Usage of hash function in python in data structure

Hashing is a data structure that is used to store a large amount of data, which can be accessed in O(1) time by operations such as search, insert and delete. Various Applications of Hashing are:

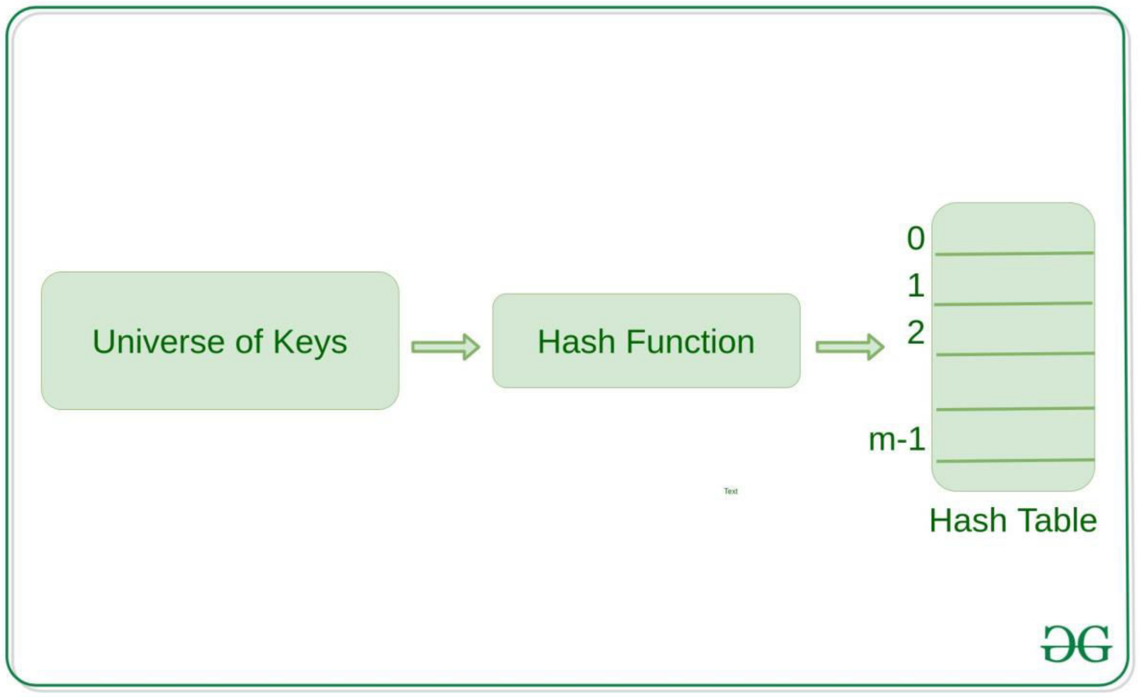
- Indexing in database
- Cryptography
- Symbol Tables in Compiler/Interpreter
- Dictionaries, caches, etc.

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Example:

```
h(large_value) = large_value % m
```

Here, h() is the required hash function and 'm' is the size of the hash table. For large values, hash functions produce value in a given range



How Hash Function Works?

- It should always map large keys to small keys.
- It should always generate values between 0 to m-1 where m is the size of the hash table.
- It should uniformly distribute large keys into hash table slots

Collision Handling:

If we know the keys beforehand, then we have can have perfect hashing. In perfect hashing, we do not have any collisions. However, If we do not know the keys, then we can use the following methods to avoid collisions:

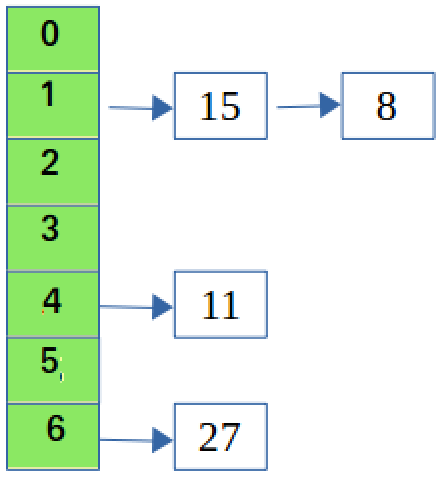
- **Chaining**
- **Open Addressing (Linear Probing, Quadratic Probing, Double Hashing)**

Chaining

While hashing, the hashing function may lead to a collision that is two or more keys are mapped to the same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)

Keys arrive in the Order (15, 11 , 27 , 8)



Note: In Linear Probing, whenever a collision occurs, we probe to the next empty slot. While in Quadratic Probing, whenever a collision occurs, we probe for i^2th slot in the ith iteration and we keep probing until an empty slot in the hashtable is found.

Performance of Hashing:

The performance of hashing is evaluated on the basis that each key is equally likely to be hashed for any slot of the hash table**.**

m = Length of Hash Table

n = Total keys to be inserted in the hash table

Load factor lf = n/m

Expected time to search = O(1 +lf)

Expected time to insert/delete = O(1 + lf)

The time complexity of search insert and delete is

O(1) if   lf is O(1)

Python Implementation of Hashing :

# Function to display hashtable

```
def display_hash(hashTable):

    for i in range(len(hashTable)):
        print(i, end = " ")
        for j in hashTable[i]:
            print("-->", end = " ")
            print(j, end = " ")
        print()
```

# Creating Hashtable as

# a nested list.

```
HashTable = [[] for _ in range(10)]
```

```
# Hashing Function to return
# key for every value.

def Hashing(keyvalue):
    return keyvalue % len(HashTable)



# Insert Function to add
# values to the hash table
def insert(Hashtable, keyvalue, value):
    
    hash_key = Hashing(keyvalue)
    Hashtable[hash_key].append(value)

# Driver Code
insert(HashTable, 10, 'Allahabad')
insert(HashTable, 25, 'Mumbai')
insert(HashTable, 20, 'Mathura')
insert(HashTable, 9, 'Delhi')
insert(HashTable, 21, 'Punjab')
insert(HashTable, 21, 'Noida')

display_hash (HashTable)
```

Output:

```
0 --> Allahabad --> Mathura
```

1 --> Punjab --> Noida

2

3

4

5 --> Mumbai

6

7

8

9 --> Delhi

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.

A Hash Function is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function **maps** a significant number or string to a small integer that can be used as the **index** in the hash table.

The pair is of the form **(key, value),** where for a given key, one can find a value using some kind of a “function” that maps keys to values. The key for a given object can be calculated using a function called a hash function. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i].

Types of Hash functions

There are many hash functions that use numeric or alphanumeric keys

1. Division Method.
2. Mid Square Method.
3. Folding Method.
4. Multiplication Method.

1. Division Method:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

$h(K) = k \bmod M$

Here,

k is the key value, and

M is the size of the hash table.

It is best suited that M is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division

Example:

$k = 12345$

$M = 95$

$h(12345) = 12345 \bmod 95$

= 90

k = 1276

M = 11

h(1276) = 1276 mod 11

= 0

Pros:

This method is quite good for any value of M.

The division method is very fast since it requires only a single division operation.

Cons:

This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.

Sometimes extra care should be taken to choose the value of M.

2.Mid Square Method:

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k2
2. Extract the middle r digits as the hash value.

Formula:

h(K) = h(k x k)

Here,

k is the key value.

The value of r can be decided based on the size of the table.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60

k x k = 60 x 60

= 3600

h(60) = 60

The hash value obtained is 60

Pros:

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

1. The size of the key is one of the limitations of this method, as the key is of big

size then its square will double the number of digits.

2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

3. Digit Folding Method:

This method involves two steps:

1. Divide the key-value k into a number of parts i.e. k1, k2, k3,….,kn, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

Here,

s is obtained by adding the parts of the key k

Example:

k = 12345

k1 = 12, k2 = 34, k3 = 5

s = k1 + k2 + k3

  = 12 + 34 + 5

  = 51

h(K) = 51

Note:

The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

4. Multiplication Method:

This method involves the following steps:

1. Choose a constant value A such that $0 < A < 1$.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

<u>Formula:</u>

$h(K) = \text{floor}(M (kA \bmod 1))$

Here,

M is the size of the hash table.

k is the key value.

A is a constant value.

Example:

$k = 12345$

A = 0.357840

M = 100

```
h(12345) = floor[ 100 (12345*0.357840 mod 1)]
         = floor[ 100 (4417.5348 mod 1) ]
         = floor[ 100 (0.5348) ]
         = floor[ 53.48 ]
         = 53
```

Pros:

The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

Cons:

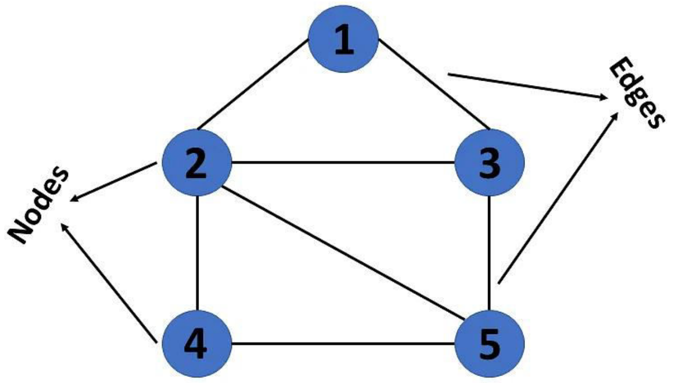
The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

Graph implementation in data structure

The graph data structure is used to store data required in computation to solve many computer programming problems. Graphs are used to address real-world problems in which the problem area is represented as a network, such as telephone networks, circuit networks, LinkedIn, Facebook, etc.

In computer science, The graph is an abstract data type used to implement the undirected and directed graph notions from graph theory in mathematics.

A graph data structure is made up of a finite and potentially mutable set of vertices (also known as nodes or points), as well as a set of unordered pairs for an undirected graph or a set of ordered pairs for a directed graph. These pairs are recognized as edges, links, or lines in a directed graph but are also known as arrows or arcs. The vertices could be internal graph elements or external items represented by integer indices or references.



A graph is a non-linear kind of data structure made up of nodes or vertices and edges. The edges connect any two nodes in the graph, and the nodes are also known as vertices.

This graph has a set of vertices V= { 1,2,3,4,5} and a set of edges E= { (1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(4,50 }.

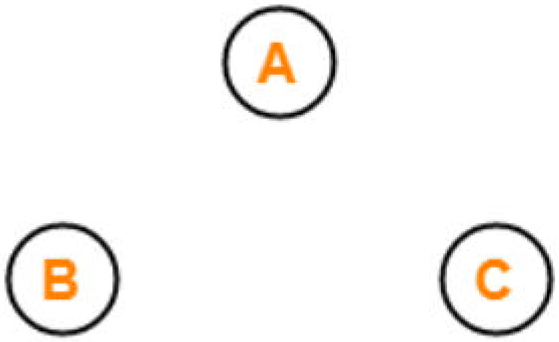
So depending upon the position of these nodes and vertices, there are different types of graphs, such as

1. Null Graph
2. Trivial Graph
3. Non-directed Graph
4. Directed Graph
5. Connected Graph
6. Disconnected Graph
7. Regular Graph

8. Complete Graph
9. Cycle Graph
10. Cyclic Graph
11. Acyclic Graph
12. Finite Graph
13. Infinite Graph
14. Bipartite Graph
15. Planar Graph
16. Simple Graph
17. Multi Graph
18. Pseudo Graph
19. Euler Graph
20. Hamiltonian Graph

Null Graph

The Null Graph is also known as the order zero graph. The term "null graph" refers to a graph with an empty edge set. In other words, a null graph has no edges, and the null graph is present with only isolated vertices in the graph.



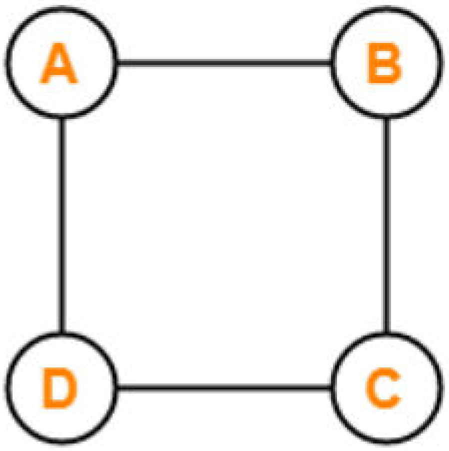
The image displayed above is a null or zero graphs because it has zero edges between the three vertices of the graph.

Trivial Graph

A graph is called a trivial graph if it has only one vertex present in it. The trivial graph is the smallest possible graph that can be created with the least number of vertices that is one vertex only.

Non-Directed Graph

A graph is called a non-directed graph if all the edges present between any graph nodes are non-directed. By non-directed edges, we mean the edges of the graph that cannot be determined from the node it is starting and at which node it is ending. All the edges for a graph need to be non-directed to call it a non-directed graph. All the edges of a non-directed graph don't have any direction.

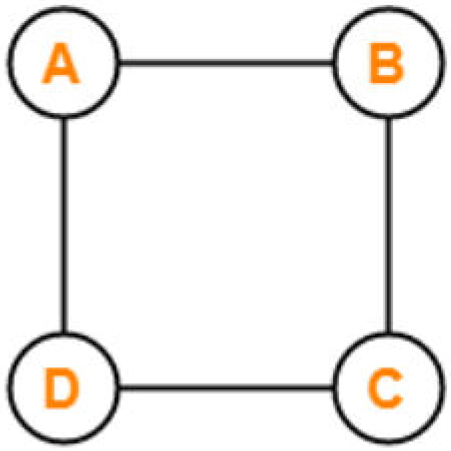


The graph that is displayed above is an example of a disconnected graph. This graph is called a disconnected graph because there are four vertices named vertex A, vertex B, vertex C, and vertex D. There are also exactly four edges between these vertices of the graph. And all the vertices that are present between the different nodes of the graph are not directed, which means the edges don't have any specific direction.

For example, the edge between vertex A and vertex B doesn't have any direction, so we cannot determine whether the edge between vertex A and vertex B starts from vertex A or vertex B. Similarly, we can't determine the ending vertex of this edge between these nodes.

Directed Graph

Another name for the directed graphs is digraphs. A graph is called a directed graph or digraph if all the edges present between any vertices or nodes of the graph are directed or have a defined direction. By directed edges, we mean the edges of the graph that have a direction to determine from which node it is starting and at which node it is ending.



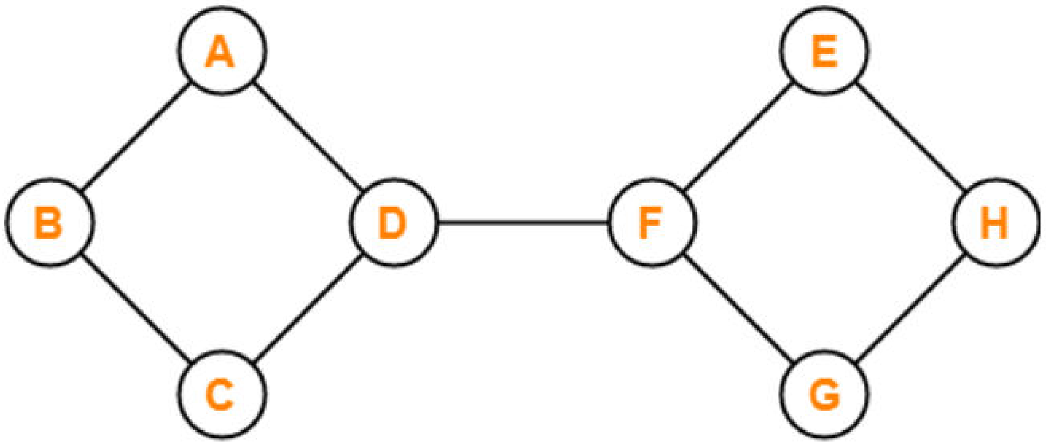
All the edges for a graph need to be directed to call it a directed graph or digraph. All the edges of a directed graph or digraph have a direction that will start from one vertex and end at another.

The graph that is displayed above is an example of a connected graph. This graph is called a connected graph because there are four vertices in the graph named vertex A, vertex B, vertex C, and vertex D. There are also exactly four edges between these vertices of the graph and all the vertices that are present between the different nodes of the graph are directed (or pointing to some of the vertices) which means the edges have a specific direction assigned to them.

For example, consider the edge that is present between vertex D and vertex A. This edge shows that an arrowhead is pointing towards vertex A, which means this edge starts from vertex D and ends at vertex A.

Connected Graph

For a graph to be labelled as a connected graph, there must be at least a single path between every pair of the graph's vertices. In other words, we can say that if we start from one vertex, we should be able to move to any of the vertices that are present in that particular graph, which means there exists at least one path between all the vertices of the graph.



The graph shown above is an example of a connected graph because we start from any one of the vertices of the graph and start moving towards any other remaining vertices of the graph. There will exist at least one path for traversing the graph.

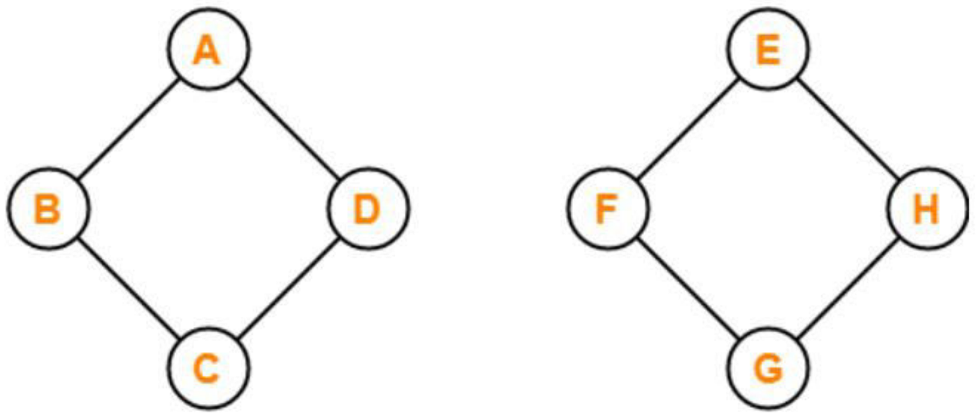
For example, if we begin from vertex B and traverse to vertex H, there are various paths for traversing. One of the paths is

Vertice B -> vertice C -> vertice D -> vertice F -> vertice E -> vertice H

Similarly, there are other paths for traversing the graph from vertex B to vertex H. there is at least one path between all the graph nodes. In other words, we can say that all the vertices or nodes of the graph are connected to each other via edge or number of edges.

Disconnected Graph

A graph is said to be a disconnected graph where there does not exist any path between at least one pair of vertices. In other words, we can say that if we start from any one of the vertices of the graph and try to move to the remaining present vertices of the graph and there exists not even a single path to move to that vertex, then it is the case of the disconnected graph. If any one of such a pair of vertices doesn't have a path between them, it is called a disconnected graph.

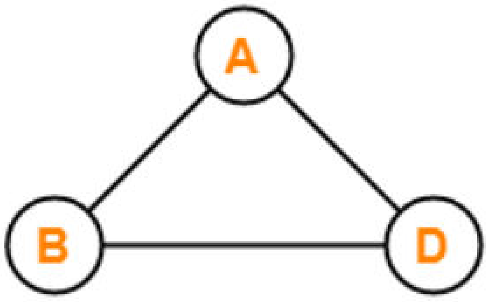


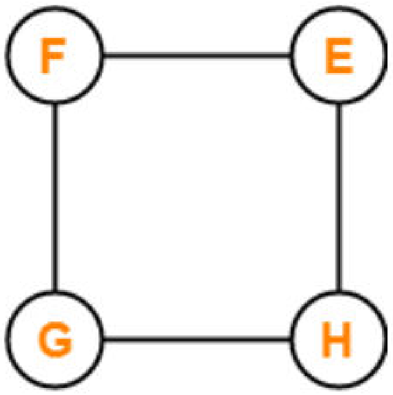
The graph shown above is a disconnected graph. The above graph is called a disconnected graph because at least one pair of vertices doesn't have a path to traverse starting from either node.

For example, a single path between both vertices doesn't exist if we want to traverse from vertex A to vertex G. In other words, we can say that all the vertices or nodes of the graph are not connected to each other via edge or number of edges so that they can be traversed.

Regular Graph

For a graph to be called a regular, it should satisfy one primary condition: all graph vertices should have the same degree. By the degree of vertices, we mean the number of nodes associated with a particular vertex. If all the graph nodes have the same degree value, then the graph is called a regular graph. If all the vertices of a graph have the degree value of 6, then the graph is called a 6-regular graph. If all the vertices in a graph are of degree 'k', then it is called a "k-regular graph".





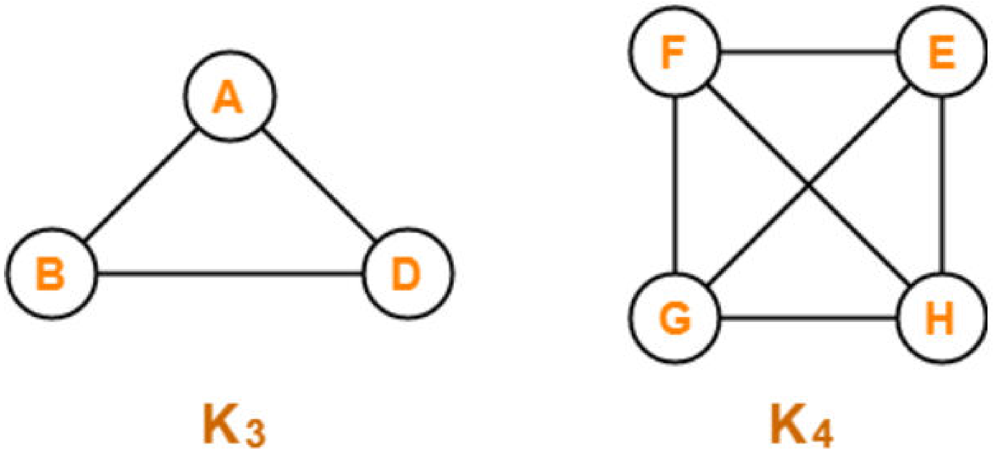
The graphs that are displayed above are regular graphs. In graph 1, there are three vertices named vertex A, vertex B, and vertex C, All the vertices in graph 1, have the degree of each node as 2. The degree of each vertex is calculated by counting the number of edges connected to that particular vertex.

For vertex A in graph 1, there are two edges associated with vertex A, one from vertex B and another from vertex D. Thus, the degree of vertex A of graph one is 2. Similarly, for other vertices of the graph, there are only two edges associated with each vertex, vertex B and vertex D. Therefore, vertex B and vertex D are 2. As the degree of all the three nodes of the graph is the same, that is 2. Therefore, this graph is called a 2-regular graph.

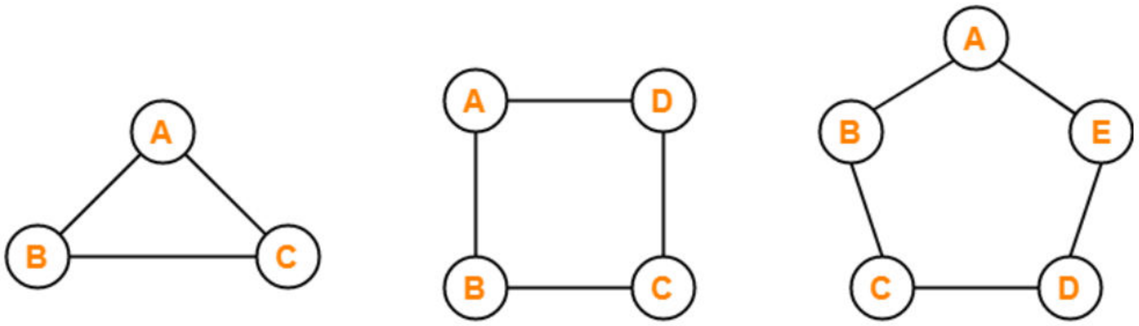
Similarly, for the second graph shown above, there are four vertices named vertex E, vertex F, vertex G, and vertex F. The degree of all the four vertices of this graph is 2. Each vertex of the graph is 2 because only two edges are associated with all of the graph's vertices. As all the nodes of this graph have the same degree of 2, this graph is called a regular graph.

Complete Graph

A graph is said to be a complete graph if, for all the vertices of the graph, there exists an edge between every pair of the vertices. In other words, we can say that all the vertices are connected to the rest of all the vertices of the graph. A complete graph of 'n' vertices contains exactly nC2 edges, and a complete graph of 'n' vertices is represented as Kn.

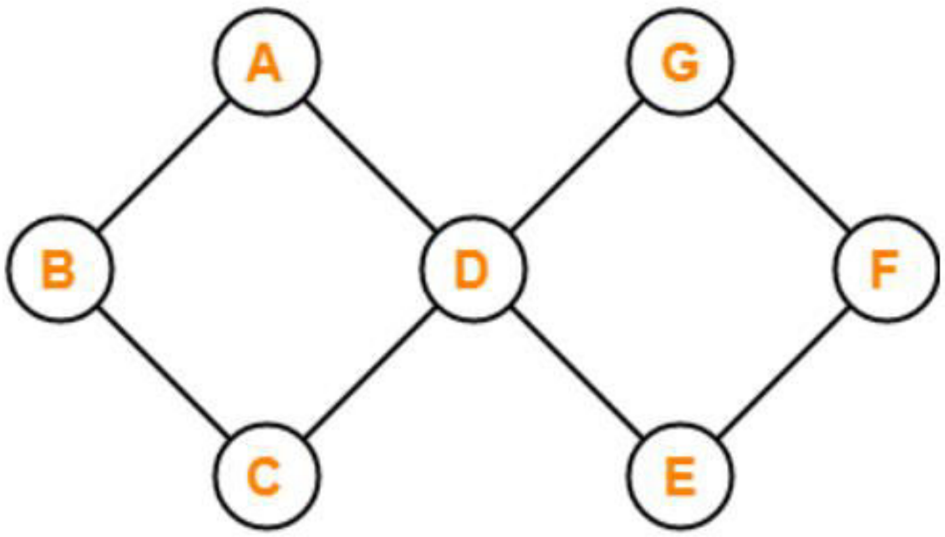


There are two graphs name K3 and K4 shown in the above image, and both graphs are complete graphs. Graph K3 has three vertices, and each vertex has at least one edge with the rest of the vertices. Similarly, for graph K4, there are four nodes named vertex E, vertex F, vertex G, and vertex H. For example, the vertex F has three edges connected to it to connect it to the respective three remaining vertices of the graph. Likewise, for the other three reaming vertices, there are three edges associated with each one of them. As all the vertices of this graph have a separate edge for other vertices, it is called a complete graph.



Cyclic Graph

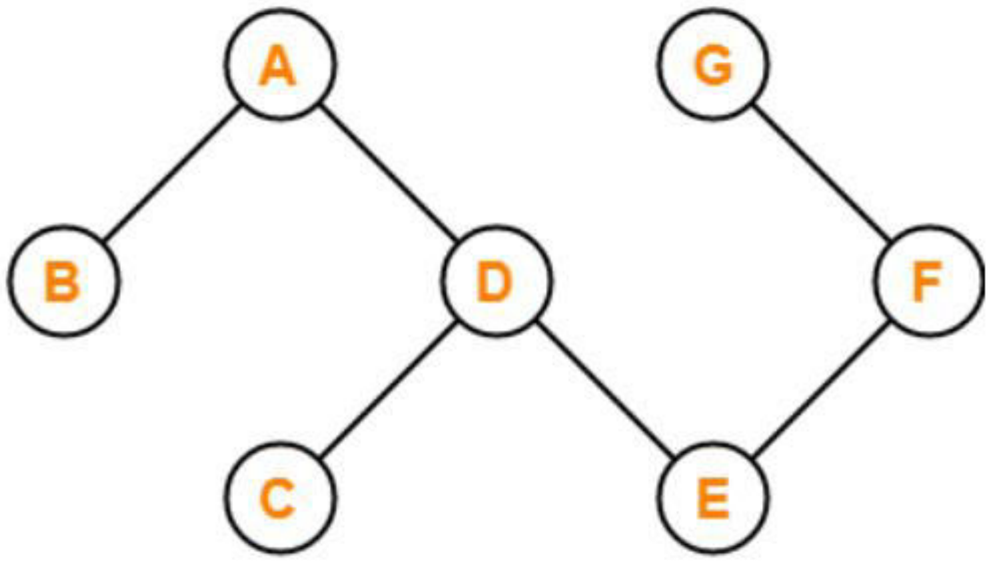
For a graph to be called a cyclic graph, it should consist of at least one cycle. If a graph has a minimum of one cycle present, it is called a cyclic graph.



The graph shown in the image has two cycles present, satisfying the required condition for a graph to be cyclic, thus making it a cyclic graph.

Acyclic Graph

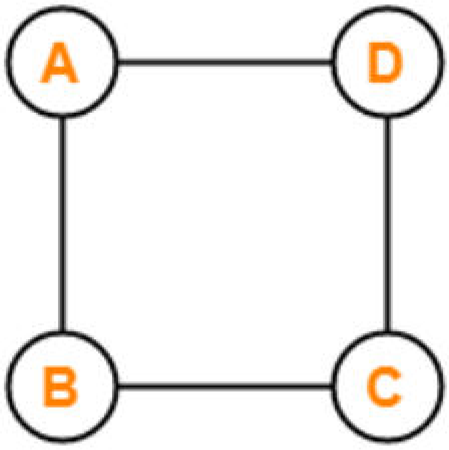
A graph is called an acyclic graph if zero cycles are present, and an acyclic graph is the complete opposite of a cyclic graph.



The graph shown in the above image is acyclic because it has zero cycles present in it. That means if we begin traversing the graph from vertex B, then a single path doesn't exist that will traverse all the vertices and end at the same vertex that is vertex B.

Finite Graph

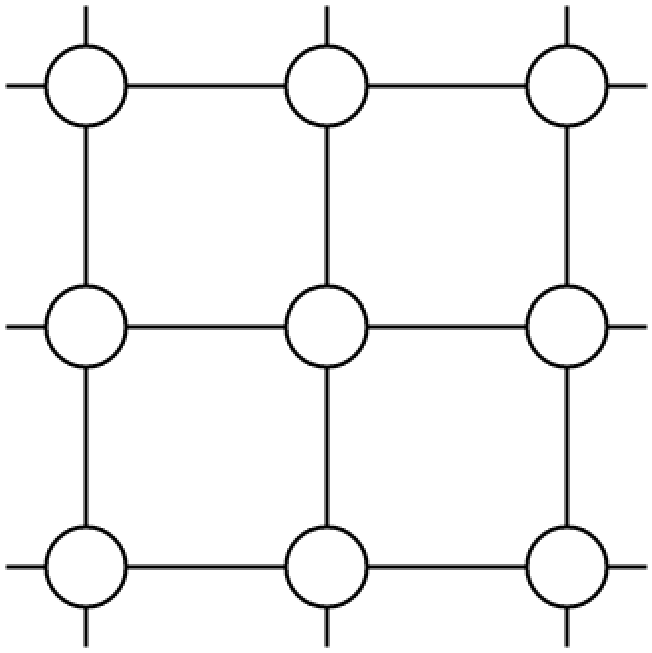
If the number of vertices and the number of edges that are present in a graph are finite in number, then that graph is called a finite graph.



The graph shown in the above image is the finite graph. There are four vertices named vertex A, vertex B, vertex C, and vertex D, and the number of edges present in this graph is also four, as both the number of nodes and vertices of this graph is finite in number it is called a finite graph.

Infinite Graph

If the number of vertices in the graph and the number of edges in the graph are infinite in number, that means the vertices and the edges of the graph cannot be counted, then that graph is called an infinite graph.



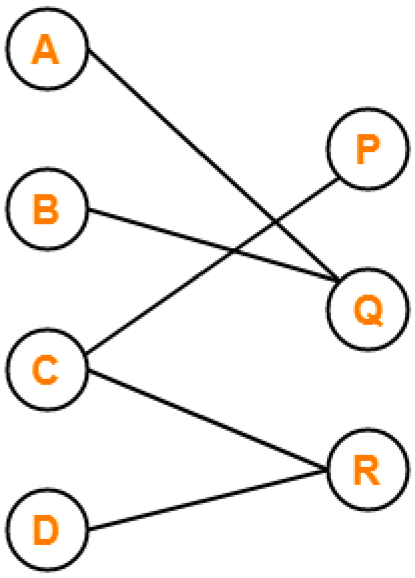
.

As we can see in the above image, the number of vertices in the graph and the number of edges in the graph are infinite, so this graph is called an infinite graph.

Bipartite Graph

For a graph to be a Bipartite graph, it needs to satisfy some of the basic preconditions. These conditions are:

1. All the vertices of the graph should be divided into two distinct sets of vertices X and Y.
2. All the vertices present in the set X should only be connected to the vertices present in the set Y with some edges. That means the vertices present in a set should not be connected to the vertex that is present in the same set.
3. Both the sets that are created should be distinct that means both should not have the same vertices in them.



The graph shown in the above image is divided into two vertices named set X and set Y. The contents of these sets are,

Set X = {vertex A, vertex B, vertex C, vertex D}

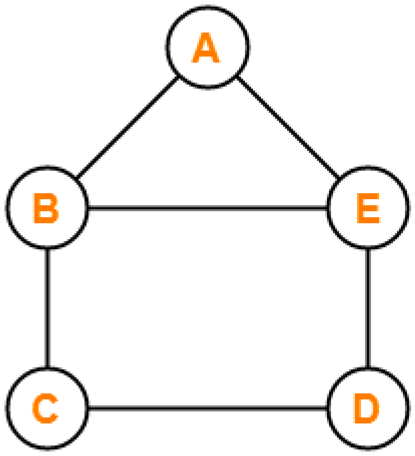
Set Y = {vertex P, vertex Q, vertex R}

The vertex A of the set X is associated with the vertex Q of the set Y. And the vertex B is also connected to the vertex Q. The vertex C of the set X is connected to the two vertices of the set Y named vertex P and vertex R. The vertex D of the set X is associated with the vertex Q of the set R.

Similarly, all the vertices present in the set Y are only connected to the vertices from the set X. And both set X and set Y have non-repeating or distinct elements present in them. The graph shown in the above image satisfies all the conditions for the Bipartite graph, and thus it is a Bipartite graph.

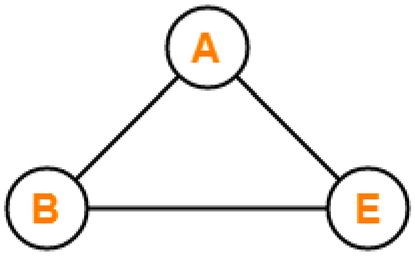
Planar Graph

A graph is called a planar graph if that graph can be drawn in a single plane with any two of the edges intersecting each other.



Simple Graph

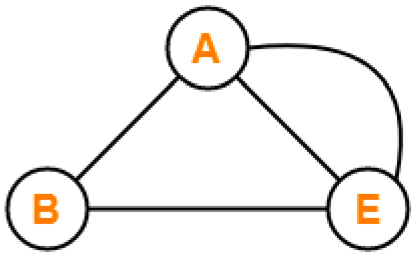
A graph is said to be a simple graph if the graph doesn't consist of no self-loops and no parallel edges in the graph.



We have three vertices and three edges for the graph that is shown in the above image. This graph has no self-loops and no parallel edges; therefore, it is called a simple graph.

Multi Graph

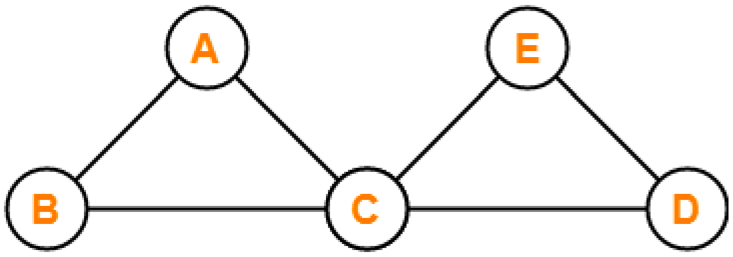
A graph is said to be a multigraph if the graph doesn't consist of any self-loops, but parallel edges are present in the graph. If there is more than one edge present between two vertices, then that pair of vertices is said to be having parallel edges.



We have three vertices and three edges for the graph that is shown in the above image. There are no self-loops, but two edges connect these two vertices between vertex A and vertex E of the graph. In other words, we can say that if two vertices of a graph are connected with more than one edge in a graph, then it is said to be having parallel edges, thus making it a multigraph.

Euler Graph

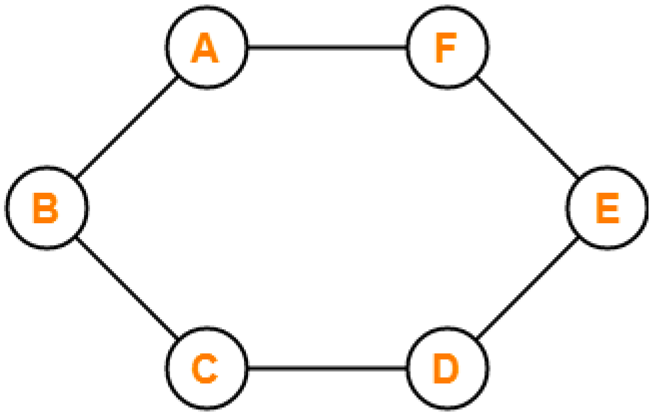
If all the vertices present in a graph have an even degree, then the graph is known as an Euler graph. By degree of a vertex, we mean the number of edges that are associated with a vertex. So for a graph to be an Euler graph, it is required that all the vertices in the graph should be associated with an even number of edges.



In the graph shown in the above image, we have five vertices named vertex A, vertex B, vertex C, vertex D and vertex E. All the vertices except vertex C have a degree of 2, which means they are associated with two edges each of the vertex. At the same time, vertex C is associated with four edges, thus making it degree 4. The degree of vertex C and other vertices is 4 and 2, respectively, which are even. Therefore, the graph displayed above is an Euler graph.

Hamilton Graph

Suppose a closed walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges. Such a graph is called a Hamiltonian graph, and such a walk is called a Hamiltonian path. The Hamiltonian circuit is also known as Hamiltonian Cycle



In other words, A Hamiltonian path that starts and ends at the same vertex is called a Hamiltonian circuit. Every graph that contains a Hamiltonian circuit also contains a Hamiltonian path, but vice versa is not true. There may exist more than one Hamiltonian path and Hamiltonian circuit in a graph.

The graph shown in the above image consists of a closed path ABCDEFA which starts from vertex A and traverses all other vertices or nodes without traversing any of the nodes twice other than vertex A in the path of traversal. Therefore, the graph shown in the above image is a Hamilton graph

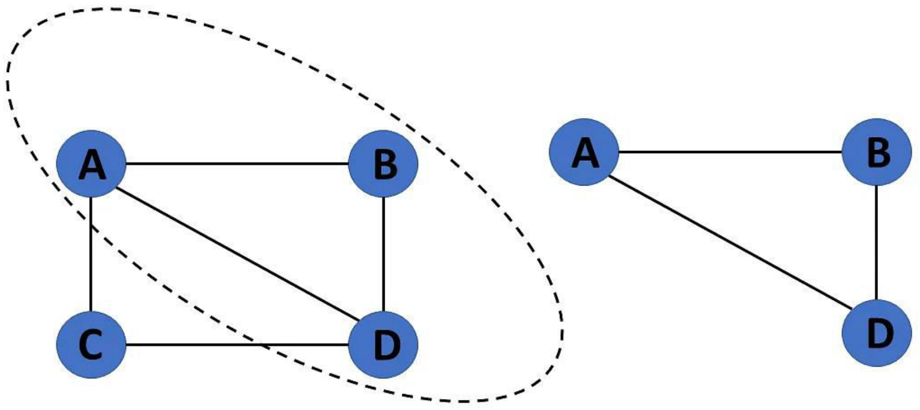
Terminologies of Graphs in Data Structures

Following are the basic terminologies of graphs in data structures:

1. An edge is one of the two primary units used to form graphs. Each edge has two ends, which are vertices to which it is attached.
2. If two vertices are endpoints of the same edge, they are adjacent.
3. A vertex's outgoing edges are directed edges that point to the origin.
4. A vertex's incoming edges are directed edges that point to the vertex's destination.
5. The total number of edges occurring to a vertex in a graph is its degree.
6. The out-degree of a vertex in a directed graph is the total number of outgoing edges, whereas the in-degree is the total number of incoming edges.
7. A vertex with an in-degree of zero is referred to as a source vertex, while one with an out-degree of zero is known as sink vertex.
8. An isolated vertex is a zero-degree vertex that is not an edge's endpoint.
9. A path is a set of alternating vertices and edges, with each vertex connected by an edge.
10. The path that starts and finishes at the same vertex is known as a cycle.
11. A path with unique vertices is called a simple path.
12. For each pair of vertices x, y, a graph is strongly connected if it contains a directed path from x to y and a directed path from y to x.
13. A directed graph is weakly connected if all of its directed edges are replaced with undirected edges, resulting in a connected graph. A weakly linked graph's vertices have at least one out-degree or in-degree.
14. A tree is a connected forest. The primary form of the tree is called a rooted tree, which is a free tree.
15. A spanning subgraph that is also a tree is known as a spanning tree.

    The vertices and edges of a graph that are subsets of another graph are known

    a subgraph.



16. A connected component is the unconnected graph's most connected subgraph.
17. A bridge, which is an edge of removal, would sever the graph.
18. Forest is a graph without a cycle.

Representation of Graphs in Data Structures

Graphs in data structures are used to represent the relationships between objects. Every graph consists of a set of points known as vertices or nodes connected by lines known as edges. The vertices in a network represent entities.

The most frequent graph representations are the two that follow:

1. Adjacency matrix
2. Adjacency list

Adjacency Matrix

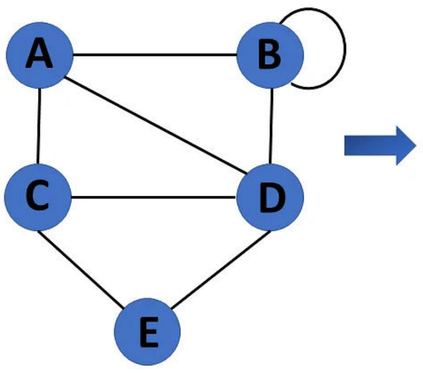
A sequential representation is an adjacency matrix.

It's used to show which nodes are next to one another. I.e., is there any connection between nodes in a graph?

You create an MXM matrix G for this representation. If an edge exists between vertex a and vertex b, the corresponding element of G, gi,j = 1, otherwise gi,j = 0.

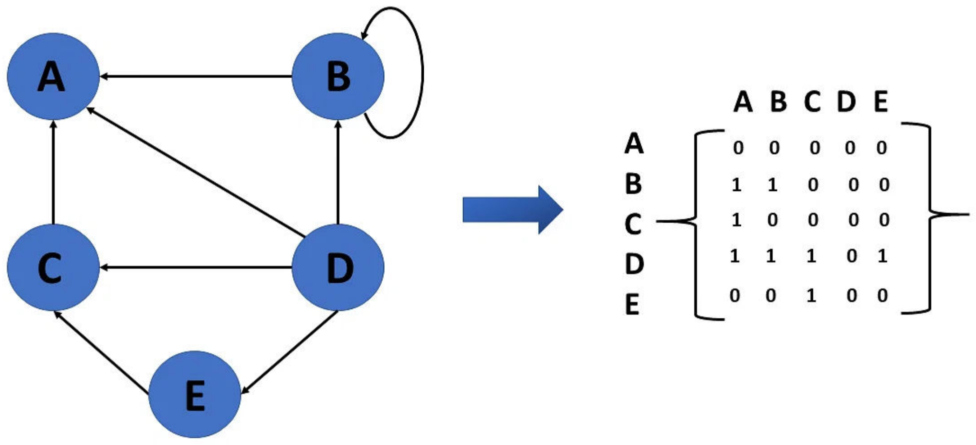
If there is a weighted graph, you can record the edge's weight instead of 1s and 0s.

Undirected Graph Representation



$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 1 & 0 \\
B & 1 & 1 & 0 & 1 & 0 \\
C & 1 & 0 & 0 & 1 & 1 \\
D & 1 & 1 & 1 & 0 & 1 \\
E & 0 & 0 & 1 & 1 & 0
\end{array}$$

Directed Graph Representation

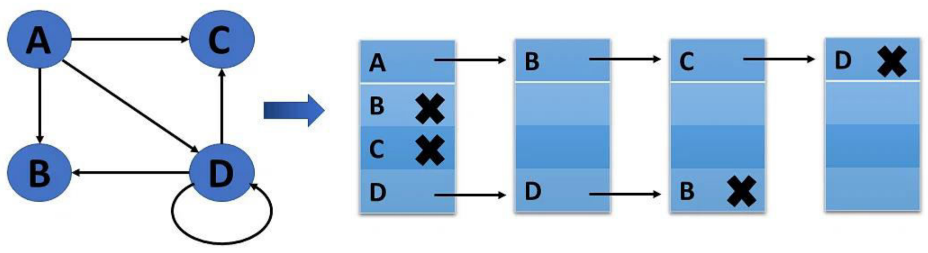


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 0 & 0 & 0 & 0 \\ B & 1 & 1 & 0 & 0 & 0 \\ C & 1 & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 1 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

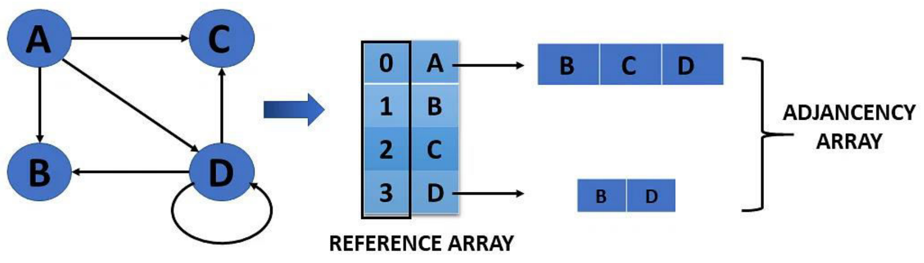
Adjacency List

1. A linked representation is an adjacency list.
2. You keep a list of neighbors for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighboring vertices.
3. You have an arra of vertices indexed by the vertex number, and the corresponding array member for each vertex x points to a singly linked list of x's neighbors.

Weighted Undirected Graph Representation Using Linked-List



Weighted Undirected Graph Representation Using an .Array



Operations on Graphs in Data Structures

The operations you perform on the graphs in data structures are listed below:

- Creating graphs

- Insert vertex
- Delete vertex
- Insert edge
- Delete edge

Creating Graphs

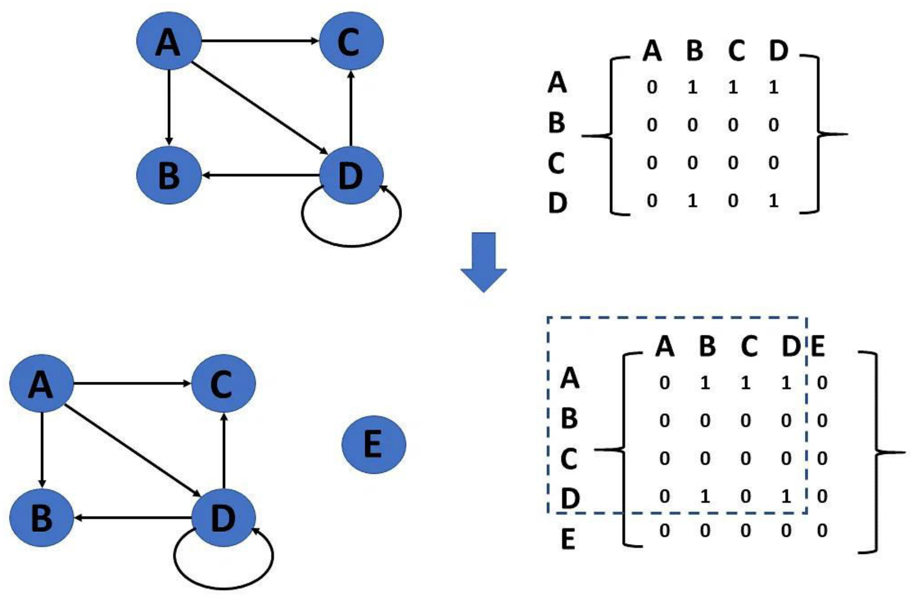
There are two techniques to make a graph:

1. Adjacency Matrix

The adjacency matrix of a simple labeled graph, also known as the connection matrix, is a matrix with rows and columns labeled by graph vertices and a 1 or 0 in position depending on whether they are adjacent or not.

2. Adjacency List

A finite graph is represented by an adjacency list, which is a collection of unordered lists. Each unordered list describes the set of neighbors of a particular vertex in the graph within an adjacency list.

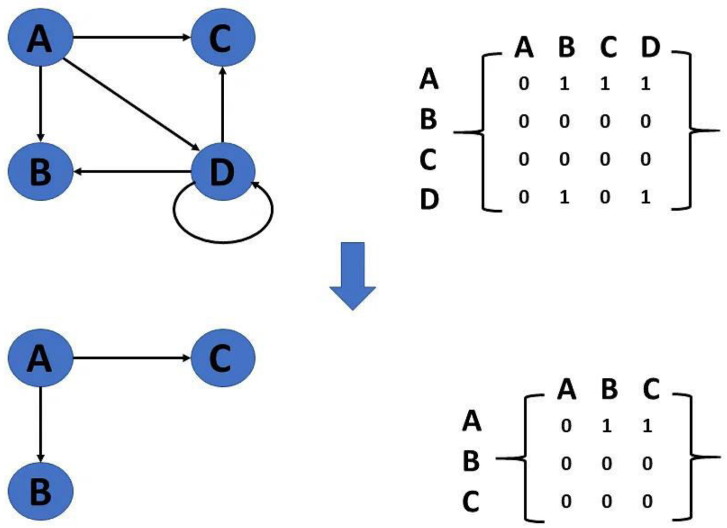


$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & 1 \end{array}$$

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 0 & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & 1 & 0 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

Delete Vertex

- Deleting a vertex refers to removing a specific node or vertex from a graph that has been saved.
- If a removed node appears in the graph, the matrix returns that node. If a deleted node does not appear in the graph, the matrix returns the node not available.

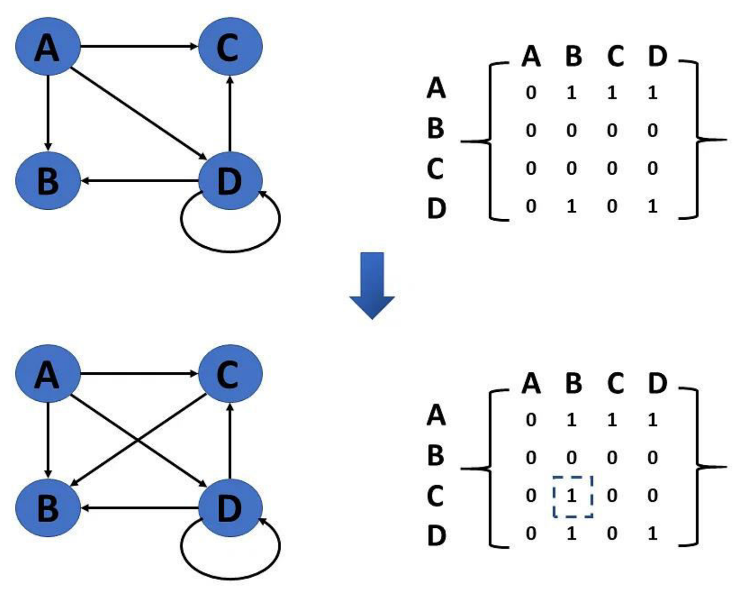


$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & 1 \end{array}$$

$$\begin{array}{c|ccc} & A & B & C \\ \hline A & 0 & 1 & 1 \\ B & 0 & 0 & 0 \\ C & 0 & 0 & 0 \end{array}$$

Insert Edge

Connecting two provided vertices can be used to add an edge to a graph.

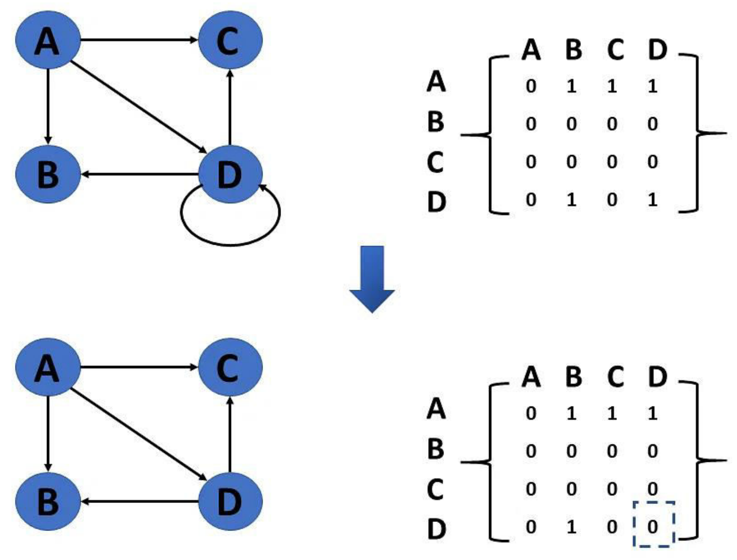


$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & 1 \end{array}$$

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 0 & 0 \\ C & 0 & \boxed{1} & 0 & 0 \\ D & 0 & 1 & 0 & 1 \end{array}$$

Delete Edge

The connection between the vertices or nodes can be removed to delete an edge.



$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & 1 \end{array}$$

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & \boxed{0} \end{array}$$