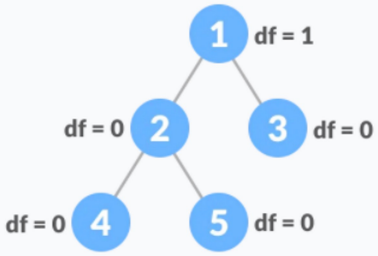
Balanced Binary Tree

In this tutorial, you will learn about a balanced binary tree and its different types. Also, you will find working examples of a balanced binary tree in C, C++, Java and Python.

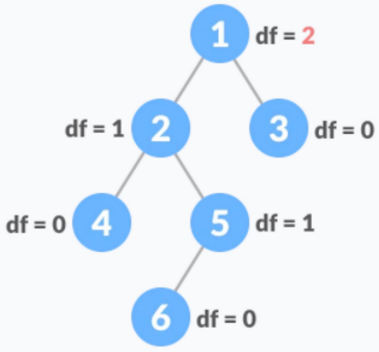
A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

To learn more about the height of a tree/node, visit Tree Data Structure.Following are the conditions for a height-balanced binary tree:

1. difference between the left and the right subtree for any node is not more than one
2. the left subtree is balanced
3. the right subtree is balanced



Balanced Binary Tree with depth at each level



df = |height of left child - height of right child|

Unbalanced Binary Tree with depth at each level

```java
package BST;
public class BST_to_Balanced_BST {

}
// Checking if a binary tree is height balanced in Java
// Node creation
class Node {
    int data;
    Node left, right;

    Node(int d) {
      data = d;
      left = right = null;
    }
  }

  // Calculate height
  class Height {
    int height = 0;
  }

  class BinaryTree {

    Node root;

    // Check height balance
    boolean checkHeightBalance(Node root, Height height) {

      // Check for emptiness
      if (root == null) {
        height.height = 0;
        return true;
      }

      Height leftHeighteight = new Height(), rightHeighteight = new Height();

      boolean l = checkHeightBalance(root.left, leftHeighteight);

      boolean r = checkHeightBalance(root.right, rightHeighteight);

      int leftHeight = leftHeighteight.height, rightHeight = rightHeighteight.height;

      height.height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

      if ((leftHeight - rightHeight >= 2) || (rightHeight - leftHeight >= 2))
        return false;

      else
        return l && r;
    }

    public static void main(String args[]) {
      Height height = new Height();

      BinaryTree tree = new BinaryTree();

      tree.root = new Node(1);

      tree.root.left = new Node(2);
      tree.root.right = new Node(3);

      tree.root.left.left = new Node(4);
      tree.root.left.right = new Node(5);

      if (tree.checkHeightBalance(tree.root, height))
        System.out.println("The tree is balanced");

      else
        System.out.println("The tree is not balanced");
    }
  }
```

AVL Tree

In this tutorial, you will learn what an avl tree is. Also, you will find working examples of various operations performed on an avl tree in C, C++, Java and Python.

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

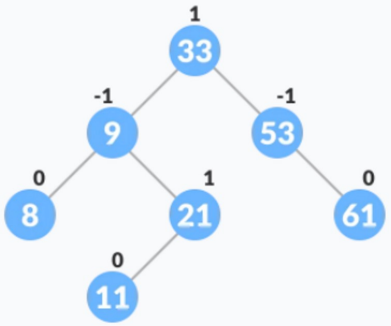
Balance Factor

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 (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced avl tree is:



Avl tree

Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

Rotating the subtrees in an AVL Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

In rotation operation, the positions of the nodes of a subtree are interchanged.

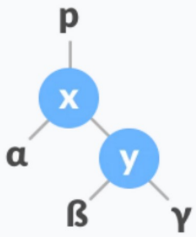
There are two types of rotations:

Left Rotate

In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

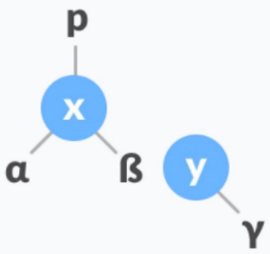
Algorithm

1. Let the initial tree be:



Left rotate

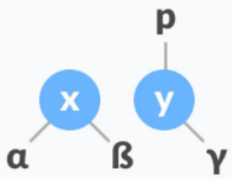
2. If `y` has a left subtree, assign `x` as the parent of the left subtree of `y`.



Assign x as the parent of the left subtree of y

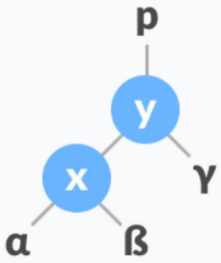
3. If the parent of `x` is `NULL`, make `y` as the root of the tree.
4. Else if `x` is the left child of `p`, make `y` as the left child of `p`.

5. Else assign `y` as the right child of `p`.



Change the parent of x to that of y

6. Make `y` as the parent of `x`.

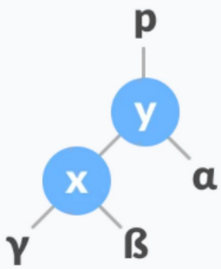


Assign y as the parent of x.

Right Rotate

In left-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

1. Let the initial tree be:



Initial tree

```
// Tree class
class AVLTree {
  Node root;
```

```
    int height(Node N) {
      if (N == null)
        return 0;
      return N.height;
    }

    int max(int a, int b) {
      return (a > b) ? a : b;
    }

    Node rightRotate(Node y) {
      Node x = y.left;
      Node T2 = x.right;
      x.right = y;
      y.left = T2;
      y.height = max(height(y.left), height(y.right)) + 1;
      x.height = max(height(x.left), height(x.right)) + 1;
      return x;
    }

    Node leftRotate(Node x) {
      Node y = x.right;
      Node T2 = y.left;
      y.left = x;
      x.right = T2;
      x.height = max(height(x.left), height(x.right)) + 1;
      y.height = max(height(y.left), height(y.right)) + 1;
      return y;
    }

    // Get balance factor of a node
    int getBalanceFactor(Node N) {
      if (N == null)
        return 0;
      return height(N.left) - height(N.right);
    }

    // Insert a node
    Node insertNode(Node node, int item) {

      // Find the position and insert the node
      if (node == null)
        return (new Node(item));

      if (item < node.item)
        node.left = insertNode(node.left, item);

      else if (item > node.item)
        node.right = insertNode(node.right, item);

      else
        return node;

      // Update the balance factor of each node
      // And, balance the tree
      node.height = 1 + max(height(node.left), height(node.right));

      int balanceFactor = getBalanceFactor(node);

      if (balanceFactor > 1) {

        if (item < node.left.item) {

          return rightRotate(node);
        }
else if (item > node.left.item) {

          node.left = leftRotate(node.left);

          return rightRotate(node);
        }
      }
      if (balanceFactor < -1) {
        if (item > node.right.item) {

          return leftRotate(node);

        }
```

```
else if (item < node.right.item) {

          node.right = rightRotate(node.right);
          return leftRotate(node);
        }
      }
      return node;
    }

    Node nodeWithMimumValue(Node node) {
      Node current = node;
      while (current.left != null)
        current = current.left;
      return current;
    }

    // Delete a node
    Node deleteNode(Node root, int item) {

      // Find the node to be deleted and remove it
      if (root == null)
        return root;
      if (item < root.item)
        root.left = deleteNode(root.left, item);

      else if (item > root.item)
        root.right = deleteNode(root.right, item);
      else {

        if ((root.left == null) || (root.right == null)) {

          Node temp = null;

          if (temp == root.left)
            temp = root.right;
          else
            temp = root.left;


          if (temp == null) {

            temp = root;

            root = null;

          } else
            root = temp;

        } else {
          Node temp = nodeWithMimumValue(root.right);

          root.item = temp.item;

          root.right = deleteNode(root.right, temp.item);
        }
      }
      if (root == null)
        return root;

      // Update the balance factor of each node and balance the tree
      root.height = max(height(root.left), height(root.right)) + 1;

      int balanceFactor = getBalanceFactor(root);

      if (balanceFactor > 1) {

        if (getBalanceFactor(root.left) >= 0) {

          return rightRotate(root);

        } else {

          root.left = leftRotate(root.left);

          return rightRotate(root);
        }
      }
      if (balanceFactor < -1) {
```

```
      if (getBalanceFactor(root.right) <= 0) {

        return leftRotate(root);

      } else {

        root.right = rightRotate(root.right);

        return leftRotate(root);
      }
    }
    return root;
  }

  void preOrder(Node node) {
    if (node != null) {
      System.out.print(node.item + " ");

      preOrder(node.left);

      preOrder(node.right);

    }
  }

  // Print the tree
  private void printTree(Node currPtr, String indent, boolean last) {

    if (currPtr != null) {
      System.out.print(indent);

      if (last) {
        System.out.print("R----");

        indent += "   ";
      } else {

        System.out.print("L----");

        indent += "|  ";
      }

      System.out.println(currPtr.item);

      printTree(currPtr.left, indent, false);

      printTree(currPtr.right, indent, true);
    }
  }

  // Driver code
  public static void main(String[] args) {
    AVLTree tree = new AVLTree();
    tree.root = tree.insertNode(tree.root, 33);

    tree.root = tree.insertNode(tree.root, 13);

    tree.root = tree.insertNode(tree.root, 53);

    tree.root = tree.insertNode(tree.root, 9);
    tree.root = tree.insertNode(tree.root, 21);

    tree.root = tree.insertNode(tree.root, 61);
    tree.root = tree.insertNode(tree.root, 8);

    tree.root = tree.insertNode(tree.root, 11);
    tree.printTree(tree.root, "", true);
    tree.root = tree.deleteNode(tree.root, 13);

    System.out.println("After Deletion: ");
    tree.printTree(tree.root, "", true);
  }
}
```

Complexities of Different Operations on an AVL Tree

Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)