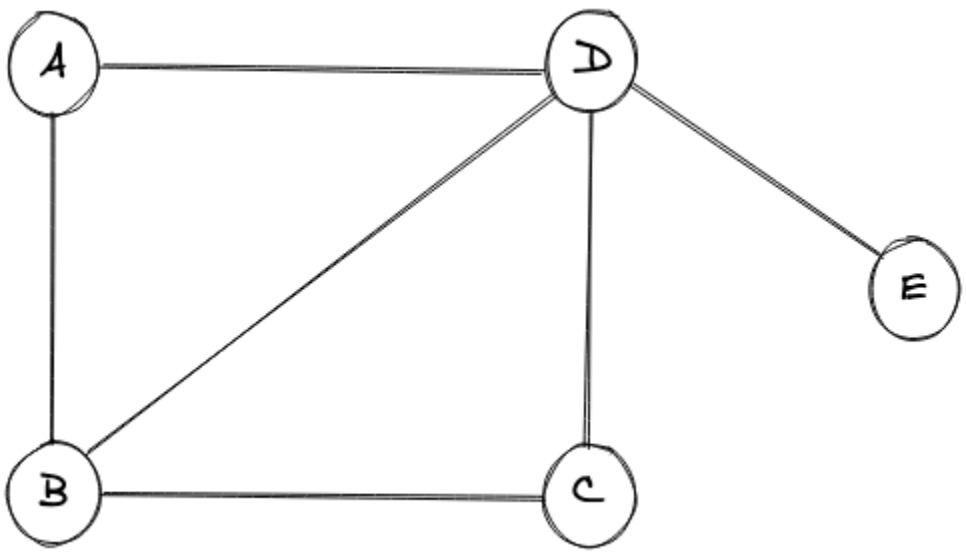
Introduction to Graphs

A **graph is an advanced data structure** that is used to organize items in an **interconnected network**. Each item in a graph is known as a **node**(or **vertex**) and these nodes are connected by **edges**.

In the figure below, we have a simple graph where there are five nodes in total and six edges.

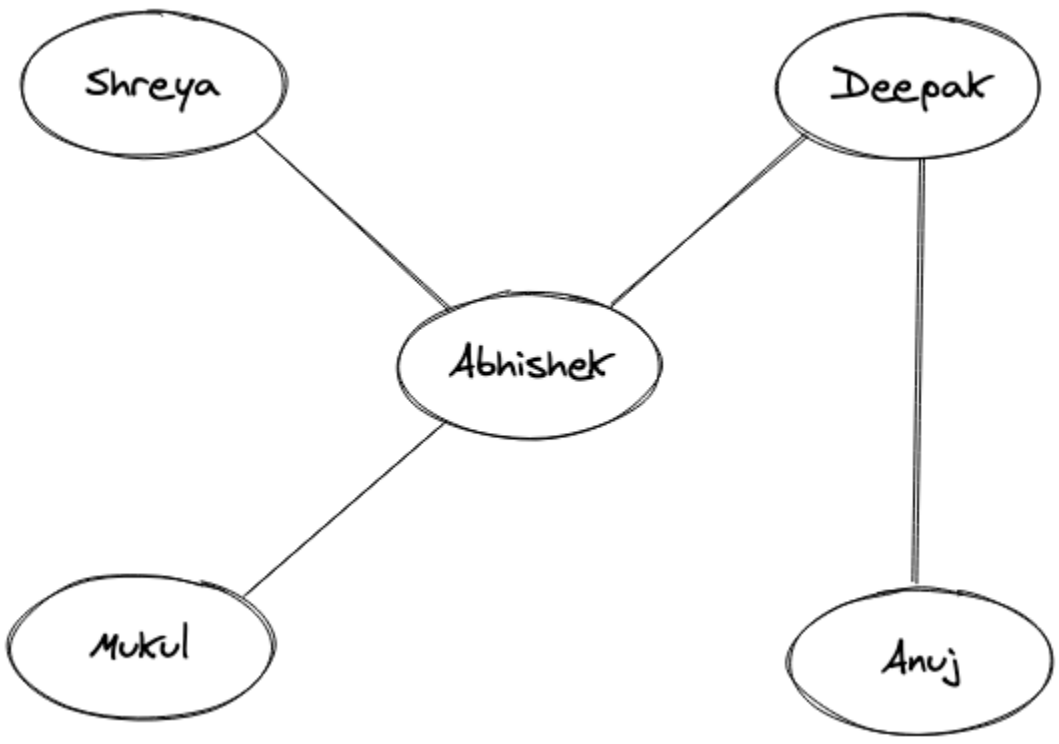


The nodes in any graph can be referred to as **entities** and the edges that connect different nodes define the **relationships between these entities**. In the above graph we have a set of nodes **{V} = {A, B, C, D, E}** and a set of edges, **{E} = {A-B, A-D, B-C, B-D, C-D, D-E}** respectively.

Real-World Example

A very good example of graphs is a **network of socially connected people**, connected by a simple connection which is whether they know each other or not.

Consider the figure below, where a pictorial representation of a social network is shown, in which there are five people in total.



A line in the above representation between two people mean that they know each other. If there's no line in between the names, then they simply don't know each other. The names here are equivalent to the nodes of a graph and the lines that define the relationship of "knowing each other" is simply the equivalent of an edge of a graph. It should also be noted that the relationship of knowing each other goes both ways like "Abhishek" knows "Mukul" and "Mukul" knows "Abhishek".

The social network depicted above is nothing but a graph.

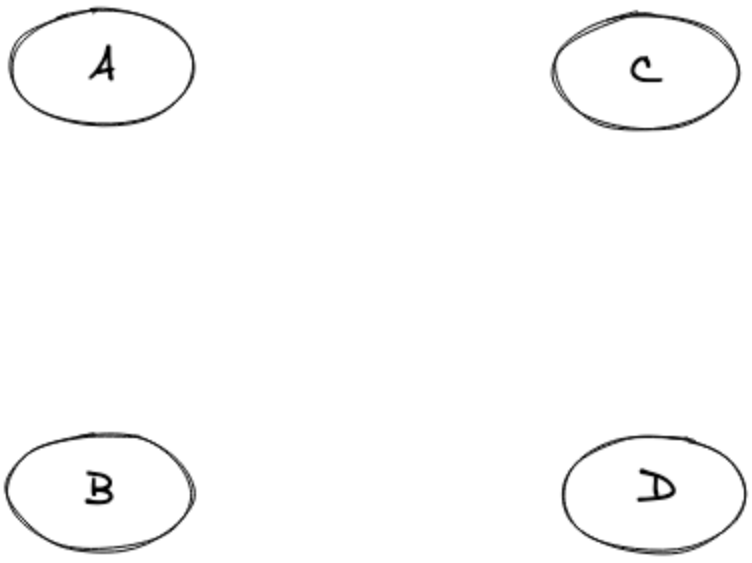
Types of Graphs

Let's cover various different types of graphs.

1. Null Graphs

A graph is said to be null if there are no edges in that graph.

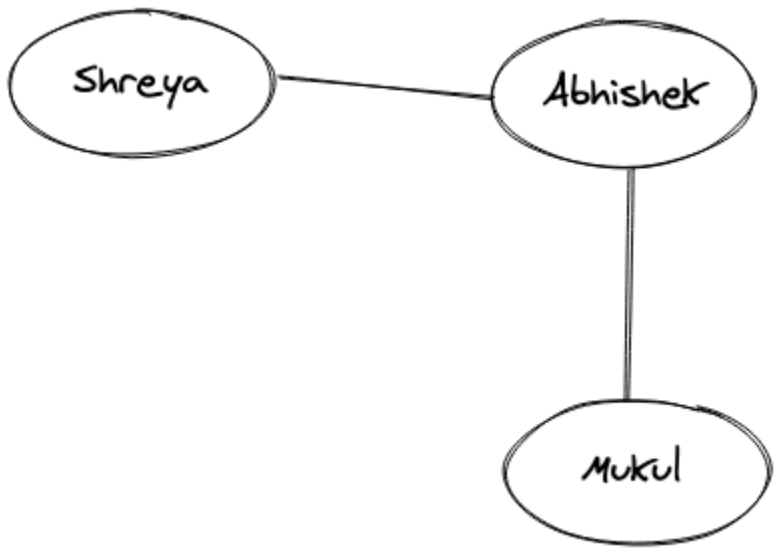
A pictorial representation of the null graph is given below:



2. Undirected Graphs

If we take a look at the pictorial representation that we had in the Real-world example above, we can clearly see that different nodes are connected by a link (i.e. edge) and that edge doesn't have any kind of direction associated with it. For example, the edge between "Anuj" and "Deepak" is bi-directional and hence the relationship between them is two ways, which turns out to be that "Anuj" knows "Deepak" and "Deepak" also knows about "Anuj". These types of graphs where the relation is bi-directional or there is not a single direction, are known as Undirected graphs.

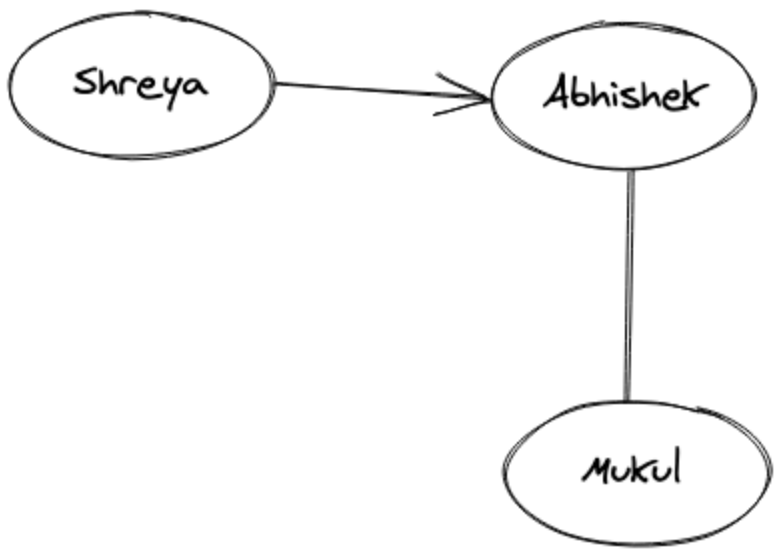
A pictorial representation of another undirected graph is given below:



3. Directed Graphs

What if the relation between the two people is something like, "Shreya" know "Abhishek" but "Abhishek" doesn't know "Shreya". This type of relationship is one-way, and it does include a direction. The edges with arrows basically denote the direction of the relationship and such graphs are known as directed graphs.

A pictorial representation of the graph is given below:

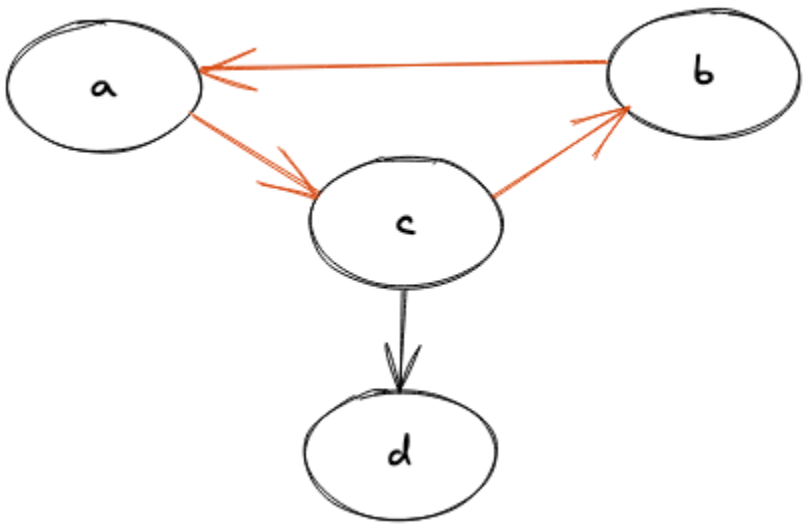


It can also be noted that the edge from "Shreya" to "Abhishek" is an outgoing edge for "Shreya" and an incoming edge for "Abhishek".

4. Cyclic Graph

A graph that contains at least one node that traverses back to itself is known as a cyclic graph. In simple words, a graph should have at least one cycle formation for it to be called a cyclic graph.

A pictorial representation of a cyclic graph is given below:

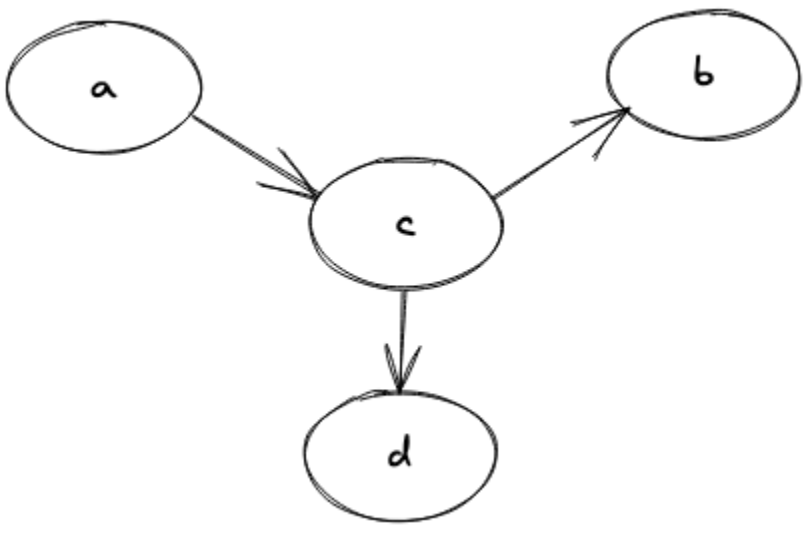


It can be easily seen that there exists a cycle between the nodes (a, b, c) and hence it is a cyclic graph.

5. Acyclic Graph

A graph where there's no way we can start from one node and can traverse back to the same one, or simply doesn't have a single cycle is known as an acyclic graph.

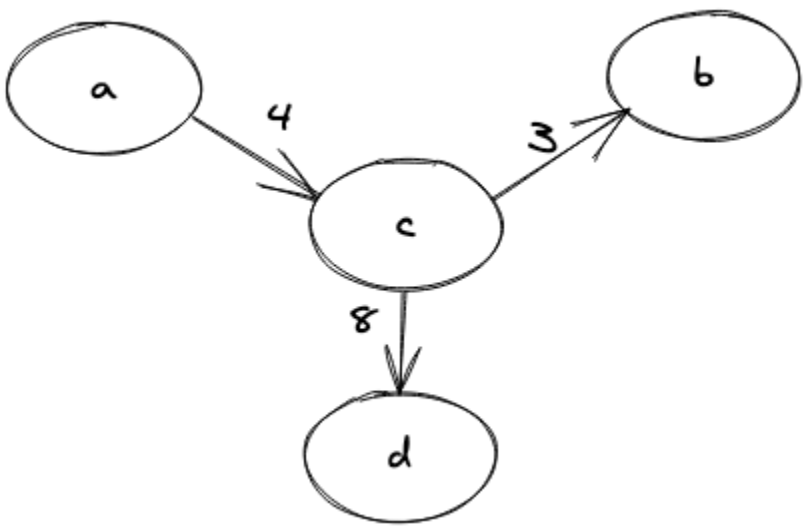
A pictorial representation of an acyclic graph is given below:



6. Weighted Graph

When the edge in a graph has some weight associated with it, we call that graph as a weighted graph. The weight is generally a number that could mean anything, totally dependent on the relationship between the nodes of that graph.

A pictorial representation of the weighted graph is given below:

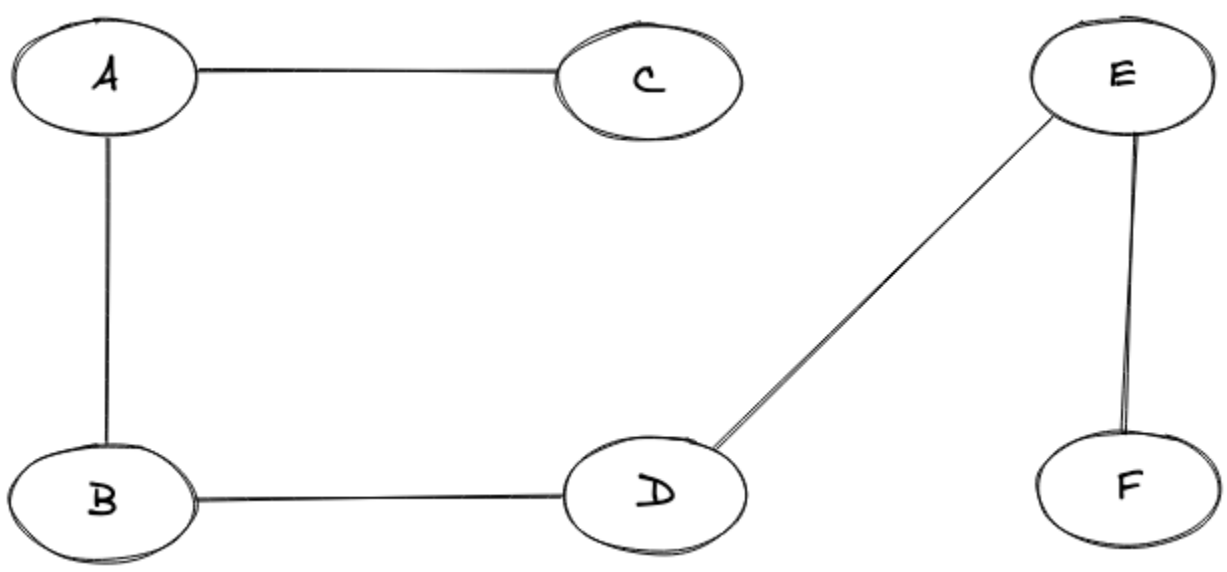


It can also be noted that if any graph doesn't have any weight associated with it, we simply call it an unweighted graph.

7. Connected Graph

A graph where we have a path between every two nodes of the graph is known as a connected graph. A path here means that we are able to traverse from a node "A" to say any node "B". In simple terms, we can say that if we start from one node of the graph we will always be able to traverse to all the other nodes of the graph from that node, hence the connectivity.

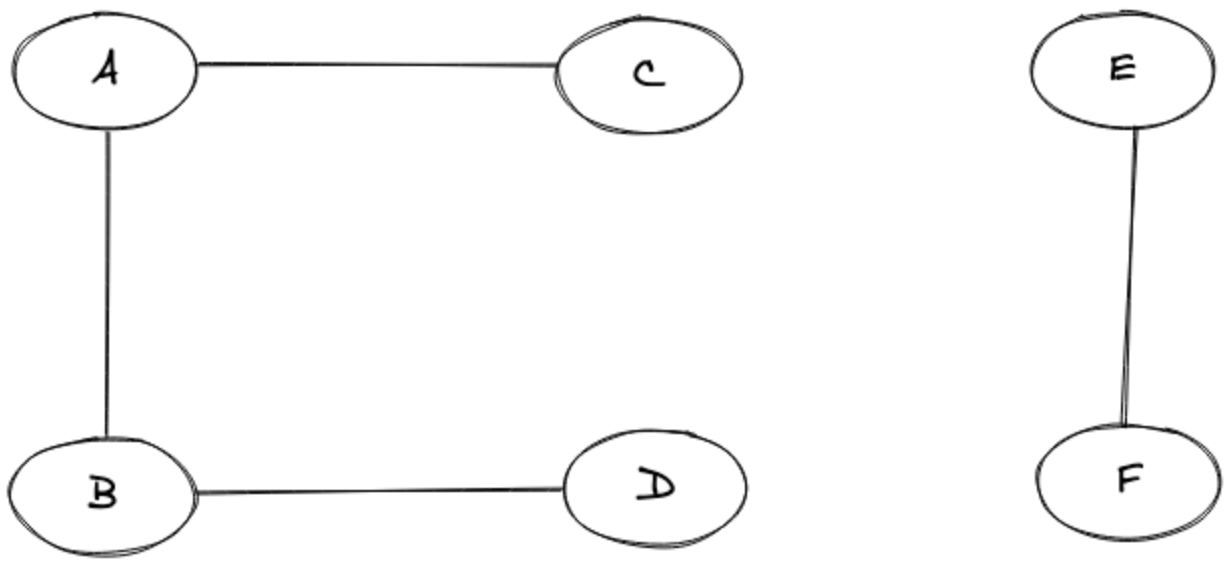
A pictorial representation of the connected graph is given below:



8. Disconnected Graph

A graph that is not connected is simply known as a disconnected graph. In a disconnected graph, we will not be able to find a path from between every two nodes of the graph.

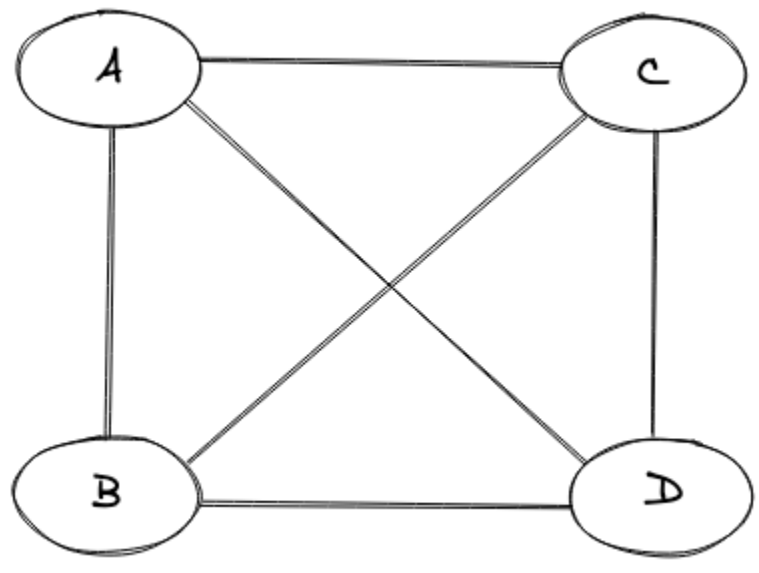
A pictorial representation of the disconnected graph is given below:



9. Complete Graph

A graph is said to be a complete graph if there exists an edge for every pair of vertices(nodes) of that graph.

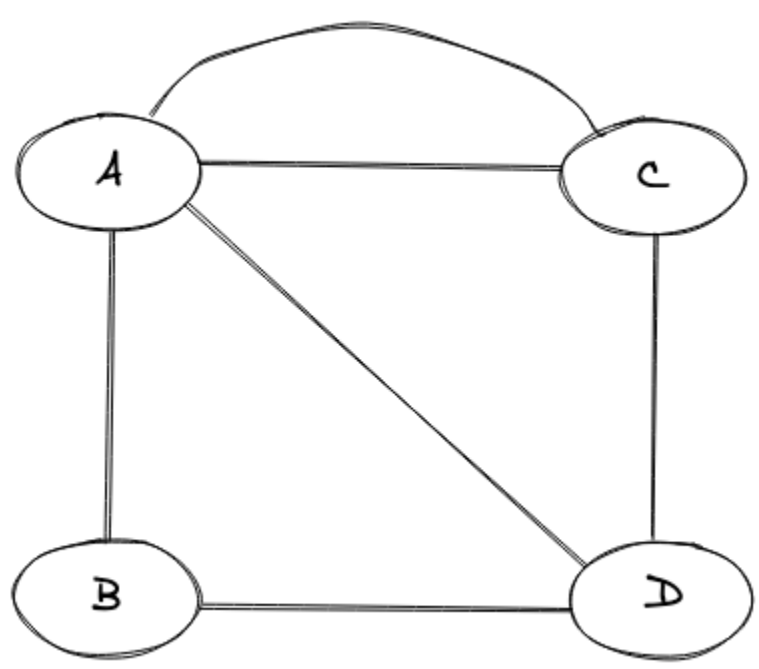
A pictorial representation of the complete graph is given below:



10. Multigraph

A graph is said to be a multigraph if there exist two or more than two edges between any pair of nodes in the graph.

A pictorial representation of the multigraph is given below:



Commonly Used Graph Terminologies

- **Path** - A sequence of alternating nodes and edges such that each of the successive nodes are connected by the edge.

- **Cycle** - A path where the starting and the ending node is the same.
- **Simple Path** - A path where we do not encounter a vertex again.
- **Bridge** - An edge whose removal will simply make the graph disconnected.
- **Forest** - A forest is a graph without cycles.
- **Tree** - A connected graph that doesn't have any cycles.
- **Degree** - The degree in a graph is the number of edges that are incident on a particular node.
- **Neighbour** - We say vertex "A" and "B" are neighbours if there exists an edge between them.

Conclusions

- We learned about what a graph is with a help of an undirected graph between different people.
- We learned how many types of graphs are there in total.
- We learned about the common terminologies that we use while talking about graphs and their subparts.

Graph Representations - Adjacency Matrix and List

There are two ways in which we represent graphs, these are:

- Adjacency Matrix
- Adjacency List

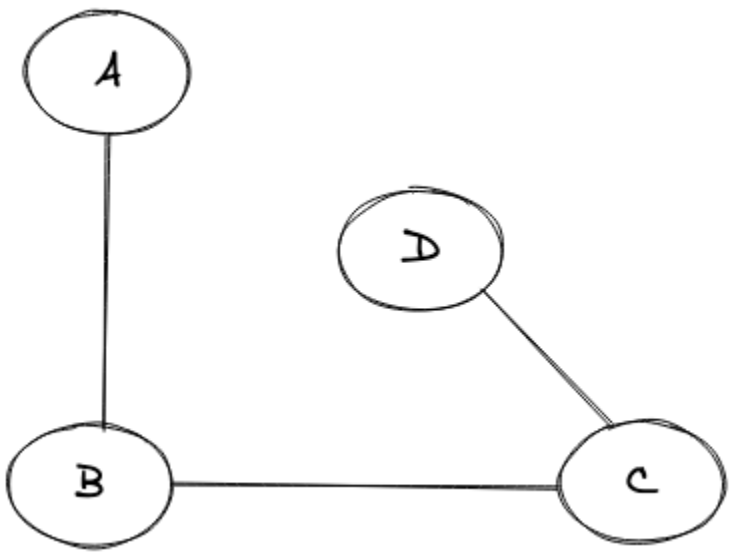
Both these have their advantages and disadvantages. In this tutorial, we will cover both of these graph representation along with how to implement them.

Adjacency Matrix

Adjacency matrix representation makes use of a matrix (table) where the **first row and first column of the matrix denote the nodes** (vertices) of the graph. The **rest of the cells contains either 0 or 1** (can contain an associated **weight w** if it is a weighted graph).

Each **row X column** intersection points to a cell and the value of that cell will help us in determining that whether the vertex denoted by the row and the vertex denoted by the column are connected or not. If the value of the cell for v1 X v2 is equal to 1, then we can conclude that these two vertices v1 and v2 are connected by an edge, else they aren't connected at all.

Consider the given graph below:

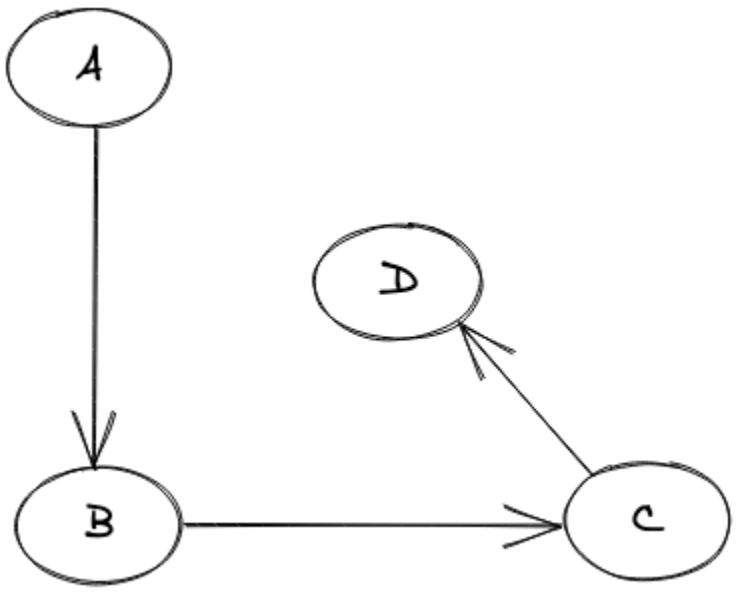


The graph shown above is an undirected one and the adjacency matrix for the same looks as:

-	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	0	1
D	0	0	1	0

The above matrix is the adjacency matrix representation of the graph shown above. If we look closely, we can see that the matrix is symmetric. Now let's see how the adjacency matrix changes for a directed graph.

Consider the given graph below:



For the directed graph shown above the adjacency matrix will look something like this:

-	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	0	0	0	1
D	0	0	0	0

Implementation of Adjacency Matrix

The structure (constructor in Java) for the adjacency matrix will look something like this:

```
public AdjacencyMatrix(int vertex){

        this.vertex = vertex;

        matrix = new int[vertex][vertex];

}
```

Copy

It should also be noted that we have two class-level variables, like:

```
int vertex;

int[][] matrix;
```

Copy

We have a constructor above named AdjacencyMatrix which takes the count of the number of the vertices that are present in the graph and then assigns our global vertex variable that value and also creates a 2D matrix of the same size. Now since our structure part is complete, we are simply left with adding the edges together, and the way we do that is:

```
public void addEdge(int start,int destination){

        matrix[start][destination] = 1;

        matrix[destination][start] = 1; // for un-directed graph

}
```

Copy

In the above `addEdge` function we also assigned 1 for the direction from the destination to the start node, as in this code we looked at the example of the undirected graph, in which the relationship is a two-way process. If it had been a directed graph, then we can simply make this value equal to 0, and we would have a valid adjacency matrix.

Now the only thing left is to print the graph.

```
public void printGraph(){

        System.out.println("Adjacency Matrix : ");

        for (int i = 0; i < vertex; i++) {

            for (int j = 0; j <vertex ; j++) {

                System.out.print(matrix[i][j]+ " ");

            }

            System.out.println();

        }

}
```

Copy

The entire code looks something like this:

```
public class AdjacencyMatrix {

    int vertex;

    int[][] matrix;


    // constructor

    public AdjacencyMatrix(int vertex){

        this.vertex = vertex;

        matrix = new int[vertex][vertex];

    }


    public void addEdge(int start,int destination){
```

```
        matrix[start][destination] = 1;

        matrix[destination][start] = 1;

    }


    public void printGraph(){

        System.out.println("Adjacency Matrix : ");

        for (int i = 0; i < vertex; i++) {

            for (int j = 0; j <vertex ; j++) {

                System.out.print(matrix[i][j]+ " ");

            }

            System.out.println();

        }

    }


    public static void main(String[] args) {

        AdjacencyMatrix adj = new AdjacencyMatrix(4);

        adj.addEdge(0,1); // 0 as the array is 0-indexed

        adj.addEdge(1,2);

        adj.addEdge(2,3);

        adj.printGraph();

    }

}
```

Copy

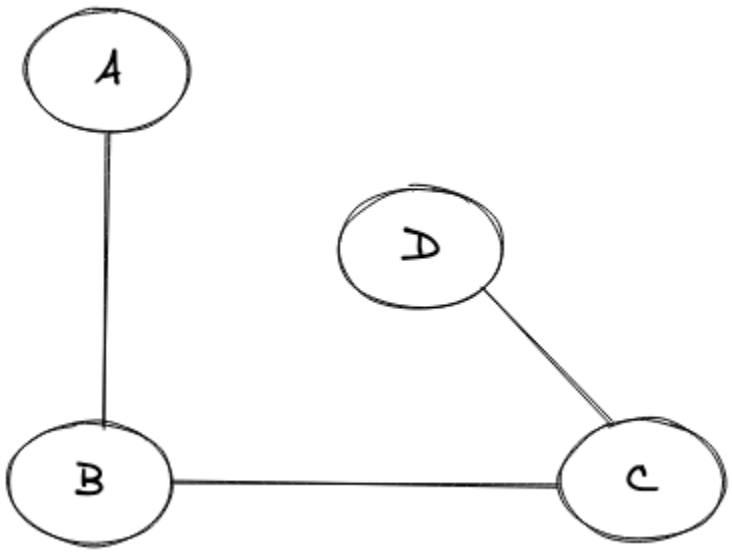
The output of the above looks like:

```
Adjacency Matrix :

0 1 0 0

1 0 1 0

0 1 0 1

0 0 1 0
```

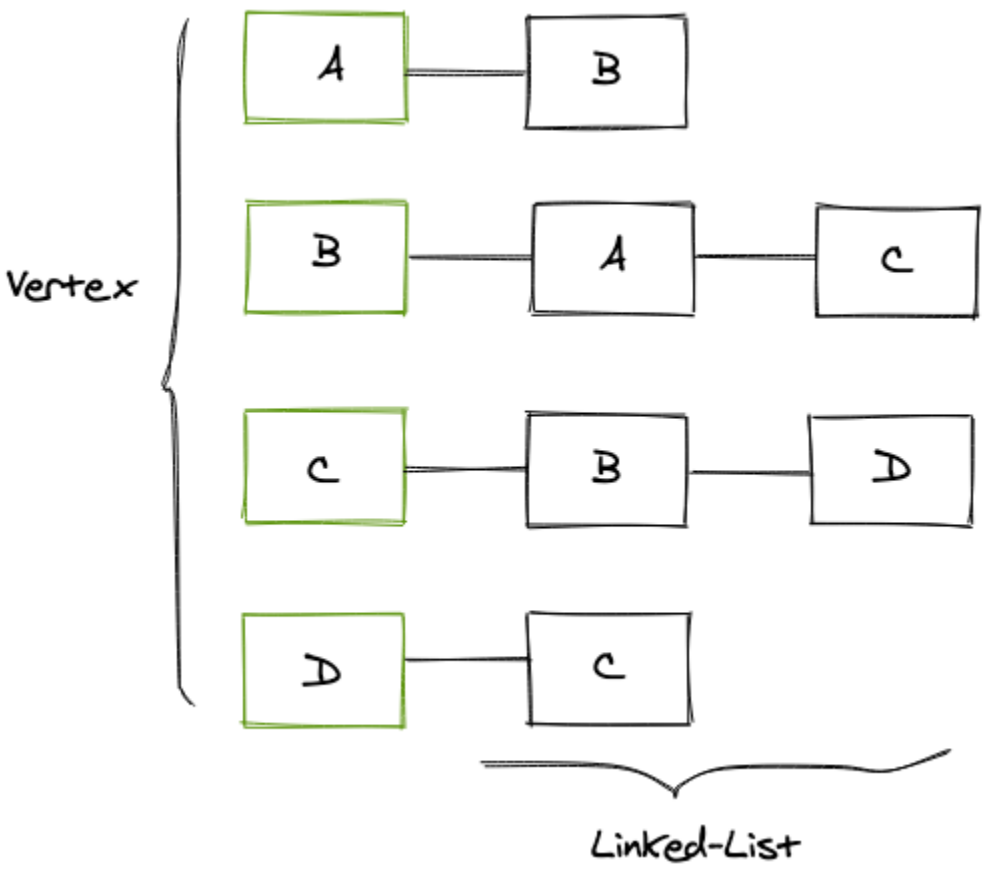
Adjacency List

In the adjacency list representation, we have an array of linked-list where the size of the array is the number of the vertex (nodes) present in the graph. Each vertex has its own linked-list that contains the nodes that it is connected to.

Consider the graph shown below:

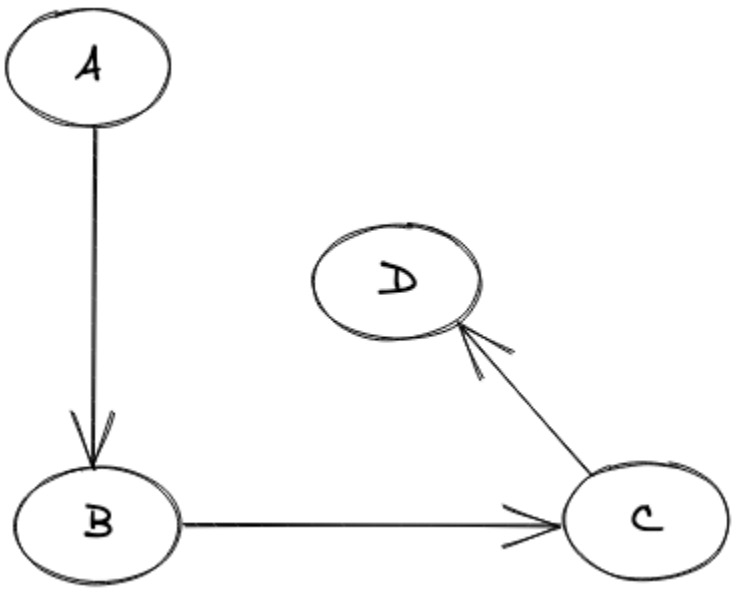


The above graph is an undirected one and the Adjacency list for it looks like:

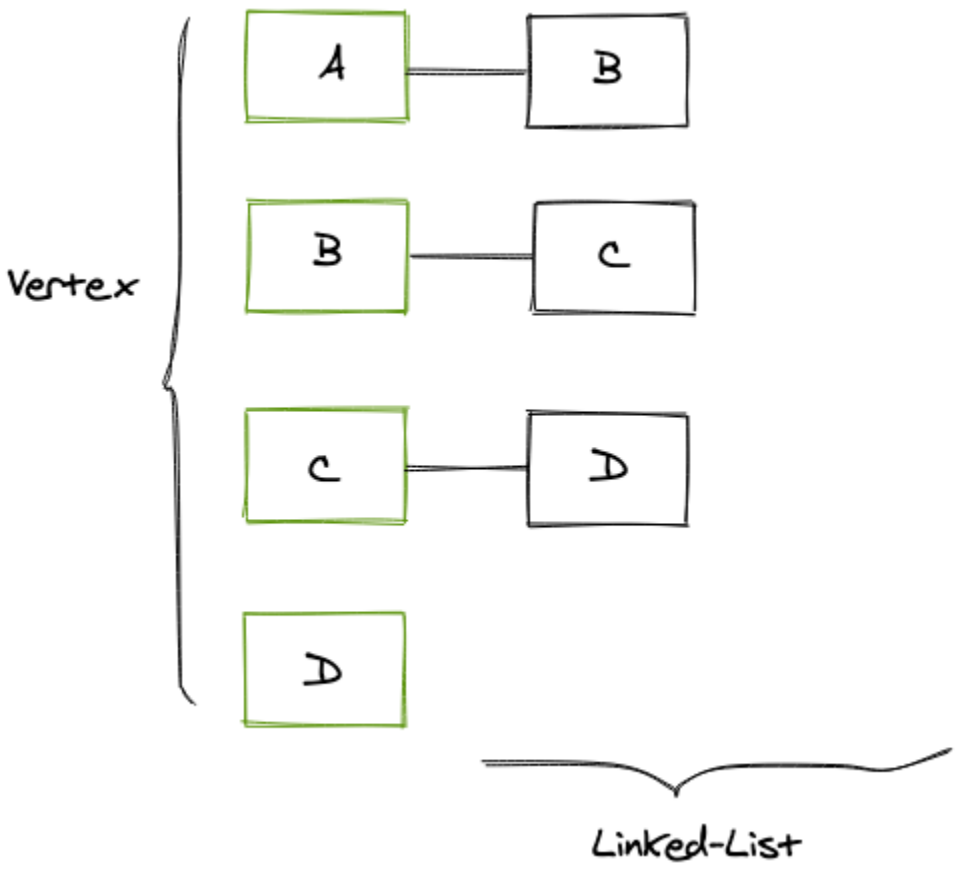


The first column contains all the vertices we have in the graph above and then each of these vertices contains a linked list that in turn contains the nodes that each vertex is connected to. For a directed graph the only change would be that the linked list will only contain the node on which the incident edge is present.

Consider the graph shown below:



The above graph is a directed one and the Adjacency list for this looks like:



Implementation of Adjacency List

The structure (constructor in Java) for the adjacency list will look something like this:

```
public AdjacencyList(int vertex){

        this.vertex = vertex;

        list = new LinkedList[vertex];

        for(int i=0;i<vertex;i++){

            list[i] = new LinkedList<Integer>();

        }

}
```

Copy

The above constructor takes the number of vertices as an argument and then assigns the class level variable this value, and then we create an array of LinkedList of the size of the vertices present in the graph. Finally, we create an empty LinkedList for each item of this array of LinkedList.

It should also be noted that we have two class-level variables, like:

```
int vertex;

LinkedList<Integer> []list;
```

Copy

Now we have laid the foundations and the only thing left is to add the edges together, we do that like this:

```
public void addEdge(int start,int destination){

        list[start].addFirst(destination);
```

```
        list[destination].addFirst(start); // un-directed graph

}
```

Copy

We are taking the vertices from which an edge starts and ends, and we are simply inserting the destination vertex in the LinkedList of the start vertex and vice-versa (as it is for the undirected graph).

Now the only thing left is to print the graph.

```
public void printGraph(){

        for (int i = 0; i < vertex ; i++) {

            if(list[i].size()>0) {

                System.out.print("Node " + i + " is connected to: ");

                for (int j = 0; j < list[i].size(); j++) {

                    System.out.print(list[i].get(j) + " ");

                }

                System.out.println();

            }

        }

}
```

Copy

The entire code looks something like this:

```
import java.util.LinkedList;


public class AdjacencyList {

    int vertex;

    LinkedList<Integer> []list;


    public AdjacencyList(int vertex){

        this.vertex = vertex;

        list = new LinkedList[vertex];

        for(int i=0;i<vertex;i++){

            list[i] = new LinkedList<Integer>();

        }
```

```
    }


    public void addEdge(int start,int destination){
        list[start].addFirst(destination);
        list[destination].addFirst(start); // un-directed graph
    }


    public void printGraph(){
        for (int i = 0; i < vertex ; i++) {
            if(list[i].size()>0) {
                System.out.print("Node " + i + " is connected to: ");
                for (int j = 0; j < list[i].size(); j++) {
                    System.out.print(list[i].get(j) + " ");
                }
                System.out.println();
            }
        }
    }


    public static void main(String[] args) {
        AdjacencyList adl = new AdjacencyList(4);
        adl.addEdge(0,1);
        adl.addEdge(1,2);
        adl.addEdge(2,3);
        adl.printGraph();
    }


}
```

Copy

The output of the above looks like:

```
Node 0 is connected to: 1

Node 1 is connected to: 2 0

Node 2 is connected to: 3 1

Node 3 is connected to: 2
```

Conclusion

- We learned how to represent the graphs in programming, via adjacency matrix and adjacency lists.