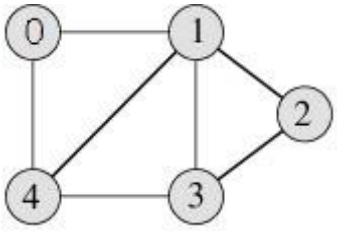
# GRAPH

## Graph and its representations:

Graphs are used to represent many real life applications: 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 and locale.

Following is an example undirected graph with 5 vertices.




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 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.

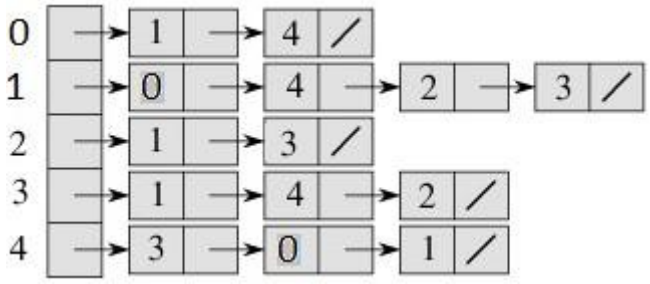
The adjacency matrix for the above example graph is:

| | 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.

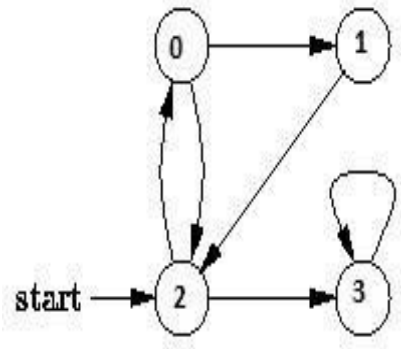



Adjacency List Representation of the above Graph

**Breadth First Traversal for a Graph(BFS)**

Breadth First Traversal (or 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 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. Breadth First Traversal of the following graph is 2, 0, 3, 1.



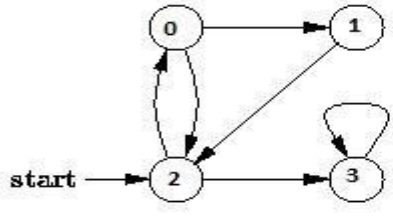
**Algorithm: Breadth-First Search**

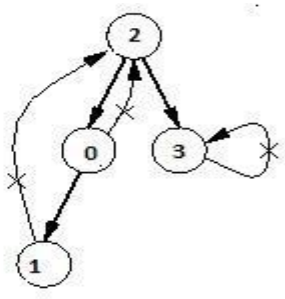
**Traversal BFS(V, E, *s*)**

```
 for each u in V − {s}
     do color[u] ←
        WHITE d[u]
        ← infinity
        π[u] ← NIL
        color[s] ←
        GRAY
       d[s]
      ← 0
      π[s]
      ←
      NIL Q
      ← {}
   ENQUEUE(Q, s)
   while Q is non-empty
do u ← DEQUEUE(Q)
         for each v adjacent to u
             do if color[v] ← WHITE
                then color[v] ←
                     GRAY d[v]
                     ← d[u] + 1
                       π[v] ← u
                       ENQUEU
           E(Q, v) DEQUEUE(Q)
       color[u] ← BLACK
```

**Depth First Traversal for a Graph(DFS)**

Depth First Traversal (or Search) for a graph is similar to Depth 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 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. Depth First Traversal of the following graph is 2, 0, 1, 3.



**Algorithm Depth-First Search**

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges $(u, v)$ such that $u$ is gray and $v$ is white when edge $(u, v)$ is explored. The following pseudocode for DFS uses a global timestamp time.

**DFS (V, E)**

```
 for each vertex u in V[G]
do color[u] ← WHITE π[u] ← NIL
    time ← 0
 for each vertex u in V[G]
   do if color[u] ← WHITE
        then DFS-Visit(u)

DFS-Visit(u)
   color[u] ← GRAY
   time ← time + 1 d[u] ← time
   for each vertex v adjacent to u
 do if color[v] ← WHITE
        then π[v] ← u
                DFS-
   Visit(v) color[u]
   ← BLACK time
   ← time + 1
   f[u] ← time
```



*"The only way to learn a new programming language is by writing programs in it."*

**With Regards**
**Yours Sam Sir**