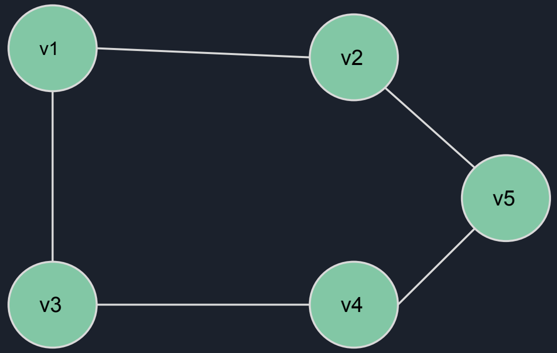
# GRAPH

*By* **Harsh Prakash**

# Importance of Graph

# TREE VS GRAPH

# GRAPH

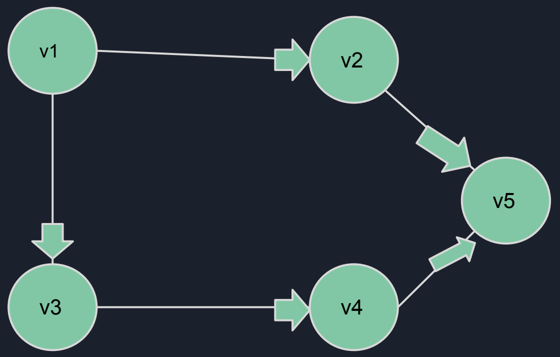


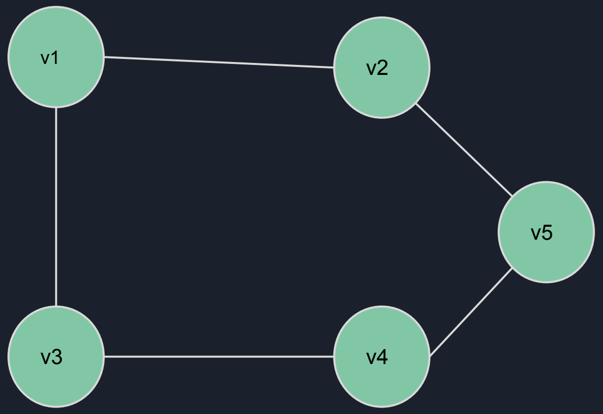

G = (V,E) (Vertices and Edges)

V = {V1, V2, V3 , V4, V5, V6}

E = {(V1, V2), (V1, V3), (V2, V4), (V3,V4), (V3,V5)}

# DIRECTED VS UNDIRECTED GRAPH



# DIRECTED VS UNDIRECTED GRAPH

- Directed Graph has Indegree and Outdegree for its every node , whereas undirected graph has just degree for its node.
- Degree is the number of edges passing through a node.
- Indegree is the number of incoming edges and outdegree is the number of outgoing edges from a node in directed graph.
- Sum of Indegrees and Outdegrees respectively are both equal to number of edges;
- Sum of degree in an undirected graph is twice the number of edges;

# DIRECTED VS UNDIRECTED GRAPH

- Undirected Graph are bi-directional whereas directed graph are one-direction(they represent a single direction of flow in a relationship between two graph nodes).

- In Directed Graph (V1, V2) is an ordered pair that means (V1,V2) is not similar/equal to (V2,V1), whereas in Undirected Graph (V1,V2) is similar to (V2,V1)

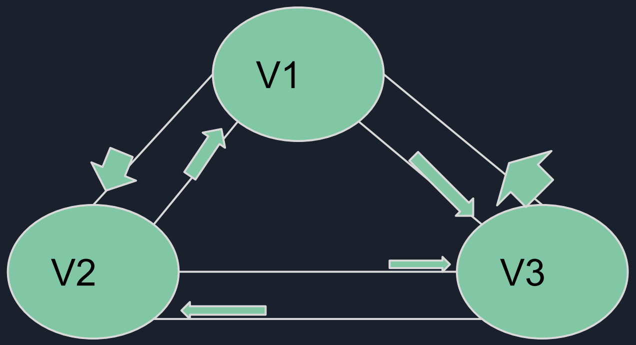
- Example of Directed Graph cabe World  Wide Web and example of Undirected Graph can be a Social Network of Friends and Relatives.

# DIRECTED GRAPH VS UNDIRECTED GRAPH

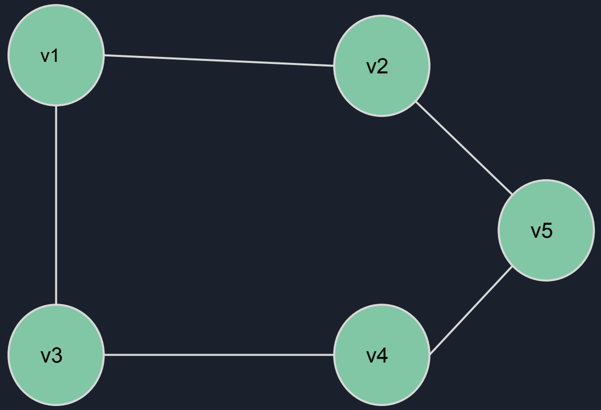
Max Number of Edges in Directed Graph (|V| * (|v|-1)), where V is number of vertices

Such a graph is known as **COMPLETE GRAPH.**

Max Number of Edges in Undirected Graph (|V| * (|v|-1))/2;



# WALK AND PATH




**WALK** : V1, V2, V5,v4

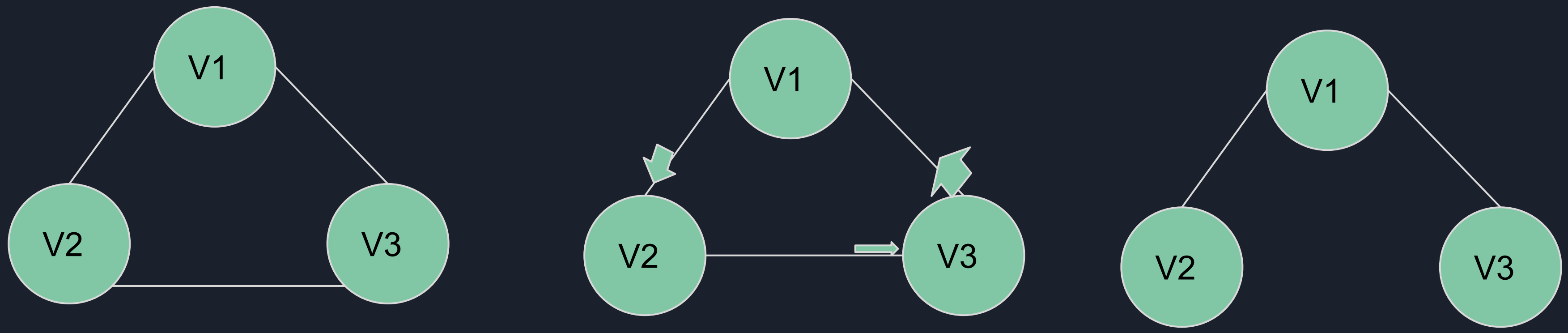
**PATH** : V1, V2 , V5 (no repetition allowed

**CYCLIC :** There exists a walk that begins and ends with the same vertex.

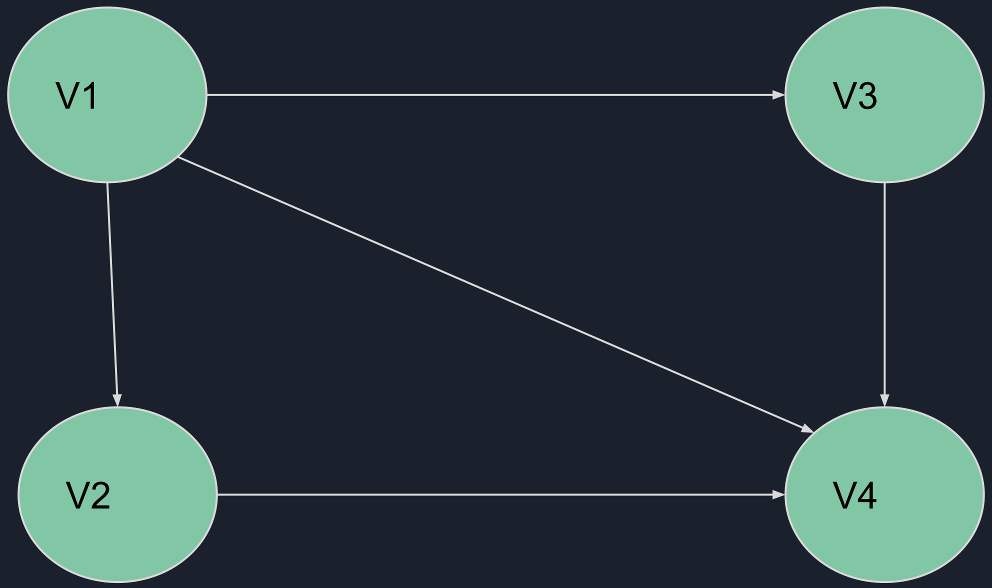
Some textbooks name “walk “as “path” and “path” as

“Simple path”

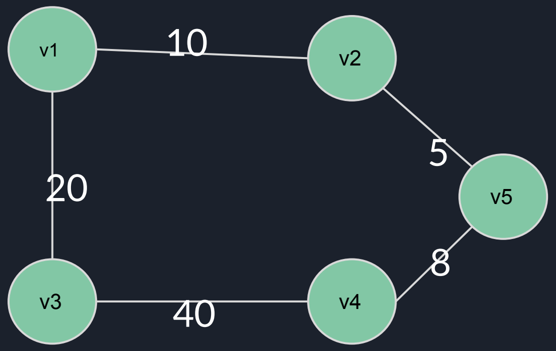
# NAME THE TYPES OF GRAPHS

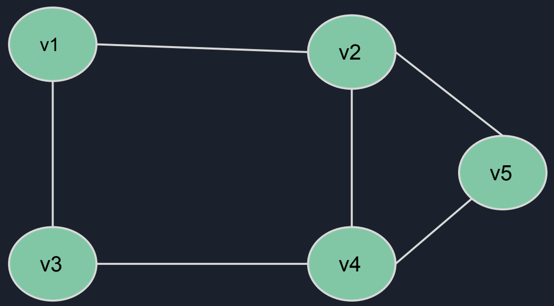


# Name the type of this Graph



# Weighted and Unweighted Graphs




**Weighted Graphs:**

Contains some magnitude on the edges

Example => Google Maps or any other Navigational Service

**Unweighted Graphs:**

**Example =>Social Media Network of Friends/Colleagues.**

# GRAPH REPRESENTATION

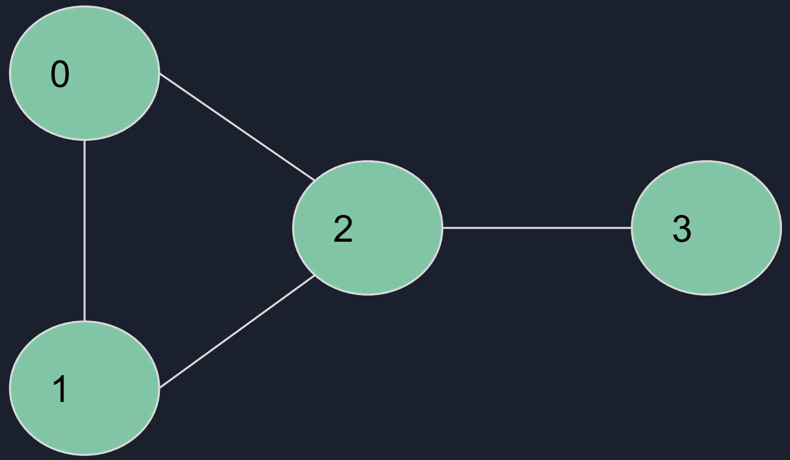
# GRAPH REPRESENTATION

GRAPH CAN BE REPRESENTED BY MANY FORMS :

MOSTLY IT IS IMPLEMENTED BY TWO POPULAR METHODS -

1. **ADJACENCY MATRIX (**for directed and undirected graph)
2. **ADJACENCY LIST** (for directed and undirected graph**)**

# ADJACENCY MATRIX (undirected graph)




| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 |

# ADJACENCY MATRIX (undirected graph)

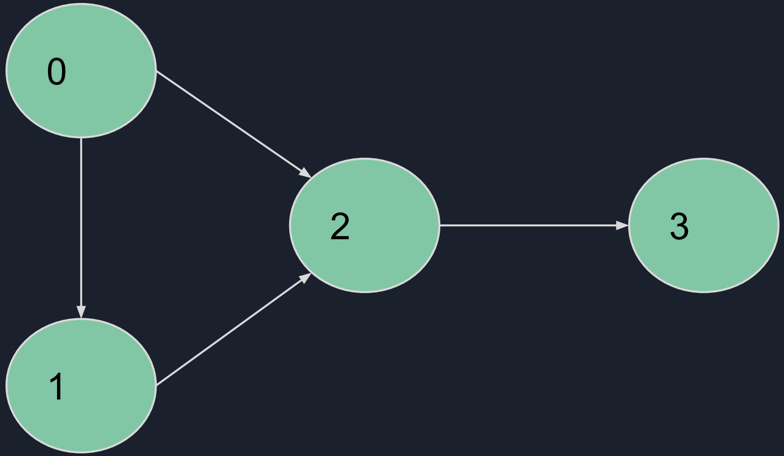
Size of Matrix = |V| * |V|

For undirected Graph =>

It is a symmetric matrix

Mat[i][j] = | 1 - if there exists a edge from vertex i to vertex j
| 0 - if there is no mapping between the two vertices. |

# ADJACENCY MATRIX (directed graph)




| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 |
| 3 | 0 | 0 | 0 | 0 |

# ADJACENCY MATRIX (directed graph)

Size of Matrix = |V| * |V|

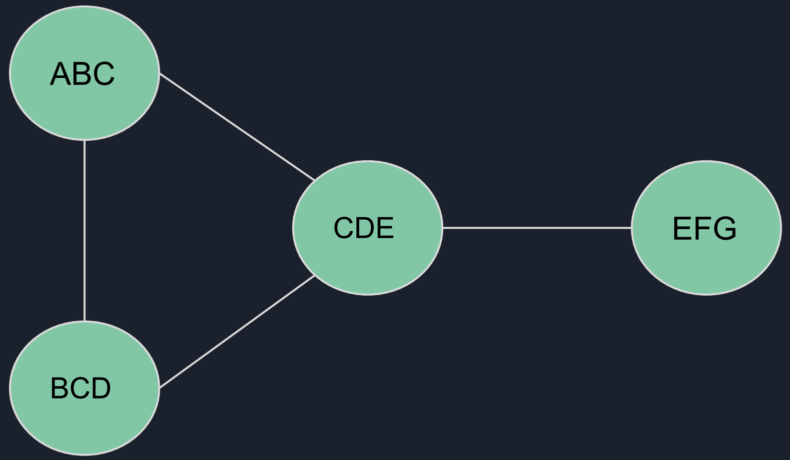
For directed Graph =>

It may be a symmetric matrix or not.

Mat[i][j] = | 1 - if there exists a outgoing edge from vertex i to vertex j |

| 0 - if there is no outgoing edge from i to j. |

# HOW TO HANDLE VERTICES WITH ARBITRARY NAMES ?

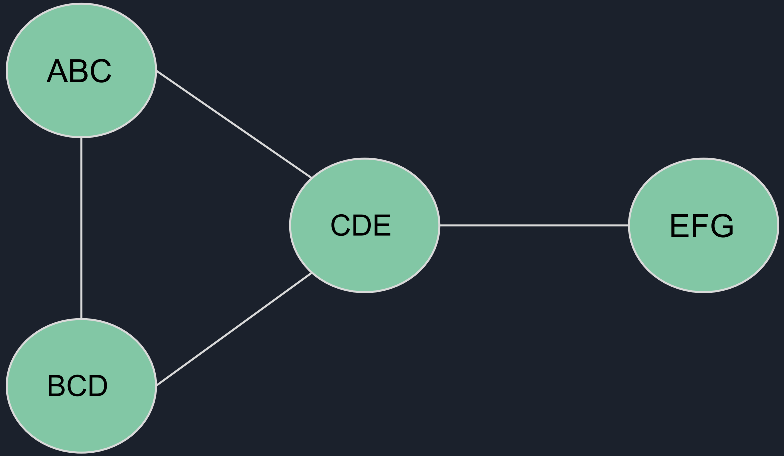



Additional DS is required for strings.

(e.g. we can take an array)

| | |
|---|---|
| 0 | ABC |
| 1 | BCD |
| 2 | CDE |
| 3 | EFG |

# HOW TO HANDLE VERTICES WITH ARBITRARY NAMES ?




| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 |

# HOW TO HANDLE VERTICES WITH ARBITRARY NAMES ?

For Efficient implementation one hash Table(h) would also be required to do “ **REVERSE MAPPING** “.

- **h {ABC} = 0**
- **h {BCD} = 1**
- **h {CDE} = 2**
- **h {EFG} = 3**

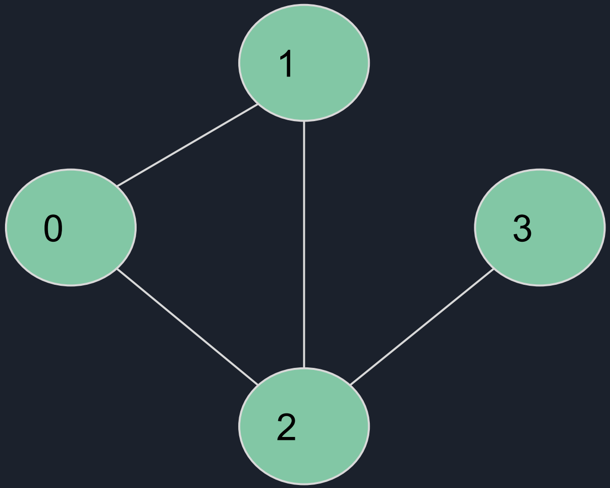
# Properties of Adjacency Matrix

**SPACE REQUIRED : O(V*V)**

**OPERATIONS :**

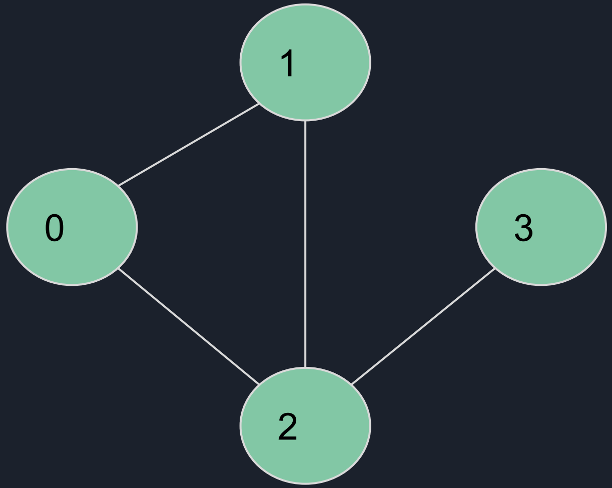
- Check if u and v are adjacent to each other : O(1)
- Find all vertices adjacent to u : O(V)
- Find all degree of u : O(V)
- Add/Remove an EDGE : O(1)
- Add/Remove a Vertex : O(V*V) -> O(V^2)

# Adjacency List



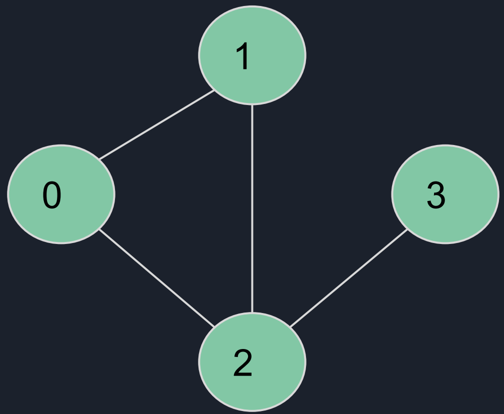

Adjacency Matrix stores redundant data, so the need of Adjacency List arises.

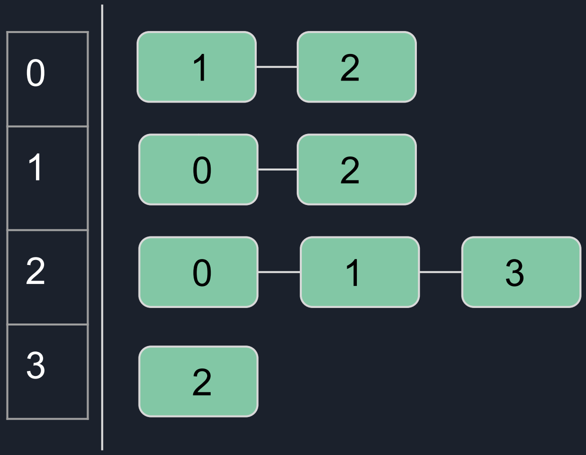
# Adjacency List (for undirected graph)




| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 |

# Adjacency List




L.L / Vectors

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 |

Matrix

# Adjacency List for undirected Graph

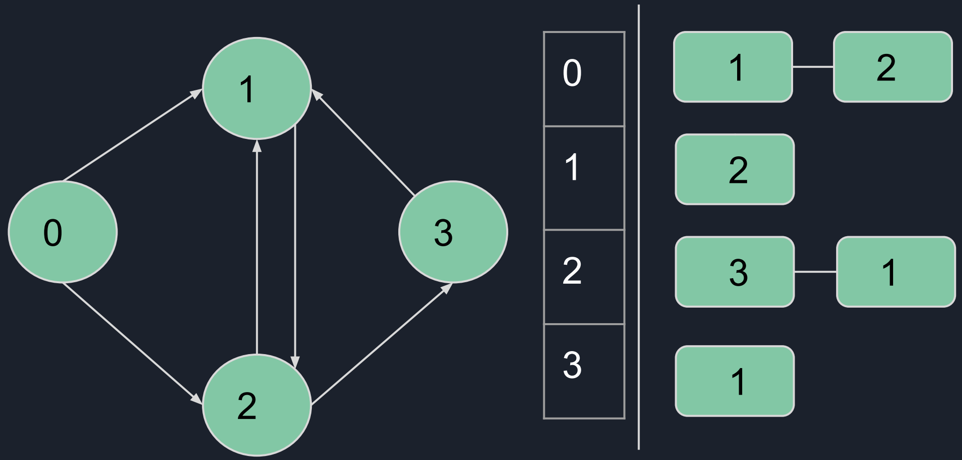
An array of lists , where lists are mostly represented as :

1. **DYNAMIC SIZED ARRAYS**
2. **LINKED LISTS**

# Operations in Adjacency List

- **Check if there is an edge from u to v : O(V)**
- **Find all adjacent of u : theta(degree(u))**
- **Find degree of u : theta(1)**
- **Add an Edge : theta(1)**
- **Remove an Edge : theta(V)**

# Adjacency List for directed graph



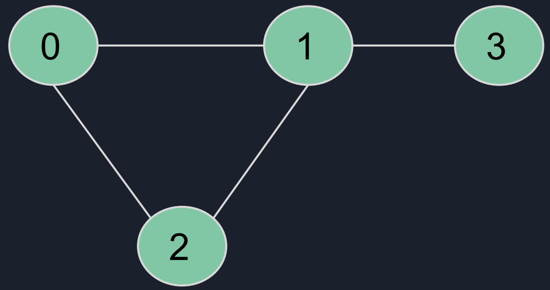

**Space** =>

O(V + E) - Undirected Graph
O(V+ 2E) - Directed Graph

# Adjacency List Implementation

```
Class Test{
Static void addEdge(ArrayList<ArrayList<Integer>> adj, int u , int v){
        adj.get(u).add(v);
        adj.get(v).add(u);
}
Public static void main(String [] args){
    Int v = 5;
    ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(v);
    For (int i  = 0 ; i  < v ; i++){
        adj.add(new ArrayList<Integer>());
        }
        addEdge(adj, 0 , 1);
        addEdge(adj, 0, 2);
        addEdge(adj, 1 , 2);
        addEdge(adj, 1, 3);


}
```



# Comparison of Adjacent Matrix vs List

| PARAMETERS | LIST | MATRIX |
|---|---|---|
| 1 - MEMORY | O(V+E) | O(V*V) |
| 2- Check edge between U to V | O(V) | theta(1) |
| 3 - Find all adjacent to u | theta(degree(u)) | theta(V) |
| 4 - Add an edge | theta(1) | theta(1) |
| 5 - Remove an edge | O(V) | O(1) |

THEORY OF **GRAPH**
ENDS HERE.