UNIT-III

SEARCH TREES

Objective:

To explore Search Trees

Syllabus:

Search Trees (Part1): AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Splay Trees

STUDY MATERIAL

Introduction:

Tree:

A Tree consists of a finite set of elements, called nodes, and set of directed lines called branches, that connect the nodes.

The no. of branches associated with a node is the degree of the node.

i.e. in - degree and out - degree.

- A **leaf** is any node with an out degree of zero. i.e. no successor
- A node that is not a root or leaf is known as an **internal node**
- A node is a **parent** if it has successor node, conversely a node with a predecessor is called **child**
- A **path** is a sequence of nodes in which each node is adjacent to the next one
- The **level** of a node is its distance from the root
- The **height** of the tree is the level of the leaf in the longest path from the root plus 1
- A **sub tree** is any connected structure below the root. Sub tree can also be further divided into sub trees

Binary tree:

A **binary tree** is a tree in which no node can have more than two sub trees designated as the **left sub tree** and the **right sub tree.**

Note: each sub tree is itself a binary tree.

Balance factor:

The balance factor of a binary tree is the difference in height between its left and right sub trees.
i.e. **Balance factor = H_L - H_R**

In a balanced binary tree, the height of its sub trees differs by no more than one (its balance factor is -1, 0, +1) and also its sub trees are also balanced.
We now turn our attention to operations: **search, insertion, deletion**

In the design of the linear list structure, we had two choices: an array or a linked list

- The array structure provides a very efficient search algorithm, but its insertion and deletion algorithm are very inefficient.
- The linked list structure provides efficient insertion and deletion, but its search algorithm is very inefficient.

What we need is a structure that provides an efficient search, at the same time efficient insertion and deletion algorithms.
The **binary search tree** and the **AVL tree** provide that structure.

Binary search tree:

A binary search tree (BST) is a binary tree with the following properties:

- All items in the left sub tree are less than the root.
- All items in the right sub tree are greater than or equal to the root.
- Each sub tree is itself a binary search tree.

While the binary search tree is simple and easy to understand, it has one major problem:
It is not balance.

To over come this problem, AVL trees are designed, which are balanced.

AVL TREES

In 1962, two Russian mathematicians, G.M Adelson – velskii and E.M Landis, aerated the balanced binary tree structure that is named after them – the AVL tree.

An AVL tree is a search tree in which the heights of the sub trees differ by no more than 1. It is thus a balanced binary tree.

An AVL tree is a binary tree that either is empty or consists of two AVL sub tree, T_L and T_R, whose heights differ by no more than 1.

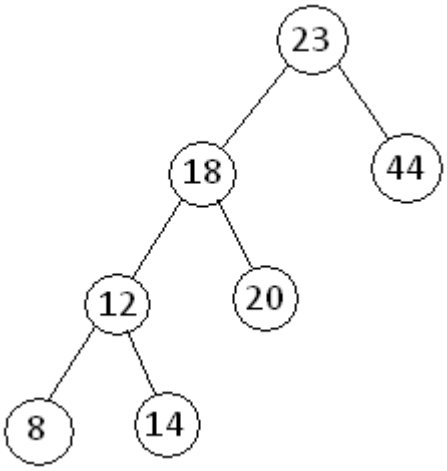
$$| H_L - H_R | <= 1$$

Where H_L is the height of the left sub tree, H_R is the height of the right sub tree

The bar symbols indicate absolute value.

NOTE: An AVL tree is a height balanced binary search tree.

Consider an example: AVL tree



AVL Tree Balance factor:

The Balance factor for any node in an AVL tree must be +1, 0, -1.

We use the descriptive identifiers

- **LH** for left high (+1) to indicate that the length sub tree is higher than the right sub tree
- **EH** for even high (0) to indicate that the sub tree are the same height
- **RH** for right high (-1) to indicate that the left sub tree is shortest than the right sub tree

Balancing Trees:

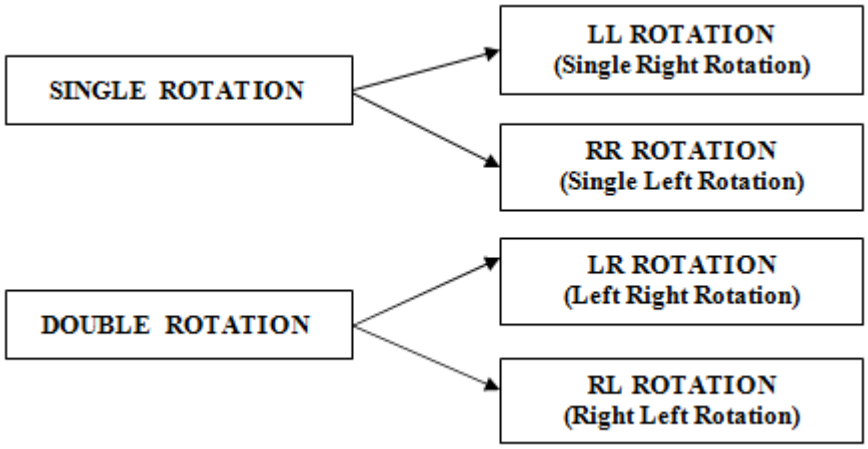
When ever we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced then we must rebalance it.

AVL trees are balanced by rotating nodes either to the left or to the right.

Rotations:

Rotations are mechanisms which shift some of the sub trees of the unbalanced tree either to left or right to obtain a balanced tree.

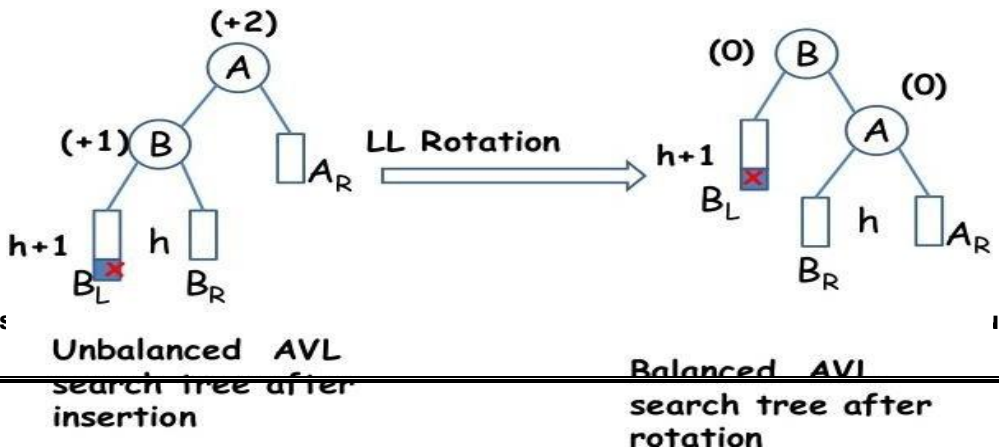
There are four rotations and they are classified into two types.



LL Rotation (Single Right Rotation)

- The new node ‘x’ is inserted in the Left sub tree of left child of node A. As a result, the balance of A becomes +2.
- To restore balance at A Single right rotation need to be applied at A.

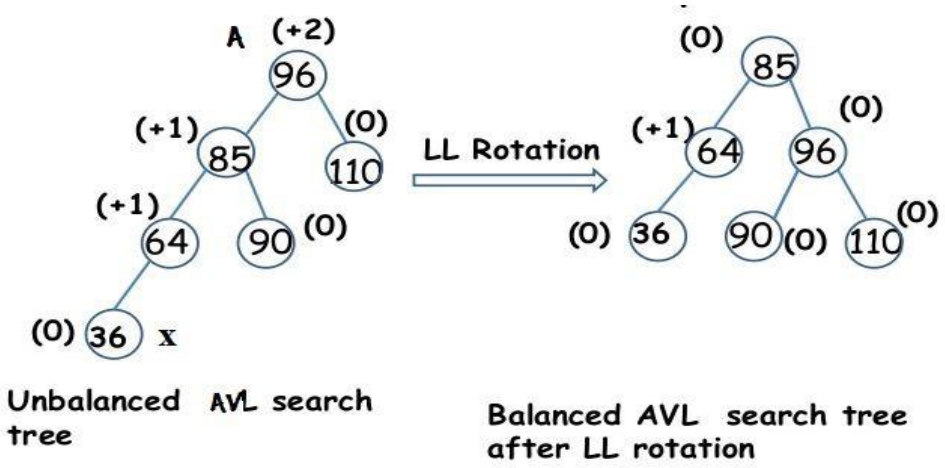
General Notation:



Unbalanced AVL search tree after insertion

Balanced AVL search tree after rotation

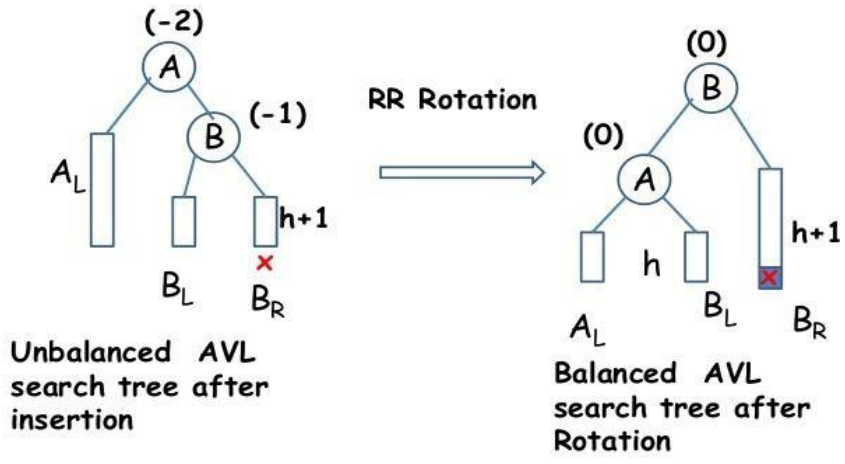
Example:



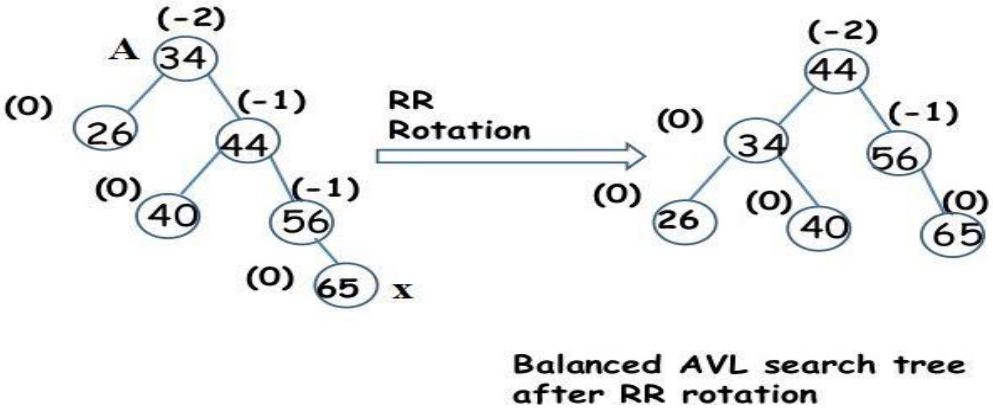
RR Rotation (Single Left Rotation)

- The new node 'x' is inserted in the Right sub tree of right child of node A. As a result, the balance of A becomes -2.
- To restore balance at A Single left rotation need to be applied at A.

General Notation:



Example:



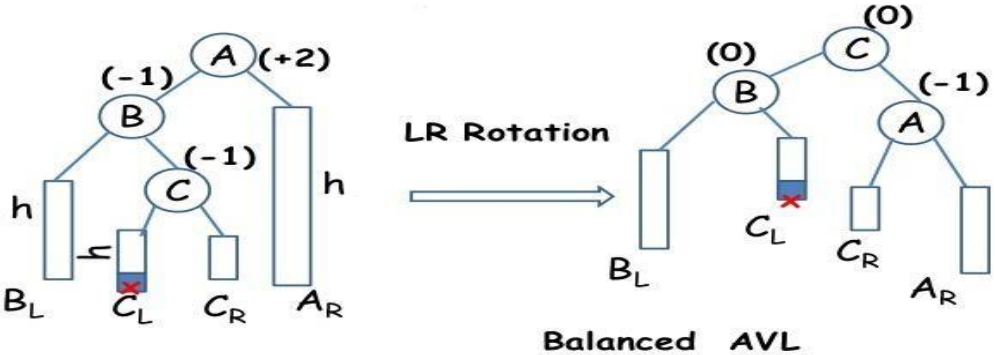
Balanced AVL search tree after RR rotation

LR Rotation (Left Right Rotation)

Imbalance occurred at A due to the insertion of node x in the right sub tree of left child of node A.

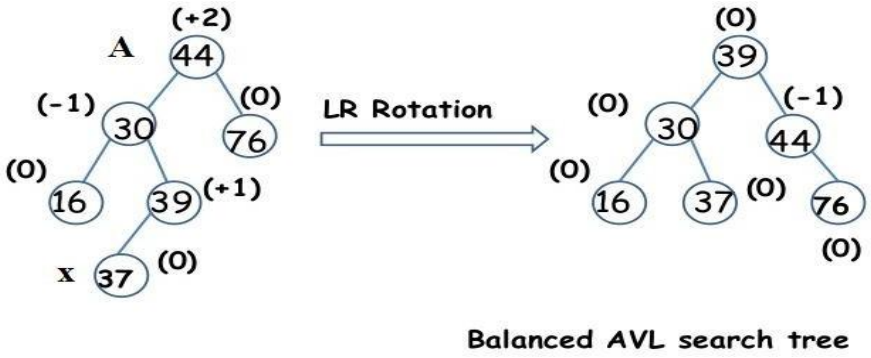
- LR rotation involves sequence of two rotations
    - Single Left rotation/RR rotation
    - Single Right rotation/LL rotation

General Notation:



Balanced AVL search tree after LR Rotation

Example:

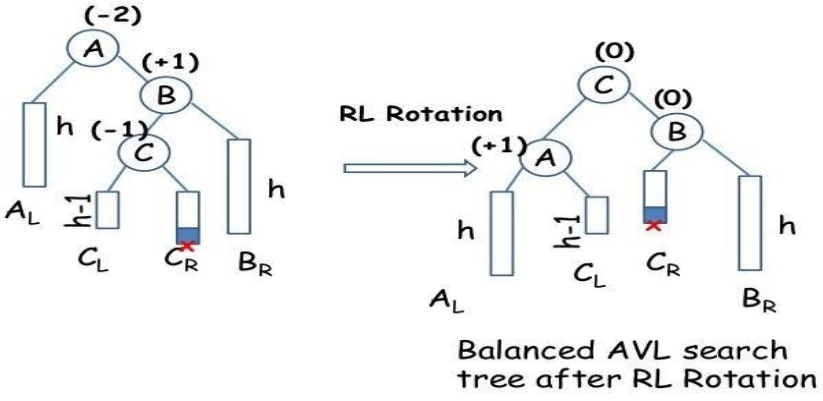


RL Rotation (Right Left Rotation)

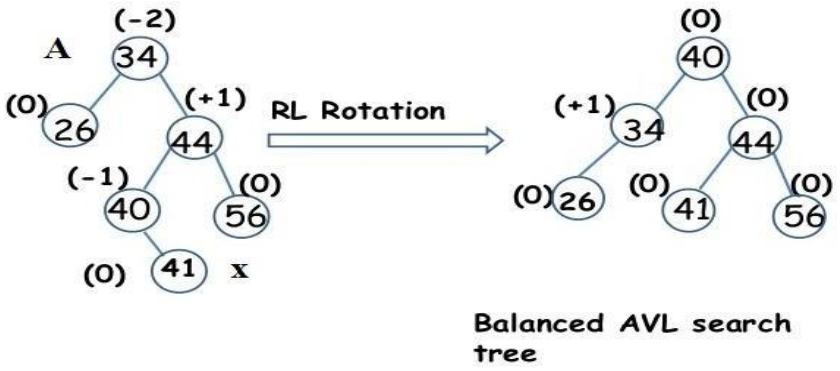
Imbalance occurred at A due to the insertion of node x in the Left sub tree of right child of node A.

- RL rotation involves sequence of two rotations
  - Single Right rotation/LL rotation
  - Single Left rotation/RR rotation

General Notation:



Example:



OPERATIONS ON AN AVL TREES

The basic operations performed on an AVL tree are

- Searching an element
- Inserting a new element
- Deleting an element

INSERTION OPERATION ON AVL TREES

In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows....

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

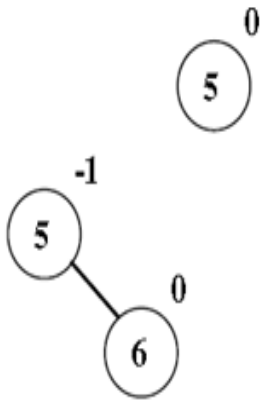
Step 4: If after insertion the balance factor of any of the nodes turns out to be anything other than 0, +1 or -1, then the tree is said to be unbalanced.

CONSTRUCTION OF AN AVL TREE

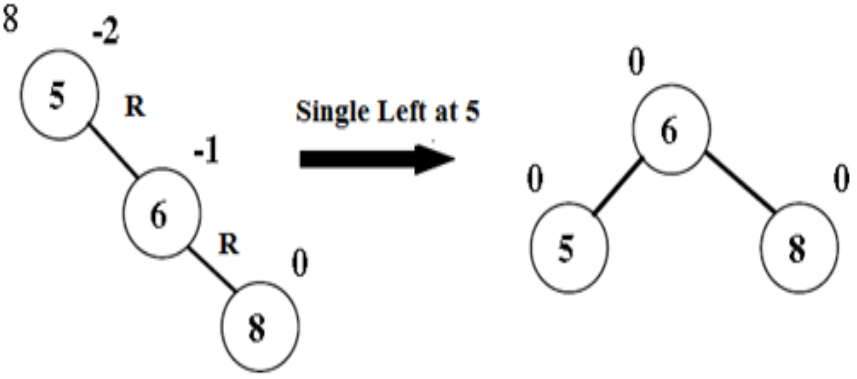
Construct AVL tree for the list by successive insertion: 5, 6, 8, 3, 2, 4, 7

i Insert 5 into empty tree

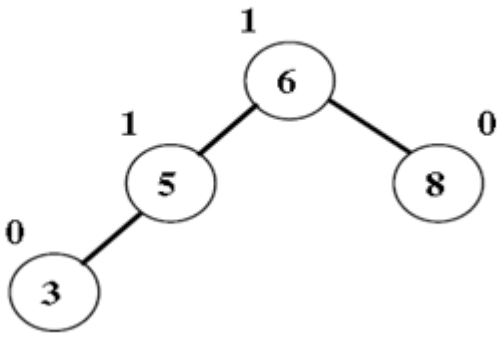
ii. Insert 6



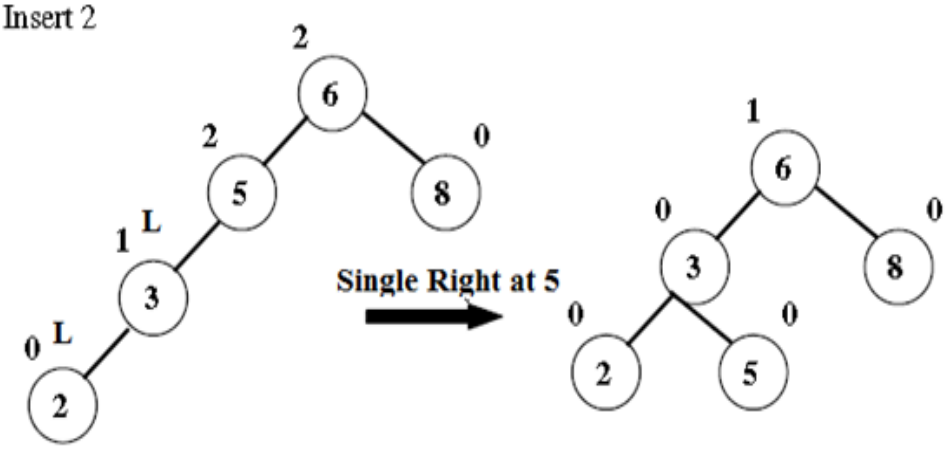
iii. Insert 8



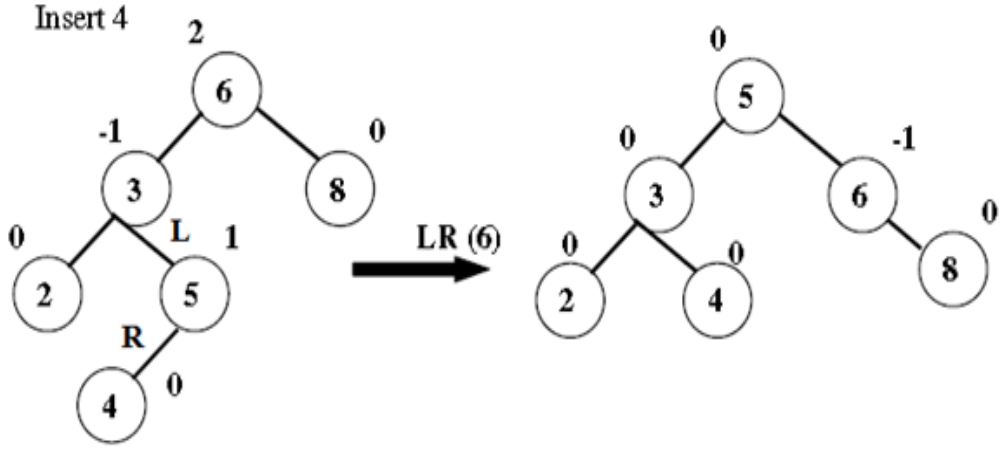
iv. Insert 3



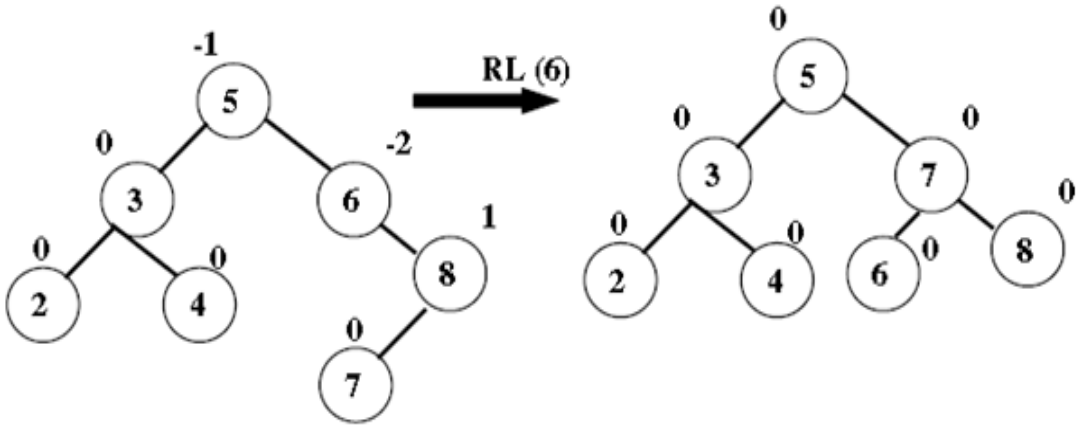
v. Insert 2



vi. Insert 4



vii. Insert 7



DELETION OPERATION ON AVL TREES

The sequence of steps to be followed in deletion are

1. Initially, the AVL tree is searched to find the node to be deleted
2. If the search is successful, delete the node. The following are the 3 possibilities for the node 'x' that is to be deleted
    - x is a leaf – In this case the leaf node is discarded
    - x has exactly one non empty sub tree – If x has no parent, the root of its sub tree becomes the new search tree root. If x has a parent P, then we change the pointer from parent p so that it points to x's only child
    - x has two non empty sub trees – x is replaced with either the largest element in the left sub tree or the smallest element in its right sub tree.
3. After deletion of node, check the balance factor of each node
4. Rebalance the tree if the tree is unbalanced. For this AVL tree deletion rotations are used.

- On deletion of a node x from the AVL tree. Let A be the closest ancestor node on the path from x to the root node, with a balance factor of +2 or -2. To restores balance at node A, we classify the type of imbalance as follows:

L – type imbalance:

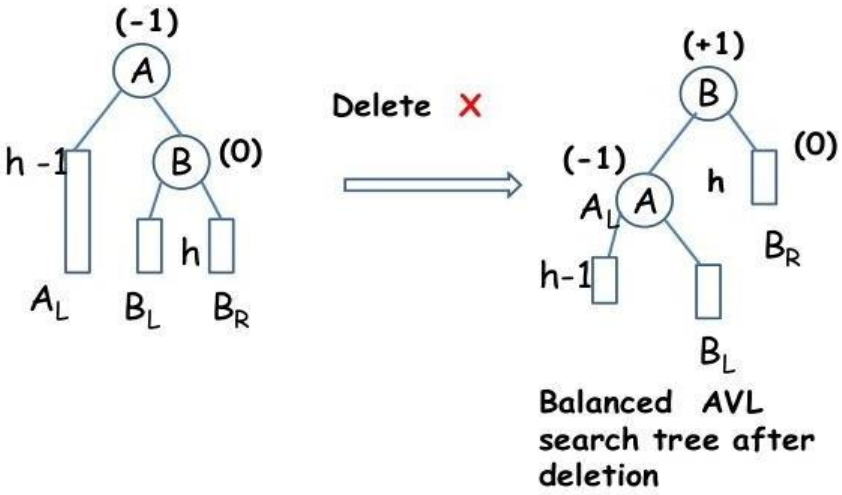
- The imbalance is of type L if the deletion took place from A's left sub tree.
- The balance factor of A = -2
- A has a right sub tree with root B
- L-type imbalance is sub classified into types L0, L1 and L-1 depending on the balance factor of B.

1. L0 ROTATION

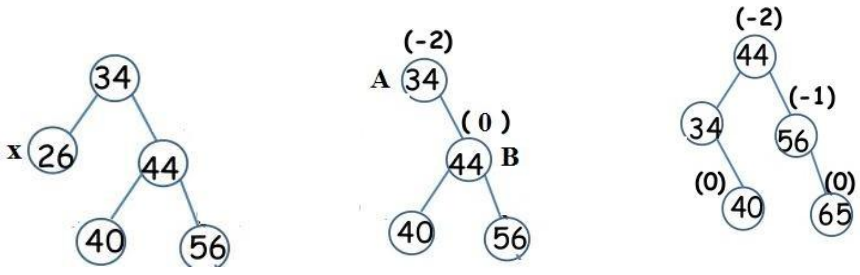
L0 imbalance occurs if the deletion takes place from the left sub tree of A and balance factor of B is 0

L0 rotation is Single Left rotation, that is applied at node A

General Notation:



Example

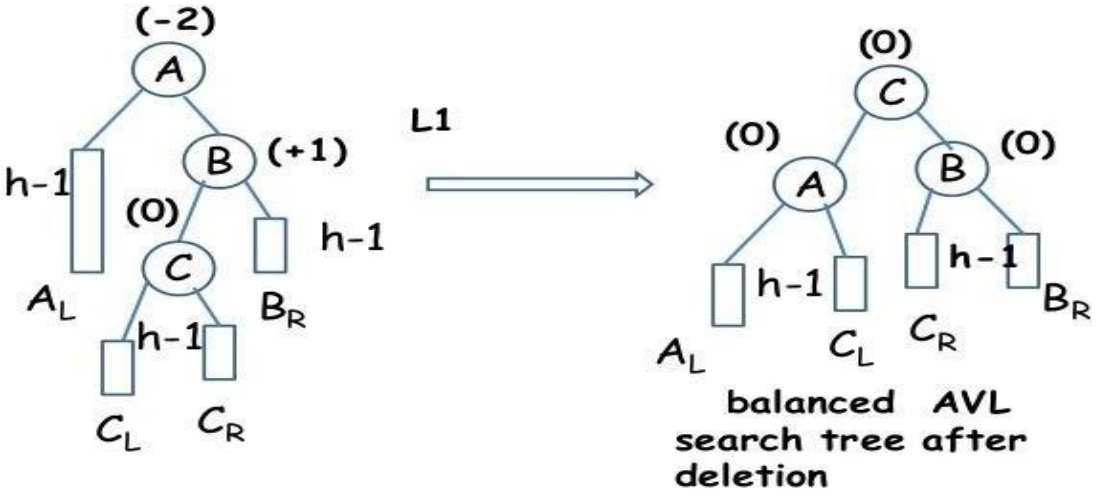


Before deletion After deleting 26 After L0 rotation

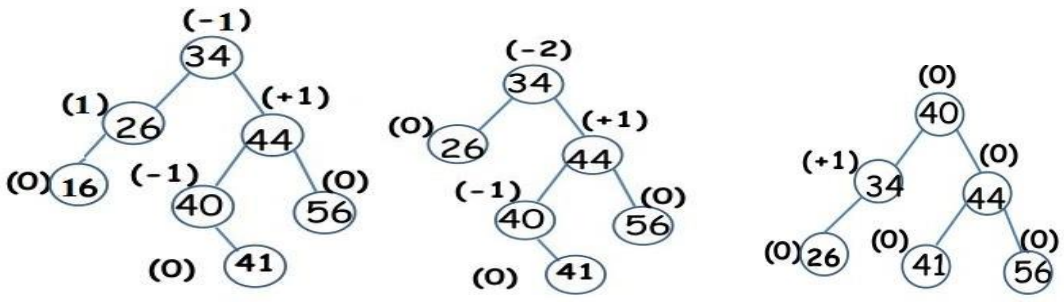
2. L1 ROTATION

- L1 imbalance occurs if the deletion takes place from the left sub tree of A and balance factor of B is 1
- L1 rotation is RL rotation, which involves 2 rotations:
  - Single Right
  - Single Left

General Notation:



Example:

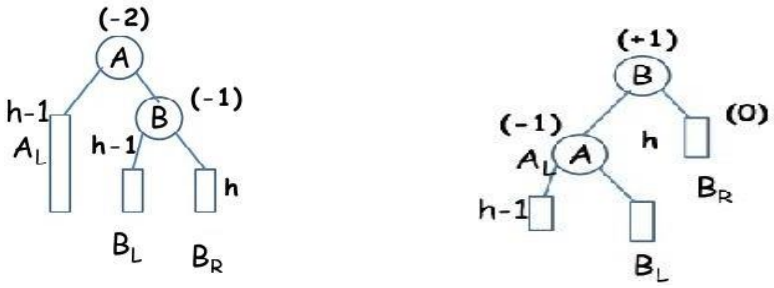


Before deletion | After deleting 16 | After L1 rotation

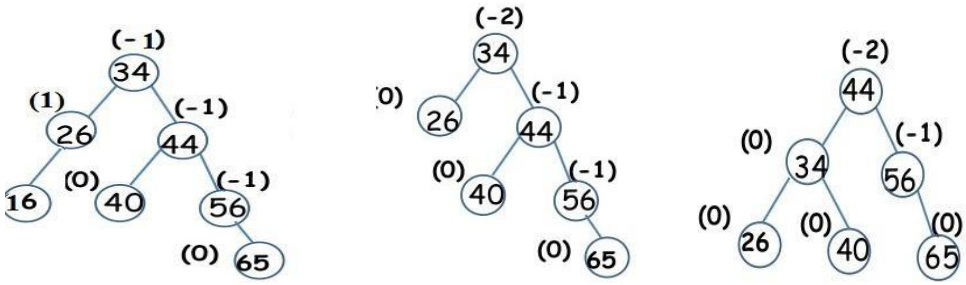
3. L-1 ROTATION

- L-1 imbalance occurs if the deletion takes place from the left sub tree of A and balance factor of B is -1
- L-1 rotation is Single Left rotation, that is applied at node A

General Notation:



Example:



Before deletion After deleting 16 After L-1 rotation

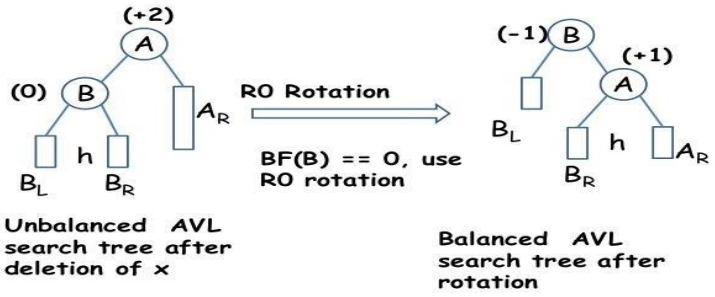
R – type Imbalance:

- The imbalance is of type R if the deletion took place from A's Right sub tree.
- The balance factor of A = **2**
- A has a left sub tree with root B
- R-type imbalance is sub classified into types R0, R1 and R-1 depending on the balance factor of B

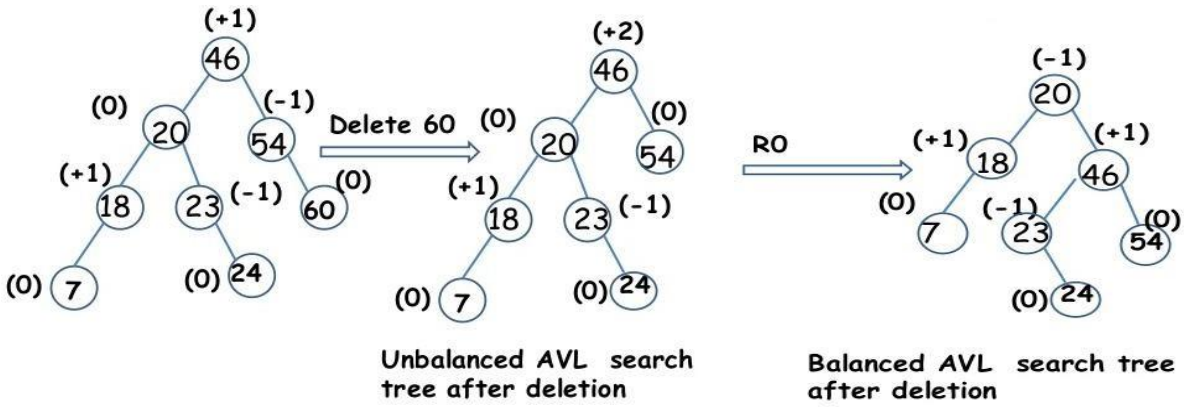
1. R0 ROTATION

- R0 imbalance occurs if the deletion takes place from the Right sub tree of A and balance factor of B is **0**
- R0 rotation is Single Right rotation, that is applied at node A

General Notation:



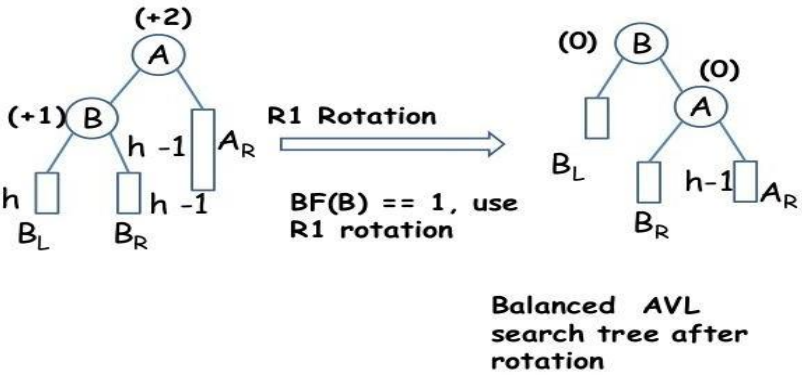
Example:



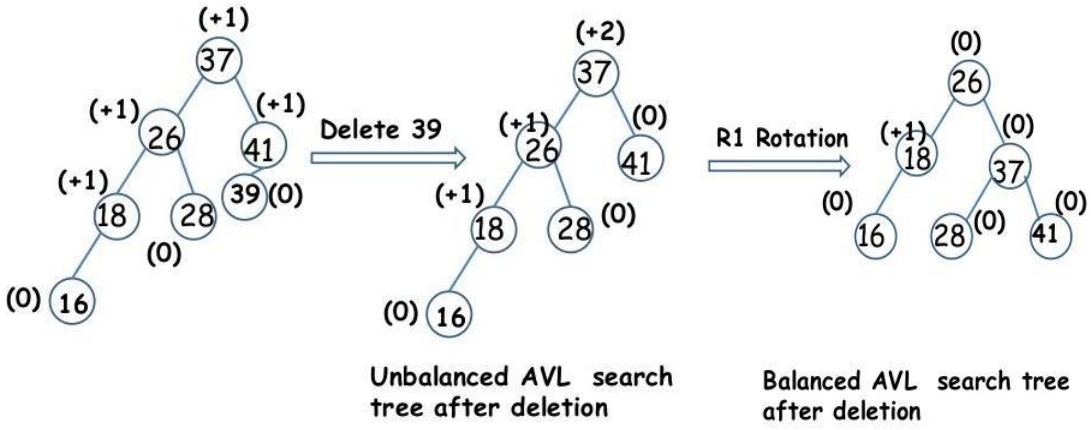
2. R1 ROTATION

- R1 imbalance occurs if the deletion takes place from the Right sub tree of A and balance factor of B is **1**
- R1 rotation is Single Right rotation, that is applied at node A

General Notation:



Example:



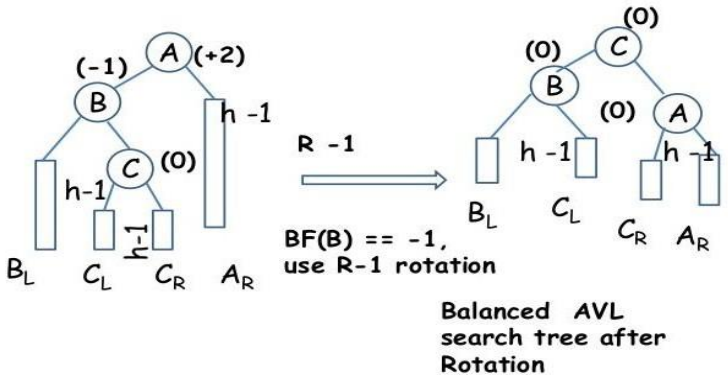
3. R-1 ROTATION

- R-1 imbalance occurs if the deletion takes place from the Right sub tree of A and balance factor of B is **-1**
- R-1 rotation is LR rotation, which involves 2 rotations:

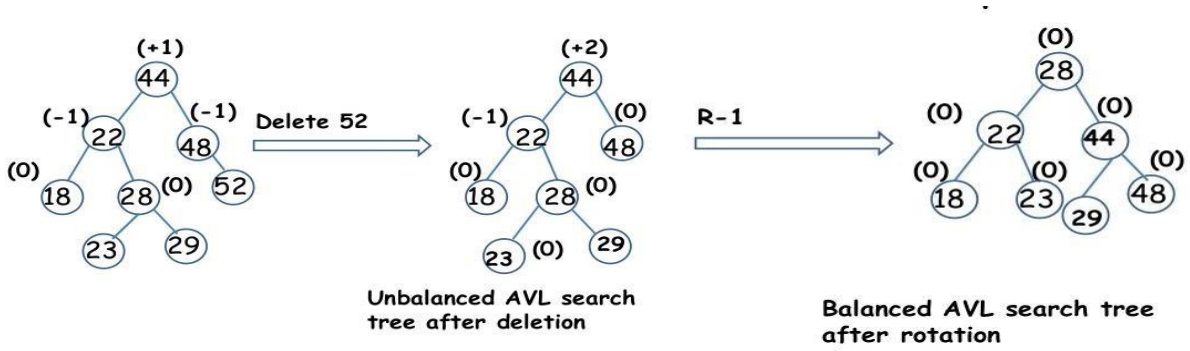
i. Single Left

ii. Single Right

General Notation:



Example:



Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.

Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

SPLAY TREES

- Splay tree is self adjusting or self balancing tree data structure.
- Splay tree is another variant of binary search tree.
- In a splay tree, the recently accessed element is placed at the root of the tree.

Definition: A splay tree is a self-adjusting binary search tree in which every operation on an element rearranges the tree so that the element is placed at the root position of the tree".

- All the operations on a splay tree are involved with a common operation called "Splaying".
- ***Splaying:*** Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.
- In a splay tree, splaying an element rearrange all the elements in the tree so that splayed element is placed at root of the tree.
- With the help of splaying an element we can bring most frequently used element closer to the root of the tree so that any operation on those element performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Advantages of Splaying:

1. Data accessed once, is often soon accessed again. Splaying does implicit *caching* by bringing it to the root
2. No heights to maintain, no imbalance to check for. Therefore, Less storage per node, easier to code

SPLAY ROTATIONS

- To splay an element we use sequence of rotation operations.
- Splay rotations are more or less similar to AVL tree rotations and will precede bottom up from node towards the root node.

There are 6 rotations in case of Splay Trees:

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

- The splay rotations are performed with regard to the specific node U, its parent parent-U and grandparent grandparent-U, until perhaps root node becomes U.
- The main aim of splaying is to move to access node U, up by two levels at every step.
- To do this we track the path from root node to access node U.

Every time the path turns LEFT is termed as ZIG, and every time the path turns RIGHT is termed as ZAG.

- In this case of single step down the tree the path could be either zig or zag.
- If two steps were considered, the path could be any one of zig-zig, zig-zag, zag-zig, zag-zag.

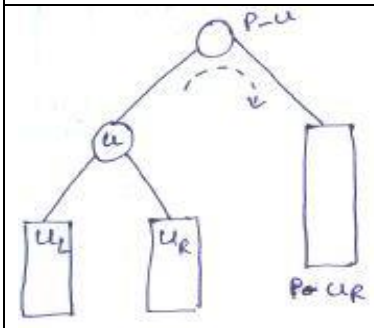
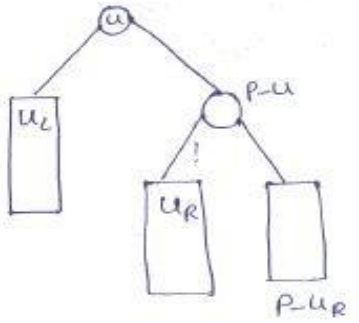
- Since splaying proceeds bottom up, if the length of the path from root to accessed node u is even, then the rotations appropriate to the two step series i.e., zig-zig, zig-zag, zag-zig, zag- zag are undertaken.
- If the length of the path from root node to access node u is odd, then the rotations may turn out to be one corresponding to a single step series either a zig or zag.

Zig: left path, then corresponding rotation is towards right

Zag: right path, then corresponding rotation is towards left

1.Zig Rotation: The Zig Rotation in a splay tree is similar to the single right rotation in AVL Tree rotations

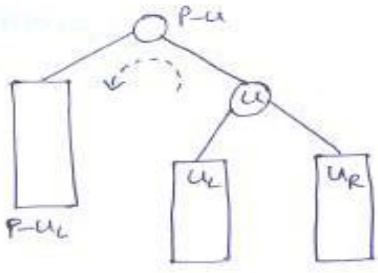
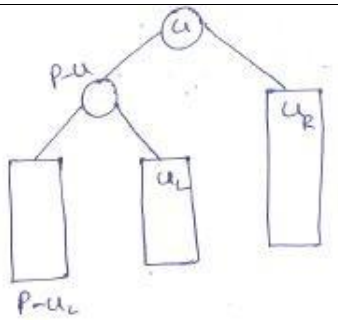
General Notation:

Before splaying	After splaying
	

Example:

2.Zag Rotation: The Zag Rotation in a splay tree is similar to the single left rotation in AVL Tree rotations.

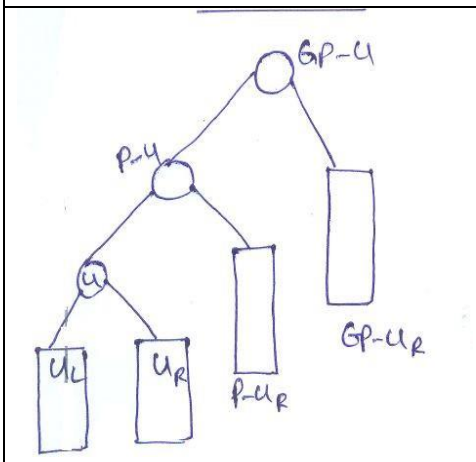
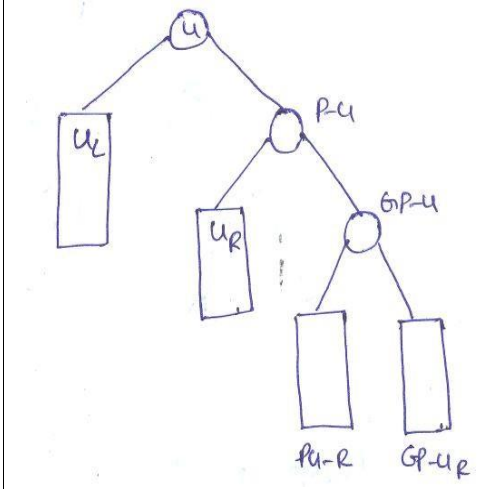
General Notation:

Before splaying	After splaying
	

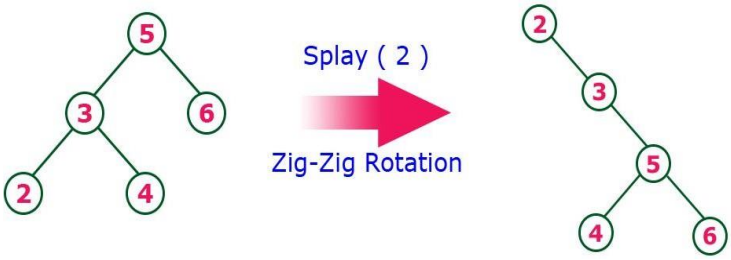
Example:

3.Zig-Zig Rotation: The Zig-Zig Rotation in a splay tree is a double zig rotation.

General Notation:

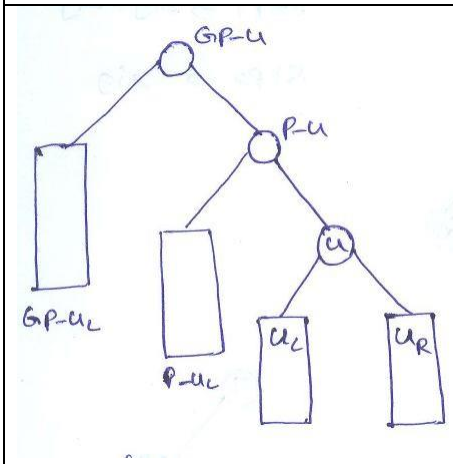
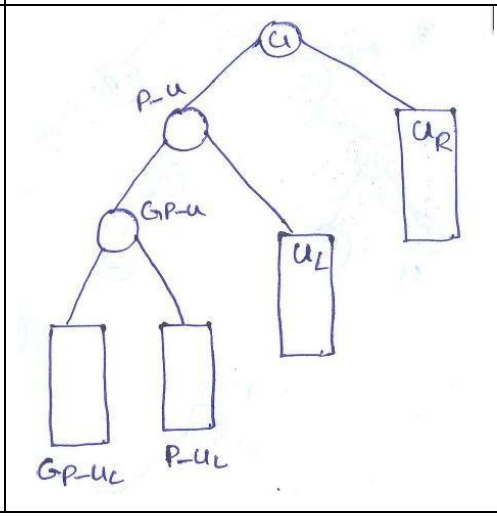
Before splaying	After splaying
	

Example:

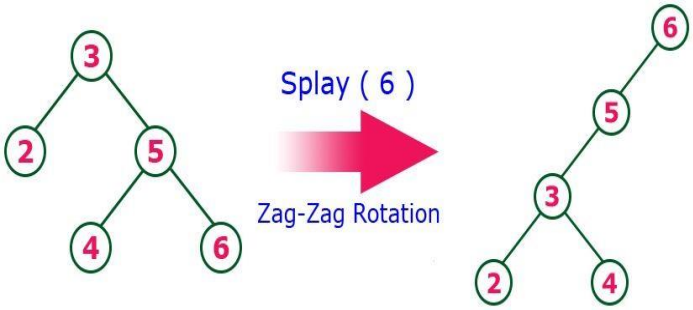


4.Zag-Zag Rotation: The Zag-Zag Rotation in a splay tree is a double zag rotation.

General Notation:

Before splaying	After splaying
	

Example:

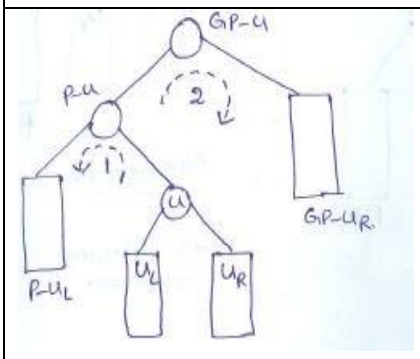
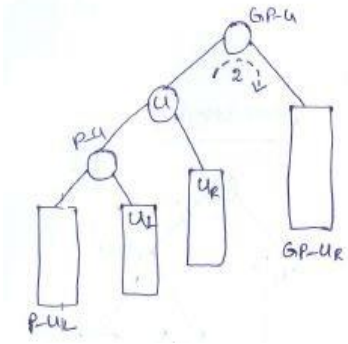
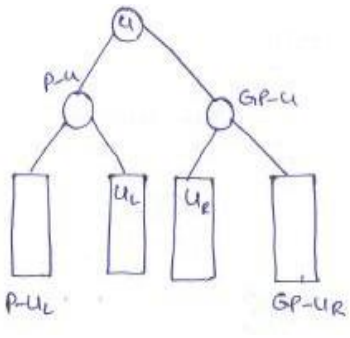


5.Zig-Zag Rotation: The Zig-Zag Rotation in a splay tree is a sequence of zig rotation followed by zag rotation.

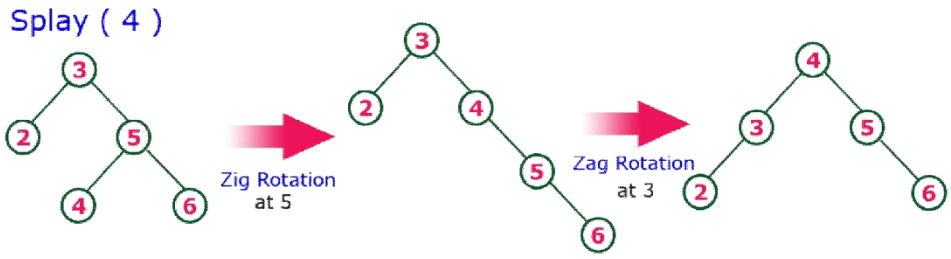
a. Rotation 1 is performed at p-u towards left

b. Rotation-2 is performed at gp-u towards right.

General Notation:

Initial tree	After rotation 1	After rotation 2
		

Example

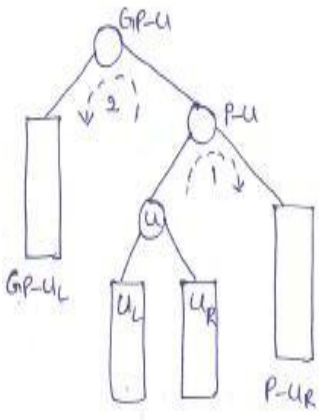
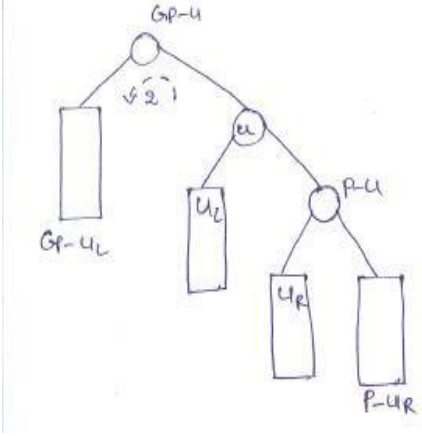
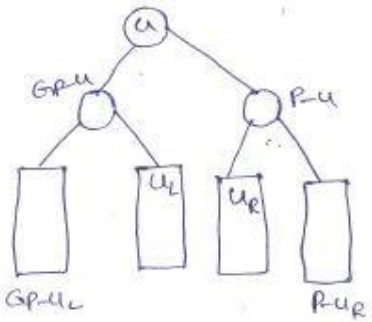


6.Zag-Zig: The Zag-Zig Rotation in a splay tree is a sequence of zag rotation followed by zig rotation.

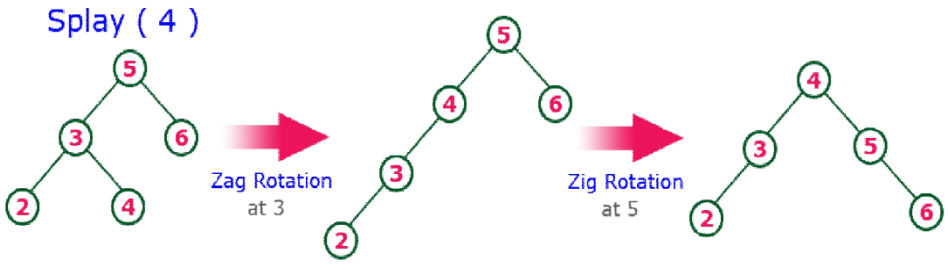
a. Rotation 1 is performed at p-u towards right

b. Rotation-2 is performed at gp-u towards left.

General Notation:

Initial tree	After rotation-1	After rotation 2
		

Example:



Here zig and zag represents single rotations corresponding to AVL rotations, and zig-zag and zag-zig represents double rotations corresponding to AVL rotations. However, zig-zig and zag-zag are not same as performing two single rotations

OPERATIONS ON SPLAY TREES

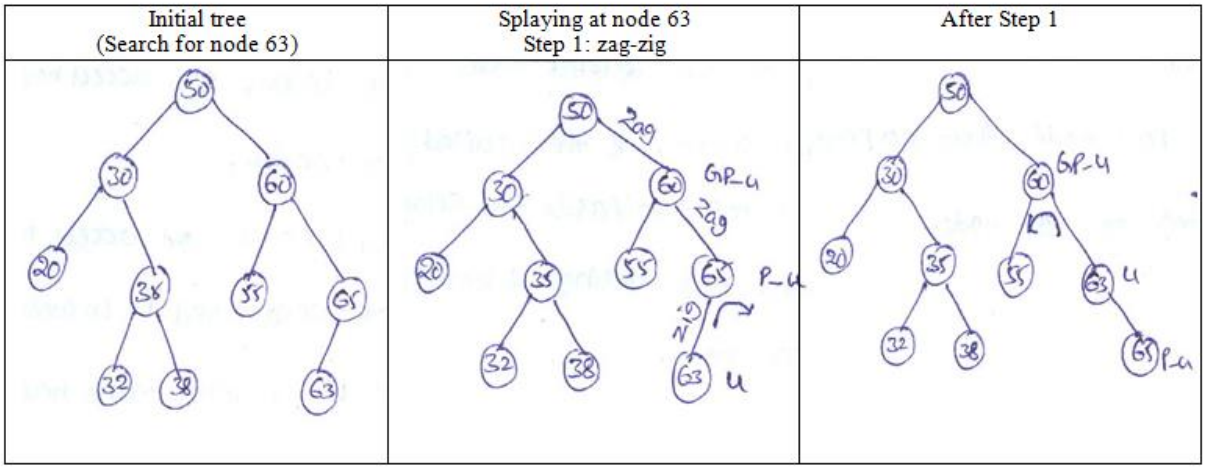
- All the operations on splay trees can be performed in the same way as in case of an ordinary Binary Search Tree except that, every operation is followed by splaying.
    1. Search
    2. Insert
    3. Delete

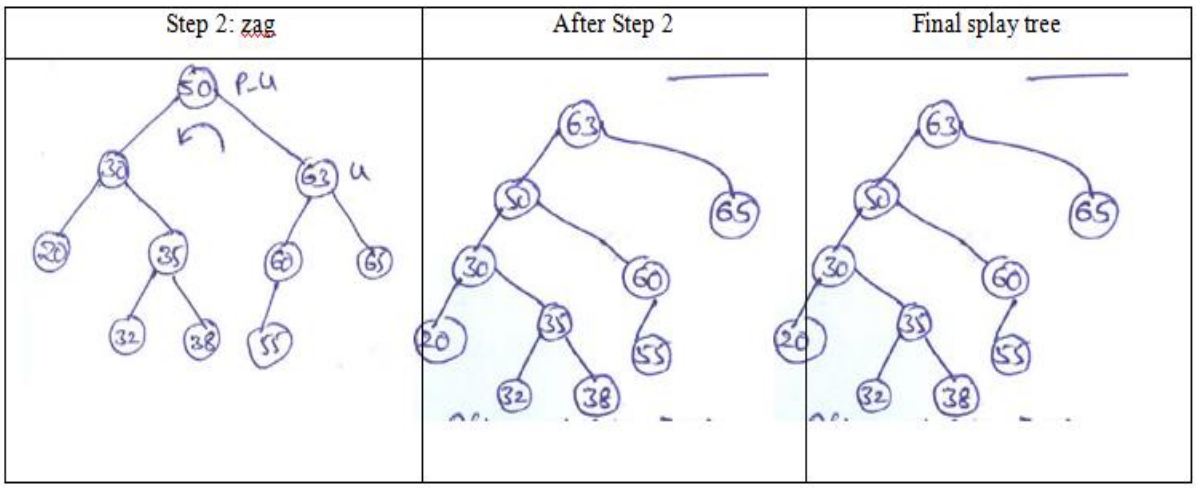
1. SEARCH

The last node accessed during the search is splayed

- If the search is successful, then the node that is found is splayed and becomes the new root.
- If the search is unsuccessful, the last node accessed prior to reaching the NULL pointer is splayed and becomes the new root.

Example:

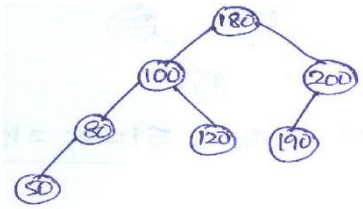




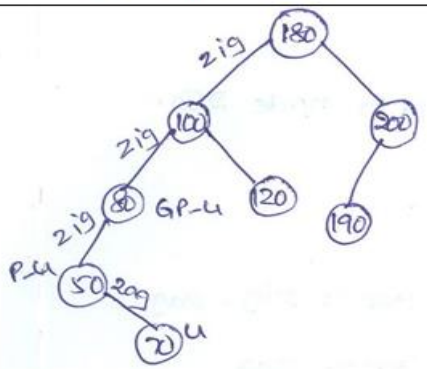
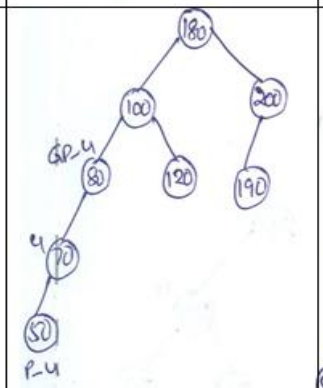
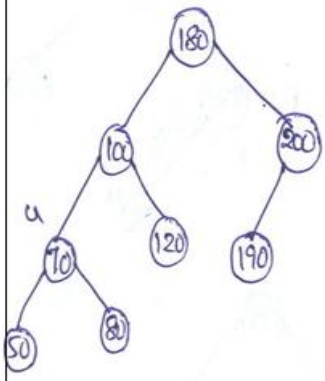
2. INSERTION

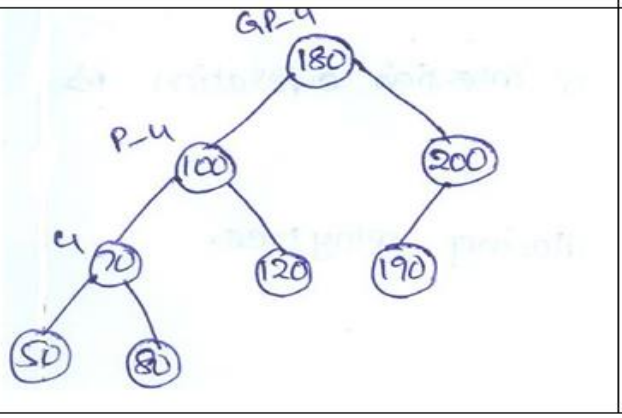
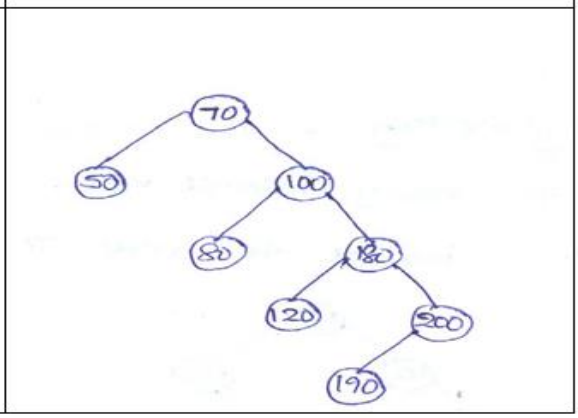
When an item is inserted, a splay is performed. As a result, the newly inserted item becomes the root of the tree.

Example: Insert the node 70 into the following splay tree



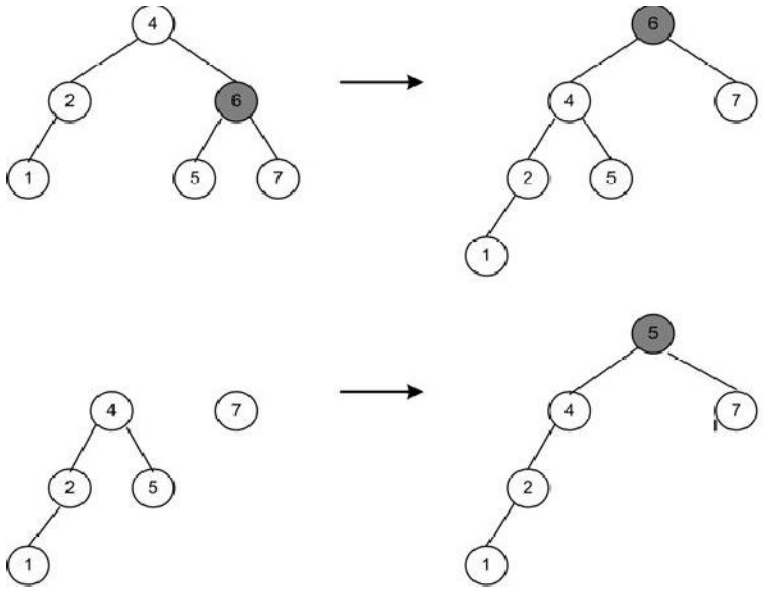
- Here node to be inserted as Right child to node 50
- Here node 70 becomes access node.

After 70 insert into tree	Step 1 : zig-zag	After zig-zag splaying
		

Step 2 zig- zig rotation	After Step2
	

3. DELETION

- Access the node to be deleted bringing it to the root.
- Delete the root leaving two Subtrees L (left) and R (right).
- Find the largest element in L, thus the root of L will have no right child.
- Make R the right child of L's root.



Example: Deleting 6 from the splay tree

Time complexity for various operations on Splay tree:

Operation	Average case	Worst case
Search	O(logn)	amortized O(logn)
Insertion	O(logn)	amortized O(logn)
Deletion	O(logn)	amortized O(logn)

Space Complexity : O(n)