Graphs are complex, non-linear data structures that are characterized by a group of vertices, connected by edges. Every vertex has a value associated with it. For example, if we represent a list of cities using a graph, the vertices would represent the cities. Edges represent the relationship between the vertices in the graph. Edges may or may not have a value associated with them. For example, if we represent a list of cities using a graph, the edges would represent the path between the cities.

There are many applications of graphs, from schooling to business, and many operations can be performed with them. Besides visualising data, graphs can be used by social media platforms, for example, to represent different users as vertices and edges to represent the connections between them. As mentioned before, a mapping application can use graphs to represent places and the path (distance) between them.

Graphs are based on three properties:

- Weight: they can be weighted or unweighted.
- Direction: they can be directed or undirected.
- Interconnectivity: they can be cyclic or acyclic.

A graph can be represented in 3 ways: Adjacency List, Adjacency Matrix and Edge List.

An Adjacency List stores the nodes with which a particular node is connected to in a linked list or array. All these lists can be stored in a hash table, with the keys being the nodes, and the values being their respective lists. Example, A → [(B, 4), (C, 1)] represents an adjacency list where the vertex A is connected to B (weight 4) and C (weight 1). This works really well for sparse graphs. Adjacency lists have got the advantage when iterating over the edges, as it is more efficient. The edge weight lookup is O(E) - worse case.

An Adjacency Matrix is a very simple way to represent a graph. In a weighted graph, the element A[i][j] represents the cost of moving from vertex i to vertex j. In an unweighted graph, the element A[i][j] represents a Boolean value that determines whether a path exists from vertex i to vertex j. If A[i][j] == 0, then no path from vertex i to vertex j exists. If A[i][j] == 1, there is a path from vertex i to vertex j. The space complexity of this data structure is O(V^2). Iterating through the edges takes O(V^2) time. The time complexity of getting an edge weight is O(1).

An Edge List represents the graph as an unstructured list of edges. It contains all the pairs of nodes which are connected, and if the graph is weighted, then the weight of each edge is represented as well. They are not widely used because this representation lacks structure, but its representation is extremely simple. Like the Adjacency List, to look up the edge weight on an Edge List, the time complexity is O(E) - worse case.

Below, we will build an undirected graph using an Adjacency List:

In [2]:

```
class Graph():

# The constructior initializes the number of vertices in the graph on zero, and the adjcency list will be an empty
    def __init__(self):
        self.number_of_nodes = 0
        self.adjacency_list = {}


    def insert_node(self, data):
        if data not in self.adjacency_list:
            self.adjacency_list[data] = []
            self.number_of_nodes += 1
            return

# We start by implementing the insert node method.
# It adds the value of the node as a key in the adjacency list and initializes the value of the key as an empty ar

    def insert_edge(self, vertex1, vertex2):
        if vertex2 not in self.adjacency_list[vertex1]:
            self.adjacency_list[vertex1].append(vertex2)
            self.adjacency_list[vertex2].append(vertex1)
            return
        return "Edge already exists"

# Next up, we implement the insert edge method, where we specify two nodes.
# Then, between those two nodes, the method will add an edge.
# First, the method checks if an edge already exists by checking the adjacency list of either of the two nodes.
# If the other node is present, it means the edge already exists.
# Otherwise, it inserts the edge by adding the complimentary node in the adjacency list of either node.

    def show_connections(self):
        for node in self.adjacency_list:
            print(f'{node} -->> {" ".join(map(str, self.adjacency_list[node]))}')

# Finally, we implement a custom print method that prints the nodes and their connections.
```

To test the blueprint we use it to create a new graph:

In [3]:

```
# Initializing a new graph
new_graph = Graph()

# Inserting nodes and edges to the newly created graph
new_graph.insert_node(1)
new_graph.insert_node(2)
new_graph.insert_node(3)
new_graph.insert_edge(1,2)
new_graph.insert_edge(1,3)
new_graph.insert_edge(2,3)

# Showing the connections
new_graph.show_connections()
```

```
1 -->> 2 3
2 -->> 1 3
3 -->> 1 2
```

In [6]:

```
print(new_graph.adjacency_list)
```

```
{1: [2, 3], 2: [1, 3], 3: [1, 2]}
```

In [7]:

```
print(new_graph.number_of_nodes)
```

```
3
```