**Name: Riya Jain**

**Section: A (CSE)**

**Roll No: 17**

**EXPERIMENT: 7**

## Aim: To study binary search tree (BST) and implement various operations on it.

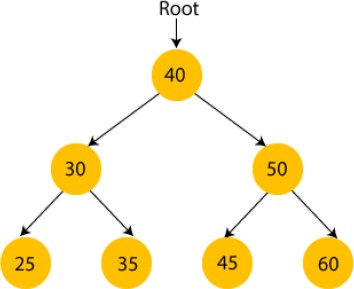
**Problem Definition: Create a self-referential structure, TNode to represent a node of a binary search tree. Implement the routines too - (1) insert a node in BST (2) delete a node from BST (3) perform an inorder walk of the tree, (4) perform a**

**preorder walk of the tree, (5) perform a postorder walk of the tree, (6) find the 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:

A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called nodes that are linked together to simulate a hierarchy. The tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially. A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of the left node must be smaller than the parent node, and the value of the right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.



In the above figure, we can observe that the root node is 40, all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node. Similarly, we can see the left child of the root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search trees. Therefore, we can say that the tree in the above image is a binary search

tree.

**Advantages of Binary search tree:**

* Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
* As compared to array and linked lists, insertion and deletion operations are faster in BST

**Basic Operations**

Following are the basic operations of a tree:

* Search − Searches an element in a tree.
* Insert − Inserts an element in a tree.
* Pre-order Traversal − Traverses a tree in a pre-order manner.
* In-order Traversal − Traverses a tree in an in-order manner.
* Post-order Traversal − Traverses a tree in a post-order manner.

**Algorithms:**

* 1. **Insert(T, VAL):**

1. IF TREE = NULL Allocate memory for TREE T →DATA = VAL

T → LEFT = T →RIGHT = NULL

1. ELSE

IF VAL < T →DATA

T → LEFT = Insert(T →LEFT, VAL) ELSE

T →RIGHT = Insert(T →RIGHT, VAL)

1. Return T

**2. Deletion(T,VAL):**

1. IF T = NULL

Write "VAL not found in the tree"

1. ELSE IF VAL < T → DATA

T → LEFT =Delete(T → LEFT, VAL)

1. ELSE IF VAL > T → DATA

T → RIGHT =Delete(T → RIGHT, VAL)

if key is same as root's key, 4.ELSE

IF T → LEFT AND T → RIGHT

TEMP = LargestElement (T → LEFT) T → DATA = TEMP → DATA Delete(T → LEFT, TEMP→DATA) ELSE

TEMP = T

IF T → LEFT = NULL AND T → RIGHT = NULL T = NULL

ELSE IF T → LEFT != NULL T = T → LEFT

ELSE

T = T → RIGHT FREE TEMP

5. 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

**CODE**:

#include<stdio.h>

#include<stdlib.h>

struct Tree

{

  int data;

  struct Tree\*left;

  struct Tree\*right;

};

typedef struct Tree tree;

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

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

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 height);

int totalInternalNodes(tree\*root);

int totalExternalNodes(tree\*root);

tree\*Findmin(tree\*root);

tree\*findLargestNode(tree\*root);

int totalnodes(tree\*root);

int main()

{

tree\*root=NULL;

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

 while(l)

{

 printf("\nPRESS:(1) insert a node in BST\n (2) delete a node from BST \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 BST!\n");

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

        scanf("%d",&data);

        root=insert(root,data);

        break;

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

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

        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 BST are: %d",res1);

        int res2=totalExternalNodes(root);

        printf("\nThe total External nodes (leaf nodes) present in BST 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 BST:");

        scanf("%d",&element);

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

        if(res==0)

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

        else if(res==1)

        printf("\nSearched element %d is present in BST",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;

      root=ptr;

      return root;

  }

   if(data<root->data)

   {

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

   }

    if(data>root->data)

   {

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

   }

   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);

 }

}

return root;

}

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 present 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);

    }

    }

}

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); //root(parent) node

    }

}

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)

    {

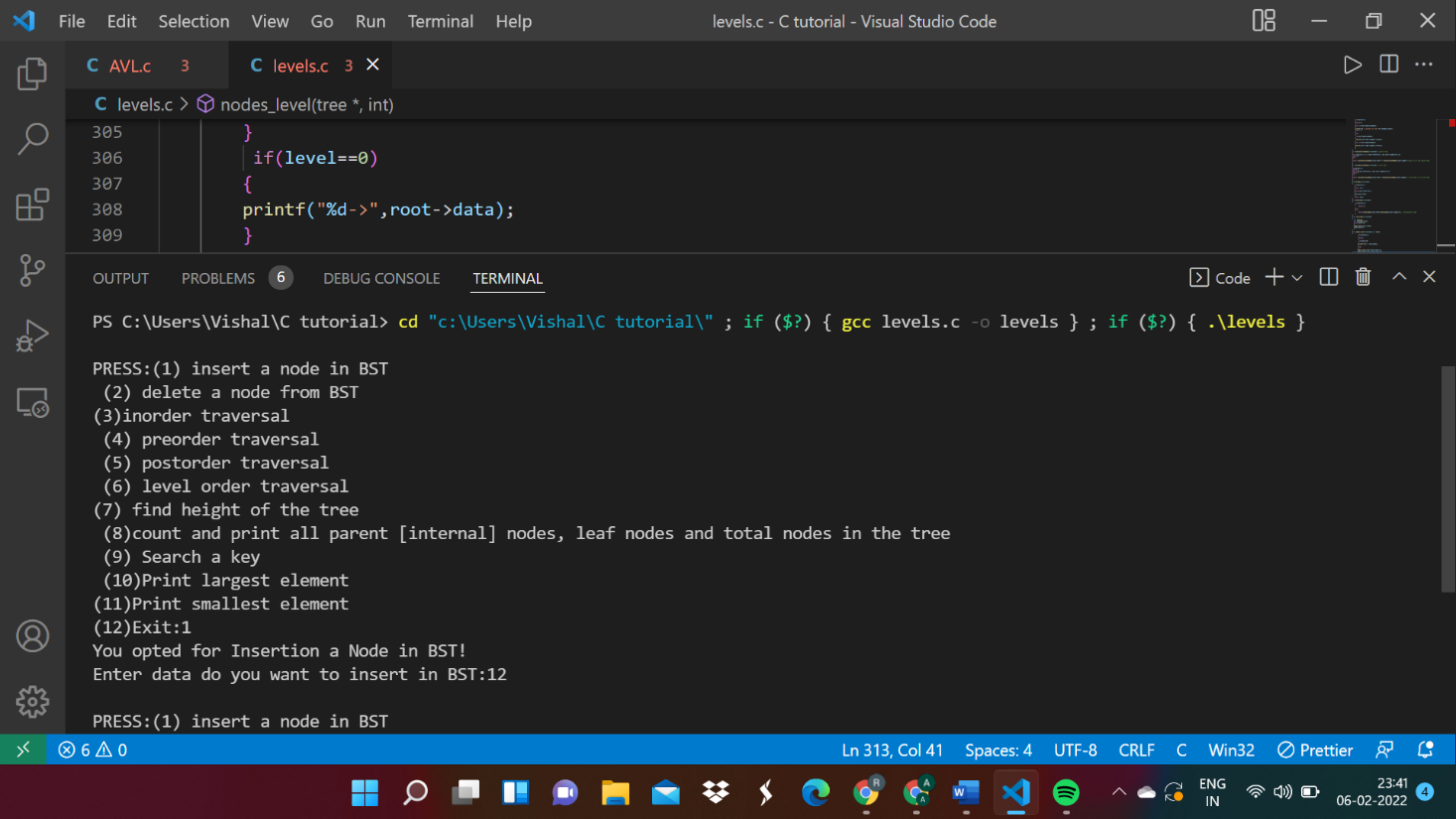
        root=root->right;

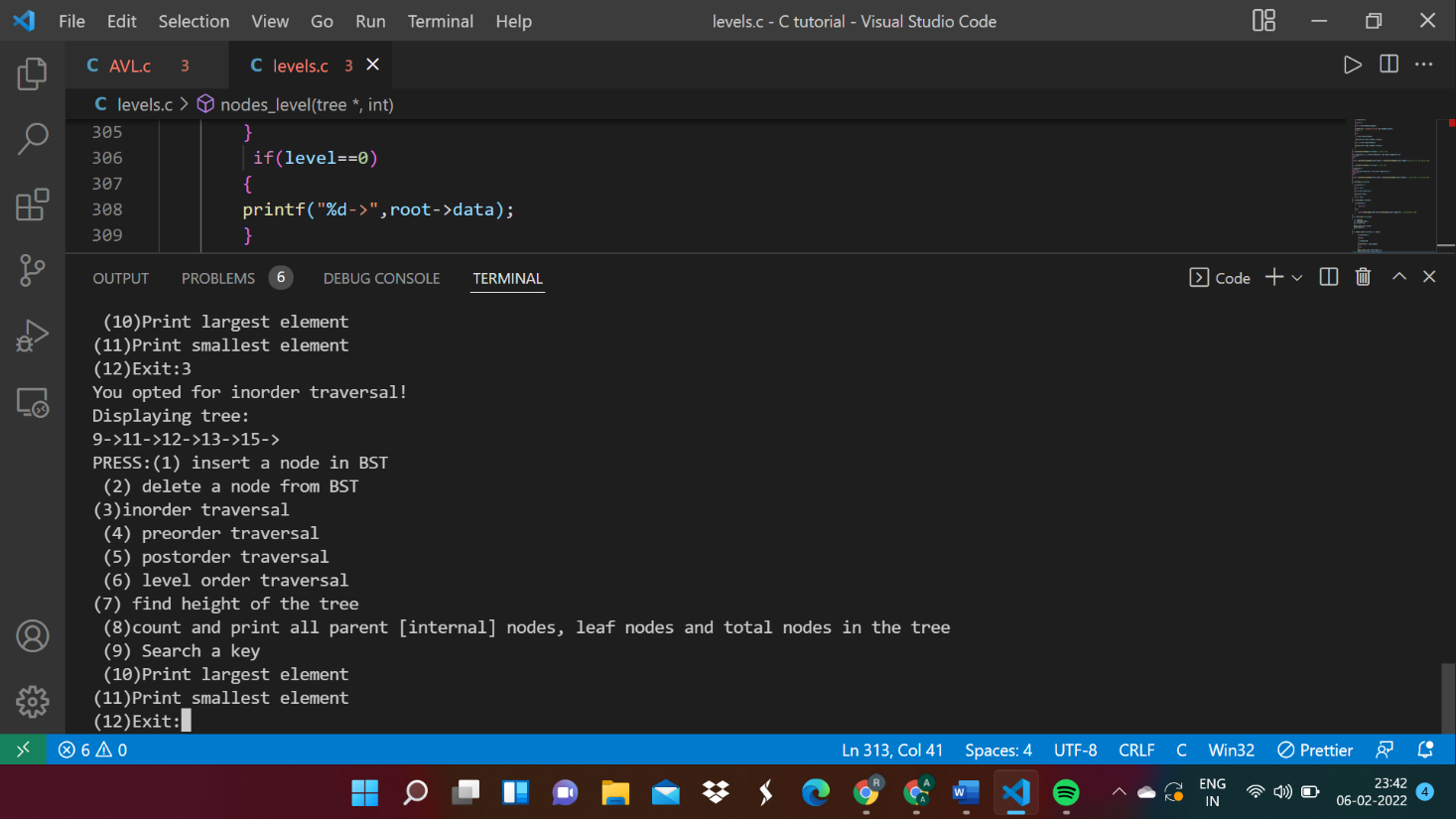
    }

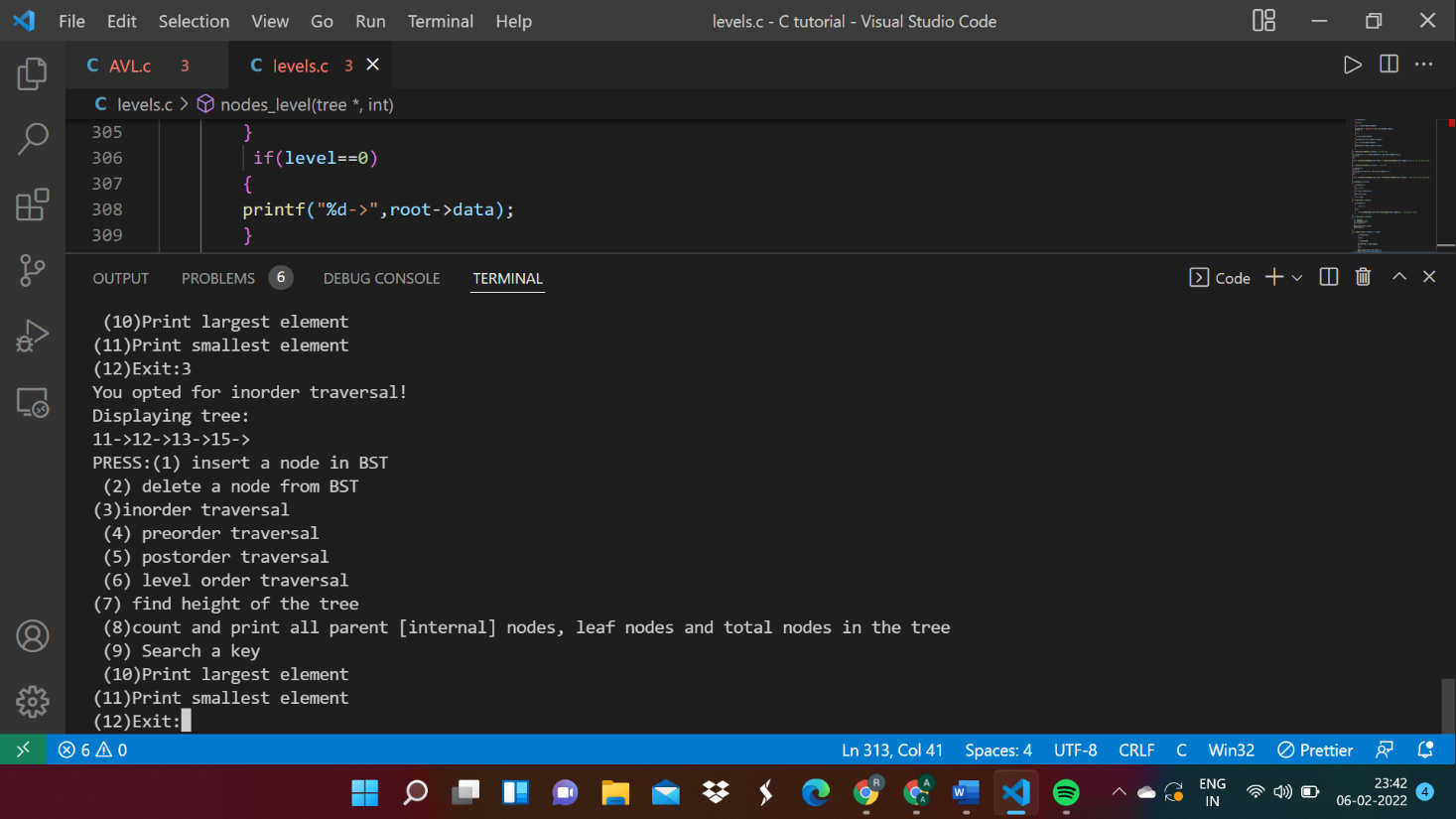
    return root;

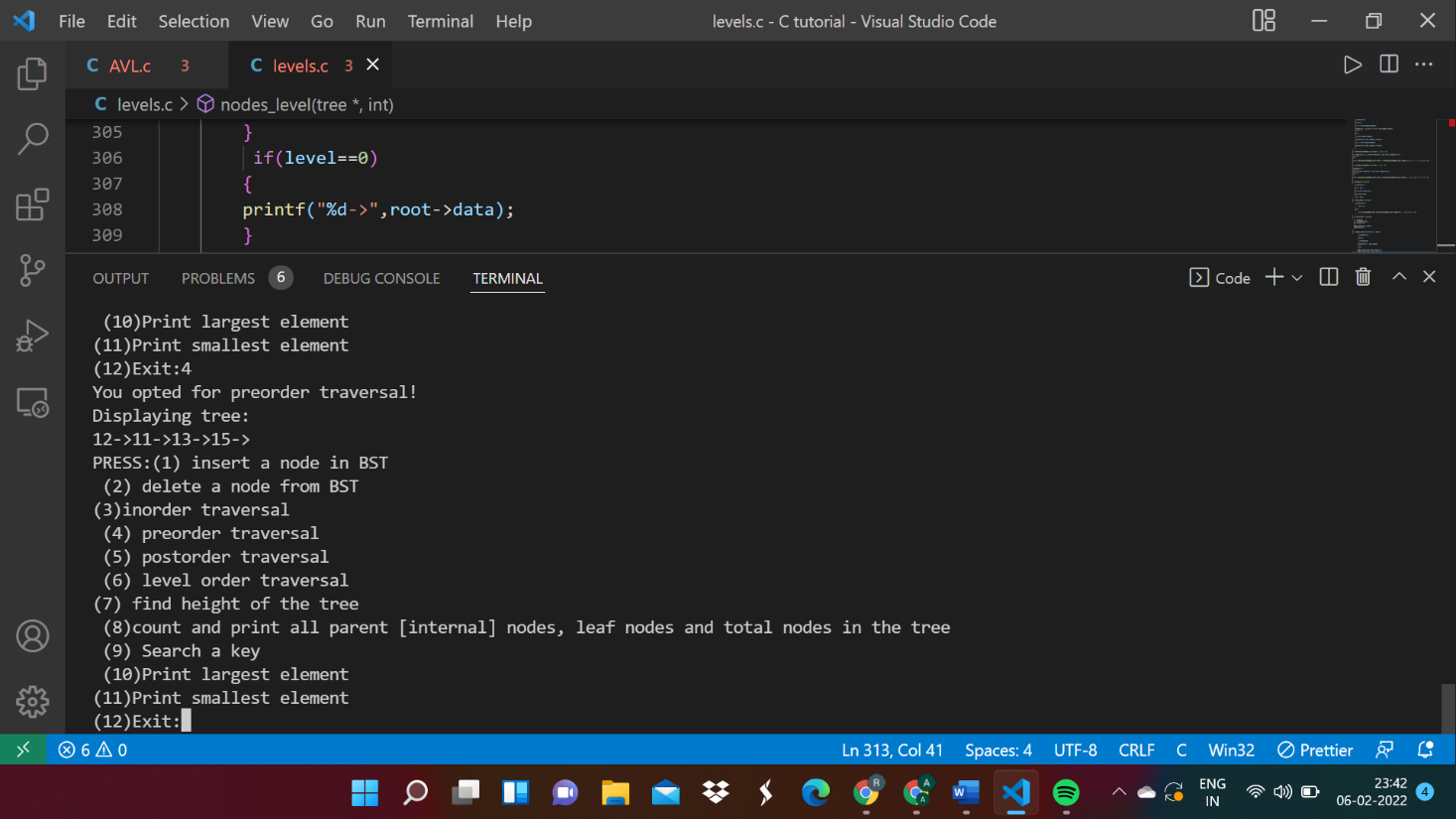
}

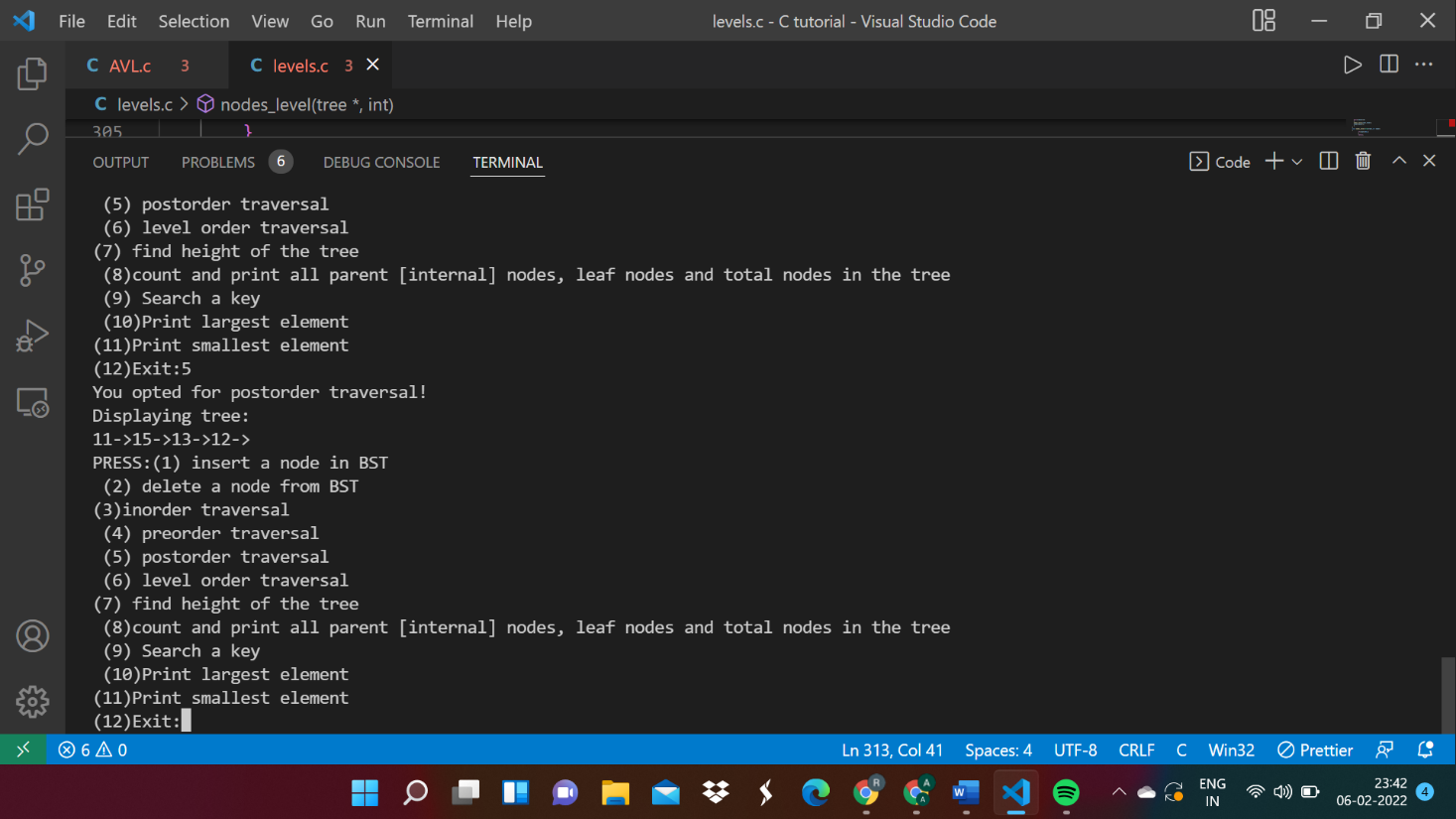
**OUTPUTS:**

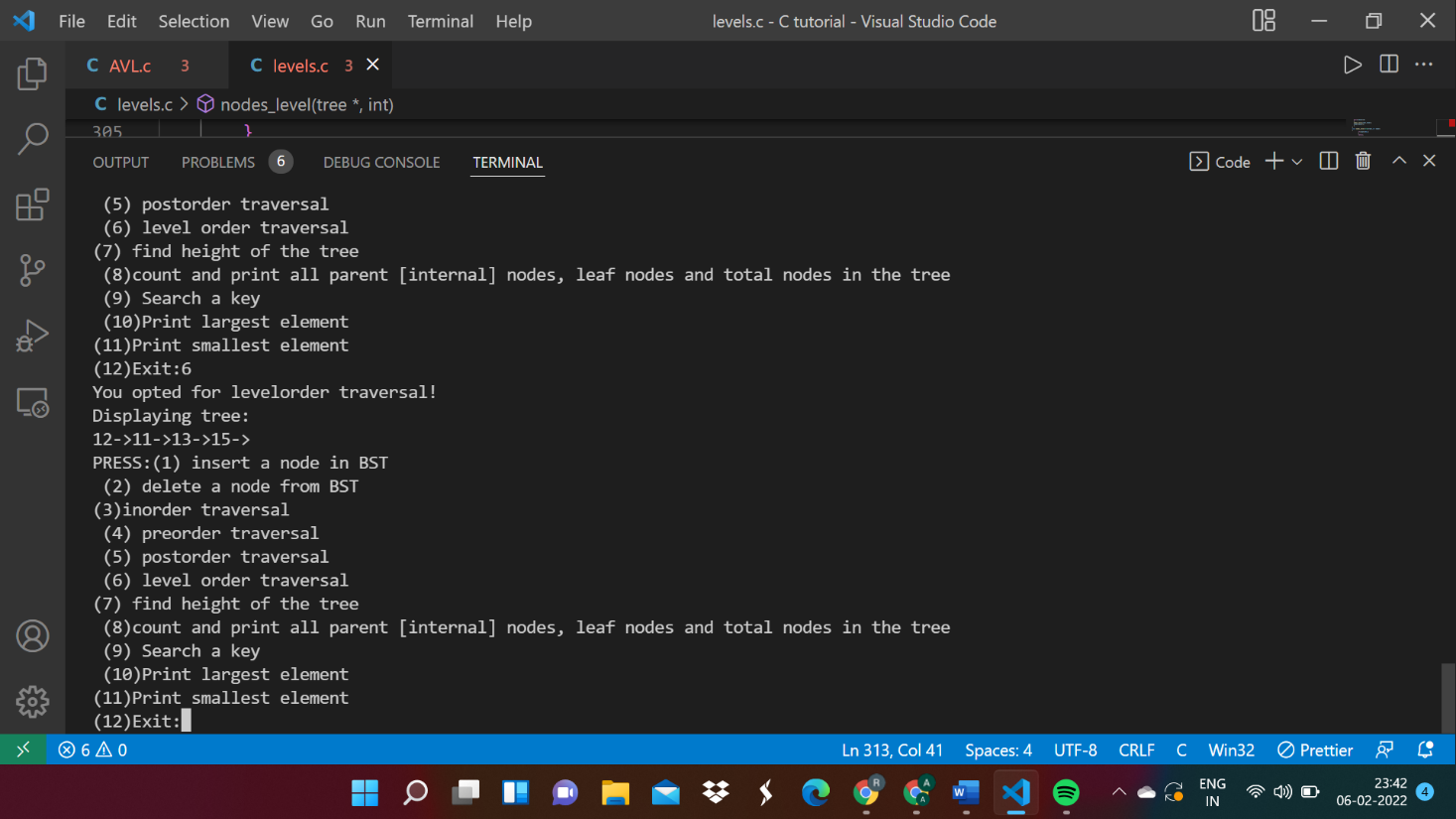


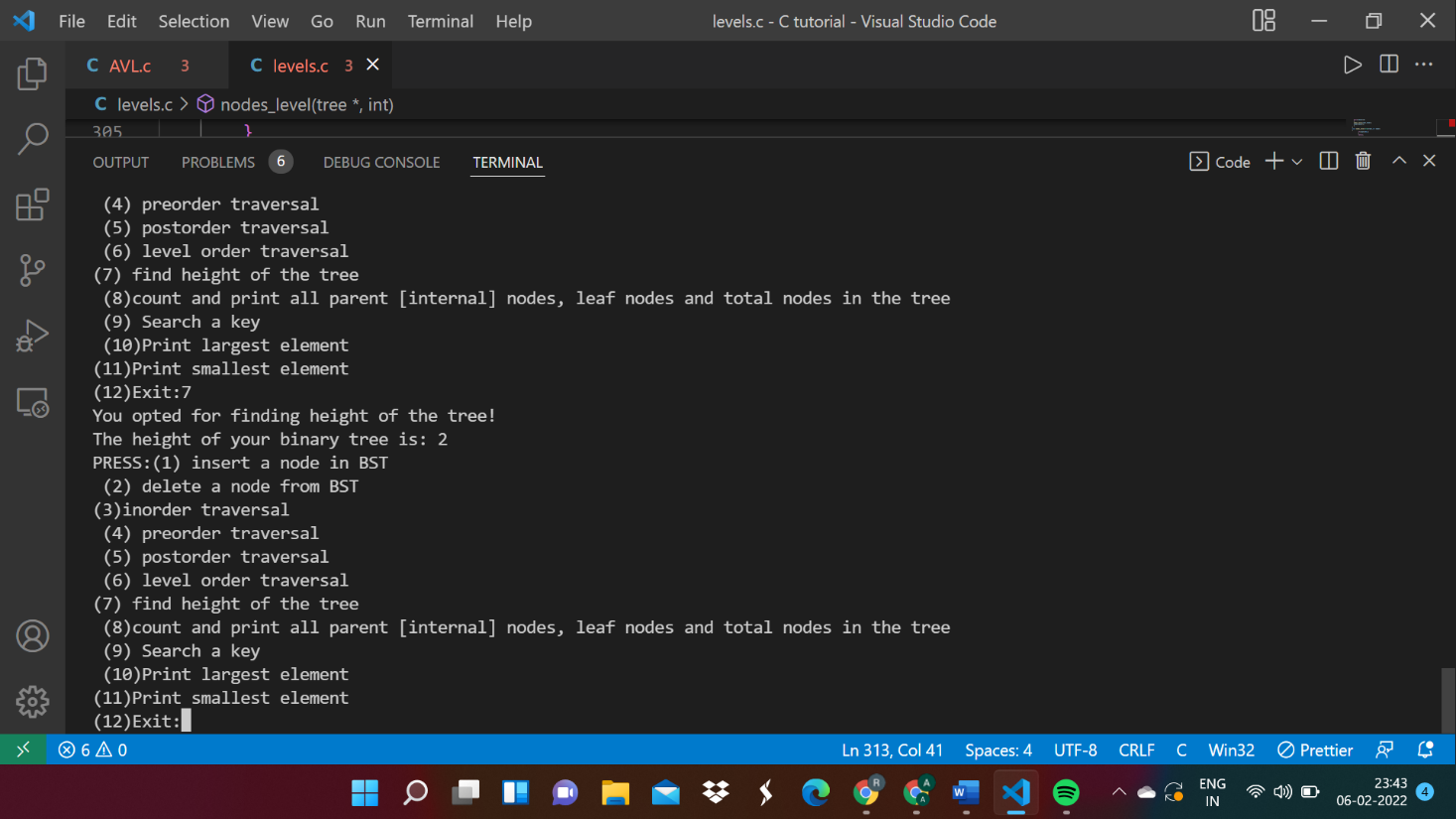


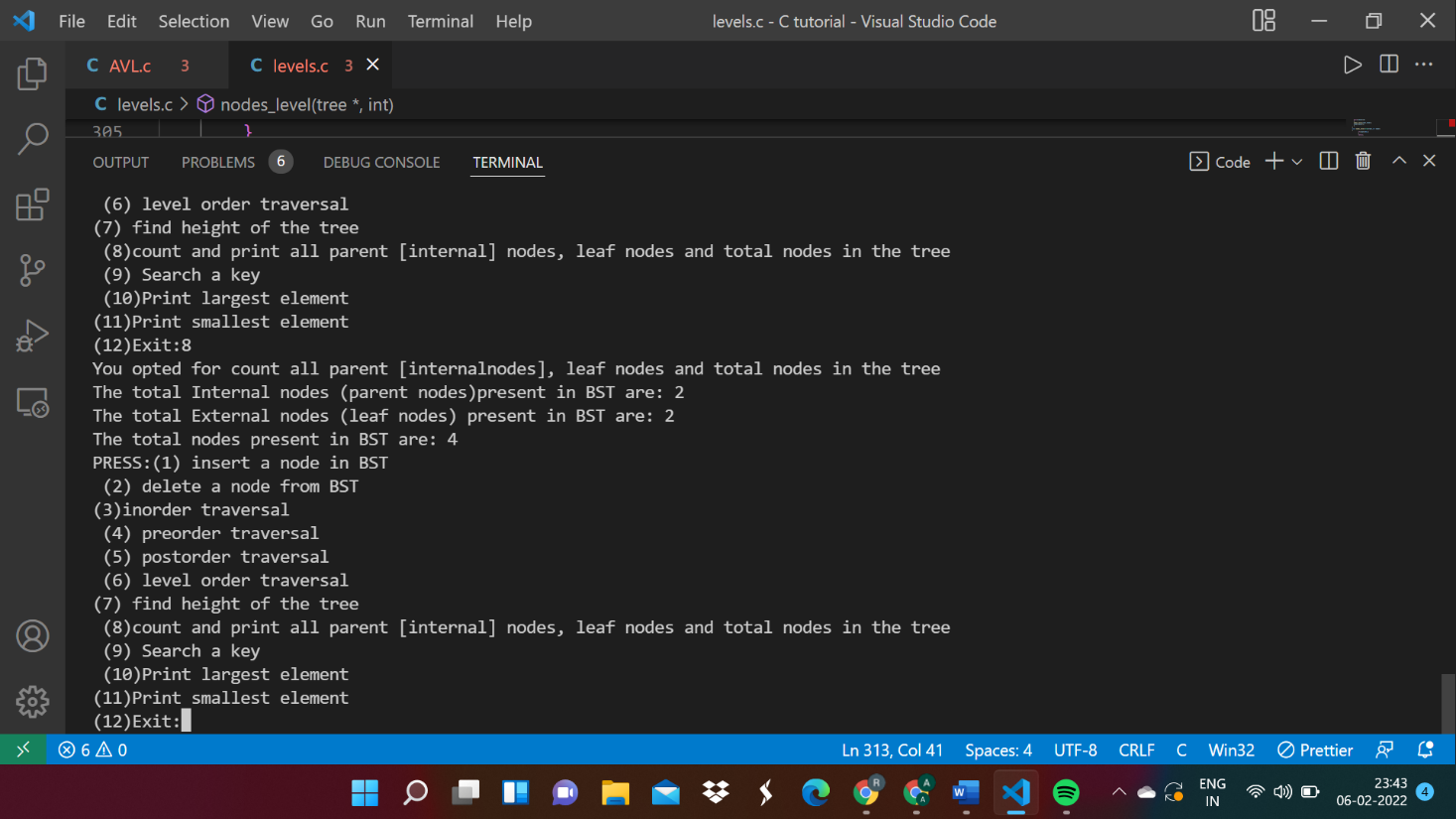


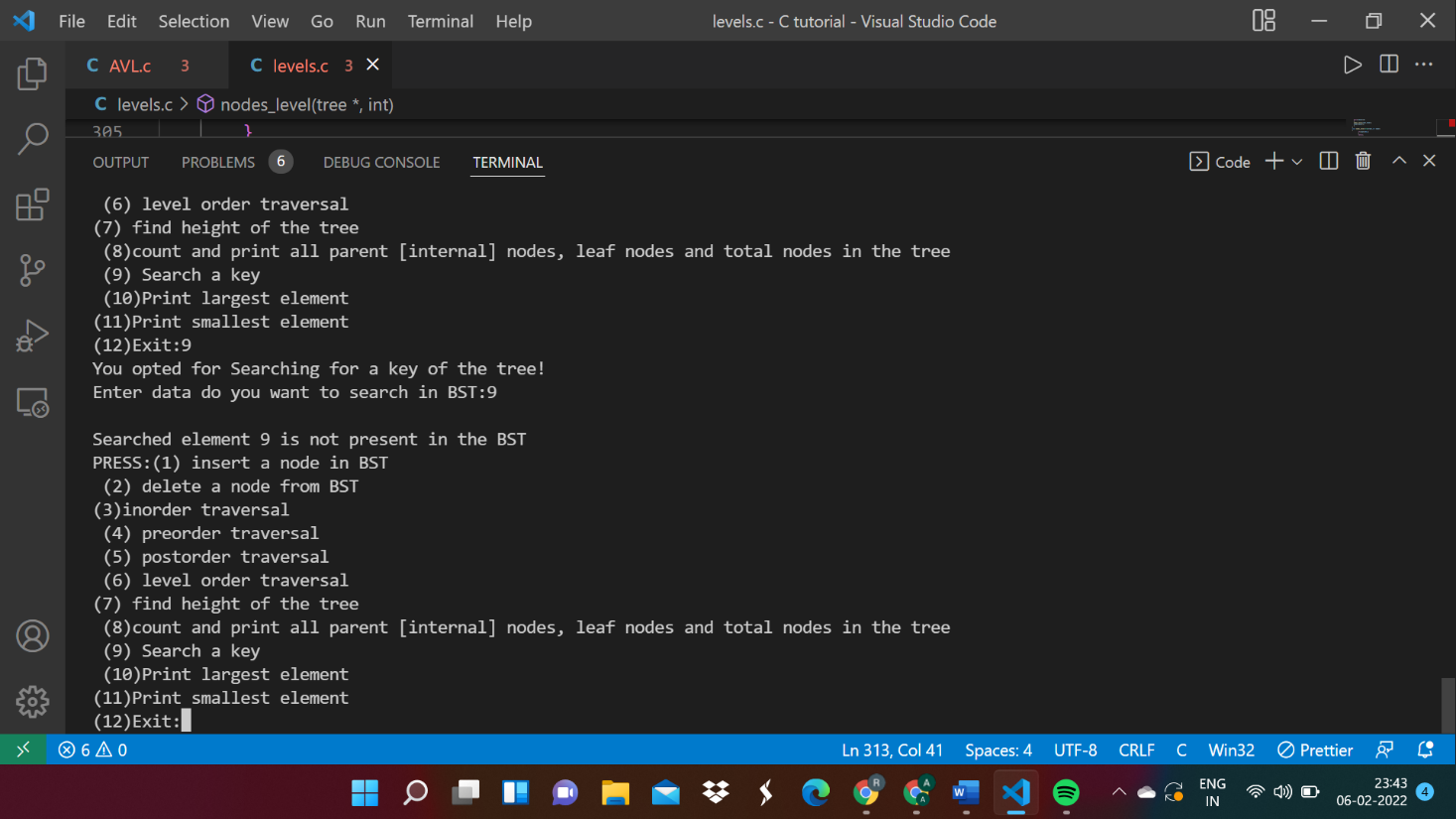


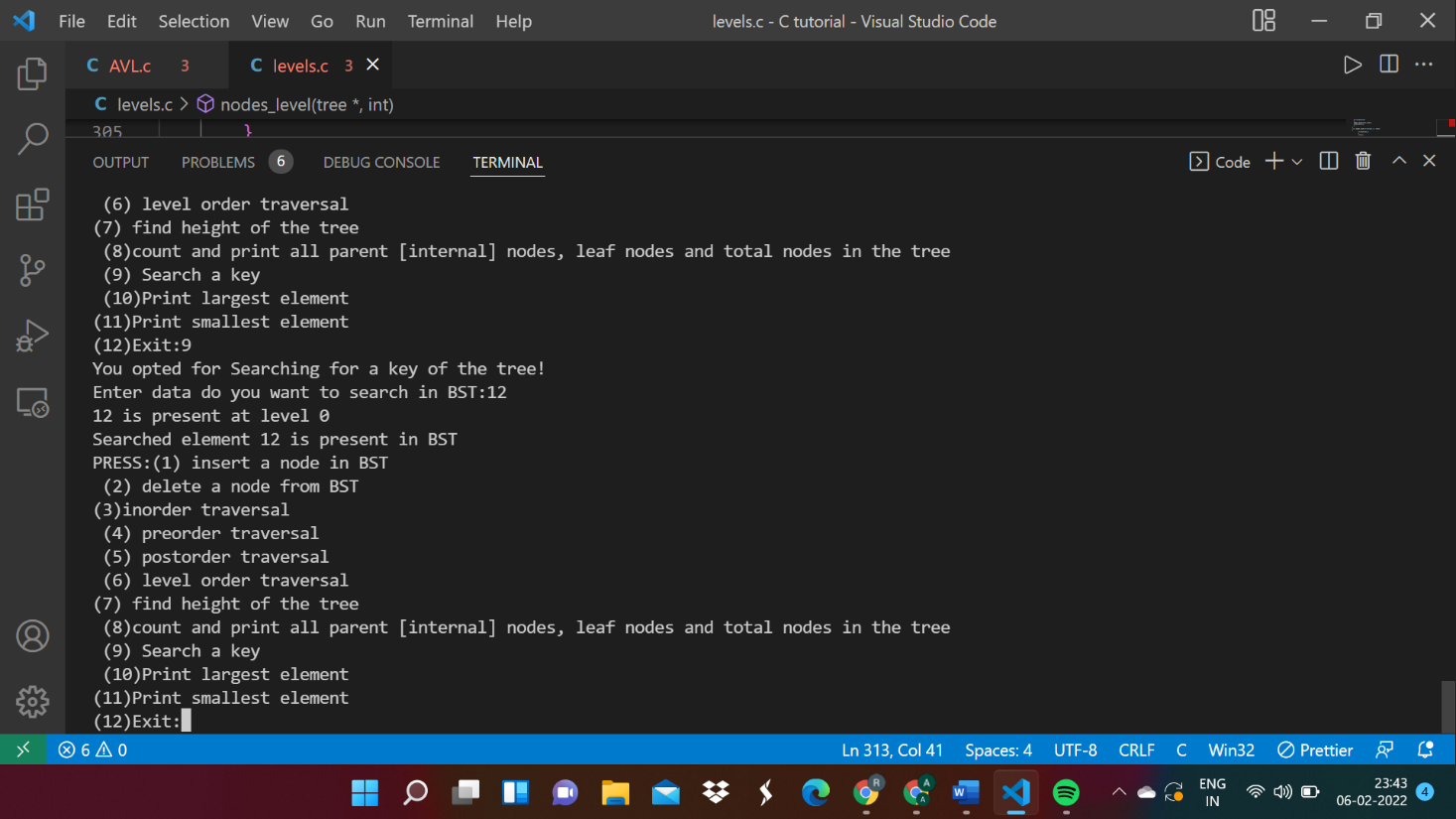


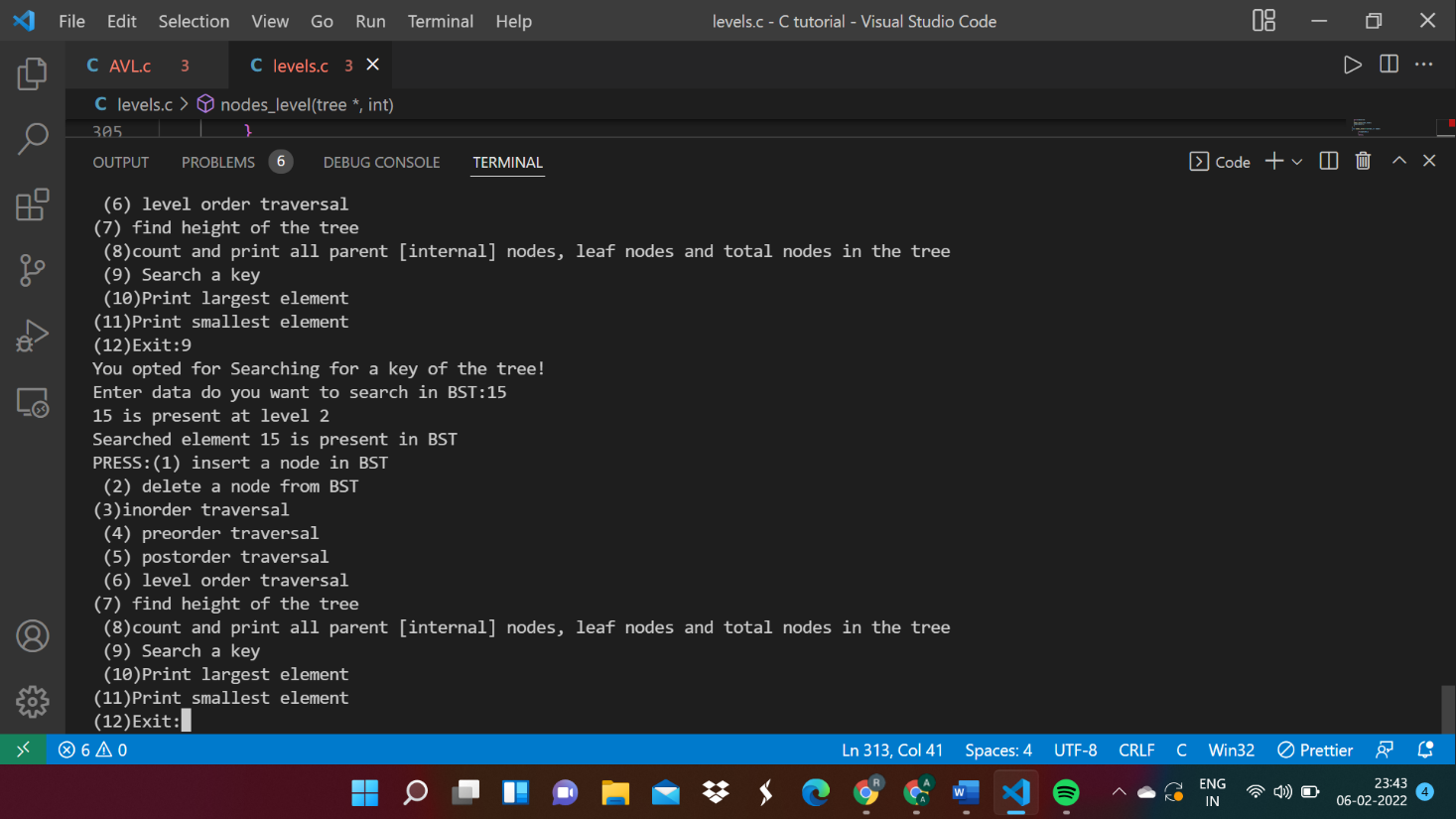


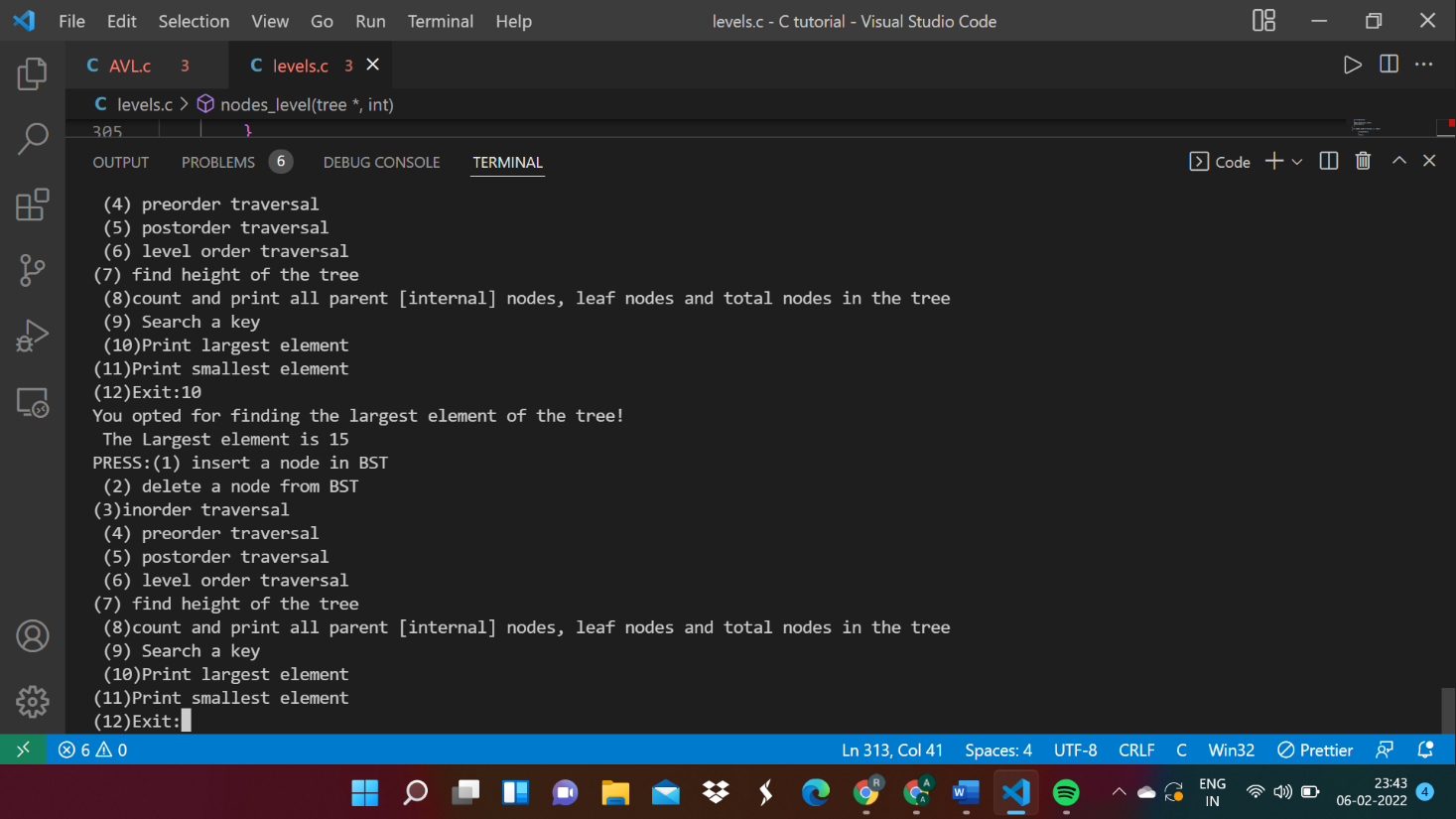


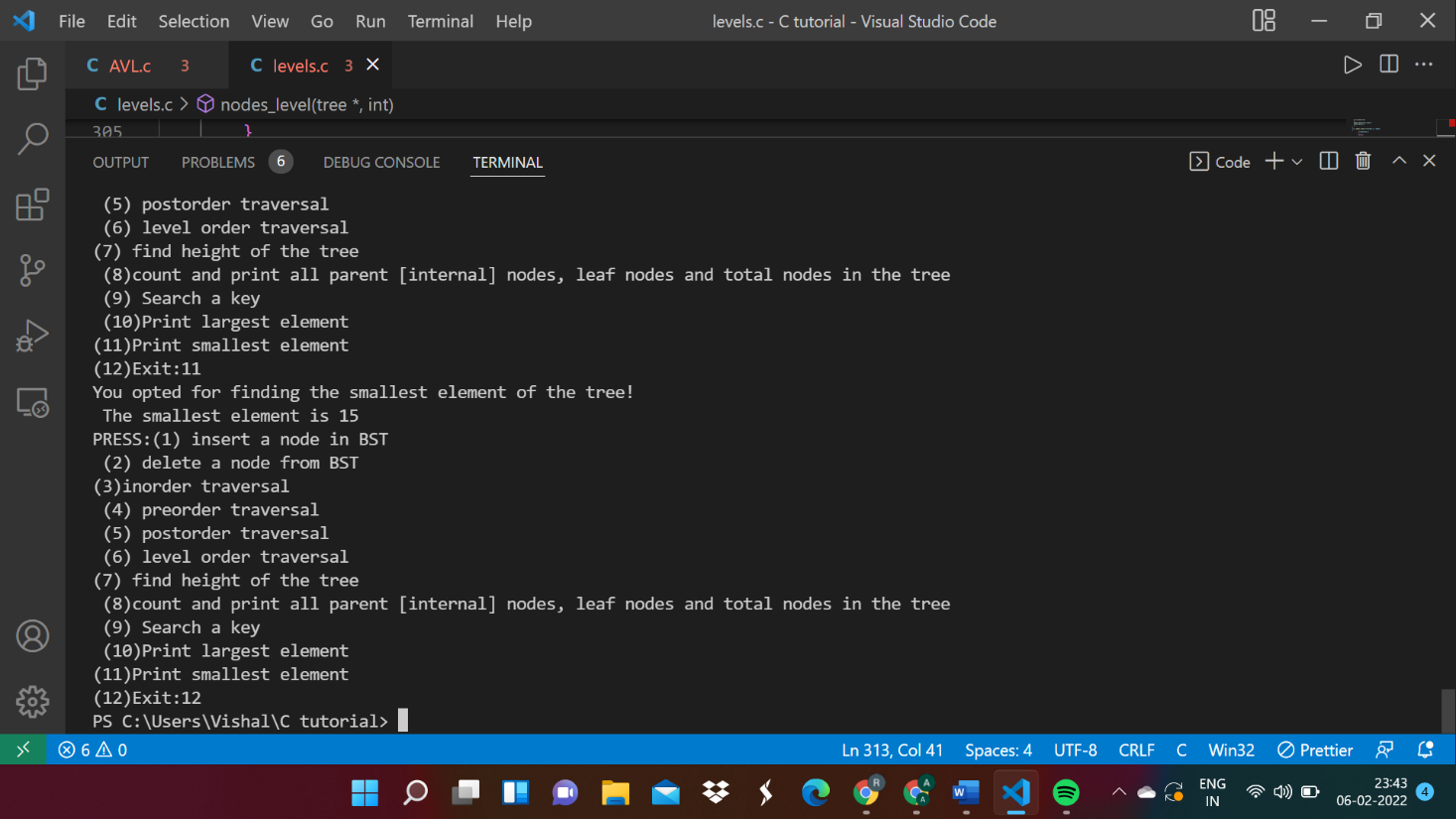












# Conclusion:

In this experiment, we implemented various operations on Binary Search Tree (BST).

We know that Binary Search Tree is a node-based binary tree data structure. It is called a search tree because it can be used to search for the presence of a number in

O(log(n)) time.

The following are some of the properties of the Binary Search Tree:

* 1. The left subtree of a node contains only nodes with keys lesser than the node’s keys.
  2. The right subtree of a node contains only nodes with keys greater than the node's key.
  3. The left and right subtree each must also be a binary search tree.

BSTs are used for a lot of applications due to their ordered structure. They are as follows:

1. BSTs are used for indexing and multi-level indexing.
2. They are also helpful to implement various searching algorithms.
3. It is helpful in maintaining a sorted stream of data.
4. TreeMap and TreeSet data structures are internally implemented using self-balancing BSTs

In conclusion, Binary Search Trees are a very powerful (but not perfect) data structure to have in your programming tool belt. If done correctly, handling large amounts of sorted data becomes quick and easier.