**K L HYDERABAD**

**FRESHMAN ENGINEERING DEPARTMENT**

**A Project-Based Lab Report**

**On**

**AVL Trees**

**SUBMITTED BY:**

2110030359 B Chihnita Reddy

**UNDER THE ESTEEMED GUIDANCE OF**

**Dr. G. Rekha**

**Assistant Professor**



**KONERU LAKSHMAIAH EDUCATION FOUNDATION**

(Deemed to be University)

Moinabad Road, Aziz Nagar, Hyderabad - 500075

**DEPARTMENT OF BASIC ENGINEERING SCIENCES**

**CERTIFICATE**

This is to certify that the project-based laboratory report entitled “AVL TREES” submitted by B CHIHNITA REDDY(2110030359) to the **Department of Basic Engineering Sciences, KL University** in partial fulfillment of the requirements for the completion of a project in the “Data Structures-21SC1202” course in I B Tech II Semester, is a bonafide record of the work carried out by him/her under my supervision during the academic year 2021-22.

PROJECT SUPERVISOR HEAD OF THE DEPARTMENT

Dr. G. Rekha Dr. G. REKHA

**ACKNOWLEDGEMENTS**

It is great pleasure for me to express my gratitude to our honorable President **Sri. Koneru Satyanarayana**, for giving the opportunity and platform with facilities in accomplishing the project-based laboratory report.

I express sincere gratitude to our Principal **Dr.L. Koteswara Rao** for his administration of our academic growth.

I express sincere gratitude to our Coordinator **Dr. G. Rekha** for her leadership and constant motivation provided in the successful completion of our academic semester. I record it as my privilege to deeply thank you for providing us with the efficient faculty and facilities to make our ideas into reality.

I express my sincere thanks to our project supervisor **Dr.G. Rekha** for her novel association of ideas, encouragement, appreciation, and intellectual zeal which motivated us to venture into this project successfully.

Finally, it is pleased to acknowledge the indebtedness to all those who devoted themselves directly or indirectly to making this project report success.

**ABSTRACT**

This project is about AVL trees in data structures.

The main aim of this project is to create and operate an AVL Tree. It’s a self-balancing binary search tree. In the AVL tree, the difference between the right and left subtree heights 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.

We can learn about AVL trees in data structures. in this project

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TITLE** | **PAGE NO** |
| 1 | Introduction | <6> |
| 2 | Aim of the Project | <9> |
| 2.1 | Advantages | <9> |
| 2.2 | Disadvantages | <9> |
| 3 | Algorithm | <9> |
| 4 | Implementation, code | <10-22> |
| 5 | Output screenshots | <23> |
| 6 | Conclusion | <24> |
|  |  |  |

**INTRODUCTION**

The main aim of this project is to create and operate an AVL Tree. 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. A balanced binary search tree where the height of the two subtrees (children) of a node differs by at most one.

AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations: -

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

Left Rotation: -

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left subtree of B.

Right Rotation: -

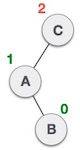
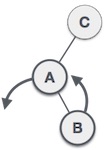
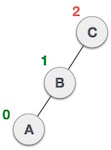
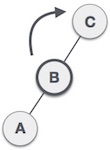
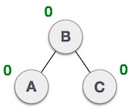
AVL tree may become unbalanced if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

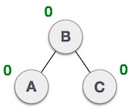
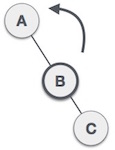
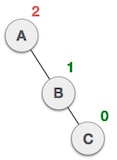
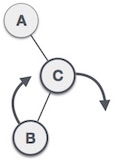
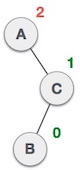
Left-Right Rotation: -

Double rotations are a slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of a left rotation followed by a right rotation.

### Right-Left Rotation: -

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by a left rotation.



**Requirements:** To implement this project students should know on

1. **Creating a Binary tree**
2. **Know about AVL tree**
3. **Binary factor finding**
4. **Rotations and operations**

**AIM**

The main aim of this project is to create and operate an AVL Tree

**Advantages: -**

* The height of the **AVL tree** is always balanced. The height never grows beyond log N, where N is the total number of nodes in the **tree**.
* It gives better search time complexity when compared to simple Binary Search **trees**.
* **AVL trees** have self-balancing capabilities.

**Disadvantages: -**

* AVL trees have high constant factors for some operations.
* AVL trees are difficult to implement.

ALGORITHM

* Step1: Start.
* Start2: It shows 5 options to select

1. Create a node

2. Insert a node

3. Delete a node

4. print a node (inorder, preorder, postorder)

5. Quit

* Step3: Enter Your Choice
* Step4: Depending on what you choose it will show the result

**IMPLEMENTATION**

**CODE**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*left,\*right;

int ht;

};

struct node \*insert(struct node \*,int);

struct node \*Delete(struct node \*,int);

void preorder(struct node \*);

void inorder(struct node \*);

void postorder(struct node \*);

int height( struct node \*);

struct node \*rotateright(struct node \*);

struct node \*rotateleft(struct node \*);

struct node \*RightRight(struct node \*);

struct node \*LeftLeft(struct node \*);

struct node \*LeftRight(struct node \*);

struct node \*RightLeft(struct node \*);

int BF(struct node \*);

int main()

{

struct node \*root=NULL;

int x,n,i,op;

do

{

printf("\n1.Create:");

printf("\n2.Insert:");

printf("\n3.Delete:");

printf("\n4.Print:");

printf("\n5.Quit:");

printf("\n\nEnter Your Choice");

scanf("%d",&op);

switch(op)

{

case 1:

printf("\nEnter no.of elements:");

scanf("%d",&n);

printf("\nEnter tree data:");

root=NULL;

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

{

scanf("%d",&x);

root=insert(root,x);

}

break;

case 2:

printf("\nEnter a data:");

scanf("%d",&x);

root=insert(root,x);

break;

case 3:

printf("\nEnter a data:");

scanf("%d",&x);

root=Delete(root,x);

break;

case 4:

printf("\nPreorder sequence:\n");

preorder(root);

printf("\n\nInorder sequence:\n");

inorder(root);

printf("\n\nPostorder sequence:\n");

postorder(root);

printf("\n");

break;

}

}

while(op!=5);

return 0;

}

struct node \* insert(struct node \*temp,int x)

{

if(temp==NULL)

{

temp=(struct node\*)malloc(sizeof(struct node));

temp->data=x;

temp->left=NULL;

temp->right=NULL;

}

else

if(x > temp->data)

{

temp->right=insert(temp->right,x);

if(BF(temp)==-2)

if(x>temp->right->data)

temp=RightRight(temp);

else

temp=RightLeft(temp);

}

else

if(x<temp->data)

{

temp->left=insert(temp->left,x);

if(BF(temp)==2)

if(x < temp->left->data)

temp=LeftLeft(temp);

else

temp=LeftRight(temp);

}

temp->ht=height(temp);

return(temp);

}

struct node \* Delete(struct node \*temp,int x)

{

struct node \*p;

if(temp==NULL)

{

return NULL;

}

else

if(x > temp->data)

{

temp->right=Delete(temp->right,x);

if(BF(temp)==2)

if(BF(temp->left)>=0)

temp=LeftLeft(temp);

else

temp=LeftRight(temp);

}

else

if(x<temp->data)

{

temp->left=Delete(temp->left,x);

if(BF(temp)==-2)

if(BF(temp->right)<=0)

temp=RightRight(temp);

else

temp=RightLeft(temp);

}

else

{

//data to be deleted is found

if(temp->right!=NULL)

{

//delete its inorder succesor

p=temp->right;

while(p->left!=NULL)

p=p->left;

temp->data=p->data;

temp->right=Delete(temp->right,p->data);

if(BF(temp)==2)

if(BF(temp->left)>=0)

temp=LeftLeft(temp);

else

temp=LeftRight(temp);

}

else

return(temp->left);

}

temp->ht=height(temp);

return(temp);

}

int height(struct node \*temp)

{

int lh,rh;

if(temp==NULL)

return(0);

if(temp->left==NULL)

lh=0;

else

lh=1+temp->left->ht;

if(temp->right==NULL)

rh=0;

else

rh=1+temp->right->ht;

if(lh>rh)

return(lh);

return(rh);

}

struct node \* rotateright(struct node \*x)

{

struct node \*y;

y=x->left;

x->left=y->right;

y->right=x;

x->ht=height(x);

y->ht=height(y);

return(y);

}

struct node \* rotateleft(struct node \*x)

{

struct node \*y;

y=x->right;

x->right=y->left;

y->left=x;

x->ht=height(x);

y->ht=height(y);

return(y);

}

struct node \* RightRight(struct node \*temp)

{

temp=rotateleft(temp);

return(temp);

}

struct node \* LeftLeft(struct node \*temp)

{

temp=rotateright(temp);

return(temp);

}

struct node \* LeftRight(struct node \*temp)

{

temp->left=rotateleft(temp->left);

temp=rotateright(temp);

return(temp);

}

struct node \* RightLeft(struct node \*temp)

{

temp->right=rotateright(temp->right);

temp=rotateleft(temp);

return(temp);

}

int BF(struct node \*temp)

{

int lh,rh;

if(temp==NULL)

return(0);

if(temp->left==NULL)

lh=0;

else

lh=1+temp->left->ht;

if(temp->right==NULL)

rh=0;

else

rh=1+temp->right->ht;

return(lh-rh);

}

void preorder(struct node \*temp)

{

if(temp!=NULL)

{

printf("%d(Bf=%d)",temp->data,BF(temp));

preorder(temp->left);

preorder(temp->right);

}

}

void inorder(struct node \*temp)

{

if(temp!=NULL)

{

inorder(temp->left);

printf("%d(Bf=%d)",temp->data,BF(temp));

inorder(temp->right);

}

}

void postorder(struct node \*temp)

{

if(temp!=NULL)

{

postorder(temp->left);

postorder(temp->right);

printf("%d(Bf=%d)",temp->data,BF(temp));

}

}

**OUTPUTS**

|  |
| --- |
|  |
|  |

**CONCLUSION**.

This project is about AVL trees in data structures.

The main aim of this project is to create and operate an AVL Tree using the rotations.

Time Complexity of an AVL tree: -

* insertion and deletion - O(log n)
* Searching – O(n)

This time complexity is better than the binary search tree and red-black tree.

**-------THE END-------**