VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belgaum-590018

A File Structures Mini Project Report

on

"AVL Tree"

Submitted in Partial fulfillment of the Requirements for the VI Semester of the Degree

of

Bachelor of Engineering

In

Information Science & Engineering

By

UMAR FAROUQUE

(1CR18IS162)

VARUN C ACHARYA

(1CR18IS168)

YUVRAJ GANDHI

(1CR18IS177)

Under the Guidance of

Mrs. Shilpa Mangesh Pande

Assoc. Professor, Dept. of ISE

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

CMR INSTITUTE OF TECHNOLOGY

#132, AECS LAYOUT, IT PARK ROAD, KUNDALAHALLI,

BANGALORE-560037

CMR INSTITUTE OF TECHNOLOGY

#132, AECS LAYOUT, IT PARK ROAD, KUNDALAHALLI,

BANGALORE-560037

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the File Structures Project work entitled **"AVL Tree"** has been carried out by **Umar Farouque (1CR18IS162), Varun C Acharya (1CR18IS168) and Yuvraj Gandhi (1CR18IS177)** bonafide students of CMR Institute of Technology in partial fulfillment for the award of **Bachelor of Engineering** in **Information Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year **2018-2019**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library. This File Structures mini Project Report has been approved as it satisfies the academic requirements in respect of project work prescribed for the said degree.

------------------------	**-----------------------**
Signature of Guide	**Signature of HOD**
Mrs Shilpa Mangesh Pande **Assoc. Professor** **Dept. of ISE, CMRIT**	**Dr. M Farida Begam** **Assoc. Professor** **Dept. of ISE, CMRIT**

External Viva

Name of the examiners	Signature with date
1.	
2.	

TABLE OF CONTENT

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany a successful completion of any task would be incomplete without the mention of people who made it possible. Success is the epitome of hard work and perseverance, but steadfast of all is encouraging guidance.

So with gratitude I acknowledge all those whose guidance and encouragement served as beacon of light and crowned our effort with success.

I would like to thank **Dr. Sanjay Jain,** Principal, CMRIT, Bangalore for providing an excellent academic environment in the college and his never-ending support for the B.E program.

I would also like to thank **Dr. M Farida Begam,** Assoc. Professor and HOD, Department of Information Science and Engineering, CMRIT, Bangalore who shared his opinions and experiences through which I received the required information crucial for the project.

I consider it a privilege and honour to express my sincere gratitude to my internal guide **Mrs. Shilpa Mangesh Pande**, Assoc. Professor, Information Science and Engineering, CMRIT, Bangalore for their valuable guidance throughout the tenure of this project work.

I would also like to thank all the faculty members who have always been very Co-operative and generous. Conclusively, I also thank all the non-teaching staff and all others who have done immense help directly or indirectly during my project

Umar Farouque

Varun C Acharya

Yuvraj Gandhi

ABSTRACT

AVL Tree is the first dynamic tree in data structure which minimizes its height during insertion and deletion operations. This is because search time is directly proportional to the height of the binary search tree (BST).

When insertion operation is performed it may result in increasing the height of the tree and when deletion is performed it may result in decreasing the height of the tree. To make the BST a height balance tree (AVL Tree) creators of AVL Tree proposed various rotations. This paper tells us about the AVL Tree, the operations that are performed such as rotations, insertion, deletion and tree traversals.

Chapter 1

INTRODUCTION

AVL Tree is a height-balanced binary tree. Each node is associated with a balanced factor which is calculated as the difference between the height of its left subtree and the right subtree.

1.1 Brief history of AVL Tree

In computer science, an **AVL tree** (named after inventors **A**delson-**V**elsky and **L**andis) is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take ***O(log n)*** time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

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

AVL trees are often compared with red–black trees because both support the same set of operations and take ***O(log n)*** time for the basic operations. For lookup-intensive applications, AVL trees are faster than red–black trees because they are more strictly balanced. Similar to red–black trees, AVL trees are height-balanced. Both are, in general, neither weight-balanced nor μ-balanced for any $\mu \leq 1/2$ that is, sibling nodes can have hugely differing numbers of descendants.

Chapter 2

Problem Statement

2.1 AVL Tree Data Structure

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**. The AVL tree was introduced in 1962 by G.M. Adelson-Velsky and E.M. Landis.

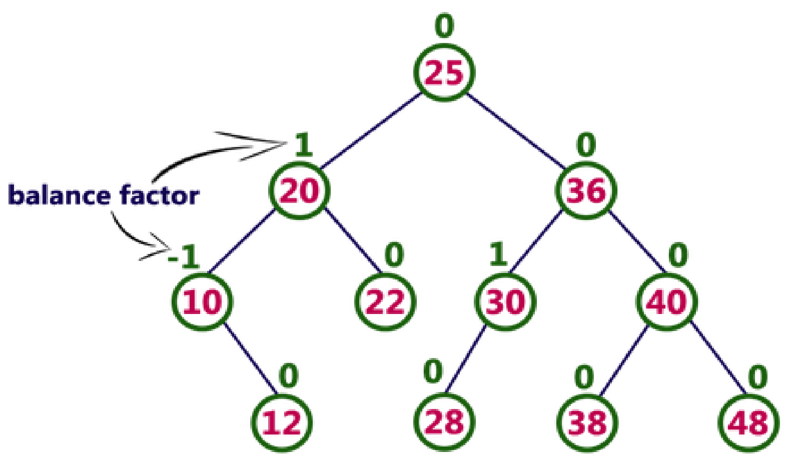
An AVL tree is defined as follows…

An AVL tree is a balanced binary search tree. In an AVL tree, the balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we calculate as follows…

Balance factor = Height Of Left Subtree - Height Of Right Subtree

Example of AVL Tree



The above tree is a binary search tree and every node satisfies the balance factor condition. So this tree is said to be an AVL tree.

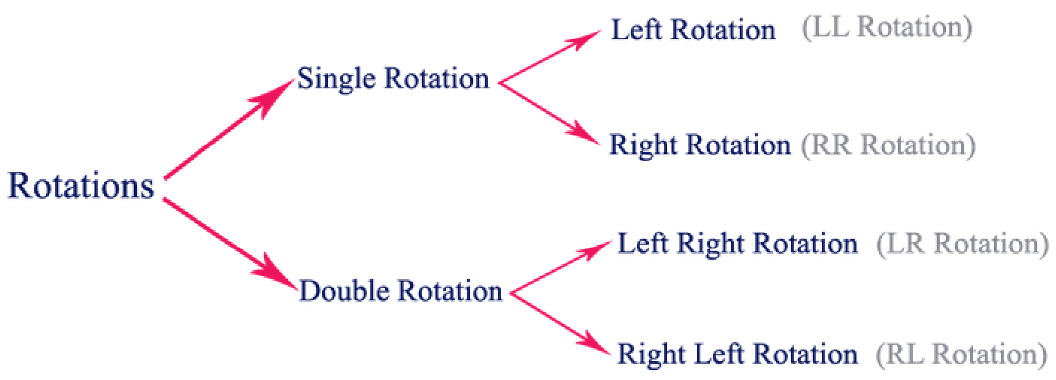
2.2 AVL Tree Rotations

n AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

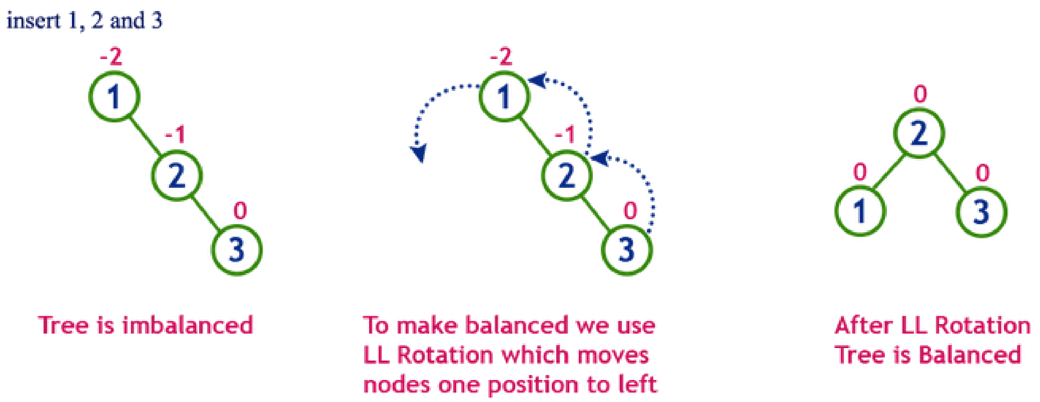
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types:



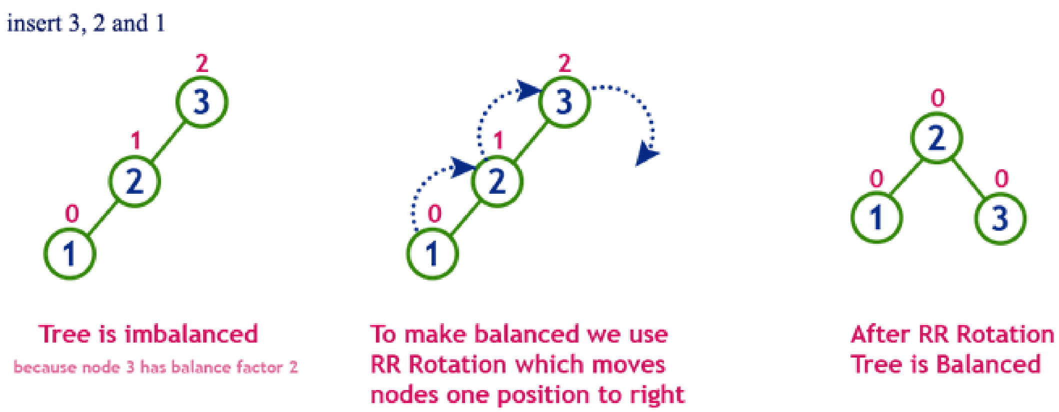
2.2.1 Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree…



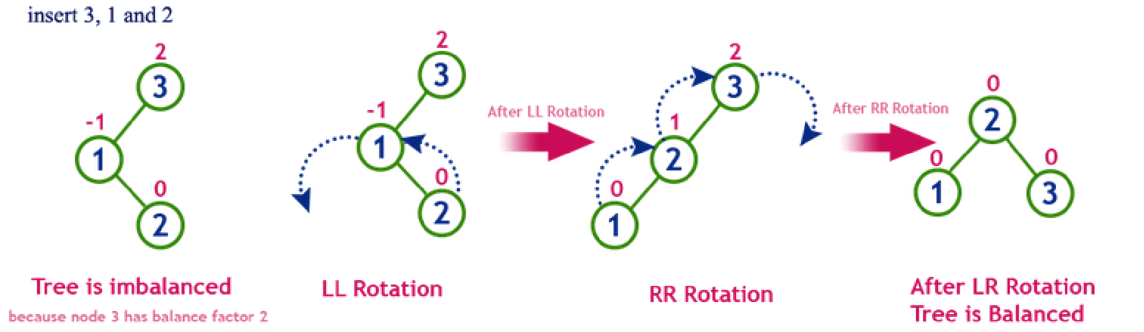
2.2.2 Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



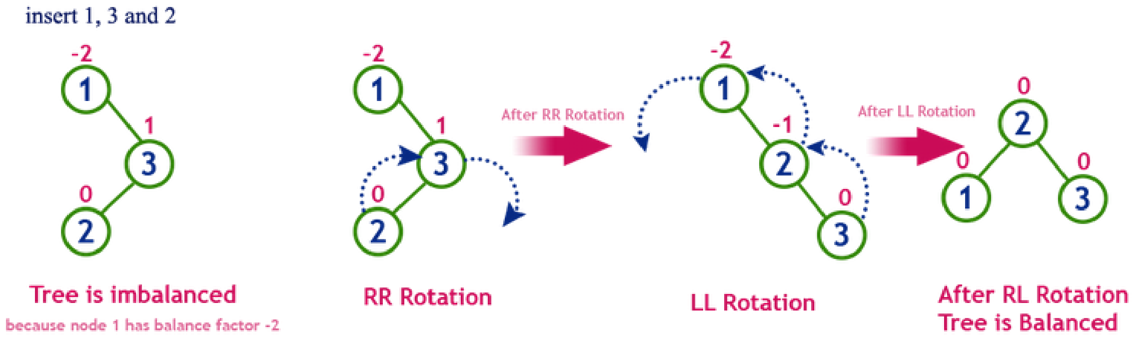
2.2.3 Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree…



2.2.4 Right Left Rotation (RL Rotation)

The RL Rotation is a sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Chapter 3

Operations on AVL Tree

The following operations are performed on AVL Tree…

- **Search**
- **Insertion**
- **Deletion**

3.1 Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search for an element in the AVL tree…

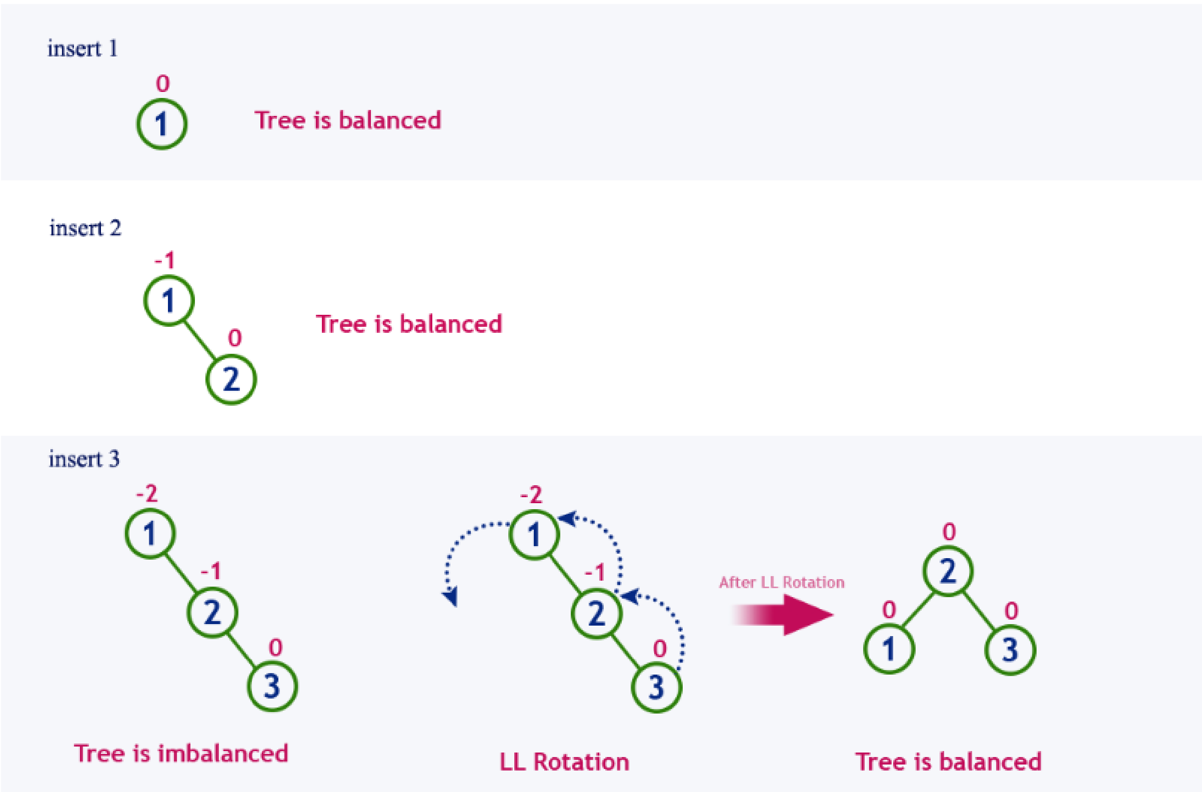
- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with the value of the root node in the tree.
- **Step 3** - If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4** - If both are not matched, then check whether the search element is smaller or larger than that node value.
- **Step 5** - If the search element is smaller, then continue the search process in the left subtree.
- **Step 6** - If the search element is larger, then continue the search process in the right subtree.
- **Step 7** - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8** - If we reach the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9** - If we reach the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

3.2 Insertion Operation in AVL Tree

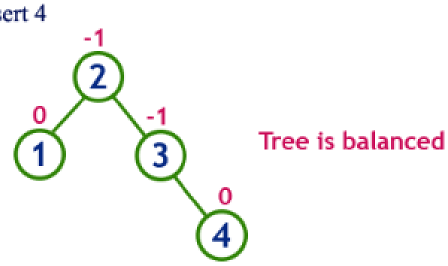
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows…

- **Step 1** - Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2** - After insertion, check the **Balance Factor** of every node.
- **Step 3** - If the **Balance Factor** of every node is **0 or 1 or -1** then go for the next operation.
- **Step 4** - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform a suitable **Rotation** to make it balanced and go for the next operation.

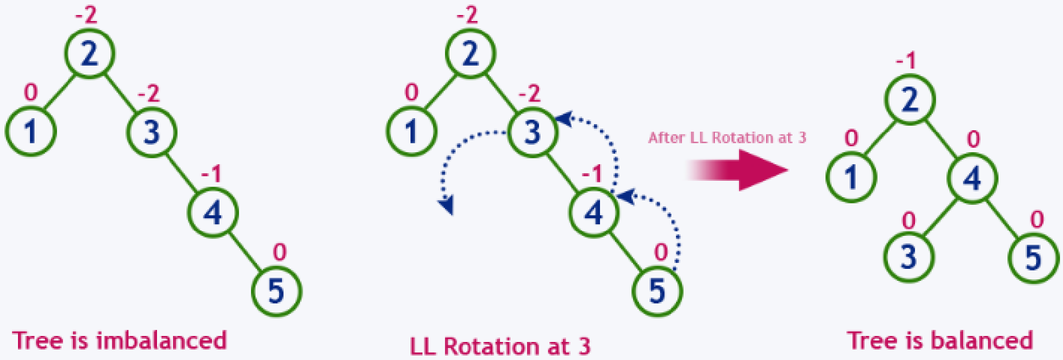
Example: Construct an AVL Tree by inserting numbers from 1 to 8.



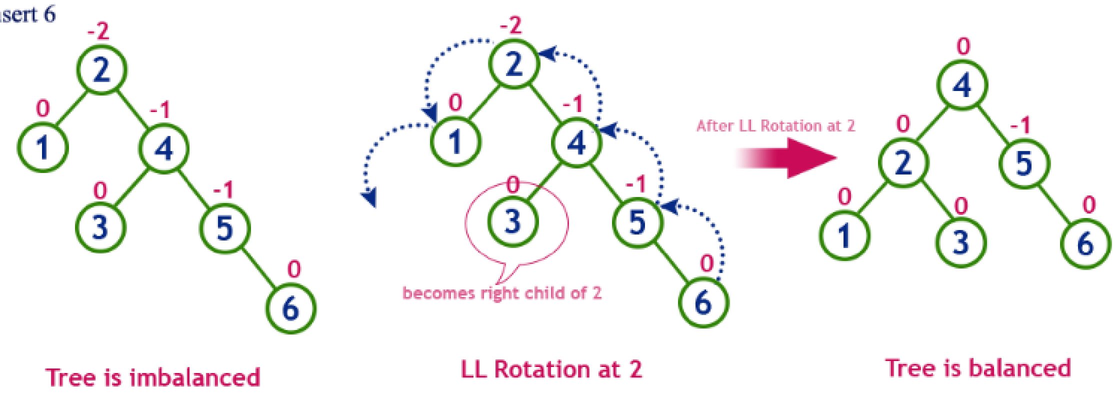
insert 4



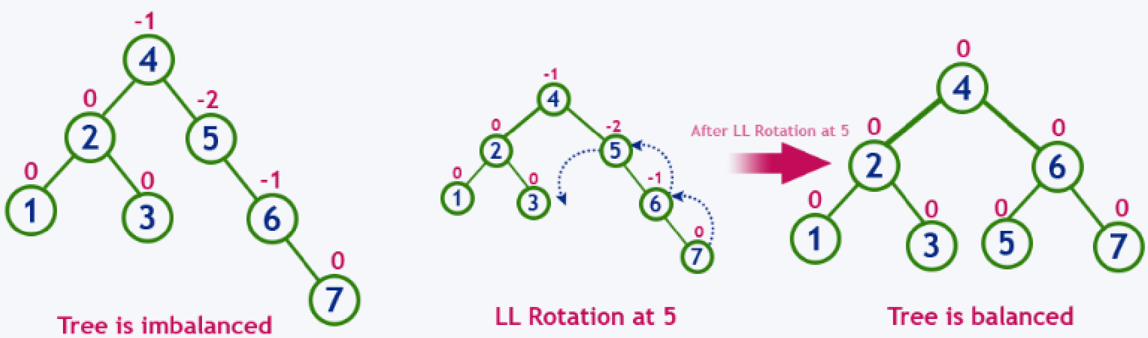
insert 5

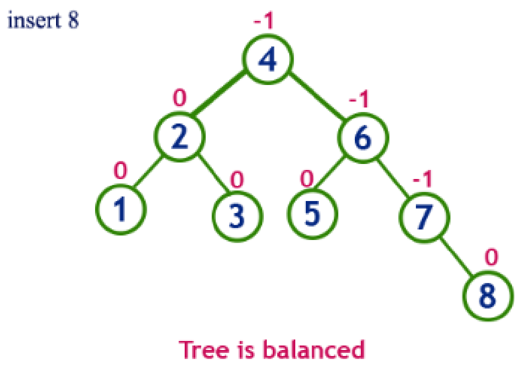


insert 6



insert 7





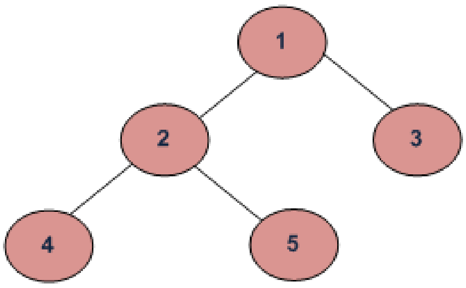
3.3 Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

- **Step 1 -** Perform the normal BST deletion.
- **Step 2 -** The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- **Step 3 -** Get the balance factor (left subtree height – right subtree height) of the current node.
- **Step 4 -** 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 Left Right case, get the balance factor of the left subtree. If the balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
- **Step 5 -** 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 Right Right case or Right Left case, get the balance factor of the 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.

3.4 Tree Traversal

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

3.4.1 Inorder Traversal

Algorithm Inorder(tree)

1. *Traverse the left subtree, i.e., call Inorder(left-subtree)*
2. *Visit the root.*
3. *Traverse the right subtree, i.e., call Inorder(right-subtree)*

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

3.4.2 Preorder Traversal

Algorithm Preorder(tree)

1. *Visit the root.*
2. *Traverse the left subtree, i.e., call Preorder(left-subtree)*
3. *Traverse the right subtree, i.e., call Preorder(right-subtree)*

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

3.4.3 Postorder Traversal

Algorithm Postorder(tree)

1. *Traverse the left subtree, i.e., call Postorder(left-subtree)*
2. *Traverse the right subtree, i.e., call Postorder(right-subtree)*
3. *Visit the root.*

Uses of Postorder

Postorder traversal is used to delete the tree.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Chapter 4

Implementation

```
#include<iostream>
#include<cstdio>
#include<sstream>
#include<algorithm>

#define pow2(n)(1 << (n))

using namespace std;

//Node Declaration

struct avl_node

{
  int data;
  struct avl_node * left;
  struct avl_node * right;
}* root;

//Class Declaration

class avlTree {
  public:
          int height(avl_node * );
  int diff(avl_node * );

  avl_node * rr_rotation(avl_node * );
  avl_node * ll_rotation(avl_node * );
  avl_node * lr_rotation(avl_node * );
  avl_node * rl_rotation(avl_node * );

  avl_node * balance(avl_node * );
  avl_node * insert(avl_node * , int);

  void display(avl_node * , int);
  void inorder(avl_node * );
  void preorder(avl_node * );
  void postorder(avl_node * );

  avlTree() {
          root = NULL;
  }
```

```
};

//Main Contains Menu

int main() {
  int choice, item;
  avlTree avl;
  while (1) {
          cout << "\n---------------------" << endl;
          cout << "AVL Tree Implementation" << endl;
          cout << "---------------------" << endl;
          cout << "1.Insert Element into the tree" << endl;
          cout << "2.Display Balanced AVL Tree" << endl;
          cout << "3.InOrder traversal" << endl;
          cout << "4.PreOrder traversal" << endl;
          cout << "5.PostOrder traversal" << endl;
          cout << "6.Exit" << endl;
          cout << "Enter your Choice: ";
          cin >> choice;

          switch (choice) {
          case 1:
          cout << "Enter value to be inserted: ";
          cin >> item;
          root = avl.insert(root, item);
          break;

          case 2:
          if (root == NULL) {
          cout << "Tree is Empty" << endl;
          continue;
          }
          cout << "Balanced AVL Tree:" << endl;
          avl.display(root, 1);
          break;

          case 3:
          cout << "Inorder Traversal:" << endl;
          avl.inorder(root);
          cout << endl;
          break;

          case 4:
          cout << "Preorder Traversal:" << endl;
          avl.preorder(root);
          cout << endl;
          break;
```

```
        case 5:
        cout << "Postorder Traversal:" << endl;
        avl.postorder(root);
        cout << endl;
        break;

        case 6:
        exit(1);
        break;

        default:
        cout << "Wrong Choice" << endl;
        }
  }
  return 0;
}

//Height of AVL Tree

int avlTree::height(avl_node * temp) {
  int h = 0;

  if (temp != NULL) {
        int l_height = height(temp -> left);
        int r_height = height(temp -> right);
        int max_height = max(l_height, r_height);
        h = max_height + 1;
  }
  return h;
}

//Height Difference

int avlTree::diff(avl_node * temp) {
  int l_height = height(temp -> left);
  int r_height = height(temp -> right);
  int b_factor = l_height - r_height;
  return b_factor;
}

//Right- Right Rotation

avl_node * avlTree::rr_rotation(avl_node * parent) {
  avl_node * temp;
  temp = parent -> right;
  parent -> right = temp -> left;
  temp -> left = parent;
```

```
  return temp;
}

//Left- Left Rotation

avl_node * avlTree::ll_rotation(avl_node * parent) {
  avl_node * temp;
  temp = parent -> left;
  parent -> left = temp -> right;
  temp -> right = parent;

  return temp;
}

//Left - Right Rotation

avl_node * avlTree::lr_rotation(avl_node * parent) {
  avl_node * temp;
  temp = parent -> left;
  parent -> left = rr_rotation(temp);

  return ll_rotation(parent);
}

//Right- Left Rotation

avl_node * avlTree::rl_rotation(avl_node * parent) {
  avl_node * temp;
  temp = parent -> right;
  parent -> right = ll_rotation(temp);

  return rr_rotation(parent);
}

//Balancing AVL Tree

avl_node * avlTree::balance(avl_node * temp) {
  int bal_factor = diff(temp);

  if (bal_factor > 1) {
        if (diff(temp -> left) > 0)
        temp = ll_rotation(temp);
        else
        temp = lr_rotation(temp);
  } else if (bal_factor < -1) {
        if (diff(temp -> right) > 0)
        temp = rl_rotation(temp);
        else
```

```
        temp = rr_rotation(temp);
  }
  return temp;
}
```

//Insert Element into the tree

```
avl_node * avlTree::insert(avl_node * root, int value) {
  if (root == NULL) {
        root = new avl_node;
        root -> data = value;
        root -> left = NULL;
        root -> right = NULL;

        return root;
  } else if (value < root -> data) {
        root -> left = insert(root -> left, value);
        root = balance(root);
  } else if (value >= root -> data) {
        root -> right = insert(root -> right, value);
        root = balance(root);
  }
  return root;
}
```

//Display AVL Tree

```
void avlTree::display(avl_node * ptr, int level) {
  int i;

  if (ptr != NULL) {
        display(ptr -> right, level + 1);
        printf("\n");

        if (ptr == root)
        cout << "Root -> ";

        for (i = 0; i < level && ptr != root; i++)
        cout << "        ";
        cout << ptr -> data;
        display(ptr -> left, level + 1);
  }
}
```

//Inorder Traversal of AVL Tree

```
void avlTree::inorder(avl_node * tree)
```

```
{
  if (tree == NULL)
          return;

  inorder(tree -> left);
  cout << tree -> data << "  ";
  inorder(tree -> right);
}
```

```
//Preorder Traversal of AVL Tree

void avlTree::preorder(avl_node * tree) {
  if (tree == NULL)
          return;

  cout << tree -> data << "  ";
  preorder(tree -> left);
  preorder(tree -> right);
}

//Postorder Traversal of AVL Tree

void avlTree::postorder(avl_node * tree) {

  if (tree == NULL)
          return;

  postorder(tree -> left);
  postorder(tree -> right);
  cout << tree -> data << "  ";
}
```

Chapter 5

Results and Analysis

```
---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Tree is Empty

---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 8

---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
```

```
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

Root -> 8
---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 5

---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:
```

Output 5.1 & 5.2

```
Balanced AVL Tree:

Root -> 8
                5
--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 4

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

                8
Root -> 5
                4
--------------------
AVL Tree Implementation
```

```
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 15

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

                        15
                11
                        8
Root -> 5
                4
--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
```

Output 5.3 & 5.4

```
6.Exit
Enter your Choice: 1
Enter value to be inserted: 3

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

                        15
                11
                        8
Root -> 5
                4
                        3
--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
```

```
Enter your Choice: 1
Enter value to be inserted: 6

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

                        15
                11
                        8
                                6
Root -> 5
                4
                        3
--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
```

Output 5.5 & 5.6

```
Enter your Choice: 1
Enter value to be inserted: 2

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

                        15
                11
                        8
                                6
Root -> 5
                        4
                3
                        2
--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 4
```

```
Enter your Choice: 4
Preorder Traversal:
5  3  2  4  11  8  6  15

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 5
Postorder Traversal:
2  4  3  6  8  15  11  5

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 3
Inorder Traversal:
2  3  4  5  6  8  11  15

--------------------
AVL Tree Implementation
```

Output 5.7 & 5.8

```
Inorder Traversal:
2  3  4  5  6  8  11  15

--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:

                        15
                11
                        8
                                6
Root -> 5
                        4
                3
                        2
--------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 6
```

Output 5.9

Chapter 6

Conclusion

AVL trees are balanced binary trees that are mostly used in database indexing.

All the operations performed on AVL trees are similar to those of binary search trees but the only difference in the case of AVL trees is that we need to maintain the balance factor i.e. the data structure should remain a balanced tree as a result of various operations. This is achieved by using the AVL Tree Rotation operation.

They have efficient time and space complexities. As a programmer, it is key to understand the functioning and the implementation of an AVL tree such that it can be mapped to a real world scenario.

Some real world applications are:

1. AVL trees are mostly used for in-memory sorts of sets and dictionaries.
2. AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
3. It is used in applications that require improved searching apart from the database applications.

References

- https://en.wikipedia.org
- Youtube videos on AVL Tree insertion, deletion and rotation operations.
- https://www.geeksforgeeks.org