**AVL Tree**

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

**A self-balancing tree is a binary search tree that balances the height after insertion and deletion according to some balancing rules.**

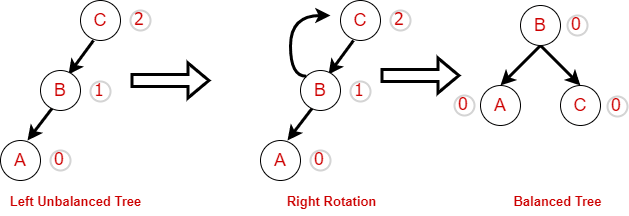
The balance factor of node N is height(left(N))-height(right(N)).  **In an AVL Tree, the balance factor of a node could be only one of 1, 0, or -1 values.**

The AVL Tree checks the balance factor of its nodes after the insertion or deletion of a node. If the balance factor of a node is greater than one or less than -1, the tree rebalances itself.

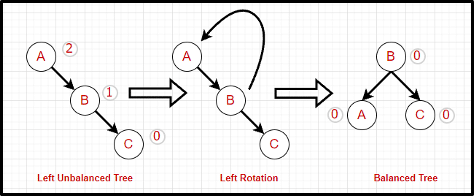
There are four operations to rebalance a tree:

* right rotation and
* left rotation.
* Left-Right Rotation.
* Right-Left Rotation

### Right Rotation



Left Rotation



#### **Left-Right Rotation**

A picture containing text, pool ball, sport

Description automatically generated A picture containing text, pool ball, sport, pool table

Description automatically generated



#### **Right-Left Rotation**

A picture containing text, pool ball, sport, pool table

Description automatically generated A picture containing text, pool ball, sport, pool table

Description automatically generated



**Implementation:**

class AVLNode

{

AVLNode left, right;

int data;

int height;

public AVLNode()

{

left = null;

right = null;

data = 0;

height = 0;

}

public AVLNode(int n)

{

left = null;

right = null;

data = n;

height = 0;

}

}

*// AVL Tree Class*

class AVLTree

{

private AVLNode root;

public AVLTree()

{

root = null;

}

private int height(AVLNode avlNode)

{

return avlNode == null ? -1 : avlNode.height;

}

private int max(int lHeight, int rHeight)

{

return lHeight > rHeight ? lHeight : rHeight;

}

public void insert(int data)

{

root = insert(data, root);

}

private AVLNode insert(int data, AVLNode avlNode)

{

if (avlNode == null)

avlNode = new AVLNode(data);

else if (data < avlNode.data)

{

avlNode.left = insert(data, avlNode.left);

if (height(avlNode.left) - height(avlNode.right) == 2)

if (data < avlNode.left.data)

avlNode = leftRotation(avlNode);

else

avlNode = leftRightRotation(avlNode);

} else if (data > avlNode.data)

{

avlNode.right = insert(data, avlNode.right);

if (height(avlNode.right) - height(avlNode.left) == 2)

if (data > avlNode.right.data)

avlNode = rightRotation(avlNode);

else

avlNode = rightLeftRotation(avlNode);

} else

;

avlNode.height = max(height(avlNode.left), height(avlNode.right)) + 1;

return avlNode;

}

private AVLNode leftRotation(AVLNode avlNode)

{

AVLNode k1 = avlNode.left;

avlNode.left = k1.right;

k1.right = avlNode;

avlNode.height = max(height(avlNode.left), height(avlNode.right)) + 1;

k1.height = max(height(k1.left), avlNode.height) + 1;

return k1;

}

private AVLNode rightRotation(AVLNode avlNode)

{

AVLNode node = avlNode.right;

avlNode.right = node.left;

node.left = avlNode;

avlNode.height = max(height(avlNode.left), height(avlNode.right)) + 1;

node.height = max(height(node.right), avlNode.height) + 1;

return node;

}

private AVLNode leftRightRotation(AVLNode avlNode)

{

avlNode.left = rightRotation(avlNode.left);

return leftRotation(avlNode);

}

private AVLNode rightLeftRotation(AVLNode avlNode)

{

avlNode.right = leftRotation(avlNode.right);

return rightRotation(avlNode);

}

public int countNodes()

{

return countNodes(root);

}

private int countNodes(AVLNode avlNode)

{

if (avlNode == null)

return 0;

else

{

int l = 1;

l += countNodes(avlNode.left);

l += countNodes(avlNode.right);

return l;

}

}

public boolean search(int data)

{

return search(root, data);

}

private boolean search(AVLNode avlNode, int data)

{

boolean found = false;

while ((avlNode != null) && !found)

{

int rval = avlNode.data;

if (data < rval)

avlNode = avlNode.left;

else if (data > rval)

avlNode = avlNode.right;

else

{

found = true;

break;

}

found = search(avlNode, data);

}

return found;

}

public void inorder() {

inorder(root);

}

private void inorder(AVLNode avlNode)

{

if (avlNode != null)

{

inorder(avlNode.left);

System.out.print(avlNode.data + " ");

inorder(avlNode.right);

}

}

}

import java.util.Scanner;

public class AVLTreeHelper

{

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

AVLTree avlTree = new AVLTree();

char ch;

do

{

System.out.println("\nAVLTree Operations\n");

System.out.println("1. insert ");

System.out.println("2. search");

System.out.println("3. count nodes");

int choice = scanner.nextInt();

switch (choice)

{

case 1:

System.out.println("Enter integer element to insert");

avlTree.insert(scanner.nextInt());

break;

case 2:

System.out.println("Enter integer element to search");

System.out.println("Search result : " + avlTree.search(scanner.nextInt()));

break;

case 3:

System.out.println("Nodes = " + avlTree.countNodes());

break;

default:

System.out.println("Wrong Entry \n ");

break;

}

System.out.print("\nIn order : ");

avlTree.inorder();

System.out.println("\nDo you want to continue (Type y or n) \n");

ch = scanner.next().charAt(0);

} while (ch == 'Y' || ch == 'y');

}

}