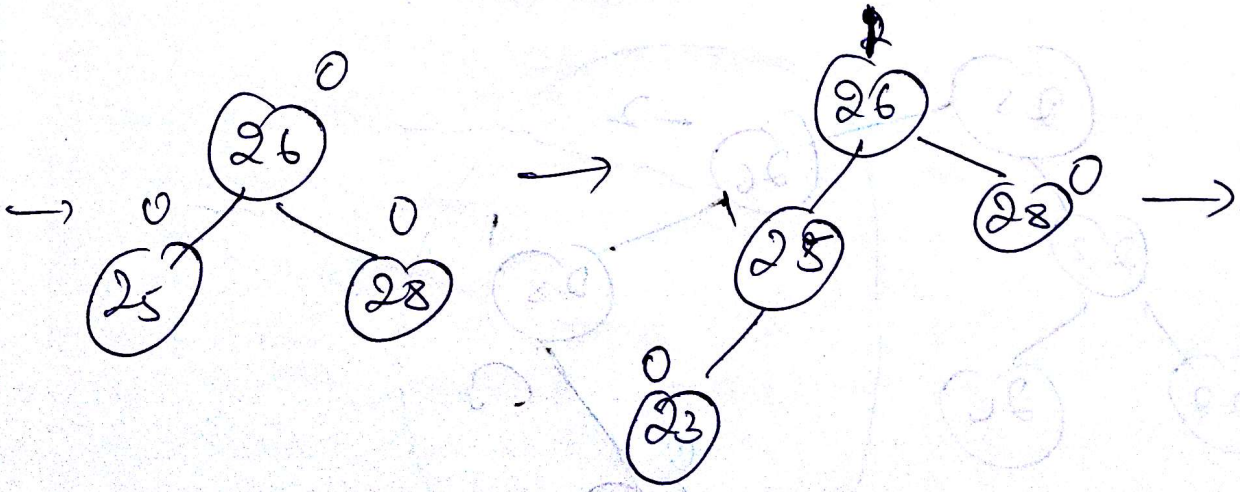






(2) Construct AVL tree → 25, 26, 28, 23, 29, 24 & 27

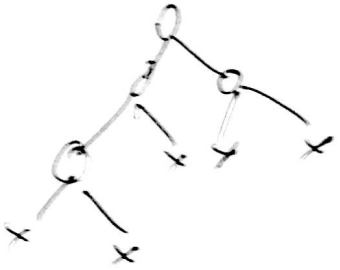




Red Black Trees

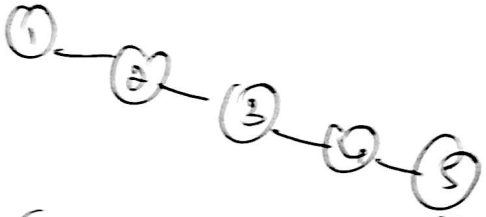
Properties:

(1) It is a Binary Search Trees



key(L child) ≤ key(root) ≤ key(R right child)

Ex:



To balance something like this, we have AVL, RBT etc.

(2) (1) Every node is either Red or Black.

(2) Every nil node is Black.

(3) Every Red node has 2 Black child nodes (A red node ~~CANNOT~~ have a Red child node)

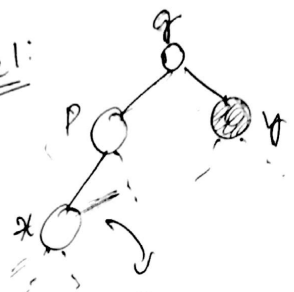
(4) Every path from node x down to a leaf node has the same no of Black nodes

(1) (5) Root node is always Black.

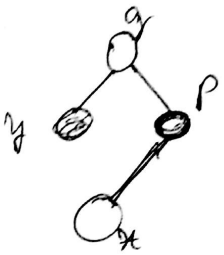
Violations

y = uncle.

Case 1:



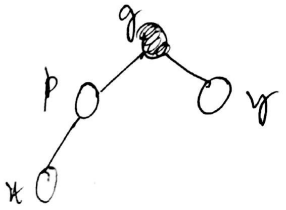
UNCLE = Red

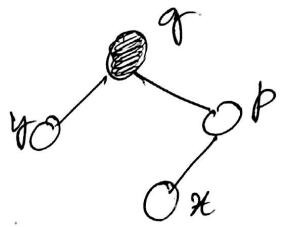


Red node cannot have a Red child

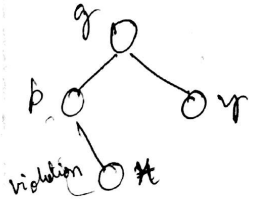
If a red node has a red child, & if uncle = Red, change the color of the G, P & y

After change of ~~color~~ color:

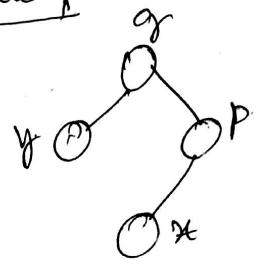




Case 2:



UNCLE = Black

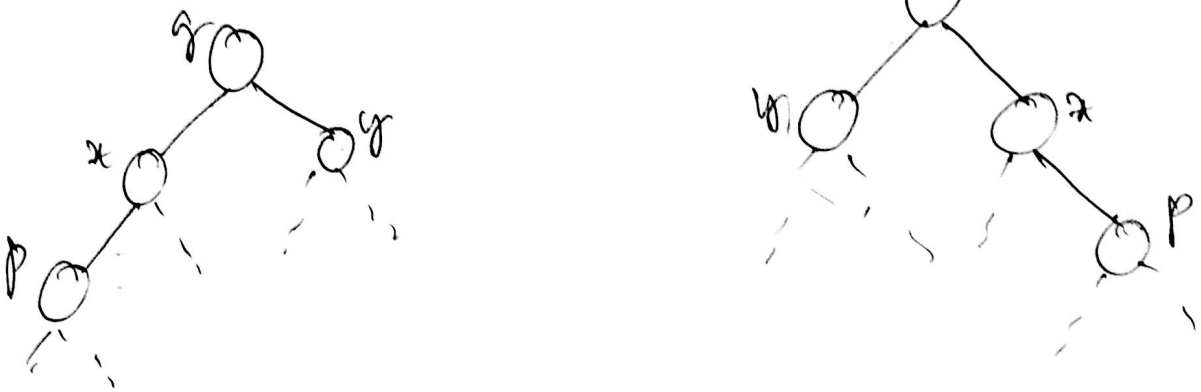


x = right child of a left child

OR

x = left child of a right child.

Fix: Rotate parent → leads to Case 3

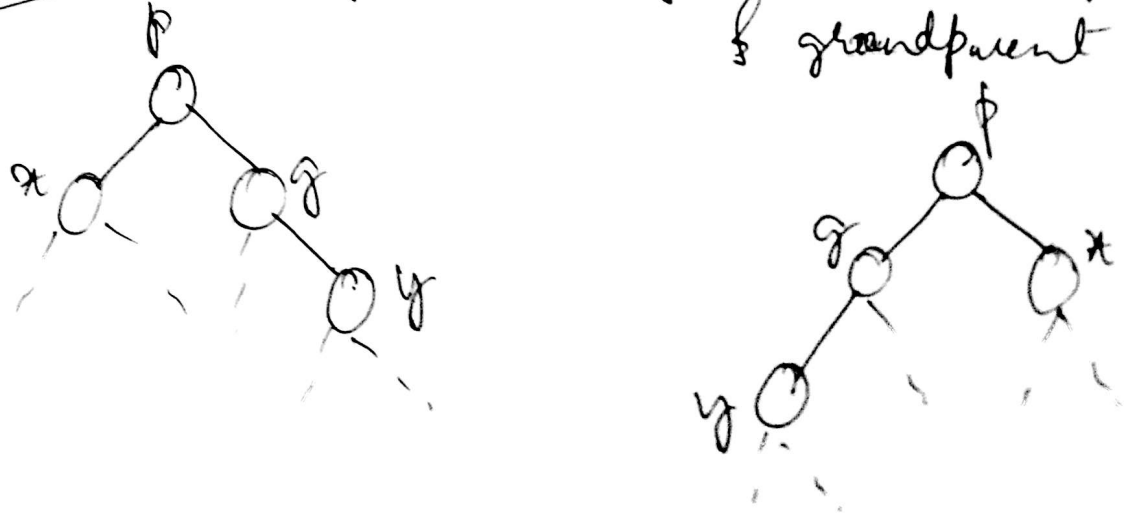


(3)

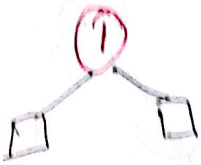
UNCLE = Black

Case 3:

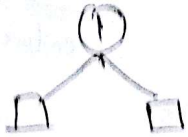
Fix: Rotate grandparent & change color of Parent & grandparent



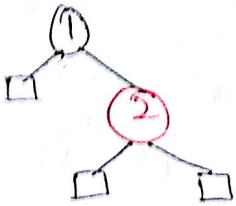
Eg (1) 1 2 3 4 5 6



Colors ⇒

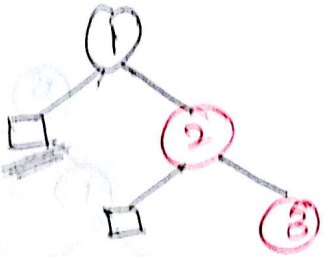


(2) ⇒

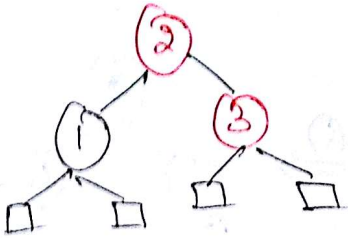


NULL nodes

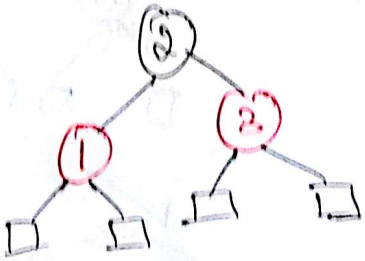
(3) ⇒



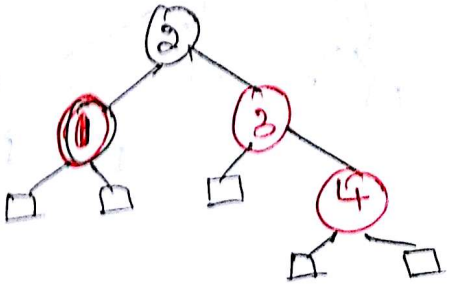
Case 3 ⇒ Left rotate of gp



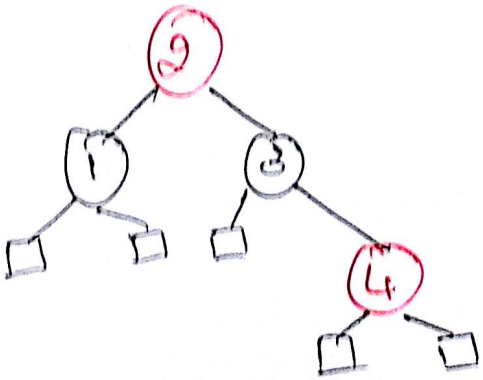
Change colors ⇒



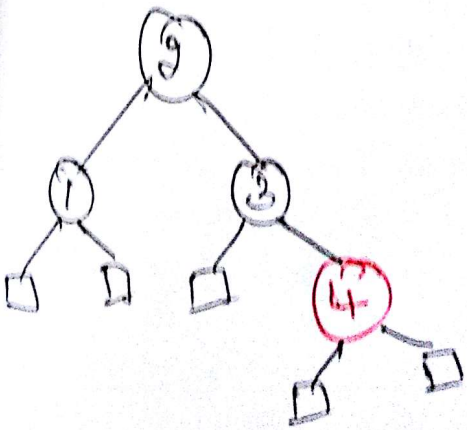
(4) ⇒



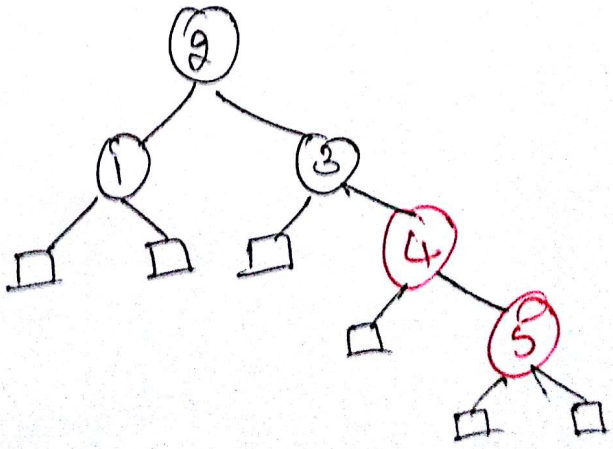
Case 1 ⇒ Uncle = Red (Colors)



Root ⇒ should be black



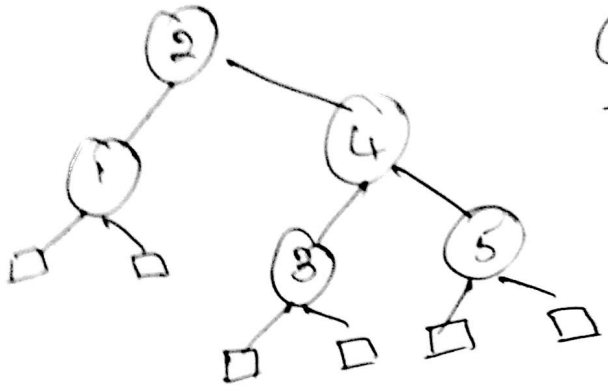
(5) ⇒



DA.

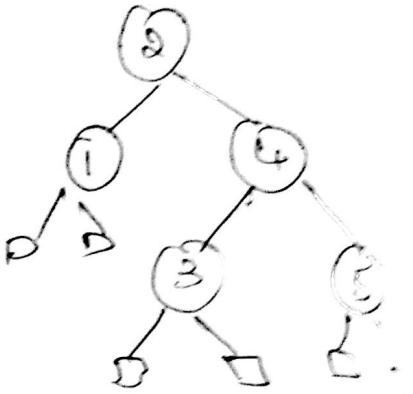
Case 3

Invoke: RR

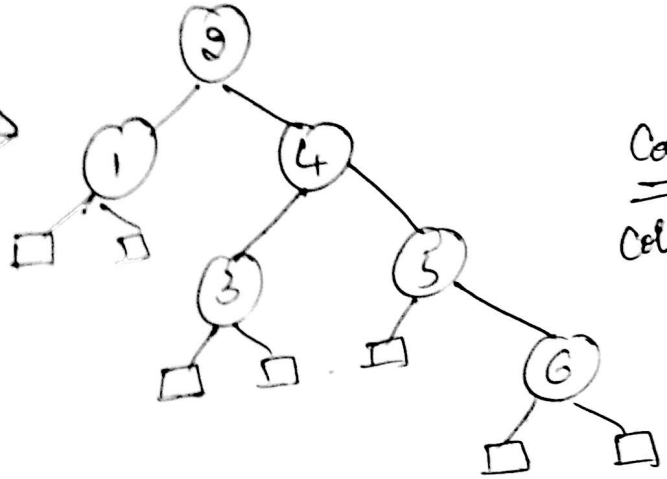


G, P

color



(6)



Case 1

color(G, P, y)