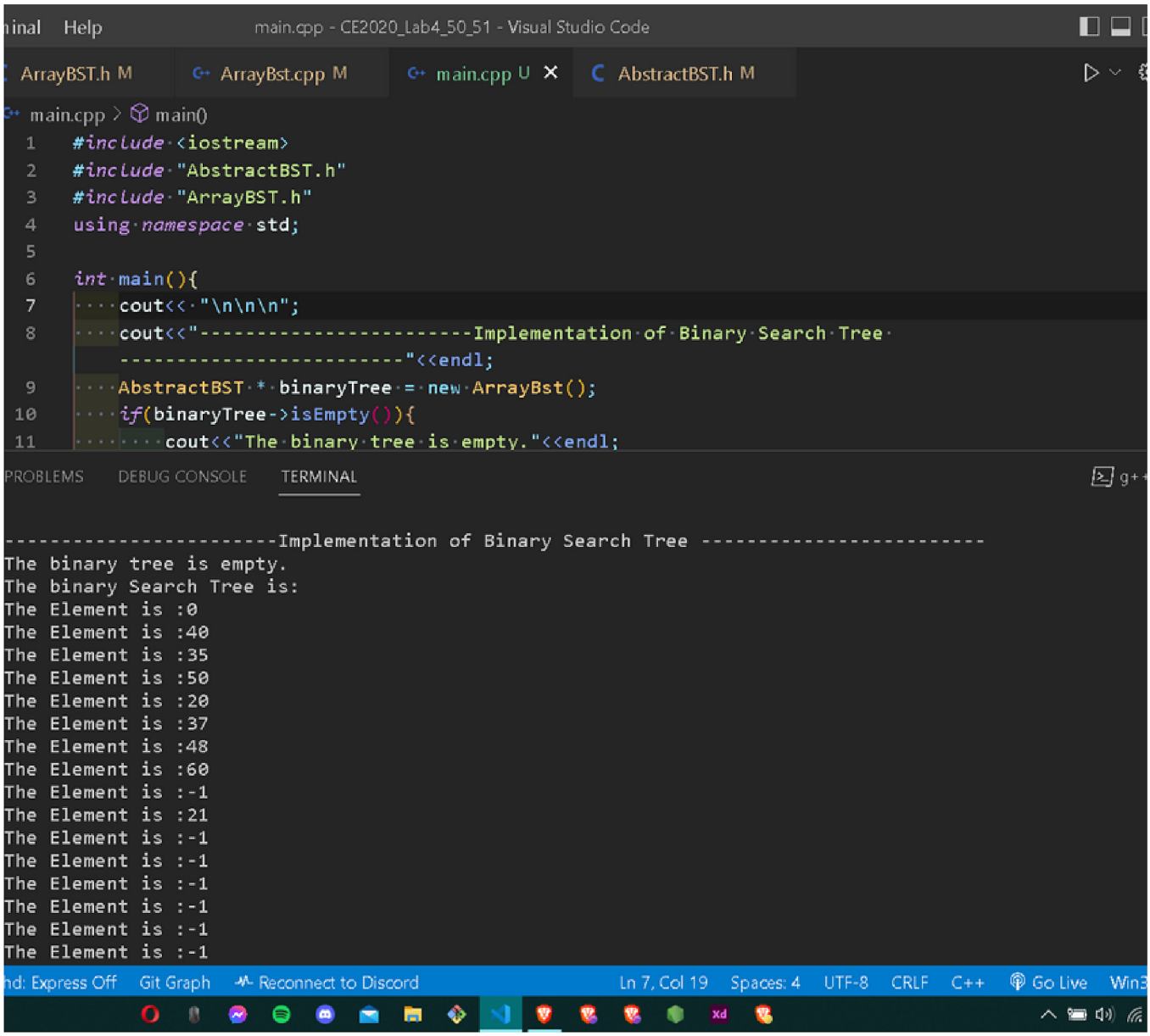
LAB 4

Implementaion of A BST

1.isEmpty()

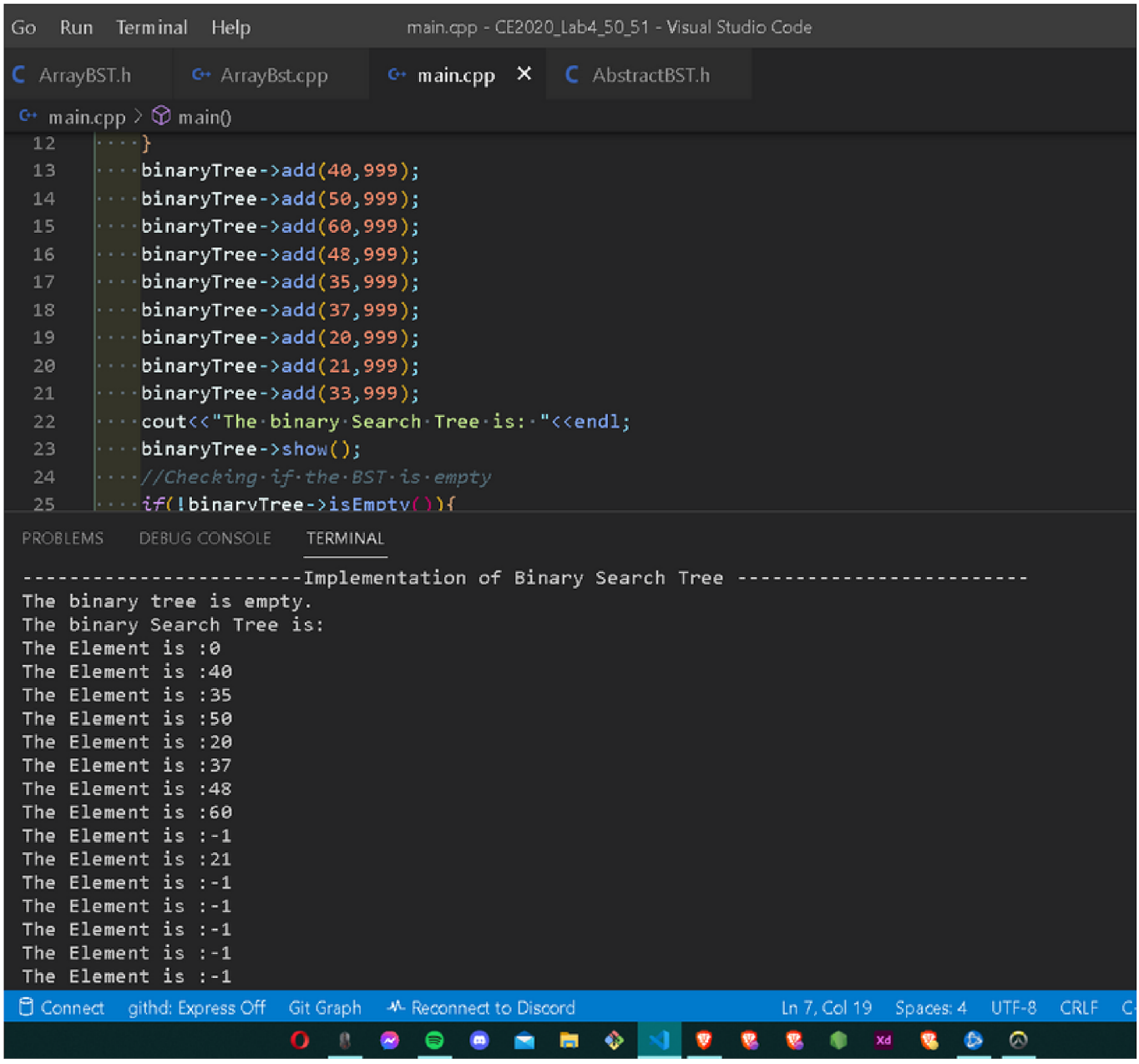
The function returns true if the search Tree is empty.
With ARRAY:
The function checks if the element at the first position is -1 (considering -1 as a null value). IF so the function returns true.



2.Add(key,value)

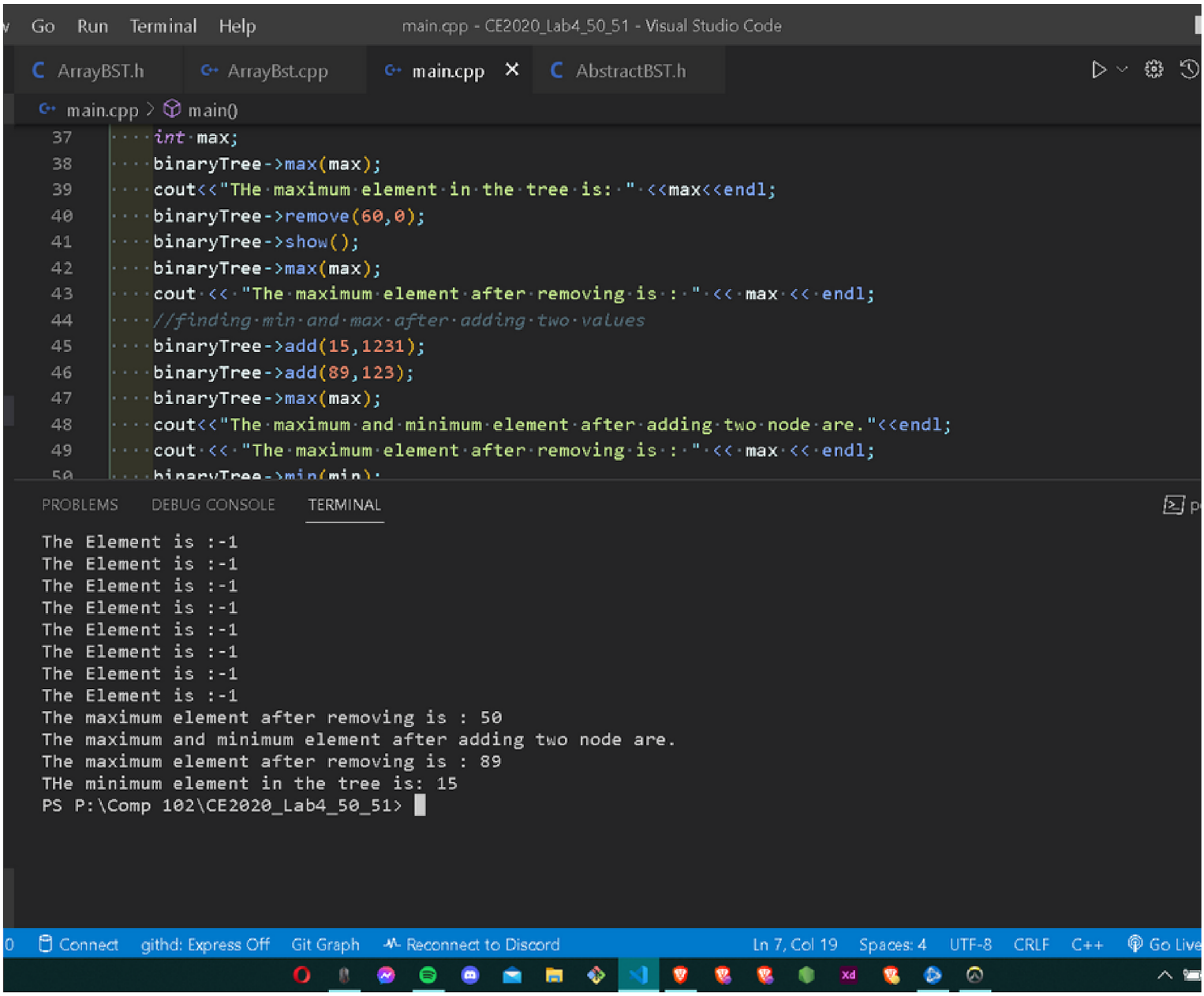
The function adds an element on the correct position in the binary search tree.

At first the function checks if the tree is empty .A while loop runs until the key is -1: If so the new node is added as the root of the tree. If the tree is not empty, the key of the node ToBeAdded is compared with the key of the current root.
if root.key > nodeToBeAdded.key :
New root is set to be the left child:
else:
new root is set to be the right child:
This process is iterated until an empty node is found.



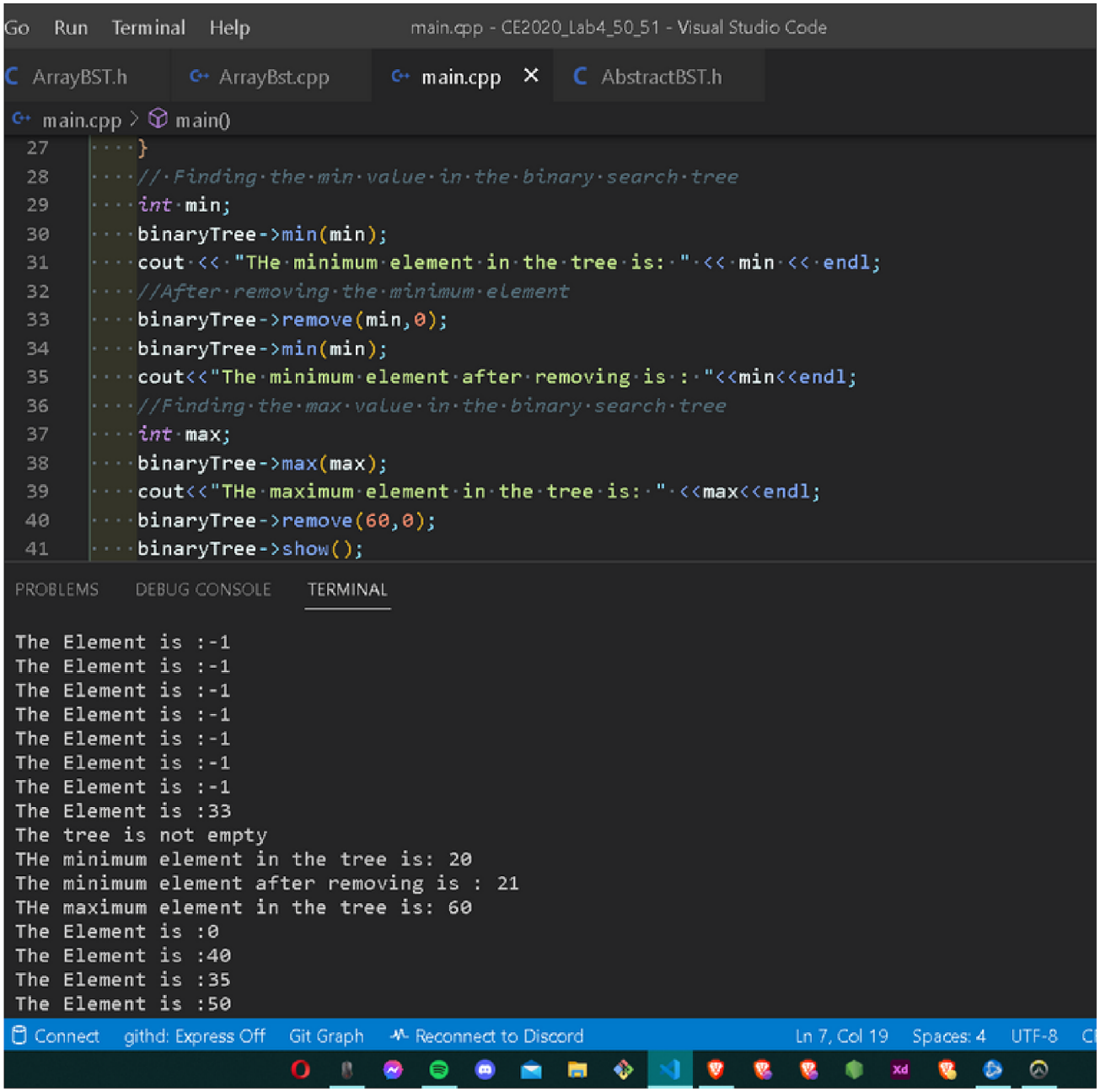
3.max()

A property of a binary search tree states that the element greater than the node is kept as the right child of the parent node. So the leftmost node is the maximum element in the binary search tree.
A for loop runs until the left most element is figured out.



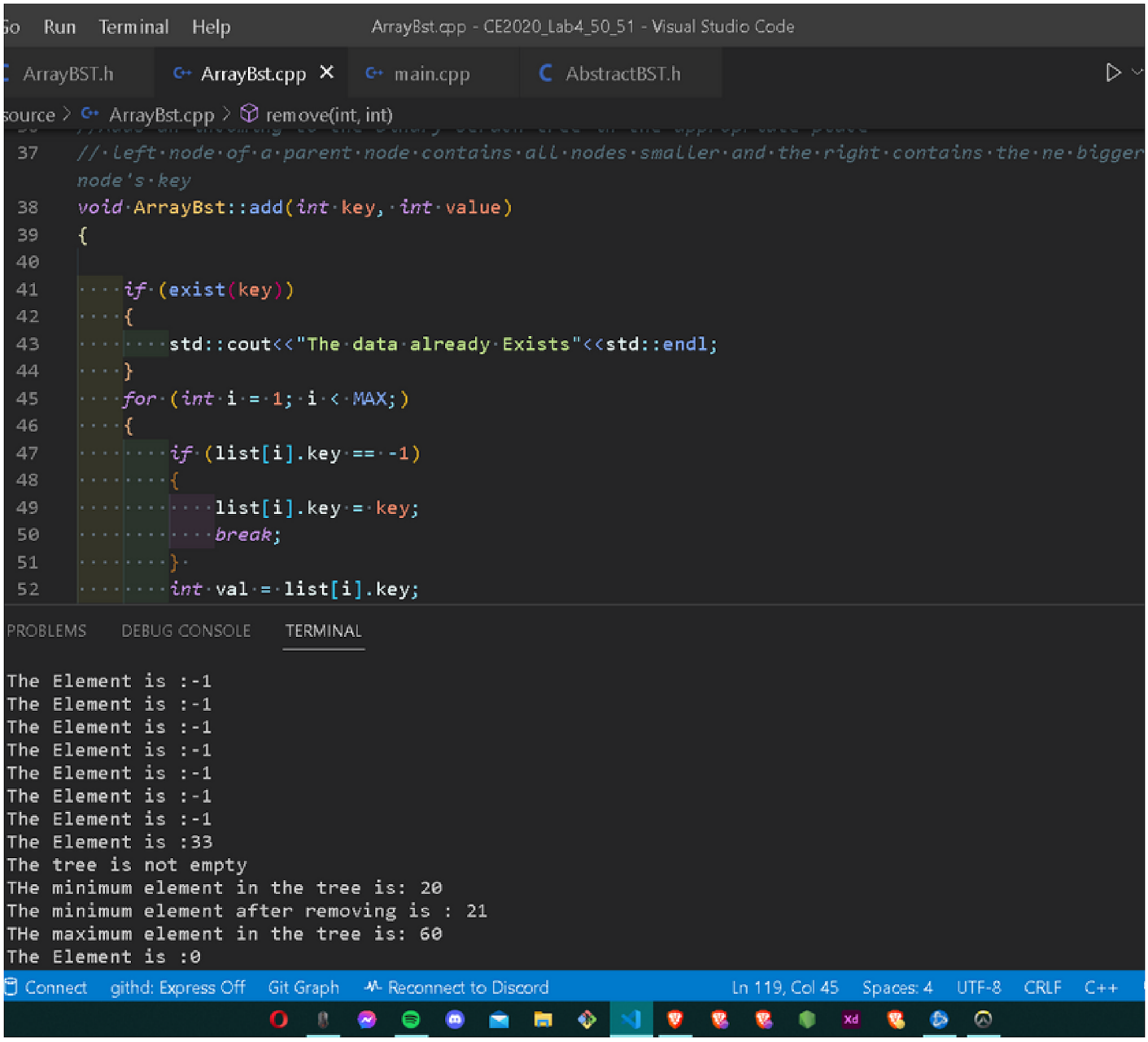
4.min()

A property of a binary search tree states that the element smaller than the node is kept as the left child of the parent node. So the rightmost node is the minimum element in the binary search tree. A for loop runs until the righ most element is figured out.



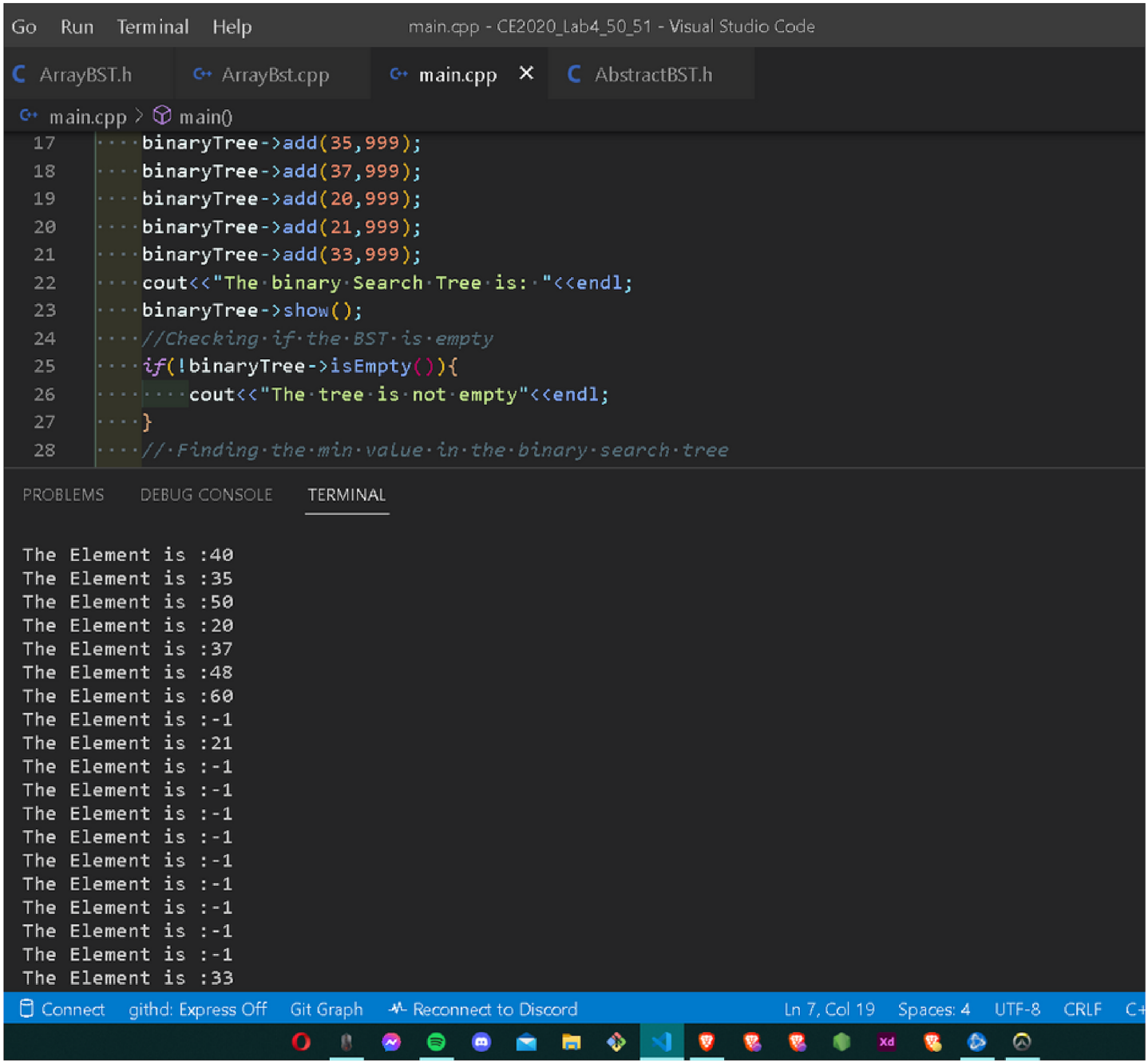
5.exist()

The exist function works similar to that of add. Only exceptions is that the key of node is also checked if it is equal to that of the root. If so true is returned , else iterates until a null node is found and false is returned in that case.



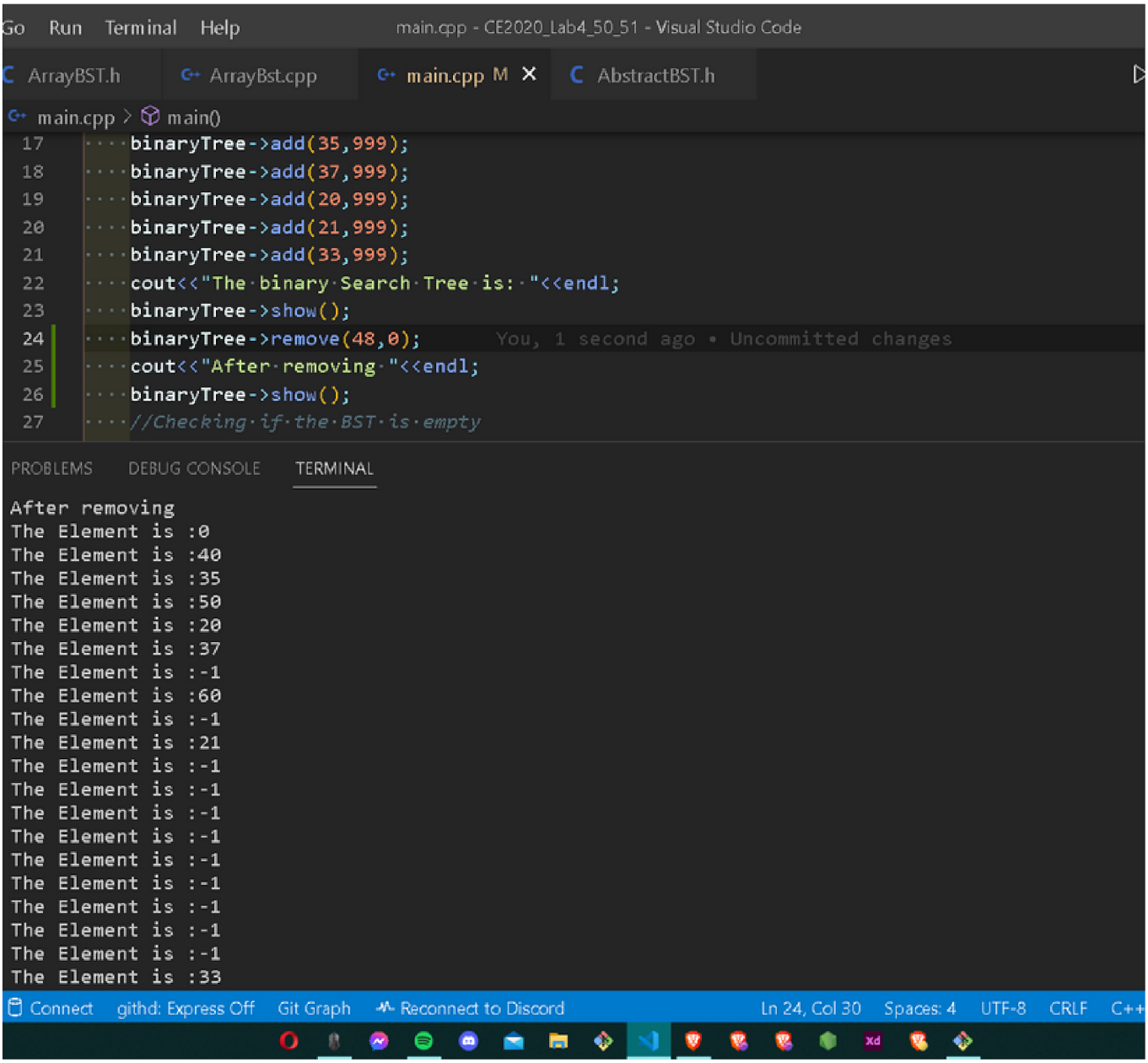
6.show()

The function iterates through each element of the array and prints it in the console. The null node is also printed in this case.



7.remove()

This function works similar to the exist function in the aspect of finding the node to be deleted.
After the node is found , node can have :
i) One child:
In this case , the node is removed and the child is kept as the new node in the parent's place. . And a function call i s made to delete child.
ii) Two children.
In this case the largest element from the left sub tree replaces the node to be deleted. And a function call i s made to delete the node with the largest element.
iii)No child:
The node is just deleted.



Lab Report: Lab 4

Linked List Implementation of BST

1. Bool isEmpty():

This function checks whether the tree is empty or not. If the tree is empty it returns 0 otherwise it returns 1.

```
//function to determine if the tree is empty or not//
bool  BST_linkedlist ::isEmpty (){
    if (root == NULL)
    {
        return true;
    }
    else {
        return false;
    }
}
```

2. Bool searchBST(*int targetkey*)

This function checks whether the key inserted is present in the **BST** or not. If the key is present, it returns 1(true) with a message “the Key is in the tree”, whereas returns 0(false) with a message “the key is not in the tree” if not

present.

```
bool BST_linkedlist ::searchBST(int targetkey){//function to check whether the key is present in the tree or not
    if(isEmpty()){
        cout<<"The tree is empty"<<endl;
        return 0;
    }
    node*temp = root;
    while(temp != NULL){
        if (temp -> key == targetkey) {
            cout<<"The key is in the tree"<<endl;
            return true;
        }
        else if(temp->key>targetkey){
            temp=temp->left_child;

        }else if(temp->key<targetkey){
            temp=temp->right_child;
        }
    }
    cout<<"The key is not in the tree"<<endl;
    return false;
}
```

3. Void addBST(*node *targetnode, node *newnode*):

This function adds node into the tree. The targetnode is root by default,

where as the newnode is the node to be added.

```
void BST_linkedlist :: addBST(node *targetnode, node *newnode){//function to add node into the tree

    if(this->root==NULL){
        root = newnode;

        cout<<"Root added "<<endl;
        //showData(root);
    }

    else if(targetnode->key > newnode->key){
        if(targetnode->left_child == nullptr) // if left child's position is empty
        {
            targetnode->left_child = newnode;
        }
        else
        {
            addBST(targetnode->left_child, newnode);
        }
    }
    else {
        if(targetnode->right_child == nullptr) // if right child's position is empty
        {
            targetnode->right_child = newnode;
        }
        else
        {
            addBST(targetnode->right_child, newnode);
        }
    }
}
```

4. Void adddata(*int key, int value*):

This function passes the key and the data of the node that is to be inserted into the tree.

```
void BST_linkedlist ::adddata(int key, int value){ //function to add data into the tree
    node *addednode = new node();
    addednode ->key = key;
    addednode ->data = value;

    addBST(root, addednode);
}
```

5. Void removeBST(*int keyToDelete*):

This function is supposed to remove the node with the given key.

```
88  void BST_linkedlist ::removeBST(int keyToDelete)// function to remove the key from the tree
89  {
90      node *temp = root;
91      if(temp->key == keyToDelete){
92          node *temp2 = temp->left_child;
93          node *parent2 = temp;
94          while(temp2->right_child != nullptr){
95              parent2 = temp;
96              temp2 = temp2->right_child;
97          }
98          parent2->right_child = nullptr;
99          temp2->left_child = temp->left_child;
100         temp2->right_child = temp->right_child;
101         delete[] temp;
102         root = temp2;
103         return;
104     }
105     node *parent = root;
106     while(true){
107         if(temp->key == keyToDelete
108     )
109         {
110             break;
111         }else if(temp->key < keyToDelete
112     ){
113             parent = temp;
114             temp = temp->right_child;
115         }else if(temp->key > keyToDelete
116     ){
117             parent = temp;
118             temp = temp->left_child;
119         }
120         if(temp == nullptr){
121             return;
122         }
```

```
            }
        }
        if(temp->left_child == nullptr && temp->right_child == nullptr)
        {

            delete[] temp;
            return;
        }else if(temp->left_child == nullptr && temp->right_child != nullptr){
            if(parent->left_child ==  temp){
                parent->left_child = temp->right_child;
            }else{
                parent->right_child = temp->right_child;
            }
            delete[] temp;
            return;
        }else if(temp->left_child != nullptr && temp->right_child == nullptr){
            if(parent->left_child ==  temp){
                parent->left_child = temp->left_child;
            }else{
                parent->right_child = temp->left_child;
            }
            delete[] temp;
            return;
        }else{
            node *temp2 = temp->left_child;
            node *parent2 = temp;
            while(temp2->right_child != nullptr){
                parent2 = temp;
                temp2 = temp2->right_child;
            }
```

```
        parent2->right_child = nullptr;
        if(parent->left_child ==  temp){
            parent->left_child = temp2;
        }else{
            parent->right_child = temp2;
        }
        temp2->left_child = temp->left_child;
        temp2->right_child = temp->right_child;
        delete[] temp;
        return;
    }

}
```

6. int min (*node *tempo*):

This returns the minimum valued key in the tree.

7. int max (*node *tempo*):

This returns the maximum valued key in the tree.

```
int BST_linkedlist :: min(node *tempo){//function to return the minimum key in the tree
    node *temporary = tempo;
    while (temporary->left_child != NULL){
        temporary = temporary->left_child;

    }
    return temporary->key;

}

int BST_linkedlist :: max(node *tempo){ //function to calculate the maximum key in the tree and return it
    node *temporary = tempo;
    while (temporary->right_child != NULL){
        temporary = temporary->right_child;

    }
    return temporary->key;

}
```

8. void showdata(*node *n*):

This shows the data of the desired node.

```
void BST_linkedlist ::showData(node *n){ //shows the data at the desired node
    if (n== NULL){
        cout<<"\n\nkey==NULL"<<"value==NULL"<<endl;
    }
    cout <<"Key = "<< n->key<<endl;
    cout <<"Data = "<<n->data <<endl;



}
```

OUTPUT:

```
PS D:\Labworks\LAB $\CE2020_Lab4_50_51> g++ main.cpp .\source\linkedlistBST.cpp .\source\ArrayBst.cpp -I include
PS D:\Labworks\LAB $\CE2020_Lab4_50_51> ./a.exe



Which implementation do you want to see?
 Press 1 for Array
 Press 2 for Linkedlist
```

```
  ----------------------Implementation of BST using linkedlist----------------------------

 The list is empty
 Root added

 Checking if the tree is empty
 The list is not empty
 The smallest key in the tree is 1
 The largest key in the tree is 52
 Key = 12
 Data = 222

  Node removed
 Key = 12
 Data = 222
 The key is in the tree
 The key is not in the tree
 PS D:\Labworks\LAB $\CE2020_Lab4_50_51>
```