Karnatak Law Society’s

GOGTE INSTITUTE OF TECHNOLOGY

Udyambag Belagavi -590008

Karnataka, India.



A Course Project Report on

**AVL Trees**

Submitted for the requirements of 3rd semester B.E. in CSE

for **“Data Structures with C (18CS32)”**

**Submitted by**

|  |  |
| --- | --- |
| **NAME** | **USN** |
| **Miss Shradha Mallikarjun Patil** | **2GI20CS144** |
| **Miss Shivani Shivapur** | **2GI20CS141** |
| **Miss Shrushti Gani** | **2GI20CS150** |
| **Mr Siddesh Gurunath Naik** | **2GI20CS152** |

**Under the guidance of**

**Dr. Manjula Ramannavar**

**Asst. Prof., Dept. of CSE**

**Academic Year 2021 - 22(Odd semester)**

**Table of contents**

**S.No. Content Page No.**

1. Title of the project 3
2. Statement of the Problem 3
3. Objectives and scope of the project 3
4. Hardware and software to be used 3
5. Background Details 4
6. Methodology / Algorithm / Pseudocode 6
7. Source Code 8
8. Screenshots 15
9. Conclusion 15
10. References 15

**Title of The Project:**

AVL TREES

**Problem Statement:**

Develop and execute a C program to construct AVL Tree for a given sequence of numbers.

**Objectives and scope of the project:**

1. To understand the concept of AVL Trees

2. We use sequence of numbers to contruct AVL tree

3. Learn how to apply AVL Trees concept in other real real world application problems.

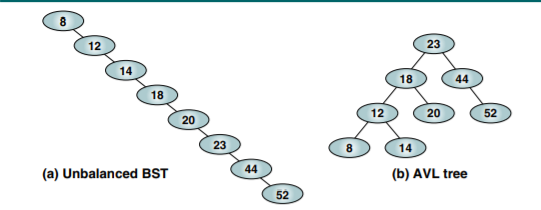
**Hardware and Software to be used:**

Hardware: No hardware is needed.

Software: Any IDE with a C compiler.

**Background Details:**

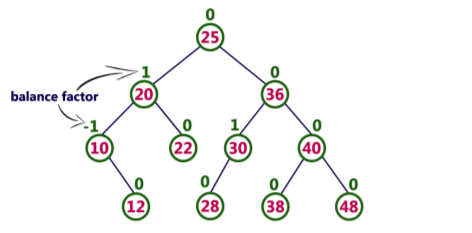
An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1.



|HL – HR| <= 1

AVL Tree Balance Factor

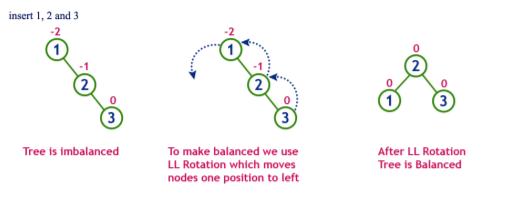
the balance factor for any node in an AVL tree must be +1, 0, or –1, we use the descriptive identifiers LH for left high (+1) to indicate that the left subtree is higher than the right subtree



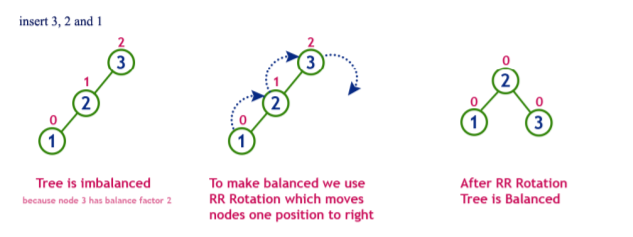
Balancing Trees

Whenever we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced. When we detect that a tree has become unbalanced, we must rebalance it. AVL trees are balanced by rotating nodes either to the left or to the right. In this section we discuss the basic balancing algorithms. We consider four cases that require rebalancing. All unbalanced trees fall into one of these four cases:

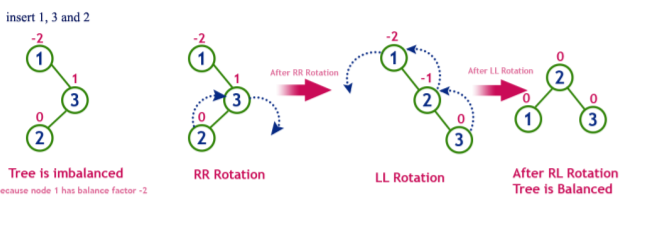
1. Left of left—A subtree of a tree that is left high has also become left high.



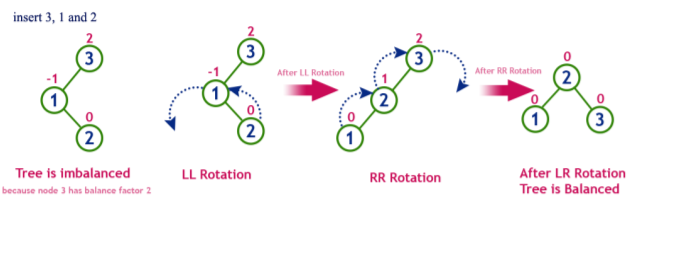
1. Right of right—A subtree of a tree that is right high has also become right high



3.Right of left—A subtree of a tree that is left high has become right high.



4.Left of right—A subtree of a tree that is right high has become left high.



**Methodology / Algorithm / Pseudocode**

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don’t need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.

4) If balance factor is greater than 1, then the current node is unbalanced and we are either in 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 left subtree root.

5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

**Source Code:**

#include<stdio.h>

#include<stdlib.h>

// An AVL tree node

typedef struct Node{

int data;

struct Node \*left;

struct Node \*right;

int height;

}NODE;

// Function to get the height of the tree

int height(struct Node \*N)

{

if (N == NULL)

return 0;

return N->height;

}

// Function to get maximum of two integers

int max(int a, int b)

{

if(a>b)

return a;

if(b>a)

return b;

}

// allocates new node

NODE\* newNode(int key)

{

NODE\* node = (NODE\*)malloc(sizeof(struct Node));

node->data = key;

node->left = NULL;

node->right = NULL;

node->height = 1; // new node is initially added at leaf

return(node);

}

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

x->height = max(height(x->left), height(x->right))+1;

// Return new root

return x;

}

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

y->height = max(height(y->left), height(y->right))+1;

// Return new root

return y;

}

// Get Balance factor of node N

int getBalance(NODE \*N)

{

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

// Recursive function to insert a key in the subtree rooted

// with node and returns the new root of the subtree.

NODE\* insert(NODE\* node, int key)

{

//1. Perform the normal BST insertion

if (node == NULL)

return(newNode(key));

if (key < node->data)

node->left = insert(node->left, key);

else if (key > node->data)

node->right = insert(node->right, key);

else // Equal keys are not allowed in BST

return node;

// 2. Update height of this ancestor node

node->height = 1 + max(height(node->left),height(node->right));

// 3. Get the balance factor of this ancestor

// node to check whether this node became unbalanced

int balance = getBalance(node);

if (balance > 1 && key < node->left->data)

return rightRotate(node);

// Right Right Case

if (balance < -1 && key > node->right->data)

return leftRotate(node);

// Left Right Case

if (balance > 1 && key > node->left->data)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node->right->data)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

// A utility function to print preorder traversal

// of the tree.

// The function also prints height of every node

void postOrder(NODE \*root)

{

if(root != NULL)

{

printf("%d ", root->data);

postOrder(root->left);

postOrder(root->right);

}

}

//Driver program to test above function

int main()

{

NODE \*root = NULL;

int n,item,i;

printf("\nEnter the number of nodes\n");

scanf("%d",&n);

printf("\nEnter the nodes\n");

for(i=0;i<n;i++)

{

scanf("%d",&item);

root = insert(root,item);

}

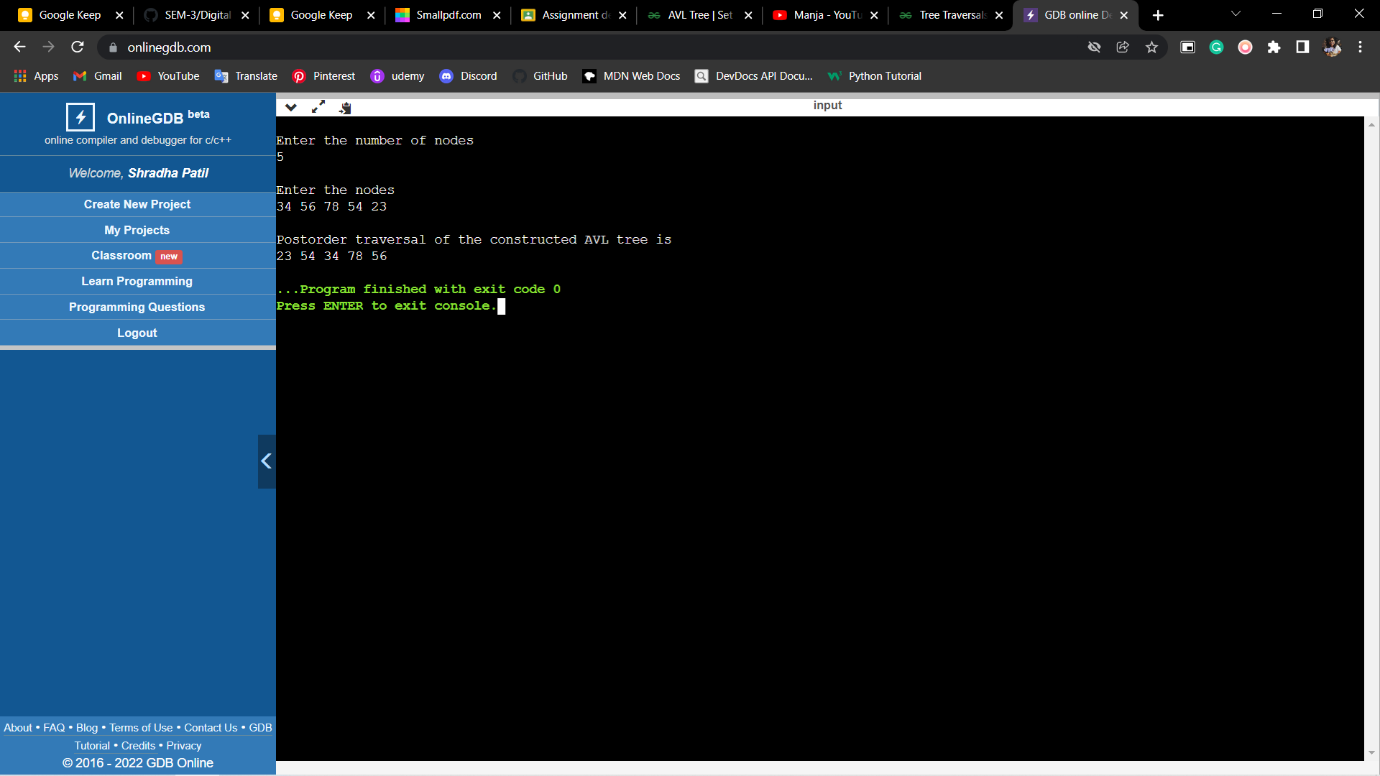
printf("\nPostorder traversal of the constructed AVL tree is \n");

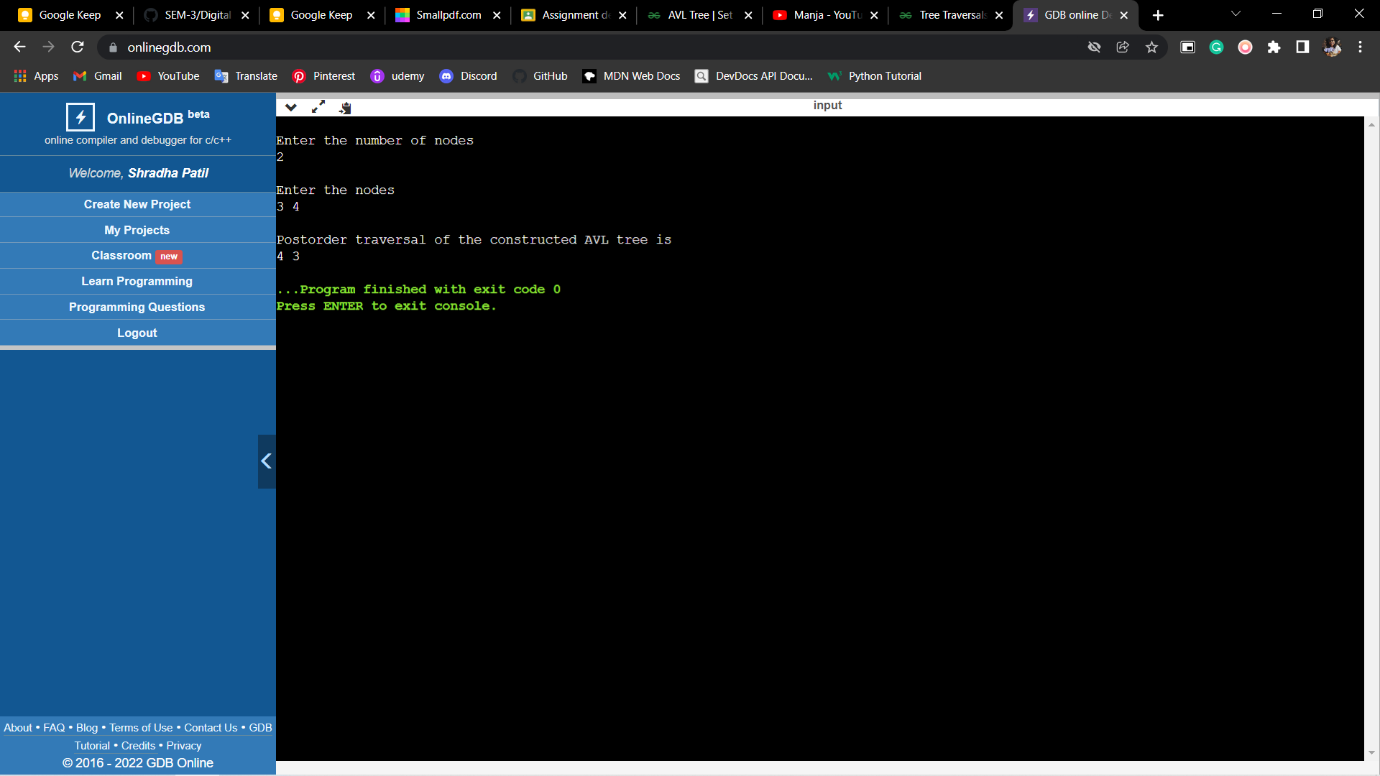
postOrder(root);

return 0;

}

**OUTPUT:**

****

****

**Conclusion:**

AVL tree is a descendant of Binary Search Tree but overcomes its drawback of increasing complexity if the elements are sorted. It monitors the balance factor of the tree to be 0 or 1 or -1. In case it tree becomes unbalanced corresponding rotation techniques are performed to balance the tree.

**References:**

1.Richard.F.Gilberg, Behrouz.A. Forouzan, Data Structures: A Pseudocode Approach with C,

Cengage Learning, 2nd edition 2007 and onwards

2. Horowitz, Sahni, Anderson-Freed, Fundamentals of Data Structures in C, Universities

Press, 2nd Edition, 2007 and onwards.