N o. EXPERIMENT : _____

PG. NO. | DATE

Assignment - 8

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 comparisions may require for finding any keyword. Use Height Balance Tree and find the complexity for finding a keyword.

Objective -

1. To understand concept of height balanced tree or AVL tree.
2. To understand why we need height balanced trees.

Outcome -

1. Use of AVL Tree.
2. Reduced no. of comparisons for searching elements.

Theory -

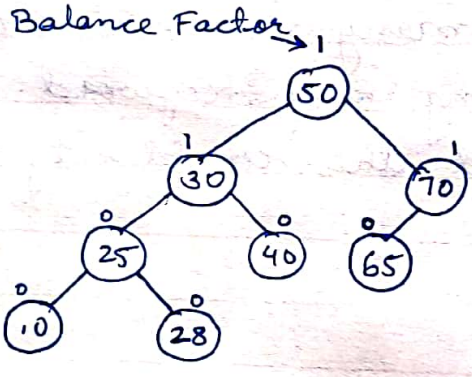
AVL tree is a height - balanced tree where the difference between heights of left and right subtree cannot be more than one or -1 for all nodes.

Most of the BST operations eg., search, max, min, insert, delete, etc take O(n) time where n is the height of BST. The cost of these

operations may become O(n) for a skewed binary tree.

If we make sure that height of the tree remains O(log n) after every insertion and deletion, then we guarantee an upper bound of O(log n) for all these operations.

The height of an AVL tree is always O(log n) where n is the no. of nodes in the tree.



Class Structure -

```
class Node {
        string key, meaning;
        Node * left, * right;
};
class AVL {
        Node * root;
public:  AVL()
        { root = NULL; }
};
```

friend class AVL;

Pseudo code-

```
Node * insert (Node *t, string s1, string s2)
{   if ( t == NULL)
    { t = new Node (s1, s2);
      return t;
    }
    if ( t -> key > s1) {
    t -> left = insert ( t -> left , s1, s2);
    t = checkrotate (t);
    return t;                }
    else if ( t -> key < s1) {
      t -> right = insert ( t -> right , s1, s2);
      t = checkrotate (t);
      return t;
    }
    else {      cout << " cannot insert ! " ;
                return t;
    }
}
int height (Node * t) {
    if ( t == NULL)
        return 0;
    int hl = 0 , hr = 0;
    hl = height ( t -> left);
    hr = height ( t -> right);
    if ( hl > hr)
        return 1 + hl;
    else
        return 1 + hr;
}
```

Teacher's Sign : _____

3. int balance (Node * t) {

```
int hl = 0, hr = 0;
  if ( t == NULL) {
      return 0; }
hl = height (t -> left) ;
hr = height ( t -> right);
return ( hl - hr);
}
```

4. Node * checkrotate (Node * t) {

```
if ( balance (t) == 2)
{   if ( balance ( t -> left)== 0 || balance (t -> left) ==1)
        t = Rightrotate ( t);
    else
        t = LR (t);
}
if ( balance (t) == -2)
{ if ( balance (t -> right) == -1 || balance (t -> right) ==0)
        t = Leftrotate ( t);
  else
        t = RL ( t);
}
return t;
}
```

5. Node * Rightrotate (Node * t)

```
Node * p = t -> left;
t -> left = p -> right;
p -> right = t;
return p;
}
```

6.
```
Node * Leftrotate (Node * t) {
    Node * p2 = t -> right;
    t -> right = p2 -> left
    p2 -> left = t;
    return p2;
}
```

7.
```
Node * LR (Node * t) {
    t -> left = Leftrotate (t -> left);
    t = Rightrotate (t);
    return t;
}
```

8.
```
Node * RL (Node * t) {
    t -> right = Rightrotate (t -> right);
    t = Leftrotate (t);
    return t;
}
```

9.
```
Node * deleterec (Node * t, string s) {
    if ( t == NULL)
        return NULL;
    if ( t -> key > s) {
        t -> left = deleterec ( t -> left);
        t = checkrotate (t);
        return t;
    }
    else if ( t -> key < s) {
        t -> right = deleterec (t -> right);
        t = checkrotate (t);
        return t;
    }
```

Teacher's Sign :

```
if ( t -> right == NULL && t -> left == NULL)
{ delete t;
  return NULL;  }
if ( t -> right == NULL) {
   Node * fz = t -> left;
     delete t;
     return fz;
}
 if ( t -> left == NULL) {
     Node * fz = t -> right;
     delete t;
     return fz;
 }
Node * fz = findmin (t -> right);
 t -> key = fz -> key;
 t -> meaning = fz -> meaning;
 t -> right = deleterec (t -> right, fz -> key);
 t = checkrotate (t);
  return t;
}
```

10.
```
void updaterec (Node * t, string s1, string s2)
{   if (t == NULL)
        return;
    if (t -> key > s1)
        updaterec (t -> ~~right~~ left, s1, s2);
    else if ( t -> key < s1)
        updaterec (t -> right, s1, s2);
    cout << " key found!";
     t -> meaning = s2;
     return;
}
```

Test cases -

Case	Expected Output				Actual O/P
Insert -	Key	Meaning	B.F	Height	
f, 6	c	3	0	0	Success
c, 3	root→f	6	1	1	
Insert -	a	1	0	0	
a, 1	root→c	3	0	1	Success
	f	6	0	0	
Update -	a	1	0	0	
f, 5+1	root→c	3	0	1	Success
	f	5+1	0	0	
Delete -	a	1	0	0	Success
c	root→f	5+1	1	1	

Conclusion -

We successfully implemented height balanced tree (AVL tree) and learnt out comparison optimization.