**AVL**

**EXPT NO: 7**  **DATES: 17/12/21**

**AIM**

**1)** Write a program to perform insertion and deletion operation in AVL Trees.

**THEORY**

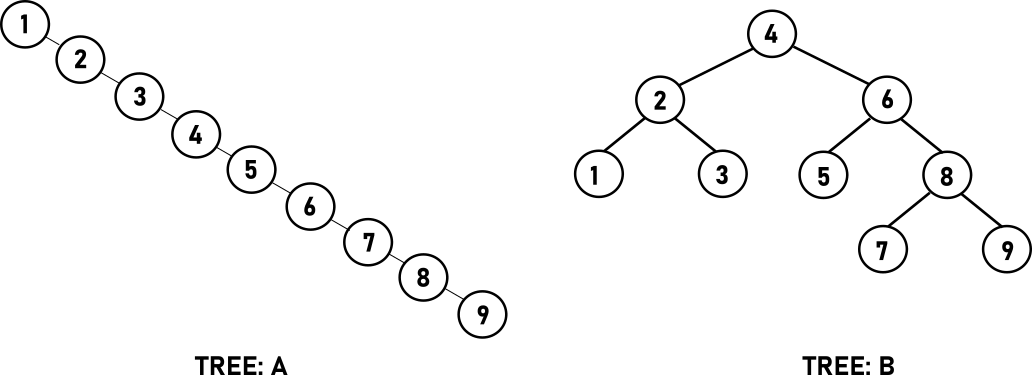
A set of data can produce different binary search trees if the sequence of insertion of elements is different.

**AVL Trees**

If we enter data in the following sequences, it will yield 2 different types of binary tree structures

**Sequence 1:** 1 2 3 4 5 6 7 8 9

**Sequence 2:** 4 2 6 1 3 5 8 7 9



Both the trees have the same data but their structures are different because of different sequence of insertion of elements. It is not possible to control the order of insertion so the concept of height balanced binary tree came in. The technique for balancing a binary search tree was introduced by Russian mathematician G. H. Adelson-Velskii and E. M Landis in 1962. The height balanced binary search tree is know as AVL Tree in their honor.

The primary aim of AVL tree is to perform efficient search, insertion and deletion operations. Searching is efficient when the height of the left and right subtrees of the nodes are almost the same. This is possible in a full or a complete binary search tree, which is an ideal situation and is not always achievable. This ideal situation is very nearly approximated by ACL trees.

An AVL tree is a binary search tree where the difference in the height of left and right subtrees of any node can be at most 1

A tree is not an AVL tree where there are two nodes for which difference in height of left and right subtree exceeds 1.

|  |
| --- |
| **Balanced factor of a Node = Height of its left subtree - Height of its right subtree** |

From the definition of AVL tree, it is obvious that only possible values from the balance factor of any node are -1, 0, 1.

|  |  |
| --- | --- |
| **TERMS** | **EXPLANATION** |
| Right Heavy | If height of its right subtree is more than height of its left subtree.  **Balanced Factor =-1** |
| Left Heavy | If height of its left subtree is more than height of its right subtree.  **Balanced Factor =1** |
| Balanced | If height of its left and right subtrees are same.  **Balanced Factor = 0** |

**Structure of AVL trees**

struct node {

struct node \*left;

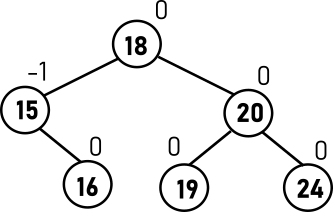
struct node \*right;

int info;

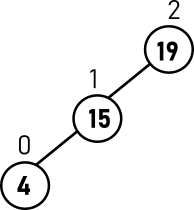
int balance;

};

**Examples of AVL Trees**



**Examples of Non AVL trees**



**Tree Rotations**

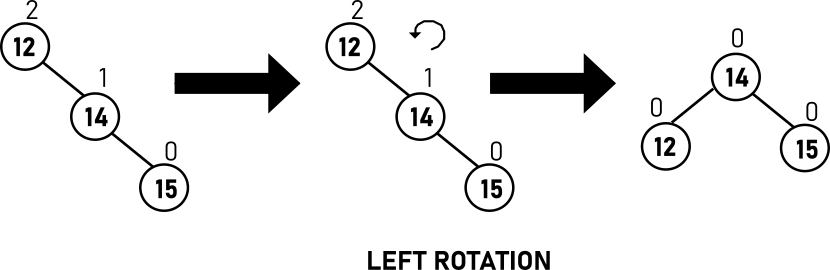
After insertion or deletion operations the balanced factor of the nodes are affected and the tree might become unbalanced. The balance of the tree is restored by performing rotations.

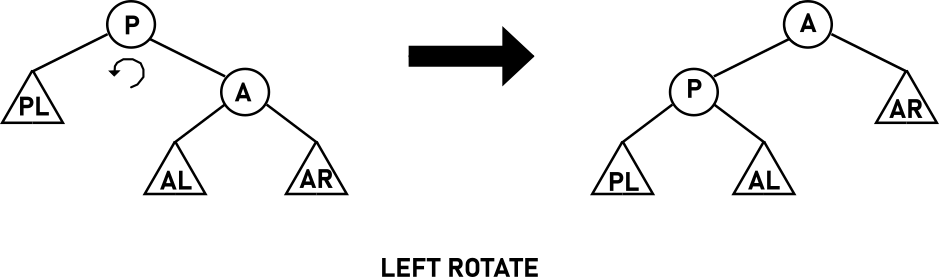
**Rotations:** They are simple manipulation of pointers which convert the tree in such a way that the knew converted tree retains binary search tree property with Inorder traversal same as that of the original tree.

**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

Node **12** has become unbalanced as a node is inserted in the right subtree of 12's right subtree. We perform the left rotation by making **12** the left-subtree of **14**.

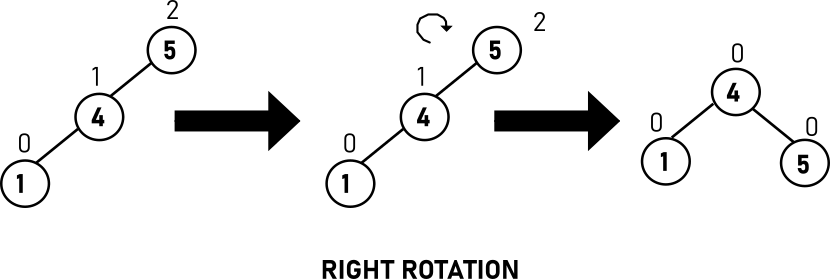


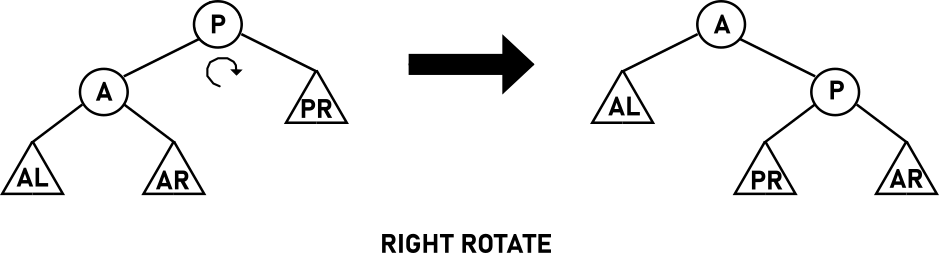


**RIGHT ROTATION**

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**

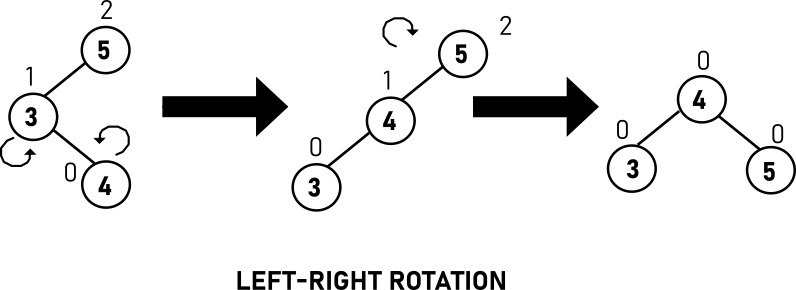
1) A node has been inserted into the right subtree of the left subtree. This makes **5** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

2) We first perform the left rotation on the left subtree of **5**. This makes **3**, the left subtree of **4**.

3) Node **5** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.

4) We shall now right-rotate the tree, making **4** the new root node of this subtree. **5** now becomes the right subtree of its own left subtree.

5) The tree is now balanced.



**RIGHT-LEFT ROTATION**

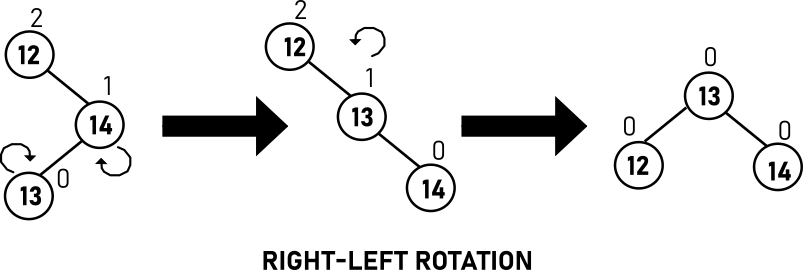
1) A node has been inserted into the left subtree of the right subtree. This makes **12**, an unbalanced node with balance factor 2.

2) First, we perform the right rotation along **14** node, making **14** the right subtree of its own left subtree **13**. Now, **13** becomes the right subtree of **12**.

3) Node **12** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.

4) A left rotation is performed by making **13** the new root node of the subtree. **12** becomes the left subtree of its right subtree **13**

5) The tree is now balanced.



**ALGORITHMS**

**struct node\*insert(struct node\*pptr,int item)**

1. if pptr==NULL

Dynamically allocate memory to node tmp

pptr->data=item

pptr->lchild=NULL

pptr->rchild=NULL

pptr->balance=0

taller=TRUE

2. if item<pptr->data

insert item in pptr->lchild by recursively calling insert() function

if taller==TRUE

a. pptr->lchild=insert(pptr->lchild,item)

3. else if item>pptr->data

pptr->rchild= insert(pptr->rchild,item)

if taller==TRUE

a. pptr=insert\_right\_check(pptr,&taller)

4. else printf "Duplicated Item!!"

1. taller=FALSE

5. return pptr

6. return pptr

**struct node\*rotateleft(struct node\*pptr)**

1. Accept pptr from function call

2. aptr=pptr->rchild

3. pptr->rchild=aptr->lchild

4. aptr->lchild=pptr

5. return aptr

**struct node\*rotateright(struct node\*pptr)**

1. Accept pptr from function call

2. aptr=pptr->lchild

3. pptr->lchild=aptr->rchild

4. aptr->rchild=pptr

5. return aptr

**struct node\*insert\_left\_check(struct node\*pptr,int\*ptaller)**

1. if pptr->balance is 0

i. pptr->balance=1

2. if pptr->balance is -1

* + - * 1. pptr->balance=0
        2. \*ptaller=FALSE

3. if pptr->balance is 1

* + - * 1. Call fuction insert\_LeftBalance() to update and balance factor about pptr by passing pptr
        2. \*ptaller=FALSE

4.return pptr

**struct node\*insert\_right\_check(struct node\*pptr,int\*ptaller)**

1. if pptr->balance is 0

i. pptr->balance=-1

2. if pptr->balance is 1

* + - * 1. pptr->balance=0
        2. \*ptaller=FALSE

3. if pptr->balance is -1

* + - * 1. Call fuction insert\_RightBalance() to update and balance factor about pptr by passing pptr
        2. \*ptaller=FALSE

4. return pptr

**struct node\*insert\_LeftBalance(struct node\*pptr)**

1.aptr=pptr->lchild

2. If aptr->balance is 1

* + - 1. pptr->balance=0
      2. aptr->balance=0
      3. Perform right rotation about pptr by

calling rotateright() function

3. else bptr=aptr->rchild

1.if bptr->balance is -1

1. pptr->balance=0
2. aptr->balance=1

2. if bptr->balance is 1

1. pptr->balance=-1
2. aptr->balance=0

3. if bptr->balance is 0

1. pptr->balance=0
2. aptr->balance=0

4. bptr->balance =0

5. Perform left rotation about aptr by calling rotateleft()

6. Perform right rotation about pptr by calling rotateright()

4. return pptr

**struct node\*del\_right\_check(struct node\*pptr,int\*pshorter)**

1. if pptr->balance is 0

* + - 1. pptr->balance=1
      2. \*pshorter=FALSE

2. if pptr->balance is -1

i. pptr->balance=0

3. if pptr->balance is 1

1. Call function del\_LeftBalance() to update and

balance factor about pptr by passing pptr and pshorter.

4. return pptr

**struct node\*del\_left\_check(struct node\*pptr,int \*pshorter)**

1. if pptr->balance is 0

* + - * 1. pptr->balance=-1
        2. \*pshorter=FALSE

2. if pptr->balance is 1

i. pptr->balance=0

3. if pptr->balance is -1

1. Call function del\_rightBalance() to update and balance factor about pptr by passing pptr and pshorter

4. return ppt

**struct node\*insert\_RightBalance(struct node\*pptr)**

1. aptr=pptr->lchild

2. If aptr->balance is -1

* + - * 1. pptr->balance=0
        2. aptr->balance=0
        3. Perform left rotation about pptr by calling rotateleft() function

3. else bptr=aptr->lchild

1. if bptr->balance is -1

* + - * 1. pptr->balance=1
        2. aptr->balance=0

2. if bptr->balance is 1

* + - * 1. pptr->balance=0
        2. aptr->balance=-1

3. if bptr->balance is 0

* + - * 1. pptr->balance=0
        2. aptr->balance=0

4. bptr->balance=0

5.Perform right rotation about aptr by calling rotateright() function

6Perform left rotation about pptr by calling rotateleft() function

4. return pptr

**struct node\*del(struct node\*pptr,int item)**

1. if pptr==NULL

* + - * 1. Output element not present in the list
        2. shorter=FALSE
        3. return pptr

2. if item<pptr->data

* + - * 1. Delete item from pptr->lchild by recursively calling del() function
        2. if shorter==TRUE

a. pptr=del\_left\_check(pptr,&shorter)

3. else if item>pptr->data

* + - * 1. Delete item from pptr->rchild by recursively calling del() function
        2. If shorter==TRUE
        3. pptr=del\_right\_check(pptr,&shorter);

4.else

1. if pptr->lchild!=NULL AND pptr->rchild!=NULL

* + - * 1. succ=pptr->rchild
        2. while succ->lchild!=NULL

a. succ=succ->lchild

iii. pptr->data=succ->data

* + - * 1. pptr->rchild=del(pptr->rchild,succ->data)
        2. If shorter==TRUE

a. pptr=del\_right\_check(pptr,&shorter)

2 else

1.tmp=pptr

2.if pptr->lchild!=NULL

i. pptr=pptr->lchild

3.else if pptr->rchild!=NULL

i. pptr=pptr->rchild

4.pptr=NULL

5.free(tmp)

6.shorter=TRUE

5.return pptr

**struct node\*del\_LeftBalance(struct node\*pptr,int\*pshorter)**

1.aptr=pptr->lchild

2. if aptr->balance is 0

* + - * 1. pptr->balance=1
        2. aptr->balance=-1
        3. \*pshorter=FALSE
        4. Perform right rotation

about pptr by calling rotateright() function

3. if aptr->balance is 1

* + - * 1. pptr->balance=0
        2. aptr->balance=0
        3. Perform right rotation

about pptr by calling rotateright() function

4.else 1. bptr=aptr->rchild

2.if bptr->balance is 0

* + - * 1. pptr->balance=0
        2. aptr->balance=0

3 if bptr->balance is 1

* + - * 1. pptr->balance=-1
        2. aptr->balance=0

4. if bptr->balance is 0-1

* + - * 1. pptr->balance=0
        2. aptr->balance=1

5.bptr->balance=0

6.Perform left rotation about aptr by calling rotateleft() function

7.Perform right rotation about pptr by calling rotateright() function

5. return pptr

**struct node\*del\_RightBalance(struct node\*pptr,int\*pshorter)**

1. aptr=pptr->rchild

2. if aptr->balance =0

pptr->balance=-1

aptr->balance=1

\*pshorter=FALSE

Perform left rotation about pptr by calling rotateleft() function

3. if aptr->balance=-1

pptr->balance=0

aptr->balance=0

Perform left rotation about pptr by calling rotateleft() function

4.else 1bptr=aptr->lchild

2. if bptr->balance is 0

pptr->balance=0

aptr->balance=0

3.if bptr->balance is 1

pptr->balance=0

aptr->balance=-1

4.if bptr->balance is -1

pptr->balance=1

aptr->balance=0

5. bptr->balance=0

6. Perform right rotation about aptr by calling rotateright() function

7. Perform left rotation about pptr by calling rotateleft() function

11. return pptr.

**CODES**

#include<stdio.h>

#include<stdlib.h>

#define CNT 10

#define TRUE 1

#define FALSE 0

struct node

{

int data;

struct node\* left;

struct node\* right;

int balance;

};

struct node\* rightRotate(struct node\* pptr);

struct node\* leftRotate(struct node\* pptr);

struct node\* insert(struct node\* p, int data);

struct node\* insert\_left\_check(struct node\* pptr, int \*taller);

struct node\* leftbalance(struct node\* pptr);

struct node\* insert\_right\_check(struct node\* pptr, int \*taller);

struct node\* rightbalance(struct node\* pptr);

struct node\* del(struct node\* p, int data);

struct node\* del\_left\_check(struct node\* pptr, int \*shorter);

struct node\* del\_leftbalance(struct node\* pptr, int \*shorter);

struct node\* del\_right\_check(struct node\* pptr, int \*shorter);

struct node\* del\_rightbalance(struct node\* pptr, int \*shorter);

void display(struct node \*root, int space);

int main()

{

struct node \*root=NULL;

int ch, data;

do

{

printf("Press 1. INSERT NODE\n");

printf("Press 2. DELETE A NODE\n");

printf("Press 3. DISPLAY\n");

printf("Press 4. EXIT\n");

printf("ENTER YOUR CHOICE: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("ENTER THE DATA: ");

scanf("%d",&data);

root=insert(root, data);

break;

case 2:

printf("ENTER THE DATA: ");

scanf("%d",&data);

root=del(root, data);

break;

case 3:

display(root, 0);

break;

case 4:

break;

default:

printf("INVALID INPUT");

}

printf("\n\n");

} while(ch!=4);

return 0;

}

struct node\* rightRotate(struct node\* pptr)

{

struct node\* aptr=pptr->left;

pptr->left=aptr->right;

aptr->right=pptr;

return aptr;

}

struct node\* leftRotate(struct node\* pptr)

{

struct node\* aptr=pptr->right;

pptr->right=aptr->left;

aptr->left=pptr;

return aptr;

}

struct node\* insert(struct node\* p, int data)

{

static int taller;

if(p==NULL)

{

p=(struct node\*)malloc(sizeof(struct node));

p->data=data;

p->left=NULL;

p->right=NULL;

p->balance=0;

taller=TRUE;

}

else if(p->data > data)

{

p->left=insert(p->left, data);

if(taller==TRUE)

p=insert\_left\_check(p, &taller);

}

else if(p->data < data)

{

p->right=insert(p->right, data);

if(taller==TRUE)

p=insert\_right\_check(p, &taller);

}

else

{

printf("DUPLICATE DATA\n");

taller=FALSE;

}

return p;

}

struct node\* insert\_left\_check(struct node\* pptr, int \*taller)

{

switch(pptr->balance)

{

case 0:

pptr->balance=1;

break;

case -1:

pptr->balance=0;

\*taller=FALSE;

break;

case 1:

pptr=leftbalance(pptr);

\*taller=FALSE;

break;

}

return pptr;

}

struct node\* leftbalance(struct node\* pptr)

{

struct node \*aptr=pptr->left, \*bptr;

if(aptr->balance==1)

{

aptr->balance=0;

pptr->balance=0;

pptr=rightRotate(pptr);

}

else

{

bptr=aptr->right;

switch (bptr->balance)

{

case -1:

aptr->balance=1;

pptr->balance=0;

break;

case 1:

aptr->balance=0;

pptr->balance=-1;

break;

case 0:

aptr->balance=0;

pptr->balance=0;

break;

}

bptr->balance=0;

pptr->left=leftRotate(aptr);

pptr=rightRotate(pptr);

}

return pptr;

}

struct node\* insert\_right\_check(struct node\* pptr, int \*taller)

{

switch(pptr->balance)

{

case 0:

pptr->balance=-1;

break;

case 1:

pptr->balance=0;

\*taller=FALSE;

break;

case -1:

pptr=rightbalance(pptr);

\*taller=FALSE;

break;

}

return pptr;

}

struct node\* rightbalance(struct node\* pptr)

{

struct node \*aptr=pptr->right, \*bptr;

if(aptr->balance==-1)

{

aptr->balance=0;

pptr->balance=0;

pptr=leftRotate(pptr);

}

else

{

bptr=aptr->left;

switch (bptr->balance)

{

case -1:

aptr->balance=0;

pptr->balance=1;

break;

case 1:

aptr->balance=-1;

pptr->balance=0;

break;

case 0:

aptr->balance=0;

pptr->balance=0;

break;

}

bptr->balance=0;

pptr->right=rightRotate(aptr);

pptr=leftRotate(pptr);

}

return pptr;

}

struct node\* del(struct node\* p, int data)

{

static int shorter;

struct node\* t;

if(p==NULL)

{

printf("DATA NOT FOUND\n");

shorter=FALSE;

}

else if(p->data > data)

{

p->left=del(p->left, data);

if(shorter==TRUE)

p=del\_left\_check(p, &shorter);

}

else if(p->data < data)

{

p->right=del(p->right, data);

if(shorter==TRUE)

p=del\_right\_check(p, &shorter);

}

else

{

if(p->left!=NULL && p->right!=NULL)

{

t=p->right;

while(t->left!=NULL)

t=t->left;

p->data=t->data;

p->right=del(p->right, t->data);

if(shorter==TRUE)

p=del\_right\_check(p, &shorter);

}

else

{

t=p;

if(p->left!=NULL)

p=p->left;

else if(p->right!=NULL)

p=p->right;

else

p=NULL;

shorter=TRUE;

free(t);

}

}

return p;

}

struct node\* del\_left\_check(struct node\* pptr, int \*shorter)

{

switch(pptr->balance)

{

case 0:

pptr->balance=-1;

\*shorter=FALSE;

break;

case 1:

pptr->balance=0;

break;

case -1:

pptr=del\_rightbalance(pptr, shorter);

break;

}

return pptr;

}

struct node\* del\_rightbalance(struct node\* pptr, int \*shorter)

{

struct node \*aptr=pptr->right, \*bptr;

if(aptr->balance==0)

{

aptr->balance=1;

pptr->balance=-1;

\*shorter=FALSE;

pptr=leftRotate(pptr);

}

else if(aptr->balance==-1)

{

aptr->balance=0;

pptr->balance=0;

pptr=leftRotate(pptr);

}

else

{

bptr=aptr->left;

switch(bptr->balance)

{

case 0:

aptr->balance=0;

pptr->balance=0;

break;

case 1:

aptr->balance=-1;

pptr->balance=0;

break;

case -1:

aptr->balance=0;

pptr->balance=1;

}

bptr->balance=0;

pptr->right=rightRotate(aptr);

pptr=leftRotate(pptr);

}

return pptr;

}

struct node\* del\_right\_check(struct node\* pptr, int \*shorter)

{

switch(pptr->balance)

{

case 0:

pptr->balance=1;

\*shorter=FALSE;

break;

case -1:

pptr->balance=0;

break;

case 1:

pptr=del\_leftbalance(pptr,shorter);

break;

}

return pptr;

}

struct node\* del\_leftbalance(struct node\* pptr, int \*shorter)

{

struct node \*aptr=pptr->left, \*bptr;

if(aptr->balance==0)

{

aptr->balance=-1;

pptr->balance=1;

\*shorter=FALSE;

pptr=rightRotate(pptr);

}

else if(aptr->balance==1)

{

aptr->balance=0;

pptr->balance=0;

pptr=rightRotate(pptr);

}

else

{

bptr=aptr->right;

switch(bptr->balance)

{

case 0:

aptr->balance=0;

pptr->balance=0;

break;

case 1:

aptr->balance=0;

pptr->balance=-1;

break;

case -1:

aptr->balance=1;

pptr->balance=0;

}

bptr->balance=0;

pptr->left=leftRotate(aptr);

pptr=rightRotate(pptr);

}

return pptr;

}

void display(struct node \*root, int

space)

{

int i;

if(root==NULL)

return;

space+=CNT;

display(root->right, space);

printf("\n");

for(i=CNT;i<space;i++)

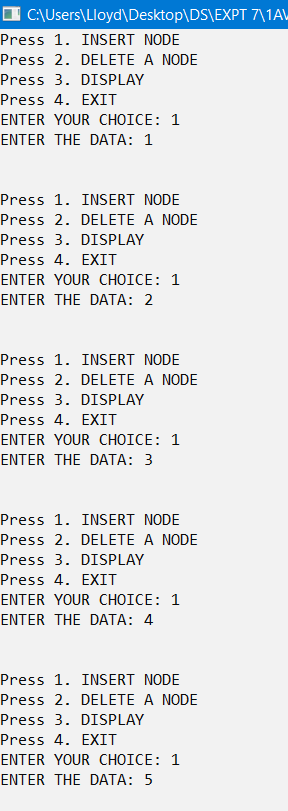
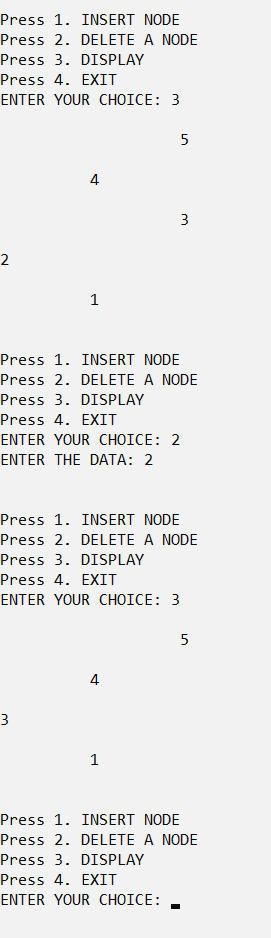
printf(" ");

printf("%d\n",root->data);

display(root->left, space);

}

**OUTPUT**



**CONCLUSION**

The given problem statements were successfully compiled and executed.

**LEARNINGS AND FINDINGS**

This experiment demonstrates

1. Concept of AVL Trees
2. Insertion and Deletion in AVL trees

Balanced factor is the fundamental attribute of AVL trees.

AVL Trees provides an efficient design in solving the problems faced in a normal binary search tree and thus makes insertion, deletions and searching very efficient.

|  |  |
| --- | --- |
| **SR. NO.** | **COMPILATION TIME** |
| 1 | 0.20 s |