International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 1 40 - 47

AVL Tree Implementation

Prof. Meenal Jabde[1]
Department of Computer Science,
Modern college, Ganeshkhind,
SavitribaiPhule University, Pune
meenal82@gmail.com

Prof. Mandar Upasani[2]
Department of Computer Science
Maeer'sArt's , Science & Commerce
college, Pune
mandar.upasani@gmail.com

Varsha A.Jadhav[3]
Department of Computer Science,
Modern college, Ganeshkhind,
SavitribaiPhule University, Pune
vj444563@gmail.com

Lina P. Bachhav[4]
Department of Computer Science,
Modern College, Ganeshkhind,
SavitribaiPhule University, Pune
lina100990@gmail.com

Abstract: This paper is about result of a series of simulation which investigates the performance of AVL tree. AVL Tree is a program develops in C++ to create and arrange data in hierarchical manner.

In computer science, an AVL tree is a self-balancing binary search tree, and it was the first such data structure to be invented. In an AVL tree, the heights of the two child sub-trees of any node differ by at most one. 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.

Keywords: *AVL, Binary Tree, Balance Factor, Rotation, Subtree, Recursive Function*

Objective: The implementation of AVL tree will provide the detail information about how to balance an unbalanced tree by examining balance Factors, height & proper rotations.

Balancing The Unbalanced Tree: This algorithm will calculate the balance factor of every node in a tree while inserting or deleting a node in a tree and if tree get unbalanced then our system convert it into a balanced tree.

User Friendly Interface For Balancing The Unbalanced Tree.

*****

I. Introduction

An AVL (Adelson-velskii and Landis) tree is a ***'height balanced tree'*** invented since 1962. In 1962, many alternatives have been proposed, with the goal of simpler implementation or better performance or both. This paper empirically examines the computational cost of insertion, deletion, and retrieval in AVL trees. An AVL tree is any rooted, binary tree with every node having the property. Balanced tree structures are efficient ways of sorting information. It provides an excellent solution for the dictionary data structure problem. For n elements the operations find, insert and delete can be done in O(log N) unit if time. These tress are binary search trees in which the heights of two siblings are not permitted more than one.

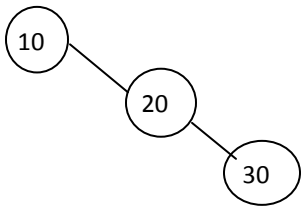
II. Height Balanced tree:

The Minimum height of the binary tree having n nodes can be $\log_2(n)+1$.But creating such tree will be difficult, because in binary search tree the data will decide the place as to where it should be attached and in binary trees it will be the users decision as where a node should be, when user decides the place of the node.

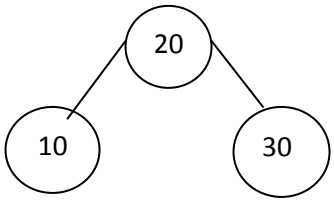
An empty tree is height balanced. A binary tree with h_l and h_r as height of left and right subtree respectively is height balanced if $| h_l - h_r | <= 1$. A binary tree is height balanced if every subtree of the given tree is height balanced.

- Example:

  For binary search tree, if data is given in sequence 10,20,30 then the tree will be created as follows



But if we change the root, we may get the tree as follows



For performing such jobs, we will have to take the decisions, as how to replace the nodes or which node will be the new root, etc.

Instead of minimizing the height as such we may define a factor which is associated with each node of the tree, say a Balance Factor.

III. Balance Factor:

The balance factor, BF(T) of node T in a binary search tree is defined as $h_l - h_r$ where h_l and $h_r$are the heights of the left and the right sub trees of T.A binary search tree with balance factors is shown in fig (a).

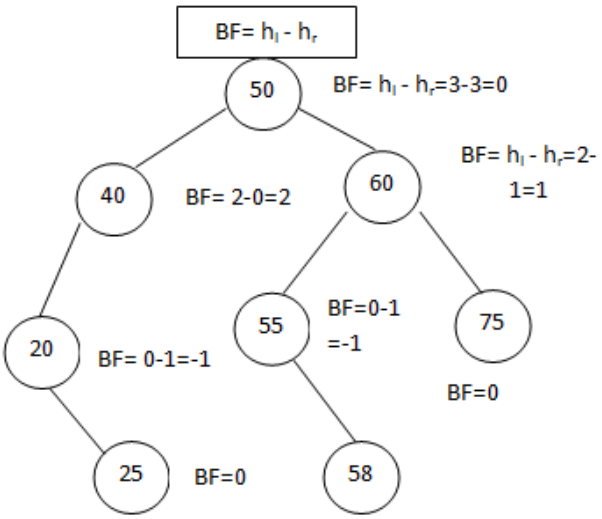
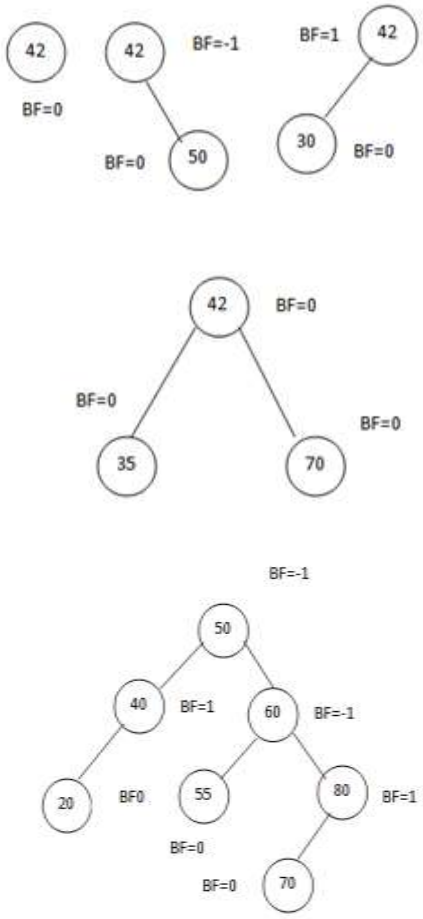


Fig. 1: A sample BST with balance factors

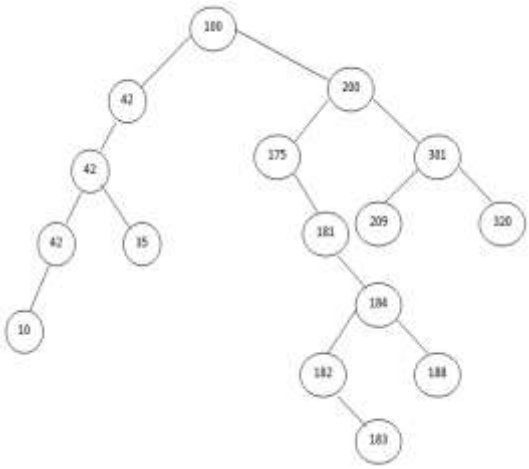
Tree of Fig. (1) is not an AVL tree. The balance factor of the node with data 40 is+2. Tress of Fig (b) is AVL- trees .



Balance Factor =Height(node→ left)- Height(temo→right)

Fig (2): AVL trees

- **Consider the following search tree**



Height of tree with root 100 i.e.T100=1+maximum(height(T64),height(T200))

height (T64)=1+height(T30)

height(T30)=1+maximum(height(T21),height(T35))

height(T21)=1+height(T10)

height(T10)=1

Therefore

height(T21)=1+1=2

height (T30)=1+max(2,1)=1+2=3

height (T64)=1+3

height(T200)=6

height(T100)=1+max(4,6)=1+6=7

Let us see the balance factor of all the nodes in the previous tree.

All the leaf nodes have a balance factor of 0.

B_Factor(21)=1-0=1
B_Factor(30)=2-1=1
B_Factor(64)=3-0=3
B_Factor(182)=0-1=-1
B_Factor(184)=2-1=1
B_Factor(181)=0-3=-3
B_Factor(175)=0-4=-4
B_Factor(301)=1-1=0
B_Factor(200)=5-2=3
B_Factor(100)=4-6=-2

Thus we can say that the tree is not balanced. Also when we find the height of the subtrees, it was defined as height of Null tree is 0 and for the other trees, it is 1+max height among its subtrees. Thus the function is recursive.

- **The recursive function can also be written as follows.**

```
typedef struct tree
{
intdat;
struct tree *left, *right;
}
```

```
int height(HBTR temp)
{
if (!temp)
        return(0);
return(1+max(height(temp→left),height(temp→right)
));
}
```

Where max is a function which return the maximum of two values as follows.

```
int max(inta,int b)
{
if(a>b)
        return(a);
        return(b);
}
```

or it can also be written as

```
int max(inta,int b)
{
return(a>b?a:b)
}
```

or we can modify the height function as follows.

```
int height(HBTR temp)
{
if(temp)
        {
        p=height(temp→left);
        q=height(temp→right);
        return(1+(p>q?p:q));
        }
return(0);
}
```

If we want to write a non-recursive function for finding the height of the tree, then we are required to follow the steps shown below.

As we want to eliminate the recursion, we will have to use stack. But observe that the height of the function has been called and only one of these two values will be returned. Hence a single stack will not serve the purpose. Instead of this complicated way, we can very well go back to BFS, ehre we have written a function for finding number of levels.

IV. Structure of a Node in AVL Tree:

Operations an AVL tree requires calculation of balance factor of nodes. To represent a node in AVL tree, a new field is introduced. This new field stores the height of the node. A node in AVL tree can be defined in the following way:

```
typedef struct node
{
        int data;
        struct node *left,*right;
        int height;
}node;
```

'C' Function for finding the Balance Factor of a node:

```
int BF(node *T)
{
  intlh,rh;
  if(T→left==NULL)
        lh=0;
  else
        lh=1+T→left→height;
  if(T→right==NULL)
        rh=0;
  else
        rh=1+T→right→height;
  return (lh-rh);
}
```

/* the function height(), for finding the height of node is given below*/

```
int height(node *T)
{
  intlh,rh;
  if(T→left==NULL)
        lh=0;
  else
        lh=1+T→left→height
  if(T→right==NULL)
        rh=0;
  else
        rh=1+T→right→height
  if(lh>rh)
        return(lh);
  return(rh);
}
```

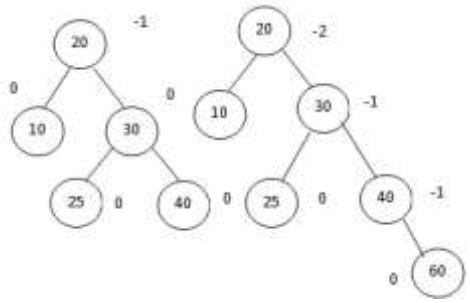
V. Insertion of a Node into an AVL Tree:

Insertion of a new data, say k into an AVL tree is carried out in two steps:

1) Insert the new element, treating the AVL tree a binary tree.
2) Update the balance factors (information, height) working upward from point of insertion to root. It should be clear that from point of insertion to the root may have their height altered.

The steps to insert a new element with value k into an AVL tree begins with comparing k with the value stored in the root.

If k is found to be larger than the value stored in T, then insertion is carried out into the right subtree else insertion is carried out into the left subtree. The recursion terminates when the left or right subtree to which insertion is to be made happens to be empty.



(a)A sample AVL tree with balance factors

(b) Tree of Fig(a) after insertion of 60. Tree is longer an AVL tree

Fig. (3)

Consider the AVL tree from the given figure. Balance factor of each node is shown against the node.

A new element with value 60 is inserted in a way we insert element in a binary search tree. After insertion of 60, the balance factor of root has become -2. Hence, it is no longer an AVL tree. The balance factor of the root has become -2, because the height of its right subtree rooted at (30) has increased by 1.

- Rebalancing of the tree is carried out through rotation of the deepest node with BF=2 or BF= -2.

VI. Rotation:

Fig.4 shows the rotation of tree of Fig.3. After rotation, the tree becomes an AVL tree.

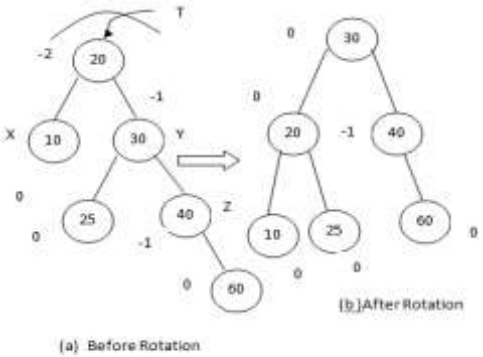


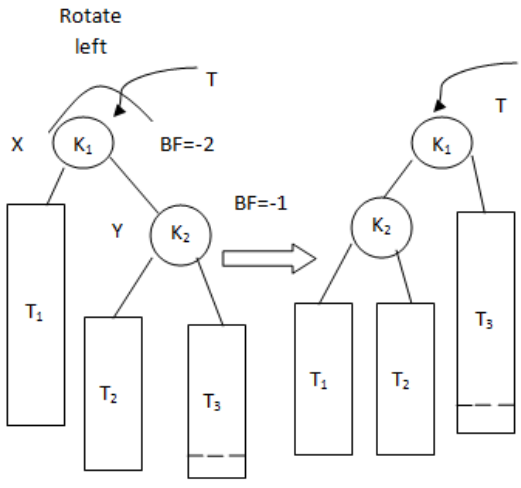
Fig.4 :AVL property is destroyed by insertion of 60. AVL property is fixed by left rotation

- Balance factor of node X in Fig.4(a) is -2.
- Balance factor of -2, tells that the right subtree of X is heavier.
- In order to rebalance the tree, the tree rooted at X (node with BF=-2) is rotated left.

5.1 Rotate Left:

- When a tree rooted at X is rotated left, the right child of X i.e Y will become the root.
- The node X will become the left child of Y.

  Tree T_2 which was the left child of Y will become the right child of X.



(a) Tree with BF=-2 at node X

(b)Tree after rotation

A program segment to rotate left a tree T, rooted at node X(as shown in Fig.5).

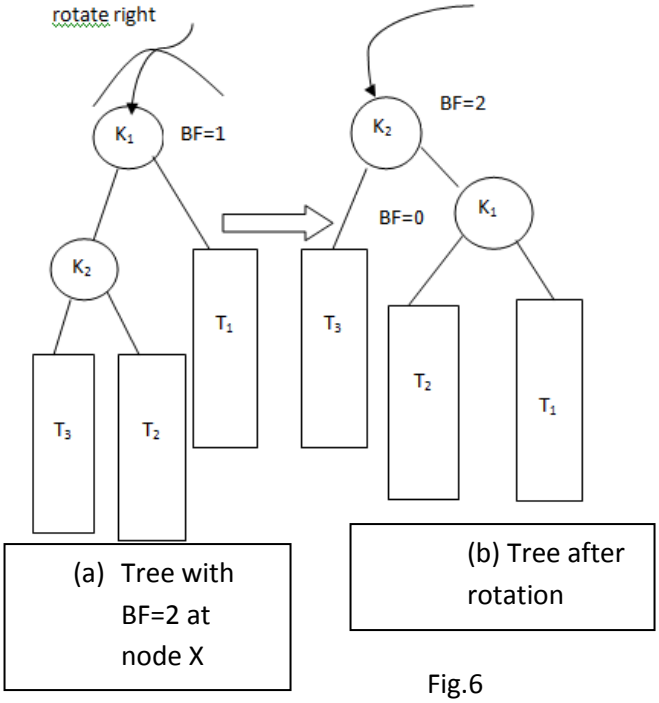
node *temp;

temp=Y →left ; save the address of left child of Y.

T=Y→left=X ; X becomes the left child of Y.

X→right=temp; left child of Y becomes the right child of X.

5.2 Rotate Right:



(a) Tree with BF=2 at node X

(b) Tree after rotation

Fig.6

- When a rooted at X is rooted right, the left child of X i.e Y will become the root.
- The node X will become the right child of Y.
- Tree T_2, which was the right child of Y will become the left child of X.

Balance factor of+2, tells that the subtree of X is heavier. In order to rebalance the tree, the tree rooted at X (node with BF=2) is rotated right;

A program segment to rotate right a tree, rooted at node X(as shown in Fig.6).

node *temp;
temp=Y→right; save the address of right child of Y.
T=Y; Node Y becomes the root node.
Y→right=X; X becomes the right child of Y.
X→left=temp; right child of Y becomes the left child of X.

5.3 Single Rotation and Double Rotation:

5.3.1 Single Rotation:

- **LL:** Let X be the node with BF equal to +2 after insertion of the new node A.

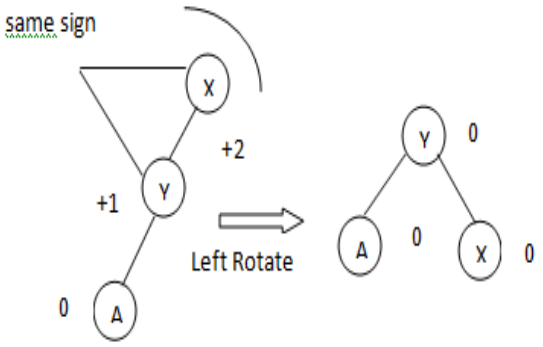


Fig.7.1: Situation known as LL(left of leaf)

New node A is inserted in the left subtree of X. Balance nature of the tree can be resyored through single right rotation of the node X. It is shown in Fig.7

- **RR:** Let X be the node with BF equals to -2, after the insertion of new node A.

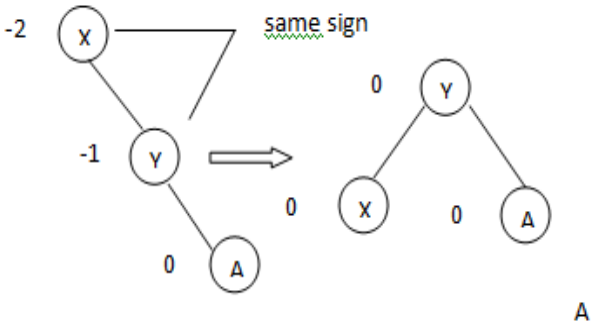


Fig.7.2: Situation known a RR (Right to right)

New node A is inserted in the right subtree of the right subtree of X. Balance nature of the tree can be restored through single left rotation of the node X is shown in Fig.7.2.

5.3.2 Double Rotation:

- **LR**: Let X be the node with Bf equal to +2, after insertion of the node. A balance factor of the node Y, the left child of the node X becomes -1 after insertion of A.

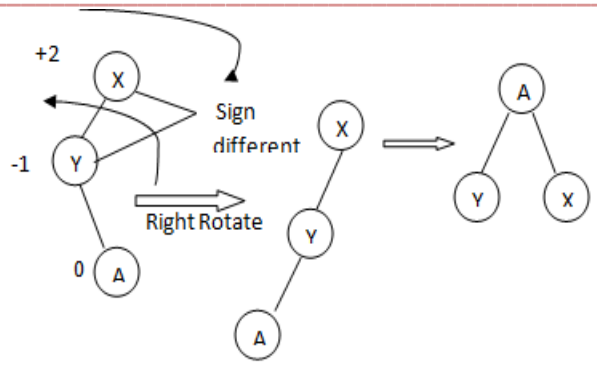


Fig.7.3: Situation known a LR (Right to Left)

New node A is inserted in the right subtree of the left subtree of X. Balance nature of the tree can be restored through double rotation.

(a) node Y is rotated left (b) node is rotated right

It is shown in fig.7.3

- **RL:** Let X be the node with BF==-2, after insertion of the new node A. Balance factor of the node Y, the child of the node X becomes +1 after insertion of A.

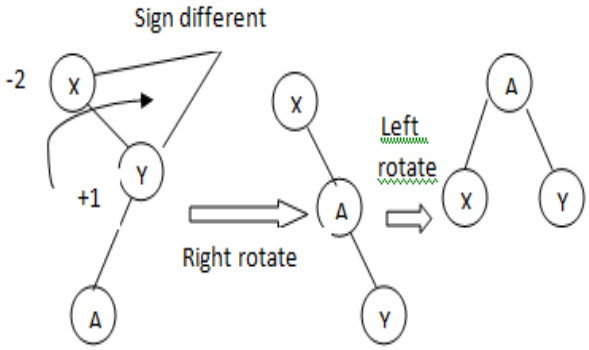


Fig.7.4: Situation known a RL (Left to right)

New node A is inserted in the left subtree of the right subtree of X. Balance nature of the tree can be restored through double rotation.

(a) Node Y is rotated right. (b) Node X is rotated left. It is shown in fig.7.4

5.4 'C' Function for insertion of an element into an AVL Tree:

```
typedef struct node
{
```

```
    int data;
    struct node *left,*right;
    intht;
}node;
node *insert(node *T,int x)
{
    if(T==NULL)
    {
        T=(node*)malloc(sizeof(node));
        T→data=x;
        T→left=NULL;
        T→right=NULL;
    }
    else
        if(x>T→data) // insert in right subtree
        {
            T→right=insert(T→right,x);
            if(BF(T==-2))
            if(x>T→right→data)
                T=RR(T);
            else
                T=RL(T);
        }
    else
            if(x<T→data)
        {
            T→left=insert(T→left,x);
            if(BF(T==2))
            if(x<T→left→data)
                T=LL(T);
            else
                T=LR(T);
        }
        T→ht=height(T);
        return(T)
    }
```

5.5 'C' Function to Find Height of AVL Tree:

```
int height(node *T)
{
    intlh,rh;
    if(T→left==NULL)
        lh=0;
    else
        lh=1+T→left→height;
    if(T→right==NULL)
        rh=0;
    else
        rh=1+T→right→height;
    if(lh>rh)
            return(lh);
        return(rh);
}
```

5.6 'C' Function to Rotate Right:

```
    node *rotateright(node *x)
    {
        node *y;
        y=x→left;
        x→left=y→right;
        y→right=x;
        x→ht=height(x);
        y→ht=height(y);
        return(y);
    }
```

5.7 'C' Function to Rotate Left:

```
    node *rotateleft(node *x)
    {
        node *y;
        y=x→right;
        x→ right =y→left;
        y→left=x;
        x→ht=height(x);
        y→ht=height(y);
        return(y);
    }
```

5.8 'C' Function to RR:

```
    node *RR(node *T)

    {
        T=rotateleft(T);
        return(T);
    }
```

5.9 'C' Function to LL:

```
    node *LL(node *T)

    {
        T=rotateright(T);
        return(T);
    }
```

5.10 'C' Function to LR:

```
node *LR(node *T)
{
    T→left=rotateleft(T→left);
    T=rotateright(T);
    return(T);
}
```

References

[1] "Data Structures, Files and Algorithms" by A.K. Abhyankar, Lecturer, Sinhgad college of Engineering.

[2] "Data Structures and Algorithms" by Dilip Kumar Sultania, B.Tech(hons.) Computer Science and Engineering.

[3] "Data Structures and Program Design in C" by Kruse Robert L .

[4] "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.

[5] Adelson-Velskii, G.M and E.M. Landis, "An algorithm for the Organization of information"

[6] Introduction to computer Organization and Data Structures by Stone,H.S. McGraw-Hill from New York.