# Department of Computing

# School of Electrical Engineering and Computer Science

**CS - 250: Data Structure and Algorithms**

**Class: BSCS 10AB**

**Lab 12 : Implement Graphs and Traverse them in Breadth-First Order**

**Date: 28th December, 2021**

**Time: 10:00 am – 12:50 pm   
&  
 02:00 pm – 4:50 pm**

# Instructor: Dr. Yasir Faheem

# Lab Engineer: Aftab Farooq

# 

# Lab 12 : Implement Graphs and Traverse them in Breadth-First Order

**Introduction**

This lab is based on graphs data structure. You should learn how to implement graphs and how to traverse one using Breadth-First Traversal algorithm.

**Objectives**

In this lab, you will implement graphs using adjacency matrix and adjacency list-based approaches. Moreover, you shall implement the Breadth-First Search traversal algorithm studied in the class.

**Tools/Software Requirement**

Visual Studio C++

**Description of Graphs:**

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On Facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link. Note that anything that has data is a node. Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship. (source: https://www.programiz.com/dsa/graph).

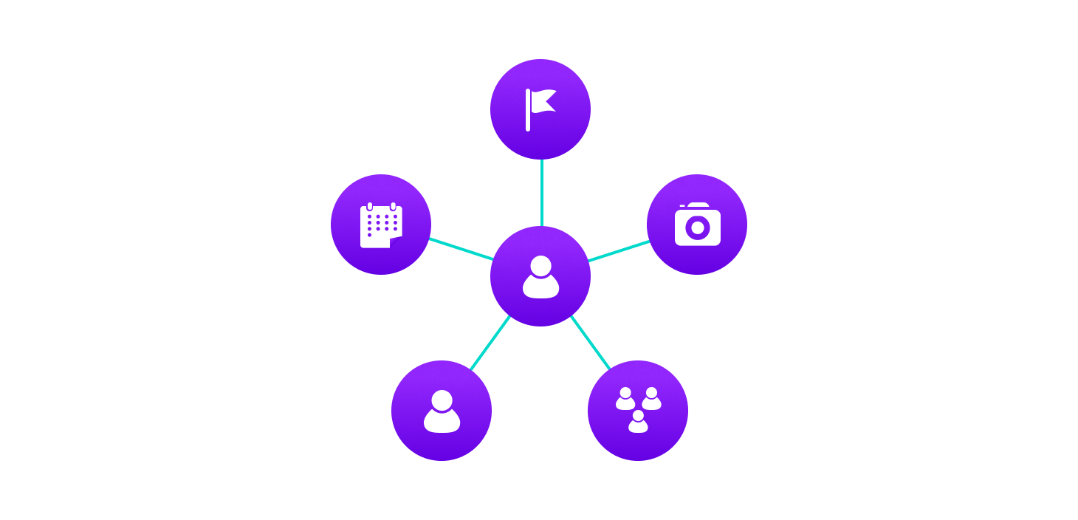


Figure 1:Example of graph data structure

All of Facebook is then a collection of these nodes and edges. This is because Facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

* A collection of vertices V
* A collection of edges E, represented as ordered pairs of vertices (u,v)

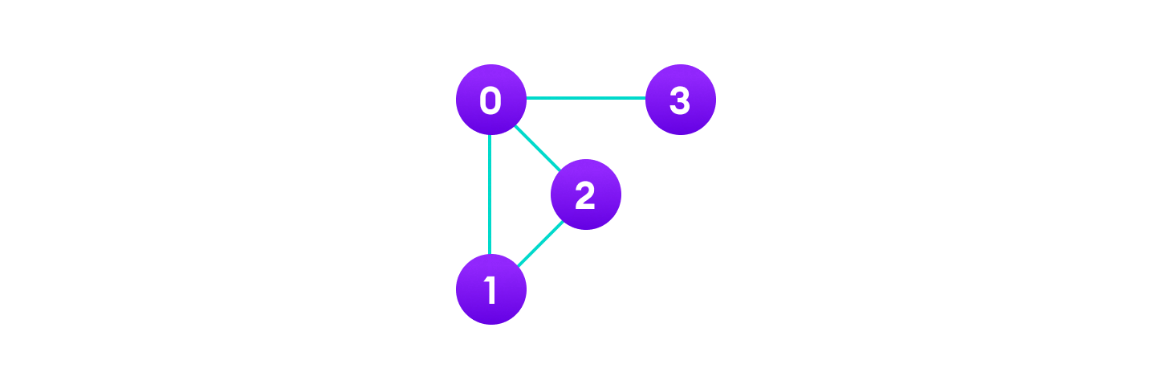


Figure 2: Vertices and edges

In the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

**Graph Terminology**

* **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
* **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
* **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

**Graph Representation**

Graphs are commonly represented in two ways:

**1. Adjacency Matrix**

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex. If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is:

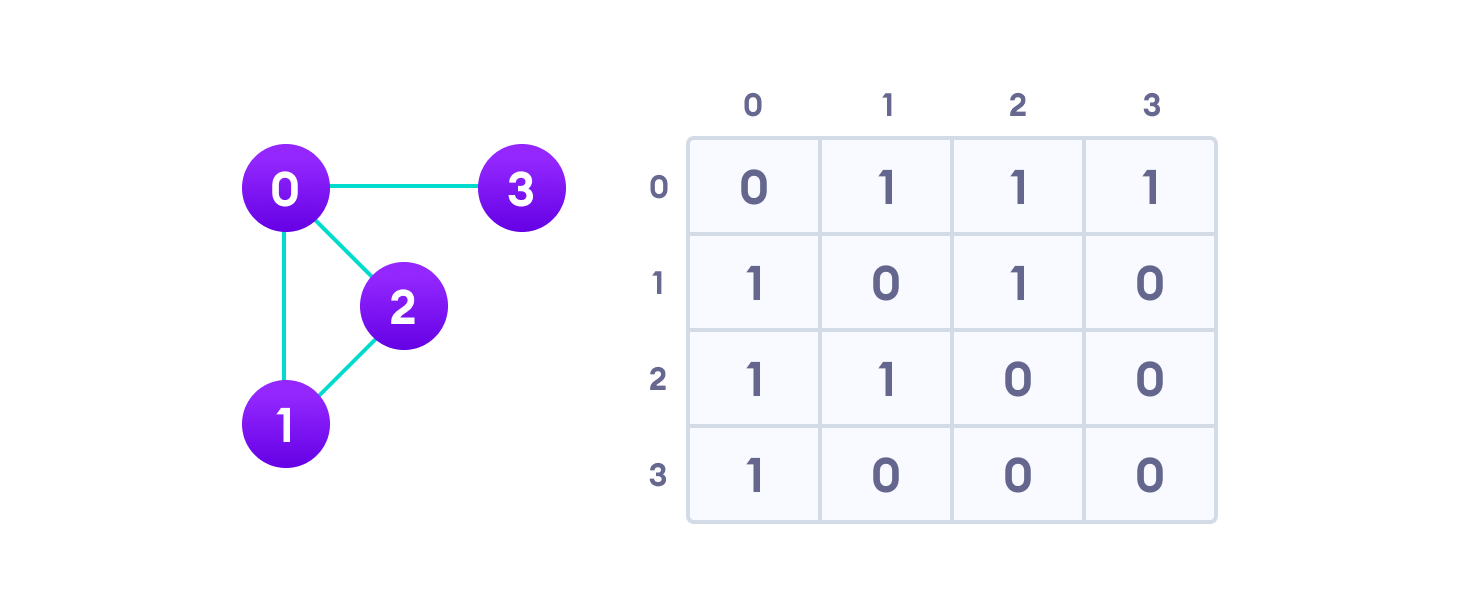


Figure 3: Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

Edge lookup (checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices (V x V), so it requires more space.

**Adjacency Matrix of a Weighted Graph:** If the graph is weighted, then you should store the weight of the edge instead of 1 between vertices vi and vj.

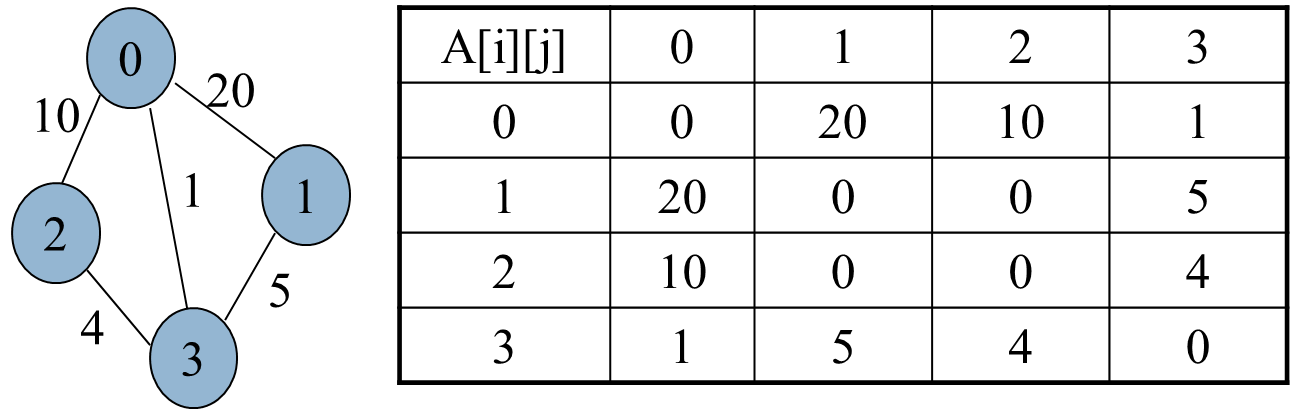


Figure 4: Adjacency Matrix of a Weighted Graph

**Adjacency Matrix of a Directed Graph:** In a directed graph, if there is an edge from vertex vi to vj, then index A[i][j] should store 1. It may store the weight of an edge if the graph is weighted.

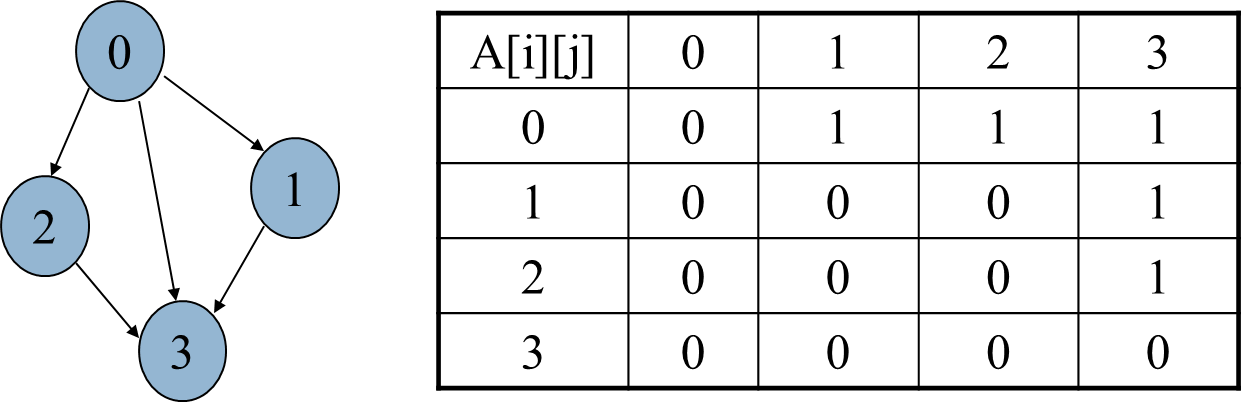
****

Figure 5: Adjacency Matrix of a Directed Graph

**2. Adjacency List**

An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex. The adjacency list for the graph we made in the first example is as follows:

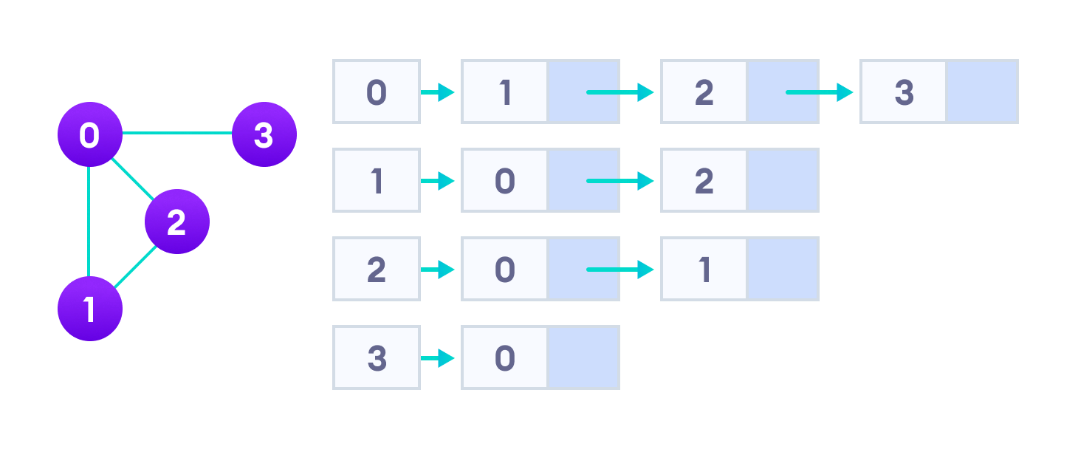


Figure 6: Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

**Graph Operations**

The most common graph operations are:

* Check if the element is present in the graph
* Graph Traversal
* Add elements (vertex, edges) to graph
* Finding the path from one vertex to another

**Lab Tasks**

In this lab, first you should implement the following operations of a graph using both Adjacency Matrix and Adjacency List based representations. Then, you shall implement the breadth-first traversal algorithm using adjacency-list based representation.

**Task A: Implement Adjacency Matrix Representation**

* **Initialize a graph G = (V, E) with |V| vertices in it.**
  + Ask user to enter the number of vertices in a graph.
  + Ask user to enter whether the graph is directed or undirected.
* **InsertEdge(AdjacencyMatrix[], int u, int v, int weight)**
  + Inserts an edge along with its weight from vertex u to v.
  + Set weight to 1 for all edges if the graph is unweighted.
  + If the graph is undirected, you should insert two edges: one from u to v, and another from v to u.
* **PrintGraph(AdjacencyMatrix[])**

This function should print the graph i.e. adjacency matrix.

* **Bool isConnected(int u, int v)**

Checks whether one can go from vertex u to v. Returns true if yes, false otherwise.

|  |
| --- |
| Source code |
| #include<iostream>  using namespace std;  # define vertices 10  class adjacencymatrix {  public:  bool isdirected;  bool isweighted;  int weight;  int size;  int\*\* admatrix;  adjacencymatrix() {  admatrix = new int\* [size];  for (size\_t i = 0; i < size; i++)  {  admatrix[i] = new int[size];  }  for (size\_t i = 0; i < size; i++)  {  for (size\_t j = 0; j < size; j++)  {  admatrix[i][j] = 0;  }  }  }  adjacencymatrix(int s, bool directed, bool weighted)  {  size = s;  isdirected = directed;  isweighted = weighted;  weight = 1;  admatrix = new int\* [size];  for (size\_t i = 0; i < size; i++)  {  admatrix[i] = new int[size];  }  for (size\_t i = 0; i < size; i++)  {  for (size\_t j = 0; j < size; j++)  {  admatrix[i][j] = 0;  }  }  }  void insertEdge( int e1, int e2, int weight) {  if (e1 < size && e2 < size) {  admatrix[e1][e2] = weight;  if (!isdirected) {  admatrix[e2][e1] = weight;  }  }  }  void printGraph() {    int i, j;  cout << " ";  for (size\_t j = 0; j < size; j++)  {  cout << j << " ";  }  cout << endl;  for (i = 0; i < size; i++)  {  cout << i << " ";  for (j = 0; j < size; j++)  {  cout << admatrix[i][j] << " ";  }  cout << endl;  }    }  bool isconnected(int e1, int e2) {  if (e1 < size && e2 < size) {  return admatrix[e1][e2];  }  else return false;  }  };  int main() {  bool isdirected;  bool isweighted;  int weight;  int size;  int option;  cout << "Enter number of vertices: ";  cin >> size;  cout << "Is your graph directed? \n";  cout << "1. Yes\t2.No\n";  cout << "option: ";  cin >> option;  if (option == 1) {  isdirected = true;  }  else {  isdirected = false;  }  cout << "Is your graph weighted? \n";  cout << "1. Yes\t2.No\n";  cout << "option: ";  cin >> option;  if (option == 1) {  isweighted = true;  }  else {  isweighted = false;  }  adjacencymatrix graph(size, isdirected, isweighted);  while (true) {  system("CLS");  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Graphs adjancency matrix\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "\t1.Insert edge\n";  cout << "\t2.Print matrix\n";  cout << "\t3.Check if edges are connected\n";  cout << "\t4.Exit\n";  cout << "Option: ";  cin >> option;  if (option == 1) {  int v1, v2;  cout << "Enter vertice no.1: ";  cin >> v1;  cout << "Enter vertice no.2: ";  cin >> v2;  weight = 1;  if (isweighted) {  cout << "Enter weight: ";  cin >> weight;  }  graph.insertEdge(v1, v2, weight);  cout << "Edge is inserted\n";  }  else if (option == 2) {  cout << endl;  graph.printGraph();  }  else if (option == 3) {  int v1, v2;  cout << "Enter vertice no.1: ";  cin >> v1;  cout << "Enter vertice no.2: ";  cin >> v2;  if (graph.isconnected(v1, v2)) {  cout << "Vertices are conected\n";  }  else  cout << "Vertices are not connected\n";  }  else if (option == 4) {  break;  }  else {  cout << "Enter a valid command\n";  }  system("pause");  }  } |
| Output |
|  |

**Task B: Implement Adjacency List Representation**

* **Initialize a graph G = (V, E) with |V| vertices in it.**
  + Ask user to enter the number of vertices in a graph.
  + Ask user to enter whether the graph is directed or undirected.
* **InsertEdge(AdjacencyMatrix[], int u, int v, int weight)**
  + Inserts an edge along with its weight from vertex u to v.
  + Set weight to 1 for all edges if the graph is unweighted; otherwise, ask the user to enter weight for each edge.
  + If the graph is undirected, you should insert two edges: one from u to v, and another from v to u.
* **PrintGraph(AdjacencyList[])**

This function should print the graph i.e. the adjacency list of each of the |V| vertices.

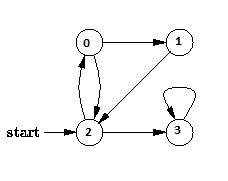
* **Bool isConnected(int u, int v)**

Checks whether one can go from vertex u to v. Returns true if yes, false otherwise.

|  |
| --- |
| Source code |
| // Abdullah Shakeel CMS: 332380 Class: BSCS 10-A  #include<iostream>  using namespace std;  class Node {  public:  int edge;  int weight;  Node\* next;  Node() {  weight = 0;  edge = 0;  next = NULL;  }  };  class LinkedList {  public:  Node\* start; // special variable which stores address of the head node.  Node\* last; // special variable which stores address of the last node.  Node\* Ploc; //to be used by Search(value) method to store address of logical predecessor of value in a list.  Node\* Loc\_; //to be used by Search(value) method to store address of the node containing the searched value in a list. If it is not found it contains NULL.  //Constructor  LinkedList() {  start = NULL;  Ploc = NULL;  Loc\_ = NULL;  }  //Functions  bool isEmpty() {  if (start == NULL)  return true;  else  return false;  }  void InsertAtLast(int edge,int weight) {  Node\* node = new Node;  node->edge = edge;  node->weight = weight;  if (isEmpty())  {  start = node;  last = node;  }  //If list is not empty  else {  last->next = node;  last = node;  }  }  void PrintList() {  if (!isEmpty()) {  Node\* temp = new Node;  temp = start;    while (temp != NULL) {  cout <<"Vertice: "<< temp->edge << "\tWeight: "<<temp->weight<<endl;  temp = temp->next;  }  cout << endl;  }  else {  cout << "Not connected" << endl;  }  }  bool Search(int value) {  Loc\_ = start;  Ploc = NULL;  if (!isEmpty()) {  while (Loc\_ != NULL && Loc\_->edge !=value) {  Ploc = Loc\_;  Loc\_ = Loc\_->next;  }  if (Loc\_ == NULL) {  return false;  }  else {  return true;  }  }  }    void Delete(int value) {  Search(value);  if (Loc\_ != NULL) {  if (Ploc == NULL)//value is at front  {  start = Loc\_->next;  }  else {  if (Loc\_ == last) {  last = Ploc;  }  Ploc->next = Loc\_->next;  }  delete Loc\_;  }  else  cout << "Data not found" << endl;  }  void DestroyList() {  Loc\_ = start;  Ploc = NULL;  while (Loc\_ != NULL) {  Ploc = Loc\_;  Loc\_ = Loc\_->next;  delete Ploc;  }  start = NULL;  last = NULL;  }  };  class adjancencyList {  public:  bool isdirected;  bool isweighted;  int weight;  int size=10;  LinkedList\* admatrix;  adjancencyList() {  admatrix = new LinkedList[size];    }  adjancencyList(int s, bool directed, bool weighted)  {  size = s;  isdirected = directed;  isweighted = weighted;  weight = 1;  admatrix = new LinkedList[size];  }  void insertEdge(int e1, int e2, int weight) {  if (e1 < size && e2 < size) {  admatrix[e1].InsertAtLast(e2,weight);  if (!isdirected) {  admatrix[e2].InsertAtLast(e1,weight);  }  }  }  void printGraph() {        for (size\_t i = 0; i < size; i++)  {  cout << "Vertice: " << i << "\n";  admatrix[i].PrintList();  cout << "\n\n";  }  }  bool isconnected(int e1, int e2) {  if (e1 < size && e2 < size) {  return admatrix[e1].Search(e2);  }  else return false;  }  };  int main() {  bool isdirected;  bool isweighted;  int weight;  int size;  int option;  cout << "Enter number of vertices: ";  cin >> size;  cout << "Is your graph directed? \n";  cout << "1. Yes\t2.No\n";  cout << "option: ";  cin >> option;  if (option == 1) {  isdirected = true;  }  else {  isdirected = false;  }  cout << "Is your graph weighted? \n";  cout << "1. Yes\t2.No\n";  cout << "option: ";  cin >> option;  if (option == 1) {  isweighted = true;  }  else {  isweighted = false;  }  adjancencyList graph(size, isdirected, isweighted);  while (true) {  system("CLS");  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Graphs adjancency matrix\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "\t1.Insert edge\n";  cout << "\t2.Print matrix\n";  cout << "\t3.Check if edges are connected\n";  cout << "\t4.Exit\n";  cout << "Option: ";  cin >> option;  if (option == 1) {  int v1, v2;  cout << "Enter vertice no.1: ";  cin >> v1;  cout << "Enter vertice no.2: ";  cin >> v2;  weight = 1;  if (isweighted) {  cout << "Enter weight: ";  cin >> weight;  }  graph.insertEdge(v1, v2, weight);  cout << "Edge is inserted\n";  }  else if (option == 2) {  cout << endl;  graph.printGraph();  }  else if (option == 3) {  int v1, v2;  cout << "Enter vertice no.1: ";  cin >> v1;  cout << "Enter vertice no.2: ";  cin >> v2;  if (graph.isconnected(v1, v2)) {  cout << "Vertices are conected\n";  }  else  cout << "Vertices are not connected\n";  }  else if (option == 4) {  break;  }  else {  cout << "Enter a valid command\n";  }  system("pause");  }  } |
| Output |
|  |

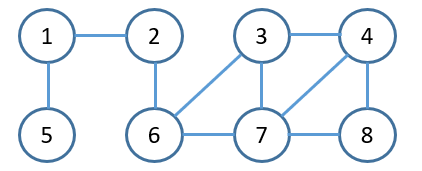
**Task C: Implement Breadth-First Traversal of a Graph Stored as Adjacency List Representation**

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

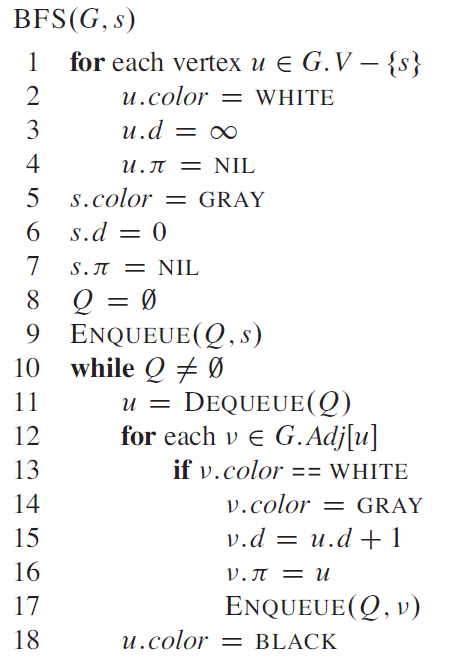
[](https://www.geeksforgeeks.org/wp-content/uploads/BFS.jpg)

**Task**

Implement BFS traversal algorithm studied in class for a graph with 8 vertices.

As an initial input we have the graph (G) with 8 nodes shown below with and a starting vertex 2.

The Pseudocode of BFS is given below:

****Your code should print the correct BFS sequence for an arbitrary starting vertex s.

|  |
| --- |
| Source code |
| // Abdullah Shakeel CMS: 332380 Class: BSCS 10-A  #include<iostream>  using namespace std;  class Node {  public:  int vertice;  int weight;  Node\* next;  Node() {  weight = 0;  vertice = 0;  next = NULL;  }  };  class LinkedList {  public:  Node\* start=NULL; // special variable which stores address of the head node.  Node\* last; // special variable which stores address of the last node.  Node\* Ploc; //to be used by Search(value) method to store address of logical predecessor of value in a list.  Node\* Loc\_; //to be used by Search(value) method to store address of the node containing the searched value in a list. If it is not found it contains NULL.  //Constructor  LinkedList() {  start = NULL;  Ploc = NULL;  Loc\_ = NULL;  last = NULL;  }  //Functions  bool isEmpty() {  if (start == NULL)  return true;  else  return false;  }  void InsertAtLast(int vertice,int weight) {  Node\* temp = new Node;  temp->vertice = vertice;  temp->weight = weight;  if (isEmpty())  {  start = temp;  last = temp;  }  else {  last->next = temp;  last = temp;  }  }  void PrintList() {  if (!isEmpty()) {  Node\* temp = new Node;  temp = start;    while (temp != NULL) {  cout <<"Vertice: "<< temp->vertice << "\tWeight: "<<temp->weight<<endl;  temp = temp->next;  }  cout << endl;  }  else {  cout << "Not connected" << endl;  }  }  bool Search(int value) {  Loc\_ = start;  Ploc = NULL;  if (!isEmpty()) {  while (Loc\_ != NULL && Loc\_->vertice !=value) {  Ploc = Loc\_;  Loc\_ = Loc\_->next;  }  if (Loc\_ == NULL) {  return false;  }  else {  return true;  }  }  }    void Delete(int value) {  Search(value);  if (Loc\_ != NULL) {  if (Ploc == NULL)//value is at front  {  start = Loc\_->next;  }  else {  if (Loc\_ == last) {  last = Ploc;  }  Ploc->next = Loc\_->next;  }  delete Loc\_;  }  else  cout << "Data not found" << endl;  }  void DestroyList() {  Loc\_ = start;  Ploc = NULL;  while (Loc\_ != NULL) {  Ploc = Loc\_;  Loc\_ = Loc\_->next;  delete Ploc;  }  start = NULL;  last = NULL;  }  };  class Queue {  private:  int arr[100];  int count = 0;  int front = 0;  int rear = -1;  public:  bool isEmpty() {  return count == 0;  }  bool isFull() {  return count == 100;  }  void Enqueue(int data) {  if (!isFull()) {  rear = ++rear % 100;  arr[rear] = data;  count++;  }  else  cout << "Queue is full\n";  }  int Dequeue() {  if (!isEmpty()) {  front = ++front % 100;  count--;  if (front != 0)  {  return arr[front - 1];  }  else  return arr[99];  }  else  {  cout << "Queue is empty\n";  return -99999;  }  }  int Clear() {  int front = 0;  int rear = -1;  }  int FirstElement() {  return arr[front];  }  };  class AdjancencyList  {  public:  int size = 10;  LinkedList\* adj;    AdjancencyList() {  adj = new LinkedList[size];    }  AdjancencyList(int s)  {  size = s;  adj = new LinkedList[size];  }  void insertEdge(int v, int w)  {  adj[v].InsertAtLast(w,1);  adj[w].InsertAtLast(v,1);  }  void BFS(int s)  {  bool\* isVisited = new bool[size];  for (int i = 0; i < size; i++)  isVisited[i] = false;  Queue queue;  isVisited[s] = true;  queue.Enqueue(s);  Node\* i = NULL;  cout << "\tBFS Traveral from "<<s<< " is: ";  while (!queue.isEmpty())  {  s = queue.Dequeue();  cout << s << " ";    i = adj[s].start;  while(i != NULL)  {  if (!isVisited[i->vertice])  {  isVisited[i->vertice] = true;  queue.Enqueue(i->vertice);  }  i=i->next;  }  }  }  };  int main() {  AdjancencyList graph(9);  int startingpoint;  graph.insertEdge(1, 5);  graph.insertEdge(1, 2);  graph.insertEdge(2, 6);  graph.insertEdge(6, 3);  graph.insertEdge(6, 7);  graph.insertEdge(3, 7);  graph.insertEdge(3, 4);  graph.insertEdge(4, 7);  graph.insertEdge(4, 8);  graph.insertEdge(7, 8);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*88\*\*\*Breadth First Seacrh\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout << "\tEnter the starting point: ";  cin >> startingpoint;  cout << endl;  graph.BFS(startingpoint);  cout << endl <<"\t";  system("pause");  return 0;  } |
| Output |
|  |

**Deliverables:**

Compile a single word document by filling in the solution part and submit this Word file on LMS. The name of word document should follow this format. i.e. **YourFullName(reg)\_Lab#.** This lab grading policy is as follows: The lab is graded between 0 to 10 marks. The submitted solution can get a maximum of 5 marks. At the end of each lab or in the next lab, there will be a viva related to the tasks. The viva has a weightage of 5 marks. Insert the solution/answer in this document. You must show the implementation of the tasks in the designing tool, along with your complete Word document to get your work graded. You must also submit this Word document on the LMS. In case of any problems discuss it by emailing it to [aftab.farooq@seecs.edu.pk](mailto:aftab.farooq@seecs.edu.pk).

**Note:** Students are required to upload the lab on LMS before deadline.

Use proper indentation and comments. Lack of comments and indentation will result in deduction of marks.