

Introduction to Data Structure and Algorithms

Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every enterprise application uses various types of data structures in one or the other way.

Why Learn Data Structure and Algorithms?

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** – Consider an inventory of 1 million (10^6) items of a store. If the application is to search an item, it has to search an item in 1 million (10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** – Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** – As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Applications of Data Structure and Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms –

- **Search** – Algorithm to search an item in a data structure.
- **Sort** – Algorithm to sort items in a certain order.
- **Insert** – Algorithm to insert item in a data structure.
- **Update** – Algorithm to update an existing item in a data structure.
- **Delete** – Algorithm to delete an existing item from a data structure.

The following computer problems can be solved using Data Structures –

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

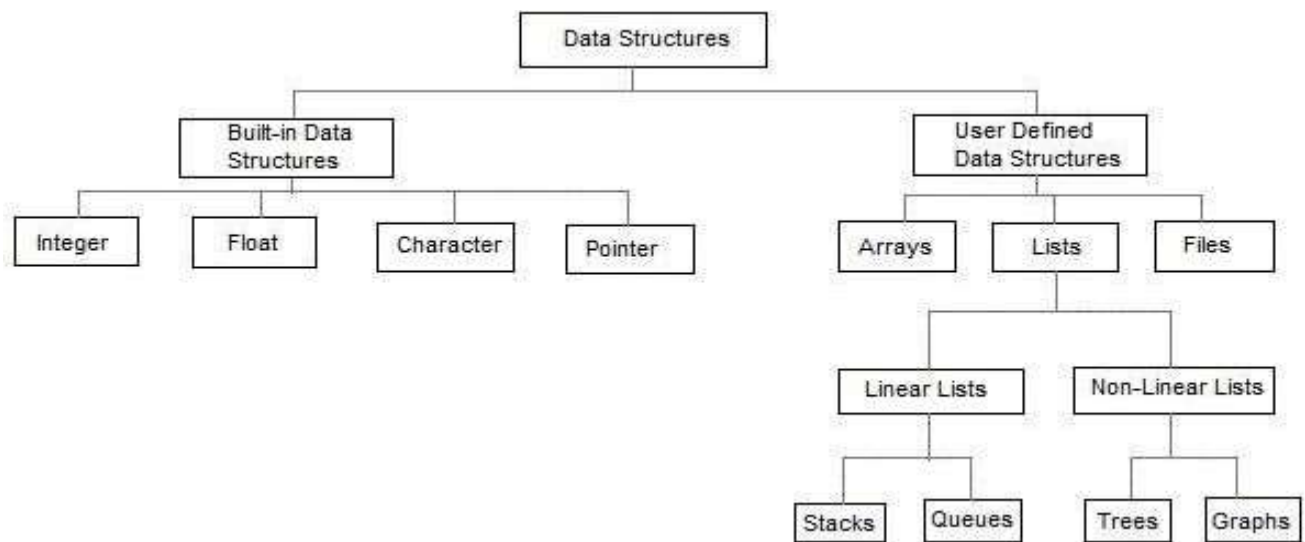
Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

- **Interface** – Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
- **Implementation** – Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Abstract Data Type

In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is usually a n object of that class. In ADT all the implementation details are hidden



- ✓ **Linear data structures** are the data structures in which data is arranged in a list or in a sequence.
- ✓ **Non linear data structures** are the data structures in which data may be arranged in a hierarchic al manner

Characteristics of a Data Structure

- **Correctness** – Data structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** – Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** – Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** – As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- **Worst Case** – This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is $f(n)$ then this operation will not take more than $f(n)$ time where $f(n)$ represents function of n .
- **Average Case** – This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes $f(n)$ time in execution, then m operations will take $mf(n)$ time.
- **Best Case** – This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes $f(n)$ time in execution, then the actual operation may take time as the random number which would be maximum as $f(n)$.

Basic Terminology

- **Data** – Data are values or set of values.
- **Data Item** – Data item refers to single unit of values.
- **Group Items** – Data items that are divided into sub items are called as Group Items.
- **Elementary Items** – Data items that cannot be divided are called as Elementary Items.
- **Attribute and Entity** – An entity is that which contains certain attributes or properties, which may be assigned values.
- **Entity Set** – Entities of similar attributes form an entity set.
- **Field** – Field is a single elementary unit of information representing an attribute of an entity.
- **Record** – Record is a collection of field values of a given entity.
- **File** – File is a collection of records of the entities in a given entity set.

Algorithm Basics

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms –

- **Search** – Algorithm to search an item in a data structure.
- **Sort** – Algorithm to sort items in a certain order.
- **Insert** – Algorithm to insert item in a data structure.
- **Update** – Algorithm to update an existing item in a data structure.
- **Delete** – Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem – Design an algorithm to add two numbers and display the result.

Step 1 – START

Step 2 – declare three integers **a**, **b** & **c**

Step 3 – define values of **a** & **b**

Step 4 – add values of **a** & **b**

Step 5 – store output of step 4 to **c**

Step 6 – print **c**

Step 7 – STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

Step 1 – START ADD

Step 2 – get values of **a** & **b**

Step 3 – $c \leftarrow a + b$

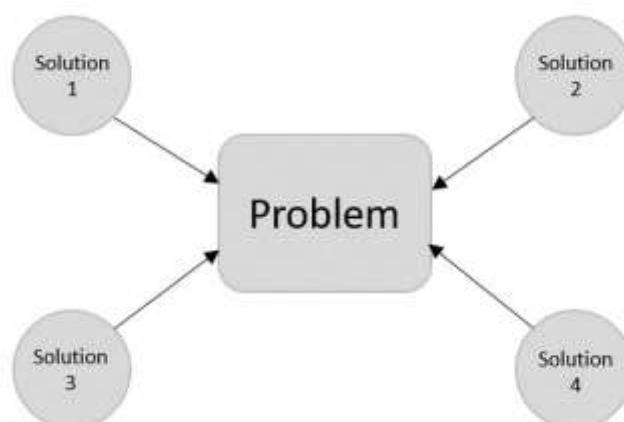
Step 4 – display **c**

Step 5 – STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- **A Priori Analysis** – This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- **A Posterior Analysis** – This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm **X** are the two main factors, which decide the efficiency of **X**.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components –

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity **S(P)** of any algorithm **P** is $S(P) = C + SP(I)$, where **C** is the fixed part and **S(I)** is the variable part of the algorithm, which depends on instance characteristic **I**. Following is a simple example that tries to explain the concept –

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - $C \leftarrow A + B + 10$

Step 3 - Stop

Here we have three variables **A**, **B**, and **C** and one constant. Hence $S(P) = 1 + 3$. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function $T(n)$, where $T(n)$ can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n -bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits. Here, we observe that $T(n)$ grows linearly as the input size increases.