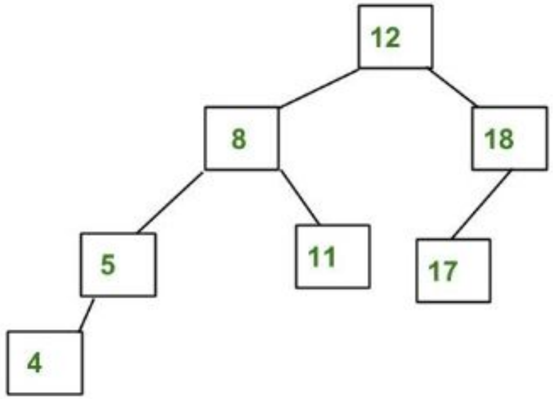
AVL & Splay Trees

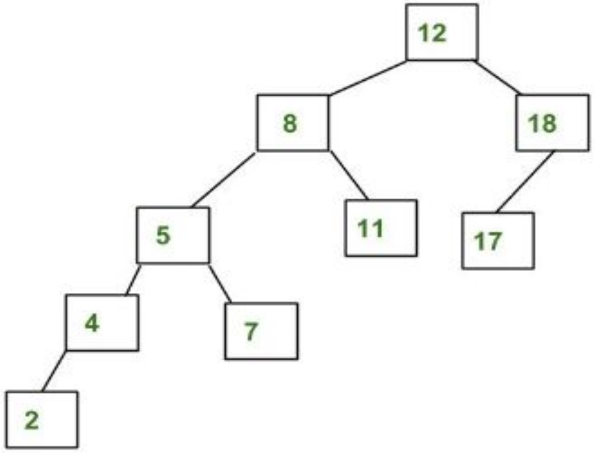
Ar. Gör Elif Ece ERDEM

AVL Trees

AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.



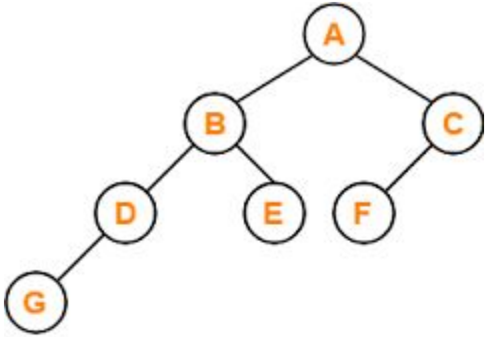
The differences between the heights of left and right subtrees for every node are less than or equal to 1 => **AVL Tree**



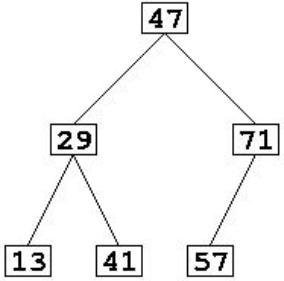
The differences between the heights of the left and right subtrees for 8 and 12 are greater than 1 => **not AVL Tree**

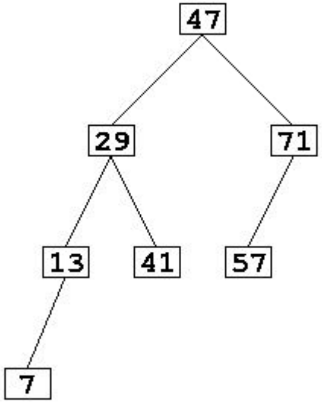
Why AVL Tree???

*Most of the BST operations (e.g., search, max, min, insert, delete.. etc) 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**. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.*



AVL Tree
(Height = 3)





For any node n in the tree, the height of the left subtree and right subtree differ by at most 1

AVL Tree - Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.
Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

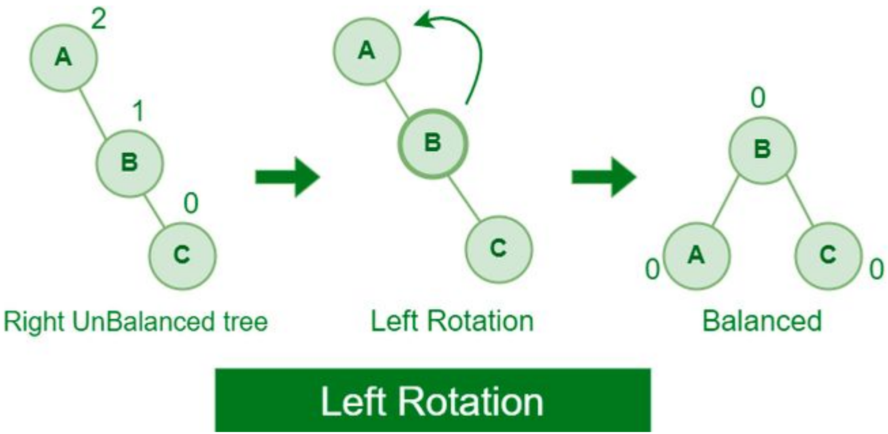
- Left Rotation
- Right Rotation

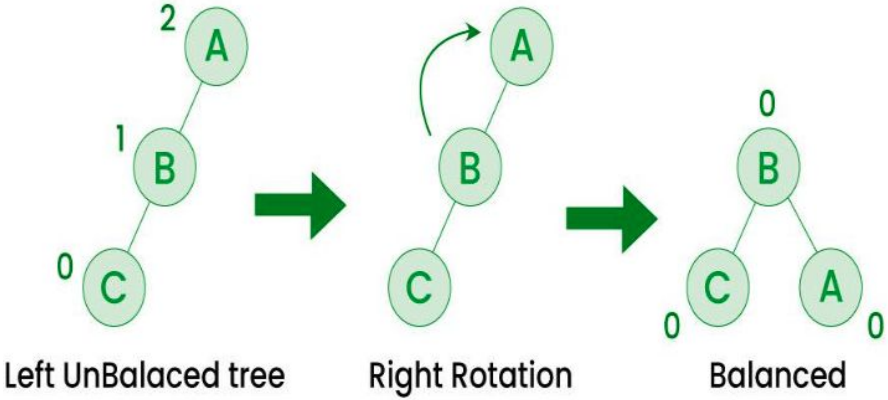
```
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right
side)

     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3

Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

AVL Tree - Rotations

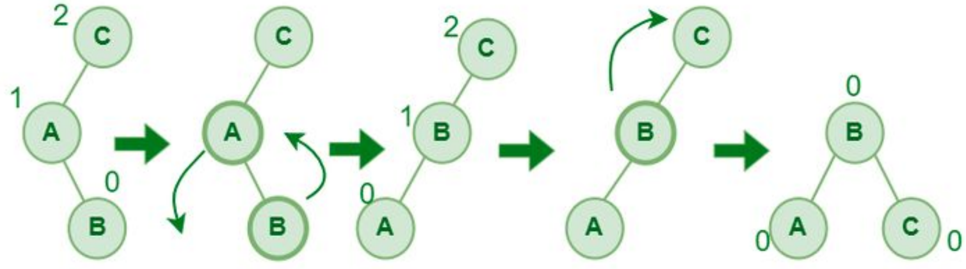




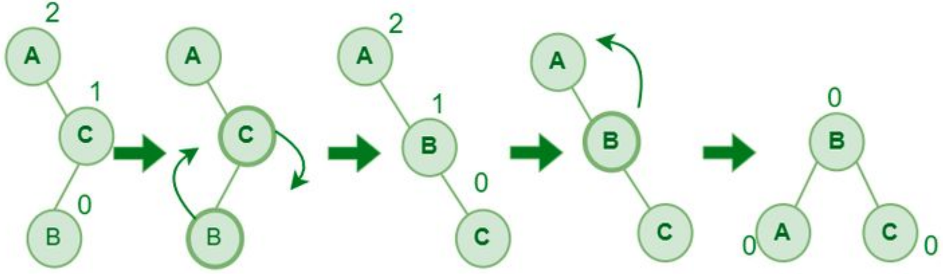
When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a **single left rotation.**

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a **single right rotation.**

AVL Tree - Rotations



Left-Right Rotation



Right-Left Rotation

```
   10                  7
  /                   / \
 5        —>>        5   10
  \
   7
```

```
10                  12
  \                /   \
   15     —>>    10    15
  /
12
```

AVL Trees - Insertion

Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
    - y is the left child of z and x is the left child of y (Left Left Case)
    - y is the left child of z and x is the right child of y (Left Right Case)
    - y is the right child of z and x is the right child of y (Right Right Case)
    - y is the right child of z and x is the left child of y (Right Left Case)

AVL Trees - Insertion

1. Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y 
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \ 
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

2. Left Right Case

```
     z                               z                           x
    / \                            /   \                        /  \ 
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

3. Right Right Case

```
  z                                y
 /  \                            /   \ 
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

4. Right Left Case

```
   z                            z                            x
  / \                          / \                          /  \ 
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

AVL Trees - Insertion Approach

The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

Follow the steps mentioned below to implement the idea:

- Perform the normal BST insertion.
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor **(left subtree height – right subtree height)** of the current node.
- If the balance factor is greater than **1,** then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root.

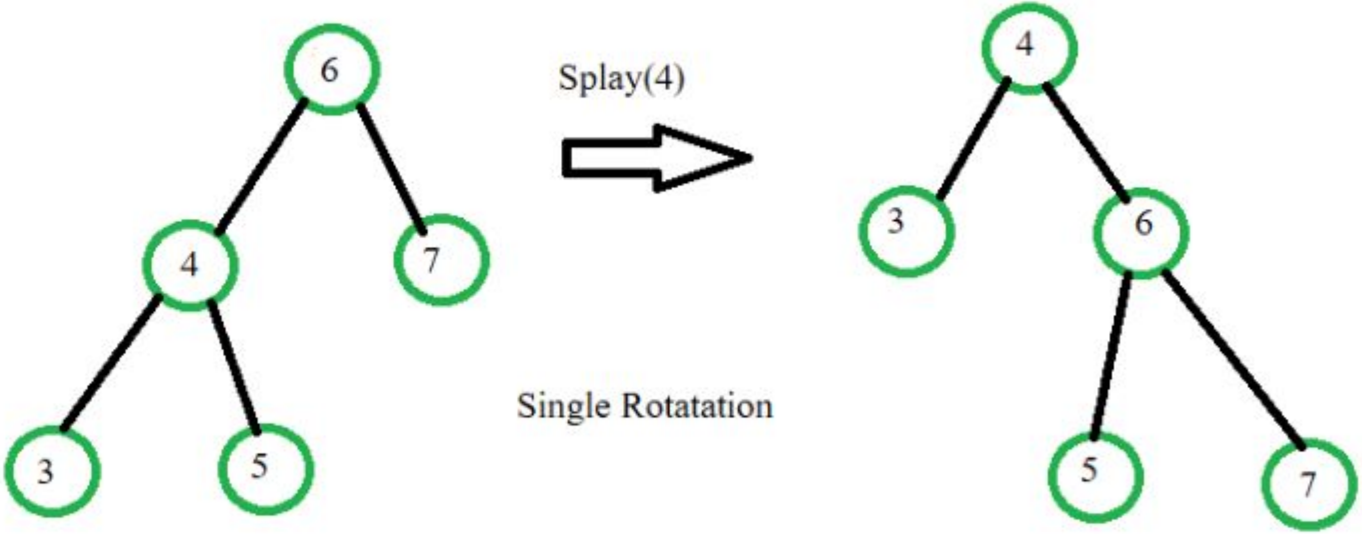
Splay Tree

A splay tree is a self-balancing binary search tree, designed for efficient access to data elements based on their key values.

- The key feature of a splay tree is that each time an element is accessed, it is **moved to the root of the tree**, creating a more balanced structure for subsequent accesses.
- Splay trees are characterized by their use of rotations, which are local transformations of the tree that change its shape but preserve the order of the elements.
- Rotations are used to bring the accessed element to the root of the tree, and also to rebalance the tree if it becomes unbalanced after multiple accesses.

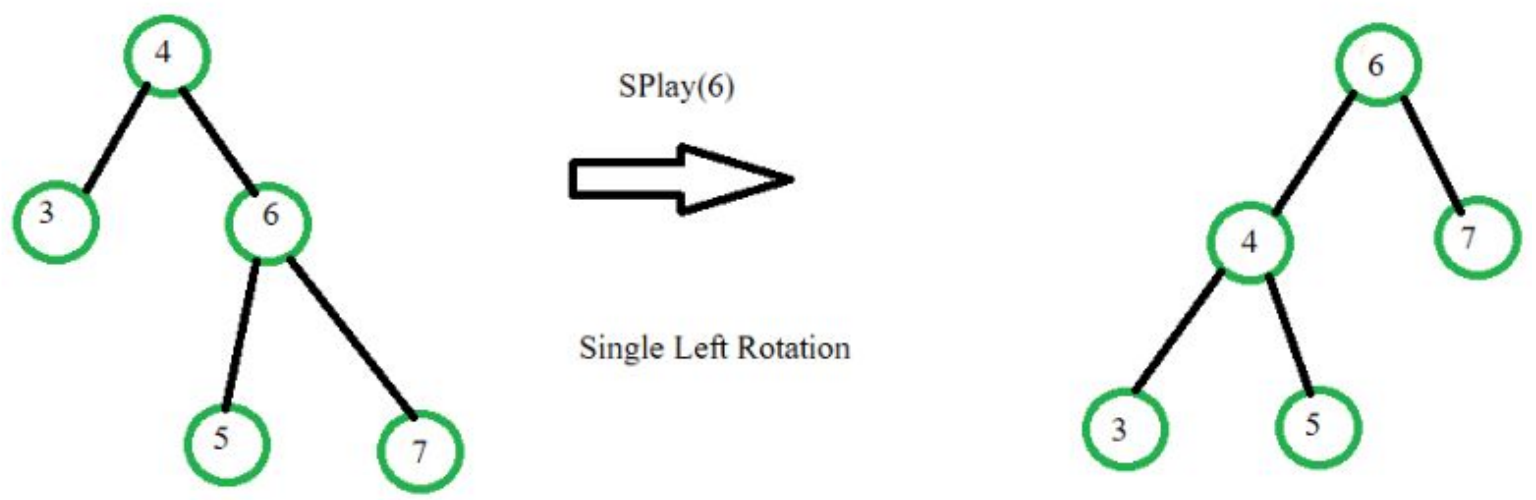
Splay Tree - Rotation Operations
Zig Rotation

The Zig Rotation in splay trees operates in a manner similar to the single right rotation in AVL Tree rotations. This rotation results in nodes moving one position to the right from their current location.



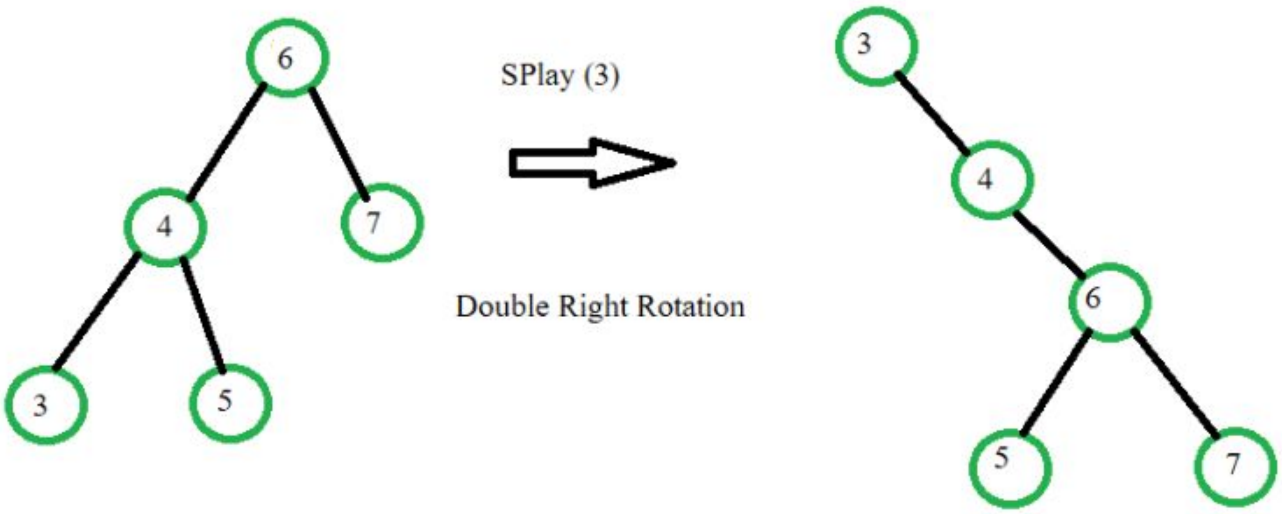
Splay Tree - Rotation Operations
Zag Rotation

The Zag Rotation in splay trees operates in a similar fashion to the single left rotation in AVL Tree rotations. During this rotation, nodes shift one position to the left from their current location.



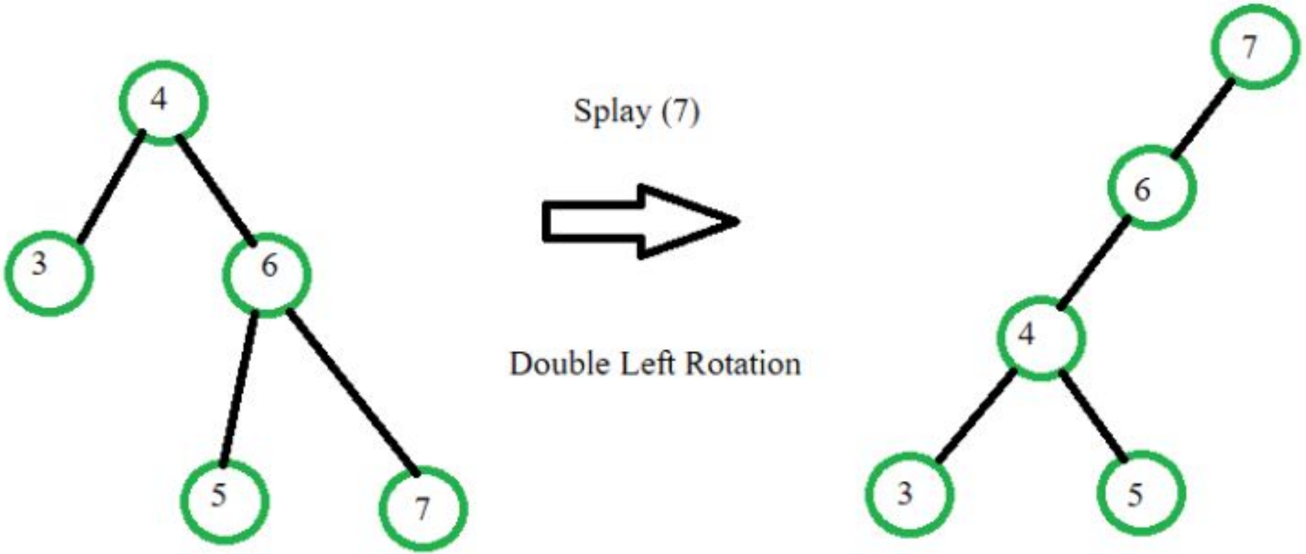
Splay Tree - Rotation Operations
Zig-Zig Rotation

The Zig-Zig Rotation in splay trees is a double zig rotation. This rotation results in nodes shifting two positions to the right from their current location.



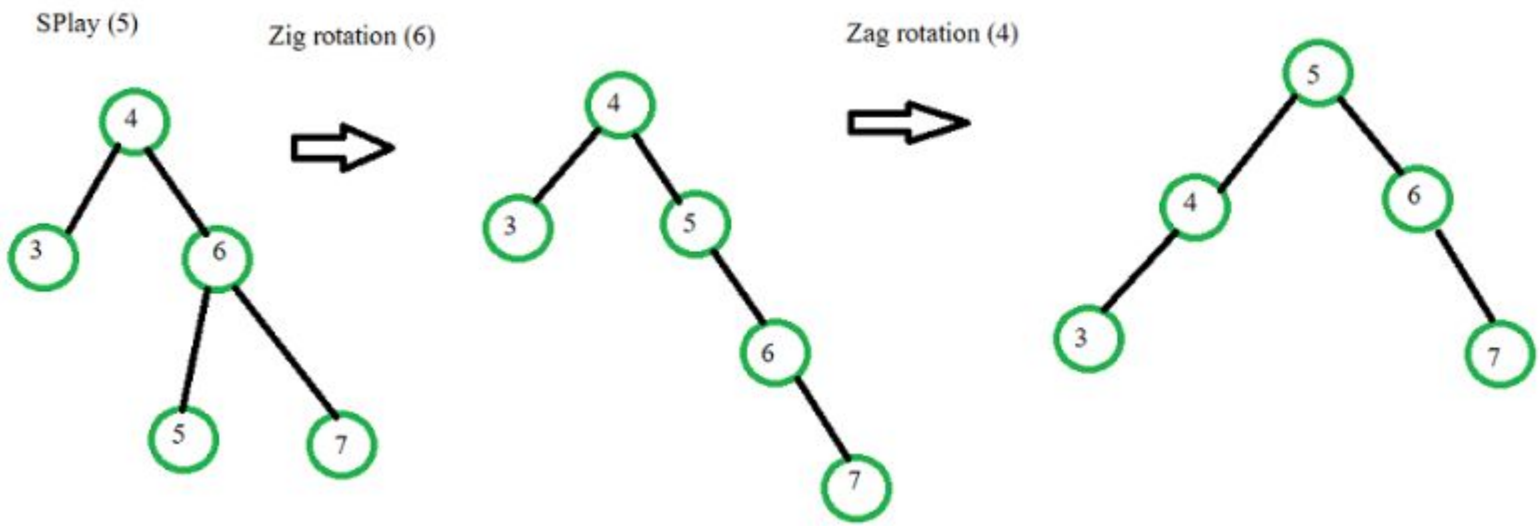
Splay Tree - Rotation Operations
Zag-Zag Rotation

In splay trees, the Zag-Zag Rotation is a double zag rotation. This rotation causes nodes to move two positions to the left from their present position.



Splay Tree - Rotation Operations
Zig-Zag Rotation

The Zig-Zag Rotation in splay trees is a combination of a zig rotation followed by a zag rotation. As a result of this rotation, nodes shift one position to the right and then one position to the left from their current location.



Splay Tree - Rotation Operations
Zag-Zig Rotation

The Zag-Zig Rotation in splay trees is a series of zag rotations followed by a zig rotation. This results in nodes moving one position to the left, followed by a shift one position to the right from their current location.