# Asymptotic Analysis: Big-O Notation and More

In this tutorial, you will learn what asymptotic notations are. Also, you will learn about Big-O notation, Theta notation and Omega notation.

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

---

## Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

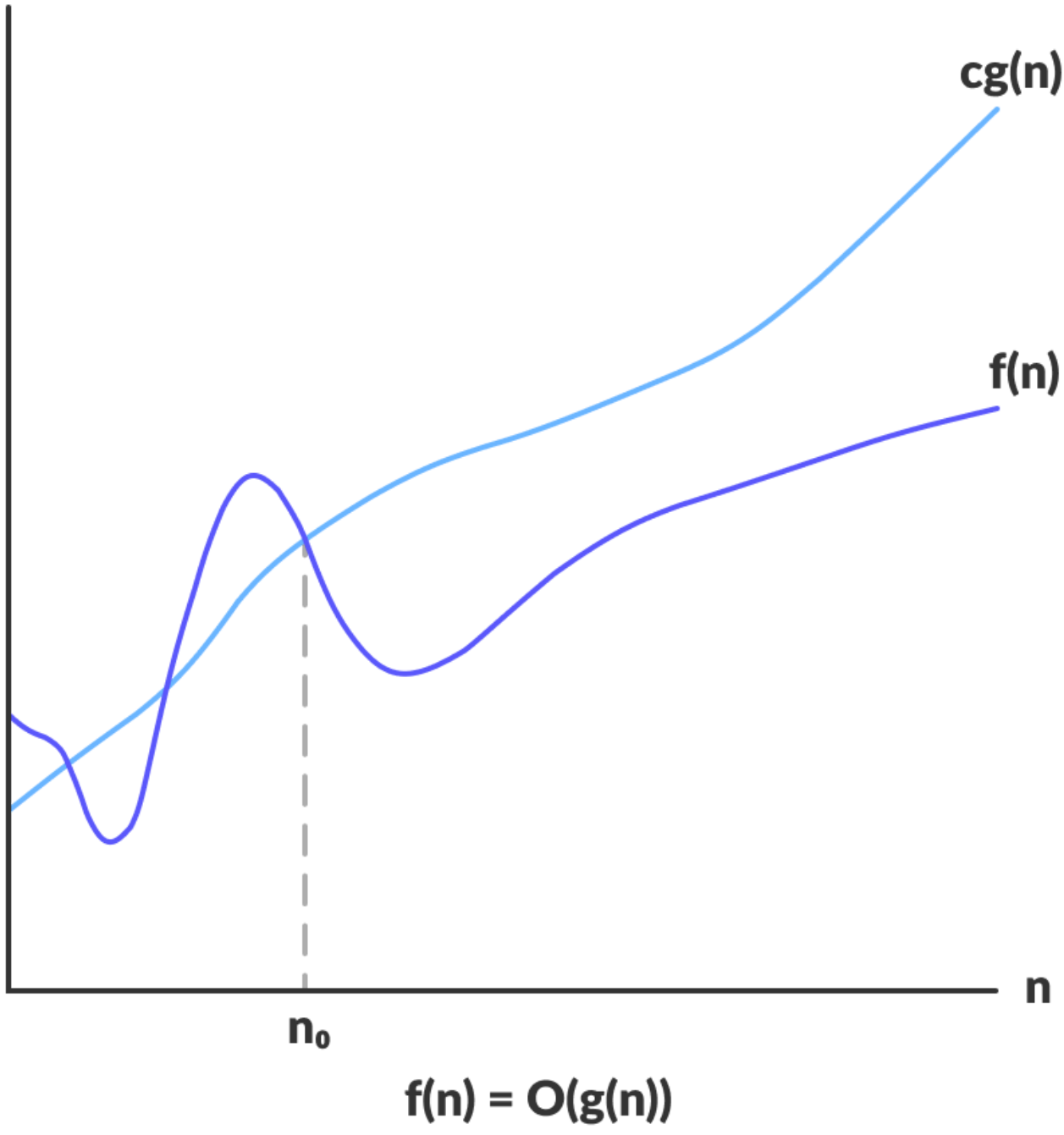
There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

---

## Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.




Big-O gives the upper bound of a function

```
O(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }
```

The above expression can be described as a function `f(n)` belongs to the set `O(g(n))` if there exists a positive constant `c` such that it lies between `0` and `cg(n)`, for sufficiently large `n`.

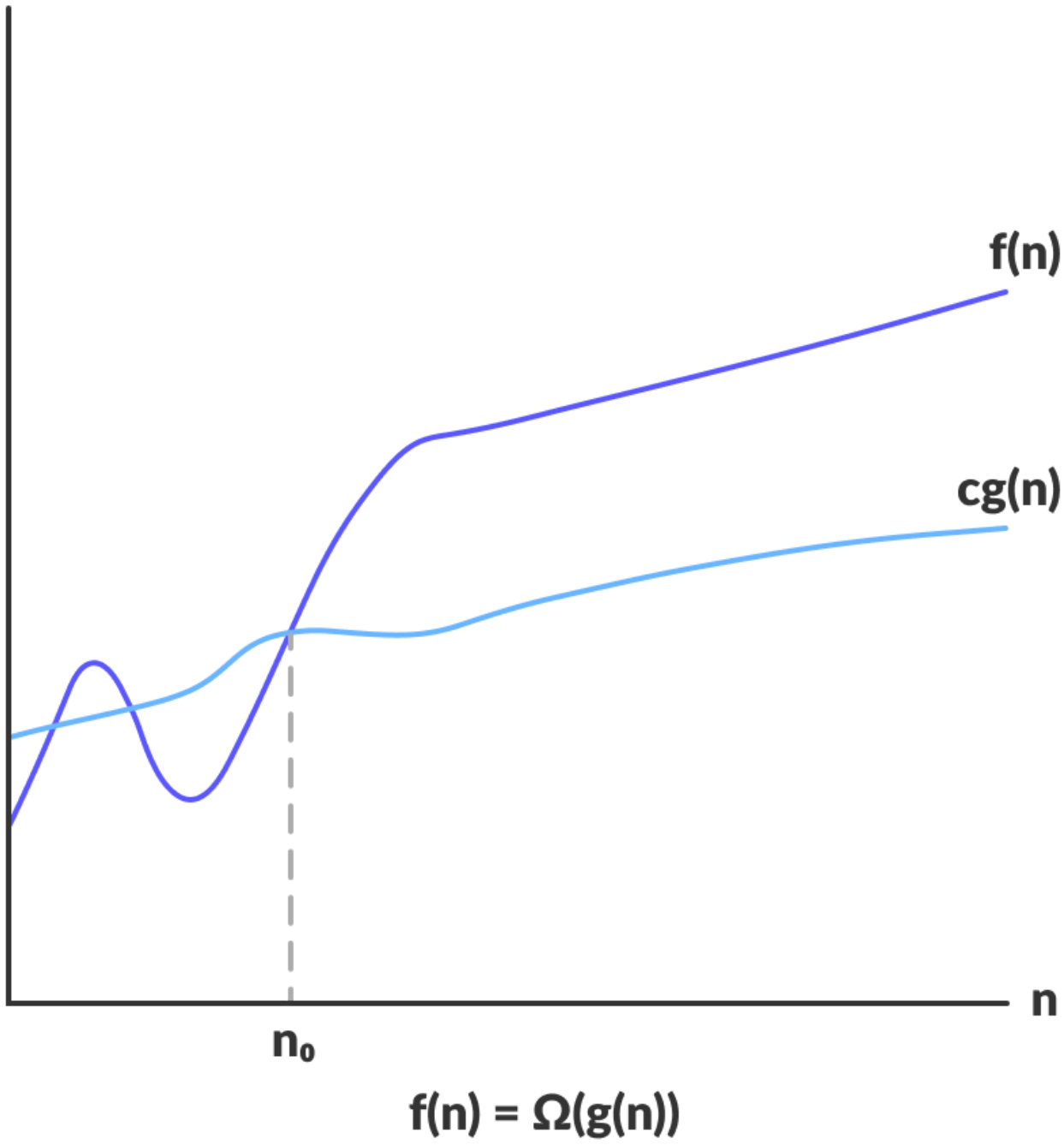
For any value of `n`, the running time of an algorithm does not cross the time provided by `O(g(n))`.

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

---

## Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.




Omega gives the lower bound of a function

```
Ω(g(n)) = { f(n): there exist positive constants c and n₀
            such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n₀ }
```

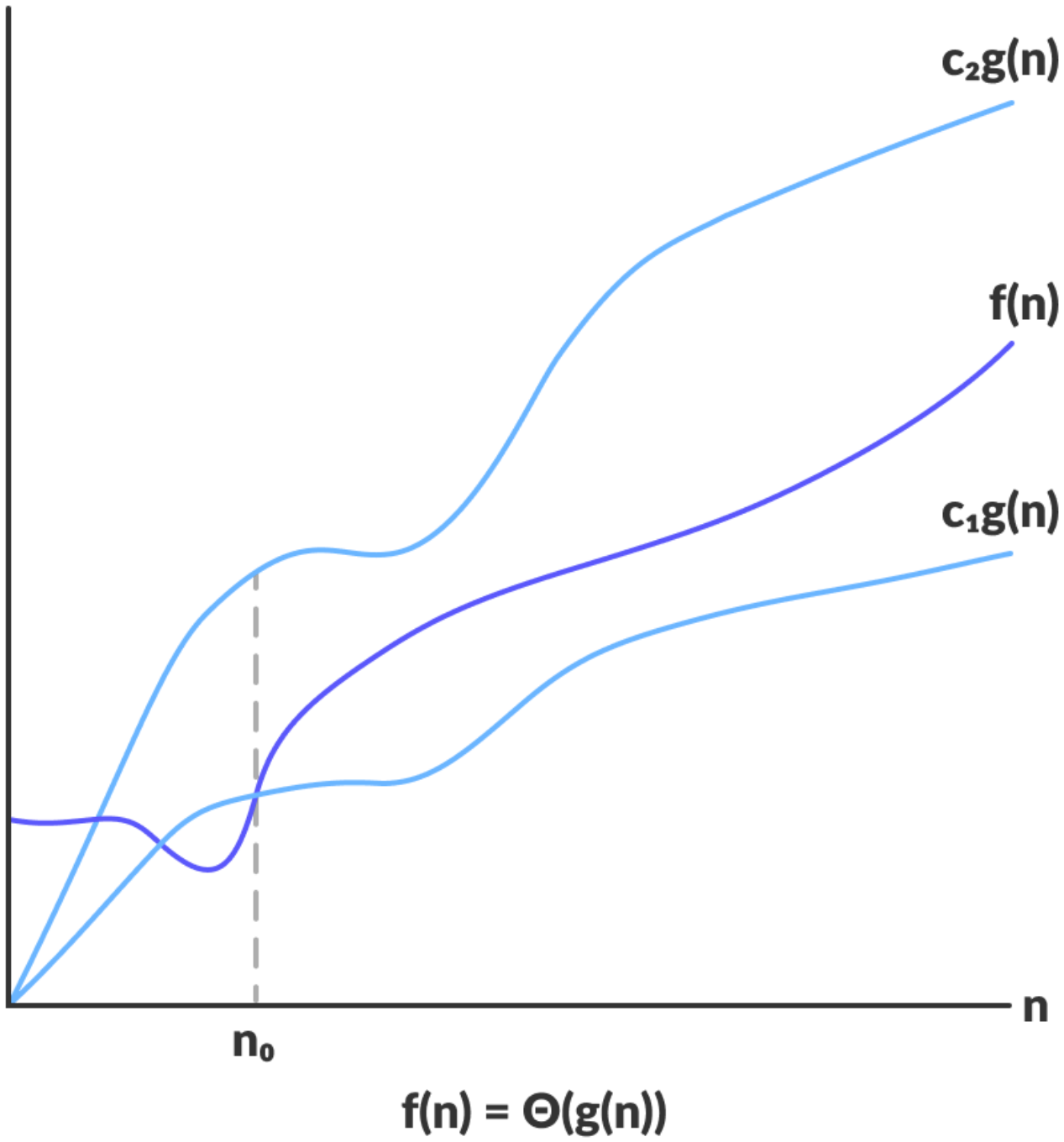
The above expression can be described as a function `f(n)` belongs to the set `Ω(g(n))` if there exists a positive constant `c` such that it lies above `cg(n)`, for sufficiently large `n`.

For any value of `n`, the minimum time required by the algorithm is given by Omega `Ω(g(n))`.

---

## Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.




Theta bounds the function within constants factors

For a function `g(n)`, `Θ(g(n))` is given by the relation:

```
Θ(g(n)) = { f(n): there exist positive constants c₁, c₂ and n₀
            such that 0 ≤ c₁g(n) ≤ f(n) ≤ c₂g(n) for all n ≥ n₀ }
```

The above expression can be described as a function $f(n)$ belongs to the set $\theta(g(n))$ if there exist positive constants $c_1$ and $c_2$ such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

If a function $f(n)$ lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n0$, then $f(n)$ is said to be asymptotically tight bound.

# HASHING

- Time complexity of Linear Search is O(n) and of Binary Search is $O(\log_2 n)$.
- Objective is to achieve O(1) time complexity so that searching doesn’t depend on the size of the input i.e. the number of elements in the array.

**Hashing** is an approach in which time required to search an element doesn't depend on the total number of elements. Using hashing data structure, a given element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

**Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.**

Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.

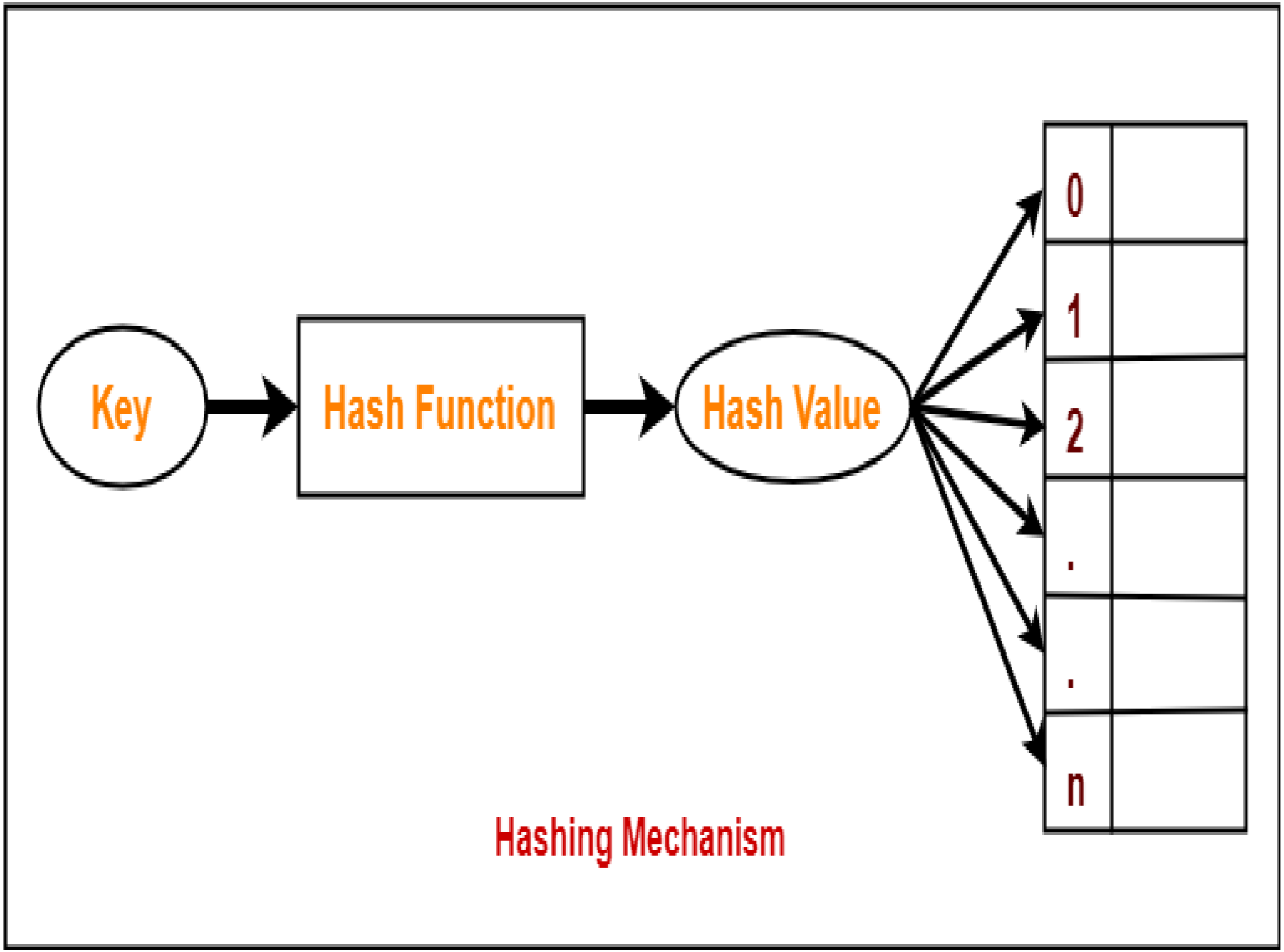
In this data structure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value. The hash key value is used to map the data with an index in the hash table. And the hash key is generated for every data using a **hash function**. That means every entry in the hash table is based on the hash key value generated using the hash function.

**Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1)).**

Hash tables are used to perform insertion, deletion and search operations very quickly in a data structure. Using hash table concept, insertion, deletion, and search operations are accomplished in constant time complexity. Generally, every hash table makes use of a function called **hash function** to map the data into the hash table.

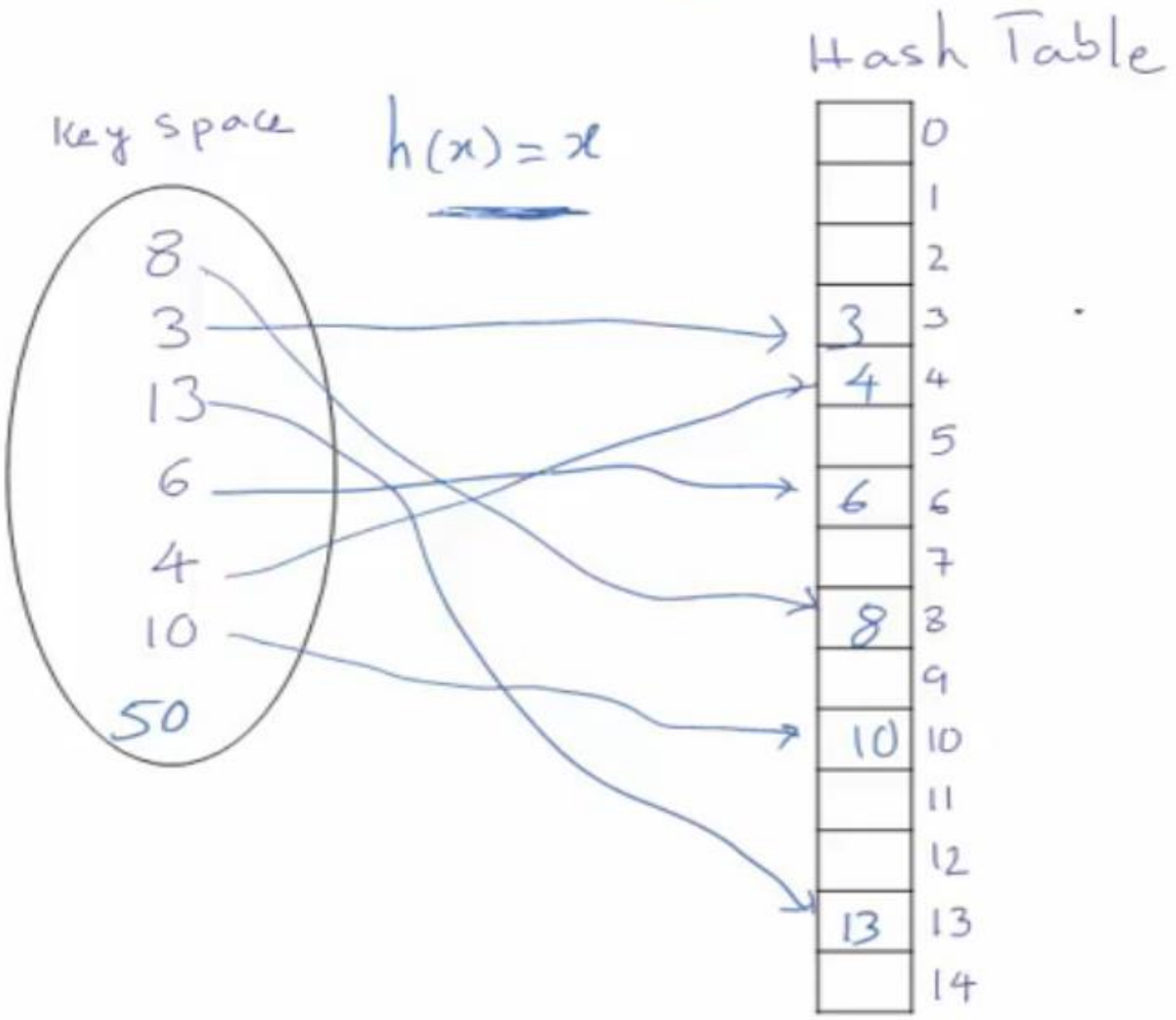
A hash function is defined as follows...

**Hash function** is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.




Hashing Mechanism

# Hashing


Hash Table
key space
h(x) = x
8
3
13
6
4
10
50
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
3
4
6
8
10
13

# Types of Hash Functions

- Division Hash Function
- Mid Square Hash Function
- Folding Hash Function etc.

## Hashing Function Methods (Hashing Methods)

- **Division Hash Method**
  - The key K is divided by some number m and the remainder is used as the hash address of K.
    - $h(k)=k \bmod m$
  - This gives the indexes in the range 0 to m-1 so the hash table should be of size m
  - This is an example of uniform hash function if value of m will be chosen carefully.
  - Generally a prime number is a best choice which will spread keys evenly.
  - A uniform hash function is designed to distribute the keys roughly evenly into the available positions within the array (or hash table).

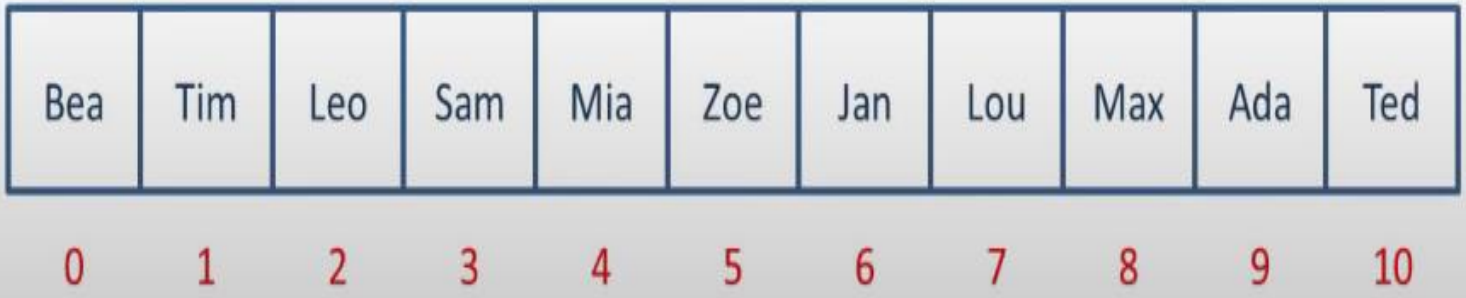
# Hashing Algorithm

- Calculation applied to a key to transform it into an address
- For numeric keys, divide the key by the number of available addresses, *n*, and take the remainder

address = *key* Mod *n*

- For alphanumeric keys, divide the sum of ASCII codes in a key by the number of available addresses, *n*, and take the remainder

Index number = *sum ASCII codes* Mod *size of array*



| | | | | | | | Sum of ASCII codes | Index=Sum%sizeofArray(11 here) |
|---|---|---|---|---|---|---|---|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Jan | J | 74 | a | 97 | n | 110 | 281 | 6 |
| Ada | A | 65 | d | 100 | a | 97 | 262 | 9 |
| Leo | L | 76 | e | 101 | o | 111 | 288 | 2 |
| Sam | S | 83 | a | 97 | m | 109 | 289 | 3 |
| Lou | L | 76 | o | 111 | u | 117 | 304 | 7 |
| Max | M | 77 | a | 97 | x | 120 | 294 | 8 |
| Ted | T | 84 | e | 101 | d | 100 | 285 | 10 |

| Bea | Tim | Leo | Sam | Mia | Zoe | Jan | Lou | Max | Ada | Ted |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Find Ada  262 Mod 11 = 9


myData = Array(9)

| Bea | Tim | Leo | Sam | Mia | Zoe | Jan | Lou | Max | Ada | Ted |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

- **The Mid- Square Method**
  - The key K is multiplied by itself and the address is obtained by selecting an appropriate number of digits from the middle of the square.
  - The number of digits selected depends on the size of the table.
  - Example: If key=123456 is to be transformed.
  - (123456)2=15241383936
  - If a three-digit address is required, positions 5 to 7 could be chosen giving address 138.

- **The Folding Method**
  - The key K is partitioned into a number of parts ,each of which has the same length as the required address with the possible exception of the last part .
  - The parts are then added together , ignoring the final carry, to form an address.
  - Example: If key=356942781 is to be transformed into a three digit address.

    P1=356, P2=942, P3=781 are added to yield 079.

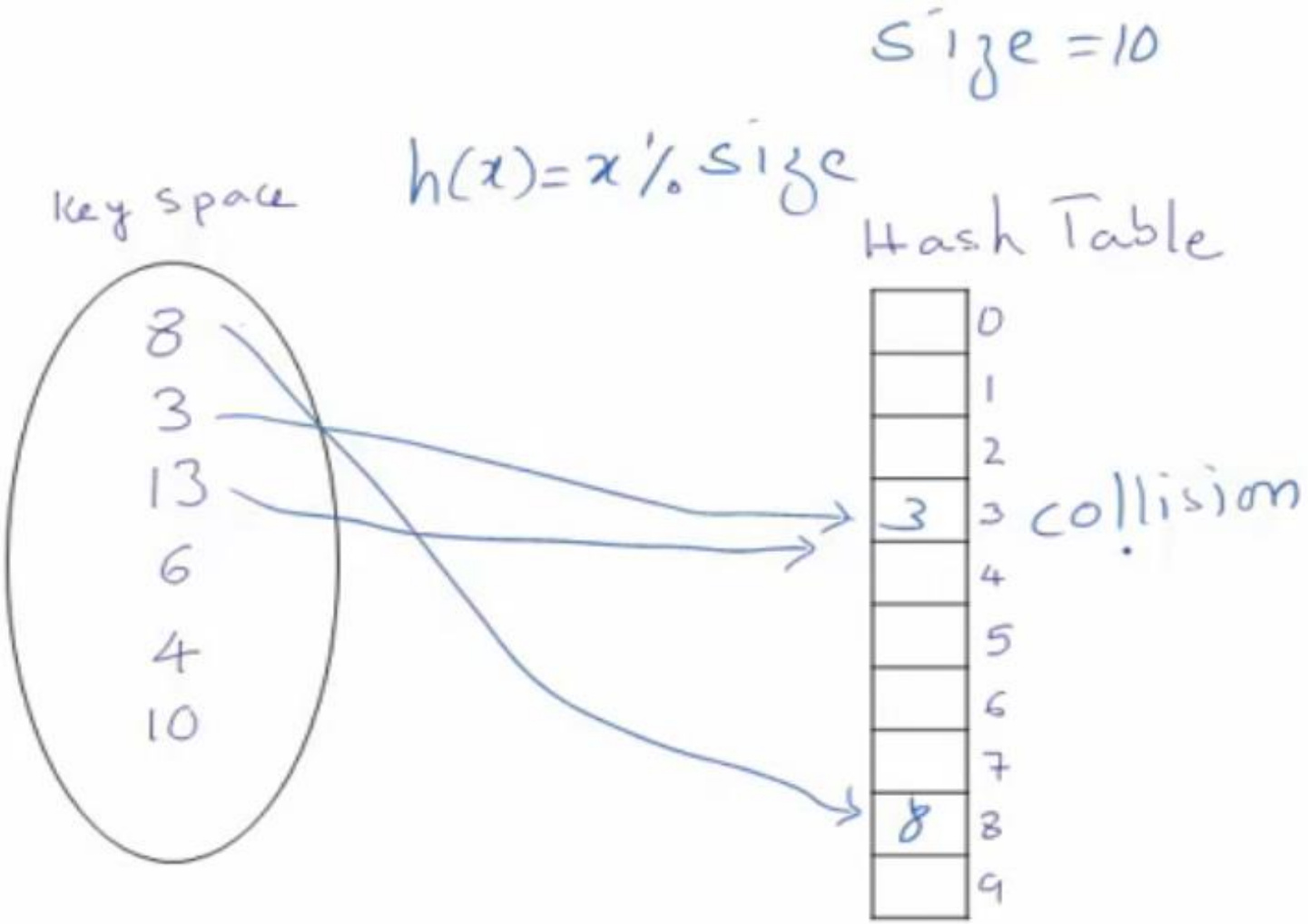
P1+P2+P3=2079, The carry 2 is ignored to form an address of three digit number

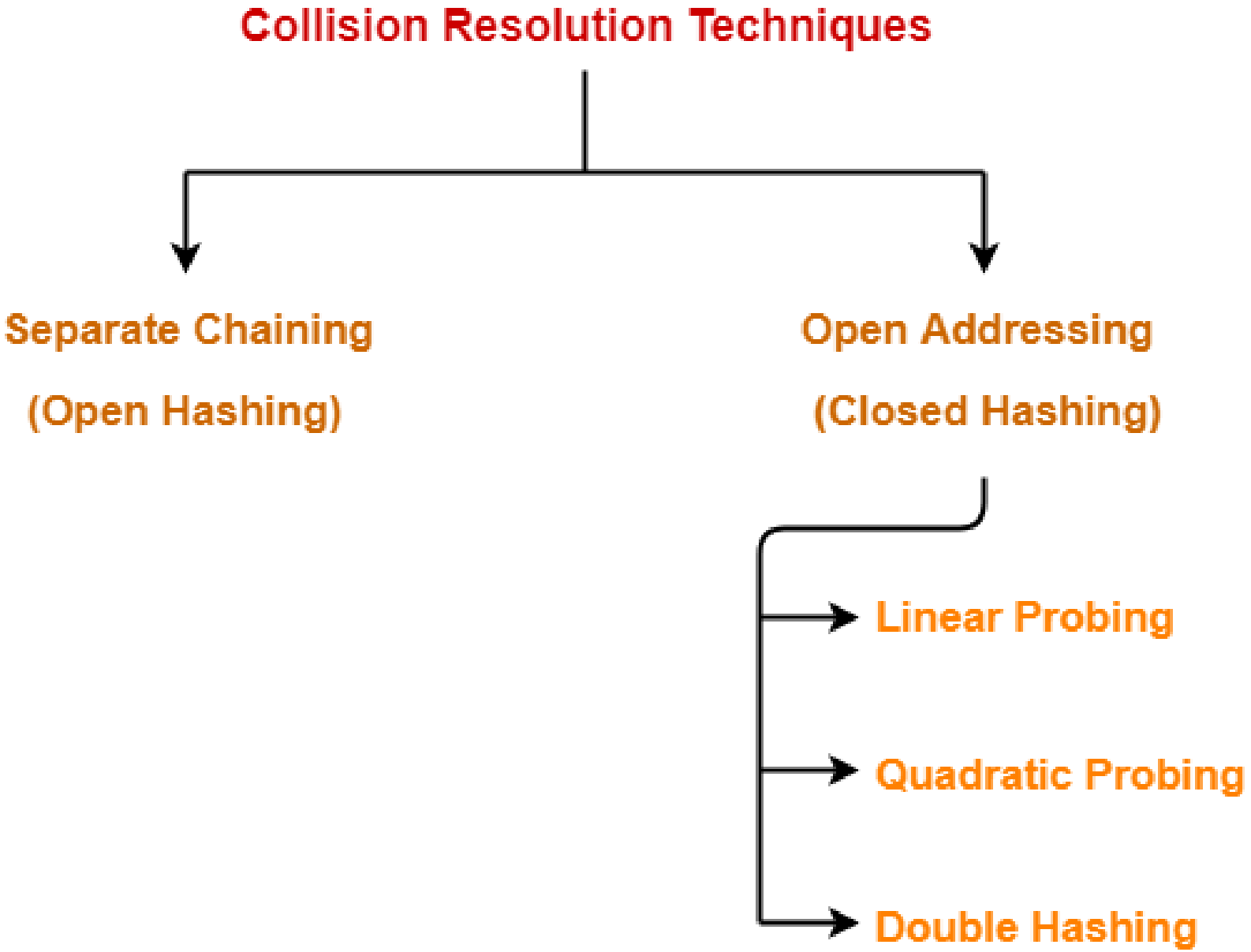
- Folding method divides key into equal parts then adds the parts together
  - The telephone number 01452 8345654, becomes 01 + 45 + 28 + 34 + 56 + 54 = 218
  - Depending on size of table, may then divide by some constant and take remainder

# Hashing a string key

- Table size [0..99]
- A..Z ---> 1,2, ...26
- 0..9 ----> 27,...36
- Key: CS1 --->3+19+28 (concat) = 31,928
- $(31,928)^2$ = 1,019,397,184 - 10 digits
- Extract middle 2 digits (5th and 6th) as table size is 0..99.
- Get 39, so: H(CS1) = 39.

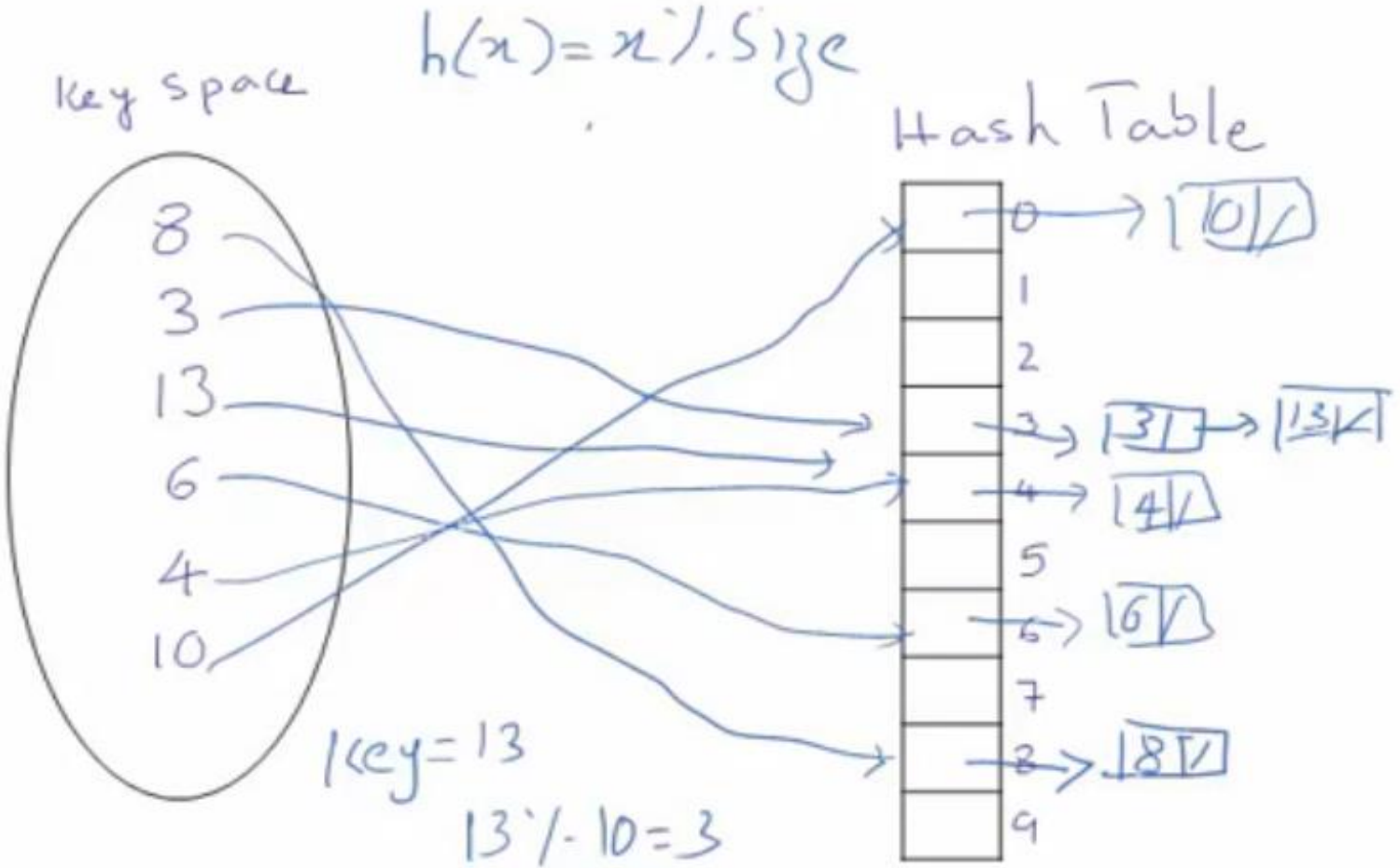
# Hashing

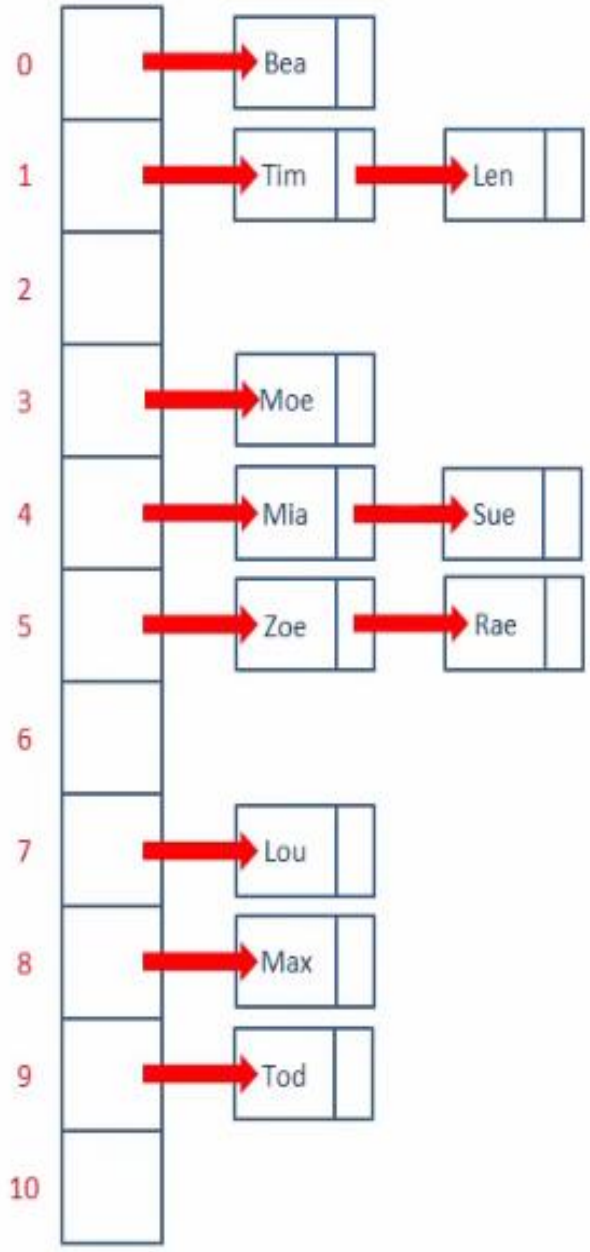

size = 10
h(x) = x % size
Key space
8
3
13
6
4
10
Hash Table
0
1
2
3
3
collision
4
5
6
7
8
8
9


Collision Resolution Techniques
Separate Chaining
(Open Hashing)
Open Addressing
(Closed Hashing)
Linear Probing
Quadratic Probing
Double Hashing

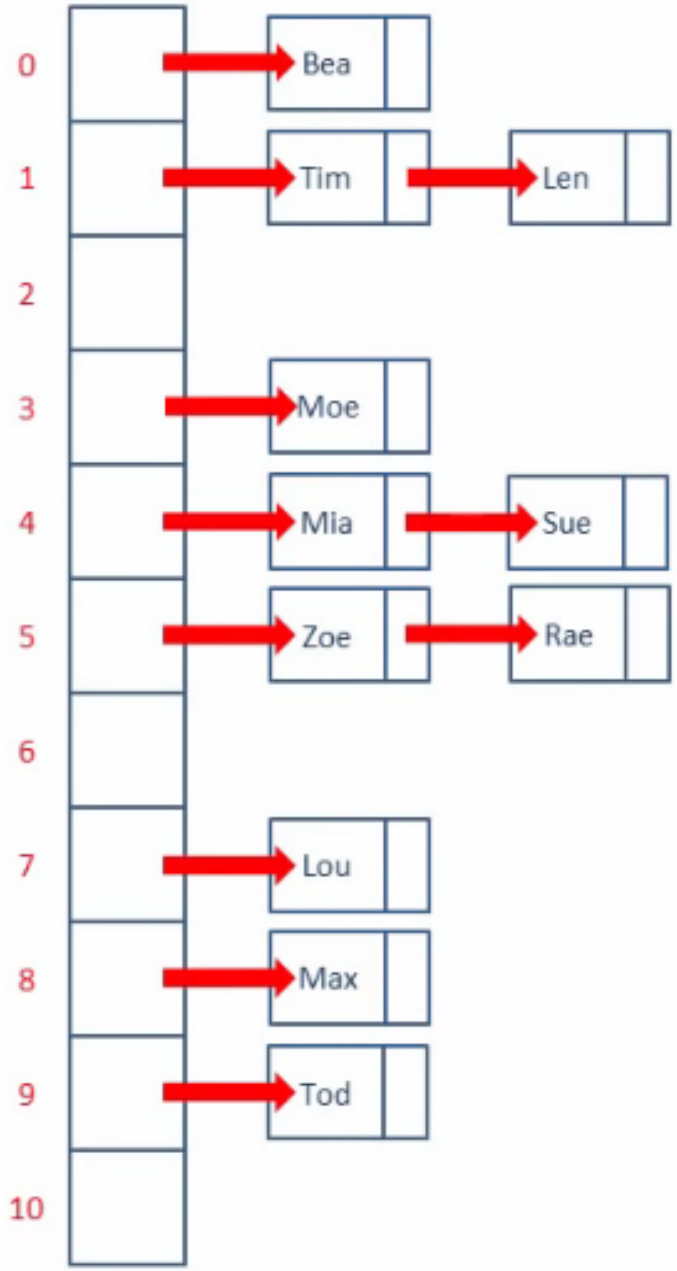
## Open Hashing

## Chaining






| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Sue | S | 83 | u | 117 | e | 101 | 301 | 4 |
| Len | L | 76 | e | 101 | n | 110 | 287 | 1 |
| Moe | M | 77 | o | 111 | e | 101 | 289 | 3 |
| Lou | L | 76 | o | 111 | u | 117 | 304 | 7 |
| Rae | R | 82 | a | 97 | e | 101 | 280 | 5 |
| Max | M | 77 | a | 97 | x | 120 | 294 | 8 |
| Tod | T | 84 | o | 111 | d | 100 | 295 | 9 |




Find Rae  280 Mod 11 = 5

myData = Array(5)

Rae

# Probing

- Probing how to:
  - First probe - given a key k, hash to h(k)
  - Second probe - if h(k) is occupied, try h(k) + f(1)
  - Third probe - if h(k) + f(1) is occupied, try h(k) + f(2)
  - And so forth
- Probing properties
  - we force f(0) = 0
  - the $i^{th}$ probe is to (h(k) + f(i)) mod size
  - if i reaches size - 1, the probe has failed
  - depending on f(), the probe may fail sooner
  - long sequences of probes are costly!

# Linear Probing

f(i) = i

- Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - …

# Linear Probing

Linear Probing

$$h_i(X) = (Hash(X) + i) \% HashTableSize$$

If $h_0$ = (Hash(X) + 0 ) % HashTableSize is full we try for $h_1$

If $h_1$ = (Hash(X) + 1 ) % HashTableSize is full we try for $h_2$

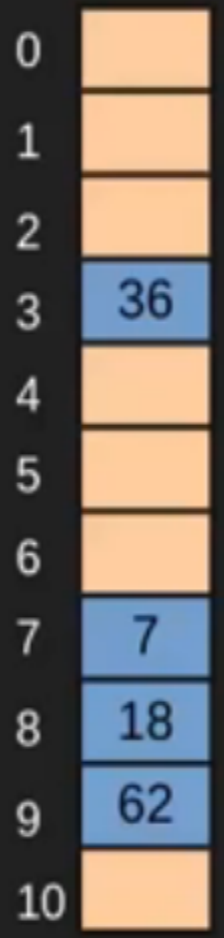
And so on ..

# Linear Probing

Keys : 7,36, 18, 62

Insert(62) :
$h_0(62) = (62 \bmod 11) = 7$
$h_1(62) = ((62+1) \bmod 11) = 8$

$h_2(62) = ((62+2) \bmod 11) = 9$




Empty  Occupied  Deleted

# Linear Probing

Keys : 7,36, 18, 62

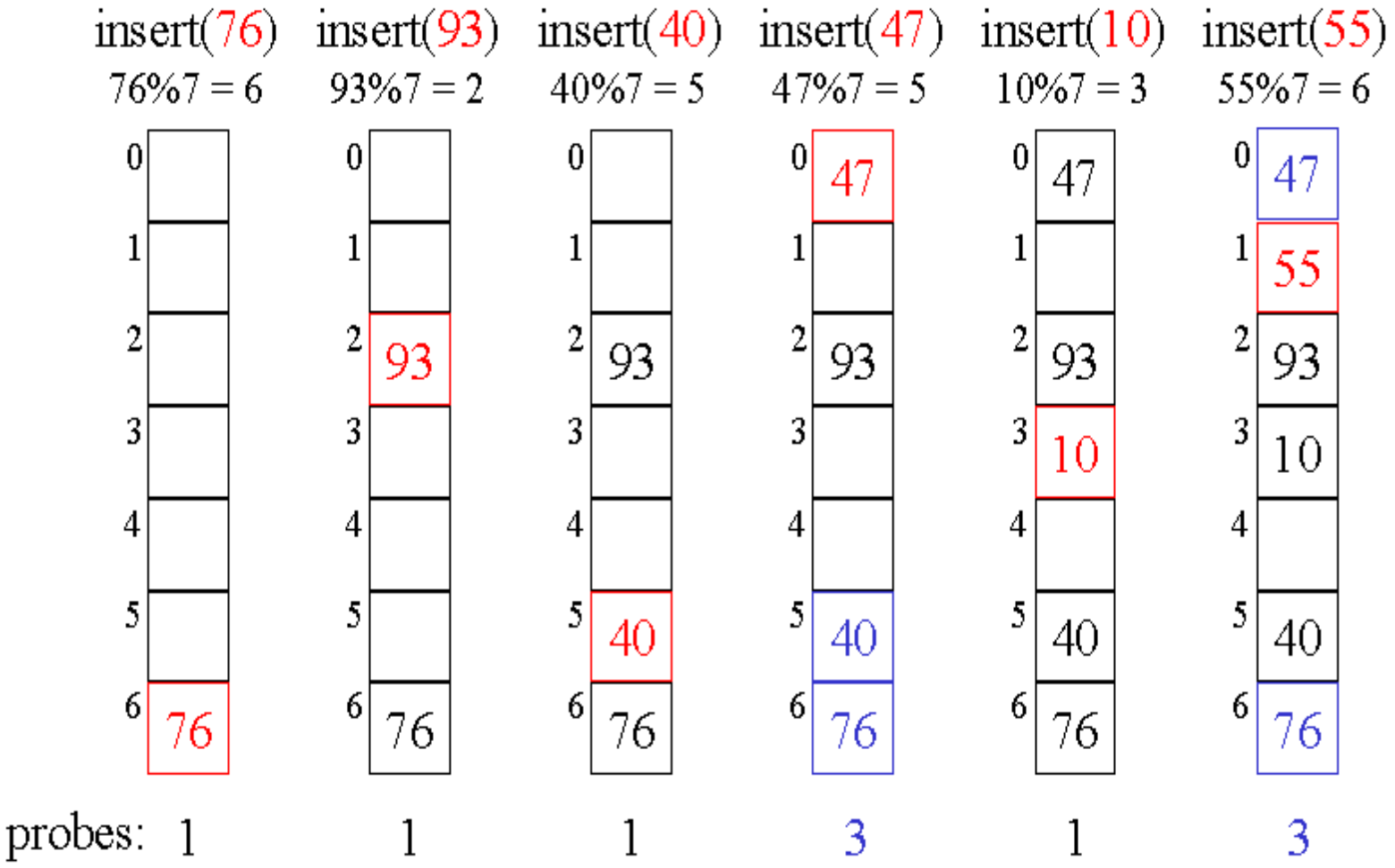
Search(18) :

$h_0(18) = (18 \bmod 11) = 7$

$h_1(18) = ((18+1) \bmod 11) = 8$


Empty  Occupied  Deleted

# Linear Probing Example



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Mia | M | 77 | i | 105 | a | 97 | 279 | 4 |
| Tim | T | 84 | i | 105 | m | 109 | 298 | 1 |
| Bea | B | 66 | e | 101 | a | 97 | 264 | 0 |
| Zoe | Z | 90 | o | 111 | e | 101 | 302 | 5 |
| Sue | S | 83 | u | 117 | e | 101 | 301 | 4 |
| Len | L | 76 | e | 101 | n | 110 | 287 | 1 |
| Moe | M | 77 | o | 111 | e | 101 | 289 | 3 |
| Lou | L | 76 | o | 111 | u | 117 | 304 | 7 |
| Rae | R | 82 | a | 97 | e | 101 | 280 | 5 |
| Max | M | 77 | a | 97 | x | 120 | 294 | 8 |
| Tod | T | 84 | o | 111 | d | 100 | 295 | 9 |

| Bea | Tim | Len | Moe | Mia | Zoe | Sue | Lou | Rae | Max | Tod |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# KCA-205, MCA -2021-22 [Q4(b), 10 marks]

Q. What do you understand by Hashing? Consider inserting the following keys

{ 76, 26, 37, 59, 21, 65, 88 } into a hashtable of size m=11 using Linear Probing. Consider the primary hash function as h(k)=k mod m

# Quadratic Probing

$f(i) = i^2$

- Probe sequence is
  - h(k) mod size
  - (h(k) + 1) mod size
  - (h(k) + 4) mod size
  - (h(k) + 9) mod size
  - …

# Quadratic Probing

Quadratic Probing :

$h_i(X) = (Hash(X) + i^2) \% HashTableSize$

If $h_0 = (Hash(X) + 0) \% HashTableSize$ is full we try for $h_1$
If $h_1 = (Hash(X) + 1) \% HashTableSize$ is full we try for $h_2$
If $h_2 = (Hash(X) + 4) \% HashTableSize$ is full we try for $h_3$
And so on ..

# Quadratic Probing

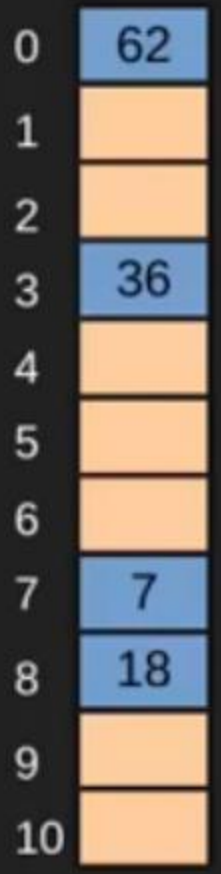
Keys : 7,36, 18, 62

Insert(62) :

$h_0(62) = (62 \bmod 11) = 7$

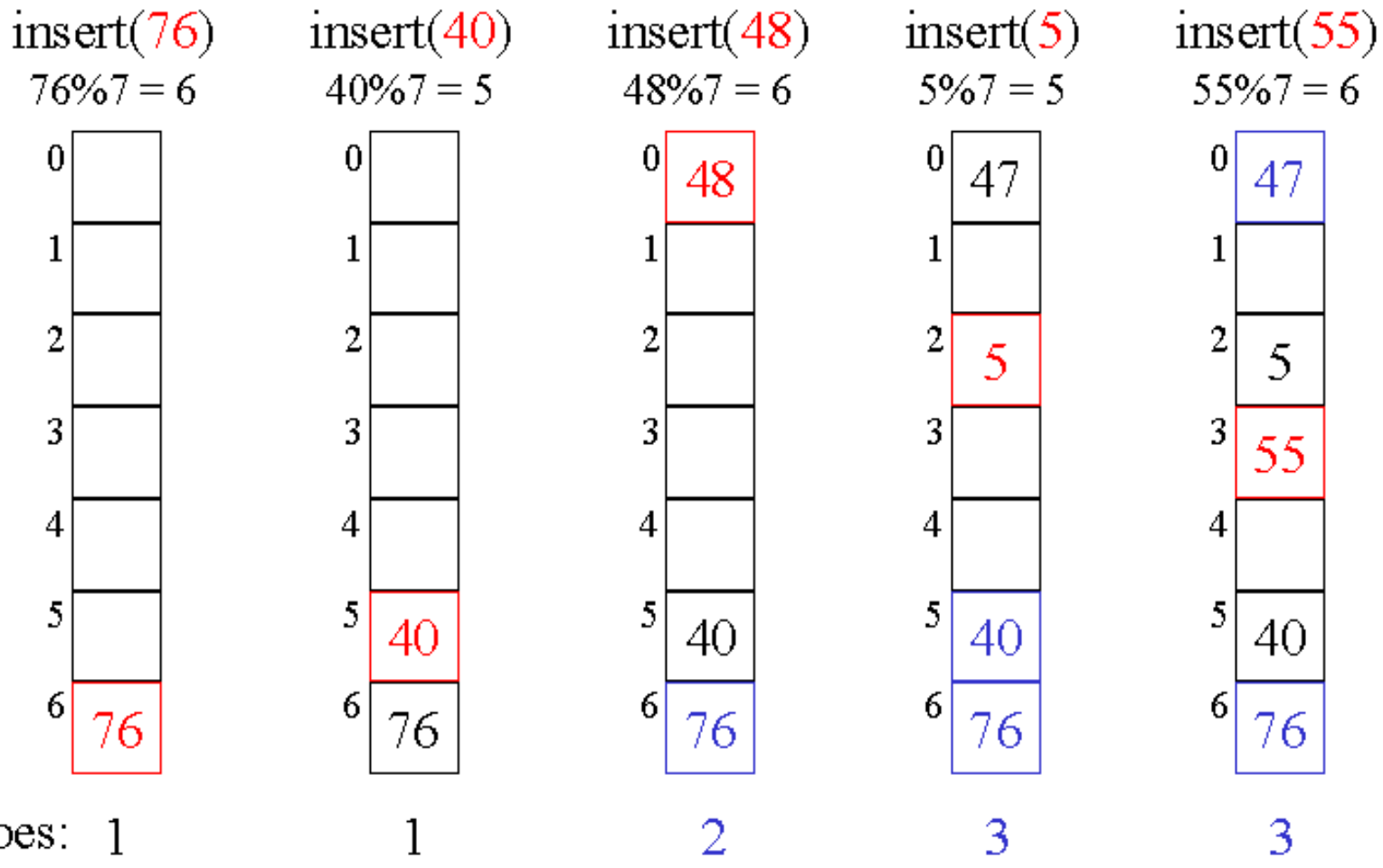
$h_1(62) = ((62+1*1) \bmod 11) = 8$

$h_2(62) = ((62+2*2) \bmod 11) = 0$

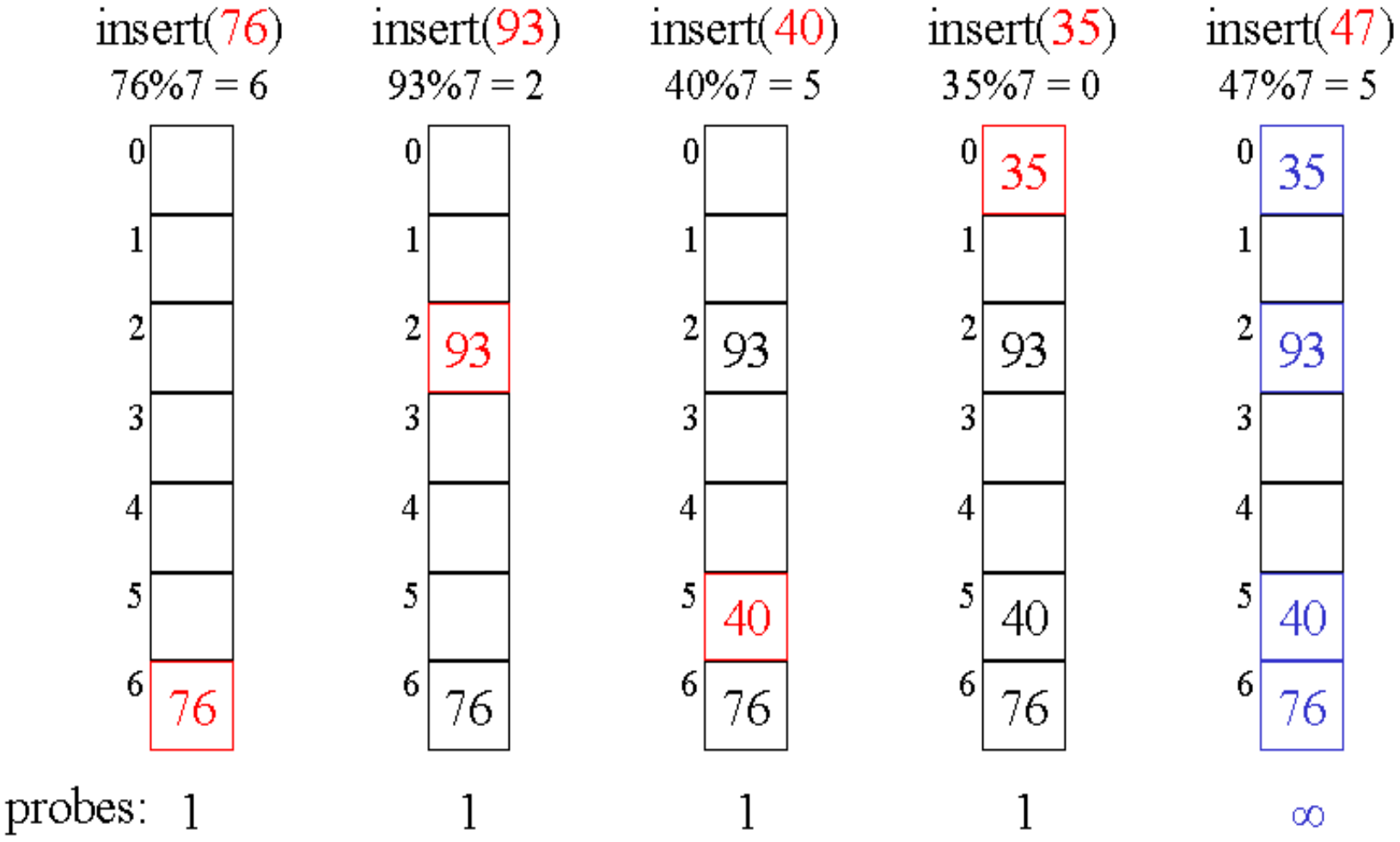



Empty  Occupied  Deleted

# Quadratic Probing Example ☺



# Quadratic Probing Example ☹



# Double Hashing

$f(i) = i \cdot hash_2(x)$

- Probe sequence is
  - $h_1(k) \bmod size$
  - $(h_1(k) + 1 \cdot h_2(x)) \bmod size$
  - $(h_1(k) + 2 \cdot h_2(x)) \bmod size$
  - ...

# A Good Double Hash Function…

…is quick to evaluate.

…differs from the original hash function.

…never evaluates to 0 (mod size).

One good choice is to choose a prime R < size and:

$hash_2(x) = R - (x \bmod R)$

# Double Hashing

Double hashing : use another hash function hash2(x) and look for i*hash2(x) slot in i'th iteration.

$$h_i(X) = (Hash(X) + i * Hash_2(X)) \% HashTableSize$$

If $h_0$ = (Hash(X) + 0 ) % HashTableSize is full we try for $h_1$

If $h_1$ = (Hash(X) + 1 * $Hash_2$(X)) % HashTableSize is full we try for $h_2$

If $h_2$ = (Hash(X) + 2 * $Hash_2$(X)) % HashTableSize is full we try for $h_3$

And so on ..

H1(k) = k%10

H2(k) = 7 - (k%7)

$H_1$ (89) = 89 % 10 = 9 (primary hashing)

$H_1$ (18) = 18 % 10 = 8

$H_1$ (49) = 49 % 10 = 9 a collision (primary hashing)

$H_2$ (49) = 7 – (49 % 7) (Secondary hashing)
= 7 positions from [9]

$H_1$ (58) = 58 % 10 = 8

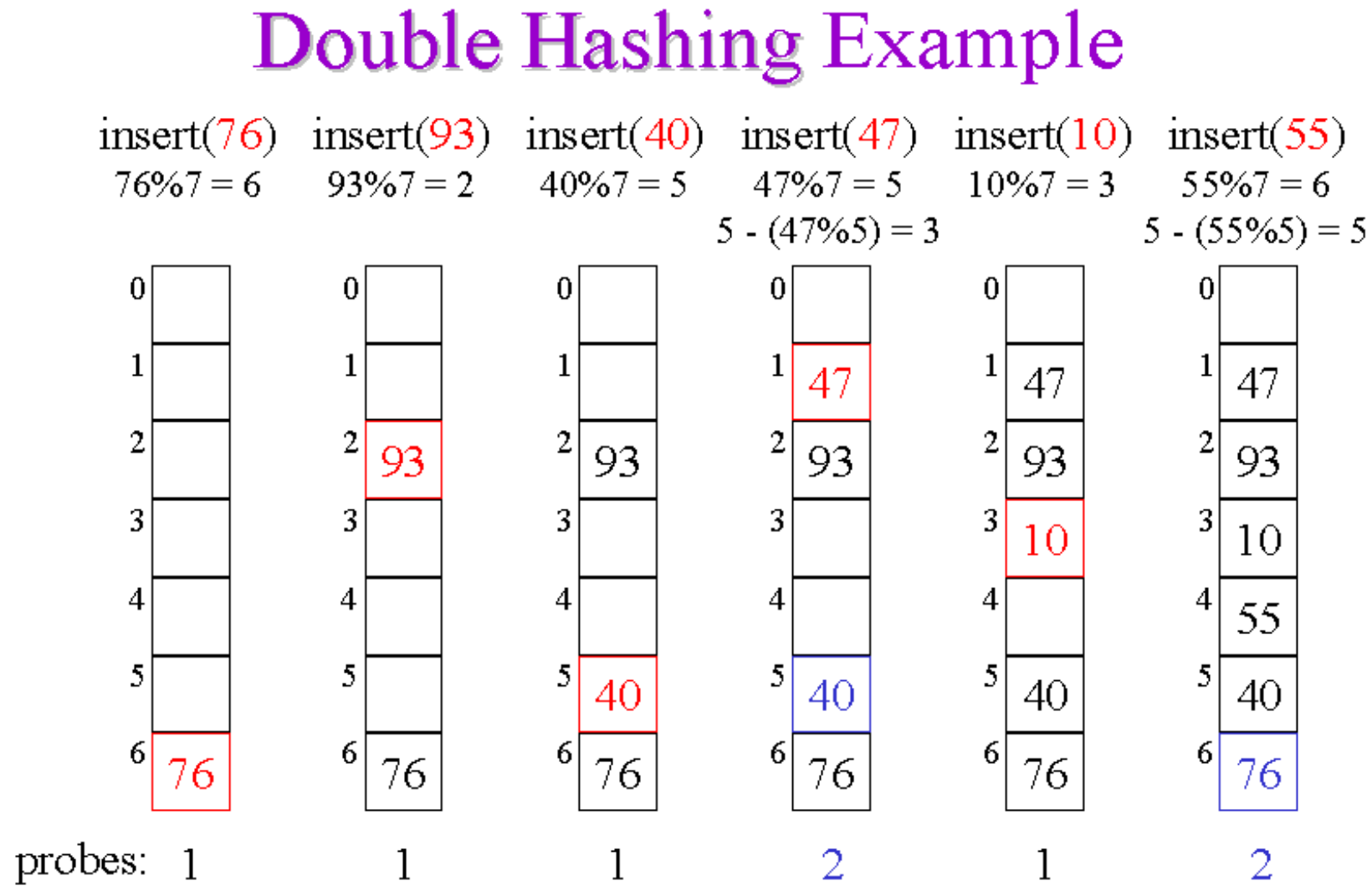
$H_2$ (58) = 7 – (58 % 7)
= 5 positions from [8]

$H_1$ (69) = 69 % 10 = 9

$H_2$ (69) = 7 – (69 % 7)
= 1 position from [9]

H1(k) = k%7

H2(k) = R - (k%R)  //R can be the nearest prime less than 7



# Comparison

**Linear Probing**

- Easy to implement
- Best Cache Performance
- Suffers from clustering

**Quadratic Probing**

- Average Cache Performance
- Suffers  a lesser clustering than linear probing

**Double Hashing**

- Poor Cache Performance
- No clustering
- Requires more computation time

# Objectives of Hash Function

- Minimize collisions
- Uniform distribution of hash values
- Easy to calculate
- Resolve any collisions

# Summary

- Used to index large amounts of data
- Address of each key calculated using the key itself
- Collisions resolved with open or closed addressing
- Hashing is widely used in database indexing, compilers, caching, password authentication, and more
- Insertion, deletion and retrieval occur in constant time

# Divide and Conquer |[redacted](Strassen's Matrix Multiplication)

geeksforgeeks.org/strassens-matrix-multiplication

Given two square matrices A and B of size n x n each, find their multiplication matrix.

***Naive Method***: Following is a simple way to multiply two matrices.

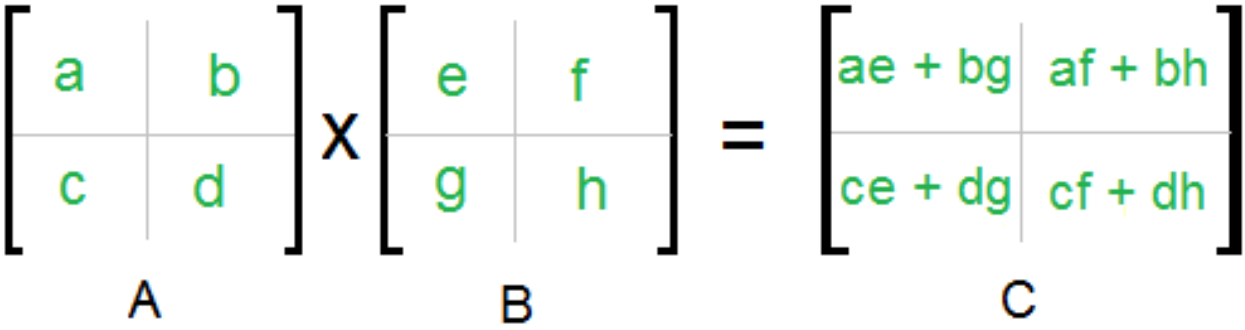
```
void multiply(int A[][N], int B[][N], int C[][N])
{
for (int i = 0; i < N; i++)
{
for (int j = 0; j < N; j++)
{
C[i][j] = 0;
for (int k = 0; k < N; k++)
{
C[i][j] += A[i][k]*B[k][j];
}
}
}
}
```

Time Complexity of above method is $O(N^3)$.

***Divide and Conquer*** *:*

Following is simple Divide and Conquer method to multiply two square matrices.

1. Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2. Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.




A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$T(N) = 8T(N/2) + O(N^2)$

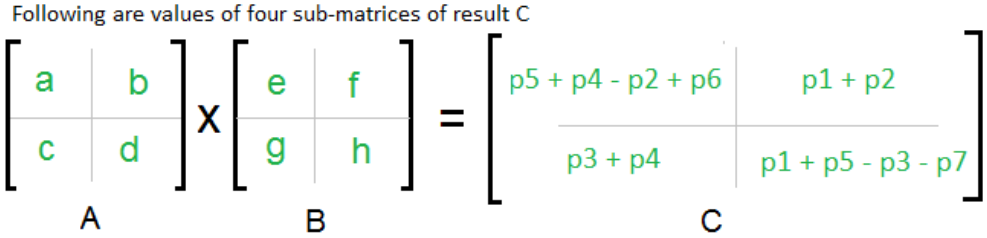
From Master's Theorem, time complexity of above method is $O(N^3)$
which is unfortunately same as the above naive method.

***Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?***

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

**p1 = a(f - h)**
**p2 = (a + b)h**
**p3 = (c + d)e**
**p4 = d(g - e)**
**p5 = (a + d)(e + h)**
**p6 = (b - d)(g + h)**
**p7 = (a - c)(e + f)**

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C




A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

### Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Generally Strassen's Method is not preferred for practical applications for following reasons.

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The submatrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method

# GRAPH THEORY - QUICK GUIDE



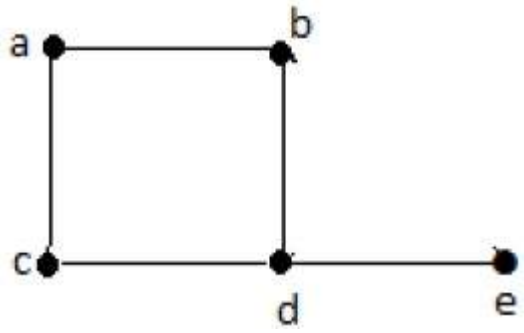
# GRAPH THEORY - INTRODUCTION

In the domain of mathematics and computer science, *graph theory is the study of graphs that concerns with the relationship among edges and vertices*. It is a popular subject having its applications in computer science, information technology, biosciences, mathematics, and linguistics to name a few. Without further ado, let us start with defining a graph.

## What is a Graph?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices,** and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets $V, E$**,** where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −




In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

## Applications of Graph Theory

Graph theory has its applications in diverse fields of engineering −

- **Electrical Engineering** − The concepts of graph theory is used extensively in designing circuit connections. The types or organization of connections are named as topologies. Some examples for topologies are star, bridge, series, and parallel topologies.
- **Computer Science** − Graph theory is used for the study of algorithms. For example,
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Dijkstra's Algorithm
- **Computer Network** − The relationships among interconnected computers in the network follows the principles of graph theory.
- **Science** − The molecular structure and chemical structure of a substance, the DNA structure of an organism, etc., are represented by graphs.
- **Linguistics** − The parsing tree of a language and grammar of a language uses graphs.
- **General** − Routes between the cities can be represented using graphs. Depicting hierarchical ordered information such as family tree can be used as a special type of graph called tree.

# GRAPH THEORY - FUNDAMENTALS

A graph is a diagram of points and lines connected to the points. It has at least one line joining a set of two vertices with no vertex connecting itself. The concept of graphs in graph theory stands up on some basic terms such as point, line, vertex, edge, degree of vertices, properties of graphs, etc. Here, in this chapter, we will cover these fundamentals of graph theory.

## Point

**A point** is a particular position in a one-dimensional, two-dimensional, or three-dimensional space. For better understanding, a point can be denoted by an alphabet. It can be represented with a dot.

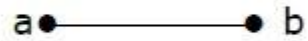
## Example


Here, the dot is a point named ‘a’.

## Line

A **Line** is a connection between two points. It can be represented with a solid line.

## Example




Here, ‘a’ and ‘b’ are the points. The link between these two points is called a line.

## Vertex

A vertex is a point where multiple lines meet. It is also called a **node**. Similar to points, a vertex is also denoted by an alphabet.

## Example


Here, the vertex is named with an alphabet ‘a’.

## Edge

An edge is the mathematical term for a line that connects two vertices. Many edges can be formed from a single vertex. Without a vertex, an edge cannot be formed. There must be a starting vertex and an ending vertex for an edge.

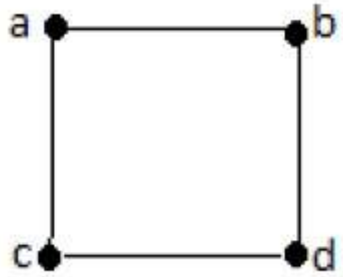
## Example

Here, ‘a’ and ‘b’ are the two vertices and the link between them is called an edge.

## Graph

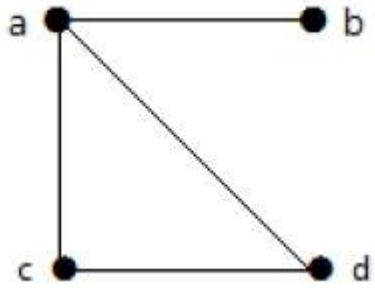
A graph ‘G’ is defined as G = $V, E$ Where V is a set of all vertices and E is a set of all edges in the graph.

## Example 1




In the above example, ab, ac, cd, and bd are the edges of the graph. Similarly, a, b, c, and d are the vertices of the graph.

## Example 2




In this graph, there are four vertices a, b, c, and d, and four edges ab, ac, ad, and cd.

## Loop

In a graph, if an edge is drawn from vertex to itself, it is called a loop.

## Example 1


In the above graph, V is a vertex for which it has an edge $V, V$ forming a loop.

## Example 2

In this graph, there are two loops which are formed at vertex a, and vertex b.

## Degree of Vertex

It is the number of vertices incident with the vertex V.

**Notation** − deg$V$.

In a simple graph with n number of vertices, the degree of any vertices is −

```
deg(v) ≤ n – 1 ∀ v ∈ G
```

A vertex can form an edge with all other vertices except by itself. So the degree of a vertex will be up to the **number of vertices in the graph minus 1**. This 1 is for the self-vertex as it cannot form a loop by itself. If there is a loop at any of the vertices, then it is not a Simple Graph.

Degree of vertex can be considered under two cases of graphs −

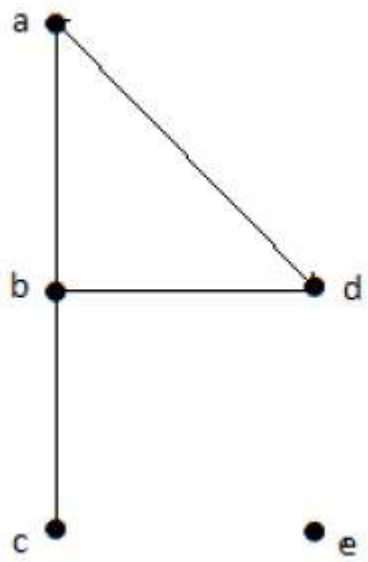
- Undirected Graph
- Directed Graph

## Degree of Vertex in an Undirected Graph

An undirected graph has no directed edges. Consider the following examples.

## Example 1

Take a look at the following graph −




In the above Undirected Graph,

- deg$a$ = 2, as there are 2 edges meeting at vertex ‘a’.
- deg$b$ = 3, as there are 3 edges meeting at vertex ‘b’.
- deg$c$ = 1, as there is 1 edge formed at vertex ‘c’

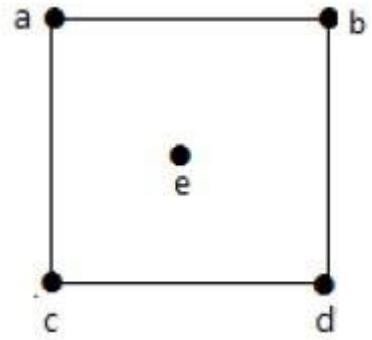
  So ‘c’ is a **pendent vertex**.
- deg$d$ = 2, as there are 2 edges meeting at vertex ‘d’.

- deg$e$ = 0, as there are 0 edges formed at vertex ‘e’.

So ‘e’ is an **isolated vertex**.

## Example 2

Take a look at the following graph −




In the above graph,

deg$a$ = 2, deg$b$ = 2, deg$c$ = 2, deg$d$ = 2, and deg$e$ = 0.

The vertex ‘e’ is an isolated vertex. The graph does not have any pendent vertex.

## Degree of Vertex in a Directed Graph

In a directed graph, each vertex has an **indegree** and an **outdegree**.

## Indegree of a Graph

- Indegree of vertex V is the number of edges which are coming into the vertex V.
- **Notation** − deg$^{+}V$.

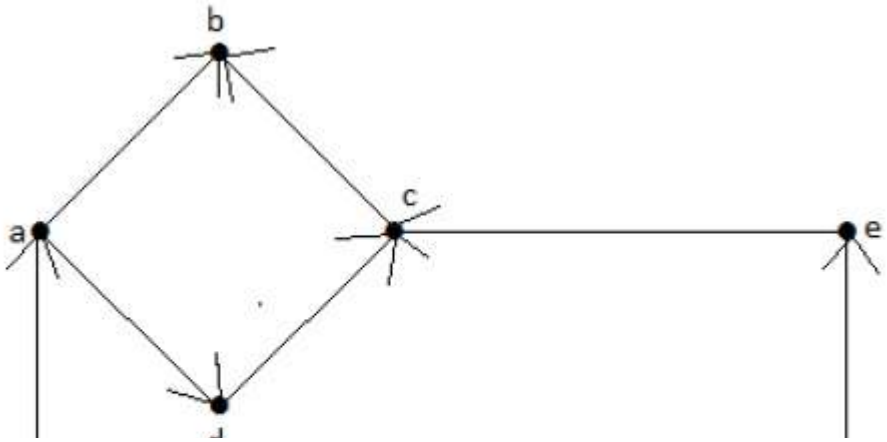
## Outdegree of a Graph

- Outdegree of vertex V is the number of edges which are going out from the vertex V.
- **Notation** − deg$^{-}V$.

Consider the following examples.

## Example 1

Take a look at the following directed graph. Vertex ‘a’ has two edges, ‘ad’ and ‘ab’, which are going outwards. Hence its outdegree is 2. Similarly, there is an edge ‘ga’, coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.

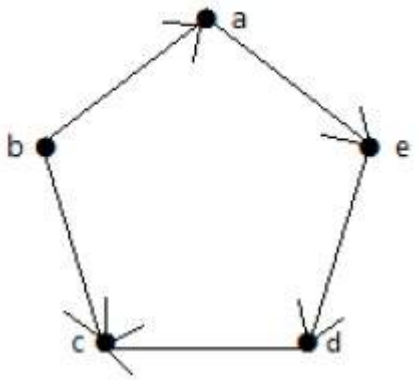



The indegree and outdegree of other vertices are shown in the following table −

| Vertex | Indegree | Outdegree |
|---|---|---|
| a | 1 | 2 |
| b | 2 | 0 |
| c | 2 | 1 |
| d | 1 | 1 |
| e | 1 | 1 |
| f | 1 | 1 |
| g | 0 | 2 |

## Example 2

Take a look at the following directed graph. Vertex ‘a’ has an edge ‘ae’ going outwards from vertex ‘a’. Hence its outdegree is 1. Similarly, the graph has an edge ‘ba’ coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.




The indegree and outdegree of other vertices are shown in the following table −

| Vertex | Indegree | Outdegree |
|---|---|---|
| a | 1 | 1 |
| b | 0 | 2 |
| c | 2 | 0 |
| d | 1 | 1 |
| e | 1 | 1 |

## Pendent Vertex

By using degree of a vertex, we have a two special types of vertices. A vertex with degree one is called a pendent vertex.

## Example


Here, in this example, vertex ‘a’ and vertex ‘b’ have a connected edge ‘ab’. So with respect to the vertex ‘a’, there is only one edge towards vertex ‘b’ and similarly with respect to the vertex ‘b’, there is only one edge towards vertex ‘a’. Finally, vertex ‘a’ and vertex ‘b’ has degree as one which are also called as the pendent vertex.

## Isolated Vertex

A vertex with degree zero is called an isolated vertex.

## Example

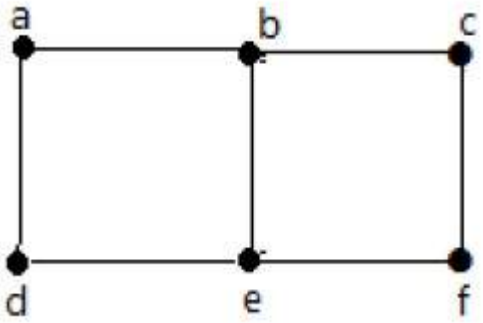

Here, the vertex ‘a’ and vertex ‘b’ has a no connectivity between each other and also to any other vertices. So the degree of both the vertices ‘a’ and ‘b’ are zero. These are also called as isolated vertices.

## Adjacency

Here are the norms of adjacency −

- In a graph, two vertices are said to be **adjacent,** if there is an edge between the two vertices. Here, the adjacency of vertices is maintained by the single edge that is connecting those two vertices.
- In a graph, two edges are said to be adjacent, if there is a common vertex between the two edges. Here, the adjacency of edges is maintained by the single vertex that is connecting two edges.

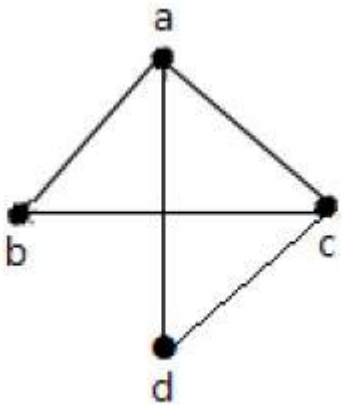
## Example 1




In the above graph −

- ‘a’ and ‘b’ are the adjacent vertices, as there is a common edge ‘ab’ between them.
- ‘a’ and ‘d’ are the adjacent vertices, as there is a common edge ‘ad’ between them.
- ab’ and ‘be’ are the adjacent edges, as there is a common vertex ‘b’ between them.
- be’ and ‘de’ are the adjacent edges, as there is a common vertex ‘e’ between them.

## Example 2

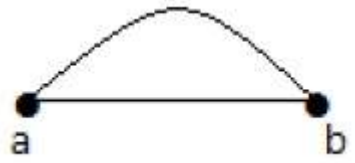



In the above graph −

- a’ and ‘d’ are the adjacent vertices, as there is a common edge ‘ad’ between them.
- ‘c’ and ‘b’ are the adjacent vertices, as there is a common edge ‘cb’ between them.
- ‘ad’ and ‘cd’ are the adjacent edges, as there is a common vertex ‘d’ between them.
- ac’ and ‘cd’ are the adjacent edges, as there is a common vertex ‘c’ between them.

## Parallel Edges

In a graph, if a pair of vertices is connected by more than one edge, then those edges are called parallel edges.

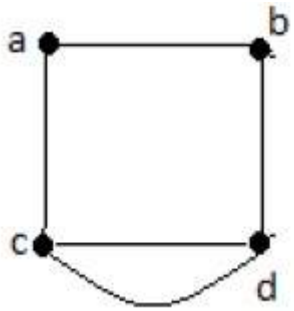



In the above graph, ‘a’ and ‘b’ are the two vertices which are connected by two edges ‘ab’ and ‘ab’ between them. So it is called as a parallel edge.

## Multi Graph

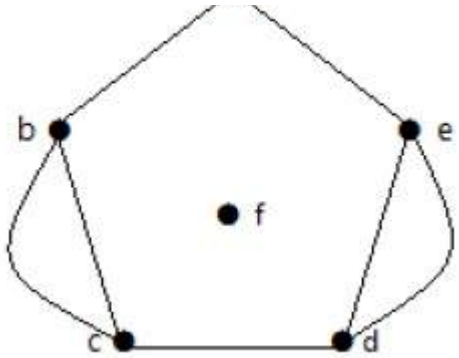
A graph having parallel edges is known as a Multigraph.

## Example 1




In the above graph, there are five edges ‘ab’, ‘ac’, ‘cd’, ‘cd’, and ‘bd’. Since ‘c’ and ‘d’ have two parallel edges between them, it a Multigraph.

## Example 2

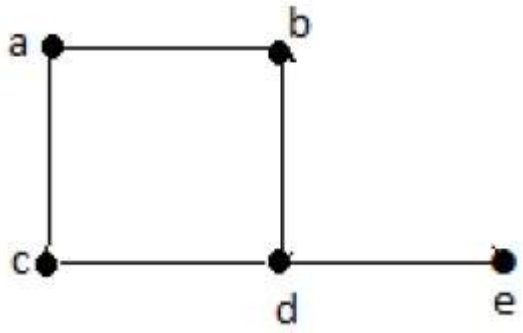



In the above graph, the vertices ‘b’ and ‘c’ have two edges. The vertices ‘e’ and ‘d’ also have two edges between them. Hence it is a Multigraph.

## Degree Sequence of a Graph

If the degrees of all vertices in a graph are arranged in descending or ascending order, then the sequence obtained is known as the degree sequence of the graph.

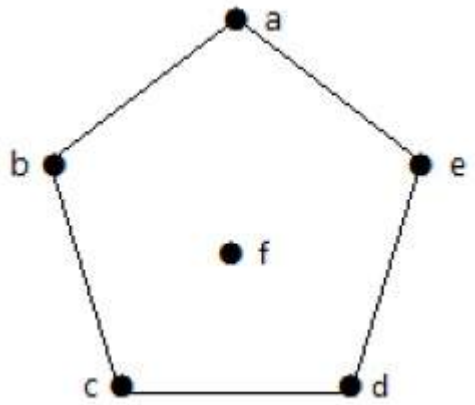
## Example 1




| **Vertex** | A | b | c | d | e |
|---|---|---|---|---|---|
| **Connecting to** | b,c | a,d | a,d | c,b,e | d |
| **Degree** | 2 | 2 | 2 | 3 | 1 |

In the above graph, for the vertices {d, a, b, c, e}, the degree sequence is {3, 2, 2, 2, 1}.

## Example 2




| **Vertex** | A | b | c | d | e | f |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| **Connecting to** | b,e | a,c | b,d | c,e | a,d | - |
| **Degree** | 2 | 2 | 2 | 2 | 2 | 0 |

In the above graph, for the vertices {a, b, c, d, e, f}, the degree sequence is {2, 2, 2, 2, 2, 0}.

# GRAPH THEORY - BASIC PROPERTIES

Graphs come with various properties which are used for characterization of graphs depending on their structures. These properties are defined in specific terms pertaining to the domain of graph theory. In this chapter, we will discuss a few basic properties that are common in all graphs.

## Distance between Two Vertices

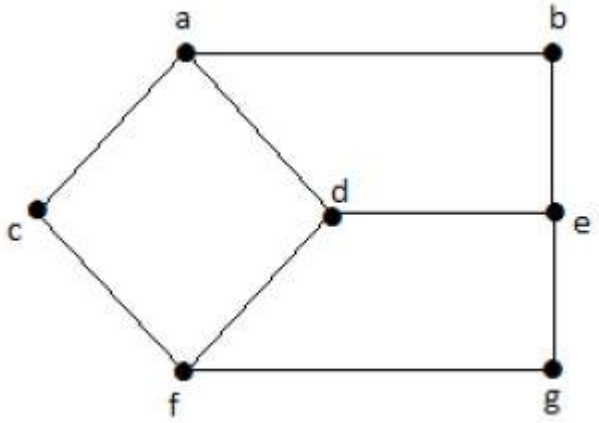
It is number of edges in a shortest path between Vertex U and Vertex V. If there are multiple paths connecting two vertices, then the shortest path is considered as the distance between the two vertices.

**Notation** − d$U, V$

There can be any number of paths present from one vertex to other. Among those, you need to choose only the shortest one.

## Example

Take a look at the following graph −




Here, the distance from vertex ‘d’ to vertex ‘e’ or simply ‘de’ is 1 as there is one edge between them. There are many paths from vertex ‘d’ to vertex ‘e’ −

- da, ab, be
- df, fg, ge
- de $Itisconsideredfordistancebetweenthevertices$
- df, fc, ca, ab, be
- da, ac, cf, fg, ge

## Eccentricity of a Vertex

The maximum distance between a vertex to all other vertices is considered as the eccentricity of vertex.

**Notation** − e$V$

The distance from a particular vertex to all other vertices in the graph is taken and among those distances, the eccentricity is the highest of distances.

## Example

In the above graph, the eccentricity of ‘a’ is 3.

The distance from ‘a’ to ‘b’ is 1 ‘$ab$’,

from ‘a’ to ‘c’ is 1 ‘$ac$’,

from ‘a’ to ‘d’ is 1 ‘$ad$’,

from ‘a’ to ‘e’ is 2 ‘$ab$’ − ‘$be$’ or ‘$ad$’ − ‘$de$’,

from ‘a’ to ‘f’ is 2 ‘$ac$’ − ‘$cf$’ or ‘$ad$’ − ‘$df$’,

from ‘a’ to ‘g’ is 3 ‘$ac$’ − ‘$cf$’ − ‘$fg$’ or ‘$ad$’ − ‘$df$’ − ‘$fg$’.

So the eccentricity is 3, which is a maximum from vertex ‘a’ from the distance between ‘ag’ which is maximum.

In other words,

e$b$ = 3

e$c$ = 3

e$d$ = 2

e$e$ = 3

e$f$ = 3

e$g$ = 3

## Radius of a Connected Graph

The minimum eccentricity from all the vertices is considered as the radius of the Graph G. The minimum among all the maximum distances between a vertex to all other vertices is considered as the radius of the Graph G.

**Notation** − r$G$

From all the eccentricities of the vertices in a graph, the radius of the connected graph is the minimum of all those eccentricities.

**Example** − In the above graph r$G$ = 2, which is the minimum eccentricity for ‘d’.

## Diameter of a Graph

The maximum eccentricity from all the vertices is considered as the diameter of the Graph G. The maximum among all the distances between a vertex to all other vertices is considered as the diameter of the Graph G.

**Notation** − d$G$

From all the eccentricities of the vertices in a graph, the diameter of the connected graph is the maximum of all those eccentricities.

**Example** − In the above graph, d$G$ = 3; which is the maximum eccentricity.

## Central Point

If the eccentricity of a graph is equal to its radius, then it is known as the central point of the graph. If

e$V$ = r$V$,

then ‘V’ is the central point of the Graph ’G’.

**Example** − In the example graph, ‘d’ is the central point of the graph.

$e_d = r_d = 2$

## Centre

The set of all central points of ‘G’ is called the centre of the Graph.

**Example** − In the example graph, {‘d’} is the centre of the Graph.

## Circumference

The **number of edges in the longest cycle of ‘G’** is called as the circumference of ‘G’.

**Example** − In the example graph, the circumference is 6, which we derived from the longest cycle a-c-f-g-e-b-a or a-c-f-d-e-b-a.

## Girth

The number of edges in the shortest cycle of ‘G’ is called its Girth.

**Notation** − $g_G$.

**Example** − In the example graph, the Girth of the graph is 4, which we derived from the shortest cycle a-c-f-d-a or d-f-g-e-d or a-b-e-d-a.

## Sum of Degrees of Vertices Theorem

If G = $V, E$ be a non-directed graph with vertices V = {$V_1$, $V_2$,...$V_n$} then

$$\sum_{i=1}^{n} deg(V_i) = 2|E|$$

## Corollary 1

If G = $V, E$ be a directed graph with vertices V = {$V_1$, $V_2$,...$V_n$}, then

$$\sum_{i=1}^{n} deg^{+}(V_i) = |E| = \sum_{i=1}^{n} deg^{-}(V_i)$$

## Corollary 2

In any non-directed graph, the number of vertices with Odd degree is Even.

## Corollary 3

In a non-directed graph, if the degree of each vertex is k, then

$$k|V| = 2|E|$$

## Corollary 4

In a non-directed graph, if the degree of each vertex is at least k, then

$$k|V| \leq 2|E|$$

## Corollary 5

In a non-directed graph, if the degree of each vertex is at most k, then

$$k|V| \geq 2|E|$$

# GRAPH THEORY - TYPES OF GRAPHS

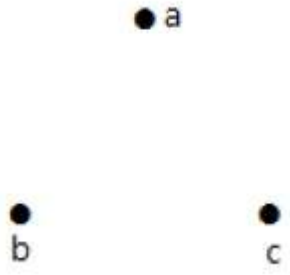
There are various types of graphs depending upon the number of vertices, number of edges,

interconnectivity, and their overall structure. We will discuss only a certain few important types of graphs in this chapter.

## Null Graph

A **graph having no edges** is called a Null Graph.

## Example




In the above graph, there are three vertices named ‘a’, ‘b’, and ‘c’, but there are no edges among them. Hence it is a Null Graph.

## Trivial Graph

A **graph with only one vertex** is called a Trivial Graph.

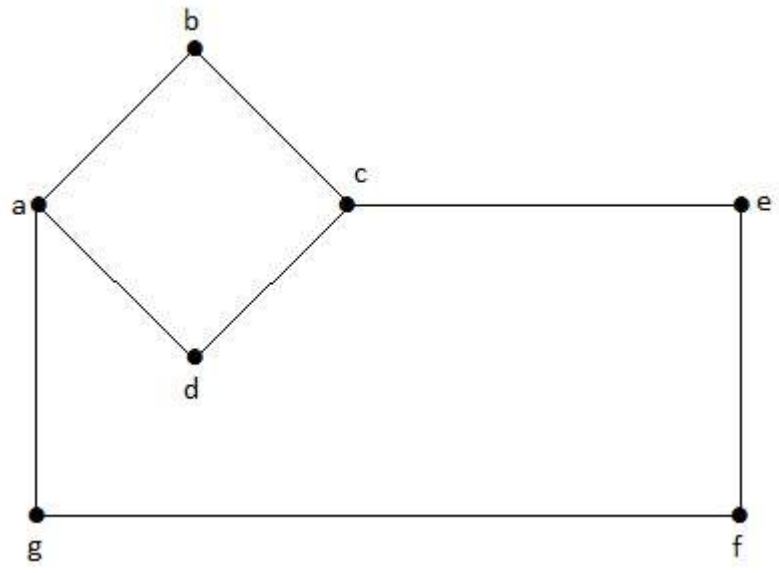
## Example


In the above shown graph, there is only one vertex ‘a’ with no other edges. Hence it is a Trivial graph.

## Non-Directed Graph

A non-directed graph contains edges but the edges are not directed ones.

## Example



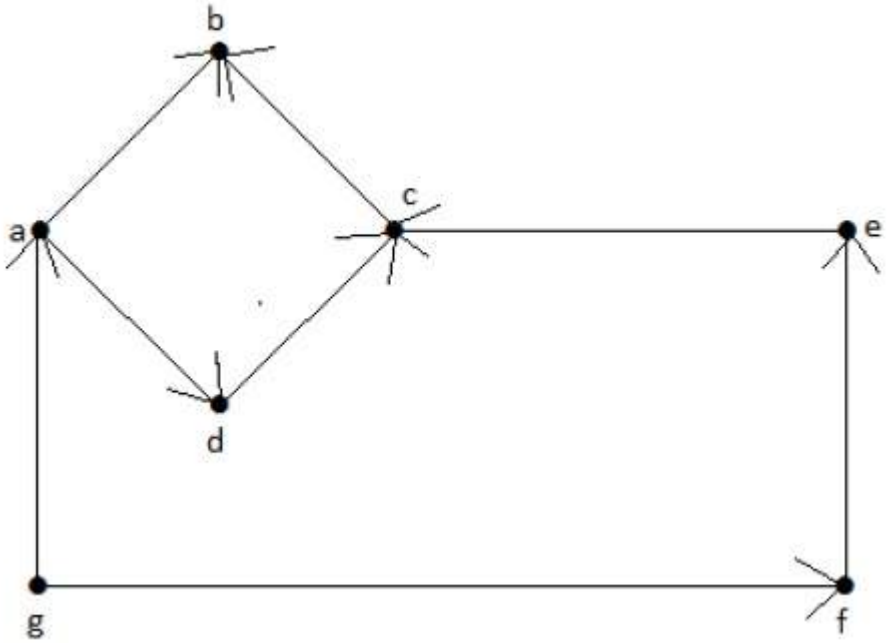

In this graph, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’ are the vertices, and ‘ab’, ‘bc’, ‘cd’, ‘da’, ‘ag’, ‘gf’, ‘ef’ are the

edges of the graph. Since it is a non-directed graph, the edges ‘ab’ and ‘ba’ are same. Similarly other edges also considered in the same way.

## Directed Graph

In a directed graph, each edge has a direction.

## Example




In the above graph, we have seven vertices ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, and ‘g’, and eight edges ‘ab’, ‘cb’, ‘dc’, ‘ad’, ‘ec’, ‘fe’, ‘gf’, and ‘ga’. As it is a directed graph, each edge bears an arrow mark that shows its direction. Note that in a directed graph, ‘ab’ is different from ‘ba’.

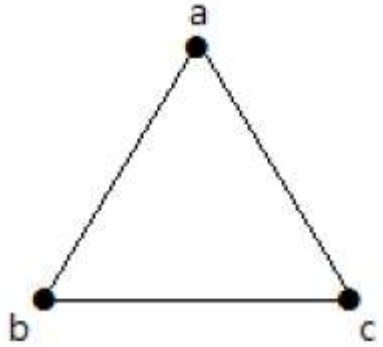
## Simple Graph

A graph **with no loops** and **no parallel edges** is called a simple graph.

- The maximum number of edges possible in a single graph with ‘n’ vertices is ${}^{n}C_{2}$ where ${}^{n}C_{2}$ = n$n-1$/2.
- The number of simple graphs possible with ‘n’ vertices = $2^{{}^{n}C_{2}} = 2^{nn-1/2}$.

## Example

In the following graph, there are 3 vertices with 3 edges which is maximum excluding the parallel edges and loops. This can be proved by using the above formulae.



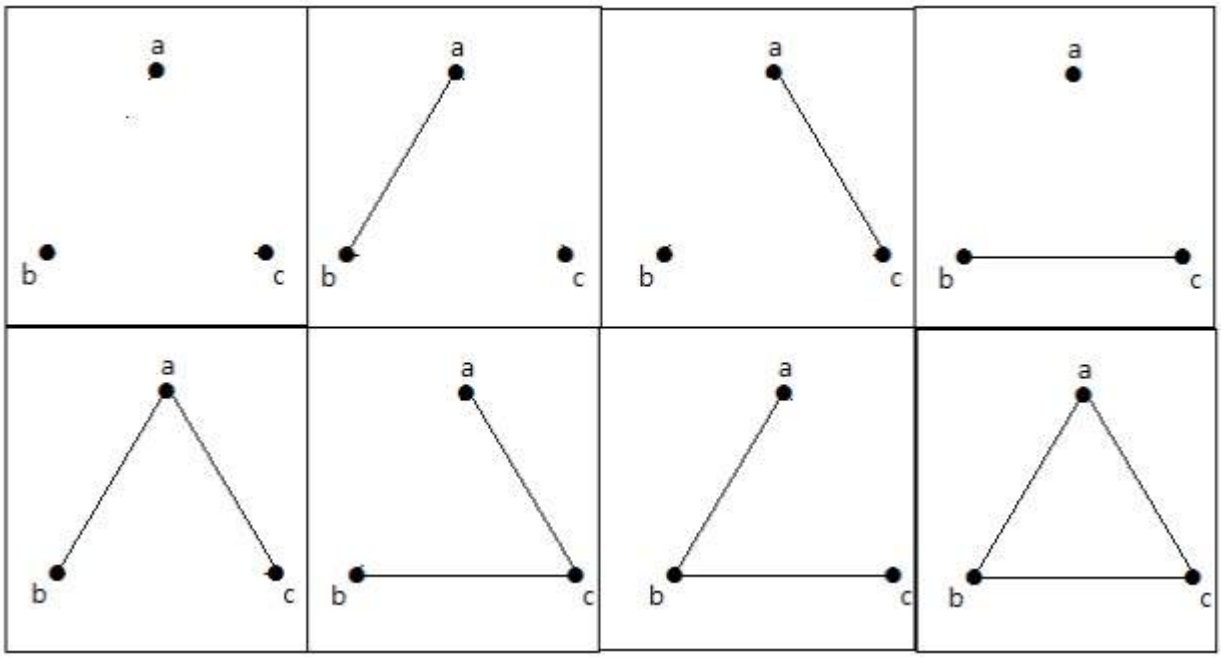

The maximum number of edges with n=3 vertices −

$^{n}C_2 = n(n-1)/2$
$= 3(3-1)/2$
$= 6/2$
$= 3$ edges

The maximum number of simple graphs with n=3 vertices −

$2^{^{n}C_2} = 2^{n(n-1)/2}$
$= 2^{3(3-1)/2}$
$= 2^3$
$= 8$

These 8 graphs are as shown below −

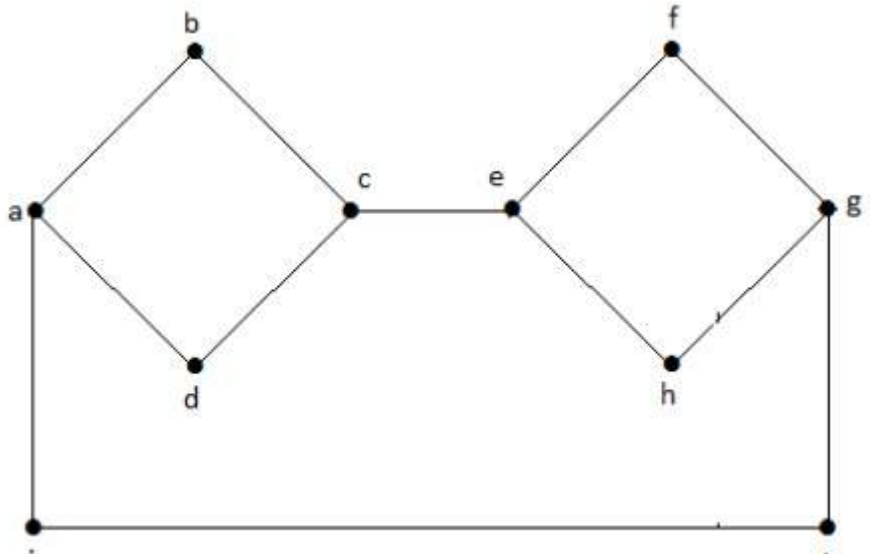



## Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

## Example

In the following graph, each vertex has its own edge connected to other edge. Hence it is a connected graph.

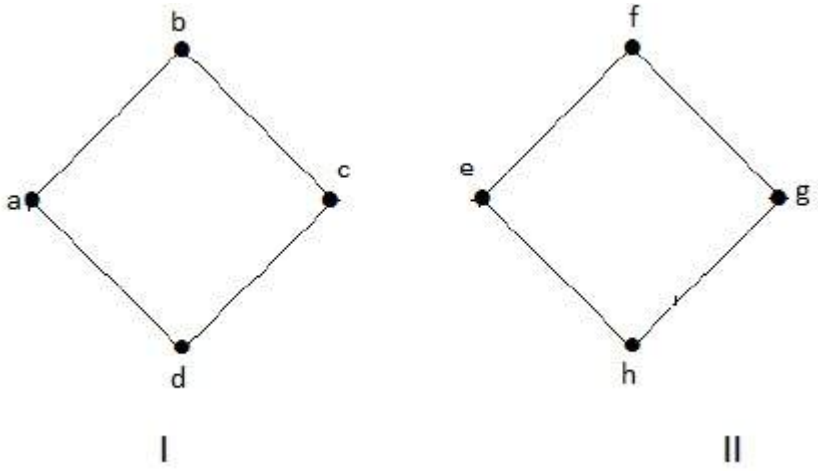


## Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

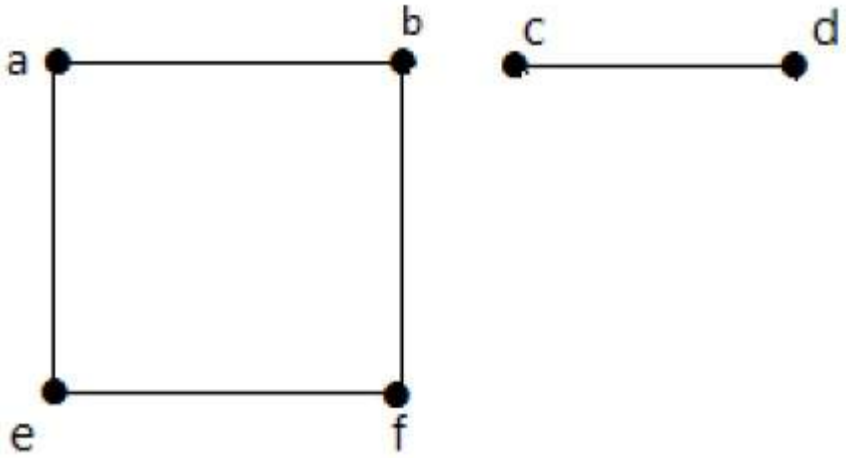
## Example 1

The following graph is an example of a Disconnected Graph, where there are two components, one with ‘a’, ‘b’, ‘c’, ‘d’ vertices and another with ‘e’, ’f’, ‘g’, ‘h’ vertices.




The two components are independent and not connected to each other. Hence it is called disconnected graph.

## Example 2



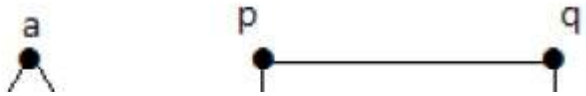

In this example, there are two independent components, a-b-f-e and c-d, which are not connected to each other. Hence this is a disconnected graph.

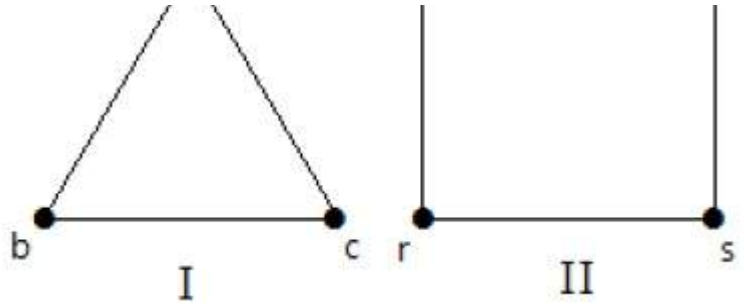
## Regular Graph

A graph G is said to be regular, **if all its vertices have the same degree**. In a graph, if the degree of each vertex is ‘k’, then the graph is called a ‘k-regular graph’.

## Example

In the following graphs, all the vertices have the same degree. So these graphs are called regular graphs.






In both the graphs, all the vertices have degree 2. They are called 2-Regular Graphs.

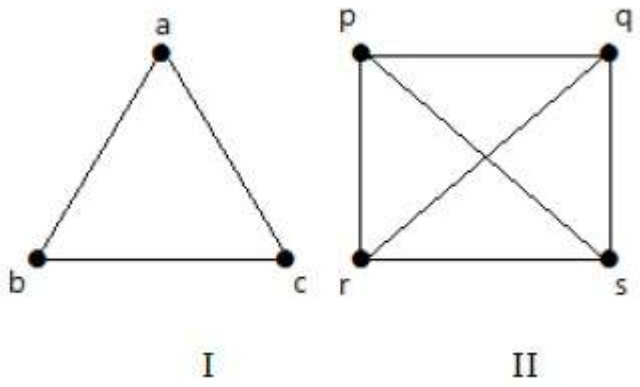
## Complete Graph

A simple graph with 'n' mutual vertices is called a complete graph and it is **denoted by '$K_n$'**. In the graph, **a vertex should have edges with all other vertices,** then it called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

## Example

In the following graphs, each vertex in the graph is connected with all the remaining vertices in the graph except by itself.




In graph I,

| | a | b | c |
|---|---|---|---|
| **a** | Not Connected | Connected | Connected |
| **b** | Connected | Not Connected | Connected |
| **c** | Connected | Connected | Not Connected |

In graph II,

| | p | q | r | s |
|---|---|---|---|---|
| **p** | Not Connected | Connected | Connected | Connected |
| **q** | Connected | Not Connected | Connected | Connected |
| **r** | Connected | Connected | Not Connected | Connected |
| **s** | Connected | Connected | Connected | Not Connected |

## Cycle Graph

A simple graph with ‘n’ vertices $n >= 3$ and ‘n’ edges is called a cycle graph if all its edges form a cycle of length ‘n’.

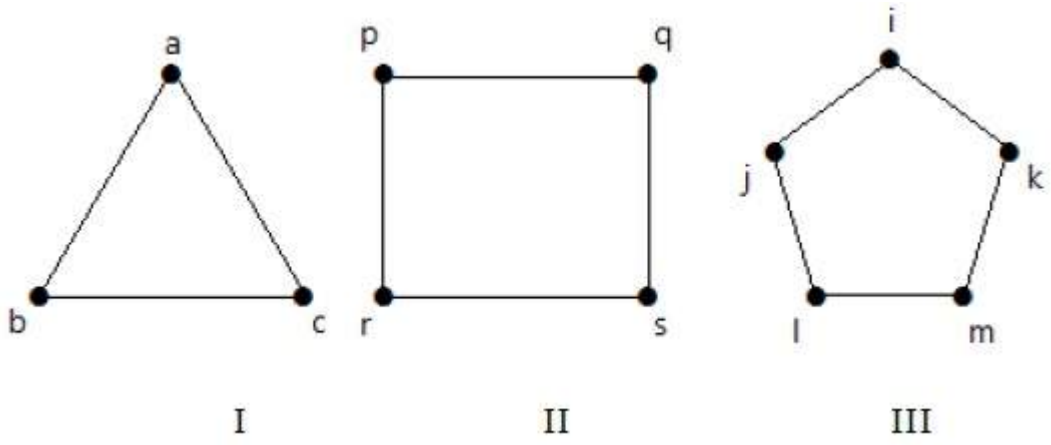
If the **degree of each vertex in the graph is two,** then it is called a Cycle Graph.

**Notation** − $C_n$

## Example

Take a look at the following graphs −

- Graph I has 3 vertices with 3 edges which is forming a cycle ‘ab-bc-ca’.
- Graph II has 4 vertices with 4 edges which is forming a cycle ‘pq-qs-sr-rp’.
- Graph III has 5 vertices with 5 edges which is forming a cycle ‘ik-km-ml-lj-ji’.




Hence all the given graphs are cycle graphs.

## Wheel Graph

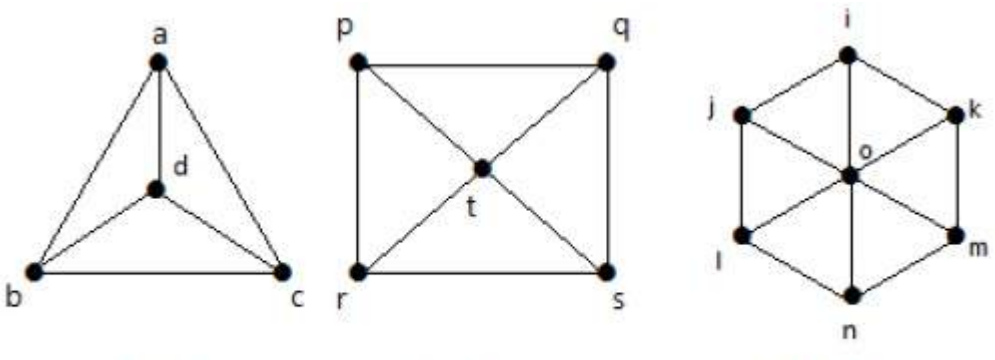
A wheel graph is obtained from a cycle graph $C_{n-1}$ by adding a new vertex. That new vertex is called a **Hub** which is connected to all the vertices of $C_n$.

**Notation** − $W_n$

```
No. of edges in Wn = No. of edges from hub to all other vertices &plus;
                     No. of edges from all other nodes in cycle graph without a hub.
                     = (n–1) + (n–1)
                     = 2(n–1)
```

## Example

Take a look at the following graphs. They are all wheel graphs.



I($W_4$) II($W_5$) III($W_7$)

In graph I, it is obtained from $C_3$ by adding an vertex at the middle named as 'd'. It is denoted as $W_4$.

```
Number of edges in W4 = 2(n-1) = 2(3) = 6
```

In graph II, it is obtained from $C_4$ by adding a vertex at the middle named as 't'. It is denoted as $W_5$.

```
Number of edges in W5 = 2(n-1) = 2(4) = 8
```

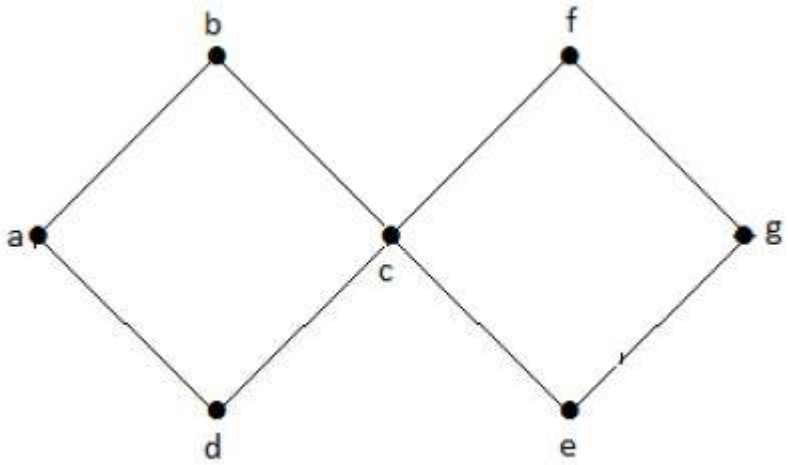
In graph III, it is obtained from $C_6$ by adding a vertex at the middle named as 'o'. It is denoted as $W_7$.

```
Number of edges in W4 = 2(n-1) = 2(6) = 12
```

## Cyclic Graph

A graph **with at least one** cycle is called a cyclic graph.

## Example

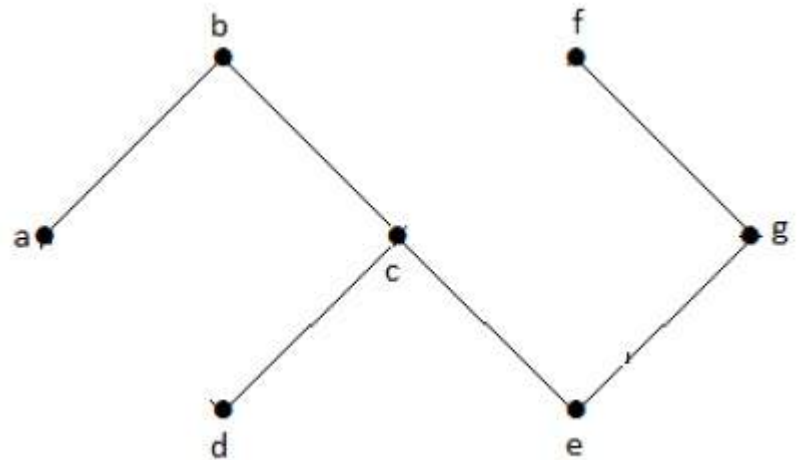



In the above example graph, we have two cycles a-b-c-d-a and c-f-g-e-c. Hence it is called a cyclic graph.

## Acyclic Graph

A graph **with no cycles** is called an acyclic graph.

## Example



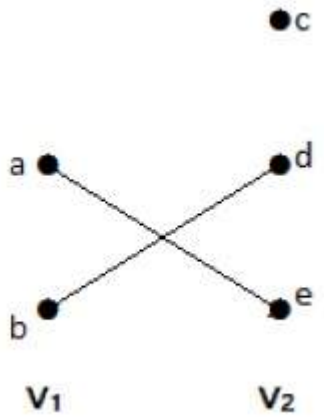
In the above example graph, we do not have any cycles. Hence it is a non-cyclic graph.

## Bipartite Graph

A simple graph G = $V, E$ with vertex partition V = {$V_1$, $V_2$} is called a bipartite graph **if every edge of E joins a vertex in $V_1$ to a vertex in $V_2$**.

In general, a Bipertite graph has two sets of vertices, let us say, $V_1$ and $V_2$, and if an edge is drawn, it should connect any vertex in set $V_1$ to any vertex in set $V_2$.

## Example




In this graph, you can observe two sets of vertices − $V_1$ and $V_2$. Here, two edges named 'ae' and 'bd' are connecting the vertices of two sets $V_1$ and $V_2$.

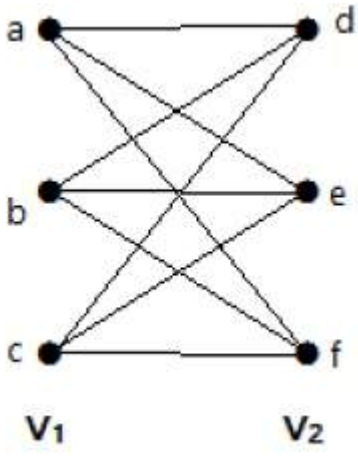
## Complete Bipartite Graph

A bipartite graph 'G', G = $V, E$ with partition V = {$V_1$, $V_2$} is said to be a complete bipartite graph if every vertex in $V_1$ is connected to every vertex of $V_2$.

In general, a complete bipartite graph connects each vertex from set $V_1$ to each vertex from set $V_2$.

## Example

The following graph is a complete bipartite graph because it has edges connecting each vertex from set $V_1$ to each vertex from set $V_2$.




If $|V_1|$ = m and $|V_2|$ = n, then the complete bipartite graph is denoted by $K_{m, n}$.

- $K_{m,n}$ has $m + n$ vertices and $mn$ edges.

- $K_{m,n}$ is a regular graph if m=n.

In general, **a complete bipartite graph is not a complete graph**.

$K_{m,n}$ is a complete graph if m=n=1.

The maximum number of edges in a bipartite graph with **n** vertices is

$$\lfloor n^2 / 4 \rfloor$$

If n=10, k5, 5= $\lfloor n^2 / 4 \rfloor = \lfloor 10^2 / 4 \rfloor = 25$

Similarly K6, 4=24

K7, 3=21

K8, 2=16

K9, 1=9

If n=9, k5, 4 = $\lfloor n^2 / 4 \rfloor = \lfloor 9^2 / 4 \rfloor = 20$

Similarly K6, 3=18

K7, 2=14

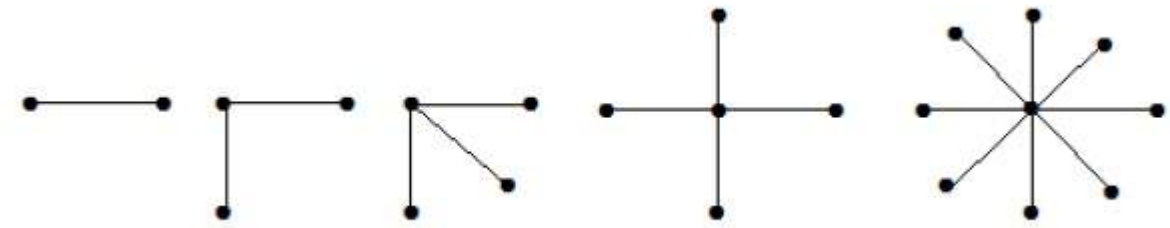
K8, 1=8

'G' is a bipartite graph if 'G' has no cycles of odd length. A special case of bipartite graph is a **star graph**.

## Star Graph

A complete bipartite graph of the form $K_{1, n-1}$ is a star graph with n-vertices. A star graph is a complete bipartite graph if a single vertex belongs to one set and all the remaining vertices belong to the other set.

## Example



In the above graphs, out of 'n' vertices, all the 'n–1' vertices are connected to a single vertex. Hence it is in the form of $K_{1, n-1}$ which are star graphs.

## Complement of a Graph

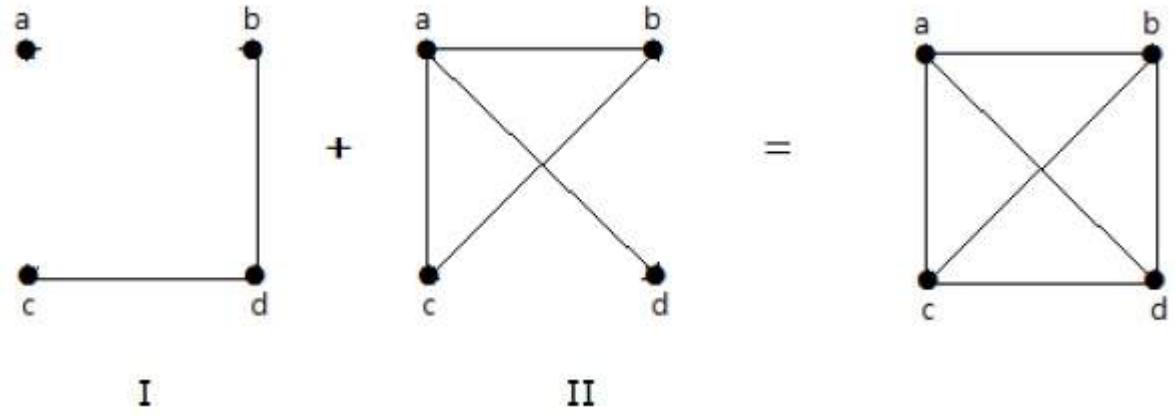
Let '−' if the two vertices are not adjacent in G.

If the edges that exist in graph I are absent in another graph II, and if both graph I and graph II are combined together to form a complete graph, then graph I and graph II are called complements of each other.

## Example

In the following example, graph-I has two edges 'cd' and 'bd'. Its complement graph-II has four

edges.




Note that the edges in graph-I are not present in graph-II and vice versa. Hence, the combination of both the graphs gives a complete graph of ‘n’ vertices.

**Note** − A combination of two complementary graphs gives a complete graph.

If ‘G’ is any simple graph, then

$|EG| + |E('-')| = |E(K_n)|$, where n = number of vertices in the graph.

## Example

Let ‘G’ be a simple graph with nine vertices and twelve edges, find the number of edges in '-'.

You have, $|EG| + |E('-')| = |E(K_n)|$

12 + |E('

$9^{9-1}/2 = {}^{9}C_2$

*12 + |E('-')| = 36*

*|E('-')| = 24*

*‘G’ is a simple graph with 40 edges and its complement* '−'.

*Let the number of vertices in the graph be ‘n’.*

*We have,* $|EG| + |E('-')| = |E(K_n)|$

$40 \+ 38 = n^{n-1}/2$

$156 = n^{n-1}$

$13^{12} = n^{n-1}$

$n = 13$

# GRAPH THEORY - TREES

*Trees are graphs that do not contain even a single cycle. They represent hierarchical structure in a graphical form. Trees belong to the simplest class of graphs. Despite their simplicity, they have a rich structure.*

*Trees provide a range of useful applications as simple as a family tree to as complex as trees in data structures of computer science.*

## *Tree*

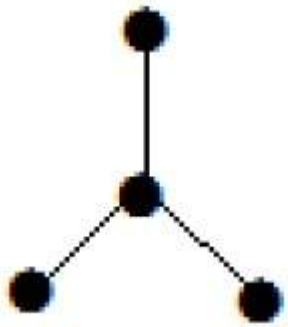
*A **connected acyclic graph** is called a tree. In other words, a connected graph with no cycles is called a tree.*

*The edges of a tree are known as **branches**. Elements of trees are called their **nodes**. The nodes without child nodes are called **leaf nodes**.*

*A tree with 'n' vertices has 'n-1' edges. If it has one more edge extra than 'n-1', then the extra edge should obviously has to pair up with two vertices which leads to form a cycle. Then, it becomes a cyclic graph which is a violation for the tree graph.*

## *Example 1*

*The graph shown here is a tree because it has no cycles and it is connected. It has four vertices and three edges, i.e., for 'n' vertices 'n-1' edges as mentioned in the definition.*



***Note** − Every tree has at least two vertices of degree one.*

## *Example 2*

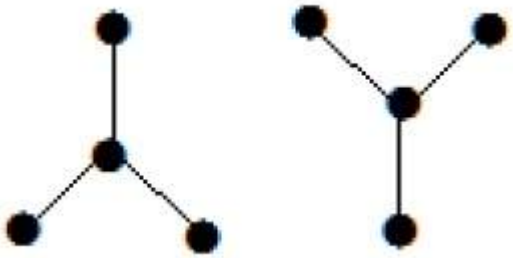

*In the above example, the vertices 'a' and 'd' has degree one. And the other two vertices 'b' and 'c' has degree two. This is possible because for not forming a cycle, there should be at least two single edges anywhere in the graph. It is nothing but two edges with a degree of one.*

## *Forest*

*A **disconnected acyclic graph** is called a forest. In other words, a disjoint collection of trees is called a forest.*

## *Example*

*The following graph looks like two sub-graphs; but it is a single disconnected graph. There are no cycles in this graph. Hence, clearly it is a forest.*



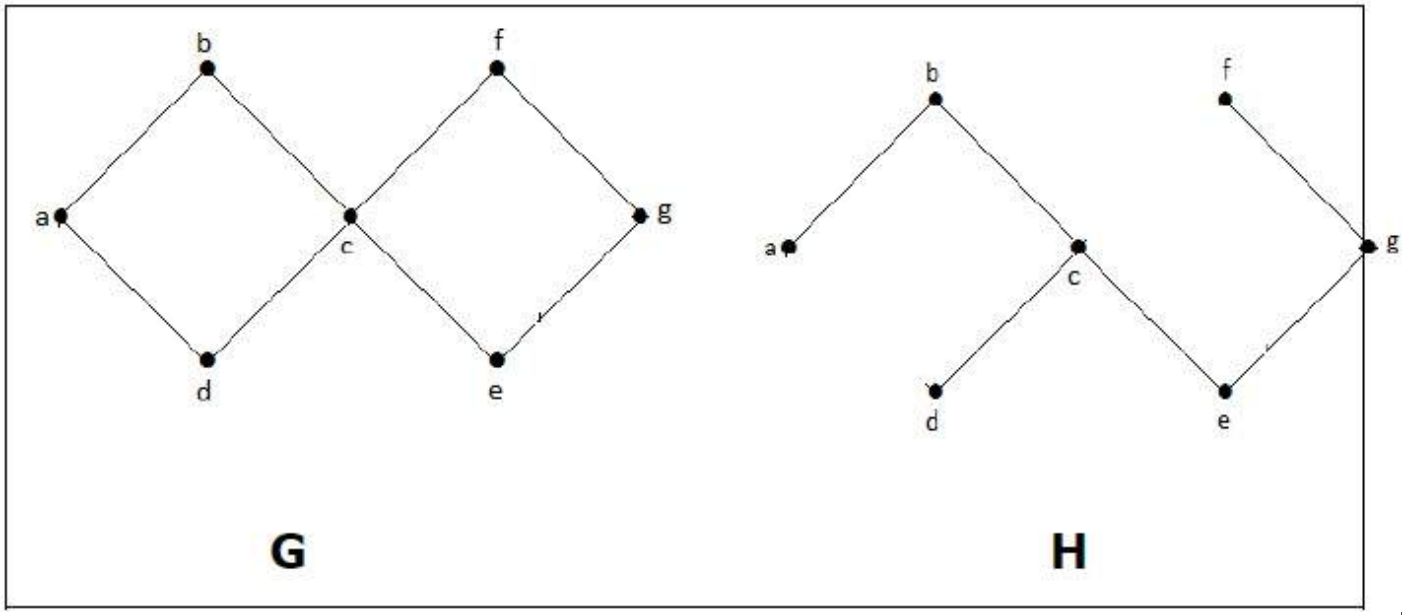
## *Spanning Trees*

*Let G be a connected graph, then the sub-graph H of G is called a spanning tree of G if −*

- *H is a tree*
- *H contains all vertices of G.*

*A spanning tree T of an undirected graph G is a subgraph that includes all of the vertices of G.*

## *Example*




*In the above example, G is a connected graph and H is a sub-graph of G.*

*Clearly, the graph H has no cycles, it is a tree with six edges which is one less than the total number of vertices. Hence H is the Spanning tree of G.*

## *Circuit Rank*

*Let 'G' be a connected graph with 'n' vertices and 'm' edges. A spanning tree 'T' of G contains* $n - 1$ *edges.*

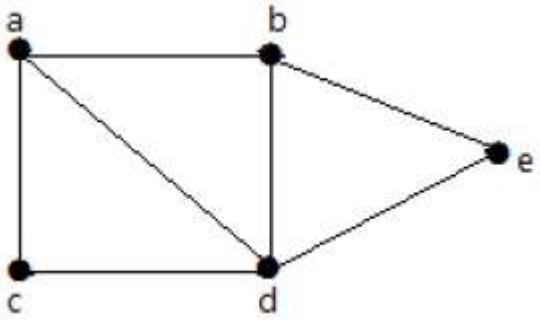
*Therefore, the number of edges you need to delete from 'G' in order to get a spanning tree = m-* $n - 1$*, which is called the circuit rank of G.*

*This formula is true, because in a spanning tree you need to have 'n-1' edges. Out of 'm' edges, you need to keep 'n–1' edges in the graph.*

*Hence, deleting 'n–1' edges from 'm' gives the edges to be removed from the graph in order to get a spanning tree, which should not form a cycle.*

## *Example*

*Take a look at the following graph −*




*For the graph given in the above example, you have m=7 edges and n=5 vertices.*

*Then the circuit rank is*

```
G = m – (n – 1)
  = 7 – (5 – 1)
  = 3
```

## *Example*

*Let 'G' be a connected graph with six vertices and the degree of each vertex is three. Find the circuit rank of 'G'.*

*By the sum of degree of vertices theorem,*

$$n \sum i=1 \ deg(V_i) = 2|E|$$

$$6 \times 3 = 2|E|$$

$$|E| = 9$$

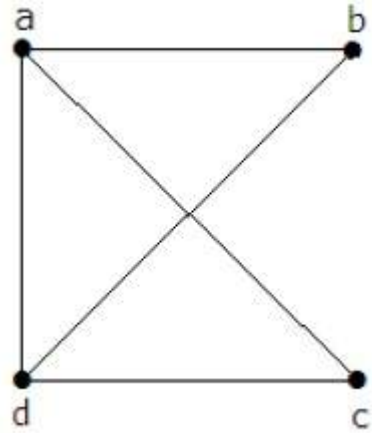
*Circuit rank* $= |E| - |V| - 1$

$$= 9 - 6 - 1 = 4$$

## *Kirchoff's Theorem*

*Kirchoff's theorem is useful in finding the number of spanning trees that can be formed from a connected graph.*

## *Example*




*The matrix 'A' be filled as, if there is an edge between two vertices, then it should be given as '1', else '0'.*

$$A = \begin{vmatrix} 0 & a & b & c & d \\ a & 0 & 1 & 1 & 1 \\ b & 1 & 0 & 0 & 1 \\ c & 1 & 0 & 0 & 1 \\ d & 1 & 1 & 1 & 0 \end{vmatrix} = \begin{vmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{vmatrix}$$

By using kirchoff's theorem, it should be changed as replacing the principle diagonal values with the degree of vertices and all other elements with -1.A

$$= \begin{vmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{vmatrix} = M$$

$$\begin{vmatrix} 3 & -1 & -1 & -1 \end{vmatrix}$$

$$M = \begin{vmatrix} -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{vmatrix}$$

$$\text{Cofactor of m1} = \begin{vmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ -1 & -1 & 3 \end{vmatrix} = 8$$

Thus, the number of spanning trees = 8.

# GRAPH THEORY - CONNECTIVITY

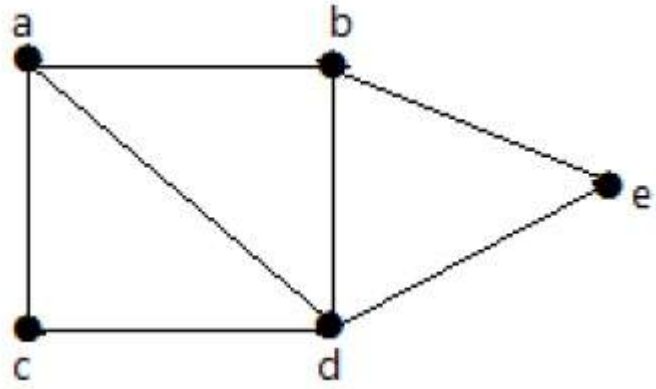
*Whether it is possible to traverse a graph from one vertex to another is determined by how a graph is connected. Connectivity is a basic concept in Graph Theory. Connectivity defines whether a graph is connected or disconnected. It has subtopics based on edge and vertex, known as edge connectivity and vertex connectivity. Let us discuss them in detail.*

## Connectivity

*A graph is said to be **connected if there is a path between every pair of vertex**. From every vertex to any other vertex, there should be some path to traverse. That is called the connectivity of a graph. A graph with multiple disconnected vertices and edges is said to be disconnected.*

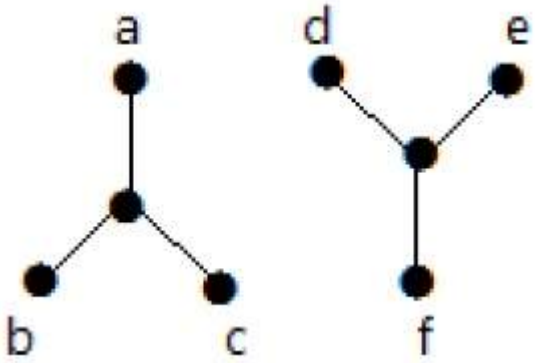
## Example 1

*In the following graph, it is possible to travel from one vertex to any other vertex. For example, one can traverse from vertex 'a' to vertex 'e' using the path 'a-b-e'.*




## Example 2

*In the following example, traversing from vertex 'a' to vertex 'f' is not possible because there is no path between them directly or indirectly. Hence it is a disconnected graph.*




## Cut Vertex

*Let 'G' be a connected graph. A vertex V ∈ G is called a cut vertex of 'G', if 'G-V'* $Delete`V'from`G'$

# Graph Representations

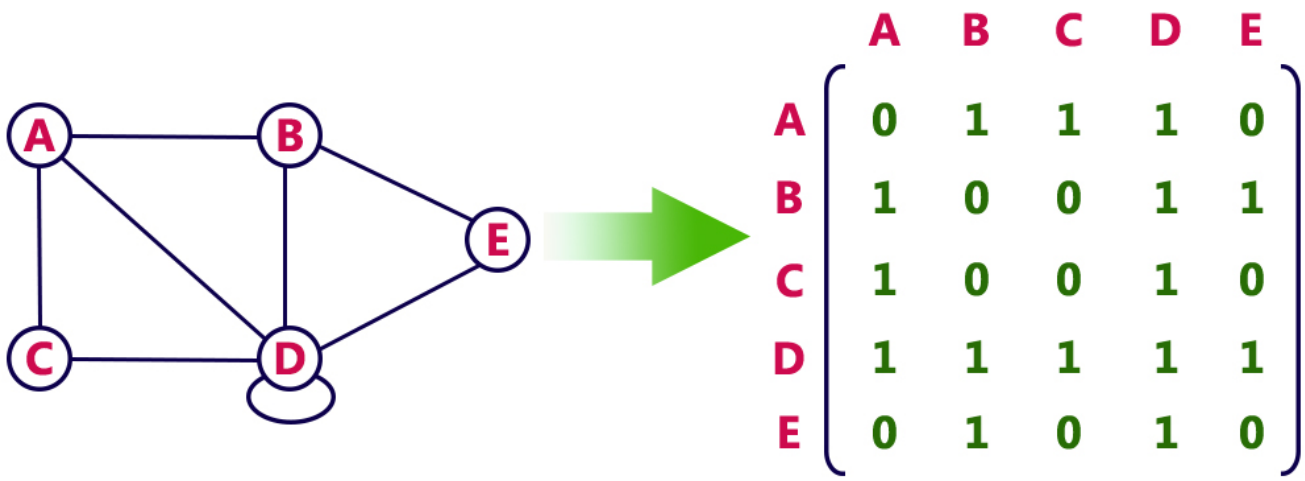
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

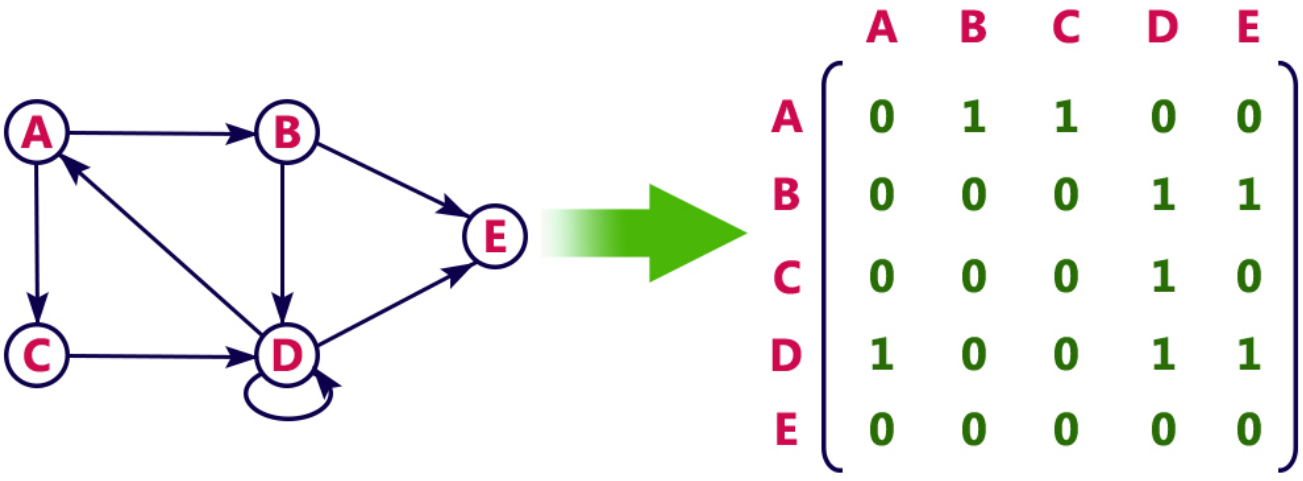
## Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



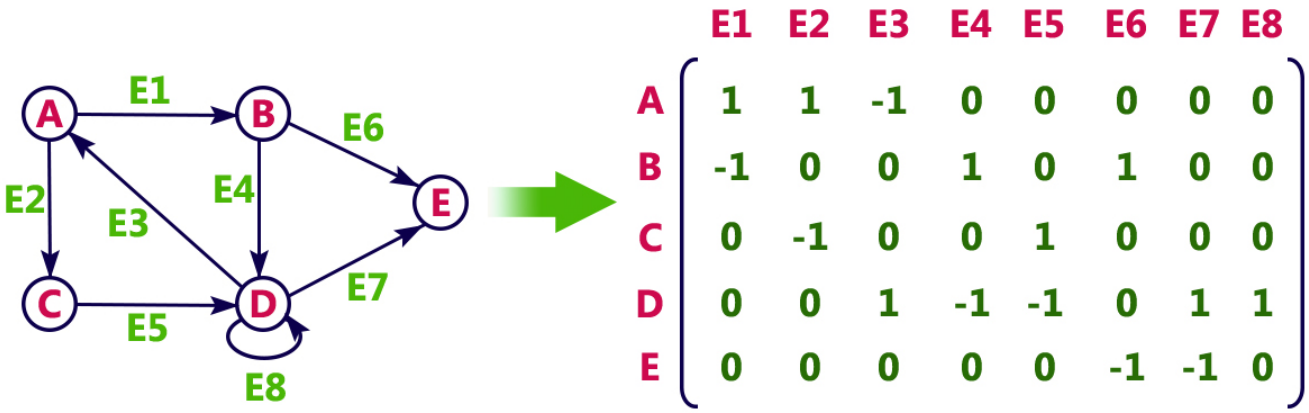

Directed graph representation...




## Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

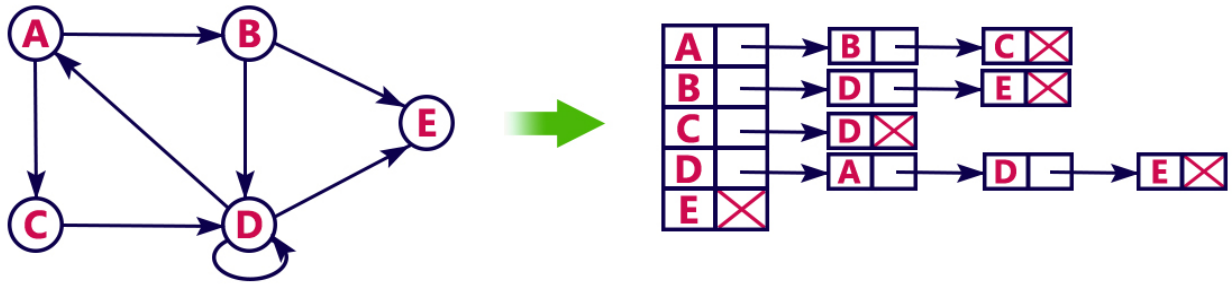
For example, consider the following directed graph representation...



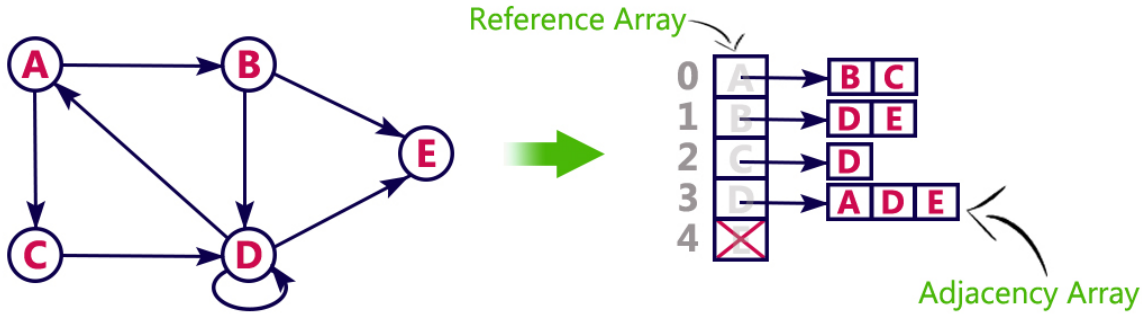

## Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



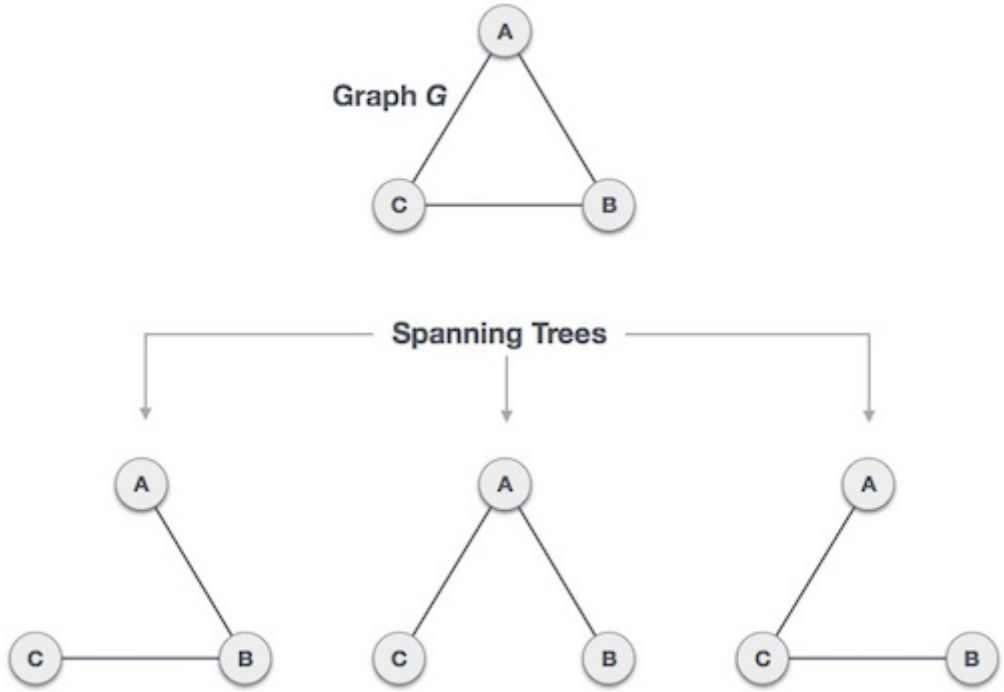

This representation can also be implemented using an array as follows..



# Data Structure & Algorithms - Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.




We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

## General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.

- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

## Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

## Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

## Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

## Minimum Spanning-Tree Algorithm

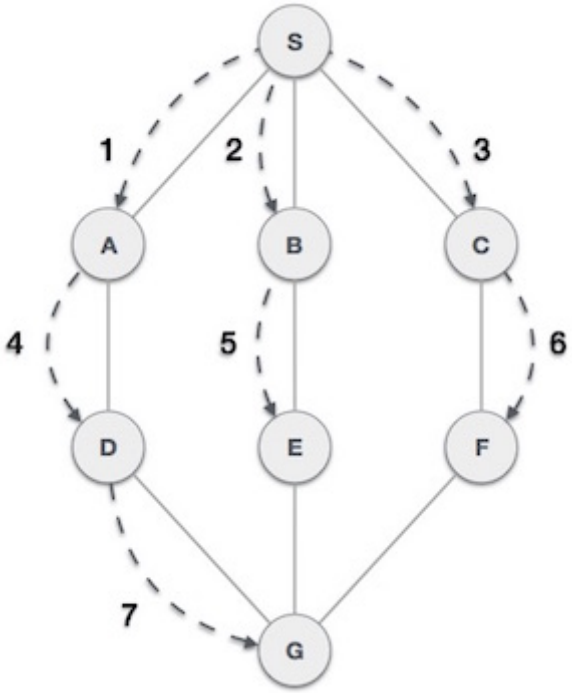
We shall learn about two most important spanning tree algorithms here −

- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

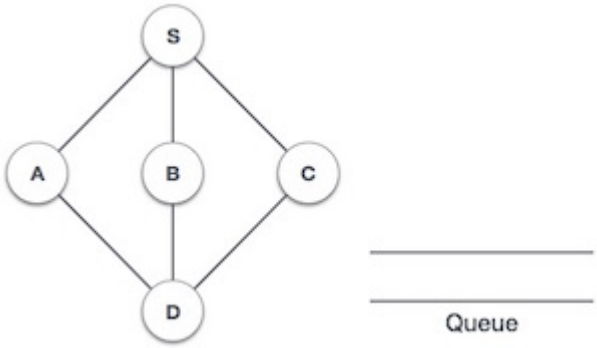
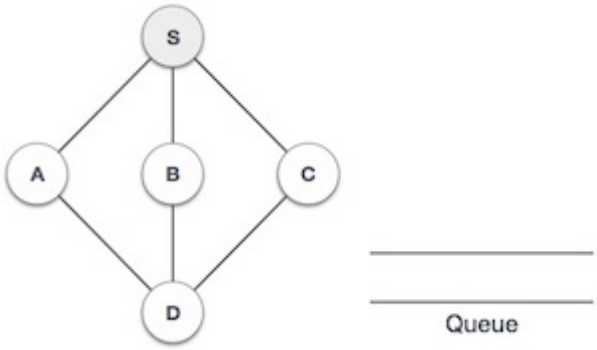
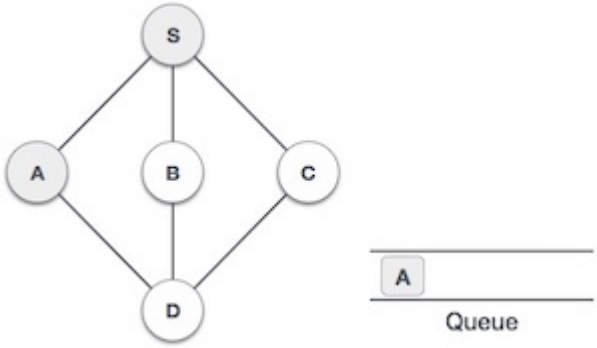
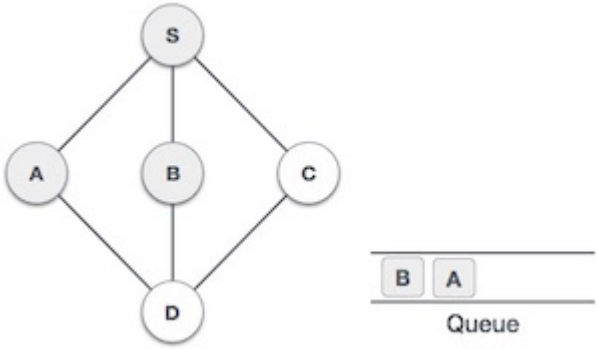
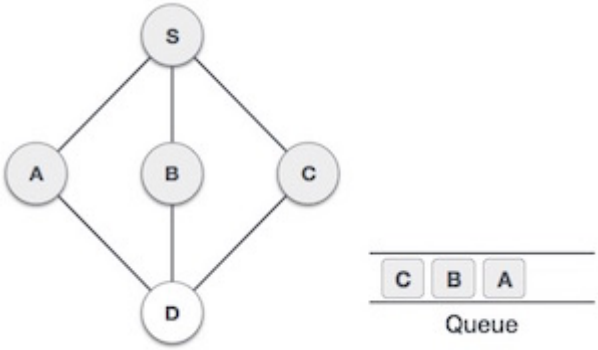
# Data Structure - Breadth First Traversal

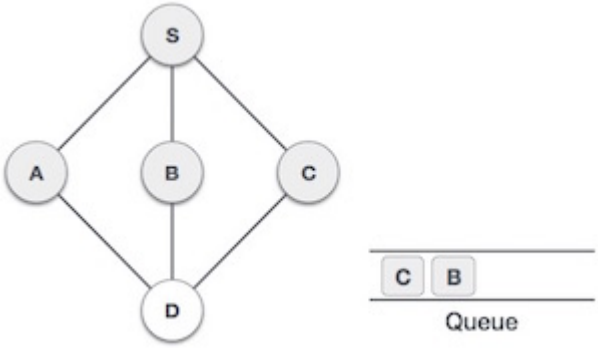
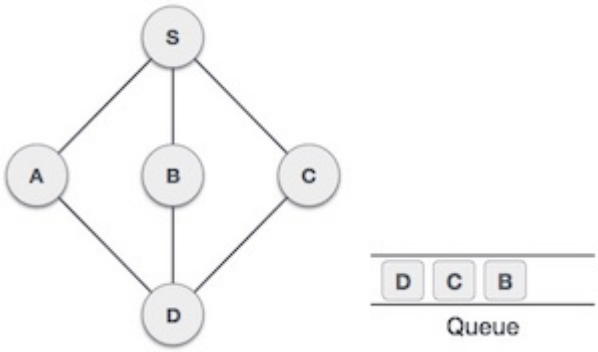
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.




As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

| Step | Traversal | Description |
| --- | --- | --- |
| 1 |  | Initialize the queue. |
| 2 |  | We start from visiting S (starting node), and mark it as visited. |
| 3 |  | We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it. |
| 4 |  | Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it. |
| 5 |  | Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it. |

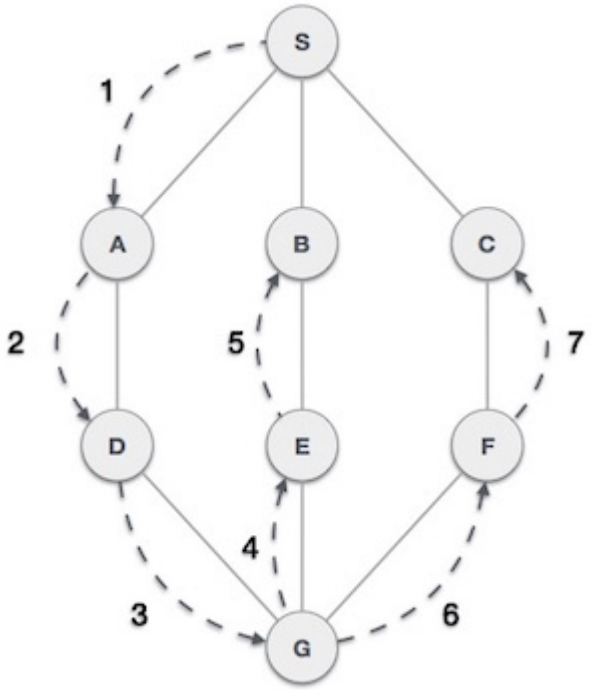
| | | |
|---|---|---|
| 6 |  | Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A. |
| 7 |  | From A we have D as unvisited adjacent node. We mark it as visited and enqueue it. |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

The implementation of this algorithm in C programming language can be seen here .

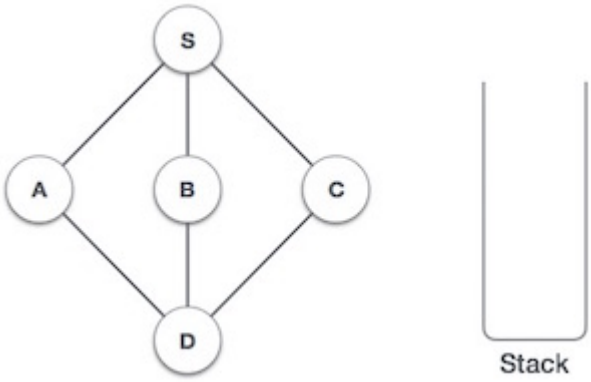
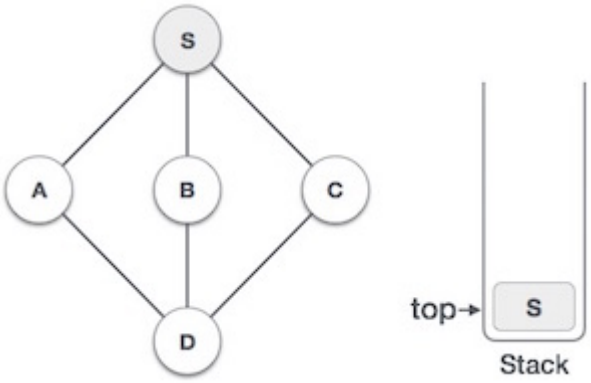
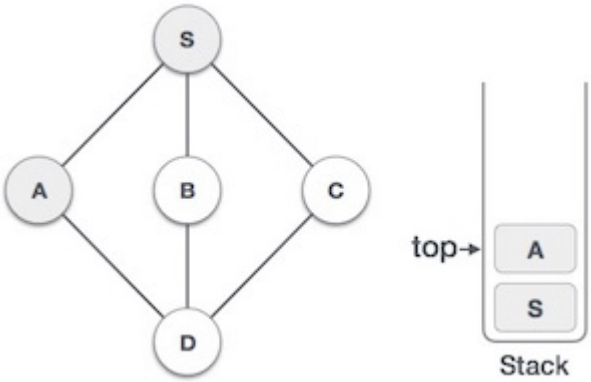
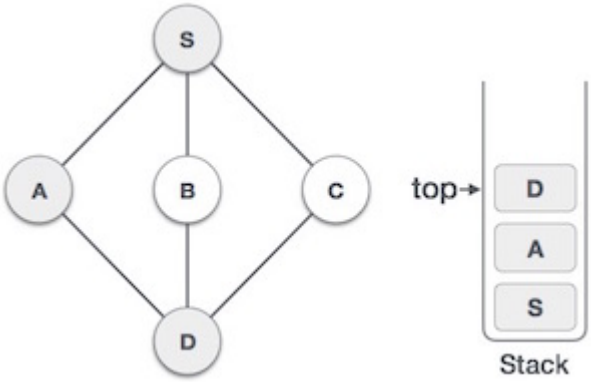
# Data Structure - Depth First Traversal

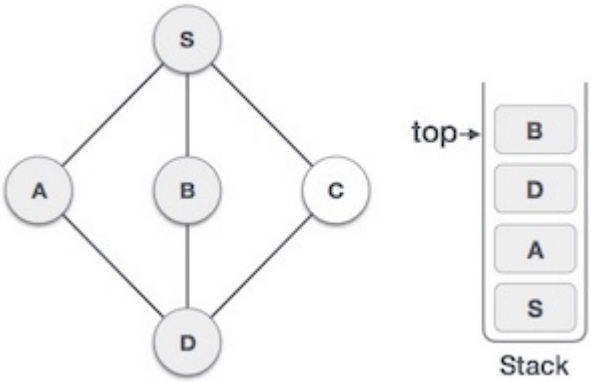
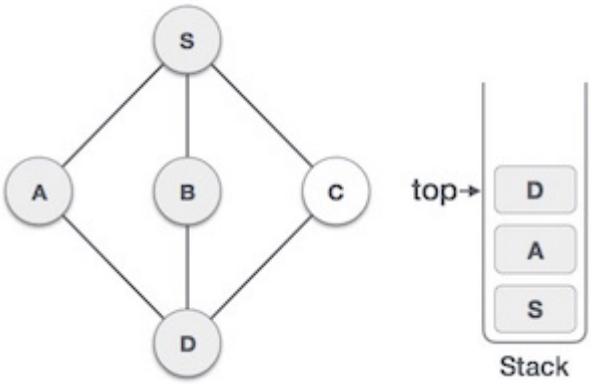
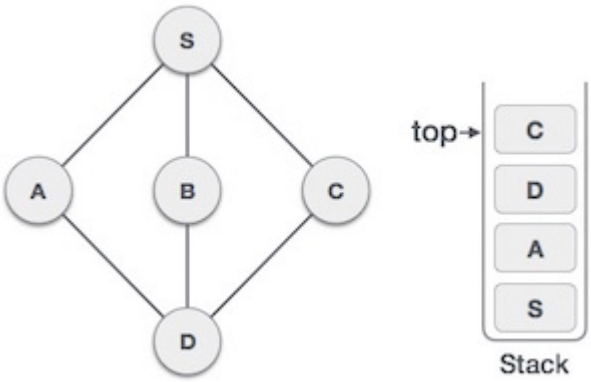
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.




As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

| Step | Traversal | Description |
|---|---|---|
| 1 |  | Initialize the stack. |
| 2 |  | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. |
| 3 |  | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only. |
| 4 |  | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. |
| 5 | | We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack. |

| | | |
|---|---|---|
| |  | |
| 6 |  | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. |
| 7 |  | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

To know about the implementation of this algorithm in C programming language, click here .

# Graph Traversal - BFS

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

## BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

**Step 1 -** Define a Queue of size total number of vertices in the graph.

**Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

**Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

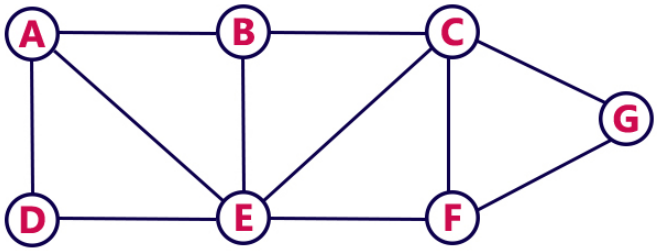
**Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

**Step 5 -** Repeat steps 3 and 4 until queue becomes empty.

**Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

## Example

Consider the following example graph to perform BFS traversal




**Step 1:**

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.


**Queue**

# Graph Traversal - DFS

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

## DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

**Step 1 -** Define a Stack of size total number of vertices in the graph.

**Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

**Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

**Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

**Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

**Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.

**Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

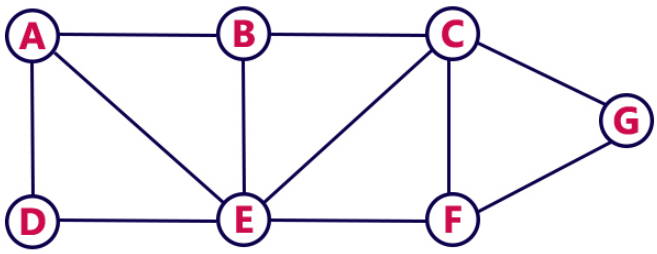
---

**Back tracking** is coming back to the vertex from which we reached the current vertex.

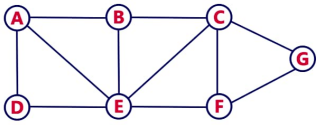
---

## Example

Consider the following example graph to perform DFS traversal

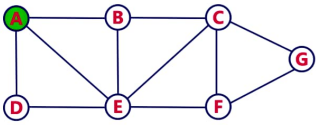


Consider the following example graph to perform BFS traversal




**Step 1:**

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

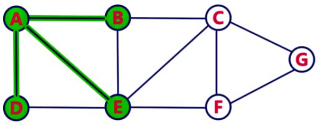



**Queue**

| A | | | | | | |
|---|---|---|---|---|---|---|

**Step 2:**

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

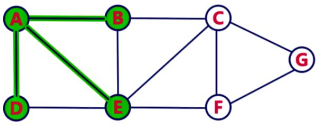



**Queue**

| | D | E | B | | | |
|---|---|---|---|---|---|---|

**Step 3:**

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

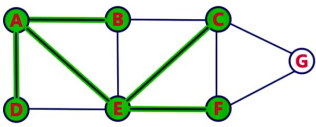



**Queue**

| | | E | B | | | |
|---|---|---|---|---|---|---|

**Step 4:**

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

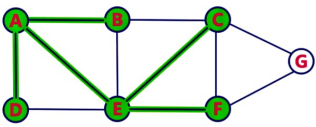



**Queue**

| | | | B | C | F | |
|---|---|---|---|---|---|---|

**Step 5:**

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

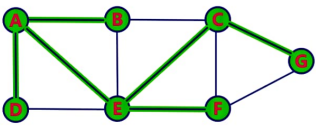



**Queue**

| | | | | C | F | |
|---|---|---|---|---|---|---|

**Step 6:**

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

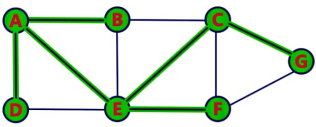



**Queue**

| | | | | | F | G |
|---|---|---|---|---|---|---|

**Step 7:**

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

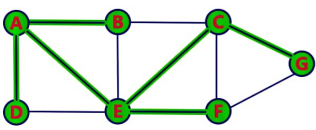



**Queue**

| | | | | | | G |
|---|---|---|---|---|---|---|

**Step 8:**

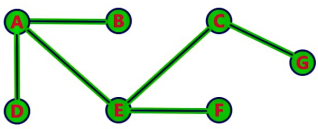
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



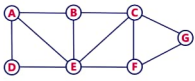

**Queue**

| | | | | | | |
|---|---|---|---|---|---|---|

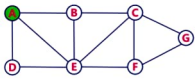
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



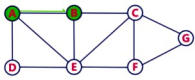
Consider the following example graph to perform DFS traversal



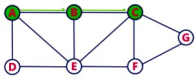

**Step 1:**
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



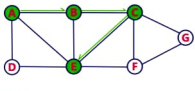

**Step 2:**
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



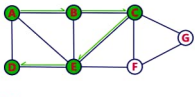

**Step 3:**
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



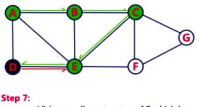

**Step 4:**
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack




**Step 5:**
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



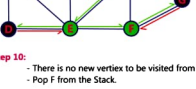

**Step 6:**
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.




**Step 7:**
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push F on to the Stack.


**Step 8:**
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

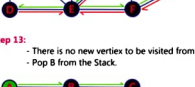

**Step 9:**
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



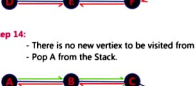

**Step 10:**
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.


**Step 11:**
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

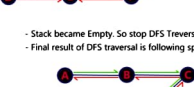

**Step 12:**
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



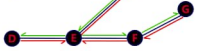

**Step 13:**
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.




**Step 14:**
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



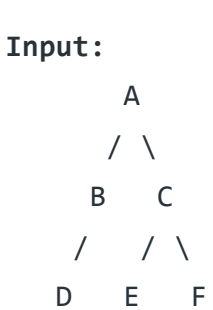
# Difference between BFS and DFS

Difficulty Level : Easy  Last Updated : 23 Jun, 2022

**Breadth-First Search:**

**BFS** stands for **Breadth-First Search** is a vertex-based technique for finding the shortest path in the graph. It uses a Queue data structure that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

**Example**:



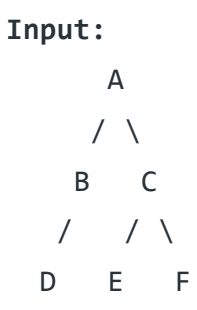

**Output:**

```
A, B, C, D, E, F
```

**Depth First Search:**

**DFS** stands for **Depth First Search** is an edge-based technique. It uses the Stack data structure and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.

**Example:**




**Output:**

```
A, B, D, C, E, F
```

**BFS vs DFS**

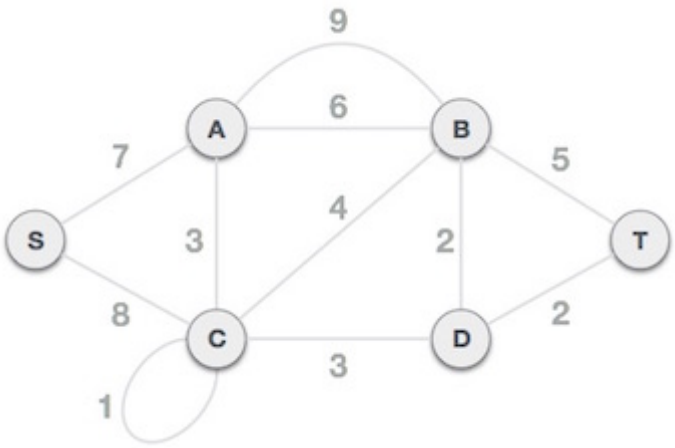
| S. No. | Parameters | BFS | DFS |
|---|---|---|---|
| 1. | Stands for | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |
| 2. | Data | BFS(Breadth First Search) uses Queue data | DFS(Depth First Search) uses Stack data structure. |

| | | | |
|---|---|---|---|
| | Structure | structure for finding the shortest path. | |
| 3. | Definition | BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level. | DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes. |
| 4. | Technique | BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex. | In DFS, we might traverse through more edges to reach a destination vertex from a source. |
| 5. | Conceptual Difference | BFS builds the tree level by level. | DFS builds the tree sub-tree by sub-tree. |
| 6. | Approach used | It works on the concept of FIFO (First In First Out). | It works on the concept of LIFO (Last In First Out). |
| 7. | Suitable for | BFS is more suitable for searching vertices which are closer to the given source. | DFS is more suitable when there are solutions away from source. |
| 8. | Suitable for Decision Tress | BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles. | DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop. |
| 9. | Time Complexity | The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. | The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. |
| 10. | Visiting of Siblings/ Children | Here, siblings are visited before the children. | Here, children are visited before the siblings. |
| 11. | Removal of Traversed Nodes | Nodes that are traversed several times are deleted from the queue. | The visited nodes are added to the stack and then removed when there are no more nodes to visit. |
| 12. | Backtracking | In BFS there is no concept of backtracking. | DFS algorithm is a recursive algorithm that uses the idea of backtracking |
| 13. | Applications | BFS is used in various application such as bipartite graph, and shortest path etc. | DFS is used in various application such as acyclic graph and topological order etc. |
| 14. | Memory | BFS requires more memory. | DFS requires less memory. |
| 15. | Optimality | BFS is optimal for finding the shortest path. | DFS is not optimal for finding the shortest path. |
| 16. | Space complexity | In BFS, the space complexity is more critical as compared to time complexity. | DFS has lesser space complexity, because at a time it needs to store only single path from the root to leaf node. |
| 17. | Speed | BFS is slow as compared to DFS. | DFS is fast as compared to BFS. |
| 18. | When to use? | When the target is close to the source, BFS performs better. | When the target is far from the source, DFS is preferable. |

# Kruskal's Spanning Tree Algorithm

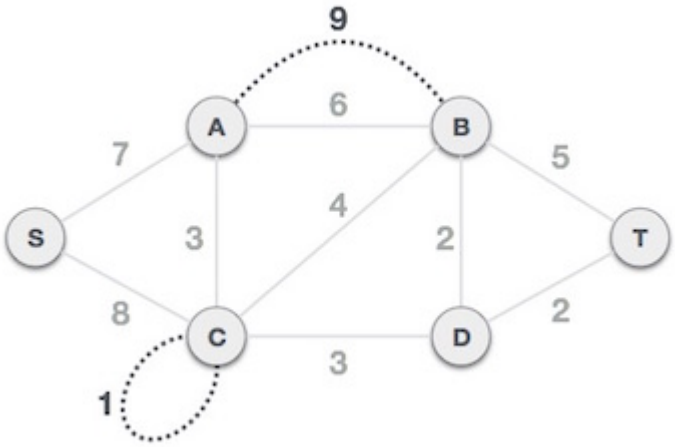
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

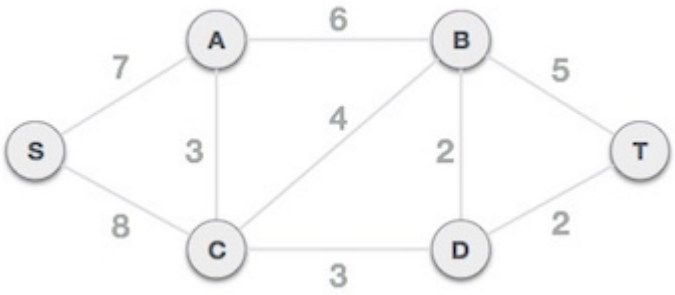



## Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.




In case of parallel edges, keep the one which has the least cost associated and remove all others.



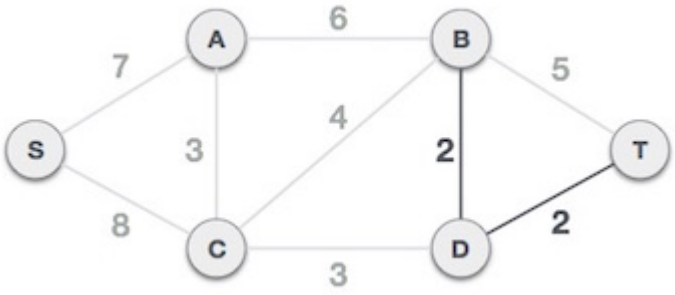

## Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |

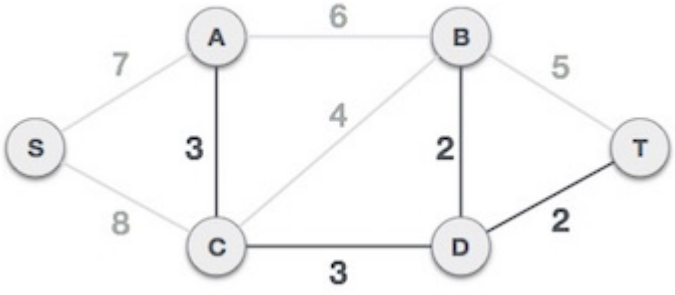
## Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

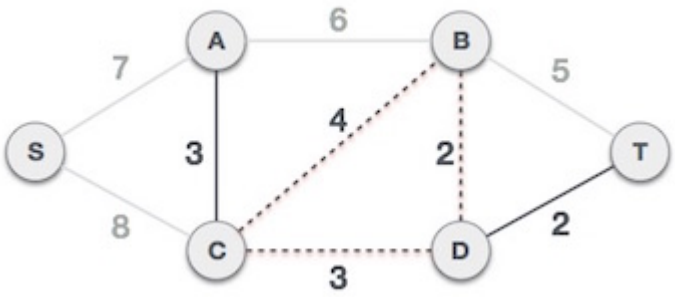



The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

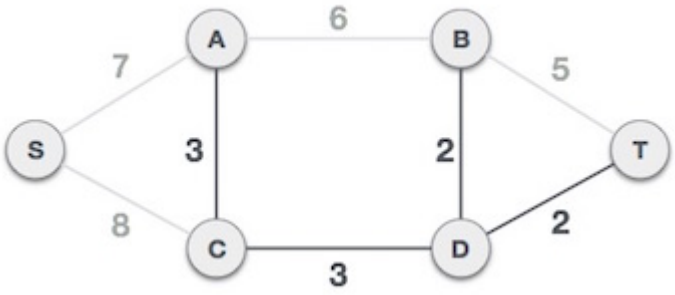
Next cost is 3, and associated edges are A,C and C,D. We add them again −



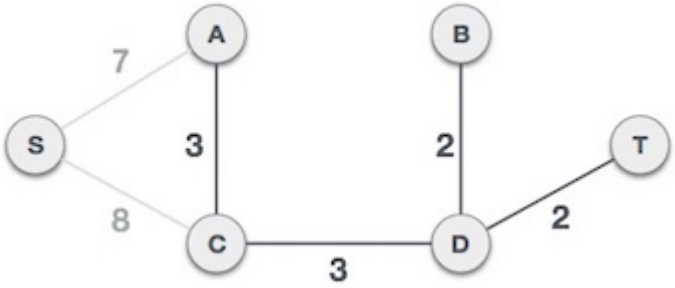

Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



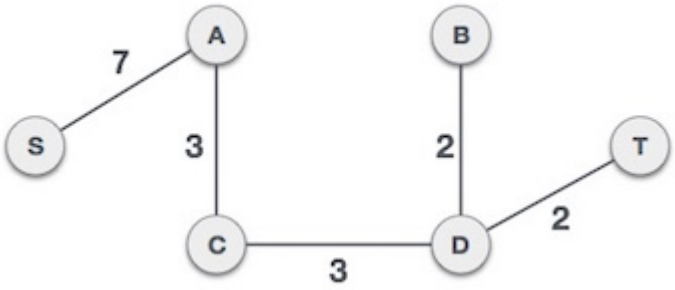

We ignore it. In the process we shall ignore/avoid all edges that create a circuit.




We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.




Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



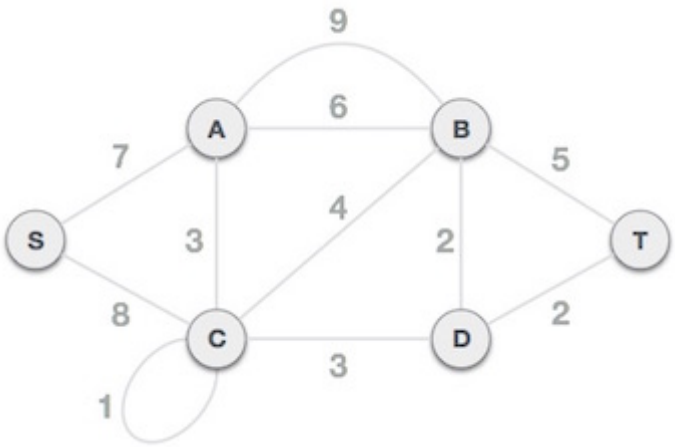

By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

# Prim's Spanning Tree Algorithm

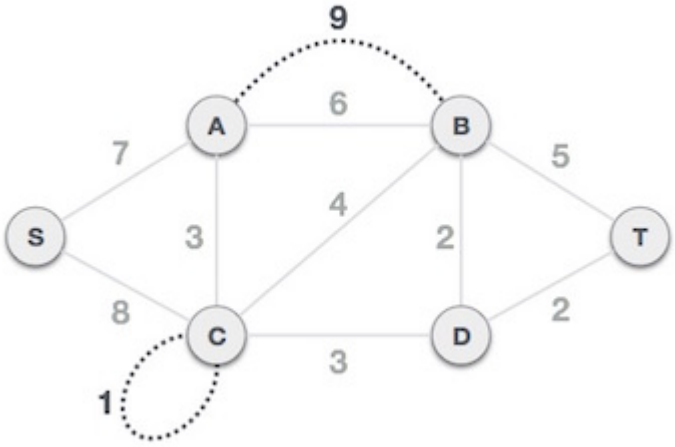
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

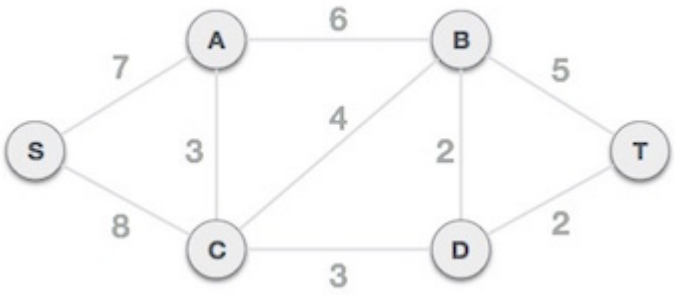
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −




## Step 1 - Remove all loops and parallel edges




Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

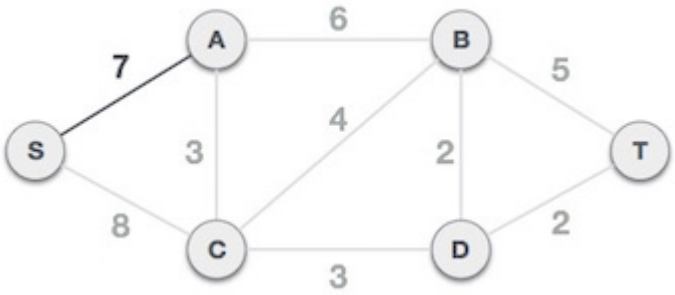



## Step 2 - Choose any arbitrary node as root node

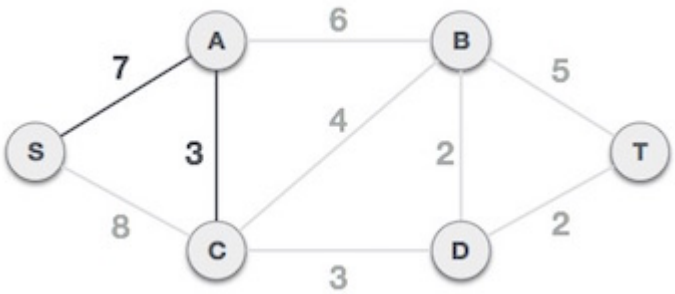
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

## Step 3 - Check outgoing edges and select the one with less cost

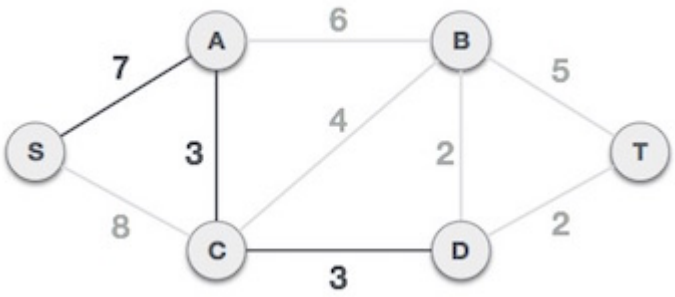
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



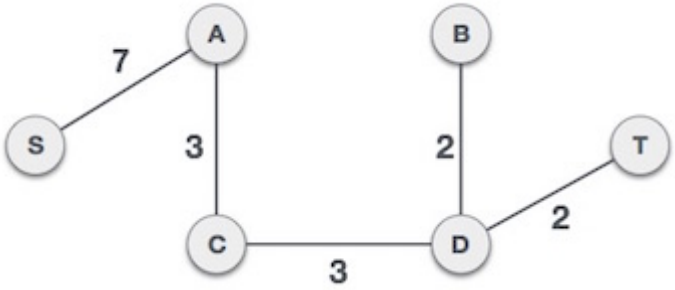

Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.




After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.




After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.




We may find that the output spanning tree of the same graph using two different algorithms is same.

# Difference between Prim's and Kruskal's algorithm for MST



## Kruskal's algorithm for MST

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

**Below are the steps for finding MST using Kruskal's algorithm**

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning-tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

## Prim's algorithm for MST

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

**Below are the steps for finding MST using Prim's algorithm**

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

3. While mstSet doesn't include all vertices
    - Pick a vertex u which is not there in mstSet and has minimum key value.
    - Include u to mstSet.
    - Update the key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v

Both **Prim's and Kruskal's algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few **major differences between them**.

| **Prim's Algorithm** | **Kruskal's Algorithm** |
|---|---|
| It starts to build the Minimum Spanning Tree from any vertex in the graph. | It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph. |
| It traverses one node more than one time to get the minimum distance. | It traverses one node only once. |
| Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps. | Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices. |
| Prim's algorithm gives connected component as well as it works only on connected graph. | Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components |
| Prim's algorithm runs faster in dense graphs. | Kruskal's algorithm runs faster in sparse graphs. |
| It generates the minimum spanning tree starting from the root vertex. | It generates the minimum spanning tree starting from the least weighted edge. |
| Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc. | Applications of Kruskal algorithm are LAN connection, TV Network etc. |
| Prim's algorithm prefer list data structures. | Kruskal's algorithm prefer heap data structures. |

# Why Prim's and Kruskal's MST algorithm fails for Directed Graph?

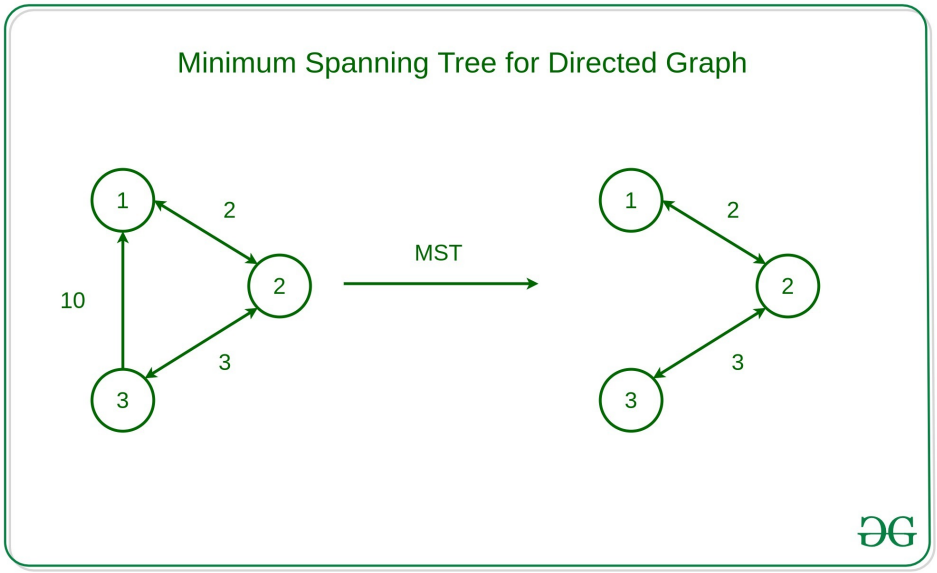
geeksforgeeks.org/why-prims-and-kruskals-mst-algorithm-fails-for-directed-graph

March 31, 2020

**Pre-requisites:**

Given a directed graph **D = < V, E >**, the task is to find the minimum spanning tree for the given directed graph

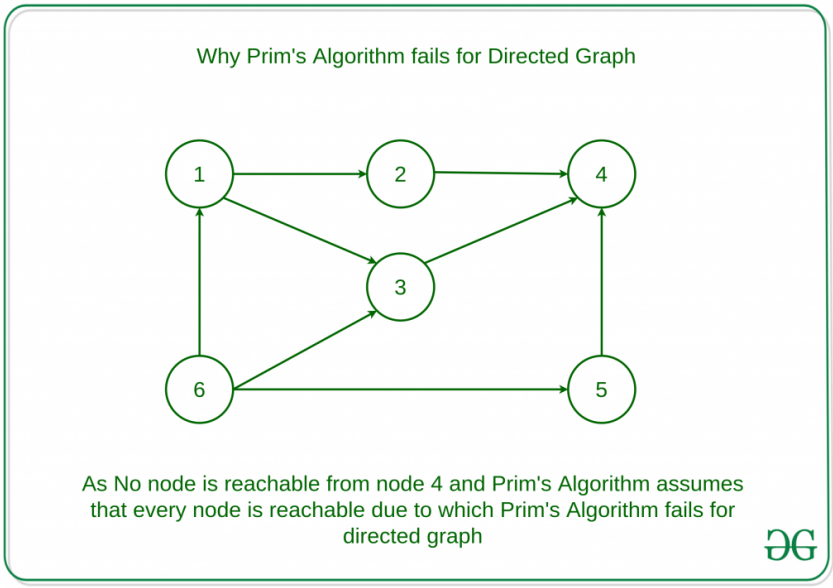
**Example:**




But the Prim's Minimum Spanning Tree and Kruskal's algorithm fails for directed graphs. Let us see why

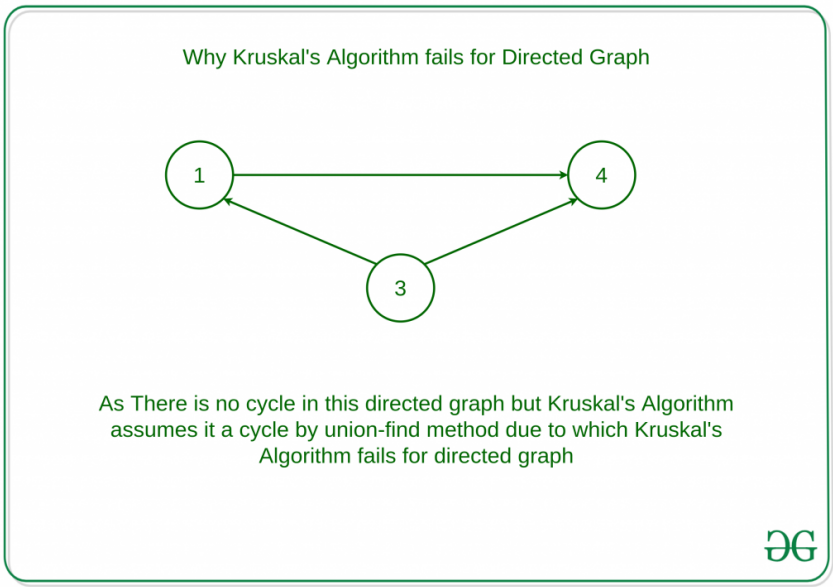
**Why Prim's Algorithm Fails for Directed Graph ?**

Prim's algorithm assumes that all vertices are connected. But in a directed graph, every node is not reachable from every other node. So, Prim's algorithm fails due to this reason.
**For Example:**




As it is visible in the graph, no node is reachable from node 4. Directed graphs fail the requirement that all vertices are connected.

**Why Kruskal's Algorithm fails for directed graph ?**
In Kruskal's algorithm, In each step, it is checked that if the edges form a cycle with the spanning-tree formed so far. But Kruskal's algorithm fails to detect the cycles in a directed graph as there are cases when there is no cycle between the vertices but Kruskal's Algorithm assumes it to cycle and don't take consider some edges due to which Kruskal's Algorithm fails for directed graph.
**For example:**




This graph will be reported to contain a cycle with the Union-Find method, but this graph has no cycle.

The equivalent of minimum spanning tree in directed graphs is, “Minimum Spanning Arborescence”(also known as optimum branching) can be solved by Edmonds’ algorithm with a running time of O(EV). This algorithm is directed analog of the minimum spanning tree problem.

,

# Longest Common Subsequence

programiz.com/dsa/longest-common-subsequence

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If *S1* and *S2* are the two given sequences then, *Z* is the common subsequence of *S1* and *S2* if *Z* is a subsequence of both *S1* and *S2*. Furthermore, *Z* must be a **strictly increasing sequence** of the indices of both *S1* and *S2*.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in *Z*.

If

```
S1 = {B, C, D, A, A, C, D}
```

Then, `{A, D, B}` cannot be a subsequence of *S1* as the order of the elements is not the same (ie. not strictly increasing sequence).

---

Let us understand LCS with an example.

If

```
S1 = {B, C, D, A, A, C, D}
S2 = {A, C, D, B, A, C}
```

Then, common subsequences are `{B, C}`, `{C, D, A, C}`, `{D, A, C}`, `{A, A, C}`, `{A, C}`, `{C, D}`, `...`

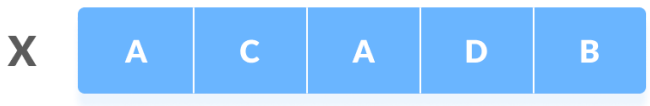
Among these subsequences, `{C, D, A, C}` is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

Before proceeding further, if you do not already know about dynamic programming, please go through dynamic programming.

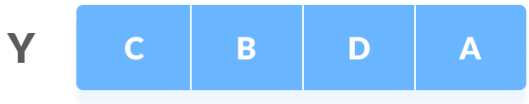
---

## Using Dynamic Programming to find the LCS

Let us take two sequences:



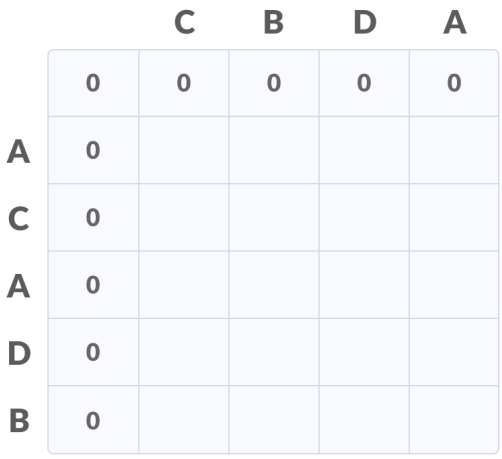

The first sequence




Second Sequence

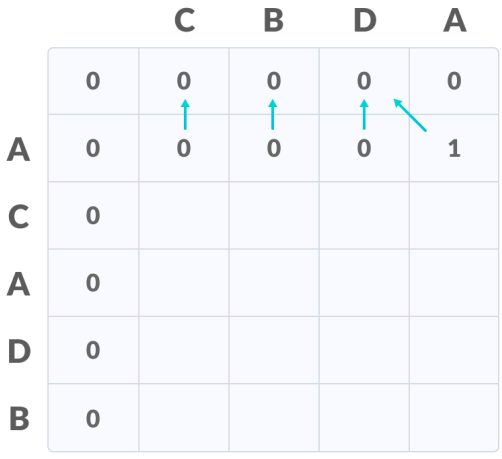
The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension `n+1*m+1` where n and m are the lengths of *X* and *Y* respectively. The first row and the first column are filled with zeros.



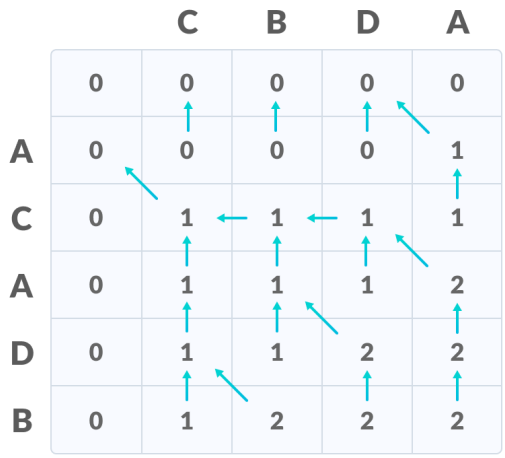

Initialise a table

2. Fill each cell of the table using the following logic.
3. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.



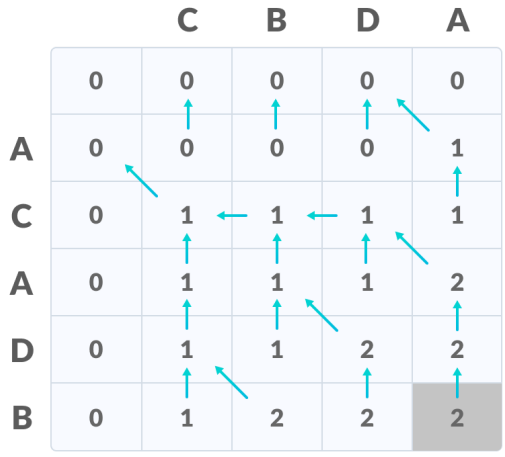

Fill the values

5. **Step 2** is repeated until the table is filled.



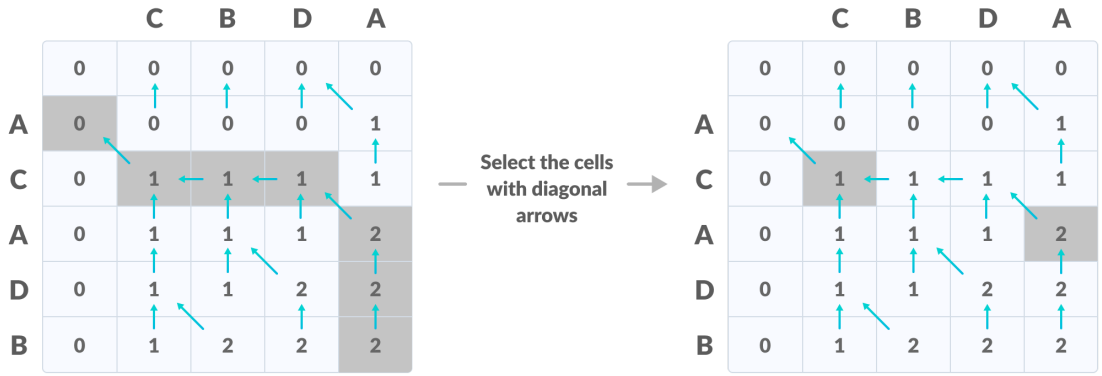

Fill all the values

6. The value in the last row and the last column is the length of the longest common subsequence.




The bottom right corner is the length of the LCS

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.




Create a path according to the arrows

Thus, the longest common subsequence is *CA*.


LCS

---

**How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?**

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of *X* and the elements of *Y* are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)). Whereas, the recursion algorithm has the complexity of $2^{max(m, n)}$.

---

## Longest Common Subsequence Algorithm

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
    If X[i] = Y[j]
        LCS[i][j] = 1 + LCS[i-1, j-1]
        Point an arrow to LCS[i][j]
    Else
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
        Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

---

## Python, Java and C/C++ Examples

```
// The longest common subsequence in C

#include <stdio.h>
#include <string.h>

int i, j, m, n, LCS_table[20][20];
char S1[20] = "ACADB", S2[20] = "CBDA", b[20][20];

void lcsAlgo() {
  m = strlen(S1);
  n = strlen(S2);

  // Filling 0's in the matrix
  for (i = 0; i <= m; i++)
    LCS_table[i][0] = 0;
  for (i = 0; i <= n; i++)
    LCS_table[0][i] = 0;

  // Building the mtrix in bottom-up way
  for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
      if (S1[i - 1] == S2[j - 1]) {
        LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
      } else if (LCS_table[i - 1][j] >= LCS_table[i][j - 1]) {
        LCS_table[i][j] = LCS_table[i - 1][j];
      } else {
        LCS_table[i][j] = LCS_table[i][j - 1];
      }
    }

  int index = LCS_table[m][n];
  char lcsAlgo[index + 1];
  lcsAlgo[index] = '\0';

  int i = m, j = n;
  while (i > 0 && j > 0) {
    if (S1[i - 1] == S2[j - 1]) {
      lcsAlgo[index - 1] = S1[i - 1];
      i--;
      j--;
      index--;
    }

    else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
      i--;
    else
      j--;
  }

  // Printing the sub sequences
  printf("S1 : %s \nS2 : %s \n", S1, S2);
  printf("LCS: %s", lcsAlgo);
}

int main() {
  lcsAlgo();
```

```
  printf("\n");
}
```

## Longest Common Subsequence Applications

1. in compressing genome resequencing data
2. to authenticate users within their mobile phone through in-air signatures

# Floyd-Warshall Algorithm

programiz.com/dsa/floyd-warshall-algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

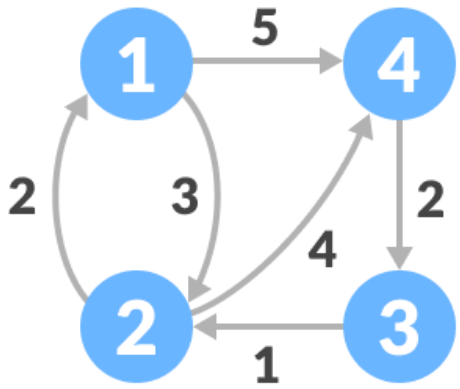
A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming approach to find the shortest paths.

## How Floyd-Warshall Algorithm Works?

Let the given graph be:




Initial graph

Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix $A^0$ of dimension $n*n$ where n is the number of vertices. The row and the column are indexed as *i* and *j* respectively. *i* and *j* are the vertices of the graph.

   Each cell *A[i][j]* is filled with the distance from the $i^{th}$ vertex to the $j^{th}$ vertex. If there is no path from $i^{th}$ vertex to $j^{th}$ vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Fill each cell with the distance between
ith and jth vertex

2. Now, create a matrix $A^1$ using matrix $A^0$. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

   Let *k* be the intermediate vertex in the shortest path from source to destination. In this step, *k* is the first vertex. `A[i][j]` is filled with `(A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j])`.

   That is, if the direct distance from the source to the destination is greater than the path through the vertex *k*, then the cell is filled with `A[i][k] + A[k][j]`.

   In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

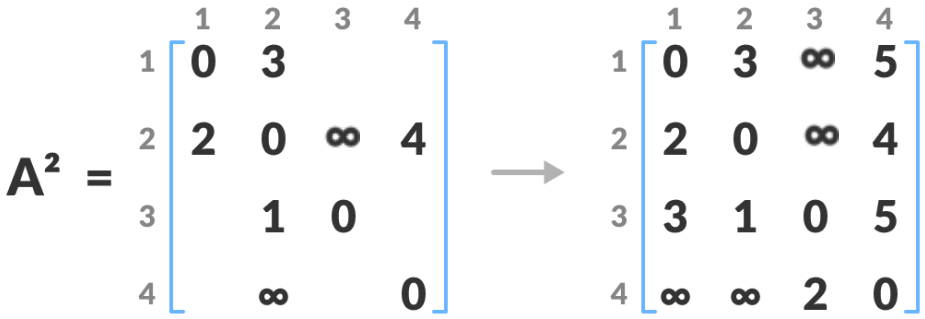
$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex k

   For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

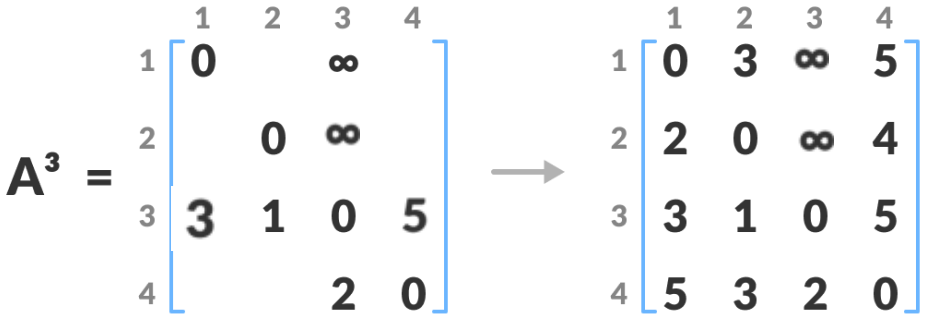
3. Similarly, $A^2$ is created using $A^1$. The elements in the second column and the second row are left as they are.

   In this step, *k* is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

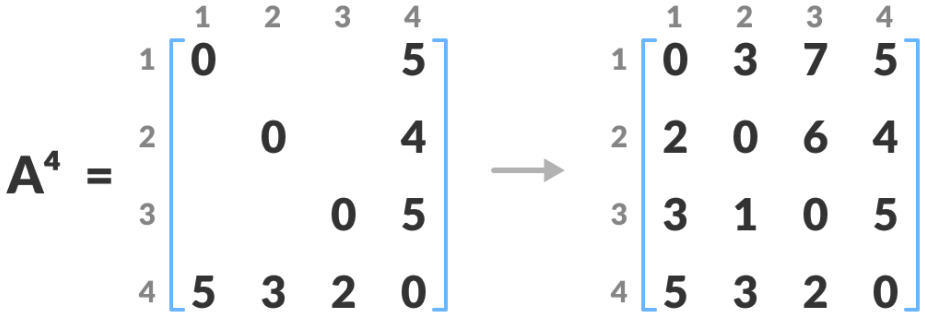



Calculate the distance from the source vertex to destination vertex through this vertex 2

4. Similarly, $A^3$ and $A^4$ is also created.




Calculate the distance from the source vertex to destination vertex through this vertex 3




Calculate the distance from the source vertex to destination vertex through this vertex 4

5. $A^4$ gives the shortest path between each pair of vertices.

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            A^k[i, j] = min (A^k-1[i, j], A^k-1[i, k] + A^k-1[k, j])
return A
```

```
// Floyd-Warshall Algorithm in C

#include <stdio.h>

// defining the number of vertices
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];

  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      if (matrix[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
              {2, 0, INF, 4},
              {INF, 1, 0, INF},
              {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```

## Floyd Warshall Algorithm Complexity

### Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

### Space Complexity

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

## Floyd Warshall Algorithm Applications

- To find the shortest path is a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite