Introduction to Data Structures and Algorithms

Data Structure is a systematic way to organize data in order to use it efficiently. A data structure is a named location that can be used to store and organize data. And, an algorithm is a collection of steps to solve a particular problem. Learning data structures and algorithms allow us to write efficient and optimized computer programs. 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.

## **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(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) 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.

Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.

Then we also have some complex Data Structures, which are used to store large and connected data. Some examples of Abstract Data Structure are:

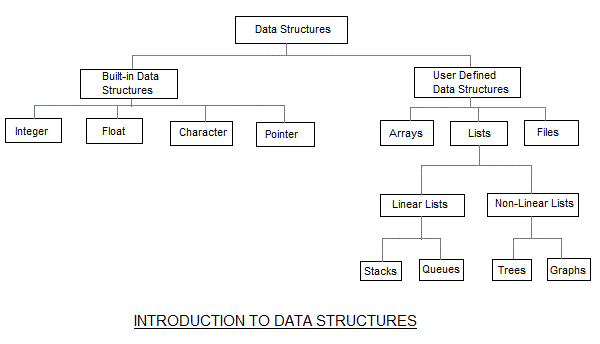
Linked List

Tree

Graph

Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |  |  |  |

The data structures can also be classified on the basis of the following characteristics:

|  |  |
| --- | --- |
| **Characteristic** | **Description** |
| Linear | In Linear data structures, the data items are arranged in a linear sequence. Example: **Array** |
| Non-Linear | In Non-Linear data structures, the data items are not in sequence. Example: **Tree**, **Graph** |
| Homogeneous | In homogeneous data structures, all the elements are of same type. Example: **Array** |
| Non-Homogeneous | In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures** |
| Static | Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array** |
| Dynamic | Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers** |

**What is an Algorithm?**

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high-level description as pseudocode or using a flowchart.

Every Algorithm must satisfy the following properties:

Input- There should be 0 or more inputs supplied externally to the algorithm.

Output- There should be at least 1 output obtained.

Definiteness- Every step of the algorithm should be clear and well defined.

Finiteness- The algorithm should have finite number of steps.

Correctness- Every step of the algorithm must generate a correct output.

**An algorithm** is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties

* 1. **Time Complexity**
  2. **Space Complexity**

**Space Complexity**

It is the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components:

Instruction Space: It is the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.

Data Space: It is the space required to store all the constants and variables (including temporary variables) value.

Environment Space: It is the space required to store the environment information needed to resume the suspended function.

**Time Complexity**

Time Complexity is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes its execution in the minimum time possible. We will study about Time Complexity in details in later sections.

**Data Type**

A *data type* is the most basic and the most common classification of data. It is this through which the compiler gets to know the form or the type of information that will be used throughout the code. So basically, data type is a type of information transmitted between the programmer and the compiler where the programmer informs the compiler about what type of data is to be stored and also tells how much space it requires in the memory. Some basic examples are int, string etc. It is the type of any variable used in the code.

|  |
| --- |
| #include <iostream.h>  **using** **namespace** std;    **void** main()  {  **int** a;      a = 5;    **float** b;      b = 5.0;    **char** c;      c = 'A';    **char** d[10];      d = "example";  } |

As seen from the theory explained above, we come to know that in the above code, the variable ‘a’ is of data type integer which is denoted by int a. So, the variable ‘a’ will be used as an integer type variable throughout the process of the code. And, in the same way, the variables ‘b’, ‘c’ and ‘d’ are of type float, character and string respectively. And all these are kinds of data types.

**Data Structure**

A [*data structure*](https://www.geeksforgeeks.org/data-structures/) is a collection of different forms and different types of data that has a set of specific operations that can be performed. It is a collection of data types. It is a way of organizing the items in terms of memory, and also the way of accessing each item through some defined logic. Some examples of data structures are [stacks](https://www.geeksforgeeks.org/stack-data-structure/), [queues](https://www.geeksforgeeks.org/queue-data-structure/), [linked lists](https://www.geeksforgeeks.org/data-structures/linked-list/), [binary tree](https://www.geeksforgeeks.org/binary-tree-data-structure/) and many more.

Data structures perform some special operations only like insertion, deletion and traversal. For example, you have to store data for many employees where each employee has his name, employee id and a mobile number. So, this kind of data requires complex data management, which means it requires data structure comprised of multiple primitive data types. So, data structures are one of the most important aspects when implementing coding concepts in real-world applications.

**Difference between data type and data structure:**

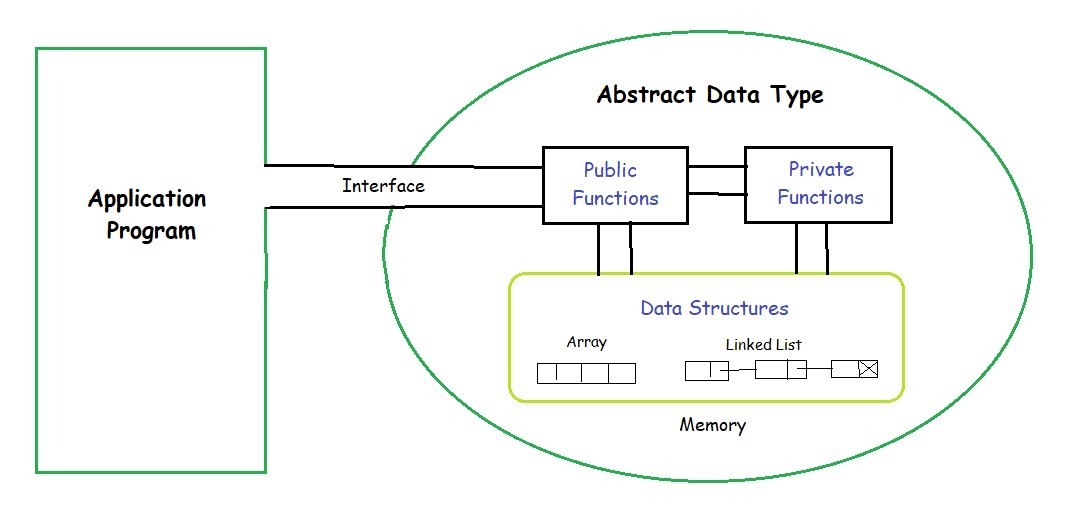
| Data Types | Data Structures |
| --- | --- |
| Data Type is the kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of the given data type only | Data Structure is the collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program. |
| Implementation through Data Types is a form of abstract implementation | Implementation through Data Structures is called concrete implementation |
| Can hold values and not data, so it is data less | Can hold different kind and types of data within one single object |
| Values can directly be assigned to the data type variables | The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on. |
| No problem of time complexity | Time complexity comes into play when working with data structures |
| Examples: int, float, double | Examples: stacks, queues, tree |

**Abstract Data Types**

Data types such as int, float, double, long, etc. are considered to be in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.

[](https://media.geeksforgeeks.org/wp-content/uploads/20190828194629/ADT.jpg)

The user of [data type](https://www.geeksforgeeks.org/data-types-in-c/) does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

So, a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we’ll define three ADTs namely [List](https://www.geeksforgeeks.org/linked-list-set-1-introduction/) ADT, [Stack](https://www.geeksforgeeks.org/stack-data-structure-introduction-program/) ADT, [Queue](https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/) ADT.

1. **List ADT**
   * The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.
   * The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.
   * The **List ADT Functions** is given below:
   * get() – Return an element from the list at any given position.
   * insert() – Insert an element at any position of the list.
   * remove() – Remove the first occurrence of any element from a non-empty list.
   * removeAt() – Remove the element at a specified location from a non-empty list.
   * replace() – Replace an element at any position by another element.
   * size() – Return the number of elements in the list.
   * isEmpty() – Return true if the list is empty, otherwise return false.
   * isFull() – Return true if the list is full, otherwise return false.
2. **Stack ADT**
   * In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
   * The program allocates memory for the *data* and *address* is passed to the stack ADT.
   * The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
   * The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.
   * push() – Insert an element at one end of the stack called top.
   * pop() – Remove and return the element at the top of the stack, if it is not empty.
   * peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
   * size() – Return the number of elements in the stack.
   * isEmpty() – Return true if the stack is empty, otherwise return false.
   * isFull() – Return true if the stack is full, otherwise return false.
3. **Queue ADT**
   * The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
   * Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program’s responsibility is to allocate memory for storing the data.
   * enqueue() – Insert an element at the end of the queue.
   * dequeue() – Remove and return the first element of the queue, if the queue is not empty.
   * peek() – Return the element of the queue without removing it, if the queue is not empty.
   * size() – Return the number of elements in the queue.
   * isEmpty() – Return true if the queue is empty, otherwise return false.
   * isFull() – Return true if the queue is full, otherwise return false.

**Features of ADT:**

* **Abstraction:**The user does not need to know the implementation of the data structure.
* **Better Conceptualization:**ADT gives us a better conceptualization of the real world.
* **Robust:**The program is robust and has the ability to catch errors.

From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.