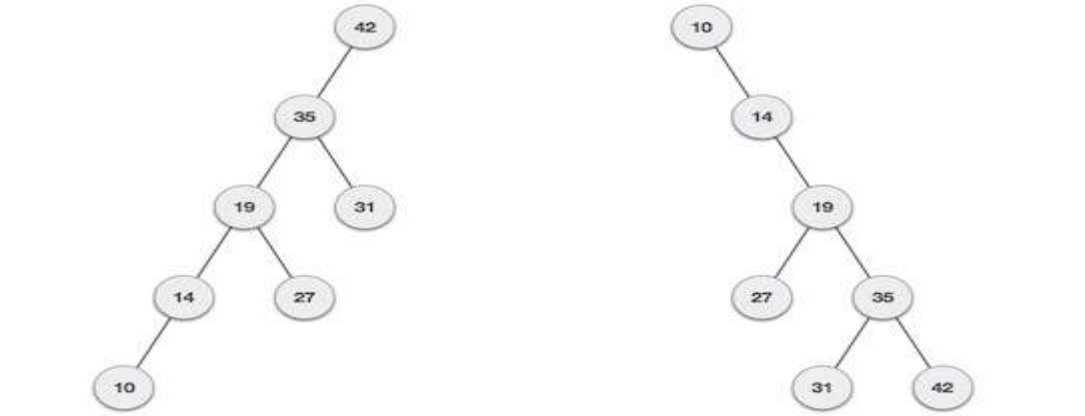
- # Avl Trees

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –

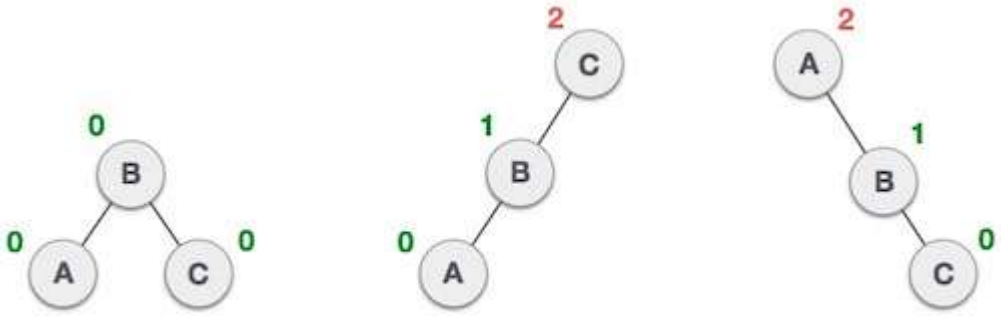


If input 'appears' non-increasing manner If input 'appears' in non-decreasing manner

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 and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –



Balanced Not balanced Not balanced

In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

```
BalanceFactor = height(left-sutree) – height(right-sutree)
```

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

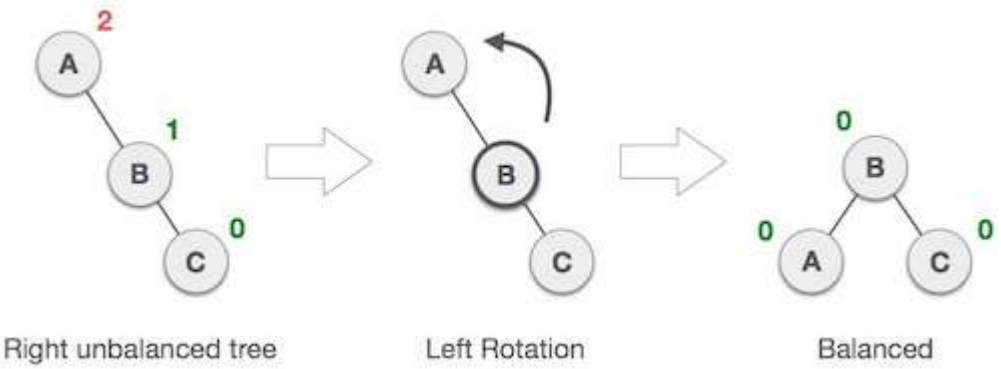
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

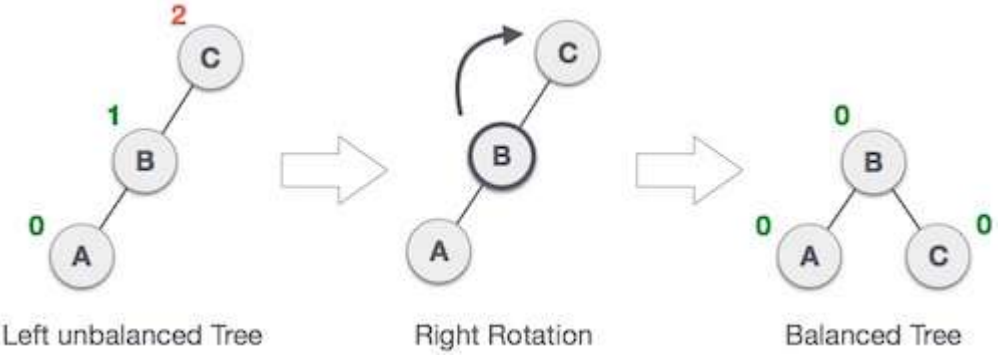
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A**the left-subtree of B.

Right Rotation

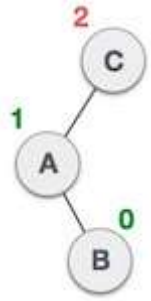
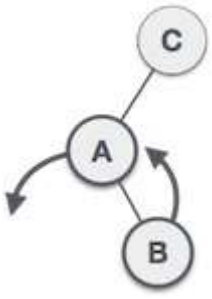
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

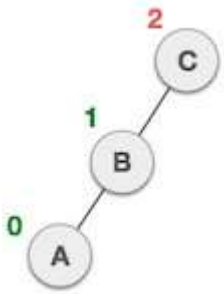
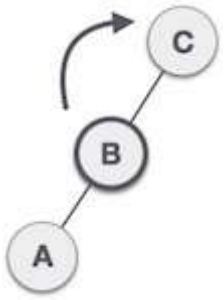
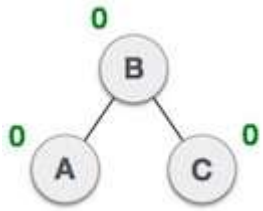


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

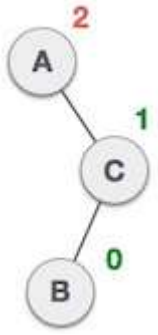
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

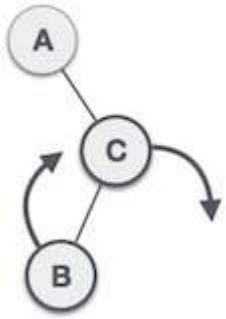
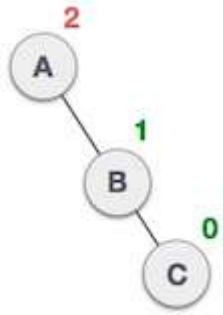
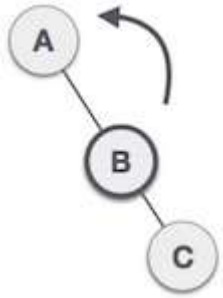
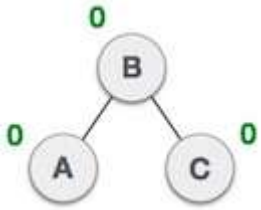
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.

	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.

	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

- Red black Trees:

The problem with BST is that, depending on the order of inserting elements in the tree, the tree shape can vary. In the worst cases (such as inserting elements in order) the tree will look like a linked list in which each node has only a right child. This yields O(n) for primitive operations on the BST, with n the number of nodes in the tree.To solve this problem many variations of binary search trees exist. Of

these variations, red-black trees provide a well-balanced BST that guarantees a logarithmic bound on primitive operations.

Red-black Trees

Red-black trees are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations. This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one.

A red-black tree is a binary search tree with the following properties:

1. Every node is colored with either red or black.
2. All leaf (nil) nodes are colored with black; if a node's child is missing then we will assume that it has a nil child in that place and this nil child is always colored black.
3. Both children of a red node must be black nodes.
4. Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). We call this number the**black height** of n, which is denoted by bh(n).
   Figure 5 shows an example of a red-black tree.

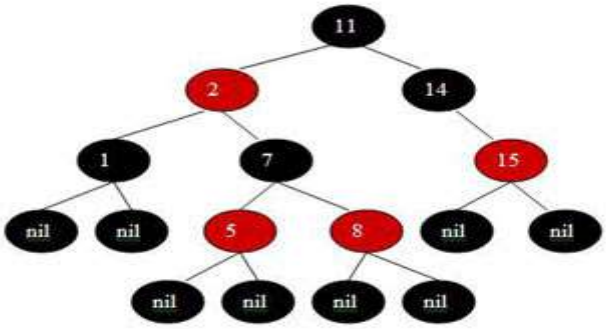


Figure 5. an example of a red-black tree

Using these properties, we can show in two steps that a red-black tree which contains n nodes has a height of O(log n), thus all primitive operations on the tree will be of O(log n) since their order is a function of tree height.

Rotation is a binary operation, between a parent node and one of its children, that swaps nodes and modifys their pointers while preserving the inorder traversal of the tree (so that elements are still sorted).

There are two types of rotations: left rotation and right rotation. Left rotation swaps the parent node with its right child, while right rotation swaps the parent node with its left child. Here are the steps involved in for left rotation (for right rotations just change "left" to "right" below):

- Assume node x is the parent and node y is a non-leaf right child.
- Let y be the parent and x be its left child.
- Let y's left child be x's right child.

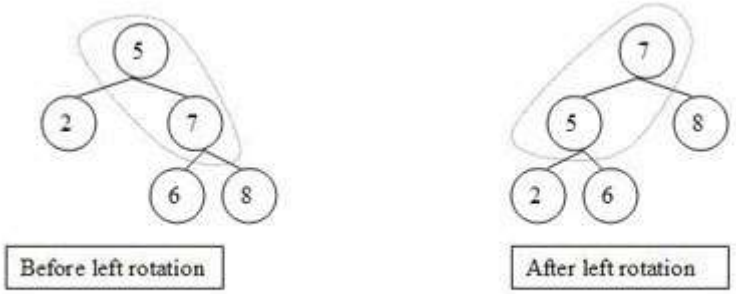


Figure 6. Left rotation

Operations on red-black tree (insertion, deletion and retrieval)

Red-black tree operations are a modified version of BST operations, with the modifications aiming to preserve the properties of red-black trees while keeping the operations complexity a function of tree height.

Red-black tree insertion:

Inserting a node in a red-black tree is a two step process:

1. A BST insertion, which takes O(log n) as shown before.
2. Fixing any violations to red-black tree properties that may occur after applying step 1. This step is O(log n) also, as we start by fixing the newly inserted node, continuing up along the path to the root node and fixing nodes along that path. Fixing a node is done in constant time and involves re-coloring some nodes and doing rotations.

Accordingly the total running time of the insertion process is O(log n). Figure 7 shows the red-black tree in figure 5 before and after insertion of a node with value 4.

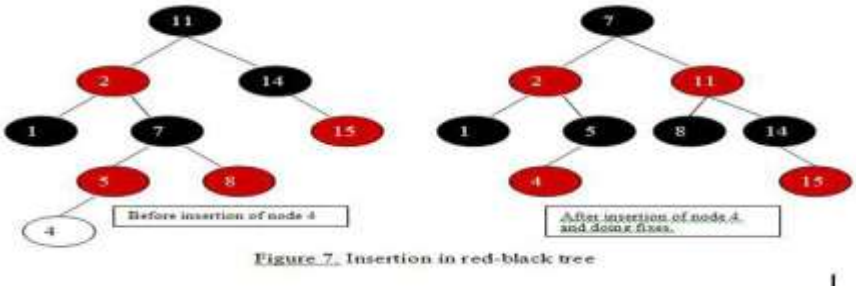


Figure 7. Insertion in red-black tree

You can see how the swap operations modified the tree structure to keep it balanced.ep it balanced.

Red-black tree deletion:

The same concept behind red-black tree insertions applies here. Removing a node from a red-black tree makes use of the BST deletion procedure and then restores the red-black tree properties in O(log

n). The total running time for the deletion process takes O(log n) time, then, which meets the complexity requirements for the primitive operations.

Red-black tree retrieval:

Retrieving a node from a red-black tree doesn't require more than the use of the BST procedure, which takes O(log n) time.

Conclusion

Although you may never need to implement your own set or map classes, thanks to their common built-in support, understanding how these data structures work should help you better assess the performance of your applications and give you more insight into what structure is right for a given task.

- # Splay Trees

Splay tree is another varient of binary search tree. In a splay tree, the recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on an element rearrange the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at root of the tree. All the operations on a splay tree are involved with a common operation called "**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.

Every operation on a splay tree performs the splaying operation. For example, the insertion operation first inserts the new element as it inserted into the binary search tree, after insertion the newly inserted element is splayed so that it is placed at root of the tree. The search operation in a splay tree is search the element using binary search process then splay the searched element so that it placed at the root of the tree.

In a splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

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

Example

Zig Rotation

The **Zig Rotation** in a splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation every node moves one position to thrrigh from its current position. Consider the following example...

Zag Rotation

The **Zag Rotation** in a splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation every node moves one position to the left from its current position. Consider the following example...

Zig-Zig Rotation

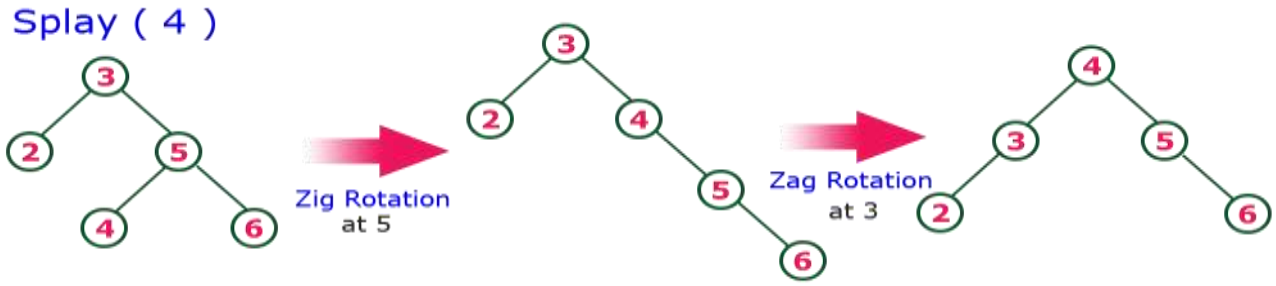
The **Zig-Zig Rotation** in a splay tree is a double zig rotation. In zig-zig rotation every node moves two position to the right from its current position. Consider the following example...

Zag-Zag Rotation

The **Zag-Zag Rotation** in a splay tree is a double zag rotation. In zag-zag rotation every node moves two position to the left from its current position. Consider the following example...

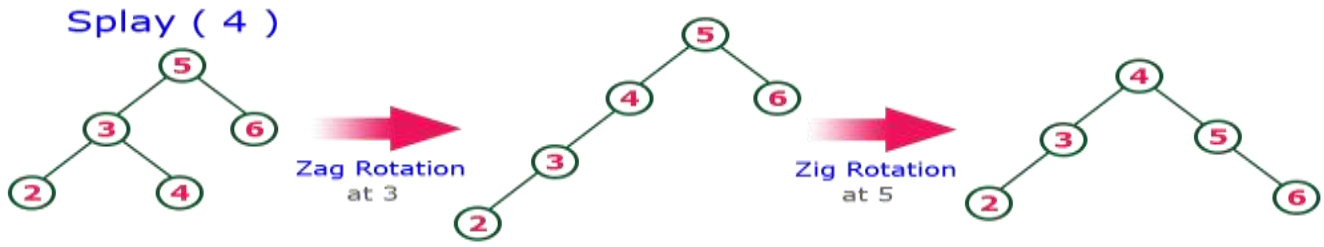
Zig-Zag Rotation

The **Zig-Zag Rotation** in a splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in a splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **step 3:** If tree is not Empty then insert the newNode as a leaf node using Binary Search tree insertion logic.
- **Step 4:** After insertion, **Splay** the **newNode**

Deletion Operation in Splay Tree

In a Splay Tree, the deletion operation is similar to deletion operation in Binary Search Tree. But before deleting the element first we need to **splay** that node then delete it from the root position then join the remaining tree.

- # Binary Heap:

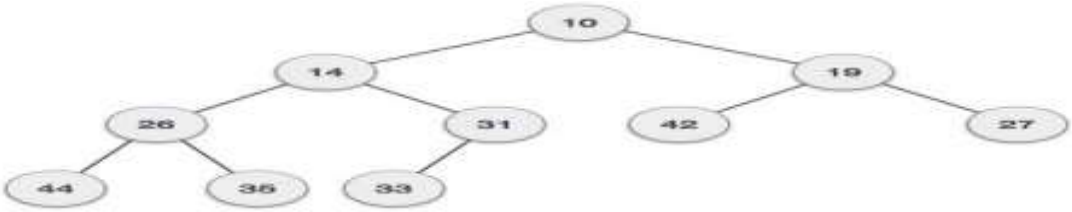
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

$$key(\alpha) \geq key(\beta)$$

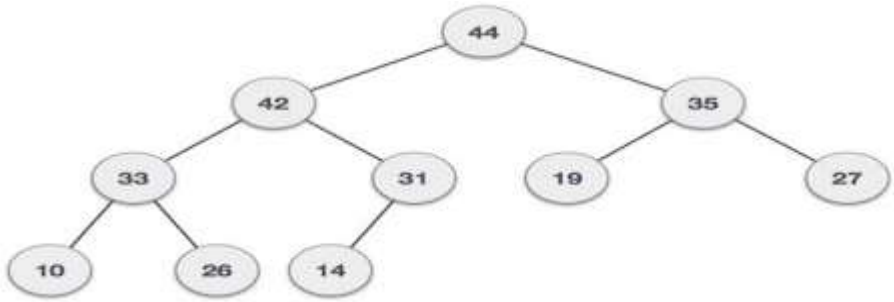
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

```
For Input →35 33 42 10 14 19 27 44 26 31
```

Min-Heap − Where the value of the root node is less than or equal to either of its children.



Max-Heap – Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

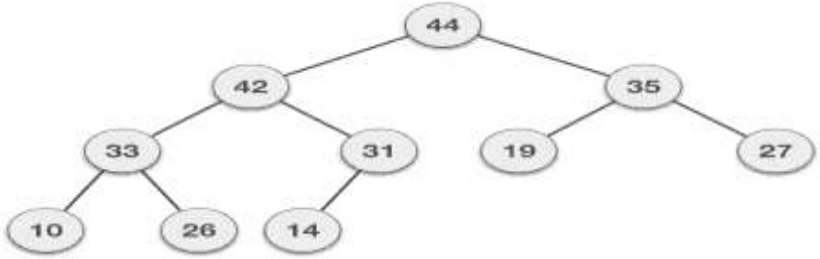
We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

```
Step 1 – Create a new node at the end of heap.
Step 2 – Assign new value to the node.
Step 3 – Compare the value of this child node with its parent.
Step 4 – If value of parent is less than child, then swap them.
Step 5 – Repeat step 3 & 4 until Heap property holds.
```

Note – In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

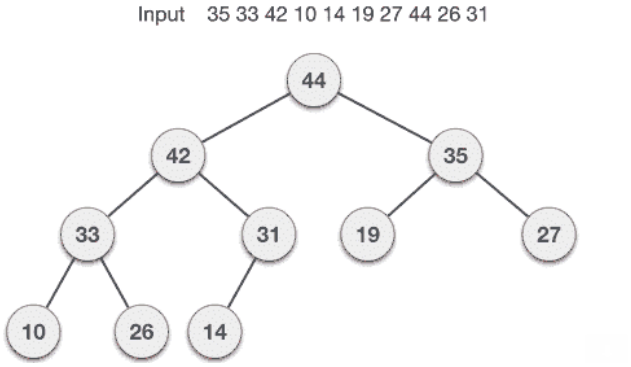
Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.



Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

```
Step 1 - Remove root node.
Step 2 - Move the last element of last level to root.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.
```



- # Leftist Heap:

Leftist Tree / Leftist Heap

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a complete binary tree), a leftist tree may be very unbalanced.

Below are time complexities of **Leftist Tree / Heap**.

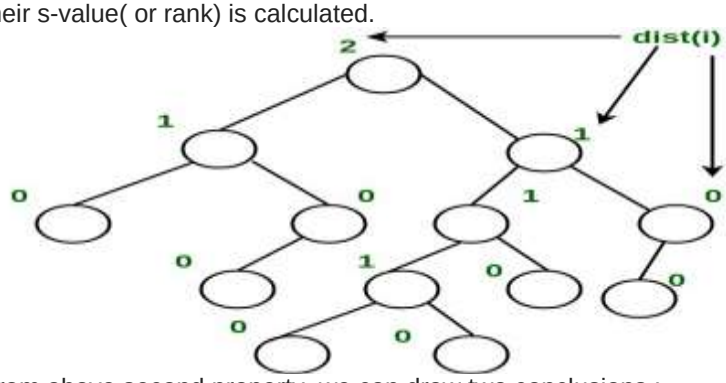
```
Function       Complexity             Comparison
1) Get Min:       O(1)       [same as both Binary and Binomial]
2) Delete Min:    O(Log n)   [same as both Binary and Binomial]
3) Insert:        O(Log n)   [O(Log n) in Binary and O(1) in
                             Binomial and O(Log n) for worst case]
4) Merge:        O(Log n)  [O(Log n) in Binomial]
```

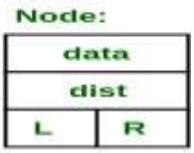
A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property :**key(i) >= key(parent(i))
2. **Heavier on left side :** dist(right(i)) <= dist(left(i)). Here, dist(i) is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and dist(i) = 1 + dist(right(i)).

Example: The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right subtree of this node is null

and its parent has a distance of 1 by dist(i) = 1 + dist(right(i)). The same is followed for each node and their s-value(or rank) is calculated.





From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has x nodes, then leftist heap has atleast 2x – 1 nodes. This means the length of path to rightmost leaf is O(log n) for a leftist heap with n nodes.

Operations :

1. The main operation is merge().
2. deleteMin() (or extractMin() can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done be create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

Idea behind Merging:

Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion:
   – Update dist() of merged root.
5. – Swap left and right subtrees just below root, if needed, to keep leftist property of merged result.

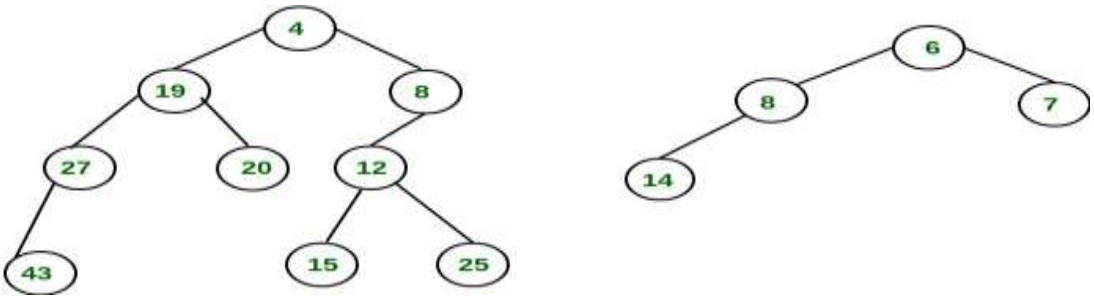
6. Detailed Steps for Merge:

7

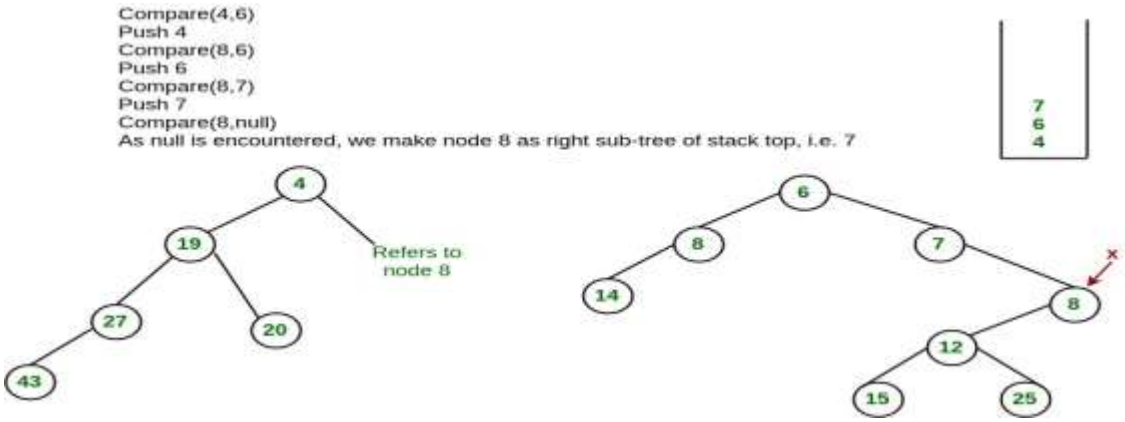
1. Compare the roots of two heaps.
2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

Example:

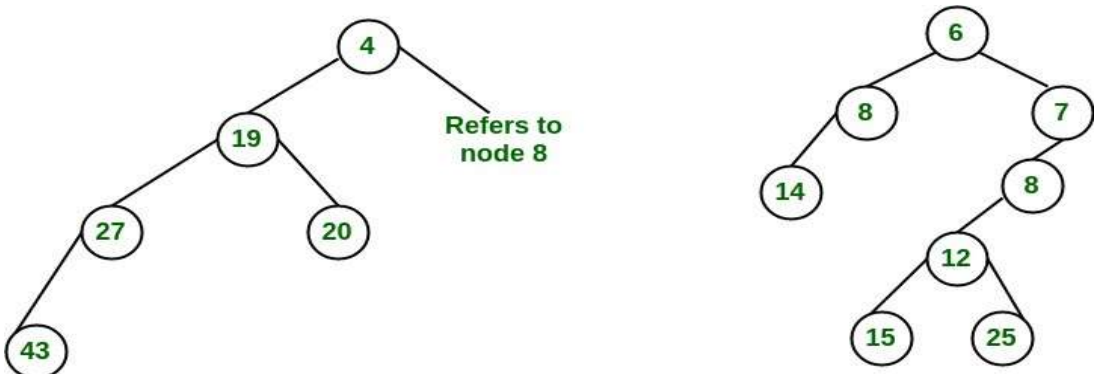
Consider two leftist heaps given below:



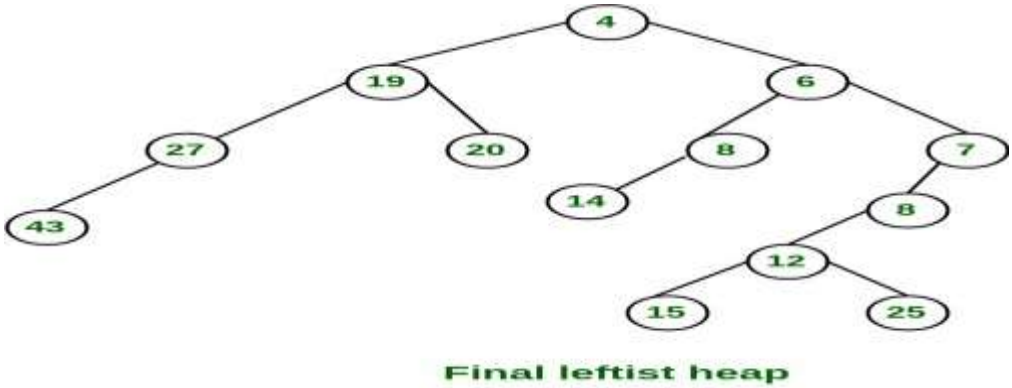
Merge them into a single leftist heap



The subtree at node 7 violates the property of leftist heap so we swap it with the left child and retain the property of leftist heap.



Convert to leftist heap. Repeat the process



Final leftist heap

The worst case time complexity of this algorithm is O(log n) in the worst case, where n is the number of nodes in the leftist heap.

Properties of Splay Tree:

Property 1. (Static Finger Bound): The amortized cost **cˆf (x)** of searching for x in a splay tree, given a fixed node f is: **cˆf (x) = O(log (1 + |x − f|))**
where |x − f| is the distance from node x to f in the total order.

Property 2. (Sequential access) : Accessing n elements in sorted order in a splay tree takes O(n) time (O(1) amortized per element).

Property 3. (Dynamic finger) : After accessing element x, the amortized cost of accessing y in splay trees is O(log (1 + |x − y|)).

Property 4. (Working set) : The amortized cost of searching for x in a splay tree is O(1 + log tx), where tx is the number of distinct elements searched for since the last search for x.

Differences between AVL & RED-BLACK Trees

S.No	AVL TREE	RED BLACK TREE
1	AVL tree is more rigidly balanced than RED-Black Tree	RED-BLACK Tree is not that much rigidly balanced than AVL Tree
2	Insertion & deletion is slower than RED-Black Tree due to more rotation operations.	Insertion and Deletion is comparatively more faster than AVL Tree.
3.	In Searching a Node AVL Tree works more faster than RED-Black Tree	Number of Look ups is comparatively more than AVL Tree.
4	The time complexity to traverse from root node to the deepest Leaf is ~ log(n+2)	The time complexity to traverse from root node to the deepest node is ~2 log(n+1)
5	AVL Tree is mostly used for balanced Binary Search Tree.	RED-Black Tree is also used for Balanced Binary Search Tree

AVL Tree Data-structure

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right sub-trees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

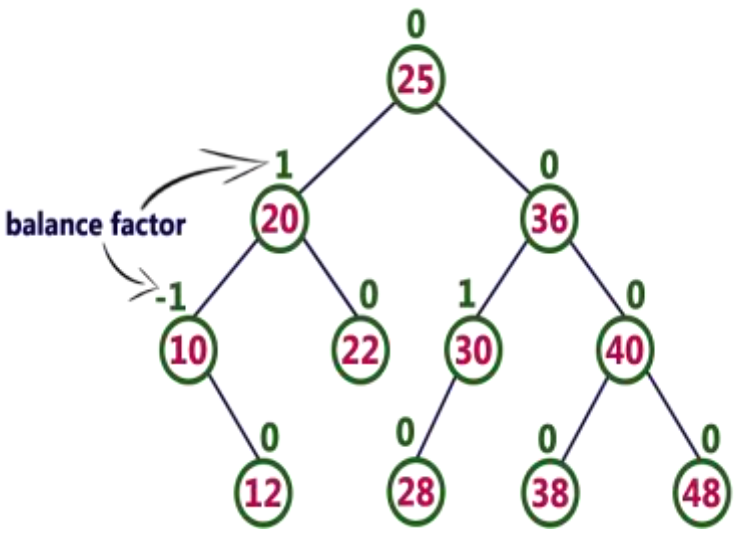
An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right sub-trees of that node. The balance factor of a node is calculated either **height of left sub-tree - height of right sub-tree** (OR) **height of right subtree - height of left sub-tree**. In the following explanation, we calculate as follows...

Balance factor = height Of LeftSub-tree – height Of RightSub-tree

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

AVL Tree Rotations

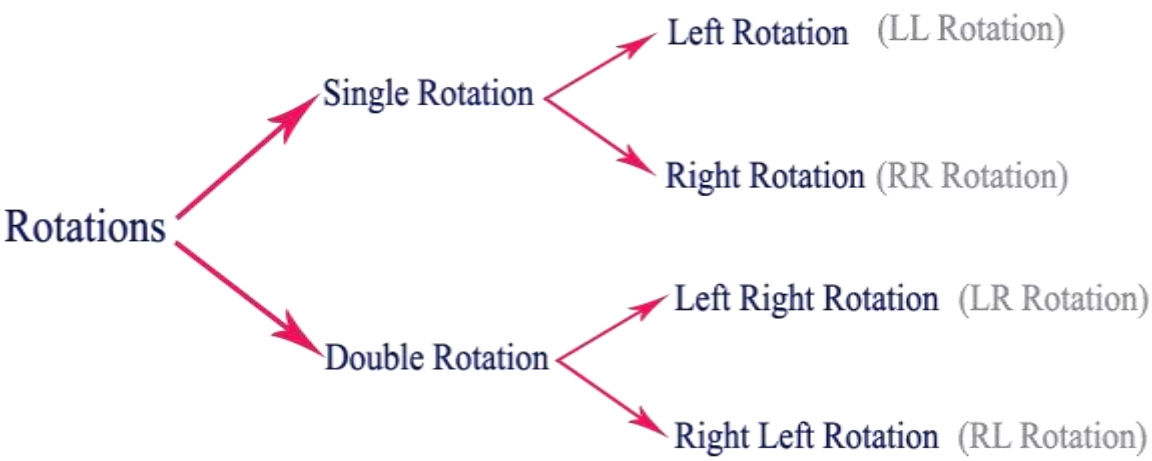
In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude

the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

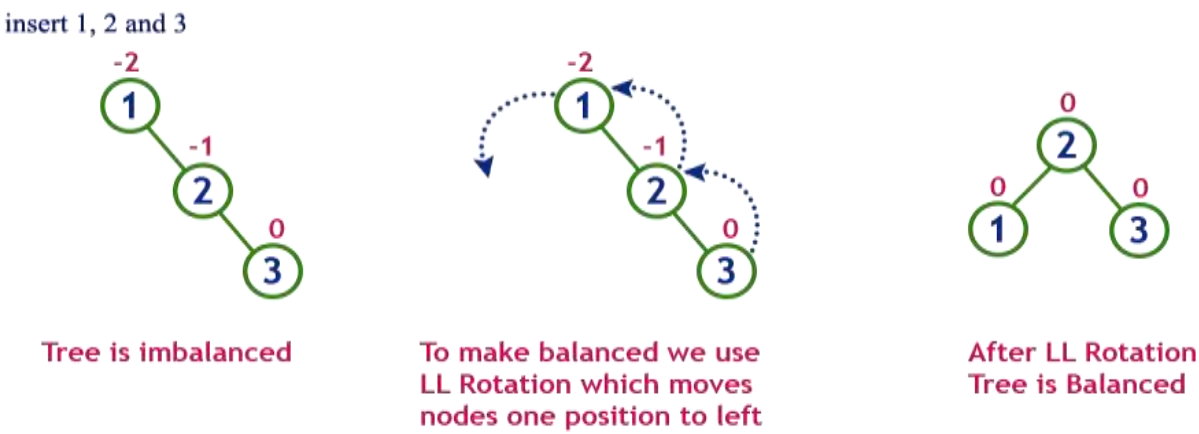
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

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



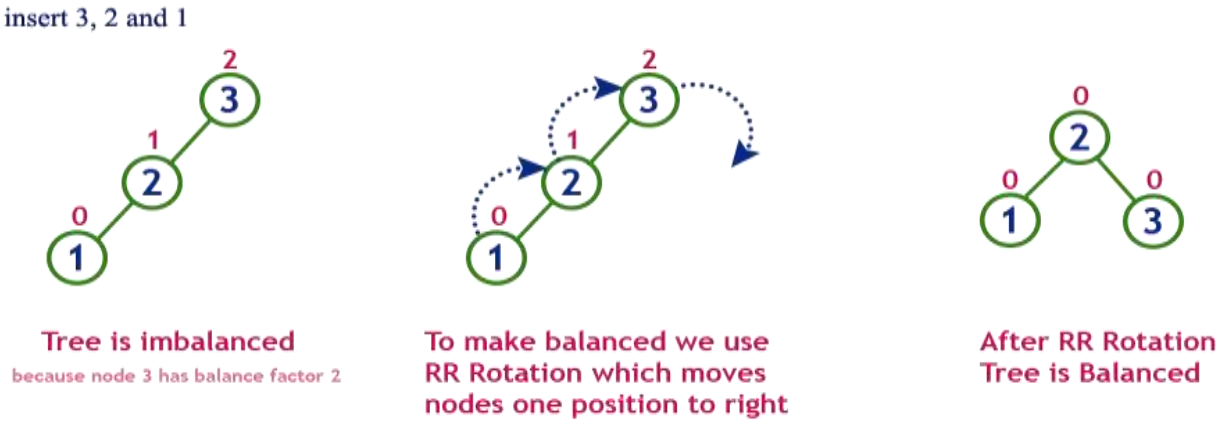
Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



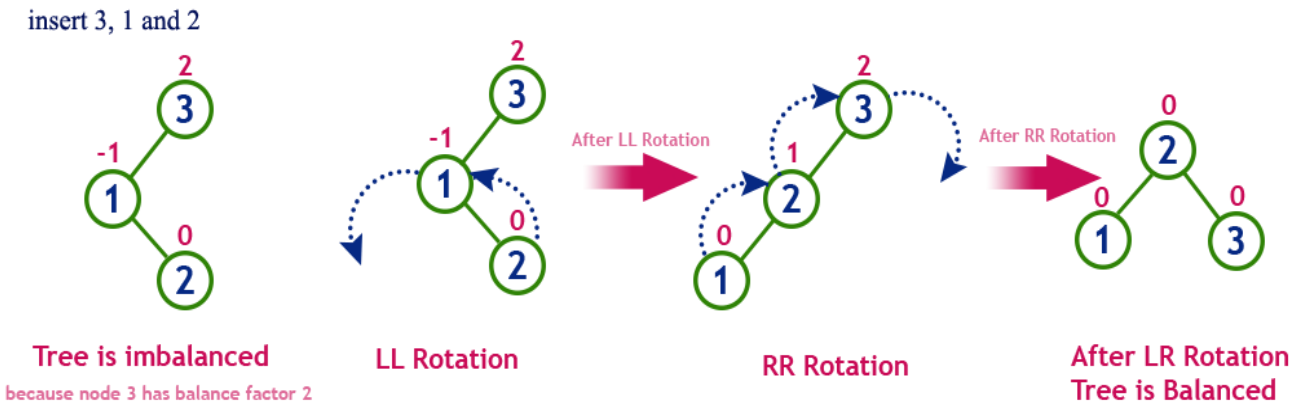
Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



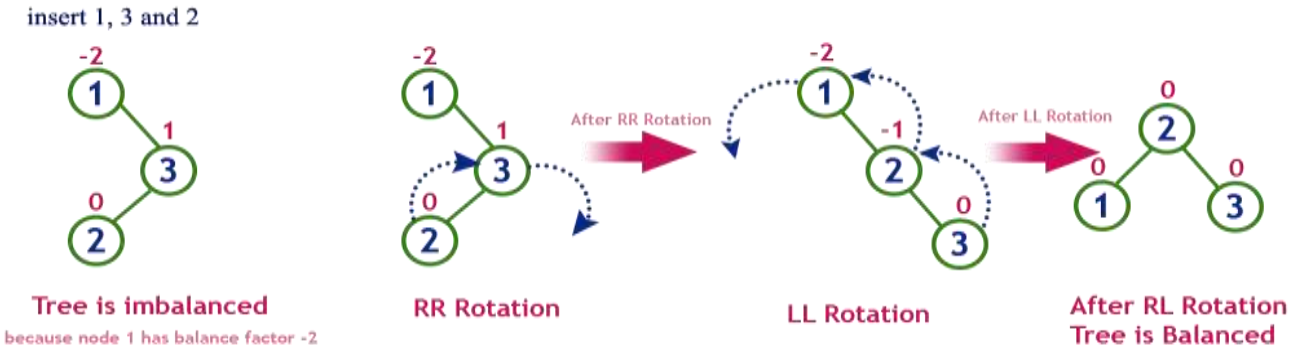
Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL 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 and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

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.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a 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 the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.

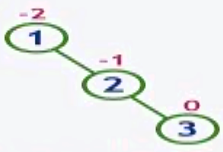
insert 1

Tree is balanced

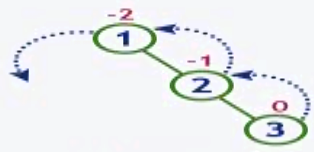
insert 2

Tree is balanced

insert 3



Tree is imbalanced

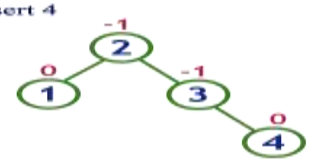


LL Rotation

After LL Rotation

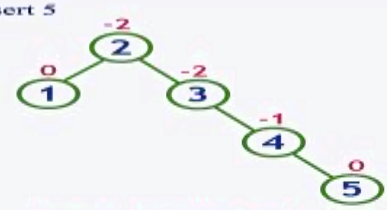
Tree is balanced

insert 4

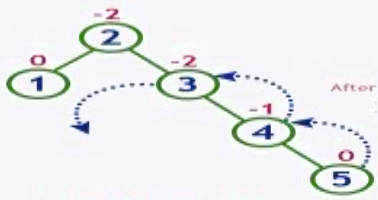


Tree is balanced

insert 5

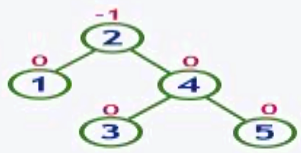


Tree is imbalanced



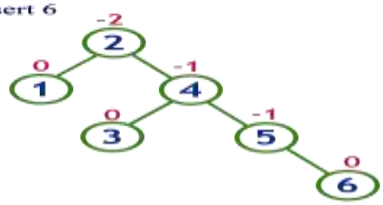
LL Rotation at 3

After LL Rotation at 3

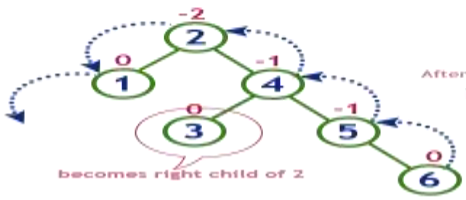


Tree is balanced

insert 6

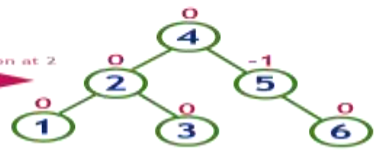


Tree is imbalanced



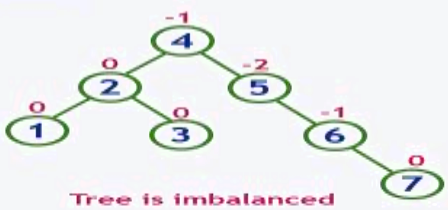
LL Rotation at 2

After LL Rotation at 2



Tree is balanced

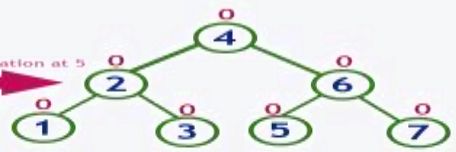
insert 7



Tree is imbalanced

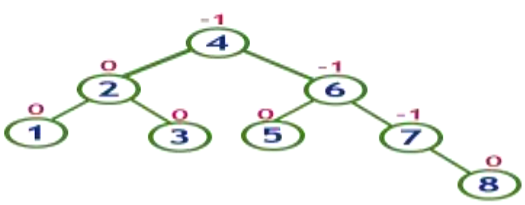
LL Rotation at 5

After LL Rotation at 5



Tree is balanced

insert 8



Tree is balanced

Deletion Operation in AVL Tree

Deletion in AVL Tree have 3 cases

1. Deleting node without any child
2. Deleting node with one child
3. Deleting node with two children

1. Deleting node without any child

Step 1: Find given node in AVL Tree by performing search operation

Step 2: Remove given node in AVL Tree by using 'delete' operator

Step 3: Check balance factor of each node in AVL tree

Step 4: If tree is not balanced perform suitable Rotation operation to make tree balanced

2. Deleting node with one child

Step 1: Find given node in AVL Tree by performing search operation

Step 2 : Make a link between its Parent and its Child

Step 3: Remove given node in AVL Tree by using 'delete' operator

Step 4: Check balance factor of each node in AVL tree

Step 5: If tree is not balanced perform suitable Rotation operation to make tree balanced

3. Deleting a node with Two Children

Step 1: Find given node in AVL Tree by performing search operation

Step 2: Find the Smallest node in its Right Sub-tree

Step 3: Swap both given node and the smallest node in its right sub-tree

Step 4: Remove given node in AVL Tree by using 'delete' operator

Step 5: Check balance factor of each node in AVL tree

Step 6: If tree is not balanced perform suitable Rotation operation to make tree balanced

Red - Black Tree Data-structure

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

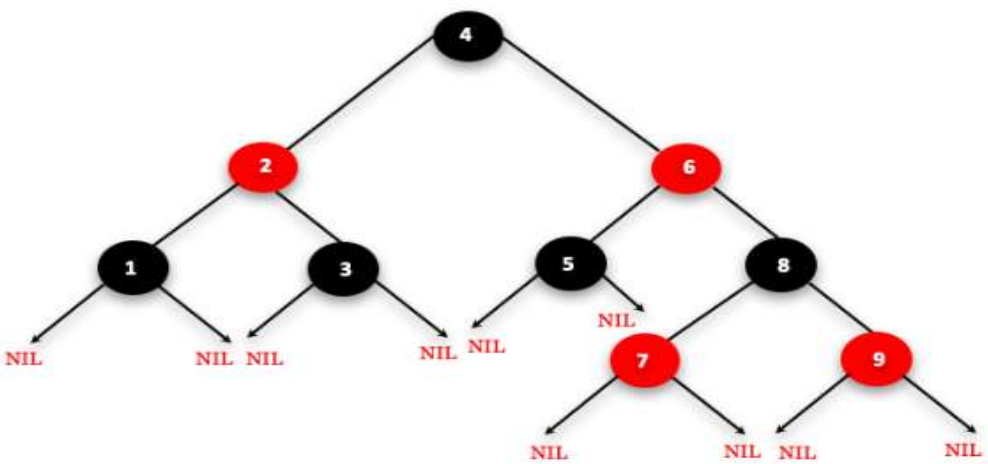
In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

- **Property #1:** Red - Black Tree must be a Binary Search Tree.
- **Property #2:** The ROOT node must be colored BLACK.
- **Property #3:** The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
- **Property #4:** In all the paths of the tree, there should be same number of BLACK colored nodes.
- **Property #5:** Every new node must be inserted with RED color.
- **Property #6:** Every leaf (e.i. NULL node) must be colored BLACK.

Example

Following is a Red-Black Tree which is created by inserting numbers from 1 to 9.



The above tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

- **1. Recolor**
- **2. Rotation**
- **3. Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **new Node** as Root node with color **Black** and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the new Node as leaf node with color Red.
- **Step 4 -** If the parent of new Node is Black then exit from the operation.
- **Step 5 -** If the parent of new Node is Red then check the color of parent node's sibling of new Node.
- **Step 6 -** If it is colored Black or NULL then make suitable Rotation and Recolor it.
- **Step 7 -** If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Below is the Example for Insertion

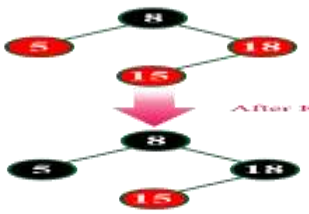
Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

Insert (8)
Tree is Empty. So insert newNode as Root node with black color.

Insert (18)
Tree is not Empty. So insert newNode with red color.

Insert (5)
Tree is not Empty. So insert newNode with red color.

Insert (15)
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is reset node.
So we use RECOLOR to make it Red Black Tree.

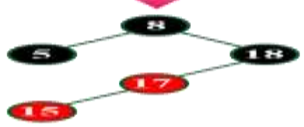
After RECOLOR

After Recolor operation, the tree is satisfying all Red Black Tree properties.

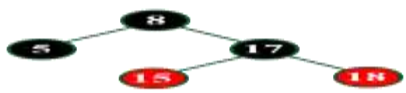
Insert (17)
Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation



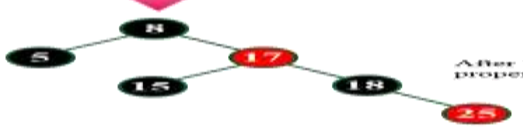
After Right Rotation & Recolor



Insert (25)
Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

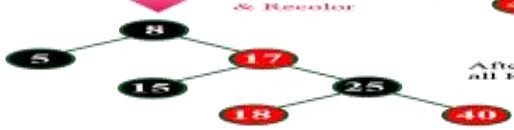


After Recolor operation, the tree is satisfying all Red Black Tree properties.

Insert (40)
Tree is not Empty. So insert newNode with red color.

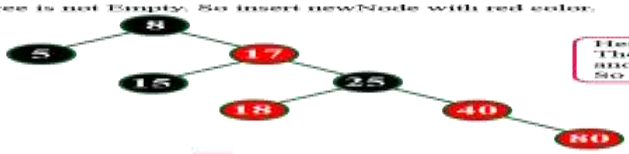
Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL.
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

After LL Rotation
& Recolor



After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

Insert (80)
Tree is not Empty. So insert newNode with red color.

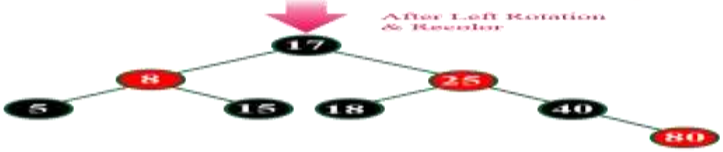


Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

After Left Rotation
& Recolor



Finally above tree is satisfying all the properties of Red Black Tree and
it is a perfect Red Black tree.

Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red-Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.