TEMASEK JUNIOR COLLEGE

Temasek Junior College

2022 JC1 H2 Computing

Data Structures 1 – Static and Dynamic Data Structures

§1.1 Atomic Data Types (Recapitulation)

Almost all programs process data in some form, and data needs to be stored in such a way that it can be easily manipulated by program code. We have thus far looked at how individual pieces of data can be stored as **atomic data types**. For example, your age can be stored as an integer, your name as a string and your birthday as a date. To recapitulate, the **atomic data types** that you will encounter in H2 Computing 9569 are:

Atomic Data Type	Description	Pseudocode Keyword	Python Keyword
Character	Single character such as single letter, digit, symbol or control code. Example pseudocode Delimited by single quotes e.g. `'x', 'C', '@'`	`CHAR`	–
String	Sequence of zero or more characters. Example pseudocode Delimited by double quotes. e.g. `"Hello World"` String with no characters is an empty string. It can be written as `""`	`STRING`	`str`
Integer	A whole number with no decimal (fractional) part. Example pseudocode Written as a denary number e.g. `5, -3`	`INTEGER`	`int`
Real	A number containing a decimal (fractional) part. Example pseudocode Written with at least one digit on either side of the decimal point, e.g. `4.712, 0.3, -5.03` Zeros are added if necessary e.g. `-4.0, 0.0`	`REAL`	`float`
Boolean	Logical values `TRUE` and `FALSE`.	`BOOLEAN`	`bool`
Date	Valid calendar date. Example pseudocode Normally written as `dd/mm/yyyy` Good to state that the value is of data type `DATE` and to explain the format as dates are represented differently across the world	`DATE`	`class datetime`

§1.2 Data Structures and Abstract Data Types (ADTs)

Atomic data types such as an integer or a Boolean allow you to store a single value. It is possible to combine these primitive data types to create **compound data types** such as a record, or to create more complex **data structures** such as arrays.

In H2 Computing 9569, we shall progress beyond storing individual pieces of data, to a more advanced level, where we look at methods of storing larger volumes of data in formats that make it easy for programs and users to access and analyse.

Abstract Data Type (ADT)
An **abstract data type (ADT)** is a conceptual model of how data can be stored and the operations that can be carried out on the data.

Data Structure
A **data structure** is a common storage format for large volumes of related data in an organised an accessible format, which enables different programs to manipulate the stored data in different ways. It is an implementation of an **abstract data type** in a particular programming language.

Different data structures tend to lend themselves to different types of applications. For example, a text file may be suitable for a database whereas a stack is suitable for handling exceptions.

At a more advanced level, **static** and **dynamic data structures**, together with **abstract data types**, allow more sophisticated organisation and manipulation of data, and provide methods for adding, removing, and traversing the data.

§1.3 Static and Dynamic Data Structures

More often than not, the amount of data stored within a data structure will vary while the program is being run. Depending on the volume of data that needs to be stored and therefore the memory required, data can be stored in either **static** or **dynamic** data structures.

Static Data Structure

A **static data structure** is a format that stores a set amount of data. This is done by pre-allocating a set amount of memory to the data structure prior to program execution (at compile time), which is a form of **static allocation of memory**.

Accessing individual elements of data within a static data structure is very quick as their memory location is fixed.

However, the data structure will still take up all the pre-allocated memory even if the volume of data stored does not require that large a memory.

Examples: records and some arrays (Though in modern day programming, it is possible for an array to be dynamic data structure instead of being a static data structure.)

Dynamic Data Structure

A **dynamic data structure** can use different amounts of memory depending on the volume of data to be stored. This is achieved through a **heap**.

A **heap** is a pool of unused memory that can be allocated to (and deallocated from) a dynamic data structure depending on the amount of memory required to store data within the dynamic data structure.

Unused blocks of memory are placed on a heap, which are then usable within a program. During program execution, a dynamic data structure is able to take more memory from the heap when additional memory is required to store data. Blocks of unused memory are returned to the heap when excess memory is no longer required to store data. This is a form of **dynamic allocation of memory**.

The result is a more efficient use of memory resources and a more flexible data storage solution as elements can be added and removed much more easily.

Examples: stacks and queues (Stacks and queues are often implemented as dynamic structures. Nevertheless, it is possible for them to be implemented as static data structures as well. We will do both types of implementation in subsequent lessons.)

Static Data Structures vs. Dynamic Data Structures

The table below provides a comparison between static and dynamic data structures:

Category	Static Data Structures	Dynamic Data Structures
Resource management	Inefficient use of memory resources as memory allocated might be more than what is needed.	Efficient use of memory resources as the amount of memory allocated can vary based on what is needed.
	The memory allocation is fixed and usually sufficient for use by data structure. Hence there will be no memory overflow.[1]	Overflow might occur if the data structure becomes too large and the heap does not contain sufficient memory for allocation. Underflow may also occur if the data structure becomes empty.
Efficiency in accessing data	Fast access to each data element as memory location is fixed when program is written (or at compile time before execution).	Slower access to each data element as memory location is only allocated at runtime during execution.
	Memory addresses allocated will be contiguous (i.e. "adjacent" to one another). Hence access is faster.	Memory addresses allocated may be fragmented and non-contiguous. Hence access is slower.
Data management	Data structures are fixed in size, making them more predictable to work with. E.g. they can contain a header.	Data structures vary in size so there needs to be a mechanism for knowing the size of the current structure.
	Relationship between different data elements does not change .	Data can be added or removed from the data structure. Hence relationship between data elements might change as the program runs.
Programming	Easier to program as there is no need to check on data structure size at any point.	Harder to program as there is a need to always keep track of data structure size and addresses of locations of data elements.

[1] The is still a small possibility of overflow if insufficient memory is allocated for the data that is to be stored to the data structure.