B-Trees

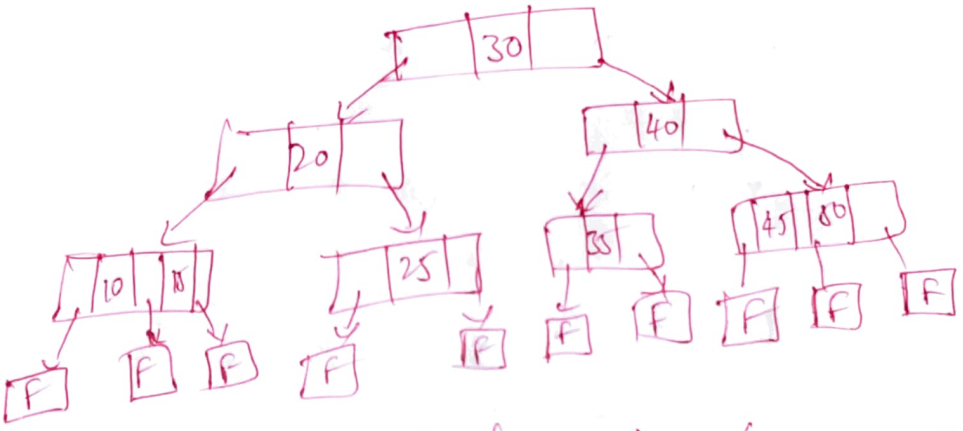
→ The BST uses tree indexing, where node may have key value, pointers to left & right subtrees.

→ B tree is a Balanced tree T of order m, ie, either it is empty (or) it satisfies these properties as,

1. Root node has atleast 2 children.
2. All nodes other than root node have atleast $\lceil m/2 \rceil$ children.
3. All failure nodes are at the same level.

(failure nodes ↓ (empty nodes given as □)).

Ex:-



B tree of order 3 ie, (m-1 no of children).

→ Operations of B-Trees:-

The major operations of B-tree are,

i. Creation
ii. Insertion
iii. Searching
iv. Deletion.

i. Creation:

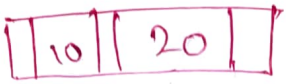
When the B-tree is empty. Get a node as root node and insert key into it.

ii) Insertion :-

For the above empty B-tree insert elements 10, 20, 30, 40, 50, 60, 70, 80, 90 with order as 3.

Insert 20 :-

To insert node in B-Tree order m can have (m-1) key values. So root node itself holds 20 with it as follows.

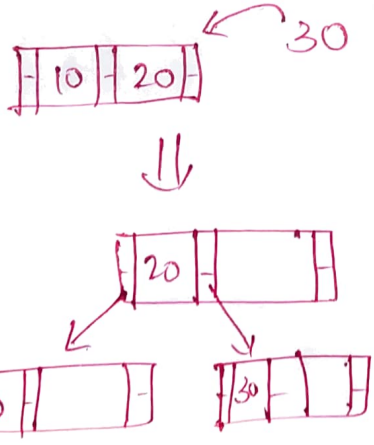


Insert 30 :-

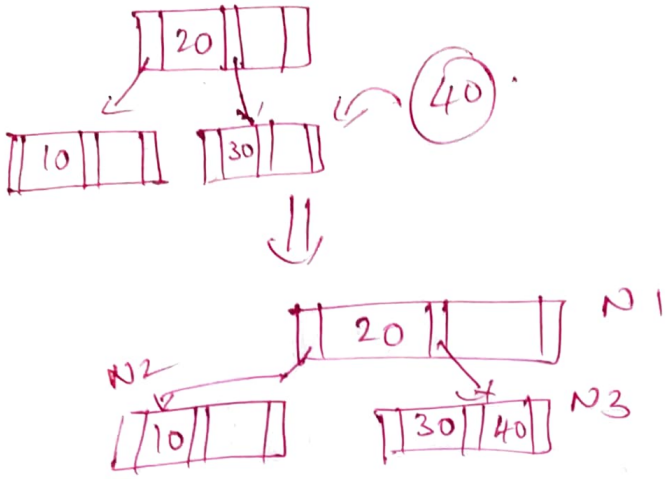
→ To insert 30, already B-tree is with max^m no. of key values (m-1) So follow these steps as,

- Insert value x, into list of values in node.
- Split list of values as P_1, P_2, P_3.
  - Where P_1 has first $\lceil m/2 \rceil - 1$ keys
  - P_3 has $\lceil m/2 \rceil + 1^{th}$ keys
  - P_2 has $\lceil m/2 \rceil^{th}$ value.
- With these steps $\lceil m/2 \rceil^{th}$ value is inserted into parent node of current node.

Insert 40 :-

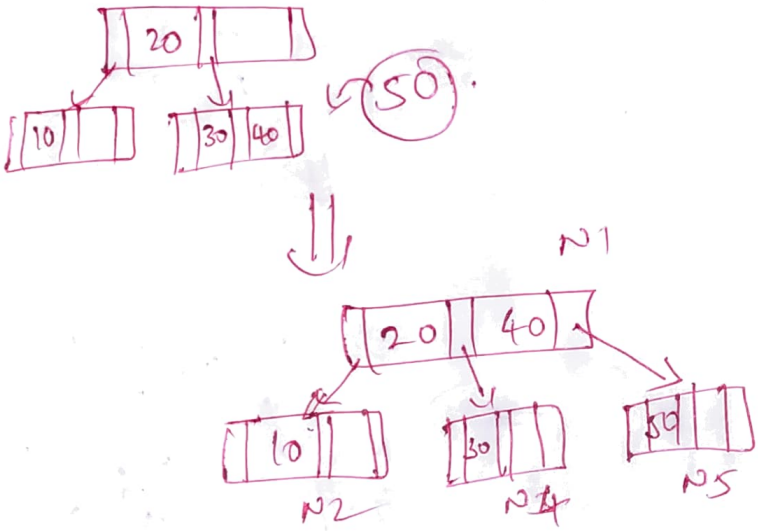


Insert 40:-

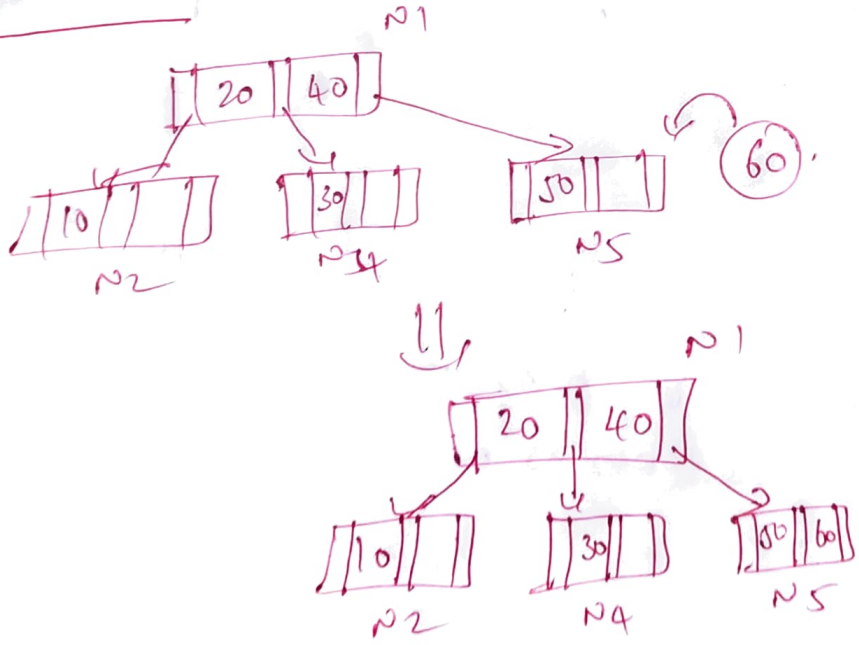


Insert 50:-

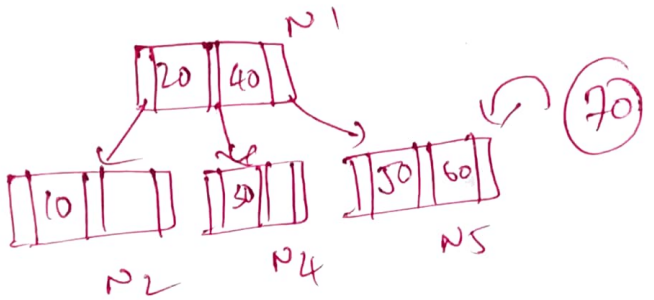
Here 30 should be splitted ∵ Node 3 is full.



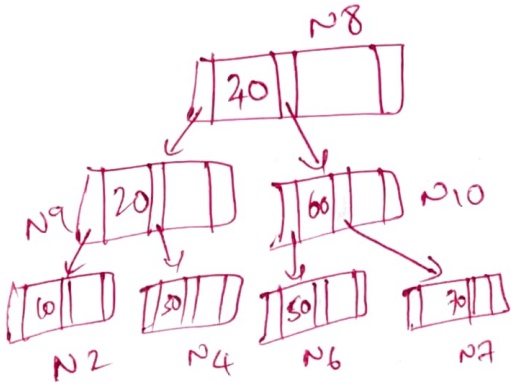
Insert 60:-



Insert 70 :-



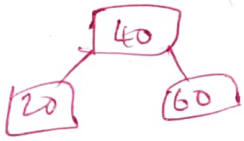
- Split N5 as it is full already. So split it as N6, N7.
- This again need to split N1 into N8 N9 N10 as we need to place 60.



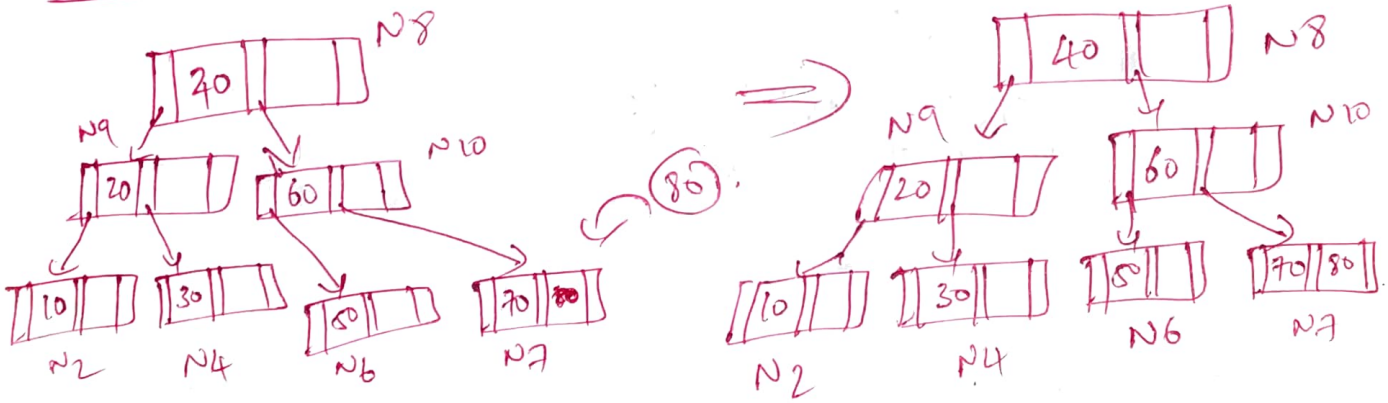
- Here To split 50 | 60 | 70
  60 is moved to root ie, Arg

  20 | 40 | 60

- So again this root need to be splitted So we make it as,

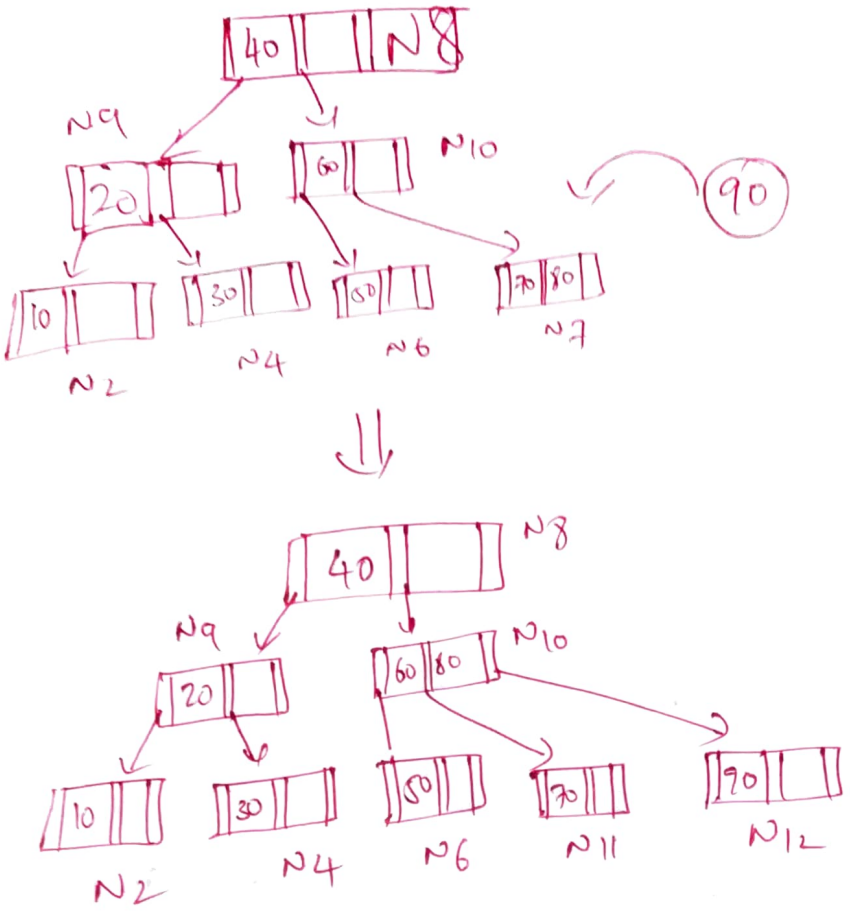


Insert 80 :-

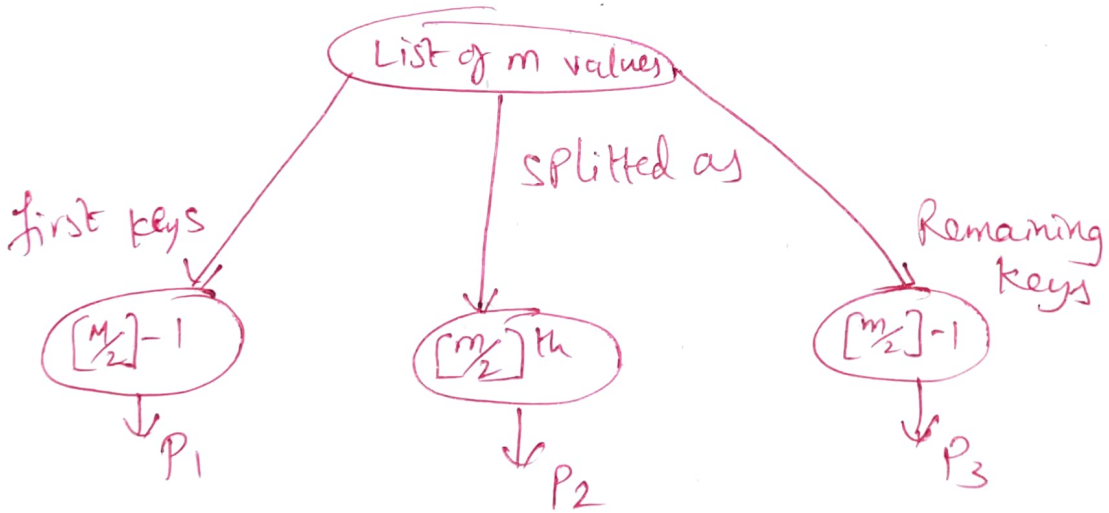


Insert 90 :-

- To insert 90 again 70, 80, 90 will be splitted as, by moving 80 to root,
- Again 70 & 90 are taken as N11 & N12 as follows,



Note :- The general format of B-tree is,



iii) Searching :-

- The searching in B-Tree is as same as BST's.
- The only difference is in BST, nodes are considered as of only one key value (2 children), whereas in B-Tree, order 'm' will have about (m-1) keys and m children.

- So while searching for an element, it should be checked by comparing with root node elements, resulting in less than, in between node value elements & higher than the root elements.

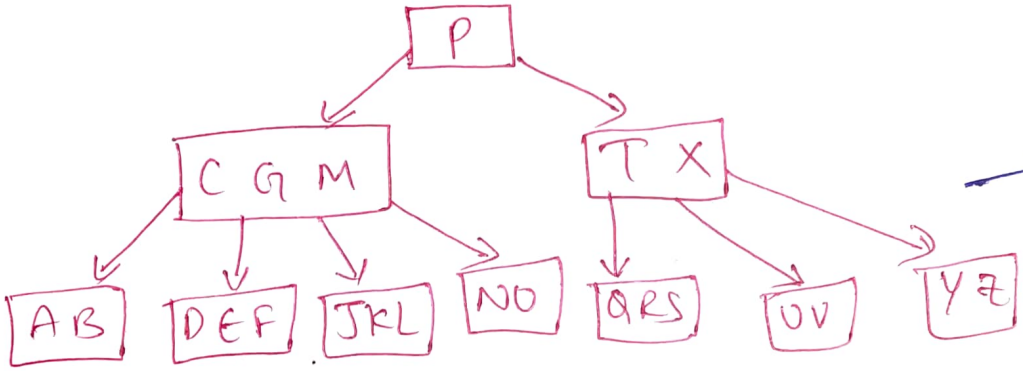
<Refer example>

iv) Deletion :-

- Let k be the key to be deleted, And x be the node having the key k. Then there arises different cases as,

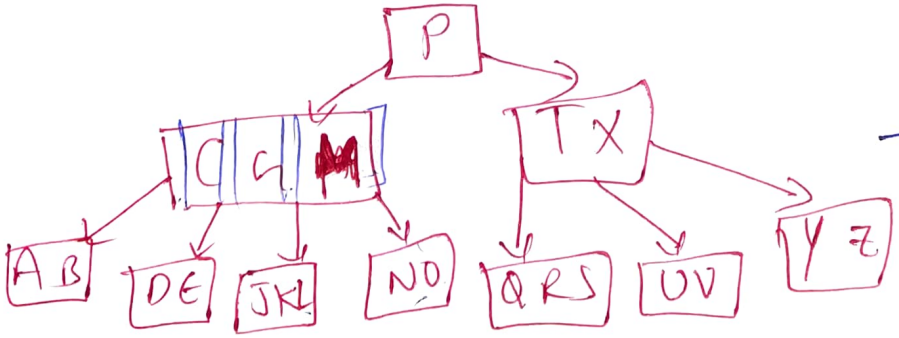
Case 1:

If key is already in leaf node, remove it directly. But, removing it should not result the leaf node with too few keys.



→ fig (1)

↓ Delete (F)

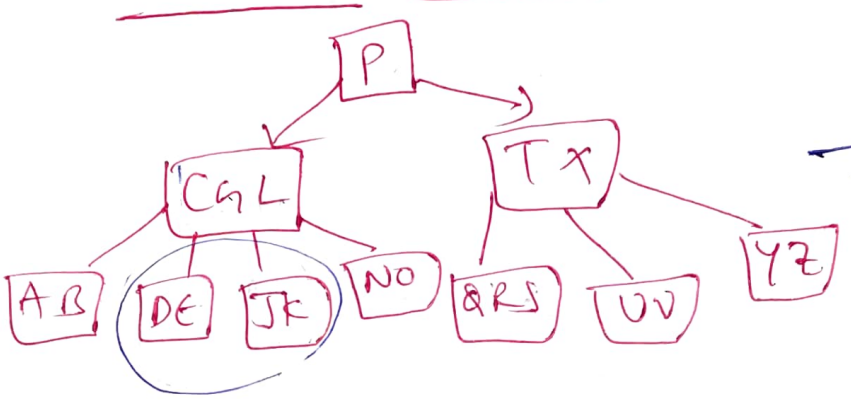


→ fig (2)

Case 2: If key k is in node x, x is an internal node then again 2 cases are raised as,

2a: If child y precedes/follows k in node x has atleast t no. of keys (more than min.) then replace with predecessor/successor key k' in its subtree of y. Then delete it & replace it.

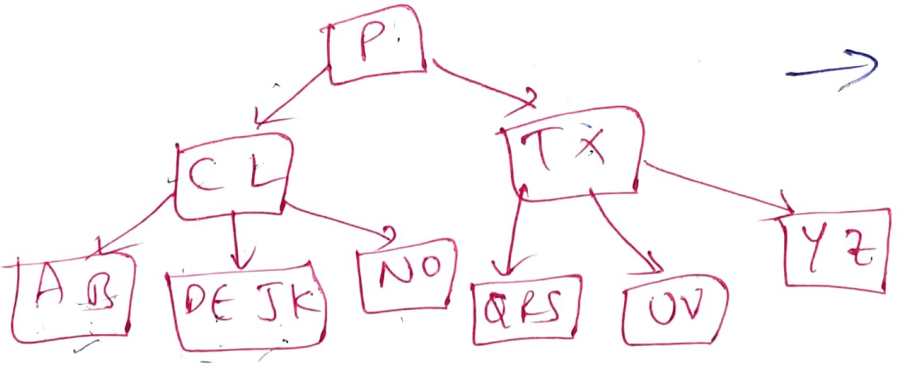
Now Delete (M) from fig (2) ie, tree (2).



→ fig (3)

2b: If both y, z has (t-1) min. keys, merge k & all of z to y so that both k and z are removed from x.

Now Delete (G) fm fig (3)



→ fig (4)

case 3: If k is not in internal node x, then find root of subtree with k.

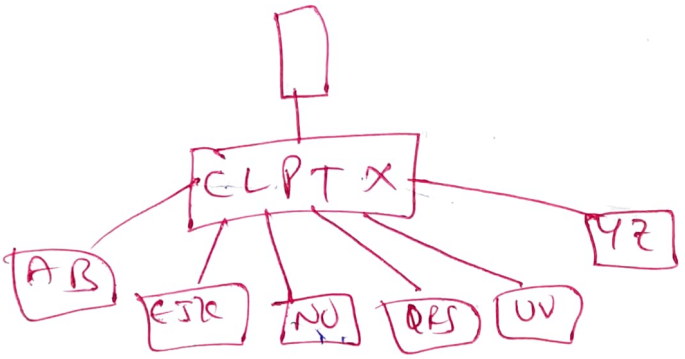
3a. If root has only t-1 keys but has sibling with t keys, then give root with an extra key by,

- moving key from x to root,
- moving key from root of L & R till x,
- moving child from sibling x.

Now Delete B from fig (4)

3b. If root and all its siblings has t-1 keys then merge root with sibling.

Now Delete D from fig-(4)



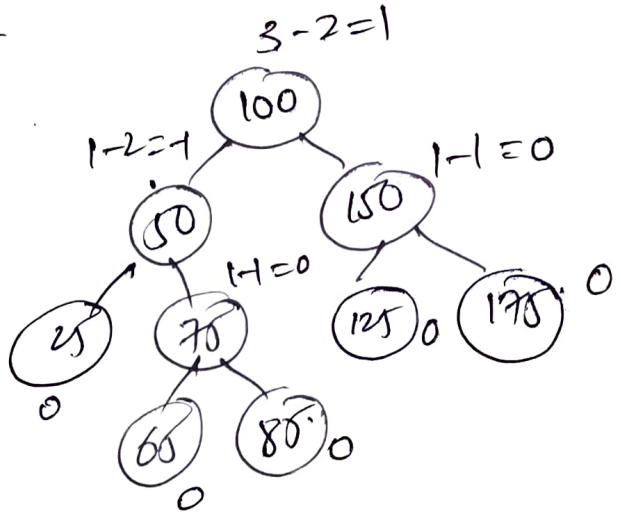
⇒ AVL Trees :-

- AVL Tree is invented By GM Adelson-Velsky and EM Landis in 1962.
- The AVL Tree is defined as height balanced binary search tree where each node is associated with a balance factor, given by the difference of Right sub tree from that of its left subtree.
- The Tree is said to be balanced if balance factor of each node lies b/w -1 to 1. Otherwise tree is unbalanced one and need to be balanced.
  - If both Left subtree & right subtree has equal height, then its balance factor is 0.
  - If left subtree is one level higher than right subtree, then its balance factor is 1.
  - If Right subtree is one level higher than left subtree, then its balance factor is -1.

$$\text{Balance factor(key)} = \text{height(left(k))} - \text{height(right(k))}$$

$$\text{ie, } |bf| = |h_L - h_R| \leq 1$$

Ex :-



⇒ Height balanced Tree / AVL Tree.

- Whenever a tree is said to be unbalanced tree (ie, a balance factor other than $-1, 0, 1$) then it is changed to a balanced tree by the AVL rotations.

INSERTION

- This unbalance tree is obtained when we insert some new elements into a tree. Then following steps are followed.
  1. Insert node into BST
  2. Compute balance factors
  3. Decide pivot node (A node which has its absolute value of balance factors from 1 to 2)
  4. Balance the unbalanced tree (By AVL rotations)

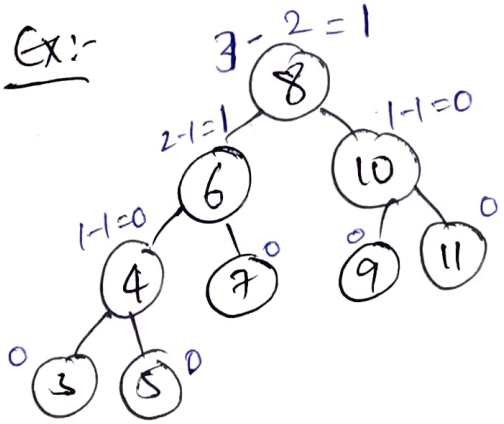
There are four cases of rotations possible in AVL inorder to balance a tree.

Case 1: When Insertion of Left subtree of left child of pivot node,

then,
- Right subtree (AR) of leftchild (A) of pivot node (P) becomes left subtree of P.
- P becomes the right child of A.
- Left subtree (AL) of A remains the same.

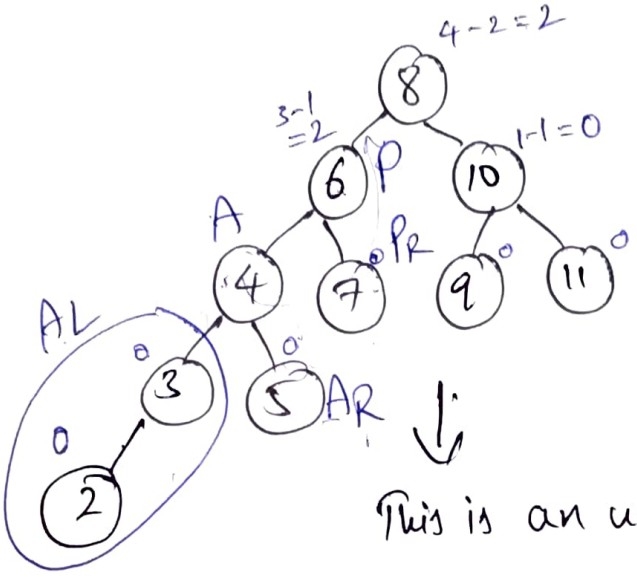
* (In simple LL Rotation (Left-to-Left rotation) occurs when new node is inserted to left subtree of left subtree of pivot node)

Ex:-



→ AVL | Height balanced tree as $|bf| \leq 1$

Insert a node 2 into above tree (just as BST) then we get, (Al-3)



This is an unbalanced tree ∵ |bf| > 1.

- So, here for node 8 & 6 |bf| = 2. so, to make such a balanced tree we follow rotations.

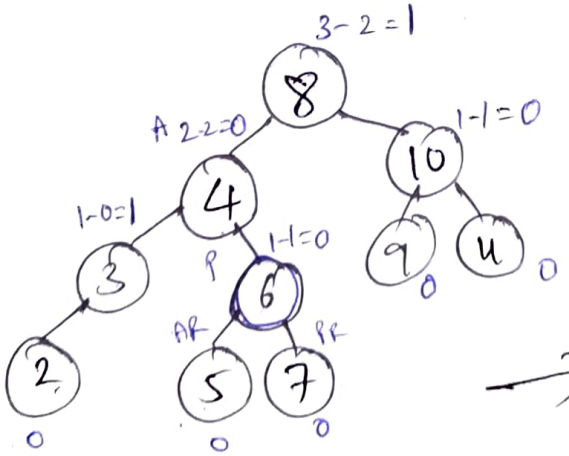
(Also we should follow LL rotation as new node is inserted towards left side root node-left node).

Let 6 be pivot and 4 be a node 'A'.

Then ~~BCC~~ to LL rotation,

- change P as right child of A.
- AR of A of P becomes left subtree of P.
- AL remains same.

Here,
6 = P
4 = A
5 = AR
3, 2 = AL
7 = PR

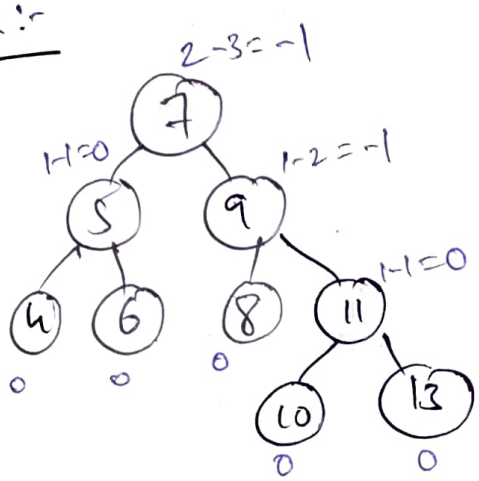


→ Balanced tree / AVL tree through LL rotation.

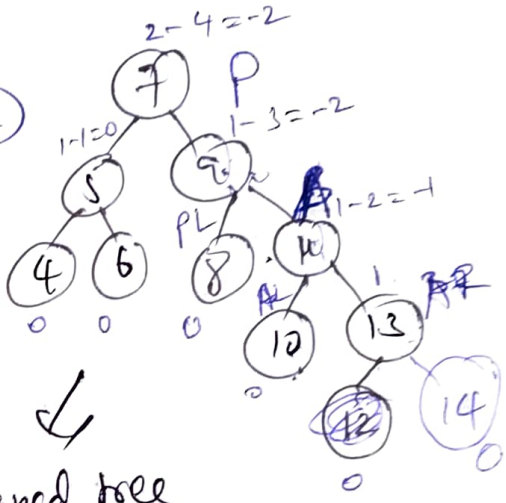
Case 2 :- If node is inserted into Right of Right subtree of node A then tree becomes unbalanced. In this case we use RR rotation.

- In this RR following steps are used as,
  - Left subtree (AL) of Rightchild (A) of First (P) becomes right subtree of P.
  - P becomes leftchild of A
  - Right subtree (AR) of A remains same.

Ex :-



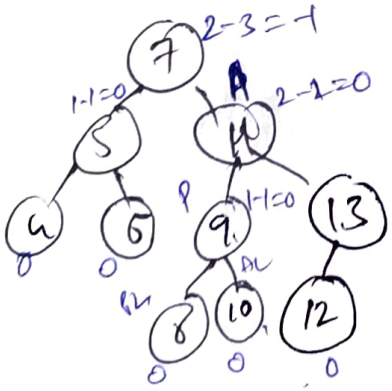
Insert (14)



(Unbalanced tree with |bf| as -2 for (7 & 9) nodes)

- Then Acc. to RR rotation we get,
  - Change B as root.
  - P as left of B, PL & BL being child nodes for P.
  - BR remains same.

P = 9
A = 11
PL = 8
AL = 10
13 } = AR
12



→ Balanced / AVL tree with RR rotation.

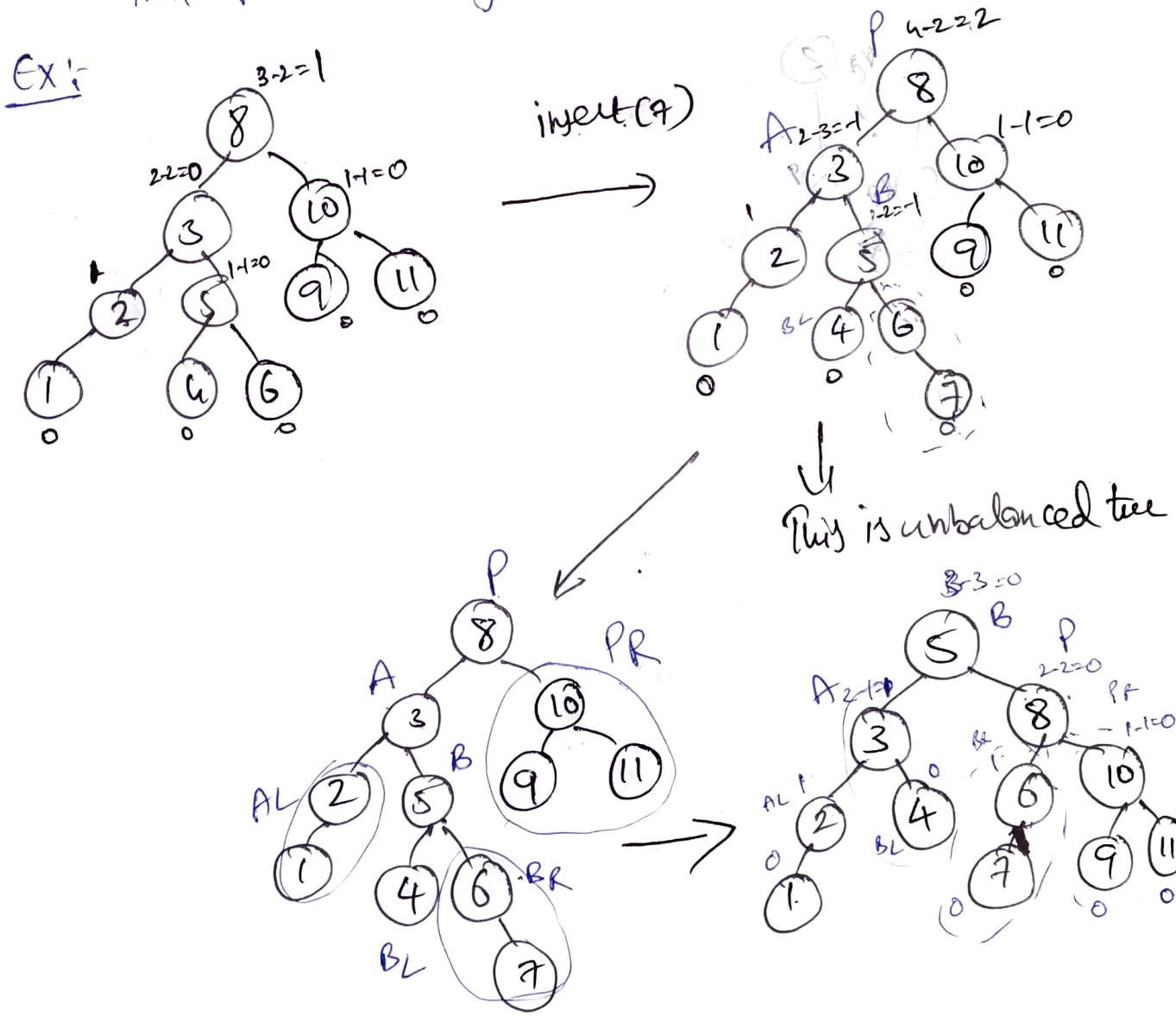
Case 3:- If node is inserted into right subtree of left child of pivot node then tree becomes unbalanced. In this case we use Left-to-Right rotation (ie, LR rotation).

This LR rotation can happen in two ways (or) in two steps as,

3a) (Left subtree of Right child) of left child of pivot (P) becomes right subtree of leftchild.
Then leftchild of pivot node becomes left child of B.

3)b) Right subtree of right child of left child of pivot (P) becomes left subtree of P.
Then P becomes right child of B.

Ex:-



Case 4 :- If node is inserted into left subtree of right child of pivot node then tree becomes unbalanced. In this case we use Right-to-left rotation.

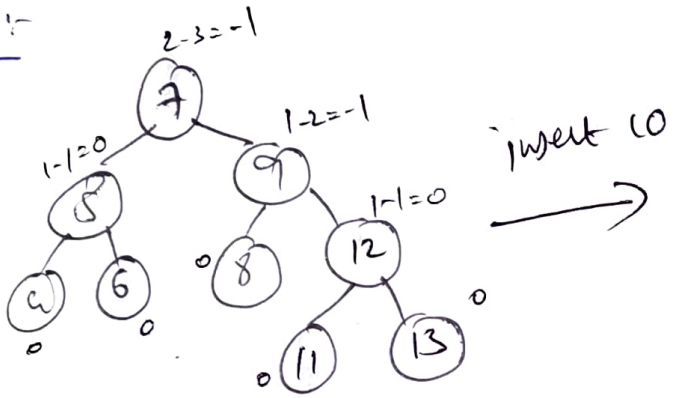
This RL rotation can happen in two steps as,

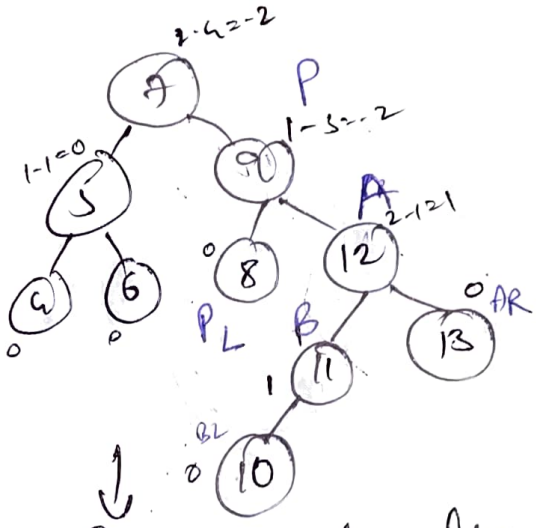
4a) Right subtree (BL) of left child (B) of right child becomes left subtree of right child (A). Then right child of pivot (P) becomes right child of B.

4b) Left subtree (BL) of right child (B) of right child (A) of pivot node (P). becomes right subtree of P.

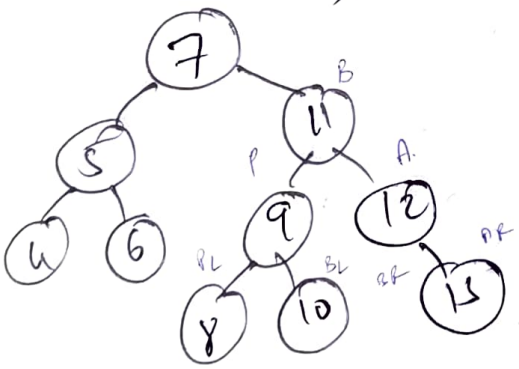
Then P becomes left child of B.

Ex :-





This is unbalanced tree



⇒ DELETION ⇐

→ While deleting a node from the AVL tree the process remains as same to that of deletion of node in BST.

→ But soon after deleting every single node from AVL balance factor need to be checked.

→ If after deleting a node tree becomes unbalanced then to make the tree as a balanced one by performing two rotations as,

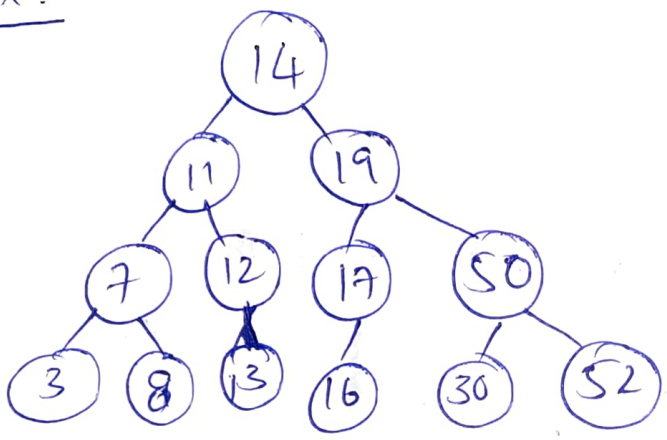
1. Left rotation
2. Right rotation.

} This includes all the other rotations as discussed such as (LR, RL)*

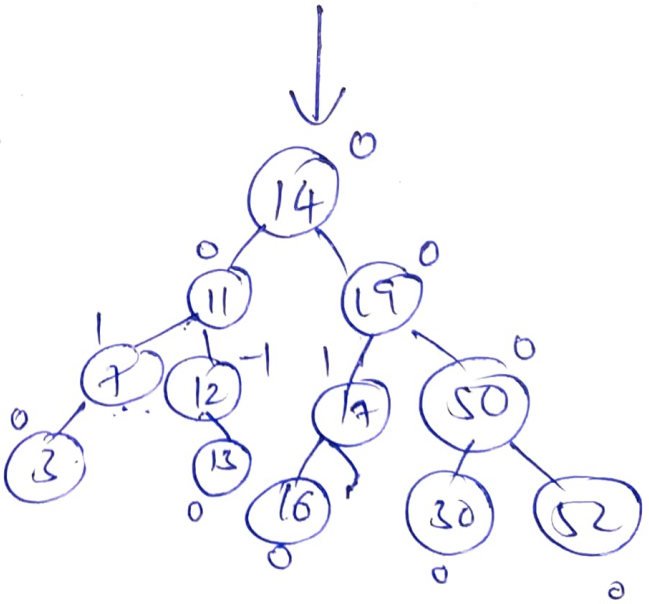
↓

(These both rotations are as same as rotations in insertion process)

Ex :-



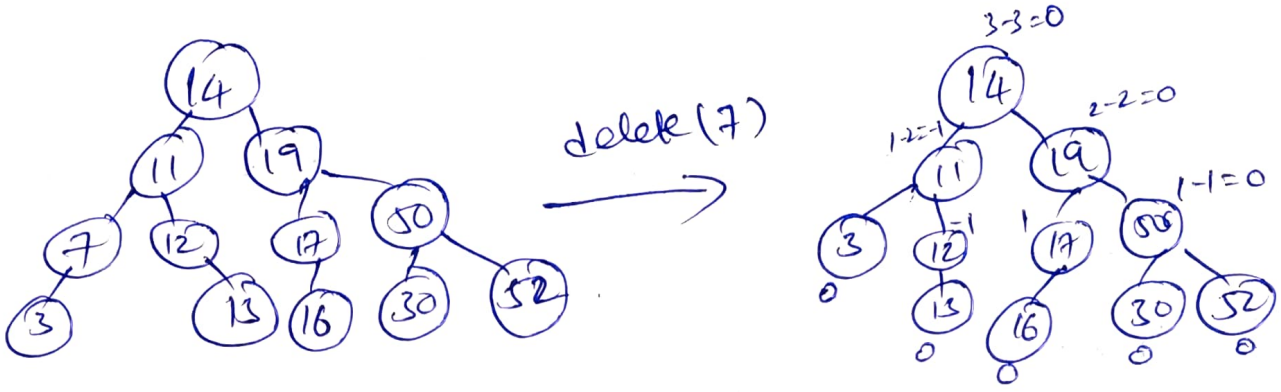
delete(8)



Even after deleting node 8 the tree is balanced. So no other rotations have to be done.

ii) Delete 7 from resultant tree :-

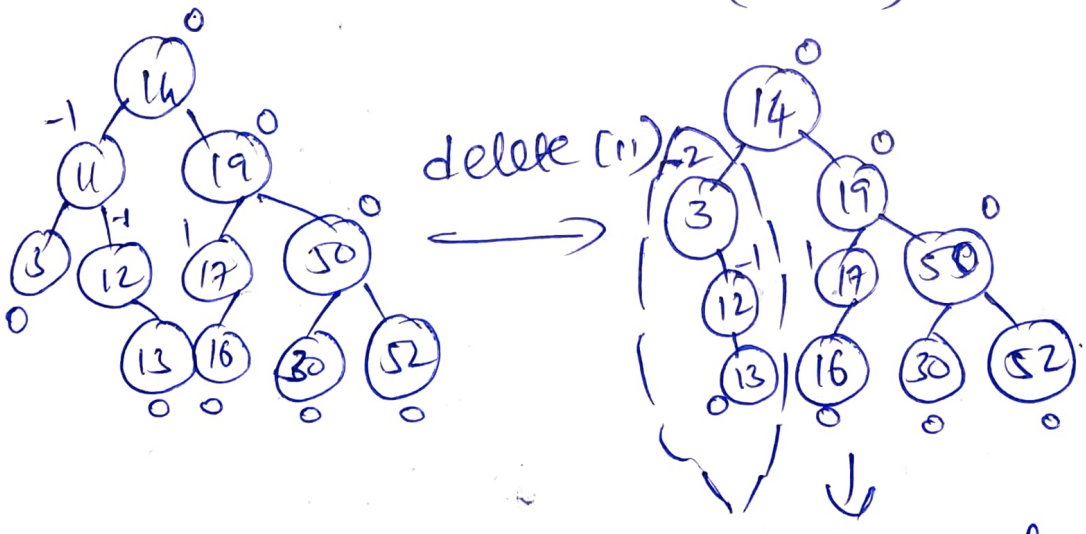
- Search for element 7 using BST rules.
- If any node with a single child is to be deleted then make its child to be as a root and delete the node.



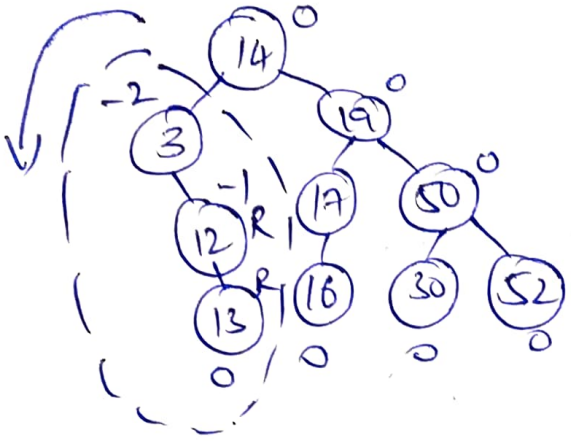
- So, here delete 7 and its only child 3 to '14'.
- Then check for balance factor for every node. If it is other than -1, 0, 1 then go for rotations, else it remains same.

iii) Delete 11 :-

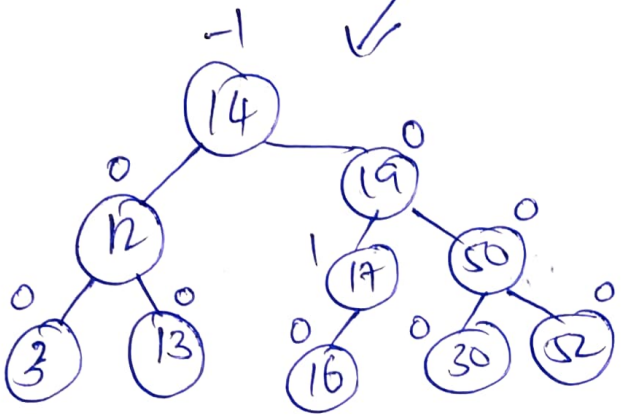
- To delete '11' from tree, it has both right and left children. So, replace the deleted position with its predecessor node (Inorder)



unbalanced tree so balance it by rotation.

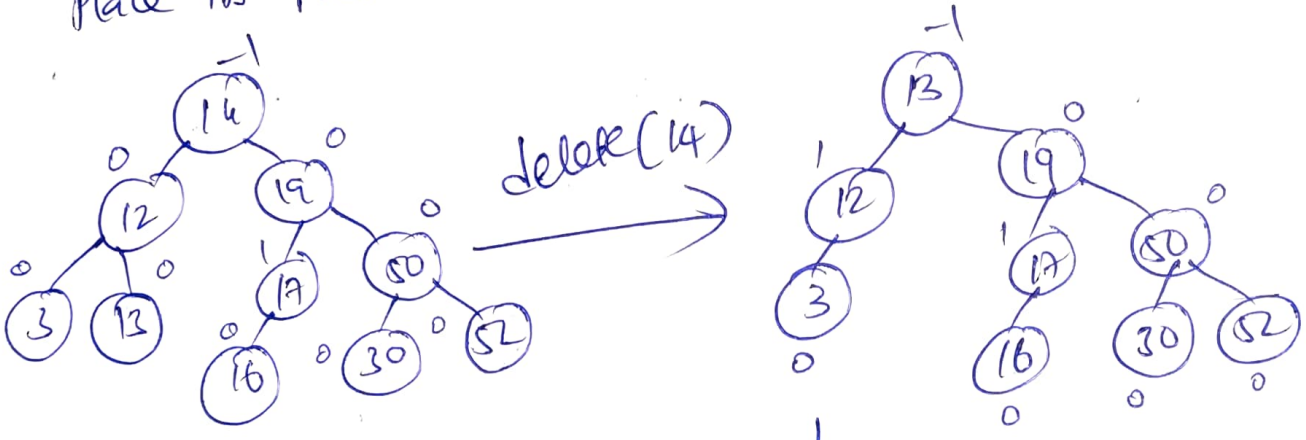


→ To make this balanced, as balance factor is -2 & its connected nodes are towards R-R, this faces RR rotation.



iv) Delete 14 :-

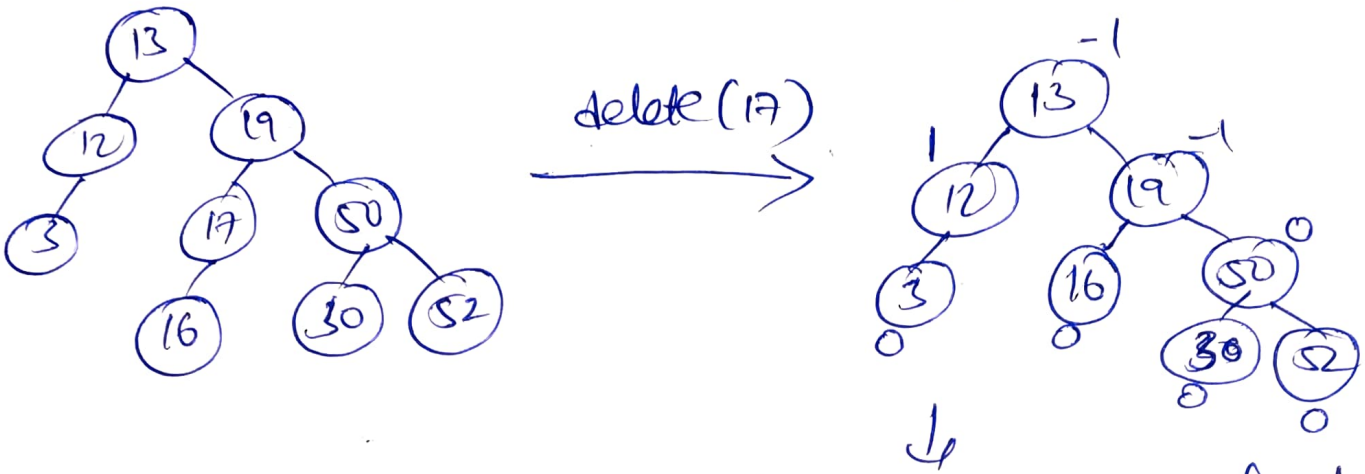
- To delete 14, go with BST deletion rules and place its Predecessor in its place.



here tree is a balanced tree. So, no need to use concept of rotations.

v) Delete 17 :-

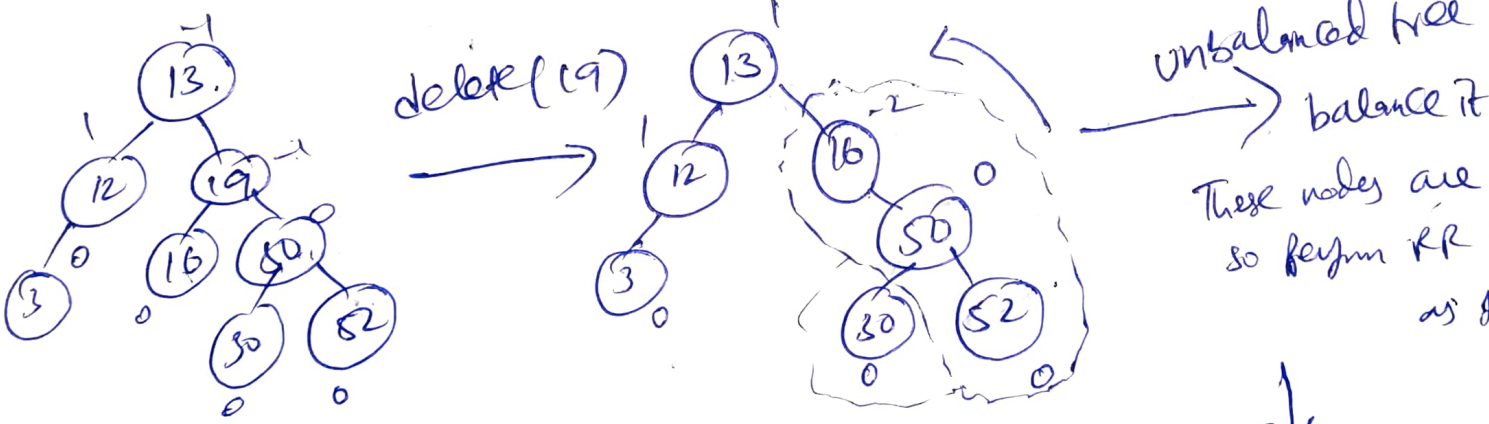
- To delete 17, from tree, as it has a single child replace its place with its only child.

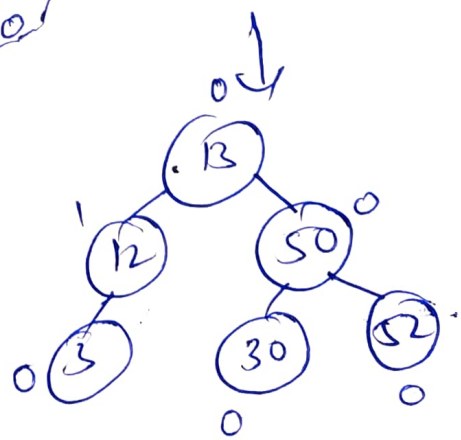


Then it is a balanced tree.

vi) Delete 19 :-

To delete '19' from tree, it has both right and left children. so, replace the deleted position with its predecessor node (inorder)





Thus, it is a balanced tree

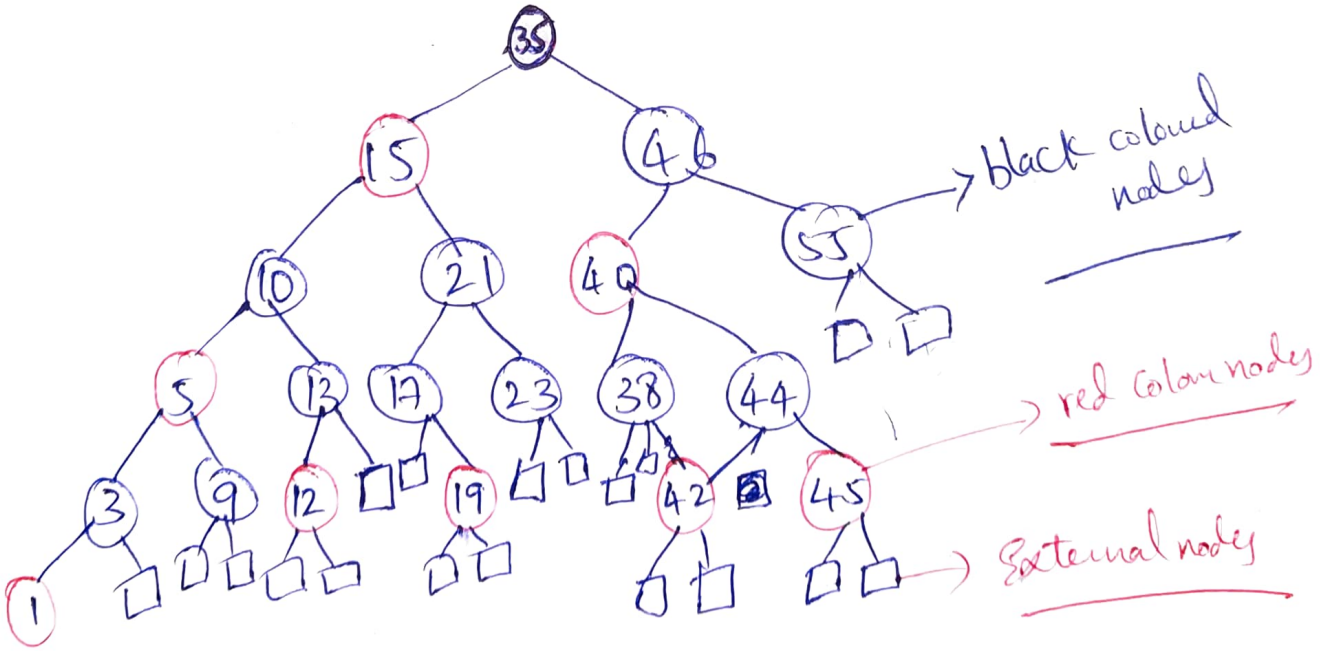
⇒ Red Black tree :-

→ A red-black tree is a kind of height balanced BST,

→ This was given by Rudolf Bayer and Ed McCreight. in the year 1972.

→ A Redblack tree is a BST that satisfies following properties.,

i) Colour property – Every node is either red | black.
ii) Root property – The root is always coloured with black.
iii) External property – Every External node is black.
iv) Internal property – If a node is red then both its children are black.
v) Depth property – for every node, all paths from that node to descendant External nodes contain same number of black nodes.



→ From the above tree we call it as Red-Black tree as it follows these properties as,

- It is a BST (Basic property)
- Every node is either red / black (Colour property)
- The root is coloured black (root property)
- All external nodes are coloured black (External property)
- Children of every red-coloured node are black (Internal property)
- For every node in the tree, & all paths from node to External nodes, counts of black coloured nodes are same (depth property).

Note:-

- Internal property holds for red nodes only, ie, both child nodes of red are black, but if node is black node then its both child nodes are of either black or red or both.
- Depth property is called as black depth. ie, no. of nodes in black from root to external node should be of same depth.
- External nodes are used to terminate the search process.

→ Operations of Red Black Tree:-

- The basic operations of RB Tree are as follows,
  - i) creation.
  - ii) Insertion
  - iii) Deletion
  - iv) Searching.

i) Creation:-

Whenever a new RB tree is to be created then the initial node inserted will be the root node. And after inserting node change its colour to black (as root should be of black).

⇒ i) Insertion in Red Black tree :-

- Whenever a node need to be inserted there are basic steps to be followed as,
  - Allocate node for RB Tree
  - Assign the value x into DATA.
  - Initially color of node inserted will be of RED.
  - The node has two black children as External node.
- The basic properties are,

i) Whenever a new node is attached to a RB Tree, assign the node a color.

- If tree was Empty before insertion, then new node is the root and is colored black.
- If tree was not empty & if newnode is given a black color then we may have Extra black node on paths from root to external nodes.
- If tree was not empty & if newnode is given a red color then we may have two Consecutive red nodes, obtaining violation of RB Tree property (Red-Red / double Red Problem). So, in this case we call tree as an imbalanced one.

ii) we may have different kinds of imbalances as when 'u' is the new node inserted as follows,

- When Pu is left child of gu, u is left child of Pu and other child of gu is black, resulting **LLb.**
- When Pu is left child of gu, u is left child of Pu, and other child of gu is red, resulting **LLr.**

- when P_u is left child of g_u, u is right child of P_u, and other child of g_u is black it results in LR_b.
- Similarly the other possibilities we obtain are,

RR_b, RR_r, LR_r, RL_b, RL_r → (P_u is right child of g_u, v is left child of P_u, other child of g_u is red)

(P_u is right child of g_u, v is right child of P_u, other child of g_u is black)

(P_u is right child of g_u, v is right child of P_u other child of g_u is red)

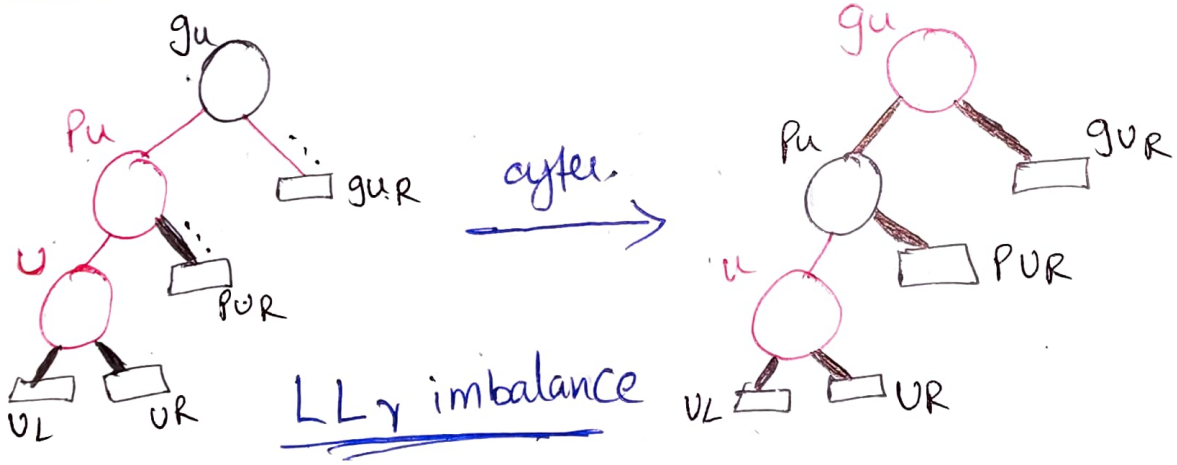
(P_u is left child of g_u v is right child of P_u other child of g_u is red)

(P_u is right child of g_u, v is left child of P_u other child of g_u is black)

Note:-

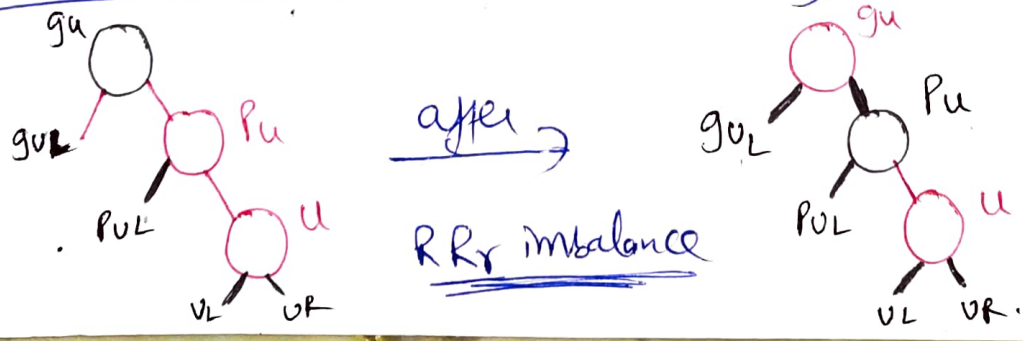
- Whenever LL_r, RR_r, LR_r, RL_r imbalance occurs then only change in color takes place.
- whenever LL_b, RR_b, LR_b, RL_b imbalance occurs then bothe colour change and rotations takes place.

a) LL_r Colour change ⇒ as it is LL_r only colour change happens.



LL_r imbalance

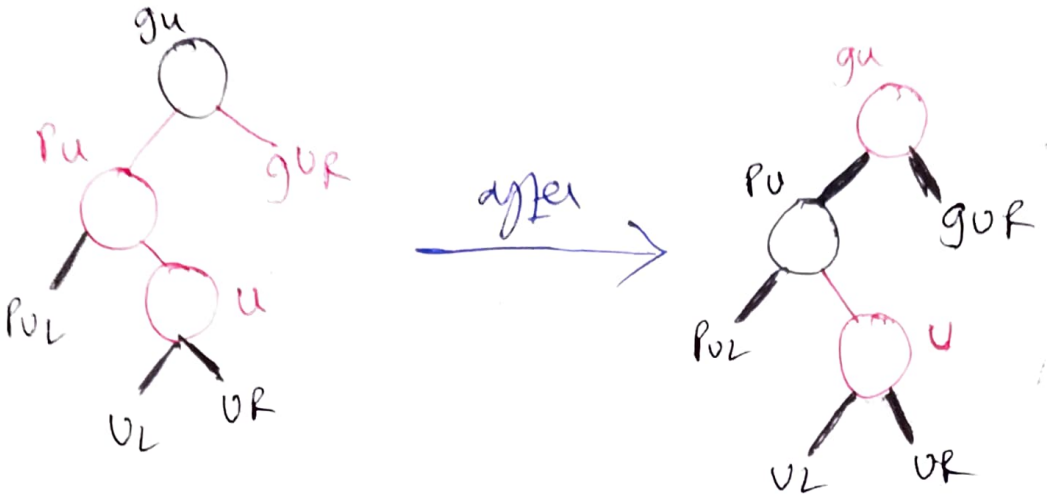
b) RR_r colour change ⇒ as it is RR_r only colour change happens.



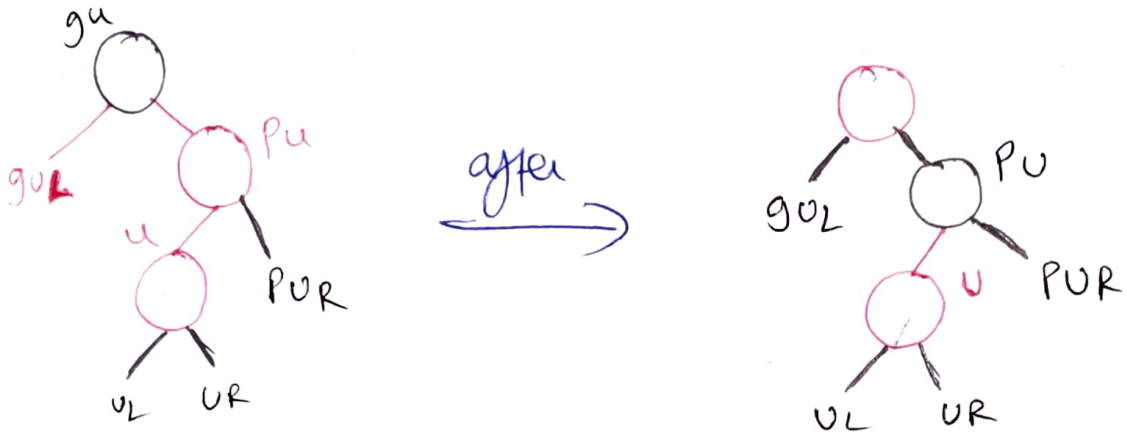
RR_r imbalance

RB-3

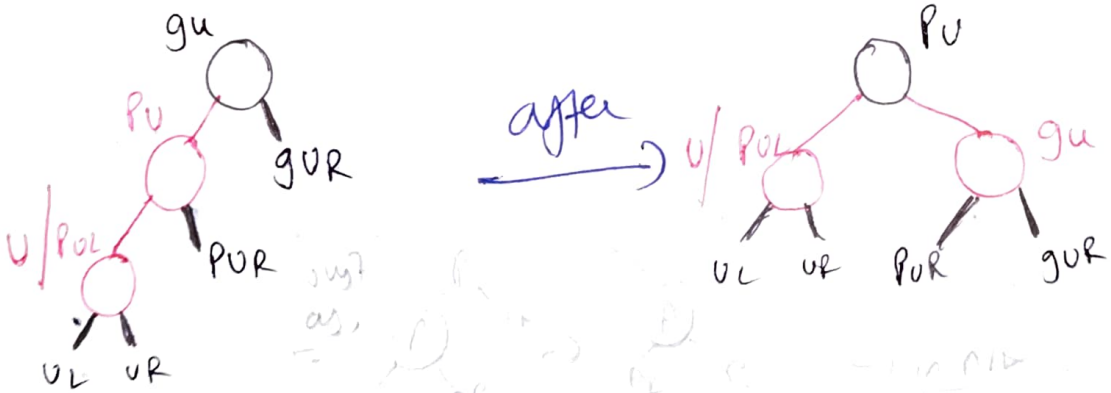
c) LR$_r$ Colour change => as it is LR$_r$ only colour changes



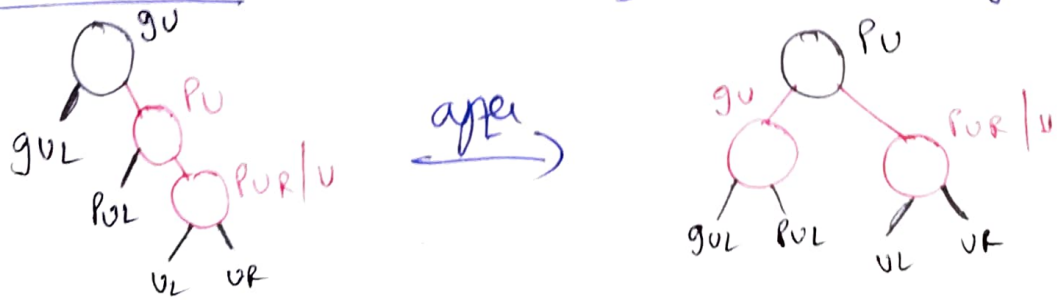
d) RL$_r$ Colour change => as it is RL$_r$ only color changes.



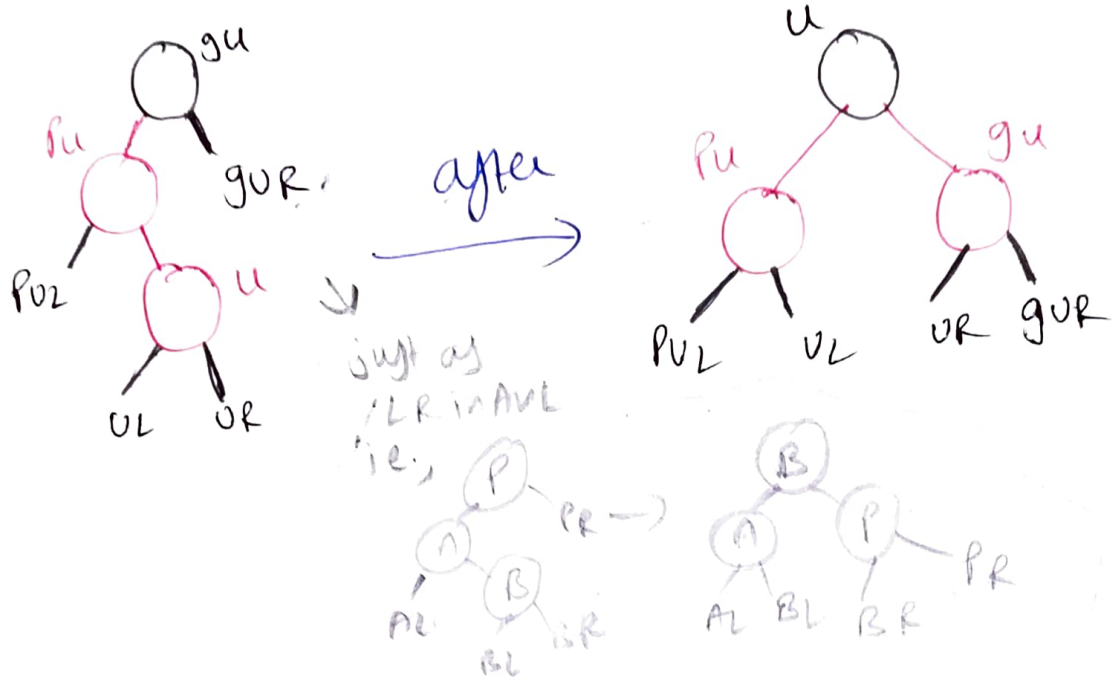
e) LL$_b$ imbalance => as it is LL$_b$ we do color change + rotation (LL)



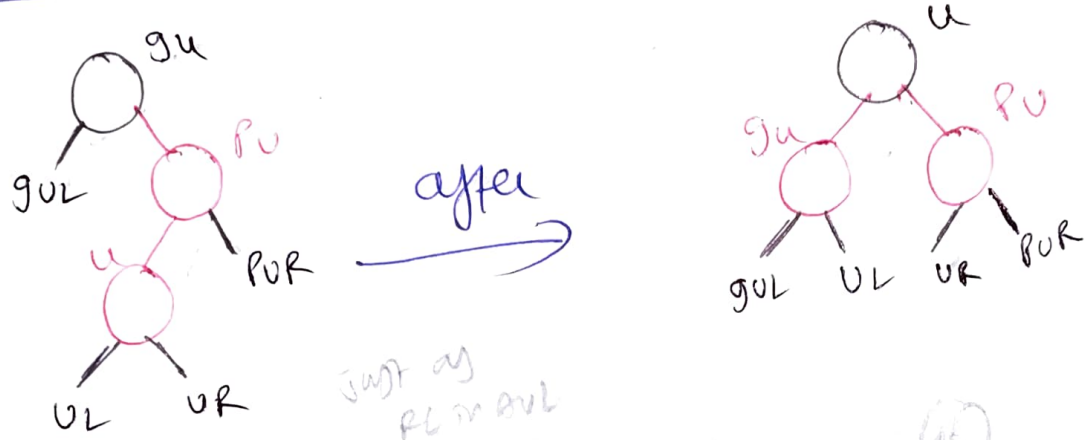
f) RR$_b$ imbalance => as it is RR$_b$ we do Colour change + rotation. (RR)



g) LRb imbalance => as it is LRb we do colourchange + Rotation (LR)



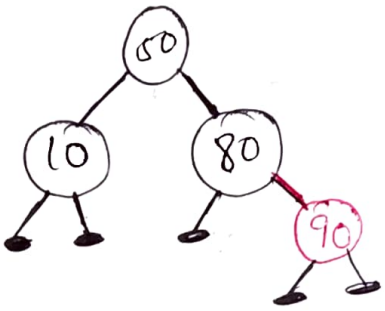
h) RLb imbalance => as it is RLb we do colour change + Rotation (RL)



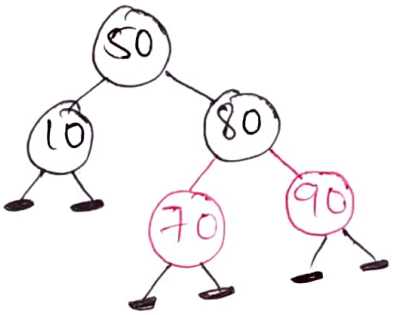
Note:-

* LLr & RRr are mirror image imbalances. Ily LLb & RRb
* LRr & RLr are mirror image imbalances. Ily LRb & RLb.

Example :-

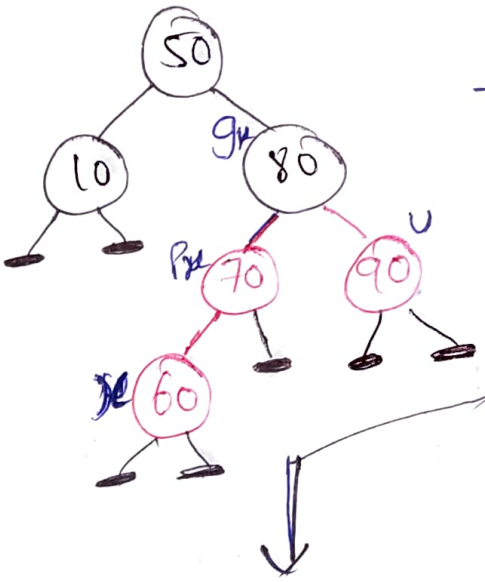


↓ Insert 70

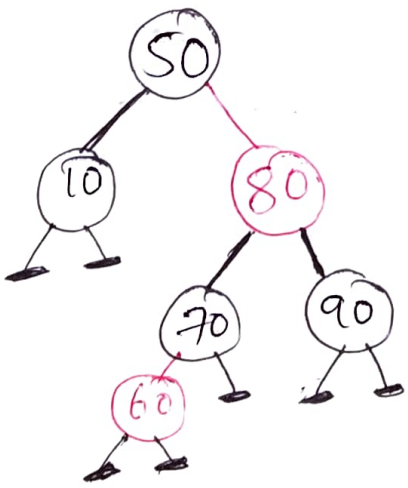


- This has root as black
- Also satisfies black-depth property
- And no Red-Red problem.

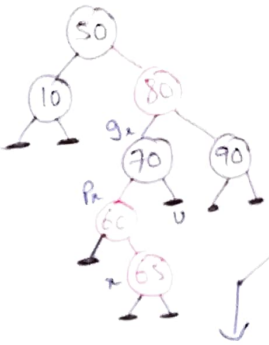
↓ Insert 60



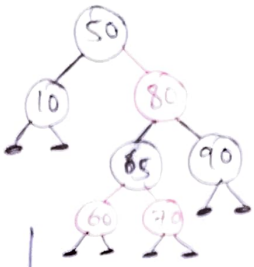
- This has Red-red problem towards LL so we have LLr (∵ U's uncle color is red),
- So at LLr imbalance we just change colour.
(ie, make g as red, P & U as black & x as Red)



↓ Insert 65



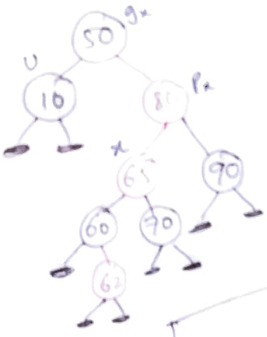
- Here we encountered Red-Red problem at LR. So we have to perform LR b imbalance (∵ uncle of x is black)
- As it is LR b we perform Color change & rotation too (as that of AVL LR rotation)



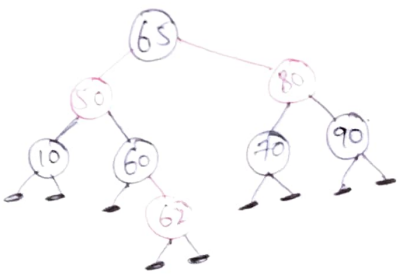
Insert (62)

- Here we encountered Red-Red problem at LR_y. So we perform LR_y imbalance (∵ uncle of x is red)
- As it is LR_y we do only Color change.
(change g to red, P & U to black & x as red)

P.G-9



- But here after LR_y we do have Red-Red problem at RL.
- So we do RL b imbalance (∵ uncle of x is black) we perform colour change & rotation.
↓
(as of AVL-RL rotation)



⇒ Deletion:

- Deletion of node from RB Tree is as same/similar to deletion of node from BST.
- But color changes & rotations may be required to rebalance the tree after deletion occurs depending upon original structure of RB Tree.
- This deletion involves in different cases,
- Let P_y be the parent of y and v is y's sibling then,

① X_b imbalance :-

- If v is black then imbalance is at L_b (or) R_b then it is called X_b imbalance.
- Here X is either 'L' (or) 'R'.

② X_r imbalance :-

- If v is red then imbalance is either L_r (or) R_r, then it is called X_r imbalance.
- Here X is either L (or) R

Removal of imbalances :-

(i) Removal of X_b imbalance :-

- The X_b imbalance can happen in 3 ways based on number of children of v node.
- Here v is in Black colour

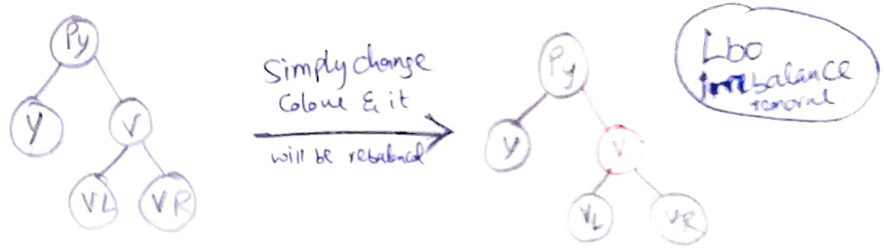
Case i) :- (with no red children)

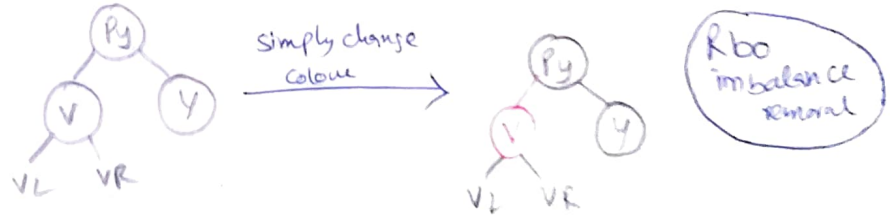
- Rb_0 - y is right child of P_y, v is black with no red child
- Lb_0 - y is left child of P_y, v is black with no red child.

↓

So under these both cases only the colour of node changes.

- when y is in black and need to be deleted.
- Check y's sibling v's colour. Even if v is black with having no children in Red, then simply we change colour of v to red as follows,





Case ii) :- (with one red child)

- Rb_1 - y is right child of P_y, v is black node with one red child
- Lb_1 - y is left child of P_y, v is black node with one red child.

↓

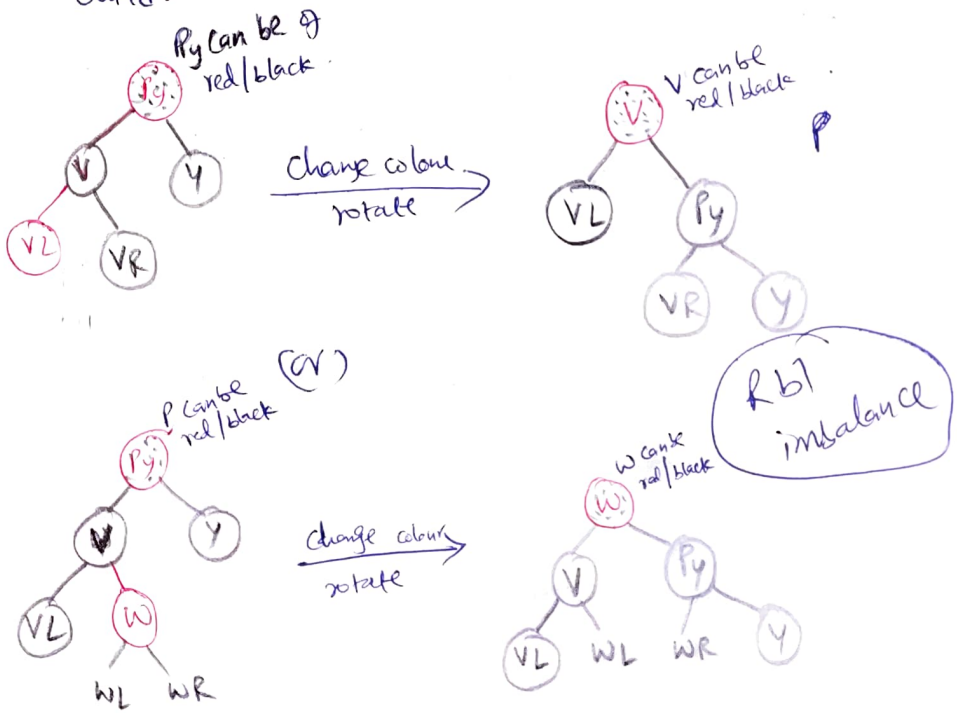
So under these both Rb_1 & Lb_1 cases both colour change and rotations takes place.

- when y need to be deleted check its sibling v if v has only one red child then go for colourchange & rotation.

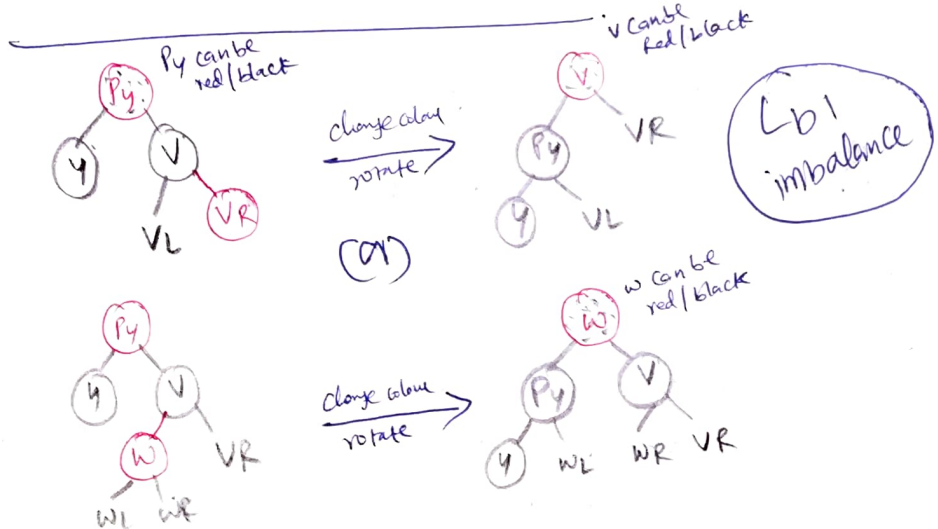
- Then make to rotate 'v' (if it has only one red child) ie, (towards right if it is black colour)

↓

(ie, if y is towards right in black colour and its sibling v is ~~black~~ with black colour with one red child.



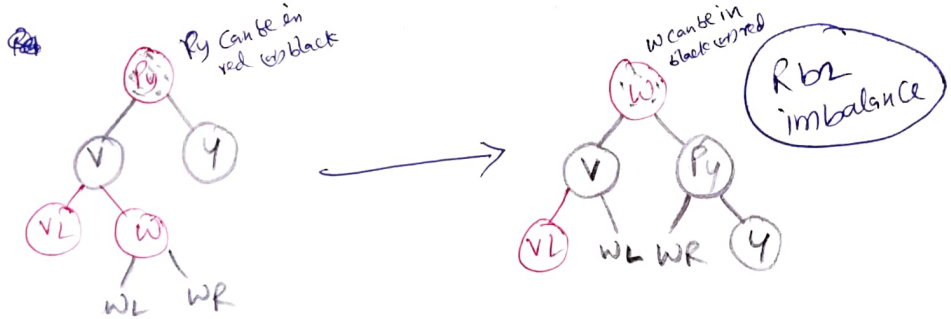
(or)

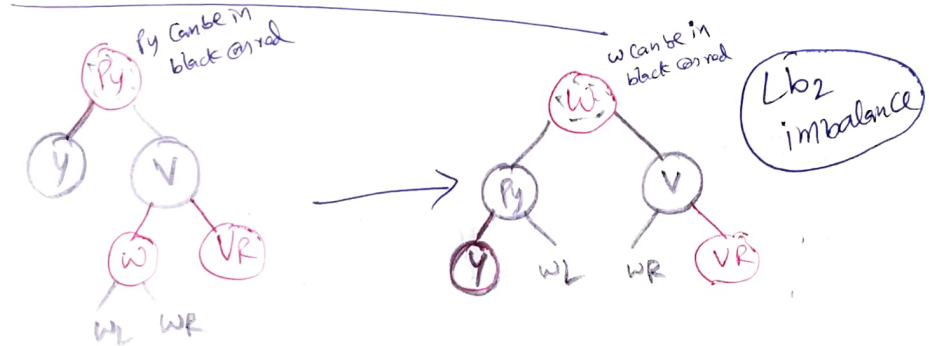


Case 3:- (with two red children)

- Rb_2 - y is right child of Py, v is black node with 2 red children
- Lb_2 - y is left child of Py, v is black node with 2 red children.

↳ Under these cases Rb_2, Lb_2 change the colour & rotate.





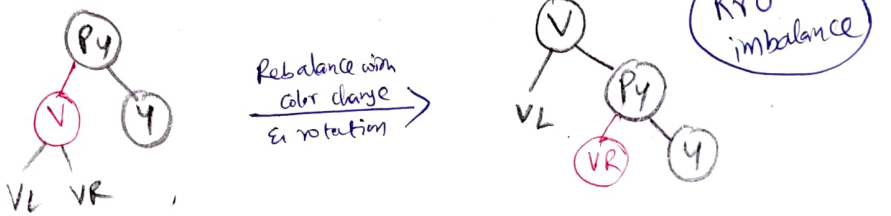
(2) Removal of X_r imbalance :-

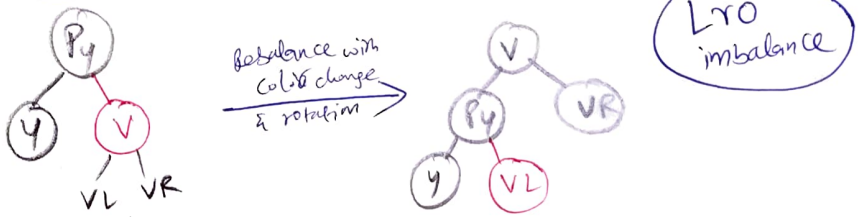
- The X_r imbalance again constitutes 3 cases, based on number of children of node V.
- Here V is in red colour.

Case 1 :- (with no red children)

R_{r0} – y is right child of P_y, v is red node with no red child

L_{r0} – y is left child of P_y, v is red node with no red child.

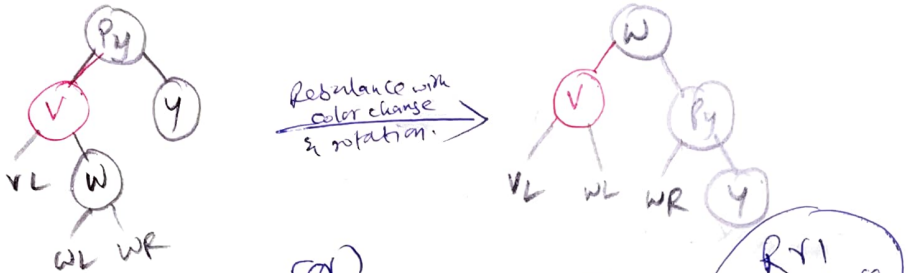




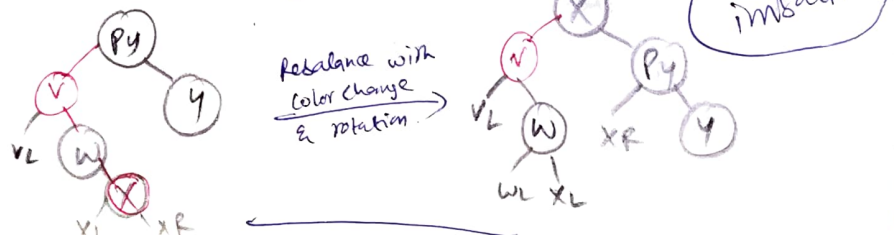
Case 2 :- (with one red child)

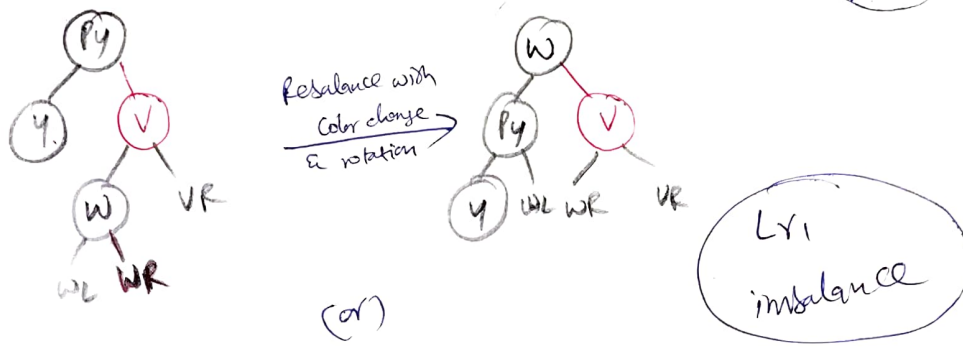
R_{r1} – y is right child of P_y, v is red node with one red child

L_{r1} – y is left child of P_y, v is red node with one red child

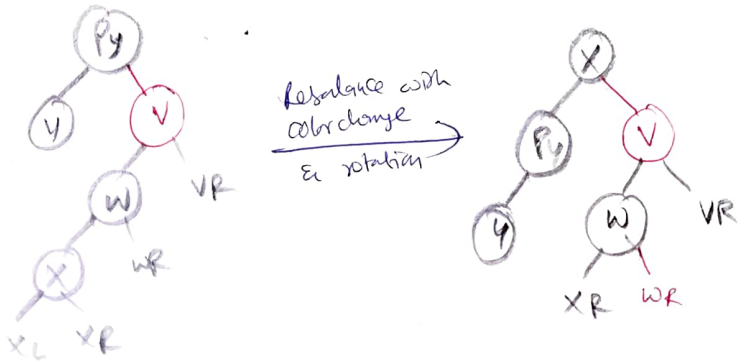


(or)





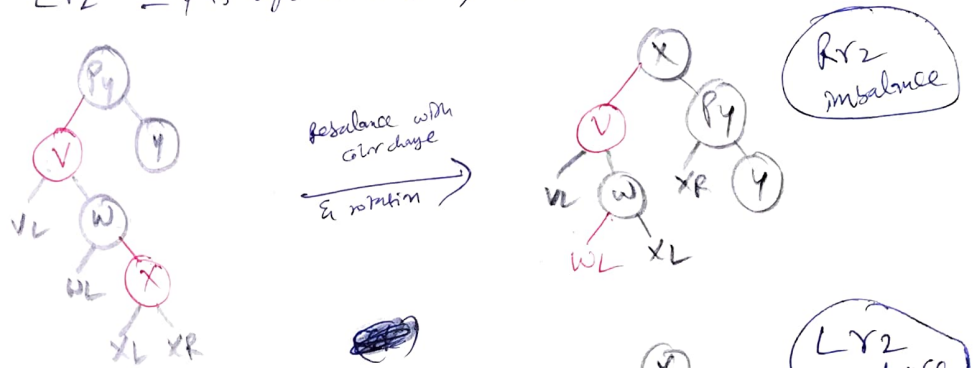
(or)

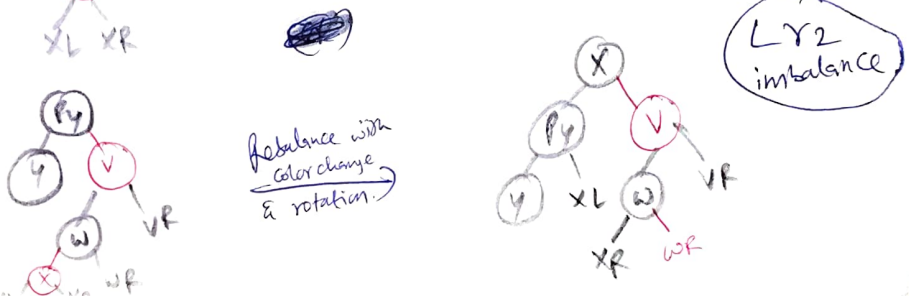


Case 3 :- (with two red children)

R_{r2} – y is right child of P_y, v is red node with 2 red children

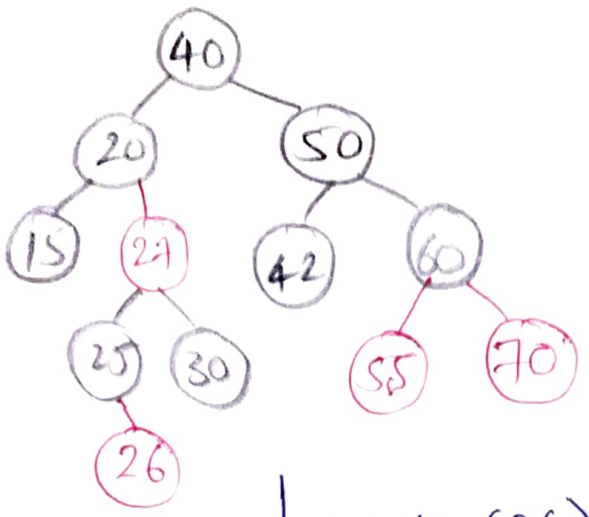
L_{r2} – y is left child of P_y, v is red node with 2 red children.



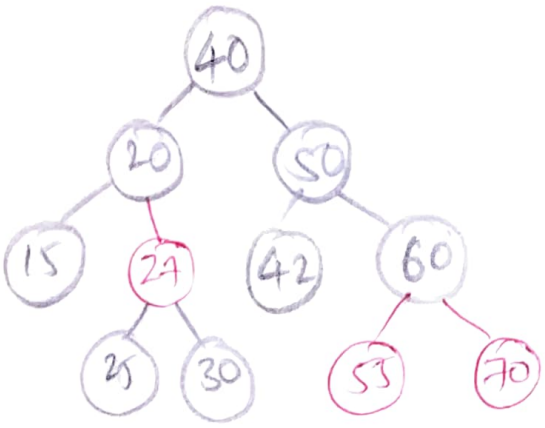


Ex:-

RB-16

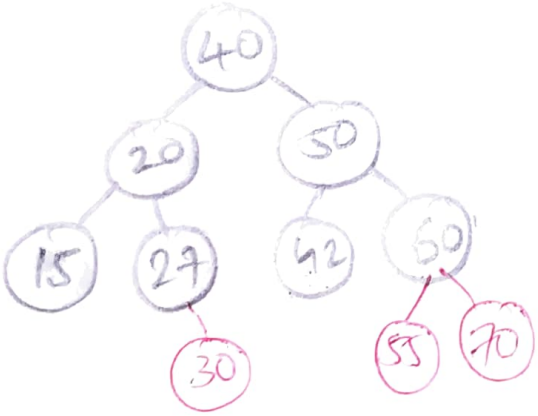


↓ delete (26) → If we need to delete 26 node it can be removed simply as it is a red node & last node no changes happens



It is LBo bcoz '25' is given towards left '2' & sibling is ie, 30 is in black '0' & so has no children '0'

↓ delete (25) → If 25 node need to be deleted then replacing of LBo type. Hence change color of V node to red & remove 25



↓ delete(42) → If 42 is deleted then it becomes LR2 imbalance