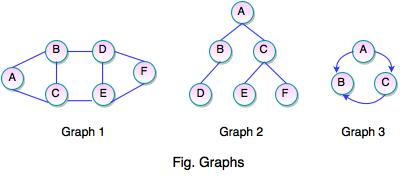
Graphs

A graph is a non-linear data structure. It can be defined as group of vertices and edges represented as G(V,E) . It consists of set of i) vertices and ii) edges.

V is a finite number of vertices also called as nodes

E is a set of ordered pair of vertices representing edges.

Example:



the graphs can be represented as:

Graph 1:

V={ A,B,C,D,E,F}

E={(A,B),(A,C),(B,C),(B,D),(D,E),(D,F),(E,F)}

Graph2:

V = {A, B, C, D, E, F}  
E = {(A, B), (A, C), (B, D), (C, E), (C, F)}

GRAPH 3:

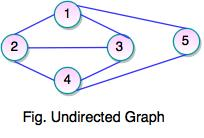
V = {A, B, C}  
E = {(A, B), (A, C), (C, B)}

A graph can be

* directed
* undirected.

Undirected graph

In an undirected graph, edges are not associated with the directions within them. In the above figure Graph1 and Graph2 are undirected graph.If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

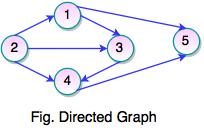


In the above figure :

Set of Vertices V = {1, 2, 3, 4, 5}  
Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (3,1),(3,2),(4, 5)..}

Directed Graph

* a graph consisting of ordered pairs of vertices and has direction, is said to be directed graph
* if an edge is represented using a pair of vertices(V1,V2), the edge is said to be directed from V1 to V2.
* Graph 3 in the above is an example of directed graph. Path is said to be exists from A to C only not from C to A.



In the above figure :

Set of Vertices V = {1, 2, 3, 4, 5, 5}  
Set of Edges W = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

***Graph Terminology***

**Path:**

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

### Closed Path

A path will be called a closed path if the initial node is the same as the terminal node. A path will be a closed path if V0=VN.

### Simple Path

If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called a closed simple path.

### Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

A simple graph with ‘n’ vertices (n >= 3) and ‘n’ edges is called a cycle graph if all its edges form a cycle of length ‘n’.

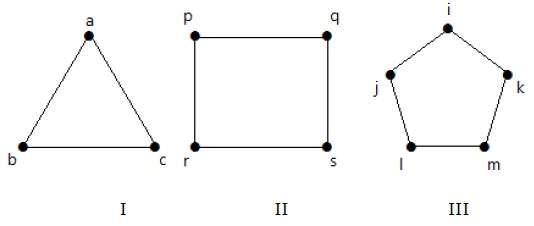
If the **degree of each vertex in the graph is two,** then it is called a Cycle Graph.

**Notation** − Cn

### Example

Take a look at the following graphs −

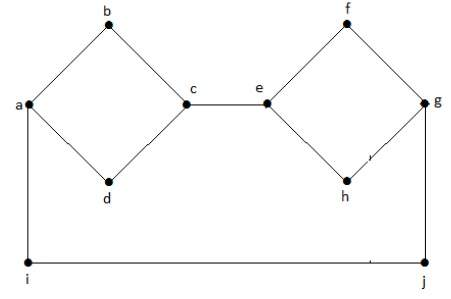
* Graph I has 3 vertices with 3 edges which form a cycle ‘ab-bc-ca’.
* Graph II has 4 vertices with 4 edges which form a cycle ‘pq-qs-sr-rp’.
* Graph III has 5 vertices with 5 edges which is forming a cycle ‘ik-km-ml-lj-ji’.



Hence all the given graphs are cycle graphs.

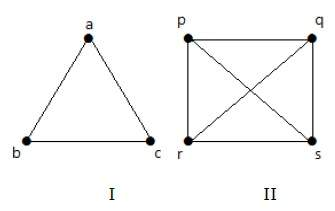
### Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in the connected graph.



### Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contains n(n-1)/2 edges where n is the number of nodes in the graph.



### Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

### Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

### Loop

An edge that is associated with the similar endpoints can be called a Loop. In a graph, if an edge is drawn from vertex to itself, it is called a loop.

### Example 1



In the above graph, V is a vertex for which it has an edge (V, V) forming a loop.

### Example 2



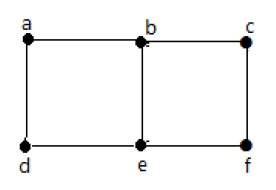
In this graph, there are two loops which are formed at vertex a, and vertex b.

### Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called neighbours or adjacent nodes.

* In a graph, two vertices are said to be **adjacent,** if there is an edge between the two vertices. Here, the adjacency of vertices is maintained by the single edge that is connecting those two vertices.
* In a graph, two edges are said to be adjacent, if there is a common vertex between the two edges. Here, the adjacency of edges is maintained by the single vertex that is connecting two edges.

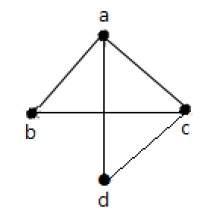
### Example 1



In the above graph −

* ‘a’ and ‘b’ are the adjacent vertices, as there is a common edge ‘ab’ between them.
* ‘a’ and ‘d’ are the adjacent vertices, as there is a common edge ‘ad’ between them.
* ab’ and ‘be’ are the adjacent edges, as there is a common vertex ‘b’ between them.
* be’ and ‘de’ are the adjacent edges, as there is a common vertex ‘e’ between them.

### Example 2



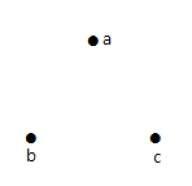
In the above graph −

* a’ and ‘d’ are the adjacent vertices, as there is a common edge ‘ad’ between them.
* ‘c’ and ‘b’ are the adjacent vertices, as there is a common edge ‘cb’ between them.
* ‘ad’ and ‘cd’ are the adjacent edges, as there is a common vertex ‘d’ between them.
* ac’ and ‘cd’ are the adjacent edges, as there is a common vertex ‘c’ between them.

## Null Graph

A **graph having no edges** is called a Null Graph.

### Example



In the above graph, there are three vertices named ‘a’, ‘b’, and ‘c’, but there are no edges among them. Hence it is a Null Graph.

## Trivial Graph

A **graph with only one vertex** is called a Trivial Graph.

### Example



In the above shown graph, there is only one vertex ‘a’ with no other edges. Hence it is a Trivial graph.

### Degree of the Node

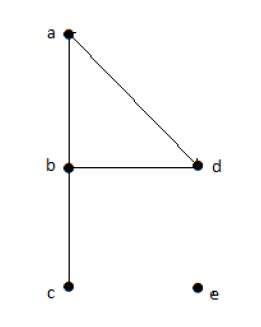
A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node. It is the number of vertices adjacent to a vertex V.

It can be represented as deg(V). In a simple graph with n number of vertices, the degree of any vertices is:

**Degree representation for undirected graph:**

### Example 1

Take a look at the following graph −



In the above Undirected Graph,

* deg(a) = 2, as there are 2 edges meeting at vertex ‘a’.
* deg(b) = 3, as there are 3 edges meeting at vertex ‘b’.
* deg(c) = 1, as there is 1 edge formed at vertex ‘c’

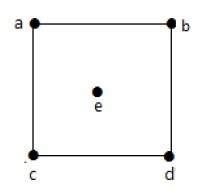
So ‘c’ is a **pendent vertex**.

* deg(d) = 2, as there are 2 edges meeting at vertex ‘d’.
* deg(e) = 0, as there are 0 edges formed at vertex ‘e’.

So ‘e’ is an **isolated vertex**.

### Example 2

Take a look at the following graph −



In the above graph,

deg(a) = 2, deg(b) = 2, deg(c) = 2, deg(d) = 2, and deg(e) = 0.

The vertex ‘e’ is an isolated vertex. The graph does not have any pendent vertex.

## Degree of Vertex in a Directed Graph

In a directed graph, each vertex has an **indegree** and an **outdegree**.

### Indegree of a Graph

* Indegree of vertex V is the number of edges which are coming into the vertex V.
* **Notation** − deg−(V).

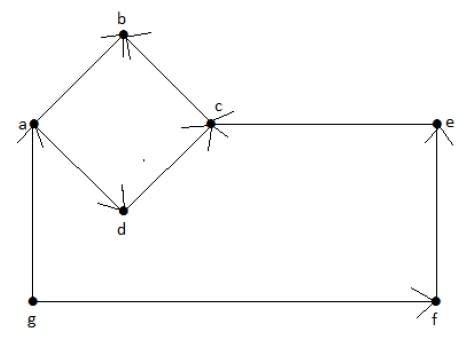
### Outdegree of a Graph

* Outdegree of vertex V is the number of edges which are going out from the vertex V.
* **Notation** − deg+(V).

Consider the following examples.

### Example 1

Take a look at the following directed graph. Vertex ‘a’ has two edges, ‘ad’ and ‘ab’, which are going outwards. Hence its outdegree is 2. Similarly, there is an edge ‘ga’, coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.

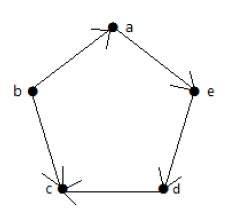


The indegree and outdegree of other vertices are shown in the following table −

| **Vertex** | **Indegree** | **Outdegree** |
| --- | --- | --- |
| a | 1 | 2 |
| b | 2 | 0 |
| c | 2 | 1 |
| d | 1 | 1 |
| e | 1 | 1 |
| f | 1 | 1 |
| g | 0 | 2 |

### Example 2

Take a look at the following directed graph. Vertex ‘a’ has an edge ‘ae’ going outwards from vertex ‘a’. Hence its outdegree is 1. Similarly, the graph has an edge ‘ba’ coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.

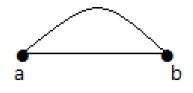


The indegree and outdegree of other vertices are shown in the following table −

| **Vertex** | **Indegree** | **Outdegree** |
| --- | --- | --- |
| a | 1 | 1 |
| b | 0 | 2 |
| c | 2 | 0 |
| d | 1 | 1 |
| e | 1 | 1 |

## Parallel Edges

In a graph, if a pair of vertices is connected by more than one edge, then those edges are called parallel edges.

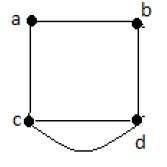


In the above graph, ‘a’ and ‘b’ are the two vertices which are connected by two edges ‘ab’ and ‘ab’ between them. So it is called as a parallel edge.

## Multi Graph

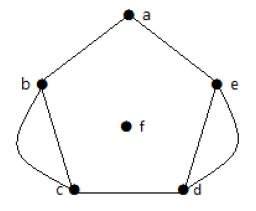
A graph having parallel edges is known as a Multigraph.

### Example 1



In the above graph, there are five edges ‘ab’, ‘ac’, ‘cd’, ‘cd’, and ‘bd’. Since ‘c’ and ‘d’ have two parallel edges between them, it a Multigraph.

### Example 2



In the above graph, the vertices ‘b’ and ‘c’ have two edges. The vertices ‘e’ and ‘d’ also have two edges between them. Hence it is a Multigraph.

Graphs Representation and its operations

Graphs are generally represented in the following scheme as,

* Incidence matrix
* Adjacency matrix
* Adjacency list
* **Incidence matrix**

A graph containing ***m*** vertices and ***n*** edges can be represented by a matrix with ***m*** rows and ***n*** columns. The matrix is formed by storing 1 in its ith row and jth column corresponding to the matrix, if there exists a ith vertex, connected to one end of the jth edge, and a 0 , if there is no ith vertex, connected to any end of the jth edge of the graph.

i.e

incidence\_matrix[i][j]=1, if there is and edge Ej from vertex Vi

= 0 , otherwise.

* Below are example graphs and their incidence matrix

|  | |  | E1 | E2 | E3 | | --- | --- | --- | --- | | V1 | 1 | 0 | 0 | | V2 | 0 | 1 | 0 | | V3 | 0 | 0 | 1 | | V4 | 1 | 1 | 1 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | E1 | E2 | E3 | E4 | | --- | --- | --- | --- | --- | | V1 | 1 | 1 | 0 | 0 | | V2 | 0 | 0 | 1 | 1 | | V3 | 1 | 0 | 1 | 0 | | V4 | 0 | 1 | 0 | 1 | |
|  | |  | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | V1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | | V2 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | | V3 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | | V4 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | | V5 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | |

**Adjacency matrix**

A graph containing ***n*** vertices can be represented by a matrix with ***n*** rows and ***n*** columns. The matrix is formed by storing the edge weight in its ith row and jth column of the matrix, if there exists an edge between ith and jth vertex of teh graph, and a 0 , if there is no edge between ith and jth vertex of the graph.

i.e

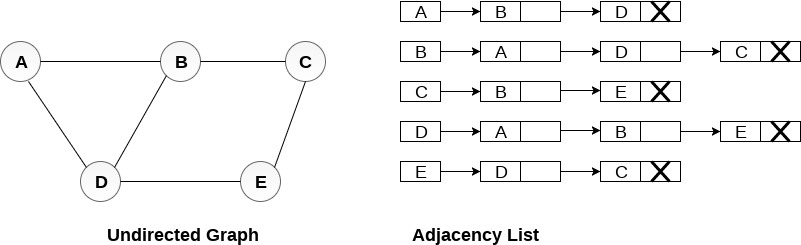
adjacency\_matrix[i][j]=weight of the edge, if there is a path from vertex Vi to Vj

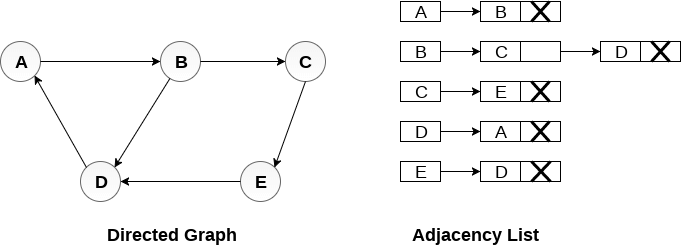
= 0 , otherwise.

|  | |  | V1 | V2 | V3 | V4 | | --- | --- | --- | --- | --- | | V1 | 0 | 0 | 0 | 1 | | V2 | 0 | 0 | 0 | 1 | | V3 | 0 | 0 | 0 | 1 | | V4 | 1 | 1 | 1 | 0 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | V1 | V2 | V3 | V4 | | --- | --- | --- | --- | --- | | V1 | 0 | 0 | 1 | 1 | | V2 | 0 | 0 | 1 | 1 | | V3 | 1 | 1 | 0 | 0 | | V4 | 1 | 1 | 0 | 0 | |
|  | |  | V1 | V2 | V3 | V4 | V5 | | --- | --- | --- | --- | --- | --- | | V1 | 0 | 1 | 0 | 1 | 1 | | V2 | 1 | 0 | 1 | 1 | 0 | | V3 | 0 | 1 | 0 | 1 | 1 | | V4 | 1 | 1 | 1 | 0 | 1 | | V5 | 1 | 0 | 1 | 1 | 0 | |

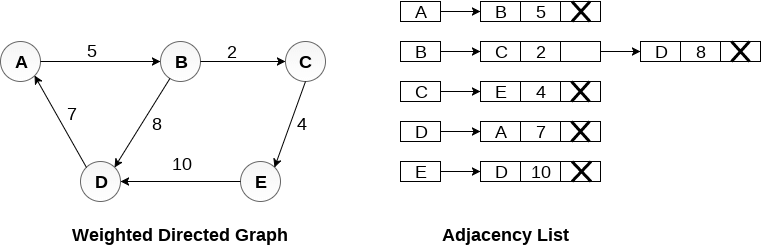
Adjacency List:

A graph containing ***m*** vertices and ***n*** edges can be represented using a linked list, referred to as adjacency list. The number of vertices in a graph forms a singly linked list. Each vertex have a seperated linked list, with nodes equal to the number of edges connected from the corresponding vertex.





In the case of weighted directed graphs, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



**Graph Traversals**

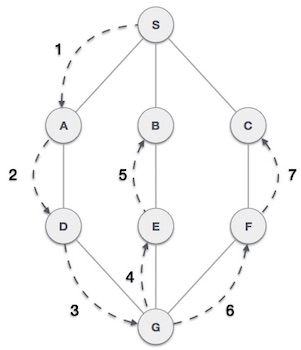
Traversing a graph means visiting all the nodes in the graph. The two important graph traversal methods are:

depth-first traversal (or) Depth-first search (DFS)

Breadth-first traversal (or) Breadth-first search(BFS)

**Depth-first traversal**

The logic of DFS is similar to preorder traversal of a tree. It traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S -> A -> D -> G -> E -> B first, then to F and lastly to C. It employs the following rules.

* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
* **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
* **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

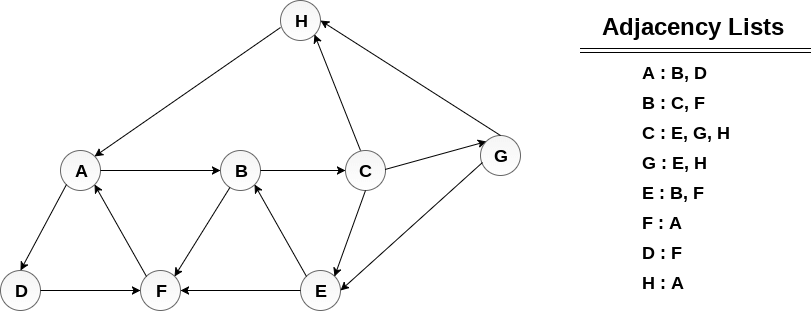
| **Step** | **Traversal** | **Description** |
| --- | --- | --- |
| 1 |  | Initialize the stack. |
| 2 |  | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. |
| 3 |  | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only. |
| 4 |  | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. |
| 5 |  | We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack. |
| 6 |  | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. |
| 7 |  | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Example:

### Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



### Solution :

Push H onto the stack

1. STACK:H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are in

| H |
| --- |

ready state.

1. Print H
2. STACK:A

| A |
| --- |

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

1. Print A
2. Stack:B,D

| D |
| --- |
| B |

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

1. Print D
2. Stack:B,F

| F |
| --- |
| B |

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state. F neighbour is A , already visited. so the next value is popped which is B

1. Print F
2. Stack:B

Pop the top of the stack i.e. B and push all the neighbours as F is already visited, it is not pushed

1. Print B
2. Stack: C

|  |
| --- |
| C |

Pop the top of the stack i.e. C and push all the neighbours.

1. Print C
2. Stack:E,G

| G |
| --- |
| E |

Pop the top of the stack i.e. G and push all its neighbours.

1. Print G
2. Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

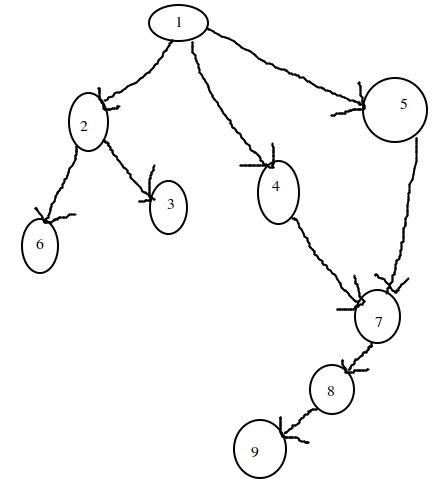
1. Print E
2. Stack:

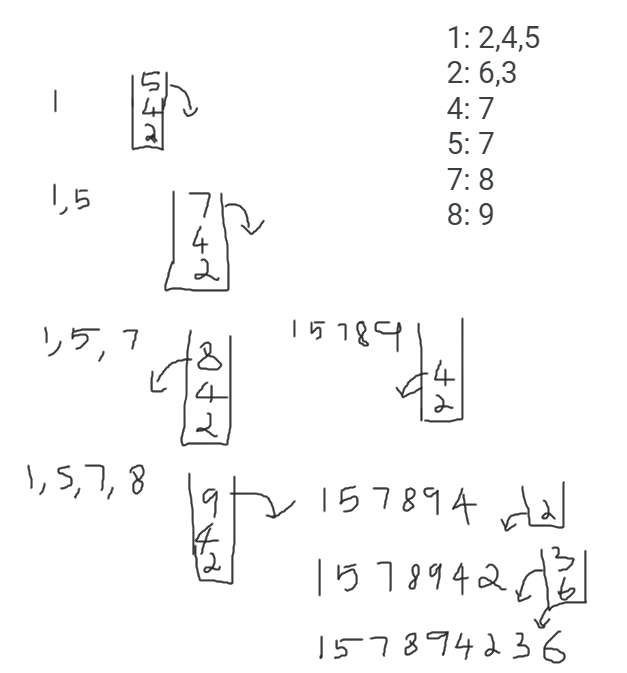
Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

1. H→A→D→F→B→C→G→E

example 2





Depth first program

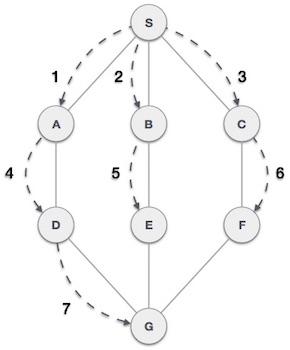
| # using python dictionary to act as an adjacency list  graph={  '5':['3','7'],  '3':['2','4'],  '7':['8'],  '2':[],  '4':['8'],  '8':[]  }  # set to keep track of visited nodes of graph  visited=set()  def dfs(visisted,graph,node):  if node not in visited:  print(node,end=" -> ")  visited.add(node)  for neighbour in graph[node]:  dfs(visited,graph,neighbour)  print("following is the DFS")  dfs(visited,graph,'5') |
| --- |

output

following is the DFS

5 -> 3 -> 2 -> 4 -> 8 -> 7 ->

**Breadth First Search (BFS)** algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to C to D first then to E and F lastly to G. It employs the following rules.

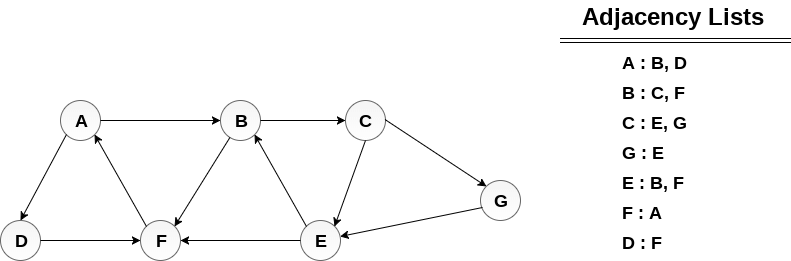
* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
* **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
* **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

| **Step** | **Traversal** | **Description** |
| --- | --- | --- |
| 1 |  | Initialize the queue. |
| 2 |  | We start from visiting **S** (starting node), and mark it as visited. |
| 3 |  | We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it. |
| 4 |  | Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it. |
| 5 |  | Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it. |
| 6 |  | Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**. |
| 7 |  | From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it. |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

### Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

**Lets start examining the graph from Node A.**

1. Add A to QUEUE1 and NULL to QUEUE2.

1. QUEUE1={A}
2. QUEUE2={NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

1. QUEUE1={B, D}
2. QUEUE2={A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

1. QUEUE1={D,C,F}
2. QUEUE2={A,B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

1. QUEUE1={C,F}
2. QUEUE2={A,B,D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

1. QUEUE1={F,E,G}
2. QUEUE2={A,B,D,C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

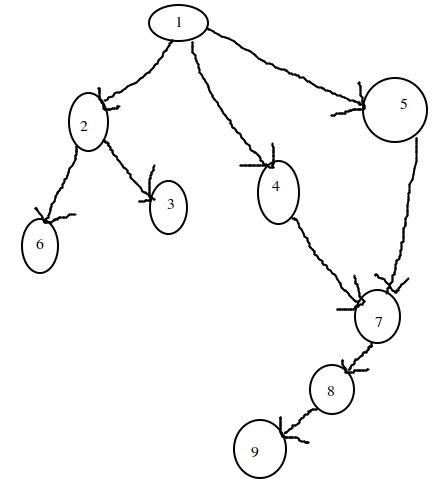
1. QUEUE1={E,G}
2. QUEUE2={A,B,D,C,F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

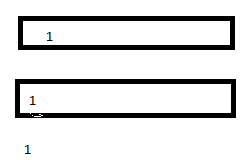
1. QUEUE1={G}
2. QUEUE2={A,B,D,C,F,E}

Now, backtrack from E to A, using the nodes available in QUEUE2.

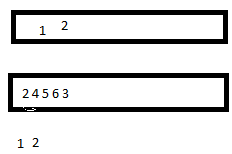
The minimum path will be **A → B → C → E**.



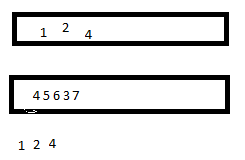
step 1



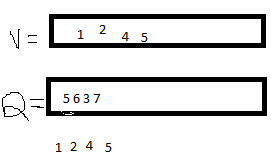
step 2



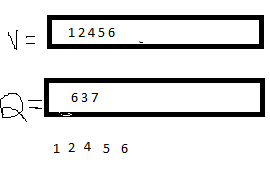
step 3



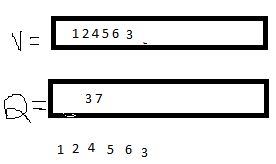
step 4



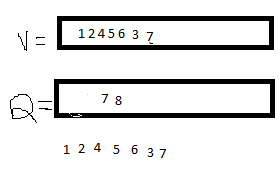
step 5



step 6



step 7



step 8



| #include <iostream>  using namespace std;  int adjacency[10][10], i,j,k,n,queue[10],front,rear,v,visit[10],visited[10];  int main()  {  int m;  cout<<"enter no of vertices : ";  cin>>n;  cout<<"enter no of edges : ";  cin>>m;  cout<<"\n Edges \n";  for(k=1;k<=m;k++)  {  cout<<"enter from vertex : ";  cin>>i;  cout<<"enter to vertex : ";  cin>>j;  adjacency[i][j]=1;  }  cout<<"adjacency matrix is "<<endl;  cout<<" 1 2 3 4 5 6 7 8 9"<<endl;  for(i=1;i<=m;i++)  {  cout<<i<<" ";  for(j=1;j<=n;j++)  {  cout<<adjacency[i][j]<<" ";  }    cout<<endl;  }  cout<<"enter inital vertex to traverse from : ";  cin>>v;  cout<<"BFS order of visited vertices : ";  cout<<v<<"-> ";    visited[v]=1;  k=1;  while(k<n)  {  for(j=1;j<=n;j++)  {  if(adjacency[v][j]!=0 && visited[j]!=1 && visit[j]!=1)  {  visit[j]=1;  queue[rear++]=j;  }  }  v=queue[front++];  cout<<v<<"->";  k++;  visit[v]=0;  visited[v]=1;  }  } |
| --- |

enter no of vertices : 9

enter no of edges : 9

Edges

enter from vertex : 1

enter to vertex : 2

enter from vertex : 2

enter to vertex : 3

enter from vertex : 1

enter to vertex : 5

enter from vertex : 1

enter to vertex : 4

enter from vertex : 4

enter to vertex : 7

enter from vertex : 7

enter to vertex : 8

enter from vertex : 8

enter to vertex : 9

enter from vertex : 2

enter to vertex : 6

enter from vertex : 5

enter to vertex : 7

adjacency matrix is

1 2 3 4 5 6 7 8 9

1 0 1 0 1 1 0 0 0 0

2 0 0 1 0 0 1 0 0 0

3 0 0 0 0 0 0 0 0 0

4 0 0 0 0 0 0 1 0 0

5 0 0 0 0 0 0 1 0 0

6 0 0 0 0 0 0 0 0 0

7 0 0 0 0 0 0 0 1 0

8 0 0 0 0 0 0 0 0 1

9 0 0 0 0 0 0 0 0 0

enter inital vertex to traverse from : 1

BFS order of visited vertices : 1-> 2->4->5->3->6->7->8->9->

------------------

(program exited with code: 0)

Press return to continue

Hashing

<https://www.tutorialride.com/data-structures/hashing-in-data-structure.htm>