CS-250

Data Structures and Algorithms

Lab 14

GRAPHS

Name: Mahum Samar

Class: BSCS-9B

CMS ID: 290647

# Code:

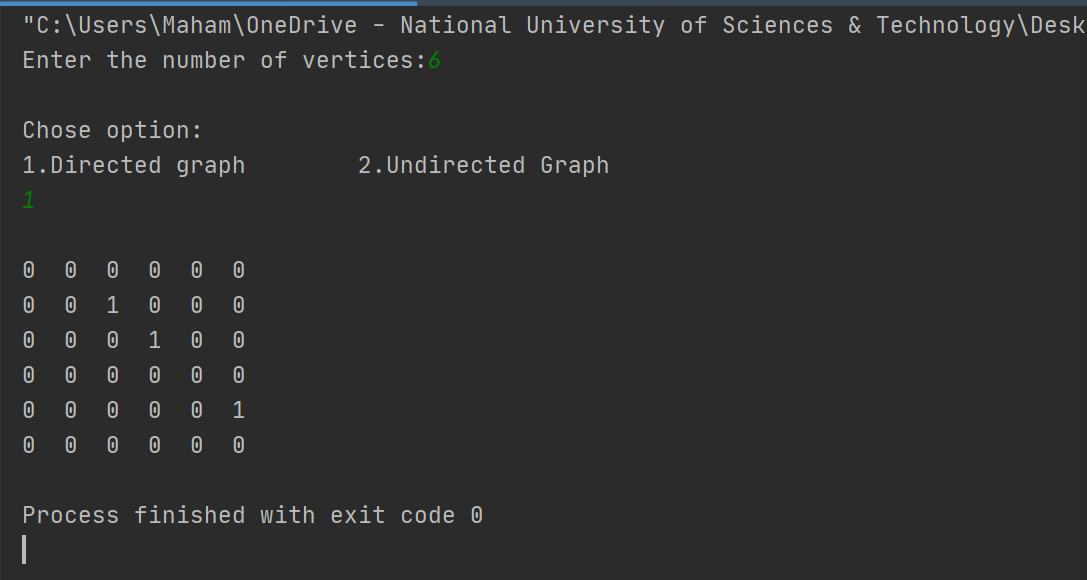
|  |
| --- |
| *//Mahum Samar //BSCS-9B //290647* #include <iostream>  # define MAX 100  using namespace std;  class GraphAdjacencyList;  template<typename t> class LinearQueue {  *//a generic class for creating linear queue* public: *// creates queue of type specified by the user* t queue[MAX];  int front;  int rear;    LinearQueue() {  *//constructor* front = 0;  rear = -1;  }   bool isEmpty() {  *//method to check if the list is empty.* if (rear < front) {  return true;  } else {  return false;  }  }   bool isFull() {  *//method to check if the list is full* if (rear == MAX - 1) {  return true;  } else {  return false;  }  }   void EnQueue(t x) {  *//method to input the element at the end of the queue.* if (isFull()) {  cout << "Queue is Full. Queue Overflow." << endl;  } else {  rear++;  queue[rear] = x;  }  }    t DeQueue() {  *//method to remove the first element of the queue* if (isEmpty()) {  cout << "Queue is empty. Queue Underflow." << endl;  } else {  *//if elements in the Queue* t x = queue[front];  front++;  return x;  }  }    void FirstElement() {  *//method to print the first element of the queue* if (isEmpty()) {  cout << "Queue is Empty." << endl;  } else {  cout << queue[front] << endl;  }  }   void Clear() {  *//method to empty the queue* if (isEmpty()) {  cout << "Queue is already empty." << endl;  } else {  front = 0;  rear = -1;  }  }   void ClearAndDisplay() {  *//method to empty the queue as well as display its elements* if (isEmpty()) {  cout << "Queue is already empty." << endl;  } else {  while (!isEmpty()) {  cout << DeQueue() << " ";  }  cout << endl;  }  }   void Display() {  *//method to display the queue but not empty it.* LinearQueue<t> \*lq = new LinearQueue<t>();   if (isEmpty()) {  cout << "Queue is already empty." << endl;  } else {  while (!isEmpty()) {  *//printing the elemnts of the queue as well as storing it in the temporary queue* t x = DeQueue();  cout << x << " ";  lq->EnQueue(x);  }  while (!(lq->isEmpty())) {  *//removing the elemnts from the temporary queue and moving it back to the original queue* t x = lq->DeQueue();  EnQueue(x);  }  }   cout << endl;  }  };  class GraphAdjacencyMatrix {  *// class for implementing adjacency matrix form of graph  //in this class 1 dimensional array is used.  //and it is accessed as a 2 dimensional array for storing the graph edges* public:  int vertex; *//total number of vertices* int matrixSize; *//maximum size of array* void InitializeGraph(int AdjacencyMatrix[]) {  *//method to initialize 1 dimensional array* for (int i = 0; i < matrixSize; i++) {  AdjacencyMatrix[i] = 0;  }  }   void InsertEdgeUndirected(int AdjacencyMatrix[], int u, int v, int weight) {  *//method used to insert the undirected edges in graph* int n1 = u \* vertex + v; *//calculates from vertex u to v* int n2 = v \* vertex + u; *//calculates from vertex u to v  //adds the user defined weight* AdjacencyMatrix[n1] = weight;  AdjacencyMatrix[n2] = weight;  }   void InsertEdgeDirected(int AdjacencyMatrix[], int u, int v, int weight) { *// method for undirected graph  //calculates from vertex u to v* u = u \* vertex + v;  AdjacencyMatrix[u] = weight;  }   void PrintGraph(int AdjacencyMatrix[]) {  *//method to print graph on screen* for (int i = 0; i < matrixSize; i++) {  if (i % vertex == 0) { *// adds new line after the vertex length of array values are printed* cout << "\n";  }  cout << AdjacencyMatrix[i] << " ";  }  cout << "\n";  }   bool IsConnected(int AdjacencyMatrix[], int u, int v) {  *//method ot check whether u has edge towards v* u = u \* vertex + v;  if (AdjacencyMatrix[u] == 0) {  return false;  } else {  return true;  }  }  };  class AdjacencyListNode {  *// class to create node for adjacent vertices singly list* public:  int data;  int weight;  AdjacencyListNode \*next; };  class GraphArray { *// class to create node for graph array* public:  int data;  string color;  int distance;  int previousNode; *// stores reference to the list containing adjacent vertices* GraphAdjacencyList \*graphAdjacencyList; };  class GraphAdjacencyList { *// class to create singly linked list* public:  int vertex;  AdjacencyListNode \*headNode;  AdjacencyListNode \*lastNode;   bool IsEmpty(GraphAdjacencyList \*list) { *// method to check whether the list is empty or not* return list == NULL;  }   void InitializeAdjacencyListArray(GraphArray AdjacencyList[]) { *// method to initialize graph array* for (int i = 0; i < vertex; ++i) { *// the label is given same as the index number* AdjacencyList[i].data = i;  AdjacencyList[i].graphAdjacencyList = NULL;  }   }   void InsertAdjacencyListNode(GraphArray \*arrayIndex, int value, int weight) { *// method to insert the node in adjacent node singly list* AdjacencyListNode \*adjacencyListNode = new AdjacencyListNode();  adjacencyListNode->data = value;  adjacencyListNode->weight = weight;  *// cout << "New node made.\n";* if (IsEmpty(arrayIndex->graphAdjacencyList)) { *// if the list is empty // cout << "Empty list initialized for.\t" << value << "\n";* arrayIndex->graphAdjacencyList = new GraphAdjacencyList();  arrayIndex->graphAdjacencyList->headNode = adjacencyListNode;  arrayIndex->graphAdjacencyList->lastNode = adjacencyListNode;  } else { *// list already has some nodes* arrayIndex->graphAdjacencyList->lastNode->next = adjacencyListNode;  arrayIndex->graphAdjacencyList->lastNode = adjacencyListNode;  }  }   void InsertEdgeUndirected(GraphArray AdjacencyList[], int u, int v, int weight) { *// method to insert edge in an undirected graph* InsertAdjacencyListNode(&AdjacencyList[u], v, weight);  InsertAdjacencyListNode(&AdjacencyList[v], u, weight);  }   void InsertEdgeDirected(GraphArray AdjacencyList[], int u, int v, int weight) { *// method to insert edge in directed graph* InsertAdjacencyListNode(&AdjacencyList[u], v, weight);  }   void PrintAdjacencyList(GraphAdjacencyList \*graphAdjacencyList) { *// Method to print the adjacent vertices of one vertex* AdjacencyListNode \*temp = graphAdjacencyList->headNode;  while (temp != NULL) {  cout << "Adjacent vertex: " << temp->data << " "; *// cout << "Weight: "<<temp->weight << " ";* temp = temp->next;  }  cout << "\n";  }   void PrintAdjacencyVertex(GraphArray AdjacencyList[]) { *// Method to print all the vertices included their adjacent vertices* for (int i = 0; i < vertex; ++i) {  cout << "For vertex :\t" << AdjacencyList[i].data << "\t";  cout << "Its Adjacency List is :\t";  if (AdjacencyList[i].graphAdjacencyList == NULL) {  cout << "There is no adjacent vertex.\n";  } else { *// If there are some adjacent matrices they are printed calling their PrintAdjacencyList method* PrintAdjacencyList(AdjacencyList[i].graphAdjacencyList);  }  }  }   bool IsConnected(GraphArray AdjacencyList[], int u, int v) { *// Method to check if the given two vertices are directly connected or not //it returns a boolean value* AdjacencyListNode \*temp = AdjacencyList[u].graphAdjacencyList->headNode;  while (temp->data != v) {  temp = temp->next;  }  if (temp->data == v) {  return true;  } else {  return false;  }  }   void BreadthFirstSearch(GraphArray AdjacencyList[], GraphArray \*u) { *// Method for implementing breadth first search algorithm // It uses queue data structure* for (int i = 0; i < vertex; ++i) { *// Initialize each vertex color to white and infinity distance is taken as 10,000 // previous vertex initiallize to zero* AdjacencyList[i].color = "white";  AdjacencyList[i].distance = 10000;  AdjacencyList[i].previousNode = NULL;  } *// First vertex color set to grey* AdjacencyList[u->data].color = "gray";  AdjacencyList[u->data].distance = 0;  AdjacencyList[u->data].previousNode = NULL; *// Linear queue initialized* LinearQueue<GraphArray \*> linearQueue = \*new LinearQueue<GraphArray \*>(); *// First vertex is enqueued* linearQueue.EnQueue(u);  while (!(linearQueue.isEmpty())) { *// While the queue is not empty* u = linearQueue.DeQueue();  if (u->graphAdjacencyList != NULL) { *// if adjacent vertices are present* AdjacencyListNode \*temp = u->graphAdjacencyList->headNode;  for (AdjacencyListNode \*v = temp; v != NULL; v = temp) { *// Until there are adjacent nodes* if (AdjacencyList[v->data].color == "white") {  AdjacencyList[v->data].color = "gray";  AdjacencyList[v->data].distance =  AdjacencyList[AdjacencyList[v->data].previousNode].distance + 1;  AdjacencyList[v->data].previousNode = u->data; *// the adjacent nodes are enqueued* linearQueue.EnQueue(&AdjacencyList[v->data]);  }  temp = temp->next;  }   } else { *// If there are no adjacent vertices then pointer goes to next array index of the graph* int j = u->data + 1;  linearQueue.EnQueue(&AdjacencyList[j]);  } *// When all neighbors are visited the color of the vertex is black* AdjacencyList[u->data].color = "black";  }  }   void PrintBreadthFirst(GraphArray AdjacencyList[]) { *// Method to print the vertices of graph containing all their information* for (int i = 0; i < vertex; ++i) {  cout << "data: " << AdjacencyList[i].data << " ";  cout << "color: " << AdjacencyList[i].color << " ";  cout << "distance: " << AdjacencyList[i].distance << " ";  cout << "previous node: " << AdjacencyList[i].previousNode << "\n";  }  cout << "\n";  }  }; |

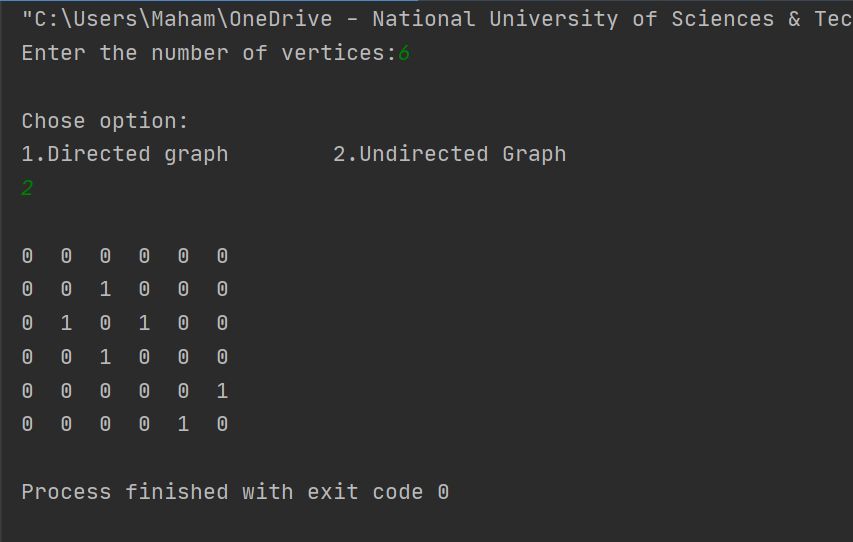
Task 01

# Code:

|  |
| --- |
| int main() { *// Adjacency Matrix* GraphAdjacencyMatrix \*graphAdjacencyMatrix = new GraphAdjacencyMatrix();  cout<< "Enter the number of vertices: \t";  cin>>graphAdjacencyMatrix->vertex;  *// initializing graph* graphAdjacencyMatrix->matrixSize = graphAdjacencyMatrix->vertex \* graphAdjacencyMatrix->vertex;  int AdjacencyMatrix[graphAdjacencyMatrix->matrixSize];  graphAdjacencyMatrix->InitializeGraph(AdjacencyMatrix);  int option;  cout<<"\nChose option:\n1.Directed graph\t2.Undirected Graph\n";  cin>>option;  if(option==1)  {  graphAdjacencyMatrix->InsertEdgeDirected(AdjacencyMatrix,1,2,1);  graphAdjacencyMatrix->InsertEdgeDirected(AdjacencyMatrix,2,3,1);  graphAdjacencyMatrix->InsertEdgeDirected(AdjacencyMatrix,4,5,1);  }  else{  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix,1,2,1);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix,2,3,1);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix,4,5,1);  }   graphAdjacencyMatrix->PrintGraph(AdjacencyMatrix);  return 0; } |

# Output:





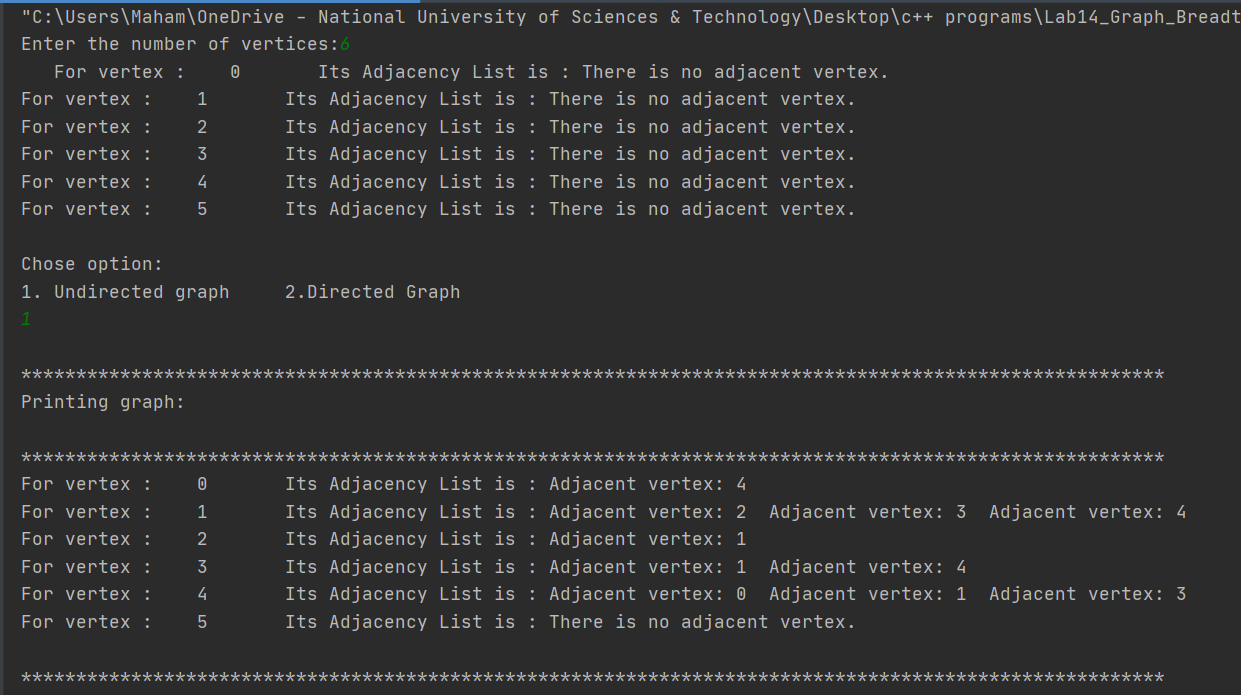
Task 01

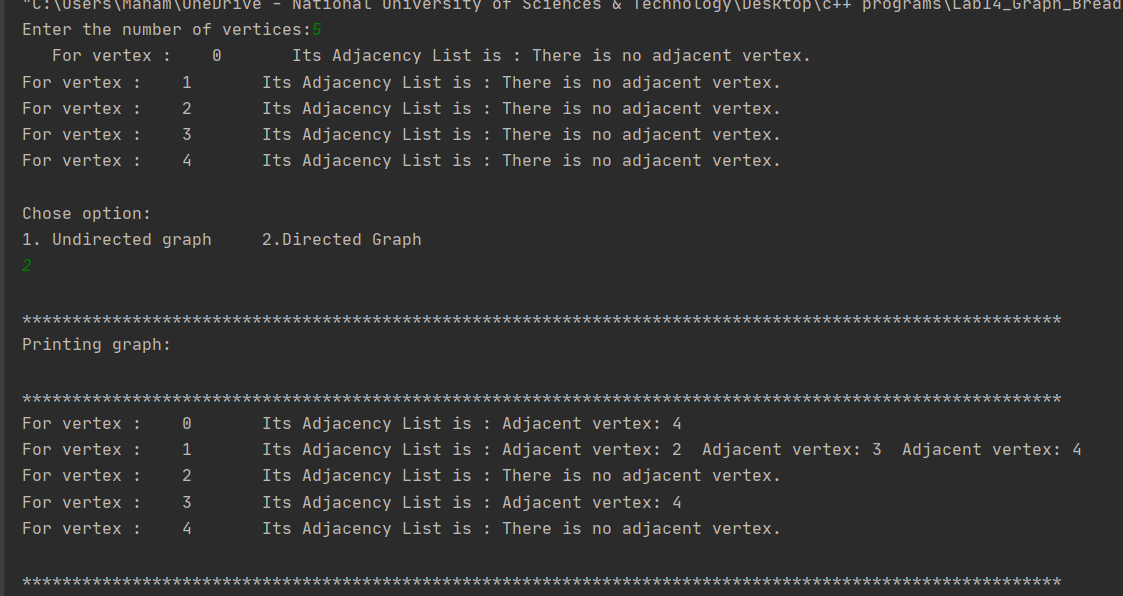
Task 02

# Code:

|  |
| --- |
| int main() {  GraphAdjacencyList \*graphAdjacencyList = new GraphAdjacencyList();  cout<< "Enter the number of vertices: \t";  cin>>graphAdjacencyList->vertex;   GraphArray graph[graphAdjacencyList->vertex];  graphAdjacencyList->InitializeAdjacencyListArray(graph);  graphAdjacencyList->PrintAdjacencyVertex(graph);   int option;  cout<<"\nChose option:\n1. Undirected graph\t2.Directed Graph\n";  cin>>option;  if(option==1)  {  *// When graph is undirected* graphAdjacencyList->InsertEdgeUndirected(graph, 0, 4, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 1, 2, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 1, 3, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 1, 4, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 3, 4, 1);  }  else{  *// Graph is directed* graphAdjacencyList->InsertEdgeDirected(graph, 0, 4, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 1, 2, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 1, 3, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 1, 4, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 3, 4, 1);  }  cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout<<"Printing graph:\n";  cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  graphAdjacencyList->PrintAdjacencyVertex(graph); *// breadth first search method called* cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";   return 0; } |

# Output:





Task 03

# Code:

|  |
| --- |
| int main() {  GraphAdjacencyList \*graphAdjacencyList = new GraphAdjacencyList();  cout<< "Enter the number of vertices: \t";  cin>>graphAdjacencyList->vertex;   GraphArray graph[graphAdjacencyList->vertex];  graphAdjacencyList->InitializeAdjacencyListArray(graph);  graphAdjacencyList->PrintAdjacencyVertex(graph);   int option;  cout<<"\nChose option:\n1. Undirected graph\t2.Directed Graph\n";  cin>>option;  if(option==1)  {  *// When graph is undirected* graphAdjacencyList->InsertEdgeUndirected(graph, 0, 4, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 1, 2, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 1, 3, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 1, 4, 1);  graphAdjacencyList->InsertEdgeUndirected(graph, 3, 4, 1);  }  else{  *// Graph is directed* graphAdjacencyList->InsertEdgeDirected(graph, 0, 4, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 1, 2, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 1, 3, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 1, 4, 1);  graphAdjacencyList->InsertEdgeDirected(graph, 3, 4, 1);  }  cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  cout<<"Printing graph:\n";  cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  graphAdjacencyList->PrintAdjacencyVertex(graph); *// breadth first search method called* cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"; cout<<"Breadth First Search\n";  cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  graphAdjacencyList->BreadthFirstSearch(graph, &graph[0]); *// Method to print graph nodes after BFS* graphAdjacencyList->PrintBreadthFirst(graph);  cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";  return 0; } |

# Output:

