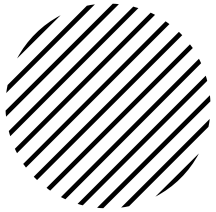
CHAPTER 7
GRAPH

ผศ.ดร.สิลดา อินทรโสธรฉันท์

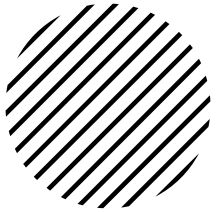
สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยขอนแก่น

Graph



- A graph data structure mainly represents a network connecting various points.
- Termed as vertices and the links connecting these vertices are called ‘Edges’.
- A graph g is defined as a set of vertices V and edges E that connect these vertices.

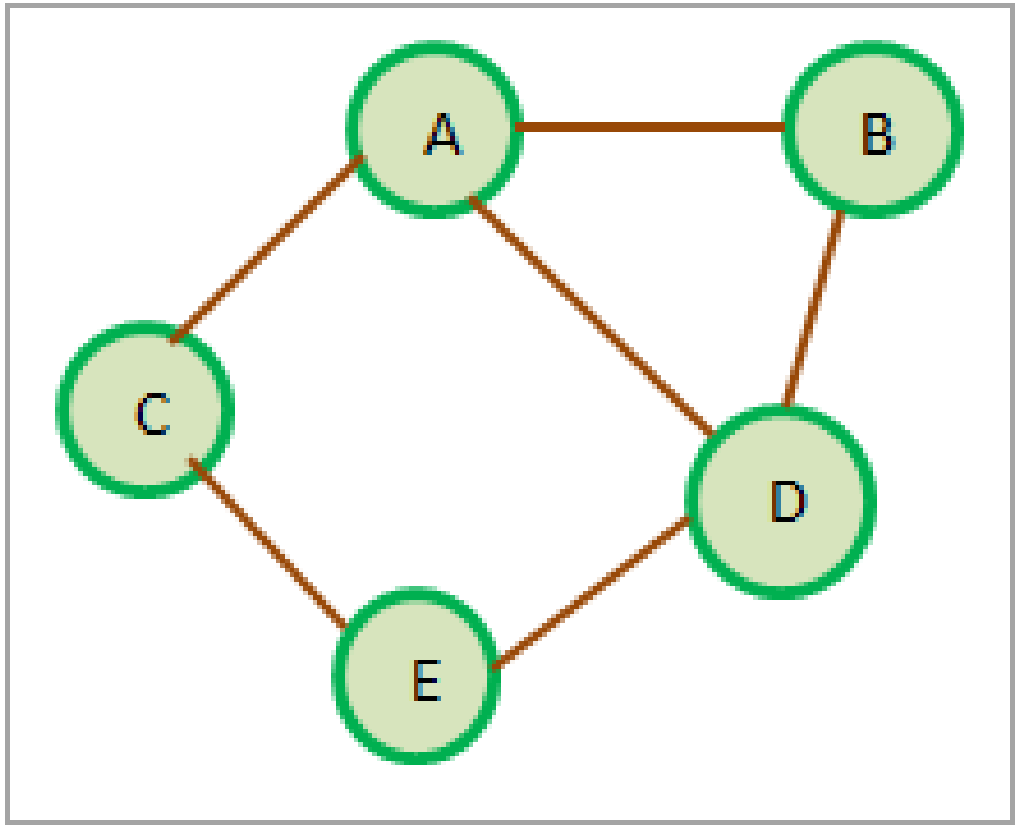
Graph



- Graphs are mostly used to represent various networks like computer networks, social networks, etc.
- They can also be used to represent various dependencies in software or architectures.
- These dependency graphs are very useful in analyzing the software and also at times debugging it.

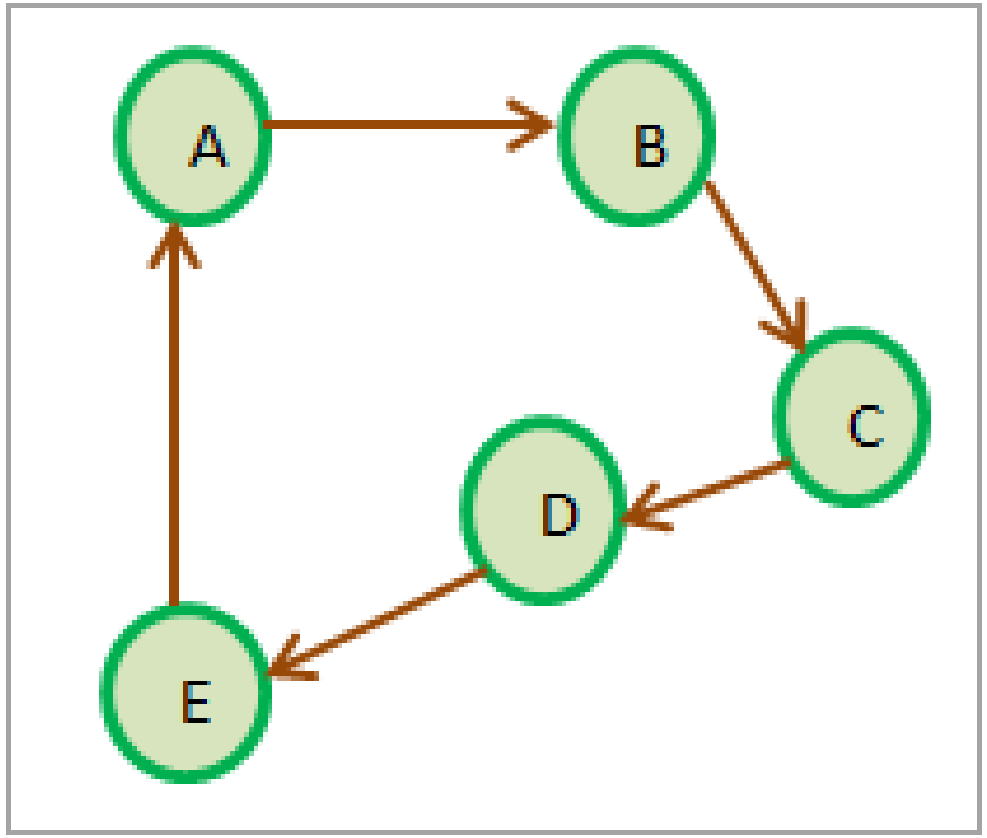
Undirected Graph

- A graph having five vertices {A,B,C,D,E}
- Edges given by {{AB},{AC},{AD},{BD}, {CE},{ED}}.
- As the edges do not show any directions, this graph is known as '**undirected graph**'.



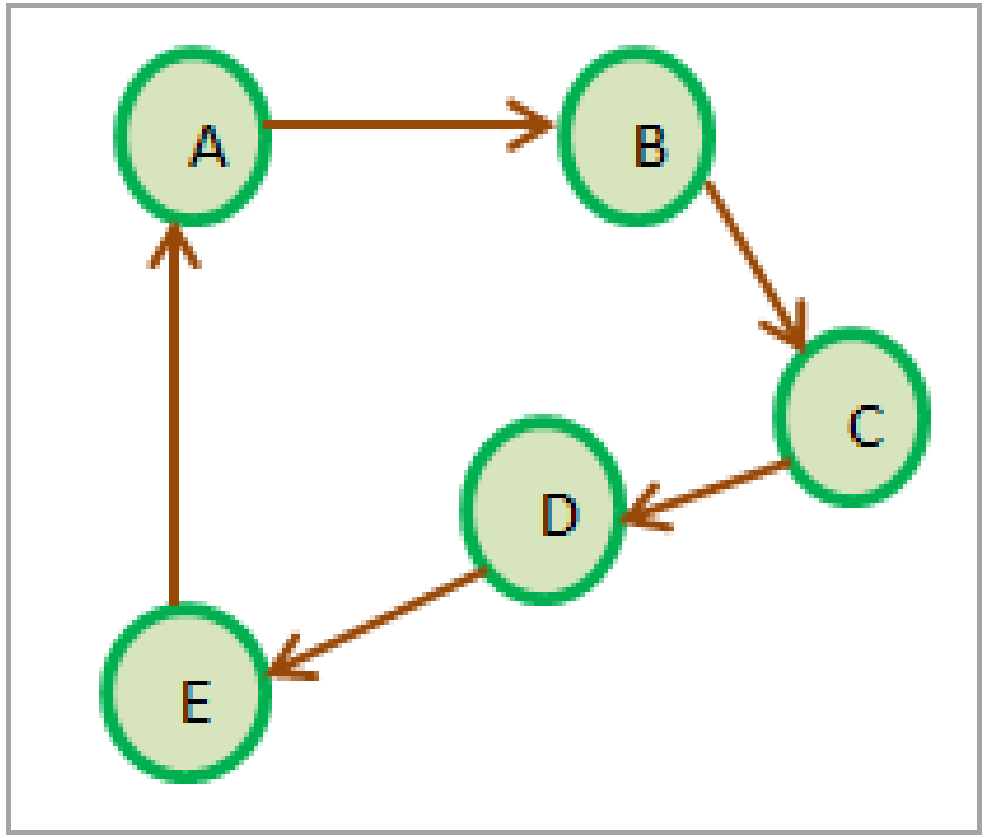
Directed Graph

- A directed graph or digraph is a graph data structure in which the edges have a specific direction.
- They originate from one vertex and culminate into another vertex.
- An edge from vertex A to vertex B is not the same as B to A.



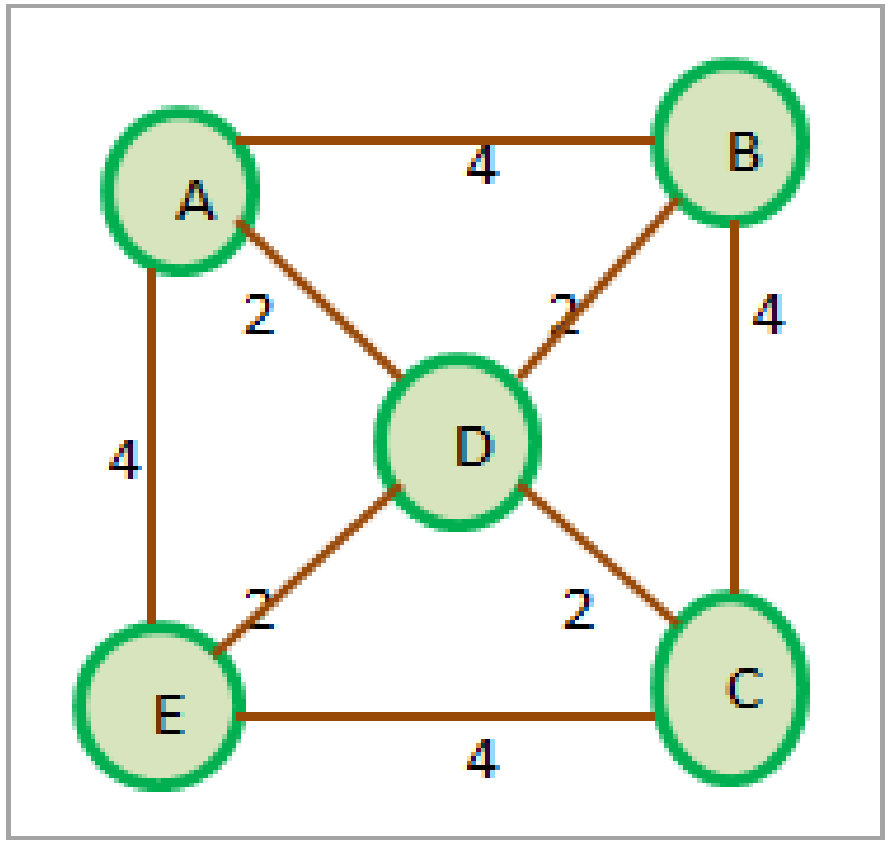
Directed Graph

- A directed graph is cyclic if there is at least one path that has its first and last vertex as same.
- A path A->B->C-> D->E->A forms a directed cycle or cyclic graph.



Weighted Graph

- A weight is associated with each edge of the graph.
- The weight normally indicates the distance between the two vertices.
- A weighted graph can be directed or undirected.



Graph Representation

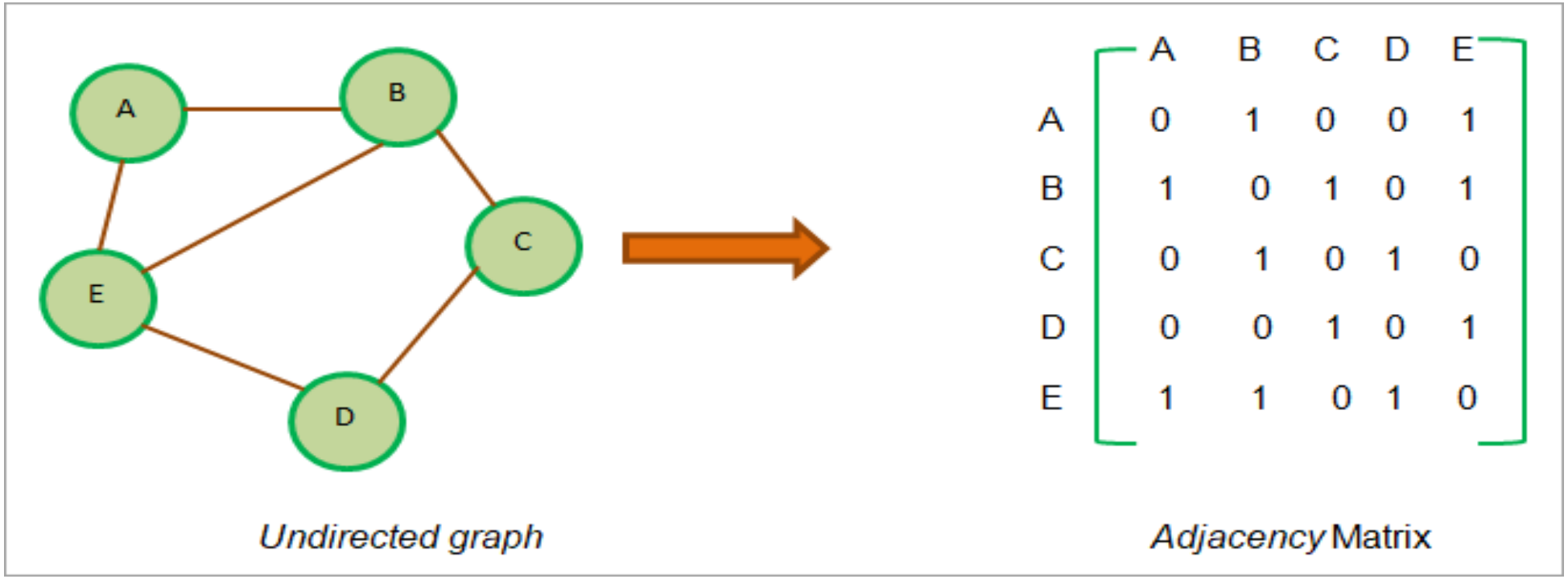
- Adjacency Matrix
- Adjacency List

Adjacency Matrix

- Adjacency Matrix is a linear representation of graphs.
- This matrix stores the mapping of vertices and edges of the graph.
- In the adjacency matrix, vertices of the graph represent rows and columns.
- This means if the graph has N vertices, then the adjacency matrix will have size NxN.
- If V is a set of vertices of the graph then intersection M_{ij} in the adjacency list = 1 means there is an edge existing between vertices i and j.

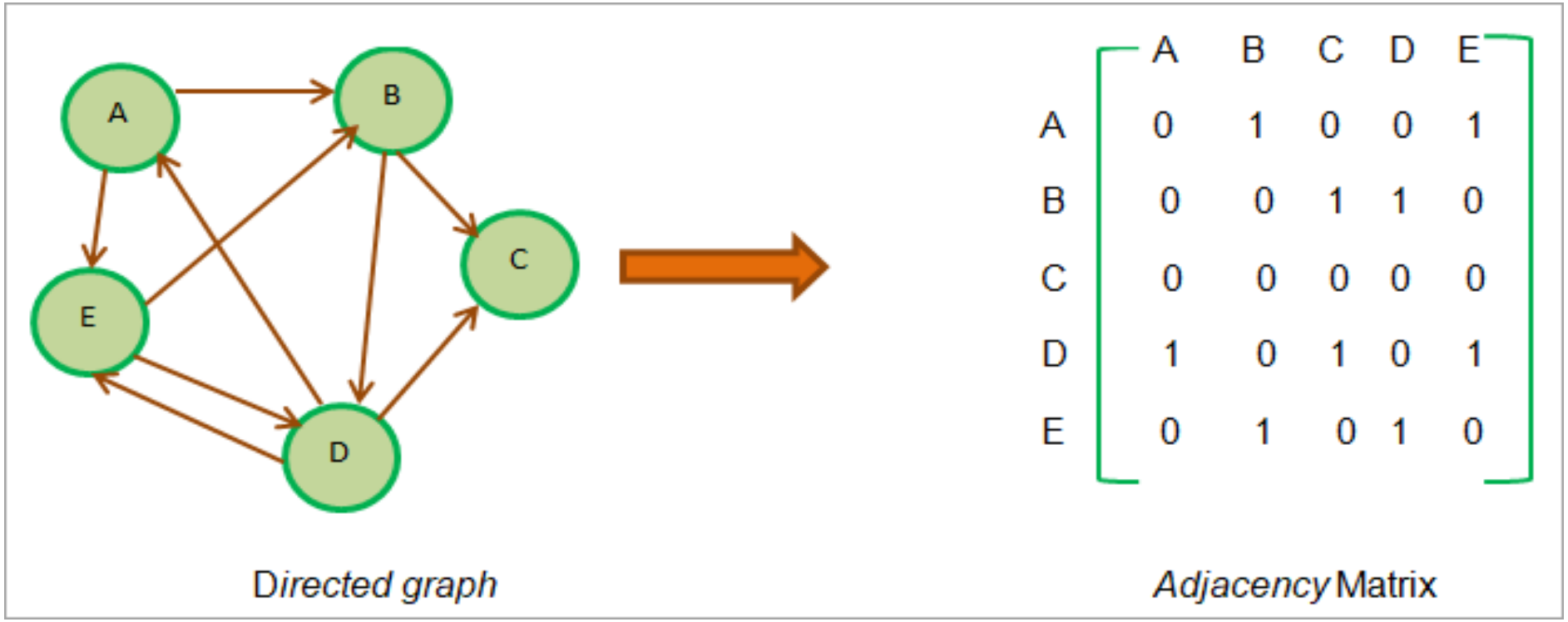
Adjacency Matrix

- Vertex A, the intersections AB and AE are set to 1 as there is an edge from A to B and A to E.



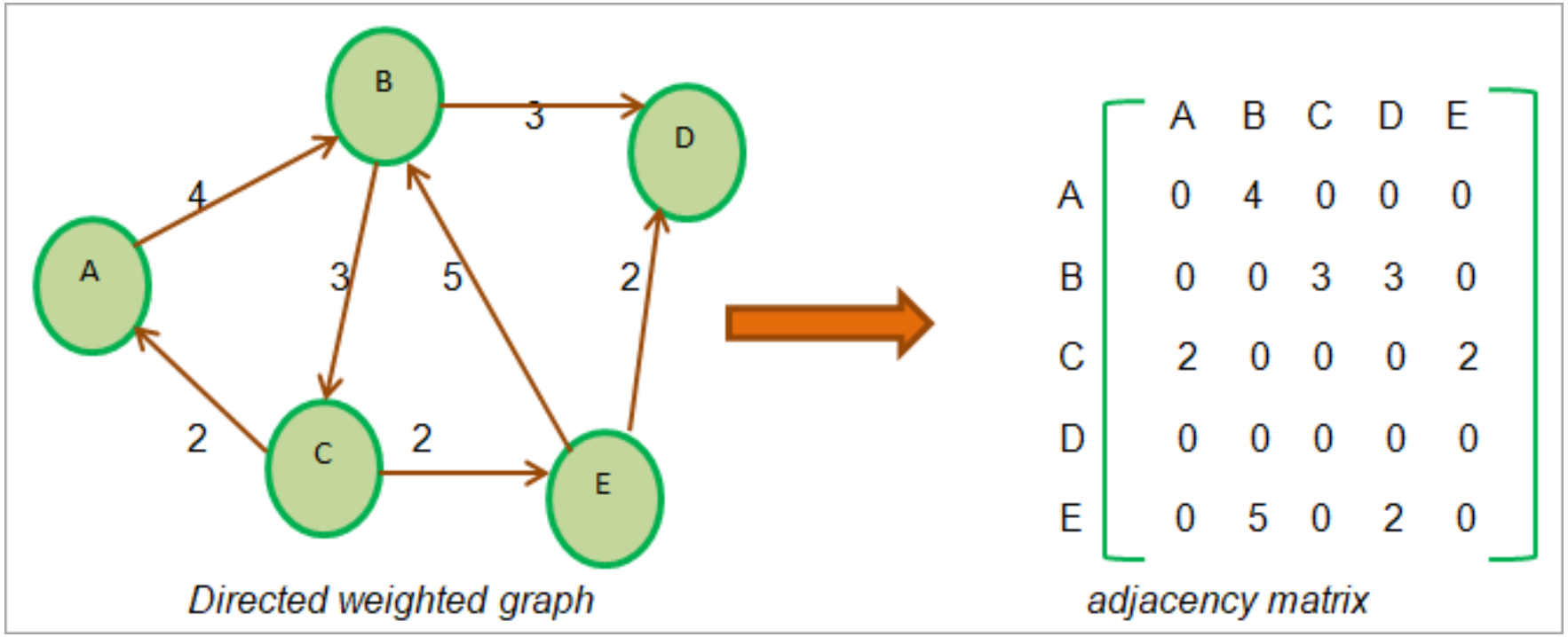
Adjacency Matrix

- An edge from A to B. So intersection AB is set to 1 but intersection BA is set to 0. This is because there is no edge directed from B to A.



Adjacency Matrix

- In weighted graph, this weight is specified whenever there is an edge from one vertex to another instead of ‘1’.

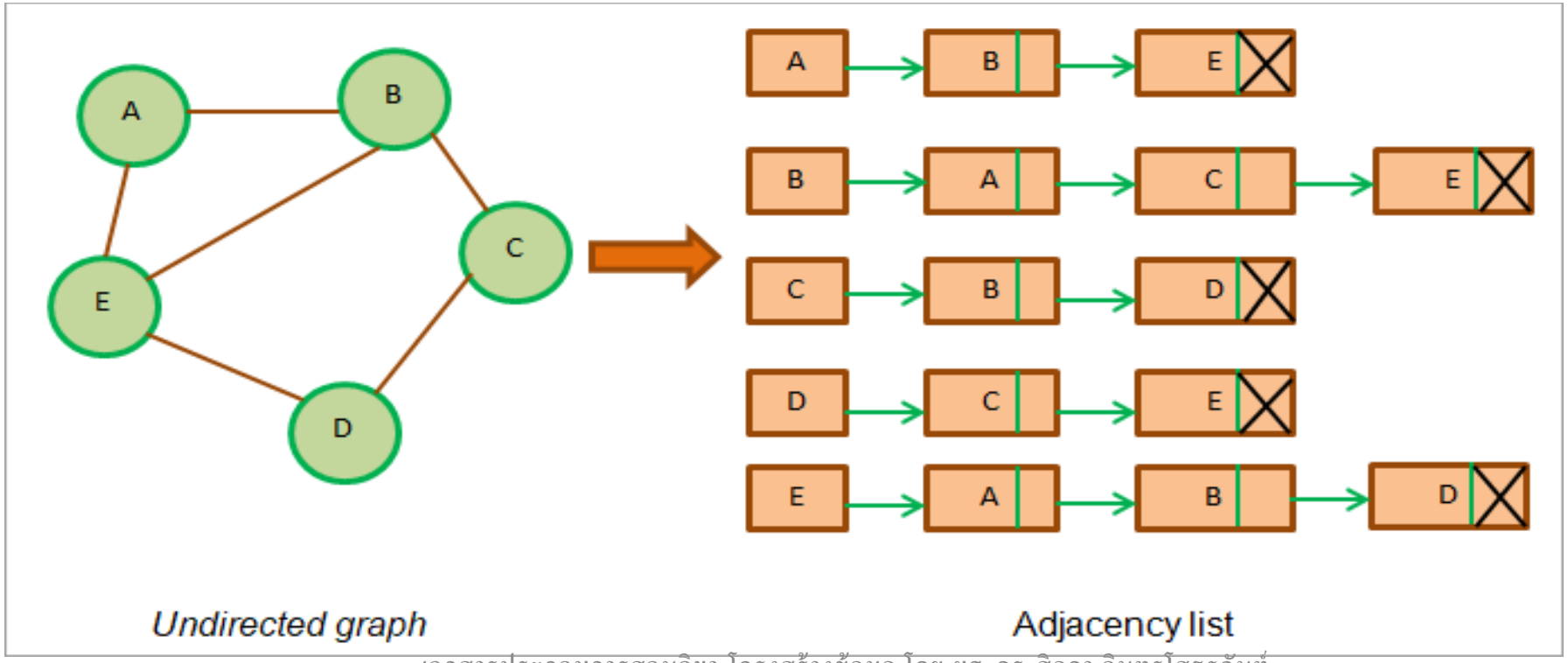


Adjacency List

- This linked representation is known as the adjacency list.
- An adjacency list is nothing but a linked list and each node in the list represents a vertex.
- The presence of an edge between two vertices is denoted by a pointer from the first vertex to the second.
- This adjacency list is maintained for each vertex in the graph.
- When we have traversed all the adjacent nodes for a particular node, we store NULL in the next pointer field of the last node of the adjacency list.

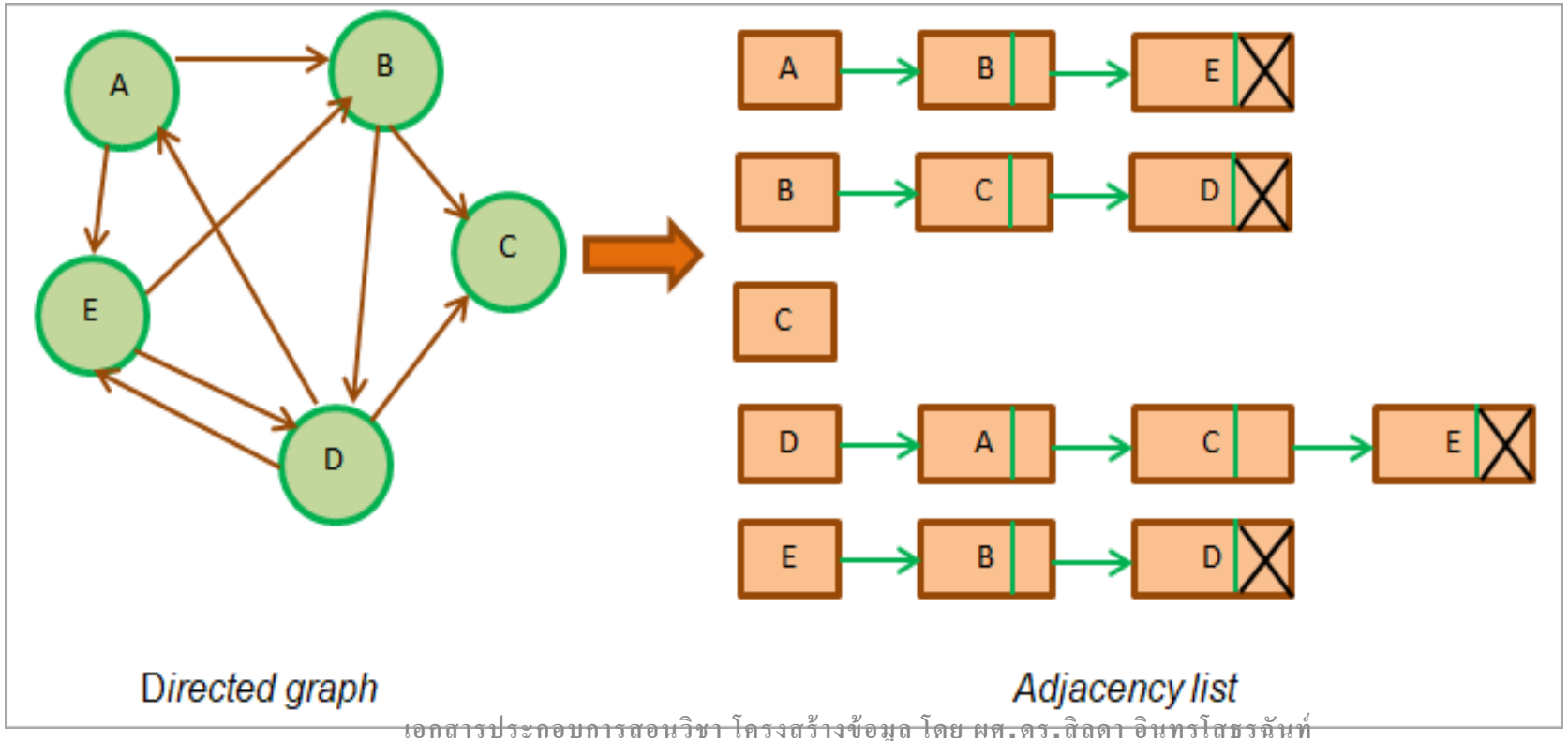
Adjacency List

- the total lengths of adjacency lists are usually twice the number of edges.
- The total number of edges is 6 and the total of the length of all the adjacency list is 12.



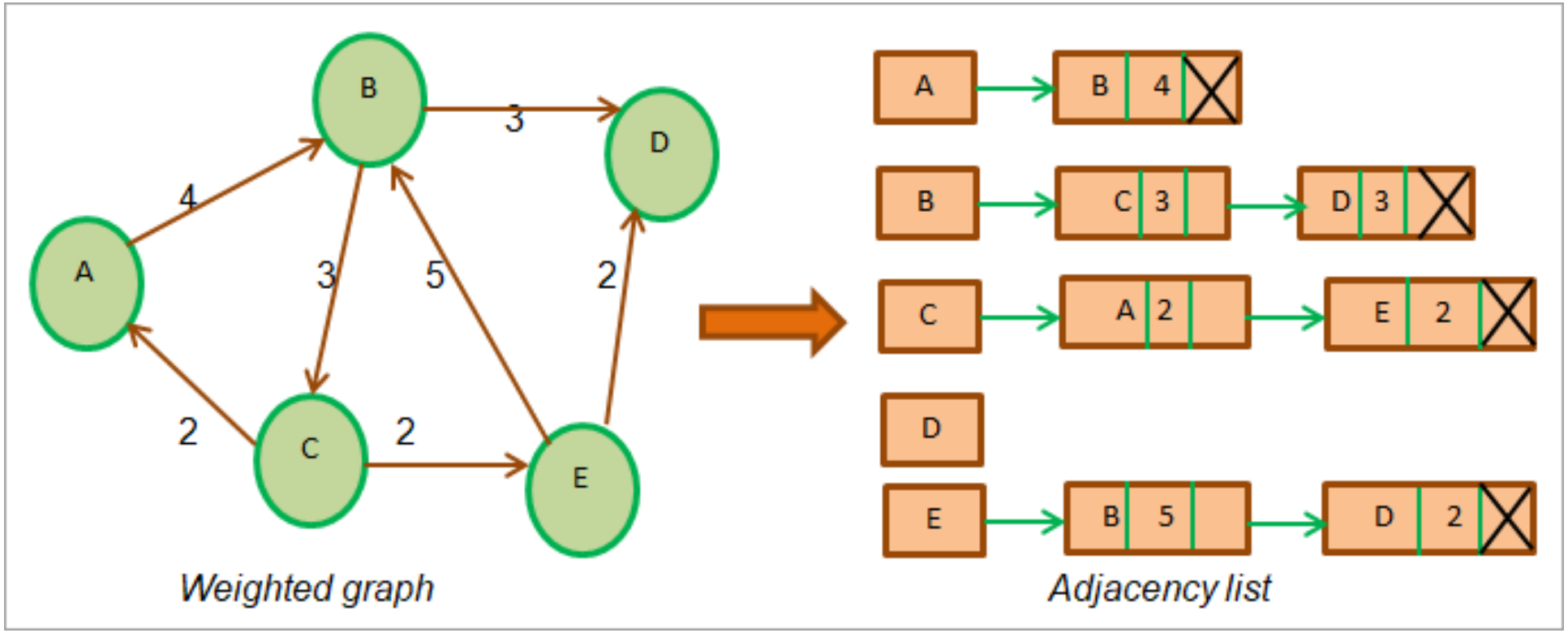
Adjacency List

- In the directed graph, the total length of the adjacency lists of the graph is equal to the number of edges in the graph.



Adjacency List

- In weighted graph, the weight of the edge is added a new field to each list node.



Graph implementation in Java

- In weighted graph, the weight of the edge is added a new field to each list node.

```
//class to store edges of the weighted graph
class Edge {
    int src, dest, weight;
    Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
}
```

Graph implementation in Java

```
import java.util.*;
// Graph class
class Graph {
    // node of adjacency list
    static class Node {
        int value, weight;
        Node(int value, int weight)  {
            this.value = value;
            this.weight = weight;
        }
    };

// define adjacency list
 List<List<Node>> adj_list = new ArrayList<>();
```

Graph implementation in Java

```
 //Graph Constructor
public Graph(List<Edge> edges)
{
    // adjacency list memory allocation
    for (int i = 0; i < edges.size(); i++)
        adj_list.add(i, new ArrayList<>());

    // add edges to the graph
    for (Edge e : edges)
    {
        // allocate new node in adjacency List from src to dest
        adj_list.get(e.src).add(new Node(e.dest, e.weight));
    }
}
```

Graph implementation in Java

```
// print adjacency list for the graph
public static void printGraph(Graph graph)  {
    int src_vertex = 0;
    int list_size = graph.adj_list.size();
    System.out.println("The contents of the graph:");
    while (src_vertex < list_size) {
    //traverse through the adjacency list and print the edges
      for (Node edge : graph.adj_list.get(src_vertex)) {
          System.out.print("Vertex:" + src_vertex + " ==> " +
                  edge.value + " (" + edge.weight + ")\t");
      }
      System.out.println();
      src_vertex++;
    }
  }
}
```

Graph implementation in Java

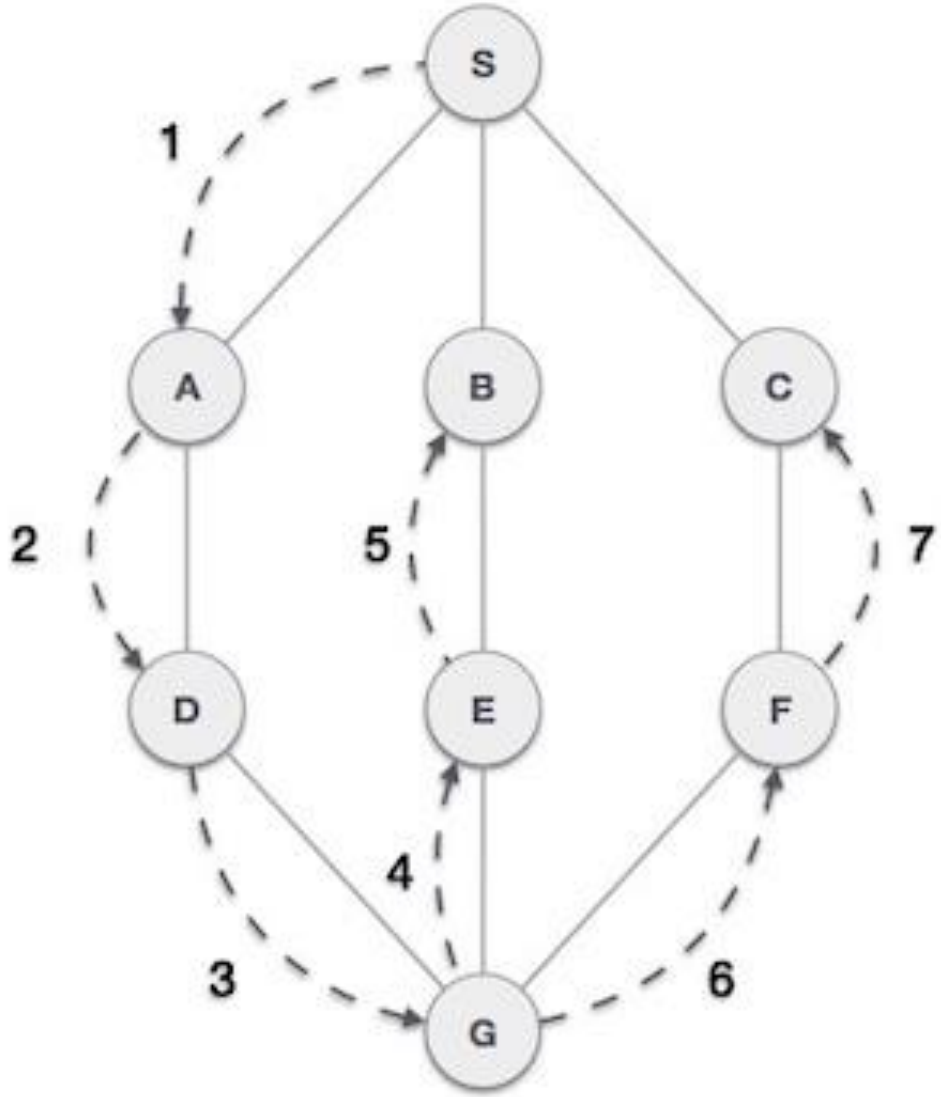
```
import java.util.*;
class Main{
  public static void main (String[] args) {
    // define edges of the graph
    List<Edge> edges = Arrays.asList(new Edge(0, 1, 2),
      new Edge(0, 2, 4), new Edge(1, 2, 4),new Edge(2, 0, 5),
      new Edge(2, 1, 4), new Edge(3, 2, 3), new Edge(4, 5, 1),
      new Edge(5, 4, 3));
      // call graph class Constructor to construct a graph
       Graph graph = new Graph(edges);
      // print the graph as an adjacency list
      Graph.printGraph(graph);
    }
}
```

Graph Traversal

- Depth-first traversal
- Breadth-first traversal

Depth-first traversal

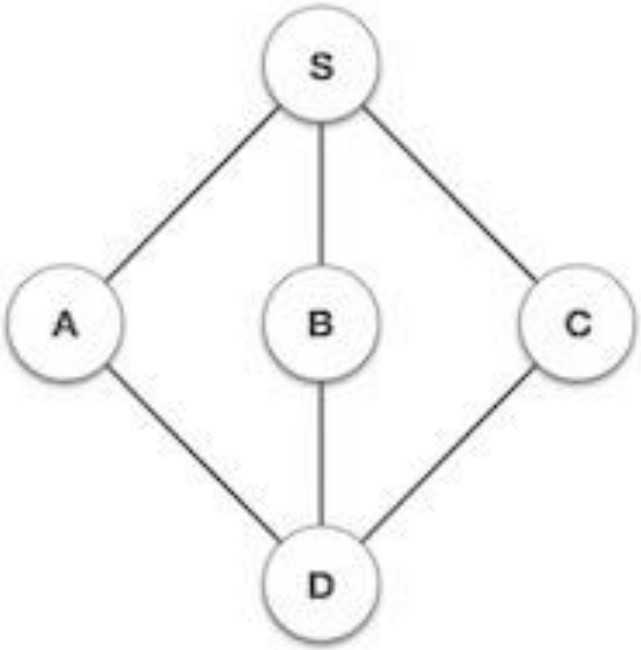
- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



Depth-first traversal

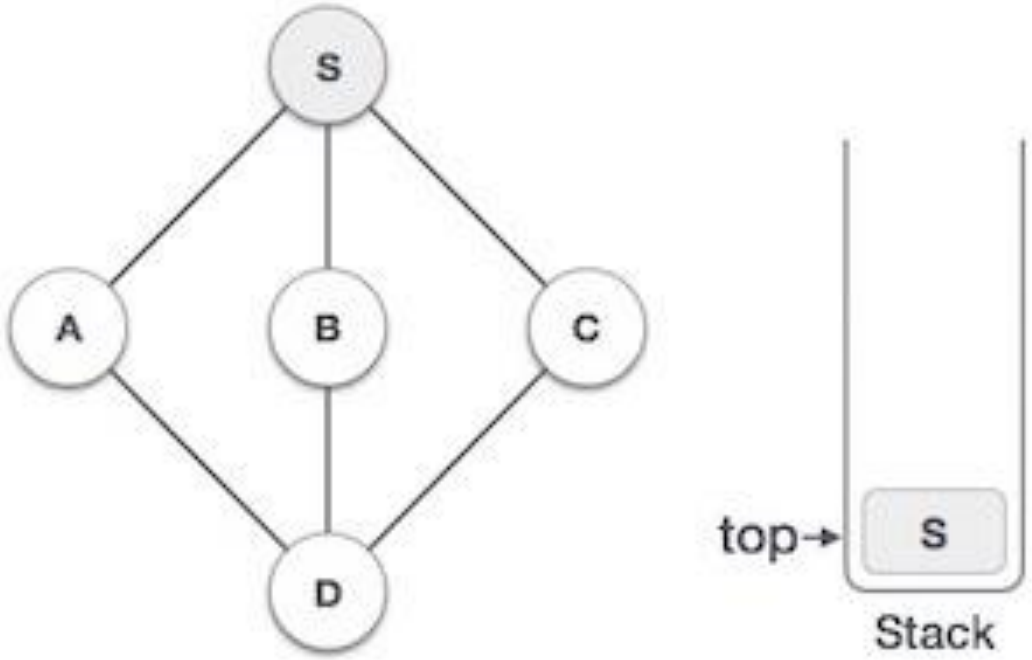
- As in the example, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Depth-first traversal



- Initialize the stack.

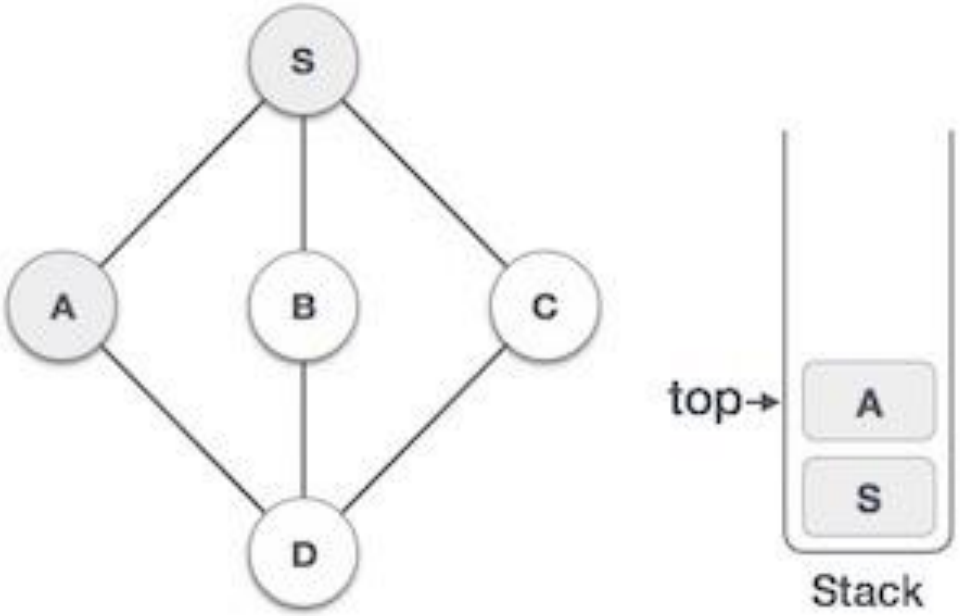
Depth-first traversal



- Mark **S** as visited and put it onto the stack.
- Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

Start : S

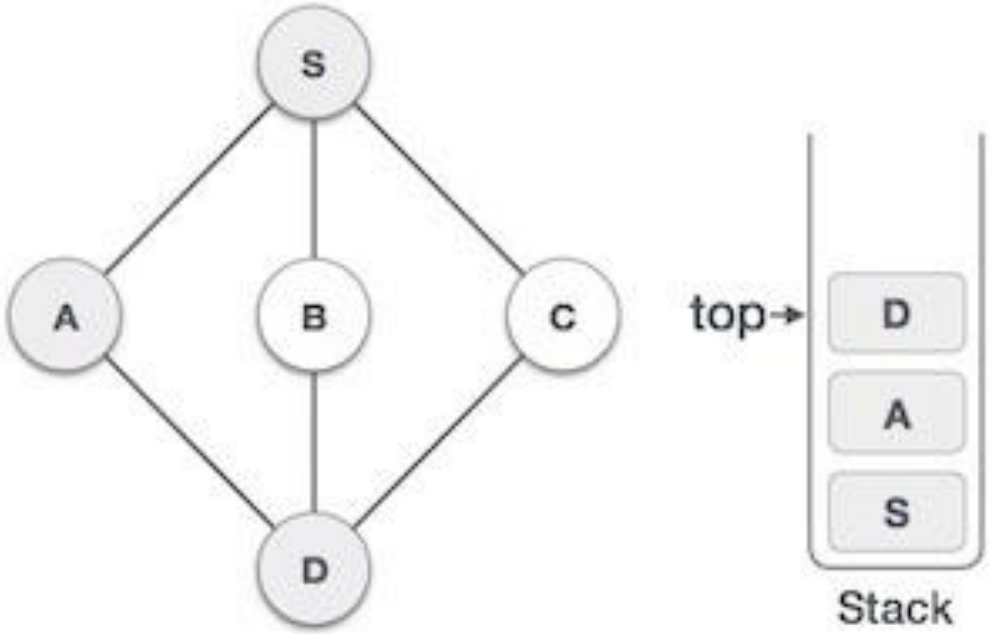
Depth-first traversal



- Mark **A** as visited and put it onto the stack.
- Explore any unvisited adjacent node from A.
- Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

S -> A

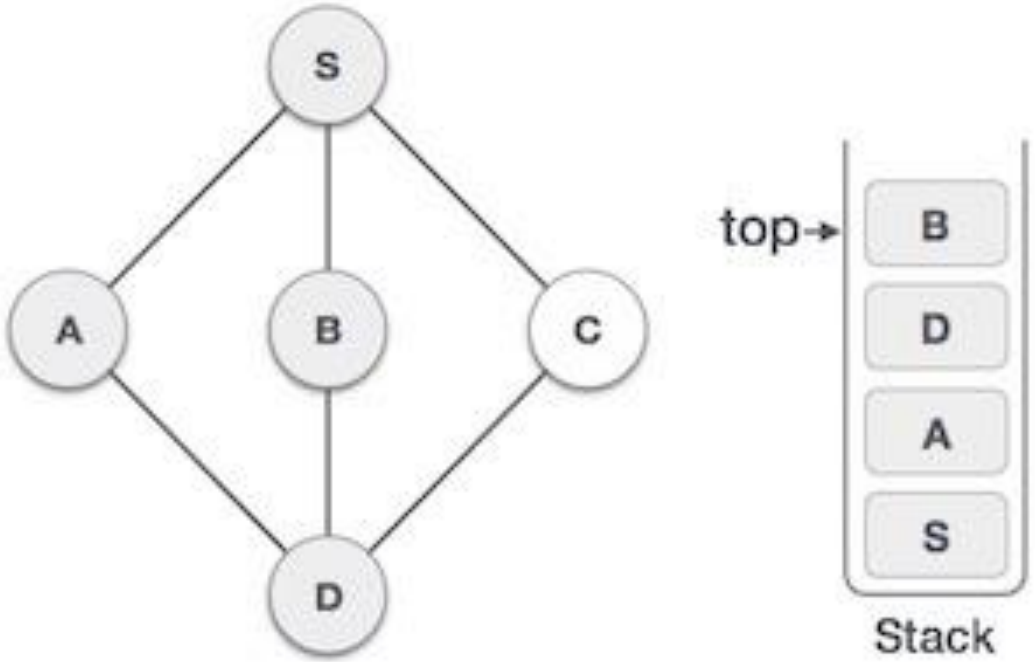
Depth-first traversal



- Visit **D** and mark it as visited and put onto the stack.
- Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

S -> A -> D

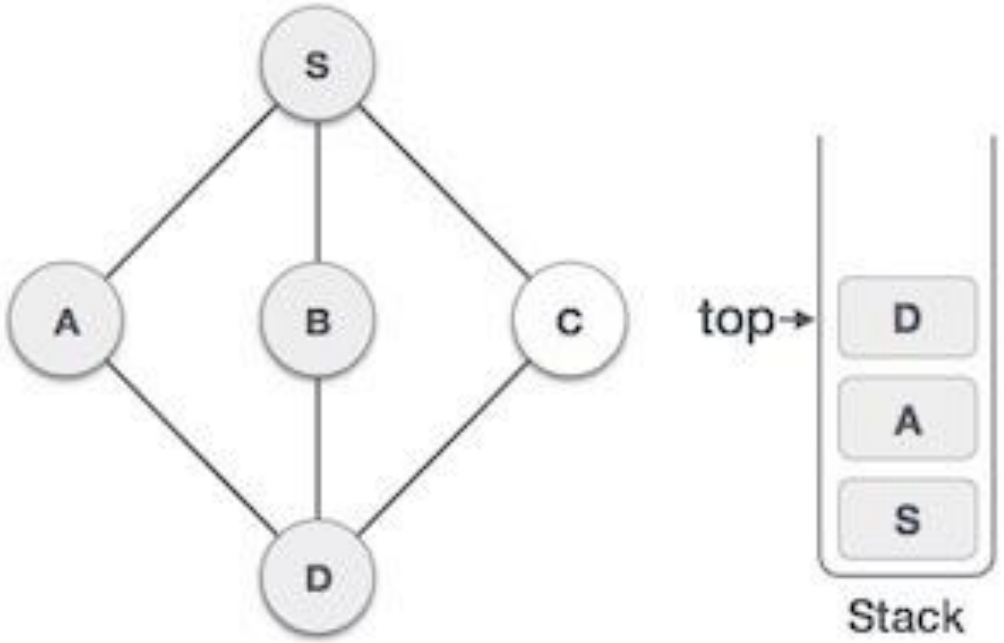
Depth-first traversal



- We choose **B**, mark it as visited and put onto the stack.
- Here **B** does not have any unvisited adjacent node.
- So, we pop **B** from the stack.

S -> A -> D -> B

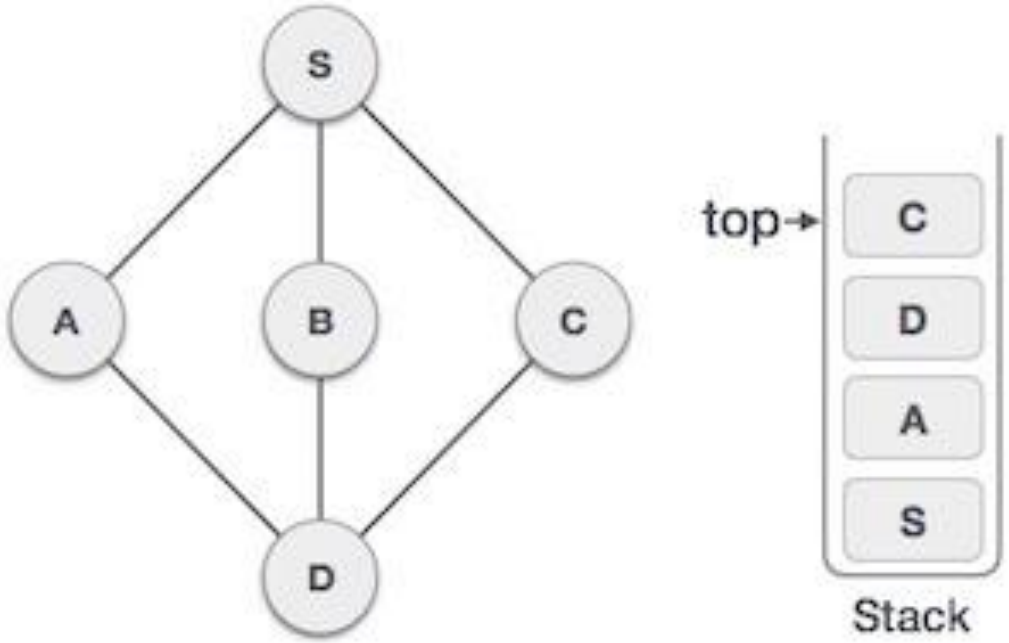
Depth-first traversal



- We check the stack top for return to the previous node and check if it has any unvisited nodes.
- Here, we find **D** to be on the top of the stack.

S -> A -> D -> B

Depth-first traversal

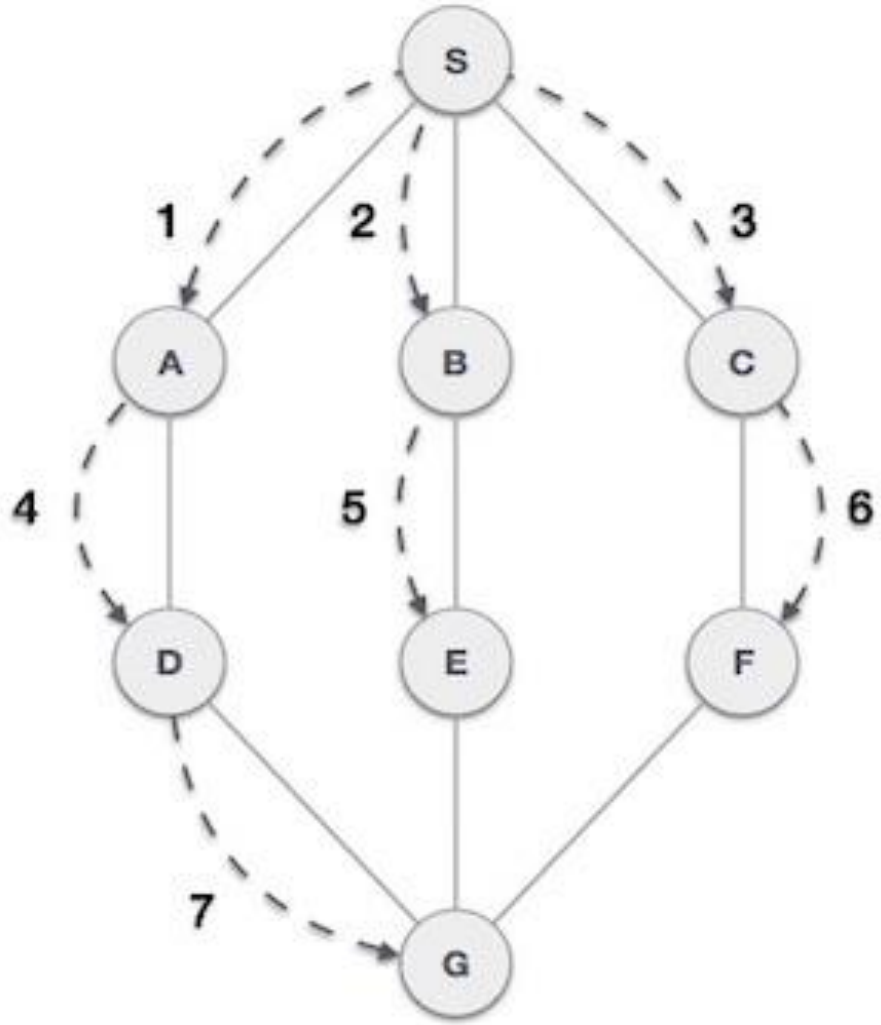


S -> A -> D -> B -> C

- Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.
- As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node.

Breadth-first traversal

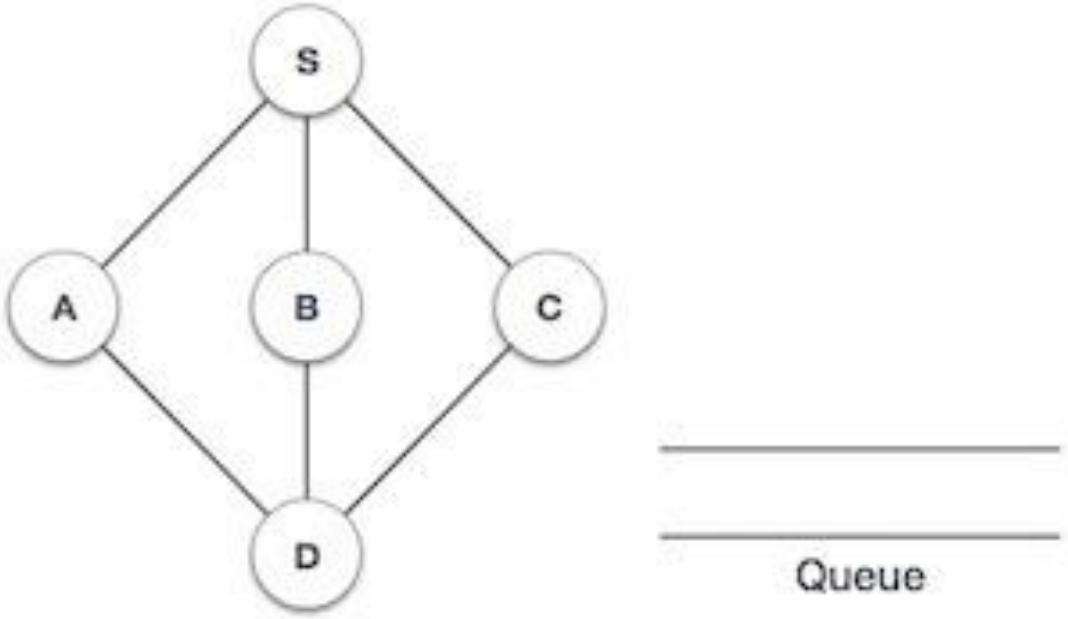
- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



Breadth-first traversal

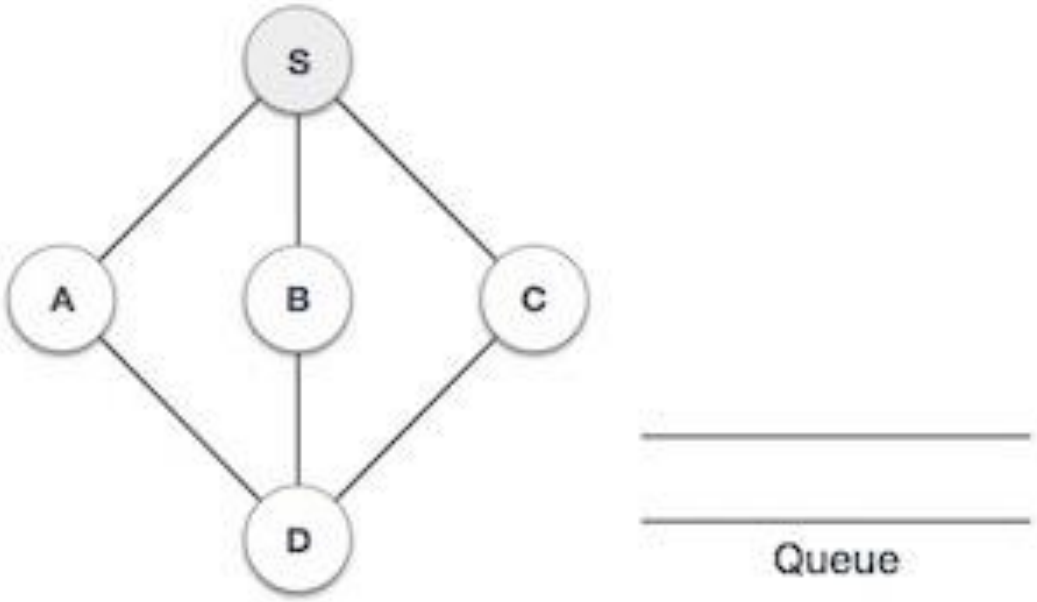
- As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Breadth-first traversal



- Initialize the queue.

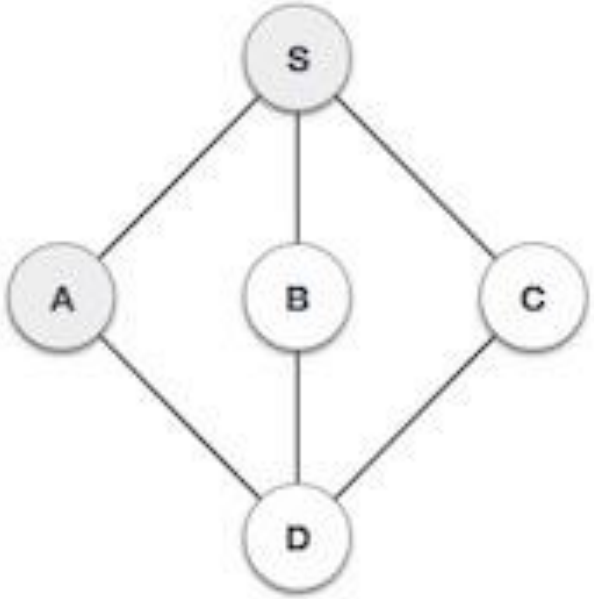
Breadth-first traversal

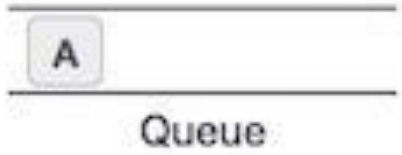


- We start from visiting **S** (starting node), and mark it as visited.

Start : S

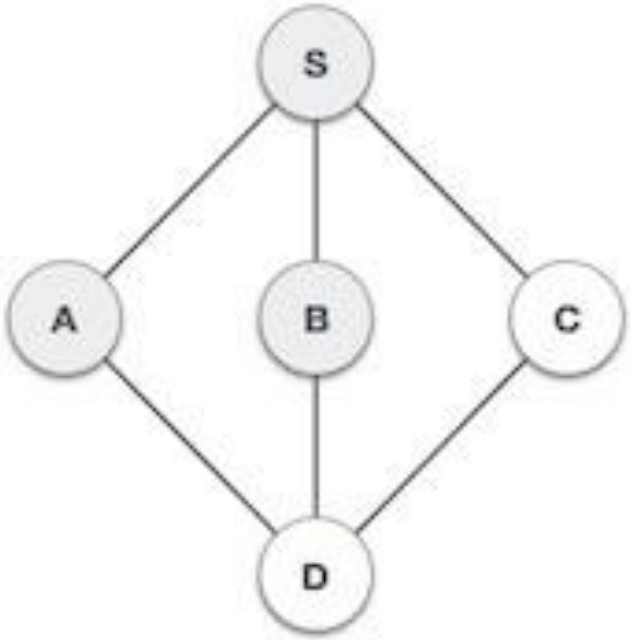
Breadth-first traversal

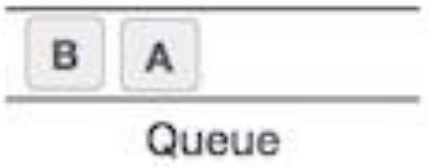




- We then see an unvisited adjacent node from **S**.
- In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

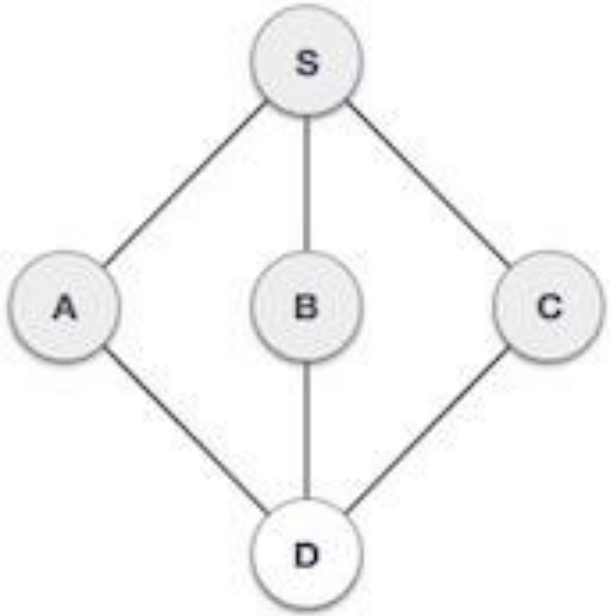
Breadth-first traversal

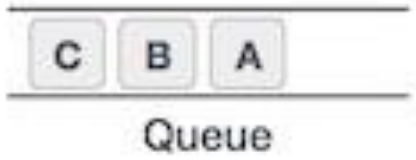




- Next, the unvisited adjacent node from **S** is **B**.
- We mark it as visited and enqueue it.

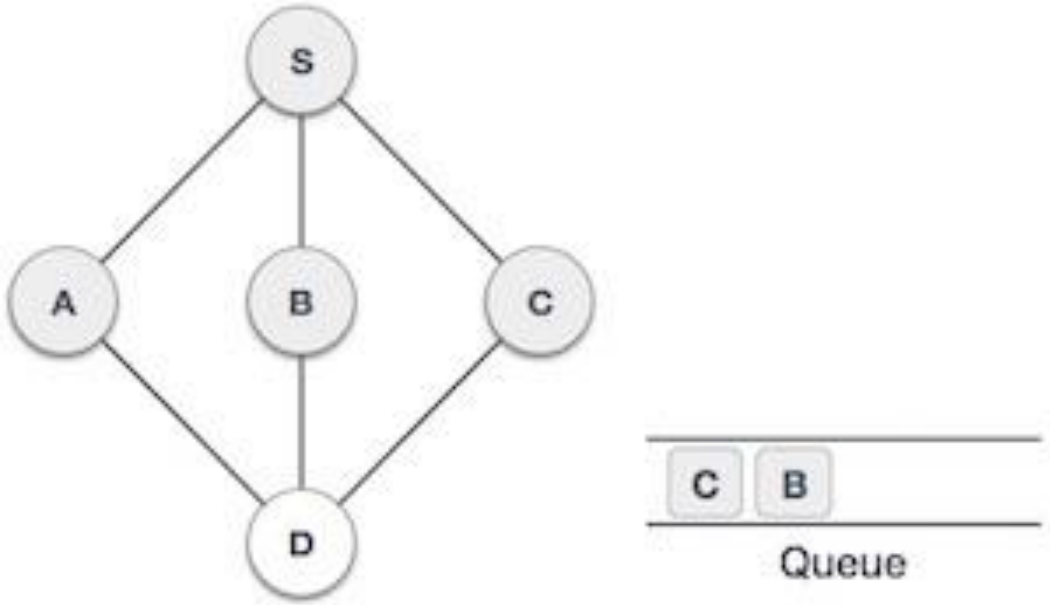
Breadth-first traversal





- Next, the unvisited adjacent node from **S** is **C**.
- We mark it as visited and enqueue it.

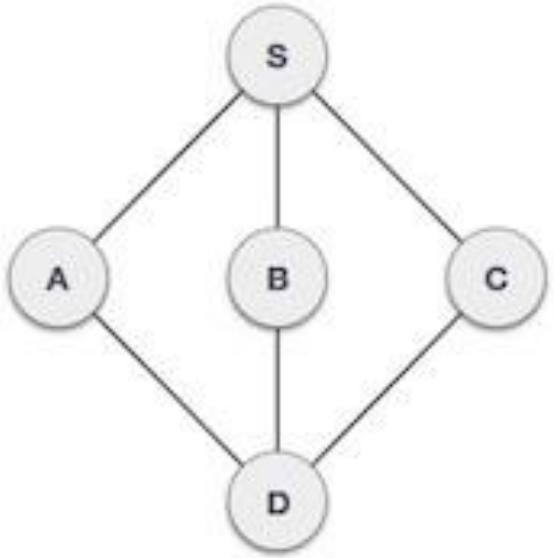
Breadth-first traversal

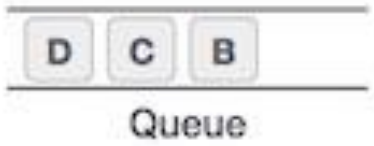


- Now, **S** is left with no unvisited adjacent nodes.
- So, we dequeue and find **A**.

S -> A

Breadth-first traversal





- From **A** we have **D** as unvisited adjacent node.
- We mark it as visited and enqueue it.
- At this stage, we are left with no unmarked (unvisited) nodes.
- Dequeue in order to get all unvisited nodes. When the queue gets emptied, the program is over.

S -> A -> B -> C -> D