Data Structures and Algorithms

Alan Sexton

(with thanks to Paul Levy, Tomáš Jakl,
Eliseo Ferrante and Anis Zarrad)

Autumn 2020

Programs = Algorithms + Data Structures

Data Structures efficiently organise data in computer memory.

Algorithms manipulate data structures to achieve a given goal.

In order for a program to terminate fast (or in time), it has to use *appropriate* data structures and efficient algorithms.

In this module we focus on:

- various data structures
- basic algorithms
- understanding the strengths and weaknesses of those, in terms of their time and space complexities

Learning outcome

After completing this module, you should be able to:

- Design and implement data structures and algorithms
- Argue that algorithms are correct, and derive time and space complexity measures
- Explain and apply data structures in solving programming problems
- Make informed choices between alternative data structures, algorithms and implementations, justifying choices on grounds such as computational efficiency

Abstract data types (ADT)

An *abstract data type* is

- a type
- with associated operations
- whose representation is hidden to the user

Example
Integers are an abstract data type with operations `+`, `-`, `*`, `mod`, `div`, ...

- A type is a collection of values, e.g. integers, Boolean values (true and false).
- The operations on ADT might come with mathematically specified constraints, for example on the time complexity of the operations.
- Advantages of ADT's as explained by Aho, Hopcroft and Ullman (1983):

  > *"At first, it may seem tedious writing procedures to govern all accesses to the underlying structures. However, if we discipline ourselves to writing programs in terms of the operations for manipulating abstract data types rather than making use of particular implementations details, then we can modify programs more readily by reimplementing the operations rather than searching all programs for places where we have made accesses to the underlying data structures. This flexibility can be particularly important in large software efforts, and the reader should not judge the concept by the necessarily tiny examples found in this book."*

List is an ADT

An example of a list of numbers

$$\langle 2, 5, 1, 8, 23, 1 \rangle \quad \text{(ordered collection of elements)}$$

List is an ADT; list operations are:

- insert an entry (on a certain position)
- delete an entry
- access data by position
- search
- concatenate two lists
- sort
- ...

Different representations of lists

Depending on what operations are needed for our application, we choose from different data structures (some implement certain operations faster than the others):

- Arrays.
- Linked lists.
- Dynamic arrays.
- Unrolled linked lists.

(Simplified) memory model

To demonstrate how memory management in an Operating System (OS) works we are going to treat memory as a gigantic array

```
Mem[-]
```

(for simplicity we assume that every entry can contain either an integer or a string).

We defined a OS-level pseudo-code language and we call it **OS++**

Two operations provided by OS++:

- `allocate_memory(n)` – the OS finds a continuous segment of `n` unused locations, designates that memory as used, and returns the address of the first location.
- `free_memory(address,n)` – the OS designates the `n` locations starting from `address` as free.

This is a simplification from what is in Java, C and other languages. In reality, strings, for example, are stored as blocks of items in `Mem` (i.e. they occupy more than just one location).