7SENG010W Data Structres & Algorithms

Week 1 Lecture

Introduction

Overview of Week 1 Lecture: Introduction

Aim is to introduce the main themes of the module:

- *Data Structures*
  - Categories of data structures
  - Examples: arrays, stacks, queues, trees & graphs
- *Algorithms*
  - Types of Algorithms
- *Analysis of Algorithms*
  - Complexity of an Algorithm, its *"Big-O"*
- *.NET Framework – Collections Library*
  - C♯ Collection classes

PART I

Introduction to Data Structures

What is a Data Structure?

- A *data structure* is a *structure* used to organise & store a *collection of one or more items of information* that are usually related in some way.
- For example, a *student's record* containing: student number, personal details, course title, year of entry, list of modules & marks, etc.
- Programs have to store *collections of data items* in order to process them, this is why we need data structures.
- Programming languages provide some *basic primitive data structures* as part of the language, e.g. integers, characters, booleans, arrays, etc.
- A language's available data structures may be satisfactory for the needs of a particular program.
- If they are not sufficient then the programmer needs to be able to construct new data structures to meet her/his needs.
- In OO languages this is done using the "*class*" construct that allows programmers to *construct new data types & structures*, & provides controlled access to the data structure via the class's methods.

Aspects of a Data Structure

The problem of representing & manipulating a given data structure within a computer has three aspects:

- *Formal definition:* of the data structure & the rules controlling its use.

  This is the *Abstract* representation & is usually defined using a mathematical notation.

  Virtually all programming language data structures are based on & are computer representations of mathematical structures, e.g. *sequences* – arrays, lists; trees; graphs; etc.

- *Operations:* that are appropriate to be performed on the data structure, & the facilities available in the particular programming language being used.

  This is the *operational* view.

- *Method of storage:* how the elements of the structure are kept in the sequentially addressed computer's storage.

  This is the *Internal* representation.

Examples & Visualisations of Data Structures

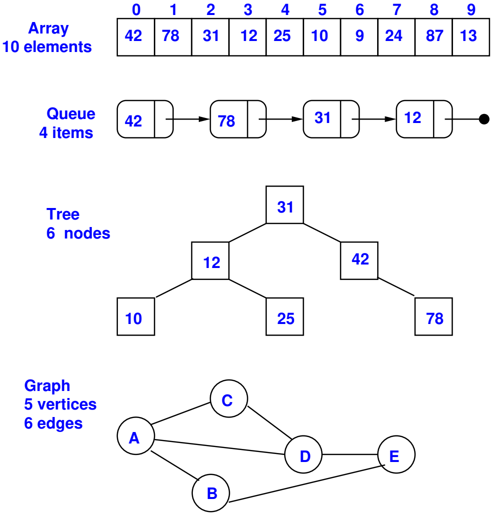

Figure : 1.1 Examples of Data Structures

Types of Data Structures

- *Linear* data structures:
  - A collection of data items organised using a *sequence* structure.
  - Examples: Arrays, List, Stacks, Queues.
- *Non-Linear* data structures:
  - A collection of data items organised using a *non-sequential* structure, e.g. tree like, as a graph.
  - Examples: Trees, Heaps, Graphs.
- *Static* data structures:
  - A collection of data items that has *a fixed size*, usually *sequential*.
  - Examples: Arrays, Stacks, Queues.
- *Dynamic* data structures:
  - a collection of data items that *does not have a fixed size*,
  - allows *new data items to be added* to it, & *existing data items to be deleted* from it.
  - Examples: List, Stacks, Queues, Trees, Heaps, Graphs.

Types of Data Structures (Cont)

- *Indexed* data structures:
  - A collection of data items accessed via one or more *indexes* into the structure.
  - Examples: Arrays, Matrices (2-dimensional Arrays), n-dimensional Arrays, etc.
  - E.g. Arrays – `numbers[5]`, Matrices – `matrix[5,6]`.
- *Linked (Non-indexed)* data structures:
  - A collection of data items in a *structure* made up of "*data nodes*" connected by "*links*" to other nodes.
  - How the nodes are linked determines the type of data structure.
  - Accessed by following the links between the nodes.
  - Examples: List, Stacks, Queues, Trees, Graphs.

Static data structures & Memory management

- Arrays are static which means that the compiler allocates a *fixed amount of memory space* for storage of the array.

- It is possible to use arrays to store data structures such as stacks, queues, lists, etc.

- Even more complex data structures such as trees & graphs can be represented using arrays.

- However, such a static implementation has **disadvantages**:
  1. **Size of an array is fixed**, so consequently the maximum number of items it can store is fixed.
     This is a significant disadvantage since the maximum size of the data structure may not be known in advance.
  2. Compiler allocates memory locations to accommodate all elements of the array, but **when not full memory space is wasted**.
  3. **Insertion or deletion operations can be inefficient**.
     E.g. inserting into a full array, need to create a bigger array & copy in existing elements into the new array; deletions leave “gaps”.

Dynamic data structures & Memory management

- *Dynamic data structures* (e.g. queues, trees, etc) are used when the amount of data that has to be dealt with is unknown at compile time.
- "*Dynamic*" means that the data structure does not have a fixed size, but has the facility to "*grow*" or "*shrink*" in size, i.e. add or delete data items.
- Dynamic data structures usually start in an "*empty*" state containing no data items, & then additional data items are *added* as required.

  E.g. create an *empty queue* of items & then adding new items to the end of the queue as needed.
- This *dynamic* aspect of a data structure requires *dynamic memory management*, to support their ability to grow by adding new items.
- Managed by the program's *run-time system* that dynamically allocates additional memory locations to store the new data items.
- In OO programming languages done using "**`new`**" to create a new object.
- The new object is referred to using a "*pointer*" to its memory location, either implicitly as a "*reference*" in C♯ & Java, or explicitly in C++.

Abstract Data Types (ADTs)

- An *abstract data type* (ADT) is a data type together with a set of operations which define how the type may be manipulated.
- So it is **not just** a *set of values* alone.
- **But** it is a *combination of two things*:
  - a *set of values* that are *all of the same "type"*,
  - **together with** a *collection of operations* that access, modify & manipulate the elements in the *set of values*.
- Example: the *Integers* – the negative & positive whole numbers,
  - *set of values*: $\ldots, -3, \ -2, \ -1, \ 0, \ 1, \ 2, \ 3, \ \ldots$
  - *collection of operations*:
    Monadic Operators: $pred, \ succ, \ \ldots$
    Binary Operators: $+, \ -, \ *, \ \div, \ \%, \ \ldots$
    Relations: $=, \ <, \ >, \ \leq, \ \geq, \ \ldots$

Implementing Abstract Data Types (ADTs)

- In OO programming languages like C♯, abstract data types can be implemented via the *class* construct, where the:
  - *instances of the class* i.e. objects, are the values of the ADT.
  - the *class's public methods* are the operations on the ADT.

- Key elements of using a class to implement an ADT are:
  - The class's *data members* are used to represent a value of the ADT.
  - An essential aspect of a "*secure*" ADT is that its internal representation is *hidden*, this is called *data encapsulation* or *data hiding*.
  - Achieved by making all data members either `private` or `protected`.
  - This internal representation of the ADT should only be accessible from outside the class by using the ADT's "*interface*", i.e. its `public` methods.
  - The ADT's *interface* should include methods that implement all of the operations associated with the ADT that are required for its use.
  - Approach known as "*data abstraction*" – the ADT's users are only concerned with the data objects (e.g. Integers) & their operations, without knowing or caring about the details of how they are implemented.

Advantages of using ADTs

- Use of ADTs allows the programmer to be provided with a type & a set of operations that can be performed on objects of that type without needing to know how it is implemented.
- Implementing a *data type* as an ADT allows the programmer to control access to the internal data structure by using *data encapsulation* & its *interface*.
- Results in *safer* & more *robust* software development since:
  - Data encapsulation guarantees greater integrity of the values of an ADT.
  - So values are protected from inadvertent corruption, as the user cannot directly access the types representation.
  - Access to the type is strictly confined to the operations provided in its interface.
- Easier *software maintenance* since the implementation of the ADT can be changed without changing its *interface*, programs which use the type via its interface will remain unchanged.

PART II

Introduction to Algorithms

Why are Algorithms important?

Today many/most things are run & controlled by computers.

Internet: Web searching, data packet routing, data sharing, ...

Science: Gene mapping, protein folding, Hadron Collider particle collision simulation, space flights, asteroid path prediction, climate modelling, ...

Computers: Chip layout, Software design, Operating systems, applications, ...

Cyber Security: IoT, Mobile phones, Banking, e-Commerce, Hacking detection & prevention, ...

Multimedia: streaming, creating, editing & decoding MP4, MP3, JPG, ...

Social networks: Recommendations, News feeds, Targeted Advertising, ...

Transport: Road vehicles, Driverless trains & metro, Plane's auto-pilot, ...

This can happen because the computers are executing *algorithms* that have been designed to do these tasks, (to varying degrees of success).

In the future this control by computers & their algorithms is only likely to expand, e.g. IoT, driver-less cars, robots, etc.

What is an Algorithm?

- An algorithm in its most general sense is *a sequence of steps/instructions designed to solve a problem or achieve a goal*.
- Examples: cooking recipes, directions to travel from A to B, instructions for assembling a piece of furniture, a *software program*, etc.
- We are just concerned with the software sense, i.e. program instructions (code) written in either a *programming language* or *pseudo code*.
- Data Structures & Algorithms are intrinsically linked & form the essential components of programming & software development.
- In programs it is often necessary to structure data in a particular way to enable it to be processed conveniently & efficiently by an algorithm.
- Also the choice of data structure (array, list, tree, etc) generally determines the type of algorithm to use, & vice versa.
- Generally the more complex the data structure the more complex its associated algorithms, e.g. arrays vs. graphs.

A Simple Algorithm: Search an Array for location of a value

```
namespace Lecture_1
{
  class ArraySearch
  {
     const int NOT_FOUND = -1 ; // NOT a valid array index

     public static int Search( int[] array, int value )
     {
        for (int i = 0; i < array.Length; i++)
        {
            if ( value == array[i] ) {
                 return i ;            // found value at index i
            }
        }
        return NOT_FOUND ;
     }

     static void Main( string[] args )
     {
       int[] numbers = { 42, 78, 31, 12, 25, 10, 9, 24, 87, 13 } ;
       int value     = 13 ;
       int index     = NOT_FOUND ;

       index = Search( numbers, value ) ;
       System.Console.WriteLine( "Index of Value {0} is {1}",
                                 value, index ) ;
     }
  } // ArraySearch
} // Lecture_1
```

Picking your Algorithm

- When presented with a list of algorithms that solve a particular problem, then it is usual to choose the *"most efficient"* one or ones from the list.

- However, you may choose an algorithm that is *slower* than the other algorithms, because it is:
  - simpler to *understand*, e.g. uses `for`-loops, rather than *recursion*,
  - simpler to *implement*, e.g. uses arrays, rather than lists, graphs, etc.
  - an *implementation already exists & is available to use*, such as in a software library, e.g. .NET class libraries, Java's JDK, or C++'s Standard Template library.

- Algorithm Efficiency:
  - *Time*: how fast an algorithm runs for a given size of input data.
  - *Space*: how much extra storage space the algorithm requires.
  - There's often a trade-off between the two.

Types of Algorithms

There are a number of different *generalised approaches* that an algorithm can take to solving a problem.

Examples of the most common algorithm types are the following:

- *Brute-force*
- *Divide-and-conquer*
- *Greedy*

These algorithm types are used to classify algorithms.

The purpose of doing this is that it indicates the general characteristics & approach of an algorithm, & thus helps a programmer to decide whether it is appropriate for the programming task at hand.

We will encounter examples of these types of algorithms in the module.

Brute Force Algorithms

- *Brute force* algorithms are usually the most straightforward, least sophisticated & “obvious” algorithms to solve a problem.
- A brute force algorithm is a solution that is based directly on the problem definition.
- It is often easy to establish the correctness of a brute force algorithm.
- This algorithmic strategy applies to almost all problems.
- **Big Disadvantage**: except for generally quite basic problems, this algorithmic strategy produces algorithms that are **far too slow** at solving the problem.

  For some complex problems this approach may take several years, hundreds of years or even thousands of years to solve, e.g. graph problems.

Divide & Conquer Algorithms

- A problem is divided into several sub-problems of the same type, ideally of about equal size.
- Usually the sub-problems are solved using recursion.
- However, sometimes a different type of algorithm may be used, when sub-problems become small enough, i.e. a simpler (& quicker?) approach would work better.
- If necessary, the solutions to the sub-problems are "*combined*" in some way to get a solution to the original problem.
- Examples: Merge-Sort.

Greedy Algorithms

- The solution to a problem is constructed as a sequence of steps.
- Where there is *a choice of alternatives* for a given step, the essence of the greedy algorithm approach is that the *choice made should be locally optimal*, i.e. it has to be the best immediate choice among all feasible choices available at that step.
- In addition, the choice made *must be irrevocable*; once made, it **cannot be changed on subsequent steps** of the algorithm i.e. *no backtracking*.
- On each step, a *greedy choice* is made of the best alternative available in the hope that a sequence of locally optimal choices will lead to a (globally) optimal solution to the entire problem.
- Greedy algorithms **do not always lead to a solution** let alone, an optimal solution.
- Greedy algorithms often used to solve *optimisation problems*, i.e. find the "*best*" way of doing something, according to some criteria.
- Examples: Travelling Salesman Problem.

PART III

Analysis of Algorithms

Algorithm Analysis

- *Algorithm analysis* estimates the "*resource*" (e.g. computation/running time, storage space) consumption of an algorithm.
- This allows us to compare the *relative costs* of a group of algorithms for solving the same task, e.g. sorting or searching a collection of data.
- Algorithm analysis is a method for estimating whether a proposed solution for a task is likely to meet the resource constraints for a problem, e.g. complete the task within a specific time.
- Algorithm analysis measures the *efficiency of an algorithm* or its implementation as a *program* as the:

  "*size of the input data the algorithm or program is applied to increases*".
- Typical types of analysis are:
  - calculate the "*order of complexity*" of an algorithm to complete its task, or
  - estimate or measure the *running time* for a program to complete its task, or
  - estimate the storage space required for a data structure.

An Algorithm's – "amount of input data" – N

- We often investigate the *order of complexity* (efficiency) of an algorithm in terms of the "*amount of data*" the algorithm will be applied to.
- We denote the *amount of data* (or *size of the input data*) the algorithm will be applied to by the parameter N (or n); but N is not always obvious.
- Obvious case: an algorithm that sorts a list of N numbers, the input size is the count of numbers in the list – N.
- For some algorithms, several values may need to be combined to get the size of the input.
- Non-Obvious case: algorithms that operate on *graphs* are dependent on the *size of the graph*, but this is defined in terms of:
  - the number of *vertices* (nodes) in the graph, (Figure 1.1 graph – 5) &
  - the number of *edges* (arcs) in the graph, (Figure 1.1 graph – 6).

An Algorithm's – "running time" – $T(N)$

The "*running time of an algorithm or program*" is defined by $T(N)$, an expression in terms of the input size N.

Examples: $T(N) = \log_2(N),\ T(N) = 10N,\ T(N) = N^2 + 2N,\ T(N) = 2N^3$

Interested in the following aspects of the running time equation $T(N)$:

- Concept of "*growth rate*" is used to compare the $T(N)$ of algorithms without having to write the programs & run them on the same computer.
- Aim is to "*classify*" $T(N)$ in terms of its *"order of complexity*", i.e. *growth rate*.
- The *upper bound* for the running time of the algorithm $T(N)$ indicates the *upper* (or *highest*) *growth rate* that the algorithm can have.
- We measure this upper bound on the *worst-case*, *average-case* or *best-case* data inputs.
- Make statements like:
  "*This algorithm has an upper bound to its growth rate* $T(N)$ *of* N^2 *in the worst-case.*"

Graphs of Example $T(N)$ Equations

Figure 1.2, shows the graphs of five example *growth rate running time* equations $T(N)$ (y-axis), plotted against the *input size* N (x-axis).

They illustrate the significant differences between the five growth rate $T(n)$ equations even for very small values of $n \leq 50$.

A *steeper gradient* means a longer running time, i.e. a higher upper bound for the algorithm, e.g. $T(n) = 2^n$ is much higher then $T(n) = 20n$.

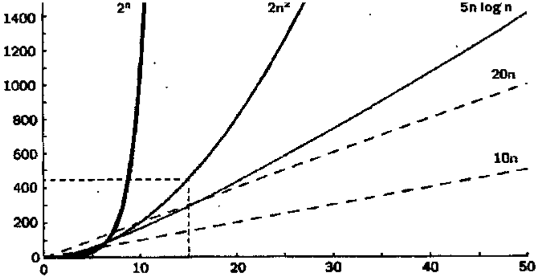

Figure : 1.2 Examples of $T(N)$: 2^n, $2n^2$, $5n\ log_2(n)$, $20n$, $10n$

Types of Algorithm Complexity Analysis

- *Worst-case complexity – “Big-O”:*
  - complexity for the *worst-case* of input of size N, i.e. the *longest running time* $T(N)$, of the algorithm out of all possible inputs of size N.
  - Example: searches all 10 (N) items in `numbers` array if the value, e.g. 99, is not there.
  - Performance guarantee of algorithm for any input, i.e. **cannot be slower**.
  - Examples: $O(N),\ O(N^2),\ O(N^3),\ O(2^N),\ \ldots$

- *Average-case complexity – “Big-Theta”:*
  - complexity for the *average-case* (or *random*) input of size N.
  - Example: on average searches 5 $(N/2)$ items in `numbers` for a value, e.g. 25, is in the array.
  - Examples: $\Theta(N),\ \Theta(\log_2 N),\ \Theta(N \log_2 N),\ \ldots$

- *Best-case complexity – “Big-Omega”:*
  - complexity for the *best-case* (or *easiest*) input of size N. i.e. the *fastest running time* of the algorithm out of all possible inputs of size N.
  - Example: search finds the value, e.g. 42, at the first position `numbers[0]`, so checks just 1 value.
  - Examples: $\Omega(1),\ \Omega(N),\ \ldots$

Big-O for Simple Programs

1. *Constant* $O(1)$, for an atomic (basic) statement:

```
value = 13 ;
```

2. *Linear* $O(N)$, for one `for`-loop of N iterations:

```
for (int i = 0; i < N; i++)
    array[i] = i * 10 ;
```

3. *Quadratic* $O(N^2)$, for two nested `for`-loops each of N iterations, total of $N \times N = N^2$:

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; i++)    // could also be j < i
        array[i,j] = i * j ;
```

4. *Cubic* $O(N^3)$, for three nested `for`-loops each of N iterations, total of $N \times N \times N = N^3$:

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)        // i equivalent to N
        for (int k = 0; k < N; k++)    // could also be k < j
            total = total + (i + j + k) ;
```

Common Complexity Classes & Big-Os

- Given the $T(N)$ (running time) of an algorithm for the *worst-case* input, we want to find its *order of complexity*, i.e. its *Big-O* value – $O(\textbf{?})$.
- The red terms in the example $T(N)$s are the *dominant term*, this means that as the value of N gets larger, the red term's value gets much bigger than all the other terms in $T(N)$.
- Consequently, when determining the Big-O for $T(N)$ we **ignore all the other terms** in $T(N)$ except the red term.

Complexity Class	**Example** $T(N)$	**O**(N)	**O**(10)
Constant	42	1	1
Logarithmic (log_2)	$\log_2(N) + 10$	$\log_2(N)$	$\lceil 3.321 \rceil = 4$
Linear	$10N, \quad 20N, \quad 999N + 7$	N	10
Linearithmic	$5N \log_2(N)$	$N \log_2(N)$	40
Quadratic	$2N^2, \quad 2N^2 + 3N + 4$	N^2	100
Cubic	$6N^3 + 5N^2 + 7N + 10$	N^3	1000
Exponential	$2^N, \quad 2^N + 6N^2 + 100$	2^N	1024
Factorial	$N! + 2N^2$	$N!$	3,628,800

Summary

- Complexity analysis measures an algorithm's running time for a given problem size N by using a growth-rate function $T(N)$.
- It is an *implementation-independent* way of measuring an algorithm.
- Complexity analysis focuses on large problems, i.e. very large N.
- Focus on the *Worst-case* analysis (Big-O), that considers the maximum amount of work an algorithm will require on a problem of a given size N.
- Big-O provides the *upper bound* on the running time $T(N)$ required.
- *Average-case* analysis (Big-Theta) considers the expected amount of work that it will require.
- *Best-case* analysis (Big-Omega) considers the least amount of work that it will require.

PART IV

.NET Framework – Collections Libraries

C♯ Collections Data Structures

- In C♯ (& other OO languages) groups of similar types of data are usually stored & manipulated as a "*collection*".
- A *collection* is a data structure that holds a collection of data objects, usually all of the same data type, in a *single logical structure*.
- Typical collection operations are: add, remove & modify either individual elements or a range of elements in a collection.
- In addition many also include methods that implement useful algorithms on the collection, e.g. searching & sorting.
- C♯ has a number of these *collection* classes available as part of its .NET development environment.

.NET Framework Class Library Overview

- The .NET Framework class libraries provide implementations for many general & application specific: types, algorithms & utility functions.
- These class libraries provide a *set of reusable types* (classes) with well-defined APIs that are available for programmers to use to develop their own programs.
- This encourages the good software engineering practise of *code reuse*.
- The .NET Framework collection classes also implement a *set of interfaces* for developing your own collection classes.
- E.g. `System.Array` class & the classes in the `System.Collections` & `System.Collections.Generic` namespaces.
- `System.Collections` namespace contains *array list* (variable sized arrays), *list*, *queue*, *stack* & *dictionary* collection classes.