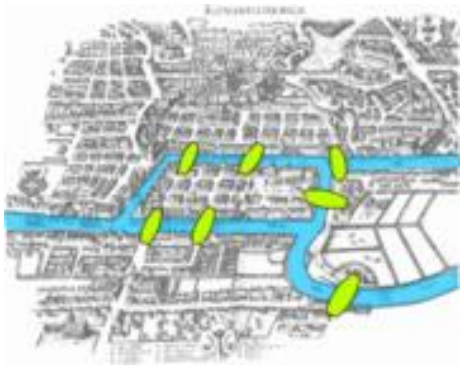
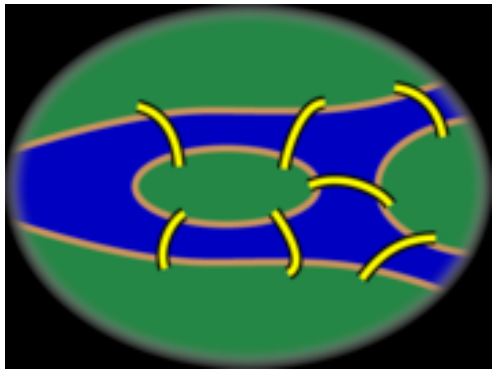
# Introduction to Graphs

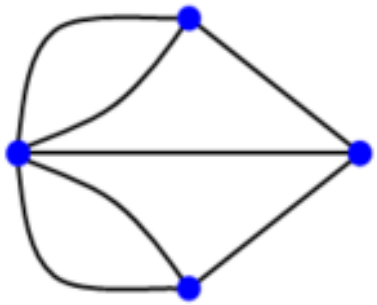
Lecture 14

# Graph Theory - History

Leonhard Euler's paper on "*Seven Bridges of Königsberg*" , in 1736.

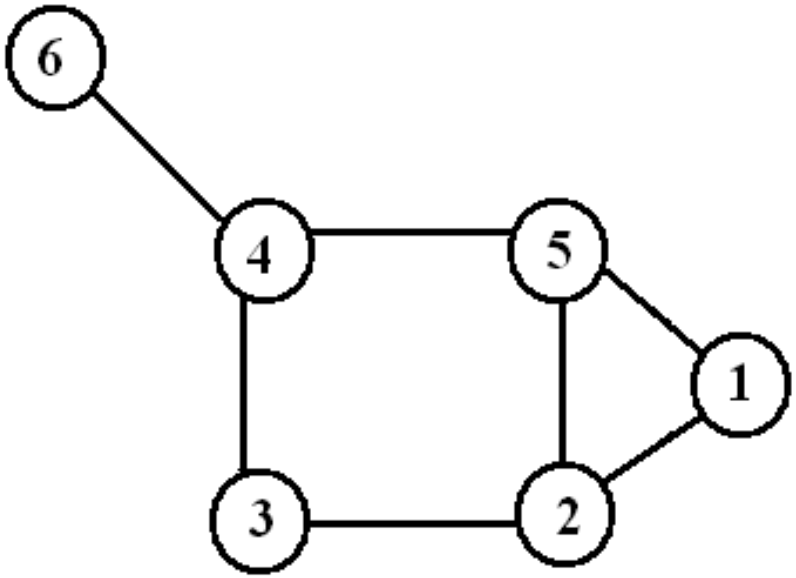






# Graphs

- A ***graph*** $G = (V, E)$ is:
  - Collection of **nodes** called ***vertices***
  - **Connections** between them, called ***edges***




**V**:={1,2,3,4,5,6}

**E**:={{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

- **Vertex**
  - Basic Element
  - Drawn as a *node* or a *dot*.
  - V**ertex set** of *G* is usually denoted by *V*(*G*), or *V*
- **Edge**
  - A set of two elements
  - Drawn as a line connecting two vertices, called end vertices, or endpoints
  - The edge set of *G* is usually denoted by E(G), or E

# Graph-based Representations

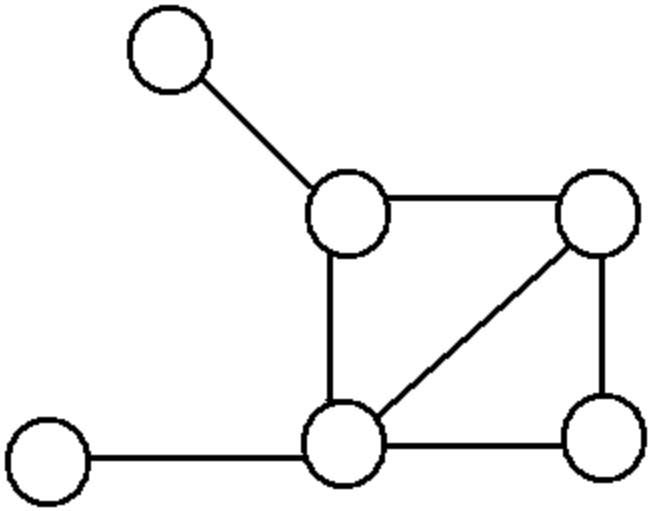
- Graphs represent the **relationships** among data items
- Representing a problem as a graph can provide a different point of view
- Representing a problem as a graph can make a problem much simpler
  - More accurately, it can provide the appropriate tools for solving the problem

# Graph Applications

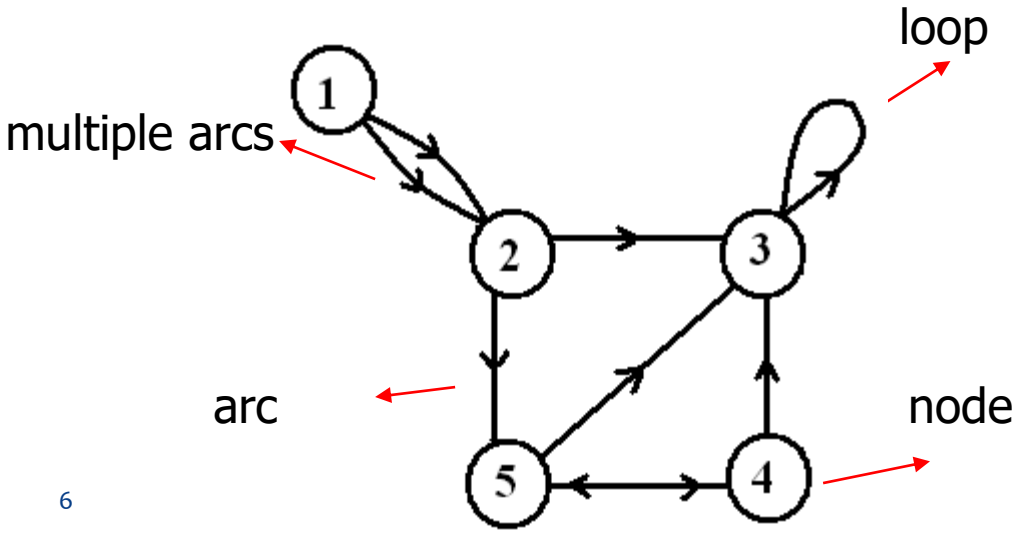
- Storing things that are graphs by nature
  - distance between cities
  - airline flights, travel options
  - relationships between people, things
  - distances between rooms in Clue
- Networks of communication
- Data organization
- Computational devices
- Flow of computation

# Simple & Directed Graphs

- *Simple graphs* are graphs without multiple edges or self-loops

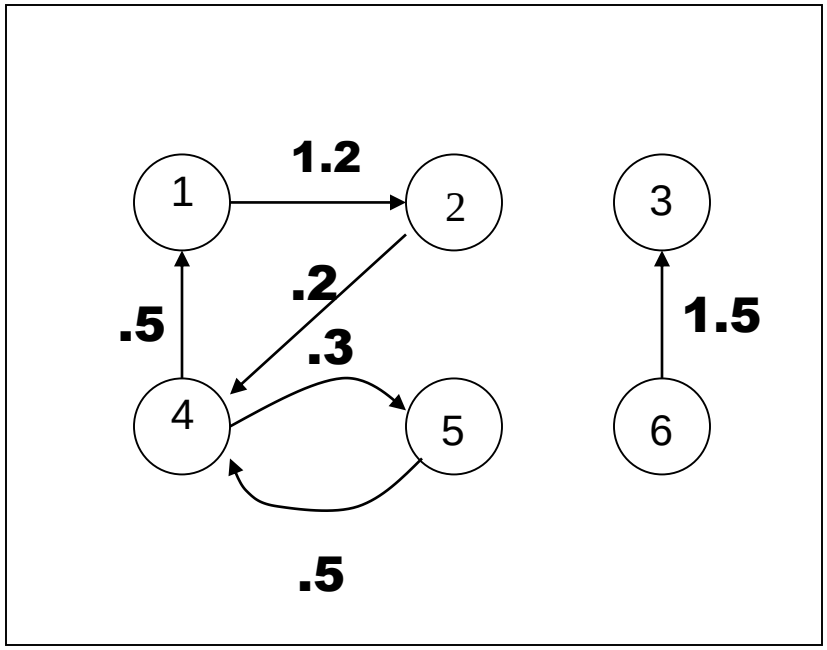


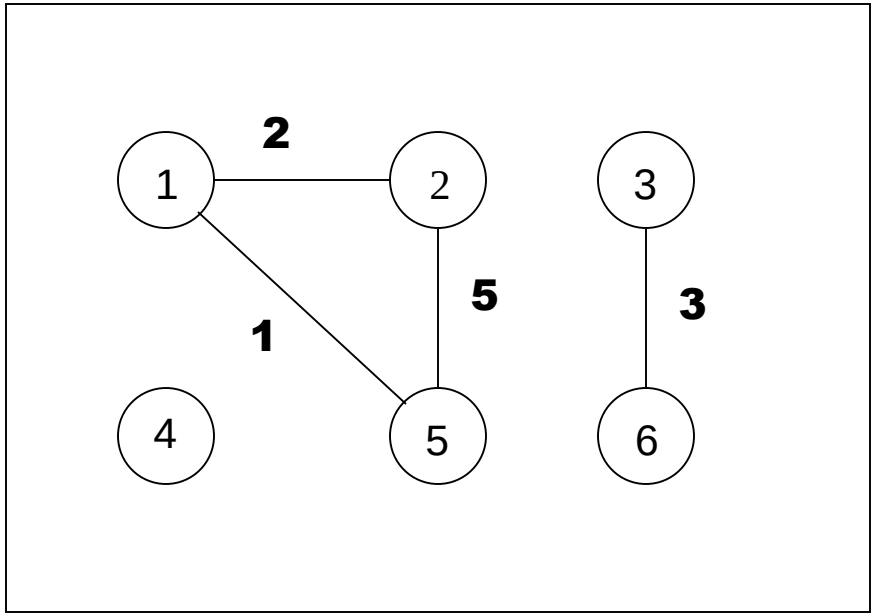
- Edges have directions
  - An edge is an *ordered* pair of nodes



# Weighted graphs

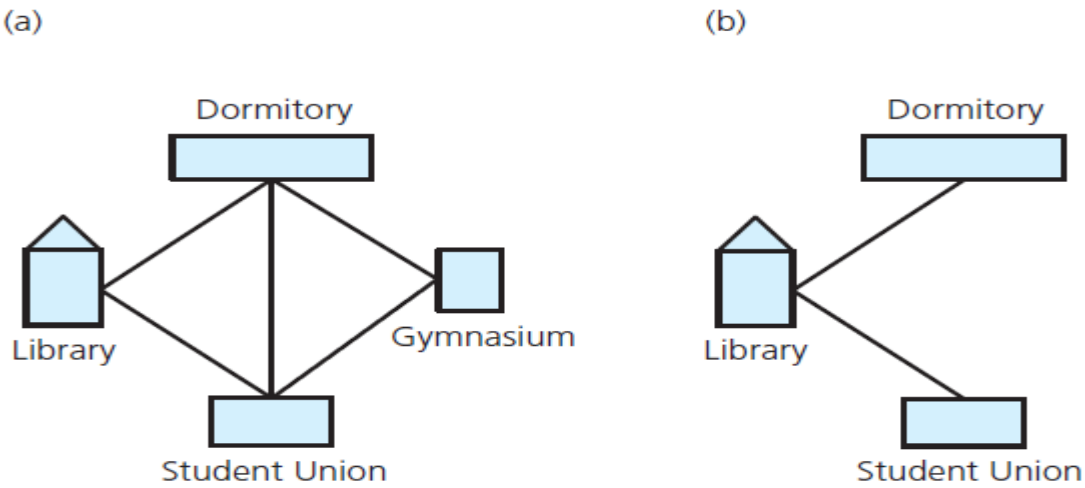
- A graph for which each edge has an associated ***weight***, usually given by a ***weight function*** *w: E* $\rightarrow$ **R**



# Subgraph

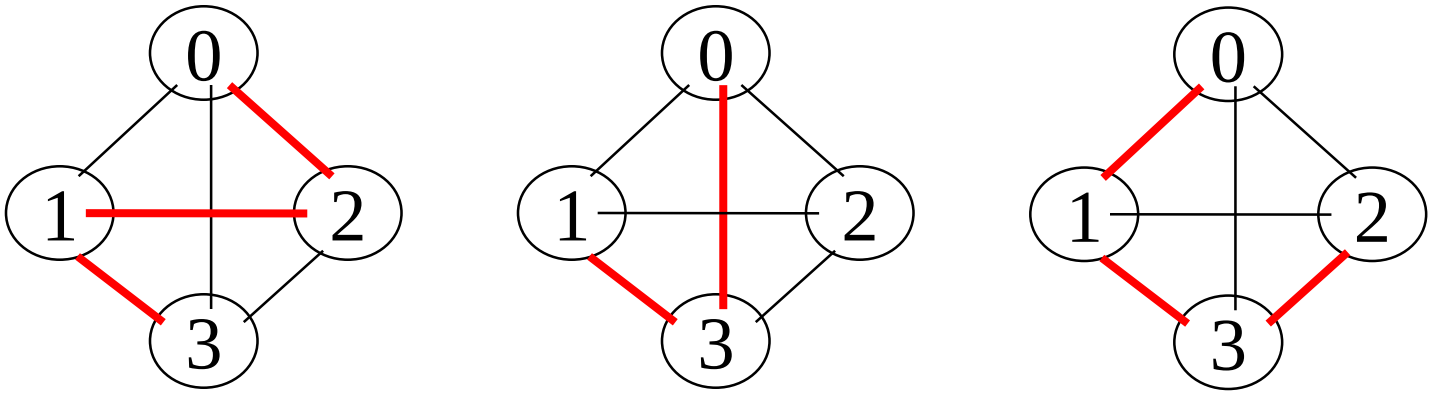
- G = { V , E }; that is, a graph is a set of vertices and edges
- A subgraph consists of a subset of a graph's vertices and a subset of its edges




(a) A campus map as a graph
(b) A subgraph

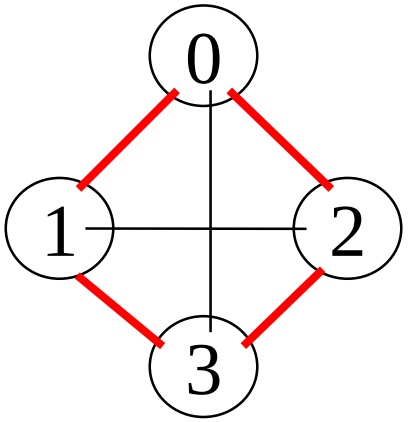
# Terminology

- **Path**
  - A **path** from vertex *vp* to vertex *vq* in a graph *G*, is a sequence of vertices, *vp*, *vi*1, *vi*2, ..., *vi*n, vq, such that (*vp*, *vi*1), (*vi*1, *vi*2), ..., (*vi*n, v*q*) are edges in an undirected graph
    - A path such as **(0, 2), (2, 1), (1, 3)** is also written as **0, 2, 1, 3**
  - The **length of a path** is the number of edges on it



# Terminology

- **Simple path and cycle**
  - **Simple path** (simple directed path): a path in which all vertices, except possibly the first and the last, are distinct
  - A **cycle** is a simple path that **begins** and **ends** at same vertex

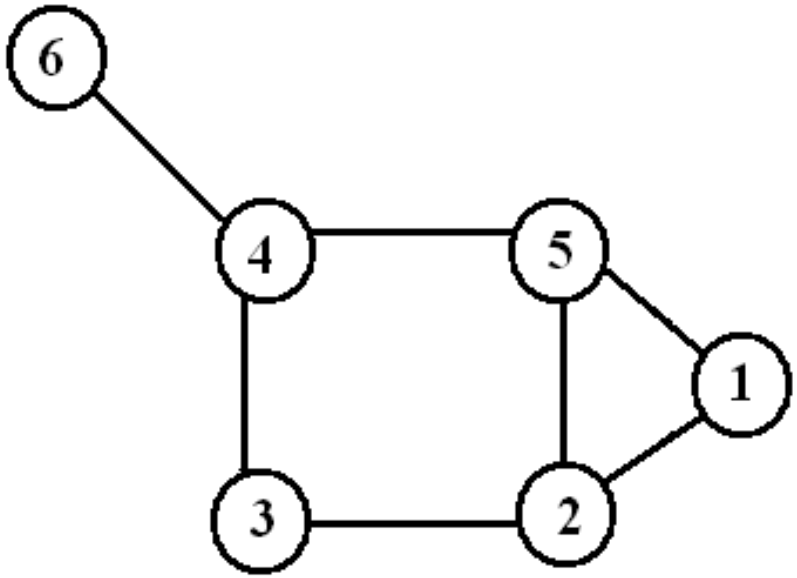


# Terminology


(a) **Connected**
(b) **Disconnected**
(c) **Complete**

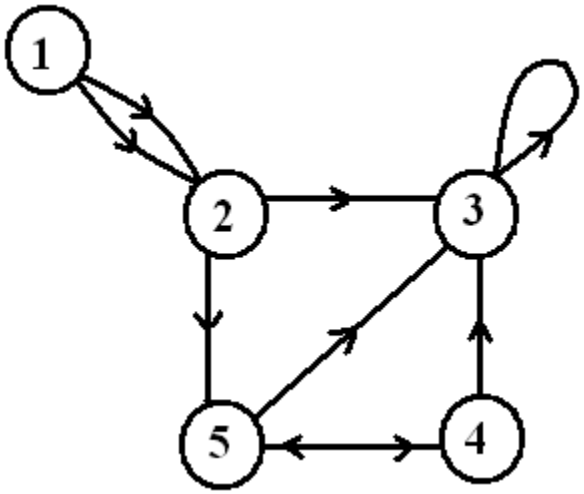
# Terminology

- **Degree**
  - Number of edges incident on a node




The degree of 5 is 3

- **Degree (Directed Graphs)**
  - **In-degree:** Number of edges entering
  - **Out-degree:** Number of edges leaving




outdeg(1)=2
indeg(1)=0

outdeg(2)=2
indeg(2)=2

outdeg(3)=1
indeg(3)=4

# Graph Representation

- Graphs represented in computer memory
  - Two common ways
    - **Adjacency matrices**
    - **Adjacency lists**

# Adjacency Matrix Representation

- A two-dimensional matrix or array that has one row and one column for each node in the graph
- For each edge of the graph ($V_i$, $V_j$), the location of the matrix at row *i* and column *j* is 1
- All other locations are 0

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 |

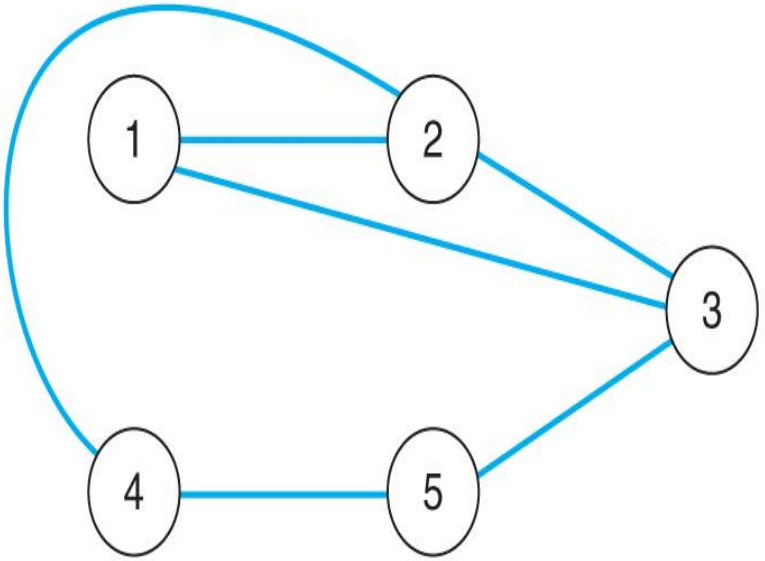
The adjacency matrix for the graph

# Adjacency Matrix Representation

- For an undirected graph, the matrix will be symmetric along the diagonal

- For a weighted graph, the adjacency matrix would have the weight for edges in the graph, zeros along the diagonal, and infinity (∞) every place else

# Adjacency Matrix Example 1

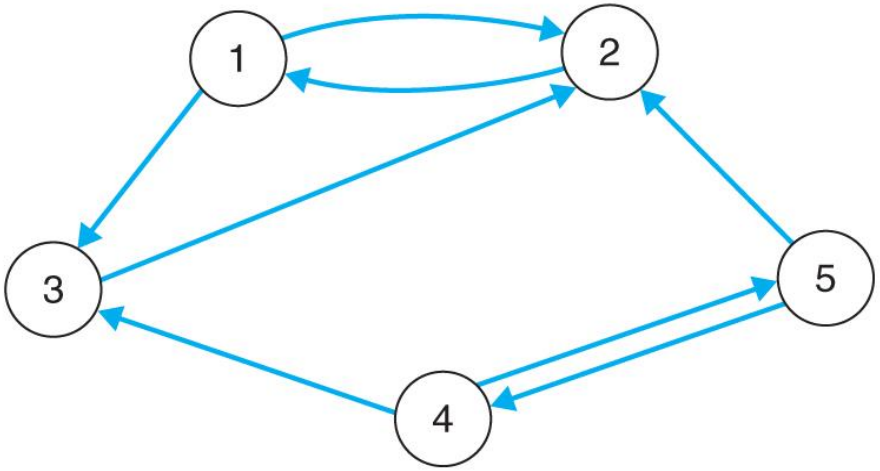



The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 |

The adjacency matrix for the graph

# Adjacency Matrix Example 2




The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

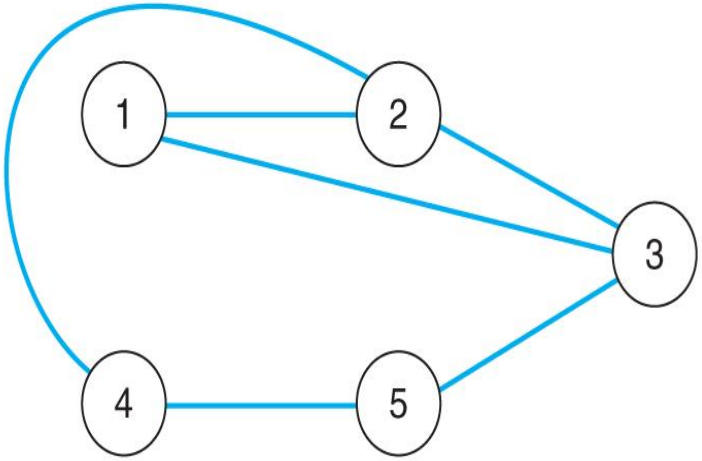
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 |

The adjacency matrix for the digraph

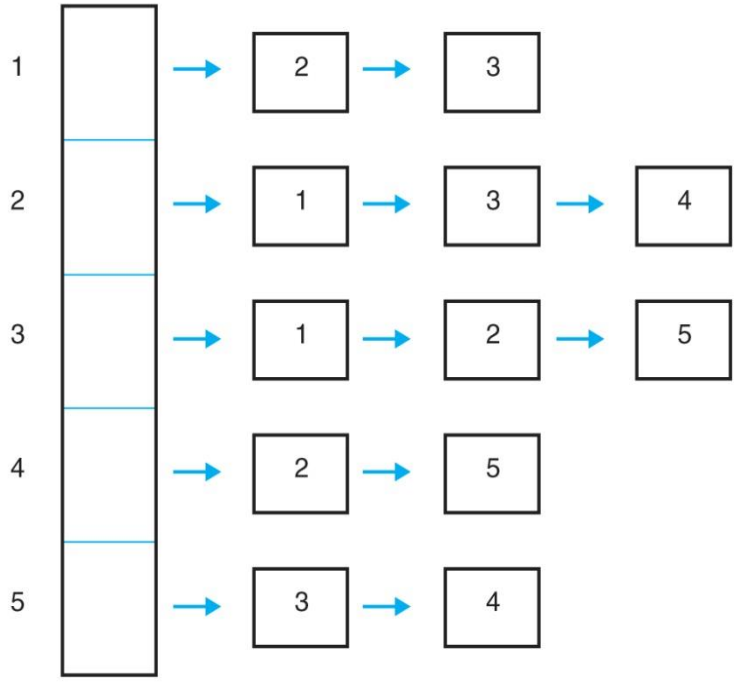
# Adjacency List

- A list of pointers, one for each node of the graph
- These pointers are the start of a linked list of nodes that can be reached by one edge of the graph
- For a weighted graph, this list would also include the weight for each edge

# Adjacency List Example 1

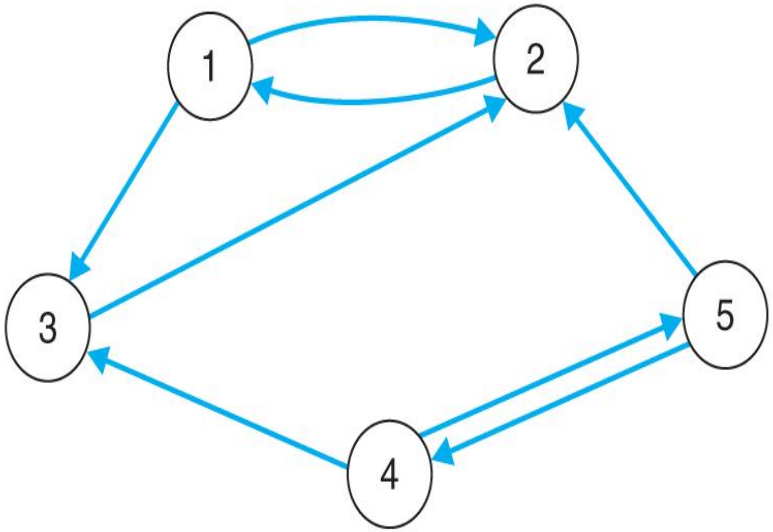



The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

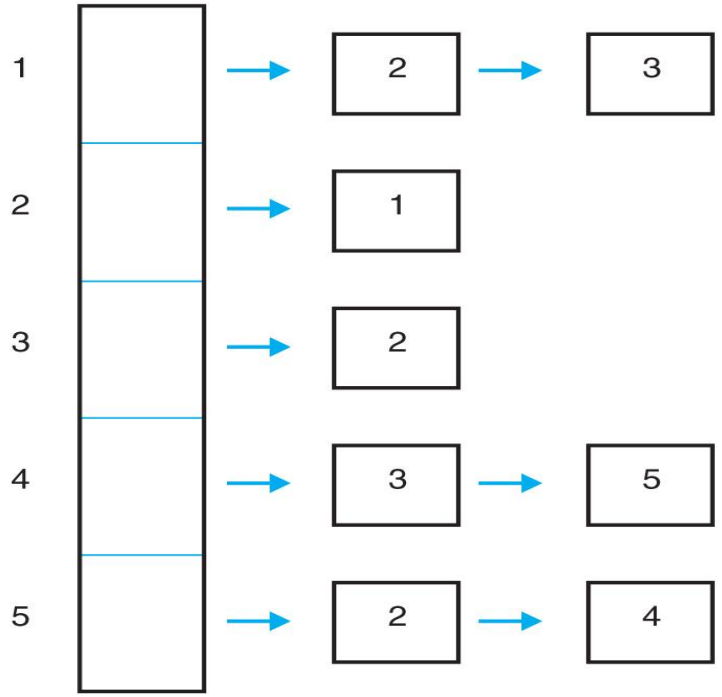



The adjacency list for the graph

# Adjacency List Example 2




The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})




The adjacency list 1

# Operations On Graph

- Creating a graph
- Traverse
- Search
- Delete

# Creating A Graph

- To create a graph, first adjacency list array is created to store the vertices name, dynamically at the run time. Then the node is created and linked to the list array if an edge is there to the vertex.

- **Step 1:** Input the total number of vertices in the graph, say n.
- **Step 2:** Allocate the memory dynamically for the vertices to store in list array.
- **Step 3:** Input the first vertex and the vertices through which it has edge(s) by linking the node from list array through nodes.
- **Step 4:** Repeat the process by incrementing the list array to add other vertices and edges.
- **Step 5:** Exit.

# Searching And Deletion From Graph

- Suppose an edge is to be deleted from the graph G. First we will search through the list array whether the initial vertex of the edge is in list array or not by incrementing the list array index. Once the initial vertex is found in the list array, the corresponding link list will be search for the terminal vertex.

***Step 1:*** Input an edge to be searched

***Step 2:*** Search for an initial vertex of edge in list arrays by incrementing the array index.

***Step 3:*** Once it is found, search through the link list for the terminal vertex of the edge.

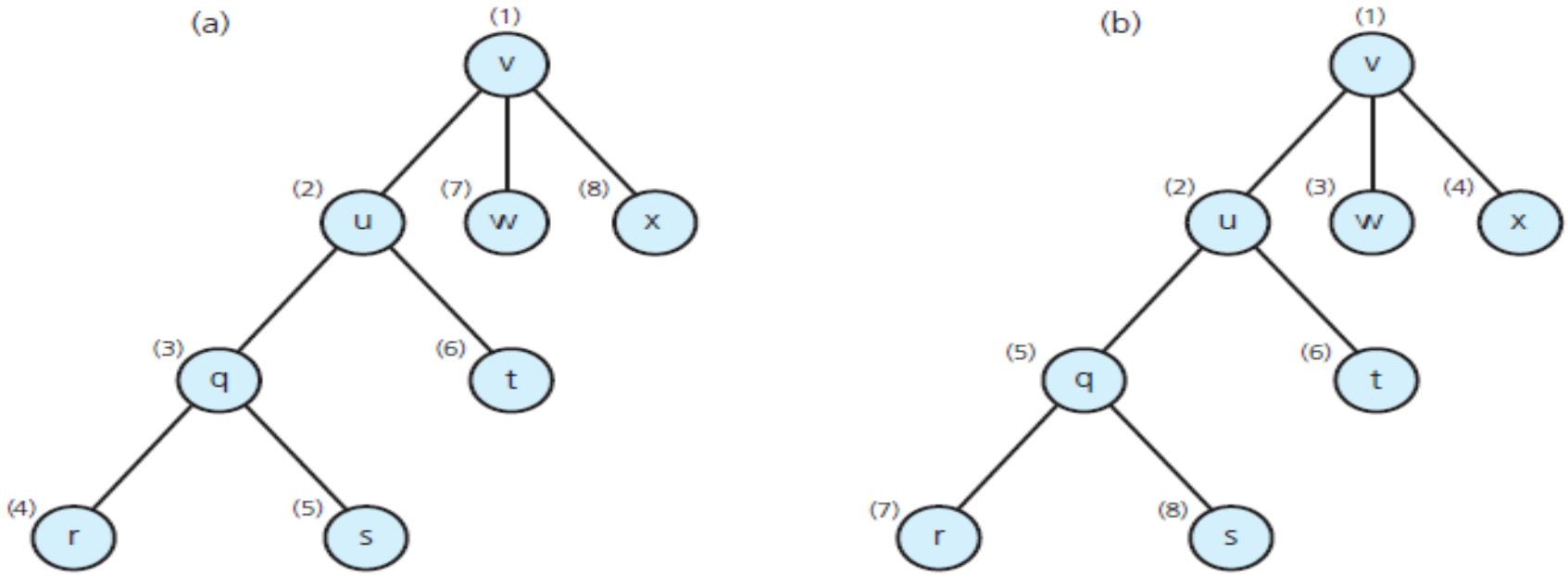
***Step 4:*** If found display "the edge is present in the graph".

***Step 5:*** Then delete the node where the terminal vertex is found and rearrange the link list.

***Step 6:*** Exit

# Graph Traversals

- Visits all of the vertices that it can reach
  - Happens if and only if graph is connected
  1) **Depth First Search (DFS): preorder traversal**
  2) **Breadth First Search (BFS): level order traversal**




Visitation order for (a) a depth-first search; (b) a breadth-first search

# Depth-First Traversal

- We follow a path through the graph until we reach a dead end
- We then back up until we reach a node with an edge to an unvisited node
- We take this edge and again follow it until we reach a dead end
- This process continues until we back up to the starting node and it has no edges to unvisited nodes

# Depth First Search (DFS): preorder traversal

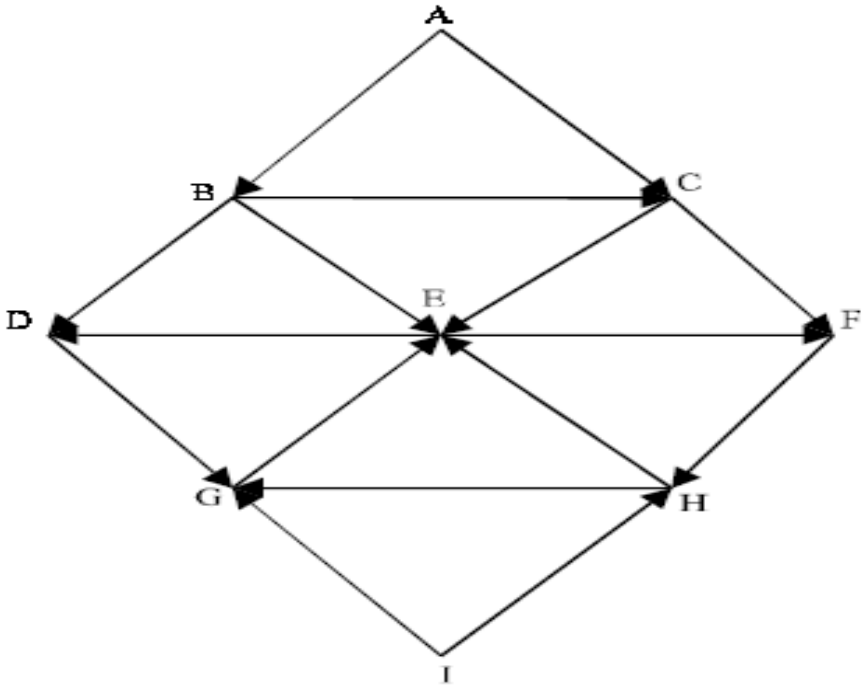
- General algorithm for depth first traversal at a given node *v*
  - Goes as far as possible from a vertex before backing up
  - Recursive algorithm

```
1. mark node v as visited
2. visit the node
3. for each vertex u adjacent to v
       if u is not visited
           start the depth first traversal at u
```

# Algorithm

1. Input the vertices and edges of the graph G = (V, E).
2. Input the source vertex and assign it to the variable S.
3. Push the source vertex to the stack.
4. Repeat the steps 5 and 6 until the stack is empty.
5. Pop the top element of the stack and display it.
6. Push the vertices which is neighbor to just popped element, if it is not in the queue and displayed (ie; not visited).
7. Exit

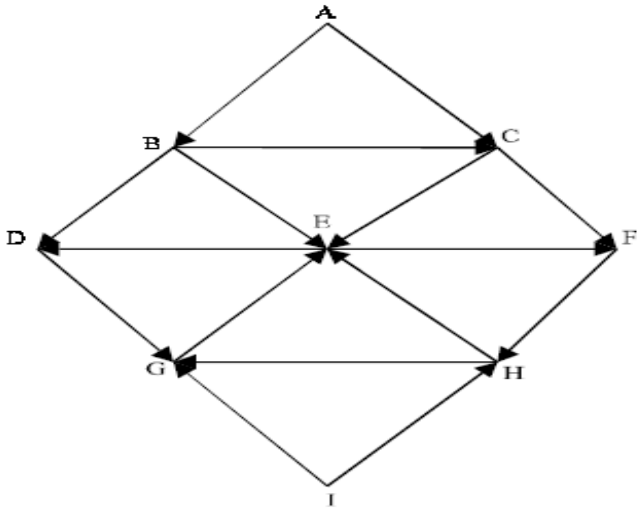
# Depth-First Traversal Example




| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

- Consider the graph in and its linked list representation. Suppose the **source** vertex is **I.**

# Depth-First Traversal Example




| Vertex | Adjacency list |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

***Step 1:*** *Initially* ***push I*** *on to the stack.*
***STACK:*** I
***DISPLAY:***

***Step 2:*** *Pop and display the top element, and then push all the neighbors of popped* element (*i.e., I) onto the stack, if it is not visited (or displayed or not in the stack.*
**STACK:** G, H
**DISPLAY:** I

# Depth-First Traversal Example

***Step 3: Pop and display the top element*** *and then* ***push all the neighbors*** *of popped* the element *(i.e., H) onto top of the stack, if it is not visited.*

**STACK:** G, E

**DISPLAY:** I, H

The popped element H has two neighbors E and G. G is **already visited**, means G is either **in the stack or displayed**. Here G is in the stack. So only E is pushed onto the top of the stack.

| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

***Step 4: Pop and display*** *the top element of the stack.*
***Push all the neighbors*** *of the* popped element on to the stack, if it is not visited.

**STACK:** G, D, F

**DISPLAY:** I, H, E

# Depth-First Traversal Example

***Step 5:*** *Pop and display the top element of the stack. Push all the neighbors of the* popped element onto the stack, if it is not visited.
**STACK:** G, D
**DISPLAY:** I, H, E, F
The popped element (or vertex) F has neighbor(s) H, which is already visited. Then H is displayed, and will not be pushed again on to the stack.

| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

***Step 6:*** *The process is repeated as:*
**STACK:** G
**DISPLAY:** I, H, E, F, D

# Depth-First Traversal Example

**STACK:** //now the stack is empty
**DISPLAY:** I, H, E, F, D, G
So **I, H, E, F, D, G** is the **DFS traversal** of graph from the source vertex I.

| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

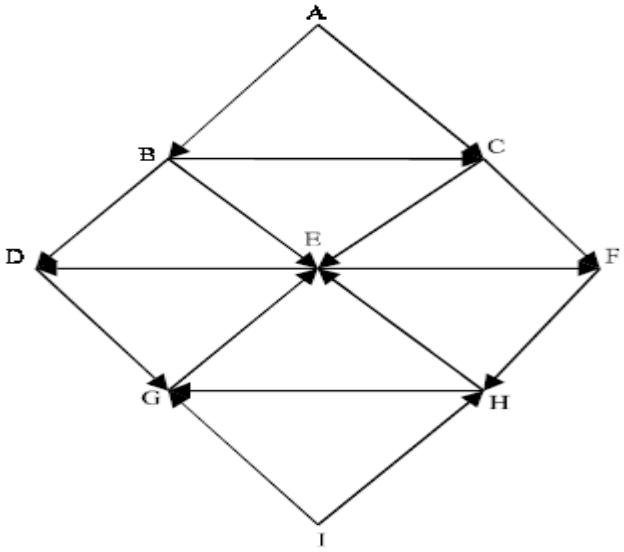
# Breadth First Traversal

- Visits all vertices adjacent to vertex before going forward
- Breadth-first search uses a queue

**ALGORITHM:**

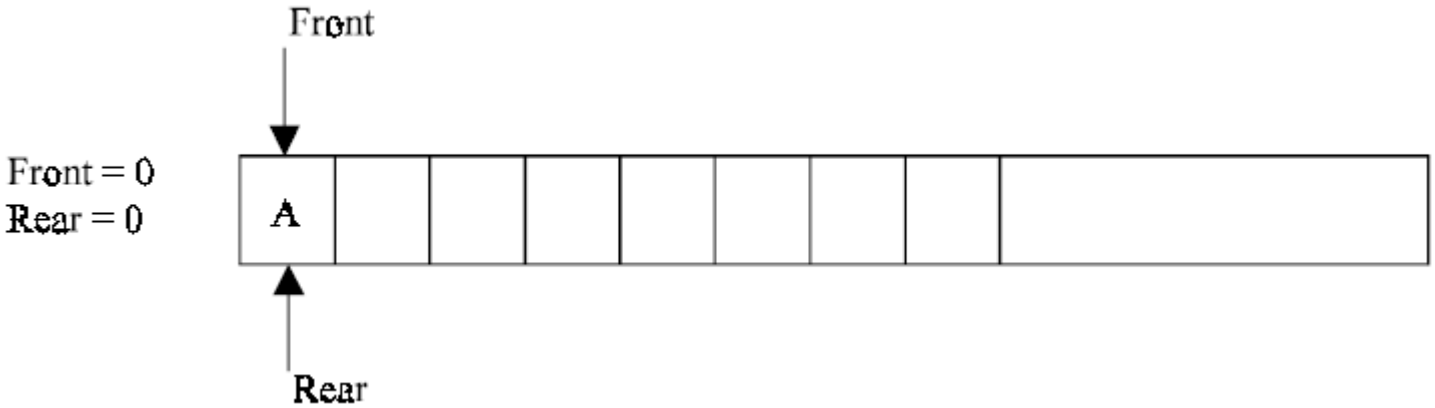
1. Input the vertices of the graph and its edges G = (V, E)
2. Input the source vertex and assign it to the variable S.
3. Add or push the source vertex to the queue.
4. Repeat the steps 5 and 6 until the queue is empty (*i.e., front > rear)*
5. Pop the front element of the queue and display it as visited.
6. Push the vertices, which is neighbor to just, popped element, if it is not in the queue and displayed (*i.e., not visited)*.
7. Exit.

# Breadth First Example




| Vertex | Adjacency list |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

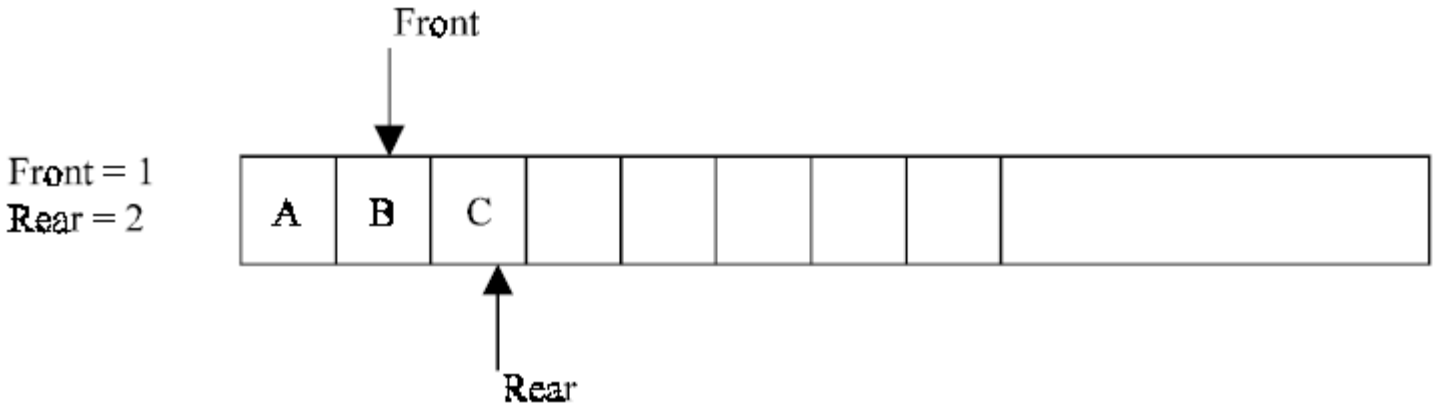
***Step 1:*** Initially **push A** (the source vertex) to the queue.



# Breadth First Example

| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

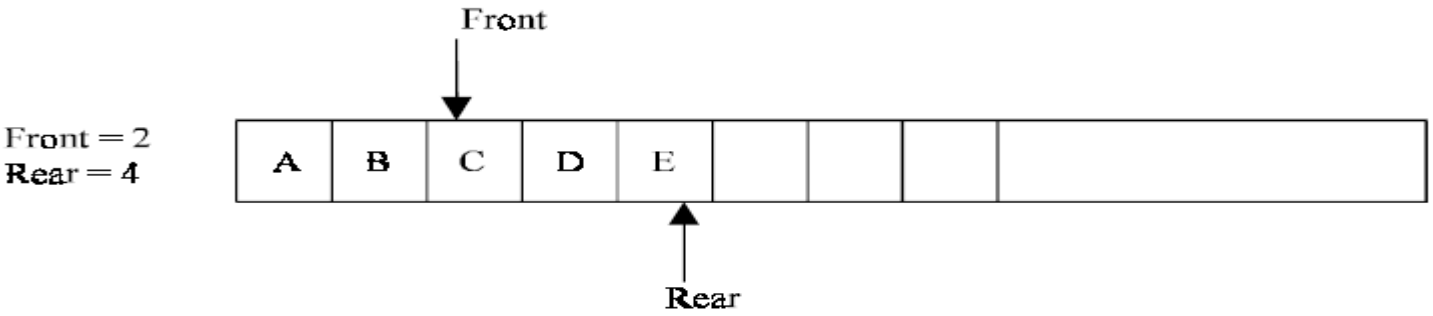
***Step 2:*** Pop (or remove) the front element A from the queue (by incrementing front = front +1) and display it. Then push (or add) the neighboring vertices of A to the queue, (by incrementing Rear = Rear +1) if it is not in queue.



# Breadth First Example

| Vertex | Adjacency list |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

***Step 3:*** Pop the front element B from the queue and display it. Then add the neighboring vertices of B to the queue, if it is not in queue.

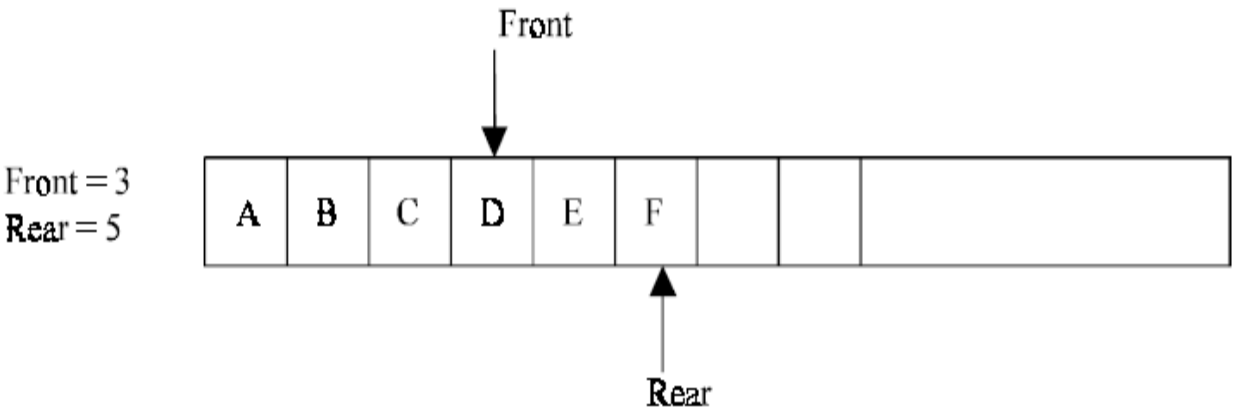



One of the neighboring element C of B is present in the queue, So C is not added to queue.

# Breadth First Example

| Vertex | Adjacency list |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

***Step 4:*** Remove the front element C and display it. Add the neighboring vertices of C, if it is not present in queue.

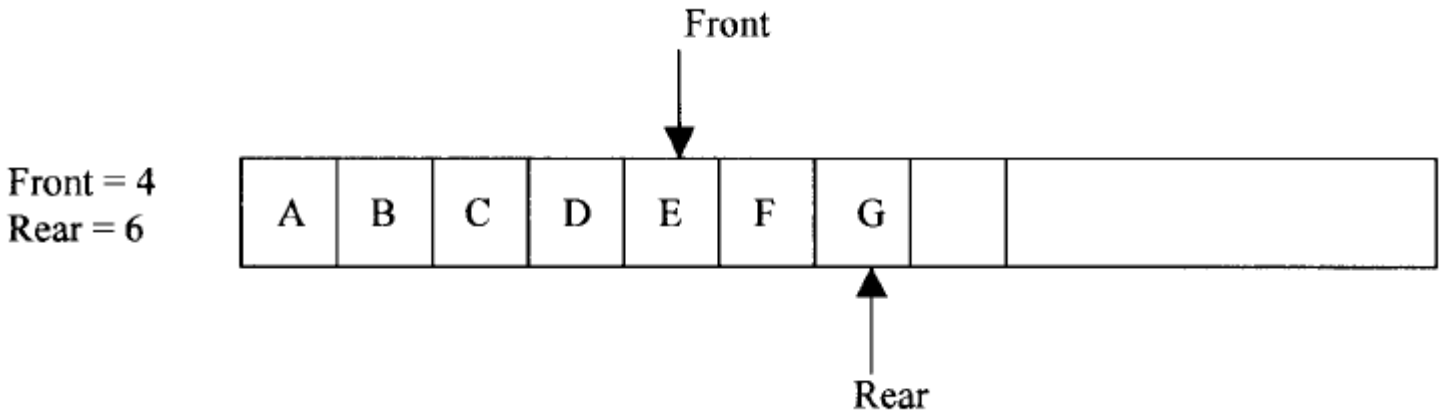


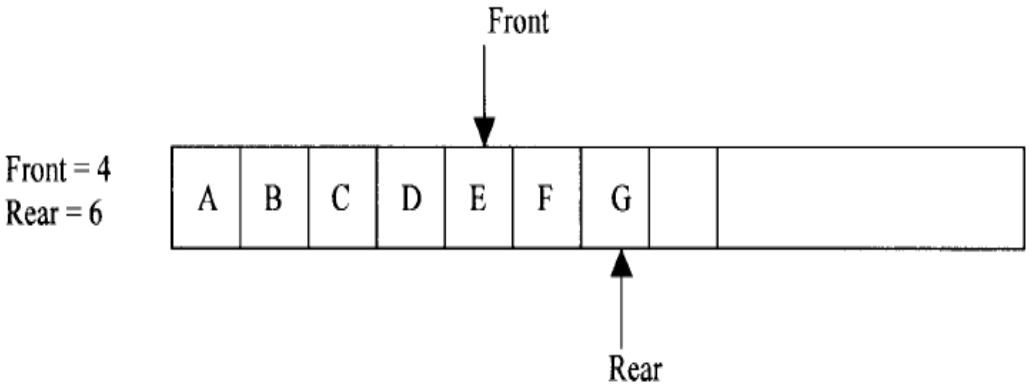

One of the neighboring elements E of C is present in the queue. So E is not added.

# Breadth First Example

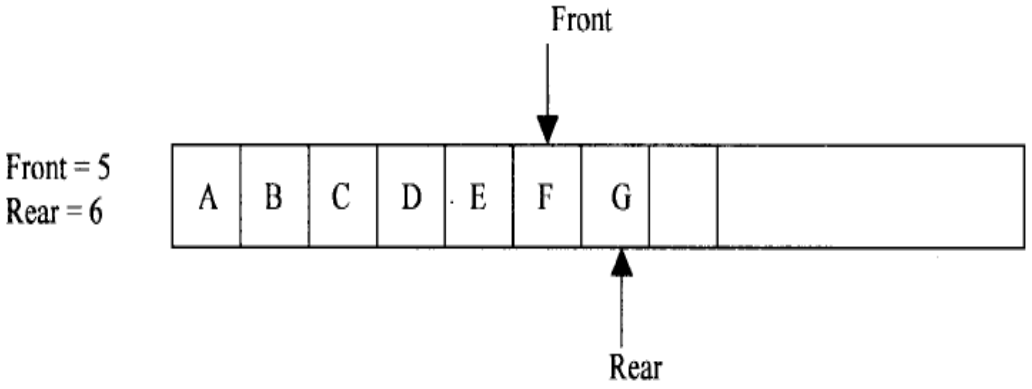
| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

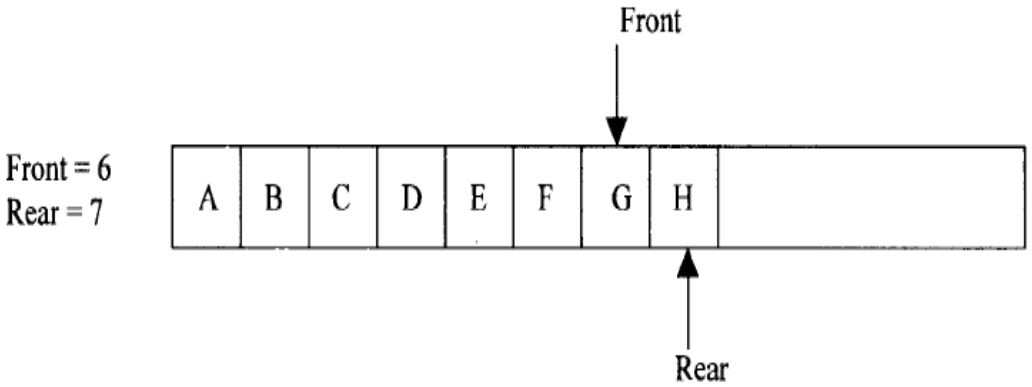
***Step 5:*** Remove the front element D, and add the neighboring vertex if it is not present in queue.





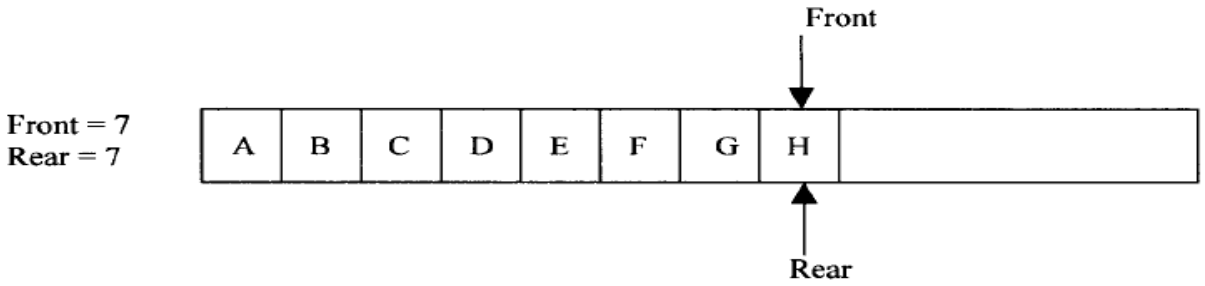

*Step 6:* Again the process is repeated (Until front > Rear). That is remove the front element E of the queue and add the neighboring vertex if it is not present in queue.

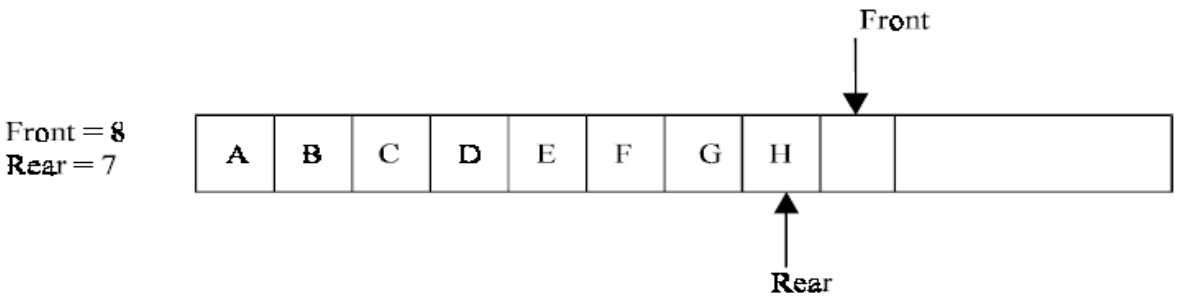



| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

# Breadth First Example




| *Vertex* | *Adjacency list* |
|---|---|
| A | B, C |
| B | C, D, E |
| C | E, F |
| D | G |
| E | D, F |
| F | H |
| G | E |
| H | E, G |
| I | G, H |

So **A, B, C, D, E, F, G, H** is the BFS traversal of the graph