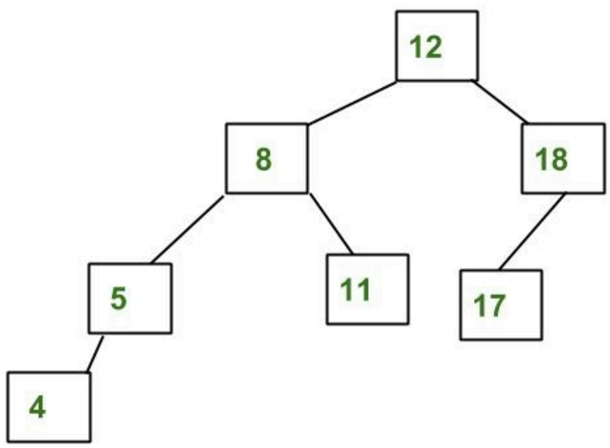
Insertion in an AVL Tree

Last Updated : 22 Feb, 2025

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.

Example of AVL Tree:

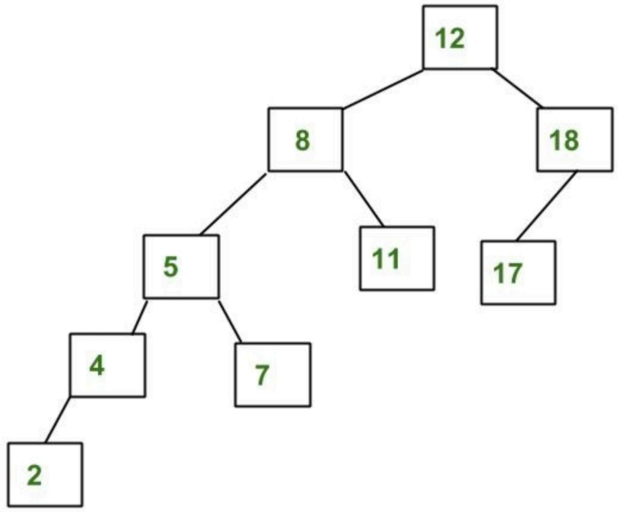


The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Got It !



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees? Most of the BST operations (e.g., search, max, min, insert, delete, floor and ceiling) take **O(h)** time where **h** is the height of the BST. The cost of these operations may become **O(n)** for a **skewed Binary tree**. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

```
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left
side) or x (on the right side)
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

```
     / \     Right Rotation          /  \
    x   T3   - - - - - - - >        T1   y
   / \       < - - - - - - -            / \
  T1  T2     Left Rotation            T2  T3
```

```
Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

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)

*Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to **re-balance** the subtree rooted with **z** and the complete tree becomes balanced as the height of the subtree (After appropriate rotations) rooted with **z** becomes the same as it was before insertion.*

1. Left Left Case

```
T1, T2, T3 and T4 are subtrees.
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

```
         y   T4      Right Rotate (z)          x      z
        / \          - - - - - - - - ->       /  \    /  \
       x   T3                               T1  T2  T3  T4
      / \
    T1   T2
```

2. Left Right Case

```
       z                               z                           x
      / \                            /   \                        /
\
     y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y
z
    / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \
/ \
T1   x                            y    T3                    T1  T2
T3  T4
    / \                          / \
  T2   T3                      T1   T2
```

3. Right Right Case

```
   z                                y
 /  \                             /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

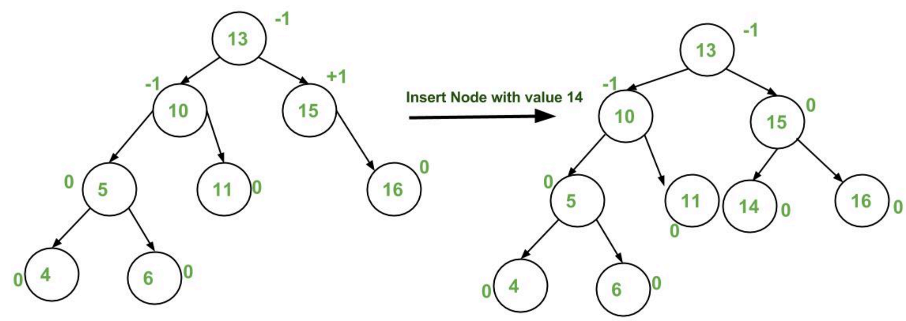
4. Right Left Case

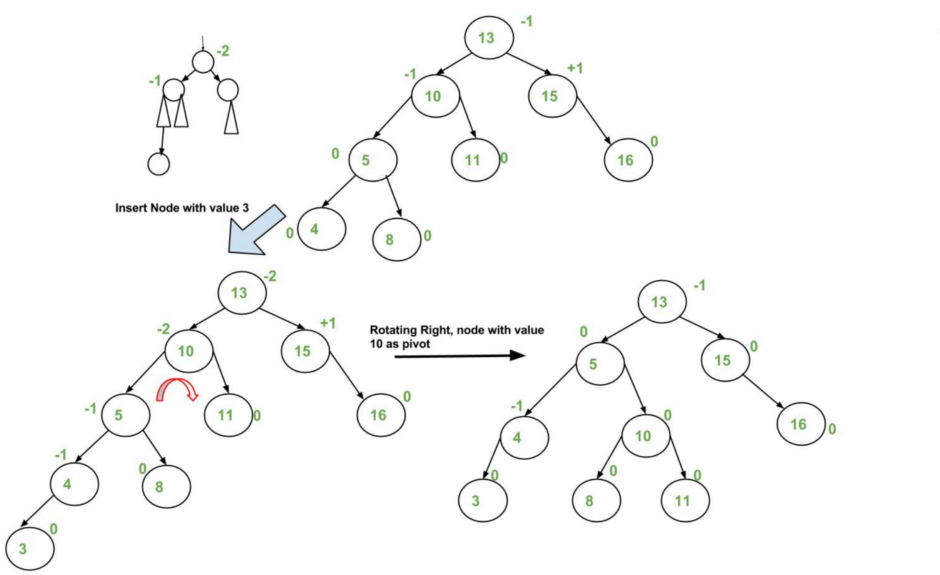
```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    /
\
   x   T4                      T2   y                  T1  T2  T3
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

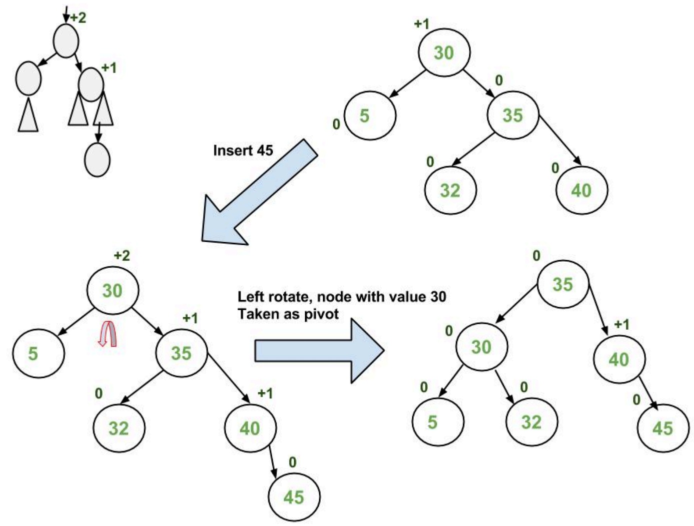
```
         / \                                      /   \
       T2   T3                                  T3    T4
```

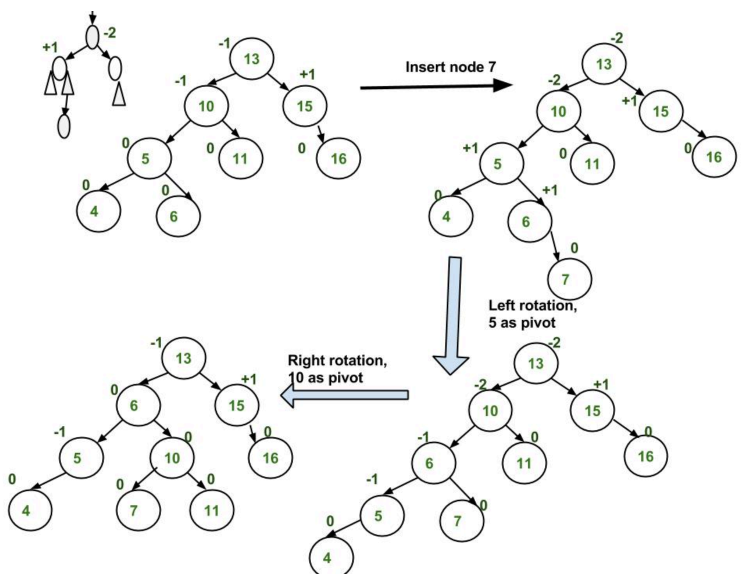
Illustration of Insertion at AVL Tree



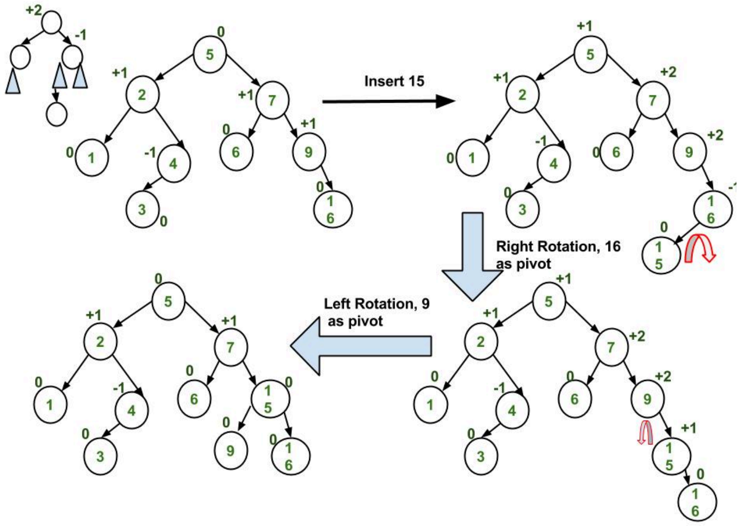


We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy





We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy



Approach: The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

Follow the steps mentioned below to implement the idea:

- Perform the normal [BST insertion.](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor **(left subtree height – right subtree height)** of the current node.
- If the balance factor is greater than **1,** then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key

DSA Interview Problems on Tree Practice Tree MCQs on Tree Tutorial on Tree Types of Trees Basic operati

Below is the implementation of the above approach:

C++14 C Java Python C# JavaScript

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

```
// An AVL tree node
struct Node {
    int key;
    Node *left;
    Node *right;
    int height;

    Node(int k) {
        key = k;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

// A utility function to
// get the height of the tree
int height(Node *N) {
    if (N == nullptr)
        return 0;
    return N->height;
}

// A utility function to right
// rotate subtree rooted with y
Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = 1 + max(height(y->left),
                    height(y->right));
    x->height = 1 + max(height(x->left),
                        height(x->right));

    // Return new root
    return x;
}

// A utility function to left rotate
// subtree rooted with x
Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = 1 + max(height(x->left),
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

```
    // Return new root
    return y;
}

// Get balance factor of node N
int getBalance(Node *N) {
    if (N == nullptr)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in
// the subtree rooted with node
Node* insert(Node* node, int key) {

    // Perform the normal BST insertion
    if (node == nullptr)
        return new Node(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    // Update height of this ancestor node
    node->height = 1 + max(height(node->left),
                           height(node->right));

    // Get the balance factor of this ancestor node
    int balance = getBalance(node);

    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

```
}

// A utility function to print
// preorder traversal of the tree
void preOrder(Node *root) {
    if (root != nullptr) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main() {
    Node *root = nullptr;

    // Constructing tree given in the above figure
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
              30
             /  \
           20    40
          /  \     \
        10   25    50
    */
    cout << "Preorder traversal : \n";
    preOrder(root);

    return 0;
}
```

Output

```
Preorder traversal :
30 20 10 25 40 50
```

Time Complexity: O(log(n)), For Insertion

Auxiliary Space: O(Log n) for recursion call stack as we have written a recursive method to insert

> *The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of the*

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Comparison with Red Black Tree:

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then the AVL tree should be preferred over Red Black Tree.

AVL Tree | Set 2 (Deletion)

Comment More info Advertise with us

Next Article

Insertion, Searching and Deletion in AVL trees containing a parent node pointer

Similar Reads

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 absolute difference between the heights o...

4 min read

What is AVL Tree | AVL Tree meaning

An AVL is a self-balancing Binary Search Tree (BST) where the difference between the heights of left and right subtrees of any node cannot be more than one. KEY POINTSIt is height balanced treeIt is a binary searc...

2 min read

Insertion in an AVL Tree

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Insertion, Searching and Deletion in AVL trees containing a parent node pointer

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. The insertion and deletion in AVL trees have been discussed...

15+ min read

Deletion in an AVL Tree

We have discussed AVL insertion in the previous post. In this post, we will follow a similar approach for deletion. Steps to follow for deletion. To make sure that the given tree remains AVL after every deletion, we...

15+ min read

How is an AVL tree different from a B-tree?

AVL Trees: AVL tree is a self-balancing binary search tree in which each node maintain an extra factor which is called balance factor whose value is either -1, 0 or 1. B-Tree: A B-tree is a self - balancing tree data structure...

1 min read

Practice questions on Height balanced/AVL Tree

AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node canâ€™t be more than 1. Here are some key points about AVL trees: If there are n...

4 min read

AVL with duplicate keys

Please refer below post before reading about AVL tree handling of duplicates. How to handle duplicates in Binary Search Tree?This is to augment AVL tree node to store count together with regular fields like key, left...

15+ min read

Count greater nodes in AVL tree

In this article we will see that how to calculate number of elements which are greater than given value in AVL tree. Examples: Input : x = 5 Root of below AVL tree 9 / \ 1 10 / \ \ 0 5 11 / / \ -1 2 6 Output : 4 Explanation:...

15+ min read

Difference between Binary Search Tree and AVL Tree

Binary Search Tree:A binary Search Tree is a node-based binary tree data structure that has the following properties: The left subtree of a node contains only nodes with keys lesser than the nodeâ€™s key.The right...

2 min read

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Corporate & Communications Address:
A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:
K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305

GET IT ON Google Play
Download on the App Store

Advertise with us

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy

Bootstrap
Web Design

Django

Computer Science
Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths
Software Development
Software Testing

DevOps
Git
Linux
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

System Design
High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

Inteview Preparation
Competitive Programming
Top DS or Algo for CP
Company-Wise Recruitment Process
Company-Wise Preparation
Aptitude Preparation
Puzzles

School Subjects
Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar
Commerce
World GK

GeeksforGeeks Videos
DSA
Python
Java
C++
Web Development
Data Science
CS Subjects

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy