**DATA STRUCTURE**

**LAB 12**

**NAME**: **Faizan Tariq**

**Roll No:** 22F-3858

TASK 1:

#include <iostream>

using namespace std;

class Graph {

int\*\* adjMat;

int noOfVertices;

public:

Graph(int num) : noOfVertices(num) {

adjMat = new int\* [num];

for (int i = 0; i < noOfVertices; i++) {

adjMat[i] = new int[noOfVertices];

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

adjMat[i][j] = 0;

}

}

}

bool makeConnect(int a, int b) {

adjMat[a][b] = 1;

adjMat[b][a] = 1;

return true;

}

void disp() {

for (int i = 0; i < noOfVertices; i++) {

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

if (adjMat[i][j] == 1) {

cout << "This has " << i << " Connection with " << j << endl;

}

}

}

cout << endl;

}

~Graph() {

for (int i = 0; i < noOfVertices; i++) {

delete[] adjMat[i];

}

delete[] adjMat;

}

};

int main() {

Graph graph(10);

graph.makeConnect(0, 1);

graph.makeConnect(0, 6);

graph.makeConnect(0, 8);

graph.makeConnect(1, 4);

graph.makeConnect(1, 6);

graph.makeConnect(1, 9);

graph.makeConnect(2, 4);

graph.makeConnect(2, 6);

graph.makeConnect(3, 4);

graph.makeConnect(3, 5);

graph.makeConnect(3, 8);

graph.makeConnect(4, 5);

graph.makeConnect(4, 9);

graph.makeConnect(7, 8);

graph.makeConnect(7, 9);

graph.disp();

return 0;

}

OUTPUT:  


**TASK 2:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

class Graph {

int vertices;

Node\*\* adjList;

public:

Graph(int num) : vertices(num) {

adjList = new Node \* [num];

for (int i = 0; i < num; ++i) {

adjList[i] = NULL;

}

}

void addEdge(int from, int to) { // undirected graph

Node\* newNodeFrom = new Node;

newNodeFrom->data = to;

newNodeFrom->next = adjList[from];

adjList[from] = newNodeFrom;

Node\* newNodeTo = new Node;

newNodeTo->data = from;

newNodeTo->next = adjList[to];

adjList[to] = newNodeTo;

}

void displayList() {

cout << "\nAdjacency list:\n";

for (int i = 0; i < vertices; ++i) {

cout << "Node " << i << ": ";

Node\* cur = adjList[i];

while (cur != NULL) {

cout << cur->data << " ";

cur = cur->next;

}

cout << "\n";

}

}

~Graph() {

for (int i = 0; i < vertices; ++i) {

Node\* cur = adjList[i];

while (cur != NULL) {

Node\* temp = cur->next;

delete cur;

cur = temp;

}

}

delete[] adjList;

}

};

int main() {

Graph graph(5);

graph.addEdge(0, 2);

graph.addEdge(0, 1);

graph.addEdge(2, 3);

graph.addEdge(2, 3);

graph.addEdge(1, 4);

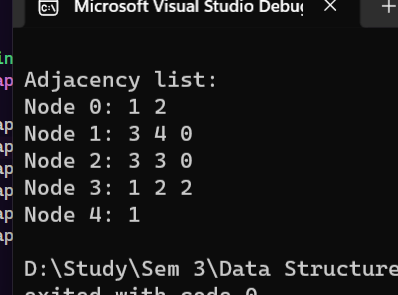
graph.addEdge(1, 3);

graph.displayList();

return 0;

}

**OUTPUT:**

****

**TASK 3:**

#include <iostream>

using namespace std;

class QueueArrayAdt {

int size;

int rear;

int front;

int numItems;

int\* array;

public:

QueueArrayAdt(int size) {

this->size = size;

front = -1;

rear = -1;

numItems = 0;

array = new int[size];

}

bool isEmpty() {

return numItems == 0;

}

bool isFull() {

return numItems == size - 1;

}

void enqueue(int data) {

if (isFull()) {

cout << "\nQueue is Full.\n";

return;

}

rear = (rear + 1) % size;

array[rear] = data;

numItems++;

}

void dequeue() {

if (isEmpty()) {

cout << "\nQueue is Empty.\n";

return;

}

front = (front + 1) % size;

numItems--;

}

int peek() {

if (isEmpty()) {

cout << "\nQueue is Empty.\n";

return -1; // Return a special value indicating an empty queue

}

return array[(front + 1) % size];

}

~QueueArrayAdt() {

front = -1;

rear = -1;

numItems = 0;

delete[] array;

}

};

class Graph {

int\*\* adjMat;

int noOfVertices;

public:

Graph(int num) : noOfVertices(num) {

adjMat = new int\* [num];

for (int i = 0; i < noOfVertices; i++) {

adjMat[i] = new int[noOfVertices];

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

adjMat[i][j] = 0;

}

}

}

bool makeConnect(int a, int b) {

adjMat[a][b] = 1;

adjMat[b][a] = 1;

return true;

}

void bfsTraversal(int initialVertex) {

bool\* visited = new bool[noOfVertices] {false};

QueueArrayAdt queue(noOfVertices);

visited[initialVertex] = true;

queue.enqueue(initialVertex);

while (!queue.isEmpty()) {

int currentVertex = queue.peek();

cout << currentVertex << " ";

queue.dequeue();

for (int i = 0; i < noOfVertices; ++i) {

if (adjMat[currentVertex][i] == 1 && !visited[i]) {

visited[i] = true;

queue.enqueue(i);

}

}

}

delete[] visited;

}

~Graph() {

for (int i = 0; i < noOfVertices; i++) {

delete[] adjMat[i];

}

delete[] adjMat;

}

};

int main() {

Graph graph(10);

graph.makeConnect(0, 1);

graph.makeConnect(0, 6);

graph.makeConnect(0, 8);

graph.makeConnect(1, 4);

graph.makeConnect(1, 6);

graph.makeConnect(1, 9);

graph.makeConnect(2, 4);

graph.makeConnect(2, 6);

graph.makeConnect(3, 4);

graph.makeConnect(3, 5);

graph.makeConnect(3, 8);

graph.makeConnect(4, 5);

graph.makeConnect(4, 9);

graph.makeConnect(7, 8);

graph.makeConnect(7, 9);

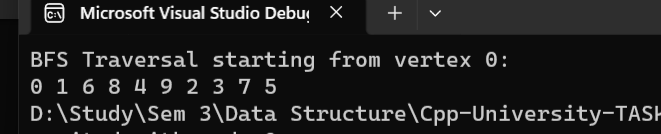
cout << "BFS Traversal starting from vertex 0:\n";

graph.bfsTraversal(0);

return 0;

}

**OUTPUT:**

****

**TASK 4:**

#include <iostream>

using namespace std;

class IntStack {

int\* stackArray;

int size;

int top;

public:

IntStack() {

stackArray = NULL;

size = 0;

top = -1;

}

IntStack(int Size) {

size = Size;

stackArray = new int[size];

top = -1;

}

bool isEmpty() {

return top == -1;

}

bool isFull() {

return top == size - 1;

}

bool push(int x) {

if (isFull()) {

return false;

}

top++;

stackArray[top] = x;

return true;

}

int pop() {

if (isEmpty()) {

return -1; // Return a special value indicating an empty stack

}

int x = stackArray[top];

top--;

return x;

}

~IntStack() {

delete[] stackArray;

}

};

struct Node {

int data;

Node\* next;

};

class Graph {

int noOfvertices = 0;

Node\*\* adjList;

public:

Graph(int num) : noOfvertices(num) {

adjList = new Node \* [num];

for (int i = 0; i < num; ++i) {

adjList[i] = NULL;

}

}

void addEdge(int from, int to) { // undirected graph

Node\* newNodeFrom = new Node;

newNodeFrom->data = to;

newNodeFrom->next = adjList[from];

adjList[from] = newNodeFrom;

Node\* newNodeTo = new Node;

newNodeTo->data = from;

newNodeTo->next = adjList[to];

adjList[to] = newNodeTo;

}

void dfsTraversal(int initialVertex) {

bool\* visited = new bool[noOfvertices];

for (int i = 0; i < noOfvertices; ++i) {

visited[i] = false;

}

IntStack stack(noOfvertices);

visited[initialVertex] = true;

stack.push(initialVertex);

while (!stack.isEmpty()) {

int currentVertex = stack.pop();

cout << currentVertex << " ";

Node\* cur = adjList[currentVertex];

while (cur != NULL) {

if (!visited[cur->data]) {

visited[cur->data] = true;

stack.push(cur->data);

}

cur = cur->next;

}

}

delete[] visited;

}

void displayList() {

cout << "\nAdjacency list:\n";

for (int i = 0; i < noOfvertices; ++i) {

cout << "Node " << i << ": ";

Node\* cur = adjList[i];

while (cur != NULL) {

cout << cur->data << " ";

cur = cur->next;

}

cout << "\n";

}

}

~Graph() {

for (int i = 0; i < noOfvertices; ++i) {

Node\* cur = adjList[i];

while (cur != NULL) {

Node\* temp = cur->next;

delete cur;

cur = temp;

}

}

delete[] adjList;

}

};

int main() {

Graph graph(5);

graph.addEdge(0, 2);

graph.addEdge(0, 1);

graph.addEdge(2, 3);

graph.addEdge(2, 3);

graph.addEdge(1, 4);

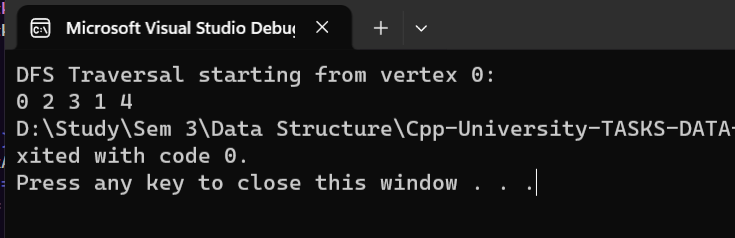
graph.addEdge(1, 3);

cout << "DFS Traversal starting from vertex 0:\n";

graph.dfsTraversal(0);

return 0;

}

**OUTPUT:  
**