Data Structure

In computer science, a **data structure** is a **data** organization, management, and storage format that enables efficient access and modification.

More precisely, a **data structure** is a collection of **data** values, the relationships among them, and the functions or operations that can be applied to the **data**.

ADT

Data structures serve as the basis for **abstract data types** (ADT).

- **The ADT defines the logical form of the data type.**
- **The data structure implements the physical form of the data type.**

Data Types

A data type is an attribute of data which tells the compiler (or interpreter) how the programmer intends to use the data.

- **Primitive**: basic building block (boolean, integer, float, char etc.)
- **Composite**: any data type (struct, array, string etc.) composed of primitives or composite types.
- **Abstract**: data type that is defined by its behaviour (tuple, set, stack, queue, graph etc).

If we consider a composite type, such as a 'string', it ***describes*** a data structure which contains a sequence of char primitives (characters), and as such is referred to as being a 'composite' type.

Whereas the underlying ***implementation*** of the string composite type is typically implemented using an array data structure.

Note: in a language like C the length of the string's underlying array will be the number of characters in the string followed by a 'null terminator'.

An abstract data type (ADT) describes the expected ***behaviour*** associated with a concrete data structure.

For example, a ‘list’ is an abstract data type which represents a countable number of ordered values, but again the ***implementation*** of such a data type could be implemented using a variety of different data structures, one being a ‘linked list’.

Note: an ADT describes behaviour from the perspective of a consumer of that type (e.g. it describes certain operations that can be performed on the data itself).
For example, a list data type can be considered a sequence of values and so one available operation/behaviour would be that it must be iterable.

Data Structures

A data structure is a collection of data type ‘values’ which are stored and organized in such a way that it allows for efficient access and modification.

In some cases a data structure can become the underlying implementation for a particular data type.

For example, composite data types are data structures that are composed of primitive data types and/or other composite types, whereas an abstract data type will define a set of behaviours (almost like an ‘interface’ in a sense) for which a particular data structure can be used as the concrete implementation for that data type.

Uses of Data Structure:

Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.

- For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.
- Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services.
- Usually, efficient data structures are key to designing efficient algorithms.

- Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.
- Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

Implementation

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by a pointer—a bit string, representing a memory address, that can be itself stored in memory and manipulated by the program.

Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself.

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure.

The efficiency of a data structure cannot be analyzed separately from those operations.

This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

Examples

There are numerous types of data structures, generally built upon simpler primitive data types:

When we think of data structures, there are generally four forms:

1. **Linear**: arrays, lists
2. **Tree**: binary, heaps, space partitioning etc.
3. **Hash**: distributed hash table, hash tree etc.
4. **Graphs**: decision, directed, acyclic etc.

Array:

- An *array* is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type).
- Elements are accessed using an integer index to specify which element is required.
- Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity).
- Arrays may be fixed-length or resizable.

Linked List:

- A *linked list* (also just called *list*) is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list.
- The principal advantage of a linked list over an array is that values can always be efficiently inserted and removed without relocating the rest of the list.
- Certain other operations, such as random access to a certain element, are however slower on lists than on arrays.

Record:

- A *record* (also called *tuple* or *struct*) is an aggregate data structure.
- A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names.
- The elements of records are usually called *fields* or *members*.

Union:

- A *union* is a data structure that specifies which of a number of permitted primitive types may be stored in its instances, e.g. *float* or *long integer*.
- Contrast with a record, which could be defined to contain a float *and* an integer; whereas in a union, there is only one value at a time.
- Enough space is allocated to contain the widest member datatype.

- A *tagged union* (also called *variant*, *variant record*, *discriminated union*, or *disjoint union*) contains an additional field indicating its current type, for enhanced type safety.
- An *object* is a data structure that contains data fields, like a record does, as well as various methods which operate on the data contents.
- An object is an in-memory instance of a class from a taxonomy.
- In the context of object-oriented programming, records are known as plain old data structures to distinguish them from objects.

In addition, *graphs* and *binary trees* are other commonly used data structures.

Language support:

- Most assembly languages and some low-level languages, such as BCPL (Basic Combined Programming Language), lack built-in support for data structures.
- On the other hand, many high-level programming languages and some higher-level assembly languages, such as MASM, have special syntax or other built-in support for certain data structures, such as records and arrays.
- For example, the C (a direct descendant of BCPL) and Pascal languages support structs and records, respectively, in addition to vectors (one-dimensional arrays) and multi-dimensional arrays.
- Most programming languages feature some sort of library mechanism that allows data structure implementations to be reused by different programs.
- Modern languages usually come with standard libraries that implement the most common data structures. Examples are the C++ Standard Template Library, the Java Collections Framework, and the Microsoft .NET Framework.
- Modern languages also generally support modular programming, the separation between the interface of a library module and its implementation.

- Some provide opaque data types that allow clients to hide implementation details. Object-oriented programming languages, such as C++, Java, and Smalltalk, typically use classes for this purpose.
- Many known data structures have concurrent versions which allow multiple computing threads to access a single concrete instance of a data structure simultaneously.