

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CS 201 – DATA STRUCTURES LAB

Instructors: Faizan Yousuf & Abdul Aziz

LAB SESSION # 12

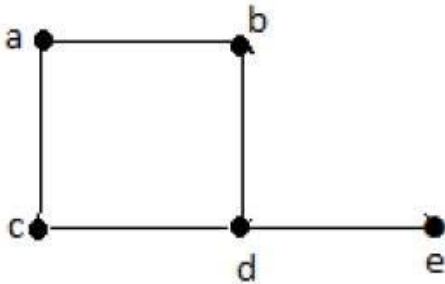
Outline

- Introduction to Graph
- Graph Representation
- DFS
- BFS
- Topological Sort Algorithm
- Exercise

What is a Graph?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

Applications of Graph Theory

Graph theory has its applications in diverse fields of engineering –

- **Electrical Engineering** – The concepts of graph theory is used extensively in designing circuit connections. The types or organization of connections are named as topologies. Some examples for topologies are star, bridge, series, and parallel topologies.
- **Computer Science** – Graph theory is used for the study of algorithms. For example,
 - Kruskal's Algorithm
 - Prim's Algorithm
 - Dijkstra's Algorithm
- **Computer Network** – The relationships among interconnected computers in the network follows the principles of graph theory.
- **Science** – The molecular structure and chemical structure of a substance, the DNA structure of an organism, etc., are represented by graphs.
- **Linguistics** – The parsing tree of a language and grammar of a language uses graphs.

- **General** – Routes between the cities can be represented using graphs. Depicting hierarchical ordered information such as family tree can be used as a special type of graph called tree.

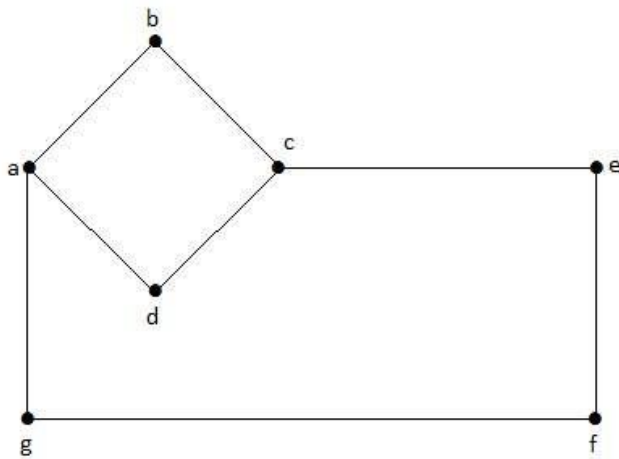
Types of Graphs

There are various types of graphs depending upon the number of vertices, number of edges, interconnectivity, and their overall structure. We will discuss only a two important types of graphs in this lab.

Non-Directed Graph

A non-directed graph contains edges but the edges are not directed ones.

Example

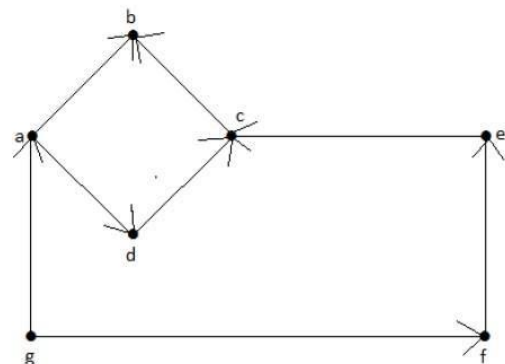


In this graph, 'a', 'b', 'c', 'd', 'e', 'f', 'g' are the vertices, and 'ab', 'bc', 'cd', 'da', 'ag', 'gf', 'ef' are the edges of the graph. Since it is a non-directed graph, the edges 'ab' and 'ba' are same. Similarly, other edges also considered in the same way.

Directed Graph

In a directed graph, each edge has a direction.

Example



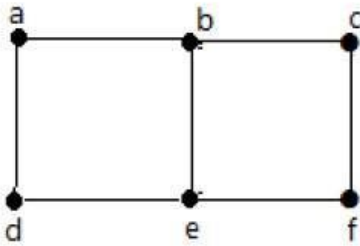
In the above graph, we have seven vertices 'a', 'b', 'c', 'd', 'e', 'f', and 'g', and eight edges 'ab', 'cb', 'dc', 'ad', 'ec', 'fe', 'gf', and 'ga'. As it is a directed graph, each edge bears an arrow mark that shows its direction. Note that in a directed graph, 'ab' is different from 'ba'.

Adjacency

Here are the norms of adjacency –

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

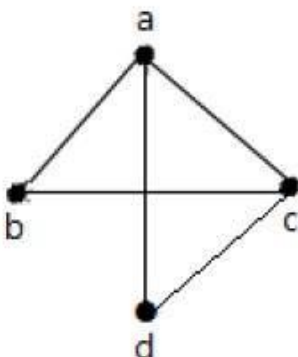
Example 1



In the above graph –

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

Example 2



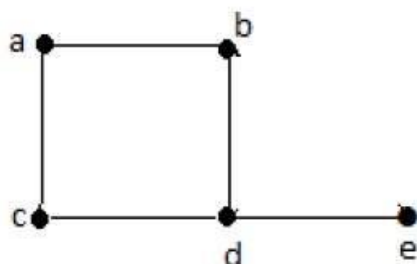
In the above graph –

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

Degree Sequence of a Graph

If the degrees of all vertices in a graph are arranged in descending or ascending order, then the sequence obtained is known as the degree sequence of the graph.

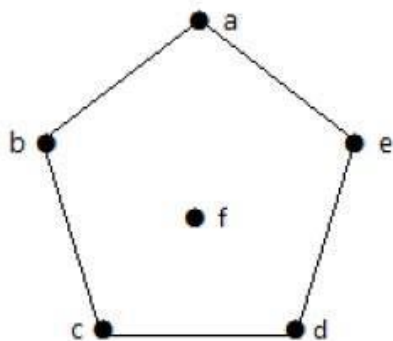
Example 1



Vertex	A	b	c	d	e
Connecting to	b,c	a,d	a,d	c,b,e	d
Degree	2	2	2	3	1

In the above graph, for the vertices {d, a, b, c, e}, the degree sequence is {3, 2, 2, 2, 1}.

Example 2



Vertex	A	b	c	d	e	f
Connecting to	b,e	a,c	b,d	c,e	a,d	-
Degree	2	2	2	2	2	0

In the above graph, for the vertices $\{a, b, c, d, e, f\}$, the degree sequence is $\{2, 2, 2, 2, 2, 0\}$.

Representation of Graph:

Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

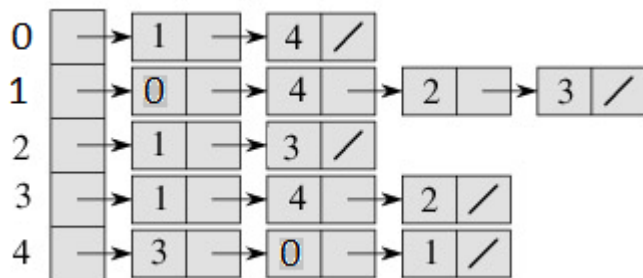
Adjacency Matrix Representation of the above graph

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex ' u ' to vertex ' v ' are efficient and can be done $O(1)$.

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

Adjacency List:

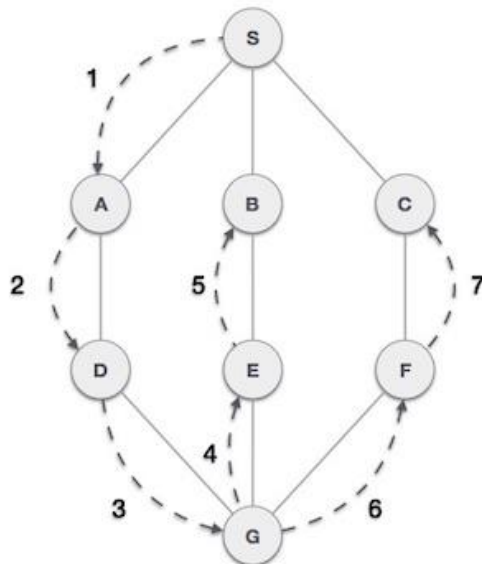
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be `array[]`. An entry `array[i]` represents the linked list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph

Depth First Traversal

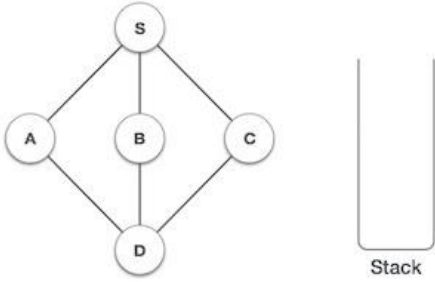
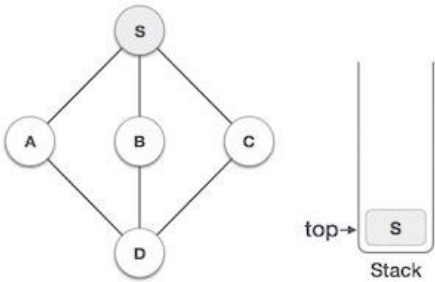
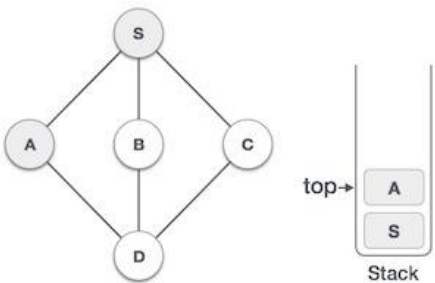
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

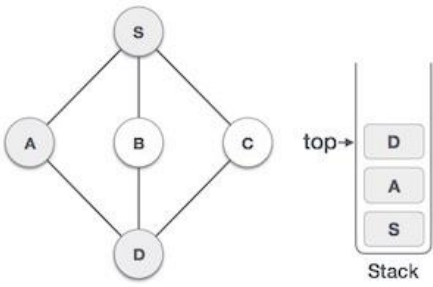
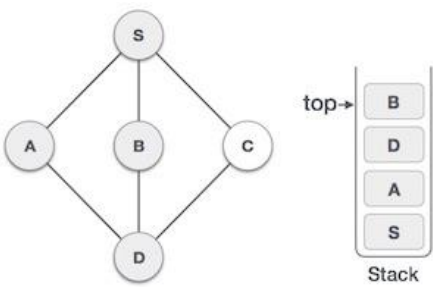
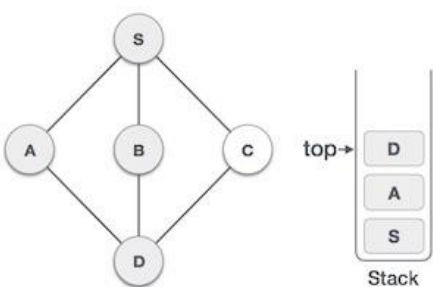
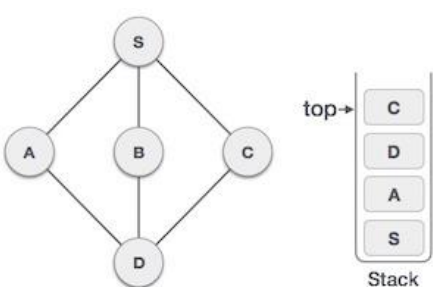


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to G. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

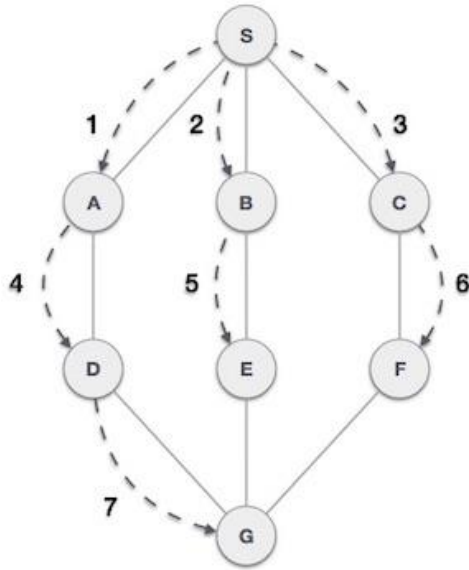
Step	Traversal	Description
1.		Initialize the stack.
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.

4.		<p>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.</p>
5.		<p>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.</p>
6.		<p>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.</p>
7.		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

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

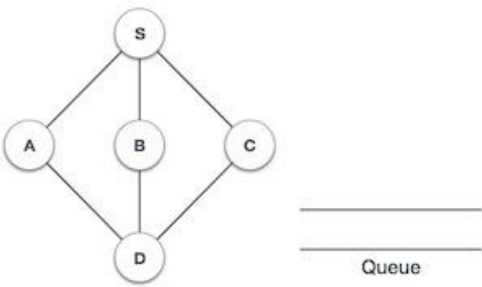
Breadth First Traversal

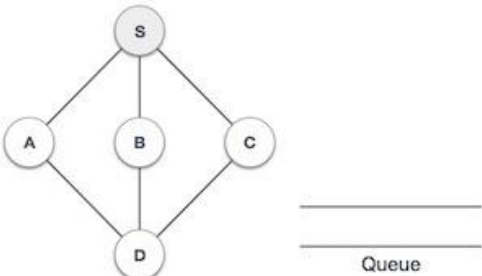
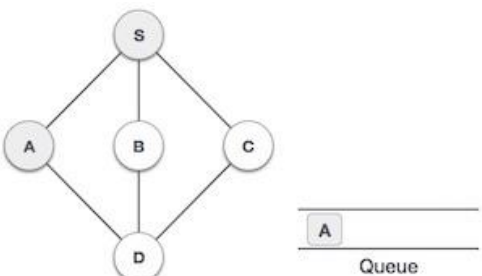
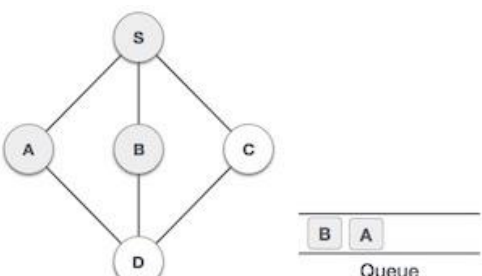
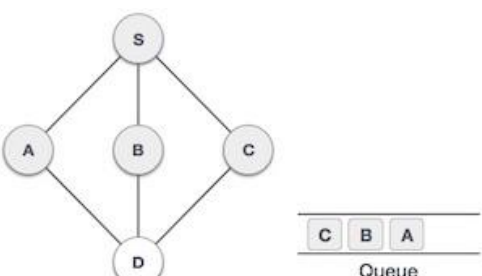
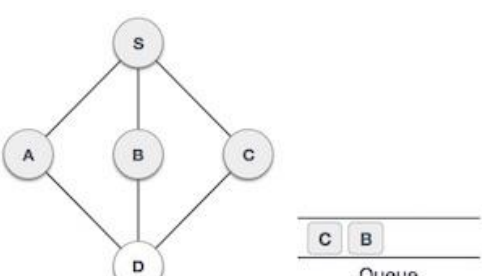
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

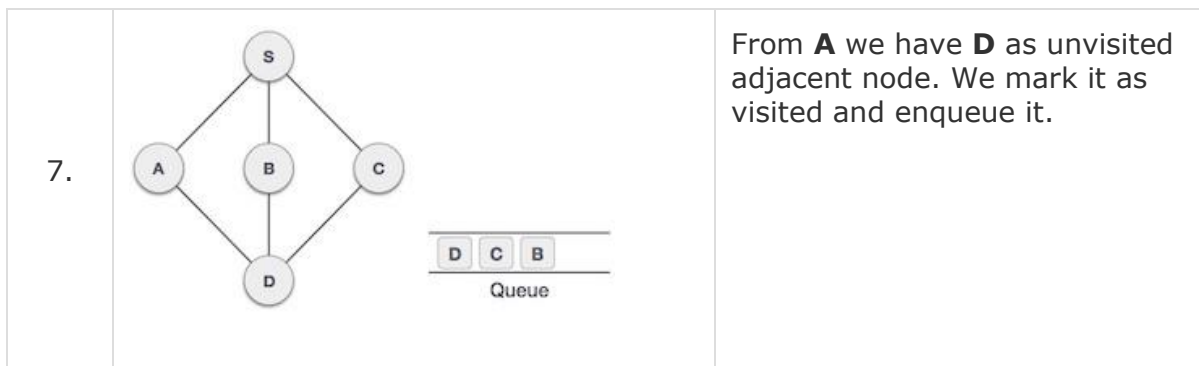


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **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.

Step	Traversal	Description
1.		Initialize the queue.

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

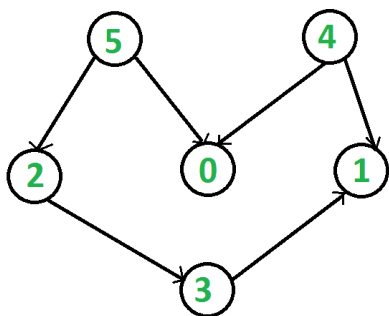


At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



Topological Sorting vs Depth First Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a topological sorting

Algorithm to find Topological Sorting:

```
topological_sort(N, adj[N][N])
    T = []
    visited = []
    in_degree = []
    for i = 0 to N
        in_degree[i] = visited[i] = 0

    for i = 0 to N
        for j = 0 to N
            if adj[i][j] is TRUE
                in_degree[j] = in_degree[j] + 1

    for i = 0 to N
        if in_degree[i] is 0
            enqueue(Queue, i)
            visited[i] = TRUE

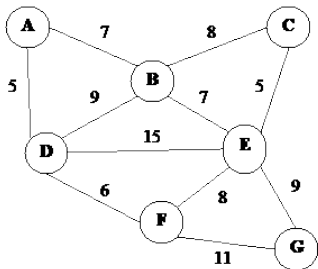
    while Queue is not Empty
        vertex = get_front(Queue)
        dequeue(Queue)
        T.append(vertex)
        for j = 0 to N
            if adj[vertex][j] is TRUE and visited[j] is FALSE
                in_degree[j] = in_degree[j] - 1
                if in_degree[j] is 0
                    enqueue(Queue, j)
                    visited[j] = TRUE

    return T
```

EXERCISE:

Question 1:

Write a code to construct following Graph using Adjacency List and Adjacency Matrix



Question 2:

Implement the following graph traversal algorithms on above Graph:

1. Depth first traversal.
2. Breadth first Traversal.

Question 3:

Implement the Topological Sort algorithm on above Graph.