

**EX NO:7****IMPLEMENTATION OF AVL TREE****AIM:-**

To write a C program to implement insertion in AVL trees.

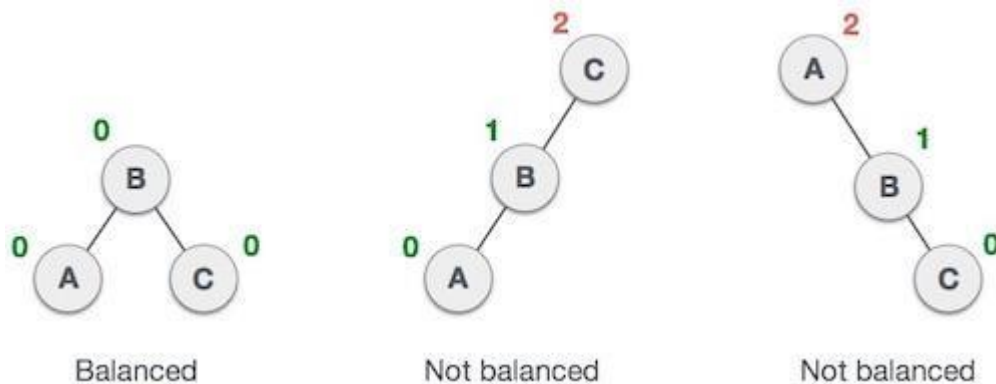
**ALGORITHM:-**

- 1.Initialize all variables and functions.
- 2.To insert and element read the value.
- 3.Check whether root is null
- 4.If yes make the new value as root.
- 5.Else check whether the value is equal to root value. 6.if yes, print “duplicate value”.
- 7.Otherwise insert the value at its proper position and balance the tree using rotations.
- 8.To display the tree values check whether the tree is null.
- 9.If yes, print “tree is empty”.
- 10.Else print all the values in the tree form and in order of the tree.
- 11.Repeat the steps 2 to 10 for more values.
- 12.End

**DESCRIPTION:**

**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 –



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-subtree) – height(right-subtree)

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 –

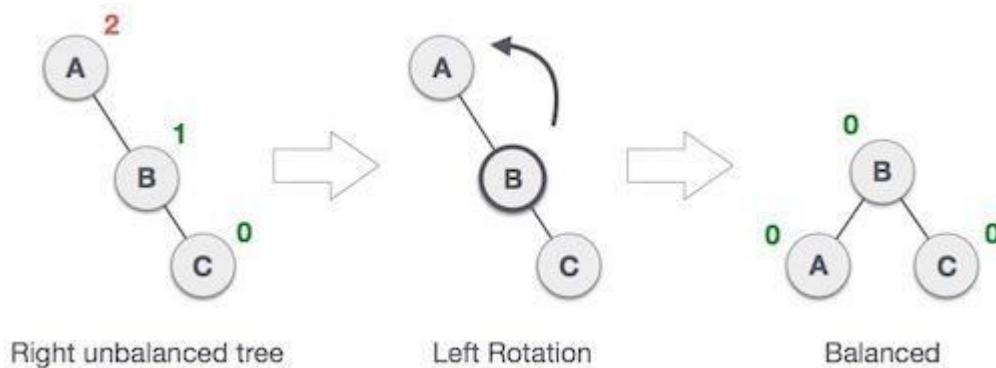
- Left rotation

- 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

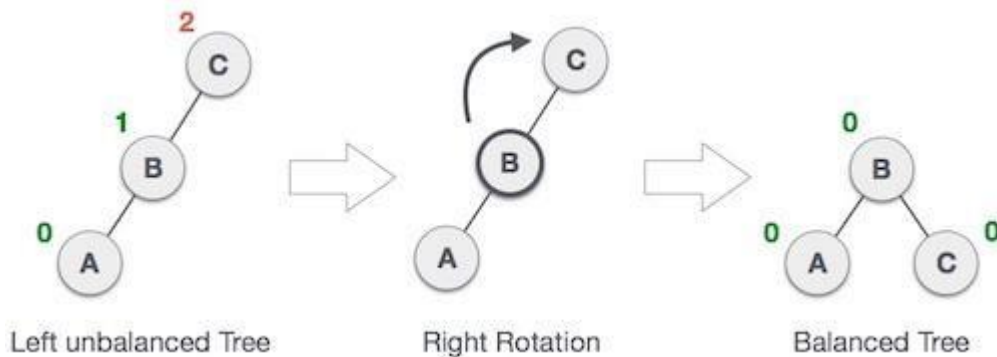
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



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.



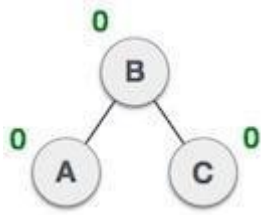
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

### *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.

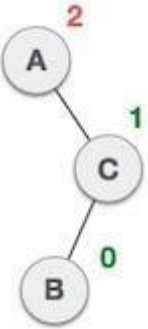
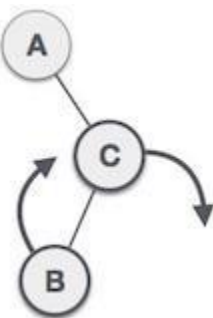
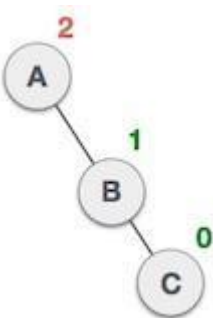
State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes <b>C</b> an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of <b>C</b>. This makes <b>A</b>, the left subtree of <b>B</b>.</p>
	<p>Node <b>C</b> is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making <b>B</b> the new root node of this subtree. <b>C</b> now becomes the right subtree of its own left subtree.</p>

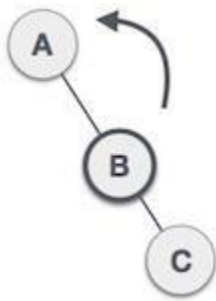
The tree is now balanced.



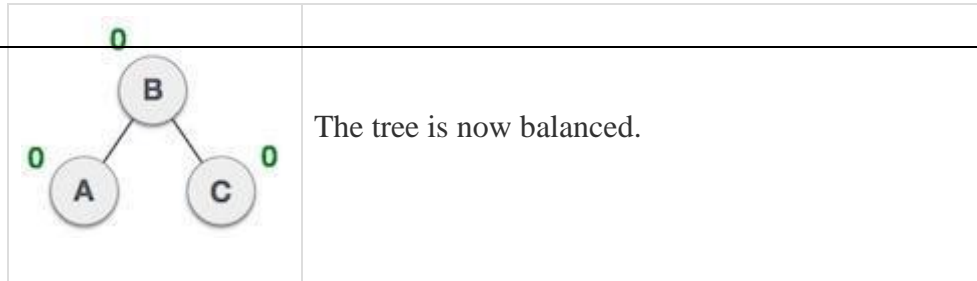
### Right-Left Rotation

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

State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes <b>A</b>, an unbalanced node with balance factor 2.</p>
	<p>First, we perform the right rotation along <b>C</b> node, making <b>C</b> the right subtree of its own left subtree <b>B</b>. Now, <b>B</b> becomes the right subtree of <b>A</b>.</p>
	<p>Node <b>A</b> is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>



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**.



### PROGRAM

```
#include<stdio.h>

#include<malloc.h>

typedef enum { FALSE ,TRUE } bool;

struct node
{
    int info;
    int balance;
    struct node *lchild;
    struct node *rchild;
};

struct node *insert (int , struct node *, int *);
struct node* search(struct node *,int);

main()
{
    bool ht_inc;
    int info ;
    int choice;

    struct node *root = (struct node *)malloc(sizeof(struct node));
    root = NULL;

    while(1)
    {
        printf("1.Insert\n");
        printf("2.Display\n");
        printf("3.Quit\n");
        printf("Enter your choice : ");
```





```
scanf("%d",&choice);  
  
switch(choice)  
{ case 1:  
printf("Enter the value to be inserted  
: "); scanf("%d", &info); if(  
search(root,info) == NULL ) root =  
insert(info, root, &ht_inc); else  
printf("Duplicate value ignored\n");  
break; case 2: if(root==NULL) {  
printf("Tree is empty\n"); continue; }  
printf("Tree is :\n"); display(root,  
1); printf("\n\n"); printf("Inorder  
Traversal is: "); inorder(root);  
printf("\n"); break; case 3: exit(1);  
default:  
printf("Wrong choice\n");  
}/*End of switch*/  
}/*End of while*/
```

```
}/*End of main()*/ struct node*
```

```

search(struct node *ptr,int info)

{ if(ptr!=NULL) if(info < ptr->info) ptr=search(ptr-
>lchild,info); else if( info > ptr->info) ptr=search(ptr-
>rchild,info); return(ptr); }/*End of search()*/ struct
node *insert (int info, struct node *pptr, int *ht_inc)
{ struct node
*aptr; struct
node *bptr;

if(pptr==NULL
)

{ pptr = (struct node *) malloc(sizeof(struct
node)); pptr->info = info; pptr->lchild =
NULL; pptr->rchild = NULL; pptr->balance = 0;
*ht_inc = TRUE; return (pptr);
} if(info <
pptr->info)

{ pptr->lchild = insert(info, pptr->lchild,
ht_inc); if(*ht_inc==TRUE)
{ switch(pptr-
>balance)
{

```

```
case -1: /* Right heavy */
```

```

pptr->balance = 0; *ht_inc =
FALSE; break; case 0: /*
Balanced */ pptr->balance = 1;
break; case 1: /* Left heavy
*/ aptr = pptr->lchild;
if(aptr->balance == 1) {
printf("Left to Left
Rotation\n"); pptr->lchild=
aptr->rchild; aptr->rchild =
pptr; pptr->balance = 0; aptr-
>balance=0; pptr = aptr; }
else { printf("Left to right
rotation\n"); bptr = aptr-
>rchild; aptr->rchild = bptr-
>lchild; bptr->lchild = aptr;
pptr->lchild = bptr->rchild;
bptr->rchild = pptr; if(bptr-
>balance == 1 ) pptr->balance
= -1; else pptr->balance = 0;
if(bptr->balance == -1) aptr-
>balance = 1; else aptr-
>balance = 0; bptr->balance=0;
pptr=bptr; }
*ht_inc = FALSE;
}/*End of switch */
}/*End of if */
}/*End of if*/

```

```

if(info > pptr->info)
{ pptr->rchild = insert(info, pptr->rchild,
ht_inc); if(*ht_inc==TRUE)
{ switch(pptr->balance)
{ case 1: /* Left
heavy */ pptr->balance = 0; *ht_inc
= FALSE; break; case
0: /* Balanced */
pptr->balance = -1;
break; case -1: /*
Right heavy */ aptr =
pptr->rchild;
if(aptr->balance == -
1)
{ printf("Right to Right
Rotation\n"); pptr->rchild=
aptr->lchild; aptr->lchild =
pptr;

```

```
pptr->balance = 0; aptr-
```



```
>balance=0; pptr = aptr; }  
  
else { printf("Right to Left  
Rotation\n"); bptr = aptr-  
>lchild; aptr->lchild = bptr-  
>rchild; bptr->rchild = aptr;  
pptr->rchild = bptr->lchild;  
bptr->lchild = pptr; if(bptr-  
>balance == -1) pptr->balance  
= 1; else pptr->balance = 0;  
if(bptr->balance == 1) aptr-  
>balance = -1; else aptr-  
>balance = 0; bptr->balance=0;  
pptr = bptr;  
  
}/*End of else*/  
  
*ht_inc = FALSE;  
  
}/*End of switch */  
  
}/*End of if*/ }/*End of if*/  
  
return(pptr); }/*End of  
insert()*/ display(struct node  
*ptr,int level)  
  
{
```

```
int i; if (
```

```

ptr!=NULL )

{ display(ptr->rchild,
level+1); printf("\n");
for (i = 0; i < level;
i++) printf(" ");
printf("%d", ptr->info);
display(ptr->lchild,
level+1);
}/*End of if*/

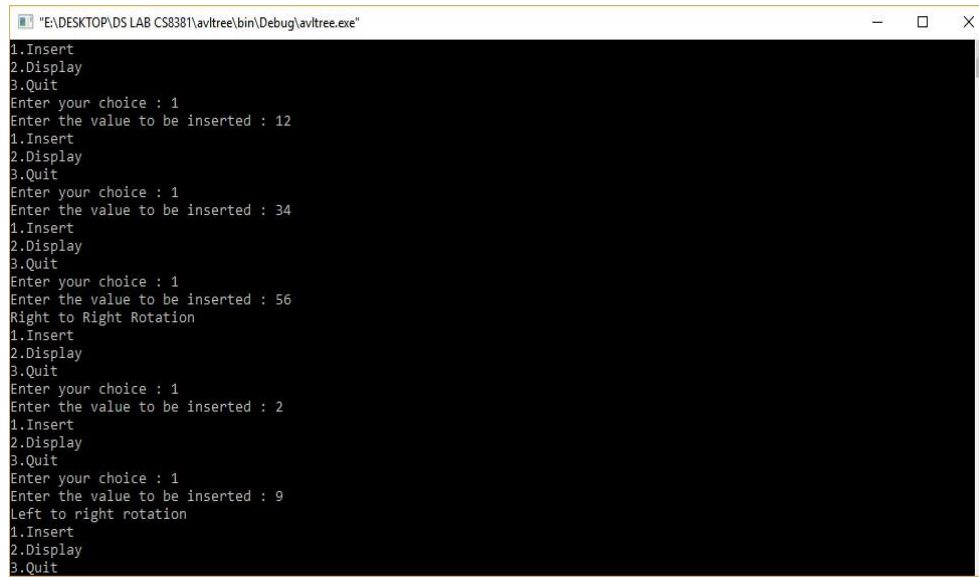
}/*End of
display()*/

inorder(struct node
*ptr)

{ if(ptr!=NULL) {
inorder(ptr->
lchild);
printf("%d ",ptr->
info);
inorder(ptr->
rchild);
}}/*End of inorder()*/

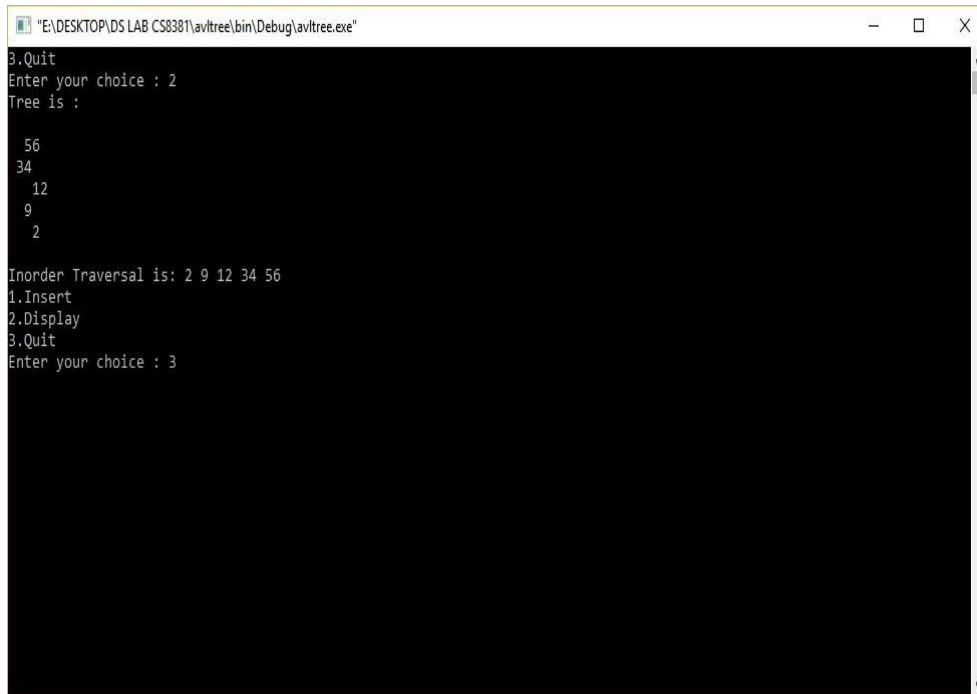
```

### OUTPUT



```
"E:\DESKTOP\DS LAB CS8381\avltree\bin\Debug\avltree.exe"
1.Insert
2.Display
3.Quit
Enter your choice : 1
Enter the value to be inserted : 12
1.Insert
2.Display
3.Quit
Enter your choice : 1
Enter the value to be inserted : 34
1.Insert
2.Display
3.Quit
Enter your choice : 1
Enter the value to be inserted : 56
Right to Right Rotation
1.Insert
2.Display
3.Quit
Enter your choice : 1
Enter the value to be inserted : 2
1.Insert
2.Display
3.Quit
Enter your choice : 1
Enter the value to be inserted : 9
Left to right rotation
1.Insert
2.Display
3.Quit
```





```
"E:\DESKTOP\DS LAB CS8381\avltree\bin\Debug\avltree.exe"
3.Quit
Enter your choice : 2
Tree is :

56
34
12
9
2

Inorder Traversal is: 2 9 12 34 56
1.Insert
2.Display
3.Quit
Enter your choice : 3
```

## **RESULT**

Thus the 'C' program to implement an AVL trees . Produce its pre-Sequence, In-Sequence, and PostSequence traversals