ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN

VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY

한-베정보통신기술대학교

Nhân bản – Phụng sự – Khai phóng

Graphs

Data Structures & Algorithms

CONTENT

- Terminology
- Graph Representations
- Graph Traversals

CONTENT

- **Terminology**
- Graph Representations
- Graph Traversals

- **A graph G=(V,E), V and E are two sets**
  - V: finite non-empty set of vertices
  - E: set of edges (pairs of vertices)

- **Undirected graph**
  - The pair of vertices representing any edge is unordered.

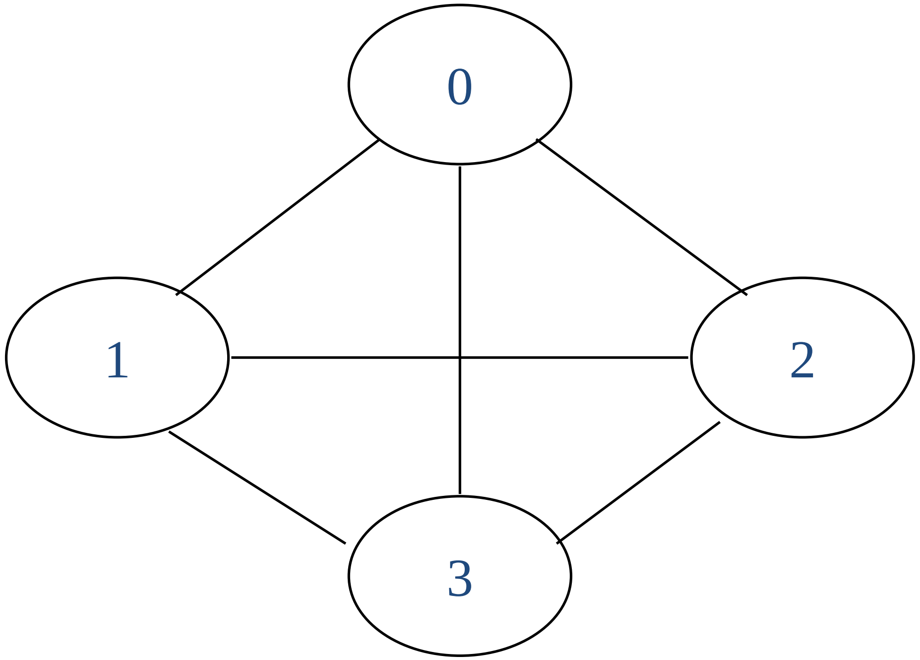
    Thus, the pairs (u,v) and (v,u) represent the same edge

- **Directed graph**
  - each edge is represented by an ordered pair <u,v>

- **Examples**

  Graph G1=(V,E):

  - V(G1)={0,1,2,3}
  - E(G1)={(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}



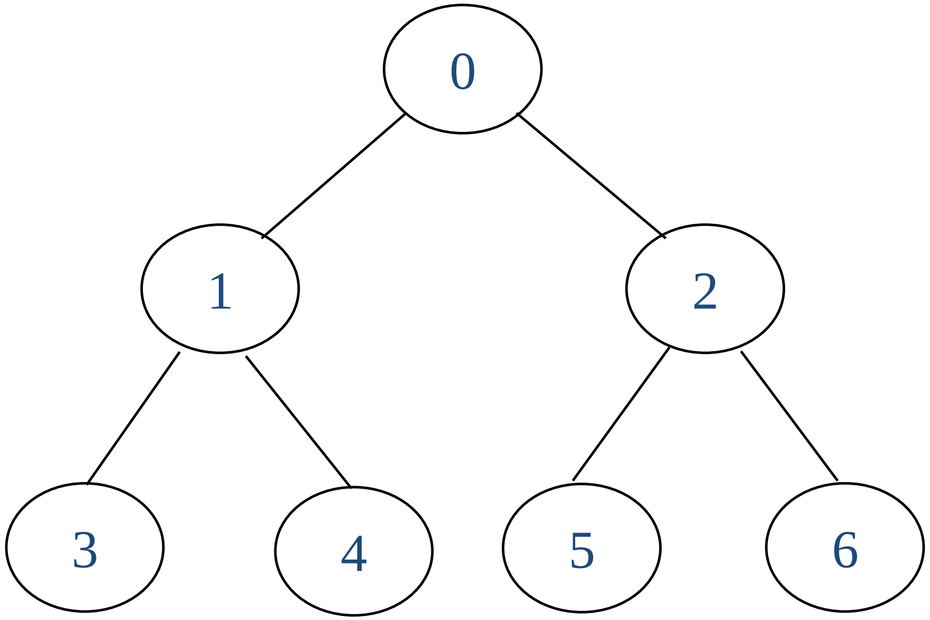
- **Examples**

Graph G2=(V,E):

  - V(G2)={0,1,2,3,4,5,6}
  - E(G2)={(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}

G_2 is also a tree

Tree is a special case of graph

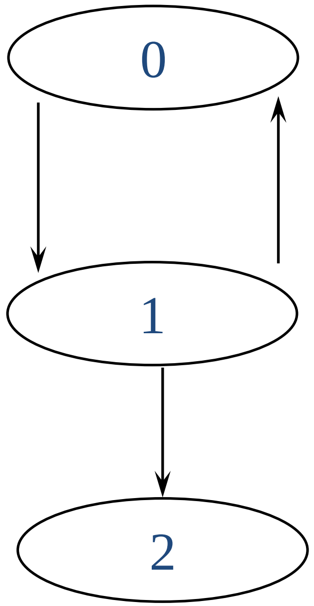


- **Examples**

  Graph G3=(V,E):

  - V(G3)={0,1,2}
  - E(G3)={<0,1>,<1,0>,<1,2>}

  Directed graph (digraph)



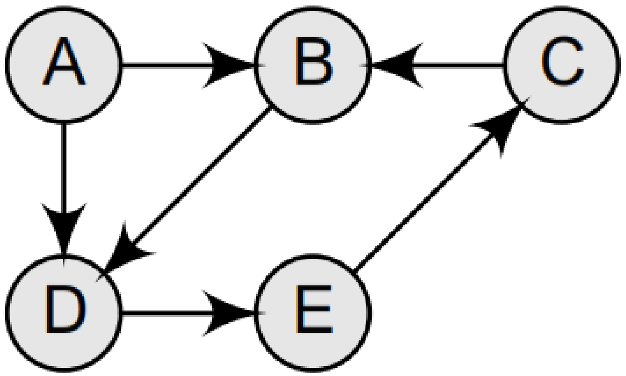
- **Examples**

  Directed graph

  a graph with 5 vertices and 6 edges:

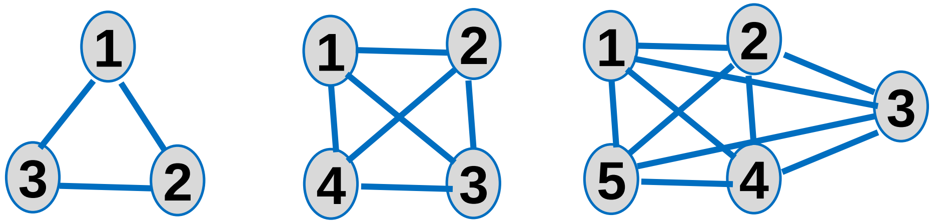
  V(G) = {A, B, C, D, E} and

  E(G) = {<A, B>, <C,B>, <A, D>, <B, D>, <D, E>, <E,C>}.



- Complete Graph
  - **Complete Graph** is a graph that has the maximum number of edges
  - For undirected graph with n vertices, the maximum number of edges is n(n-1)/2
  - For directed graph with n vertices, the maximum number of edges is n(n-1)

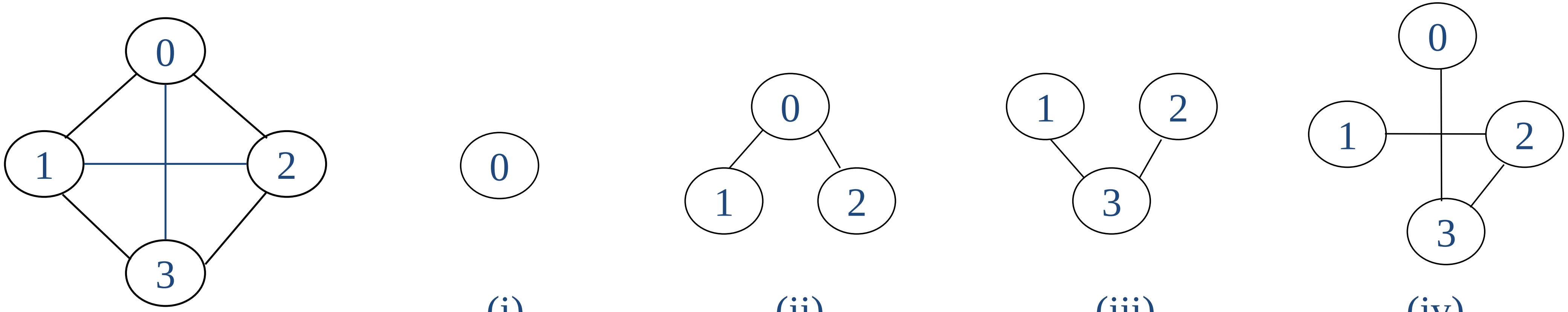
Example:



…Terminology

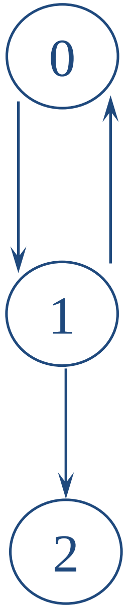
- **Adjacent and Incident**
  - If (u,v) is an edge in an undirected graph,
    - Adjacent: u and v are adjacent
    - Incident: The edge (u,v) is incident on vertices u and v
  - If <u,v> is an edge in a directed graph
    - Adjacent: u is adjacent to v, and v is adjacent from u
    - Incident: The edge <u,v> is incident on u and v

- **Subgraph**
  - A subgraph of G is a graph G’ such that
    - V(G’) $\subseteq$ V(G)
    - E(G’) $\subseteq$ E(G)
  - Some of the subgraph of G_1

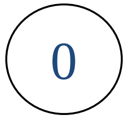


G_1 (i) (ii) (iii) (iv)

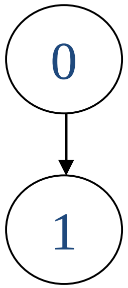
- **Some of the subgraphs of G_3**



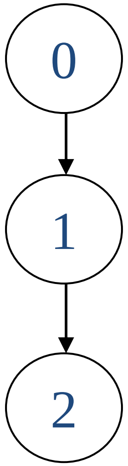
G_3



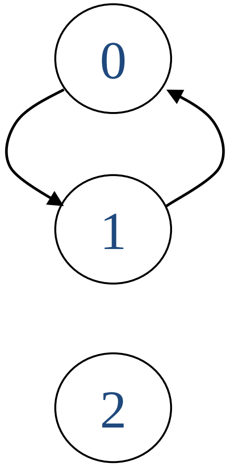
(i)



(ii)

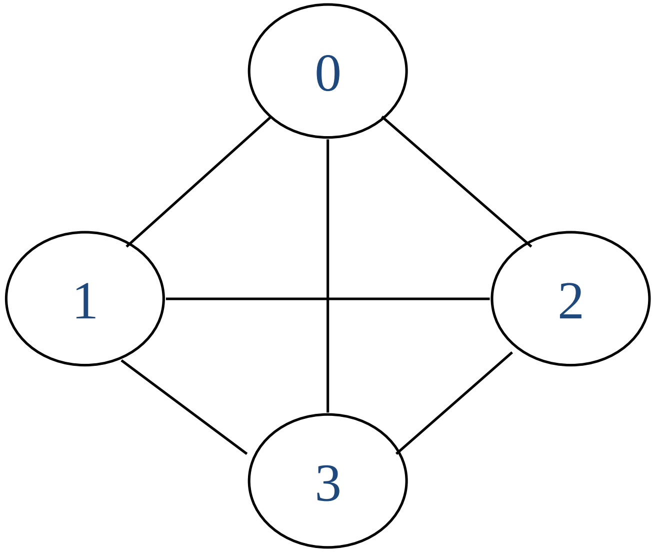


(iii)



(iv)

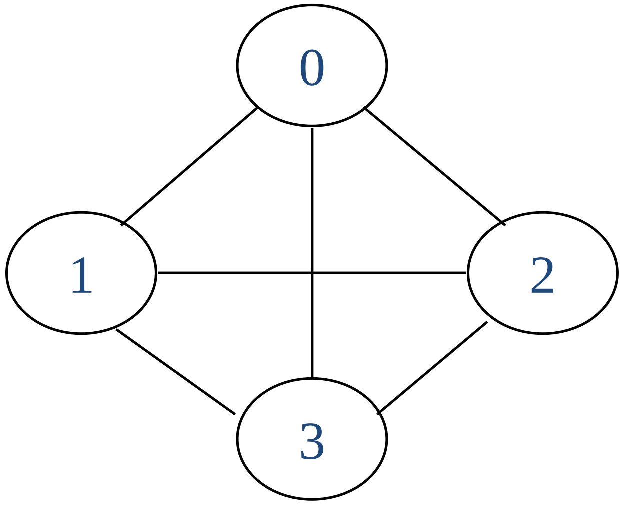
- **Path**
  - Path from u to v in G
    - a sequence of vertices u, i_1, i_2,…,i_k, v
    - If G is undirected: $(u,i_1), (i_1,i_2),\ldots, (i_k,v) \in E(G)$
    - If G is directed: $<u,i_1>,<i_1,i_2>,\ldots,<i_k,v> \in E(G)$
  - Length
    - The length of a path is the number of edges on it.
    - Length of 0,1,3,2 is 3



- **Simple Path**
  - is a path in which all vertices except possibly the first and last are distinct.

    ⇨ 0,1,3,2 is simple path

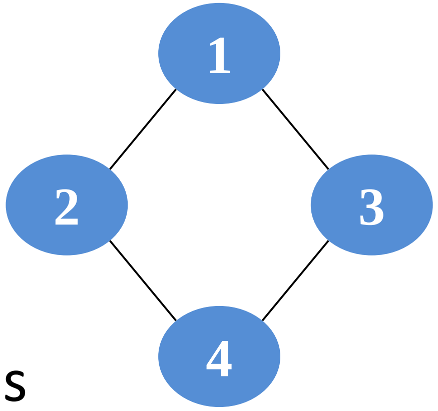
    0,1,3,1 is path but not simple



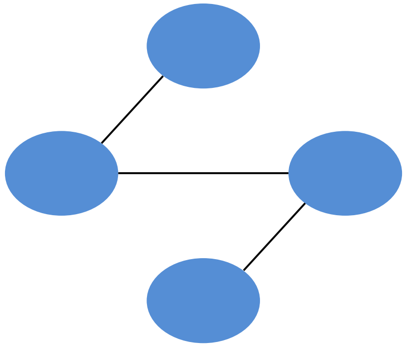
- **Cycle**
  – a simple path, first and last vertices are same.

    ⇨ 0,1,2,0 is a cycle
- **Acyclic graph**
  – No cycle is in graph

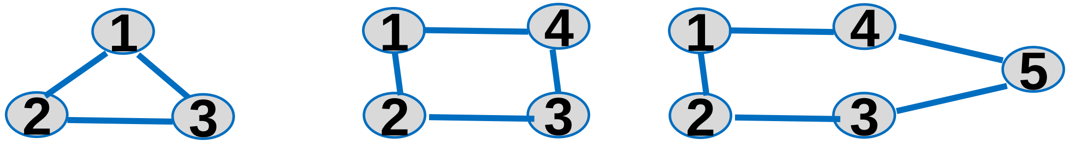
- **Connected**
  - Two vertices u and v are connected if in an undirected graph G, ∃ a path in G from u to v
  - A graph G is connected, if any vertex pair u,v is connected



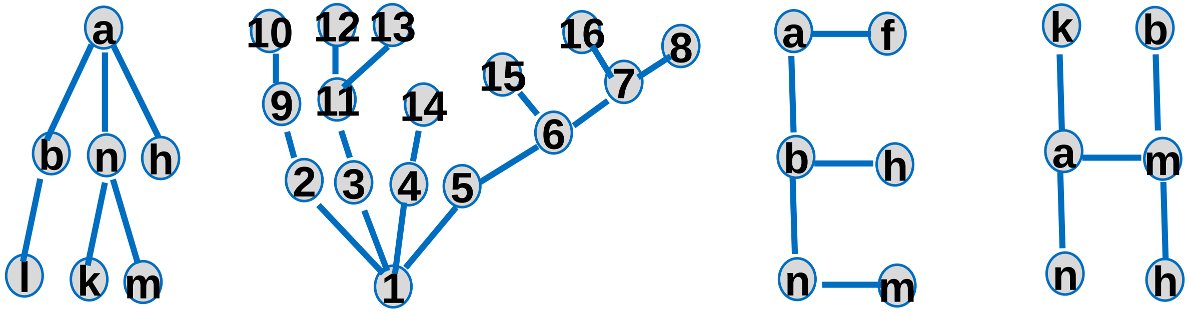
connected

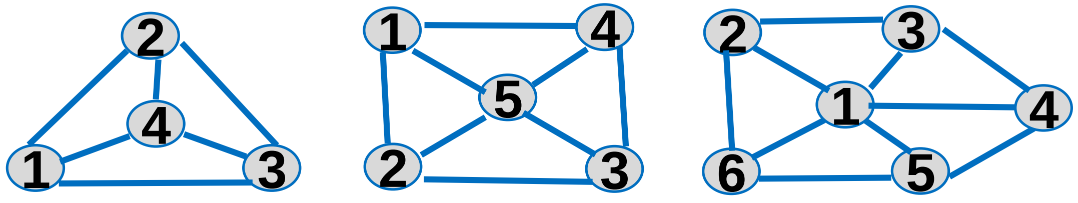

connected

- **Connected Component**
  - a maximal connected subgraph.

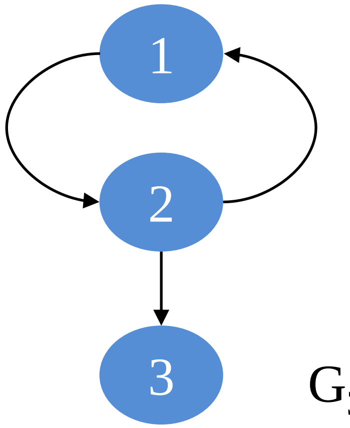


- **Tree is a connected acyclic graph**



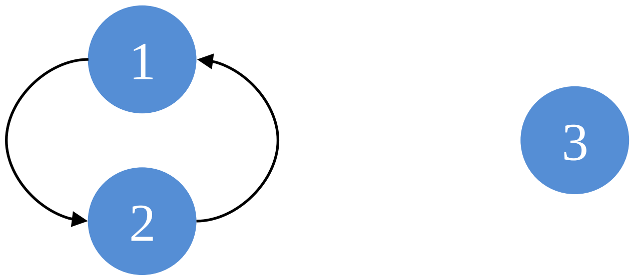


- **Strongly Connected**
  - u, v are strongly connected if in a directed graph (digraph) G, ∃ a path in G from u to v.
  - A directed graph G is strongly connected, if any vertex pair u,v is connected

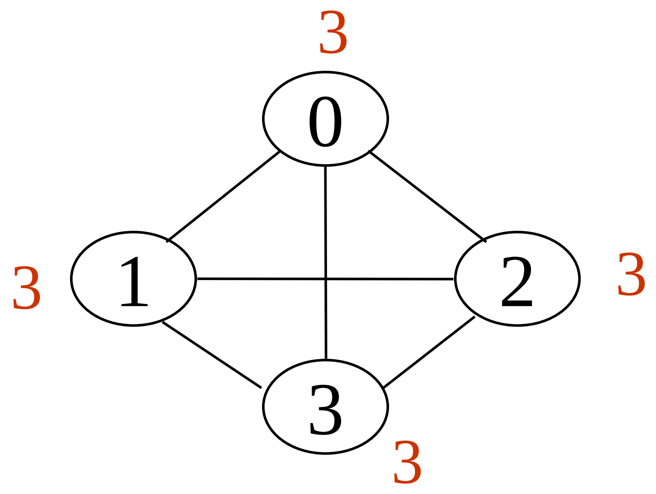


G_3

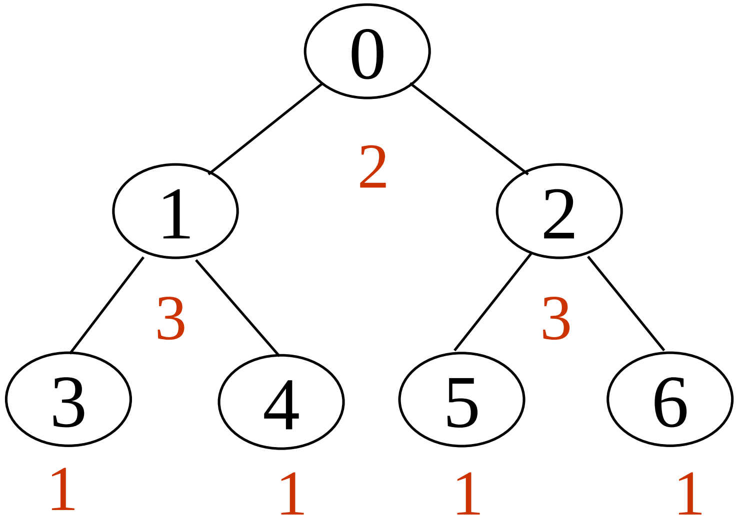
- **Strongly Connected Component**
  - a maximal strongly connected subgraph



- **Degree of Vertex**
  – is the number of edges incident to that vertex
- **Degree in directed graph**
  – Indegree
  – Outdegree
- **Summation of all vertices' degrees are 2|E|**

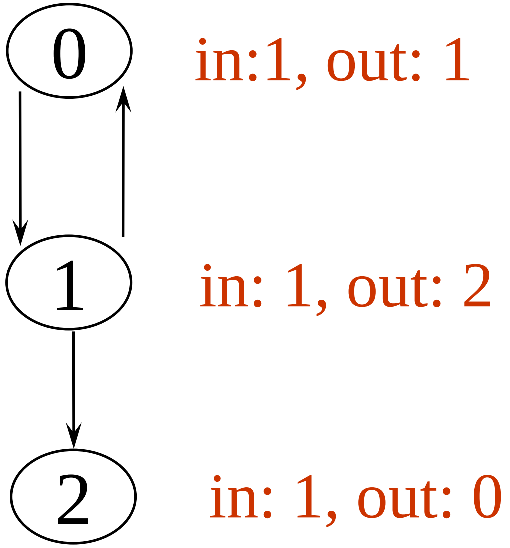


G1



G2

undirected graph



directed graph

in-degree

out-degree

- **Weighted Edge**
  - In many applications, the edges of a graph are assigned weights
  - These weights may represent the distance from one vertex to another
  - A graph with weighted edges is called a **network**

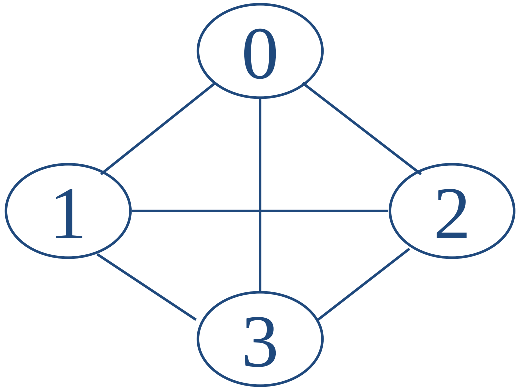
CONTENT

- Terminology
- **Graph Representations**
- Graph Traversals

- **Graph Representations**
  - **Adjacency Matrix**
  - **Adjacency Lists**
  - Adjacency Multi-lists

- **Adjacency Matrix**
  - Let G = (V, E) with n vertices, $n \geq 1$. The adjacency matrix of G is a 2-dimensional $n \times n$ matrix, A
    - $A(i, j) = 1$ iff $(v_i, v_j) \in E(G)$ ($\langle v_i, v_j \rangle$ for a digraph)
    - $A(i, j) = 0$ otherwise
  - The adjacency matrix for an undirected graph is symmetric
  - The adjacency matrix for a digraph need not be symmetric

- **Example**

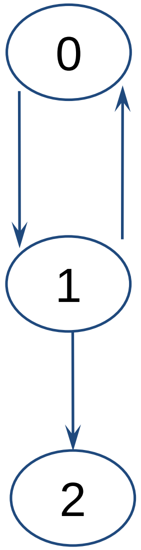


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

undirected: $n^2/2$

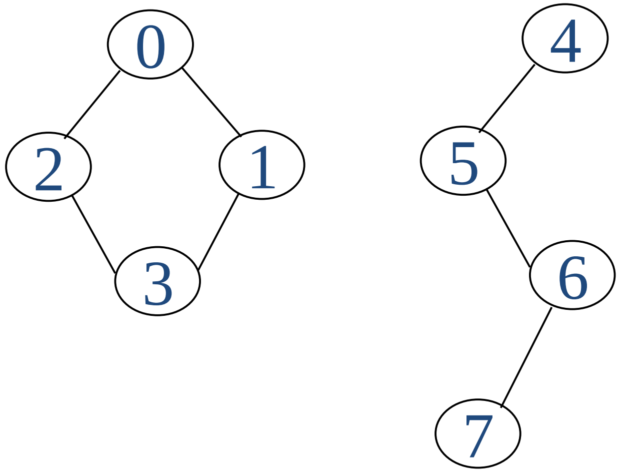
directed: n^2



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G2

symmetric



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

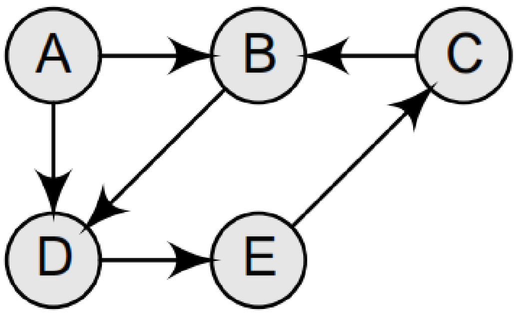
Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i]$$

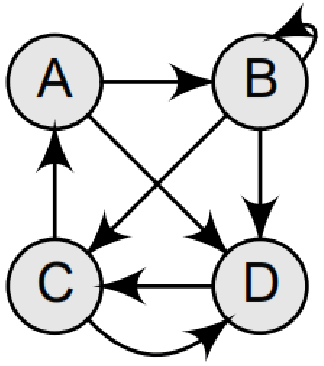
$$outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

- **Adjacency Matrix**



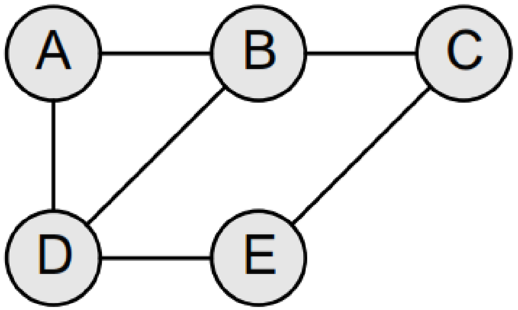
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(a) Directed graph



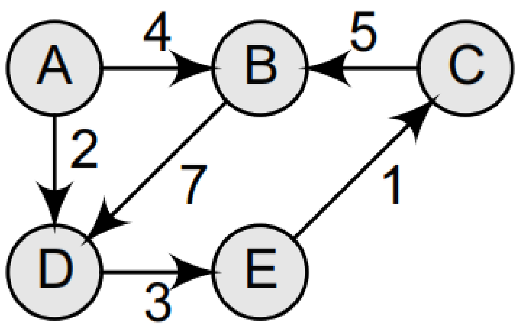
$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

(b) Directed graph with loop



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

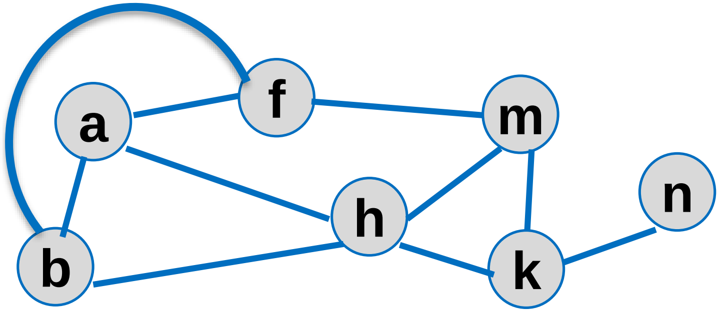
(c) Undirected graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 2 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 5 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 3 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

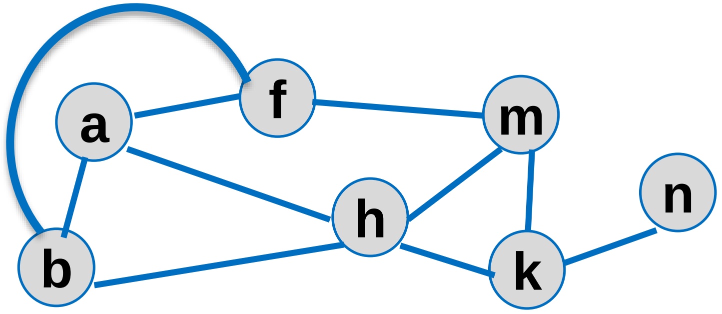
(d) Weighted graph

- **Adjacency Matrix**

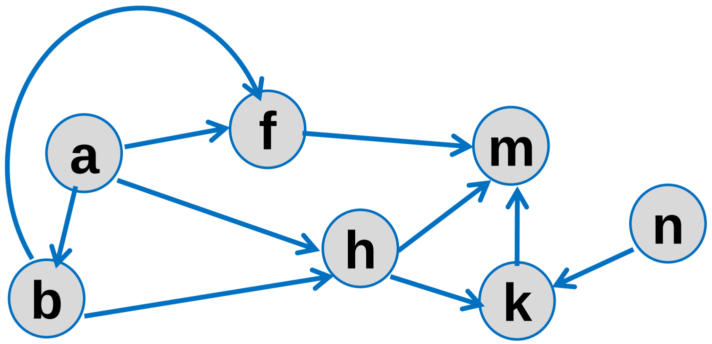


- **Adjacency Matrix**

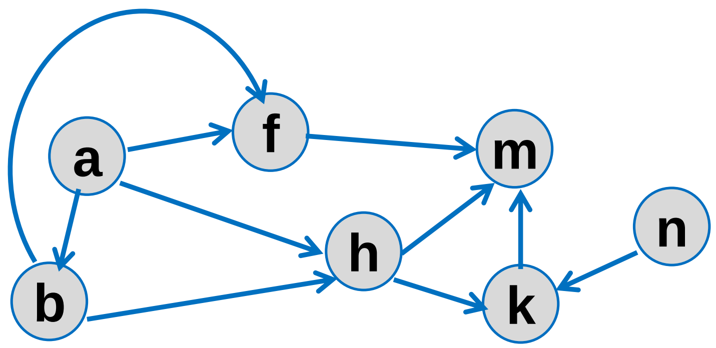
- $$\begin{array}{c c} & \begin{array}{ccccccc} a & b & f & h & m & n & k \end{array} \\ \begin{array}{c} a \\ b \\ f \\ h \\ m \\ n \\ k \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$



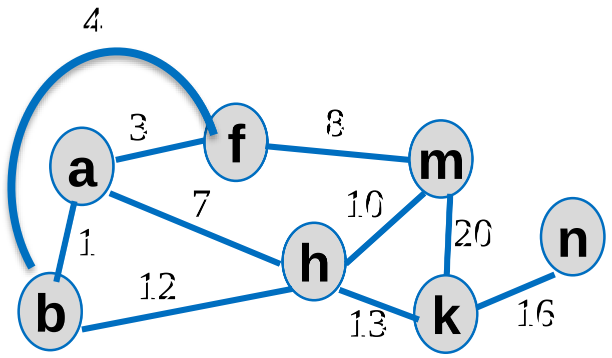
- **Adjacency Matrix**



- **Adjacency Matrix**
-
- $$\begin{array}{c} \\ a \\ b \\ f \\ h \\ m \\ n \\ k \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & f & h & m & n & k \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

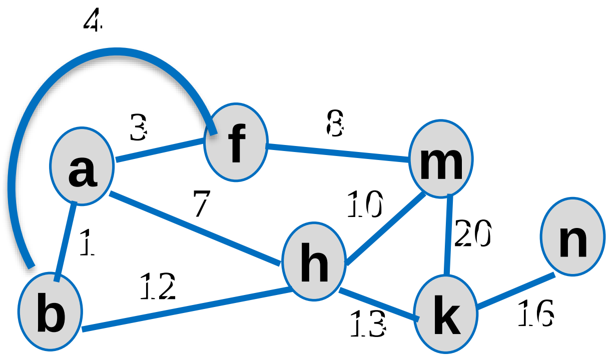


- **Adjacency Matrix**

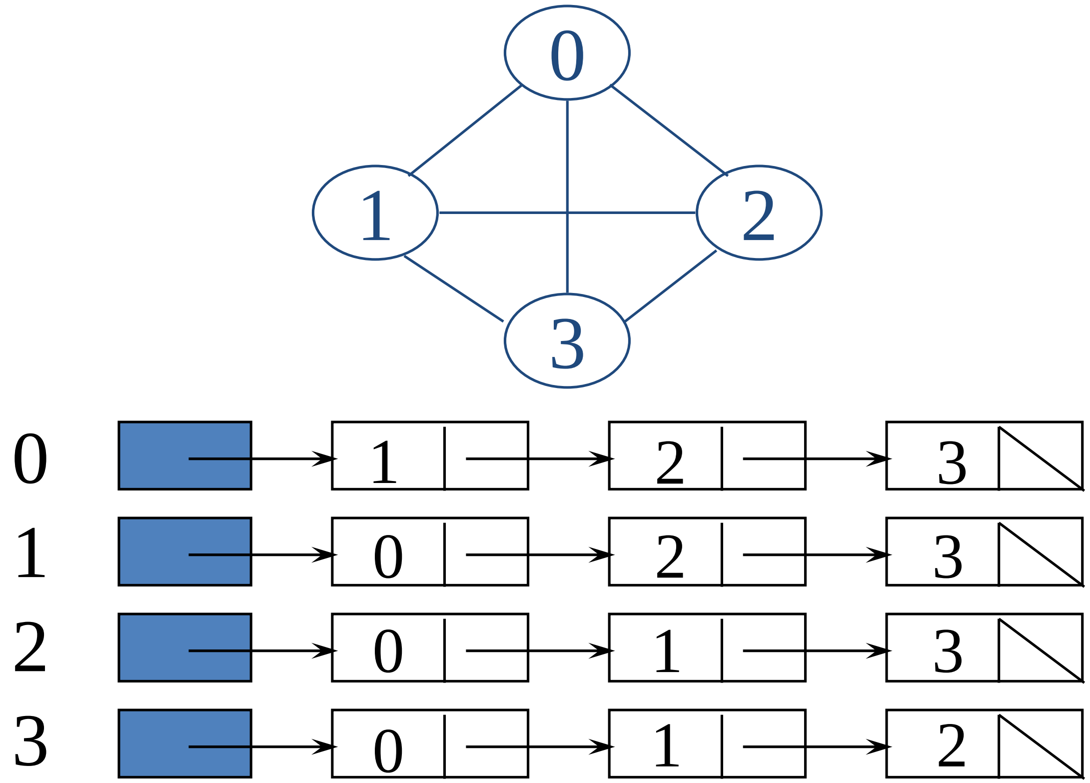


- **Adjacency Matrix**

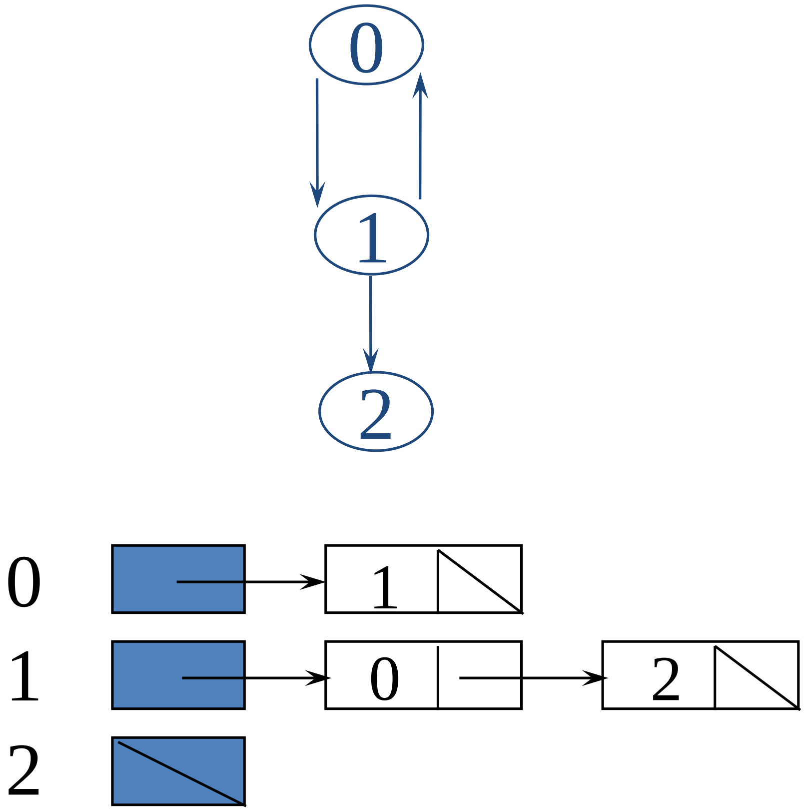
- $$\begin{array}{c c}
 & \begin{array}{ccccccc} a & b & f & h & m & n & k \end{array} \\
\begin{array}{c} a \\ b \\ f \\ h \\ m \\ n \\ k \end{array} &
\begin{bmatrix}
0 & 1 & 3 & 7 & 0 & 0 & 0 \\
1 & 0 & 4 & 12 & 0 & 0 & 0 \\
3 & 4 & 0 & 0 & 8 & 0 & 0 \\
7 & 12 & 0 & 0 & 10 & 0 & 13 \\
0 & 0 & 8 & 10 & 0 & 0 & 20 \\
0 & 0 & 0 & 0 & 0 & 0 & 16 \\
0 & 0 & 0 & 13 & 20 & 16 & 0
\end{bmatrix}
\end{array}$$



- **Adjacency List**
  - Replace *n* rows of the adjacency matrix with *n* linked list
  - Example (1)

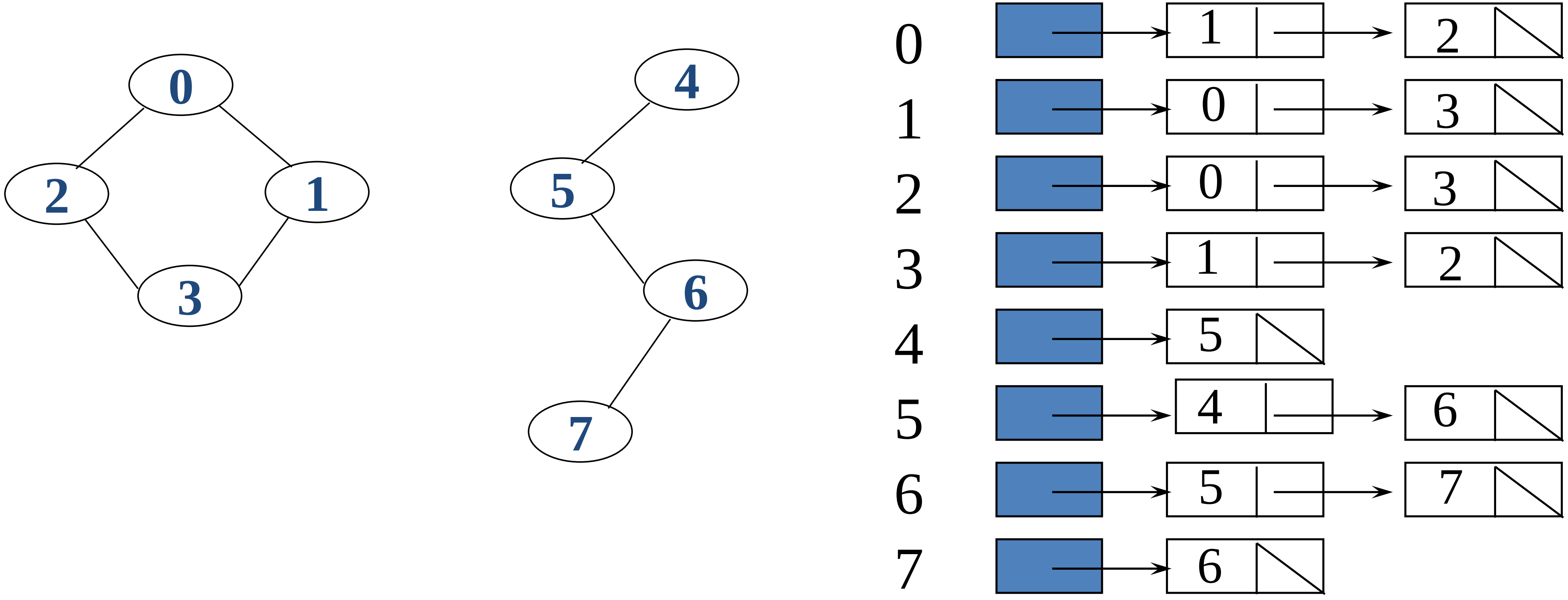


G_1



G_3

- **Adjacency List**
  - Replace *n* rows of the adjacency matrix with *n* linked list
  - Example (2)



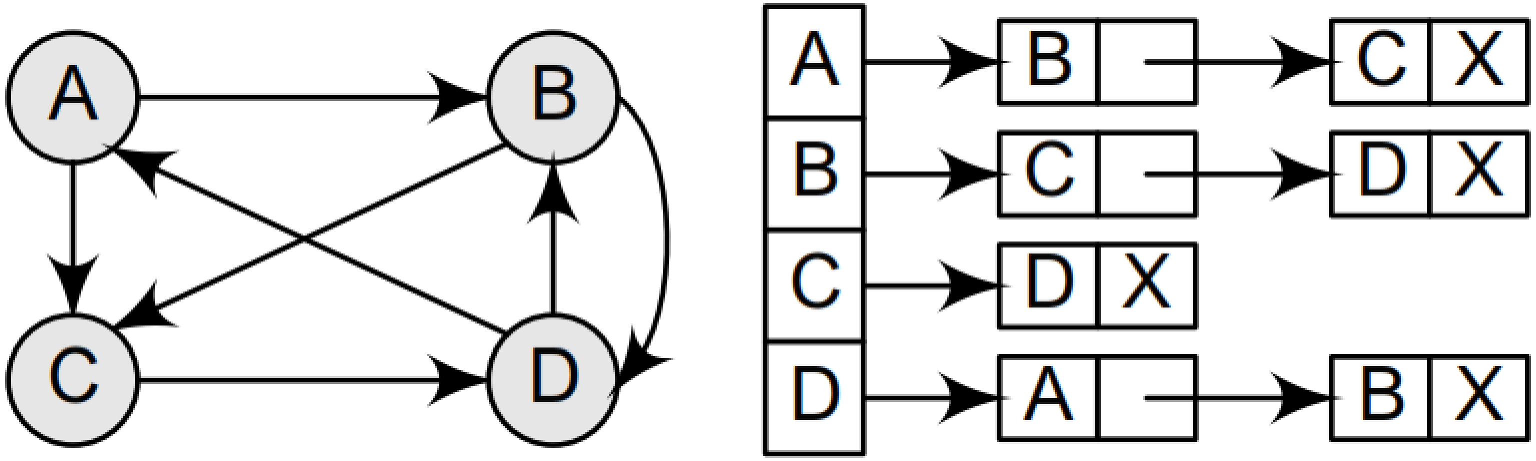
G_4

An undirected graph with n vertices and e edges => n head nodes and 2e list nodes

- **Adjacency List**
  - Data Structures
    - Each row in adjacency matrix is represented as an adjacency list

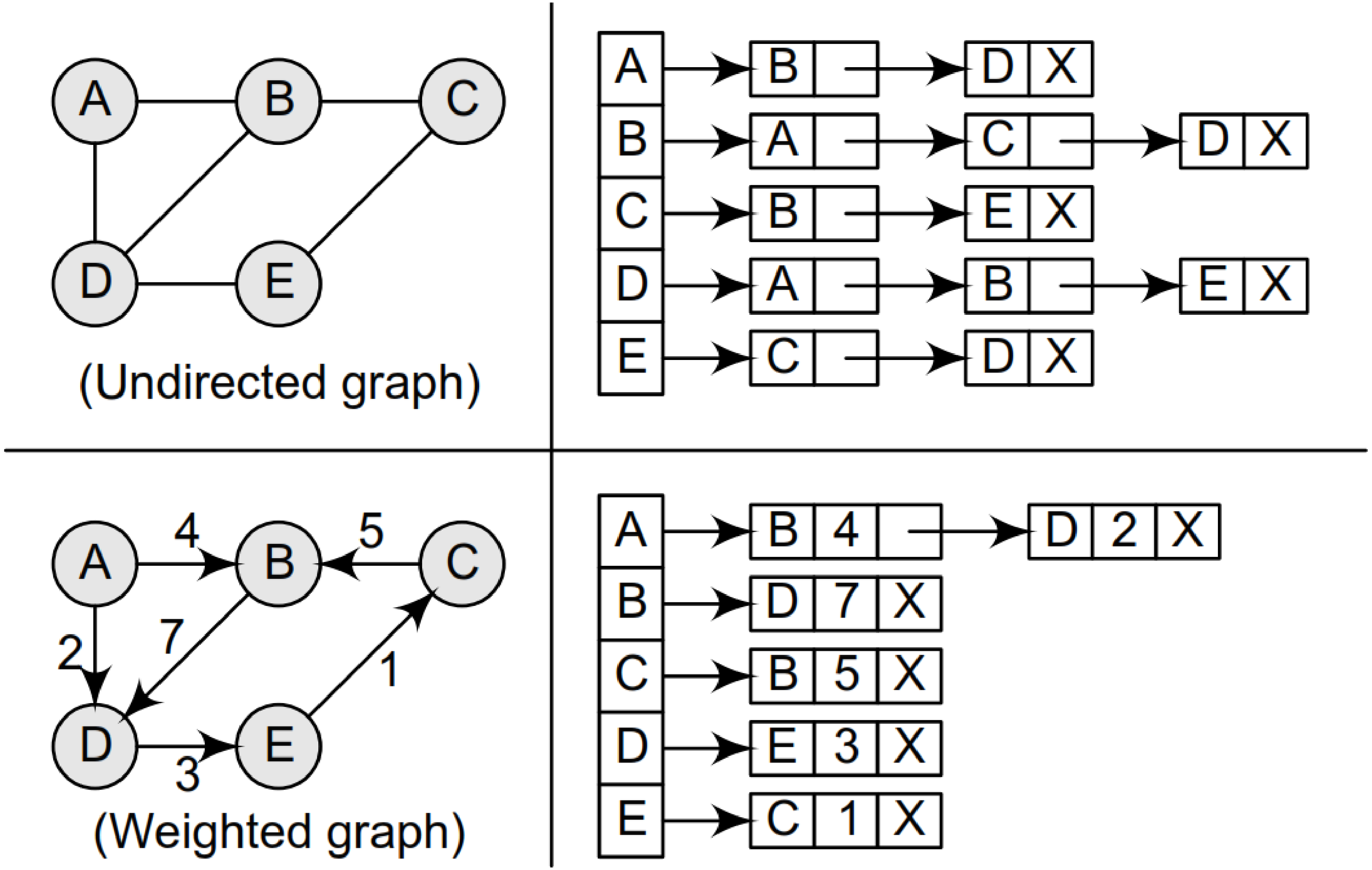
```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct  node {
            int vertex;
           struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0;          /* vertices currently in use */
```

- **Adjacency List**

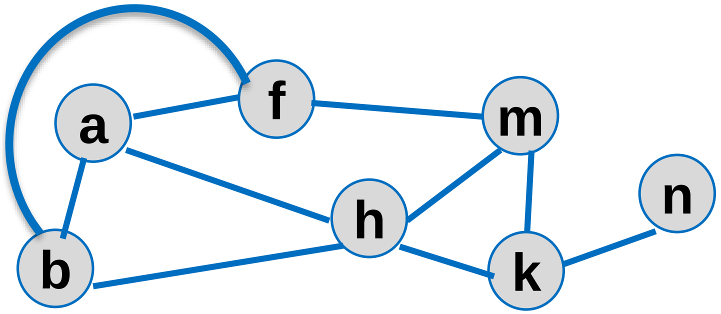


Graph G and its adjacency list

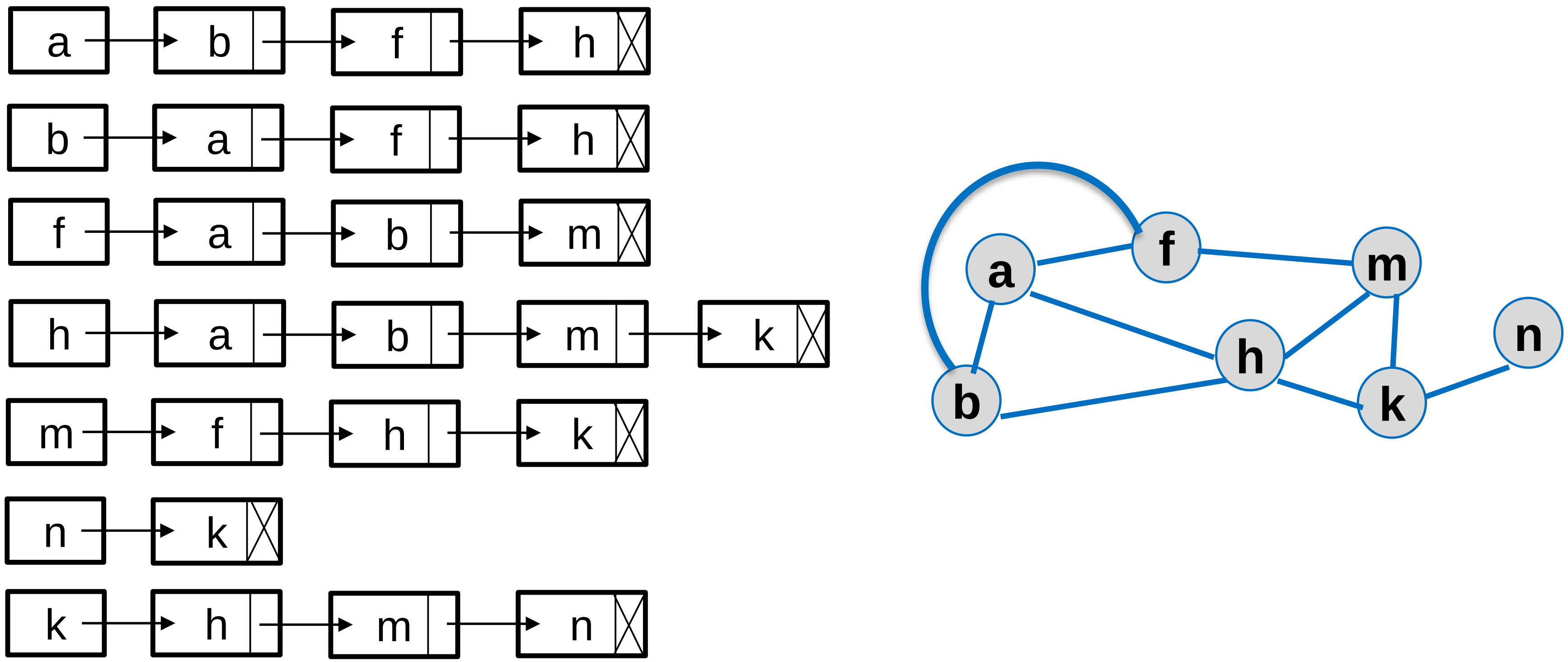
- **Adjacency List**



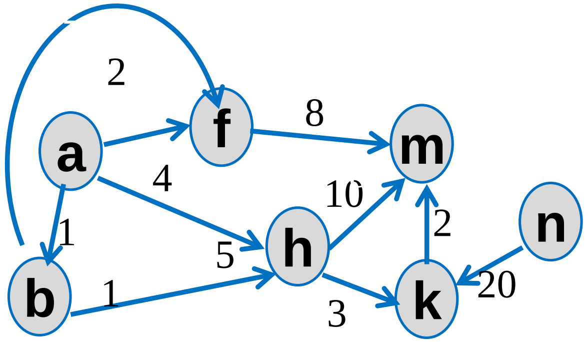
- **Adjacency List**



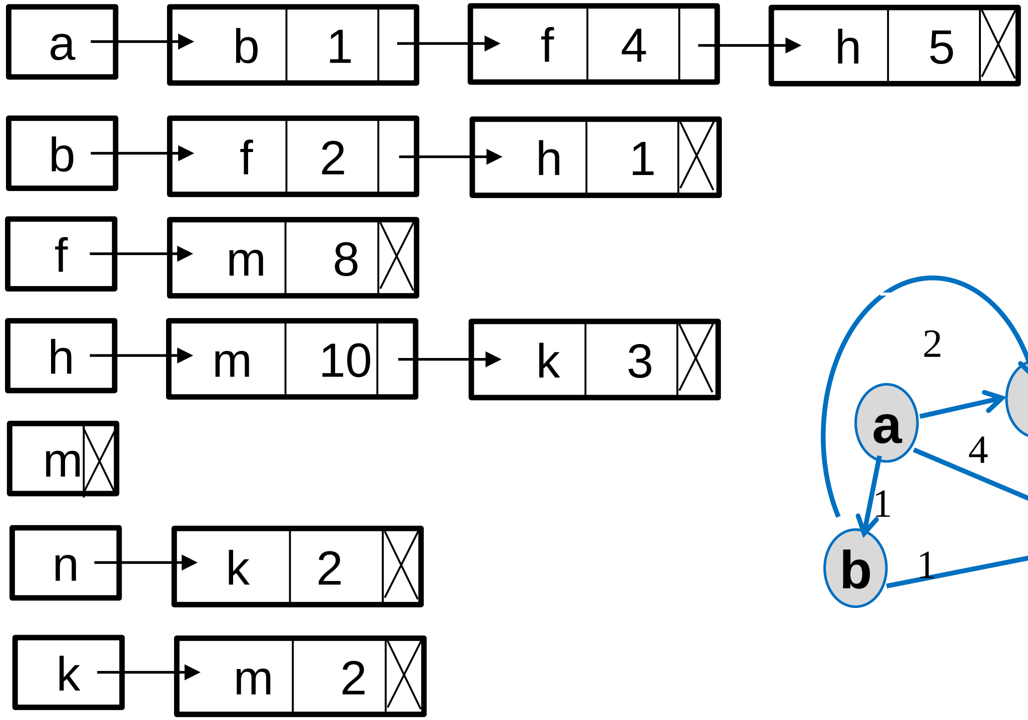
- **Adjacency List**

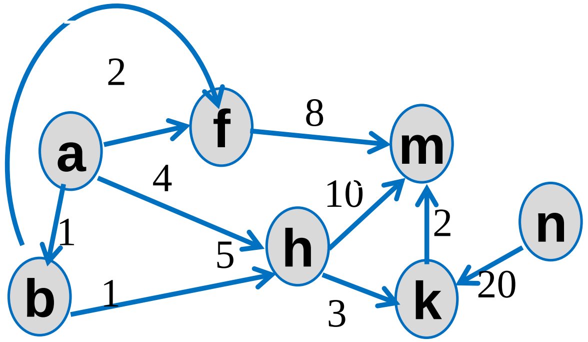


- **Adjacency List**



- **Adjacency List**

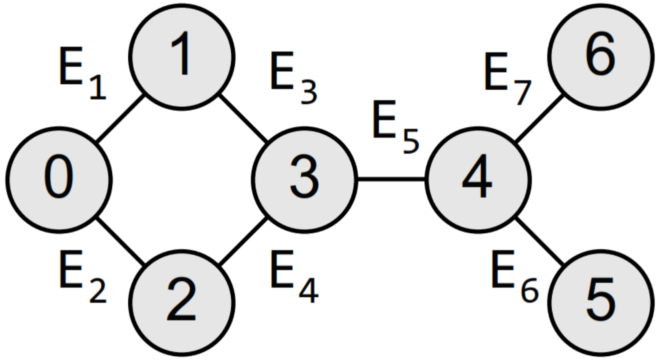




- **Adjacency Multi-lists**

  A multi-lists representation basically consists of two parts:

  - nodes' information
  - linked lists storing information about edges.



VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7

CONTENT

- Terminology
- Graph Representations
- **Graph Traversals**

- **Traversal**

  Given G = (V,E) and vertex v, find or visit all $w \in V$, such that w connects v

  - Depth First Search (DFS)
  - Breadth First Search (BFS)

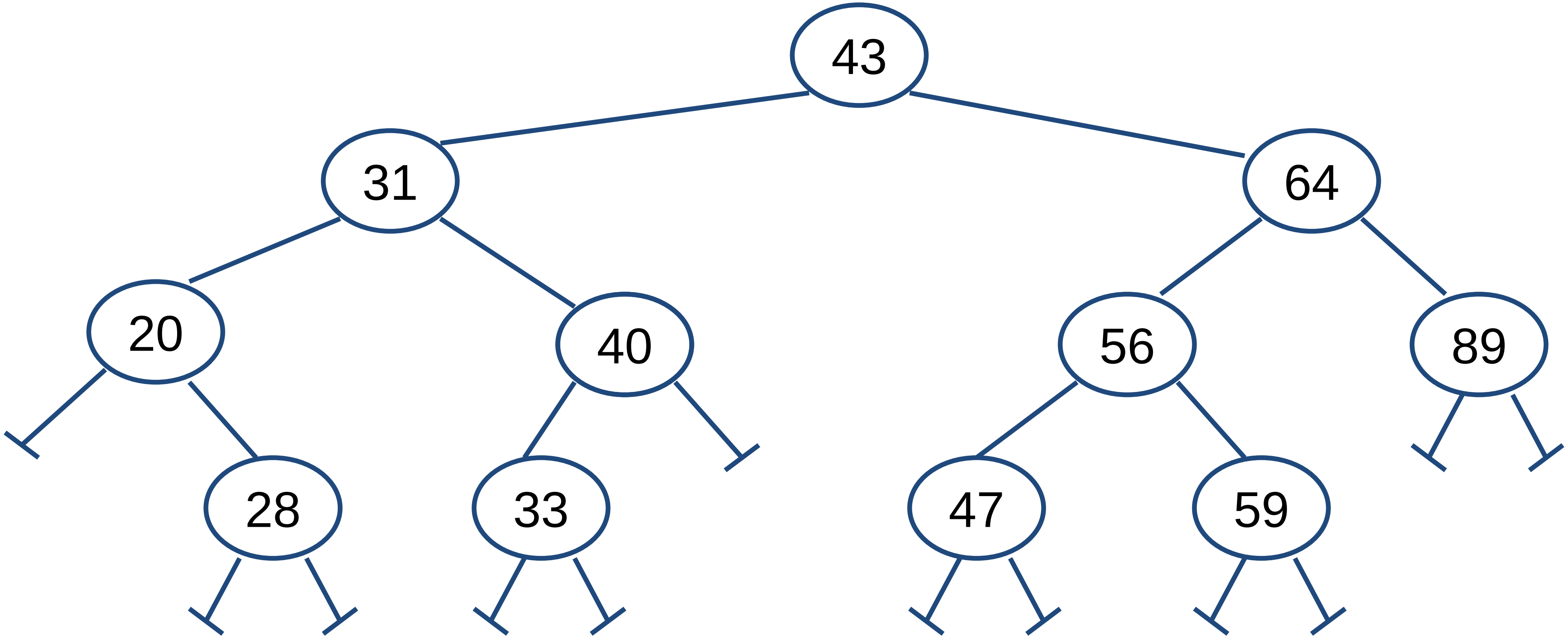
- **Applications**
  - Connected component
  - Spanning trees
  - Biconnected component

- **Depth-First Search** (DFS)
  - like depth-first search in a tree, we search **as deeply as possible** by visiting a node, and then recursively performing depth-first search on each adjacent node

- **Breadth-First Search** (BFS)
  - like breadth-first search in a tree, we search **as broadly as possible** by visiting a node, and then immediately visiting all nodes adjacent to that node

- **Depth First Search** (DFS)
  - Begin the search by visiting the start vertex **v**
    - If **v** has an unvisited neighbor, traverse it recursively
    - Otherwise, backtrack
  - Very similar to preorder traversal of a binary tree (node, left, right)

- **Example: Preorder**



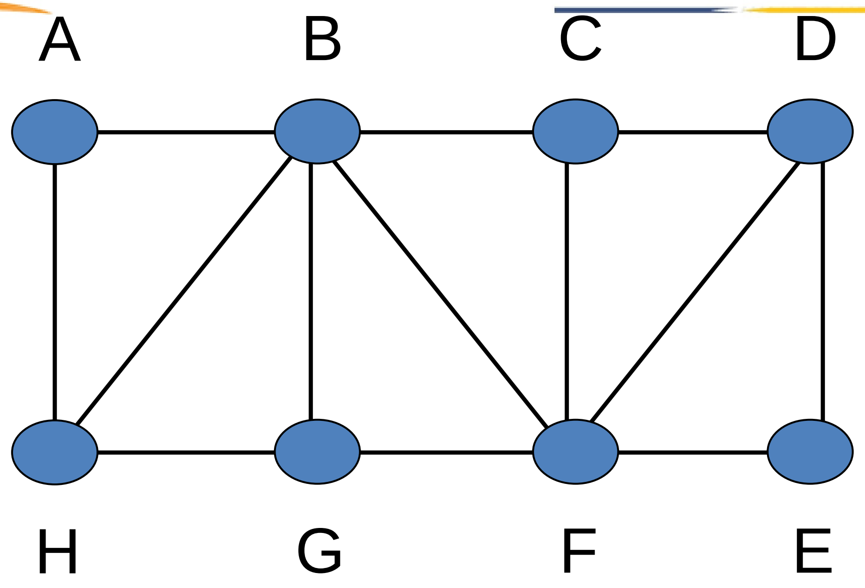
43	31	20	28	40	33	64	56	47	59	89

- **Algorithm**

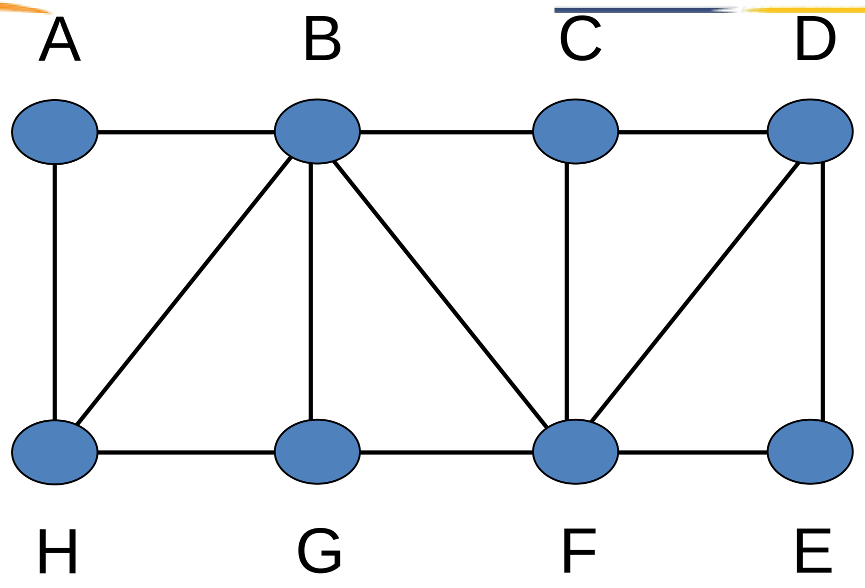
```
Depth_First_Search (VERTEX V){
    Visit V;
    Set the visit flag for the vertex V to TRUE;
    For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        If (Vi has not been previously visited)
            Depth_First_Search (Vi)
}
```

- **Time is *$O(n + e)$* for adjacency lists**
- **Time is *$O(n^2)$* for adjacency matrices**

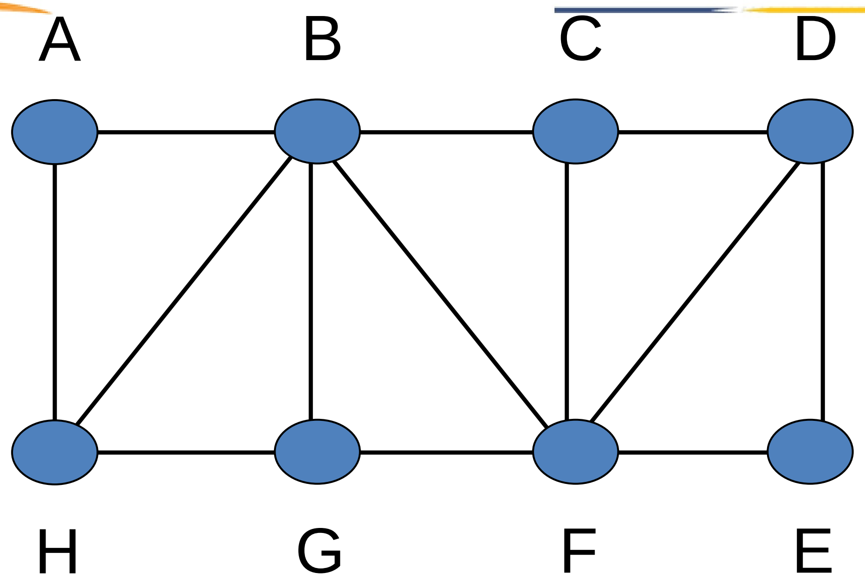
```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];

/* graph is represented as an adjacency list */
void dfs(int v){
        node_pointer w;
        visited[v]= TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
                if (!visited[w->vertex])
                        dfs(w->vertex);
}
```



Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

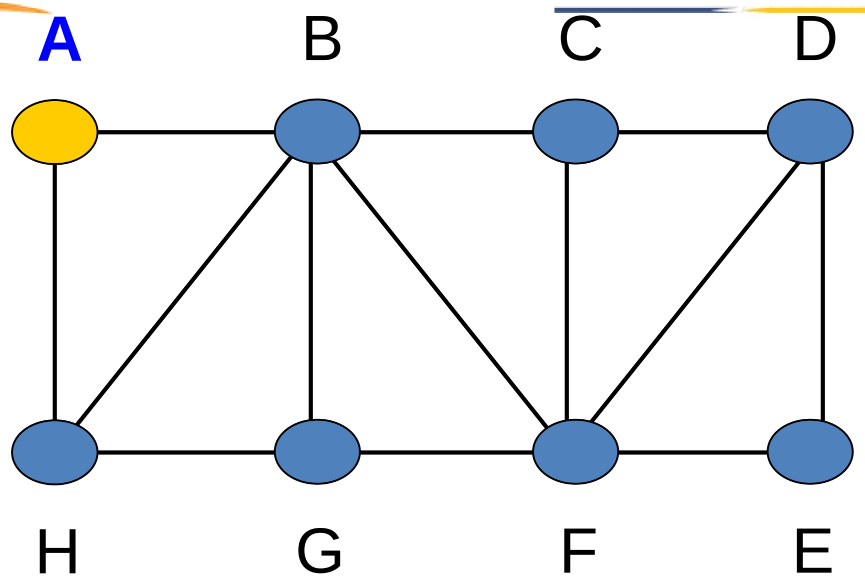


Vertex (label, visited?)		Adjacent Vertices
A *F*	→	B, H
B *F*	→	A, C, G, F, H
C *F*	→	B, D, F
D *F*	→	C, E, F
E *F*	→	D, F
F *F*	→	B, C, D, E, G
G *F*	→	B, F, H
H *F*	→	A, B, G




```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

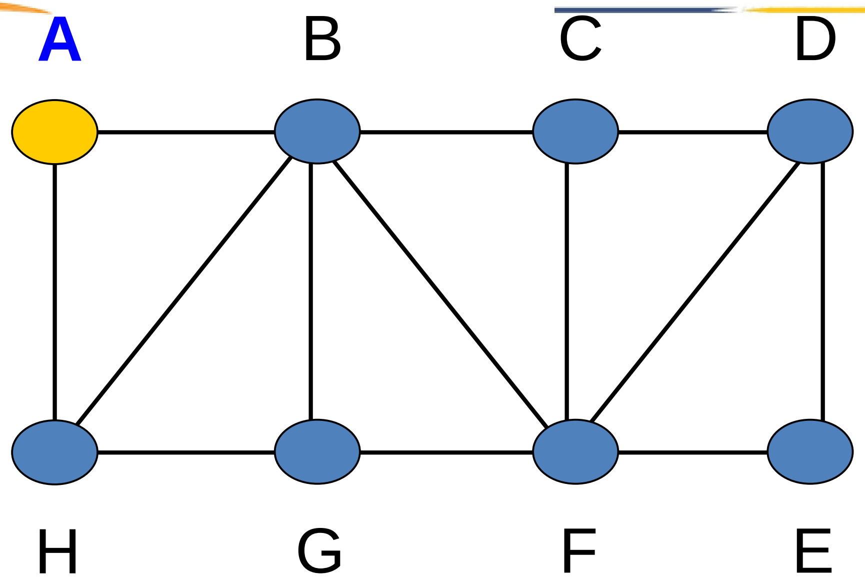
Vertex (label, visited?)		**Adjacent Vertices**
A *F*	⟶	B, H
B *F*	⟶	A, C, G, F, H
C *F*	⟶	B, D, F
D *F*	⟶	C, E, F
E *F*	⟶	D, F
F *F*	⟶	B, C, D, E, G
G *F*	⟶	B, F, H
H *F*	⟶	A, B, G




```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

Vertex (label, visited?)		**Adjacent Vertices**
A *F*	⟶	B, H
B *F*	⟶	A, C, G, F, H
C *F*	⟶	B, D, F
D *F*	⟶	C, E, F
E *F*	⟶	D, F
F *F*	⟶	B, C, D, E, G
G *F*	⟶	B, F, H
H *F*	⟶	A, B, G

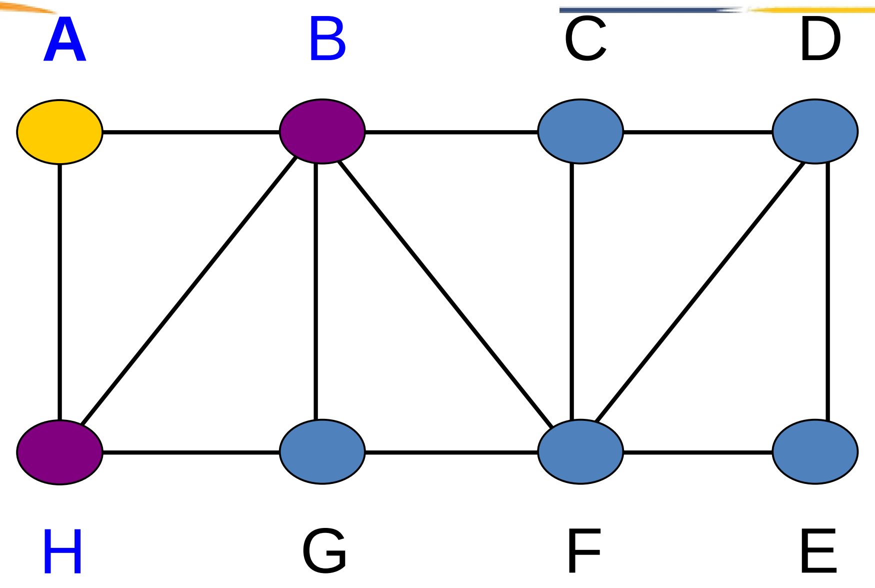
A




```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

Vertex (label, visited?)		**Adjacent Vertices**
A ***T***	⟶	B, H
B *F*	⟶	A, C, G, F, H
C *F*	⟶	B, D, F
D *F*	⟶	C, E, F
E *F*	⟶	D, F
F *F*	⟶	B, C, D, E, G
G *F*	⟶	B, F, H
H *F*	⟶	A, B, G

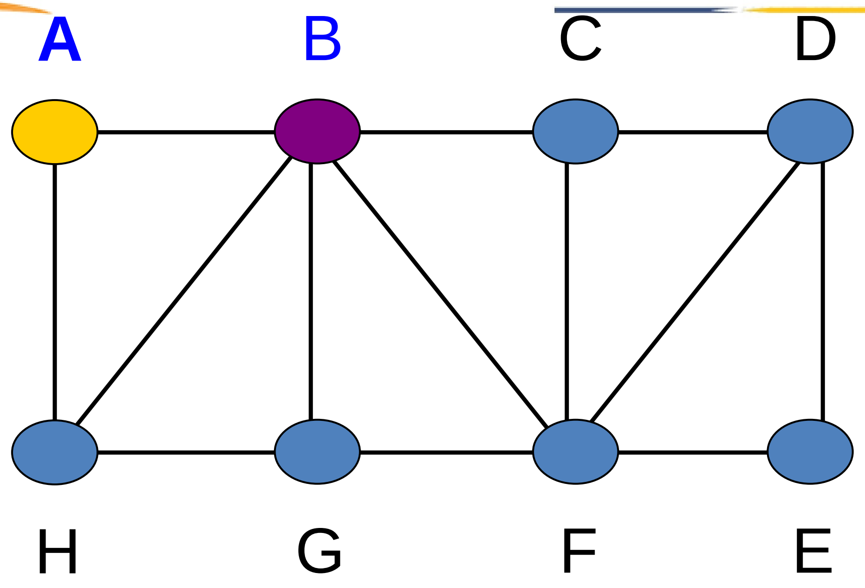
A



Vertex (label, visited?)		Adjacent Vertices
A T	→	B, H
B F	→	A, C, G, F, H
C F	→	B, D, F
D F	→	C, E, F
E F	→	D, F
F F	→	B, C, D, E, G
G F	→	B, F, H
H F	→	A, B, G

```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

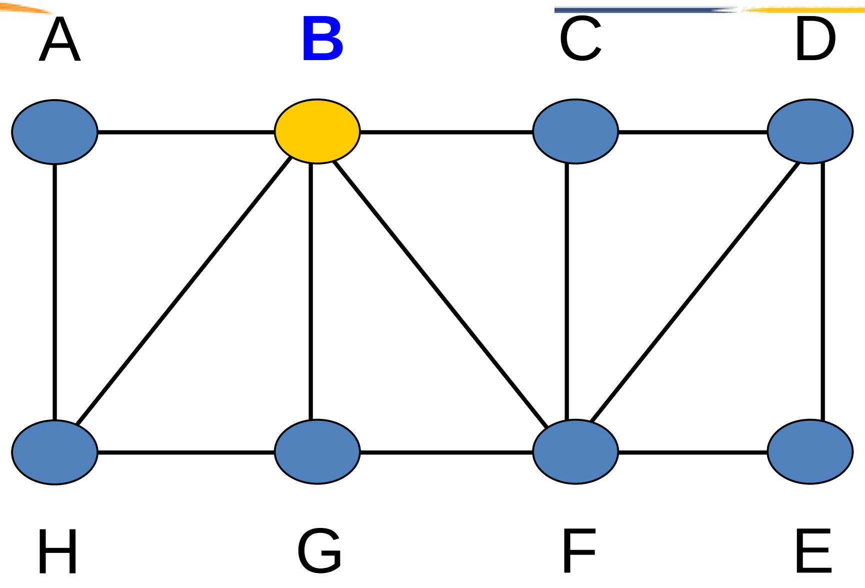
A



Vertex (label, visited?)	Adjacent Vertices
A T	B, H
B F	A, C, G, F, H
C F	B, D, F
D F	C, E, F
E F	D, F
F F	B, C, D, E, G
G F	B, F, H
H F	A, B, G

```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

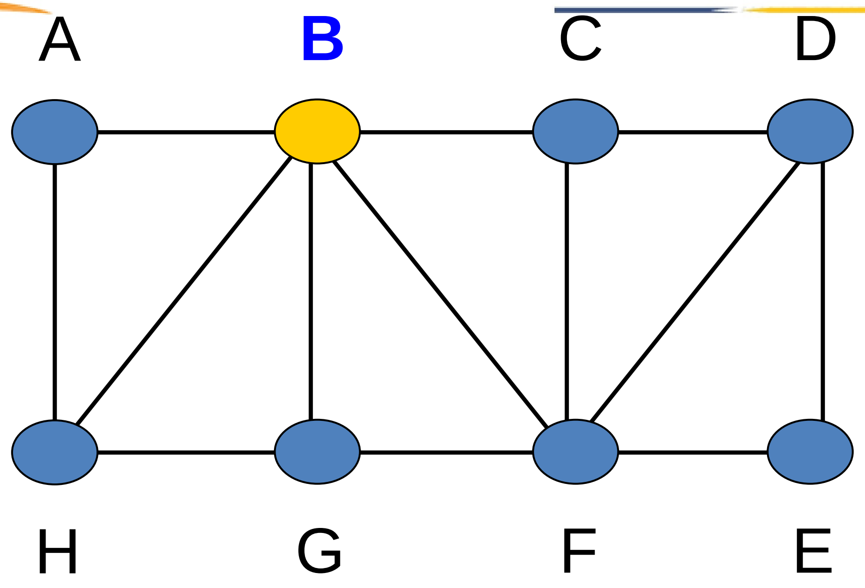
A



Vertex (label, visited?)		Adjacent Vertices
A T	→	B, H
B F	→	A, C, G, F, H
C F	→	B, D, F
D F	→	C, E, F
E F	→	D, F
F F	→	B, C, D, E, G
G F	→	B, F, H
H F	→	A, B, G

```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

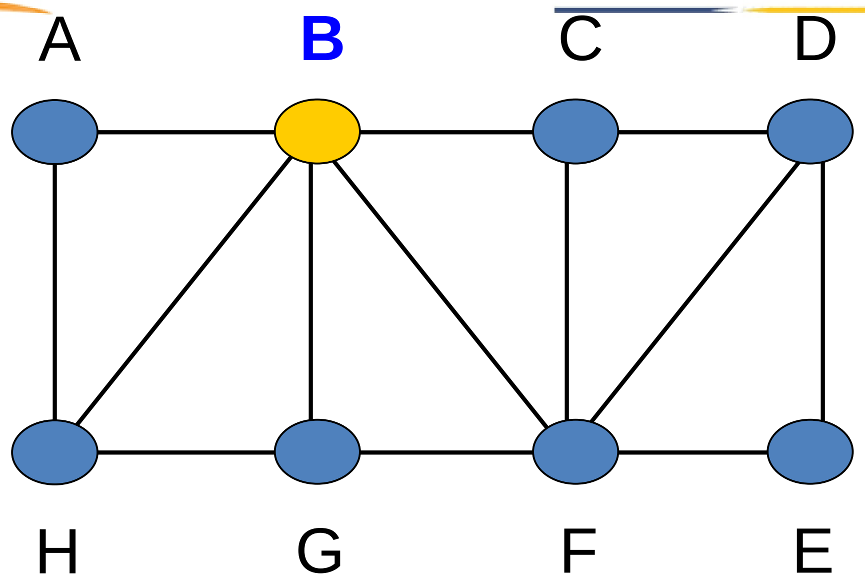
A



Vertex (label, visited?)		Adjacent Vertices
A *T*	→	B, H
B ***F***	→	A, C, G, F, H
C *F*	→	B, D, F
D *F*	→	C, E, F
E *F*	→	D, F
F *F*	→	B, C, D, E, G
G *F*	→	B, F, H
H *F*	→	A, B, G

```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

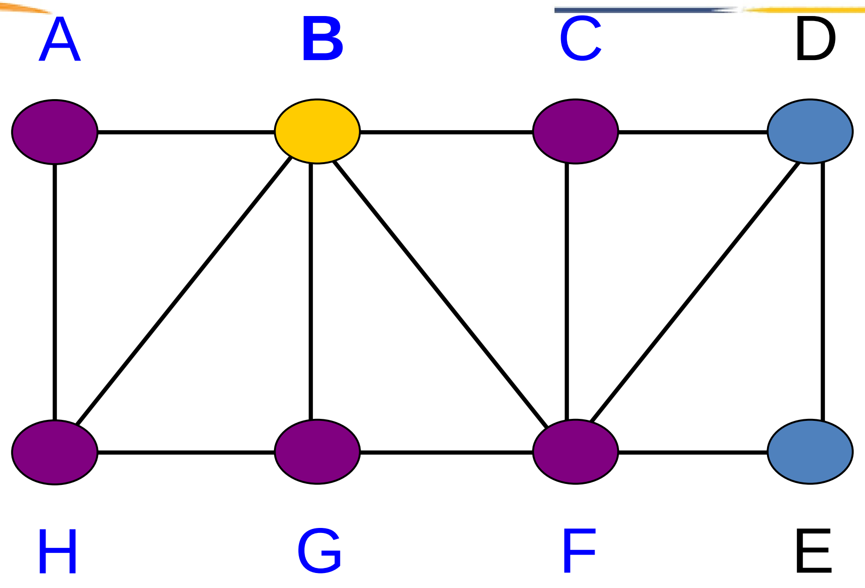
A B




```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

Vertex (label, visited?)		Adjacent Vertices
A *T*	→	B, H
B *T*	→	A, C, G, F, H
C *F*	→	B, D, F
D *F*	→	C, E, F
E *F*	→	D, F
F *F*	→	B, C, D, E, G
G *F*	→	B, F, H
H *F*	→	A, B, G

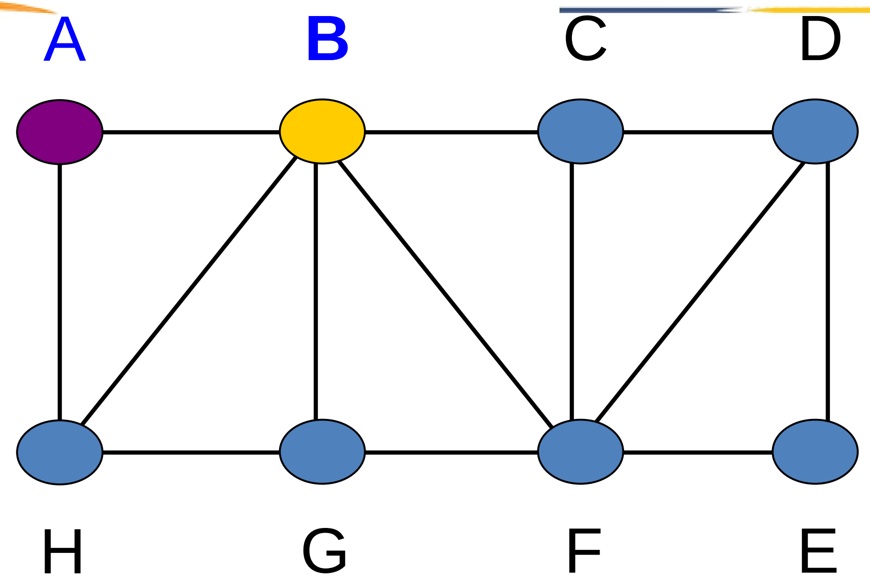
A B




```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

Vertex (label, visited?)		**Adjacent Vertices**
A *T*	→	B, H
B *T*	→	A, C, G, F, H
C *F*	→	B, D, F
D *F*	→	C, E, F
E *F*	→	D, F
F *F*	→	B, C, D, E, G
G *F*	→	B, F, H
H *F*	→	A, B, G

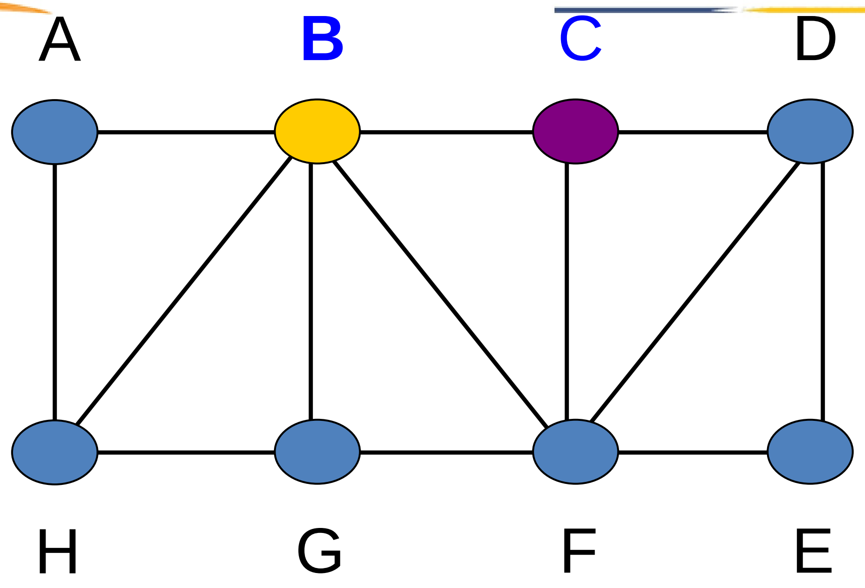
A B



Vertex (label, visited?)		Adjacent Vertices
A *T*	→	B, H
B *T*	→	A, C, G, F, H
C *F*	→	B, D, F
D *F*	→	C, E, F
E *F*	→	D, F
F *F*	→	B, C, D, E, G
G *F*	→	B, F, H
H *F*	→	A, B, G

```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

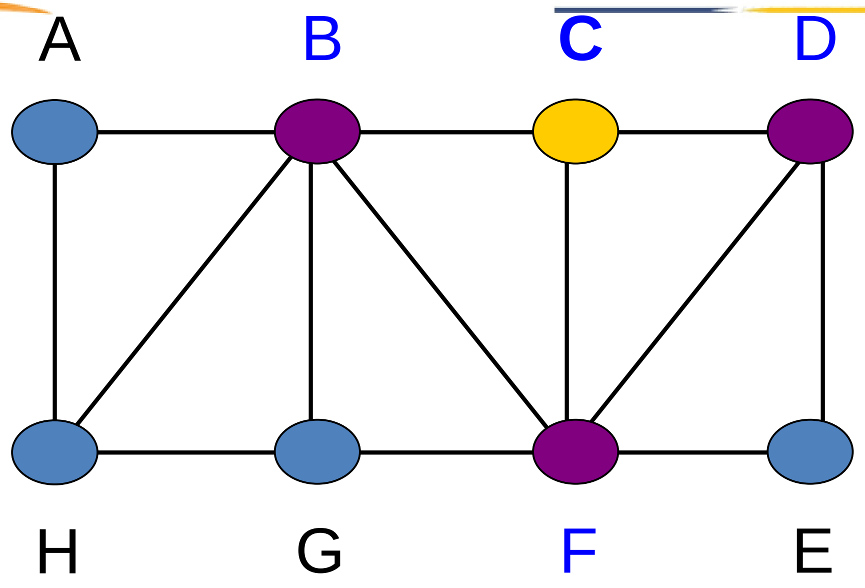
A B



Vertex (label, visited?)		Adjacent Vertices
A T	→	B, H
B T	→	A, C, G, F, H
C F	→	B, D, F
D F	→	C, E, F
E F	→	D, F
F F	→	B, C, D, E, G
G F	→	B, F, H
H F	→	A, B, G

```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

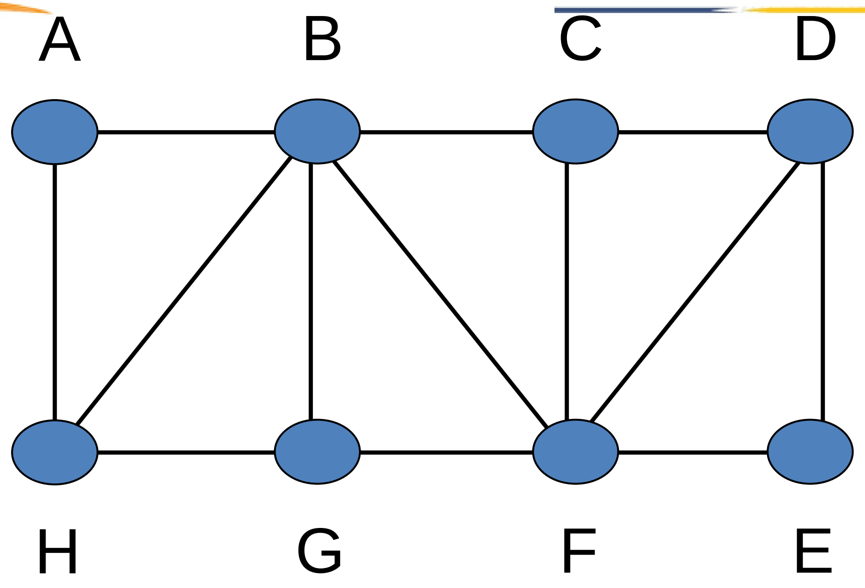
A B




```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

Vertex (label, visited?)		Adjacent Vertices
A *T*	→	B, H
B *T*	→	A, C, G, F, H
C *T*	→	B, D, F
D *F*	→	C, E, F
E *F*	→	D, F
F *F*	→	B, C, D, E, G
G *F*	→	B, F, H
H *F*	→	A, B, G

A B C



Vertex (label, visited?)		Adjacent Vertices
A *T*	⟶	B, H
B *T*	⟶	A, C, G, F, H
C *T*	⟶	B, D, F
D *T*	⟶	C, E, F
E *T*	⟶	D, F
F *T*	⟶	B, C, D, E, G
G *T*	⟶	B, F, H
H *T*	⟶	A, B, G

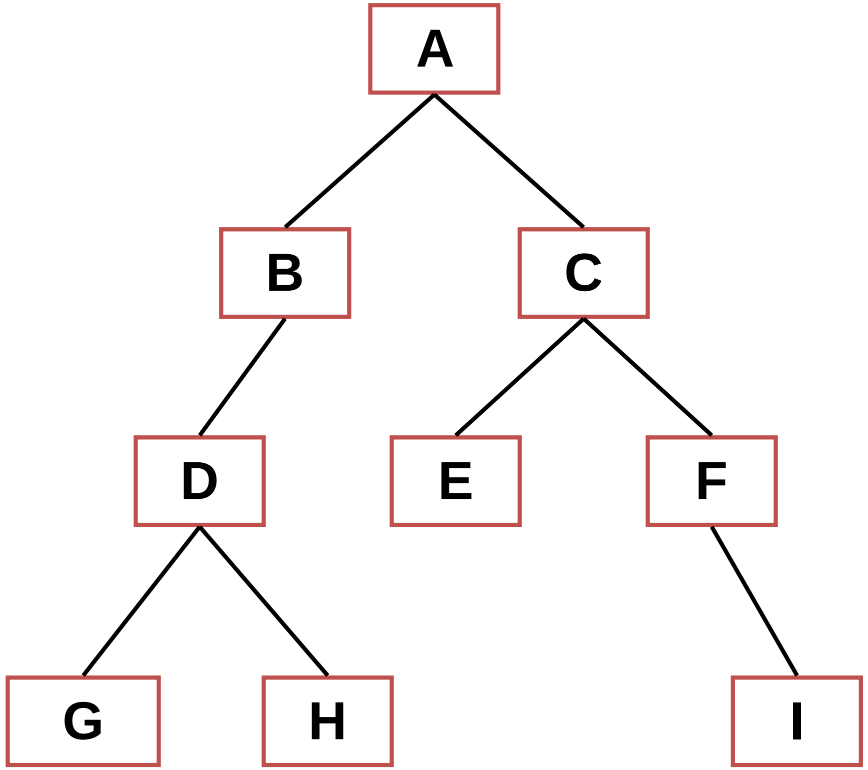
```
Depth_First_Search (VERTEX V)
{
Visit V;

Set the visit flag for the vertex V to TRUE;

For all adjacent vertices Vi (i = 1, 2, ….., n) of V
        if (Vi has not been previously visited)
                Depth_First_Search (Vi)
}
```

A B C D E F G H

- **Breadth First Search** (BFS)
  - Very similar to level-order traversal of a binary tree (left, node, right)

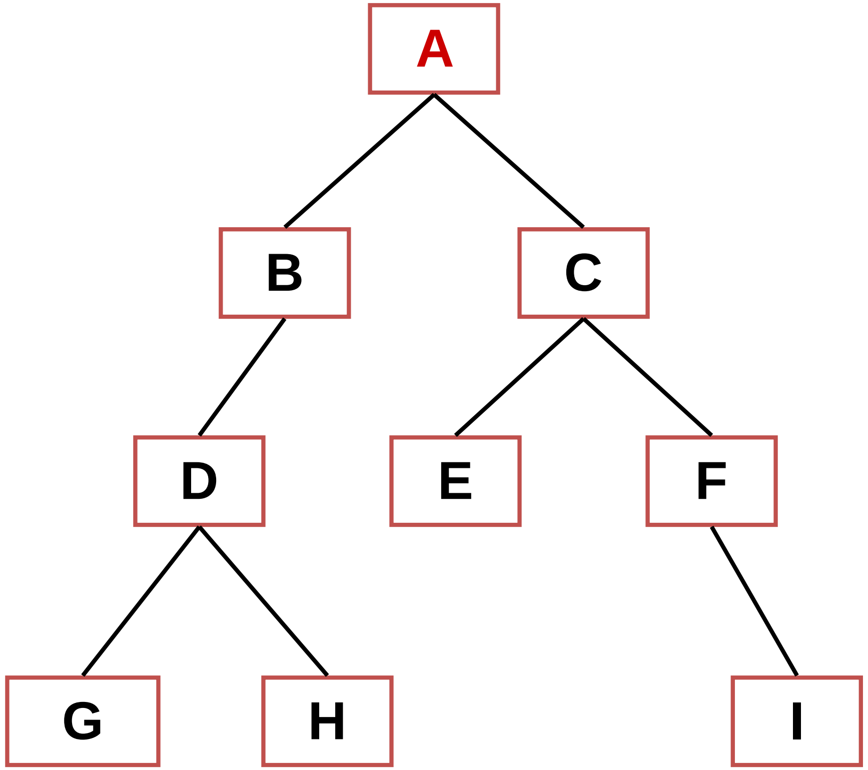
  - Use a *queue* to track unvisited nodes
  - For each node that is deleted from the queue,
    - add each of its children to the queue
    - until the queue is empty

Level-Order



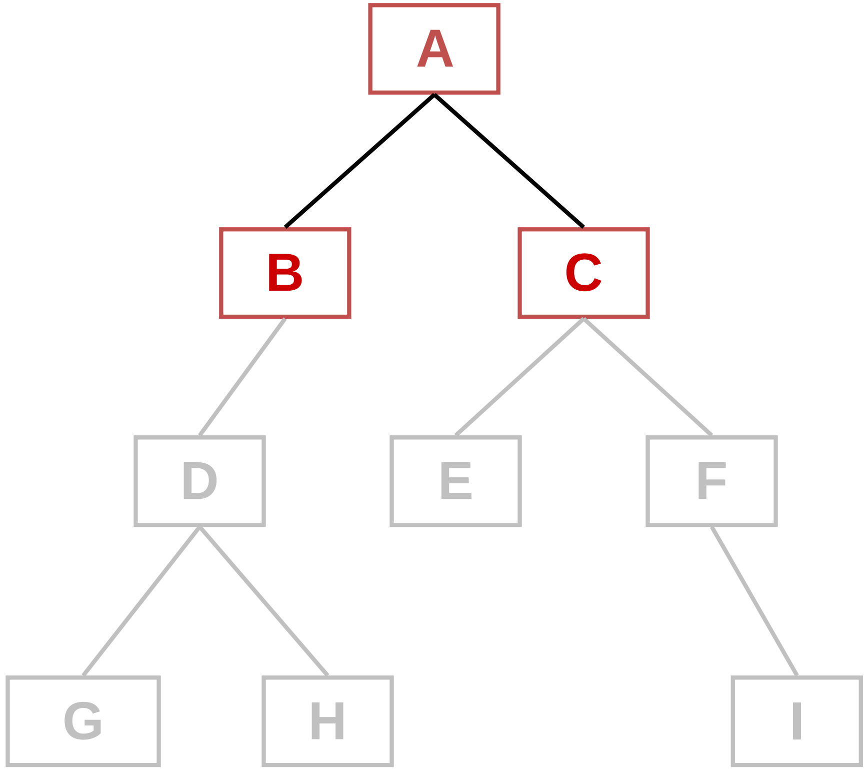
Queue Output

Level-Order



	Queue	Output
Init	[A]	-

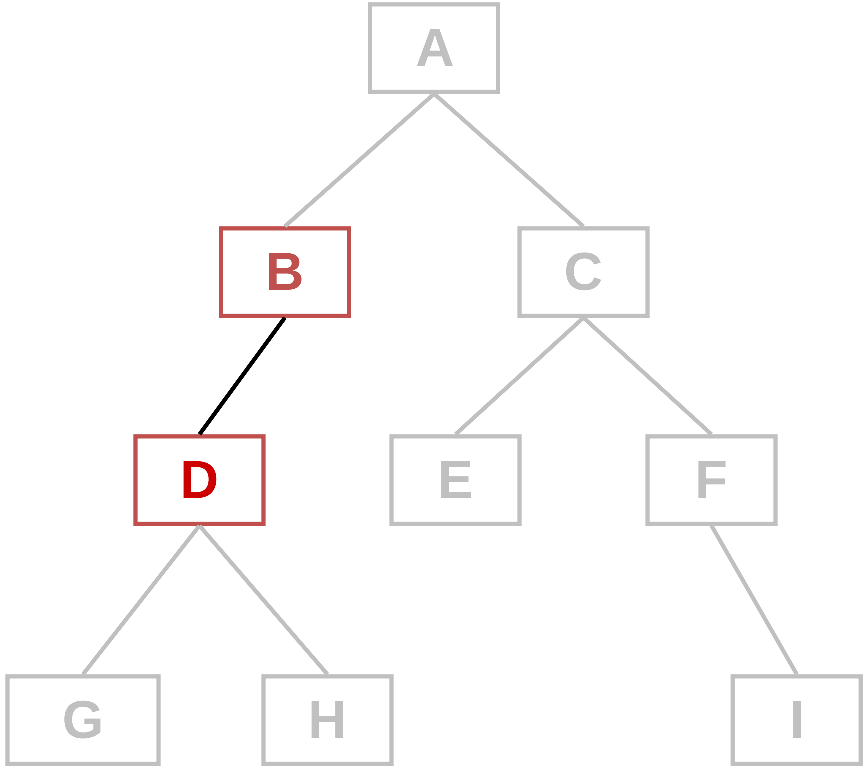
Level-Order



	Queue	Output
Init	[A]	-
Step 1	[B,C]	A

Dequeue A
Print A
Enqueue children of A

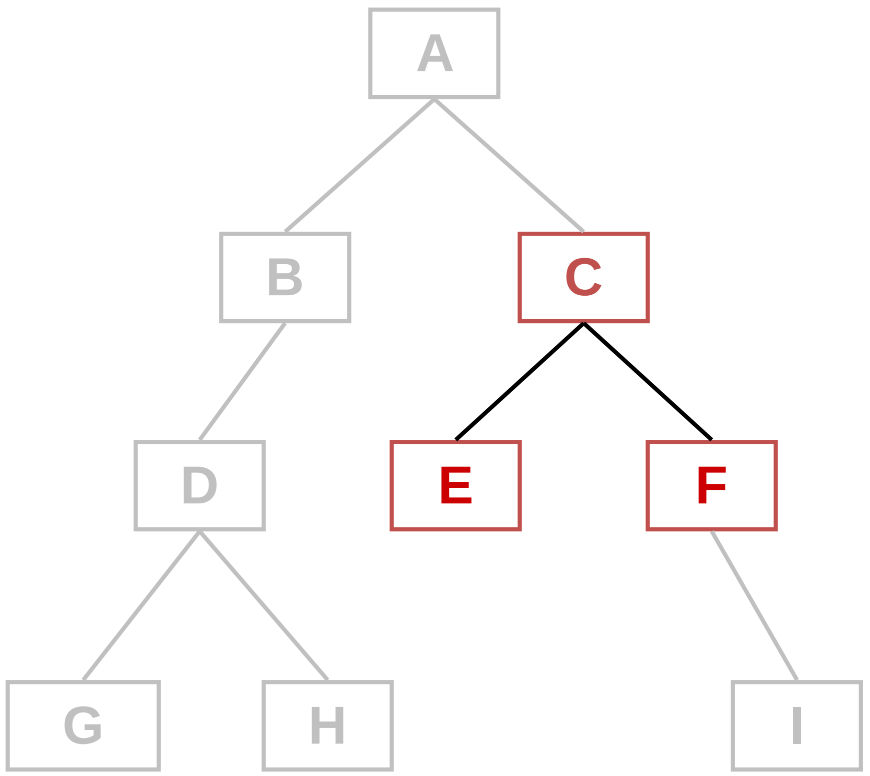
Level-Order



	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A **B**

Dequeue B
Print B
Enqueue children of B

Level-Order

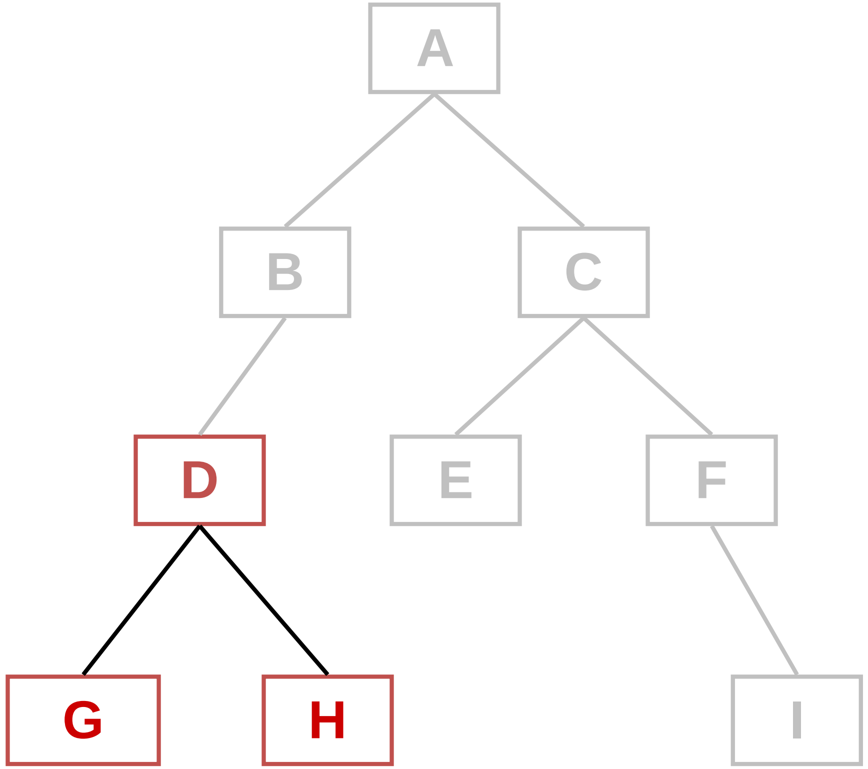


	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A B
Step 3	[D,E,F]	A B **C**

Dequeue C
Print C
Enqueue children of C

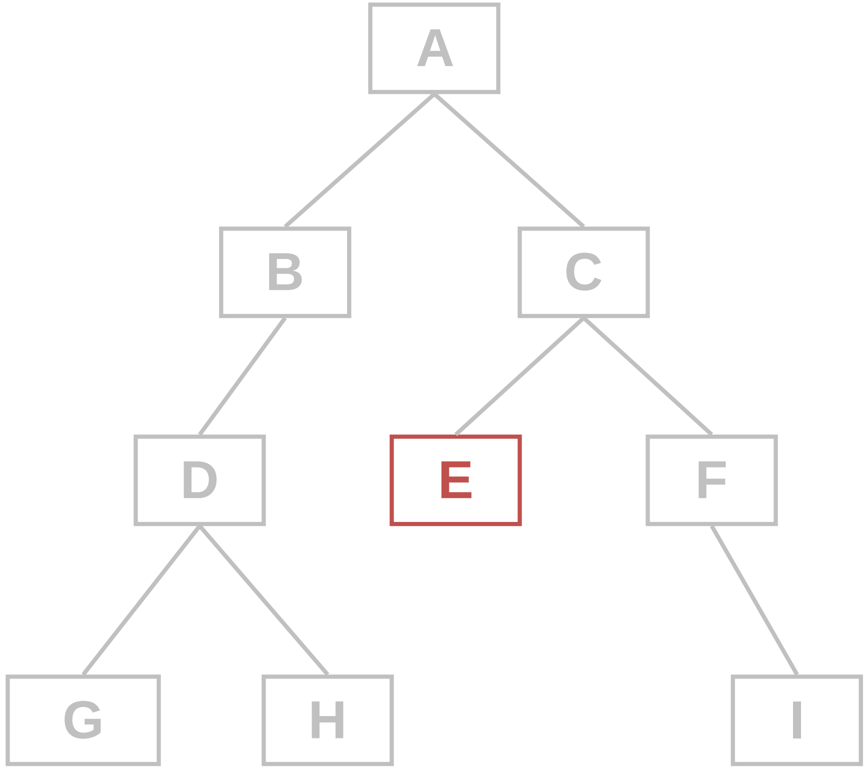
…

Level-Order



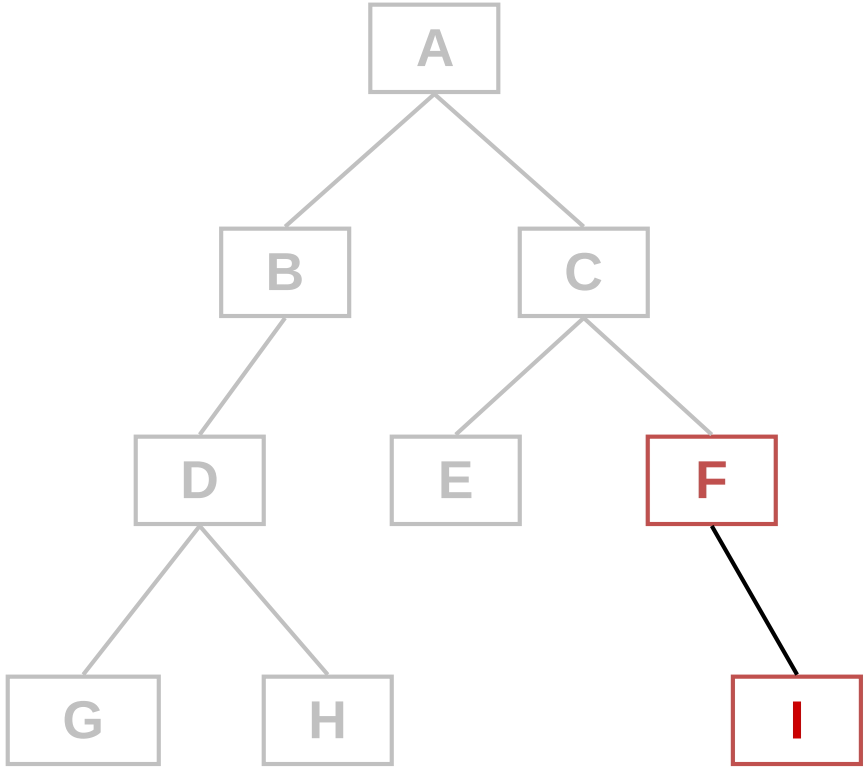
	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A B
Step 3	[**D**,E,F]	A B C
Step 4	[E,F,**G**,**H**]	A B C **D**

Level-Order



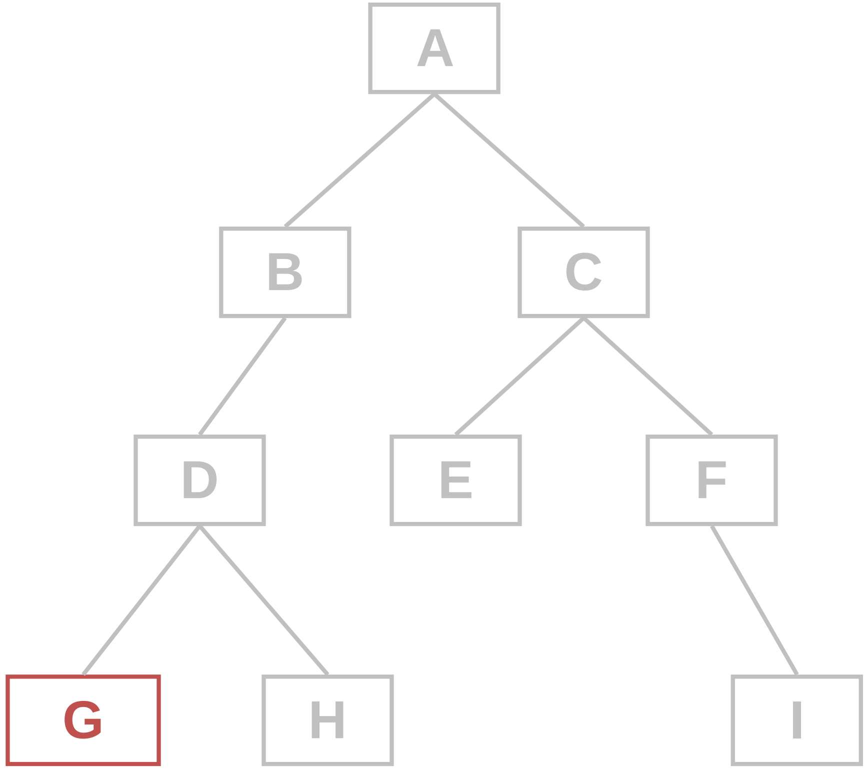
	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A B
Step 3	[D,E,F]	A B C
Step 4	[E,F,G,H]	A B C D
Step 5	[F,G,H]	A B C D **E**

Level-Order



	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A B
Step 3	[D,E,F]	A B C
Step 4	[E,F,G,H]	A B C D
Step 5	[**F**,G,H]	A B C D E
Step 6	[G,H,**I**]	A B C D E **F**

Level-Order



	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A B
Step 3	[D,E,F]	A B C
Step 4	[E,F,G,H]	A B C D
Step 5	[F,G,H]	A B C D E
Step 6	[G,H,I]	A B C D E F
Step 7	[H,I]	A B C D E F **G**

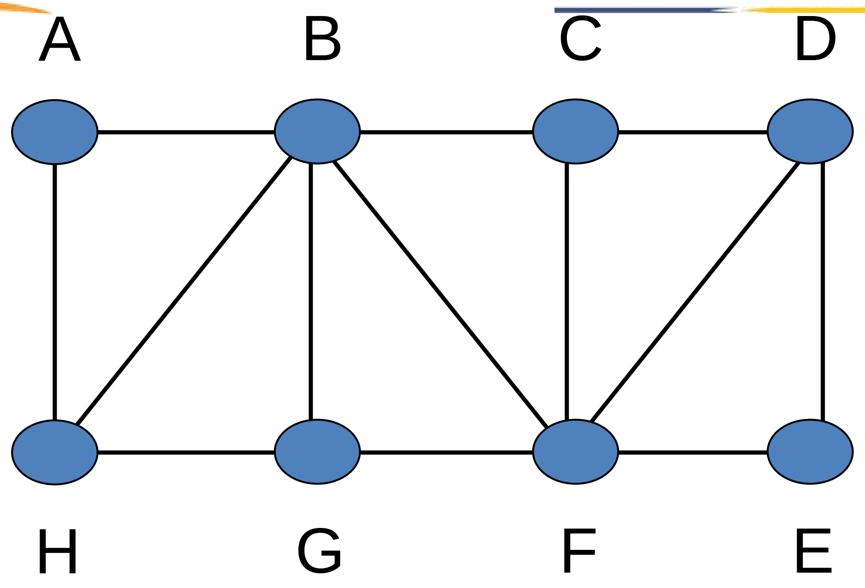
- **Algorithm**

```
bfs(v)   { /* v is the starting vertex */
   push v into an empty queue Q;
   while Q is not empty do
            v = delete(Q);
            if v is not visited {
                        mark v as visited;
                        push v's neighbors into Q;
            }
   }
}
```

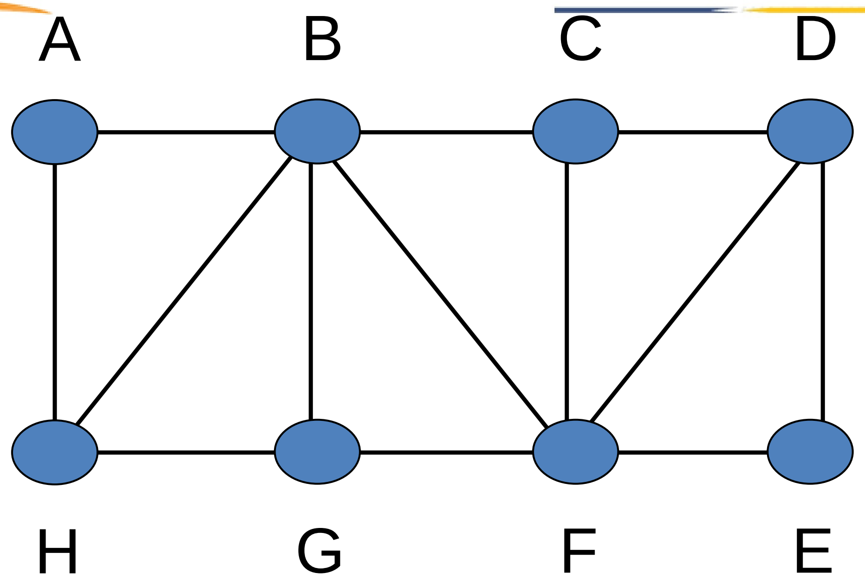
- **Time is O(*e*) for adjacency lists and O(n^2) for adjacency matrices**

```
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
void addq(queue_pointer *, queue_pointer *, int);
int deleteq(queue_pointer *);
```

```
void bfs(int v){
        node_pointer w;
        queue_pointer front, rear;
        front = rear = NULL;
        printf("%5d", v);
        visited[v] = TRUE;
        addq(&front, &rear, v);
        while (front) {
                v = deleteq(&front);
                for (w=graph[v]; w; w=w->link)
                        if (!visited[w->vertex]) {
                                printf("%5d", w->vertex);
                                addq(&front, &rear, w->vertex);
                                visited[w->vertex] = TRUE;
                        }
        }
}
```



Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

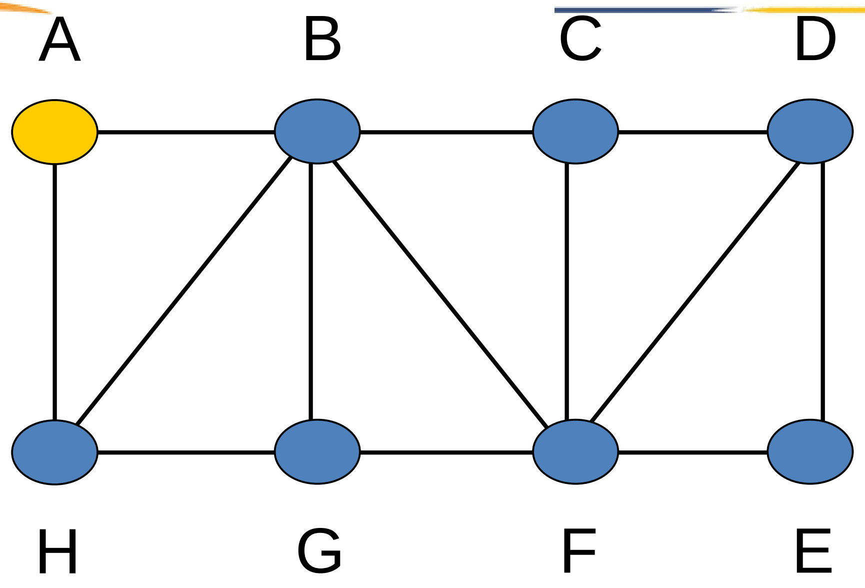


Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [F, F, F, F, F, F, F, F]

Q: [ ]

v:



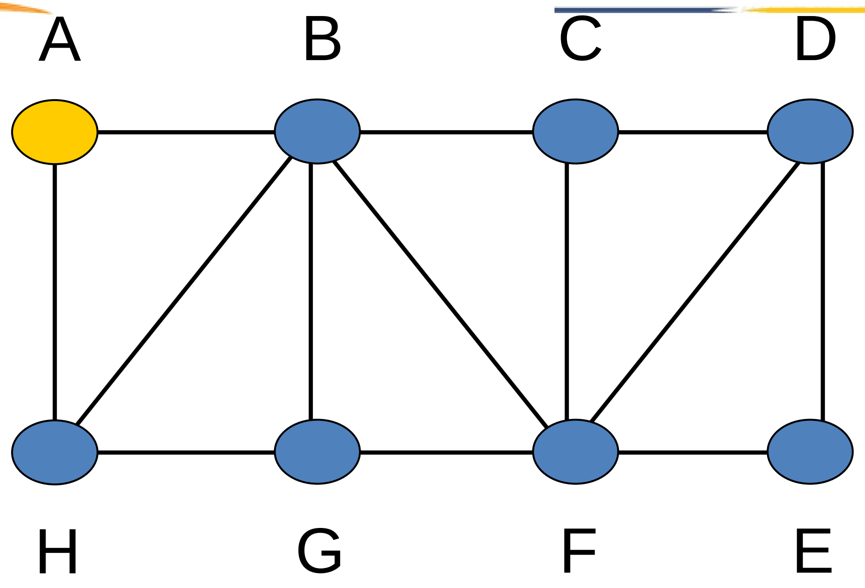
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [**T**, F, F, F, F, F, F, F]

Q: [A]

v:

A



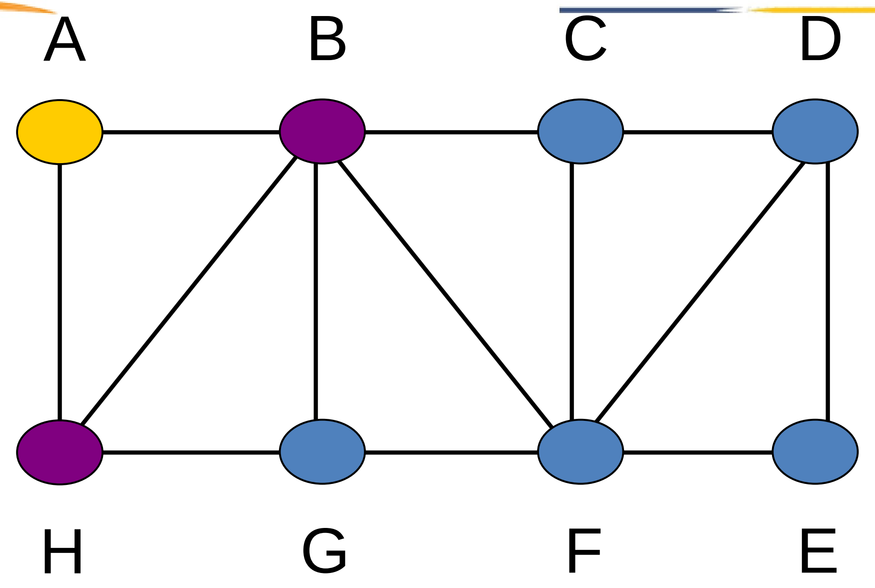
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, F, F, F, F, F, F, F]

Q: [ ]

v: A

A



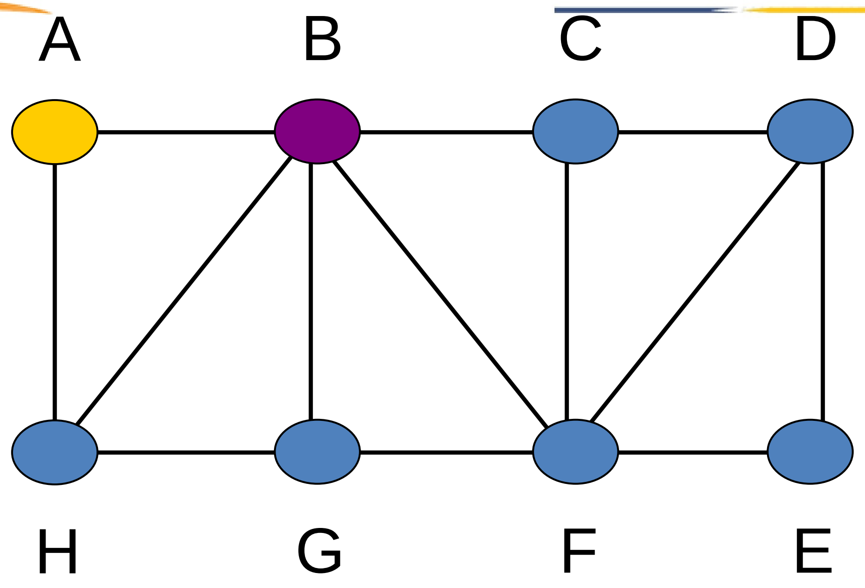
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, F, F, F, F, F, F, F]

Q: [ ]

v: **A → B, H**

A



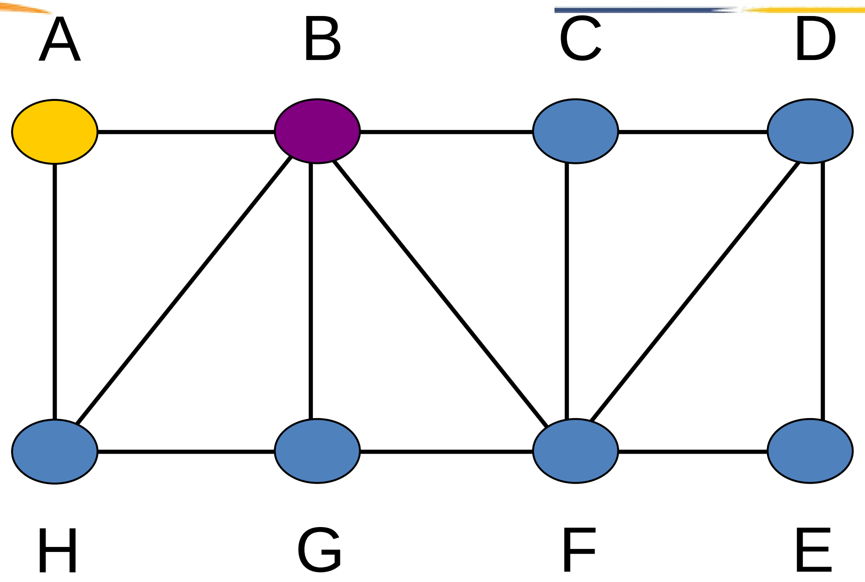
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, F, F, F, F, F, F, F]

Q: [ ]

v: A → **B**, H

A



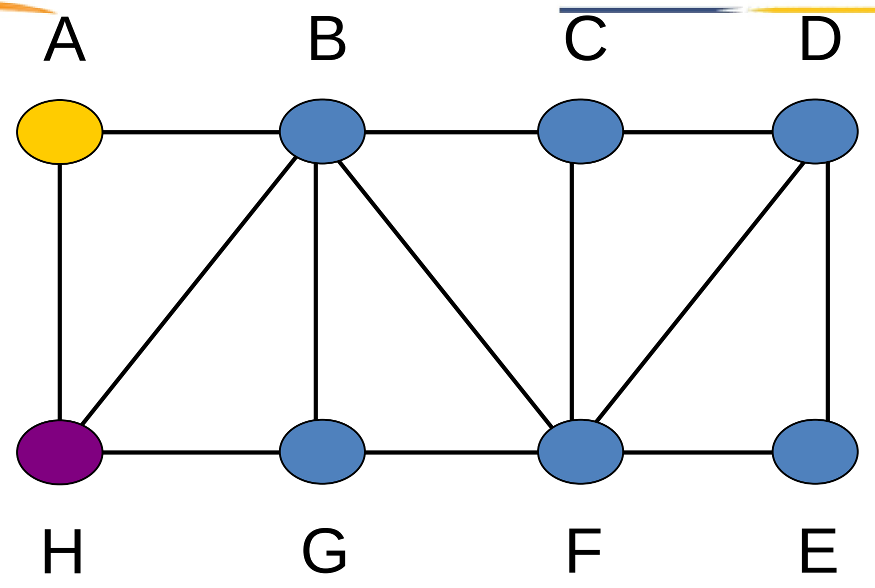
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, **T**, F, F, F, F, F, F]

Q: [**B**]

v: A → **B**, H

A **B**



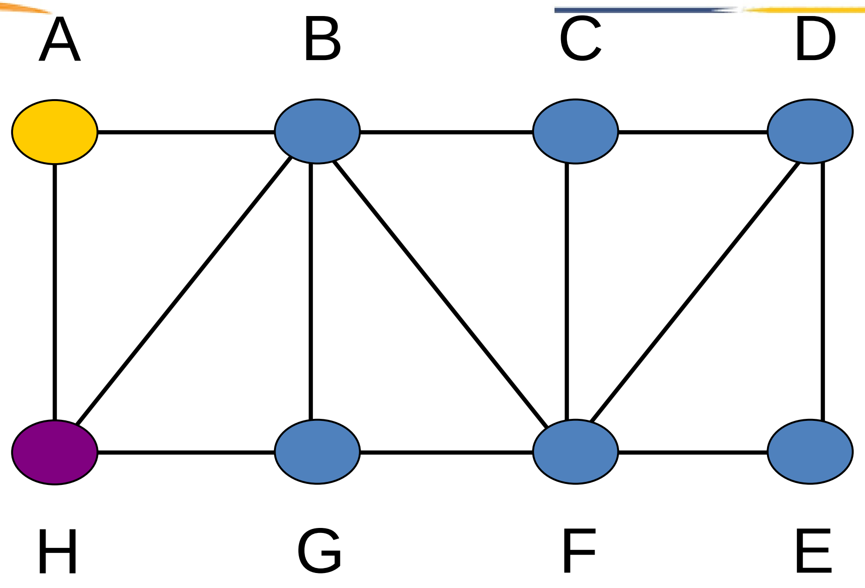
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, F, F, F, F, F, F]

Q: [B]

v: A → B, H

A B



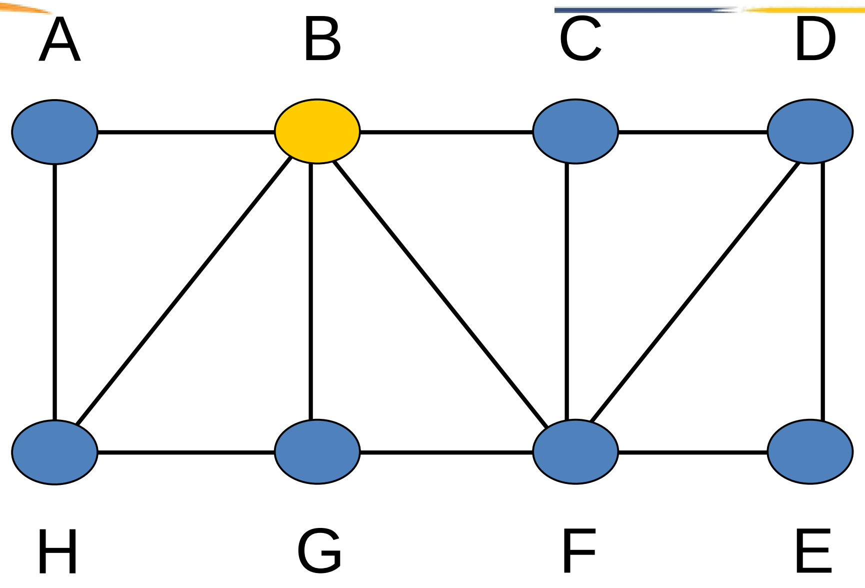
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, F, F, F, F, F, **T**]

Q: [B, **H**]

v: A → B, **H**

A B **H**



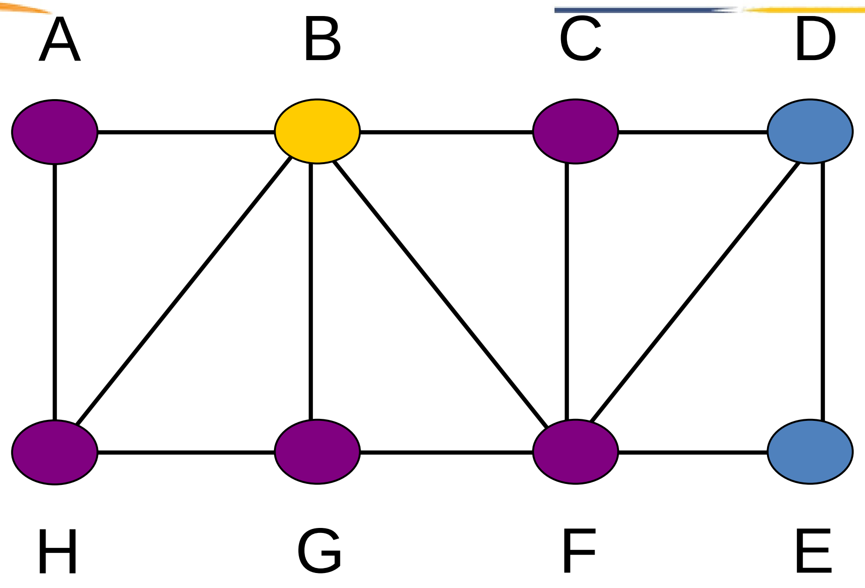
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, F, F, F, F, F, T]

Q: [H]

v: **B**

A B H



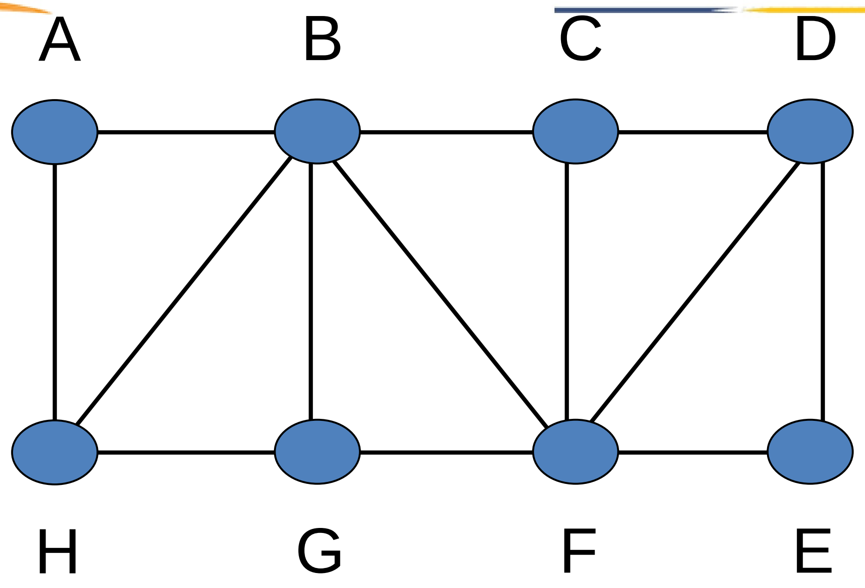
Vertex		Adjacent Vertices
A	→	B, H
B	→	**A, C, G, F, H**
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, F, F, F, F, F, T]

Q: [H]

v: **B → A, C, G, F, H**

A B H



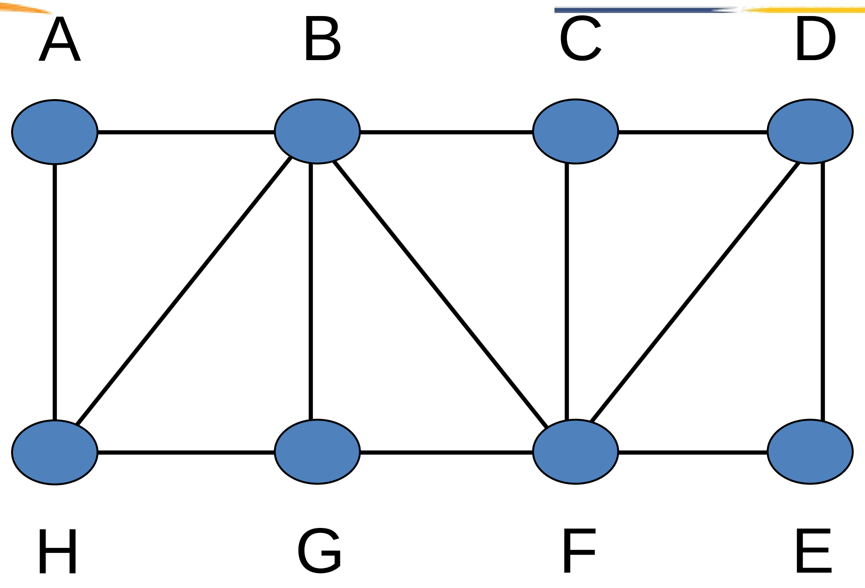
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, F, F, F, F, F, T]

Q: [H]

v: B → A, C, G, F, H

A B H



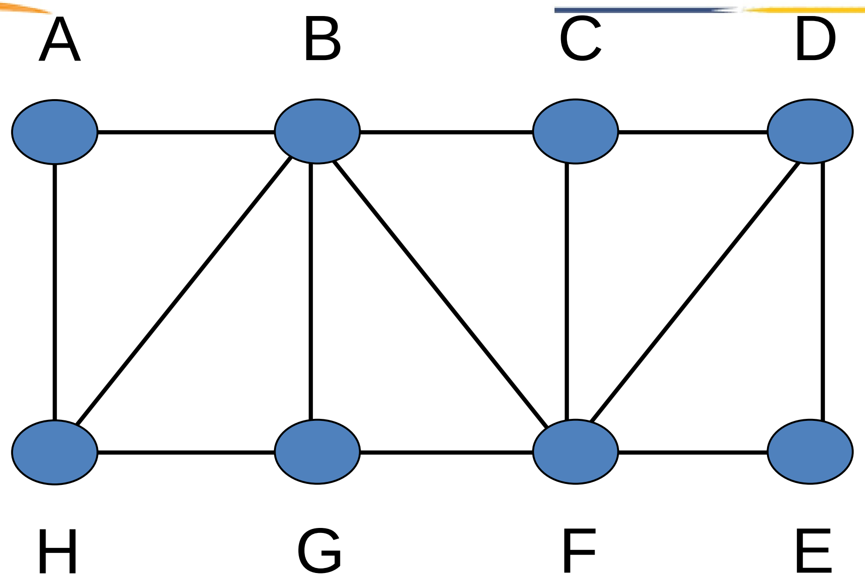
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, F, F, F, F, F, T]

Q: [H]

v: B → A, C, G, F, H

A B H



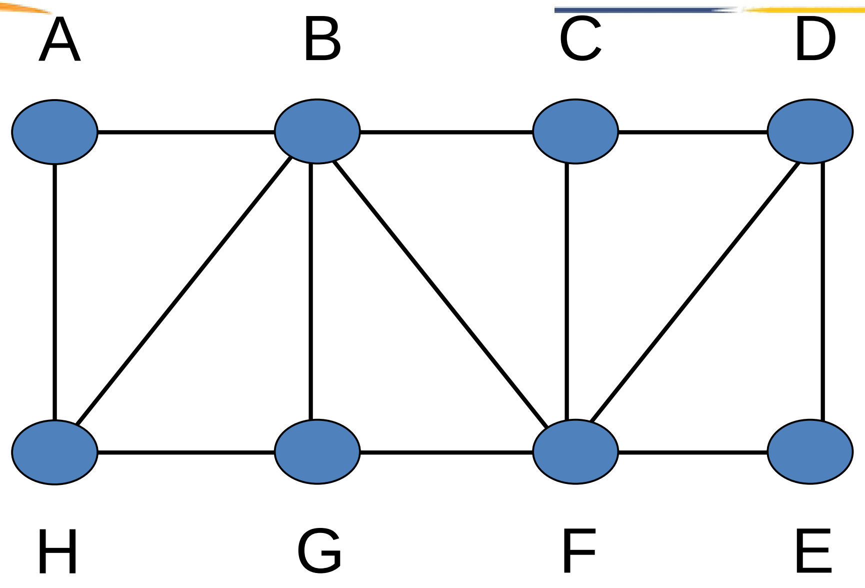
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, F, F, F, F, T]

Q: [H, C]

v: B → A, C, G, F, H

A B H C



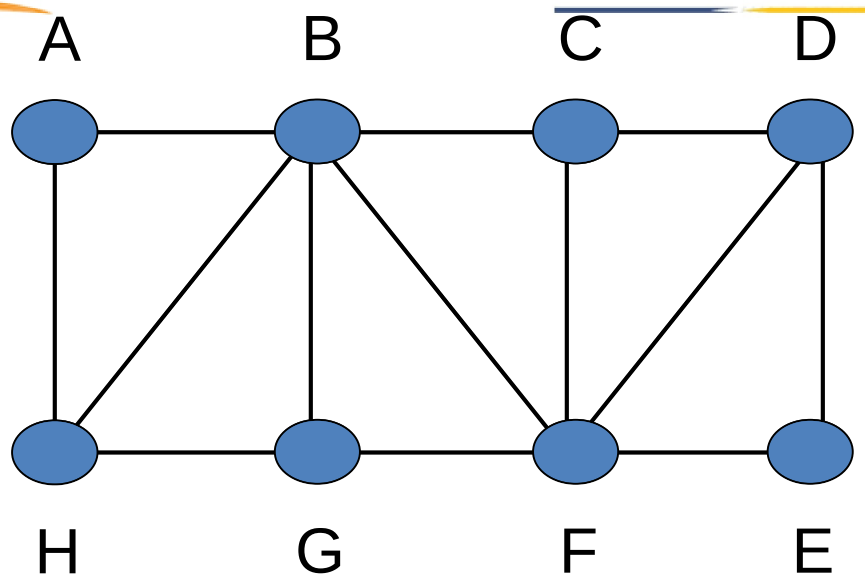
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, T, F, F, F, F, T]

Q: [H, C]

v: B → A, C, G, F, H

A B H C



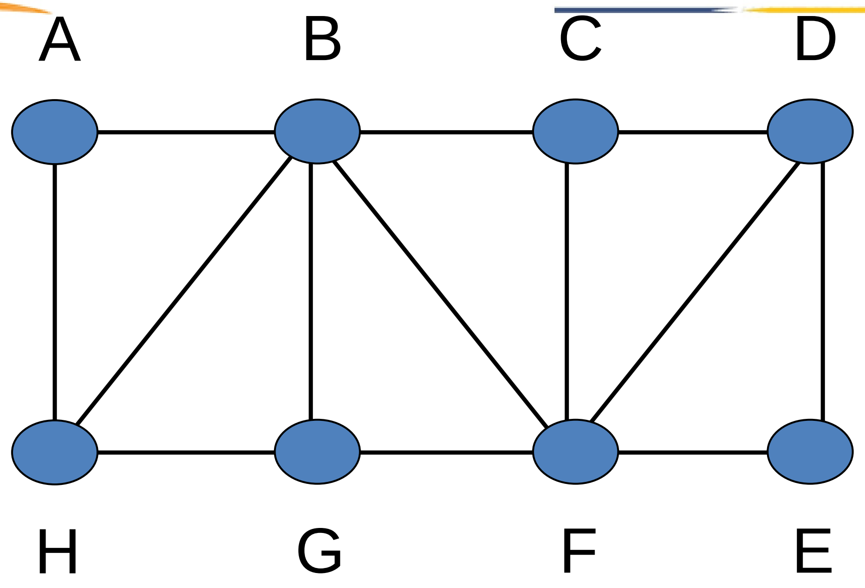
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, F, F, F, T, T]

Q: [H, C, G]

v: B → A, C, G, F, H

A B H C G



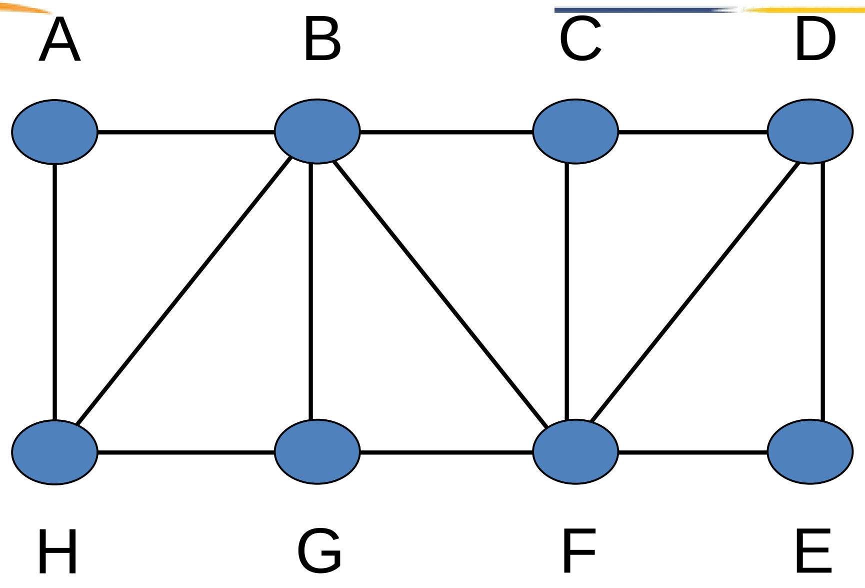
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, T, F, F, F, T, T]

Q: [H, C, G]

v: B → A, C, G, F, H

A B H C G



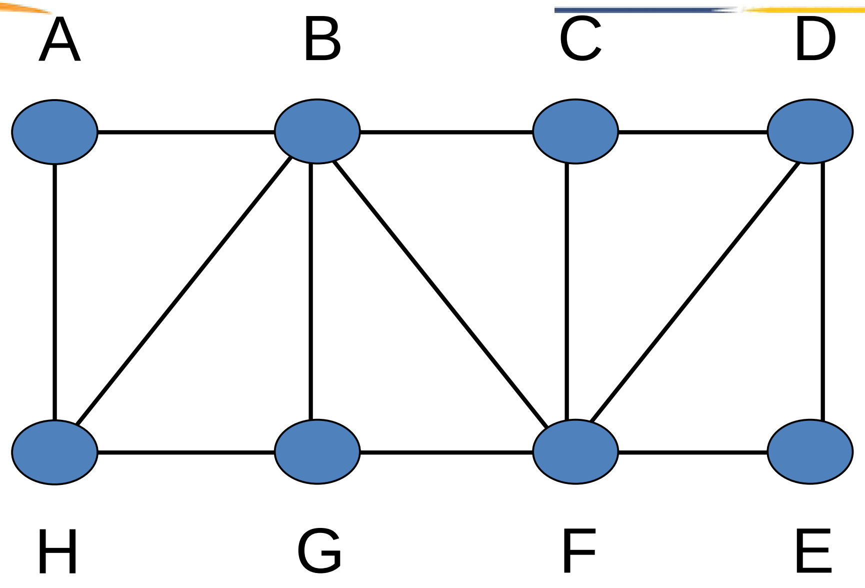
Vertex	Adjacent Vertices
A →	B, H
B →	A, C, G, F, H
C →	B, D, F
D →	C, E, F
E →	D, F
F →	B, C, D, E, G
G →	B, F, H
H →	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [H, C, G, F]

v: B → A, C, G, F, H

A B H C G F



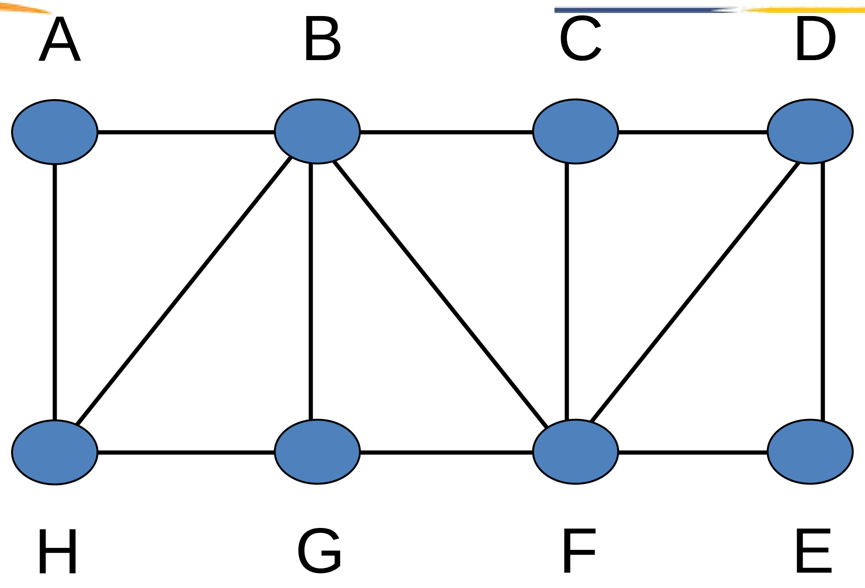
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [H, C, G, F]

v: B → A, C, G, F, H

A B H C G F



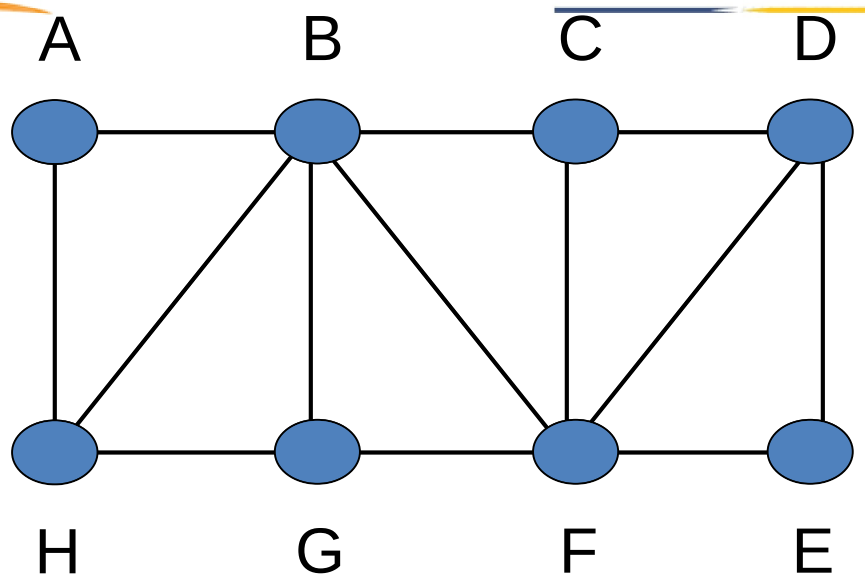
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [C, G, F]

v: H

A B H C G F



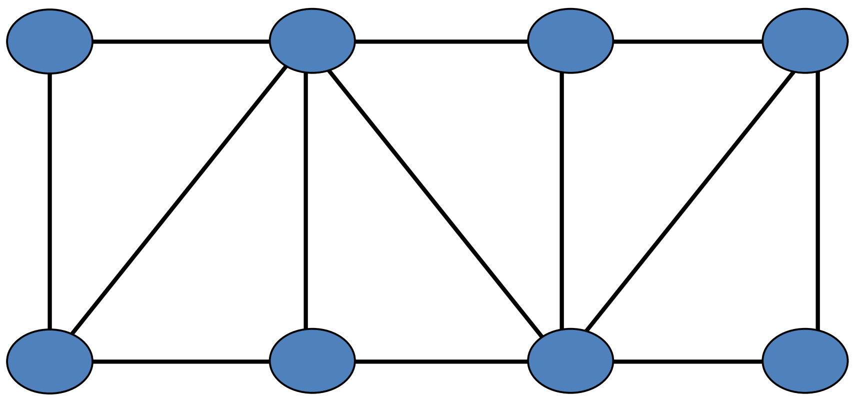
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [C, G, F]

v: H → A, B, G

A B H C G F



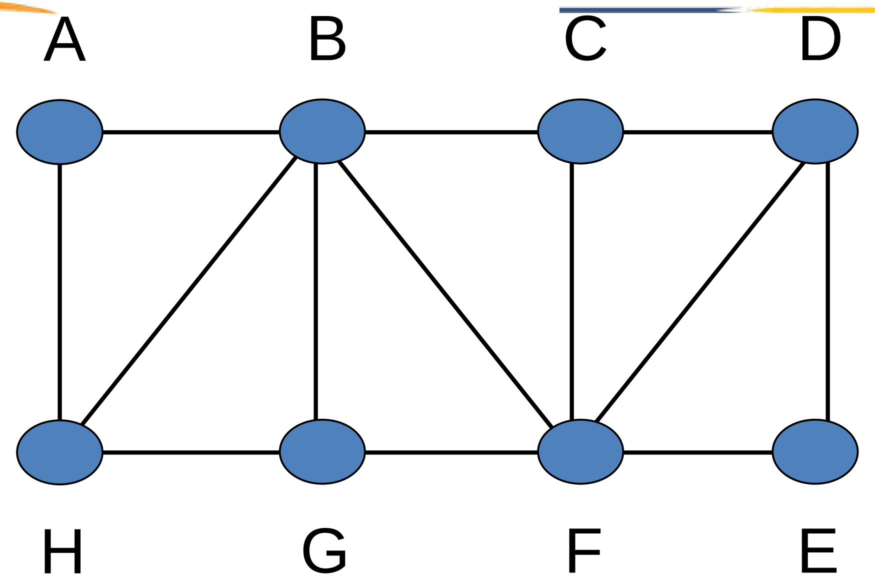
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [C, G, F]

v: H → A, B, G

A B H C G F



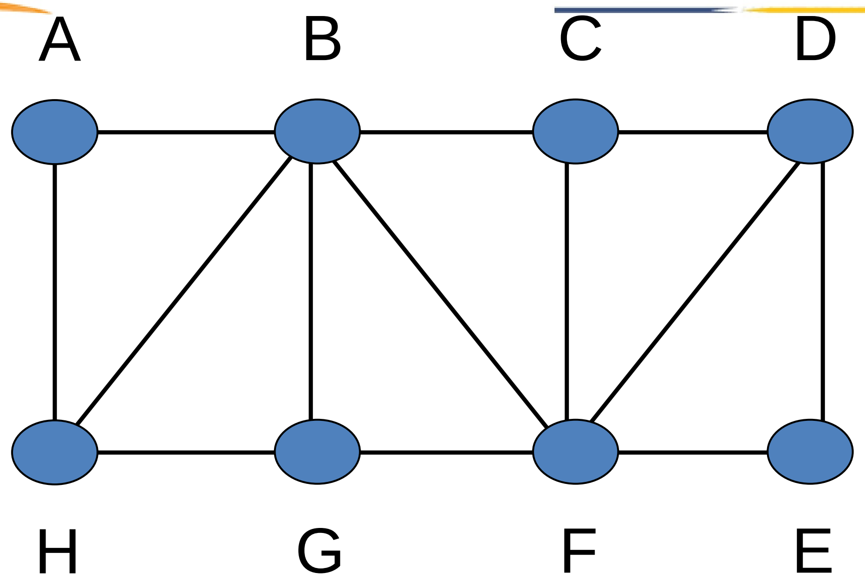
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [C, G, F]

v: H → A, B, G

A B H C G F



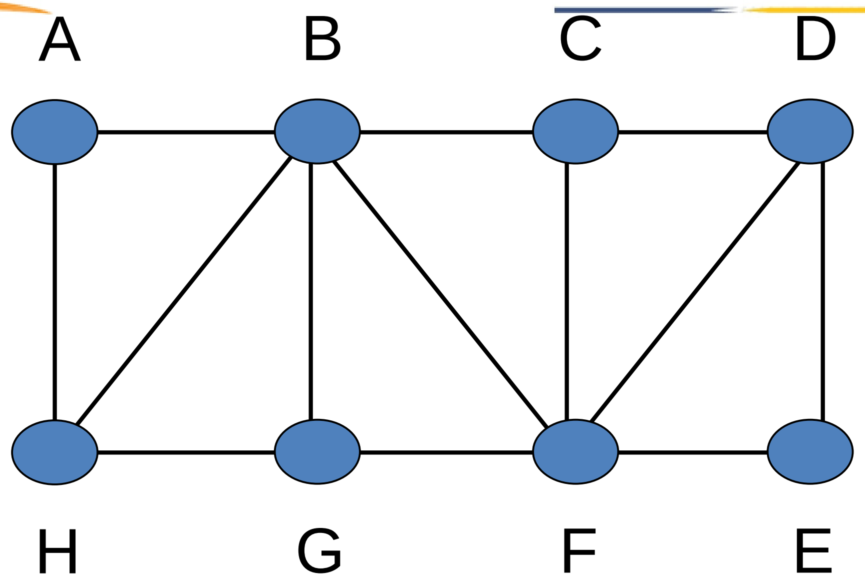
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [C, G, F]

v: H → A, B, G

A B H C G F



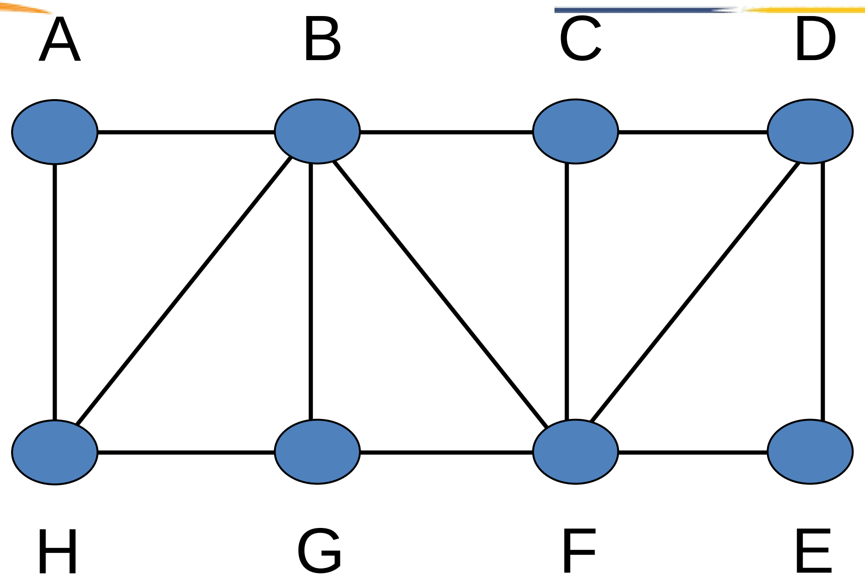
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [C, G, F]

v:

A B H C G F



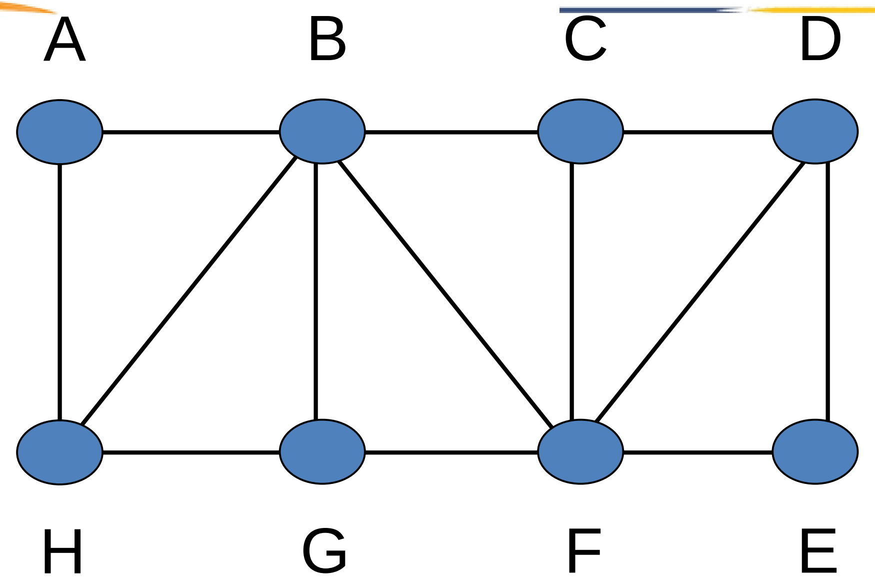
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [G, F]

v: C → B, D, F

A B H C G F



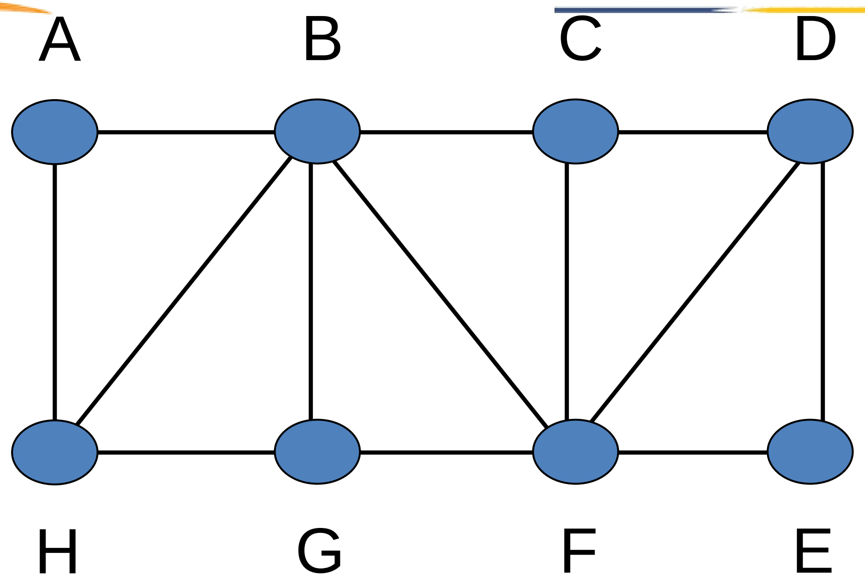
Vertex		Adjacent Vertices
A	⟶	B, H
B	⟶	A, C, G, F, H
C	⟶	B, D, F
D	⟶	C, E, F
E	⟶	D, F
F	⟶	B, C, D, E, G
G	⟶	B, F, H
H	⟶	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [G, F]

v: C → B, D, F

A B H C G F



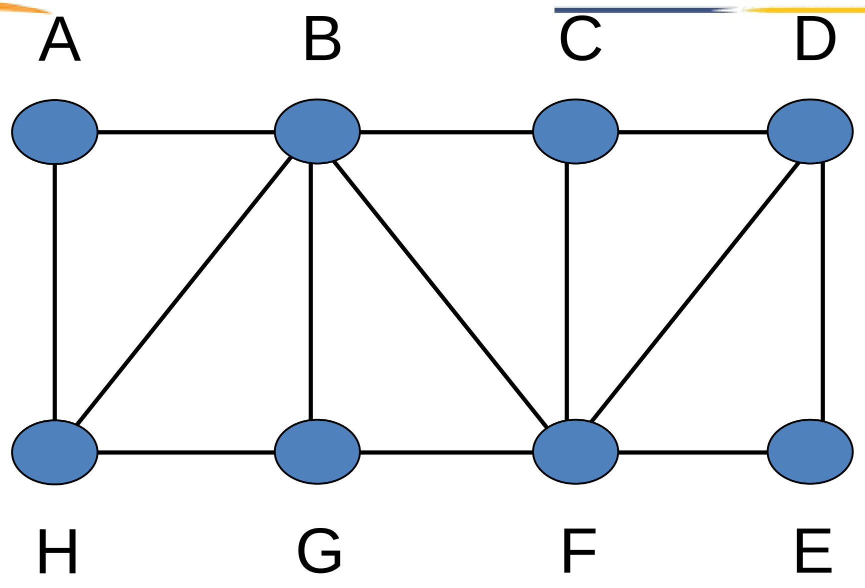
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, F, F, T, T, T]

Q: [G, F]

v: C → B, D, F

A B H C G F



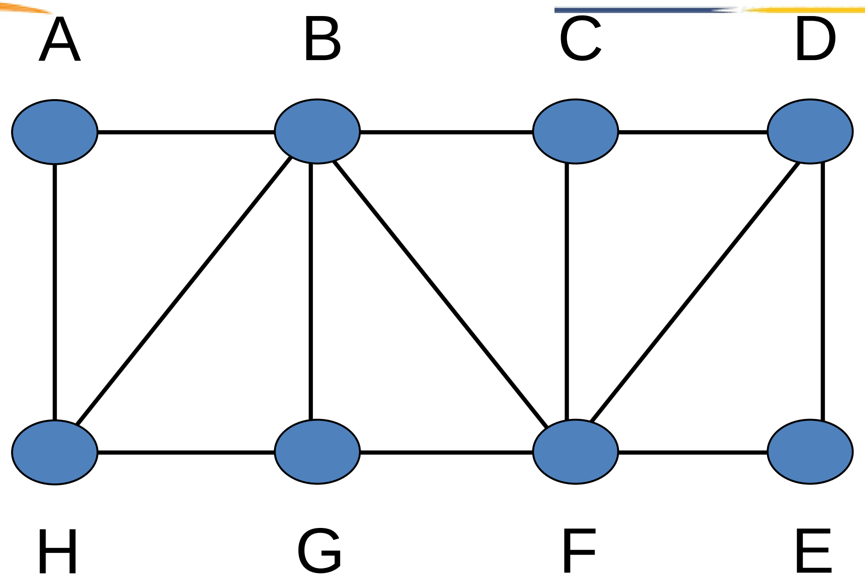
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, T, F, T, T, T]

Q: [G, F, D]

v: C → B, D, F

A B H C G F D



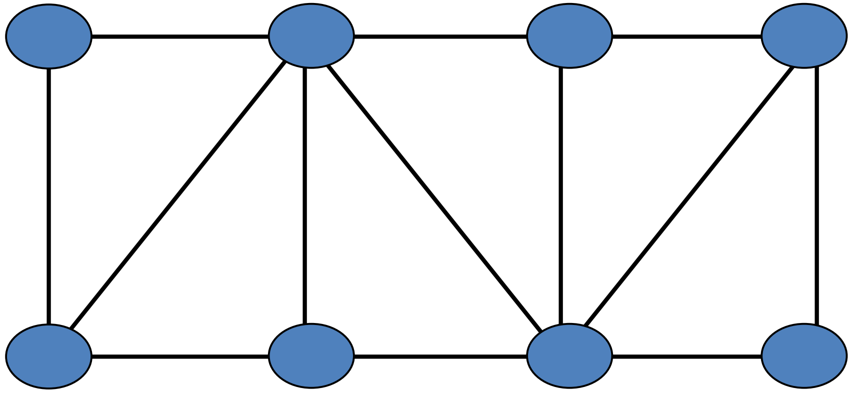
Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, T, F, T, T, T]

Q: [G, F, D]

v: C → B, D, F

A B H C G F D



Vertex		Adjacent Vertices
A	→	B, H
B	→	A, C, G, F, H
C	→	B, D, F
D	→	C, E, F
E	→	D, F
F	→	B, C, D, E, G
G	→	B, F, H
H	→	A, B, G

Visit [T, T, T, T, T, T, T, T]

Q: [ ]

v:

A B H C G F D E

SUMMARY

- Terminology
- Graph Representations
- Graph Traversals

Enjoy the Course…!