**Expt. 5 operations on AVL trees**

Aim:-

Operations on AVL tree that is insertion and deletion of nodes.

Theory:-

Height Balanced Trees (AVL TREE):

Adelson-Velskii and Landis in 1962 introduced a binary tree structure that is balanced with respect to the heights of subtrees at the same time it should be binary search tree. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in O( log n ) time if the tree has n nodes in it. At the same time a new identifier can be entered or deleted from such a tree in time O( log n ) . the resulting tree remains height balanced. The tree structure introduced by them is given name AVL tree. As with binary trees it is natural to define AVL trees recursively.

Definition: An empty tree is height balanced. If T is a nonempty binary tree with TL and TR as its left and right subtrees, then T is height balanced iff (i) TL and TR are height balanced and (ii) | hL – hR | <= 1 where hL and hR are the heights of TL and TR respectively.

Main objective of Height balanced tree is to keep all the nodes in the tree as closer as to the root ; as it will improve the efficiency in searching as well as traversing the complete tree.

In every node along with the identifier the balanced factor is also stored as shown in the figure. Balanced factor is calculated as height of left subtree – height of the right sub tree. If the balanced factor exceeds the range -1...+1 the node will violate the rule of height balanced tree.

After every insertion of the node the balanced factor of its ancestors is updated.

As height balanced tree is also a binary search tree ; insertion of new node is always carried out as terminal node.

<https://upload.wikimedia.org/wikipedia/commons/f/fd/AVL_Tree_Example.gif>

PROGRAM:-

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

typedef struct node

{

char str[20];

struct node\* right;

struct node\* left;

struct node\* parent;

int height;

}node;

int balance\_factor(node\* root)

{

if(root==NULL)

return 0;

int ha,hb;

node\* left=root->left;

node\* right=root->right;

if(left==NULL)

ha=0;

else

ha=left->height;

if(right==NULL)

hb=0;

else

hb=right->height;

return (ha-hb);

}

int max(int a, int b)

{

return (a > b)? a : b;

}

int height(node \*root)

{

if(root==NULL)

return 0;

return max(height(root->left),height(root->right))+1;

}

void RR(node \*unbal)

{

node \*X=unbal->right;

node \*leftOfX=X->left;

node \*parentOfunbal=unbal->parent;

if(parentOfunbal!=NULL)

{

if(parentOfunbal->left==unbal)

parentOfunbal->left=X;

else

parentOfunbal->right=X;

}

X->parent=unbal->parent;

unbal->parent=X;

X->left=unbal;

unbal->right=leftOfX;

X->height = max(height(X->left), height(X->right))+1;

unbal->height = max(height(unbal->left), height(unbal->right))+1;

}

void LL(node \*unbal)

{

node \*X=unbal->left;

node \*rightOfX=X->right;

node \*parentOfunbal=unbal->parent;

if(parentOfunbal!=NULL)

{

if(parentOfunbal->left==unbal)

parentOfunbal->left=X;

else

parentOfunbal->right=X;

}

X->parent=unbal->parent;

unbal->parent=X;

X->right=unbal;

unbal->left=rightOfX;

X->height = max(height(X->left), height(X->right))+1;

unbal->height = max(height(unbal->left), height(unbal->right))+1;

}

void LR(node \*unbal)

{

node \*X=unbal->left;

node \*parentOfunbal=unbal->parent;

node \*rightOfX=X->right;

node \*rightOfY=rightOfX->right;

node \*leftOfY=rightOfX->left;

rightOfX->parent=parentOfunbal;

if(parentOfunbal!=NULL)

{

if(parentOfunbal->left==unbal)

parentOfunbal->left=rightOfX;

else

parentOfunbal->right=rightOfX;

}

rightOfX->right=unbal;

unbal->parent=rightOfX;

rightOfX->left=X;

X->parent=rightOfX;

X->right=leftOfY;

if(leftOfY!=NULL)

leftOfY->parent=X;

unbal->left=rightOfY;

if(rightOfY!=NULL)

rightOfY->parent=unbal;

unbal->height=max(height(unbal->left), height(unbal->right))+1;

X->height=max(height(X->left), height(X->right))+1;

rightOfX->height=max(height(rightOfX->left), height(rightOfX->right))+1;

}

void RL(node\* unbal)

{

node \*X=unbal->right;

node \*parentOfunbal=unbal->parent;

node \*leftOfX=X->left;

node \*rightOfY=leftOfX->right;

node \*leftOfY=leftOfX->left;

leftOfX->parent=parentOfunbal;

if(parentOfunbal!=NULL)

{

if(parentOfunbal->left==unbal)

parentOfunbal->left=leftOfX;

else

parentOfunbal->right=leftOfX;

}

leftOfX->left=unbal;

unbal->parent=leftOfX;

leftOfX->right=X;

X->parent=leftOfX;

X->left=rightOfY;

if(rightOfY!=NULL)

rightOfY->parent=X;

unbal->right=leftOfY;

if(leftOfY!=NULL)

leftOfY->parent=unbal;

unbal->height=max(height(unbal->left), height(unbal->right))+1;

X->height=max(height(X->left), height(X->right))+1;

leftOfX->height=max(height(leftOfX->left), height(leftOfX->right))+1;

}

node \*unbalOfAvl(node\* root,node\* p)

{

int bf=balance\_factor(p);

node \*unbal=p;

while((bf<=1&&bf>=-1) && unbal!=NULL)

{

bf=balance\_factor(unbal=unbal->parent);

}

if(unbal!=NULL)

{

if(bf>1&&strcmp(p->str,unbal->left->str)<0)//LL

{

LL(unbal);

}

else if(bf<-1&&strcmp(p->str,unbal->right->str)>0)//RR

{

RR(unbal);

}

else if(bf>1&&strcmp(p->str,unbal->left->str)>0)//LR

{

LR(unbal);

}

else if(bf<-1&&strcmp(p->str,unbal->right->str)<0)//RL

{

RL(unbal);

}

node \*temp\_par=unbal;

node \*par=unbal;

while(temp\_par!=NULL)

{

temp\_par->height=height(temp\_par);

par=temp\_par;

temp\_par=temp\_par->parent;

}

return unbal;

}

else

return NULL;

}

node \*insert(node \*p,node \*root)

{

node \*temp=root,\*follow;

int i;

while(temp!=NULL)

{

i=strcmp(temp->str,p->str);

if(i>0)

{

follow=temp;

temp=temp->left;

}

else

{

follow=temp;

temp=temp->right;

}

}

if(i>0)

{

follow->left=p;

p->parent=follow;

}

else

{

follow->right=p;

p->parent=follow;

}

node \*temp\_par=p;

while(temp\_par!=NULL)

{

temp\_par->height=height(temp\_par);

temp\_par=temp\_par->parent;

}

node \*unbal=unbalOfAvl(root,p);

if(unbal==NULL)

return root;

else

{

node \*follow=unbal;

while(unbal!=NULL)

{

follow=unbal;

unbal=unbal->parent;

}

return follow;

}

}

void printPreorder(struct node\* root)

{

if (root == NULL)

return;

/\* first print data of node \*/

printf("%s \t", root->str);

printf("%d\n",balance\_factor(root));

/\* then recur on left sutree \*/

printPreorder(root->left);

/\* now recur on right subtree \*/

printPreorder(root->right);

}

node\* search(char \*name,node \*root)

{

if(root==NULL)

return NULL;

int i;

node \*temp=root;

while(temp!=NULL)

{

i=strcmp(temp->str,name);

if(i==0)

return temp;

if(i>0)

{

temp=temp->left;

}

else

{

temp=temp->right;

}

}

return NULL;

}

node \*delOfAvl(node \*root,node \*p)

{

int bf=balance\_factor(p);

node \*unbal=p;

while((bf<=1&&bf>=-1) && unbal!=NULL)

{

bf=balance\_factor(unbal=unbal->parent);

}

if(unbal!=NULL)

{

int lh,rh;

node \*a=unbal->left;node \*b=unbal->right;

if(a!=NULL)

lh=a->height;

else

lh=0;

if(b!=NULL)

rh=b->height;

else

rh=0;

if(bf>1)

{

if(lh>rh)

{

LL(unbal);

}

else

{

LR(unbal);

}

}

else if(bf<-1)

{

if(lh>rh)

{

RL(unbal);

}

else

{

RR(unbal);

}

}

node \*temp\_par=unbal;

node \*par=unbal;

while(temp\_par!=NULL)

{

temp\_par->height=height(temp\_par);

par=temp\_par;

temp\_par=temp\_par->parent;

}

return unbal;

}

else

return NULL;

}

//this will return root after deletion

node \*delete(node \*target,node \*root)

{

int i=0;

if(target->left==NULL&&target->right==NULL)

i=1;

else if(target->right==NULL||target->left==NULL)

i=2;

else if(target->right!=NULL&&target->left!=NULL)

i=3;

else{}

if(i==1)

{

node \*parTarget=(target->parent);

if((parTarget->left)==target)

parTarget->left=NULL;

else

parTarget->right=NULL;

free(target);

node \*temp\_par=parTarget;

while(temp\_par!=NULL)

{

temp\_par->height=height(temp\_par);

temp\_par=temp\_par->parent;

}

node \*temp=parTarget;

node \*result;

while(temp!=NULL)

{

result=delOfAvl(root,temp);

if(result==NULL)

temp=temp->parent;

else

temp=result->parent;

}

node \*temproot=parTarget;

while(parTarget!=NULL)

{

temproot=parTarget;

parTarget=parTarget->parent;

}

return temproot;

}

else if(i==2)

{

node \*parTarget=target->parent;

if(parTarget->left==target)

{

if(target->left!=NULL)

{

parTarget->left=target->left;

(target->left)->parent=parTarget;

}

else

{

parTarget->left=target->right;

(target->right)->parent=parTarget;

}

}

else if(parTarget->right==target)

{

if(target->left!=NULL)

{

parTarget->right=target->left;

(target->left)->parent=parTarget;

}

else

{

parTarget->right=target->right;

(target->right)->parent=parTarget;

}

}

else{}

free(target);

node \*temp\_par=parTarget;

while(temp\_par!=NULL)

{

temp\_par->height=height(temp\_par);

temp\_par=temp\_par->parent;

}

node \*temp=parTarget;node \*result;

while(temp!=NULL)

{

result=delOfAvl(root,temp);

if(result==NULL)

temp=temp->parent;

else

temp=result->parent;

}

node \*temproot=parTarget;

while(parTarget!=NULL)

{

temproot=parTarget;

parTarget=parTarget->parent;

}

return temproot;

}

else if(i==3)

{

node \*temp,\*follow;

temp=follow=target->right;

while(temp!=NULL)

{

follow=temp;

temp=temp->left;

}

strcpy(target->str,follow->str);

node \*parTarget=follow->parent;

if(follow->right==NULL&&follow->left==NULL)

{

temp=follow->parent;

temp->left=NULL;

}

else

{

(follow->parent)->left=follow->right;

(follow->right)->parent=(follow->parent);

}

free(follow);

node \*temp\_par=parTarget;

while(temp\_par!=NULL)

{

temp\_par->height=height(temp\_par);

temp\_par=temp\_par->parent;

}

node \*temp1;

node \*result=delOfAvl(root,parTarget);

if(result==NULL)

temp1=parTarget->parent;

else

temp1=result;

while(temp1!=NULL)

{

result=delOfAvl(root,temp);

if(result==NULL)

temp1=temp1->parent;

else

temp1=result->parent;

}

node \*temproot=parTarget;

while(parTarget!=NULL)

{

temproot=parTarget;

parTarget=parTarget->parent;

}

return temproot;

}

else{return;}

}

void main()

{

printf("now you will create avl tree\n");

printf("enter string for root\n");

node \*root=(node \*)malloc(sizeof(node));

scanf("%s",root->str);

root->parent=NULL;

root->height=0;

int comp;

printf("enter 'stop' when you want to stop adding node\n");

do{

printf("enter string \n");

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

scanf("%s",p->str);

p->right=NULL;

p->left=NULL;

if((comp=strcmp(p->str,"stop"))!=0)

{

root=insert(p,root);

}

}while(comp!=0);

printf("=======Creation is complete!=======\n");

printf("Preorder traversal of AVL tree is\nNode\t\tBalance\_factor\n");

printPreorder(root);

printf("now we will start deletion\n");

printf("enter how many nodes you will delete(make sure it is less than number of nodes in tree)\n");

int n,count=0;

scanf("%d",&n);

char name[20];

while(count<n)

{

printf("enter node to be deleted\n");

scanf("%s", name);

node \*result=search(name,root);

if(result!=NULL)

{

root=delete(result,root);

printf("Node deleted successfully\n\n");

printf("Preorder traversal of AVL tree is\nNode\t\tBalance\_factor\n\n");

printPreorder(root);

}

else{

printf("no such element found\n");

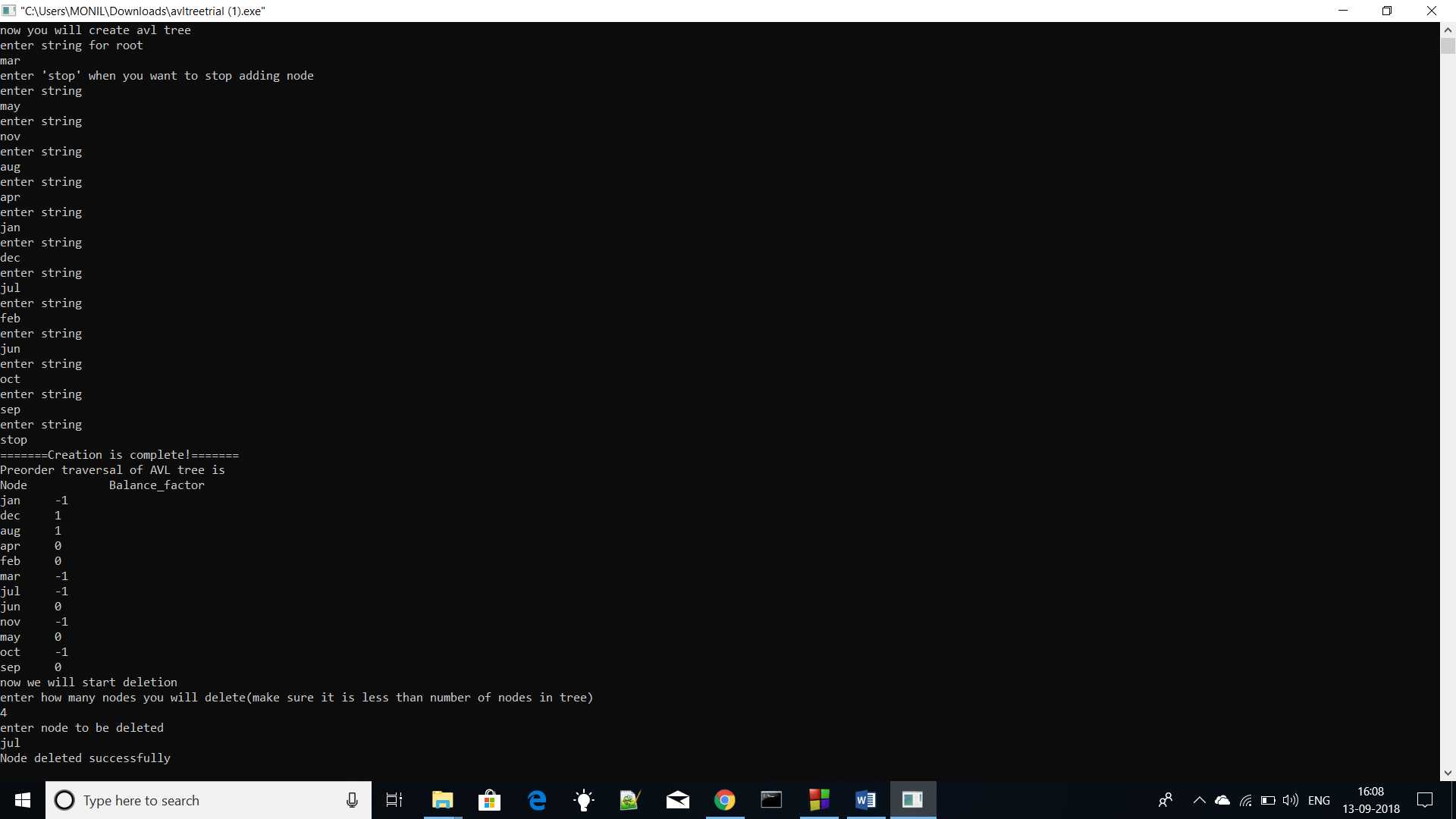
}

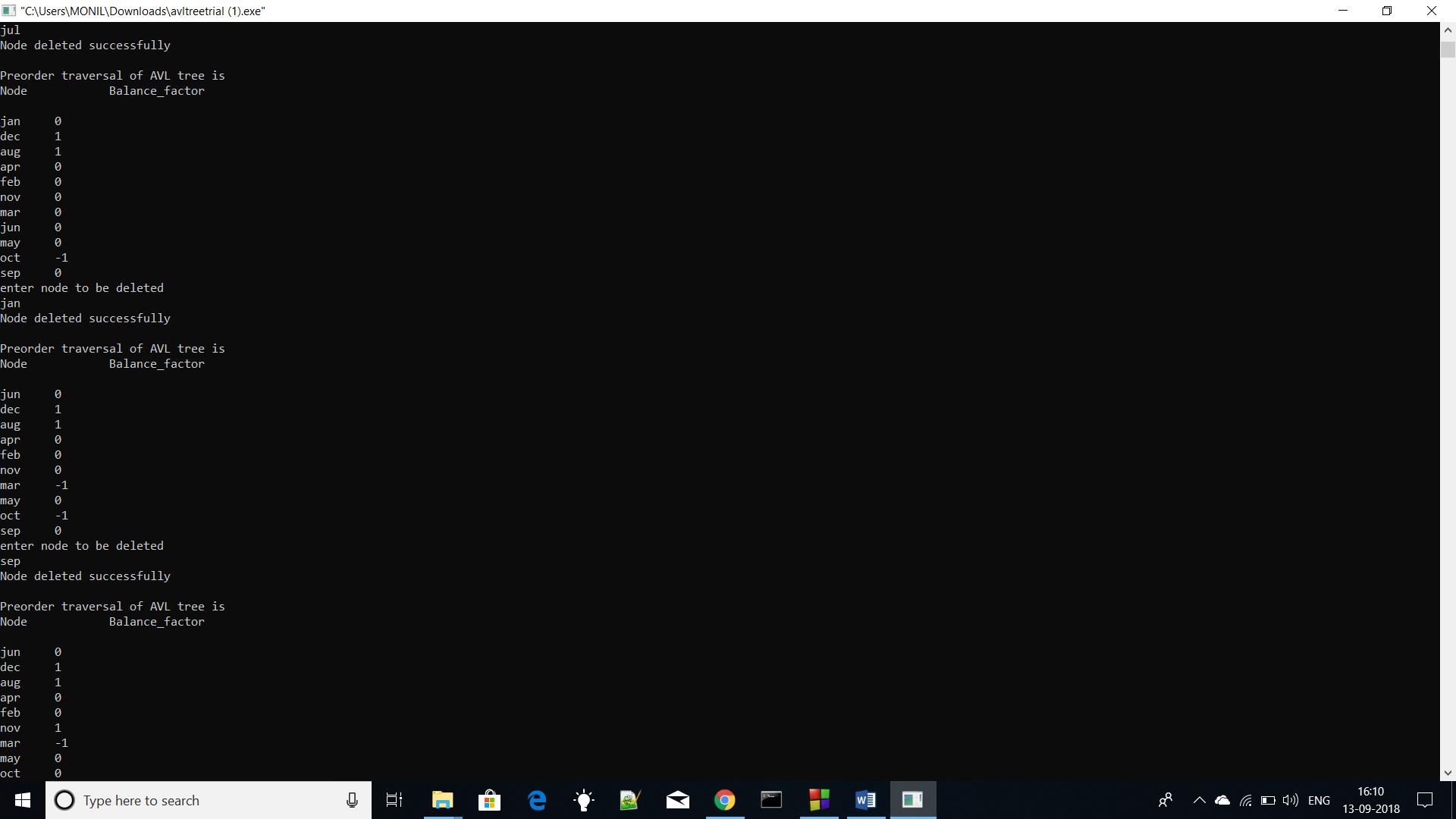
count++;

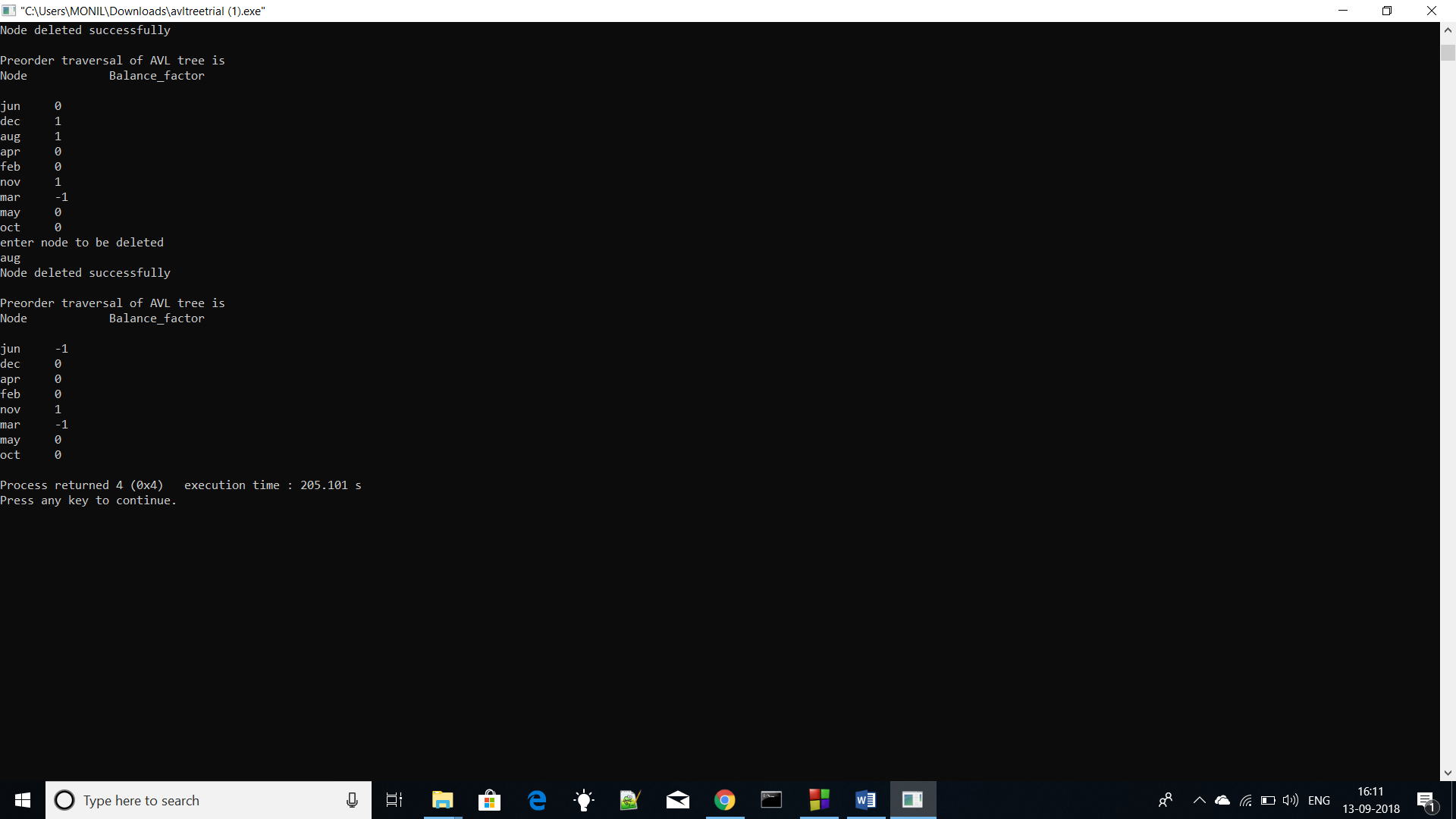
}

}

OUTPUT:-







CONCLUSION:-AVL tree were invented to make searching more faster. One such application for this is compiler makes AVL trees of variables and when needed it is searched in AVL tree and we get error when variable is undeclared and warning when redefined.