****

**PES University, Bangalore.**

**Final Semester Assessment (FSA) – B.Tech. (CSE) – IV Sem**

**Session: January – May, 2019**

**UE17CS251 – Design and Analysis of Algorithms**

Project Report

On

“Implementation of AVL Trees in C Language.”

Submitted by

|  |  |
| --- | --- |
| Name 1 Akanksha Tonne | SRN 1 PES1201700275 |
| Name 2 Rachana Dani | SRN 2 PES1201700950 |

**Name of the Examiners Signature with Date**

.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Department of Computer Science & Engineering**

**PES University**

(Established under Karnataka Act No. 16 of 2013)

100 Feet Ring Road, Bengaluru – 560 085, Karnataka, India

**Title**: Implementation of AVL Trees in C language.

**Introduction**: AVL Trees are self balancing binary search trees. For each node in AVL trees a condition must be satisfied that the balancing factor of every node ε {-1, 0, 1}, where the balancing factor is defined as the height of the left sub-tree of a node – height of right sub-tree.

**Idea**: To make sure that the balancing factor of the nodes in the AVL tree always remains in range we use the preorder traversal. Whenever a new node is inserted or a node (which is present in the tree) is deleted we recursively check the balancing factor of each node in the AVL tree by performing preorder traversal. In this way, we first determine the node closest to the leaf nodes that have lost balance then followed by other nodes that are far from the leaf nodes. Once we determine the critical node (node whose balancing factor is not in the node) we perform the appropriate rotate operation to balance that sub-tree of the AVL tree whose root has a balancing factor out of the range.

**AVL Tree Operations**:

1. Search
2. Delete
3. Insert
4. Inorder
5. Postorder
6. Preorder
7. Height
8. Count

**Github link**:

<https://github.com/Akanksha-Tonne/AVL_Trees_Implementation.git>

**Algorithms**:

ALGORITHM Right\_Rotate( parent )

// performs right rotate on sub-tree rooted at parent

if parent is not null and parent → left is not null

t ← parent

temp ← t → left → right

parent ← t → left

t → left ← temp

parent → right ← t

ALGORITHM Left\_Rotate( parent )

// performs right rotate on sub-tree rooted at parent

if parent is not null and parent → right is not null

temp ← parent

temp2 ← temp → right → left

parent ← temp1 → right

t → left ← temp1

parent → right ← temp1

ALGORITHM Right\_left\_Rotate( parent )

//performs right left rotate on sub-tree rooted at parent

//uses predefined Left\_Rotate algorithm to perform left rotate operation

if parent is not null and parent → right is not null

temp1 ← parent → right

temp2 ← temp1 → left

parent → right ← temp2

temp1 → left ← temp2 →right

temp2 → right ← temp1

Left\_Rotate(parent)

ALGORITHM Left\_Right\_Rotate( parent )

//performs left right rotate on sub-tree rooted at parent

//uses predefined Right\_Rotate algorithm to perform left rotate operation

if parent is not null and parent → left is not null

temp1 ← parent → left

temp2 ← temp1 → right

parent → left ← temp2

temp1 → right ← temp2 →left

temp2 → left ← temp1

Right\_Rotate(parent)

ALGORITHM Delete\_Main( parent, key )

//deletes the given key from the avl tree

//deletes the given key from the binary search tree and calls the Balance\_Tree function

delete key from the binary search tree

Balance\_Tree( parent, key )

ALGORITHM Balance\_Tree( parent, key)

// balances using the approach mentioned in the idea section of the report

//uses the rotate functions defined previously

If count of nodes in less than 3

return

else

Balance\_Tree(parent→left, key)

Balance\_Tree(parent→left, key)

if bf(parent ) > 1

critical\_node ← parent

if critical\_node ← data < data

bf\_left ← bf( critical\_node → left )

if bf\_left =0 or bf\_right = 1

Right\_Rotate(parent)

else

Left\_Right\_Rotate( parent )

Else

bf\_right = bf(critical\_node → right)

if bf\_right = 0 or bf\_right =-1

Left\_Rotate( parent )

else

Right\_Left\_Rotate( parent )

ALGORITHM Balance (root, key)

//Balances the AVL tree on insertion of a node

//Input: Pointer to the root node of tree and the key which was inserted

//Output: Balanced AVL tree

If root is not null then

               Balance (root→ left, key)

               Balance (root→ right, key)

               bf ← get\_bf(root);

               if |bf| > 1 then

                              G←root data

                              If key < G then

                                             S← root→left→data

                                             If key > S

                                                            left\_right\_rotate(root)

                                             else

                                                            right\_rotate(root)

                              else

                                             P← root→right→data;

                                             If key > P

                                                            left\_rotate(root)

                                             else

                                                            right\_left\_rotate(root)

ALGORITHM get\_bc(node)

// Gets the balancing factor of the node, by taking the difference between its left and right subtrees

// by using the height function

//Input: pointer to the node

//Output: balancing factor of the node

x ←height(node→left)-height(node→right)

return x

**Text cases**:

Initial tree: 50 20 13 33 70

Insertion 12:

R rotation

20 13 12 50 33 70

Insertion 25:

20 13 12 50 33 25 70

Insertion 30:

LR rotation

20 13 12 50 30 25 33 70

Insertion 31:

LR rotation

20 13 12 33 30 25 31 50 70

Deletion 12:

L Rotation

33 20 13 30 25 31 50 70

Deletion 50:

LR rotation

30 20 13 25 33 31 70

**Time complexity**:

Rotate operations have time complexity Ө(1) as the rate of growth does not depend on input size.

Insertion has a time complexity of  O(log(n) + n).

Deletion has a time complexity of O(n)

Searching has a time complexity of log(n)