2021

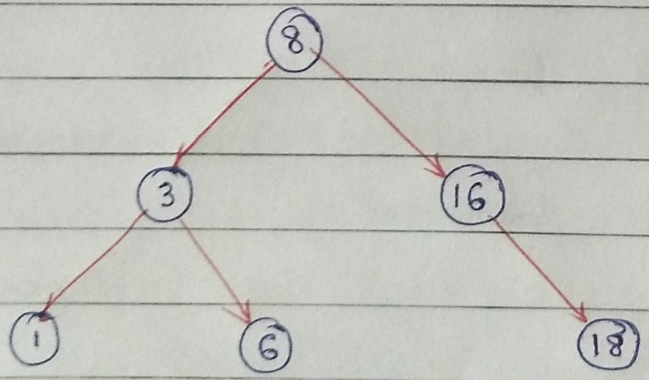
Module 2

Binary Search Tree

• BST is a node-based binary tree data structure has the following properties.

- The left subtree of a node contains only ~~one~~ nodes with keys lesser than the node's key.
- The right ~~subtree~~ subtree of a node contains only nodes with keys ~~lesser~~ greater than the node's key.
- The left & right subtree each must also be a BST.

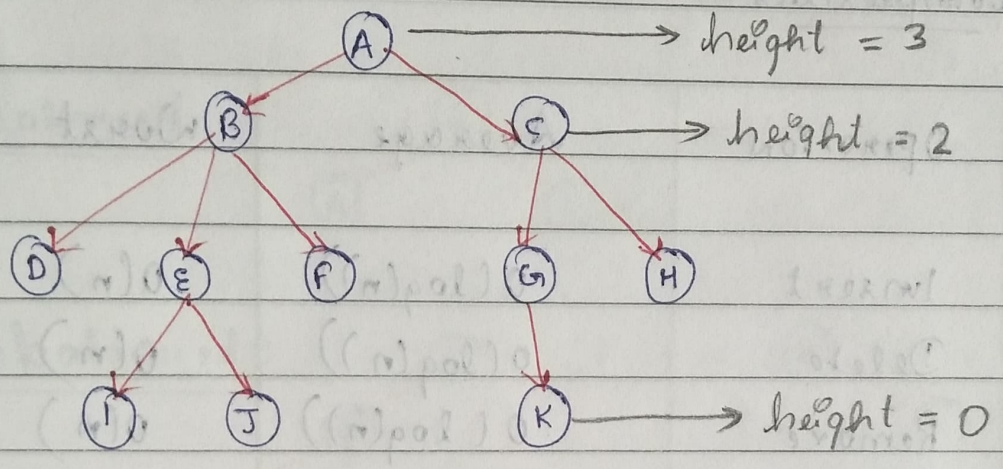
Eg:



Height of a Node & Tree

In a tree data structure, the total number of edges from leaf node to a particular node

in the longest path is called a height of that node. In a tree, height of the root node is said to be height of the tree. In a tree, height of a leaf node is '0'



Here height of tree is 3

- In any tree, height of node is total no. of edges from leaf to that node in longest path.
- In any tree, height of tree is the height of the root Node.

BALANCED BINARY SEARCH TREE

* A balanced Binary Search tree is a self balanced BST.
* This type of tree will adjust itself in

order to maintain a low ~~weight~~ height allowing for faster operation such as insertion & deletion.

Complexities

Operation	Average	Worst
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Remove	O(log(n))	O(n)
Search	O(log(n))	O(n)

- It is observed that BST's worst case performance is closest to linear search algorithms, that is O(n)
- In real time data, we cannot predict data pattern & their frequencies.
- So, a need arises to balance out the existing BST.

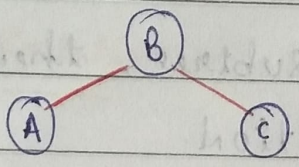
* Named after their inventor Adelson Velski & Landis, AVL ~~trees~~ trees are height balancing

Page No :
Date :

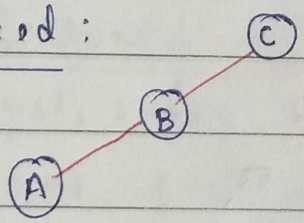
BST.

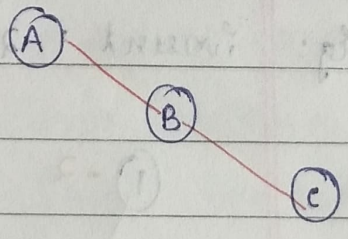
* AVL trees checks the height of the left & right sub-trees & assures that the difference is not more than 1.
* This difference is called the **Balance Factor**.

Balanced :



Unbalanced :





In the second tree, the left subtree of C has height 2 & the right subtree has height 0, so the difference is 2 in the third tree, the right subtree of A has height 2 & left is missing, so it is 0 & the difference is 2 again. AVL tree permits difference to be only 1.

Balance factor = height(left-tree) - height(right)

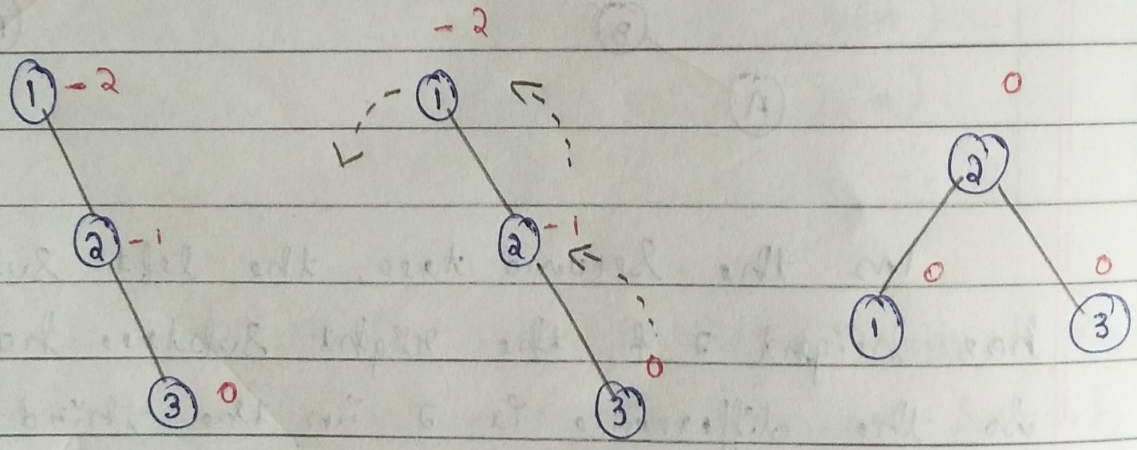
If the difference in the height of left & right sub tree is more than 1, the tree is

balanced using some rotation techniques.

1) Single Left Rotation (LL Rotation)

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.

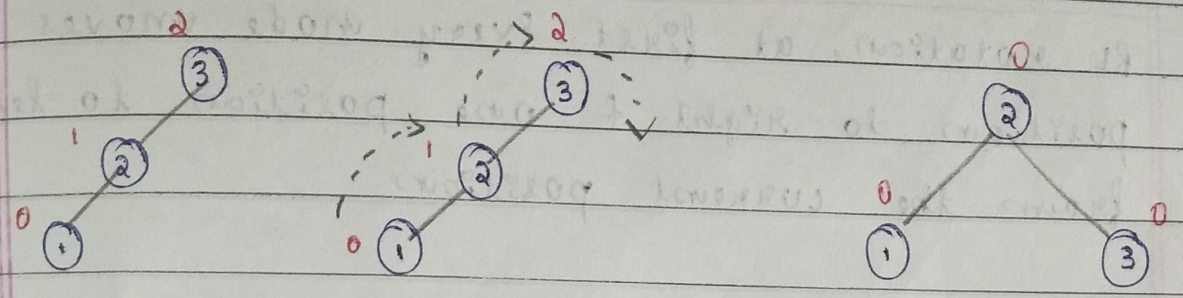
: insert 1, 2 and 3



2) Single Right Rotation (RR Rotation)

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

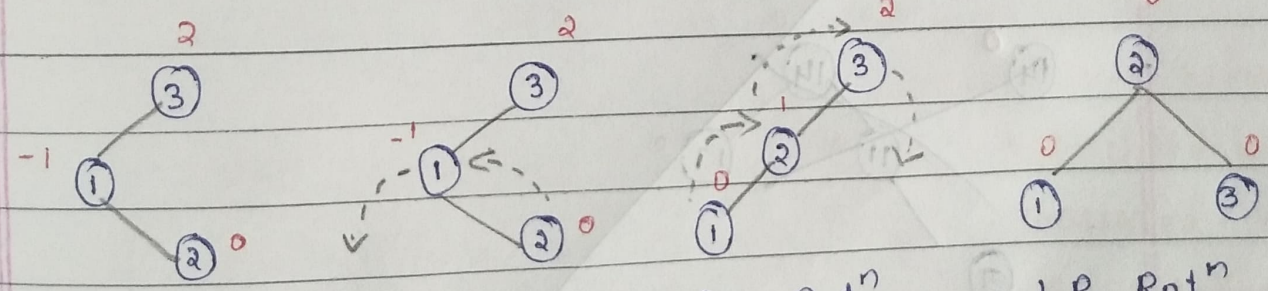
Eg: insert 3, 2 and 1



3) Left Right Rotation

A left-right rotation is a combination of left rotation followed by right rotation. In LR rotation, at first, every node moves one position to the left & one position to right from the current position. To ~~understand LR Rotation~~,

Eg: insert 3, 1 & 2



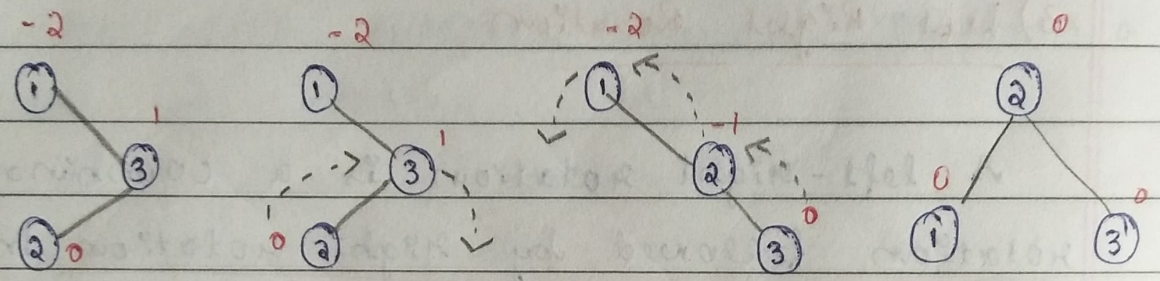
LL Rotⁿ RR Rotⁿ LR Rotⁿ

4) Right Left Rotation

The RL Rotation is sequence of single right

rotation followed by Single left Rotation. In RL rotation, at first every node moves one position to right & one position to left from the current position.

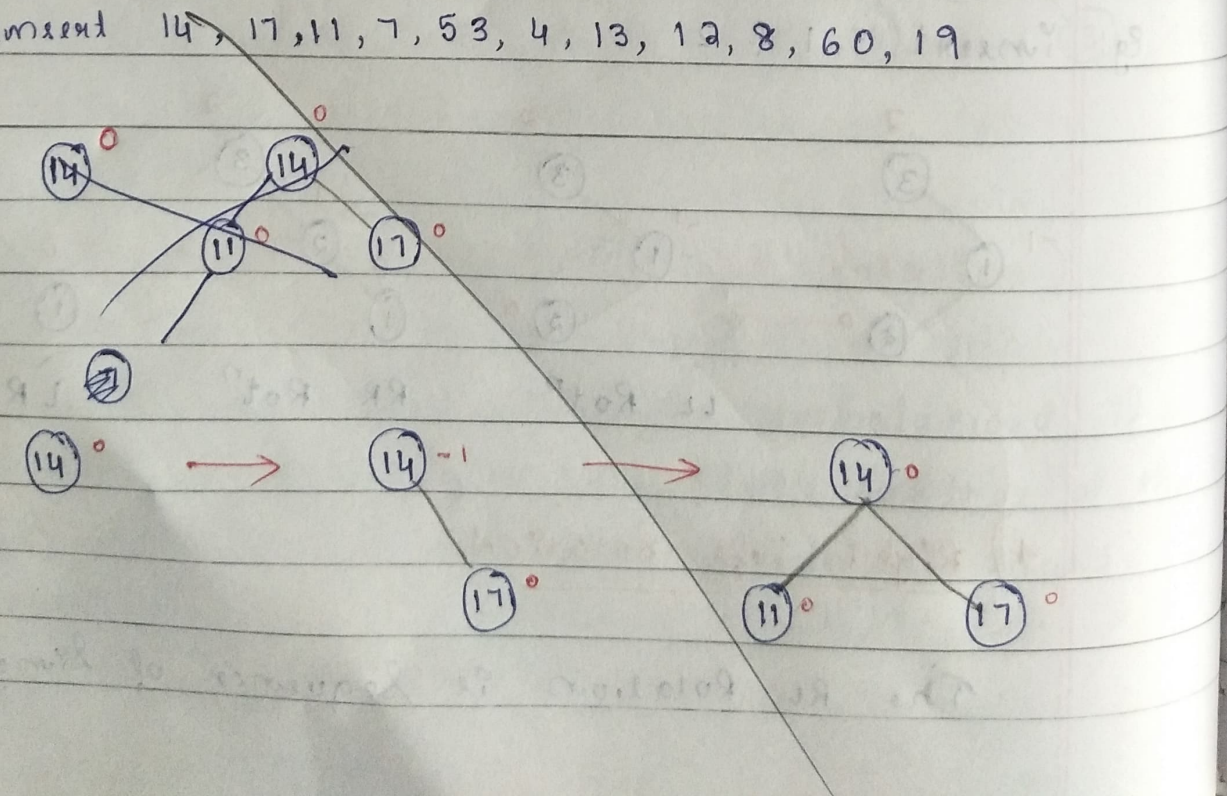
Eg: insert 1, 3 and 2

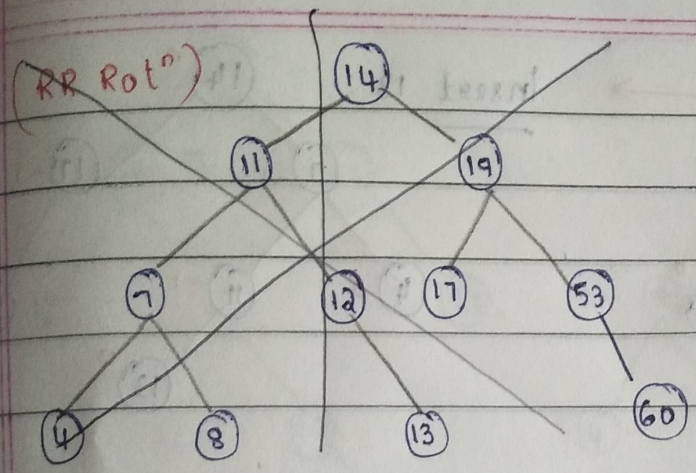


RR Rotation LL Rotation After RL

12021

Q. Insert 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19

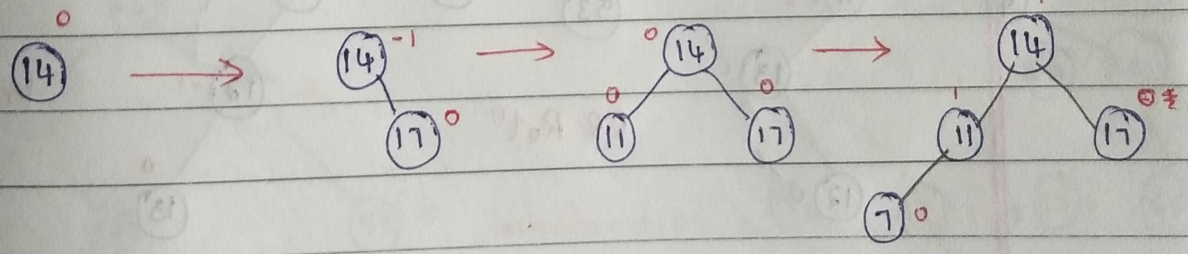




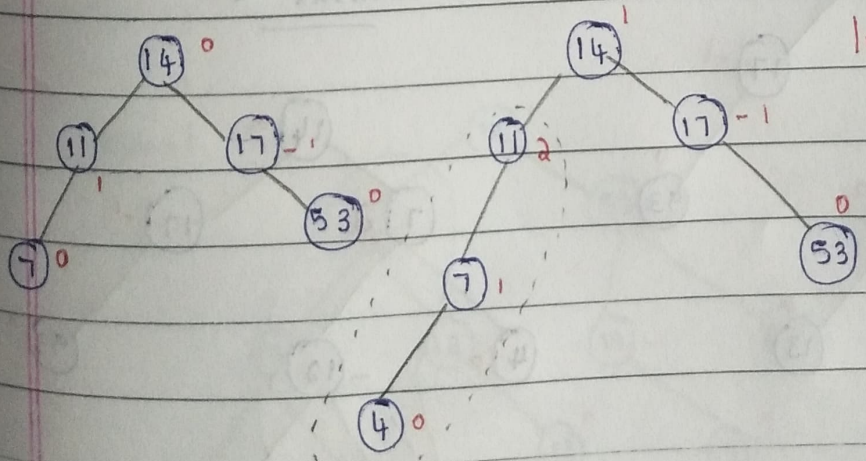
6/1/2021

Q) Insert 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19

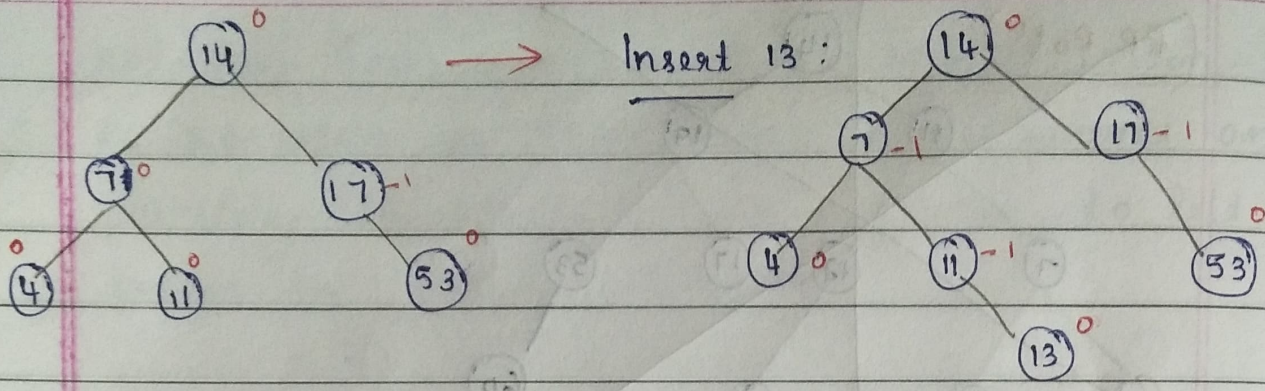
Insert 14 | Insert 17 | Insert 11 | Insert 7



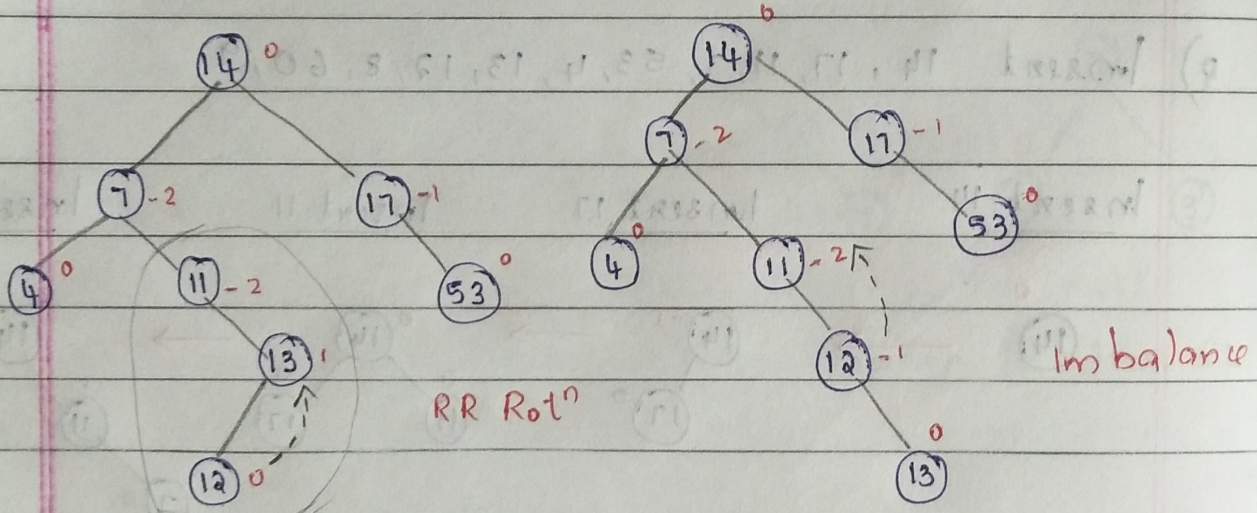
Insert 53 → Insert 4

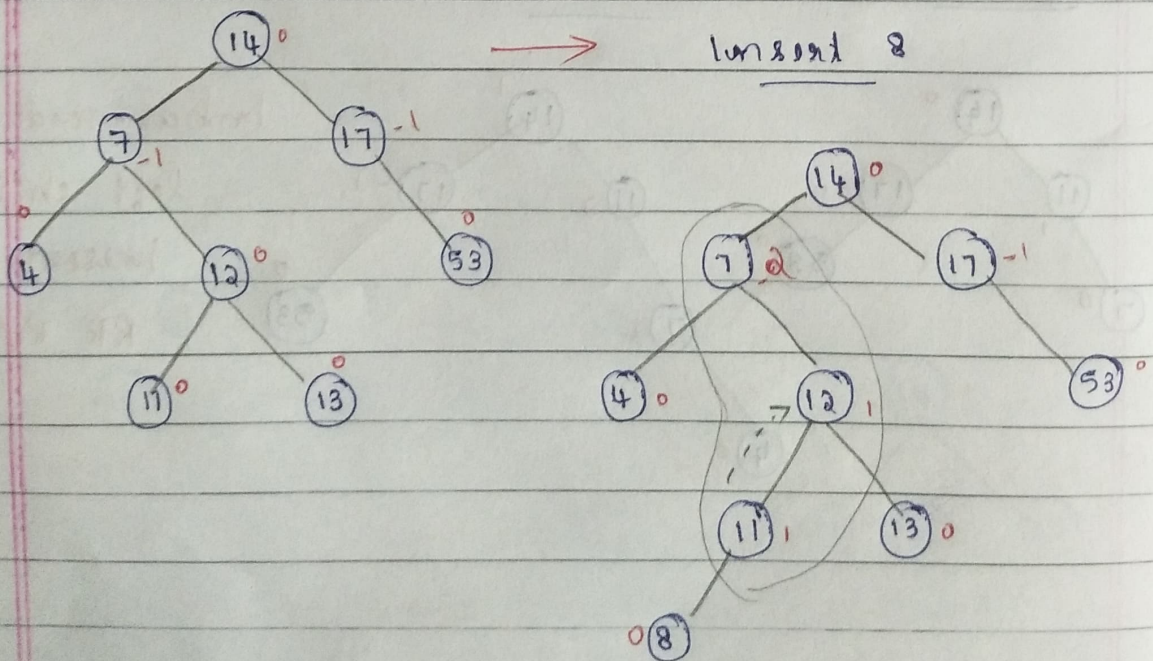


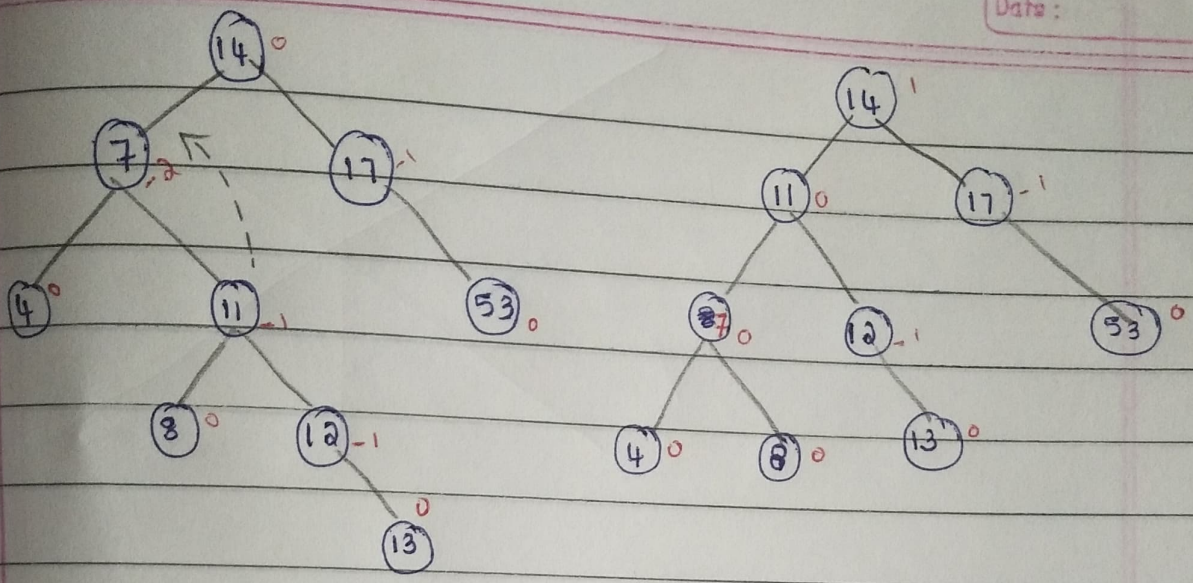
Imbalanced when left child is inserted. so RR Rotation


14
7
17
4
11
53
Insert 13 :
13

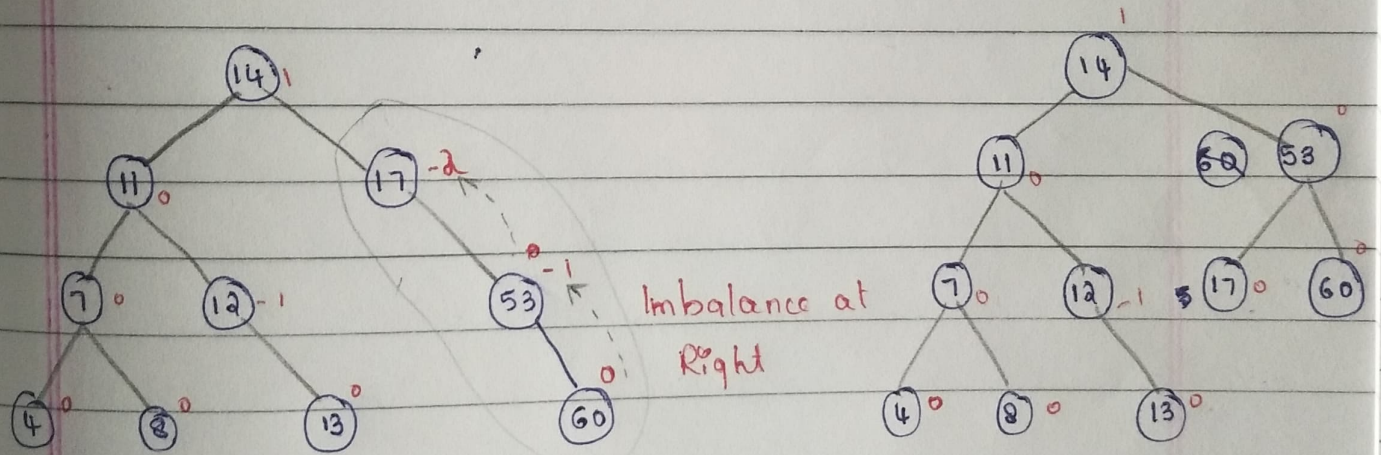
Insert 12


14
7
17
4
11
53
13
12
RR Rotn
Imbalance

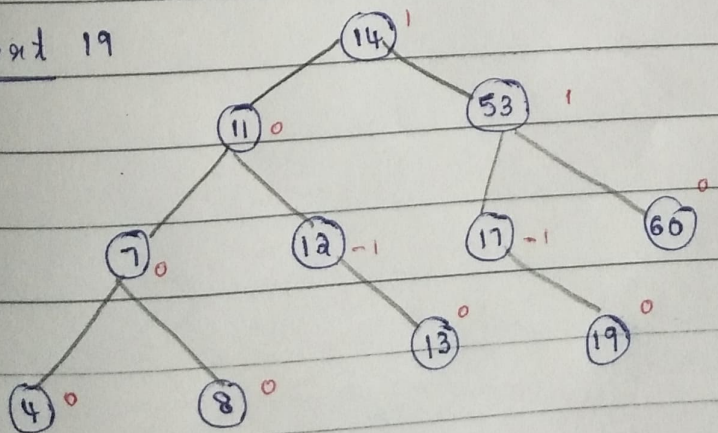

14
7
17
4
12
53
11
13
Insert 8
8



Insert 60



Insert 19



Red-Black Tree Properties

A RBT is a binary tree that satisfies the following red-black properties.

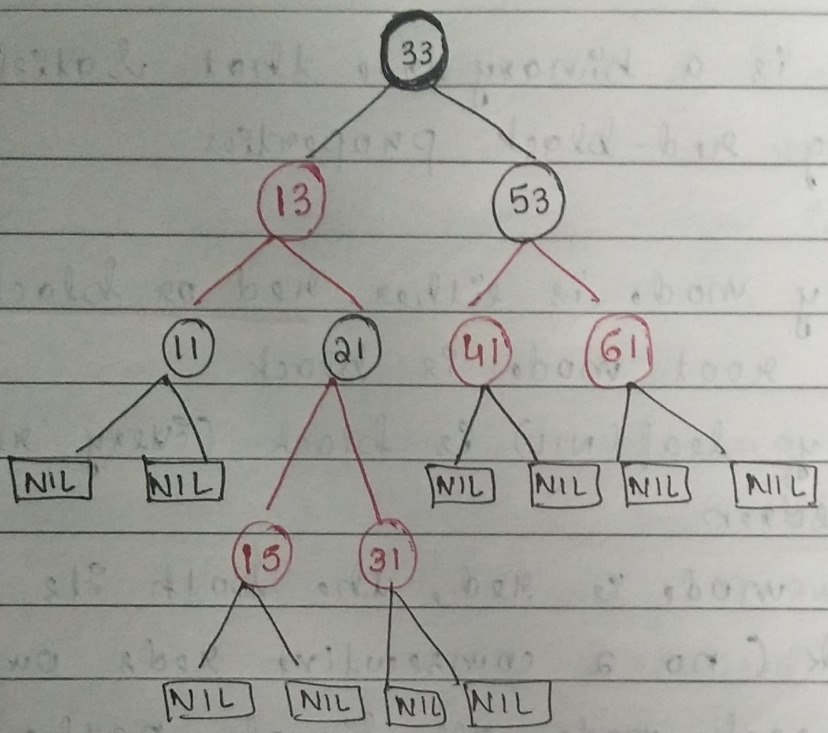
1) Every node is either red or black.
2) The root node is black.
3) Every leaf (NIL) is black (Every 'real' node has 2 children
4) If a node is red, the both its children are black. (no 2 consecutive reds on a path)
5) For each node, all simple paths from the node to descendant leaves contains the same number of black nodes.

(Most of the BST operations take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed binary tree.)

The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such paths. It helps in maintaining

the Self-balancing property of the RBT

Eg:



2021

Rotation

→ Inorder to alter the structure of a tree by rearranging subtrees.

→ goal is to decrease the height of the tree

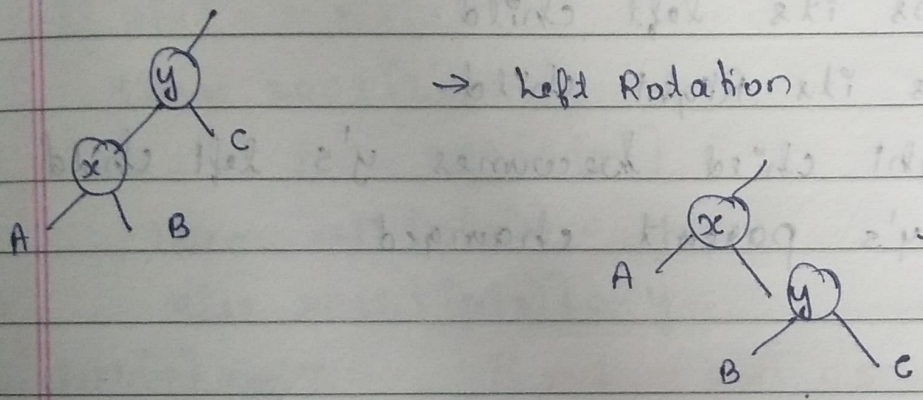
• red-black trees: maximum height of O(log n)

• larger subtree up, smaller subtrees down.

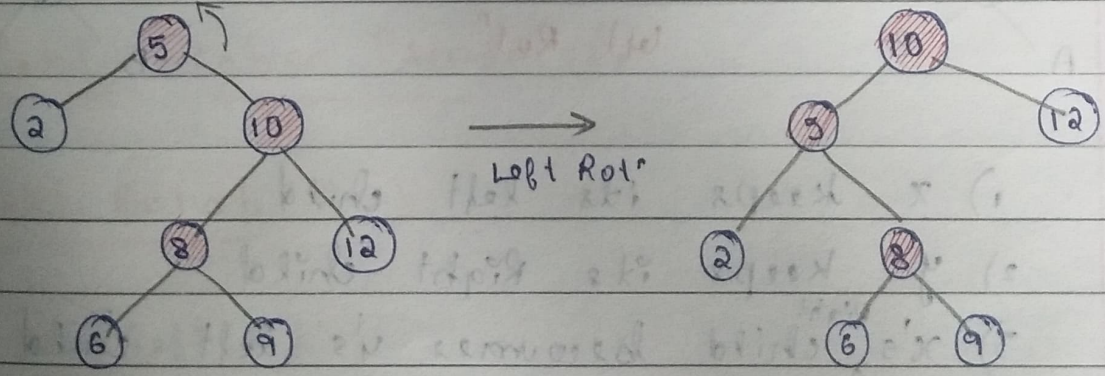
→ doesn't affect the order of elements

2 Types of Rotation : 1) LEFT ROTATION
2) RIGHT ROTATION

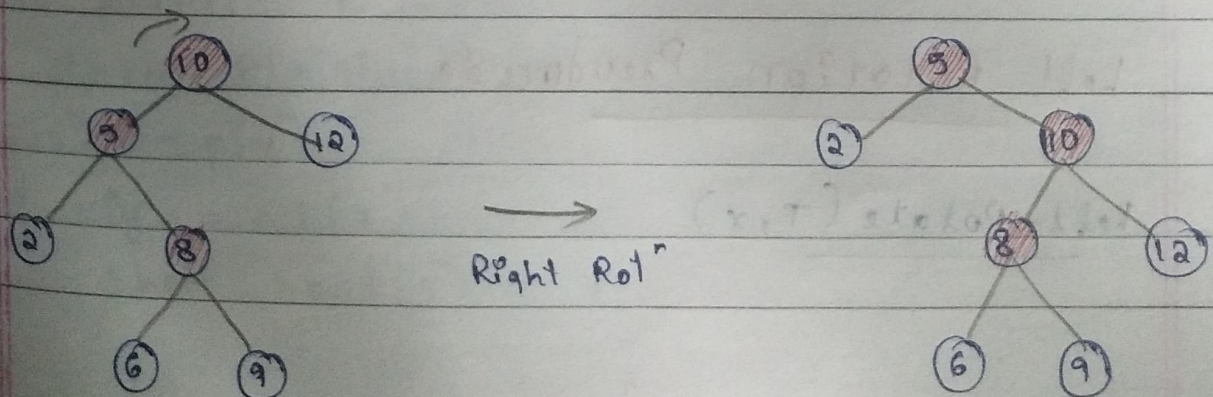
The basic Operation for changing tree structure is called Rotation.



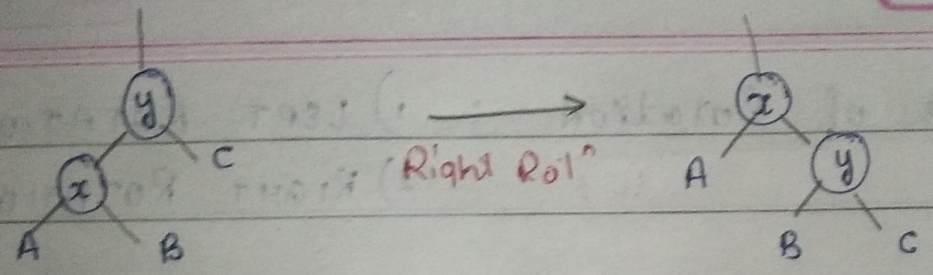
Left Rotate



Right Rotation



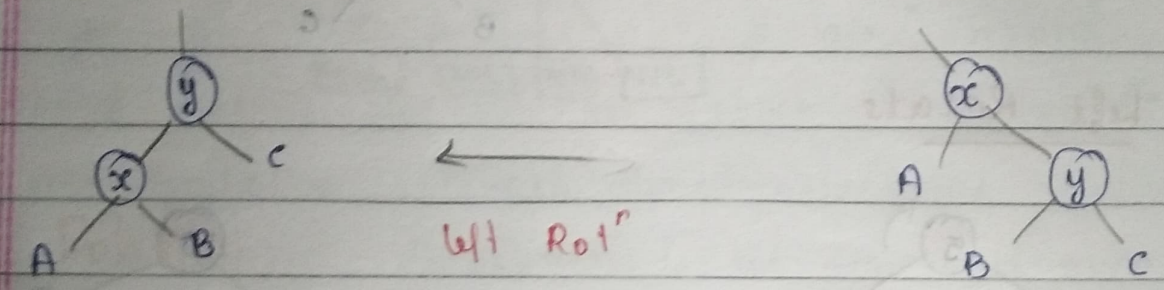
Right Rotation



- A lot of pointer Manipulation

1) x keeps its left child.
2) y keeps its Right child.
3) x's right child becomes y's left child
4) x's & y's parent changed.

Left Rotation



1) x keeps its left child
2) y keeps its Right child
3) x's Right child becomes y's left child
4) x's & y's parent changed

Left Rotation Pseudocode

Left-Rotate (T, x)

```
1.  y ← right[x]          // set y
2.  right[x] ← left[y]    // Turn y's left subtree into
3.  if left[y] ≠ nil[T]      x's right subtree
4.     then p[left[y]] ← x
5.  p[y] ← p[x]           // Link x's parent to y
6.  if p[x] == nil[T]
7.     then root[T] ← y
8.     else if x == left[p[x]]
9.        then left[p[x]] ← y
10.       else right[p[x]] ← y
11. left[y] ← x           // put x on y's left
12. p[x] ← y
```

Right Rotation Pseudocode

```
right-Rotate(T, y)

1.  x ← left[y]
2.  left[y] ← right[x]
3.  if right[x] ≠ nil[T]
4.     then p[right[x]] ← y
5.  p[x] ← p[y]
6.  if p[y] == nil(T)
7.     then root(T) ← x
```

Date :

8. else if y == left[p[y]]
9. then left[p[y]] ← x
10. else right[p[y]] ← x
11. right[x] ← y
12. p[y] ← x

Insertion

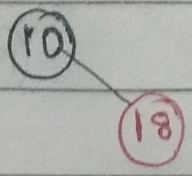
1) If a tree is empty, create newnode as root node with color black.
2) If a tree is not empty, create a newnode as leafnode with color Red.
3) If parent of newnode is black, then Exit.
4) If parent of newnode is Red, then check the color of parent's uncle sibling of newnode.
5) a) If color is black or null, then do suitable rotation & recolor.
6) b) If colour is Red then recolor & also check if parent's parent of newnode is not root node. then recolor it & recheck.

12/1/21

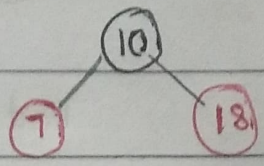
Q. Construct a RBT with following nodes
10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70

→ Insert 10

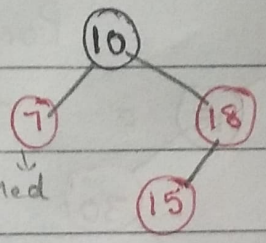
→ Insert 18



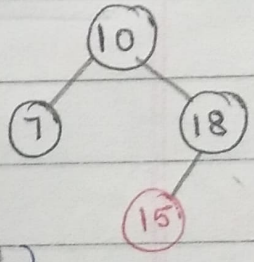
→ Insert 7



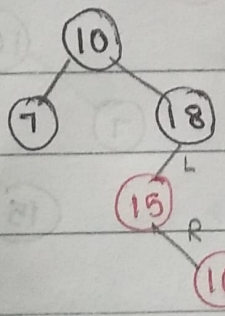
→ Insert 15



→ violates RBT Property (Recolor uncle & Parent)

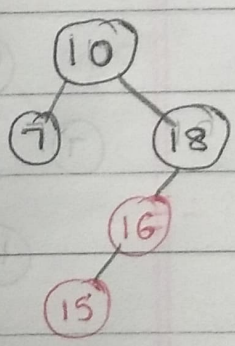


→ Insert 16

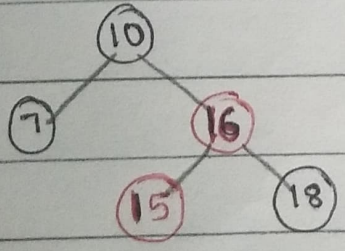


→ violate RBT Prop.
Parent Red → Parent Sibling

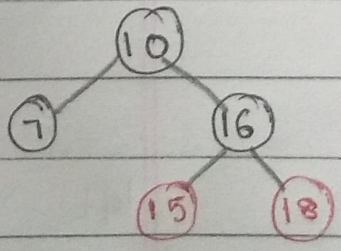
LR



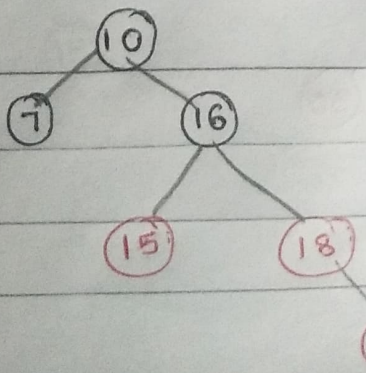
→



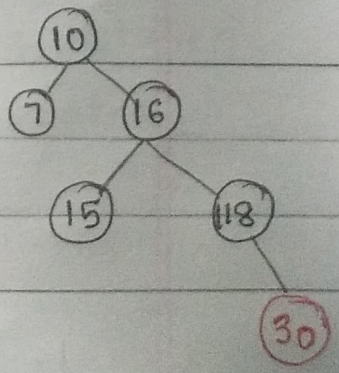
→ Recoloring
Recolor Parent & Grandparent After Left Rotation



→ Insert 30

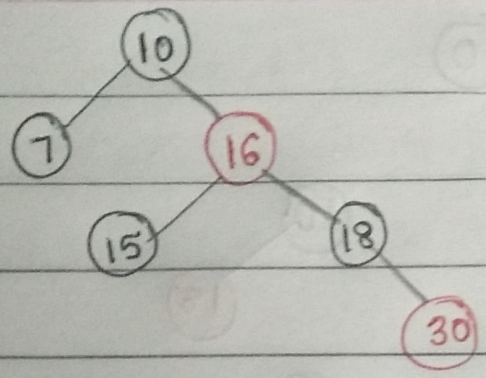


→ uncle color is Red - Recolor

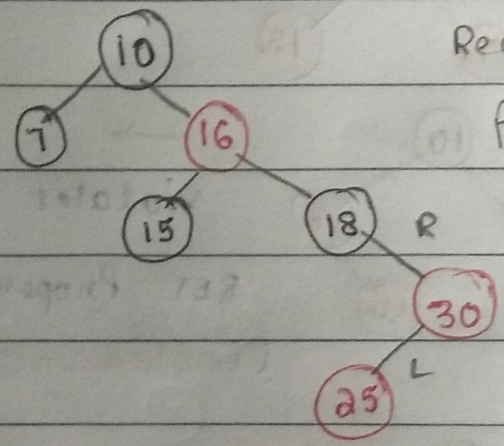


Date :

GrandParent Not Root Node. So Recolor GP



→ Insert 25



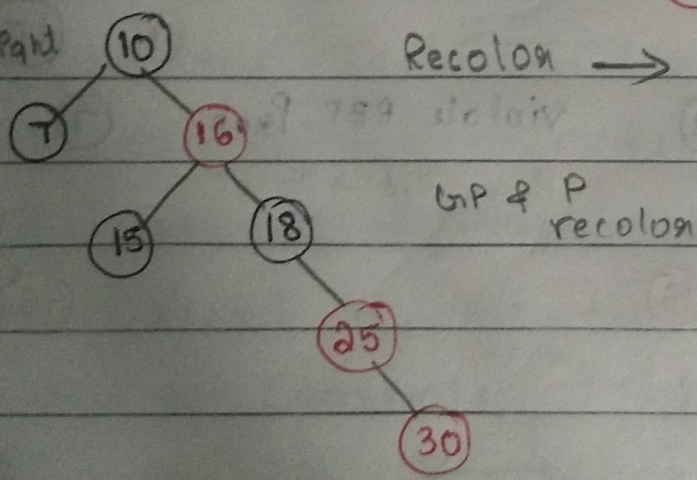
Red consecutive

Parent color = R

Color of uncle = NIL

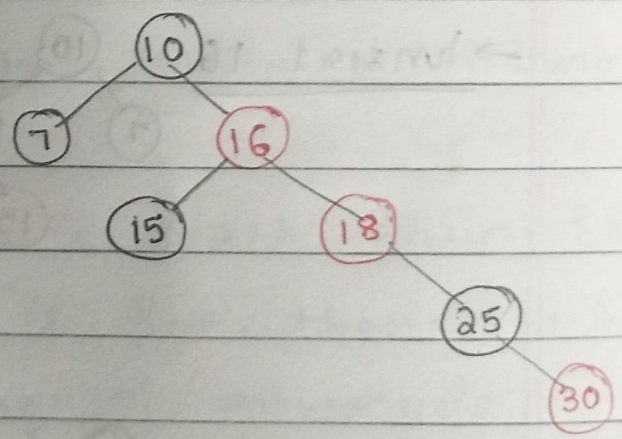
Rotation & Recolor

Right

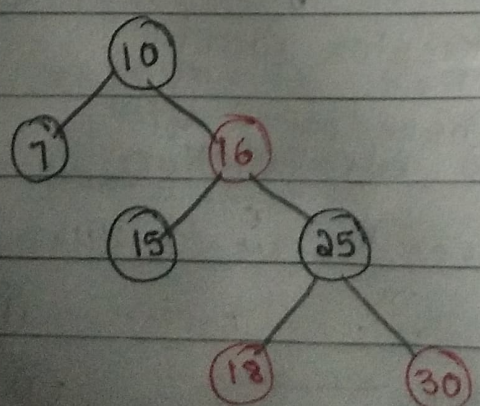


Recolor →

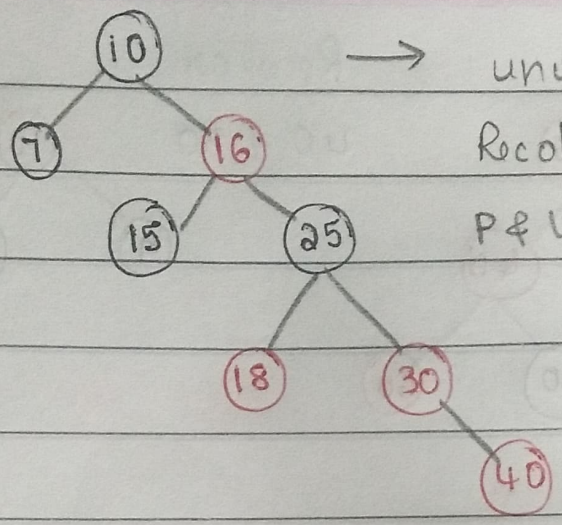
GP & P recolor



Left



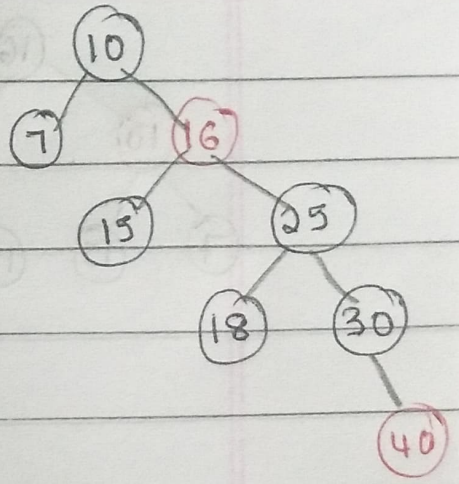
→ Insert 40



→ uncle Red

Recolor

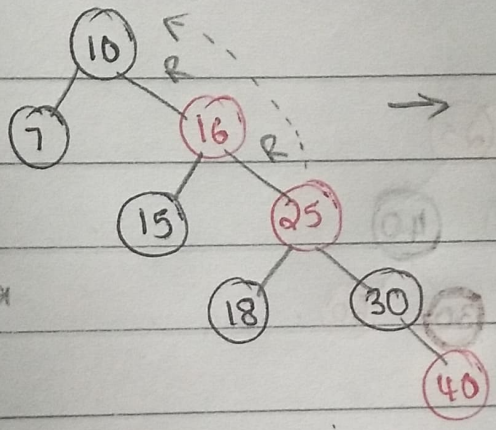
P & Uncle



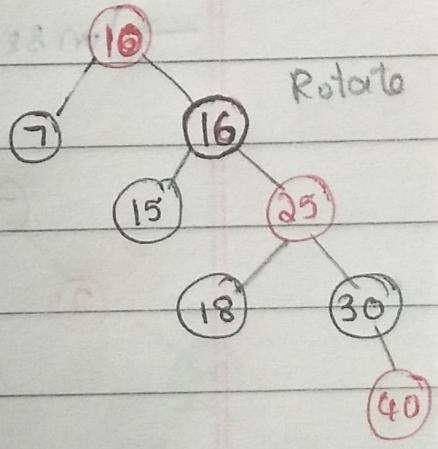
→

GP is not Root Node.

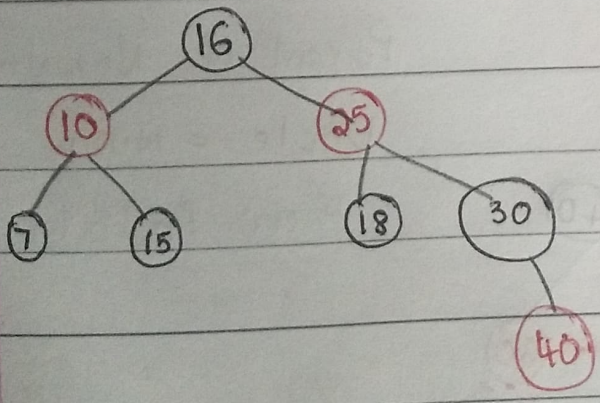
then Recolor



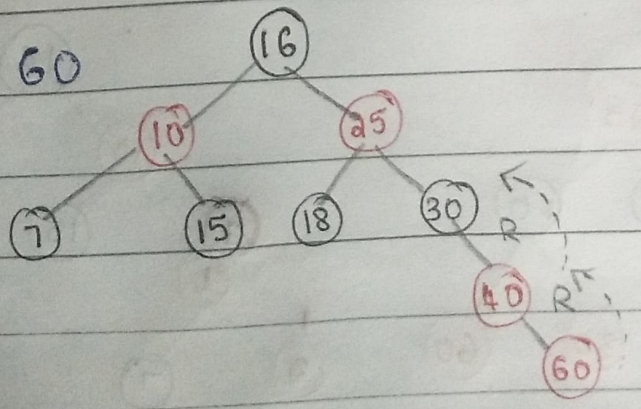
→ uncle of 16 is 7 = black



Rotate



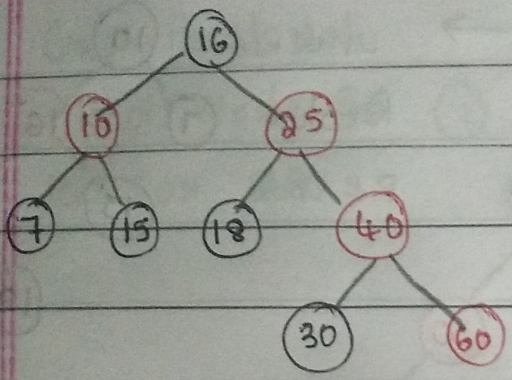
→ Insert 60



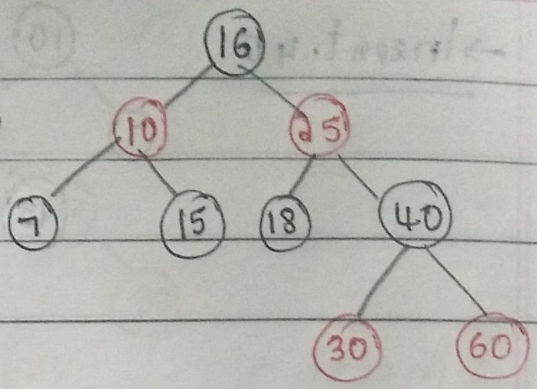
Parent Sibling = NULL

then do Rotation & Recolor

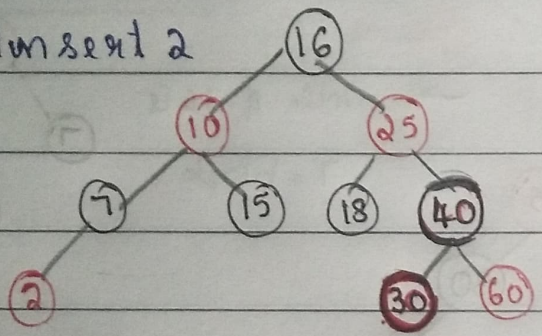
Date :



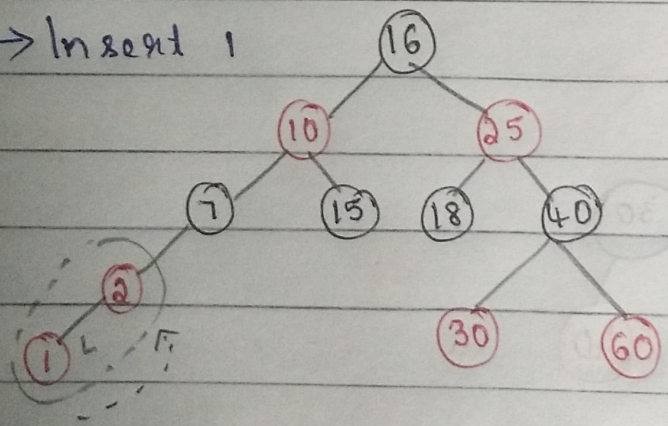
Recolor
40 & 30



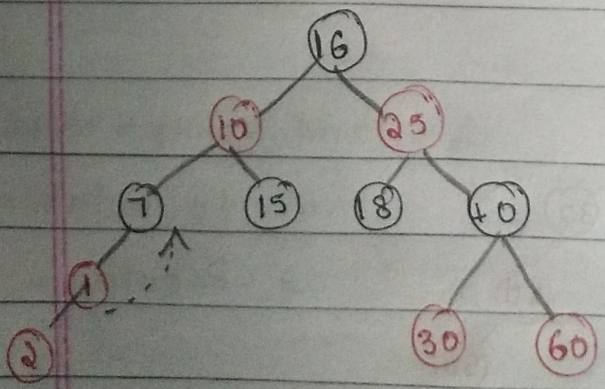
→ Insert 2

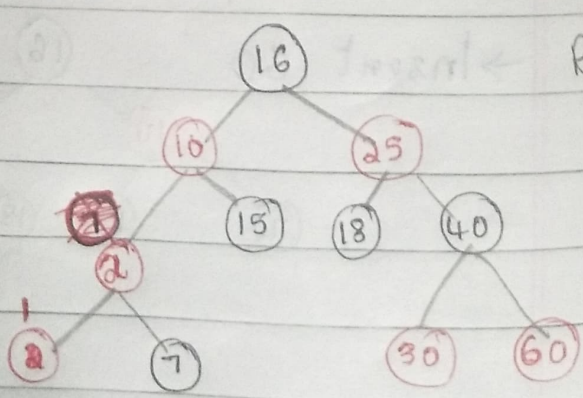


→ Insert 1

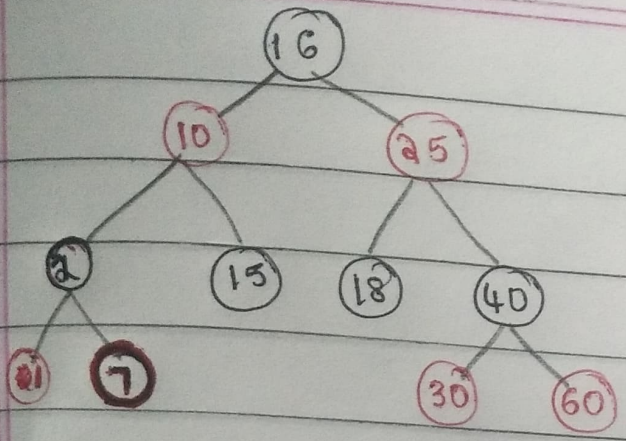


Parent of NewNode = Red
Uncle = NULL
then Rotate

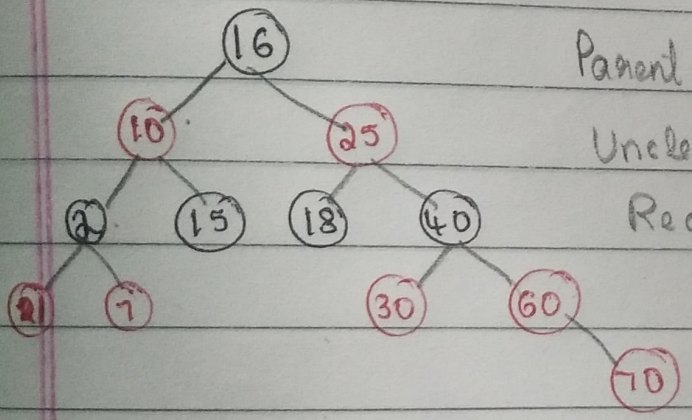




Recolor



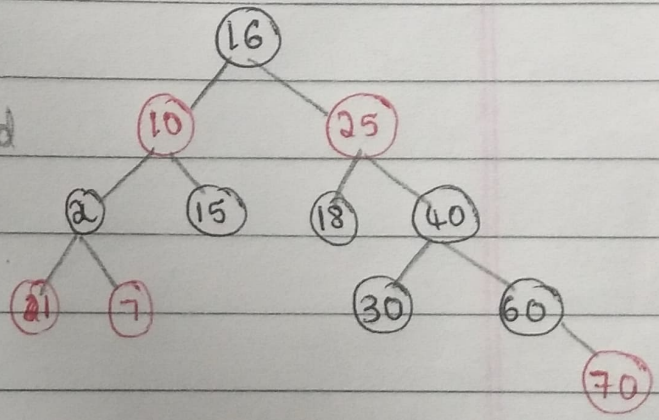
→ Insert 70



Parent = Red

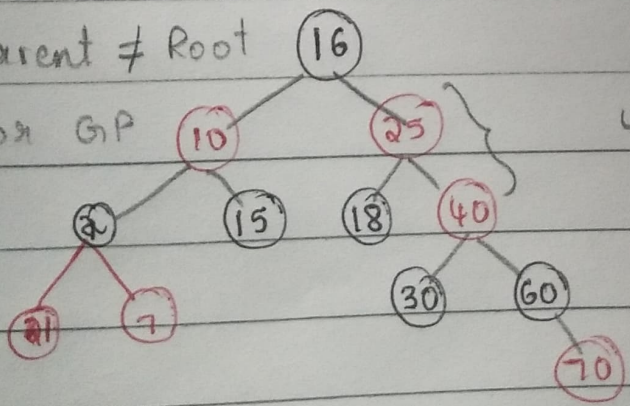
Uncle = Red

Recolor



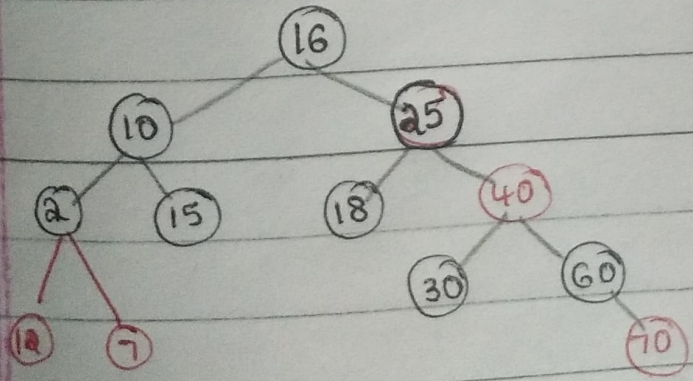
GrandParent ≠ Root

Recolor GP



uncle = red (10)

Recolor uncle & Parent



Page No :
Date :

Deletion

1) Perform deletion as in BST

2)

i) If node to be deleted is red, just delete it

ii) If root is double black, just remove double black.

iii) If double black's sibling is black & both its children are black

a) Remove double black

b) Add black to double black's parent.

* If parent is red, it becomes black

* If parent is black, it becomes double black.

c) Make Sibling Red

d) If still double black Exists apply other cases

iv) If double black's Sibling is red

a) Swap colours of parent of double black & its Sibling.

b) Rotate parent in double black's direction.

c) Reapply cases

v) If double black's sibling is black, Siblings child who is far from double black is black but child which is near to double

black is red.

a) Swap colors of double black's Sibling & Sibling's child who is near to double black

b) Rotate Sibling in opposite direction to double black

c) ~~Rotate Sibling in opposite direction to~~ double black Apply case (vi)

vi) If double black's Sibling is black, far child is red

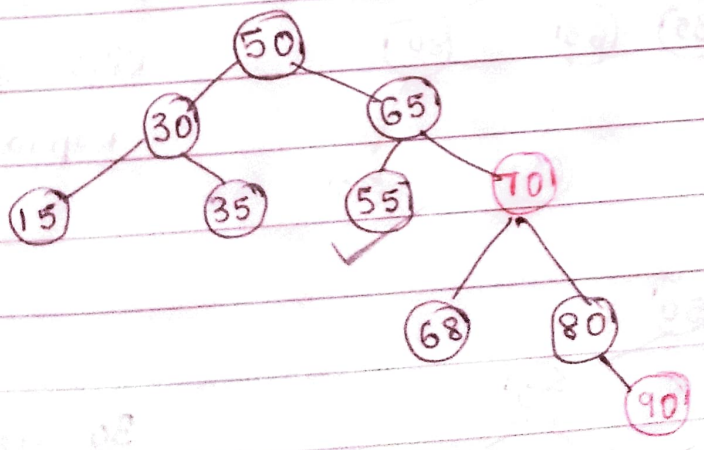
a) Swap colors of parent of double black & Sibling.

b) Rotate parent in double black's direction.

c) Remove double black.

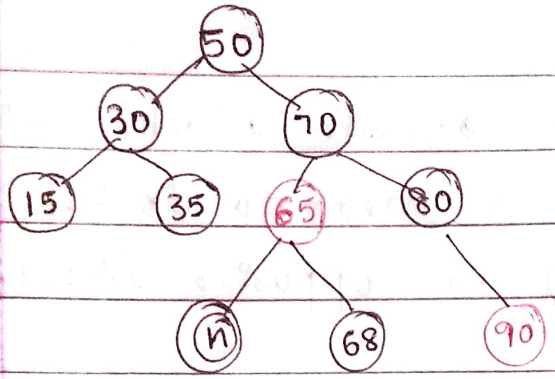
d) Change color of red child to black.

Eg. Delete node 55

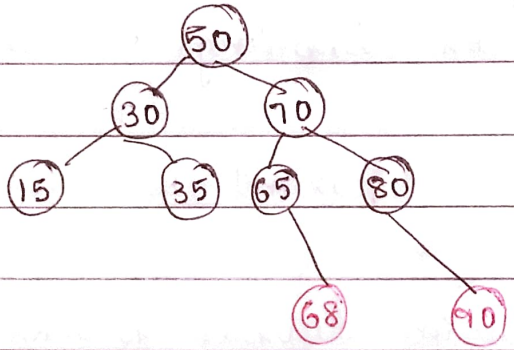


55 is a leaf Node. We can delete it But it is a black Node. Arises a double black situation

Case (iv)

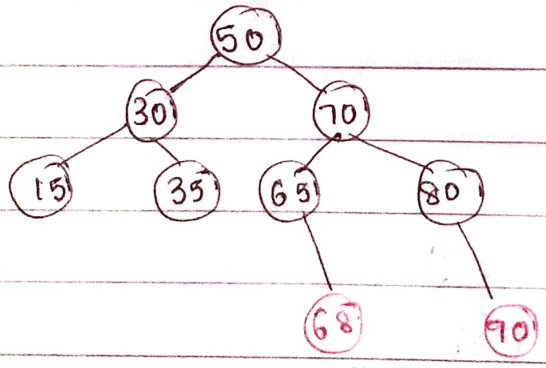


55 → black Node
Sibling = Red
Swap color of Parent & Sibling
Rotate parent in DB
→ n is DB — case 3

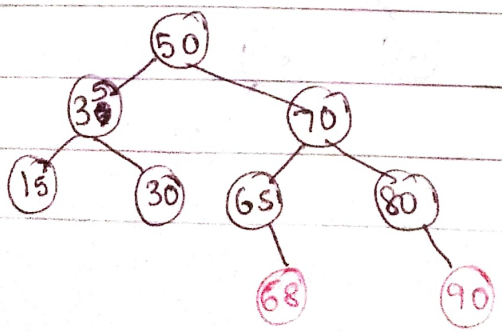


black sibling (68) with black child (NULL).
Remove DB. Add Black to Parent & Make Sibling Red.

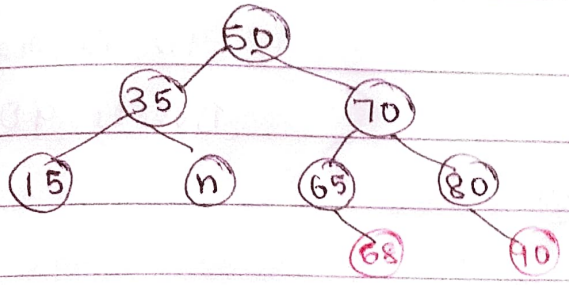
2) Delete 30



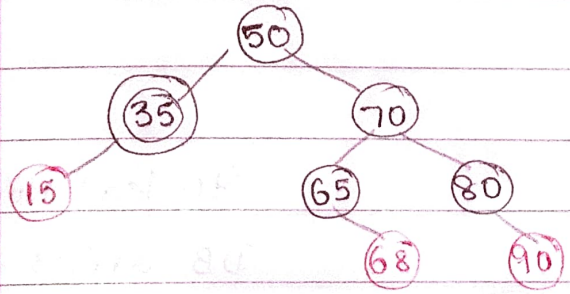
30 have 2 children (15 & 35): Inorder Successor. Right subtree (35). then Replace



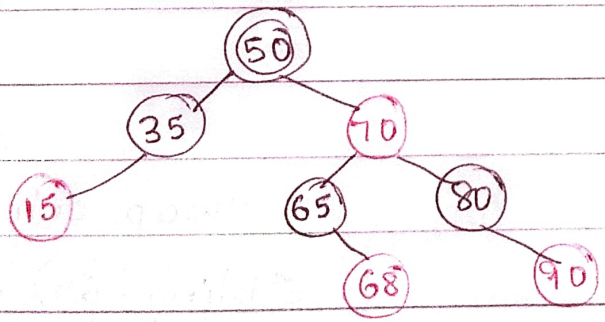
30 delete. DB arises colour of sibling (15) is black with black child



then change color of parent (35). 35 is black so 35 become DB & sibling(15) become red

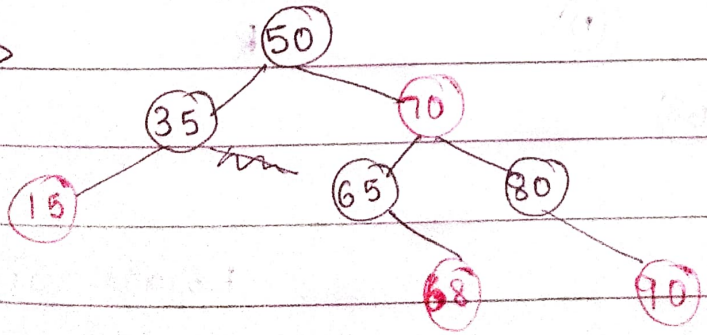


⇒ DB so sibling of 35 is 70. 70 is black with 2 children(65 & 80) make Parent Black(50) & sibling(70) Red



DB at root. so Remove DB

⇒



3) Delete 90