## Data structure

Data structure is a storage that is used to store and organise data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.

#### Types of Data Structure

Basically, data structures are divided into two categories:

* Linear data structure
* Non-linear data structure

##### Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

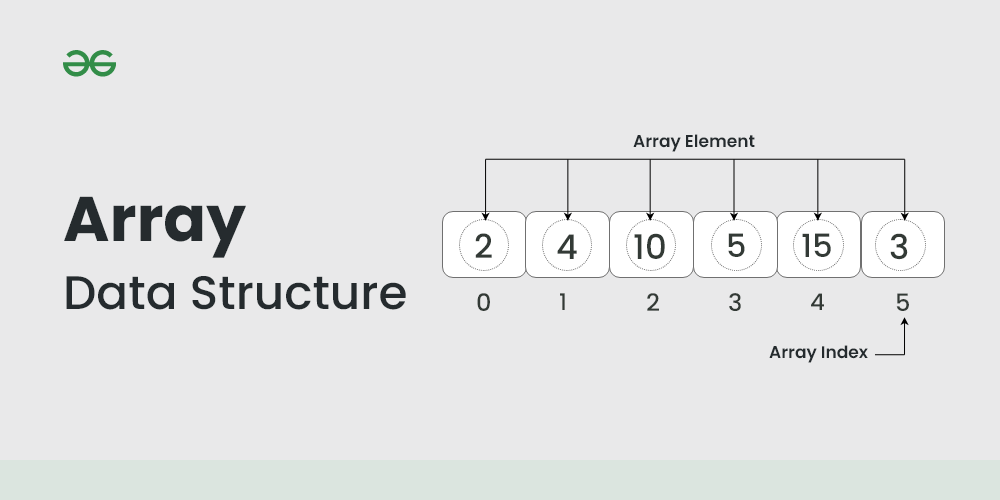
However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities

Popular linear data structures are:

1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

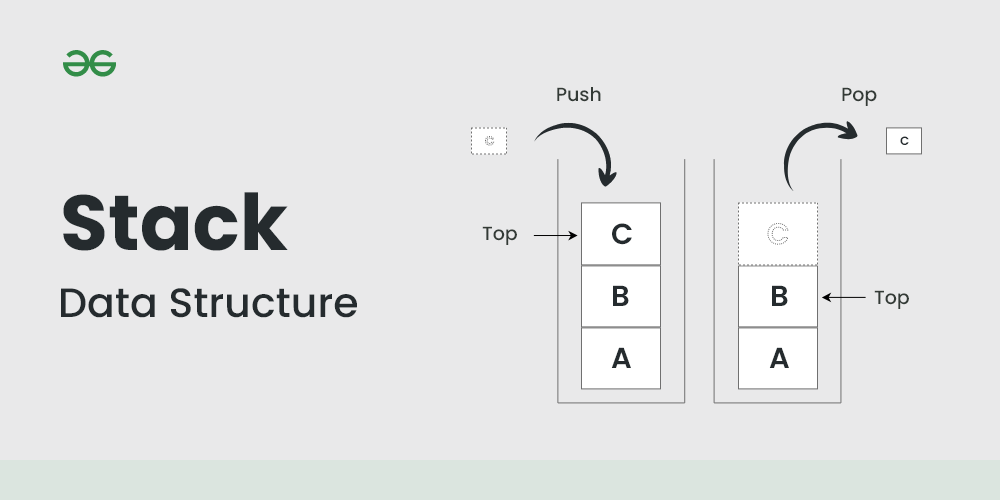
An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

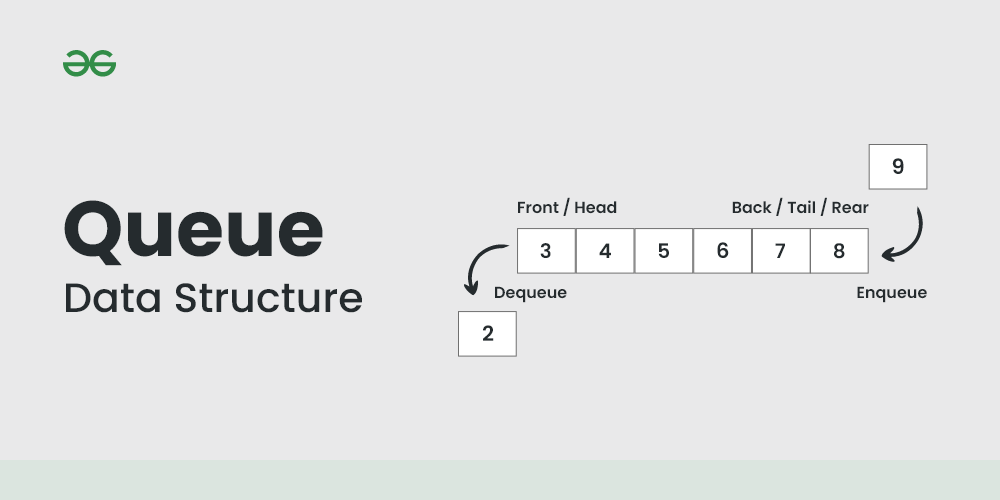
It works just like a pile of plates where the last plate kept on the pile will be removed first.



3. Queue Data Structure

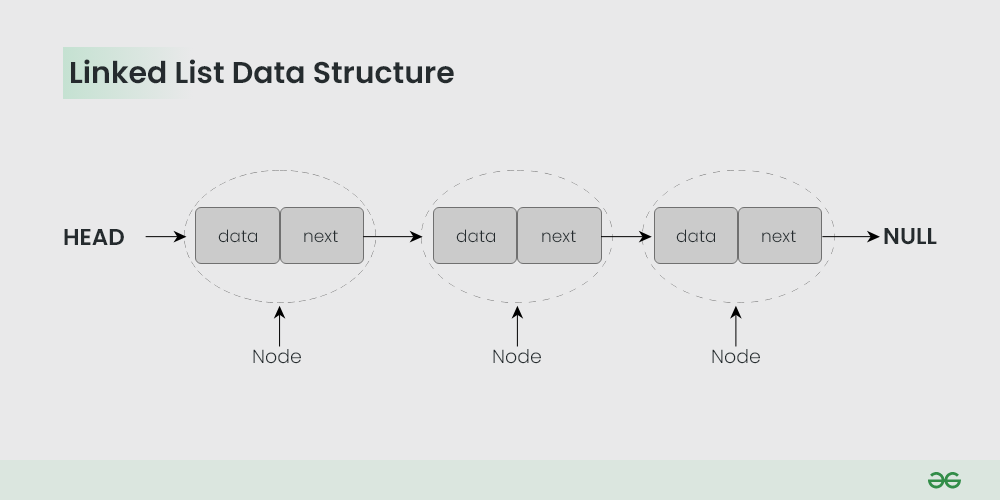
Unlike stack, the queue data structure works in the FIFO principle where the first element stored in the queue will be removed first.

It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first



4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.



##### Non linear data structures

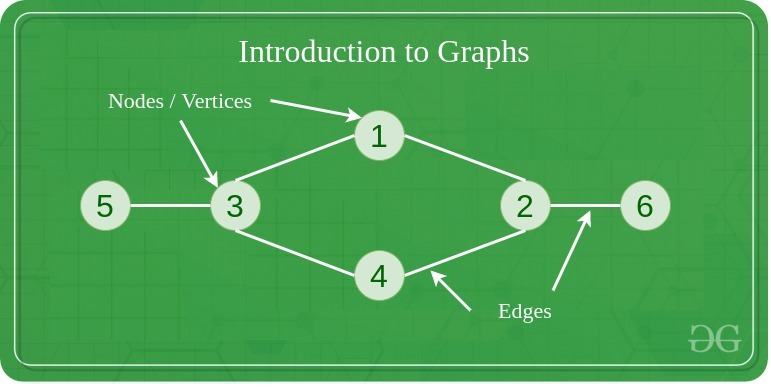
non-linear data structures are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.

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



2. Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.

##### Difference Linear Vs Non-linear Data Structures

| Linear Data Structures | Non Linear Data Structures |
| --- | --- |
| The data items are arranged in sequential order, one after the other. | The data items are arranged in non-sequential order (hierarchical manner). |
| All the items are present on the single layer. | The data items are present at different layers. |
| It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass. | It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass. |
| The memory utilization is not efficient. | Different structures utilize memory in different efficient ways depending on the need. |
| The time complexity increase with the data size. | Time complexity remains the same. |
| Example: Arrays, Stack, Queue | Example: Tree, Graph, Map |

###### **What are Algorithms?**

Informally, an algorithm is nothing but a mention of steps to solve a problem. They are essentially a solution.

The time taken by the computer to run code is:

Time to run code = number of instructions \* time to execute each instruction

Data structures are used to hold data while algorithms are used to solve the problem using that data.

—-------------------------------------------------------------------------------------------------

# **Asymptotic Analysis: Big-O Notation and More**

## **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.

# Asymptotic Notations: Big O, Big Omega and Big Theta Explained (With Notes)

Asymptotic notation gives us an idea about how good a given algorithm is compared to some other algorithm.

Now let's look at the mathematical definition of 'order of.' Primarily there are three types of widely used asymptotic notations.

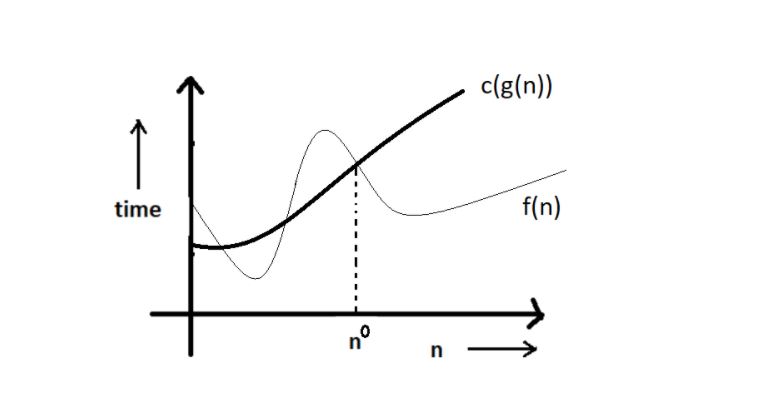
1. Big oh notation ( O )
2. Big omega notation ( Ω )
3. Big theta notation ( θ ) – Widely used one

#### Big oh notation ( O ):

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 oh notation is used to describe an asymptotic upper bound.
* Mathematically, if f(n) describes the running time of an algorithm; f(n) is O(g(n)) if and only if there exist positive constants c and n° such that:  
    
  0 ≤ f(n) ≤ c g(n) for all n ≥ n°.
* Here, n is the input size, and g(n) is any complexity function, for, e.g. n, n2, etc. (It is used to give upper bound on a function)
* If a function is O(n), it is automatically O(n2) as well! Because it satisfies the equation given above.

#### **Graphic example for Big oh ( O ):**



#### Big Omega Notation ( Ω ):

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

* Just like O notation provides an asymptotic upper bound, Ω notation provides an asymptotic lower bound.
* Let f(n) define the running time of an algorithm; f(n) is said to be Ω (g(n)) if and only if there exist positive constants c and n° such that:  
    
  0 ≤ c g(n) ≤ f(n) for all n ≥ n°.
* It is used to give the lower bound on a function.
* If a function is Ω (n2) it is automatically Ω (n) as well since it satisfies the above equation.

#### Graphic example for Big Omega (Ω):



#### Big theta notation ( θ ):

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.

* Let f(n) define the running time of an algorithm.
* F(n) is said to be θ (g(n)) if f(n) is O (g(n)) and f(x) is Ω (g(n)) both.

Mathematically,



Merging both the equations, we get:

0 ≤ c2 g(n) ≤ f(n) ≤ c1 g(n) ∀ n ≥ no.

Copy

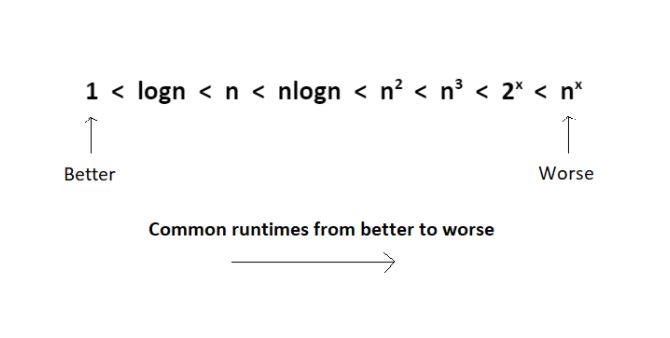
The equation simply means that there exist positive constants c1 and c2 such that f(n) is sandwiched between c2 g(n) and c1 g(n).

#### Graphic example of Big theta ( θ ):



#### Increasing order of common runtimes:

Below mentioned are some common runtimes which you will come across in your coding career.



How to calculate time complexity?

code 1

for (let i = 0; i < n; ++i) {

console.log(i);

}

Here the loop will run n times  
Time Complexity: O(n)

code 2

for (let i = 0; i < n; ++i) {

for (let j = 0; j < n; ++j) {

console.log(i, j);

}

}

Here the loop will run n² times  
Time Complexity: O(n²)

code 3

for (let i = 1; i < n; i \*= 2) {

console.log(i);

}

Here the loop will run (log n)-1 times

Time Complexity: O(log n)

What is space complexity?

Space complexity of a program is a simple measurement of how fast the space taken by a program grows, if the input increases.

Why should we care about space complexity?

A good algorithm keeps space complexity as low as possible. An algorithm with lower space complexity is always better than the one with higher.

There is often a time-space tradeoff involved. A case where an algorithm increases space usage with decreased time or vice versa.

# Examples

Method 1

function fibonacci(n) {

const arr = [0, 1];

for (let i = 2; i <= n; ++i) {

arr.push(arr[i - 2] + arr[ i - 1]);

}

return arr[n - 1];

}

Here we are using an array of size n and a fixed space for iteration. Hence the space complexity is O(n).

Method 2

function fibonacci(n) {

let x = 0, y = 1, z;

if (n === 0) {

return x;

}

if (n === 1) {

return y;

}

for (let i = 2; i <= n; ++i) {

z = x + y;

x = y;

y = z;

}

return z;

}

Here we are using a fixed space for 4 variables. Hence the space complexity is O(1)

# **Conclusion**

The second method is better.There is no point in using more space to solve a problem if we can do the same with lesser space complexity.

For reference :<https://medium.com/@manishsundriyal/overview-time-space-complexity-f973513b701e>

Tower of Hanoi