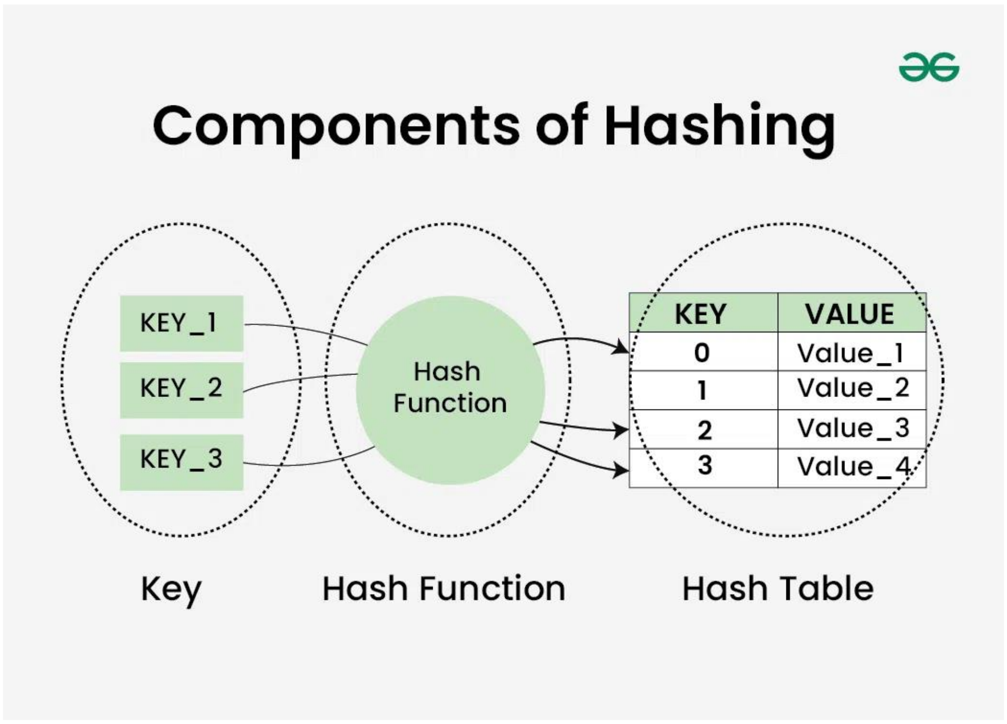
What is Hash Table?

A Hash table is defined as a data structure used to insert, look up, and remove key-value pairs quickly. It operates on the hashing concept, where each key is translated by a hash function into a distinct index in an array. The index functions as a storage location for the matching value. In simple words, it maps the keys with the value.



What is a Hash function?

A Function that translates keys to array indices is known as a hash function. The keys should be evenly distributed across the array via a decent hash function to reduce collisions and ensure quick lookup speeds.

- **Integer universe assumption:** The keys are assumed to be integers within a certain range according to the integer universe assumption. This enables the use of basic hashing operations like division or multiplication hashing.

- **Hashing by division:** This straightforward hashing technique uses the key's remaining value after dividing it by the array's size as the index. When an array size is a prime number and the keys are evenly spaced out, it performs well.
- **Hashing by multiplication:** This straightforward hashing operation multiplies the key by a constant between 0 and 1 before taking the fractional portion of the outcome. After that, the index is determined by multiplying the fractional component by the array's size. Also, it functions effectively when the keys are scattered equally.

Applications of Hash Table:

- Hash tables are frequently used for indexing and searching massive volumes of data. A search engine might use a hash table to store the web pages that it has indexed.
- Data is usually cached in memory via hash tables, enabling rapid access to frequently used information.
- Hash functions are frequently used in cryptography to create digital signatures, validate data, and guarantee data integrity.
- Hash tables can be used for implementing database indexes, enabling fast access to data based on key values.

Implementing a Hash Table

To implement a hash table, you must use an array because you have to be able to access each position of the array **directly**. You do this by specifying the index of the position (within the array).

In hash tables the positions within an array are sometimes called buckets; each bucket is used to store data. This can be a single piece of data (such as in our examples) or can be a more complex object such as a record. The key must be stored as part of, or alongside, the data.

The size of the array must be planned carefully. It must be big enough to store all of the data but not so large that you waste space. However, an effective hash table must always have some free space.

The load factor of a hash table can be defined as:

$loadfactor=kn$

where *k* is the number of buckets (positions) in the array and *n* is the number of occupied buckets. Keeping the load factor at around 0.75 is optimal.

Introduction to Graph Data Structure

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).

Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges. This web of connections is exactly what a graph data structure represents, and it's the key to unlocking insights into team performance and player dynamics in sports.

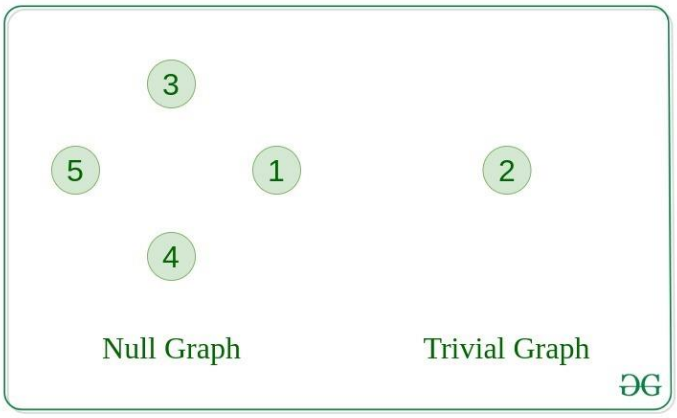
Types Of Graphs in Data Structure and Algorithms

1. Null Graph

A graph is known as a null graph if there are no edges in the graph.

2. Trivial Graph

Graph having only a single vertex, it is also the smallest graph possible.



Representation of Graph Data Structure:

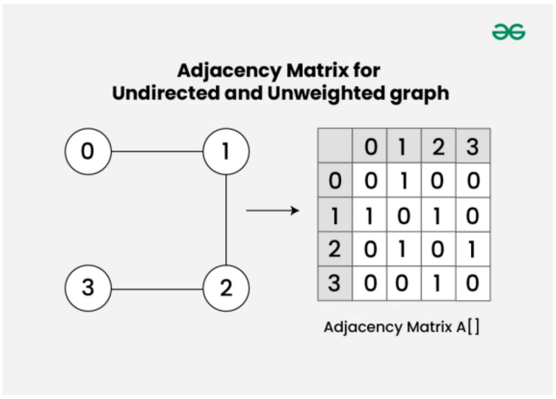
There are multiple ways to store a graph: The following are the most common representations.

- Adjacency Matrix

- Adjacency List

Adjacency Matrix Representation of Graph Data Structure:

In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.

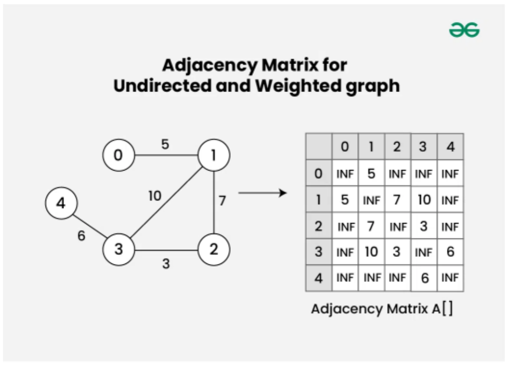


	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

Adjacency Matrix A[]

2. Adjacency Matrix for Undirected and Weighted graph:

Consider an Undirected and Weighted graph G with 5 vertices and 5 edges. For the graph G, the adjacency matrix would look like:



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF

Adjacency Matrix A[]

Advantages of Adjacency Matrix:

- Simple: Simple and Easy to implement.
- Space Efficient for Dense Graphs: Space efficient when the graph is dense as it requires V * V space to represent the entire graph.

- Faster access to Edges: Adjacency Matrix allows constant look up to check whether there exists an edge between a pair of vertices.

Disadvantages of Adjacency Matrix:

- Space inefficient for Sparse Graphs: Takes up O(V* V) space even if the graph is sparse.
- Costly Insertions and Deletions: Adding or deleting a vertex can be costly as it requires resizing the matrix.
- Slow Traversals: Graph traversals like DFS, BFS takes O(V * V) time to visit all the vertices whereas Adjacency List takes only O(V + E).