**Muhammad Abdullah**

**SE(3A) | 19F-0916**

DS Course Assignment

Assignment # 5

**Question # 1:**

**PROGRAM**

#include <iostream>

using namespace std;

class Graph

{

public:

bool\*\* Matrix;

int Visited;

Graph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new bool\*[Visited];

for (int i = 1; i <= Visited; i++)

{

Matrix[i] = new bool[Visited];

for (int j = 1; j <= Visited; j++)

Matrix[i][j] = false;

}

}

void Add\_Edge(int i, int j) // Adding Edge

{

Matrix[i][j] = true;

Matrix[j][i] = true;

}

void Remove\_Edge(int i, int j) //Removing Edge (if needed)

{

Matrix[i][j] = false;

Matrix[j][i] = false;

}

void Adjencency\_Matrix() // Displaying Adjececy Matrix

{

cout << endl;

for (int i = 1; i <= Visited; i++)

{

for (int j = 1; j <= Visited; j++)

cout <<" " << Matrix[i][j] << " ";

cout << endl << endl;;

}

}

};

int main() {

Graph Adj\_Matrix(8);

Adj\_Matrix.Add\_Edge(1, 2);

Adj\_Matrix.Add\_Edge(2, 3);

Adj\_Matrix.Add\_Edge(4, 5);

Adj\_Matrix.Add\_Edge(1, 5);

Adj\_Matrix.Add\_Edge(6, 1);

Adj\_Matrix.Add\_Edge(7, 4);

Adj\_Matrix.Add\_Edge(3, 8);

Adj\_Matrix.Adjencency\_Matrix();

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, screenshot, electronics

Description automatically generated**

**Question # 2:**

**PROGRAM**

#include <iostream>

using namespace std;

struct Node

{

int Data;

Node \*Next\_Node;

};

struct Adjecency\_List

{

Node \*Head;

};

class Graph

{

public:

Adjecency\_List \*Matrix;

int Visited;

Graph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new Adjecency\_List[Visited];

for (int i = 1; i <= Visited; i++)

{

Matrix[i].Head = NULL;

}

}

Node \*List\_Node(int Data)

{

Node \*New\_Node = new Node;

New\_Node->Data = Data;

New\_Node->Next\_Node = NULL;

return New\_Node;

}

void Add\_Edge(int Coming, int Data) // Adding Edge

{

Node \*New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Coming].Head;

Matrix[Coming].Head = New\_Node;

New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Data].Head;

Matrix[Data].Head = New\_Node;

}

void Adjencency\_List() // Displaying Adjecency List

{

cout << endl;

for (int i = 1; i <= Visited; i++)

{

Node \*Traverse = Matrix[i].Head;

cout << "Vertex " << i <<" : ";

while (Traverse != NULL)

{

cout << " -> " << Traverse->Data;

Traverse = Traverse->Next\_Node;

}

cout << endl;

}

}

};

int main() {

Graph Adj\_Matrix(8);

Adj\_Matrix.Add\_Edge(1, 2);

Adj\_Matrix.Add\_Edge(2, 3);

Adj\_Matrix.Add\_Edge(4, 5);

Adj\_Matrix.Add\_Edge(1, 5);

Adj\_Matrix.Add\_Edge(6, 1);

Adj\_Matrix.Add\_Edge(7, 4);

Adj\_Matrix.Add\_Edge(3, 8);

Adj\_Matrix.Adjencency\_List();

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, screenshot, electronics

Description automatically generated**

**Question # 3:**

**PROGRAM**

#include <iostream>

using namespace std;

struct Node

{

int Data;

Node \*Next\_Node;

};

struct Adjecency\_List

{

Node \*Head;

};

class Graph

{

public:

Adjecency\_List \*Matrix;

int Visited;

Graph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new Adjecency\_List[Visited];

for (int i = 1; i <= Visited; i++)

{

Matrix[i].Head = NULL;

}

}

Node \*List\_Node(int Data)

{

Node \*New\_Node = new Node;

New\_Node->Data = Data;

New\_Node->Next\_Node = NULL;

return New\_Node;

}

void Add\_Edge(int Coming, int Data) // Adding Edge

{

Node \*New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Coming].Head;

Matrix[Coming].Head = New\_Node;

New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Data].Head;

Matrix[Data].Head = New\_Node;

}

void Adjencency\_List() // Displaying Adjecency List

{

cout << "\t\t\t\tA=1 , B=2 , C=3 , D=4 , E=5 , F=6\t(List To Check Values)" << endl;

for (int i = 1; i <= Visited; i++)

{

Node \*Traverse = Matrix[i].Head;

cout << " Vertex " << i <<" : ";

while (Traverse != NULL)

{

cout << " -> " << Traverse->Data;

Traverse = Traverse->Next\_Node;

}

cout << endl;

}

}

};

class GGraph

{

public:

bool\*\* Matrix;

int Visited;

GGraph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new bool\*[Visited];

for (int i = 1; i <= Visited; i++)

{

Matrix[i] = new bool[Visited];

for (int j = 1; j <= Visited; j++)

Matrix[i][j] = false;

}

}

void Add\_Edge(int i, int j) // Adding Edge

{

Matrix[i][j] = true;

Matrix[j][i] = true;

}

void Remove\_Edge(int i, int j) //Removing Edge (if needed)

{

Matrix[i][j] = false;

Matrix[j][i] = false;

}

void Adjencency\_Matrix() // Displaying Adjececy Matrix

{

char a[] = { 'A','B','C','D','E','F' };

int aa = 0;

cout << endl << " A B C D E F" ;

cout << endl << " | | | | | |" << endl << endl;

for (int i = 1; i <= Visited; i++)

{

cout << a[aa++]<<" --";

for (int j = 1; j <= Visited; j++)

cout << " " << Matrix[i][j] << " ";

cout << endl << endl;;

}

}

};

int main() {

Graph Adj\_Matrix(6);

GGraph ADJ(6);

/\*Adj\_Matrix.Add\_Edge(1, 2);

Adj\_Matrix.Add\_Edge(1, 3);

Adj\_Matrix.Add\_Edge(1, 6);

Adj\_Matrix.Add\_Edge(2, 3);

Adj\_Matrix.Add\_Edge(2, 4);

Adj\_Matrix.Add\_Edge(3, 6);

Adj\_Matrix.Add\_Edge(3, 4);

Adj\_Matrix.Add\_Edge(4, 5);

Adj\_Matrix.Add\_Edge(5, 6);

//Adj\_Matrix.Add\_Edge(6, 1);

//Adj\_Matrix.Add\_Edge(6, 3);

//Adj\_Matrix.Add\_Edge(6, 5);

Adj\_Matrix.Adjencency\_List();\*/

ADJ.Add\_Edge(1, 2);

ADJ.Add\_Edge(1, 3);

ADJ.Add\_Edge(1, 6);

ADJ.Add\_Edge(2, 3);

ADJ.Add\_Edge(2, 4);

ADJ.Add\_Edge(3, 6);

ADJ.Add\_Edge(3, 4);

ADJ.Add\_Edge(4, 5);

ADJ.Add\_Edge(5, 6);

ADJ.Adjencency\_Matrix();

cout << endl << endl;

system("pause");

}

**A picture containing text, screenshot, monitor, indoor

Description automatically generated**

**Question # 4:**

**PROGRAM**

#include <iostream>

using namespace std;

struct Node

{

int Data;

Node \*Next\_Node;

};

struct Adjecency\_List

{

Node \*Head;

};

class Graph

{

public:

Adjecency\_List \*Matrix;

int Visited;

Graph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new Adjecency\_List[Visited];

for (int i = 1; i <= Visited; i++)

{

Matrix[i].Head = NULL;

}

}

Node \*List\_Node(int Data)

{

Node \*New\_Node = new Node;

New\_Node->Data = Data;

New\_Node->Next\_Node = NULL;

return New\_Node;

}

void Add\_Edge(int Coming, int Data) // Adding Edge

{

Node \*New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Coming].Head;

Matrix[Coming].Head = New\_Node;

New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Data].Head;

Matrix[Data].Head = New\_Node;

}

void Adjencency\_List() // Displaying Adjecency List

{

cout << "\t\t\t\t A=1 , B=2 , C=3 , D=4 , E=5 , F=6\t(List To Check Values)" << endl;

for (int i = 1; i <= Visited; i++)

{

Node \*Traverse = Matrix[i].Head;

cout << " Vertex " << i << " : ";

while (Traverse != NULL)

{

cout << " -> " << Traverse->Data;

Traverse = Traverse->Next\_Node;

}

cout << endl;

}

}

};

int main() {

Graph Adj\_Matrix(6);

Adj\_Matrix.Add\_Edge(1, 2);

Adj\_Matrix.Add\_Edge(1, 3);

Adj\_Matrix.Add\_Edge(1, 6);

Adj\_Matrix.Add\_Edge(2, 3);

Adj\_Matrix.Add\_Edge(2, 4);

Adj\_Matrix.Add\_Edge(3, 6);

Adj\_Matrix.Add\_Edge(3, 4);

Adj\_Matrix.Add\_Edge(4, 5);

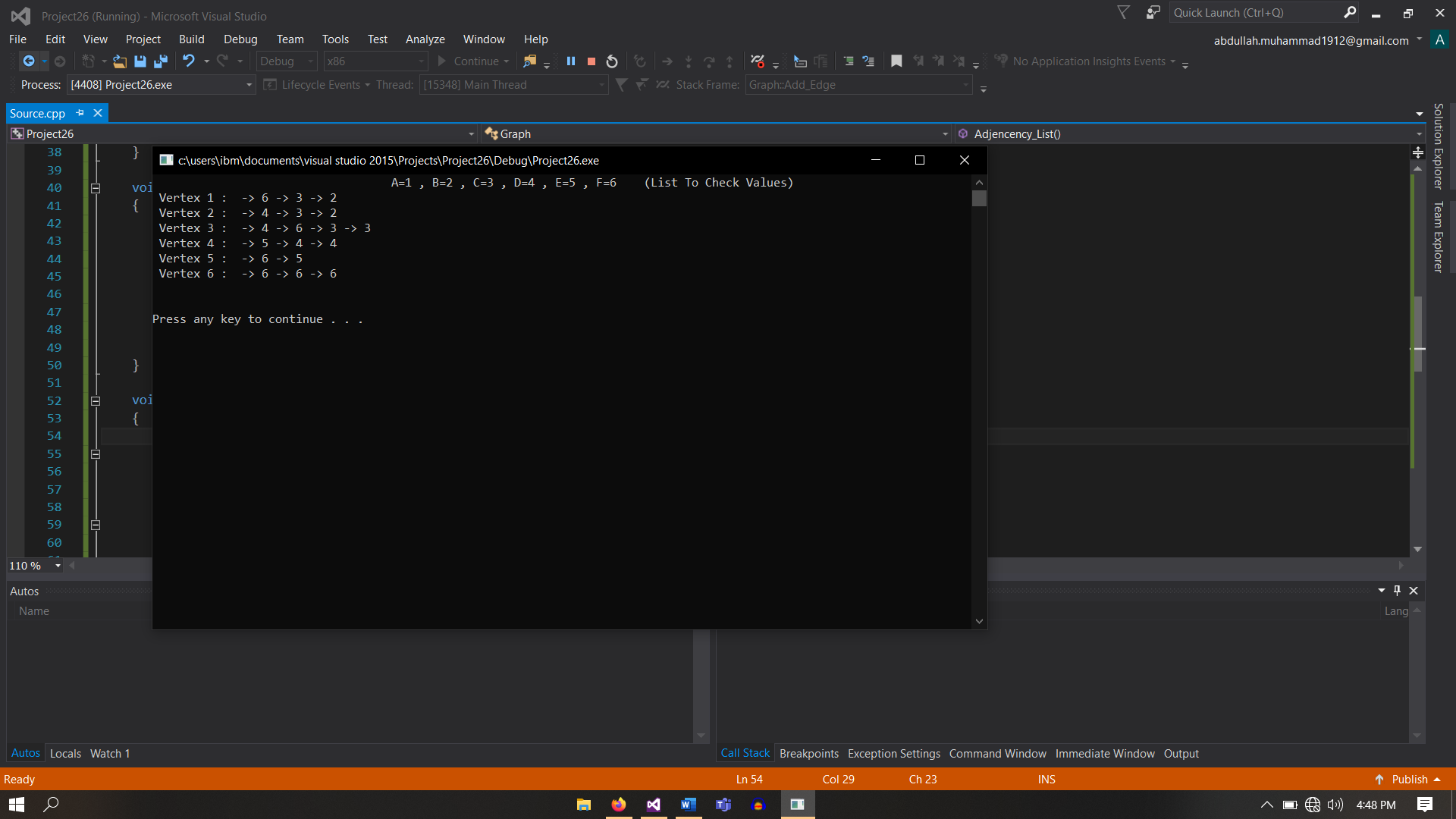
Adj\_Matrix.Add\_Edge(5, 6);

Adj\_Matrix.Adjencency\_List();

cout << endl << endl;

system("pause");

}

****

**Question # 5:**

**PROGRAM**

#include <iostream>

using namespace std;

struct Node

{

int Data;

Node \*Next\_Node;

};

struct Adjecency\_List

{

Node \*Head;

};

class Graph

{

public:

Adjecency\_List \*Matrix;

int Visited;

Graph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new Adjecency\_List[Visited];

for (int i = 0; i < Visited; i++)

{

Matrix[i].Head = NULL;

}

}

Node \*List\_Node(int Data)

{

Node \*New\_Node = new Node;

New\_Node->Data = Data;

New\_Node->Next\_Node = NULL;

return New\_Node;

}

void Add\_Edge(int Coming, int Data) // Adding Edge

{

Node \*New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Coming].Head;

Matrix[Coming].Head = New\_Node;

New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Data].Head;

Matrix[Data].Head = New\_Node;

}

void Adjencency\_List() // Displaying Adjecency List

{

cout << "\t\t\t\tA=1 , B=2 , C=3 , D=4 , E=5 , F=6\t(List To Check Values)" << endl;

for (int i = 0; i < Visited; i++)

{

Node \*Traverse = Matrix[i].Head;

cout << " Vertex " << i << " : ";

while (Traverse != NULL)

{

cout << " -> " << Traverse->Data;

Traverse = Traverse->Next\_Node;

}

cout << endl;

}

}

};

class GGraph

{

public:

bool\*\* Matrix;

int Visited;

GGraph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new bool\*[Visited];

for (int i = 0; i < Visited; i++)

{

Matrix[i] = new bool[Visited];

for (int j = 0; j < Visited; j++)

Matrix[i][j] = false;

}

}

void Add\_Edge(int i, int j) // Adding Edge

{

Matrix[i][j] = true;

Matrix[j][i] = true;

}

void Remove\_Edge(int i, int j) //Removing Edge (if needed)

{

Matrix[i][j] = false;

Matrix[j][i] = false;

}

void Adjencency\_Matrix() // Displaying Adjececy Matrix

{

int a[] = { 0,1,2,3,4,5,6,7,8,9 };

int count[10] = { 0 };

int aa = 0;

cout << endl << " 0 1 2 3 4 5 6 7 8 9";

cout << endl << " | | | | | | | | | |" << endl << endl;

for (int i = 0; i < Visited; i++)

{

cout << " " << a[aa++] << " --";

for (int j = 0; j < Visited; j++)

{

cout << " " << Matrix[i][j] << " ";

if (Matrix[i][j] == 1)

{

count[i] = count[i] + 1;

}

}

cout << endl << endl;;

}

cout << " Total Numbers Edges Each Node Have : " << endl<<endl;

for (int i = 0; i < 10; i++)

{

cout << " " << i << ": " << count[i] << endl;

}

}

};

int main() {

GGraph ADJ(10);

ADJ.Add\_Edge(0, 1);

ADJ.Add\_Edge(0, 6);

ADJ.Add\_Edge(0, 8);

ADJ.Add\_Edge(1, 4);

ADJ.Add\_Edge(1, 6);

ADJ.Add\_Edge(1, 9);

ADJ.Add\_Edge(2, 4);

ADJ.Add\_Edge(2, 6);

ADJ.Add\_Edge(3, 4);

ADJ.Add\_Edge(3, 5);

ADJ.Add\_Edge(3, 8);

ADJ.Add\_Edge(4, 5);

ADJ.Add\_Edge(4, 9);

ADJ.Add\_Edge(7, 8);

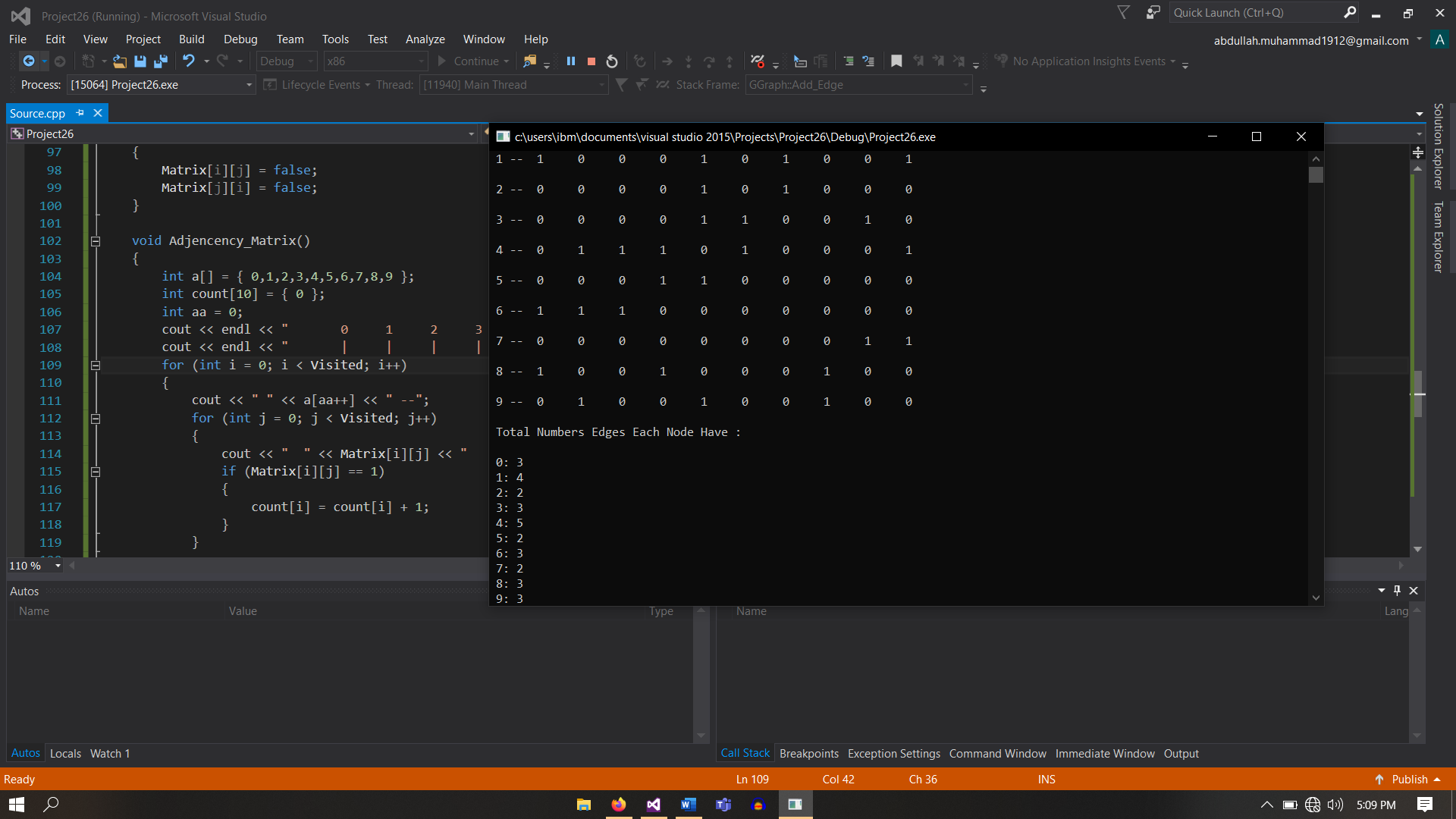
ADJ.Add\_Edge(7, 9);

ADJ.Adjencency\_Matrix();

cout << endl << endl;

system("pause");

}

****

**Question # 6:**

**PROGRAM**

#include <iostream>

using namespace std;

struct Node

{

int Data;

Node \*Next\_Node;

};

struct Adjecency\_List

{

Node \*Head;

};

class Graph

{

public:

Adjecency\_List \*Matrix;

int Visited;

Graph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new Adjecency\_List[Visited];

for (int i = 0; i < Visited; i++)

{

Matrix[i].Head = NULL;

}

}

Node \*List\_Node(int Data)

{

Node \*New\_Node = new Node;

New\_Node->Data = Data;

New\_Node->Next\_Node = NULL;

return New\_Node;

}

void Add\_Edge(int Coming, int Data) // Adding Edge

{

Node \*New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Coming].Head;

Matrix[Coming].Head = New\_Node;

New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Data].Head;

Matrix[Data].Head = New\_Node;

}

void Adjencency\_List() // Displaying Adjecency List

{

cout << "\t\t\t\tA=1 , B=2 , C=3 , D=4 , E=5 , F=6\t(List To Check Values)" << endl;

for (int i = 0; i < Visited; i++)

{

Node \*Traverse = Matrix[i].Head;

cout << " Vertex " << i << " : ";

while (Traverse != NULL)

{

cout << " -> " << Traverse->Data;

Traverse = Traverse->Next\_Node;

}

cout << endl;

}

}

};

class GGraph

{

public:

bool\*\* Matrix;

int Visited;

GGraph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new bool\*[Visited];

for (int i = 0; i < Visited; i++)

{

Matrix[i] = new bool[Visited];

for (int j = 0; j < Visited; j++)

Matrix[i][j] = false;

}

}

void Add\_Edge(int i, int j) // Adding Edge

{

Matrix[i][j] = true;

Matrix[j][i] = true;

}

void Remove\_Edge(int i, int j) //Removing Edge (if needed)

{

Matrix[i][j] = false;

Matrix[j][i] = false;

}

void Adjencency\_Matrix() // Displaying Adjececy Matrix

{

int a[] = { 0,1,2,3,4 };

int count[5] = { 0 };

int aa = 0;

cout << endl << " 0 1 2 3 4";

cout << endl << " | | | | |" << endl << endl;

for (int i = 0; i < Visited; i++)

{

cout << " " << a[aa++] << " --";

for (int j = 0; j < Visited; j++)

{

cout << " " << Matrix[i][j] << " ";

if (i==j && (Matrix[i][j] == 1 || Matrix[i][j] == 1))

{

count[i] = count[i] + 1;

}

}

cout << endl << endl;;

}

cout << " Total Numbers Edges That Do Not Share Same Edge : " << endl<<endl;

for (int i = 0; i < 5; i++)

{

cout << " " << i << ": " << count[i] << endl;

}

}

};

int main() {

GGraph ADJ(5);

ADJ.Add\_Edge(0, 2);

ADJ.Add\_Edge(1, 1);

ADJ.Add\_Edge(1, 3);

ADJ.Add\_Edge(2, 4);

ADJ.Add\_Edge(3, 3);

ADJ.Adjencency\_Matrix();

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, indoor, electronics

Description automatically generated**

**Question # 7:**

**PROGRAM**

#include <iostream>

using namespace std;

struct Node

{

int Data;

Node \*Next\_Node;

};

struct Adjecency\_List

{

Node \*Head;

};

class Graph

{

public:

Adjecency\_List \*Matrix;

int Visited;

Graph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new Adjecency\_List[Visited];

for (int i = 0; i < Visited; i++)

{

Matrix[i].Head = NULL;

}

}

Node \*List\_Node(int Data)

{

Node \*New\_Node = new Node;

New\_Node->Data = Data;

New\_Node->Next\_Node = NULL;

return New\_Node;

}

void Add\_Edge(int Coming, int Data) // Adding Edge

{

Node \*New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Coming].Head;

Matrix[Coming].Head = New\_Node;

New\_Node = List\_Node(Data);

New\_Node->Next\_Node = Matrix[Data].Head;

Matrix[Data].Head = New\_Node;

}

void Adjencency\_List() // Displaying Adjecency List

{

for (int i = 0; i < Visited; i++)

{

Node \*Traverse = Matrix[i].Head;

cout << " Vertex " << i << " : ";

Traverse = Traverse->Next\_Node;

while (Traverse != NULL)

{

cout << " -> " << Traverse->Data;

Traverse = Traverse->Next\_Node;

}

cout << endl;

}

}

};

class GGraph

{

public:

bool\*\* Matrix;

int Visited;

GGraph(int Visited) // Starting From Zero

{

this->Visited = Visited;

Matrix = new bool\*[Visited];

for (int i = 0; i < Visited; i++)

{

Matrix[i] = new bool[Visited];

for (int j = 0; j < Visited; j++)

Matrix[i][j] = false;

}

}

void Add\_Edge(int i, int j) // Adding Edge

{

Matrix[i][j] = true;

// Matrix[j][i] = true;

}

void Remove\_Edge(int i, int j) //Removing Edge (if needed)

{

Matrix[i][j] = false;

Matrix[j][i] = false;

}

void Transpose\_Multiplication()

{

int row = 4, col = 4;

int transpose[4][4] = { 0 };

for (int i = 0; i < row; ++i)

for (int j = 0; j < col; ++j) {

transpose[j][i] = Matrix[i][j];

}

cout << endl << " Transpose of Matrix :" << endl << endl;

for (int i = 0; i < col; ++i)

for (int j = 0; j < row; ++j) {

cout << " " << transpose[i][j] << " ";

if (j == row - 1)

cout << endl << endl;

}

cout << endl << " After Multiplication of Orignal with Transpose : " << endl << endl;

int Mul[4][4] = { 0 };

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

{

Mul[i][j] = Matrix[i][j] \* transpose[i][j];

}

}

for (int i = 0; i < col; ++i)

{

for (int j = 0; j < row; ++j)

{

cout << " " << Mul[i][j] << " ";

}

cout << endl << endl;

}

Graph A(4);

for (int i = 0; i < row; ++i)

{

for (int j = 0; j < col; ++j)

{

if (Mul[i][j] == 1 )

{

A.Add\_Edge(i, j);

}

}

}

cout << endl;

A.Adjencency\_List();

}

void Adjencency\_Matrix() // Displaying Adjececy Matrix

{

for (int i = 0; i < Visited; i++)

{

for (int j = 0; j < Visited; j++)

{

cout << " " << Matrix[i][j] << " ";

}

cout << endl << endl;

}

}

};

int main()

{

GGraph ADJ(4);

ADJ.Add\_Edge(0, 1);

ADJ.Add\_Edge(0, 2);

ADJ.Add\_Edge(1, 0);

ADJ.Add\_Edge(1, 1);

ADJ.Add\_Edge(1, 2);

ADJ.Add\_Edge(1, 3);

ADJ.Add\_Edge(2, 1);

ADJ.Add\_Edge(2, 3);

ADJ.Add\_Edge(3, 0);

ADJ.Add\_Edge(3, 3);

cout << " Original Matrix : " << endl << endl;

ADJ.Adjencency\_Matrix();

ADJ.Transpose\_Multiplication();

cout << endl << endl;

system("pause");

}

**A picture containing text, screenshot, monitor, indoor

Description automatically generated**

**A picture containing text, screenshot, monitor, indoor

Description automatically generated**

**Question # 8:**

**PROGRAM**

#include <iostream>

using namespace std; // Not Sure About Splittion, Tried but....

struct Node // Node Construction

{

int Data;

Node \*Left\_Node;

Node \*Right\_Node;

};

class Bst\_Tree // Class for Tree Formation

{

public:

Bst\_Tree()

{

Root = NULL;

}

Node \*Root;

Node \*Insertion(Node \*Current, int data) // Insertion of Nodes

{

if (Current == NULL)

{

Current = new Node;

Current->Data = data;

Current->Left\_Node = NULL;

Current->Right\_Node = NULL;

return Current;

}

else if (data < Current->Data) // If Data is lesser than Root

{

Current->Left\_Node = Insertion(Current->Left\_Node, data);

}

else if (data >= Current->Data) // If Data is greater than Root

{

Current->Right\_Node = Insertion(Current->Right\_Node, data);

}

return Current;

}

void In\_Order(Node \*C) // In order traversal

{

if (Root == NULL)

{

cout << endl << "Tree is Empty !" << endl;

return;

}

else

{

if (C == NULL)

return;

In\_Order(C->Left\_Node);

cout << " " << C->Data;

In\_Order(C->Right\_Node);

}

}

void Pre\_Order(Node \*C) // Pre Order traversal

{

if (Root == NULL)

{

cout << endl << "Tree is Empty !" << endl;

return;

}

else

{

Node \*temp = C;

if (temp == NULL)

return;

cout << " " << C->Data;

Pre\_Order(temp->Left\_Node);

Pre\_Order(temp->Right\_Node);

}

}

void Post\_Order(Node \*C) // Post Order traversal

{

if (Root == NULL)

{

cout << endl << "Tree is Empty !" << endl;

return;

}

else

{

Node \*temp = C;

if (temp == NULL)

return;

Post\_Order(temp->Left\_Node);

Post\_Order(temp->Right\_Node);

cout << " " << C->Data;

}

}

int BST\_OR\_NOT(Node\* root, Node\* Left = NULL, Node\* Right = NULL)

{

if (root == NULL)

return 1;

if (Left != NULL && root->Data <= Left->Data) // Checking Left Data

return 0;

if (Right != NULL && root->Data >= Right->Data) // Checking Right Data

return 0;

return BST\_OR\_NOT(root->Left\_Node, Left, root) && BST\_OR\_NOT(root->Right\_Node, root, Right); // Traversing in Tree

}

};

int main()

{

Bst\_Tree Tree;

int choice = 0, value = 0, i = 1;

while (i != 0)

{

system("cls");

cout << "=================================================" << endl;

cout << " Press 1 to Enter a Value in Tree !" << endl << endl;

cout << " Press 2 to Perform In-Order Traversal !" << endl;

cout << " Press 3 to Perform Pre-Order Traversal !" << endl;

cout << " Press 4 to Perform Post-Order Traversal !" << endl << endl;

cout << " Press 5 to Check That Tree is BST or NOT !" << endl;

cout << " Press 0 To Exit From The System !" << endl;

cout << "=================================================" << endl;

cout << " Enter Choice : ";

cin >> choice;

switch (choice)

{

case 1:

{

cout << endl << "Enter Value to Enter it in Tree : ";

cin >> value;

Tree.Root = Tree.Insertion(Tree.Root, value);

system("pause");

break;

}

case 2:

{

cout << endl << "In-Order Traversal : ";

Tree.In\_Order(Tree.Root);

cout << endl;

system("pause");

break;

}

case 3:

{

cout << endl<<"Pre-Order Traversal : ";

Tree.Pre\_Order(Tree.Root);

cout << endl;

system("pause");

break;

}

case 4:

{

cout << endl << "Post-Order Traversal : ";

Tree.Post\_Order(Tree.Root);

cout << endl;

system("pause");

break;

}

case 5:

{

cout << endl;

value = Tree.BST\_OR\_NOT(Tree.Root);

if (value == 1)

cout << endl << "Tree is Binary Search Tree !" << endl;

else

cout << endl << "Tree is Not Binary Search Tree !" << endl;

cout << endl;

system("pause");

break;

}

case 0:

{

cout << endl << "You have exited from the system !" << endl;

i = 0;

break;

}

default:

cout << endl << "Invalid Entry !" << endl;

system("pause");

break;

}

}

cout << endl << endl;

system("pause");

}