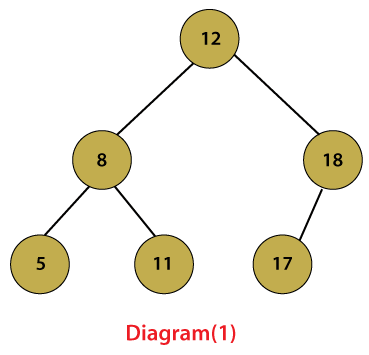
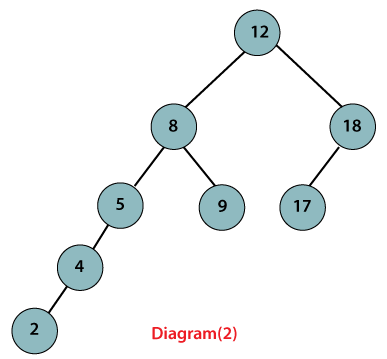
AVL TREE

the AVL tree is another self-balancing BST(Binary Search Tree) in [Java](https://www.javatpoint.com/java-tutorial). In the AVL tree, the difference between heights of the right and left subtree doesn't exceed one for all nodes. It takes O(h) time to perform the search, max, min, insert, and delete BST operations. Here, the h is the height of the [Binary Search Tree](https://www.javatpoint.com/binary-search-tree).

Let's take an example of an [AVL tree](https://www.javatpoint.com/avl-tree) and a tree that is not AVL to understand the difference between both of them,

Diagram(1) is an example of the AVL tree because the difference between the heights of the left and right sub-tree is 1. Diagram (2) is not an AVL tree because the difference between the heights of the left and right subtree is not 1.

**Algorithm**

Let's understand the algorithm of inserting a node in the AVL Tree:

Suppose the newNode is the newly inserted node in the AVL Tree.

1. We will insert the newNode in the AVL Tree by performing the standard BST insert operation.
2. We will go through the AVL Tree from the newNode and check for that node that is unbalanced. Suppose unBalNode is the first unbalanced node, childNode is the child node of the unBalNode that comes on the path from newNode to unBalNode, and newNode is the grandchild of the unBalNode that comes from the path from newNode to unBalNode.
3. After that, we perform the appropriate rotations on the subtree rooted with unBalNode to re-balance the AVL Tree. These are the following four cases which we need to be handled.
   1. When childNode is the left child of the unBalNode and newNode is the left child of the childNode. This case is known as **Left Left Case**.
   2. When the childNode is the left child of the unBalNode, and the newNode is the right child of the childNode. This case is known as **Left Right Case**.
   3. When the childNode is the right child of the unBalNode, and the newNode is the right child of the childNode. This case is known as **Right Right Case**.
   4. When the childNode is the right child of the unBalNode, and the newNode is the left child of the childNode. This case is known as **Right Left Case**.
4. In all the cases which we have discussed above, the subtree rooted with the unBalNode is only needed to be re-balanced. After that, the complete tree will be balanced and the same as it was before insertion.

Pseudocode for AVL Balanced Binary Search Tree Methods

CODE

import java.util.Scanner;

// create Node class to design the structure of the AVL Tree Node

class Node

{

int element;

int h; //for height

Node leftChild;

Node rightChild;

//default constructor to create null node

public Node()

{

leftChild = null;

rightChild = null;

element = 0;

h = 0;

}

// parameterized constructor

public Node(int element)

{

leftChild = null;

rightChild = null;

this.element = element;

h = 0;

}

}

// create class ConstructAVLTree for constructing AVL Tree

class ConstructAVLTree

{

private Node rootNode;

//Constructor to set null value to the rootNode

public ConstructAVLTree()

{

rootNode = null;

}

//create removeAll() method to make AVL Tree empty

public void removeAll()

{

rootNode = null;

}

// create checkEmpty() method to check whether the AVL Tree is empty or not

public boolean checkEmpty()

{

if(rootNode == null)

return true;

else

return false;

}

// create insertElement() to insert element to to the AVL Tree

public void insertElement(int element)

{

rootNode = insertElement(element, rootNode);

}

//create getHeight() method to get the height of the AVL Tree

private int getHeight(Node node )

{

return node == null ? -1 : node.h;

}

//create maxNode() method to get the maximum height from left and right node

private int getMaxHeight(int leftNodeHeight, int rightNodeHeight)

{

return leftNodeHeight > rightNodeHeight ? leftNodeHeight : rightNodeHeight;

}

//create insertElement() method to insert data in the AVL Tree recursively

private Node insertElement(int element, Node node)

{

//check whether the node is null or not

if (node == null)

node = new Node(element);

//insert a node in case when the given element is lesser than the element of the root node

else if (element < node.element)

{

node.leftChild = insertElement( element, node.leftChild );

if( getHeight( node.leftChild ) - getHeight( node.rightChild ) == 2 )

if( element < node.leftChild.element )

node = rotateWithLeftChild( node );

else

node = doubleWithLeftChild( node );

}

else if( element > node.element )

{

node.rightChild = insertElement( element, node.rightChild );

if( getHeight( node.rightChild ) - getHeight( node.leftChild ) == 2 )

if( element > node.rightChild.element)

node = rotateWithRightChild( node );

else

node = doubleWithRightChild( node );

}

else

; // if the element is already present in the tree, we will do nothing

node.h = getMaxHeight( getHeight( node.leftChild ), getHeight( node.rightChild ) ) + 1;

return node;

}

// creating rotateWithLeftChild() method to perform rotation of binary tree node with left child

private Node rotateWithLeftChild(Node node2)

{

Node node1 = node2.leftChild;

node2.leftChild = node1.rightChild;

node1.rightChild = node2;

node2.h = getMaxHeight( getHeight( node2.leftChild ), getHeight( node2.rightChild ) ) + 1;

node1.h = getMaxHeight( getHeight( node1.leftChild ), node2.h ) + 1;

return node1;

}

// creating rotateWithRightChild() method to perform rotation of binary tree node with right child

private Node rotateWithRightChild(Node node1)

{

Node node2 = node1.rightChild;

node1.rightChild = node2.leftChild;

node2.leftChild = node1;

node1.h = getMaxHeight( getHeight( node1.leftChild ), getHeight( node1.rightChild ) ) + 1;

node2.h = getMaxHeight( getHeight( node2.rightChild ), node1.h ) + 1;

return node2;

}

//create doubleWithLeftChild() method to perform double rotation of binary tree node. This method first rotate the left child with its right child, and after that, node3 with the new left child

private Node doubleWithLeftChild(Node node3)

{

node3.leftChild = rotateWithRightChild( node3.leftChild );

return rotateWithLeftChild( node3 );

}

//create doubleWithRightChild() method to perform double rotation of binary tree node. This method first rotate the right child with its left child and after that node1 with the new right child

private Node doubleWithRightChild(Node node1)

{

node1.rightChild = rotateWithLeftChild( node1.rightChild );

return rotateWithRightChild( node1 );

}

//create getTotalNumberOfNodes() method to get total number of nodes in the AVL Tree

public int getTotalNumberOfNodes()

{

return getTotalNumberOfNodes(rootNode);

}

private int getTotalNumberOfNodes(Node head)

{

if (head == null)

return 0;

else

{

int length = 1;

length = length + getTotalNumberOfNodes(head.leftChild);

length = length + getTotalNumberOfNodes(head.rightChild);

return length;

}

}

//create searchElement() method to find an element in the AVL Tree

public boolean searchElement(int element)

{

return searchElement(rootNode, element);

}

private boolean searchElement(Node head, int element)

{

boolean check = false;

while ((head != null) && !check)

{

int headElement = head.element;

if (element < headElement)

head = head.leftChild;

else if (element > headElement)

head = head.rightChild;

else

{

check = true;

break;

}

check = searchElement(head, element);

}

return check;

}

// create inorderTraversal() method for traversing AVL Tree in in-order form

public void inorderTraversal()

{

inorderTraversal(rootNode);

}

private void inorderTraversal(Node head)

{

if (head != null)

{

inorderTraversal(head.leftChild);

System.out.print(head.element+" ");

inorderTraversal(head.rightChild);

}

}

// create preorderTraversal() method for traversing AVL Tree in pre-order form

public void preorderTraversal()

{

preorderTraversal(rootNode);

}

private void preorderTraversal(Node head)

{

if (head != null)

{

System.out.print(head.element+" ");

preorderTraversal(head.leftChild);

preorderTraversal(head.rightChild);

}

}

// create postorderTraversal() method for traversing AVL Tree in post-order form

public void postorderTraversal()

{

postorderTraversal(rootNode);

}

private void postorderTraversal(Node head)

{

if (head != null)

{

postorderTraversal(head.leftChild);

postorderTraversal(head.rightChild);

System.out.print(head.element+" ");

}

}

}

// create AVLTree class to construct AVL Tree

public class AVLTreeExample

{

//main() method starts

public static void main(String[] args)

{

//creating Scanner class object to get input from user

Scanner sc = new Scanner(System.in);

// create object of ConstructAVLTree class object for costructing AVL Tree

ConstructAVLTree obj = new ConstructAVLTree();

char choice; //initialize a character type variable to choice

// perform operation of AVL Tree using switch

do

{

System.out.println("\nSelect an operation:\n");

System.out.println("1. Insert a node");

System.out.println("2. Search a node");

System.out.println("3. Get total number of nodes in AVL Tree");

System.out.println("4. Is AVL Tree empty?");

System.out.println("5. Remove all nodes from AVL Tree");

System.out.println("6. Display AVL Tree in Post order");

System.out.println("7. Display AVL Tree in Pre order");

System.out.println("8. Display AVL Tree in In order");

//get choice from user

int ch = sc.nextInt();

switch (ch)

{

case 1 :

System.out.println("Please enter an element to insert in AVL Tree");

obj.insertElement( sc.nextInt() );

break;

case 2 :

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

System.out.println(obj.searchElement( sc.nextInt() ));

break;

case 3 :

System.out.println(obj.getTotalNumberOfNodes());

break;

case 4 :

System.out.println(obj.checkEmpty());

break;

case 5 :

obj.removeAll();

System.out.println("\nTree Cleared successfully");

break;

case 6 :

System.out.println("\nDisplay AVL Tree in Post order");

obj.postorderTraversal();

break;

case 7 :

System.out.println("\nDisplay AVL Tree in Pre order");

obj.preorderTraversal();

break;

case 8 :

System.out.println("\nDisplay AVL Tree in In order");

obj.inorderTraversal();

break;

default :

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

break;

}

System.out.println("\nPress 'y' or 'Y' to continue \n");

choice = sc.next().charAt(0);

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

}

}

Output

