Introduction to Graph

Reference:

https://www.geeksforgeeks.org/graph-types-and-applications/

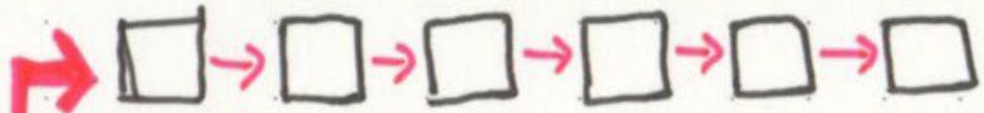
https://medium.com/basecs/a-gentle-introduction-to-graph-theory-77969829ead8

https://www.britannica.com/topic/graph-theory

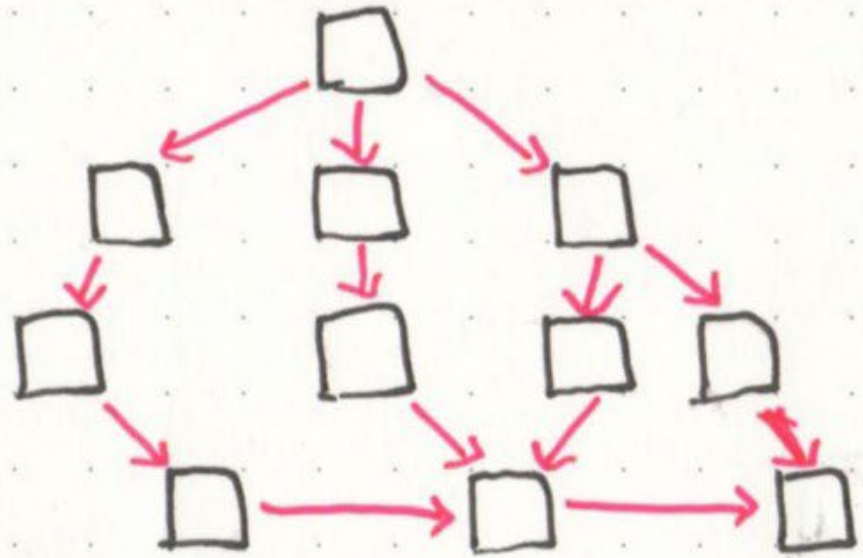
https://medium.com/basecs/whats-a-linked-list-anyway-part-1-d8b7e6508b9d#.3zvtp22ui

https://dev.to/thisdotmedia/5-minute-introduction-to-cypher-query-language-45e2

Linear vs. Non-linear structures

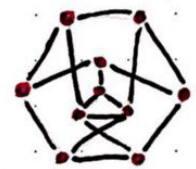


In linear data structures, we traverse through elements/nodes sequentially.



In non-linear data structures, one node might be connected to several other nodes, so it can't always be traversed sequentially.

A VERY BRIEF INTRODUCTION TO *graph theory*



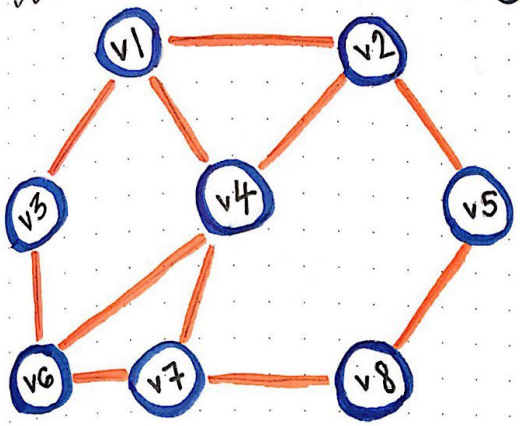
→ Graphs are a way to formally represent a network, or a collection of interconnected objects.

→ In mathematics, graphs are defined as ordered pairs, with two parts: vertices & edges.

So, what's the definition of a graph?

it looks like this! → $G = (V, E)$ where **V** is a set of nodes, also called vertices and **E** is a set of edges, also called links.

(Formally) Defining a Graph



8 vertices/nodes

11 edges/links

$G = (V, E)$

$V = \{v1, v2, v3, v4, v5, v6, v7, v8\}$

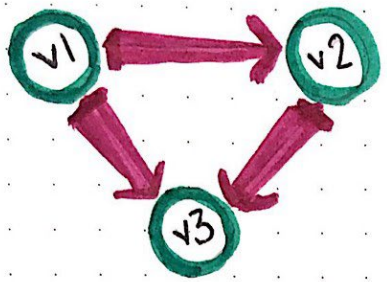
$E = \{\{v1, v2\}, \{v1, v3\}, \{v1, v4\}, \{v2, v4\}, \{v2, v5\}, \{v3, v6\}, \{v4, v6\}, \{v4, v7\}, \{v5, v8\}, \{v6, v7\}, \{v7, v8\}\}$

these edge definitions are unordered pairs!

→ G=(V,E) is the formal mathematical notation for defining graphs.

→ A graph G is an ordered pair of a set V vertices and E, a set of edges.

→ An ordered pair is a pair of mathematical objects in which the order of objects in the pair matters.



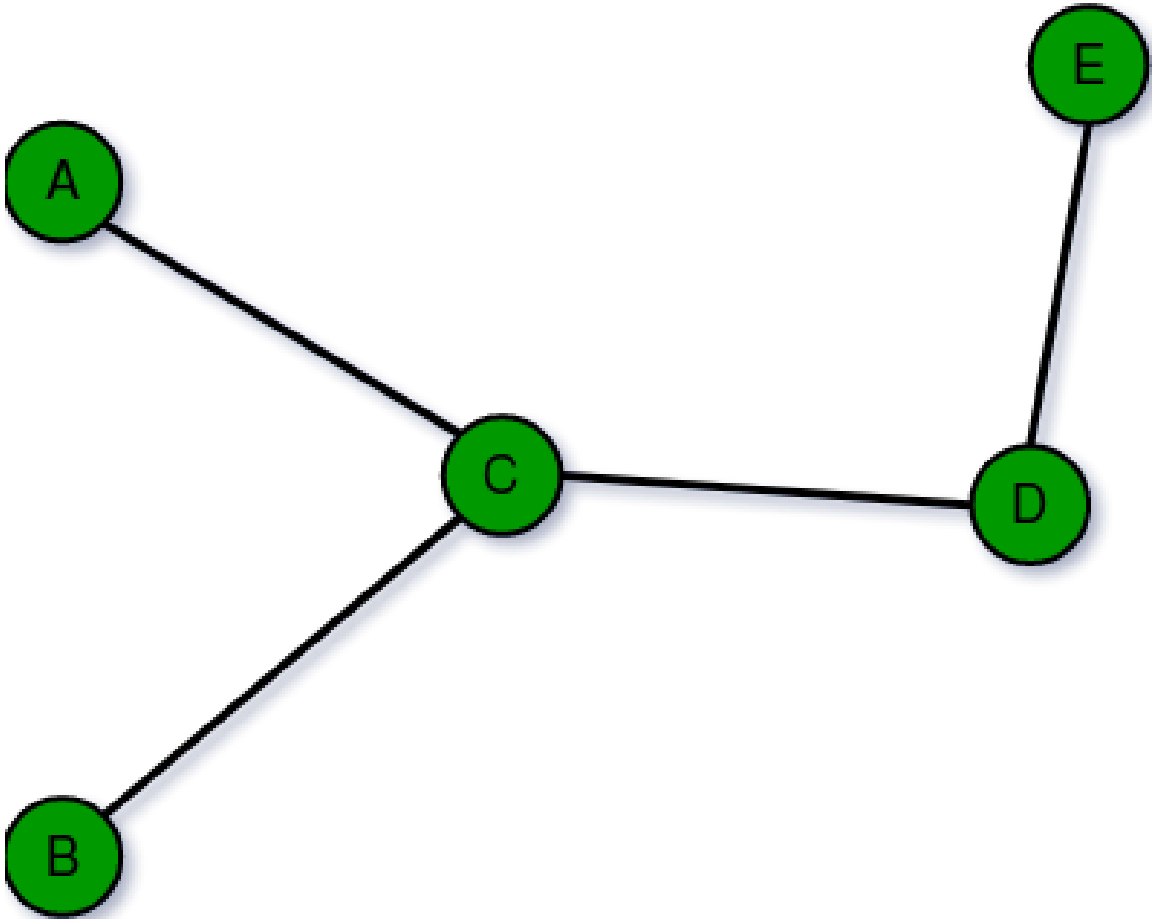
But what about a directed graph?

$G = (V, E)$ → how would our edge objects be different?

$V = \{v1, v2, v3\}$

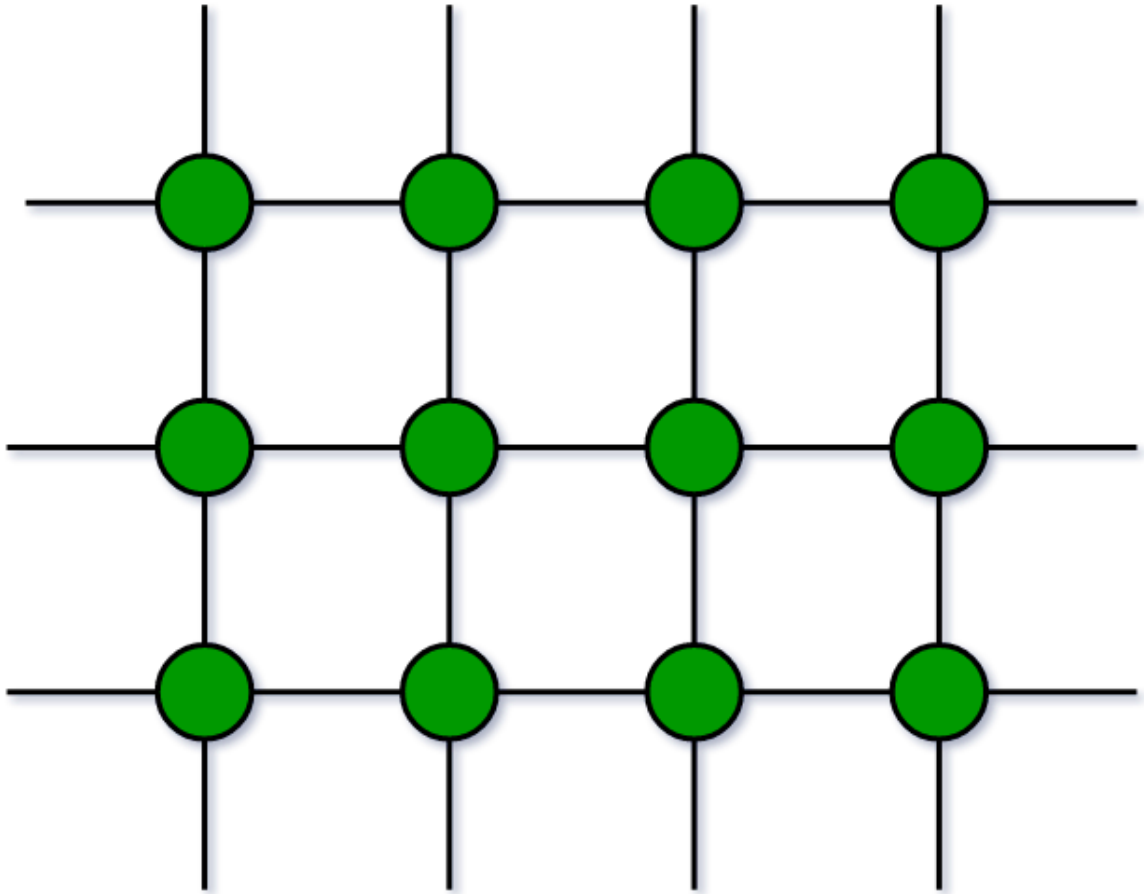
$E = \{(v1, v2), (v1, v3), (v2, v3)\}$

these edge definitions are ordered pairs, because direction matters!



Finite Graphs

- A graph is said to be finite if it has a finite number of vertices and a finite number of edges.

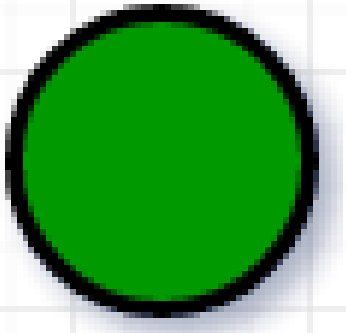


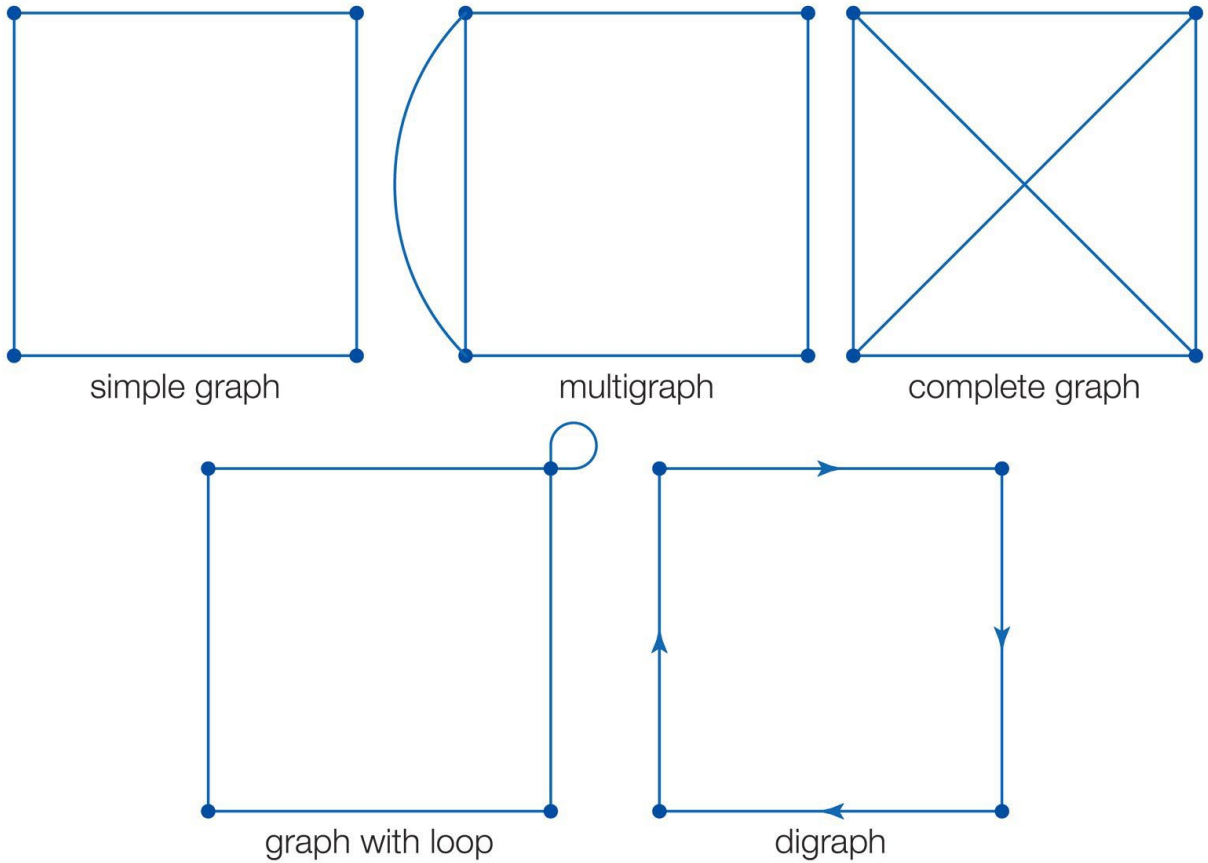
Infinite Graph:

- A graph is said to be infinite if it has an infinite number of vertices as well as an infinite number of edges.

Trivial Graph:

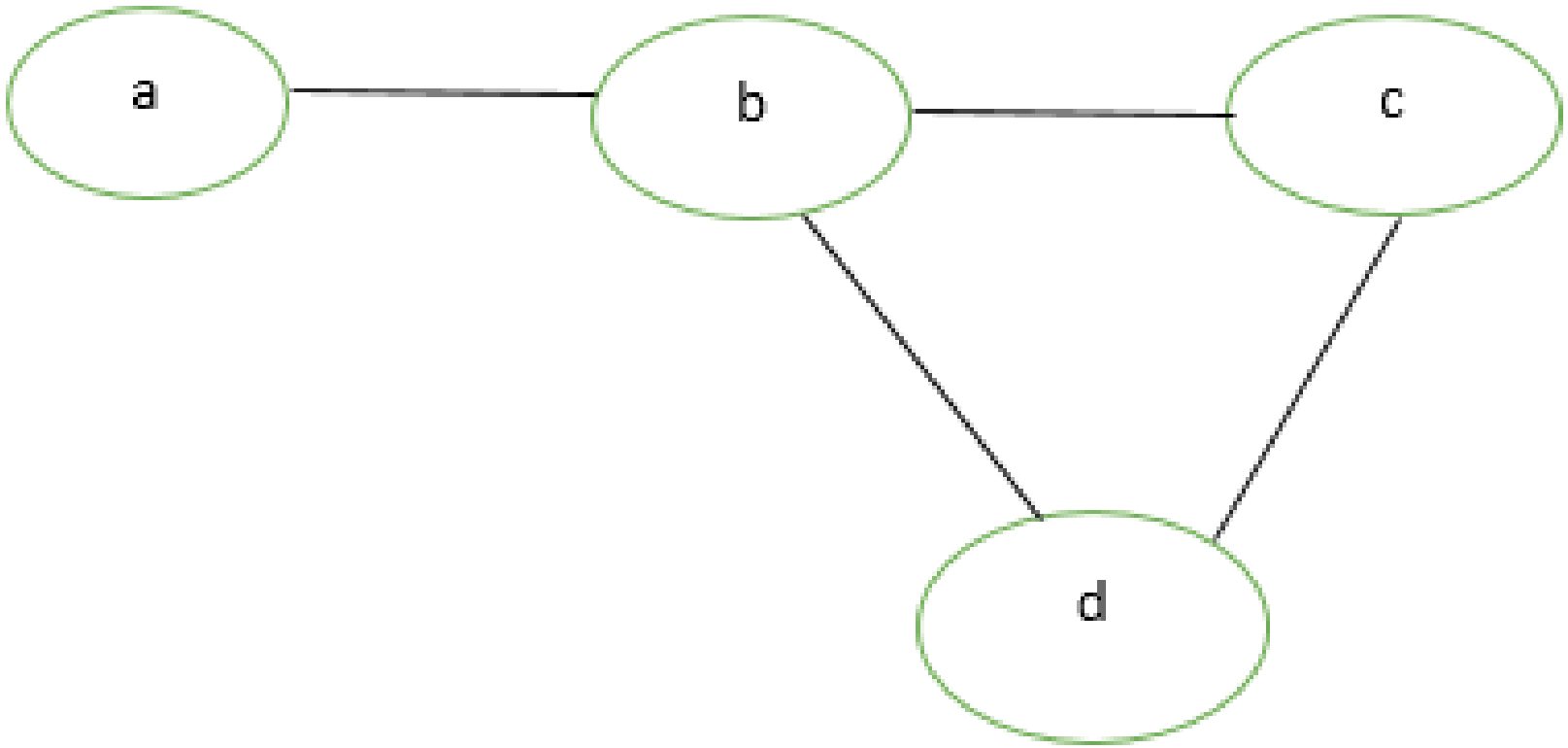
A graph is said to be trivial if a finite graph contains only one vertex and no edge.





Simple Graph:

A simple graph is a graph that does not contain more than one edge between the pair of vertices. A simple railway track connecting different cities is an example of a simple graph.

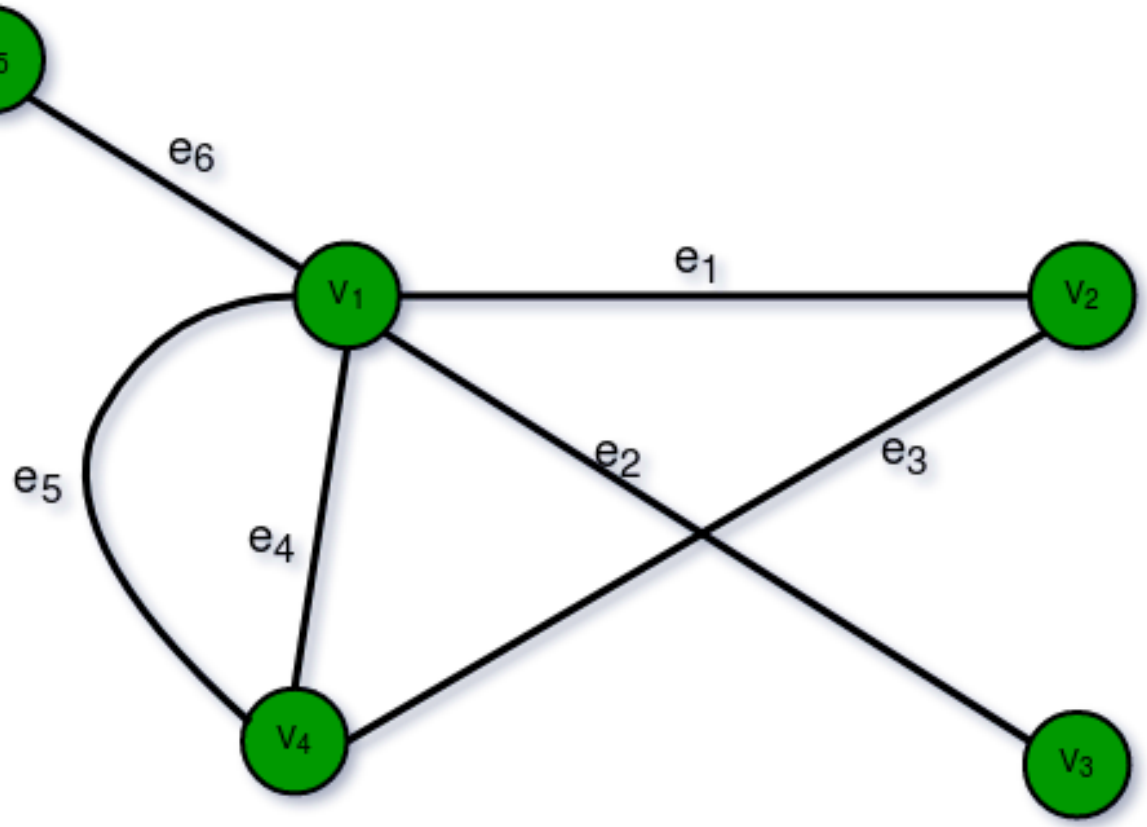


Multi Graph:

Any graph which contains some parallel edges but doesn't contain any self-loop is called a multigraph. For example a Road Map.

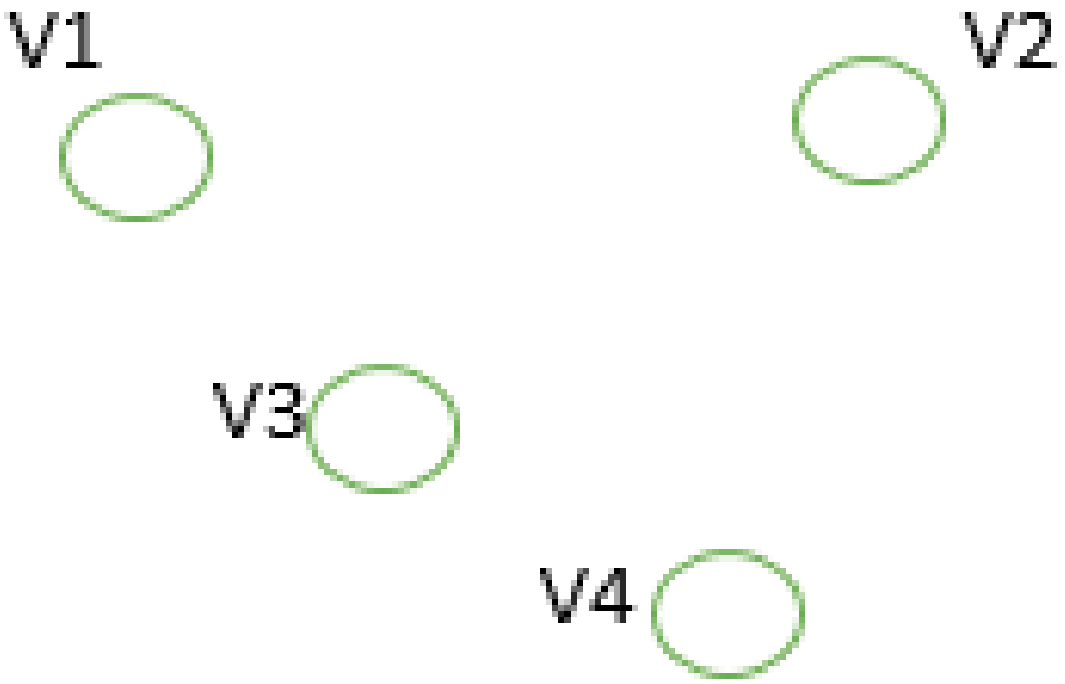
•**Parallel Edges:** If two vertices are connected with more than one edge then such edges are called parallel edges that are many routes but one destination.

•**Loop:** An edge of a graph that starts from a vertex and ends at the same vertex is called a loop or a self-loop.



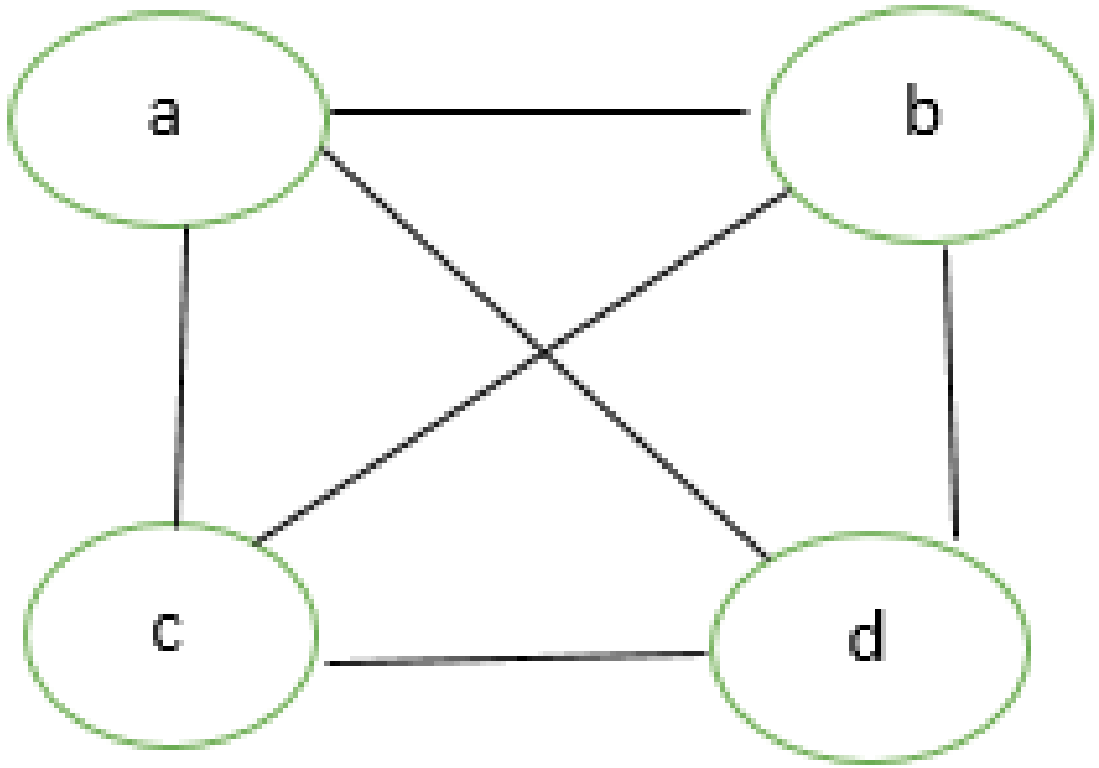
6. Null Graph:

A graph of order n and size zero is a graph where there are only isolated vertices with no edges connecting any pair of vertices.



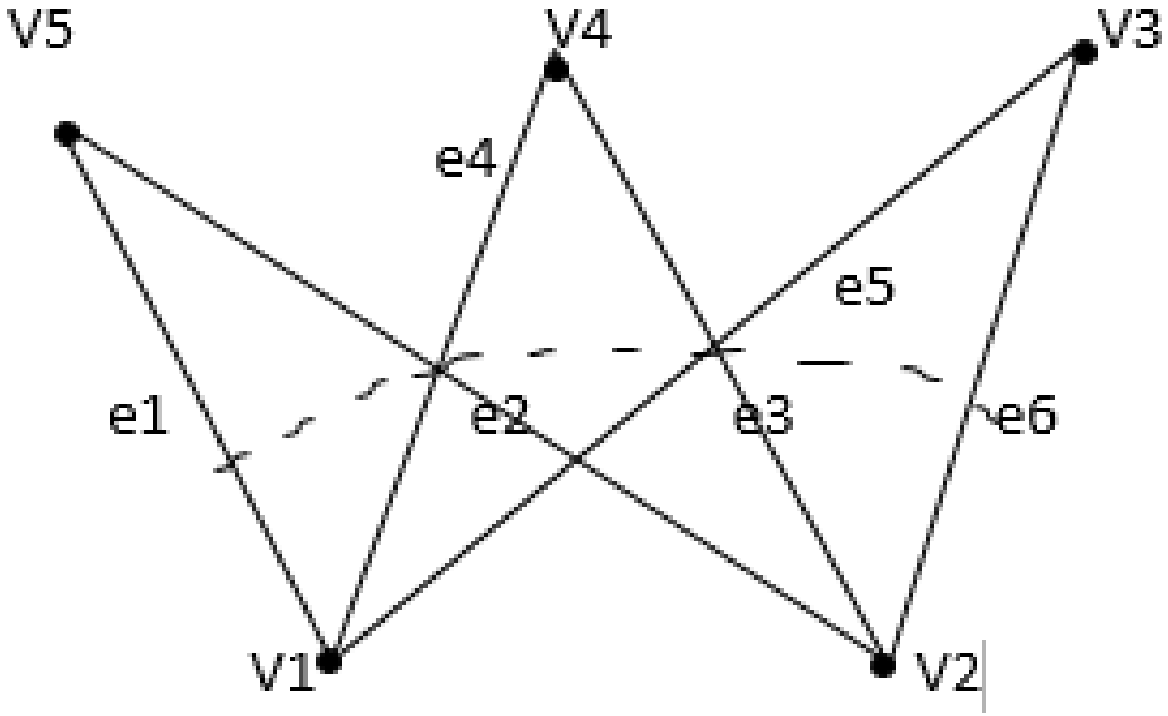
Complete Graph

A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attached with n-1 edges or the rest of the vertices in the graph. A complete graph is also called Full Graph.



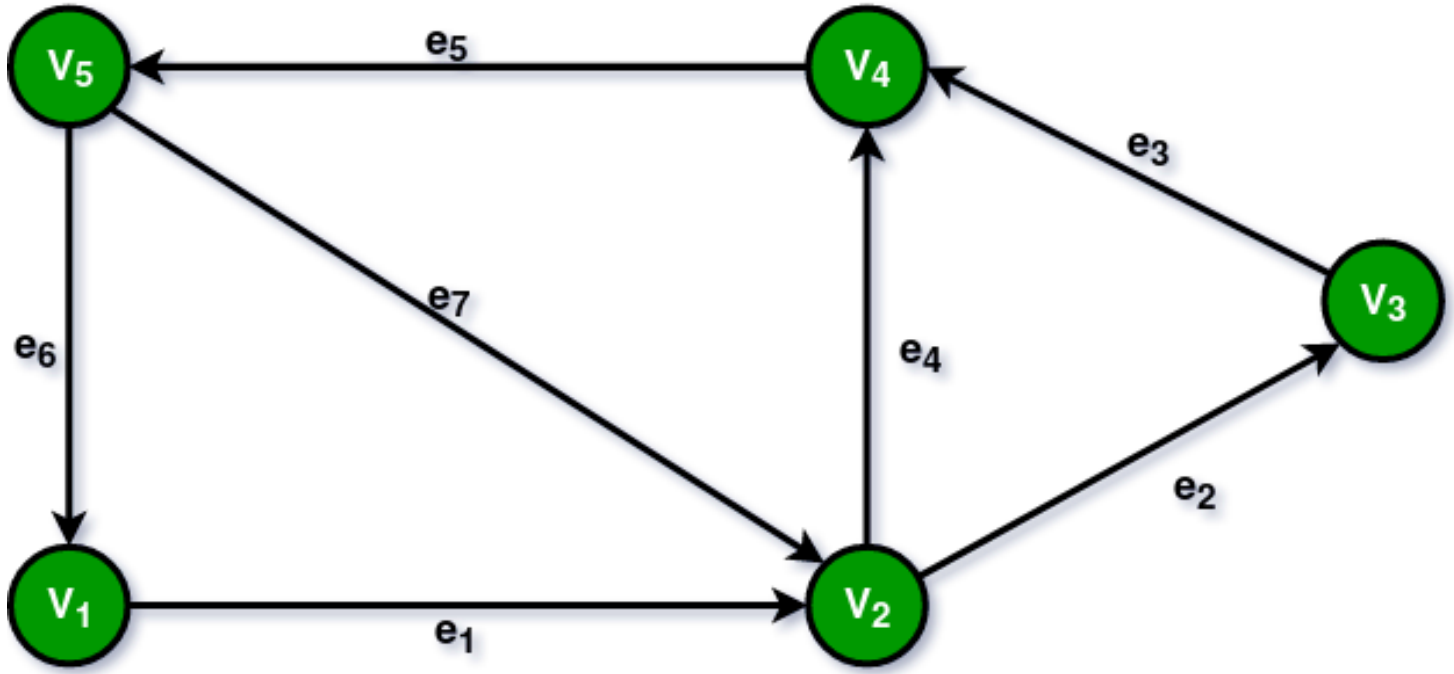
Bipartite Graph:

A graph G = (V, E) is said to be a bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has one end in V1(G) and another end in V2(G). The partition V1 U V2 = V is called Bipartite of G. Here in the figure: V1(G)={V5, V4, V3} and V2(G)={V1, V2}



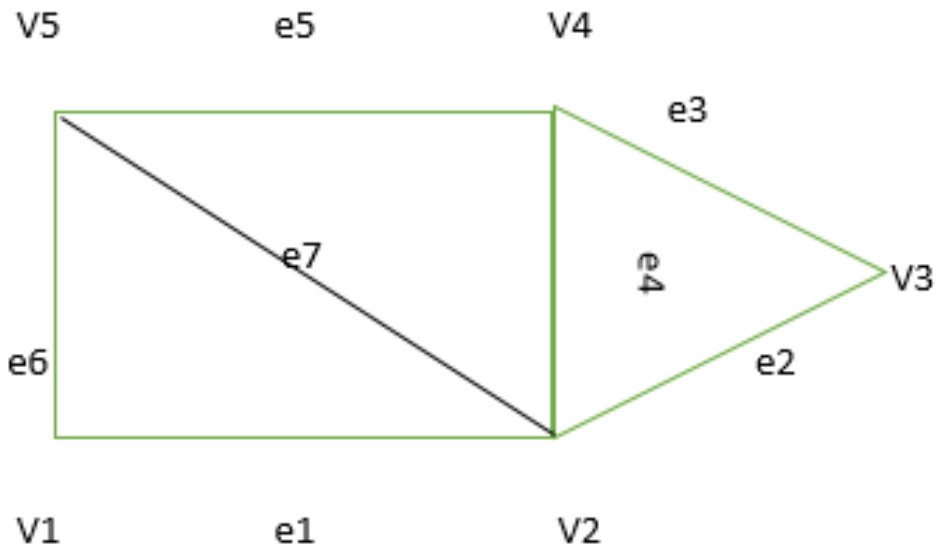
Digraph Graph

A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) are called a Digraph. It is also called *Directed Graph*. The ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj. Here in the figure: e1 = (V1, V2) e2 = (V2, V3) e4 = (V2, V4)



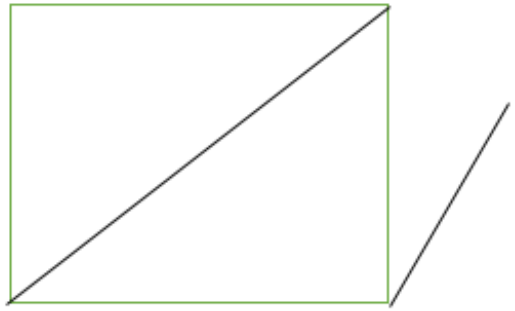
Subgraph

A graph G1 = (V1, E1) is called a subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.

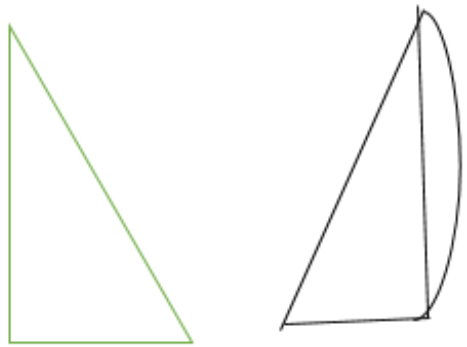


Connected or Disconnected Graph:

Graph G is said to be connected if any pair of vertices (Vi, Vj) of a graph G is reachable from one another. Or a graph is said to be connected if there exists at least one path between each and every pair of vertices in graph G, otherwise, it is disconnected. A null graph with n vertices is a disconnected graph consisting of n components. Each component consists of one vertex and no edge.



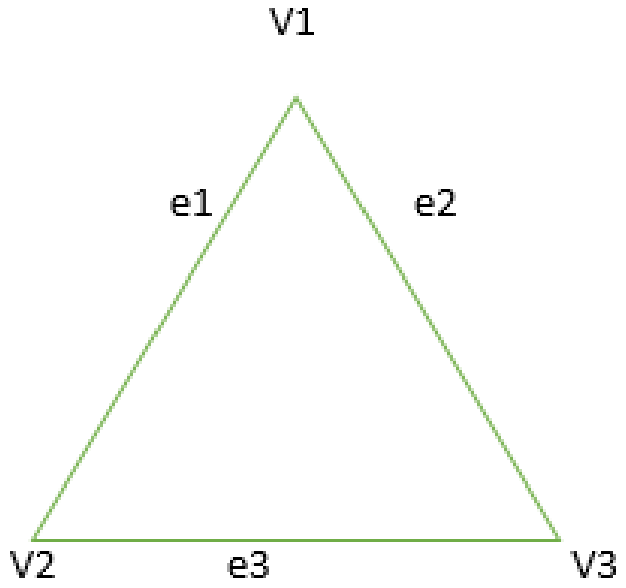
(a)

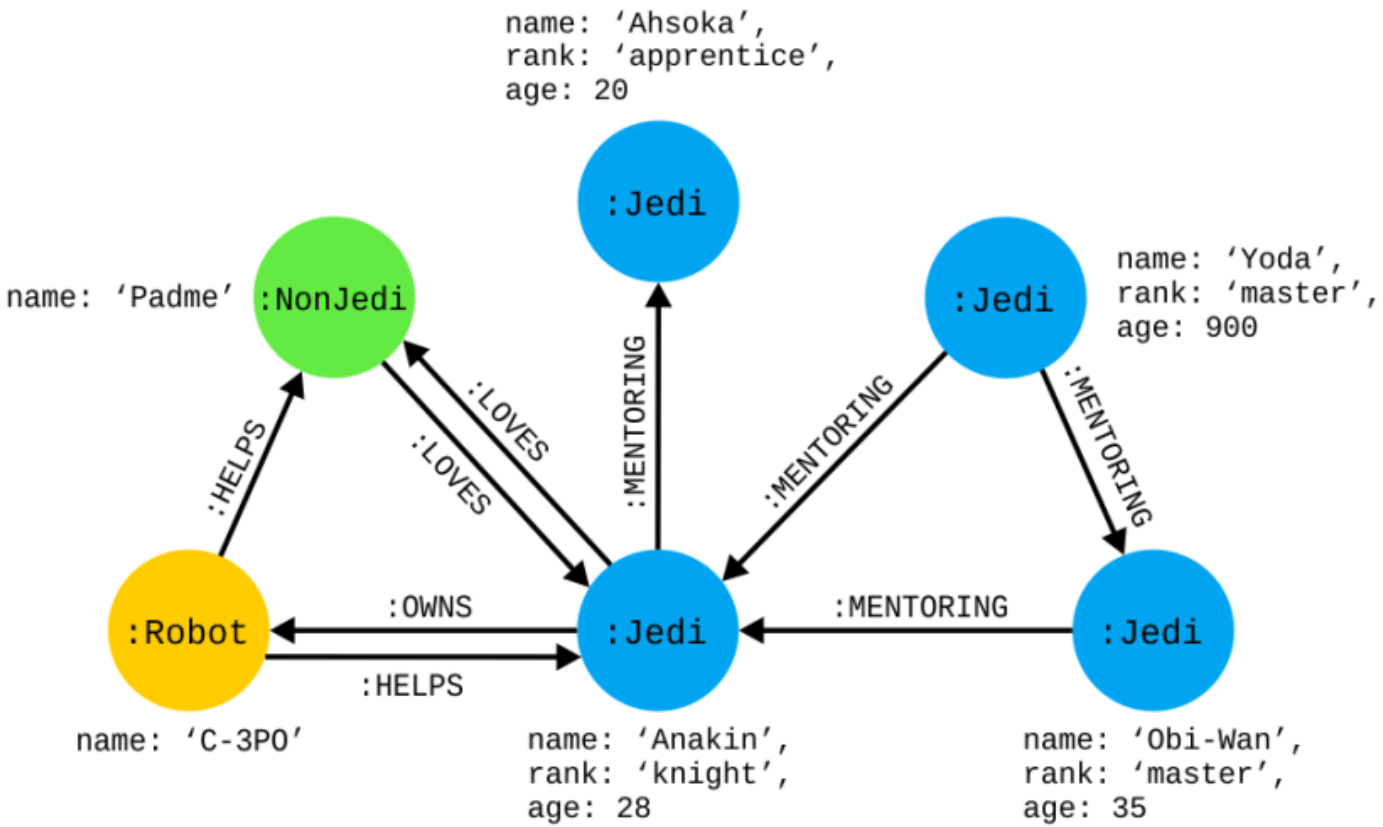


(b)

Cyclic Graph

A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – (Vn, V1) are called cyclic graph.





Our data

Since we need something to work on first, we will introduce the following graph database:

Writing queries in Cypher

After having a basic understanding of what a graph is, and how the data is composed in the database, you'll find the syntax of Cypher very simple and intuitive to read and write. It is inspired by ASCII art, so you actually model your queries in a some graphical manner. Let's get started.

Nodes and relationships

Let’s start with the core structures of the graph -- nodes and relationships -- they are represented by `()` and `[]` plus `<` or `>`. For example, let's take a look at the relationship between Anakin and Padme, and how it can be represented with Cypher:

```
(:Jedi)-[:LOVES]->(:NonJedi)
```

Simple, isn't it? The arrows show how the nodes relate based on the relationship type that we have. Also, we have labels (or tags), which indicate the type of the nodes and the relationship.

Getting data with `MATCH`

We'll start with some basic examples of fetching data. I'll re-use the query from the previous section. Along with the `MATCH` keyword, we will be able to ask the database whether there are nodes that match our provided pattern, or subgraph.

The following query will return all jedi that are in love with non-jedi characters.

```
MATCH (j:Jedi)-[:LOVES]->(:NonJedi)
RETURN j
```

You probably noticed that we introduced a `j` in the query. This is simply a variable used for referring to the jedi node. Note that we can use these for relationships as well.

Conditional fetching

Just like in a conventional relational database, we can specify some criteria when fetching data. With Cypher, we have two ways to do that:

1. Inlined JSON-like object

Let's take a look at the following query:

```
MATCH
  (j1:Jedi { rank: 'master' })-[:MENTORING]->(j2:Jedi)
RETURN
  j2.name
```

_ **Note:** Both single and double quotes are valid._

We can describe it as: "*Return the names of all jedis who are mentored by a jedi master*. Again, this results in getting Anakin's name, but not Ahsoka. Now, we'll write the same query but a bit differently:

2. Using `WHERE`

```
MATCH
  (j1:Jedi)-[:MENTORING]->(j2:Jedi)
WHERE
  j1.rank = 'master'
RETURN
  j2.name
```

Using the `WHERE` clause is equivalent to inlining the JSON-like object in a node. However, it is a fuller and more powerful way of specifying our desired criteria compared to the latter. For example, we can perform a boundary check of a numerical data, whereas that would be impossible with the first approach:

```
MATCH
  (j1:Jedi)-[:MENTORING]->(j2:Jedi)
WHERE
  j1.age < 100
RETURN
  j2.name
```

Keep in mind that we can apply all of these for the relationships as well.

Updating the existing data

After we have a good perception of how data is fetched from the database, we can spend a minute on how it can be updated. This happens with the help of the `SET` keyword:

```
MATCH
  (j:Jedi { name: 'Anakin' })
SET
  j.placeOfBirth = 'Tatooine'
  j.dateOfBirth = '41 BBY'
RETURN
  j
```

The explanation is obvious -- based on the provided search criteria, we perform an update via `SET` on the returned results. That's it!

Creating new data

In this sub-section, we will introduce two more keywords needed for adding new data to our database. These are `CREATE` and `MERGE`. Let's start with a simple example of "create":

```
CREATE (:Jedi { name: 'Qui-Gon' })
```

As you might have guessed, it simply adds a new node to our database. But what if we want to create a relation with this node? Well, here, `MERGE` comes in handy. It is intended for ensuring that the provided pattern exists in the database, which means that if it doesn't, it will be created.

Using the example we have, we can extend it so that Obi-Wan is an apprentice of Qui-Gon:

```
CREATE
  (qg:Jedi { name: 'Qui-Gon Jinn' })
MERGE
  (qg)-[:MENTORING]->(:Jedi { name: 'Obi-Wan' })
```