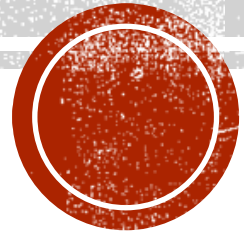
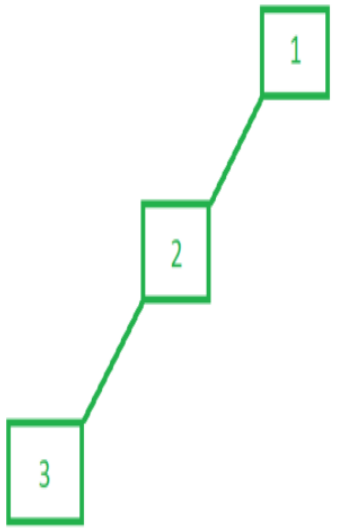
DATA STRUCTURES AND ALGORITHMS

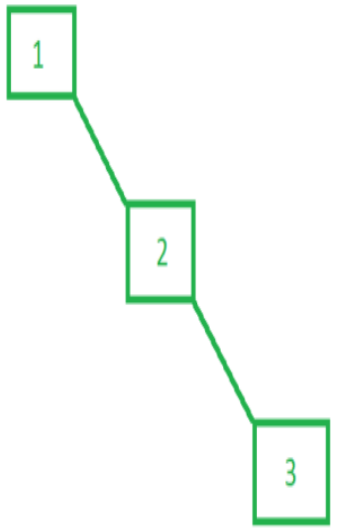


PROBLEM WITH BST

- Typically, a binary search tree will support insertion, deletion, and search operations.
- The cost of each operation depends upon the height of the tree – in the worst case, an operation will need to traverse all the nodes on the path from the root to the deepest leaf.
- A problem starts to emerge here if our tree is heavily skewed.



LEFT SKEWED



RIGHT SKEWED

SELF BALANCING TREES

- AVL Trees
- Red Black Trees
- Splay Trees

APPLICATIONS OF SELF BALANCED TREES

- Most in-memory sets and dictionaries are stored using AVL trees.
- Database applications, where insertions and deletions are less common but frequent data lookups are necessary, also frequently employ AVL trees.
- In addition to database applications, it is employed in other applications that call for better searching.
- A balanced binary search tree called an AVL tree uses rotation to keep things balanced.
- It may be used in games with plotlines as well.
- It is mostly utilized in business sectors where it is necessary to keep records on the employees that work there and their shift changes.

AVL TREES

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as balanced binary search tree
- In AVL trees, each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

BALANCE FACTOR

- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = Height of Left Subtree - Height of Right Subtree

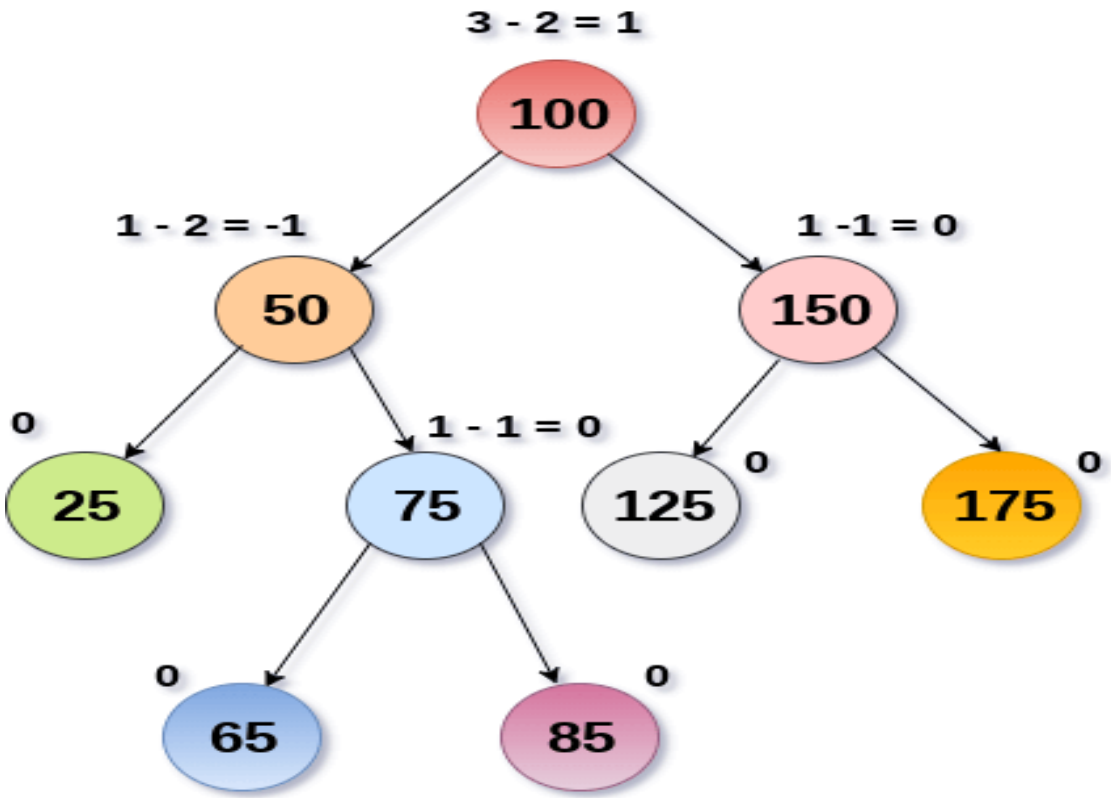
or

Balance Factor = Height of Right Subtree - Height of Left Subtree

BALANCE FACTOR

- In AVL trees, **Balance Factor** must be at most one.
- Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again.
- If the value of balance factor is 0, it means that height of both subtrees are equal.
- If the value of balance factor is -1 or +1 it means that its left or right subtree height is not equal. However, the tree is in balanced condition.
- Tree is not balanced if the value exceeds +1 or -1

AVL TREE



AVL Tree

HOW TO BALANCE THE TREE?

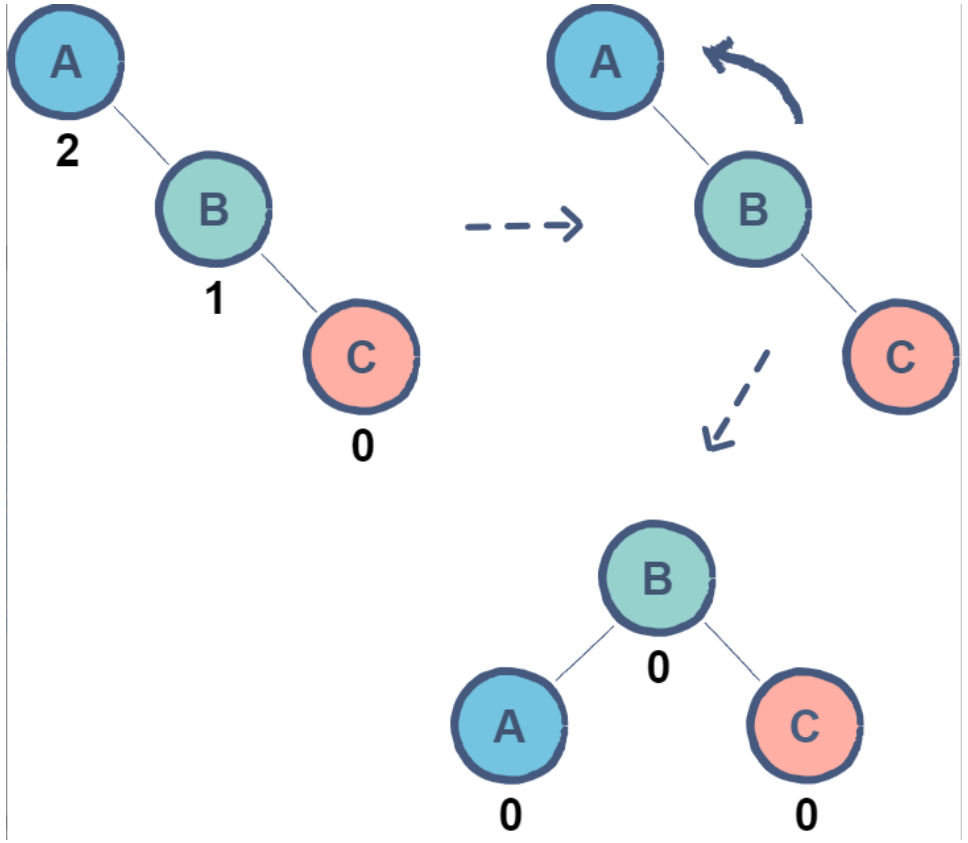
- If the balance factor is not 0, +1, -1 then the tree is needed to be balanced.
- Basically, balance factor is checked each time an item is inserted or deleted from the tree
- To make a balanced BST/ AVL tree, rotations need to be performed.
- Hence, at each insertion or deletion balance factor of the tree is checked and rotation is performed if needed

ROTATIONS IN AVL TREES

- There are following types of rotations:
  - **Single Rotation**
    - Left Rotation
    - Right Rotation
  - **Double Rotation**
    - Left Right Rotation
    - Right Left Rotation

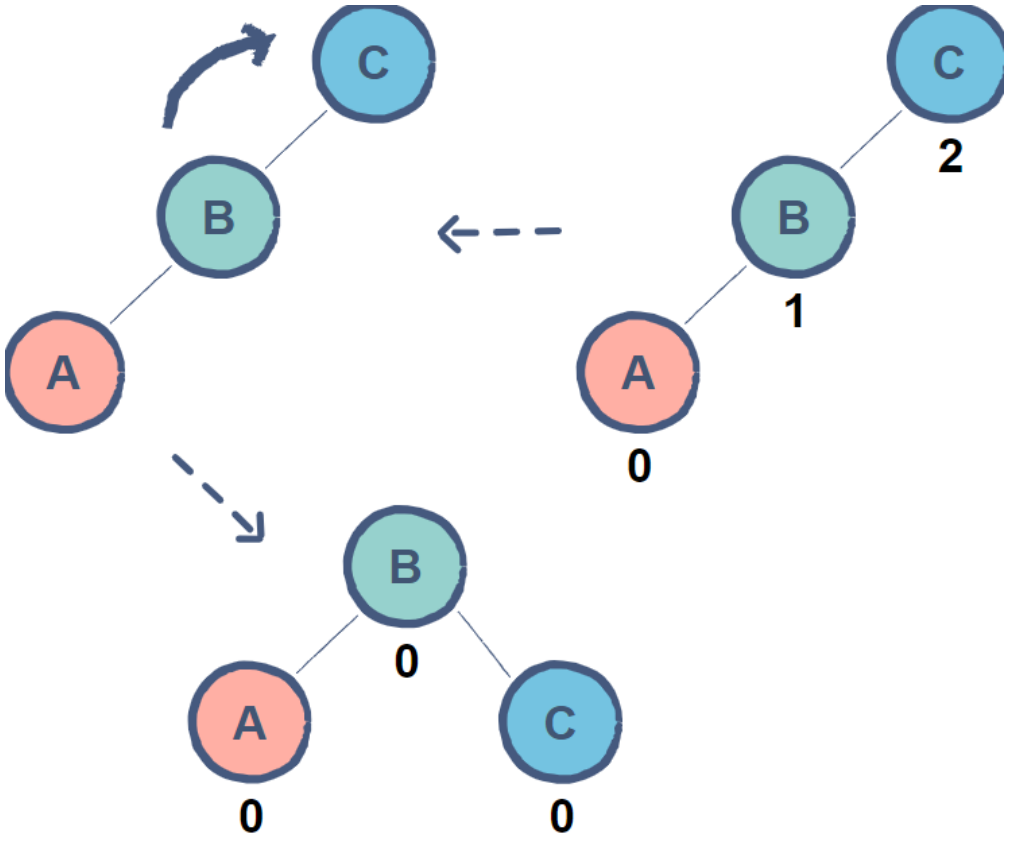
LEFT ROTATION

- When RR imbalance occurs, then left rotation is performed
- A single rotation applied when a node is inserted in the right subtree of a right subtree.
- In the given example, node A has a balance factor of 2 after the insertion of node C.
- By rotating the tree left, node B becomes the root resulting in a balanced tree.



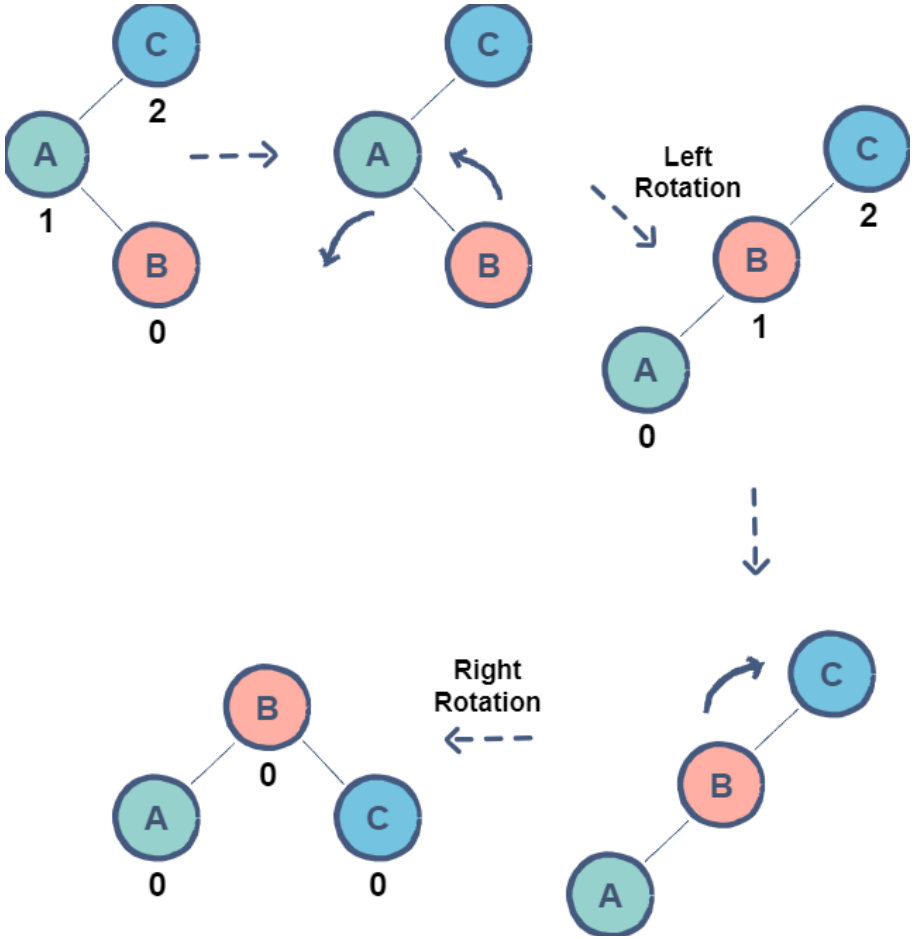
RIGHT ROTATION

- When LL imbalance occurs, then right rotation is performed
- A single rotation applied when a node is inserted in the left subtree of a left subtree.
- In the given example, node A has a balance factor of 2 after the insertion of node C.
- By rotating the tree left, node B becomes the root resulting in a balanced tree.



LEFT RIGHT ROTATION

- A double rotation in which a left rotation is followed by a right rotation.
- Convert the LR rotation to LL imbalance using **left rotation**
- Then solve the LL imbalance using **right rotation**
- In the given example, node B is causing an imbalance resulting in node C to have a balance factor of 2. As node B is inserted in the right subtree of node A, a left rotation needs to be applied.
- Now, all we have to do is apply the right rotation as shown before to achieve a balanced tree.



RIGHT LEFT ROTATION

- A double rotation in which a right rotation is followed by a left rotation.
- Convert the RL rotation to RR imbalance using **right rotation**
- Then solve the RR imbalance using **left rotation**
- In the given example, node B is causing an imbalance resulting in node A to have a balance factor of 2. As node B is inserted in the left subtree of node C, a right rotation needs to be applied.
- Now, by applying the left rotation as shown before, we can achieve a balanced tree.