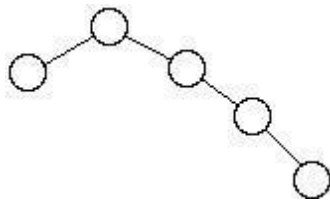


**Title:** - Implementation of AVL- tree.

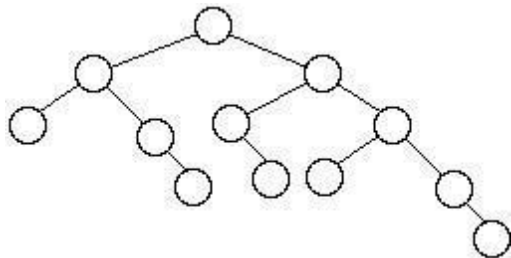
**Problem Statement:** A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword

Height of a tree is the length of the longest path from the root to a leaf. A binary search tree  $T$  is a height-balanced  $k$ -tree or Height Balanced  $[k]$ -tree, if each node in the tree has the HB $[k]$  property.

Below Tree (Fig 1) is not a balanced tree because there exist at least one node for which difference of left height and right height is not in  $\{-1, 0, +1\}$



But tree (Fig 2) is balanced tree.



**AVL TREE:** 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.

The balance factor of a node is the height of its left sub-tree minus the height of its right sub-tree (sometimes opposite) and a node with balance factor 1, 0, or  $-1$  is considered balanced.

A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

### INSERTION:

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains  $-1$ ,  $0$ , or  $+1$  then no rotations are necessary. However, if the magnitude of the balance factor becomes greater or equal to  $2$  then the sub-tree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most one of the following cases needs to be resolved to restore the entire tree to the rules of AVL.

**There are four cases which need to be considered**, of which two are symmetric to the other two. Let  $P$  be the root of the unbalanced sub-tree, with  $R$  and  $L$  denoting the right and left children of  $P$  respectively.

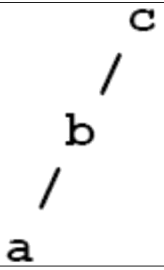
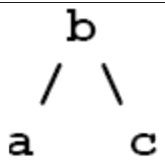
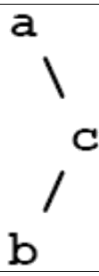
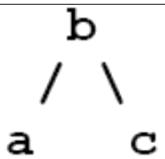
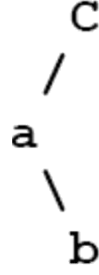
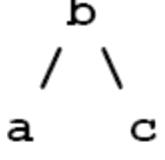
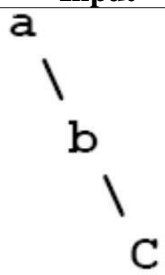
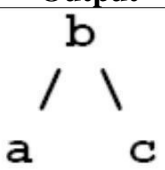
#### Right-Right case and Right-Left case:

- If the balance factor of  $P$  is  $-2$  then the right sub-tree outweighs the left sub-tree of the given node, and the balance factor of the right child ( $R$ ) must be checked. The left rotation with  $P$  as the root is necessary.□
- 
- If the balance factor of  $R$  is  $-1$ , a **single left rotation** (with  $P$  as the root) is needed (Right-Right case).□
- 
- If the balance factor of  $R$  is  $+1$ , two different rotations are needed. The first rotation is a **right rotation** with  $R$  as the root. The second is a **left rotation** with  $P$  as the root (Right-Left case).□

#### Left-Left case and Left-Right case:

- If the balance factor of  $P$  is  $+2$ , then the left sub-tree outweighs the right sub-tree of the given node, and the balance factor of the left child ( $L$ ) must be checked. The right rotation with  $P$  as the root is necessary.□
- 
- If the balance factor of  $L$  is  $+1$ , a **single right rotation** (with  $P$  as the root) is needed (Left-Left case).□
- 
- If the balance factor of  $L$  is  $-1$ , two different rotations are needed. The first rotation is a **left rotation** with  $L$  as the root. The second is a **right rotation** with  $P$  as the root (Left-Right case).□

### ROTATIONS:

<b>Single Rotation</b>	LL		
<b>Double Rotation</b>	RL		
	LR		
	<b>Rotation name</b>	<b>Input</b>	<b>Output</b>
	RR		

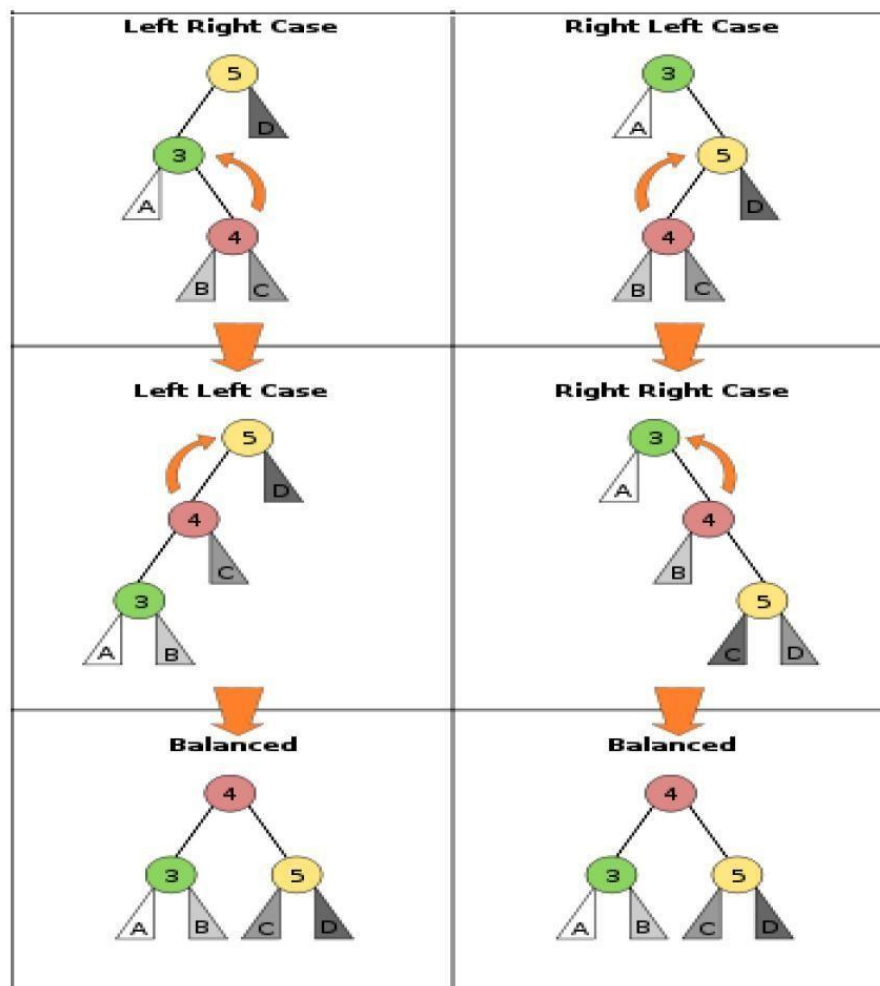
**Rotation Type**

### Single Rotation:

In this rotation, root is rotated (moved) to down level so that tree is balanced. Only one rotation is enough to make an unbalanced tree to balanced tree. This is done by rotating root (unbalanced) node to either left hand side of right hand side of the same node. Hence it can be LL or RR Rotation.

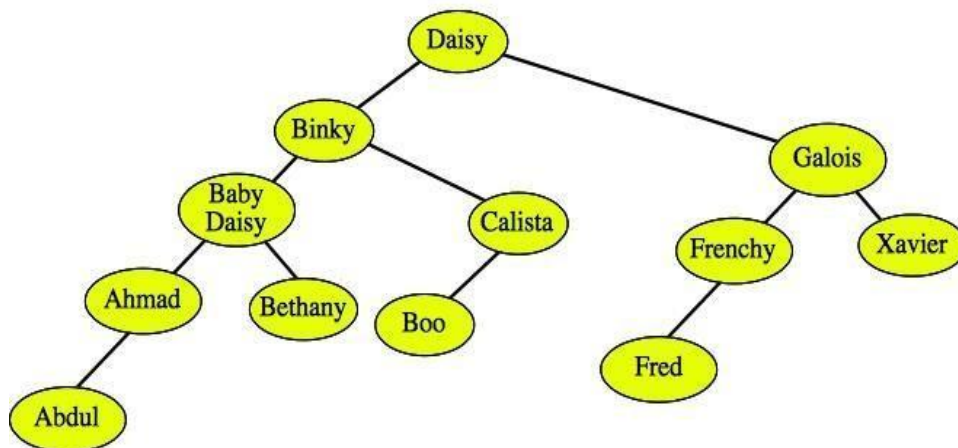
### Double Rotation:

In this rotation, root is rotated (moved) to down level so that tree is balanced. Only one rotation is **NOT** enough to make an unbalanced tree to balanced tree. This is done by rotating root (unbalanced) node to either left hand side of right hand side of the same node first then current node is again moved to left or right side. Hence it can be LR or RL Rotation.



**Example:**

**Example: of AVL TREE with character input**



```

class node
{
    //int data;
    char key[10],mean[10];
    node *left,*right;
    int ht;
};

```

```

class AVL
{
    public:
    node *root;
    node *insert(node *,char[], char[]); //Recursive counterpart of insert
    node *Delete1(node *,char[]); //Recursive counterpart of delete
    void preorder1(node *); //Recursive counterpart of preorder void
    inorder1(node *); //Recursive counterpart of inorder

    node *rotateright(node *);
    node *rotateleft(node *);
    node *RR(node *); node
    *LL(node *); node
    *LR(node *); node
    *RL(node *); int
    height(node *); int
    BF(node *);
    AVL()
    {
        root=NULL;
    } void create(char *x,char
    *y)
    {
        root=insert(root,x,y);
    }
    void Delete(char *x)
    {
        root=Delete1(root,x);
    }

    void levelwise(); void
    makenull()
    {
        root=NULL;
    }
};

```

## OPERATIONS ON AVL TREE

1. **Construct AVL Tree** /\* Psuedocode Algorithm Create \*/
2. **Display AVL** /\* Psuedocode Algorithm Display\*/
3. **Insert into AVL tree** /\* Psuedocode Algorithm Insert\*/
4. **Exit.**
5. **Enter your option**

**Algorithm Create ( )** // This is used to Create AVL tree with Data Element “x” as input and return new //root.

1. {
2. root=insert(root,x);
3. }

**Algorithm insert( )**

// This is used to Insert AVL tree with Data Element “Keyword, Meaning and Existing root ( T ) as input and return new root.

1. {
2. if(T==NULL)
3. {
4. T=new node;
5. strcpy(T->key,k);
6. strcpy(T->mean,m);
7. T->left=NULL;
8. T->right=NULL;
9. }
10. else
11. {
12. for(int i=0;i<=strlen(k);i++)
13. {
14. if(k[i] > T->key[i]) // insert in right subtree
15. {
16. T->right=insert(T->right,k,m);
17. if(BF(T)==-2)
18. if(k [i] >T->right->key[i])
19. T=RR(T); 20. else
21. T=RL(T);
22. break;

```

23.                                     }
24.                                     else
25.                                     //if(x<T->key)
26.                                     {
27.                                     T->left=insert(T->left,k,m);
28.                                     if(BF(T)==2)
29.                                     if(k[i] < T->left->key[i])
30.                                     T=LL(T); 31.         else
32.         T=LR(T);
33.         break;
34.     }      35.     }      36.     }
37.         T->ht=height(T);
38.         return(T);
39.     }

```

**Algorithm height ( )** //This is used to find height of current root (T) and it returns the height.

```
1.      {
2.      if(T==NULL)
3.      return(0);
4.      if(T->left==NULL)
5.      lh=0;
6.      else
7.      lh=1+T->left->ht;
8.      if(T->right==NULL)
9.      rh=0;
10.     else
11.     rh=1+T->right->ht;
12.     if(lh>rh)
13.     return(lh);
14.     else
15.     return(rh);
16.     }
```

**Algorithm rotateright ( )** //This is used to rotate present root (x) to its right side and returns new root.

```
1.      {
2.      y=x->left;
3.      x->left=y->right;
4.      y->right=x;
5.      x->ht=height(x);
6.      y->ht=height(y);
7.      return(y);
8.      }
```

**Algorithm rotateleft ( )** //This is used to rotate present root (x) to its left side and returns new root.

```
1.      {
2.      node *y;
3.      y=x->right;
4.      x->right=y->left;
5.      y->left=x;
6.      x->ht=height(x);
7.      y->ht=height(y);
8.      return(y);
}
```



9. }

**Algorithm RR ( )** //This is used to apply RR rotation to present unbalanced root (T) and returns new root.

```
1. {
2.   T=rotateleft(T);
3.   return(T);
4. }
```

**Algorithm LL ( )** //This is used to apply LL rotation to present unbalanced root (T) and returns new root .

```
1. {
2.   T=rotateright(T);
3.   return(T);
4. }
```

**Algorithm LR ( )** //This is used to apply LR rotation to present unbalanced root (T) and returns new root.

```
1. {
2.   T->left=rotateleft(T->left);
3.   T=rotateright(T);
4.   return(T);
5. }
```

**Algorithm RL ( )** //This is used to apply RL rotation to present unbalanced root (T) and returns new root.

```
1. {
2.   T->right=rotateright(T->right);
3.   T=rotateleft(T);
4.   return(T);
5. }
```

**Algorithm BF ( )** //This is used to find balancing factor of current root (T) and it returns the BF.

```
1. {
2.   int lh,rh;
3.   if(T==NULL)
4.     return(0);
5.   if(T->left==NULL)
6.     lh=0;
7.   else
8.     lh=1+T->left->ht;
```

```
9.         if(T->right==NULL)
10.        rh=0;
11.        else
12.        rh=1+T->right->ht;
13.        return(lh-rh);
14.    }
```

Department of Computer

Engineering, SITS

---

**Algorithm Inorder (temp) // This is used to display the nodes in LVR way**

```
1.    {
2.    if(temp is not NULL) then
3.    {
4.    Call Inorder(temp->lptr);
5.    Write (temp->data);
6.    Call Inorder(temp->rptr);
7.    }
8.    }
```

**Algorithm Preorder (temp) // This is used to display the nodes in VLR way**

```
1.    {
2.    if(temp is not NULL) then
3.    {
4.    Write (temp->data);
5.    Call Preorder(temp->lptr);
6.    Call Preorder(temp->rptr);
7.    }
8.    }
```

**Algorithm Delete(char \*x) //This is used to delete the node from tree.**

```
1.    {
2.    root=Delete1(root,x);
3.    }
```

**Algorithm Delete1( )**

```
1.    {
2.    if(T==NULL)
3.    {
4.    return NULL;
5.    }
```

6.	Else
7.	{
8.	for(int i=0;i<strlen(x);i++)
9.	if(x[i] > T->key[i]) // insert in right subtree
10.	{
11.	T->right>Delete1(T->right,x);
12.	if(BF(T)==2)
13.	if(BF(T->left)>=0)
14.	T=LL(T); 15. Else
16.	T=LR(T);
17.	break;
18.	}

19.

Else

20.

if( $x[i] < T \rightarrow \text{key}[i]$ )

---

```

21.      {
22.      T->left=Delete1(T->left,x);
23.      if(BF(T)==-2)//Rebalance during
        windup
24.      if(BF(T->right)<=0)
25.      T=RR(T); 26. Else
                T=RL(T);
                break;
                }
                Else
                {
27.                //key to be
28.                deleted is found
29.                if(T->right
30.                !=NULL)
31.                { //delete its
32.                inorder
33.                successor
34.                p=T->right;
35.                while(p->left !=
36.                NULL)
37.                p=p->left;
38.                strcpy(T-
39.                >key,p->key);
                T-
                >right=Delete1(
                T->right,p-
                >key);
40.                if(BF(T)==2)//
                Rebalance
                during windup
41.                if(BF(T-
                >left)>=0)
42.                T=LL(T); 43.
                        Else
44.                T=LR(T);
45.                }
46.                Else
47.                return(T->left);
48.                break;
49.                }

```

```
50.                                     }  
51.                                     T->ht=height(T);  
52.                                     return(T);  
53.                                     }
```

***Frequently Asked Questions:***

- 1) What is AVL tree?
- 2) How many rotations are perform to make AVL tree balanced? 3)  
What is LL & RR rotation?
- 4) What is LR & RLrotation?
- 5) What are the difference between AVL tree and B tree?
- 6) What is the time complexity of AVL tree in worst case?
- 10) AVL tree's height is strictly less than?

**Flowchart:**

Department of Computer Engineering, SITS

-----

-----

**Conclusion:**