Title:

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.

Objectives:

1. To understand concept of height balanced tree data structure.
2. To understand procedure to create height balanced tree.

Learning Objectives:

- ✓ To understand concept of height balanced tree data structure.
- ✓ To understand procedure to create height balanced tree.

Learning Outcome:

- Define class for AVL using Object Oriented features.
- Analyze working of various operations on AVL Tree .

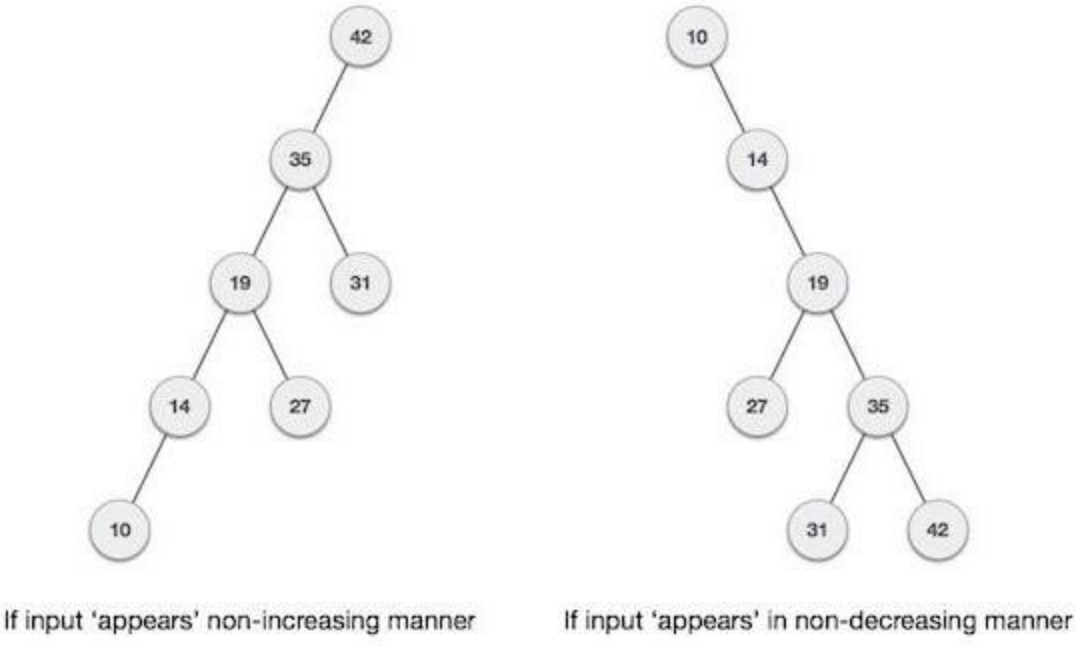
Software Required: g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Theory:

An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

AVL (Adelson- Velskii and Landis) Tree: A balance binary search tree. The best search time, that is O (log N) search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1.

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



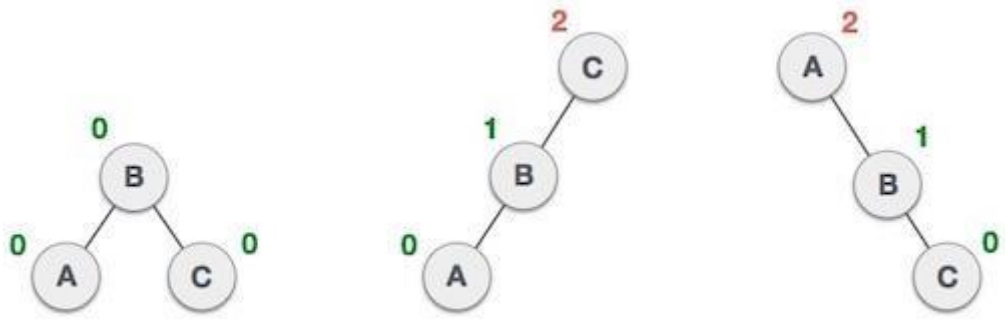
If input 'appears' non-increasing manner

If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced −



Balanced

Not balanced

Not balanced

In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

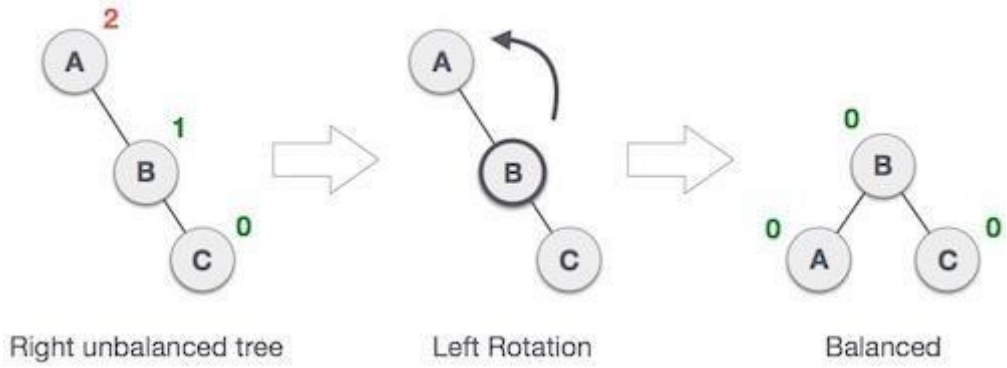
To balance itself, an AVL tree may perform the following four kinds of rotations −

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

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

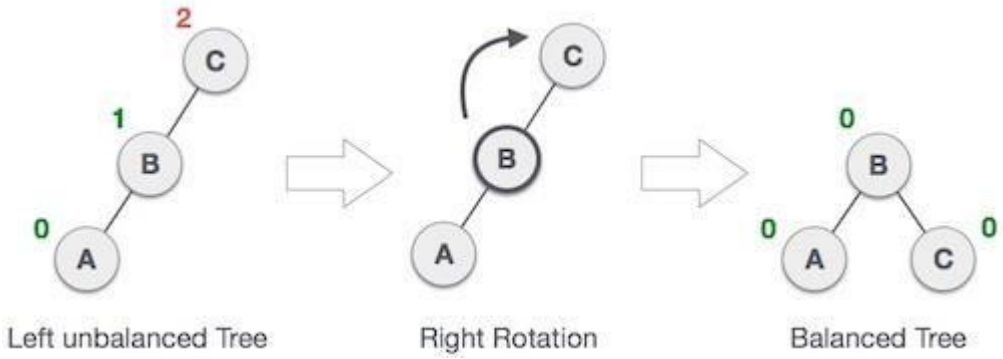
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



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

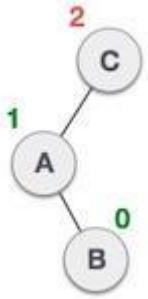
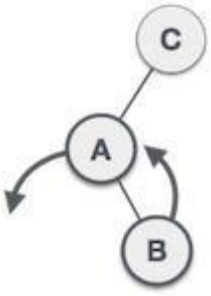
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

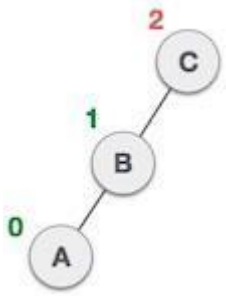
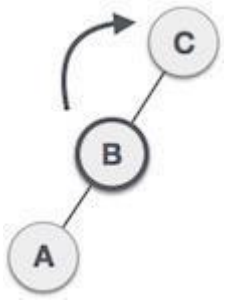
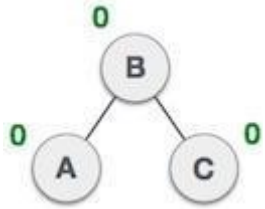


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

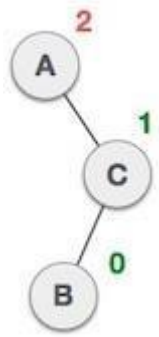
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right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

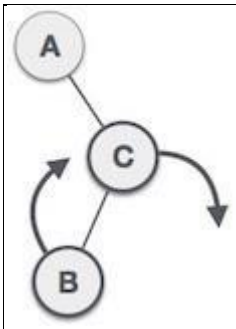
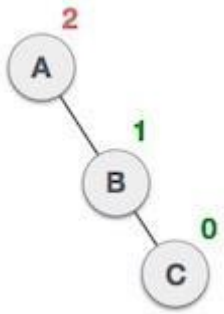
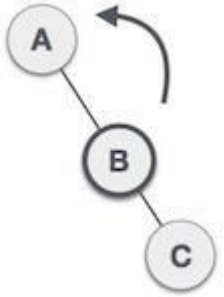
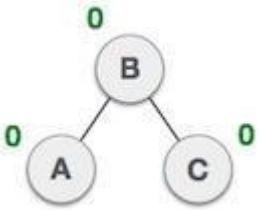
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.

	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

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

State	**Action**
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.

	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Algorithm AVL TREE:

Insert:-

1. If P is NULL, then
    I. P = new node
    II. P ->element = x
    III. P ->left = NULL
    IV. P ->right = NULL
    V. P ->height = 0
2. else if x>1 => x<P ->element
    a.) insert(x, P ->left)

b.) if height of P->left -height of P ->right =2

1. insert(x, P ->left)
2. if height(P ->left) -height(P ->right) =2

if x<P ->left ->element

P =singlerotateleft(P)

else

P =doublerotateleft(P)

3. else

if x<P ->element

a.) insert(x, P -> right)

b.) if height (P -> right) -height (P ->left) =2

if(x<P ->right) ->element

P =singlerotateright(P)

else

P =doublerotateright(P)

4. else

Print already exits

5. int m, n, d.

6. m = AVL height (P->left)
7. n = AVL height (P->right)
8. d = max(m, n)
9. P->height = d+1
10. Stop

RotateWithLeftChild(AvlNode k2)

- AvlNode k1 = k2.left;
- k2.left = k1.right;
- k1.right = k2;
- k2.height = max(height(k2.left), height(k2.right)) + 1;
- k1.height = max(height(k1.left), k2.height) + 1;
- return k1;

RotateWithRightChild(AvlNode k1)

- AvlNode k2 = k1.right;
- k1.right = k2.left;
- k2.left = k1;
- k1.height = max(height(k1.left), height(k1.right)) + 1;
- k2.height = max(height(k2.right), k1.height) + 1;
- return k2;

doubleWithLeftChild(AvlNode k3)

- k3.left = rotateWithRightChild(k3.left);
- return rotateWithLeftChild(k3);

doubleWithRightChild(AvlNode k1)

- k1.right = rotateWithLeftChild(k1.right);
- return rotateWithRightChild(k1);

Conclusion: This program gives us the knowledge height balanced binary tree.

```
#include<conio.h>
        #include<stdio.h>
        #include<iostream.h>
        #include<string.h>
        #include<fstream.h>
        struct word
         {
          char key[15];
            char meaning[20];
          };
                struct node
                {
                word data;
                node*left,*right;
                int ht;
                };
                class AVL
                {
                node *AVLroot;
                node *insert1(node*,word);
                void preorder1(node*);
                void inorder1(node*);
                node*rotateright(node*);
                node*rotateleft(node*);
                node*RR(node*);
                node*LL(node*);
                node*LR(node*);
                node*RL(node*);
                int BF(node*);
                int height(node*T);
                public:
                AVL()
                {
                AVLroot=NULL;
```

```
}
void insert(word x){AVLroot=insert1(AVLroot,x);}
void preorder(){preorder1(AVLroot);}
void inorder(){inorder1(AVLroot);}
void levelwise();
void search(word x);
void makenull()
{
AVLroot=NULL;
}
};
void main()
{
AVL A;
int n,i,op;
word x;
ifstream f1;
f1.open("indata.txt",ios::in);
do
{
cout<<"\n1)create:";
cout<<"\n2)search:";
cout<<"\n3)print:";
cout<<"\n4)quit:";
cout<<"\n Enter your choice:";
cin>>op;
switch(op)
{
        case 1:A.makenull();
        while(!f1.eof())
        {
        f1>>x.key>>x.meaning;
        A.insert(x);
        }
        break;
        case 2:cout<<"\n Enter a data:";
        cin>>x.key;
        A.search(x);
        break;
        case 3:cout<<"\npreorder sequence:\n";

        A.preorder();
        cout<<"\nInorder sequence:\n";
```

```
A.inorder();
break;
}
}while(op!=4);
}
void AVL::search(word x)
{
node*T=AVLroot;
while(T!= NULL)
{
if(strcmp(x.key,T->data.key)==0)
{
cout<<"Meaning is:"<<T->data.meaning;
return;
}
if(strcmp(x.key,T->data.key)>0)
T=T->right;
else
T=T->left;
}
cout<<"\n key not found:";
}
node*AVL::insert1(node*T,word x)
{
if(T==NULL)
{
T=new node;
T->data=x;
T->left=NULL;
T->right=NULL;
}
else
        if(strcmp(x.key,T->data.key)>0)
        {
                T->right=insert1(T->right,x);
                if(BF(T)==-2)
                        if(strcmp(x.key,T->right->data.key)>0)
                        T=RR(T);

                        else T=RL(T);
        }
else
        if(strcmp(x.key,T->data.key)<0)
        {
```

```
                        T->left=insert1(T->left,x);
        if(BF(T)==2)
                if(strcmp(x.key,T->left->data.key)<0)
                T=LL(T);
        else
        T=LR(T);
        }
        T->ht=height(T);
        return(T);
}


int AVL::height(node*T)
{
int lh,rh;
if(T==NULL)
return(0);
if(T->left==NULL)
lh=0;
else
lh=1+T->left->ht;
if(T->right==NULL)
rh=0;
else
rh=1+T->right->ht;
if(lh>rh)
return(lh);
return(rh);
}
node * AVL ::rotateright(node *x)
{
node * y;
y=x->left;
x->left=y->right;
y->right=x;
x->ht=height(x);

x->ht=height(y);
   return(y);
}
node * AVL ::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);
}
node * AVL ::RR(node *T)
{
T=rotateleft(T);
return(T);
}
  node * AVL ::LL(node *T)
{
T=rotateright(T);
return(T);
}
node * AVL ::LR(node *T)
{
T->left=rotateleft(T->left);
T=rotateright(T);
return(T);
}
node * AVL ::RL(node *T)
{
T->left=rotateright(T->left);
T=rotateleft(T);
return(T);
}
int AVL::BF(node *T)
{
int lh,rh;
if(T==NULL)
return(0);

if(T->left==NULL)
lh=0;
else
lh=1+T->left->ht;
if(T->right==NULL)
rh=0;
else
rh=1+T->right->ht;
return(lh-rh);
}
```

```
void AVL::preorder1( node *T)
{
if(T!=NULL)
{
cout<<"("<<T->data.key<<"-"<<T->data.meaning<<")";
preorder1(T->left);
preorder1(T->right);
}
}
void AVL::inorder1(node*T)
{
if(T!=NULL)
{
inorder1(T->left);
cout<<"("<<T->data.key<<"---"<<T->data.meaning<<")";
inorder1(T->right);
}
}
```