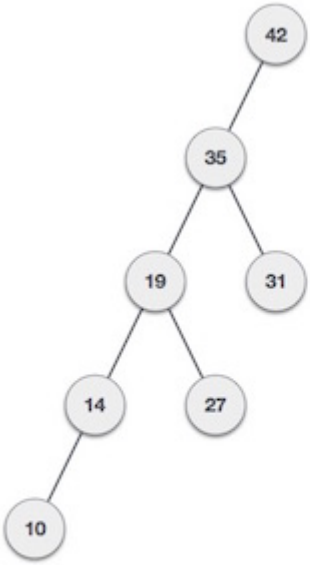
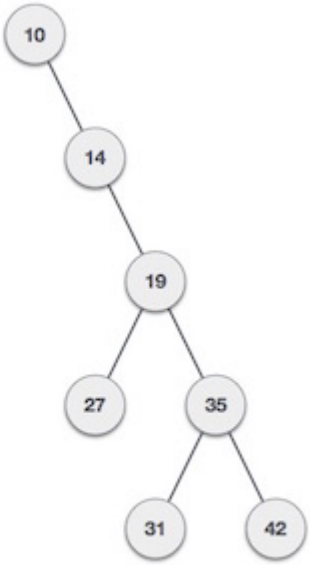
DATA STRUCTURE - AVL TREES

What if the input to binary search tree comes in sorted *ascendingordescending* manner? It will then look like this −



If input 'appears' non-increasing manner

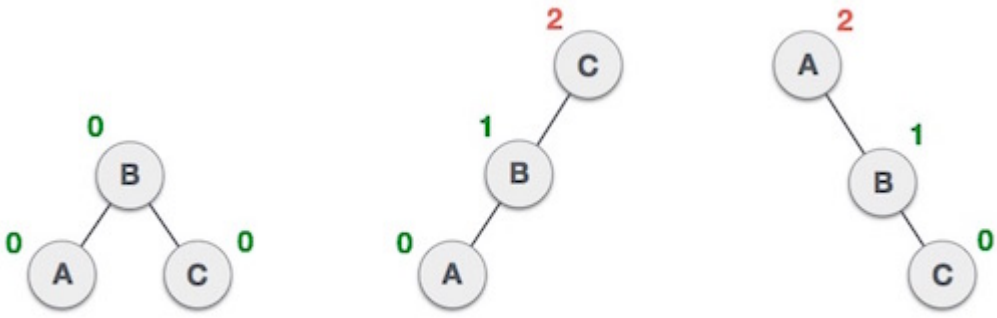


If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance closes 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 existing BST.

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

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



In second tree, the left subtree of **C** has height 2 and right subtree has height 0, so the difference is 2. In third tree, the right subtree of **A** has height 2 and left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference *balancefactor* 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

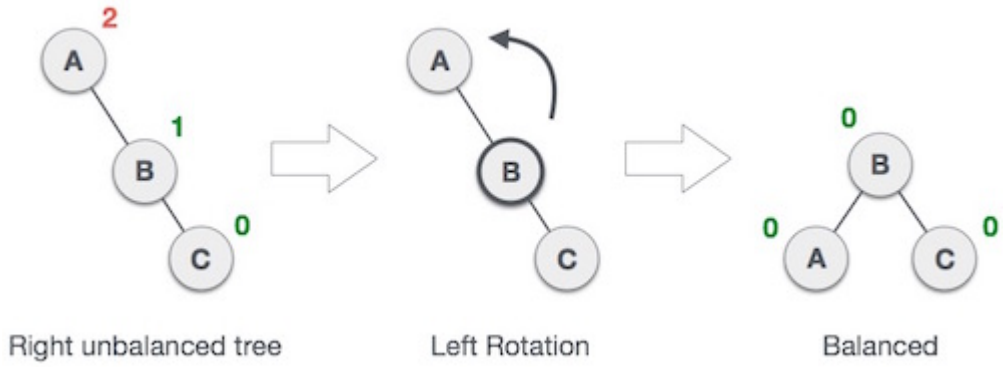
To make itself balanced, an AVL tree may perform four kinds of rotations −

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

First two rotations are single rotations and next two rotations are double rotations. Two 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

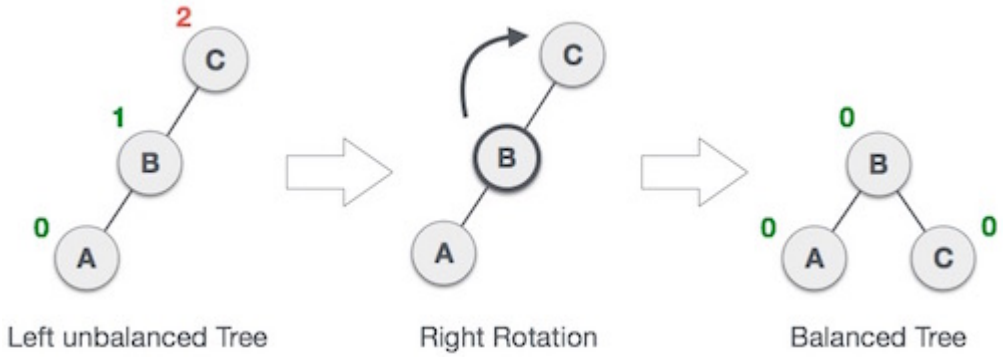
If a tree become unbalanced, when a node is inserted into the right subtree of right subtree, then we perform single left rotation −



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

Right Rotation

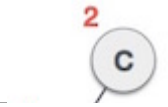
AVL tree may become unbalanced if a node is inserted in the left subtree of left subtree. The tree then needs a right rotation.

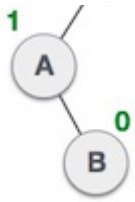
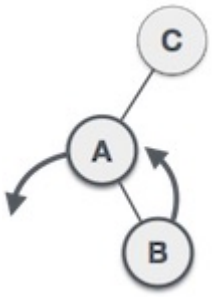
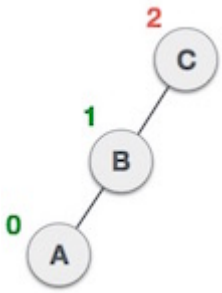
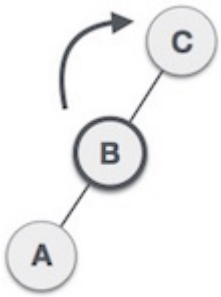
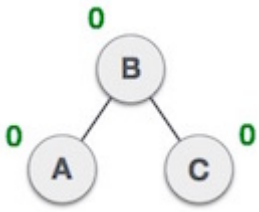


As depicted, the unbalanced node becomes right child of its left child by performing a right rotation.

Left-Right Rotation

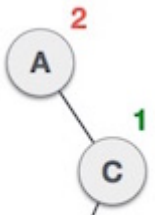
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is combination of left rotation followed by right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

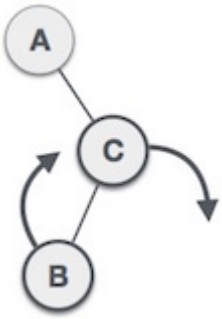
State	Action
	

	A node has been inserted into right subtree of left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform left rotation on left subtree of **C**. This makes **A**, left subtree of **B**.
	Node **C** is still unbalanced but now, it is because of left-subtree of left-subtree.
	We shall now right-rotate the tree making **B** new root node of this subtree. **C** now becomes right subtree of its own left subtree.
	The tree is now balanced.

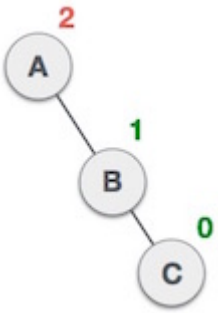
Right-Left Rotation

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

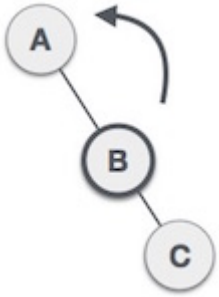
State	Action
	A node has been inserted into left subtree of right subtree. This makes **A** an unbalanced node, with balance factor 2.



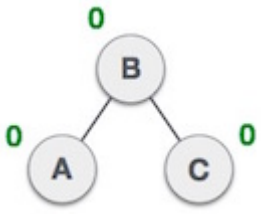
First, we perform right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes right subtree of **A**.



Node **A** is still unbalanced because of 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 left subtree of its right subtree **B**.



The tree is now balanced.