Data Structures and Algorithms

Alan Sexton[1]

[1]Thanks to many staff from the School of Computer Science, Uni of Birmingham, UK

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 Outcomes

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 for them
- 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)

A *type* is

- a set of possible values
- with a set of allowed operations on those values

An *abstract data type (ADT)* is a type whose internal representation is hidden to the user.

Thus users of an abstract data type may have no information about how the ADT is implemented, but depends only on the published information about how it behaves.

This means that the implementation of an abstract type can be changed without having to change the code that uses it.

Example
"Integer" is an abstract data type consisting of integer values with operations `+`, `-`, `*`, `mod`, `div`, ...

- A type is a collection of values, e.g. integers, Boolean values (true and false) with their operations.
- The operations on an 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 may include:

- 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
- . . .

We will study some of these in detail later in the module.

Computer Memory to a Program

Data is stored, managed and manipulated in computer memory.

- The computer architecture (the hardware) and the operating system work together to allow each running program to see an illusion that they have all the memory on the computer to themselves.
- This memory is organised as a long list of memory cells, each 1 byte or 8 bits large. A bit can hold either a zero or a one.
- Every one of these 1 byte cells has an address, a number in the range 0 to the highest possible address for this system combination
- On older, 32 bit systems, this highest address is hexadecimal FFFFFFFF (i.e. 8 'F's), or 4,294,967,295 (i.e. $2^{32} - 1$)
- On newer, 64 bit systems, this is FFFFFFFFFFFFFFFF (i.e. 16 'F's), or approximately $1.844674407 \times 10^{19}$ (i.e. $2^{64} - 1$)
- For technical reasons, the hardware actually manipulates memory a whole word at a time, where is word is either 4 bytes (32 bits) or 8 bytes (64 bits)

Memory Management

Of course, computers can not hold such gigantic amounts of memory, and if a program tried to access every one of those bytes of memory, it would reach addresses in memory where the illusion breaks down, and an error would be triggered.

To support the illusion of all this memory for a program, programs have to explicitly request the operating system to add large sections of memory to their memory space via *operating system calls*

However, programs themselves need to manage their own memory with much finer and more efficient control than these expensive operating system calls

So the *Runtime System*, a software library that is integral to each programming language, typically provides one of two options

1. Explicit Memory Management, with *allocate* and *free* or
2. Implicit Memory Management with Garbage Collection

Explicit Memory Management

The programming languages C and C++ are examples of programming languages with explicit memory management.

C's runtime system maintains it's own data structure to organise the memory it has obtained from the operating system. This data structure is usually accessed by two functions:

1. *malloc*: allows the program to request a contiguous amount of memory. If available, this is recorded in the memory managment data structure, the address of the start of the block is returned to the program and the data structure is updated accordingly.

   If not available, a system call requests more memory from the operating system. If successful, it integrates the memory into its data structure and continues as before. If unsuccessful, it reports an error back to the program.
2. *free*: Given the address of a block that was previously allocated, marks it as available for future allocation

Explicit Memory Management

Explicit Memory Management is a simple model and very efficient. However, it suffers from a number of disadvantages:

- The program needs to keep track of every block requested with *malloc* so that it can be freed later. Otherwise, the program will have *memory leaks* — these are blocks of allocated memory that can no longer be used or freed. This causes the program to use more memory than it needs (which causes performance problems), and may cause the program to crash if it eventually runs out of memory.
- If an error is made in giving an address to *free* that had not previously been returned by *malloc*, or if that address had already been freed, then the memory management data structure can be corrupted, causing unpredictable errors or crashes

Implicit Memory Management

Implicit Memory Management is much more sophisticated. The programming language itself, through its runtime system, provides mechanisms to allocate data structures (without exposing memory addresses to programmers), and identifies allocated structures that can no longer be accessed by the running program and adds them back into its data structure of available free memory without requiring the program to specifically ask for the memory to be freed.

This mechanism is used by Java, Python, Functional programming languages like Haskell and OCaml, and many more.

It relieves the heavy burden of keeping track of allocated memory from the programmer, and avoids most of the bugs and problems that address manipulation in languages like C are famous for.

Pseudocode

In this module we will be discussing details of algorithms. This necessitates using a programming language to capture the precise steps of the processing involved. However, standard programming languages require a lot of administrative detail that is necessary to get the program working on the computer but ends up obscuring the important aspects of the algorithm we wish to explain.

Instead we will use a simplified, less rigorous style of a programming language that will not run on any computer but makes clear to a human reader the steps involved. This is called *pseudocode*.

Throughout this module, when you are asked to write some pseudocode, you are free to use pseudocode, Java code or any mix between the two. In such cases any syntactic errors will not be penalised so long as the intent is clear to a human reader.

Pseudocode

The core elements of a programming language like Java are

- Variables: named memory cells that can hold values of some type, e.g. Integers, Strings, Arrays, etc.
- Expressions: how to calculate new values
- Assignment statements: how to modify variables
- Sequences and Blocks: how to execute a sequence of steps and group them
- Conditionals: How to choose between different steps
- Loops: how to run steps multiple times
- Input/Output: How to write out or read in values
- Function definitions: How to combine steps into a function that can be executed from multiple locations in the code
- Function calls: How to invoke a function

Pseudocode: Variables, Expressions

- For variables we use simple names with no spaces that start with a letter and can contain letters, digits and the underscore character "_". Examples include
    - "total", "name", "length", "account_holder_2"
- Expressions can be any meaningful (to humans) mathematical, logical or text expression using:
    - Arithmetic operations: $(+,\ -,\ \times, \div, *, /$ etc.)
        - (sum1 + sum2) / 2
    - Logical operations: (AND, OR, NOT, etc)
        - isRaining AND haveUmbrella
        - balance $>$ 100 AND dayOfMonth $==$ 1
    - Strings can be joined together using "+" as a string concatenation operator
        - "hello " + name
    - Extra operators from Java can be used.

Pseudocode: Assignments

- Assignments put values (possibly calculated by expressions, into variables using a single "=" symbol
    - n = (25 + n) MOD 50
    - isRaining = TRUE
    - message = "Hello " + name
- When you first use a variable, you can specify its type if it is not clear from the context:
    - float sum = 0
- You should not use different types in the same variable at different times
- You should never use a variable in an expression whose value has not first been set

Pseudocode: Sequences and Blocks of Statements

- To specify a sequence of statements, you can put them one per line, indented to the same level. Optionally, you can put a semicolon, ";", at the end of each line:

  -
    ```
    a = a+10
    b = a*2-4
    ```
- To group them into a block, e.g. to be able to loop over them, you can surround them with braces, "{", "}", or with BEGIN, END:

  -
    ```
    BEGIN
        a = a+10
        b = a*2-4
    END
    ```
- **Always** indent the contents of the block correctly

Pseudocode: Conditional Statements

- Here we use one of the forms (always separated onto different lines and indented:
  - IF condition THEN statement ENDIF
    - IF balance < 0 THEN

      ```
          print("Your balance is overdrawn")
          print("You owe £", -balance )
      ENDIF
      ```
  - IF condition THEN statement1 ELSE statement2 ENDIF
  - IF condition1 THEN statement1 ELIF condition2 THEN statement2 ... ELSE statementN ENDIF
- Alternatively you can use a more Java-like syntax:
  - if (balance < 0)

    ```
    {
        print("your balance is overdrawn")
        print("You owe £", -balance )
    }
    ```

Pseudocode: Loops

- There are a number of forms for loops
  - WHILE condition DO statement ENDWHILE
  - DO statement WHILE condition
  - REPEAT statement UNTIL condition
  - FOR (initial statement; condition ; step) statement ENDFOR

    -
      ```
      sum = 0
      FOR (i = 0 ; i < 100 ; i=i+1)
          sum = sum + i
      ENDFOR
      ```
- These all have Java-like versions:

  -
    ```
    sum = 0
    for (i = 0 ; i < 100 ; i=i+1)
    {
        sum = sum + i
    }
    ```

Pseudocode: Input/Output

- To output, use the function “print”, with as many arguments as you need
  - print(”answer is: ”, sum)
- To input, use the function “read”, assigning the return value to a variable. There is no need to specify precisely the details of the input format. The intention should be clear from the surrounding code and the name of the variable.
  - numberOfStudents = read()

Pseudocode: Functions

- Function definitions specify a sequence of statements to be executed. Values can be passed in to the function by parameters, results can be returned from the function using a RETURN statement. You should specify the type of value that the function returns, or specify the type as VOID if it does not return anything (e.g. if it just prints values out).

  -
    ```
    int max(int a, int b)
    BEGIN
        IF a > b
            RETURN a
        RETURN b
    END
    ```

  -
    ```
    int max(int a, int b)
    {
        if (a > b)
            return a
        return b
    }
    ```

- Such a function can be called as follows:
  - maximum = max(myValue, 10)