Table of Contents

List of Figures

List of Algorithms

1

1. Collections, ADTs, and Arrays

In this session we introduce the problems associated with storing and managing data within a program and begin learning how to solve these problems. We introduce the concept of an Abstract Data Type, examining how they are used to provide a specification for data structures and how they relate to object-oriented development and the Unified Modelling Language (UML). We then begin our study of how to manage the storage and retrieval of data in our programs by building on pre-requisite studies of arrays to build our first reusable generic collection.

Objectives

In this session you will learn

- The problem of managing data;
- The role and impact of classes, libraries, and algorithms;
- The concept of an Abstract Data Type, and how to read and write them; and
- How to store data in array, retrieve data from an array, and resize an array; and
- The concept of a generic collection and how to build a generic collection using arrays.

1.1. The Problem of Data

The vast majority of software applications written today are required to perform processing on data in one way or another. Modern society has evolved in such a manner that vast amounts of data are generated in every moment of every day. Consider the volumes of data involved in the following applications:

- Web search – web sites such as Google and Bing actively download publicly accessible web pages using software known as a web crawler, creating a very large index (database) of the content found on those sites. Considering the size and growth of the web, the amount of data being indexed is enormous, but even though vast amounts of data are being captured and stored what is more impressive is the ability of these web sites to allow massive numbers of queries to be performed on this data by the public, at the same time, and with a very short response time.

- Shopping loyalty programs – these programs often involve a customer registering their purchases in exchange for some kind of rewards. When considered simply, these programs seem to primarily benefit the customer, who receives rewards such as flights, white-goods, etc. However the companies offering such programs also record the items purchased and are able to draw a great deal of information by looking for trends among purchases. For example, companies can identify products that are commonly purchased together (placing them closer together), promote such combinations using products with higher margins, send vouchers to customers likely to be interested in purchasing an item based on other customer purchases, and so on.
- Google web history – in many respects performing similar functionality to loyalty programs, Google keeps a record of the web searches you have performed and potentially web sites you have visited (if you have Google Toolbar installed with the PageRank meter switched on). The intention is to develop an understanding of how both yourself and other users are using the web in order to provide better targeted advertising, and so on. Consider the amount of data being fed to the Google database by the millions of users browsing the web at any time of day.
- Credit card anti-fraud – credit card providers and banks use software to automatically search through transactions on credit cards in an attempt to find transactions that may be fraudulent. Fraudulent transactions may be discovered by examining transactions with untrustworthy vendors (or vendors that are regularly targeted), payment gateway providers which are frequently used by criminals, etc., or may be simply unusual/strange/unlikely patterns of spending, e.g., paying for grocery shopping in Sydney then 20 minutes later buying a plane ticket in New York.
- Share market and currency exchange[1] – huge volumes of financial transactions for the purchase and sale of shares and currency occur almost every minute of the day. In terms of the share market, a large volume of trading is now either driven by a person using a computer terminal, or by automated trading systems (commonly used by managed investment schemes, superannuation schemes, and so on), e.g., if the share price falls to $4 then sell all shares. Currency exchange is driven primarily by trading between nations either through direct exchange of their own currency or through an intermediate reference currency or global currency (for which the US dollar is used) results in massive exchanges of currency throughout the day.

There are many more example applications involving high volumes of data that could be provided, but the implication should be clear – to be a successful software developer you need to learn how best to store, retrieve, and process data. This represents the primary topic of SIT221 Classes, Libraries and Algorithms.

[1] Another related example related to the share market was seen recently following a fake posting from the Associated Press Twitter account. Within a few minutes of the fake tweet automated trading systems caused a loss of over $US 135 billion on the US share markets. See http://www.abc.net.au/news/2013-04-24/ap-twitter-feed-hacked/4647630 for more information.

1.2. Classes, Libraries and Algorithms

The content of this unit can be broken primarily into three areas which are indicated in the name of the unit:

- Classes – referring to a study of the structures used within the software applications we write, the most important of which are data structures;
- Libraries – important in many ways, we explore the applications, development, and documentation of code libraries; and
- Algorithms – algorithms represent the functionality defined in our software and in this unit we undertake a study of algorithms to learn more about their behaviour, new ways to construct algorithms, and how to use the data structures we introduce (under Classes, above).

We will undertake a study of these concepts using the C# programming language which we learned in the pre-requisites, however it is important to note that we have effectively completed our study of programming language concepts and our focus now shifts to learning how best to apply such concepts to solving problems. The use of the C# programming language offers many advantages to our studies:

- The majority of students enrolled this unit will already be familiar with the C# programming language, having completed a study of the programming language in the pre-requisite units;
- Students who have recently transferred to Deakin's Bachelor of Information Technology and have received credit for the pre-requisite units in most cases would have studied the Java programming language whose syntax is almost identical to C#;
- The C# programming language is available for free to users of all hardware platforms, either through commercial (Visual Studio / Visual C#) or open source (MonoDevelop);
- The C# programming language supports many of the features important for developing reusable data structures and algorithms while also providing many pre-built versions;

1.3. Collections

When developing applications we are often dealing with groups of similar data, i.e., multiple objects of the same type (class), which are often referred to as collections. From the examples presented in Section 1.1 we might consider collections of:

- Web pages, each of which contains a number of key words and phrases;
- Purchases, each of which contains a number of items purchased;
- Uniform Resource Locators (URLs), each of which may have been referred by one or more other web pages[2];
- Credit cards, each of which may have one or more transactions; and
- Share holdings, each of which will have an owner/investor, each of which may have one or more other share holdings or currency holdings.

[2] When a web page contains a link to a second page, the web server for the second page receives the URL of the first web page (even if from a different server/company) known as the referring web page or the referrer.

The above examples have been used to show not only obvious collections of data, but also that each element in a collection may also contain/reference one or more other collections. If this is not clear, consider the following example: each country has a number of educational institutions, each of which have a number of students, each of which have completed zero or more subjects/units and also are enrolled in zero or more subjects/units, each of which has a list of students enrolled in that unit, each of which have allocated class times, exam times, assignments, and so on.

From the pre-requisite unit/s, the primary collection we have seen to date is the array, which represents the most commonly used collection in programming. Arrays are provided by practically every programming language and we take a closer look at how to work with arrays as collections in Section 1.6as we will see during our studies in this unit, we also use arrays in the construction of many other types of collections.

Fundamentally, when undertaking a programming task you will come across many examples of where a collection of data must be used. For this reason, collections usually represent one of the best examples of reusability to be found in programming. For example, consider the concept of a list. From the examples above, you could have a list of keywords, a list of items purchased, a list of web pages referencing a particular URL, a list of transactions on a credit card, or a list of share holders. If you consider these examples carefully, you will see that although the type of data being stored in the collection varies, the concept of the list remains consistent. Object-oriented programming concepts are particularly useful in developing highly reusable collections, which is demonstrated by the collections often built in to such platforms:

- Microsoft.Net/C# provides collections in the `System.Collections` namespace (using the `object` type) and in the `System.Collections.Generic` namespace (introduced after support generics were added to provide a type safe alternative to `System.Collections`);
- Java provides a number of collections in the `java.util` namespace; and
- C++ provides a number of collections in the Standard Template Library (STL).

We will examine the collections provided by Microsoft.Net/C# in Section 4.2. For now, we will focus on how to develop such highly reusable collection classes, the key to which is to focus on the object-oriented concepts of ***abstraction*** and ***encapsulation***.

1.4. Basic Terminology

When undertaking a study of data structures and algorithms, there are a number of terms that you will see appearing regularly throughout the literature and this unit. Here we attempt to provide a reasonable summary of these terms to allow you to comprehend what you are reading:

- Simple data type – refers to a primitive or fundamental piece of data, e.g., characters, integers, floating point numbers, etc. are all simple data types;
- Complex data type – usually represented in programming language by classes and/or structs, a complex data type refers to the combination of zero or more simple and/or complex data types into a single unit of data, e.g., a person’s contact information, information

for a book (author, title, publisher, etc.), a product catalogue, etc., could all be represented as complex data types;

- Collection – a data structure used to store any number of data elements, usually defined as a reusable class;
- Element – an element refers to a single piece of data to be stored in a collection; and
- Traversal – the process of stepping through a data structure/collection to access every data element stored within the data structure only once.

1.5. Abstract Data Types

Before we begin examining our first data structure in Section 1.6, the array, it is important to first consider the concept of an ***abstract data type*** (ADT). ADTs are used throughout the literature when examining data structures as they provide a means through which the requirements and/or features of the data structure can be clearly specified. ADTs specify the data to be stored in the structure and the operations that can be performed upon that data. Critically, ADTs do not specify (i) how data is to be represented/stored in the data structure; or (ii) how the operations on that data are to be implemented. This means that for any particular ADT, many different implementations may be possible.

There is no standard or official notation for writing an ADT, however most textbooks examining data structures and algorithms will usually define their own notation and use it consistently throughout the textbook. However a good ADT must include clear specifications of both the data to be stored and the operations that can be performed upon that data. Critically, all specification should not imply any requirements for the implementation.

Specifying Data

For many data structures, the type of data to be stored will be indicated as a generic or unspecified type, e.g., typeA or Element. For example, consider that an ADT could be written for a list – but a list of what? A shopping list, list of students, list of items in an order, list of phone numbers, etc., suggest different types of data elements that could be stored but the list concept remains consistent throughout.

Other data structures however may be more specific with regards to the data being stored, either indicating simple data types (integers, characters, etc.) or complex types (often represented by another ADT). For example, consider writing an ADT to represent a (calendar) date. A date consists of three numeric values: the *day* (one or two digits in the range 1-31), *month* (1-12), and *year*[3] (1-9999). These different data elements could be specified using an integral data type, however it should be noted that by specifying there are still no requirements imposed upon an implementation of the ADT. In particular, this input data could be represented in an implementation in a number of ways:

[3] The question of what makes a value for a year valid will depend on the context, e.g., if you were representing the birth date for your clients you might specify a valid year as 1900 to the current year, however for a database storing information about archaeological studies you may need to consider negative values (for dates BCE) up to the current day. Note however that there is no year 0 CE.

- By using two single byte variables (day and month) and an unsigned short variable (year);

```
byte _Day = day;
byte _Month = month;
ushort _Year = year;
```

- By using a single unsigned integer variable to represent the reversed date (useful for ordering data by date), e.g., 20120630;

```
unsigned int _Date = year * 10000 + month * 100 + day;
```

- By using the built-in `DateTime` data type[4];

```
DateTime _Date = new DateTime(year, month, day);
```

- By storing the date textually in a string, e.g., Jun 30, 2012;

```
string [] months = { "Jan", "Feb", "Mar", "Apr", "May",
"Jun",
               "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
string _Date = string.Format("{0} {1}, {2}",
               months[month - 1], day, year);
```

- And so on.

Specifying Operations

The specifications for operations that can be performed upon the data must describe the semantics of the operation (what it does) without describing how that operation is implemented (how it works). The following elements should appear for the specification of an operation for an ADT:

- The name of the operation;
- A description of both the (input and output) parameters and return values, whose data types are specified as per the guidelines above for specifying data;
- An unambiguous description of what the operation does, e.g., using mathematical expressions, pseudocode[5], descriptive text, etc.; and
- Preconditions and postconditions (described below).

Preconditions and Postconditions

[4] Information on the built-in `DateTime` data type can be found at:
http://msdn.microsoft.com/en-us/library/system.datetime.aspx
[5] Note that pseudocode is often presented in textbooks using a notation that is very similar to a programming language. This is useful when starting to teach students programming but pseudocode can be a much higher level list of major steps.

Preconditions and postconditions provide a means to describe the requirements and impacts of an operation on the ADT itself. To explain preconditions and postconditions, we will consider operations on the list concept, as described above.

Preconditions are used to describe conditions that must be true before the execution of an operation. Example preconditions that might apply to an operation to insert an item in the list could include:

- The list exists;
- The list can be modified/is not read-only;
- The item is not currently stored in the list (if the list cannot contain duplicates); and/or
- There is room in the list for another item or there is enough memory to expand the list to store the item.

Similarly, preconditions that might apply to an operation to delete an item from a list could include:

- The list exists;
- The list can be modified/is not read-only; and/or
- The item to be removed appears in the list.

Alternatively, ***postconditions*** describe conditions that must be true after the execution of an operation. Continuing our examples, postconditions that might apply after the operation to insert an item in the list could include:

- The item appears in the list; and/or
- There is one more item stored in the list than before.

Similarly, postconditions that might apply after the operation to insert an item in the list could include:

- The item no longer appears in the list; and/or
- There is one less item stored in the list than before.

Critically, the preconditions and postconditions listed above are only examples, many more preconditions and postconditions may also be possible and only a subset of the examples above may apply to any particularly list ADT.

ADTs and Object-Oriented Development

By now you are most likely identifying similarities between the concepts of an ADT and the concepts that we have learned to date about object-oriented development in the pre-requisite units. In particular, there are clear similarities between the interface that is defined by an ADT and the interface that is defined by a class. Many authors do in fact consider ADTs to be an important step in the development of the object-oriented development paradigm. Moreover, classes are often said to be the implementation of the ADT concept.

ADTs provide the specification for a data type in the same way that UML provides a specification for an object-oriented system. Typically this specification identifies a collection of data, e.g., an array, stack, or queue, however this is not strictly necessary. Where an ADT specifies a particular combination of data and specifies the semantics of operations on that data, object-oriented development considers the implementation of the ADT. In modern software development, UML is considered the primary means of specifying a system, however, most textbooks covering data structures and algorithms still choose ADTs for specification as they provide much more information and detail for this purpose.

ADTs and The UML

As noted above, ADTs and the UML both have similar objectives in terms of providing a specification for elements of software. Importantly, the UML can be used when writing an ADT, e.g., UML class diagrams can be used to specify the name of the ADT, the types of data stored by the ADT, and some parts of the specifications of operations (name, parameters, and return types in particular). The missing parts of an ADT specification can be added to a UML class diagram through a combination of comments and other elements such as the Object Constraint Language (OCL)[6]. The critical point here is that UML diagrams cannot, on their own, be used as a complete ADT specification.

Example 1: Integer ADT

ADT_begin Integer
data:

val

Stores the current value of the Integer, a positive or negative decimal number with no fractional component

operations:

create(x: Integer)

Creates an Integer and assigns the value of x to val
Parameter x: the initial value of the integer
Precondition: x != null
Postcondition: this.val = x

get: Integer

Returns the current value of val
Precondition: none
Postcondition: returned this.val

set(x: Integer)

Assigns the value of x to val
Parameter x: the new value of the integer
Precondition: x != null
Postcondition: this.val = x

add(x: Integer): Integer

Returns the result of adding the value of x to the current value of val
Parameter x: an integer value to be added to the current stored value
Precondition: x != null and val != null
Postcondition: returned this.val + x.val and no change to this.val or x.val

[6] Comments are actually defined in the UML specification (see Section 9.5.1 of the UML Unfrastructure Specification (http://www.omg.org/spec/UML/2.4.1/)

```
    subtract(x: Integer): Integer
        Returns the result of subtracting the value of x from the current value of val
        Parameter x: an integer value to be subtracted from the current stored value
        Precondition: x != null and val != null
        Postcondition: returned this.val – x.val and no change to this.val or x.val
    increment
        Increases the current value of val by one
        Precondition: this != null
        Postcondition: this.val = pre_this.val + 1
    decrement
        Decreases the current value of val by one
        Precondition: this != null
        Postcondition: this.val = pre_this.val – 1
    …
ADT_end
```

Example 2: Library Catalogue ADT

```
ADT_begin Book
data:
    author: String
        Stores the author of the book
    title: String
        Stores the title of the book
    year: Integer
        Stores the year in which the book was published
operations:
    create(a: String, t: String, y: Integer)
        Creates a Book and assigns the value of a to author, t to title, and y to year
        Parameter a: the author of the new Book
        Parameter t: the title of the new Book
        Parameter y: the year the new Book was published
        Precondition: a != null
        Precondition: t != null
        Precondition: y != null
        Postcondition: this.author = a
        Postcondition: this.title = t
        Postcondition: this.year = y
    get_author: String
        Returns the current value of author
        Precondition: none
        Postcondition: returned this.author
    set_author(a: String)
        Assigns the value of a to author
        Parameter a: the new author of the Book
        Precondition: a != null
        Postcondition: this.author = a
    …
ADT_end
```

```
ADT_begin Entry
data:
    book: Book
        Stores a book listed in the catalog
    dewey: String
        Stores the dewey-decimal reference allocated to the book
operations:
    create(b: Book, d: String)
        Creates an Entry and assigns the value of b to book, and the value of d to dewey
        Precondition: b != null
        Precondition: d != null
        Postcondition: this.book = b
        Postcondition: this.dewey = d
    get_book: Book
        Returns the current value of book
        Precondition: none
        Postcondition: returned this.book
    set_book(b: String)
        Assigns the value of b to book
        Parameter b: the new Book referred to by this Entry
        Precondition: b != null
        Postcondition: this.book = b
    …
ADT_end

ADT_begin Catalog
data:
    books: List of Entry objects
        A collection of books that represent the catalog
operations:
    create
        Creates a Catalog
        Precondition: none
        Postcondition: books.Empty == true
    add_entry(e: Entry)
        Adds Entry e to the books list
        Parameter e: the Entry to be added to the Catalog
        Precondition: e != null
        Postcondition: books.Contains(e) == true
    find_book(author: String): Book[ ]
        Searches the catalog for all Books written by the indicated author
        Parameter author: the name of the author to be searched for
        Precondition: author != null
        Postcondition-success: returned list of Books written by author
        Postcondition-fail: returned null
    find_book(title: String): Book[ ]
        Searches the catalog for all Books with the indicated title
        Parameter title: the title of the book to be searched for
```

Precondition: title != null
Postcondition-success: returned list of Books with specified title
Postcondition-fail: returned null

find_book(dewey: String): Entry

Searches the catalog for an Entry with the indicated dewey decimal number
Parameter dewey: the dewey decimal-decimal reference to search for
Precondition: dewey != null
Postcondition-success: returned Entry where Entry.dewey = dewey
Postcondition-fail: returned null

…

ADT_end

1.6. Arrays

An array represents the simplest structure that can be used for storing a collection of data. Arrays are provided by most programming languages and it is critical that you are able to work with these collections. Fundamentally there are three types of arrays:

- One-dimensional array (Figure 1.1a) – a simple array that contains a number of elements, e.g., an array containing 5 elements;
- Multi-dimensional array[7] (Figure 1.1b) – an array that contains elements in more than one dimension, e.g., a two-dimensional array with 5 rows and 4 columns (20 elements in total);
- Jagged array (Figure 1.1c) – similar to multi-dimensional arrays, except that the number of elements in a particular dimension do not need to be equal, e.g,. a two-dimensional jagged array with 3 rows with 6, 4, and 5 columns (15 elements in total).

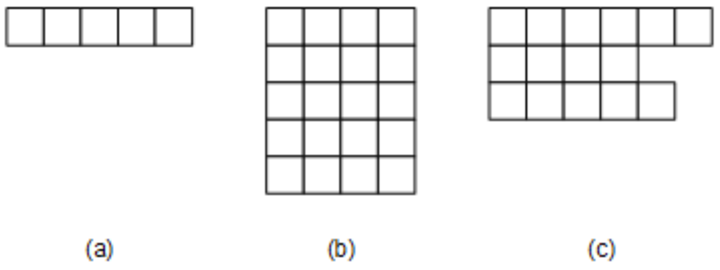


Figure 1.1: Visual Representation of Array Types

The syntax for declaring a one-dimensional array is as follows:

```
type [] name = new type[size];
```

or separating the declaration and creation of the array:

```
type [] name ;
name = new type[size];
```

Similarly, the syntax for declaring a multi-dimensional array is as follows:

[7] Multi-dimensional arrays are sometimes referred to as n-dimensional arrays.

```
type [,[,[...]]] name = new type[size1,size2[,...]];
```

or

```
type [,[,[...]]] name;
name = new type[size1, size2[,...]];
```

Finally, the syntax for declaring a jagged array[8] is as follows:

```
type [][] name = new type[rows][];
name[0] = new type[size1];
name[1] = new type[size2];
[...]
```

or

```
type [][] name;
name = new type[rows][];
name[0] = new type[size1];
name[1] = new type[size2];
[...]
```

Note that the first line of the syntax for creating a jagged array creates the rows of the jagged array, whereas the second and further statements create the elements for the individual rows, which do not need to be equal in length. An example matching Figure 1.1c is as follows:

```
int [][] jaggedArray = new int[3][];
jaggedArray[0] = new int[6];
jaggedArray[1] = new int[4];
jaggedArray[1] = new int[5];
```

The most apparent limitation of arrays, their fixed size, also results in the most important advantage of arrays. In particular, the elements of an array are usually stored contiguously in memory, i.e., in a single uninterrupted sequence of bytes in memory. This prevents arrays being resized as the array will be surround by other data used by your program. By having array elements stored contiguously, it is possible to access the elements of the array directly – given the location of the array (where it starts in memory), it is possible to determine the location of any element stored in the array by a simple calculation, as follows:

$$address_{element} = address_{base} + (index \times \text{sizeof}(element))$$

For example, consider attempting to access the 45th element (index 44) in an array of integers which is stored in memory starting at location 5000[9]:

$$address_{element} = 5000_{10} + (44_{10} \times 4_{10}) = 5176_{10}$$

[8] Note that this syntax only illustrates creating a two-dimensional jagged array.
[9] For simplicity we use decimal numbers (base 10) for addressing here. Computers of course work in binary (base 2) which is usually shortened in programming using either octal (base 8) or more commonly hexadecimal (base 16).

By demonstrating this simple calculation, it should also be apparent why most programming languages use indexes which start at a zero value – the index is used to calculate an offset from the start of the array, and using an index of zero results immediately results in a zero offset. Importantly it may not yet be apparent why direct access to data elements is important - surely all data stored in the memory/RAM of a process can be accessed directly? As we will see during our study of alternative data structures, direct access it not always possible as we first need to determine the location of an element of data before it can be read/modified.

The ability to work effectively with arrays is fundamental for any programmer. In this section we examine several techniques for store data in and retrieving data from an array. We will continue this study later in the trimester when examining how to sort data in an array (Section 7.2) and how to perform a binary search on sorted data in an array (Section 8.1.2).

1.6.1. Storing Data in an Array

How to store data in an array is always covered in courses introducing fundamental programming, e.g.,

```
array_name[index] = value;
```

However, the problems new programmers are required to solve are usually simplistic, e.g.,

- Prompt the user for 10 numbers and store them in an array;
- Create an array with 5 numbers and write methods to calculate the sum and product of their values;
- And so on.

These kinds of problems eliminate a number of questions that need to be answered when developing software for real applications – what size array should be declared? what to do all the elements of an array aren't used? what to do if the array isn't large enough/is too big? In other words, such courses usually instruct students on the syntax of using arrays but rarely on how to solve problems using arrays. In this section, we examine alternatives for tracking how much data is used by an array and how to dynamically change the size of an array. Consider a problem which requires you to read data from a file and store it into an array. The two problems we encounter are:

1. How many elements are to be read from the file and stored in the array? The answer to this question tells us what size our array needs to be. Unfortunately a file could contain an arbitrary number of elements so there may be no answer to this question. There are three apparent solutions:
    - Create an array of sufficient size for all possible file inputs – this solution is wasteful of main memory (how many files will have the maximum number?) but also assumes that there is a maximum size;
    - Make two passes through the file, the first to determine how many elements there are, then create the array and re-read the file storing the values – this solution has the disadvantage that the file must be read twice, which may add significant overhead to our programs (working with disk is vastly slower than working with memory);

- Create an array of some small arbitrary size then increase the size of the array as necessary (see below for how to change the size of an array) – as we will see below, increasing the size of an array will involve some overhead and two problems faced are how many times do we need to increase the size of the array and will there be enough memory to increase the size of the array?

The actual solution to use for this problem will depend upon the actual application/situation that this problem arises.

2. How do we keep track of which elements are used in the array? For the problem presented, the solution to this problem is straightforward, we keep a reference to the last element used in the array (or next free element), shown in Algorithm 1.1, using this reference to store each data element in turn.

```
ASSUMPTION: An array named array, of sufficient size, already exists
lastUsed ← –1
data ← read(file)
while NOT eof(file)
    lastUsed ← lastUsed + 1
    array[lastUsed] ← data
    data ← read(file)
end-while
```

Algorithm 1.1: Storing data in an array from a file tracking the last element used

Although we have now solved the problem of how to store data into the array initially, what now happens if we want to insert a new element into the middle of the array or if we want to delete an element from an array? Given that arrays are stored contiguous, the only way to insert a new element is to first shuffle the elements stored in the array to make an empty space. Algorithm 1.2 demonstrates this process:

```
ASSUMPTION: An array named array, of sufficient size, already exists
ASSUMPTION: A variable named lastUsed contains index of the last used element in the array
ASSUMPTION: A variable named insertLocation specifies the index where data is to be inserted
ASSUMPTION: A variable named data contains the new element of data to be inserted
lastUsed ← lastUsed + 1
shuffleLocation ← lastUsed
while shuffleLocation > insertLocation
    array[shuffleLocation] ← array[shuffleLocation – 1]
    shuffleLocation ← shuffleLocation – 1
end-while
array[insertLocation] ← data
```

Algorithm 1.2: Inserting data in an array by shuffling elements

Similarly, it is also possible to delete an element by shuffling the elements stored in the array to eliminate that element, as shown in Algorithm 1.3:

```
ASSUMPTION: An array named array already exists
ASSUMPTION: A variable named lastUsed contains index of the last used element in the array
ASSUMPTION: A variable named deleteLocation specifies the index of the data element to be deleted
shuffleLocation ← deleteLocation
while shuffleLocation < lastUsed
        array[shuffleLocation] ← array[shuffleLocation + 1]
        shuffleLocation ← shuffleLocation + 1
end-while
lastUsed ← lastUsed – 1
```

Algorithm 1.3: Deleting data from an array by shuffling elements

Importantly, although this is probably the most obvious way to delete an element from at this point, if you think about the problem carefully it is also possible, for an array containing data without any ordering, to delete an element by replacing it with the last element in the array, as shown in Algorithm 1.4:

```
ASSUMPTION: An array named array already exists
ASSUMPTION: A variable named lastUsed contains index of the last used element in the array
ASSUMPTION: A variable named deleteLocation specifies the index of the data element to be deleted
array[deleteLocation] ← array[lastUsed]
lastUsed ← lastUsed – 1
```

Algorithm 1.4: Deleting data from an array by replacing with last used

The important advantage of this approach, is that we have eliminated the loops required to delete the element, substantially improving our algorithm (see Algorithm Complexity in Section 7.1).

Important tip!

> ***IMPORTANT TIP!*** *Whenever implementing the above algorithms for deleting data from an array, it is always important to think about how they will behave in the programming language you are using. For example, consider using these algorithms for an array of reference types in C#. If you recall, reference types are created with the new keyword and are freed by the garbage collector once they are no longer referenced. Implementing the above algorithms, as presented, could easily lead to one or more objects being referenced even though they are no longer needed. To prevent this, you would need to insert an additional step to set the reference to the last element to null, immediately before the last used reference is decremented..*

Note that it is also possible to combine the above concepts to create the original array already sorted by inserting data in the array in-order, as shown in Algorithm 1.5.

```
ASSUMPTION: An array named array, of sufficient size, already exists
lastUsed ← –1
data ← read(file)
while NOT eof(file)
    // Determine insert location
    insertLocation ← 0
    lastUsed ← lastUsed + 1
    while insertLocation < lastUsed AND data > array[insertLocation]
        insertLocation ← insertLocation + 1
    end-while
    shuffleLocation ← lastUsed
    while shuffleLocation > insertLocation
        array[shuffleLocation] ← array[shuffleLocation – 1]
        shuffleLocation ← shuffleLocation – 1
    end-while
    array[insertLocation] ← data
    data ← read(file)
end-while
```

Algorithm 1.5: Storing data in-order in an array from a file tracking the last element used

Shuffling elements in an array however is a relatively expensive operation, requiring one memory read and one memory write for each element of data to be moved. This is particularly a problem when inserting data into an array in-order (Algorithm 1.5). For example, consider inserting storing 10 elements into an empty array. Assuming that on average half the elements would need to be shuffled and rounding down any halves, the number of elements to be shuffled would be: 0, then 0, 1, 1, 2, 2, 3, 3, 4, and 4 (a total of 20 elements shuffled). This sequence continues growing exponentially, e.g., for storing 100 elements a total of 2,450 elements would need to be shuffled, and for 1000 elements a total of 249,500 elements would need to be shuffled.

Unfortunately there is no way to improve the algorithms presented here for shuffling data in an array, nor are there better algorithms for performing the same task. However as we will learn in this unit, the inability to improve an algorithm does not imply an inability to improve the performance of the relevant functionality. The problem we are facing is that we are treating an array in a linear manner. By introducing a different structure to our data, or even by using a different data structure, i.e., to not use an array, we can improve the performance of the functionality we wish to achieve.

1.6.2. Retrieving Data from an Array

During the trimester we will study a number of ways for improving the way we retrieve data from an array (or any other linear list), however here we will only consider the linear search, also known as the sequential search, which you should already be familiar with. Very simply, the input to a linear search algorithm is the collection itself (the array) and the target to search for. The actual algorithm then checks each element in turn until either the target is found, or the end of the collection is reached, as shown in Algorithm 1.6.

```
ASSUMPTION: An array named array already exists
ASSUMPTION: A variable named lastUsed contains the index of the last used element in the array
ASSUMPTION: A variable named target contains the desired search target
targetIndex ← -1
for i ← 0 to lastUsed
        if array[i] = target
                targetIndex ← i
                break;
        end-if
end-for
// if targetIndex = -1 then element not found, otherwise target is found at index stored in targetIndex
```

Algorithm 1.6: Searching for data in an array using a linear search

1.6.3. Resizing an Array

The title of this section is somewhat misleading, as an array has a fixed size and thus cannot be resized. Instead when we need to change the size of an array, we replace it with a new array and copy the elements from the old array, as shown in Algorithm 1.7, which shows how to increase the size of an array. Decreasing the size of an array only requires minor adjustments to this algorithm.

```
ASSUMPTION: An array named array already exists
ASSUMPTION: A variable named lastUsed contains index of the last used element in the old array
ASSUMPTION: A variable named newSize specifies the size of the new array
newArray ← new array[newSize]
for i ← 0 to lastUsed
        newArray[i] = array[i]
end-for
array ← newArray
```

Algorithm 1.7: Replace an existing array with a new larger array

The result of this algorithm is illustrated in Figure 1.2, which shows a scenario where we have an existing array of size five containing the elements a, b, c, d, and e, (the array is full), and we wish to store an additional sixth element. To do so, the new larger array is created, the elements are copied across, and the old array is discarded, and the new sixth element (f) is stored.

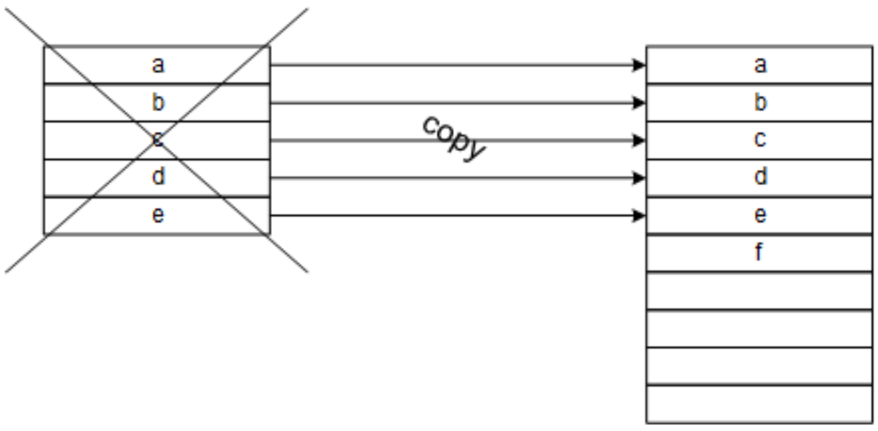


Figure 1.2: Resizing an Array

1.7. Implementing Collections

The concept that a collection can be used for different data types and different applications is common to all data structures we will examine in this unit. The developers of programming languages also recognise the importance of this concept and provide us with the means to develop our code independent of any specific data type. Most object-oriented programming languages provide a base type from which all other types inherit from. In C# this is known as the `object` type, which represents the Microsoft.Net `System.Object` class. Collections can be built in C# using the `object` type and a number of pre-built collections are provided using this approach in the `System.Collections` hierarchy.

The use of the common `object` base type for collections has a number of disadvantages however. Firstly, any objects retrieved/extracted from a collection need to be cast to their actual data type before they can be used, e.g.,

```
Invoice currentInvoice = (Invoice)invoiceCollection[i];
```

Although this may be inconvenient, it doesn't represent a significant problem. Secondly, collections using the `object` base type are not efficient when used to store simple types (`int`, `char`, `float`, etc.). Although these types are considered objects, Microsoft.Net optimises the performance of applications by mapping these types directly to the hardware (CPU registers etc.). Before they can be stored in such a collection these values must first be encapsulated into an object, known as ***boxing*** (extracting the value from an object and returning it to a simple type is referred to as ***unboxing***).

Finally, and most critically, collections using the `object` base type are not completely ***type safe***. A historically common error in program code occurred when programmers would accidentally interpret data stored in memory using the wrong data type, e.g., interpreting an `Invoice` object as a `Book` object. Modern programming languages provide additional checking to reduce and/or eliminate this error, e.g., the following line would cause an exception to be thrown if the `invoiceCollection` only contained `Invoice` objects:

```
Book currentBook = (Book)invoiceCollection[i];
```

Importantly however, if the `invoiceCollection` was written using the `object` base type, both of the following lines would be syntactically correct and not cause an exception to be thrown:

```
invoiceCollection.Add(new Invoice(…));
invoiceCollection.Add(new Book(…));
```

These lines are both acceptable because the `Add` method would accept a parameter of type `object`, for which of course all objects in the system, regardless of type, would be accepted.

A much better approach is through the provision of ***parameterised types***, which allow code to written independent of any particular data type(s). Unlike the use of the `object` base type however, the data types being used must be specified when the collection is instantiated. Moreover, whereas the `object` base type allows any data type to be stored, parameterised types allow the question of which data types are to be stored to be deferred until the collection object is created.

Parameterised types are provided in C# through the provision of ***generics***. Generics allow both methods (***generic methods***) and classes (***generic classes***) to be defined where the data type used is ***deferred*** until the method or class is actually used. A number of pre-built collections are provided using generics in the `System.Collections.Generic` hierarchy.

Consider the following method that has been written using generics:

```
TYPE Minimum<TYPE>(TYPE first, TYPE second)
    where TYPE : IComparable<TYPE>
{
    TYPE result = first;
    if (second.CompareTo(first) < 0)
        result = second;
    return result;
}
```

The method takes two parameters, `first` and `second`, which are compared and the minimum value of the two is returned. This is a method is particularly unusual except that the data type of the parameters and return type are indicated as `TYPE` instead of a data type that we are familiar with. This is because the data type is specified as generic, indicated by `<TYPE>` appearing after the method name. Several different data types can be specified using a comma separated list inside the angled brackets ('<' and '>').

Note that the deferred data type (whose name is usually capitalised) can be used for parameters, the return value, and local variables. Also note that a constraint has been specified for the generic `TYPE`, as follows:

```
where TYPE : IComparable<TYPE>
```

This constraint has been specified to ensure that any objects used with this method implement the `IComparable<>` interface which ensures the `CompareTo` method, used in the implementation of the method, has been defined.

It is also possible to apply generics to whole classes, using the following syntax:

```
class class_name<type_name[, ...]>
     [where type_name : class_or_interface_name
     [...]]
{
     ...
}
```

Objects can then be created from generic classes as follows:

```
class_name<data_type> variable_name = new class_name<data_type>(parameters);
```

For example:

```
Vector<Invoice> invoiceCollection = new Vector<Invoice>();
```

Note that it is also possible to nest the use of generic types as well, as follows[10]:

```
Vector<Vector<int>> listOfIntegerVectors = new Vect-
or<Vector<int>>();
```

Task 1.1. Develop a Reusable Collection: Vector

Objective: In this task we begin developing our first reusable collection class, which we will enhance in future weeks. This task represents our first steps moving beyond learning how to use a programming language (pre-requisite units) into how to solve problems in programming languages. The need to develop custom collection classes is common in more advanced programming and represents an important skill to learn.

For this task you are required to develop a generic collection `class Vector`, using the C# programming language, which allows data to be stored in a one-dimensional array which is dynamically resized as necessary. The following features are to be provided by your class:

- Namespace:
  - `namespace SIT221_Library`
    Your class should be defined in the namespace `SIT221_Library`;
- Properties:
  - `public TYPE this[int index]`
    A read/write indexer property which either retrieves a value stored in the array at the specified index or replaces the value stored at a particular index;
  - `public int Capacity`
    A read/write property which either returns the current array size or resizes the array to the specified size if possible (note that the new size must be large enough to hold the stored elements to be successful);
    Hint: The size of a one-dimensional array can be determined using it's length property, e.g., myArray.Length
  - `public int Count`
    A read-only property which returns how many elements are currently stored in the array;
- Constructors:
  - `public Vector()`
    A parameter-less constructor which creates an array using the default capacity (use a default capacity of 4);
    Hint: You should always avoid magic numbers and define this in the class using a (private) constant!
  - `public Vector(int capacity)`
    A custom constructor which takes a single integer parameter to allow the programmer to specify the initial capacity;

[10] Note that two greater than symbols appear together in this example ('>>'). Some programming languages, such as C++, may interpret this as the right-shift operator and will require an additional space between the symbols ('> >').

- Methods:
  - `public void Add(TYPE element)`
    A method `Add` which accepts a single element and stores it in the array (if the array is full, replace the array with a new one that is double the capacity);
  - `public void Clear()`
    A method Clear which removes all elements stored in the array;
  - `public bool Contains(TYPE targetValue)`
    A method Contains which searches the array to determine whether a value is stored in the array (returns true) or not (returns false);
    Hint: You will need to use the object.Equals() method to determine if the values are equal or not.
- Exceptions:
  - If an invalid index is specified as the argument to the indexer property, throw an `IndexOutOfRangeException` object with an appropriate descriptive message.
    If an invalid value is specified as the new size to the capacity property, throw an `ArgumentOutOfRangeException` object with an appropriate descriptive message.
- You may create any private elements (attributes/methods) as required.

Finally, test your code using the `Main` method provided in this week's provided code (see ZIP file in CloudDeakin).

Note: Arrays in C#/Microsoft.Net provide a built-in Resize method. You should not use this as part of completing this task – most platforms do not provide this functionality and it is important that you learn correctly how to resize an array independent of platform support.

2

2. Dynamic Data Structures with Linked Lists

In this session we begin by examining two programming language concepts that are used regularly in the development of data structures: structs and nested classes. We then begin our examination of dynamic data structures by introducing the concept of a linked list, identifying how it solves the problems of arrays while also introducing new problems. Algorithms for inserting and deleting nodes in a singly linked list are considered, including how to use these algorithms to create an ordered linked list, before finally examining the doubly linked list structure and the benefits offered by this slightly different structure.

Objectives

In this session you will learn

- The concepts of a struct and nested classes;
- The concepts of, and algorithms for, a singly linked list;
- The concepts of, and algorithms for, a doubly linked list; and
- The concepts of, and algorithms for, an ordered linked list (either singly or doubly linked).

2.1. Programming Skills

By completing the pre-requisite units you have gained an understanding of the fundamental building blocks of software development, however there is still much to learn. When preparing for a career in Information Technology involving software development, regardless of whether programming is a primary responsibility of your position or not, it is critical that you constantly look for new concepts and ways of building solutions. In this unit, whilst our focus will be examining the storage and manipulation of data, we will introduce a number of techniques that are useful to know and helpful when solving different problems. The more you are exposed to different techniques and practice solving problems in different ways, the easier programming becomes and the more valuable you become as a programmer.

In this section we examine two concepts that are useful when developing solutions to programming problems. First, we consider the concept of a struct, which is often used collect data elements into a single allocatable unit, much like we have learned to use classes for. We then examine the concepts of nested classes, i.e., defining one class as part of/inside of another class, which provides us with the ability to encapsulate more complex structures in our classes, which we will use regularly when building reusable collections.

2.1.1. Structs and C#

When reading information about data structures, particularly where example code is presented, you will often come across the definition of a data structure using the keyword `struct`. Structs are commonly used when developing in procedural programming languages such as the C programming language, but also often appear in object-oriented programming languages including C++ and C#.

Value Types and Reference Types

Before we examine the use of structs in the C# programming language, it is critical to first review the concepts of ***value types*** and ***reference types***. The difference between value types and reference types can be illustrated easily using the following two declarations:

```
int accountNumber = 12345;
Account account = new Acccount(54321);
```

In particular, the first declaration is creating a value type, and the second declaration is creating a reference type. The most obvious difference is the use of the `new` keyword for the reference type, however on its own this does not guarantee the use of a reference type, e.g., the following statement creates a value type:

```
int number = new int();
```

The difference between value types and reference types are clear however when you examine how data is stored for the above variables, as illustrated in Figure 2.1. Value types such as the `accountNumber` variable refer directly to the data they store. Reference types however, such as the `account` variable, store a memory address (a reference) to record where the object is stored in memory. When data stored in the reference type is accessed, the memory address is first retrieved and only then can the data be accessed (a two step process).

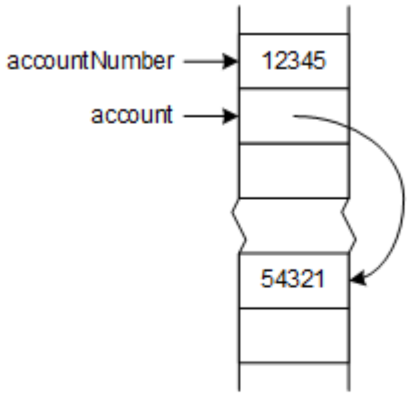


Figure 2.1: Memory Storage of Value Types versus Reference Types

This also helps explain what happens when we don't use a combined declaration for creating objects, e.g.,

```
Account account;
account = new Account(54321);
```

Using this syntax, the first line allocates the memory slot shown immediately below the `accoun-tNumber` variable and gives it the name `account`. The second line is divided into two parts. The code on the right of the assignment operator (=) creates the `Account` object and initialises its account number to `54321` (shown in the bottom half of the memory in the figure). The result of this object creation, using the `new` operator, is that the memory address of the new object is returned and assigned to the `account` variable created in the previous step.

Structs in C#

The declaration of a `struct` in the C# programming language appears almost identical to the declaration of a class, as shown in Figure 2.2. What should be immediately apparent is that this code could easily be used to define a class, except for the obvious difference that the keyword `class` has been replaced with the keyword `struct`.

```
struct StudentStruct
{
    private string _ID;
    public string ID
    {
        get { return _ID; }
        set { _ID = value; }
    }

    private string _Name;
    public string Name
    {
        get { return _Name; }
        set { _Name = value; }
    }

    public StudentStruct(string id, string name)
    {
        _ID = id;
        _Name = name;
    }
}
```

Figure 2.2: Example struct for a Student in C#

The most important difference between objects created from classes and structs in C# is that objects created from structs are value types, not reference types. To demonstrate the impact of this, we extend the above code into a small example. First, we begin by creating a class identical to the struct above, i.e., the declaration line is changed from:

```
struct StudentStruct
```

to:

```
class StudentClass
```

The name of the constructor is similarly adjusted[1]:

```
public StudentStruct(int id, string name)
```

to:

```
public StudentClass(int id, string name)
```

Now consider the `Main` method shown in Figure 2.3, where we have created two identical objects, one from the `struct` (`aStruct`), and one from the `class` (`aClass`). We then assign each of these objects to a second variable (`bStruct` and `bClass`) and then modify the `ID` stored within using the second variable.

```
static void Main(string[] args)
{
    StudentStruct aStruct = new StudentStruct("12345", "J Doe");
    StudentStruct bStruct = aStruct;
    StudentClass aClass = new StudentClass("12345", "J Doe");
    StudentClass bClass = aClass;

    bStruct.ID = "54321";
    bClass.ID = "54321";

    Console.WriteLine("aStruct: {0} {1}", aStruct.ID, aStruct.Name);
    Console.WriteLine("bStruct: {0} {1}", bStruct.ID, bStruct.Name);
    Console.WriteLine(" aClass: {0} {1}", aClass.ID, aClass.Name);
    Console.WriteLine(" bClass: {0} {1}", bClass.ID, bClass.Name);
}
```

Figure 2.3: Main Method for struct vs class Example

If you have followed the code closely, given that we only created two objects the output that you may be expecting might be four identical sets of data showing a student with ID `54321` and name `J Doe`. However, the actual output is as follows:

```
aStruct: 12345 J Doe
bStruct: 54321 J Doe
aClass: 54321 J Doe
bClass: 54321 J Doe
```

Examining this output, there appears to be more than two objects – if we only created two objects, and both were modified, how can the original ID of `12345` still exist? This is in fact exactly what happened, we created a third object when the following line was executed:

```
StudentStruct bStruct = aStruct;
```

[1] The full code for the class declaration, which is identical to the declaration in Figure 2.2 except for the two changes indicated, can be found in the provided code ZIP file for this session in CloudDeakin.

Recall that `struct` data types are value types, and as such they behave the same as simple types, such as `int` or `char`, would do. The third object created, `bStruct`, receives its values from the assignment operator, which performs a ***shallow copy***[2] of the `aStruct` object.

Other differences between using a `struct` or `class` also exist, such as how memory is used, however these differences are beyond the scope of this unit. In the C# programming language, data types are usually only created using a `struct` to represent the combination of simple elements of data, e.g., consider the `System.Drawing.Point struct` which is used to represent a point in a Cartesian plane consisting of an X coordinate and Y coordinate. This could be declared simply as follows:

```
public struct Point
{
    public int X, Y;
}
```

The `struct` defined by Microsoft.Net actually includes much more functionality[3], however regardless the use of a `struct` is ideal for this data type due to the simple data stored in the structure.

For the purposes of this unit, it is unlikely that we will use a `struct` very often, if at all, the reasons for which will become clear when we start using references between objects of the same type (for which a `struct` would not work). Regardless, it is useful to know how `struct` data types are defined and their semantics for your own/future programming projects and so you understand why we don't use them.

2.1.2. Nested Classes

When developing reusable collections, there is often a need to introduce additional structure on the data being stored. The principles of object-oriented development tell us that this additional structure should be encapsulated (hidden) from any client classes using our reusable collection, i.e., the client class should be able to insert data, delete data, retrieve data, and so on, however the client class should not be aware of any additional data used to manage the storage of data in the collection. For this purpose, we use nested classes.

Developing code using nested classes requires no additional skills or knowledge than for developing a single class, the nested class is simply defined as a member of another class, i.e.,

```
[access_modifier ]class outer_class_name
{
        [access_modifier ]class inner_class_name
        {
                [access_modifier ]inner_class_member
                …
```

[2] Recall that a shallow copy is where the addresses of any referenced objects are created, not the objects themselves.
[3] To view the details of the Microsoft.Net `struct`, see: http://msdn.microsoft.com/en-us/library/system.drawing.point.aspx

```
        }
        [access_modifier ]outer_class_member
        ...
    }
```

There are however two important points to emphasise when using a nested class:

1. The *access_modifier* applied to the inner class (nested class) has the same effect as for any other member, determining whether or not that class is visible to clients of the outer class, e.g., using the `private` access modifier would mean that the inner class is not visible to clients of the outer class but is still usable by the outer class; and
2. As a member of the outer class, all encapsulated members (`private`/`protected members`) of the outer class are visible to the members of the inner class[4], however encapsulated members of the inner class are not visible to the members of the outer class.

A simple example using a nested class is shown in Figure 2.4. In this example, the outer class stores a name but does not provide any ability to retrieve the name. The inner class however does provide this functionality, and both the class itself (`InnerClass`) and an object of the inner class type (`InnerObject`) are defined as public to allow clients of the outer class to see its members.

```
class OuterClass
{
    public class InnerClass
    {
        private OuterClass _OuterObject;
        public InnerClass(OuterClass outerObject)
        {
            _OuterObject = outerObject;
        }

        public string GetName()
        {
            return _OuterObject._Name;
        }
    }
    public InnerClass InnerObject { get; private set; }

    private string _Name;
    public OuterClass(string name)
    {
        _Name = name;
        InnerObject = new InnerClass(this);
    }
}
```

Figure 2.4: Example Nested Class

After creating an object of `OuterClass`, it is simple to retrieve the stored name as follows:

[4] Note that objects of the inner class would need to be provided with a reference to the outer class objects however (see Figure 2.4 for an example).

```
    outerObject.InnerObject.GetName()
```

Finally, note that in the example that the address of the outer object (`this`) is passed to the inner object during its creation so it can store a reference to the outer object used for accessing its private members later.

2.2. Singly Linked Lists

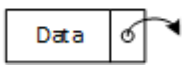
Recall from Section 1.6 the two major problems we identified when using arrays:

1. When inserting an element into/deleting an element from an array, we often need to shuffle the remaining elements stored in the array either to make a new slot or to remove the empty slot; and
2. To overcome the fixed size of an array, it is possible to replace one array with another with a different size to increase or decrease the number of empty slots, however this requires all elements to be copied from the old array to the new array.

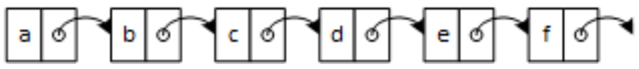
Both of these problems can be eliminated through the use of a linked lists. Simplistically, a linked list is where elements are stored in a collection connected together by links, rather than being stored consecutively in an array. Linked lists solve the above problems by (1) separating the order of elements in the collection from their relative locations, and (2) allowing the storage for elements to be allocated separately for each individual element.

2.2.1. Singly Linked List Concepts

Before we can develop a linked list, we first need to introduce the concept of a ***node***. A node is a simple data structure that is used to hold a single element of data and a link to another node, the ***next*** node, i.e.,



This data structure is also known as a ***self-referential structure***, as it contains a reference to another object of the same type. By using this reference to connect several nodes together into a chain, we construct a linked list, e.g.,



In addition to the linked list itself, it is also necessary to keep a record of at least the first node in the list, referred to as the ***head***, and it is also often useful to keep track of the last node in the list, referred to as the ***tail***. The need for a reference to the head node is immediately apparent as it allows us to access the list itself. The reference to the tail node however is not necessary however – it is possible to locate the tail node by traversing the list until the last node is reached. The use or lack of use of a tail reference however is simply a trade-off between using more memory (to store the tail reference) versus more computation time (to step through the list to find the tail), and the decision of whether or not to maintain a tail reference is usually application dependent.

The real advantage of linked lists compared to an array are apparent if you consider how to insert or delete a node from the list. Inserting a node involves three basic steps:

1. Allocate a new node;
2. Store the data in the new node; and
3. Modify the links in the list to include the new node.

Similarly, deleting a node from the list involves:

1. Modify the links in the list to exclude the node to be deleted;
2. (Optional/if required) Keep a copy the data to return to the user; and
3. Deallocate the node to be deleted.

These operations eliminate any need for the shuffling of elements we have seen previously for arrays (Section 1.6). Similarly, each node is allocated and added to the list separately, eliminating the need to copy elements when resizing the array. It is critical to understand however that the linked list also loses the biggest advantage of an array – the array's use of contiguous memory allows direct access to elements stored anywhere in the list. Moreover, to access the k th element in an array, it is possible to simply specify `array[k]`. The location for this element is then calculated using a simple formula (see Section 1.6). To access the k th element in a linked list however, you must start with a reference to the head node and move to the next node $k - 1$ times. Moreover, a linked list only allows sequential access to the elements stored in its nodes.

2.2.2. Creating and Inserting Nodes into Singly Linked Lists

To implement a singly linked list there are several algorithms we will require. First, we need to initialise the linked list, as follows:

```
headNode ← null
tailNode ← null
```

Algorithm 2.1: Initialising a Linked List

The head and tail variables are set to a value of a ***null*** to indicate that there is no node in the list that is the head or tail node[5], i.e., the list contains no nodes (is empty). To store the first node in the list, the algorithm is relatively straightforward, we simply need to allocate and initialise the new node before setting both head and tail, as follows:

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named data contains the element to be stored
newNode ← allocate node
newNode.data ← data
newNode.next ← null
headNode ← newNode
tailNode ← newNode
```

Algorithm 2.2: Storing the First Element in a Singly Linked List

[5] Note that these algorithms will demonstrate the use of the tail reference for convenience. Eliminating the reference to the tail is trivial, remove all references to updating the tail node, and where a reference is required to the tail, traverse the list as discussed earlier in this chapter.

To store further nodes in the list, there are three possible scenarios for where the new node could be stored: before the head node, after the tail node, or between two nodes. We examine each of these scenarios in turn, however as we will see later, these can be combined into a single algorithm.

To store a new node before the current head, the new node must refer to the current head before the head variable is updated, as follows:

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named data contains the element to be stored
newNode ← allocate node
newNode.data ← data
newNode.next ← headNode
headNode ← newNode
```

Algorithm 2.3: Storing a New Element Before the Head of a Singly Linked List

To store a new node after the current tail, the current tail must refer to the new node before the tail variable is updated, as follows:

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named data contains the element to be stored
newNode ← allocate node
newNode.data ← data
newNode.next ← null
tailNode.next ← newNode
tailNode ← newNode
```

Algorithm 2.4: Storing a New Element after the Tail of a Singly Linked List

Finally, to store a new node between two other nodes is slightly more complex. We no longer require the variables referring to the head or tail, instead we require variables referring to the nodes that will appear before and after the new node. In particular, the next reference in the new node will need to be set to the node that should appear after it, and the node that should appear before it will need to have its next node set to the new node, as follows:

```
ASSUMPTION: A variable prevNode contains a reference to the node to appear before the new node
ASSUMPTION: A variable nextNode contains a reference to the node to appear after the new node
ASSUMPTION: A variable named data contains the element to be stored
newNode ← allocate node
newNode.data ← data
prevNode.next ← newNode
newNode.next ← nextNode
```

Algorithm 2.5: Storing a New Element between Two Nodes in a Singly Linked List

The question that we have not resolved in the algorithm to store a new node between two other nodes is how to obtain the ***prevNode*** and ***nextNode*** references. The answer can be found by considering how to traverse through a linked list. To traverse a linked list, we use a variable to refer to the current node we are examining, which starts at the head node and steps through each node until it reaches the end of the list, i.e.,

```
ASSUMPTION: A variable named headNode exists and refers to the list head
currNode ← headNode
while NOT currNode = null
    // process the current node
    currNode ← currNode.Next
end-while
```

Algorithm 2.6: Simple Traversal of a Linked List

In this example, we have simply stepped through the nodes in the list. However we can adapt this code to terminate upon reaching the desired location, which could either be a numerical position, e.g., insert a node into the fifth location, or could be done for an ***ordered linked list***, as we will illustrate. Given that we want two references to the list, we will also introduce a new variable to keep track of the previous node, as follows:

```
ASSUMPTION: A variable named headNode exists and refers to the list head
ASSUMPTION: A variable named data contains the element to be stored in the list
prevNode ← null
currNode ← headNode
while NOT currNode = null
    if data < currNode.data
        break
    end-if
    prevNode ← currNode
    currNode ← currNode.Next
end-while
```

Algorithm 2.7: Traversal of a Singly Linked List for Constructing an Ordered Linked List

From the above code, ***currNode*** would contain the value for the ***nextNode*** used in Algorithm 2.5, above. Importantly, there are four possible termination scenarios for the above traversal:

- Both ***prevNode*** and ***currNode*** are set to ***null*** – the list is currently empty (it is usually simpler to check for this separately, as we will see below);
- ***prevNode*** is ***null*** and ***currNode*** refers to the head of the list – the new node must be inserted before the current head node;
- ***currNode*** is ***null*** and ***prevNode*** refers to the tail of the list – the new node must be added after the current tail node;
- Both ***prevNode*** and ***currNode*** are not null – we are inserting between two elements.

Combining all of the algorithms above, we can form a single algorithm for inserting a node into an ordered linked list, shown in Algorithm 2.8.

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named data contains the element to be stored in the list
newNode ← allocate node
newNode.data ← data
newNode.next ← null
if headNode = null // empty list
        headNode ← newNode
        tailNode ← newNode
else
        prevNode ← null
        currNode ← headNode
        while NOT currNode = null
                if data < currNode.data
                        break
                end-if
                prevNode ← currNode
                currNode ← currNode.Next
        end-while
        if prevNode = null // insert before head
                newNode.next ← headNode
                headNode ← newNode
        else if currNode = null // insert after tail
                tailNode.next ← newNode
                tailNode ← newNode
        else // insert between two nodes
                prevNode.next ← newNode
                newNode.next ← currNode
        end-if
end-if
```

Algorithm 2.8: Complete algorithm for inserting a node into an ordered singly linked list

2.2.3. Deleting Nodes from a Singly Linked List

Deleting nodes from a singly linked list follows a similar pattern to insertion whereby the operation is fundamentally the same but we consider all of the following scenarios:

- Deleting a node from the middle of the list;
- Deleting the node that is currently the head of the list;
- Deleting the node that is currently the tail of the list; and
- Deleting the last node in the list (list will become empty).

Rather than separate each of the elements as for insertion, we will simply describe the major elements of the algorithm:

1. Begin by traversing the list to locate the element to be deleted, keeping track of both the previous and current nodes as before;
2. If the element is not found, an error has occurred (return error message or throw exception);
3. If the head and tail variables refer to the same node, the last node is being deleted from the list (both head and tail references need to be reset to null);
4. If the previous variable is set to null, the head node is being deleted from the list;
5. If the current node's next variable is set to null, the tail node is being deleted from the list; and

6. If neither the previous or current node's next variables are set to null, we are deleting a node from the middle of the list.

The actual algorithm is shown in Algorithm 2.9. Note that this algorithm would work for any singly linked list, however in the case of an ordered linked list the traversal component of the algorithm could be improved by allowing for the ordering of the list, as follows:

- If the element to be deleted should appear after the tail node, it does not exist in the list; and
- Adjust the terminating condition for the while loop to terminate once the data in the current node should appear after element to be deleted.

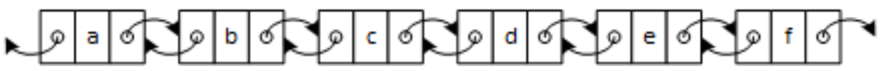
Note that with the new termination condition, the traversal loop may now terminate either upon finding the correct element or upon reaching an element that should appear after the element to be deleted. Thus, it is also necessary to check if the data in the current node matches the element to be deleted after the loop terminates.

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named data contains the element to be stored in the list
prevNode ← null
currNode ← headNode
while NOT currNode = null
    if data = currNode.data
        break
    end-if
    prevNode ← currNode
    currNode ← currNode.Next
end-while
if currNode = null
    // error state, data not found
else
    if headNode = tailNode // deleting last node in the list
        headNode ← null
        tailNode ← null
    else if prevNode = null // deleting the head node
        headNode ← currNode.next
    else if currNode.next = null // deleting the tail node
        prevNode.next ← null
        tailNode ← prevNode
    else // deleting node from middle of list
        prevNode.next ← currNode.next
    end-if
    deallocate currNode
end-if
```

Algorithm 2.9: Complete algorithm for deleting a node from a singly linked list

2.3. Doubly Linked Lists

Fundamentally, the doubly linked list uses the same techniques as a singly linked list, except two links are now stored in the node structure, next and previous. The link to the next node is unchanged from the singly linked list. The link to the previous performs the same purpose as the link to the next node, except for the obvious difference that it links to the node that appears before, i.e.,



The benefits of using a doubly linked list over a singly linked list are not immediately apparent. In particular, the addition of the link to the previous node increases the complexity of some operations while decreasing the complexity of others, while additional space is taken by the additional link. For this reason, the choice of a singly or doubly linked list is usually application dependent. The most obvious advantage is that it is now possible to traverse the list in both directions, i.e., it is possible to start at the tail of the list and step through each node until reaching the head of the list. The biggest advantage however is that given a reference to any node in the list, it is possible to insert a new node before that node, insert a node after that node, or delete that node from the list in only a few short steps – there is no need to traverse the list to determine the location of the previous node, you simply use ***node.prev***, as shown in the following examples.

Insertion of a node before the current:

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named currNode refers to the node to appear after the new node
ASSUMPTION: A variable named data contains the element to be stored
newNode ← allocate node
newNode.data ← data
newNode.next ← currNode
newNode.prev ← currNode.prev
currNode.prev ← newNode
if newNode.prev = null // inserting new head
        headNode ← newNode
else
        newNode.prev.next ← newNode
end-if
```

Algorithm 2.10: Storing a new element before a referenced node in a doubly linked list

Note the use of the statement:

newNode.prev.next ← newNode

in the previous algorithm. This statement uses the reference to the previous node (***newNode.-prev***) to set the value of its next reference to the new node, i.e., start at ***newNode***, navigate to ***prev***, and set ***next*** to be ***newNode***.

Insertion of a node after the current follows a similar pattern:

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named currNode refers to the node to appear before the new node
ASSUMPTION: A variable named data contains the element to be stored
newNode ← allocate node
newNode.data ← data
newNode.next ← currNode.next
newNode.prev ← currNode
currNode.next ← newNode
if newNode.next = null // appending new tail
        tailNode ← newNode
else
        newNode.next.prev ← newNode
end-if
```

Algorithm 2.11: Storing a new element after a referenced node in a doubly linked list

Finally, deletion of a referenced node seems almost trivial:

```
ASSUMPTION: Variables named headNode and tailNode exist and refer to the list head and tail
ASSUMPTION: A variable named currNode refers to the node to be deleted
if currNode.prev = null // deleting head
        headNode ← currNode.next
else
        currNode.prev.next ← currNode.next
end-if
if currNode.next = null // deleting tail
        tailNode ← tailNode.prev
else
        currNode.next.prev ← currNode.prev
end-if
```

Algorithm 2.12: Deleting a referenced node from a doubly linked list

Note that the above code works for all scenarios previously identified for deleting a node from a singly linked list.

Task 2.1. Linked Lists Using Dynamic Memory

Objective: Mastering the creation of linked list data structures is a critical skill in order to (a) successfully study more advanced data structures and (b) to successfully undertake a career in IT involving programming. The average programmer will implement linked lists in their programs many hundreds or thousands of times throughout their career. This task will require you to gain an understanding of linked lists and how they are implemented.

Hint: Before implementing a linked list for the first time it is often worth drawing out what you are trying to achieve before writing the code. This approach ensures you understand what steps are required in your code and the possible scenarios that must be allowed for.

For this task you are required to develop a generic collection `class OrderedLinkedList`, using the C# programming language, which allows data to be stored in a doubly-linked list that is maintained in sorted order. Note that you will need to implement this class using generics, and stipulate that the data type must implement the `IComparable<>` interface (see Section 1.7).

The features of this class represent are similar to those required in Task 1.1, however there are variations in the requirements:

- Namespace:
  - `namespace SIT221_Library`
    Your class should be defined in the namespace `SIT221_Library`;
- Node structure:
  - An appropriate node structure must be declared using a nested class;
- Properties:
  - `public TYPE this[int index]`
    A read/write indexer property which either retrieves a value stored in the doubly linked list at the specified index or replaces the value stored at a particular index;
    Hint: When a value is changed its location in the list may also need to change. To simplify this operation, remove the node from the list, change its value, then reinsert the node in the correct location.
  - `public int Count`
    A read-only property which returns how many elements are currently stored in the doubly linked list;
- Constructors:
  - `public OrderedLinkedList()`
    A parameter-less constructor which initialises the doubly linked list;
- Methods:
  - `public void Add(TYPE element)`
    A method Add which accepts a single element and stores it in the doubly linked list in sorted order;
  - `public void Clear()`
    A method Clear which removes all elements stored in the doubly linked list;
  - `public bool Contains(TYPE targetValue)`
    A method Contains which searches the doubly linked list to determine whether a value is stored in the doubly linked list (returns true) or not (returns false);
    Hint: You will need to use the object.Equals() method to determine if the values are equal or not.
  - `public bool Remove(TYPE targetValue)`
    A method Remove which deletes the first occurrence of the specified target value from the linked list and returns true, or returns false if the target value did not appear in the list;
- Exceptions:
- It is possible that the code using your class may specify an invalid index or other value when using your class. Whenever this occurs, you should throw an ArgumentOutOfRangeException object with an appropriate descriptive message or an IndexOutOfRangeException object.
- You may create any private elements (attributes/methods) as required.

Finally, test your code using the `Main` method provided in this week's provided code (see ZIP file in CloudDeakin).

3

3. Programming Languages and Code Libraries

One of the core concepts of computing is abstraction. In our pre-requisite studies we have already considered abstraction in terms of object-oriented development, where we focus on important concepts while ignoring (hiding) the irrelevant concepts. However abstraction extends much further. For example, consider the physical hardware components in a computer versus the virtual worlds presented in modern computer games. These virtual world represent many levels of abstraction over the physical hardware components which work with binary ones and zeros. In this chapter, we will look behind the abstractions we are familiar with in the software development world to expose the underlying components and how they are used to present us with the environment that we are used to. In particular, we begin by examining how the abstraction presented to us by programming languages is converted into machine executable code by examining the operation of compilers.

The focus of this chapter then shifts to the concept of libraries, which we begin by explaining how programming languages consist of two major elements: the language itself and a library of reusable classes that is provided with the language. In examining libraries further, we explain what libraries are used for and how they are constructed, before undertaking the task of developing our own libraries and associated reference documentation.

Objectives

In this session you will learn

- How compilers convert high level programming languages to machine executable code;
- How programming language functionality is divided between the language itself and their standard library;
- The concept of a library and what they are used for;
- How to construct your own libraries and link them to your applications; and
- How to prepare reference documentation for your libraries using Doxygen.

3.1. How Programming Works

Building software using a programming language represents a task that is completed entirely within an abstract world. Consider the concept of a string which we use regularly throughout programming to represent names, titles, descriptions, and so on. Most likely you have already recognised that a string is in fact just an abstract concept formed from a sequence of characters, but to a computer, a character itself is just another number. Character numbers are then mapped to fonts and/or bitmaps that are drawn on the screen to allow us to read and write with. The actual numbers used for characters in modern computing are typically defined by the Unicode standard which uses 16 bit unsigned integers used to represent characters for many languages including Latin, Greek, Coptic, Cyrillic, Hebrew, Arabic, unified Han, and so on.

To understand how programming works, it is necessary to first understand how computation works on a CPU. Every CPU can perform a collection of operations referred to as the instruction set. The instruction set defines a number of operations which require zero or more operands (parameters). These operations are defined in ***machine code***, where each operation is represented by a number. Given that instruction sets usually contain many different operations, and that people aren't very good at remembering so many different numbers, the first level of abstraction is usually ***assembly language***, where each instruction is given a symbolic name, e.g., one common instruction is `movl`, which represents the operation to move (copy) a long number (usually 32 bits). A CPU may actually provide several variations of an operation to define different types of parameters, e.g., to copy from register to register, register to RAM, RAM to register, RAM to RAM, and so on, each variation forming a separate operation number in machine code. Before a program written in assembly can be executed on a CPU, it must first be translated to the native machine code by an ***assembler***. This is simply a matter of translating each assembly instruction and any operands into the appropriate machine code.

The machine code is generally referred to as a first-generation language (1GL), and assembly as a second-generation language (2GL)[1]. Programming languages like C#, Java, and C++ then represent the next level of abstraction, known as third-generation languages (3GLs). Before programs written in 3GLs can be executed on a CPU, they also must undergo translation to the native machine code. 3GLs include powerful mechanisms for modularising code including methods, classes, separate code files, and code libraries, resulting in a much more complex process of translation than the simple one step process for assembly. Older 3GLs like C or C++ used the following process[2]:

1. ***Preprocessing*** – the output of which is still code written in the same 3GL but with some changes, involves processing of simple directives used in code to import content from separate files, to select which code should be included for a particular build (e.g., debug versus release builds), to apply macros (trivial expansions of common code structures), and so on;

[1] Note that the 'nth-generation' terminology is no longer widely used, however it does help to provide clarity to our discussions here.

[2] We examine these steps for older languages because they are broken down into clear steps that are easier to follow and comprehend. Newer languages, such as C#, implement many/all of the same concepts, but differently.

2. ***Compilation*** – the output of which is usually 2GL/assembly code, involves translating the 3GL instructions and control structures into matching 2GL/assembly instructions, however many elements of the code will remain unresolved, e.g., a method being called will still be identified by its name (must be translated to its address in the final machine code);
3. ***Assembly*** – the output of which is object code, whereby the assembly code is translated to machine code similar to the process we identified above, however unresolved references still remain;
4. ***Linking*** – the output of which is an executable file, involves combining all of the object files and any referenced libraries together, eliminating any unresolved references (by substituting method names for their final address in memory, etc.), and writing the result to a single file that can be interpreted by the operating system to create the initial memory image of the running program.

This whole process is often referred to generally as compiling, although as you can see above compilation is only one part of a larger process. Modern programming languages such as C# and Java follow a similar process, however the final output from these systems is not code that can be directly executed by the CPU, instead a language is used that is very close to the final CPU instructions is used and the supporting environment then makes the final translations. In particular, code written for Microsoft.Net, including C#, is translated to Common Intermediate Language[3] (CIL) which is then executed on the Microsoft.Net Common Language Runtime (CLR). Similarly, Java code is translated into Java Bytecode for execution on the Java Virtual Machine.

Intermediate language code (either CIL or Java Bytecode) are translated to actual machine code for the CPU immediately before executing in a process that is known as Just In Time (JIT) compilation. This last step translation is often incorrectly blamed for poor performance of applications written in C# and Java however once the final translation has happened for the relevant code blocks, it does not need to be translated again. In most cases, observed poor performance of C# and Java applications usually is the result of programmers who don't properly understand how the managed environments works, e.g., garbage collection is one element of the C# and Java managed environments.

The result of compiling a C# program can be seen by using the Intermediate Language Disassembler tool (usually referred to simply as ILDASM). This tool takes the compiled executable and shows the MSIL instructions contained within. For example, consider the following simplest of programs:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
```

[3] The Common Intermediate Language was formerly known as the Microsoft Intermediate Language (MSIL)

```
    }
```

for which the matching MSIL produced by ILDASM is:

```
.method private hidebysig static void Main(string[] args) cil man-
aged
{
    .entrypoint
    // Code size 11 (0xb)
    .maxstack 8
    IL_0000: ldstr "Hello World!"
    IL_0005: call void [mscorlib]System.Console::WriteLine
(string)
    IL_000a: ret
} // end of method Program::Main
```

3.2. Programming Language versus Code Library

Having learned how programming works in general, it is also important to understand where the functionality that we work with comes from. In considering the simple Hello World program above, it is natural to think that all of the code is defined by the C# programming language. In fact, there are two elements being used above, both the programming language, and the standard library for C#. The actual specifications for the C# programming language are available in a document from:

http://msdn.microsoft.com/en-us/library/ms228593

If you consider the specifications document you will notice that it has the following chapters[4] and their major features:

1. Introduction – a basic overview of the major elements of the C# language;
2. Lexical structure – basic statement structures, identifiers (variable names, etc.), literals, and preprocessing directives;
3. Basic concepts – application start-up (Main method) and termination, membership accessibility, scope, namespace introduction;
4. Types – value types and reference types, boxing and unboxing;
5. Variables – static versus instance variables, arrays, pass-by-value/by-reference, how/when values are assigned to a variable;
6. Conversions – implicit (data type promotion) and explicit (casting etc.) data type conversions;
7. Expressions – static versus dynamic binding (polymorphism), method overloading, accessing class/object members, this and base keywords, increment and decrement operators, the new operator (for creating objects), arithmetic operators, logical operators;
8. Statements – code blocks, variable declarations, control structures (if, while, for, etc.), throw and try/catch;

[4] The C# programming language is often updated with each new major release of Visual Studio. Although these changes tend to be minor or even obscure, it is possible these chapters will change in future versions. The chapter list presented here is current as of July 2013 / Visual Studio 2012.

9. Namespaces – using statements, namespace declarations;
10. Classes – class declarations, abstract/sealed/static classes, nested types;
11. Structs – struct declarations, differences between classes and structs;
12. Arrays – types of arrays, creation of arrays;
13. Interfaces – interface declarations, implementing interfaces;
14. Enums – enumerated data types;
15. Delegates – delegate declaration, instantiation, and invocation;
16. Exceptions – causes of exceptions, how exceptions are handled;
17. Attributes – defining and using attribute classes (used for meta data about classes, members, etc., not attributes as in data members of classes);
18. Unsafe code – rarely used advanced coding concepts including use of pointers.

What you may have noticed reading through this list is that there is an absence of common functional elements we use in programming including console input/output, files, GUI functionality, and so on. This is because these elements are provided by a library of code (***code library***) that is provided with the programming language generally referred to as the ***standard library***. The standard library for C# is defined by/provided by the Microsoft.Net class library. The Microsoft.Net class library is also used for other programming languages, the most well known including Visual Basic.Net, Visual F#, and Managed C++[5]. Major features of the Microsoft.Net library (and therefore the C# standard library) include:

- `System` – commonly used functionality, e.g., console input and output, garbage collector, Array class, Exception class (and common exception types), Math class, String class;
- `System.Addin` – support for application plug-in functionality;
- `System.Collections` – reusable collections using the object base type;
- `System.Collections.Generic` – reusable collections using parameterised types (generics);
- `System.Data` – database connectivity;
- `System.GDI` – basic graphics (both 2D and vector graphics);
- `System.Globalization` – cultural information (country, language, dates, currency, etc.);
- `System.IO` – basic support for input and output, including files and directories;
- `System.Linq` – Language Integrated Query (LINQ);
- `System.Media` – basic audio functionality including system configured sound files;
- `System.Net` – support for network communications including sockets, Multipurpose Internet Mail Exchange (MIME) encoding, and web;
- `System.Printing` – printing support and access to the Windows printing system;
- `System.Runtime` – interaction with the common language runtime[6] (CLR) and COM interop (used for interacting with many non-Micsoft.Net software components, including many operating system components);
- `System.Speech` – speech recognition support;
- `System.Text` – advanced string manipulation functionality including regular expressions;
- `System.Threading` – support for multithreaded applications;

[5] Other programming languages that have been ported to run on Microsoft.Net include Ada, COBOL, Eiffel, FORTRAN, Java, LISP, PASCAL, PHP, Python, and Ruby.
[6] The Common Language Runtime is responsible for managing the execution of a Microsoft.Net executable.

- `System.Web` – functionality for interacting with and producing web content, including HTTP, AJAX, JSON, and web services;
- `System.Windows` – functionality for constructing both traditional windows applications (Win32) and those built using Windows Presentation Foundation (WPF); and
- `System.Xml` – support for XML including serialisation, XSD, XQuery and XPath.

Importantly, code libraries are not only restricted to the standard library bundled with a programming language, libraries are also regularly used to provide additional functionality to an application. For example, three of the most common libraries provided by Microsoft that you will encounter include:

- A collection of classes for writing extensions for Microsoft Office under the `Microsoft.Office` namespace; and http://msdn.microsoft.com/en-us/library/bb771305.aspx
- A collection of classes for writing extensions for Visual Studio under the `Microsoft.VisualStudio` namespace; http://msdn.microsoft.com/en-us/library/bb166217.aspx
- A collection of classes for writing games under the `Microsoft.Xna` namespace; http://msdn.microsoft.com/en-us/library/bb203940.aspx
- Microsoft provides an alternative version of the standard Microsoft.Net library for mobile devices known as the Microsoft.Net Compact Framework. http://msdn.microsoft.com/en-us/library/aa286583.aspx

There are also many libraries provided third party vendors containing various functionality. The largest collection of such libraries are sold as reusable components for developing advanced applications[7], e.g.,

- Data grid components – Visual Studio includes a basic data grid component however users will usually expect functionality similar to what they find in commercial spreadsheet products such as Microsoft Excel;
- Graphing/charting components – Visual Studio includes a basic charting component however this component does not provide as much flexibility as commercially available components;
- Reporting and PDF generation – Visual Studio includes a basic version of Crystal Reports, however it is missing a large amount of functionality available in the commercial version and in competitor products;
- Ribbon UI – with the addition of the Ribbon UI to Microsoft Office and Windows applications (as of Windows 8), many users expect this interface in modern products, however Microsoft currently only provides the Ribbon UI to applications written using MFC in C++ or for applications developed for the Windows Presentation Framework (WPF) GUI;
- Visual Studio add-ins/plug-ins – many tools are available to provide enhanced code editing including additional code refactorings, support for third party version control schemes, etc.; and

[7] There are some free alternatives however in general these are often either low quality or have licensing terms that prevent their use in commercial products. There are a number of high quality popular components that are free however.

- Specialist functionality, e.g., serial port functionality, email automation, network management and system administration functionality, and so on.

All of the components above could be developed by any individual or group of programmers, however there is a need to consider the cost of development versus the cost of purchasing such components ready made. For example, the task of developing a component to provide the Ribbon UI is not in itself particularly difficult, however to do so does a thorough require thorough knowledge of how the Windows UI functionality works at a low level (particularly the Win32 API which is encapsulated/hidden by Microsoft.Net). The time, effort, and risk (from unexpected delays, undiscovered errors, etc.) required to develop and thoroughly test such a component must be compared to the cost of purchasing a pre-built component.

Libraries are an important mechanism for modularising the applications that we write. Well engineered applications have a clear separation between the major functional elements including interface, data management, and business logic, but also potentially other elements such as network communication, storage of data in files/databases, and so on. The ability to construct libraries separate to the application also provides a useful mechanism for dividing the development task for an application into separate development teams, with the final application only needing to be linked prior to testing.

3.3. Developing Libraries for Reusability

There are two aspects that need to be considered when developing libraries, the physical construction of the library (examined in Section 3.4), but it is also important to consider what code is to be placed in the library itself. The code that is found in libraries is no different to what you have already been developing while learning programming. This code may be built for a purpose, e.g., a number of classes, routines, etc., used for interacting with a specific database for an application, or may be built with the intention of being reusable for several projects.

The ability to reuse previously developed code offers a number of advantages, for example:

- A reduction in development time – reused code is already complete and does not need to be developed from scratch;
- A reduction in testing time – reused code has usually been tested thoroughly and can be relied upon in a new project;
- A reduction in time/effort to maintain existing code – bug fixes, etc., to code can quickly and easily be carried to all projects in which it was reused; and
- Improved quality of code – code that has been developed to be reusable will usually have been more carefully designed, coded, and tested.

Moreover, reusability reduces the risks associated with developing software (risk of late delivery, risk of bugs, etc.). However, there are also disadvantages and/or concerns with developing reusable code, for example:

- Reusable code takes longer to develop than purpose-built code;
- Reusable code is sometimes rarely, if ever, reused, thus wasting the extra effort to develop it in the first place;
- Reusable code can take a long time to learn and/or adapt for (or plug-in to) a new for project;

- Reused code can be restrictive, i.e., if the code does not support a particular feature you may not be able to extend/easily extend that code to support that feature; and
- Bugs in reused code will be present in all projects using that code.

In developing classes it is possible to encourage and improve reusability by following a number of simple style rules[8]:

- Keep methods coherent – well written methods have a very clear and distinct purpose in a program. A well written method will perform either a single function or a group of closely related functions;
- Keep methods small – a very long method (greater than one or two pages) is usually a sign of a method that is doing a very specific function that is not particularly reusable. If the function can be broken into smaller parts, it is possible that those individual parts may have a higher level of reusability even if the whole method was not;
- Keep methods consistent – methods that perform the same kind of function should generally have the same method names, argument order, data types, return value, and semantics;
- Separate policy and implementation – keep the code that makes decisions (policy) and the code that performs the logic of the program (implementation) separate methods. Policy methods are application specific whereas implementation methods may be reusable by other applications;
- Provide uniform coverage – if input can be provided in different format/combinations, provide methods to handle all possibilities, not just those formats that immediately required;
- Broaden the method as much as possible – often the reusability of a method can be increased substantially with very few actual modifications, e.g., modify the method to accept different types of data, make fewer assumptions about how the method will be used, provide more meaningful results for empty/extreme/invalid inputs, and so on;
- Avoid global information – avoid/minimise use of data outside of a method as this can change the functionality of the method depending on that external data. This can often be corrected by passing the data as a parameter instead;
- Avoid methods with state – methods that change depending on how they have been used previously are unlikely to be reused. This dependency on method call history can often be eliminated by replacing the method with several methods.

3.4. Building and Linking Libraries

Physically, libraries are somewhat similar to an archive, e.g., a ZIP file, where the index contains a list of object files or symbols (global variables, classes, structs, etc). Recall from Section 3.1 how the last phase of compiling a program is the linking phase, where all of the compiled object files are brought together to eliminate any unresolved references. Normally, after linking the object files from a programming project there will usually remain some unresolved elements

[8] Blaha, M., and Rumbaugh, J., "Object-Oriented Modeling and Design with UML", Second Edition, Prentice-Hall, 2005.

(method calls, etc.). Any libraries referenced by the programming project[9], both the standard library and any other code libraries, are checked for these unresolved elements in the last steps of linking.

Importantly, libraries can be linked in one of two ways:

- Statically – the most obvious approach, whereby the code from the library is copied into the final executable; or
- Dynamically – the code from the library is not copied into the final executable and is instead made available at run-time.

There are some simple differences between these two approaches. The use of statically linked libraries result in a larger executable as at least part of the library has been copied into the file. However, the library is only needed at compile-time, whereas dynamically linked libraries need to be loaded into memory and associated with the program at run-time. Importantly, this also represents the key advantage of dynamically linked libraries, as the library code in memory can be associated with more than one copy of the program at the same time or even associated with multiple programs running in memory. This is made possible by memory abstraction concepts implemented by modern hardware which is used by all modern mainstream operating systems[10].

The concepts of associating a dynamic library with a program at run-time and sharing the library between programs is illustrated in Figure 3.1. In this figure, two processes are shown using a very simplistic model with memory regions for the program code, library code, heap space (used for dynamically allocated memory) and the system stack (used for temporary data, local variables, etc.). Also shown is how these regions have been mapped to physical memory. In particular, the program code, heap, and stacks for each process are stored in separate locations. However both programs reference the same library code in memory, thus reducing the overall demand for memory.

[9] It is possible to see these references in Visual Studio in the Solution Explorer (where the files in your project are listed). When using additional libraries you often need to add references to this list which you may have had to do already when working XML or other technologies.
[10] The concept of memory abstraction is a major topic of the unit SIT222 Operating Systems.

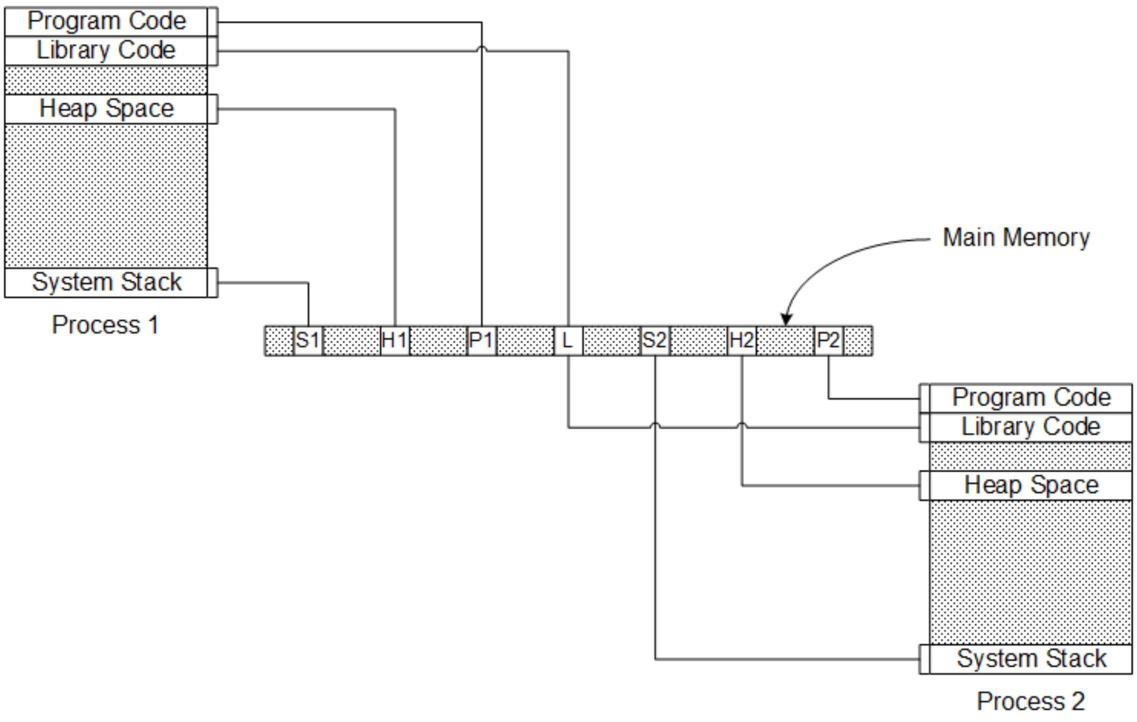
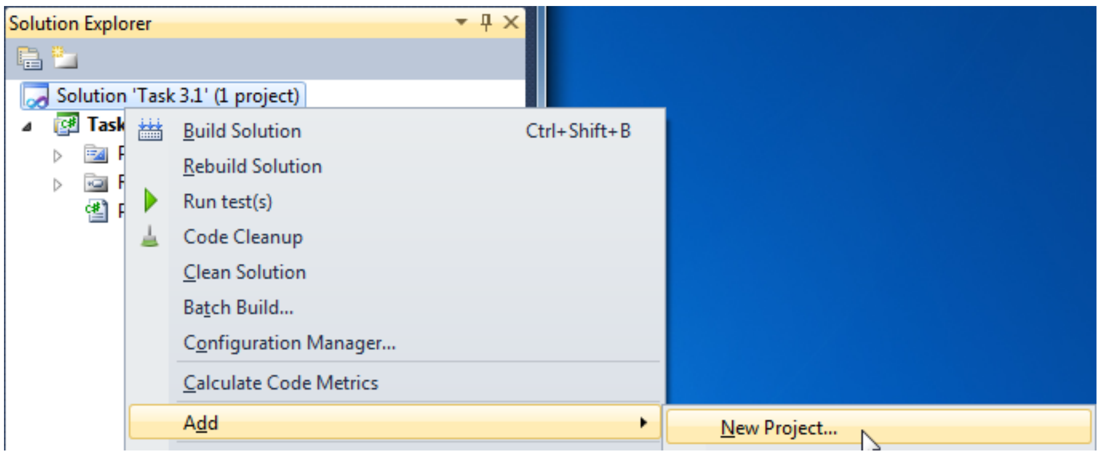


Figure 3.1: Dynamically Linked Libraries

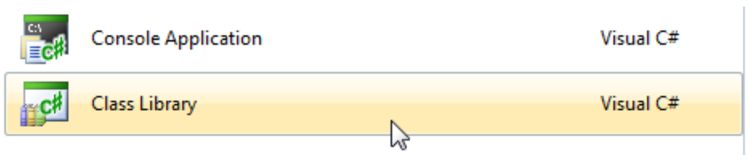
Importantly, dynamically linked libraries also have their own problems. If a dynamically linked library is updated while there are active programs using the library, they may crash unexpectedly. Additionally, when developing an application it is not unusual to find an error in a library and develop a solution in your application to work around the error – if the dynamic library is then updated to correct the error, your application may no longer work as expected without further corrections. This was one of the factors that led to what was commonly referred to as "DLL Hell" in Windows.

Working with Libraries in Visual Studio

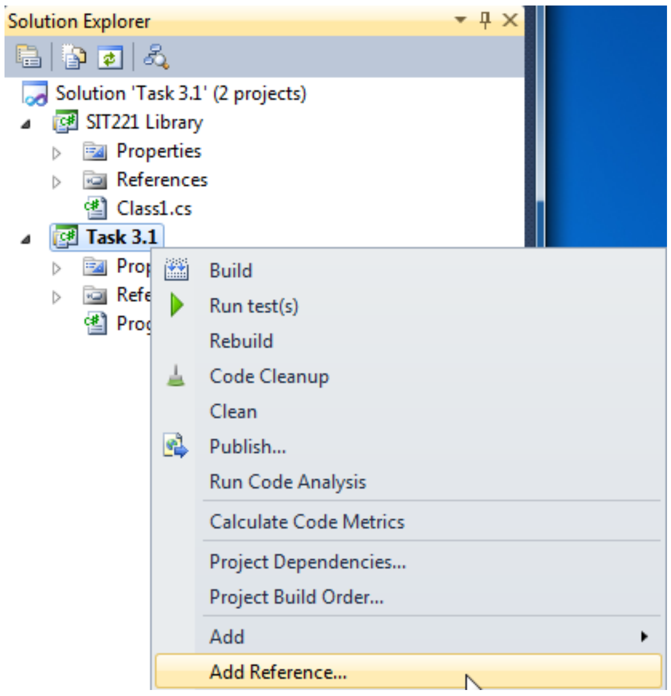
To create a dynamically linked library using C# in Visual Studio, begin by right clicking on the solution entry in the Solution Explorer, then select Add, and New Project...



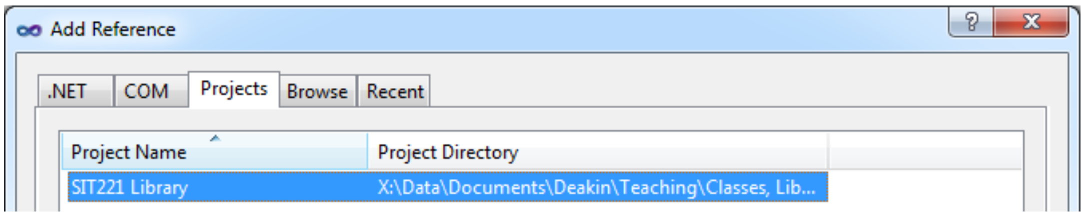
Select the Class Library project type, give it an appropriate name, e.g., SIT221 Library, and click on the OK button.



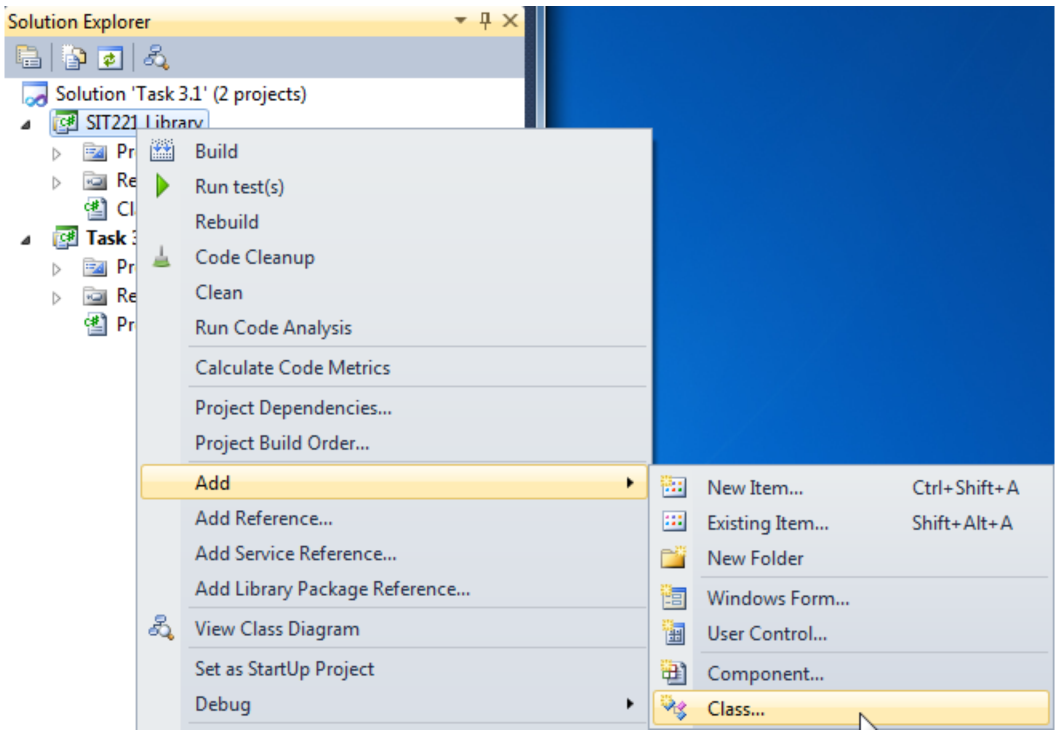
This creates and adds a library project to your current solution. The last step is to tell Visual Studio to include your new library when linking the application's executable. To do so, right click on the application's project entry in the Solution Explorer and select Add Reference...



By default, the window that pops up should already be selecting your new library, but if not click on the Projects tab and select your new library, then click OK.



At this point you can now begin adding and developing classes in the library by right clicking the library project and adding classes.



The only thing to remember when developing with libraries is that the classes of the library will be in a different namespace to those in the application project, and you will need to add a using statement to your application code files, e.g., assuming you named your library SIT221 Library, the following using statement would be required

```
using SIT221_Library;
```

3.5. Library Documentation

So far we have seen how code libraries are used, how to construct them, and how to link them with our applications. One of the most important elements however is the need to provide quality documentation for your libraries. Documentation primarily falls into two categories:

1. Usage manuals/documentation – provide instructions for a programmer on how to get started using the library, e.g., how to initialise the library, how to achieve basic tasks using the library, introduction to the major components of the library, etc. This documentation is common for libraries that provide major functionality to an application such as graphics engines (e.g., DirectX), database engines (how to connect and query the database), and so on.
2. Reference documentation – provide an overview of each class (or other element) that can be found in the library including information and examples on how those elements are used.

In this session, we will only consider the development of reference documentation. Reference documentation for the Microsoft.Net Framework can be found at:

http://msdn.microsoft.com/en-us/library/gg145045

For comparison, you may also wish to consider the Java API reference:

http://docs.oracle.com/javase/7/docs/api/index.html

Regardless, if you examine either of these references, you will notice some or all of the following elements in the documentation for each class:

- A description of the class, its purpose, and any relevant general background information;
- A description of each of the members (attributes, properties, methods) of each class;
- One or more example programs demonstrating how the class can be used and basic tasks;
- Information describing any changes made to the class over time;
- An indication of whether or not the class is considered thread safe, and any implications of using the class with threads[11]; and
- A list of related classes, functionality, and/or documentation ("see also").

Other code elements may have additional documentation if relevant, e.g., for a method you would usually also document each of the parameters and the return value.

It is possible to write library reference documentation using a word processor or HTML editor, however there are a number of useful tools available to help automate this process. In this unit we consider the use of Doxygen[12], a popular and free software tool for generating such documentation:

http://www.stack.nl/~dimitri/doxygen/

Doxygen has direct support for XML comments that can be written into a C# program, and uses the information written by a programmer in these comments to generate the documentation. A reference for C# XML comments, including the list of possible tags, is available at:

http://msdn.microsoft.com/en-us/library/b2s063f7.aspx

Getting started with C# XML comments is very simple with Visual Studio. To insert comments in the code with appropriate XML comments, create a blank line before the element you wish to document and insert three forward slashes (`///`), e.g.,

```
///
public bool Contains(TYPE targetValue)
{
    ...
}
```

Visual Studio will automatically expand the forward slashes to a larger comment containing the appropriate XML tags, as follows:

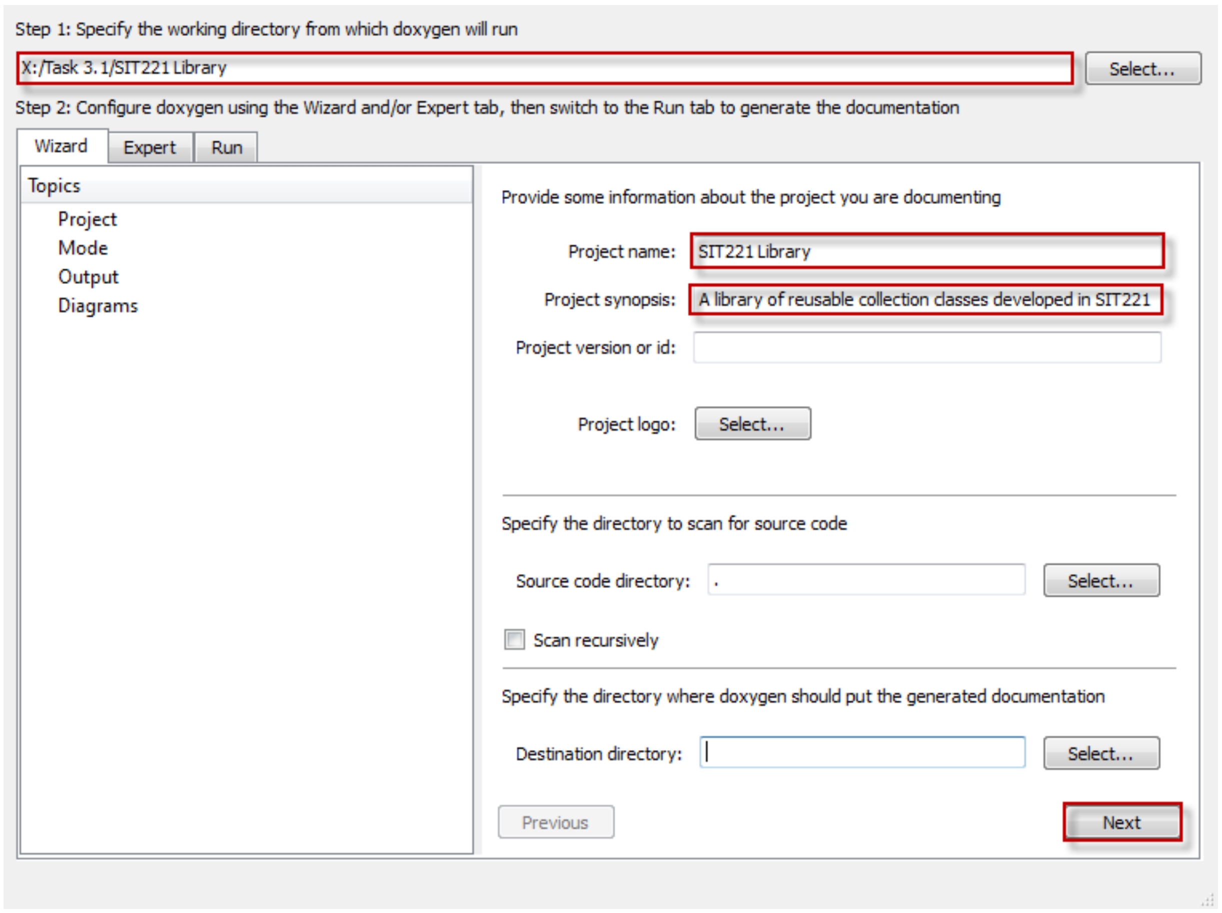
[11] The concept of thread safe code is code which works correctly in multithreaded applications (code that is not thread safe may experience errors due to race conditions and other problems). Further consideration of thread safety is outside the scope of this unit.
[12] A ZIP file containing the main executables for Doxygen is available in CloudDeakin, however you may wish to download the setup file from the Doxygen web site if you are using your own computer. On-campus students may find Doxygen pre-installed on Deakin computers.

```
/// <summary>
///
/// </summary>
/// <param name="targetValue"></param>
/// <returns></returns>
public bool Contains(TYPE targetValue)
{
    ...
}
```

Note in the above example for a method, the following tags have been created:

- `summary` – used to provide a brief description of the method;
- `param` – used to provide a brief description of each parameter to the method; and
- `returns` – used to provide a description of the different return values for the method.

Once you have documented all of your code using these comments, you can then use the Doxygen tool to extract the documentation. Doxygen itself is a command line tool for generating documentation with many advanced features, however the simplest way to get started is to use the provided GUI tool. The following screenshots show a useful set of options to get started with (highlighted by red boxes), however you should experiment with Doxygen and explore the documentation on their web site to properly learn how to use the tool.


Step 1: Specify the working directory from which doxygen will run

X:/Task 3.1/SIT221 Library

Step 2: Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation

Provide some information about the project you are documenting

Project name: SIT221 Library

Project synopsis: A library of reusable collection classes developed in SIT221

Project version or id:

Project logo:

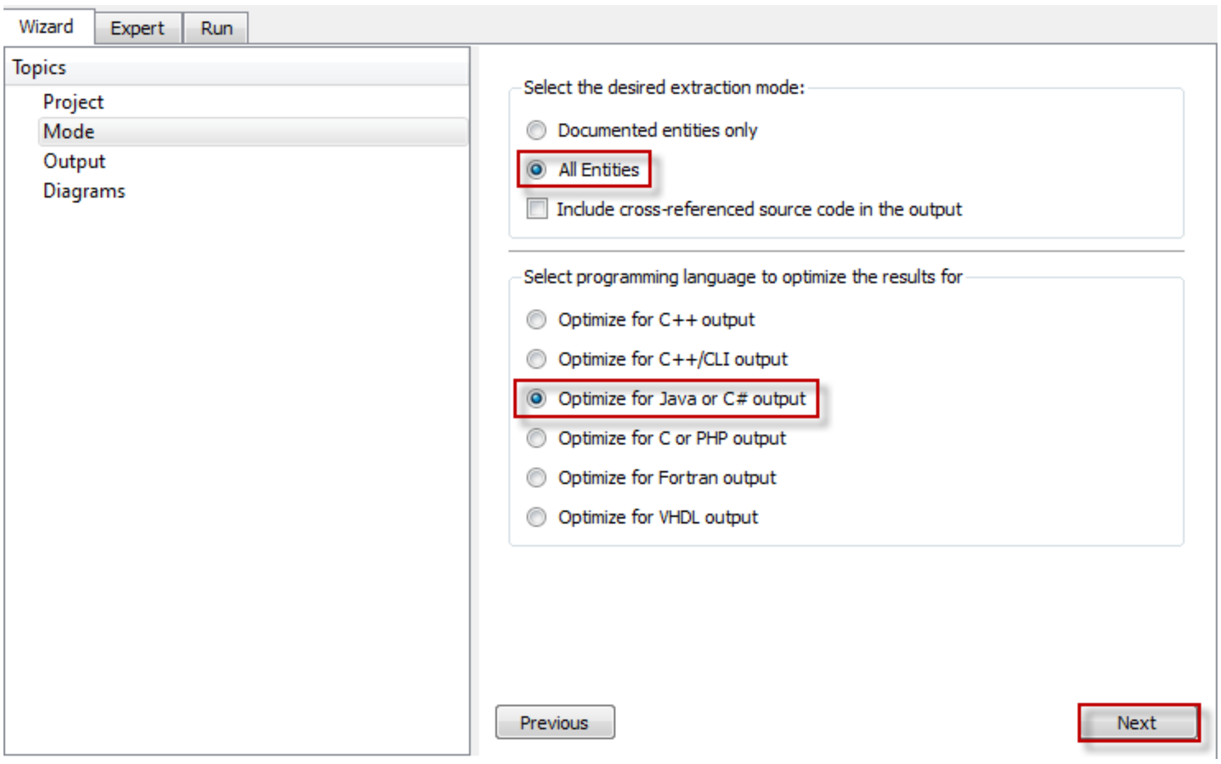
Specify the directory to scan for source code

Source code directory: .

Scan recursively

Specify the directory where doxygen should put the generated documentation

Destination directory:

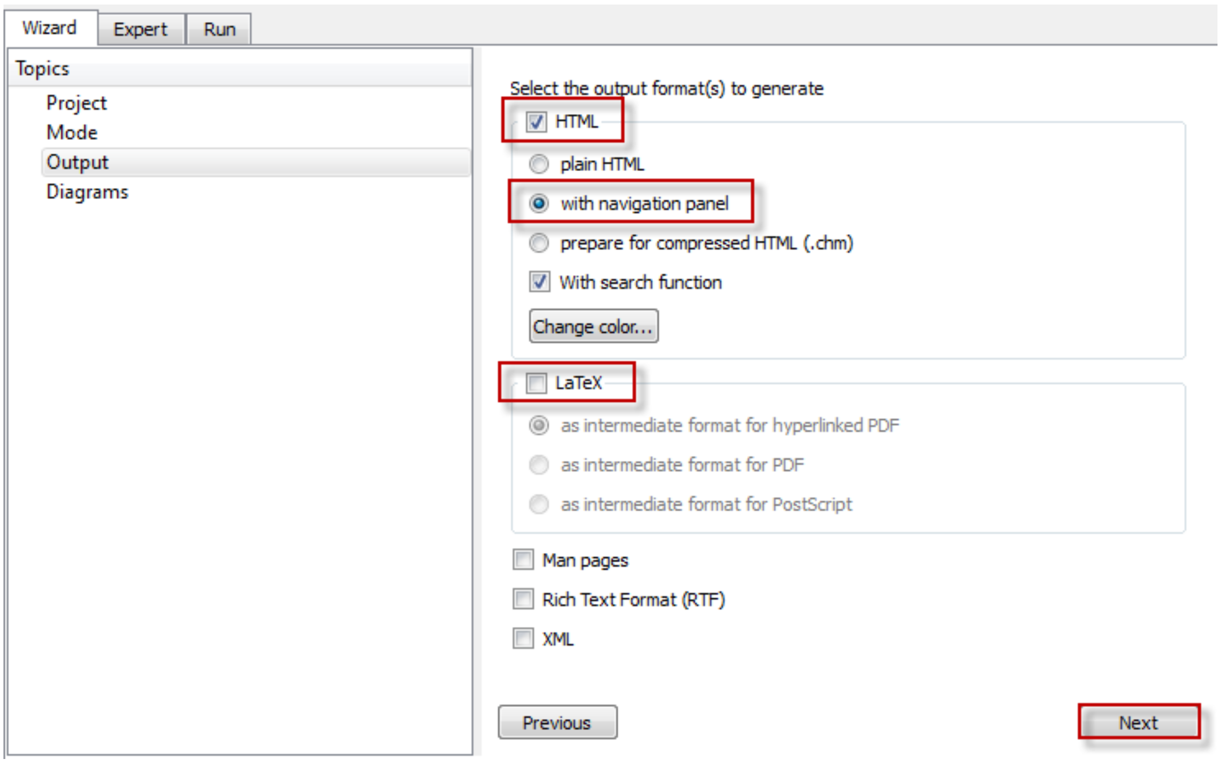


Select the desired extraction mode:

- ○ Documented entities only
- [✓ ● All Entities]
- ☐ Include cross-referenced source code in the output

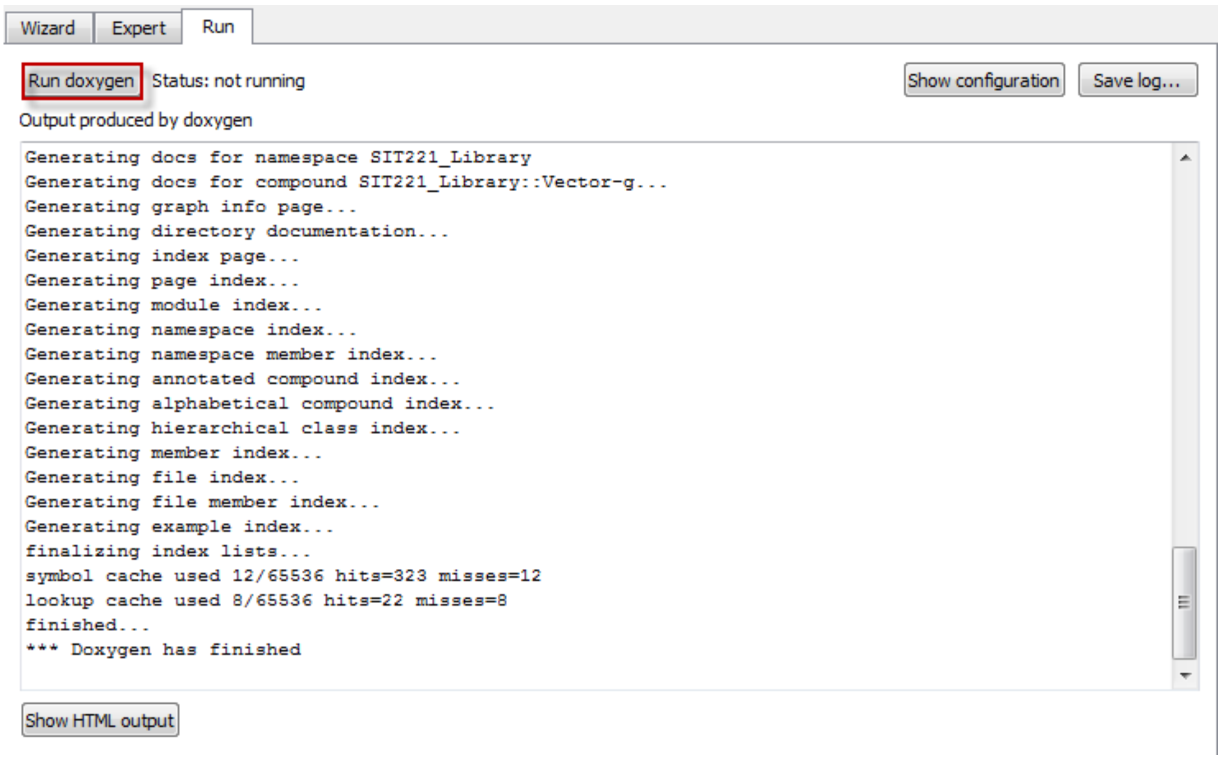
Select programming language to optimize the results for

- ○ Optimize for C++ output
- ○ Optimize for C++/CLI output
- [✓ ● Optimize for Java or C# output]
- ○ Optimize for C or PHP output
- ○ Optimize for Fortran output
- ○ Optimize for VHDL output



Select the output format(s) to generate

- [✓ HTML]
  - ○ plain HTML
  - [✓ ● with navigation panel]
  - ○ prepare for compressed HTML (.chm)
  - ✓ With search function
- ☐ LaTeX
  - ● as intermediate format for hyperlinked PDF
  - ○ as intermediate format for PDF
  - ○ as intermediate format for PostScript
- ☐ Man pages
- ☐ Rich Text Format (RTF)
- ☐ XML



```
Generating docs for namespace SIT221_Library
Generating docs for compound SIT221_Library::Vector-g...
Generating graph info page...
Generating directory documentation...
Generating index page...
Generating page index...
Generating module index...
Generating namespace index...
Generating namespace member index...
Generating annotated compound index...
Generating alphabetical compound index...
Generating hierarchical class index...
Generating member index...
Generating file index...
Generating file member index...
Generating example index...
finalizing index lists...
symbol cache used 12/65536 hits=323 misses=12
lookup cache used 8/65536 hits=22 misses=8
finished...
*** Doxygen has finished
```

Hints:

- On the first screen, set the working directory to the location where your library source files are located;
- Once you have completed the configuration using the Wizard, you can then explore the options on the Expert tab which will reflect your settings;
- Once configured, you can click on the “Run doxygen” button as many times as needed to regenerate the documentation;
- The documentation can be found in a new folder “html” located in the working directory configured on the first screen; and
- Remember to save the Doxygen project so you can call it up again later.

Task 3.1. Build Your Own Library

In this task we will begin constructing a library of reusable collections that we will continue to develop throughout the trimester. Libraries are an critical mechanism for modularity in large projects and although this task is simple, it sets the foundation for our continued study of libraries in the next session.

For this task you are required to construct a library consisting of a single class: the Vector class we developed in Task 1.1. Note that you can use either your own solution or the specimen solution available in CloudDeakin for this purpose. Your tasks are as follows:

1. Create a new project and add a class library to the solution.
2. Copy the Vector class source code into the library.
   Hint: The easiest way to do this using Visual Studio is to add a new class called Vector to the class library, then use copy and paste to replace the contents of the new file with the source code for the provided Vector class.

3. Document the Vector class and all members using XML comments.
4. In the application project, write and test a simple example of how to use the Vector class – the example should demonstrate how to use all public members of the Vector class. Do not use the provided code from earlier weeks for this purpose.
5. Include this example code in the XML documentation for the Vector class in an appropriate location using the correct XML tag.
6. Generate documentation for your class library using Doxygen and experiment with the settings it provides until you are happy with the output.

Note: You will need to refer to the XML tag documentation at http://msdn.microsoft.com/en-us/library/b2s063f7.aspx to determine the correct tags for your code.

4

4. Building Libraries with C# and Microsoft.Net

In the Session 3 we introduced the concept of a library and broadly examined the functionality provided by the standard library (Microsoft.Net). We also examined the basics of how to construct code libraries and how to use them in our programming projects. In this session we extend our study of libraries by examining the collections provided by Microsoft.Net that are ready made for use in our programs. We identify how interfaces are used to ensure consistency, and thereby improve reusability by ensuring the public members of the different collections are the same, wherever possible and sensible. After reviewing the interfaces defined and used by Microsoft.Net for collections, we consider how to use these interfaces when building our own collections. Finally, we examine in detail how to develop classes that support the `foreach` loop in C# by implementing the `IEnumerable<T>` interface.

Objectives

In this session you will learn

- The different collections available in Microsoft.Net;
- How interfaces are applied to ensure consistency in libraries;
- The different interfaces defined and used by Microsoft.Net for collections; and
- How to implement support for the `foreach` loop in your classes.

4.1. Using Pre-Built Collections

Modern programming environments often provide some of the fundamental data structures we use in programming as pre-built collections (reusable classes). Two types of collections are usually seen: collections for storing a number of elements, and collections for storing a number of key-value pairs[1]. The first type is similar to the collections that we are already familiar with, such as an array or linked list.

The second type of collection, storing key-value pairs, effectively stores two pieces of data per element in the collection: the key and the value. Examples of a collections using key-value pairs can be seen if you consider a dictionary or telephone directory. In considering a dictionary, each element consists of a word (the key) and an associated definition (the value). Similarly, telephone directories contain elements consisting of the name of the person/business (key) and their address and/or telephone number (value). Collections for storing key-value pairs usually allow elements to be retrieved quickly by using the key.

Having a programming environment that provides pre-built collections does not negate the need to study how to implement the underlying data structures, on the contrary, our study of using arrays in Session 1 (used by the `List<>` collection) and linked lists in Session 2 (used by the `LinkedList<>` collection), should already have made it apparent that understanding these collections is important.

While it is often possible to save time by using a pre-built collection, it is critical to recognise that different collections have their own advantages and disadvantages that must be considered. For example, if you needed to progressively store a large unknown number of elements, the `LinkedList<T>` class type would provide acceptable/good performance due to its use of a doubly linked list. Using the `List<T>`class on the other hand would result in significantly worse performance due to its use of an array which is progressively reallocated and copied. This represents an important lesson – by undertaking a study of data structures and their implementation, we are able to gain a thorough understanding of how these classes work and where they are best applied in our programs.

In addition to being able to select an appropriate collection for a particular application, knowledge of data structures also allows you to implement your own collections in programming environments where they are not provided in pre-built form, or to develop more complex data structures using a combination of pre-built and/or custom-built data structures. Moreover, it is not always possible to use a built-in collection and professional software developers require the skills to develop different data structures than those provided by a programming language environment.

4.2. Microsoft.Net Pre-Built Collections

Microsoft.Net provides two sets of collections in two separate namespaces:

- `System.Collections` – these collections have been available since the very first release and use the base `object` type to allow the collections to be used for different types of objects. These collections should not be used in new projects, if there is an equivalent collection in `System.Collections.Generic`, for the reasons outlined in Section 1.7;
- `System.Collections.Generic` – these collections were added in the second version of Microsoft.Net and supersede many of the collections those provided in the namespace

[1] In some programming environments, such as Perl, key-value pair collections are called an ***associative array*** or ***associative lists***.

`System.Collections`. These collections use generics (parameterised types) to ensure type safety.

Other languages provide similar collections including Java (the Collections Framework) and C++ (the Standard Template Library). The following table summarises the main collections provided by Microsoft.Net:

Generic Class	Old Class	Description	Reference
n/a	`BitArray`	Used for storing a large collection of true/-false values using a single bit per value.	n/a
`Dictionary<TK, TV>`	`HashTable`	A collection storing key/value pairs using a hash table.	Session 8
`HashSet<T>`	n/a	A collection of unique values stored in a hash table.	Session 8
`LinkedList<T>`	n/a	Stores a collection of values using a doubly linked list structure.	Session 2
`List<T>`	`ArrayList`	Stores a collection of values using an array that is resized as required.	Session 1
`Queue<T>`	`Queue`	A data structure providing First In First Out (FIFO) semantics.	Session 6
`SortedDictionary<TK, TV>`	n/a	A collection storing key/value pairs using a binary search tree.	Session 9
`SortedList<TK, TV>`	`SortedList`	Similar to the `List<>` type but uses an array of key-value pairs sorted by the key.	n/a
`SortedSet<T>`	n/a	A collection of unique values that are stored in a red-black tree.	n/a
`Stack<T>`	`Stack`	A data structure providing Last In First Out (LIFO) semantics.	Session 5

Note: <T> indicates the requirement to specify the type of the objects to be stored in the collection. Similarly, <TK, TV> indicates the requirement to specify both the object type for the keys (TK) and the object type for the values (TV), used in collections that store key-value pairs.

You may also have noticed in reviewing these classes that we have already partially developed two of these classes. In particular:

- The `Vector<T>` class we began developing in Task 1.1 is very similar to the `List<T>` class; and
- The `OrderedLinkedList<T>` class we began developing in Task 2.1 is very similar to the `LinkedList<T>` class, except that the collection we implemented is an ordered linked list. There are also similarities with the `SortedSet<T>` class, however this class does not permit duplicate values and uses a tree data structure instead.

4.3. Applying Interfaces for Consistency

Recall from Section 3.3 one of the key points for improving the reusability of code:

- Keep methods consistent – methods that perform the same kind of function should generally have the same method names, argument order, data types, return value, and semantics;

This is a critical requirement when developing libraries to ensure that they are easy for other programmers to learn and use – it is not practical to have to spend several hours learning each different class in the library. One of the mechanisms provided by modern object-oriented programming languages for ensuring consistency of methods is interfaces.

Recall from the pre-requisite that interfaces are similar to inheritance in a number of respects, which is also sometimes referred to as ***implementation inheritance***. The use of this term illustrates that applying inheritance in the program code results in the implementation of one class, the base class, being duplicated in another, the derived class. Interfaces can similarly be described as ***interface inheritance***, where only the interface (signature of members) is duplicated from the interface to the class that implements that interface. Although our examination of interfaces in this unit focuses on their application to collections, it is important to note that interfaces are used throughout the Microsoft.Net library (and other object-oriented programming libraries) for ensuring consistency.

Simply put, an ***interface*** defines the signature/s for one or more members that must be present in any class that chooses to implement that interface. Any number of interfaces can then be implemented by a class. The syntax for defining an interface in C# is as follows:

```
[access_modifier ]interface name[ : base_interface[, ...]]
{
        interface_member
        ...
}
```

In the above syntax, the *access_modifier* represents the required access level, i.e., `public`, `protected`, or `private`, for members defined by the interface. If more than access modifier is required, i.e., to define members with different access modifiers, separate interfaces must be defined (interfaces can only define members that use the same access modifier). The name of the interface is specified by the *name* field and has the same naming requirements as variables, classes, methods, etc., however the name is usually preceded by the letter '`I`', e.g., `IList` which we will examine in a moment. Finally, the *base_interface* represents one or more interfaces which are inherited by this interface, i.e., it is possible to build an interface using one or more other interfaces as a base.

Interfaces can specify either properties, methods, or both, but not instance variables. The syntax for properties is as follows:

```
type name { [get;] [set;] }
```

and for methods:

```
return_type method_name([parameter[, ...]]);
```

Implementing an interface for a class uses the same syntax as for inheritance, as follows:

```
[access_modifier ]class class_name : interface_name
{
        [access_modifier ]class_member
        ...
}
```

Defining the actual class members is identical to what we have already seen, except the data types and names in the property and method signatures must match.

4.4. Interfaces used by Microsoft.Net for Collections

In examining the collections provided by Microsoft.Net, you will notice that there are a number of common members with identical syntax. For example, consider the following methods that are common to both the `HashSet<T>` and `SortedSet<T>` classes:

- `public bool Add(T item)`
- `public void Clear()`
- `public bool Contains(T item)`
- `public void CopyTo(T[] array)`
- `public void CopyTo(T[] array, int index)`
- `public void CopyTo(T[] array, int index, int count)`
- `public void ExceptWith(IEnumerable<T> other)`
- `public void IntersectWith(IEnumerable<T> other)`
- `public bool IsProperSubsetOf(IEnumerable<T> other)`
- `public bool IsProperSupersetOf(IEnumerable<T> other)`
- `public bool IsSubsetOf(IEnumerable<T> other)`
- `public bool IsSupersetOf(IEnumerable<T> other)`
- `public bool Overlaps(IEnumerable<T> other)`
- `public bool Remove(T item)`
- `public bool SetEquals(IEnumerable<T> other)`
- `public void UnionWith(IEnumerable<T> other)`

The consistency in these common methods has been imposed through the application of interfaces for these classes. If you were to implement the same interfaces (and semantics) in your own classes they will have the same look and feel as the Microsoft.Net collections. This means other developers familiar with the above collections would then be able to start using your classes immediately. In this section, we will examine five of the most common interfaces implemented by Microsoft.Net collections, as illustrated in the UML class diagram in Figure 4.1. In considering this diagram, notice that these interfaces are actually related by inheritance. This simplifies implementing these interfaces somewhat as the collection needs to only select from the `IDictionary<>`, `IList<>`, and `ISet<>` interfaces, as we will see.

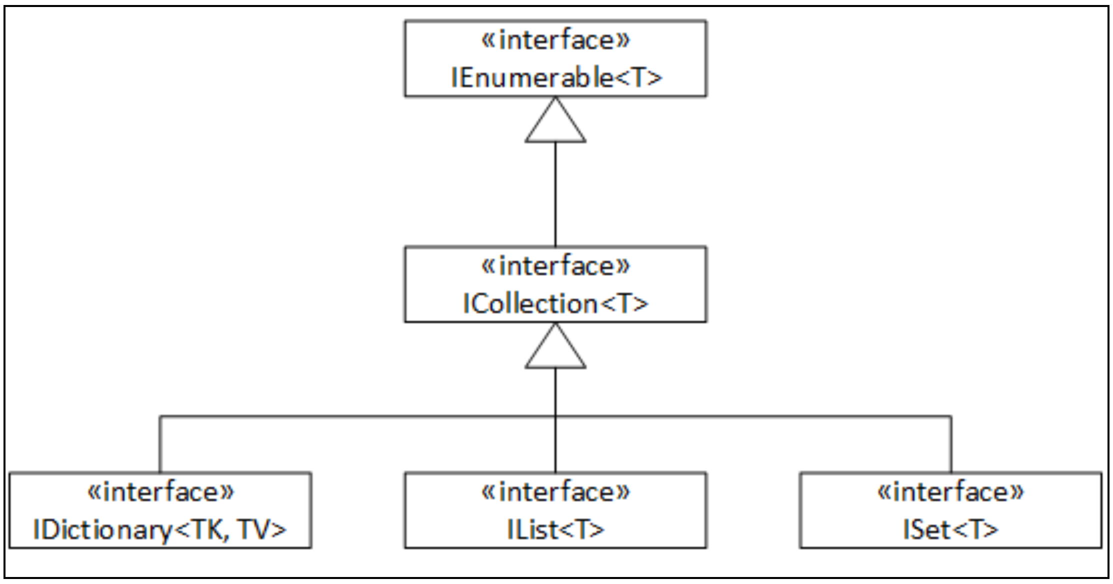


Figure 4.1: Interface Hierarchy used in Microsoft.Net Collections

The following members are defined by the above interfaces are as follows:

- `ICollection<T>` – implemented by all collections, provides basic functionality for manipulating the elements stored in a collection. Important members:
  - `int Count { get; }` – read-only property that indicates the current number of elements stored in the collection;
  - `bool IsReadOnly { get; }` – read-only property that indicates whether the collection can be modified or not;
  - `void Add(T item)` – add an element to the collection;
  - `void Clear()` – remove all items stored in the collection;
  - `bool Contains(T item)` – test to see if an item is stored in the collection;
  - `void CopyTo(T[] array, int index)` – copy elements stored the collection to a provided array; and
  - `bool Remove(T item)` – remove the first element found in the collection with the specified value.

- `IDictionary<TK, TV>` – implemented by all collections storing key/value pairs as described in Section 4.1. Important members:
  - `TV this[TK key} { get; set; }` – property that allows the value for the element with the specified key to be retrieved/modified;
  - `ICollection<T> Keys { get; }` – read-only property that provides the keys of all elements stored in the collection;
  - `ICollection<T> Values { get; }` – read-only property that provides the values of all elements stored in the collection;
  - `void Add(TK key, TV value)` – adds a key/value pair to the collection;
  - `bool ContainsKey(TK key)` – tests to see if there is an element stored with the specified key;
  - `bool Remove(TK key)` – removes an element in the collection with the specified key; and

    - `bool TryGetValue(TK key, out TV value)` – retrieves a value from the element with the specified key (if successful).

- `IList<T>` – implemented by collections which can be accessed by an index. Important members:

    - `T this[int index] { get; set; }` – property that allows values for the element stored at the specified index to be retrieved/modified;
    - `int IndexOf(T item)` – returns the index of the first occurrence of the item value in the collection;
    - `void Insert(int index, T item)` – inserts a new element into the collection at the specified location; and
    - `void RemoveAt(int index)` – removes the element in the collection at the specified index.

- `ISet<T>` – implemented by collections which consist of unique values, i.e., collections that represent a set[2]. Important members:

    - `void ExceptWith(IEnumerable<T> other)` – removes all elements in the collection that are found in the specified collection;
    - `void IntersectWith(IEnumerable<T> other)` – removes all elements in the collection except for those found in the specified collection;
    - `bool IsProperSubsetOf(IEnumerable<T> other)` – tests if the set represented by the collection is a proper subset of the specified collection (this.Count < other.Count);
    - `bool IsProperSupersetOf(IEnumerable<T> other)` – tests if the set represented by the collection is a proper superset of the specified collection (this.Count > other.Count);
    - `bool IsSubsetOf(IEnumerable<T> other)` – tests if the set represented by the collection is a subset of the specified collection (this.Count <= other.Count);
    - `bool IsSupersetOf(IEnumerable<T> other)` – tests if the set represented by the collection is a superset of the specified collection (this.Count >= other.Count);
    - `bool Overlaps(IEnumerable<T> other)` – tests if the set represented by the collection has at least one element in common with the specified collection;
    - `bool SetEquals(IEnumerable<T> other)` – tests if the collection has the same elements as those in the specified collection;
    - `void SymmetricExceptWith(IEnumerable<T> other)` – modifies the collection so that it contains those elements present in both the current and specified collections, but not both; and
    - `void UnionWith(IEnumerable<T> other)` – adds those elements in the specified collection that are not present in the current collection.

- `IEnumerable<T>` – allows a collection to be enumerated, enabling the `foreach` loop (discussed in Section 4.5). Important member:

[2] Note how these methods appeared in the earlier list of methods that were common to both the Microsoft.Net `HashSet<>` and `SortedSet<>` collections.

- `IEnumerator<T> GetEnumerator()` – returns an enumerator object that can be used to perform a traversal through the elements stored in a collection.

The last point to note regarding the `IEnumerable<T>` interface is that it has an equivalent interface in the `System.Collections` hierarchy, `IEnumerable`, which use the object base type. This ensures that the new parameterised collections are interoperable with other classes in Microsoft.Net which interact with collections. Thus, when implementing the `IEnumerable<T>` interface, you will also need to implement the `IEnumerable` interface. This is a trivial exercise however, requiring the same code as we will explain in Section 4.5.

4.5. Implementing Support for the foreach Loop

Implementing support for the `foreach` loop in C# will make the classes we develop much simpler to work with. The actual task of implementing support for `foreach` sounds quite simple – all you have to do is implement the `IEnumerable<T>` interface, which has a single member method `GetEnumerator()` (see above). However this method returns a new object that implements the `IEnumerator<T>` interface which, as we will see, is not necessarily a simple task.

First, we begin by explaining what an ***enumerator*** is. We have been using enumerators almost since we first began writing code. In particular, the first time we wrote a `for` loop to display the elements of an array, we used an enumerator. Consider the following code:

```
for(int i = 0 ; i < array.Length ; i++)
    Console.WriteLine(array[i].ToString());
```

The above code represents a for loop which displays each of the members stored in an array on the console, one per line. The enumerator that we have used here is the integer `i`, which is used to refer to each element stored in the array in turn. Similarly, consider the following equivalent code that could be used for a linked list:

```
Node current = head;
while(current != null)
{
    Console.WriteLine(current.data.ToString());
    current = current.next;
}
```

In this example, the enumerator is the variable `current`, which is used to refer to each element stored in the linked list in turn.

There are a number of points to consider when enumerating a collection:

- How is the enumerator represented? e.g., an integer for an array or a reference to a node for a linked list;
- How is the enumerator initialised? e.g., refer to the first array element (`0`) or first linked list node (`head`);
- How is the enumerator moved to the next element in the collection? e.g., increment the integer (`i++`) or move to the next node (`current = current.next`); and
- How do we test that we haven't run out of elements? e.g., the integer is less than the array length (`i < array.Length`) or the node reference isn't null (`current != null`).

For the two data structures we have been working with (arrays and linked lists), both of which are linear structures, these operations are relatively straightforward. In future weeks we will consider more complex structures, however the same concepts apply.

Critically, consider what would happen in either of the above examples if we were adding or deleting elements in the loops enumerating the collections. There are many possible outcomes in terms of these loops – it depends on the change/s being made to the collection and how the enumerator will move to the next element. For example, what would happen if the element currently referenced by the enumerator were to be deleted? It is important to note that it is impossible to provide a general solution to this problem that deals with all possible scenarios. Thus, the enumerators that we develop only continue to work as long as the collection is not modified. This is done simply by keeping a version number (an integer) with our collection – each change to the collection increases the version number by one. If the version number has changed since the enumerator was last used, an exception is thrown (`InvalidOperationException`).

Finally, the last point to emphasise regarding implementing enumerators for our collections, is that they must be done in a manner that is fully encapsulated, i.e., regardless of the data structure we are using (array, linked list, or other), the implementation of the enumerator must be hidden from client classes. We will see how to do this below.

As discussed above, to implement an enumerator for our classes we have to create a new class which implements the `IEnumerator<T>` interface. Given that this enumerator class will only work for a specific collection, and it will require access to the internal implementation of our collection, we develop this as a nested class (see Section 2.1.2). Note however that given objects created from this nested class will be returned by the `IEnumerable<T>`'s `GetEnumerator()` method, this nested class must be `public`. We also need to provide our enumerator objects with a reference to the collection object that they are being used to traverse, which we achieve by passing a reference to the collection object to the constructor for the enumerator object. Finally, note that the name of the enumerator class should end with the word `Enumerator`, e.g., for the `Vector<TYPE>` class we developed in Task 1.1, the enumerator class would usually be named `VectorEnumerator<TYPE>`.

The members of the `IEnumerator<T>` interface are quite straightforward:

- `T Current { get; }` – read-only property that returns the value of the current element in the collection;
- `bool MoveNext()` – moves the enumerator to the next element in the collection, returning false if there are no more elements in the list; and
- `void Reset()` – returns the enumerator to its initial position.

Note that both the `MoveNext()` and `Reset()` methods must throw an `InvalidOperationException` object if the collection has changed since they were created, i.e., if the version number has changed. Critically, also note how the `Reset()` method description states that the enumerator is returned "to its initial position", not "to the first element". This is because the enumerator does not begin on the first element. The following scenario illustrates the correct behaviour of an iterator:

1. An enumerator is created for a collection, `Current` returns the default value for the data type stored in the collection, e.g., a reference type would return null;
2. The `MoveNext()` method is invoked and returns true, `Current` now returns the value of the first element stored in the collection;
3. The `MoveNext()` method is invoked and returns true, `Current` now returns the value of the second element stored in the collection;
4. …
5. The `MoveNext()` method is invoked and returns true, `Current` now returns the value of the last element stored in the collection;
6. The `MoveNext()` method is invoked and returns false, `Current` now returns the default value for the data type stored in the collection; and
7. The `Reset()` method is invoked, the enumerator is reinitialised and `Current` now returns the default value for the data type stored in the collection, as in Step 1.

Note that the default value for any data type can be determined using the `default` operator,e.g., `default(int)` returns the value `0`.

To illustrate these concepts, we present the implementation of an enumerator for the `Vector<TYPE>` class we developed in Task 1.1[3]:

```
public class Vector<TYPE> : IEnumerable<TYPE>
{
    ...
    public class VectorEnumerator : IEnumerator<TYPE>
    {
        private Vector<TYPE> _Vector;
        private int _Version;
        private int _Index;
        private TYPE _Current;

        internal VectorEnumerator(Vector<TYPE> vector)
        {
            _Vector = vector;
            _Version = vector._Version;
            Reset();
        }

        public TYPE Current
        {
            get { return _Current; }
        }

        public void Dispose()
        {
        }

        object System.Collections.IEnumerator.Current
        {
```

[3] Note that this class is nested inside the Vector<TYPE> class, the full updated code for which, including support for the version number, will be included in the solution file for this week.

```
            get { return _Current; }
        }

        public bool MoveNext()
        {
            if (_Version != _Vector._Version)
                throw new InvalidOperationException();

            if (_Index < _Vector._LastUsed)
            {
                _Current = _Vector._Data[++_Index];
                return true;
            }
            else
            {
                _Current = default(TYPE);
                return false;
            }
        }

        public void Reset()
        {
            if (_Version != _Vector._Version)
                throw new InvalidOperationException();

            _Index = -1;
            _Current = default(TYPE);
        }
    }

    public IEnumerator<TYPE> GetEnumerator()
    {
        return new VectorEnumerator(this);
    }

    System.Collections.IEnumerator Sys-
tem.Collections.IEnumerable.GetEnumerator()
    {
        return new VectorEnumerator(this);
    }
}
```

Notes:

- The instance variable `_Current` has been used to track the current value in the collection. This variable is initialised to the default value in the constructor and reset methods to represent the initial reference "before the first element".
- The instance variable `_Index` refers to the next element to set `_Current` to. Once `_Current` is updated, `_Index` is incremented to refer to the next element in the array (this variable performs the same task as the integer `i` in the earlier `for` loop example to enumerate an array).
- Both the `MoveNext()` and `Reset()` methods first confirm that the version has not changed and throw an exception if applicable.

- The constructor invokes the `Reset()` method to avoid duplicate code, however given that this class is tiny this isn't a critical issue.

Task 4.1. Implementing Platform Interfaces

Objective: As discussed above, implementing the platform's own interfaces will make your classes have the same look and feel and be easier for developers to use. In this task we will gain experience extending one of our classes further to ensure they are consistent with the platform and implement necessary functionality to support the foreach loop.

For this task you are required to extend the `OrderedLinkedList<TYPE>` class we developed in Task 2.1 to implement the `IList<T>` interface. This may initially sound like a large task, given that the `IList<T>` interface defines many members (including those inherited from other interfaces), however we have already implemented most of them.

Note: You may use your own solution to Task 2.1 if you wish, however you should only do so if you have reviewed the specimen solution in CloudDeakin and confirmed that your solution was correct/equivalent.

You will need to complete the following tasks:

1. Add the interface to the class declaration, i.e.,

```
public class OrderedLinkedList<TYPE>
    : IList<TYPE>
    where TYPE : IComparable<TYPE>
{
    ...
}
```

2. Implement each of the new members, according to the following notes:
   Hint: the quickest way to get started in Visual Studio is to hover your mouse cursor over the interface name in the class declaration, at which point a button will appear allowing you to automatically add all of these members (their implementation will just be to throw an exception however).
   - `public void CopyTo(TYPE[] array, int arrayIndex)` – copy all of the elements in the linked list to the `array`, starting at `arrayIndex`;
   - `public IEnumerator<TYPE> GetEnumerator()` – implement the enumerator method as discussed above in Section 4.5 (note you will need to do the same for the `IEnumerator` equivalent);
   - `public bool IsReadOnly { get; }` – return `false` (the linked list is not read-only);
   - `public int IndexOf(TYPE item)` – traverse the linked list until you locate an node whose `data` matches `item`, returning its `index` (return `-1` if no node is found with a matching value);
   - `public void Insert(int index, TYPE item)` – this method does not make sense (as our list is ordered), thus throw a `NotSupportedException` object; and
   - `public void RemoveAt(int index)` – remove the node at the specified `index` from the list.

3. Test your new methods thoroughly, either by modifying the `Main` method provided for Task 2.1 or by writing your own.

Note: You do not need to submit the Main method.

5

5. Recursion and Stacks

In this section we begin by reviewing the concept of recursion from the pre-requisite. Recursion provides an alternative to iteration for implementing repetition and is an important tool for solving programming problems. To explain why recursion is often slower than iteration, we examine how the memory of a process is used for recursion, in particular the system stack. Finally, we examine the stack as a data structure that can be used to solve programming problems.

Objectives

In this session you will learn

- The concept of recursion and how it is used to solve problems;
- Why recursive algorithms are often slower than their iterative equivalents; and
- The stack data structure and how it is used to solve problems.

5.1. Recursive Algorithms

Most programmers are familiar with using iterative control structures for implementing repetition in their programs using control structures for repetition, i.e., use of the `while` loop, the `do-while` loop, the `for` loop, and the `foreach` loop (where available). An alternative way of achieving repetition is ***recursion***, whereby a method contains a call to itself (a recursive call). Recursive algorithms solve problems by progressively reducing the problem until a new problem is reached with a known solution, i.e., the algorithm results in the same problem to be solved, only smaller.

There are two key components to any recursive algorithm: the ***general case*** and the ***base case***. The general case represents one or more expressions that reduce the problem to something smaller, then perform the recursive call to solve the next step. The base case defines the known solution (or solutions depending on the algorithm) and usually represent the smallest problem/s solved by the algorithm. Critical to the success of any recursive algorithm is that the general case must always progress a problem towards the base case, otherwise the recursive algorithm will never terminate and the program will eventually crash (the program's stack will eventually grow too large).

When applying recursion it is important to understand that given two equivalent algorithms, one using iteration and the other recursion, the recursive algorithm will usually perform worse than an equivalent iterative algorithm. This observation may suggest that recursive solutions are not useful, however this is not correct as there are a number of reasons why we still recursive algorithms are used regularly in software development, including:

- Some problems cannot be solved without recursion (there are no equivalent iterative algorithms);
- Some algorithms are more simply/clearly expressed using recursion, leading to reduced development time, reduced likelihood of bugs, and so on.
- Recursion enables backtracking, useful for algorithms such as path finding.

Example: Factorial

One of the simplest examples of a recursive algorithm can be seen in the mathematical problem of factorial. The factorial of a number ($n!$) is the product of all values up to and including that number, e.g.,

$$\begin{aligned}5! &= 5 \times 4 \times 3 \times 2 \times 1\\ 4! &= 4 \times 3 \times 2 \times 1\\ 3! &= 3 \times 2 \times 1\\ 2! &= 2 \times 1\\ 1! &= 1\end{aligned}$$

Y may have noticed in the above examples that the solution to each line also appears in the previous line, i.e., we could equally have written:

$$\begin{aligned}5! &= 5 \times 4!\\ 4! &= 4 \times 3!\\ 3! &= 3 \times 2!\\ 2! &= 2 \times 1!\\ 1! &= 1 \times 0!\end{aligned}$$

The general case for Factorial can then easily be derived as follows:

$$n! = n \times (n - 1)!$$

The base case (known solution) for the Factorial problem is then:

$$0! = 1$$

The algorithm for solving factorial recursively is as follows:

```
ASSUMPTION: The parameter value is unsigned (negative values are not possible)
Factorial(value)
begin
    if value = 0
```

```
            Factorial ← 1
        else
            Factorial ← value * Factorial(value – 1)
        end-if
end
```

Algorithm 5.1: Recursive algorithm for calculating $n!$

In this algorithm, the input value is first tested to see if it matches the base case (0), in which case the result is known (1), otherwise the general case is used to reduce the problem one step towards the base case. To explain how this algorithm works recursively, we can expand the algorithm progressively, as follows, to demonstrate the calculation of the factorial of 5! (parenthesis are used to illustrate the outcome of each recursive call):

- Factorial(5) = (5 * Factorial(4))
- Factorial(5) = (5 * (4 * Factorial(3)))
- Factorial(5) = (5 * (4 * (3 * Factorial(2))))
- Factorial(5) = (5 * (4 * (3 * (2 * Factorial(1)))))
- Factorial(5) = (5 * (4 * (3 * (2 * (1 * Factorial(0))))))
- Factorial(5) = (5 * (4 * (3 * (2 * (1 * 1)))))
- Factorial(5) = (5 * (4 * (3 * (2 * 1))))
- Factorial(5) = (5 * (4 * (3 * 2)))
- Factorial(5) = (5 * (4 * 6))
- Factorial(5) = (5 * 24)
- Factorial(5) = 120

If we consider an iterative solution to the factorial problem, we could use the following algorithm[1]:

```
ASSUMPTION: The parameter value is unsigned (negative values are not possible)
Factorial(value)
begin
        if value < 2
            Factorial ← 1
        else
            result ← value
            while value > 2
                value ← value – 1
                result ← result * value
            end-while
            Factorial ← result
        end-if
end
```

Algorithm 5.2: Iterative algorithm for calculating $n!$

Now that we have both algorithms, we can consider their behaviours. For the recursive version, at the deepest level of recursion the resulting equation is simply:

[1] Note that the iterative algorithm could be written in many ways, this solution is used as it results in a similar calculation to the recursive version.

Factorial(5) = 5 * 4 * 3 * 2 * 1 * 1

For the iterative version, the equation is similar:

Factorial(5) = 5 * 4 * 3 * 2

Note that the additional * 1 calculations occur in the recursive version due to the way that the calculation progresses down as Factorial(0), however in the iterative version this has been improved by eliminating the need to calculate either Factorial(0) or Factorial(1), which both evaluate to a multiplier of one. We could easily improve the recursive version of Factorial to eliminate these redundant multiplications, however the iterative version will always perform better and is still easily understood. The lesson here is that recursive algorithms are not always simpler and often result in unnecessary overhead.

Example: Fibonacci

The Fibonacci sequence is one of the most well known numerical sequences. It begins with the first two values, 0 and 1 (these are the base cases), and then each of the following values is simply the addition of the preceding two values, i.e.,

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

This numerical sequence is named after its inventor/discoverer Leonardo Fibonacci, an early 13th century mathematician who was investigating the breeding of rabbits in ideal conditions. In particular, assume:

- Begin with one newly born pair of rabbits, one male and one female;
- Rabbits reach maturity after one month and begin breeding immediately;
- The gestation period for rabbits is also one month, and will give birth to two rabbits, one male and one female, who will in turn begin breeding upon reaching maturity.

Based on these assumptions, we arrive at the numerical sequence, i.e.,

- 1 pair – Newly born pair of rabbits;
- 1 pair – First pair of rabbits reach maturity and begin to mate;
- 2 pair – First pair give birth to a new pair of rabbits;
- 3 pair – First pair give birth to a new pair, second pair reach maturity and begin to mate;
- 5 pair – First and second pair give birth to a new pair, third pair reach maturity;
- And so on.

As stated above, each number in the sequence is the sum of the previous two numbers, i.e., the general case is:

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$

The two base cases are:

$$Fib(0) = 0$$
$$Fib(1) = 1$$

Combining these together, we get the following algorithm:

```
ASSUMPTION: The parameter value is unsigned (negative values are not possible)
Fibonacci(value)
begin
    if value = 0
        Fibonacci ← 0
    else if value = 1
        Fibonacci ← 1
    else
        Fibonacci ← Fibonacci(value – 1) + Fibonacci(value – 2)
    end-if
end
```

Algorithm 5.3: Recursive algorithm for calculating Fibonacci(n)

As for Factorial, we can also consider also show how the calculation is performed (Fibonacci is shortened to F):

- F(5) = (F(4) + F(3))
- F(5) = ((F(3) + F(2)) + (F(2) + F(1)))
- F(5) = (((F(2) + F(1)) + (F(1) + F(0))) + ((F(1) + F(0)) + 1))
- F(5) = ((((F(1) + F(0)) + 1) + (1 + 0)) + ((1 + 0) + 1))
- F(5) = ((((1 + 0) + 1) + 1) + (1 + 1))
- F(5) = (((1 + 1) + 1) + 2)
- F(5) = ((2 + 1) + 2)
- F(5) = (3 + 2)
- F(5) = 5

Importantly, the above steps actually are showing calculations being performed in parallel, e.g., in the second step, F(4) is expanded to (F(3) + F(2)) at the same time that F(3) is being expanded to (F(2) + F(1)). In reality however, these calculations would be done in sequence. By illustrating the computation as above, each separate call to Fibonacci is shown only once, which highlights that many Fibonacci numbers are calculated several times, in particular:

- F(5) is calculated one time
- F(4) is calculated one time
- F(3) is calculated two times
- F(2) is calculated three times
- F(1) is calculated five times
- F(0) is calculated three times

This results in a total of 15 separate calls to the Fibonacci method. This occurs because the recursive version of Fibonacci discards intermediate results, e.g., in the first expansion F(5) we must first calculate F(4), which in turn requires the calculation of F(3), which is discarded even though we also require that result to complete the calculation of F(5).

Now consider the following iterative algorithm for solving Fibonacci:

```
ASSUMPTION: The parameter value is unsigned (negative values are not possible)
Fibonacci(value)
begin
```

```
        if value = 0
                Fibonacci ← 0
        else
                a ← 0
                b ← 1
                result ← 1
                while value > 2
                        a ← b
                        b ← result
                        result ← a + b
                        value ← value – 1
                end-while
                Fibonacci ← result
        end-if
end
```

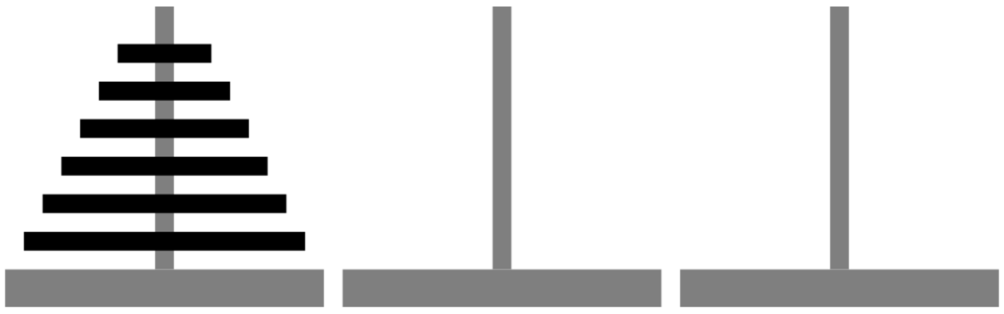
Algorithm 5.4: Iterative algorithm for calculating Fibonacci(n)

This algorithm is quite different to the recursive version. Where the recursive version starts with the desirable value and works its way back to the known solution, this iterative version begins with the known solutions and then works out the target number the same way a person would, by adding each consecutive pair until the target is reached.

If you write a program to test the performance of these algorithms, the improvement in speed of the iterative version over the recursive version becomes noticeable very quickly[2]. This is an important lesson, that even though the recursive algorithm may appear to be simpler, it may take substantially longer.

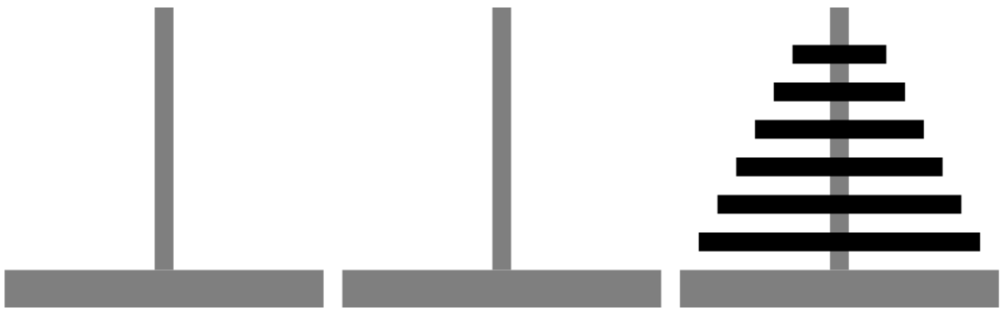
Example: Towers of Hanoi

So far we have considered two problems that are mathematical in nature. The applications of recursion however are not restricted to mathematics. Consider the simple game commonly known as the Towers of Hanoi. In this game, the player is presented with three pins, one of which contains a stack of movable discs, each disc being slightly smaller than the one below it, as follows:



[2] Calculating the 50th Fibonacci number on a 3.6GHz Core i7 CPU is almost instant using the iterative algorithm but takes minutes using the recursive algorithm. A simple program representing these algorithms can be found in this week's provided code.

The objective of the game is to move all of the discs from one of the pins to another, usually the right hand pin, i.e.,

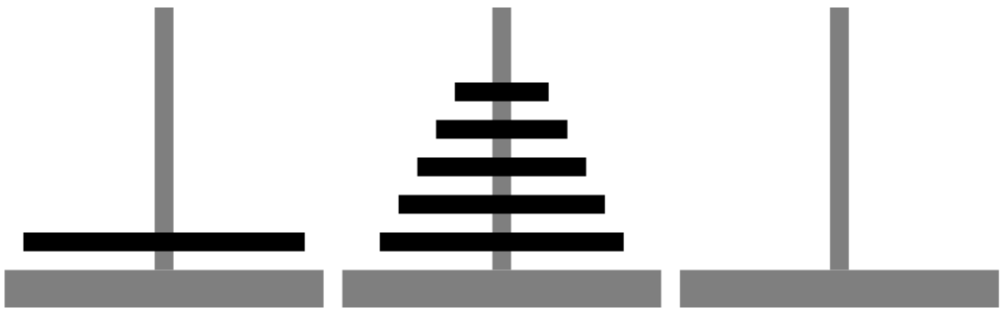


Although this may seem simple enough, two rules make this rather problematic:

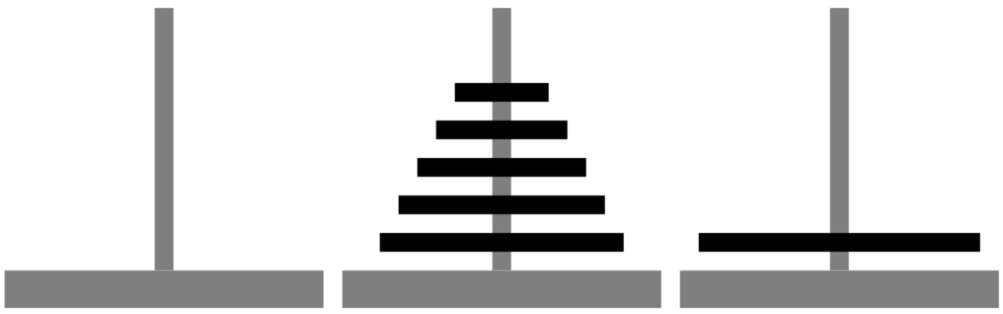
- Only one disc may be moved at any one time; and
- No disc may be placed on top of a smaller disc.

Critically however, the solution to the problem presented by the Towers of Hanoi is simple when you apply recursion. In the above problem, which consists of six discs to be moved, if you can determine how to move five discs to the middle pin, the last disc can then be moved to the destination pin and then you only need to repeat the process of moving the five discs to the destination pin to complete the solution, i.e.,

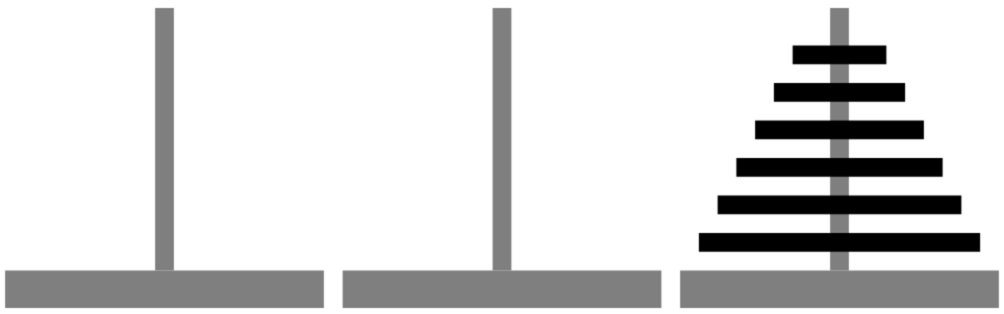
Step 1 – Move five discs to the middle pin:



Step 2 – Move the last disc to the destination pin:



Step 3 – Move the five discs from the middle pin to the destination pin:



More generally, to move *n* discs from a given *source* pin to a given *destination* pin with a third *alternative* pin:

- Move *n* – 1 discs from the *source* to the *alternative*;
- Move 1 disc from the *source* to the *destination*; and
- Move *n* – 1 discs from the *alternative* to the *destination*

This results in the following algorithm:

```
ASSUMPTION: The parameter number is unsigned and >= 1, representing the number of discs to move
ASSUMPTION: The parameters src, dest, and alt contain text representing each pin for a single move
Towers(number, src, dest, alt)
begin
    if number = 1
        print "Move 1 disc from " + src + " to " + dest
    else
        Towers(number – 1, src, alt, dest)
        print "Move 1 disc from " + src + " to " + dest
        Towers(number – 1, alt, dest, src)
    end-if
end
```

Algorithm 5.5: Recursive algorithm for solving Towers of Hanoi

If you follow the algorithm closely, you will notice:

- If there is only one disc to move, perform the move;
- If there is more than one disc to move
  - Move *n* – 1 discs from the *source* to the *alternative* pin, using the *destination* pin as the *alternative* for this move;
  - Move 1 disc to the *destination* pin; and
  - Move *n* – 1 discs from the *alternative* to the *destination* pin, using the *source* pin as the *alternative* for this move.

Implementing a program representative of the above algorithm yields the following results[3]:

```
1: Move 1 disc from L to M
2: Move 1 disc from L to R
```

[3] This program represents the first part of Task 5.1. An additional counter has been added to the output to indicate the move number.

```
     3: Move 1 disc from M to R
     4: Move 1 disc from L to M
     5: Move 1 disc from R to L
     6: Move 1 disc from R to M
     7: Move 1 disc from L to M
     8: Move 1 disc from L to R
     9: Move 1 disc from M to R
    10: Move 1 disc from M to L
    11: Move 1 disc from R to L
    12: Move 1 disc from M to R
    13: Move 1 disc from L to M
    14: Move 1 disc from L to R
    15: Move 1 disc from M to R
```

To solve this problem in the same manner using iteration would not be a simple task. However, a simple iterative solution using an alternative approach does exist. Consider the following solution for an odd number of discs:

- Move the smallest disc one pin to the left (if already on the left-most pin, move it to the right-most pin);
- Leaving the smallest disc in place, only one other move is possible – make this move; and
- Repeat until solved.

The solution for an even number of discs is identical, except the smallest disc is moved one pin to the right instead. Although this iterative solution does work, additional functionality would be required to allow the location of each disc to be known, e.g., modelling in memory which discs are currently on which pegs.

5.2. Equivalent Algorithms - Why Recursion is Slower

We have already identified that recursive solutions are usually slower than an equivalent iterative solution. Critical in this statement however is the question of equivalence. In considering the above solution to factorial, the algorithms used were practically equivalent, starting with the input value and progressively multiplying each smaller value.

However, in considering the algorithms for determining the Fibonacci sequence, or more correctly a specific number in the Fibonacci sequence, two different algorithms were used – the recursive algorithm started with the desired number, determining each of the two previous numbers recursively, whereas the iterative solution simply built the sequence from the start until the desired number was reached. In other words, although the iterative solution to Fibonacci was (much) faster than the recursive solution, the solutions were fundamentally different.

The applications of recursion identified above are still valid however and in future weeks we will demonstrate some algorithms where recursion provides better solutions. However the issue of why recursion would perform worse than an equivalent iterative algorithm remains unexplained. To understand why, it is necessary to first understand what the system stack is and how it is used in the execution of a program.

A program is represented by an executable file (produced by the compiler) that is typically stored in non-volatile memory, e.g., on a hard disk. When the user selects a program to run, the operating system loads some or all of the content of the executable file into the memory of the computer and associates a number of other resources with the memory contents to form a process[4]. The contents of the executable file is not simply loaded into one location in memory however. Instead, the memory of a process is usually divided into at least three regions:

- The code/text region – an area of memory that contains the machine code of the program (see Section 3.1 for an explanation of machine code);
- The data/heap region – an area of memory that contains any initialised data, e.g., string literals (values) that appear in the code, and may grow throughout the lifetime of the program as it is also used for dynamically allocated memory; and
- The stack region/system stack – an area of memory that is used to temporarily store data.

The stack is used for many purposes, which may include[5]:

- Storage for local variables;
- A temporary storage area for interim results in large calculations;
- A temporary storage area used for method calls including:
    - A copy of register contents prior to a method invocation which are restored after the method terminates;
    - A copy of any parameters passed to a method;
    - A reserved storage location to store any return value; and/or
    - A record of the location where the method was invoked from so the hardware knows which previous instruction to return to after the method completes.

The last points here, regarding the use of the stack in method calls, provides a hint to the reason why recursion is slower. Consider the two images shown in Figure 5.1 representing how the stack is used when calculating 5! using the recursive and iterative Factorial algorithms. In particular, you will notice that for the recursive version, more stack space has been used due to the recursive calls to the same method, whereas for the iterative version only one call is used. The same pattern will occur regardless of the number of times the recursive method must be invoked to solve a problem, i.e., for a larger problem the stack usage by the iterative version will remain the same however the recursive version will continue to use more space.

[4] Simply put, a process is a program in execution, i.e., it is the run-time representation of a program.

[5] Critically, how the stack is used depends upon many factors including the hardware platform itself and the compiler used, more specifically how the compiler translates high level language constructs into machine code.

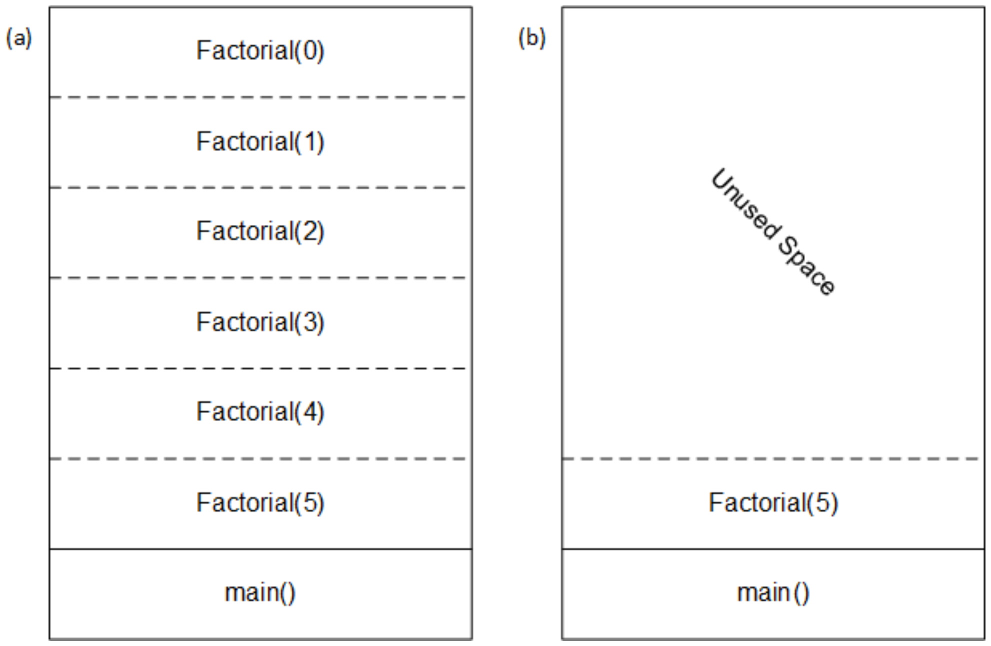


Figure 5.1: System Stack Usage for (a) Recursive and (b) Iterative Factorial

The increased memory usage does not in itself account for all of the overhead encountered when using recursion. However when combined with the copying of data/to from this additional memory, does account for a large part of the overhead. The remaining overhead is due to how the stack is managed by the operating system. In particular, the stack will usually start off very small (one or two pages which are often 4KB each), and the operating system will grow that memory region as needed, i.e., as the stack usage grows the operating system will intervene to increase the size of the stack as required.

The need to grow the stack is presented to the operating system as an exception caused by the process – the process doesn't know it is reaching the stack limit and accesses an area of memory it doesn't own (yet). The operating system then checks to determine the cause of the exception (stack needs to grow), allocates the memory to the stack (which may require some paging/swapping), and restarts the process at the operation that initially triggered the exception. This whole process adds further overhead to the execution of the recursive method. The growth of the stack in this manner also implies that as a programmer you must design recursive algorithms to minimise the amount of space taken by each recursive call.

It is also important to recognise that the growth of the system stack is actually limited by the amount of free memory that exists between the stack and the other regions of memory (text and heap). If the stack were to reach another region of memory, the operating system will terminate the process with an out of memory error (even though the amount of memory used may not be significant). In these circumstances it may be necessary to change the algorithms and/or design of an application to eliminate the recursion.

5.3. The Stack Data Structure

Although we have examined the concept of recursion and how recursive methods make use of the system stack, a stack can also be implemented as a data structure that implements Last-In-

First-Out (LIFO) semantics and has some useful applications. To illustrate the operation of a stack, consider the examples shown in Figure 5.2. For each of these examples, consider adding and removing a single item from the stack. It is possible to add or remove items at the top of the stack, but not from the middle or bottom of the stack. This is precisely the behaviour of the stack data structure.

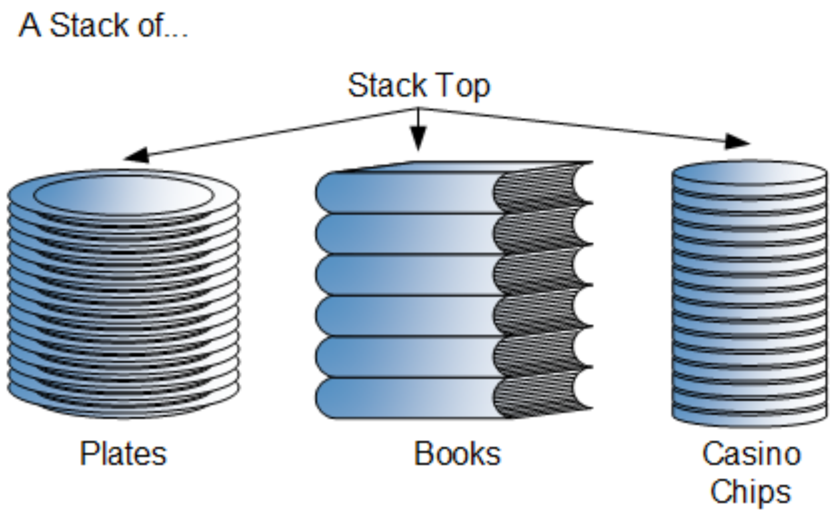


Figure 5.2: Real Life Stack Examples

The following operations are common to all stacks:

- Push – add an item to a stack (push an item on the stack);
- Pop – remove an item from a stack (pop an item off the stack); and
- Top – examine the item on the top of the stack without removing it.

Additional operations that are often provided for working with a stack include:

- Empty – test whether the stack is currently empty (no items on the stack);
- Full – test whether the stack has space for more items to be added; and
- Count – determine how many items are currently stored on the stack.

There are also two important concepts to be aware of when working with a stack:

- Underflow – an error that occurs when attempting to examine the top element or pop an element from the stack when the stack is empty; and
- Overflow – an error that occurs when attempting to push an element onto the stack when the stack is full (implementation dependent).

Stacks can be implemented either by using an array or a linked list. For the array implementation, an array of any size greater than or equal to the number of elements stored in the stack must be used, and a record of the current location of the stack top must also be kept (same as the last used reference discussed in Section 1.6.1), as shown in Figure 5.3. Implementing the push operation is then a simple case of checking that enough space exists in the array (or resizing the array if necessary), incrementing the stack top index/reference, and storing the data in the array. Similarly, implementing the pop operation requires checking that the stack isn't empty, decrementing the stack top index/reference, and optionally returning the value that was stored[6].

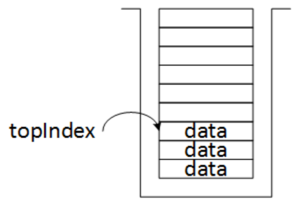


Figure 5.3: Array Implementation of a Stack

For the linked list implementation, all changes occur to one end of the linked list. As a result, a singly linked list structure is usually used for simplicity, and only one reference is kept to the list (the head reference, usually referred to as top/stack top due instead). This structure is shown in Figure 5.4. Adding and removing elements to the linked list is no different to inserting and removing elements from a linked list, as addressed in Section 2.2, except those operations are only performed to one end of the linked list.

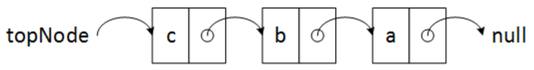


Figure 5.4: Singly Linked List Implementation of a Stack

Example: String Reversal

An important side effect of the LIFO semantics of the stack data structure is for reversing the order of data that is stored in the stack. The simplest application of this property is to reverse a string. The algorithm for this is very simple:

```
ASSUMPTION: The parameter text is an enumerable string
Reverse(text)
begin
        foreach char in text
                push char
        end-foreach
        while NOT stack.empty()
                pop char
                print char
        end-while
end
```

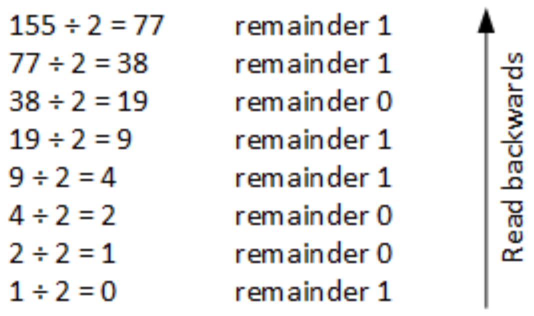
Algorithm 5.6: Reverse a string using a stack

In this example, the last character pushed on to the stack (the last character in the string) is the first character that is popped off. This process continues until all characters have been popped off the stack and the string has been displayed in reverse order.

[6] If using C#, you should also set the array element to the default value for that data type to allow any reference types to be deleted by the garbage collector if they are no longer referenced elsewhere.

Example: Decimal to Binary Conversion

It is possible to convert a decimal number (base 10) to its binary equivalent (base 2) by continuously dividing the decimal number by two using integer division and recording any remainder from the division. Once a value of zero is reached, the remainder values (each either 0 or 1) are read backwards to determine the binary number. For example:



$155 \div 2 = 77$	remainder 1
$77 \div 2 = 38$	remainder 1
$38 \div 2 = 19$	remainder 0
$19 \div 2 = 9$	remainder 1
$9 \div 2 = 4$	remainder 1
$4 \div 2 = 2$	remainder 0
$2 \div 2 = 1$	remainder 0
$1 \div 2 = 0$	remainder 1

Read backwards

The algorithm required to perform this is straightforward, consisting only of the fundamental arithmetic operators modulus (to calculate the remainder) and division (to actually divide the number)[7]. Each remainder value is then pushed onto a stack, and then displaying the reverse is simply a matter of repeating the pop/print loop done for string reversal, above.

5.4. Elimination of Recursion

Recall from Section 5.2 how recursive algorithms make use of the system stack when calling methods as part of their operation. It is possible to eliminate recursion in an algorithm by replacing the use of the system stack (through method calls) with a stack data structure (using push and pop). There are two key points to remember when undertaking this task:

- The relevant data that is pushed on the system stack must be reflected in the data placed on the stack data structure, including the initial problem; and
- You must allow for the reversal property of the stack when using such a data structure directly.

A simple loop is then used whereby a problem to be solved is popped off the stack for which either a solution is found/known, or one or more sub-problems are pushed onto the stack in its place (in reverse order).

For example, consider Algorithm 5.5 presented earlier for solving the Towers of Hanoi recursively. There are three locations where information is placed on the system stack:

- In the original call to this algorithm, representing the original problem;
- In the recursive call: ***Towers(number – 1, src, alt, dest)***; and
- In the recursive call: ***Towers(number – 1, alt, dest, src)***.

[7] Conversion from decimal to binary is also possible using bitwise operators as an alternative to modulus and division, with potentially improved performance, however for simplicity we do not consider them here.

Using this information, we can adjust this algorithm to eliminate recursion using a stack data structure, as follows:

```
ASSUMPTION: The parameter number is unsigned and >= 1, representing the number of discs to move
ASSUMPTION: The parameters src, dest, and alt contain text representing each pin for a single move
Towers(number, src, dest, alt)
begin
    push { number, src, dest, alt }
    while NOT stack.empty
        pop { number, src, dest, alt }
        if number = 1
            print "Move 1 disc from " + src + " to " + dest
        else
            push { number – 1, alt, dest, src }
            push { 1, src, dest, alt }
            push { number – 1, src, alt, dest }
        end-if
    end-while
end
```

Algorithm 5.7: Solution to Towers of Hanoi using a stack data structure

Notice the following points in the above algorithm:

- The initial problem is immediately pushed onto the stack at the start of the method;
- The two recursive calls appear as push statements in reverse order; and
- The middle step also appears as a push statement, in between the previously recursive calls.

Task 5.1. Recursion and Stacks

Objective: Recursion represents an important skill for solving programming problems that all programming professionals must be comfortable with. Tied closely to understanding recursion is the need to understand stacks and how they are used. This task explores both recursion and stacks to help you develop this skill.

There are three parts to this task that should result in a single C# program class. The requirements are as follows:

1. Implement two methods that implement the Towers of Hanoi solutions presented in Algorithm 5.5 and Algorithm 5.7;
2. Using a stack, implement a method that examines an expression input by the user that contains different brackets: ( ), { }, and [ ]. Your task is to validate the expression using the stack, e.g., the expression "({ [ ] })" is valid, however the expression "(({ [ ] })" is not (an open bracket without a matching close bracket); and
   Hint: Step through each character of the string, pushing any open brackets on to the stack. When a close bracket is discovered, pop the matching bracket off the stack and see if it is a matching bracket. If the brackets don't match, the expression is invalid. If the stack is empty when attempting to pop a bracket, the expression is invalid. If the stack is not empty when the end of the expression is reached, the expression is invalid.

Otherwise, the expression is valid (all brackets matched and the stack is empty at the end of the expression).

3. Implement a Main method that allows the above methods to be tested.

Note: For the stack required for the above problems, you may either implement your own stack (array-based or linked list-based), or you may use the built-in stack described in Section 4.2 and documented here:

http://msdn.microsoft.com/en-us/library/3278tedw.aspx

You will not be required to submit your stack solution however.

6

6. Queues, Priority Queues, and Deques

In this session we examine the concepts of the queue, priority queue, and deque. The queue is a data structure used to solve many problems in programming, some of which we will examine in later sessions. The priority queue is a variation on the queue data structure, where items are ordered by some assigned priority, e.g., consider your "to-do list" of study and assessment tasks for the trimester and how the ordering of this list can change as due dates approach. Finally we examine the concepts of the double-ended queue (deque), a more generalised version of the data structure, which we can use to implement both stack and queue data structures.

Objectives

In this session you will learn

- The concepts of the queue data structure;
- The concepts of the priority queue;
- The concepts of the deque data structure; and
- How to implement stacks and queues by exploiting the functionality of a deque.

6.1. The Queue Data Structure

Queues are a common data structure that are used where a First-In, First-Out (FIFO) ordering of data is required[1]. Queues are found regularly in real life situations, e.g.,

- A queue of people for service at supermarket/bank/cafe;
- A queue of people on a hospital waiting list;
- A queue of patients to attend;
- A queue to board an aeroplane/train/bus/escalator;
- A queue of cars at traffic lights or other street corner;
- A queue of tasks in a to-do list/assignment; and
- A queue of performers in a circus.

[1] The semantics of a queue can equally be referred to as ***Last-In Last-Out*** (LILO).

Some of these queues have a strict behaviour (a fixed ordering), e.g., people using an escalator that is only wide enough for one person, or the ordering may change, e.g., a queue of patients being treated at a hospital (which are triaged to treat priority patients first). In this section we will examine in detail queues that have a strict behaviour. We will examine priority queues in Section 6.2. Similarly, queues are found regularly throughout computing:

- A queue of documents to be printed;
- A queue of processes waiting to execute;
- A queue of read/write tasks to be performed on disk;
- A queue of incoming/outgoing network transmissions;
- A queue of transactions to be processed;
- A queue of terminal/user input ("input queue");
- A queue of terminal output ("output queue");
- A queue of players to login to an online game; and
- A queue of music files for playing.

We will also examine some of the applications of the queue in other data structures in future weeks.

The following operations are common to all queues:

- Enqueue – add an item to the rear of the queue;
- Dequeue – remove an item from the front of queue; and
- Front – examine the item at the front of the queue without removing it.

Other operations that are regularly provided for working with a queue include:

- Rear – examine the item at the rear of the queue without removing it;
- Empty – test whether the queue is currently empty (no items in the queue);
- Full – test whether the queue has space for more items to be added; and
- Count – determine how many items are currently stored in the queue.

The concepts of underflow and overflow, as described for stacks (Section 5.3) also apply:

- Underflow – an error that occurs when attempting to examine the front/rear element or dequeue an element from the queue when the queue is empty; and
- Overflow – an error that occurs when attempting to enqueue an element in the queue when the queue is full (implementation dependent).

Queues can be implemented either by using an array or a linked list. The simplest of these is the linked list implementation, for which a singly linked list is usually adequate. New elements are enqueued/added at the tail of the linked list (referred to as the rear), and elements are dequeued/removed from the head of the linked list (referred to as the front), as shown in Figure 6.1. These are implemented the same as the normal linked list operations, but are simplified due to the known locations.

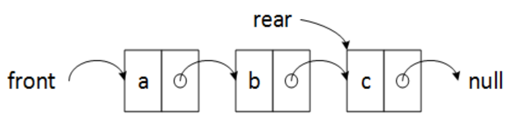


Figure 6.1: Singly Linked List Implementation of a Queue

The array implementation for a queue is more involved. Simplistically, we can implement a queue using an array by maintaining two indexes to the front and rear elements, as shown in Figure 6.2. To enqueue an element, the new element is stored in the next free slot and the rear index is incremented. To dequeue an element, the front element is removed from the array and the front index is incremented, potentially returning the removed element. The problem presented by this approach however is also apparent in Figure 6.2 – there is currently no room left in the array for new elements to be enqueued, even though there are two unused elements in the array.

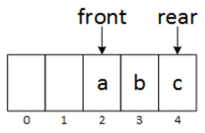


Figure 6.2: Array Implementation of a Queue (Linear)

The most obvious solution to this problem might appear to be to shuffle the elements of the array as needed, i.e., when there is no longer any space left at the end of the array, move the elements stored in the queue to the start of the array, freeing up space for further enqueue operations. However as we have discussed previously, shuffling elements in an array is a costly operation, and a better solution can be found by using what is known as a circular queue structure, illustrated in Figure 6.3. In this example, the same array is used, with the same contents, but notice how the array has been illustrated as a circle.

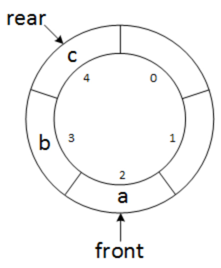


Figure 6.3: Array Implementation of a Queue (Circular)

The circular queue is implemented by wrapping the rear and front pointers using the modulus operator, e.g.,to move the rear index to the next element, we use:

rear ← (rear + 1) MOD array.size

Similarly, for moving the front index:

front ← (front + 1) MOD array.size

The modulus operator (***MOD*** above or `%` in C#) is used to wrap the index back to zero when it reaches the size of the array. In Figure 6.3, the rear index is currently referencing location 4 in an array size 5. Adding a value of 1 to the rear index results in 5, and the remainder after dividing by the array size (5) is zero (5 ÷ 5 = 1 remainder 0). However if we apply the same to the front operator, the new value would be 3, and the remainder after dividing by the array size would also be 3 (3 ÷ 5 = 0 remainder 3).

Although we have explained how to move the indexes further forward, we face a problem, as illustrated by Figure 6.4. In this scenario, we have continued our earlier example by enqueuing two additional elements to fill the array, then dequeued all elements. If you move the rear index for each enqueue operation and move the front index for each dequeue operation, you will see that both front and rear are at the same place. The problem that is highlighted by this example is that it is not possible to tell the difference between an empty queue and a full queue.

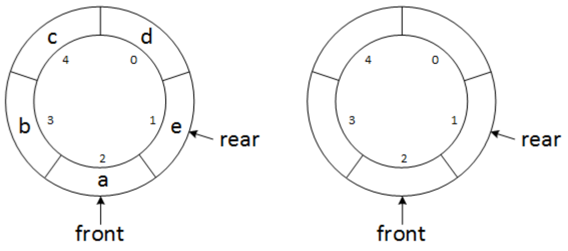


Figure 6.4: Full and Empty States of a Circular Queue

There are two approaches that can be used to solve this problem:

1. Introduce a new variable to record the number of elements stored in the queue, e.g., `_Count`; or
2. Create an array that is one element larger than the desired queue size, e.g., `_Array = new TYPE[queueSize + 1]`.

The first approach does not attempt to eliminate the problem of the front and rear indexes being the same for both empty and full queue states, instead the use of a count variable makes it straightforward to determine if the queue is empty (count equal to zero) or if the queue is full (count equal to array size).

The second approach eliminates the problem of the front and rear indexes being the same for both empty and full queue states as illustrated in Figure 6.5. Note however that now the rear index no longer points to the last element, but to the next free element in the array. With this solution, it is now possible to determine if the queue is full if the rear index is one place before the front index, and to determine if the queue is empty if both front and rear indexes are the same.

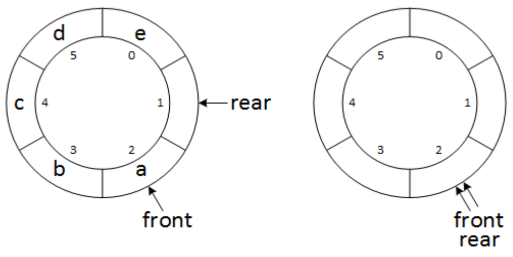


Figure 6.5: Solution to Full and Empty States of a Circular Queue

The concepts of the circular queue are illustrated in the following code which implements the data structure using the second approach.

```
public class CircularQueue<TYPE>
{
    private TYPE[] _Arr;
    private int _QFront;
    private int _QRear;

    public CircularQueue(int size)
    {
        _Arr = new TYPE[size + 1];
        _QFront = 0;
        _QRear = 0;
    }

    public bool Empty()
    {
        return _QFront == _QRear;
    }

    public bool Full()
    {
        return (_QRear + 1) % _Arr.Length == _QFront;
    }

    public void Enqueue(TYPE data)
    {
        if (Full())
            throw new InvalidOperationException("Overflow");

        _Arr[_QRear] = data;
        _QRear = (_QRear + 1) % _Arr.Length;
    }

    public TYPE Dequeue()
    {
        if (Empty())
            throw new InvalidOperationException("Underflow");

        TYPE result = _Arr[_QFront];
        _Arr[_QFront] = default(TYPE);
        _QFront = (_QFront + 1) % _Arr.Length;
```

```
            return result;
        }

        public TYPE Front()
        {
            if (Empty())
                throw new InvalidOperationException("Underflow");

            return _Arr[_QFront];
        }

        public TYPE Rear()
        {
            if (Empty())
                throw new InvalidOperationException("Underflow");

            if (_QRear == 0)
                return _Arr[_Arr.Length - 1];
            else
                return _Arr[_QRear - 1];
        }

        public int Count
        {
            get
            {
                int result = 0;

                if (!Empty())
                {
                    if (_QFront < _QRear)
                        result = _QRear - _QFront;
                    else
                        result = (_QRear) + (_Arr.Length - _QFront);
                }

                return result;
            }
        }
    }
```

Of particular note in the above code example are the following points:

- Note how both the enqueue and dequeue methods move the `index` as described above;
- Note how to determine the location of the rear element, the `_QRear` index must be checked (`_QRear` refers to the next free slot in the array, so the rear element appears in the slot before, which may need to wrap around to the end of end of the array); and
- Note how the `Count` method must check to see if the queue is currently stored within the array (`_QFront < _QRear`) or is currently stored partly at the start and partly at the end of the array (`_QFront > _QRear`).

6.2. The Priority Queue

Priority queues are a variation of the queue data structure, whereby each element stored in the queue has a priority associated with it. The most common implementation for a priority queue is to use a heap data structure, a type of tree which we examine in Section 9.2.3. Otherwise, it is necessary to modify the implementation of the enqueue and/or dequeue operations of the queue to support priorities.

The actual modifications needed to these methods is application dependent, i.e., how these methods are modified depends upon how the priority queue is to be used. However any such modifications must also consider a problem known as starvation – low priority entries may never be dequeued if enough high priority entries are being added. This may not be an acceptable solution for some applications, e.g., this is not acceptable for queues of processes in an operating system, where higher priority processes should simply receive more access to the CPU/more time on the CPU than lower priority processes, but not to the point of excluding lower priority processes.

There are a two main solutions to the problem of starvation: scheduling and aging. Scheduling addresses the decision of which priority to dequeue from, i.e., the highest priority will not always be dequeued, for example:

- Dequeue priority 1, 1, 1, 2, 2, 3, 1, 1, 1, 2, 2, 3, 1, 1, 1, 2, 2, 3, 1, 1, 1, 2, 2, 3, ... (dequeue three priority 1, two priority 2, and one priority 3);
- Dequeue priority 1, 1, 2, 1, 1, 2, 3, 1, 1, 2, 1, 1, 2, 3, 1, 1, 2, 1, 1, 2, 3, ... (every second dequeue priority n, dequeue a priority n – 1);

Aging represents an alternative approach where the highest priorities will always be selected for dequeuing, however after some time has passed, or after the occurrence of some event, the priority of elements already in the queue is increased, i.e., the priority of an item increases over time. Thus, an element that is enqueued with priority 3 will over time become priority 2, then eventually priority 1.

6.3. The Deque Data Structure

In this section we examine the double-ended queue (***deque***, pronounced "deck" to avoid confusion with the dequeue operation of a queue). The deque data structure is very similar to a queue, except that elements may be enqueued and dequeued at either end of the queue, and are usually implemented using a doubly linked list. The concept of the deque is illustrated in Figure 6.6.

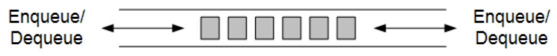


Figure 6.6: The Deque Data Structure

The following operations are common to all deques:

- Insert an element before the front of the deque (insert at doubly linked list head);
- Append an element after the rear of the deque (append at list tail);

- Remove the element at the front of the deque (remove list head); and
- Remove the element at the rear of the deque (remove list tail).

As suggested above, the actual implementation of a deque is straightforward given that the operations map directly to the operations of a doubly linked list (see Session 2). However one of the useful characteristic of the deque data structure is that it can be used to implement both a stack and queue, by mapping the calls for the stack/queue to one or both ends of the deque, i.e.,

```
Stack.Push(element)
begin
        deque.AddFront(element)
end

Stack.Pop()
begin
        Stack.Pop ← deque.RemoveFront()
end

Queue.Enqueue(element)
begin
        deque.AddRear(element)
end

Queue.Dequeue()
begin
        Queue.Dequeue ← dequeue.RemoveFront()
end
```

We will explore this further in this week's task (below).

Task 6.1. Deques, Queues, and Stacks

Objective: The deque data structure effectively implements both the queue and stack data structures at the same time. Moreover, to understand how to implement a deque implies that you also understand how to implement both the queue and stack. In completing this task we will explore this by first implementing a deque, then encapsulating its functionality to generate the stack and queue structures trivially.

Note: although this task may appear particularly large, most of the code for the Deque class can be adapted from the OrderedLinkedList class we have already written. The Stack and Queue classes are then almost entirely just calling the methods you have defined in the Deque. Thus, the requirements of this task focus more on understanding the problem conceptually than coding.

For this task you are required to implement three classes: a reusable `Deque` collection, and then encapsulate that class to create reusable `Stack` and `Queue` collections. The `Deque` collection class is to be implemented using a doubly linked list (not an ordered linked list!) and has the following features:

- Namespace:
  - `namespace SIT221_Library`
    Your class should be defined in the namespace `SIT221_Library`;
- Interfaces:
  - `ICollection<TYPE>`
    Your class must implement the `ICollection<TYPE>` interface, including the associated enumerator support.
- Properties:
  - `public int Count`
    A read-only property which returns how many elements are currently stored in the doubly linked list;
  - `public bool IsReadOnly { get; }`
    Returns `false` (the deque is not read-only);
- Constructors:
  - `public Deque()`
    A parameter-less constructor which initialises the deque's doubly linked list structure;
- Methods:
  - `public void AddFront(TYPE data)`
    A method which accepts a single element and inserts it at the head of the doubly linked list;
  - `public void AddRear(TYPE data)`
    A method which accepts a single element and appends it after the tail of the doubly linked list;
  - `public void Add(TYPE data)`
    A method which accepts a single element simply passes the information through to the `AddRear()` method, defined above;
  - `public void Clear()`
    A method which removes all elements stored in the deque;
  - `public bool Contains(TYPE item)`
    A method which searches the deque contents to determine whether a value is stored in the deque (returns `true`) or not (returns `false`);
  - `public void CopyTo(TYPE[] array, int arrayIndex)`
    A method which copies all of the elements in the deque to the `array`, starting at `arrayIndex`;
  - `public TYPE PeekFront()`
    A method which returns the value stored in the head of the linked list;
  - `public TYPE PeekRear()`
    A method which returns the value stored in the tail of the linked list;
  - `public bool Remove(TYPE item)`
    A method which deletes the first occurrence of the specified item from the deque and returns `true`, or returns `false` if the specified item did not appear in the list;
  - `public TYPE RemoveFront()`
    A method which removes the head element from the linked list and returns its value;
  - `public TYPE RemoveRear()`
    A method which removes the tail element from the linked list and returns its value;
- Enumerator support:
  - Given the use of the same doubly-linked list structure as we used in the ordered linked list example, you should be able to reuse your solution's / the specimen solution's implementation of the enumerator with minor changes.

The Queue collection is implemented by using a Deque in its implementation by exploiting delegation[2]. The Queue class will also implement the `ICollection<TYPE>` interface, and all the members specified by this interface are simply mapped/passed to the equivalent members of the Deque class, e.g., for the `Add` method you would simply write:

```
public void Add(TYPE item)
{
    _MyDequeObject.Add(item);
}
```

The following changes and/or additional methods are also required:

- It is not possible to remove an element from the middle of a queue (only from the front), so the `Remove()` method defined by the `ICollection<TYPE>` interface must throw a `NotSupportedException` instead.
- `public void Enqueue(TYPE item)`
  Add a method that provides the normal enqueue operation functionality, which is mapped/passed to the deque's `AddRear()` method;
- `public TYPE Dequeue()`
  Add a method that provides the normal dequeue operation functionality, which is mapped/passed to the deque's `RemoveFront()` method;
- `public TYPE Front()`
  Add a method that provides the normal front operation functionality, which is mapped/-passed to the deque's `PeekFront()` method;
- `public TYPE Rear()`
  Add a method that provides the normal rear operation functionality, which is mapped/-passed to the deque's `PeekRear()` method.

Similarly, the Stack collection is also implemented by using a `Deque` in its implementation by exploiting delegation and similarly implements the `ICollection<TYPE>` interface. The following changes and/or additional methods are also required:

- It is not possible to remove an element from the middle of a stack (only from the top), so the `Remove()` method defined by the `ICollection<TYPE>` interface must throw a `NotSupportedException` instead.
- `public void Push(TYPE item)`
  Add a method that provides the normal push operation functionality, which is mapped/-passed to the deque's `AddFront()` method;
- `public TYPE Pop()`
  Add a method that provides the normal pop operation functionality, which is mapped/-passed to the deque's `RemoveFront()` method;
- `public TYPE Top()`
  Add a method that provides the normal top operation functionality, which is mapped/-passed to the deque's `PeekFront()` method;

[2] Note that delegation does not mean to use delegates in C#, which is a separate concept. Delegation refers to the concept of implementing the functionality for some/all of one class by using the functionality provided by another class, which is usually hidden/encapsulated. For this task, you would create a private `Deque` member inside the `Queue` class to implement most of the queue functionality.

Finally, test your three classes by writing a simple Main method (you don’t need to submit this).

7

7. Algorithm Complexity and Sorting

Sorting represents one of the most common operations that we may wish to perform on data. Importantly, sorting also happens to be one of the most expensive operations that we can apply to data, and as we will learn in future weeks, can often be avoided. In the first part of this session, we begin by considering how to analyse the complexity of an algorithm and express that complexity in mathematical terms. These mathematical expressions provide an indication of how an algorithm will behave depending upon the amount of data input to the algorithm. As software developers, we can use this information to identify areas of code in our programs that are responsible for poor performance and replace that code with better performing algorithms. In the second part of the session we present a number of sorting algorithms and consider the complexity of those algorithms, highlighting why sorting should, in general, be avoided.

Objectives

In this session you will learn

- How to determine the efficiency of an algorithm, and thus estimate its behaviour/performance;
- How to express the efficiency of an algorithm using determine the big-O notation; and
- The bubble, selection, insertion, and quick sort algorithms.

7.1. Algorithm Complexity

When faced with a software development task, there are almost always different algorithms that can be applied to solve any individual problem. The performance and/or efficiency of these algorithms can vary substantially. For example, consider the algorithms we have considered for storing data in a reusable collection:

- For storage using a dynamically resized array (Session 1), adding an element may be a simple assignment (a simple operation, if there is space in the array), or may first require the reallocation of the array and copying of all existing stored elements (a more complex

operation, which increases in computation requirements as the array size increases); and
- For storage using a linked list (Session 2), adding an element is always a simple operation, requiring the creation of a node that is then linked to the list.

Conversely however, consider the algorithms used to retrieve an item located at a particular index:

- For an array, the index is directly mapped to the matching array location (a simple operation); and
- For a linked list, the list must be traversed until the node matching the index is reached (a more complex operation).

Such examples, of which there are many, tell us that we must be careful when selecting data structures for our applications, regardless of whether we are using pre-built collections or custom-built data structures. However the difference between operations is not always clear, for which we are able to measure the efficiency of an algorithm to determine how the algorithm is likely to behave.

Measuring Algorithm Efficiency

The efficiency of any algorithm can be expressed as a mathematical function, $f(n)$, where *n* represents the number of elements being considered by an algorithm, e.g., the number of elements in an array. The simplest of algorithms involve only a sequence of instructions, possibly with some decision structures (`if`, `switch`, etc.), known as ***linear algorithms***. These algorithms will behave identically, regardless of the number of elements involved. Thus, algorithms such as inserting an element at the head or tail of a linked list (Session 2) are linear algorithms, and their efficiency can be expressed as:

$$f(n) = C$$

Where *C* in the above equation represents some constant value. The most important observation to draw from this however, is that applications spend most of their time where loops are used, regardless of whether iteration or recursion is used, and these form the focus of any examination of algorithm efficiency. Moreover, beyond processing the smallest amounts of data, any linear algorihtms will represent an insigificant amount of computation whereas loops will have a substantial impact on application performance.

The simplest of loops to consider is known as a ***linear loop***, e.g.,

```
for(int i = 1 ; i <= 1000 ; i++)
    // application code
```

This loop is known as a linear loop due to the use of either an addition or subtraction operation to move the enumerator towards the termination condition (`i++` in this example). We have already been using linear loops when performing a linear search (Section 1.6.2).

Note that the actual value that the enumerator is increased/decreased by does not matter. For example, the above loop will complete 1000 iterations, however if the terminating value was

2000 (double the number), then 2000 iterations would be completed (also double). Similarly, if the enumerator was changed by a value of 10 for each loop, i.e., `i += 10`, a terminating condition of 1000 would result in 100 loops whereas a terminating condition of 2000 would result in 200 loops (still doubles, i.e., a linear increase). Thus, the efficiency of linear loops can be expressed as:

$$f(n) = n$$

The next type of loop to consider is known as a ***logarithmic loop***, e.g.,

```
for(int i = 1 ; i <= 1000 ; i *= 2)
    // application code
```

In this case, the enumerator is changed through either a multiplication or division operation. The behaviour of a logarithmic loop is quite different to a linear loop, which is illustrated clearly by considering the specific values of the enumerator for the above loop. For a terminating value of 1000, the following values will be used for the enumerator:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512

The next value of the enumerator (1024) would cause the loop to terminate. Importantly however, if we change the terminating value to 2000, the following values are used for the enumerator:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

Again, the next value of the enumerator (2048) would cause the loop to terminate. From these example enumerator values, we can see for 1000 elements, only 10 iterations would be required in such a loop. More importantly, doubling the number of elements to 2000 would only cause one more iteration of the loop. Given this information, it should be clear that if you are choosing between two algorithms, one linear and the other logarithmic in nature, clearly the logarithmic algorithm should be chosen[1]. More generally, the efficiency of logarithmic loops is stated as[2]:

$$f(n) = \log n$$

We will see logarithmic loops in algorithms examined later in this unit.

The linear loop and logarithmic loop represent the two types of loops that you will encounter in programming. However loops are also often nested, in which case we need to consider additional possibilities. The simplest of these is known as the ***quadratic loop***, as follows:

```
for(int i = 1 ; i <= 10 ; i++)
```

[1] It is not uncommon for modern programmers to struggle with the mathematics expressed in this section. However, the lesson highlighted at this point is straightforward and you should be able to apply this when developing your own code in the future, even if just to avoid linear loops!

[2] This particular algorithm is in fact $f(n) = \log_2 n$ due to its doubling of the enumerator

```
        for(int j = 1 ; j <= 10 ; j++)
            // application code
```

In this case we see two linear loops, one nested inside the other. We calculate the efficiency of nested loops by multiplying the efficiency of the individual loops together, i.e.,

$$\begin{aligned} f(n) &= outer \times inner \\ &= n \times n \\ &= n^2 \end{aligned}$$

Similarly, we can have what is known as a ***linear logarithmic loop***, as follows:

```
    for(int i = 1 ; i <= 10 ; i++)
        for(int j = 1 ; j <= 10 ; j *= 2)
            // application code
```

Where we have a logarithmic loop nested inside a linear loop (or vice-versa), the efficiency of which is specified as:

$$\begin{aligned} f(n) &= n \times \log n \\ &= n\log n \end{aligned}$$

A more complex example can be seen in the ***dependent quadratic loop***, as follows:

```
    for(int i = 1 ; i <= 10 ; i++)
        for(int j = 1 ; j <= i ; j++)
            // application code
```

In this case, the two loops are linear, however the nested linear loop is dependent upon the outer loop. In particular, for each iteration of the outer loop, the nested loop will have the following number of iterations:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

This can be expressed using Gauss's summation formula, as follows:

$$f\left(n\right) = \frac{n(n+1)}{2}$$

Notations for Algorithm Efficiency

Although we can precisely state the behaviour of an algorithm, in general we are only interested in the approximate behaviour of an algorithm. There are three important notations for this purpose:

- Big-O – $O(f(n))$ – a function which grows at least as fast as the algorithm's behaviour (the upper-bound);
- Big-theta – $\Theta(f(n))$ – a function which grows the same as the algorithm's behaviour; and

- Big-omega – □$(f(n))$ – a function which approximates the lower-bound of the algorithm complexity.

Of these, the most common is the big-O notation. Determining the big-O notation for an algorithm is straightforward using the following steps:

1. Determine the function that represents the algorithm, as above;
2. Set the coefficient of each term to 1; and
3. Discard all but the largest term.

For reference, the ranking of terms, from smallest to largest, is as follows:

$\log n$, n, $n \log n$, n^2, n^3, ..., n^k, 2^n, $n!$

For the example loops we considered, above, only the dependent quadratic is not already in big-O notation. In particular, the division by two represents a coefficient which must be set to a value of 1 (step 2), which we achieve by first expanding the equation as follows:

$$\begin{aligned} f(n) &= \frac{n(n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n \end{aligned}$$

then setting these coefficients to 1 (which eliminates the coefficients)

$$f(n) = n^2 + n$$

then finally discarding all but the largest term, i.e.,

$$f(n) = n^2$$

Moreover, a dependent quadratic is $O(n^2)$ (read as "on the order of n^2").

7.2. Sorting Algorithms

Sorting algorithms are regularly used in programming. Critically, as we will learn, sorting algorithms also represent a very expensive operation, i.e., an inefficient operation, which are best avoided wherever possible. Before we examine sorting algorithms, there are a number of fundamental concepts that must be introduced first.

Sorting algorithms can be classified into

- ***Internal sorts*** – all sorting is performed in memory; and
- ***External sorts*** – only data currently being sorted is in memory, the remaining data is on secondary storage (disk).

Data can be sorted either into

- ***Ascending order***, the most common, e.g., 0-9, A-Z, etc; or
- ***Descending order***, e.g., 9-0, Z-A, etc.

A ***stable sort*** represents a sorting algorithm that will maintain any existing ordering on the data, i.e., if two or more elements have the same key, the first instance of that key appearing in a list before sorting will remain the first instance of that key in the list after sorting. This is illustrated in Figure 7.1, where the key is indicated as a number, and the order of their appearance is indicated by a letter, i.e., the first appearance of key '1' is indicated as '1a', the second '1b', and so on. After sorting is complete, the data is ordered by keys (1, 2, 3), however the order those keys appeared in the unsorted list remains the same (a, b, c, etc.).

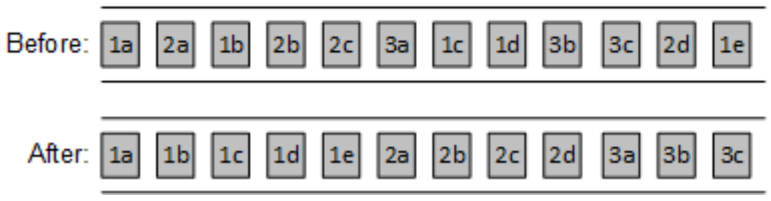


Figure 7.1: Sort Stability

Also critical to understanding sorting algorithms is the concept of a ***sort pass*** (or just ***pass***). When performing a sort, the data stored in a list will be traversed many times, each of which is referred to as a pass. Each of these may traverse either the whole list, or only part of the list. Finally, many sorting algorithms require two elements in the list to be exchanged, for which we will use a common method `Swap`, as follows:

```
ASSUMPTION: Two elements from a list being sorted, first and second, are passed by-reference
Swap(first, second)
begin
    temp ← first
    first ← second
    second ← temp
end
```

Algorithm 7.1: Swap method used in many sorting algorithms

In this session we only consider four internal sorts:

- The bubble sort (including the shaker sort variation);
- The selection sort;
- The insertion sort; and
- The quick sort.

7.2.1. Bubble Sort

The bubble sort is a reasonably straightforward algorithm, which repetitively compares two adjacent elements in the list and exchanging them if they are out of order. Each pass of this sorting algorithm results in one element being placed in its final position at the end of the list. There are three variations of the bubble sort (assuming an ascending sort)[3]:

- Bubble-up – the smallest element in the list is "bubbled" to the start of the list;
- Bubble-down – the largest element in the list is "bubbled" to the end of the list;
- Shaker sort – alternating passes of bubble-up and bubble-down.

The following illustrates the steps involved in sorting five elements using bubble-up. Underlining is used to indicate the two elements being compared (and possibly then exchanged), and *italics* is used to indicate elements that are in their final position and are no longer considered:

- Pass 1:
  - 10, 3, 6, 4, 1
  - 10, 3, 6, 1, 4
  - 10, 3, 1, 6, 4
  - 10, 1, 3, 6, 4
- • Pass 2:
  - *1*, 10, 3, 6, 4
  - *1*, 10, 3, 4, 6
  - *1*, 10, 3, 4, 6
- • Pass 3:
  - *1*, *3*, 10, 4, 6
  - *1*, *3*, 10, 4, 6
- Pass 4:
  - *1*, *3*, *4*, 10, 6
- Output:
  - *1*, *3*, *4*, *6*, *10*

The algorithm for bubble-up is as follows:[4]

```
BubbleUpSort(array, arraySize)
begin
    firstSorted ← –1
    while firstSorted < arraySize – 1
        current ← arraySize – 2
        while current > firstSorted
            if array[current] > array[current + 1]
                Swap(array[current], array[current + 1])
            end-if
            current ← current – 1
        end-while
        firstSorted ← firstSorted + 1
    end-while
end
```

[3] Note that the exact semantics/direction of the bubble-up and bubble-down sorts are inconsistent in the literature. Sometimes they appear as described here, other times they are presented as the reverse directions. Regardless, bubble-up and bubble-down represent performing a bubble sort in opposite directions.
[4] Note that while loops have been used for clarity, however in most programming languages a for loop may result in code that is clearer.

Algorithm 7.2: Bubble sort algorithm (bubble-up)

If you consider the above algorithm for efficiency, you will notice that there are two linear loops:

```
while firstSorted < arraySize – 1
          ...
          while current > firstSorted
                    ...
                    current ← current – 1
          end-while
          firstSorted ← firstSorted + 1
end-while
```

Thus, the bubble sort algorithm is $O(n^2)$ (including bubble-up, bubble-down, and shaker sort variations).

7.2.2. Selection Sort

The selection sort takes a very different approach to the bubble sort. You may have noticed that with each exchange in the bubble sort, an element can only move one step closer to its final resting location. The selection sort improves upon this, by traversing through the list of unsorted elements to identify the smallest value and then exchanging that element with the first element in the list, moving it straight to its final location (assuming an ascending sort).

The following illustrates the steps involved in sorting five elements using selection sort. Underlining is used to indicate the two elements being compared (and possibly then exchanged), **emboldening** is used to indicate the current known smallest element, and *italics* is used to indicate elements that are in their final position and are no longer considered:

- Pass 1:
  - **10**, 3, 6, 4, 1
  - 10, **3**, 6, 4, 1
  - 10, **3**, 6, 4, 1
  - 10, **3**, 6, 4, 1
    Note: the last element is found to be the smallest and then exchanged with the first element at this point.
- Pass 2:
  - *1*, **3**, 6, 4, 10
  - *1*, **3**, 6, 4, 10
  - *1*, **3**, 6, 4, 10
- Pass 3:
  - *1*, *3*, **6**, 4, 10
  - *1*, *3*, 6, **4**, 10
- Pass 4:
  - *1*, *3*, *4*, **6**, 10
- Output:
  - *1*, *3*, *4*, *6*, *10*

The algorithm for selection sort is as follows:

```
SelectionSort(array, arraySize)
begin
    firstUnsorted ← 0
    while firstUnsorted < arraySize – 1
        smallest ← firstUnsorted
        current ← smallest + 1
        while current < arraySize
            if array[current] < array[smallest]
                smallest ← current
            end-if
            current ← current + 1
        end-while
        Swap(array[firstUnsorted], array[smallest])
        firstUnsorted ← firstUnsorted + 1
    end-while
end
```

Algorithm 7.3: Selection sort algorithm

Given that there are much less calls to the Swap method (many calls to Swap per pass reduced to one per pass), this algorithm does improve the speed of sorting somewhat. However, we again see two linear loops:

```
while firstUnsorted < arraySize – 1
    ...
    while current < arraySize
        ...
        current ← current + 1
    end-while
    ...
    firstUnsorted ← firstUnsorted + 1
end-while
```

Thus, selection sort remains $O(n^2)$.

7.2.3. Insertion Sort

Both the bubble sort and selection sort algorithms traversed the entire list (of unsorted elements) in each pass. The insertion sort removes the need for traversing the entire list by conceptually dividing the list into two: sorted and unsorted. The first element in the unsorted list is then removed to a temporary location, then the elements in the sorted list are shuffled to make a space available where the unsorted element should be placed. Finally, the previously removed unsorted element is placed in the sorted list. The strategy used by insertion sort is also commonly used by card players to order their hand of playing cards.

The following illustrates the steps involved in sorting five elements using insertion sort. Underlining is used to indicate a spare slot in the array/list, caused by removing the first unsorted element to a temporary variable. *Italics* is used as before to indicate elements that are in their final position and are no longer considered. Note that the insertion sort algorithm begins by assuming that the sorted list consists of the first element in the list (a list of only one element must be ordered, it can never be out of order!):

- Pass 1:
  - *Hold: 3*
  - *10*, _, 6, 4, 1
  - _, *10*, 6, 4, 1
  - *3*, *10*, 6, 4, 1
- Pass 2:
  - *Hold: 6*
  - *3*, *10*, _, 4, 1
  - *3*, _, *10*, 4, 1
  - *3*, *6*, *10*, 4, 1
- Pass 3:
  - *Hold: 4*
  - *3*, *6*, *10*, _, 1
  - *3*, *6*, _, *10*, 1
  - *3*, _, *6*, *10*, 1
  - *3*, *4*, *6*, *10*, 1
- Pass 4:
  - *Hold: 1*
  - *3*, *4*, *6*, *10*, _
  - *3*, *4*, *6*, _, *10*
  - *3*, *4*, _, *6*, *10*
  - *3*, _, *4*, *6*, *10*
  - _, *3*, *4*, *6*, *10*
  - *1*, *3*, *4*, *6*, *10*
- Output:
  - *1*, *3*, *4*, *6*, *10*

The algorithm for insertion sort is as follows (note the absence of any calls to the Swap method):

```
InsertionSort(array, arraySize)
begin
    firstUnsorted ← 1
    while firstUnsorted < arraySize
        hold ← array[firstUnsorted]
        current ← firstUnsorted - 1
        while current >= 0 AND data[current] > hold
            array[current + 1] = array[current]
            current ← current – 1
        end-while
        array[current + 1] ← hold
        firstUnsorted ← firstUnsorted + 1
    end-while
end
```

Algorithm 7.4: Insertion sort algorithm

Although the insertion sort algorithm offers the best performance of the sorting algorithms considered so far, it still makes use of two linear loops:

```
while firstUnsorted < arraySize
    ...
```

```
        while current >= 0 AND data[current] > hold
            ...
            current ← current – 1
        end-while
        ...
        firstUnsorted ← firstUnsorted + 1
    end-while
```

Thus, the insertion sort is still $O(n^2)$.

7.2.4. Quick Sort

The quick sort algorithm takes a fundamentally different approach to sorting data, including the use of a recursive algorithm, to achieve vastly improved performance. Interestingly, the functionality of the quick sort algorithm has more in common with the bubble sort algorithm than the others – both the bubble sort and quick sort algorithms are known as ***exchange sorts***.

Recall that the operation of the bubble sort moves each element one step closer to its destination. The quick sort aims to improve this by moving elements much closer to their destination in each exchange. To do this, quick sort begins by selecting an element from the unsorted data referred to as the ***pivot*** element. The aim is to select a pivot which, when moved to its final position, will appear relatively close to the middle of the sorted data. In the first version of quick sort, the first unsorted element was used for this purpose. This selection has been improved however by selecting the middle of three elements from the list, using an algorithm known as ***median left*** (assuming ascending sort):

```
MedianLeft(array, left, right)
begin
    middle ← (left + right) / 2 // integer division
    if array[left] > array[middle]
        Swap(array[left], array[middle]
    end-if
    if array[left] > array[right])
        Swap(array[left], array[right])
    end-if
    if (array[middle] > array[right])
        Swap(array[middle], array[right])
    end-if
    Swap(data[left], data[middle])
end
```

Algorithm 7.5: Quick sort's median left algorithm

The median left algorithm works by ordering the left-most, middle, and right-most unsorted elements. The middle of these elements is then considered to be the pivot element, which in the last step is moved to the left-most location. This results in the following ordering of elements:

{pivot element} {unsorted elements}

The next stage of the quick sort algorithm, ***partitioning***, processes the unsorted elements and results in the following ordering of elements:

{pivot element} {elements < pivot} {elements >= pivot}

This is achieved by searching from the left-most end of the unsorted elements for the first element that is greater than or equal to the pivot, then searching from the right-most end of the unsorted elements for the first element that is less than the pivot. These elements are then exchanged, and the process continues until the two searches reach each other.

The next step is to exchange the pivot element with the right-most element in first partition of elements ({elements < pivot}), which results in the following ordering of elements:

{elements < pivot} {pivot element} {elements >= pivot}

Importantly, this means that the pivot element is now in its final place in the list and will not be moved further, however the elements in the two partitions remain unsorted. All of the steps to this point represent a pass of the quick sort algorithm, which results in one element being placed in its final position

To continue sorting the list, the quick sort algorithm is then repeated for both of the unsorted partitions (a recursive call). This process can continue until the entire list is sorted. Importantly however, recall that recursion involves significant overhead that would otherwise reduce the performance of the quick sort algorithm. To avoid the worst of the overheads, a threshold is introduced, which represents the minimum number of elements that will be sorted by the quick sort algorithm (in the overall call or in a recursive call for one of the partitions). If the number of elements is less than the threshold, the overhead of recursion will outweigh the benefits of the quick sort algorithm, thus the insertion sort is used instead.

This results in the following algorithm for quick sort:

```
QuickSort(array, left, right)
begin
    if right – left < 1
        end
    end-if
    if (right – left < threshold)
        InsertionSort(array, left, right)
    else
        MedianLeft(array, left, right)
        leftHigh ← left + 1
        rightLow ← right
        while leftHigh <= rightLow
            while leftHigh <= right && array[leftHigh] < array[left]
                leftHigh ← leftHigh + 1
            end-while
            while (rightLow > left && array[rightLow] >= array[left])
                rightLow ← rightLow - 1
            end-while
            if leftHigh <= rightLow
                Swap(array[leftHigh], array[rightLow])
                leftHigh ← leftHigh + 1
                rightLow ← rightLow – 1
            end-if
        end-while
        Swap(array[left], array[leftHigh – 1])
```

```
            if left < rightLow
                QuickSort(array, left, rightLow – 1)
            end-if
            if leftHigh < right
                QuickSort(array, leftHigh, right)
            end-if
        end-if
end
```

Algorithm 7.6: Quick sort algorithm

Note that the InsertionSort algorithm is modified slightly for use with quick sort, through the introduction of left and right boundaries – the insertion sort algorithm shown earlier effectively uses a left boundary of 0 (the first element in an array), and a right boundary of size – 1 (the last element in an array).

Considering the additional code of the quick sort algorithm, it may initially seem strange that this algorithm performs vastly better than the simpler sorting algorithms described earlier. To understand why, it is important to consider the use of loops in this algorithm:

- The three while loops used in the QuickSort method in total will access every element in the unsorted data collectively, i.e., $O(n)$:
  - The first nested loop will seek from the left until it reaches an element >= pivot;
  - The second nested loop will seek from the right until it reaches an element < pivot;
  - The outer loop keeps the two nested loops going until they meet in at some middle point;
- After the pivot is moved into its final place, the list is divided into two, i.e., the equivalent of a logarithmic loop or $O(\log n)$.

Thus, the overall efficiency of the quick sort algorithm is $O(n \log n)$.

Task 7.1. Sorting Algorithms and their Performance

Objective: Understanding the behaviour of algorithms and how their efficiency does not relate to the number of lines of code is a critical lesson for all programmers. In this task we explore these concepts through the implementation of various sorting algorithms and compare their performance.

In this task we will implement the sorting algorithms covered in this chapter. For this purpose, two files have been provided (see provided code in CloudDeakin):

- Program.cs – this code file is complete and includes the Main method which will drive the testing of your sorting algorithms and measure the time taken to complete the sorting (this file will not change and does not need to be submitted); and
- ArraySorter.cs – this code file contains a skeleton of the class invoked by the provided Main method along with the completed implementation of the bubble-up variation of the bubble sort algorithm.

Your task is to complete the provided `ArraySorter` class as follows:

a. Implement the bubble-down and shaker variations of the bubble-sort algorithm (you will need to work these out given the concepts above);
b. Translate the Selection sort algorithm provided above into the `Selection()` method;
c. Translate the Insertion sort algorithm provided above into the `InsertionWorker()` method, adjusting the boundaries of the algorithm to match the parameters to the method; ***Hint: start by just translating the algorithm as is, ignoring the boundaries. Then work on replacing the boundaries with the parameters before moving onto quick sort. Note that this task will be simple if you are able to read and understand the algorithm provided.***
d. Translate the Quick sort algorithm provided above into the `QuickSortWorker()` method.

As a final task (which does not need to be submitted), try recording in a spreadsheet the running times for each sorting algorithm for different problem sizes, e.g., 5,000, 10,000, 15,000, 20,000, etc. and preparing a graph of their performance. Observe the different times taken by each of the algorithms, particularly the variation in performance of the bubble sort, selection sort, and insertion sort algorithms which are all $O(n^2)$, and the significant improvement of the quick sort algorithm! Also try experimenting with different threshold values to see how it impacts the performance of the sorting algorithm.

8

8. Searching and Hashing

The problem of searching for data based on some criteria is one of the most fundamental problems faced by computer programmers. In many tasks we complete in computing, you will find a search problem. For example, consider what happens when you visit a web site using your web browser. Students who have studied computer networks will realise that one of the first steps is to convert the name of the host, e.g., www.google.com, to an IP address that can be used to send data to the relevant server. The number of different host names in use on the Internet, each of which is mapped to an IP address, represents a vast amount of data that must be searched for each time it is accessed regardless of the application. Clearly the solution to this search problem must be structured in a way in which the computational load can be substantially reduced/minimised for the Internet to be functional.

Solving such large search problems is not a trivial matter and involves the application of multiple techniques to ensure high speed results. In this chapter we consider the problem of searching by first reviewing the linear search and what improvements can be made to this algorithm before we introduce one of the main techniques for improving search times, the concept of hashing. This should highlight one of the most important lessons in programming, that you may have already realised in our studies in this unit to date – improvements in performance can be gained either by improving an existing algorithm, or by introducing structure into our data to provide new approaches to solving a problem.

Objectives

In this session you will learn

- The sentinel search optimisation to the linear search algorithm;
- The probability search optimisation to the linear search algorithm;
- The ordered list search optimisation to the linear search algorithm;
- The binary search algorithm; and
- The concept of hashing and how it is applied to reduce the search problem.

8.1. Searching

In Section 1.7.2 we introduced the concept of the linear search, the simplest algorithm we can use to search for data. The algorithm for a linear search is repeated below for reference[1]:

```
ASSUMPTION: An array named array already exists
ASSUMPTION: A variable named lastUsed contains the index of the last used element in the array
ASSUMPTION: A variable named target contains the desired search target
targetIndex ← -1
for i ← 0 to lastUsed
    if array[i] = target
        targetIndex ← i
        break;
    end-if
end-for
// if targetIndex = -1 then element not found, otherwise target is found at index stored in targetIndex
```

Algorithm 8.1: Searching for data in an array using a linear search

For those who were paying close attention to our examination of algorithm complexity in Section 7.2, it should be immediately apparent that the above algorithm is $O(n)$, i.e., the algorithm is linear (also implied in its name) and the amount of time required to complete the algorithm is directly proportionate to the amount of data to be considered. Although not as bad as sorting algorithms which were $O(n^2)$ or best case $O(n\log n)$, in general we would still want to avoid using this algorithm if possible.

8.1.1. Improving the Linear Search

It is actually possible to improve the performance of the linear search by applying a number of techniques that we examine in this section. Importantly, these improvements do not fundamentally change the performance of the linear search, i.e., the algorithm remains $O(n)$, thus do not make the linear search any more appealing for use in our programs. However the techniques used in these algorithms can be used in solving other problems and are still useful to know.

Sentinel Search

We begin by applying one of the simplest techniques for optimising the performance of an algorithm[2]:

> *"When the inner loop of a program tests two or more conditions, we should try to reduce the testing to just one condition."*

To explain this, reconsider how the loop in the linear search can terminate:

[1] Note that although the linear search algorithm is shown written for use with an array, it can easily be adapted for any linear data structure (array, linked list, etc).
[2] Donald E. Knuth, "The Art of Computer Programming", Volume 3 Sorting and Searching, Second Edition, Addison-Wesley, 1998.

1. The condition on the for loop will terminate the loop when the end of the array is reached; and
2. The if statement, containing a break statement, will terminate the loop if the target element is found.

To apply the improvement described above, we need to eliminate one of these conditions, however it is not immediately apparent how we can do this. If we eliminate the first condition, the loop will attempt to continue searching beyond the end of the array, which would result in a run time error (an exception in C#). Alternatively, if we eliminate the second condition, the algorithm will never check to see if the the target element is found and the algorithm is no longer useful.

By considerating the conditions independently, as above, it is unclear how we could eliminate either terminating condition. However by considering the conditions together to understand the overall behaviour of the algorithm, it is apparent that if the target element is found the end of the array will never be reached, making the first condition redundant. Therefore if we could guarantee that the target element will be found we could eliminate the first condition.

This may initially seem to be impossible – how would it be possible guarantee that an element is stored in an array without first searching the array to see if it is there? The answer is to deliberately place the target at the end of the array, knwon as the ***sentinel***. Now when we perform the linear search, we only need to check for the target data. If the target is found at the end of the array (the data we added for the search), the target is not present in the array. However if the target is found before the end of the array, that data was not added for the purpose of searching and was present beforehand. before we begin the linear search. The linear search algorithm is updated to implement the sentinel search as follows[3]:

```
ASSUMPTION: An array named array already exists which has at least one unused element at the end
ASSUMPTION: A variable named lastUsed contains the index of the last used element in the array
ASSUMPTION: A variable named target contains the desired search target
array[lastUsed + 1] = target
targetIndex ← 0
while NOT array[targetIndex] = target
        targetIndex ← targetIndex + 1
end-while
// if targetIndex = lastUsed + 1 then element not found, otherwise target is found at targetIndex
```

Algorithm 8.2: Linear search with Sentinel Search optimisation

Probability Search

Recall our earlier observation that the linear search algorithm will terminate upon finding the target data. This implies that if the target data is stored close to the start of the array, the linear search algorithm will return very quickly (approaching $O(1)$). However, data that is stored closer to the end of the array will take much longer to find (approaching $O(n)$).

[3] Note that in C# it would be important to reset the `array[lastUsed + 1]` element to its default value, i.e., `array[lastUsed + 1] = default(data_type);`

In many applications, only a subset of data is retrieved regularly. For example, the Deakin University student database will contain millions of records for both past and present students, however the records for past students will rarely be retrieved. Additionally, the records for students who have recently joined the University will likely be retrieved more regularly than other students currently attending the University, at least until their enrolment is finalised. If we can modify the way data is ordered in the array to ensure that current students appeared at the start of the array, the overall performance of the application would be improved.

The second improvement we consider, the probability search, achieves this outcome by moving elements closer to the start of the array after they have been found, as follows:

```
ASSUMPTION: An array named array already exists
ASSUMPTION: A variable named lastUsed contains the index of the last used element in the array
ASSUMPTION: A variable named target contains the desired search target
targetIndex ← -1
for i ← 0 to lastUsed
        if array[i] = target
                targetIndex ← i
                break;
        end-if
end-for
if targetIndex > 0
        Swap(array[targetIndex - 1], array[targetIndex])
        targetIndex ← targetIndex – 1
end-if
// if targetIndex = -1 then element not found, otherwise target is found at index stored in targetIndex
```

Algorithm 8.3: Linear search with the Probability Search optimisation

Note: Although both the Sentinel Search and Probability Search have been presented separately, they can in fact be combined together.

Ordered List Search

The last improvement that we consider for the linear search is possible only if the data is ordered, either because the data has been sorted (Section 7.3) or if the data was maintained in order, such as our implementation of an ordered linked list (Task 2.1). Understanding this optimisation is straightforward. Consider looking up a name in a (printed) telephone directory or a word in a (printed) dictionary. If you were searching these documents, you would keep in mind the relative location of the data (name/word) you were searching for. In particular, if you were searching for the name 'Jones' in the telephone directory, you would not continue searching after reaching names beginning with the letter 'K'. Thus, we can improve the linear search by terminating the search early if the value we are currently considering would normally appear after our desired target value.

Importantly, it no longer makes sense to apply the probability search optimisation, which would result in the data in the array no longer being ordered. It would be possible to use the sentinel search optimisation, however it is important to recognise that as long as there is at least one element in the list that should appear after the target value, there is no need for a sentinel. Instead, we solve this problem by first checking to see if the target would appear after the last element in the list, i.e., if the target value would normally appear after the last element of data stored in the array, there is no point searching the array (the target cannot be in the array). If this is not the case, i.e., the target value would normally appear before the last element in the array, we can proceed to search the array until we either reach the target element or the current element being considered would appear after the target data, as follows (can be used either for an array or linked list):

```
ASSUMPTION: An array named array already exists containing data that is ordered
ASSUMPTION: A variable named lastUsed contains the index of the last used element in the array
ASSUMPTION: A variable named target contains the desired search target
targetIndex ← -1
if target <= array[lastUsed]
    i ← 0
    while(array[i] < target)
        i ← i + 1
    end-while
    if(array[i] = target)
        targetIndex ← i
    end-if
end-if
// if targetIndex = -1 then element not found, otherwise target is found at index stored in targetIndex
```

Algorithm 8.4: Linear search with the Ordered List Search optimisation

8.1.2. Binary Search

When discussing the last optimisation that we considered to the linear search, the ordered list search, we considered real-world analogies of searching for entries in (printed) telephone directories or dictionaries. If you review the algorithm presented however, it still does not reflect how we in fact physically use such publications. In particular, the algorithm began the search at the start of the data, equivalent to starting a search in a telephone directory/dictionary from the letter 'A'. In continuing the analogies, we would normally start by opening the telephone directory/dictionary close to the data that we were looking for before performing other improvements to our search, e.g., fanning the pages to determine a much closer location to start our search.

Although we cannot easily mimic this behaviour in an algorithm, e.g., what would the equivalent of fanning pages be in terms of searching an array?, we can learn from this approach and improve our algorithm in what is known as a binary search. As for the ordered list search, the binary search requires the data to be stored in order. Unlike the ordered list search however, the binary search cannot be applied to data stored in a linked list, as it requires the ability to access the stored data randomly (recall that a linked list only allows data to be accessed sequentially[4]).

The binary search works in the following manner:

- Compare the target value against the element in the middle of the stored data;
- If the target value should appear before middle element, repeat this process for the elements to the left;

[4] Note that the equivalent of the binary search can be implemented in a linked structure using what is known as the binary search tree, which we examine in Section 9.2.1.

- If the target value should appear after the middle element, repeat this process for the elements to the right;
- Continue until the target value is found.

Note how the binary search effectively halves the number of elements to be checked with each comparison, i.e., the binary search a logarithmic algorithm. Also note that although the algorithm sound recursive in nature, it can be implemented simply using iteration, as follows:

```
ASSUMPTION: An array named array already exists containing data that is ordered
ASSUMPTION: A variable named lastUsed contains the index of the last used element in the array
ASSUMPTION: A variable named target contains the desired search target
left ← 0
right ← lastUsed
while left <= right
        middle ← (left + right) / 2 // integer division
        if target > array[middle]
                left ← middle + 1
        else if target < array[middle]
                right ← middle – 1
        else
                break
        end-if
end-while
if left <= right
        // element found at array[middle]
else
        // element not found, would normally appear either one element to the left/right of middle
end-if
```

Algorithm 8.5: Binary search algorithm

8.2. Hashing

In considering the above discussion to solving the search problem, it should be apparent that the biggest improvement was made in the binary search, which rapidly reduced the number of elements that needed to be checked by eliminating half of the elements from the search problem at each step, i.e., the search space (amount of data to be searched) was reduced by half at each step. Unfortunately however, the binary search imposed two requirements: (1) an array must be used; and (2) the data must be ordered. Hashing is an approach that also significantly reduces the search space however it does not require the data to be ordered/sorted. Importantly, arrays are still used when implementing hashing, however arrays form part of the data structure and are used to reduce the search space.

Hashing works by applying one or more algorithms to the data (or key in a key/value pair) to determine the location in the array where the data will be stored, which we examine in Section 8.2.1. For example, consider a collection of student records where to find a student with ID 211123456 you could just use:

```
    Student target = studentList[211123456];
```

Clearly this could be achieved by using an array of sufficient size, and would result in a search algorithm that was $O(1)$, however the size of the array would be prohibitive[5]. Moreover, the array would be sparsely populated, i.e., not many of the array locations would actually have data stored in them, and would be wasteful.

In practical terms, the algorithms we use are written to produce a hash value within a smaller range, e.g., to 0 and 999. It is important to note that this can result in multiple values/keys with the same hash, referred to as ***collisions***, e.g., clearly if we were storing more than 1000 students at least two students must have the same hash value in the range 0-999. There are a number of different techniques we can use to resolve collisions, which we examine in Section 8.2.2. Although achieving searches that are $O(1)$ is extremely difficult, we are still able to substantially reduce the size of the search space for any search problem.

8.2.1. Hashing Techniques

The first problem we consider is how to determine a hash value for given input data (also referred to as a hash code). The example above used student ID numbers, which are already in numeric form. However the data to be stored is often not numerical in nature, e.g., a person's name, which is text. However to a computer, textual data is in fact just a sequence of numbers (ASCII or Unicode encoding), and we can use these numbers to represent a numeric value[6], e.g., by combining the ASCII/Unicode values for each character in a string using addition, bitwise XOR, etc.

Once an initial numeric value is determined, we apply the hashing algorithm to scale that value to the target range, i.e., the size of the array we have selected. The objective of any algoirthm/s we select for this purpose is to have a relatively even distribution of hash values across the range of the array, i.e., if we were storing 10,000 students in a hash table (array size) of 1000, ideally our hashing algorithm would result in 10 students per hash value. The best way to do this varies depending upon the problem being considered, in particular the nature of the data being stored, however a number of techniques are commonly used, as follows.

Direct Method

The direct method was actually presented above, i.e., the data is used as is with no scaling. Although the direct method was not useful for student ID numbers (resulting in a large and sparsely populated array), it remains useful for very restricted data sets, e.g., days of the week (1-7), days of the month (1-31), months of a year (1-12) and so on.

Modulo-Division Method

[5] Assuming a 32 bit address space which would require four byte memory addresses, the array would require just over 800MB of memory before any data was actually stored. Similarly, 1.6GB would be required for a 64 bit address space / eight byte memory addresses.
[6] Note that in C#, this task is done for you and a numerical hash value for any string can be retrieved by invoking the built-in GetHashCode() method, e.g., someString.GetHashCode()

The modulo division method simply requires the application of the modulus operator (remainder after division) to scale a value down to a limited range, e.g.,

211123456 % 1000 = 456

Unfortunately, this approach can obviously result in many collisions, e.g., students enrolled in the same year:

211000456 % 1000 = 456
211001456 % 1000 = 456
211011456 % 1000 = 456
211111456 % 1000 = 456

or even students enrolled in different years:

210000456 % 1000 = 456
212000456 % 1000 = 456
213000456 % 1000 = 456
214000456 % 1000 = 456

However it is important to focus on whether or not the hashing algorithm will result in a relatively even distribution of hash values, thus this approach may be adequate and has the advantage of being simple. In the case of a student ID, this method would probably work quite well given that the last three digits of a student number would vary significantly. It is also worth noting that the modulo-division method is commonly applied after other techniques to simplify the scaling of hash values to the desired target range.

Digit Extraction Method

In the digit extraction method, individual digits are extracted from the input hash value and combined together to form the hash value. In this example, we have used the first, middle, and last digits:

$\underline{2}111\underline{2}345\underline{6} \rightarrow 226$
$\underline{2}110\underline{0}045\underline{6} \rightarrow 206$
$\underline{2}110\underline{0}145\underline{6} \rightarrow 206$
$\underline{2}110\underline{1}145\underline{6} \rightarrow 216$
$\underline{2}111\underline{1}145\underline{6} \rightarrow 216$

Note that some collisions have still occurred, although it appears to be less so. Importantly however, this approach would result in a poor distribution for student ID numbers, given that there is minimal variation in the first digit (all students who have applied since 2010 have an ID starting with 2, so far at least).

Mid-Square Method

In the mid-square method, the input value is first squared (multiplied by itself) and the middle digits are then extracted for use as the hash value, i.e.,

$211123456^2 = 4457311\underline{367}3383936 \rightarrow 367$
$211000456^2 = 4452119\underline{243}2207936 \rightarrow 243$
$211001456^2 = 4452161\underline{443}4119936 \rightarrow 443$
$211011456^2 = 4452583\underline{456}3239936 \rightarrow 456$
$211111456^2 = 4456804\underline{685}4439936 \rightarrow 685$

In this example, we can see that all collisions have been eliminated and there appears to be a reasonable distribution of values across the target range.

Folding Method – Fold Shift

In the fold shift method, the input number is divided into an equal number of digits which are then added together to form the hash value, i.e.,

$211123456 \rightarrow 211 + 123 + 456 = 790 \rightarrow 790$
$211000456 \rightarrow 211 + 000 + 456 = 667 \rightarrow 667$
$211001456 \rightarrow 211 + 001 + 456 = 668 \rightarrow 668$
$211011456 \rightarrow 211 + 011 + 456 = 678 \rightarrow 678$
$211111456 \rightarrow 211 + 111 + 456 = 778 \rightarrow 778$

In this example, although all collisions have been eliminated you can see that there is a clustering of hash values with three values in the high six hundreds and two in the high seven hundreds. Due to the strong variability of the last three digits of a student number, the pattern above most likely occurs due to the use of the same last three digits (456) and this algorithm would still likely result in a relatively balanced distribution of values across the target range.

Folding Method – Fold Boundary

In the fold boundary method, the same approach is used as fold shift except every second number in the addition is reversed, i.e.,

$211123456 \rightarrow 211 + \underline{321} + 456 = 988 \rightarrow 988$
$211000456 \rightarrow 211 + \underline{000} + 456 = 667 \rightarrow 667$
$211001456 \rightarrow 211 + \underline{100} + 456 = 767 \rightarrow 767$
$211011456 \rightarrow 211 + \underline{110} + 456 = 777 \rightarrow 777$
$211111456 \rightarrow 211 + \underline{111} + 456 = 778 \rightarrow 778$

Note that underlines have been used to highlight those numbers that have been reversed. The results of this algorithm are similar to the fold shift method, with the same cause of clustering of results and it is still likely that this approach would result in a relatively balanced distribution of values across the target range.

Other Methods

Finally, note that there are many other methods that could be used to determine a hash code, including the application of different mathematical operations, e.g., e^x, application of different algorithms to determine an input value, e.g., MD5, or simply by combining or adjusting one or more of the above approaches.

8.2.2. Collision Resolution

Collisions are caused by two or more sets of input data having the same hash value which would obviously cannot be stored in a single array location. In this section we examine common techniques for overcoming this problem.

Linear Probe Resolution

The linear probe is a very simple solution to the problem – if the array location referred to by a hash value already has an element stored, increment the hash value until a free slot is found, i.e.,

211123456 % 1000 = 456 → 456(stored)
211000456 % 1000 = 456 → 456(!) → 457(stored)
211001456 % 1000 = 456 → 456(!) → 457(!) → 458(stored)
211011456 % 1000 = 456 → 456(!) → 457(!) → 458(!) → 459(stored)
211111456 % 1000 = 456 → 456(!) → 457(!) → 458(!) → 459(!) → 460(stored)

Note that this approach requires that the array still be large enough to store all of the input data. If the array were to be resized, note that the hash values would need to be recalculated before data could be stored in an array (not just a simple copy of data as we saw in Section 1.6

Linked List Resolution

In this approach, rather than using a simple array, each array location contains a single element and an overflow area that is used to store any further values that have the same hash value. Note that although the name of this approach suggests that a linked list is used (and usually is), any data structure could in fact be used for this purpose. Also note that to simplify the code for storing and retrieving data, there is usually no differentiation between the first element and overflow elements, i.e., an array of linked lists is used instead.

For example, if the above sequence of student IDs were inserted using modulo division to calculate a hash, the following could result:

```
...
array[455] → null
array[456] → 211123456 → 211000456 → 211001456 →
211011456 →
211111456 → null
array[457] → null
...
```

Bucket Resolution

In this approach, rather than each hash value referring to a single array location, multiple array elements are used for each hash value, each grouping of elements for a single hash value are referred to as a bucket. For example, if a bucket size of 10 elements was used, the array elements at indexes 4560 → 4569 would be used to store elements with a hash value of 456. Alternatively, a two dimensional array could be used for the same purpose, with the first index used for the hash value and the second index used for the individual element in the bucket.

Under this approach, in theory no collision scheme would be required until the bucket for an hash value has been filled. Critically however, this approach is much more complex than the linked list resolution approach and would be unlikely to be used if the linked list resolution approach were possible[7].

Task 8.1. Searching and Hashes

Objective: In this problem we examine the implementation of optimisations to the linear search and hashing. Although the linear search is often avoided in all but the simplest of search problems, the techniques used in the optimisations are relevant to many programming problems and it is important that you gain experience with them. Hashing is a very common technique used in programming and it is important that you master its implementation as soon as possible.

You are required to complete the following tasks:

a. Modify the Contains method in the Vector class from previous weeks to implement both the sentinel search and probability search optimisations.
b. Write a Main method which performs the following functions:
   - Creates a `Vector` of integers and stores the values 1–10
   - Uses a `foreach` loop to display the contents of the `Vector`
   - Tests whether the values 0–11 are stored in the above `Vector` using your updated `Contains` method
   - Re-displays the contents of the Vector using a foreach method
c. To demonstrate your understanding of hashing, continue writing the `Main` method by adding the following features:
   - Creates an array of 10 integer `Vector` objects, i.e.,

     ```
     Vector<int>[] hashedVectors = new Vector<int>[10];
     for (int i = 0; i < 10; i++)
         hashedVectors[i] = new Vector<int>();
     ```

   - Creates and stores 100 random integers using the modulo-division method
   - Displays the contents of each vector to demonstrate how the data was distributed among each location using this method

[7] Note that linked list resolution is not always possible or desirable, e.g., when hashing is used to improve the storage and retrieval of data in a file on disk rather than in the memory of a computer.

9

9. Trees

Trees, when used well, are a useful data structure for the storage and retrieval of data in a program. Trees represent a different type of data structure to what we have studied in the unit so far – so far all of our data has stored in a linear format, e.g., an array, a linked list, a stack, a queue, and so on. In this session we begin by introducing the concepts and terminology of trees in general, before undertaking a closer examination of binary trees and the Binary Search Tree in general. We examine the problem of imbalance which can affect the Binary Search Tree and discuss one of the known solutions to this problem, the AVL Tree. We finish the session with an examination of the heap data structure, which effectively overlays a tree structure onto an array to solve different problems, including the problem of the priority queue and sorting.

Objectives

In this session you will learn

- The concepts of trees and binary trees in particular;
- How to implement and use a Binary Search Tree;
- How the problem of balance in the Binary Search Tree is solved by the AVL Tree; and
- The concepts of the Heap.

9.1. General Concepts

Trees are a data structures that have a number of similarities with linked lists. Unlike linked lists however, which are considered a linear list, trees are non-linear in nature. tree is a non-linear list that consists of nodes, representing a finite set of elements (typically one node represents a single element of data stored in the tree), and branches, representing a finite set of directed lines interconnecting the nodes. An example tree is illustrated in Figure 9.1.

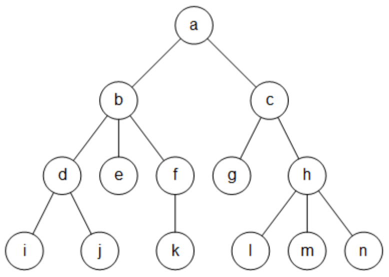


Figure 9.1: An Example of a Tree Data Structure

There are many terms with which we can describe a tree and the structure of that tree, as follows:

- ***Degree*** – The number of branches associated with a node;
- ***Indegree branch*** – A branch directed towards the node;
- ***Outdegree branch*** – A branch directed away from the node;
- ***Root*** – The first node in the tree;
- ***Parent*** – A node that has successor nodes (outdegree branches > 0);
- ***Child*** – A node that has a predecessor node (indegree branches == 1);
- ***Siblings*** – Two more nodes with the same parent;
- ***Path*** – A sequence of nodes in which each node is adjacent to the next one;
- ***Ancestor*** – any node in the path between the root and the current node;
- ***Descendent*** – any node in the paths from the current node to the leaf nodes;
- ***Leaf*** – any node that has no descendents;
- ***Internal node*** – any node with one or more descendents;
- ***Level*** – distance from the root;
- ***Height/depth*** – Number of levels + 1; and
- ***Sub-tree*** – any connected structure below the root.

These concepts are illustrated in Figure 9.2.

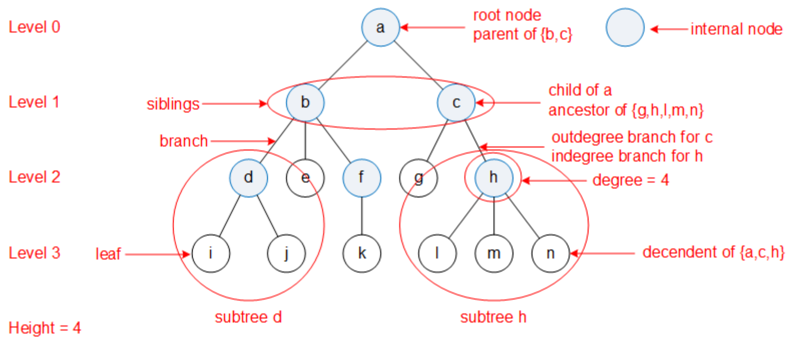


Figure 9.2: Illustration of Tree Concepts

9.2. Binary Trees

The binary tree is one of the most common types of trees whereby each node in the tree has between zero and two children, illustrated in Figure 9.3. The outdegree branches are named left and right for simplicity.

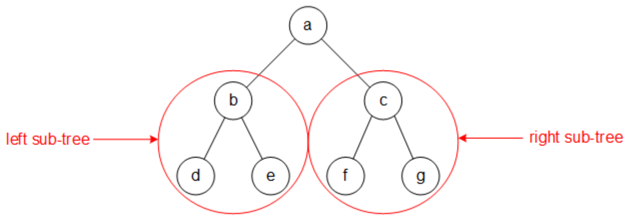


Figure 9.3: The Binary Tree

Given the above property of the binary tree, it is possible to calculate the range of the height of the tree (H) using the following formulas. The maximum height of a binary tree is straight-forward – it is possible that each node in the binary search tree could have only one child node, i.e., there is only one node at each level of the tree, in which case:

$$H_{\max} = n$$

Where represents the number of elements stored in the binary tree. The minimum height of the binary search tree can then be calculated as follows:

$$H_{\min} = \lfloor \log_2 n \rfloor + 1$$

For example, a binary tree storing 20 elements must have a height in the range 5-20.

We can also consider the structure of a binary tree, in particular whether or not the tree is considered ***balanced***, where the height of the sub-trees are either equal or one different, i.e., we calculate the balance (B) of a binary tree according to the formula:

$$B = H_{left} - H_{right}$$

The binary tree is then considered balanced where

$|B|$ □1

We can also define the a binary tree as ***complete*** if it contains the maximum number of nodes for the given height. Note that the binary tree previously illustrated in Figure 9.3 is considered complete as it contains the maximum number of nodes for a tree of height 3. Similarly, we can define a binary tree to be ***nearly complete*** if the tree is the minimum height for the given number of nodes, i.e., the height of the tree is equal to $H_{\min}$ defined earlier. Figure 9.4 illustrates a number of nearly complete binary trees.

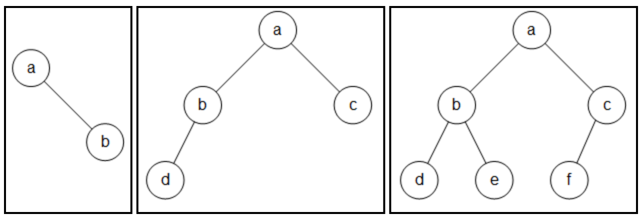


Figure 9.4: Examples of Nearly Complete Binary Trees

Traversal of a binary tree is divided into two fundamental approaches: breadth-first traversal and depth-first traversal. Breath-first traversal involves processing all nodes at a particular level of the tree before processing any nodes at the next level of the tree, as illustrated in Figure 9.5.

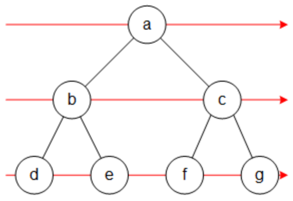


Figure 9.5: Breadth-First Traversal of a Binary Tree

Given the difficulty of navigating the tree to find all nodes for a particular level, a queue is used to simplify the task using the following algorithm:

```
ASSUMPTION: A variable named treeRoot contains a reference to the root node of the binary tree
ASSUMPTION: A method called processNode() performs any processing required for a node
queue.enqueue(treeRoot)
while NOT queue.empty()
        current ← queue.dequeue()
        queue.enqueue(current.left)
        queue.enqueue (current.right)
        processNode(current)
end-while
```

Algorithm 9.1: Breadth-First Traversal of a Binary Tree

Depth-first traversal however follows the branches of a node to process all descendent nodes before other nodes at the same level of the tree are considered. There are three types of depth-first traversals: pre-order, in-order, and post-order traversals, as illustrated in Figure 9.6a, b, and c, respectively. The illustration shows a simple three node binary tree with a red line that closely outlines the structure of the tree. Note the small filled circles where the red line makes contact with the tree, which provides a simple way of manually determining the order of nodes in a traversal. In particular, to determine the order of nodes accessed during pre-order traversal, we begin tracing the red line from above and left of the root node. As the line makes its way around the tree, if the left edge of a node can be reached, that node is the next node accessed during traversal. A similar process is used for in-order traversal (when the line touches the bottom of a node) and post-order traversal (when the line touches the right side node).

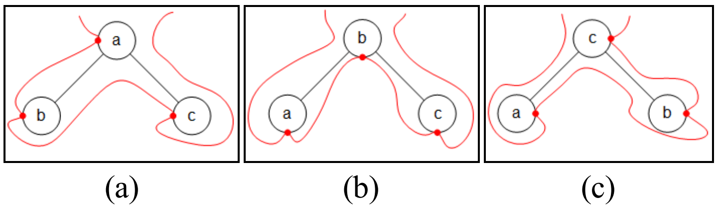


Figure 9.6: Depth-First Traversals of a Binary Tree

The actual algorithmic steps are illustrated in the following table:

pre_order(node)	***in_order(node)***	***post_order(node)***

processNode(current)
pre_order(left)
pre_order(right)
	in_order(left)
	processNode(current)
	in_order(right)
		post_order(left)
		post_order(right)
		processNode(current)

Notice how each of the algorithms are recursive and that the only difference between the three traversals is the location of the call to process the current node (***processNode(current)***).

We now consider three types of binary trees: the Binary Search Tree, the AVL tree, and the Heap.

9.2.1. The Binary Search Tree

The Binary Search Tree (BST) is a binary tree with the following rules applied:

- All descendent nodes in the left sub-tree have a value less than the value in the root node;
- All descendent nodes in the right sub-tree have a value greater than or equal to the root node; and
- Each sub-tree is also a binary search tree.

An example of a BST is shown in Figure 9.7, which also includes an indication of an in-order traversal of the binary search tree.

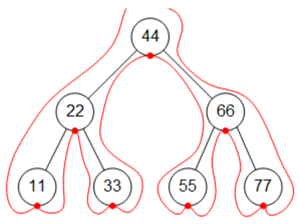


Figure 9.7: Example Binary Search Tree showing an In-Order Traversal

If you follow the order in which nodes are accessed using an in-order traversal, you will notice the following sequence of values:

11, 22, 33, 44, 55, 66, 77

This represents one of the useful properties of a BST, where an in-order traversal results in the data being accessed in sorted order. The most important feature of the BST however, is that the number of comparisons required to perform a search on a complete/nearly complete tree is equal to the minimum height of the tree ($H_{\min}$) and is $O(\log_2 n)$. Note that this is the same as for the Binary Search we examined in Section 8.1.2, however unlike the Binary Search which requires an ordered/sorted array, the BST can be progressively constructed in the same way as a linked list, as we will see.

Before we examine how to use and manipulate a BST, we must first introduce the node structure that is used. The BST node is remarkably similar to the structure used for a doubly linked list (Section 2.3), containing storage for a data element and two references to the same node structure (self-referential), left and right. For example, the following declaration could be used in C#:

```
private class BSTNode<TYPE>
{
    public TYPE data;
    public BSTNode<TYPE> left;
    public BSTNode<TYPE> right;
}
```

Note that the structure of the BST is actually recursive (every sub-tree is also a BST), thus the algorithms that we use could all be written using recursion. However to avoid the overheads of recursion most of the algorithms are written using iteration instead.

From the rules presented above for the BST, if follow a reference to the left we will always find a lower value, thus, we can determine the minimum value stored in the BST trivially:

```
ASSUMPTION: A variable named treeRoot contains a reference to the root node of the BST
ASSUMPTION: A variable named target contains the desired search target
current ← treeRoot
while NOT current.left = null
    current ← current.left
end-while
// current.data is the minimum value stored in the BST
```

Algorithm 9.2: BST Minimum

The algorithm to determine the maximum value stored in the BST is equally straightforward:

```
ASSUMPTION: A variable named treeRoot contains a reference to the root node of the BST
ASSUMPTION: A variable named target contains the desired search target
current ← treeRoot
while NOT current.right = null
    current ← current.right
end-while
// current.data is the minimum value stored in the BST
```

Algorithm 9.3: BST Maximum

To search for an item stored in the BST, we utilise the same rules, this time deciding whether to go left or right based on the value of the currently referenced node, as follows:

```
ASSUMPTION: A variable named treeRoot contains a reference to the root node of the BST
ASSUMPTION: A variable named target contains the desired search target
current ← treeRoot
while NOT current = null AND NOT current.data = target
    if target < current.data
        current ← current.left
    else
        current ← current.right
    end-if
end-while
// if current = null then element not found, otherwise target is found at current
```

Algorithm 9.4: BST Traversal to Search for a Target Value

Inserting a new value into a BST is a very similar algorithm. In particular, we search through the tree to find the correct location for the new value, however instead of terminating our search we attach the new node, as follows:

```
ASSUMPTION: A variable named treeRoot contains a reference to the root node of the BST
ASSUMPTION: A variable newNode is initialised with the new value and left/right set to null
if treeRoot = null
    treeRoot ← newNode
else
    current ← treeRoot
    while true
        if newNode.data < current.data
            if current.left = null
                current.left ← newNode
                break
            else
                current ← current.left
            end-if
        else
            if current.right = null
                current.right ← newNode
                break;
            else
                current ← current.right
            end-if
        end-if
    end-while
end-if
```

Algorithm 9.5: BST Insertion (1)

Alternatively, we could introduce a reference to the previous node, as we did for singly linked list insertion, i.e.,

```
ASSUMPTION: A variable named treeRoot contains a reference to the root node of the BST
ASSUMPTION: A variable newNode is initialised with the new value and left/right set to null
if treeRoot = null
    treeRoot ← newNode
else
    previous ← null
    current ← treeRoot
    while NOT current = null
        previous ← current
        if newNode.data < current.data
            current ← current.left
        else
            current ← current.right
        end-if
    end-while
    if newNode.data < previous.data
        previous.left ← newNode
    else
        previous.right ← newNode
    end-if
end-if
```

Algorithm 9.6: BST Insertion (2)

Deletion of a value from a BST is more complicated however. In particular we must consider three possibilities:

- Deletion of a leaf node – this is straightforward, we just set the reference to the leaf to null to disconnect the node from the BST;
- Deletion of a node with a single outdegree branch/one child node – this is also relatively straightforward, the reference to the deleted node can just be replaced with a reference to the child of the deleted node;
- Deletion of a node with two outdegree branches/two child nodes – this is much more complicated, each sub-tree of the deleted node could be many levels deep, and it isn't possible to replace two references with one.

The solution to the problem however is reasonably trivial, at least conceptually. In particular, if you consider carefully the ordering of values in a BST, we can see the following:

- The value that immediately precedes the value in the node to be deleted can be found is the maximum of the left sub-tree; and
- The value that immediately follows the value in the node to be deleted can be found is the minimum of the right sub-tree.

Given this information, rather than delete a node with two outdegree branches, we instead replace it with the value immediately before/after and delete that node instead. These concepts are illustrated in Figure 9.8a (using maximum of the left sub-tree) and Figure 9.8b (using minimum of the right sub-tree).

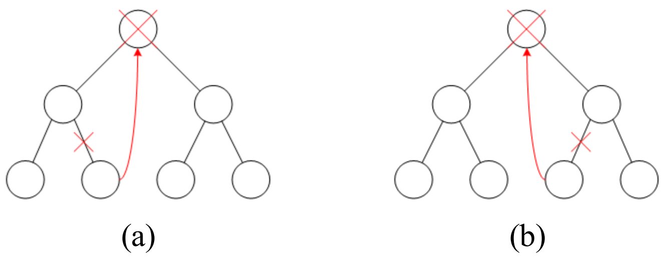


Figure 9.8: BST Deletion of a Node with Two Outdegree Branches

The algorithm for deleting a node from a BST is presented below using recursion for simplicity. It is also possible to write an equivalent iterative algorithm, however this would be longer and less clear

```
ASSUMPTION: A variable named treeRoot contains a reference to the root node of the BST
ASSUMPTION: The parameter target contains the value to be deleted from the BST
RETURNS: true if the value was successfully deleted, false otherwise
Delete(target)
begin
    Delete ← DeleteWorker(treeRoot, target)
end

ASSUMPTION: The parameter node is passed by-reference to allow its value to be modified
DeleteWorker(node, target)
begin
    if node = null
        result ← false
    else
        if target < node.data
            result ← DeleteWorker(node.left, target)
        else if target = node.data
            result ← true
            if node.left = null
                node ← node.right
            else if node.right = null
                node ← node.left
            else
                // Find maximum value of left sub-tree
                maxLeft ← node.left
                while NOT maxLeft.right = null
                    maxLeft ← maxLeft.right
                end-while
                // Move its data to node to be deleted
                node.data ← maxLeft.data
                // Remove node from left sub-tree containing maximum value
                DeleteWorker(node.left, maxLeft.data)
            end-if
        else
            result ← DeleteWorker(node.right, target)
        end-if
    end-if
    DeleteWorker ← result
end
```

Algorithm 9.7: BST Deletion

9.2.2. The AVL Tree

The problem with the Binary Search Tree is that its performance is strongly linked to the actual structure of the tree. As noted above in Section 9.3.1, the performance of a search for a complete or nearly-complete tree will be . However, consider constructing a BST from the input data {a, b, c, d, e}, which would result in the BST illustrated in Figure 9.9.

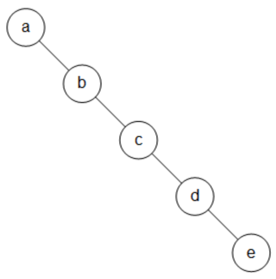


Figure 9.9: Binary Search Tree Suffering Imbalance

The AVL tree, named after its developers G. M. Adelson-Velskii and E. M. Landis, is actually a BST where the balance of the tree is maintained throughout insertion and deletion operations. The same input data would result in the AVL tree illustrated in Figure 9.10, a nearly-complete BST.

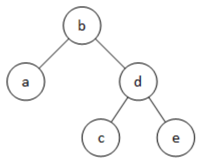


Figure 9.10: AVL Tree Correcting BST Imbalance

During construction, the AVL tree algorithms consider the balance (B) of the sub-trees of each node, defined as follows:

$$B = \left| H_{left} - H_{right} \right|$$

The rules of the AVL tree then supplement those for the Binary Search Tree with:

- The balance of the tree B □ 1
- Each sub-tree is also an AVL tree

The tree balance is then maintained during insertion and deletion operations through the application of rotations to four different problems. To understand these rotations, we must first define the following:

- A tree/sub-tree is ***left-high*** when
- A tree/sub-tree is ***right-high*** when

The four problems rotations are applied to are now presented in the form "s-of-p" where s represents the balance of the sub-tree and p represents the balance of the parent of the sub-tree, as follows:

1. ***Left-of-left*** – a left-high sub-tree on a left-high tree, solved with a single clockwise rotation;
2. ***Right-of-right*** – a right-high sub-tree on a right-high tree, solved with a single anti-clockwise rotation;
3. ***Right-of-left*** – a right-high sub-tree on a left-high tree, solved with an anti-clockwise rotation on the sub-tree followed by a clockwise rotation on the parent tree; and

4. ***Left-of-right*** – a left-high sub-tree on a right-high tree, solved with a clockwise rotation on the sub-tree followed by an anti-clockwise rotation on the parent tree.

The four tree imbalances, and their respective rotations, are illustrated in Figure 9.11.

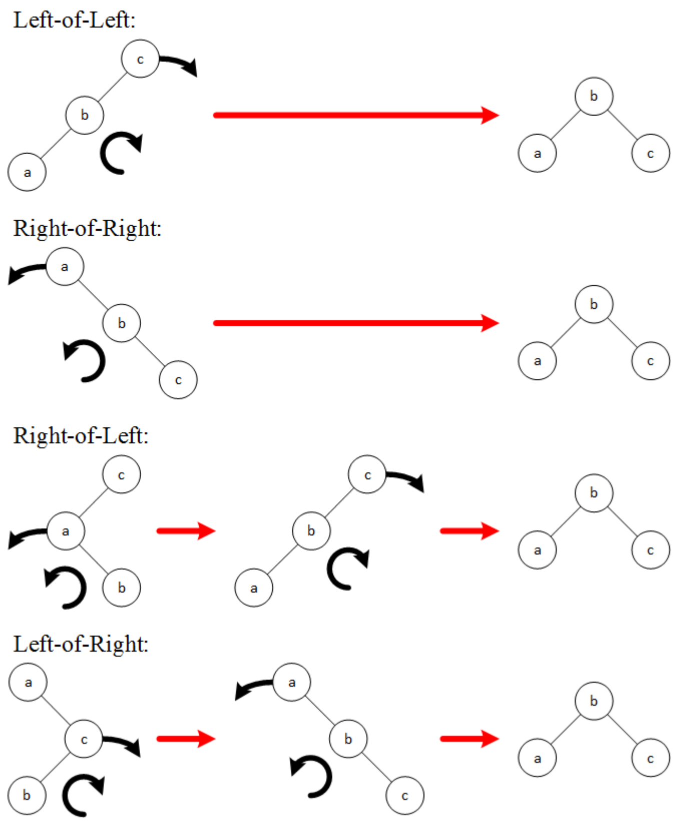


Figure 9.11: Tree Imbalance and Rotations in AVL Trees

9.2.3. The Heap

The Heap is also a binary tree, although different to the BST and AVL tree, with the following rules applied:

- All descendent nodes have a value that is less than or equal to the root node; and
- Each sub-tree is also a heap.

The heap also has the property that it is always either complete or nearly complete, an example of which is illustrated in Figure 9.12.

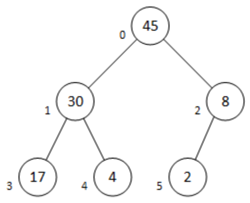


Figure 9.12: An Example of a Heap

Also note the numbers shown in the figure which represent the array locations for the nodes stored in the heap. It is possible to derive the following formulas to determine the location of related nodes, both parent nodes (P) and child nodes (C), where represents the index of a node being referenced:

$$P(x) = \left\lfloor \frac{x - 1}{2} \right\rfloor$$

$$C_{left}(x) = (x \times 2) + 1$$

$$C_{right}(x) = (x \times 2) + 2$$

To allocate an array to store a heap, it is important to first calculate the correct size for the array. For example, if a request is received to create a heap of size 10, an array should be allocated to store 15 elements, due to the structure of a tree:

- Level 0 – 1 node (root node)
- Level 1 – 2 nodes
- Level 3 – 4 nodes
- Level 4 – 8 nodes
- Total: 1 + 2 + 4 + 8 = 15

This can be calculated using the following formula:

$$size = 2^{\lceil \log_2 n \rceil} - 1$$

Insertion of a new value into a heap involves storing the new data value in the next free location in the array, then rearranging the data stored in the array to reestablish the heap. This is divided into two methods for clarity, as follows:

```
ASSUMPTION: A variable named heap refers to the array that stores the heap
ASSUMPTION: A variable named lastUsed stores the index of the last element in the heap
RETURNS: true if the value was successfully added to the heap, false otherwise
Insert(data)
begin
    if lastUsed = heap.length – 1
        Insert ← false
    else
        lastUsed ← lastUsed + 1
        heap[lastUsed] ← data
        ReheapUp(lastUsed)
        Insert ← true
    end-if
end

ReheapUp(node)
begin
    if NOT node = 0
        parent ← (node - 1) / 2 // integer division
        if heap[node] > heap[parent]
            Swap(heap[parent], heap[node])
            ReheapUp(parent)
        end-if
    end-if
end
```

Algorithm 9.8: Heap Insertion

Deleting a node from a heap is slightly different – the root node is always the node that is deleted/removed from the heap. This is achieved by replacing the data in the root node with the last node stored in the array, then rearranging the data stored in the array to reestablish the heap. This is divided into two methods for clarity, as follows:

```
ASSUMPTION: A variable named heap refers to the array that stores the heap
ASSUMPTION: A variable named lastUsed stores the index of the last element in the heap
ASSUMPTION: The parameter data is an output parameter
RETURNS: true if a value was successfully deleted from the heap, false otherwise
Delete(data)
begin
    if lastUsed = –1
        Delete ← false
    else
        data ← array[0]
        array[0] ← array[lastUsed]
        lastUsed ← lastUsed - 1
        ReheapDown(0)
        Delete ← true
    end-if
end

ReheapDown(node)
begin
    leftChild ← node * 2 + 1
    rightChild ← node * 2 + 2
    if leftChild <= lastUsed
        largest ← leftChild
        if rightChild <= lastUsed AND array[largest] < array[rightChild]
            largest ← rightChild
        end-if
        if array[node] < array[largest]
            Swap(array[node], array[largest])
            ReheapDown(largest)
        end-if
    end-if
end
```

There are two interesting points to note about the algorithm used to delete a node from a heap[1]:

1. By always removing the element with the highest value, the heap represents a good structure for implementing a simple priority queue (operations on a heap are $O(\log_2 n)$);
2. The value from the last used element in the array replaces the root node each time. However if instead the value of the root and last used nodes are swapped, the deletion algorithm transforms into another sorting algorithm known as the ***heap sort***. The heap sort is $O(n \log n)$, the same as for quick sort (Section 7.2.4), however the quick sort is in fact faster.

Task 9.1. Binary Search Tree

Objective: The Binary Search Tree implements many of the concepts that are important to understanding trees in an example that is easy to understand and, when balanced, performs

[1] The keen observer may have also noticed that the above algorithms may be improved by eliminating the repetitive use of the Swap method and instead using a temporary hold like the insertion sort does. Swap has been used to improve clarity in these examples.

well. In this task we examine the implementation of the Set data structure using a Binary Search Tree implementation.

Two fundamental data structures that we haven't considered in this unit are sets and bags. These data structures are simple, where a set represents a collection of unique values and a bag represents a collection of non-unique values. These two data structures are most commonly implemented using a Binary Search Tree, which you are required to undertake for this task.

Your tasks are as follows:

1. Create a new class `Set<TYPE>` that implements the interface `ICollection<TYPE>` according to the following notes:
   - The class should exist in the `SIT221_Library` namespace;
   - All members of `ICollection<TYPE>` are to be implemented except for the `GetEnumerator()` methods (these should throw a `NotImplementedException`);
   - Note that as a set, duplicate values are not permitted. You will need to consider this when implementing the `Add` method, which should throw an `InvalidOperationException` when this occurs;
   - Use an in-order traversal to implement the `CopyTo()` method. Note that given that the in-order traversal is recursive, you will need to consider carefully how to track which location in the array to copy to.
2. Add three additional methods to your `Set<TYPE>` class as follows:
   - `int Height()`
     Calculates the current height of the tree (calculate the heights of each sub-tree, select the largest value, and add one);
   - `int MinHeight()`
     Calculates the minimum height that the tree could be/could have been according to the information provided in this workbook; and
   - `int MaxHeight()`
     Returns the number of elements stored in the tree (which equates to the worst possible height for the tree).
3. Write a Main method that tests your `Set<TYPE>` class by creating a set of integers and allowing the user to add or remove as they wish. Notes:
   - The program must display relevant error messages if a duplicate is stored, etc., not crash;
   - Use the `CopyTo()` method of your set class to retrieve the values stored in the tree and display them on the screen after each step;
   - Also display the current, minimum, and maximum heights after each step.

10

10. Graphs

Previously in this unit we have introduced two data structures that consist of nodes interconnected by links. In particular, we examined linked lists in Session 2 and trees in Session 9. These two data structures, and many others that are outside the scope of this unit, can be generalised by the concept of the graph, which we examine in this session. We begin by introducing the general concepts and terminology of graphs (Section 10.1), before examining how they are implemented (Section 10.2) and an example in the Shortest Path algorithm (Section 10.4). The Shortest Path algorithm is one of the most common graph algorithms used in modern computing, including applications for computer networks (Open Shortest Path First routing algorithm) and GPS mapping systems (turn-by-turn navigation).

Objectives

In this session you will learn

- The concepts of a graph and how they relates to other data structures;
- How graphs are represented using an adjacency matrix;
- How graphs are represented using an adjacency list; and
- How the shortest path algorithm works.

10.1. Graph Concepts

Graphs, similar to linked lists and trees, consist of a number of ***vertices*** (nodes) that are interconnected by ***edges*** (links). Fundamentally there are two types of graphs: ***undirected graphs*** (Figure 10.1a) and ***directed graphs*** (Figure 10.1b), also known as ***digraphs***. As suggested by their names, the key difference is that directed graphs impose a direction on the edges and only permit navigation between nodes in one direction, as opposed to undirected graphs which permit navigation in both directions. The ability to navigate from one vertex, a, to a second vertex, b, can be indicated simply for a graph, as follows:

(a, b)

In the case of an undirected graph, there must also be a matching pair in the opposite direction, i.e.,

(b, a)

It is also important to note that the edges in a graph may or may not be weighted. A graph which has weightings applied to its edges is referred to as a ***network***, or more simplistically, a ***weighted graph***.

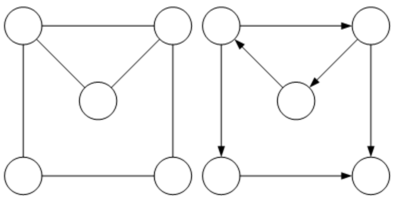


Figure 10.1: Types of Graphs

We can use additional terminology to describe the connection of vertices in a graph by edges. Any two vertices that are connected (in one or both directions) are ***adjacent*** (Figure 10.2a and Figure 10.2b). The edge is then said to be ***incident*** upon those two vertices. An edge can also be incident upon a single vertex, in which case a ***loop*** is formed (Figure 10.2b). It is also possible that two or more vertices may be incident upon the same pair of vertices, in which case those edges are said to be ***parallel*** (Figure 10.2c). A ***simple graph*** is one which contains no loops or parallel edges.

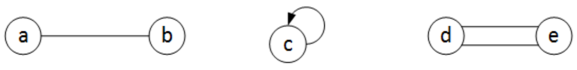


Figure 10.2: Interconnection of Vertices in a Graph

We can also describe sequences of vertices connected by edges in a graph. Within a graph, a ***path*** represents a sequence of vertices and edges connecting two non-adjacent vertices (Figure 10.3a), which are referred to as ***connected*** vertices. If the vertices forming a path are unique, i.e., each vertex is visited only once, this is known as a ***simple path*** (also Figure 10.3a). If a path begins and ends at the same vertex, a ***cycle*** is formed (Figure 10.3b).

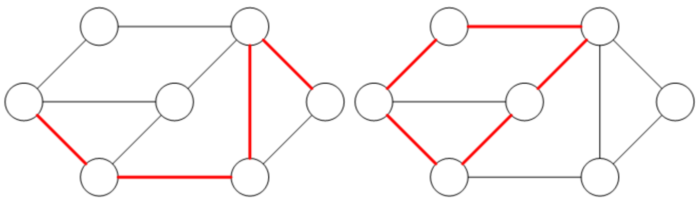
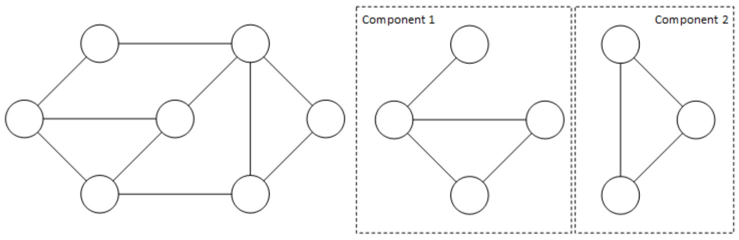


Figure 10.3: Paths in a Graph

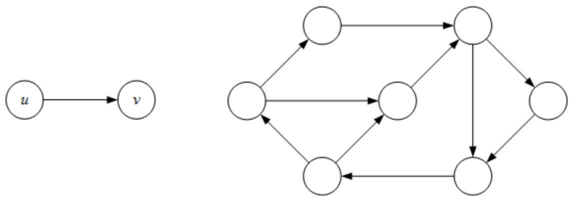
We can also describe the overall interconnection of a graph, whereby a ***connected*** graph is where a path exists between any two vertices (Figure 10.4a). Where are graph is not connected, a maximal subset of connected vertices in that graph is referred to as a ***component*** of that graph (Figure 10.4b).



(a) (b)

Figure 10.4: Interconnection of Graphs

Finally for digraphs, where a path exists from vertex *u* to vertex *v*, it is said that *u* is ***adjacent tov***, and *v* is ***adjacent from*** *u* (Figure 10.5a). Similarly, if a path exists between any two vertices in the graph, it is said to be ***strongly connected*** (Figure 10.5b).



(a) (c)

Figure 10.5: Interconnection of Directed Graphs

10.2. Programming Graphs

There are primarily two approaches to working with graphs in code: the ***adjacency matrix*** and the ***adjacency list***. Put simplistically, the difference between these approaches is the difference between programming using arrays (used by the adjacency matrix) and using linked lists (used by the adjacency list).

The Adjacency Matrix

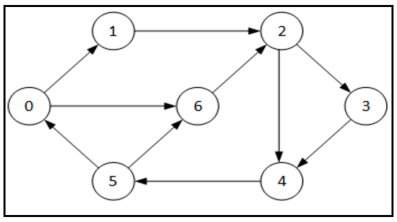
The adjacency matrix is implemented using a two dimensional array (representing the matrix). Both the rows and columns of the array represent the vertices appearing in the graph. Moreover, each vertex in the graph is represented by one row and one column. For a non-weighted graph, the values stored in array then represent whether an edge is present (value = 1) or not (value = 0), i.e.,

- If ***matrix[i, j] == 1***, there exists an edge (v_i, v_j); similarly
- If ***matrix[i, j] == 0***, there is no edge (v_i, v_j).

For weighted graphs, the weight assigned to that particular edge is assigned instead of a value of 1. Importantly, the above examples may suggest that an adjacency matrix is only suitable for digraphs, however this is not the case. Instead for an undirected graph, entries are made in the matrix for both directions i.e.,

$(v_i, v_j) = (v_i, v_j) = 1$

An example adjacency matrix for a non-weighted digraph is shown in Figure 10.6. Notice how in the first row (row 0) that there are edges indicated for vertices 1 and 6. In examining the graph, you can see that the vertex labelled 0 is connected to vertices 1, 5, and 6, however it is not possible to navigate to vertex 5 from vertex 0 (only the reverse, which can also be found in the table under row 5).



$$\begin{array}{c|ccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 5 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 6 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Figure 10.6: Graph Representation using an Adjacency Matrix

The Adjacency List

The alternative to an adjacency matrix, an adjacency list, uses a collection of linked lists, one linked list per vertex in the graph. The nodes in each linked list represent an edge that can be navigated from that vertex, indicating the vertex that the edge connects to and any weight (for a weighted graph). The adjacency list for the same non-weighted digraph as before is shown in Figure 10.7. Again, notice the connections from vertex 0 to vertices 1 and 6, however not vertex 5 (which appears in the list for vertex 5 instead).

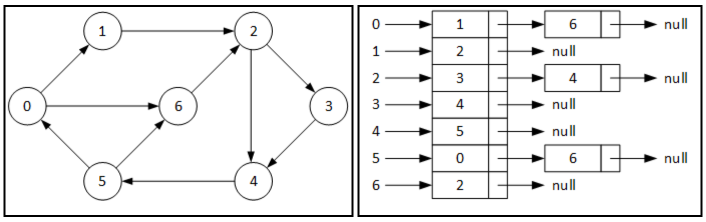


Figure 10.7: Graph Representation using an Adjacency List

Graph Traversal

The task of traversing a graph is similar to what we have seen for trees. However graph traversal suffers a major complication in the case of graphs that are not connected, i.e., the graph may be divided into two or more components. Thus, we require a mechanism to ensure that each vertex is included in the traversal, as we will see in Section 10.3.

For breadth-first traversal:

- For each vertex in the graph not yet visited:
  - Process the current vertex; and
  - Apply the same algorithm to each directly connected vertex before proceeding to the next adjacent vertices.

For depth-first traversal:

- For each vertex in the graph that has not yet been visited:
  - Process the current vertex; and
  - Apply the same algorithm recursively to each adjacent vertex that has not been visited.

10.3. Adjacency List Example

In this section we present an example implementation of a graph using an adjacency list. For this example, we will implement the graph used previously in Figure 10.6 and Figure 10.7. We first examine the (nested) class that will be used to represent an individual Vertex, as follows:

```
public class Vertex
{
    internal EdgeNode head;
    internal EdgeNode tail;
    internal bool visited;

    private TYPE _Data;
    public TYPE Data
    {
        get { return _Data; }
    }

    internal Vertex(TYPE data)
    {
        head = tail = null;
        _Data = data;
    }

    public void Connect(Vertex target)
    {
        EdgeNode node = new EdgeNode(target);

        if (head == null)
            head = tail = node;
        else
        {
            tail.next = node;
            tail = node;
        }
    }
}
```

The main features of this class to note are:

- The use of `head` and `tail` references (for storing the adjacency list);
- The use of a boolean variable (`visited`) which will be used during traversals to keep track of which vertices have been included in the traversal;
- The provision of a variable for storing custom data (`_Data`) and associated read-only property (`Data`);
- The provision of a method to record an adjacency between two vertices (`Connect`),

which creates a new linked list node and adds it to the tail of the list; and
- A constructor to simplify creation and initialisation of a `Vertex` object.

The `EdgeNode` (nested) class, which is used as the linked list node structure for the adjacency list, is straightforward given our previous coverage of linked lists is as follows:

```
internal class EdgeNode
{
    public Vertex vertex;
    public EdgeNode next;
    public EdgeNode(Vertex v)
    {
        vertex = v;
        next = null;
    }
}
```

Given the above structures, we can now consider how to implement the graph traversals as discussed above in Section 10.2[1]. The code for the breadth-first traversal is as follows:

```
public void BreadthFirstTraversal(out List<Vertex> result)
{
    result = new List<Vertex>();
    Queue<Vertex> bftQueue = new Queue<Vertex>();

    foreach (Vertex v in _Items)
        v.visited = false;

    // Breadth first traversal
    for (int i = 0; i < _Items.Count; i++)
    {
        if (_Items[i].visited == false)
        {
            bftQueue.Enqueue(_Items[i]);
            while (bftQueue.Count > 0)
            {
                Vertex v = bftQueue.Dequeue();
                if (v.visited == false)
                {
                    result.Add(v); // process node
                    v.visited = true;
                    for (EdgeNode n = v.head; n != null; n =
ptr.next)
                        bftQueue.Enqueue(n.vertex);
                }
            }
```

[1] In this section we only examine the code for implementing graph traversals. The complete code for the graph class can be found in the provided code ZIP file for this session in CloudDeakin. Also note that some variable names have been shortened to improve presentation.

```
            }
        }
    }
```

Note how a queue is used, just as for breadth-first search in binary trees (Section 9.2). The first step in the above algorithm however is to reset the `visited` flag in all vertices. After this, a loop is used to check that each of the vertices is considered in the traversal (to handle a graph that isn't fully connected). Upon considering the first vertex, it is placed in the queue. A loop is then used to traverse any vertices in the queue (only one to begin with). The first step for which is to process the node (simultaneously marking the vertex as having been visited), then to enqueue each of the vertices that it is adjacent to. The processing in this example only adds the nodes to a list in the order of traversal (an output parameter).

Depth-first traversal requires the addition of a second method due to the use of recursion. The main traversal method is as follows:

```
    public void DepthFirstTraversal(out List<Vertex> result)
    {
        result = new List<Vertex>();

        foreach (Vertex v in _Items)
            v.visited = false;

        for (int i = 0; i < _Items.Count; i++)
        {
            if (_Items[i].visited == false)
                DepthFirstWorker(result, _Items[i]);
        }
    }
```

Note how this method similarly begins by resetting the `visited` flag and then has a loop that ensures that all vertices have been visited up on completion (to handle a graph that isn't fully connected). However there is no processing of vertices in this method, which is performed by the `DepthFirstWorker` method, as follows:

```
    private void DepthFirstWorker(List<Vertex> result, Vertex vertex)
    {
        result.Add(vertex); // process node
        vertex.visited = true;
        for (EdgeNode ptr = vertex.head; ptr != null; ptr = ptr.next)
        {
            if (ptr.vertex.visited == false)
            {
                DepthFirstWorker(result, ptr.vertex);
            }
        }
    }
```

The recursive worker method begins by processing the node before recursively calling itself for each adjacent vertex that hasn't been included in the traversal yet.

The `Main` method for our example program is as follows:

```
static void Main(string[] args)
{
    ALGraph<int> graph = new ALGraph<int>();
    ALGraph<int>.Vertex [] vertices = new ALGraph<int>.Vertex[7];

    for (int i = 0; i < 7; i++)
        vertices[i] = graph.Add(i);

    vertices[0].Connect(vertices[1]);
    vertices[0].Connect(vertices[6]);
    vertices[1].Connect(vertices[2]);
    vertices[2].Connect(vertices[3]);
    vertices[2].Connect(vertices[4]);
    vertices[3].Connect(vertices[4]);
    vertices[4].Connect(vertices[5]);
    vertices[5].Connect(vertices[0]);
    vertices[5].Connect(vertices[6]);
    vertices[6].Connect(vertices[2]);

    // Breadth first traversal
    List<ALGraph<int>.Vertex> result;
    graph.BreadthFirstTraversal(out result);
    Console.Write("Breadth first: ");
    foreach (ALGraph<int>.Vertex v in result)
        Console.Write("{0} ", v.Data);
    Console.WriteLine();

    // Depth first traversal
    graph.DepthFirstTraversal(out result);
    Console.Write("Depth first: ");
    foreach (ALGraph<int>.Vertex v in result)
        Console.Write("{0} ", v.Data);
    Console.WriteLine();
}
```

The `Main` method is relatively straightforward, creating the graph and vertices within the graph, then establishing the connections between the vertices, before finally displaying the output of the traversals. The resulting output on the console is as follows:

```
Breadth first: 0 1 6 2 3 4 5
Depth first: 0 1 2 3 4 5 6
```

Although the output of the breadth first traversal is of no use in this particular example, note that the depth first traversal returns the vertices in order. Both traversals begin at the first vector (0), however the depth-first search recursively considers the first connected vertex. If you review the Main method's code, you'll notice that the first vertex added for vertex 0 is vertex 1, the first added for vertex 1 is vertex 2, and so on.

10.4. Example: Shortest Path

The above example helps illustrate the behaviour of graphs however a more useful example is to consider the Shortest Path algorithm. The Shortest Path algorithm is used somewhat regularly in solving problems, including calculating a path for a journey in a GPS (vertices represent intersections and edges represent roads), or for calculating the most efficient routes through a computer network, e.g., the Open Shortest Path First algorithm (vertices represent routers and edges represent the network links between them).

The general steps of the Shortest Path algorithm are as follows:

- Initialise the array ***smallestWeight*** so that ***smallestWeight[u] = weights[vertex, u]***.
- Set ***smallestWeight[vertex] = 0***.
- Find the vertex, ***v***, that is closest to vertex for which the shortest path has not been determined.
- Mark ***v*** as the (next) vertex for which the smallest weight is found.
- For each vertex ***w*** in ***G***, such that the shortest path from ***vertex*** to ***w*** has not been determined and an edge (v, w) exists, if the weight of the path to ***w*** via ***v*** is smaller than its current weight, update the weight of ***w*** to the weight of ***v*** + the weight of the edge (v, w).

The last three steps are then repeated ***n – 1*** times to process the remaining vertices.

Although this algorithm sounds complicated, a less mathematically oriented description is simpler[2]:

- Record the distance from the starting vertex to itself as zero, with a null/empty path, and mark it as complete.
- Assume that the distance/cost to reach the remaining vertices is infinite.
- Set the current vertex to the starting vertex
- Repeat until finished
  - For each vertex adjacent from current that is not yet complete
    - If the total cost of the path to reach the adjacent vertex through the current vertex is less than the best-known distance/cost, update the best-known cost and path for that adjacent vertex
  - From the list of all vertices not marked complete, set current to the vertex with the smallest known cost and mark it as complete.

Example code for shortest path has been included in the provided code for this week and is the subject of Task 9.1, below.

Task 10.1. Shortest Path

Objective: Graphs represent one of the most complicated data structures (and associated algorithms) that we consider. Although you are not expected to be able to write code for graphs, you should be able to adapt the provided code to solve a new problem.

For this week's task we will look at determining the shortest travel distances from Melbourne to the other capital cities on the Australian mainland (Adelaide, Brisbane, Canberra, Darwin,

[2] If this is still unclear, review the lecture slides/recording for this week which includes an example demonstrated visually.

Perth, and Sydney). However in completing this task, we will assume that our journey/s will always begin in Melbourne and that it is only possible to travel from Melbourne to either Adelaide or Canberra (and back again). The other capital cities must then be reached through either of those two capitals. Your tasks are as follows:

1. Using Google Maps (http://maps.google.com.au), determine the distances by road between each of the capital cities on the mainland. This can be done simply by entering a query such as "Melbourne to Canberra" and the distance (by road) will be shown.
2. Load the information you have retrieved into the Shortest Path code provided along with this week’s workbook, ensuring that city names are displayed instead of numbers.

11

11. Algorithms Revisited

In this chapter we will undertake a review of the concepts in algorithm development that we have learned throughout our study of programming. This unit represents the final subject in a sequence that focuses on the fundamental programming skills. In particular, in completing this unit you have undertaken a study of:

- SIT105 Critical Thinking and Problem Solving– learning how to analyse a problem and general techniques for solving such problems, including the fundamental algorithm concepts;
- SIT102 Introduction to Programming – learning the role and application of programming language concepts in solving relatively simple programming problems using a programming language;
- SIT232 Object-Oriented Development – learning how to construct software to solve larger programming problems through the application of the concepts of the object oriented methodology, effectively breaking down a large problem into more manageable modules; and
- SIT221 Classes, Libraries and Algorithms – learning different approaches to storing, querying and manipulating data, resulting in much improved performance in the software you develop.

We began by learning the fundamentals of constructing algorithms by learning the structure theorem (Section 11.1). Over time we learned to construct more complex algorithms, of which there are many different types (Section 11.2). We finish our study of this unit with an examination of the Language Integrated Query (LINQ), a relatively new programming feature provided by the C# programming language which provides us with new options for developing algorithms (Section 11.3).

Objectives

In this session you will learn

- The different types of algorithms;
- The basic concepts of LINQ; and

11.1. The Structure Theorem

When developing an algorithm for a computer, we rely upon the fundamental principles defined by the structure theorem[1], which tells us that any algorithm can be constructed using a combination of only three elements:

1. Sequence – there is an ordering imposed upon the steps of the algorithm;
2. Selection – making decisions on the basis of some condition, and then choose to perform/not to perform one or more of the steps of the algorithm; and
3. Repetition – performing one or more steps of the algorithm a number of times until some condition is achieved/no longer maintained.

If you consider the other features of a programming language, introduced in Section 3.2, you will notice that they have very little impact on algorithms and generally fall into:

- Syntactical issues, e.g., statement format, specification of literals, variable declarations, and so on; and
- Structural issues, e.g., definition of methods, properties, classes, interfaces, namespaces, and so on.

The introduction of structure to our data does not change how we develop algorithms, however it does allow different algorithms to be developed.

11.2. Types of Algorithms

In this section we examine the different types of algorithms that we have seen in this unit. In particular, we consider brute force algorithms (Section 11.2.1), algorithms that exploit space and time trade offs (Section 11.2.2), divide and conquer (Section 11.2.3), decrease and conquer (Section 11.2.4), transform and conquer (Section 11.2.5), and greedy algorithms (Section 11.2.6).

11.2.1. Brute Force Algorithms

The brute force algorithm represents the simplest approach, and often easiest approach, to developing algorithms[2]:

> *"Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved."*

[1] Structure theorem is widely acknowledged to have first been introduced in Böhm, C., and Jacopini, G., "Flow diagrams, turing machines and languages with only two formation rules", in Communications of the ACM, Volume 9(5), May 1966, pp366-371.
[2] Levitin, A., Introduction to The Design and Analysis of Algorithms, Second Edition, Addison Wesley, 2007.

Brute force algorithms are applicable to the widest range of problems, typically solving a problem in the simplest or most direct/obvious way. The down side to brute force algorithms are that they are rarely efficient and there are regularly better algorithms available for the same task. However, brute force algorithms are also usually very simple to implement and the cost of using such an algorithm may be insignificant. Moreover, for small data sets a brute force algorithm may even be more efficient.

We have considered the following brute force algorithms in this unit:

- $O(n^2)$ – Sequential search;
- $O(n^2)$ – Bubble sort; and
- $O(n^2)$ – Selection sort.

11.2.2. Space and Time Tradeoffs

In undertaking the task of software development, for all but the most simplest of applications, you will regularly come across one or more decision points where you can choose to either use more memory (space) or more computation (time). The most obvious example of spaces and time tradeoffs can be found in performing a complex calculation (time) or instead using a lookup table (space).

We have considered the following approaches involving space/time tradeoffs:

- Hashing.

11.2.3. Divide and Conquer Algorithms

Many of the most efficient algorithms that we work with follow the divide and conquer approach. Simply put, divide and conquer works by dividing the problem into several smaller problems, then solving those smaller problems separately. Sometimes, the solutions to those smaller problems need to be processed in some way to combine them into the final solution.

We have considered the following divide and conquer algorithms in this unit:

- $O(\log_2 n)$ – Binary search;
- $O(n\log n)$ – Quick sort; and
- $O(n)$ – Binary tree depth-first traversals.

11.2.4. Decrease and Conquer Algorithms

Decrease and conquer algorithms exploit the relationship between a problem and a smaller instance of the same problem. These relationships can either be exploited using a top down approach (involving recursion) or bottom-up (no recursion). There are three variations, decreasing by a constant (e.g., decrease by one each iteration), by a constant factor (decrease by half), or by a variable amount.

We have considered the following decrease and conquer algorithms in this unit:

- $O(n^2)$ – Insertion sort;
- $O(\log_2 n)$ – Binary tree search;
- $O(\log_2 n)$ – Binary tree insertion;
- $O(n)$ – Graph breadth-first traversals; and
- $O(n)$ – Graph depth-first traversals.

11.2.5. Transform and Conquer Algorithms

Transform and conquer algorithms work by changing the representation of a problem before attempting to solve the problem. There are three types of possible transformations: instance simplification (a simpler version of the same problem), representation change (different representation of the same problem), or problem reduction (transform to a different problem we can solve).

We have considered the following reduction algorithms in this unit:

- AVL trees;
- $O(n\log n)$ – Heap sort;

11.2.6. Greedy Algorithms

Greedy algorithms are useful for optimization problems, however they must be applied carefully as it is possible that a greedy algorithm may not result in a correct solution. Greedy algorithms are constructed whereby a choice is made during each step of the algorithm with the following constraints:

1. The choice must be feasible, i.e., it must satisfy the requirements of the problem;
2. The choice must be locally optimal, i.e., given the choices available it must be the best choice to make at that time; and
3. The choice must be irrevocable, i.e., the choice cannot be changed after it has been made.

We have considered the following greedy algorithms in this unit:

- Shortest path.

11.3. Language Integrated Query

With the release of Visual Studio 2008, Microsoft introduced new functionality that can be used for solving a number of programming problems related to querying data through the introduction of Language Integrated Query, most commonly referred to by its shortened name LINQ. LINQ does not represent a single technology, but a collection of technologies that have been added to the C# programming language (and other languages). LINQ can also be used for building functionality applies to a number of functional areas:

- LINQ to Objects – the ability to apply queries to objects in memory that implement the `IEnumerable` or `IEnumerable<T>` interfaces (see Section 4.5);
- LINQ to XML – the ability to apply queries to XML data; and

- LINQ to ADO.NET – the ability to apply queries to database objects, consisting of three sub-elements: LINQ to DataSet, LINQ to SQL, and LINQ to Entities.

In this section we only consider LINQ to Objects, however the concepts we will introduce here are easily adapted to XML and/or ADO.NET.

To use LINQ in your programs, it is necessary to have the following using statement at the top of your programs (added by Visual Studio by default):

```
using System.Linq;
```

To begin to understand LINQ, consider the example program shown in Figure 11.1. This simple example begins by creating a Vector containing (random numbers) in the range 0-10, displays those numbers, then creates a new collection containing those numbers from the original Vector that have a value of less than 5 and displays those numbers. Although this is a very simple example that doesn't show the true power of LINQ, those who are familiar with SQL (a language commonly used for querying databases), will notice that the LINQ present in this example is very similar to SQL (albeit somewhat backwards). In particular, the following LINQ statement:

```
var smallNumbers = from int num in randomNumbers
                   where num < 5
                   select num;
```

can be seen as similar to the following SQL query:

```
SELECT num FROM randomNumbers WHERE num < 5
```

```
static void Main(string[] args)
{
    Vector<int> randomNumbers = new Vector<int>();
    Random rnd = new Random();

    for (int i = 0; i < 20; i++)
        randomNumbers.Add(rnd.Next(0, 11));

    Console.Write("Random numbers: ");
    foreach (int num in randomNumbers)
        Console.Write("{0} ", num);
    Console.WriteLine();

    var smallNumbers = from int num in randomNumbers
                       where num < 5
                       select num;

    Console.Write("Small numbers: ");
    foreach (int num in smallNumbers)
        Console.Write("{0} ", num);
    Console.WriteLine();
}
```

Figure 11.1: Simple LINQ Example

Note the use of the `var` keyword that represents a variant type[3] (effectively the data type is undefined until a value is assigned to that variable). Also note that this query uses what is known as ***deferred execution***. In particular, nothing actually happens when the LINQ statement is executed. Instead, the actual functionality of the LINQ expression executes when the `smallNumbers` variable is enumerated by the `foreach` loop. This has two important implications:

- The `Vector` collection will be processed by the LINQ expression each time the `smallNumbers` variable is enumerated, i.e., if the `randomNumbers` collection changes, the contents of the `smallNumbers` collection may also change (if any numbers are added with a value less than 5); and
- If there is a problem with the LINQ statement that would result in an exception, the exception will not be thrown until the `foreach` loop is executed (this can be confusing for first time LINQ developers).

The above example uses what is known as a query expression. It is also possible to perform the above query using standard dot notation, which in this case appears much simpler, at least at first glance:

```
var smallNumbers = randomNumbers.Where(num => num < 5);
```

If you look closely at this example, you will notice that a method `Where()` is being invoked on the `Vector` collection that we developed earlier in the trimester. However if you review your implementation of this class, the `System.Object` class from which it inherits, and the `IEnumerable<T>` interface from which it inherits, you will notice that there is no mention of a method `Where()` in any of these elements. This is because the `Where()` method is defined by an ***extension method*** that is implemented by a class in the Microsoft.Net class framework: `System.Linq.Enumerable`.

Extension methods allow new methods to be written that extend objects of any type, without requiring any modification to those types. In this case, the `Where()` method is an extension to any object that implements `IEnumerable<T>`. The `System.Linq.Enumerable` class provides a number of extension methods that are used regularly when developing with LINQ, including:

- `Average()` – calculate the average value for (some field in) the objects in a sequence;
- `Cast()` – convert the objects in a sequence to another type (performs a cast on every object);
- `Concat()` – concatenate (join together) two sequences;
- `Count()` – counts the number of objects in a sequence;
- `Distinct()` – eliminate duplicates from the objects in a sequence;
- `First()` – returns the first object;
- `GroupBy()` – groups the elements according to some field in the objects;
- `Intersect()` – returns objects common in two sequences;
- `Join()` – correlates elements in two sequences according to some key;

[3] Note that in this case, the type returned by the expression is in fact an `IEnumerable<int>`. In general it is good programming practice to always use actual types where possible, however for simplicity we will only use the variant type in this unit.

- `Last()` – returns the last object in the sequence;
- `Max()` – returns the object with the maximum value (for some field);
- `Min()` – returns the object with the minimum value (for some field);
- `OfType()` – returns only the objects matching a particular data type from the sequence;
- `OrderBy()` – returns the objects sorted in ascending order (for some field);
- `OrderByDescending()` – returns the objects in descending order (for some field);
- `Reverse()` – reverses the order of a sequence;
- `Select()` – used to determine how the values in a sequence are returned;
- `Single()` – returns the only element of a sequence (throws `Inval-idOperationException` if there are zero or more than one element in the sequence);
- `SingleOrDefault()` – returns the only element of a sequence, or if there are zero/more than one element in the sequence returns the default value for the relevant type);
- `Sum()` – returns the sum/total of (some field in) the objects;
- `ThenBy()` – used for sorting in ascending order by a secondary field in a sequence where the primary keys used by the `OrderBy()`/`OrderByDescending()` extension methods are equal;
- `ThenByDescending()` – used by sorting in descending order by a secondary field in a sequence where the primary keys used by the `OrderBy()`/`OrderByDescending()` extension methods are equal;
- `ToArray()` – used to convert a sequence to an array;
- `ToList()` – used to convert a sequence to a `List<T>`;
- `ToDictionary()` – used to convert a sequence to a `Dictionary<TK, TV>`; and
- `Union()` – merges two sequences.

The other unusual part of the standard dot notation example that is the parameter being passed to the method:

```
num => num < 5
```

This expression is what is known as a ***lambda expression***, introduced along with the LINQ functionality for defining ***anonymous methods***. Anonymous methods are, as the name implies, methods which do not have names. The actual parameter to the `Where()` method is a delegate, which takes a single parameter (representing a single object stored in the `Vector` collection) and returns a boolean value (true if the object should be included in the output). Effectively, the above lambda expression translates to:

```
bool anonymous_method(int num)
{
    return num < 5;
}
```

The syntax for a lamda expression is as follows:

```
param_name => expression
```

for a single parameter, or:

```
(param1, param2[, ...]) => expression
```

for multiple parameters. Note that it is also possible to use a code block if multiple statements are required, resulting in a definition that looks similar to a method, i.e.,

```
param_name =>
{
        statement
        [...]
        return result;
}
```

Multiple parameters can also be used as above. Understanding lambda expressions is critical to being able to work effectively with LINQ.

The above information, supported by the the many tutorials and examples available on the web, should be enough to get you started with LINQ in your own programs. To truly master LINQ however would require several dedicated weeks. The intention of this chapter is only to familiarise you with LINQ and allow you to start the learning process.

11.4. A Final Word

The task of learning programming is well known to be extraordinarily difficult. Upon successful completion of this unit, all students should have adequate knowledge to begin working towards a successful career in programming. This moment represents an excellent time to reflect on all that you have learned, beginning with the absolute fundamentals (declaring variables, writing conditions, applying loops, etc.) through to the end of the content presented in this unit.

In reflecting on what you have learned, consider carefully the difficulties you experienced when you first began learning programming, and how remarkably trivial these tasks now appear to you. This single point represents the most important lesson you can learn about programming – the more programming you do, the easier it becomes. Knowledge of programming represents what will become only the smallest part of your toolkit. The experience you have gained throughout your studies already dominates your programming skills. At this point in time, if you choose to continue to practice programming, you will be well prepared for what is one of the most challenging, rewarding, and satisfying occupations available in the IT industry. Consider your next moves carefully!

Task 11.1. Getting Started with LINQ

Objective: Learning to work effectively with LINQ will take a great deal of practice and there are entire books dedicated to the task of teaching LINQ. Instead, our objective is only to gain familiarity with LINQ expressions and some of the fundamental tasks that can be completed using LINQ, rather than mastering this new concept.

This week's task consists of two sub-tasks. The first sub-task is to create a class `Customer` with the following features:

- Private attributes for family name (`string`), given name (`string`), and balance (`decimal`);
- Public read-only properties encapsulating the above attributes;

- A custom constructor accepting three parameters (family name, given name, and balance) to initialise the above attributes; and
- Override the `ToString()` method to produce the following output for positive balances:

  name balance`CR`

  and for negative balances:

  name balance`DR`
- Notes:
  - The name should be output in the format:

    family_name, *given_name*

    and should have a total field width of 40 characters.
  - The numeric balance value should always be output as a positive value and use currency format with a field width of 10 characters (do not change the attribute value).

The second sub-task is to write a `Main` method with the following features:

- Declare and initialise a collection of customers using the following code:

```
List<Customer> customerList = new List<Customer>();
customerList.AddRange(
    new Customer[] {
        new Customer("Anderson",    "A", -100M),
        new Customer("Brown",       "B", 200),
        new Customer("Harris",      "C", -700),
        new Customer("Johnson",     "D", 800),
        new Customer("Jones",       "E", -300),
        new Customer("Kelly",       "F", 400),
        new Customer("King",        "G", -900),
        new Customer("Lee",         "H", 1000),
        new Customer("Martin",      "I", -500),
        new Customer("Nguyen",      "J", 600),
        new Customer("Robinson",    "K", -600),
        new Customer("Ryan",        "L", 500),
        new Customer("Smith",       "M", -1000),
        new Customer("Taylor",      "N", 900),
        new Customer("Thomas",      "O", -400),
        new Customer("Thompson",    "P", 300),
        new Customer("Williams",    "Q", -800),
        new Customer("Walker",      "R", 700),
        new Customer("Wilson",      "S", -200),
        new Customer("White",       "T", 100)
    }
);
```

- Using only `Console` output statements, LINQ statements, and `foreach` loops, produce the following output:

```
Customers in credit:
        Lee, H                                   $1,000.00CR
        Taylor, N                                  $900.00CR
        Johnson, D                                 $800.00CR
        Walker, R                                  $700.00CR
```

```
            Nguyen, J                                   $600.00CR
            Ryan, L                                     $500.00CR
            Kelly, F                                    $400.00CR
            Thompson, P                                 $300.00CR
            Brown, B                                    $200.00CR
            White, T                                    $100.00CR

    Customers in debit:
            Smith, M                                  $1,000.00DR
            King, G                                     $900.00DR
            Williams, Q                                 $800.00DR
            Harris, C                                   $700.00DR
            Robinson, K                                 $600.00DR
            Martin, I                                   $500.00DR
            Thomas, O                                   $400.00DR
            Jones, E                                    $300.00DR
            Wilson, S                                   $200.00DR
            Anderson, A                                 $100.00DR
```

Note that there are many examples and guides for LINQ available on the web that may help you complete the above task, however the solution to this problem only requires use of the where, orderby, and select elements of a query expression (or `Where`*,* `OrderBy`*, and* `Select` *extension methods).*

Index