DATA STRUCTURES

LECTURE-15

GRAPH

Dr. Sumitra Kisan

- Graph is a non-linear data structure consisting of vertices and edges.
- The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.
- Graph is composed of a set of vertices(**V**) and a set of edges(**E**).
- The graph is denoted by **G(V, E).**

Components of a Graph:

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

Operations on Graphs:

- Insertion of Nodes/Edges in the graph – Insert a node into the graph.
- Deletion of Nodes/Edges in the graph – Delete a node from the graph.
- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph**.**
- Shortest Paths : From a source to a destination, a source to all other nodes and between all pairs.
- Minimum Spanning Tee : In a weighted, connected undirected graph, finding the minimum weight edges to connect all.

- **Degree of a Vertex**

The Degree of a Vertex in a graph is the **number of edges incident** to that vertex. In a directed graph, the degree is further categorized into the in-degree (number of incoming edges) and out-degree (number of outgoing edges) of the vertex.

- **Path**

A Path in a graph is a **sequence of vertices** where each adjacent pair is connected by an edge. Paths can be of varying lengths and may or may not visit the same vertex more than once. The shortest path between two vertices is of particular interest in algorithms such as Dijkstra's algorithm for finding the shortest path in weighted graphs.

- **Source and Sink**

A vertex in a directed graph with indegree zero and outdegree one or more is referred to as a source. A vertex with outdegree zero and indegree one or more is referred to as a sink.

- **Tree**

A Tree is a connected graph with no cycles.

Representations of Graph

Here are the two most common ways to represent a graph :

1. Adjacency Matrix
2. Adjacency List

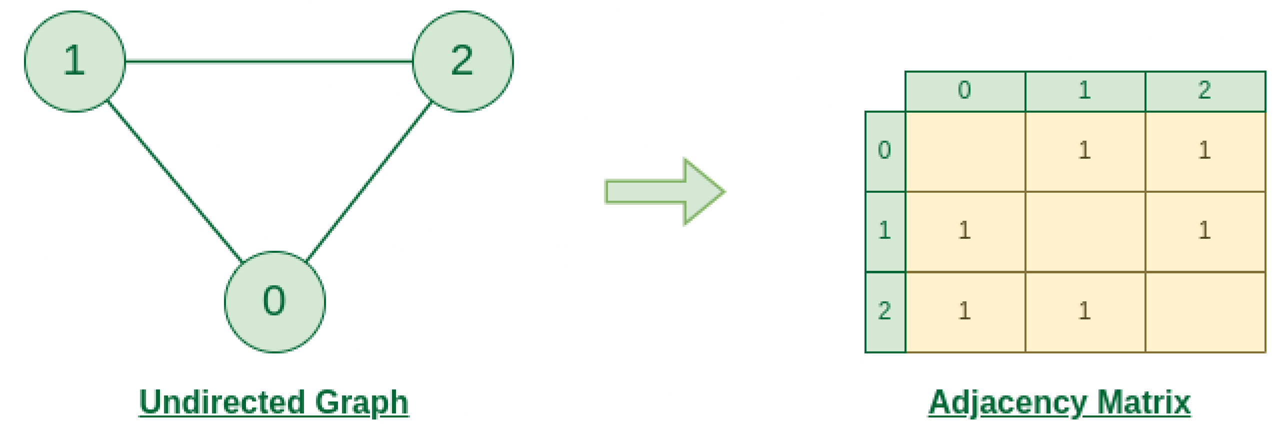
1. Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of boolean (0’s and 1’s)
Let’s assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- *If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.*
- *If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0***

Representation of Undirected Graph as Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat**[**destination])** because we can go either way.

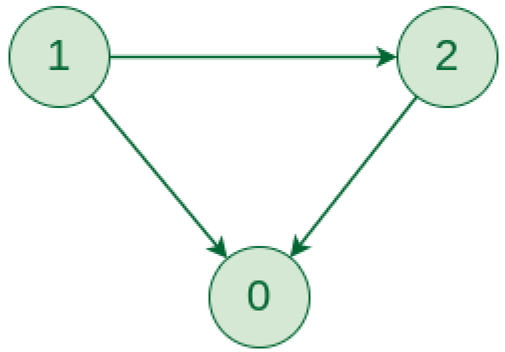


	0	1	2
0		1	1
1	1		1
2	1	1	

Graph Representation of Undirected graph to Adjacency Matrix

Representation of Directed Graph as Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



Directed Graph

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & & & \\ 1 & 1 & & 1 \\ 2 & 1 & & \end{array}$$

Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

Adjacency List

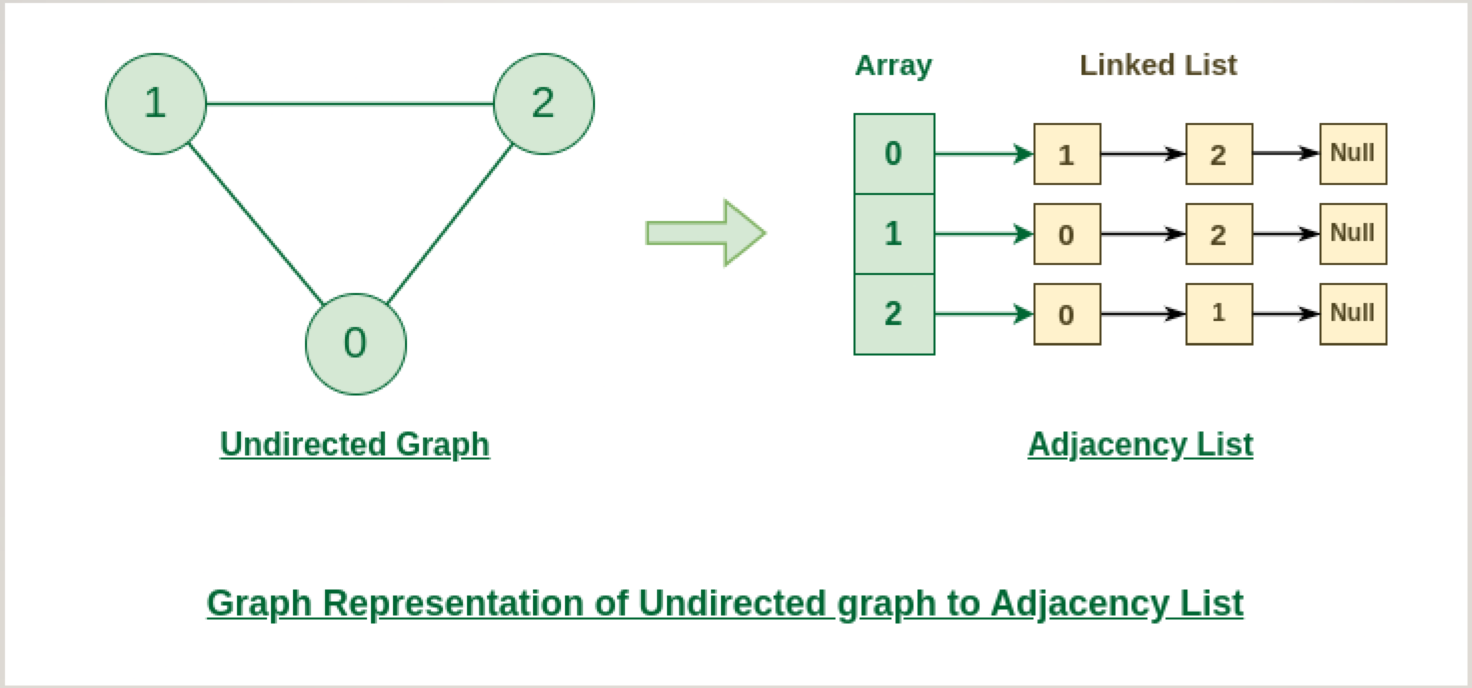
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of vertices (i.e, n). Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i.

Let’s assume there are n vertices in the graph So, create an array of list of size n as adjList[n].

- adjList[0] will have all the nodes which are connected (neighbour) to vertex 0.
- adjList[1] will have all the nodes which are connected (neighbour) to vertex 1 and so on.

Representation of Undirected Graph as Adjacency list:

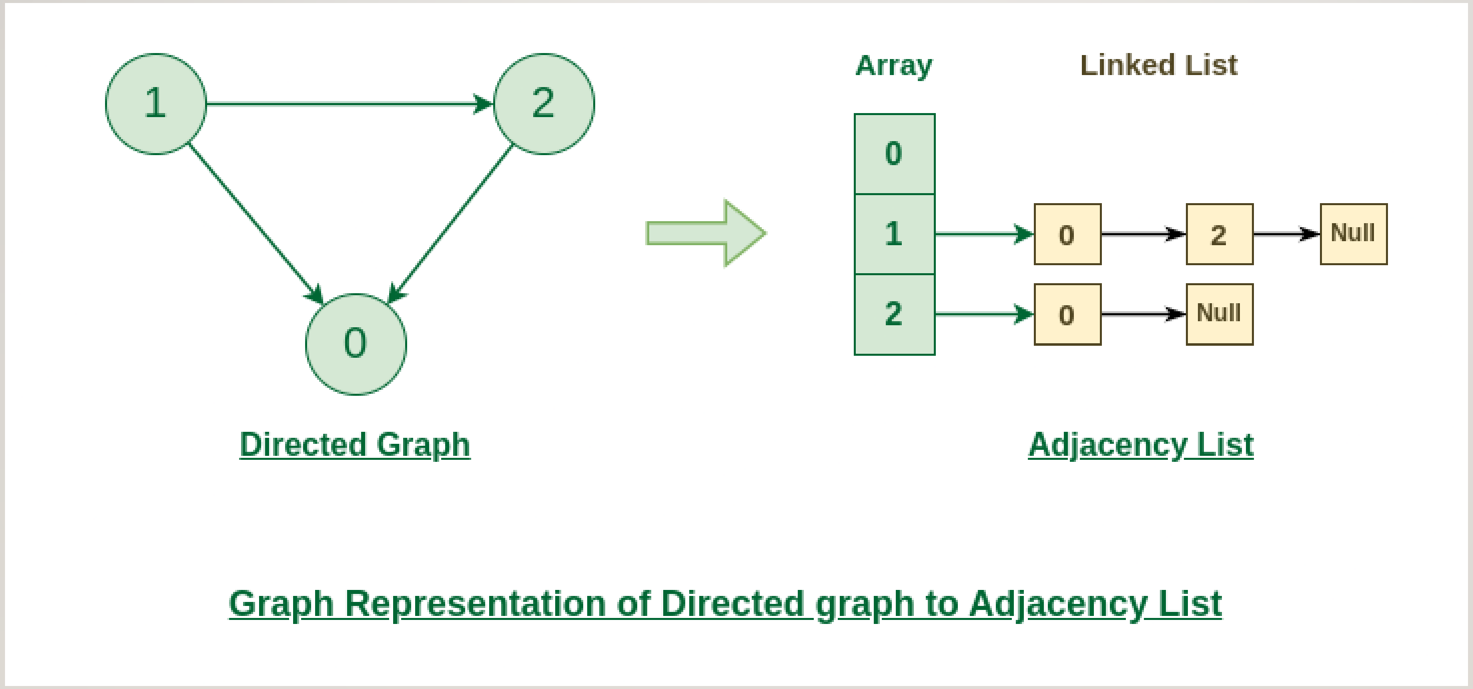
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Undirected graph to Adjacency List

Representation of Directed Graph as Adjacency list:

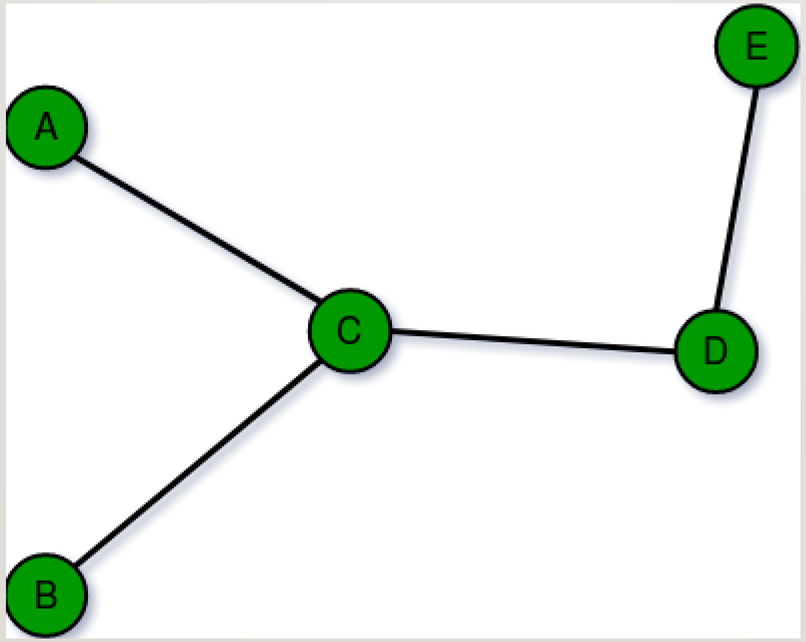
The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Types of Graphs

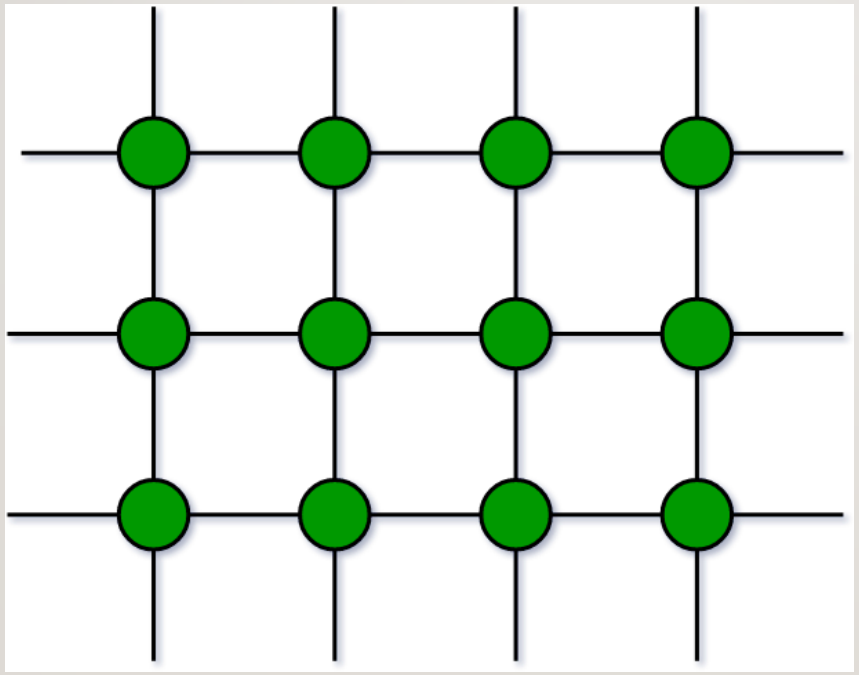
1. Finite Graphs

A graph is said to be finite if it has a finite number of vertices and a finite number of edges.



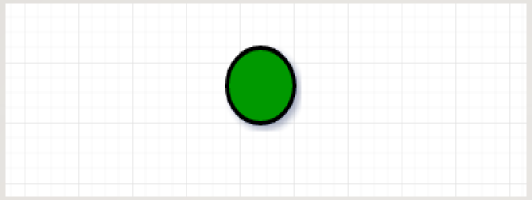
2. Infinite Graph:

A graph is said to be infinite if it has an infinite number of vertices as well as an infinite number of edges.



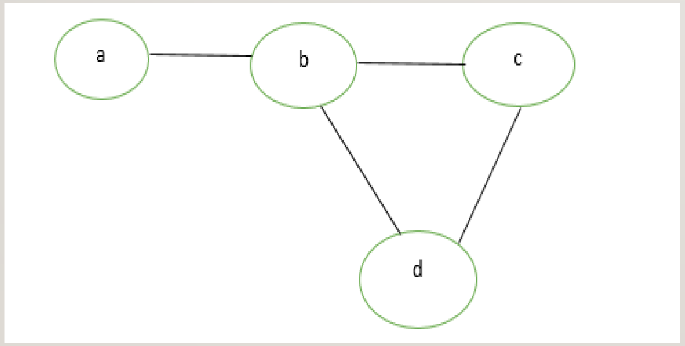
3. Trivial Graph:

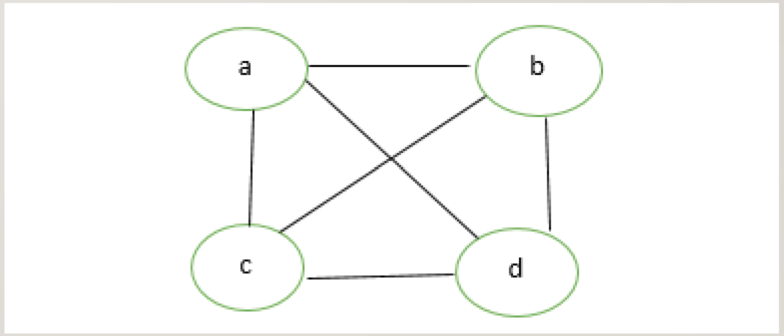
A trivial graph is a graph with only one vertex and no edges. It is also known as a singleton graph or a single vertex graph.



4. Simple Graph:

A simple graph is a graph that does not contain more than one edge between the pair of vertices. A simple railway track connecting different cities is an example of a simple graph.

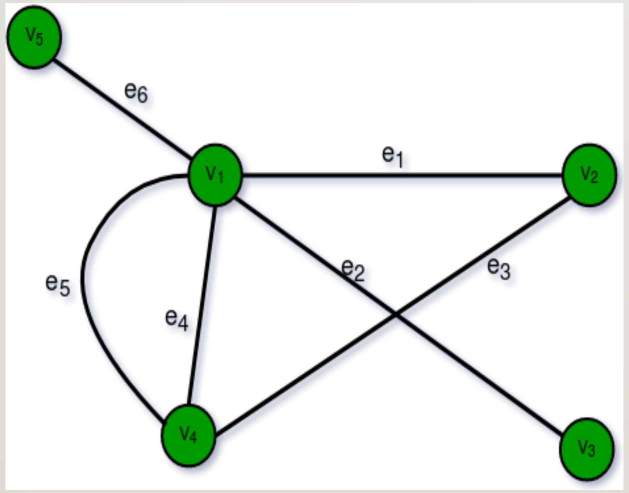




5. Multi Graph:

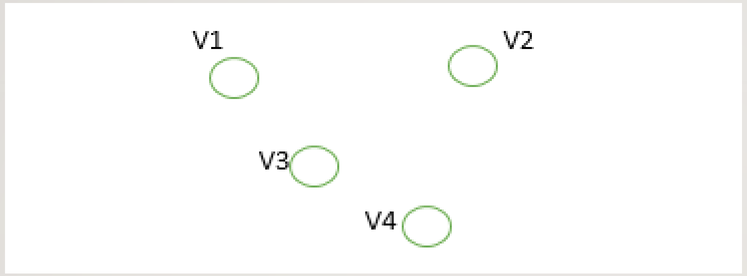
Any graph which contains some parallel edges but doesn't contain any self-loop is called a multigraph. For example a Road Map.

- **Parallel Edges:** If two vertices are connected with more than one edge then such edges are called parallel edges that are many routes but one destination.
- **Loop:** An edge of a graph that starts from a vertex and ends at the same vertex is called a loop or a self-loop



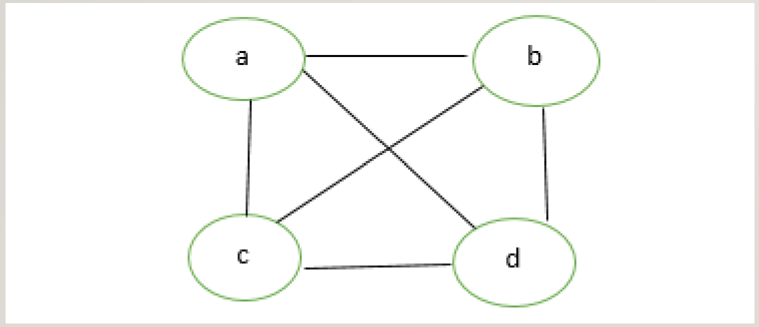
6. Null Graph:

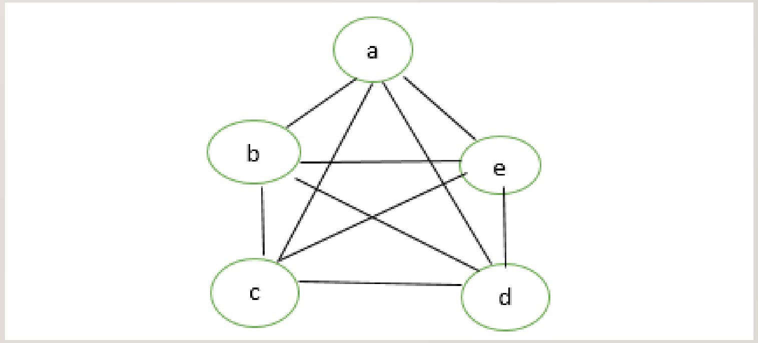
it is a graph with only vertices and no connections between them. A null graph can also be referred to as an edgeless graph, an isolated graph, or a discrete graph.



7. Complete Graph:

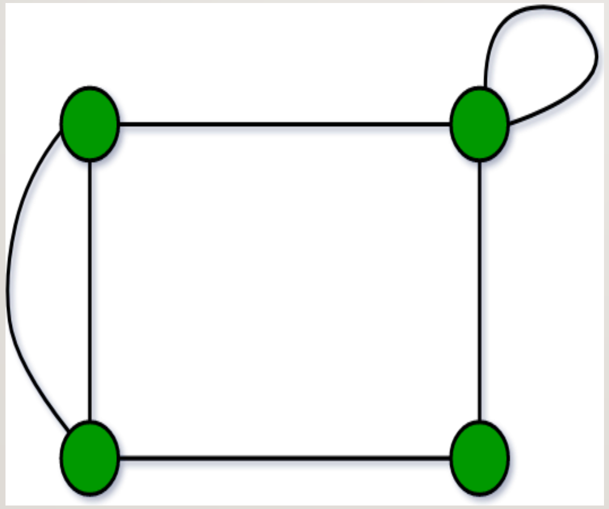
A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attached with n-1 edges or the rest of the vertices in the graph. A complete graph is also called Full Graph.





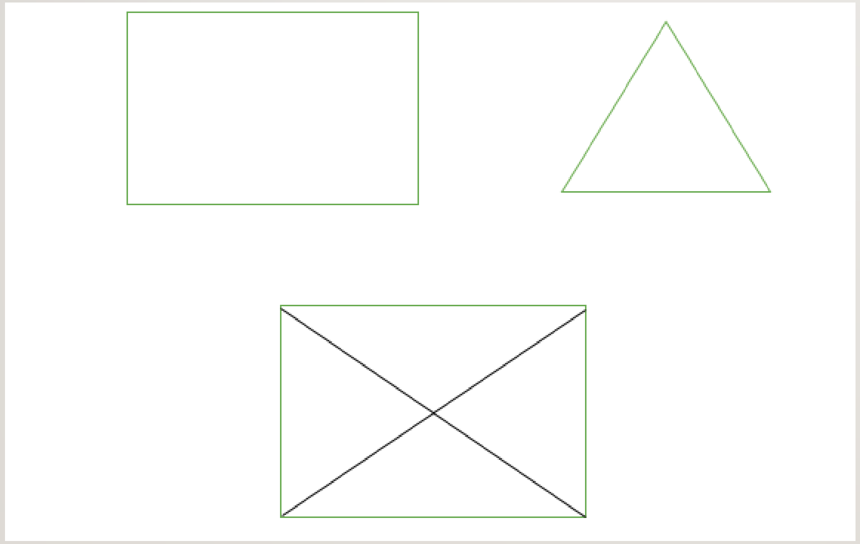
8. Pseudo Graph:

A graph G with a self-loop and some multiple edges is called a pseudo graph. A pseudograph is a type of graph that allows for the existence of self-loops (edges that connect a vertex to itself) and multiple edges (more than one edge connecting two vertices). In contrast, a simple graph is a graph that does not allow for loops or multiple edges.



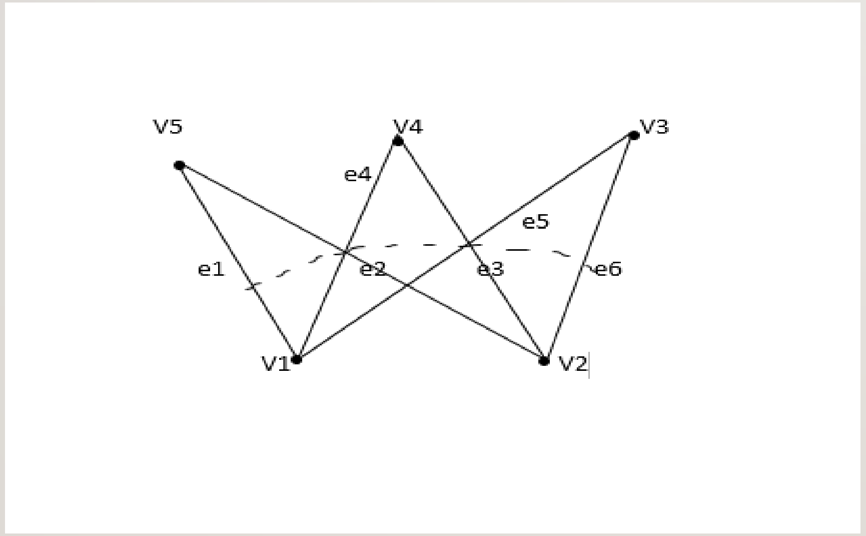
9. Regular Graph:

A simple graph is said to be regular if all vertices of graph G are of equal degree. All complete graphs are regular but vice versa is not possible. A regular graph is a type of undirected graph where every vertex has the same number of edges or neighbors. In other words, if a graph is regular, then every vertex has the same degree.



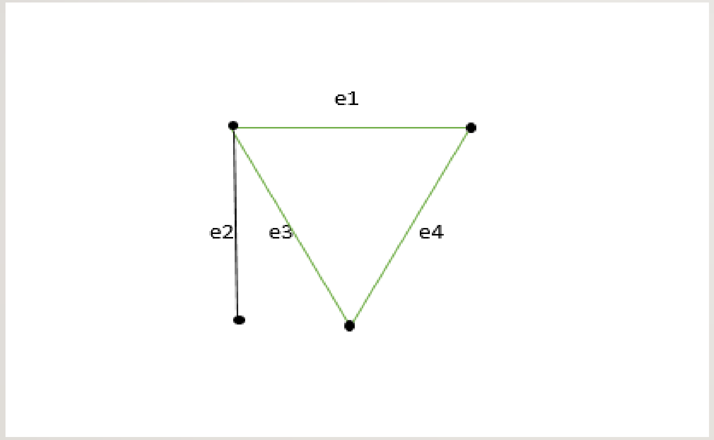
10. Bipartite Graph:

A graph G = (V, E) is said to be a bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has one end in V1(G) and another end in V2(G). The partition V1 U V2 = V is called Bipartite of G. Here in the figure: V1(G)={V5, V4, V3} and V2(G)={V1, V2}



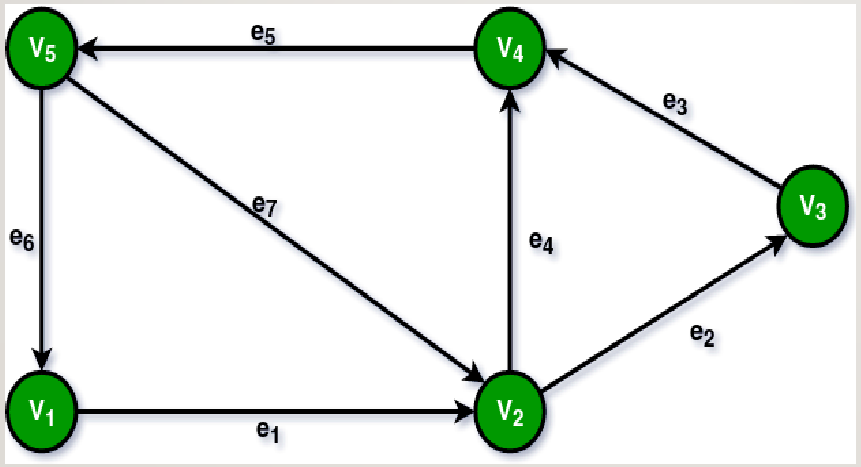
11. Labeled Graph:

If the vertices and edges of a graph are labeled with name, date, or weight then it is called a labeled graph. It is also called Weighted Graph.



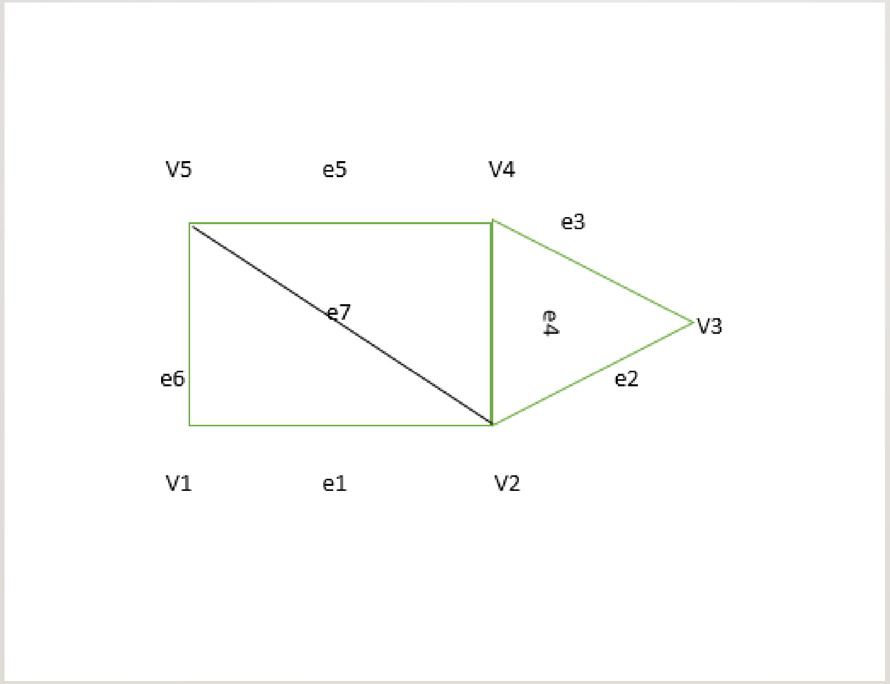
12. Digraph Graph:

A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) are called a Digraph. It is also called *Directed Graph*. The ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj. Here in the figure: e1 = (V1, V2) e2 = (V2, V3) e4 = (V2, V4)



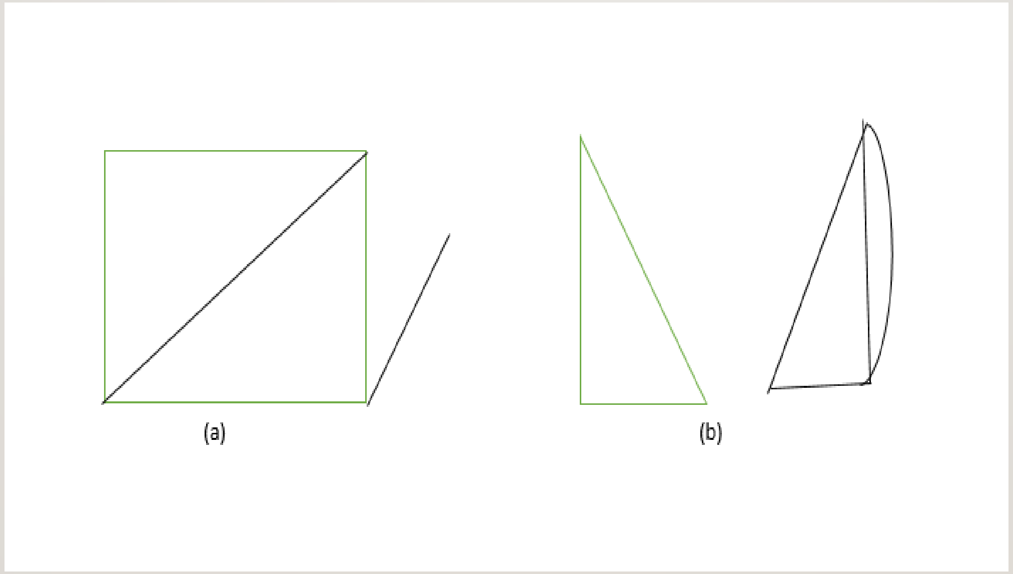
13. Subgraph:

A graph G1 = (V1, E1) is called a subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.



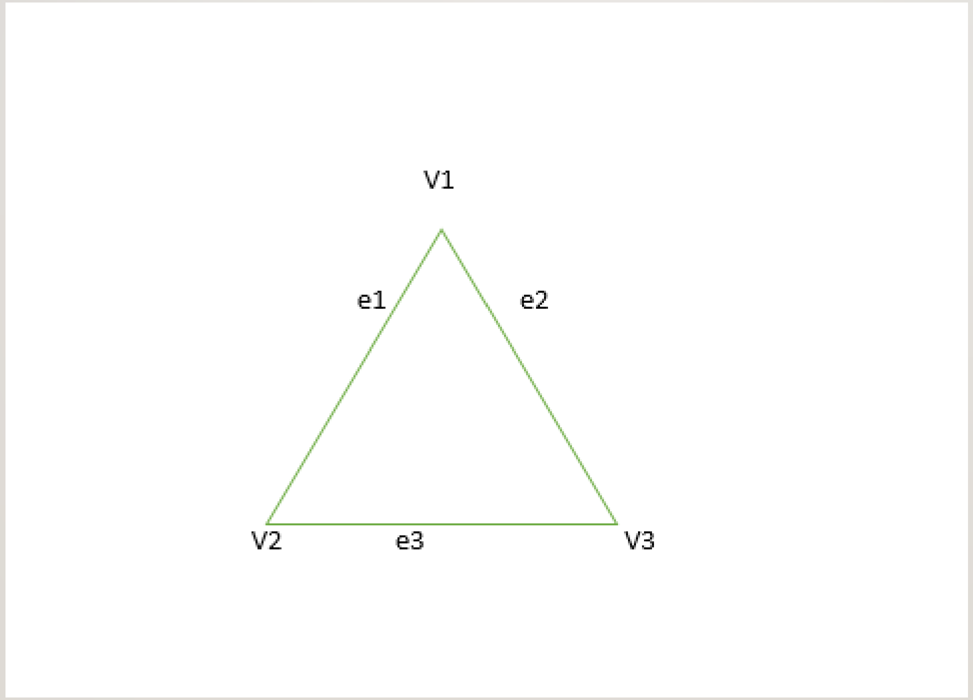
14. Connected or Disconnected Graph:

Graph G is said to be connected if any pair of vertices (Vi, Vj) of a graph G is reachable from one another. Or a graph is said to be connected if there exists at least one path between each and every pair of vertices in graph G, otherwise, it is disconnected. A null graph with n vertices is a disconnected graph consisting of n components. Each component consists of one vertex and no edge.



15. Cyclic Graph:

A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – (Vn, V1) are called cyclic graph.



Difference Between Graph and Tree

Feature	Graph	Tree
Definition	A collection of nodes (vertices) and edges, where edges connect nodes.	A hierarchical data structure consisting of nodes connected by edges with a single root node.
Structure	Can have cycles (loops) and disconnected components.	No cycles; connected structure with exactly one path between any two nodes.
Root Node	No root node; nodes may have multiple parents or no parents at all.	Has a designated root node that has no parent.
Node Relationship	Relationships between nodes are arbitrary.	Parent-child relationship; each node (except the root) has exactly one parent.
Edges	Each node can have any number of edges.	If there is **n** nodes then there would be **n-1** number of edges
Traversal Complexity	Traversal can be complex due to cycles and disconnected components.	Traversal is straightforward and can be done in linear time.
Application	Used in various scenarios like social networks, maps, network optimization, etc.	Commonly used for hierarchical data representation like file systems, organization charts, HTML DOM, XML documents, etc.
Examples	Social networks, road networks, computer networks.	File systems, family trees, HTML DOM structure.

Graphs Traversal

To traverse a Graph means to start in one vertex, and go along the edges to visit other vertices until all vertices, or as many as possible, have been visited.

The two most common ways a Graph can be traversed are:

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

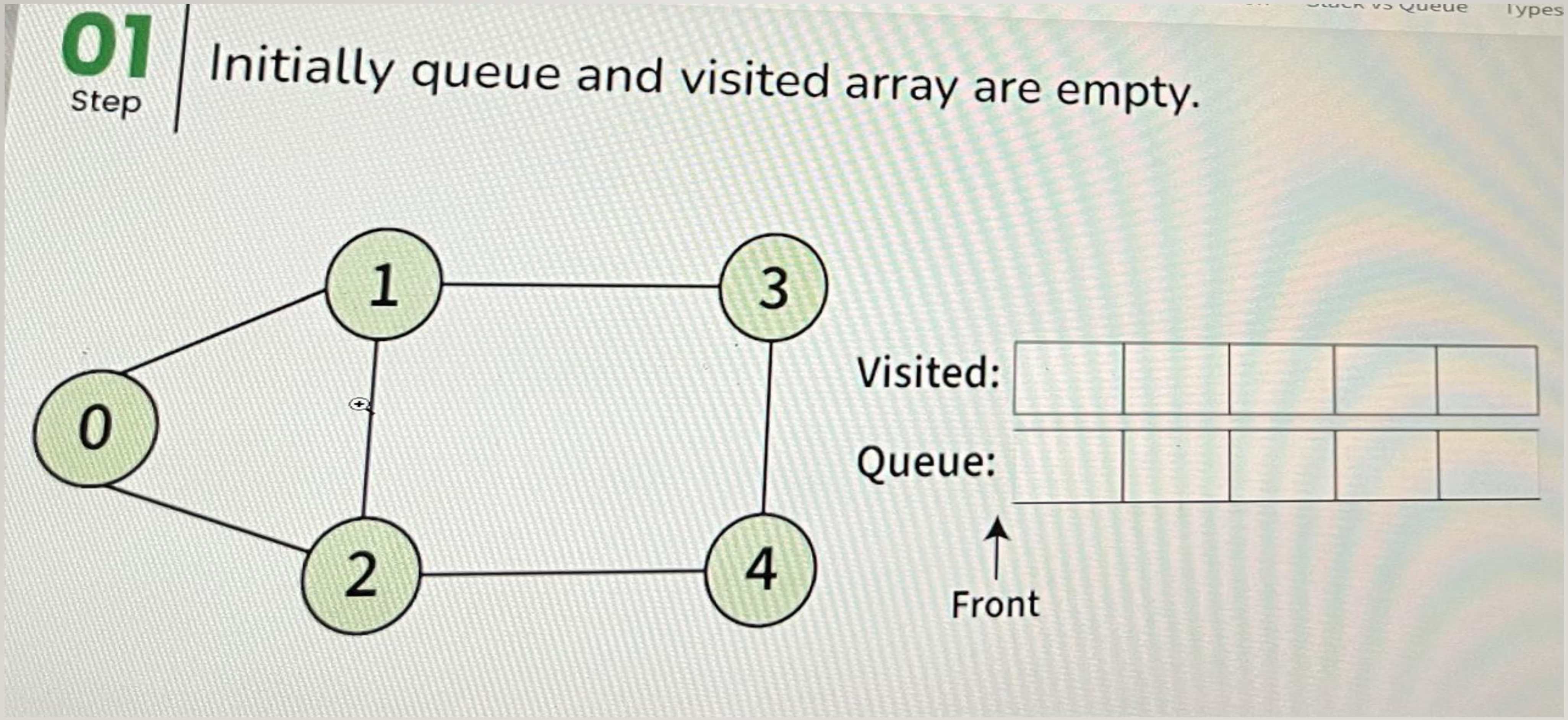
BFS from a Given Source:

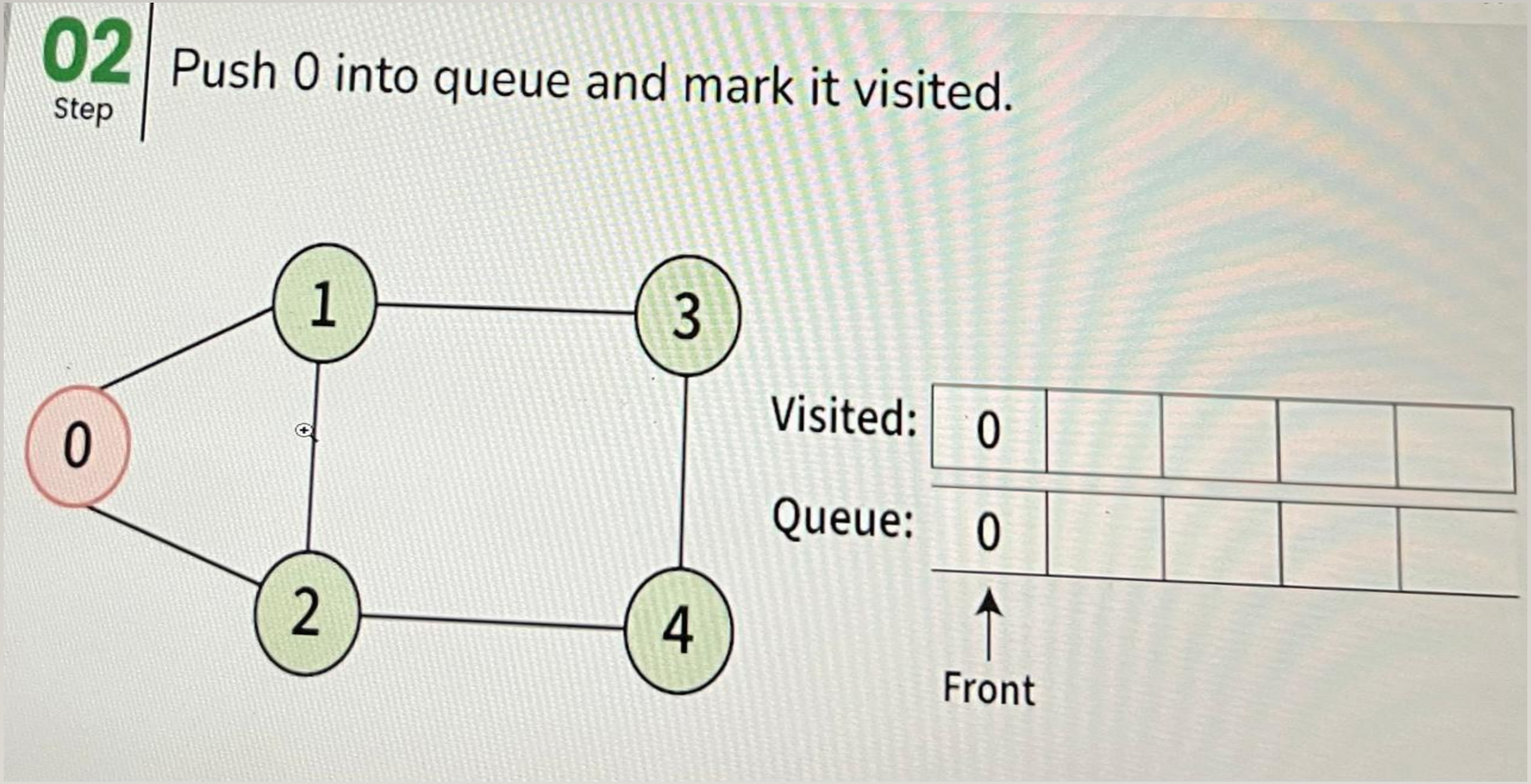
Initialization: Enqueue the given source vertex into a queue and mark it as visited.

1. **Exploration:** While the queue is not empty:
    1. Dequeue a node from the queue and visit it (e.g., print its value).
    2. For each unvisited neighbor of the dequeued node:
        1. Enqueue the neighbor into the queue.
        2. Mark the neighbor as visited.
2. **Termination:** Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

Example:

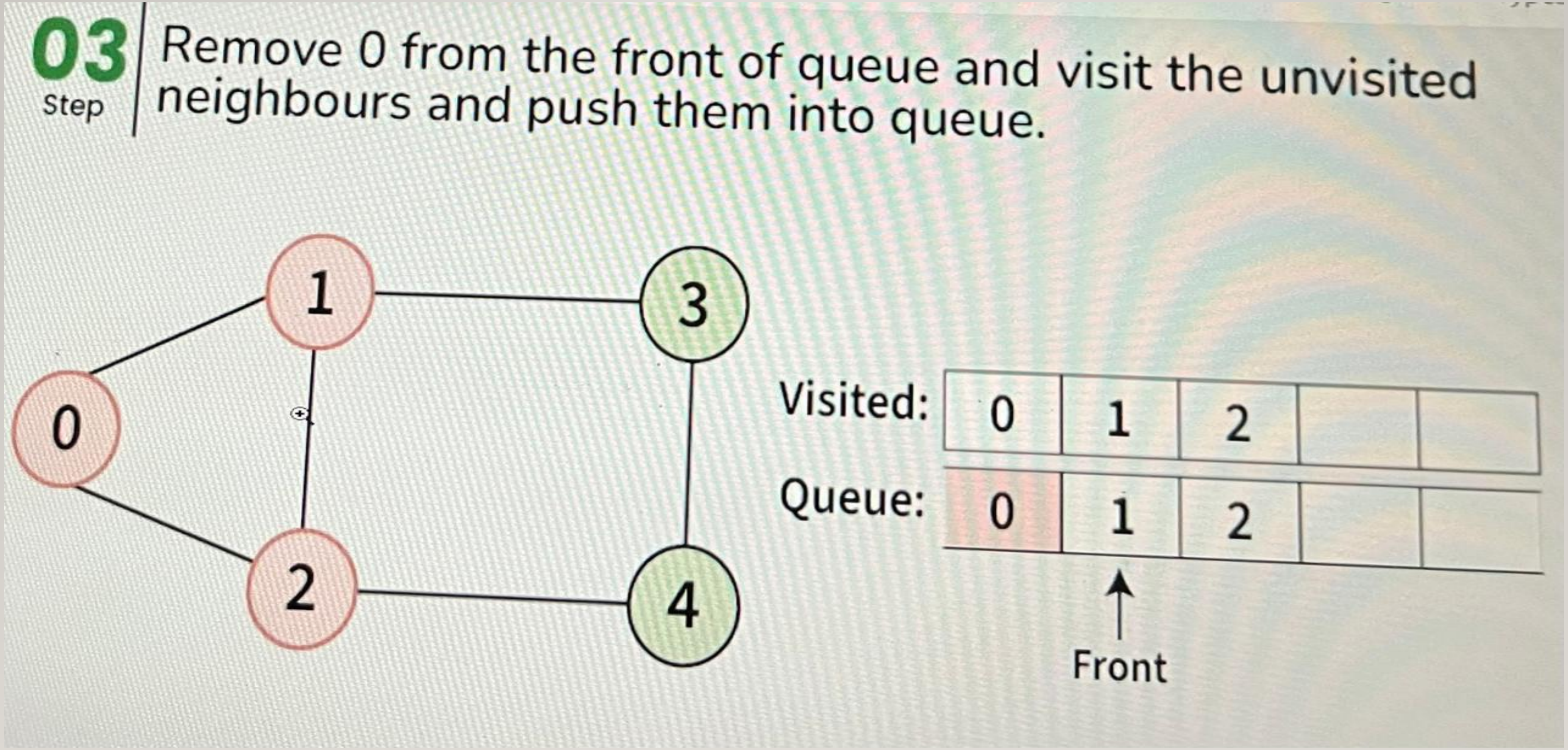

Stack vs Queue
Types
01
Step
Initially queue and visited array are empty.
0
1
2
3
4
Visited:
Queue:
Front



02 Step

Push 0 into queue and mark it visited.

Visited:	0				
Queue:	0				



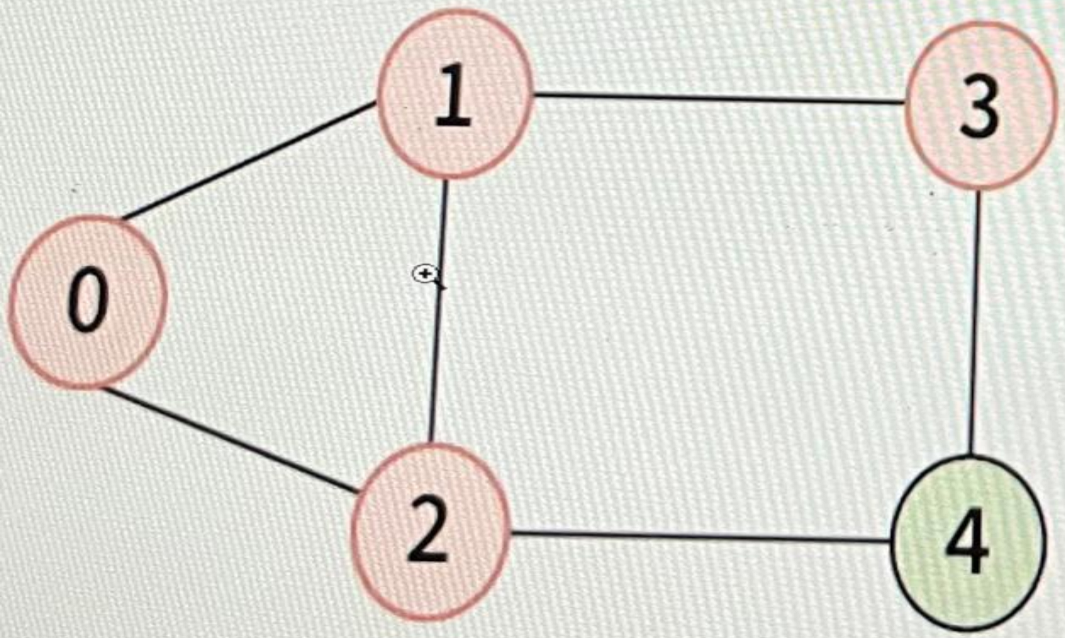
03 Step

Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.

Visited:	0	1	2		
Queue:	0	1	2		

04 Step

Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.

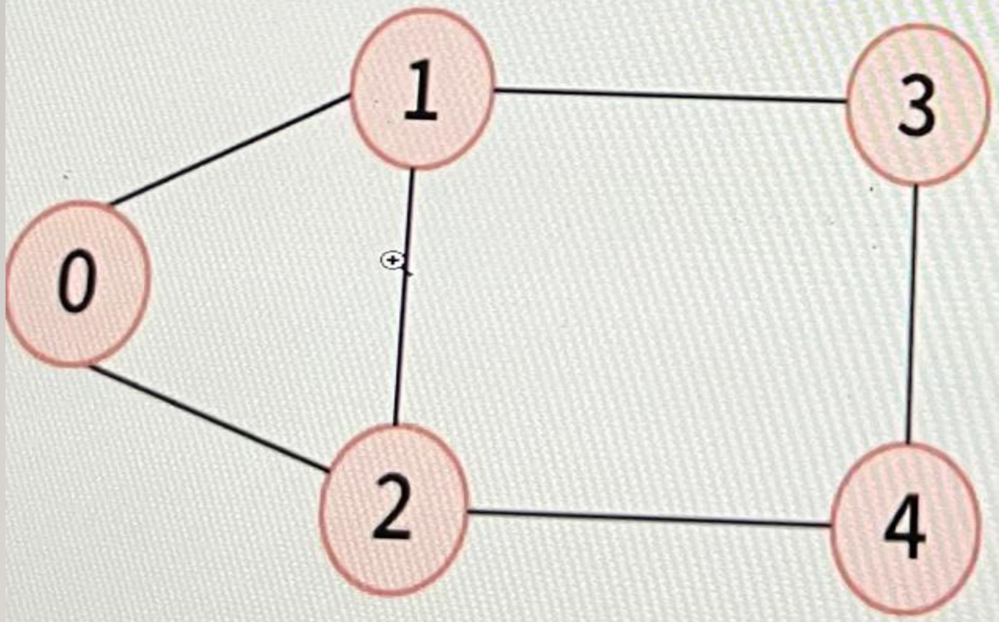


Visited:	0	1	2	3	
Queue:	1	2	3		

↑ Front (at 2)

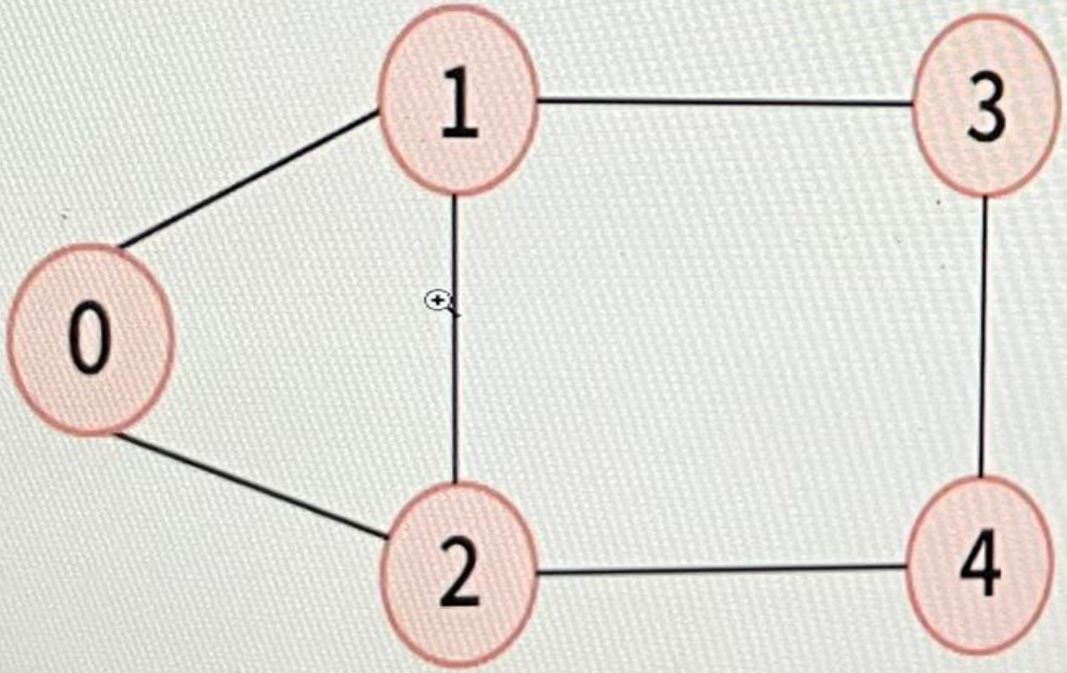
05 Step

Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2	3	4
Queue:	2	3	4		
		↑ Front			

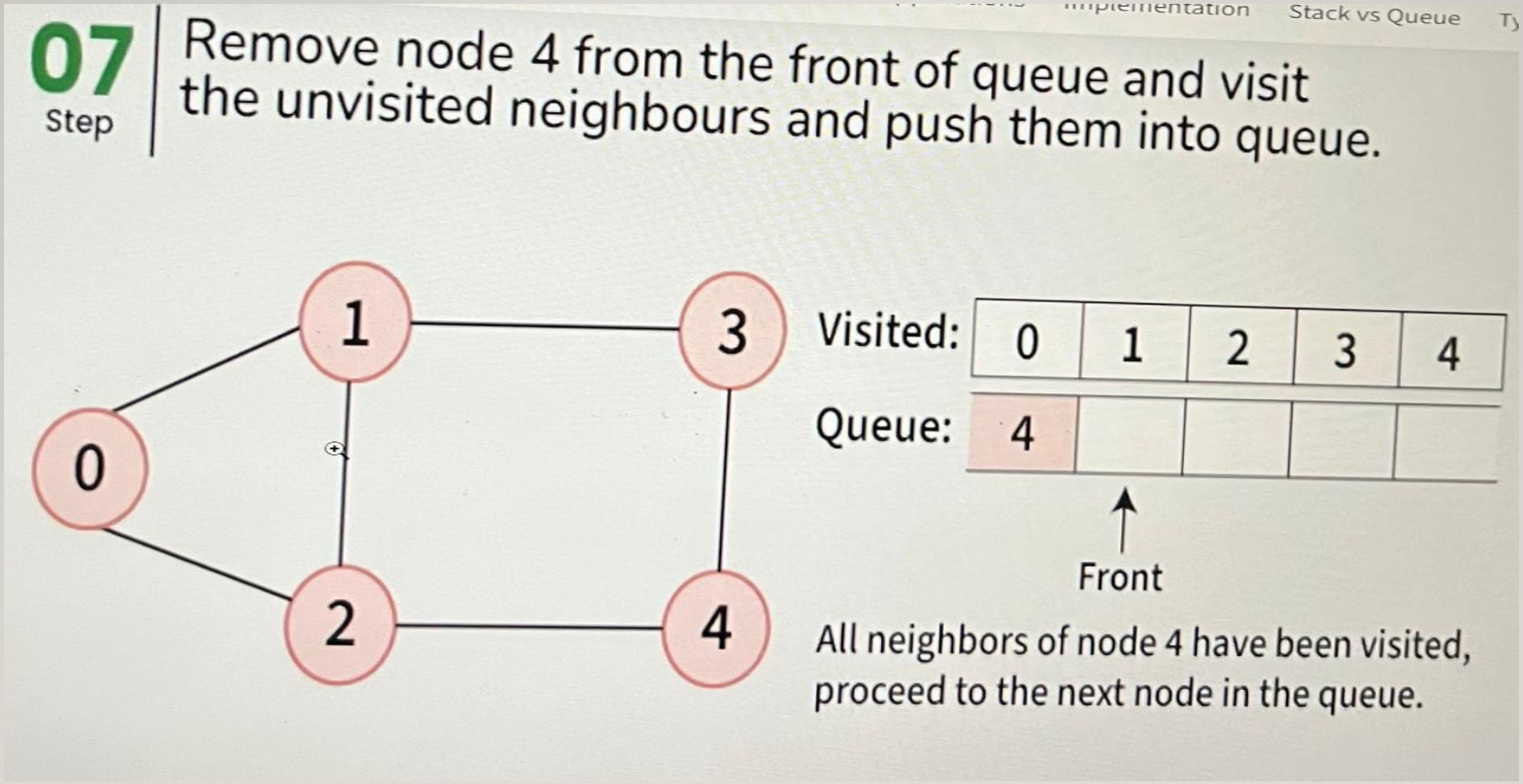
06 Step | Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2	3	4
Queue:	3	4			

$\uparrow$
Front

All neighbors of node 3 have been visited, proceed to the next node in the queue.



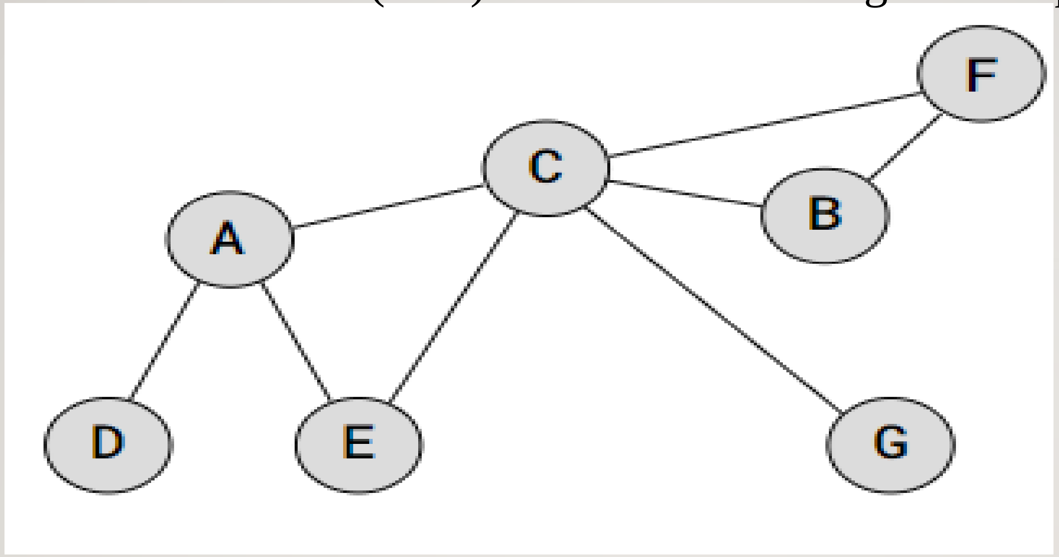
07 Step

Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Visited:	0	1	2	3	4
Queue:	4				

All neighbors of node 4 have been visited, proceed to the next node in the queue.

Example: How Breadth First Search (BFS) traversal runs on a given Graph, starting in vertex D.



Result: D,A,C,E,B,F,G

Complexity of BFS algorithm

The time complexity of BFS algorithm is **O(V+E)**, since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

Depth First Search or DFS for a Graph

It is a recursive algorithm to search all the vertices of a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

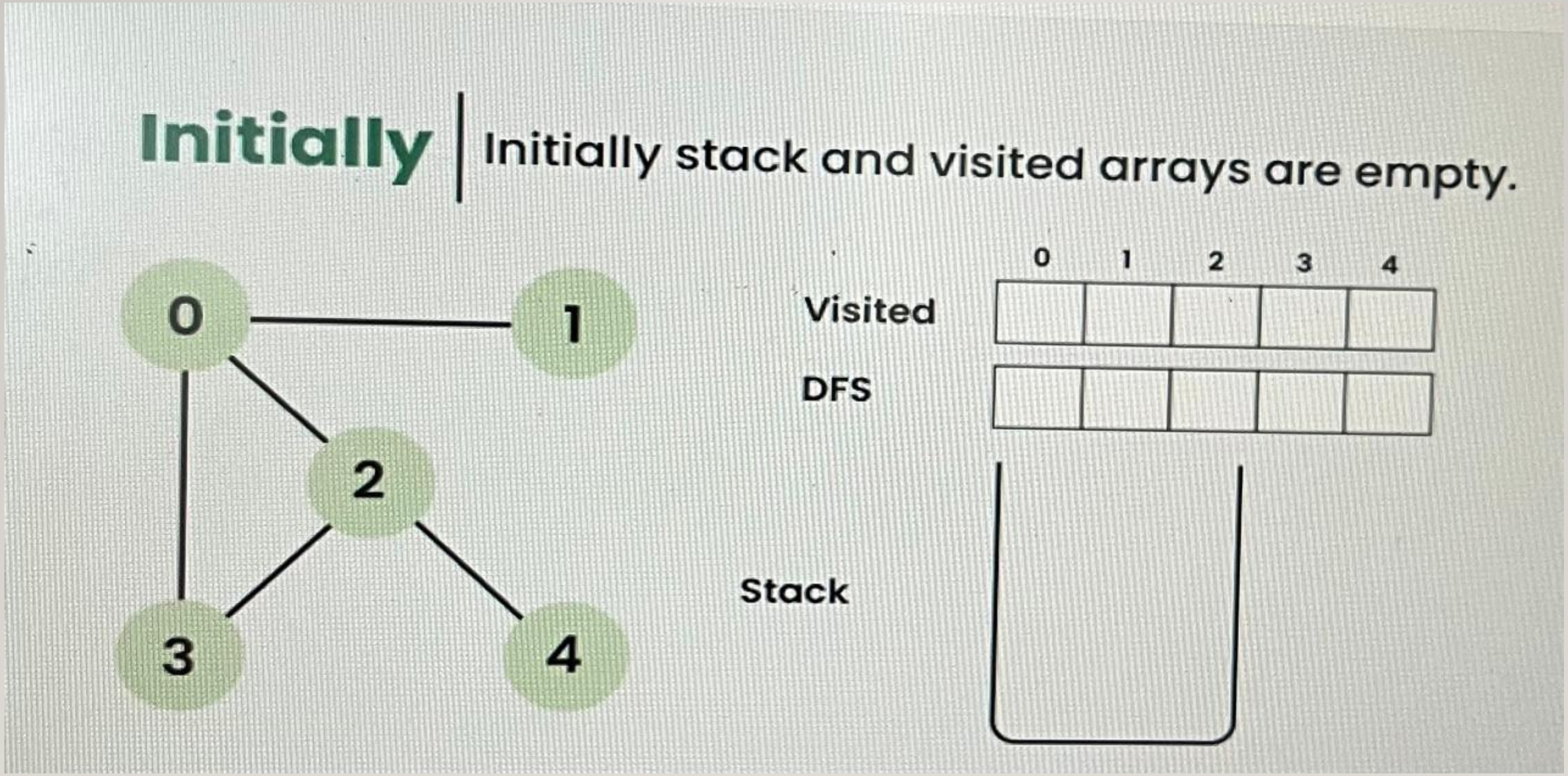
The step by step process to implement the DFS traversal is given as follows –

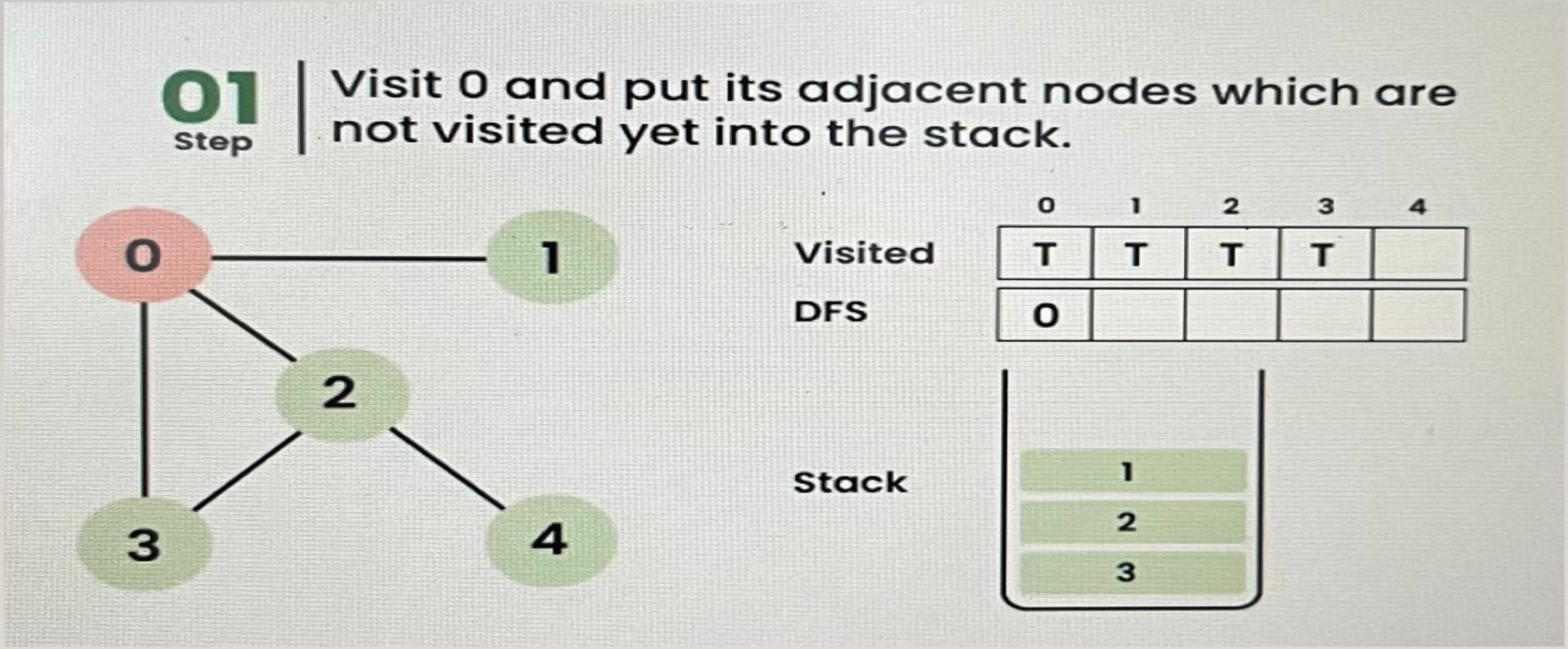
1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Pseudocode

```
1.DFS(G,v)   ( v is the vertex where the search starts )
2.       Stack S := {};   ( start with an empty stack )
3.       for each vertex u, set visited[u] := false;
4.       push S, v;
5.       while (S is not empty) do
6.          u := pop S;
7.          if (not visited[u]) then
8.             visited[u] := true;
9.             for each unvisited neighbour w of u
10.                 push S, w;
11.           end if
12.        end while
13.     END DFS()
```

Example:


Initially
Initially stack and visited arrays are empty.
0
1
2
3
4
Visited
DFS
Stack

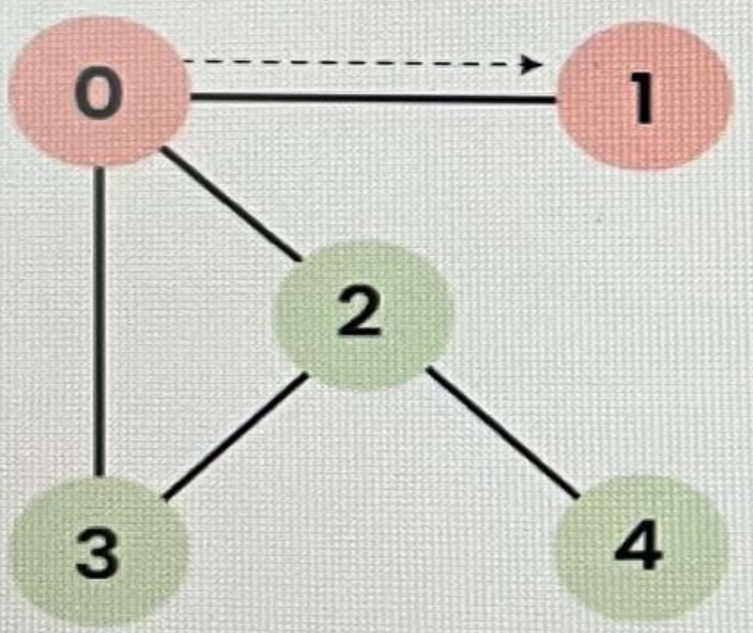


01 Step | Visit 0 and put its adjacent nodes which are not visited yet into the stack.

	0	1	2	3	4
Visited	T	T	T	T	
DFS	0				

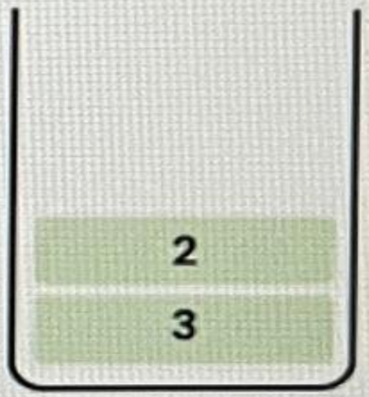
02
Step

Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



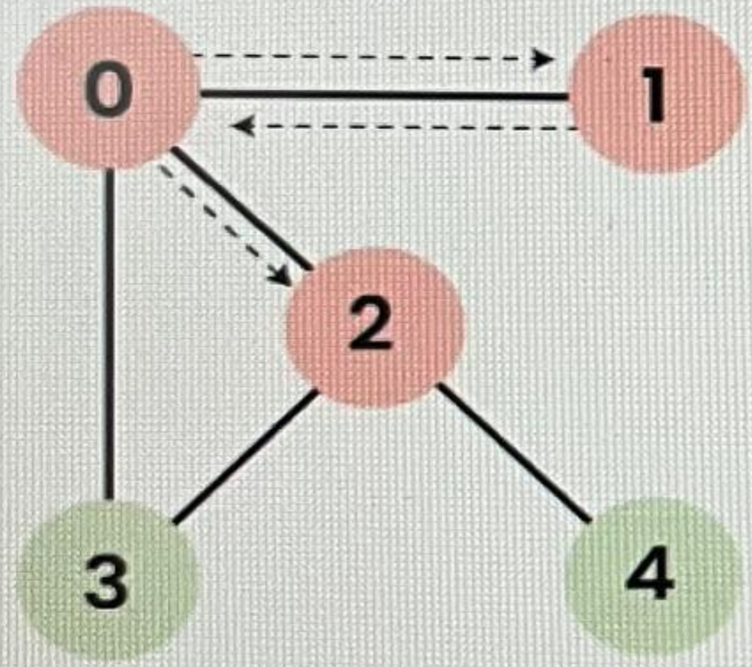
	0	1	2	3	4
Visited	T	T	T	T	
DFS	0	1			

Stack



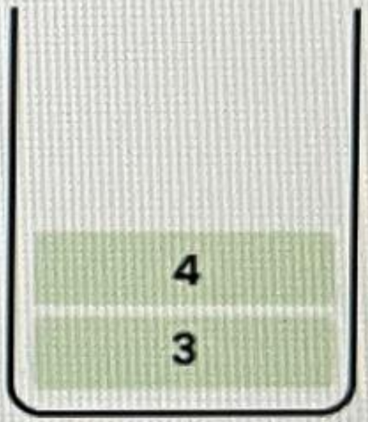
03 Step

Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



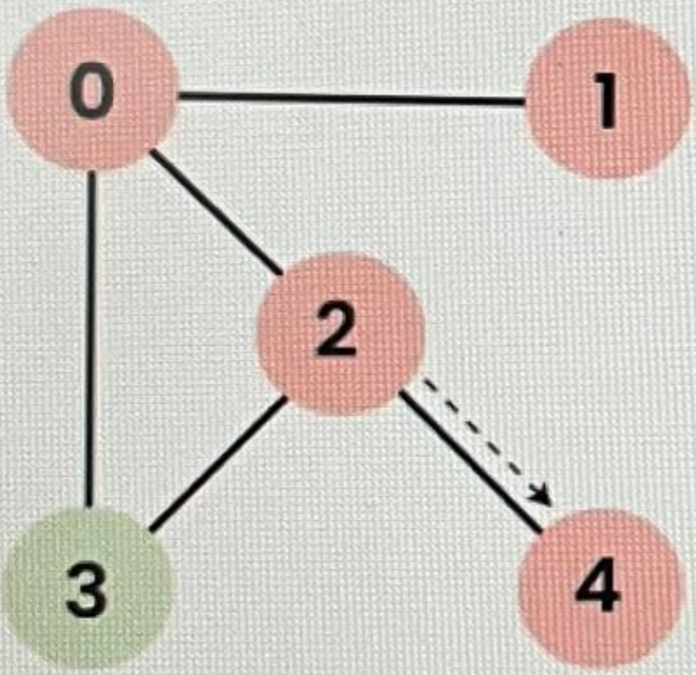
	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2		

Stack



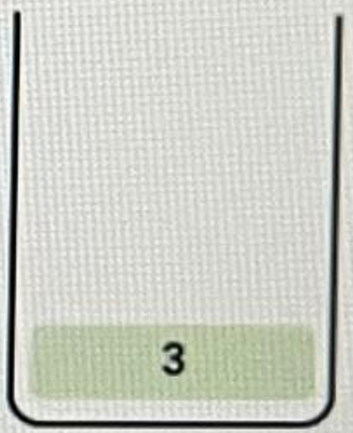
04 Step

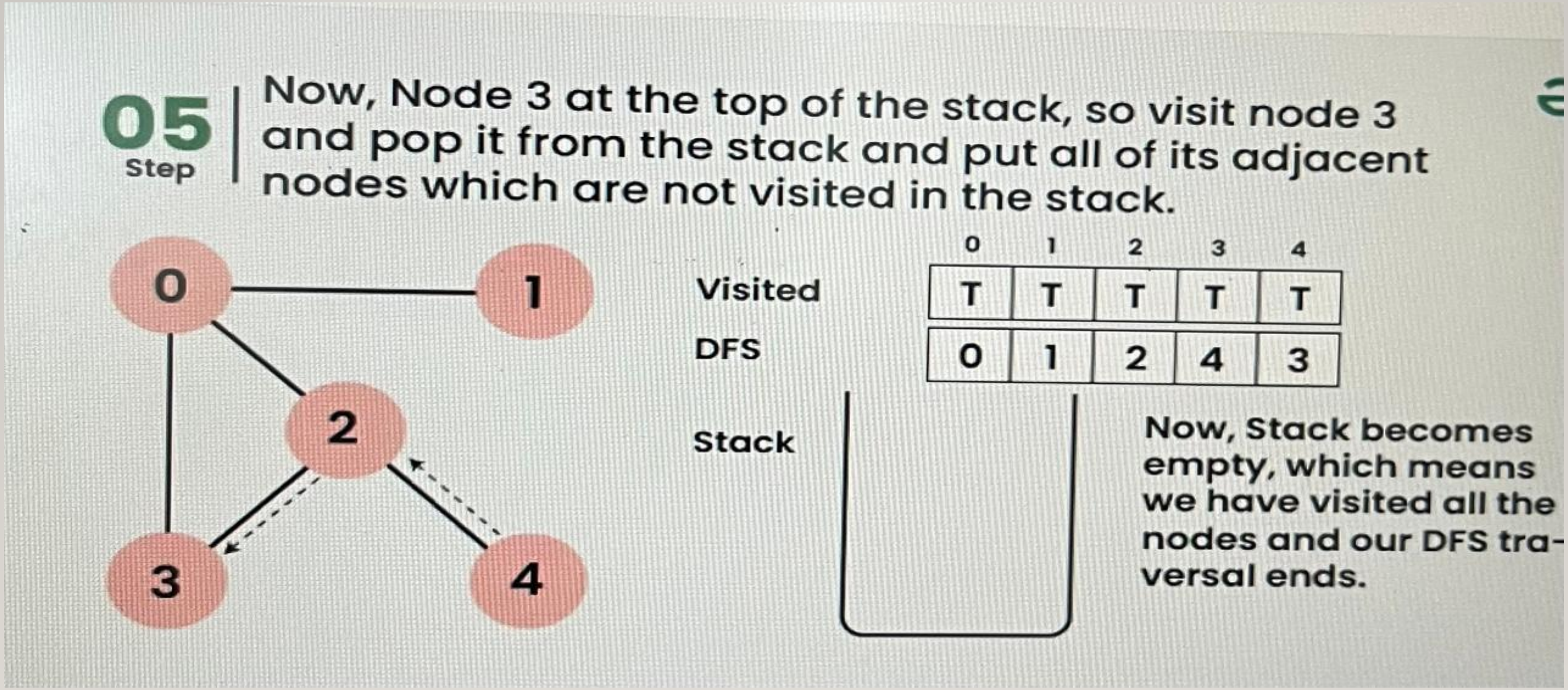
Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	

Stack





05 Step

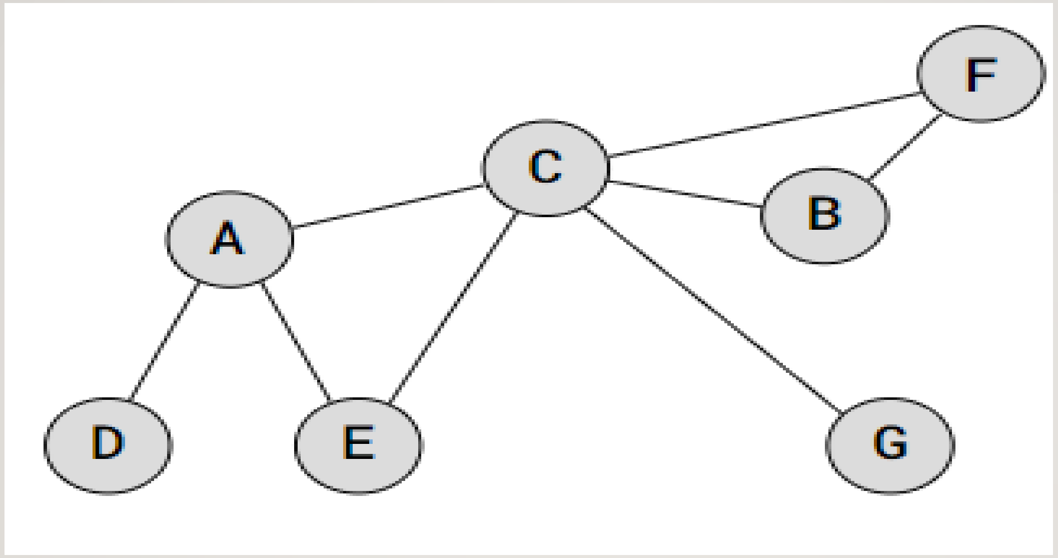
Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	3

Stack

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Example: How Depth First Search (DFS) traversal runs on a given Graph, starting in vertex D.



Result: D,A,C,B,F,E,G

Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Auxiliary Space: O(V), since an extra visited array of size V is required.