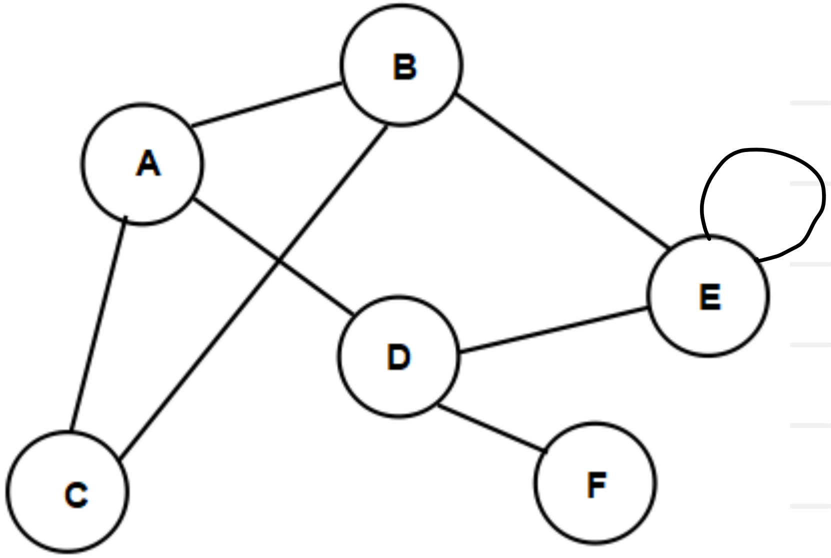
# Graph : Terminologies

- **Graph** is a non linear data structure having set of vertices (nodes) and set of edges (arcs).
  - G = {V, E}

    Where V is a set of vertices and E is a set of edges
  - **Vertex (node)** is an element in the graph

    V = {A, B, C, D, E, F}
  - **Edge (arc)** is a line connecting two vertices

    E = {(A,B), (A,C), (B,C), (B,E), (D, E), (D,F),(A,D)}

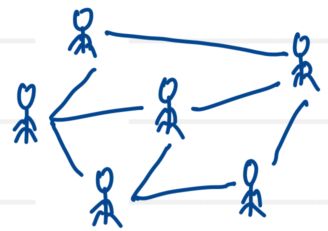


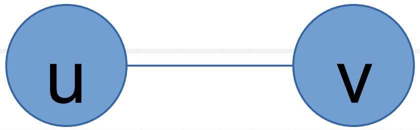

- Vertex A is set be **adjacent** to B, if and only if there is an edge from A to B.
- **Degree of vertex :-** Number of vertices adjacent to given vertex
- **Path :-** Set of edges connecting any two vertices is called as path between those two vertices.
  - Path between A to D = {(A, B), (B, E), (E, D)}
- **Cycle :-** Set of edges connecting to a node itself is called as cycle.
  - {(A, B), (B, E), (E, D), (D, A)}
- **Loop :-** An edge connecting a node to itself is called as loop. Loop is smallest cycle.

# Graph : Types

- **Undirected graph.**
  - If we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**
  - **graph which contains undirected edges referred as undirected graph.**

Degree : no. of edges



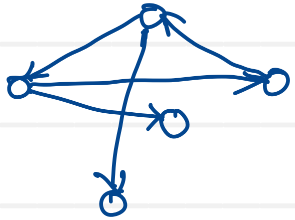


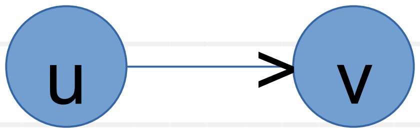

(u, v) == (v, u)

- **Directed Graph (Di-graph)**
  - If we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **ordered pair of vertices** i.e. directed edge.
  - **graph which contains set of directed edges referred as directed graph (di-graph).**
  - graph in which each edge has some direction

in degree : no. of edges coming to vertex

out degree : no. of edges going from vertex

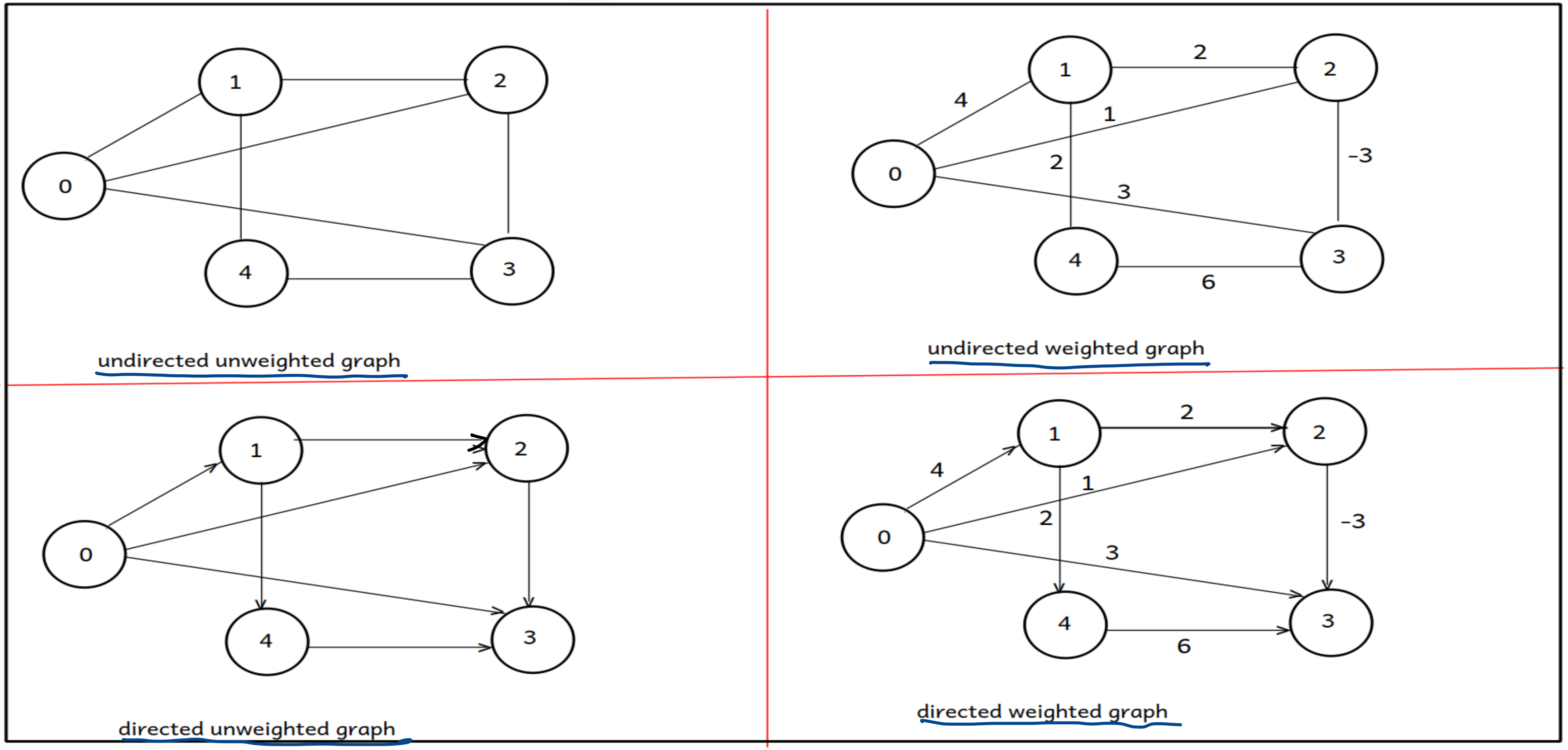





(u, v) != (v, u)

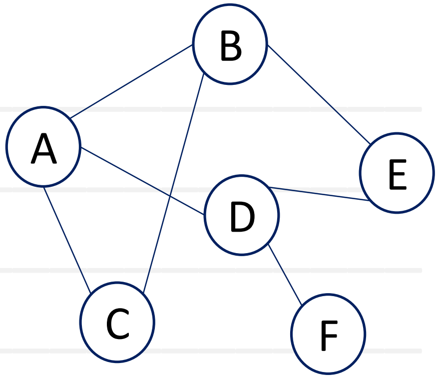
# Graph : Types

- **Weighted Graph**
  - A graph in which edge is associated with a number (ie weight)



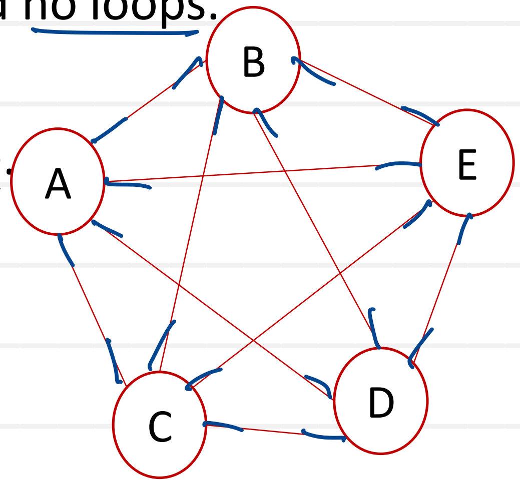
# Graph : Types

- **Simple Graph**
  - Graph not having multiple edges between adjacent nodes and no loops.

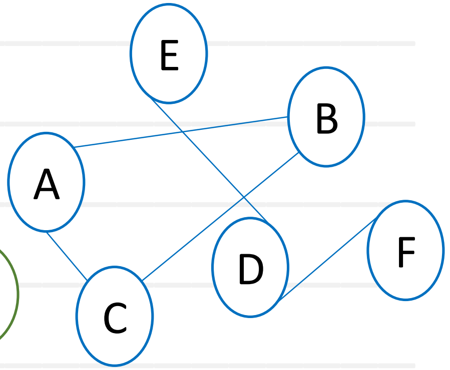



- **Complete Graph**
  - Simple graph in which node is adjacent with every other node.
  - Un-Directed graph: Number of Edges = n (n -1 ) / 2

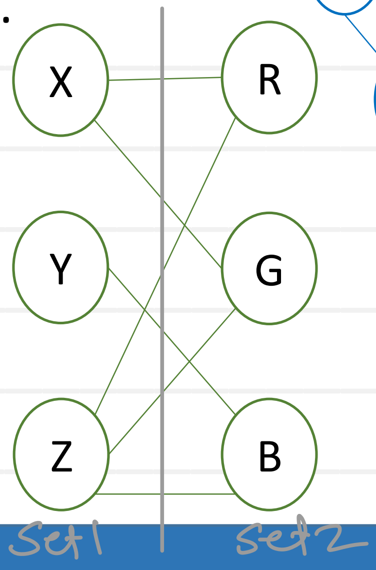
    where, n – number of vertices
  - Directed graph: Number of edges = n (n-1)




- **Connected Graph**
  - Simple graph in which there is some path exist between any two vertices.
  - Can traverse the entire graph starting from any vertex.

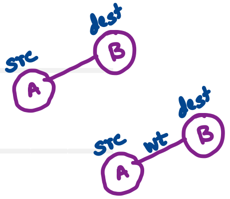


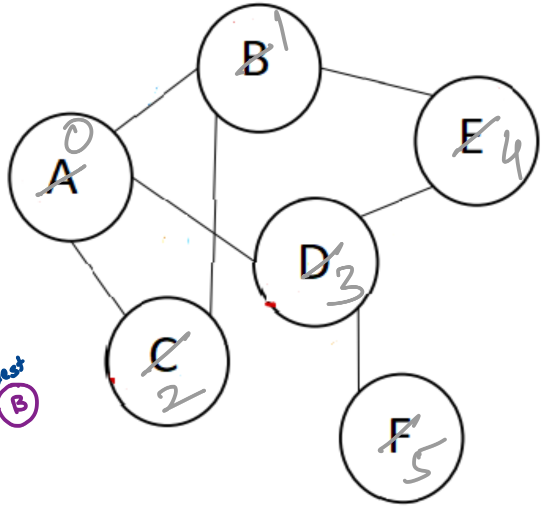

- **Bi-partite graph**
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.

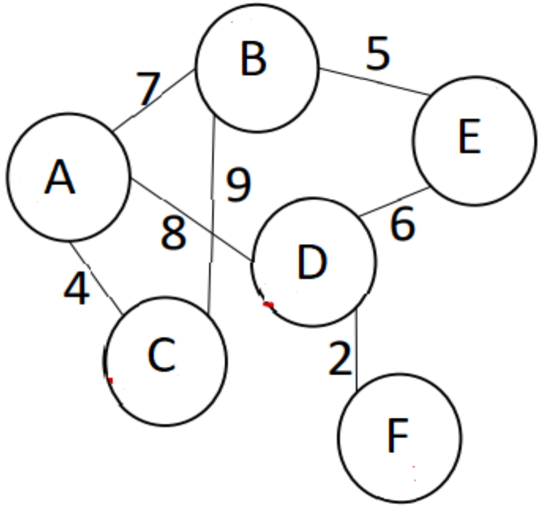


# Graph Implementation – Adjacency Matrix

- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For non-weighted graph, 1 indicate edge and 0 indicate no edge.
- For weighted graph, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is O(V^2).



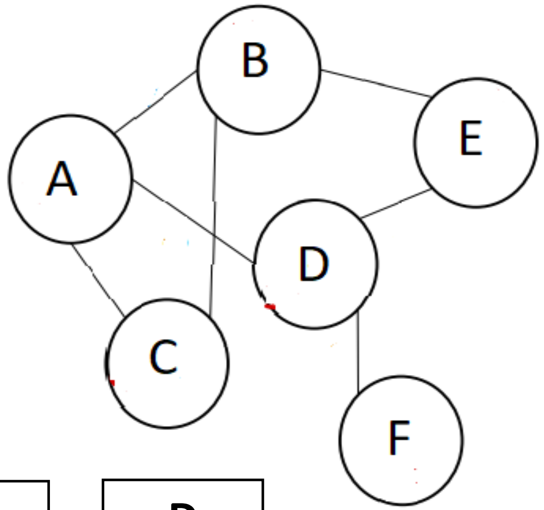



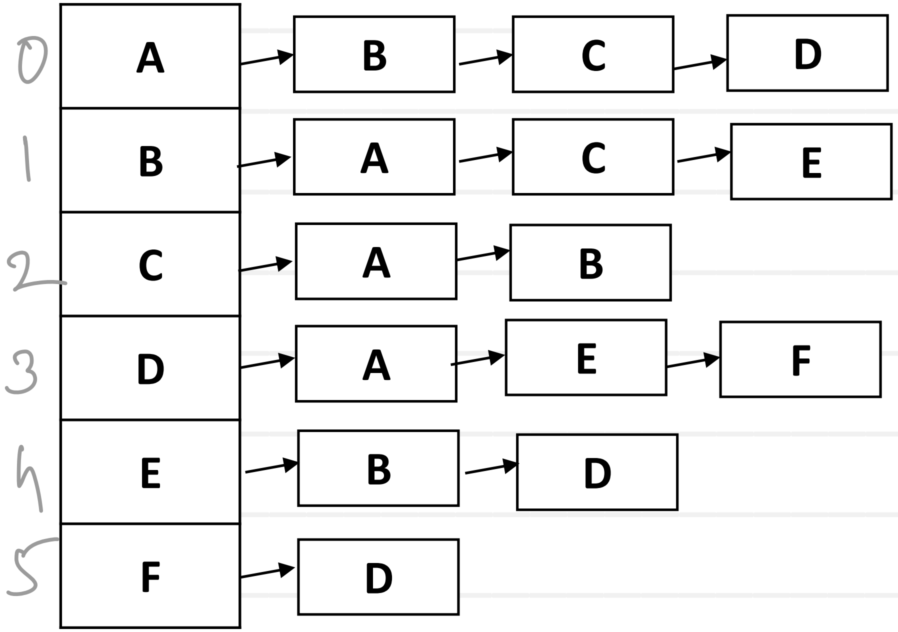

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 0 | 0 |
| B | 1 | 0 | 1 | 0 | 1 | 0 |
| C | 1 | 1 | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 0 | 0 | 1 | 1 |
| E | 0 | 1 | 0 | 1 | 0 | 0 |
| F | 0 | 0 | 0 | 1 | 0 | 0 |

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | ∞ | 7 | 4 | 8 | ∞ | ∞ |
| B | 7 | ∞ | 9 | ∞ | 5 | ∞ |
| C | 4 | 9 | ∞ | ∞ | ∞ | ∞ |
| D | 8 | ∞ | ∞ | ∞ | 6 | 2 |
| E | ∞ | 5 | ∞ | 6 | ∞ | ∞ |
| F | ∞ | ∞ | ∞ | 2 | ∞ | ∞ |

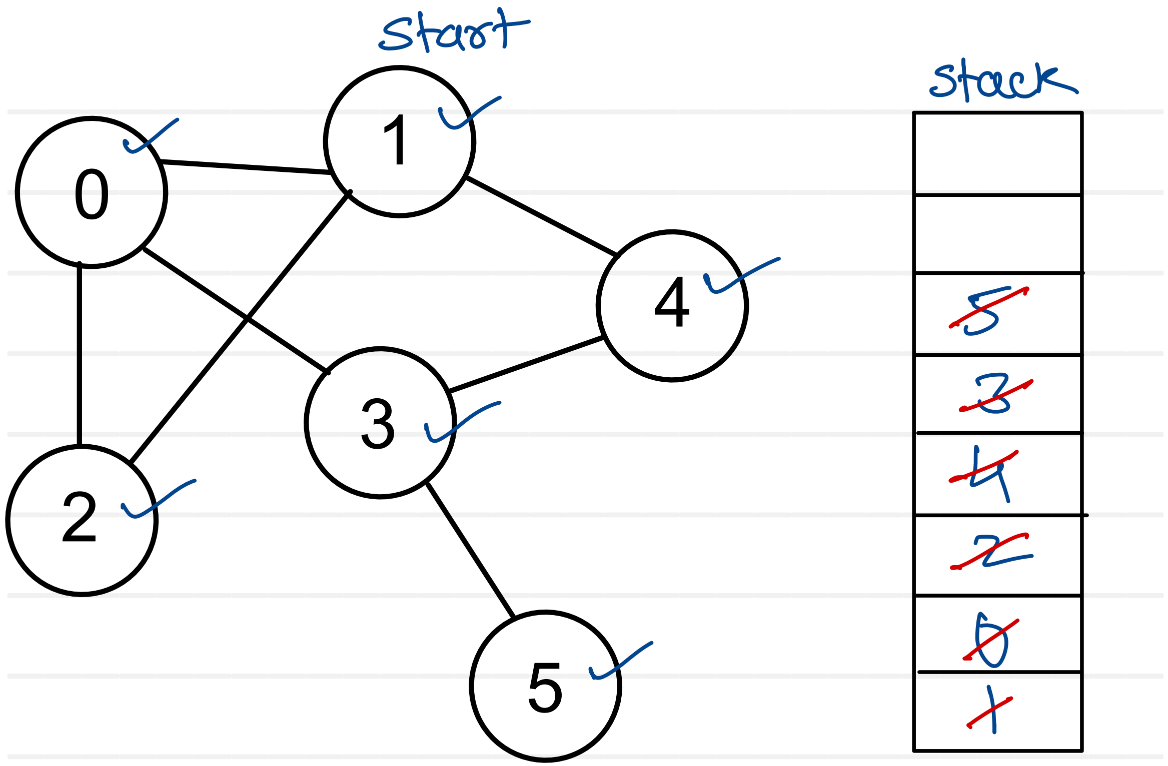
# Graph Implementation – Adjacency List

- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbor vertices are stored.
- For weighted graph, neighbor vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).



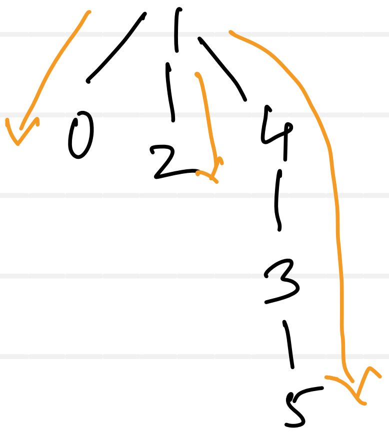



# DFS Traversal



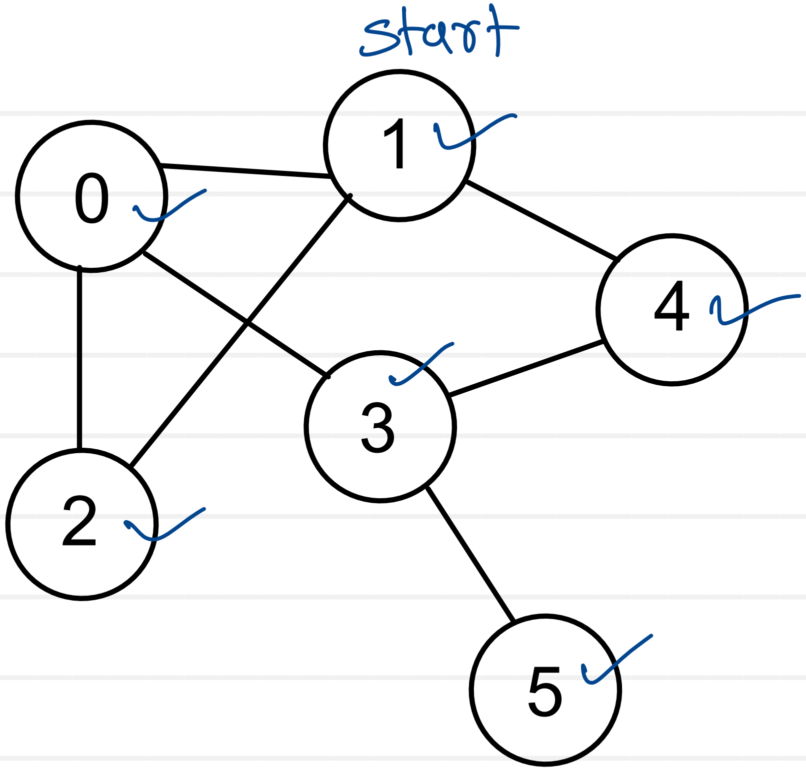

Traversal : 1, 4, 3, 5, 2, 0

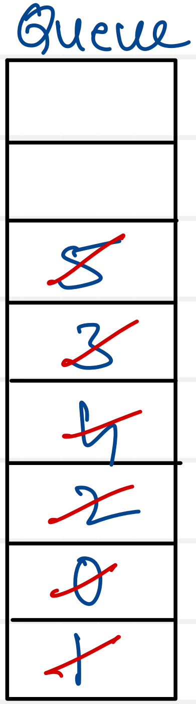
1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Print the vertex.
5. Put all non-visited (marked) neighbours (adjacent) of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.




$\boxed{T(V) = O(V^2)}$

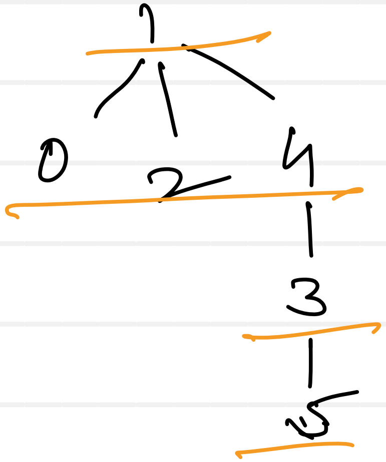
# BFS Traversal



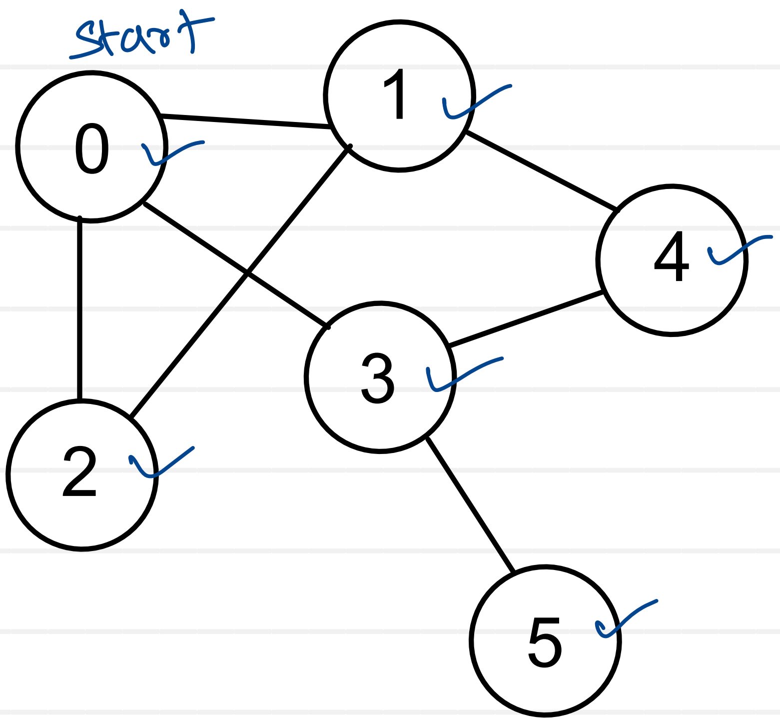

1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it
3. Pop vertex from queue.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.

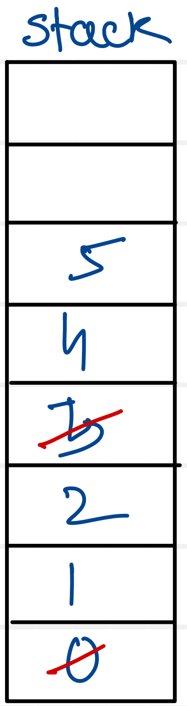
Traversal : 1, 0, 2, 4, 3, 5

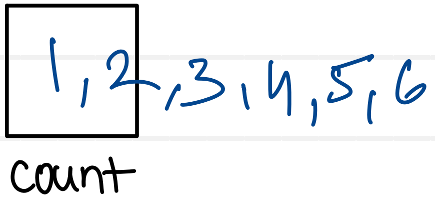



$\boxed{T(V) = O(V^2)}$

# Check connected ness

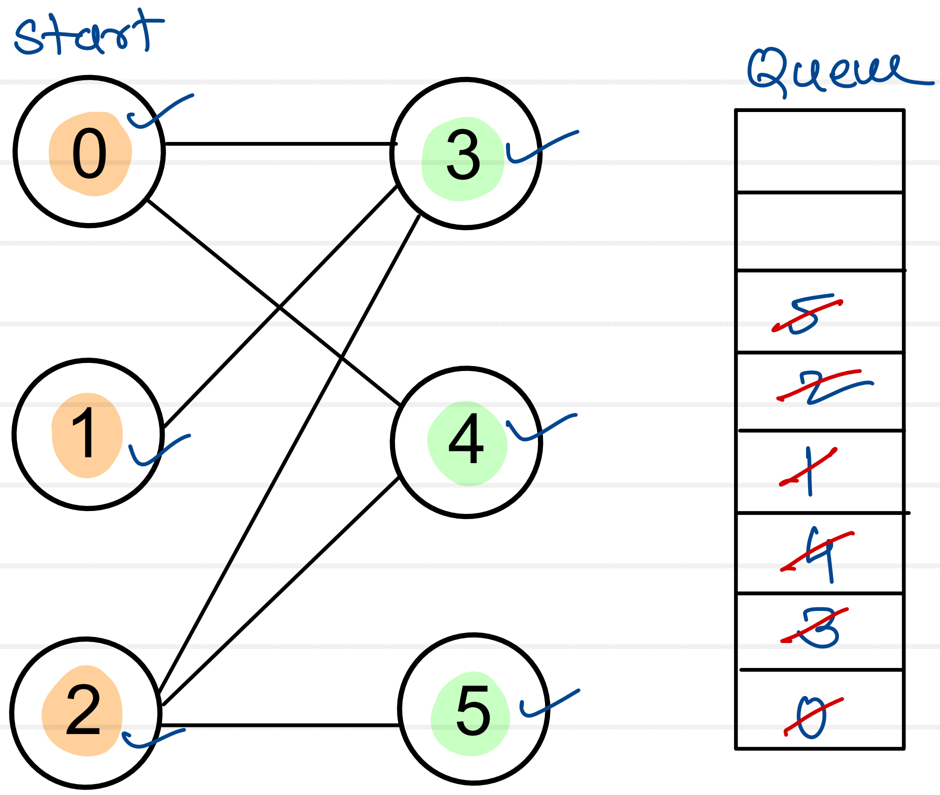



1. Push start vertex on stack and mark it.
2. Begin counting marked vertices from 1.
3. Pop and print a vertex.
4. Push all its non marked neighbours on stack, mark them and increment count.
5. If count is same as number of vertices, graph is connected (return).
6. Repeat steps 3 to 5 until stack is empty.
7. Graph is not connected (return).

# Check bipartite ness




1. Keep colours of all vertices in an array. Initially vertices have no colour.
2. Push start on queue and mark it. Assign it colour 1.
3. Pop the vertex.
4. Push all its neighbours on the queue.
5. For each such vertex if no colour is assigned yet, assign opposite colour of current vertex ( c1-c2, c2-c1).
6. If vertex is already coloured with same of current vertex, graph is not bipartite.
7. Repeat steps 3-6 until queue is empty.

colour 1 = +1

No colour = 0

colour 2 = -1

v = +1 * -1 = -1

v = -1 * -1 = +1

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com