**Data Structure and Algorithm(Lab)**

**Name: Zeeshan Ali (SU92-BSITM-F22-019)**

**Assignment # 03 (Lab # 11)**

**Question no 01:**

**Code:**

#include <iostream>

#include <queue>

using namespace std;

class BST {

int data;

BST \*left, \*right;

public:

BST();

BST(int);

BST\* Insert(BST\*, int);

void Inorder(BST\*);

void preorder(BST\*);

void postorder(BST\*);

void DFS\_InOrder(BST\*);

void DFS\_PreOrder(BST\*);

void DFS\_PostOrder(BST\*);

void BFS\_LevelOrder(BST\*);

};

BST ::BST(){

data=0;

left=NULL;

right=NULL;

}

BST ::BST(int value)

{

data = value;

left = right = NULL;

}

BST\* BST ::Insert(BST\* root, int value)

{

if (root==NULL) {

return new BST(value);

}

if (value > root->data) {

root->right = Insert(root->right, value);

}

else if (value < root->data){

root->left = Insert(root->left, value);

}

return root;

}

void BST::BFS\_LevelOrder(BST\* root) {

if (root == NULL) {

return;

}

queue<BST\*> q;

q.push(root);

while (q.empty()==NULL) {

BST\* current = q.front();

cout << current->data << " ";

if (current->left != NULL) {

q.push(current->left);

}

if (current->right != NULL) {

q.push(current->right);

}

q.pop();

}

}

int main()

{

BST b, \*root = NULL;

root = b.Insert(root, 8);

b.Insert(root, 3);

b.Insert(root, 10);

b.Insert(root, 1);

b.Insert(root, 6);

b.Insert(root, 14);

b.Insert(root, 4);

b.Insert(root, 7);

b.Insert(root, 13);

b.Insert(root, 15);

b.Insert(root, 2);

b.Insert(root, 5);

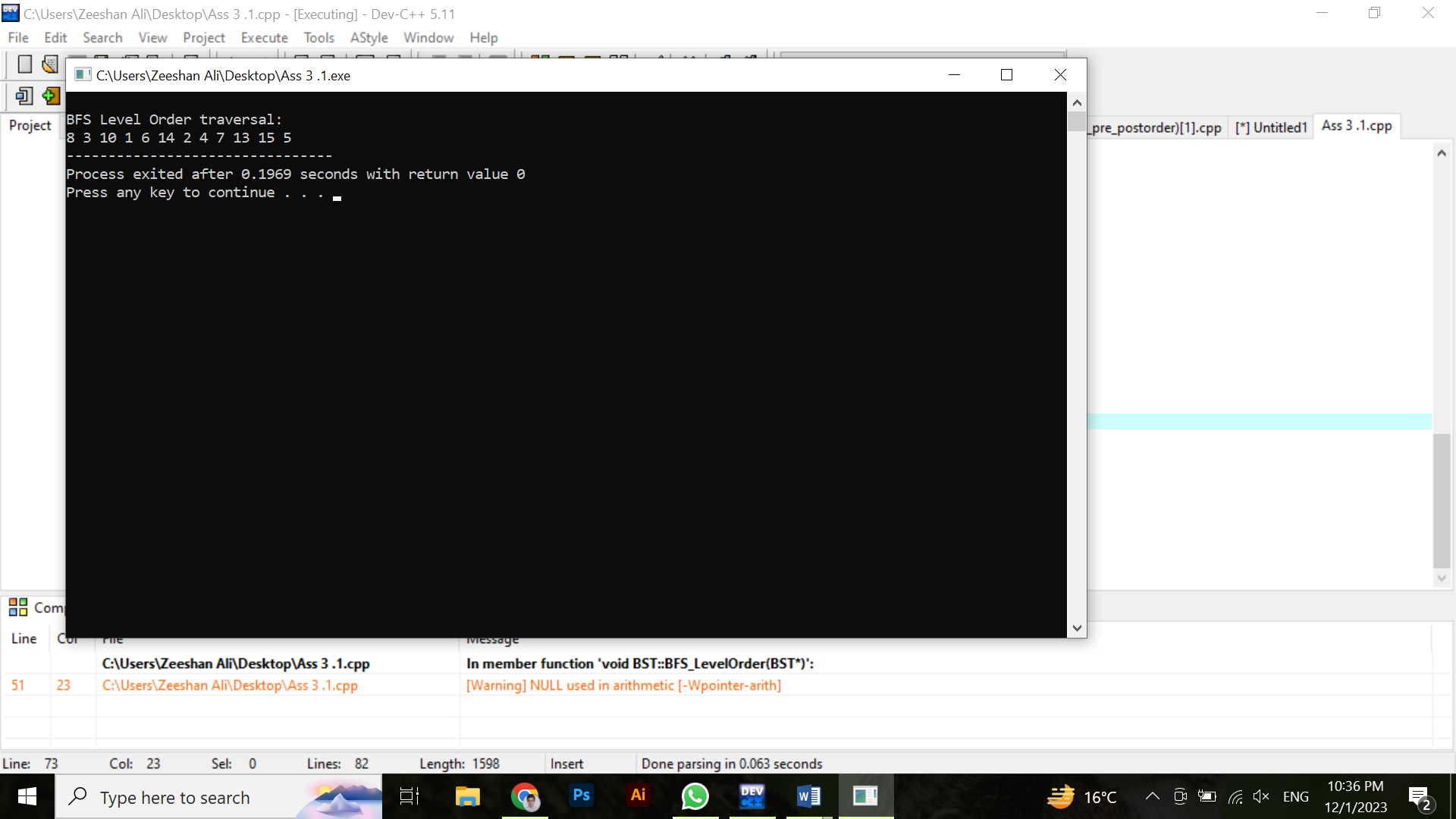
cout << "\nBFS Level Order traversal: " << endl;

b.BFS\_LevelOrder(root);

return 0;

}

**Output:**



**Question no 02:**

**Code:**

#include <iostream>a

#include <queue>

using namespace std;

class BST {

public:

int data;

BST \*left, \*right;

public:

BST();

BST(int);

void Inorder(BST\*);

void preorder(BST\*);

void postorder(BST\*);

void DFS\_InOrder(BST\*);

void DFS\_PreOrder(BST\*);

void DFS\_PostOrder(BST\*);

void BFS\_LevelOrder(BST\*);

BST\* inorderSuccessor(BST\*, BST\*);

};

BST ::BST(){

data=0;

left=NULL;

right=NULL;

}

BST ::BST(int value)

{

data = value;

left = right = NULL;

}

BST\* BST ::Insert(BST\* root, int value)

{

if (root==NULL) {

return new BST(value);

}

if (value > root->data) {

root->right = Insert(root->right, value);

}

else if (value < root->data){

root->left = Insert(root->left, value);

}

return root;

}

void BST ::Inorder(BST\* root)

{

if (root==NULL) {

return;

}

Inorder(root->left);

cout << root->data << endl;

Inorder(root->right);

}

BST\* BST ::inorderSuccessor(BST\* root, BST\* node) {

if (node->right != NULL) {

BST\* successor = node->right;

while (successor->left != NULL) {

successor = successor->left;

}

return successor;

}

BST\* successor = NULL;

while (root != NULL) {

if (node->data < root->data) {

successor = root;

root = root->left;

} else if (node->data > root->data) {

root = root->right;

} else {

break;

}

}

return successor;

}

int main()

{

BST b, \*root = NULL;

root = b.Insert(root, 8);

b.Insert(root, 3);

b.Insert(root, 10);

b.Insert(root, 1);

b.Insert(root, 6);

b.Insert(root, 14);

b.Insert(root, 4);

b.Insert(root, 7);

b.Insert(root, 13);

b.Insert(root, 15);

b.Insert(root, 2);

b.Insert(root, 5);

// Assuming you have a node for which you want to find the inorder successor

int nodeValue = 6;

BST\* nodeToFind = root;

while (nodeToFind != NULL && nodeToFind->data != nodeValue) {

if (nodeValue < nodeToFind->data) {

nodeToFind = nodeToFind->left;

} else {

nodeToFind = nodeToFind->right;

}

}

if (nodeToFind != NULL) {

BST\* successor = b.inorderSuccessor(root, nodeToFind);

if (successor != NULL) {

cout << "Inorder successor of " << nodeValue << " is: " << successor->data << std::endl;

} else {

cout << "No inorder successor found for " << nodeValue << std::endl;

}

} else {

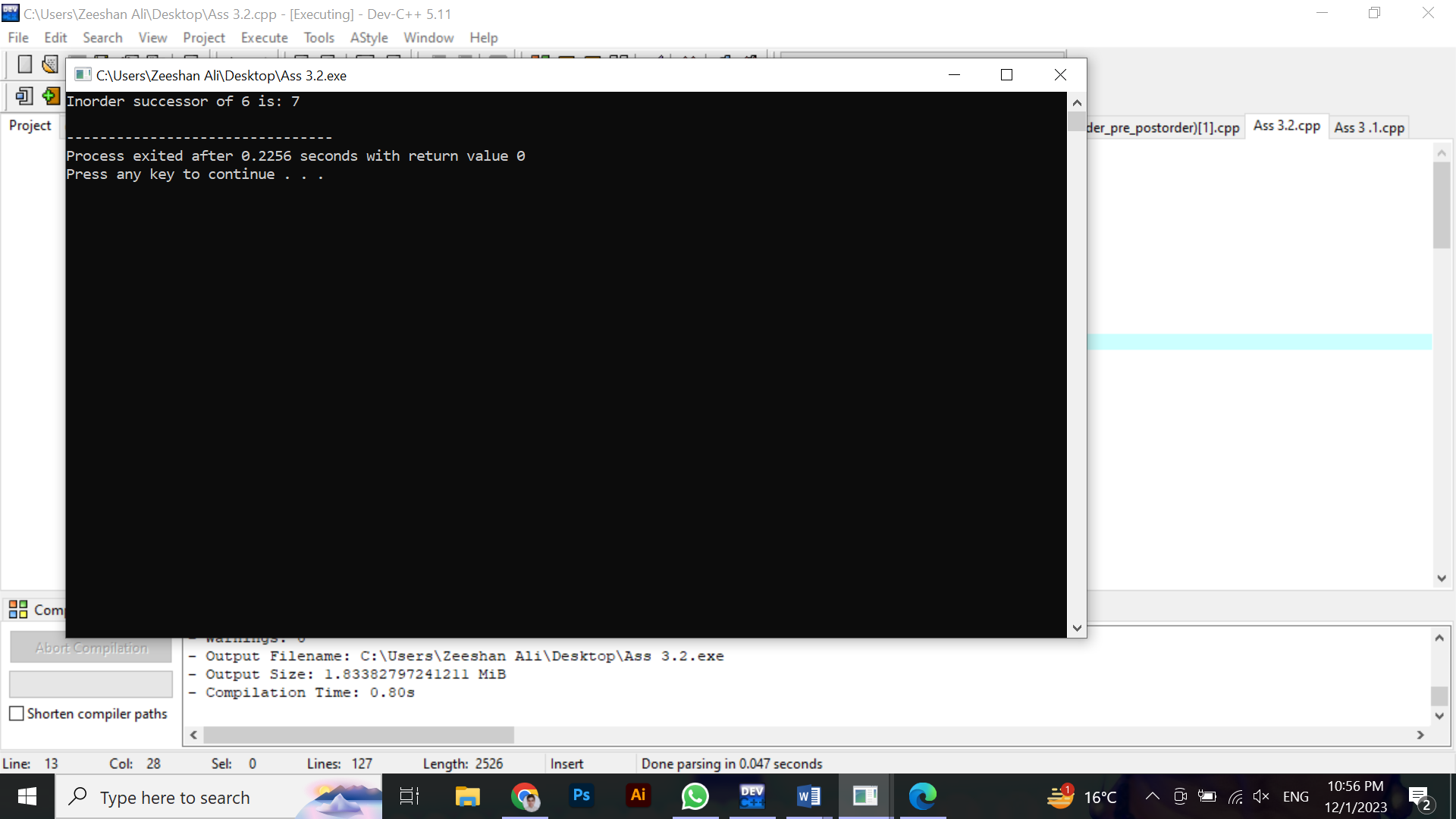
cout << "Node with value " << nodeValue << " not found in the BST." << std::endl;

}

return 0;

}

**Output:**



**Question no 03:**

**Code:**

#include <iostream>

#include <queue>

using namespace std;

class BST {

public:

int data;

BST \*left, \*right;

public:

BST();

BST(int);

BST\* Insert(BST\*, int);

void Inorder(BST\*);

void preorder(BST\*);

void postorder(BST\*);

void DFS\_InOrder(BST\*);

void DFS\_PreOrder(BST\*);

void DFS\_PostOrder(BST\*);

void BFS\_LevelOrder(BST\*);

BST\* findLevelOrderSuccessor(BST\*, int);

};

BST::BST(){

data=0;

left=NULL;

right=NULL;

}

BST::BST(int value){

data = value;

left = right = NULL;

}

BST\* BST::Insert(BST\* root, int value){

if (root==NULL) {

return new BST(value);

}

if (value > root->data) {

root->right = Insert(root->right, value);

}

else if (value < root->data){

root->left = Insert(root->left, value);

}

return root;

}

void BST::BFS\_LevelOrder(BST\* root) {

if (root == NULL) {

return;

}

queue<BST\*> q;

q.push(root);

while (!q.empty()) {

BST\* current = q.front();

cout << current->data << " ";

if (current->left != NULL) {

q.push(current->left);

}

if (current->right != NULL) {

q.push(current->right);

}

q.pop();

}

}

BST\* BST::findLevelOrderSuccessor(BST\* root, int value) {

if (root == NULL) {

return NULL;

}

queue<BST\*> q;

q.push(root);

while (!q.empty()) {

BST\* current = q.front();

q.pop();

if (current->data == value) {

if (!q.empty()) {

return q.front();

} else {

return NULL;

}

}

if (current->left != NULL) {

q.push(current->left);

}

if (current->right != NULL) {

q.push(current->right);

}

}

return NULL;

}

int main()

{

BST b, \*root = NULL;

root = b.Insert(root, 8);

b.Insert(root, 3);

b.Insert(root, 10);

b.Insert(root, 1);

b.Insert(root, 6);

b.Insert(root, 14);

b.Insert(root, 4);

b.Insert(root, 7);

b.Insert(root, 13);

b.Insert(root, 15);

b.Insert(root, 2);

b.Insert(root, 5);

cout << "\nBFS Level Order traversal: " << endl;

b.BFS\_LevelOrder(root);

int nodeValue = 6;

BST\* successor = b.findLevelOrderSuccessor(root, nodeValue);

if (successor != NULL) {

cout << "\nLevel Order successor of " << nodeValue << " is: " << successor->data << endl;

} else {

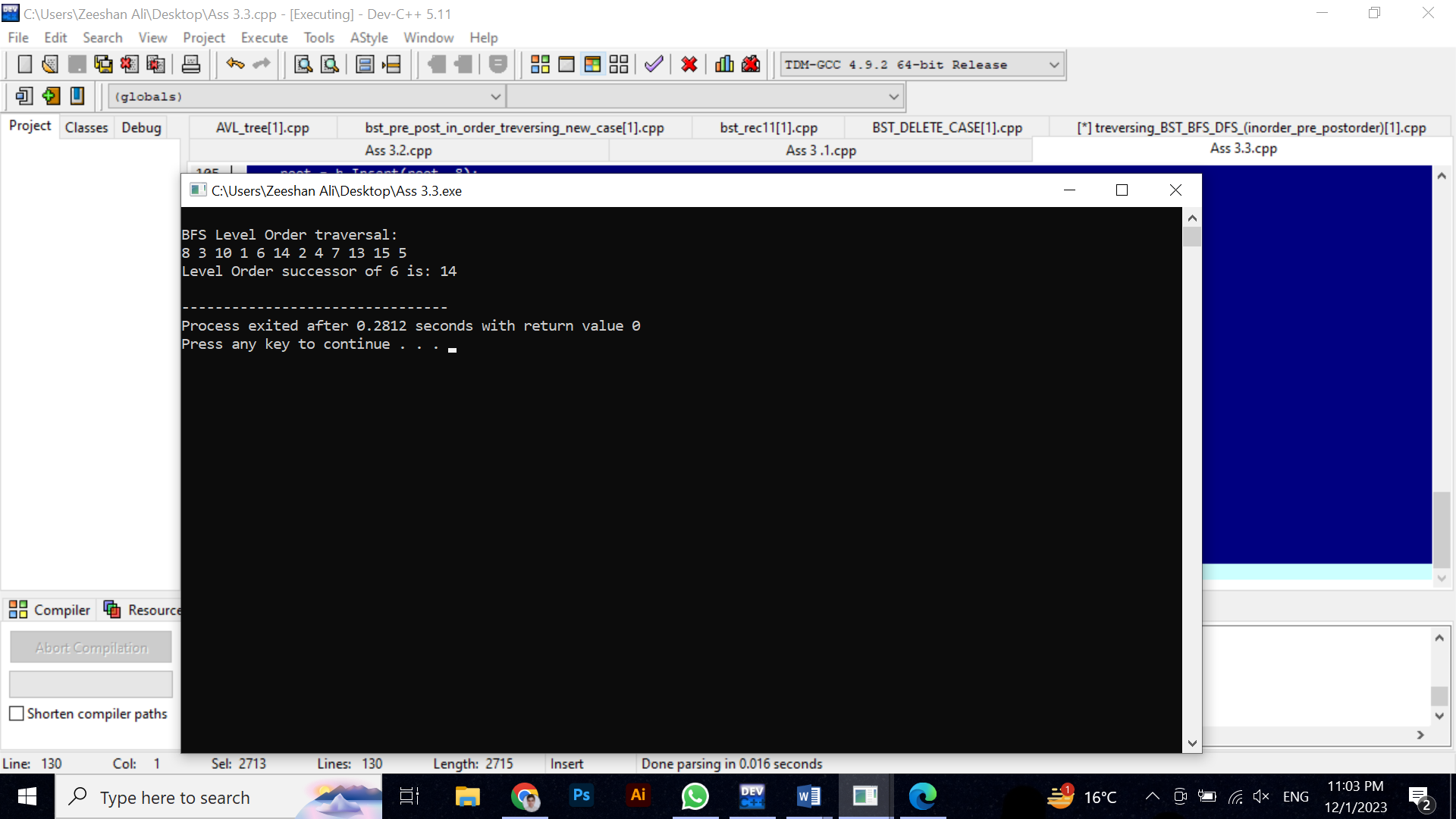
cout << "\nNo Level Order successor found for " << nodeValue << endl;

}

return 0;

}

**Output:**



**Question no 04:**

**Code:**

#include <iostream>

using namespace std;

class BST {

int data;

BST \*left, \*right;

public:

BST();

BST(int);

BST\* Insert(BST\*, int);

bool searchRecursive(BST\*, int);

bool searchIterative(BST\*, int);

};

BST::BST(){

data=0;

left=NULL;

right=NULL;

}

BST::BST(int value){

data = value;

left = right = NULL;

}

BST\* BST::Insert(BST\* root, int value){

if (root==NULL) {

return new BST(value);

}

if (value > root->data) {

root->right = Insert(root->right, value);

}

else if (value < root->data){

root->left = Insert(root->left, value);

}

return root;

}

bool BST::searchRecursive(BST\* root, int key) {

if (root == NULL) {

return false;

}

if (root->data == key) {

return true;

} else if (key < root->data) {

return searchRecursive(root->left, key);

} else {

return searchRecursive(root->right, key);

}

}

bool BST::searchIterative(BST\* root, int key) {

while (root != NULL) {

if (root->data == key) {

return true;

} else if (key < root->data) {

root = root->left;

} else {

root = root->right;

}

}

return false;

}

int main() {

BST b, \*root = NULL;

root = b.Insert(root, 8);

b.Insert(root, 3);

b.Insert(root, 10);

b.Insert(root, 1);

b.Insert(root, 6);

b.Insert(root, 14);

b.Insert(root, 4);

b.Insert(root, 7);

b.Insert(root, 13);

b.Insert(root, 15);

b.Insert(root, 2);

b.Insert(root, 5);

int keyToSearch = 6;

if (b.searchRecursive(root, keyToSearch)) {

cout << "Key " << keyToSearch << " found using recursive search." << endl;

} else {

cout << "Key " << keyToSearch << " not found using recursive search." << endl;

}

if (b.searchIterative(root, keyToSearch)) {

cout << "Key " << keyToSearch << " found using iterative search." << endl;

} else {

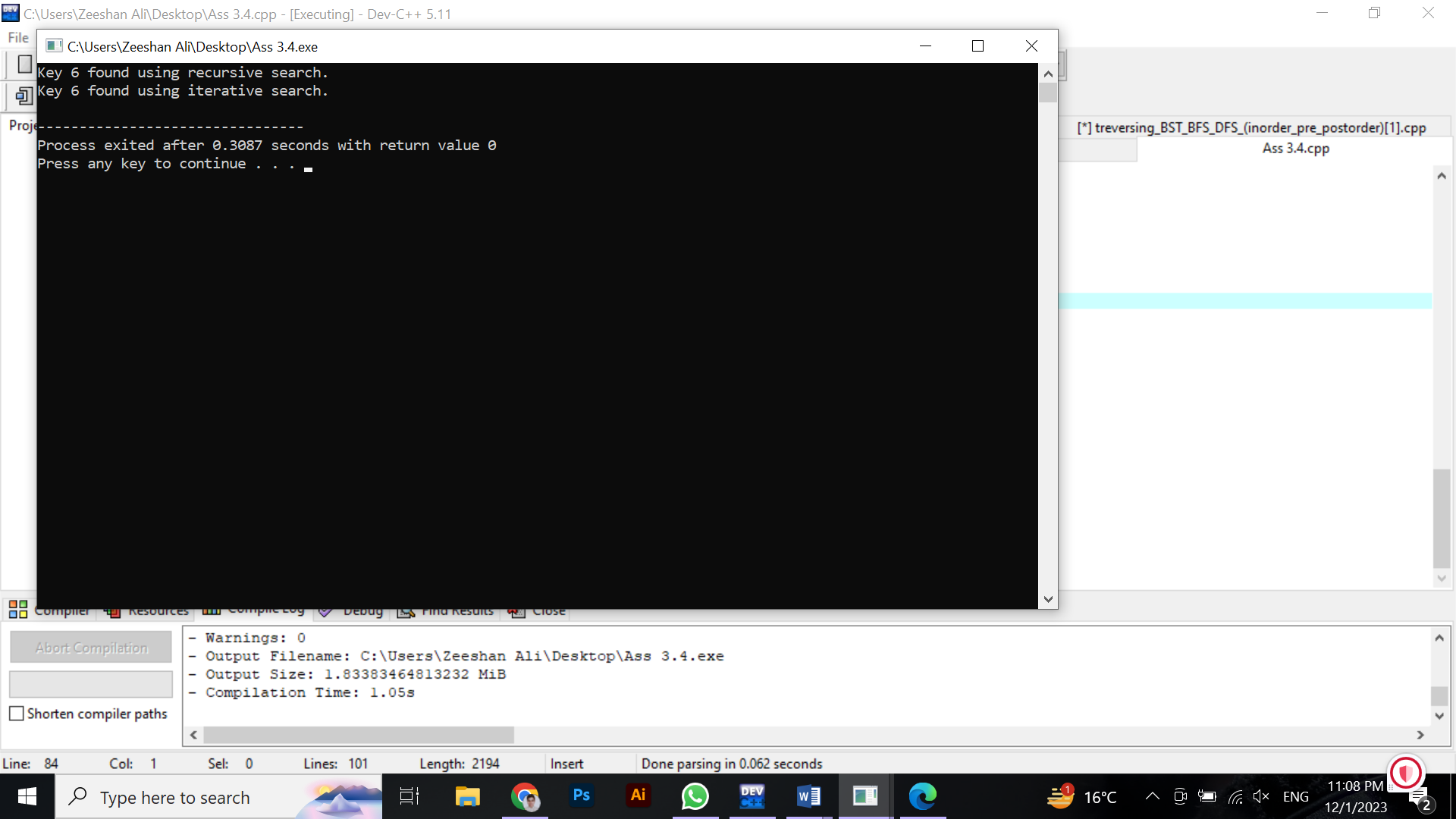
cout << "Key " << keyToSearch << " not found using iterative search." << endl;

}

return 0;

}

**Output:**



**Question no 05:**

**Code:**

#include <iostream>

#include <queue>

using namespace std;

class BST {

int data;

BST \*left, \*right;

public:

BST();

BST(int);

BST\* Insert(BST\*, int);

void Inorder(BST\*);

int findKthSmallest(BST\*, int, int&);

};

BST ::BST(){

data=0;

left=NULL;

right=NULL;

}

BST ::BST(int value)

{

data = value;

left = right = NULL;

}

BST\* BST ::Insert(BST\* root, int value)

{

if (root==NULL) {

return new BST(value);

}

if (value > root->data) {

root->right = Insert(root->right, value);

}

else if (value < root->data){

root->left = Insert(root->left, value);

}

return root;

}

void BST ::Inorder(BST\* root)

{

if (root==NULL) {

return;

}

Inorder(root->left);

cout << root->data << " ";

Inorder(root->right);

}

int BST::findKthSmallest(BST\* root, int k, int& count) {

if (root == NULL) {

return -1;

}

int leftResult = findKthSmallest(root->left, k, count);

if (leftResult != -1) {

return leftResult;

}

count++;

if (count == k) {

return root->data;

}

return findKthSmallest(root->right, k, count);

}

int main()

{

BST b, \*root = NULL;

root = b.Insert(root, 8);

b.Insert(root, 3);

b.Insert(root, 10);

b.Insert(root, 1);

b.Insert(root, 6);

b.Insert(root, 14);

b.Insert(root, 4);

b.Insert(root, 7);

b.Insert(root, 13);

b.Insert(root, 15);

b.Insert(root, 2);

b.Insert(root, 5);

int k = 3;

int count = 0;

int kthSmallest = b.findKthSmallest(root, k, count);

if (kthSmallest != -1) {

cout << "The " << k << "th smallest element is: " << kthSmallest << endl;

} else {

cout << "Invalid value of K." << endl;

}

return 0;

}

**Output:**

