**TREES, HASING $ HASH TABLE AND GRAPHS IN DATA STRUCTURE**

**Introduction to trees in data structure;**

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges”, both directed and undirected.

**The necessity of trees in data structure:**

* Other data structures like arrays, linked-list, stacks, and queues are linear data structures, and all these data structures store data in sequential order. Time complexity increases with increasing data size to perform operations like insertion and deletion on these linear data structures. But it is not acceptable for today's world of computation.
* The non-linear structure of trees enhances the data storing, data accessing, and manipulation processes by employing advanced control methods traversal through it. You will learn about tree traversal in the upcoming section.

**Tree node**: A node is a structure that contains a key or value and pointers in its child node in the tree data structure.

**Tree terminologies**

**Root**

* In a tree data structure, the root is the first node of the tree. The root node is the initial node of the tree in data structures.
* In the tree data structure, there must be only one root node.

Edge

* In a tree in data structures, the connecting link of any two nodes is called the edge of the tree data structure.
* In the tree data structure, N number of nodes connecting with N -1 number of edges.

Parent

In the tree in data structures, the node that is the predecessor of any node is known as a parent node, or a node with a branch from itself to any other successive node is called the parent node

### Child

* The node, a descendant of any node, is known as child nodes in data structures.
* In a tree, any number of parent nodes can have any number of child nodes.
* In a tree, every node except the root node is a child node.

### Siblings

In trees in the data structure, nodes that belong to the same parent are called siblings.

**Leaf**

* Trees in the data structure, the node with no child, is known as a leaf node.
* In trees, leaf nodes are also called external nodes or terminal nodes.

### Internal nodes

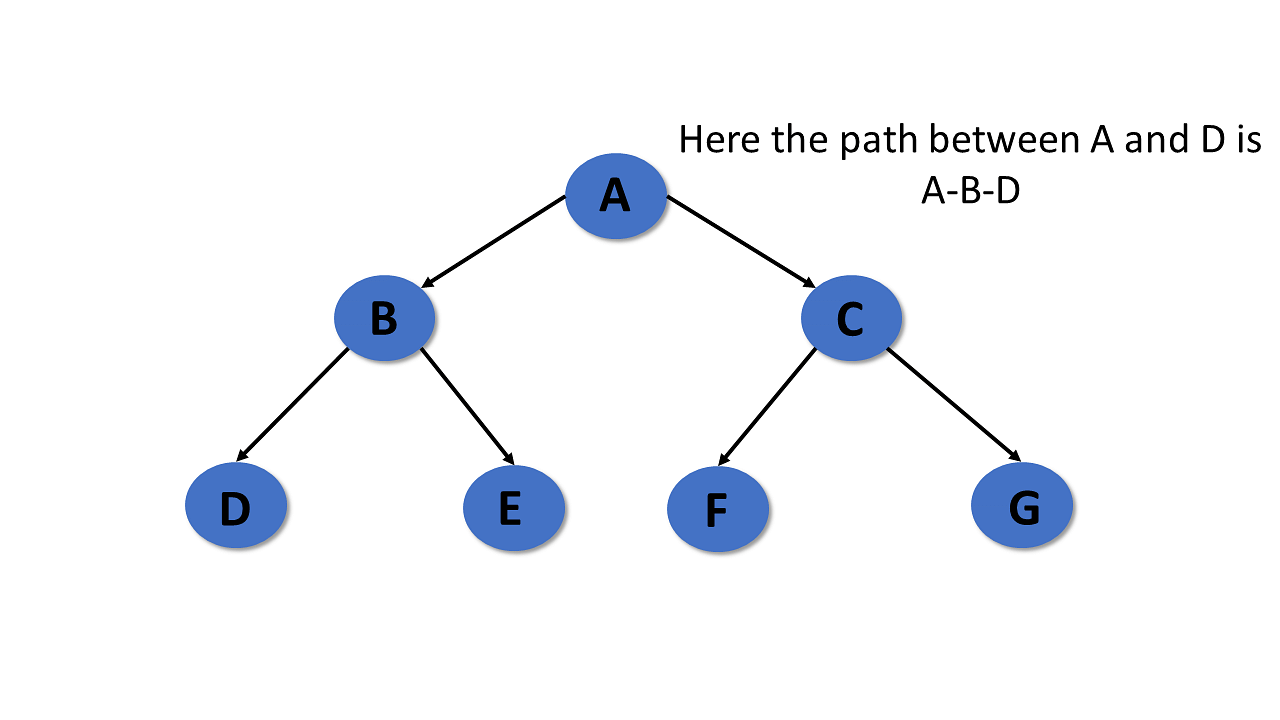
* Trees in the data structure have at least one child node known as internal nodes.
* In trees, nodes other than leaf nodes are internal nodes.
* Sometimes root nodes are also called internal nodes if the tree has more than one node.

### Degree

* In the tree data structure, the total number of children of a node is called the degree of the node.
* The highest degree of the node among all the nodes in a tree is called the Degree of Tree.

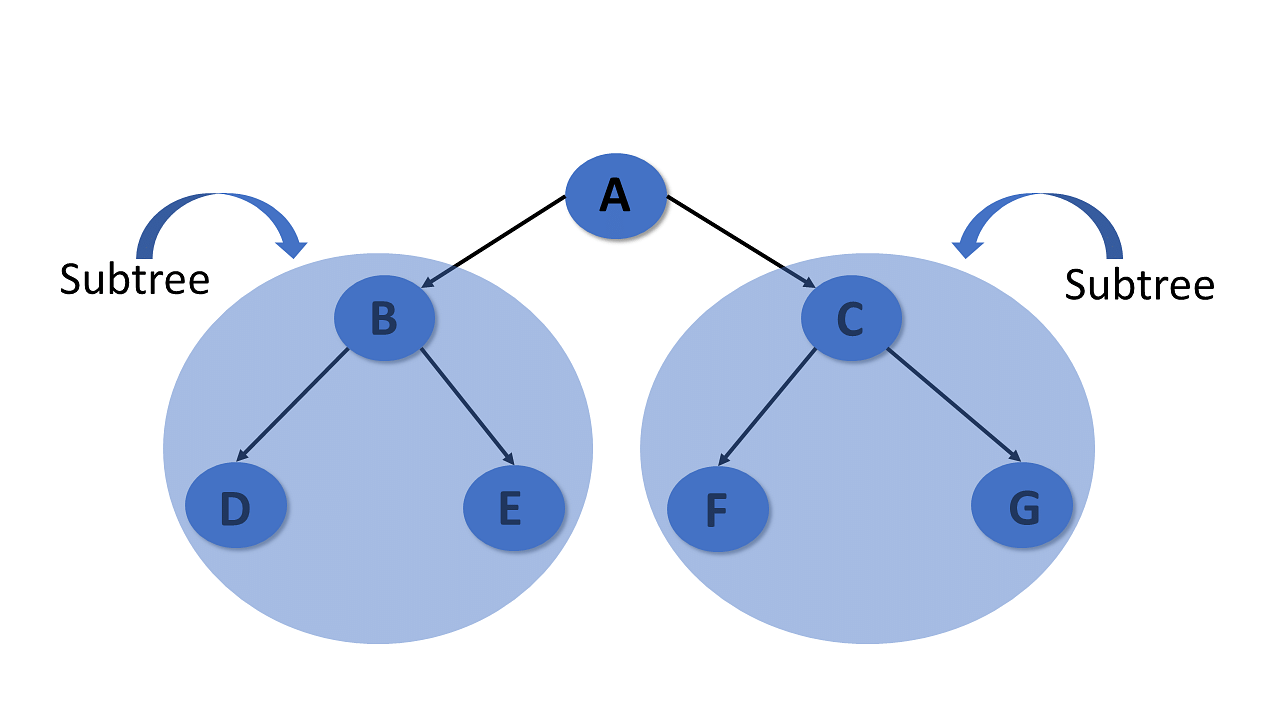
### Path

* In the tree in data structures, the sequence of nodes and edges from one node to another node is called the path between those two nodes.
* The length of a path is the total number of nodes in a path.zx



### Subtree

In the tree in data structures, each child from a node shapes a sub-tree recursively and every child in the tree will form a sub-tree on its parent node.



Now you will look into the types of trees in data structures.

## **Types of Tree in Data Structures**

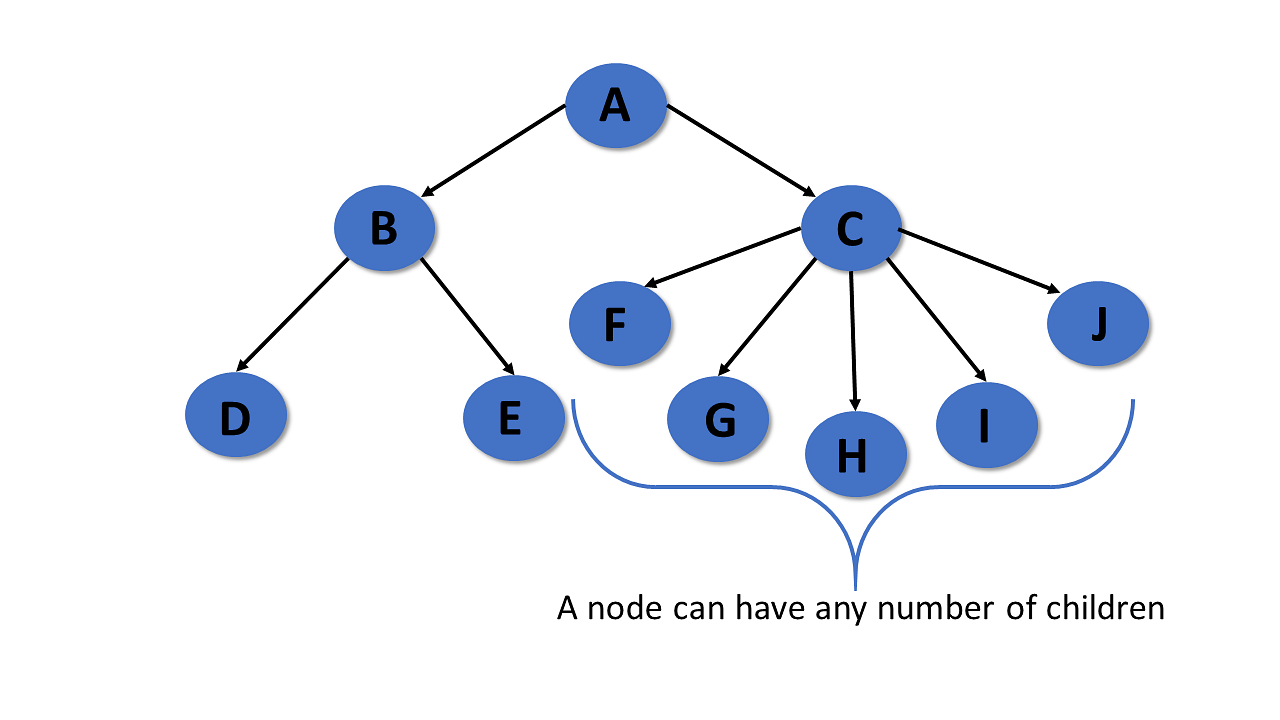
Here are the different kinds of tree in data structures:

### General Tree

The general tree is the type of tree where there are no constraints on the hierarchical structure.

#### **Properties**

* The general tree follows all properties of the tree data structure.
* A node can have any number of nodes.

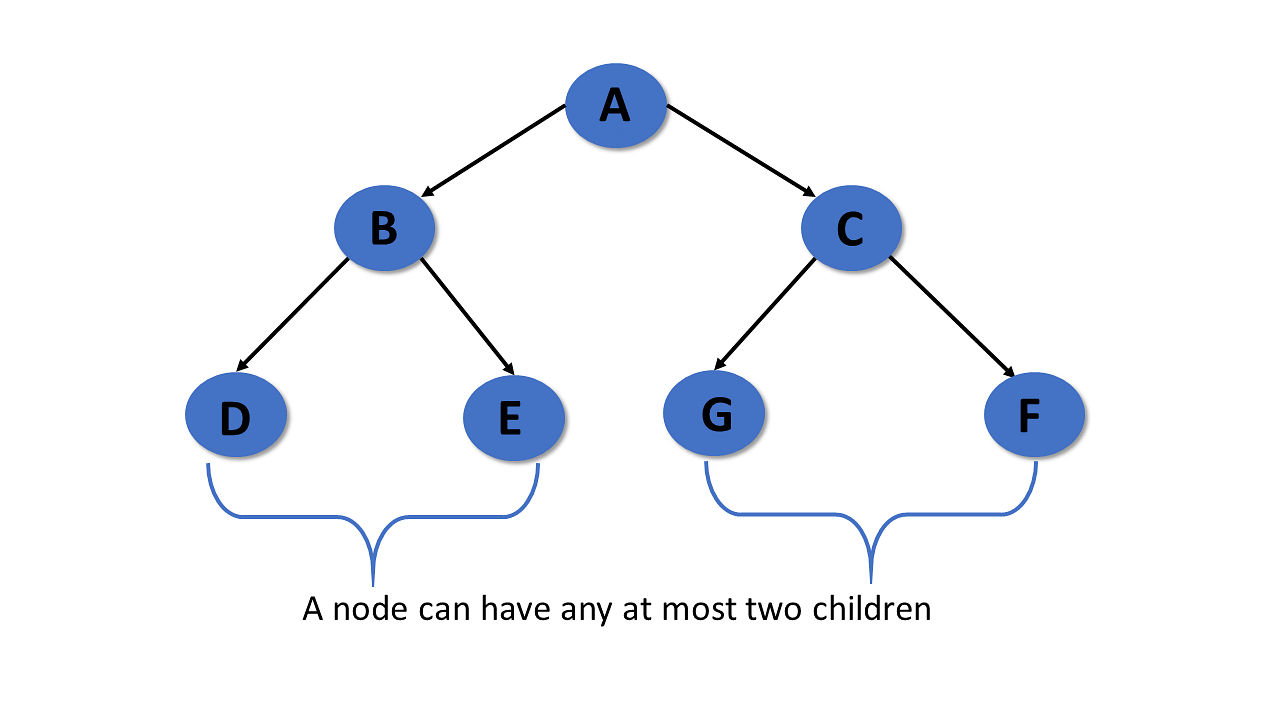


### Binary Tree

A binary tree has the following properties:

#### **Properties**

* Follows all properties of the tree data structure.
* Binary trees can have at most two child nodes.
* These two children are called the left child and the right child.

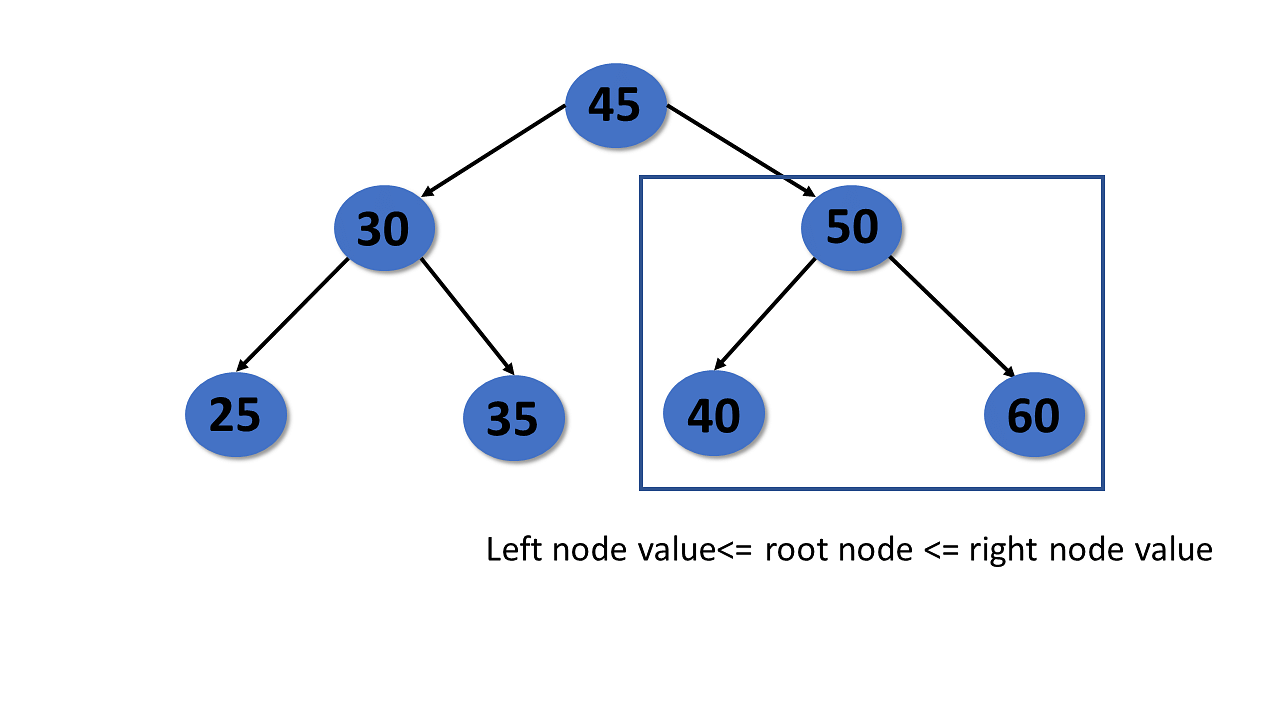


### Binary Search Tree

A binary search tree is a type of tree that is a more constricted extension of a binary tree data structure.

#### **Properties**

* Follows all properties of the tree data structure.
* The binary search tree has a unique property known as the binary search property. This states that the value of a left child node of the tree should be less than or equal to the parent node value of the tree. And the value of the right child node should be greater than or equal to the parent value.

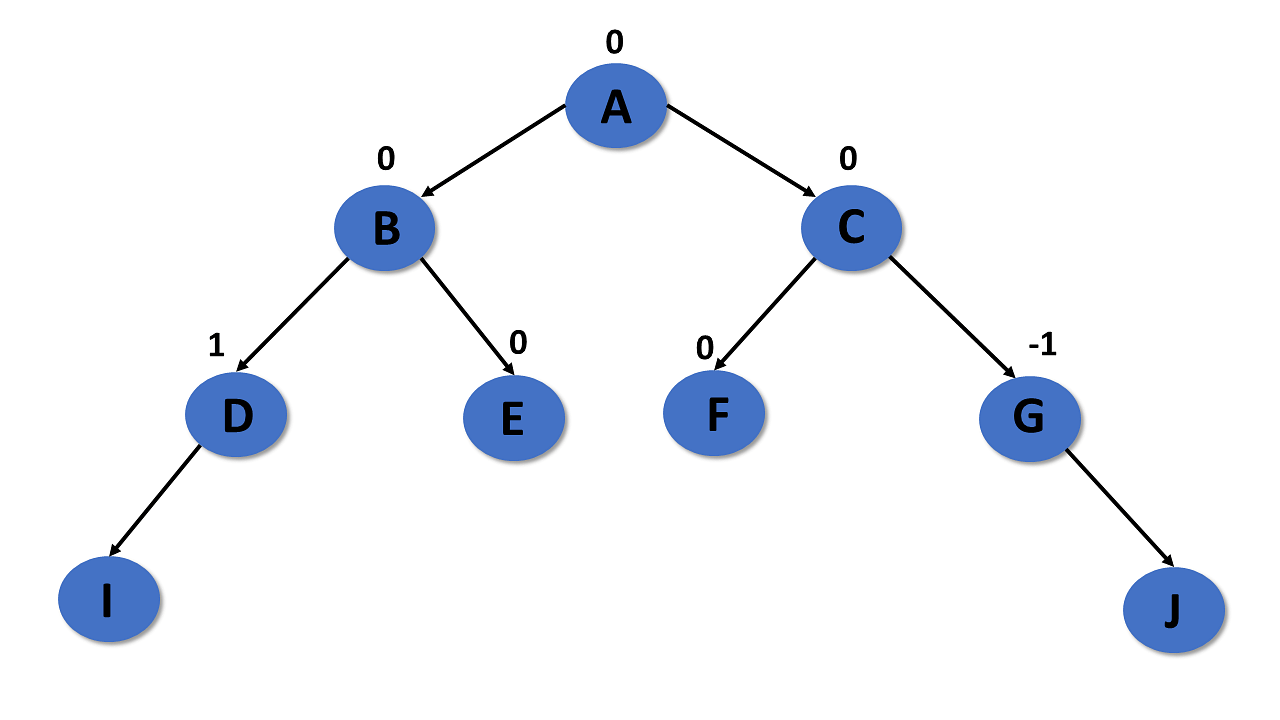


### AVL Tree

An AVL tree is a type of tree that is a self-balancing binary search tree.

#### **Properties**

* Follows all properties of the tree data structure.
* Self-balancing.
* Each node stores a value called a balanced factor, which is the difference in the height of the left sub-tree and right sub-tree.
* All the nodes in the AVL tree must have a balance factor of -1, 0, and 1.



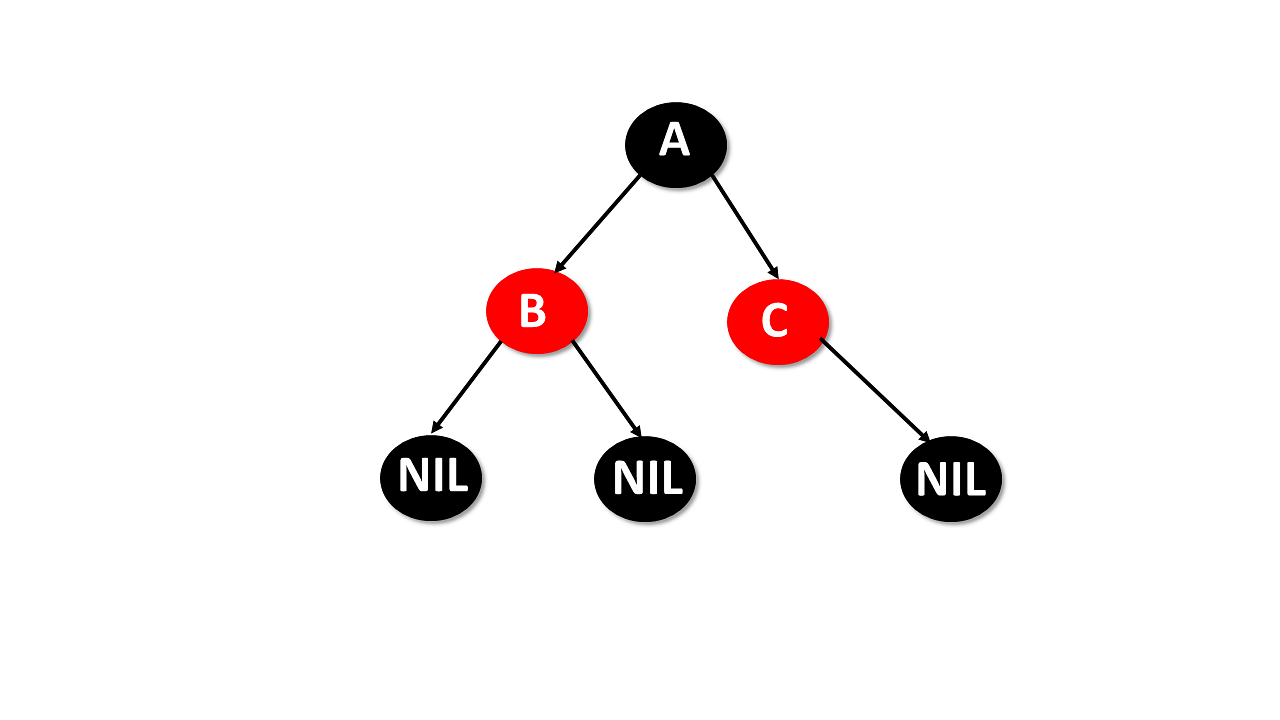
### Red-Black Tree

* A red-black tree is a self-balancing binary search tree, where each node has either the color of red or black.
* The colors of the nodes are used to make sure that the tree remains approximately balanced during insertion and deletion.

#### **Properties**

* Follow all properties of binary tree data structure.
* Self-balancing.
* Each node is either red or black.
* The root and leaves nodes are black.
* If the node is red, then both children are black.
* Every path from a given node to any of its nodes must go through the same number of black nodes.

.



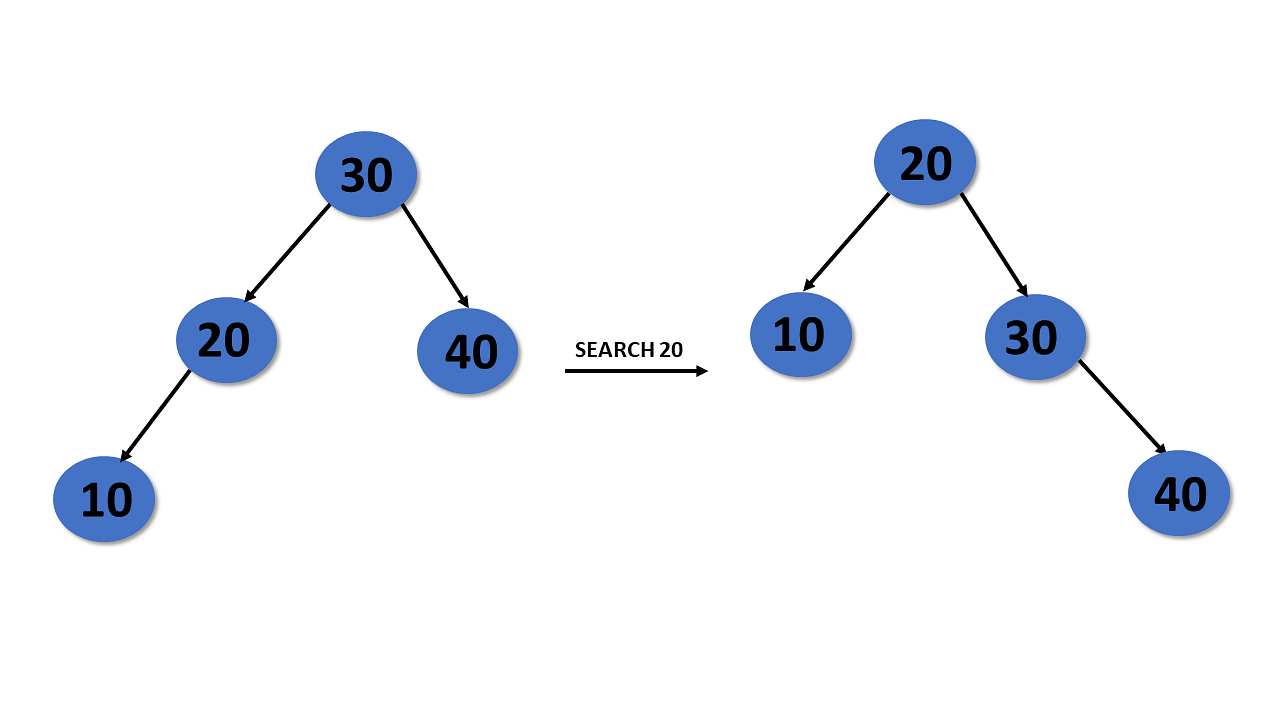
### Splay Tree

A splay tree is a self-balancing binary search tree.

#### **Properties**

* Follows properties of binary tree data structure.
* Self-balancing.
* Recently accessed elements are quicker to access again.

After you perform operations such as insertion and deletion, the splay tree acts, which is called splaying. Here it rearranges the tree so that the particular elements are placed at the root of the tree.

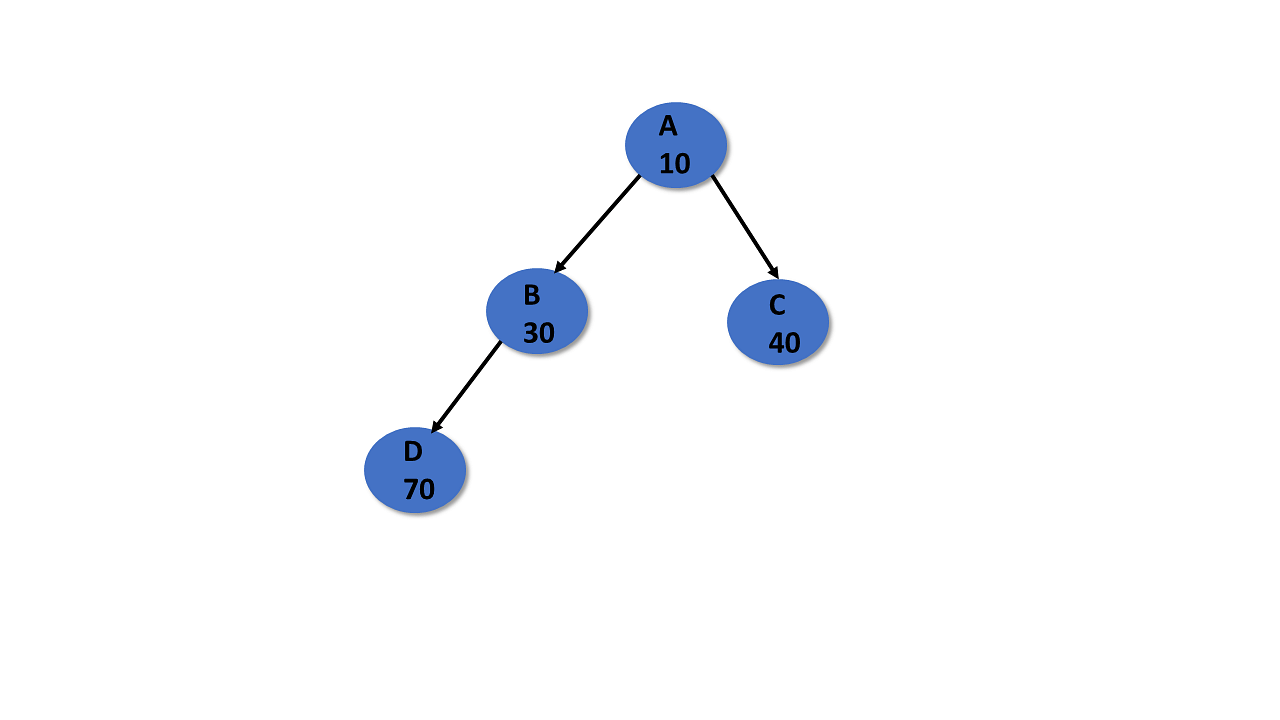


### Treap Tree

The Treap tree is made up of a tree, and the heap is a binary search tree.

#### **Properties**

* Each node has two values: a key and a priority.
* Follows a binary search tree property.
* Priority of the treap tree follows the heap property.



## **Tree Traversal**

Traversal of the tree in data structures is a process of visiting each node and prints their value. There are three ways to traverse tree data structure.

* Pre-order Traversal
* In-Order Traversal
* Post-order Traversal

### In-Order Traversal

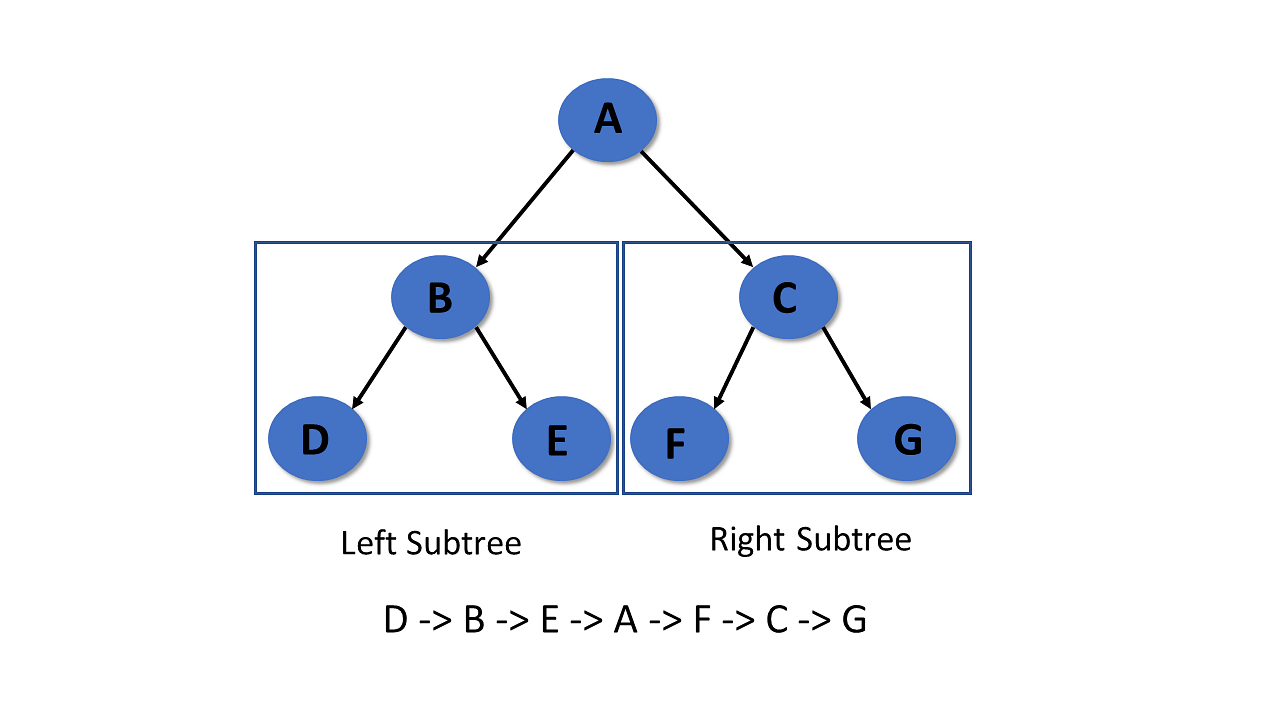
In the in-order traversal, the left subtree is visited first, then the root, and later the right subtree.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree



### Pre-Order Traversal

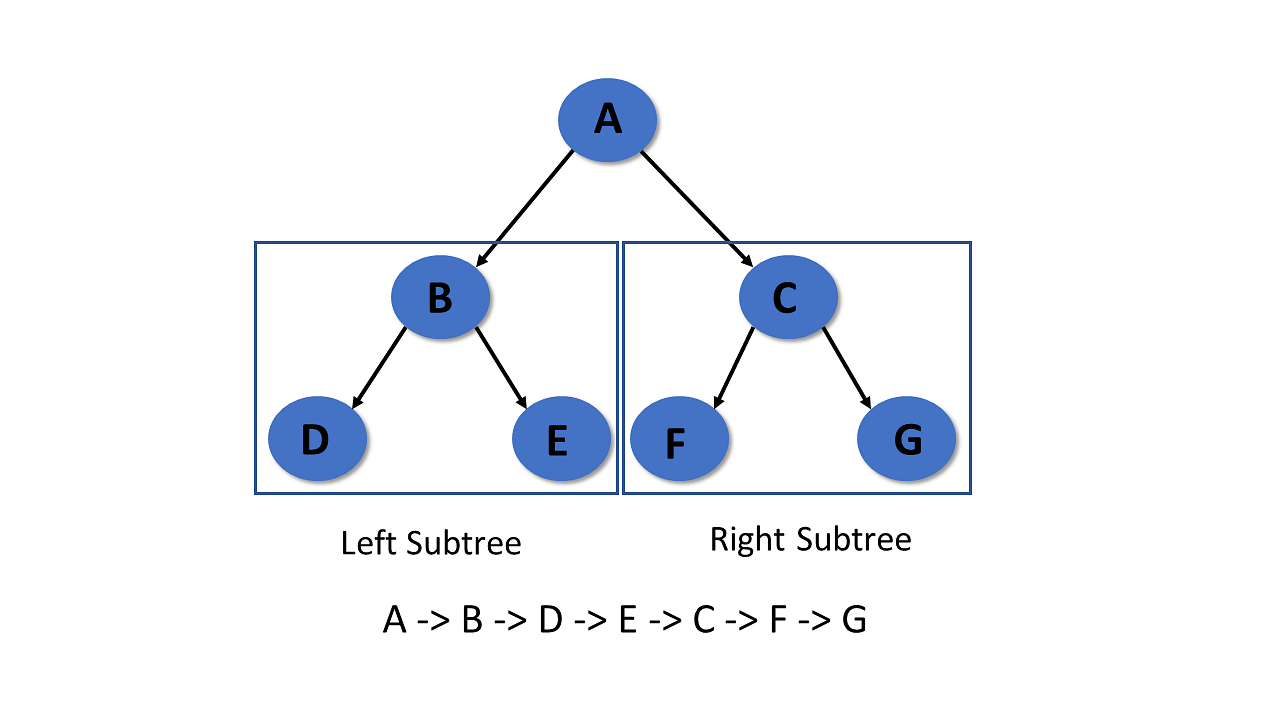
In pre-order traversal, it visits the root node first, then the left subtree, and lastly right subtree.

Algorithm:

Step 1- Visit root node

Step 2- Recursively traverse the left subtree

Step 3- Recursively traverse right subtree



### Post-Order Traversal

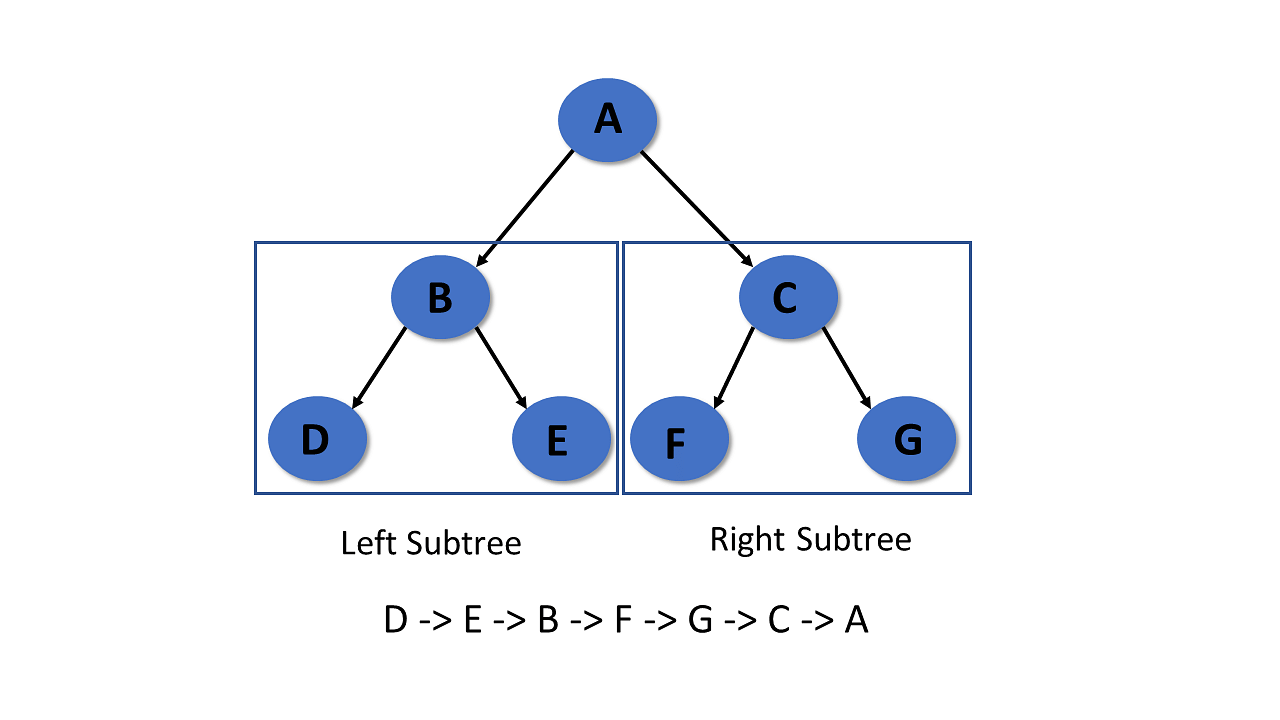
It visits the left subtree first in post-order traversal, then the right subtree, and finally the root node.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree



## **Application of Tree in Data Structures**

* Binary Search Tree (BST) is used to check whether elements present or not.
* Heap is a type of tree that is used to heap sort.
* Tries are the modified version of the tree used in modem routing to information of the router.
* The widespread database uses B-tree.
* Compilers use syntax trees to check every syntax of a program.

With this, you’ve come to the end of this tutorial about the tree in data structures.

*Reference: simplelean.com*

**HASHING AND HASH TABLE**

**Defination:**

***Hashing*** refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.

## **Need for Hash data structure**

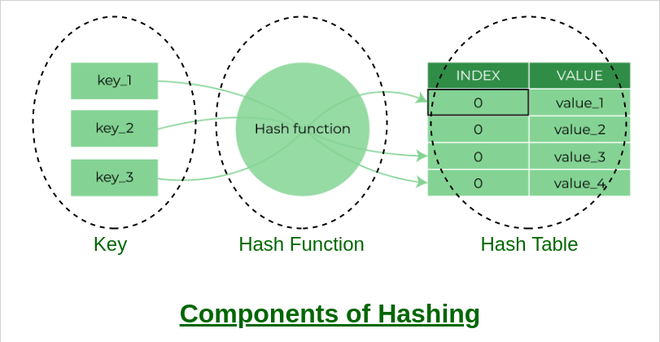
Every day, the data on the internet is increasing multifold and it is always a struggle to store this data efficiently. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A very common data structure that is used for such a purpose is the Array data structure.

Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word “**efficiency**“. Though storing in Array takes O(1) time, searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient. So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in O(1) time. This is how Hashing data structure came into play. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well.

## Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:**The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the**hash index**.
3. **Hash Table:**Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.



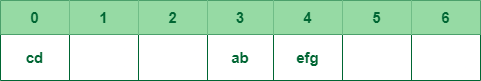
***Components of Hashing***

## How does Hashing work?

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

Our main objective here is to search or update the values stored in the table quickly in O(1) time and we are not concerned about the ordering of strings in the table. So the given set of strings can act as a key and the string itself will act as the value of the string but how to store the value corresponding to the key?

* **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
* **Step 2:** So, let’s assign
  + “a” = 1,
  + “b”=2, .. etc, to all alphabetical characters.
* **Step 3:**Therefore, the numerical value by summation of all characters of the string:
* *“ab” = 1 + 2 = 3,*
* *“cd” = 3 + 4 = 7 ,*
* *“efg” = 5 + 6 + 7 = 18*
* **Step 4:**Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.
* **Step 5:** So we will then store
  + “ab” in 3 mod 7 = 3,
  + “cd” in 7 mod 7 = 0, and
  + “efg” in 18 mod 7 = 4.



The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table.

## What is a Hash function?

The [hash function](https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/) creates a mapping between key and value, this is done through the use of mathematical formulas known as hash functions. The result of the hash function is referred to as a hash value or hash. The hash value is a representation of the original string of characters but usually smaller than the original.

For example: Consider an array as a Map where the key is the index and the value is the value at that index. So for an array A if we have index **i** which will be treated as the key then we can find the value by simply looking at the value at A[i].  
simply looking up A[i].

### Types of Hash functions:

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing **[different hash functions](https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/):**

1. [Division Method.](https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/#:~:text=1.%20Division%20Method,the%20remainder%20obtained.)
2. [Mid Square Method.](https://www.geeksforgeeks.org/introduction-to-hashing-data-structure-and-algorithm-tutorials/The%20mid%20square%20method%20is%20a%20very%20good%20hashing%20method.%20It%20involves%20two%20steps%20to%20compute%20the%20hash%20value-%20%20Square%20the%20value%20of%20the%20key%20k%20i.e.%20k2%20Extract%20the%20middle%20r%20digits%20as%20the%20hash%20value.)
3. [Folding Method.](https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/#:~:text=3.%20Digit%20Folding,carry%20if%20any.)
4. [Multiplication Method](https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/)

### Properties of a Good hash function

A hash function that maps every item into its own unique slot is known as a perfect hash function. We can construct a perfect hash function if we know the items and the collection will never change but the problem is that there is no systematic way to construct a perfect hash function given an arbitrary collection of items. Fortunately, we will still gain performance efficiency even if the hash function isn’t perfect. We can achieve a perfect hash function by increasing the size of the hash table so that every possible value can be accommodated. As a result, each item will have a unique slot. Although this approach is feasible for a small number of items, it is not practical when the number of possibilities is large.

So, We can construct our hash function to do the same but the things that we must be careful about while constructing our own hash function.

A good hash function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position is equally likely for each.
3. Should minimize collisions.
4. Should have a low load factor(number of items in the table divided by the size of the table).

### Complexity of calculating hash value using the hash function

Time complexity: O(n)

Space complexity: O(1)

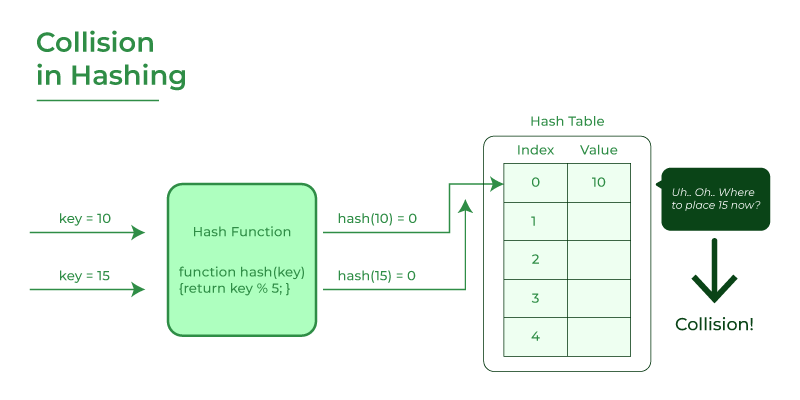
## Problem with Hashing

If we consider the above example, the hash function we used is the sum of the letters, but if we examined the hash function closely then the problem can be easily visualized that for different strings same hash value is begin generated by the hash function.

For example: {“ab”, “ba”} both have the same hash value, and string {“cd”,”be”} also generate the same hash value, etc. This is known as **collision** and it creates problem in searching, insertion, deletion, and updating of value.

## What is collision?

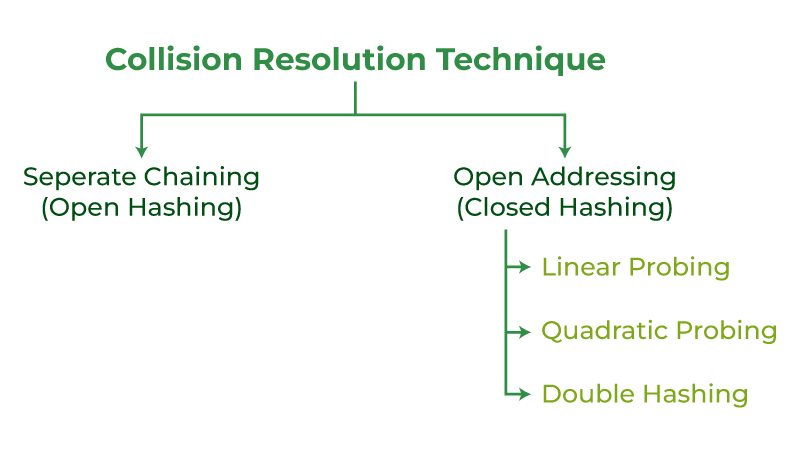
The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/20220706102035/Collision-in-Hashing.png)

## How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining:
2. Open Addressing:

\

### 1) [Separate Chaining](https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/)

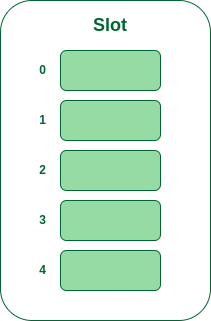
The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

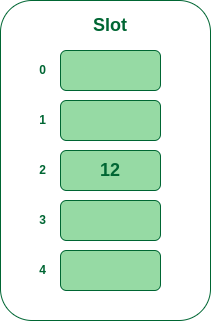
Hash function = key % 5,   
Elements = 12, 15, 22, 25 and 37.

Let’s see step by step approach to how to solve the above problem:

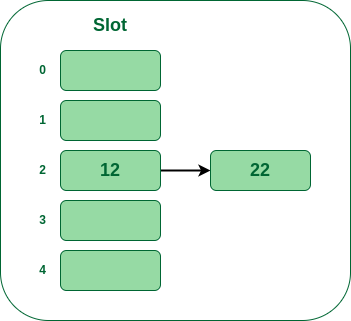
* **Step 1:**First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



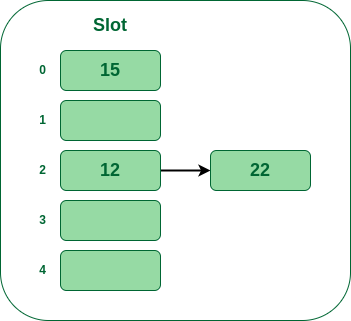
* **Step 2:** Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.



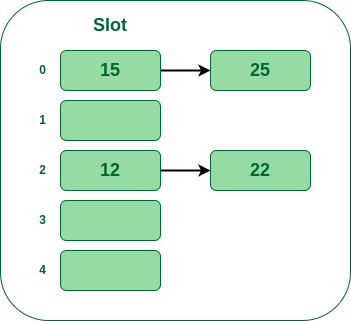
* **Step 3:** Now the next key is 22. It will map to bucket number 2 because 22%5=2. But bucket 2 is already occupied by key 12.



* **Step 4:** The next key is 15. It will map to slot number 0 because 15%5=0.



* **Step 5:** Now the next key is 25. Its bucket number will be 25%5=0. But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket



Hence In this way, the separate chaining method is used as the collision resolution technique.

### 2) [Open Addressing](https://www.geeksforgeeks.org/hashing-set-3-open-addressing/?ref=lbp)

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

### 2.a) Linear Probing

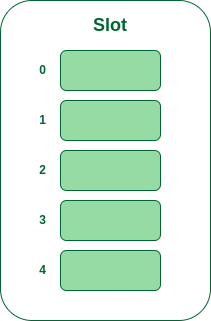
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

**Algorithm:**

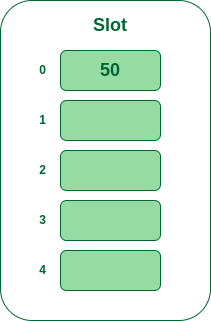
1. *Calculate the hash key. i.e.****key = data % size***
2. *Check, if****hashTable[key]****is empty*
   * *store the value directly by****hashTable[key] = data***
3. *If the hash index already has some value then*
   * *check for next index using****key = (key+1) % size***
4. *Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.*
5. *Do the above process till we find the space.*

**Example:** Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

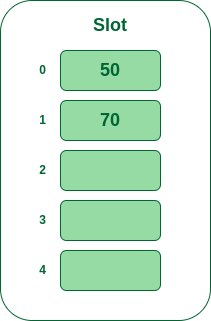
* **Step 1:**First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



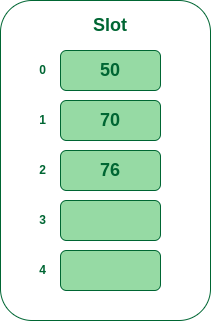
* **Step 2:** Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.



* **Step 3:**The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.

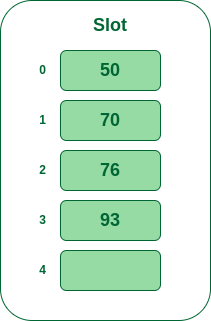


* **Step 4:**The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.



* **Step 5:**The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3

.



### 2.b) Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

An example sequence using quadratic probing is:

***H****+ 12,****H****+ 22,****H****+ 32,****H****+ 42………………….****H****+ k2*

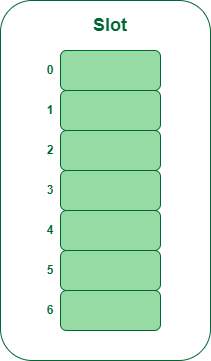
This method is also known as the mid-square method because in this method we look for i2‘th probe (slot) in i’th iteration and the value of i = 0, 1, . . . n – 1. We always start from the original hash location. If only the location is occupied then we check the other slots.

Let hash(x) be the slot index computed using the hash function and n be the size of the hash table.

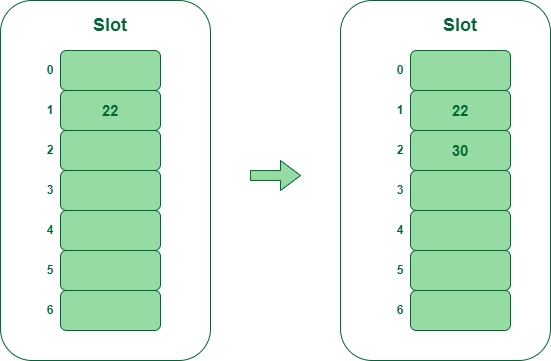
*If the slot hash(x) % n is full, then we try (hash(x) + 12) % n.  
If (hash(x) + 12) % n is also full, then we try (hash(x) + 22) % n.  
If (hash(x) + 22) % n is also full, then we try (hash(x) + 32) % n.  
This process will be repeated for all the values of****i****until an empty slot is found*

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i2 . Insert = 22, 30, and 50

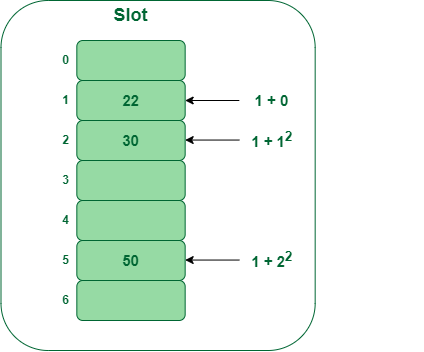
* **Step 1:** Create a table of size 7.



* **Step 2** – Insert 22 and 30
  + Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
  + Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



* **Step 3:** Inserting 50
  + Hash(50) = 50 % 7 = 1
  + In our hash table slot 1 is already occupied. So, we will search for slot 1+12, i.e. 1+1 = 2,
  + Again slot 2 is found occupied, so we will search for cell 1+22, i.e.1+4 = 5,
  + Now, cell 5 is not occupied so we will place 50 in slot 5.



### 2.c) Double Hashing

Double hashing is a collision resolving technique in [Open Addressed](https://www.geeksforgeeks.org/hashing-set-3-open-addressing/) Hash tables. Double hashing make use of two hash function,

* The first hash function is **h1(k)** which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key.
* But in case the location is occupied (collision) we will use secondary hash-function **h2(k)** in combination with the first hash-function **h1(k)** to find the new location on the hash table.

This combination of hash functions is of the form

**h(k, i) = (h1(k) + i \* h2(k)) % n**

where

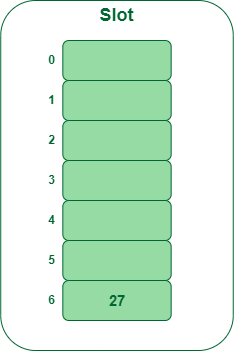
* i is a non-negative integer that indicates a collision number,
* k = element/key which is being hashed
* n = hash table size.

**Complexity of the Double hashing algorithm:**

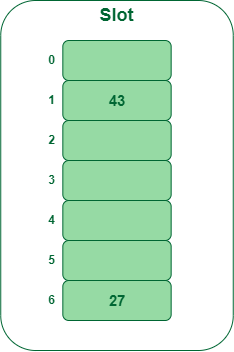
Time complexity: O(n)

**Example:**Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is**h1​(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

* **Step 1:**Insert 27
  + 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.

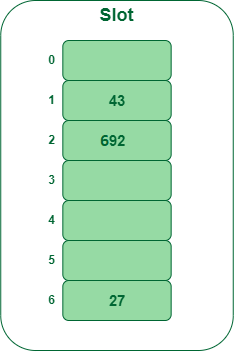


* **Step 2:** Insert 43
  + 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



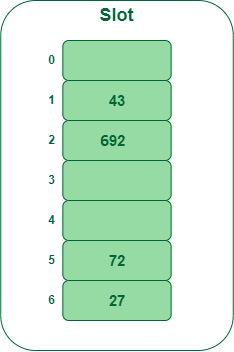
* **Step 3:** Insert 692
  + 692 % 7 = 6, but location 6 is already being occupied and this is a collision
  + So we need to resolve this collision using double hashing.

hnew = [h1(692) + i \* (h2(692)] % 7  
= [6 + 1 \* (1 + 692 % 5)] % 7  
= 9 % 7  
= 2  
  
Now, as 2 is an empty slot,   
so we can insert 692 into 2nd slot.



* **Step 4:** Insert 72
  + 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
  + So we need to resolve this collision using double hashing.

hnew = [h1(72) + i \* (h2(72)] % 7  
= [2 + 1 \* (1 + 72 % 5)] % 7  
= 5 % 7  
= 5,   
  
Now, as 5 is an empty slot,   
so we can insert 72 into 5th slot.



## What is meant by Load Factor in Hashing?

The [load factor](https://www.geeksforgeeks.org/load-factor-and-rehashing) of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hash function or want to add more elements to the existing hash table.

It helps us in determining the efficiency of the hash function i.e. it tells whether the hash function which we are using is distributing the keys uniformly or not in the hash table.

Load Factor = Total elements in hash table/ Size of hash table

## What is Rehashing?

As the name suggests, [rehashing](https://www.geeksforgeeks.org/load-factor-and-rehashing/?ref=lbp#:~:text=Rehashing%3A,and%20low%20complexity.) means hashing again. Basically, when the load factor increases to more than its predefined value (the default value of the load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double-sized array to maintain a low load factor and low complexity.

## Applications of Hash Data structure

* Hash is used in databases for indexing.
* Hash is used in disk-based data structures.
* In some programming languages like Python, JavaScript hash is used to implement objects.

## Real-Time Applications of Hash Data structure

* Hash is used for cache mapping for fast access to the data.
* Hash can be used for password verification.
* Hash is used in cryptography as a message digest.
* Rabin-Karp algorithm for pattern matching in a string.
* Calculating the number of different substrings of a string.

## Advantages of Hash Data structure

* Hash provides better synchronization than other data structures.
* Hash tables are more efficient than search trees or other data structures
* Hash provides constant time for searching, insertion, and deletion operations on average.

## Disadvantages of Hash Data structure

* Hash is inefficient when there are many collisions.
* Hash collisions are practically not avoided for a large set of possible keys.
* Hash does not allow null values.

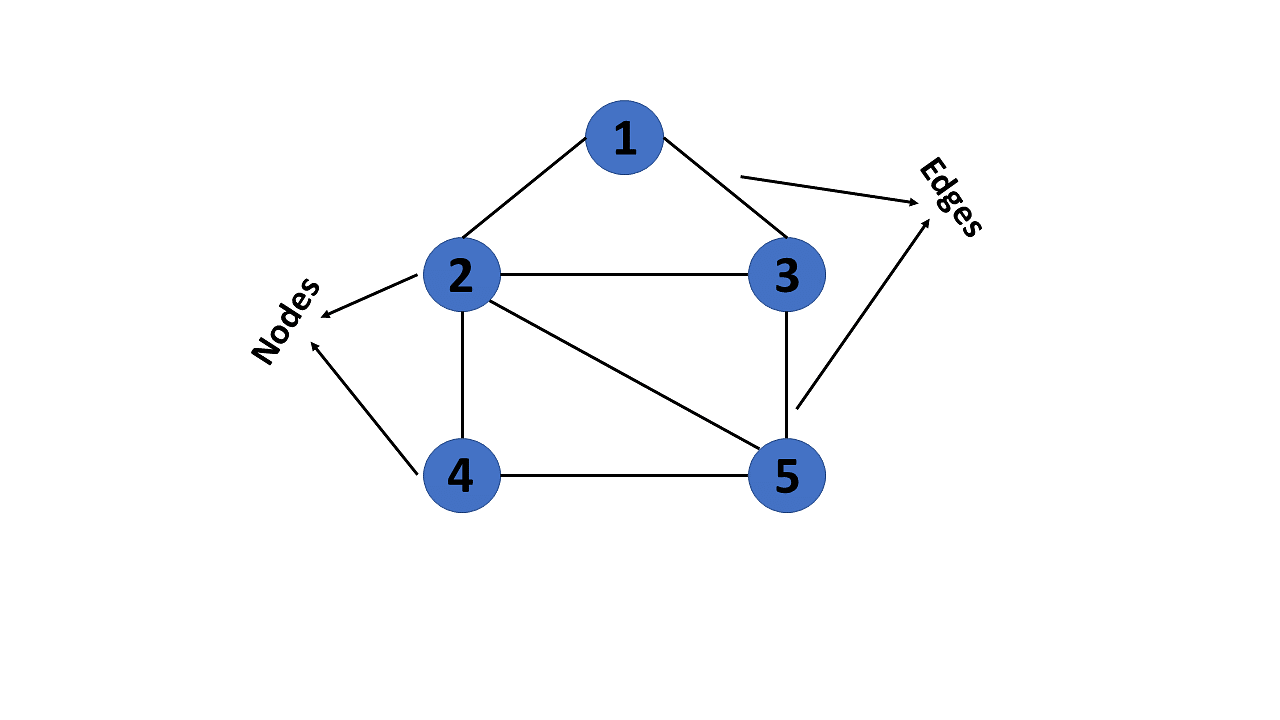
## Conclusion

From the above discussion, we conclude that the goal of hashing is to resolve the challenge of finding an item quickly in a collection. For example, if we have a list of millions of English words and we wish to find a particular term then we would use hashing to locate and find it more efficiently. It would be inefficient to check each item on the millions of lists until we find a match. Hashing reduces search time by restricting the search to a smaller set of words at the beginning.

**GRAPHS IN DATA STRUCTURE**

## **What Are Graphs in Data Structure?**

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



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

Now that you’ve learned about the definition of graphs in data structures, you will learn about their various types.

## **Types of Graphs in Data Structures**

There are different types of graphs in data structures, each of which is detailed below.

### 1. Finite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



### 2. Infinite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.



### 3. Trivial Graph

A graph G= (V, E) is trivial if it contains only a single vertex and no edges.



### 4. Simple Graph

If each pair of nodes or vertices in a graph G=(V, E) has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



### 5. Multi Graph

If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



### 6. Null Graph

It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph.



### 7. Complete Graph

If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.



### 8. Pseudo Graph

If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudograph.



### 9. Regular Graph

If a graph G= (V, E) is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a regular graph.



### 10. Weighted Graph

A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



### 11. Directed Graph

A directed graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction.



### 12. Undirected Graph

An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.



### 13. Connected Graph

If there is a path between one vertex of a graph data structure and any other vertex, the graph is connected.



### 14. Disconnected Graph

When there is no edge linking the vertices, you refer to the null graph as a disconnected graph.



### 15. Cyclic Graph

If a graph contains at least one graph cycle, it is considered to be cyclic.



### 16. Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.



### 17. Directed Acyclic Graph

It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



### 18. Subgraph

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



After you learn about the many types of graphs in graphs in data structures, you will move on to graph terminologies.

## **Terminologies of Graphs in Data Structures**

Following are the basic terminologies of graphs in data structures:

* An edge is one of the two primary units used to form graphs. Each edge has two ends, which are vertices to which it is attached.
* If two vertices are endpoints of the same edge, they are adjacent.
* A vertex's outgoing edges are directed edges that point to the origin.
* A vertex's incoming edges are directed edges that point to the vertex's destination.
* The total number of edges occurring to a vertex in a graph is its degree.
* The out-degree of a vertex in a directed graph is the total number of outgoing edges, whereas the in-degree is the total number of incoming edges.
* A vertex with an in-degree of zero is referred to as a source vertex, while one with an out-degree of zero is known as sink vertex.
* An isolated vertex is a zero-degree vertex that is not an edge's endpoint.
* A path is a set of alternating vertices and edges, with each vertex connected by an edge.
* The path that starts and finishes at the same vertex is known as a cycle.
* A path with unique vertices is called a simple path.
* For each pair of vertices x, y, a graph is strongly connected if it contains a directed path from x to y and a directed path from y to x.
* A directed graph is weakly connected if all of its directed edges are replaced with undirected edges, resulting in a connected graph. A weakly linked graph's vertices have at least one out-degree or in-degree.
* A tree is a connected forest. The primary form of the tree is called a rooted tree, which is a free tree.
* A spanning subgraph that is also a tree is known as a [spanning tree.](https://www.simplilearn.com/tutorials/data-structure-tutorial/spanning-tree-in-data-structure)
* A connected component is the unconnected graph's most connected subgraph.
* A bridge, which is an edge of removal, would sever the graph.
* Forest is a graph without a cycle.

Following that, you will look at the graph representation in this data structures tutorial.

## **Representation of Graphs in Data Structures**

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

The most frequent graph representations are the two that follow:

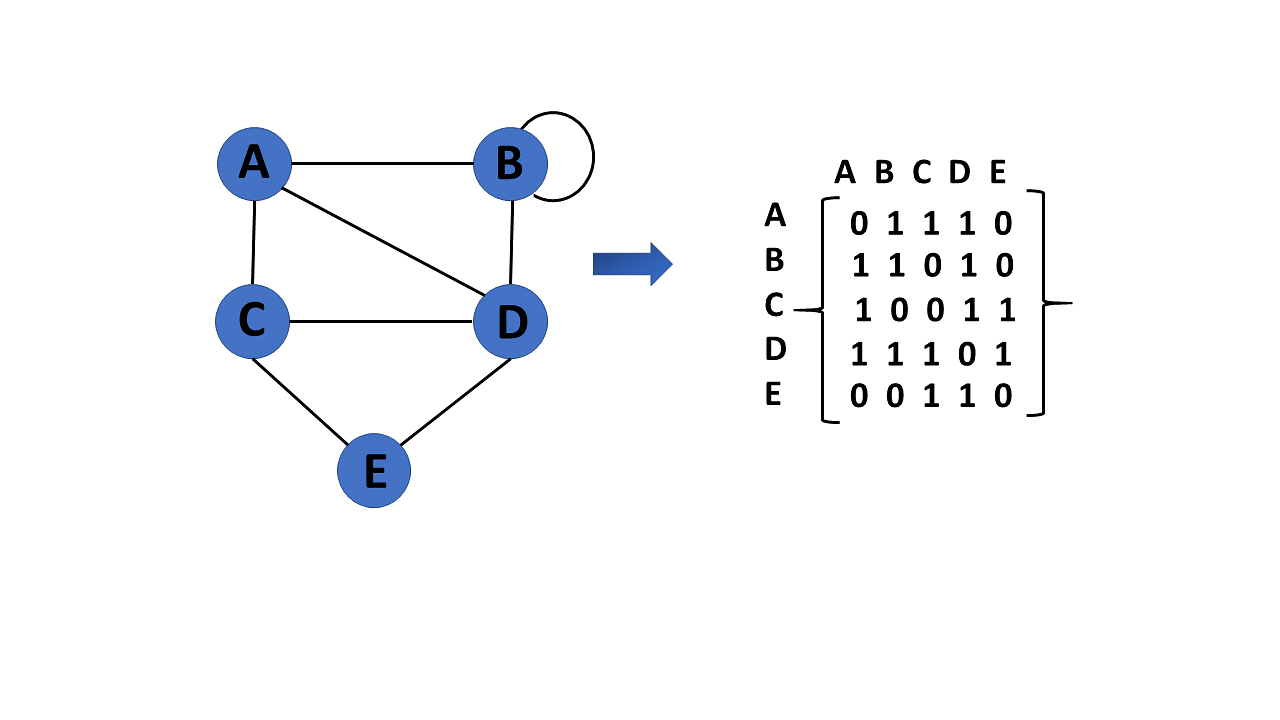
* Adjacency matrix
* Adjacency list

You’ll look at these two representations of graphs in data structures in more detail:

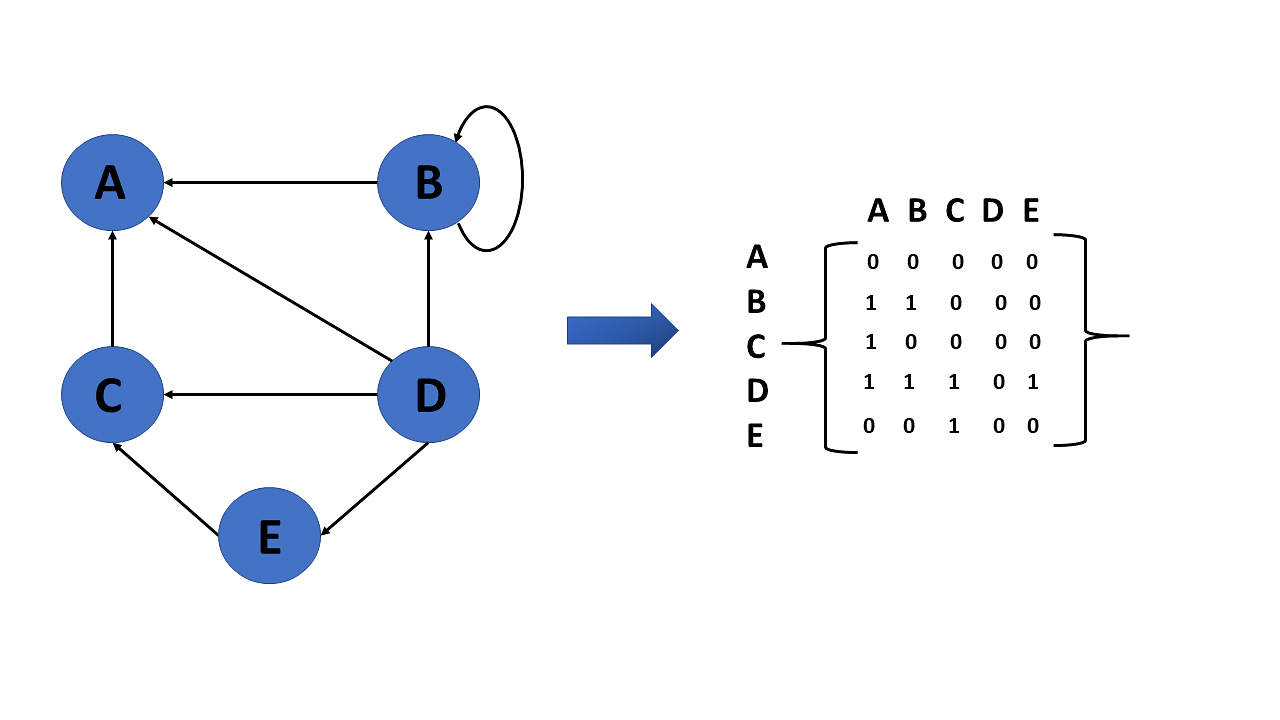
### Adjacency Matrix

* A sequential representation is an adjacency matrix.
* It's used to show which nodes are next to one another. I.e., is there any connection between nodes in a graph?
* You create an MXM matrix G for this representation. If an edge exists between vertex a and vertex b, the corresponding element of G, gi,j = 1, otherwise gi,j = 0.
* If there is a weighted graph, you can record the edge's weight instead of 1s and 0s.

#### **Undirected Graph Representation**

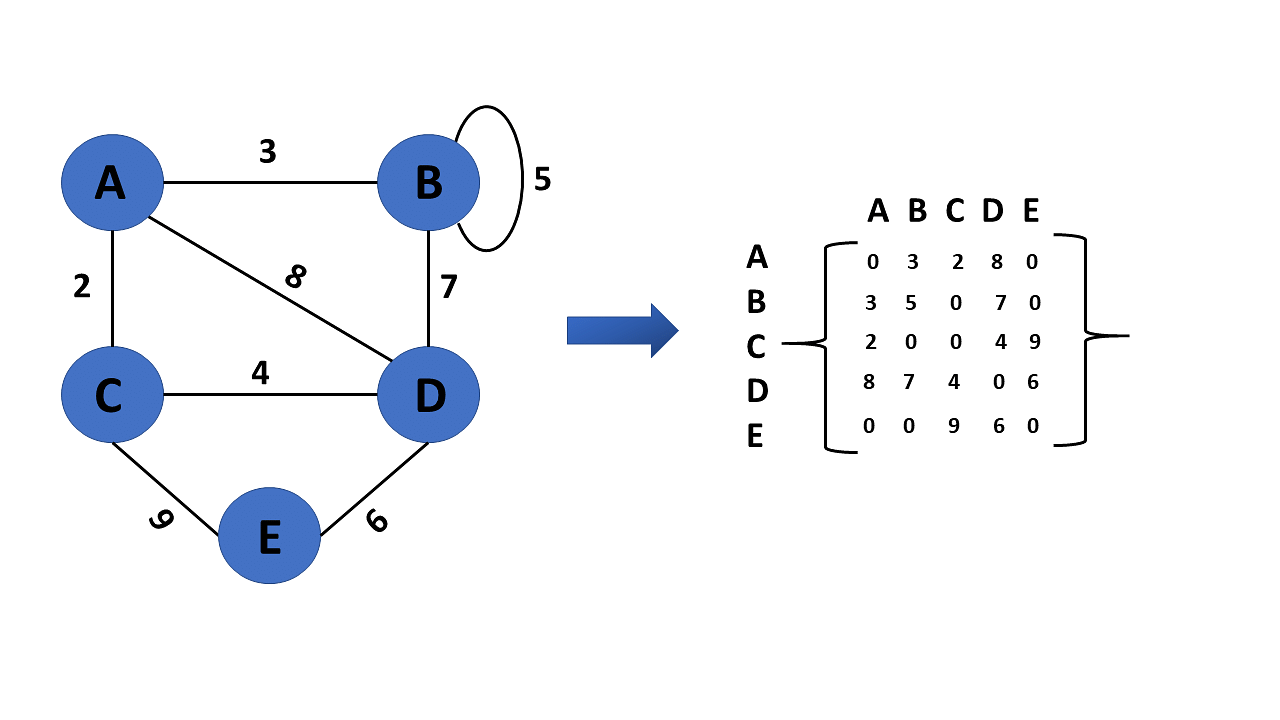


#### **Directed Graph Representation**



#### **Weighted Undirected Graph Representation**

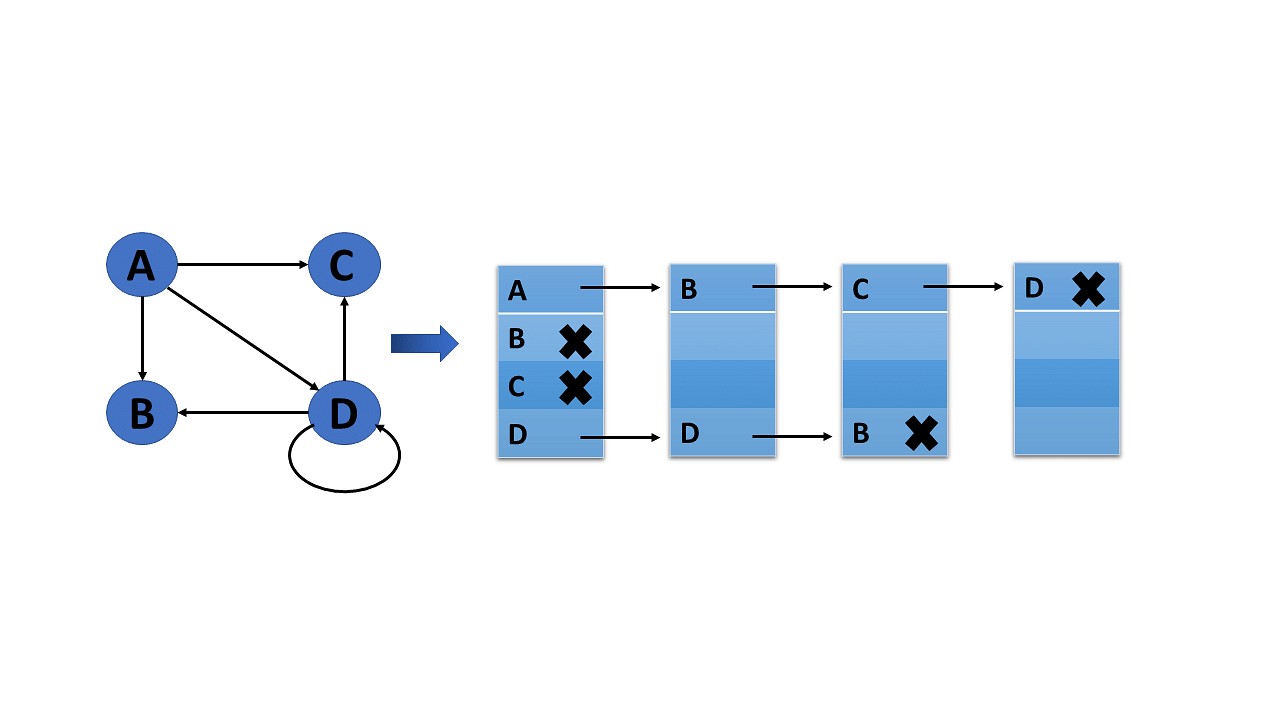
Weight or cost is indicated at the graph's edge, a weighted graph representing these values in the matrix.



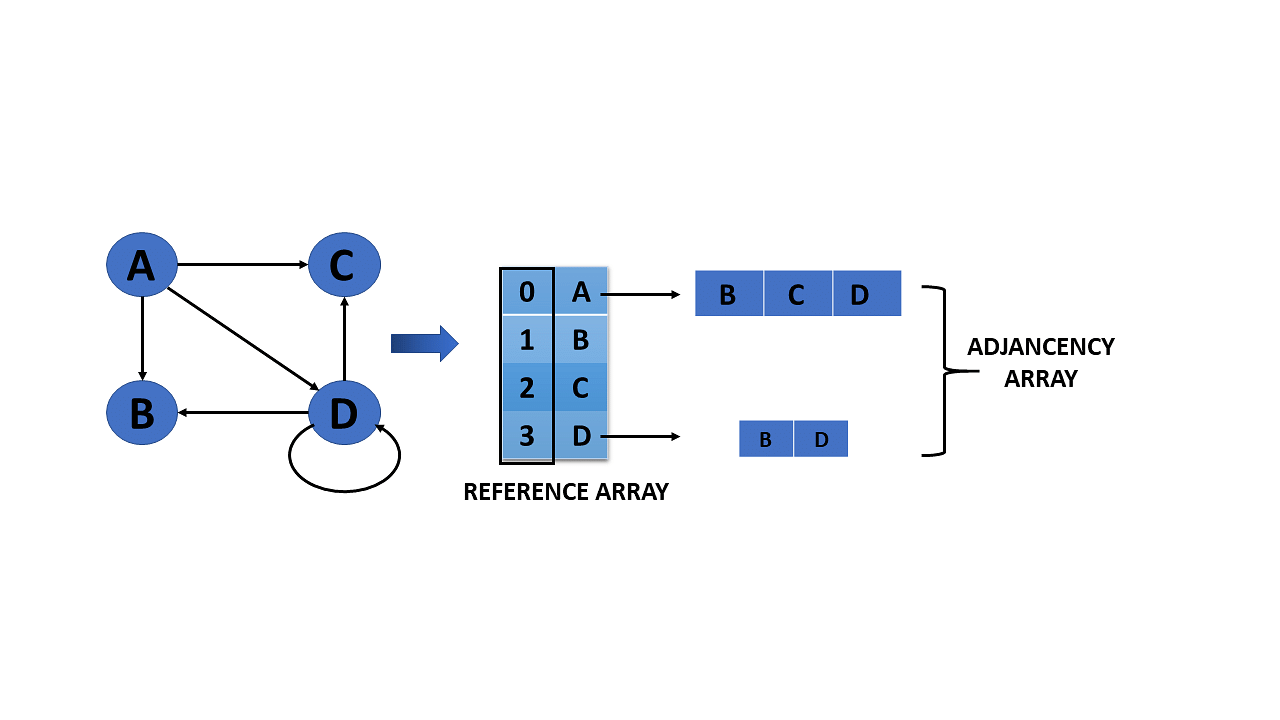
### Adjacency List

* A linked representation is an adjacency list.
* You keep a list of neighbors for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighboring vertices.
* You have an arra of vertices indexed by the vertex number, and the corresponding [array](https://www.simplilearn.com/tutorials/data-structure-tutorial/arrays-in-data-structure) member for each vertex x points to a [singly linked list](https://www.simplilearn.com/tutorials/data-structure-tutorial/singly-linked-list) of x's neighbors.

#### **Weighted Undirected Graph Representation Using Linked-List**



#### **Weighted Undirected Graph Representation Using an Array**



You will now see which all operations are conducted in graphs data structure after understanding the representation of graphs in the data structure.

**Operations on Graphs in Data Structures**

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

* Creating graphs
* Insert vertex
* Delete vertex
* Insert edge
* Delete edge

You will go over each operation in detail one by one:

### Creating Graphs

There are two techniques to make a graph:

#### **1. Adjacency Matrix**

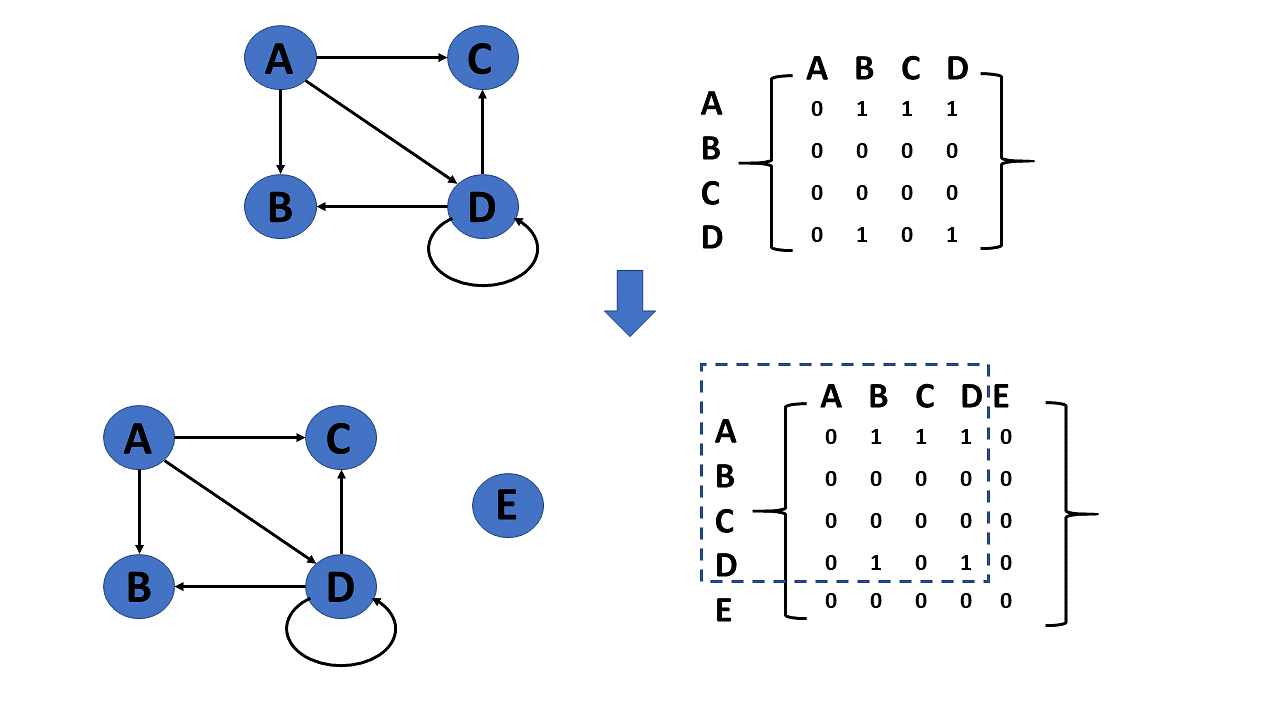
The adjacency matrix of a simple labeled graph, also known as the connection matrix, is a matrix with rows and columns labeled by graph vertices and a 1 or 0 in position depending on whether they are adjacent or not.

#### **2. Adjacency List**

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

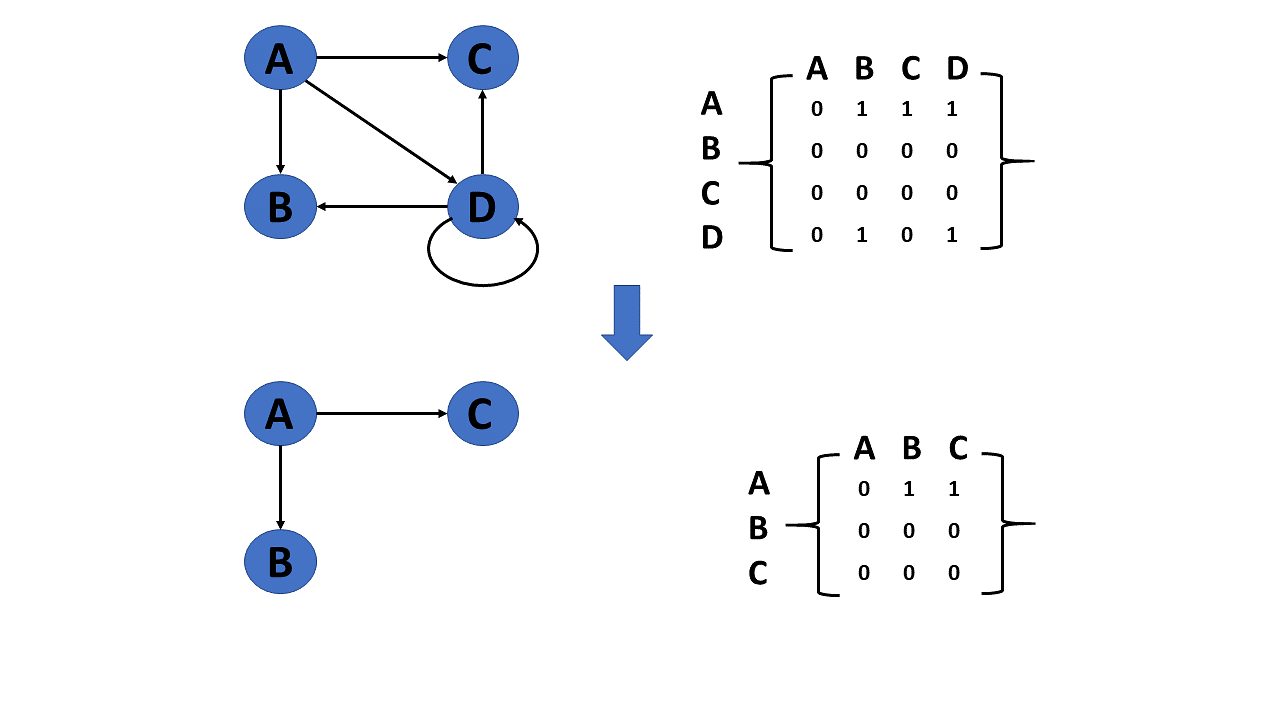
### Insert Vertex

When you add a vertex that after introducing one or more vertices or nodes, the graph's size grows by one, increasing the matrix's size by one at the row and column levels.



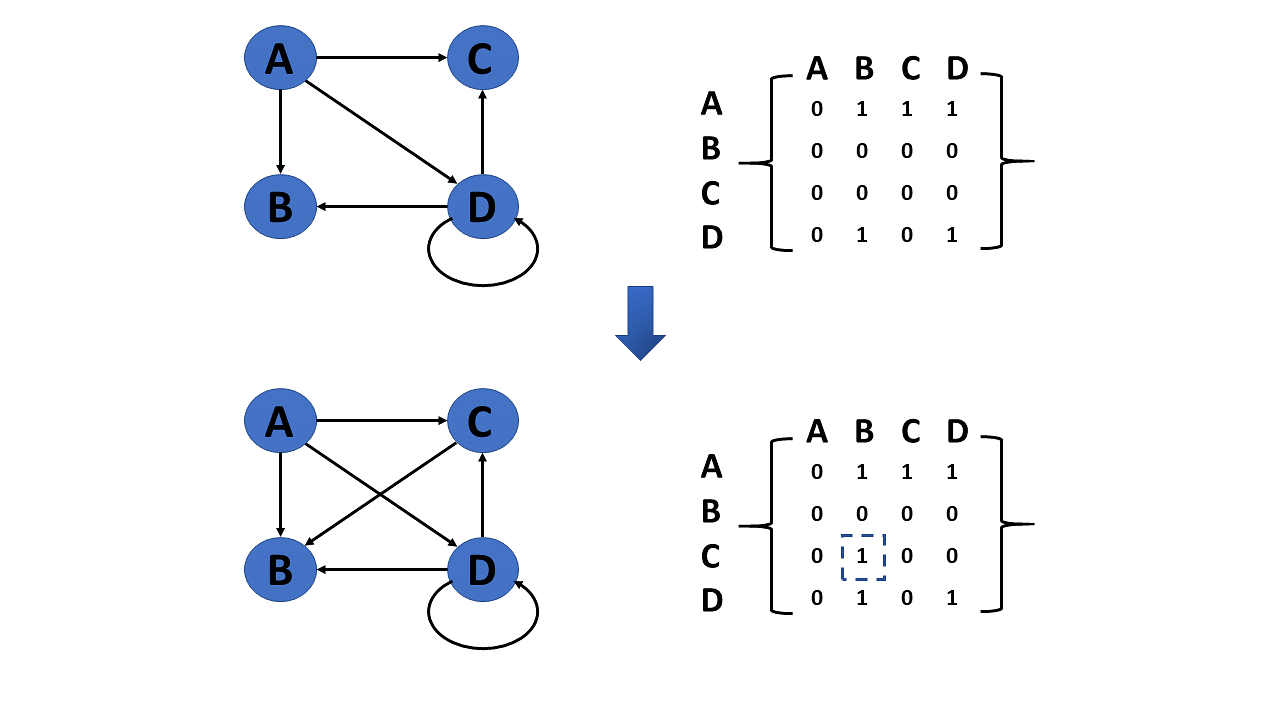
### Delete Vertex

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



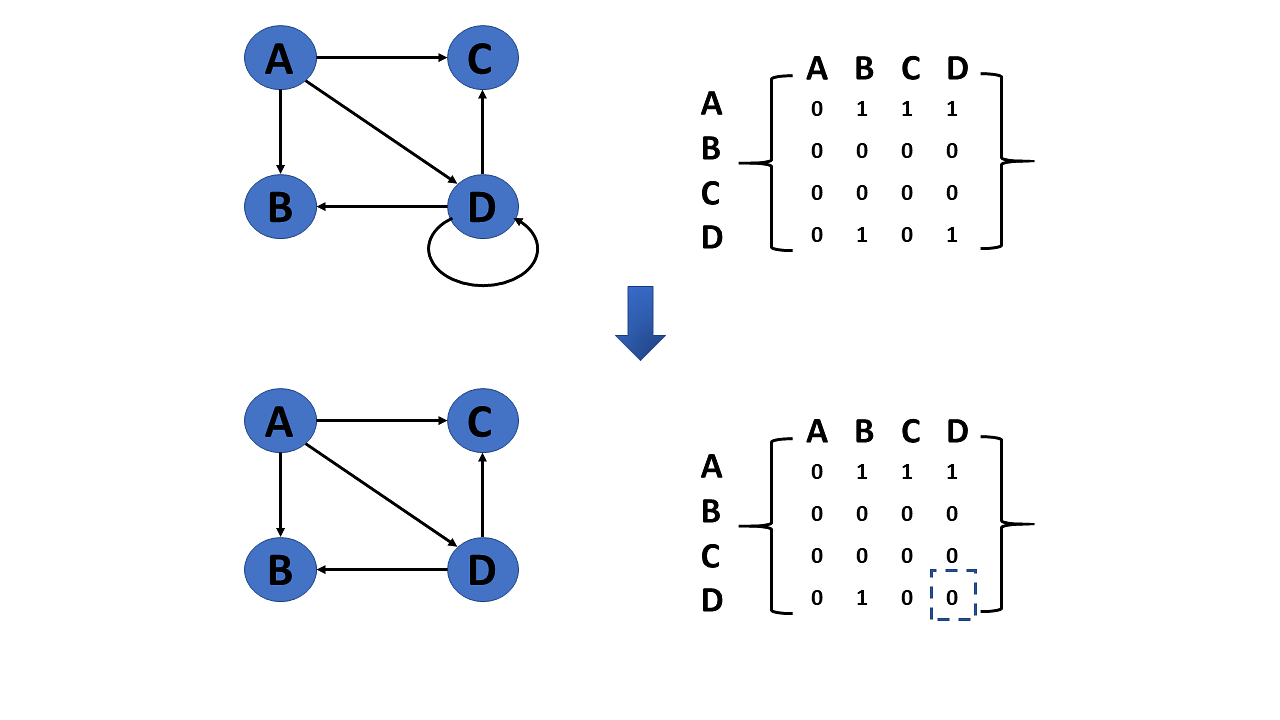
### Insert Edge

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



### Delete Edge

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



The types of graph traversal algorithms will be discussed next in the graphs in this data structures tutorial.

## **Graph Traversal Algorithm**

The process of visiting or updating each vertex in a graph is known as graph traversal. The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a subset of tree traversal.

There are two techniques to implement a graph traversal algorithm:

* Breadth-first search
* Depth-first search

### Breadth-First Search or BFS

[***BFS***](https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm) is a search technique for finding a node in a graph data structure that meets a set of criteria.

* It begins at the root of the graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.
* To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally you require a [queue.](https://www.simplilearn.com/tutorials/data-structure-tutorial/queue-in-data-structure)

Algorithm of breadth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in your graph, say v1, from which you want to traverse the graph.

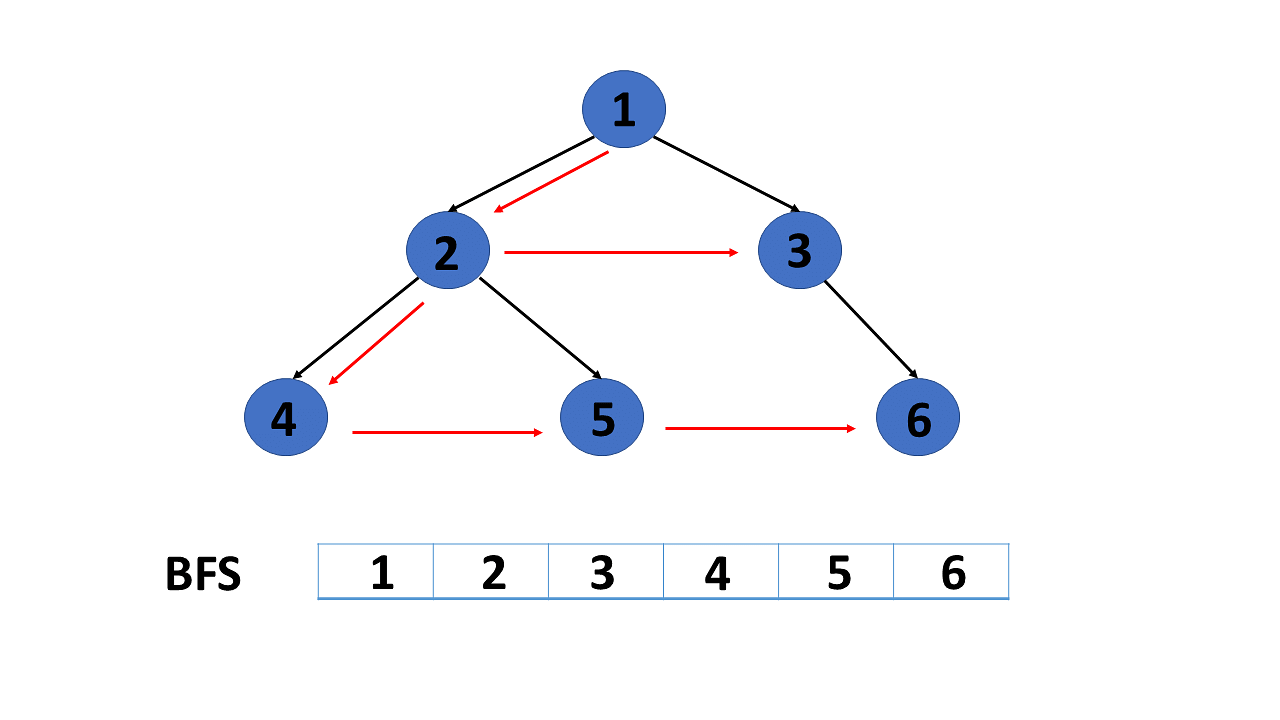
Step 3: Examine any two data structures for traversing the graph.

* Visited array (size of the graph)
* Queue data structure

Step 4: Starting from the vertex, you will add to the visited array, and afterward, you will v1's adjacent vertices to the queue data structure.

Step 5: Now, using the FIFO concept, you must remove the element from the queue, put it into the visited array, and then return to the queue to add the adjacent vertices of the removed element.

Step 6: Repeat step 5 until the queue is not empty and no vertex is left to be visited.



### Depth-First Search or DFS

[DFS](https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm) is a search technique for finding a node in a graph data structure that meets a set of criteria.

* The depth-first search (DFS) algorithm traverses or explores data structures such as trees and graphs. The DFS algorithm begins at the root node and examines each branch as far as feasible before backtracking.
* To maintain track of the child nodes that have been encountered but not yet inspected, more memory, [generally a stack](https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures), is required.

Algorithm of depth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in our graph, say v1, from which you want to begin traversing the graph.

Step 3: Examine any two data structures for traversing the graph.

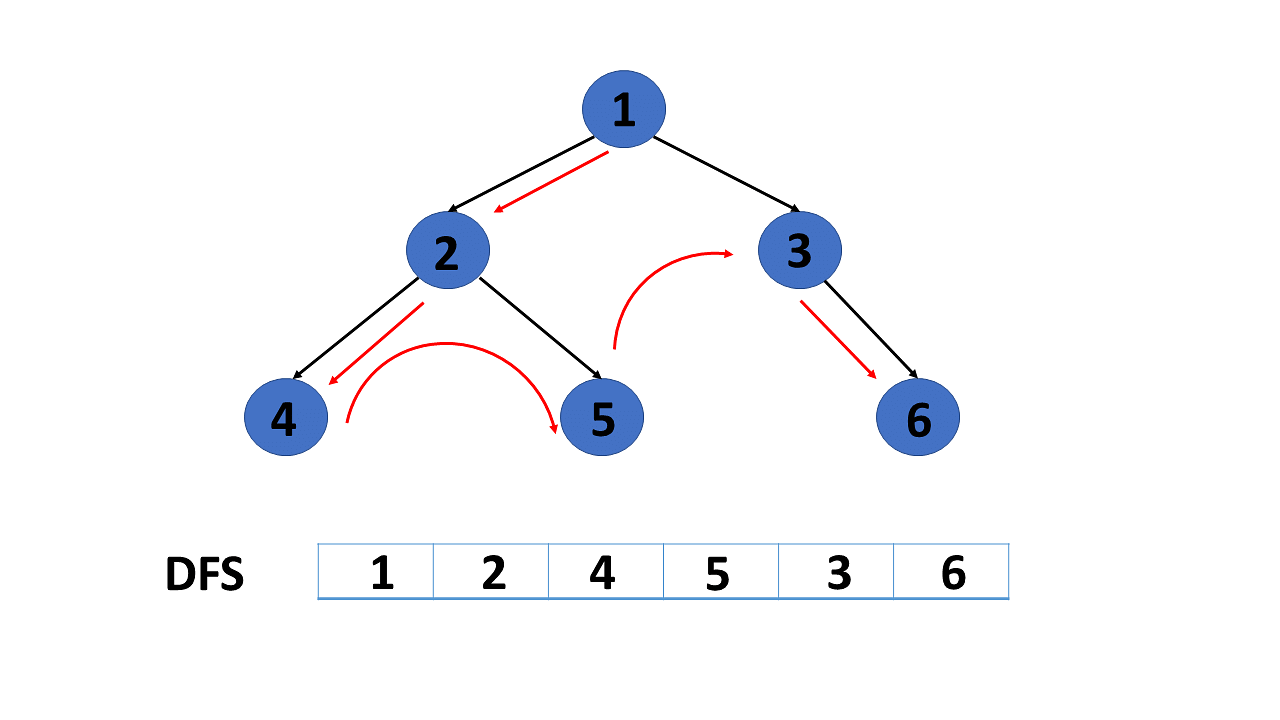
* Visited array (size of the graph)
* Stack data structure

Step 4: Insert v1 into the array's first block and push all the adjacent nodes or vertices of vertex v1 into the stack.

Step 5: Now, using the FIFO principle, pop the topmost element and put it into the visited array, pushing all of the popped element's nearby nodes into it.

Step 6: If the topmost element of the stack is already present in the array, discard it instead of inserting it into the visited array.

Step 7: Repeat step 6 until the stack data structure isn't empty.



You will now look at applications of graph data structures after understanding the graph traversal algorithm in this tutorial.

## **Application of Graphs in Data Structures**

Following  are some applications of graphs in data structures:

* Graphs are used in computer science to depict the flow of computation.
* Users on Facebook are referred to as vertices, and if they are friends, there is an edge connecting them. The Friend Suggestion system on Facebook is based on graph theory.
* You come across the Resource Allocation Graph in the Operating System, where each process and resource are regarded vertically. Edges are drawn from resources to assigned functions or from the requesting process to the desired resource. A stalemate will develop if this results in the establishment of a cycle.
* Web pages are referred to as vertices on the World Wide Web. Suppose there is a link from page A to page B that can represent an edge. This application is an illustration of a directed graph.
* Graph transformation systems manipulate graphs in memory using rules. Graph databases store and query graph-structured data in a transaction-safe, permanent manner.

.