People's Democratic Republic of Algeria
Ministry of higher education and scientific research
University of Mohamed Boudiaf - M'sila

Faculty of Mathematics and Computer Science
Department of Computer Science

2nd Year License (2L)

Data Structures and Algorithms 3 (DSA3)

Semester : 03
2025/2026

Directed by

Dr. DABBA ALI

Contact us

alidabba@gmail.com

ali.dabba@univ-msila.dz

- In case of problems or difficulties, please contact me or your TD / TP teacher.
- We are at your disposal to help you

CHAPTER 5

Graphs and Elementary Graph Algorithms

Plan

I. Introduction

II. Graph Definition

III. Graph Terminology

IV. Graph Representations

V. Graph Traversal

I. Introduction

- In the real world, many problems are represented in terms of objects and the relationships among them.
- For example, in an airline route map, we might be interested in questions like: “**What’s the fastest way to go from Alger to M'sila?**” or “**What is the cheapest way to go from Alger to M'sila?**”.

I. Introduction

- To answer these questions, we need information about **connections** (airline routes) **between objects** (towns).
- Graph technologies and analytics provide **powerful tools** for connected data that are used to solve these types of problems.
- In this chapter, we consider how graphs can be represented, and take a look at algorithms for some basic operations such as searching and traversal.

II. Graph Definition

- Graph is a Non-Linear data structure comprising **a finite set of nodes and edges.**
- Generally, a **G** graph is a pair $\boldsymbol{G = (V, E)}$, where $\boldsymbol{V = \{v_1, v_2, \cdots, v_n\}}$ is a finite set of nodes, called **vertices**, and $\boldsymbol{E = \{e_1, e_2, \cdots, e_m\}}$ is a finite set of collections of vertex pairs, called **edges**.
  - We will often denote $\boldsymbol{n} = |\boldsymbol{V}|$, and $\boldsymbol{e} = |\boldsymbol{E}|$.
  - Vertices and edges are positions and storage elements.

II. Graph Definition

- Graph $\boldsymbol{G}$ can be defined as $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ where :

  - $\boldsymbol{V} = \{\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}, \boldsymbol{E}\}$, and

  - $\boldsymbol{E} = \{\{\boldsymbol{A}, \boldsymbol{B}\}, \{\boldsymbol{A}, \boldsymbol{C}\}, \{\boldsymbol{A}, \boldsymbol{D}\},$

    $\{\boldsymbol{B}, \boldsymbol{D}\}, \{\boldsymbol{C}, \boldsymbol{D}\}, \{\boldsymbol{B}, \boldsymbol{E}\}, \{\boldsymbol{E}, \boldsymbol{D}\}\}.$

II. Graph Definition

- A graph is generally displayed as figure 5.1, in which the **vertices are represented by circles** and the **edges by lines**.

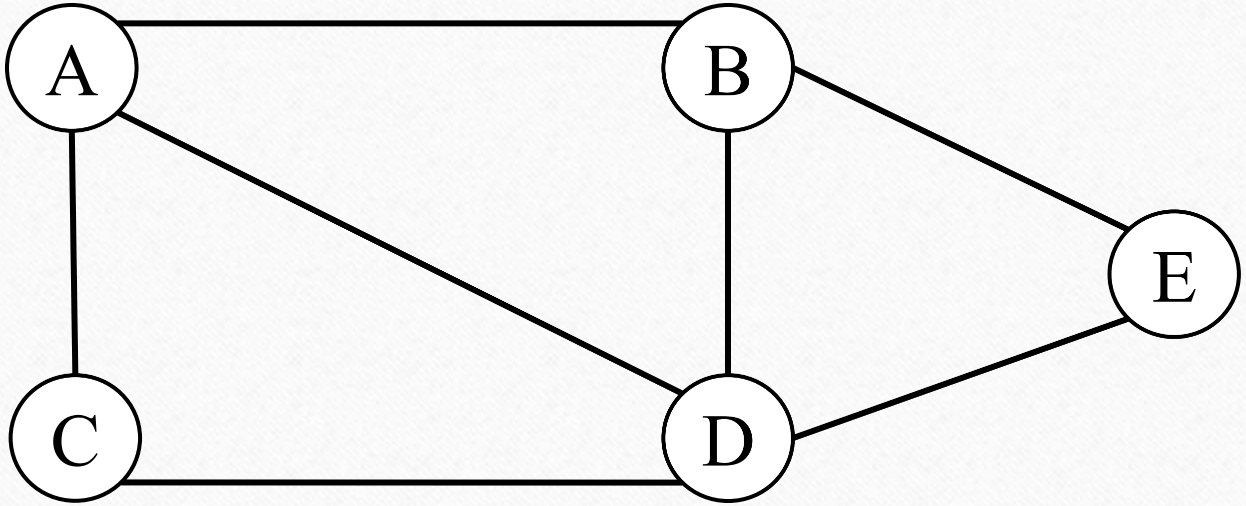
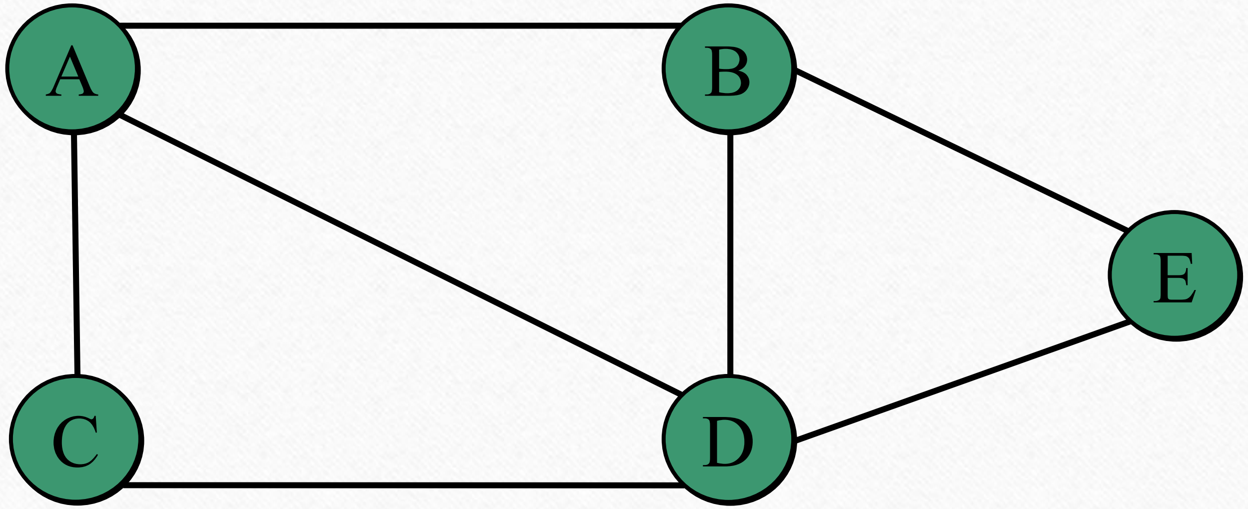


Figure 5.1 : A Graph.

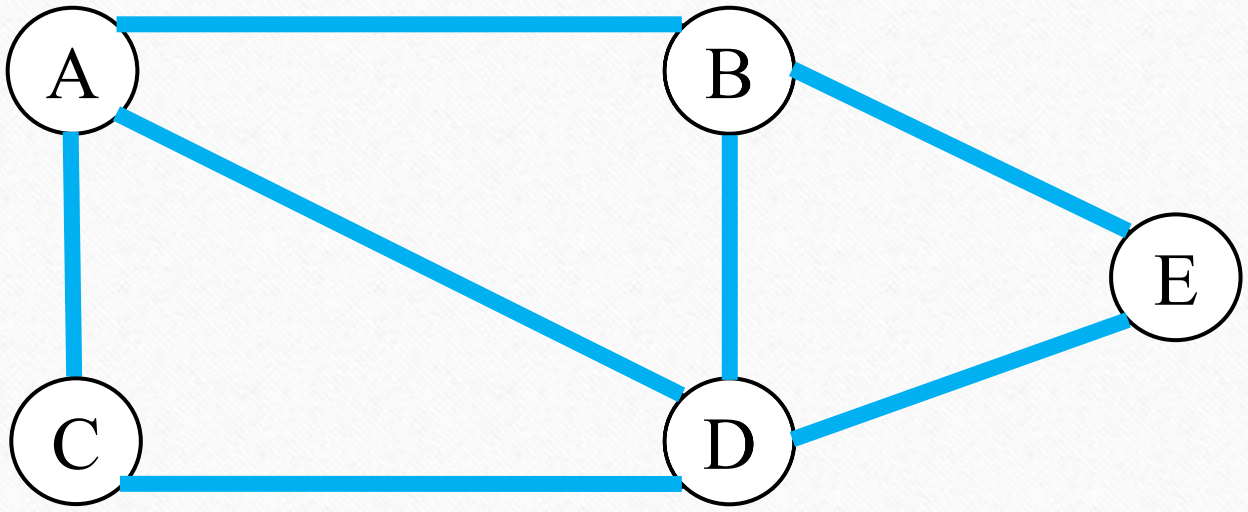
III. Graph Terminology

- **Vertex** : Individual data element of a graph is called as Vertex. **Vertex** is also known as **node.**
- In above example graph, $\boldsymbol{A, B, C, D}$, and $\boldsymbol{E}$ are known as vertices.



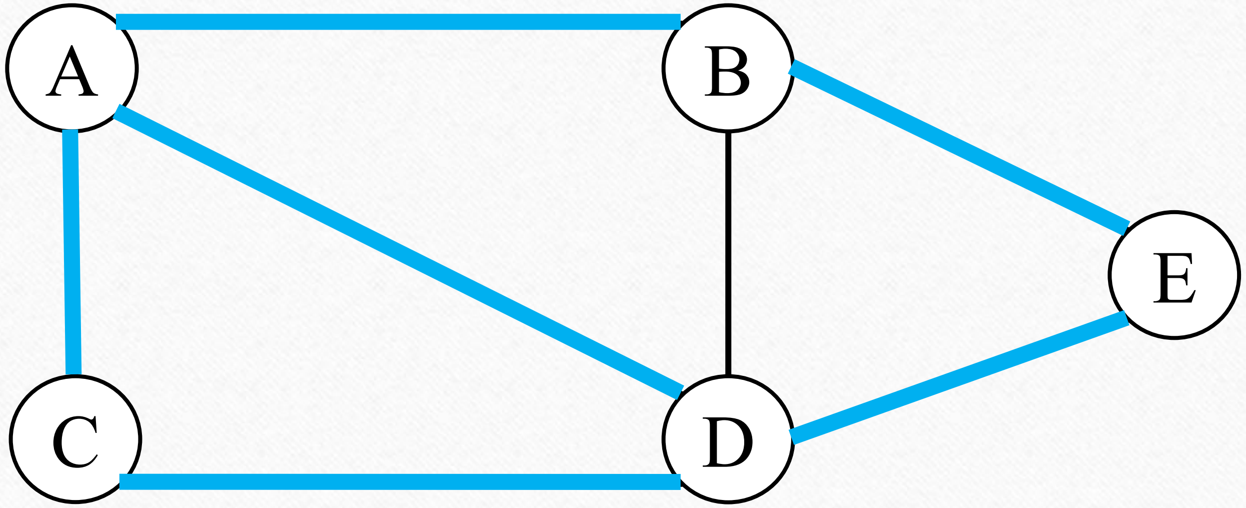
III. Graph Terminology

- **Edge** : An edge is a connecting link between two vertices.
- Edges are three types :



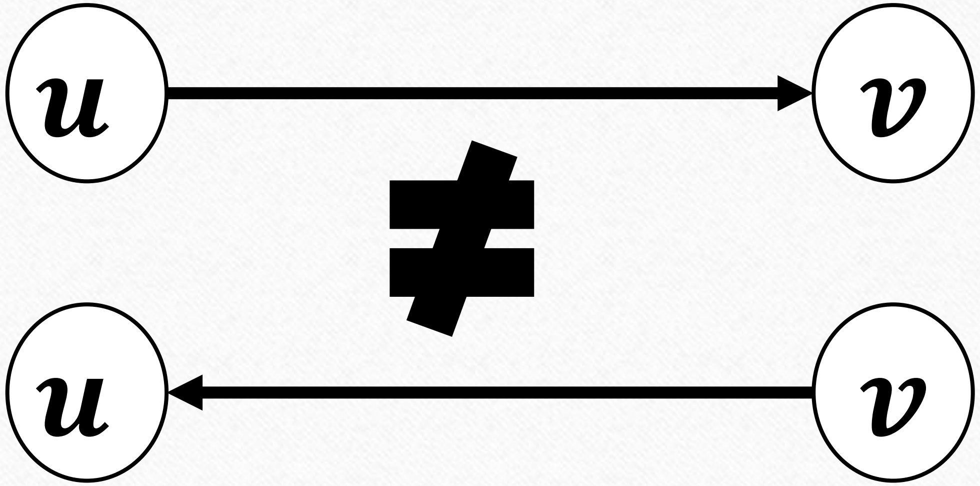
III. Graph Terminology

- **Directed Edge :** A directed edge is a unidirectional edge. If there is directed edge between vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ then edge $(\boldsymbol{u}, \boldsymbol{v})$ is not equal to edge $(\boldsymbol{v}, \boldsymbol{u})$.
-



III. Graph Terminology

- **Directed Edge :** A directed edge is a unidirectional edge.
- If there is directed edge between vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ then edge $(\boldsymbol{u}, \boldsymbol{v})$ is not equal to edge $(\boldsymbol{v}, \boldsymbol{u})$.

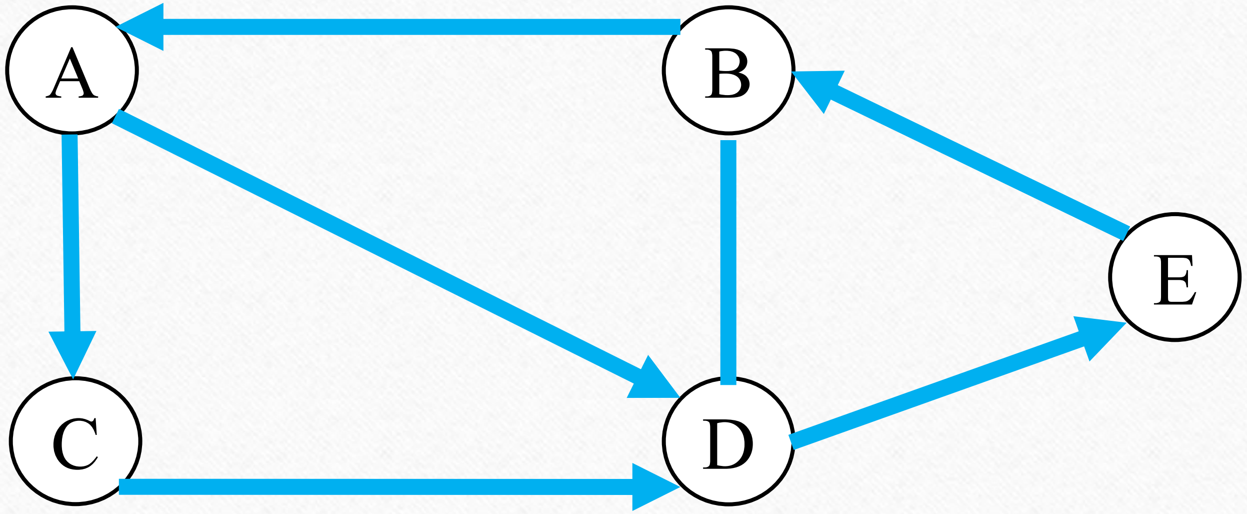


III. Graph Terminology

- **Directed Edge**
  - ordered pair of vertices $(\boldsymbol{u}, \boldsymbol{v}) \neq (\boldsymbol{v}, \boldsymbol{u})$
  - first vertex $\boldsymbol{u}$ is the **origin**
  - second vertex $\boldsymbol{v}$ is the **destination**
  - **Example:** one-way road traffic

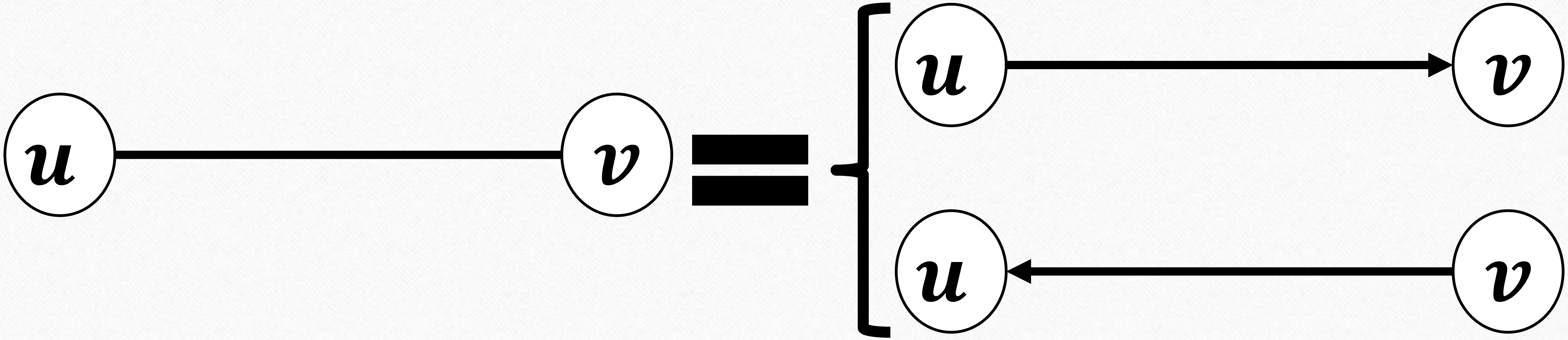
III. Graph Terminology

➢ **Directed Edge :** A directed edge is a unidirectional edge.



III. Graph Terminology

- **Undirected Edge** : An undirected edge is a bidirectional edge.
- If there is undirected edge between vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ then edge $\{\boldsymbol{u}, \boldsymbol{v}\}$ is equal to edges $(\boldsymbol{u}, \boldsymbol{v})$ and $(\boldsymbol{v}, \boldsymbol{u})$

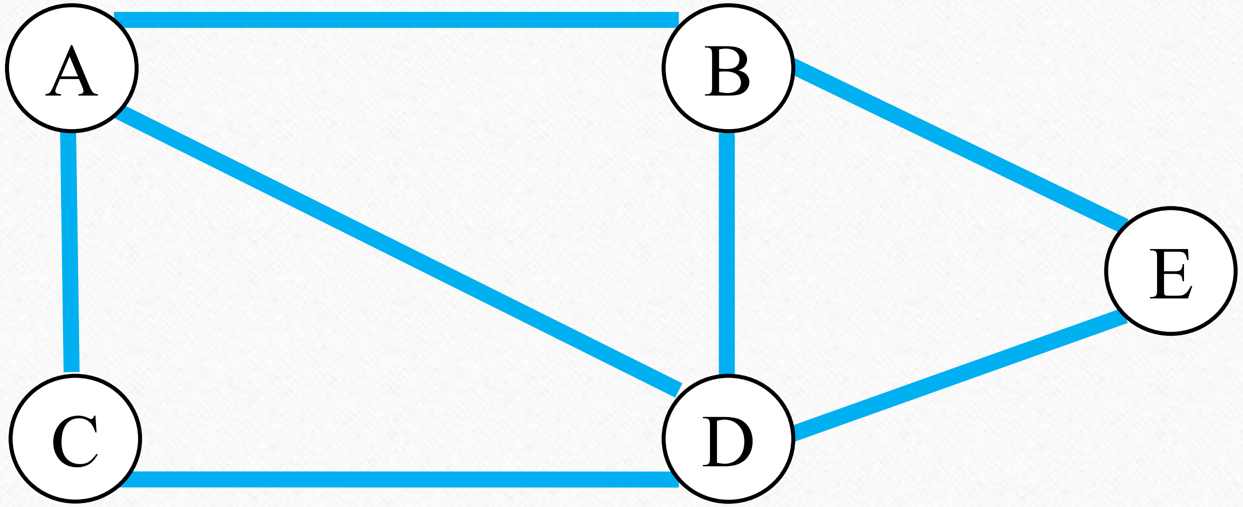


III. Graph Terminology

- **Undirected Edge**
  - unordered pair of vertices $\{\boldsymbol{u}\,,\boldsymbol{v}\}$
  - **Example:** one-way road traffic railway lines

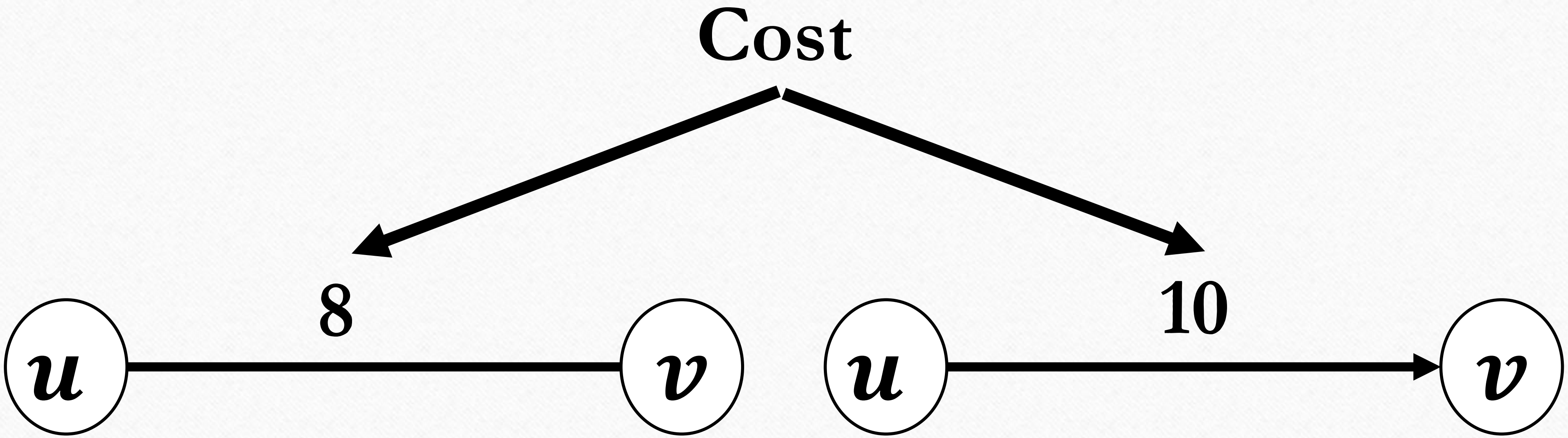
III. Graph Terminology

- **Undirected Edge :** An undirected edge is a bidirectional edge.



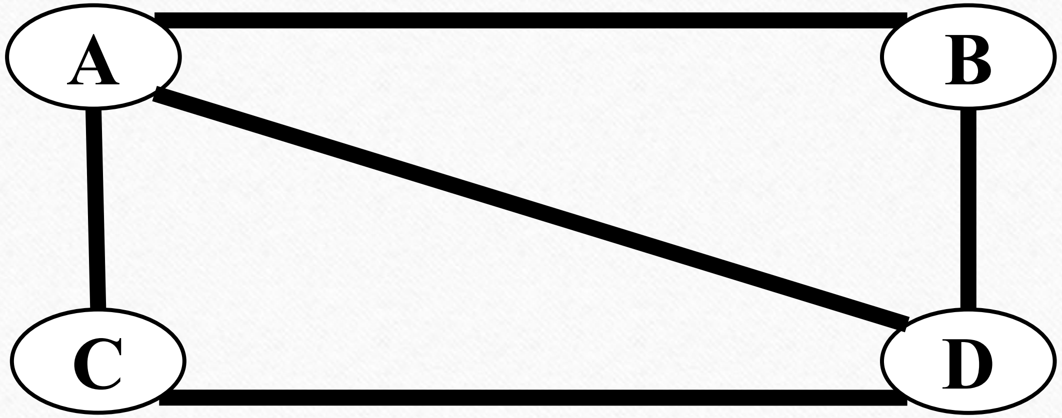
III. Graph Terminology

- **Weighted Edge :** A weighted edge is an edge with value (cost) on it.



III. Graph Terminology

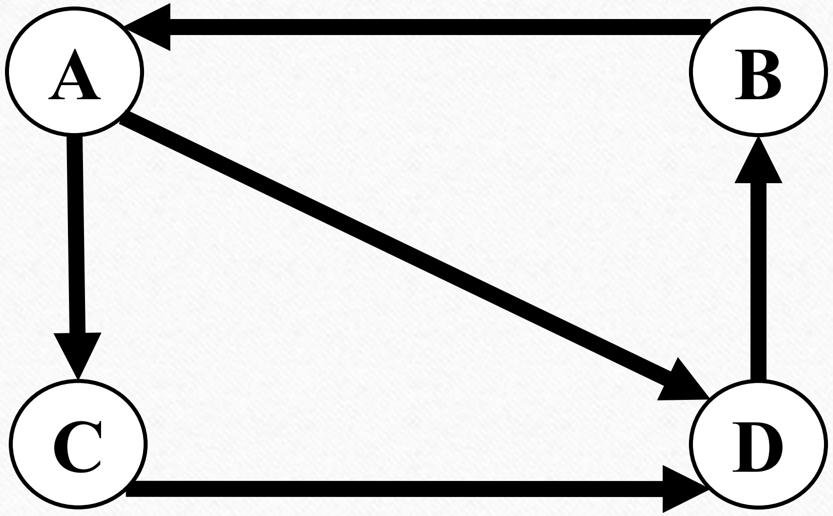
- **Undirected Graph :** A graph with **only undirected** edges is said to be undirected graph.



Undirected Graph

III. Graph Terminology

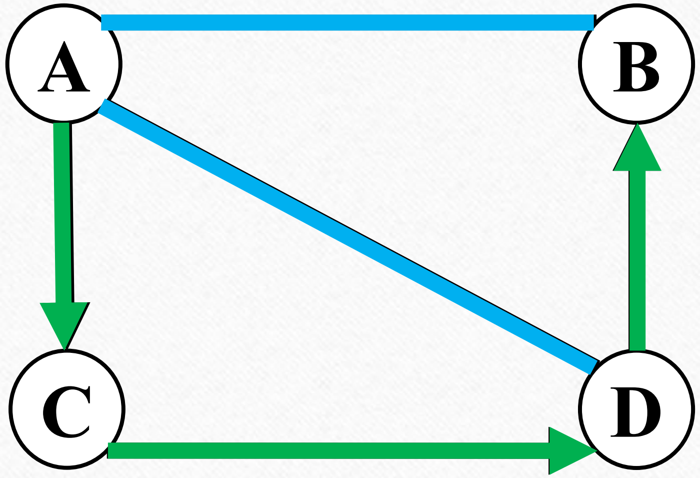
- **Directed Graph** : A graph with **only directed** edges is said to be directed graph.



Directed Graph

III. Graph Terminology

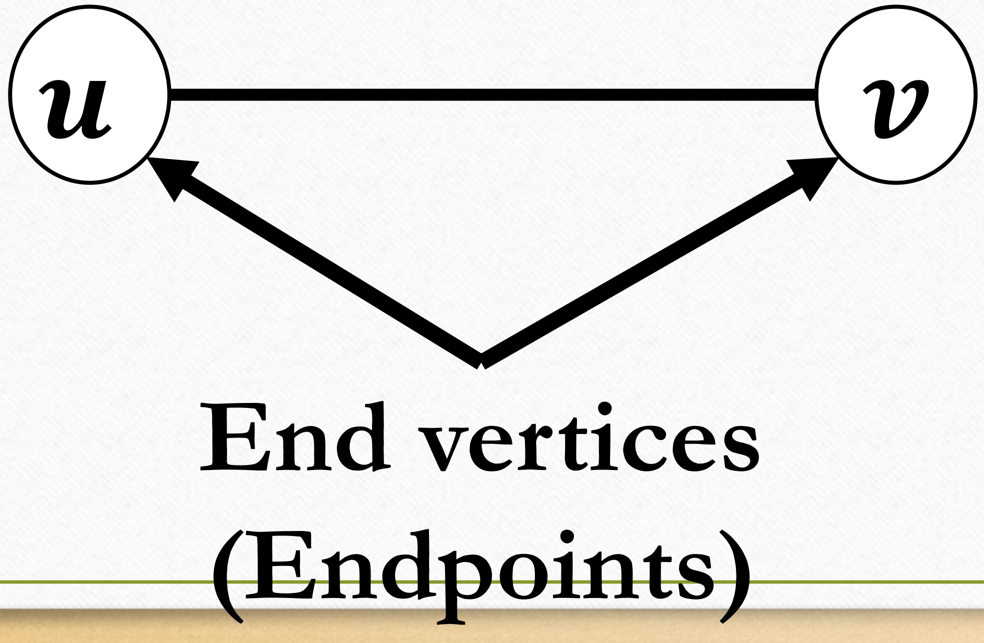
- **Mixed Graph** : A graph with **both undirected and directed** edges is said to be mixed graph

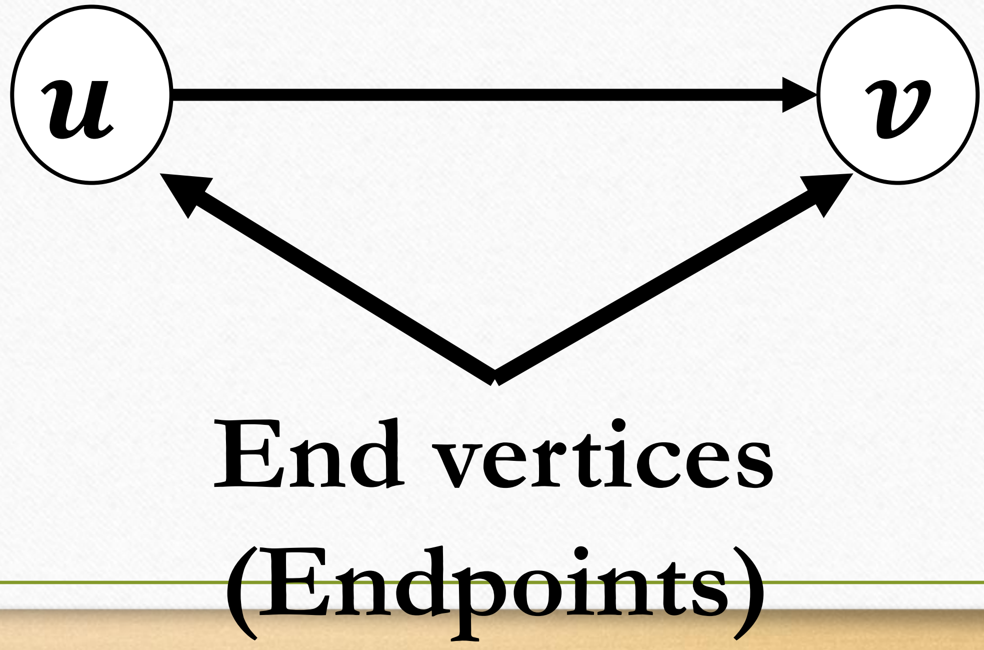


Mixed Graph

III. Graph Terminology

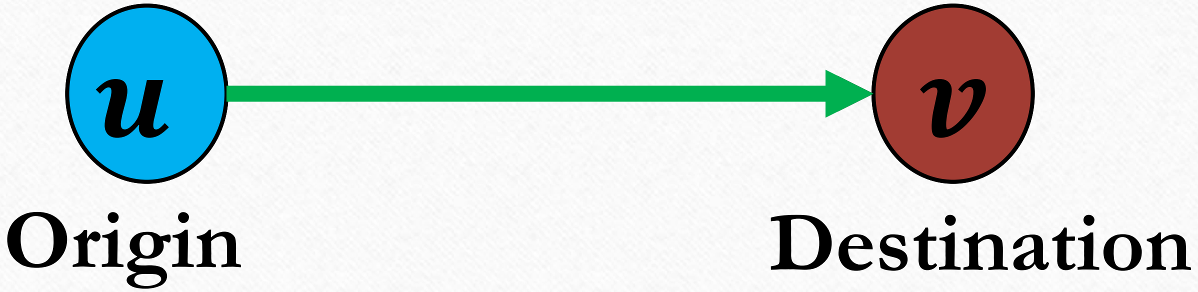
➢ **End vertices or Endpoints :** The two vertices joined by edge are called end vertices (or endpoints) of that edge.





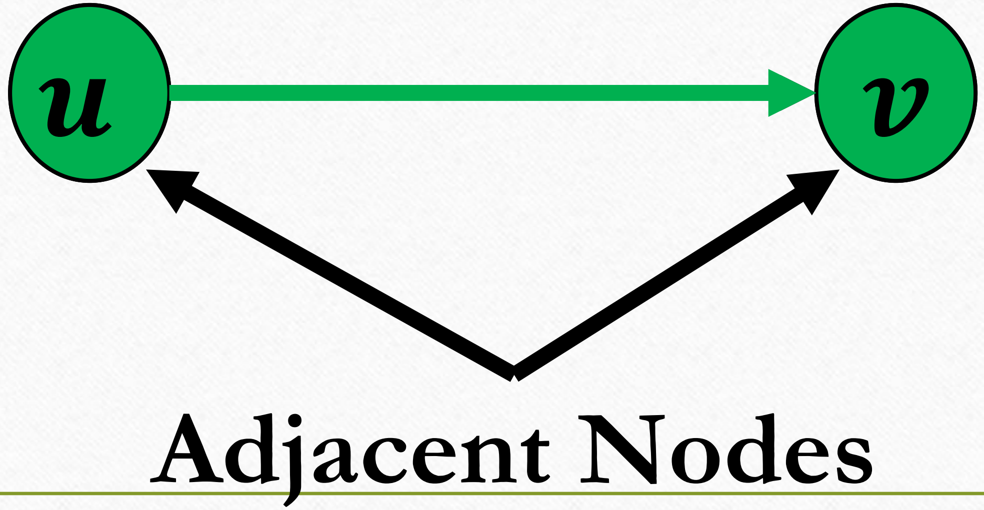
III. Graph Terminology

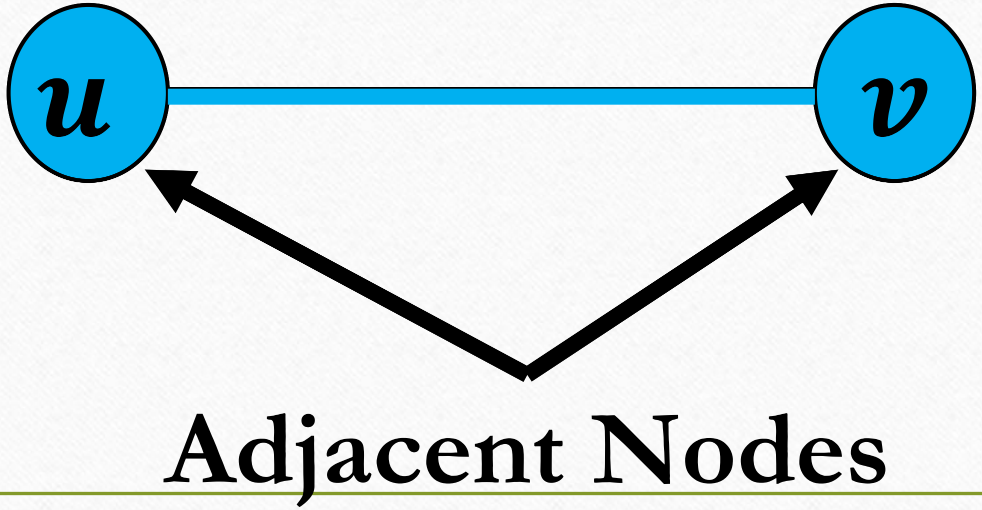
- **Origin :** If an edge is directed, its first endpoint is said to be the origin of it.
- **Destination :** If an edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.



III. Graph Terminology

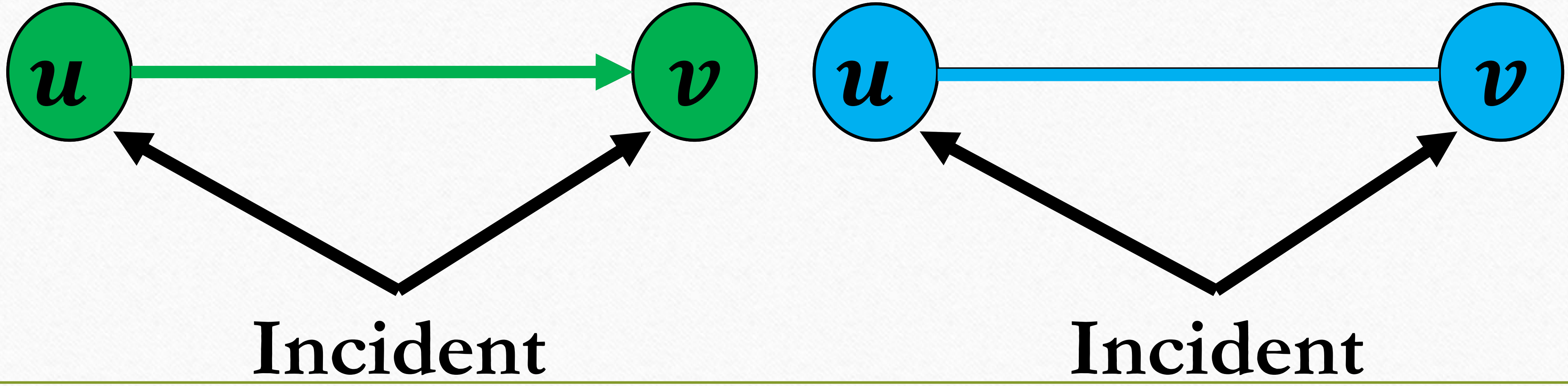
- **Adjacent Nodes :** If there is an edge between vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ then both $\boldsymbol{u}$ and $\boldsymbol{v}$ are said to be adjacent. In other words, vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ are said to be adjacent if there is an edge between them.





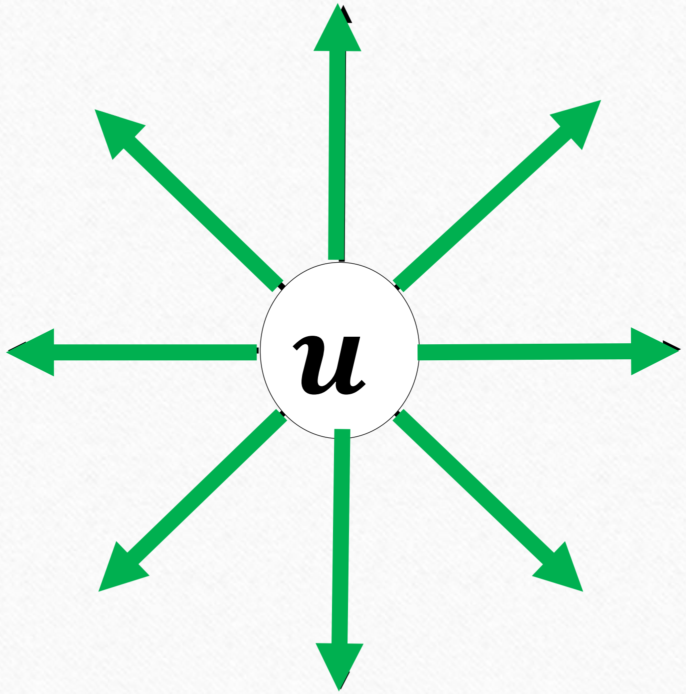
III. Graph Terminology

- **Incident :** Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.



III. Graph Terminology

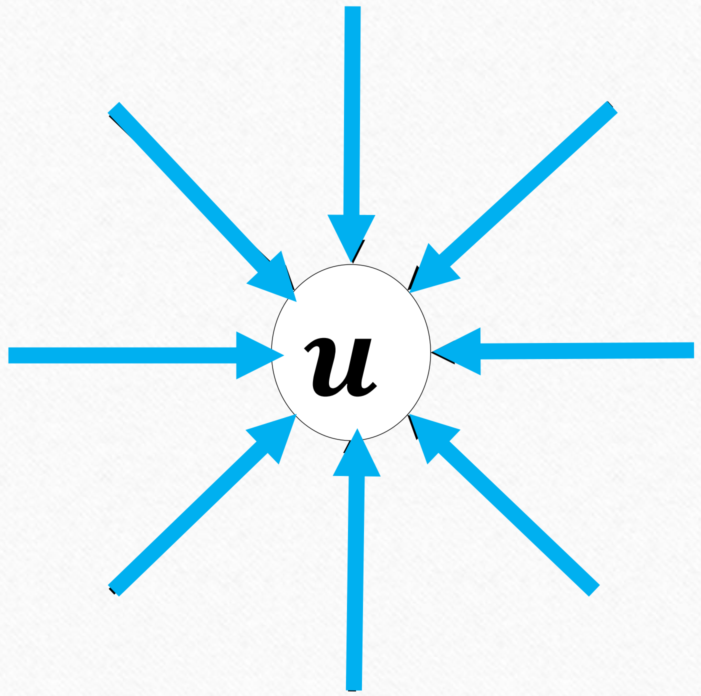
- **Outgoing Edge** : A **directed** edge is said to be **outgoing** edge on its origin vertex.



Outgoing Edge

III. Graph Terminology

- **Incoming Edge :** A directed edge is said to be incoming edge on its destination vertex.

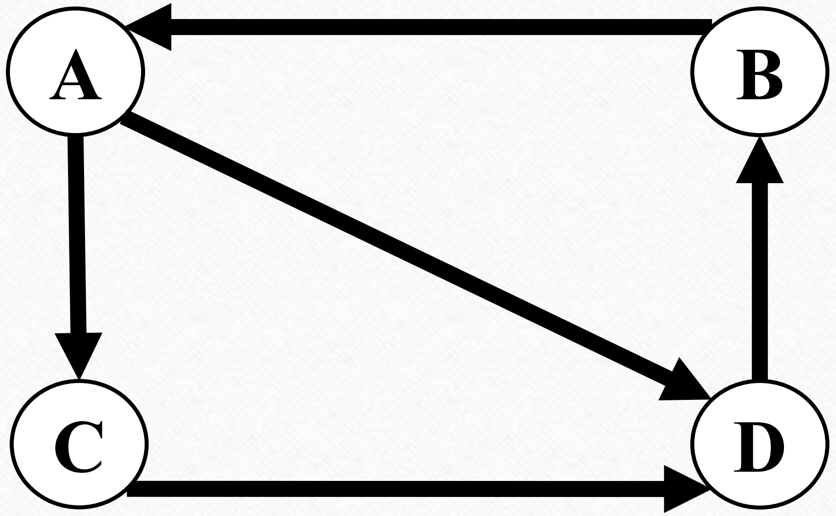


Incoming Edge

III. Graph Terminology

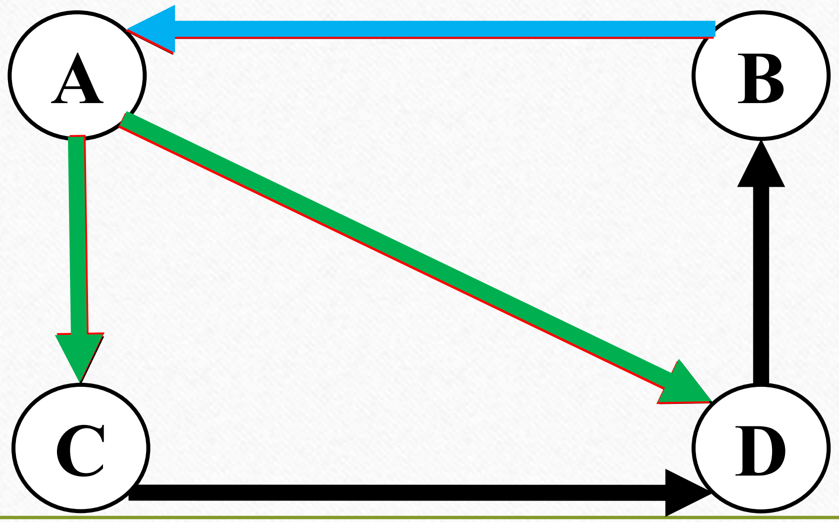
- **Degree** : Total number of edges connected to a vertex is said to be degree of that vertex.

- **Degree (A) = 3**
- **Degree (B) = 2**
- **Degree (C) = 2**
- **Degree (D) = 3**



III. Graph Terminology

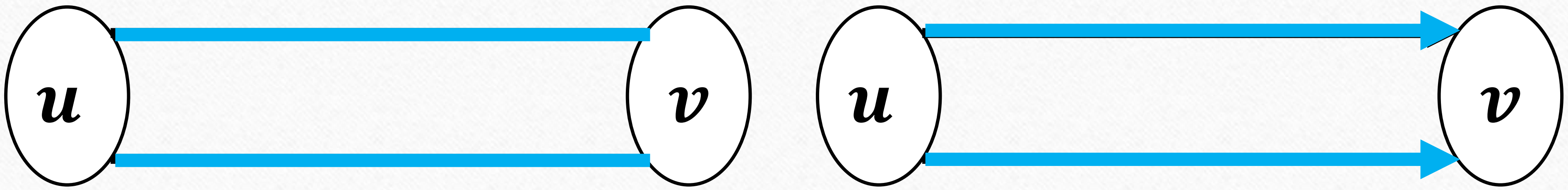
- **Indegree :** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
- **Outdegree :** Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
  - ❑ **Degree (A) = 3**
  - ❑ **Indegree (A) = 1**
  - ❑ **Outdegree (A) = 2**



Degree (X) = Indegree (X) + Outdegree (X)

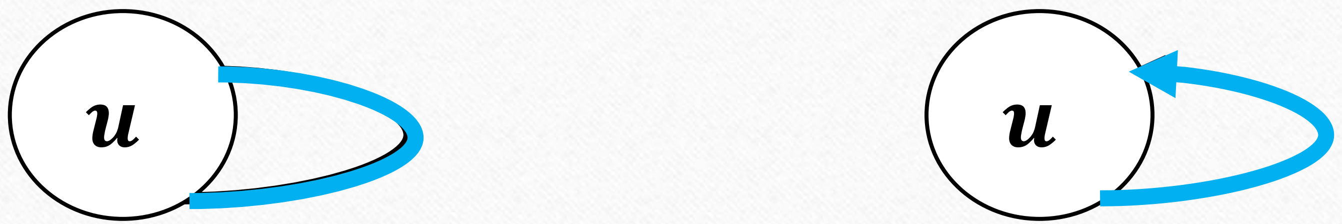
III. Graph Terminology

- **Parallel edges or Multiple edges :** If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges



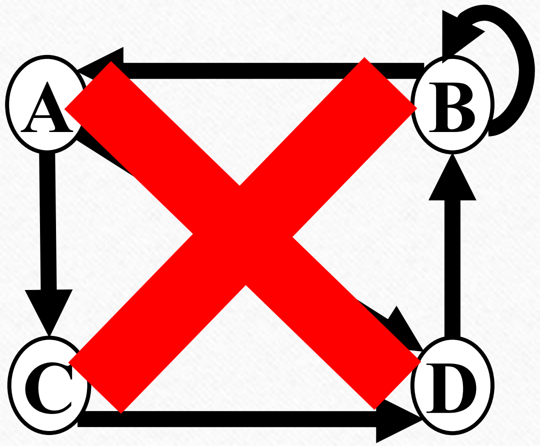
III. Graph Terminology

- **Self-loop :** Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

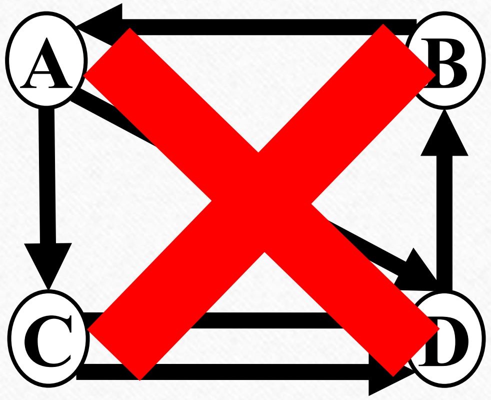


III. Graph Terminology

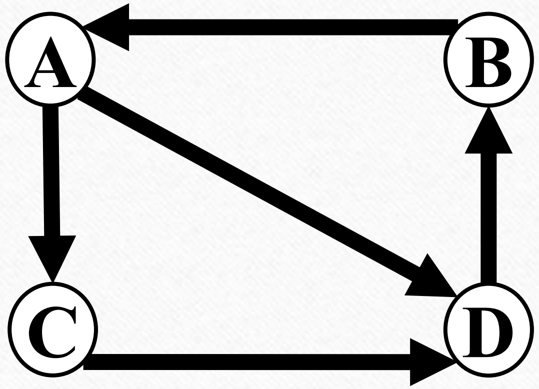
➢ **Simple Graph :** A graph is said to be simple if there are no parallel and self-loop edges.



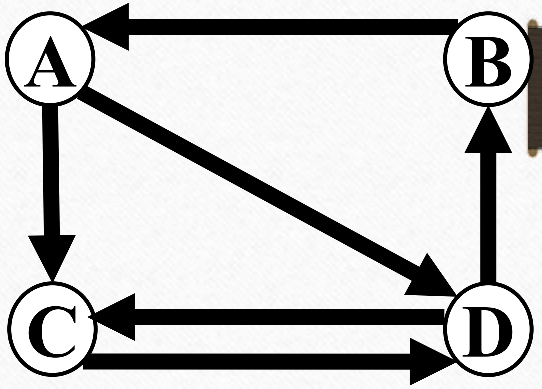
Not a simple graph



Not a simple graph



Simple graph



Simple graph

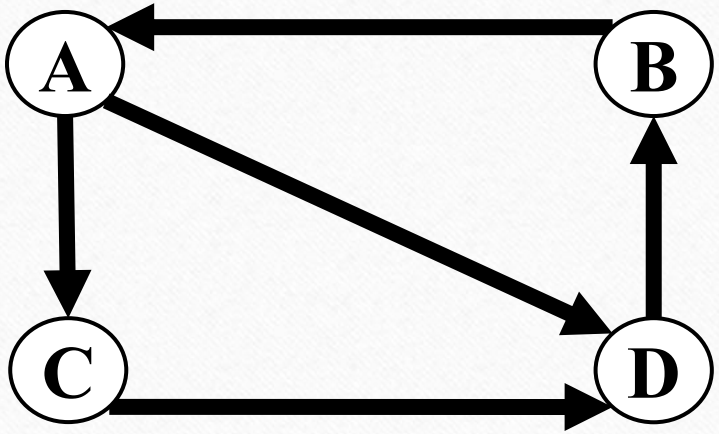
III. Graph Terminology

- **Path** : A path is a sequence of alternate vertices and edges that starts at a vertex and ends at another vertex such that each edge is incident to its predecessor and successor vertex.
- **Simple path is a path with no repeated vertices.**

Paths

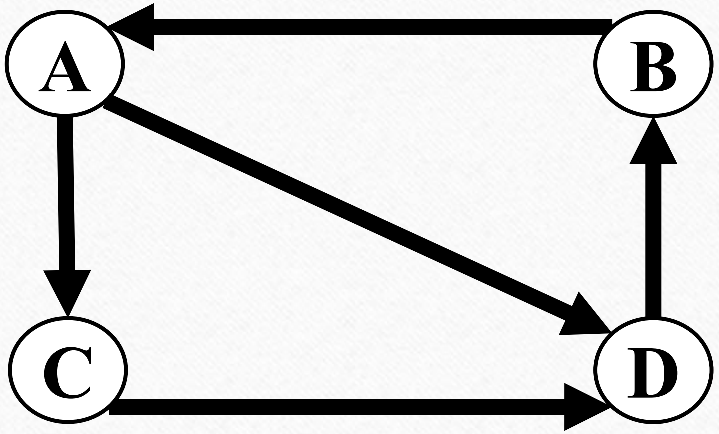
- D -> B -> A   Simple path
- A -> C -> D -> B   Simple path
- A -> C -> ~~D~~ -> B ->A ->D

  Not a Simple path



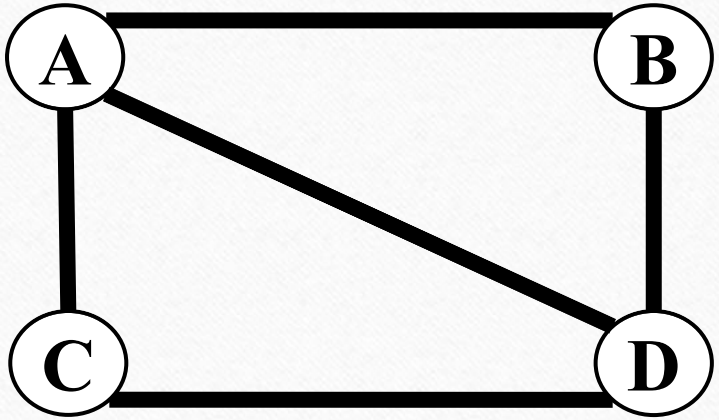
III. Graph Terminology

- **Cycle** : A Cycle in a graph is a **path that starts and ends at the same vertex**, with no repetitions of vertices (except the starting and ending vertex, which are the same).
- D -> B -> A Not a Cycle
- A -> C -> D -> B Not a Cycle
- A -> C -> D -> B ->A Cycle
- B -> A -> D -> B -> A -> C -> D -> B Not a Cycle

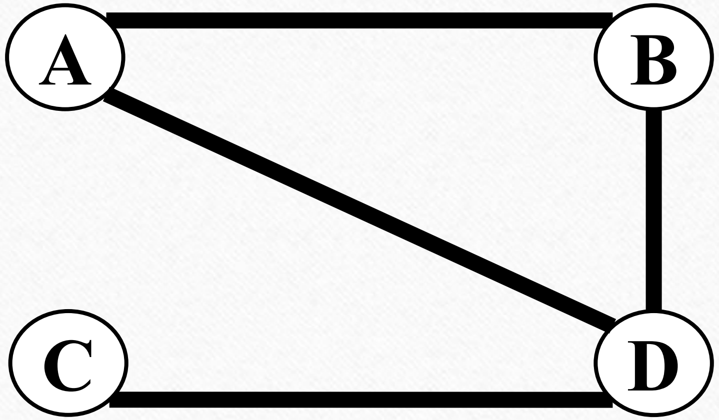


III. Graph Terminology

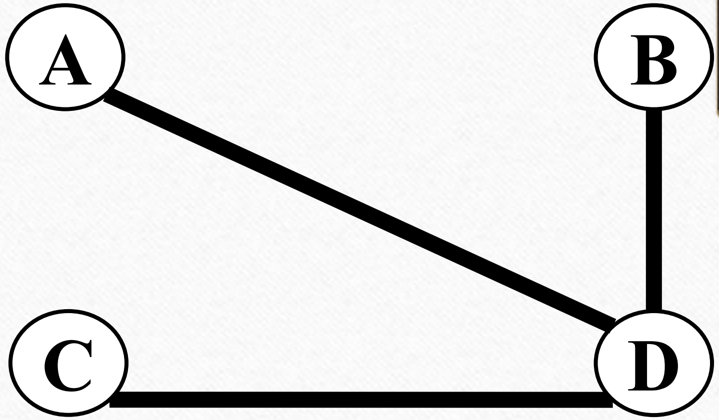
➢ **A graph with no cycles** is called a **tree**. A tree is an acyclic connected graph.



A graph with cycles



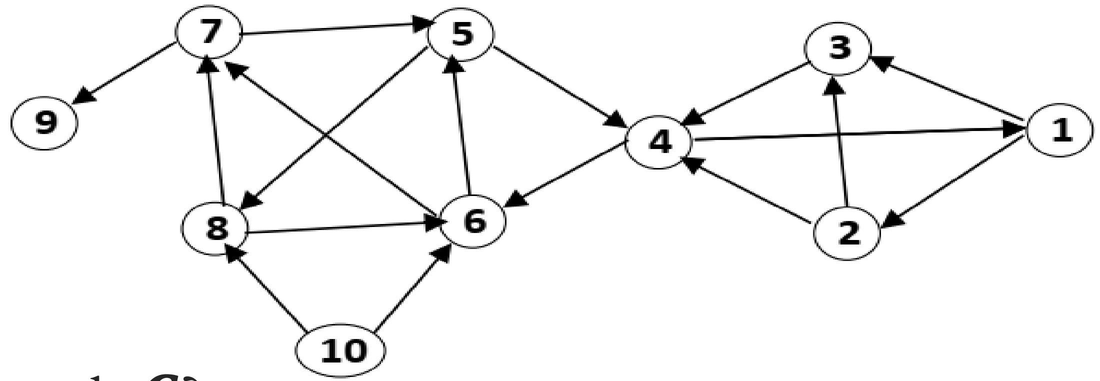
A graph with cycles



A graph with no cycles (Tree)

III. Graph Terminology

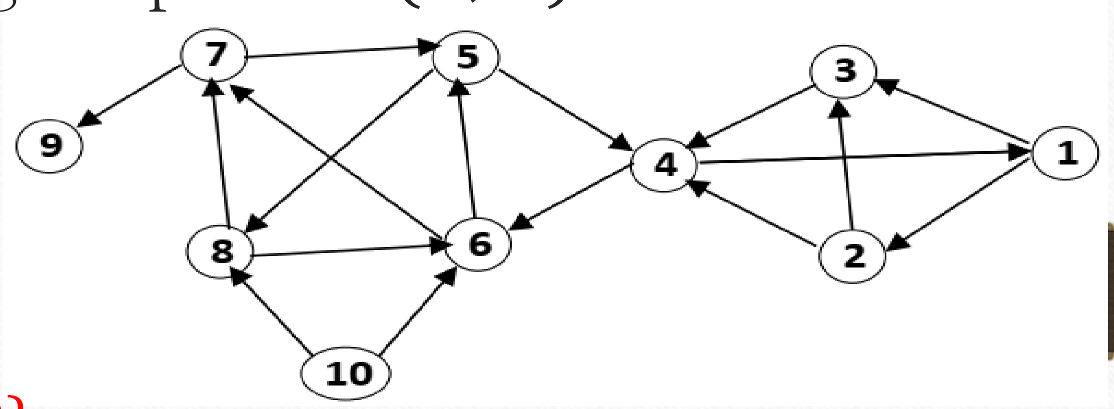
Example : Consider the following Graph $\boldsymbol{G = (V, E)}$.



1) What is the type of graph $\boldsymbol{G}$?
2) What are the **vertices** $\boldsymbol{V}$ of $\boldsymbol{G}$?
3) What are the **edges** $\boldsymbol{E}$ of $\boldsymbol{G}$?
4) What is the **order** and **size** of the graph $\boldsymbol{G}$?
5) Find the **number of outgoing** and **incoming edges** of vertex **6**.
6) Find the **degree , indegree , and outdegree** of vertex **4**.
7) Find a path between vertices **4** and **6**, Is it simple?
8) Does the graph have a cycle?

III. Graph Terminology

Example : Consider the following Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$.



1) What is the type of graph $\boldsymbol{G}$?

   Directed Graph

2) What are the **vertices** $\boldsymbol{V}$ of $\boldsymbol{G}$?

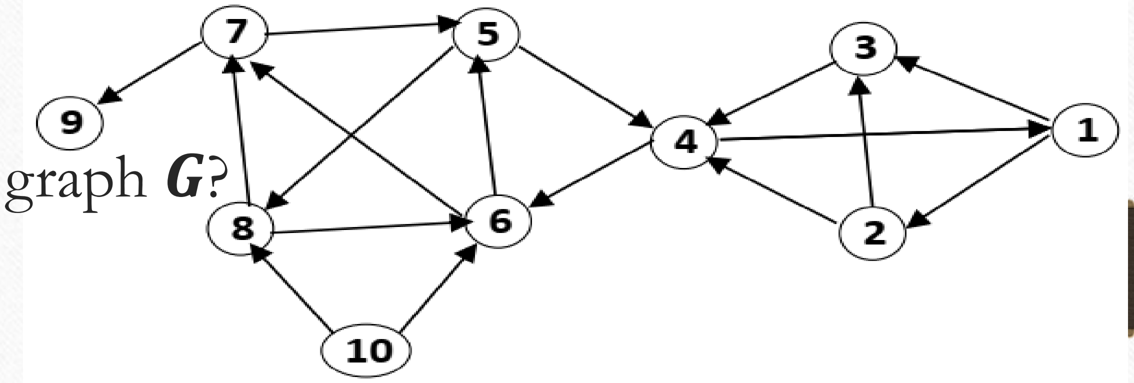
   $$\boldsymbol{V} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

3) What are the **edges** $\boldsymbol{E}$ of $\boldsymbol{G}$?

   $$\boldsymbol{E} = \{(1,2), (1,3), (2,3), (2,4), (3,4), (4,1), (4,6), (5,4), (5,8), (6,5), (6,7), (7,5), (7,9), (8,6), (8,7), (10,6), (10,8)\}$$

III. Graph Terminology

Example : Consider the following Graph $\boldsymbol{G = (V, E)}$.



4) What is the **order** and **size** of the graph $\boldsymbol{G}$?

Order of Graph G : n=|V|=10

Size of Graph G : e=|E|=17

5) Find the **number of outgoing** and **incoming edges** of vertex **6**.

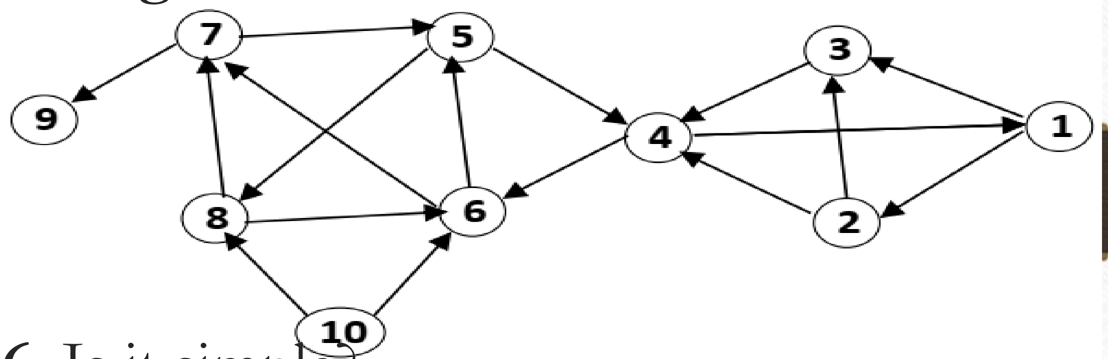
The number of **outgoing** edges from vertex 6 is 2

The number of **incoming** edges from vertex 6 is 3

III. Graph Terminology

Example : Consider the following Graph $\boldsymbol{G = (V, E)}$.

6) Find the **degree , indegree , and outdegree** of vertex **4**.



The degree of vertex 4 is 5

The indegree of vertex 4 is 3

The outdegree of vertex 4 is 2

7) Find a path between vertices **4** and **6**, Is it simple?

C1 = (4, 6) is a simple path

C2 = (4, 6, 7, 5, 8, 6) is not a simple path

8) Does the graph have a cycle?

Yes, the graph have a cycle ➔ C3 = (6, 7, 5, 8, 6)

IV. Graph Representations

- The graphs are **non-linear, non-regular** structures.
- Two important graph abstract data types (ADTs) that can be used to represent a graph in memory are the **adjacency matrix** and **adjacency list**.
- The benefits of choosing one implementation over the other depend on the type of graph problem you are working on.

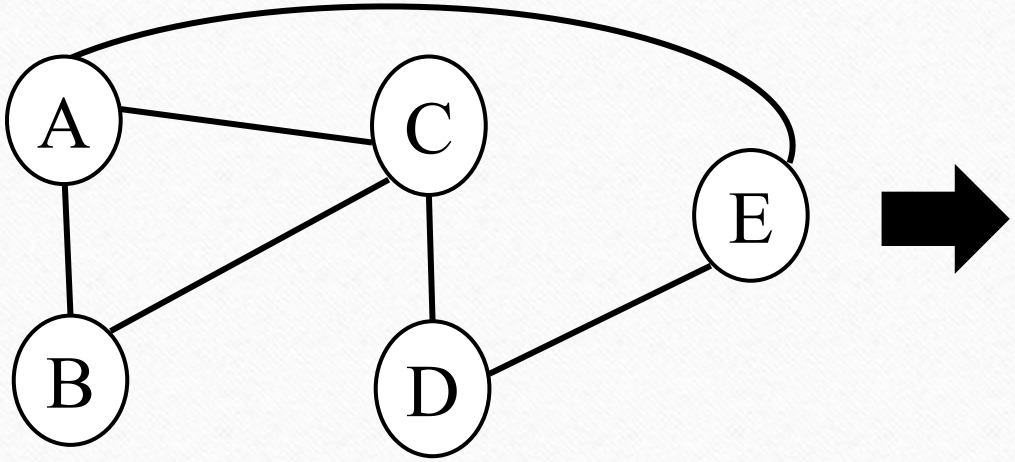
IV. Graph Representations

Adjacency Matrix

- An adjacency matrix is a matrix $\boldsymbol{M}$ where the cell $\boldsymbol{M}[\boldsymbol{i}][\boldsymbol{j}]$ represents whether an edge exists from vertex $\boldsymbol{i}$ to vertex $\boldsymbol{j}$.
- In an unweighted graph, the element at row $\boldsymbol{i}$, column $\boldsymbol{j}$ of the adjacency matrix is $\mathbf{1}$ if and only if the edge $(\boldsymbol{v_i}, \boldsymbol{v_j})$ exists in the set of edges $\boldsymbol{E}$, and $\mathbf{0}$ otherwise.
- For **undirected graphs**, an adjacency matrix is always symmetric across the diagonal (since a connection between $\boldsymbol{i}$ and $\boldsymbol{j}$ also implies a connection between $\boldsymbol{j}$ and $\boldsymbol{i}$).

IV. Graph Representations (Adjacency Matrix)

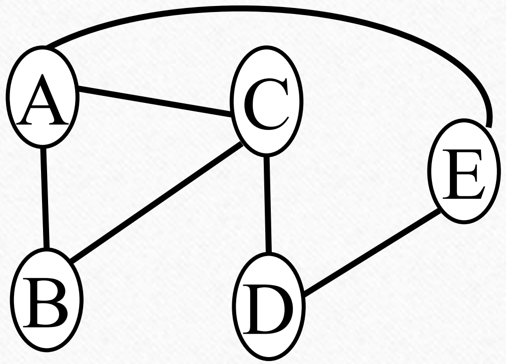
- An example of an adjacency matrix for an undirected graph is shown below :



	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

IV. Graph Representations (Adjacency Matrix)

- Because the adjacency matrix of an undirected graph is always symmetric, we can save time and space by only filling out half of the matrix. If we do this, we must make sure to access the rows and columns in the correct order (such as making sure that the row index is never larger than the column index).

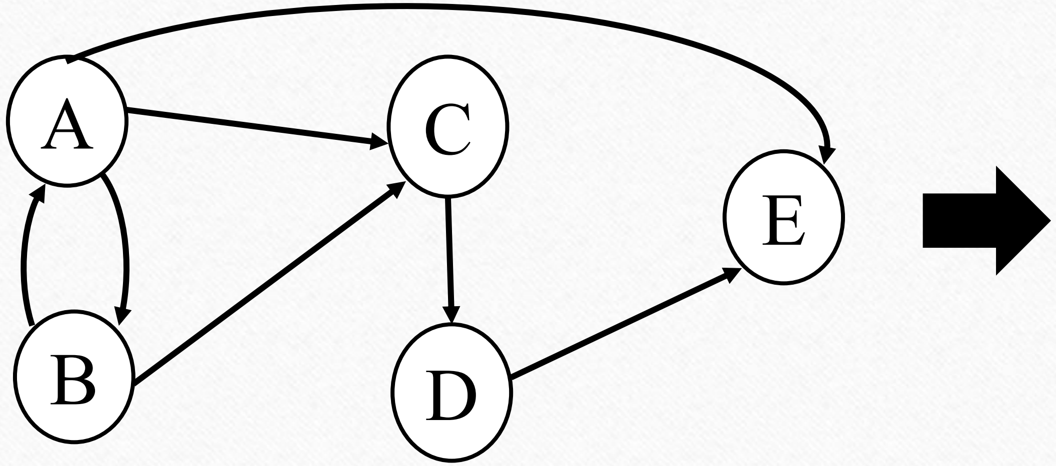


	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

	A	B	C	D	E
A	0	1	1	0	1
B	-	0	1	0	0
C	-	-	0	1	0
D	-	-	-	0	1
E	-	-	-	-	0

IV. Graph Representations (Adjacency Matrix)

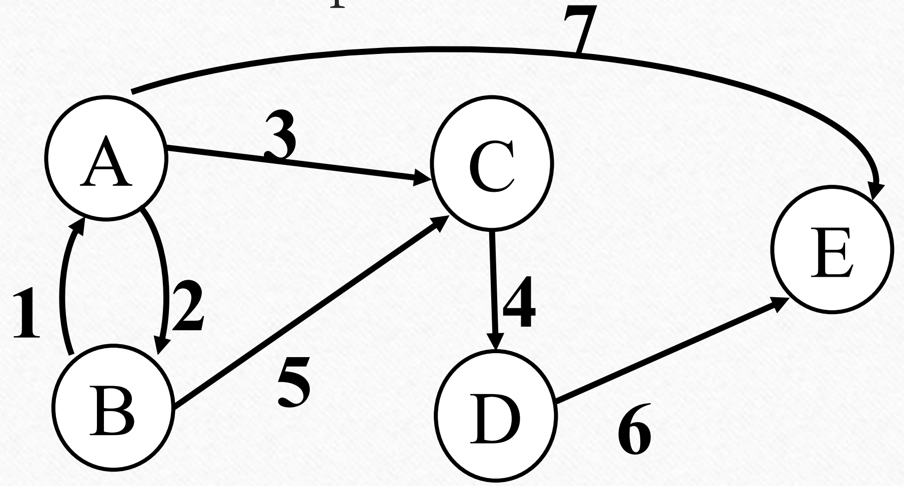
- However, in a **directed graph**, we will have to fill out the entire matrix. This is because the existence of a path from vertex ***A*** to ***B*** does not mean that there is one the other way around.
- An example of an adjacency matrix for a directed graph is shown below:



	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

IV. Graph Representations (Adjacency Matrix)

- An adjacency matrix can also be used to represent weighted graphs. If a graph is weighted, the cell $M[i][j]$ would store the weight of the edge from vertex i to vertex j. If no edge exists between vertex i and vertex j, then the cell $M[i][j]$ is assigned an edge weight of infinity.
- An example is shown below:



	A	B	C	D	E
A	∞	2	3	∞	7
B	1	∞	5	∞	∞
C	∞	∞	∞	4	∞
D	∞	∞	∞	∞	6
E	∞	∞	∞	∞	∞

Remark : To initialize a value to ∞, you can use `std::numeric_limits<double>::infinity()` from the `<limits>` library.

IV. Graph Representations (Adjacency Matrix)

Declarations

```
// N vertices and M Edges
        int N, M;
// Given Edges
        int arr_edges[][2];
// For Adjacency Matrix
        int Adj[N + 1][N + 1];
```

IV. Graph Representations (Adjacency Matrix)

Complexity

- Given a graph with $|\boldsymbol{V}|$ vertices, the time complexity to build an adjacency matrix is $\mathcal{O}(|\boldsymbol{V}^2|)$, since you have to construct a $|\boldsymbol{V}| \times |\boldsymbol{V}|$ matrix.
- The adjacency matrix representation requires a lot of space: for a graph with $\boldsymbol{V}$ vertices we must allocate space in $\mathcal{O}(\boldsymbol{V}^2)$.
- The time required to iterate over **all adjacent** vertices of a vertex in an adjacency matrix is $\mathcal{O}(|\boldsymbol{V}|)$, and the time required to iterate over **all edges** is $\mathcal{O}(|\boldsymbol{V}^2|)$.
- However, the benefit of the adjacency matrix representation is that **adding** an edge and **checking** for the existence of an edge are both $\mathcal{O}(\mathbf{1})$ operations.

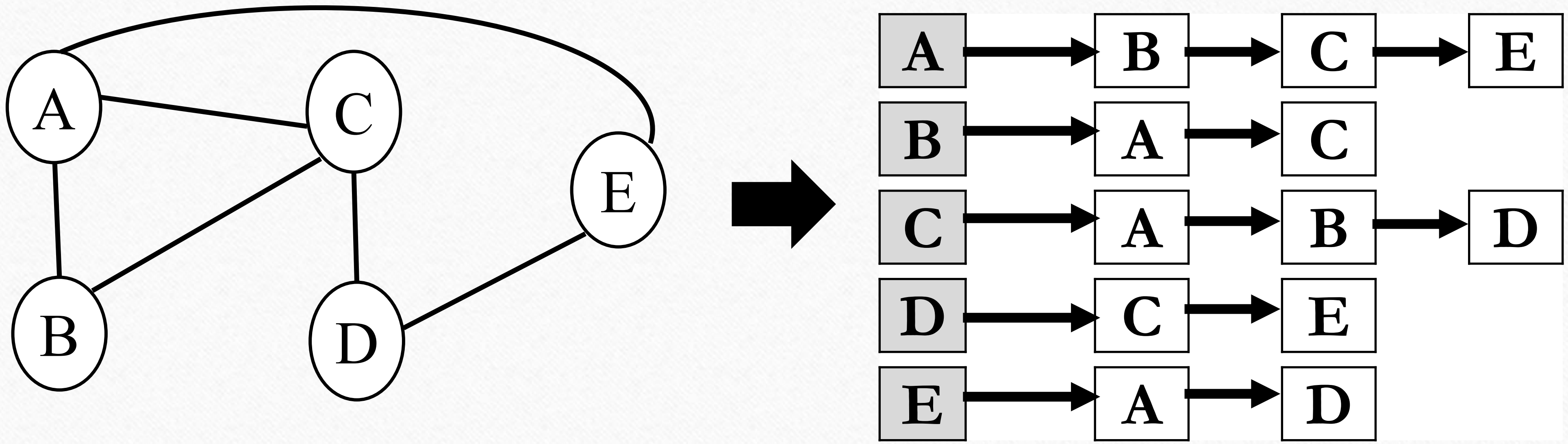
IV. Graph Representations

Adjacency List

- Instead, if a graph is sparse, a better alternative is to use an **adjacency list** as the underlying representation.
- In an adjacency list, each vertex $\boldsymbol{v}$ is mapped to a list of edges that originate from $\boldsymbol{v}$.

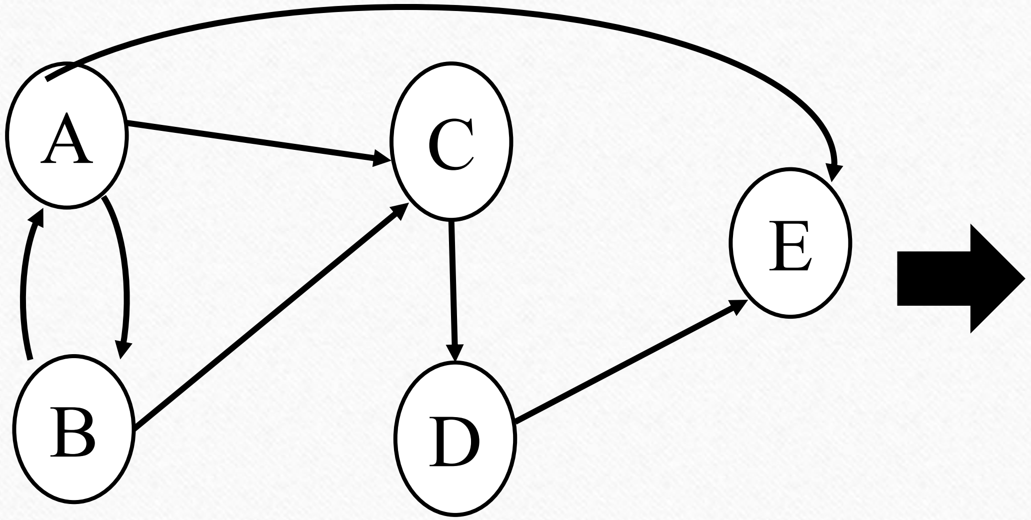
IV. Graph Representations (Adjacency List)

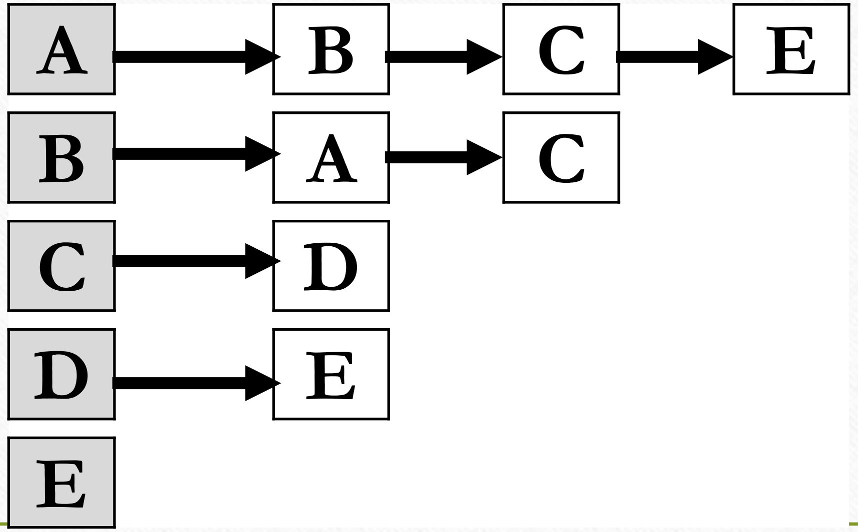
- For example, the following depicts an adjacency list for an undirected graph, where the presence of $\boldsymbol{A} \rightarrow [\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{E}]$ indicates that vertex A has a connection to vertices $\boldsymbol{B}, \boldsymbol{C}$, and $\boldsymbol{E}$.



IV. Graph Representations (Adjacency List)

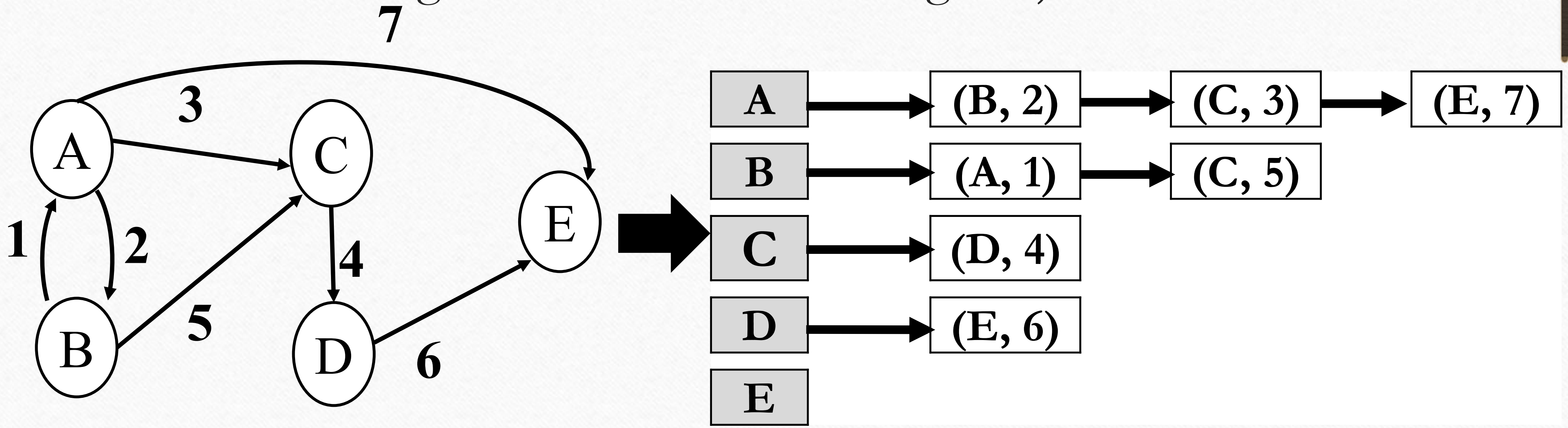
- Adjacency lists also work with **directed graphs** and **weighted graphs**.
- The following depicts an adjacency list for a directed graph (which is similar to the undirected representation, with the exception that $\boldsymbol{X} \rightarrow \boldsymbol{Y}$ does not necessarily guarantee $\boldsymbol{Y} \rightarrow \boldsymbol{X}$).





IV. Graph Representations (Adjacency List)

- The following depicts an adjacency list for a weighted graph. When dealing with weighted graphs, each edge in the adjacency list is paired with its weight (e.g., $\boldsymbol{A} \rightarrow (\boldsymbol{B}, \mathbf{2})$ indicates that there is an edge from $\boldsymbol{A}$ to $\boldsymbol{B}$ with weight $\mathbf{2}$).

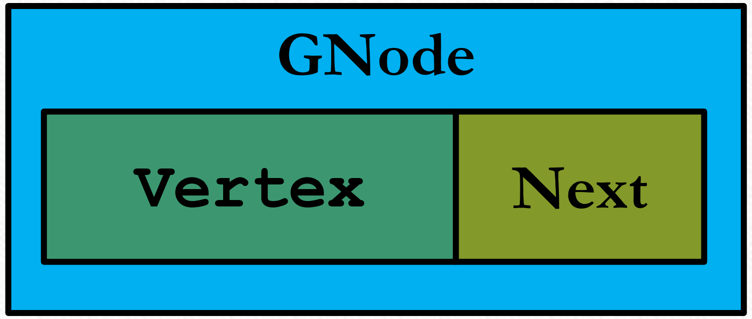


IV. Graph Representations (Adjacency List)

Declarations

```
struct GNode{
        int Vertex;
        GNode* Next;
};
```

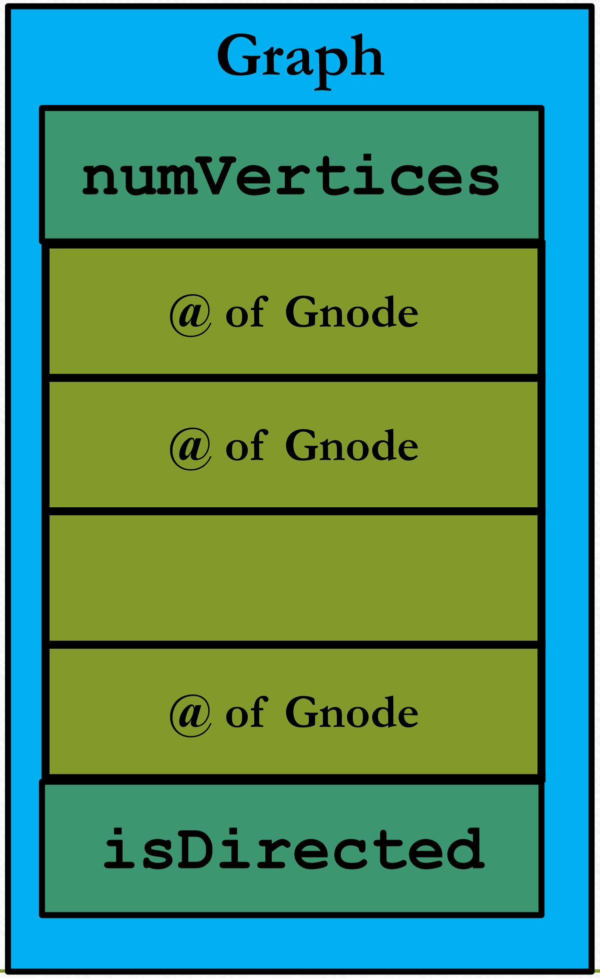
Graph Node



IV. Graph Representations (Adjacency List)

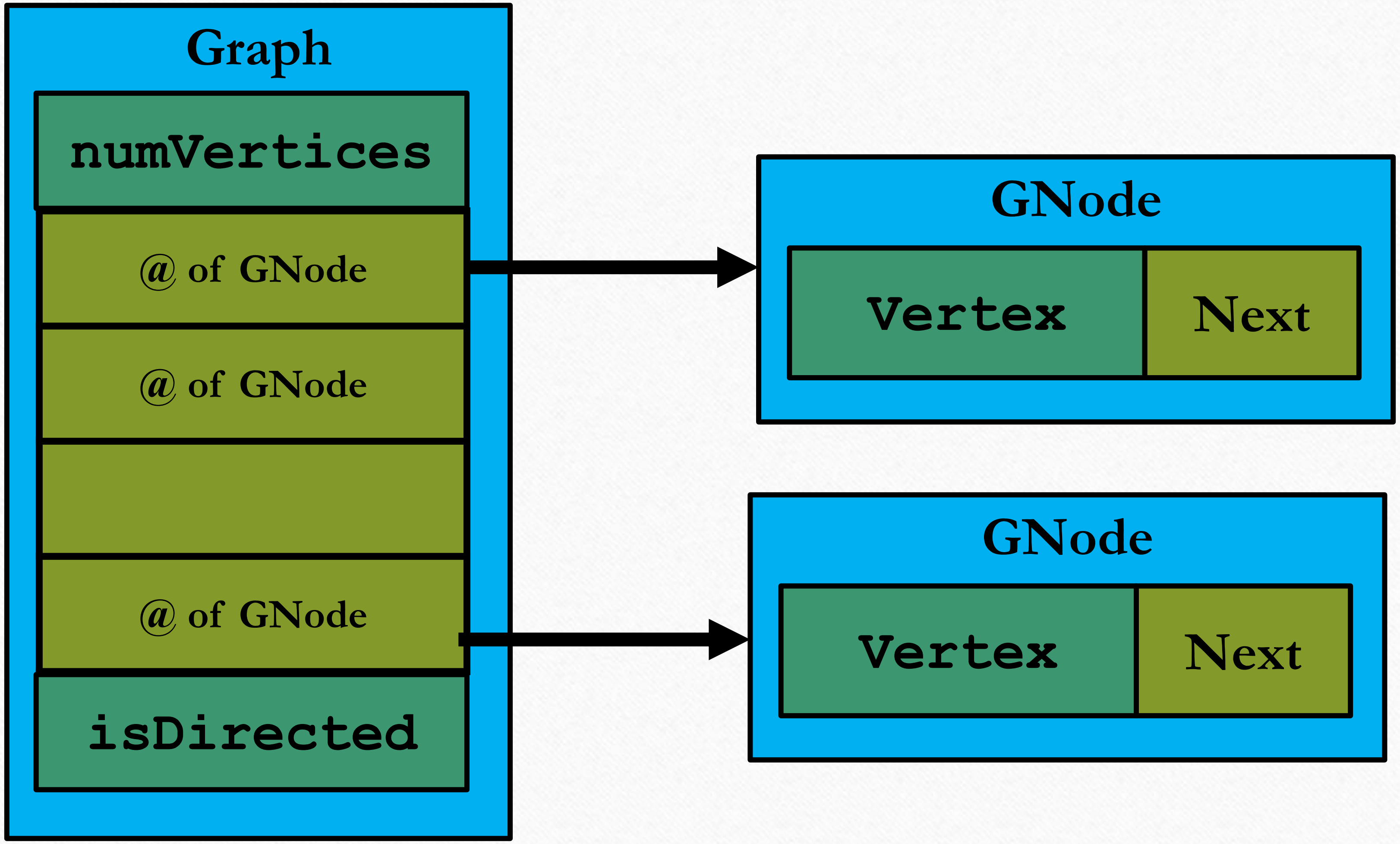
Declarations

```
struct Graph {
    unsigned int numVertices;
    GNode** adjLists;
    bool isDirected;
};
```



IV. Graph Representations (Adjacency List)

Declarations



IV. Graph Representations (Adjacency List)

Declarations

```
struct GNode{
      int Vertex;
      GNode* Next;
};
struct Graph {
   unsigned int numVertices;
   GNode** adjLists;
   bool isDirected;
};
```

IV. Graph Representations (Adjacency List)

Complexity

- Given a graph with $|V|$ vertices, the time complexity to build an adjacency list is $\mathcal{O}(|E|)$, and the space complexity is $\mathcal{O}(|V| + |\mathbf{E}|)$.
- The time complexity is $\mathcal{O}(|E|)$ because you have to iterate through each edge and insert it into the adjacency list, where each insertion takes $\mathcal{O}(\mathbf{1})$ time (assuming the adjacency list is stored in a **2-D** vector or a hash table).
- The space complexity is $\mathcal{O}(|V| + |\mathbf{E}|)$ because storing all the vertices requires $\mathcal{O}(|V|)$ space, and storing all the edges in the lists requires $\mathcal{O}(|\mathbf{E}|)$ space.

V. Graph Traversal

- Perhaps the most fundamental problem with graphs is to **systematically visit every edge and vertex of a graph**.
- Indeed, all basic graph accounting operations (such as printing or copying graphs, and converting between alternative representations) are applications of graph traversal.
- To solve these problems on graphs, we need a mechanism for traversing the graphs.

V. Graph Traversal

- Graph traversal algorithms are also called **graph search algorithms**. Like trees traversal algorithms (In-order, Pre-order, Post-order, and Level-Order traversals), graph search algorithms can be thought of as starting at some source vertex in a graph and "searching" the graph by going through the edges and marking the vertices.
- Now, we will discuss two such algorithms for traversing the graphs.
  - **Depth First Search (DFS)**
  - **Breadth First Search (BFS)**

V. Graph Traversal

Depth First Search (DFS)

- The depth-first search (DFS) algorithm works **similarly** to **preordered** tree traversal, but this algorithm also uses the stack.
- The DFS algorithm is a core graph traversal algorithm that can be used to systematically explore the vertices and edges of a graph that are **reachable** from a given source vertex.
- In a DFS, the algorithm **starts at a source vertex** and **explores each branch of the graph as far as possible until** there are no more undiscovered vertices along its path, at this point, it **backtracks** and continues along another branch.

V. Graph Traversal (Depth First Search (DFS))

Implementing a DFS generally relies on recursion or the **stack**<> data structure. An outline of the DFS algorithm is as follows:

1) Mark the **source vertex as visited**, and then **push its neighbors** (i.e., adjacent vertices) **into a stack**. This can be done by pushing all neighbors into an actual **stack<>**, or by performing a recursive call on every neighbor (which pushes neighbors onto the program call stack).

2) **Pop the vertex at the top** of the **stack** and set it as the **current vertex** (this behavior is automatically done by a recursive call if a recursive approach is used).

V. Graph Traversal (Depth First Search (DFS))

Implementing a DFS generally relies on recursion or the **stack<>** data structure. An outline of the DFS algorithm is as follows:

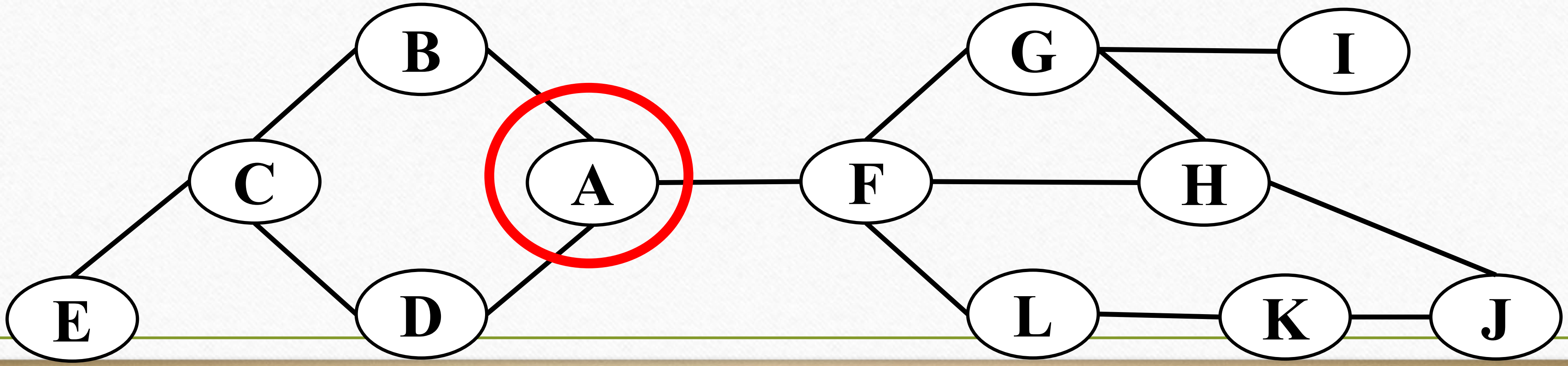
3) **Push the unvisited neighbors** of the current vertex **into** the **stack** (or perform a recursive call on each neighbor, if a recursive approach is used) and **mark them as visited**.

4) Repeat **steps 2** and **3** until the **stack** is empty.

V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

- A depth-first search can be done both iteratively (using a **stack<>**) or recursively (using the program stack).
- For example, consider the following graph, which we will traverse using an **iterative** depth-first search starting at vertex **A**.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

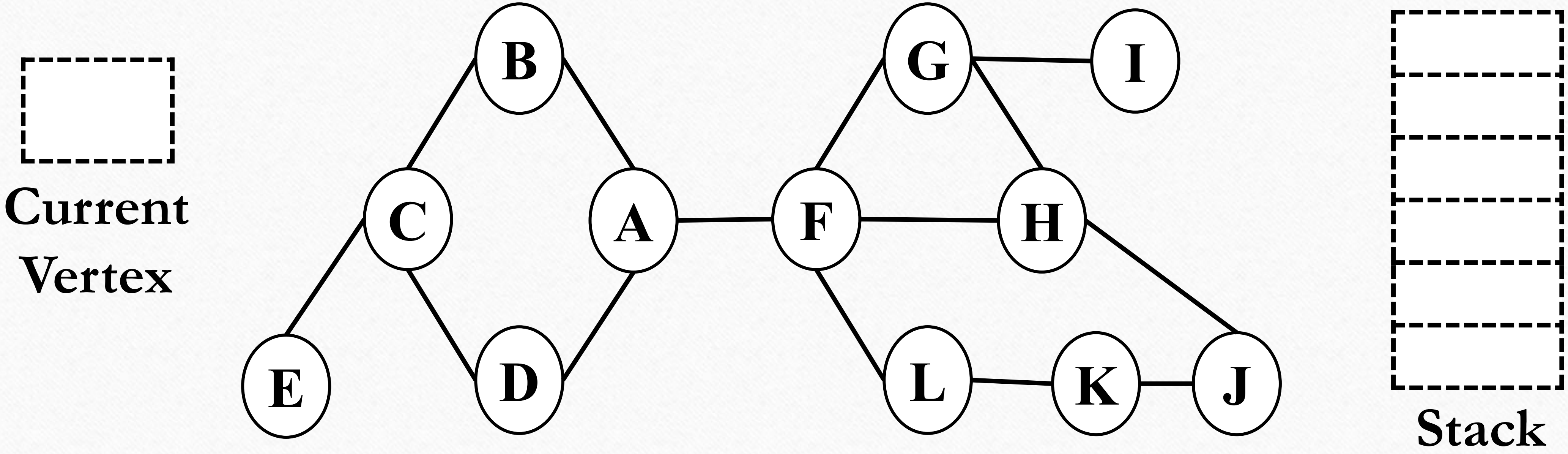
Here, we will classify visited nodes into two groups: **discovered** and **processed**.

- A **discovered** vertex is a vertex that has been encountered and pushed into the **stack**; discovered vertices will be represented with a **light gray** shading.
- A **processed** vertex is a vertex that has already been taken out of the **stack** and had its adjacency list fully examined; processed vertices will be represented with a **dark gray** shading.

V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

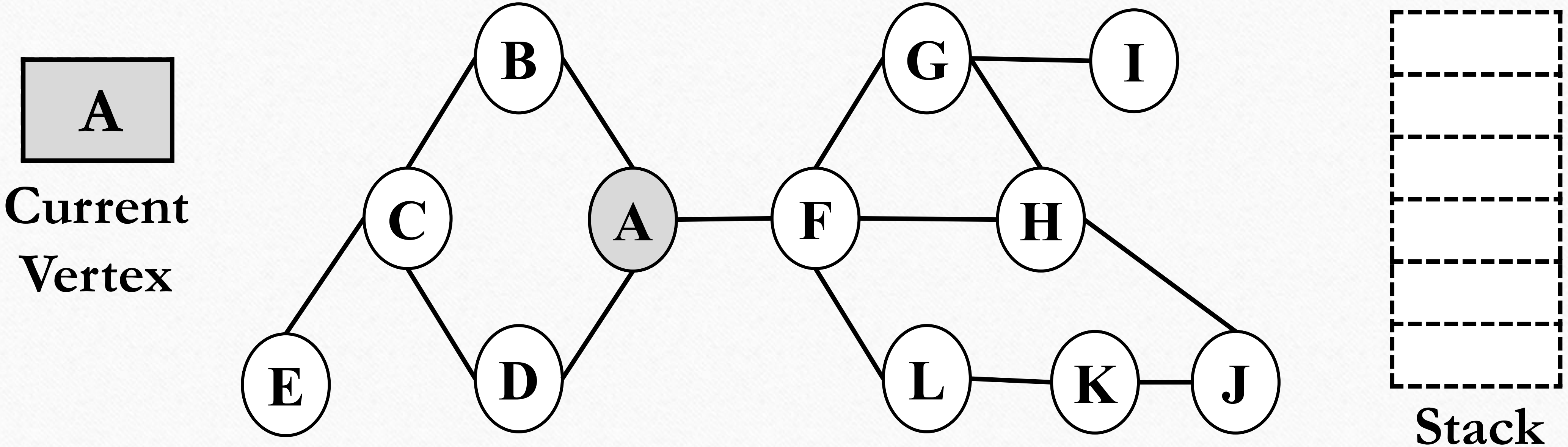
We start by initializing a **stack<>** to track vertices yet to be explored.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

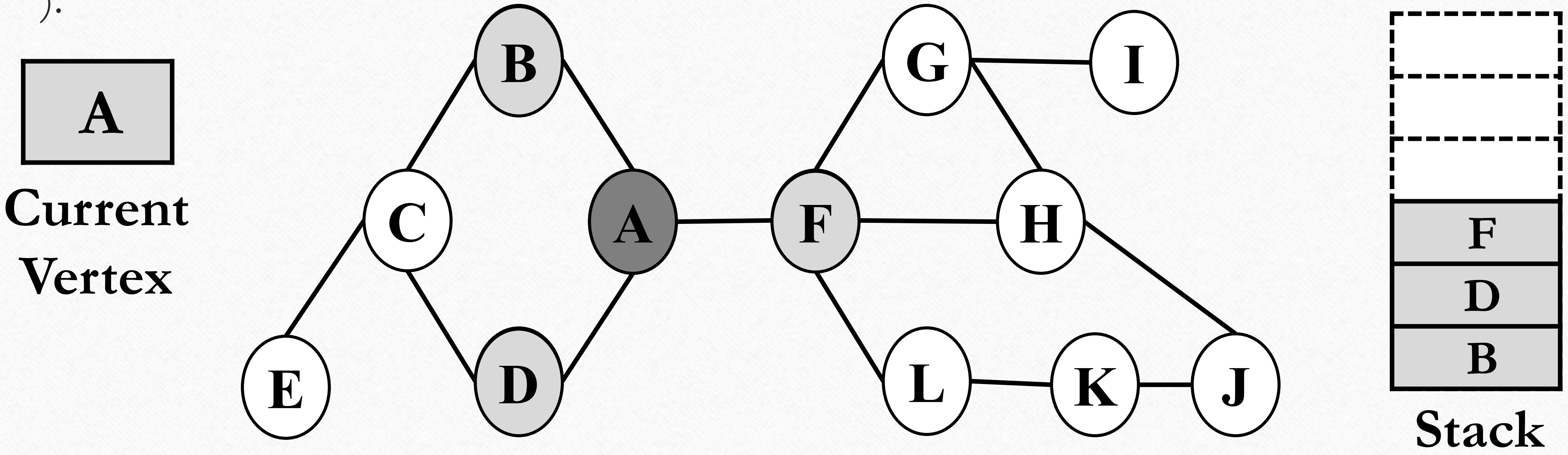
First, we will mark the starting vertex as visited. In this case, our starting vertex is vertex **A**, so we will mark **A** as visited.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

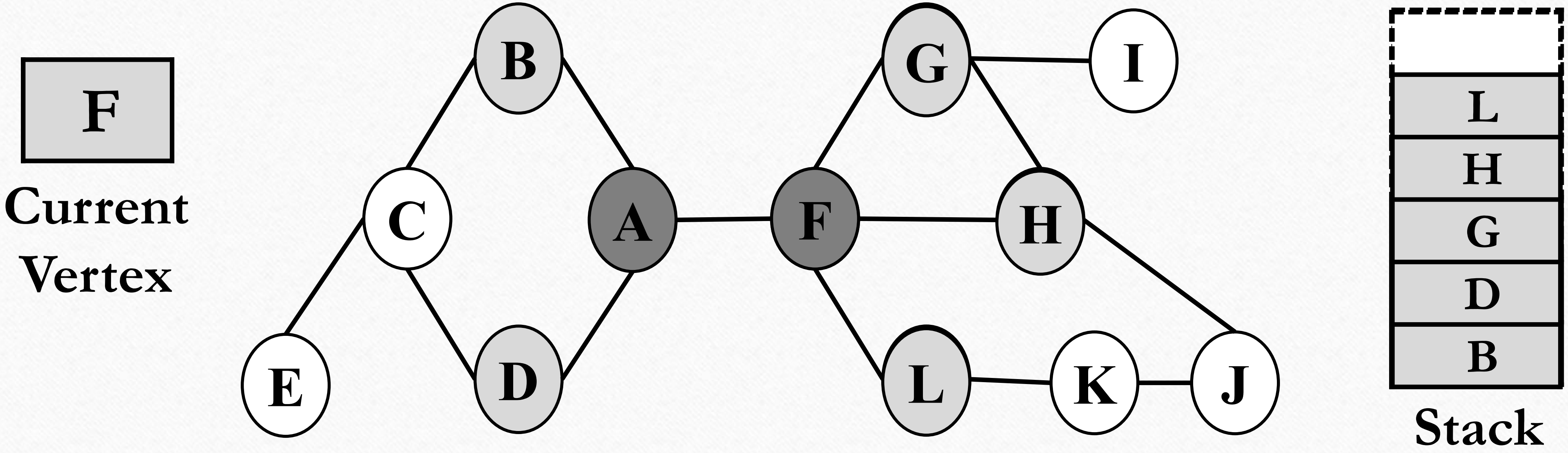
Next, we look at the vertices that vertex **A** is connected to, mark them as visited, and add them to the **stack**. The order we add these vertices doesn't matter, but for our example, we will add them in alphabetical order (i.e., **B** is inserted first, then **D**, then **F**).



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

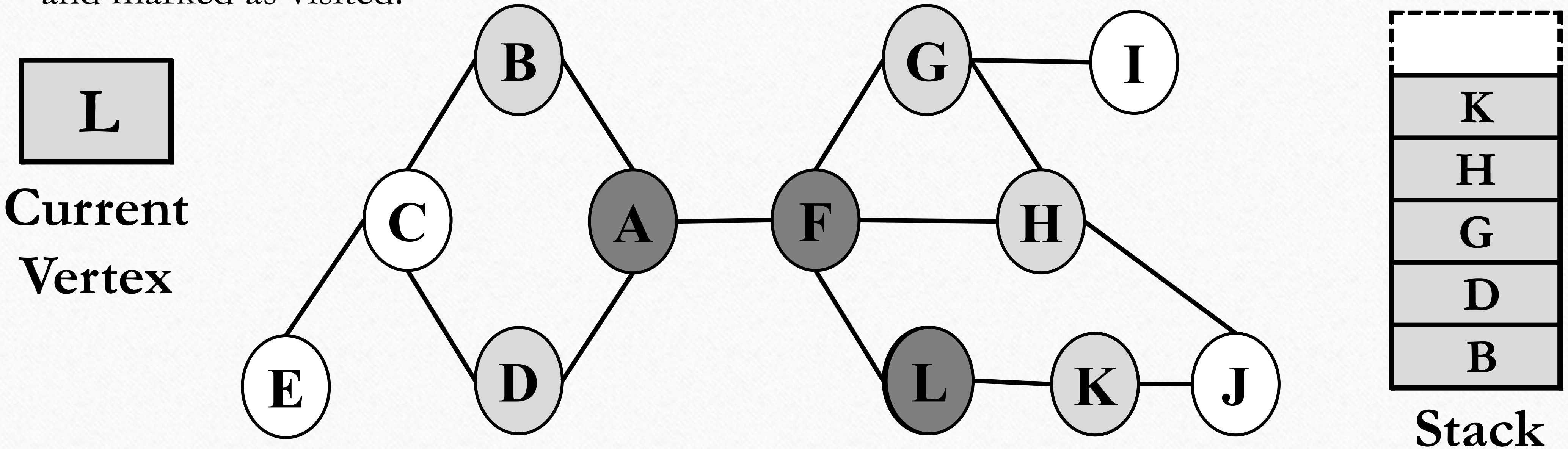
Next, we take out the vertex at the top of the stack and set it as our current vertex. Here, vertex **F** is at the top of the **stack**, so we pop it and set it as our current vertex. We then push all of **F** 's unvisited neighbors into the **stack**. Vertex F is connected to vertices **A, G, H**, and **L**, but only **G, H**, and **L** are unvisited. Thus, vertices **G, H,** and **L** are pushed into the **stack** and marked as visited.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

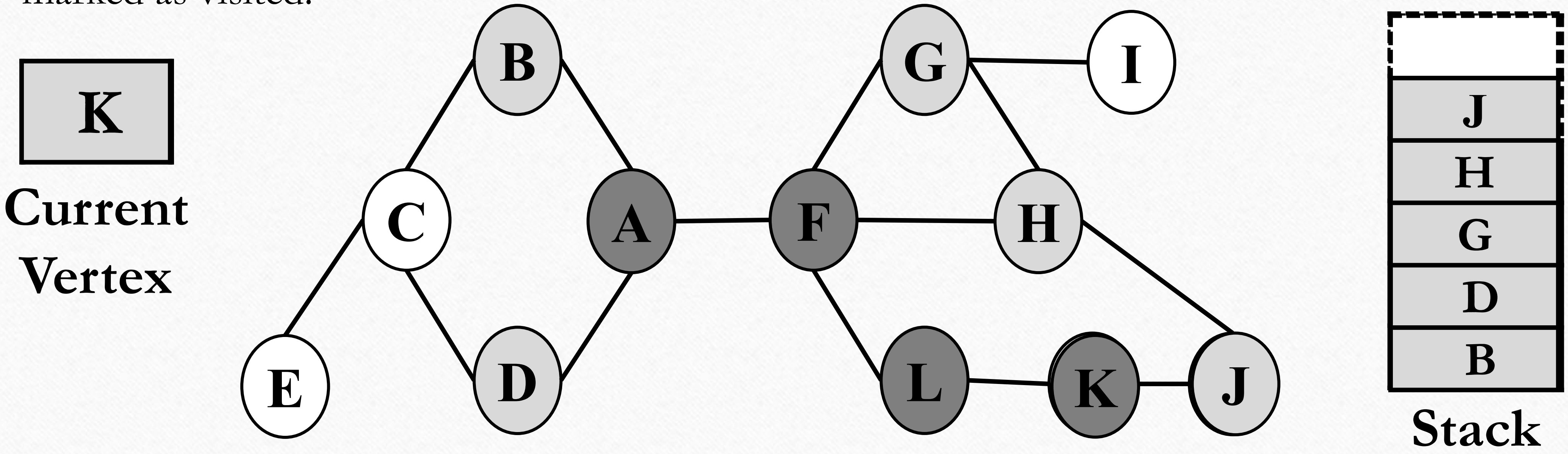
Vertex **L** is now at the top of the **`stack`**, so we pop it and set it as our current vertex. We then push all of vertex **L**'s unvisited neighbors into the **`stack`**. Vertex **L** is connected to vertices **F** and **K**, but only **K** is unvisited. Thus, only vertex **K** is pushed into the **`stack`** and marked as visited.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

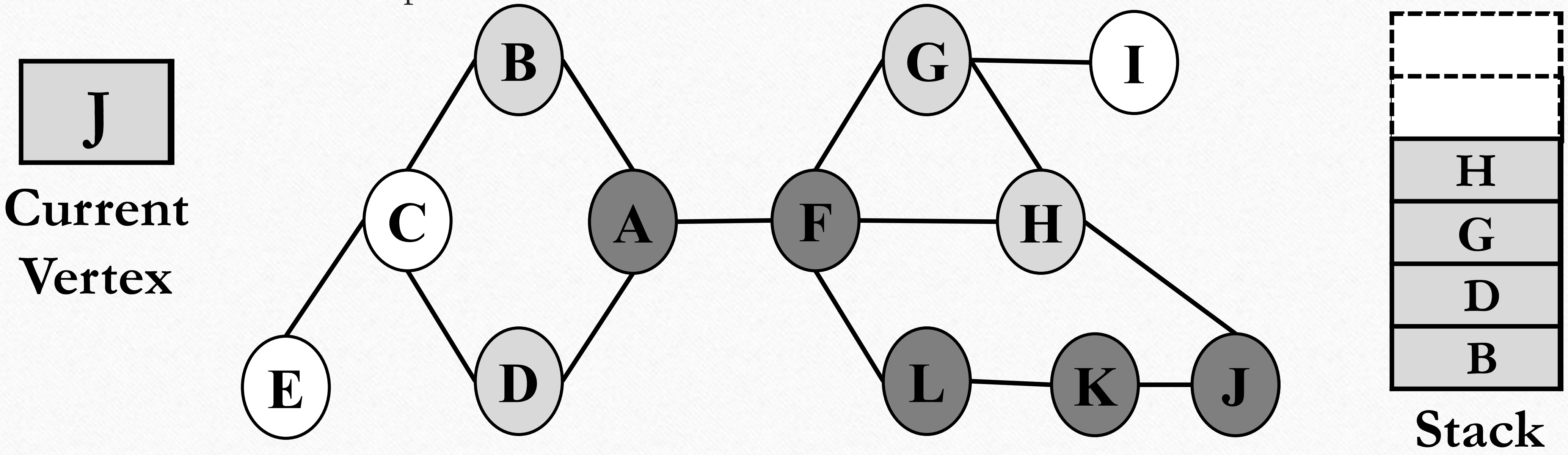
Vertex **K** is now at the top of the **`stack`**, so we pop it off and set it as our current vertex. We then push all of vertex **K**'s unvisited neighbors into the **`stack`**. Vertex **K** is connected to vertices **L** and **J**, but only **J** is unvisited. Thus, only vertex **J** is pushed into the stack and marked as visited.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

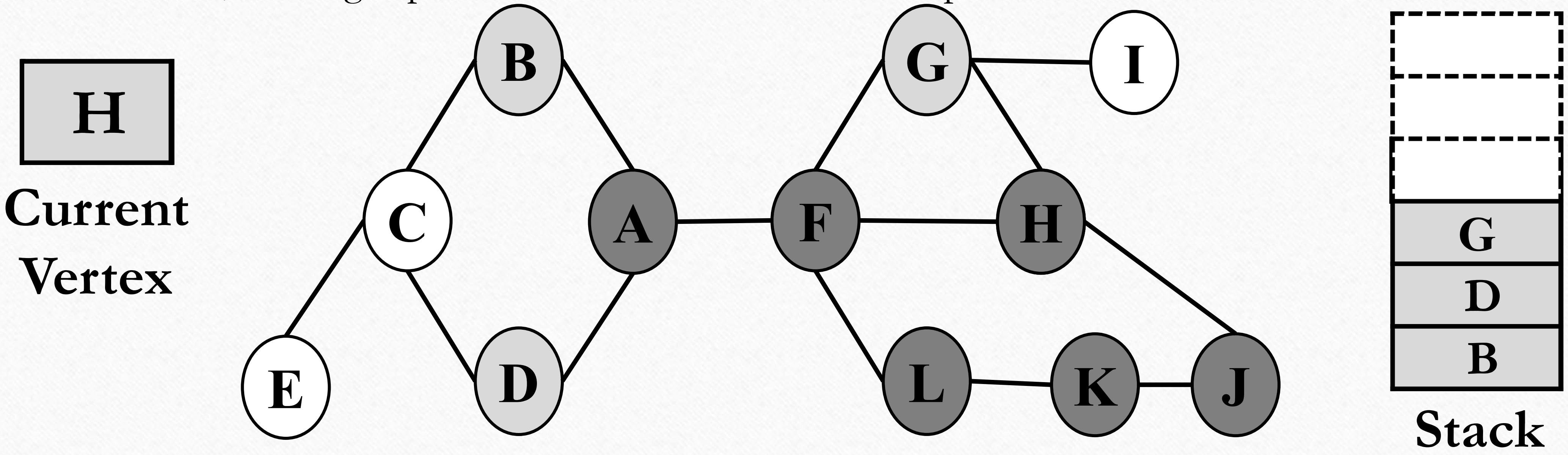
Vertex **J** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **J**'s unvisited neighbors into the **stack**. Vertex **J** is connected to vertices **H** and **K** however, both **H** and **K** have already been marked as visited. Thus, nothing is pushed into the **stack** at this step.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

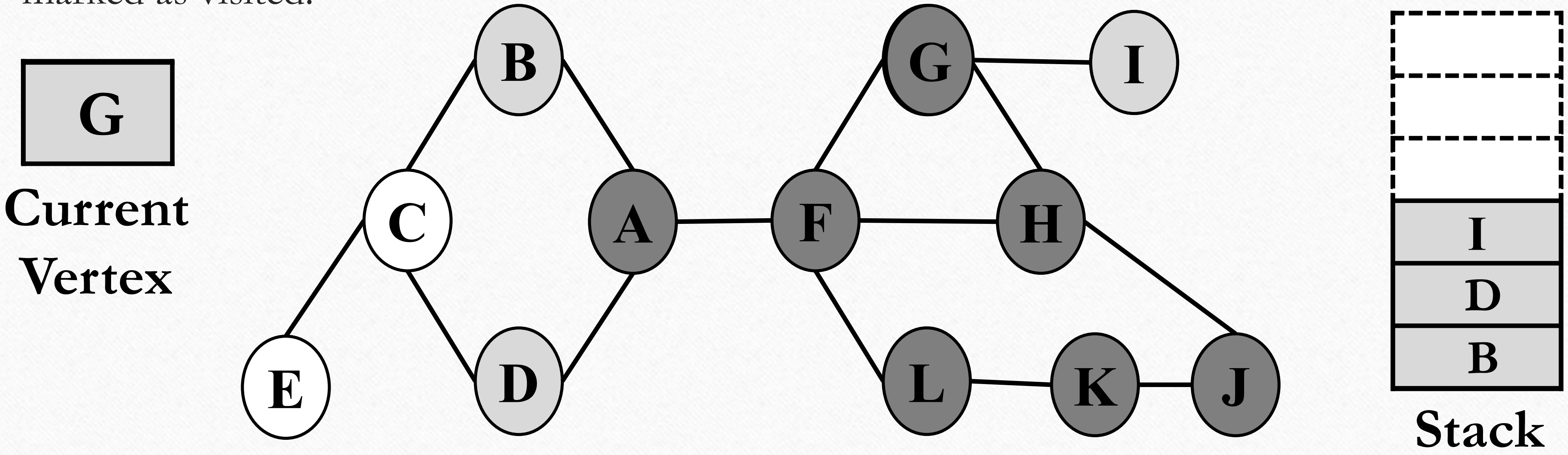
Vertex **H** is now at the top of the **`stack`**, so we pop it off and set it as our current vertex. We then push all of vertex **H**'s unvisited neighbors into the **`stack`**. Vertex H is connected to vertices **F**, **G**, and **J**, but similar to before, all three of these vertices have been marked as visited. Thus, nothing is pushed into the **`stack`** at this step.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

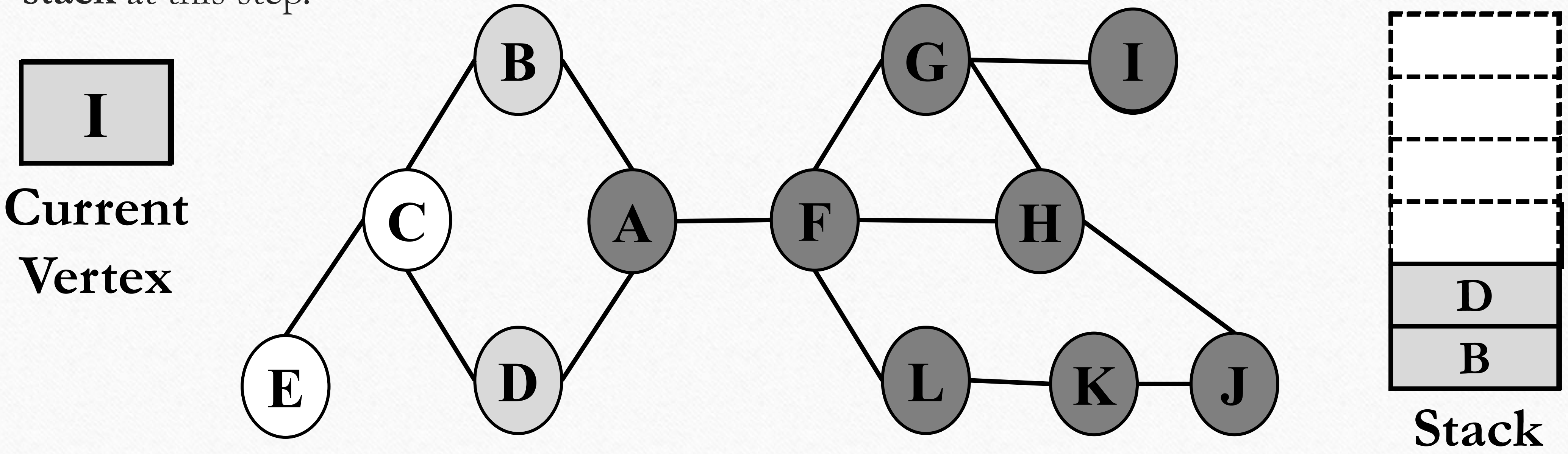
Vertex **G** is at the top of the **stack**, so we pop it and set it as our current vertex. We then push all of vertex **G**'s unvisited neighbors into the **stack**. Vertex **G** is connected to vertices **F** , **H**, and **I**, but only vertex **I** is unvisited. Thus, vertex **I** is pushed into the **stack** and marked as visited.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

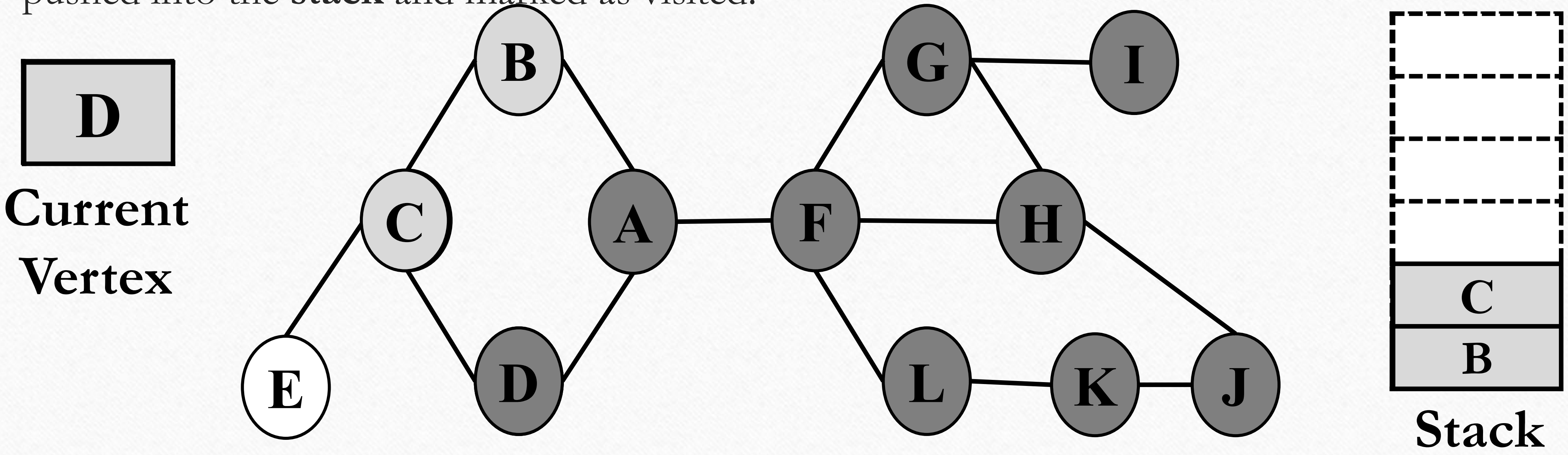
Vertex **I** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **I**'s unvisited neighbors into the **stack**. Vertex **I** is connected to vertex **G**, but vertex **G** has already been marked as visited. Thus, nothing is pushed into the **stack** at this step.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

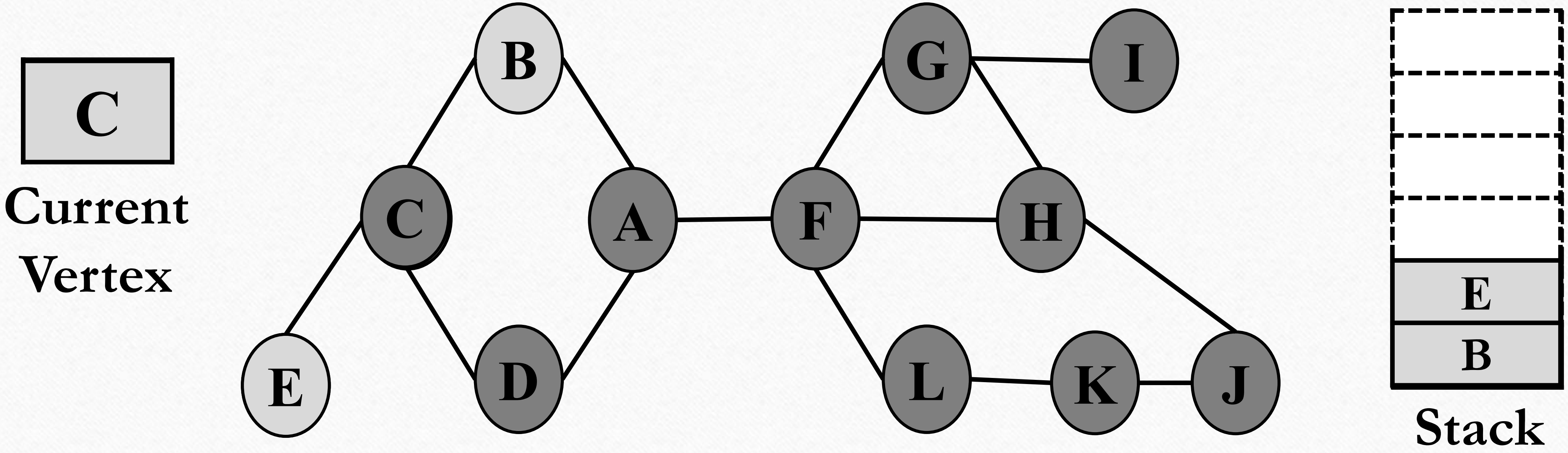
Vertex **D** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **D**'s unvisited neighbors into the **stack**. Vertex **D** is connected to vertices **A** and **C**, but vertex **A** has already been marked as visited. Thus, only vertex **C** is pushed into the **stack** and marked as visited.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

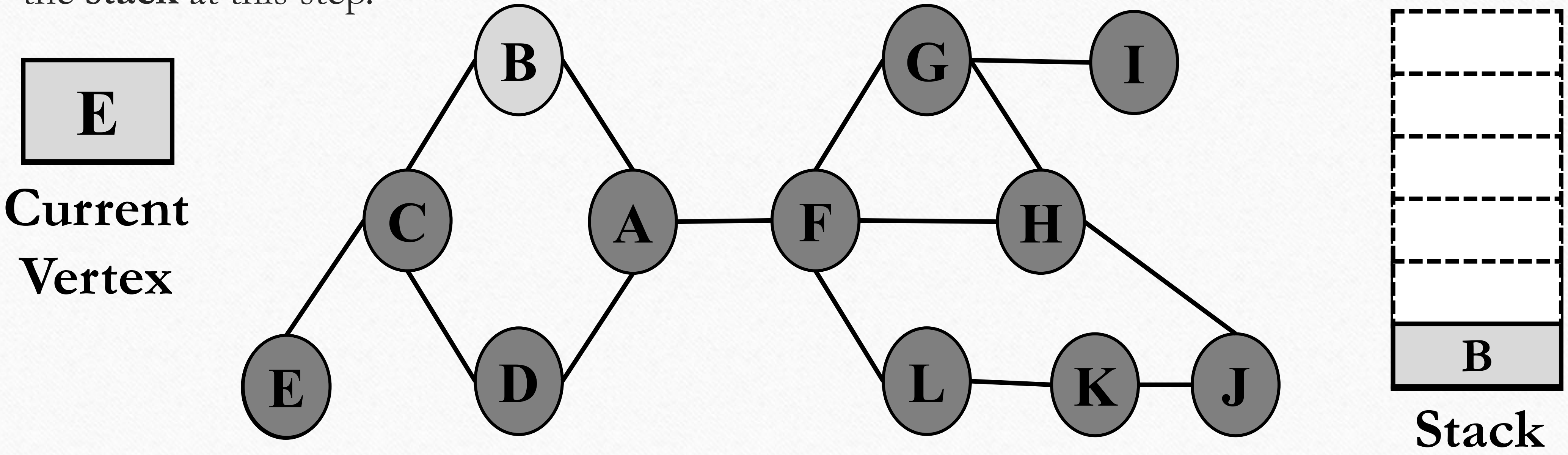
Vertex **C** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **C**'s unvisited neighbors into the **stack**. Vertex **C** is connected to vertices **B, D**, and **E**, but only vertex **E** has not yet been visited. Thus, vertex **E** is pushed into the **stack** and marked as visited.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

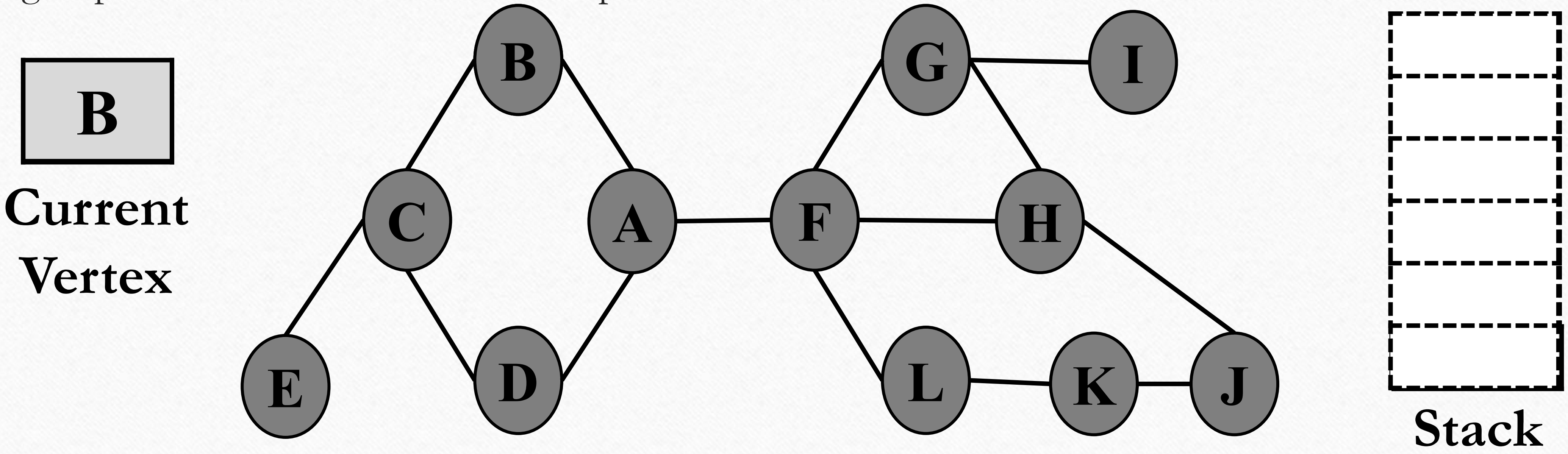
Vertex **E** is now at the top of the **stack**, so we pop it and set it as our current vertex. We then push all of vertex **E**'s unvisited neighbors into the **stack**. Vertex **E** is connected to vertex **C**, but vertex **C** has already been marked as visited. Thus, nothing gets pushed into the **stack** at this step.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

Vertex **B** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **B**'s unvisited neighbors into the **stack**. Vertex **B** is connected to vertices **A** and **C**, but both of these vertices have already been marked as visited. Thus, nothing gets pushed into the **stack** at this step.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

- The `stack` is now empty, so our **depth-first search (DFS)** is complete, and we have processed every vertex reachable from vertex **A**.

The DSF traversal order of the vertices is as follows:

A →F → L → K → J → H → G → I → D → C → E → B

V. Graph Traversal (Depth First Search (DFS))

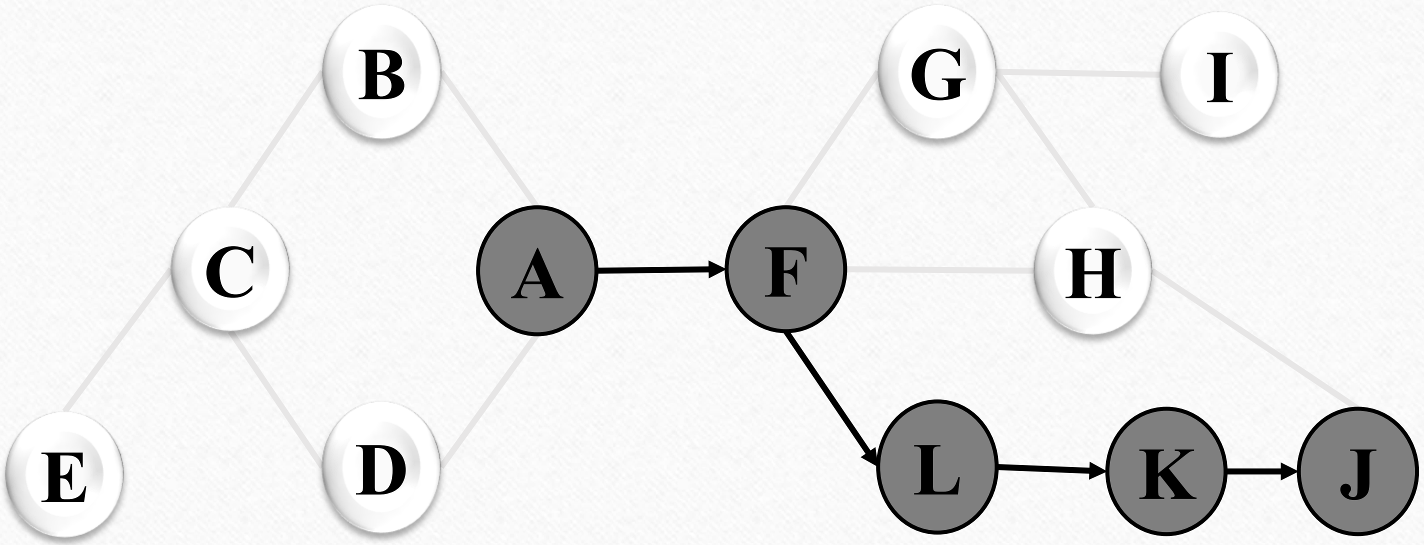
Example : (Algorithm DFS)

- Notice that, because a depth-first search relies on a **`stack`** to process the nodes in a graph, the algorithm explores and processes vertices in the graph one branch at a time in **LIFO** order.

V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

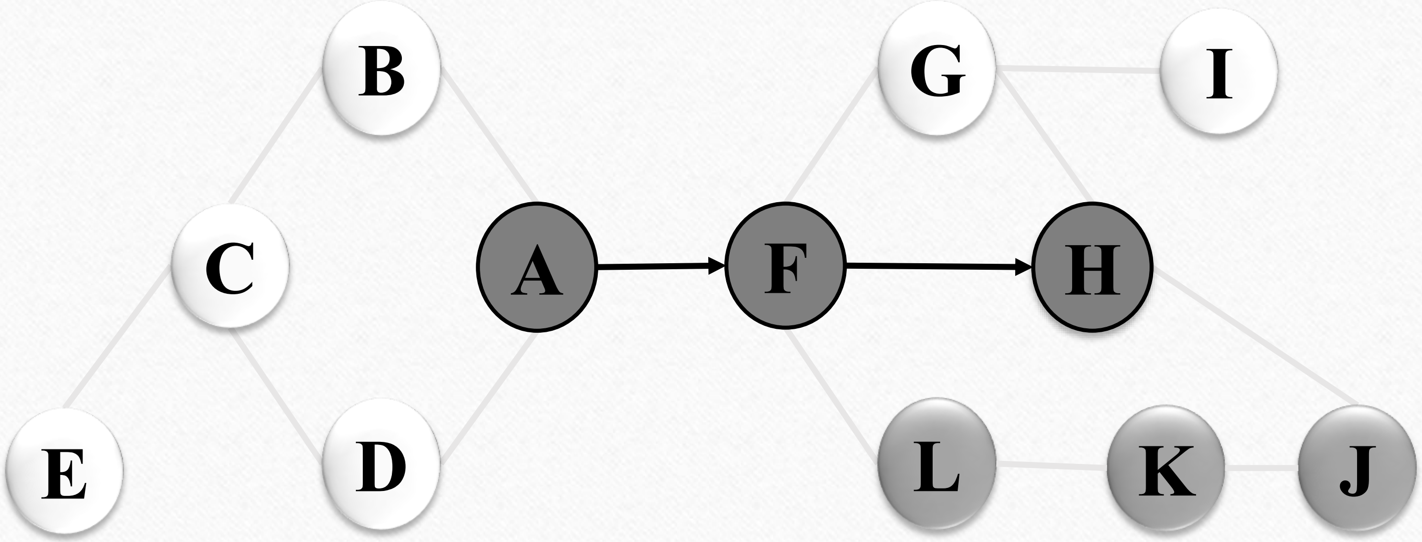
- Here, the depth-first search first processed the branch $A \rightarrow F \rightarrow L \rightarrow K \rightarrow J$:



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

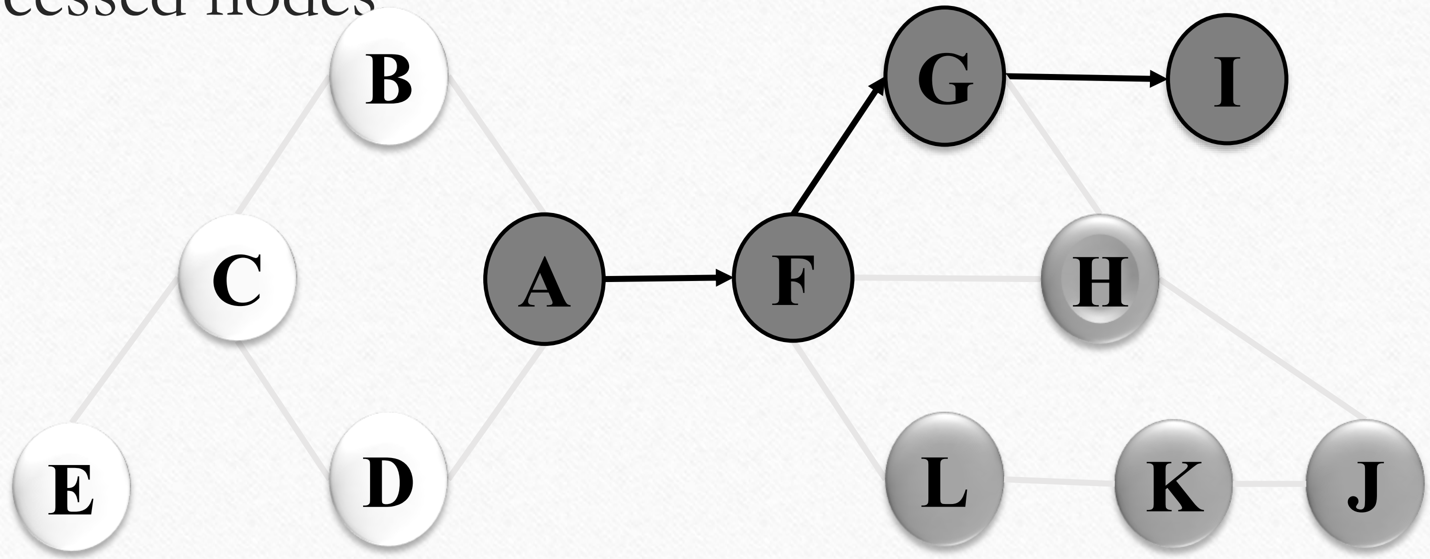
- After processing these five vertices, the search then backtracks to vertex **F** and explores the new branch $\boldsymbol{A \rightarrow F \rightarrow H}$, which adds vertex **H** to the list of processed nodes.



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

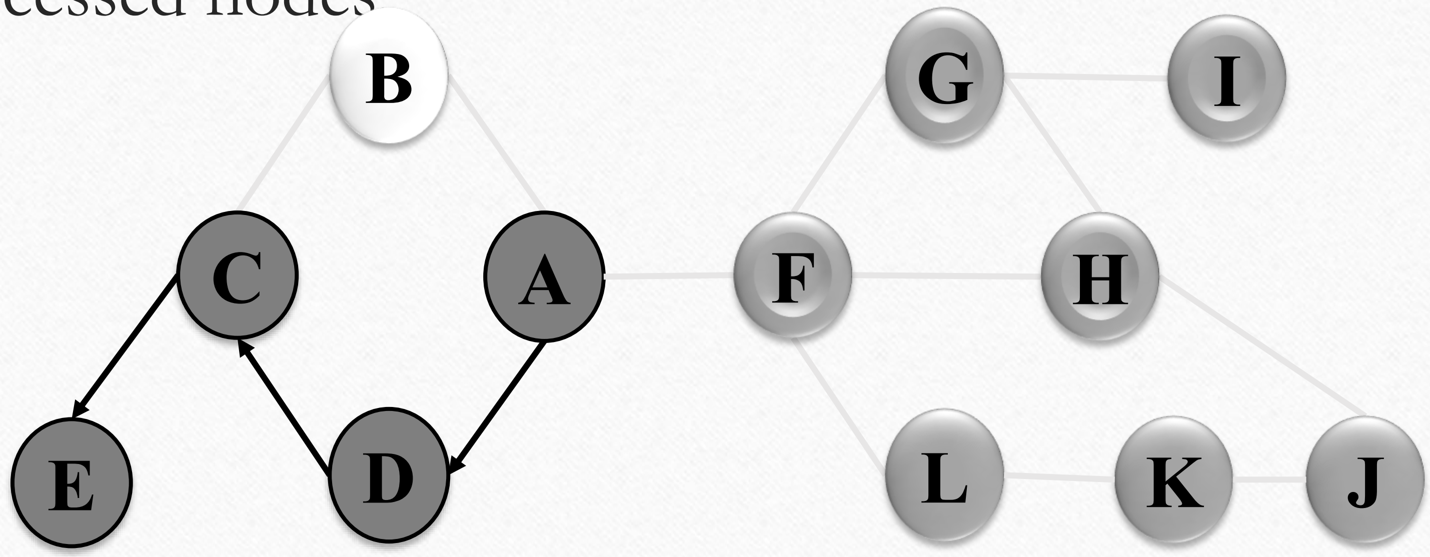
- The search then backtracks to vertex **F** and explores the branch $A \rightarrow F \rightarrow G \rightarrow I$, which adds vertices **G** and **I** to the list of processed nodes



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

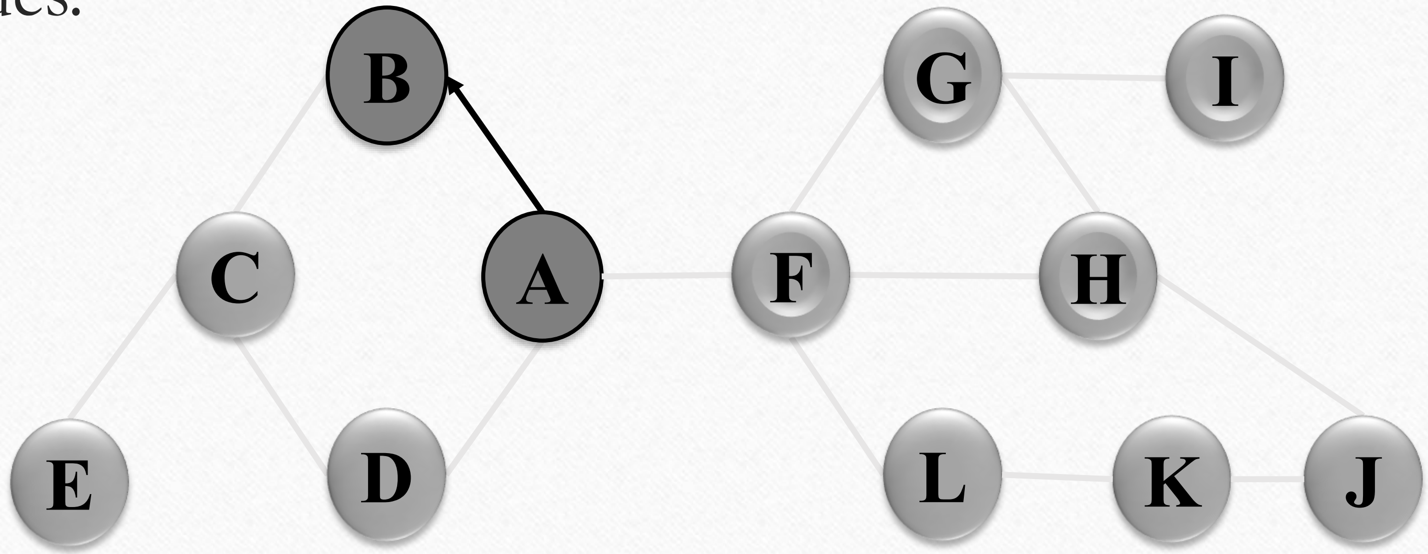
- The search then backtracks to vertex **A** and explores the branch ***A* → *D* → *C* → *E***, adding vertices **D, C,** and **E** to the list of processed nodes



V. Graph Traversal (Depth First Search (DFS))

Example : (Algorithm DFS)

- The search then backtracks to vertex **A** and explores the final branch ***A*** → ***B***, which adds vertex **B** to the list of processed nodes.



V. Graph Traversal

Breadth First Search (BFS)

- The breadth-first search (BFS) algorithm works **similar to level order traversal of the trees**, but this algorithm also uses **queues**.
- In fact, level order traversal got inspired from **BFS**.
- BFS works level by level.
- The breadth-first search (BFS) algorithm is another graph traversal algorithm that can be used to explore the vertices and edges of a graph that are **reachable** from a given source vertex.
- In a breadth-first search, the algorithm **starts at a source vertex** and explores vertices of the graph in a breadth-ward fashion, **in order of increasing edge distance from the source node.**

V. Graph Traversal (Breadth First Search (BFS))

The implementation of a breadth-first search relies on the **queue<>** data structure. An outline of the algorithm is as follows:

1) Mark the source vertex as visited, and then push its neighbors (i.e., adjacent vertices) into a **queue**.

2) Pop the vertex at the front of the **queue** and set it as the current vertex.

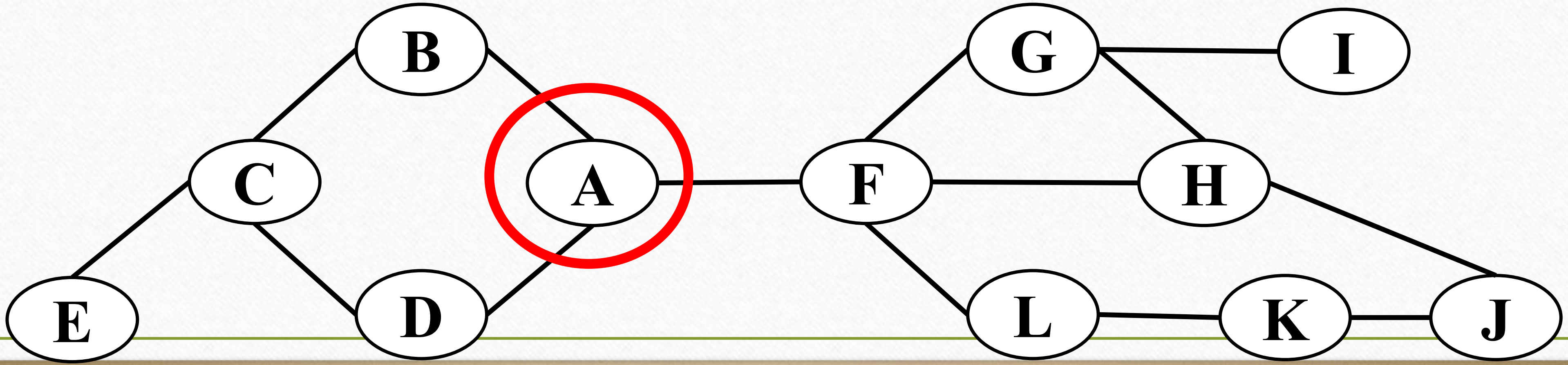
3) Push the unvisited neighbors of the current vertex into the **queue** and mark them as visited.

4) Repeat **steps 2** and **3** until the **queue** is empty.

V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

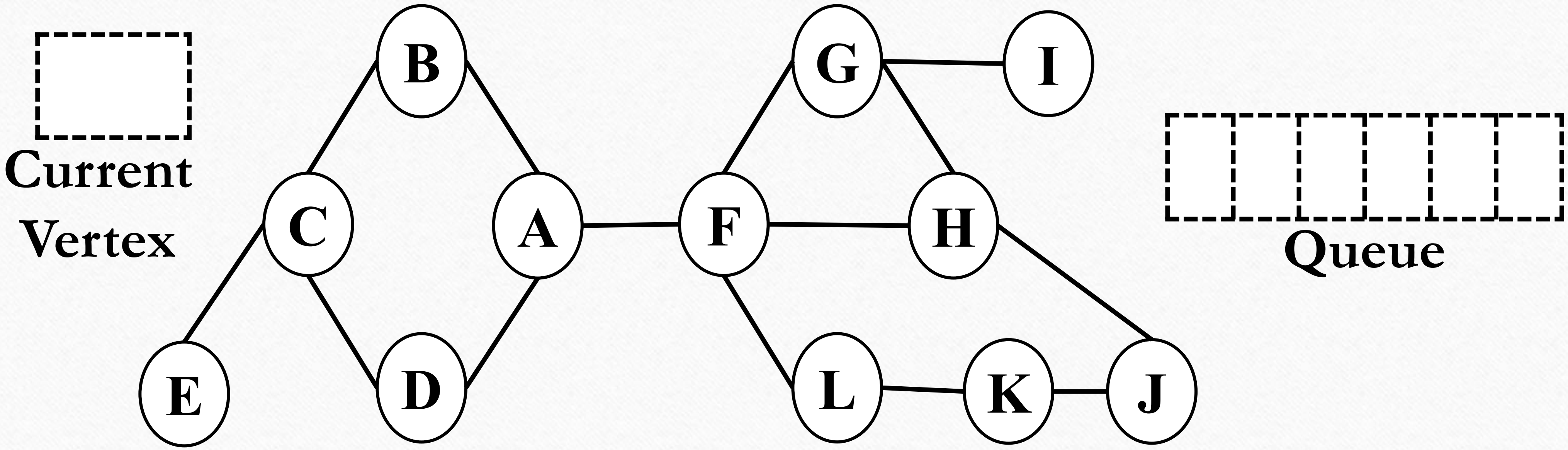
- As an example, consider the following graph, which we will traverse using a breadth-first search starting at vertex **A**.
- Similar to the depth-first search example, **discovered** vertices will be given a **light gray** shading,
- while **processed** vertices will be given a **dark gray** shading.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

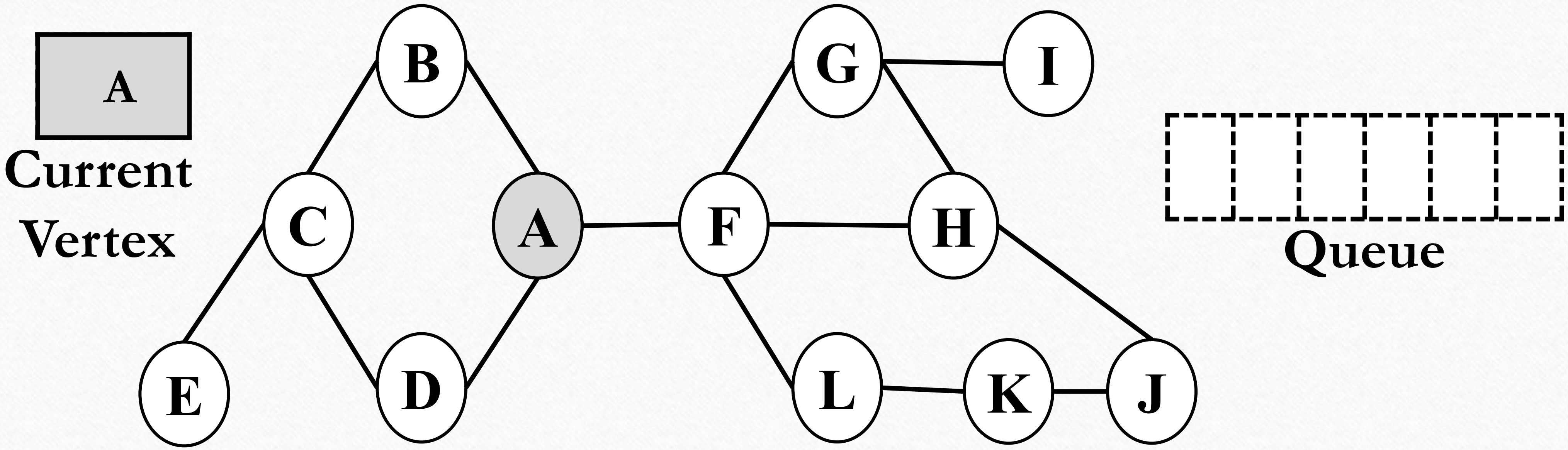
We start the breadth-first search process by creating a **queue<>** that will be used to keep track of the vertices that we still need to explore.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

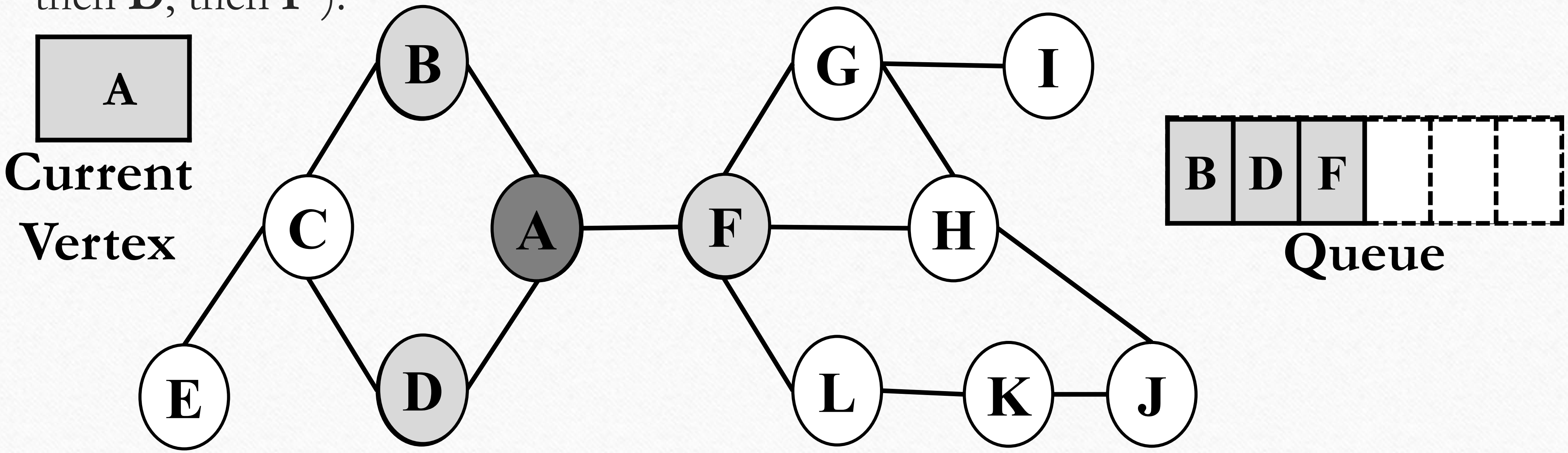
The first step is to mark the starting vertex as visited. In this case, our starting vertex is vertex **A**, so we will mark **A** as visited.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

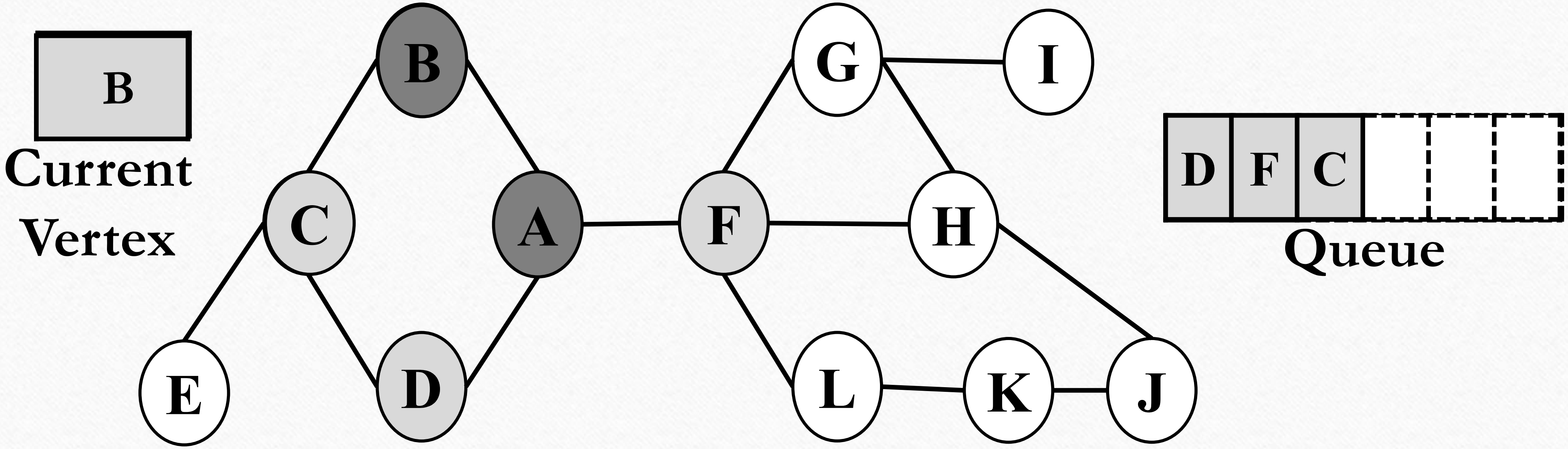
Next, we look at the vertices that vertex **A** is connected to, mark them as visited, and add them to the **queue**. The order we add these vertices does not matter, but for our example, we will add them in alphabetical order (i.e., **B** is inserted first, then **D**, then **F**).



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

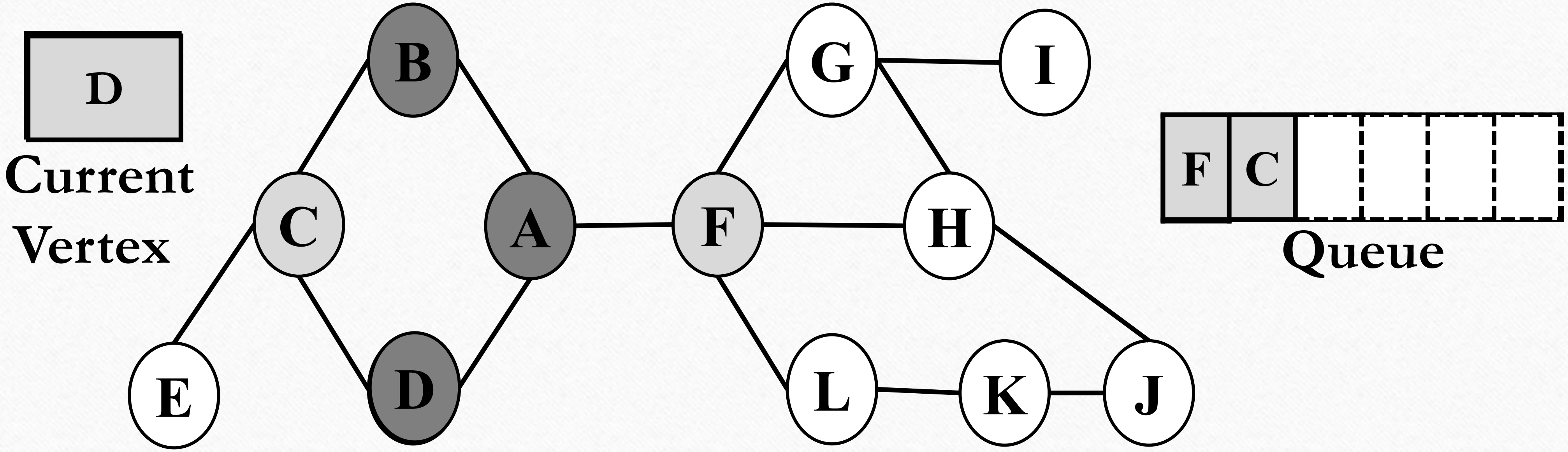
We then take out the vertex at the front of the **queue**, vertex **B**, and set it as our current vertex. Since vertex **B** is our current vertex, we push all of its unvisited neighbors into the **queue**. Vertex **B** is connected to vertices **A** and **C**, but only vertex **C** is marked as unvisited, so vertex **C** gets pushed into the **queue** and marked as visited.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

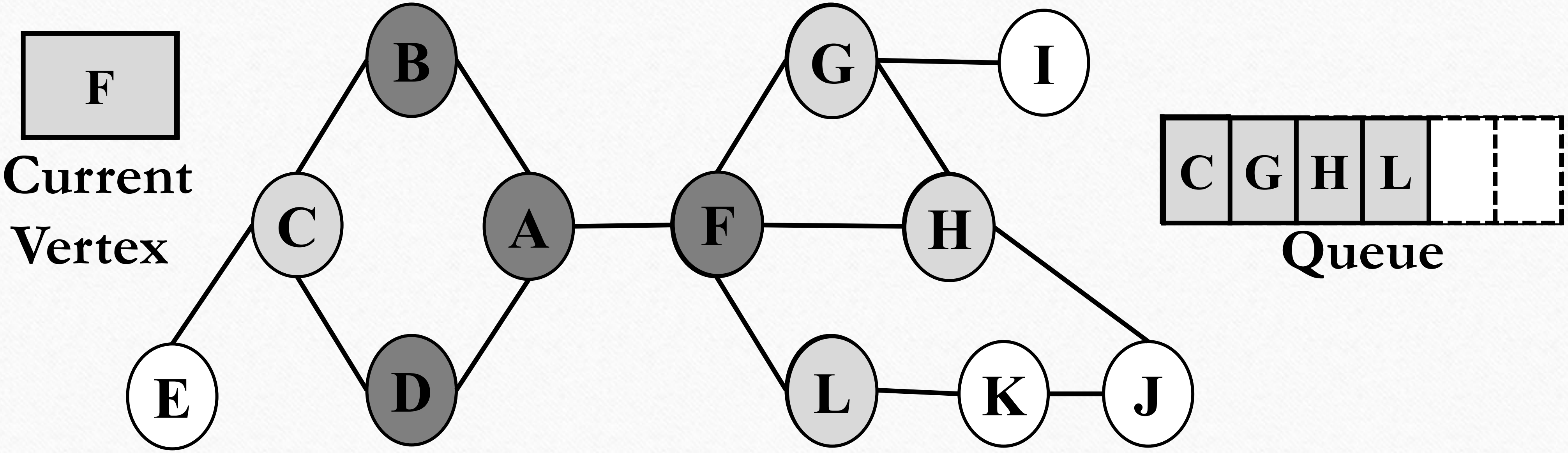
Vertex **D** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **D**'s unvisited neighbors into the **queue**. Vertex **D** is connected to vertices **A** and **C**, but both **A** and **C** have been marked as visited. Thus, nothing is pushed into the **queue** at this step.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

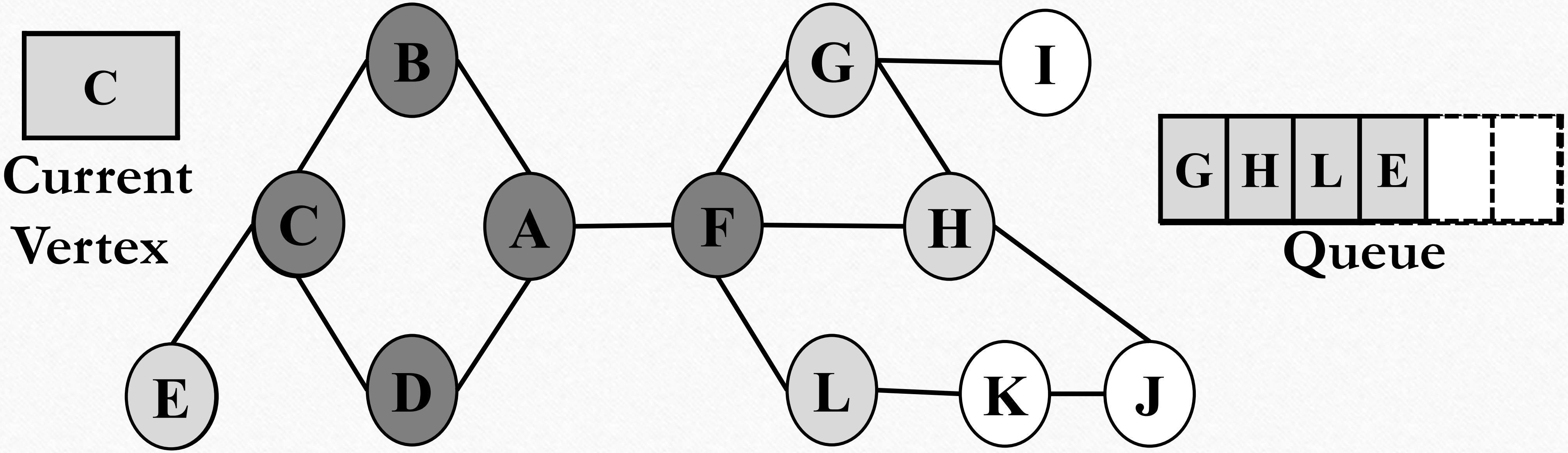
Vertex **F** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **F** 's unvisited neighbors into the **queue**. Vertex **F** is connected to vertices **A, G, H**, and **L**, but only vertex **A** has been marked as visited. Thus, vertices **G, H,** and **L** are pushed into the **queue** and marked as visited.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

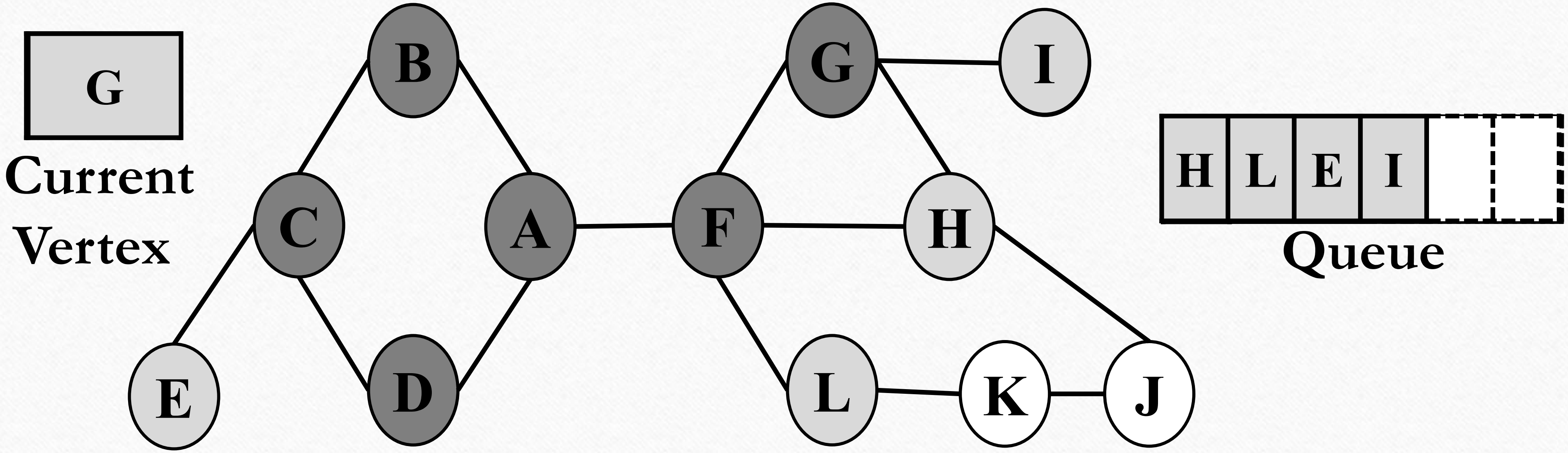
Vertex **C** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **C**’s unvisited neighbors into the **queue**. Vertex **C** is connected to vertices **B, D,** and **E**, but only vertex **E** has not yet been marked as visited. Thus, vertex **E** is pushed into the **queue** and marked as visited.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

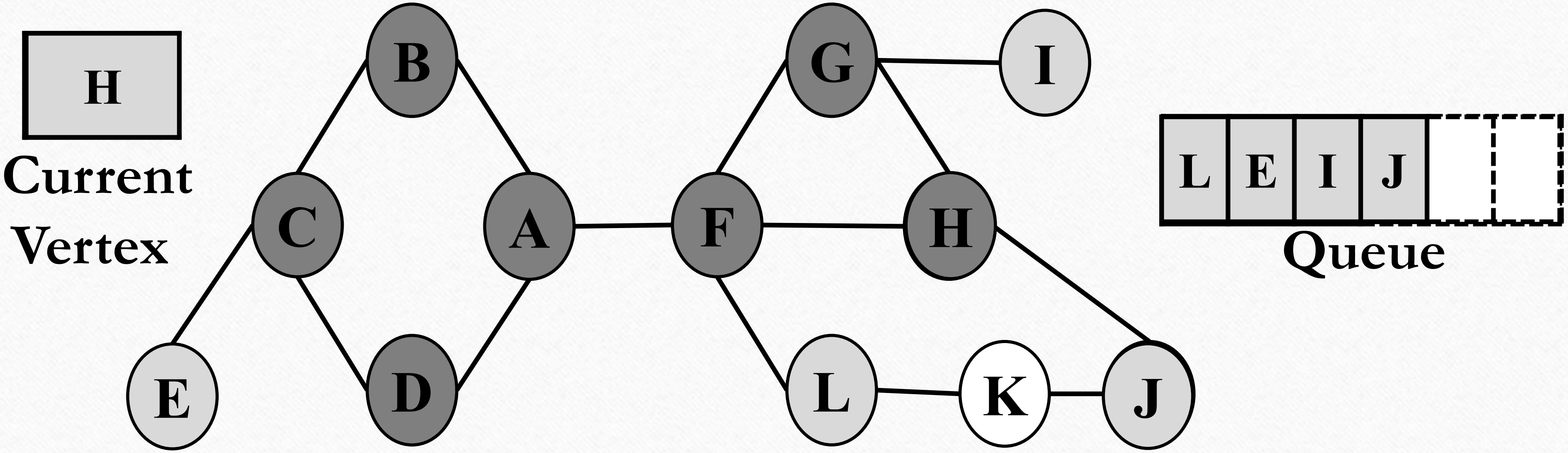
Vertex **G** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **G**'s unvisited neighbors into the **queue**. Vertex **G** is connected to vertices **F, H**, and **I**, but only vertex **I** has not yet been marked as visited. Thus, vertex **I** is pushed into the **queue** and marked as visited.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

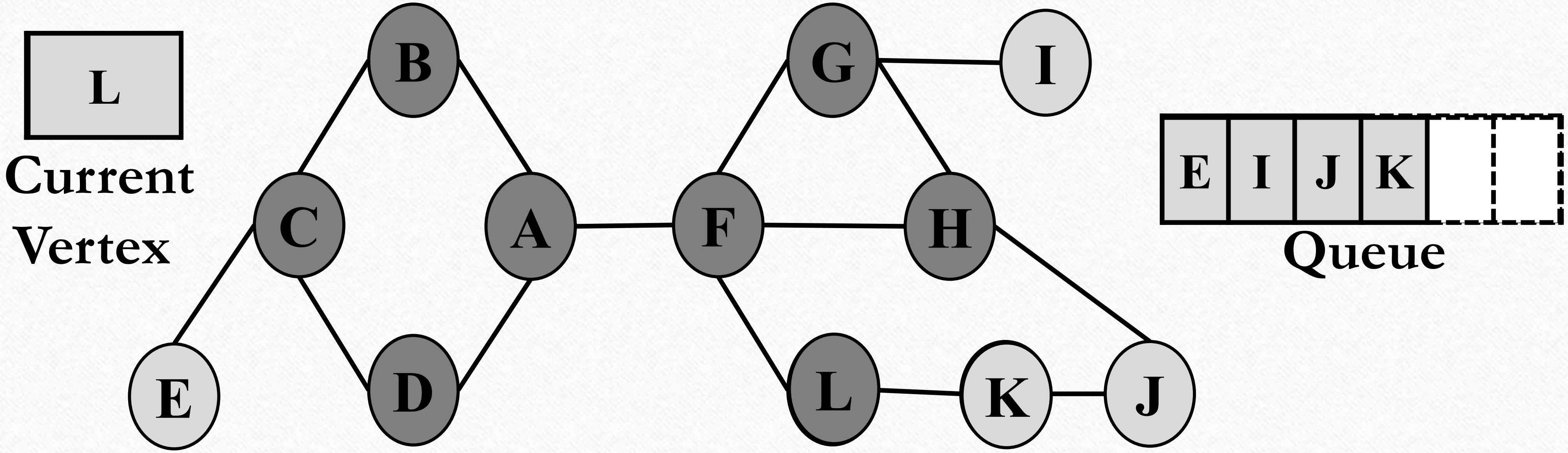
Vertex **H** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **H**'s unvisited neighbors into the **queue**. Vertex **H** is connected to vertices **F, G**, and **J**, but only vertex **J** has not yet been marked as visited. Thus, vertex **J** is pushed into the **queue** and marked as visited.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

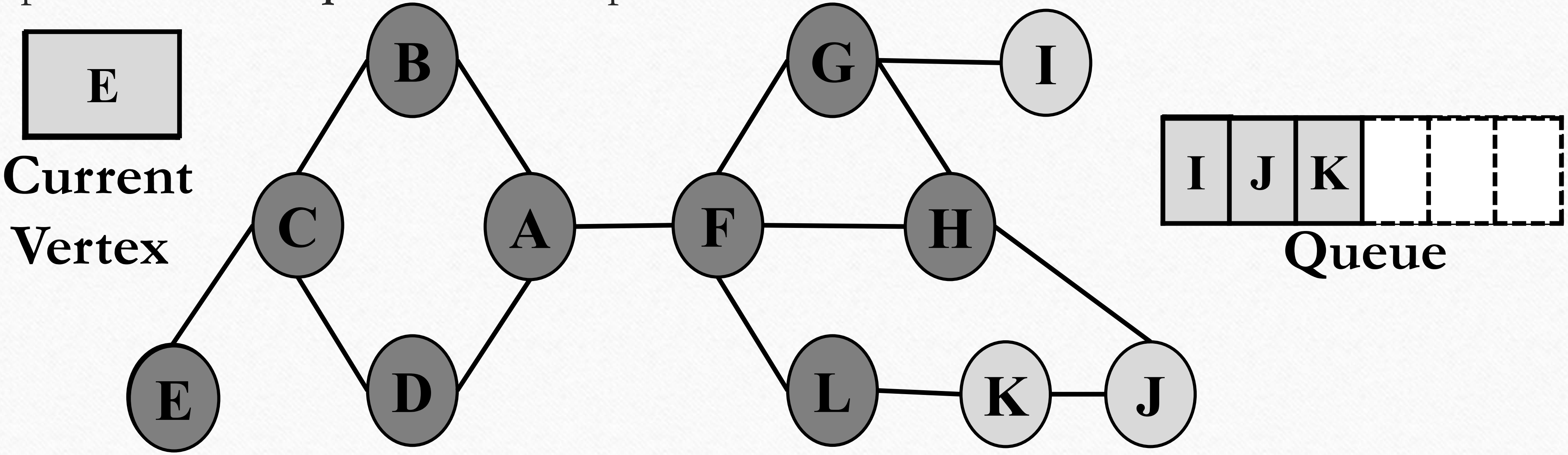
Vertex **L** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **L**'s unvisited neighbors into the **queue**. Vertex **L** is connected to vertices **F** and **K**, but only vertex **K** has not yet been marked as visited. Thus, vertex **K** is pushed into the **queue** and marked as visited.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

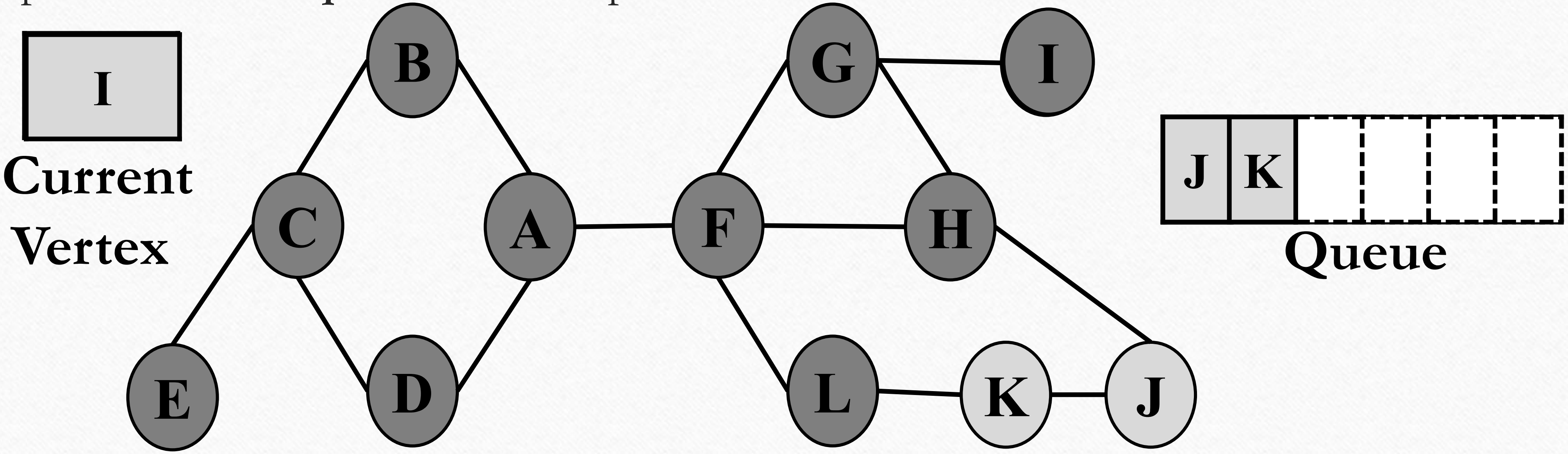
Vertex **E** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **E**'s unvisited neighbors into the **queue**. Vertex **E** is connected to vertex **C**, which has already been marked as visited. Thus, nothing is pushed into the **queue** at this step.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

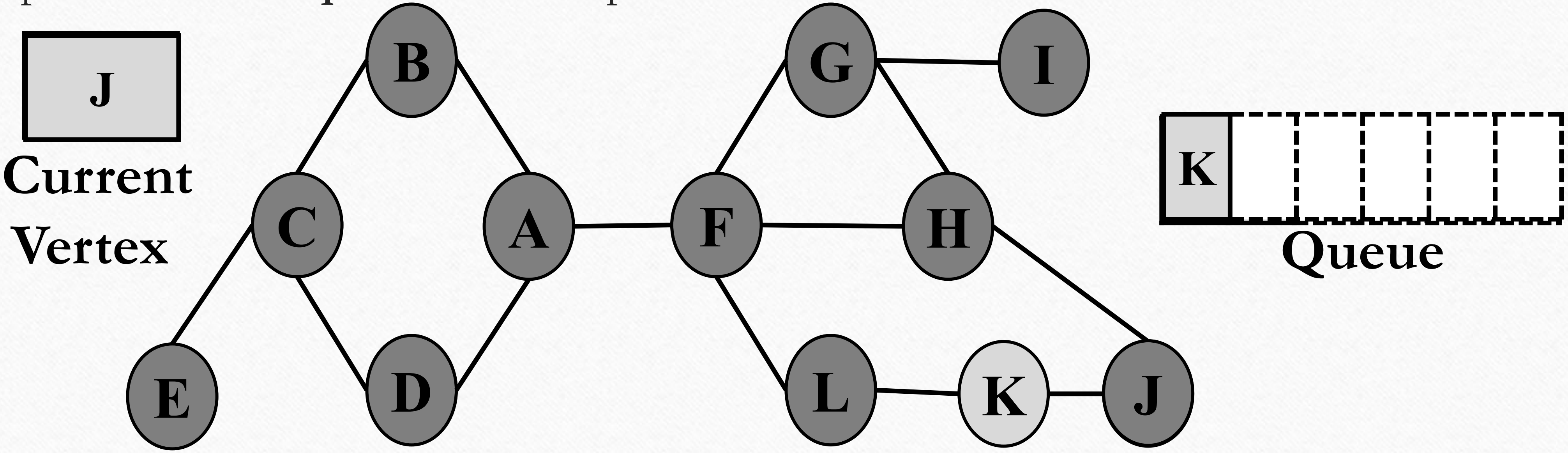
Vertex **I** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **I**'s unvisited neighbors into the **queue**. Vertex **I** is connected to vertex **G**, which has already been marked as visited. Thus, nothing is pushed into the **queue** at this step.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

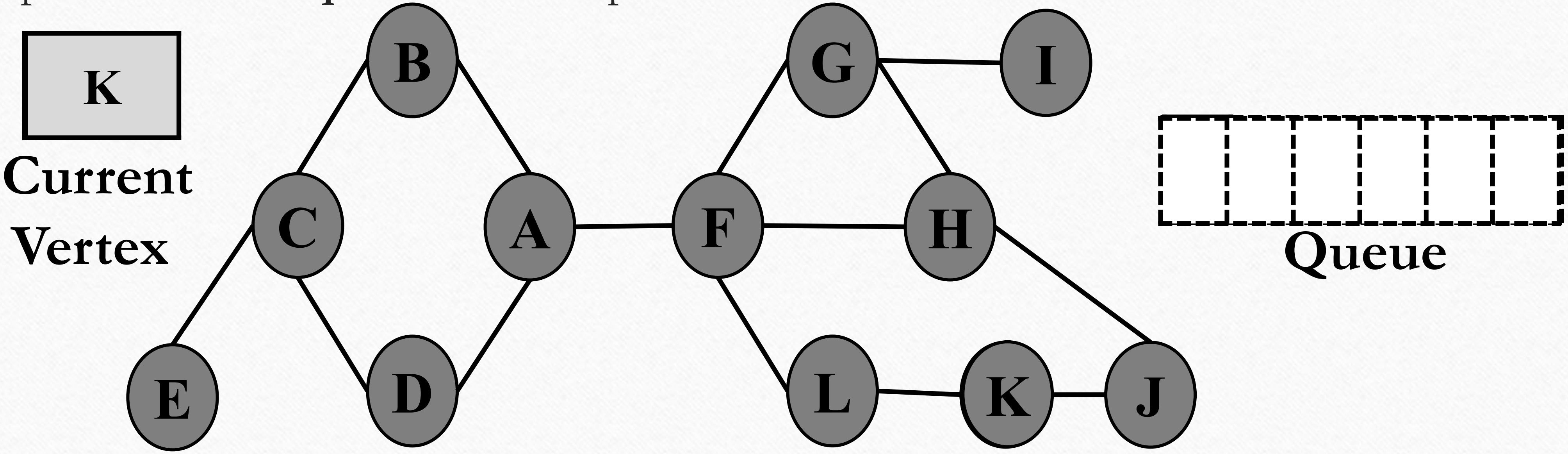
Vertex **J** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **J**'s unvisited neighbors into the **queue**. Vertex **J** is connected to vertices **H** and **K**, but both have already been visited. Thus, nothing is pushed into the **queue** at this step.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

Vertex **K** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **K**'s unvisited neighbors into the **queue**. Vertex **K** is connected to vertices **J** and **L**, but both have already been visited. Thus, nothing is pushed into the **queue** at this step.



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

- The **queue** is now empty, so our breadth-first search is complete, and we have successfully processed every vertex in the graph.

The BSF traversal order of the vertices is as follows:

A → B → D → F → C → G → H → L → E → I → J → K

V. Graph Traversal (Breadth First Search (BFS))

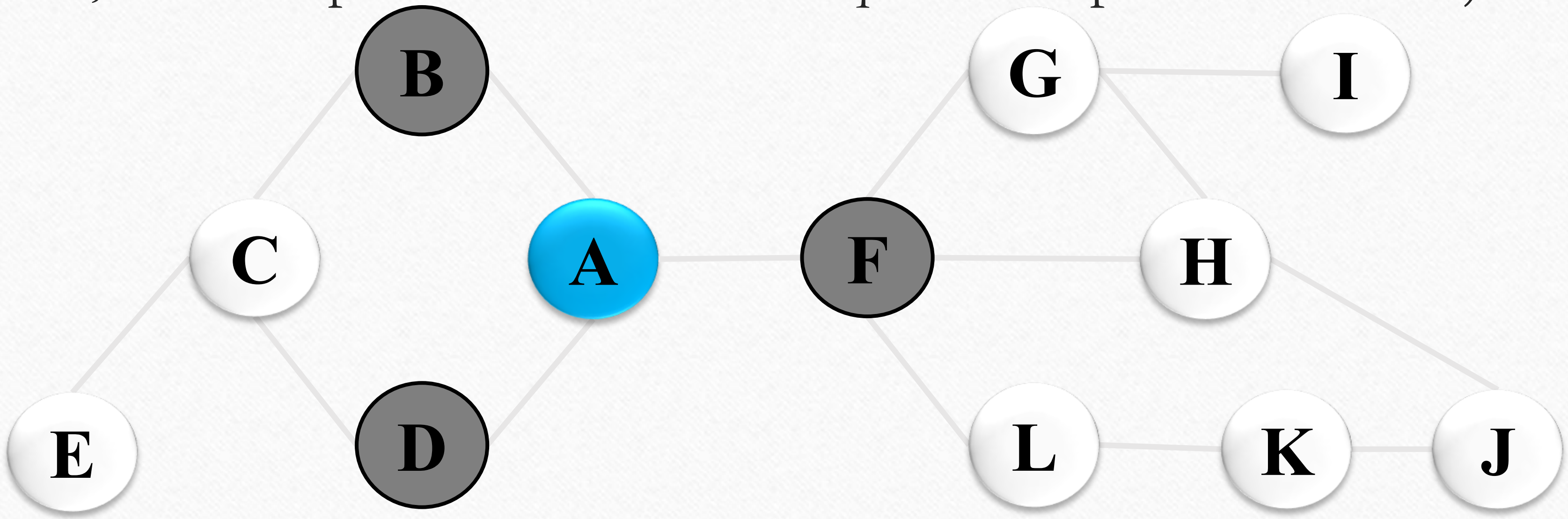
Example : (Algorithm BFS)

- Because a breadth-first search relies on a **queue** to process nodes in a graph, the algorithm processes vertices in the graph in **FIFO** order.
- As a result, a breadth-first search ends up visiting vertices in order of increasing edge distance from the source vertex:
- vertices that are closer to the source vertex are pushed into the **queue before** vertices that are farther away, and are thus taken out of the queue and processed earlier as well!

V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

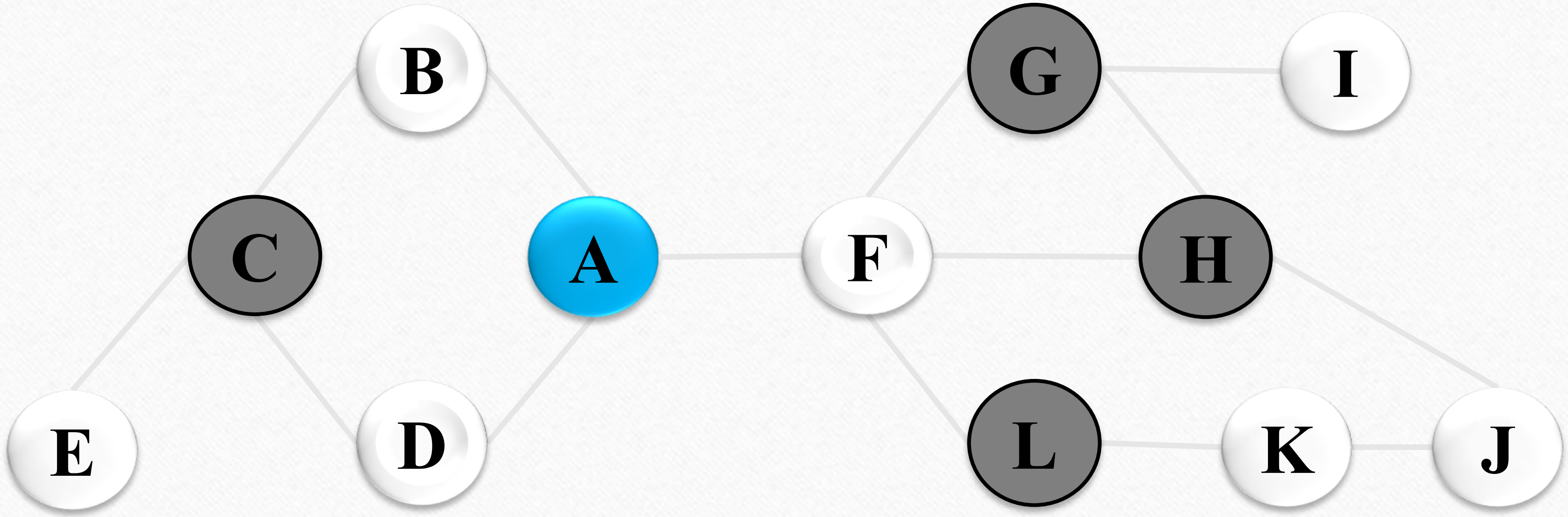
- In our example, the breadth-first search first processed vertices that were one edge away from the source vertex **A**: vertices **B, D,** and **F** (in this order, since we pushed them into the queue in alphabetical order).



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

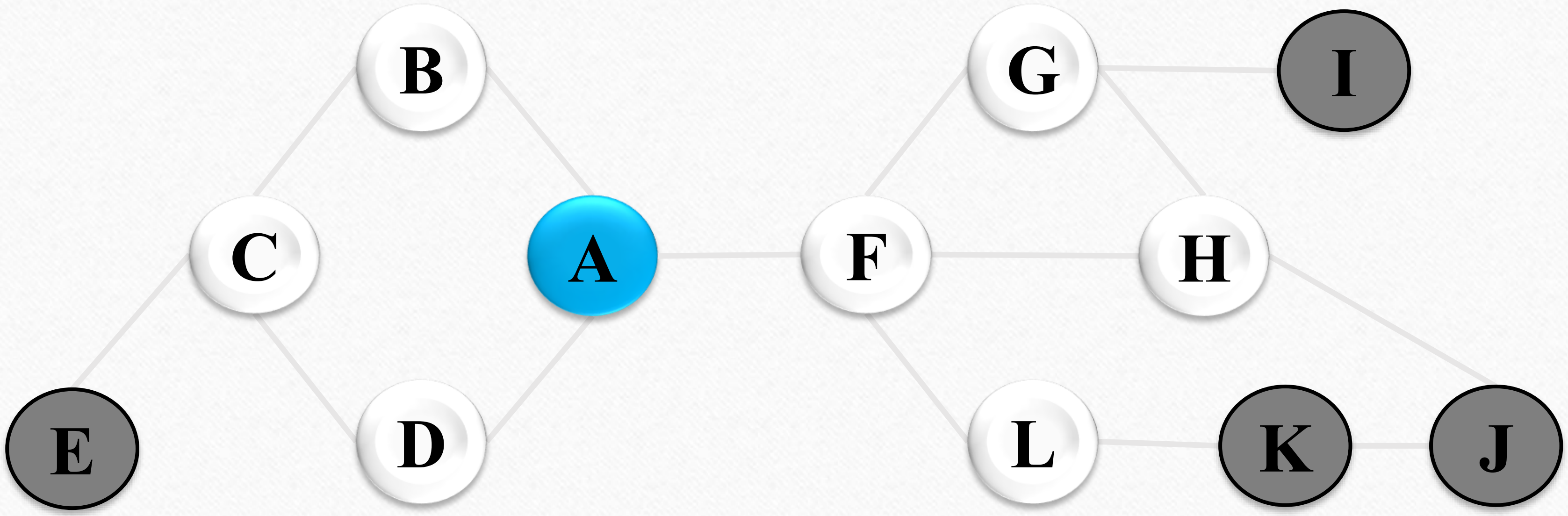
- Then, the breadth-first search processed vertices that were two edges away from the source vertex **A**:



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

- Then, the breadth-first search processed vertices that were three edges away from the source vertex **A**:



V. Graph Traversal (Breadth First Search (BFS))

Example : (Algorithm BFS)

- Had the graph been larger, the breadth-first search would then have processed vertices four edges away, then five edges away, and so on. Because breadth-first searches exhibit this behavior, they can always be used to find the **shortest path between any two vertices in an unweighted graph, or a graph where all edge weights are the same.**
- As a result, breadth-first searches are often useful for solving shortest path graph problems, particularly for unweighted graphs.

V. Graph Traversal (Breadth First Search (BFS))

DFS and BFS Time Complexity

Graph Representation	Adjacency List	Adjacency Matrix
DFS Time Complexity	$\mathcal{O}(\lvert V\rvert + \lvert \mathbf{E}\rvert)$	$\mathcal{O}(\lvert V\rvert^2)$
BFS Time Complexity	$\mathcal{O}(\lvert V\rvert + \lvert \mathbf{E}\rvert)$	$\mathcal{O}(\lvert V\rvert^2)$

Questions ?