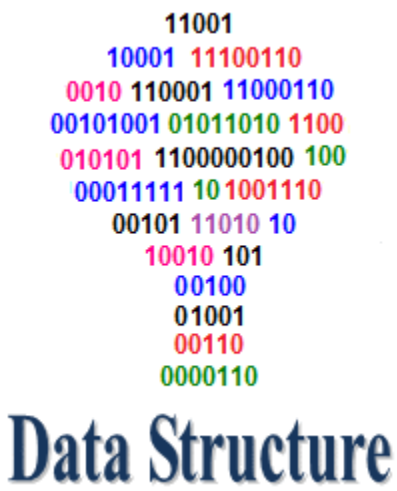
# Data Structures Tutorial




Data Structures (DS) tutorial provides basic and advanced concepts of Data Structure. Our Data Structure tutorial is designed for beginners and professionals.

Data Structure is a way to store and organize data so that it can be used efficiently.

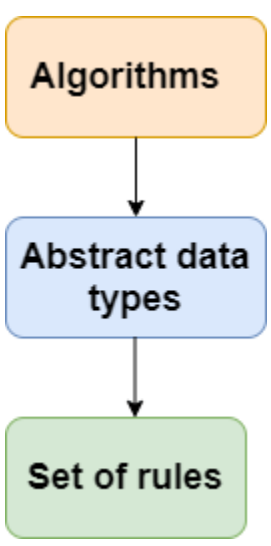
Our Data Structure tutorial includes all topics of Data Structure such as Array, Pointer, Structure, Linked List, Stack, Queue, Graph, Searching, Sorting, Programs, etc.

## What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.




## Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

**Primitive Data structure**

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

**Non-Primitive Data structure**

The non-primitive data structure is divided into two types:

- Linear data structure

- Non-linear data structure

**Linear Data Structure**

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

**When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.**

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures.

**Data structures can also be classified as:**

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

## Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

## Which Data Structure?

A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time. For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the implementation. Therefore, we conclude that we require some data structure to implement a particular ADT.

An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part. Now the question arises: how can one get to know which data structure to be used for a particular ADT?.

As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space. For example, the Stack ADT can be implemented by both Arrays and linked list. Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.

## Advantages of Data structures

**The following are the advantages of a data structure:**

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structure provides reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

# Data Structure

## Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

## Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned.

**Data:** Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.


**Group Items:** Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

**Record:** Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

**File:** A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

**Attribute and Entity:** An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:** Field is a single elementary unit of information representing the attribute of an entity.

## Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arrise the following problems:

**Processor speed:** To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

**Data Search:** Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

**Multiple requests:** If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

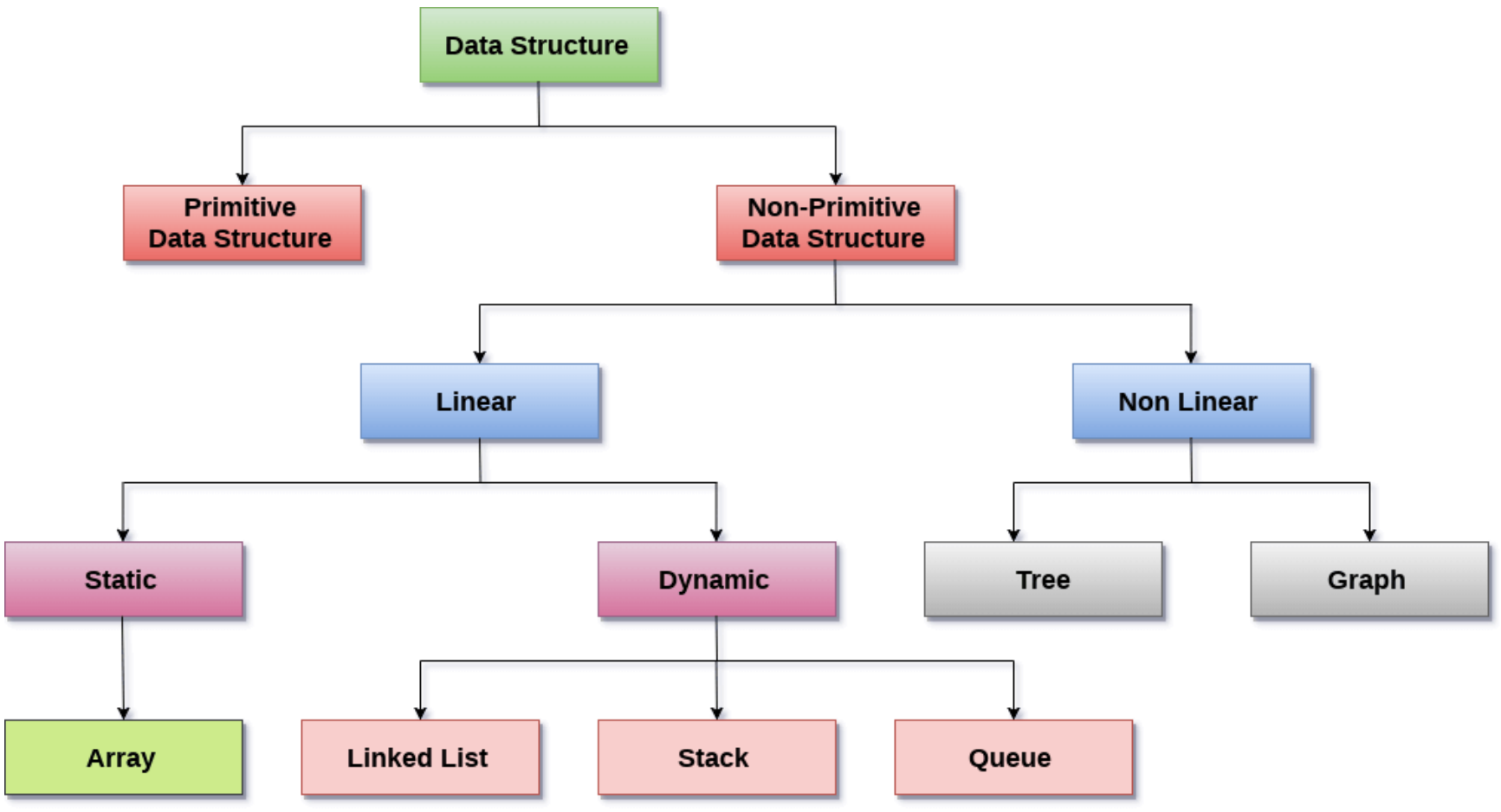
## Advantages of Data Structures

**Efficiency:** Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

**Reusability:** Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

**Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

# Data Structure Classification




**Linear Data Structures:** A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

**Arrays:** An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

**Linked List:** Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

**Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

**Queue:** Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

**Non Linear Data Structures:** This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

**Trees:** Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

**Graphs:** Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

## Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

**Example:** If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

# DS Algorithm

## What is an Algorithm?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

## Characteristics of an Algorithm

**The following are the characteristics of an algorithm:**

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

## Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

## Why do we need Algorithms?

**We need algorithms because of the following reasons:**

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Let's understand the algorithm through a real-world example. Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:

Step 1: First, we will cut the lemon into half.

Step 2: Squeeze the lemon as much you can and take out its juice in a container.

Step 3: Add two tablespoon sugar in it.

Step 4: Stir the container until the sugar gets dissolved.

Step 5: When sugar gets dissolved, add some water and ice in it.

Step 6: Store the juice in a fridge for 5 to minutes.

Step 7: Now, it's ready to drink.

The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Now we will look an example of an algorithm in programming.

We will write an algorithm to add two numbers entered by the user.

**The following are the steps required to add two numbers entered by the user:**

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.

Step 5: Print sum

Step 6: Stop

## Factors of an Algorithm

**The following are the factors that we need to consider for designing an algorithm:**

- **Modularity:** If any problem is given and we can break that problem into small-small modules or small-small steps, which is a basic definition of an algorithm, it means that this feature has been perfectly designed for the algorithm.
- **Correctness:** The correctness of an algorithm is defined as when the given inputs produce the desired output, which means that the algorithm has been designed algorithm. The analysis of an algorithm has been done correctly.

- **Maintainability:** Here, maintainability means that the algorithm should be designed in a very simple structured way so that when we redefine the algorithm, no major change will be done in the algorithm.
- **Functionality:** It considers various logical steps to solve the real-world problem.
- **Robustness:** Robustness means that how an algorithm can clearly define our problem.
- **User-friendly:** If the algorithm is not user-friendly, then the designer will not be able to explain it to the programmer.
- **Simplicity:** If the algorithm is simple then it is easy to understand.
- **Extensibility:** If any other algorithm designer or programmer wants to use your algorithm then it should be extensible.

## Importance of Algorithms

1. **Theoretical importance:** When any real-world problem is given to us and we break the problem into small-small modules. To break down the problem, we should know all the theoretical aspects.
2. **Practical importance:** As we know that theory cannot be completed without the practical implementation. So, the importance of algorithm can be considered as both theoretical and practical.

## Issues of Algorithms

**The following are the issues that come while designing an algorithm:**

- **How to design algorithms:** As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.
- **How to analyze algorithm efficiency**

## Approaches of Algorithm

**The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:**

- **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:
    1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
    2. **Sacrificing:** As soon as the best solution is found, then it will stop.
- **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.
- **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But, there are very rare cases in which it provides the optimal solution.
- **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
    1. It breaks down the problem into a subproblem to find the optimal solution.
    2. After breaking down the problem, it finds the optimal solution out of these subproblems.
    3. Stores the result of the subproblems is known as memorization.
    4. Reuse the result so that it cannot be recomputed for the same subproblems.
    5. Finally, it computes the result of the complex program.
- **Branch and Bound Algorithm:** The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.
- **Randomized Algorithm:** As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm?. In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.
- **Backtracking:** Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

The major categories of algorithms are given below:

- **Sort:** Algorithm developed for sorting the items in a certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

## Algorithm Analysis

The algorithm can be analyzed in two levels, i.e., first is before creating the algorithm, and second is after creating the algorithm. The following are the two analysis of an algorithm:

- Priori Analysis: Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.
- Posterior Analysis: Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

## Algorithm Complexity

The performance of the algorithm can be measured in two factors:

- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
1. sum=0;
2. // Suppose we have to calculate the sum of n numbers.
3. for i=1 to n
4. sum=sum+i;
5. // when the loop ends then sum holds the sum of the n numbers
6. return sum;
```

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

- **Space complexity:** An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

So,

**Space complexity = Auxiliary space + Input size.**

## Types of Algorithms

**The following are the types of algorithm:**

- **Search Algorithm**

- **Sort Algorithm**

**Search Algorithm**

On each day, we search for something in our day to day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search the data in an array:

- **Linear search**
- **Binary search**

**Linear Search**

Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found. It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1. This algorithm can be implemented on the unsorted list.

**Binary Search**

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

## Sorting Algorithms

Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.

## Why do we need a sorting algorithm?

- An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
- It produces information in a sorted order, which is a human-readable format.
- Searching a particular element in a sorted list is faster than the unsorted list.

# Asymptotic Analysis

As we know that data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space. So, the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space. Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm.

The main question arises in our mind that on what basis should we compare the time complexity of data structures?. The time complexity can be compared based on operations performed on them. Let's consider a simple example.

Suppose we have an array of 100 elements, and we want to insert a new element at the beginning of the array. This becomes a very tedious task as we first need to shift the elements towards the right, and we will add new element at the starting of the array.

Suppose we consider the linked list as a data structure to add the element at the beginning. The linked list contains two parts, i.e., data and address of the next node. We simply add the address of the first node in the new node, and head pointer will now point to the newly added node. Therefore, we conclude that adding the data at the beginning of the linked list is faster than the arrays. In this way, we can compare the data structures and select the best possible data structure for performing the operations.

## How to find the Time Complexity or running time for performing the operations?

The measuring of the actual running time is not practical at all. The running time to perform any operation depends on the size of the input. Let's understand this statement through a simple example.

Suppose we have an array of five elements, and we want to add a new element at the beginning of the array. To achieve this, we need to shift each element towards right, and suppose each element takes one unit of time. There are five elements, so five units of time would be taken. Suppose there are 1000 elements in an array, then it takes 1000 units of time to shift. It concludes that time complexity depends upon the input size.

Therefore, if the input size is n, then f(n) is a function of n that denotes the time complexity.

## How to calculate f(n)?

Calculating the value of f(n) for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their f(n) values. We can compare the data structures by comparing their f(n) values. We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes. Now, how to find f(n).

Let's look at a simple example.

$f(n) = 5n^2 + 6n + 12$

where n is the number of instructions executed, and it depends on the size of the input.

When n=1

% of running time due to $5n^2 = \frac{5}{5+6+12} * 100 = 21.74\%$

% of running time due to $6n = \frac{6}{5+6+12} * 100 = 26.09\%$

% of running time due to $12 = \frac{12}{5+6+12} * 100 = 52.17\%$

From the above calculation, it is observed that most of the time is taken by 12. But, we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of f(n).

| n | $5n^2$ | 6n | 12 |
|---|---|---|---|
| 1 | 21.74% | 26.09% | 52.17% |
| 10 | 87.41% | 10.49% | 2.09% |
| 100 | 98.79% | 1.19% | 0.02% |
| 1000 | 99.88% | 0.12% | 0.0002% |

As we can observe in the above table that with the increase in the value of n, the running time of $5n^2$ increases while the running time of 6n and 12 also decreases. Therefore, it is observed that for larger values of n, the squared term consumes almost 99% of the time. As the $n^2$ term is contributing most of the time, so we can eliminate the rest two terms.

**Therefore,**

$f(n) = 5n^2$

Here, we are getting the approximate time complexity whose result is very close to the actual result. And this approximate measure of time complexity is known as an Asymptotic complexity. Here, we are not calculating the exact running time, we are eliminating the unnecessary terms, and we are just considering the term which is taking most of the time.

In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.

It is used to mathematically calculate the running time of any operation inside an algorithm.

**Example:** Running time of one operation is x(n) and for another operation, it is calculated as f(n2). It refers to running time will increase linearly with an increase in 'n' for the first operation, and running time will increase exponentially for the second operation. Similarly, the running time of both operations will be the same if n is significantly small.

Usually, the time required by an algorithm comes under three types:

**Worst case:** It defines the input for which the algorithm takes a huge time.

**Average case:** It takes average time for the program execution.

**Best case:** It defines the input for which the algorithm takes the lowest time

## Asymptotic Notations

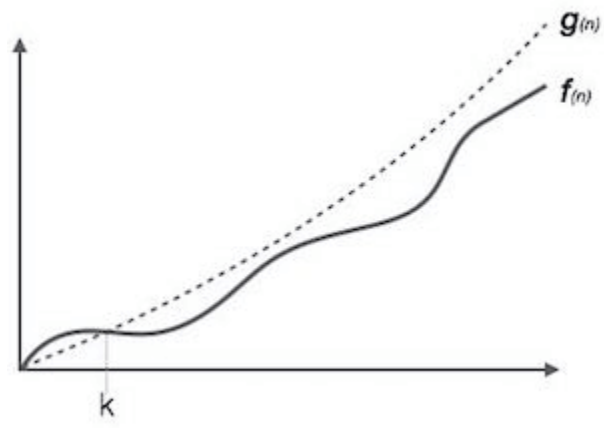
The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (?)
- Omega Notation (Ω)
- Theta Notation (θ)

## Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:




**For example:**

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

**f(n)≤c.g(n) for all n≥no**

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

**Let's understand through examples**

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

**f(n)<=c.g(n)**

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

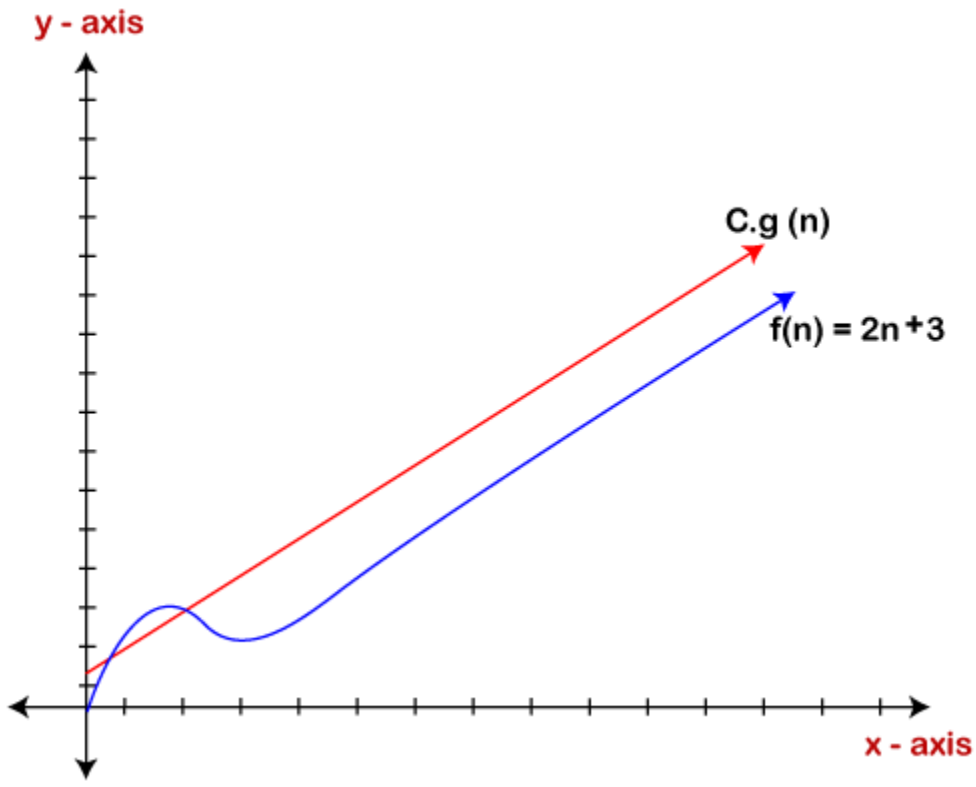
If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:




The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

## Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

**f(n)>=c.g(n) for all n≥no and c>0**

**Let's consider a simple example.**

If f(n) = 2n+3, g(n) = n,

Is f(n)= **Ω** (g(n))?

It must satisfy the condition:

**f(n)>=c.g(n)**

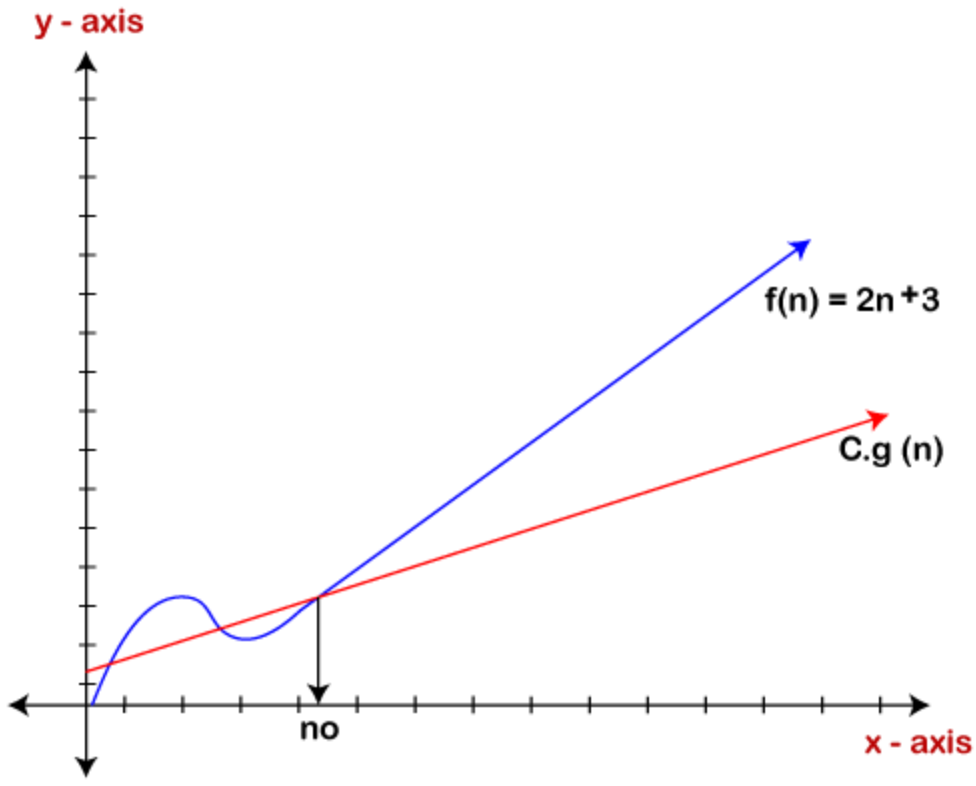
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

**2n+3>=c*n**

Suppose c=1

**2n+3>=n** (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.




As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

## Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let's understand the big theta notation mathematically:

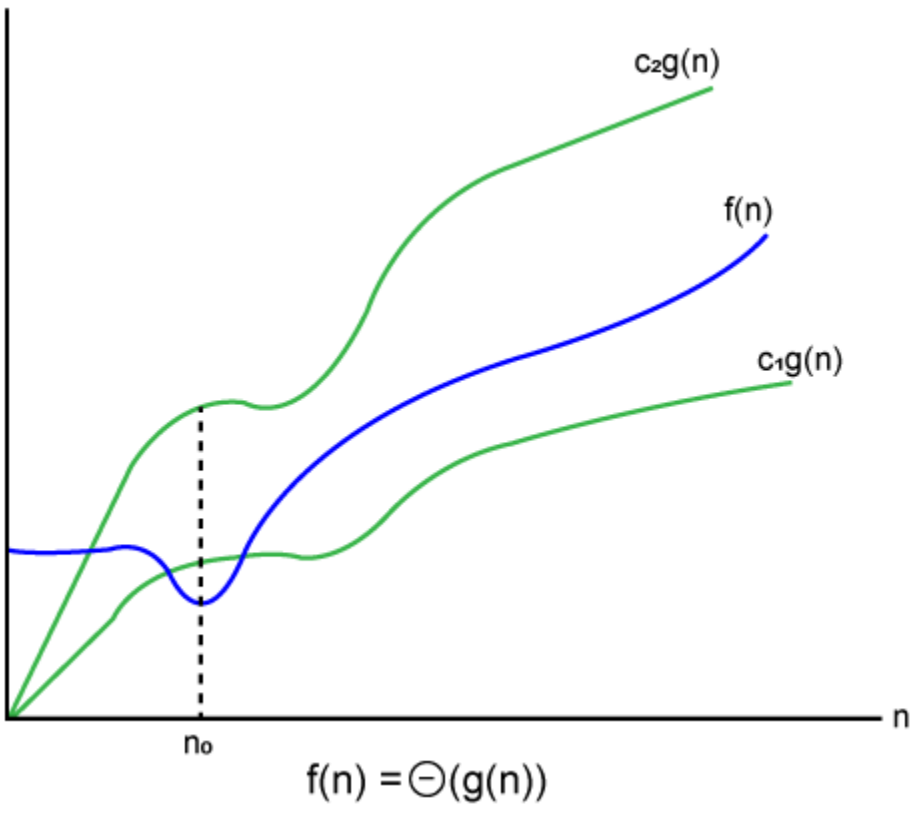
Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

**f(n)= θg(n)**

The above condition is satisfied only if when

**c1.g(n)<=f(n)<=c2.g(n)**

where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



Let's consider the same example where
f(n)=2n+3
g(n)=n

As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).

c1.g(n)<=f(n)<=c2.g(n)

Replace g(n) by n and f(n) by 2n+3

c1.n <=2n+3<=c2.n

if c1=1, c2=2, n=1

1*1 <=2*1+3 <=2*1

**1** <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

**If n=2**

1*2<=2*2+3<=2*2

2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

## Why we have three different asymptotic analysis?

As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm.

Suppose we have an array of n numbers, and we want to find the particular element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose, if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst case would be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.

So, three different analysis provide the proper bounding between the actual functions. Here, bounding means that we have upper as well as lower limit which assures that the algorithm will behave between these limits only, i.e., it will not go beyond these limits.

## Common Asymptotic Notations

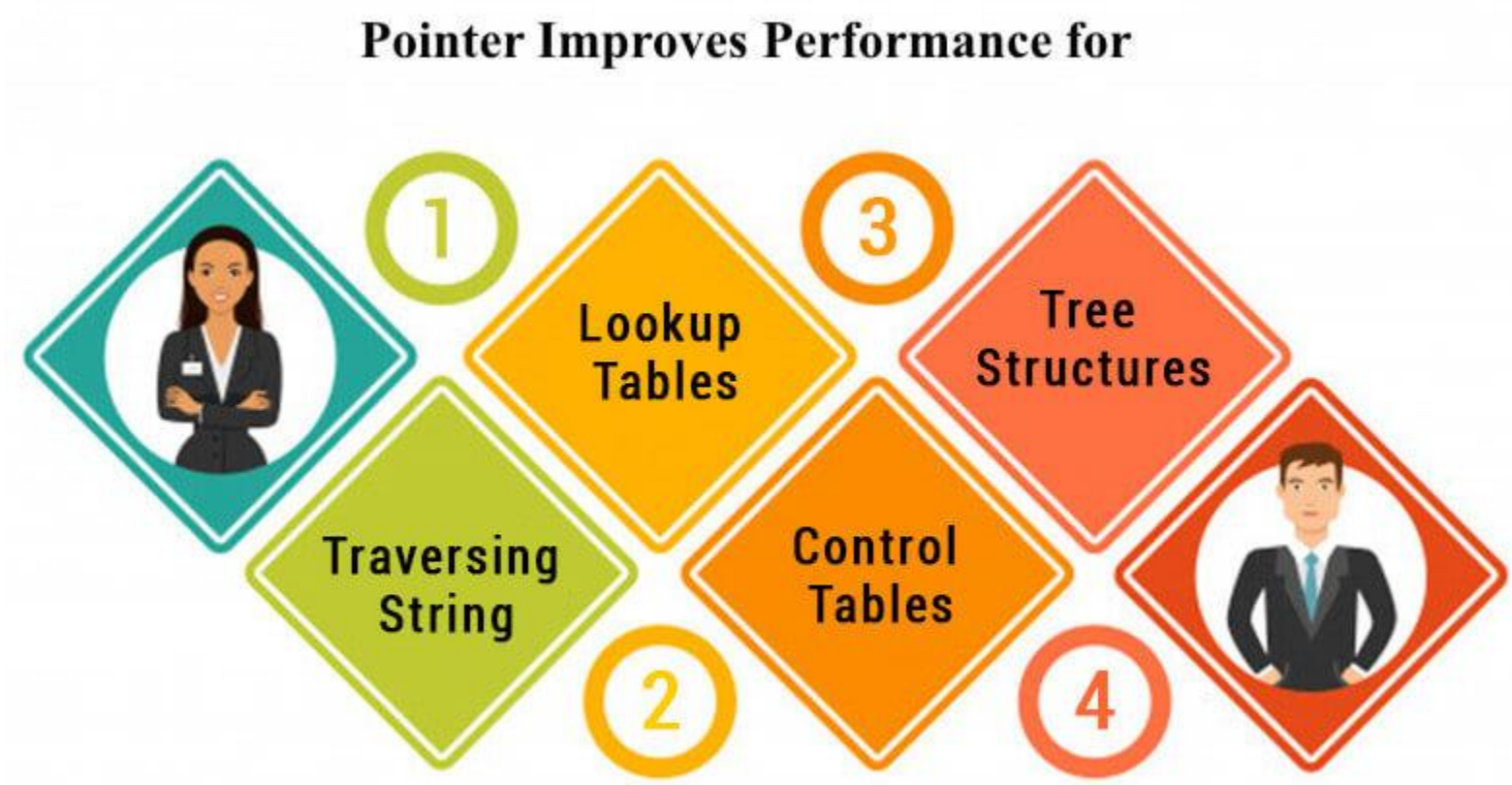
| | | |
|---|---|---|
| constant | - | ?(1) |
| linear | - | ?(n) |
| logarithmic | - | ?(log n) |
| n log n | - | ?(n log n) |
| exponential | - | 2?(n) |
| cubic | - | ?(n3) |
| polynomial | - | n?(1) |
| quadratic | - | ?(n2) |

# Pointer

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

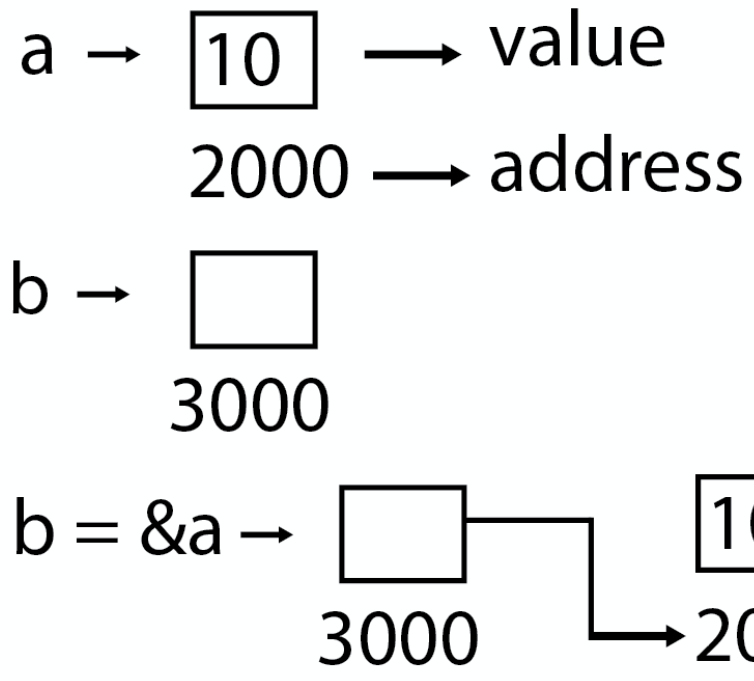
- Traversing String
- Lookup Tables
- Control Tables

- Tree Structures




## Pointer Details

- **Pointer arithmetic:** There are four arithmetic operators that can be used in pointers: ++, --, +, -
- **Array of pointers:** You can define arrays to hold a number of pointers.
- **Pointer to pointer:** C allows you to have pointer on a pointer and so on.
- **Passing pointers to functions in C:** Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
- **Return pointer from functions in C:** C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.




### Program

***Pointer***

```
1. #include <stdio.h>
2.
3. int main( )
4. {
```

```
5.  int a = 5;
6.  int *b;
7.  b = &a;
8.
9.  printf ("value of a = %d\n", a);
10. printf ("value of a = %d\n", *(&a));
11. printf ("value of a = %d\n", *b);
12. printf ("address of a = %u\n", &a);
13. printf ("address of a = %d\n", b);
14. printf ("address of b = %u\n", &b);
15. printf ("value of b = address of a = %u", b);
16. return 0;
17. }
```

***Output***

```
1. value of a = 5
2. value of a = 5
3. address of a = 3010494292
4. address of a = -1284473004
5. address of b = 3010494296
6. value of b = address of a = 3010494292
```

## Program

### *Pointer to Pointer*

```
1.  #include <stdio.h>
2.
3.  int main( )
4.  {
5.  int a = 5;
6.  int *b;
7.  int **c;
8.  b = &a;
9.  c = &b;
10. printf ("value of a = %d\n", a);
11. printf ("value of a = %d\n", *(&a));
12. printf ("value of a = %d\n", *b);
13. printf ("value of a = %d\n", **c);
14. printf ("value of b = address of a = %u\n", b);
15. printf ("value of c = address of b = %u\n", c);
16. printf ("address of a = %u\n", &a);
17. printf ("address of a = %u\n", b);
18. printf ("address of a = %u\n", *c);
19. printf ("address of b = %u\n", &b);
20. printf ("address of b = %u\n", c);
21. printf ("address of c = %u\n", &c);
22. return 0;
23. }
```

### *Pointer to Pointer*

```
1. value of a = 5
2. value of a = 5
3. value of a = 5
4. value of a = 5
5. value of b = address of a = 2831685116
6. value of c = address of b = 2831685120
7. address of a = 2831685116
8. address of a = 2831685116
```

```
address of a = 2831685116
address of b = 2831685120
address of b = 2831685120
address of c = 2831685128
```

# Structure

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

**Syntax**

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeber;
};
```

## Advantages

- It can hold variables of different data types.
- We can create objects containing different types of attributes.
- It allows us to re-use the data layout across programs.
- It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

**Program**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
struct employee
{
int id ;
float salary ;
int mobile ;
} ;
struct employee e1,e2,e3 ;
clrscr();
printf ("\nEnter ids, salary & mobile no. of 3 employee\n"
scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);
scanf ("%d%f %d", &e2.id, &e2.salary, &e2.mobile);
scanf ("%d %f %d", &e3.id, &e3.salary, &e3.mobile);
printf ("\n Entered Result ");
printf ("\n%d %f %d", e1.id, e1.salary, e1.mobile);
printf ("\n%d%f %d", e2.id, e2.salary, e2.mobile);
printf ("\n%d %f %d", e3.id, e3.salary, e3.mobile);
getch();
}
```