Data Structures Lab 9

Course: Data Structures (CL2001) **Semester:** Fall 2023
Instructor: Shafique Rehman **T.A:** N/A

Note:

- Lab manual cover following below Stack and Queue topics
  {Self balancing binary trees, AVL trees}
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.
- Don't just blatantly copy the same code make changes to it accordingly

---

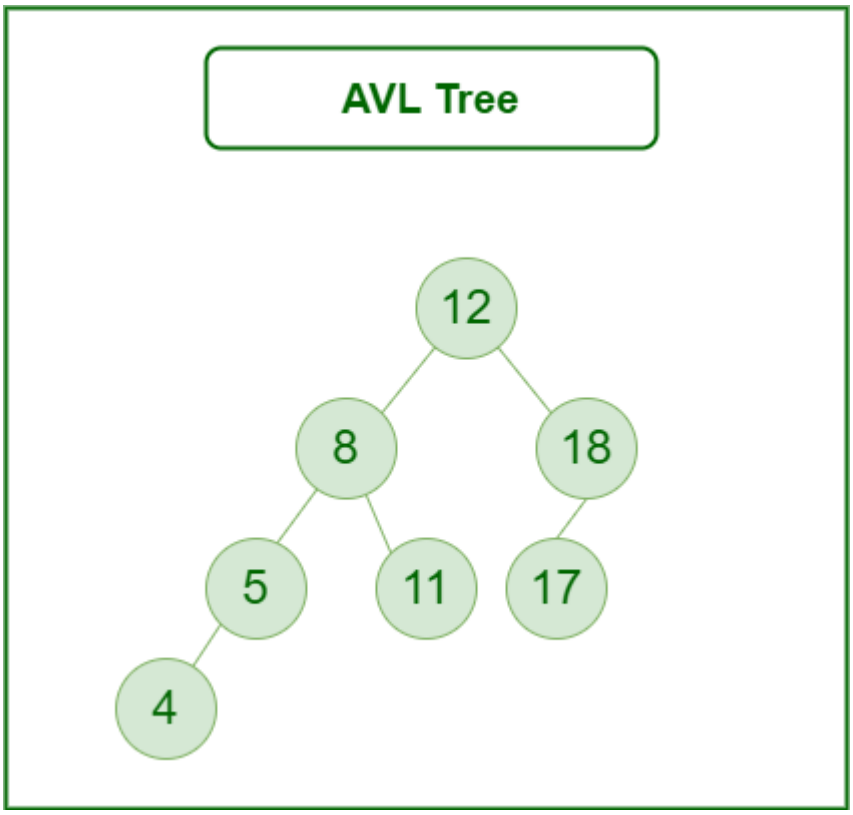
AVL Tree Data Structure

*An **AVL tree** defined as a self-balancing **Binary Search Tree** (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.*

The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node.

The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper "An algorithm for the organization of information".

Example of AVL Trees:



AVL tree

The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

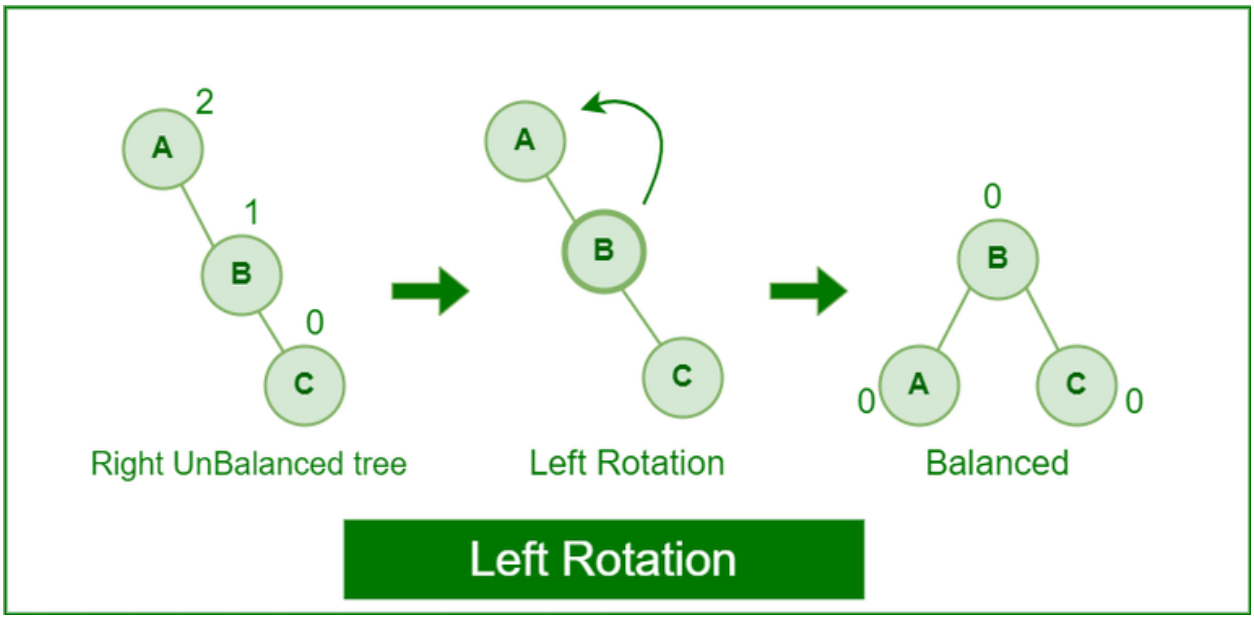
Operations on an AVL Tree:

- Insertion
- Deletion
- Searching [It is similar to performing a search in BST]

Rotating the subtrees in an AVL Tree while inserting:

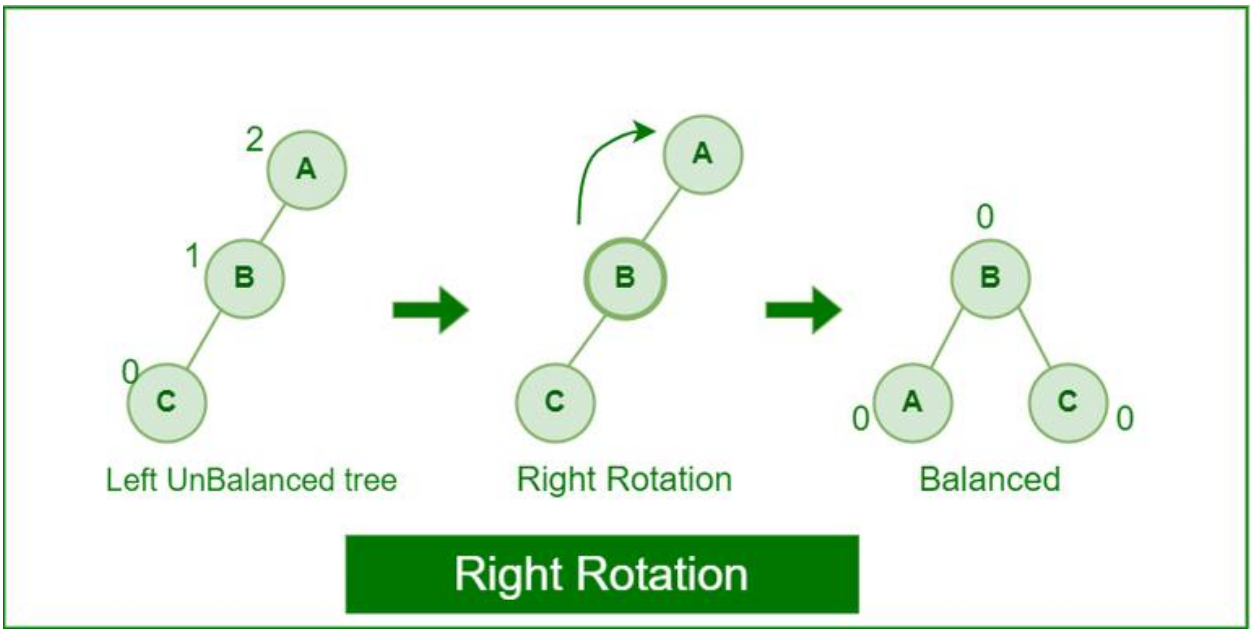
An AVL tree may rotate in one of the following four ways to keep itself balanced:

Left Rotation:
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.



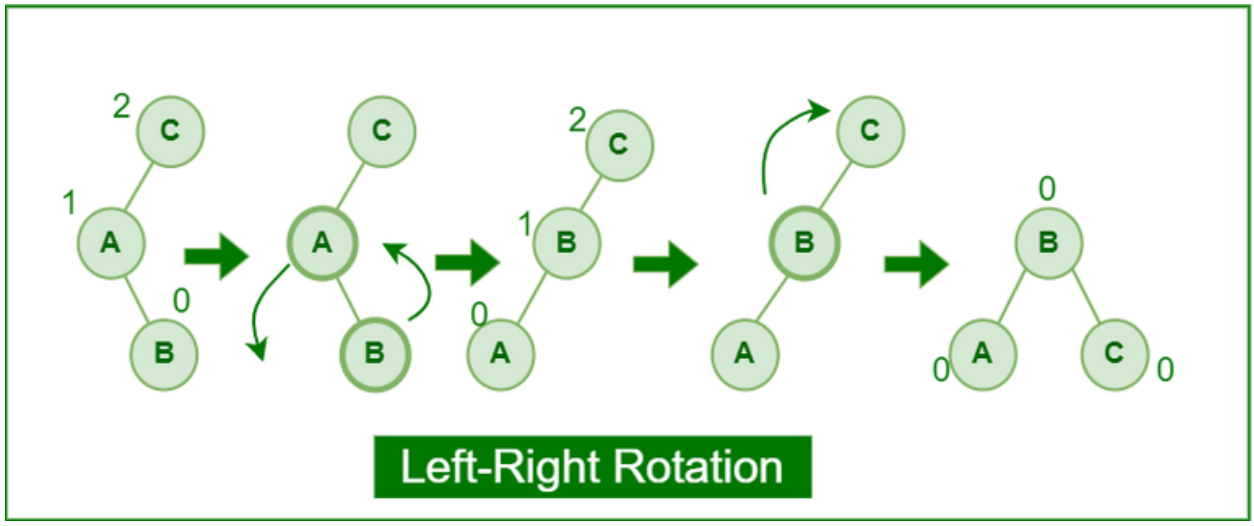
Left-Rotation in AVL tree

Right 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.



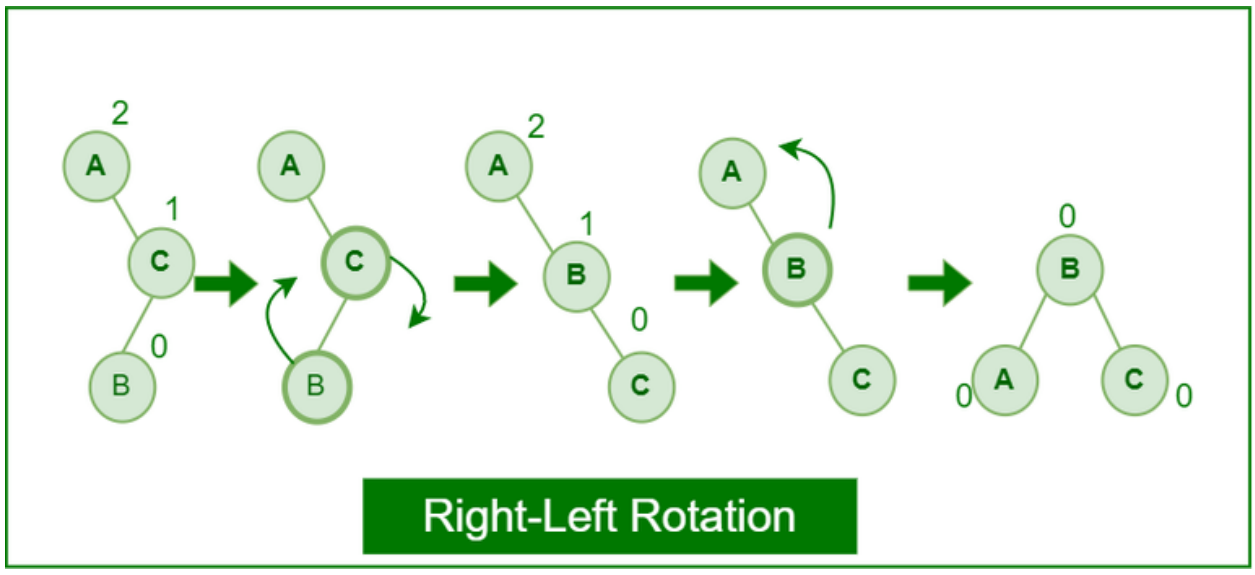
Right Rotation in AVL tree

Left-Right Rotation:
A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



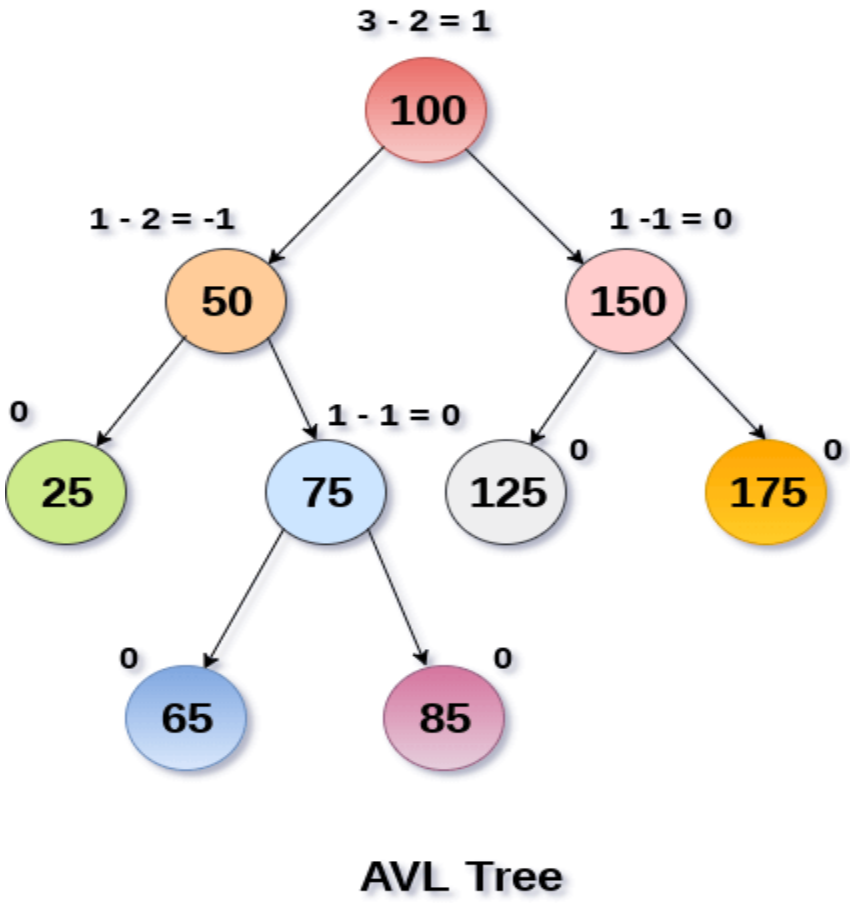
Left-Right Rotation in AVL tree

Right-Left Rotation:
A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



Right-Left Rotation in AVL tree

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Applications of AVL Tree:

1. It is used to index huge records in a database and also to efficiently search in that.
2. For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
3. Database applications, where insertions and deletions are less common but frequent data lookups are necessary
4. Software that needs optimized search.
5. It is applied in corporate areas and storyline games.

Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed log(N), where, N is the total number of nodes in the tree.

Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

Steps to follow for 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)

Case for Deletion

1. Perform the normal BST deletion.
2. The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3. Get the balance factor (left subtree height – right subtree height) of the current node.
4. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

Insertion Algorithms Steps:

Algorithm to insert a newNode
A newNode is always inserted as a leaf node with balance factor equal to 0.

1. Insert a node: Go to the appropriate leaf node to insert a newNode using the following recursive steps.
   Compare newKey with rootKey of the current tree.
   If newKey < rootKey, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
   Else if newKey > rootKey, call insertion algorithm on the right subtree of current node until the leaf node is reached.
   Else, return leafNode.

2. Compare `leafKey` obtained from the above steps with `newKey`:

a. If newKey < leafKey, make newNode as the leftChild of leafNode.
   Else, make newNode as rightChild of leafNode.

3. Update balanceFactor of the nodes.

4. If the nodes are unbalanced, then rebalance the node.
    a. If balanceFactor > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation
        a. If newNodeKey < leftChildKey do right rotation.
        b. Else, do left-right rotation.
    b. If balanceFactor < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation
        a. If newNodeKey > rightChildKey do left rotation.
        b. Else, do right-left rotation

Tasks:
1. Implement the following insertions in the AVL tree (1,2,3,4,5,6,7)
2. Delete value 3 from the tree and balance it.
3. Do a pre-order, in order and post-order traversal of the tree before deletion and after deletion.
4. Search for any value in the tree if it is present print, it with its index (key) value otherwise inserts it into the tree and balances it with the appropriate rotations.
5. Find the kth smallest and largest value in the AVL tree and print its key also print both the left side and right side height of the tree starting from root.