**Name: Riya Jain**

**Section: A (CSE)**

**Roll No: 17**

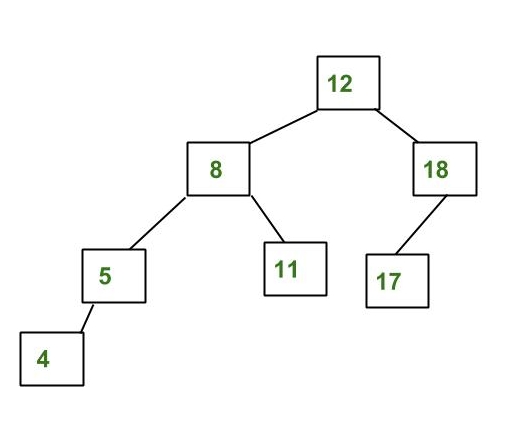
**EXPERIMENT: 8**

# Aim: To study AVL trees and implement various operations on it.

# Problem Definition: Create a self-referential structure, TNode to represent a node of a AVL tree. Implement the routines to - (1) insert a node in AVL tree (2) delete a node from Tree (3)perform inorder walk of the tree, (4) perform preorder walk of the tree, (5) perform postorder walk of the tree, (6) find height of the tree [assume ROOT at height 0], (7)count and print all parent [internal] nodes, leaf nodes in the tree, and (8) to locate a KEY in the tree. Create a menu-driven program to test these routines.

# Theory: AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

**An Example Tree that is an AVL Tree**



# The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

# Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the 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(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

# ****Insertion****  To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).  1) Left Rotation  2) Right Rotation

a) Left Left Case

T1, T2, T3 and T4 are subtrees.

z y

/ \ / \

y T4 Right Rotate (z) x z

/ \ - - - - - - - - -> / \ / \

x T3 T1 T2 T3 T4

/ \

T1 T2

**b) Left Right Case**

z z x

/ \ / \ / \

y T4 Left Rotate (y) x T4 Right Rotate(z) y z

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

T1 x y T3 T1 T2 T3 T4

/ \ / \

T2 T3 T1 T2

**c) Right Right Case**

z y

/ \ / \

T1 y Left Rotate(z) z x

/ \ - - - - - - - -> / \ / \

T2 x T1 T2 T3 T4

/ \

T3 T4

**d) Right Left Case**

z z x

/ \ / \ / \

T1 y Right Rotate (y) T1 x Left Rotate(z) z y

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

x T4 T2 y T1 T2 T3T4

/ \ / \

T2 T3 T3 T4

**Time Complexity:** The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

**Deletion:**

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).   
1) Left Rotation   
2) Right Rotation

**ALGORITHMS:**

1. **Insert(T,val)**

1.IF T = NULL

Allocate memory for TREE

2.T →DATA = VAL, T →LEFT = NULL, T →RIGHT =NULL

T → HEIGHT = 0

3. ELSE IF  (VAL < T →DATA)

T →LEFT = Insert(T →LEFT, VAL)

ELSE IF( VAL > T →DATA)

T →RIGHT = Insert(T →RIGHT, VAL)

4.Balance = HEIGHT(T →LEFT)- HEIGHT(T →RIGHT )

5. // Left Left Case

If (Balance > 1 && VAL < T → LEFT→DATA)

return RightRotate(T)

// Right Right Case

If (Balance <-1 && VAL > T → RIGHT→DATA)

return LeftRotate(T)

// Left Right Case

If (Balance > 1 && VAL > T → LEFT→DATA)

T → LEFT = leftRotate(T → LEFT)

return RightRotate(T)

// Right Left Case

If (Balance < -1 && VAL < T → RIGHT→DATA)

T → RIGHT = RightRotate(T → RIGHT)

return LeftRotate(T)

6. Return T

1. **Delete(T,val)**

1. Deletion in BST

2. T → HEIGHT = MAX(HEIGHT(T →LEFT),HEIGHT(T →RIGHT))+1

3.Balance = Calculate\_bal(T)

4. // Left Left Case

If (Balance > 1 && Calculate\_bal(T->left) >= 0)

return RightRotate(T)

// Left Right Case

If(Balance > 1 && Calculate\_bal(T->left) < 0)

T->left = LeftRotate(T->left)

return RightRotate(T)

// Right Right Case

If (Balance < -1 && Calculate\_bal(T->RIGHT) <= 0)

return LeftRotate(T)

// Right Left Case

If (Balance < -1 && Calculate\_bal(T->RIGHT) > 0)

T->RIGHT = RightRotate(T->RIGHT)

return LeftRotate(T)

// Right Left Case

If (Balance < -1 && VAL < T → RIGHT→DATA)

T → RIGHT = RightRotate(T → RIGHT)

return LeftRotate(T)

6. Return T

**3. LargestElement (T):**

1. IF T = NULL OR T-> RIGHT = NULL Return T
2. ELSE

Return LargestElement(T-> RIGHT)

1. EXIT
2. **SmallestElement (T):**
3. IF T = NULL OR T-> LEFT = NULL Return T
4. ELSE

Return SmallestElement(T-> LEFT)

1. EXIT

**5. Height(T):**

1. IF T = NULL Return -1
2. IF(T->LEFT=NULL AND T->RIGHT=NULL) Return 0
3. SET LeftHeight = Height(T →LEFT) SET RightHeight = Height(T →RIGHT) IF LeftHeight > RightHeight

Return LeftHeight + 1 ELSE

Return RightHeight + 1

3. EXIT

**6. Preorder(T):**

1. If T<>NULL

1. Process root node Write T→DATA
2. Process left sub-tree PREORDER(T→LEFT)
3. Process right subtree PREORDER(T→ RIGHT)

2. Exit

**7. Postorder(T):**

1. If T<>NULL

1. Process left sub-tree POSTORDER(T→LEFT)
2. Process right subtree POSTORDER(T→ RIGHT)
3. Process root node Write T→DATA

2. Exit

**8. Inorder(T):**

1. If T<>NULL

1. Process left sub-tree

INORDER(T→LEFT)

1. Process root node Write T→DATA
2. Process right subtree INORDER(T→ RIGHT)

2. Exit

**9. Level \_Order (T):**

1. Initialize variable level = 0

h = HEIGHT(T)

1. Traverse the Tree while(level<=h)

Call Nodes\_Level(T, level) level=level+1

1. EXIT

**10. Nodes\_Level(T, level):**

1. Empty Tree If T = NULL Return
2. Process root node If level = 0

Write T→DATA

1. Process left subtree and right subtree Else

Call Nodes\_Level(T→LEFT, level-1) Call Nodes\_Level(T→RIGHT, level-1)

**11.Search (root, item):**

1. if (item = root → data) or (root = NULL)
2. return root
3. else if (item < root → data)
4. return Search(root → left, item)
5. else
6. return Search(root → right, item)
7. END if
8. END

**TotalNodes(T)**

1. IF T = NULL

Return 0

1. ELSE

Return(TotalNodes(T → LEFT)

1. EXIT

**InternalNodes(T)**

1 IF T = NULL

Return 0

1. IF(T → LEFT=NULL AND T → RIGHT=NULL) Return 0
2. ELSE

Return (InternalNodes(T → LEFT) + InternalNodes(T → RIGHT) + 1)

3. EXIT

**ExternalNodes(T)**

1. IF T = NULL Return 0
2. IF(T → LEFT=NULL AND T → RIGHT=NULL) Return 1
3. ELSE

Return (ExternalNodes(T → LEFT) + ExternalNodes(T → RIGHT) )

1. EXIT

**12. Left Rotation(T):**

1. X = T->RIGHT

2. Z = X->LEFT

3. X->LEFT = T

4. T->RIGHT = Z

5. T->HEIGHT = HEIGHT(T)

6. X->HEIGHT = HEIGHT(X)

**13. Right Rotation(T):**

1. X = T → LEFT

2. Z = X → RIGHT

3. X →RIGHT = T

4. T → LEFT = Z

5. Update heights

T → HEIGHT = HEIGHT(T)

X→ HEIGHT = HEIGHT(X)

**CODE**:

#include<stdio.h>

#include<stdlib.h>

struct Tree

{

  int data;

  struct Tree\*left;

  struct Tree\*right;

  int height;

};

typedef struct Tree tree;

tree\*insert(tree\*root,int data);

tree\*delete(tree\*root,int data);

tree\*leftrotate(tree\*root);

tree\*rightrotate(tree\*root);

void inorder(tree\*root);

void preorder(tree\*root);

void postorder(tree\*root);

void levelorder(tree\*root);

void nodes\_level(tree\*root,int level);

int height(tree\*root);

int search(tree\*root,int element,int level);

int totalInternalNodes(tree\*root);

int totalExternalNodes(tree\*root);

tree\*Findmin(tree\*root);

tree\*findLargestNode(tree\*root);

int totalnodes(tree\*root);

int Calculate\_bal(tree\*root);

int main()

{

tree\*root=NULL;

int ch,l=1,data,element,smallest,largest;

 while(l)

{

 printf("\nPRESS:(1) insert a node in AVL tree\n (2) delete a node from AVL tree \n(3)inorder traversal\n (4) preorder traversal\n (5) postorder traversal\n (6) level order traversal\n(7) find height of the tree\n (8)count and print all parent [internal] nodes, leaf nodes and total nodes in the tree\n (9) Search a key\n (10)Print largest element\n(11)Print smallest element\n(12)Exit:");

 scanf("%d",&ch);

 switch(ch)

{

case 1: printf("You opted for Insertion a Node in AVL tree!\n");

        printf("Enter data do you want to insert in AVL tree:");

        scanf("%d",&data);

        root=insert(root,data);

        break;

case 2: printf("You opted for Deletion a Node from AVL tree!\n");

        printf("Enter data do you want to delete from AVL tree:");

        scanf("%d",&data);

        root=delete(root,data);

        break;

case 3: printf("You opted for inorder traversal!");

        printf("\nDisplaying tree:\n");

        inorder(root);

        break;

case 4: printf("You opted for preorder traversal!");

        printf("\nDisplaying tree:\n");

        preorder(root);

        break;

case 5: printf("You opted for postorder traversal!");

        printf("\nDisplaying tree:\n");

        postorder(root);

        break;

case 6: printf("You opted for levelorder traversal!");

        printf("\nDisplaying tree:\n");

        levelorder(root);

        break;

case 7: printf("You opted for finding height of the tree!");

        int h=height(root);

        printf("\nThe height of your binary tree is: %d",h);

        break;

case 8: printf("You opted for count all parent [internalnodes], leaf nodes and total nodes in the tree");

        int res1=totalInternalNodes(root);

        printf("\nThe total Internal nodes (parent nodes)present in AVL Tree are: %d",res1);

        int res2=totalExternalNodes(root);

        printf("\nThe total External nodes (leaf nodes) present in AVL Tree are: %d",res2);

        int res3=totalnodes(root);

        printf("\nThe total nodes present in BST are: %d",res3);

        break;

case 9: printf("You opted for Searching for a key of the tree!");

        printf("\nEnter data do you want to search in AVL Tree:");

        scanf("%d",&element);

        int res=search(root,element,0);

        if(res==0)

        printf("\nSearched element %d is not present in the AVL tree",element);

        else if(res==1)

        printf("\nSearched element %d is present in AVL Tree",element);

        break;

case 10:printf("You opted for finding the largest element of the tree!");

        root=findLargestNode(root);

        printf("\n The Largest element is %d",root->data);

        break;

case 11:printf("You opted for finding the smallest element of the tree!");

         root=Findmin(root);

        printf("\n The smallest element is %d",root->data);

         break;

case 12: exit(0);

default: printf("You entered invalid choice");

}

}

return 0;

}

tree\*insert(tree\*root,int data)

{

  if(root==NULL)

  {   tree\*ptr=(tree\*)malloc(sizeof(tree));

      ptr->data=data;

      ptr->left=NULL;

      ptr->right=NULL;

      ptr->height=0;

      root=ptr;

      return root;

  }

   if(data<root->data)

   {

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

   }

    if(data>root->data)

   {

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

   }

   int bf=height(root->left)-height(root->right);//Calculating balance factor

   if(bf>1 && data<root->left->data) // LL Rotation

   {   printf("LL Rotation\n");

       return rightrotate(root);

   }

   else if(bf<-1 && data>root->right->data) //RR Rotation

   {   printf("RR Rotation\n");

       return leftrotate(root);

   }

   else if(bf>1 && data>root->left->data) //LR Rotation

   {   printf("LR Rotation");

       root->left=leftrotate(root->left);

       levelorder(root);

       return rightrotate(root);

   }

   else if(bf<-1 && data<root->right->data) //RL Rotation

   {  printf("RL Rotation\n");

       root->right=rightrotate(root->right);

       levelorder(root);

       return leftrotate(root);

   }

   return root;

}

void inorder(tree\*root) //left root right

{

 if(root==NULL)

 {

 return;

 }

 inorder(root->left);

 //printf("%d ",root->left);

 printf("%d->",root->data);

 //printf("%d ",root->right);

 inorder(root->right);

}

tree\*delete(tree\*root,int data)

{

 if(root==NULL)

 {

 printf("Deletion not possible\n");

 return NULL;

 }

 else if(root->data>data)

 {

  root->left=delete(root->left,data);

 }

 else if(root->data<data)

 {

  root->right=delete(root->right,data);

 }

 else

 {

//case 1: No child(LEAF NODE)

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

 {

         free(root);

         root=NULL;//root is a dangling pointer before to be assigned to null means pointing to any unavail location

         return root;

 }

 //case 2: Single child

 else if(root->left==NULL)

 {

  tree\*temp=root;

  root=root->right;

  free(temp);

 }

 else if(root->right==NULL)

 {

  tree\*temp=root;

  root=root->left;

  free(temp);

 }

 //case 3: Two childs

 else

 {

  tree\*temp=Findmin(root->right); //successor in the right part of BST

   root->data=temp->data;

   root->right=delete(root->right,temp->data);

 }

}

//root->height=MAX(height(root->left),height(root->right))+1;

//considering the root node our height is changing everytime we are deleting some value

//int res=maxheight(root);

root->height=height(root);

int balance=Calculate\_bal(root);

// Left Left Case

if(balance>1 && Calculate\_bal(root->left) >= 0)

{

printf("LL Rotation\n");

return rightrotate(root);

}

 // Left Right Case

if(balance > 1 && Calculate\_bal(root->left) < 0)

{

printf("LR Rotation\n");

root->left=leftrotate(root->left);

levelorder(root);

return rightrotate(root);

}

// Right Right Case

if(balance< -1 && Calculate\_bal(root->right)<=0)

{

printf("RR Rotation\n");

return leftrotate(root);

}

// Right Left Case

if(balance<-1 && Calculate\_bal(root->right)>0)

{

 printf("RL Rotation\n");

 root->right=rightrotate(root->right);

 levelorder(root);

 return leftrotate(root);

}

return root; // when no changes been done

}

void preorder(tree\*root) //root left right

{

if(root==NULL)

 {

 return;

 }

 //printf("%d ",root->left);

 printf("%d->",root->data);

 //printf("%d ",root->right);

 preorder(root->left);

 preorder(root->right);

}

void postorder(tree\*root) //left right root

{

if(root==NULL)

 {

 return;

 }

 postorder(root->left);

 postorder(root->right);

 //printf("%d ",root->left);

 printf("%d->",root->data);

//printf("%d ",root->right);

}

int height(tree\*root)

{

    int lht,rht;

   if(root==NULL) //empty

   {

       return -1;

   }

   if(root->left==NULL && root->right==NULL) //leaf node (EK HI NODE HAI)

   {

   return 0;

   }

   lht=height(root->left);

   rht=height(root->right);

   if(lht>rht)

   {

       return lht+1;

   }

   else

   {

       return rht+1;

   }

}

int search(tree\*root,int element,int level)

{

    if(root==NULL)

    {

    return 0;

    }

    else if(root->data==element)

    {

    printf("%d is found at level %d",element,level);

    return 1;

    }

    else if(root->data>element)

    {

     search(root->left,element,++level);

    }

    else if(root->data<element)

    {

     search(root->right,element,++level); //return result1 or result2

    }

}

int totalInternalNodes(tree\*root) //parent node

{

if( (root==NULL) || ((root->left==NULL) && (root->right==NULL)))

return 0;

else

{

return (totalInternalNodes(root->left) + totalInternalNodes(root->right) + 1);//+1 for the parent node

}

}

int totalExternalNodes(tree\*root) //leaf node

{

if(root==NULL)

return 0;

else if((root->left==NULL) && (root->right==NULL))

return 1;

else

{

return (totalExternalNodes(root->left) +totalExternalNodes(root->right)); //root node is not leaf node

}

}

tree\*Findmin(tree\*root)

{

   if(root==NULL)

   {

   return NULL;

   }

   while(root->left!=NULL)

   {

   root=root->left;

   }

   return root;

}

int totalnodes(tree\*root)

{

    if(root==NULL)

    {

        return 0;

    }

    else

    {

        return(totalnodes(root->left)+totalnodes(root->right)+1);

    }

}

void levelorder(tree\*root)

{

  int level=0;

  int h=height(root);

  while(level<=h)

  {

  nodes\_level(root,level);

  level=level+1;

  }

}

void nodes\_level(tree\*root,int level)

{

        if(root==NULL)

        {

        return;

        }

         if(level==0)

        {

        printf("%d->",root->data);

        }

        else

        {

       nodes\_level(root->left,level-1);

       nodes\_level(root->right,level-1);

        }

}

tree\*findLargestNode(tree\*root)

{

    if(root==NULL)

    return NULL;

    while(root->right!=NULL)

    {

        root=root->right;

    }

    return root;

}

tree\*rightrotate(tree\*root)

{

    tree\*x=root->left;

    tree\*y=x->right;

    x->right=root;

    root->left=y;

    root->height=height(root);

    x->height=height(x);

    return x; //new root

}

tree\*leftrotate(tree\*root)

{

    tree\*x=root->right;

    tree\*y=x->left;

    x->left=root;

    root->right=y;

    root->height=height(root);

    x->height=height(x);

    return x;

}

int Calculate\_bal(tree\*root)

{

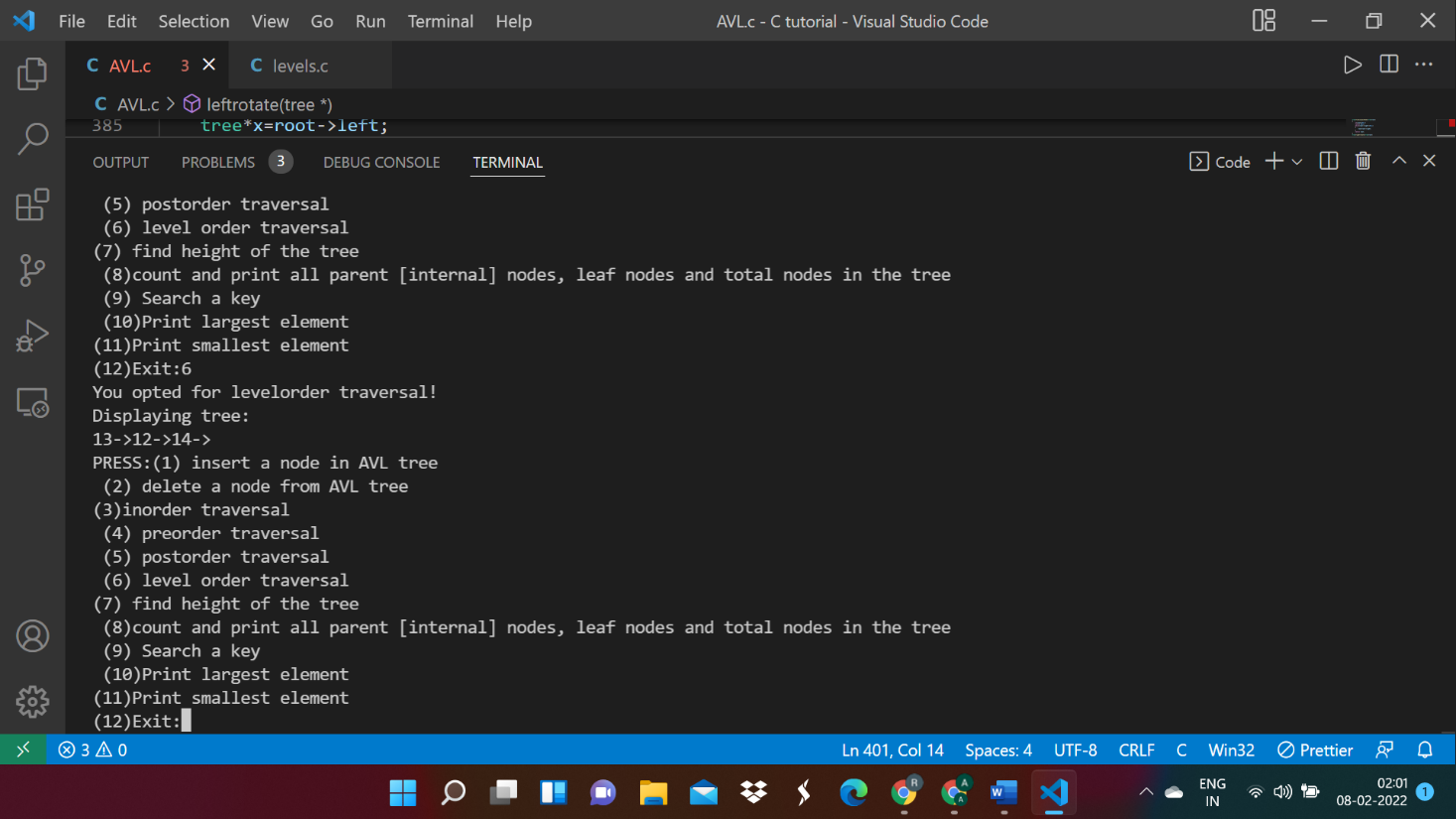
int bf=height(root->left)-height(root->right);

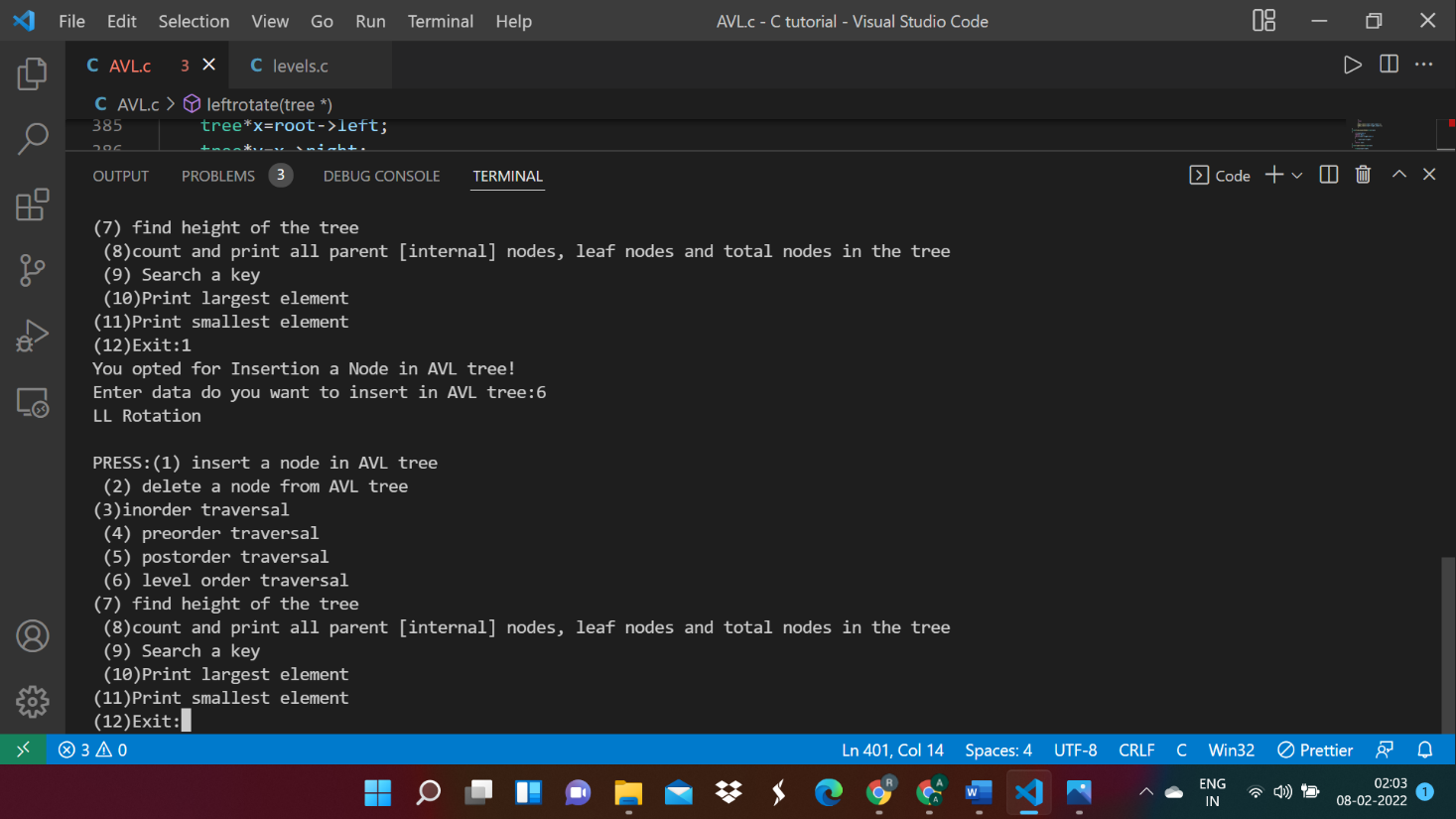
return bf;

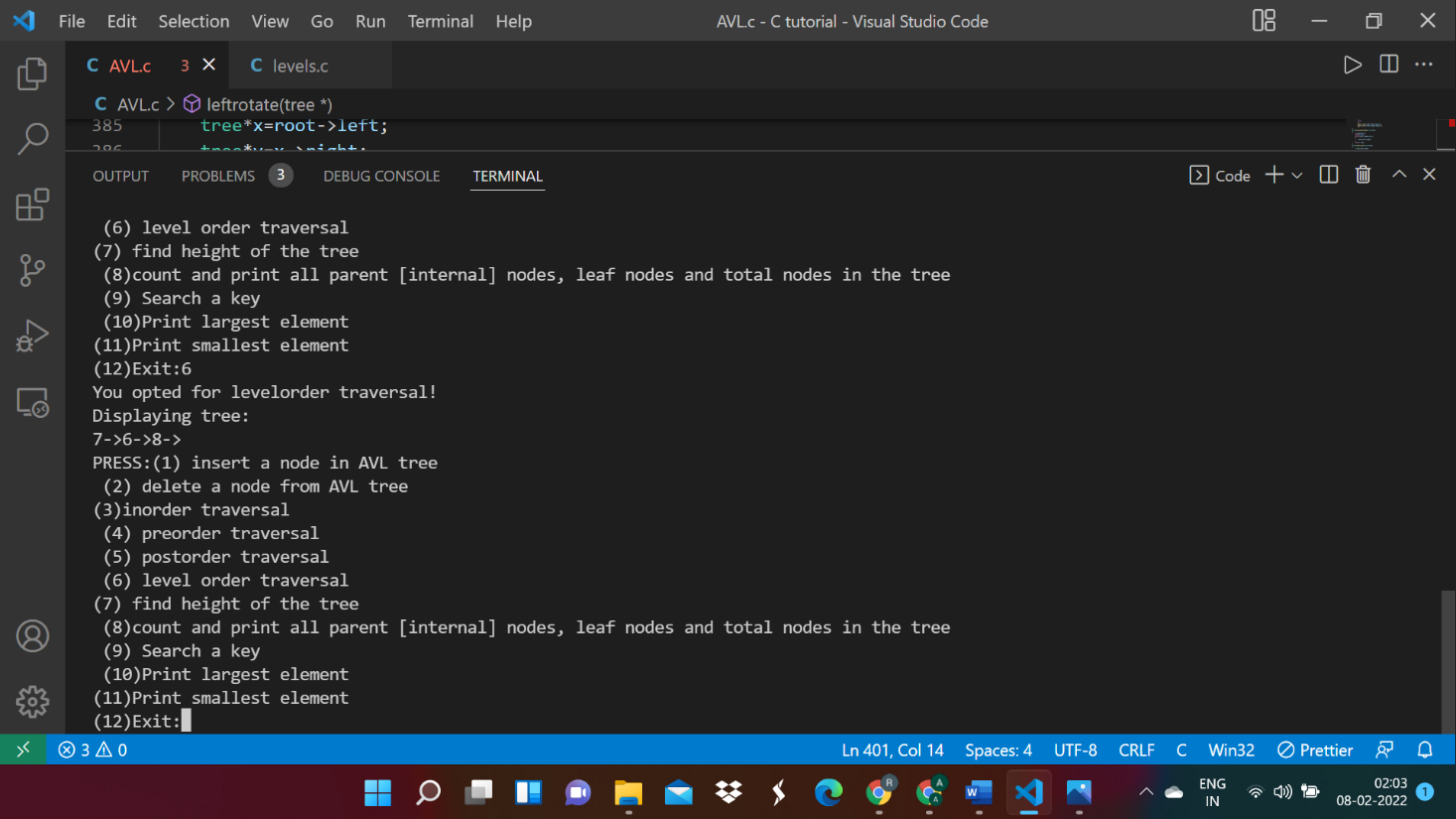
}

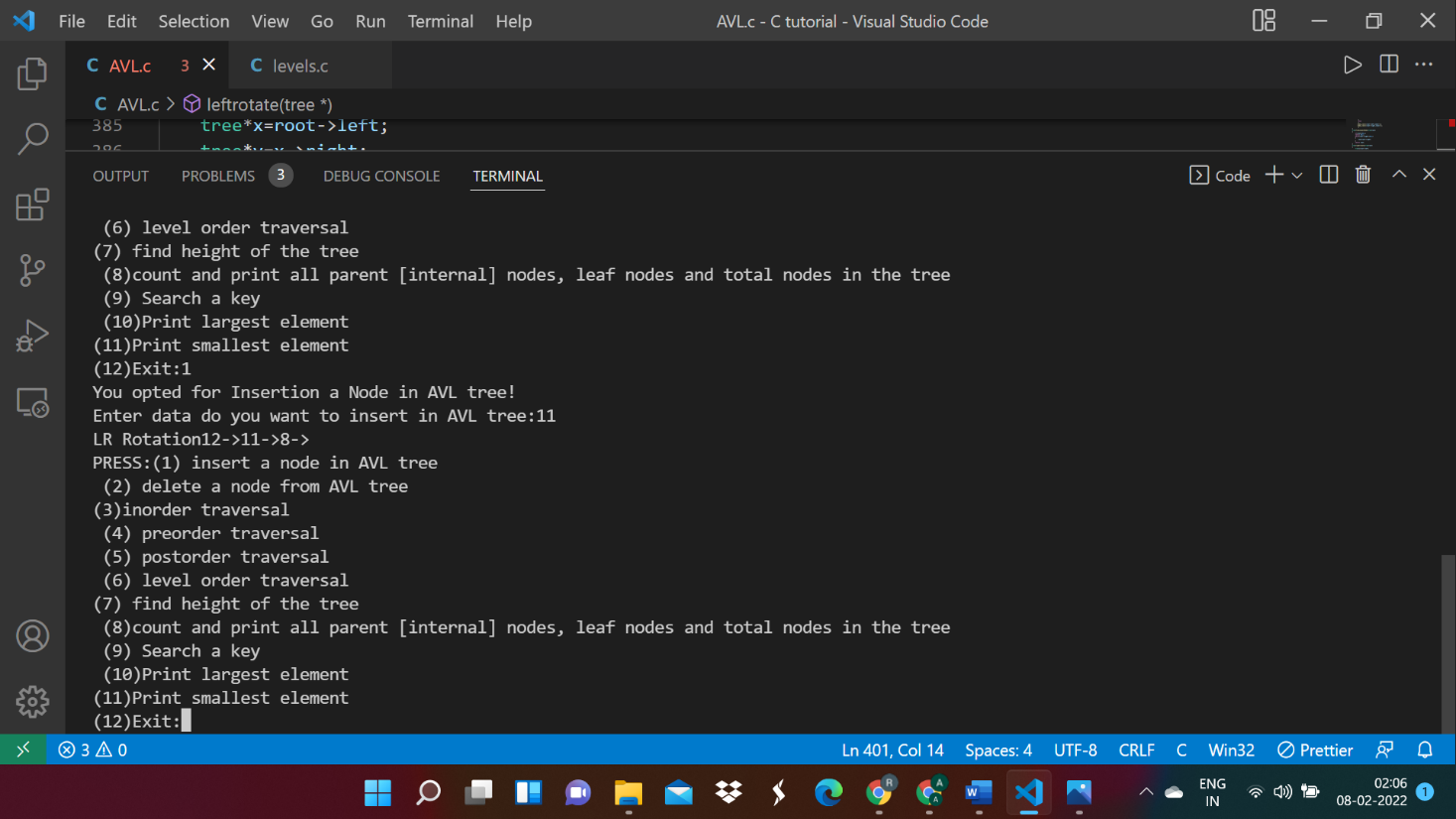
**OUTPUTS:**

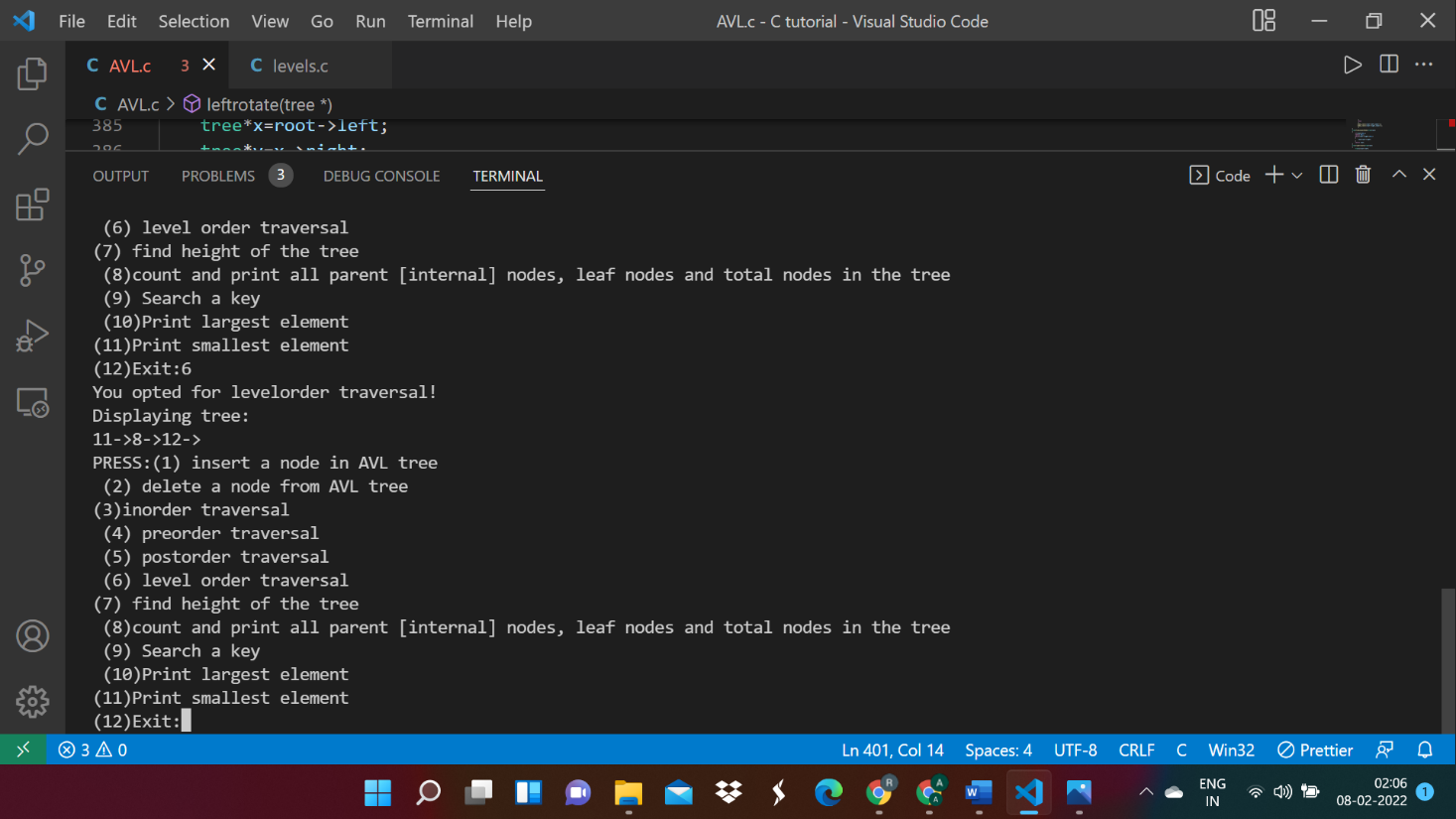


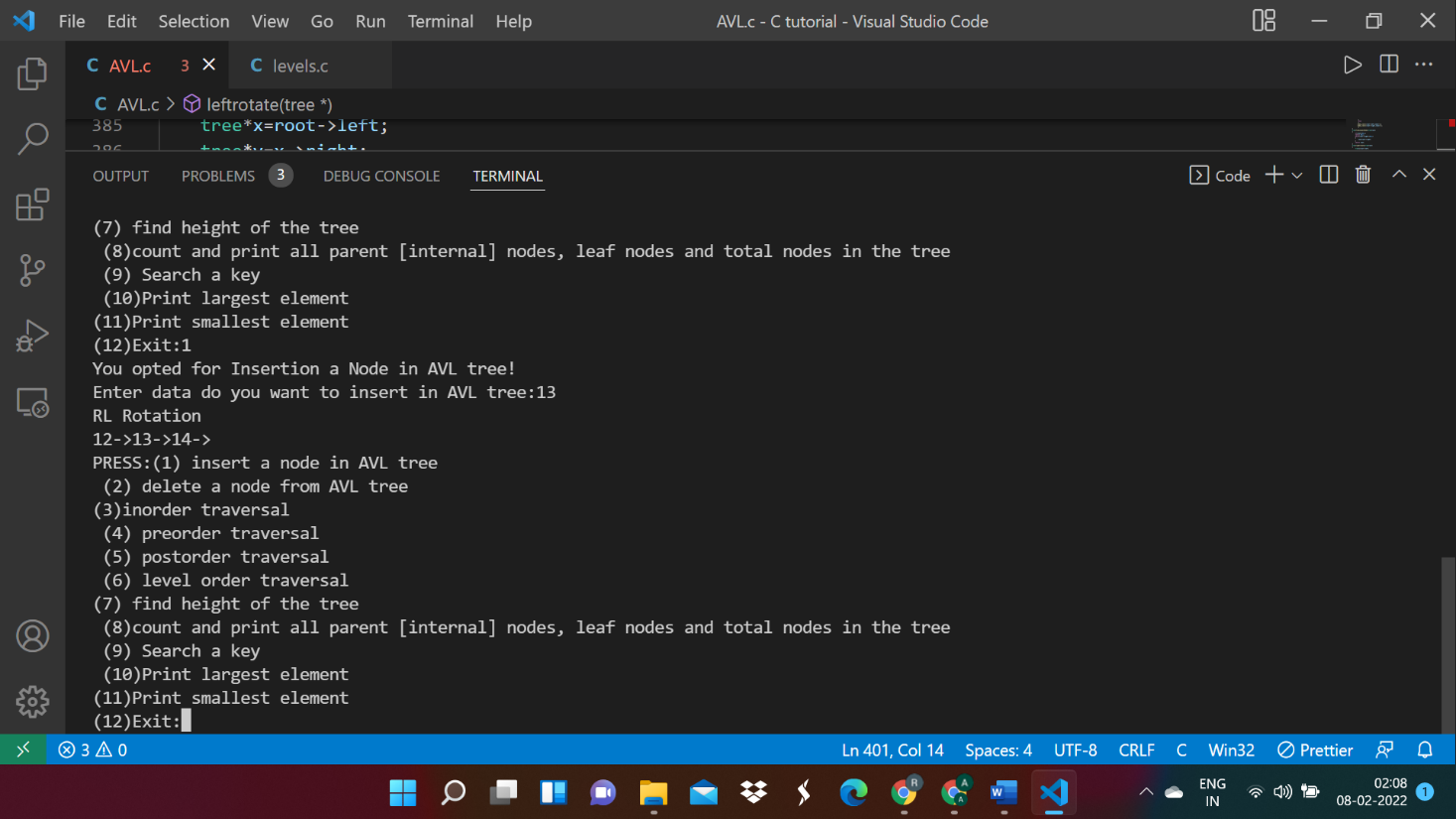




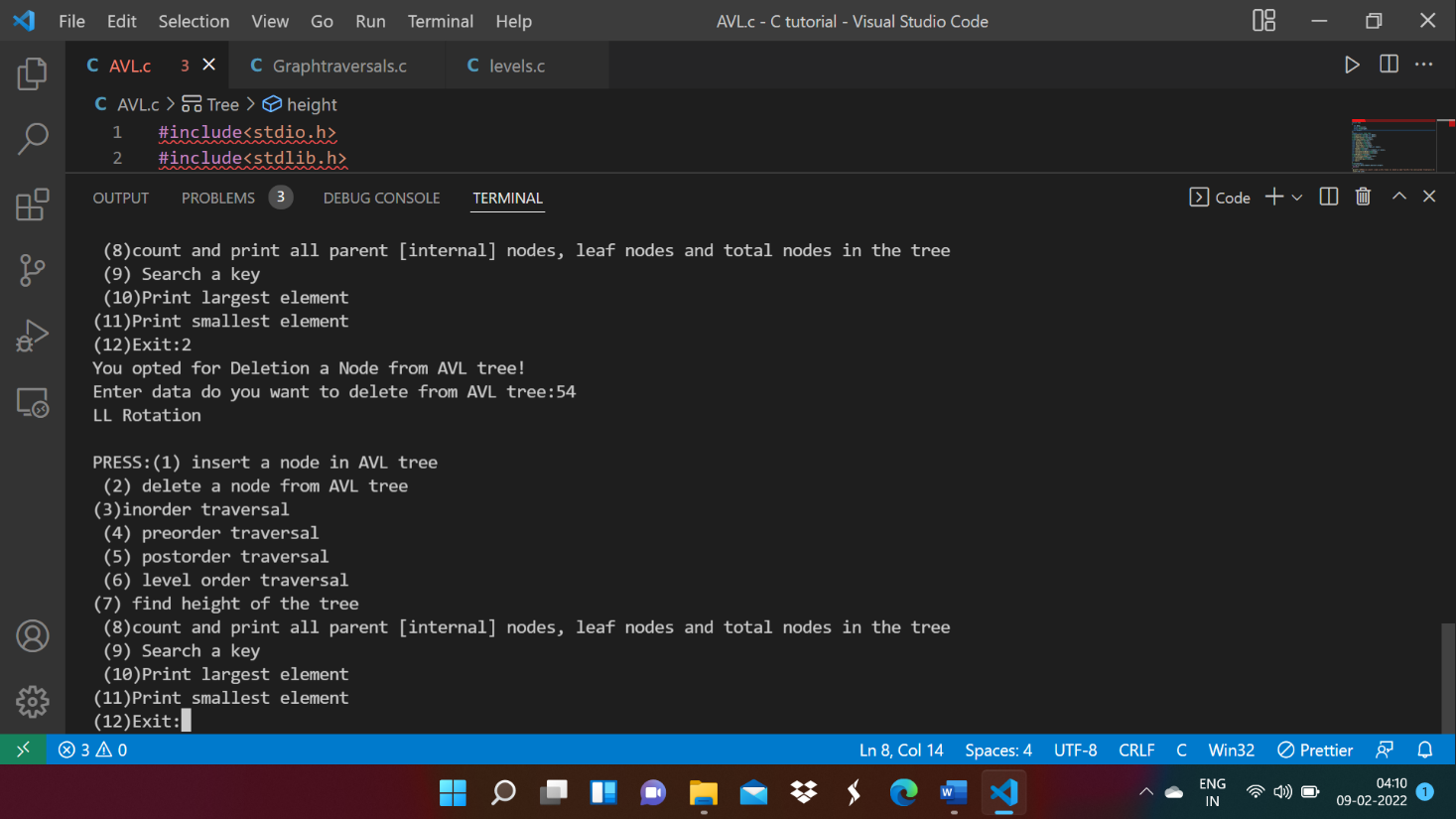


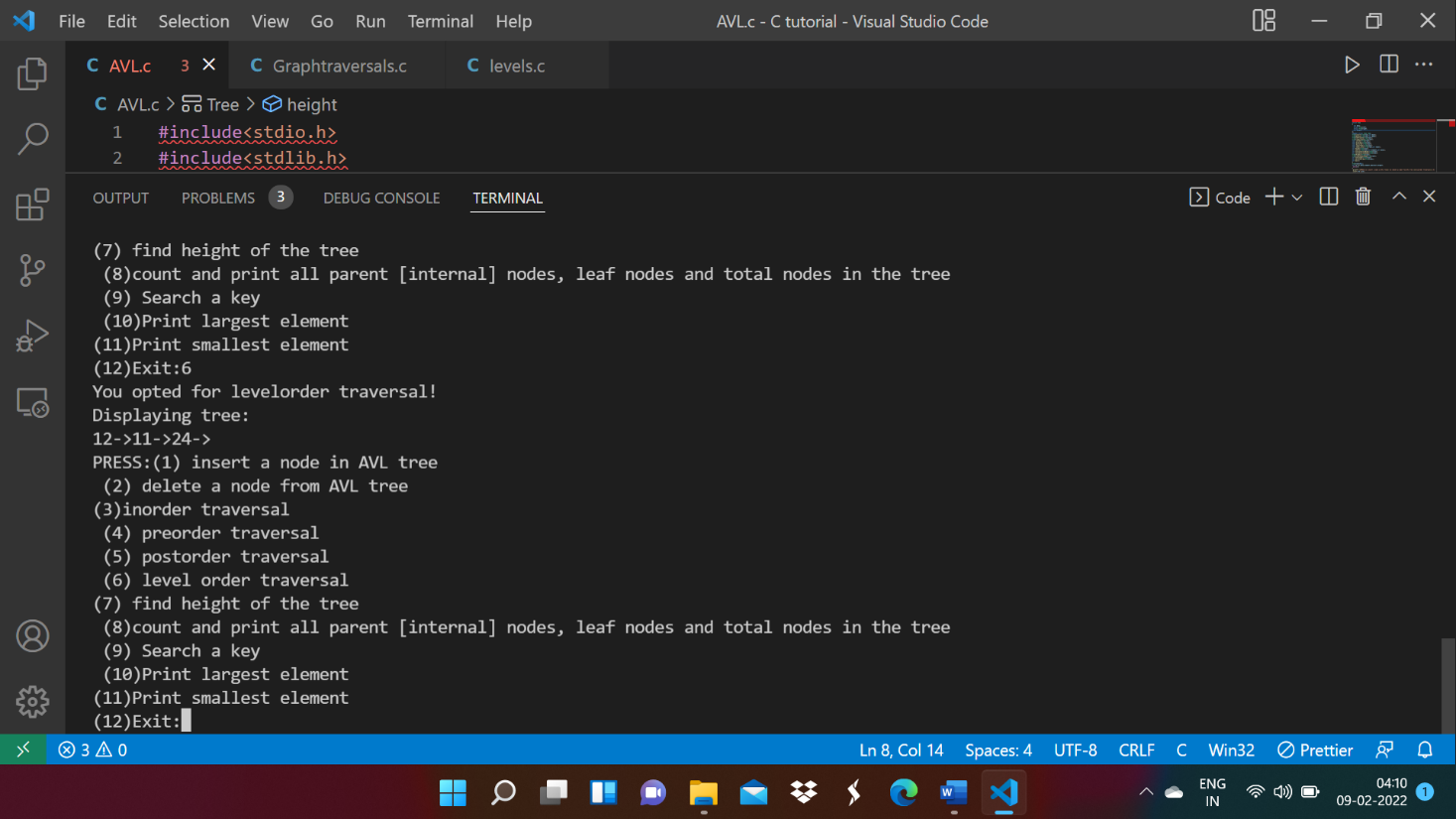












# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# MY EXAMPLES:

# 

# 

# 

# 

# Conclusion:

Through this experiment we come to know:

1. AVL is an extension of Binary Search Tree.

2. It arranges the tree in such a manner so that the tree is balanced from both sides.

3. Thus, it is very useful for searching operation and reduces its complexity.