**GRAPHS**

* A Graph is a non-linear data structure consisting of nodes and edges.
* The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
* A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



* In the above Graph, the set of vertices V = {0, 1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.
* Graphs are used to represent networks.
* The networks may include paths in a city or telephone network or circuit network.
* Graphs are also used in social networks like linkedIn, Facebook.
* For example, in Facebook, each person is represented with a vertex(or node).
* Each node is a structure and contains information like person id, name, gender, locale etc.

**Graph and its representations**

1. A graph is a data structure that consists of the following two components:
2. A finite set of vertices also called as nodes.
3. A finite set of ordered pair of the form (u, v) called as edge.
4. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph).
5. The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v.
6. The edges may contain weight/value/cost.

https://cdncontribute.geeksforgeeks.org/wp-content/uploads/undirectedgraph.png

The following two are the most commonly used representations of a graph.   
**1. Adjacency Matrix   
2. Adjacency List**   
 **Adjacency Matrix:**

Adjacency Matrix is a 2D array of size V x 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:

Adjacency Matrix Representation https://cdncontribute.geeksforgeeks.org/wp-content/uploads/undirectedgraph.png

**Advantages**

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).

**Disadvantages**

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 lists is used.
* The size of the array is equal to the number of vertices.
* Let the array be an array[].
* An entry array[i] represents the 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 represented as lists of pairs.

Following is the adjacency list representation of the above graph. 

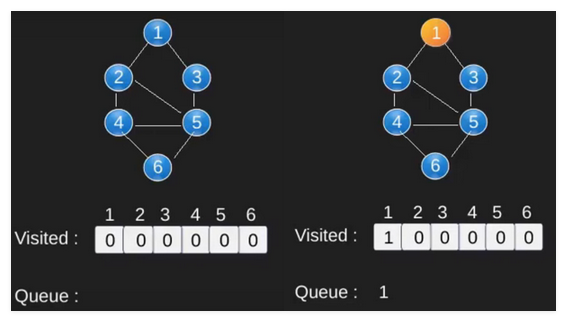
Adjacency List Representation of Graph

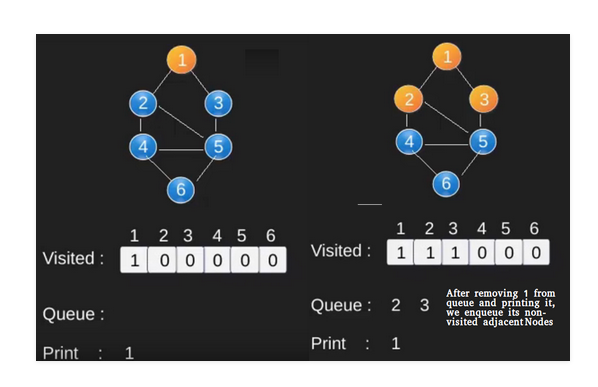
**Breadth First Search**

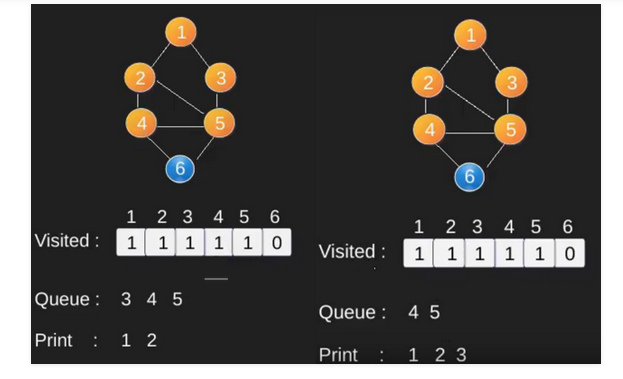
* [Breadth-First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth-First Traversal of a tree.
* The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.
* For simplicity, it is assumed that all vertices are reachable from the starting vertex.
* For example, in the following graph, we start traversal from vertex 2.
* When we come to vertex 0, we look for all adjacent vertices of it.
* 2 is also an adjacent vertex of 0.
* If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

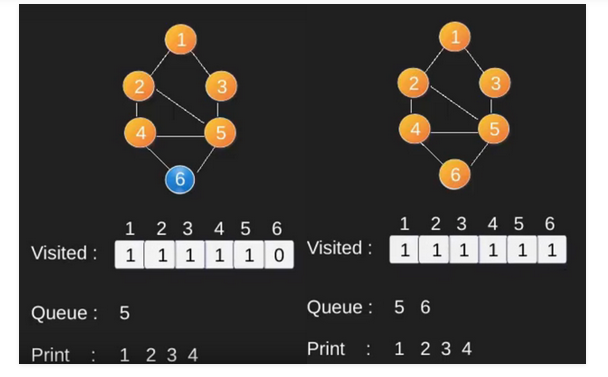
A Breadth-First Traversal of the following graph is 2, 0, 3, 1.

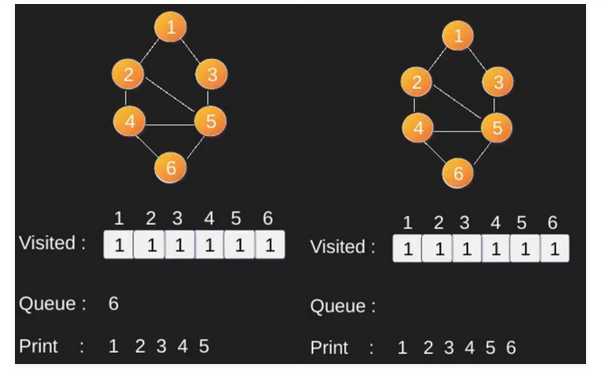












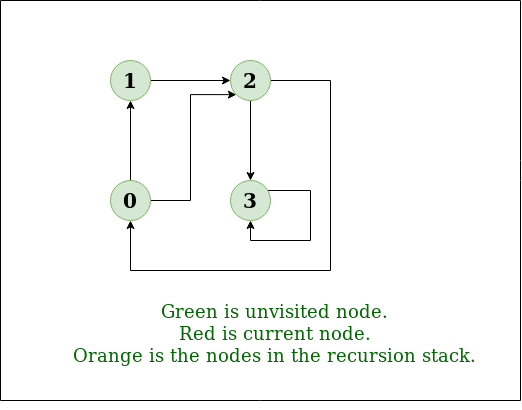
**Depth First Search**

* **Depth First Traversal** (or Search) for a graph is similar to [Depth First Traversal of a tree.](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/)
* The only catch here is, 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.

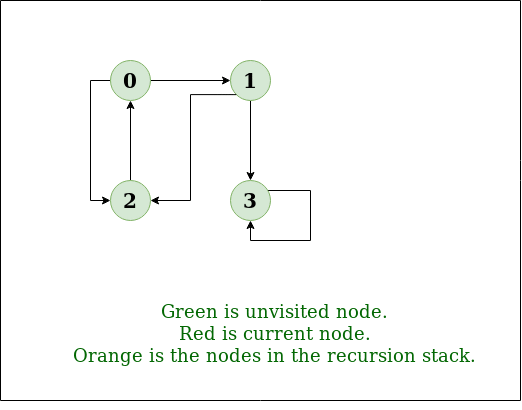
**Example:**

**Input:** n = 4, e = 6   
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3   
**Output:** DFS from vertex 1 : 1 2 0 3 

**Explanation:**   
**DFS Diagram:** 



**Input:** n = 4, e = 6   
2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3   
**Output:** DFS from vertex 2 : 2 0 1 3   
**Explanation:**

DFS Diagram:   
 

**Approach:** 

* Depth-first search is an algorithm for traversing or searching tree or graph data structures.
* The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
* So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node.
* Then backtrack and check for other unmarked nodes and traverse them.
* Finally, print the nodes in the path.

**Algorithm:**

Create a recursive function that takes the index of the node and a visited array.

1. Mark the current node as visited and print the node.
2. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.