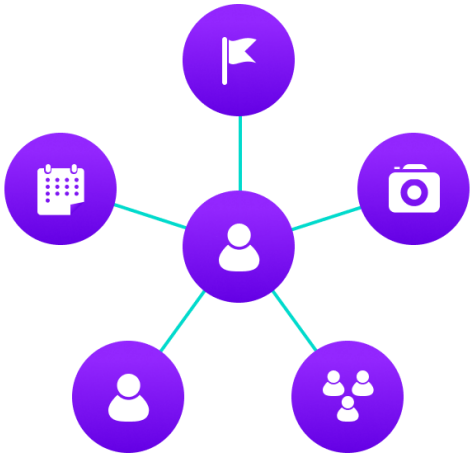
Graph Data Stucture

A graph data structure is a **collection of nodes that have data** and **are connected to other nodes**.

Let's try to understand this through an example. On facebook, **everything is a node**. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

Every **relationship is an edge** from one node to another. Whether you post a photo, join a group, like a page, etc., a **new edge is created for that relationship**.

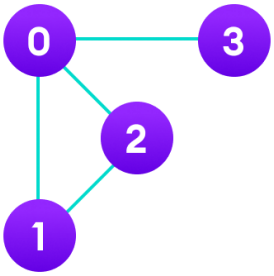


Example Of Graph Data Structure

All of facebook is then a **collection** of **these nodes and edges**. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of **vertices V**
- A collection of **edges E**

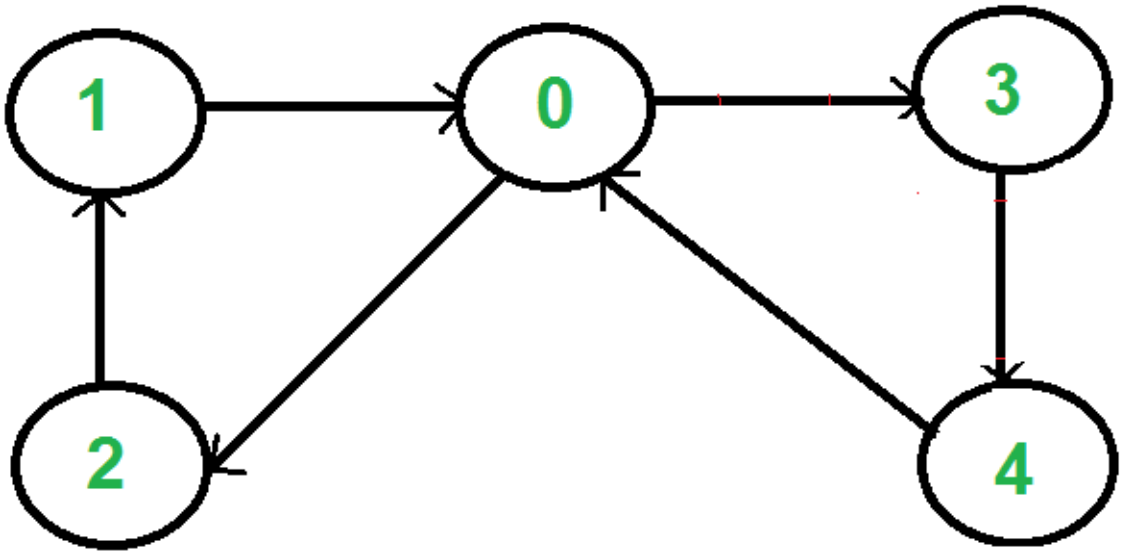


Vertices and edges

Graph Terminology

- **Adjacency**: A **vertex is said to be adjacent** to **another vertex if there is an edge** connecting them.
- **Path**: A **sequence of edges** that **allows you** to **go from vertex A to vertex B is called a path**. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.

- **Directed Graph**: A **directed graph** is defined as a type of **graph** where the **edges have a direction associated with them.**

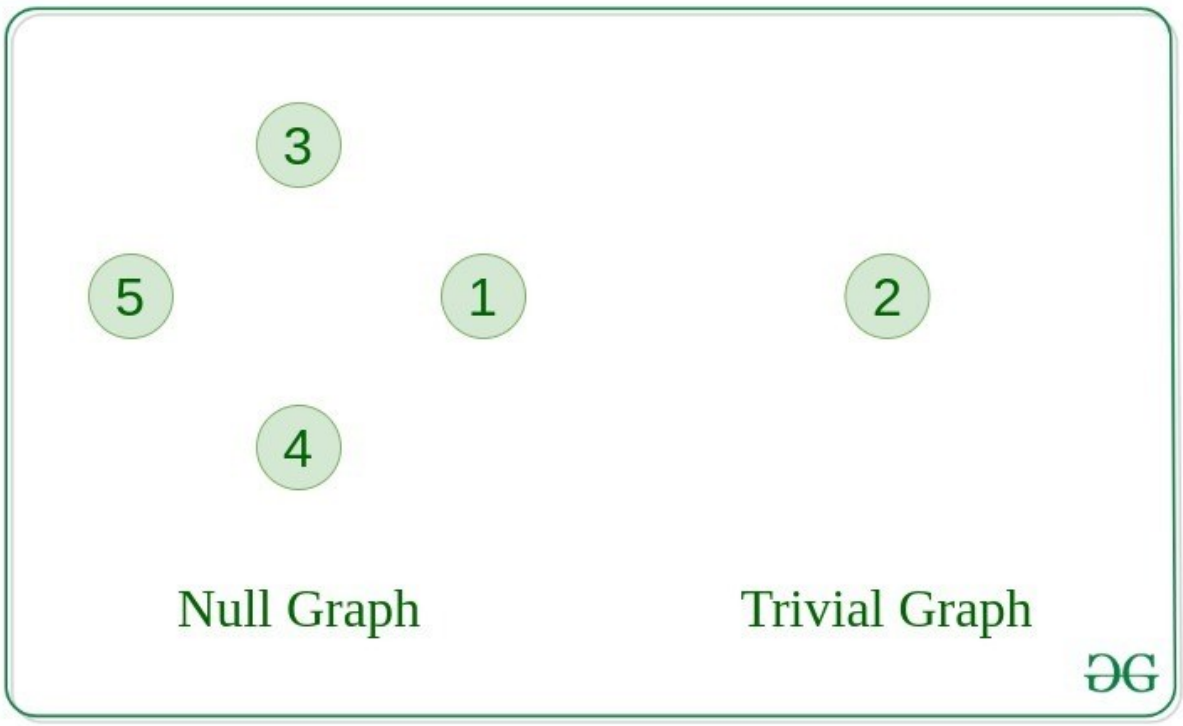


Null Graph

A graph is known as a null graph if there are no edges in the graph.

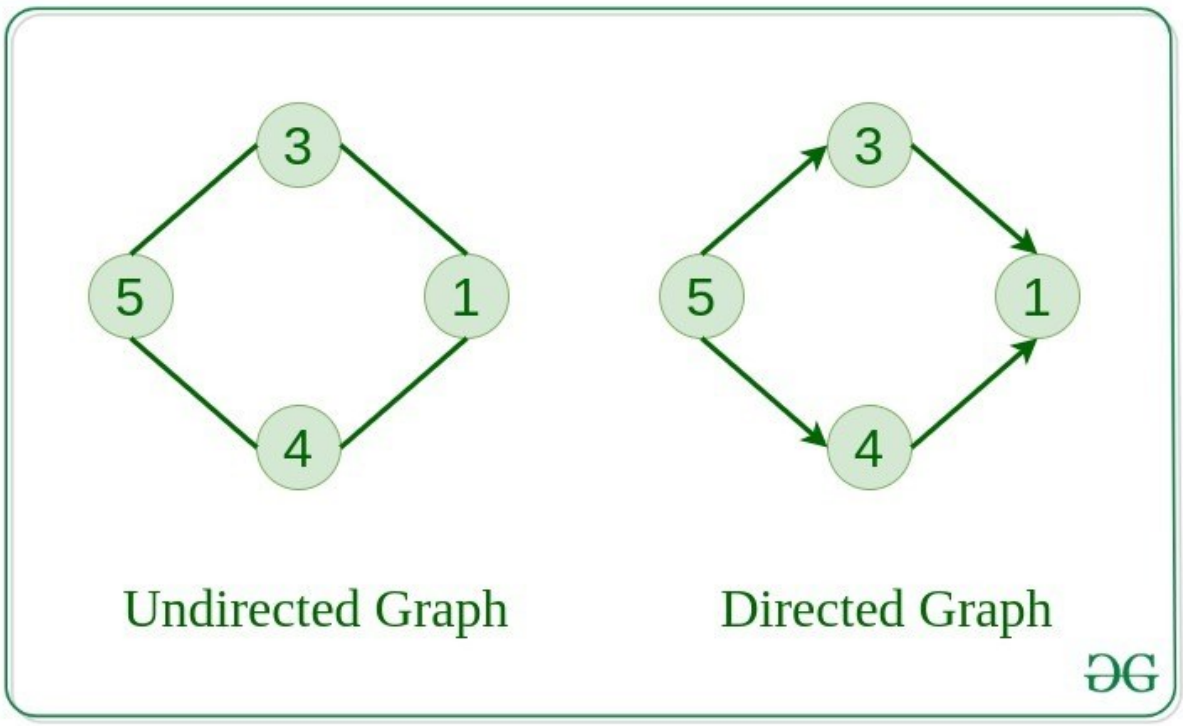
Trivial Graph

Graph having only a single vertex, it is also the smallest graph possible.



Undirected Graph

A graph in which edges do not have any direction.

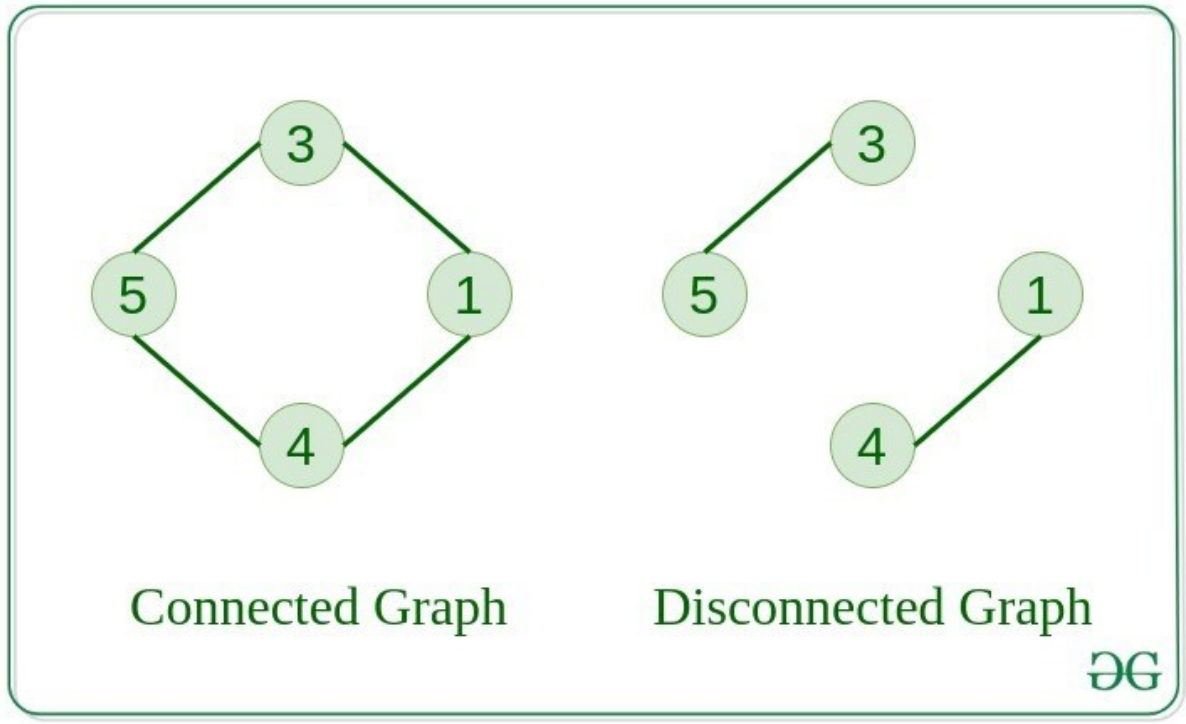


Connected Graph

The graph in which from one node we can visit any other node in the graph is known as a connected graph.

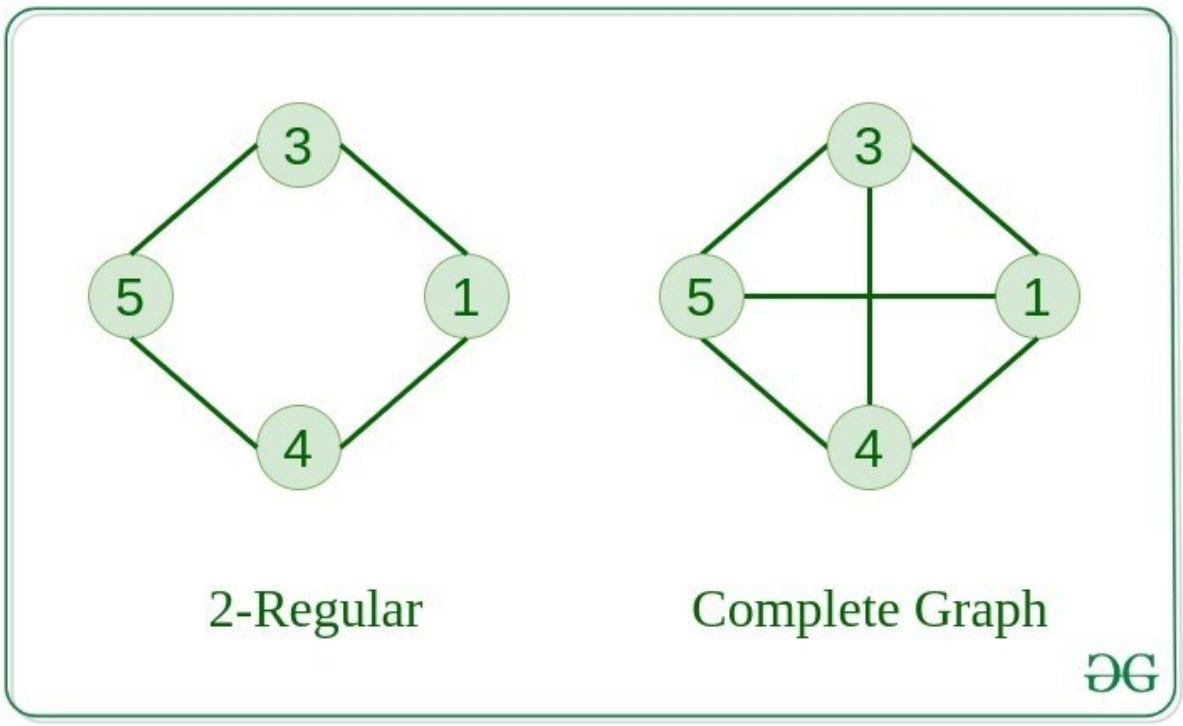
Disconnected Graph

The graph in which at least one node is not reachable from a node is known as a disconnected graph.



Complete Graph

The graph in which from each node there is an edge to each other node.

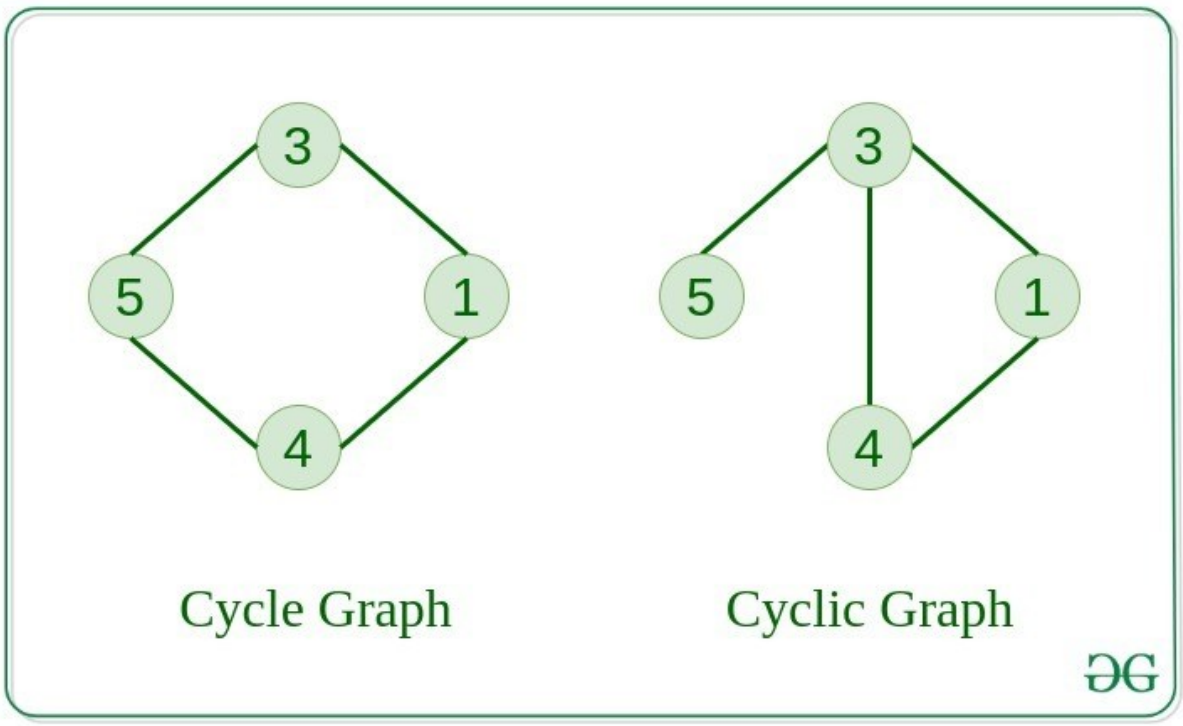


Regular Graph

The graph in which the **degree of every vertex** is **equal to K** is called K regular graph.

Cyclic Graph

A graph containing at least one cycle is known as a Cyclic graph



Graph Representation

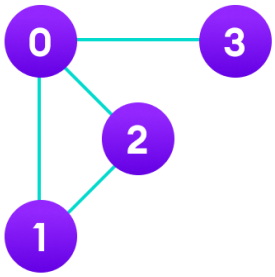
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a **2D array of V x V vertices**. Each row and column represent a **vertex**.

If the value of any element `a[i][j]` **is 1**, it **represents** that **there is an edge** connecting **vertex i and vertex j**.

The adjacency matrix for the graph we created above is

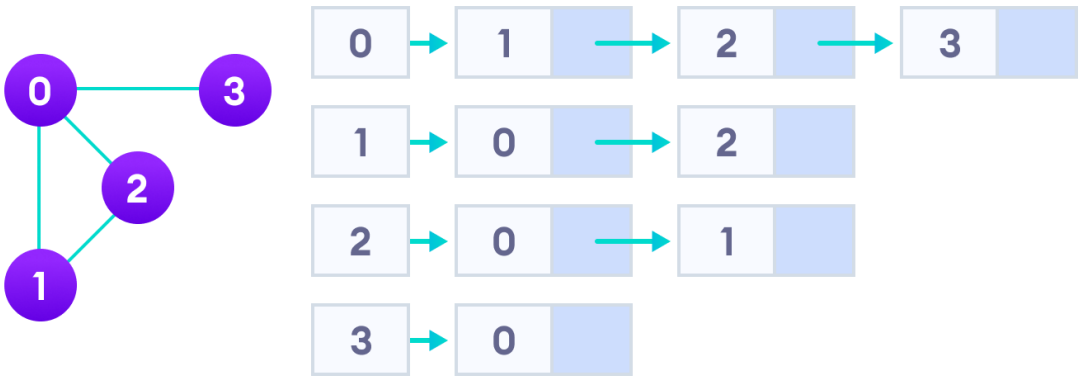


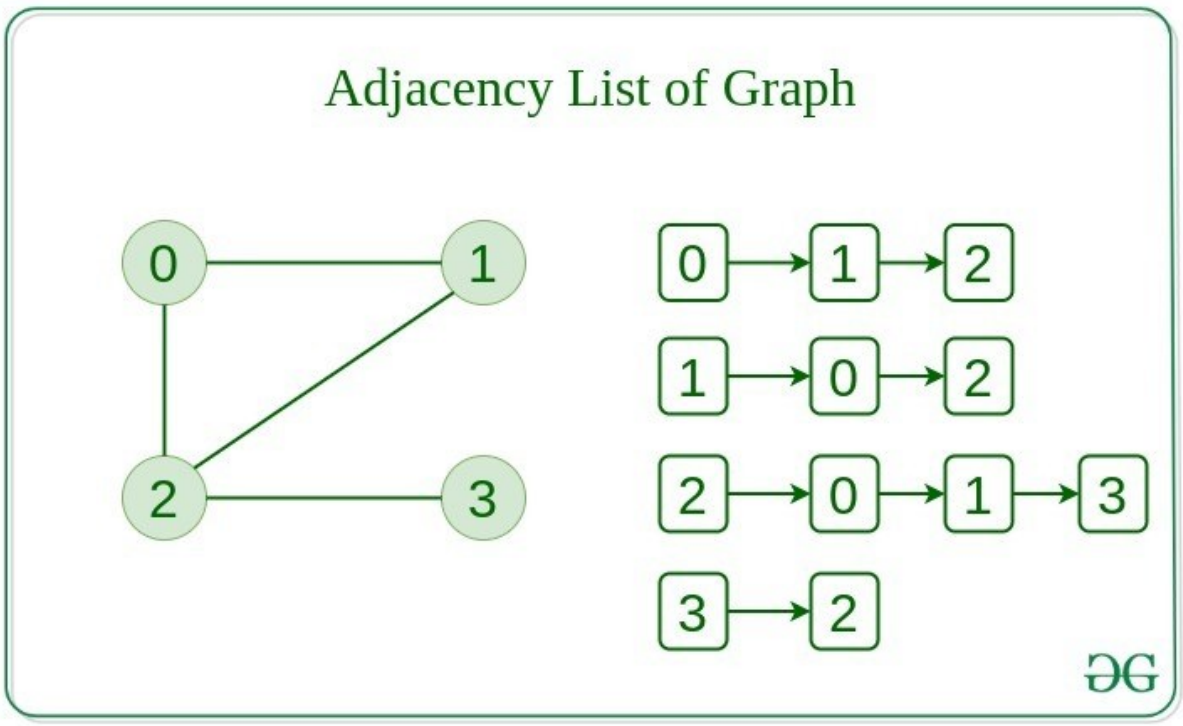
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Graph Adjacency Matrix

2. Adjacency List

An adjacency list represents a graph **as an array of linked lists.**





Adjacency List Representation
An adjacency list is **efficient** in terms of **storage because we only need to store the values** for the **edges**. For a graph with millions of vertices, this can mean a lot of saved space.

Graph Operations

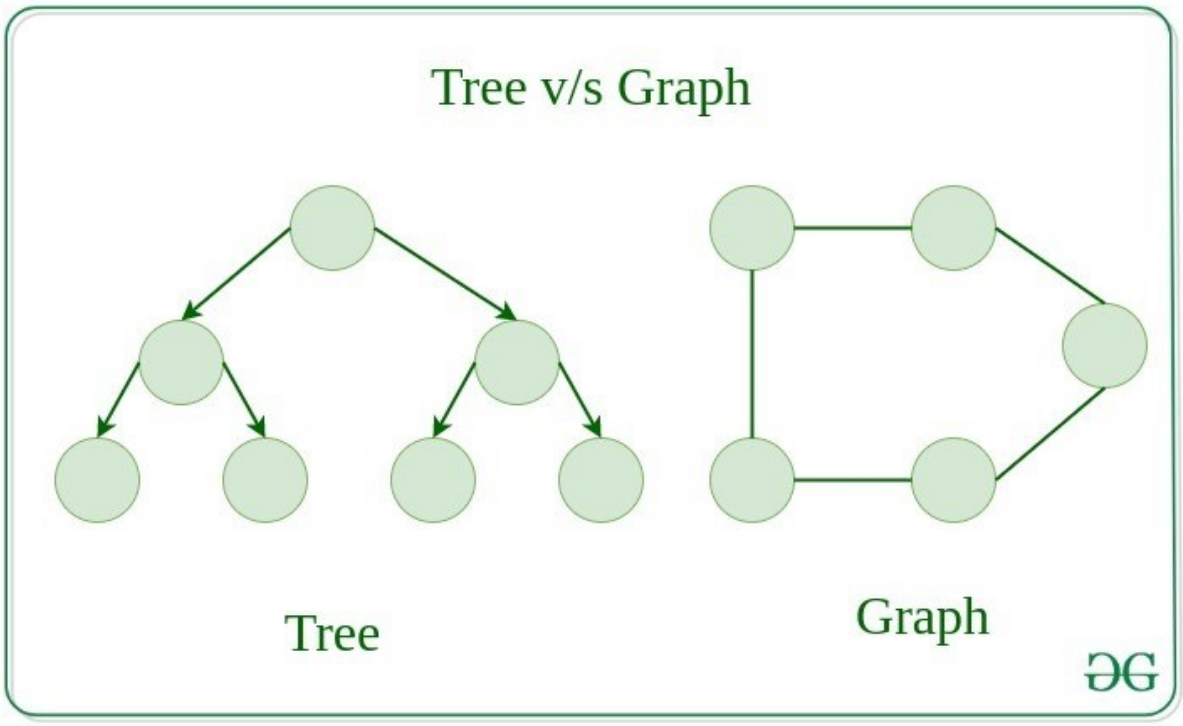
The most common graph operations are:

- **Check if the element is present in the graph**
- **Graph Traversal**
- **Add elements (vertex, edges) to graph**

- **Finding the path from one vertex to another**

Difference Between Tree And Graph:

Tree is a **restricted** type of **Graph** Data Structure, just with some more rules. **Every tree will always be a graph** but not all graphs will be trees.



Breadth First Search or BFS for a Graph

The **Breadth First Search (BFS)** algorithm is used to **search a graph data structure** for a **node** that **meets a set of criteria.**

It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Relation between BFS for Graph and Tree traversal:

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles, **so we may come to the same node again.** To avoid processing a node more than once, we divide the vertices into two categories:

- **Visited and**
- **Not visited.**

A boolean visited array is **used to mark the visited vertices.** For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

How does BFS work?

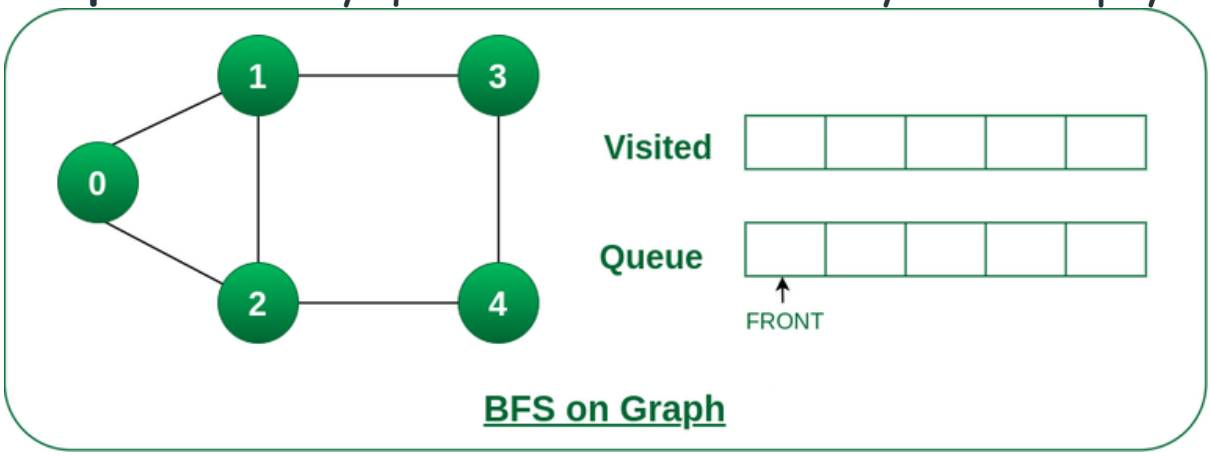
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this **a queue is used.** All the **adjacent unvisited nodes** of the current level are pushed into the queue and the **nodes of the current level are marked visited** and popped from the queue.

Illustration:

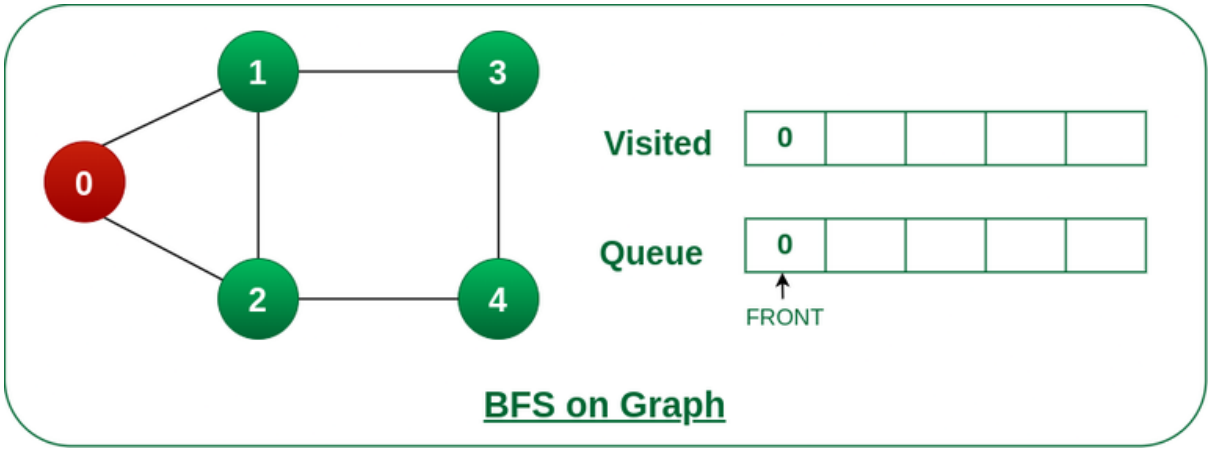
Let us understand the working of the algorithm with the help of the following example.

Step1: Initially queue and visited arrays are empty.



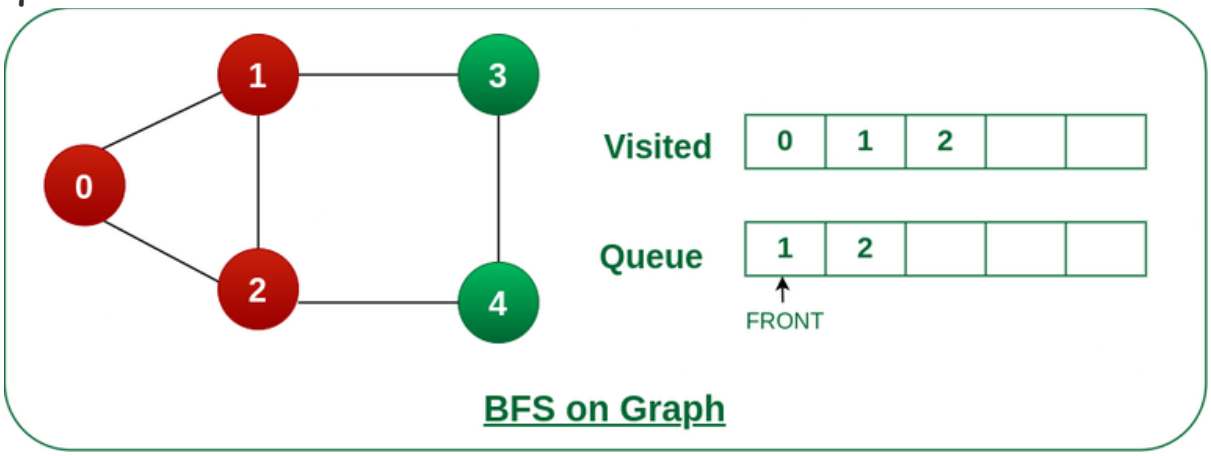
Queue and visited arrays are empty initially.

Step2: Push node 0 into queue and mark it visited.



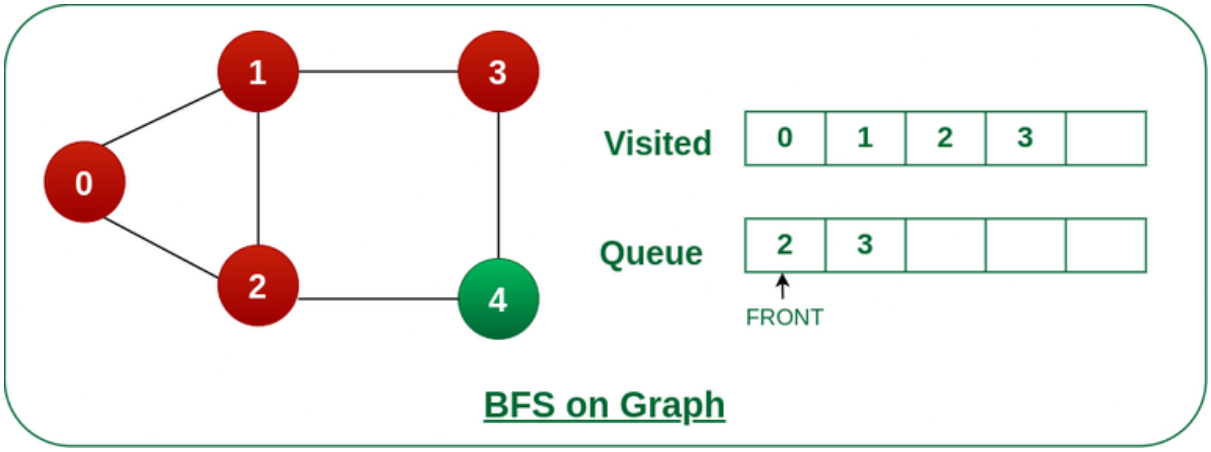
Push node 0 into queue and mark it visited.

Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



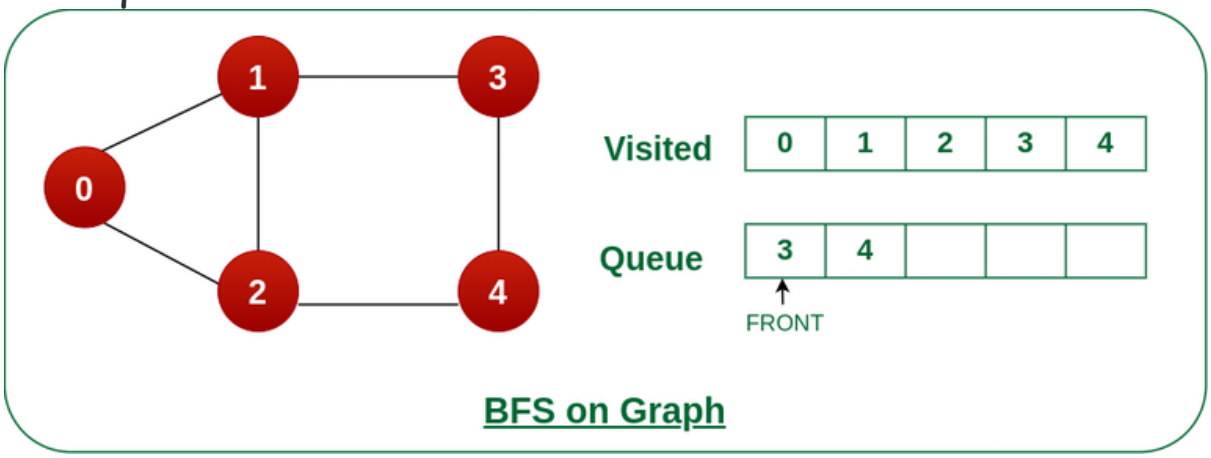
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: Remove node 1 from the front of queue and **visit the unvisited neighbours and push them into queue.**



Remove node 1 from the front of queue and visited the unvisited neighbours and push

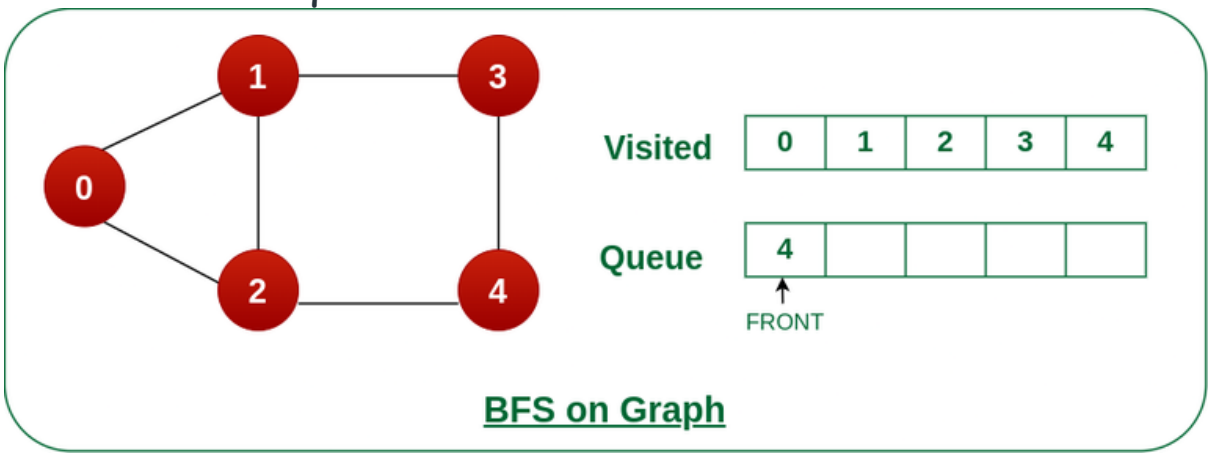
Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and **push** them into queue.



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

Step 6: Remove node 3 from the front of queue and **visit the unvisited neighbours** and **push them into queue.**

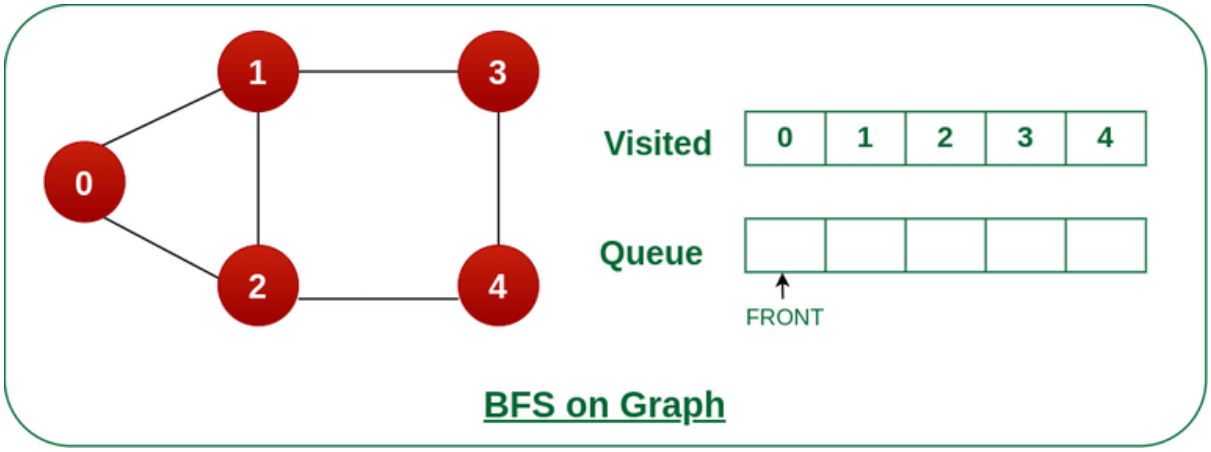
As we can see that **every neighbours of node 3 is visited**, so move to the next node that are in the front of the queue.



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

Steps 7: Remove node 4 from the front of queue and **visit the unvisited neighbours** and **push them into queue.**

As we can see that every **neighbours of node 4 are visited,** so move to the next node that is in the front of the queue.

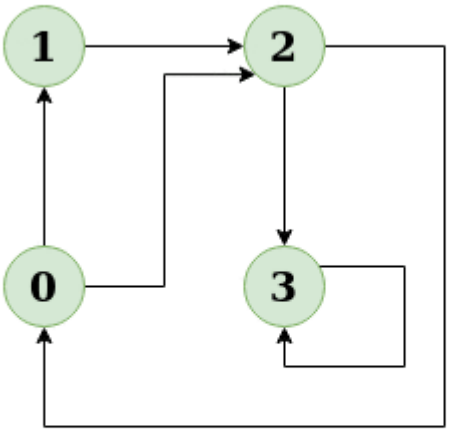


Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

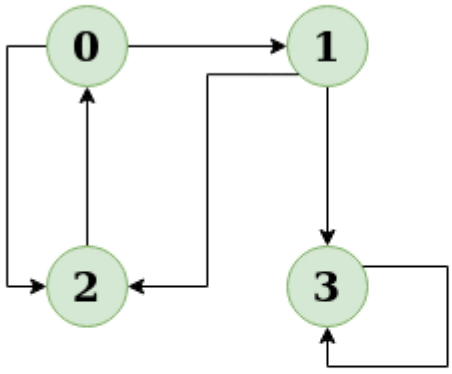
Now, Queue becomes empty, **So, terminate these process of iteration.**

Depth First Search or DFS for a Graph

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). **To avoid processing a node more than once, use a boolean visited array.** A graph can have more than one DFS traversal.



Green is unvisited node.
Red is current node.
Orange is the nodes in the recursion stack.

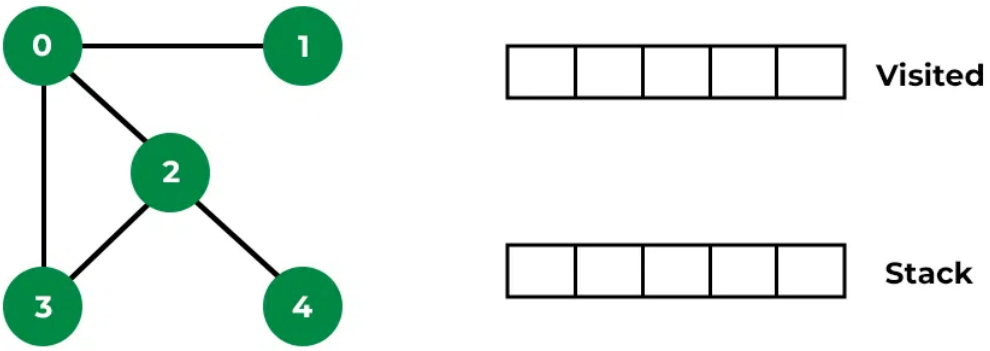


Green is unvisited node.
Red is current node.
Orange is the nodes in the recursion stack.

How does DFS work?

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The **algorithm starts at the root node** and **explores as far as possible** along each branch before backtracking.

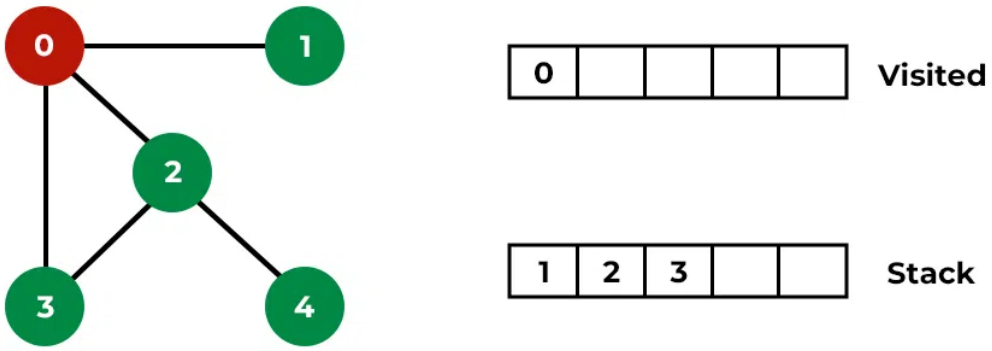
Let us understand the working of **Depth First Search** with the help of the following illustration:
Step1: Initially stack and visited arrays are empty.



DFS on Graph

Stack and visited arrays are empty initially.

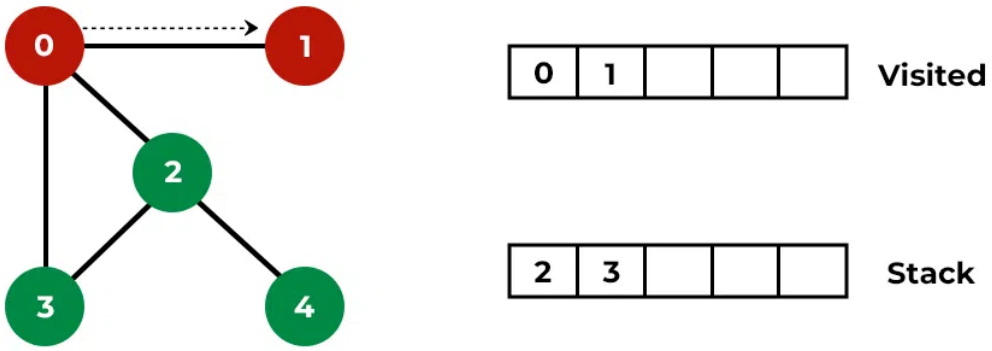
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.



DFS on Graph

Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack

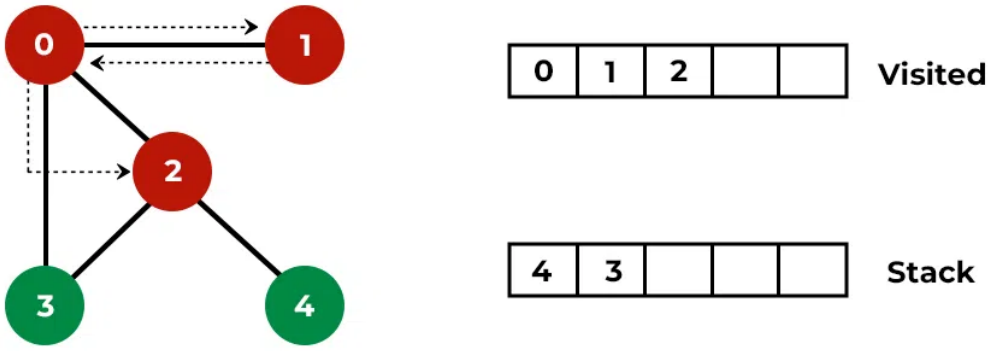
Step 3: Now, **Node 1 at the top of the stack**, so **visit node 1 and pop it from the stack** and **put all of its adjacent nodes which are not visited** in the stack.



DFS on Graph

Visit node 1

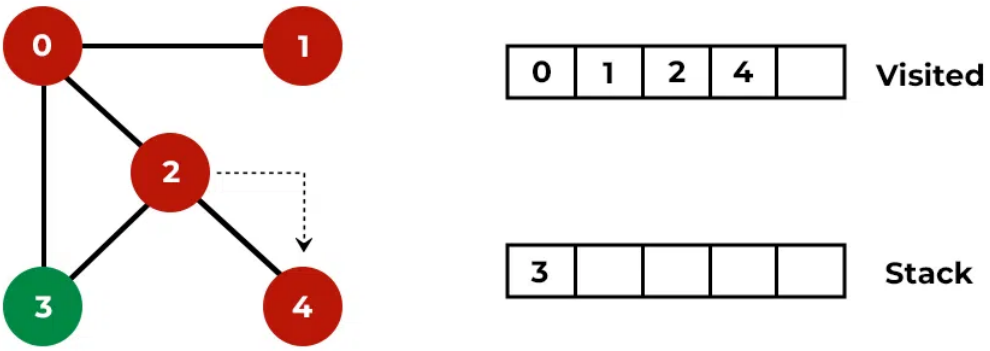
Step 4: Now, **Node 2 at the top of the stack**, so visit node 2 and pop it from the stack and p**ut all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.**



DFS on Graph

Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack

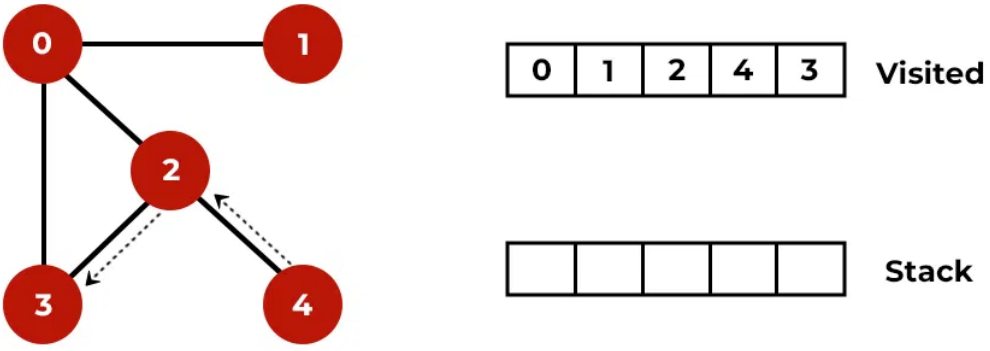
Step 5: Now, **Node 4 at the top of the stack**, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



DFS on Graph

Visit node 4

Step 6: 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.



DFS on Graph

Visit node 3

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