

# Chapter 1

## Introduction

These lecture notes cover the key ideas involved in designing *algorithms*. We shall see how they depend on the design of suitable *data structures*, and how some structures and algorithms are more *efficient* than others for the same task. We will concentrate on a few basic tasks, such as storing, sorting and searching data, that underlie much of computer science, but the techniques discussed will be applicable much more generally.

We will start by studying some key data structures, such as arrays, lists, queues, stacks and trees, and then move on to explore their use in a range of different searching and sorting algorithms. This leads on to the consideration of approaches for more efficient storage of data in hash tables. Finally, we will look at graph based representations and cover the kinds of algorithms needed to work efficiently with them. Throughout, we will investigate the computational efficiency of the algorithms we develop, and gain intuitions about the pros and cons of the various potential approaches for each task.

We will not restrict ourselves to implementing the various data structures and algorithms in particular computer programming languages (e.g., *Java*, *C*, *OCaml*), but specify them in simple *pseudocode* that can easily be implemented in any appropriate language.

### 1.1 Algorithms as opposed to programs

An *algorithm* for a particular task can be defined as “a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time”. As such, an *algorithm* must be precise enough to be understood by *human beings*. However, in order to be *executed* by a *computer*, we will generally need a *program* that is written in a rigorous formal language; and since computers are quite inflexible compared to the human mind, programs usually need to contain more details than algorithms. Here we shall ignore most of those programming details and concentrate on the design of algorithms rather than programs.

The task of *implementing* the discussed algorithms as computer programs is important, of course, but these notes will concentrate on the theoretical aspects and leave the practical programming aspects to be studied elsewhere. Having said that, we will often find it useful to write down segments of actual programs in order to clarify and test certain theoretical aspects of algorithms and their data structures. It is also worth bearing in mind the distinction between different programming paradigms: *Imperative Programming* describes computation in terms of instructions that change the program/data state, whereas *Declarative Programming*

specifies what the program should accomplish without describing how to do it. These notes will primarily be concerned with developing algorithms that map easily onto the imperative programming approach.

Algorithms can obviously be described in plain English, and we will sometimes do that. However, for computer scientists it is usually easier and clearer to use something that comes somewhere in between formatted English and computer program code, but is not runnable because certain details are omitted. This is called *pseudocode*, which comes in a variety of forms. Often these notes will present segments of pseudocode that are very similar to the languages we are mainly interested in, namely the overlap of *C* and *Java*, with the advantage that they can easily be inserted into runnable programs.

## 1.2 Fundamental questions about algorithms

Given an algorithm to solve a particular problem, we are naturally led to ask:

1. What is it supposed to do?
2. Does it really do what it is supposed to do?
3. How efficiently does it do it?

The technical terms normally used for these three aspects are:

1. Specification.
2. Verification.
3. Performance analysis.

The details of these three aspects will usually be rather problem dependent.

The *specification* should formalize the crucial details of the problem that the algorithm is intended to solve. Sometimes that will be based on a particular representation of the associated data, and sometimes it will be presented more abstractly. Typically, it will have to specify how the inputs and outputs of the algorithm are related, though there is no general requirement that the specification is complete or non-ambiguous.

For simple problems, it is often easy to see that a particular algorithm will always work, i.e. that it satisfies its specification. However, for more complicated specifications and/or algorithms, the fact that an algorithm satisfies its specification may not be obvious at all. In this case, we need to spend some effort *verifying* whether the algorithm is indeed correct. In general, testing on a few particular inputs can be enough to show that the algorithm is incorrect. However, since the number of different potential inputs for most algorithms is infinite in theory, and huge in practice, more than just testing on particular cases is needed to be sure that the algorithm satisfies its specification. We need *correctness proofs*. Although we will discuss proofs in these notes, and useful relevant ideas like *invariants*, we will usually only do so in a rather informal manner (though, of course, we will attempt to be rigorous). The reason is that we want to concentrate on the data structures and algorithms. Formal verification techniques are complex and will normally be left till after the basic ideas of these notes have been studied.

Finally, the *efficiency* or *performance* of an algorithm relates to the *resources* required by it, such as how quickly it will run, or how much computer memory it will use. This will

usually depend on the problem instance size, the choice of data representation, and the details of the algorithm. Indeed, this is what normally drives the development of new data structures and algorithms. We shall study the general ideas concerning efficiency in Chapter 5, and then apply them throughout the remainder of these notes.

### 1.3 Data structures, abstract data types, design patterns

For many problems, the ability to formulate an efficient algorithm depends on being able to organize the data in an appropriate manner. The term *data structure* is used to denote a particular way of organizing data for particular types of operation. These notes will look at numerous data structures ranging from familiar arrays and lists to more complex structures such as trees, heaps and graphs, and we will see how their choice affects the efficiency of the algorithms based upon them.

Often we want to talk about data structures without having to worry about all the implementational details associated with particular programming languages, or how the data is stored in computer memory. We can do this by formulating abstract mathematical models of particular classes of data structures or data types which have common features. These are called *abstract data types*, and are defined only by the operations that may be performed on them. Typically, we specify how they are built out of more *primitive data types* (e.g., integers or strings), how to extract that data from them, and some basic checks to control the flow of processing in algorithms. The idea that the implementational details are hidden from the user and protected from outside access is known as *encapsulation*. We shall see many examples of abstract data types throughout these notes.

At an even higher level of abstraction are *design patterns* which describe the design of algorithms, rather the design of data structures. These embody and generalize important design concepts that appear repeatedly in many problem contexts. They provide a general structure for algorithms, leaving the details to be added as required for particular problems. These can speed up the development of algorithms by providing familiar proven algorithm structures that can be applied straightforwardly to new problems. We shall see a number of familiar design patterns throughout these notes.

### 1.4 Textbooks and web-resources

To fully understand data structures and algorithms you will almost certainly need to complement the introductory material in these notes with textbooks or other sources of information. The lectures associated with these notes are designed to help you understand them and fill in some of the gaps they contain, but that is unlikely to be enough because often you will need to see more than one explanation of something before it can be fully understood.

There is no single best textbook that will suit everyone. The subject of these notes is a classical topic, so there is no need to use a textbook published recently. Books published 10 or 20 years ago are still good, and new good books continue to be published every year. The reason is that these notes cover important fundamental material that is taught in all university degrees in computer science. These days there is also a lot of very useful information to be found on the internet, including complete freely-downloadable books. It is a good idea to go to your library and browse the shelves of books on data structures and algorithms. If you like any of them, download, borrow or buy a copy for yourself, but make sure that most of the

topics in the above contents list are covered. Wikipedia is generally a good source of fairly reliable information on all the relevant topics, but you hopefully shouldn't need reminding that not everything you read on the internet is necessarily true. It is also worth pointing out that there are often many different equally-good ways to solve the same task, different equally-sensible names used for the same thing, and different equally-valid conventions used by different people, so don't expect all the sources of information you find to be an exact match with each other or with what you find in these notes.

## 1.5 Overview

These notes will cover the principal fundamental data structures and algorithms used in computer science, and bring together a broad range of topics covered elsewhere into a coherent framework. Data structures will be formulated to represent various types of information in such a way that it can be conveniently and efficiently manipulated by the algorithms we develop. Throughout, the recurring practical issues of algorithm specification, verification and performance analysis will be discussed.

We shall begin by looking at some widely used basic data structures (namely arrays, linked lists, stacks and queues), and the advantages and disadvantages of the associated abstract data types. Then we consider the ubiquitous problem of searching, and how that leads on to the general ideas of computational efficiency and complexity. That will leave us with the necessary tools to study three particularly important data structures: trees (in particular, binary search trees and heap trees), hash tables, and graphs. We shall learn how to develop and analyse increasingly efficient algorithms for manipulating and performing useful operations on those structures, and look in detail at developing efficient processes for data storing, sorting, searching and analysis. The idea is that once the basic ideas and examples covered in these notes are understood, dealing with more complex problems in the future should be straightforward.

## Chapter 2

# Arrays, Iteration, Invariants

Data is ultimately *stored* in computers as patterns of bits, though these days most programming languages deal with higher level objects, such as characters, integers, and floating point numbers. Generally, we need to build algorithms that manipulate collections of such objects, so we need procedures for storing and sequentially processing them.

### 2.1 Arrays

In computer science, the obvious way to store an ordered collection of items is as an *array*. Array items are typically stored in a sequence of computer memory locations, but to discuss them, we need a convenient way to write them down on paper. We can just write the items in order, separated by commas and enclosed by square brackets. Thus,

$$[1, 4, 17, 3, 90, 79, 4, 6, 81]$$

is an example of an array of integers. If we call this array  $a$ , we can write it as:

$$a = [1, 4, 17, 3, 90, 79, 4, 6, 81]$$

This array  $a$  has 9 items, and hence we say that its *size* is 9. In everyday life, we usually start counting from 1. When we work with arrays in computer science, however, we more often (though not always) start from 0. Thus, for our array  $a$ , its positions are  $0, 1, 2, \dots, 7, 8$ . The element in the 8<sup>th</sup> position is 81, and we use the notation  $a[8]$  to denote this element. More generally, for any integer  $i$  denoting a position, we write  $a[i]$  to denote the element in the  $i^{\text{th}}$  position. This position  $i$  is called an *index* (and the plural is *indices*). Then, in the above example,  $a[0] = 1$ ,  $a[1] = 4$ ,  $a[2] = 17$ , and so on.

It is worth noting at this point that the symbol  $=$  is quite *overloaded*. In mathematics, it stands for equality. In most modern programming languages,  $=$  denotes assignment, while equality is expressed by  $==$ . We will typically use  $=$  in its mathematical meaning, unless it is written as part of code or pseudocode.

We say that the individual items  $a[i]$  in the array  $a$  are *accessed* using their index  $i$ , and one can move sequentially through the array by incrementing or decrementing that index, or jump straight to a particular item given its index value. Algorithms that process data stored as arrays will typically need to visit systematically all the items in the array, and apply appropriate operations on them.

## 2.2 Loops and Iteration

The standard approach in most programming languages for repeating a process a certain number of times, such as moving sequentially through an array to perform the same operations on each item, involves a *loop*. In *pseudocode*, this would typically take the general form

```
For i = 1, ..., N,  
    do something
```

and in programming languages like *C* and *Java* this would be written as the *for-loop*

```
for( i = 0 ; i < N ; i++ ) {  
    // do something  
}
```

in which a *counter*  $i$  keep tracks of doing “the something”  $N$  times. For example, we could compute the sum of all 20 items in an array  $a$  using

```
for( i = 0, sum = 0 ; i < 20 ; i++ ) {  
    sum += a[i];  
}
```

We say that there is *iteration* over the index  $i$ . The general *for-loop* structure is

```
for( INITIALIZATION ; CONDITION ; UPDATE ) {  
    REPEATED PROCESS  
}
```

in which any of the four parts are optional. One way to write this out explicitly is

```
INITIALIZATION  
if ( not CONDITION ) go to LOOP FINISHED  
LOOP START  
    REPEATED PROCESS  
    UPDATE  
    if ( CONDITION ) go to LOOP START  
LOOP FINISHED
```

In these notes, we will regularly make use of this basic loop structure when operating on data stored in arrays, but it is important to remember that different programming languages use different syntax, and there are numerous variations that check the condition to terminate the repetition at different points.

## 2.3 Invariants

An *invariant*, as the name suggests, is a condition that does not change during execution of a given program or algorithm. It may be a simple inequality, such as “ $i < 20$ ”, or something more abstract, such as “the items in the array are sorted”. Invariants are important for data structures and algorithms because they enable *correctness proofs* and *verification*.

In particular, a *loop-invariant* is a condition that is true at the beginning and end of every iteration of the given loop. Consider the standard simple example of a procedure that finds the minimum of  $n$  numbers stored in an array  $a$ :

```

minimum(int n, float a[n]) {
    float min = a[0];
    // min equals the minimum item in a[0],...,a[0]
    for(int i = 1 ; i != n ; i++) {
        // min equals the minimum item in a[0],...,a[i-1]
        if (a[i] < min) min = a[i];
    }
    // min equals the minimum item in a[0],...,a[i-1], and i==n
    return min;
}

```

At the beginning of each iteration, and end of any iterations before, the invariant “*min* equals the minimum item in  $a[0], \dots, a[i-1]$ ” is true – it starts off true, and the repeated process and update clearly maintain its truth. Hence, when the loop terminates with “ $i == n$ ”, we know that “*min* equals the minimum item in  $a[0], \dots, a[n-1]$ ” and hence we can be sure that *min* can be returned as the required minimum value. This is a kind of *proof by induction*: the invariant is true at the start of the loop, and is preserved by each iteration of the loop, therefore it must be true at the end of the loop.

As we noted earlier, formal proofs of correctness are beyond the scope of these notes, but identifying suitable loop invariants and their implications for algorithm correctness as we go along will certainly be a useful exercise. We will also see how invariants (sometimes called *inductive assertions*) can be used to formulate similar correctness proofs concerning properties of data structures that are defined inductively.

## Chapter 3

# Lists, Recursion, Stacks, Queues

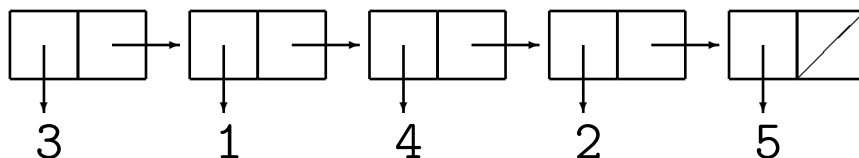
We have seen how arrays are a convenient way to *store* collections of items, and how loops and iteration allow us to sequentially process those items. However, arrays are not always the most efficient way to store collections of items. In this section, we shall see that lists may be a better way to store collections of items, and how recursion may be used to process them. As we explore the details of storing collections as lists, the advantages and disadvantages of doing so for different situations will become apparent.

### 3.1 Linked Lists

A list can involve virtually anything, for example, a list of integers [3, 2, 4, 2, 5], a shopping list [apples, butter, bread, cheese], or a list of web pages each containing a picture and a link to the next web page. When considering *lists*, we can speak about them on different levels - on a very abstract level (on which we can define what we mean by a list), on a level on which we can depict lists and communicate as humans about them, on a level on which computers can communicate, or on a machine level in which they can be implemented.

#### Graphical Representation

Non-empty *lists* can be represented by *two-cells*, in each of which the first cell contains a pointer to a list element and the second cell contains a *pointer* to either the empty list or another two-cell. We can depict a pointer to the empty list by a diagonal bar or cross through the cell. For instance, the list [3, 1, 4, 2, 5] can be represented as:



#### Abstract Data Type “List”

On an abstract level, a list can be *constructed* by the two *constructors*:

- `EmptyList`, which gives you the *empty list*, and



- `MakeList(element, list)`, which puts an *element* at the top of an existing *list*.

Using those, our last example list can be constructed as

```
MakeList(3, MakeList(1, MakeList(4, MakeList(2, MakeList(5, EmptyList)))))
```

and it is clearly possible to *construct* any list in this way.

This *inductive* approach to data structure creation is very powerful, and we shall use it many times throughout these notes. It starts with the “*base case*”, the `EmptyList`, and then builds up increasingly complex lists by repeatedly applying the “*induction step*”, the `MakeList(element, list)` operator.

It is obviously also important to be able to get back the elements of a list, and we no longer have an item index to use like we have with an array. The way to proceed is to note that a list is always constructed from the first element and the rest of the list. So, conversely, from a non-empty list it must always be possible to get the first element and the rest. This can be done using the two *selectors*, also called *accessor methods*:

- `first(list)`, and
- `rest(list)`.

The selectors will only work for non-empty lists (and give an error or exception on the empty list), so we need a *condition* which tells us whether a given list is empty:

- `isEmpty(list)`

This will need to be used to check every list before passing it to a selector.

We call everything a list that can be constructed by the constructors `EmptyList` and `MakeList`, so that with the selectors `first` and `rest` and the condition `isEmpty`, the following relationships are automatically satisfied (i.e. true):

- `isEmpty(EmptyList)`
- `not isEmpty(MakeList(x, l))` (for any `x` and `l`)
- `first(MakeList(x, l)) = x`
- `rest(MakeList(x, l)) = l`

In addition to constructing and getting back the components of lists, one may also wish to *destructively* change lists. This would be done by so-called *mutators* which change either the first element or the rest of a non-empty list:

- `replaceFirst(x, l)`
- `replaceRest(r, l)`

For instance, with `l = [3, 1, 4, 2, 5]`, applying `replaceFirst(9, l)` changes `l` to `[9, 1, 4, 2, 5]`. and then applying `replaceRest([6, 2, 3, 4], l)` changes it to `[9, 6, 2, 3, 4]`.

We shall see that the concepts of *constructors*, *selectors* and *conditions* are common to virtually all abstract data types. Throughout these notes, we will be formulating our data representations and algorithms in terms of appropriate definitions of them.

## XML Representation

In order to communicate data structures between different computers and possibly different programming languages, *XML* (eXtensible Markup Language) has become a quasi-standard. The above list could be represented in XML as:

```
<ol>
  <li>3</li>
  <li>1</li>
  <li>4</li>
  <li>2</li>
  <li>5</li>
</ol>
```

However, there are usually many different ways to represent the same object in XML. For instance, a cell-oriented representation of the above list would be:

```
<cell>
  <first>3</first>
  <rest>
    <cell>
      <first>1</first>
      <rest>
        <cell>
          <first>4</first>
          <rest>
            <cell>
              <first>2</first>
              <rest>
                <first>5</first>
                <rest>EmptyList</rest>
              </rest>
            </cell>
          </rest>
        </cell>
      </rest>
    </cell>
  </rest>
</cell>
```

While this looks complicated for a simple list, it is not, it is just a bit lengthy. XML is flexible enough to represent and communicate very complicated structures in a uniform way.

## Implementation of Lists

There are many different *implementations* possible for lists, and which one is best will depend on the primitives offered by the programming language being used.

The programming language *Lisp* and its derivatives, for instance, take lists as the most important primitive data structure. In some other languages, it is more natural to implement

lists as arrays. However, that can be problematic because lists are conceptually not limited in size, which means array based implementation with fixed-sized arrays can only approximate the general concept. For many applications, this is not a problem because a maximal number of list members can be determined a priori (e.g., the maximum number of students taking one particular module is limited by the total number of students in the University). More general purpose implementations follow a pointer based approach, which is close to the diagrammatic representation given above. We will not go into the details of all the possible implementations of lists here, but such information is readily available in the standard textbooks.

## 3.2 Recursion

We previously saw how iteration based on for-loops was a natural way to process collections of items stored in arrays. When items are stored as linked-lists, there is no index for each item, and *recursion* provides the natural way to process them. The idea is to formulate procedures which involve at least one step that invokes (or calls) the procedure itself. We will now look at how to implement two important *derived procedures* on lists, **last** and **append**, which illustrate how recursion works.

To find the last element of a list **l** we can simply keep removing the first remaining item till there are no more left. This *algorithm* can be written in *pseudocode* as:

```
last(l) {
  if ( isEmpty(l) )
    error('Error: empty list in last')
  elseif ( isEmpty(rest(l)) )
    return first(l)
  else
    return last(rest(l))
}
```

The running time of this depends on the length of the list, and is proportional to that length, since **last** is called as often as there are elements in the list. We say that the procedure has *linear time complexity*, that is, if the length of the list is increased by some factor, the execution time is increased by the same factor. Compared to the *constant time complexity* which access to the last element of an array has, this is quite bad. It does not mean, however, that lists are inferior to arrays in general, it just means that lists are not the ideal data structure when a program has to access the last element of a long list very often.

Another useful procedure allows us to *append* one list **l2** to another list **l1**. Again, this needs to be done one item at a time, and that can be accomplished by repeatedly taking the first remaining item of **l1** and adding it to the front of the remainder appended to **l2**:

```
append(l1,l2) {
  if ( isEmpty(l1) )
    return l2
  else
    return MakeList(first(l1),append(rest(l1),l2))
}
```

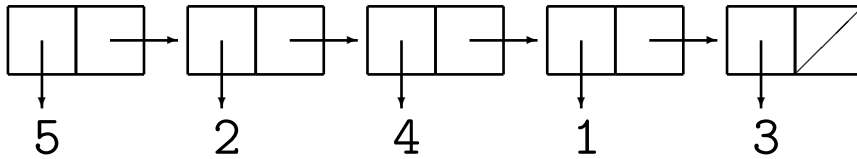
The time complexity of this procedure is proportional to the length of the first list, **l1**, since we have to call **append** as often as there are elements in **l1**.

### 3.3 Stacks

*Stacks* are, on an abstract level, equivalent to linked lists. They are the ideal data structure to model a *First-In-Last-Out (FILO)*, or *Last-In-First-Out (LIFO)*, strategy in search.

#### Graphical Representation

Their relation to linked lists means that their graphical representation can be the same, but one has to be careful about the order of the items. For instance, the stack created by inserting the numbers [3, 1, 4, 2, 5] in that order would be represented as:



#### Abstract Data Type “Stack”

Despite their relation to linked lists, their different use means the *primitive operators* for stacks are usually given different names. The two *constructors* are:

- `EmptyStack`, the empty stack, and
- `push(element, stack)`, which takes an element and pushes it on top of an existing stack,

and the two *selectors* are:

- `top(stack)`, which gives back the top most element of a stack, and
- `pop(stack)`, which gives back the stack without the top most element.

The selectors will work only for non-empty stacks, hence we need a *condition* which tells whether a stack is empty:

- `isEmpty(stack)`

We have equivalent automatically-true relationships to those we had for the lists:

- `isEmpty(EmptyStack)`
- `not isEmpty(push(x, s))` (for any `x` and `s`)
- `top(push(x, s)) = x`
- `pop(push(x, s)) = s`

In summary, we have the direct correspondences:

	constructors		selectors		condition
List	<code>EmptyList</code>	<code>MakeList</code>	<code>first</code>	<code>rest</code>	<code>isEmpty</code>
Stack	<code>EmptyStack</code>	<code>push</code>	<code>top</code>	<code>pop</code>	<code>isEmpty</code>

So, stacks and linked lists are the same thing, apart from the different names that are used for their constructors and selectors.

## Implementation of Stacks

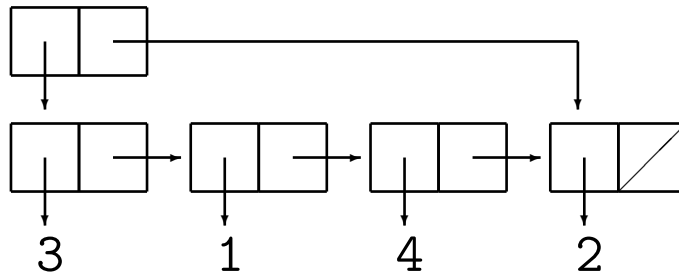
There are two different ways we can think about implementing stacks. So far we have implied a *functional* approach. That is, `push` does not change the original stack, but creates a new stack out of the original stack and a new element. That is, there are at least two stacks around, the original one and the newly created one. This functional view is quite convenient. If we apply `top` to a particular stack, we will always get the same element. However, from a practical point of view, we may not want to create lots of new stacks in a program, because of the obvious memory management implications. Instead it might be better to think of a single stack which is destructively changed, so that after applying `push` the original stack no longer exists, but has been changed into a new stack with an extra element. This is conceptually more difficult, since now applying `top` to a given stack may give different answers, depending on how the state of the system has changed. However, as long as we keep this difference in mind, ignoring such implementational details should not cause any problems.

## 3.4 Queues

A *queue* is a data structure used to model a *First-In-First-Out (FIFO)* strategy. Conceptually, we add to the end of a queue and take away elements from its front.

### Graphical Representation

A queue can be graphically represented in a similar way to a list or stack, but with an additional two-cell in which the first element points to the front of the list of all the elements in the queue, and the second element points to the last element of the list. For instance, if we insert the elements [3, 1, 4, 2] into an initially empty queue, we get:



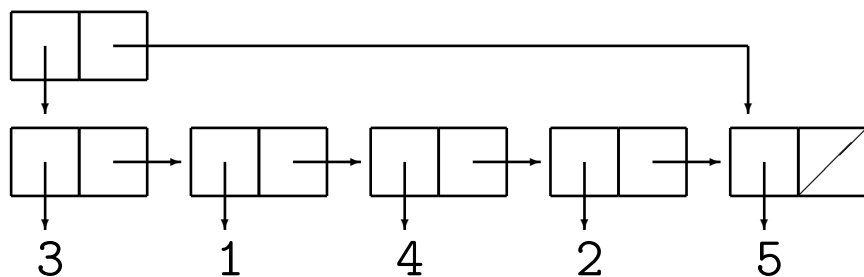
This arrangement means that taking the first element of the queue, or adding an element to the back of the queue, can both be done efficiently. In particular, they can both be done with constant effort, i.e. independently of the queue length.

### Abstract Data Type “Queue”

On an abstract level, a queue can be *constructed* by the two *constructors*:

- `EmptyQueue`, the empty queue, and
- `push(element, queue)`, which takes an element and a queue and returns a queue in which the element is added to the original queue at the end.

For instance, by applying `push(5, q)` where `q` is the queue above, we get



The two *selectors* are the same as for stacks:

- `top(queue)`, which gives the top element of a queue, that is, 3 in the example, and
- `pop(queue)`, which gives the queue without the top element.

And, as with stacks, the selectors only work for non-empty queues, so we again need a *condition* which returns whether a queue is empty:

- `isEmpty(queue)`

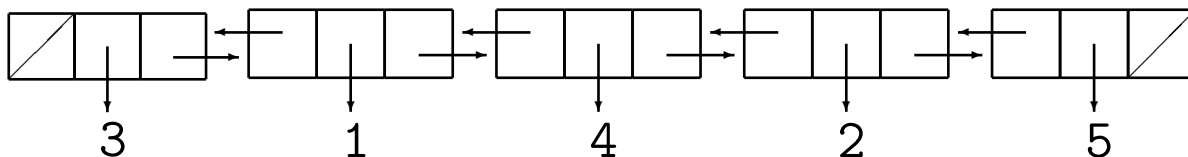
In later chapters we shall see practical examples of how queues and stacks operate with different effect.

### 3.5 Doubly Linked Lists

A *doubly linked list* might be useful when working with something like a list of web pages, which has each page containing a picture, a link to the previous page, and a link to the next page. For a simple list of numbers, a linked list and a doubly linked list may look the same, e.g., [3, 1, 4, 2, 5]. However, the doubly linked list also has an easy way to get the previous element, as well as to the next element.

#### Graphical Representation

Non-empty doubly linked lists can be represented by *three-cells*, where the first cell contains a pointer to another three-cell or to the empty list, the second cell contains a pointer to the list element and the third cell contains a pointer to another three-cell or the empty list. Again, we depict the empty list by a diagonal bar or cross through the appropriate cell. For instance, [3, 1, 4, 2, 5] would be represented as doubly linked list as:



#### Abstract Data Type “Doubly Linked List”

On an abstract level, a doubly linked list can be *constructed* by the three *constructors*:

- `EmptyList`, the empty list, and

- `MakeListLeft(element, list)`, which takes an element and a doubly linked list and returns a new doubly linked list with the element added to the left of the original doubly linked list.
- `MakeListRight(element, list)`, which takes an element and a doubly linked list and returns a new doubly linked list with the element added to the right of the original doubly linked list.

It is clear that it may possible to construct a given doubly linked list in more than one way. For example, the doubly linked list represented above can be constructed by either of:

```
MakeListLeft(3, MakeListLeft(1, MakeListLeft(4, MakeListLeft(2,
                                                    MakeListLeft(5, EmptyList))))))

MakeListLeft(3, MakeListLeft(1, MakeListRight(5, MakeListRight(2,
                                                                MakeListLeft(4, EmptyList))))))
```

In the case of doubly linked lists, we have four *selectors*:

- `firstLeft(list)`,
- `restLeft(list)`,
- `firstRight(list)`, and
- `restRight(list)`.

Then, since the selectors only work for non-empty lists, we also need a *condition* which returns whether a list is empty:

- `isEmpty(list)`

This leads to automatically-true relationships such as:

- `isEmpty(EmptyList)`
- `not isEmpty(MakeListLeft(x, l))` (for any `x` and `l`)
- `not isEmpty(MakeListRight(x, l))` (for any `x` and `l`)
- `firstLeft(MakeListLeft(x, l)) = x`
- `restLeft(MakeListLeft(x, l)) = l`
- `firstRight(MakeListRight(x, l)) = x`
- `restRight(MakeListRight(x, l)) = l`

## Circular Doubly Linked List

As a simple extension of the standard doubly linked list, one can define a *circular doubly linked list* in which the left-most element points to the right-most element, and vice versa. This is useful when we might need to move efficiently through a whole list of items, but might not be starting from one of two particular end points.

### 3.6 Advantage of Abstract Data Types

It is clear that the implementation of the abstract linked-list data type has the disadvantage that certain useful procedures may not be directly accessible. For instance, the standard *abstract data type* of a list does not offer an efficient procedure `last(1)` to give the last element in the list, whereas it would be trivial to find the last element of an array of a known number of elements. One could modify the linked-list data type by maintaining a pointer to the last item, as we did for the queue data type, but we still wouldn't have an easy way to access intermediate items. While `last(1)` and `getItem(i,1)` procedures can easily be implemented using the primitive constructors, selectors, and conditions, they are likely to be less efficient than making use of certain aspects of the underlying implementation.

That disadvantage leads to an obvious question: Why should we want to use abstract data types when they often lead to less efficient algorithms? Aho, Hopcroft and Ullman (1983) provide a clear answer in their book:

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

This advantage will become clearer when we study more complex abstract data types and algorithms in later chapters.



## Chapter 4

# Searching

An important and recurring problem in computing is that of *locating information*. More succinctly, this problem is known as *searching*. This is a good topic to use for a preliminary exploration of the various issues involved in algorithm design.

### 4.1 Requirements for searching

Clearly, the information to be searched has to first be *represented* (or *encoded*) somehow. This is where *data structures* come in. Of course, in a computer, everything is ultimately represented as sequences of binary digits (bits), but this is too low level for most purposes. We need to develop and study useful data structures that are closer to the way humans think, or at least more structured than mere sequences of bits. This is because it is humans who have to develop and maintain the software systems – computers merely run them.

After we have chosen a suitable representation, the represented information has to be processed somehow. This is what leads to the need for *algorithms*. In this case, the process of interest is that of searching. In order to simplify matters, let us assume that we want to search a collection of integer numbers (though we could equally well deal with strings of characters, or any other data type of interest). To begin with, let us consider:

1. The most obvious and simple representation.
2. Two potential algorithms for processing with that representation.

As we have already noted, *arrays* are one of the simplest possible ways of representing collections of numbers (or strings, or whatever), so we shall use that to store the information to be searched. Later we shall look at more complex data structures that may make storing and searching more efficient.

Suppose, for example, that the set of integers we wish to search is  $\{1, 4, 17, 3, 90, 79, 4, 6, 81\}$ . We can write them in an array  $a$  as

$$a = [1, 4, 17, 3, 90, 79, 4, 6, 81]$$

If we ask where 17 is in this array, the answer is 2, the index of that element. If we ask where 91 is, the answer is *nowhere*. It is useful to be able to represent *nowhere* by a number that is not used as a possible index. Since we start our index counting from 0, any negative number would do. We shall follow the convention of using the number  $-1$  to represent *nowhere*. Other (perhaps better) conventions are possible, but we will stick to this here.

## 4.2 Specification of the search problem

We can now formulate a *specification* of our search problem using that data structure:

*Given an array  $a$  and integer  $x$ , find an integer  $i$  such that*

- 1. if there is no  $j$  such that  $a[j]$  is  $x$ , then  $i$  is  $-1$ ,*
- 2. otherwise,  $i$  is any  $j$  for which  $a[j]$  is  $x$ .*

The first clause says that if  $x$  does not occur in the array  $a$  then  $i$  should be  $-1$ , and the second says that if it does occur then  $i$  should be a position where it occurs. If there is more than one position where  $x$  occurs, then this specification allows you to return any of them – for example, this would be the case if  $a$  were  $[17, 13, 17]$  and  $x$  were  $17$ . Thus, the specification is ambiguous. Hence different algorithms with different behaviours can satisfy the same specification – for example, one algorithm may return the smallest position at which  $x$  occurs, and another may return the largest. There is nothing wrong with ambiguous specifications. In fact, in practice, they occur quite often.

## 4.3 A simple algorithm: Linear Search

We can conveniently express the simplest possible *algorithm* in a form of *pseudocode* which reads like English, but resembles a computer program without some of the precision or detail that a computer usually requires:

```
// This assumes we are given an array a of size n and a key x.
For i = 0,1,...,n-1,
    if a[i] is equal to x,
        then we have a suitable i and can terminate returning i.
If we reach this point,
    then x is not in a and hence we must terminate returning -1.
```

Some aspects, such as the ellipsis “...”, are potentially ambiguous, but we, as human beings, know exactly what is meant, so we do not need to worry about them. In a programming language such as *C* or *Java*, one would write something that is more precise like:

```
for ( i = 0 ; i < n ; i++ ) {
    if ( a[i] == x ) return i;
}
return -1;
```

In the case of *Java*, this would be within a method of a class, and more details are needed, such as the parameter  $a$  for the method and a declaration of the auxiliary variable  $i$ . In the case of *C*, this would be within a function, and similar missing details are needed. In either, there would need to be additional code to output the result in a suitable format.

In this case, it is easy to see that the algorithm satisfies the specification (assuming  $n$  is the correct size of the array) – we just have to observe that, because we start counting from zero, the last position of the array is its size minus one. If we forget this, and let  $i$  run from 0 to  $n$  instead, we get an incorrect algorithm. The practical effect of this mistake is that the execution of this algorithm gives rise to an error when the item to be located in the array is

actually not there, because a non-existing location is attempted to be accessed. Depending on the particular language, operating system and machine you are using, the actual effect of this error will be different. For example, in *C* running under Unix, you may get execution aborted followed by the message “segmentation fault”, or you may be given the wrong answer as the output. In *Java*, you will always get an *error message*.

## 4.4 A more efficient algorithm: Binary Search

One always needs to consider whether it is possible to improve upon the performance of a particular algorithm, such as the one we have just created. In the worst case, searching an array of size  $n$  takes  $n$  steps. On average, it will take  $n/2$  steps. For large collections of data, such as all web-pages on the internet, this will be unacceptable in practice. Thus, we should try to organize the collection in such a way that a more efficient algorithm is possible. As we shall see later, there are many possibilities, and the more we demand in terms of efficiency, the more complicated the data structures representing the collections tend to become. Here we shall consider one of the simplest – we still represent the collections by arrays, but now we enumerate the elements in ascending order. The problem of obtaining an ordered list from any given list is known as *sorting* and will be studied in detail in a later chapter.

Thus, instead of working with the previous array [1, 4, 17, 3, 90, 79, 4, 6, 81], we would work with [1, 3, 4, 4, 6, 17, 79, 81, 90], which has the same items but listed in ascending order. Then we can use an improved *algorithm*, which in English-like pseudocode form is:

```
// This assumes we are given a sorted array a of size n and a key x.
// Use integers left and right (initially set to 0 and n-1) and mid.
While left is less than right,
    set mid to the integer part of (left+right)/2, and
    if x is greater than a[mid],
        then      set left to mid+1,
        otherwise set right to mid.
If a[left] is equal to x,
    then      terminate returning left,
    otherwise terminate returning -1.
```

and would correspond to a segment of *C* or *Java* code like:

```
/* DATA */
int a = [1,3,4,4,6,17,79,81,90];
int n = 9;
int x = 79;
/* PROGRAM */
int left = 0, right = n-1, mid;
while ( left < right ) {
    mid = ( left + right ) / 2;
    if ( x > a[mid] ) left = mid+1;
    else right = mid;
}
if ( a[left] == x ) return left;
else return -1;
```

This algorithm works by repeatedly splitting the array into two segments, one going from *left* to *mid*, and the other going from *mid* + 1 to *right*, where *mid* is the position half way from *left* to *right*, and where, initially, *left* and *right* are the leftmost and rightmost positions of the array. Because the array is sorted, it is easy to see which of each pair of segments the searched-for item *x* is in, and the search can then be restricted to that segment. Moreover, because the size of the sub-array going from locations *left* to *right* is halved at each iteration of the while-loop, we only need  $\log_2 n$  steps in either the average or worst case. To see that this runtime behaviour is a big improvement, in practice, over the earlier linear-search algorithm, notice that  $\log_2 1000000$  is approximately 20, so that for an array of size 1000000 only 20 iterations are needed in the worst case of the binary-search algorithm, whereas 1000000 are needed in the worst case of the linear-search algorithm.

With the binary search algorithm, it is not so obvious that we have taken proper care of the boundary condition in the while loop. Also, strictly speaking, this algorithm is not correct because it does not work for the empty array (that has size zero), but that can easily be fixed. Apart from that, is it correct? Try to convince yourself that it is, and then try to explain your argument-for-correctness to a colleague. Having done that, try to *write down* some convincing arguments, maybe one that involves a *loop invariant* and one that doesn't. Most algorithm developers stop at the first stage, but experience shows that it is only when we attempt to write down seemingly convincing arguments that we actually find all the subtle mistakes. Moreover, it is not unusual to end up with a better/clearer algorithm after it has been modified to make its correctness easier to argue.

It is worth considering whether linked-list versions of our two algorithms would work, or offer any advantages. It is fairly clear that we could perform a linear search through a linked list in essentially the same way as with an array, with the relevant pointer returned rather than an index. Converting the binary search to linked list form is problematic, because there is no efficient way to split a linked list into two segments. It seems that our array-based approach is the best we can do with the data structures we have studied so far. However, we shall see later how more complex data structures (trees) can be used to formulate efficient recursive search algorithms.

Notice that we have not yet taken into account how much effort will be required to sort the array so that the binary search algorithm can work on it. Until we know that, we cannot be sure that using the binary search algorithm really is more efficient overall than using the linear search algorithm on the original unsorted array. That may also depend on further details, such as how many times we need to perform a search on the set of *n* items – just once, or as many as *n* times. We shall return to these issues later. First we need to consider in more detail how to compare algorithm efficiency in a reliable manner.