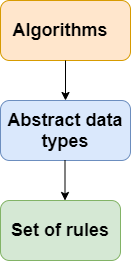
**DATA STRUCTURES**

**DS - TUTORIAL**

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.

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.



ADTs provide a way to abstract the complexity of data structures, allowing programmers to work with high-level specifications without worrying about the low-level details. The actual implementation of an ADT can vary, and different data structures can satisfy the requirements of the same ADT.

Top of Form

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

**Compile Time:**

* **Definition:** Compile time, also known as compilation time, is the time during which a program's source code is translated into machine code or an intermediate code by a compiler.

**Run Time:**

* **Definition:** Run time, also known as execution time, is the time when a compiled program is actually running and performing its tasks.

**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.

**--DS tutorial completed**

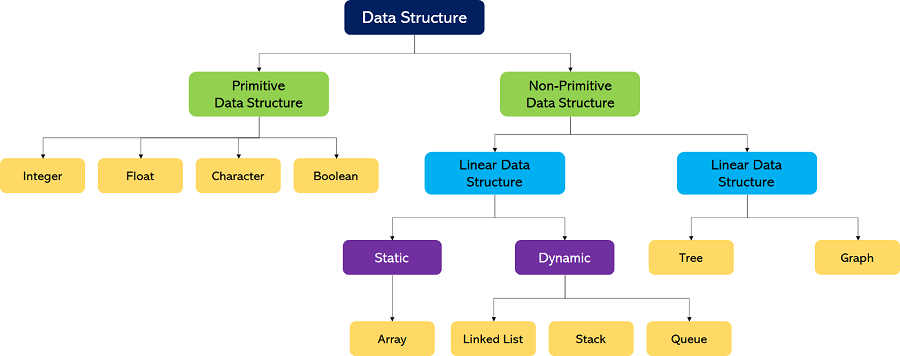
**DS – INTRODUCTION**

**Data** is a collection of facts and figures or a set of values or values of a specific format that refers to a single set of item values. The data items are then classified into sub-items, which is the group of items that are not known as the simple primary form of the item.

## **Classification of Data Structures**

1. Primitive Data Structure
2. Non-Primitive Data Structure

The following figure shows the different classifications of Data Structures.



### Primitive Data Structures

1. These data structures can be manipulated or operated directly by machine-level instructions.
2. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.

### Non-Primitive Data Structures

1. These data structures can't be manipulated or operated directly by machine-level instructions.
2. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
3. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
   1. Linear Data Structures
   2. Non-Linear Data Structures

### Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element

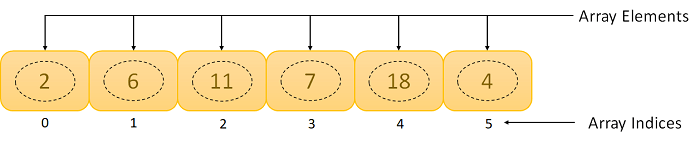
Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.  
   The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code.  
   **Linked Lists, Stacks**, and **Queues** are common examples of dynamic data structures

**1. Arrays**

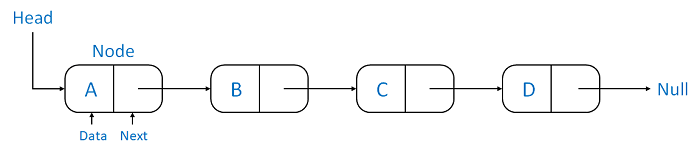
An **Array** is a data structure used to collect multiple data elements of the same data type into one variable. We can access any data element from the list with the help of its location in the list. Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.

|  |  |
| --- | --- |
| Contiguous memory allocation allocates consecutive blocks of memory to a file/process. | Non-Contiguous memory allocation allocates separate blocks of memory to a file/process. |



**2. Linked Lists**

A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

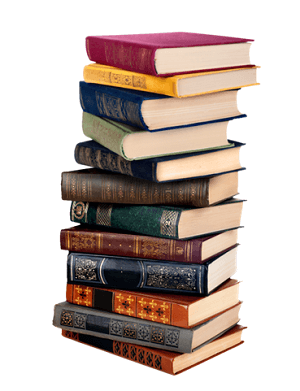


**Linked Lists can be classified into different types:**

1. **Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.
2. **Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).
3. **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

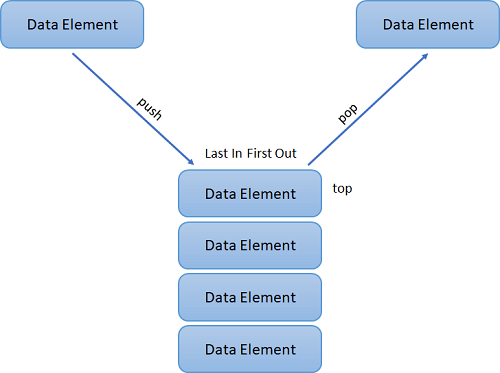
**3. Stacks**

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.



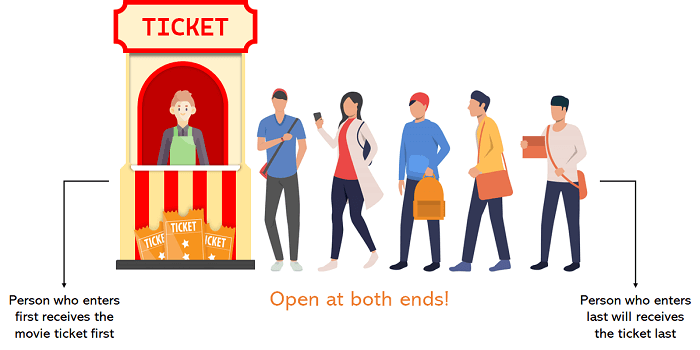
**The primary operations in the Stack are as follows:**

1. **Push:** Operation to insert a new element in the Stack is termed as Push Operation.
2. **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.



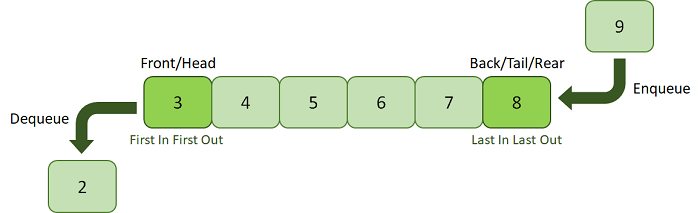
**4. Queues**

A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.



The following are the primary operations of the Queue:

1. **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.
2. **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.



Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

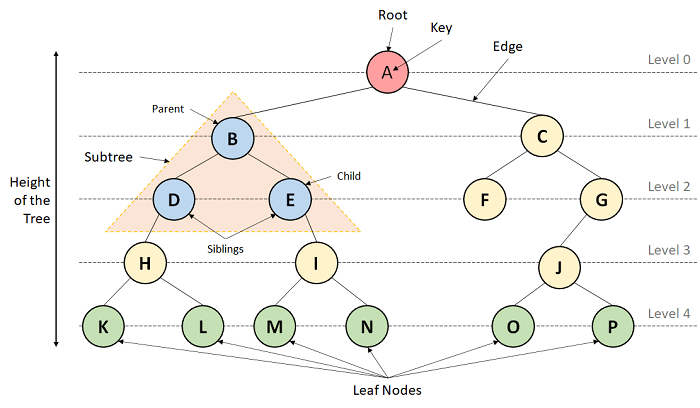
Types of Non-Linear Data Structures

The following is the list of Non-Linear Data Structures that we generally use:

**1. Trees**

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.



**Trees can be classified into different types:**

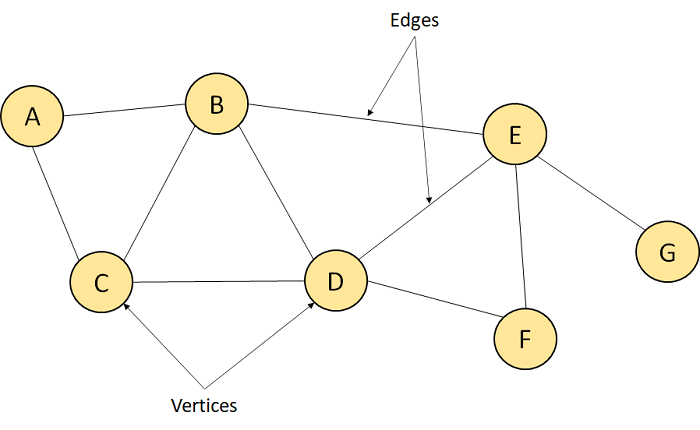
1. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.
2. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.

**2. Graphs**

A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below:

G = (V,E)



## **Basic Operations of Data Structures**

In the following section, we will discuss the different types of operations that we can perform to manipulate data in every data structure:

1. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.
2. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
3. **Insertion:** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.
4. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.
5. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.
6. **Merge:** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.
7. **Create:** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:
   1. Compile-time
   2. Run-time  
      For example, the **malloc()** function is used in C Language to create data structure.
8. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.
9. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.
10. **Splitting:** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

## **Understanding the Abstract Data Type**

we can conclude that the operations in data structure include:

1. A high level of abstractions like addition or deletion of an item from a list.
2. Searching and sorting an item in a list.
3. Accessing the highest priority item in a list.

Whenever the data structure does such operations, it is known as an **Abstract Data Type (ADT)**.

We can define it as a set of data elements along with the operations on the data. The term "abstract" refers to the fact that the data and the fundamental operations defined on it are being studied independently of their implementation. It includes what we can do with the data, not how we can do it.

An ADI implementation contains a storage structure in order to store the data elements and algorithms for fundamental operation. All the data structures, like an array, linked list, queue, stack, etc., are examples of ADT

**DS –- INTRODUCTION COMPLETED**

# **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.

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

* **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
* **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.
5. 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.

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

Types of Algorithms

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

* **Search Algorithm**
* **Sort Algorithm**

**Search Algorithm**

* **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.

**DS ALGORITHM – completed**

### Asymptotic AnalysisAsymptotic 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)

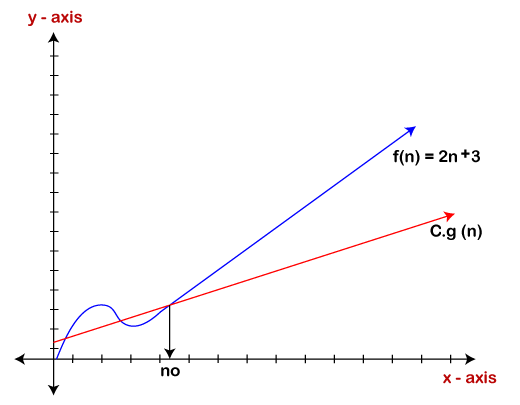
Big oh Notation (O)

* 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.

# Asymptotic Analysis

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.



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.



### 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)**.

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) |

Asymptotic Notations – completed

# **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.



**POINTER – COMPLETED**

# **Array in Data Structure**

Why are arrays required?

Arrays are useful because -

* Sorting and searching a value in an array is easier.
* Arrays are best to process multiple values quickly and easily.
* **Arrays are good for storing multiple values in a single variable -** In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

## **Basic operations**

Now, let's discuss the basic operations supported in the array -

* Traversal - This operation is used to print the elements of the array.
* Insertion - It is used to add an element at a particular index.
* Deletion - It is used to delete an element from a particular index.
* Search - It is used to search an element using the given index or by the value.
* Update - It updates an element at a particular index.
* **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Average Case** | **Worst Case** |
| Access | O(1) | O(1) |
| Search | O(n) | O(n) |
| Insertion | O(n) | O(n) |
| Deletion | O(n) | O(n) |