mood-book

UNIT- III

Search Trees: Binary Search Trees, Definition, Implementation, Operations - Searching, Insertion and Deletion. AVL Trees, Definition, Height of an AVL-Trees, Operations - Insertion, Deletion and Searching. Red-Black, Splay Trees.

Binary Search Trees:

* A Binary Search Tree is a special kind of binary tree that satisfies the following conditions.

(*) The data elements of the left subtree are smaller than the root of the tree.

(*) The data elements of the right subtree are greater than or equal to the root of the tree.

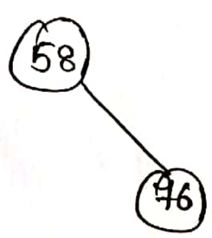
(*) The left sub tree and right subtree are also the binary search trees. i.e., they must also follow the above two rules.

Construction of a Binary Search Tree (BST):-

* Construct the BST of 58, 76, 14, 63, 63, 78, 43, 6, 11, 61.

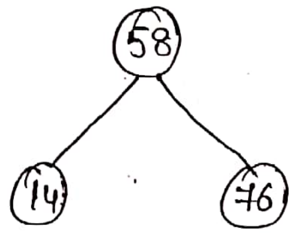
Step 1: Initially, the tree is empty so place the first no of at the root.

Step 2: Compare the next number (76) with the root. If the incoming no. is greater than or equal to the root then place it in the right child position. Otherwise place it into the left child position.

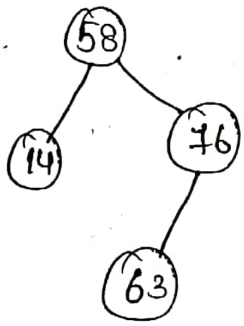


Step 3: Compare the next number (14) with the root number

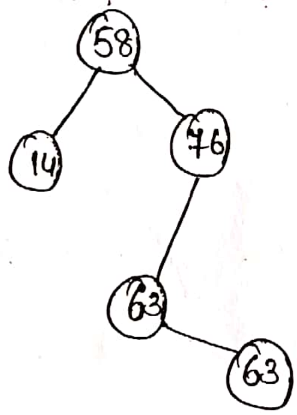
if it is less than the root so examine the left subtree.



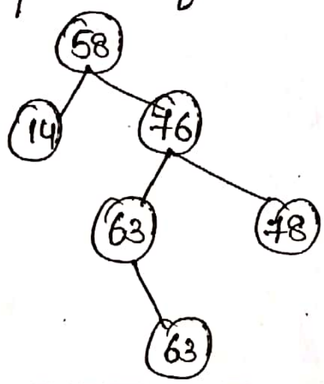
Step 4: Now, the next incoming number 63 is greater than 58 so go to the right subtree where compare it with 76.
Since $63 < 76$ so place it in the left child's position of 76.



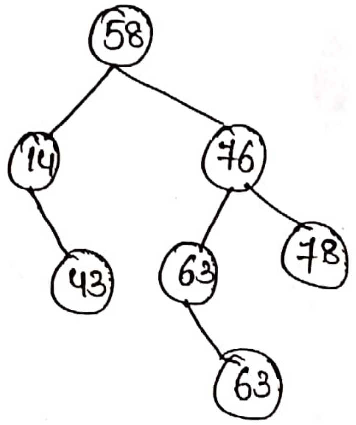
Step 5: Repeat the process for the next number 63.



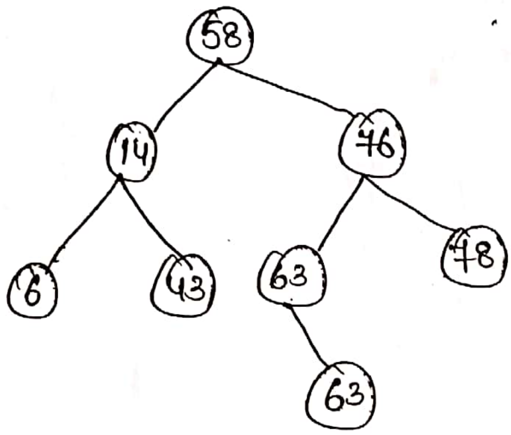
Step 6: Repeat the process for the next number 78.



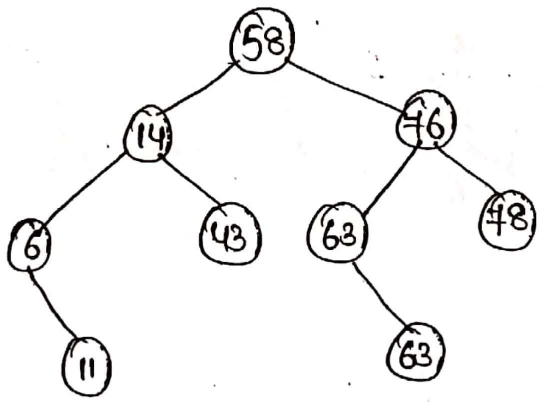
Step 7: Repeat the process for the next number 43.



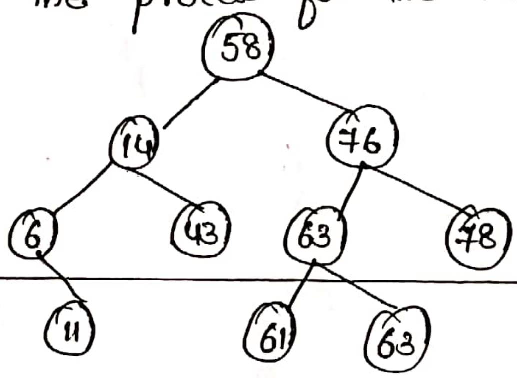
Step 8: Repeat the process for the next number 6



Step 9: Repeat the process for the next number 11.



Step 10: Repeat the process for the next number 61.

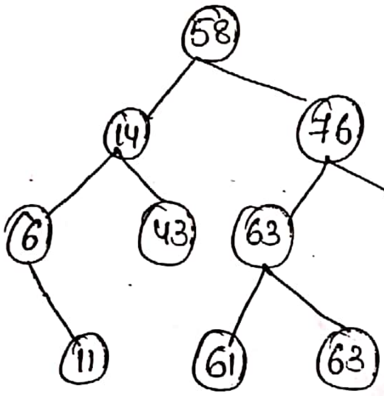


Operations on the BST:-

Traversal of BST:

* BST is also a binary tree so it can be traversed in 3 ways.

1. Pre order 2. Post-order 3. In-order.



Pre-order: 58 - 14 - 6 - 11 - 43 - 76 - 63 - 61 - 63 - 78

Post-order: 11 - 6 - 43 - 14 - 61 - 63 - 63 - 78 - 76 - 58

In-order: 6 - 11 - 14 - 43 - 58 - 61 - 63 - 63 - 76 - 78.

* Inorder traversal of BST produces the list of tree elements in ascending order. This is an important property of BST that in-order traversal of it arranges the tree elements in ascending order.

* Insertion into the BST:

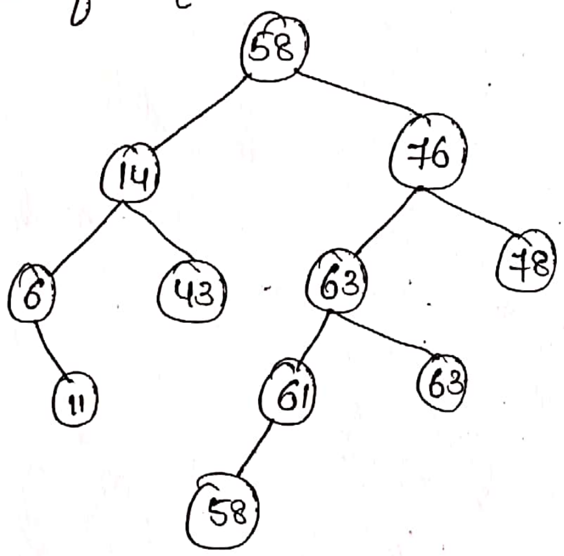
* The insertion of an element into BST follows the process as given below.

① Compare the number with the root of the BST.

② If the number is less than the root value then follow the left subtree.

③ If the number is greater than the root value then follow the right subtree.

④ Apply the same process (1, 2 & 3) to the left subtree and right subtree if required.



Deletion from the BST:-

* The deletion of data element from the BST can be perform in '3' different ways as follows:

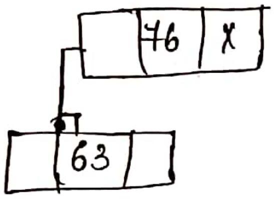
1. Deletion of the leaf (terminal) node.
2. Deletion of the node having only one node/child.
3. Deletion of the node having two children.

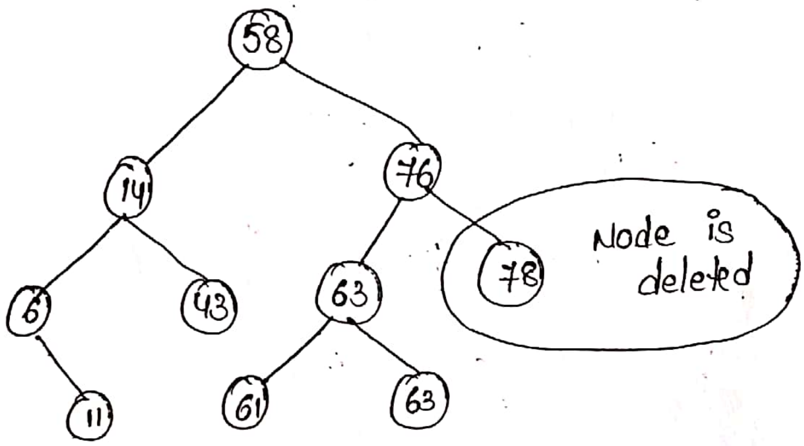
→ Deletion of the leaf node:

* In this case, we have to change the deleted nodes entry in the parent node to NULL.

Eg: The above tree, a node containing 78 is to be deleted, the right node of 76 will be deleted, then the right node of 76 is null. Therefore the parent node will now change as shown below.

* The BST of deleted element is also shown below.





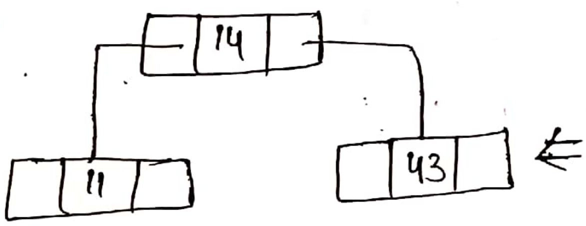
→ Deletion of a node having only one child:-

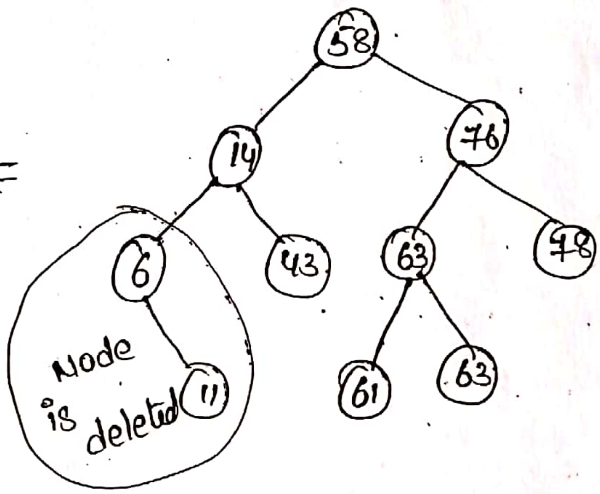
* In this case, there are 2 possibilities the node may have the left child only (or) the right child only.

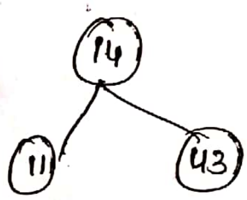
* If there is left child only then we need to attach the node's left child to the node's parent in place for the deleted node.

* And if there is a right child only then we need to attach the right child to the node's parents in place of the deleted node.

Eg:







→ Deletion of a node having two children:-

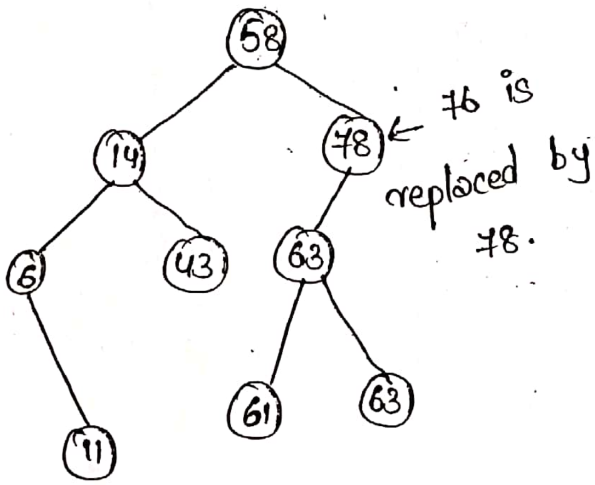
* In this case, the data element can be deleted from the middle of the tree also, but the structure & integreted of BST will not be maintained.

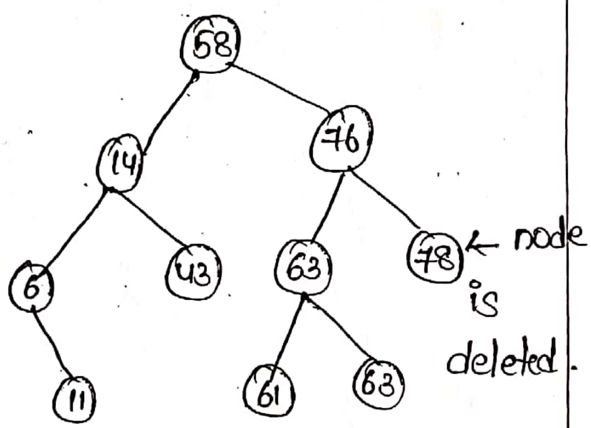
* Therefore, in this we replace the node (to be deleted) by the in-order successor of that node.

* Therefore, first find the in-order successor of the node then replace the selected node with the in-order successor and then delete the in-order successor of the node from the tree.

* For example, if we want to delete the node containing 76 from the tree, first we have to find the in-order traversal sequence of the tree i.e., 6-11-14-43-58-61-63-63-76-78.

* In the in-order sequence, we can see that the in-order successor of the selected node (76) is 78. So first replace 76 by 78 and then delete the node containing 78.



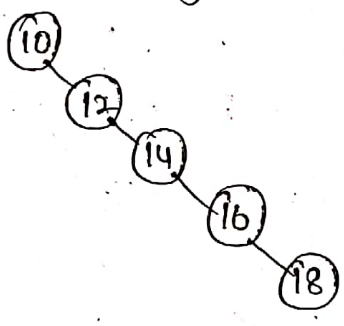


Balanced Tree:-

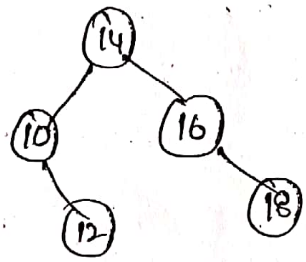
* The Binary search tree makes searching easier if it contains half of the elements in the left subtree and almost half of the elements in the right subtree.

* The left and right subtree themselves follow the same composition. However, it is not possible all the time because it depends on the elements inserted.

Eg: The following BST.



(a) Unbalanced BST



(b) Balanced BST

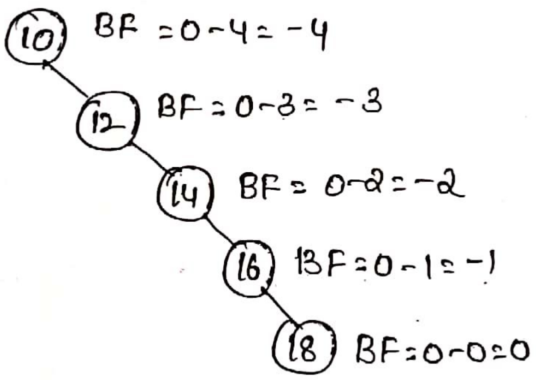
* This BST in figure (a) requires '5' comparisions to search 16 in the tree, 4 comparisions to search 14.

* However, if this is converted into the tree as shown in figure (b), it will require only 1 comparision to search 14 and 2 comparisions to search 16.

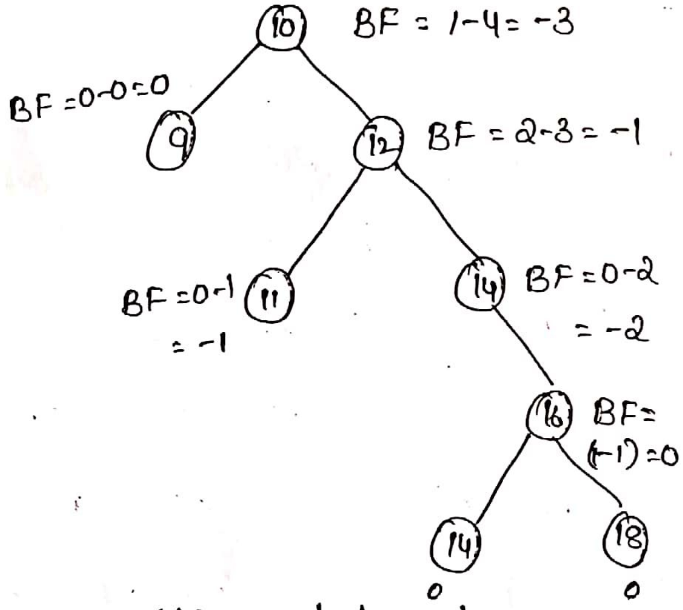
* Here, the tree is balanced.

Balance Factor (BF) = Height of the left subtree − Height of the right subtree.

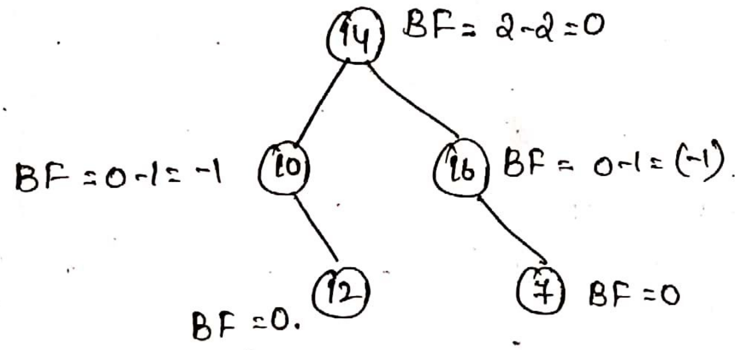
* The balanced factor of the BST is assigned in figure (a), (b) and (c).



(a) Unbalanced BST



(b) Unbalanced BST.



(C) Balanced BST

* Therefore, the searching depends on the heights of the tree in the BST, and the height changes as the insertion or deletion takes place.

* The insertion and deletion of nodes makes the tree unbalanced.

* There are different kinds of balanced trees which are balanced by working with their heights.

* Some of such trees are as follows:

1. AVL trees
2. B-Tree
3. Red-black Tree
4. Splay Tree.

~~3. 2.~~

AVL Trees:

* AVL tree is a special kind of balance tree in which the balance factor of each node cannot be other than 0, -1 or 1.
* The AVL tree was named after the Russian Scientists G.M. Adelson, Velski & E.M. Landis.
* It is also called as a "height-balanced tree". In other words, the height of the left subtree and right subtree of the node differs at the most by 1 where left and right subtrees are given AVL.
* If the balance factor of the node is -1, then the right subtree is said to be higher than left subtree. It is called "right heavy" and if the balance factor is 1, then left subtree is said to be higher than the right subtree. It is called "left heavy". and if the balance factor is 0, then both subtrees are on same height.

Representation of AVL Tree:-

* The AVL Tree can easily be implemented in the memory like other trees with an additional field to keep track of balance factor.
* The balance factor of a node represents the difference of the heights between the left and right subtrees of the node.
* Therefore, each node of the AVL tree will be represented as follows:

left	Info	Right	BF

* The nodes in the AVL trees are divided into 4 fields, the left, right pointers, info and the balance factor.

* It can be represented by the programming lang 'C' as

```
struct node
{
node * left;
int info;
node * right;
int BF;
};
struct node * AVL;
```

* **Building a Height Balanced Tree:-**

* The AVL tree is constructed by using the same procedure as applied to construct the BST.

* If the ~~small~~ new element is smaller than the root value, then it examines the left subtree, and if it is greater than or equal to the root then it examines the right subtree.

* The only thing we must remember while constructing the AVL (height balanced tree) tree is the property of the height balanced tree.

* The balance factor of each node can be 0, -1 or +1, while the insertion or deletion is performed, the tree becomes unbalanced and violates the property of AVL tree.

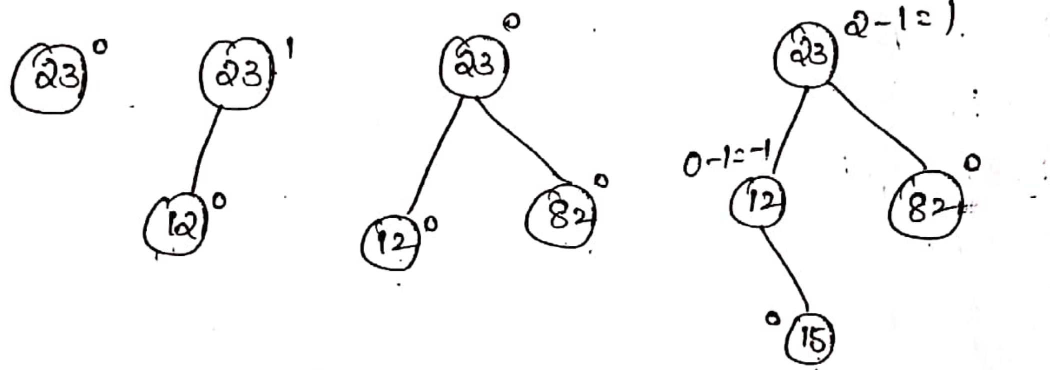
* There are different rotation methods to balance the tree.

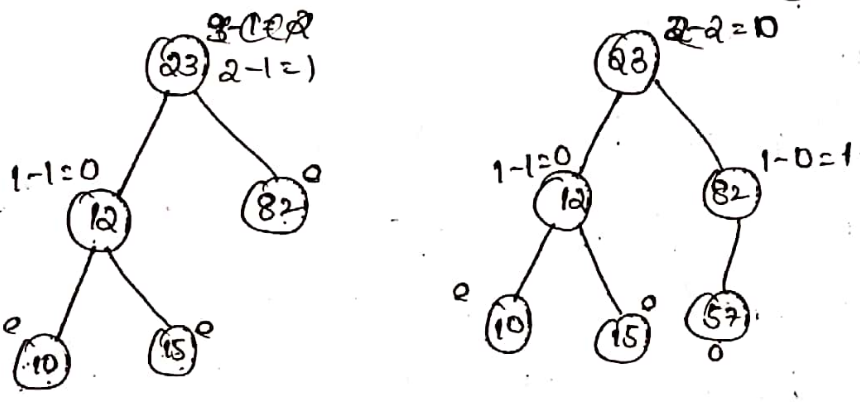
Construction of AVL tree:-

Eg: 23, 12, 82, 15, 10, 57.

Sol: The nodes of the tree with the balance factor are shown below.

BF = Height of left subtree - Height of Right subtree.





Rotations:

* Rotation is an operation on a Binary Tree that changes the structure without interfering with the order of the elements.

* The tree rotation moves one node up in the tree and one node down. It is used to change the shape of the tree and in particular to decrease its height by moving smaller subtree down and larger subtrees up, resulting in improved performance of many tree operations.

* 4 types of Rotations are there.

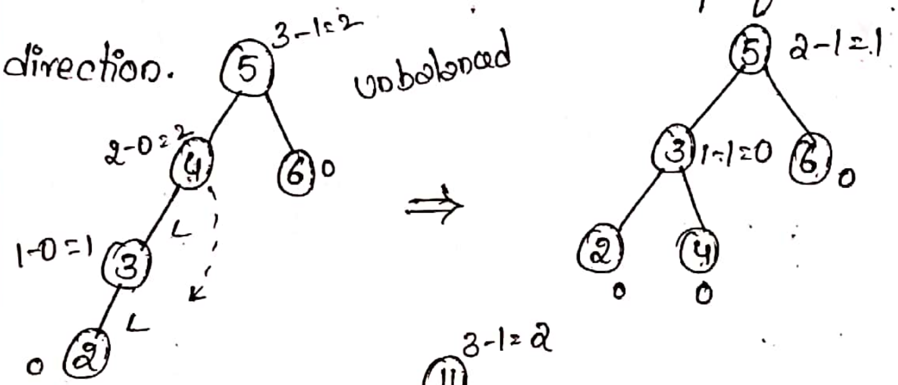
(1) Right Rotation (RR) (2) Left Rotation (LL) → Single Rotations

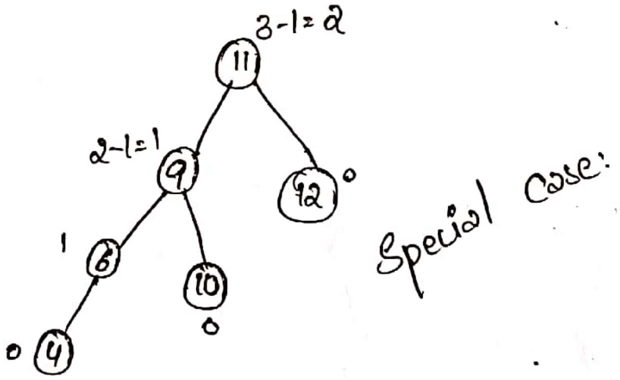
③ Left - Right Rotation ( )
④ Right - Left Rotation ( ) } Double Rotations.

LL Rotations:

* LL Rotation is a single rotation that can be applied when a node is inserted in the left subtree of the left child of a node.

* In this, rotation is performed in a clockwise direction.

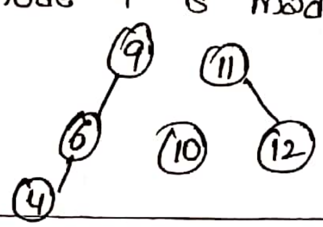




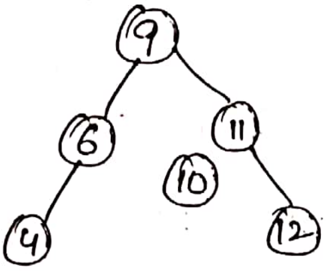
Imbalanced tree.

* After the insertion, the tree becomes imbalanced because node 11 has a balance factor 2. Thus to rebalance the tree in accordance to the balance factor -1, 0, +1, the following operations must be performed.

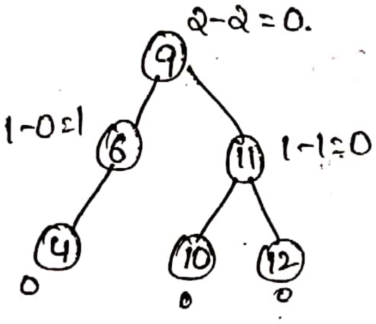
* The root of the subtree in which the node is inserted i.e., node '9' is made as a new root node.



* The original root node '11' is made as the right sub child of the new root node '9'.



* The right child of node '9' i.e., '10' is made as the left subchild of 11, where as the right child of 11, i.e., 12 remains unchanged.



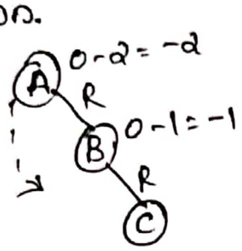
pseudo code:

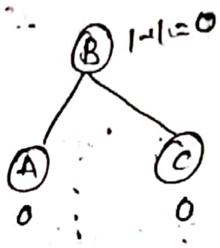
```
Node left Rotate (Node root)
{
Node newRoot = root right;
root right = new root left;
new root left = root;
root. height = max (root right, root left)+1;
new root. height = max (new root+right, new root+left)+1;
return newRoot;
}
```

RR Rotation:-

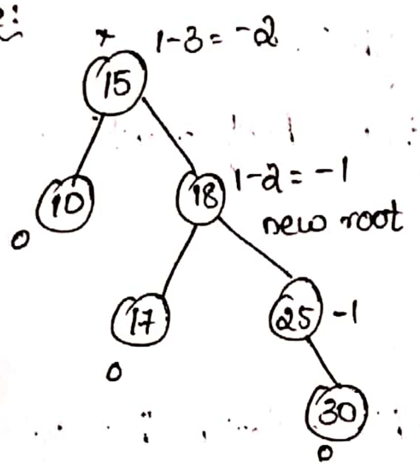
* RR rotation is also single rotation that can be performed when a node is inserted in the right subtree of the right child of a node.

* In this, the rotation is performed in an anticlockwise direction.





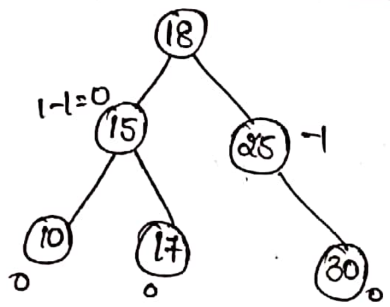
Example:



* After insertion of the '30' node, the tree becomes imbalance because node '15' has a Balance factor of (-2).

* The root of the subtree in which the node '30' is inserted (i.e., 18) is made as the new root node.

* The original root node 15, is made the left subtree/child of the new root node 18.



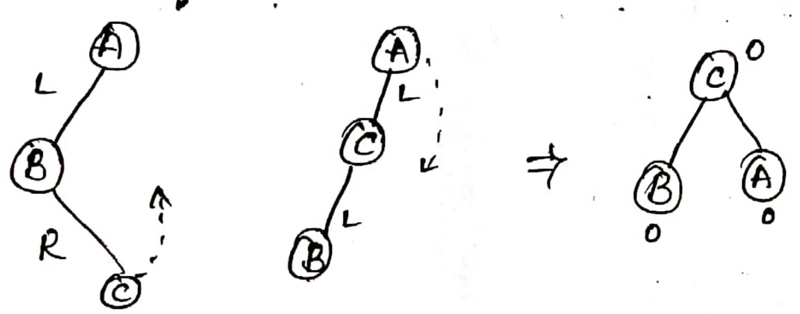
pseudo code:-

```
Node rightRotate (Node root)
{
 Node newRoot = root left;
 root left = new root right;
 new root . right = root;
root. height = max (root left, root right)+1;
new root height = max (newRoot left, newRoot right)+1;
return newRoot;
}
```

LR Rotation:-

* LR Rotation is a double rotation that can be performed when a node is inserted in the right subtree of the left child of a node.

* In this type of rotation, RR rotation followed by LL rotation are performed.

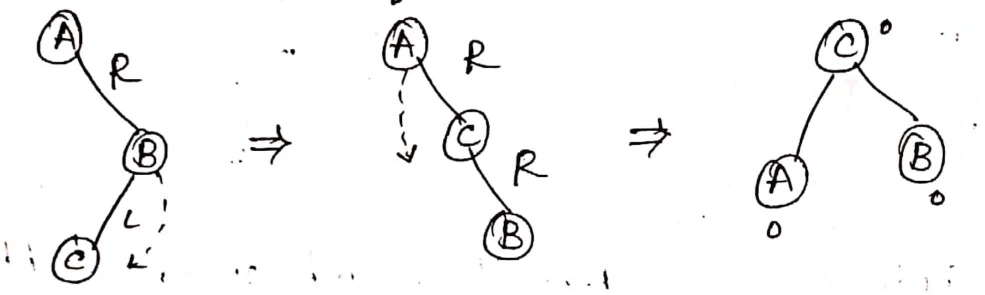


* First apply RR rotation, then apply LL Rotation.

RL Rotations:

* RL Rotation is also a double rotation that can be performed when a node is inserted in the left subtree of the right child of a node.

* In this type of rotation, LL rotation followed by the RR rotation are performed.

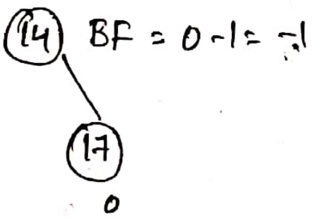


Apply LL Rotation then apply RR Rotation.

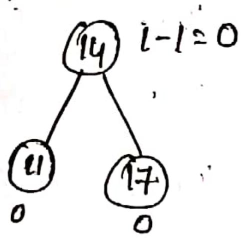
Example: 14, 17, 11, 7, 53, 4, 13, 12.

Step 1:

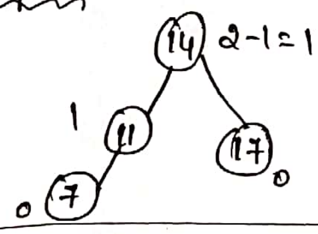
Step 2:



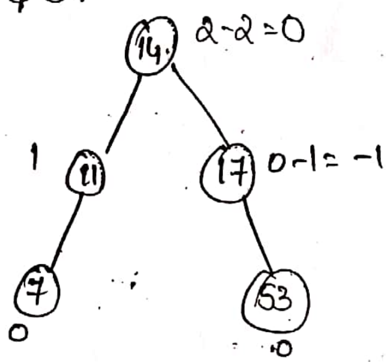
Step 3: insert 11.



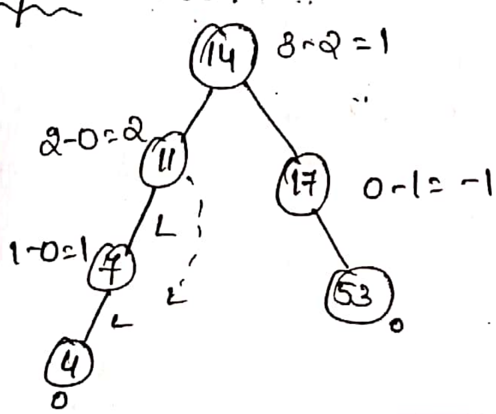
Step 4: insert 7.



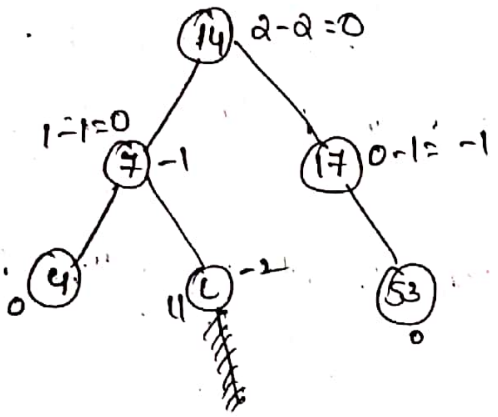
Step 5: insert 53.



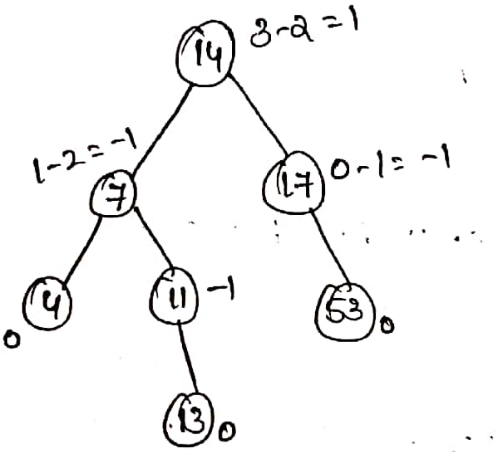
Step 6: insert 4.



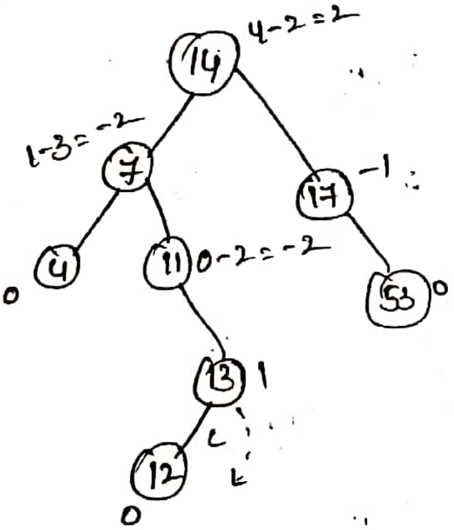
Step 7: Apply LL Rotation.



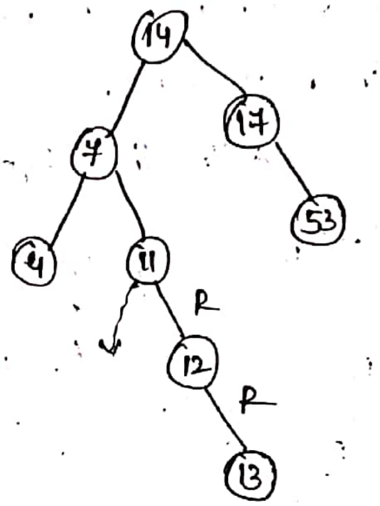
Step 8: insert 13.



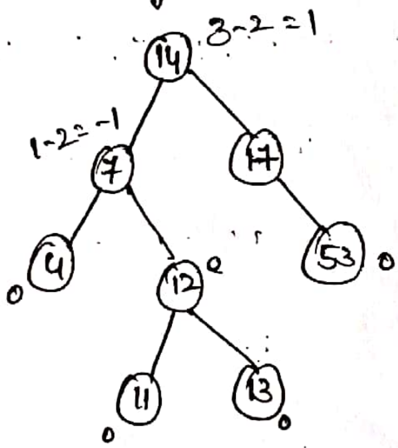
Step 9: insert 12.



⇒ Apply RL Rotation 1st apply LL Rotation.



Now, Tree imbalanced so RR rotation apply.



	Avg case	worst case
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Red - Black Tree - Rules:

(i) It should follow BST. (ii) Every node has a color either Red (or) Black.

2. Check whether tree is Empty

3. If tree is Empty then insert the New node as Root node with color Black & exit from the operation.

4. If tree is not Empty then insert the newnode as leaf node with color Red.

5. If the parent of newNode is Red then check the color of parent node's sibling of new Node.

6. If it is colored Black or Null then make suitable Rotation & Recolor it.

7. If it is colored Red, then perform Recolor. Repeat the same until tree becomes Red Black Tree.

8. Root is always Black.

9. Every path from Root to a NULL node has same number of black nodes.

operations:

Red - Black Trees:

* Red-Black tree is a binary Search tree in which every node is colored with either Red or Black.

* It is a type of self-balancing Binary Search tree.

* It has a good, efficient, worst case running time complexity.

* The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties:

1. The root and the external nodes are always black nodes.
2. [Red condition] No two red nodes can occur consecutively on the path from the root node to an external node.
3. [Black condition] The number of black nodes on the path from the root node to an external node must be the same for all external nodes (i.e., Black).

* Insertion:-

* Every new node which is to be inserted is marked red.

* Not every insertion causes imbalancing but if imbalancing occurs then that can be removed depending upon the configuration of tree before new insertion node.

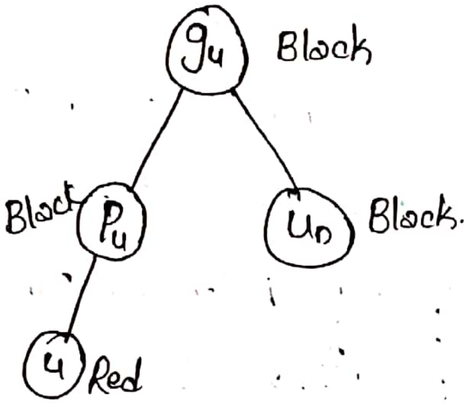
* The configuration of tree by defining following roles.

→ Let 'u' is newly inserted node.

'P_u' is the parent node of 'u'

'g_u' is the grand parent of 'u' and parent node of 'P_u'

'U_n' is the uncle node of 'u' i.e., its a right child 'g_{ru}'.



* The tree is said to be imbalanced if properties of Red - Black tree are violated.

* When insertion occurs, the new node is inserted in already balanced tree. If this insertion causes any imbalancing then balancing of the tree is to be done at '2' levels.

1. at grand parent level i.e., gu.
2. at parent level i.e., Pu.

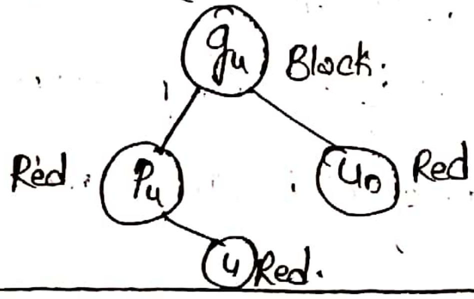
* The imbalancing is concerned with the root of grand parent's child i.e., uncle node.

* If uncle node is red then there are 4 cases.

1. LRr
2. LLr
3. RRr
4. RLr

LRr imbalance:

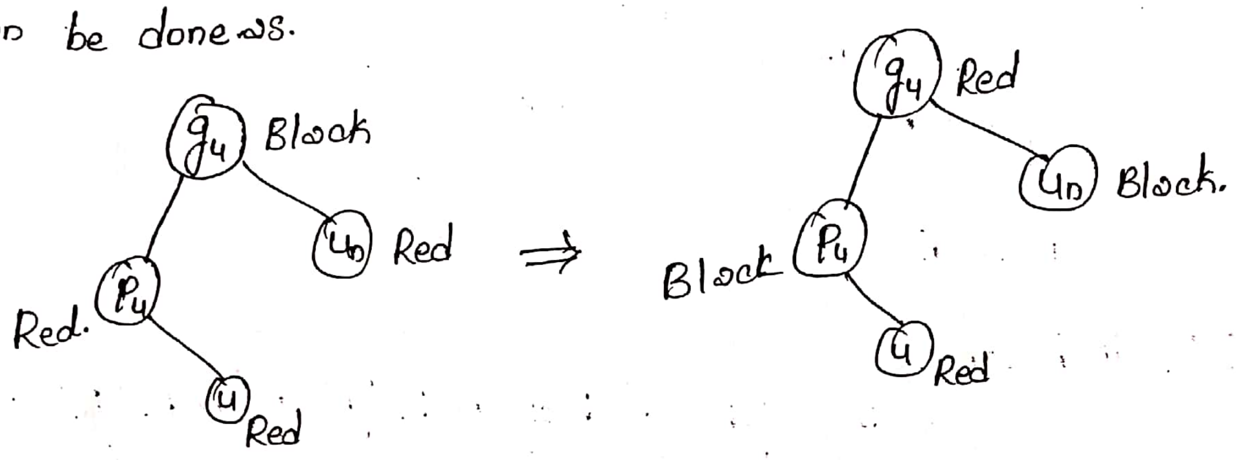
* The left child of 'gu' is 'Pu' and 'u' is right child of 'Pu' and 'Un' node (uncle node) is red.



Removal of LRr imbalancing:-

* Before color change, note that if 'g_u' in given figure is ~~most~~ root then there should not be any color change of 'g_u' (Because root is always Black).

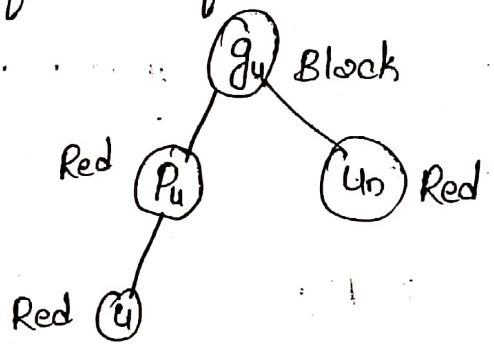
* But if 'g_u' happens to be red then the rebalancing can be done as.



1. Change color of 'P_u' from Red to Black.
2. Change color of 'U_n' from Red to Black.
3. Change color of 'g_u' from Black to red, when 'g_u' is not a root node.

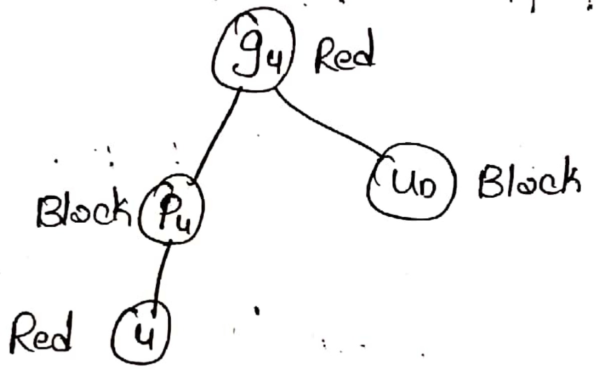
LLr imbalancing:-

* The node 'P_u' is a left child of 'g_u' and 'u' is inserted as left child of 'P_u'. Node 'U_n' is red.



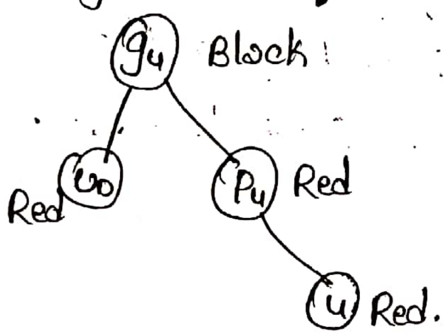
Removal of LL_r imbalancing:-

1. Change color of 'P_u' from red to Black.
2. Change color of 'U_n' from Red to Black.
3. Change color of 'g_u' from Black to Red when 'g_u' is not a root node.



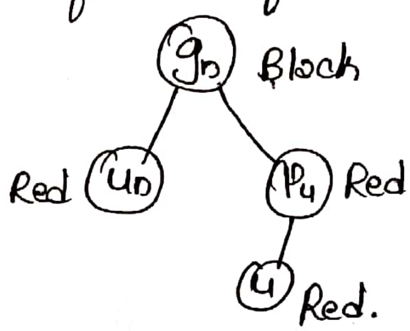
RR_r imbalance:-

* The right child of node 'g_u' is node 'P_u' and 'u' is inserted as right child of 'P_u'. The 'U_n' node is red.

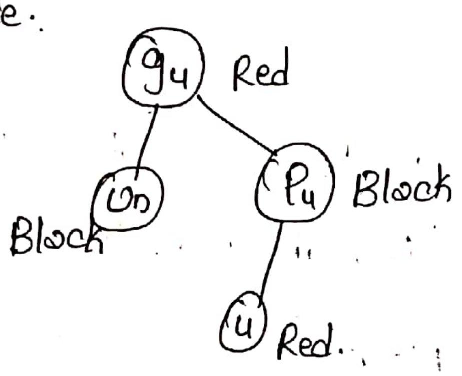


Removal of RR_r imbalancing:-

* The node 'P_u' is right child of 'g_u' and 'u' is inserted as a left child of 'P_u'. The uncle node U_n is Red.



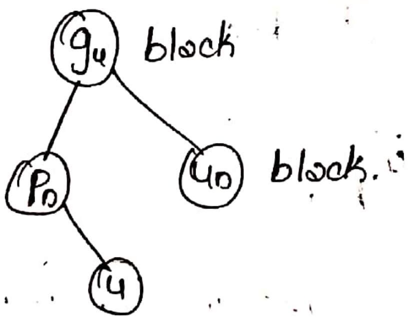
1. Change color of 'P_u' from Red to Black.
2. Change color of 'U_n' from Red to Black.
3. Change color of 'g_u' from Black to Red, when 'g_u' is not a Root node.



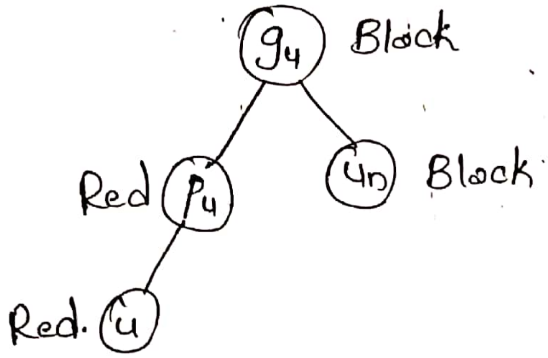
[NOTE:- To Remove these imbalancing Rotations are not required. Simply by changing the colors required Balancing can be obtained.]

* Now when other child of 'g_u' i.e., uncle node 'U_n' is black then there arise four cases.

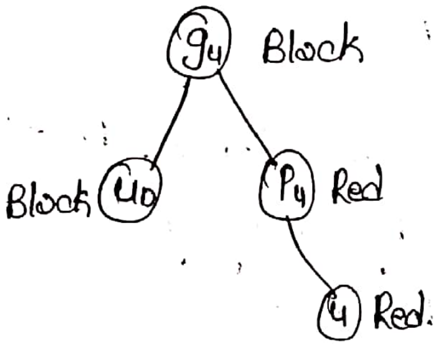
1. LR_b imbalancing:-
   * The 'P_u' node is attached as a left child of 'g_u' and 'u' is inserted as a right child of 'P_u'. The node 'U_n' is black.



2. LL_b imbalancing:-
   * The 'P_u' node is attached as a left child of 'g_u' and 'u' node is a left child of 'P_u'. The node 'U_n' is block.

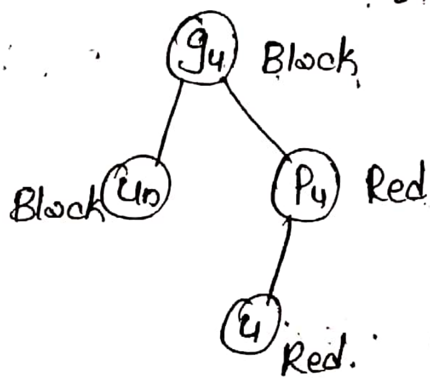


→ RR_b imbalancing:- The node 'Pu' is a right child of 'gu' and node 'u' is a right child of 'Pu'. The node 'Un' is block.



RL_b imbalancing:-

* The node 'Pu' is a right child of node 'gu' and node 'u' is a right child of 'Pu'. The node 'un' is block.

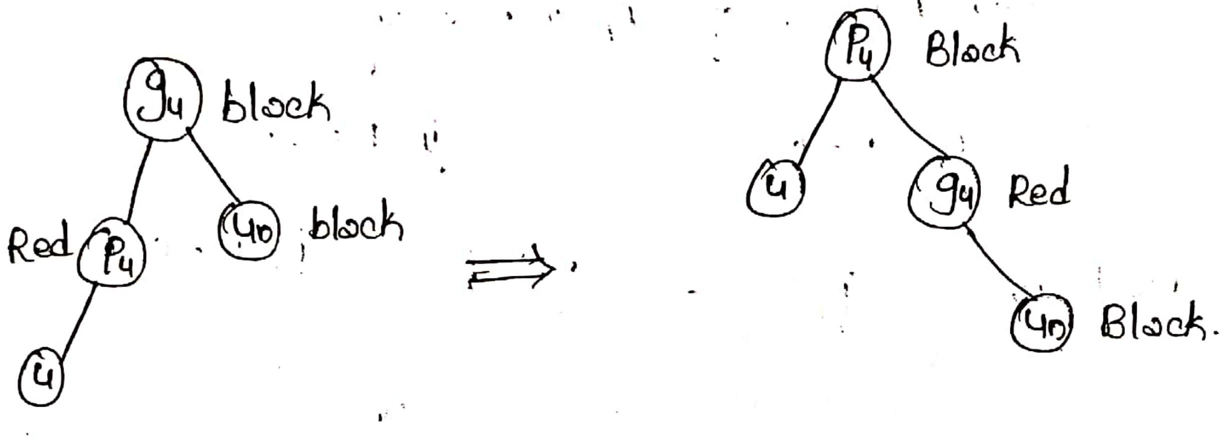


* As 'u' node gets inserted rebalancing must be performed.

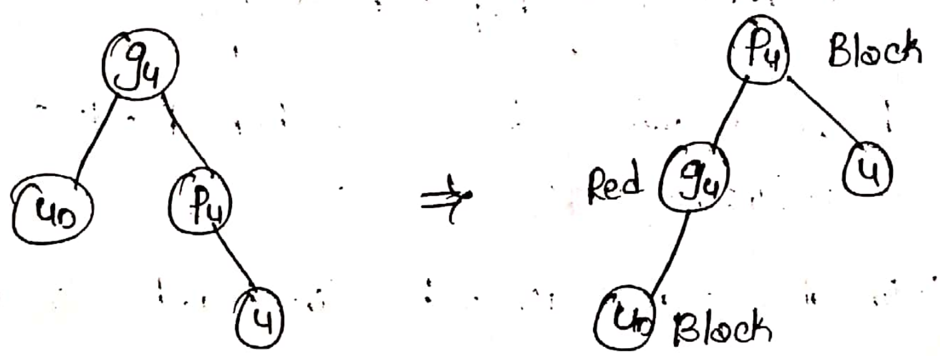
* LL_b and RR_b cases require Single Rotation followed by Recoloring.
* LR_b and RL_b cases require Double Rotation followed by Recoloring.

*** Removing LL_b and RR_b imbalances:-** [Parent node becomes root node]

1. Apply Single Rotation of "P_u" and "g_u".
2. Recolor 'P_u' to Black and 'g_u' to Red.



* Removal of LL_b imbalancing.



* Removal of RR_b imbalancing.

Removing LR_b and RL_b imbalances:-

* ~~Applying~~ Apply double Rotation of 'u' about 'P_u' followed by 'u' about 'g_u'.

2. For LR_b Recolor 'u' to black and Recolor 'Pu' and 'gu' to Red.

3. For RL_b Recolor 'Pu' to black.

fig: Removal of LR_b imbalancing.

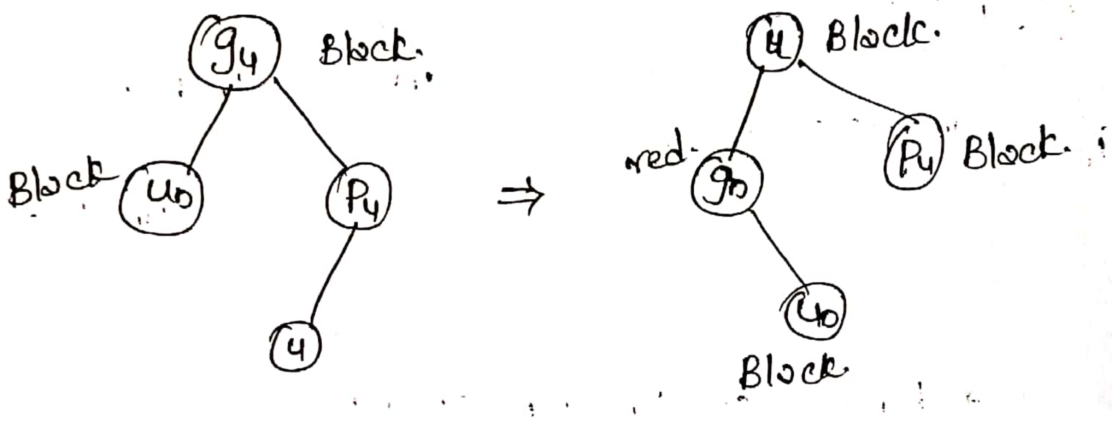


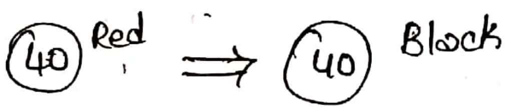
fig: Removal of RL_b imbalancing.

Eg: Insert below values and form a Red-Black tree.

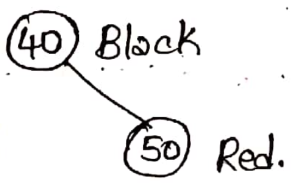
40, 50, 70, 30, 42, 15, 20, 25.

Sol: Initially insert node 40 with color Red. Recolor with node to black.

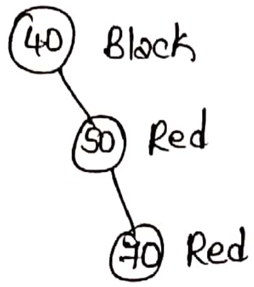
Step 1:



Step 2: Insert 50.

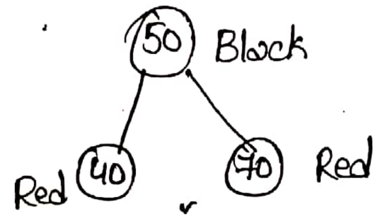


Step 3: Insert 70.

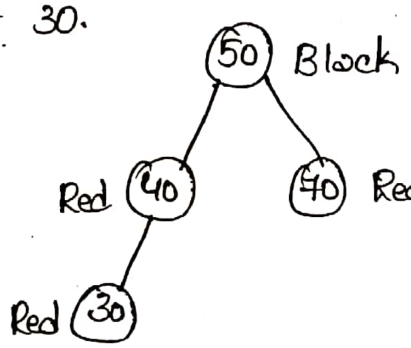


(Empty (or) black - Uncle node)

Recolor and Rotate it (RR_b imbalancing)

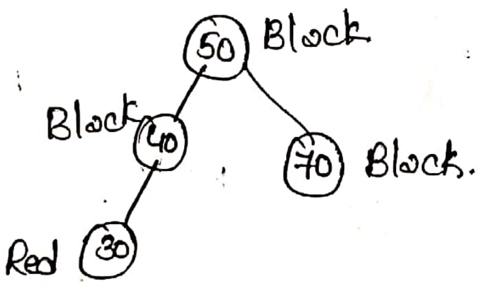


Step 4: Insert 30.

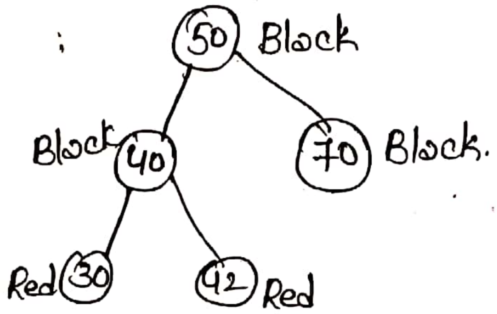


(Recolor Pu, gu, & Un)

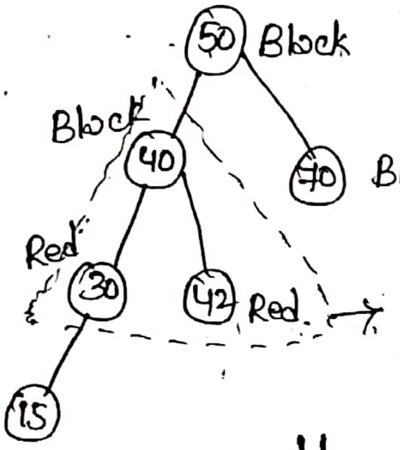
Apply LL_r imbalancing.



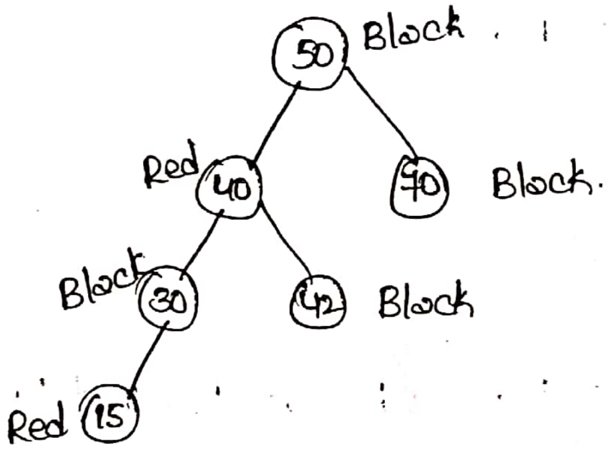
Step 5: Insert 42.



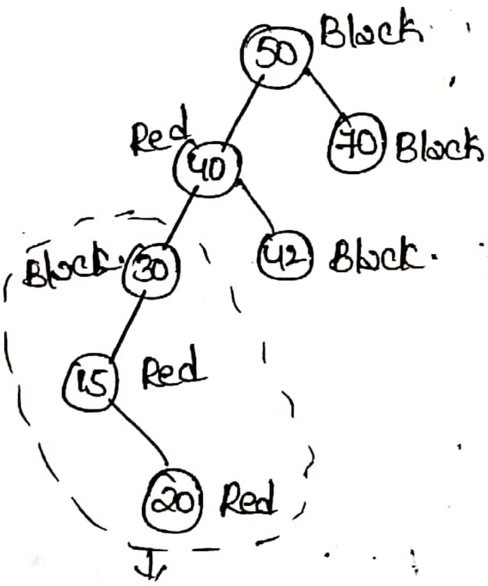
Step 6: Insert 15

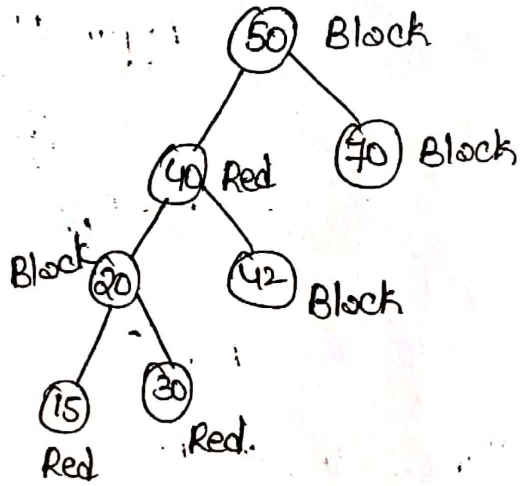


[Recolor marked nodes
LLr imbaloncing]



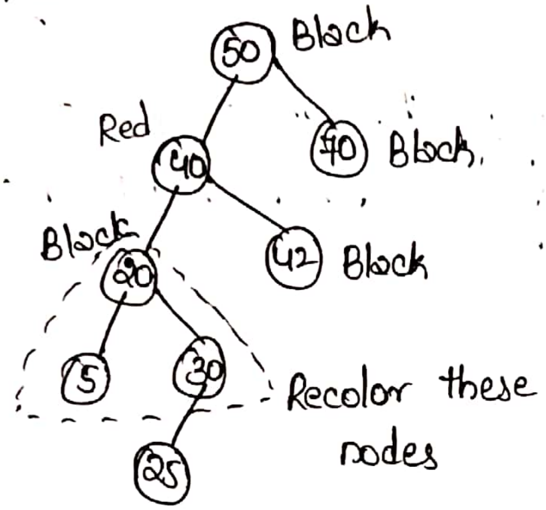
Step 7: Insert 20.





LR_b imbaloncing
hence Rotate.

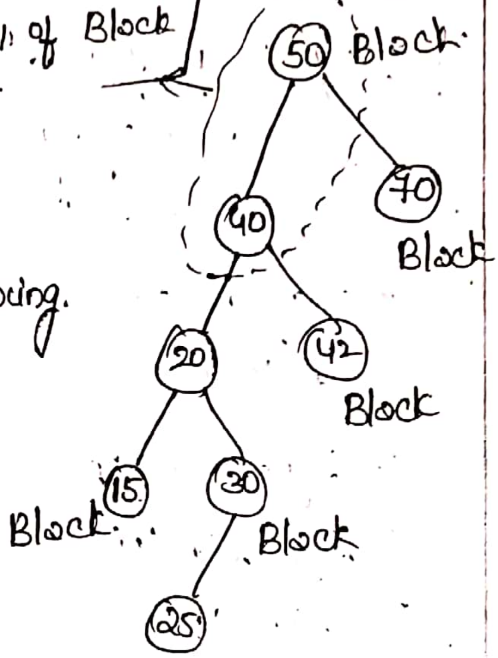
Step 8: Insert 25.



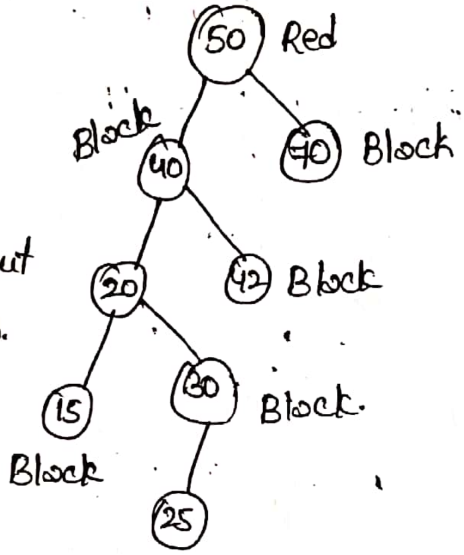
[Path doesnot have same no. of Black nodes.]

$\Rightarrow$ RL_r imbalancing.

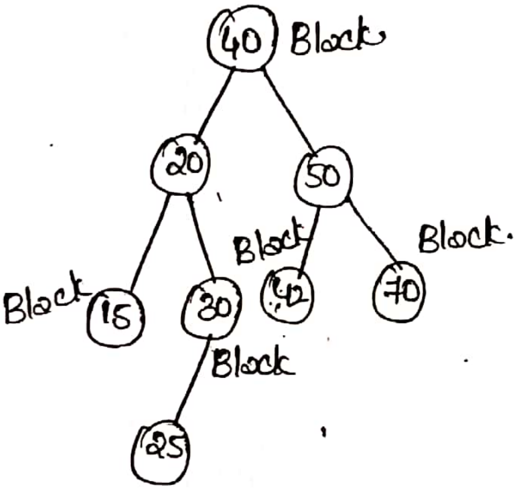
Recolor these nodes.



$\Downarrow$



Rotating about 40. $\Leftarrow$



Splay Trees:-

* A splay Tree is a self balancing binary search tree with no explicit balance condition.

* The splay tree has a property that recently accessed elements can be accessed Quickly.

* All normal operations that are performed on Binary Search tree are performed on splay tree.

* But there is a special ~~tree~~ operation called Splaying is performed on splay tree.

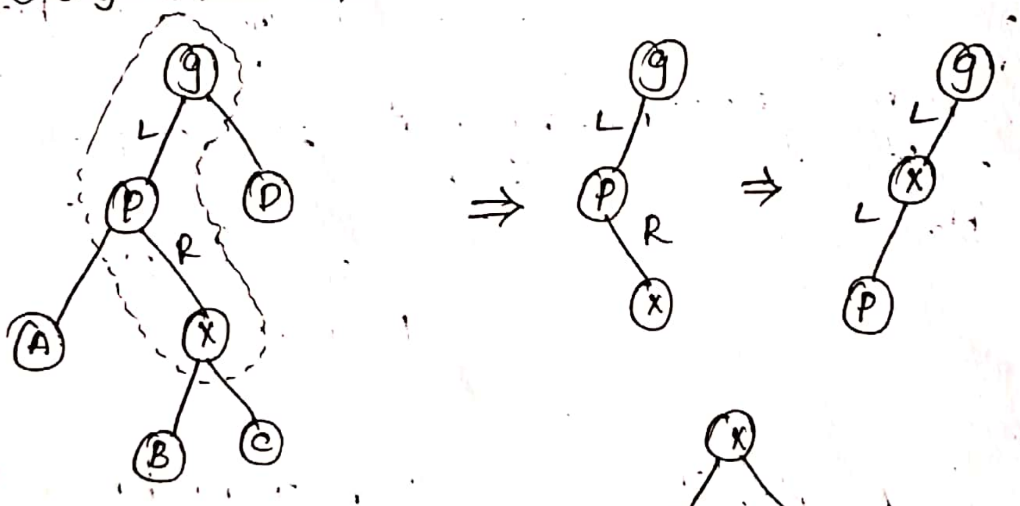
* Splaying means arranging the elements of a tree in such a way that the most recently accessed node will be placed as root of the tree.

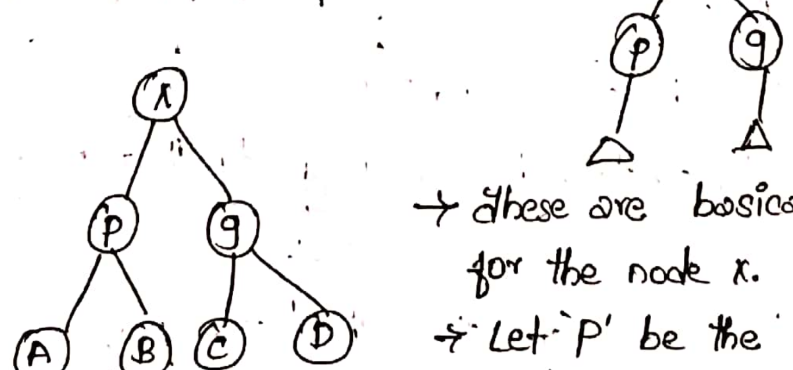
* Various cases that arise are

1. Zig-Zag (LR)
2. Zig-Zig (LL)
3. Zig (L)

* Zig means "Left" and Zag means "Right".

1. Zig-Zag case (LR):-





→ These are basically Rotations applied for the node x.

→ Let 'P' be the parent node of 'x' and 'g' be the grand parent of 'x'.

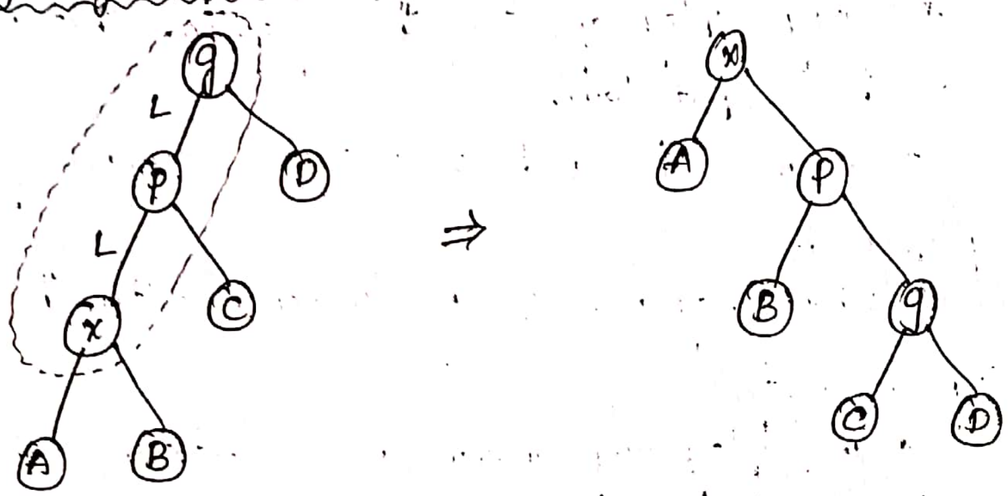
* When 'x' is a right child of node 'P' and 'P' is a left child of node 'g'.

* When the rearrangement is made for most recent accessed node 'x'.

* The 'x' becomes root, 'p' becomes its left child & 'g' becomes the right child.

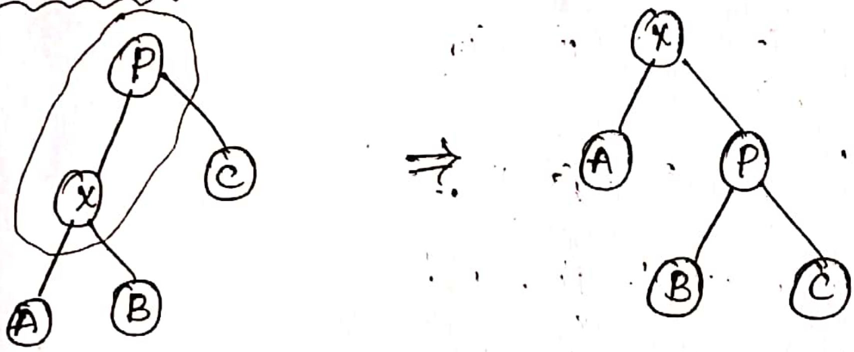
* The Zig-Zag step is equivalent to rotation above.

* Zig-Zig ~~Color~~: case (LL):-



* The 'p' node becomes right child of 'x' and 'g' becomes right child of 'P', where elements get relocated.

* Zig case (L):-



Splay Trees:

→ ~~Binary~~ Self Balanced binary Search tree that operate on O(logN) complexity.

→ Splay means "Recently accessed node".

→ There is no explicit balance condition.

→ Splay nodes are placed as root node after every operations like insert, delete, Search.

→ Splay trees are not height balanced but it is a balanced one.

Rotation in Splay involves 3 nodes,

* recently accessed node - X
* parent node of X - P
* Grandparent node of X - G

Cases:-

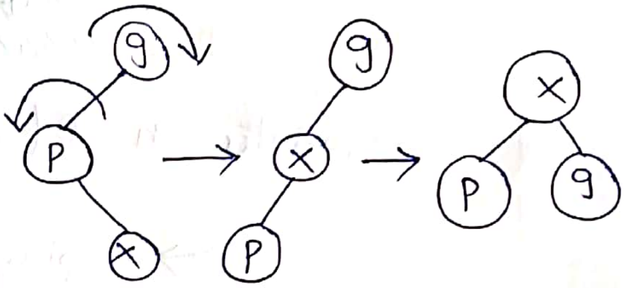
1. Zig Step
2. Zig- Zag Step
3. Zig- Zig Step
4. Zag- Zag Step
5. Zag- Zig Step
6. Zag Step

Zig means left

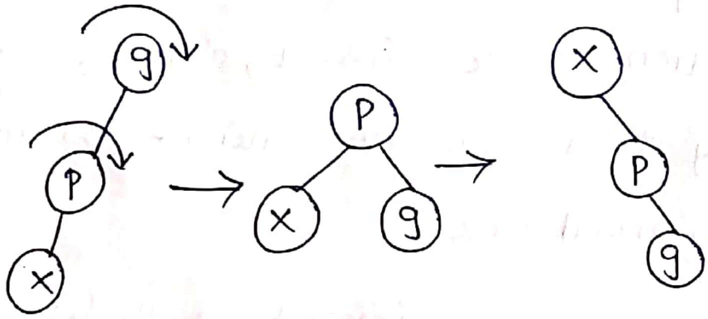
Zag means right.

1. Zig Step

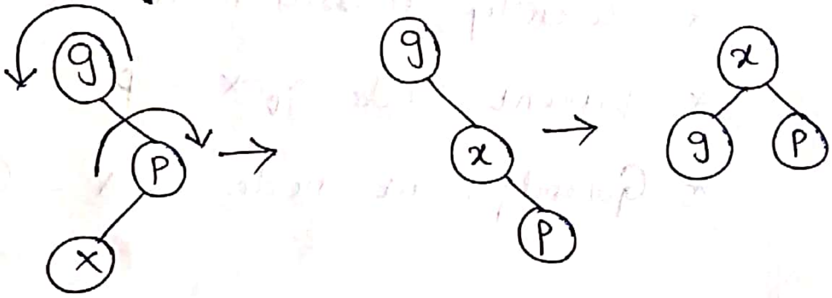
2. Zig-Zag Step



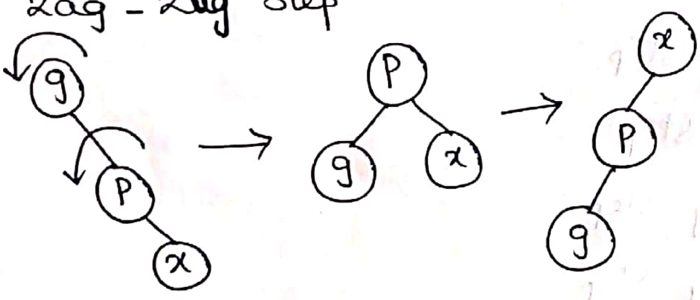
3. Zig-Zig Step



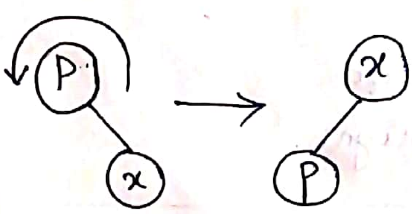
4. Zag-Zig Step



5. Zag-Zag Step

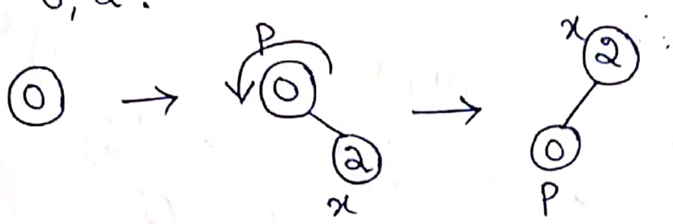


6. Zag Step

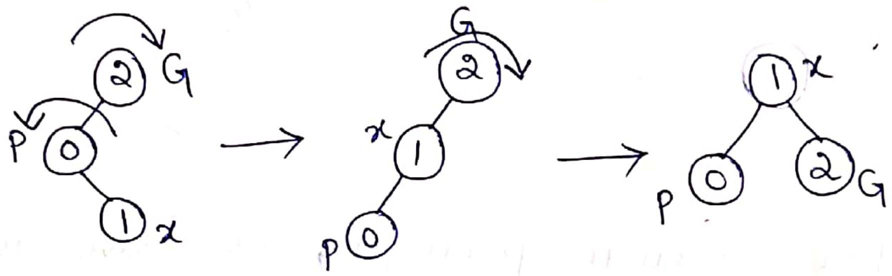


Insertion:

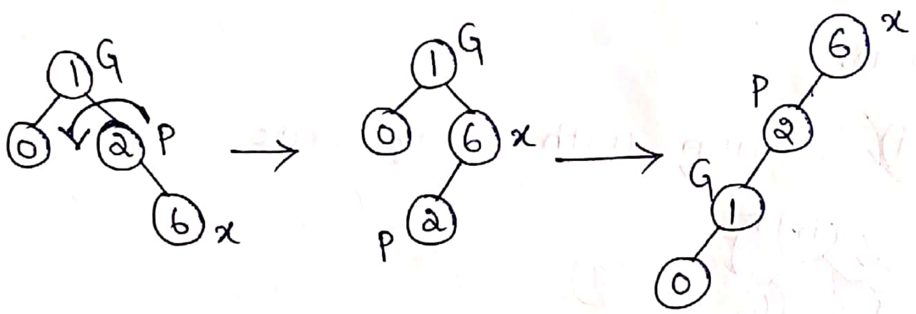
Insert 0, 2 :



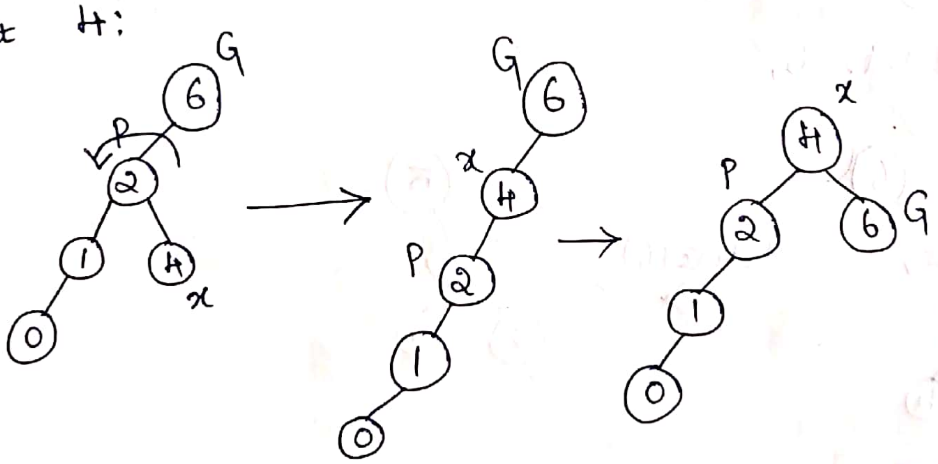
Insert 1 :



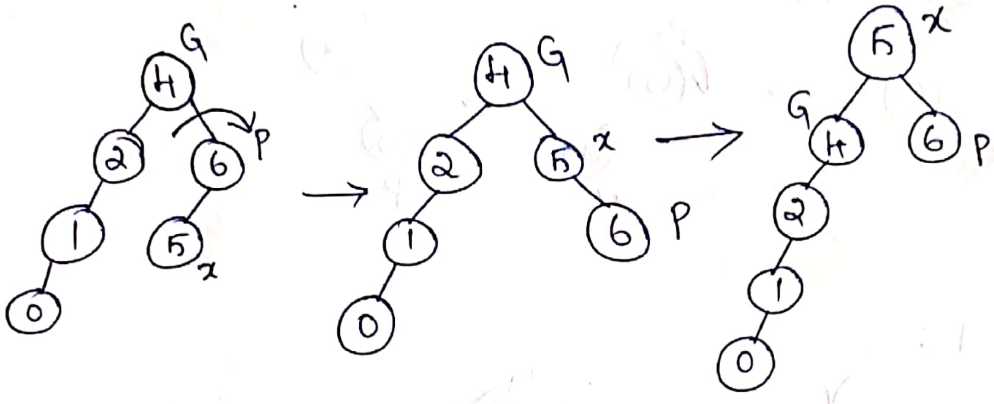
Insert 6 :



Insert 4 :



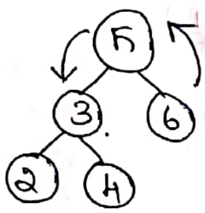
Insert 5 :



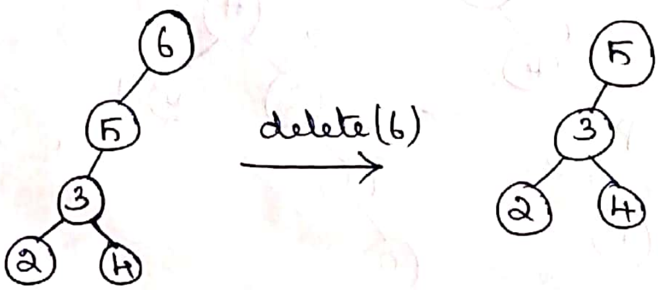
Deletion :

For deletion operation, we have to splay the node that is to be deleted and then perform deletion.

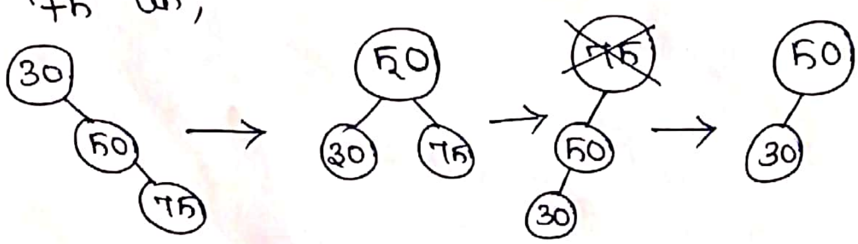
Eg:- i) Deletion with leaf node.



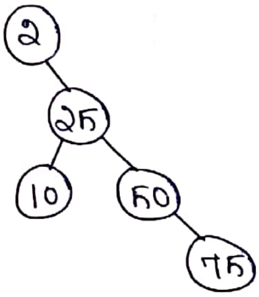
Now delete 6,



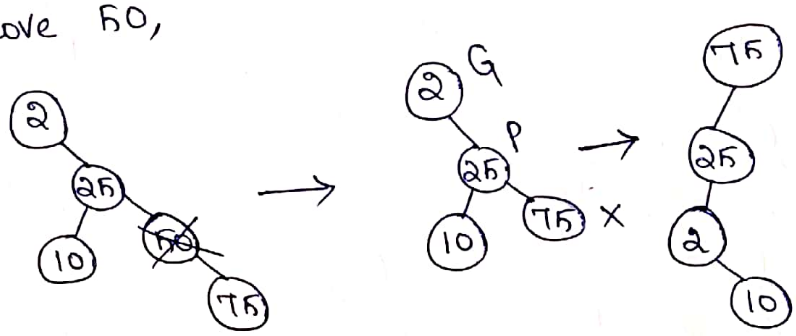
Remove 75 in,



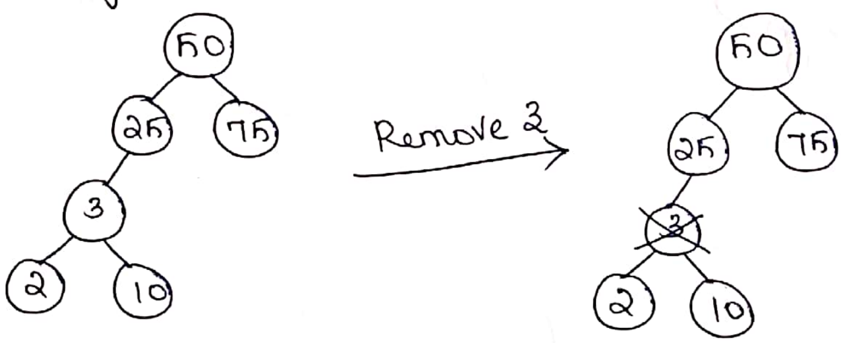
ii) Deleting a node with one child.



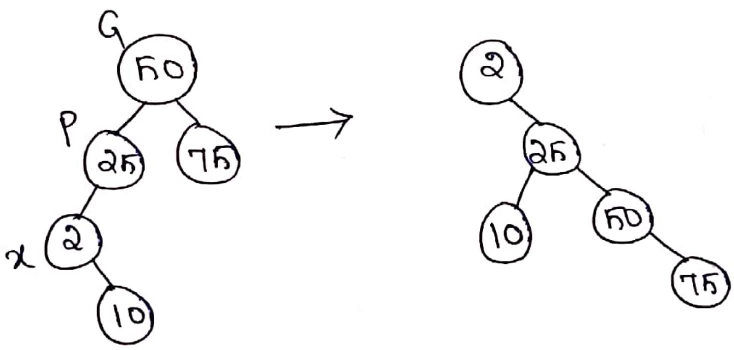
Now remove 50,



iii) Deleting a node with two children.

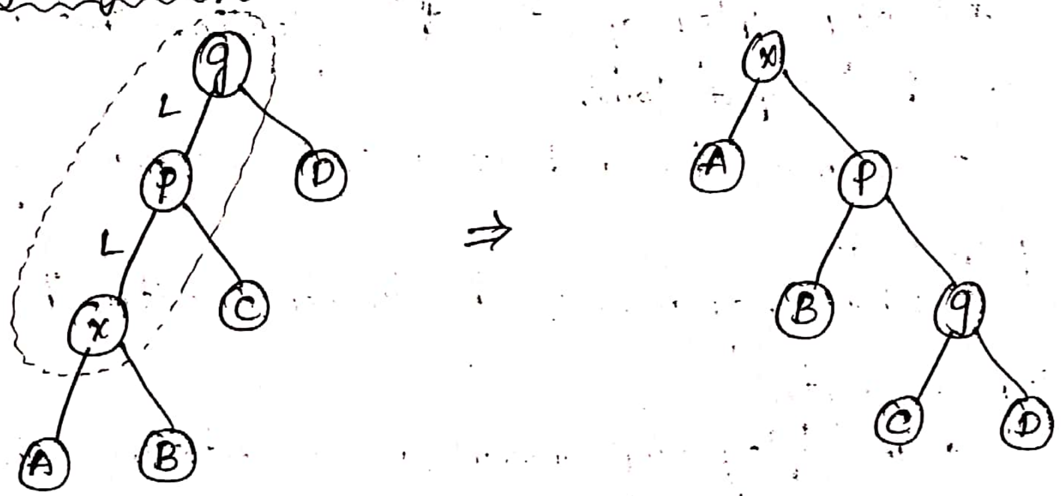


Inorder ~~Successor~~ predecessor → 2



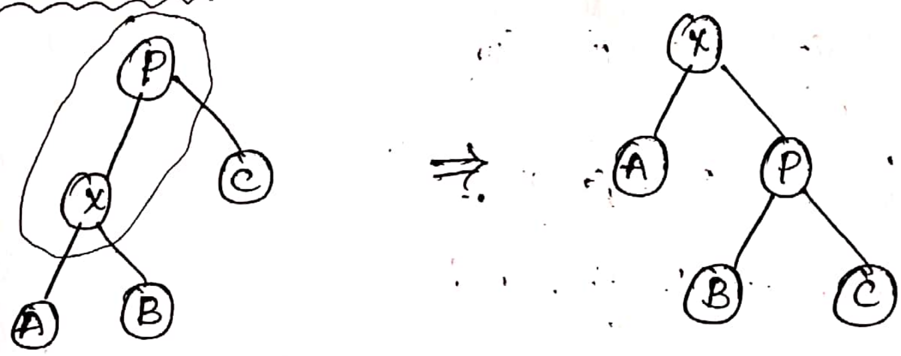
* When 'x' is a right child of node 'p' and 'p' is a left child of node 'g'.
* When the rearrangement is made for most recent accessed node 'x'.
* The 'x' becomes root, 'p' becomes its left child & 'g' becomes the right child.
* The Zig-Zag step is equivalent to rotation above.

* Zig-Zig ~~Color~~: case (LL):-



* The 'p' node becomes right child of 'x' and 'g' becomes right child of 'p', where elements get relocated.

* Zig case (L):-



Splay Operations:-

* Various operations supported by splay trees are.

1. Splay
2. Find
3. Insert
4. Delete

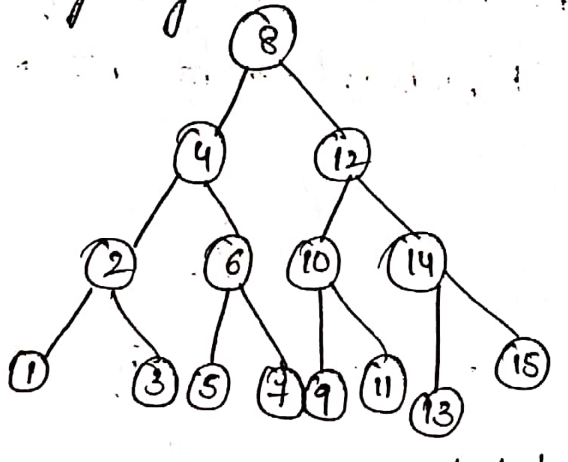
* The splay operation starts at splay node. The splay node is basically root of the Binary Search tree.

* During splay operations the splay node that is been selected is at the deepest level.

* To move this node at root the sequence of splay steps are performed.

* When splay node reaches to root position the sequence of splay step is empty.

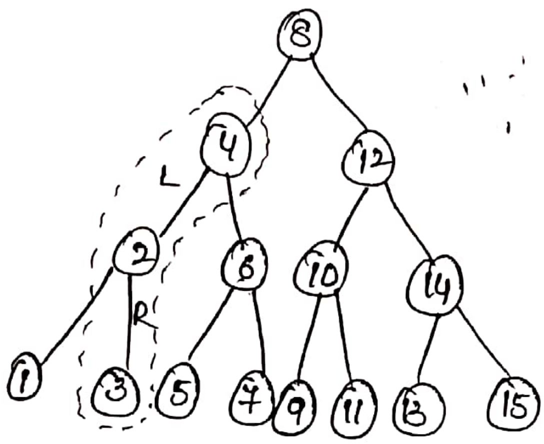
* The splay step involves various rotations such as Zig-Zag, Zig-Zig, Zig

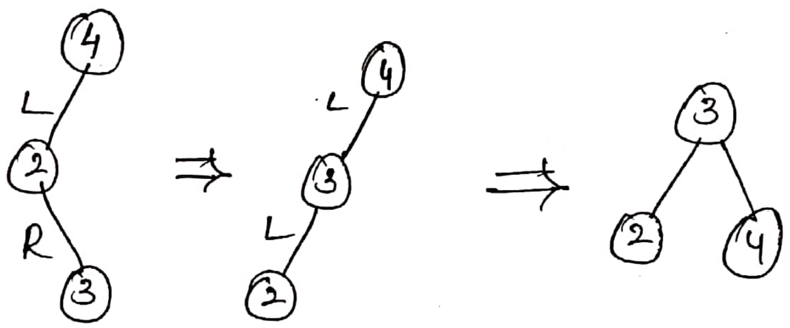


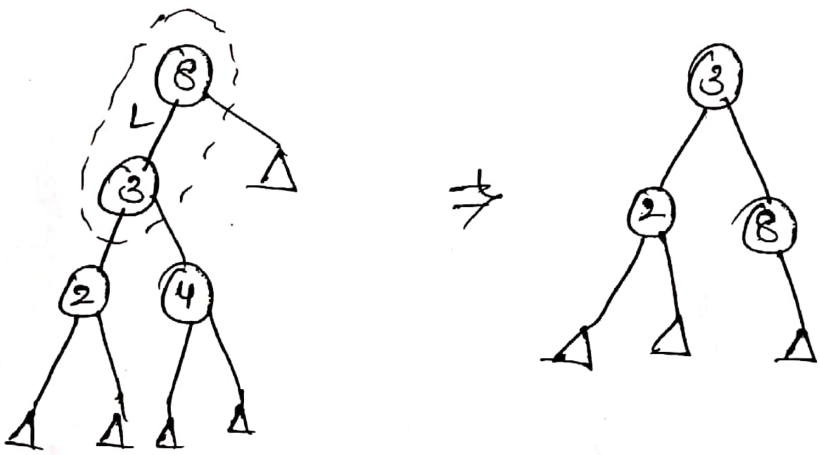
Sol: Now, if we want to find node '3'.

* Then focus of flow operation consists of node being splayed its parents and its grand parent.

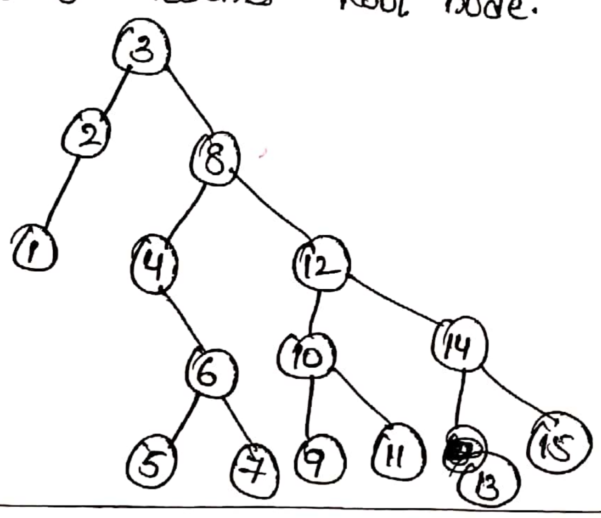
* The node involved in splay operation indicate Zig-Zag (or) LR Rotation.







Finally the node '3' reaches Root node.



Avg case : $O(\log n)$

worst case : $O(n)$

Any operation : $O(n \log n)$