Well Balanced Trees

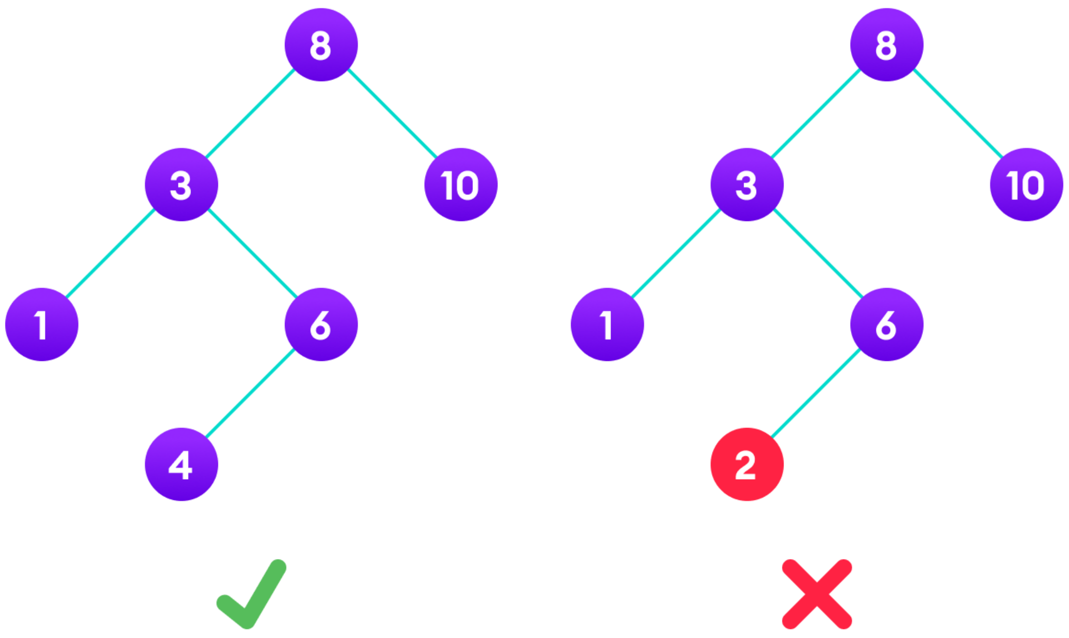
To explain what do we mean by “Well Balanced Trees”? We have taken an example of a Binary Search Tree to understand the concept.
In this section we will have a brief discussion on the topics such as binary search tree ,problems/drawbacks with binary search tree (BST) and how can a BST be improved ? And later on we’ll discuss AVL Trees.

What do we mean by Binary Search Tree?
Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties



The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

Time Complexity of Binary Search Tree:-

Where, n is the number of nodes in the tree.

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

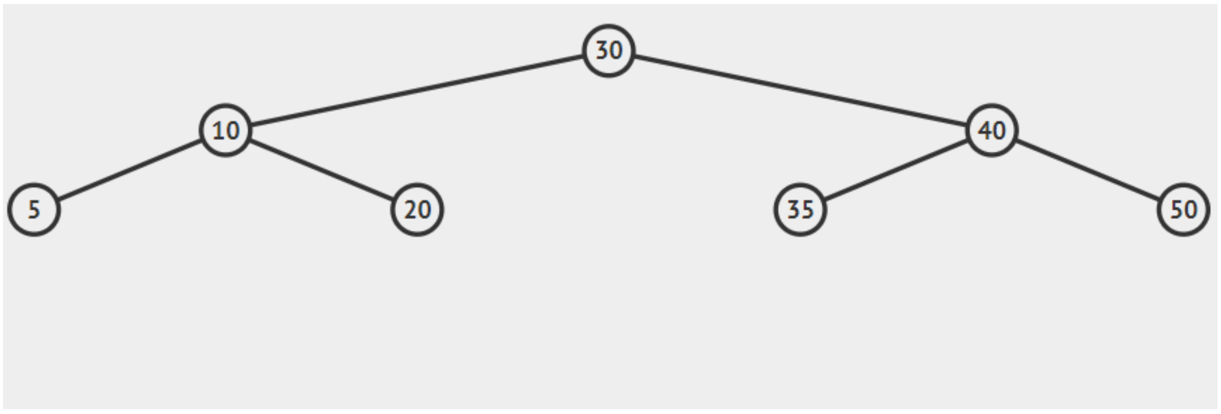
Binary Search Tree Applications

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

Problems/Drawbacks with Binary Search tree

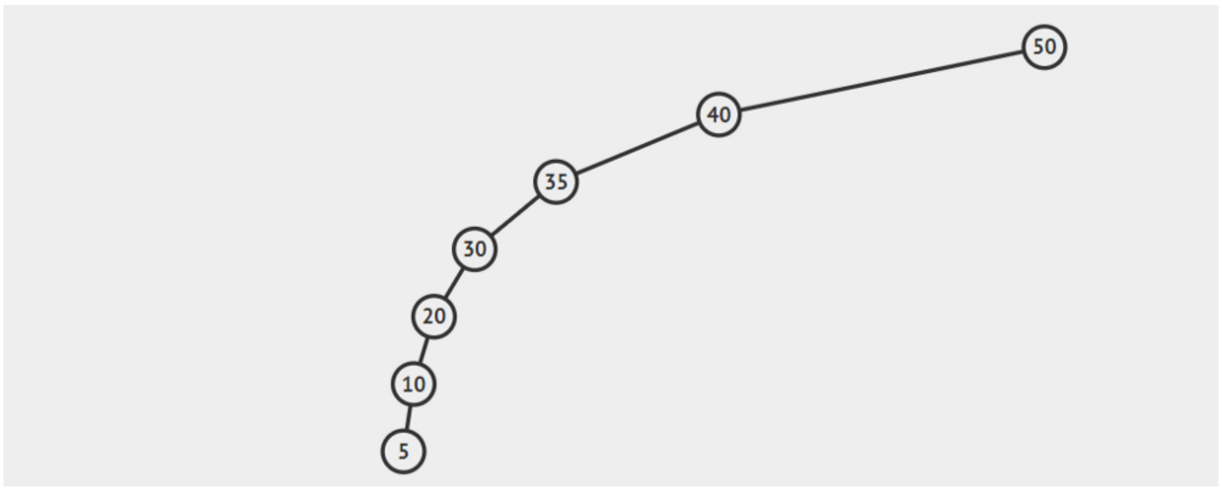
The height of a binary search tree is determined by the order in which the elements are inserted. Say, we take the same set of keys but the different order of insertion as an example to clarify this scenario .The keys are 5, 10, 20, 25, 30, 40, 50.

Case I — When the order of insertion is 30, 40, 10, 50 ,20 ,5, 35 -> height of the BST= log n .



height of the BST = log n

Case II — When the order of insertion is 50, 40, 35, 30, 20, 10, 5 ->height of the BST=n .

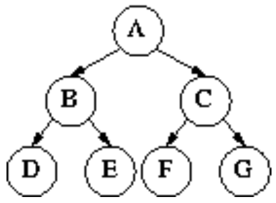


height of the BST = n

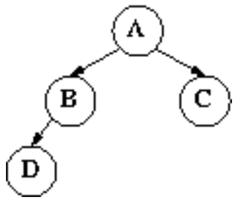
So the problem with a BST is that we cannot control the height to be in the logarithmic range (log n) and it becomes equal to linear (n), which is the same as linear search.

So how do we know if a tree is balanced?

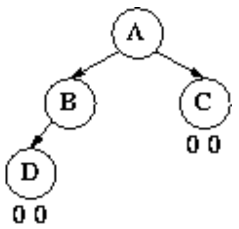
A tree is *perfectly* height-balanced if the left and right subtrees of any node are the same height. e.g.



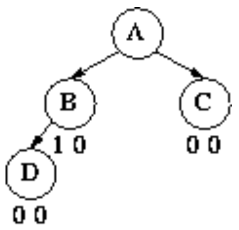
It is clear that at every level there are twice as many nodes as at the previous level, so we do indeed get H = O(logN). However, perfect height balance is very rare: it is only possible if there are exactly 2^H-1 nodes!
As a practical alternative, we use trees that are `almost' perfectly height balanced. We will say that a tree is height-balanced if the heights of the left and right subtrees of each node are within 1. The following tree fits this definition:



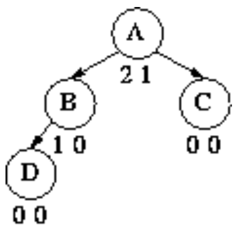
We will say this tree is height-balanced.
How can we tell if a tree is height-balanced? We have to check every node. The fastest is way is to start at the leaves and work your way up. When you reach a node, you will know the heights of its two subtrees; from that you can tell whether it is height-balanced and also compute the node's height (for use by its parent). For example, in the tree above



C and D are leaves. Their subtrees are all height 0 so C and D are both perfectly balanced. Having finished D we can compute the heights of B's subtrees.



B is not perfectly balanced, but the heights of of its subtrees differ only by 1, so B is regarded as height-balanced. Now we can compute the balance of A.



Like B, A's two subtrees differ by 1 in height. We have now looked at every node; every one is height-balanced, so the tree as a whole is considered to be height-balanced.

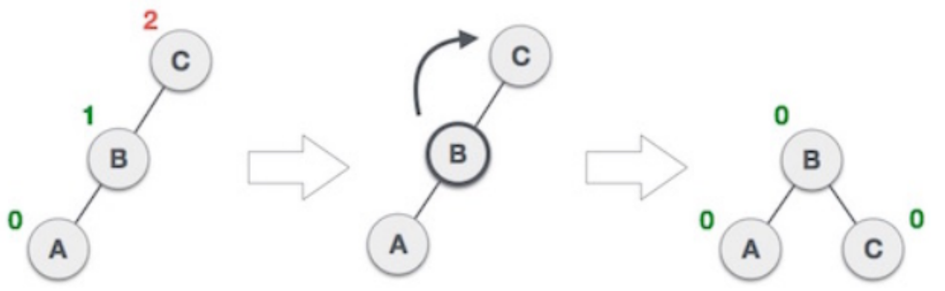
Now we are ready to understand the concept of AVL trees.

What are AVL trees?

In computer science , an AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree.
To balance the height of a BST we define a factor called as the “balance factor”.
The balance factor of a binary tree is the difference in heights of its two subtrees (hR — hL).
Balance factor = height of the left subtree-height of the right subtree
The balance factor (bf) of a height balanced binary tree may take on one of the values -1, 0, +1.
Condition for balanced node of a BST:
|bf| = |hl-hr| ≤1.

What are the different types of rotations in an AVL Tree?
To balance itself, an AVL tree may perform the following four kinds of rotations −
i. Left-Left rotation
ii. Right-Right rotation
iii. Left-Right rotation
iv. Right-Left rotation

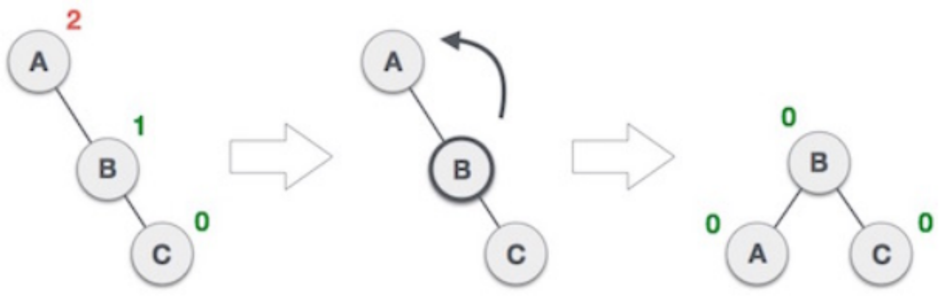
i. Left-Left rotation



Balancing a BST via LL rotation

If a tree becomes unbalanced, when a node is inserted into the left subtree of the left subtree, then we call it a ***LL-imbalance*** , so let us call this rotation also as ***LL-rotation*** .

ii. Right-Right rotation

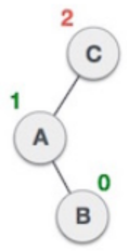
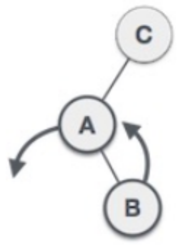
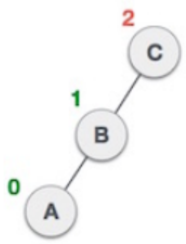
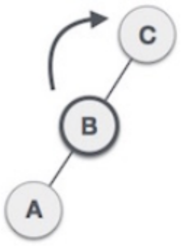
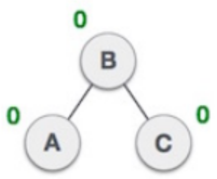


Balancing a BST via RR rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we call it a ***RR-imbalance*** , so let us call this rotation also as ***RR-rotation*** .

iii. Left-Right rotation

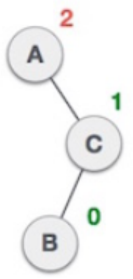
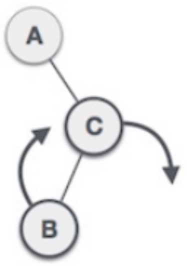
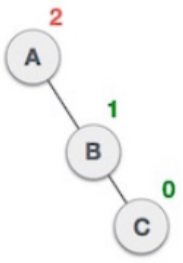
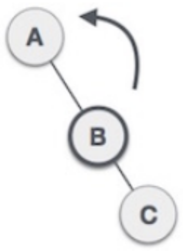
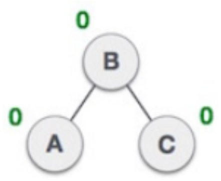
A left-right rotation is a combination of left rotation followed by right rotation.

	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.

Balancing a BST via LR rotation

iv. Right-Left rotation

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

	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.

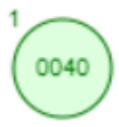
Balancing a BST via RL rotation

6. How to create an AVL tree.

Since we now have a clear understanding of the basic fundamentals, let's go ahead and create our own AVL tree from scratch.

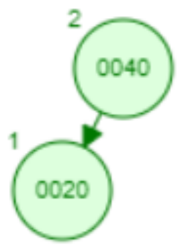
Say, we have to insert keys ***40, 20, 10, 25, 30, 22, 50*** in the order as mentioned.

Step-1: Insert 40



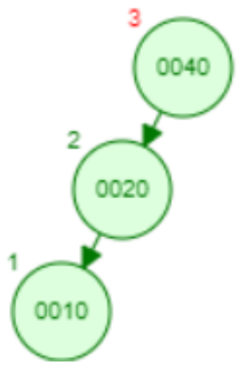
Step-1: Insert 40

Step-2 : Insert 20



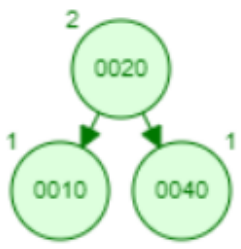
Step-2 : Insert 20

Step 3 : Insert 10



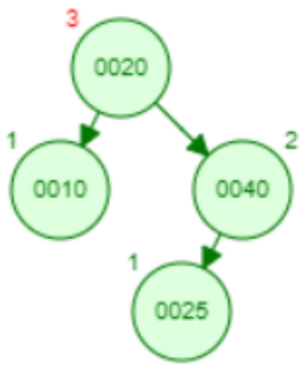
Step 3(A): Insert 10

As soon as we insert 10, the tree becomes imbalanced due to LL-insertion, therefore perform a LL rotation about the key value 40.



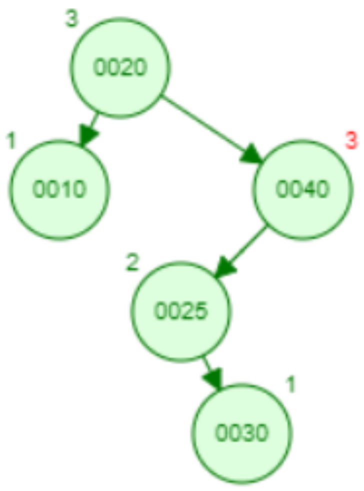
Step 3(B): Balanced AVL after inserting 10

Step 4 : Insert 25



Step 4 : Insert 25

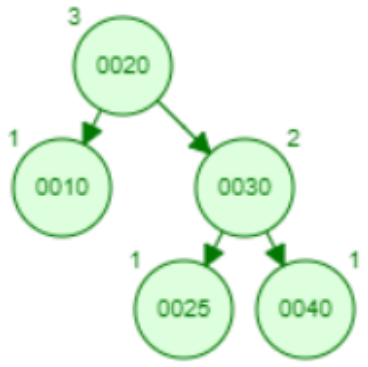
Step 5 : Insert 30



Step 5(A) : Insert 30

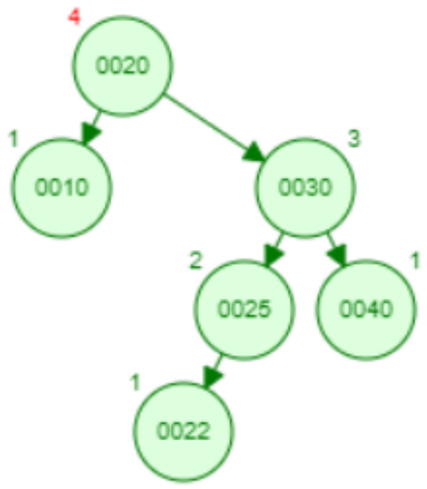
On inserting 30, keys 40 as well as 20 becomes imbalanced.

Whenever 2 nodes become imbalanced, we start form the node which we inserted and go upwards towards its ancestors, the first imbalanced node that we encounter has to be balanced first.



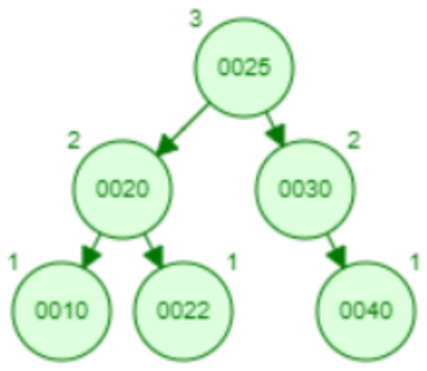
Step 5(B) : Balanced AVL after inserting 30

Step 6 : Insert 22



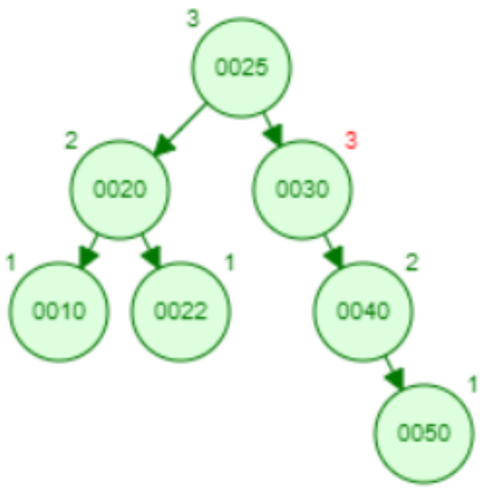
Step 6(A) : Insert 22

The root node becomes imbalanced, due to insertion in RLL, so do we have to perform a RLL rotation? Nope. We always consider the 1st two of all the notations, therefore we perform RL rotation.



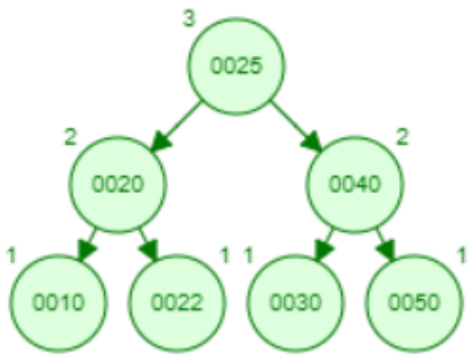
Step 6(B) : Balanced AVL after inserting 22

Step 7 : Insert 50



Step 7(A) : Insert 50

This imbalanced tree can be balanced by an RR rotation about the key value 30.



Step 7(B) : Balanced AVL after inserting 50

Hence, the AVL tree is perfectly balanced. That is all we have to discuss about “Well Balanced Trees”. The rotation operation itself is very fast and simple, and is an excellent illustration of the usefulness in balancing the tree. So here we conclude our discussion. Thank you for reading.