Lecture

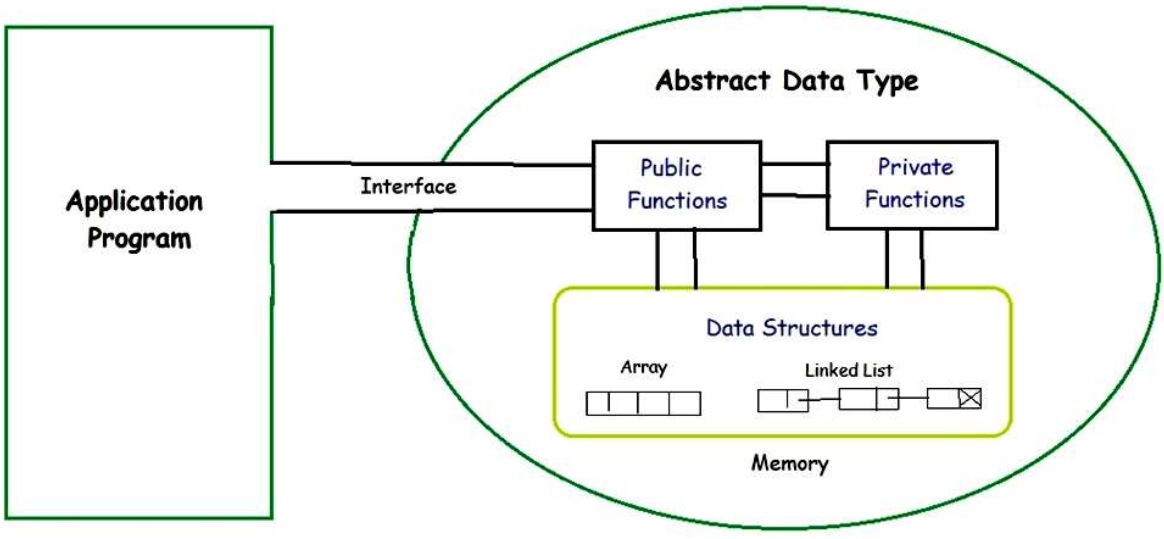
NOTE: FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

Introduction to Data Structures

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.
A data type is just a class of values that can be concretely constructed and represented. Data type can't be reduced anymore, while a data structure can, as it consists of multiple fields of different data.

Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value 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.



The user of data type 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.

Types of Data Structures

The data structures are divided into several categories based on their properties and behavior. A data structure can belong to multiple categories. These categories are divided into following:

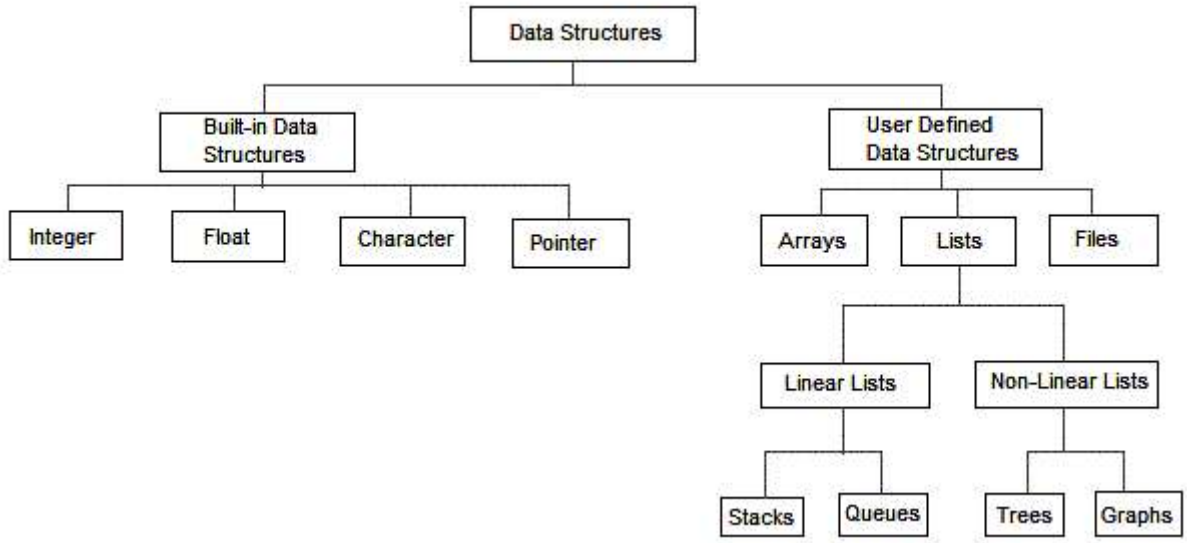
1. Primitive and Non-Primitive

As we 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 Non-Primitive (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.



2. Linear and Non-Linear

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Data elements in a liner data structure are traversed one after the other and only one element can be directly reached while traversing. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues. An arrays is a collection of data elements where each element could be identified using an index. A linked list is a sequence of nodes, where each node is made up of a data element and a reference to the next node in the sequence. A stack is actually a list where data elements can only be added or removed from the top of

the list. A queue is also a list, where data elements can be added from one end of the list and removed from the other end of the list.

In nonlinear data structures, data elements are not organized in a sequential fashion. A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run. Data structures like multidimensional arrays, trees and graphs are some examples of widely used nonlinear data structures. A multidimensional array is simply a collection of one-dimensional arrays. A tree is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements. A graph is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

3. Static and Dynamic

With a static data structure, the size of the structure is fixed.

Static data structures are very good for storing a well-defined number of data items.

For example a programmer might be coding an 'Undo' function where the last 10 user actions are kept in case they want to undo their actions. In this case the maximum allowed is 10 steps and so he decides to form a 10 item data structure.

There are many situations where the number of items to be stored is not known before hand.

In this case the programmer will consider using a dynamic data structure. This means the data structure is allowed to grow and shrink as the demand for storage arises. The programmer should also set a maximum size to help avoid memory collisions.

For example a programmer coding a print spooler will have to maintain a data structure to store print jobs, but he cannot know before hand how many jobs there will be.

4. Sequential and Random Access

Sequential access means that a group of elements (such as data in a memory array or a disk file or on magnetic tape data storage) is accessed in a predetermined, ordered sequence. Sequential access is sometimes the only way of accessing the data, for example if it is on a tape.

In data structures, a data structure is said to have sequential access if one can only visit the values it contains in one particular order. The canonical example is the linked list. Indexing into a list that has sequential access requires O(k) time, where k is the index. As a result, many algorithms such as quicksort and binary search degenerate into bad algorithms that are even less efficient than their naïve alternatives; these algorithms are impractical without random access.

Random access (more precisely and more generally called direct access) is the ability to access an item of data at any given coordinates in a population of addressable elements. As a rule the assumption is that each element can be accessed roughly as easily and efficiently as any other, no matter how many elements may be in the set, nor how many coordinates may be available for addressing the data.

In data structures, direct access implies the ability to access any entry in a list in constant time (independent of its position in the list and of list's size). Very few data structures can guarantee this, other than arrays (and related structures like dynamic arrays). Direct

access is required, or at least valuable, in many algorithms such as binary search and integer sorting.

5. Ephemeral and Persistent

The distinction is best explained in terms of the logical future of a value.
Whenever a value of a data structure is created it may be subsequently acted upon by the operations of the type. Each of these operations may yield (other) values of that abstract type, which may themselves be handed off to further operations of the type. Ultimately a value of some other type, say a string or an integer, is obtained as an observable outcome of the succession of operations on the abstract value. The sequence of operations performed on a value of an abstract type constitutes a logical future of that type --- a computation that starts with that value and ends with a value of some observable type. We say that a type is ephemeral if every value of that type has at most one logical future, which is to say that it is handed off from one operation of the type to another until an observable value is obtained from it. This is the normal case in familiar imperative programming languages because in such languages the operations of an abstract type destructively modify the value upon which they operate; its original state is irretrievably lost by the performance of an operation.
In contrast, values of a data structure in functional languages such as ML may have many different logical futures, precisely because the operations do not "destroy" the value upon which they operate, but rather create fresh values of that type to yield as results. Such values are said to be persistent because they persist after application of an operation of the type, and in fact may serve as arguments to further operations of that type.