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 $0, 1, 2, \ldots, 7, 8$. The element in the 8^{th} 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^{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 $a[0], ..., a[i-1]$" 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 $a[0], ..., a[n-1]$" 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

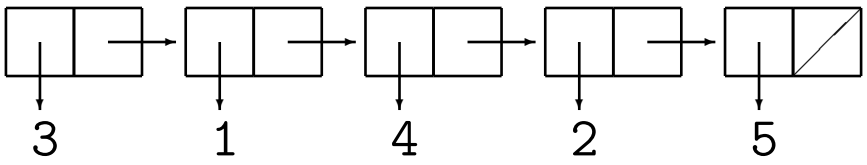
We have seen how arrays are a convenient way to *store* collections of items, and how loops and iteration allow us to sequentially process those items. However, arrays are not always the most efficient way to store collections of items. In this section, we shall see that lists may be a better way to 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 $[\texttt{apples}, \texttt{butter}, \texttt{bread}, \texttt{cheese}]$, 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

$$\texttt{MakeList}(3, \texttt{MakeList}(1, \texttt{MakeList}(4, \texttt{MakeList}(2, \texttt{MakeList}(5, \texttt{EmptyList}))))).$$

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`)
- $\texttt{first}(\texttt{MakeList}(\texttt{x}, \texttt{l})) = \texttt{x}$
- $\texttt{rest}(\texttt{MakeList}(\texttt{x}, \texttt{l})) = \texttt{l}$

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 $\texttt{l} = [3, 1, 4, 2, 5]$, applying $\texttt{replaceFirst}(9, \texttt{l})$ changes `l` to $[9, 1, 4, 2, 5]$. and then applying $\texttt{replaceRest}([6, 2, 3, 4], \texttt{l})$ 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 derivates, 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))
}
```

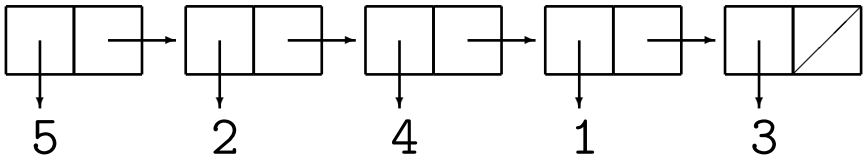
The time complexity of this procedure is proportional to the length of the first list, `l1`, since we have to call `append` as often as there are elements in `l1`.

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	`EmptyList`	`MakeList`	`first`	`rest`	`isEmpty`
Stack	`EmptyStack`	`push`	`top`	`pop`	`isEmpty`

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

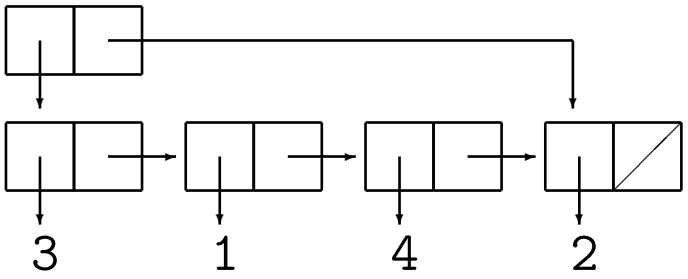
There are two different ways we can think about implementing stacks. So far we have implied a *functional* approach. That is, `push` does not change the original stack, but creates a new stack out of the original stack and a new element. That is, there are at least two stacks around, the original one 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 exits, 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:



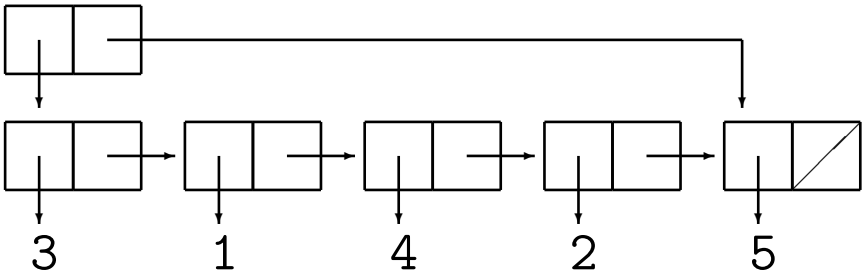
This arrangement means that taking the first element of the queue, or adding an element to the back of the queue, can both be done efficiently. In particular, they can both be done with constant effort, i.e. independently of the queue length.

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`)

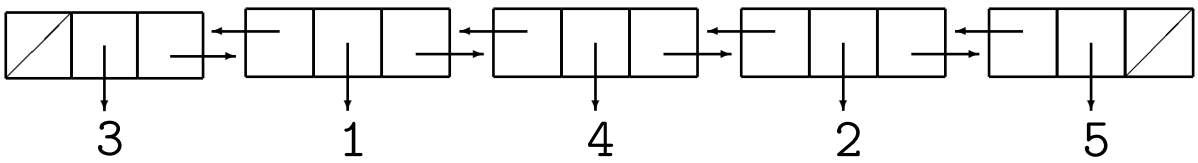
In later chapters we shall see practical examples of how queues and stacks operate with different effect.

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 that 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(l)` 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(l)` and `getItem(i,l)` 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 performa 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.

Chapter 5

Efficiency and Complexity

We have already noted that, when developing algorithms, it is important to consider how *efficient* they are, so we can make informed choices about which are best to use in particular circumstances. So, before moving on to study increasingly complex data structures and algorithms, we first look in more detail at how to measure and describe their efficiency.

5.1 Time versus space complexity

When creating software for serious applications, there is usually a need to judge how quickly an algorithm or program can complete the given tasks. For example, if you are programming a flight booking system, it will not be considered acceptable if the travel agent and customer have to wait for half an hour for a transaction to complete. It certainly has to be ensured that the waiting time is reasonable for the size of the problem, and normally faster execution is better. We talk about the *time complexity* of the algorithm as an indicator of how the execution time depends on the *size* of the data structure.

Another important efficiency consideration is how much memory a given program will require for a particular task, though with modern computers this tends to be less of an issue than it used to be. Here we talk about the *space complexity* as how the memory requirement depends on the size of the data structure.

For a given task, there are often algorithms which trade time for space, and vice versa. For example, we will see that, as a data storage device, *hash tables* have a very good time complexity at the expense of using more memory than is needed by other algorithms. It is usually up to the algorithm/program designer to decide how best to balance the *trade-off* for the application they are designing.

5.2 Worst versus average complexity

Another thing that has to be decided when making efficiency considerations is whether it is the *average case* performance of an algorithm/program that is important, or whether it is more important to guarantee that even in the *worst case* the performance obeys certain rules. For many applications, the average case is more important, because saving time overall is usually more important than guaranteeing good behaviour in the worst case. However, for time-critical problems, such as keeping track of aeroplanes in certain sectors of air space, it may be totally unacceptable for the software to take too long if the worst case arises.

Again, algorithms/programs often trade-off efficiency of the average case against efficiency of the worst case. For example, the most efficient algorithm on average might have a particularly bad worst case efficiency. We will see particular examples of this when we consider efficient algorithms for sorting and searching.

5.3 Concrete measures for performance

These days, we are mostly interested in *time complexity*. For this, we first have to decide how to measure it. Something one might try to do is to just implement the algorithm and run it, and see how long it takes to run, but that approach has a number of problems. For one, if it is a big application and there are several potential algorithms, they would all have to be programmed first before they can be compared. So a considerable amount of time would be wasted on writing programs which will not get used in the final product. Also, the machine on which the program is run, or even the compiler used, might influence the running time. You would also have to make sure that the *data* with which you tested your program is typical for the application it is created for. Again, particularly with big applications, this is not really feasible. This empirical method has another disadvantage: it will not tell you anything useful about the next time you are considering a similar problem.

Therefore complexity is usually best measured in a different way. First, in order to not be bound to a particular programming language or machine architecture, it is better to measure the efficiency of the *algorithm* rather than that of its *implementation*. For this to be possible, however, the algorithm has to be described in a way which very much *looks* like the program to be implemented, which is why algorithms are usually best expressed in a form of *pseudocode* that comes close to the implementation language.

What we need to do to determine the time complexity of an algorithm is count the number of times each operation will occur, which will usually depend on the *size* of the problem. The size of a problem is typically expressed as an integer, and that is typically the number of items that are manipulated. For example, when describing a search algorithm, it is the number of items amongst which we are searching, and when describing a sorting algorithm, it is the number of items to be sorted. So the *complexity* of an algorithm will be given by a function which maps the number of items to the (usually approximate) number of time steps the algorithm will take when performed on that many items.

In the early days of computers, the various operations were each counted in proportion to their particular 'time cost', and added up, with multiplication of integers typically considered much more expensive than their addition. In today's world, where computers have become much faster, and often have dedicated floating-point hardware, the differences in time costs have become less important. However, we still we need to be careful when deciding to consider all operations as being equally costly – applying some function, for example, can take much longer than simply adding two numbers, and swaps generally take many times longer than comparisons. Just counting the most costly operations is often a good strategy.

5.4 Big-O notation for complexity class

Very often, we are not interested in the actual function $C(n)$ that describes the time complexity of an algorithm in terms of the problem size n, but just its *complexity class*. This ignores any constant overheads and small constant factors, and just tells us about the principal growth

of the complexity function with problem size, and hence something about the performance of the algorithm on large numbers of items.

If an algorithm is such that we may consider all steps equally costly, then usually the complexity class of the algorithm is simply determined by the number of loops and how often the content of those loops are being executed. The reason for this is that adding a constant number of instructions which does not change with the size of the problem has no significant effect on the overall complexity for large problems.

There is a standard notation, called the *Big-O notation*, for expressing the fact that constant factors and other insignificant details are being ignored. For example, we saw that the procedure `last(l)` on a list `l` had time complexity that depended linearly on the size n of the list, so we would say that the time complexity of that algorithm is $O(n)$. Similarly, linear search is $O(n)$. For binary search, however, the time complexity is $O(\log_2 n)$.

Before we define complexity classes in a more formal manner, it is worth trying to gain some intuition about what they actually mean. For this purpose, it is useful to choose one function as a representative of each of the classes we wish to consider. Recall that we are considering functions which map natural numbers (the size of the problem) to the set of non-negative real numbers $\mathbb{R}^+$, so the classes will correspond to common mathematical functions such as powers and logarithms. We shall consider later to what degree a representative can be considered 'typical' for its class.

The most common complexity classes (in increasing order) are the following:

- $O(1)$, pronounced 'Oh of one', or *constant* complexity;
- $O(\log_2 \log_2 n)$, 'Oh of log log en';
- $O(\log_2 n)$, 'Oh of log en', or *logarithmic* complexity;
- $O(n)$, 'Oh of en', or *linear* complexity;
- $O(n\log_2 n)$, 'Oh of en log en';
- $O(n^2)$, 'Oh of en squared', or *quadratic* complexity;
- $O(n^3)$, 'Oh of en cubed', or *cubic* complexity;
- $O(2^n)$, 'Oh of two to the en', or *exponential* complexity.

As a representative, we choose the function which gives the class its name – e.g. for $O(n)$ we choose the function $f(n) = n$, for $O(\log_2 n)$ we choose $f(n) = \log_2 n$, and so on. So assume we have algorithms with these functions describing their complexity. The following table lists how many operations it will take them to deal with a problem of a given size:

$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1	1	1	1.00×10^0	1.00×10^0
$\log_2 \log_2 n$	1	2	3	3.32×10^0	4.32×10^0
$\log_2 n$	2	4	8	1.00×10^1	2.00×10^1
n	4	16	2.56×10^2	1.02×10^3	1.05×10^6
$n\log_2 n$	8	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	16	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	64	4096	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	16	65536	1.16×10^{77}	1.80×10^{308}	6.74×10^{315652}

Some of these numbers are so large that it is rather difficult to imagine just how long a time span they describe. Hence the following table gives time spans rather than instruction counts, based on the assumption that we have a computer which can operate at a speed of 1 MIP, where one MIP = a million instructions per second:

$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec
$\log_2 \log_2 n$	1 μsec	2 μsec	3 μsec	3.32 μsec	4.32 μsec
$\log_2 n$	2 μsec	4 μsec	8 μsec	10 μsec	20 μsec
n	4 μsec	16 μsec	256 μsec	1.02 msec	1.05 sec
$n\log_2 n$	8 μsec	64 μsec	2.05 msec	1.02 msec	21 sec
n^2	16 μsec	256 μsec	65.5 msec	1.05 sec	1.8 wk
n^3	64 μsec	4.1 msec	16.8 sec	17.9 min	36,559 yr
2^n	16 μsec	65.5 msec	3.7×10^{63} yr	5.7×10^{294} yr	2.1×10^{315639} yr

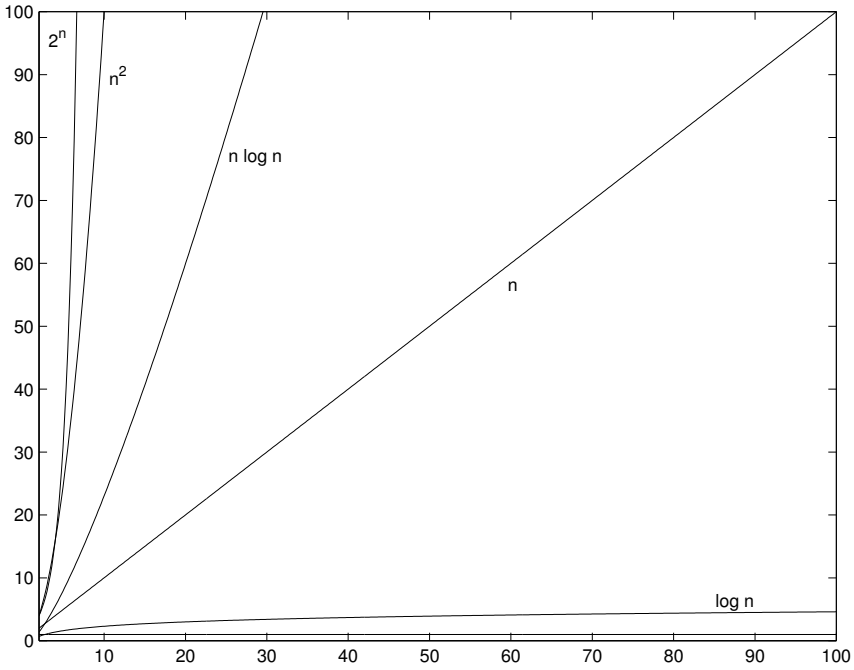
It is clear that, as the sizes of the problems get really big, there can be huge differences in the time it takes to run algorithms from different complexity classes. For algorithms with exponential complexity, $O(2^n)$, even modest sized problems have run times that are greater than the age of the universe (about 1.4×10^{10} yr), and current computers rarely run uninterrupted for more than a few years. This is why complexity classes are so important – they tell us how feasible it is likely to be to run a program with a particular large number of data items. Typically, people do not worry much about complexity for sizes below 10, or maybe 20, but the above numbers make it clear why it is worth thinking about complexity classes where bigger applications are concerned.

Another useful way of thinking about growth classes involves considering how the compute time will vary if the problem size doubles. The following table shows what happens for the various complexity classes:

$f(n)$	If the size of the problem doubles then $f(n)$ will be	
1	the same,	$f(2n) = f(n)$
$\log_2 \log_2 n$	almost the same,	$\log_2 (\log_2 (2n)) = \log_2 (\log_2 (n) + 1)$
$\log_2 n$	more by $1 = \log_2 2$,	$f(2n) = f(n) + 1$
n	twice as big as before,	$f(2n) = 2f(n)$
$n\log_2 n$	a bit more than twice as big as before,	$2n\log_2 (2n) = 2(n\log_2 n) + 2n$
n^2	four times as big as before,	$f(2n) = 4f(n)$
n^3	eight times as big as before,	$f(2n) = 8f(n)$
2^n	the square of what it was before,	$f(2n) = (f(n))^2$

This kind of information can be very useful in practice. We can test our program on a problem that is a half or quarter or one eighth of the full size, and have a good idea of how long we will have to wait for the full size problem to finish. Moreover, that estimate won't be affected by any constant factors ignored in computing the growth class, or the speed of the particular computer it is run on.

The following graph plots some of the complexity class functions from the table. Note that although these functions are only defined on natural numbers, they are drawn as though they were defined for all real numbers, because that makes it easier to take in the information presented.



It is clear from these plots why the non-principal growth terms can be safely ignored when computing algorithm complexity.

5.5 Formal definition of complexity classes

We have noted that complexity classes are concerned with *growth*, and the tables and graph above have provided an idea of what different behaviours mean when it comes to growth. There we have chosen a representative for each of the complexity classes considered, but we have not said anything about just how 'representative' such an element is. Let us now consider a more formal definition of a 'big O' class:

Definition. A function g belongs to the *complexity class* $O(f)$ if there is a number $n_0 \in \mathbb{N}$ and a constant $c > 0$ such that for all $n \geq n_0$, we have that $g(n) \leq c * f(n)$. We say that the function g is 'eventually smaller' than the function $c * f$.

It is not totally obvious what this implies. First, we do not need to know exactly *when* g becomes smaller than $c * f$. We are only interested in the *existence* of n_0 such that, from then on, g is smaller than $c * f$. Second, we wish to consider the efficiency of an algorithm independently of the speed of the computer that is going to execute it. This is why f is multiplied by a constant c. The idea is that when we measure the time of the steps of a particular algorithm, we are not sure how long each of them takes. By definition, $g \in O(f)$ means that eventually (namely beyond the point n_0), the growth of g will be at most as much as the growth of $c * f$. This definition also makes it clear that *constant factors* do not change the growth class (or O-class) of a function. Hence $C(n) = n^2$ is in the same growth class as $C(n) = 1/1000000 * n^2$ or $C(n) = 1000000 * n^2$. So we can write $O(n^2) = O(1000000 * n^2) = O(1/1000000 * n^2)$. Typically, however, we choose the *simplest* representative, as we did in the tables above. In this case it is $O(n^2)$.

The various classes we mentioned above are related as follows:

$$O(1) \subseteq O(\log_2 \log_2 n) \subseteq O(\log_2 (n)) \subseteq O(n) \subseteq O(n\log_2 n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n)$$

We only consider the principal growth class, so when adding functions from different growth classes, their sum will always be in the larger growth class. This allows us to simplify terms. For example, the growth class of $C(n) = 500000\log_2 n + 4n^2 + 0.3n + 100$ can be determined as follows. The summand with the largest growth class is $4n^2$ (we say that this is the 'principal sub-term' or 'dominating sub-term' of the function), and we are allowed to drop constant factors, so this function is in the class $O(n^2)$.

When we say that an algorithm 'belongs to' some class $O(f)$, we mean that it is *at most* as fast growing as f. We have seen that 'linear searching' (where one searches in a collection of data items which is unsorted) has linear complexity, i.e. it is in growth class $O(n)$. This holds for the *average case* as well as the *worst case*. The operations needed are comparisons of the item we are searching for with all the items appearing in the data collection. In the worst case, we have to check all n entries until we find the right one, which means we make n comparisons. On average, however, we will only have to check $n/2$ entries until we hit the correct one, leaving us with $n/2$ operations. Both those functions, $C(n) = n$ and $C(n) = n/2$ belong to the same complexity class, namely $O(n)$. However, it would be equally correct to say that the algorithm belongs to $O(n^2)$, since that class contains all of $O(n)$. But this would be *less informative*, and we would not say that an algorithm has quadratic complexity if we know that, in fact, it is linear. Sometimes it is difficult to be sure what the exact complexity is (as is the case with the famous NP = P problem), in which case one might say that an algorithm is 'at most', say, quadratic.

The issue of efficiency and complexity class, and their computation, will be a recurring feature throughout the chapters to come. We shall see that concentrating only on the complexity class, rather than finding exact complexity functions, can render the whole process of considering efficiency much easier. In most cases, we can determine the time complexity by a simple counting of the loops and tree heights. However, we will also see at least one case where that results in an overestimate, and a more exact computation is required.

Chapter 6

Trees

In computer science, a *tree* is a very general and powerful data structure that resembles a real tree. It consists of an ordered set of linked *nodes* in a connected *graph*, in which each node has at most one *parent* node, and zero or more *children* nodes with a specific order.

6.1 General specification of trees

Generally, we can specify a *tree* as consisting of *nodes* (also called *vertices* or *points*) and *edges* (also called *lines*, or, in order to stress the directedness, *arcs*) with a tree-like structure. It is usually easiest to represent trees pictorially, so we shall frequently do that. A simple example is given in Figure 6.1:

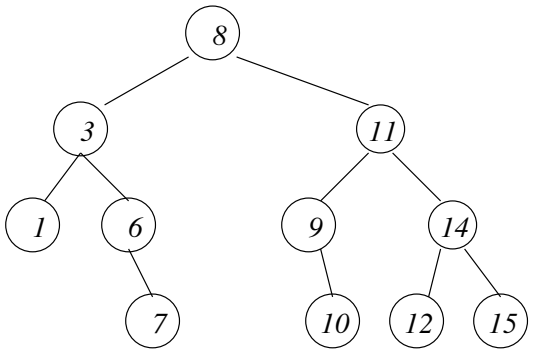


Figure 6.1: Example of a tree.

More formally, a *tree* can be defined as either the empty tree, or a node with a list of successor trees. Nodes are usually, though not always, *labelled* with a data item (such as a *number* or *search key*). We will refer to the label of a node as its *value*. In our examples, we will generally use nodes labelled by integers, but one could just as easily choose something else, e.g. strings of characters.

In order to talk rigorously about trees, it is convenient to have some terminology: There always has to be a unique 'top level' node known as the *root*. In Figure 6.1, this is the node labelled with 8. It is important to note that, in computer science, trees are normally displayed upside-down, with the root forming the top level. Then, given a node, every node on the next level 'down', that is connected to the given node via a branch, is a *child* of that node. In

Figure 6.1, the children of node 8 are nodes 3 and 11. Conversely, the node (there is at most one) connected to the given node (via an edge) on the level above, is its *parent*. For instance, node 11 is the parent of node 9 (and of node 14 as well). Nodes that have the same parent are known as *siblings* – siblings are, by definition, always on the same level.

If a node is the child of a child of ... of a another node then we say that the first node is a *descendent* of the second node. Conversely, the second node is an *ancestor* of the first node. Nodes which do not have any children are known as *leaves* (e.g., the nodes labelled with 1, 7, 10, 12, and 15 in Figure 6.1).

A *path* is a sequence of connected edges from one node to another. Trees have the property that for every node there is a unique path connecting it with the root. In fact, that is another possible definition of a tree. The *depth* or *level* of a node is given by the length of this path. Hence the root has level 0, its children have level 1, and so on. The maximal length of a path in a tree is also called the *height* of the tree. A path of maximal length always goes from the root to a leaf. The *size* of a tree is given by the number of nodes it contains. We shall normally assume that every tree is finite, though generally that need not be the case. The tree in Figure 6.1 has height 3 and size 11. A tree consisting of just of one node has height 0 and size 1. The empty tree obviously has size 0 and is defined (conveniently, though somewhat artificially) to have height -1.

Like most data structures, we need a set of *primitive operators* (constructors, selectors and conditions) to *build* and manipulate the trees. The details of those depend on the type and purpose of the tree. We will now look at some particularly useful types of tree.

6.2 Quad-trees

A *quadtree* is a particular type of tree in which each leaf-node is labelled by a value and each non-leaf node has exactly four children. It is used most often to partition a two dimensional space (e.g., a pixelated image) by recursively dividing it into four quadrants.

Formally, a quadtree can be defined to be either a single node with a number or value (e.g., in the range 0 to 255), or a node without a value but with four quadtree children: `lu`, `ll`, `ru`, and `rl`. It can thus be defined "*inductively*" by the following rules:

Definition. A *quad tree* is either

(Rule 1) a root node with a value, or

(Rule 2) a root node without a value and four quad tree children: `lu`, `ll`, `ru`, and `rl`.

in which Rule 1 is the "*base case*" and Rule 2 is the "*induction step*".

We say that a quadtree is *primitive* if it consists of a single node/number, and that can be tested by the corresponding *condition*:

- `isValue(qt)`, which returns true if quad-tree `qt` is a single node.

To *build* a quad-tree we have two *constructors*:

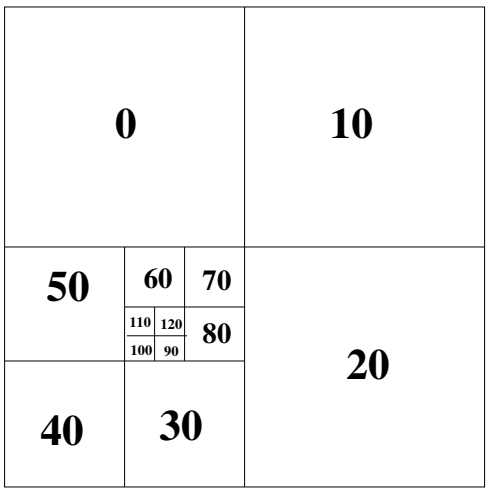
- `baseQT(value)`, which returns a single node quad-tree with label `value`.
- `makeQT(luqt, ruqt, llqt, rlqt)`, which builds a quad-tree from four constituent quad-trees `luqt`, `llqt`, `ruqt`, `rlqt`.

Then to extract components from a quad-tree we have four *selectors*:

- `lu(qt)`, which returns the left-upper quad-tree.
- `ru(qt)`, which returns the right-upper quad-tree.
- `ll(qt)`, which returns the left-lower quad-tree.
- `rl(qt)`, which returns the right-lower quad-tree.

which can be applied whenever `isValue(qt)` is false. For cases when `isValue(qt)` is true, we could define an operator `value(qt)` that returns the value, but conventionally we simply say that `qt` itself is the required value.

Quad-trees of this type are most commonly used to store grey-value pictures (with 0 representing black and 255 white). A simple example would be:



We can then create algorithms using the operators to perform useful manipulations of the representation. For example, we could rotate a picture `qt` by 180° using:

```
rotate(qt) {
   if ( isValue(qt) )
      return qt
   else return makeQT( rotate(rl(qt)), rotate(ll(qt)),
                       rotate(ru(qt)), rotate(lu(qt)) )
}
```

or we could compute average values by recursively averaging the constituent sub-trees.

There exist numerous variations of this general idea, such coloured quadtrees which store value-triples that represent colours rather than grey-scale, and *edge quad-trees* which store lines and allow curves to be represented with arbitrary precision.

6.3 Binary trees

Binary trees are the most common type of tree used in computer science. A *binary tree* is a tree in which every node has at most two children, and can be defined "inductively" by the following rules:

Definition. A *binary tree* is either

(Rule 1) the empty tree `EmptyTree`, or

(Rule 2) it consists of a node and two binary trees, the *left subtree* and *right subtree*.

Again, Rule 1 is the "*base case*" and Rule 2 is the "*induction step*". This definition may appear circular, but actually it is not, because the subtrees are always simpler than the original one, and we eventually end up with an empty tree.

You can imagine that the (infinite) collection of (finite) trees is created in a sequence of days. Day 0 is when you "get off the ground" by applying Rule 1 to get the empty tree. On later days, you are allowed to use any trees that you have created on earlier days to construct new trees using Rule 2. Thus, for example, on day 1 you can create exactly trees that have a root with a value, but no children (i.e. both the left and right subtrees are the empty tree, created at day 0). On day 2 you can use a new node with value, with the empty tree and/or the one-node tree, to create more trees. Thus, binary trees are the objects created by the above two rules in a finite number of steps. The height of a tree, defined above, is the number of days it takes to create it using the above two rules, where we assume that only one rule is used per day, as we have just discussed. (Exercise: work out the sequence of steps needed to create the tree in Figure 6.1 and hence prove that it is in fact a binary tree.)

6.4 Primitive operations on binary trees

The *primitive operators* for binary trees are fairly obvious. We have two *constructors* which are used to *build* trees:

- `EmptyTree`, which returns an empty tree,
- `MakeTree(v, l, r)`, which builds a binary tree from a root node with label `v` and two constituent binary trees `l` and `r`,

a *condition* to test whether a tree is empty:

- `isEmpty(t)`, which returns true if tree `t` is the `EmptyTree`,

and three *selectors* to break a non-empty tree into its constituent parts:

- `root(t)`, which returns the value of the root node of binary tree `t`,
- `left(t)`, which returns the left sub-tree of binary tree `t`,
- `right(t)`, which returns the right sub-tree of binary tree `t`.

These operators can be used to create all the algorithms we might need for manipulating binary trees.

For convenience though, it is often a good idea to define *derived operators* that allow us to write simpler, more readable algorithms. For example, we can define a derived constructor:

- `Leaf(v)` = `MakeTree(v, EmptyTree, EmptyTree)`

that creates a tree consisting of a single node with label `v`, which is the root and the unique leaf of the tree at the same time. Then the tree in Figure 6.1 can be constructed as:

```
t = MakeTree(8, MakeTree(3,Leaf(1),MakeTree(6,EmptyTree,Leaf(7))),
  MakeTree(11,MakeTree(9,EmptyTree,Leaf(10)),MakeTree(14,Leaf(12),Leaf(15))))
```

which is much simpler than the construction using the primitive operators:

```
t = MakeTree(8, MakeTree(3,MakeTree(1,EmptyTree,EmptyTree),
  MakeTree(6,EmptyTree,MakeTree(7,EmptyTree,EmptyTree))),
    MakeTree(11,MakeTree(9,EmptyTree,MakeTree(10,EmptyTree,EmptyTree)),
      MakeTree(14,MakeTree(12,EmptyTree,EmptyTree),
        MakeTree(15,EmptyTree,EmptyTree))))
```

Note that the selectors can only operate on non-empty trees. For example, for the tree `t` defined above we have

```
root(left(left(t)) = 1,
```

but the expression

```
root(left(left(left(t))))
```

does not make sense because

```
left(left(left(t))) = EmptyTree
```

and the empty tree does not have a root. In a language such as *Java*, this would typically raise an exception. In a language such as *C*, this would cause an unpredictable behaviour, but if you are lucky, a core dump will be produced and the program will be aborted with no further harm. When writing algorithms, we need to check the selector arguments using `isEmpty(t)` before allowing their use.

The following equations should be obvious from the primitive operator definitions:

```
root(MakeTree(v,l,r)) = v
left(MakeTree(v,l,r)) = l
right(MakeTree(v,l,r)) = r
isEmpty(EmptyTree) = true
isEmpty(MakeTree(v,l,r)) = false
```

The following makes sense only under the assumption that `t` is a non-empty tree:

```
MakeTree(root(t),left(t),right(t)) = t
```

It just says that if we break apart a non-empty tree and use the pieces to build a new tree, then we get an identical tree back.

It is worth emphasizing that the above specifications of quad-trees and binary trees are further examples of *abstract data types*: Data types for which we exhibit the constructors and destructors and describe their behaviour (using equations such as defined above for lists, stacks, queues, quad-trees and binary trees), but for which we explicitly hide the implementational details. The concrete data type used in an implementation is called a *data structure*. For example, the usual data structures used to implement the list and tree data types are records and pointers – but other implementations are possible.

The important advantage of abstract data types is that we can develop algorithms without having to worry about the details of the representation of the data or the implementation. Of course, everything will ultimately be represented as sequences of bits in a computer, but we clearly do not generally want to have to think in such low level terms.

6.5 The height of a binary tree

Binary trees don't have a simple relation between their size n and height h. The maximum height of a binary tree with n nodes is $(n-1)$, which happens when all non-leaf nodes have precisely one child, forming something that looks like a chain. On the other hand, suppose we have n nodes and want to build from them a binary tree with minimal height. We can achieve this by 'filling' each successive level in turn, starting from the root. It does not matter where we place the nodes on the last (bottom) level of the tree, as long as we don't start adding to the next level before the previous level is full. Terminology varies, but we shall say that such trees are *perfectly balanced* or *height balanced*, and we shall see later why they are optimal for many of our purposes. Basically, if done appropriately, many important tree-based operations (such as searching) take as many steps as the height of the tree, so minimizing the height minimizes the time needed to perform those operations.

We can easily determine the maximum number of nodes that can fit into a binary tree of a given height h. Calling this size function $s(h)$, we obtain:

h	$s(h)$
0	1
1	3
2	7
3	15

In fact, it seems fairly obvious that $s(h) = 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$. This hypothesis can be proved by induction using the definition of a binary tree as follows:

(a) The base case applies to the empty tree that has height $h = -1$, which is consistent with $s(-1) = 2^{-1+1} - 1 = 2^0 - 1 = 1 - 1 = 0$ nodes being stored.

(b) Then for the induction step, a tree of height $h+1$ has a root node plus two subtrees of height h. By the induction hypothesis, each subtree can store $s(h) = 2^{h+1} - 1$ nodes, so the total number of nodes that can fit in a height $h+1$ tree is $1 + 2 \times (2^{h+1} - 1) = 1 + 2^{h+2} - 2 = 2^{(h+1)+1} - 1 = s(h+1)$. It follows that if $s(h)$ is correct for the empty tree, which it was shown to be in the base case above, then it is correct for all h.

An obvious potential problem with any *proof by induction* like this, however, is the need to identify an induction hypothesis to start with, and that is not always easy.

Another way to proceed here would be to simply sum the series $s(h) = 1 + 2 + 4 + \cdots + 2^h$ algebraically to get the answer. Sometimes, however, the relevant series is too complicated to sum easily. An alternative is to try to identify two different expressions for $s(h+1)$ as a function of $s(h)$, and solve them for $s(h)$. Here, since level h of a tree clearly has 2^h nodes, we can explicitly add in the 2^{h+1} nodes of the last level of the height $h+1$ tree to give

$$s(h+1) = s(h) + 2^{h+1}$$

Also, since a height $h+1$ tree is made up of a root node plus two trees of height h

$$s(h+1) = 1 + 2s(h)$$

Then subtracting the second equation from the first gives

$$s(h) = 2^{h+1} - 1$$

which is the required answer. From this we can get an expression for h

$$h = \log_2 (s + 1) - 1 \approx \log_2 s$$

in which the approximation is valid for large s.

Hence a perfectly balanced tree consisting of n nodes has height approximately $\log_2 n$. This is good, because $\log_2 n$ is very small, even for relatively large n:

n	$\log_2 n$
2	1
32	5
1,024	10
1,048,576	20

We shall see later how we can use binary trees to hold data in such a way that any search has at most as many steps as the height of the tree. Therefore, for perfectly balanced trees we can reduce the search time considerably as the table demonstrates. However, it is not always easy to create perfectly balanced trees, as we shall also see later.

6.6 The size of a binary tree

Usually a binary tree will not be perfectly balanced, so we will need an algorithm to determine its *size*, i.e. the number of nodes it contains.

This is easy if we use *recursion*. The terminating case is very simple: the empty tree has size 0. Otherwise, any binary tree will always be assembled from a root node, a left sub-tree `l`, and a right sub-tree `r`, and its size will be the sum of the sizes of its components, i.e. 1 for the root, plus the size of `l`, plus the size of `r`. We have already defined the primitive operator `isEmpty(t)` to check whether a binary tree `t` is empty, and the selectors `left(t)` and `right(t)` which return the left and right sub-trees of binary tree `t`. Thus we can easily define the procedure `size(t)`, which takes a binary tree `t` and returns its size, as follows:

```
size(t) {
   if ( isEmpty(t) )
      return 0
   else return (1 + size(left(t)) + size(right(t)))
}
```

This recursively processes the whole tree, and we know it will terminate because the trees being processed get smaller with each call, and will eventually reach an empty tree which returns a simple value.

6.7 Implementation of trees

The natural way to *implement* trees is in terms of *records* and *pointers*, in a similar way to how linked lists were represented as two-cells consisting of a pointer to a list element and a pointer to the next two-cell. Obviously, the details will depend on how many children each node can have, but trees can generally be represented as data structures consisting of a pointer to the root-node content (if any) and pointers to the children sub-trees. The inductive definition

of trees then allows recursive algorithms on trees to operate efficiently by simply passing the pointer to the relevant root-node, rather than having to pass complete copies of whole trees. How data structures and pointers are implemented in different programming languages will vary, of course, but the general idea is the same.

A binary tree can be implemented as a data record for each node consisting simply of the node value and two pointers to the children nodes. Then `MakeTree` simply creates a new data record of that form, and `root`, `left` and `right` simply read out the relevant contents of the record. The absence of a child node can be simply represented by a Null Pointer.

6.8 Recursive algorithms

Some people have difficulties with *recursion*. A source of confusion is that it appears that "the algorithm calls itself" and it might therefore get confused about what it is operating on. This way of putting things, although suggestive, can be misleading. The algorithm itself is a passive entity, which actually cannot do anything at all, let alone call itself. What happens is that a *processor* (which can be a machine or a person) *executes* the algorithm. So what goes on when a processor executes a recursive algorithm such as the `size(t)` algorithm above? An easy way of understanding this is to imagine that whenever a recursive call is encountered, new processors are given the task with a copy of the same algorithm.

For example, suppose that John (the first processor in this task) wants to compute the size of a given tree `t` using the above recursive algorithm. Then, according to the above algorithm, John first checks whether it is empty. If it is, he simply returns zero and finishes his computation. If it isn't empty, then his tree `t` must have left and right subtrees `l` and `r` (which may, or may not, be empty) and he can extract them using the selectors `left(t)` and `right(t)`. He can then ask two of his students, say Steve and Mary, to execute the same algorithm, but for the trees `l` and `r`. When they finish, say returning results m and n respectively, he computes and returns $1+m+n$, because his tree has a root node in addition to the left and right sub-trees. If Steve and Mary aren't given empty trees, they will themselves have to delegate executions of the same algorithm, with their sub-trees, to other people. Thus, the algorithm is not calling itself. What happens, is that there are many people running their own copies of the same algorithm on different trees.

In this example, in order to make things understandable, we assumed that each person executes a single copy of the algorithm. However, the same processor, with some difficulty, can impersonate several processors, in such a way that it achieves the same result as the execution involving many processors. This is achieved via the use of a *stack* that keeps track of the various positions of the same algorithm that are currently being executed – but this knowledge is not needed for our purposes.

Note that there is nothing to stop us keeping count of the recursions by passing integers along with any data structures being operated on, for example:

```
function(int n, tree t) {
   // terminating condition and return
      .
   // procedure details
      .
   return function(n-1, t2)
}
```

so we can do something `n` times, or look for the `n`th item, etc. The classic example is the recursive factorial function:

```
factorial(int n) {
   if ( n == 0 ) return 1
   return n*factorial(n-1)
}
```

Another example, with two termination or base-case conditions, is a direct implementation of the recursive definition of *Fibonacci numbers* (see Appendix A.5):

```
F(int n) {
   if ( n == 0 ) return 0
   if ( n == 1 ) return 1
   return F(n-1) + F(n-2)
}
```

though this is an extremely inefficient algorithm for computing these numbers. Exercise: Show that the time complexity of this algorithm is $O(2^n)$, and that there exists a straightforward iterative algorithm that has only $O(n)$ time complexity. Is it possible to create an $O(n)$ recursive algorithm to compute these numbers?

In most cases, however, we won't need to worry about counters, because the relevant data structure has a natural end point condition, such as `isEmpty(x)`, that will bring the recursion to an end.

Chapter 7

Binary Search Trees

We now look at Binary Search Trees, which are a particular type of binary tree that provide an efficient way of *storing* data that allows particular items to be found as quickly as possible. Then we consider further elaborations of these trees, namely AVL trees and B-trees, which operate more efficiently at the expense of requiring more sophisticated algorithms.

7.1 Searching with arrays or lists

As we have already seen in Chapter 4, many computer science applications involve *searching* for a particular item in a collection of data. If the data is stored as an unsorted array or list, then to find the item in question, one obviously has to check each entry in turn until the correct one is found, or the collection is exhausted. On average, if there are n items, this will take $n/2$ checks, and in the worst case, all n items will have to be checked. If the collection is large, such as all items accessible via the internet, that will take too much time. We also saw that if the items are sorted before storing in an array, one can perform binary search which only requires $\log_2 n$ checks in the average and worst cases. However, that involves an overhead of sorting the array in the first place, or maintaining a sorted array if items are inserted or deleted over time. The idea here is that, with the help of binary trees, we can speed up the storing and search process without needing to maintain a sorted array.

7.2 Search keys

If the items to be searched are labelled by comparable *keys*, one can order them and store them in such a way that they are *sorted* already. Being 'sorted' may mean different things for different keys, and which key to choose is an important design decision.

In our examples, the search keys will, for simplicity, usually be integer numbers (such as student ID numbers), but other choices occur in practice. For example, the comparable keys could be words. In that case, comparability usually refers to the *alphabetical* order. If `w` and `t` are words, we write `w < t` to mean that `w` precedes `t` in the alphabetical order. If `w` = *bed* and `t` = *sky* then the relation `w < t` holds, but this is not the case if `w` = *bed* and `t` = *abacus*. A classic example of a collection to be searched is a dictionary. Each entry of the dictionary is a pair consisting of a word and a definition. The definition is a sequence of words and punctuation symbols. The search key, in this example, is the word (to which a definition is attached in the dictionary entry). Thus, *abstractly*, a dictionary is a sequence of

entries, where an entry is a pair consisting of a word and its definition. This is what matters from the point of view of the search algorithms we are going to consider. In what follows, we shall concentrate on the search keys, but should always bear in mind that there is usually a more substantial data entry associated with it.

Notice the use of the word "abstract" here. What we mean is that we abstract or remove any details that are irrelevant from the point of view of the algorithms. For example, a dictionary usually comes in the form of a book, which is a sequence of pages – but for us, the distribution of dictionary entries into pages is an accidental feature of the dictionary. All that matters for us is that the dictionary is a sequence of entries. So "abstraction" means "getting rid of irrelevant details". For our purposes, only the search key is important, so we will ignore the fact that the entries of the collection will typically be more complex objects (as in the example of a dictionary or a phone book).

Note that we should always employ the data structure to hold the items which performs best for the typical application. There is no easy answer as to what the best choice is – the particular circumstances have to be inspected, and a decision has to be made based on that. However, for many applications, the kind of *binary trees* we studied in the last chapter are particularly useful here.

7.3 Binary search trees

The solution to our search problem is to store the collection of data to be searched using a binary tree in such a way that searching for a particular item takes minimal effort. The underlying idea is simple: At each tree node, we want the value of that node to either tell us that we have found the required item, or tell us which of its two subtrees we should search for it in. For the moment, we shall assume that all the items in the data collection are distinct, with different search keys, so each possible node value occurs at most once, but we shall see later that it is easy to relax this assumption. Hence we define:

Definition. A *binary search tree* is a binary tree that is either empty or satisfies the following conditions:

- All values occurring in the left subtree are smaller than that of the root.
- All values occurring in the right subtree are larger than that of the root.
- The left and right subtrees are themselves binary search trees.

So this is just a particular type of binary tree, with node values that are the search keys. This means we can *inherit* many of the operators and algorithms we defined for general binary trees. In particular, the primitive operators `MakeTree(v,l,r)`, `root(t)`, `left(t)`, `right(t)` and `isEmpty(t)` are the same – we just have to maintain the additional node value ordering.

7.4 Building binary search trees

When building a binary search tree, one naturally starts with the root and then adds further new nodes as needed. So, to insert a new value v, the following cases arise:

- If the given tree is empty, then simply assign the new value v to the root, and leave the left and right subtrees empty.

- If the given tree is non-empty, then insert a node with value v as follows:
  - If v is smaller than the value of the root: insert v into the left sub-tree.
  - If v is larger than the value of the root: insert v into the right sub-tree.
  - If v is equal to the value of the root: report a violated assumption.

Thus, using the primitive binary tree operators, we have the procedure:

```
insert(v,bst) {
   if ( isEmpty(bst) )
      return MakeTree(v, EmptyTree, EmptyTree)
   elseif ( v < root(bst) )
      return MakeTree(root(bst), insert(v,left(bst)), right(bst))
   elseif ( v > root(bst) )
      return MakeTree(root(bst), left(bst), insert(v,right(bst)))
   else error('Error: violated assumption in procedure insert.')
}
```

which inserts a node with value `v` into an existing binary search tree `bst`. Note that the node added is always a leaf. The resulting tree is once again a binary search tree. This can be proved rigorously via an inductive argument.

Note that this procedure creates a new tree out of a given tree `bst` and new value `v`, with the new value inserted at the right position. The original tree `bst` is not modified, it is merely inspected. However, when the tree represents a large database, it would clearly be more efficient to modify the given tree, rather than to construct a whole new tree. That can easily be done by using *pointers*, similar to the way we set up linked lists. For the moment, though, we shall not concern ourselves with such implementational details.

7.5 Searching a binary search tree

Searching a binary search tree is not dissimilar to the process performed when inserting a new item. We simply have to compare the item being looked for with the root, and then keep 'pushing' the comparison down into the left or right subtree depending on the result of each root comparison, until a match is found or a leaf is reached.

Algorithms can be expressed in many ways. Here is a concise description in words of the search algorithm that we have just outlined:

> In order to search for a value `v` in a binary search tree `t`, proceed as follows. If `t` is empty, then `v` does not occur in `t`, and hence we stop with `false`. Otherwise, if `v` is equal to the root of `t`, then `v` does occur in `t`, and hence we stop returning `true`. If, on the other hand, `v` is smaller than the root, then, by definition of a binary search tree, it is enough to search the left sub-tree of `t`. Hence replace `t` by its left sub-tree and carry on in the same way. Similarly, if `v` is bigger than the root, replace `t` by its right sub-tree and carry on in the same way.

Notice that such a description of an algorithm embodies both the steps that need to be carried out *and* the reason why this gives a correct solution to the problem. This way of describing algorithms is very common when we do not intend to run them on a computer.

When we do want to run them, we need to provide a more precise specification, and would normally write the algorithm in pseudocode, such as the following recursive procedure:

```
isIn(value v, tree t) {
   if ( isEmpty(t) )
      return false
   elseif ( v == root(t) )
      return true
   elseif ( v < root(t) )
      return isIn(v, left(t))
   else
      return isIn(v, right(t))
}
```

Each recursion restricts the search to either the left or right subtree as appropriate, reducing the search tree height by one, so the algorithm is guaranteed to terminate eventually.

In this case, the recursion can easily be transformed into a while-loop:

```
isIn(value v, tree t) {
   while ( (not isEmpty(t)) and (v != root(t)) )
      if (v < root(t) )
         t = left(t)
      else
         t = right(t)
   return ( not isEmpty(t) )
}
```

Here, each iteration of the while-loop restricts the search to either the left or right subtree as appropriate. The only way to leave the loop is to have found the required value, or to only have an empty tree remaining, so the procedure only needs to return whether or not the final tree is empty.

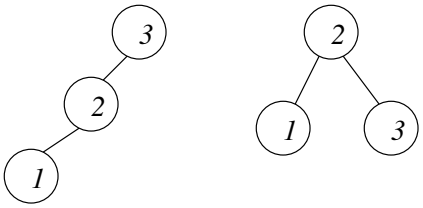
In practice, we often want to have more than a simple `true`/`false` returned. For example, if we are searching for a student ID, we usually want a pointer to the full record for that student, not just a confirmation that they exist. In that case, we could store a record pointer associated with the search key (ID) at each tree node, and return the record pointer or a null pointer, rather than a simple `true` or `false`, when an item is found or not found. Clearly, the basic tree structures we have been discussing can be elaborated in many different ways like this to form whatever data-structure is most appropriate for the problem at hand, but, as noted above, we can abstract out such details for current purposes.

7.6 Time complexity of insertion and search

As always, it is important to understand the time complexity of our algorithms. Both item insertion and search in a binary search tree will take at most as many comparisons as the height of the tree plus one. At worst, this will be the number of nodes in the tree. But how many comparisons are required on average? To answer this question, we need to know the average height of a binary search tree. This can be calculated by taking all possible binary search trees of a given size n and measuring each of their heights, which is by no means an

easy task. The trouble is that there are many ways of building the same binary search tree by successive insertions.

As we have seen above, perfectly balanced trees achieve minimal height for a given number of nodes, and it turns out that the more balanced a tree, the more ways there are of building it. This is demonstrated in the figure below:



The only way of getting the tree on the left hand side is by inserting 3, 2, 1 into the empty tree in that order. The tree on the right, however, can be reached in two ways: Inserting in the order 2, 1, 3 or in the order 2, 3, 1. Ideally, of course, one would only use well-balanced trees to keep the height minimal, but they do not have to be perfectly balanced to perform better than binary search trees without restrictions.

Carrying out exact tree height calculations is not straightforward, so we will not do that here. However, if we assume that all the possible orders in which a set of n nodes might be inserted into a binary search tree are equally likely, then the average height of a binary search tree turns out to be $O(\log_2 n)$. It follows that the average number of comparisons needed to search a binary search tree is $O(\log_2 n)$, which is the same complexity we found for binary search of a sorted array. However, inserting a new node into a binary search tree also depends on the tree height and requires $O(\log_2 n)$ steps, which is better than the $O(n)$ complexity of inserting an item into the appropriate point of a sorted array.

Interestingly, the average height of a binary search tree is quite a bit better than the average height of a general binary tree consisting of the same n nodes that have not been built into a binary search tree. The average height of a general binary tree is actually $O(\sqrt{n})$. The reason for that is that there is a relatively large proportion of high binary trees that are not valid binary search trees.

7.7 Deleting nodes from a binary search tree

Suppose, for some reason, an item needs to be removed or deleted from a binary search tree. It would obviously be rather inefficient if we had to rebuild the remaining search tree again from scratch. For n items that would require n steps of $O(\log_2 n)$ complexity, and hence have overall time complexity of $O(n\log_2 n)$. By comparison, deleting an item from a sorted array would only have time complexity $O(n)$, and we certainly want to do better than that. Instead, we need an algorithm that produces an updated binary search tree more efficiently. This is more complicated than one might assume at first sight, but it turns out that the following algorithm works as desired:

- If the node in question is a leaf, just remove it.
- If only one of the node's subtrees is non-empty, 'move up' the remaining subtree.
- If the node has two non-empty sub-trees, find the 'left-most' node occurring in the right sub-tree (this is the smallest item in the right subtree). Use this node to overwrite the

one that is to be deleted. Replace the left-most node by its right subtree, if this exists; otherwise just delete it.

The last part works because the left-most node in the right sub-tree is guaranteed to be bigger than all nodes in the left sub-tree, smaller than all the other nodes in the right sub-tree, and have no left sub-tree itself. For instance, if we delete the node with value `11` from the tree in Figure 6.1, we get the tree displayed in Figure 7.1.

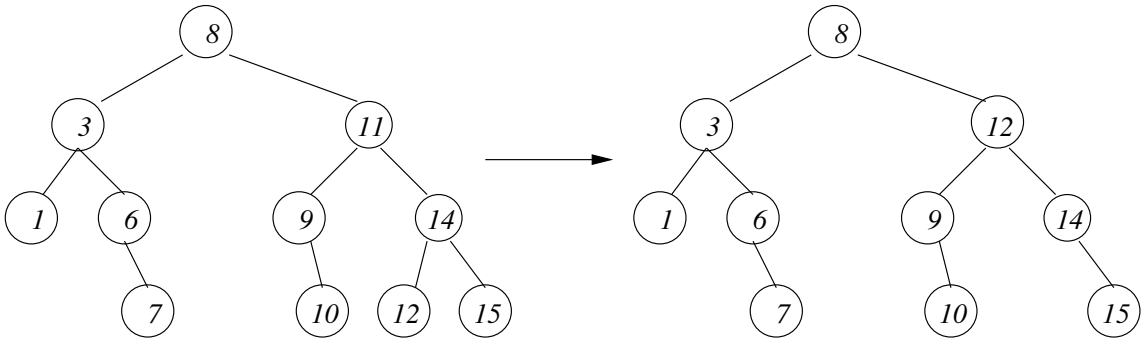


Figure 7.1: Example of node deletion in a binary search tree.

In practice, we need to turn the above algorithm (specified in words) into a more detailed algorithm specified using the primitive binary tree operators:

```
delete(value v, tree t) {
  if ( isEmpty(t) )
     error('Error: given item is not in given tree')
  else
     if ( v < root(t) )    // delete from left sub-tree
        return MakeTree(root(t), delete(v,left(t)), right(t));
     else if ( v > root(t) )     // delete from right sub-tree
        return MakeTree(root(t), left(t), delete(v,right(t)));
     else    // the item v to be deleted is root(t)
        if ( isEmpty(left(t)) )
           return right(t)
        elseif ( isEmpty(right(t)) )
           return left(t)
        else    // difficult case with both subtrees non-empty
           return MakeTree(smallestNode(right(t)), left(t),
                                          removeSmallestNode(right(t))
}
```

If the empty tree condition is met, it means the search item is not in the tree, and an appropriate error message should be returned.

The `delete` procedure uses two sub-algorithms to find and remove the smallest item of a given sub-tree. Since the relevant sub-trees will always be non-empty, these sub-algorithms can be written with that *precondition*. However, it is always the responsibility of the programmer to ensure that any preconditions are met whenever a given procedure is used, so it is important to say explicitly what the preconditions are. It is often safest to start each procedure with a

check to determine whether the preconditions are satisfied, with an appropriate *error message* produced when they are not, but that may have a significant time cost if the procedure is called many times. First, to find the smallest node, we have:

```
smallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return root(t)
  else
     return smallestNode(left(t));
}
```

which uses the fact that, by the definition of a binary search tree, the smallest node of `t` is the left-most node. It recursively looks in the left sub-tree till it reaches an empty tree, at which point it can return the root. The second sub-algorithm uses the same idea:

```
removeSmallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return right(t)
  else
     return MakeTree(root(t), removeSmallestNode(left(t)), right(t))
}
```

except that the remaining tree is returned rather than the smallest node.

These procedures are further examples of recursive algorithms. In each case, the recursion is guaranteed to terminate, because every recursive call involves a smaller tree, which means that we will eventually find what we are looking for or reach an empty tree.

It is clear from the algorithm that the deletion of a node requires the same number of steps as searching for a node, or inserting a new node, i.e. the average height of the binary search tree, or $O(\log_2 n)$ where n is the total number of nodes on the tree.

7.8 Checking whether a binary tree is a binary search tree

Building and using binary search trees as discussed above is usually enough. However, another thing we sometimes need to do is *check* whether or not a given binary tree is a binary search tree, so we need an algorithm to do that. We know that an empty tree is a (trivial) binary search tree, and also that all nodes in the left sub-tree must be smaller than the root and themselves form a binary search tree, and all nodes in the right sub-tree must be greater than the root and themselves form a binary search tree. Thus the obvious algorithm is:

```
isbst(tree t) {
   if ( isEmpty(t) )
      return true
   else
      return ( allsmaller(left(t),root(t)) and isbst(left(t))
                  and allbigger(right(t),root(t)) and isbst(right(t)) )
}
```

```
allsmaller(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) < v) and allsmaller(left(t),v)
                                          and allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) > v) and allbigger(left(t),v)
                                          and allbigger(right(t),v) )
}
```

However, the simplest or most obvious algorithm is not always the most efficient. Exercise: identify what is inefficient about this algorithm, and formulate a more efficient algorithm.

7.9 Sorting using binary search trees

Sorting is the process of putting a collection of items in order. We shall formulate and discuss many sorting algorithms later, but we are already able to present one of them.

The node values stored in a binary search tree can be printed in ascending order by recursively printing each left sub-tree, root, and right sub-tree in the right order as follows:

```
printInOrder(tree t) {
   if ( not isEmpty(t) ) {
      printInOrder(left(t))
      print(root(t))
      printInOrder(right(t))
   }
}
```

Then, if the collection of items to be sorted is given as an array `a` of known size `n`, they can be printed in sorted order by the algorithm:

```
sort(array a of size n) {
   t = EmptyTree
   for i = 0,1,...,n-1
      t = insert(a[i],t)
   printInOrder(t)
}
```

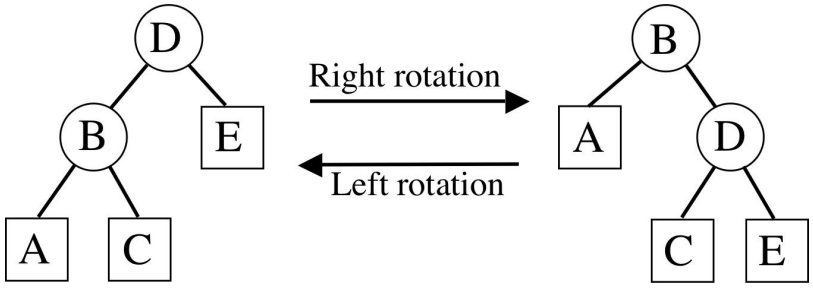
which starts with an empty tree, inserts all the items into it using `insert`(`v`, `t`) to give a binary search tree, and then prints them in order using `printInOrder`(`t`). Exercise: modify this algorithm so that instead of printing the sorted values, they are put back into the original array in ascending order.

7.10 Balancing binary search trees

If the items are added to a binary search tree in random order, the tree tends to be fairly well balanced with height not much more than $\log_2 n$. However, there are many situations where the added items are not in random order, such as when adding new student IDs. In the extreme case of the new items being added in ascending order, the tree will be one long branch off to the right, with height $n \gg \log_2 n$.

If all the items to be inserted into a binary search tree are already sorted, it is straightforward to build a perfectly balanced binary tree from them. One simply has to recursively build a binary tree with the middle (i.e., median) item as the root, the left subtree made up of the smaller items, and the right subtree made up of the larger items. This idea can be used to *rebalance* any existing binary search tree, because the existing tree can easily be output into a sorted array as discussed in Section 7.9. Exercise: Write an algorithm that rebalances a binary search tree in this way, and work out its time complexity.

Another way to avoid unbalanced binary search trees is to *rebalance* them from time to time using *tree rotations*. Such tree rotations are best understood as follows: Any binary search tree containing at least two nodes can clearly be drawn in one of the two forms:



where B and D are the required two nodes to be rotated, and A, C and E are binary search sub-trees (any of which may be empty). The two forms are related by left and right tree rotations which clearly preserve the binary search tree property. In this case, any nodes in sub-tree A would be shifted up the tree by a right rotation, and any nodes in sub-tree E would be shifted up the tree by a left rotation. For example, if the left form had A consisting of two nodes, and C and E consisting of one node, the height of the tree would be reduced by one and become *perfectly balanced* by a right tree rotation.

Typically, such tree rotations would need to be applied to many different sub-trees of a full tree to make it perfectly balanced. For example, if the left form had C consisting of two nodes, and A and E consisting of one node, the tree would be balanced by first performing a left rotation of the A-B-C sub-tree, followed by a right rotation of the whole tree. In practice, finding suitable sequences of appropriate tree rotations to rebalance an arbitrary binary search tree is not straightforward, but it is possible to formulate systematic balancing algorithms that are more efficient than outputting the whole tree and rebuilding it.

7.11 Self-balancing AVL trees

Self-balancing binary search trees avoid the problem of unbalanced trees by automatically *rebalancing* the tree throughout the insertion process to keep the height close to $\log_2 n$ at each stage. Obviously, there will be a cost involved in such rebalancing, and there will be a

trade-off between the time involved in rebalancing and the time saved by the reduced height of the tree, but generally it is worthwhile.

The earliest type of self-balancing binary search tree was the *AVL tree* (named after its inventors G.M. Adelson-Velskii and E.M. Landis). These maintain the difference in heights of the two sub-trees of all nodes to be at most one. This requires the tree to be periodically rebalanced by performing one or more *tree rotations* as discussed above, but the complexity of insertion, deletion and search remain at $O(\log_2 n)$.

The general idea is to keep track of the *balance factor* for each node, which is the height of the left sub-tree minus the height of the right sub-tree. By definition, all the nodes in an AVL-tree will have a balance factor in the integer range $[-1, 1]$. However, insertion or deletion of a node could leave that in the wider range $[-2, 2]$ requiring a tree-rotation to bring it back into AVL form. Exercise: Find some suitable algorithms for performing efficient AVL tree rotations. Compare them with other self-balancing approaches such as *red-black trees*.

7.12 B-trees

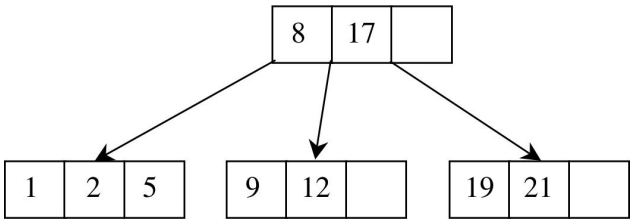
A *B-tree* is a generalization of a self-balancing binary search tree in which each node can hold more than one search key and have more than two children. The structure is designed to allow more efficient self-balancing, and offers particular advantages when the node data needs to be kept in external storage such as disk drives. The standard (Knuth) definition is:

Definition. A *B-tree* of order m is a tree which satisfies the following conditions:

- Every node has at most m children.
- Every non-leaf node (except the root node) has at least $m/2$ children.
- The root node, if it is not a leaf node, has at least two children.
- A non-leaf node with c children contains $c-1$ search keys which act as separation values to divide its sub-trees.
- All leaf nodes appear in the same level, and carry information.

There appears to be no definitive answer to the question of what the "B" in "B-Tree" stands for. It is certainly not "Binary", but it could equally well be "balanced", "broad" or "bushy", or even "Boeing" because they were invented by people at Boeing Research Labs.

The standard representation of simple order 4 example with 9 search keys would be:



The search keys held in each node are ordered (e.g., 1, 2, 5 in the example), and the non-leaf node's search keys (i.e., the items 8 and 17 in the example) act as separation values to divide

the contents of its sub-trees in much the same way that a node's value in a binary search tree separates the values held in its two sub-trees. For example, if a node has 3 child nodes (or sub-trees) then it must have 2 separation values $s1$ and $s2$. All values in the leftmost subtree will be less than $s1$, all values in the middle subtree will be between $s1$ and $s2$, and all values in the rightmost subtree will be greater than $s2$. That allows insertion and searching to proceed from the root down in a similar way to binary search trees.

The restriction on the number of children to lie between $m/2$ and m means that the best case height of an order m B-tree containing n search keys is $\log_m n$ and the worst case height is $\log_{m/2} n$. Clearly the costs of insertion, deletion and searching will all be proportional to the tree height, as in a binary search tree, which makes them very efficient. The requirement that all the leaf nodes are at the same level means that B-trees are always balanced and thus have minimal height, though *rebalancing* will often be required to restore that property after insertions and deletions.

The *order* of a B-tree is typically chosen to optimize a particular application and implementation. To maintain the conditions of the B-tree definition, non-leaf nodes often have to be split or joined when new items are inserted into or deleted from the tree (which is why there is a factor of two between the minimum and maximum number of children), and *rebalancing* is often required. This renders the insertion and deletion algorithms somewhat more complicated than for binary search trees. An advantage of B-trees over self balancing binary search trees, however, is that the range of child nodes means that rebalancing is required less frequently. A disadvantage is that there may be more space wastage because nodes will rarely be completely full. There is also the cost of keeping the items within each node ordered, and having to search among them, but for reasonably small orders m, that cost is low. Exercise: find some suitable insertion, deletion and rebalancing algorithms for B-trees.

Chapter 8

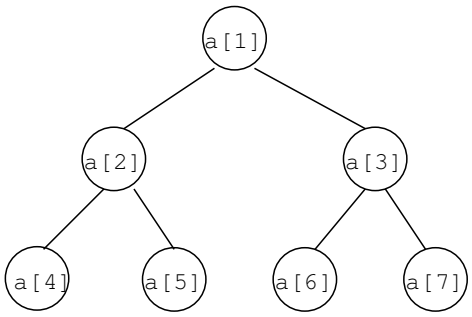
Priority Queues and Heap Trees

8.1 Trees stored in arrays

It was noted earlier that binary trees can be *stored* with the help of *pointer*-like structures, in which each item contains references to its children. If the tree in question is a *complete* binary tree, there is a useful *array* based alternative.

Definition. A binary tree is *complete* if every level, except possibly the last, is completely filled, and all the leaves on the last level are placed as far to the left as possible.

Intuitively, a complete binary tree is one that can be obtained by filling the nodes starting with the root, and then each next level in turn, always from the left, until one runs out of nodes. Complete binary trees always have minimal height for their size n, namely $\log_2 n$, and are always *perfectly balanced* (but not every perfectly balanced tree is complete in the sense of the above definition). Moreover, and more importantly, it is possible for them to be stored straightforwardly in arrays, top-to-bottom left-to-right, as in the following example:



For complete binary trees, such arrays provide very tight representations.

Notice that this time we have chosen to start the array with index 1 rather than 0. This has several computational advantages. The nodes on level i then have indices $2^i, \cdots, 2^{i+1}-1$. The level of a node with index i is $\lfloor \log_2 i \rfloor$, that is, $\log_2 i$ rounded down. The children of a node with index i, if they exist, have indices $2i$ and $2i+1$. The parent of a child with index i has index $i/2$ (using integer division). This allows the following simple algorithms:

```
boolean isRoot(int i) {
   return i == 1
}
```

```
int level(int i) {
   return log(i)
}

int parent(int i) {
   return i / 2
}

int left(int i) {
   return 2 * i
}

int right(int i) {
   return 2 * i + 1
}
```

which make the processing of these trees much easier.

This way of storing a binary tree as an array, however, will not be efficient if the tree is not complete, because it involves reserving space in the array for every possible node in the tree. Since keeping binary search trees balanced is a difficult problem, it is therefore not really a viable option to adapt the algorithms for binary search trees to work with them stored as arrays. Array-based representations will also be inefficient for binary search trees because node insertion or deletion will usually involve shifting large portions of the array. However, we shall now see that there is another kind of binary tree for which array-based representations allow very efficient processing.

8.2 Priority queues and binary heap trees

While most queues in every-day life operate on a first come, first served basis, it is sometimes important to be able to assign a *priority* to the items in the queue, and always serve the item with the highest priority next. An example of this would be in a hospital casualty department, where life-threatening injuries need to be treated first. The structure of a complete binary tree in array form is particularly useful for representing such *priority queues*.

It turns out that these queues can be implemented efficiently by a particular type of complete binary tree known as a *binary heap tree*. The idea is that the node labels, which were the search keys when talking about binary search trees, are now numbers representing the priority of each item in question (with higher numbers meaning a higher priority in our examples). With heap trees, it is possible to insert and delete elements efficiently without having to keep the whole tree sorted like a binary search tree. This is because we only ever want to remove one element at a time, namely the one with the highest priority present, and the idea is that the highest priority item will always be found at the root of the tree.

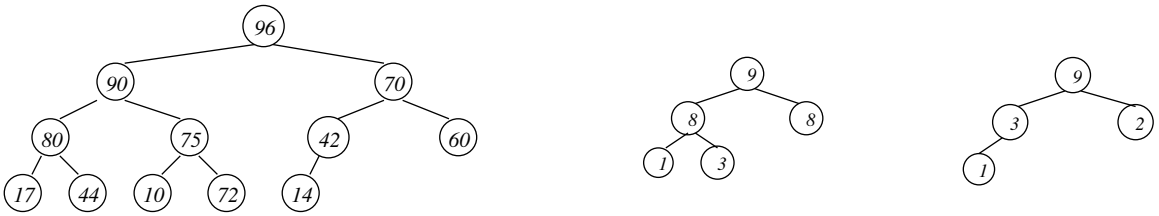
Definition. A *binary heap tree* is a complete binary tree which is either empty or satisfies the following conditions:

- The priority of the root is higher than (or equal to) that of its children.
- The left and right subtrees of the root are heap trees.

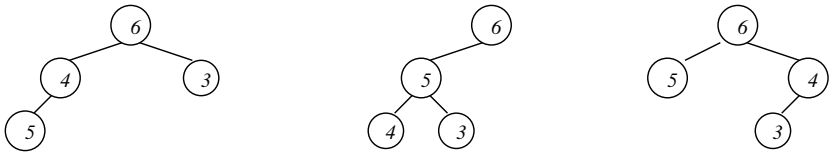
Alternatively, one could define a heap tree as a complete binary tree such that the priority of every node is higher than (or equal to) that of all its descendants. Or, as a complete binary tree for which the priorities become smaller along every path down through the tree.

The most obvious difference between a binary heap tree and a binary search trees is that the biggest number now occurs at the root rather than at the right-most node. Secondly, whereas with binary search trees, the left and right sub-trees connected to a given parent node play very different rôles, they are interchangeable in binary heap trees.

Three examples of binary trees that are valid heap trees are:



and three which are not valid heap trees are:



the first because $5 > 4$ violates the required priority ordering, the second because it is not perfectly balanced and hence not complete, and the third because it is not complete due to the node on the last level not being as far to the left as possible.

8.3 Basic operations on binary heap trees

In order to develop algorithms using an array representation, we need to allocate memory and keep track of the largest position that has been filled so far, which is the same as the current number of nodes in the heap tree. This will involve something like:

```
int MAX = 100    // Maximum number of nodes allowed
int heap[MAX+1]  // Stores priority values of nodes of heap tree
int n = 0        // Largest position that has been filled so far
```

For heap trees to be a useful representation of priority queues, we must be able to *insert* new nodes (or customers) with a given priority, *delete* unwanted nodes, and identify and remove the top-priority node, i.e. the *root* (that is, 'serve' the highest priority customer). We also need to be able to determine when the queue/tree is empty. Thus, assuming the priorities are given by integers, we need a constructor, mutators/selectors, and a condition:

```
insert(int p, array heap, int n)
delete(int i, array heap, int n)
int root(array heap, int n)
boolean heapEmpty(array heap, int n)
```

Identifying whether the heap tree is empty, and getting the root and last leaf, is easy:

```
boolean heapEmpty(array heap, int n) {
   return n == 0
}

int root(array heap, int n) {
   if ( heapEmpty(heap,n) )
      error('Heap is empty')
   else return heap[1]
}

int lastLeaf(array heap, int n) {
   if ( heapEmpty(heap,n) )
      error('Heap is empty')
   else return heap[n]
}
```

Inserting and deleting heap tree nodes is also straightforward, but not quite so easy.

8.4 Inserting a new heap tree node

Since we always keep track of the last position `n` in the tree which has been filled so far, we can easily insert a new element at position `n + 1`, provided there is still room in the array, and increment `n`. The tree that results will still be a complete binary tree, but the heap tree priority ordering property might have been violated. Hence we may need to '*bubble up*' the new element into a valid position. This can be done easily by comparing its priority with that of its parent, and if the new element has higher priority, then it is exchanged with its parent. We may have to repeat this process, but once we reach a parent that has higher or equal priority, we can stop because we know there can be no lower priority items further up the tree. Hence an algorithm which inserts a new heap tree node with priority `p` is:

```
insert(int p, array heap, int n) {
   if ( n == MAX ) {
      error('Heap is full')
   else {
      heap[n+1] = p
      bubbleUp(n+1,heap,n+1)
   }
 }

bubbleUp(int i, array heap, int n) {
   if ( isRoot(i) )
      return
   elseif ( heap[i] > heap[parent(i)] ) {
      swap heap[i] and heap[parent(i)]
      bubbleUp(parent(i),heap,n)
   }
}
```

Note that this `insert` algorithm does not increment the heap size `n` – that has to be done separately by whatever algorithm calls it.

Inserting a node takes at most $O(\log_2 n)$ steps, because the maximum number of times we may have to 'bubble up' the new element is the height of the tree which is $\log_2 n$.

8.5 Deleting a heap tree node

To use a binary heap tree as a priority queue, we will regularly need to delete the root, i.e. remove the node with the highest priority. We will then be left with something which is not a binary tree at all. However, we can easily make it into a complete binary tree again by taking the node at the 'last' position and using that to fill the new vacancy at the root. However, as with insertion of a new item, the heap tree (priority ordering) property might be violated. In that case, we will need to '*bubble down*' the new root by comparing it with both its children and exchanging it with the largest. This process is then repeated until the new root element has found a valid place. Thus, a suitable algorithm is:

```
deleteRoot(array heap, int n) {
   if ( n < 1 )
      error('Node does not exist')
   else {
      heap[1] = heap[n]
      bubbleDown(1,heap,n-1)
   }
}
```

A similar process can also be applied if we need to delete any other node from the heap tree, but in that case we may need to '*bubble up*' the shifted last node rather than bubble it down. Since the original heap tree is ordered, items will only ever need to be bubbled up or down, never both, so we can simply call both, because neither procedure changes anything if it is not required. Thus, an algorithm which deletes any node `i` from a heap tree is:

```
delete(int i, array heap, int n) {
   if ( n < i )
      error('Node does not exist')
   else {
      heap[i] = heap[n]
      bubbleUp(i,heap,n-1)
      bubbleDown(i,heap,n-1)
   }
}
```

The bubble down process is more difficult to implement than bubble up, because a node may have none, one or two children, and those three cases have to be handled differently. In the case of two children, it is crucial that when both children have higher priority than the given node, it is the highest priority one that is swapped up, or their priority ordering will be violated. Thus we have:

```
bubbleDown(int i, array heap, int n) {
   if ( left(i) > n )                    // no children
      return
   elseif ( right(i) > n )               // only left child
      if ( heap[i] < heap[left(i)] )
         swap heap[i] and heap[left(i)]
   else                                  // two children
      if ( heap[left(i)] > heap[right(i)] and heap[i] < heap[left(i)] ) {
          swap heap[i] and heap[left(i)]
          bubbleDown(left(i),heap,n)
       }
      elseif ( heap[i] < heap[right(i)] ) {
          swap heap[i] and heap[right(i)]
          bubbleDown(right(i),heap,n)
      }
   }
}
```

In the same way that the `insert` algorithm does not increment the heap size, this `delete` algorithm does not decrement the heap size `n` – that has to be done separately by whatever algorithm calls it. Note also that this algorithm does not attempt to be *fair* in the sense that if two or more nodes have the same priority, it is not necessarily the one that has been waiting longest that will be removed first. However, this factor could easily be fixed, if required, by keeping track of arrival times and using that in cases of equal priority.

As with insertion, deletion takes at most $O(\log_2 n)$ steps, because the maximum number of times it may have to bubble down or bubble up the replacement element is the height of the tree which is $\log_2 n$.

8.6 Building a new heap tree from scratch

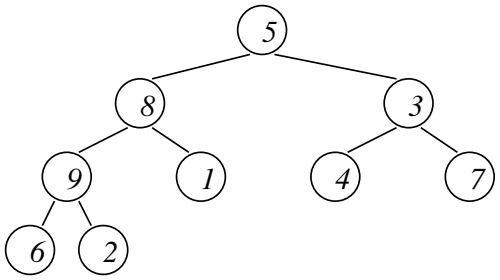
Sometimes one is given a whole set of n new items in one go, and there is a need to *build* a binary heap tree containing them. In other words, we have a set of items that we wish to *heapify*. One obvious possibility would be to insert the n items one by one into a heap tree, starting from an empty tree, using the $O(\log_2 n)$ '*bubble up*' based `insert` algorithm discussed earlier. That would clearly have overall time complexity of $O(n\log_2 n)$.

It turns out, however, that rearranging an array of items into heap tree form can be done more efficiently using '*bubble down*'. First note that, if we have the n items in an array `a` in positions $1, \ldots, n$, then all the items with an index greater than $n/2$ will be leaves, and not need bubbling down. Therefore, if we just bubble down all the non-leaf items `a[n/2],...,a[1]` by exchanging them with the larger of their children until they either are positioned at a leaf, or until their children are both smaller, we obtain a valid heap tree.

Consider a simple example array if items from which a heap tree must be built:

5	8	3	9	1	4	7	6	2

We can start by simply drawing the array as a tree, and see that the last 5 entries (those with indices greater than $9/2 = 4$) are leaves of the tree, as follows:



Then the rearrangement algorithm starts by bubbling down `a[n/2]` = `a[4]` = `9`, which turns out not to be necessary, so the array remains the same. Next `a[3]` = `3` is bubbled down, swapping with `a[7]` = `7`, giving:

5	8	7	9	1	4	3	6	2

Next `a[2]` = `8` is bubbled down, swapping with `a[4]` = `9`, giving:

5	9	7	8	1	4	3	6	2

Finally, `a[1]` = `5` is bubbled down, swapping with `a[2]` = `9`, to give first:

9	5	7	8	1	4	3	6	2

then swapping with `a[4]` = `8` to give:

9	8	7	5	1	4	3	6	2

and finally swapping with `a[8]` = `6` to give:

9	8	7	6	1	4	3	5	2

which has the array rearranged as the required heap tree.

Thus, using the above `bubbleDown` procedure, the algorithm to build a complete binary heap tree from any given array `a` of size `n` is simply:

```
heapify(array a, int n) {
   for( i = n/2 ; i > 0 ; i-- )
      bubbleDown(i,a,n)
}
```

The time complexity of this heap tree creation algorithm might be computed as follows: It potentially bubbles down $\lfloor n/2 \rfloor$ items, namely those with indices $1, \ldots, \lfloor n/2 \rfloor$. The maximum number of bubble down steps for each of those items is the height of the tree, which is $\log_2 n$, and each step involves two comparisons – one to find the highest priority child node, and one to compare the item with that child node. So the total number of comparisons involved is at most $(n/2).\log_2 n.2 = n\log_2 n$, which is the same as we would have by inserting the array items one at a time into an initially empty tree.

In fact, this is a good example of a situation in which a naive counting of loops and tree heights over-estimates the time complexity. This is because the number of bubble down steps

will usually be less than the full height of the tree. In fact, at each level as you go down the tree, there are more nodes, and fewer potential bubble down steps, so the total number of operations will actually be much less than $n\log_2 n$. To be sure of the complexity class, we need to perform a more accurate calculation. At each level i of a tree of height h there will be 2^i nodes, with at most $h-i$ bubble down steps, each with 2 comparisons, so the total number of comparisons for a tree of height h will on average be

$$C(h) = \sum_{i=0}^{h} 2^i(h-i) = 2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

The final sum converges to 2 as h increases (see Appendix A.4), so for large h we have

$$C(h) \approx 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} = 2^h.2 = 2^{h+1} \approx n$$

and the worst case will be no more than twice that. Thus, the total number of operations is $O(2^{h+1}) = O(n)$, meaning that the complexity class of `heapify` is actually $O(n)$, which is better than the $O(n\log_2 n)$ complexity of inserting the items one at a time.

8.7 Merging binary heap trees

Frequently one needs to *merge* two existing priority queues based on binary heap trees into a single priority queue. To achieve this, there are three obvious ways of merging two binary heap trees `s` and `t` of a similar size n into a single binary heap tree:

1. Move all the items from the smaller heap tree one at a time into the larger heap tree using the standard `insert` algorithm. This will involve moving $O(n)$ items, and each of them will need to be bubbled up at cost $O(\log_2 n)$, giving an overall time complexity of $O(n\log_2 n)$.

2. Repeatedly move the last items from one heap tree to the other using the standard `insert` algorithm, until the new binary tree `makeTree(0,t,s)` is complete. Then move the last item of the new tree to replace the dummy root "0", and bubble down that new root. How this is best done will depend on the sizes of the two trees, so this algorithm is not totally straightforward. On average, around half the items in the last level of one tree will need moving and bubbling, so that will be $O(n)$ moves, each with a cost of $O(\log_2 n)$, again giving an overall time complexity of $O(n\log_2 n)$. However, the actual number of operations required will, on average, be a lot less than the previous approach, by something like a factor of four, so this approach is more efficient, even though the algorithm is more complex.

3. Simply concatenate the array forms of the heap trees `s` and `t` and use the standard `heapify` algorithm to convert that array into a new binary heap tree. The `heapify` algorithm has time complexity $O(n)$, and the concatenation need be no more than that, so this approach has $O(n)$ overall time complexity, making it in the best general approach of all three.

Thus, the merging of binary heap trees generally has $O(n)$ time complexity.

If the two binary heap trees are such that very few moves are required for the second approach, then that may look like a better choice of approach than the third approach. However, `makeTree` will itself generally be an $O(n)$ procedure if the trees are array-based, rather than pointer-based, which they usually are for binary heap trees. So, for array-based similarly-sized binary heaps, the third approach is usually best.

If the heap trees to be merged have very different sizes n and $m < n$, the first approach will have overall time complexity $O(m\log_2 n)$, which could be more efficient than an $O(n)$ approach if $m \ll n$. In practice, a good general purpose merge algorithm would check the sizes of the two trees and use them to determine the best approach to apply.

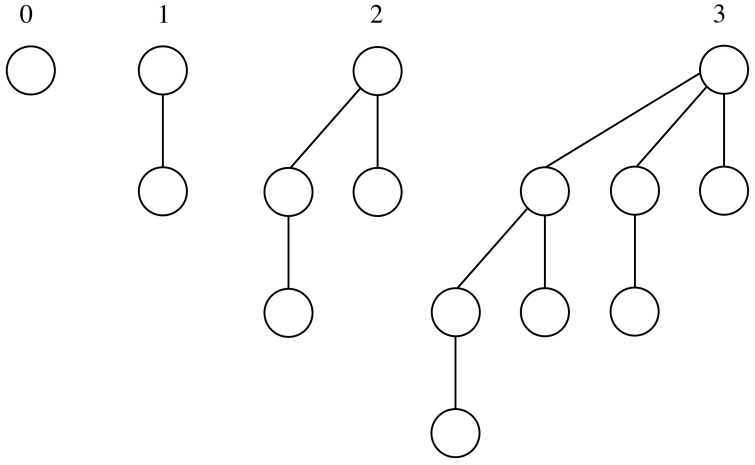
8.8 Binomial heaps

A *Binomial heap* is similar to a binary heap as described above, but has the advantage of more efficient procedures for insertion and merging. Unlike a binary heap, which consists of a single binary tree, a binomial heap is implemented as a collection of binomial trees.

Definition. A *binomial tree* is defined recursively as follows:

- A binomial tree of order 0 is a single node.
- A binomial tree of order k has a root node with children that are roots of binomial trees of orders $k-1$, $k-2$, ..., 2, 1, 0 (in that order).

Thus, a binomial tree of order k has height k, contains 2^k nodes, and is trivially constructed by attaching one order $k-1$ binomial tree as the left-most child of another order $k-1$ binomial tree. Binomial trees of order 0, 1, 2 and 3 take the form:



and it is clear from these what higher order trees will look like.

A *Binomial heap* is constructed as a collection of binomial trees with a particular structure and node ordering properties:

- There can only be zero or one binomial tree of each order.
- Each constituent binomial tree must satisfy the priority ordering property, i.e. each node must have priority less than or equal to its parent.

The structure of such a heap is easily understood by noting that a binomial tree of order k contains exactly 2^k nodes, and a binomial heap can only contain zero or one binomial tree of each order, so the total number of nodes in a Binomial Heap must be

$$n = \sum_{k=0}^{\infty} b_k 2^k \qquad b_k \in [0, 1]$$

where b_k specifies the number of trees of order k. Thus there is a one-to-one mapping between the binomial heap structure and the standard binary representation of the number n, and since the binary representation is clearly unique, so is the binomial heap structure. The maximum number of trees in a heap with n nodes therefore equals the number of digits when n is written in binary without leading zeros, i.e. $\log_2 n + 1$. The heap can be stored efficiently as a linked list of root nodes ordered by increasing tree order.

The most important operation for binomial heaps is *merge*, because that can be used as a sub-process for most other operations. Underlying that is the merge of two binomial trees of order j into a binomial tree of order $j + 1$. By definition, that is achieved by adding one of those trees as the left most sub-tree of the root of the other, and preservation of the priority ordering simply requires that it is the tree with the highest priority root that provides the root of the combined tree. This clearly has $O(1)$ time complexity. Then merging two whole binomial heaps is achieved by merging the constituent trees whenever there are two of the same order, in a sequential manner analogous to the addition of two binary numbers. In this case, the $O(1)$ insert complexity will be multiplied by the number of trees, which is $O(\log_2 n)$, so the overall time complexity of merge is $O(\log_2 n)$. This is better than the $O(n)$ complexity of merging binary heaps that can be achieved by concatenating the heap arrays and using the $O(n)$ heapify algorithm.

Insertion of a new element into an existing binomial heap can easily be done by treating the new element as a binomial heap consisting of a single node (i.e., an order zero tree), and merging that using the standard merge algorithm. The average time complexity of that *insert* is given by computing the average number of $O(1)$ tree combinations required. The probability of needing the order zero combination is 0.5, the probability of needing a second combination is 0.5^2, and the third is 0.5^3, and so on, which sum to one. So insertion has $O(1)$ overall time complexity. That is better than the $O(\log_2 n)$ complexity of insertion into a standard binary heap.

Creating a whole new binomial heap from scratch can be achieved by using the $O(1)$ insert process for each of the n items, giving an overall time complexity of $O(n)$. In this case, there is no better process, so heapify here has the same time complexity as the *heapify* algorithm for binary heaps.

Another important heap operation in practice is that of updating the heap after increasing a node priority. For standard binary heaps, that simply requires application of the usual bubble-up process with $O(\log_2 n)$ complexity. Clearly, a similar process can be used in binomial heaps, and that will also be of $O(\log_2 n)$ complexity.

The highest priority node in a binomial heap will clearly be the highest priority root node, and a pointer to that can be maintained by each heap update operation without increasing the complexity of the operation. Serving the highest priority item requires deleting the highest priority node from the order j tree it appears in, and that will break it up into another binomial heap consisting of trees of all orders from 0 to $j - 1$. However, those trees can easily be merged back into the original heap using the standard merge algorithm, with the

standard merge complexity of $O(\log_2 n)$. Deleting non-root nodes can also be achieved with the existing operations by increasing the relevant node priority to infinity, bubbling-up, and using the root delete operation, again with $O(\log_2 n)$ complexity overall. So, the complexity of *delete* is always $O(\log_2 n)$.

Exercise: Find pseudocode versions of the merge, insert and delete algorithms for binomial heaps, and see exactly how their time complexities arise.

8.9 Fibonacci heaps

A *Fibonacci heap* is another collection of trees that satisfy the standard priority-ordering property. It can be used to implement a priority queue in a similar way to binary or binomial heaps, but the structure of Fibonacci heaps are more flexible and efficient, which allows them to have better time complexities. They are named after the *Fibonacci numbers* that restrict the tree sizes and appear in their time complexity analysis.

The flexibility and efficiency of Fibonacci heaps comes at the cost of more complexity: the trees do not have a fixed shape, and in the extreme cases every element in the heap can be in a separate tree. Normally, the roots of all the trees are stored using a *circular doubly linked list*, and the children of each node are handled in the same way. A pointer to the highest priority root node is maintained, making it trivial to find the highest priority node in the heap. The efficiency is achieved by performing many operations in a *lazy* manner, with much of the work postponed for later operations to deal with.

Fibonacci heaps can easily be *merged* with $O(1)$ complexity by simply concatenating the two lists of root nodes, and then *insertion* can be done by merging the existing heap with a new heap consisting only of the new node. By inserting n items one at a time, a whole heap can be created from scratch with $O(n)$ complexity.

Obviously, at some point, order needs to be introduced into the heap to achieve the overall efficiency. This is done by keeping the number of children of all nodes to be at most $O(\log_2 n)$, and the size of a subtree rooted in a node with k children is at least F_{k+2}, where F_k is the kth Fibonacci number. The number of trees in the heap is decreased as part of the *delete* operation that is used to remove the highest priority node and update the pointer to the highest priority root. This delete algorithm is quite complex. First it removes the highest priority root, leaving its children to become roots of new trees within the heap, the processing of which will be $O(\log_2 n)$. Then the number of trees is reduced by linking together trees that have roots with the same number of children, similar to a Binomial heap, until every root has a different number of children, leaving at most $O(\log_2 n)$ trees. Finally the roots of those trees are checked to reset the pointer to the highest priority. It can be shown that all the required processes can be completed with $O(\log_2 n)$ average time complexity.

For each node, a record is kept of its number of children and whether it is marked. The mark indicates that at least one of its children has been separated since the node was made a child of another node, so all roots are unmarked. The mark is used by the algorithm for increasing a node priority, which is also complex, but can be achieved with $O(1)$ complexity. This gives Fibonacci heaps an important advantage over both binary and binomial heaps for which this operation has $O(\log_2 n)$ time complexity.

Finally, an arbitrary node can be deleted from the heap by increasing its node priority to infinity and applying the delete highest priority algorithm, resulting in an overall time complexity of $O(\log_2 n)$.

Exercise: Find pseudocode versions of the various Fibonacci heap operations, and work out how Fibonacci numbers are involved in computing their time complexities.

8.10 Comparison of heap time complexities

It is clear that the more complex Binomial and Fibonacci Heaps offer average time complexity advantages over simple Binary Heap Trees. The following table summarizes the average time complexities of the crucial heap operations:

Heap type	Insert	Delete	Merge	Heapify	Up priority
Binary	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$	$O(n)$	$O(\log_2 n)$
Binomial	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$	$O(\log_2 n)$
Fibonacci	$O(1)$	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(1)$

Obviously it will depend on the application in question whether using a more complicated heap is worth the effort. We shall see later that Fibonacci heaps are important in practice because they are used in the most efficient versions of many algorithms that can be implemented using priority queues, such as *Dijkstra's algorithm* for finding shortest routes, and *Prim's algorithm* for finding minimal spanning trees.

Chapter 9

Sorting

9.1 The problem of sorting

In computer science, 'sorting' usually refers to bringing a set of items into some well-defined order. To be able to do this, we first need to specify the notion of *order* on the items we are considering. For example, for numbers we can use the usual numerical order (that is, defined by the mathematical 'less than' or '$<$' relation) and for strings the so-called *lexicographic* or *alphabetic* order, which is the one dictionaries and encyclopedias use.

Usually, what is meant by *sorting* is that once the sorting process is finished, there is a simple way of 'visiting' all the items in order, for example to print out the contents of a database. This may well mean different things depending on how the data is being stored. For example, if all the objects are sorted and stored in an array a of size n, then

```
for i = 0,...,n-1
   print(a[i])
```

would print the items in ascending order. If the objects are stored in a linked list, we would expect that the first entry is the smallest, the next the second-smallest, and so on. Often, more complicated structures such as binary search trees or heap trees are used to sort the items, which can then be printed, or written into an array or linked list, as desired.

Sorting is important because having the items in order makes it much easier to *find* a given item, such as the cheapest item or the file corresponding to a particular student. It is thus closely related to the problem of *search*, as we saw with the discussion of binary search tress. If the sorting can be done beforehand (*off-line*), this enables faster *access* to the required item, which is important because that often has to be done on the fly (*on-line*). We have already seen that, by having the data items stored in a sorted array or binary search tree, we can reduce the average (and worst case) complexity of searching for a particular item to $O(\log_2 n)$ steps, whereas it would be $O(n)$ steps without sorting. So, if we often have to look up items, it is worth the effort to sort the whole collection first. Imagine using a dictionary or phone book in which the entries do not appear in some known logical order.

It follows that sorting algorithms are important tools for program designers. Different algorithms are suited to different situations, and we shall see that there is no 'best' sorting algorithm for everything, and therefore a number of them will be introduced in these notes. It is worth noting that we will be far from covering *all* existing sorting algorithms – in fact, the field is still very much alive, and new developments are taking place all the time. However,

the general strategies can now be considered to be well-understood, and most of the latest new algorithms tend to be derived by simply tweaking existing principles, although we still do not have accurate measures of performance for some sorting algorithms.

9.2 Common sorting strategies

One way of organizing the various sorting algorithms is by classifying the underlying idea, or 'strategy'. Some of the key strategies are:

enumeration sorting Consider all items. If we know that there are N items which are smaller than the one we are currently considering, then its final position will be at number $N + 1$.

exchange sorting If two items are found to be out of order, exchange them. Repeat till all items are in order.

selection sorting Find the smallest item, put it in the first position, find the smallest of the remaining items, put it in the second position ...

insertion sorting Take the items one at a time and insert them into an initially empty data structure such that the data structure continues to be sorted at each stage.

divide and conquer Recursively split the problem into smaller sub-problems till you just have single items that are trivial to sort. Then put the sorted 'parts' back together in a way that preserves the sorting.

All these strategies are based on *comparing* items and then rearranging them accordingly. These are known as comparison-based sorting algorithms. We will later consider other non-comparison-based algorithms which are possible when we have specific prior knowledge about the items that can occur, or restrictions on the range of items that can occur.

The ideas above are based on the assumption that all the items to be sorted will fit into the computer's internal memory, which is why they are often referred to as being *internal sorting algorithms*. If the whole set of items cannot be stored in the internal memory at one time, different techniques have to be used. These days, given the growing power and memory of computers, external storage is becoming much less commonly needed when sorting, so we will not consider *external sorting algorithms* in detail. Suffice to say, they generally work by splitting the set of items into subsets containing as many items as can be handled at one time, sorting each subset in turn, and then carefully merging the results.

9.3 How many comparisons must it take?

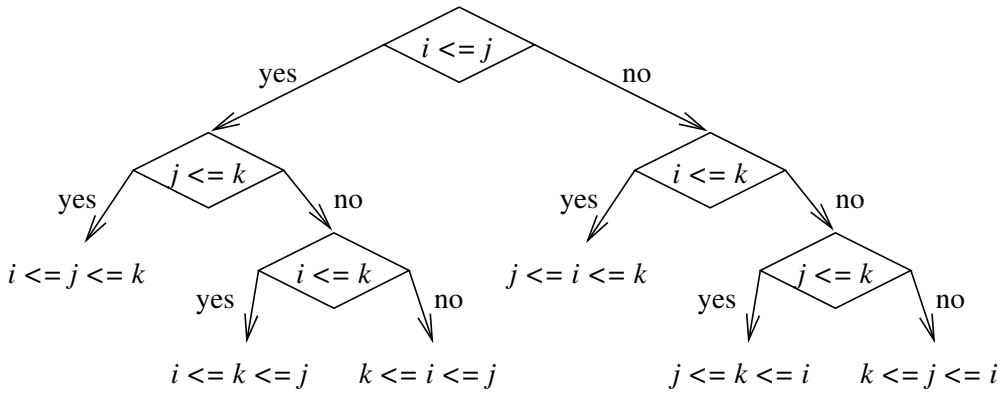
An obvious way to compute the *time complexity* of sorting algorithms is to count the number of comparisons they need to carry out, as a function of the number of items to be sorted. There is clearly no general *upper bound* on the number of comparisons used, since a particularly stupid algorithm might compare the same two items indefinitely. We are more interested in having a *lower bound* for the number of comparisons needed for the best algorithm in the worst case. In other words, we want to know the minimum number of comparisons required

to have all the information needed to sort an arbitrary collection of items. Then we can see how well particular sorting algorithms compare against that theoretical lower bound.

In general, questions of this kind are rather hard, because of the need to consider *all* possible algorithms. In fact, for some problems, optimal lower bounds are not yet known. One important example is the so-called *Travelling Salesman Problem* (TSP), for which all algorithms, which are known to give the correct shortest route solution, are extremely inefficient in the worst case (many to the extent of being useless in practice). In these cases, one generally has to relax the problem to find solutions which are *probably approximately correct*. For the TSP, it is still an open problem whether there exists a feasible algorithm that is guaranteed to give the exact shortest route.

For sorting algorithms based on comparisons, however, it turns out that a tight lower bound does exist. Clearly, even if the given collection of items is already sorted, we must still check all the items one at a time to see whether they are in the correct order. Thus, the lower bound must be at least n, the number of items to be sorted, since we need at least n steps to examine every element. If we already knew a sorting algorithm that works in n steps, then we could stop looking for a better algorithm: n would be both a lower bound and an upper bound to the minimum number of steps, and hence an *exact bound*. However, as we shall shortly see, no algorithm can actually take fewer than $O(n\log_2 n)$ comparisons in the worst case. If, in addition, we can design an algorithm that works in $O(n\log_2 n)$ steps, then we will have obtained an exact bound. We shall start by demonstrating that every algorithm needs at least $O(n\log_2 n)$ comparisons.

To begin with, let us assume that we only have three items, i, j, and k. If we have found that $i \leq j$ and $j \leq k$, then we know that the sorted order is: i, j, k. So it took us two comparisons to find this out. In some cases, however, it is clear that we will need as many as three comparisons. For example, if the first two comparisons tell us that $i > j$ and $j \leq k$, then we know that j is the smallest of the three items, but we cannot say from this information how i and k relate. A third comparison is needed. So what is the *average* and *worst* number of comparisons that are needed? This can best be determined from the so-called *decision tree*, where we keep track of the information gathered so far and count the number of comparisons needed. The decision tree for the three item example we were discussing is:



So what can we deduce from this about the general case? The decision tree will obviously always be a binary tree. It is also clear that its *height* will tell us how many comparisons will be needed in the worst case, and that the average length of a path from the root to a leaf will give us the average number of comparisons required. The leaves of the decision tree are

all the possible *outcomes*. These are given by the different possible orders we can have on n items, so we are asking how many ways there are of arranging n items. The first item can be any of the n items, the second can be any of the remaining $n-1$ items, and so forth, so their total number is $n(n-1)(n-2)\cdots 3\cdot 2\cdot 1 = n!$. Thus we want to know the height h of a binary tree that can accommodate as many as $n!$ leaves. The number of leaves of a tree of height h is at most 2^h, so we want to find h such that

$$2^h \geq n! \quad \text{or} \quad h \geq \log_2(n!)$$

There are numerous approximate expressions that have been derived for $\log_2(n!)$ for large n, but they all have the same dominant term, namely $n\log_2 n$. (Remember that, when talking about time complexity, we ignore any sub-dominant terns and constant factors.) Hence, no sorting algorithm based on comparing items can have a better average or worst case performance than using a number of comparisons that is approximately $n\log_2 n$ for large n. It remains to be seen whether this $O(n\log_2 n)$ complexity can actually be achieved in practice. To do this, we would have to exhibit at least one algorithm with this performance behaviour (and convince ourselves that it really does have this behaviour). In fact, we shall shortly see that there are several algorithms with this behaviour.

We shall proceed now by looking in turn at a number of sorting algorithms of increasing sophistication, that involve the various strategies listed above. The way they work depends on what kind of data structure contains the items we wish to sort. We start with approaches that work with simple arrays, and then move on to using more complex data structures that lead to more efficient algorithms.

9.4 Bubble Sort

Bubble Sort follows the *exchange sort* approach. It is very easy to implement, but tends to be particularly slow to run. Assume we have array `a` of size `n` that we wish to sort. Bubble Sort starts by comparing `a[n-1]` with `a[n-2]` and swaps them if they are in the wrong order. It then compares `a[n-2]` and `a[n-3]` and swaps those if need be, and so on. This means that once it reaches `a[0]`, the smallest entry will be in the correct place. It then starts from the back again, comparing pairs of 'neighbours', but leaving the zeroth entry alone (which is known to be correct). After it has reached the front again, the second-smallest entry will be in place. It keeps making 'passes' over the array until it is sorted. More generally, at the ith stage Bubble Sort compares neighbouring entries 'from the back', swapping them as needed. The item with the lowest index that is compared to its right neighbour is `a[i-1]`. After the ith stage, the entries `a[0],...,a[i-1]` are in their final position.

At this point it is worth introducing a simple 'test-case' of size `n = 4` to demonstrate how the various sorting algorithms work:

4	1	3	2

Bubble Sort starts by comparing `a[3]=2` with `a[2]=3`. Since they are not in order, it swaps them, giving | 4 | 1 | 2 | 3 |. It then compares `a[2]=2` with `a[1]=1`. Since those are in order, it leaves them where they are. Then it compares `a[1]=1` with `a[0]=4`, and those are not in order once again, so they have to be swapped. We get | 1 | 4 | 2 | 3 |. Note that the smallest entry has reached its final place. This will *always* happen after Bubble Sort has done its first 'pass' over the array.

Now that the algorithm has reached the zeroth entry, it starts at the back again, comparing `a[3]=3` with `a[2]=2`. These entries are in order, so nothing happens. (Note that these numbers have been compared before – there is nothing in Bubble Sort that prevents it from repeating comparisons, which is why it tends to be pretty slow!) Then it compares `a[2]=2` and `a[1]=4`. These are not in order, so they have to be swapped, giving | 1 | 2 | 4 | 3 |. Since we already know that `a[0]` contains the smallest item, we leave it alone, and the second pass is finished. Note that now the second-smallest entry is in place, too.

The algorithm now starts the third and final pass, comparing `a[3]=3` and `a[2]=4`. Again these are out of order and have to be swapped, giving | 1 | 2 | 3 | 4 |. Since it is known that `a[0]` and `a[1]` contain the correct items already, they are not touched. Furthermore, the third-smallest item is in place now, which means that the fourth-smallest *has to be* correct, too. Thus the whole array is sorted.

It is now clear that Bubble Sort can be implemented as follows:

```
for ( i = 1 ; i < n ; i++ )
   for ( j = n-1 ; j >= i ; j-- )
      if ( a[j] < a[j-1] )
         swap a[j] and a[j-1]
```

The outer loop goes over all `n` $-$ `1` positions that may still need to be swapped to the left, and the inner loop goes from the end of the array back to that position.

As is usual for comparison-based sorting algorithms, the time complexity will be measured by counting the number of comparisons that are being made. The outer loop is carried out $n-1$ times. The inner loop is carried out $(n-1)-(i-1) = n-i$ times. So the number of comparisons is the same in each case, namely

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i}^{n-1} 1 &= \sum_{i=1}^{n-1}(n-i) \\ &= (n-1)+(n-2)+\cdots+1 \\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Thus the worst case and average case number of comparisons are both proportional to n^2, and hence the average and worst case time complexities are $O(n^2)$.

9.5 Insertion Sort

Insertion Sort is (not surprisingly) a form of *insertion sorting*. It starts by treating the first entry `a[0]` as an already sorted array, then checks the second entry `a[1]` and compares it with the first. If they are in the wrong order, it swaps the two. That leaves `a[0],a[1]` sorted. Then it takes the third entry and positions it in the right place, leaving `a[0],a[1],a[2]` sorted, and so on. More generally, at the beginning of the ith stage, Insertion Sort has the entries `a[0],..., a[i-1]` sorted and inserts `a[i]`, giving sorted entries `a[0],...,a[i]`.

For the example starting array | 4 | 1 | 3 | 2 |, Insertion Sort starts by considering `a[0]=4` as sorted, then picks up `a[1]` and 'inserts it' into the already sorted array, increasing the size of it by 1. Since `a[1]=1` is smaller than `a[0]=4`, it has to be inserted in the zeroth slot,

but that slot is holding a value already. So we first move `a[0]` 'up' one slot into `a[1]` (care being taken to remember `a[1]` first!), and then we can move the old `a[1]` to `a[0]`, giving

1	4	3	2

.

At the next step, the algorithm treats `a[0],a[1]` as an already sorted array and tries to insert `a[2]=3`. This value obviously has to fit between `a[0]=1` and `a[1]=4`. This is achieved by moving `a[1]` 'up' one slot to `a[2]` (the value of which we assume we have remembered), allowing us to move the current value into `a[1]`, giving

1	3	4	2

.

Finally, `a[3]=2` has to be inserted into the sorted array `a[0],...,a[2]`. Since `a[2]=4` is bigger than 2, it is moved 'up' one slot, and the same happens for `a[1]=3`. Comparison with `a[0]=1` shows that `a[1]` was the slot we were looking for, giving

1	2	3	4

.

The general algorithm for Insertion Sort can therefore be written:

```
for ( i = 1 ; i < n ; i++ ) {
   for( j = i ; j > 0 ; j-- )
      if ( a[j] < a[j-1] )
         swap a[j] and a[j-1]
      else break
}
```

The outer loop goes over the `n - 1` items to be inserted, and the inner loop takes each next item and swaps it back through the currently sorted portion till it reaches its correct position. However, this typically involves swapping each next item many times to get it into its right position, so it is more efficient to store each next item in a temporary variable `t` and only insert it into its correct position when that has been found and its content moved:

```
for ( i = 1 ; i < n ; i++ ) {
   j = i
   t = a[j]
   while ( j > 0 && t < a[j-1] ) {
      a[j] = a[j-1]
      j--
   }
   a[j] = t
}
```

The outer loop again goes over `n-1` items, and the inner loop goes back through the currently sorted portion till it finds the correct position for the next item to be inserted.

The time complexity is again taken to be the number of comparisons performed. The outer loop is always carried out $n-1$ times. How many times the inner loop is carried out depends on the items being sorted. In the worst case, it will be carried out i times; on average, it will be half that often. Hence the number of comparison in the worst case is:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=1}^{i} 1 &= \sum_{i=1}^{n-1} i \\ &= 1+2+\cdots+(n-1) \\ &= \frac{n(n-1)}{2};\end{aligned}$$

and in the average case it is half that, namely $n(n-1)/4$. Thus average and worst case number of steps for of Insertion Sort are both proportional to n^2, and hence the average and worst case time complexities are both $O(n^2)$.

9.6 Selection Sort

Selection Sort is (not surprisingly) a form of *selection sorting*. It first finds the smallest item and puts it into `a[0]` by exchanging it with whichever item is in that position already. Then it finds the second-smallest item and exchanges it with the item in `a[1]`. It continues this way until the whole array is sorted. More generally, at the ith stage, Selection Sort finds the ith-smallest item and swaps it with the item in `a[i-1]`. Obviously there is no need to check for the ith-smallest item in the first $i-1$ elements of the array.

For the example starting array | 4 | 1 | 3 | 2 |, Selection Sort first finds the smallest item in the whole array, which is `a[1]=1`, and swaps this value with that in `a[0]` giving | 1 | 4 | 3 | 2 |. Then, for the second step, it finds the smallest item in the reduced array `a[1],a[2],a[3]`, that is `a[3]=2`, and swaps that into `a[1]`, giving | 1 | 2 | 3 | 4 |. Finally, it finds the smallest of the reduced array `a[2],a[3]`, that is `a[2]=3`, and swaps that into `a[2]`, or recognizes that a swap is not needed, giving | 1 | 2 | 3 | 4 |.

The general algorithm for Selection Sort can be written:

```
for ( i = 0 ; i < n-1 ; i++ ) {
   k = i
   for ( j = i+1 ; j < n ; j++ )
      if ( a[j] < a[k] )
         k = j
   swap a[i] and a[k]
}
```

The outer loop goes over the first `n-1` positions to be filled, and the inner loop goes through the currently unsorted portion to find the next smallest item to fill the next position. Note that, unlike with Bubble Sort and Insertion Sort, there is exactly one swap for each iteration of the outer loop,

The time complexity is again the number of comparisons carried out. The outer loop is carried out $n-1$ times. In the inner loop, which is carried out $(n-1)-i = n-1-i$ times, one comparison occurs. Hence the total number of comparisons is:

$$\begin{aligned}\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 &= \sum_{i=0}^{n-2}(n-1-i)\\ &= (n-1)+\cdots+2+1\\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Therefore the number of comparisons for Selection Sort is proportional to n^2, in the worst case as well as in the average case, and hence the average and worst case time complexities are both $O(n^2)$.

Note that Bubblesort, Insertion Sort and Selection Sort all involve two nested for loops over $O(n)$ items, so it is easy to see that their overall complexities will be $O(n^2)$ without having to compute the exact number of comparisons.

9.7 Comparison of $O(n^2)$ sorting algorithms

We have now seen three different array based sorting algorithms, all based on different sorting strategies, and all with $O(n^2)$ time complexity. So one might imagine that it does not make much difference which of these algorithms is used. However, in practice, it can actually make a big difference which algorithm is chosen. The following table shows the measured running times of the three algorithms applied to arrays of integers of the size given in the top row:

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Insertion Sort	15	69	276	1137	6	2200	4536
Selection Sort	12	45	164	634	643	833	2497

Here O1024 denotes an array with 1024 entries which are already sorted, and R1024 is an array which is sorted in the *reverse* order, that is, from biggest to smallest. All the other arrays were filled randomly. Warning: tables of measurements like this are *always* dependent on the random ordering used, the implementation of the programming language involved, and on the machine it was run on, and so will never be exactly the same.

So where exactly do these differences come from? For a start, Selection Sort always makes $n(n-1)/2$ comparisons, but carries out *at most* $n-1$ swaps. Each swap requires three assignments and takes, in fact, more time than a comparison. Bubble Sort, on the other hand, does a lot of swaps. Insertion Sort does particularly well on data which is sorted already – in such a case, it only makes $n-1$ comparisons. It is worth bearing this in mind for some applications, because if only a few entries are out of place, Insertion Sort can be very quick. These comparisons serve to show that complexity considerations can be rather delicate, and require good judgement concerning what operations to count. It is often a good idea to run some experiments to test the theoretical considerations and see whether any simplifications made are realistic in practice. For instance, we have assumed here that all comparisons cost the same, but that may not be true for big numbers or strings of characters.

What exactly to count when considering the complexity of a particular algorithm is always a judgement call. You will have to gain experience before you feel comfortable with making such decisions yourself. Furthermore, when you want to improve the performance of an algorithm, you may want to determine the biggest user of computing *resources* and focus on improving that. Something else to be aware of when making these calculations is that it is not a bad idea to keep track of any constant factors, in particular those that go with the dominating sub-term. In the above examples, the factor applied to the dominating sub-term, namely n^2, varies. It is $1/2$ for the average case of Bubble Sort and Selection Sort, but only $1/4$ for Insertion Sort. It is certainly useful to know that an algorithm that is linear will perform better than a quadratic one *provided the size of the problem is large enough*, but if you know that your problem has a size of, say, at most 100, then a complexity of $(1/20)n^2$ will be preferable to one of $20n$. Or if you know that your program is only ever used on fairly small samples, then using the simplest algorithm you can find might be beneficial overall – it is easier to program, and there is not a lot of compute time to be saved.

Finally, the above numbers give you some idea why, for program designers, the general rule is to *never* use Bubble Sort. It is certainly easy to program, but that is about all it has going for it. You are better off avoiding it altogether.

9.8 Sorting algorithm stability

One often wants to sort items which might have identical keys (e.g., ages in years) in such a way that items with identical keys are kept in their original order, particularly if the items have already been sorted according to a different criteria (e.g., alphabetical). So, if we denote the original order of an array of items by subscripts, we want the subscripts to end up in order for each set of items with identical keys. For example, if we start out with the array $[5_1, 4_2, 6_3, 5_4, 6_5, 7_6, 5_7, 2_8, 9_9]$, it should be sorted to $[2_8, 4_2, 5_1, 5_4, 5_7, 6_3, 6_5, 7_6, 9_9]$ and not to $[2_8, 4_2, 5_4, 5_1, 5_7, 6_3, 6_5, 7_6, 9_9]$. Sorting algorithms which satisfy this useful property are said to be *stable*.

The easiest way to determine whether a given algorithm is stable is to consider whether the algorithm can ever swap identical items past each other. In this way, the stability of the sorting algorithms studied so far can easily be established:

Bubble Sort This is stable because no item is swapped past another unless they are in the wrong order. So items with identical keys will have their original order preserved.

Insertion Sort This is stable because no item is swapped past another unless it has a smaller key. So items with identical keys will have their original order preserved.

Selection Sort This is not stable, because there is nothing to stop an item being swapped past another item that has an identical key. For example, the array $[2_1, 2_2, 1_3]$ would be sorted to $[1_3, 2_2, 2_1]$ which has items 2_2 and 2_1 in the wrong order.

The issue of sorting stability needs to be considered when developing more complex sorting algorithms. Often there are stable and non-stable versions of the algorithms, and one has to consider whether the extra cost of maintaining stability is worth the effort.

9.9 Treesort

Let us now consider a way of implementing an *insertion sorting* algorithm using a data structure better suited to the problem. The idea here, which we have already seen before, involves inserting the items to be sorted into an initially empty binary search tree. Then, when all items have been inserted, we know that we can traverse the binary search tree to visit all the items in the right order. This sorting algorithm is called *Treesort*, and for the basic version, we require that all the search keys be different.

Obviously, the tree must be kept balanced in order to minimize the number of comparisons, since that depends on the height of the tree. For a balanced tree that is $O(\log_2 n)$. If the tree is not kept balanced, it will be more than that, and potentially $O(n)$.

Treesort can be difficult to compare with other sorting algorithms, since it returns a tree, rather than an array, as the sorted data structure. It should be chosen if it is desirable to have the items stored in a binary search tree anyway. This is usually the case if items are frequently deleted or inserted, since a binary search tree allows these operations to be implemented efficiently, with time complexity $O(\log_2 n)$ per item. Moreover, as we have seen before, searching for items is also efficient, again with time complexity $O(\log_2 n)$.

Even if we have an array of items to start with, and want to finish with a sorted array, we can still use Treesort. However, to output the sorted items into the original array, we will need another procedure `fillArray(tree t, array a, int j)` to traverse the tree `t` and fill the array `a`. That is easiest done by passing and returning an index `j` that keeps track of the next array position to be filled. This results in the complete Treesort algorithm:

```
treeSort(array a) {
   t = EmptyTree
   for ( i = 0 ; i < size(a) ; i++ )
      t = insert(a[i],t)
   fillArray(t,a,0)
}

fillArray(tree t, array a, int j) {
   if ( not isEmpty(t) ) {
      j = fillArray(left(t),a,j)
      a[j++] = root(t)
      j = fillArray(right(t),a,j)
   }
   return j
}
```

which assumes that `a` is a pointer to the array location and that its elements can be accessed and updated given that and the relevant array index.

Since there are n items to insert into the tree, and each insertion has time complexity $O(\log_2 n)$, Treesort has an overall average time complexity of $O(n\log_2 n)$. So, we already have one algorithm that achieves the theoretical best average case time complexity of $O(n\log_2 n)$. Note, however, that if the tree is not kept balanced while the items are being inserted, and the items are already sorted, the height of the tree and number of comparisons per insertion will be $O(n)$, leading to a worst case time complexity of $O(n^2)$, which is no better than the simpler array-based algorithms we have already considered.

Exercise: We have assumed so far that the items stored in a Binary Search Tree must not contain any duplicates. Find the simplest ways to relax that restriction and determine how the choice of approach affects the *stability* of the associated Treesort algorithm.

9.10 Heapsort

We now consider another way of implementing a *selection sorting* algorithm using a more efficient data structure we have already studied. The underlying idea here is that it would help if we could pre-arrange the data so that selecting the smallest/biggest entry becomes easier. For that, remember the idea of a *priority queue* discussed earlier. We can take the value of each item to be its priority and then queue the items accordingly. Then, if we remove the item with the highest priority at each step we can fill an array in order 'from the rear', starting with the biggest item.

Priority queues can be implemented in a number of different ways, and we have already studied a straightforward implementation using *binary heap trees* in Chapter 8. However, there may be a better way, so it is worth considering the other possibilities.

An obvious way of implementing them would be using a sorted array, so that the entry with the highest priority appears in `a[n]`. Removing this item would be very simple, but inserting a new item would always involve finding the right position and shifting a number of items to the right to make room for it. For example, inserting a `3` into the queue $[1, 2, 4]$:

n	0	1	2	3	4	5
`a[n]`	1	2	4			

n	0	1	2	3	4	5
`a[n]`	1	2		4		

n	0	1	2	3	4	5
`a[n]`	1	2	3	4		

That kind of item insertion is effectively insertion sort and clearly inefficient in general, of $O(n)$ complexity rather than $O(\log_2 n)$ with a binary heap tree.

Another approach would be to use an unsorted array. In this case, a new item would be inserted by just putting it into `a[n+1]`, but to delete the entry with the highest priority would involve having to find it first. Then, after that, the last item would have to be swapped into the gap, or all items with a higher index 'shifted down'. Again, that kind of item deletion is clearly inefficient in general, of $O(n)$ complexity rather than $O(\log_2 n)$ with a heap tree.

Thus, of those three representations, only one is of use in carrying out the above idea efficiently. An unsorted array is what we started from, so that is not any help, and ordering the array is what we are trying to achieve, so heaps are the way forward.

To make use of binary heap trees, we first have to take the unsorted array and re-arrange it so that it satisfies the heap tree priority ordering. We have already studied the *heapify* algorithm which can do that with $O(n)$ time complexity. Then we need to extract the sorted array from it. In the heap tree, the item with the highest priority, that is the largest item, is always in `a[1]`. In the sorted array, it should be in the last position `a[n]`. If we simply swap the two, we will have that item at the right position of the array, and also have begun the standard procedure of removing the root of the heap-tree, since `a[n]` is precisely the item that would be moved into the root position at the next step. Since `a[n]` now contains the correct item, we will never have to look at it again. Instead, we just take the items `a[1],...,a[n-1]` and bring them back into a heap-tree form using the *bubble down* procedure on the new root, which we know to have complexity $O(\log_2 n)$.

Now the second largest item is in position `a[1]`, and its final position should be `a[n-1]`, so we now swap these two items. Then we rearrange `a[1],...,a[n-2]` back into a heap tree using the bubble down procedure on the new root. And so on.

When the ith step has been completed, the items `a[n-i+1],...,a[n]` will have the correct entries, and there will be a heap tree for the items `a[1],...,a[n-i]`. Note that the size, and therefore the height, of the heap tree decreases at each step. As a part of the ith step, we have to bubble down the new root. This will take at most twice as many comparisons as the height of the original heap tree, which is $\log_2 n$. So overall there are $n - 1$ steps, with at most $2\log_2 n$ comparisons, totalling $2(n-1)\log_2 n$. The number of comparisons will actually be less than that, because the number of bubble down steps will usually be less than the full height of the tree, but usually not much less, so the time complexity is still $O(n\log_2 n)$.

The full Heapsort algorithm can thus be written in a very simple form, using the bubble down and heapify procedures we already have from Chapter 8. First `heapify` converts the

array into a binary heap tree, and then the for loop moves each successive root one item at a time into the correct position in the sorted array:

```
heapSort(array a, int n) {
   heapify(a,n)
   for( j = n ; j > 1 ; j-- ) {
      swap a[1] and a[j]
      bubbleDown(1,a,j-1)
   }
}
```

It is clear from the swap step that the order of identical items can easily be reversed, so there is no way to render the Heapsort algorithm *stable*.

The average and worst-case time complexities of the entire Heapsort algorithm are given by the *sum* of two complexity functions, first that of `heapify` rearranging the original unsorted array into a heap tree which is $O(n)$, and then that of making the sorted array out of the heap tree which is $O(n\log_2 n)$ coming from the $O(n)$ bubble-downs each of which has $O(\log_2 n)$ complexity. Thus the overall average and worst-case complexities are both $O(n\log_2 n)$, and we now have a sorting algorithm that achieves the theoretical best worst-case time complexity. Using more sophisticated priority queues, such as Binomial or Fibonacci heaps, cannot improve on this because they have the same delete time complexity.

A useful feature of Heapsort is that if only the largest $m \ll n$ items need to be found and sorted, rather than all n, the complexity of the second stage is only $O(m\log_2 n)$, which can easily be less than $O(n)$ and thus render the whole algorithm only $O(n)$.

9.11 Divide and conquer algorithms

All the sorting algorithms considered so far work on the whole set of items together. Instead, *divide and conquer* algorithms recursively split the sorting problem into more manageable sub-problems. The idea is that it will usually be easier to sort many smaller collections of items than one big one, and sorting single items is trivial. So we repeatedly split the given collection into two smaller parts until we reach the 'base case' of one-item collections, which require no effort to sort, and then merge them back together again. There are two main approaches for doing this:

Assuming we are working on an array `a` of size n with entries `a[0],...,a[n-1]`, then the obvious approach is to simply split the set of indices. That is, we split the array at item $n/2$ and consider the two sub-arrays `a[0],...,a[(n-1)/2]` and `a[(n+1)/2],...,a[n-1]`. This method has the advantage that the splitting of the collection into two collections of equal (or nearly equal) size at each stage is easy. However, the two sorted arrays that result from each split have to be *merged* together carefully to maintain the ordering. This is the underlying idea for a sorting algorithm called *mergesort*.

Another approach would be to split the array in such a way that, at each stage, all the items in the first collection are no bigger than all the items in the second collection. The splitting here is obviously more complex, but all we have to do to put the pieces back together again at each stage is to take the first sorted array followed by the second sorted array. This is the underlying idea for a sorting algorithm called *Quicksort*.

We shall now look in detail at how these two approaches work in practice.

9.12 Quicksort

The general idea here is to repeatedly split (or partition) the given array in such a way that all the items in the first sub-array are smaller than all the items in the second sub-array, and then concatenate all the sub-arrays to give the sorted full array.

How to partition. The important question is how to perform this kind of splitting most efficiently. If the array is very simple, for example [4,3,7,8,1,6], then a good split would be to put all the items smaller than 5 into one part, giving [4,3,1], and all the items bigger than or equal to 5 into the other, that is [7,8,6]. Indeed, moving all items with a smaller key than some given value into one sub-array, and all entries with a bigger or equal key into the other sub-array is the standard *Quicksort* strategy. The value that defines the split is called the *pivot*. However, it is not obvious what is the best way to choose the pivot value.

One situation that we absolutely have to avoid is splitting the array into an *empty* sub-array and the whole array again. If we do this, the algorithm will not just perform badly, it will not even terminate. However, if the pivot is chosen to be an item in the array, and the pivot is kept in between and separate from both sub-arrays, then the sub-arrays being sorted at each recursion will always be at least one item shorter than the previous array, and the algorithm is guaranteed to terminate.

Thus, it proves convenient to split the array at each stage into the sub-array of values smaller than or equal to some chosen pivot item, followed by that chosen pivot item, followed by the sub-array of values greater than or equal to the chosen pivot item. Moreover, to save space, we do not actually split the array into smaller arrays. Instead, we simply *rearrange* the whole array to reflect the splitting. We say that we *partition* the array, and the *Quicksort* algorithm is then applied to the sub-arrays of this partitioned array.

In order for the algorithm to be called recursively, to sort ever smaller parts of the original array, we need to tell it *which part* of the array is currently under consideration. Therefore, Quicksort is called giving the lowest index (`left`) and highest index (`right`) of the sub-array it must work on. Thus the algorithm takes the form:

```
quicksort(array a, int left, int right) {
   if ( left < right ) {
      pivotindex = partition(a,left,right)
      quicksort(a,left,pivotindex-1)
      quicksort(a,pivotindex+1,right)
   }
}
```

for which the initial call would be `quicksort(a,0,n-1)` and the array `a` at the end is sorted. The crucial part of this is clearly the `partition`(`a`, `left`, `right`) procedure that rearranges the array so that it can be split around an appropriate pivot `a`[`pivotindex`].

If we were to split off only one item at a time, Quicksort would have n recursive calls, where n is the number of items in the array. If, on the other hand, we halve the array at each stage, it would only need $\log_2 n$ recursive calls. This can be made clear by drawing a binary tree whose nodes are labelled by the sub-arrays that have been split off at each stage, and measuring its height. Ideally then, we would like to get two sub-arrays of roughly equal size (namely half of the given array) since that is the most efficient way of doing this. Of course, that depends on choosing a good pivot.

Choosing the pivot. If we get the pivot 'just right' (e.g., choosing 5 in the above example), then the split will be as even as possible. Unfortunately, there is no quick guaranteed way of finding the optimal pivot. If the keys are integers, one could take the average value of all the keys, but that requires visiting *all* the entries to sample their key, adding considerable overhead to the algorithm, and if the keys are more complicated, such as strings, you cannot do this at all. More importantly, it would not necessarily give a pivot that is a value in the array. Some sensible *heuristic* pivot choice strategies are:

- Use a random number generator to produce an index `k` and then use `a[k]`.
- Take a key from 'the middle' of the array, that is `a[(n-1)/2]`.
- Take a small sample (e.g., 3 or 5 items) and take the 'middle' key of those.

Note that one should *never* simply choose the first or last key in the array as the pivot, because if the array is almost sorted already, that will lead to the particularly bad choice mentioned above, and this situation is actually quite common in practice.

Since there are so many reasonable possibilities, and they are all fairly straightforward, we will not give a specific implementation for any of these pivot choosing strategies, but just assume that we have a `choosePivot(a,left,right)` procedure that returns the index of the pivot for a particular sub-array (rather than the pivot value itself).

The partitioning. In order to carry out the partitioning within the given array, some thought is required as to how this may be best achieved. This is more easily demonstrated by an example than put into words. For a change, we will consider an array of strings, namely the programming languages: [c, fortran, java, ada, pascal, basic, haskell, ocaml]. The ordering we choose is the standard lexicographic one, and let the chosen pivot be "fortran".

We will use markers | to denote a partition of the array. To the left of the left marker, there will be items we know to have a key smaller than or equal to the pivot. To the right of the right marker, there will be items we know to have a key bigger than or equal to the pivot. In the middle, there will be the items we have not yet considered. Note that this algorithm proceeds to investigate the items in the array *from two sides*.

We begin by swapping the pivot value to the end of the array where it can easily be kept separate from the sub-array creation process, so we have the array: [|c, ocaml, java, ada, pascal, basic, haskell | fortran]. Starting from the left, we find "c" is less than "fortran", so we move the left marker one step to the right to give [c | ocaml, java, ada, pascal, basic, haskell | fortran]. Now "ocaml" is greater than "fortran", so we stop on the left and proceed from the right instead, without moving the left marker. We then find "haskell" is bigger than "fortran", so we move the right marker to the left by one, giving [c | ocaml, java, ada, pascal, basic, | haskell, fortran]. Now "basic" is smaller than "fortran", so we have two keys, "ocaml" and "basic", which are 'on the wrong side'. We therefore swap them, which allows us to move both the left and the right marker one step further towards the middle. This brings us to [c, basic | java, ada, pascal | ocaml, haskell, fortran]. Now we proceed from the left once again, but "java" is bigger than "fortran", so we stop there and switch to the right. Then "pascal" is bigger than "fortran", so we move the right marker again. We then find "ada", which is smaller than the pivot, so we stop. We have now got [c, basic | java, ada, | pascal, ocaml, haskell, fortran]. As before, we want to swap "java" and "ada", which leaves the left and the right markers in the same place: [c, basic, ada, java | | pascal, ocaml, haskell, fortran], so we

stop. Finally, we swap the pivot back from the last position into the position immediately after the markers to give [c, basic, ada, java | | fortran, ocaml, haskell, pascal].

Since we obviously cannot have the marker indices 'between' array entries, we will assume the left marker is on the left of `a[leftmark]` and the right marker is to the right of `a[rightmark]`. The markers are therefore 'in the same place' once `rightmark` becomes smaller than `leftmark`, which is when we stop. If we assume that the keys are integers, we can write the partitioning procedure, that needs to return the final pivot position, as:

```
partition(array a, int left, int right) {
   pivotindex = choosePivot(a, left, right)
   pivot = a[pivotindex]
   swap a[pivotindex] and a[right]
   leftmark = left
   rightmark = right - 1
   while (leftmark <= rightmark) {
      while (leftmark <= rightmark  and  a[leftmark] <= pivot)
          leftmark++
      while (leftmark <= rightmark  and  a[rightmark] >= pivot)
          rightmark--
      if (leftmark < rightmark)
         swap a[leftmark++] and a[rightmark--]
   }
   swap a[leftmark] and a[right]
   return leftmark
}
```

This achieves a partitioning that ends with the same items in the array, but in a different order, with all items to the left of the returned pivot position smaller or equal to the pivot value, and all items to the right greater or equal to the pivot value.

Note that this algorithm doesn't require any extra memory – it just swaps the items in the original array. However, the swapping of items means the algorithm is not *stable*. To render quicksort stable, the partitioning must be done in such a way that the order of identical items can never be reversed. A conceptually simple approach that does this, but requires more memory and copying, is to simply go systematically through the whole array, re-filling the array `a` with items less than or equal to the pivot, and filling a second array `b` with items greater or equal to the pivot, and finally copying the array `b` into the end of `a`:

```
partition2(array a, int left, int right) {
   create new array b of size right-left+1
   pivotindex = choosePivot(a, left, right)
   pivot = a[pivotindex]
   acount = left
   bcount = 1
   for ( i = left ; i <= right ; i++ ) {
      if ( i == pivotindex )
         b[0] = a[i]
      else if ( a[i] < pivot || (a[i] == pivot && i < pivotindex) )
```

```
            a[acount++] = a[i]
         else
            b[bcount++] = a[i]
      }
      for ( i = 0 ; i < bcount ; i++ )
         a[acount++] = b[i]
      return right-bcount+1
   }
```

Like the first partition procedure, this also achieves a partitioning with the same items in the array, but in a different order, with all items to the left of the returned pivot position smaller or equal to the pivot value, and all items to the right greater or equal to the pivot value.

Complexity of Quicksort. Once again we shall determine complexity based on the number of comparisons performed. The partitioning step compares each of n items against the pivot, and therefore has complexity $O(n)$. Clearly, some partition and pivot choice algorithms are less efficient than others, like `partition2` involving more copying of items than `partition`, but that does not generally affect the overall complexity class.

In the *worst case*, when an array is partitioned, we have one empty sub-array. If this happens at each step, we apply the partitioning method to arrays of size n, then $n-1$, then $n-2$, until we reach 1. Those complexity functions then add up to

$$n + (n-1) + (n-2) + \cdots 2 + 1 = n(n+1)/2$$

Ignoring the constant factor and the non-dominant term $n/2$, this shows that, in the worst case, the number of comparisons performed by Quicksort is $O(n^2)$.

In the *best case*, whenever we partition the array, the resulting sub-arrays will differ in size by at most one. Then we have n comparisons in the first case, two lots of $\lfloor n/2 \rfloor$ comparisons for the two sub-arrays, four times $\lfloor n/4 \rfloor$, eight times $\lfloor n/8 \rfloor$, and so on, down to $2^{\log_2 n - 1}$ times $\lfloor n/2^{\log_2 n - 1} \rfloor = \lfloor 2 \rfloor$. That gives the total number of comparisons as

$$n + 2^1\lfloor n/2^1 \rfloor + 2^2\lfloor n/2^2 \rfloor + 2^3\lfloor n/2^3 \rfloor + \cdots + 2^{\log_2 n - 1}\lfloor n/2^{\log_2 n - 1} \rfloor \approx n\log_2 n$$

which matches the theoretical best possible time complexity of $O(n\log_2 n)$.

More interesting and important is how well Quicksort does in the *average case*. However, that is much harder to analyze exactly. The strategy for choosing a pivot at each stage affects that, though as long as it avoids the problems outlined above, that does not change the complexity class. It also makes a difference whether there can be duplicate values, but again that doesn't change the complexity class. In the end, *all* reasonable variations involve comparing $O(n)$ items against a pivot, for each of $O(\log_2 n)$ recursions, so the total number of comparisons, and hence the overall time complexity, in the average case is $O(n\log_2 n)$.

Like Heapsort, when only the largest $m \ll n$ items need to be found and sorted, rather than all n, Quicksort can be modified to result in reduced time complexity. In this case, only the first sub-array needs to be processed at each stage, until the sub-array sizes exceed m. In that situation, for the best case, the total number of comparisons is reduced to

$$n + 1\lfloor n/2^1 \rfloor + 1\lfloor n/2^2 \rfloor + 1\lfloor n/2^3 \rfloor + \cdots + m\log_2 m \approx 2n.$$

rendering the time complexity of the whole modified algorithm only $O(n)$. For the average case, the computation is again more difficult, but as long as the key problems outlined above are avoided, the average-case complexity of this special case is also $O(n)$.

Improving Quicksort. It is always worthwhile spending some time optimizing the strategy for defining the pivot, since the particular problem in question might well allow for a more refined approach. Generally, the pivot will be better if more items are sampled before it is being chosen. For example, one could check several randomly chosen items and take the 'middle' one of those, the so called *median*. Note that in order to find the median of all the items, without sorting them first, we would end up having to make n^2 comparisons, so we cannot do that without making Quicksort unattractively slow.

Quicksort is rarely the most suitable algorithm if the problem size is small. The reason for this is all the overheads from the recursion (e.g., storing all the return addresses and formal parameters). Hence once the sub-problem become 'small' (a size of 16 is often suggested in the literature), Quicksort should stop calling itself and instead sort the remaining sub-arrays using a simpler algorithm such as Selection Sort.

9.13 Mergesort

The other divide and conquer sorting strategy based on repeatedly splitting the array of items into two sub-arrays, mentioned in Section 9.11, is called *mergesort*. This simply splits the array at each stage into its first and last half, without any reordering of the items in it. However, that will obviously not result in a set of sorted sub-arrays that we can just append to each other at the end. So mergesort needs another procedure `merge` that merges two sorted sub-arrays into another sorted array. As with binary search in Section 4.4, integer variables `left` and `right` can be used to refer to the lower and upper index of the relevant array, and `mid` refers to the end of its left sub-array. Thus a suitable mergesort algorithm is:

```
mergesort(array a, int left, int right) {
   if ( left < right ) {
      mid = (left + right) / 2
      mergesort(a, left, mid)
      mergesort(a, mid+1, right)
      merge(a, left, mid, right)
   }
}
```

Note that it would be relatively simple to modify this mergesort algorithm to operate on linked lists (of known length) rather than arrays. To 'split' such a list into two, all one has to do is set the pointer of the $\lfloor n/2 \rfloor$th list entry to null, and use the previously-pointed-to next entry as the head of the new second list. Of course, care needs to be taken to keep the list size information intact, and effort is required to find the crucial pointer for each split.

The `merge` algorithm. The principle of merging two sorted collections (whether they be lists, arrays, or something else) is quite simple: Since they are sorted, it is clear that the smallest item overall must be either the smallest item in the first collection or the smallest item in the second collection. Let us assume it is the smallest key in the first collection. Now the second smallest item overall must be either the second-smallest item in the first collection, or the smallest item in the second collection, and so on. In other words, we just work through both collections and at each stage, the 'next' item is the current item in either the first or the second collection.

The implementation will be quite different, however, depending on which data structure we are using. When arrays are used, it is actually necessary for the `merge` algorithm to create a new array to hold the result of the operation at least temporarily. In contrast, when using linked lists, it would be possible for `merge` to work by just changing the reference to the next node. This does make for somewhat more confusing code, however.

For arrays, a suitable *merge* algorithm would start by creating a new array `b` to store the results, then repeatedly add the next smallest item into it until one sub-array is finished, then copy the remainder of the unfinished sub-array, and finally copy `b` back into `a`:

```
merge(array a, int left, int mid, int right) {
   create new array b of size right-left+1
   bcount = 0
   lcount = left
   rcount = mid+1
   while ( (lcount <= mid) and (rcount <= right) ) {
      if ( a[lcount] <= a[rcount] )
         b[bcount++] = a[lcount++]
      else
         b[bcount++] = a[rcount++]
   }
   if ( lcount > mid )
      while ( rcount <= right )
         b[bcount++] = a[rcount++]
   else
      while ( lcount <= mid )
         b[bcount++] = a[lcount++]
   for ( bcount = 0 ; bcount < right-left+1 ; bcount++ )
      a[left+bcount] = b[bcount]
}
```

It is instructive to compare this with the `partition2` algorithm for Quicksort to see exactly where the two sort algorithms differ. As with `partition2`, the merge algorithm never swaps identical items past each other, and the splitting does not change the ordering at all, so the whole Mergesort algorithm is *stable*.

Complexity of Mergesort. The total number of comparisons needed at each recursion level of mergesort is the number of items needing merging which is $O(n)$, and the number of recursions needed to get to the single item level is $O(\log_2 n)$, so the total number of comparisons and its time complexity are $O(n\log_2 n)$. This holds for the worst case as well as the average case. Like Quicksort, it is possible to speed up mergesort by abandoning the recursive algorithm when the sizes of the sub-collections become small. For arrays, 16 would once again be a suitable size to switch to an algorithm like Selection Sort.

Note that, with Mergesort, for the special case when only the largest/smallest $m \ll n$ items need to be found and sorted, rather than all n, there is no way to reduce the time complexity in the way it was possible with Heapsort and Quicksort. This is because the ordering of the required items only emerges at the very last stage after the large majority of the comparisons have already been carried out.

9.14 Summary of comparison-based sorting algorithms

The following table summarizes the key properties of all the comparison-based sorting algorithms we have considered:

Sorting Algorithm	Strategy employed	Objects manipulated	Worst case complexity	Average case complexity	Stable
Bubble Sort	Exchange	arrays	$O(n^2)$	$O(n^2)$	Yes
Selection Sort	Selection	arrays	$O(n^2)$	$O(n^2)$	No
Insertion Sort	Insertion	arrays/lists	$O(n^2)$	$O(n^2)$	Yes
Treesort	Insertion	trees/lists	$O(n^2)$	$O(n\log_2 n)$	Yes
Heapsort	Selection	arrays	$O(n\log_2 n)$	$O(n\log_2 n)$	No
Quicksort	D & C	arrays	$O(n^2)$	$O(n\log_2 n)$	Maybe
Mergesort	D & C	arrays/lists	$O(n\log_2 n)$	$O(n\log_2 n)$	Yes

To see what the time complexities mean in practice, the following table compares the typical run times of those of the above algorithms that operate directly on arrays:

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Selection Sort	12	45	164	634	643	833	2497
Insertion Sort	15	69	276	1137	6	2200	4536
Heapsort	21	45	103	236	215	249	527
Quicksort	12	27	55	112	1131	1200	230
Quicksort2	6	12	24	57	1115	1191	134
Mergesort	18	36	88	188	166	170	409
Mergesort2	6	22	48	112	94	93	254

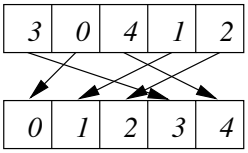
As before, arrays of the stated sizes are filled randomly, except O1024 that denotes an array with 1024 entries which are already sorted, and R1024 that denotes an array which is sorted in the *reverse* order. Quicksort2 and Mergesort2 are algorithms where the recursive procedure is abandoned in favour of Selection Sort once the size of the array falls to 16 or below. It should be emphasized again that these numbers are of limited accuracy, since they vary somewhat depending on machine and language implementation.

What has to be stressed here is that there is no 'best sorting algorithm' in general, but that there are usually good and bad choices of sorting algorithms *for particular circumstances*. It is up to the program designer to make sure that an appropriate one is picked, depending on the properties of the data to be sorted, how it is best stored, whether all the sorted items are required rather than some sub-set, and so on.

9.15 Non-comparison-based sorts

All the above sorting algorithms have been based on comparisons of the items to be sorted, and we have seen that we can't get time complexity better than $O(n\log_2 n)$ with comparison based algorithms. However, in some circumstances it is possible to do better than that with sorting algorithms that are not based on comparisons.

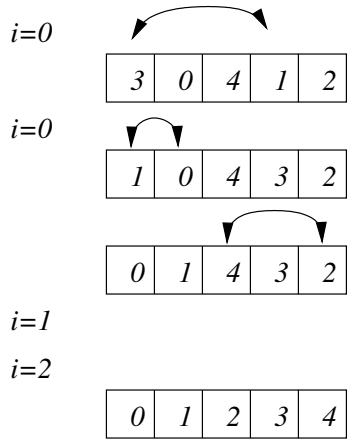
It is always worth thinking about the data that needs to be sorted, and whether comparisons really are required. For example, suppose you know the items to be sorted are the numbers from 0 to $n-1$. How would you sort those? The answer is surprisingly simple. We know that we have n entries in the array and we know *exactly which items should go there and in which order.* This is a very unusual situation as far as general sorting is concerned, yet this kind of thing often comes up in every-day life. For example, when a hotel needs to sort the room keys for its 100 rooms. Rather than employing one of the comparison-based sorting algorithms, in this situation we can do something much simpler. We can simply put the items directly in the appropriate places, using an algorithm such as that as shown in Figure 9.1:




```
create array b of size n
for ( i = 0 ; i < n ; i++ )
    b[a[i]] = a[i]
copy array b into array a
```

Figure 9.1: Simply put the items in the right order using their values.

This algorithm uses a second array `b` to hold the results, which is clearly not very memory efficient, but it is possible to do without that. One can use a series of swaps within array `a` to get the items in the right positions as shown in Figure 9.2:




```
for ( i = 0 ; i < n; i++ ) {
   while ( a[i] != i )
      swap a[a[i]] and a[i]
}
```

Figure 9.2: Swapping the items into the right order without using a new array.

As far as time complexity is concerned, it is obviously not appropriate here to count the number of comparisons. Instead, it is the number of swaps or copies that is important. The algorithm of Figure 9.1 performs n copies to fill array `b` and then another n to return the result to array a, so the overall time complexity is $O(n)$. The time complexity of the algorithm of Figure 9.2 looks worse than it really is. This algorithm performs at most $n-1$ swaps, since one item, namely `a[a[i]]` is always swapped into its final position. So at worst, this has time complexity $O(n)$ too.

This example should make it clear that in particular situations, sorting might be performed by much simpler (and quicker) means than the standard comparison sorts, though most realistic situations will not be quite as simple as the case here. Once again, it is the responsibility of the program designer to take this possibility into account.

9.16 Bin, Bucket, Radix Sorts

Bin, Bucket, and Radix Sorts are all names for essentially the same non-comparison-based sorting algorithm that works well when the items are labelled by small sets of values. For example, suppose you are given a number of dates, by day and month, and need to sort them into order. One way of doing this would be to create a queue for each day, and place the items (dates) one at a time into the right queue according to their day (without sorting them further). Then form one big queue out of these, by concatenating all the day queues starting with day 1 and continuing up to day 31. Then for the second phase, create a queue for each month, and place the dates into the right queues *in the order that they appear in the queue created by the first phase*. Again form a big queue by concatenating these month queues in order. This final queue is sorted in the intended order.

This may seem surprising at first sight, so let us consider a simple example:

[25/12, 28/08, 29/05, 01/05, 24/04, 03/01, 04/01, 25/04, 26/12, 26/04, 05/01, 20/04].

We first create and fill queues for the days as follows:

01: [01/05]
03: [03/01]
04: [04/01]
05: [05/01]
20: [20/04]
24: [24/04]
25: [25/12, 25/04]
26: [26/12, 26/04]
28: [28/08]
29: [29/05]

The empty queues are not shown – there is no need to create queues before we hit an item that belongs to them. Then concatenation of the queues gives:

[01/05, 03/01, 04/01, 05/01, 20/04, 24/04, 25/12, 25/04, 26/12, 26/04,28/08, 29/05].

Next we create and fill queues for the months that are present, giving:

01: [03/01, 04/01, 05/01]
04: [20/04, 24/04, 25/04, 26/04]
05: [01/05, 29/05]
08: [28/08]
12: [25/12, 26/12]

Finally, concatenating all these queues gives the items in the required order:

[03/01, 04/01, 05/01, 20/04, 24/04, 25/04, 26/04, 01/05, 29/05, 28/08, 25/12, 26/12].

This is called *Two-phase Radix Sorting*, since there are clearly two phases to it.

The extension of this idea to give a general sorting algorithm should be obvious: For each phase, create an ordered set of queues corresponding to the possible values, then add each item in the order they appear to the end of the relevant queue, and finally concatenate the the queues in order. Repeat this process for each sorting criterion. The crucial additional detail is that the queuing phases must be performed in the order of the significance of each criteria, with the *least significant* criteria first.

For example, if you know that your items to be sorted are all (at most) two-digit integers, you can use Radix Sort to sort them. First create and fill queues for the last digit, concatenate, then create and fill queues for the first digit, and concatenate to leave the items in sorted order. Similarly, if you know that your keys are all strings consisting of three characters, you can again apply Radix Sort. You would first queue according to the third character, then the second, and finally the first, giving a *Three phase* Radix Sort.

Note that *at no point*, does the the algorithm actually *compare* any items at all. This kind of algorithm makes use of the fact that for each phase the items are *from a strictly restricted set*, or, in other words, the items are of a particular form which is known *a priori*. The complexity class of this algorithm is $O(n)$, since at every phase, each item is dealt with precisely once, and the number of phases is assumed to be small and constant. If the *restricted sets* are small, the number of operations involved in finding the right queue for each item and placing it at the end of it will be small, but this could become significant if the sets are large. The concatenation of the queues will involve some overheads, of course, but these will be small if the sets are small and linked lists, rather than arrays, are used. One has to be careful, however, because if the total number of operations for each item exceeds $\log_2 n$, then the overall complexity is likely to be greater than the $O(n\log_2 n)$ complexity of the more efficient comparison-based algorithms. Also, if the *restricted sets* are not known in advance, and potentially large, the overheads of finding and sorting them could render Radix sort worse than using a comparison-based approach. Once again, it is the responsibility of the program designer to decide whether a given problem can be solved more efficiently with Radix Sort rather than a comparison-based sort.

Chapter 10

Hash Tables

10.1 Storing data

We have already seen a number of different ways of *storing* items in a computer: arrays and variants thereof (e.g., sorted and unsorted arrays, heap trees), linked lists (e.g., queues, stacks), and trees (e.g., binary search trees, heap trees). We have also seen that these approaches can perform quite differently when it comes to the particular tasks we expect to carry out on the items, such as insertion, deletion and searching, and that *the best way* of storing data does not exist in general, but depends on the particular application.

This chapter looks at another way of storing data, that is quite different from the ones we have seen so far. The idea is to simply put each item in an easily determined location, so we never need to search for it, and have no ordering to maintain when inserting or deleting items. This has impressive performance as far as *time* is concerned, but that advantage is payed for by needing more *space* (i.e., memory), as well as by being more complicated and therefore harder to describe and implement.

We first need to specify what we expect to be able to do with this way of storing data, without considering how it is actually implemented. In other words, we need to outline an *abstract data type*. This is similar to what you will generally do when first trying to implement a class in *Java*: You should think about the operations you wish to perform on the objects of that class. You may also want to specify a few variables that you know will definitely be needed for that class, but this does not usually come into defining an abstract data type. The approach we have been following for defining abstract data types in these notes is by describing the crucial operations in plain English, trusting that they are simple enough to not need further explanations. In general, what is needed is a *specification* for the abstract data type in question. An important aspect of studying software engineering is to learn about and use more formal approaches to this way of operating.

After we have decided what our specification is, we then need to choose a *data structure* in order to *implement* the abstract data type. The data structure to be considered in this chapter is a particular type of *table* known as a *hash table*.

10.2 The Table abstract data type

The specification of the *table* abstract data type is as follows:

1. A table can be used to store objects, for example

012	Johnny	English	Spy
007	James	Bond	Spy
583	Alex	Rider	Spy
721	Sherlock	Holmes	Detective
722	James	Moriarty	Villain

2. The objects can be arbitrarily complicated. However, for our purposes, the only relevant detail is that each object has a unique *key*, and that their keys can be compared for equality. The keys are used in order to identify objects in much the way we have done for searching and sorting.

3. We assume that there are methods or procedures for:

   (a) determining whether the table is empty or full;

   (b) inserting a new object into the table, provided the table is not already full;

   (c) given a key, retrieving the object with that key;

   (d) given a key, updating the item with that key (usually by replacing the item with a new one with the same key, which is what we will assume here, or by overwriting some of the item's variables);

   (e) given a key, deleting the object with that key, provided that object is already stored in the table;

   (f) listing or traversing all the items in the table (if there is an order on the keys then we would expect this to occur in increasing order).

Notice that we are assuming that each object is uniquely identified by its key.

In a programming language such as *Java*, we could write an interface for this abstract data type as follows, where we assume here that keys are objects of a class we call `Key` and we have records of a class called `Record`:

```
interface Table {
   Boolean isEmpty();
   Boolean isFull();
   void Insert(Record);
   Record Retrieve(Key);
   void Update(Record);
   void Delete{Key};
   void Traverse();
}
```

Note that we have not fixed how exactly the storage of records should work – that is something that comes with the *implementation*. Also note that you could give an interface to somebody else, who could then write a program which performs operations on tables *without ever knowing how they are implemented*. You could certainly carry out all those operations with binary search trees and sorted or unsorted arrays if you wished. The former even has the advantage that a binary search tree never becomes full as such, because it is only limited by the size of the memory.

This general approach follows the sensible and commonly used way to go about defining a *Java* class: First think about what it is you want to do with the class, and only then wonder

about how exactly you might implement the methods. Thus, languages such as *Java* support mechanisms for defining abstract data types. But notice that, as opposed to a specification in plain English, such as the above, a definition of an interface is only a partial specification of an abstract data type, because it does not explain *what* the methods are supposed to do; it only explains how they are called.

10.3 Implementations of the table data structure

There are three key approaches for implementing the table data structure. The first two we have studied already, and the third is the topic of this chapter:

Implementation via sorted arrays. Let us assume that we want to implement the table data structure using a sorted *array*. Whether it is full or empty can easily be determined in constant time if we have a variable for the size. Then to insert an element we first have to find its proper position, which will take on average the same time as finding an element. To find an element (which is necessary for all other operations apart from traversal), we can use binary search as described in in Section 4.4, so this takes $O(\log_2 n)$. This is also the complexity for retrieval and update. However, if we wish to delete or insert an item, we will have to shift what is 'to the right' of the location in question by one, either to the left (for deletion) or to the right (for insertion). This will take on average $n/2$ steps, so these operations have $O(n)$ complexity. Traversal in order is simple, and is of $O(n)$ complexity as well.

Implementation via binary search trees A possible alternative implementation would involve using binary search trees. However, we know already that in the worst case, the tree can be very deep and narrow, and that these trees will have linear complexity when it comes to looking up an entry. We have seen that there is a variant of binary search trees which keeps the worst case the same as the average case, the so-called *self-balancing binary search tree*, but that is more complicated to both understand and program. For those trees, insertion, deletion, search, retrieval and update, can all be done with time complexity $O(\log_2 n)$, and traversal has $O(n)$ complexity.

Implementation via Hash tables The idea here is that, at the expense of using more space than strictly needed, we can speed up the table operations. The remainder of this chapter will describe how this is done, and what the various computational costs are.

10.4 Hash Tables

The underlying idea of a *hash table* is very simple, and quite appealing: Assume that, given a key, there was a way of jumping straight to the entry for that key. Then we would never have to search at all, we could just go there! Of course, we still have to work out a way for that to be achieved. Assume that we have an array `data` to hold our entries. Now if we had a function `h(k)` that maps each key `k` to the index (an integer) where the associated entry will be stored, then we could just look up `data[h(k)]` to find the entry with the key `k`.

It would be easiest if we could just make the data array big enough to hold *all* the keys that might appear. For example, if we knew that the keys were the numbers from 0 to 99, then we could just create an array of size 100 and store the entry with key 67 in `data[67]`,

and so on. In this case, the function `h` would be the identity function `h(k) = k`. However, this idea is not very practical if we are dealing with a relatively small number of keys out of a huge collection of possible keys. For example, many American companies use their employees' 9-digit social security number as a key, even though they have nowhere near 10^9 employees. British National Insurance Numbers are even worse, because they are just as long and usually contain a mixture of letters and numbers. Clearly it would be very inefficient, if not impossible, to reserve space for all 10^9 social security numbers which might occur.

Instead, we use a non-trivial function `h`, the so-called *hash function*, to map the space of possible keys to the set of indices of our array. For example, if we had to store entries about 500 employees, we might create an array with 1000 entries and use three digits from their social security number (maybe the first or last three) to determine the place in the array where the records for each particular employee should be stored.

This approach sounds like a good idea, but there is a pretty obvious problem with it: What happens if two employees happen to have the same three digits? This is called a *collision* between the two keys. Much of the remainder of this chapter will be spent on the various strategies for dealing with such collisions.

First of all, of course, one should try to avoid collisions. If the keys that are likely to actually occur are not evenly spread throughout the space of all possible keys, particular attention should be paid to choosing the hash function `h` in such a way that collisions among them are less likely to occur. If, for example, the first three digits of a social security number had geographical meaning, then employees are particularly likely to have the three digits signifying the region where the company resides, and so choosing the first three digits as a hash function might result in many collisions. However, that problem might easily be avoided by a more prudent choice, such as the last three digits.

10.5 Collision likelihoods and load factors for hash tables

One might be tempted to assume that collisions do not occur very often when only a small subset of the set of possible keys is chosen, but this assumption is mistaken.

The von Mises birthday paradox. As an example, consider a collection of people, and a hash function that gives their birthdays as the number of the day in the year, i.e. 1st January is 1, 2nd January is 2, ..., 31st December is 365. One might think that if all we want to do is store a modest number of 24 people in this way in an array with 365 locations, collisions will be rather unlikely. However, it turns out that the probability of collision is bigger than 50%. This is so surprising at first sight that this phenomenon has become known as the *von Mises birthday paradox*, although it is not really a paradox in the strict sense.

It is easy to understand what is happening. Suppose we have a group of n people and want to find out how likely it is that two of them have the same birthday, assuming that the birthdays are uniformly distributed over the 365 days of the year. Let us call this probability $p(n)$. It is actually easier to first compute the probability $q(n)$ that no two of them share a birthday, and then $p(n) = 1 - q(n)$. For $n = 1$ this probability is clearly $q(1) = 1$. For $n = 2$ we get $q(2) = 364/365$ because, for the added second person, 364 of the 365 days are not the birthday of the first person. For $n = 3$ we get

$$q(3) = \frac{365 \cdot 364 \cdot 363}{365^3} = 1 - p(3)$$

and for the general $n > 1$ case we have

$$q(n) = \frac{365 \cdot 364 \cdot 363 \cdots (365 - n + 1)}{365^n} = \frac{365!}{365^n (365 - n)!} = 1 - p(n)$$

It may be surprising that $p(22) = 0.476$ and $p(23) = 0.507$, which means that as soon as there are more than 22 people in a group, it is more likely that two of them share a birthday than not. Note that in the real world, the distribution of birthdays over the year is not precisely uniform, but this only increases the probability that two people have the same birthday. In other words, birthday collisions are much more likely than one might think at first.

Implications for hash tables. If 23 random locations in a table of size 365 have more than a 50% chance of overlapping, it seems inevitable that collisions will occur in any hash table that does not waste an enormous amount of memory. And collisions will be even more likely if the hash function does not distribute the items randomly throughout the table.

To compute the computational efficiency of a hash table, we need some way of quantifying how full the table is, so we can compute the probability of collisions, and hence determine how much effort will be required to deal with them.

The load factor of a hash table. Suppose we have a hash table of *size* m, and it currently has n entries. Then we call $\lambda = n/m$ the *load factor* of the hash table. This load factor is the obvious way of describing how full the table currently is: A hash table with load factor 0.25 is 25% full, one with load factor 0.50 is 50% full, and so on. Then if we have a hash table with load factor λ, the probability that a collision occurs for the next key we wish to insert is λ. This assumes that each key from the key space is equally likely, and that the hash function `h` spreads the key space evenly over the set of indices of our array. If these optimistic assumptions fail, then the probability may be even higher.

Therefore, to minimize collisions, it is prudent to keep the load factor low. Fifty percent is an often quoted good maximum figure, while beyond an eighty percent load the performance deteriorates considerably. We shall see later exactly what effect the table's load factor has on the speed of the operations we are interested in.

10.6 A simple Hash Table in operation

Let us assume that we have a small data array we wish to use, of size 11, and that our set of possible keys is the set of 3-character strings, where each character is in the range from A to Z. Obviously, this example is designed to illustrate the principle – typical real-world hash tables are usually very much bigger, involving arrays that may have a size of thousands, millions, or tens of millions, depending on the problem.

We now have to define a hash function which maps each string to an integer in the range 0 to 10. Let us consider one of the many possibilities. We first map each string to a number as follows: each character is mapped to an integer from 0 to 25 using its place in the alphabet (A is the first letter, so it goes to 0, B the second so it goes to 1, and so on, with Z getting value 25). The string $X_1X_2X_3$ therefore gives us three numbers from 0 to 25, say k_1, k_2, and k_3. We can then map the whole string to the number calculated as

$$k = k_1 * 26^2 + k_2 * 26^1 + k_3 * 26^0 = k_1 * 26^2 + k_2 * 26 + k_3.$$

That is, we think of the strings as coding numbers in base 26.

Now it is quite easy to go from any *number* k (rather than a string) to a number from 0 to 10. For example, we can take the remainder the number leaves when divided by 11. This is the *C* or *Java* modulus operation `k % 11`. So our hash function is

$$h(X_1X_2X_3) = (\mathtt{k}_1 * 26^2 + \mathtt{k}_2 * 26 + \mathtt{k}_3)\%11 = \mathtt{k}\%11.$$

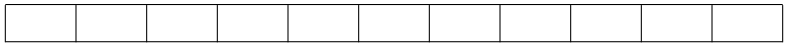
This *modulo* operation, and *modular arithmetic* more generally, are widely used when constructing good hash functions.

As a simple example of a hash table in operation, assume that we now wish to insert the following three-letter airport acronyms as keys (in this order) into our hash table: PHL, ORY, GCM, HKG, GLA, AKL, FRA, LAX, DCA. To make this easier, it is a good idea to start by listing the values the hash function takes for each of the keys:

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA
$h(X_1X_2X_3)$	4	8	6	4	8	7	5	1	1

It is clear already that we will have hash collisions to deal with.

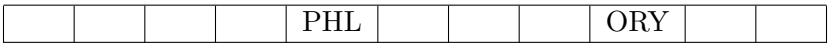
We naturally start off with an empty table of the required size, i.e. 11:



Clearly we have to be able to tell whether a particular location in the array is still empty, or whether it has already been filled. We can assume that there is a *unique* key or entry (which is *never* associated with a record) which denotes that the position has not been filled yet. However, for clarity, this key will not appear in the pictures we use.

Now we can begin inserting the keys in order. The number associated with the first item PHL is 4, so we place it at index 4, giving:

Next is ORY, which gives us the number 8, so we get:



Then we have GCM, with value 6, giving:

Then HKG, which also has value 4, results in our first collision since the corresponding position has already been filled with PHL. Now we could, of course, try to deal with this by simply saying the table is full, but this gives such poor performance (due to the frequency with which collisions occur) that it is unacceptable.

10.7 Strategies for dealing with collisions

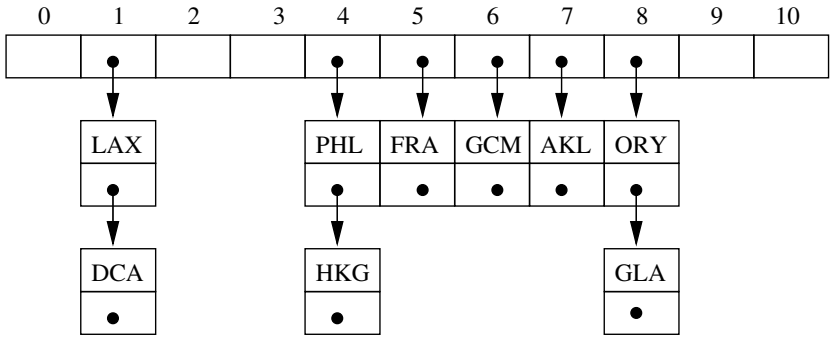
We now look at three standard approaches, of increasing complexity, for dealing with hash collisions:

Buckets. One obvious option is to reserve a two-dimensional array from the start. We can think of each column as a bucket in which we throw all the elements which give a particular result when the hash function is supplied, so the fifth column contains all the keys for which the hash function evaluates to 4. Then we could put HKG into the slot 'beneath' PHL, and GLA in the one beneath ORY, and continue filling the table in the order given until we reach:

0	1	2	3	4	5	6	7	8	9	10
	LAX			PHL	FRA	GCM	AKL	ORY		
	DCA			HKG				GLA		

The disadvantage of this approach is that it has to reserve quite a bit more space than will be eventually required, since it must take into account the likely maximal number of collisions. Even while the table is still quite empty overall, collisions will become increasingly likely. Moreover, when searching for a particular key, it will be necessary to search the entire column associated with its expected position, at least until an empty slot is reached. If there is an order on the keys, they can be stored in ascending order, which means we can use the more efficient binary search rather than linear search, but the ordering will have an overhead of its own. The average complexity of searching for a particular item depends on how many entries in the array have been filled already. This approach turns out to be slower than the other techniques we shall consider, so we shall not spend any more time on it, apart from noting that it does prove useful when the entries are held in slow external storage.

Direct chaining. Rather than reserving entire sub-arrays (the columns above) for keys that collide, one can instead create a linked list for the set of entries corresponding to each key. The result for the above example can be pictured something like this:



This approach does not reserve any space that will not be taken up, but has the disadvantage that in order to find a particular item, lists will have to be traversed. However, adding the hashing step still speeds up retrieval considerably.

We can compute the size of the average non-empty list occurring in the hash table as follows. With n items in an array of size m, the probability than no items land in a particular slot is $q(n,m) = (\frac{m-1}{m})^n$. So the number of slots with at least one item falling in it is

$$N(n,m) = m.\Big(1 - q(n,m)\Big) = m.\Big(1 - (\frac{m-1}{m})^n\Big)$$

and since there are n items altogether, the average number of items in a non-empty list is:

$$k(n,m) = \frac{n}{N(n,m)} = \frac{n}{m.\Big(1 - (\frac{m-1}{m})^n\Big)}.$$

Then a linear search for an item in a list of size k takes on average

$$\frac{1}{k}\Big(1 + 2 + \cdots + k\Big) = \frac{k(k+1)}{2k} = \frac{k+1}{2}$$

comparisons. It is difficult to visualize what these formulae mean in practice, but if we assume the hash table is large but not overloaded, i.e. n and m are both large with $n \ll m$, we can perform a Taylor approximation for small loading factor $\lambda = n/m$. That shows there are

$$\frac{k+1}{2} = 1 + \frac{\lambda}{4} + \frac{\lambda^2}{24} + O(\lambda^3)$$

comparisons on average for a successful search, i.e. that this has $O(1)$ complexity.

For an unsuccessful search, we need the average list size including the empty slots. That will clearly be $n/m = \lambda$, and so in an unsuccessful search the average number of comparisons made to decide the item in question is not present will be λ, which is again $O(1)$.

Thus, neither the successful nor unsuccessful search times depend on the number of keys in the table, but only on the load factor, which can be kept low by choosing the size of the hash table to be big enough. Note also that insertion is done even more speedily, since all we have to do is to insert a new element at the front of the appropriate list. Hence, apart from traversal, the complexity class of all operations is *constant*, i.e. $O(1)$. For traversal, we need to sort the keys, which can be done in $O(n\log_2 n)$, as we know from Chapter 9. A variant would be to make each linked list sorted, which will speed up finding an item, as well as speed up traversal slightly, although this will not put either operation into a different complexity class. This speed-up would be paid for by making the insertion operation more expensive, i.e. take slightly longer, but it will still have constant complexity.

Overall, all the time complexities for this approach are clearly very impressive compared to those for sorted arrays or (balanced) binary search trees.

Open addressing. The last fundamentally different approach to collision avoidance is called *open addressing*, and that involves finding another open location for any entry which cannot be placed where its hash function points. We refer to that position as a key's *primary position* (so in the earlier example, ORY and GLA have the same primary position). The easiest strategy for achieving this is to search for open locations by simply decreasing the index considered by one until we find an empty space. If this reaches the beginning of the array, i.e. index 0, we start again at the end. This process is called *linear probing*. A better approach is to search for an empty location using a *secondary hash function*. This process is called *double hashing*. We will now look at both of these approaches in some detail.

10.8 Linear Probing

We now proceed with the earlier example using *linear probing*. We had reached the stage:

				PHL		GCM		ORY		

and then wanted to put HKG at index 4, where we found PHL.

Linear probing reduces the index by one to 3, and finds an empty location in that position, so we put HKG there giving:

			HKG	PHL		GCM		ORY		

Next we wish to insert GLA, with hash value 8, but the location with that index is already filled by ORY. Again linear probing reduces the index by one, and since that slot one to the left is free, we insert GLA there:

			HKG	PHL		GCM	GLA	ORY		

Then we have AKL, and although we have not had the value 7 before, the corresponding location is filled by GLA. So we try the next index down, but that contains GCM, so we continue to the next one at index 5 which s empty, so we put AKL there:

			HKG	PHL	AKL	GCM	GLA	ORY		

We now continue in the same way with the remaining keys, eventually reaching:

DCA	LAX	FRA	HKG	PHL	AKL	GCM	GLA	ORY		

This looks quite convincing - all the keys have been inserted in a way that seems to make good use of the space we have reserved.

However, what happens now if we wish to find a particular key? It will no longer be good enough to simply apply the hash function to it and check there. Instead, we will have to follow its possible insertion locations until we hit an empty one, which tells us that the key we were looking for is not present, after all, because it would have been inserted there. This is why every hash table that uses open addressing should have at least one empty slot at any time, and be declared full when only one empty location is left. However, as we shall see, hash tables lose much of their speed advantage if they have a high load factor, so as a matter of policy, many more locations should be kept empty.

So, to find the key AKL, we would first check at index 7, then at 6, and 5, where we are successful. Searching for JFK, on the other hand, we would start with its proper position, given by the hash function value 8, so we would check indices 8, 7, 6, . . . , 1, 0, 10 in that order until we find an empty space which tells us that JFK is, in fact, not present at all. This looks pretty bad at first sight, but bear in mind that we said that we will aim towards keeping the load factor at around 50 percent, so there would be many more empty slots which effectively stop any further search.

But this idea brings another problem with it. Suppose we now delete GCM from the table and then search for AKL again. We would find the array empty at index 6 and stop searching, and therefore wrongly conclude that AKL is not present. This is clearly not acceptable, but equally, we do not wish to have to search through the entire array to be sure that an entry is not there. The solution is that we reserve another key to mean that a position is empty, but that it did hold a key at some point. Let us assume that we use the character '!' for that. Then after deleting GCM, the array would be:

DCA	LAX	FRA	HKG	PHL	AKL	!	GLA	ORY		

and when searching for AKL we would know to continue beyond the exclamation mark. If, on the other hand, we are trying to *insert* a key, then we can ignore any exclamation marks and fill the position once again. This now does take care of all our problems, although if we do a lot of deleting and inserting, we will end up with a table which is a bit of a mess. A large number of exclamation marks means that we have to keep looking for a long time to find a particular entry despite the fact that the load factor may not be all that high. This happens if deletion is a frequent operation. In such cases, it may be better to re-fill a new hash table again from scratch, or use another implementation.

Search complexity. The complexity of open addressing with linear probing is rather difficult to compute, so we will not attempt to present a full account of it here. If λ is once again the load factor of the table, then a successful search can be shown to take $\frac{1}{2}(1 + \frac{1}{1-\lambda})$ comparisons on average, while an unsuccessful search takes approximately $\frac{1}{2}(1 + \frac{1}{(1-\lambda)^2})$. For relatively small load factors, this is quite impressive, and even for larger ones, it is not bad. Thus, the hash table time complexity for search is again constant, i.e. $O(1)$.

Clustering. There is a particular problem with linear probing, namely what is known as *primary* and *secondary clustering*. Consider what happens if we try to insert two keys that have the same result when the hash function is applied to them. Take the above example with hash table at the stage where we just inserted GLA:

			HKG	PHL		GCM	GLA	ORY		

If we next try to insert JFK we note that the hash function evaluates to 8 once again. So we keep checking *the same* locations we only just checked in order to insert GLA. This seems a rather inefficient way of doing this. This effect is known as *primary clustering* because the new key JFK will be inserted close to the previous key with the same primary position, GLA. It means that we get a continuous 'block' of filled slots, and whenever we try to insert any key which is sent into the block by the hash function, we will have to test all locations until we hit the end of the block, and then make such block even bigger by appending another entry at its end. So these blocks, or clusters, keep growing, not only if we hit the same primary location repeatedly, but also if we hit anything that is part of the same cluster. The last effect is called *secondary clustering*. Note that searching for keys is also adversely affected by these clustering effects.

10.9 Double Hashing

The obvious way to avoid the clustering problems of *linear probing* is to do something slightly more sophisticated than trying every position to the left until we find an empty one. This is known as *double hashing*. We apply a *secondary hash function* to tell us how many slots to jump to look for an empty slot if a key's *primary position* has been filled already.

Like the primary hash function, there are many possible choices of the secondary hash function. In the above example, one thing we could do is take the same number k associated with the three-character code, and use the result of integer division by 11, instead of the remainder, as the secondary hash function. However, the resulting value might be bigger than 10, so to prevent the jump looping round back to, or beyond, the starting point, we *first* take

the result of integer division by 11, and *then* take the remainder this result leaves when again divided by 11. Thus we would like to use as our secondary hash function $h_2(n) = (\texttt{k/11})\%\texttt{11}$. However, this has yet another problem: it might give zero at some point, and we obviously cannot test 'every zeroth location'. An easy solution is to simply make the secondary hash function one if the above would evaluate to zero, that is:

$$h_2(n) = \begin{cases} (\texttt{k/11})\%\texttt{11} & \text{if } (\texttt{k/11})\%\texttt{11} \neq \texttt{0}, \\ 1 & \text{otherwise.} \end{cases}$$

The values of this for our example set of keys are given in the following table:

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA	BHX
$h_2(X_1X_2X_3)$	4	1	1	3	9	2	6	7	2	3

We can then proceed from the situation we were in when the first collision occurred:

				PHL		GCM		ORY		

with HKG the next key to insert, which gives a collision with PHL. Since $h_2(\text{HKG}) = 3$ we now try *every third location* to the left in order to find a free slot:

	HKG			PHL		GCM		ORY		

Note that this did not create a block. When we now try to insert GLA, we once again find its primary location blocked by ORY. Since $h_2(\text{GLA}) = 9$, we now try every ninth location. Counting to the left from ORY, that gets us (starting again from the back when we reach the first slot) to the last location overall:

	HKG			PHL		GCM		ORY		GLA

.

Note that we still have not got any blocks, which is good. Further note that most keys which share the same primary location with ORY and GLA will follow a different route when trying to find an empty slot, thus avoiding primary clustering. Here is the result when filling the table with the remaining keys given:

	HKG	DCA		PHL	FRA	GCM	AKL	ORY	LAX	GLA

Our example is too small to show convincingly that this method also avoids secondary clustering, but in general it does.

It is clear that the trivial secondary hash function $h_2(n) = 1$ reduces this approach to that of linear probing. It is also worth noting that, in both cases, proceeding to secondary positions *to the left* is merely a convention – it could equally well be *to the right* – but obviously it has to be made clear which direction has been chosen for a particular hash table.

Search complexity. The efficiency of double hashing is even more difficult to compute than that of linear probing, and therefore we shall just give the results without a derivation. With load factor λ, a successful search requires $(1/\lambda)\ln(1/(1-\lambda))$ comparisons on average, and an unsuccessful one requires $1/(1-\lambda)$. Note that it is the natural logarithm (to base $e = 2.71828\ldots$) that occurs here, rather than the usual logarithm to base 2. Thus, the hash table time complexity for search is again constant, i.e. $O(1)$.

10.10 Choosing good hash functions

In principle, any convenient function can be used as a primary hash function. However, what is important when choosing a *good* hash function is to make sure that it spreads the space of possible keys onto the set of hash table indices as evenly as possible, or more collisions than necessary will occur. Secondly, it is advantageous if any potential clusters in the space of possible keys are broken up (something that the remainder in a division will *not* do), because in that case we could end up with a 'continuous run' and associated clustering problems in the hash table. Therefore, when defining hash functions of strings of characters, it is never a good idea to make the last (or even the first) few characters decisive.

When choosing secondary hash functions, in order to avoid primary clustering, one has to make sure that different keys with the same primary position give *different* results when the secondary hash function is applied. Secondly, one has to be careful to ensure that the secondary hash function cannot result in a number which has a common divisor with the size of the hash table. For example, if the hash table has size 10, and we get a secondary hash function which gives 2 (or 4, 6 or 8) as a result, then only *half* of the locations will be checked, which might result in failure (an endless loop, for example) while the table is still half empty. Even for large hash tables, this can still be a problem if the secondary hash keys can be similarly large. A simple remedy for this is to always make the size of the hash table a prime number.

10.11 Complexity of hash tables

We have already seen that insert, search and delete all have $O(1)$ time complexity if the load factor of the hash table is kept reasonably low, e.g. below 0.5, but having higher load factors can considerably slow down the operations.

The crucial search time complexity of a particular form of hash table is determined by counting the average number of location checks that are needed when searching for items in the table when it has a particular load factor, and that will depend on whether the item is found. The following table shows the average number of locations that need to be checked to conduct successful and unsuccessful searches in hash tables with different collision handling strategies, depending on the load factor given in the top row. It shows how the different approaches and cases vary differently as the table becomes closer to fully loaded.

Strategy	0.10	0.25	0.50	0.75	0.90	0.99
Successful Search						
Direct chaining	1.05	1.12	1.25	1.37	1.45	1.48
Linear probing	1.06	1.17	1.50	2.50	5.50	50.50
Double hashing	1.05	1.15	1.39	1.85	2.56	4.65
Unsuccessful search						
Direct chaining	0.10	0.25	0.50	0.75	0.90	0.99
Linear probing	1.12	1.39	2.50	8.50	50.50	5000.00
Double hashing	1.11	1.33	2.00	4.00	10.00	100.00

It also shows the considerable advantage that double hashing has over linear probing, particularly when the load factors become large. Whether or not double hashing is preferable to

direct chaining (which appears far superior, but is generally more complex to implement and maintain) is dependent on the circumstances.

The following table shows a comparison of the average time complexities for the different possible implementations of the `table` interface:

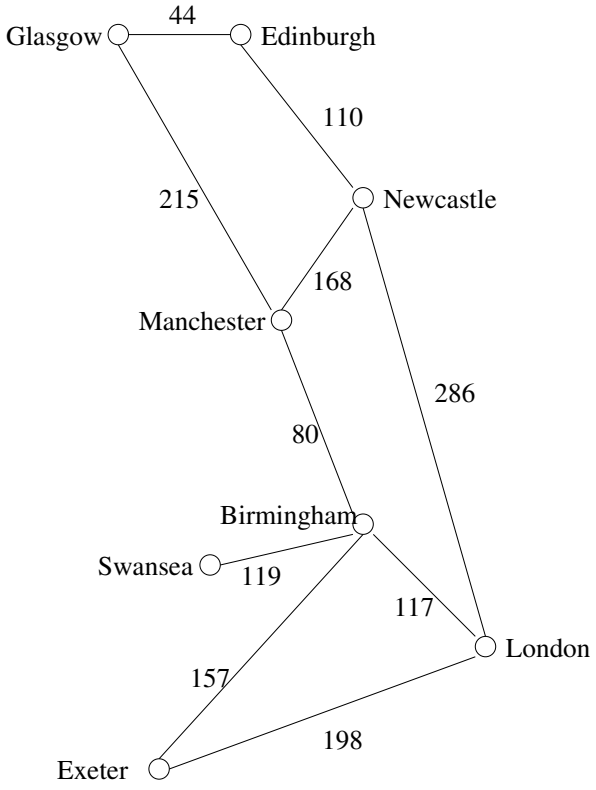
	Search	Insert	Delete	Traverse
Sorted array	$O(\log_2 n)$	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$
Hash table	$O(1)$	$O(1)$	$O(1)$	$O(n\log_2 n)$

Hash tables are seen to perform rather well: the complexity of searching, updating and retrieving are all independent of table size. In practice, however, when deciding what approach to use, it will depend on the mix of operations typically performed. For example, lots of repeated deletions and insertions can cause efficiency problems with some hash table strategies, as explained above. To give a concrete example, if there are 4096 entries in a balanced binary search tree, it takes on average 12.25 comparisons to complete a successful search. On the other hand, we can need as few as 1.39 comparisons if we use a hash table, provided that we keep its load factor below 50 percent. Of course, despite their time advantage, we should never forget that hash tables have a considerable disadvantage in terms of the memory required to implement them efficiently.

Chapter 11

Graphs

Often it is useful to represent information in a more general graphical form than considered so far, such as the following representation of the distances between towns:



With similar structures (maybe leaving out the distances, or replacing them by something else), we could represent many other situations, like an underground tunnel network, or a network of pipes (where the number label might give the pipe diameters), or a railway map, or an indication of which cities are linked by flights, or ferries, or political alliances. Even if we assume it is a network of paths or roads, the numbers do not necessarily have to represent distances, they might be an indication of how long it takes to cover the distance in question on foot, so a given distance up a steep hill would take longer than on even ground.

There is much more that can be done with such a picture of a situation than just reading off which place is directly connected with another place: For example, we can ask ourselves

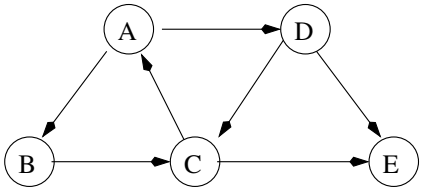
whether there is a way of getting from A to B at all, or what is the shortest path, or what would be the shortest set of pipes connecting all the locations. There is also the famous *Travelling Salesman Problem* which involves finding the shortest route through the structure that visits each city precisely once.

11.1 Graph terminology

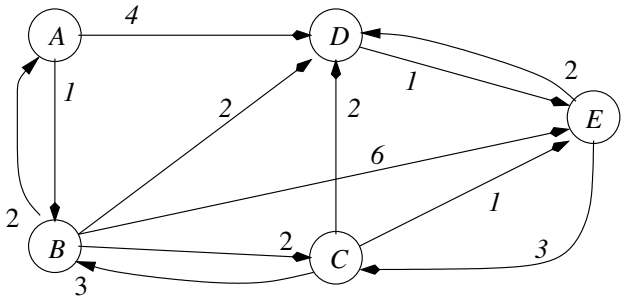
The kind of structure in the above figure is known formally as a *graph*. A graph consists of a series of *nodes* (also called *vertices* or *points*), displayed as nodes, and *edges* (also called *lines*, *links* or, in directed graphs, *arcs*), displayed as *connections* between the nodes. There exists quite a lot of terminology that allows us to specify graphs precisely:

A graph is said to be *simple* if it has no *self-loops* (i.e., edges connected at both ends to the same vertex) and no more than one edge connecting any pair of vertices. The remainder of this Chapter will assume that, which is sufficient for most practical applications.

If there are labels on the edges (usually non-negative real numbers), we say that the graph is *weighted*. We distinguish between *directed* and *undirected* graphs. In directed graphs (also called *digraphs*), each edge comes with one or two directions, which are usually indicated by arrows. Think of them as representing roads, where some roads may be one-way only. Or think of the associated numbers as applying to travel in one way only: such as going up a hill which takes longer than coming down. An example of an unweighted digraph is:



and an example of a *weighted* digraph, because it has labels on its edges, is:



In undirected graphs, we assume that every edge can be viewed as going both ways, that is, an edge between A and B goes from A to B as well as from B to A. The first graph given at the beginning of this chapter is weighted and undirected.

A *path* is a sequence of nodes or vertices $v_1, v_2, \ldots, v_n$ such that v_i and v_{i+1} are connected by an edge for all $1 \leq i \leq n-1$. Note that in a directed graph, the edge from v_i to v_{i+1} is the one which has the corresponding direction. A *circle* is a non-empty path whose first vertex is the same as its last vertex. A path is *simple* if no vertex appears on it twice (with the exception of a circle, where the first and last vertex may be the same – this is because we have to 'cut open' the circle at some point to get a path, so this is inevitable).

An undirected graph is *connected* if every pair of vertices has a path connecting them. For directed graphs, the notion of connectedness has two distinct versions: We say that a digraph is *weakly connected* if for every two vertices A and B there is either a path from A to B or a path from B to A. We say it is *strongly connected* if there are paths leading both ways. So, in a weakly connected digraph, there may be two vertices i and j such that there exists no path from i to j.

A graph clearly has many properties similar to a *tree*. In fact, any tree can be viewed as a simple graph of a particular kind, namely one that is connected and contains no circles. Because a graph, unlike a tree, does not come with a natural 'starting point' from which there is a unique path to each vertex, it does not make sense to speak of parents and children in a graph. Instead, if two vertices A and B are connected by an edge e, we say that they are *neighbours*, and the edge connecting them is said to be *incident* to A and B. Two edges that have a vertex in common (for example, one connecting A and B and one connecting B and C) are said to be *adjacent*.

11.2 Implementing graphs

All the data structures we have considered so far were designed to hold certain information, and we wanted to perform certain actions on them which mostly centred around inserting new items, deleting particular items, searching for particular items, and sorting the collection. At no time was there ever a *connection* between all the items represented, apart from the order in which their keys appeared. Moreover, that connection was never something that was inherent in the structure and that we therefore tried to represent somehow – it was just a property that we used to store the items in a way which made sorting and searching quicker. Now, on the other hand, it is the connections that are the crucial information we need to encode in the data structure. We are *given* a structure which comes with specified connections, and we need to design an implementation that efficiently keeps track of them.

Array-based implementation. The first underlying idea for *array*-based implementations is that we can conveniently rename the vertices of the graph so that they are labelled by non-negative integer indices, say from 0 to $n-1$, if they do not have these labels already. However, this only works if the graph is given *explicitly*, that is, if we know in advance how many vertices there will be, and which pairs will have edges between them. Then we only need to keep track of which vertex has an edge to which other vertex, and, for weighted graphs, what the weights on the edges are. For unweighted graphs, we can do this quite easily in an $n \times n$ two-dimensional binary array `adj`, also called a *matrix*, the so-called *adjacency matrix*. In the case of weighted graphs, we instead have an $n \times n$ *weight matrix* `weights`. The array/matrix representations for the two example graphs shown above are then:

		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	0	1	0
B	1	0	0	1	0	0
C	2	1	0	0	0	1
D	3	0	0	1	0	1
E	4	0	0	0	0	0

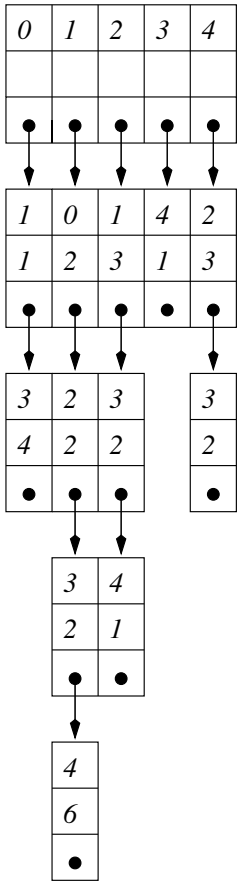
		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	∞	4	∞
B	1	2	0	2	2	6
C	2	∞	3	0	2	1
D	3	∞	∞	∞	0	1
E	4	∞	∞	3	2	0

In the first case, for the unweighted graph, a '0' in position `adj[i][j]` reads as `false`, that is, there is no edge from the vertex i to the vertex j. A '1', on the other hand, reads as `true`, indicating that there is an edge. It is often useful to use boolean values here, rather than the numbers 0 and 1, because it allows us to carry out operations on the booleans. In the second case, we have a weighted graph, and we have the real-valued weights in the matrix instead, using the infinity symbol ∞ to indicate when there is no edge.

For an undirected graph, if there is a 1 in the ith column and the jth row, we know that there is an edge from vertex i to the vertex with the number j, which means there is also an edge from vertex j to vertex i. This means that `adj[i][j] == adj[j][i]` will hold for all i and j from 0 to $n-1$, so there is some redundant information here. We say that such a matrix is *symmetric* – it equals its mirror image along the main diagonal.

Mixed implementation. There is a potential problem with the adjacency/weight matrix representation: If the graph has very many vertices, the associated array will be extremely large (e.g., 10,000 entries are needed if the graph has just 100 vertices). Then, if the graph is *sparse* (i.e., has relatively few edges), the adjacency matrix contains many 0s and only a few 1s, and it is a waste of space to reserve so much memory for so little information.

A solution to this problem is to number all the vertices as before, but, rather than using a two-dimensional array, use a one-dimensional array that points to a linked list of neighbours for each vertex. For example, the above weighted graph can be represented as follows, with each triple consisting of a vertex name, connection weight and pointer to the next triple:



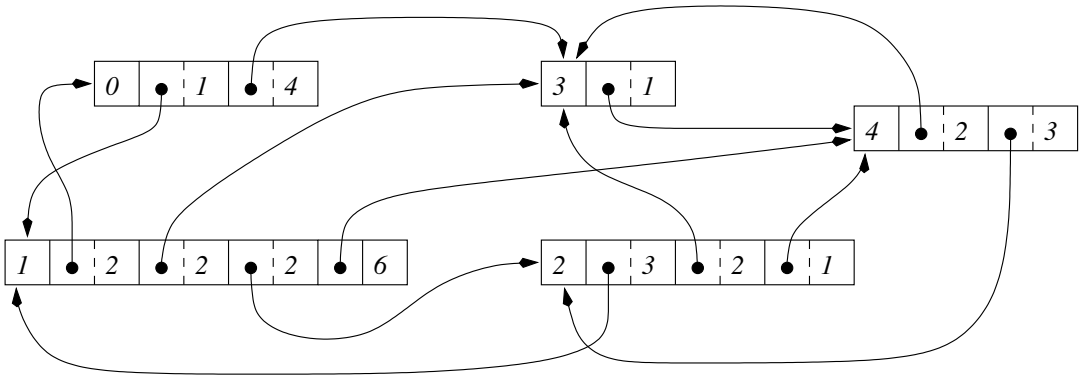
If there are very few edges, we will have very short lists at each entry of the array, thus saving space over the adjacency/weight matrix representation. This implementation is using so-called *adjacency lists*. Note that if we are considering undirected graphs, there is still a certain amount of redundancy in this representation, since every edge is represented twice, once in each list corresponding to the two vertices it connects. In *Java*, this could be accomplished with something like:

```
class Graph {
   Vertex[] heads;
   private class Vertex {
      int name;
      double weight;
      Vertex next;
      ...//methods for vertices
   }
   ...//methods for graphs
}
```

Pointer-based implementation. The standard *pointer*-based implementation of binary trees, which is essentially a generalization of linked lists, can be generalized for graphs. In a language such as Java, a class `Graph` might have the following as an internal class:

```
class Vertex {
   string name;
   Vertex[] neighbours;
   double[] weights;
}
```

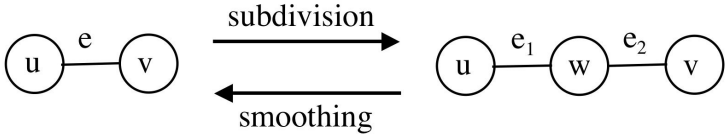
When each vertex is created, an array `neighbours` big enough to accommodate (pointers to) all its neighbours is allocated, with (for weighted graphs) an equal sized array `weights` to accommodate the associated weights. We then place the neighbours of each vertex into those arrays in some arbitrary order. Any entries in the `neighbours` array that are not needed will hold a `null` pointer as usual. For example, the above weighted graph would be represented as follows, with each weight shown following the associated pointer:



11.3 Relations between graphs

Many important theorems about graphs rely on formal definitions of the relations between them, so we now define the main relevant concepts. Two graphs are said to be *isomorphic* if they contain the same number of vertices with the same pattern of adjacency, i.e. there is a *bijection* between their vertices which preserves the adjacency relations. A *subgraph* of a graph G is defined as any graph that has a vertex set which is a subset of that of G, with adjacency relations which are a subset of those of G. Conversely, a *supergraph* of a graph G is defined as any graph which has G as a subgraph. Finally, a graph G is said to *contain* another graph H if there exists a subgraph of G that is either H or isomorphic to H.

A *subdivision* of an edge e with endpoints u and v is simply the pair of edges e_1, with endpoints u and w, and e_2, with endpoints w and v, for some new vertex w. The reverse operation of *smoothing* removes a vertex w with exactly two edges e_1 and e_2, leaving an edge e connecting the two adjacent vertices u and v:



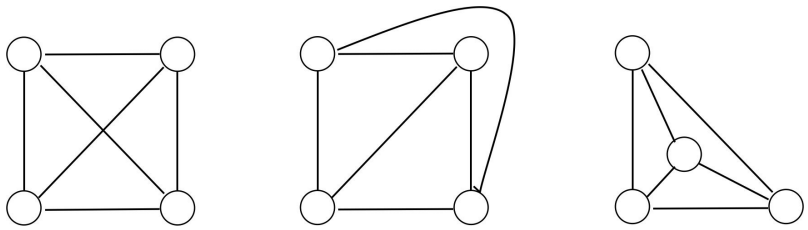
A *subdivision* of a graph G can be defined as a graph resulting from the subdivision of edges in G. Two graphs G and H can then be defined as being *homeomorphic* if there is a graph isomorphism from some subdivision of G to some subdivision of H.

An *edge contraction* removes an edge from a graph and merges the two vertices previously connected by it. This can lead to multiple edges between a pair of vertices, or *self-loops* connecting a vertex to itself. These are not allowed in *simple graphs*, in which case some edges may need to be deleted. Then an undirected graph H is said to be a *minor* of another undirected graph G if a graph isomorphic to H can be obtained from G by contracting some edges, deleting some edges, and deleting some isolated vertices.

11.4 Planarity

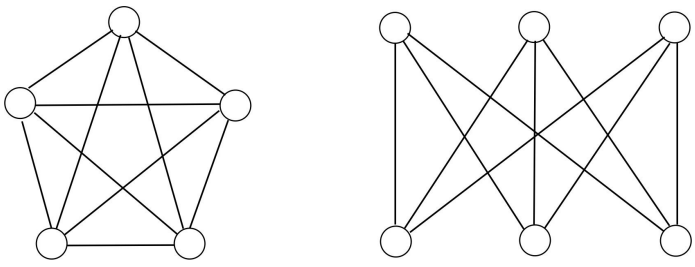
A *planar graph* is a graph that can be *embeded* in a plane. In other words, it can be drawn on a sheet of paper in such a way that no edges cross each other. This is important in applications such as printed circuit design.

Note that it is clearly possible for planar graphs to be drawn in such a way that their edges do cross each other, but the crucial thing is that they can be transformed (by moving vertices and/or deforming the edges) into a form without any edges crossing. For example, the following three diagrams all represent the same planar graph:



This graph is the fully connected graph with four vertices, known as K_4. Clearly all sub-graphs of this will also be planar.

It is actually quite difficult to formulate general algorithms for determining whether a given graph is planar. For small graphs, it is easy to check systematically that there are no possible vertex repositionings or edge deformations that will bring the graph into explicitly planar form. Two slightly larger graphs than K_4 that can be shown to be non-planar in this way are the fully connected graph with five vertices, known as K_5, and the graph with three vertices fully connected to three other vertices, known as $K_{3,3}$:



Clearly, any larger graph that contains one of these two non-planar graphs as a *subgraph* must also be non-planar iteslf, and any *subdivision* or *smoothing* of edges will have no effect

on the planarity. In fact, it can be proved that these two graphs form the basis of some useful theorems about planarity. The most well-known of these is *Kuratowski's theorem* which states that "a finite graph is planar if and only if it does not contain a *subgraph* that is *homeomorphic* to, or a *subdivision* of, K_5 or $K_{3,3}$". Another, based on the concept of *minors*, is *Wagner's theorem* which states that "a finite graph is planar if and only if it does not have K_5 or $K_{3,3}$ as a minor".

A good general approach for testing planarity is therefore to search for subgraphs of the given graph that can be transformed into K_5 or $K_{3,3}$. This is not entirely straightforward, but algorithms do exist which allow a graph with n vertices to be tested for planarity with time complexity $O(n)$. Exercise: find out exactly how these algorithms work.

11.5 Traversals – systematically visiting all vertices

In order to *traverse* a graph, i.e. systematically visit all its vertices, we clearly need a strategy for exploring graphs which guarantees that we do not miss any edges or vertices. Because, unlike trees, graphs do not have a root vertex, there is no natural place to start a traversal, and therefore we assume that we are given, or randomly pick, a starting vertex i. There are two strategies for performing graph traversal.

The first is known as *breadth first traversal*: We start with the given vertex i. Then we visit its neighbours one by one (which must be possible no matter which implementation we use), placing them in an initially empty *queue*. We then remove the first vertex from the queue and one by one put its neighbours at the end of the queue. We then visit the next vertex in the queue and again put its neighbours at the end of the queue. We do this until the queue is empty.

However, there is no reason why this basic algorithm should ever terminate. If there is a circle in the graph, like A, B, C in the first unweighted graph above, we would revisit a vertex *we have already visited*, and thus we would run into an infinite loop (visiting A's neighbours puts B onto the queue, visiting that (eventually) gives us C, and once we reach C in the queue, we get A again). To avoid this we create a second array `done` of booleans, where `done[j]` is `true` if we have already visited the vertex with number j, and it is `false` otherwise. In the above algorithm, we only add a vertex j to the queue if `done[j]` is `false`. Then we mark it as done by setting `done[j] = true`. This way, we will not visit any vertex more than once, and for a finite graph, our algorithm is bound to terminate. In the example we are discussing, breadth first search starting at A might yield: A, B, D, C, E.

To see why this is called breadth first search, we can imagine a tree being built up in this way, where the starting vertex is the root, and the children of each vertex are its neighbours (that haven't already been visited). We would then first follow all the edges emanating from the root, leading to all the vertices on level 1, then find all the vertices on the level below, and so on, until we find all the vertices on the 'lowest' level.

The second traversal strategy is known as *depth first traversal*: Given a vertex i to start from, we now put it on a *stack* rather than a queue (recall that in a stack, the next item to be removed at any time is the last one that was put on the stack). Then we take it from the stack, mark it as done as for breadth first traversal, look up its neighbours one after the other, and put them onto the stack. We then repeatedly pop the next vertex from the stack, mark it as done, and put its neighbours on the stack, provided they have not been marked as done, just as we did for breadth first traversal. For the example discussed above, we might (starting from A) get: A, B, C, E, D. Again, we can see why this is called depth first by

formulating the traversal as a search tree and looking at the order in which the items are added and processed.

Note that with both breadth first and depth first, the order of the vertices depends on the implementation. There is no reason why A's neighbour B should be visited before D in the example. So it is better to speak of *a* result of depth first or breadth first traversal, rather than of *the* result. Note also that the only vertices that will be listed are those in the same *connected component* as A. If we have to ensure that all vertices are visited, we may need to start the traversal process with a number of different starting vertices, each time choosing one that has not been marked as done when the previous traversal terminated.

Exercises: Write algorithms, in pseudocode, to (1) visit *all* nodes of a graph, and (2) decide whether a given graph is connected or not. For (2) you will actually need two algorithms, one for the strong notion of connectedness, and another for the weak notion.

11.6 Shortest paths – Dijkstra's algorithm

A common graph based problem is that we have some situation represented as a weighted digraph with edges labelled by non-negative numbers and need to answer the following question: For two particular vertices, what is the *shortest route* from one to the other?

Here, by "shortest route" we mean a path which, when we add up the weights along its edges, gives the smallest overall weight for the path. This number is called the *length* of the path. Thus, a shortest path is one with minimal length. Note that there need not be a unique shortest path, since several paths might have the same length. In a disconnected graph there will not be a path between vertices in different *components*, but we can take care of this by using ∞ once again to stand for "no path at all".

Note that the weights do not necessarily have to correspond to distances; they could, for example, be time (in which case we could speak of "quickest paths") or money (in which case we could speak of "cheapest paths"), among other possibilities. By considering "abstract" graphs in which the numerical weights are left uninterpreted, we can take care of all such situations and others. But notice that we do need to restrict the edge weights to be non-negative numbers, because if there are negative numbers and cycles, we can have increasingly long paths with lower and lower costs, and no path with minimal cost.

Applications of shortest-path algorithms include internet packet routing (because, if you send an email message from your computer to someone else, it has to go through various email routers, until it reaches its final destination), train-ticket reservation systems (that need to figure out the best connecting stations), and driving route finders (that need to find an optimum route in some sense).

Dijkstra's algorithm. In turns out that, if we want to compute the shortest path from a given start node s to a given end node z, it is actually most convenient to compute the shortest paths from s to all other nodes, not just the given node z that we are interested in. Given the start node, Dijkstra's algorithm computes shortest paths starting from s and ending at each possible node. It maintains all the information it needs in simple arrays, which are iteratively updated until the solution is reached. Because the algorithm, although elegant and short, is fairly complicated, we shall consider it one component at a time.

Overestimation of shortest paths. We keep an array D of distances indexed by the vertices. The idea is that $D[z]$ will hold the distance of the shortest path from s to z *when*

the algorithm finishes. However, *before the algorithm finishes*, $D[z]$ is the best *overestimate* we currently have of the distance from s to z. We initially have $D[s] = 0$, and set $D[z] = \infty$ for all vertices z other than the start node s. Then the algorithm repeatedly decreases the overestimates until it is no longer possible to decrease them further. When this happens, the algorithm terminates, with each estimate fully constrained and said to be *tight*.

Improving estimates. The general idea is to look systematically for *shortcuts*. Suppose that, for two given vertices u and z, it happens that $D[u] + \text{weight}[u][z] < D[z]$. Then there is a way of going from s to u and then to z whose total length is smaller than the current overestimate $D[z]$ of the distance from s to z, and hence we can replace $D[z]$ by this better estimate. This corresponds to the code fragment

```
if ( D[u] + weight[u][z] < D[z] )
   D[z] = D[u] + weight[u][z]
```

of the full algorithm given below. The problem is thus reduced to developing an algorithm that will systematically apply this improvement so that (1) we eventually get the tight estimates promised above, and (2) that is done as efficiently as possible.

Dijkstra's algorithm, Version 1. The first version of such an algorithm is not as efficient as it could be, but it is relatively simple and certainly correct. (It is always a good idea to start with an inefficient simple algorithm, so that the results from it can be used to check the operation of a more complex efficient algorithm.) The general idea is that, at each stage of the algorithm's operation, if an entry $D[u]$ of the array D has the minimal value among all the values recorded in D, then the overestimate $D[u]$ must actually be tight, because the improvement algorithm discussed above cannot possibly find a shortcut.

The following algorithm implements that idea:

```
// Input:  A directed graph with weight matrix 'weight' and
//         a start vertex 's'.
// Output: An array 'D' of distances as explained above.

// We begin by buiding the distance overestimates.

D[s] = 0   // The shortest path from s to itself has length zero.

for ( each vertex z of the graph ) {
   if ( z is not the start vertex s )
      D[z] = infinity    // This is certainly an overestimate.
}

// We use an auxiliary array 'tight' indexed by the vertices,
// that records for which nodes the shortest path estimates
// are ''known'' to be tight by the algorithm.

for ( each vertex z of the graph ) {
   tight[z] = false
}
```

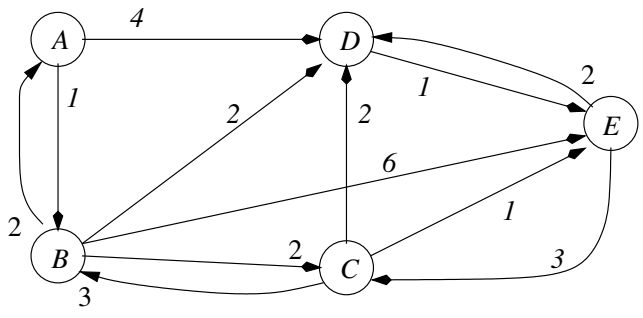
```
// We now repeatedly update the arrays 'D' and 'tight' until
// all entries in the array 'tight' hold the value true.

repeat as many times as there are vertices in the graph {
   find a vertex u with tight[u] false and minimal estimate D[u]
   tight[u] = true
   for ( each vertex z adjacent to u )
      if ( D[u] + weight[u][z] < D[z] )
         D[z] = D[u] + weight[u][z]    // Lower overestimate exists.
}

// At this point, all entries of array 'D' hold tight estimates.
```

It is clear that when this algorithm finishes, the entries of D cannot hold under-estimates of the lengths of the shortest paths. What is perhaps not so clear is why the estimates it holds are actually tight, i.e. are the minimal path lengths. In order to understand why, first notice that an initial sub-path of a shortest path is itself a shortest path. To see this, suppose that you wish to navigate from a vertex s to a vertex z, and that the shortest path from s to z happens to go through a certain vertex u. Then your path from s to z can be split into two paths, one going from s to u (an initial sub-path) and the other going from u to z (a final sub-path). Given that the whole, unsplit path is a shortest path from s to z, the initial sub-path has to be a shortest path from s to u, for if not, then you could shorten your path from s to z by replacing the initial sub-path to u by a shorter path, which would not only give a shorter path from s to u but also from s to the final destination z. Now it follows that for any start vertex, there is a tree of shortest paths from that vertex to all other vertices. The reason is that shortest paths cannot have cycles. Implicitly, Dijkstra's algorithm constructs this tree starting from the root, that is, the start vertex.

If, as tends to be the case in practice, we also wish to compute the route of shortest path, rather than just its length, we also need to introduce a third array `pred` to keep track of the 'predecessor' or 'previous vertex' of each vertex, so that the path can be followed back from the end point to the start point. The algorithm can clearly also be adapted to work with non-weighted graphs by assigning a suitable weight matrix of 1s for connected vertices and 0s for non-connected vertices.

The time complexity of this algorithm is clearly $O(n^2)$ where n is the number of vertices, since there are operations of $O(n)$ nested within the repeat of $O(n)$.

A simple example. Suppose we want to compute the shortest path from A (node 0) to E (node 4) in the weighted graph we looked at before:



A direct implementation of the above algorithm, with some code added to print out the status of the three arrays at each intermediate stage, gives the following output, in which"oo" is used to represent the infinity symbol "∞":

```
Computing shortest paths from A

        |A      B       C       D       E
--------+-------------------------------------
D       |0      oo      oo      oo      oo
tight   |no     no      no      no      no
pred.   |none   none    none    none    none

Vertex A has minimal estimate, and so is tight.

Neighbour B has estimate decreased from oo to 1 taking a shortcut via A.
Neighbour D has estimate decreased from oo to 4 taking a shortcut via A.

        |A      B       C       D       E
--------+-------------------------------------
D       |0      1       oo      4       oo
tight   |yes    no      no      no      no
pred.   |none   A       none    A       none

Vertex B has minimal estimate, and so is tight.

Neighbour A is already tight.
Neighbour C has estimate decreased from oo to 3 taking a shortcut via B.
Neighbour D has estimate decreased from  4 to 3 taking a shortcut via B.
Neighbour E has estimate decreased from oo to 7 taking a shortcut via B.

        |A      B       C       D       E
--------+-------------------------------------
D       |0      1       3       3       7
tight   |yes    yes     no      no      no
pred.   |none   A       B       B       B

Vertex C has minimal estimate, and so is tight.

Neighbour B is already tight.
Neighbour D has estimate unchanged.
Neighbour E has estimate decreased from 7 to 4 taking a shortcut via C.

        |A      B       C       D       E
--------+-------------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     no      no
pred.   |none   A       B       B       C
```

```
Vertex D has minimal estimate, and so is tight.

Neighbour E has estimate unchanged.

        |A      B       C       D       E
--------+---------------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     no
pred.   |none   A       B       B       C

Vertex E has minimal estimate, and so is tight.

Neighbour C is already tight.
Neighbour D is already tight.

        |A      B       C       D       E
--------+---------------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     yes
pred.   |none   A       B       B       C

End of Dijkstra's computation.

A shortest path from A to E is:  A B C E.
```

Once it is clear what is happening at each stage, it is usually more convenient to adopt a shorthand notation that allows the whole process to be represented in a single table. For example, using a "*" to represent tight, the distance, status and predecessor for each node at each stage of the above example can be listed more concisely as follows:

```
Stage  |   A        B         C         D         E
-------+-----------------------------------------------
1      |  0        oo        oo        oo        oo
2      |  0 *      1   A     oo        4   A     oo
3      |  0 *      1 * A     3   B     3   B     7   B
4      |  0 *      1 * A     3 * B     3   B     4   C
5      |  0 *      1 * A     3 * B     3 * B     4   C
6      |  0 *      1 * A     3 * B     3 * B     4 * C

A shortest path from A to E is:  A B C E.
```

Dijkstra's algorithm, Version 2. The time complexity of Dijkstra's algorithm can be improved by making use of a *priority queue* (e.g., some form of heap) to keep track of which node's distance estimate becomes tight next. Here it is convenient to use the convention that lower numbers have higher priority. The previous algorithm then becomes:

```
// Input:  A directed graph with weight matrix 'weight' and
//         a start vertex 's'.
// Output: An array 'D' of distances as explained above.

// We begin by buiding the distance overestimates.

D[s] = 0   // The shortest path from s to itself has length zero.

for ( each vertex z of the graph ) {
   if ( z is not the start vertex s )
      D[z] = infinity    // This is certainly an overestimate.
}

// Then we set up a priority queue based on the overestimates.

Create a priority queue containing all the vertices of the graph,
with the entries of D as the priorities

// Then we implicitly build the path tree discussed above.

while ( priority queue is not empty ) {
   // The next vertex of the path tree is called u.
   u = remove vertex with smallest priority from queue
   for ( each vertex z in the queue which is adjacent to u ) {
      if ( D[u] + weight[u][z] < D[z] ) {
         D[z] = D[u] + weight[u][z]    // Lower overestimate exists.
         Change the priority of vertex z in queue to D[z]
      }
   }
}

// At this point, all entries of array 'D' hold tight estimates.
```

If the priority queue is implemented as a *Binary* or *Binomial heap*, initializing D and creating the priority queue both have complexity $O(n)$, where n is the number of vertices of the graph, and that is negligible compared to the rest of the algorithm. Then removing vertices and changing the priorities of elements in the priority queue require some rearrangement of the heap tree by "bubbling up", and that takes $O(\log_2 n)$ steps, because that is the maximum height of the tree. Removals happen $O(n)$ times, and priority changes happen $O(e)$ times, where e is the number of edges in the graph, so the cost of maintaining the queue and updating D is $O((e+n)\log_2 n)$. Thus, the total time complexity of this form of Dijkstra's algorithm is $O((e+n)\log_2 n)$. Using a *Fibonacci heap* for the priority queue allows priority updates of $O(1)$ complexity, improving the overall complexity to $O(e + n\log_2 n)$.

In a *fully connected* graph, the number of edges e will be $O(n^2)$, and hence the time complexity of this algorithm is $O(n^2\log_2 n)$ or $O(n^2)$ depending on which kind of priority queue is used. So, in that case, the time complexity is actually greater than or equal to the previous simpler $O(n^2)$ algorithm. However, in practice, many graphs tend to be much more

sparse with $e = O(n)$. That is, there are usually not many more edges than vertices, and in this case the time complexity for both priority queue versions is $O(n\log_2 n)$, which is a clear improvement over the previous $O(n^2)$ algorithm.

11.7 Shortest paths – Floyd's algorithm

If we are not only interested in finding the shortest path from one specific vertex to all the others, but the shortest paths between every pair of vertices, we could, of course, apply Dijkstra's algorithm to every starting vertex. But there is actually a simpler way of doing this, known as *Floyd's algorithm*. This maintains a square matrix 'distance' which contains the overestimates of the shortest paths between every pair of vertices, and systematically decreases the overestimates using the same *shortcut* idea as above. If we also wish to keep track of the routes of the shortest paths, rather than just their lengths, we simply introduce a second square matrix 'predecessor' to keep track of all the 'previous vertices'.

In the algorithm below, we attempt to decrease the estimate of the distance from each vertex s to each vertex z by going systematically via each possible vertex u to see whether that is a shortcut; and if it is, the overestimate of the distance is decreased to the smaller overestimate, and the predecessor updated:

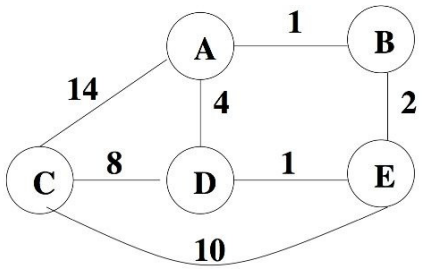
```
// Store initial estimates and predecessors.

for ( each vertex s ) {
   for ( each vertex z ) {
      distance[s][z] = weight[s][z]
      predecessor[s][z] = s
   }
}

// Improve them by considering all possible shortcuts u.

for ( each vertex u ) {
   for ( each vertex s ) {
      for ( each vertex z ) {
         if ( distance[s][u]+distance[u][z] < distance[s][z] ) {
            distance[s][z] = distance[s][u]+distance[u][z]
            predecessor[s][z] = predecessor[u][z]
         }
      }
   }
}
```

As with Dijkstra's algorithm, this can easily be adapted to the case of non-weighted graphs by assigning a suitable weight matrix of 0s and 1s.

The time complexity here is clearly $O(n^3)$, since it involves three nested for loops of $O(n)$. This is the same complexity as running the $O(n^2)$ Dijkstra's algorithm once for each of the n possible starting vertices. In general, however, Floyd's algorithm will be faster than Dijkstra's, even though they are both in the same complexity class, because the former performs fewer

instructions in each run through the loops. However, if the graph is *sparse* with $e = O(n)$, then multiple runs of Dijkstra's algorithm can be made to perform with time complexity $O(n^2 \log_2 n)$, and be faster than Floyd's algorithm.

A simple example. Suppose we want to compute the lengths of the shortest paths between all vertices in the following undirected weighted graph:



We start with distance matrix based on the connection weights, and trivial predecessors:

Start

	A	B	C	D	E
A	0	1	14	4	∞
B	1	0	∞	∞	2
C	14	∞	0	8	10
D	4	∞	8	0	1
E	∞	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	A
B	B	B	B	B	B
C	C	C	C	C	C
D	D	D	D	D	D
E	E	E	E	E	E

Then for each vertex in turn we test whether a shortcut via that vertex reduces any of the distances, and update the distance and predecessor arrays with any reductions found. The five steps, with the updated entries in quotes, are as follows::

A :

	A	B	C	D	E
A	0	1	14	4	∞
B	1	0	'15'	'5'	2
C	14	'15'	0	8	10
D	4	'5'	8	0	1
E	∞	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	A
B	B	B	'A'	'A'	B
C	C	'A'	C	C	C
D	D	'A'	D	D	D
E	E	E	E	E	E

B :

	A	B	C	D	E
A	0	1	14	4	'3'
B	1	0	15	5	2
C	14	15	0	8	10
D	4	5	8	0	1
E	'3'	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	'B'
B	B	B	A	A	B
C	C	A	C	C	C
D	D	A	D	D	D
E	'B'	E	E	E	E

C :

	A	B	C	D	E
A	0	1	14	4	3
B	1	0	15	5	2
C	14	15	0	8	10
D	4	5	8	0	1
E	3	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	B
B	B	B	A	A	B
C	C	A	C	C	C
D	D	A	D	D	D
E	B	E	E	E	E

D :

	A	B	C	D	E
A	0	1	‘12’	4	3
B	1	0	‘13’	5	2
C	‘12’	‘13’	0	8	‘9’
D	4	5	8	0	1
E	3	2	‘9’	1	0

	A	B	C	D	E
A	A	A	‘D’	A	B
B	B	B	‘D’	A	B
C	‘D’	A	C	C	‘D’
D	D	A	D	D	D
E	B	E	‘D’	E	E

E :

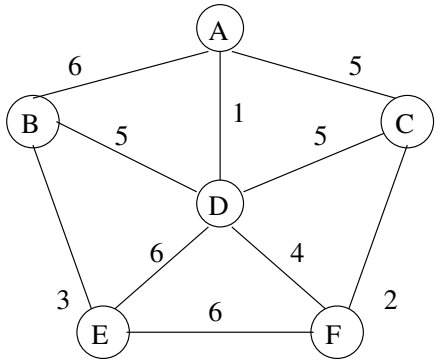
	A	B	C	D	E
A	0	1	12	4	3
B	1	0	‘11’	‘3’	2
C	12	‘11’	0	8	9
D	4	‘3’	8	0	1
E	3	2	9	1	0

	A	B	C	D	E
A	A	A	D	A	B
B	B	B	D	‘E’	B
C	D	‘E’	C	C	D
D	D	‘E’	D	D	D
E	B	E	D	E	E

The algorithm finishes with the matrix of shortest distances and the matrix of associated predecessors. So the shortest distance from C to B is 11, and the predecessors of B are E, then D, then C, giving the path C D E B. Note that updating a distance does not necessarily mean updating the associated predecessor – for example, when introducing D as a shortcut between C and B, the predecessor of B remains A.

11.8 Minimal spanning trees

We now move on to another common graph-based problem. Suppose you have been given a weighted undirected graph such as the following:

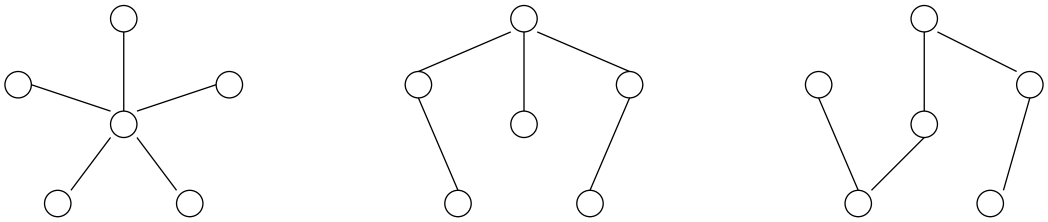


We could think of the vertices as representing houses, and the weights as the distances between them. Now imagine that you are tasked with supplying all these houses with some commodity such as water, gas, or electricity. For obvious reasons, you will want to keep the amount of digging and laying of pipes or cable to a minimum. So, what is the best pipe or cable layout that you can find, i.e. what layout has the shortest overall length?

Obviously, we will have to choose some of the edges to dig along, but not all of them. For example, if we have already chosen the edge between A and D, and the one between B and D, then there is no reason to also have the one between A and B. More generally, it is clear that we want to avoid *circles*. Also, assuming that we have only one feeding-in point (it is of no importance which of the vertices that is), we need the whole layout to be *connected*. We have seen already that a connected graph without circles is a tree.

Hence, what we are looking for is a *minimal spanning tree* of the graph. A *spanning tree* of a graph is a *subgraph* that is a tree which connects all the vertices together, so it 'spans' the original graph but using fewer edges. Here, *minimal* refers to the sum of all the weights of the edges contained in that tree, so a *minimal spanning tree* has total weight less than or equal to the total weight of every other spanning tree. As we shall see, there will not necessarily be a unique minimal spanning tree for a given graph.

Observations concerning spanning trees. For the other graph algorithms we have covered so far, we started by making some observations which allowed us to come up with an idea for an algorithm, as well as a strategy for formulating a proof that the algorithm did indeed perform as desired. So, to come up with some ideas which will allow us to develop an algorithm for the minimal spanning tree problem, we shall need to make some observations about minimal spanning trees. Let us assume, for the time being, that all the weights in the above graph were equal, to give us some idea of what kind of shape a minimal spanning tree might have under those circumstances. Here are some examples:



We can immediately notice that their general shape is such that if we add any of the remaining edges, we would create a circle. Then we can see that going from one spanning tree to another can be achieved by removing an edge and replacing it by another (to the vertex which would otherwise be unconnected) such that no circle is created. These observations are not quite sufficient to lead to an algorithm, but they are good enough to let us prove that the algorithms we find do actually work.

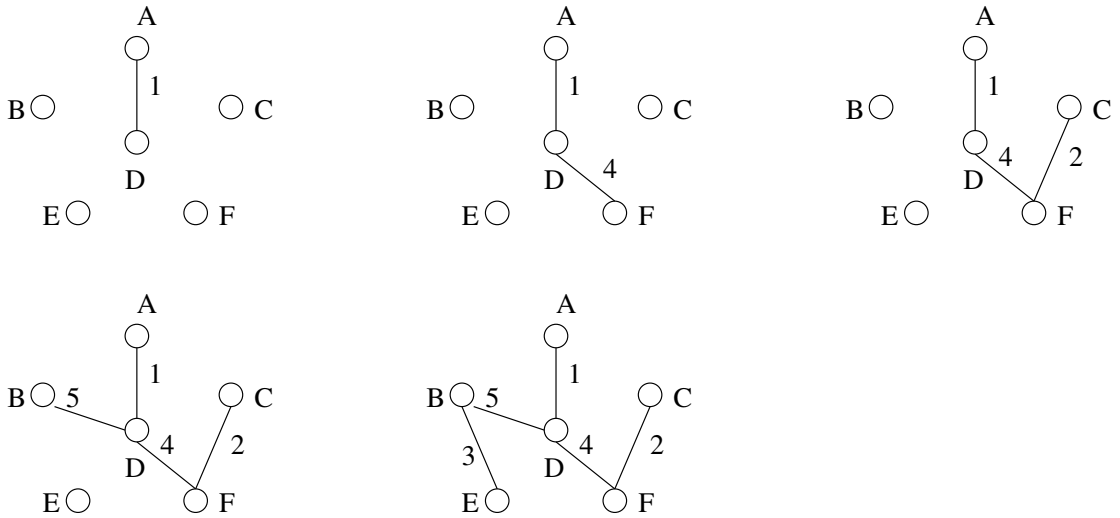
Greedy Algorithms. We say that an algorithm is *greedy* if it makes its decisions based only on what is best from the point of view of 'local considerations', with no concern about how the decision might affect the overall picture. The general idea is to start with an approximation, as we did in Dijkstra's algorithm, and then refine it in a series of steps. The algorithm is greedy in the sense that the decision at each step is based only on what is best for that next step, and does not consider how that will affect the quality of the final overall solution. We shall now consider some greedy approaches to the minimal spanning tree problem.

Prim's Algorithm – A greedy vertex-based approach. Suppose that we already have a spanning tree connecting some set of vertices S. Then we can consider all the edges which connect a vertex in S to one outside of S, and add to S one of those that has minimal weight. This cannot possibly create a circle, since it must add a vertex not yet in S. This process can be repeated, starting with any vertex to be the sole element of S, which is a trivial minimal spanning tree containing no edges. This approach is known as *Prim's algorithm*.

When implementing Prim's algorithm, one can use either an array or a list to keep track of the set of vertices S reached so far. One could then maintain another array or list `closest` which, for each vertex i not yet in S, keeps track of the vertex in S closest to i. That is, the

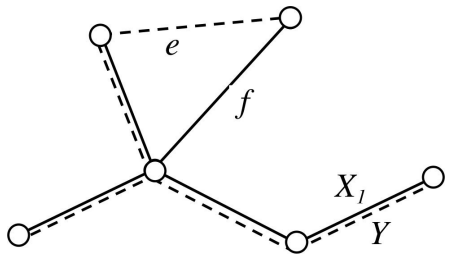
vertex in S which has an edge to i with minimal weight. If `closest` also keeps track of the weights of those edges, we could save time, because we would then only have to check the weights mentioned in that array or list.

For the above graph, starting with $S = \{A\}$, the tree is built up as follows:



It is slightly more challenging to produce a convincing argument that this algorithm really works than it has been for the other algorithms we have seen so far. It is clear that Prim's algorithm must result in a spanning tree, because it generates a tree that spans all the vertices, but it is not obvious that it is minimal. There are several possible proofs that it is, but none are straightforward. The simplest works by showing that the set of all possible minimal spanning trees X_i must include the output of Prim's algorithm.

Let Y be the output of Prim's algorithm, and X_1 be any minimal spanning tree. The following illustrates such a situation:



We don't actually need to know what X_1 is – we just need to know the properties it must satisfy, and then systematically work through all the possibilities, showing that Y is a minimal spanning tree in each case. Clearly, if $X_1 = Y$, then Prim's algorithm has generated a minimal spanning tree. Otherwise, let e be the first edge added to Y that is not in X_1. Then, since X_1 is a spanning tree, it must include a path connecting the two endpoints of e, and because circles are not allowed, there must be an edge in X_1 that is not in Y, which we can call f. Since Prim's algorithm added e rather than f, we know weight$(e) \leq$ weight(f). Then create tree X_2 that is X_1 with f replaced by e. Clearly X_2 is connected, has the same number of edges as X_1, spans all the vertices, and has total weight no greater than X_1, so it must also

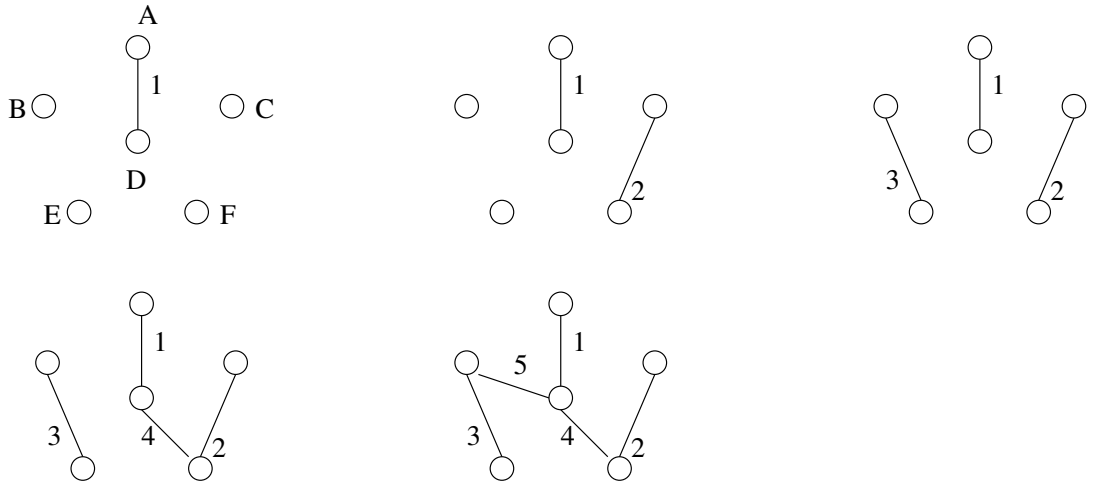
be a minimal spanning tree. Now we can repeat this process until we have replaced all the edges in X_1 that are not in Y, and we end up with the minimal spanning tree $X_n = Y$, which completes the proof that Y is a minimal spanning tree.

The time complexity of the standard Prim's algorithm is $O(n^2)$ because at each step we need to choose a vertex to add to S, and then update the `closest` array, not dissimilar to the simplest form of Dijkstra's algorithm. However, as with Dijkstra's algorithm, a *Binary* or *Binomial heap* based priority queue can be used to speed things up by keeping track of which is the minimal weight vertex to be added next. With an adjacency list representation, this can bring the complexity down to $O((e+n)\log_2 n)$. Finally, using the more sophisticated *Fibonacci heap* for the priority queue can improve this further to $O(e+n\log_2 n)$. Thus, using the optimal approach in each case, Prim's algorithm is $O(n\log_2 n)$ for sparse graphs that have $e = O(n)$, and $O(n^2)$ for highly connected graphs that have $e = O(n^2)$.

Just as with Floyd's versus Dijkstra's algorithm, we should consider whether it eally is necessary to process every vertex at each stage, because it could be sufficient to only check actually existing edges. We therefore now consider an alternative edge-based strategy:

Kruskal's algorithm – A greedy edge-based approach. This algorithm does not consider the vertices directly at all, but builds a minimal spanning tree by considering and adding edges as follows: Assume that we already have a collection of edges T. Then, from all the edges not yet in T, choose one with minimal weight such that its addition to T does not produce a circle, and add that to T. If we start with T being the empty set, and continue until no more edges can be added, a minimal spanning tree will be produced. This approach is known as *Kruskal's algorithm*.

For the same graph as used for Prim's algorithm, this algorithm proceeds as follows:



In practice, Kruskal's algorithm is implemented in a rather different way to Prim's algorithm. The general idea of the most efficient approaches is to start by sorting the edges according to their weights, and then simply go through that list of edges in order of increasing weight, and either add them to T, or reject them if they would produce a circle. There are implementations of that which can be achieved with overall time complexity $O(e\log_2 e)$, which is dominated by the $O(e\log_2 e)$ complexity of sorting the e edges in the first place.

This means that the choice between Prim's algorithm and Kruskal's algorithm depends on the connectivity of the particular graph under consideration. If the graph is sparse, i.e. the

number of edges is not much more than the number of vertices, then Kruskal's algorithm will have the same $O(n\log_2 n)$ complexity as the optimal priority queue based versions of Prim's algorithm, but will be faster than the standard $O(n^2)$ Prim's algorithm. However, if the graph is highly connected, i.e. the number of edges is near the square of the number of vertices, it will have complexity $O(n^2\log_2 n)$ and be slower than the optimal $O(n^2)$ versions of Prim's algorithm.

11.9 Travelling Salesmen and Vehicle Routing

Note that all the graph algorithms we have considered so far have had *polynomial time complexity*. There are further graph based problems that are even more complex.

Probably the most well known of these is the *Travelling Salesman Problem*, which involves finding the shortest path through a graph which visits each node precisely once. There are currently no known polynomial time algorithms for solving this. Since only algorithms with *exponential complexity* are known, this makes the Travelling Salesman Problem difficult even for moderately sized n (e.g., all capital cities). Exercise: write an algorithm in pseudocode that solves the Travelling Salesman Problem, and determine its time complexity.

A variation of the shortest path problem with enormous practical importance in transportation is the *Vehicle Routing Problem*. This involves finding a series of routes to service a number of customers with a fleet of vehicles with minimal cost, where that cost may be the number of vehicles required, the total distance covered, or the total driver time required. Often, for practical instances, there are conflicts between the various objectives, and there is a *trade-off* between the various costs which have to be balanced. In such cases, a *multi-objective optimization* approach is required which returns a *Pareto front* of non-dominated solutions, i.e. a set solutions for which there are no other solutions which are better on all objectives. Also, in practice, there are usually various *constraints* involved, such as fixed delivery time-windows, or limited capacity vehicles, that must be satisfied, and that makes finding good solutions even more difficult.

Since exact solutions to these problems are currently impossible for all but the smallest cases, *heuristic* approaches are usually employed, such as *evolutionary computation*, which deliver solutions that are probably good but cannot be proved to be optimal. One popular approach is to maintain a whole population of solutions, and use simulated evolution by natural selection to iteratively improve the quality of those solutions. That has the additional advantage of being able to generate a whole Pareto front of solutions rather than just a single solution. This is currently still a very active research area.

Chapter 12

Epilogue

Hopefully the reader will agree that these notes have achieved their objective of introducing the basic *data structures* used in computer science, and showing how they can be used in the design of useful and efficient *algorithms*. The basic data structures (arrays, lists, stacks, queues and trees) have been employed throughout, and used as the basis of the crucial processes, such as storing, sorting and searching data, which underly many computer science applications. It has been demonstrated how ideas from combinatorics and probability theory can be used to compute the efficiency of algorithms as a function of problem size. We have seen how considerations of computational efficiency then drive the development of more complex data structures (such as binary search trees, heaps and hash tables) and associated algorithms. General ideas such as recursion and divide-and-conquer have been used to provide more efficient algorithms, and inductive definitions and invariants have been used to establish proofs of correctness of algorithms. Throughout, abstract data types and pseudo-code have allowed an implementation independent approach that facilitates application to any programming environment in the future.

Clearly, these notes have only been able to provide a brief introduction to the topic, and the algorithms and data structures and efficiency computations discussed have been relatively simple examples. However, having a good understanding of these fundamental ideas and design patterns allows them to be easily expanded and elaborated to deal with the more complex applications that will inevitably arise in the future.