Algorithms and Data Structures in Scheme

Wolfgang De Meuter

Vrije Universiteit Brussel
Programming Technology Lab
Pleinlaan 2
1050 Brussels
Belgium

November 30, 2008

Ackowledgements

This is the third version of a text originally written in 2006 and first updated in 2007. I would like to thank my teaching assistants Charlotte Herzeel, Peter Ebraert, Steven Claes, Andy Kellens and Matthias Stevens for comments and improvements on early versions of the text and the code it discusses.

I would also like to thank all the students that had to work with early versions of the text. Special thanks to Paul Rozie, Jeffrey Geyssens, Clovis Six, Kim Bauters and Brecht De Rooms for their comments and bug reports.

Contents

Chapter 1

Introduction

Understanding a scientific domain consists of a healthy dosis of reasoning along with a vast body of knowledge. Knowing lots of individual facts without having good reasoning capabilities does not bring one very far. A lawyer that has a perfect knowledge of all the laws in his country, but who is not smart enough to apply them to a concrete case will generally not be considered a good lawyer. However, being extremely smart without having the factual knowledge accumulated by previous thinkers will require one to accumulate that knowledge all over again. It is sometimes said that those who do not know history are bound to live it all over again. The same goes for the knowledge accumulated in a scientific field and thus also in the field of computer science.

This course presents an important slice of factual knowledge that any computer scientist ought to have acquired during his academic training. It provides students with an encyclopedic overview of programming knowledge that has been accumulated by computer scientists over the past six decades. That knowledge concerns both knowledge about data structures and knowledge about computations that operate on these structures. These computations are also termed *algorithms*. As we will see, there exist a virtual infinite number of algorithms and a virtually infinite number of ways to construct data structures in a computer memory. On first sight, every time a programmer writes a new program, he has to think about the way his data will be structured in memory. Similarly, on first sight, every time a programmer writes a new program, he has to think about how to organise that program from scratch. However, if this were true, we would not be able to speak about computer science as a science. Doing everything from scratch every time again is known a craftsmanship as opposed to science. On second sight, however, there are quite a number of data structures and many algorithms that seem to be popping up over and over again when writing programs. It is therefore useful to bundle them in a catalogue of algorithms and data structures and to teach that catalogue to newly bred computer scientists. This is what this course is all about. The chapters of the course present a systematic overview of "the" standard textbook algorithms and data structures. But if there exists something like "the" standard textbook

set of algorithms, then why do we need (yet) another textbook on this subject matter? The reason is that a course on algorithms and data structures somehow has to *precisely* specify those algorithms and explicitly show how the data structures are effectively created in a real computer memory. This is done in a *programming language*. Although there exist many textbooks on the subject matter that use C, C++, Java, Pascal and many other programming languages, there is no such book that uses the Scheme programming language. This is why we have written this text.

One might ask the question whether the programming language really matters. Can't we use any programming language to program a certain algorithm? This is indeed the case. The algorithms presented in this text can be programmed in any programming language and this is an important raison d'être for this course. It provides a number of insights about algorithms and data structures that any computer programmer needs, irrespective of the programming language he or she uses. Nevertheless, every programming language has its own technical peculiarities and its own particular style of writing programs. This causes algorithms to be written in slightly different ways when using different programming languages. Hence, the same algorithm written in Java or Pascal will probably look slightly different when using Scheme simply because Scheme offers programmers different ways of writing programs; ways which not always know their equivalent in Java or Pascal. Although an algorithm written in one programming language can always be translated to another programming language, the precise nuances will not be identical. The difference is comparable to that of a poem originally written in Dutch not being exactly the same when translated to English.

We start this chapter by introducing some important terminology that will be used throughout the course. In sections 1.1 and 1.2 we define what we mean by terms such as data, data structure, data type, algorithm and procedure type. As we will see, algorithms and data structures soon get very complex. Sections 1.3 and 1.4 discuss ways to tackle this complexity. An important aspect of this course is that we will be studying data structures and algorithms in a scientific way. E.g., when comparing two different algorithms that solve the same problem, we want to know which algorithm performs best. In section 1.5, we present a scientific technique that can be used to measure the quality of algorithms. We end this chapter by explaining how to apply this technique to measure the quality of Scheme programs.

1.1 Data and Data Constructors

A program can be roughly described as a recipe that prescribes a computer to perform a number of computational steps given some input data. That input data can be as simple as a number in the case of a program that computes factorial numbers. The input data can be as complex as an entire book (consisting of chapters, figures, tables, ...) in the case of a word processor. Therefore, every programming language has ways to describe simple data already existing in the

programming language (such as numbers) and ways to construct new data from the already existing data. The simple data that comes with the programming language itself is usually called *primitive data*. It is built into the programming language and it is characterised by the fact that it cannot be decomposed into smaller units of data that make sense in the programming language. In Scheme, examples of primitive data are built-in numbers, characters and so on. A fairly complete list of Scheme's different kinds of primitive data is shown in figure 1.1. The left hand side of the table shows a name for the kind of data we are considering. The right hand side shows some examples of data of the corresponding kind. We call them *data values* or *data elements*. A complete list of Scheme's different kinds of primitive data values can be found in R6RS (i.e. the official description of standard Scheme).

Name	Example Data Values
number	3, 0+4i, 3.1415, 22/7
boolean	#t, #f
symbol	'apple
character	#\a, #\newline, #\space
procedure	#<primitive:sin>

Figure 1.1: Some of Scheme's primitive data types

Apart from primitive data values, every programming language features ways for constructing new *compound data elements* given a series of 'already existing' data elements. With 'already existing' data elements, we mean both primitive data elements as well as previously constructed compound data elements. Compound data elements are also called *data structures*. The defining characteristic of compound data elements is that they can be decomposed into (simpler) constituent data elements that make sense in the same programming language. As already said, the data elements constituting a compound data elements can be either primitive data values or compound data values that were constructed in their turn. Scheme contains three sorts of compound data values. They are summarized in figure 1.2. Again, the left hand side of the table shows a name for the kind of data elements we are considering. The second column gives us a sample data value of the corresponding kind. The third column is explained below.

Name	Example Data Values	Data Constructor
pair	(1 . 2)	`cons`
string	"Hello World"	`make-string`
vector	#3(1 2 3)	`make-vector`

Figure 1.2: Scheme's compound data types

Constructing a new compound data value is accomplished by applying a Scheme procedure that is known as a *data constructor*. As illustrated in figure 1.2, Scheme features `cons`, `make-string` and `make-vector` as its three main data constructors. Data constructors are procedures that actually create a new data structure: they reserve the necessary amount of computer memory to store the data elements that make up the compound data value and they make sure the compound data value is properly initialized. Initialisation is necessary in order to make sure that the data structure is made of meaningful data elements. For instance, merely reserving the memory needed to store a dotted pair without properly initializing its contents would result in gibberish when applying `car` and `cdr` to the newly constructed pair. This is because the part of the computer memory that is used to store the new pair does not necessarily contain meaningful data. It might contain old "garbage" resulting from pairs that were previously stored in the same part of memory. In order to avoid this, `cons` takes exactly two arguments which are used to properly initialize the dotted pair after having reserved the amount of memory necessary to store it. In brief, data constructors initialize a data structure after having reserved the necessary computer memory. Data constructors can take several forms:

Procedural Data Constructors are constructors such as `cons`, `make-vector` and `make-string`. These Scheme procedures have to be called explicitly by the programmer in order to create a data structure in computer memory. `cons` is a procedure that takes two arguments used to initialize the constituents of the newly constructed pair. In the case of `make-vector` and `make-string`, the constructed data structure still needs to be "filled up" afterwards by calling additional procedures defined for this purpose. For example, by running (`make-vector` 10), we explicitly ask the Scheme interpreter to construct a new vector with 10 entries. However, it is up to the programmer to fill up the entries after the data constructor has been executed. In the example of vectors, this is accomplished by calling `vector-set!`. Strings are the topic of chapter 2.

Literal Data Constructors are notations such as #(...) and "...". These notations create new data structures without explicitly calling a Scheme procedure. Just like procedural data constructors, literal data constructors reserve the necessary amount of memory. Moreover, they initialize that memory by using the data values used in the notation. For example, in Scheme, a string can be constructed by "just" writing its contents between double quotes: "Hello `World`". Hence, the double quotes have to be seen as a data constructor that first reserves the computer memory needed to store the string's contents and subsequently fills up that memory with the characters that come with the notation. The string is said to be written down literally. Hence the name 'literal data constructors'. Apart from the double quotes used to construct strings, Scheme features the #(...) notation to create vectors. For example, writing #(1 2 "Hello `World`") creates a vector with three entries, namely, 1, 2 and "Hello `World`".

Having constructed a compound data value (i.e. a data structure) using a constructor, one will typically encounter three kinds of Scheme procedures that perform meaningful computations on that data value:

Accessors are procedures whose purpose it is to externalize the data elements residing in the data structure. Accessors are used to read data elements from the data structure. Accessors are also known as "getters". Typical examples of accessors are `car` and `cdr`: given a dotted pair (constructed by `cons`), then `car` and `cdr` are used to access the data values residing in the dotted pair. Similarly, `vector-ref` is used to access the data values residing in the entries of a vector. Likewise, `string-ref` can be used to access a string's individual characters.

Mutators can be regarded as the conceptual counterpart of accessors. Mutators are procedures that store a data value in a compound data structure. Mutators are also referred to as "setters". In the case of dotted pairs, `set-car!` and `set-cdr!` are the mutators. These procedures are applied to a dotted pair `p` and a data value `v` (e.g. (`set-car! p v`)). They store the data value `v` in the dotted pair `p`. Likewise, `vector-set!` is a mutator for vectors. Some data structures are said to be *immutable* because they lack mutators. E.g., in Scheme, there are no mutators that can be used to modify the characters of a given string data value.

Operations constitute a third category of procedures defined on compound data values. Operations are procedures that operate on the data structure *without* revealing the internal details of that data structure. An example of an operation is (`reverse a-list`) which reverses a given list (i.e. returns the elements of the list in reverse order). `reverse` operates on the list without revealing its constituting elements. Another example is `member?` which checks whether a given element is a member of a given list. Again, the operation operates on the list without giving its user explicit access to the individual data elements that constitute the list.

Finally, the important notion of a *data type* has to be explained. Every data value in Scheme can be classified in a set. E.g., the data value `3` is said to belong to the set of Scheme numbers. The name of that set (in this case `number`) is said to be the data type of that data value. Other examples of data types are `character`, `pair`, `string` and `vector`. Apart from serving as a name for the kind of the data elements it represents, a data type is also crucial in order to know exactly the procedures that are applicable to its data values. E.g., it is clear that the Scheme procedure + is not applicable to the compound data elements produced by `cons`. Indeed, dotted pairs do not belong to the data type `number` and thus cannot be added. Conversely, the procedure `car` is not applicable to the value `3`. This is because `car` is only applicable to data values whose data type is `pair`. To summarize, a data type is *the* indicator for the set of procedures (i.e. accessors, mutators and operations) that are applicable to the data values of that data type. For Scheme's primitive data types, this set of

operations is simply too big to be listed in this text. We refer to the R6RS for a complete list of operations defined on Scheme's primitive data types. Table 1.3 shows a number of compound Scheme data types along with the most common accessors, mutators and operations that are applicable to the data values of the corresponding data type.

Name of the Type	Operations applicable

`pair`	`car`
	`cdr`
	`set-car!`
	`set-cdr!`
	`equal?`
	`eq?`
	`eqv?`
	...

`string`	`string-length`
	`string-ref`
	`string<?`
	...
	`string-ci<?`
	...
	`substring`
	`string-append`
	`string->list`
	`list->string`
	`string-copy`
	`string-fill!`

`vector`	`vector-length`
	`vector-ref`
	`vector-set!`
	`vector->list`
	`list->vector`
	`vector-fill!`

Figure 1.3: Operations applicable to some of Scheme's compound data types

1.2 Algorithms and Algorithmic Constructors

Now that we have developed a precise vocabulary to talk about data, data values, data structures and data types, let us turn our attention to algorithms. Technically spoken, an algorithm is just a Scheme procedure. It is a textually specified recipe for a certain number of computational steps to be executed on a given input. You are probably already familiar with a number of algorithms

from your Scheme course. Examples include procedures to compute the square root of a number, numerical procedures for computing integrals and a procedure to reverse a Scheme list.

The origin of the word *algorithm* is to be found in the name of the 9th century Persian mathematician Abu Abdullah Muhammad ibn Musa al-Khwarizmi ("the one from Khwarizmi"). The word "algorism" originally referred to the rules needed for doing arithmetic in our decimal system (in which numbers are written using ten symbols having the values 0 through 9 and in which each symbol has ten times the weight of the one to its right). The meaning of the word has evolved, via European Latin translation of al-Khwarizmi's name, into "algorithm" in the 18th century. The word evolved to include all definite procedures for solving problems or performing computational tasks.

The two foremost important properties of algorithms is that they are generally applicable procedures that can be executed by following a well-defined set of elementary computational steps:

Generality. An algorithm has to be generally applicable to all possible inputs of a certain data type. If one was to write a Scheme procedure `fac` that is only applicable to odd numbers smaller than 101, then this would not be considered a valid algorithm for computing factorials. The reason is that we *know* that factorials exist for all other positive numbers as well and that we *know* that there are easily executable procedures that lead to those factorial numbers. Hence, the Scheme procedure would not be considered an algorithm for computing factorials because it is not general enough.

Computability. An algorithm should consist of a number of clearly specified computational steps. For example, a procedure that includes instructions such as "ask an oracle to crack the code of my bank card and then print this code on the screen" would not be considered a valid algorithm. The underlying point is that we demand from an algorithm that it can be mapped onto expressions in a programming language such as Scheme. When specifying algorithms without using a real technical programming language (but instead use plain English for instance), there is a potential danger that the result cannot be programmed at all. In order to avoid confusion, this text therefore specifies all algorithms directly in Scheme.

Anyone who already had an introductory course on Scheme will recognise this as a mere Scheme procedure. So what is the difference between the algorithms presented in an introductory course on Scheme and the algorithms presented here? In order to see this difference, it is necessary to have a look at the inputs for algorithms. In a typical introductory course on programming, the algorithms presented usually operate on fairly simple types of input data. In Scheme this usually implies algorithms that operate on numbers (like e.g. computing the square root of a given number with a given precision) or lists (like e.g. reversing a list or appending two lists). Seen from the perspective of this text, these are extremely simple kinds of input. As a result, the algorithms

that operate on them are fairly simple as well. Things get more interesting if we consider algorithms that operate on richer data structures. This is exactly what this course is about. In general one might say that the more complex the data structures get, the more complex (and thus the more interesting) the algorithms get that operate on those data structures.

Let us now develop some vocabulary to talk about algorithms in the same spirit of the vocabulary for data structures which we presented in the previous section. Similar to terms like data value, data type, primitive data, compound data and data constructor, we introduce the terms algorithm, algorithmic type, primitive algorithms, compound algorithms and algorithmic constructor.

We will call any procedural Scheme element that cannot be divided into more atomic Scheme building blocks a *primitive algorithm*. Examples of primitive algorithms include Scheme's built-in procedures such as +, `sin` and `display`. Even some parenthesis can be considered as primitive Scheme algorithms: those parenthesis that are used to denote a procedure call cannot be decomposed into more primitive computations. Therefore, calling a procedure is one of Scheme's most fundamental building blocks for writing programs. It is a mechanism that is so inherent to Scheme that we consider it as a primitive algorithmic construction. For example, the Scheme expression (`sin 3.14`) consists of two primitive algorithms: one — the parentheses — to call a procedure and another one — the `sin` operator — that corresponds to the built-in sine procedure. However, not all parentheses denote primitive algorithms. To see this, we define a *compound algorithm* as any Scheme algorithm that was created by applying a number of *algorithm constructors* to primitive algorithms or previously built compound algorithms. The algorithm constructors are those Scheme special forms that form more complex algorithms from simple algorithms. Examples of algorithmic constructors are `do`, `let`, `let*`, `letrec`, `lambda` and `if`. These special forms are used to build compound algorithms given some primitive algorithms or a set of previously built compound algorithms. For example, consider the following factorial function implemented in Scheme.

```
(define fac
  (lambda (n)
    (if (= n 0)
      1
      (* n (fac (- n 1))))))
```

Based on the primitive algorithms +, *, −, = and Scheme's procedure calling mechanism, the `if` and `lambda` algorithmic constructors are used to build a compound algorithm that is clever enough to compute factorials.

Notice that we do not consider `define` as an algorithmic constructor because `define` doesn't really create a new compound algorithm. Instead, `define` allows us to give a *name* to (compound) algorithms. In the aforementioned factorial example, it is `lambda` that creates the compound factorial algorithm. `define` is merely used to associate the name `fac` with the algorithm. `define` therefore allows us to abstract away from the details of a compound algorithm by referring to its name instead of referring to its constituting details. This is known as

procedural abstraction.

1.3 Procedural Abstraction and Procedure Types

Given the data constructors and algorithmic constructors discussed above, programmers can create an infinite number of data structures and algorithms. However, it soon appears that the intellectual effort to understand the complexity of large data structures and big algorithms gets enormous. This is very problematic since understanding algorithms and data structures (written by others) is one of the most important activities performed by programmers. Programmers need to understand algorithms and data structures whenever they try to use other people's work for example when they have to maintain a software system that was written by people that are no longer around (because they have moved to a different company e.g.).

In order to reduce the complexity arising from huge amounts of applications of data constructors and algorithmic constructors, computer scientist have developed abstraction techniques. Abstraction is one of the key means to tackle complexity in any science and this is no different in computer science. The major way to realize abstraction in computer science is by giving a name to complex things. In the case of complex data structures, this is called *data abstraction*. In the case of algorithms it is called *procedural abstraction*.

Procedural abstraction consist of giving a name to a Scheme procedure by means of `define`. This has two important benefits. First, by giving a meaningful name to a compound algorithm, someone reading the algorithm will understand much easier what the algorithm is about. Indeed, the classical version of our good old factorial algorithm is much easier understood when the name `fac` is used instead of just any other name (like, say `qfxttq`). In other words, procedural abstraction — when correctly used — can enhance the readability of programs enormously. Second, by giving a name to a compound algorithm, it becomes possible to refer to the compound algorithm simply by mentioning its name in a procedure call. This means that we do not need to copy the algorithm whenever we need it. We simply call it by using its name. In other words, procedural abstraction avoids the duplication of code. This is an important factor in computer science since duplication of code also means duplication of errors and duplication of maintenance efforts.

We conclude that programs that heavily rely on procedural abstraction are easier to read and easier to maintain. Of course, it is crucial to use good names in doing so. Using such names as `temp` or `my-procedure` soon results in programs that are totally unreadable.

Now that we know what primitive algorithms, compound algorithms, algorithm constructors and procedural (i.e. algorithmic) abstraction are, we can turn our attention to the notion of an *algorithmic type*, also called *procedural type* or simply *type of a procedure*. The type of a procedure is a description of the data types of the data values to be used as the input of the procedure, along with the data type of the output values that are produced by the procedure.

For example, for the primitive procedure `sin` we will say that its procedure type is (`number` → `number`). We use this expression to express that the type of `sin` is "from number to number". Similarly, we can say that the type of `reverse`, the compound Scheme procedure for reversing lists, is (`pair` → `pair`). The procedure takes a dotted pair (representing a list) and produces a dotted pair (representing the reversed version of the input list). We say `reverse` is "from dotted pair to dotted pair". Whenever a procedure has more than one argument, we simply enumerate all their data types on the left hand side of the arrow. For example, the type of `append` (which consumes two lists and returns the list representing the second list appended to the first one) is (`pair pair` → `pair`).

Finally, let us have a look at the procedural type of a higher procedure. Consider for example the procedure (`zero f a b epsilon`) which computes a number `v` between `a` and `b` such that (`f v`) equals zero with a precision `epsilon`. Its implementation is as follows:

```
(define (zero f a b epsilon)
  (define c (/ (+ a b) 2))
  (cond ((< (abs (f c)) epsilon) c)
        ((< (* (f a) (f c)) 0) (zero f a c epsilon))
        (else (zero f c b epsilon))))
```

Clearly, this procedure takes four arguments, the three latter of which are of type `number`. But what is the type of `f`? The algorithm assumes that `f` is a procedure that operates on numbers. Hence, the data type of `f` is (`number` → `number`). Therefore, the type for `zero` is:

((`number` → `number`) `number number number` → `number`)

This procedural type expression formally shows us that `zero` takes four arguments: one function of type (`number` → `number`) and three other arguments of type `number`. The result of the procedure is a `number`.

It is important to notice that procedural type expressions like (`number` → `number`) are *not* a part of the Scheme programming language. Instead, they are a convention that we use to specify the types of the arguments and the results of Scheme procedures.

1.4 Data Abstraction: Abstract Data Types (ADTs)

Just like procedural abstraction allows us to give a name to a complex compound algorithm, *data abstraction* allows us to group a number of data elements into one single compound data element by giving it a meaningful name. As an example, suppose we are to write a mathematical software system that helps engineers perform complicated calculations using complex numbers. As Scheme programmers, we might be tempted to think of complex numbers as dotted pairs in which the car is used to store the real part of a complex number and in which the cdr is used to store its imaginary part. In this setting, the algorithms that

operate on complex numbers refer to (`car c`) whenever they need the real part of some number `c`. Similarly, (`cdr c`) would be used to refer to `c`'s imaginary part. Although this solution is simple, it has two important drawbacks that have huge repercussions on the maintainability of software systems.

- First, we observe the same problems w.r.t. readability of programs as the ones described in the section on procedural abstraction. Not using meaningful names to denote data structures and the elements they contain means that all data manipulation has to be done directly in terms of Scheme's accessors like `car`, `cdr` and `vector-ref`. For complex data structures this can result in extremely complex expressions. E.g., just from reading a program, one would probably never guess that an expression like (`vector-ref` (`caddr` (`vector-ref invoices` 10)) 12) might designate the 12th most important company that sells the product that was sold by the company listed on the 10th invoice stored by our system. Instead of thinking in terms of "a vector of lists of vectors", it is much more comprehensible to think in terms of "a collection of companies that sell products whose product name on the invoice contained in the collection of invoices". Hence, it pays off to give data structures a meaningful name.

- Apart from the readability of the procedures that operate on a data structure, adaptability is also an important issue when it comes to designing data structures. E.g. in the mathematical software system described above, one might decide to change the representation of complex numbers from dotted pairs to vectors with two entries. Without abstraction, this would require one to manually replace all occurrences of `car`'s and `cdr`'s that relate to the manipulation of complex numbers by equivalent calls to `vector-ref`. Moreover, one cannot *blindly* replace all occurrences of `car` and `cdr` in the code since there are probably many occurrences of `car` that have nothing to do with complex numbers. Obviously, those occurrences do not have to be replaced. Needless to say, manually looking for those occurrences that do need to be replaced is a very error-prone process since the chances that one misses one such occurrence are quite high.

To alleviate these drawbacks, computer scientists resort to data abstraction by using a device called *Abstract Data Types* or *ADT*s for short. An abstract data type is a data type in the sense of the definition presented in section 1.1. In other words, it is a name for a set of data values. The idea of a data type being abstract means that the name is used to draw an abstraction border between the *users* of the ADT and the *implementor* of the ADT. As a consequence, the implementor of the ADT can choose whichever *implementation of the ADT* that he wants to provide. For example, given the abstract data type `complex`, a programmer might want to use dotted pairs to represent data values of type `complex`. In a later version of the system, he might want to choose vectors (of size two) to represent the data values of type `complex`. However, an ADT is more than just a new name for a compound data type. Remember from section 1.1 that a data type is inherently associated with the name and the procedural

types of the procedures that operate on the data values of that data type. This is also the case for *abstract* data types. The procedural types of these applicable procedures form an integral part of the definition of the ADT. We therefore define:

An **abstract data type** *is a name for a data type along with a set of procedural types that prescribe the kind of procedures (i.e. constructors, accessors, mutators and operations) that can be applied to the data elements of the data type.*

As an example, the following shows the ADT `complex`. Firstly, the ADT specifies a name for a set of data values: `complex`. It is the data type for those data values. Secondly, the ADT also lists the procedural types for the constructor, the accessors (this ADT does not have mutators) and the operations that are to operate on the data values of type `complex`. `new` is the name of the constructor. Its procedural type teaches us that it takes two Scheme numbers and that it returns a new complex number that has the two given numbers as real and imaginary parts. `real` and `imag` are the accessors that can be used to access these parts. `complex?` is an operation that can be applied to any scheme value. It returns `#t` if that value is a complex number and `#f` in all other cases. The other procedure types specify the operations that can be performed on complex numbers. Notice that we use the name `any` to indicate the data type consisting of all possible Scheme values.

```
1  ADT complex
2
3          new        ( number number → complex )
4          complex?   ( any → boolean )
5          +          ( complex complex → complex )
6          −          ( complex complex → complex )
7          /          ( complex complex → complex )
8          *          ( complex complex → complex )
9          modulus    ( complex → number )
10         argument   ( complex → number )
11         real       ( complex → number )
12         imag       ( complex → number )
```

Given a particular complex number data value `c` that was constructed using `new`, the fact that its data type is `complex` really is the only thing that users can assume about `c`. The concrete implementation of the procedures and the concrete representation of `c` is hidden from the user of the ADT. Users of the ADT are only allowed to manipulate complex numbers using these abstract descriptions. This is the essence of data abstraction.

A programmer that wishes to *implement* the ADT must decide on a concrete *representation* for the data values of type `complex`. He could choose to represent those values as dotted pairs containing the real and imaginary parts. However, he could also choose to represent the values as dotted pairs containing their argument and modulus. After all, given the real and the imaginary part, one can

always compute the modulus and the argument of a complex number, and vice versa. Many different representations are possible. E.g., in a system where speed is of utmost importance, he could avoid the computations converting between real and imaginary parts, and, argument and modulus. To achieve this, he could represent complex numbers as vectors of size four in order to explicitly store the real part, the imaginary part, the modulus and the argument. A virtually infinite number of representations for the ADT's data values are imaginable. For each representation, a different implementation will need to be provided for all the procedural types listed in the ADT's definition. According to the principle of data abstraction, users of the ADT cannot explicitly rely on this representation but use the ADT's procedures instead. This makes their code much more readable as that code is clearly about complex numbers instead of dotted pairs or vectors. It also makes the user code easier to adapt since it is independent of the implementation of the ADT.

Before we move our attention to the definition of some other useful ADTs, let us first have a look at how to implement ADTs in Scheme.

1.4.1 Implementing Data Abstraction in Scheme

Scheme has two radically different programming styles that can be used to implement ADTs. They are known as *the procedural style* and *the object-based style*. We present them one after the other.

Procedural Style

The simplest way to implement ADTs in Scheme is to write Scheme procedures for the constructors, accessors, mutators and operations as specified by the ADT definition. Let us have a look at an implementation for the `complex` ADT presented in the previous section. The implementation is said to follow the procedural style because all the operations are implemented using plain Scheme procedures that take the data elements of the ADT as explicit arguments.

The procedural implementation shown below uses R6RS Scheme's library system. A library can be thought of as a collection of Scheme names, some of which are exported towards users of the library. Every library has a name (`complex` in our case), has an `export` clause listing the names it exports and has an `import` clause that specifies which other libraries it relies on. The import clauses allows the library to access the names that are exported by these libraries in its turn. For instance, by stating (`import` (`rnrs io simple`)), a library can access Scheme's standard input/output-facilities such as `display` and `newline`. A simple import clause such as (`rnrs io simple`) imports *all* the names exported by the corresponding libraries. Sometimes, this can be problematic when names imported from two different libraries are identical. In order to resolve such situations, we can chose not to import certain names using the `except` clause. By importing (`except` (`rnrs base` (6) `complex?`) we state that we are importing the library (`rnrs base` (6)) except for the procedure `complex?`. In our case, this is because the procedure `complex?` that we will program as part of our `complex`

ADT has the same name as the standard procedure `complex?`. We therefore decide not to import the standard procedure. Sometimes we need to resolve a name clash by renaming a number of procedures imported from a library. This is accomplished using the `rename` import clause. E.g., by using the clause (`rename` (`rnrs base` (6)) (+ `number+`) (∗ `number∗`)) we state that we are importing the standard library (`rnrs base` (6)) but that we desire to refer to the standard procedures + and ∗ by using the names `number+` and `number∗`. This allows us to use the names + and ∗ without ending up with name clashes.

The following code shows the procedural implementation of the `complex` ADT. It shows a library that provides an implementation for all the procedures listed in the `ADT` definition. This implementation represents complex data values as lists of length three: a tag used to identify complex numbers, and two numbers that correspond to the real and the imaginary part of a complex number.

```
(library
 (complex)
 (export new complex? real imag + − / ∗ modulus argument)
 (import (rename (except (rnrs base (6))
                         complex?)
                 (+ number+) (∗ number∗) (/ number/) (− number-)))

 (define complex-tag 'complex)
 (define (get-real c)
   (cadr c))
 (define (get-imag c)
   (caddr c))

 (define (new r i)
   (list complex-tag r i))

 (define (complex? any)
   (and (pair? any)
        (eq? (car any) complex-tag)))

 (define (real c)
   (get-real c))

 (define (imag c)
   (get-imag c))

 (define (+ c1 c2)
   (define real (number+ (get-real c1) (get-real c2)))
   (define imag (number+ (get-imag c1) (get-imag c2)))
   (new real imag))

 (define (∗ c1 c2)
   (define real (number- (number∗ (get-real c1) (get-real c2))
```

```
                        (number* (get-imag c1) (get-imag c2))))
  (define imag (number+ (number* (get-real c1) (get-imag c2))
                        (number* (get-imag c1) (get-real c2))))
  (new real imag))

(define (- c1 c2)
  (define real (number- (get-real c1) (get-real c2)))
  (define imag (number- (get-imag c1) (get-imag c2)))
  (new real imag))

(define (/ c1 c2)
  (define denom (number+ (number* (get-real c2)
                                  (get-real c2))
                         (number* (get-imag c2)
                                  (get-imag c2))))
  (define real (number+ (number* (get-real c1)
                                 (get-real c2))
                        (number* (get-imag c1)
                                 (get-imag c2))))
  (define imag (number- (number* (get-imag c1)
                                 (get-real c2))
                        (number* (get-real c1)
                                 (get-imag c2))))
  (new (number/ real denom) (number/ imag denom)))

(define (modulus c)
  (sqrt (number+ (number* (get-real c) (get-real c))
                 (number* (get-imag c) (get-imag c)))))

(define (argument c)
  (atan (get-imag c) (get-real c))))
```

Some of the Scheme procedures (`get-real` and `get-imag` to be precise) of the above library are merely needed to implement the operations of the ADT in a more convenient way; they are not part of the definition of the `complex` ADT itself. These procedures are said to be *private to the ADT implementation*. Technically, this is simply achieved by not listing these procedures in the `export` clause of the library. As a consequence, programs (or other libraries) importing the library will not be able to access those names. In our example, `get-real` and `get-imag` are not part of the `export` clause of our `complex` library.

The following Scheme program illustrates how to use an ADT that was implemented in the procedural style. The program imports three libraries: the standard library (`rnrs base` (6), the `complex` library[1] and the standard in-

[1]The number (1) designates the version of in the library in case different versions exist. In our case, this is indeed the case. The first version is the library just discussed. The second version is the object-styled implementation studied in the following section.

put/output library (`rnrs io simple`). Notice that importing our complex library and the standard library result in name clashes. E.g., both provide procedures named +, ∗ and so forth. We therefore decided to refer to the procedures of our `complex` ADT using locally different names: the (`prefix` (`a-d examples complex` (1)) `complex:`) states that we desire to import the procedures of the (`a-d examples complex` (1)) library by first prefixing all its exported names with `complex:`. Like this, we can refer to the standard addition procedure for numbers (using the name +) as well as to our complex addition procedure (using the name `complex:+`).

```
(import (prefix (a-d examples complex (1)) complex:)
        (rnrs base (6))
        (rnrs io simple))

(define cpx1 (complex:new 1 4))
(define cpx2 (complex:new 5 3))
(display (complex:+ cpx1 cpx2)) (newline)
(display (complex:∗ cpx1 cpx2)) (newline)
(display (complex:- cpx1 cpx2)) (newline)
(display (complex:/ cpx1 cpx2)) (newline)
(display (complex:real cpx1))   (newline)
(display (complex:imag cpx2))   (newline)
(display (complex:modulus cpx1)) (newline)
(display (complex:argument cpx2))
```

Object-based Style: Encapsulation

One problem with the procedural style of implementing ADTs is that the abstraction barrier offered by the ADT is a mere convention that can easily be circumvented. Consider a complex number `c` that was constructed using the aforementioned ADT implementation. It suffices to evaluate the expression (`cadr c`) to bypass the abstraction barrier of the ADT. In other words, the procedural style of programming ADTs is entirely based on conventions. It does not provide us with a way to *enforce* the abstraction barrier. This is exactly what the object-based style of of implementing ADTs tries to achieve. To accomplish this, it uses a powerful technique to *encapsulate* the representation details of the ADT's data values inside a Scheme procedure, called a *dispatcher*.

As an example, we present an implementation of the `complex` ADT using the object-based style. The precise technical details of this particular implementation are not important right now: the implementation uses a number of advanced Scheme features such as variable size argument lists (notice the . in the dispatcher) which are not always widely known among beginning Scheme programmers. The focus of the discussion is on the way the ADT is implemented: the constructor `make-complex` returns a dispatching procedure that represents a data value of the ADT.

```
(define (make-complex r i)
```

```
(define (complex+ c)
  (make-complex (+ r (c 'real))
                (+ i (c 'imag))))
(define (complex* c)
  (make-complex (- (* r (c 'real))
                   (* i (c 'imag)))
                (+ (* r (c 'imag))
                   (* i (c 'real)))))
(define (complex- c)
  (make-complex (- r (c 'real))
                (- i (c 'imag))))
(define (complex/ c)
  (define denom (+ (* (c 'real)
                      (c 'real))
                   (* (c 'imag)
                      (c 'imag))))
  (define real (+ (* r (c 'real)) (* i (c 'imag))))
  (define imag (- (* i (c 'real)) (* r (c 'imag))))
  (make-complex (/ real denom) (/ imag denom)))
(define (modulus)
  (sqrt (+ (* r r) (* i i))))
(define (argument)
  (atan i r))
(define (real)
  r)
(define (imag)
  i)
(lambda (message . args)
  (cond ((eq? message '+) (apply complex+ args))
        ((eq? message '-) (apply complex- args))
        ((eq? message '*) (apply complex* args))
        ((eq? message '/) (apply complex/ args))
        ((eq? message 'modulus) (modulus))
        ((eq? message 'argument) (argument))
        ((eq? message 'real) (real))
        ((eq? message 'imag) (imag))
        ((eq? message 'complex->list) (list 'complex r i))
        (else (error "Complex Number Message Not Understood")))))
```

The aspects of the ADT's implementation which are important to understand is that there is only one single procedure (namely `make-complex`) installed in the global environment. That procedures "captures" (technically we say *encapsulates*) a number of local procedures (such as `real`) which are stored in the local environment of the anonymous dispatcher that is returned from the constructor. The returned dispatcher is referred to as an *object* in this style of programming since it can be considered as an indivisible thing. It has to be

called with quoted symbols (such as '+). These are said to be *messages* that are *sent* to this object. This is the reason why this style of implementing ADTs is called the object-based style. Notice that the technique of using dispatchers automatically makes private procedures inaccessible to users of the ADT. They are said to be encapsulated along with the data representing the data values of the ADT.

This implementation style has a number of advantages over the procedural implementation:

Encapsulation By encapsulating the representation details of the ADT in a dispatcher, we are *guaranteed* not to externalise these details. As a consequence, user programs will never be able to rely on these details which makes it much easier to adapt the representation of the ADT without affecting the validity of the user program. In other words, encapsulation facilitates the maintenance of both the implementation of the ADT as well as the maintenance of the code that uses the ADT.

Name Space Management Only the constructor for the ADT's data values is installed in the global Scheme environment. All procedures private to the implementation of the ADT are stored inside the dispatcher's local environment of definition. As a result it gets easier to manage one's name space in systems with lots of ADTs. In the procedural implementation style, each and every procedure of each and every ADT is included in the global environment. This makes it much harder to avoid name clashes between different procedures that belong to different ADT.

Code Size The code for the operations in the object-based style is often smaller than the code in the equivalent procedural style implementation. Because of Scheme's lexical scoping rules, one typically needs less accessors. In the `complex` example we use `r` and `i` whereas the procedural style implementation has to call the accessor procedures `get-real` and `get-imag` for obtaining these values.

However, the style also has a number of disadvantages:

Encapsulation Breached The object-based style only lives up half to its expectations. E.g., in the implementation of binary operations (i.e. operations that need two arguments) one needs two objects: the object receiving the message and the argument object. As the implementation for complex number addition shows, it is necessary to bypass the encapsulation of the argument object in order to get access to its implementation details. In the case of the addition, it is necessary to access the real and the imaginary part of the argument in order to be able to calculate the sum of the complex numbers. In our `complex` example this is not a problem since `real` and `imag` are a part of the ADT specification. However, suppose that the designer of the `complex` ADT would decide to remove these two accessors from the ADT. Our implementation for the addition would still

need them. In other words, the object-based style sometimes requires one to add accessors just to be able to implement binary operations, even when those accessors are not part of the ADT definition. Hence, encapsulation is not guaranteed.

Code Complexity Object-based code is often a bit more complex than the equivalent procedural code. As we can see from the implementation, we need more complex scoping. Furthermore, advanced features such as variable length arguments (notice the dot in the dispatcher) are needed. The code is also a bit slower because of the conditional in the dispatcher.

Space Inefficiency The object-based style has a huge problem when it comes to space efficiency. Looking back at the object-based implementation of our `complex` ADT. *Every* time we call the constructor procedure `new`, local procedures such as `complex+` are created that implement the operations of the ADT. In our case, there are seven such local procedures. This means that in a system with one thousand complex numbers, we have seven thousand procedures stored in our computer memory, only seven of which are distinct. Solving this problem requires us to factor out these procedures from the constructor. But then we gradually move back to an implementation with global procedures as is the case with the procedural style.

Although good solutions exist to alleviate these drawbacks, they render the resulting Scheme code more complex. These solutions deserve attention in advanced courses on object-oriented programming. They fall beyond the scope of this text. Therefore, we have opted for the procedural ADT implementation style in this text.

1.4.2 Genericity

From chapter 3 on, we will be *storing* data elements in data structures. A data structure whose main purpose it is to "store" (i.e. "remember") data values for later retrieval is called a *storage data structure* and the data values which it stores are called *records*. Typical examples of storage data structures include a phone index and a library catalogue. These can be considered as data structures whose purpose it is to store personal data and books for later retrieval. This is in sharp contrast with the `complex` ADT, the implementation of which (see section 1.4.1) is technically spoken also a data structure (since it combines two Scheme numbers into a compound data value that represents the complex number). However, we can hardly claim that the raison d'être of complex numbers is to *store* data elements for later retrieval. That is why we do not say that the real and the imaginary values sitting in a complex number are records.

The following example shows the `max-o-mem` ADT. A max-o-mem is a storage data structure that can remember exactly one record. The idea is that user programmers can keep on writing values to the max-o-mem, but that the max-o-mem only remembers the "greatest" value it was ever given. The ADT itself

is very simple. It has a constructor `new`, a procedure `max-o-mem?` that can be used to verify whether or not a given scheme value is a max-o-mem, a procedure `write!` to write a value to the max-o-mem (which will be ignored in case the max-o-mem already contains a value that was greater) and a procedure `read` to read the max-o-mem's "current greatest" value.

```
1 ADT max-o-mem< T >
2
3       new        ( ( T T → boolean ) T → max-o-mem< T > )
4       max-o-mem? ( any → boolean )
5       write!     ( max-o-mem< T > T → max-o-mem< T > )
6       read       ( max-o-mem< T > → T )
```

What can we say about the procedural type of a procedure such as `write!`? Clearly, it requires a max-o-mem and an data element that has to be written to that max-o-mem. It returns the (potentially) modified max-o-mem. On first sight, this results in a procedural type for `write!` that is given by (`max-o-mem any` → `max-o-mem`). But is it really possible to write any scheme value to a max-o-mem? The answer is no. In order to understand this, try to imagine an implementation of `write!`. At some point it will have to compare the procedure's argument with the value that is currently stored in the max-o-mem. In order to do so, it might use the Scheme procedure < ("lesser than"). However, this implies that the max-o-mem can only store Scheme numbers. Suppose that we want to use a max-o-mem to store complex numbers as defined by the ADT presented in the previous section. How can we tell the max-o-mem ADT implementation not to use Scheme's < but to use our own special procedure `complex.<` instead? This is achieved by *parametrizing* the `max-o-mem` ADT by the data type `T` of the data elements it will contain. This parametrisation of the ADT is denoted using angular brackets < and > in the name of the ADT. Hence `max-o-mem<T>` refers to "a max-o-mem that can store data elements of data type `T`". If we use a max-o-mem to store numbers, we say the max-o-mem has the data type `max-o-mem<number>` (i.e. selecting `T = number`). If we use a max-o-mem to store dotted pairs, we refer to that max-o-mem as having data type `max-o-mem<pair>` (i.e. we select `T=pair`). Using this knowledge, we can explain the procedural type of `read` in the ADT. `read` takes any max-o-mem that stores elements of type `T`, i.e. it takes a parameter of type `max-o-mem<T>`. It returns the greatest value currently stored by that max-o-mem, i.e. a value of type `T`. Hence, `read`'s procedural type is (`max-o-mem<T>` → `T`). Similarly, `write!` takes a `max-o-mem<T>` and a value of type `T`. It returns the modified max-o-mem of type `max-o-mem<T>`. Hence its procedural type is (`max-o-mem<T> T` → `max-o-mem<T>`).

How can we provide an implementation in Scheme for such an ADT that is parametrized with a data type `T`? I.e., how can we implement a data structure that does not depend on the data type of the records it stores? In the case of the `max-o-mem` ADT, all we have to do is to make sure that the code of the implementation does not contain a hardwired reference to Scheme's <

procedure. This can be easily done by providing the correct version of "lesser than" when constructing a max-o-mem. Max-o-mems that are required to store Scheme numbers will be provided Scheme's <. Max-o-mems that are required to store complex numbers can be provided our own special procedure `complex<`. Hence, the constructor `new` has to be a higher order procedure that takes a "lesser than" procedure as its first argument. Any such "lesser than" procedure is to decide whichever is the smallest value, given two values of data type `T`, i.e. it has procedural type (`T T` → `boolean`). `new`'s second argument is the initial value stored in the max-o-mem. It is therefore of data type `T`. This explains the procedural type of `new`. It takes a "lesser than" procedure and an initial smallest element. It returns a max-o-mem of type `max-o-mem<T>`. Hence it has procedural type ((`T T` → `boolean`) `T` → `max-o-mem<T>`).

ADTs representing storage data structures that are independent of the data type of the records they are supposed to store are known as *generic ADTs*. At the implementation level, generic ADTs are implemented by *generic data structures*. Generic data structures are data structures whose implementation is independent of the data type of the records they store. In Scheme, generic data structures are realized by turning their constructor into a higher order function that take all the procedures that *do* depend on the data type of the records stored (such as the "lesser than" in our example). Using the procedural style, the implementation for the `max-o-mem` ADT now looks as follows:

```
(library
 (max-o-mem)
 (export new max-o-mem? read write!)
 (import (rnrs base (6))
         (rnrs mutable-pairs))

 (define max-o-mem-tag 'max-o-mem)

 (define (new << init)
   (list max-o-mem-tag << init))

 (define (max-o-mem? any)
   (and (pair? any)
        (eq? (car any) max-o-mem-tag)))

 (define (read mom)
   (caddr mom))

 (define (write! mom new-value)
   (define << (cadr mom))
   (define value (caddr mom))
   (if (<< value new-value)
     (set-car! (cddr mom) new-value))
   mom))
```

Let us have a look at the implementation for the constructor `new`. The idea is to represent a max-o-mem as a tiny list with three elements: the type tag 'max-o-mem, the "lesser than" procedure << and the max-o-mem's initial value `init`. In the implementation of `write!` we observe how the max-o-mem's *own* "lesser than" operator is selected using (`cadr mom`) and subsequently used for the comparison. We can make max-o-mems of numbers by calling the constructor (`max-o-mem.new < v`) where `v` is some initial value for the max-o-mem. If we would like to make a max-o-mem storing complex numbers in which we consider $a + b.i < c + d.i$ whenever $\sqrt{a^2 + b^2} < \sqrt{c^2 + d^2}$, then we proceed as follows:

```
(import (rnrs base (6))
        (prefix (a-d examples max-o-mem) mom:)
        (prefix (a-d examples complex (2)) complex:))


(define (complex< c1 c2)
  (define (square x) (* x x))
  (< (sqrt (+ (square (complex:real c1))
              (square (complex:imag c1))))
     (sqrt (+ (square (complex:real c2))
              (square (complex:imag c2))))))

(define complex-mom (mom:new complex< (complex:new 0 0)))
```

The following code excerpt shows how to use these abstractions to write a procedure `greatest` that computes the greatest element of a list of data elements by iterating over the elements of the list and by sequentially storing all those elements in a max-o-mem. At the end of the procedure, the max-o-mem contains the greatest element. It is read and returned as the result of the procedure.

```
(import (rnrs base (6))
        (rnrs mutable-pairs (6))
        (prefix (a-d examples max-o-mem) mom:)
        (prefix (a-d examples complex (2)) complex:))


(define (greatest lst << init)
  (define max (mom:new << init))
  (define (iter lst)
    (mom:write! max (car lst))
    (if (not (null? (cdr lst)))
      (iter (cdr lst))))
  (iter lst)
  (mom:read max))


(define integer-list (list 1 2 3 4 5))
(define complex-list (list (complex:new 1 0) (complex:new 0 1)
                           (complex:new 3 4) (complex:new 4 3)))
```

The point of all this is that we have turned `max-o-mem` into an ADT the constructor of which requires us to pass along an abstract "lesser than" operator. This extra parametrisation turns the `max-o-mem<T>` ADT into a generic ADT since its specification has gotten independent from the data type `T` of the records that it stores. The max-o-mem is therefore much more reusable than would be the case if we would have used some specific built-in "lesser than" operator from Scheme.

1.4.3 The Dictionary ADT

One of the central ADTs studied in this text is the `dictionary` ADT. The `dictionary` ADT is an abstraction that is used in many computer applications. The most trivial example is an actual dictionary program, e.g. a dictionary Dutch-English. The characterising property of dictionaries is that they store so called key-value pairs. In other words, they associate *keys* with *values*. Therefore, dictionaries are also called *associative memories*. In the Dutch-English example, keys are Dutch words. With every Dutch word, a list of translations is associated. This list is the value that is associated with the key. However, the `dictionary` ADT is an abstraction that has many applications that go beyond translating dictionaries. E.g., a phone index associates an address and a phone number with any combination of a first name and a family name. It is said that a phone index is a dictionary that has the names as keys and that has the address and the phone number pairs as values. Similarly, a library catalogue might be considered as a `dictionary` in which names of authors form the keys and in which a book title along with an ISBN number form the value. In all three examples, a `dictionary` is a data structure that associates keys (i.e. Scheme values of a certain data type `K`) with values (i.e. Scheme values of a certain data type `V`). E.g., the Dutch-English dictionary can be thought of as a data structure having data type `dictionary<string list-of-strings>` (i.e. `K` = `string` and `V` = `list-of-strings`).

The `dictionary` abstraction barrier is formally defined by the following ADT specification:

```
1  ADT dictionary < K V >
2
3  new
4      ( ( K K → boolean ) → dictionary < K V > )
5  dictionary?
6      ( any → boolean )
7  insert!
8      ( dictionary < K V > K V → dictionary < K V > )
9  delete!
10     ( dictionary < K V > K → dictionary < K V > )
11 find
12     ( dictionary < K V > K → V ⋃ {#f} )
13 empty?
```

```
14    ( dictionary < K V > → boolean )
15  full?
16    ( dictionary < K V > → boolean )
```

In this ADT definition, `empty?` and `full?` are predicates that can be used by user programs in order to check whether or not a dictionary is empty or full. They both take a dictionary of data type `dictionary<K V>` and return a `boolean`. The most interesting operations are `insert!`, `delete!` and `find`. `insert!` takes a dictionary of type `dictionary<K V>`, a key or type `K` and an associated value of type `V`. It adds the key-value pair to the dictionary and returns the destructively modified dictionary; i.e. a value of type `dictionary<K V>`. `delete!` takes a key and removes the corresponding key-value pair from the dictionary on the condition that the dictionary contains a key-value pair the key of which matches `delete!`'s argument. The destructively modified dictionary is returned from the operation. `find` takes a dictionary and a key. It searches the dictionary for the corresponding key-value pair and returns the value that corresponds to the key. `#f` is returned if the dictionary does not contain a key-value pair that matches the given key. During this searching process, `find` will need a procedure to check whether or not two keys of type `K` are the same. Therefore, the constructor `new` requires an equality operator that is to be used for checking equality on the key data type `K`. Obviously, `new` returns a newly created dictionary of type `dictionary<K V>`.

Remember that the data values that reside in a storage data structure are called records. The individual data values that make up records are called the *fields* of the record. In our library catalogue the title, the author and the ISBN number are all fields which together constitute one record sitting in the catalogue. The fields of the record that identify the record are said to be the *key fields* (or *key* for short) of the record. All the other fields (i.e. the fields that constitute the value) are said to be *value fields*, *peripheral fields* or *satellite fields*. Hence, in a dictionary, every record consists of the key fields along with the value fields. E.g., in a phone index, the name of the person to search for is typically considered the key field while the other data (e.g. the phone number and the address) is peripheral. Using this new terminology, we can say that it is the task of `find` to find the satellite fields that are associated with the requested key fields.

As already explained, dictionaries are used extremely frequently in computer applications. Therefore, large parts of the text are devoted to the study of different implementations of the `dictionary` ADT.

1.5 Measuring Algorithm Performance

One of the reasons for studying algorithms and data structures in a systematic way is to compare their characteristics in order to determine which algorithm or which data structure is "best". But how can we tell whether an algorithm is "good" or when one algorithm is "better" than another? To answer this question

we have to look at the economic factors that are relevant in computing: people find computers "good" when they are fast and when they have lots of memory. This means that — since time and memory are scarce — software is "good" if it doesn't use a lot of memory and if it is fast. We will therefore try to estimate how much time and how much memory an algorithm consumes.

1.5.1 Performance Measure 1: Speed

In order to measure the amount of time needed to execute an algorithm, we might be tempted to consider using a stopwatch. Another possibility — in some Scheme implementations — is to use the expression (`time expression`) which times the evaluation of the `expression` and puts the resulting numbers on the screen. Hence, we might consider using (`time` (`algorithm input`)) in order to find out how long the `algorithm` runs on the given `input`. We might run such experiments for a few (or even for an extensive set of) inputs in order to get a statistically relevant table of values that describes the speed of an algorithm. This is known as *the experimental approach* to measuring algorithm performance and it has a number of important disadvantages:

Non-Generality The experimental approach is not a general approach. Suppose we have an algorithm and suppose we have tested its speed for a number of different inputs. Suppose we have tried to get a general picture of the algorithm's performance based on the test data. How can we be sure that the algorithm really performs in the way prescribed by that general picture? In fact we cannot. Take for example the problem of sorting a list of data elements. In chapter 5 we will see examples of sorting algorithms that appear to perform quite well for *most* input data, but which have horrible performance characteristics for some pathological cases of input data. The famous QuickSort algorithm (see section 5.4.1) is one of the fastest sorting algorithms for most input data. However, if the input data is already sorted, it becomes extremely slow. This shows us that timing an algorithm with a stopwatch is not enough and that a deeper understanding is necessary about why an algorithm performs in a certain way. Just putting some experimentally acquired numbers in a table is not general enough.

Absoluteness A problem with the experimental approach is that it gives us absolute numbers. These numbers depend on a particular implementation of the algorithm (i.e. the test implementation). This implementation and the experimental data it generates are heavily influenced by the particular hardware it is run on, the Scheme evaluator (or compiler) that was used, the operating system on which it runs, and so on. As a result, the test data gets quite useless after a few years have gone by: processors get faster and Scheme implementations get optimised. The test data is said to be absolute data. What is really needed to compare two algorithms is a *relative* measuring technique which allows us to select the best algorithm

by comparing those algorithms. This is impossible if we are provided with absolute test data that was supplied by two unrelated experiments. Our only option would be to implement both algorithms again (on *our* computer, using *our* Scheme on *our* operating system) and conduct the experiments all over again. What we really need is a general technique that allows us to pick two algorithms "from the shelf" along with their performance characteristics, such that they allow us to tell us something about how the algorithms perform w.r.t. one another. In other words, we need a relative way for measuring algorithm performance.

Because of these two important disadvantages, we do not pursue the experimental track any further. Instead we present a technique that takes *all* possible inputs into account and that allows us to *compare* two performance studies. The basic idea of the technique is to consider an algorithm in terms of the size n of its input and to determine the *number of computational steps* the algorithm needs for an input of that size. For example, we might be interested in how many computational steps the famous QuickSort algorithm needs in order to sort n numbers. Clearly, this amount depends on n. Hence we try to determine a function $f_A(n)$ that gives us a count for the number of computational steps needed to execute an algorithm A on inputs of size n. This function is known as the *performance characteristic of the algorithm.* Notice that the function can be a constant function. For instance, $f_{\texttt{car}}(n) = 1$ for the `car` algorithm (which can be applied to any list of length n) because `car` is always equally fast. However, in most cases, $f(n)$ will not be constant. Given two algorithms A and B and given their performance characteristics $f_A(n)$ and $f_B(n)$, then comparing both algorithms is a matter of (relatively simple) mathematics. For example, if we have two algorithms A and B such that $f_A(n) = 5n$ and $f_B(n) = 10n$ then we can say that A is twice as fast as B.

Let us now try to establish such a performance characteristic. Let us consider the following Scheme procedure `greatest` which takes a list of positive numbers and which returns the greatest element of that list. Our goal is to come up with a function $f_{\texttt{greatest}}(n)$ which gives us an expression (depending on n, the length of the list that gets bound to the `lst` parameter) that allows us to calculate the amount of computation needed to execute the algorithm.

```
(define (greatest l)
  (define (iter el l)
    (cond
       ((null? l) el)
       ((< el (car l)) (iter (car l) (cdr l)))
       (else (iter el (cdr l)))))
  (iter 0 l))
```

We will try to analyse as precisely as possible the number of computational steps that this procedure performs, given a list `l` = (`cons a d`) of length n (i.e., the length of `d` is $n - 1$). We will use T(`(greatest l)`) to denote the number of computational steps needed to perform the call to `greatest` with `l`

as its argument. Hence, $f_{\texttt{greatest}}(n) = T(\texttt{(greatest l)})$ where `l` is a list of length n. When calling `greatest`, a call to `iter` immediately follows. Hence, $T(\texttt{(greatest l)}) = T(\texttt{(iter 0 l)}) + 1$. Inside `iter`, a `null?` test is executed and subsequently (if the list is not empty), the car of the list is compared to the current `el`. In Scheme, all this requires 3 computational steps. If this comparison succeeds, a recursive call is made after having called `car` and `cdr` (i.e. 3 steps) or a recursive call is made after having called only `cdr` (i.e. 2 steps). This means that, either $T(\texttt{(iter el l)}) = 6 + T(\texttt{(iter a d)})$, or $T(\texttt{(iter el l)}) = 5 + T(\texttt{(iter el d)})$. In the very last step of the recursion, all we have to do is the `null?` test which is a single computational step. As a result, we have two extreme situations. In the worst-case scenario, the comparison fails in every step of the recursion. This only happens when the list contains its numbers in ascending order, such that a new maximum is found in every call of `iter`. Consequently, we have $T(\texttt{(iter el l)}) = 6n + 1$, and thus $f_{\texttt{greatest}}(n) = 6n + 2$. In the best-case scenario, the comparison succeeds in every step of the recursion. It occurs when no element in the list is greater than 0 such that the `car` procedure will never be used in recursive calls. In that case, $T(\texttt{(iter el l)}) = 5n + 1$, and thus $f_{\texttt{greatest}}(n) = 5n + 2$.

This clearly illustrates that we have three kinds of analyses to measure the time efficiency of an algorithm in terms of the number of computational steps performed by the algorithm:

Worst-case Analysis In a worst-case analysis, we try to estimate the number of execution steps performed by an algorithm by selecting the worst possible choice of every option offered by the algorithm. This of course, depends on the input provided to the algorithm. A worst-case analysis assumes the worst possible imaginable input. It means that every `if` test or `cond` in the algorithm is scrutinized and the worst possible branch (in terms of number of computational steps to be executed) is selected. For the majority of the algorithms presented in this text, a worst-case analysis will be presented. Given a worst-case algorithm performance characteristic $f(n)$, we can say that the algorithm will never perform more poorly than $f(n)$ for inputs of size n.

Best-case Analysis A best-case analysis is an analysis of the algorithm in which we select the most optimistic branch in every possible `if` test and `cond` the algorithm has to execute. This is the branch with the least number of potential computational steps to be taken. Clearly, *in general* this is not very useful an analysis. Given an algorithm A for which the best-case performance characteristic is $f(n) = 1$ but for which the characteristic gives $f(n) = 2^n$ for *all other inputs* except for the best case, then we clearly have a very poor algorithm even though it is very fast for *some* case (i.e. the best case). Therefore, a best-case analysis is not a very useful analysis. Nevertheless, as we will see, some algorithms (notably sorting algorithms) do have interesting best-case analyses. If the chances on the best-case input are reasonably high, then one might consider selecting the algorithm anyhow.

Average-case Analysis One might ask the question "Ok, say we have a best-case analysis and a worst-case analysis. What would be the algorithm's performance characteristic for the *average* input?". Unfortunately, answering this question is not an easy task. In order to perform an average case analysis, one needs to find out what kinds of inputs are likely to occur and what kinds of inputs are very unlikely to occur. This usually requires a good deal of knowledge on probability theory and the resulting mathematical analysis is often not even worth the effort as the outcome is usually not fundamentally different from a worst-case analysis. However, in some occasions, the worst-case is so rare that an average case analysis is more representative.

Given these considerations, most of the algorithms presented in this text will be analyzed using a worst-case analysis. We present an average-case analysis for a few algorithms for which the probabilistic distribution of the input is easy to determine. We also present a few best-case analyses when the outcome of a best-case analysis is fundamentally different from the worst-case analysis *and* when the input for that best-case execution of the algorithm is likely to occur. But again, most of our time will be devoted to worst-case analyses.

1.5.2 The Big Oh Notation

Looking back at the analyses of $T($`(iter el l)`$)$ presented in the previous session, we can make the following observations:

- We are counting the number of basic computational steps and *not* the execution time of those steps. But in a real Scheme implementation it might as well be the case that the execution of the $<$ predicate requires much more time than the execution of the `null?` predicate. So, maybe "6" would be a better estimate than "5", at least for some implementations of Scheme. Hence, hiding both the execution time for $<$ and the time needed to execution `null?` in the factor "5" is not very precise.

- Suppose that we replace the `null?` test in the algorithm by an (`eq?` '()) test and suppose that our Scheme implementation runs the (`eq?` '()) test faster than the `null?` test. If we would do our analysis of the new version of the algorithm (i.e. the one using (`eq?` '())) we would get a factor $7n$ instead of $6n$ in our final formula (since evaluating '() requires 1 computational step as well) while the actual execution time would be faster than the version of the algorithm that gave us $6n$. Again, this argument shows us that the numbers "6" or "7" are not very meaningful.

- The same thing can be said about a number of constant steps. Suppose that the first call of `iter` would be performed using the car of the list as the first maximum (instead of `0`). In that case, our worst-cast analysis would give $6n + 3$ instead of $6n + 2$. Is this additional constant factor "1" really relevant in the way we look at this algorithm? Suppose that

yet another variant of the algorithm would call two procedures before starting the iteration (e.g. a call to `display` followed by a call of `car` as before). Now our analysis would give $6n + 4$. Again we can ask ourselves the question whether the difference between $6n + 2$, $6n + 3$ and $6n + 4$ is big enough to justify this kind of precise reasoning. Especially when n gets larger and larger, we see that the role of the constant factor (i.e. 2, 3 and 4) gets smaller and smaller. When applying the algorithm to lists containing thousands of elements, the role of the constant gets neglectable.

Given these arguments, it seems like the only reasonable thing to say is that the performance characteristic of our algorithm is "something of the form $f(n) = an + b$ where a and b are relatively small numbers that do not depend on n (i.e. they are constant)". It seems that a and b are not really meaningful: b is not relevant for non-small values of n and a's exact value depends on so many technical factors (i.e. factors that depend on a particular Scheme implementation) that its precise determination is both imposible and useless. Therefore, the only thing we can say is that the execution time of our algorithm is linearly dependent on the size of its input list. The longer the list, the longer the algorithm runs in a linear sense: lists that are k times as long will require k times as much time. What we really want to grasp is *how* $f(n)$ grows as n gets larger.

Perhaps it is more instructive to have a look at some numbers. The numbers in the table of figure 1.5 show the different growth rates for different types of $f(n)$ *for large* n. In order to see the impact of the different types of functions (as opposed to small difference in constants like a and b *within* e.g. the linear case), we advise the reader to compare n^3 with $n.log(n)$ for $n = 1024$. Even though in practice, the input for algorithms is typically much bigger than 1024 (e.g. many people have an iTunes play list that contains 5000 songs), the case $n = 1024$ already reveals that using an $n.log(n)$ algorithm really pays off w.r.t. using an n^3 or even an n^2 algorithm. From the table it clearly shows that a version of iTunes in which the "sort" is implemented by an $n.log(n)$ algorithm is much faster than a version in which it is implemented by an n^2 algorithm!

In what follows, we present a number of mathematical tools that allow us to express this kind of coarse grained reasoning. They allow us to express that the functions $f_1(n) = n$, $f_2(n) = 2n + 5$ and $f_3(n) = 15n - 44$ are all "the same". We need such tools because we want to express that these functions are *not* the same as $g(n) = n^2$. Indeed, for large inputs (i.e. large n) it is not hard to see that n^2 is *considerably* larger than any combination that looks like $an + b$. In order to fully grasp this, we invite the reader to have a look at figure 1.4 which displays the relative growth of four functions. Two of them are linear (i.e. of the form $an+b$) and the other two are quadratic (i.e. of the form an^2+bn+c). We clearly see that for large n (notice that the figure merely shows the growth untill $n = 25$, a fairly small input size) the quadratic functions grow *much* faster than the linear ones (which hardly differ from the x-axis). The point of our reasoning is that n^2 grows *much* faster than n or $5n$. Hence, n and $5n$ are "the same" while $5n$ and n^2 are not.

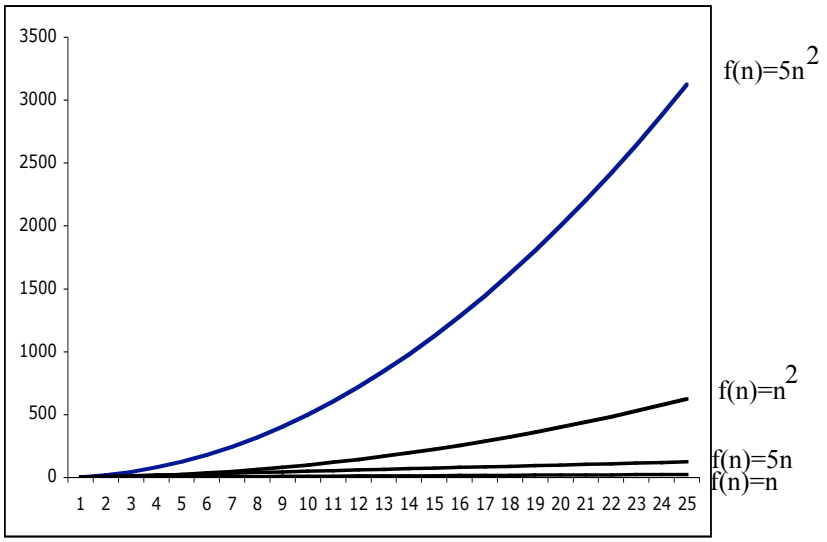

Figure 1.4: Comparing the growth of function

Let us now have study the mathematical tools that allow us to express this notion of "sameness". The following sections will present three different views on how to express that two functions are "roughly the same" or, conversely, "considerable different". All three of them have in common that they make an *asymptotic comparison* of the functions; i.e., they compare the functions "for large n" thereby omitting the smaller n. The reason for this is as follows. Suppose we have an algorithm the performance characteristic of which is $f(n) = 10n$ and suppose we have another one for which the characteristic is $g(n) = n^2$. Following the above reasoning, we agree that the g characteristic is significantly worse than the f characteristic. Applying the algorithms to inputs of huge sizes (e.g. $n = 10^7$) makes this difference painfully tangible. Nevertheless, it is easy to see that $n^2 \leq 10n$ for all $n \leq 10$. In other words, g is better than f for some small number of uninteresting n. Therefore, it would be a mathematical error to say that g *is* bigger than f because this is only a true statement if $g(n) > f(n)$ for *all* n. We therefore only compare g and f "only for large n", i.e. for $n \geq n_0$ where n_0 is a certain constant whose exact value is not really relevant.

Big Theta

We now present the first of a set of mathematical tools that allow us to say that a function $f_1(n)$ is behaving "similar" to a function $f_2(n)$. We will do so by looking at a set of functions $\Theta(f_1(n))$ which is the set of all functions that are roughly the same as $f_1(n)$. Once we have defined this set, we can then say that

n	$log(n)$	$\sqrt{n}$	n	$n.log(n)$	n^2	n^3	2^n
2	1	2	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	3	8	24	64	512	256
16	4	4	16	64	256	4096	65536
32	5	6	32	160	1024	32768	4294967296
64	6	8	64	384	4096	262144	1.85×10^{19}
128	7	12	128	896	16384	2097152	3.40×10^{38}
256	8	16	256	2048	65536	16777216	1.16×10^{77}
512	9	23	512	4608	262144	134217728	1.34×10^{154}
1024	10	32	1024	10240	1048576	1073741824	1.79×10^{308}

Figure 1.5: Growth of Functions of n

$f_2(n)$ is similar to $f_1(n)$ by writing $f_2(n) \in \Theta(f_1(n))$.

Here is the definition:

$$\Theta(g(n)) = \{f | \exists c_1, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\}$$

In order to understand this definition formally, we refer to figure 1.6 which shows an imaginary function $g(n)$ and an imaginary function $f(n)$ which is in $\Theta(g(n))$. The idea of n_0 is to ignore "initial" irregularities for small n in the behaviour of $f(n)$ and $g(n)$ because we are only interested how $f(n)$ and $g(n)$ relate to each other in general, for big values of n.

The above definition formally defines what it means for an f to be "roughly the same as" g. Figure 1.6 shows the situation graphically. As we can see, the idea of Big Theta is to determine two constants c_1 and c_2 such that g lies between "a c_1-fold of f" and "a c_2-fold of f". In other words, apart from the factors c_1 and c_2, we can say that $f(n)$ is "roughly the same as" $g(n)$. More precisely, we say that $g(n)$ *is an asymptotically tight bound to* $f(n)$.

Big Theta is a very convenient mathematical tool to express that a function f is roughly the same as g. Unfortunately it is not always easy to prove this property. An even bigger problem is that Big Theta is sometimes too precise a mathematical tool. Remember from section 1.5.1 that we are often concerned with a worst-case analysis and this does not always fit Big Theta since our algorithm *might* have a best-case analysis performance characteristic that is of another Bit Theta. For instance, in section 5.3.2, we will present a sorting algorithm called insertion sort. Given a vector of length n, it will turn out to be the case that the sorting algorithm is $\Theta(n^2)$ if we perform a worst-case analysis (this will be the situation where the vector is completely unsorted or even sorted in reverse order). However, performing a best-case analysis on the same algorithm will reveal that the algorithm is $\Theta(n)$ (the best case will be the situation where the vector is already sorted before applying the sorting algorithm). Hence, it will be technically incorrect to say that the insertion sort algorithm "is roughly n^2" or "is roughly n" for *all* inputs if we use the Θ

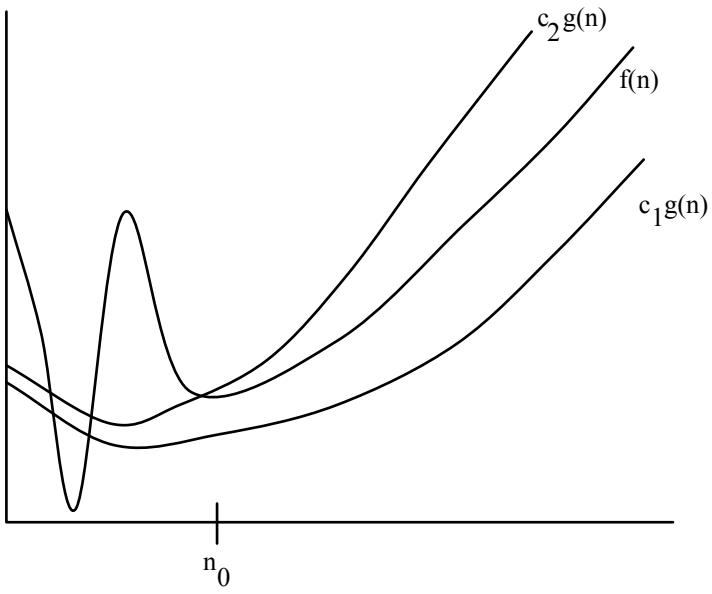

Figure 1.6: $f(n) \in \Theta(g(n))$

tool. The tool seems to be too precise if we want to express the performance characteristic irrespective of the properties (i.e. worst-case or best-case) of the input.

Big Oh

The problem with Θ is that it requires us to approximate functions both with an upper bound as well as a lower bound. The Big Oh notation is less strict in that it only requires us to find upper bounds. Here is the definition:

$$O(g(n)) = \{f | \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)\}$$

Now let us reconsider the problem with the insertion sort described above. If we were to say that "insertion sort *is* an $O(n^2)$" algorithm, this would not be a mathematical error. After all, the worst-case analysis of insertion sort tells us that it "is an n^2 algorithm" while the best-case analysis tell us that it "is an n algorithm". Therefore we are *not* allowed to say that insertion sort *is* $\Theta(n^2)$ since $n \notin \Theta(n^2)$. But we *are* allowed to say that "insertion sort is an $O(n^2)$ algorithm" since both $n^2 \in O(n^2)$ and $n \in O(n^2)$. Figure 1.7 shows what it means for a function $f(n)$ to be $O(g(n))$. Notice that we are no longer speaking about the functions being "roughly the same". O merely gives an *asymptotic upper bound* and is therefore *much* less precise than Θ. For example, it would

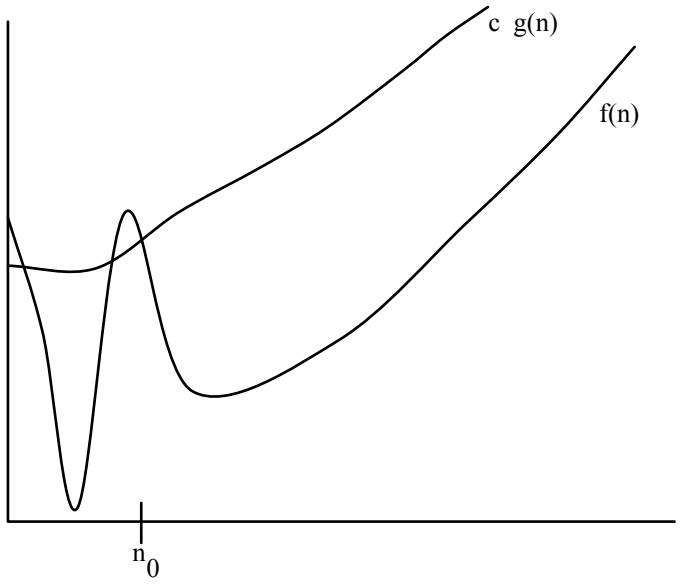

Figure 1.7: $f(n) \in O(g(n))$

be perfectly valid to say that $n \in O(2^n)$ since 2^n is clearly an upper bound for n. Hence, whenever we want to use $O(g)$ to estimate the performance of an algorithm we will try to be as precise as possible by selecting a g as small as possible. Informally, $f(n) \in O(g(n))$ means that the worst-case for f will be no worse than g.

Big Omega

Instead of looking for asymptotic *upper* bounds for functions, we might also look for asymptotic *lower* bounds. This is the role of Big Omega. It is defined as follows:

$$\Omega(g(n)) = \{f | \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

Figure 1.8 shows us what it means for a function $f(n)$ to be in $\Omega(g(n))$. Informally it means that f will be at least as big as g for large n. Since Big Omega is a lower bound, we mainly use it to speak about best-case analyses for algorithms. Just like O, Ω is not very precise a tool. For example, it is mathematically correct to say that $2^n \in \Omega(n)$ since n is indeed a lower bound for the exponential function. Needless to say, this is pretty meaningless a statement. Remember that we said that a best-case analysis of the insertion sort algorithm will reveal that the algorithm performs "like n" in the best case (i.e. when the data is already sorted). More formally we can now say that insertion sort is an

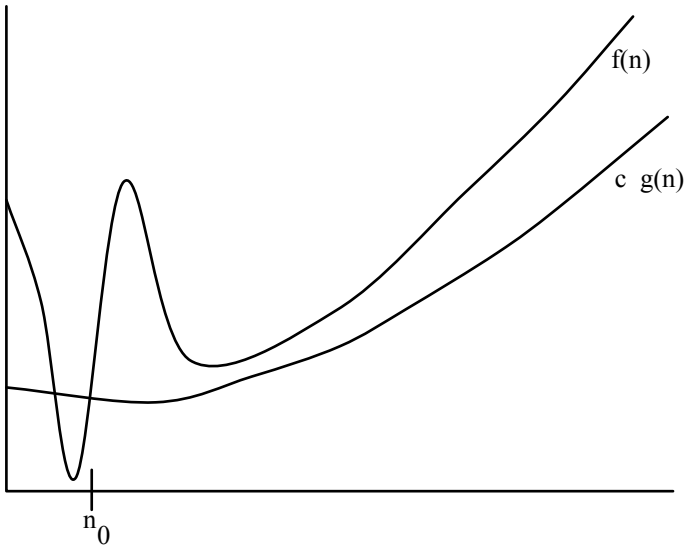

Figure 1.8: $f(n) \in \Omega(g(n))$

$\Omega(n)$ algorithm: given any input, it will always take more execution time than n. Stated otherwise, $f(n) \in \Omega(g(n))$ expresses the fact that the best case for f will not be better than g.

Theorem: Relations between Θ, Ω and O

For the sake of completeness, we relate the three notations with each other. This is done formally by the following theorem:

For any two functions f and g, $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$

This is actually not a surprising result: the theorem merely states that a function f is *tightly* bound by g if and only if f has g both as a *lower* bound and as an *upper* bound.

Remarks

From the previous discussion, we conclude that Θ is the most precise instrument for measuring the performance of algorithms. Nevertheless, since Θ is not always easy to determine and pathological input cases (e.g. a few specific input values) can cause a good candidate for Θ not to hold in general. Therefore, we will usually opt for O in which we will try to be as precise as possible by taking the smallest f such that an estimate $O(f(n))$ holds for our algorithm. After all, saying that our algorithm is in $O(n)$ is more precise than saying that it is in

$O(n^2)$. Moreover, O usually fits a worst-case analysis better and this is what we are usually interested in. O gives us a rough classification of algorithms. However, blindly using O to compare algorithms is not always without danger.

- One of the practical consequences of using the O notation is that constant coefficients do not matter. Indeed, in the definition of O, a constant c has to be found to find an upper bound for $f(n)$. This means that, whenever we have $f'(n)$ that is a constant factor bigger than $f(n)$ (in other words: $f'(n) = a.f(n)$) then it suffices to take $c' = c.a$ in order to have a constant c' that is an upper bound for $f'(n)$. Hence $f(n)$ and $a.f(n)$ belong to the same Big Oh. However, sometimes caution is required. Clearly, $10^{10}n$ is much bigger than n^2 for most of the practically occurring n. Hence, an algorithm with quadratic time might *in some cases* be more beneficial than a linear algorithm, even though $O(n)$ is theoretically better than $O(n^2)$. Nevertheless, as computers get faster, the constant coefficient gets less relevant (remember that we count the *amount* of steps and not the absolute time) and the quadratic algorithm gets less attractive.

- A second word of caution is needed when comparing two algorithms with performance characteristics that are of the *same* order of growth. As we will see in section 3.4.2, using some clever tricks here and there in one's source code can make the code twice as fast. Even though both algorithms are of the same order (e.g. $O(n/2)$ and $O(n)$), one algorithm can be substantially faster than another one. Comparing the performance characteristic of two different algorithms is much more instructive when their performance characteristics are of different orders of growth than when they are of the same order of growth.

Simplification Rules

The following rules make it much simpler to come up with an expression for $O(f_{\texttt{A}}(n))$ for a given Scheme procedure `A`. Their proof follows directly from the definition of O.

1. Constants in front of terms (i.e. coefficients) play no role: $O(an^k) = O(n^k)$. This was already shown above.

2. A corollary of this is that $O(c) = O(1)$ for all constants c. An algorithm the performance characteristic of which is $O(1)$ executes in constant time, i.e. it is equally fast for all possible inputs.

3. Another corollary is that the basis for logarithms is of no importance. Since $log(a^b) = b.log(a)$, we can prove that $log_a(x) = \frac{log(x)}{log(a)}$. Hence, $O(log_a(x)) = O(log(x))$ since coefficients are not important.

4. Only dominant terms are important: If $f(n) = t_1 + t_2$ where the order of t_1 is higher that the order of t_2 then $O(f(n)) = O(max(t_1, t_2)) = O(t_1)$.

At this point we refere back to the table in figure 1.5 which shows us the orders that occur most often. We conclude that for O:

$$1 < log(n) < \sqrt{(}n) < n < n.log(n) < n^{k-1} < n^k < 2^n < n!$$

Finding O for Scheme procedures

Now that we have developed the mathematical machinery necessary to reason about the performance $O(f_A(n))$ of an algorithm A, it remains to be discussed, given a concrete Scheme procedure `A`, how we can — based on the source text of `A` — come up with the function $f_A(n)$. We will do this in two phases. First, we consider Scheme's primitive computational elements one by one and estimate how much computational effort they require. Second we will see how to combine this knowledge in order to determine the performance characteristic of a compound Scheme procedure, the body of which is composed out of these expressions. Hence, the idea is to distill the performance characteristic of a Scheme procedure by combining the performance characteristics of the expressions that make up the body of the procedure.

1.5.3 Analyzing Scheme Expressions

Let us systematically consider Scheme's expression types in order to come up with a systematic methodology for finding $f_A(n)$ given the source text of the body of the Scheme procedure `A`. The following list enumerates all Scheme's expression types and estimates their performance characteristic in terms of the size n of the input of the procedure to which they belong:

- $O(f_{\texttt{(define v E)}}(n)) = O(f_{\texttt{E}}(n))$. This is because a `define` expression contains a subexpression `E`. The number of steps needed for executing the `define` expression itself is $O(1)$. However its subexpression `E` can be computationally intensive. Therefore, the order of growth of the entire expression is $O(f_{\texttt{E}}(n))$ where $f_{\texttt{E}}(n)$ is `E`'s performance characteristic. For example, the expression (`define x 3`) will be $O(1)$. However, the expression (`define x (fac n)`) requires $O(n)$ computational steps because $f_{\texttt{fac}} \in O(n)$. In other words, the efficiency of the compound `define` expression depends on the efficiency of its subexpression.

- $O(f_{\texttt{(set! v E)}}(n)) = O(1) + O(f_{\texttt{E}}(n)) = O(f_{\texttt{E}}(n))$. The reasoning for `set!` is exactly identical to that for `define`.

- $O(f_{\texttt{(set-car! p E)}}(n)) = O(f_{\texttt{(set-cdr! p E)}}(n))$ which is the same as $O(f_{\texttt{p}}(n) + f_{\texttt{E}}(n))$ and thus $O(max(f_{\texttt{p}}(n), f_{\texttt{E}}(n)))$. In the case of `set-cdr!` and `set-car!`, there are two subexpressions: one to calculate the pair `p` in which to set the car or cdr, and another one to calculate its new value designated by the expression `E`. Hence, we have to add the performance characteristics of both subexpressions. Furthermore, we only have to take the greatest term of this sum.

- $O(f_{\texttt{(lambda lst body)}}(n)) = O(1)$. Evaluating an anonymous `lambda` form only takes one computational step in order to *create* the procedure object.

- $O(f_{\texttt{(if c t e)}}(n)) = O(max\{f_{\texttt{c}}(n), f_{\texttt{t}}(n), f_{\texttt{e}}(n)\})$. Evaluating an `if` expression causes the condition `c` to be evaluated, followed by the evaluation of `t` or `e`. Hence, the performance characteristic is $f_{\texttt{c}}(n) + f_{\texttt{t}}(n)$ or $f_{\texttt{c}}(n) + f_{\texttt{e}}(n)$. Hence, we have to take the maximum of these three performance characteristics.

- $O(f_{\texttt{(let ((v1 e1) (v2 e2) ... (vn en) b1 b2 .. bk)}}(n))$ $= O(max_i(f_{\texttt{ei}}(n)) + max_j(f_{\texttt{bj}}(n))) = O(max_{i,j}\{f_{\texttt{ei}}(n), f_{\texttt{bj}}(n)\})$. In the case of a `let` expression that consists of several subexpressions, we have to add the performance characteristics of *all* the subexpressions. However, we know that only the dominant term is relevant in such an addition. Therefore, taking the maximum of the performance characteristics of the subexpression suffices.

- $O(f_{\texttt{(cond ((c1 a1) ... (cn an)))}}(n)) = O(max_i\{f_{\texttt{ci}}(n), f_{\texttt{ai}}(n)\})$. For `cond`, the same reasoning applies.

- $O(f_{\texttt{(begin e1 ... en)}}(n)) = O(f_{\texttt{e1}}(n)+...+f_{\texttt{en}}(n)) = O(max_i(f_{\texttt{ei}}(n)))$. Similarly, the performance characteristic of a `begin` expression is the dominant term of the sum of the performance characteristics of its subexpressions.

- $O(f_{\texttt{(prim a1 ... an)}}(n)) = O(1) + O(f_{\texttt{a1}}(n)) + ... + O(f_{\texttt{an}}(n))$ for $\texttt{prim} \in \{\texttt{+}, \texttt{-}, \texttt{and}, \texttt{or}, \texttt{eq?}, \texttt{eqv?}, \texttt{vector-ref}, \texttt{vector-set}, \texttt{car}, \texttt{cdr}, ...\}$. Calls of most *primitive* procedures such as $+$, $-$, etc result in a performance characteristic that is in $O(1)$ since they only require one computational step. On top of this, the arguments have to be evaluated which gives rise to performance characteristics $f_{\texttt{ai}}(n)$. Again, the dominant term in the sum of all these subexpressions survives. However, caution is required when using primitive procedures that work with Scheme's built-in list data structure. Scheme features quite a lot of primitives that are used to convert data values into lists and vice versa. Examples are `list->vector`, `vector->list`, `string->list` and `list->string`. Needless to say, these do not exhibit $O(1)$ behaviour but depend on the size of the vector, string or list that is passed as an argument. They exhibit an $O(n)$ behaviour where n is the length of the input list or the input vector.

- $O(f_{\texttt{(f a1 ... an)}}(n)) = O(f_{\texttt{f}}(n) + f_{\texttt{a1}}(n) + ... + f_{\texttt{an}}(n))$. Determining the performance characteristic for a *non-primitive* procedure call has to be done by adding (i.e. selecting the dominant term) of the performance characteristics of all subexpressions, augmented with the performance characteristic of the procedure `f` that is actually executed. The

following section explains how to establish this characteristic for arbitrary Scheme procedures.

An important Scheme expression that was omitted in this list is the so called *named* `let`. In the following example, a named `let` is used to implement an iterative version of the good old factorial procedure:

```
(define (fac n)
  (let fac-loop
    ((result 1)
     (factor n))
    (if (= factor 0)
      result
      (fac-loop (* result factor) (- factor 1)))))
```

We will use the named `let` construct quite frequently in this text. What can we say about the performance characteristic of the named `let` construct given the performance characteristics of its subexpressions? Unfortunately, the situation is not simple. Fortunately there is a way to get rid of the named `let` construct when establishing performance characteristics. The idea is to replace the named `let` by two subexpressions. The first subexpression defines a local procedure whose purpose it is to implement the loop expressed with the named `let`. The second expression launches the loop by calling that local procedure. For the `fac` example given above, this transformation results in the following code:

```
(define (fac n)
  (define (fac-loop result factor)
    (if (= factor 0)
      result
      (fac-loop (* result factor) (- factor 1))))
  (fac-loop 1 n))
```

This code has the benefit that it is a (recursive) procedure that does not use the named `let` construct. Establishing its performance characteristic is easy as soon as we know how to establish the performance characteristic of a procedure. This is the topic of the following section. We conclude that it is always possible to transform a named `let` expression into two subexpressions; one to define a local procedure and another one to call that procedure.

1.5.4 Analysing Scheme Procedures

Now that we have a precise definition of the performance characteristic of Scheme's primitive computational building blocks, the final topic of this chapter is to present a technique that allows us to distill the performance characteristic of a scheme procedure `P` based on the expression `B` that forms the body of `P`. We distinguish between recursive procedures and non-recursive procedures.

Non-recursive Procedures If `P` is a non-recursive procedure (i.e. a procedure that does not call itself directly or indirectly) then the situation is simple: we simply analyse the body expression `B` using the rules presented in section 1.5.3. In case the body has multiple subexpressions then the performance characteristic is nothing but the sum of the performance characteristics of those subexpressions. For example, consider the following Scheme procedure.

```
(define (weird vector)
  (define len (vector-length vector))
  (if (odd? len)
    (a-linear-function len)
    (a-quadratic-function len)))
```

Applying what we know gives us a performance characteristic $f_{\texttt{weird}}(n) \in O(n^2)$ where n is the length of the input vector. The reasoning is as follows: the characteristic for the procedure is the sum of the characteristics of the subexpressions. However, we know from the simplification rules that only the dominant term is of importance in such a sum. Therefore, we only have to consider the maximum of the performance characteristics of the subexpressions. There are two such subexpressions: the `define` expression and the `if` expression. The `define` subexpression gives $O(1)$ since its subexpression consists of a call to a primitive procedure of $O(1)$ (i.e. `vector-length`). The performance characteristic for the `if` expression is the maximum of the performance characteristics of its subexpressions. Assuming that $f_{\texttt{a-quadratic-function}}(n) \in O(n^2)$ and $f_{\texttt{a-linear-function}}(n) \in O(n)$, we thus get $f_{\texttt{weird}}(n) \in O(n^2)$.

Recursive Procedures If `P` is a recursive procedure because its body `B` calls `P` again[2], then the situation is much more complex. Let us consider our good old `fac` again:

```
(define (fac n)
  (if (= n 0)
    1
    (* n (fac (- n 1)))))
```

When applying the above rules, one soon comes up with the fact that $f_{\texttt{fac}}(n) = 1 + f_{\texttt{fac}}(n-1)$. In other words, in order to come up with the performance characteristic for a recursive procedure, we need the performance characteristic of the procedure in its turn. Solving such mathematical "recursive equations" can be quite simple. However, in many cases solving the equation is all but trivial. A precise mathematical derivation of performance characteristics of recursive functions is therefor outside the

[2]Remember that there is a difference between a recursive *procedure* and a recursive *process*. A recursive procedure is a procedure that syntactically calls itself. This can give rise to both a recursive as well as to an iterative process.

scope of this text. To avoid such complex analysis, we will take a rather intuitive approach in this text by applying the following rule of thumb:

Rule of Thumb: If we have a recursive procedure that takes an argument the input size of which depends on n, then first determine the performance characteristic $O(b(n))$ for the body of the procedure without taking the recursive procedure calls into account. Then determine an estimate $r(n)$ for the number of recursive calls (depending on n) that will be made. The performance characteristic for the entire recursive procedure will be $O(b(n).r(n))$.

Examples

Example 1: For our good old `fac` shown above, it should come as no surprise that the characteristic of its body is $O(1)$ since that body merely consists of primitive procedure applications. The number of times `fac` will be called is determined exactly by the size of its input n since every call with argument `n` (except for the last one) gives rise to exactly one call with argument `n-1`. In other words $b(n) \in O(1)$ and $r(n) \in O(n)$. Hence $f_{\texttt{fac}} \in O(1.n) = O(n)$.

Example 2: As a second example, let us have a look at procedures for computing Fibonacci numbers. The following code excerpt shows two Scheme procedures for computing Fibonacci numbers. `fib1` is the canonical recursive algorithm. `fib2` is an iterative algorithm.

```
(define (fib1 n)
  (if (< n 2)
    1
    (+ (fib1 (- n 1)) (fib1 (- n 2)))))

(define (fib2 n)
  (define (iter n a b)
    (if (= n 0)
      b
      (iter (- n 1) b (+ a b))))
  (iter n 0 1))
```

Let us now try to come up with the performance characteristics $f_{\texttt{fib1}}(n)$ and $f_{\texttt{fib2}}(n)$ for `fib1` and `fib2` respectively.

- **The `fib1` case:** `fib1` is a recursive procedure. Following our rule of thumb, we have to find $b(n)$ and $r(n)$. Clearly, $b(n) = 1$ since (apart from the recursive calls), the body of `fib1` solely consist of applying primitive procedures such as $+$, $<$ and $-$. In order to come up with $r(n)$, we observe that every call to `fib1` gives rise to two recursive procedure calls. Notice that the number of recursive calls generated by `(fib1 (- n 2))` will be smaller than `(fib1 (- n 1))`. Hence, it is fair to say that the number of recursive calls generated by `(+ (fib1 (- n 1)) (fib1 (- n 2)))` is less than the number of recursive calls generated by `(+ (fib1 (- n 1))`

`(fib1 (- n 1))`. If we were to replace the body of `fib1` by the latter expression, we would have a recursive procedure in which every procedure call produces *exactly* two recursive calls. This means that $r_{\texttt{fib1}}(n) \leq 2^n$. As a consequence, $f_{\texttt{fib1}} \in O(2^n)$.

We can ask ourselves whether we can be more precise than this. As a matter of fact we can, but this requires more complex mathematical reasoning which is beyond the scope of this text. It is possible to prove that the number of recursive calls made by `fib1` is actually a bit smaller than 2^n, but not much. It can be shown that the number of calls is always proportional to ϕ^n where $\phi = 1.61$ which is the famous golden ratio. In other words, we can prove that $r_{\texttt{fib1}}(n) = \phi^n$. Hence, we can say that $f_{\texttt{fib1}} \in \Theta(\phi^n)$.

- **The `fib2` case:** `fib2` calls `iter` and `iter` is a recursive procedure as well (even though it generates an iterative process). Again, we have $b(n) = 1$ since the body of `iter` only consists of applications of primitive Scheme procedures. It should be clear that $r(n) = n$ since every call to `iter` either stops or gives rise to one more recursive call with an argument that is exactly 1 smaller than the argument of the original call. In other words, $f_{\texttt{fib2}} \in O(1.n) = O(n)$.

Multiple Parameters

Until now, we have only considered performance characteristics for Scheme procedures that take one argument the size of which is given by n. But what does it mean to establish a performance characteristic for algorithms that take multiple arguments? Consider for example the following algorithms that calculate the sum and product of two Scheme numbers `n` and `m`.

```
(define (sum n m)
  (if (= n 0)
    m
    (+ 1 (sum (- n 1) m))))

(define (times n m)
  (if (= m 1)
    n
    (sum n (times n (- m 1)))))
```

What can we say about $f_{\texttt{sum}}$ and $f_{\texttt{product}}$? Since these procedures take two arguments n and m, the performance characteristics will be mathematical functions that depend on two arguments as well. Hence, we are looking for two functions $f_{\texttt{sum}}(n, m)$ and $f_{\texttt{times}}(n, m)$.

Clearly, `sum` is a procedure whose body is in $O(1)$ provided that we ignore the recursive call. Hence, $b_{\texttt{sum}}(n, m) = 1$. Since it breaks down its first parameter `n` until it reaches zero, it obviously executes the recursion n times. Hence we can say that $r_{\texttt{sum}}(n, m) = n$. By applying our rule of thumb, we conclude that

$f_{\texttt{sum}}(n, m) \in O(n)$. Looking at `times`, we observe that its body consists of a call to `sum`. Hence, $b_{\texttt{times}}(n, m) = n$. Furthermore, `times` is called recursively m times, resulting in $r_{\texttt{times}}(n, m) = m$. Again, our rule of thumb yields the overal result for `times` which is $f_{\texttt{times}}(n, m) \in O(n.m)$.

1.5.5 Performance Measure 2: Memory Consumption

Most of the algorithms presented in this text operate on data structures that already reside in our computer's central memory. For example, a sorting algorithm sorts the entries of an iTunes playlist. The playlist is implemented as a data structure (e.g. a list or a vector) residing in main memory. Furthermore, most of the algorithms do not require *additional* memory on top of the memory that is already occupied by the data structure on which the algorithm operates. Algorithms that meet this property are said to be *in-place*. We only discuss very few algorithms which are not in-place. For those algorithms that are not in-place, we will estimate the amount of memory that is needed for the execution of the algorithm. Luckily, we can use the same mathematical tools (i.e. Ω, O and Θ) to express the amount of memory. For example, we might say that a certain algorithm consumes an mount of memory that is $O(n)$. This means that the algorithm itself will (while it is running) reserve a number of memory cells on top of the memory cells that are occupied by the input. The amount of memory is directly proportional to n.

Examples

On first sight, the implementation of the Fibonacci algorithm `fib1` shown above does not consume any memory apart from the argument itself (i.e. it is in-place). However, since the procedure generates a recursive process, this means that every recursive call has to remember what to do after returning from that recursive call. From the procedure body, we see that a call to `fib1` with argument `n` will call `fib1` with argument `n-1` after it has called `fib1` with argument `n-2` (or the other way around, depending on the order in which Scheme evaluates the arguments of +). This means that the recursion depth of `fib1` will be n. Given the fact that `fib1` generates a true recursive process, our Scheme interpreter will have to remember n times what to do after returning from the chain of recursive calls. In other words, `fib1` requires $\Theta(n)$ memory to execute. Hence, `fib1` is *not* an in-place algorithm.

Now let's consider the implementation of `fib2`. In contrast to `fib1`, `fib2` generates an iterative process. Even though every call to `fib2` generates a call to `fib2` in its turn (apart from the final call), the Scheme evaluator does not have to remember anything about what to do when returning from the latter call. The only thing that remains to be done after returning from the latter call, is to return from the former call. This means that we can safely return from the very last call to the very first call while ignoring all intermediate calls. No additional memory is needed to remember partial results from intermediate

calls. Hence, `fib2` requires $\Theta(1)$ memory which means that it is a true in-place algorithm.

Chapter 2

Strings and Pattern Matching

As explained in section, 1.1, strings form one of Scheme's built-in compound data types. In this chapter we discuss the string data structure in more detail. We present an overview of the most important procedures that have been built into Scheme for manipulating strings.

The bulk of our attention is devoted to the study of algorithms designed to find some string (called a pattern) in another string (called a text) that is typically much longer than the first one. Finding patterns in texts is also known as *pattern matching*. Several dozens of pattern matching algorithms have been invented. Most of them exhibit different performance characteristics depending on the properties of the pattern and the text. Studying a representative selection of these algorithms is the central theme of this chapter. At the end of the chapter, we present an algorithm — called QuickSearch — that doesn't necessarily have the best worst-case performance characteristic but that seems to beat all other algorithms in practice. Let us first start by establishing some terminology that allows us to talk about strings.

2.1 Strings in Scheme

A string is a compound data element of the data type `string` which is built into Scheme. Strings are finite sequences of characters. The simplest way to get hold of a string is to use the literate constructor "..." as explained in section 1.1. For example, "`Madam, I'm Adam`" is a string consisting of 14 characters including two whitespace characters. However, "..." is not the only way to create new strings. Procedural constructors `make-string` and `string` can be used as well. For example, given the character `#\a`, then (`make-string 10 #\a`) creates a string that consists of 10 a's. Given a list of characters, the `string` procedure can be used to create a string by juxtaposing the characters contained in the list. E.g., (`string #\S #\c #\h #\e #\m #\e`) creates a new string that is

identical to the string obtained by evaluating the literate expression "Scheme".

Having constructed strings using one of these constructors, Scheme's primitive procedures `string-length` and `string-ref` act as accessors for strings. `string-length` returns the *length* of a string. The length is defined as the number of characters contained by the string. For example, (`string-length` "hello") returns 5. Using a slightly more mathematical notation, we denote the length of a string s by $|s|$. For example $|\texttt{''hello''}| = 5$. The *empty string* is a string that has no characters. It is denoted by "" and its length is — by definition — zero. `string-ref` is used to access the individual characters of a string given an index i which denotes the location of the character in the string. i may vary from 0 to $|s| - 1$. E.g., given the string (`define my-string` "Scheme"), then the expression (`string-ref my-string 3`) evaluates to #\e. Strings are said to be *immutable data values*. This means that it is impossible to change the composition of a string after construction[1].

After having described the constructors and accessors for strings, let us now have a look at the operations Scheme provides on the `string` data type. Below we merely list a few operations that are extremely frequently used. For a complete list, we refer to the R6RS.

Conversion Operations: Two primitive procedures can be used to convert strings to lists of characters and the other way around. E.g., (`string->list` "Scheme") results in the Scheme list (#\S #\c #\h #\e #\m #\e). Conversely, (`list->string` (`list` #\a #\S #\t #\r #\i #\n #\g)) yields the string "aString". Both operations are in $O(n)$ where n is the length of the list or the length of the string at hand. This is because the Scheme evaluator has to process the entire sequence of characters in both cases.

Comparison Operations: The following procedures can be used to compare strings with one another. String comparison can be done in two ways. *Case sensitive comparison* makes a distinction between upper case and lower case characters. *Case insensitive comparison* does not distinguish between upper case and lower case characters. Whether or not a comparison procedure distinguishes cases is reflected by the presence or absence of `ci` (= case insensitive) in the name of the procedure. The table in figure 2.1 shows a complete overview of Scheme's string comparison procedures.

These comparison procedures are based on the *lexicographic ordering* that is defined for strings. In general, this means that shorter strings come before longer strings that start with the same sequence of characters. For example "hello" comes before "hello `world`". Furthermore, it means that #\a comes before #\b as one would expect. The exact order for individual characters is prescribed by an extension[2] of the ASCII code — the American Standard Code for Information Interchange. This code assigns

[1] Since R6RS, Scheme's standard libraries feature so-called mutable strings as well. We refer to the R6RS for more details.

[2] The extension is called the Unicode standard. Again, we refer to the R6RS for more details.

String comparison operation	Functionality
(`string=? s1 s2`)	String equality
(`string-ci=? s1 s2`)	String equality, case insensitive
(`string<? s1 s2`)	String before
(`string>? s1 s2`)	String after
(`string<=? s1 s2`)	String before-equal
(`string>=? s1 s2`)	String after-equal
(`string-ci<? s1 s2`)	String before, case insensitive
(`string-ci>? s1 s2`)	String after, case insensitive
(`string-ci<=? s1 s2`)	String before-equal, case insensitive
(`string-ci>=? s1 s2`)	String after-equal, case insensitive

Figure 2.1: Scheme's string comparison procedures

an number between 0 and 255 to all regularly occurring characters. The Scheme procedures `char->integer` and `integer->char` can be used to convert characters to their ASCII value and the other way around. For example, (`char->integer` #\a) yields 97 and (`integer->char` 65) yields #\A. These ASCII values are used to define the lexicographic ordering: a character `c1` is "smaller" than a character `c2` if (< (`char->integer c1`) (`char->integer c2`)) where < is the usual Scheme procedure for comparing numbers.

String addition and substraction: Strings can be "added" using the `string-append` procedure. It takes an arbitrary number of argument strings and it produces a new string that is the concatenation (also known as the juxtaposition) of the argument strings. E.g., (`string-append` "peek-" "a" "-boo") yields the string "peek-a-boo". Strings can also be "substracted" using the `substring` procedure. `substring` takes three arguments: a string, a starting index and an end index. It "reads" the characters from the string, starting at the start index and ending and the end index. It returns a *new* string that consists of the corresponding characters in the input string. For example, the expression (`substring "Scheme is the greatest language!" 14 22`) yields "greatest". `substring` does not modify its argument string.

This concludes our overview of Scheme's built-in string processing procedures. This overview is far from complete and the R6RS lists a huge number of built-in string processing procedures that are worthwhile studying whenever one has to deal with strings when writing Scheme applications.

There is one important aspect of string processing that is not included in Scheme's list of built-in string processing procedures. It is known as the *pattern matching problem* and it is a frequently occurring problem in computer science. The reason for not including a default Scheme procedure that solves the pattern matching problem in the list of standard Scheme procedures is that there exists no standard solution to the problem. Different solutions each have their

advantages and disadvantages. A study and comparison of these solutions to the pattern matching problem is the topic of the rest of this chapter.

2.2 The Pattern Matching Problem

The pattern matching problem is formulated as follows. Suppose that we have a string `t` (also known as "the text" or "the haystack") and suppose that we have another string `p` (also known as "the pattern" or "the needle"). The pattern matching problem is the question of finding a procedure (`match t p`) that is capable of computing an index `i` such that (`string=? p` (`substring t i` (+ `i` (`string-length p`)))). In other words, we are looking for the index i in `t` that corresponds to the location of `p` in `t`. It is said that `t` *matches* `p` *at position* i. The position is also known as the *offset* or *shift* of `p` in `t`.

For example, if the haystack is "`madam, I'm adam`" and the needle is "`adam`" then `match` should return 1 since "`adam`" occurs in the haystack starting at index 1 (remember that string indexes start counting from 0). This example shows that a text `t` may match a pattern `p` several times at various offsets. We speak about different *occurrences* of the pattern in the text. Clearly, if we have developed an algorithm that is capable of finding a single occurrence of the pattern in the text, then we also have an algorithm to find multiple occurrences in the text. All we have to do is apply the original algorithm multiple times.

The pattern matching problem has an obvious application in word processing programs. Anyone who has ever used such a program is acquainted with the "find" command which allows one to look for a pattern in the text file one is editing. In this application of pattern matching, the pattern usually consists of a small sequence of characters. More recently, the pattern matching problem has also found applications in bioinformatics, a new branch of computer science that uses computer science techniques in the context of biology and biotechnology. One frequently occurring problem in bio-informatics consists of finding sequences of genetic codes in long strings of DNA. This is a formulation of the pattern matching problem where the "text" consist of millions of characters whilst the "pattern" (i.e. the genetic code that is searched for) consists of a few hundreds of thousands of "characters". This example shows that, in contrast to the "find" command customary found in text processors, patterns can be quite long as well.

Before we start exploring the realm of pattern matching algorithms, we need to develop some additional terminology and notation that can be used to talk about strings in precise ways. Given a string `s`, then a string `e` is said to be a *prefix* of `s` if there exists a string `u` such that (`string=? (append e u) s`). `u` is then said to be a *suffix* of `s`. If `e` is a non-empty string (i.e. $|\texttt{e}| > 0$) and $\texttt{e} \neq \texttt{s}$ then `e` is said to be a *proper prefix* of `s`. Similarly, if `u` is a non-empty string (i.e. $|\texttt{u}| > 0$) and $\texttt{u} \neq \texttt{s}$, then `u` is said to be a *proper suffix* of `s`. In what follows, we shall mean "proper prefix" (resp. proper suffix) whenever we say "prefix" (resp. suffix). Sometimes a slightly more mathematical notation is preferred. Whenever we want to express that a string s consists of two

parts, u and v, then we write $s = u.v$. In other words, the dot notation is a mathematical shorthand for Scheme's `string-append`. In what follows, we sometimes need a prefix of a string s that is exactly k characters long. We shall indicate such prefix by $s_{0\rightarrow k-1}$. Similarly, a suffix consisting of k characters will be referred to as $s_{|s|-k\rightarrow|s|-1}$. The k'th character of the string s is simply denoted by s_k. For example, let us consider the string s of the form "Madam, I'm Adam". Then $s_{0\rightarrow 4}$ is the string "Madam". It is a prefix of length 5. Similarly, $s_{15-4\rightarrow 15-1} = s_{11\rightarrow 14} =''$ Adam$''$ is a suffix of s of length 4. $s_6 = \#\backslash$space is the 6'th character of s.

2.3 The Brute-Force Algorithm

We start our study of pattern matching algorithms by presenting the "brute-force algorithm". The name of the algorithm comes from the fact that it doesn't use any clever tricks to speed up the matching process in any way. Instead it simply considers *all* potential matches, one by one, until a match has been found. The brute-force algorithm is simple to program which is probably the reason why it is not a very efficient algorithm.

The following procedure is an implementation of the brute-force algorithm in Scheme. The procedure takes a text `t` and a pattern `p` as arguments. It either returns #f when the pattern does not occur in the text, or a number indicating the offset of the pattern in the text.

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
      (loop i-t (+ i-p 1)))
      (else
       (loop (+ i-t 1) 0))))))
```

Before explaining and analysing this procedure, let us first introduce some additional terminology and notation. We often need to refer to the length of `t` (resp. `p`). Depending on the font used, it is denoted by `n-t` (resp. `n-p`) or n_t (resp. n_p). All algorithms presented in this chapter use two index variables, namely `i-t` and `i-p`. They are used in iterations in order to denote the offset of the pattern in the text and the index used to designate a character inside the pattern. Depending on the font used, these numbers are denoted by i_t (resp. i_p) or

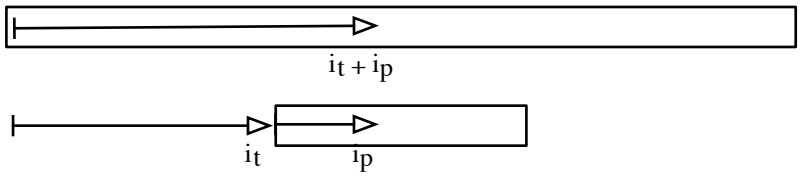

Figure 2.2: Indexes in String Matching Algorithms

`i-t` (resp. `i-p`). The particular values of these index variables are also referred to as *an alignment* of the pattern and the text. Figure 2.2 shows an alignment of a pattern and the text at offset i_t. It shows a phase in the algorithm where the $(i_t + i_p)$'th character of the text is compared with the i_p'th character of the pattern. In the description of the executions of the algorithms, we often refer to "the current characters" in a given alignment. This is the couple of characters consisting of the character that resides in location `i-p` in `p` and the one residing in location (+ `i-t i-p`) in `t`. In other words, the current characters is the couple of characters that are compared in a one particular iteration of the pattern matching procedure. When the current characters do not match, we speak of a *mismatch*.

During the execution of our algorithms, we say that "we move the pattern to the right" whenever we increment `i-t` by some amount. For example, moving the pattern one position to the right means that we replace `i-t` by (+ `i-t 1`) in the next iteration of the loop. Clearly, the quality of an algorithm can be measured by the amount of positions that we can move to the right in one single iteration of the loop without overshooting potential successful alignments.

The brute-force algorithm is a Scheme loop that makes `i-t` vary between `0` and the last offset where an occurrence of the pattern could possibly start, i.e. (− `n-t n-p`). For every value of `i-t`, `i-p` varies from `0` to (− `n-p 1`). The conditional in the body of the loop has four branches:

- The first test checks whether the algorithm has consumed the entire pattern. If this is the case, then the previous iteration of the loop has successfully compared the last character of the pattern with its corresponding character in the text. This means that we have found a match at offset `i-t` which is the location of the first character in the text against which we are aligning `p`.

- If the first test has failed (i.e., the pattern hasn't been entirely matched yet), then we check to see whether we have consumed the entire text. If this is the case, the text has been entirely consumed without having found a match. As a result, we return #f.

- Having checked both the pattern and the text against their boundaries, the algorithm subsequently checks whether the current character in `p` matches

the corresponding character in `t`. If this is indeed the case, we simply continue the loop by considering the next pattern character in the current alignment. This is done by incrementing `i-p` by one in the next iteration of the loop.

- If the current characters do not match, we start from scratch by trying to match the pattern with the next alignment in `t`. In other words, we cycle through the loop with the call (`loop` (+ `i-t 1`) `0`). This resets `i-p` to zero such that the process of checking the pattern restarts entirely.

Performance

What can we say about the efficiency of the brute-force algorithm? From the code we can see that in the worst case, the loop is executed for i_t varying from 0 to $n_t - i_t$ *and* that for every such i_t, it is being executed with i_p varying from 0 to $n_p - 1$. This means that the loop is executed $O(n_t.n_p)$ times[3]. Since the body of the look is in $O(1)$, we thus conclude that the brute-force algorithm has a worst-case performance characteristic that is in $O(n_t.n_p)$. In practice, the brute-force algorithm does not perform too bad in simple cases where the pattern is extremely small (say, a few characters). In other words, if n_p is small, then the brute-force algorithm exhibits linear behaviour. This situation occurs quite frequently when launching the "find" command in word processors. However, when used with large values for n_t and n_p (such as in bioinformatics applications), the brute-force algorithm has a horrible performance.

2.4 The Knutt-Morris-Pratt Algorithm

The reason why the brute-force algorithm is so slow is that, whenever a mismatch occurs, it only shifts the pattern one single position to the right. Moreover, after doing so it reconsiders the entire pattern from scratch, i.e. starting at index 0. This can be sped up considerably. Several algorithms have been invented to do so. One of the best known algorithms is the Knutt-Morris-Pratt algorithm (or KMP for short). The KMP algorithm was discovered in 1977 by Knuth and Pratt, and independently by Morris. The general idea of the KMP algorithm is as follows.

In order to explain the KMP algorithm, let us refer to figure 2.3(a) which shows a non-matching alignment of a pattern and a text. The figure shows the situation where a prefix *part* of the pattern has been found to match part the text. It also assumes that the first character of *rest*1 does not match the first character of *rest*2. In other words, the text is of the form *pre.part.rest*1.*post* and the pattern is of the form *part.rest*2. For example, given the text "Madam, `I'm Adam`" and given the pattern "`I'm cool!`", then *part* corresponds to "`I'm `" (including the whitespace), *res*1 corresponds to "`Adam`" and *res*2 corresponds to "`cool`".

[3]Remember from section 1.5.2 that we can omit constant factors like a in product expressions like $O(a.f)$. However, in the expression $O(n_t.n_p)$ none of the factors is a constant. Both variables depend on the input strings.

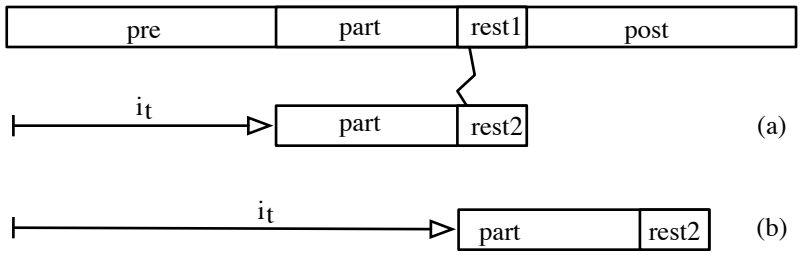

Figure 2.3: Basic (naive) idea of the KMP algorithm

The basic idea of KMP is depicted in figure 2.3(b). The idea is to shift the pattern to the right in order to realign it with *rest*1. In other words, we make the first character of the pattern align with the character of the text that gave rise to the mismatch. In our example, this would align the pattern "`I'm cool`" with "`Adam`" instead of "`'m Adam`" as would be done by the brute-force algorithm. In other words, by aligning the pattern with the character in the text that gave rise to the mismatch, we shift the pattern 3 positions to the right instead of just 1. We will show that this solution is not entirely correct. Nevertheless, this basic idea is important to understand the KMP algorithm. We shall henceforth refer to it as the naive KMP.

The reason why this is a naive solution, is that we can have repetition in the pattern. In order to understand the problem, let us have a look at figure 2.4(a). The grey zones in the pattern and the text indicate repetitions of a certain character sequence. As we can see from figure 2.4(b), we have shifted the pattern too far because the second occurrence of the grey zone *might* be the start of a successful alignment. By applying the naive algorithm, we have been *overshooting* a potential match. The reason is that the first occurrence of the grey zone in the pattern is moved *beyond* the second occurrence of the grey zone in the text such that its second occurrence in the text will never again be aligned with the first occurrence in the pattern. The correct shift is shown in figure 2.4(c).

For example, in the pattern "`lala or lalala`", a grey zone might be "`la`" or even "`lala`". Let us apply the aforementioned naive algorithm to this pattern and to the text "`lalala or lalalala is what I like to sing`". We start at $i_t = 0$ and $i_p = 0$ and we soon find that the prefix "`lala`" matches but that the third `l` of the text doesn't match the whitespace in the pattern. Applying the naive KMP algorithm would align the pattern against the third `l` (i.e. $i_t = 4$) thereby clearly overshooting the solution that we get when aligning the pattern using the offset $i_t = 2$. Notice that this problem only occurs when there is repetition in the pattern. The amount of characters overshot is indicated by k in figure 2.4(c). Notice that k only depends on the pattern and not on the text: the length of the repetitions can easily be determined by analysing the pattern

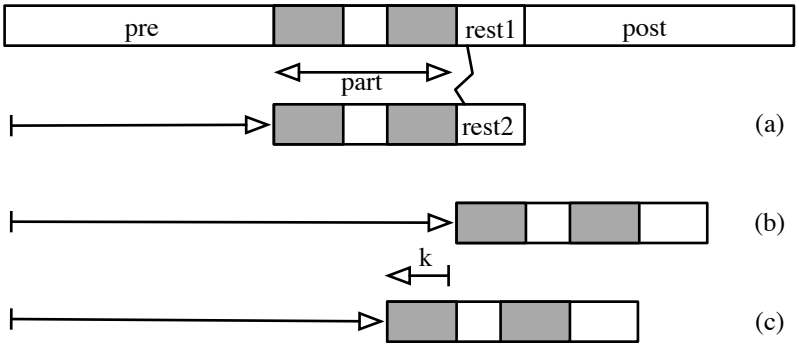

Figure 2.4: Patterns containing repetition

before the actual pattern matching algorithm starts. This *preprocessing phase* is the essence of the actual KMP algorithm.

The following Scheme procedure implements the Knuth-Morris-Pratt algorithm. The structure of the algorithm is identical to the structure of the brute-force algorithm. It is a loop, the body of which consists of four branches: either we have processed the entire pattern successfully, or we have consumed the entire text without finding an occurrence of the pattern, or the current alignment keeps on being successful, or — the `else` branch — a mismatch is encountered. The KMP algorithm differs from the brute-force algorithm in the fourth branch. Instead of shifting the pattern a single character to the right after a mismatch (i.e. $i_t + 1$), we shift it by i_p (as indicated by the naive solution) *minus* an amount that corresponds to the k explained above. We will also denote this k by $\sigma(i_p)$. In the Scheme code, this amount is obtained by calling (`sigma i-p`). `sigma` returns the amount of characters that *cannot* be skipped after a mismatch of p's i_p'th character without running the risk of overshooting a solution. `sigma` is known as the *failure function* for the given pattern. As already said, it is determined *before* the pattern matching loop is started. Assuming that we have a procedure `compute-failure-function` that determines σ, the KMP algorithm can be programmed as shown below. By convention we define (`sigma 0`) to be -1. This makes the fourth branch of the KMP algorithm evaluate the expression (+ `i-t` (− `i-p` (`sigma i-p`))) to (+ `i-t 1`) whenever `i-p` is 0. In other words, just like the brute-force algorithm, the KMP algorithm shifts the pattern one single position to the right whenever no characters match.

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define sigma (compute-failure-function p))
  (let loop
    ((i-t 0)
```

```
 (i-p 0))
(cond
  ((> i-p (- n-p 1))
   i-t)
  ((> i-t (- n-t n-p))
   #f)
  ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
   (loop i-t (+ i-p 1)))
  (else
   (loop (+ i-t (- i-p (sigma i-p))) (if (> i-p 0)
                                         (sigma i-p)
                                         0)))))))
```

Apart from shifting the pattern more than one character to the right, the KMP algorithm also differs from the brute-force algorithm in that it skips some prefix of the pattern. This is taken care of by the final `if`-test in the reiteration. If `i-p` did not increase in the looping process (i.e. it is equal to zero), then none of the pattern's characters have been checked yet. This means that we still have to check the entire pattern in the next iteration. In other words, `i-p` has to stay zero. However, if `i-p` is strictly bigger than zero (i.e. at least one character of the pattern matches the text in the current alignment), then we do not need to reconsider those characters and we take `i-p` to be (`shift i-p`) in the next alignment. They correspond to the grey zone in figure 2.4 about which we already know that it matches the text in the new alignment that will be checked in the reiteration. This is explained further below. Let us now first have a look at an example to understand the behaviour of the pattern matching procedure.

Suppose that we run the procedure as follows:

```
(match "I'm singing lalala down in lalaland" "lalaland")
```

The listing shown below shows the evolution of the indices i_t and i_p throughout the evaluation of the algorithm. The behaviour of the procedure shows that i_t gradually climbs to the value 12 where a first near-match occurs. The near-match is 6 characters long (namely "lalala"). When the algorithm subsequently finds out that #\n does not match #\space, it doesn't simply restart the matching process at $i_t = 13$ which is what the brute-force algorithm would do. Instead, it tries to align the pattern with the text at position $i_t + i_p = 18$, minus the amount of characters in the pattern that might be overshot. This happens to be 4 since "lalaland" aligns with "lalala down..." by shifting the former 4 characters to the left starting from the non-matching whitespace. Therefore the matching process restarts at $18 - 4 = 14$, i.e. "lala down...". But also notice that i_p does not restart from 0 either. Since we have shifted the pattern 4 characters to the left (because we know they will match), we don't need to reconsider them again. Hence the matching restarts at $i_p = 4$, i.e. we immediately proceed by checking the third #\l of "lalaland" against its corresponding character in the text. In other words, we match "lalaland" against "lala down..." starting

at $i_p = 4$. Then #\l in the pattern does not match the #\space. Again, i_t is set to $i_t + i_p - k = 14 + 4 - 2 = 16$ because realigning the start of the pattern with the character causing the mismatch (i.e. #\space) overshoots the second occurrence of "la" in "lala down...". In order to avoid the overshooting, we have to substract 2 from this naive realignment. Hence, the next alignment is against "la down..." (i.e. $i_t = 16$). Furthermore, since we just checked the "la", we don't need to check this again. Hence, $i_p = 2$. This reasoning is repeated once more for this new alignment with $i_t + i_p - k = 16 + 2 - 0$ and $i_p = 0$. From that point on, the algorithm gradually makes i_t climb to 27 where a successful match is found.

```
i-t = 0      i-p = 0
i-t = 1      i-p = 0
i-t = 2      i-p = 0
i-t = 3      i-p = 0
i-t = 4      i-p = 0
i-t = 5      i-p = 0
i-t = 6      i-p = 0
i-t = 7      i-p = 0
i-t = 8      i-p = 0
i-t = 9      i-p = 0
i-t = 10     i-p = 0
i-t = 11     i-p = 0
i-t = 12     i-p = 0, 1, 2, 3, 4, 5, 6
i-t = 14     i-p = 4
i-t = 16     i-p = 2
i-t = 18     i-p = 0
i-t = 19     i-p = 0
i-t = 20     i-p = 0
i-t = 21     i-p = 0
i-t = 22     i-p = 0
i-t = 23     i-p = 0
i-t = 24     i-p = 0
i-t = 25     i-p = 0
i-t = 26     i-p = 0
i-t = 27     i-p = 0, 1, 2, 3, 4, 5, 6, 7, 8
```

Let us now explain how to obtain the `sigma` function that contains the information about repetitions in the pattern. Consider the following hypothetical situation during a particular execution of the KMP matching algorithm:

```
text    = b a b b a b b a b b a b c
pattern = b a b b a b b a b c
```

We observe that the pattern matches the text for 9 characters, after which we encounter a mismatch of `b` against `c`. The naive solution would simply shift the pattern in order to make its first character align with the character in the text that gave rise to the mismatch:

```
text    = b a b b a b b a b b a b c
pattern =                   b a b b a b b a b c
```

However, as we have explained, this is too naive a solution. Since the pattern contains internal repetition, some characters need to be reconsidered in order not to overshoot the solution. At this point we have two options. In the first option, we shift the pattern three characters to the left w.r.t. the positioning of the naive solution. In the second option, we shift the pattern six characters to the left w.r.t. that positioning. They are both shown below.

```
text    = b a b b a b b a b b a b c
pattern =             b a b b a b b a b c
pattern =       b a b b a b b a b c
```

Clearly, the second option is the correct one since the first option gives rise to an overshot solution. The point is that we have to shift the pattern *the maximal amount* of characters to the left in order to be sure. This is the k we were intuitively referring at in figure 2.4. Hence, given a shift i_t and given the fact that i_p characters successfully match (but the $(i_p + 1)$'th character residing at index i_p does not match), then we have to shift the pattern to position $i_t + i_p - k = i_t + i_p - \sigma(i_p)$ for the maximal $k = \sigma(i_p)$. If none of the characters match, we have $i_p = 0$ which requires $\sigma(i_p) = -1$ (such that $i_t + i_p - \sigma(i_p) = i_t + 0 - (-1)$). Hence, if none of the characters match, we shift the pattern one position to the right, exactly like the brute-force algorithm. Let us now figure out how to come up with the other values for σ.

Since we shift the pattern k characters to the left (w.r.t. the naive solution) and since we just finished the previous iteration of the loop by successfully checking those characters against the text, this necessarily implies that the first k characters of the pattern have to be identical to the last k characters in the part that was just checked, i.e. in $p_{0 \to i_p - 1}$. Hence, k is the length of a prefix of p that is also a suffix of $p_{0 \to i_p - 1}$. But because we need to shift the pattern the *maximal* number of positions to the left in order not to overshoot any match, we want $k = \sigma(i_p)$ to be the length of the *longest* such prefix of p that is also a suffix of $p_{0 \to i_p - 1}$. This also explains why there is no need to reconsider those $\sigma(i_p)$ characters in the next alignment. We know that they have already been checked against the text (since they are a suffix of the part which we just finished checking). Hence, i_p can safely restart from $\sigma(i_p)$ in the next iteration.

The following Scheme procedure establishes σ for any given pattern `p`.

```
(define (compute-failure-function p)
  (define n-p (string-length p))
  (define sigma-table (make-vector n-p 0))
  (let loop
    ((i-p 2)
     (k 0))
    (cond
      ((>= i-p n-p)
```

```
      (vector-set! sigma-table (- n-p 1) k))
     ((eq? (string-ref p k)
           (string-ref p (- i-p 1)))
      (vector-set! sigma-table i-p (+ k 1))
      (loop (+ i-p 1) (+ k 1)))
     ((> k 0)
      (loop i-p (vector-ref sigma-table k)))
     (else ; k=0
      (vector-set! sigma-table i-p 0)
      (loop (+ i-p 1) k))))
  (vector-set! sigma-table 0 -1)
  (lambda (q)
    (vector-ref sigma-table q)))
```

The failure function is a Scheme lambda that is returned from this procedure. It encapsulates a vector `sigma-table` that maps an index `q` (which is the `i-p` from the KMP algorithm) to the length of the longest prefix of `p` that is also a suffix of $p_{0 \to q-1}$. As explained, by convention $\sigma(0) = -1$. Furthermore, $\sigma(1) = 0$ since the length of the longest prefix of p that is also a suffix of p_0 is zero. The algorithm therefore starts looking for suffixes starting from $i_p = 2$ until $i_p = n_p - 1$. At the beginning of every iteration, k is the length of the longest prefix of p that is also a suffix of $p_{0 \to i_p - 2}$. The goal of the body is to determine the length of the longest prefix of p that is also a suffix of $p_{0 \to i_p - 1}$.

Suppose that we have found out — in the previous iteration of the loop — that $k = \sigma(i_p - 1)$ is the length of the longest prefix of p that is also a suffix of $p_{0 \to i_p - 2}$. In the next iteration of the loop, we have the following possibilities:

- If $i_p = n_p$ we have reached the end the pattern. The k computed so far is the length of the longest prefix of p that is also a suffix of $p_{0 \to n_p - 1}$. All we have to do is store it in the last position of the table and quit the algorithm.

- If the $(k+1)$'th character of p (residing at index k) continues matching the i_p'th character of p (residing at index $i_p - 1$), then this means that the suffix of $p_{0 \to i_p - 1}$ is just one character longer than the suffix of $p_{0 \to i_p - 2}$ computed by the previous iteration. Hence. $\sigma(i_p) = k+1$ and we continue the loop for the next i_p and the next k.

- If the character doesn't match, but $k > 0$, then we investigate the previously visited prefixes (by reducing k) that are also suffixes of $p_{0 \to i_p - 2}$ in order to see whether one of *those* prefixes *is* followed by the character at position $i_p - 1$. But the lengths of those prefixes already reside in σ from previous iterations. Hence we continue the loop by reducing k to $\sigma(k)$ in the hope that this prefix is a prefix for which the next character *is* equal to the character at position $i_p - 1$. Having found such a prefix, both k and i_p can start to increment again in subsequent iterations in order to find the *longest* such prefix.

- If the character doesn't match and $k = 0$, then we have unsuccessfully visited all possible prefixes of p. As such, the length of the longest prefix of p which is also a suffix of $p_{0 \to i_p - 1}$ turns out to be zero. Hence, we assign the value 0 to $\sigma(i_p)$.

Performance

Assume for a moment that σ has been successfully computed and focus on the main KMP algorithm. We focus on how the sum $i_t + i_p$ evolves since that sum is an indication for the progress of the matching process. In the brute-force algorithm, this sum continuously increases (by 1) as more characters of the pattern match the text. However, that sum is violently reset to $i_t + 1$ every time a mismatch is encountered. Let us now analyse the evolution of the sum for the KMP algorithm. In every execution of the loop, $i_t + i_p$ is incremented by 1 whenever the current characters match. When the characters do not match, i_t is replaced by $i_t + i_p - \sigma(i_p)$ and i_p is replaced by $\sigma(i_p)$. Hence, in this case the sum $i_t + i_p$ remains identical to $i_t + i_p$ in the previous iteration of the loop. Hence, the worst thing that could happen is an alternation between these two situations (i.e. a mismatch following every match) since that behaviour makes the sum increase in the slowest possible way. Since the sum can never exceed n_t, this causes the loop to execute at most $O(2.n_t) = O(n_t)$ times.

In order to see how long it takes to establish σ, we first observe that for all x, $\sigma(x) < x$. Indeed, the length of the longest prefix of p that is also a suffix of $p_{0 \to x-1}$ can never be longer than $x - 1$ since we are only interested in proper prefixes and suffixes. Let us focus on the number $i_p - k$ for a moment.

- In the second branch of the conditional, we observe that the reiteration replaces i_p by $i_p + 1$ and k by $k + 1$. This means that $i_p - k$ remains the same.

- In the third branch of the conditional, the reiteration keeps i_p and replaces k by $\sigma(k)$. Since $\sigma(k) < k$, this means that $i_p - k$ increases.

- The fourth branch of the conditional causes a reiteration with the same k but by replacing i_p by $i_p + 1$. Again, $i_p - k$ increases.

This case-by-case analysis shows that every iteration either keeps $i_p - k$ constant, or makes this number increase by some amount. Because $i_p - k \leq i_p$ and $i_p \leq n_p$, and because i_p and $i_p - k$ cannot stagnate at the same time (check this in the code!), this means that the loop can execute no more than $2.n_p$ times. We conclude that `compute-failure-function` is in $O(n_p)$.

The performance of the KMP algorithm is a combination of establishing the failure function and executing the actual algorithm. Since the former is in $O(n_p)$ and the latter is in $O(n_t)$, the entire KMP algorithm is in $O(n_t + n_p)$. For large n_p and n_t this is *considerably* better than the $O(n_p.n_t)$ result of the brute-force algorithm. Therefore, the KMP algorithm is quite successful in bio-informatics where the patterns are extremely long (about one hundred thousand "characters") and a lot of repetition occurs. For more mundane applications

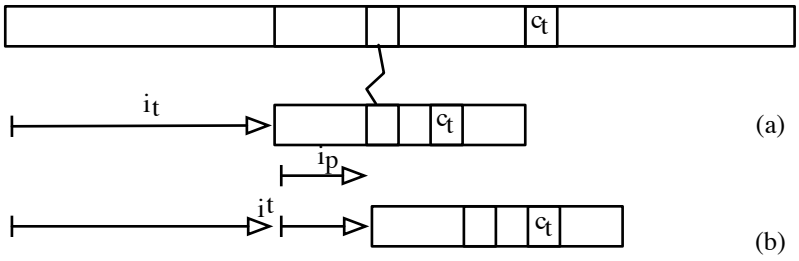

Figure 2.5: The QuickSearch Alorithm

such as finding a short word in a web page or in a word processing document, KMP does not significantly outperform the brute-force algorithm. In those cases, it appears that the average case behaviour for the brute-force algorithm is quite similar to the KMP algorithm.

2.5 The QuickSearch Algorithm

Many clever pattern matching algorithms have been discovered in the past decades. The QuickSearch algorithm was discovered in 1990 by D.M. Sunday. It outperforms the algorithms we have discussed so far. On top of that, QuickSearch is much simpler than the Knuth-Morris-Pratt algorithm and most other algorithms known in the literature. The fact that a simpler algorithm beats more complex algorithms is fairly exceptional in computer science.

In order to understand the QuickSearch algorithm, we invite the reader to have a look at figure 2.5. The figure shows the QuickSearch algorithm right before (figure 2.5(a)) and right after (figure 2.5(b)) a mismatch. Whenever a mismatch occurs, the algorithm considers the text character c_t that is located at the first position to the right of the current alignment. c_t is then searched for in the pattern. In case c_t does not occur in the pattern at all, then the pattern is shifted entirely beyond c_t. No intermediate alignment can result in a successful match since the text contains c_t and the pattern doesn't. However, if c_t *does* occur in the pattern, then we realign the pattern with the text in such a way that the occurrence of c_t in the pattern is aligned with the occurrence of c_t in the text. When there is more than one occurrence of c_t in the pattern then we have to select the *rightmost* such occurrence. If we would take any other occurrence then we run the risk of overshooting a solution by sliding the pattern too far. After the pattern has been realigned like this, we restart comparing the pattern against the text beginning with the very first character of the pattern.

A Scheme procedure implementing the QuickSearch algorithm is presented below. Again the procedure is conceived as a loop with a conditional body. The first branch checks whether the entire pattern has been checked. The second branch checks whether the entire text has been unsuccessfully consumed. The

third branch checks for an additional matching character and continues the loop by considering the next character. Finally, the fourth branch is applied whenever the third branch does not succeed, i.e. when a mismatch occurs. In that case, the algorithm realigns the pattern against the text by sliding the pattern to the right (by the amount indicated by applying the `shift` function to the character `c-t` that is depicted in figure 2.5) and by resetting i_p to zero in order to restart checking the pattern from scratch. The call to `mod` is to cover the exceptional case that occurs when the very last character of the text gives rise to a mismatch. In that case, there is no c_t left and trying to access it would cause us to read a non-existing character at position n_t.

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define shift (compute-shift-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (let ((c-t (string-ref t (mod (+ i-t n-p) n-t))))
         (loop (+ i-t (shift c-t)) 0)))))))
```

The clever thing about the QuickSearch algorithm is that the `shift` function can be established upfront by preprocessing the pattern. This is shown below. The procedure `compute-shift-function` returns a lambda that encapsulates a shift table. The shift table is indexed by ASCII values that vary between the smallest and the biggest ASCII value of the characters occurring in the pattern. For example, consider the pattern "hello". Using `char->integer` we know that the smallest character is #\e with ASCII value 101. Similarly, the greatest value is the character #\o the ASCII value of which is 111. Therefore, the shift table has 11 entries, one for every character *between* #\e and #\o. The shift for all other characters (i.e. all characters that are smaller than #\e and greater than #\o) is the length of the entire pattern, i.e. n_p. This distinction is made in the body of the lambda expression, before the actual shift table is consulted. For all characters that *do* lie between #\e and #\o, the shift table contains the correct shift. For the characters that do not occur in the pattern (such as for example #\f, the shift is $n_p + 1$ as well. Indeed, when c_t does not occur in the pattern, we can safely slide the pattern beyond c_t. Therefore, the (`make-vector` ...) expression returned by `create-table` initialises all entries of the vector to $n_p + 1$. The procedure creates this vector after having established the smallest

and the greatest ASCII values of the characters of the pattern. These ASCII values are kept in the variables `min-ascii` and `max-ascii` while `create-table` processes the entire pattern. The table contains `max-ascii` − `min-ascii` +1 entries. Notice that the table is a Scheme vector that is necessarily indexed starting at index 0 and ending at index `max-ascii` − `min-ascii`. This is the reason why we have to "normalize" a given ASCII value when looking it up in the table; i.e. we substract `min-ascii` in order to obtain an index that lies in the range of the vector.

After having created the table, (`fill-table` 0) traverses the pattern a second time, from left to right. For every character it encounters at a certain `index`, n_p minus `index` is written into the corresponding entry in the shift table. For instance, if an #\a is encountered at the third position in a pattern of length 10, then the table entry that corresponds to the #\a contains 7. This means that the pattern can be shifted 7 positions to the right after encountering a mismatch where c_t equals #\a. By filling the vector from left to right we guarantee that only the position in the pattern of the rightmost occurrence of a character is remembered in the vector.

```
(define (compute-shift-function p)
  (define n-p (string-length p))
  (define min-ascii (char->integer (string-ref p 0)))
  (define max-ascii min-ascii)

  (define (create-table index)
    (if (< index n-p)
      (begin
        (set! min-ascii (min min-ascii (char->integer (string-ref p index))))
        (set! max-ascii (max max-ascii (char->integer (string-ref p index))))
        (create-table (+ index 1)))
      (make-vector (− max-ascii min-ascii −1) (+ n-p 1))))

  (define (fill-table index)
    (if (< index n-p)
      (let* ((ascii (char->integer (string-ref p index))))
        (vector-set! shift-table (− ascii min-ascii) (− n-p index))
        (fill-table (+ index 1)))))

  (define shift-table (create-table 0))
  (fill-table 0)
  (lambda (c)
    (let ((ascii (char->integer c)))
      (if (>= max-ascii ascii min-ascii)
        (vector-ref shift-table (− ascii min-ascii))
        n-p))))
```

Performance

This QuickSearch has the potential of exhibiting sublinear behaviour: $O(N + \frac{n_t}{n_p+1})$ in the best case. The first N comes from the work needed to establish the shift table. By analyzing the code of `compute-shift-function`, we observe that a vector has to be initialized (all entries are taken to be (+ `n-p 1`)). The size of the vector depends on the number of different characters in the pattern. After initializing the vector, the pattern is traversed in order to fill the vector. Hence $N = max(\delta_p, n_p)$ where δ_p is the number of different characters occurring in the pattern (this is also known as the *alphabet size* of the pattern). The second $\frac{n_t}{n_p+1}$ comes from the fact that, in the best case, the pattern immediately gives rise to a mismatch for each and every alignement and the character c_t following the alignment does not occur in the pattern. This results in a $n_p + 1$ shift for every comparison.

The fact that QuickSearch has the potential of exhibiting sublinear behaviour stems from the fact that it is able to skip large portions of the input text. In other words, the algorithm has the benefit of not checking every single character in the text. This is in sharp contrast with the brute-force algorithm and the KMP algorithm, which examine every single character in the text at least once. However, this can also be problematic. If the application of a pattern matching algorithm in a software application has an additional goal, then it might be desirable to check every single character. For example, suppose that —apart from finding the pattern — we also want to count the number of words in a text. This actually means that we need to count the number of whitespace characters. By skipping large portions of the text we clearly will not end up with the correct number of whitespace characters. In such cases, the KMP algorithm is a good solution.

It is not hard to come up with a worst-case example that shows that QuickSearch is actually in $O(n_p.n_t)$ (find it!). However, in practice, QuickSearch beats all other approaches.

2.6 Strings vs. Data Storage

In this chapter, we have studied ways to retrieve information from strings. Although strings are easily composed using `string-append`, the algorithms presented show that retrieving information from strings is not easy. As explained in section 1.4.3, one of the central themes of the course is how to implement dictionaries. Suppose that we were to implement dictionaries using strings. We might consider representing a dictionary as a large string that contains all the key-value pairs in textual format. `string-append` would be used to implement the `insert!` function. Indeed, inserting a key-value pair would boil down to appending the key-value pair to the string representing the dictionary. However, as the algorithms in this chapter show, the implementation of `find` would not be very efficient. As the dictionary grows, `find` gets slower and slower. Moreover, the efficiency of `find` not only depends on the *number* of key-value pairs, but also on the number of characters that make up those pairs.

The deeper reason why strings are not a good way to represent large collec-

tions of data is that strings are extremely structure-shy. As we will see in the rest of the course, richer data structures (i.e. data structures which exhibit more internal structure than mere "flat" strings) allow for much faster implementations of `find`. Strings are not a very good representation for storing information in a computer. In the optimal situation, they are merely used by programs in order to communicate with users (e.g. using `display`). Inside the guts of our programs, we try to shun strings as much as possible.

Chapter 3

Linear Data Structures

From the previous chapter, we conclude that strings are among the poorest data structures imaginable. Therefore, from this chapter on, we start our investigation of richer data structures that allow for easier and faster retrieval of stored information. We start by studying linear data structures.

One of the most elementary ways to structure a collection of items in our daily life is to put them in a row, one next to the other. E.g., in a bookcase, we put the books next to each other, possibly sorted by author. Other examples include customers standing in line to pay their groceries in a supermarket, a pile of files waiting to be processed by a secretary and so on. Such a linear organisation of data elements is as natural in computer science as it is in the world that surrounds us. Think about a list of contacts in your favourite chat client, a list of songs in iTunes, a list of candidates in an electronic voting system, and so on.

Linear data structures are among the oldest ones discovered in computer science. They were extensively studied during the invention of the programming language Lisp (which is the second oldest programming language in existence). In fact, the name "Lisp" is actually an acronym for "list processing". Since Lisp was the direct precursor of Scheme, it should come as no surprise that Scheme is extremely well-suited to process linear data structures as well. However, as we will see in this chapter there is more to linear data structures than just 'storing things in a Scheme list'. Linear data structures come in several flavours and certain decisions concerning the way they are represented in computer memory have tremendous repercussions on the performance characteristics of the algorithms that operate on them.

We start by defining exactly what we mean by linearity. Based hereupon, we present a number of ADTs that define linear data structures abstractly and we discuss a number of implementation strategies for these ADTs. We discuss the performance characteristics for all implemented operations and whenever different from $O(1)$, we seek for ways to speed them up. The most notorious such operation is `find`. It is — as explained in section 1.4.3 — one of the central operations studied in this text. We will analyse its performance characteristic

for every data structure studied. As such, `find` is also studied for the linear data structures presented in this chapter. Section 3.4 studies the different versions of `find` for linear data structures and reveals some clever tricks that can speed up `find` considerably.

3.1 Using Scheme's Linear Data Structures

Scheme features two ways of organising data in a linear way. Indeed, both Scheme's built-in vectors as well as its dotted pairs can be used to construct sequences of data elements. So why spending an entire chapter on linear data structures in Scheme? The answer is twofold. First, there are a number of important disadvantages that have to be dealt with when using "naked" Scheme vectors and dotted pairs. As we will see, it turns out to be very beneficial for the performance characteristic of many procedures if we "dress up" Scheme's naked pairs and vectors a bit by enhancing them with auxiliary information. Second, just knowing how to store data into vectors and lists doesn't suffice. As computer scientists, we also need to study algorithms operating on a linear data structure from a performance characteristic perspective.

3.1.1 "Naked" Vectors

The simplest way to organise a collection of data values in a linear way is probably to store them in a Scheme vector. Remember that a vector is a linearly ordered compound data structure that consists of indexed entries and that allows one to store and retrieve those entries using `vector-ref` and `vector-set!` in $O(1)$. This is called the *fast random access property* of vectors. However, there is a price to pay for this efficient behaviour:

- When using vectors, one has to know the exact number of data values one wishes to store upfront because the length of the vector is fixed at creation time. Whenever the number of data values one wishes to store is not known upfront, one typically estimates some upper bound in order to create a vector that is "big enough" to contain any "reasonable" number of elements that one might wish to store during the lifetime of the vector. This has two important drawbacks. First, it can result in a waste of memory if a particular execution of the program doesn't use the vector to its full extent. Second, it requires us to maintain an additional "counter" variable to remember the index that is considered to be the "last" vector entry that contains meaningful information. In big software systems with a few dozen vectors we thus have to maintain such a counter variable for every single vector. Storing all those counters in different Scheme variables is extremely error-prone. It would be much simpler to group together each vector with its counter variable (e.g. by storing them in a dotted pair). This is exactly what we do in the implementations presented in this chapter. Techniques like this make us refer to ordinary Scheme vectors as "naked" vectors.

- Another important problem arising from the fact that the length of a vector is always predefined, is that we have to decide what is to happen whenever elements are added beyond the vector's capacity. One possibility is to produce an error. Another is to make the vector grow. However, making the vector grow is not an operation that is built into Scheme. It requires us to create a *new* vector which is big enough and it subsequently requires us to copy all elements from the old vector into the new one. This results in an operation — called a *storage move* — that is $O(n)$ which is disappointing given the fact that having an $O(1)$ accessor (i.e. `vector-ref`) and mutator (i.e. `vector-set!`) is one of the main reasons to opt for vectors in the first place.

Although these considerations make us conclude that naked Scheme vectors are not very well-suited to store data in practical applications, vectors remain interesting data structures. As the rest of this chapter shows, provided that we build the right abstractions "on top of" vectors, vectors remain an extremely useful compound data type that has attractive performance characteristics for some particular use cases.

3.1.2 Scheme's Built-in Lists

Scheme lists are often poorly understood by computer science students. The reason for the misunderstanding is the omnipresent confusion between Scheme's *lists* and Scheme's *dotted pairs*. A dotted pair is a "cell" with exactly two components (that was constructed using `cons`). In order to really grasp the difference between dotted pairs and lists, it is important to keep in mind that Scheme's lists are actually nothing but a conventional way to organise a series of dotted pairs. They do not exist in Scheme *as a separate data type*. One speaks of a list in Scheme when a series of dotted pairs is linked up in a particular way: a Scheme list is a dotted pair whose "cdr" is either '() (called "the empty list") or a dotted pair in its turn. Hence, (`cons 1` (`cons 2` '())) is considered to be a Scheme list while (`cons 1 2`) is not.

In Scheme, lists are often presented as an alternative that circumvent the main disadvantages of: when building a list, it is not necessary to know its capacity upfront. Using `cons`, one can always create a new entry to store a record and link that entry to an existing list.

However, using "naked Scheme lists" in practice poses many problems:

- Scheme's lists only allow one to "cons something upfront" since a Scheme list is actually nothing more than a reference to its first dotted pair. There is no direct reference to the last pair in the list. This means that operations that need to add an object to or change something about the list at any position which is not the first one, turn out to be slow. Examples are `add-to-end` that adds an element to the end of the list and `append` which concatenates two lists and thus needs access to the last pair of the first list. Both operations are in $O(n)$ since they need to traverse the entire list

before doing any interesting work: starting at the first pair, they need to recursively work their way to the end of the list (following "cdr pointers") in order to reach the final dotted pair. This loss of efficiency is the price to pay for the flexibility offered by lists. However, as we will see, "dressing up" lists with additional information (e.g. maintaining a direct reference to the very last pair of the list) can speed up things considerably.

- Another drawback that comes with naked Scheme lists is a consequence of the way arguments are passed to procedures in Scheme. Whenever a procedure is called with dotted pairs, Scheme passes the dotted pairs themselves to that procedure instead of any variables containing those dotted pairs. Manipulating parameters of a procedure will thus not change any other variables. This is exemplified by the following code excerpt:

```
(define (swap p1 p2)
  (let ((temp p1))
       (set! p1 p2)
       (set! p2 temp)))

(define my-pair1 (cons 1 2))
(define my-pair2 (cons 3 4))
(swap my-pair1 my-pair2)
```

  When executing this code, `swap` receives the pair contained by the variable `my-pair1` and the pair contained by the variable `my-pair2`. Hence, it assigns the first pair to the variable `p2` and the second pair to the variable `p1`. However, *after* returning from `swap`, the original variables `my-pair1` and `my-pair2` still contain the original values. Scheme passes the values stored in the variables to the procedure `swap` and not the variables themselves. Hence, the contents of the variables `my-pair1` and `my-pair2` is not affected by `swap`.

  This behaviour of Scheme is particularly problematic when storing data in "naked" Scheme lists. Indeed, suppose we implement a procedure `add-to-first` which is to add a new element to the head of a Scheme list. Its implementation might look as follows:

```
(define (add-to-first l e)
  (set! l (cons e l)))
```

  Because of the parameter passing mechanism just described, this procedure only affects the local parameter `l` and not the variables that are used during the call of the procedure. Hence calling the procedure with an expression like (`add-to-first my-list my-element`) has no effect on the list contained by the variable `my-list`.

- A final drawback of Scheme lists is that they are not generic in the way genericity was introduced in section 1.4.2. Scheme features several built-in

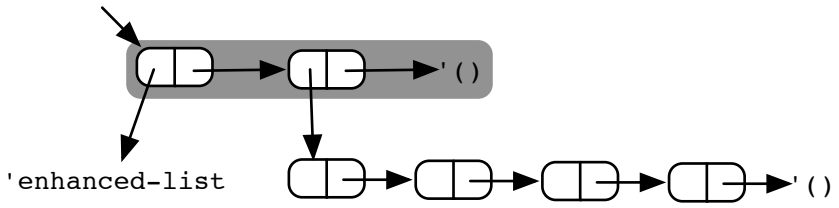

Figure 3.1: A typical enhanced list

operations (such as `member?`, `memv?` and `memq?`) to find out whether or not a data element belongs to a list. However, these procedures explicitly rely on `equal?`, `eqv?` and `eq?`. It is impossible to alter these operations if one wishes to use them using a different definition for equality. For instance, suppose one has a list containing persons (i.e. data values beloning to some ADT `person`, the details of which are not relevant here) and suppose one wishes to search the list for a person given his or her name. None of the aforementioned equality operators will do the job because we actually want the membership test to use our own (`person-equal? p1 p2`) procedure which is designed to return `#t` when `p1` and `p2` have an identical first name. In other words, we would actually want to call one of the built-in membership tests in such a way that it uses *our* equality procedure. Unfortunately, this kind of genericity is not built into the standard list processing procedures that come with Scheme. We conclude by saying that Scheme's lists are not generic.

3.1.3 Enhanced Lists and Enhanced Vectors

The standard technique that is used to solve the problems with naked Scheme lists consist of replacing those lists by *enhanced Scheme lists*. Enhanced Scheme lists are lists that have an additional level of indirection that refers to the actual list. The additional level of indirection is called *the header* of the enhanced Scheme list. Figure 3.1 shows an enhanced list containing four elements. Its header consists of the two dotted pairs that are displayed on a grey background. Technically spoken, enhanced Scheme lists consist of a data structure for the header (e.g. a list of dotted pairs) that holds a reference to the first dotted pair of the actual Scheme list. The code below shows an enhanced list:

```
(define tag 'enhanced-list)

(define (make-enhanced-list a-scheme-list)
  (list tag a-scheme-list))
```

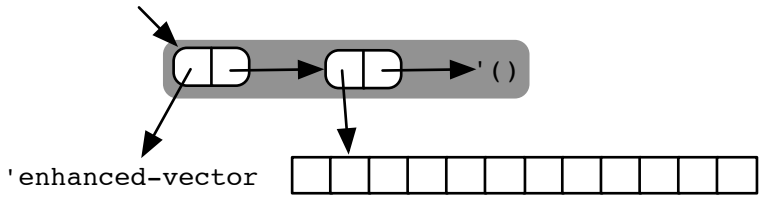

Figure 3.2: A typical enhanced vector

Enhanced lists can solve the first problem discussed in section 3.1.2. Apart from a descriptive tag[1] and a reference to the first dotted pair of the actual list, the header can store a number of useful additional data fields such as e.g. a direct reference to the very last element in the Scheme list. Like this, an $O(1)$ access is provided to the last element of the list. This can speed up a number of list operations considerably. Enhanced Scheme list are actually what most of this chapter is about. We will see that storing different kinds of additional data fields in the enhanced Scheme list results in better performance characteristics for a number of important operations defined on lists. E.g., instead of *computing* the length of a list, we might as well *store* the length explicitly in the header.

Enhanced lists also solve the second problem of section 3.1.2. Since the header stores a reference to the first dotted pair of the actual list, the first element of the list can be altered by modifying the reference *inside* the header instead of modifying the list itself. For instance, given our Scheme definition of enhanced lists presented above, we *might* destructively change the first dotted pair of a list as follows.

```
(define (add-to-front l e)
  (set-cdr! l (cons e (cdr l))))
```

The solution to the third problem of section 3.1.2 lies in using higher order procedures in the constructors for enhanced lists. Remember from section 1.4.2 that turning a constructor into a higher order procedure is the standard recipe to achieve genericity in Scheme. Every time we create a new list, we pass a comparator function to the constructor. The idea is to store that comparator function in the header as well. For example, if `new` is the name of the constructor, then we might use the call (`new eq?`) to create an enhanced list that uses Scheme's `eq?` whenever it needs to compare data values. For instance, the ADT's procedure `find` can use that equality operator by accessing it in the enhanced list's header.

We finish this section by mentioning that it is also meaningful to talk about *enhanced vectors*. Similarly, these are data structures that consist of a header — that stores additional information — that maintains a reference to the actual vector in which the actual data elements reside. The additional information can

[1]The tag is also known as a manifest type.

be used to speed up some operations or to make the information in the vector more meaningful, e.g. by storing the counter that designates the last position in the vector that contains meaningful information. Figure 3.2 shows a typical example of an enhanced vector. Its header is displayed on a grey background again.

In what follows, we present a number of useful linear ADTs and we describe how to implement them in Scheme using enhanced vectors and enhanced lists. As we will see, depending on what is stored in the header, different performance characteristics will be obtained for the operations of the ADTs

3.2 Positional Lists

Before we delve into the realm of implementation strategies for linear ADTs, let us first present a number of definitions that will help us focus the discussion.

3.2.1 Definitions

We define a linear data structure as a data structure in which each data element has a unique *position*. The positions are linearly organized: this means that each position has a unique *next* position as well as a unique *previous* position. There are two exceptions to this rule: the *first position* is a special position that has no previous position and the *last position* is a special position that has no next position. A data structure that fullfils these properties will be referred to as *a positional list*.

Notice that this abstract specification is not necessarily related to Scheme's lists. The fact that we have a unique first position, a unique last position and that every other position has a unique next position and a unique previous position also holds for Scheme's vectors. Also notice that nothing in our definition prescribes that the positions should be numeric. A position is an abstract entity that is solely defined in terms of its neighbouring positions.

Positional lists are formally defined as an ADT in section 3.2.2. Positional lists are given a vector implementation in section 3.2.5 and Scheme list implementations in sections 3.2.6, 3.2.7 and 3.2.8. Instead of talking about a Scheme list implementation, we will be talking about a *linked implementation* since Scheme lists essentially consist of dotted pairs that are linked together by their "cdr-pointers". Hence, we study one vectorial implementation and three linked implementations of the ADT. Section 3.2.9 presents a comparison of the performance characteristics of the four implementations.

3.2.2 The Positional List ADT

In the definition of the ADT shown below, we deliberately choose *not* to specify what we mean exactly by a position. As we will see, positions can be indices in a vector (i.e. numbers) as well as references to some Scheme list (i.e. dotted pairs). The actual data types that are used as positions is not really important

for users of the ADT. Following the definitions presented in section 3.2.1, the only thing that matters is that every position in a positional list (apart from the first position and the last position) has a next position and a previous position.

Since the actual type of the positions is not a part of the definition of the ADT, the ADT `positional-list` is parameterized by the position data type `P`. As we will see, different implementations of the ADT use different data types for `P`. E.g., in a vector implementation, positions correspond to vector indices which means that the role of `P` is concretised by the `number` data type. In other implementations, `P` is concretised by `pair` such that positions correspond to dotted pairs. Remember from section 1.4.2 that it is our goal to make storage data structures as generic as possible such that they can be used many times to store different types of data elements. That is why we have also parameterized the positional list ADT with the data type `V` of the value elements stored.

Given these two parameterizations, we can now define a generic ADT `positional-list < V P >` the specification of which is shown below. For a particular implementation of this ADT and for a particular usage, we have to think of concrete data types for `T` and `P`. For example, suppose that we choose the vectorial implementation that represents positions as numbers. If we use this implementation to store strings, then the resulting positional lists are of type `positional-list<string number>`. Any concrete positional list that we use in our programs has a data type like this.

```
1  ADT positional−list < V P >
2
3  new
4      ( ( V V → boolean) → positional−list < V P > )
5  from−scheme−list
6      ( pair ( V V → boolean) → positional−list < V P >) )
7  positional−list?
8      ( any → boolean )
9  length
10     ( positional−list < V P > → number )
11 full?
12     ( positional−list < V P > → boolean )
13 empty?
14     ( positional−list < V P > → boolean )
15 map
16     ( positional−list < V P >
17       ( V → V' )
18       ( V' V' → boolean) → positional−list < V' P > )
19 for−each
20     ( positional−list < V P > ( V → any ) → positional−list < V P > )
21 first
22     ( positional−list < V P > → P )
23 last
24     ( positional−list < V P > → P )
```

```
25 has−next?
26     ( positional−list < V P > P → boolean )
27 has−previous?
28     ( positional−list < V P > P → boolean )
29 next
30     ( positional−list < V P > P → P )
31 previous
32     ( positional−list < V P > P → P )
33 find
34     ( positional−list < V P > V → P ∪ { #f } )
35 update!
36     ( positional−list < V P > P V → positional−list < V P > )
37 delete!
38     ( positional−list < V P > P → positional−list < V P > )
39 peek
40     ( positional−list < V P > P → V )
41 add−before!
42     ( positional−list < V P > V . P → positional−list < V P > )
43 add−after!
44     ( positional−list < V P > V . P → positional−list < V P > )
```

The constructor `new` depends on a comparator function that can be used to compare any two values in the positional list. The procedural type of such comparators is (`V V` → `boolean`). As explained in section 1.4.2, turning the constructor into a higher order procedure is Scheme's technique to deal with generic data structures. The comparator will be used by `find` in the search process that arises when trying to retrieve a given data element from a positional list. The comparator's job is to return `#t` whenever `find`'s argument matches the data values that are being investigated (one by one) during the search process. Apart form `new`, we have `from-scheme-list` which is an alternative constructor that returns a positional list given an ordinary Scheme list of data values (which is technically just a pair) and a comparator that works for those values. E.g., (`from-scheme-list` '(1 2 3 4) =) creates a positional list (containing the four numbers contained by the Scheme list) that uses Scheme's = operator for comparing elements.

Given any Scheme value[2], then `positional-list?` can be used to check whether or not that value is a positional list. Given a positional list, the operation `length` returns the number of data elements sitting in the list and `full?` can be used as a predicate to check whether the list is full. Likewise, `empty?` is a predicate that verifies whether or not the list is empty.

The operations `first`, `last`, `has-next?`, `has-previous?`, `next` and `previous` are used to navigate through positional lists. Given a non-empty list, then `first` and `last` return the first and the last position of that list (i.e. a reference that corresponds to the first or last element). Given a position `p` in a positional

[2]Remember that we use the data type `any` for the set of all possible Scheme values.

list `l`, then (`next l p`) returns the next position in the list, i.e. a reference to the position that follows the position `p`. Similarly, (`previous l p`) returns the position that precedes `p`.

The operations `map` and `for-each` are very similar. `for-each` takes a positional list with values of type `V` and a procedure that accepts a value of type `V` (but which returns any other Scheme object). `for-each` simply traverses the positional list from the first position to the last position, and applies the procedure to every data element of type `V` that it encounters. For example, (`for-each l display`) traverses the positional list `l` and shows its elements on the screen. `map` is slightly more complicated. It also traverses a positional list from the first position to the last position. However, in contrast to `for-each`, `map` returns a new positional list. `map` takes a positional list and a procedure that maps values of type `V` onto values of another type `V'`. The result is a positional list with values of type `V'` that arises from applying the given procedure to every data element sitting in the original positional list. E.g. given a positional list `l` that stores numbers and given a procedure `p` that maps numbers onto booleans (e.g. `odd?`) then (`map l p`) returns a new positional list storing the corresponding boolean values. Surely, this new positional list also needs a comparator just like any other positional list. This explains the third parameter of `map`.

Finally, the operations `find`, `update!`, `delete!`, `peek`, `add-before!` and `add-after!` have to be discussed. `find` takes a key to be searched for in the positional list. It searches the list and returns the position of the element that matches the given argument. The comparator that was provided when the positional list was constructed is used for the matching. Given a list and a position, then `update!` changes the value that sits in the list at the given position. `delete!` removes the value from the list and `peek` reads the value (without deleting it) that is associated with the given position. `add-before` adds a new value to the list. The value is inserted right before the given position. This means that all elements residing at that position and at positions "to the right of that given position" are conceptually shifted to the right. In case this insertion behaviour should not be required, one can use `add-after!`. This operation also inserts a new value into the list and shifts existing values to the right as well. However, in contrast to `add-before!`, the value sitting at the given position itself does not move.

Caution is required with `add-before!` and `add-after!`. By default, both operations take a value and a position. The value is added right before or right after the given position. But what shall we do when there are no positions in the list? This is no hypothetical situation since it occurs whenever we try to add elements to an empty positional list. That is why both `add-before!` and `add-after!` take the position as an *optional* argument. When omitting the argument in the case of an empty list, both `add-before!` and `add-after!` simply add the value as the first and only element of the list. But what is the meaning of `add-before!` and `add-after!` when the position argument is omitted for non-empty lists? In these cases, we use the convention that `add-before!` adds the element before *all* positions. In other words, we add the element to the front of the list. Similarly, calling `add-after!` on a non-empty list without providing

a position adds the value after all positions. I.e., it adds the element to the rear of the list.

3.2.3 An Example

Before we start with our study of implementation techniques for the positional list ADT, we first give an example of how a programmer might use the ADT just described. The idea of the example is to use the list to represent a simple todo-list that might be part of an agenda application. We first implement a small auxiliary `entry` ADT which groups together a day, a month and a note describing what is to be done on that particular day.

```
(define entry-tag 'entry)
(define (make-entry day month note)
  (cons (cons entry-tag day) (cons month note)))
(define day cdar)
(define month cadr)
(define note cddr)
```

In the following code excerpt, we show how this "mini-ADT" can be used to create a number of entries that will be stored in a busy professor's todo-list.

```
(define todo-list-entry-1 (make-entry 5 10 "Give Lecture on Strings"))
(define todo-list-entry-2 (make-entry 12 10 "Give Lecture on Linearity"))
(define todo-list-entry-3 (make-entry 19 10 "Give Lecture Sorting"))
```

Let us now create a new positional list that can be used to store elements of data type `entry`. This is shown in the following code excerpt. We start by defining a procedure `entry-eq?` that compares two entries by checking whether or not they represent events that occur on the same day of the same month. This procedure is used to create a new positional list that can be used to store such entries.

```
(define entry-eq? (lambda (entry1 entry2)
                    (and (eq? (day entry1) (day entry2))
                         (eq? (month entry1) (month entry2)))))
(define todo-list (new entry-eq?))
```

The variable `todo-list` now contains a Scheme value of data type `positional-list<entry pair>` or type `positional-list<entry number>` depending on whether we opt for a linked list implementation (like the one that is presented in section 3.2.6) or for the vector implementation (studied in section 3.2.5).

The procedure calls shown below show how a particular todo-list can be composed. The three aforementioned entries are added to end of the list, one after the other.

```
(add-after! todo-list todo-list-entry-1)
(add-after! todo-list todo-list-entry-2)
(add-after! todo-list todo-list-entry-3)
```

Now suppose that our professor decides that he has to prepare his class on linearity before teaching it. `lecture-2` is the position associated to the lecture of 12 October. A new entry right before that lecture is added, to wit the preparation of the lecture. Now suppose our professor decides to have a resting period after preparing his lecture. A reference to the position of to the preparation is obtained and stored in the variable `prepare-lecture`. Subsequently, an entry for the rest is scheduled after the preparation. Finally, the title of the second lecture is changed.

```
(define lecture-2 (find todo-list (make-entry 12 10 '())))
(add-before! todo-list (make-entry 8 10 "Prepare Lecture on Linearity") lecture-2)

(define prepare-lecture (find todo-list (make-entry 8 10 '())))
(add-after! todo-list (make-entry 9 10 "Have a Rest") prepare-lecture)

(define lecture-3 (find todo-list (make-entry 19 10 '())))
(update! todo-list lecture-3 (make-entry 19 10 "Give Lecture on Linear ADTs"))
```

At this point, our professor might decide to print out his todo-list by calling:

```
(for-each
   todo-list
   (lambda (entry)
     (display (list "On " (day entry) "/" (month entry) ": " (note entry)))
     (newline)))
```

3.2.4 The ADT Implementation

In what follows, we discuss four implementations of the `positional-list` ADT. All four implementations use a different representation for the data structure. As a consequence, most procedures that implement the ADT operations are implemented differently in all four cases. However, this is not the case for all procedures. A number of procedures are implemented on top of the other — lower level — procedures and therefore do not explicitly depend on the representation. We therefore group these procedures in a Scheme library that imports the lower level procedures which do rely on the representation. Those lower level procedures are put in four libraries; one per implementation strategy. By selecting a different library to import, a different implementation strategy is chosen. This is graphically shown in figure 3.3. The library shown on the top is the one that users have to import when using positional lists. This library imports one of the four lower level implementation libraries in its turn. Before we move on to the first implementation of the ADT in section 3.2.5, we first discuss these procedures.

It may come as no surprise that `map` and `for-each` do not depend on any particular representation. The following code excerpt shows their implementations. From the code, we clearly see how the ADT procedures can be used to construct their implementation. It is very instructive to study these procedures in detail

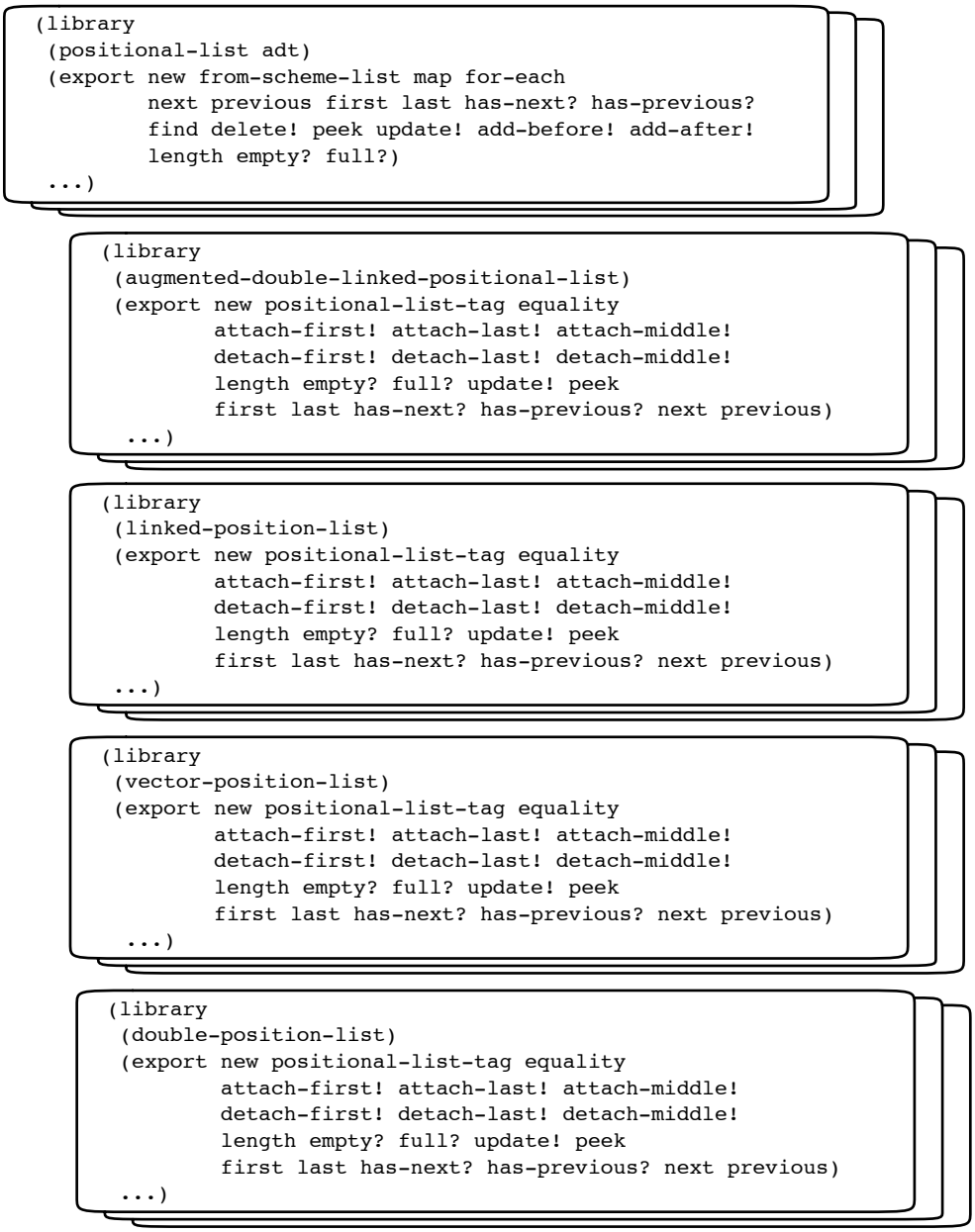

Figure 3.3: The structure of positional list implementations

because they reveal how the abstractions provided by the `positional-list` ADT are to be used to process positional lists in fairly complex ways.

```
(define (for-each plst f)
  (if (not (empty? plst))
    (let for-all
      ((curr (first plst)))
      (f (peek plst curr))
      (if (has-next? plst curr)
        (for-all (next plst curr)))))
  plst)

(define (map plst f ==?)
  (define result (new ==?))
  (if (empty? plst)
    result
    (let for-all
      ((orig (first plst))
       (curr (first
              (add-after! result (f (peek plst (first plst)))))))
      (if (has-next? plst orig)
        (for-all (next plst orig)
                 (next (add-after! result
                                   (f (peek plst (next plst orig)))
                                   curr)
                       curr))
        result))))
```

`for-each` simply considers the first position `curr` of its input list and enters the `for-all` loop. In every iteration, the procedure `f` is applied to the value sitting in the current position. As long as the current position has a next position, the loop is re-entered with the next position of the current position. As such, all values in the position list are visited and used as an argument for `f`.

The implementation of `map` takes a positional list `plist`, a function `f` and an equality operator `==?` to be used by the new positional list that is returned by `map`. The implementation creates a new positional list `result` using `new`. It makes the variable `orig` refer to the first position in the original positional list and then traverses the positional list using `next` as long as the predicate `has-next?` returns #t. In every step of the iteration, `peek` is used to read the element at position `orig` and `f` is applied to it. The result of that application is stored after position `curr`. `next` is used to determine the next position in both the original list and in the result.

Another procedure that can be built on top of the more primitive procedures of the ADT, is the "secondary" constructor `from-scheme-list` which constructs a positional list based on a naked Scheme list. Its implementation is shown below. It simply creates a new positional list using `new` and then traverses the

Scheme list. During that traversal, every element of the Scheme list is added to the positional list using `add-after!`. Notice that the performance characteristic of `from-scheme-list` heavily depends on the performance characteristic of the procedures used. As we will see, especially the efficiency of `add-after!` will be crucial to keep the performance of `from-scheme-list` within reasonable bounds.

```
(define (from-scheme-list slst ==?)
  (define result (new ==?))
  (if (null? slst)
    result
    (let for-all
      ((orig (cdr slst))
       (curr (first (add-after! result (car slst)))))
      (cond
        ((not (null? orig))
         (add-after! result (car orig) curr)
         (for-all (cdr orig) (next result curr)))
        (else
         result)))))
```

Remember from the abstract description in section 3.2.1 that every position has a previous position and a next position, except for the first position and the last position. Hence, there are three kinds of positions: positions that do not have a previous position, positions that do not have a next position and positions that have both a next as well as a previous position. This is reflected by the fact that all four implementations provide three procedures to add a new data element and three procedures that can remove a data element. Needless to say, the implementation of these six procedures will heavily depend on the concrete representation of the positional list. The procedures are named:

`attach-first!`	`detach-first!`
`attach-middle!`	`detach-middle!`
`attach-last!`	`detach-last!`

Provided that our four concrete implementations implement (and export) these procedures, we can already present an implementation for the ADT procedures `delete!`, `add-before!` and `add-after!`. Notice from figure 3.3 that these six procedures are not exported by the final implementation of the ADT!

Deleting an element from a positional list depends on the position the element occupies. For the first position and for the last position (which is detected by the fact that calling `has-next?` yields #f) we use the dedicated procedures `detach-first!` and `detach-last!`. In all other cases, we use `detach-middle!`. At this point it is impossible to establish a performance characteristic for `delete!` since it clearly depends on the representation-dependent procedures.

```
(define (delete! plst pos)
  (cond
    ((eq? pos (first plst))
```

```
    (detach-first! plst))
   ((not (has-next? plst pos))
    (detach-last! plst pos))
   (else
    (detach-middle! plst pos)))
  plst)
```

`add-before!` inserts a value before a given position. `add-after!` inserts a value right after the position provided. When `add-before!` doesn't receive a position argument, then the element is added to the front of the list. Similarly, omitting the position argument for `add-after` adds the element to the rear of the list. Their implementation is given below.

```
(define (add-before! plst val . pos)
  (cond
    ((and (empty? plst)
          (not (null? pos)))
     (error "illegal position (add-before!)" plst))
    ((or (null? pos)
         (eq? (car pos) (first plst)))
     (attach-first! plst val))
    (else
     (attach-middle! plst val (previous plst (car pos)))))
  plst)


(define (add-after! plst val . pos)
  (cond
    ((and (empty? plst)
          (not (null? pos)))
     (error "illegal position (add-after!)" plst))
    ((null? pos)
     (attach-last! plst val))
    (else
     (attach-middle! plst val (car pos))))
  plst)
```

Both procedures use `attach-middle!` to add the new element to the list. `add-after!` uses its argument position to call `attach-middle!`. `add-before!` uses the previous position of the argument position. Notice that this position first has to be computed using `previous`. When the position is omitted, `add-after!` uses `attach-last!`. Similarly, `add-before!` uses `attach-first!`. As for `delete!`, it is not possible to come with a performance characteristic without have a better look at the performance characteristic of the procedures `attach-first!`, `attach-middle!` and `attach-last!` provided by the concrete implementation.
Apart from the six attaching and detaching procedures, concrete implementations also need to provide a definition for the identifier `positional-list-tag`.

It is the tag sitting in the enhanced list or vector that is used to represent a positional list. We have chosen for this tag to be different for every implementation. Like that, the tag describes the concrete implementation that was chosen. The tag is used by the following procedure that can be used to verify whether or not a given scheme value is a positional list. We simply check whether the value is a pair (which an enhanced list or enhanced vector necessarily should be) and whether that pair contains the right tag in its car. Note that we rely on the fact that `and` is a special form and not a procedure.

```
(define (positional-list? any)
  (and (pair? any)
       (eq? (car any) positional-list-tag)))
```

The final procedure that can be implemented without explicitly relying on a particular representation is `find`. Remember that `find` is one of the central operations studied in this text. The goal of `find` is to search the positional list for a given data element — called the key. The implementation of `find` for positional lists is the simplest implementation possible. It considers the first position of the list and then verifies every single element of the list by comparing it (using the positional list's own comparator that is accessed using the procedure `equality` every representation needs to provide as well) with the key. Successive combinations of `has-next?` and `next` are used to traverse the positional list from the first position to the last position.

```
(define (find plst key)
  (define ==? (equality plst))
  (if (empty? plst)
    #f
    (let sequential-search
      ((curr (first plst)))
      (cond
        ((==? key (peek plst curr))
         curr)
        ((not (has-next? plst curr))
         #f)
        (else
         (sequential-search (next plst curr)))))))))
```

This algorithm is known as *the sequential searching algorithm* as is searches the positional list by considering the list's elements one after the other, in a sequence. In the body of the `sequential-search` loop, we observe two stop conditions to end the iteration: either the key is found, or the end of the list has been reached. Clearly the sequential searching algorithm exhibits an $O(n)$ behaviour in the worst case. The average number of work is in $O(\frac{n}{2})$ which is $O(n)$ as well. In section 3.4 we will study several techniques to speed up `find` for positional lists.

Some readers may find it strange that `find` does a lot of work to search for a value which we already have at our fingertips when calling `find`: isn't it a bit

weird to execute the call (`find a-list a-key`) in order to ask `find` to search for something we already have (namely the object bound to the variable `a-key`)? In order to understand this, have a second look at the example in section 3.2.3. In the construction of the positional list, a comparator `entry-eq?` is passed. Upon closer inspection, this comparator does not compare *all* the fields of the `entry`'s. Instead, two `entry`'s are considered equal whenever their day and their month are equal. Remember from section 1.4.3 that we make a distinction between key fields and satellite fields. In our example, the day and the month are considered to be the key fields and the note is considered to be a satellite field. The example clearly shows that the comparator used in the positional list's constructor only compares key fields and ignores satellite fields. When calling `find` to search for an `entry` in the positional list, we use a key `entry` that only bears meaningful values for the key fields. When the corresponding data element is found in the positional list, we can use `peek` to read it from the positional list and subsequently access the satellite fields as well. In our example this is done using the accessors `day`, `month` and `note`.

Towards Performance Characteristics

Let us now summarize the performance characteristics for the ADT procedures the implementation of which already has been studied. The table shown in figure 3.4 summarizes what we have said so far. The sequential searching algorithm implementing `find` was shown to be in $O(n)$. The performance characteristics for `delete`, `add-before!` and `add-after!` depend on the performance characteristic of our six representation specific procedures. They are discussed for all four representations in sections 3.2.5 (vector implementation), 3.2.6 (single linked implementation), 3.2.7 (double linked implementation) and 3.2.8(another double linked implementation). Notice that we have put an asterisk in the table entry for `add-after!` in figure 3.4 . By taking a closer look at the implementation of `add-after!`, we notice that its performance characteristic is actually the performance characteristic of `attach-middle!` *except* when we omit the `pos` argument. This is an important remark for implementations that yield an $O(1)$ version for `attach-middle!` but an $O(n)$ version for `attach-last!`. Even though this puts `add-after!` strictly spoken in $O(n)$, we know that it will exhibit an $O(1)$ behaviour as long we don't omit the third argument! Notice that this knowledge is relied upon in order to obtain that the performance characteristics of `map`. By always using an explicit position argument in the call to `add-after!` (except for the very first time), we obtain that all procedures used in its body are in $O(1)$. Hence, `map` is in $O(n)$. `for-each` is clearly in $O(n)$ as it visits all positions exactly once.

In our discussion of the four positional list implementations, we will stick to the following order when presenting the various parts of the implementation.

Representation First we describe the representation. We show how enhanced lists or enhanced vectors are used in order to store the constituents of a positional list.

Procedure	Performance Characteristic
`from-scheme-list`	$O(n)$
`map`	$O(n)$
`for-each`	$O(n)$
`find`	$O(n)$
`delete!`	$O(max\left\{\begin{array}{c} f_{\texttt{first}} \\ f_{\texttt{has-next?}} \\ f_{\texttt{detach-first!}} \\ f_{\texttt{detach-last!}} \\ f_{\texttt{detach-middle!}} \end{array}\right\})$
`add-before!`	$O(max\left\{\begin{array}{c} f_{\texttt{first}} \\ f_{\texttt{attach-first!}} \\ f_{\texttt{attach-middle!}} \\ f_{\texttt{previous}} \end{array}\right\})$
`add-after!`	$O(max\left\{\begin{array}{c} f_{\texttt{attach-last!}^\star} \\ f_{\texttt{attach-middle}} \end{array}\right\})$

Figure 3.4: Performance Characteristics for Shared Procedures

Verification Then we describe the implementations of `length`, `empty?` and `full?`, i.e. the ADT operations that allow users to verify the size and boundaries of their positional lists.

Navigation Third we show the implementations for `first`, `last`, `next`, `previous`, `has-next?` and `has-previous?`. These operations allow user code to navigate through positional lists.

Manipulation Last, we present operations that can be used to manipulate positions and the elements associated with those positions. This includes two ADT operations `peek` and `update!` and the six private procedures `attach-first!`, `attach-middle!`, `attach-last!`, `detach-first!`, `detach-middle!` and `detach-last!`.

3.2.5 The Vectorial Implementation

In the vectorial implementation, the position abstraction `P` of the ADT is filled in by plain Scheme numbers. Hence `P` = `number`. In other words, a position is an index in the underlying vector. This is important to know when we find ourselves in the role of the *implementor* of the ADT. However, this should never be relied upon when *using* the ADT. It is not a part of the ADT specification. In the linked implementation of the ADT, positions are no longer indices in a vector but references to dotted pairs. Hence, the exact representation of positions may never be explicitly relied upon in the code that *uses* the ADT. Positions may only be manipulated through the abstract operations `first`, `last`, `has-next?`, `has-previous?`, `next` and `previous`.

Before we start our study of the vector implementation, we describe two procedures that are heavily relied upon. `storage-move-right` takes a vector and two indexes `i` and `j` such that `i<j`. It starts from `j` down to `i` and moves the entries of the vector one position to the right. This frees the `i`'th entry since that entry is stored at position `i+1` after executing the procedure. Similarly, `storage-move-left` takes a vector and two indexes `i` and `j` such that `i<j`. It starts from `i` up to `j` and moves every entry one position to the left. It stores the entry of position `j` in location `j-1` thereby freeing up the `j`'th position. Both procedures are clearly in $O(n)$ where n is the length of the input vector.

```
(define (storage-move-right vector i j)
  (define (iter idx)
    (vector-set! vector (+ idx 1) (vector-ref vector idx))
    (if (> idx i)
      (iter (- idx 1))))
  (iter j))
(define (storage-move-left vector i j)
  (define (iter idx)
    (vector-set! vector (- idx 1) (vector-ref vector idx))
    (if (< idx j)
      (iter (+ idx 1))))
  (iter i))
```

Representation

In the vector implementation of the `positional-list` ADT, we represent a positional list by an enhanced vector. We define a tag to be used by the enhanced vector representation as discussed in 3.1.3. Apart from the tag, the header of the enhanced vector contains a vector that stores the positional list's elements, a comparator procedure `==?` and a variable `first-free` which is a counter indicating the next free position in the vector. It is initialized to 0. By convention we store the elements of the positional list "in the leftmost positions of the vector". Hence `first-free` is the first location in the vector (considered from left to right) that does not contain any meaningful data.

Remember that the size of a vector needs to be known upon creation. In our implementation, the size is a predefined constant (namely 50). By adjusting the definition of `positional-list-size`, the implementation generates positional lists of different sizes. Notice that we make a distinction between the ADT's constructor `new` and the private constructor `make`. Remember from section 1.1 that the constructor's job is to reserve computer memory and to initialize the memory with meaningful values. These two roles are being taken care of separately by `new` and `make`: `make` creates the data structure in memory and `new` subsequently initialises it.

```
(define positional-list-tag 'vectorial-list)
(define positional-list-size 50)

(define (make first-free size ==?)
```

```
  (list positional-list-tag first-free (make-vector size) ==?))

(define (new ==?)
  (make 0 positional-list-size ==?))
```

Given this representation, the following five accessor and mutator procedures access the constituents of a positional list `plst`. These are only used in the implementation. None of them are exported to users of the `positional-list` ADT.

```
(define (storage plst)
  (caddr plst))
(define (storage! plst vector)
  (set-car! (cddr plst) vector))
(define (size plst)
  (cadr plst))
(define (size! plst s)
  (set-car! (cdr plst) s))
(define (equality plst)
  (cadddr plst))
```

Verification

Given these procedures that nail down the representation of the vectorial implementation, we can now proceed with the implementation of the ADT operations whose implementations were not included in the shared part presented in section 3.2.4. The implementations for `length`, `empty?` and `full?` are straightforward. They access the information straight from the representation:

```
(define (length plst)
  (size plst))

(define (empty? plst)
  (= 0 (size plst)))

(define (full? plst)
  (= (+ (size plst) 1)
     (vector-length (storage plst))))
```

Navigation

The following procedures implement the navigational operations that can be used to traverse positional lists. The first position is 0 and the last position is the number (minus one) stored in the header of the enhanced vector. As can be expected, the predicates on positions are simply implemented as tests that check whether or not a given position is the smallest possible position (i.e. 0) or the biggest possible position (i.e. the number store in in the header minus one). The next position of a position is the successor of the position. The previous position is its predecessor.

```
(define (first plst)
```

```
  (if (= 0 (size plst))
    (error "empty list (first)" plst)
    0))

(define (last plst)
  (if (= 0 (size plst))
    (error "empty list (last)" plst)
    (- (size plst) 1)))

(define (has-next? plst pos)
  (< (+ pos 1) (size plst)))

(define (has-previous? plst pos)
  (< 0 pos))

(define (next plst pos)
  (if (not (has-next? plst pos))
    (error "list has no next (next)" plst)
    (+ pos 1)))

(define (previous plst pos)
  (if (not (has-previous? plst pos))
    (error "list has no previous (previous)" plst)
    (- pos 1)))
```

Needless to say, all these procedures are in $O(1)$. The fact that all navigation through a list can be achieved with $O(1)$ procedures is one of the biggest advantages of using vectors to implement positional lists.

Manipulation

Finally, we study the procedures to manipulate positions and the data elements associated with them. This includes the ADT operations `peek` and `update!` as well as the 6 private procedures that are used to attach and detach positions in a positional list. Their implementation is shown below.

```
(define (peek plst pos)
  (if (> pos (size plst))
    (error "illegal position (peek)" plst)
    (vector-ref (storage plst) pos)))

(define (update! plst pos val)
  (if (> pos (size plst))
    (error "illegal position (update!)" plst)
    (vector-set! (storage plst) pos val))))
```

The `peek` and `update!` operations are trivial. They merely use the given position to index the vector in order to read or write the corresponding positional list entry. They are clearly in $O(1)$.

```
(define (attach-first! plst val)
  (attach-middle! plst val -1))

(define (attach-last! plst val)
  (define list-storage (storage plst))
  (define first-free (size plst))
  (vector-set! list-storage first-free val)
  (size! plst (+ first-free 1)))

(define (attach-middle! plst val pos)
  (define list-storage (storage plst))
  (define first-free (size plst))
  (storage-move-right list-storage (+ pos 1) first-free)
  (vector-set! list-storage (+ pos 1) val)
  (size! plst (+ first-free 1)))
```

`attach-last!` is simple. It stores the given value in the next free position of the vector. It is clearly in $O(1)$. `attach-middle!` moves the elements to the right by copying them using `storage-move-right`. Like this the vector entry at the given position is freed such that it can be used to store the new value. Clearly, `attach-middle!` is in $O(n)$. The same goes for `attach-first!` as it calls `attach-middle!` to move *all* elements of the vector one position to the right.

```
(define (detach-first! plst)
  (detach-middle! plst 0))

(define (detach-last! plst pos)
  (define first-free (size plst))
  (size! plst (- first-free 1)))

(define (detach-middle! plst pos)
  (define list-storage (storage plst))
  (define first-free (size plst))
  (storage-move-left list-storage (+ pos 1) (- first-free 1))
  (size! plst (- first-free 1)))
```

`detach-last!` is simple again. It simply "forgets" the last meaningful entry of the vector by decrementing the value of the first free position. It is in $O(1)$. `detach-middle!` does the opposite of `attach-middle!`: it copies all the elements to the right of the position one location to the left using `storage-move-left`. `detach-first!` calls `detach-middle!` and engenders the worst case since all elements are copied. Both procedures are clearly in $O(n)$.

Performance

Table in figure 3.5 summarizes the performance characteristics for the vector implementation of positional lists. The table is a completion of the table in figure 3.4.

Procedure	Performance Characteristic
`new`	$O(1)$
`from-scheme-list`	$O(n)$
`length`	$O(1)$
`full?`	$O(1)$
`empty?`	$O(1)$
`map`	$O(n)$
`for-each`	$O(n)$
`first`	$O(1)$
`last`	$O(1)$
`has-next?`	$O(1)$
`has-previous?`	$O(1)$
`next`	$O(1)$
`previous`	$O(1)$
`find`	$O(n)$
`update!`	$O(1)$
`delete!`	$O(n)$
`peek`	$O(1)$
`add-before!`	$O(n)$
`add-after!`	$O(n)$

Figure 3.5: Vector Implementation Performance Characteristics

3.2.6 The Single Linked Implementation

The second implementation of the `positional-list` ADT uses enhanced lists instead of enhanced vectors. The implementation uses dotted pairs and is called a *linked implementation* since it uses the "cdr pointers" to link up the positions of the list. Our abstract notion of positions defined in section 3.2.1 is therefore filled in by dotted pairs instead indices in a vector. Hence, `P` = `pair`. Again, this is important knowledge when *implementing* the `positional-list` ADT, but it should never be relied upon when *using* the ADT. The nature of positions is not specified by the `positional-list` ADT!

Figure 3.6 shows a typical linked positional list. It consists of three dotted pairs that constitute the header of the list. The other dotted pairs store data values. They are referred to as the *nodes* of the linked list. In the implementation presented here, these nodes are linked up in only one direction: every node stores a reference to its successor. We therefore refer to the implementation as a *single linked implementation*. In section 3.2.7 we present an alternative linked implementation in which every node stores a reference to its successor as well as a reference to its predecessor. This will be called a *double linked list*. The example shown in figure 3.6 is a single linked list with four nodes.

Representation

Let us start with the representational issues. Again, a distinction is made

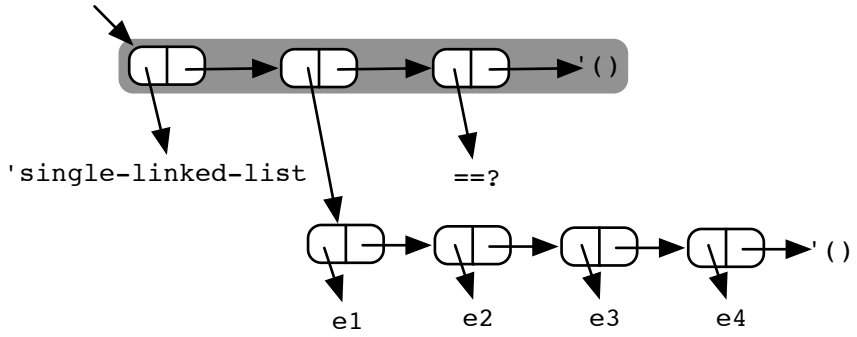

Figure 3.6: A typical linked positional list

between the constructor `new` and the unexported private procedure `make`. The latter merely allocates memory to store the positional list. The role of the former is to call the latter and provide additional arguments to properly initialize the newly created positional list. The procedures `head`, `head!` and `equal` shown below can be used to read and write the elements of the header of the positional list.

```
(define positional-list-tag 'single-linked-list)
(define (make head ==?)
  (list positional-list-tag head ==?))
(define (head plst)
  (cadr plst))
(define (head! plst node)
  (set-car! (cdr plst) node))
(define (equality plst)
  (caddr plst))

(define (new ==?)
  (make '() ==?))
```

Instead of relying directly on `cons`, `car` and `cdr` to create and manipulate the nodes of the list, we have built an additional `list-node` abstraction layer. The following procedures can be used to create a new node (given a value to be stored and a next node) and read and write the node's content. They make the code less dependent on a particular representation of nodes. E.g., if we were to decide to change our representation of nodes to vectors, then all we have to do is change the 5 procedures shown below.

```
(define (make-list-node val next)
  (cons val next))
(define (list-node-val node)
  (car node))
```

```
(define (list-node-val! node val)
  (set-car! node val))
(define (list-node-next node)
  (cdr node))
(define (list-node-next! node next)
  (set-cdr! node next))
```

Verification

The implementations of `empty?` and `full?` are not that interesting. In the linked implementation, `full?` always returns `#f` since the implementation can always create a new node to add to the list[3]. This is in contrast with the vector implementation which requires the storage capacity of the list to be fixed when calling the constructor. The `length` operation is interesting though. In order to determine the length of the linked list, we need a loop `length-iter` that traverses the nodes of the list in order to count them. In contrast to the vector implementation, this gives us an $O(n)$ implementation for `length`.

```
(define (length plst)
  (let length-iter
    ((curr (head plst))
     (size 0))
    (if (null? curr)
      size
      (length-iter (list-node-next curr) (+ size 1)))))

(define (full? plst)
  #f)

(define (empty? plst)
  (null? (head plst)))
```

Navigation

The operations `first`, `last`, `has-next?`, `has-previous?`, `next` and `previous` are used to navigate through a positional list using the abstract notion of positions. In our dotted pair representation of positions, the "cdr" stores a reference to the next position. This means that operations such as `next` and `has-next?` are readily implemented with an $O(1)$ performance characteristic. The same goes for `first` which merely accesses the first dotted pair stored in the header. `has-previous?` is an $O(1)$ operation as well since it only has to verify whether or not the given position is the first position. Unfortunately, the same cannot be said about the remaining operations. Since the nodes of the list do not store a reference to their previous position, the only way to access the previous position is by starting at the first position and iterating through the list until the node is reached of which the previous position is required. Hence, `previous` is

[3]Notice that this is not entirely true. Scheme implementations are always limited in the amount of memory they can use. In most implementations, creating a new node will cause the automatic garbage collector to clean up memory when no more nodes are available. If there are no "old" nodes that can be "thrown away", the Scheme system crashes.

an operation with performance characteristic in $O(n)$. Finally, the operation `last` is $O(n)$ as well. Since the header only stores a reference to the very first node, the only way to reach the final node is to traverse the entire list starting from the header. Hence, the operation has a performance characteristic which is $O(n)$, even in the best case. The Scheme procedures corresponding to the navigational operations follow:

```
(define (first plst)
  (if (null? (head plst))
    (error "list empty (first)" plst)
    (head plst)))

(define (last plst)
  (if (null? (head plst))
    (error "list empty (last)" plst))
  (let iter-to-last
    ((curr (head plst)))
    (if (null? (list-node-next curr))
      curr
      (iter-to-last (list-node-next curr)))))

(define (has-next? plst pos)
  (not (null? (list-node-next pos))))

(define (has-previous? plst pos)
  (not (eq? pos (head plst))))

(define (next plst pos)
  (if (not (has-next? plst pos))
    (error "list has no next (next)" plst)
    (list-node-next pos)))

(define (previous plst pos)
  (if (not (has-previous? plst pos))
    (error "list has no previous (previous)" plst))
  (let iter-to-previous
    ((curr (head plst)))
    (if (eq? (list-node-next curr) pos)
      curr
      (iter-to-previous (list-node-next curr)))))
```

Manipulation

Finally, let us have a look at the eight procedures used to manipulate data elements and the positions with which they are associated. Just as in the vector implementation, `peek` and `update!` are $O(1)$ operations. Given a position one merely has to access the "car" of the corresponding node in order to read or write the value it holds.

```
(define (update! plst pos val)
  (list-node-val! pos val)
  plst)

(define (peek plst pos)
  (list-node-val pos))
```

The implementation of the six accessors and mutators is more interesting. The implementation of `attach-first!` simply consists of inserting the new node between the header and the original first node. It is in $O(1)$. `attach-last!` is more complex. We cover some special cases such as the case when the newly added last node is the first node. This happens when adding a new node to an empty list. In the common case, we use the variable `last` to iterate towards the end of the list. This is needed to find the original last node in order to make it refer to the new last node. The operation is clearly $O(n)$. `attach-middle!`'s goal is to attach a new node right after the node that corresponds to its argument position. All we have to do is make the argument position point to the new node and make the new node point to the argument position's next node. Like that, the node is correctly inserted. The procedure is in $O(1)$.

```
(define (attach-first! plst val)
  (define first (head plst))
  (define node (make-list-node val first))
  (head! plst node))

(define (attach-last! plst val)
  (define first (head plst))
  (define node (make-list-node val '()))
  (if (null? first)
    (head! plst node) ; last is also first
    (list-node-next!
     (let iter-to-last
       ((curr first))
       (if (null? (list-node-next curr))
         curr
         (iter-to-last (list-node-next curr))))
     node)))

(define (attach-middle! plst val pos)
  (define next (list-node-next pos))
  (define node (make-list-node val next))
  (list-node-next! pos node))
```

Detaching nodes is only simple when detaching the very first node. All we have to do is make the header of the positional list refer to the node that follows the original first node such that this node now becomes the new first node. Hence, `detach-first!` is simple and in $O(1)$. `detach-middle!` and

`detach-last!` are much more complicated. The reason is that the node to be removed stores a reference to its next node but not to its previous node. The previous node is needed since we have to make sure it refers to the next node of the node to be removed. Therefore, both `detach-middle!` and `detach-last!` traverse the linked list from the first node up to the node to be removed. In the `detach-middle!` procedure we search for the previous node `prev`. In `detach-last!` we search for the node that is the previous node of the last node. This node is called the penultimate node. Hence the name of the variable `penu`. Whenever we detect the variable `last` to contain a node that has no next node, then we conclude that `penu` referes to the penultimate node. Otherwise, we continue the iteration: `last` becomes the penultimate node and `last`'s next node becomes the last node. Needless to say, both `detach-middle!` and `detach-last!` are in $O(n)$.

```
(define (detach-first! plst)
  (define first (head plst))
  (define second (list-node-next first))
  (head! plst second))

(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (let iter-to-previous
                 ((curr (head plst)))
                 (if (eq? pos (list-node-next curr))
                   curr
                   (iter-to-previous (list-node-next curr)))))
  (list-node-next! prev next))

(define (detach-last! plst pos)
  (define first (head plst))
  (if (null? (list-node-next first)) ; last is also first
    (head! plst '())
    (list-node-next!
     (let iter-to-penultimate
       ((penu first)
        (last (next plst first)))
       (if (not (has-next? plst last))
         penu
         (iter-to-penultimate last (next plst last))))
     '())))
```

Notice that `detach-last!` uses an iteration technique that uses two variables `penu` and `last` representing two consecutive positions in the list. The `last` variable is the actual node we are inspecting in each step of the iteration and the action taken in the last step of the iteration is performed on `penu`. `last` and `penu` systematically "follow each other". They are referred to as *chasing pointers*.

Performance

The performance characteristics for the single linked implementation of the positional list ADT are summarized in figure 3.7. Again, the table should be understood as a completion of the table in figure 3.4. Remember from figure 3.4 that the $O(n)$ behaviour of `attach-last!` only shows up if we call `add-after!` without a position parameter. Hence, `add-after!` is in $O(1)$ provided that we call it on a concrete position.

Procedure	Performance Characteristic
`new`	$O(1)$
`from-scheme-list`	$O(n)$
`length`	$O(n)$
`full?`	$O(1)$
`empty?`	$O(1)$
`map`	$O(n)$
`for-each`	$O(n)$
`first`	$O(1)$
`last`	$O(n)$
`has-next?`	$O(1)$
`has-previous?`	$O(1)$
`next`	$O(1)$
`previous`	$O(n)$
`find`	$O(n)$
`update!`	$O(1)$
`delete!`	$O(n)$
`peek`	$O(1)$
`add-before!`	$O(n)$
`add-after!`	$O(n)$ (or$\star$ $O(1)$)

Figure 3.7: Single Linked Implementation Performance Characteristics

3.2.7 A Double Linked Implementation

If execution speed is crucial to a program that uses positional lists, then both tables shown in figure 3.7 and figure 3.5 are unsatisfactory since they display operations with performance characteristics some of which are pretty disastrous. The situation is especially problematic when that application frequently adds and deletes elements to and from the positional list. For small lists this is not a problem given the speed of modern computers. However, for really large lists with thousands or millions of elements, the procedures soon get too slow, even on today's hardware.

In this and the next section, we present two linked list implementations that perform significantly better than the vector implementation and the single linked list implementation. However, as is often the case in computer science, an

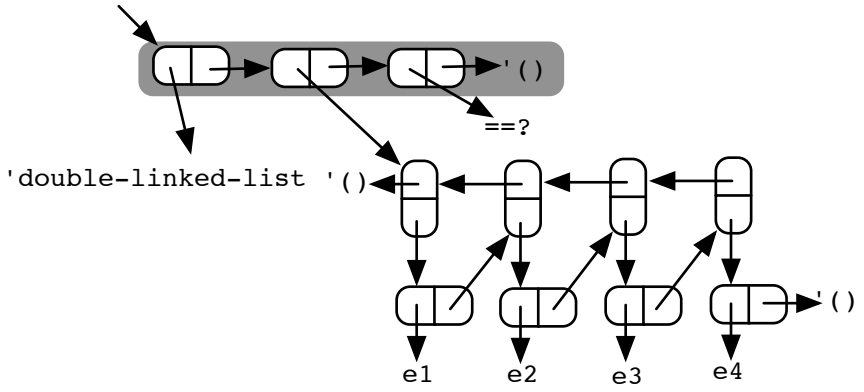

Figure 3.8: A double linked positional list

improvement in speed is to be payed for with additional memory consumption. We distinguish two improvements:

- First, a number of operations can be sped up by allowing linked list nodes to store an additional reference to the node corresponding to its previous position. The resulting data structure is called a *double linked list* (as opposed to the *single linked list* presented earlier) and the extra reference is known as a *back pointer*. A double linked version of the positional list depicted in figure 3.6 is depicted in figure 3.8. As we can see from this figure, a double linked list node consists of two dotted pairs that keep a reference to the value stored, a reference to the next node and a reference to the previous one. The code for this is explained in this section.

- Second, a number of operations can be sped up by storing additional information in the header of the positional list. E.g., instead of *computing* the length of the list in the implementation of the `length` operation, we can *store* the length of the list in the header of the list. This changes the $O(n)$ performance characteristic of `length` in figure 3.7 into $O(1)$. A similar improvement consist of storing a reference to the last node of the positional list in the header as well since the cause for one of the $O(n)$ operations in figure 3.7 (namely `last`) lies in the fact that a reference to the last node of the positional list is required. This implementation is given in section 3.2.8.

It will come as no surprise that a lot of the features of the single linked implementation are inherited by the double linked implementation. In what follows, we merely describe the procedures that differ from the code presented in the previous section. We stick to the order used to present the various parts of the implementation. So let's start with the representational issues.

Representation

In contrast to the single linked implementation, double linked list nodes also store a reference to the previous position of the node. The following code shows the constructor, the accessors and mutators for the `list-node` abstraction.

```
(define (make-list-node val prev next)
  (cons prev (cons val next)))
(define (list-node-val node)
  (cadr node))
(define (list-node-val! node val)
  (set-car! (cdr node) val))
(define (list-node-next node)
  (cddr node))
(define (list-node-next! node next)
  (set-cdr! (cdr node) next))
(define (list-node-prev node)
  (car node))
(define (list-node-prev! node prev)
  (set-car! node prev))
```

The following code excerpt shows the implementation of the constructor and the procedures to manage the constituents of the header:

```
(define positional-list-tag 'double-linked-list)

(define (make head ==?)
  (list positional-list-tag head ==?))
(define (head plst)
  (cadr plst))
(define (head! plst node)
  (set-car! (cdr plst) node))
(define (equality plst)
  (caddr plst))

(define (new ==?)
  (make '() ==?))
```

Verification

The implementation of the verification procedures `full?`, `empty?` and `length` are exactly identical to the implementations we presented in the single linked positional list implementation. We do not repeat them here.

Navigation

As can be expected, most of the navigation operations have an identical implementation as well. The only exception is the `previous` operation. In the single linked implementation, this operation was implemented by iterating from the first node to the node of which the previous node is required. In the double linked implementation, all we have to do is follow the back pointer:

```
(define (previous plst pos)
```

```
  (if (not (has-previous? plst pos))
    (error "list has no previous (previous)" plst)
    (list-node-prev pos)))
```

Manipulation

The procedures for manipulating positions and the data elements they contain are shown below. `update!` and `peek` are omitted since they didn't change. Most of the other code is fairly trivial. The additional complexity comes from the fact that nodes now have two pointers. Every time we insert a node, we have to make sure to make its next and its previous point to the correct nodes. On top of that, we have to make sure that the previous of the next node points to the inserted node (in case the node has a next node). Similarly, we have to make sure that the next node of the previous node corresponds to the inserted node (in case the inserted node has a previous node). Notice that `attach-last!` still exhibits $O(n)$ behaviour since we don't store an explicit reference to the last node in the position list's header.

```
(define (attach-first! plst val)
  (define head (first plst))
  (define node (make-list-node val '() head))
  (if (not (null? head))
    (list-node-prev! head node))
  (head! plst node))

(define (attach-last! plst val)
  (define first (head plst))
  (define last
    (if (null? first)
      first
      (let iter-to-last
        ((curr first))
        (if (not (has-next? plst curr))
          curr
          (iter-to-last (next plst curr))))))
  (define node (make-list-node val last '()))
  (if (null? first) ; last will be first as well
    (head! plst node)
    (list-node-next! last node)))

(define (attach-middle! plst val pos)
  (define next (list-node-next pos))
  (define node (make-list-node val pos next))
  (list-node-next! pos node)
  (if (not (null? next))
    (list-node-prev! next node)))
```

Detaching nodes is very similar to detaching nodes in the single linked implementation. The only notable difference is the implementation of `detach-middle!`.

In the single linked implementation, this was an $O(n)$ operation since we had to iterate over the list in order to find the previous node of the detached node. In the double linked implementation, the previous node is found by simply following the back pointer. Hence, the operation is in $O(1)$ now.

```
(define (detach-first! plst)
  (define first (head plst))
  (define next (list-node-next first))
  (head! plst next)
  (if (not (null? next))
    (list-node-prev! next '())))

(define (detach-last! plst pos)
  (define first (head plst))
  (if (null? (list-node-next first)) ; last is the first
    (head! plst '())
    (list-node-next! (list-node-prev pos) '())))

(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (list-node-prev pos))
  (list-node-prev! next prev)
  (list-node-next! prev next))
```

Performance

The performance characteristics for the double linked implementation of the positional list ADT are summarized in figure 3.9. Again, the table completes the table in figure 3.4. Here too, we note that `add-after!` only exhibits $O(n)$ behaviour when omitting the position argument. When called on a concrete argument position, `add-after!` is in $O(1)$. The most important improvement of the double linked implementation w.r.t. the single linked implementation is that the implementation of the operations `previous`, `delete!` and `add-before!` are in $O(1)$ whereas they used to be in $O(n)$ for the single linked implementation. This is because it is precisely these operations that require the previous position of a position. In the single linked implementation this required an iteration starting at the first position. In the double linked implementation, we just have to follow the back pointer of a node.

3.2.8 An Augmented Double Linked Implementation

The fourth and final implementation of the `position-list` ADT tries to speed up some of the implementations shown in figure 3.9. We observe that `last` and `add-after!` still exhibit an $O(n)$ behaviour. The latter is due to the exceptional case that occurs when we omit the position argument. `add-after!` has to iterate in order to obtain the last position in that case. The implementation we are about to present stores an additional reference in the header that explicitly refers to the last node of the list. Moreover, `length` is turned into an $O(1)$ operation

Procedure	Performance Characteristic
`new`	$O(1)$
`from-scheme-list`	$O(n)$
`length`	$O(n)$
`full?`	$O(1)$
`empty?`	$O(1)$
`map`	$O(n)$
`for-each`	$O(n)$
`first`	$O(1)$
`last`	$O(n)$
`has-next?`	$O(1)$
`has-previous?`	$O(1)$
`next`	$O(1)$
`previous`	$O(1)$
`find`	$O(n)$
`update!`	$O(1)$
`delete!`	$O(1)$
`peek`	$O(1)$
`add-before!`	$O(1)$
`add-after!`	$O(n)$ (or$\star$ $O(1)$)

Figure 3.9: Double Linked Implementation Performance Characteristics

by simply storing the length of the list in the header instead of computing it every time again.

Representation

The representation of the lists is nearly identical to the representation of double linked lists discussed above. The only thing that changes is the fact that we now store more information in the header. Accessors (`size` and `last`) and mutators (`size!` and `last!`) have been added to manipulate these extra bits of information:

```
(define positional-list-tag 'double-linked-augmented-list)
(define (make head tail size ==?)
  (list positional-list-tag head tail size ==?))
(define (head plst)
  (cadr plst))
(define (head! plst node)
  (set-car! (cdr plst) node))
(define (tail plst)
  (caddr plst))
(define (tail! plst node)
  (set-car! (cddr plst) node))
(define (size plst)
```

```
  (cadddr plst))
(define (size! plst s)
  (set-car! (cdddr plst) s))
(define (equality plst)
  (caddr (cddr plst)))

(define (new ==?)
  (make '() '() 0 ==?))
```

Verification

As can be expected, the implementation of `full?` and `empty?` does not change. The implementation of `length` does! Whereas both previous linked implementations compute the length of a list by traversing it, this implementation simply returns the length that is stored in the header. Hence it is in $O(1)$.

```
(define (length plst)
  (size plst))
```

Navigation

The only navigational procedure that changes is `last`. Both linked implementations discussed before get a reference to the last position in the list by means of an iterative process. This implementation explicitly stores the last position in the header. All we have to do is to read it from the header.

```
(define (last plst)
  (if (null? (tail plst))
    (error "list empty (last)" plst)
    (tail plst)))
```

Manipulation

As can be expected, the implementations of `attach-last!` and `detach-last!` are radically different from the previous implementations. Since we maintain an explicit reference to the last node of the list, we no longer need iterations to get to that node. Notice however, that the rest of the manipulation code is not an exact copy of the code presented for the original double linked implementation. Storing an explicit reference and storing the length of the list in the header has its price in the sense that all procedures that possibly affect these values need to take them into account. That is the reason why we have to reimplement all other manipulation procedures as well. Every time a node is added, the size has to be incremented and every time a node is removed, the size has to be decremented accordingly. Furthermore every operation that potentially modifies the last node, has to ensure that the header refers to the correct last node at all times.

```
(define (attach-first! plst val)
  (define first (head plst))
  (define node (make-list-node val '() first))
  (head! plst node)
  (if (null? (tail plst)) ; last is first
```

```
    (tail! plst node)
    (list-node-prev! first node))
  (size! plst (+ 1 (size plst))))

(define (attach-last! plst val)
  (define last (tail plst))
  (define node (make-list-node val last '()))
  (tail! plst node)
  (if (null? (head plst)) ; first is last
    (head! plst node)
    (list-node-next! last node))
  (size! plst (+ 1 (size plst))))

(define (attach-middle! plst val pos)
  (define next (list-node-next pos))
  (define node (make-list-node val pos next))
  (list-node-next! pos node)
  (if (eq? pos (tail plst))
    (tail! plst node)
    (list-node-prev! next node))
  (size! plst (+ 1 (size plst))))

(define (detach-first! plst)
  (define first (head plst))
  (define second (list-node-next first))
  (if (null? second)                ; first is the only one
    (tail! plst second))
  (head! plst second)
  (size! plst (- (size plst) 1)))

(define (detach-last! plst pos)
  (define first (head plst))
  (define second (list-node-next first))
  (define last (tail plst))
  (define penultimate (list-node-prev last))
  (if (null? second)                ; last is the first?
    (head! plst second)
    (list-node-next! penultimate '()))
  (tail! plst penultimate)
  (size! plst (- (size plst) 1)))

(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (list-node-prev pos))
  (list-node-next! prev next)
  (list-node-prev! next prev)
  (size! plst (- (size plst) 1)))
```

3.2.9 Comparing List Implementations

Let us now compare the four implementations of the `position-list` ADT.

Remember that one of the drawbacks of using vectors is the fact that they require us to make a correct estimate about the expected capacity of a position list at the time of creation. Trying to add elements to a positional list which is full has to be taken care of, either by raising an error or by extending the vector. Unfortunately, the latter solution requires us to copy the elements of the old vector into the new vector which is a costly $O(n)$ operation. In the linked implementation, this problem does not occur. Therefore, the linked implementation outperforms the vector implementation when absolutely no meaningful estimate can be made upfront about the potential size of a positional list.

Figure 3.10 shows a performance characteristic table that combines the contents of the tables we have seen in figure 3.5, figure 3.7 and figure 3.9. The table shows that there is only limited advantage in going from a vector implementation to a single linked implementation when speed is important. If a linked implementation is needed, then a double linked implementation pays off most when it comes to runtime efficiency. The real benefit of using double linked implementations lies in the fact that addition and deletion of values gets much faster. In programs that use relatively stable lists, it might be a good option to go for single linked lists anyhow: apart from the addition and deletion procedures and the procedure to find the previous position), there is not a lot of difference between the single linked and the double linked implementations.

The table in figure 3.10 basically shows us that we can "buy time with space". By storing extra information in the data structure, we can speed up most of the operations significantly. But how much space is needed? Since a single linked list explicitly needs to store all the next pointers, a positional list of n elements typically requires $2.n$ references in memory: n to store the actual elements and n to store the next pointers. Similarly, a double linked list requires $4.n$ memory locations to store a list of n elements. This is because every data value needs two dotted pairs. If we were to represent double linked list nodes by vectors of size 3, then this consumption can be reduced to $3.n$. For applications that are to run on large modern computers, $3.n$ or $4.n$ is not a problem at all as memory is quite cheap these days. For relatively small memories, the cost of storing linked lists can be significant. For example, embedded systems like household equipment (e.g. a microwave oven or a washing machine) typically have to deal with limited amounts of memory. In such cases, the vector implementation outperforms the linked ones.

Having a final look at this table, we still observe a performance characteristic of $O(n)$ for the `find` operation. In section 3.4 we will discuss several techniques to speed up the implementation for this operation as well. In the best case, we will obtain a performance characteristic of $O(log(n))$. But, as always, there is a price to pay elsewhere. Either more memory is needed to store additional pointers or a different organisation of the data structure is need (e.g. keep it sorted) which engenders slower implementations for the operations that add data to the data structure.

Operation	Vector	Linked	Double Linked 1	Double Linked 2
`new`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`from-scheme-list`	$O(n)$	$O(1)$	$O(n)$	$O(n)$
`length`	$O(1)$	$O(n)$	$O(n)$	$O(1)$
`full?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`empty?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`map`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`for-each`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`first`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`last`	$O(1)$	$O(n)$	$O(n)$	$O(1)$
`has-next?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`has-previous?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`next`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`previous`	$O(1)$	$O(n)$	$O(1)$	$O(1)$
`find`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`update!`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`delete!`	$O(n)$	$O(n)$	$O(1)$	$O(1)$
`peek`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`add-before!`	$O(n)$	$O(n)$	$O(1)$	$O(1)$
`add-after!`	$O(n)$	$O(n)$	$O(n)$	$O(1)$

Figure 3.10: Comparative Position List Performance Characteristics

3.3 Variations on Positional Lists

The implementation strategies studied in the previous sections not only work for positional lists. They are general implementation techniques that can be applied to implement other list ADTs as well. In this section we present two alternative list ADTs — `list-with-current` and `ranked-list`. These ADTs solve a number of conceptual problems that are inherent to positional lists. Hence, both `list-with-current` and `ranked-list` can be considered as the result of an exercise in linear ADT design instead of a contribution to the study of linear data structure implementation techniques.

3.3.1 The Problem

We have defined a linear data structure in section 3.2.1 as a collection of data elements that are associated with positions. The positions are organised in such a way that every position (except for the first and the last one) has a next position and a previous position. We have tried to be as abstract as possible in our conception of positions. In the four concrete implementations of the positional list ADT, positions have been represented both by numeric vector indices as well by dotted pairs that correspond to nodes.

Let us pick up our example of section 3.2.3 again. Remember that our

professor has created a todo-list containing five entries: a lecture on strings, the preparation of the lecture on linearity, a rest period, a lecture on linearity and a lecture on sorting. In order to print a schedule for his students, our professor creates a tiny Scheme list containing the positions of all three lectures:

```
(define lectures (list (find todo-list (make-entry 5 10 '()))
                       (find todo-list (make-entry 12 10 '()))
                       (find todo-list (make-entry 19 10 '()))))
```

At that point, our professor receives a call by some of his friends asking him to join them on a night out. They agree to meet on 11 October after his rest period. Hence, our professor adds the following entry to his todo-list.

```
(define rest (find todo-list (make-entry 9 10 '())))
(add-after! todo-list (make-entry 11 10 "Go out with friends") rest)
```

Our professor continues with his work and decides to print out the `lectures` list in order to send the schedule to his students. Since the lectures are contained in an naked Scheme list, he simply uses `map` to print out all entries that sit in the positional list in those positions that are contained in the `lectures` list:

```
(map (lambda (pos)
       (display (note (peek todo-list pos)))
       (newline))
     lectures)
```

Depending on the concrete implementation we use, this code gives rise to some serious problems. In the linked implementation, there is no problem. Even though the positional list has changed by adding the entry for the night out, the values contained by the `lectures` list are still referring to the correct positions in the positional list. However, in the vector implementation we get the following strange result on the screen:

```
Give Lecture on Strings
Go out with friends
Give Lecture on Linearity
```

What has happened? The problem is that the `lectures` list contains positions that refer to the positional list. In the vector implementation, these positions are mere numbers. By adding the entry for the night out, entries of the positional list are moved about and get a new vector index inside the positional list. Hence, old vector indices that reside in other data structures get a new meaning!

The essence of the problem is that, in the *specification* of our positional list ADT, positions indicate *relative positions* in a particular list at a particular time. Positions are defined in terms of their neighbours. However, in a concrete implementation, these positions are implemented by a concrete reference in Scheme that directly refers to some pair or index in an array. Given the fact that the list evolves over time, "externalised" conceptually relative positions are no longer synchronized with the *absolute* technical ones that reside in the

positional list. The reason is that the externalised positions do not evolve along when the list evolves. In order to mend the problem, it is necessary to render the positions *relative* to the representation of the list at *all* times. There are several ways to do this:

- The first way to solve the problem consists of making the implementation of the ADT's operations smarter. An operation such as `peek` would take an "externalised" position and a list. Instead of just peeking at the provided position, it would first update that position in order to synchronize it with the most recent version of the list. This solution, although simple in words, is difficult to realize technically since it requires the ADT implementation to remember (and thus store) *all* changes a positional list has undergone in order to be able to "translate" any given position (obtained in the past) to the current version of the list. This clearly consumes a lot of memory and the translation would take additional time to compute. We therefore discard this option.

- The second option is not to solve the problem, but rather to turn it into a feature of the linear data structure. The main idea consist of putting the entire responsibility with the user of the ADT. In this option, *all* positions are *by definition* relative positions, i.e. they are positions that refer to a list element at a *single* moment in the time. Manipulating the list can change the list in such ways that the semantics of a whole bunch of positions changes in one strike. Lists that have this property will be called *ranked lists*. They are the topic of section 3.3.3.

- The third way to solve our problem is to prevent the problem from happening by *internalising* positions in the list data structure. The idea is to store one position inside the list itself and then make all operations work relative to that special position. We shall henceforth call this position the *current position*. The result of this redesign is a new list ADT in which the notion of "externalised positions" has been replaced by "a current position". All list operations are expressed relatively to that current position. This ADT is presented in the next section.

.

3.3.2 Relative Positions: Lists with a Current

The `list-with-current` ADT is shown below. The ADT has many similarities with the `position-list` ADT presented in section 3.2.2. Note that the abstract notion of a position no longer appears in the definition of the ADT. Hence, the ADT is not parametetrized by a data type `P` of positions. Furthermore, there are no operations (such as `first`) that "externalise" positions. All navigation through the list is done using a "current position" that is maintained inside every list and that is hidden for the user of the ADT. There are several operations to manipulate that current position. For the sake of simplicity, we have

omitted "non-essential" operations such as `map` and `for-each` from the ADT. Interestingly, `find!` becomes a destructive operation that makes the current of a list refer to the value found.

```
1  ADT list−with−current < V >
2
3  new
4     ( ( V V → boolean ) → list−with−current < V > )
5  from−scheme−list
6     ( pair ( V V → boolean ) → list−with−current < V > )
7  list−with−current?
8     ( any → boolean )
9  length
10    ( list−with−current < V > → number )
11 full?
12    ( list−with−current < V > → boolean )
13 empty?
14    ( list−with−current < V > → boolean )
15 set−current−to−first!
16    ( list−with−current < V > → list−with−current < V > )
17 set−current−to−last!
18    ( list−with−current < V > → list−with−current < V > )
19 current−has−next?
20    ( list−with−current < V > → boolean )
21 current−has−previous?
22    ( list−with−current < V > → boolean )
23 set−current−to−next!
24    ( list−with−current < V > → list−with−current < V > )
25 set−current−to−previous!
26    ( list−with−current < V > → list−with−current < V > )
27 has−current?
28    ( list−with−current < V > → boolean )
29 find!
30    ( list−with−current < V > V → list−with−current < V > )
31 update!
32    ( list−with−current < V > V → list−with−current < V > )
33 peek
34    ( list−with−current < V > → V )
35 delete!
36    ( list−with−current < V > → list−with−current < V > )
37 add−before!
38    ( list−with−current < V > V → list−with−current < V > )
39 add−after!
40    ( list−with−current < V > V → list−with−current < V > )
```

Not every operation leaves the data structure with a meaningful current position. For example, when launching a search in the list using `find!` it might be the case that the target key does not match any of the elements in the list. In that case the current has no meaning. It is said to be *invalidated*. Similarly the current has no meaning when a list is empty (e.g. right after creation) or after deleting the element associated with the current position (using `delete!`). To be able to deal with this, the ADT features an operation `has-current?` which can be used to test whether or not the current is referring to a meaningful data value in the list at a certain moment in time. Apart from this difference, the ADT is very similar to the positional list ADT. However, it does not suffer from the problem with externalised positions.

3.3.3 Relative Positions: Ranked Lists

The `list-with-current` ADT of section 3.3.2 basically consists of avoiding the problem described in section 3.3.1 by no longer exposing positions from positional lists and by having all operations operate on a single current position that remains encapsulated in the list. A second solution to the problem consists of facing the problem in its entirety by deliberately stating in the ADT specification that positions are never to be taken meaningful after having been exposed by the list. In this design, the ADT prescribes that the notion of a position is *never* tightly coupled to any particular data element sitting in the list. A position then always refers to a position in a certain list at *this* particular moment in time. Such positions are called *ranks*.

In a linear data structure that contains n data elements, every data element stored in the data structure is associated with a *rank*. One of the elements in the data structure has rank 0. The rank for all other elements in the data structure is the number of data elements that *precede* that data element, i.e. the number of elements that have a smaller rank. It is important to understand that a rank of an element is always defined relative to a particular state of the data structure at a certain moment in time. An element with rank 2 has rank 1 from the moment that an element with rank 1 is removed. Similarly, inserting an element at the beginning of the data structure increases the rank of all the other elements by 1. Hence, the notion of ranks fully exploits the idea of relative positions as already explained in section 3.3.1. No data element has a position that can be used to address the data element outside the list. Ranks only make sense for the elements residing *in* the list. The list variation is called a *ranked list* and the ADT that defines it is the `ranked-list` ADT shown below.

```
1 ADT ranked−list < V >
2
3 new
4   ( ( V V → boolean) → ranked−list < V > )
5 from−scheme−list
6   ( pair ( V V → boolean ) → ranked−list < V > )
7 ranked−list?
```

```
8    ( any → boolean )
9  length
10   ( ranked−list < V > → number )
11 full?
12   ( ranked−list < V > → boolean )
13 empty?
14   ( ranked−list < V > → boolean )
15 find
16   ( ranked−list < V > V → number )
17 peek−at−rank
18   ( ranked−list < V > number → V )
19 update−at−rank!
20   ( ranked−list < V > number V → ranked−list < V > )
21 delete−at−rank!
22   ( ranked−list < V > number → ranked−list < V > )
23 add−at−rank!
24   ( ranked−list < V > V . number → ranked−list < V > )
```

In the ADT, we have used Scheme's `number` to represent ranks. Most operations look familiar. `find` searches for a key and returns its rank. `update-at-rank!`, `delete-at-rank!` and `add-at-rank!` are all parametrised with a `number` which is the rank of the element on which the operation is expected to operate. As explained, adding an element using `add-at-rank!` means that the rank of the elements which have a higher rank increases by one. Notice that the `add-at-rank!` operation takes the rank as an optional parameter. When omitting the rank, the element is simply added to the end of the list. In other words, omitting the rank corresponds to adding an element at rank $+\infty$.

Notice the difference between ranked lists and vectors. In the case of vectors, the operation `vector-set!` updates the data element sitting in a given index. In the case of ranked lists, the operation `update-at-rank!` has exactly the same effect. However, `add-at-rank!` shifts all elements to the right. Vectors do not feature such an operation. This should be a hint on the expected performance characteristic for our four possible implementation strategies for ranked lists (vectors, single linked lists, double linked lists and "fast" doubly linked lists). We leave them as an exercise for the reader.

3.4 Searching in Linear Data Structures

Let us refer back to the table in figure 3.10. At first sight, the fast double linked list implementation — the performance characteristics of which are listed in the fourth column — seems to be the fastest implementation we can obtain. All operations are in $O(1)$ except for the operations that have to process the entire list like `map`. They are in $O(n)$ "by definition". One might think that the same is true for `find`: since we have to compare the key with the data elements residing in the data structure, we have to traverse the entire data structure. Hence,

$O(n)$ really seems the best we can do. This is true if our `find` is solely based on comparing elements that are linearly stored without any further organization. In sections 3.4.3 and 3.4.4, we show that cleverly organising a list's elements can speed up things considerably. Indeed, just keeping the elements in the list *sorted* will allow us to speed up `find` up to the level of $O(log(n))$ for some implementations. Lists for which the data elements are always sorted are called *sorted lists*.

Before we move on to the presentation of sorted lists, we discuss a less inventive — yet useful — technique to speed up our $O(n)$ algorithm for `find`. It is the topic of section 3.4.2.

3.4.1 Sequential Searching

The implementation of `find` presented in section 3.2.4 is known as the *sequential searching algorithm*. It traverses the entire list sequentially by starting with the first element of the list and by ending at the last element of the list, possibly to find out that the key being searched for is not even in the list. Hence, the worst-case performance characteristic for the linear searching algorithm is $O(n)$.

3.4.2 Sentinel Searching

Sequential searching can be improved a lot by applying a technique that is known as *sentinel search*. Looking back at the body of `find` presented in section 3.2.4, we observe a conditional expression that has *two* stop conditions: `find` ends its recursion whenever the element is found *or* whenever the end of the list is reached.

We can avoid the second test by making sure that the key is *guaranteed* to be contained in the list. Technically, this is achieved by adding the key to the positional list at the last position of the list. After having searched the list, all we have to do is to check whether the element found is the one sitting at the last position. If this is not the case, then we found the *actual* element sitting in the list. In the other case it means that we ran off the end of the original list and that we have found the element that was just added. Needless to say, we should not forget to remove the key from the list again after having performed the search. The temporarily added data element is called *a sentinel*. Hence the name of the algorithm.

Surely, the sentinel search algorithm is still $O(n)$, but the resulting code is "k times as fast" (for some constant k) as the naive sequential searching algorithm since only one test has to be executed in each step of the iteration. The algorithm is presented below.

```
(define (find plst key)
  (if (empty? plst)
    #f
    (let
        ((==? (equality plst)))
```

```
(attach-last! plst key)
(let*
    ((pos (let search-sentinel
            ((curr (first plst)))
            (if (==? (peek plst curr) key)
              curr
              (search-sentinel (next plst curr)))))
     (res (if (has-next? plst pos)
            pos
            #f)))
  (detach-last! plst (last plst))
  res)))))
```

The algorithm starts by adding the search key to the end of the list using `attach-last!`. In the `search-sentinel` loop, we traverse the list until the element is found. Since we just added the element to the rear, we are guaranteed to find the element. All we have to do is check whether or not the key found was the added one, and finish by removing the added element again.

As explained above, the big advantage of sentinel search is that we save a test in every step of the iteration. This advantage has its price though: the `attach-last!` and `detach-last!` operations have to be in $O(1)$. In the vector implementation this is indeed the case. In linked implementations, sentinels only make sense if the list's header stores an explicit reference to the last node of the list. All other implementations need to traverse the entire list to add the sentinel before the search and to remove the sentinel again after having searched the list. Clearly this cost is higher than the speedup we obtain from the avoided test. Notice as well that the vector implementation also has to keep in mind never to fill the vector entirely: whenever a vector of n entries is allocated in computer memory, the list it represents has to be considered full once it contains $n - 1$ elements. The last entry has to be reserved to store a sentinel when a `find` is launched on a full list.

3.4.3 Sorted Lists

Without additional organisation of the list, the only way to guarantee a correct answer from `find` is to keep on traversing the entire list as long as the key is not found. In this section, we start our study of techniques that improve the efficiency of `find` by imposing additional structure on the elements sitting in the data structure. In future chapters this will have such a profound effect on the organisation of the data elements that the resulting data structure is no longer linear. Here we maintain linearity.

The idea of imposing additional organisation on a linear data structure consists of making sure that the elements of the linear data structure are *always* stored in sorted order. Such lists are known as a *sorted lists* and their specification results in a new ADT called `sorted-list`. Because the elements of the list are always stored in sorted order, `find` gets more efficient because we can

use the order of the elements to stop the search process earlier. Indeed, given a sequential search, we know for sure that the key won't show up anymore in the search process as soon as the key is "smaller" than the element in the list being visited, since all elements that are visited from that point onwards will turn out to be greater than the key[4].

Sorted lists are not just an implementation technique for the three linear ADTs that we have seen earlier. The reason is that we have to give the user of the ADT *less* control on how the elements of the ADT are to be inserted and updated. If not, the property of the elements in the list being sorted might be violated. The result is a new linear ADT with less operations than the three ADTs discussed earlier. The `sorted-list` ADT is presented below:

```
1  sorted−list < V >
2
3  new
4      ( ( V V → boolean)
5        ( V V → boolean) → sorted−list < V > )
6  from−scheme−list
7      ( pair
8        ( V V → boolean)
9        ( V V → boolean) → sorted−list < V >) )
10 sorted−list?
11     ( value → boolean )
12 length
13     ( sorted−list < V > → number )
14 empty?
15     ( sorted−list <V> → boolean )
16 full?
17     (sorted−list <V> → boolean )
18 find!
19     ( sorted−list < V > V → sorted−list < V >} )
20 delete!
21     ( sorted−list < V > → sorted−list < V > )
22 peek
23     ( sorted−list < V > → V )
24 add!
25     ( sorted−list < V > V → sorted−list < V > )
26 set−current−to−first!
27     ( sorted−list < V > → sorted−list < V > )
28 set−current−to−next!
29     ( sorted−list < V > → sorted−list < V > )
30 has−current?
```

[4]In the text, we assume an ordering from "small" to "big". We could easily inverse all terminology by ordering the elements from "big" down to "small". The notions of "big" and "small" are defined by the nature of the comparator used during construction of the sorted list.

31 (**sorted−list** < V > → **boolean**)
32 current−has−next?
33 (**sorted−list** < V > → **boolean**)

For reasons of brevity, we did not include all operations of the previously discussed linear ADTs. Generic operations like `map` have been omitted and the number of navigational operations that have to do with "the" current have been kept to a minimum: the ADT specifies the presence of a current but it can only be used to traverse the list in ascending order.

As explained above, less control has to be given to the user of the ADT on how the list is constructed. The reason for this is that the code which uses the ADT is no longer allowed to affect the organisational structure of the list for this might violate the idea of the elements being sorted at all times. Guaranteeing the fact that elements are always stored in sorted order is the responsibility of the list. Therefore, operations like `add-before!`, `add-after!` and `update!` have been removed from the ADT. The responsibility is now put entirely with the `add!` operation. It takes a sorted list and a value to be added to the list. It is the responsibility of `add!` to insert the element in the correct position in the data structure in order to preserve the property that elements are always stored in sorted order. In order to be able to do this, `add!` needs a procedure that determines the *order* of the elements it has to store. This is the reason why `new` accepts two procedures of type (`V V` → `boolean`). The first procedure is an operator `<<?` that is used by `add!` to correctly order the elements in the list. The second procedure is the classical comparator `==?` that we have been using throughout the rest of the chapter.

In what follows, we discuss a vectorial implementation. The linked implementations are left as a programming exercise to the reader. We invite the reader to compare various implementations of the `sorted-list` ADT from the point of view of their performance characteristics and the representation of the data structures. The vectorial representation is presented below. The representation resembles the representation of vectorial positional list (presented in section 3.2.5). A sorted list is created using one of the constructors `new` and `from-scheme-list`. The constructor `new` is implemented in two phases again: the private procedure `make` takes care of the memory allocation and `new` takes care of the proper initialisation of the newly created sorted list. `sorted-list?` checks whether or not a Scheme value is a sorted list. Apart from these ADT procedures, we list a number of private accessors that are used by the rest of the implementation.

```
(define sorted-list-tag 'vector-sorted-list)
(define default-size 20)
(define (make len <<? ==?)
  (list sorted-list-tag len -1 (make-vector (max default-size len)) ==? <<?))
(define (size slst)
  (cadr slst))
(define (size! slst s)
```

```
  (set-car! (cdr slst) s))
(define (current slst)
  (caddr slst))
(define (current! slst idx)
  (set-car! (cddr slst) idx))
(define (storage slst)
  (cadddr slst))
(define (storage! slst vector)
  (set-car! (cdddr slst) vector))
(define (equality slst)
  (caddr (cddr slst)))
(define (lesser slst)
  (caddr (cdddr slst)))

(define (new <<? ==?)
  (make 0 <<? ==?))

(define (from-scheme-list slst <<? ==?)
  (let loop
    ((lst slst)
     (idx 0))
    (if (null? lst)
      (make idx <<? ==?)
      (add! (loop (cdr lst) (+ idx 0)) (car lst)))))
```

The implementations of the verification procedures `sorted-list?`, `length`, `empty?` and `full?` are omitted since they are trivial.

Below we list the procedures that manipulate and rely on the current that is stored in the header. `set-current-to-first!`, `set-current-to-next!`, `has-current?` and `current-has-next?` are quite trivial. They make the header's current point to the correct index in the vector. Notice that the value -1 is used to invalidate the current, e.g. after executing `delete!`.

```
(define (set-current-to-first! slst)
  (current! slst 0))

(define (set-current-to-next! slst)
  (if (not (has-current? slst))
    (error "current has no meaningful value (set-current-to-next!" slst)
    (current! slst (+ 1 (current slst)))))

(define (has-current? slst)
  (not (= -1 (current slst))))

(define (current-has-next? slst)
  (if (not (has-current? slst))
    (error "no Current (current-has-next?)" slst)
    (< (+ (current slst) 1) (length slst))))
```

`delete!` itself is implemented using the `storage-move-left` procedure that was discussed in the vectorial implementation of positional lists in section 3.2.5:

```
(define (delete! slst)
  (define vect (storage slst))
  (define last (size slst))
  (define curr (current slst))
  (if (not (has-current? slst))
    (error "no current (delete!)" slst))
  (if (< (+ curr 1) last)
    (storage-move-left vect (+ curr 1) last))
  (size! slst (- last 1))
  (current! slst -1)
  slst)
```

The focus of our attention is the implementation of `find!`. It accepts a sorted list and a key. It traverses the list until the data element containing the matching key is found and makes the current refer to that data element. Since the list is sorted, `find!` can be optimised in the sense that it will not look any further if the element encountered during the traverse is greater than the key it is looking for. When this is the case, the test (`<<?` (`vector-ref vector idx`) `key`) returns #f and there is no point in continuing the search. This can speed up the implementation of `find!` considerably in the case of a negative answer. However, the result is still in $O(n)$ in the worst case. As we will see in section 3.4.4, the true benefit of using sorted lists stems from the fact that we are using a vector implementation for the `SortedList` ADT.

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vector (storage slst))
  (define length (size slst))
  (define (vector-iter idx)
    (cond
      ((>= idx length)
       (current! slst -1))
      ((==? key (vector-ref vector idx))
       (current! slst idx))
      ((<<? (vector-ref vector idx) key)
       (vector-iter (+ idx 1)))
      (else
       (current! slst -1))))
  (vector-iter 0)
  slst)
```

The price to pay for the gain in speed for `find!` is a slower insertion of data elements in the list: the operation `add!` has to traverse the list in order to find the correct position of the element it is to insert. The worst case performance

characteristic is $O(n)$ since we might end up traversing the entire list. Furthermore, in our vector implementation of the ADT, one also has to perform a storage move in order to shift the remaining elements "one entry to the right". Hence, sorted lists are not a very good choice when the composition of the list is unstable, especially when many insertions of new elements are expected.

Note that we have made `add!` as fast as possible by *not* using `storage-move-right`. Naively, we might implement `add!` by first searching — from left to right — the position of the element to be inserted and then moving all elements sitting on the right of that position one position to the right. This would cause `add!` to traverse the entire list. Hence it would be in $\Theta(n)$. Our implementation is a bit more clever. We traverse the list — from right to left — in order to search for the position of the element to be inserted. At the same time we copy the elements one position to the right. Having found the correct position like this, all we have to do is store the element to be inserted in the vector. This implementation of `add!` is clearly in $O(n)$. Nevertheless, on the average, it is twice as fast as the aforementioned naive insertion algorithm.

```
(define (add! slst val)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vector (storage slst))
  (define length (size slst))
  (define (vector-iter idx)
    (cond
      ((= idx 0)
       (vector-set! vector idx val))
      ((<<? val (vector-ref vector (- idx 1)))
       (vector-set! vector idx (vector-ref vector (- idx 1)))
       (vector-iter (- idx 1)))
      (else
       (vector-set! vector idx val))))
  (if (= length (vector-length vector))
    (error "list full (add!)" slst))
  (vector-iter length)
  (size! slst (+ length 1))
  slst)
```

3.4.4 Binary Searching

Although we have already presented a number of pretty inventive optimization techniques for `find`, the resulting procedure still has a performance characteristic in $O(n)$. An excellent searching algorithm — known as *binary search* — beats this performance characteristic up to the level of $O(log(n))$. The algorithm is shown below. It explicitly relies on the fact that we have chosen a vector implementation for our sorted lists. Conform with the vector representation of sorted lists, the algorithm uses the value `-1` to invalidate the current when the

key being searched for does not occur in the list.

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vector (storage slst))
  (define length (size slst))
  (let binary-search
    ((left 0)
     (right (- length 1)))
    (if (<= left right)
      (let ((mid (div (+ left right 1) 2)))
        (cond
          ((==? (vector-ref vector mid) key)
           (current! slst mid))
          ((<<? (vector-ref vector mid) key)
           (binary-search (+ mid 1) right))
          (else
           (binary-search left (- mid 1)))))
      (current! slst -1)))
  slst)
```

The idea of binary searching is extremely clever. Since a list is sorted, we can divide it into two halves and determine whether the search key is to be found in the first half or in the second half of the list. We start the search by considering all indexes between 0 and (− `length` 1). Then we divide the list into two halves by computing the `mid` position of the vector. By comparing the key with the data element residing at the `mid` position, we know that the element has to occur in the first half or whether the element has to occur in the second half — that is, if it occurs. Depending on this knowledge, the algorithm calls itself recursively with the boundaries of one of the halves. The ability of vectors to access their entries in $O(1)$ is a crucial property for the well-functioning of this algorithm. Indeed, it is crucial that the element sitting at position `mid` is accessible in $O(1)$.

In order to determine the performance characteristic of this algorithm, we merely have to determine the number of recursive calls `binary-search` can make since the entire body of `binary-search` (except for the recursive calls) is $O(1)$. In every recursive call, we divide the list in two halves and a recursive call is made with one of those halves. Hence, we are looking for the number of times, say k, a list of size n can be halved before the length of the remaining sublist is 1. In other words, we look for the k such that $\frac{n}{2^k} = 1$. Solving this equation for k, we get $k = \lceil log_2(n) \rceil$. In other words, given a list of size n, `binary-search` will recursively call itself $\lceil log_2(n) \rceil$ times. Hence, the algorithm is in $O(log(n))$ which is an extremely good result.

3.5 Rings

A final ADT that is often associated with linear data structures is the `ring` ADT. A ring is a linear data structure in which *every* element has a next and a previous element. In contrast to positional lists, there are no exceptional elements that don't have a next or previous element. The `ring` ADT is shown below.

```
1  ADT ring
2
3  new
4      ( ∅ → ring )
5  from−scheme−list
6      ( pair → ring )
7  ring?
8      ( any → boolean )
9  add−after!
10     ( ring any → ring )
11 add−before!
12     ( ring any → ring )
13 shift−forward!
14     ( ring → ring )
15 shift−backward!
16     ( ring → ring )
17 delete!
18     ( ring → ring )
19 update!
20     ( ring any → ring )
21 peek
22     ( ring → any )
23 length
24     ( ring → number )
```

Rings have an inherent notion of "a current" which always refers to a meaningful value. The only exception to this is when the ring doesn't contain any data elements. In all other cases, the header of the ring must necessarily refer to at least *some* element in the ring. This element is the ring's current. The operations `add-before!` and `add-after!` insert a data element before or after that current. The newly added data value plays the role of the current after the operation has finished execution. The operation `delete!` removes the current element from the ring and makes the current refer to the next element in the ring. `shift-forward!` and `shift-backward!` move the current of the ring one position "to the right" or "to the left". Just as in the previously described list ADTs, `update!` (resp. `peek`) allows one to overwrite (resp. read) the element residing in the current of the ring.

The following code shows a single linked implementation of the `ring` ADT. We start with the representation. Rings are represented as enhanced lists that simply maintain a reference to the ring's current element. Ring nodes are identical to the nodes used by the single linked implementation of positional lists discussed in section 3.2.6.

```
(define ring-tag 'ring)
(define (make current)
  (list ring-tag current))
(define (current ring)
  (cadr ring))
(define (current! ring node)
  (set-car! (cdr ring) node))

(define (make-ring-node val next)
  (cons val next))
(define (ring-node-val node)
  (car node))
(define (ring-node-val! node val)
  (set-car! node val))
(define (ring-node-next node)
  (cdr node))
(define (ring-node-next! node next)
  (set-cdr! node next))

(define (new)
  (make '()))

(define (from-scheme-list slst)
  (let loop
    ((scml slst)
     (ring (new)))
    (if (null? scml)
      ring
      (loop (cdr scml) (add-after! ring (car scml))))))

(define (ring? any)
  (and (pair? any)
       (eq? (car any) ring-tag)))
```

The following procedure computes the length of a ring. The implementation is in $O(n)$ but we now know that it is an easy programming exercises to store more information in the header of the ring in order to make this procedure run faster.

```
(define (length ring)
  (define curr (current ring))
  (if (null? curr)
```

```
  0
  (let loop
    ((pointer (ring-node-next curr))
     (acc 1))
    (if (eq? pointer curr)
      acc
      (loop (ring-node-next pointer) (+ acc 1))))))
```

Navigating through the ring can be done using the `shift-forward!` and `shift-backward!` procedures shown below. The former just follows the next pointer in the current node. Therefore, it is an $O(1)$ operation. The latter has to traverse the entire ring until the previous node of the current node is found. This is because we have opted for a *single* linked implementation. If we replace our implementation by a *double* linked one, then `shift-backward!` can be easily implemented in $O(1)$ since all we have to do is follow the back pointer stored in a double linked ring node.

```
(define (shift-forward! ring)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (shift-forward!)" ring))
  (current! ring (ring-node-next curr))
  ring)

(define (shift-backward! ring)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (shift-backward!)" ring)
    (current! ring (let iter-to-previous
                     ((prev curr)
                      (next (ring-node-next curr)))
                     (if (eq? curr next)
                       prev
                       (iter-to-previous next (ring-node-next next))))))
  ring)
```

`peek` and `update!` operate relatively to the current and do not cause any further navigation.

```
(define (update! ring val)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (update!)"ring)
    (ring-node-val! curr val)))

(define (peek ring)
  (define curr (current ring))
  (if (null? curr)
```

```
    (error "empty ring (peek)" ring)
    (ring-node-val curr)))
```

The mutators for rings are implemented below. `add-after!` merely inserts a new node after the current. The implementation is $O(1)$ since we only manipulate next pointers. The implementation of `add-before!` is in $O(n)$ though. The reason is that our single linked implementation does not provide a back pointer to find the previous node of the current node. This previous node is needed since we have to make it refer to the new one. The next node of the newly added node is the current. Likewise, `delete!` is in $O(n)$. In order for the result of `delete!` to be a valid ring, we have to make sure that the previous node of the current refers to the next node of the current. As such, we need an iterative process to find the previous node. Again, a double linked implementation would change `add-before!` and `delete!` into $O(1)$ operations.

```
(define (add-after! ring val)
  (define curr (current ring))
  (define node (make-ring-node val '()))
  (ring-node-next! node
                   (if (null? curr)
                     node
                     (ring-node-next curr)))
  (if (not (null? curr))
    (ring-node-next! curr node))
  (current! ring node)
  ring)

(define (add-before! ring val)
  (define curr (current ring))
  (define node (make-ring-node val curr))
  (ring-node-next!
   (if (null? curr)
     node
     (let iter-to-previous
       ((prev curr)
        (next (ring-node-next curr)))
       (if (eq? curr next)
         prev
         (iter-to-previous next (ring-node-next next))))))
   node)
  (current! ring node)
  ring)

(define (delete! ring)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (delete!)" ring))
```

```
(ring-node-next!
 (let iter-to-previous
   ((prev curr)
    (next (ring-node-next curr)))
   (if (eq? curr next)
     prev
     (iter-to-previous next (ring-node-next next))))
 (ring-node-next curr))
(if (eq? curr (ring-node-next curr))
  (current! ring '())
  (current! ring (ring-node-next curr)))
ring)
```

As already mentioned several times, the performance characteristics for the implementation of our `ring` ADT can be improved drastically by using a double linked implementation and by storing extra information (such as the length of the ring) in the header. Just as is the case for positional lists, an implementation in which nearly all operations are in $O(1)$ can be achieved this way. Rings can also be implemented using vectors but that implementation is not very efficient. We leave it as an exercise to the reader to verify that all operations (except for `shift-forward` and `shift-backward`) are necessarily in $O(n)$.

Examples

Rings occur frequently in computer science. Here are just two examples:

- A famous application of rings is the notion of a *round-robin task scheduler*. A scheduler is a program that maintains a ring of tasks. One can think of a task as a Scheme expression. A scheduler handles the tasks one by one by executing (a limited number of steps of) the task and by frequently moving on to the next task in the ring. E.g., a scheduler might choose to evaluate a number of subexpressions of a Scheme expression and then move on to the next Scheme expression in the scheduler. As a result all tasks have the impression of being executed simultaneously given the fact that they receive a fair amount of time by the scheduler. The resulting behaviour is called *time sharing* because all tasks have the impression to be executed on a separate computer even though they share the same computer. Schedulers are at the heart of operating systems such as MacOsX, Linux, Windows and Unix since these are all task-based. Hence, the notion of a ring is buried deep down in almost every computer system.

- A second example of a ring data structure can be found in graphical window-based operating systems. A very typical menu option in these systems is "Cycle Through Windows". The idea is that the windows are organised in a ring. By selecting this menu option, the next window in the ring is activated and displayed as the frontmost window on the screen. Internally, the windows are stored in a ring data structure.

Chapter 4

Linear Abstract Data Types

Over the years, computer scientists have invented a number of ADTs that are extremely frequently used when constructing software systems. The `dictionary` ADT discussed in chapter one is an example of such an ADT. It is used so often by software practitioners that it deserves to be studied in great detail. This is the topic of chapter 6 and chapter 7. This chapter discusses a few other ADTs that occur over and over in software applications. The ADTs studied — stacks, queues and priority queues — belong together in one chapter because they all exhibit *linear behaviour*. By this we mean that the operations of the ADT suggest that the elements of the corresponding data structure are organised in a linear way, pretty much in the sense studied in the previous chapter. By simply looking at the definitions of these ADTs, straightforward reasoning leads us to the conclusion that they are just special cases of lists and that they are naturally implemented as such. Therefore, a large part of this chapter consists of applying the knowledge of the previous chapter to implement the linear ADTs defined here. However, we will see that it is sometimes more beneficial to implement ADTs that exhibit lineair behaviour by non-linear data structures. An example of this is the priority queue ADT for which a non-linear implementation using heaps is the most optimal one. Heaps are also studied in this chapter.

We study the definition of three linear abstract data types, to wit stacks, queues and priority queues. We study the implementation trade-offs that have to be made for different representations. Performance characteristics are presented for every implementation.

4.1 Stacks

A first important ADT that exhibits linear behaviour is the `stack` ADT. You are probably already familiar with stacks in everyday life. In a university restaurant, plates are often arranged in a stack: putting a plate on top of the stack makes the entire stack shift down one level and picking a plate from the top of the stack makes the stack shift all plates one position up. The essence of this behaviour

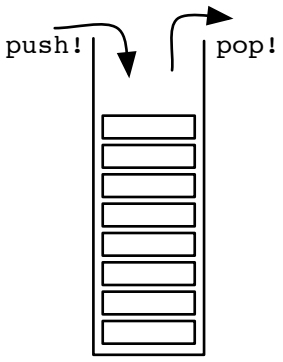

Figure 4.1: The Behaviour of a Stack

is that there are two major operations that one can apply on a stack: *push* an element on the stack and *pop* and element from the stack. Other additions and deletions are not allowed as these violate the behavioural principle that governs the order with which elements are added to and deleted from the stack. The order in which elements are pushed onto the stack and popped from the stack is governed by the LIFO principle: last in, first out. At any point in the lifetime of a stack, the only element that can be popped from the stack is the element that was pushed most recently. The behaviour of stacks is depicted in figure 4.1 This prescribed behaviour of stacks suggests that they are best implemented by means of a data structure that grows and shrinks "at one end".

In computer science as well, stacks are extremely useful. Here are two important examples:

Internet Browsers When browsing the internet, one uses a browser such as Apple's Safari, Microsoft's Internet Explorer or Firefox. In the browser, every time one clicks a link, the link is pushed on the internal stack of the browser. As such, the stack contains the navigational history of the user. Every time a link takes the user "deeper" into the net, the link is stored on the stack. At any time in the browsing process, the user can decide to go back in his or her navigational history by clicking the "Back" button. When clicking this button, the link which resides at the top of the stack is popped from the stack and used to reload and show the corresponding page.

Undo Modern applications such as word processors and drawing programs feature an "undo" menu option. At any stage during the execution of the program, the user can select "undo" which reverses the most recent action taken by the user. Whereas older applications only remember the most recent action, modern applications are much "smarter" in the sense that they can unwind a huge number of user actions. This unwinding is to be

done in reverse order. To implement this feature, the application maintains a stack of descriptions of user actions. Every time the user selects an action in the application, a description of that action is pushed onto the stack (e.g. "user typed a *t*", "user selected 'save"'or "user dragged a diagram to a new location") such that selecting "undo" always finds the most recent action on the top of the stack. By popping that action from the stack, the application can rewind the action, after which it finds the next to last action on the top of the stack again.

Notice though that application's undo-stack is a bit of a special stack since an application cannot remember *all* actions of the users: computer memory is limited. Therefore, a 'stack depth' N is fixed and the application's stack only remembers the N most recent actions. Pushing an action on the stack causes the stack to forget its oldest element after having exceeded N. Hence, the application's stack is not a pure stack. Nevertheless it helps to think of the data structure as a stack.

These are just two examples. Many other examples of stacks in computer science exist and this is the main reason for including stacks in a text on data structures. The following sections formally specify the ADT and present a number of alternative implementation strategies. Different representations are compared by studying their memory consumption as well as the performance characteristics of the implementations of the ADT operations.

4.1.1 The Stack ADT

The `stack` ADT is shown below. A stack contains values of any Scheme data type. We don't assume any operators that have to work on the values stored in the stack. This is in contrast with most of the ADTs presented in chapter 3 which all assume the presence of an operator `==?` the procedural type of which is (`V V` → `boolean`) where `V` is the data type of the values stored. Since the `stack` ADT does not put any restrictions on the data type, it can be used with any Scheme data type. Hence, when we create "a" stack, we are actually dealing with a stack that can store `any` data type. Let us look at the specification of the ADT.

```
1  ADT stack
2
3  new
4      ( ∅ → stack )
5  stack?
6      ( any → boolean )
7  push!
8      ( stack any → stack )
9  top
10     ( stack → any )
11 pop!
```

```
12      ( stack → any )
13  empty?
14      ( stack → boolean )
15  full?
16      ( stack → boolean )
```

The ADT specifies a constructor `new` that does not take any arguments. It returns an empty stack. The predicate `stack?` can be applied to any Scheme value and checks whether or not that value is a stack. `empty?` and `full?` are predicates that can be used to prevent programs from crashing due to stack overflow and stack underflow errors. Just as with the linear data structures presented in chapter 3, it will depend on the representation whether or not stacks can ever be full.

`push!` takes a stack and any data value. It pushes the value on top of the stack and returns the modified stack. This means that `push!` is a destructive operation. Hence, even though a stack is returned from the operation, the original stack is modified. Returning the stack from `push!` turns out to be quite handy. Since the result of `push!` is a stack again, expressions like (`push!` (`push!` `s` 1) 2) are possible. If it wouldn't be for this kind of *cascaded* expressions, we would have to use a cumbersome `begin` construction to group together two calls to the `push!` operation.

`pop!` takes a stack and returns the most recently added element from the stack. This element is called the `top` of the stack. The top element is returned from `pop!` and the stack is destructively changed in the sense that the top is no longer part of the stack. `top` peeks into the stack and returns its top data element. However, in contrast to `pop!`, the top element is not removed and the stack is not modified.

Let us now investigate the options we have for representing stacks and for implementing the operations defined on stacks. In what follows, the stack ADT is implemented by means of a positional list. Based on what we know from the previous chapter, we study two implementations: a vectorial implementation and a linked list implementation. It turns out that both implementations give rise to $O(1)$ performance characteristics, even if we opt for the single linked implementation. This luxury situation reduces the choice between a vectorial implementation and a linked implementation to the question of whether flexibility or efficient memory consumption is of utmost importance. The vectorial implementation is more efficient concerning memory consumption since it doesn't require storing next pointers. However the vector implementation is less flexible because we have to know the stack size upfront. The linked implementation is much more flexible but requires about twice as much memory because we need to store next pointers to link up the stack nodes.

4.1.2 Vector Implementation

The first implementation of the `stack` ADT uses the vector-based implementation of positional lists presented in section 3.2.5. The Scheme definitions needed

to represent a vector-based stack are given below. In this code, we assume that the clause importing the positional list ADT is importing the vectorial implementation of positional lists. The ellipsis shown (i.e. ...) in the library represent the procedures explained in the rest of this section.

```
(library
 (stack)
 (export new stack? push! pop! top empty? full?)
 (import (rnrs base (6))
         (prefix (a-d positional-list adt) plist:))

 (define stack-tag 'vector-stack)
 (define (make)
   (list stack-tag (plist:new eq?)))
 (define (plist stack)
   (cadr stack))

 (define (new)
   (make))

 (define (stack? any)
   (and (pair? any)
        (eq? (car any) stack-tag)))

 ...)
```

A stack is represented as an enhanced list that holds a reference to a vector-based positional list. The private accessor `plist` is used to select the list from the enhanced list that represents the stack object. Given this representation, the implementation for the stack operations is straightforward. The implementation of `empty?` and `full?` is not at all thrilling. They merely check whether the underlying list is empty or full. Both operations are obviously in $O(1)$.

```
(define (empty? stack)
  (plist:empty? (plist stack)))

(define (full? stack)
  (plist:full? (plist stack)))
```

The implementation for `push!`, `pop!` and `top` is given below. A stack is implemented as a positional list that "grows to the right" when pushing elements on the stack. This is accomplished by `add-after!`. By calling this procedure without an explicit argument, the value is added after all the elements already residing in the list. Popping elements from the stack is realized by making the list "shrink to the left". This is accomplished by peeking at the last position of the stack, deleting the last position and returning the value obtained from peeking. `top` merely peeks at the last position of the positional list underlying the stack.

```
(define (push! stack val)
  (plist:add-after! (plist stack) val)
  stack)

(define (top stack)
  (define plst (plist stack))
  (if (= (plist:length plst) 0)
    (error "stack empty (top)" stack))
  (plist:peek plst (plist:last plst)))

(define (pop! stack)
  (define plst (plist stack))
  (if (= (plist:length plst) 0)
    (error "stack empty (pop)" stack))
  (let ((val (plist:peek
              plst
              (plist:last plst))))
    (plist:delete!
     plst (plist:last plst))
    val))
```

What can we say about the performance characteristics of these procedures? Since the implementation assumes a vectorial implementation of positional lists, we have to look at the table in figure 3.5. From this table, we observe that all operations used in our `stack` implementation are in $O(1)$, except for `add-after!` which is in $O(n)$. At first sight, this requires us to conclude that `push!` is in $O(n)$ as well. However, having another look at the vector-based implementation of `add-after!`, we notice that `add-after!` has an interesting *best-case* performance characteristic that is in $O(1)$. The best case of `add-after!` occurs when adding an element beyond the very last position of the vector-based positional list. In this case, no storage move "to the right" is needed to free up space in the vector. All the operation does is insert the element in the first free position of the vector. This best-case of `add-after!` is precisely the case that is used by `push!`. Hence, `push!` is in $O(1)$ as well.

4.1.3 Linked Implementation

The linked implementation of the `stack` ADT uses the single linked implementation of the positional list ADT presented in section 3.2.6. Again, stacks are represented by enhanced lists that hold a reference to a positional list. This is accomplished in the following library. In this case, we assume that the import clause that imports the positional list does *not* import the vectorial implementation.

```
(library
 (stack)
 (export new stack? push! pop! top empty? full?)
```

```
(import (except (rnrs base (6)) map list length for-each)
        (rnrs io simple)
        (rnrs mutable-pairs)
        (prefix (a-d positional-list adt) plist:))

(define stack-tag 'linked-stack)
(define (make)
  (cons stack-tag (plist:new eq?)))
(define (plist stack)
  (cdr stack))

(define (new)
  (make))

(define (stack? any)
  (and (pair? any)
       (eq? (car any) stack-tag)))

...)
```

The verification operations for stacks are shown below. They merely access the underlying positional list and apply the same operation to that list. They are obviously in $O(1)$ because the corresponding operations on positional lists are in $O(1)$ too.

```
(define (empty? stack)
  (define plst (plist stack))
  (plist:empty? plst))

(define (full? stck)
  (define plst (plist stck))
  (plist:full? plst))
```

Finally, we describe `push!`, `pop!` and `top`. For the linked implementation, we use the exact oposite strategy as the one used in the vectorial implementation: the stack grows to the left and shrinks to the right. This is accomplished by implementing `push!` using `add-before!`. By omitting the position argument, the value is guaranteed to be added before all other elements already sitting in the positional list. Accordingly, `top` peeks at the first position in the positional list. `pop` peeks at the first position and subsequently deletes the first position. The code for linked stacks looks as follows:

```
(define (push! stack val)
  (plist:add-before! (plist stack) val)
  stack)

(define (top stack)
  (define plst (plist stack))
```

```
  (if (= (plist:length plst) 0)
    (error "stack empty (top)"stack))
  (plist:peek plst (plist:first plst)))

(define (pop! stack)
  (define plst (plist stack))
  (define first-position (plist:first plst))
  (if (= (plist:length plst) 0)
    (error "stack empty (pop)" stack))
  (let ((val (plist:peek plst first-position)))
    (plist:delete! plst first-position)
    val))
```

In order to come up with the performance characteristics, we refer to the table in figure 3.7. A remark similar to the one we made with the vector-based implementation has to be made here. By reading the table, we notice that `add-before!` is in $O(n)$. This is because a linked lists needs an iterative process to find the node that precedes the node to be added in order to make the next pointer refer to the newly added node. However, when calling `add-before!` without specifying a position, the element is added as the very first element in the list. In a single linked list, this merely requires to store a new first element in the header of the list. Hence, `add-before!` has a best-case performance characteristic in $O(1)$ for linked implementations. This is precisely the case that is used by `push!`. A similar remark is need to establish the performance characteristic of `pop!`. Since `delete!` for single linked positional lists is in $O(n)$, we would conclude that `pop!` is in $O(n)$ as well. However, deleting the very first element of a single linked list is guaranteed to be in $O(1)$ since we do not need an iteration to find the previous element of the element to be deleted. Hence, `delete!` exhibits a best-case performance characteristic in $O(1)$ and this best-case is exactly the one relied upon by `pop!`. We conclude that all operations are in $O(1)$.

4.1.4 Discussion

The performance characteristics of both implementations for the `stack` ADT are summarized in figure 4.2. As we can see from the table, all operations can be realized in $O(1)$. As a result, the tradeoff to be made when selecting an implementation is related to memory consumption instead of runtime performance. Whereas a stack of n elements requires $O(n)$ memory cells in the vectorial implementation, $O(2.n)$ cells are required in the (single) linked implementation. The price to pay for a more memory-efficient implementation is a loss of flexibility: the size of vectorial stacks has to be known upfront and pushing an element onto a full stack requires one to create a new vector and requires one to copy all elements from the old vector into the new vector.

One variation of the `stack` ADT that occurs quite frequently in computer science is the `stack-pair` ADT. A `stack-pair` is a data structure that manages

Operation	Vector	Linked
`new`	$O(1)$	$O(1)$
`stack?`	$O(1)$	$O(1)$
`empty?`	$O(1)$	$O(1)$
`full?`	$O(1)$	$O(1)$
`top`	$O(1)$	$O(1)$
`push!`	$O(1)$	$O(1)$
`pop!`	$O(1)$	$O(1)$

Figure 4.2: Comparative Stack Performance Characteristics

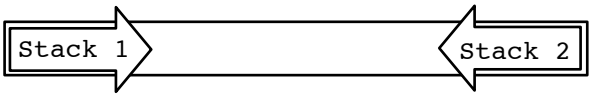

Figure 4.3: A pair of stacks in a vector

two stacks at a time. The reason why this ADT is attractive is that it has an efficient vectorial implementation. The vectorial implementation of regular stacks needs an estimate of the capacity at creation time which can result in painful wastes of memory. This pain is mitigated by the `stack-pair` ADT. The idea of the ADT consists of using the wasted memory to host a second stack. As a result, the vector contains one stack that "grows to the right" in the vector, and a second stack that "grows to the left" in the same vector. The principle is shown in figure 4.3. The specification of the ADT as well as its implementation is left as an exercise. Note that — in contrast to the ordinary `stack` ADT presented above — the vector implementation will no longer be able to rely on the positional list abstractions presented in this chapter if we want to achieve performance characteristics in $O(1)$ for both stacks. We get a much better performance if we start from scratch and implement the `stack-pair` ADT immediately on top of naked vectors instead of positional lists. This is one example that clearly shows that in order to improve efficiency, we sometimes need to circumvent abstraction barriers and access the lower levels directly.

4.2 Queues

A second frequently occurring ADT that exhibits linear behaviour is the `queue` ADT. There are many examples of queues in our daily lives. For example, at the cash register of a supermarket, customers queue up to pay for their groceries. A pile of documents that are to be processed by an administration is another example of a queue. Every time new documents arrive, they are put on the pile of documents to be processed and every time a secretary is ready to process

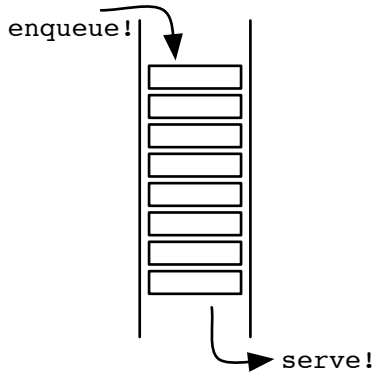

Figure 4.4: The behaviour of a queueue

a new document, he or she takes the document from the bottom of the pile. The defining property of a queue is that items appear at one end of the queue and disappear at the other end of the queue. In other words, queues exhibit a FIFO behaviour: first in, first out. This kind of behaviour is depicted in 4.4. The operation that adds a value to the rear of the queue is called *enqueue*. The operation that reads a value at the front of the queue is called *serve*. The front of the queue is usually referred to as the *head* of the queue. The rear is usually referred to as the *tail* of the queue.

In computer science, queues are extremely useful and appear very frequently. Below are just two examples:

Print Queues In many offices, labs and companies, several computers share the same printer or set of printers. This means that more than one user can possibly print a document to the printer at the same time. In order to avoid problems with colliding documents, the printer therefore maintains a queue of documents that it has received. Every time the printer receives a new document from one of the computers, the document is added to the tail of the queue. Every time a document is successfully printed, the printer serves the next document from the head of the queue and prints it.

Outgoing Mail Most modern mail reading programs allow you to send mail even though you are offline. To enable this, the mailing program internally manages a queue of outgoing messages. Each time we write a mail and try to send it while being offline, the mail is added to the tail of the queue. Whenever an internet connection is established for your computer, the mailing program flushes the queue of outgoing messages by selecting every message from the head of queue and by sending that message. Thus, emails are sent in the order in which they were written.

Many more examples of queues exist in computer science. Therefore, queues form part of the standard vocabulary of computer science which is the reason to include a systematic study of queues in this text.

4.2.1 The Queue ADT

Below we show the formal definition of the `queue` ADT. The constructor `new` takes no arguments and returns an empty queue. The operations `full?`, `empty?` and `queue?` are similar to the corresponding operations of the `stack` ADT.

The behavioural properties of queues are guaranteed by three operations. `enqueue!` takes a queue and a data element. It adds the data element to the tail of the queue. The resulting modified queue is returned from the operation. `serve!` takes a queue and returns the data element sitting at the head of the queue. The element is removed from the queue. `peek` is similar to `serve!` but does *not* remove the data element from the queue.

```
1  ADT queue
2
3  new
4      ( ∅ → queue )
5  queue?
6      ( any → boolean )
7  enqueue!
8      ( queue any → queue )
9  peek
10     ( queue → any )
11 serve!
12     ( queue → any )
13 empty?
14     ( queue → boolean )
15 full?
16     ( queue → boolean )
```

4.2.2 Implementation Strategies

Unfortunately, implementing the `queue` ADT is not as trivial as implementing the `stack` ADT because we can no longer rely on the `position-list` abstractions provided by the previous chapter. The reasons for this are:

- In the vectorial implementation of positional lists we have two options. Either we add elements at the end of the positional list or we add elements to the beginning of the positional list. In order to ensure a correct behaviour of the `queue` ADT, the former option requires us to remove elements from the start of the positional list. The latter option requires us to remove elements from the end of the positional list. Either option is

unsatisfactory since adding an element to the beginning of a vectorial positional list is in $O(n)$. Similarly, removing an element from the beginning of a vectorial positional list is in $O(n)$. Hence, both options would require a storage move which gives the following dilemma: either we choose `enqueue!` to be in $O(1)$ (but then `serve!` is in $O(n)$), or we choose `serve!` to be in $O(1)$ (but then `enqueue!` is in $O(n)$). We therefore reject a queue implementation that is based on vectorial positional lists.

- In the single linked implementation, adding an element to the start of the list is in $O(1)$ but removing an element from the end of the list is in $O(n)$. Conversely, adding an element to the end is in $O(n)$ while removing it from the start of the list is in $O(1)$. Again, choosing `enqueue!` to be in $O(1)$ causes `serve!` to be in $O(n)$ and the other way around. The double linked list implementation suffers from the same problems: since it doesn't store an explicit reference to the last position, either enqueueing or serving an element at the last position of the list is in $O(n)$.

- The enhanced double linked implementation offers better performance results. By explicitly storing a reference to the last position of the list, both `serve!` and `enqueue!` can be implemented as $O(1)$ procedures, irrespective of our choice to make the queue grow to left or to the right.

These results can be summarized in figure 4.5. From the figure, we might be tempted to conclude that an enhanced double linked list implementation is the only viable one. However, we can improve the results of this table drastically by two insights:

Operation	Vector	Single Linked	Double Linked	Double Linked 2
`new`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`empty?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`full?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`enqueue!`	$O(n)$ or $O(1)$	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	$O(1)$
`serve!`	$O(1)$ or $O(n)$	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(1)$
`peek`	$O(1)$	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(1)$

Figure 4.5: Comparative Queue Performance Characteristics

- The fact that there is not much of a difference between the single linked and the double linked implementation on the one hand, but that there is a huge difference between the double linked and the enhanced double linked implementation proves that the real reason for an $O(1)$ characteristic for all the operations is the fact that the enhanced double linked list implementation for positional lists has a direct reference to the last position in the list. In other words, we can build a more efficient implementation of the `queue` ADT by adopting this characteristic but by using a single

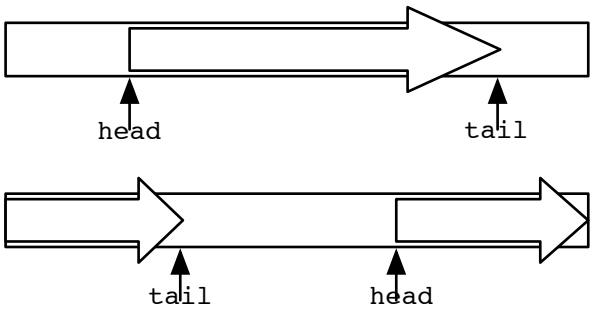

Figure 4.6: A circular vector implementation for queues

linked list. This implementation is twice as efficient concerning memory consumption as the hypothetical enhanced double linked list implementation mentioned in figure 4.5. Hence, the optimal linked implementation for queues consists of a single linked implementation which stores an explicit reference to the last node of the queue. This is the implementation presented in section 4.2.3.

- A similar remark holds for the vectorial implementation. The reason for the $O(n)$ characteristic is that the vectorial implementation for the positional list ADT performs a storage move every time we delete an element from the start of the list. This storage move ensures that the elements of the list are stored in the "leftmost" entries in the vector. This is not really necessary in an implementation for queues. There exist an extremely performant vectorial implementation for queues that has an $O(1)$ performance characteristics for all the operations. It is based on what is often referred to as *circular vectors*. The principle behind the implementation is shown in figure 4.6. The idea is to represent the queue in a vector by maintaining two indices in the vector: `head` is the index in the vector of the entry that designates the head of the queue and `tail` is the index of the vector entry that refers to the end of the queue. Whenever `enqueue!` is called, an element is added at the end of the queue, i.e. at `tail`. `serve!` removes an element from the beginning of the queue, i.e. at `head`. As a result, the queue "crawls" in the vector "from left to right". Whenever it bumps into the last position of the vector, it restarts at the very first position of the vector. As such, the queue crawls in a circular way. The queue is considered full whenever the beginning of the queue (circularly) bumps into its end. The implementation is presented in section 4.2.4.

4.2.3 Linked Implementation

Let us start by discussing the linked implementation. It is obtained by the Scheme library shown below. As usual, we have a double layered constructor consisting of the procedures `new` and `make`. Similar to the linked implementations for positional lists, we define a private `queue-node` abstraction in order to prevent our code from being polluted with Scheme list processing procedures.

```
(library
 (queue)
 (export new queue? serve! enqueue! peek full? empty?)
 (import (rnrs base (6))
         (rnrs mutable-pairs))

 (define (make-queue-node val next)
   (cons val next))
 (define (queue-node-val node)
   (car node))
 (define (queue-node-next node)
   (cdr node))
 (define (queue-node-next! node next)
   (set-cdr! node next))

 (define queue-tag 'linked-queue)
 (define (make)
   (list queue-tag '() '()))
 (define (head q)
   (cadr q))
 (define (head! q node)
   (set-car! (cdr q) node))
 (define (rear q)
   (caddr q))
 (define (rear! q node)
   (set-car! (cddr q) node))

 (define (new)
   (make))

 (define (queue? any)
   (and (pair? any)
        (eq? (car any) queue-tag)))

 ...)
```

The implementation of `empty?` and `full?` is trivial. Since we are discussing a linked implementation, a queue is never full as long as there is Scheme memory left. A queue is empty if the `head` of the enhanced list representing the queue refers to the empty list.

```
(define (empty? q)
  (null? (head q)))


(define (full? q)
  #f)
```

Below we show the implementations of `enqueue!`, `peek` and `serve!`. As explained, the queue is conceived as a single linked list that has a reference to its last node. At this point, we have two options. Either `enqueue!` adds elements to the front of the list and `serve!` removes elements from the end of the list, or vice versa. Removing elements from the end would be problematic since deleting the last node of a linked list require us to get hold of the penultimate node in order to make sure that the header's reference to the last node is updated. But the only way to get hold of the penultimate node is to iterate from the front of the list up to the last node. Hence, this would result in an $O(n)$ procedure again. That is why we have chosen the other option. Elements are added to the rear of the list and removed from the front. In the code for `enqueue!` we observe that a new node is created and that the `rear` in the queue's header is set to refer to that node. The code for `serve!` shows us that the `first` in the queue's header is set to the `next` of the original first node. It should be clear from the code that all procedures are in $O(1)$.

```
(define (enqueue! q val)
  (define last (rear q))
  (define node (make-queue-node val '()))
  (if (null? (head q))
    (head! q node)
    (queue-node-next! last node))
  (rear! q node)
  q)


(define (peek q)
  (if (null? (head q))
    (error "empty queue (peek)" q))
  (queue-node-val (head q)))


(define (serve! q)
  (define first (head q))
  (if (null? first)
    (error "empty queue (serve!)" q))
  (head! q (queue-node-next first))
  (if (null? (head q))
    (rear! q '()))
  (queue-node-val first))
```

4.2.4 Vector Implementation

The vectorial implementation is based on the circular queue principle explained above. A queue is represented as an enhanced vector and a pair of indexes `head` and `tail`. `head` refers to the head of the queue. Serving an element from the queue is accomplished by reading the vector entry that is stored in the head, and replacing the head by its successor. `tail` refers to the tail of the queue. Enqueuing a value means that it has to be stored in the vector entry at the tail and that the tail has to be replaced by its successor as well.

```
(library
 (queue)
 (export new queue? enqueue! serve! peek full? empty?)
 (import (rnrs base (6))
         (rnrs mutable-pairs))

 (define queue-tag 'vector-queue)
 (define default-size 5)
 (define (make size)
   (list queue-tag (make-vector size) 0 0))
 (define (storage q)
   (cadr q))
 (define (head q)
   (cadddr q))
 (define (head! q head)
   (set-car! (cdddr q) head))
 (define (tail q)
   (caddr q))
 (define (tail! q tail)
   (set-car! (cddr q) tail))

 (define (new)
   (make default-size))

 (define (queue? any)
   (and (pair? any)
        (eq? (car any) queue-tag)))

 ...)
```

The implementations of `enqueue!`, `peek` and `serve!` are given below. As explained, `enqueue!` adds an element by taking the successor of the tail and serve removes an element by taking the successor of the head. But taking the successor has to take into account the boundaries of the vector. If the default size of the queue is 50, then the successor of 49 should be 0 in order to get the circular effect described above. This is accomplished by applying the `mod` function to the index and the default size of the queue (i.e. the length of the vector hosting the queue).

```
(define (enqueue! q val)
  (if (full? q)
    (error "full queue (enqueue!)" q))
  (let ((new-tail (mod (+ (tail q) 1) default-size)))
    (vector-set! (storage q) (tail q) val)
    (tail! q new-tail))
  q)

(define (peek q)
  (if (empty? q)
    (error "empty queue (peek)" q))
  (vector-ref (storage q) (head q)))

(define (serve! q)
  (if (empty? q)
    (error "empty queue (peek)" q))
  (let ((result (vector-ref (storage q) (head q))))
    (head! q (mod (+ (head q) 1) default-size))
    result))
```

Checking whether or not a queue is empty or full is a bit tricky. The problem is that a queue is considered empty if the `head` index is equal to the `tail` index. Right after creating the queue, both indexes are 0. The queue is considered full when its tail bumps into its head after successive applications of `enqueue!`. Hence, the test that checks whether or not a queue is empty is exactly the same as the one that checks whether or not a queue is full. This is problematic since it precludes us from making a distinction between empty queues and full queues. There are three solutions to this problem. The first one consist of maintaining a boolean flag that is set to `#t` whenever the queue contains at least one element. The second solution consists of storing the number of elements in the header. The third solution consist of systematically "wasting" one vector entry by considering the queue full whenever the successor of the tail bumps into the head. This is the solution taken by our implementation:

```
(define (empty? q)
  (= (head q)
     (tail q)))

(define (full? q)
  (= (mod (+ (tail q) 1) default-size)
     (head q)))
```

All procedures are simple combinations of arithmetic and vector indexing. As a result, they are all in $O(1)$. Since both the vector implementation and the linked implementation have nothing but operations in $O(1)$, we can conclude that the choice to be made will depend on the amount of flexibility that is required. In an application where the size of the queue is easily established

upfront, the vector implementation outperforms the linked implementation because it is more efficient with respect to memory consumption (it doesn't have to store cdr pointers). If it is impossible to estimate the queue's size upfront, then a linked implementation is preferred. They are both equally fast.

4.3 Priority Queues

After having studied stacks and queues, we now turn our attention to the study of a third important ADT — priority queues — that is often associated with linear data structures. Priority queues are a good way to illustrate the difference between linear *data structures* and linear *ADTs*. We will see that the definition of priority queues suggests a linear implementation as much as the definition of ordinary queues does. However, as we will see in section 4.4, the optimal implementation of priority queues does not rely on a lineair data structure at all.

Abstractly spoken, priority queues are very similar to queues. The order in which elements are served from a queue is determined entirely by the order in which the elements where added to the queue. This is no longer true for priority queues. In a priority queue, every data element added to the queue is associated with a *priority*. The elements are served from the priority queue in the order determined by their priority: elements that have a higher priority are served earlier than elements with lower priorities. This principle is sometimes referred to as HPFO: highest priority first out. Applications of priority queues are abundant in computer science:

Todo lists Your favorite agenda application probably has a facility to manage "todo lists". Most applications allow you to associate a priority with the items in the todo list. On your screen, the todo list is displayed sorted by priority. For instance, items with a higher priority might be shown on the top of the screen while items with a lower priority are shown in the bottom.

Priorities can take several forms. In some applications, they are represented by symbols such as *low*, *high* and *highest*. In others, they are numbers between 1 and n where n is the highest priority. The exact representation of priorities is not important for the definition of a priority queue. What is important is that there is some ordering relation `>>?` that can tell us when one priority is higher than another one. In the first example, `>>?` operates on symbols. In the second example `>>?` simply compares numbers with one another.

Emergencies Most hospitals have an emergency service that is open day and night and which has a limited staff that has to take care of all emergencies arriving at the hospital. Because of the limitation in manpower (e.g. at night), a choice must be made in order to determine which patients require faster treatment. Clearly, a child that has pain in the tummy will have to

wait a while if suddenly an ambulance arrives with a victim of a serious car crash. To formalise this (and to avoid giving people the impression that they are being treated unfairly), every arriving patient is assigned a number that indicates the order in which he or she will be treated. However, depending on how serious his or her symptoms are, the number is printed on a ticket with a different color. For example, three different colors — say red, orange and green — might be used to discriminate between "urgent", "serious" and "not urgent" symptoms. Hence, at any moment in time, the list of patients is ordered according to the number they are assigned and according to the color of the ticket. Clearly, n is of higher priority than $n + 1$ when they are printed on a ticket of the same color. However, n has a higher priority than k when the color of the ticket on which n is printed has a higher priority than the color on which k is printed, even if n is a bigger number than k. The rules just described define an operator `>>?` that is used to order patients in the priority queue.

These are two examples of priority queues in everyday life. Examples are abundant in computer science. Priority queues occur quite frequently at the operating system level of a computer system. A typical example is a priority-based task scheduler. Remember from section 3.5 that we have described a ring to implement a task scheduler. By shifting the ring, every task is given a fair amount of time. However, in realistic operating systems we do not want to give every task an equal amount of time. Some tasks are more crucial than others. E.g., it makes sense to say that a task that is responsible for garbage collecting a Scheme system is more important than other tasks since the other tasks might need memory that has to be reclaimed by the garbage collection task. This makes us conceive the task scheduler as a priority queue in which all tasks are assigned a priority. The scheduler then serves the priority queue. This returns the task with the highest priority. After giving the task some time to execute, the task is enqueued again, possibly after having decreased its priority a little such that the other tasks get execution time as well. This kind of priority-based task scheduling lies at the heart of most modern operating systems. Many more examples of priority queues exist.

4.3.1 The Priority Queue ADT

The `priority-queue` ADT is presented below. It is parametrised by the data type `P` of the priorities that are used to order its elements. In other words, the exact type of the priorities is not fixed and every user of the ADT is entitle to choose his or her own set of priorities. In the constructor of the ADT, we therefore require a procedure `>>?` the procedural type of which is (`P P` → `boolean`). It is used to order the elements in the priority queue. Given two priorities `p1` and `p2`, then (`>>? p1 p2`) should hold whenever "`p1` is a considered to be a higher priority than `p2`".

```
1  ADT priority−queue < P >
2
3  new
4      ( ( P P → boolean ) → priority−queue < P > )
5  priority−queue?
6      ( any → boolean )
7  enqueue!
8      ( priority−queue < P > any P → priority−queue < P > )
9  peek
10     ( priority−queue < P > → any )
11 serve!
12     ( priority−queue < P > → any )
13 full?
14     ( priority−queue < P > → boolean )
15 empty?
16     ( priority−queue < P > → boolean )
```

The procedures listed in the `priority-queue` ADT are very similar to the procedures listed by the `queue` ADT. Only the procedural type of `enqueue!` is slightly different from the one of `enqueue!` of ordinary queues. Besides a priority queue and a data element to be enqueued, it requires an priority argument of type `P` that is supposed to correctly prioritise the element enqueued.

As was the case for the `stack` ADT and the `queue` ADT, priority queues seem to crave for a representation based on linear data structures. In sections 4.3.2 and 4.3.3 we present two alternative linear implementations of priority queues. However, it will turn out to be the case that lineair implementations are suboptimal. Section 4.4 introduces a non-lineair auxiliary data structure — called a *heap* — that turns out to be the optimal implementation strategy for priority queues.

4.3.2 Implementation with Sorted Lists

The first implementation is based on the `sorted-list` ADT presented in section 3.4.3. It is based on the fact that a single linked implementation of sorted lists has been selected. This will become clear later on. The principle of the implementation is simple: a priority queue is nothing but a linear list in which the elements are sorted by priority. Serving an element from the priority queue is then simply achieved by deleting the very first element from the sorted list. Enqueueing is achieved by using `add!` to add an element along with its priority to the sorted list.

The `sorted-list` ADT presented in section 3.4.3 is unaware of priorities. It just sorts "values". We therefore have to create a new kind of values that correspond to the actual values to be stored in the priority queue, paired with their priority. This new kind of values is called a *priority queue item*. In other words, priority queue items are pairs consisting of "an ordinary value" and its associated priority.

```
(library
 (priority-queue)
 (export new priority-queue? enqueue! serve! peek full? empty?)
 (import (except (rnrs base) list)
         (prefix (a-d sorted-list linked) slist:))

 (define pq-item-make cons)
 (define pq-item-val car)
 (define pq-item-priority cdr)
 (define (pq-item-specialize-for-priorities func)
   (lambda (item1 item2)
     (func (pq-item-priority item1)
           (pq-item-priority item2))))
 (define (make >>?)
   (cons pq-tag
         (slist:new (pq-item-specialize-for-priorities >>?)
                    (pq-item-specialize-for-priorities eq?))))
 (define (slist pq)
   (cdr pq))

 (define (new >>?)
   (make >>?))

 (define (priority-queue? any)
   (and (pair? any)
        (eq? (car any) pq-tag)))

 ...)
```

Let's study the notion of a priority queue item first. Our library shows a procedure `make-pq-item` to create a priority queue item, a procedure `val` to select the value from a priority queue item and a procedure `priority` to access the priority of a priority queue item. Such a priority queue item will be created each time a value and its associated priority are enqueued. In other words, a priority queue is represented as an enhanced list that contains a reference to a value of type `sorted-list<pq-item>`. Furthermore, priority queue items are represented as dotted pairs in Scheme.

Notice that the sorted list implementation needs a "smaller than" operator and that the constructor for priority queues receives a "higher priority than" operator. We consider one priority queue item smaller than another priority queue item whenever the priority of the first item is higher than the priority of the second item. The procedure `specialize-for-priorities` is a higher order procedure that takes care of this. It takes a procedure `func` that works on priorities and returns the "lifted" version of the procedure than can work on corresponding priority queue items. The lifted version of the procedure takes two priority queue items, selects their priorities and applies the original

procedure to these priorities. This procedure is used in our implementation to lift the equality operator and the "higher priority than" `>>?` operator to priority queue items. The results of the calls (`specialize-for-priorities >>?`) and (`specialize-for-priorities eq?`) is thus used to order priority queue items in our sorted list implementation.

The implementation of `full?` and `empty?` is straightforward. We just select the sorted list from the priority queue's header and we apply the equivalent procedures for sorted lists.

```
(define (empty? pq)
  (slist:empty? (slist pq)))

(define (full? pq)
  (slist:full? (slist pq)))
```

`enqueue!` simply calls `add!` on the header's sorted list in order to add a newly constructed priority queue item to the sorted list. The priority queue item pairs the element to enqueue along with its priority. From here on, the sorted list takes over. It does the necessary work to put the priority queue item in the correct position of the sorted list.

```
(define (enqueue! pq val pty)
  (slist:add! (slist pq) (pq-item-make val pty))
  pq)
```

Since the list is sorted by priority, all `serve!` has to do is to set the current to the very first position of the sorted list and call `delete!` in order to remove the element from the sorted list. Likewise, `peek` sets the current to refer to the very first position and merely reads the value without removing it from the sorted list.

```
(define (serve! pq)
  (define slst (slist pq))
  (if (empty? pq)
    (error "empty priority queue (serve!)" pq))
  (slist:set-current-to-first! slst)
  (let ((served-item (slist:peek slst)))
    (slist:delete! slst)
    (pq-item-val served-item)))

(define (peek pq)
  (define slst (slist pq))
  (if (empty? pq)
    (error "empty priority queue (peek)" pq))
  (slist:set-current-to-first! slst)
  (pq-item-val (slist:peek slst)))
```

We know from section 3.4.3 that both `add!` and `delete!` are in $O(n)$. For `add!` this result is inherent to sorted lists since we have to perform an iterative

search in order to determine the correct whereabouts of the newly added element in the list. Therefore, the performance characteristic of `enqueue!` is in $O(n)$. This does not depend the internal representation that is used by a particular implementation of sorted lists.

For `delete!`, section 3.4.3 displays $O(n)$ as well. However, this is because we have used a vectorial linked implementation for the sorted list ADT in section 3.4.3. Given a single linked list, this operation is in $O(1)$ when deleting the very first element of the list: all we have to do is forget the original first element of the list and make the list's header refer to a new first element in the list. In other words, the best-case analysis for `delete!` in single linked is in $O(1)$. It occurs whenever we remove the very first element from the list. Since our priority queue implementation *always* removes the first element from the sorted list, we always find ourselves in the best-case of `delete!` for single linked sorted lists. We conclude that, the sorted list implementation for priority queues has a performance characteristic that is in $O(n)$ for `enqueue!` and in $O(1)$ for `serve!`, provided that we select a linked implementation for the `sorted-list` ADT.

4.3.3 Implementation With Positional Lists

In an attempt to improve the performance characteristic of `enqueue!` up to the level of $O(1)$ we now try to get rid of the abstraction offered by sorted lists. Instead of relying on the automated organisation prescribed by sorted lists, we manage things manually by resorting back to ordinary positional lists and organise the priority queue "by hand".

The implementation uses the same notion of priority queue items as the sorted list implementation does. As such, we will not repeat those Scheme definitions. The representation of priority queues is very similar as well. Instead of creating a sorted list, this time, a positional list is created. The implementation selected for the positional list does not matter as long as `add-before!` is in $O(1)$. We know that this is the case for all linked implementations. Our implementation below relies on this fact by using `add-before!` to add priority queue items to the front of the positional list. Should we desire a vector implementation, then the code given below should be changed to use `add-after!` since we know that `add-after!` has a best case performance characteristic that is in $O(1)$ in the vectorial case. Here are the definitions:

```
(library
 (priority-queue)
 (export new priority-queue? enqueue! peek serve! full? empty?)
 (import (except (rnrs base (6)) list)
         (prefix (a-d positional-list adt) plist:))

 (define pq-tag 'positional-list-pq)

 (define pq-item-make cons)
 (define pq-item-val car)
```

```
(define pq-item-priority cdr)
(define (pq-item-specialize-for-priorities func)
  (lambda (item1 item2)
    (func (pq-item-priority item1)
          (pq-item-priority item2))))
(define (make >>?)
  (cons pq-tag
        (cons (plist:new eq?)
              (pq-item-specialize-for-priorities >>?))))
(define (greater pq)
  (cddr pq))
(define (plist pq)
  (cadr pq))

(define (new >>?)
  (make >>?))

(define (priority-queue? any)
  (and (pair? any)
       (eq? (car any) pq-tag)))

...)
```

Again, the implementations for `full?` and `empty?` are mere translations of the same operation to the language of positional lists.

```
(define (full? pq)
  (plist:full? (plist pq)))

(define (empty? pq)
  (plist:empty? (plist pq)))
```

The code for `enqueue!`, `serve!` and `peek` is more interesting. The costly `enqueue!` operation is replaced by an `enqueue!` operation that has a performance characteristic that is in $O(1)$. In order to achieve this, the element is just added to the beginning of the list, right before all other elements of the positional list. In the (single as well as double) linked implementation, this operation is in $O(1)$. However, there is a price to pay for this fast implementation of `enqueue!`. Since items are always added to the priority queue without paying any attention to their priorities, a search process is needed in `serve!` and `peek` in order to determine the priority queue item with the highest priority. The `serve!` procedure shown below shows a loop that traverses the entire positional list using alternating applications of `has-next?` and `next`. While doing this, it computes the *position* of the highest priority and stores it in the variable `maximum-pos`. After finishing the loop, this variable contains the position of the priority queue element with the highest priority. We finish the procedure by calling `delete!` on that position. In order to come up with a performance characteristic for this version of `serve!` we might conclude that we need the

double linked list implementation because `delete!` is in $O(n)$ for all other implementations. But even if we use the double linked implementation, the loop `loop` is used to traverse the entire priority queue. Hence, `serve!` is in $O(n)$ (in fact we can even say that it is in $\Theta(n)$!). We omit the implementation of `peek` because it is entirely similar to `serve!`. The only difference is that it lacks the call to `delete!`.

```
(define (enqueue! pq val pty)
  (plist:add-before! (plist pq) (pq-item-make val pty))
  pq)

(define (serve! pq)
  (define plst (plist pq))
  (define >>? (greater pq))
  (if (empty? pq)
    (error "priority queue empty (serve!)" pq))
  (let*
      ((highest-priority-position
        (let loop
          ((current-pos (plist:first plst))
           (maximum-pos (plist:first plst)))
          (if (plist:has-next? plst current-pos)
            (loop (plist:next plst current-pos)
                  (if (>>? (plist:peek plst current-pos)
                           (plist:peek plst maximum-pos))
                    current-pos
                    maximum-pos))
            (if (>>? (plist:peek plst current-pos)
                     (plist:peek plst maximum-pos))
              current-pos
              maximum-pos))))
       (served-item (plist:peek plst highest-priority-position)))
    (plist:delete! plst highest-priority-position)
    (pq-item-val served-item)))
```

4.3.4 Priority Queue Performance Characteristics

Figure 4.7 summarizes the performance characteristics for the two main operations of the `priority-queue` ADT. The important design decision to be made is whether or not the elements of the priority queue are stored in a way that takes the priorities into account. If this is the case, we end up with a cheap `serve!` but with an expensive `enqueue!`. If the organisation in memory does not take the priorities into account as is the case with a simple positional list, then the `enqueue!` operation becomes cheap. However, the price to pay is a search process to determine the element with the highest priority. As a result, `serve!` gets expensive.

These are pretty bad results. A priority queue is typically not a stable data structure since elements are perpetually added to and removed from the priority queue. So, should we opt for an implementation with a fast `serve!` and a slow `enqueue!` or vice versa. For most applications, it is very hard to decide on which of these two operations will be the bottleneck.

Luckily, there is a way out. The third column shows the performance characteristic of both operations for a third implementation of priority queues that is the topic of section 4.4.8 . It uses an underlying data structure called *a heap*. For the heap-based implementations, both operations have a logarithmic performance characteristic. Heaps are the topic of the next section.

Operation	Sorted List	Positional List		Heap
`enqueue!`	$O(n)$	$O(1)$		$O(log(n))$
`serve!`	$O(1)$	$O(n)$		$O(log(n))$

Figure 4.7: Comparative Priority Queue Performance Characteristics

4.4 Heaps

With the knowledge developed so far we seem to be stuck with a dilemma. Either we obtain a fast `serve!` operation because the items are stored in order but then the price to pay is an `enqueue!` operation which requires an $O(n)$ amount of wok to insert the item in the correct location. Trying to get `enqueue!` faster by just adding the item to the linear data structure has the result of making `peek` and `serve!` expensive since they have to search for the highest priority. Fortunately there is a way out. In this section, we present an auxiliary data structure — called *a heap* — that provides us with a much faster implementation for priority queues. Using heaps, both `serve!` and `enqueue!` exhibit a performance that is in $O(log(n))$. One might say that the workload is spread over both operations.

4.4.1 What is a Heap?

Heaps are not useful on their own. That is why we call them an auxiliary data structure. Heaps are a very useful data structure that is frequently used to implement other ADTs (such as priority queues) and which forms the basis for a number of algorithms (such as the heapsort algorithm and several graph algorithms).

Conceptually, a heap is a sequence of data elements e_1, e_2, ..., e_n that are *ordered* according to what is known as *the heap condition*: for all $i \geq 0$ we have $e_i < e_{2i}$ and $e_i < e_{2i+1}$. Figure 4.8 shows an example of a heap. In the figure, we have drawn an arrow from e_i to e_j whenever $e_i < e_j$. Surely, the order $<$ that is used in this definition depends on the data type of the elements stored. If we create a heap that contains numbers, then we should use Scheme's normal $<$

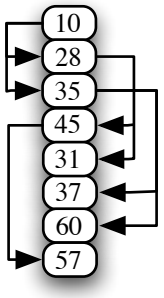

Figure 4.8: A Heap

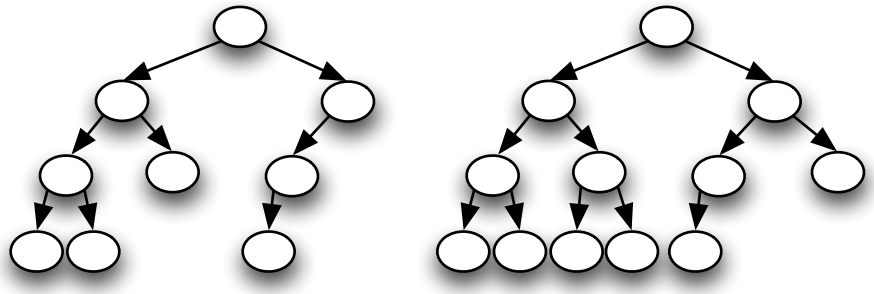

Figure 4.9: A Non-complete Binary Tree and its Completion

operator that compares numbers. If we create a heap of strings, then `string<?` might be used as the ordering needed to satisfy the heap condition.

Caution is required when storing heaps in Scheme vectors. Since Scheme vectors are indexed starting from 0, this would cause us to satisfy the heap condition $e_0 < e_{2.0}$ which is impossible. Hence, *conceptually* heaps start counting from 1 even when their underlying storage vector starts counting from 0. This will require us to do the necessary index conversions in our implementation of heaps.

It is important to understand that the elements in a heap are not necessarily sorted, even though any sorted sequence is also a valid heap since the elements of a sorted sequence automatically satisfy the heap condition . Given a number of data elements, there are many possible arrangements for those elements to form a valid heap. In other words, there doesn't exist a unique heap for a given set of data elements.

Even though a heap is actually just a sequence of elements that happen to

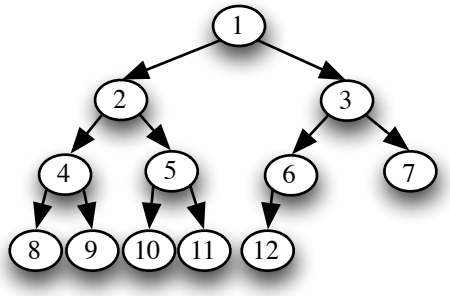

Figure 4.10: Indexing a Complete Binary Tree

be ordered in some clever way, it pays off to think of a heap as if it were a *complete binary tree*. As you probably already know, a tree is a data structure in which every data element has a number of *children*. The data element that has the children is called the *parent*. All data elements in the tree are said to reside in *nodes*. The 'top' node that is not a child of any other node is called the *root node* of the tree. Nodes that have no children are called *leaf nodes*. *Binary trees* are trees in which every node has two (or less) children. Trees do not necessarily need to be *complete* as is illustrated in figure 4.9 which shows a non-complete tree on the left hand side and its completed version on the right hand side. A complete tree is a tree in which all levels are either entirely filled, or partially filled but in such a way that all the positions which are left of a certain node, also contain nodes. A complete tree is a tree that shows no gaps when "read" from left to right, level by level.

The reason why completeness of trees is such an important property is that it forms the basis for seeing the link between a the tree and its representation as a sequence. Indeed, since a complete tree does not contain any gaps, we can assign numbers (also called indices) to its nodes from left to right, level by level. Figure 4.10 shows how this indexing scheme is applied to the completed tree of figure 4.9. If the tree were to contain gaps, this would not be possible in an unambiguous way. It *would* be possible to take a incomplete tree and assign numbers to its nodes. However, given a set of numbered nodes, there is no way to reconstruct the tree since the numbers are not enough to know where the gaps should be. Hence, the interchangeability of a tree and its representation as a numbered sequence relies on the fact that the tree is complete.

The indexing scheme allows a complete tree to be stored in a vector in such a way that the organisation of the vector is enough to redraw the corresponding tree in an unambiguous way. Given this indexing scheme, the children of a node residing in index i in the vector are to be found at indices $2i$ and $2i + 1$ in the vector (check this!). Similarly, given any index i in the vector, then the parent node of the node residing at that index resides in vector entry $\lfloor \frac{i}{2} \rfloor$.

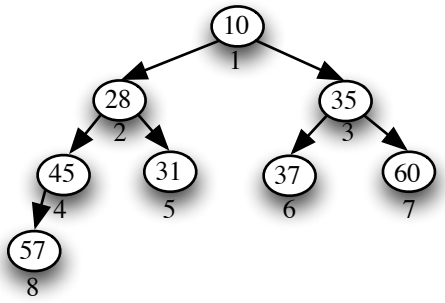

Figure 4.11: A Heap Drawn as a Complete Binary Tree

Now that we know precisely how to think of heaps as complete binary trees, we can redraw the heap displayed in figure 4.8 in as a tree. It is shown in figure 4.11. Using the indexing scheme, we can reformulate the heap condition as the requirement which states that every element in the heap has to be smaller than the elements residing in both of its subtrees. As a consequence, the root node of the heap always contains the smallest element of the heap.

For any node in the tree we define the *height of the node* as the length of the longest path from that node to a leaf node. The length of a path is defined as the number of arrows in the path. The *height of the heap* is the height of the root node, i.e. the length of the longest path in the heap. As an example, the height of the heap depicted in figure 4.11 is three.

4.4.2 Properties of Heaps

Thanks to the correspondence between heaps and complete binary trees, it is possible to derive a number of useful mathematical properties for heaps. These properties will turn out to be essential when deriving performance characteristics for heap-based algorithms.

The first property is a relation between the number of elements in a heap, say n, and the height of the heap, say h. In figure 4.12 we show a heap of height 3. As can be observed from the drawing, every level — except the last — is full: the i^{th} full level contains 2^i elements. The lowest level at the height h contains between 1 and 2^h nodes. This means that a heap of height h has minimum $\sum_{i=0}^{h-1} 2^i + 1$ nodes and maximum $\sum_{i=0}^{h} 2^i$ nodes: $\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i$. Using the fact that for a geometric series $\sum_{i=0}^{h} a^i = \frac{a^{h+1}-1}{a-1}$, we get $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$. Therefore $h \leq log_2(n) < h + 1$ and thus $h = \lfloor log_2(n) \rfloor$. This relation between the number of elements in a heap and the height of a heap is regularly used when establishing the performance characteristics for the heap operations.

Another useful property is that, *in a heap with n nodes there are maximum* $\lceil \frac{n}{2^{h+1}} \rceil$ *elements residing at height h.* This is not hard to see. At height 0 we

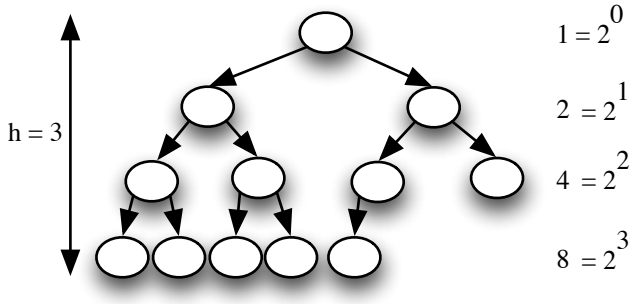

Figure 4.12: Number of Elements in a Heap

only find leaf nodes and we know that a heap has $\lceil \frac{n}{2} \rceil$ leaves. One level higher in the heap, at height 1, we have half the number of nodes we have in the lowest layer, i.e. $\lceil \frac{n}{4} \rceil$. This is because we have 1 parent for every 2 nodes (except maybe for one). Hence, at layer h, we find $\lceil \frac{n}{2^{h+1}} \rceil$. This property also turns out to be useful to establish performance characteristics.

4.4.3 The Heap ADT

.

Below we specify heaps in the form of a `heap` ADT.

```
1  ADT heap < V >
2
3  from−scheme−vector
4     ( vector ( V V → boolean ) → heap < V > )
5  new
6     ( number ( V V → boolean ) → heap < V > )
7  full?
8     ( heap < V > → boolean )
9  empty?
10    ( heap < V > → boolean )
11 insert!
12    ( heap < V > V → heap < V > )
13 delete!
14    ( heap < V > → V )
15 peek
16    ( heap < V > → V )
17 length
18    ( heap < V > → number )
```

The ADT specifies two ways to create a new heap. `new` takes a number

indicating the capacity of the newly created heap. The result of `new` is an empty heap that has enough entries to store as many data elements as indicated by the number used in the constructor call. `from-vector` takes an existing Scheme vector and stores the vector as a heap. The elements of the vector are rearranged by the constructor in order to make them satisfy the heap condition. Both constructors take a procedure `<<?` the procedural type of which is (`V V` → `Boolean`). This determines the ordering that is used to arrange the elements in the heap and to keep the heap condition satisfied at all times. `full?` and `empty?` should be self-explaining. `full?` returns `#t` whenever the heap stores as many elements as indicated by the capacity.

`insert!` takes a (non-full) heap and a data element. It adds the element to the heap thereby rearranging the heap such that its elements meet the heap condition again. Because of the heap condition, it is always the case that the element sitting in the very first position of the heap is the smallest element. One can think of that element as the root of the complete binary tree that corresponds to the heap. It is smaller than its two children, which are in their turn smaller than their children, etc. This is the element returned by `peek`. `peek` does not remove the element from the heap. `delete!` on the other hand, returns the smallest element from the heap, removes it from the heap and rearranges the heap in order for its elements to satisfy the heap condition again.

4.4.4 The Heap Representation

The following code excerpt shows the structure of the implementation of the `heap` ADT in Scheme. A heap is represented as an enhanced list containing a vector, the heap size (i.e. the number of elements that sit in the heap at a certain moment in time) and the comparison operator `<<?` that is used to organise the heap in order for it to satisfy the heap condition.

```
(library
 (heap)
 (export new from-scheme-vector full? empty? insert! delete! peek length)
 (import (except (rnrs base (6)) length)
         (rnrs mutable-pairs))

 (define (make vector size <<?)
   (list 'heap vector size <<?))

 (define (storage heap)
   (cadr heap))
 (define (storage! heap vector)
   (set-car! (cdr heap) vector))

 (define (size heap)
   (caddr heap))
 (define (size! heap s)
   (set-car! (cddr heap) s))
```

```
(define (lesser heap)
  (cadddr heap))

...)
```

The implementations of `length`, `full?` and `empty?` are trivial. As explained, satisfying the heap condition implies that the very first element of the heap is always the smallest element in the heap. Hence, `peek` simply requires us to peek into the first position of the underlying vector.

```
(define (full? heap)
  (= (vector-length (storage heap))
     (size heap)))

(define (empty? heap)
  (= (size heap) 0))

(define (length heap)
  (size heap))

(define (peek heap)
  (if (empty? heap)
    (error "heap empty" heap)
    (vector-ref (storage heap) 0)))
```

The implementations of the operations `insert!` and `delete!` are a bit more contrived. After all, we cannot just insert or delete elements anyplace in the heap. This is most likely to violate the heap condition. We therefore restrict insertion and deletion to some very specific cases. The idea is to guarantee that the very first element of the heap is the only element that is ever deleted from a heap. Conversely, we only allow an element to be added to the rear of the heap. But even with this restricted insertion and deletion scheme, the heap condition is easily violated. Two potential problems can arise:

- First, when removing the very first element of the heap, we have to replace it by another element for otherwise the heap does not have a first element which makes the resulting data structure violate the heap condition. It would correspond to a complete binary tree that has no root. To resolve this situation, we subsequently replace the very first element by the very last element of the heap. A potential problem then is that the element that appears in the very first location like this, is too big and violates the heap condition (remember that it has to be smaller than the elements sitting at its child nodes). In other words — if we think of the heap as a complete binary tree — the element resides in "too high a level" in the heap. It therefore has to be *sifted down* a few levels. This is the task of the private procedure `sift-down` explained in section 4.4.5.

- Second, when adding a new element to the end of the heap, the element might be too small in the sense that it resides in "too low a level" in the heap. The element belongs in a higher node if we think about the heap as a complete binary tree. *Sifting up* the element a few levels is the responsibility of the procedure `sift-up` that is explained in section 4.4.5 as well.

Given the procedures `sift-down` and `sift-up`, then the implementation of `insert!` and `delete!` looks as follows:

```
(define (insert! heap item)
  (if (full? heap)
    (error "heap full" heap))
  (let* ((vector (storage heap))
         (size (size heap)))
    (vector-set! vector size item)
    (if (> size 0)
      (sift-up heap (+ size 1)))
    (size! heap (+ size 1))))

(define (delete! heap)
  (if (empty? heap)
    (error "heap empty" heap))
  (let* ((vector (storage heap))
         (size (size heap))
         (first (vector-ref vector 0))
         (last (vector-ref vector (- size 1))))
    (size! heap (- size 1))
    (if (> size 1)
      (begin
        (vector-set! vector 0 last)
        (sift-down heap 1)))
    first))
```

`insert!` adds the new element to the last location of the heap and then sifts it up. This will rearrange the heap such that it is guaranteed to satisfy the heap condition again. Similarly, `delete!` replaces the first element of the heap by the very last one. Subsequently, the new element residing in the first position is sifted down in the heap in order to make the entire vector satisfy the heap condition. The original first element is returned from `delete!`.

4.4.5 Maintaining the Heap Condition

Let us now have a look at how the heap condition is restored by the `sift-down` and `sift-up` procedures. Remember from section 4.4.1 that the indexing scheme for heaps starts at 1 even though the indexing for vectors in Scheme starts counting from 0. To bridge this difference, both `sift-up` and `sift-down` have their

own local version of `vector-ref` and `vector-set!` that perform the necessary index conversions.

`sift-up` takes an index `idx` and assumes that `idx` is the last position of the `heap`. `sift-up` takes the element residing in that last position and percolates it to a higher location in the heap (i.e. to a location closer to the root of the heap). `sift-up` is an iterative process that reads the `element` at the `idx` position and moves it up the heap by shifting down all elements it encounters in the iterative process. This process continues until the `element` to be sifted up has reached its destination or until the root of the heap is reached. At that point, the `element` is stored in the vector. In every step of the iteration, the loop `sift-iter` computes the vector index of the parent using the expression (`div child 2`). In the body of the loop, we observe a conditional. The first branch checks whether we have arrived at the root. If this is the case, the `element` is stored in the root of the heap. The second branch checks whether the heap condition would be violated if we were to store the `element` at the current location. If this is the case, we store the element at the current location one level down and we continue the sifting process. The third branch is only reached if storing the `element` at the current location does not violate the heap condition. In that case, the `element` is simply stored in the vector entry that was freed by the previous iteration of the loop.

```
(define (sift-up heap idx)
  (let
      ((vector-ref
        (lambda (v i)
          (vector-ref v (- i 1))))
       (vector-set!
        (lambda (v i a)
          (vector-set! v (- i 1) a)))
       (vector (storage heap))
       (size (size heap))
       (<<? (lesser heap)))
    (let sift-iter
      ((child idx)
       (element (vector-ref vector idx)))
      (let ((parent (div child 2)))
        (cond ((= parent 0)
               (vector-set! vector child element))
              ((<<? element (vector-ref vector parent))
               (vector-set! vector child (vector-ref vector parent))
               (sift-iter parent element))
              (else
               (vector-set! vector child element)))))))
```

`sift-down` assumes that an `element` sitting at position `idx` is the root of a heap for which all other elements are guaranteed to satisfy the heap condition. Hence, if we think of the heap as a complete binary tree, then `sift-down`

assumes that the `element` sitting at the root maybe violating the heap condition when it is compared with its children. However it assumes that the two subheaps that start at the children of the root are correct heaps. `sift-down` percolates the `element` down the heap (by shifting the other elements up) until it reaches a location where it can be stored such that the overal data structure satisfies the heap condition.

```
(define (sift-down heap idx)
  (let
      ((vector-ref
        (lambda (v i)
          (vector-ref v (- i 1))))
       (vector-set!
        (lambda (v i a)
          (vector-set! v (- i 1) a)))
       (vector (storage heap))
       (size (size heap))
       (<<? (lesser heap)))
    (let sift-iter
      ((parent idx)
       (element (vector-ref vector idx)))
      (let*
          ((childL (* 2 parent))
           (childR (+ (* 2 parent) 1))
           (smallest
            (cond
              ((< childL size)
               (if (<<? (vector-ref vector childL)
                        (vector-ref vector childR))
                   (if (<<? element (vector-ref vector childL))
                       parent
                       childL)
                   (if (<<? element (vector-ref vector childR))
                       parent
                       childR)))
              ((= childL size)
               (if (<<? element (vector-ref vector childL))
                   parent
                   childL))
              (else parent))))
        (if (not (= smallest parent))
            (begin (vector-set! vector parent (vector-ref vector smallest))
                   (sift-iter smallest element))
            (vector-set! vector parent element))))))
```

At every level in the iteration, `sift-down` compares the `element` residing at the current `parent` with the elements residing at the (two or less) children. If

the *smallest* element of this 3-way comparison resides in one of the children (i.e. (`not` (= `smallest parent`)), then that element of the child is copied into the parent and the iteration is continued with the child. Once the smallest element is the one sitting in the current parent, that parent is replaced by our element. This is safe since the element sitting in the current parent was moved up one level in the previous iteration anyhow.

`sift-down` can be considered as an operation that merges two heaps. The `element` residing at location `idx` can be thought of as the root of a complete binary tree that is not a heap yet. However, both subtrees are valid heaps. By sifting the `element` down (into one of the subtrees), the entire tree with root `idx` becomes a heap as well.

4.4.6 Heap Performance Characteristics

Now that we have presented an implementation for all the operations that can be used on heaps, it is time to establish their performance characteristics. Looking back at the implementations for `insert!` and `delete!`, we observe that their entire body is $O(1)$ except for the call to `sift-down` or `sift-up`. The implementation of `sift-down` and `sift-up` consist of expressions that are in $O(1)$ except for the loops `sift-iter` that traverse the heap. Hence, in order to come up with a performance characteristic for `insert!` and `delete!`, we have to find out how often the `sift-iter` loop is executed in both cases. In the `sift-iter` loop of `sift-up`, we notice that the number `child` is divided by two in every step of the iteration. Since we start with n (i.e. the size of the heap) the question becomes how often we can divide a number (using integer division) before we reach zero. In other words, for which k is $\lfloor \frac{n}{2^k} \rfloor = 0$? Clearly the answer is $k = \lfloor log_2(n) \rfloor$. Similarly, the `sift-iter` loop of `sift-down` starts from 1 and multiplies its iteration variable `parent` by two until the correct position in the heap is reached. Clearly, the last possible position that can be reached this way is n, the size of the heap. Hence, the question becomes how often we can double 1 before we reach n. In other words, for which k is $2^k = n$? Again the answer is $k = \lfloor log_2(n) \rfloor$. Hence, both `sift-down` and `sift-up` are in $O(log(n))$. A different way to understand this result is to notice that both `sift-down` and `sift-up` traverse the complete binary tree that corresponds to the heap. `sift-down` starts at the root and moves its way down to a leaf of the tree (in the worst-case). Conversely, `sift-up` starts at the last position (i.e. a leaf of the tree) and moves its way up to the root (in the worst-case). In both cases, the number of iterative steps is bound by the height of the tree which is $h = \lfloor log_2(n) \rfloor$ as explained in section 4.4.2.

Hence, `insert!` and `delete!` are in $O(log(n))$.

4.4.7 Building a heap

Now that we have presented the basic operations of heaps as well as a way to maintain the heap condition, we show how to build a heap given a vector of randomly ordered elements. The implementation of `from-vector` takes a vector

and a comparator <<? that is used to organise the heap. It turns the vector into a heap by packing it into an enhanced list and by reorganizing its elements.

```
(define (from-scheme-vector vector <<?)
  (define size (vector-length vector))
  (define heap (make vector size <<?))
  (define (iter index)
    (sift-down heap index)
    (if (> index 1)
      (iter (- index 1))))
  (iter (div size 2))
  heap)
```

At this point, it is useful to think about the heap as a complete binary tree again. If we consider a vector containing n elements as a complete binary tree, then the $\frac{n}{2}$ last elements of the vector form the bottom level of the tree (called the *yield* of the tree). All the elements residing in the yield are 1-element heaps by definition. Hence, these do not need to be considered in order to build the heap such that the iteration to build the heap can start in the middle of the vector (i.e. we start at (`div size 2`)). The heap construction process counts backward from the middle of the vector down to the first element of the vector. In every phase of the iteration, an element is considered as the root of a new heap that has to be built on top of two smaller heaps that result from the previous step of the iteration. Since the new root might violate the heap condition, it has to be sifted down the new heap. In terms of complete binary trees, a new complete binary tree is made based on two previously constructed complete binary subtrees.

The implementation for the `from-vector` operation calls `iter` for all $\frac{n}{2}$ elements that are not leaves in the newly built heap. `iter` calls `sift-down` for every element and `sift-down` is $O(log(n))$. Hence, $O(n.log(n))$ is a worst-case performance characteristic for `from-vector`. Although this naive analysis is correct, we get significantly better results by delving a bit deeper into the mathematical properties of a heap.

Remember from section 4.4.2 that a heap with n elements has at most $\lceil \frac{n}{2^{h+1}} \rceil$ elements residing at height h. When adding an element to the heap at height h, this causes `sift-up` to do $O(h)$ work. Since this has to be done for all nodes residing at height h, constructing the heaps at height h requires $\lceil \frac{n}{2^{h+1}} \rceil O(h)$ computational steps. This has to be done for all "heights" going from 0 to the height $\lfloor log(n) \rfloor$ of the entire heap. Hence, the total amount of work is $\sum_{h=0}^{\lfloor log(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^h})$. Since $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$, we can use this result for $x = \frac{1}{2}$ which yields $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. Hence we get $O(n \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$ which is a better result than the $O(n.log(n))$ given by the naive analysis presented above. In other words, `from-vector` builds a heap from any vector in linear time.

The performance characteristics for our entire `heap` ADT implementation is summarized in figure 4.13. Notice that our implementation of `from-vector`

destructively uses the argument vector to store the target heap. If this behaviour for `from-vector` should be unwanted, we have to make a new vector and copy the argument vector. This operation is $O(n)$ as well such that it does not affect the performance characteristic for `from-vector`.

Operation	Performance
`new`	$O(1)$
`empty?`	$O(1)$
`full?`	$O(1)$
`from-vector`	$O(n)$
`insert!`	$O(log(n))$
`delete!`	$O(log(n))$
`peek`	$O(1)$
`length`	$O(1)$

Figure 4.13: Heap Performance Characteristics

4.4.8 Priority Queues and Heaps

Remember that the main reason for studying heaps is that they provide us with an extremely efficient implementation for priority queues. Below we present the heap implementation of the `priority-queue` ADT that was presented in section 4.3.

Just like in the sorted list implementation and the positional list implementation, the heap implementation for priority queues actually stores priority queue items which are pairs that consist of an actual value and its associated priority. The implementation of priority queue items is repeated below.

```
(define pq-item-make cons)
(define pq-item-val car)
(define pq-item-priority cdr)
(define (pq-item-specialize-for-priorities func)
  (lambda (item1 item2)
    (func (pq-item-priority item1)
          (pq-item-priority item2))))
```

Representing a priority queue by means of a heap is not very different from the representation that uses sorted lists or positional lists. A priority queue is represented by an enhanced list that maintains a reference to a heap. The "higher priority than" operator `>>?` is used as the "smaller than" operator that is needed by the `heap` ADT. In other words, one priority queue item is smaller than another priority queue item whenever the first has a higher priority than the second.

```
(library
 (priority-queue)
```

```
(export new priority-queue? enqueue! peek serve! full? empty?)
(import (rnrs base (6))
        (prefix (a-d heap standard) heap:))

(define default-size 50)
(define pq-tag 'heap-pq)
(define (make >>?)
  (list pq-tag (heap:new default-size >>?)))
(define (heap pq)
  (cadr pq))

(define (new >>?)
  (make (pq-item-specialize-for-priorities >>?)))

(define (priority-queue? any)
  (and (pair? any)
       (eq? (car any) pq-tag)))

...)
```

Again, `empty?` and `full?` are trivial. They merely check whether the underlying heap is empty or full.

```
(define (empty? pq)
  (heap:empty? (heap pq)))

(define (full? pq)
  (heap:full? (heap pq)))
```

Given the abstractions provided by the `heap` ADT, the implementations of `enqueue!`, `serve!` and `peek` are quite simple. `enqueue!` creates a new priority queue item and inserts the item in the heap. The heap does the rest as it will sift the item to some position needed to satisfy the heap condition. `serve!` deletes a priority queue item from the heap (by removing its smallest element) and retrieves the value of that item. Again, the heap does the sifting necessary to make a new smallest element appear in the root of the heap. `peek` is entirely equivalent.

```
(define (serve! pq)
  (if (empty? pq)
    (error "empty priority queue (serve!)" pq))
  (pq-item-val (heap:delete! (heap pq))))

(define (peek pq)
  (if (empty? pq)
    (error "empty priority queue (peek)" pq))
  (pq-item-val (heap:peek (heap pq))))

(define (enqueue! pq value pty)
```

```
(heap:insert! (heap pq) (pq-item-make value pty))
pq)
```

The implementation uses the fact that — because of the heap conditions — the smallest element (i.e. the element with highest priority) always resides at the root of the heap. Hence, the heap implementation for priority queues is similar to the sorted list implementation: serving an element from the priority queue merely requires us to remove the first element. In a sorted list implementation this had no further implications since the rest of the list is sorted as well. In a heap implementation, removing the first elements requires us to do the sifting necessary to restore the heap property for the remaining elements. This process takes $O(log(n))$ work. Figure 4.14 is a repetition of figure 4.7. We invite the reader to look back at figure 1.5 in order to understand that this is an extremely powerful result. E.g., for enqueueing an element in a priority queue that contains one million elements, only twenty computational steps are required.

Operation	Sorted List	Positional List	Heap
`enqueue!`	$O(n)$	$O(1)$	$O(log(n))$
`serve!`	$O(1)$	$O(n)$	$O(log(n))$

Figure 4.14: Comparative Priority Queue Performance Characteristics

Chapter 5

Sorting

The act of sorting and the idea of things being sorted is omnipresent in our daily life. Phone indices are sorted by last name, train schedule entries are sorted by time of departure, books on a shelf are sorted by author and so on. The reason for this is that we humans are notoriously bad at searching data elements in collections of unsorted data. The examples also show that we are extremely good at finding data in a collection of data that is sorted. Section 3.4 of the previous chapter has shown us that the same is true for computers. Sorting a collection of data elements is probably one of the most frequently needed operations in computer science applications. E.g., in the iTunes music player one can view one's playlist sorted by artist, title, genre and so on. This is a good example that shows the need for fast sorting algorithms since a user is unwilling to wait seconds for the sorted playlist to appear on the screen. Sorting a few thousands of songs in iTunes only lasts a fraction of a second.

Sorting algorithms are classified into *internal sorting algorithms* and *external sorting algorithms*. Internal sorting algorithms are applicable when all the data to be sorted can be represented in the computer's central memory. External sorting algorithms are sorting algorithms that sort data the size of which is too big in order for all the data to be represented in central memory. When this is the case, the data is stored in files on external memory such as disks or tapes. Take for example the enormous amount of data that is continuously being generated by the instruments aboard the Hubble space telescope. The result are petabytes of data which can no longer be loaded into the central memory of a computer system. Sorting such huge data sets requires special external sorting algorithms. External sorting algorithms are much complexer than internal sorting algorithms. In this chapter, we focus on internal sorting algorithms.

One might ask the question why we have to devote an entire chapter to sorting algorithms. Why can't we just select the best sorting algorithm ever invented, teach it to our students and just forget about all the others? The answer to this question is that there is no such thing as a best sorting algorithm. Nearly all sorting algorithms have a best-case depending on the amount of data

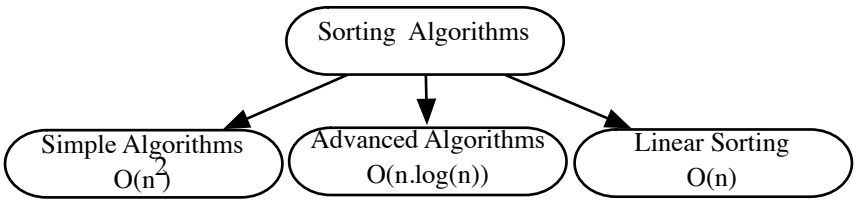

Figure 5.1: A Taxonomy of Sorting Algorithms

to be sorted, the way the non-sorted data is organized, the type of the data to be sorted, the type of hardware onto which the algorithm is deployed and so on.

Most sorting algorithms presented in this chapter operate on Scheme vectors. Many algorithms can be trivially transposed to double linked lists though. Making them operate on a single linked list is sometimes impossible because of the traverse that is needed when access is needed of one of the elements that are "to the left" of a given element. Some algorithms don't work with linked lists at all since they rely on the $O(1)$ direct indexing property of vectors. For instance, the median-of-three version of the Quicksort algorithm that is discussed in section 5.4.1 needs three elements of the list (the leftmost element, the rightmost element and the middle element) in every recursive call. This clearly requires direct indexing.

Given huge collections of data, naive sorting algorithms can be very slow. A sorting algorithm that operates on n data elements must necessarily access each and every one of those data elements in order to determine the position it will take in the newly sorted list. Therefore sorting algorithms are definitely bound to be $\Omega(n)$. As we will see in this chapter, poorly performing sorting algorithms typically compare every element with every other element in the data structure, yielding a performance characteristic that is in $O(n^2)$. By cleverly selecting the elements with which one compares an element, we are able to improve this performance characteristic up to the level of $O(n.log(n))$. By putting additional constraints on the type of data to be sorted, we even attain performance characteristics of $O(k \times n)$ or $O(k + n)$ where k is a constant that depends on the type of data elements to be sorted. These performance characteristics are used to taxonomise algorithms into *simple sorting algorithms*, *advanced sorting algorithms* and *linear sorting algorithms* as shown in figure 5.1. In what follows, we study all three categories.

5.1 Sorting Terminology

Before we proceed with the study of the actual sorting algorithms, we first introduce some terminology that is used throughout the chapter. Remember from section 1.4.3 that the data elements sitting in a storage data structure are called *records*. Usually, records are compound data elements that consist of

fields. Fields are the constituents that make up a compound data value. For example, in a phone index, ever record is a data element that consist of a name field, an address field and a phone number field. A group of fields that contain enough information to identify a record is called a *key* for the records. The fields of a key are called *key fields*. Given a key, we have enough information to find the corresponding record in the data structure. In our phone index, the names of people are usually considered as key fields since a name is enough information to identify a record. All other fields are called satellite fields. These are the fields that are not used to identify a record. In the phone index, satellite fields are the phone number and the address that are associated with the name.

Key fields are not only used to search records. They are also used to sort records in large data collections. Again, the phone index serves as an example. When creating the index, its editors have collected all the data and have subsequently sorted the data by the name of the phone index entries. Hence, the name fields are not only used for information retrieval but also for sorting that information. In fact, the reason why a searching technique like binary searching (see section 3.4.4) make sense is precisely that the key used for searching is identical to the one used for sorting. Therefore, key fields are also called *sort fields* in the literature on sorting. We use both terms interchangeably.

Most of the sorting algorithms presented in this text are iterative interplays of comparing data elements, and skipping or moving those data elements. Typically a set of loops is launched that compares data elements in a pairwise fashion. Based on the result of the comparison, those data elements can be moved to different locations in the data structure or just skipped when the data elements are already stored in their correct position. It is the perpetual comparing and moving of data elements that finally results in a sorted data structure.

Compare Since the goal of sorting is to arrange data elements "in the right order", we have to say something about the ordering used. We will not hard-code the order in our sorting algorithms. After all, the need for generic data structures explained in section 1.4.2 also holds for algorithms. Therefore, we design our sorting algorithms such that they do not depend on the type of the data elements they are sorting. This is accomplished by parametrizing the algorithms with the "ordering" procedure `<<?`. During the execution of the algorithm, the given ordering procedure `<<?` is used to compare data elements with one another. Mathematically spoken, this order is a total order. As an example, consider the following vector containing persons. A person is represented by a tiny list containing the person's name and his or her age.

```
(define persons (vector '("Paul" 41) '("Anna-Mae" 44) '("Selma" 23)
                        '("Kenny" 68) '("Roger" 41) '("Paul" 22)))
```

Given a sorting procedure `sort` that is parametrised by the vector to be sorted and the procedure for comparing persons, we sort the `persons` vector by first name as follows:

	Name	Time	Artist
1	One More Time– Aerodynamic	8:03	Daft Punk
2	Face to Face– Harder Better Faster Stronger	4:55	Daft Punk
3	Too Long	5:10	Daft Punk
4	Around the World– Harder Better Faster Stronger	7:27	Daft Punk
5	Steam Machine	1:39	Daft Punk
6	Crescendolls–Too Long– High Life	7:41	Daft Punk
7	Television Rules the Nation	2:47	Daft Punk
8	Technologic	5:29	Daft Punk
9	Superheroes– Human After All	6:13	Daft Punk
10	Da Funk	5:59	Daft Punk
11	The Brainwasher– The Primetime of Your Life– Steam Machine	12:37	Daft Punk
12	Robot Rock–Oh Yeah	6:36	Daft Punk

Figure 5.2: An mp3-playlist

```
(sort persons (lambda (p1 p2)
                (string<? (car p1) (car p2))))
```

The big advantage of parametrizing `sort` by a comparator procedure is that the same algorithm can be used to sort the `persons` vector by age. All we have to do is pass along another ordering procedure:

```
(sort persons (lambda (p1 p2)
                (< (cadr p1) (cadr p2))))
```

Move Moving about primitive data values such as numbers or booleans is not very interesting. However, whenever a compound data element has to be moved to another location in the data structure there are several options. Consider the playlist shown in figure 5.2. For the sake of the argument let's consider two radically different representations for the playlist. The first representation stores the playlist as a scheme vector (with 14 entries) that stores vectors of size 4 (track number, name of the song, time and artist). The second representation uses a vector of size 4 that stores vectors containing 14 entries. Using matrix terminology, we might say that the two representations are each other's transposition.

Now suppose that we want to sort this playlist by song title in ascending order. We use Scheme's `string<?` to compare two entries by comparing the title of the song. In the first representation, the expression used to compare two songs that sit at index `i` and `j` in the `play-list` might look as follows:

```
(string<? (vector-ref (vector-ref play-list i) 1)
          (vector-ref (vector-ref play-list j) 1))
```

In the second representation it would look as follows.

```
(string<? (vector-ref (vector-ref play-list 1) i)
```

```
(vector-ref (vector-ref play-list 1) j))
```

These expressions are each other's transposition: the position of the indexes in the vectors has been exchanged since the role of the vectors are exchanged. We observe that the change in representation doesn't affect the cost of comparing data elements. However, this is no longer true for moving elements about. Let us explain why this is the case. Moving data elements is often accomplished by swapping two data elements after having compared them. Whenever a comparison causes us to swap the songs residing at the i^{th} to the j^{th} position in the vector, then there are three options:

- In the first representation, we move the data elements *by reference*. This means that we do not change the contents of the songs but rather the references to the songs that sit in the vector representing the playlist. This is accomplished by the following code. Clearly, this is an extremely efficient solution: only 3 data values are moved.

```
(let ((song (vector-ref play-list i)))
  (vector-set! play-list i (vector-ref play-list j))
  (vector-set! play-list j song))
```

- In the second representation, things are a bit more complicated. Since our playlist consists of four columns, we have to access the individual bits of every song in the four columns explicitly. Such a "deep" way of moving data is referred to as copying data elements *by copy*. Clearly, this is much less efficient. This way of working cannot be avoided if the data happens to be organised following the second representation Scheme.

```
(let ((nr (vector-ref (vector-ref play-list 0) i))
      (title (vector-ref (vector-ref play-list 1) i))
      (time (vector-ref (vector-ref play-list 2) i))
      (artist (vector-ref (vector-ref play-list 3) i)))
  (vector-set! (vector-ref play-list 0) i (vector-ref (vector-ref play-list 0) j))
  (vector-set! (vector-ref play-list 1) i (vector-ref (vector-ref play-list 1) j))
  (vector-set! (vector-ref play-list 2) i (vector-ref (vector-ref play-list 2) j))
  (vector-set! (vector-ref play-list 3) i (vector-ref (vector-ref play-list 3) j))
  (vector-set! (vector-ref play-list 0) j nr)
  (vector-set! (vector-ref play-list 1) j title)
  (vector-set! (vector-ref play-list 2) j time)
  (vector-set! (vector-ref play-list 3) j artist)))
```

- An intermediate solution is to construct a new *index vector*. In this case, the original data structure is not changed. Instead an auxiliary vector is constructed that maps new locations (i.e. locations after applying the sort procedure) onto old locations (i.e. the locations in the data structure). After having completed the sorting procedure,

the resulting index vector contains numbers that refer to the real location of the data that is still stored in the old data structure. The new vector acts as an indexing mechanism: in order to know the title of the fourth song, we look in the fourth location of the index vector. The number found there is the real index in the original vector that contains the requested song. Creating an index vector is considered an intermediate solution since it has the benefits of copy by reference and it is a technique that is applicable even if the data is structured in the way prescribed by the second representation. The price to pay is the construction of a new vector.

Two more terms that are used frequently when studying sorting algorithms have to be defined: *stable* sorting algorithms and *in-place* sorting algorithms.

In-place sorting algorithms are sorting algorithms that require no additional memory besides the memory that is already occupied by the input data structure. In other words, we say that a sorting algorithm is in-place if the amount of memory used by the algorithm is in $\Theta(1)$. This includes the memory that may be needed to execute recursive processes such as the `fib1` example presented in section 1.5.4. In-place variants exist of all the simple sorting algorithms presented in section 5.3. The merge sort algorithm presented in section 5.4.2 will turn out not to be in-place because it requires an additional data structure to do the sorting. The famous quicksort algorithm presented in section 5.4.1 will turn out not to be in-place. Luckily, heapsort presented in section 5.4.3 is in-place.

Stable sorting algorithms are algorithms that respect the original order of records that have identical keys. Suppose that we have an iTunes playlist that is sorted by track number and suppose that we decide to sort the playlist by song title. The sorting algorithm is stable if it respects the original order of songs with the same title. E.g., suppose we have a version of the song called "Word Up" by "Korn" that has track number 5 and suppose that we have a version by "Cameo" that has track number 7. In the original playlist, the "Korn" version occurs before the "Cameo" version since the former has a smaller track number than the latter. Our algorithm is stable if this property still holds after sorting the playlist by song title. Some algorithms are easily kept stable just by choosing the right comparator. E.g., the difference between $<$ and $<=$ can make or break stability. Other algorithms are very hard to keep stable. Theoretically, it is possible to guarantee stability for every algorithm. This is achieved by artificially extending the key by including the original position of the data elements in vector. However, in practice this slows down the algorithms and it requires additional memory. We will call a sorting algorithm stable if the algorithm is stable by nature (or by using the right comparison operator) and if stability does not have to be encoded into the ordering relation this way.

5.2 Performance Characteristic Revisited

Remember the rule of thumb that was presented in section 1.5.4 for estimating the Big Oh performance characteristic of a recursive or iterative algorithm. The idea is to estimate the performance characteristic $O(b(n))$ of the algorithm's body and to estimate the number of times $O(r(n))$ the recursion will be invoked. The overal performance characteristic of the algorithm then is $O(r(n).b(n))$.

This rule implicitly assumes that the performance characteristic $b(n)$ for the body remains the same throughout every step of the recursion. In the algorithms presented in this chapter, it is very common for the expression of $b(n)$ to depend on a variable i that is percolated through the recursion. Hence, $b(i)$ is an estimation for the performance of the body when the body is being executed for the i^{th} time. When this is the case, our rule of thumb needs refinement. In the first call of the recursion, the body requires $b(1)$ number of computational steps, in the second call it requires $b(2)$ steps, and so on. Hence:

Rule of Thumb (2nd version): Suppose we have a recursive procedure that takes an argument the input size of which depends on n. First estimate $r(n)$ for the number of recursive calls which the procedure executes. Second, determine the performance characteristic $O(b(i))$ for the i^{th} time that the body of the procedure is being executed. The performance characteristic for the entire recursive procedure will be $O(\sum_{i=1}^{r(n)} b(i))$.

Our original rule of thumb presented in section 1.5.4 is a special case of this new version in which the performance characteristic $b(i)$ of the body is actually $b(n)$, i.e. only depends on n instead of i. Then we get $\sum_{i=1}^{r(n)} b(i) = \sum_{i=1}^{r(n)} b(n) = b(n).\sum_{i=1}^{r(n)} 1 = b(n).r(n)$. In many algorithms discussed so far, $b(n) = 1$ such that we got $r(n)$ as the resulting performance characteristic. I.e. the performance characteristic of the algorithm is basically nothing but the number of recursive calls. From now on, this will change since the body of the recursion itself also launches new iterations and recursive procedures.

The performance characteristics of a sorting algorithm represents the amount of computational work done by the algorithm. As explained in section 5.1 this work consists of comparing and/or moving data elements residing in the data structure to be sorted. Depending on the size of the key, comparing can be cheap or expensive. Furthermore, depending on the way data is stored, moving data can be cheap or expensive as well. Therefore, an algorithm with a small number of compares (but with lots of moves) might be preferable over an algorithm that requires lots of compares (but which requires few moves) even though their performance characteristics are identical. Similarly, an algorithm with relatively few moves might be preferable over an algorithm that moves data elements frequently. Thus, apart from establishing the performance characteristic for every algorithm, we will also try to estimate the number of compares and the number of moves that are required by an algorithm.

5.3 Simple Sorting Algorithms

Remember from figure 5.1 that different categories of sorting algorithms exist, one of which is the category of simple sorting algorithms. Simple sorting algorithms are called simple because the algorithms are easy to understand and implement. As a consequence, the algorithms are also quite naive. This gives them quadratic performance characteristics. This is in contrast to the advanced sorting algorithms that have a performance characteristic that is in $O(n.log(n))$. However, as we will show in section 5.4, the computational overhead of an advanced sorting algorithm can be so high that a simple sorting algorithm may be preferred in some cases. This is typically true for small amounts of data. Apart from didactic considerations, this is the main reason for studying simple sorting algorithms.

5.3.1 Bubble Sort

The first algorithm that we study is called *bubble sort* or *exchange sort*. Although bubble sort is one of the slowest sorting algorithms known, it is tremendously popular. This is probably a consequence of its catchy name. The only reason to study the bubble sorting algorithm is to demonstrate its bad performance and understand the deeper cause for this. This insight is useful to understand how and why other algorithms perform much better.

Bubble sort sorts the data while passing several times over the data. The main idea of bubble sort is to work backwards in the vector from the last position to the first position. For every index encountered, a pass is executed that starts at the beginning of the vector and works towards the index. During the pass, each pair of adjacent elements is compared. If the relative order of the elements is not correct, the elements are swapped. In every stage of the algorithm, the index separates the part of the vector that remains to be sorted from the part of the vector that is already sorted. The vector of unsorted elements gets shorter in every pass. The algorithm is illustrated in figure 5.3. In the figure, every pass is shown horizontally. The sequence of passes is shown vertically. Every pass makes the biggest element of the part of the vector that remains to be sorted percolate to its correct position (i.e. the location corresponding to the index). Elements that have percolated their way to their right position are indicated in boldface.

The code shown below is a Scheme implementation of bubble sort. The passes are orchestrated by an *outer loop* which perpetually starts an *inner loop*. The outer loop is realized by the `outer-loop` named `let`. In every step of the iteration, the variable `unsorted-idx` is decremented by 1. The loops starts at (− (`vector-length vector`) 2) (i.e. the penultimate element of the vector) and continues until `unsorted-idx` reaches zero in the worst case. For every value of `unsorted-idx`, the conditional of the `if` expression launches an `inner-loop` that starts from zero and that gradually grows towards the value of `outer-idx`. If that inner loop returns #t, the outer loop continues by launching the next inner loop. Once the inner loop returns #f, the outer loop stops operating. The

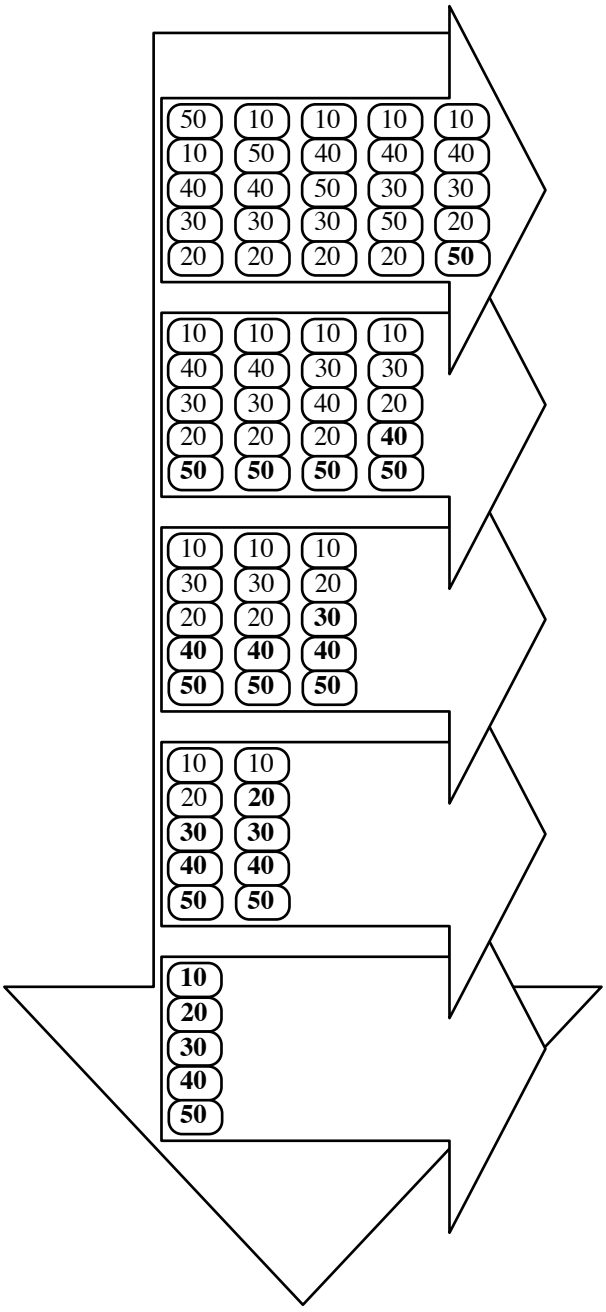

Figure 5.3: Bubble Sort

boolean returned like this is called `has-changed?` in the inner loop. It's initial value is #f and it becomes #t as soon as a call to `swap` is made. In other words, the boolean indicates whether or not at least one element was moved by the inner loop. Once the inner loop returns #f, no two adjacent elements have been swapped. This means that we have started from zero up to the beginning of the sorted list and that no elements have been encountered that are in the wrong order. This means that the vector is sorted.

```
(define (bubble-sort vector <<?)
  (define (bubble-swap vector idx1 idx2)
    (let ((keep (vector-ref vector idx1)))
      (vector-set! vector idx1 (vector-ref vector idx2))
      (vector-set! vector idx2 keep)
      #t))
  (let outer-loop
    ((unsorted-idx (- (vector-length vector) 2)))
    (if (>= unsorted-idx 0)
      (if (let inner-loop
            ((inner-idx 0)
             (has-changed? #f))
            (if (> inner-idx unsorted-idx)
              has-changed?
              (inner-loop (+ inner-idx 1)
                          (if (<<? (vector-ref vector (+ inner-idx 1))
                                   (vector-ref vector inner-idx))
                            (bubble-swap vector inner-idx (+ inner-idx 1))
                            has-changed?))))
          (outer-loop (- unsorted-idx 1))))))
```

Properties

Let us now have a look at the amount of work done by the `bubble-sort` procedure. We assume that n is the size of the input vector `vector`. The inner loop is executed

$$(n-1)+(n-2)+(n-3)+...(n-k_{end}) = \frac{1}{2}(2n-k_{end}-1)k_{end}$$

times, where k_{end} is the amount of times the outer loop is executed (i.e. the amount of times during which the execution of the inner loop keeps the value of `has-changed?` #f). In the best case, `has-changed?` already stays #f while executing the the inner loop for the very first time, i.e. $k_{end} = 1$. This happens when the input data was already sorted. In the worst case, `has-changed?` becomes #t every time the inner loop is executed, i.e. $k_{end} = n-1$. Hence, in the best case, we end up with $\frac{1}{2}(2n-2)$, which is in $O(n)$. In the worst case, we get $\frac{1}{2}n(n-1)$, which is in $O(n^2)$.

As argued, whenever possible, it is useful to look at the number of compares and moves separately since this might be relevant when comparing long keys

and when moving large data elements by copy. In the worst case, bubble sort does $\frac{1}{2}n(n-1)$ compares and $\frac{1}{2}n(n-1)$ swaps. In the best case, we get $n-1$ compares and 0 swaps. Notice that a swap consist of three moves.

A bubble sort is easily kept stable. In order to keep elements with identical keys in the original order, we have to make sure never to call `swap` for such elements. This means that the `<<?` operator passed to `bubble-sort` has to be a strict operator. If this operator yields `#t` for equal elements, then `swap` is applied for equal elements. Hence, data values with identical keys would be swapped.

For the sake of completeness we discuss the applicability of bubble sort on linked lists. For double linked lists, this is clearly no problem since the algorithms only considers adjacent data elements (i.e. it never increments and decrements indexes by more than one). For single linked lists, the situation is a bit more subtle. Bubble sort *is* applicable, but not implemented as shown here. We leave the algorithm as an exercise to the reader.

5.3.2 Insertion Sort

The second simple sorting technique we discuss is called *insertion sort*. It is the sorting technique that is performed by card players in a casino. When a player receives his cards after the croupier has distributed them, he organizes the cards in his left hand one after the other. With his right hand, he picks up the next card from the table and inserts it in the sorted position in the cards that are held in his left hand. This principle is shown in figure 5.4. The general idea of insertion sort is that — at all times during the execution of the algorithm — there is a list of data elements which have already been sorted ("the left hand"). In every iterative step of the algorithm, a data element is correctly inserted in that list. Hence the name of the algorithm.

The Scheme code that implements the algorithm is given below. Again, the algorithm consists of an outer loop that starts inner loops. The outer loop is conceived as a named `let` that binds the variable `outer-idx` to the penultimate index in the vector. The outer loop continues by decrementing `outer-idx` by 1 until the variable reaches zero. Every element encountered is used as a `current`. The idea of the inner loop is to start at (+ `outer-idx` 1) and iterate towards the end of the vector. While executing the inner loop, elements that are smaller than the `current` are shifted one position to the left. Once an element is found that is bigger than the `current` (or if we have reached the end of the vector), then the correct position of the `current` is reached. It is then simply stored in that location in the vector. In other words, all elements smaller than the `current` are moved one location to the left and the `current` is inserted in its correct position.

```
(define (insertion-sort vector <<?)
  (define (>=? x y) (not (<<? x y)))
  (let outer-loop
    ((outer-idx (- (vector-length vector) 2)))
```

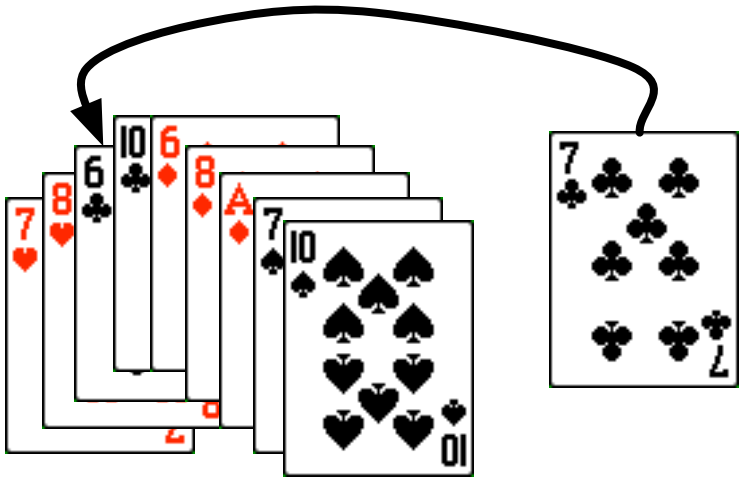

Figure 5.4: Insertionsort Illustrated

```
(let
    ((current (vector-ref vector outer-idx)))
  (vector-set!
   vector
   (let inner-loop
     ((inner-idx (+ 1 outer-idx)))
     (cond
       ((or (>= inner-idx (vector-length vector))
            (>=? (vector-ref vector inner-idx)
                 current))
        (- inner-idx 1))
       (else
        (vector-set! vector (- inner-idx 1) (vector-ref vector inner-idx))
        (inner-loop (+ inner-idx 1)))))
   current)
  (if (> outer-idx 0)
    (outer-loop (- outer-idx 1))))))
```

Properties

Let us derive the performance characteristics for insertion sort. From the code, we observe that the outer loop is always executed $n-1$ times: the `outer-idx` varies from $n-2$ down to 0. In every iteration of the outer loop, the inner loop is launched. The amount of times the inner loop is executed depends on:

- The value of `outer-index`. This value gets smaller and smaller. It indi-

cates the starting index of the list of already sorted elements which gets longer and longer. Therefore, more and more time can be spent in the inner loop as the outer loop proceeds.

- The order of the elements of the original input vector. The inner loop inserts an element in the sorted list. This requires a lot of work if the "big" elements are inserted last. In other words,

  best-case: If the original input vector is already sorted, then the inner loop is never really executed. The overall performance characteristic of insertion sort then is $O(n)$ which is entirely due to the outer loop.

  worst-case: If the original input vector is in reverse sorted order, then the inner loop is executed i times during the i^{th} execution of the outer loop. Hence the body of the inner loop is executed $\sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) = O(n^2)$ times.

  average-case: If the original input vector is randomly distributed, then the inner loop is executed $\frac{i}{2}$ times on the average (during the i^{th} execution of the outer loop). Hence, the total number of computational steps performed is $\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{4}n(n-1) = O(n^2)$.

Let us have a look at the share taken by compares and moves in these performance characteristics. In the inner loop, every time an element is compared, it is moved as well (in order to shift an element one position to the left). An exception to this rule is the last time two elements are compared. But in that case, we have one additional move to store the `current` element coming from the outer loop . When the list is already sorted, only 1 compare is done in the inner loop (that yields #f). Nevertheless, the contents of the `current` variable is immediately stored in the same position again. Hence, the number of moves is identical to the number of compares in insertion sort.

Similar to bubble sort, insertion sort is an algorithm that is easily kept stable by using a strict comparator `<<?` that returns #f on identical elements. Once a value for `inner-idx` is encountered that corresponds to an element that is equal or bigger than the value of `current`, then (− `inner-idx` 1) is returned. Hence, the value of `current` is inserted *before* that index. Hence, equal elements never "jump over each other". Therefore insertion sort is a stable algorithm.

Insertion sort is usually described as a sorting algorithm that is particularly well-suited to work on linked lists. Insertion sort is a so-called *online algorithm*: it can sort a list of elements as it receives them (e.g. from a network connection). Upon reception of new elements, they are inserted into the list of sorted elements. Remember from the section on sorted lists (see 3.4.3) that the technique used in the `add!` procedure is all about inserting an element into a sorted list. This is exactly the operation needed by an insertion sort. We invite the reader to implement an insertion sort for single linked lists based on this implementation. Notice that by cleverly designing the algorithm, *the number of moves can be reduced to zero!*

Insertion sort is generally considered to be the best simple sorting technique. It is quite efficient when the number of data elements to be sorted is small which makes it a good candidate to take over from an advanced sorting algorithm in those cases since the computational overhead of these advanced sorting algorithms causes them to be slower than the "slow" insertion sort for such small data sets (something around 10). Insertion sort performs well when the data is already sorted or when data is "nearly sorted". This property causes the inner loop to execute a small number of times because the elements arrive in the inner loop after the bigger elements have already been inserted. Hence, the inner loop will find the correct position very soon.

Priority Queue Implementation

A alternative implementation of insertion sort uses a priority queue.

The algorithm is extremely simple. We consider all the elements of the vector from left to right, and we enqueue them in a priority queue. By selecting the implementation of a priority queue that relies on sorted lists[1], the elements are inserted (hence the similarity with insertion sort) in the priority queue in the correct order. The priorities to be used in this scheme are the key fields of the data elements themselves: a data element has higher priority than a second one if it is "smaller" than the second one. After having inserted all the elements like this, all we have to do is serve the elements one by one from the priority queue. In contrast to the other implementations of insertion sort, this algorithm is not in-place since we need an additional priority queue of size n to get the elements sorted. Hence, the algorithm consumes additional memory in the order of $\Theta(n)$.

In order to understand the performance characteristic of this implementation of insertion sort, we refer to the table in figure 4.7 in section 4.3.4. At first sight, this is good news as we only see $O(n)$ in this table. However, keep in mind that the process of enqueueing (at the cost of $O(n)$) has to be done for *every* element in the vector to be sorted, i.e. n times. Hence we get $O(n^2)$ to construct the priority queue. Then we need another $O(n)$ to empty the priority queue. Together this yields a performance characteristic of $O(n^2)$.

5.3.3 Selection Sort

The final simple sorting algorithm that we discuss is called *selection sort*. Although insertion sort is the best *general* algorithm among the simple algorithms, selection sort can sometimes outperform insertion sort when the data elements to be moved are very big.

In some respect, selection sort is exactly the opposite of insertion sort. The outer loop maintains a list of sorted items which grows in every iteration. In every step of the outer loop, the inner loop is started to pick the smallest element

[1]The reader might wonder why the version based on sorted lists was selected instead of the faster implementation that uses heaps. The version that uses heaps is exactly the heap sort that is presented in section 5.4.3. Insertion sort is obtained by taking the priority queue version that is based on sorted lists.

from the shrinking list of unsorted elements. The smallest element is removed from that list and added to the rear of the list of sorted elements. Selection sort is exemplified in figure 5.5. The outer loop is depicted vertically. The task of the inner loop (depicted horizontally) is to select the smallest element. In every iteration, the current smallest element is printed in boldface. The vector entry under consideration is drawn with a thick stroke.

The Scheme implementation of selection sort follows. Again, it consists of an outer loop that starts an inner loop in every step of the iteration. This is one of the typical characteristics of simple sorting algorithms. The outer loop defines a variable `outer-index` which varies from 0 to $n-1$. In every step of the iteration, the current value residing at `outer-index` is swapped with the smallest element the index of which is returned by the inner loop. The inner loop starts from (+ `outer-idx` 1) and traverses all elements until the end of the vector is reached. The goal is to look for the smallest element. The variable `smallest-idx` remembers the index of the current smallest index. Every time an element is encountered that is smaller than the current smallest element, the inner loop is continued with the value of `inner-idx` as the new value for `smallest-idx`. When the end of the vector is reached, the value of the smallest index is returned.

```
(define (selection-sort vector <<?)
  (define (swap vector i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (let outer-loop
    ((outer-idx 0))
    (swap vector
          outer-idx
          (let inner-loop
            ((inner-idx (+ outer-idx 1))
             (smallest-idx outer-idx))
            (cond
              ((>= inner-idx (vector-length vector))
               smallest-idx)
              ((<<? (vector-ref vector inner-idx)
                    (vector-ref vector smallest-idx))
               (inner-loop (+ inner-idx 1) inner-idx))
              (else
               (inner-loop (+ inner-idx 1) smallest-idx)))))
    (if (< outer-idx (- (vector-length vector) 1))
      (outer-loop (+ outer-idx 1)))))
```

Properties

Given a vector of n entries, then the `outer-idx` varies between 0 and $n-1$. For every such index `inner-idx`, we iterate between `outer` $-$ `idx` $+1$ and $n-1$.

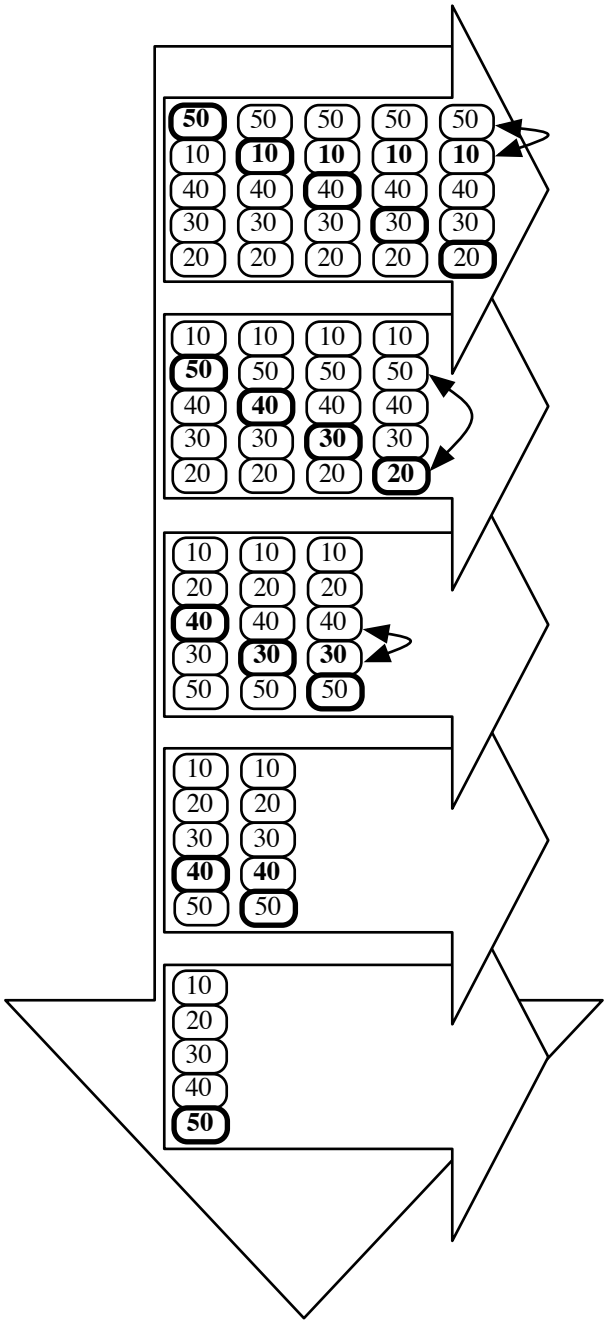

Figure 5.5: Selection Sort

Hence, the work done by selection sort is:

$$\sum_{\texttt{outer-idx}=0}^{n-1} \left(\sum_{\texttt{inner-idx}=\texttt{outer-idx}+1}^{n-1} 1 \right)$$

which is

$$\sum_{\texttt{outer-idx}=0}^{n-1} (n - 1 - \texttt{outer} - \texttt{idx}) = \frac{1}{2}(n^2 - n) = O(n^2)$$

As expected with a system of inner loops and outer loops, selection sort has a quadratic performance characteristic. A drawback of selection sort is that it performs a quadratic number of compares, irrespective of the initial ordering of the input vector. This is because the inner loop has to look for the smallest element in the list of elements to be sorted. This requires the inner loop to traverse the entire list every time again. Hence we have $\frac{1}{2}(n^2 - n)$ compares irrespective of the initial ordering. This is in contrast to insertion sort, which only traverses the sorted list up to the point where the element has to be inserted. A positive property of selection sort is that the work done by the inner loop merely consists of comparing elements. In every execution of the outer loop, the element of the outer loop is swapped with the smallest element found in the inner loop. Hence, the outer loop will perform exactly $n - 1$ swaps (i.e. 3(n-1) moves). This makes selection sort particularly well-suited for data sets with small key fields and with large satellite fields that are expensive to move about by copy.

Just like the other simple sorting algorithms, selection sort is clearly in-place. However, selection sort is not a stable sorting algorithm. The reason for this is as follows. In every phase of the sort, we select the smallest element from the list of elements that remain to be sorted and we swap that element with the element under consideration in the outer loop. Hence, the element considered by the outer loop will move to the location of the smallest element. This might cause that element to "jump over" an identical element that resides somewhere in the list of elements that remain to be sorted. Hence, preservation of the relative order of identical elements is not guaranteed.

Priority Queue Implementation

In section 5.3.2 we have shown the relation between insertion sort and priority queues. There exists a similar relation between selection sort and priority queues. If we repeat the priority queue version of insertion sort algorithm of section 5.3.2 using a positional list implementation instead of using a sorted list implementation, we obtain a version of selection sort.

The algorithm inserts the elements of the vector in a priority queue and subsequently retrieves them in sorted order as guaranteed by the specification of the priority queue ADT. If we select the positional list implementation of priority queues this gives us $O(1)$ as explained in figure 4.7 of section 4.3.4. This

has to be done for all n elements which gives us a cost of $O(n)$ for building the priority queue. Subsequently, the priority queue is emptied using `serve!` which searches for the element with the highest priority (i.e. the smallest element in our case). This is clearly an implementation of the selection sort algorithm since `serve!` *selects* the smallest element from the priority queue. Serving an element costs $O(n)$ and this has to be done *for each* element in the priority queue. Hence, the overal cost of the sorting algorithm is $O(n^2)$ again.

5.3.4 Summary

The table in figure 5.6 summarizes the overall performance characteristics obtained from our study of simple sorting algorithms. The table in figure 5.7 separately compares the number of moves M and the number of compares C for the three algorithms.

Algorithm	Worst-Case	Best-Case
Bubble Sort	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$

Figure 5.6: Comparative Overview of Simple Sorting Algorithms (1)

Algorithm	M and C (worst case)	M and C (best case)
Bubble Sort	$M \in O(n^2) \quad C \in O(n^2)$	$M \in O(0) \quad C \in O(n)$
Insertion Sort	$M \in O(n^2) \quad C \in O(n^2)$	$M \in O(n) \quad C \in O(n)$
Selection Sort	$M \in O(n) \quad C \in O(n^2)$	$M \in O(n) \quad C \in O(n^2)$

Figure 5.7: Comparative Overview of Simple Sorting Algorithms (2)

5.4 Advanced Sorting Algorithms

As explained in figure 5.1, sorting algorithms are classified into three groups depending on the nature of their performance characteristic. Simple algorithms are all characterised by the fact that their performance characteristic is quadratic and by the fact that their code consists of a system of inner loops and outer loops that is straightforward to read. This section presents a number of advanced sorting algorithms. The reason why we call them advanced is that they have performance characteristics in $O(n.log(n))$ in the average case. The price to pay is a more complex algorithm. We begin with quicksort.

5.4.1 Quicksort

Quicksort is generally considered to be the most attractive general purpose sorting algorithm. It was invented in 1960 by C.A.R. Hoare. Quicksort has an attractive average performance characteristic which is in $O(n.log(n))$ while still being relatively easy to program.

Basic Variant

Quicksort is a recursive sorting algorithm. The basic idea is as follows. Given a vector of n elements, one of the elements — called the pivot element — is selected from the vector. This constitutes the first phase of the algorithm. In the second phase of the algorithm, called the *partitioning phase*, the vector is linearly traversed in order to put the pivot element in its correct position, to put all elements which are smaller than the pivot element to the left of the pivot element, and to put all elements which are greater than the pivot element to the right of the pivot element. In the third phase, the quicksort algorithm is recursively applied to the elements left of the pivot element, as well as to the ones right of the pivot element. Since the pivot element is already in sorted position and since all elements left of the pivot as well as the ones right of the pivot element are sorted (because of the recursion), the overall vector will be sorted. We have illustrated the process in figure 5.8.

1. In the first phase, we choose the very first element of the list to be sorted as the *pivot element*. Figure 5.8 illustrates how the very first element in the vector is chosen as the pivot element in every phase of the recursion.

2. In the second phase, the list is partitioned into two halves. All elements smaller than the pivot element are put in the first half. All elements greater than the pivot element are put in the second half. In figure 5.8, the result of this process is depicted by drawing the correct location of the pivot element with a thick stroke. The pivot element is put in its correct position.

3. In the third phase, the quicksort algorithm is recursively called on the two halves that result from the partitioning phase.

Quicksort is a member of a family of algorithms which are called *divide and conquer algorithms*. Divide and conquer algorithms generally consist of three steps:

Divide The problem P is divided into two or more similar subproblems p_1, ..., p_k.

Conquer The algorithm is recursively called for some or all subproblems p_1, ..., p_k. This results in solutions s_1, ..., s_k for each of the subproblems.

Combine The solutions for the subproblems s_1, ..., s_k is subsequently combined in order to compose a solution S for P.

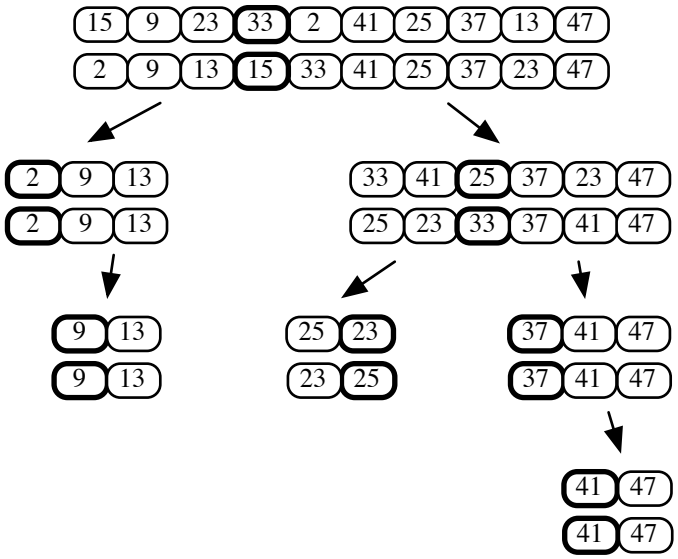

Figure 5.8: Quicksort

Quicksort applies this principle by dividing the list of elements to be sorted into two sublists that are separated by the pivot element, and by calling the algorithm on both sublists. The binary searching algorithm presented in section 3.4.4 is another example of a divide and conquer algorithm.

The code for quicksort is shown below. The recursive procedure `quicksort-main` is responsible for sorting the vector entries between the indices `l` and `r`. The recursion stops when it is no longer the case that ($<$ `l r`). The `pivot` element is chosen to be the first element, i.e. the element sitting at location `l`. As explained, the `partition` procedure moves elements smaller than the `pivot` element to the left of that pivot element. Elements greater than the pivot element are moved to the right of the pivot element. To accomplish this, `partition` traverses the vector from left to right and from right to left. Two indices `i` and `j` are used for this. All elements on the left hand side of the vector that are smaller than the pivot element are simply skipped. This is what `shift-to-right` does. Similarly, all elements on the right hand side of the vector that are greater than the pivot element are skipped as well. That is the task of `shift-to-left`. After calling these two procedures, the element sitting at index `shifted-i` is greater than the pivot element and the element sitting at index `shifted-j` is smaller than the pivot element. Both elements are swapped and then the process of moving `i` to the right and moving `j` to the left continues. When both indices bump into each other, the `partition` iteration stops.

Shifting the indices to the right (resp. to the left) continues until an element is found that is bigger (resp. smaller) than the pivot element. If such an element

would not be found, then `shift-to-right` and `shift-to-left` procedures can shift the indices too far (i.e. beyond the boundaries of the vector). In order to avoid this, we make sure such that an element will be found by using a sentinel (see section 3.4.2). That is why two elements are swapped in `quicksort-main` (*before* the partitioning phase starts) whenever the first element is bigger than the last. By swapping those elements, the first element is guaranteed to be smaller than the last element. As a consequence, `shift-to-right` is guaranteed to find an element that is bigger than the pivot element (since that is the first element). Similarly, `shift-to-left` is guaranteed to find an element smaller than or equal to the pivot element since the pivot element.

```
(define (quicksort vector <<?)
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
      (shift-to-right (+ i 1) x)
      i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
      (shift-to-left (- j 1) x)
      j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
  (define (quicksort-main l r)
    (if (< l r)
      (begin
        (if (<<? (vector-ref vector r)
                 (vector-ref vector l))
          (swap l r))
        (let ((m (partition (vector-ref vector l) (+ l 1) (- r 1))))
          (swap l m)
          (quicksort-main l (- m 1))
          (quicksort-main (+ m 1) r)))))
  (quicksort-main 0 (- (vector-length vector) 1)))
```

The drawing shown in figure 5.8 shows the structure of the recursion that is generated by running the above procedure. In every phase of the recursion, a call to `partition` is made in order to linearly traverse part of the vector. The

amount of times this is done depends on the depth d of the recursion. Suppose that every level of the recursion finds a pivot element that has a number of elements smaller than the pivot which is equal to the number of elements bigger than the pivot element. If that were the case, then every level of the recursion splits the vector in two equal halves. Hence, the recursion depth d is the amount of times the vector of size n can be split in two before a trivially sorted vector of size one is obtained. Hence we are looking for d such that $\lfloor \frac{n}{2^d} \rfloor = 1$. In other words, $d = log_2(n)$. Hence, given the assumption of always encountering pivot elements that perfectly split the vector in two halves, quicksort has a recursion depth that is $log_2(n)$.

Let us now calculate the total amount of computational work done by quicksort. Consider $W_i(n)$ as the amount of compares and swaps done in the i^{th} level of the recursion. Clearly $W_0(n) = n$ since the entire vector is traversed. $W_1(n) = n - 1$ because the first level of the recursion has to traverse (in two halves) all elements except for the pivot element that was already put in its sorted position in the zeroth level. The first level generates two new pivot nodes yielding $W_2(i) = n-3$. In general, the i^{th} level of the recursion generates $W_i(n) = n - (2^i - 1)$ amounts of work. If the height of the recursion tree is $log_2(n)$, then the total amount of work is the accumulation of the work done at all levels of the recursion, i.e. $\sum_{i=0}^{log_2(n)} W_i(n)$. Hence,

$$
\begin{aligned}
\sum_{i=0}^{log_2(n)} W_i(n) &= \sum_{i=0}^{log_2(n)} n - (2^i - 1) \\
&= \sum_{i=0}^{log_2(n)} n - \sum_{i=0}^{log_2(n)} 2^i + \sum_{i=0}^{log_2(n)} 1 \\
&= n.log_2(n) - \left(\frac{2^{log_2(n)+1} - 1}{2 - 1} \right) + log_2(n) \\
&= n.log_2(n) - \left(2.2^{log_2(n)} - 1 \right) + log_2(n) \\
&= n.log_2(n) - (2.n - 1) + log_2(n) \\
&= O(n.log(n))
\end{aligned}
$$

Notice that this is quicksort's best-case performance characteristic since we have assumed a perfect partitioning causing a minimal recursion depth.

However, things can go seriously wrong with quicksort. Suppose that the input vector is already sorted. In that case, every phase of the recursion will encounter a pivot element the sorted position of which is exactly the position at which it is sitting: the first position of the vector to be sorted. This means that when looking for a partition between `l` and `r`, the result will always be `m=l`. Hence, the left recursive call of quicksort will have no work to do. The right recursive call will have to sort a vector of size $n - 1$. Let us now have a closer look at the structure of quicksort's recursion process in this case. It is depicted in figure 5.9. As can be observed in the figure, the recursion tree is

completely *degenerated*. The depth of the recursion is n since only one element is "removed" from the vector to be sorted in every phase of the recursion. As a result, the summations shown above no longer run from 0 until $log_2(n)$ but rather from 0 until n instead. By redoing all calculations using this boundary, it is easily shown that the worst-case performance characteristic of quicksort is $\Theta(n^2)$. We leave it to the reader to verify that this performance characteristic occurs both when the elements are sorted and when they are sorted in inverse order.

From the results obtained so far we can state that the choice of a good pivot element is crucial to the efficient execution of quicksort. The more "in the middle" the pivot element, the better the partitioning. For bad partitionings, one of the recursive calls generates as good as no useful work, while the other recursive call generates an amount of work that is close to $n - 1$. In the case, of a good partitioning, both recursive calls produce a vector the size of which is close to $\frac{n-1}{2}$. In order to find an average performance characteristic, we assume a fair mixture of good and bad partitionings. I.e., we assume that in half of the times, the pivot element ends up "somewhere in the middle" of the vector to be sorted. In the other half of the times, the pivot element's target location is nearby one end of the vector which is causes a bad partitioning. Suppose that we have a worst-case partitioning that is followed by a best-case partitioning. The two consecutive splits generate three vectors, the sizes of which are 1, $\frac{(n-2)}{2}$ and $\frac{(n-2)}{2}$. When this happens regularly, the resulting recursion tree is twice as deep as the best-case recursion tree. In reality the situation is even better. It can be shown that, on the average (i.e. on randomly generated data sets), quicksort makes about 80% good splits and 20 % bad splits. This means that about a fifth of the partitionings are bad ones. Hence, in the average case, the recursion tree will be about one fifth deeper than the best-case recursion tree. It is very important to understand that this percentage does *not* depend on n. Therefore, the resulting average-case performance characteristic is still $O(n.log(n))$ but in which a bigger constant factor is hidden.

Unfortunately, the worst-case scenario for quicksort is not a theoretical situation that occurs only seldomly. Quite often, data has to be sorted that is already sorted or that is in inverse sorted order. In what follows, we present two variations of the quicksort algorithm that avoids this problem.

Randomized Quicksort

The first variation of the quicksort algorithms is known as *randomized quicksort*. The idea of randomized quicksort is to shuffle the input vector such that the chances for the worst-case behaviour are reduced to an absolute minimum. Suppose that the data elements in the vector to be sorted are randomly ordered before the quicksort algorithm is applied. The general performance of the algorithm will be the average-case performance characteristic, unless the random ordering happens to evoke the worst-case orderings. Fortunately, the chances are extremely small that a random ordering turns out to be one of the worst-case orderings. From the $n!$ possible orderings of n elements, there are not that

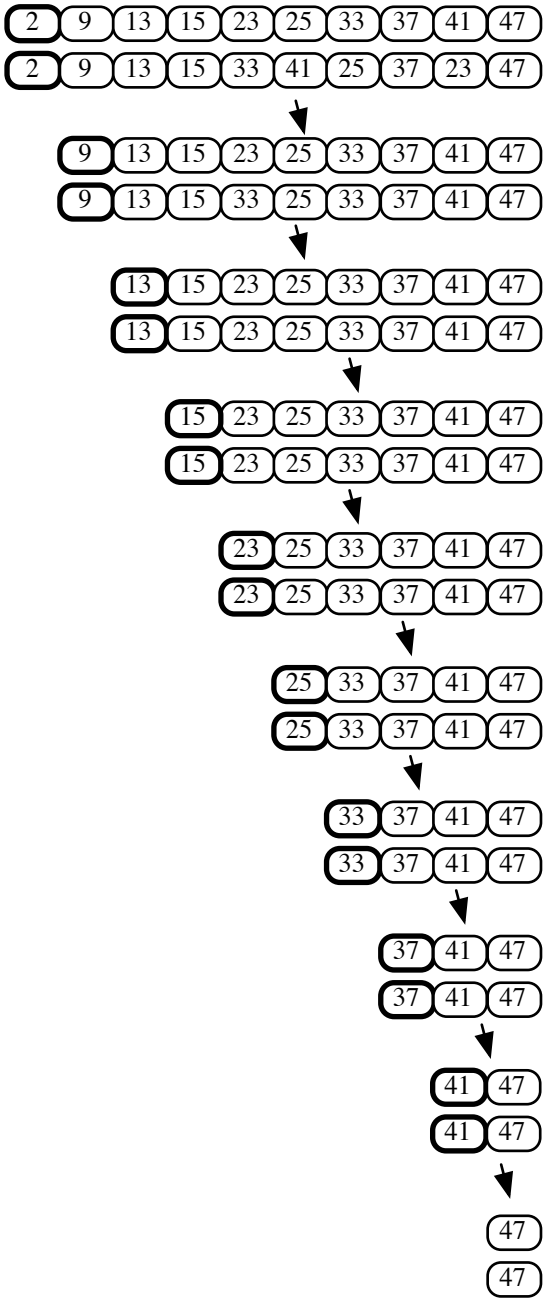

Figure 5.9: Degenerated Quicksort

many that generate a worst-case behaviour.

The code for randomized quicksort is shown below. Instead of having an additional pass to reorder the elements randomly *before* quicksort is launched, this is done *while* quicksort is operating. To enable this, every call to `partition` swaps the first element (i.e. the pivot of the original algorithm) with an arbitrary data element from the input vector (between `l` and `r`). This is accomplished in the `randomized-partition` procedure. Notice that the sentinel technique discussed above is used as well.

The code uses the Scheme function[2] (`random-integer x`) which generates a random number between 0 and $x - 1$. Hence, (+ `l` (`random` (+ (− `r l`) 1))) generates a random number between l and r.

```
(define (quicksort vector <<?)
  (define (random-inbetween l r)
    (+ l (random-integer (+ (- r l) 1))))
  (define (swap i j)
    (let ((temp (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j temp)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
      (shift-to-right (+ i 1) x)
      i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
      (shift-to-left (- j 1) x)
      j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
  (define (randomized-partition l r)
    (swap l (random-inbetween l r))
    (if (<<? (vector-ref vector r)
             (vector-ref vector l))
      (swap l r))
    (partition (vector-ref vector l) (+ l 1) (- r 1)))
  (define (quicksort-main vector l r)
    (if (< l r)
      (let ((m (randomized-partition l r)))
        (swap l m)
```

[2]This is a "standard" function that is found in the SRFI.

```
      (quicksort-main vector l (- m 1))
      (quicksort-main vector (+ m 1) r))))
  (quicksort-main vector 0 (- (vector-length vector) 1)))
```

Median of Three Quicksort

The idea of randomized quicksort is to randomly select a pivot element from the list to be sorted. A slightly better version of quicksort pushes luck a bit into the right direction by selecting a pivot element in a more controlled manner. The improved version is based on the "median" concept from statistics. The median of three is the middle of three elements that are ordered in ascending order. The median of three modification of quicksort partitions the vector to be sorted by taking the median of the leftmost element, the rightmost element and the middle element as the pivot element. The code is shown below.

```
(define (quicksort vector <<?)
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
      (shift-to-right (+ i 1) x)
      i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
      (shift-to-left (- j 1) x)
      j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
  (define (m3-partition l r)
    (let ((middle (div (+ l r) 2)))
      (swap middle (+ l 1))
      (if(<<? (vector-ref vector l)
              (vector-ref vector (+ l 1)))
        (swap l (+ l 1)))
      (if (<<? (vector-ref vector r)
               (vector-ref vector (+ l 1)))
        (swap r (+ l 1)))
      (if (<<? (vector-ref vector r)
               (vector-ref vector l))
```

```
      (swap l r))
    (partition (vector-ref vector l) (+ l 1) (- r 1))))
(define (quicksort-main vector l r)
  (if (< l r)
    (let ((m (m3-partition l r)))
      (swap l m)
      (quicksort-main vector l (- m 1))
      (quicksort-main vector (+ m 1) r))))
(quicksort-main vector 0 (- (vector-length vector) 1)))
```

The `m3-partition` procedure considers the elements at positions `l`, `r` and `middle`. The median of three of those three elements is stored at position `l` and used as the pivot element. The smallest of those elements is stored in location `l+1` and the biggest one in location `r`. These elements act as sentinels again. The element that was originally sitting at position `l+1` is stored in the space freed up in location `middle`. From that point on, the classic quicksort algorithm is applied: the median element is selected as the pivot element and the vector is further partitioned between (+ `l` 1) and (− `r` 1).

The median of three modification produces a perfectly balanced recursion tree when the input vector is already sorted or in inverse sorted order. This is because, at every level of the recursion, the middle element is selected as the pivot element which causes exactly as much elements to be smaller than the pivot element as the number of elements that is bigger than the pivot element. Hence, the worst-case input for the standard quicksort algorithm becomes the best-case input for the median of three modification.

A third improvement

A final improvement for quicksort stems from the fact that the computational overhead for quicksort is quite high. Looking at the algorithm, there are many arithmetic operations, index comparisons, and especially (recursive) procedure calls to generate the iterations. For vectors the size of which is smaller than a certain constant (which is usually around 10), this computational overhead gets bigger than the amount of actual sorting work done by the algorithm. As a result it is more advantageous to switch from quicksort to a simpler sorting algorithm — such as insertion sort — once the size of the vector to be sorted gets small enough. We leave the implementation of this algorithm as an exercise to the reader.

Properties

Quicksort is not stable. This is not hard to fathom. Consider a vector containing numbers that has two identical versions of "15", namely 15 and 15′. Suppose the vector is structured as follows:

#(20 `all-smaller-than-20` 50 ... 15 ... ... 15' all-greater-than-20)

In this input vector, 15 is located to the left of 15′. 20 is the pivot element provided that we use the default quicksort algorithm. While partitioning the vector, all elements located inbetween the first element and the occurrence of 50 will remain in their original locations. The same is true for all elements that are to the right of 15′. But then 50 and 15′ are swapped since 15′ is smaller than 20 and since 50 is bigger than 20. As a result, the relative ordering of 15 and 15′ is changed which clearly demonstrates that quicksort is not stable.

Quicksort is often said to be in-place but that is not entirely true. Although it does not require additional space to store vector elements, quicksort is a truly recursive process which means that additional memory is required to execute the algorithm. This memory corresponds to the recursion depth of the algorithm. Hence, in the best case, quicksort consumes $\Omega(log(n))$ memory. In the worst-case, this can grow up to $O(n)$. The recursive nature of the algorithm means that this cannot be circumvented without changing the algorithm considerably. It *is* possible to reduce the amount of space needed by manually maintaining a stack of recursive calls. Nevertheless, this stack will also grow in a logarithmic way.

It is easy to make quicksort operate on doubly linked lists as long as the vanilla variant is chosen. Using variations of the algorithm (i.e. randomization and median of three) is not possible since they require direct access into the list to be sorted. Applying quicksort to single linked list representations is not simple either since the shifting process requires the possibility to move through the list in both directions.

5.4.2 Mergesort

Mergesort is another advanced sorting technique that is — just like quicksort — based on the divide and conquer principle. Mergesort is slightly simpler to program than quicksort. The algorithm was invented in 1945 by J. von Neumann. It has an $O(n.log(n))$ performance characteristic as well. The downside is that its basic variant is not in-place and it is far from simple to make it in-place. A nice property of mergesort is that it is a stable sorting algorithm.

```
(define (merge-sort vector <<?)
  (define (merge vector p q r)
    (let ((working-vector (make-vector (+ (- r p) 1))))
      (define (copy-back a b)
       (vector-set! vector b (vector-ref working-vector a))
        (if (< a (- (vector-length working-vector) 1))
          (copy-back (+ a 1) (+ b 1))))
      (define (flush k i until)
        (vector-set! working-vector k (vector-ref vector i))
        (if (< i until)
          (flush (+ k 1) (+ i 1) until)
          (copy-back 0 p)))
      (define (merge-iter k i j)
```

```
      (cond ((and (<= i q) (<= j r))
             (let ((low1 (vector-ref vector i))
                   (low2 (vector-ref vector j)))
               (if (<<? low1 low2)
                   (begin
                     (vector-set! working-vector k low1)
                     (merge-iter (+ k 1) (+ i 1) j))
                   (begin
                     (vector-set! working-vector k low2)
                     (merge-iter (+ k 1) i (+ j 1))))))
            ((<= i q)
             (flush k i q))
            (else
             (flush k j r))))
    (merge-iter 0 p (+ q 1))))
(define (merge-sort-rec vector p r)
  (if (< p r)
      (let ((q (quotient (+ r p) 2)))
        (merge-sort-rec vector p q)
        (merge-sort-rec vector (+ q 1) r)
        (merge vector p q r))))
(merge-sort-rec vector 0 (- (vector-length vector) 1))
vector)
```

Just like quicksort, mergesort is a divide-and-conquer algorithm. The algorithm divides the vector in two halves and recursively calls itself for both halves. We will call these halves (and all halves of halves of ...) *regions* in the vector. The recursion terminates when the length of the region to be sorted is 1. Then the backtracking process begins. A region of size 1 in a vector is trivially sorted. At subsequent levels in the backtracking process, we may assume that we have two regions of length $n/2$ that were sorted by the previous backtrack from `merge-sort-rec`. The idea is to merge these regions (using the procedure `merge`) in order to build a new region in the vector that is sorted and the length of which is n (i.e. $n/2 + n/2$). This is accomplished by the `merge-iter` procedure via an auxiliary "working vector". `merge-iter` manages three indices. `k` is the index in the working vector and is incremented every time an element is added to the working vector. `i` is the index pointing into the first region and `j` is the index that points into the second region. In the merge process, the smallest element is selected from both regions and stored into the working vector. The index in the vector from which the smallest element was selected is systematically incremented. At the end, `flush` copies the elements that remain in one of both regions into the working vector as well. The process is terminated by calling `copy-back` in order to copy the working vector back into the space originally occupied by both subregions. The result is a region in the original vector in which all elements are sorted.

Since mergesort divides each region into two perfect halves in every phase of the recursion, the recursion tree that is generated by `merge-sort-ref` is perfectly balanced. Its height is $log(n)$. Mergesort's recursion tree is shown in figure 5.10. In the "calling phase" of the recursive process, mergesort does not do any real work: the procedure merely calls itself until regions of size 1 need to be sorted. It is during the "backtracking phase" that mergesort does the actual work: in every level of the recursion, `merge` merges two regions in the vector. If we take the example shown in figure 5.10, then we observe that, at the deepest level of the backtracking phase, 8 merges of regions of size 1 are performed. In the next level of the backtracking phase, 4 merges of size 2 are performed. Subsequently, we have 2 merges of regions of size 4. The backtracking process ends with 1 merge of two regions of size 8. Let us use $W_i(n)$ to denote the amount of work done at the i^{th} level of the backtracking phase. Clearly, i varies between 1 and $log_2(n)$ where $i = 1$ corresponds to the deepest level in the recursion; i.e. the first backtracking step. If merging two regions at level i requires $M_i(n)$ work, then $W_i(n) = 2^{log_2(n)-i}M_i(n)$ since the number of regions to merge decreases at every level. In the first backtracking step, we merge 2 regions of size 1 to a region of size 2 and we do this $8 = 2^{4-1} = 2^{log_2(16)-1}$ times. In the second step, we merge 2 regions of size 2 to a region of size 4 and we do this $4 = 2^{4-2} = 2^{log_2(16)-2}$ times. The total cost of merge sort is the accumulation of all the work done at all levels of the backtracking phase, i.e. $\sum_{i=1}^{log_2(n)} W_i(n) = \sum_{i=1}^{log_2(n)} 2^{log_2(n)-i}M_i(n)$. Merging a region at level i causes two regions of length 2^{i-1} to be merged into one region of length 2^i. In general, this causes 2^{i+1} moves (since every element has to be moved twice) and between 2^{i-1} and $2^i - 1$ compares (since, in the worst case, every element in one region has to be compared with every element in the other region, except for the last element). Therefore, $M_i(n) \leq 2^{i+1} + 2^i - 1 \leq 2^{i+1} + 2^i \leq 2^{i+2}$. Therefore, mergesort requires $\sum_{i=1}^{log_2(n)} 2^{log_2(n)-i}M_i(n) \leq \sum_{i=1}^{log_2(n)} 2^{log_2(n)-i}2^{i+2} \leq \sum_{i=1}^{log_2(n)} \frac{2^{log_2(n)}2^{i+2}}{2^i} = 2\sum_{i=1}^{log_2(n)} n = O(n.log(n))$ amount of work. In other words, mergesort is $O(n.log(n))$.

It is easily shown that mergesort is a stable algorithm. First, the tree recursion does not move any elements at all. Second, the recursion guarantees that the first recursive call defines the leftmost region while the second recursive call defines the rightmost region. Furthermore, during the merge phase, whenever possible, elements are always selected in the subregions from left to right. By using a *non-strict* test on `low1` and `low2`, an element of the leftmost region is preferred over an identical element of the rightmost region. For instance, to sort a vector containing numbers, we should use Scheme's <= operator instead of <. Hence the relative order between identical elements is preserved by the process if we call the procedure using a non-strict comparison operator.

Since the basic operation of mergesort consists of sequentially comparing elements of vectors from left to right, mergesort is particularly well-suited for sorting single linked lists. We leave this algorithm as an exercise for the reader.

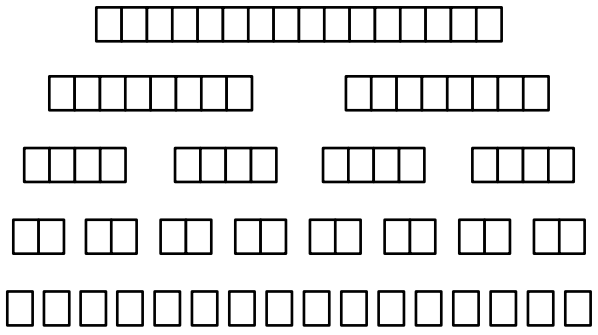

Figure 5.10: Mergesort's Recursion Tree

5.4.3 Heapsort

The relation between priority queues and sorting was already touched upon in sections 5.3.2 and 5.3.3. Given the close relation between heaps and priority queues, it is therefore no surprise that there is a relation between heaps and sorting. The resulting sorting algorithm is called *heapsort* and was first proposed by J.R.J. Williams in 1964. The idea of heapsort is quite simple. Given a vector, we first convert the vector to a heap. Then we remove the smallest element of the heap, one by one, until the heap is empty. Hence, the heap acts like a sorting device. Every time we delete its smallest element, it reorganises itself for the next smallest element to be readily present in its first position. The `heapsort` procedure is shown below.

```
(define (heapsort vector <<?)
  (define >>? (lambda (x y) (not (<<? x y))))
  (define heap (from-scheme-vector vector >>?))
  (define (extract idx)
    (vector-set! vector idx (delete! heap))
    (if (> idx 0)
      (extract (- idx 1))))
  (extract (- (vector-length vector) 1)))
```

On first thought, heapsort is not in-place since we need a second vector to store the sorted elements that are deleted from the heap. However, by a clever usage of the original vector, this can be avoided. First, remember from section 4.4 that the constructor `from-scheme-vector` takes an existing vector and that it builds a heap by reorganising the elements sitting in that vector. No new vector to store the heap is created. By using this constructor, we convert the input vector of `heapsort` into a heap. Second, we observe that the heap gets one element shorter every time its smallest element is deleted. Hence, at the end of the heap, an entry in the vector is freed up. This insight allows us to reuse those vector entries to store the elements that are deleted from the heap.

In other words, as the heap gets shorter and shorter, the list of sorted elements gets longer and longer. This is what the `extract` procedure does. It traverses the vector from (− (`vector-length vector`) `1`) down to `0`. In every step of the iteration, the smallest element of the heap is removed and stored in the vector entry that was freed up by this removal. Hence, the resulting algorithm is in-place.

Notice that an inverse order `>>?` is used to construct the heap since the rear of the sorted vector is constructed first by `extract`. If we would use the `<<?` operator to construct the heap, then the smallest elements would be removed first. As a result, the smallest elements would be stored in the rear of the vector. As a result, the vector would be sorted in descending order (instead of the ascending order suggested by `<<?`). Therefore, the negation of `<<?` is used.

Let us have a look at the performance characteristic for this sorting algorithm. First, a heap is constructed. Remember from section 4.4.7 that this requires $O(n)$ work. Subsequently, every element is deleted from the heap in the procedure `extract` which is therefore called n times. Inside the body of `extract`, the call to the `delete!` operation costs $O(log(n))$. As a result, heapsort is $O(n) + O(n.log(n)) = O(n.log(n))$. We leave it to the reader to verify that heapsort is not a stable algorithm.

5.5 Limitations of Comparative Sorting

All sorting algorithms discussed until now are so called *comparative sorting algorithms*. This means that the algorithm itself is unaware of the internal structure of the data elements to be sorted. It only depends on some abstract `<<?` operator that compares two data elements. As a consequence, the same algorithm can be used to sort Scheme numbers, persons and so on. All we have to do is call the procedure with a dedicated comparator.

It can be proven that $n.log(n)$ *is a theoretical lower bound for comparative sorting algorithms*. In other words, *no algorithm that is based on comparing data elements only can do faster than* $\Omega(n.log(n))$. The proof for this claim is as follows.

Consider a sequence s_1, s_2, s_3, ..., s_n of n data elements to be sorted. A comparative algorithm for sorting those data elements can be depicted in a tree as exemplified by figure 5.11 (where $n = 3$). Every node in the tree represents one comparison of two given elements. Depending on the result of the comparison, the algorithm proceeds and two other elements are subsequently compared. Depending on a particular input sequence, a particular path down the tree leads to the sorted result. In other words, every possible execution of the algorithm corresponds to a path from the root of the tree to one of its leaves. The tree is a picture of every possible execution trace of that sorting algorithm[3].

[3]The tree depicted in figure 5.11 is the tree that shows the possible executions of insertion sort on a vector of three elements.

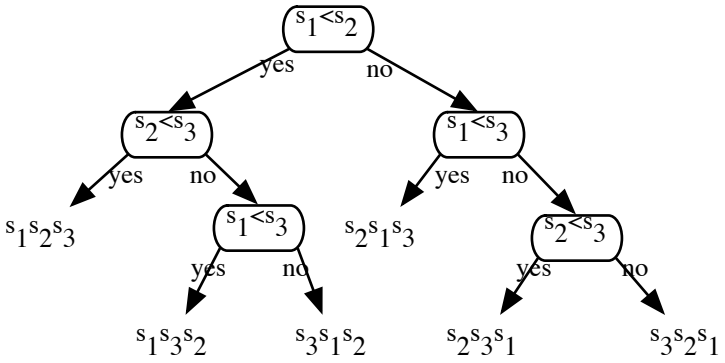

Figure 5.11: A Decision Tree for a Sort of 3 Elements

Sorting the elements means that we reorder — i.e. permute — them in a certain way. There are $n!$ such permutations possible. Since we have $n!$ possible permutations, we have $n!$ leaves in the tree. As a result, the height h of the tree is certainly greater than $log(n!)$. Hence $h \geq log(n!)$. Since $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$ (this is known as Stirling's approximation for $n!$), we have $n! > (\frac{n}{e})^n$. Hence, $h \geq log(\frac{n}{e})^n = n.log(\frac{n}{e}) = n.log(n) - n.log(e) \geq n.log(n)$. In other words $h \geq n.log(n)$. Looking back at the meaning of the tree, the height of which is h, this means that any sorting algorithm based on comparisons requires at least $n.log(n)$ comparisons and thus has a performance characteristic that is in $\Omega(n.log(n))$.

5.6 Comparing Comparative Algorithms

Remember from section 1.5.2 that the big O, big Θ and big Ω notation are more useful for comparing performance characteristics that are different than for comparing performance characteristics that are the identical. So, now that we have discussed three algorithms in $O(n^2)$ and three algorithms in $O(n.log(n))$, what are the things to keep in mind when selecting a sorting algorithm in a concrete practical situation?

1. Forget about bubble sort. The only case in which it has an acceptable performance characteristic is when the input data is already sorted. For all other cases, it is the slowest sorting algorithm that was discussed.

2. Selection sort is slightly slower than insertion sort but can be a good candidate when the number of moves becomes dominant due to the size of the records. In that case selection sort outperforms insertion sort. In all other cases, insertion sort is slightly better than selection sort.

3. Quicksort is no longer performant for vectors the size of which is smaller than ten because of its computational overhead. In that case, use insertion sort. Hence, the best versions of quicksort switch to insertion sort (instead of calling themselves recursively) once a given lower bound on the number of elements in the vector is reached. If the number of data moves is to be kept low, take selection sort. The exact vector length for which the switch is to be made depends on the concrete operating system, programming language and interpreter at hand. Nevertheless, ten is often taken as an estimate.

4. Quicksort en heapsort are both good general purpose sorting algorithms in $O(n.log(n))$. However, when taking the hidden constants in this performance characteristic into account, then quicksort usually performs two to three times better than heapsort. Since, the difference is limited to a multiplicative constant, the exact technical details of how the algorithms are programmed can make a huge difference. A big advantage of heapsort is that is entirely in-place whereas this is not the case for quicksort because of the recursion depth. When the data elements to be sorted are large, quicksort spends a lot of time moving data elements around pivot elements. Heapsort performs better in those cases.

5. Taking the leftmost element of a vector is the only crucial operation required by mergesort. This means that mergesort is particularly well-suited to sort linked lists and that it is easy to use mergesort as an external sorting algorithm: merging two sequential files is as easy as merging two vectors. Therefore, mergesort is good to sort enormous data sets since it does not require an $O(1)$ direct access into the vector. Mergesort always makes less comparisons than quicksort. As a matter of fact, it can be shown that mergesort requires about 40 % less comparisons than quicksort does in the average case. Mergesort also makes less moves than quicksort. In both cases, it can be shown that mergesort's worst case ties with quicksort's best case. But remember that it is extremely hard to make mergesort in-place.

This concludes our study of comparative sorting algorithms.

5.7 Sorting in Linear Time

Section 5.5 has presented a mathematical proof for the fact that $\Omega(n.log(n))$ is really the best we can do if we confine ourselves to sorting algorithms that compare elements using some comparison operator. The point is that these algorithms work for *any* data type. They do not depend on the structure of the keys of the data elements to be sorted. By making a number of assumptions about the structure of the keys of the data elements to be sorted, we can beat this lower bound and obtain linear behaviour. We present three such linear sorting algorithms: radix sort, bucket sort and counting sort.

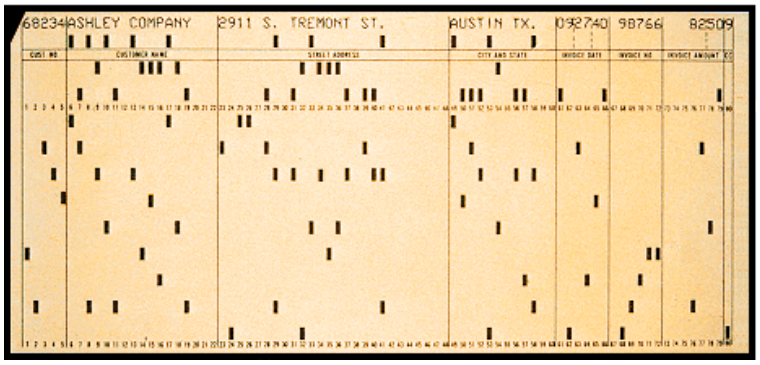

Figure 5.12: Punch Card Example

5.7.1 Radix Sort

Radix Sort is of historical significance since it is easily implemented in a mechanical way. Mechanical card sorters take a stack of cards and sort the cards according to the value of a character that is located on a certain position on the card. For instance, it is possible to construct a mechanical card sorter that has ten output bins. Cards placed in the input bin are read one by one and are put in the output bin that corresponds to the value of the character (between '0' and '9') that is punched on a particular column on the card. To get a feel of this mechanism, we refer to figure 5.12 for an example of such cards. The exact location of a punched hole causes the mechanical sorter to interprete the hole as a digit and put the card in the corresponding output bin.

Suppose that we have a stack of cards that are identified by an identification number consisting of k digits that may vary between '0' and '9'. We can sort it by first putting the cards in the mechanical sorter and sort them according to the value of their least significant digit (i.e. the rightmost digit). This puts the cards in one of the ten output bins. The ten output bin are then joined again thereby respecting the internal order of each bin. Subsequently, the resulting stack is put back into the mechanical sorter, but this time we sort the stack according to the second rightmost digit. Repeating this process for all k digits results in the stack of cards being sorted. The process is illustrated (for two digits) in figure 5.13.

It is crucial to respect the order of the digits and to start the sort with the least significant digit. Suppose we have 2-digit numbers and suppose that we first sort the cards based on the most significant digit. This would put all cards with numbers of the form "1X" in the first output bin, all cards with numbers of the form "2X" in the second output bin and so forth. Now we put all cards

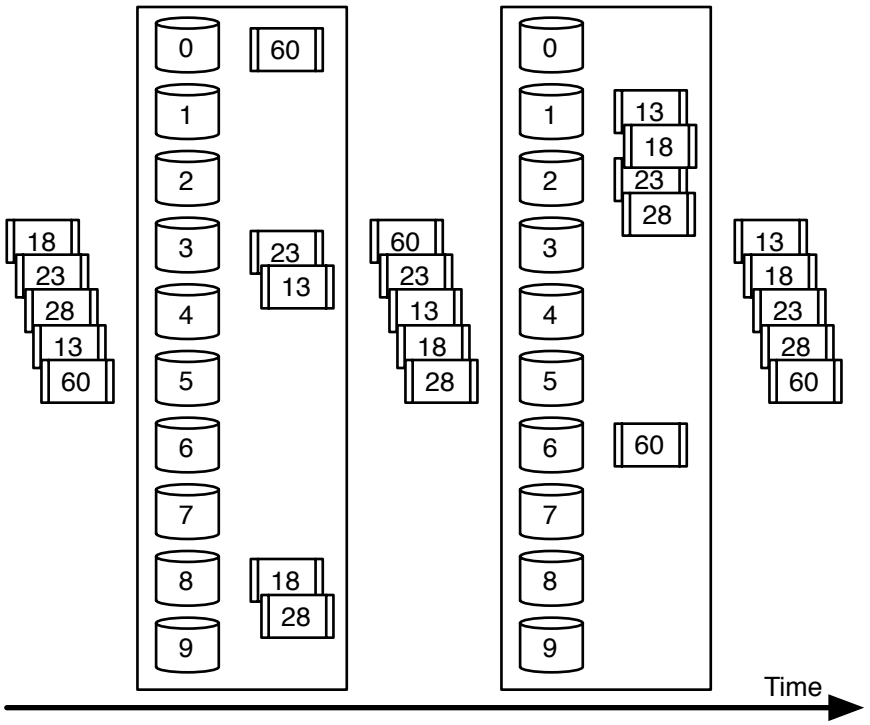

Figure 5.13: Radix Sort

back into one stack and sort them according to the second digit. Unfortunately, this distributes cards of the form "1X" into different bins depending on their second digit. Hence, the ordering that was based on the first digit is destroyed again.

Radix sort is not limited to numbers. It is applicable to any kind of keys that consist of k symbols $s_k s_{k-1} ... s_1$ and in which each symbol has N possible values. N is called the *radix* of the keys. For example, if the keys consist of six digit numbers, then $k = 6$ and $N = 10$ since every digit can take 10 possible values. Abstractly spoken, the only thing needed for radix sort to work is a total ordering $\preceq_D$ between the symbols. Using such an order defined for the individual symbols, radix sort *induces* an order on the keys $\preceq_K$ that is known as the *lexicographical ordering* of keys. The lexicographical ordering is recursively defined as follows: given two keys of identical length i, $K = d_i d_{i-1} ... d_1$ and $K' = d'_i d'_{i-1} ... d'_1$, then $K \preceq_K K'$ if and only if $d_i \preceq_D d'_i$ *or* $d_i = d'_i$ and $d_{i-1} ... d_1 \preceq_K d'_{i-1} ... d'_1$. For keys K and $K' = K''.K'''$ where the length of K is the same as the length of K'', we have $K \preceq_K K'$ if $K \preceq_K K''$.

The following code excerpt is an implementation of radix sort. Radix sort

takes a Scheme list `slst` of records and produces a Scheme list that contains those records in sorted order. The procedure `key` extracts the key from a record sitting in the Scheme list. The algorithm works for keys that consist of `key-size` digits between 0 and 9. We leave the generalisation of the algorithm towards other types of digit as an exercise to the reader.

```
(define (radix-sort slst key key-size)
  (define sort-bins (make-vector 10 ()))
  (define (spread lst digit-k)
    (define (digit item)
      (remainder (quotient item (expt 10 digit-k)) 10))
    (define (spread-iter lst)
      (let ((idx (digit (key (car lst)))))
        (vector-set! sort-bins
                     idx
                     (cons (car lst)
                           (vector-ref sort-bins idx)))
        (if (not (null? (cdr lst)))
          (spread-iter (cdr lst)))))
    (spread-iter lst))
  (define (collect)
    (define (collect-iter index acc)
      (define (collect-list lst acc)
        (if (null? lst)
          (if (> index 0)
            (collect-iter (- index 1) acc)
            acc)
          (collect-list (cdr lst) (cons (car lst) acc))))
      (let ((l-index (vector-ref sort-bins index)))
        (vector-set! sort-bins index ())
        (collect-list l-index acc)))
    (collect-iter 9 ()))
  (define (radix-iter digit-k slst)
    (spread slst digit-k)
    (if (= digit-k key-size)
      (collect)
      (radix-iter (+ 1 digit-k) (collect))))
  (radix-iter 0 slst))
```

The heart of the radix sort algorithm is a procedure `radix-iter` that processes all the digits from right to left. The bins are represented as a vector `sort-bin` of Scheme lists. In every iteration of `radix-iter`, the input list is spread out in the bins by taking the k^{th} digit into consideration. This is the job of `spread`. `collect` iterates over the bins and relinks the elements back into one single list. Notice from the implementation that `spread-iter` puts the elements in the bins in reverse order (by 'consing' the element upfront the rest of the bin which causes elements that were encountered last to reside at the

front of the bin). This is resolved by `collect` which uses an accumulator to reconstruct the list in reverse order compared to the order of the bins.

Properties

Radix Sort beats the theoretical lower bound $\Omega(n.log(n))$ because it makes a number of structural assumptions about the keys of the elements it is sorting. It relies on the fact that keys are $k - digit$ strings that can only take a limited number of different values. In other words, it does more than boldly comparing keys.

What can we say about the performance? Suppose that we have n data elements to sort and that the key encompasses k digits. The `radix-iter` procedure is called k times (for every digit). In each iteration, `spread` and `collect` generate an amount of work that is in $O(n)$ because all elements of the input list are distributed in the bins and subsequently the bins are recombined in order to form a list a gain. Hence, radix sort is in $O(k \times n)$. In presentations of radix sort, the constant factor k is usually not omitted. In most practical cases (e.g. when sorting personal records based on names) k can be quite large.

For large n, radix sort is the fastest sorting algorithm among all sorting algorithms that we have discussed so far. However, the value for n for which radix sort starts getting faster than the advanced $O(n.log(n))$ sorting algorithms may be very large. The exact value depends on the particular hardware, programming language, operating system and so on.

Radix sort can only be used in some specific cases. A first restriction is the form of the key. As explained, structural keys are needed that can be clearly divided into "digits". Furthermore, the number of values that digits can take must be known in order to know the size of the `sort-bins` upfront. Lastly, the digits must be easy to map onto integer numbers (starting from 0) since these are needed to index the vector of bins. As an example, it is not easy to apply radix sort to sort real numbers. The reason is that the number of digits in floating point numbers usually varies. Furthermore, the location of the dot is crucial in the sorting process. Another restrictive property of radix sort is that the number of values that one digit can take must be "small enough". If every digit were to take 10000 possible values, then the vector of bins would contain 10000 linked lists. In such situations, the computational overhead that is the result of the interplay between `collect-iter` and `collect-list` gets quite large.

Radix sort is not in-place. Additional memory is required to store the bins and to construct the linked lists in each bin. This is the reason why radix sort operates best for linked lists. If the data elements are already in a linked list, then we can use the "cdr pointers" of that linked list to construct the lists inside the bin. In our presentation of radix sort, this optimization was not taken into account: instead we use `cons` inside `spread` to create a new node in the bin of lists. We leave it as an exercise to the reader to modify the algorithm such that the "cdr pointers" of the input list are *reused* for this matter.

Radix sort is stable. As a matter of fact, stability of *all* the individual phases

is crucial to the well-functioning of the algorithm. Consider the lexicographical ordering again. This ordering is based on a correct ordering of the most significant digit. For two digits that are equal, it is the second digit that determines the order and so on. In order to obtain this ordering, it is necessary that the radix sort phase which puts the elements in order according to the most significant digit respects the order that was obtained in the radix sort phase that puts the elements in order according to the second digit. Hence stability at all levels has to be guaranteed for the algorithm to function.

5.7.2 Bucket Sort

The bins of the radix sorting algorithm are often referred to as *buckets*. The vector `sort-bins` containing the linked lists is then referred to as a *bucket vector*. *Bucket sort* is a simple sorting algoritm that constitutes one single phase of the k phases of the radix sorting algorithm. Obviously, bucket sort is $O(n)$. Bucket sort is useful when:

- an imperfect sort is needed. E.g., given a set of integer numbers of two digits. Then bucket sort can be used whenever we are interested in the groups between 0 and 9, between 10 and 19 and so forth. Bucket sort can create these groups in a single pass because the groups are nothing but the contents of the buckets. The ordering inside such a bucket is undefined. Hence the term imperfect sorting.

- buckets are small enough to sort using another sorting algorithm. Bucket sort can be used as a first phase in a 2-pass sorting algorithm. E.g., one might use bucket sort to form the groups of numbers as indicated above. Inside every group, insertion sort could be used to sort the individual buckets in a relatively fast way.

5.7.3 Counting Sort

Counting sort is another linear algorithm that relies on the structure of the keys of the data elements to be sorted. It was invented in 1954 by Herman Hollerith Seward. Counting sort assumes that all keys are integers in the range `0` to `max-key`. The idea is, given a data element the key of which is k, to *count* the number of data elements with a key that is smaller than k. This count can be used to determine the location of the data element in the output vector. If `max-key` is close to n, then counting sort is $O(n+$`max-key`$)$. Whenever `max-key` becomes to big (w.r.t. to n), then counting sort is no longer efficient. Counting sort is harder to adapt to other types of keys than radix sort because we really use the fact that its keys are *numbers* that vary between `0` and `max-key`. E.g. in order to sort a list of names, this would require us to map every name onto a number in such a way that we obtain a (nearly) perfect mapping that assigns a number to each name. As will be shown in chapter 7, this is practically impossible.

The code for counting sort is presented below. Counting sort is not an in-place algorithm. First, it needs an internal vector the size of which is `max-key`. Second, it needs an additional output vector that is of the same size as the input vector. The reason is that counting sort constructs the output vector by reordering the input vector in a way that does not just consist of swapping elements. As a result the algorithm might need to store data elements in locations in the input vector that contain other data elements that still needs to be taken into consideration in a later phase of the algorithm. Doing so would destroy the input vector with a loss of information. Therefore the algorithm is parametrised by two vectors: `in` is a vector containing the elements to be sorted. `out` is an empty vector that will be used to store the output of the algorithm.

```
(define (counting-sort in out max-key key)
  (let ((count (make-vector max-key 0))
        (size (vector-length in)))
    (define (fill-count-vector i)
      (let ((k (key (vector-ref in i))))
        (vector-set! count
                     k (+ (vector-ref count k) 1))
        (if (< (+ i 1) size)
          (fill-count-vector (+ i 1)))))
    (define (sum-vector i)
      (vector-set! count
                   i
                   (+ (vector-ref count (- i 1)) (vector-ref count i)))
      (if (< (+ i 1) max-key)
        (sum-vector (+ i 1))
        count))
    (define (spread-out-again i)
      (let* ((data (vector-ref in i))
             (k (- (vector-ref count (key data)) 1)))
        (vector-set! out k data)
        (vector-set! count (key data) k)
        (if (<= i 0)
          out
          (spread-out-again (- i 1)))))
    (fill-count-vector 0)
    (sum-vector 1)
    (spread-out-again (- size 1))))
```

The `counting-sort` procedure works in three consecutive phases:

1. In the first phase, all keys residing in the input vector are considered, one by one, by the procedure `fill-count-vector`. For every occurrence of key k, the counter in (`vector-ref count k`) is incremented by 1. As a result, every entry in the `count` vector contains the number of occurrences of that particular key.

2. In the second phase, the entries in the `count` vector are added by `sum-vector` such that for a given index `i`, (`vector-ref count i`) contains the number of elements smaller than or equal to `i`.

3. In the final phase, `spread-out-again` puts the elements in the output vector. In every phase of the iteration, `k` is read from the `count` vector in order to know the index in the target vector. If all elements in the input vector were distinct, then the value of index (`vector-ref in k`) in the `count` vector is the correct target index of the element originally residing in (`vector-ref in k`). But since not all elements are necessarily distinct, we decrement the value sitting at index (`vector-ref in k`) in vector `count`.

Let us establish the performance characteristic for counting sort. Filling the `count` vector in the `fill-count-vector` procedure requires $O(n)$ work. In the second phase, summing up all elements of the `count` vector requires $O(k)$ work where k is the number of keys (i.e. `max-key` in our algorithm). The final phase also requires $O(n)$ work since all elements have to be put back into the output vector, one by one. As a consequence, counting sort is an $O(n + k)$ algorithm. In practice, usually $k = O(n)$ such that counting sort operates in $O(n)$ time. Notice that counting sort does not use any comparisons at all!

Counting sort is a stable sorting algorithm. Numbers that occur first in the input vector also occur first in the output vector, since the output vector is constructed from left to right based on the input vector. The relative position of two elements is only exchanged if the ordering relation really requires this.

Chapter 6

Trees

In chapter 2 we have concluded that strings are the poorest data structure imaginable. Their "flat" organisation makes it very hard to lookup data. String searching algorithms are quite complicated and slow. It was concluded that more clever ways of organising data in computer memory are needed to make our algorithms faster and simpler. Chapter 3 has presented a number of data structures that linearly organise data in computer memory. Some linear data structures have been shown to posses some pleasant properties. E.g., sorted lists have the property of allowing for the very fast binary searching algorithm if a vector implementation is chosen. In this chapter, we pursue our quest for clever ways of structuring data in computer memory by studying *hierarchically structured* data. Hierarchical structures are also referred to as *trees*.

In our every day life, we encounter many examples of trees (apart from the ones we spot in a forest, that is). A first example is the managerial structure of an organisation. It is shown in figure 6.1. As we can see from the drawing, the hierarchy consists of nodes, one of which is the top (also called the root) of the hierarchy. Nodes can have children which are nodes in their turn. Nodes that have children are called internal nodes. Leaf nodes is the term we use to indicate nodes that have no children. A second example of a hierarchy is the kind of taxonomical hierarchies we encounter in the natural sciences. For instance, figure 6.2 shows a hierarchical representation of scientific topics. The tree starts with "earth" as root. Only part of the tree is developed in the drawing, to wit the part that deals with living objects. In the terminology of trees, we call the tree that deals with living objects a subtree of the entire tree.

Trees also occur frequently in computer applications. Here are just two obvious examples:

Folders Computer systems organise their files into folders (a.k.a. directories). These folders are organised as a tree. Folders can contain files or folders in their turn. Folders never reside in two or more parent folders at the same time. As a result, the folder structure is hierarchical. The internal nodes of the tree are the folders. Files form the leaf nodes of the tree.

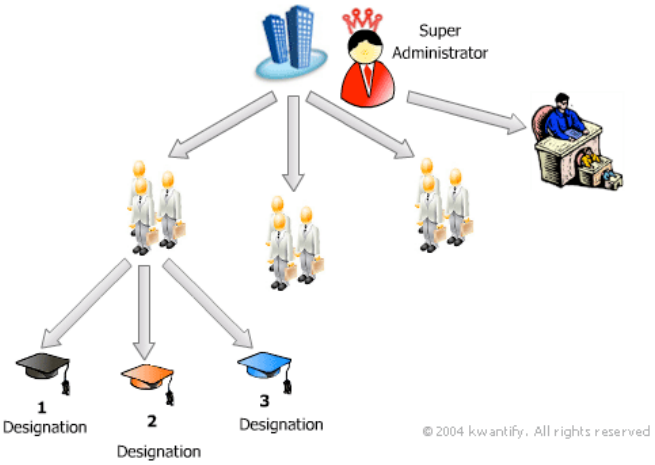

Figure 6.1: A Managerial Taxonomy

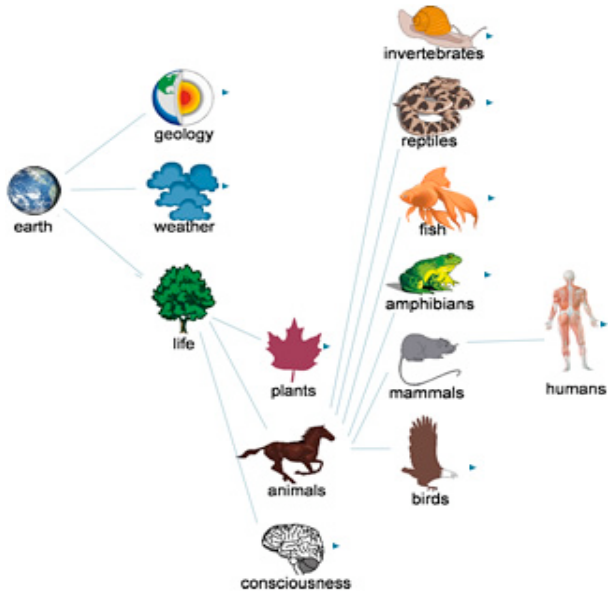

Figure 6.2: A Scientific Taxonomy

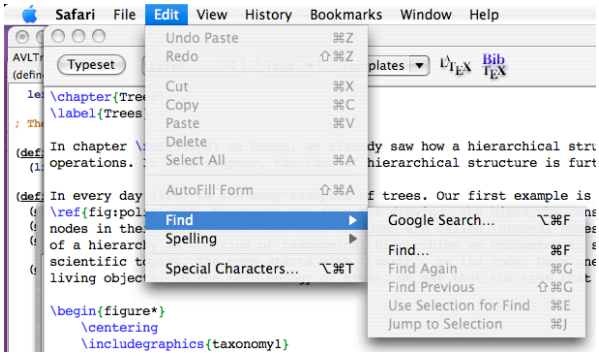

Figure 6.3: A Hierachical Menu Structure

Menu Structures Figure 6.3 shows a screenshot of one of the menus of Apple's internet browser. The menu structure is a hierarchical data structure as well. The root consist of the menu bar that is constantly shown on the screen. When selecting one of the options in the menu bar, a pull-down menu appears that can be further pulled down in order to reveal a submenu, and so on. In Microsoft's Windows, the "start" button shows a menu that can have an arbitrary number of submenus. This is another example of a tree.

6.1 The Structure of Trees

This section introduces some basic terminology about trees and discusses the main alternatives we have at our disposal when representing trees in Scheme. We conclude the section with a concrete example that illustrates one of the most frequently used applications of trees in computer science, namely the representation of arithmetic expressions.

6.1.1 Terminology

A tree is a collection of data elements residing in *nodes*, the organisation of which satisfies the following conditions:

- The *empty tree* is a tree that does not contain any nodes at all.
- Any non-empty tree has a unique *root node*.
- Every node has a direct reference to a number of nodes that are called the *children* of that node. The node itself is called *the parent node* of the children. The parent node is the only node referring to a node.

For a node n in a tree, we define its *arity* as the number of children it has. 1-ary nodes have one child, binary nodes have two children, ternary nodes have three children, and so forth. In general, we speak of $k-ary$ nodes. A tree for which all nodes have maximum k children is called a $k-ary$ tree. For instance, a binary tree is a tree the nodes of which have two or less children. Similarly, ternary trees consist of nodes with no more than three children. Nodes that have no children are called *leaf* nodes.

Remember from section 4.4 that we defined the height of a tree as the length of the longest possible path from the root node of the tree to one of its leaf nodes. Also remember from section 4.4 that the height of a *complete binary tree* with n nodes is $\lfloor log_2(n) \rfloor$. This can be generalized: the height of a *complete* $k-ary$ tree is $\lfloor log_k(n) \rfloor$.

Given a node n in a tree T. For every child n_i of n, there exists a tree T_i that has n_i as its root node. Even if n_i has no children, T_i can be considered as a tree consisting of only one node. T_i tree is called a *subtree* of T.

Let n be a node in a tree T. Another node is said to be a *descendant* of n if it is a direct child of n, or if it is a descendant of one of n's children. In other words, a node is a descendant of n if it occurs in a subtree of n. All nodes of which n is a descendant are called the *ancestors* of n. Finally, two nodes n_i and n_j that have the same parent are called *siblings*.

6.1.2 Binary Trees

We devote a special section to binary trees — i.e. trees in which every node has at most 2 subtrees — because of the tremendous importance of binary trees in computer science. Nevertheless, many concepts, properties and algorithms dealing with binary trees are trivially extended to k-ary trees for arbitrary k.

The Binary Tree ADT

We first present the `binary-tree` ADT below. This ADT is a straightforward application of the definitions of a tree presented above. In what follows, we discuss a number of representations for implementing this ADT. An example of how the ADT can be used in practice is given in section 6.1.3.

```
1  ADT binary−tree
2
3  null−tree
4      binary−tree
5  new
6      ( any binary−tree binary−tree → binary−tree )
7  null−tree?
8      ( binary−tree → boolean )
9  left!
10     ( binary−tree binary−tree → binary−tree )
11 left
```

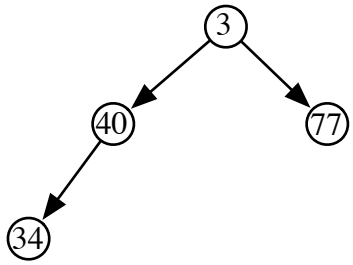

Figure 6.4: A Sample Binary Tree

```
12    ( binary–tree → binary–tree )
13  right!
14    ( binary–tree binary–tree → binary–tree )
15  right
16    ( binary–tree → binary–tree )
17  value!
18    ( binary–tree any → binary–tree )
19  value
20    ( binary–tree → any )
```

`null-tree` is the empty binary tree and given a binary tree `t`, then (`null-tree? t`) can be used to check whether or not `t` is the `null-tree` tree. Given a data element `d` and given two binary trees `t1` and `t2`, then (`new d t1 t2`) creates a new binary tree that contains that data element and that has the two argument trees as its children. The two children are called the *left child* and the *right child* of the newly created node.

To exemplify the operations of the ADT we use the following Scheme expression in order to create the tree depicted in figure 6.4.

```
(define a-tree
  (new 3
       (new 40
            (new 34
                 null-tree
                 null-tree)
            null-tree)
       (new 77
            null-tree
            null-tree)))
```

A Linked Implementation

The most natural implementation for binary trees is presented. Every node of the binary tree is represented by two Scheme pairs that store the value of that node as well as a reference to its left and right subtree.

```
(library
 (binary-tree)
 (export null-tree new null-tree? value value! left left! right right!)
 (import (rnrs base (6))
         (rnrs mutable-pairs))

 (define (new value left right)
   (cons value (cons left right)))

 (define null-tree '())

 (define (null-tree? node)
   (eq? node null-tree))

 (define (value node)
   (car node))

 (define (value! node value)
   (set-car! node value))

 (define (left node)
   (cadr node))

 (define (left! node left)
   (set-car! (cdr node) left))

 (define (right node)
   (cddr node))

 (define (right! node right)
   (set-cdr! (cdr node) right)))
```

The general structure of a tree node is shown in figure 6.5. The node is depicted as a grey box that stores a value and that has two external references pointing to its left and right subtrees. This representation of trees is referred to as the *linked representation* because it models a tree as a collection of nodes that refer to each other by means of pointers in Scheme (i.e. dotted pairs). We clearly see that a node is internally represented as a system of two pairs. We can change the internal representation of nodes without leaving this linked representation scheme. E.g., we might represent a node as a vector with three entries or we might change the organisation of the pairs constituting a node. These are all variations on the way nodes are stored. The trees that are constructed this way still remain linked trees.

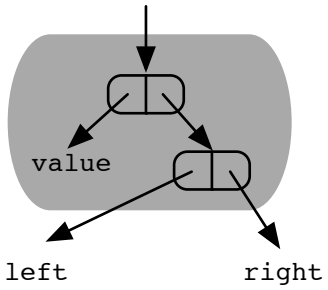

Figure 6.5: The Internal Structure of a Linked Tree Node

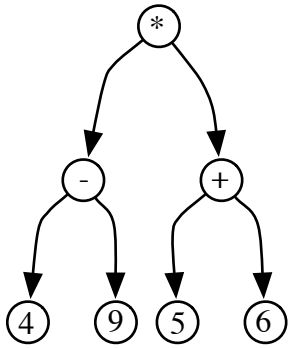

Figure 6.6: A Tree Representation of "(4-9)*(5+6)"

6.1.3 An Example: Arithmetic Expressions

One of the most frequently occurring applications of trees in computer science is found in the representation of mathematical expressions. The example given below is a tree that represents the expression whose string representation is "(4-9)∗(5+6)". The tree is depicted in figure 6.6.

```
(define t4 (new 4 null-tree null-tree))
(define t9 (new 9 null-tree null-tree))
(define t5 (new 5 null-tree null-tree))
(define t6 (new 6 null-tree null-tree))
(define minus (new '− t4 t9))
(define plus (new '+ t5 t6))
(define times (new '∗ minus plus))
```

The example shows how to represent an *arithmetic expression* as an "expression tree". From an algorithmic point of view, these expression trees are far superior to strings. The following procedure illustrates this. It is capable of

calculating the value of *any* such tree that contains arbitrary deep combinations of +, - and *. Writing such a procedure that immediately operates on the flat string representation of the same expression would be *far* more complex.

The technique of representing expressions by a tree is e.g. heavily used deep down in spreadsheet programs such as Microsoft's Excel or Apple's Numbers. Every time a user types a formula in one of the cells of the spreadsheet, the spreadsheet transforms the user's string representation of the formula to a tree representation that is stored in the cell. (Re)calculating the values of the cells of the spreadsheet is then just a matter of applying a recursive procedure like `eval` to these trees that are stored in the cells.

```
(define (eval tree)
  (cond ((eq? (value tree) '+)
         (+ (eval (left tree))
            (eval (right tree))))
        ((eq? (value tree) '*)
         (* (eval (left tree))
            (eval (right tree))))
        ((eq? (value tree) '-)
         (- (eval (left tree))
            (eval (right tree))))
        (else (value tree))))
```

This idea can be generalized up to the level where *any* Scheme expression is represented as a (not necessarily binary) tree. Transforming any Scheme expression from a string format (typed by a programmer) to an equivalent tree is exactly what Scheme's `read` phase does. The tree resulting from this phase is subsequently used by `eval` to evaluate the expression. For more information on this process, we refer to chapter 4 of the Abelson&Sussman course.

6.1.4 Alternative Representations

The linked representation presented above is only one of the many possible representations for the `binary-tree` ADT. Analogous to the list ADTs implemented in chapter 3, tree representations come in different variants. In what follows we discuss a number of variations on the single linked representation just presented.

Vector Representation

In a vector representation, the nodes of the tree have to be mapped onto indices in a vector. From section 4.4 we know that this is simple for *complete* binary trees, i.e. trees in which all levels are either full or partially filled from left to right without missing intermediate nodes. However, when trying to map a tree "with gaps" onto a vector we have to make sure that those gaps are represented in the vector as well. If not, it would be impossible to reconstruct the tree given the vector. Figure 6.7 shows an incomplete binary tree and shows how

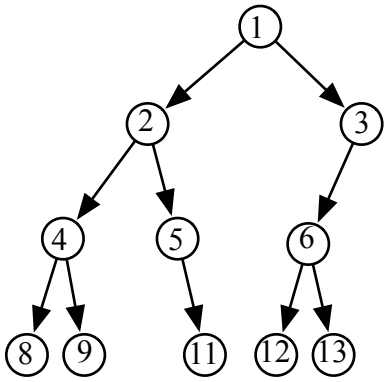

Figure 6.7: Node Numbering Scheme for a Vectorial Representation

every node is mapped onto an index in a vector. Each gap in the tree leads to a corresponding unused position in the vector. E.g., we might consider storing the symbol 'empty in the seventh and tenth entry of the vector representation. This way, we know that we have to "skip" those nodes when thinking of the vector as a hierarchical structure again.

The problem with the gaps can be solved by crunching all the tree elements into the leftmost positions of the vector. However, this is far from trivial an exercise. We leave it as an exercise to the reader to think about the consequences that this decision has for the internal representation as well as for the accessors defined on nodes (such as getting the left or right child of a node). This is not a trivial programming exercise.

A major downside of the vector representation is inherited from the usage of vectors in general: the maximum capacity of the tree has to be known upfront. If not, adding a node to the tree can require one to extend the vector by creating a new (larger) vector and by copying all the nodes from the old vector into the new one. Clearly, this is an $O(n)$ operation. An alternative might be to make sure that the vector is large enough to store all "reasonable" trees. This solution has the drawback that it can be a waste of memory.

Alternative Linked Representations

Analogous to linked list nodes, we can provide tree nodes with a "back pointer" which results in a double linked representation for trees. The back pointer of a node refers to the parent of the node. This representation is shown infigure 6.8.

Just as was the case with double linked lists, double linking can simplify and speed up certain operations on binary trees considerably. An example of such an operation is to find the parent node of a given node. In our double linked representation shown in figure 6.8, this operation is in $O(1)$ since we merely have to follow the back pointer. In the single linked representation, finding the

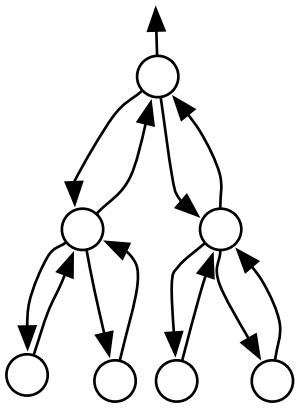

Figure 6.8: A Double Linked Representation of Binary Trees

parent of a node requires $O(n)$ work (where n is the number of elements of the tree) since we have to start at the root of the tree and work our way down the tree to the node for which the parent is required. Since we have no clue on where to find the parent, we potentially have to search the entire tree. In section 6.3.2 we discuss a clever way to organise the elements of a tree such that this effort can be reduced to $O(log(n))$. Still, storing a back pointer yields more efficient code. The price to pay is that we have to store an extra pointer per node.

Sometimes, additional navigational operations on binary trees are needed. In order to implement them efficiently, as many pointers as necessary can be stored. These pointers refer to the "next node" for that particular operation. An example of this technique are *threaded trees*, an example of which is presented in figure 6.9. Threaded trees are binary trees in which each node that has no right child uses that pointer to refer to the "next node" for some particular definition of "next'. The pointer is called *a thread*. The tree shown in figure 6.9 displays a threaded tree that is optimised for finding next nodes when traversing the tree using the in-order traversal method that is explained in section 6.2.1. Alternative traversal threads are possible as well. Tree traversing is the topic of the following section.

6.2 Tree Traversals

In chapter 3, we have defined a number of linear traversal procedures such as `map` and `for-each`. The procedures traverse the lists from left to right since this is by far the most "natural" way to traverse a list. For traversing trees, there is no such thing as "a" natural traversing order. Instead, two radically different traversal schemes exist:

- In *depth-first traversals*, the tree is traversed in a way that gives children

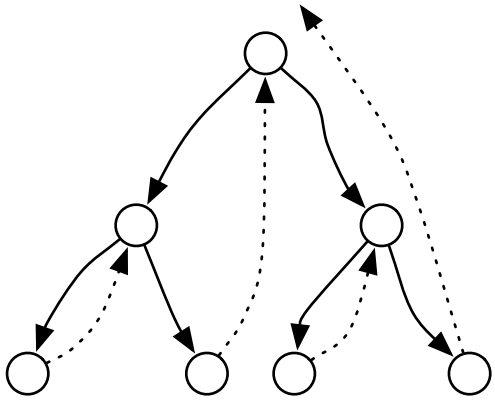

Figure 6.9: A Threaded Representation of Binary Trees

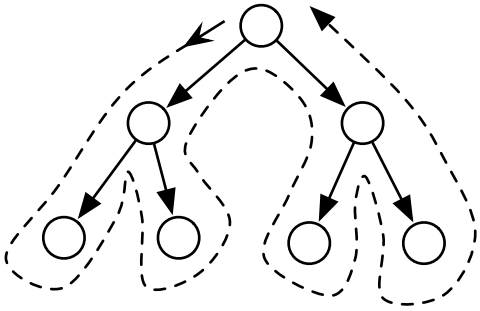

Figure 6.10: Depth First Traversal

of a node priority over the siblings of the node. As a result, the traversal immediately descends from the root of the tree down to its leaves. When all subtrees of a node have been traversed, then the siblings of that node are considered. The general idea of depth-first traversing is shown in figure 6.10. In section 6.2.1, we present three concrete implementations for this kind of traversal.

- In *breadth-first traversals*, the tree is traversed in a way that gives the siblings of a node priority over the children of the node. The children of a node are considered after having processed all the siblings of the node. The general idea of breadth-first traversing is shown in figure 6.11. Section 6.2.2 presents a concrete implementation of breadth first traversal.

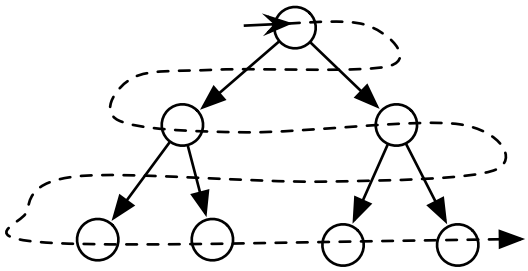

Figure 6.11: Breadth First Traversal

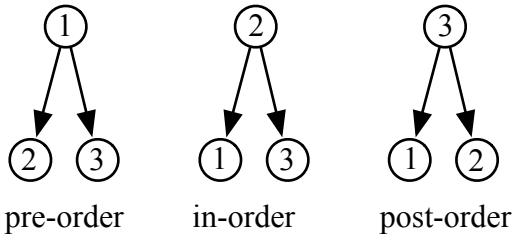

Figure 6.12: Depth First Traversal Types

6.2.1 Depth-First Traversal

As explained, processing a node in a depth-first traversal considers the children of that node before it considers the siblings of that node. In what follows, we assume that we are working with binary trees. The concepts can be generalised but this is beyond the scope of this chapter.

Given a node and its two children, there are three possible orderings in which these three components can be processed: the children are processed before the node, the children are processed after having processed the node and the node is processed inbetween the processing of both children. These orders are known as *pre-order* traversal, *post-order* traversal and *in-order* traversal respectively. They are graphically depicted in figure 6.12.

Recursive Traversal Implementations

The recursive implementation of these three traversal methods is presented below. All three procedures take a binary `tree` and a `function`. They recursively traverse the tree and apply the function to the data value stored in the nodes of the tree. Notice that this procedure is the analogue of `for-each` in linear data structures since the function is applied to every node of the tree without constructing a new result tree.

The three procedures look very similar. They consist of a local `do-traverse` procedure which starts at the root of the tree and which traverses the tree in the order prescribed. In all three cases, null trees cause the recursion to end. The only difference between the three procedures is the order of the three expressions in the body of `do-traverse`.

```
(define (in-order tree function)
  (define (do-traverse current)
    (if (not (null-tree? current))
      (begin
        (do-traverse (left current))
        (function (value current))
        (do-traverse (right current)))))
  (do-traverse tree))

(define (pre-order tree function)
  (define (do-traverse current)
    (if (not (null-tree? current))
      (begin
        (function (value current))
        (do-traverse (left current))
        (do-traverse (right current)))))
  (do-traverse tree))

(define (post-order tree function)
  (define (do-traverse current)
    (if (not (null-tree? current))
      (begin
        (do-traverse (left current))
        (do-traverse (right current))
        (function (value current)))))
  (do-traverse tree))
```

An instructive application of these three traversal procedures is related to the arithmetic expression representation technique that was explained in section 6.1.3. Given the `times` tree that was constructed in section 6.1.3, we can use it to exemplify the traversal procedures by traversing the tree in three different ways using Scheme's `display` procedure:

Pre-order traversal The call (`pre-order times display`) processes the tree in a way that displays the string `*-94+56`. With a minimum of imagination, we can insert parentheses in this string yielding (∗ (− 9 4) (+ 5 6)). This is precisely the *prefix notation* used by Scheme. Hence, we conclude that traversing a tree representation of an arithmetic expression in "pre-order" yields the prefix version of the expression.

Post-order traversal The call (`post-order times display`) results in the string `49-56+*` being printed. Again, parentheses can be inserted to make

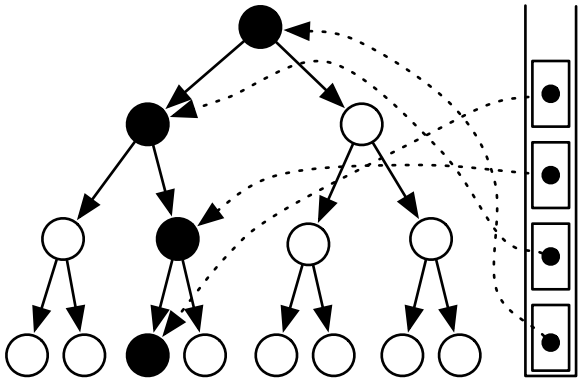

Figure 6.13: Depth First Traversals versus Stacks

this expression more readable for humans. The result is known as the *postfix notation* (or also called Polish notation) of the expression. This notation is used by some well-known programming languages (such as PostScript and Forth) and in Hewlett-Packard's scientific calculators.

In-order traversal The call (`in-order times display`) yields the string `4-9*5+6`. This is —again, with the necessary insertion of parentheses — the classic *infix notation* that we are all used to. Notice that in all three cases, the insertion of parentheses is a non-trivial programming exercise. However, it does not change the essence of the traversals. The trick is to come up with a more complicated procedure than `display` to traverse the tree.

Iterative Traversal Implementations

The recursive implementations of the tree traversal procedures is straightforward. It is extremely instructive to study the iterative versions of these procedures as well.

A key insight that is needed to understand the iterative procedures is that the steep depth-first descend in the tree (with subsequent backtracking) requires a stack to store the nodes found along the path during the descend. Figure 6.13 shows a tree in which the nodes encountered on a depth-first traversal path are shown in black. As we can see from the drawing, descending in the tree causes the traversal to create a *runtime stack*: every time we descend down the tree, the current node is pushed on the stack. As a result, when having reached a leaf, the stack is a (reverse) representation of the path of the traversal. Backtracking in the tree is accomplished by popping the topmost element from the stack. The three tree traversal methods differ in the exact way in which this principle is implemented.

The easiest application of the principle is found in the pre-order traversal. Its implementation is given below.

```
(define (iterative-pre-order tree function)
  (define stack (stack:new))
  (define (loop)
    (if (not (stack:empty? stack))
      (let ((node (stack:pop! stack)))
        (function (value node))
        (if (not (null-tree? (right node)))
          (stack:push! stack (right node)))
        (if (not (null-tree? (left node)))
          (stack:push! stack (left node)))
        (loop))))
  (stack:push! stack tree)
  (loop))
```

`iterative-pre-order` starts by pushing the root of the tree on the stack. As long as the stack contains elements, its topmost element is popped and processed by `loop`. `function` is applied to the value of the node sitting on the top of the stack. Subsequently, non-empty subtrees are pushed on the stack. Since the left subtree is pushed on top of the right subtree, the next iteration of `loop` will encounter the root node of the left subtree first. Hence, at every level, the left subtree is processed before the right subtree. Furthermore, the node itself was processed before the left subtree was pushed on the stack. Hence, we get the behaviour that was originally prescribed by the recursive version of pre-order traversal.

The in-order traversal is slightly more complicated:

```
(define (iterative-in-order tree function)
  (define stack (stack:new))
  (define (loop-up)
    (let ((node (stack:pop! stack)))
      (function (value node))
      (if (not (null-tree? (right node)))
        (begin (stack:push! stack (right node))
               (loop-down))
        (if (not (stack:empty? stack))
          (loop-up)))))
  (define (loop-down)
    (let ((node (stack:top stack)))
      (if (not (null-tree? (left node)))
        (begin (stack:push! stack (left node))
               (loop-down))
        (loop-up))))
  (stack:push! stack tree)
  (loop-down))
```

The `iterative-in-order` traversal procedure starts by pushing the root node on the stack. Then `loop-down` traverses subtrees in a leftmost way. As long as a left child exists, it is pushed on the stack and the leftmost descend down the tree continues. Having reached a leftmost node that lacks a left child, the backtracking procedure `loop-up` takes over. In every iteration, the function is applied to the node that is currently on the top of the stack. If this node has a right subtree, then it is pushed on the stack and a leftmost descend for this subtree is launched as well. If there is no right subtree, the backtracking process is continued by calling `loop-up` again. This continues until `loop-up` finds a node that does have a right subtree.

Finally, the iterative version of the post-order traversal is presented. It is the most complex one. The complexity comes from the fact that, every time we arrive at a node, we need to know whether it is the first time that we arrive at the node (which requires us to descend along the leftmost path) or whether it is the second time (which requires us to descent down the right child of the node) or whether it is the third time (which requires us to process the node itself and climb out of that subtree).

```
(define (iterative-post-order tree function)
  (define stack (stack:new))
  (define (loop-up-right)
    (if (not (stack:empty? stack))
      (let ((node (stack:pop! stack)))
        (function (value node))
        (if (and (not (stack:empty? stack))
                 (eq? (right (stack:top stack)) node))
          (loop-up-right)
          (loop-up-left)))))
  (define (loop-up-left)
    (if (not (stack:empty? stack))
      (let ((node (stack:pop! stack)))
        (cond ((not (null-tree? (right node)))
               (stack:push! stack node)
               (stack:push! stack (right node))
               (loop-down))
              ((and (not (stack:empty? stack))
                    (eq? (right (stack:top stack)) node))
               (function (value node))
               (loop-up-right))
              (else
               (function (value node))
               (loop-up-left))))))
  (define (loop-down)
    (if (not (stack:empty? stack))
      (let ((node (stack:top stack)))
        (if (null-tree? (left node))
```

```
        (loop-up-left)
        (begin
          (stack:push! stack (left node))
          (loop-down))))))
  (stack:push! stack tree)
  (loop-down))
```

The algorithm starts by pushing the root of the tree on the stack. `loop-down` descends the tree in a leftmost way. Once the left subtree is empty, we start climbing out of the left subtree. This is the task of `loop-up-left`. While climbing out of the left subtree, we check whether the current node has a right subtree. If this is the case, we select the right subtree and start a leftmost descend in that subtree. Otherwise we keep on climbing out of the left subtree until we detect that the current node is the right child of another node. This means that we have finished a climb from a left subtree and that we should "turn left" in order to climb out of a right subtree. `loop-up-right` does this climb out of the right subtree until the current node is no longer a right child. This means that a new climb out of a left subtree needs to be executed. Hence the algorithm is conceived an alternation between climbing out of left subtrees and climbing out of right subtrees. While climbing out of a left subtree, we descend into potential right subtrees. The `function` is applied while climbing out of right subtrees and while climbing out of left subtrees that don't have a right subtree. This ensures the post-order semantics.

6.2.2 Breadth-First Traversal

Breadth-first traversing is most easily implemented in an iterative way. Instead of using a stack, a queue is used. In every phase of the algorithm, a node is considered and its children are added to the end of the queue. Since children are added later than siblings (because siblings were children of the parent node and have therefore already been added), siblings end up in the front of the queue while children end up in its rear. Hence, siblings are considered first. The Scheme code shown below is an explicit manifestation of these principles. The body of `loop` perpetually serves a node from the queue, processes it and enqueues both children before continuing. The traversal stops as soon as the queue is empty.

```
(define (breadth-first tree function)
  (define q (queue:new))
  (define (loop)
    (let ((node (queue:serve! q)))
      (function (value node))
      (if (not (null-tree? (left node)))
        (queue:enqueue! q (left node)))
      (if (not (null-tree? (right node)))
        (queue:enqueue! q (right node)))
      (if (not (queue:empty? q))
```

```
      (loop))))
(queue:enqueue! q tree)
(loop))
```

Breadth-first traversal is a popular traversal method in the context of so-called *game trees*. A game tree is a tree in which the nodes are situations in a game played by a player and the computer. For example, in chess, every situation corresponds to a particular configuration of pieces on a chessboard. Every time the player makes a move, the computer starts a game tree with the situation that resulted from the player's move as root. The children of this root node are all the possible board configurations which the computer can generate based on the root and by abiding by the rules of chess. All these generated configurations are siblings. They represent possible moves for the computer to answer the player's move. Then the computer starts "thinking ahead" by calculating the possible moves with which the player can answer all these generated moves in his turn. This generates new nodes that are children of one of the nodes that correspond to a move of the computer. As such, a game tree emerges in which the levels alternate between moves that might be made by the player and moves that can be made by the computer to answer those moves. The leaves of a game tree are game situations in which either the player or the computer has won the game.

If the computer has a complete view over the entire game tree, it can do a depth-first traversal of the tree in order to find out about the paths in the game tree that lead to a winning situation for the computer.

For simple board games like tic-tac-toe, this leads to manageable trees. However, for games with big boards and complex rules (like chess), this leads to a *combinatorial explosion* with millions of tree nodes. Unfortunately, such trees are too large to fit in the computer's memory. Moreover, most of the nodes will be generated in vein because the *actual* move of the player only corresponds to *one* of all those moves generated by the computer while it was "thinking ahead". That is why the computer will not generate the entire game tree but instead develop only a few levels of this tree every time the player makes an actual move. By "thinking a few levels deep" it can evaluate the solutions at that level and decide (by using some score-assigning evaluation function) which of the temporal solutions seems most promising to lead to victory for the computer. Clearly, this level-by-level consideration of the game tree is a breadth-first traversal.

6.3 Binary Search Trees

In section 1.4.3 we have introduced dictionaries. A dictionary is a storage data structure that allows one to remember key-value associations. We have formalized dictionaries by means of a `dictionary` ADT. In this section we start our study of implementation techniques for the ADT.

Because of the fact that dictionaries occur so often in computer science, it is important to have efficient implementations for the ADT. This is what most of

this section is about. After having presented a linear implementation of dictionaries (based on the `sorted-list` ADT of chapter 3) we present a tree-based implementation that exhibits far more attractive performance characteristics. The particular kind of trees used for this — binary search trees — is the focus of our study.

6.3.1 A List-based Implementation of Dictionaries

Among all possible linear implementations, dictionaries are best implemented by means of sorted lists. Remember from section 3.4 that sorted lists allow for a number of implementation techniques that render the implementation of `find` more efficient than what we can get by using an unsorted linear data structure. The code shown below demonstrates how to implement the `dictionary` ADT based on the `sorted-list` ADT presented in section 3.4.3. In the code, we assume that an implementation of the `sorted-list` ADT is imported in such a way that all imported names are prefixed by `slist:`.

Remember that dictionaries store key-value pairs. These are also known as dictionary associations. They are implemented by the following Scheme definitions:

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
```

The constructor of the `sorted-list` ADT requires an equality operator `==?` and an ordering `<<?` procedure that is used to put the elements in order. When storing associations in a sorted list, we have to provide an implementation for `==?` and `<<?` that works for associations. Obviously, these operations should only compare the keys of associations. The following code excerpt shows how to construct a dictionary given two operators `==?` and `<<?` that are to be used to compare keys. The `for-assoc-key` takes any procedure that works on keys and produces the equivalent procedure that works on associations. A lambda is returned that compares two associations by applying the corresponding operator on the keys of those associations. `for-assoc-key` is used to "extend" the `<<?` and `==?` procedures from keys to associations. These extensions are used by the sorted list implementation.

```
(define (for-assoc-key proc)
  (lambda (assoc1 assoc2)
    (proc (assoc-key assoc1)
          (assoc-key assoc2))))

(define (new ==? <<?)
  (slist:new
   (for-assoc-key <<?)
   (for-assoc-key ==?)))
```

This implementation of the `dictionary` ADT based on sorted lists is pretty straightforward as there is almost a one-to-one mapping between the operations specified by both ADTs. `insert!` takes a key and a value. It creates an association and stores it in the sorted list. `delete!` takes a key and creates an association with a "dummy" value. First, the association is searched for in the sorted list. If the association is found, the sorted list has a current and `delete!` on lists is used to delete the value pointed to by the current. Remember that both `add!` and `delete!` for sorted lists have performance characteristics that are in $O(n)$. `find` works similarly to `delete!`. An association with a dummy value is created to be used by `find!` for sorted lists. If this makes the current refer to a meaningful value, the association found is accessed by `peek` and its value is returned. Otherwise, #f is returned.

```
(define (insert! dct key val)
  (bst:insert! dct (make-assoc key val))
  dct)

(define (delete! dct key)
  (bst:delete! dct (make-assoc  key 'ignored))
  dct)

(define (find dct key)
  (define assoc (bst:find dct (make-assoc key 'ignored)))
  (if assoc
    (assoc-value assoc)
    assoc))
```

According to the knowledge accumulated in chapter 3, we obtain the following performance characteristics for `find`:

- Either we opt for the vectorial implementation of sorted lists. This results in a fast implementation for `find` (i.e. a performance characteristic in $O(log(n))$) because the vectorial implementation allows for the binary searching algorithm to be used. Unfortunately, the major drawback of this choice is that the size of the corresponding dictionaries is determined at the time of construction.
- Either we opt for the linked version of sorted lists. This results in a dynamically extensible dictionary that allows for an unlimited number of elements to be added (for as long as our computer memory is not full). However, the linked implementation doesn't allow for the binary searching algorithm. The resulting performance characteristic for `find` is in $O(n)$. This is unacceptable in realistic applications that rely on dictionaries containing thousands or even millions of records.

In the remainder of this chapter, we present binary search trees. This is a storage data structure that takes the best of two worlds. On the one hand it is a dynamically extensible data structure that does not inherit the boundary

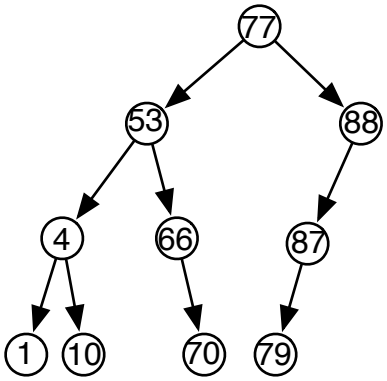

Figure 6.14: A Binary Search Tree

restrictions of the vector implementation of sorted lists. On the other hand, it is an data structure that allows for a logarithmic implementation of `find` and thus avoids the $O(n)$ performance characteristic of the linked list implementations. As usual, there is a price to pay. On the one hand, binary search trees require more pointers to "link up" the data structure. On the other hand, most of the algorithms that operate on binary search trees are quite complex.

6.3.2 Binary Search Trees

A *binary search tree* (or BST for short) is a binary tree that satisfies the following *BST condition*. For every node n in a BST we have:

- if n has a left child then the data element residing at the root of that left child is "smaller" than the data element residing at n.
- if n has a right child, then the data element residing at the root of that right child is "greater" than the data element residing at n.

Figure 6.14 shows an example of a BST. As can be observed from the drawing, all data elements that are smaller than the data element sitting in some node reside in the left subtree of that node. All data elements that are greater than the data element sitting in that node reside in the right subtree of that node. This means that the smallest data element of a BST is the leftmost data element of the tree. Likewise, the greatest data element of the tree is the rightmost data element of the tree.

The reason for organising a collection of data elements in a BST is that it enables us to implement an extremely efficient `find` operation that is very similar to the binary search algorithm. Given a key to search for and given a tree. Either the key is the one residing at the root of the tree. In that case, the key has been found. Otherwise, if the key to be found is greater than the

contents of the root, we have to continue searching in the right subtree of the root. Similarly, if the key is smaller than the contents of the root, then we have to continue searching in the left subtree of the root. As we will see, the implementation of the `find` operation has a best-case performance characteristic in $O(log(n))$. This is similar to the binary search algorithm. However, this time the searching algorithm operates on a linked data structure whereas the binary searching algorithm of section 3.4.4 only works for vectors.

The `BST` ADT is formally defined below. It's constructor takes a procedure `==?` for comparing elements and a procedure `<<?` for ordering elements. The latter is used by the BST to satisfy the BST condition. `insert!` takes a BST and a value. The value is added to the tree and the modified tree is returned. `find` takes a BST and a key to be searched for. If found, it is returned. `#f` is returned if the key does not occur in the BST. `delete!` searches for a value and removes it from the BST again. The modified tree is returned.

```
1  ADT BST< V >
2
3  new
4      ( ( V V → boolean)
5        ( V V → boolean) → BST< V > )
6  binary−search−tree?
7      ( any → boolean )
8  find
9      ( BST< V > V → V ∪ { #f } )
10 insert!
11     ( BST< V > V → BST< V > )
12 delete!
13     ( BST< V > V → BST< V > )
```

So let us implementing this ADT in Scheme. We start with the representation. A BST is represented as an enhanced list that stores a tag, a reference to a binary tree (as defined in section 6.1.2) and the two comparison procedures `<<?` and `==?`. The private procedures `root`, `root!`, `lesser` and `equality` are used to access the constituents of the enhanced list. Notice that an implementation of the `binary-tree` ADT is imported by prefixing all operations with `tree:`.

```
(define BST-tag 'binary-search-tree)
(define (make <<? ==?)
  (list BST-tag tree:null-tree <<? ==?))
(define (root bst)
  (cadr bst))
(define (root! bst node)
  (set-car! (cdr bst) node))
(define (lesser bst)
  (caddr bst))
(define (equality bst)
  (cadddr bst))
```

```
(define (new <<? ==?)
  (make <<? ==?))
```

`find` is probably the easiest procedure of this ADT implementation. It exploits the BST condition and works its way down the BST by using the `<<?` procedure stored in the BST. `find` starts at the root and searches the tree until it arrives at a leaf. This means that its performance is determined by the height of the tree. In the best-case we have a complete binary tree. We know from section 4.4 that the height of such a tree is $log_2(n)$ which results in a best-case performance characteristic for `find` that is in $O(log(n))$. However, as we will see in section 6.4, things can go seriously wrong with BSTs resulting in a worst-case behaviour that is in $O(n)$.

```
(define (find bst key)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (let find-key
    ((node (root bst)))
    (if (tree:null-tree? node)
      #f
      (let
          ((node-value (tree:value node)))
        (cond
          ((==? node-value key)
           node-value)
          ((<<? node-value key)
           (find-key (tree:right node)))
          ((<<? key node-value)
           (find-key (tree:left node))))))))
```

Inserting a node in a BST is not very difficult either. It is accomplished by the `insert!` procedure shown below. We descend our way down the tree in order to find the correct position of the node to be added. The idea is to find the location of the newly added value by looking for its location as if that value were already in the tree. Hence, the algorithm (implemented by `insert-iter` below) is exactly the same as the one used by `find`.

In order to attach the new node to the tree, we use a technique that is similar to the chasing pointers technique that was used to add elements to a sorted list in section 3.4.3. In the sorted list version of the chasing pointers technique, a 'previous' and a 'next' pointer are systematically followed while traversing the list. Whenever the next pointer is the (non-existing) location we are looking for, all we need to do is destructively change "the next of the previous" in order to add the element to the list. In the code shown below, the 'next' and 'previous' pointers are called `child` and `parent` respectively.

For BSTs, making the `parent` refer to a new `child` node is slightly more complex since the relation between the parent and the child is not unique: a parent has *two* children. This is where `insert-iter`'s `child!` parameter comes into

play: every time we descend in the left subtree, we pass along the `tree:left!` mutator as the value for `child!`. Hence, calling `child!` on a parent will actually call `tree:left!` on that parent and will thus change the left subtree of that parent. The same holds for the right subtree: every time we call `insert-iter` on the right subtree, we pass along `tree:right!` which results in calls of `child!` to update the right subtree of the parent. On the highest level, we use a lambda that makes the enhanced list refer to the newly added root. This procedure is only used when adding the very first node to the tree.

```
(define (insert! bst val)
  (define <<? (lesser bst))
  (let insert-iter
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root! bst child)))
     (child (root bst)))
    (cond
      ((tree:null-tree? child)
       (child! parent
               (tree:new val
                         tree:null-tree
                         tree:null-tree)))
      ((<<? (tree:value child) val)
       (insert-iter child tree:right!
                    (tree:right child)))
      ((<<? val (tree:value child))
       (insert-iter child tree:left!
                    (tree:left child)))
      (else
       (tree:value! child val)))))
```

Deleting a node from a subtree is more complex. First we have to find the node to be deleted. This is the task of the `find-node` iteration shown below. Again, the algorithm is identical to the one used by `find`. Once the node to be deleted is found, `delete-node` is called with the node to be deleted, its parent and the `child!` procedure that can be called to store a new child node in the parent. `delete-node` covers three cases:

- Either the node to be deleted is a leaf node (i.e. a node that has no children). Deleting the node is easy: all we have to do is use `child!` to make the node's parent refer to the empty tree.
- If the node to be deleted is a node that has one single subtree, then deleting the node boils down to making the parent of the node refer to that subtree.
- If the node is a node with two subtrees, then the situation is more complex. We cannot just remove the node since the parent can only refer to one node and we have two of them. The standard way to solve this is to

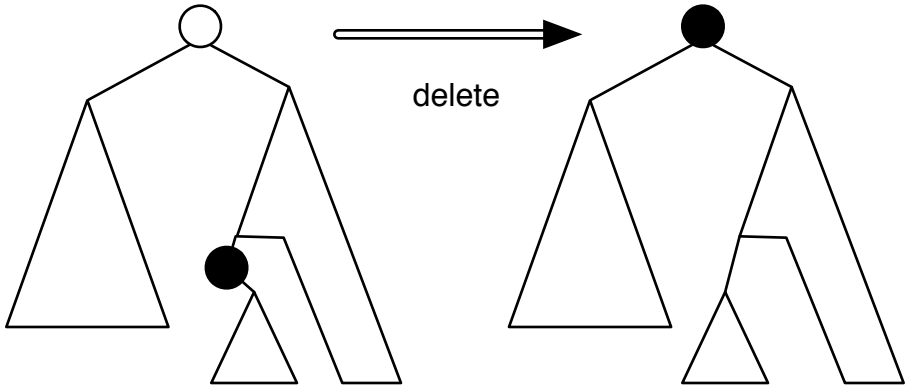

Figure 6.15: Delete from a BST

replace the node by another node that is also guaranteed to satisfy the BST condition and that requires the least possible number of modifications to both subtrees of the node we are deleting. Such a node is found in the leftmost position of the right subtree of the node to be deleted. The leftmost node in the right subtree is a node that contains a value that is the immediate successor of the value sitting in the node to be deleted (verify this!). Furthermore, it is a node that is guaranteed to have no left subtree since it is the leftmost node. As a result it is easy to remove that node from the right subtree by simply making its parent refer to its one and only subtree. This is depicted in figure 6.15. Finding the leftmost node of the right subtree is accomplished by `find-leftmost`. It works its way down the right subtree in a leftmost way. Having reached the leftmost node, its value is used to replace the value sitting in the node to be deleted. Hence, the node to be deleted is not really deleted. Its contents is simply replaced by the value sitting in the leftmost node of the right subtree. That node *is* deleted by making its parent refer to its right subtree. Scheme will garbage collect the node.

```
(define (delete! bst val)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (define (find-leftmost deleted parent child! child)
    (if (tree:null-tree? (tree:left child))
      (begin
        (tree:value! deleted (tree:value child))
        (child! parent (tree:right child)))
      (find-leftmost deleted child
                     tree:left!
```

```
                    (tree:left child))))
(define (delete-node parent child! child)
  (cond
    ((tree:null-tree? (tree:left child))
     (child! parent (tree:right child)))
    ((tree:null-tree? (tree:right child))
     (child! parent (tree:left child)))
    (else
     (find-leftmost child
                    child
                    tree:right!
                    (tree:right child)))))
(let find-node
  ((parent tree:null-tree)
   (child! (lambda (ignore child) (root! bst child)))
   (child (root bst)))
  (cond
    ((tree:null-tree? child)
     #f)
    ((==? (tree:value child) val)
     (delete-node parent child! child)
     (tree:value child))
    ((<<? (tree:value child) val)
     (find-node child tree:right! (tree:right child)))
    ((<<? val (tree:value child))
     (find-node child tree:left! (tree:left child))))))
```

What can we say about the performance characteristics for `find`, `delete!` and `insert!`? Clearly, they all have a performance characteristic that is in $O(h)$ where h is the height of the tree. After all, all three algorithms start at the root and work their way down the tree by selecting the left or right child every time. Hence, the longest path these algorithms might possibly walk down is precisely h branches long. When a tree of n elements is complete (i.e. completely filled on every level) then we know from section 4.4 that $h = \lfloor log_2(n) \rfloor$ which gives us a best-case performance characteristics for these operations which are in $O(log(n))$. In the worst-case, we have a *degenerated* tree in which every node has only one child. This is actually a linked list such that $h = n$. Hence, the worst-case performance characteristics for the operations is in $O(n)$.

6.3.3 A BST-based Implementation of Dictionaries

Let us now use the BST abstraction to implement the dictionary ADT. Just like in the sorted list implementation presented in section 6.3.1, associations group together a key and a value. They are the values to be used to store in the `BST`. The definitions for associations are exactly the same. We repeat them for the sake of completeness. Notice that we assume that the `BST` ADT is used

by importing an implementation and prefixing its procedures by `bst:`.

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
(define (for-assoc-key proc)
  (lambda (assoc1 assoc2)
    (proc (assoc-key assoc1)
          (assoc-key assoc2))))

(define (new ==? <<?)
  (bst:new
   (for-assoc-key <<?)
   (for-assoc-key ==?)))
```

The implementations for the dictionary operations `insert!`, `delete!` and `find` are shown below. They are straightforward translations to the corresponding operations defined on BSTs. Inserting a key-value pair into the dictionary is accomplished by inserting a new association in the binary tree. Finding and deleting a key makes us search for an association with that key. In the case of `find`, the value of the association is returned after it is retrieved from the BST.

```
(define (dictionary? any)
  (bst:binary-search-tree? any))

(define (insert! dct key val)
  (bst:insert! dct (make-assoc key val))
  dct)

(define (delete! dct key)
  (bst:delete! dct (make-assoc  key 'ignored))
  dct)

(define (find dct key)
  (define assoc (bst:find dct (make-assoc key 'ignored)))
  (if assoc
    (assoc-value assoc)
    assoc))
```

6.3.4 Discussion

The best-case $O(log(n))$ performance characteristic of the BST operations can be seriously disturbed if the tree is not a perfect (i.e. complete) tree. In the worst case, the tree only consists of nodes that have a left child or a right child (but not both). In that case, the tree is said to be *degenerated* and is in fact just a linear list. This means that the performance characteristic for `find` will be in $O(n)$.

Among the many possibilities to end up with a degenerated tree are the following three cases. They are displayed in figure 6.16.

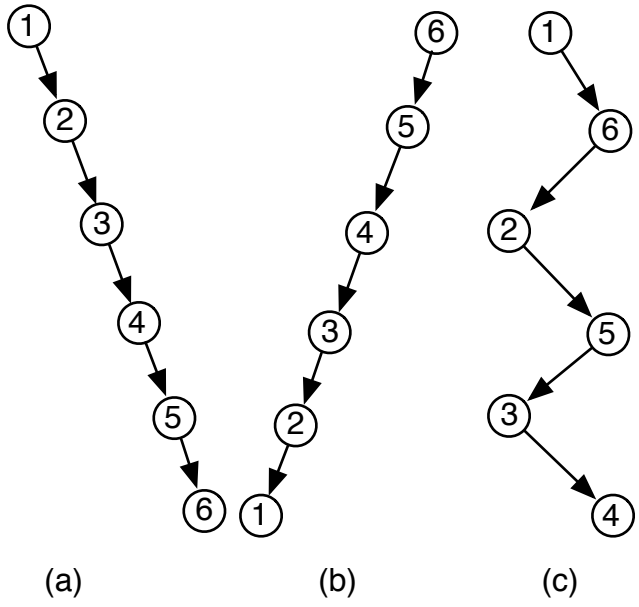

Figure 6.16: Three out of 2^{6-1} degenerated trees with 6 nodes

- When the elements of a binary search tree have been inserted in ascending order, the resulting tree is degenerated "to the right" as shown in part (a) of figure 6.16.
- When the elements of the binary search tree have been inserted in descending order, the tree is degenerated "to the left" as shown in part (b) of figure 6.16.
- Other schemes are possible as well: when the elements are inserted alternatingly outside-in, the tree is a linked list with alternating left branches and right branches as shown in part (c) of figure 6.16.

Degenerated trees are even worse than plain linked lists: the algorithms for manipulating and searching trees are more complex and require much more testing (e.g. to decide whether to descend to the left or to the right) than the equivalent algorithms that operate on plain linked lists. Hence, we get a performance that is even worse. Moreover, tree nodes have two pointers one of which is unused, whereas (single linked) lists have only one. Hence, degenerated trees are not only slower than linked lists, they also consume more memory.

What can we say about the probability of ending up with such degenerated cases? It is possible to prove that there are 2^{n-1} possible degenerated trees with n nodes. Although quite large, this number is *much* smaller (for large n) than the number of distinct binary search trees that can be constructed with

n elements, the so-called n'th Catalan number $b_n = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$. Hence, the odds for ending up with degenerated trees (or nearly degenerated trees) are quite small. It can be shown that the average tree that arises from random insertion of elements has a height which is $1.39log(n)$. Therefore, the average performance characteristic for BST operations is $O(log(n))$. However, for each average tree there are as many bad trees as there are good trees. Furthermore, this estimation only takes into account trees that have emerged from inserting elements without intermediate deletes. When deletes are also taken into account, little is known about the average behaviour of BSTs. To avoid the problems with BSTs, we introduce AVL trees.

6.4 AVL Trees

The insertion and deletion procedures of BSTs have no provisions whatsoever to make sure they produce complete trees. This is because guaranteeing complete trees at all times would be a costly matter as every insertion or deletion might cause the entire tree to be restructured. For instance, in order to make the tree in part (a) of figure 6.17 complete, a general restructuring is needed in order to obtain the tree shown in part (b) of figure 6.17. Hence, guaranteeing completeness would guarantee `find` to be in $O(log(n))$ but it would render the `insert!` and `delete!` extremely expensive since both `insert!` as well as `delete!` might cause a complete restructuring of the tree. We therefore opt for an intermediate solution that keeps the tree "reasonably balanced" after every insertion and deletion. Many valid definitions exist for "reasonably balanced trees". In this chapter, we call a tree to reasonable balanced when the difference in height between two subtrees of a node is never greater than 1. Trees that satisfy this definition of "balancedness" are known as *AVL Trees*. For example, both trees shown in 6.17 are a AVL trees. AVL trees can be shown to have a height that is maximally[1] $1.44log(n)$. AVL trees have been named in honour of the two russian mathematicians — G.M. Adelson-Velskii and E.M. Landis — who invented them.

The idea of AVL trees is to keep the tree balanced by adapting the insertion and deletion procedures of regular BSTs. The adapted versions implement the same algorithms but subsequently modify the tree in order for it to satisfy the AVL condition. Whenever the AVL condition is violated, the adapted insertion and deletion procedures apply a number of *rebalancing* procedures on the tree. The rebalancing procedures have to make sure that a tree is produced that does satisfy the AVL property. How can we know that a tree doesn't satisfy the AVL property? Surely it is not our intention to actually *calculate* the height of both subtrees after every single insertion or deletion. To avoid this, every node stores a balancing tag. We have three possible tags, namely `balanced`, `Lhigh` and `Rhigh`. They indicate whether both subtrees of the node are equally high,

[1] Notice that we found the *average* BST to be $1.39log(n)$ heigh. AVL trees are *always* $1.44log(n)$ high.

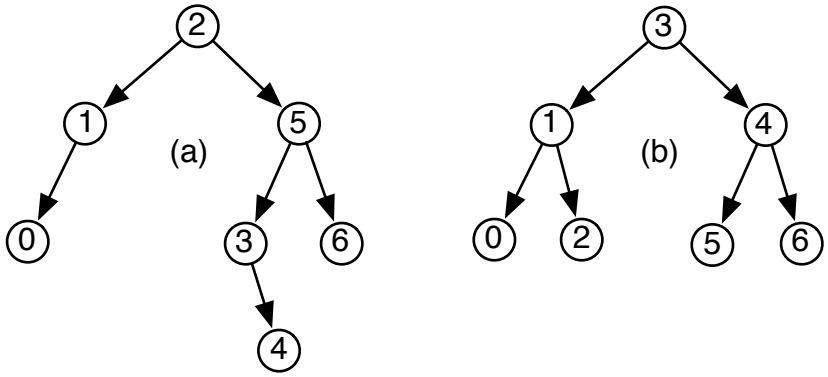

Figure 6.17: Restructuring a tree to make it complete

whether the left subtree is 1 level higher than its right subtree (it is "left high") or the other way around (it is "right high"). Other situations are prohibited by AVL trees. The following section presents this change in representation of tree nodes.

6.4.1 Node Representation

Taking the balancing information into account results in the following representation of AVL trees. It is a straightforward extension of the representation of ordinary BST trees explained in section 6.3.2. The `AVL-node` abstraction hides the way nodes are represented, namely by three pairs that store the node's value, its balancing tag, and references to its children.

```
(define balanced 'balanced)
(define Lhigh 'Lhigh)
(define Rhigh 'Rhigh)

(define null-tree '())
(define (null-tree? node)
  (eq? node null-tree))
(define (make-AVL-node left val bal right)
  (cons (cons val bal) (cons left right)))
 (define (AVL-node-left node)
  (cadr node))
(define (AVL-node-left! node left)
  (set-car! (cdr node) left))
(define (AVL-node-right node)
  (cddr node))
(define (AVL-node-right! node right)
  (set-cdr! (cdr node) right))
```

```
(define (AVL-node-value node)
  (caar node))
(define (AVL-node-value! node value)
  (set-car! (car node) value))
(define (AVL-node-balance node)
  (cdar node))
(define (AVL-node-balance! node b)
  (set-cdr! (car node) b))
```

Just like ordinary BSTs, AVL-trees are represented as enhanced lists that store a tag, two procedures and a reference to the root node of the tree.

```
(define AVL-tag 'avl-tree)
(define (make ==? <<?)
  (list AVL-tag null-tree ==? <<?))
(define (root avl)
  (cadr avl))
(define (root! avl r)
  (set-car! (cdr avl) r))
(define (equality avl)
  (car (cddr avl)))
(define (lesser avl)
  (car (cdddr avl)))

(define (new ==? <<?)
  (make ==? <<?))

(define (binary-search-tree? any)
  (and (pair? any)
       (eq? (car any) AVL-tag)))
```

In sections 6.4.3, 6.4.4 and 6.4.5 we discuss insertion, deletion and retrieval of information in AVL-trees. As explained before, insertion and deletion may have to rebalance a tree in order for it to satisfy the AVL condition again. Before we move on to the insertion and deletion procedures, we first explain how to rebalance a tree that has gotten out of balance.

6.4.2 Rebalancing by Rotation

Let us analyze what can go wrong when inserting a new data element in a tree and when deleting a data element from a tree. Consider the tree in figure 6.18(a). This tree is an AVL tree since the difference in height between both subtrees is only 1: the right subtree is one level higher than the left subtree. Now suppose that we want to insert a data element in the *right* subtree and suppose that this causes the tree to grow one level. The resulting right subtree is now two levels higher than the left subtree such that rebalancing is needed. This is shown in figure 6.18(b). Similarly, figure 6.18(c) shows an AVL tree before deletion of an element from the left subtree. Again this can cause the

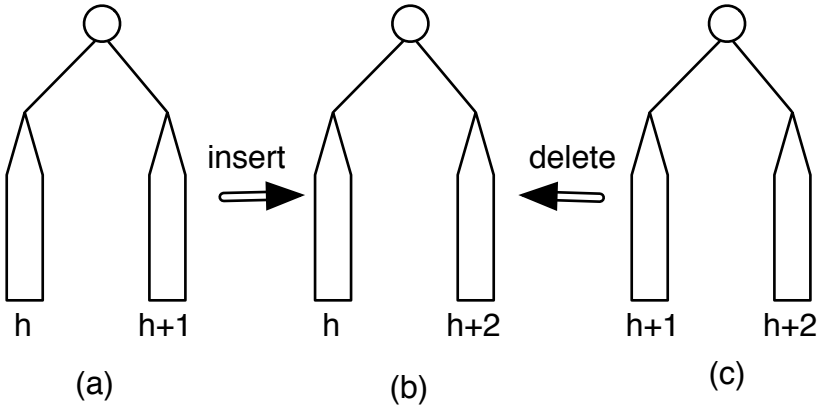

Figure 6.18: Rebalance Needed after Insertion or Deletion

left subtree to become one level less high. As shown in figure 6.18(b) this results in the same problematic tree. In what follows we will continue focussing on the situation depicted in figure 6.18(b). The analysis that covers the case of a *left* subtree that is two levels too high is entirely symmetric.

Balancing a tree is not just a matter of relinking some nodes. At all stages during the balancing process we have to guarantee the BST condition as well: all data elements sitting the left subtree of a node have to be smaller than the node's data element and all data elements sitting in the right subtree of a node have to be greater than the node's data element. To guarantee this invariant, rebalancing trees is accomplished by means of *rotations*. For a preview of what we mean by that, the reader is invited to have a glimpse at figures 6.19 and 6.20. Figure 6.19 displays a *single rotation* "to the left" and figure 6.20 displays a *double rotation*. A double rotation can be thought of as a combination of two consecutive single rotations. Figure 6.20 shows a double rotation that consists of a single rotation to the right followed by a single rotation to the left.

In Scheme, the `single-rotate-left!` procedure shown below implements a single rotation to the left. `single-rotate-right!` takes care of the symmetric case. A double rotation that consists of a single rotation to the right followed by another one to the left is implemented by the `double-rotate-right-then-left!` procedure. Again, a symmetric procedure exists as well.

```
(define (single-rotate-left! node)
  (define right (AVL-node-right node))
  (define left (AVL-node-left right))
  (AVL-node-right! node left)
  (AVL-node-left! right node)
  right)
```

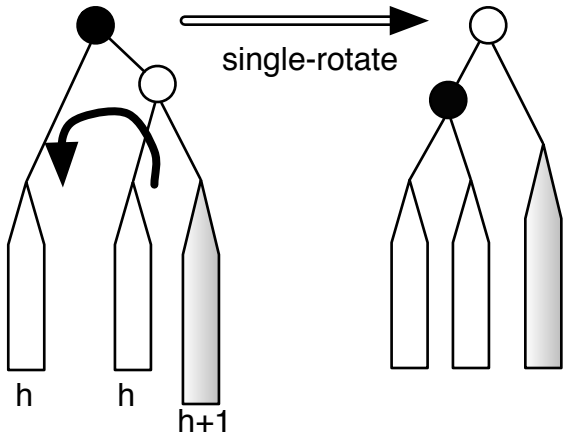

Figure 6.19: A Single Rotation to the Left

```
(define (single-rotate-right! node)
  (define left (AVL-node-left node))
  (define right (AVL-node-right left))
  (AVL-node-left! node right)
  (AVL-node-right! left node)
  left)

(define (double-rotate-left-then-right! node)
  (define left (AVL-node-left node))
  (AVL-node-left! node (single-rotate-left! left))
  (single-rotate-right! node))

(define (double-rotate-right-then-left! node)
  (define right (AVL-node-right node))
  (AVL-node-right! node (single-rotate-right! right))
  (single-rotate-left! node))
```

The rotation procedures take a `node` as their parameter. It is supposed to be the root of the tree to be rotated. In all four cases, they return a node which is the new root of the result tree. We invite the reader to verify that both single and double rotations preserve the BST conditions: the relative order between the nodes and the subtrees that are relinked is preserved at all times.

6.4.3 Insertion

The implementation of `insert!` is shown below. The basis of the algorithm is exactly the same as the insertion procedure for BSTs explained in section 6.3.2.

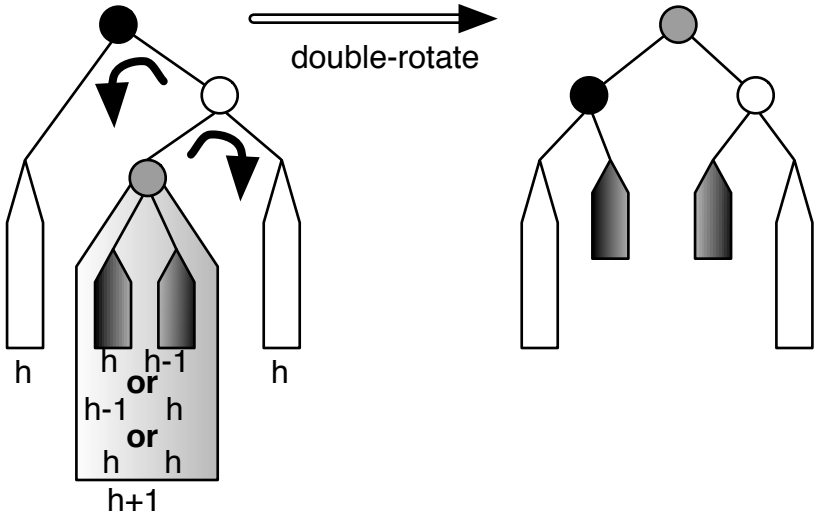

Figure 6.20: A Double Rotation Right-then-Left

By means of the recursive process defined by `insert-rec`, we descend down the tree in order to find the position for the node to be added. `insert-rec` returns a boolean value which indicates whether or not the subtree in which the insertion was achieved has grown one level. Hence, #t is returned the very first time we return from `insert-rec`. One level higher in the recursion, this will cause a call to `check-after-insert-right` or `check-after-insert-left` which update the balancing information and which verify whether or not a problematic subtree arises at that level. If this is the case, rotations are applied and #f is returned. Returning #f means that only one single rotation can take place during the insertion procedure. As a consequence, `insert!` has to descend in a tree that is $O(log(n))$ high, followed by a backtrack process that might cause just one single rotation. As a result, `insert!` is guaranteed to be $O(log(n))$.

```
(define (insert! avl val)
  (define <<? (lesser avl))
  (define ==? (equality avl))

  (let insert-rec
    ((parent null-tree)
     (child! (lambda (ignore child) (root! avl child)))
     (child (root avl)))
    (cond
      ((null-tree? child)
       (child! parent (make-AVL-node null-tree val balanced null-tree))
       #t)
```

```
      ((<<? (AVL-node-value child) val)
       (if (insert-rec child AVL-node-right! (AVL-node-right child))
         (check-after-insert-right parent child! child)
         #f))
      ((<<? val (AVL-node-value child))
       (if (insert-rec child AVL-node-left! (AVL-node-left child))
         (check-after-insert-left parent child! child)
         #f))
      (else  ; value = (AVL-node-value node)
       (AVL-node-value! child val)
       #f)))
  avl)
```

Consider `check-after-insert-right` which is called after an element was inserted in the right subtree. The other case is entirely symmetric. The procedure is shown below. If the original tree was `Lhigh`, then no rebalancing is needed. We merely have to change the balancing information of the node to `balanced`. #f is returned since the tree did not grow. When inserting an element in a right subtree of a tree that was balanced, then the resulting tree becomes `Rhigh` and the procedure returns #t since the tree has grown one level. However, after inserting in a right subtree that already was `Rhigh`, then the AVL tree condition becomes violated in the way we explained in figure 6.18(b). The resulting tree is too right-high which requires us to rotate the tree to the left, either by a single left rotation or by a double rotation that ends with a rotation to the left (i.e. "right-then-left"). In order to know exactly which of both rotations to apply, it is necessary to know which part of the right subtree is causing the problem. We have three possibilities. They are shown in figure 6.21 which is a more detailed drawing of figure 6.18(b). Either the outer (i.e. rightmost) subtree is of height $h + 1$ and causes the problem, or the inner (i.e. the middle) subtree is of height $h + 1$ and causes the problem. The rightmost situation where both subtrees are of height $h + 1$ was added to show that our reasoning is systematic and complete, but it is in fact not a real possibility: if both subtrees of the right subtree are $h + 1$ high, then one of the subtrees would already have been $h + 1$ high *before* inserting the new node. This means that the right subtree would have been $h + 2$ high *before* the insertion which is impossible given the fact that the tree was an AVL tree before the insertion. Hence, we only have to deal with the first two cases of figure 6.21. In the first case, we simply apply a single left rotation as shown in 6.19. In the second case, we apply the double rotation shown in figure 6.20.

The `check-after-insert-right` procedure shown below implements this analysis.

```
(define (check-after-insert-right parent child! child)
  (cond
    ((eq? (AVL-node-balance child) Lhigh)
     (AVL-node-balance! child balanced)
     #f)
```

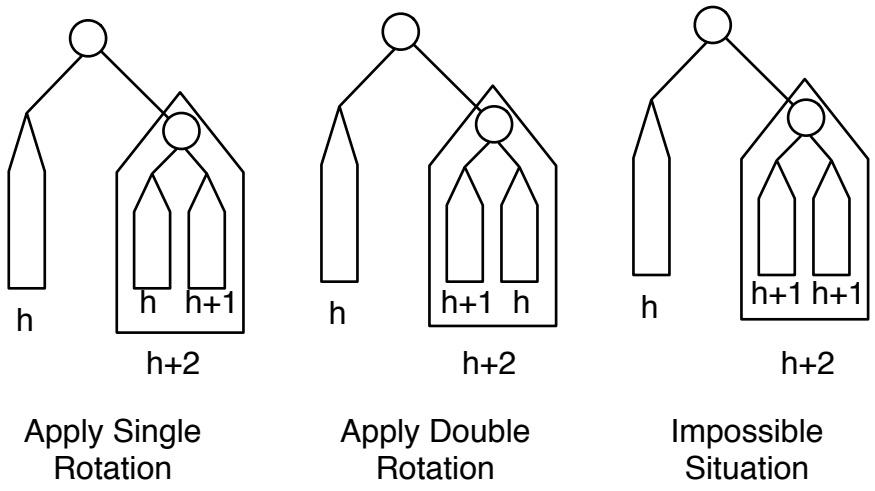

Figure 6.21: Different Causes of Problems after Insertion

```
((eq? (AVL-node-balance child) balanced)
 (AVL-node-balance! child Rhigh)
 #t)
(else ; child already was right-high
 (let* ((right (AVL-node-right child))
        (left (AVL-node-left right)))
   (if (eq? (AVL-node-balance right) Rhigh)
     (begin
       (child! parent (single-rotate-left! child))
       (single-rotate-left-update! child right))
     (begin
       (child! parent (double-rotate-right-then-left! child))
       (double-rotate-right-then-left-update! child right left)))
   #f))))
```

As explained before, rebalancing is accomplished by means of single rotations and double rotations. `check-after-insert-right` either calls `single-rotate-left!` or `double-rotate-right-then-left!`. Notice however that these procedures merely implement the rewiring of nodes. They don't update their balancing information. This is taken care of by the procedures shown below.

```
(define (single-rotate-left-update! node right)
  (cond ((eq? (AVL-node-balance right) Rhigh)
         (AVL-node-balance! node balanced)
         (AVL-node-balance! right balanced))
        (else
```

```
        (AVL-node-balance! node Rhigh)
        (AVL-node-balance! right Lhigh))))

(define (double-rotate-right-then-left-update! black white grey)
  (cond ((eq? (AVL-node-balance grey) Lhigh)
         (AVL-node-balance! white Rhigh)
         (AVL-node-balance! black balanced)
         (AVL-node-balance! grey balanced))
        ((eq? (AVL-node-balance grey) balanced)
         (AVL-node-balance! white balanced)
         (AVL-node-balance! black balanced)
         (AVL-node-balance! grey balanced))
        (else
         (AVL-node-balance! white balanced)
         (AVL-node-balance! black Lhigh)
         (AVL-node-balance! grey balanced))))
```

Updating the balancing information in the case of a single rotation is simple. The single rotation is applied when the both root node and its right child are `Rhigh`. Clearly, all we have to do is apply the single rotation and assign all balancing information to become `balanced` (the second branch of the conditional `single-rotate-left-update!` is used for deletion as explained below). This can be observed from figure 6.19. In order to understand the way the balancing information of the nodes is updated in the case of a double rotation, we refer back to the drawing in figure 6.20. The table shown in figure 6.22 translates the cases shown in figure 6.20 into rules for updating the balancing information of nodes in case of a double rotation to the left. The body of `double-rotate-right-then-left-update!` is a straightforward implementation of this table.

Grey Node:	`Lhigh`	`balanced`	`Rhigh`
White Node:	`Rhigh` $(h, h-1)$	`balanced` (h, h)	`balanced` $(h-1, h)$
Black Node:	`balanced` $(h, h-1)$	`balanced` (h, h)	`Lhigh` $(h-1, h)$
Grey Node:	`balanced`	`balanced`	`balanced`

Figure 6.22: Balancing Update for `double-rotate-right-then-left` .

6.4.4 Deletion

The `delete!` procedure is presented below. Just like the deletion procedure for BSTs described in section 6.3.2, we first have to locate the node to be deleted. This is the task of the `find-node` iteration. When the node to be deleted has been found it is the task of `delete-node` to actually delete the node. If the node has no children or only one child, then the deletion procedure itself is simple: the node is erased (by replacing it with the empty tree and relying on

Scheme's garbage collector to clean up the node) or it is replaced by its one and only subtree. Otherwise we have to find the leftmost element in the right subtree in order to find a suitable element to replace the deleted node. This is accomplished by the `find-leftmost` procedure. This is exactly the same algorithm that was used for deletion from BSTs.

```
(define (delete! avl val)
  (define ==? (equality avl))
  (define <<? (lesser avl))

  (define (find-leftmost deleted parent child! child)
    (if (null-tree? (AVL-node-left child))
      (begin
        (AVL-node-value! deleted (AVL-node-value child))
        (child! parent (AVL-node-right child))
        #t)
      (if (find-leftmost deleted child AVL-node-left! (AVL-node-left child))
        (check-after-delete-left parent child! child)
        #f)))

  (define (delete-node parent child! child)
    (cond
      ((null-tree? (AVL-node-left child))
       (child! parent (AVL-node-right child))
       #t)
      ((null-tree? (AVL-node-right child))
       (child! parent (AVL-node-left child))
       #t)
      (else
       (if (find-leftmost child child AVL-node-right! (AVL-node-right child))
         (check-after-delete-right parent child! child)
         #f))))

  (let find-node
    ((parent null-tree)
     (child! (lambda (ignore child) (root! avl child)))
     (child (root avl)))
    (cond
      ((null-tree? child)
       (error "object not in AVL tree(delete!)" val))
      ((==? (AVL-node-value child) val)
       (delete-node parent child! child))
      ((<<? (AVL-node-value child) val)
       (if (find-node child AVL-node-right! (AVL-node-right child))
         (check-after-delete-right parent child! child)
         #f))
      ((<<? val (AVL-node-value child))
```

```
      (if (find-node child  AVL-node-left! (AVL-node-left child))
        (check-after-delete-left parent child! child)
        #f))))
  avl)
```

The big difference between this code and the one for ordinary BSTs is that *all* these procedures potentially make the tree one level shorter. During the backtracking phase, it is therefore possible to arrive at a point where one subtree is two levels shorter than the other subtree. In that case, a rotation is needed to rebalance the tree at that level. This rotation rebalances the tree at that level but can make the resulting tree one level shorter than it was before the rotation as well. Hence, *every* step in the backtracking phase might trigger a rotation. Since the three iterations `find-node`, `delete-node` and `find-leftmost` descend the tree down to the deepest level, they eventually cause a backtracking process that takes $O(log(n))$ steps. This means that $O(log(n))$ rotations might be needed. The is not as bad as it may seem: a rotation is just a matter of manipulating some pointers. No expensive comparisons or data moves are involved.

Every procedure participating in the backtracking process returns a boolean that indicates whether or not the tree it processed has shrunk. If this happens, the tree that was encountered one level higher in the backtracking process is unbalanced causing another rotation. This has to be checked at all levels in the backtracking process. It is accomplished by the procedures `check-after-delete-left` and `check-after-delete-right`. The following code excerpt shows the latter. The former is the symmetric analogue.

```
(define (check-after-delete-right parent child! child)
  (cond
    ((eq? (AVL-node-balance child) Rhigh)
     (AVL-node-balance! child balanced)
     #t)
    ((eq? (AVL-node-balance child) balanced)
     (AVL-node-balance! child Lhigh)
     #f)
    (else
     (let* ((left (AVL-node-left child))
            (left-bal (AVL-node-balance left))
            (right (AVL-node-right left)))
       (if (or (eq? left-bal Lhigh)
               (eq? left-bal balanced))
         (begin
           (child! parent (single-rotate-right! child))
           (single-rotate-right-update! child left))
         (begin
           (child! parent (double-rotate-left-then-right! child))
           (double-rotate-left-then-right-update! child left right)))
       (not (eq? left-bal balanced))))))
```

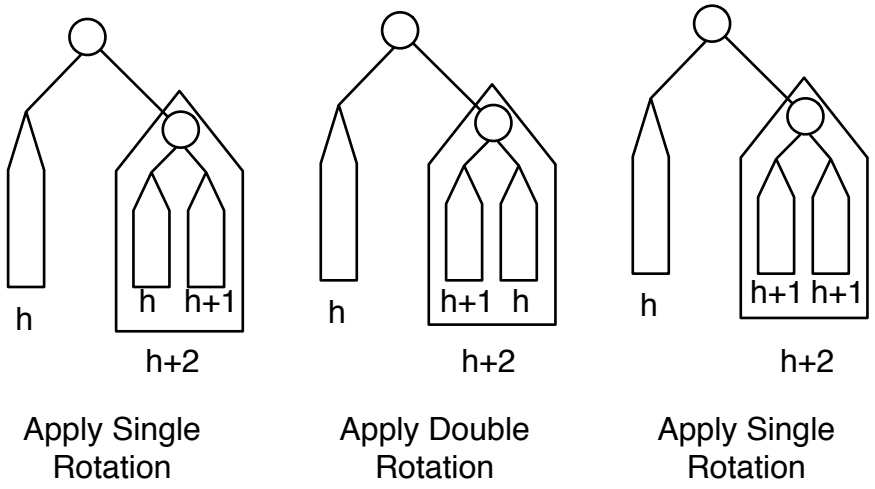

Figure 6.23: Different Causes of Problems after Deletion

In order to understand this procedure we refer to figure 6.23 which shows a more detailed version of the problem tree that was shown in figure 6.18(b). As for insertion, the problem that can arise is that a subtree gets too high after an element was deleted from the other subtree. By analyzing which part of the subtree that is too high causes the problem, we end up with the three possible situations depicted in 6.23. The three cases are handled by the if-test in the above procedure. Notice that, in contrast to insertion, we have three cases to cover this time. The third case can be handled using a single rotation. This is where the second branch of the `single-rotate-left-update!` and `single-rotate-right-update!` procedures comes into play.

6.4.5 Finding

The implementation of the `find` operation is exactly the same as the one for ordinary BSTs shown in section 6.3.2. The only difference is that its execution will always exhibit an $O(log(n))$ behaviour because the tree is guaranteed to be nearly balanced. Furthermore, the AVL conditions ensure us that the constant that is hidden in this asymptotic notation is about 1.45. We do not repeat the algorithm here.

6.5 Comparing Dictionary Implementations

Remember that our study of AVL trees was motivated by the observation that BSTs can exhibit degenerated behaviour causing `insert!`, `delete!` and `find` to have performance characteristics that are in $O(n)$ instead of $O(log(n))$. Having

studied AVL trees, we can now replace the BSTs used in section 6.3.3 by AVL trees. The resulting dictionary implementation has performance characteristics in $O(log(n))$ for all three operations.

We conclude by comparing the four implementations that we have studied for this ADT: one based on vectorial sorted lists, a second one based on linked sorted lists, a third one based on BSTs and a fourth one based on AVL trees. The table in figure 6.24 compares the performance characteristics for these implementations of the `dictionary` ADT. For the sake of completeness, we have added a column that shows the performance characteristics of an `dictionary` implementation that would use unsorted (double linked) positional lists.

Operation	Sorted List (Vector)	Sorted List (Linked)	Double Linked List	BST	AVL
`insert!`					
(worst)	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$
(average)	$O(n)$	$O(n)$	$O(1)$	$O(log(n))$	$O(log(n))$
`delete!`					
(worst)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$
(average)	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$	$O(log(n))$
`find`					
(worst)	$O(log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$
(average)	$O(log(n)))$	$O(n)$	$O(n)$	$O(log(n))$	$O(log(n))$

Figure 6.24: Comparative Dictionary Performance Characteristics

We can ask ourselves the question whether we can do better than the AVL implementation: is it possible to invent a clever data structure that allows us to beat the $O(log(n))$ performance characteristic for `find`? The answer is both "yes" and "no".

Remember from chapter 5 that it is impossible to beat the $O(n.log(n))$ performance characteristic lower bound set by advanced sorting algorithms, unless we leave the realm of general algorithms that solely rely on comparing keys. Given some extra knowledge about the internal structure of the keys, it was possible to devise linear sorting algorithms that beat the theoretical lower bound set by advanced sorting algorithms. However, it turns out that these sorting algorithms are not as generally applicable since restrictions have to be put on the structure of the keys.

The situation is very similar for information retrieval in dictionary implementations. It is possible to prove mathematically that no data structure that is solely based on comparisons can have an implementation for `find` that beats the $O(log(n))$ lower bound of balanced trees, unless that data structure is allowed to make extra assumptions about the internal structure of the keys it stores. Whenever this is the case, the internal structure of the keys can help us to obtain performance characteristics for `find` that are close to $O(1)$. This is the topic of the next chapter.

Chapter 7

Hashing

In chapter 6 we have studied a number of implementations for the `dictionary` ADT. In this chapter we continue our study of dictionaries. The underlying philosophy of the implementations given in chapter 6 was to come up with clever ways to organise data in a data structure with the goal of making `find` more efficient. We have seen that organizing the data as a sorted list already gives good results if we opt for a vector representation. For linked representations, linking up the data in the form of a BST was more beneficial. Keeping that tree balanced (e.g. using the AVL technique) still yields better results. The point of these organisational schemes is that they allow us to *search* the location of the data more efficiently.

The dictionary implementations presented in this chapter take a radically different stance. Instead of trying to organise the key-value pairs in as good a way as possible in order to render the search process faster, the idea presented in this chapter is to apply a special function h to the key in order to *compute* the location of the key in the vector-based data structure. The data structure is known as a *hash table* and h is known as a *hash function*. The technique is referred to as *hashing*. The underlying idea is to try to design the hash function such that it computes the location of the key in $O(1)$. As we will see in this chapter, this is not always possible. Nevertheless, hashing turns out to be an extremely effective technique for implementing dictionaries.

7.1 Basic Idea

Hash tables are a generalisation of plain vectors. Suppose that we have to build a dictionary that associates integer numbers between 0 and $M-1$ with values (i.e. the keys are numbers between 0 and $M-1$). In this special case, we can come up with an extremely fast implementation: we store the values in a Scheme vector `vector`. The index of the vector entry is the key. The data value stored in the vector is the value of the key-value pair. Inserting a new key-value pair (`k`,`v`) in the dictionary is extremely simple. All we have to do is store the value

in the vector location indicated by the key: (`vector-set! dict k v`). Similarly, launching a `find` for a given key `k` boils down to executing (`vector-ref dict k`). `delete!` can be realized by storing the empty list '(). The remarkable thing about this scheme is that `insert!`, `delete!` and `find` become $O(1)$ operations.

The idea of hash tables is to generalise this technique for arbitrary key data types. This requires two additional steps:

- First, a function h is needed that can be applied to any given key in order to transform the key to a number that can be used to access a vector. This function is known as a *hash function*. It is crucial that the hash function is in $O(1)$.
- Second, the number has to be made sufficiently small in order for it to be a valid index for the vector at hand. I.e., the number has to be smaller than the vector size minus one. This can be done using modulo-arithmetic: if the vector has space for M entries, then any number i can be downscaled to fit the vector by computing (`mod i M`).

Given any key k, then the number resulting from this computation is known as the *home address* of k. It is the location in the vector where we are expected to store (and thus also find) the dictionary value associated with k.

In order to give the reader an initial feeling about how to convert Scheme's data values to numbers, let us have a look at how this might be done for strings. First, we use `string->list` to convert a string to a list of characters. Subsequently, we can use `char->integer` to convert the characters to integers (i.e. the ASCII values of the characters). This is accomplised by mapping `char->integer` to every character of the string: (`map char->integer (string->list "hashing is fun")`). Finally, the numbers have to be combined in order to obtain one single number that can serve as a home address of the string key. E.g., we could decide to add the numbers. Section 7.3 explains why this algorithm is not a very good choice to serve as a hash function. Nevertheless it helps to understand the basic principle.

Although the basic ideas of hashing are extremely simple, several issues need to be resolved:

- How can we design good hash functions? For example, if the keys are student enrollment numbers of a university that consist of five digits $d_1d_2d_3d_4d_5$, then selecting the first two digits d_1d_2 is probably not a good choice since all students that have enrolled in the last couple of years are very likely to have the same value for these two digits (unless the university enrols more than 10000 *new* students every year). Hence, all these student enrollment numbers will be mapped onto the same home address in the hash table. This phenomenon is known as a *collision*.
- A hash function that avoids collisions is called a *perfect hash function*. As we will see in section 7.3.1, it is generally impossible to come up with a perfect hash function for a given set of keys. This means that we have to

accept that collisions are a fact of life. Perfect hash functions only exist in some very special cases when the exact nature and number of distinct keys is well-known in advance.

- Given a collision, then how do we resolve it. How can we make sure that two key-value pairs that have the same home address can be stored in the hash table anyhow? This is what we will call a *conflict resolution strategy*.

- What are good values for M? As we will see, it is easy to come up with extremely bad choices for M that cause all keys to be mapped onto a very limited set of home addresses. This causes the other home addresses in the table to be left unused most of the time. Needless to say, this is huge waste of memory.

In what follows, performance characteristics of several implementations for hash tables are studied. A central concept in these studies is a hash table's *load factor*. Given a hash table of size M that contains n key-value pairs at a certain moment in time, then the load factor α is the ratio that compares the number of elements contained in the hash table with the number of entries it supplies, i.e. $\alpha = \frac{n}{M}$. Stated otherwise, at any moment in time, a hash table is $\alpha \times 100$ percent filled. In hash table implementations that allow more than one key-value pair to reside in a table entry (such as the external chaining method presented in section 7.2.1), α can exceed 1. In order to keep the chances for collisions under control, it is advisable to keep the load factor below 1. As we will see, a good choice is to keep the load factor below 75%. Once the load factor exceeds this treshold, chances for collisions increase dramatically. It is then wise to make the hash table grow by *rehashing* all the elements into a new table. Usually, the table size is doubled whenever this phenomenon occurs.

7.2 Collision Resolution Strategies

Since perfect hash functions are nearly impossible to find, we devote most of our effort to the development of collision resolution strategies. Collision resolution strategies can be roughly classified into two groups, namely *external chaining* and *open addressing*.

7.2.1 External Chaining

The first collision resolution strategy we discuss is known as *external chaining*. External chaining was invented by H.P. Luhn in 1953. The basic idea is extremely simple: if two keys hash to the same home address in the hash table, a linked list is stored in the table. This turns the hash table into a vector of linked lists, also called *buckets*. An implementation that follows this collision resolution strategy is given below.

We start by presenting the representation of a hash table that relies on external chaining. A hashtable is represented by an enhanced list that stores a

vector, an equality procedure and the hash function to be used. Apart from the constructors and the private accessors, the code also presents the abstractions `make-assoc`, `assoc-key` and `assoc-value`. They are used to construct and access the dictionary associations to be stored in the hashtable.

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)

(define dictionary-tag 'external-chaining)
(define (make ==? M h)
  (list dictionary-tag (make-vector M '()) h ==?))
(define (storage table)
  (cadr table))
(define (hash-function table)
  (caddr table))
(define (equality table)
  (cadddr table))

(define (new ==? M h)
  (make ==? M (lambda (k) (mod (h k) M))))

(define (dictionary? any)
  (and (pair? any)
       (eq? (car any) dictionary-tag)))
```

`insert!` is shown below. It creates an association `assoc` and launches an iteration `insert-in-bucket` that starts at the home address of the key. A linear searching algorithm processes the linked list that forms the bucket of that particular home address. When arriving at the empty list, the new association is attached to the end of the list. In case the key is encountered during the searching process, the association is updated. Notice how a `next!` procedure is used to link the new node to its previous. In the first step of the iteration we bind `next!` to a procedure that correctly stores the first node of the list in the bucket. In subsequent steps, `next!` is just `set-cdr!`.

```
(define (insert! table key val)
  (define vector (storage table))
  (define h (hash-fct table))
  (define ==? (equality table))
  (define home-address (h key))
  (define assoc (make-assoc key val))
  (let insert-in-bucket
    ((prev '())
     (next! (lambda (ignore next)
              (vector-set! vector home-address next)))
     (next (vector-ref vector home-address)))
```

```
    (cond
      ((null? next)
       (next! prev (cons assoc next)))
      ((==? (assoc-key (car next)) key)
       (set-car! next assoc))
      (else
       (insert-in-bucket next set-cdr! (cdr next)))))
  table)
```

`find` is quite trivial. It calculates the home address of the given key and performs a lineair search in the bucket stored at the home address.

```
(define (find table key)
  (define vector (storage table))
  (define h (hash-fct table))
  (define ==? (equality table))
  (define home-address (h key))
  (let find-in-bucket
    ((next (vector-ref vector home-address)))
    (cond
      ((null? next)
       #f)
      ((==? (assoc-key (car next)) key)
       (assoc-value (car next)))
      (else
       (find-in-bucket (cdr next))))))
```

`delete!` is similar to `insert!`. The home address is computed and an iterative process searches the bucket for the association to be deleted. Once found, the association is removed be relinking the previous association to the next association. The association to be deleted is reclaimed by the Scheme garbage collector.

```
(define (delete! table key)
  (define vector (storage table))
  (define h (hash-fct table))
  (define ==? (equality table))
  (define home-address (h key))
  (let delete-from-bucket
    ((prev '())
     (next! (lambda (ignore next) (vector-set! vector home-address next)))
     (next (vector-ref vector home-address)))
    (cond
      ((null? next)
       #f)
      ((==? (assoc-key (car next)) key)
       (next! prev (cdr next))
       table)
```

```
    (else
     (delete-from-bucket next set-cdr! (cdr next)))))
table)
```

What can we say about the performance characteristics of these procedures given the assumption that the hash function h is in $O(1)$? The worst-case analysis is simple: all keys hash to the same bucket which causes the bucket to contain all n key-value pairs. Hence, in the worst case, `delete!`, `insert!` as well as `find` are all $O(n)$.

The average-case analysis is less dramatic. Suppose that every key is equally likely to be encountered and assume that the hash function *uniformly* distributes the keys over all the buckets. The latter assumption is not a trivial one as we will see in section 7.3. It boils down to the fact that the probability of a key to end up in one particular bucket is $\frac{1}{M}$. Given the fact that n elements reside in the dictionary, we can expect an average length of $\frac{n}{M} = \alpha$ for every bucket. We know from our study of linked lists in chapter 3, we find the average time spent in a linked list is $\frac{\alpha}{2}$. This results in the performance characteristics shown in figure 7.1 (the 1 stems from the fact that the hash function needs to be computed as well). Remember that, in general, it is the idea to keep α below 1. Hence, as long as $n \sim O(M)$, we have $O(1)$ behaviour in all three cases. Finally, notice that a hash table that uses external chaining can store more key-value pairs than there are table entries. In other words, it is possible that $\alpha > 1$. External chaining is the only collision resolution strategy that allows this.

Operation	Performance
`find`	$O(1+\alpha)$
`insert!`	$O(1+\alpha)$
`delete!`	$O(1+\alpha)$

Figure 7.1: Hashtable Performance Characteristics (External Chaining)

Often, the literature on hashing makes a difference between the performance characteristic for successful and unsuccessful searches. An unsuccessful search causes the entire list to be traverse which yields $\Theta(1+\alpha)$. A successful search is likely to have searched half of the list in general. Hence we get $O(1+\frac{\alpha}{2})$.

7.2.2 The table size

What can we say about the size of the hash table (i.e. M)? Does M have any influence on the performance characteristics of a hash table? In fact it does! Let us have a look at the following function $f(k)$ where $h(k)$ can be any hash function.

$$f(k) = h(k) \ mod \ M$$

The behaviour of this function is extremely important since it is used all the

time to keep the indexes produced by the hash function $h(k)$ within the range $[0 \ldots M-1]$.

Let us have a look at what happens when we use this function with $M = 25$ to store the numbers of the set $S = \{0, 5, 10, 15, 20, ..., 100\}$. For the moment we assume that $h(k) = k$. It is our goal here to study the influence of M instead of h. Given these premises, we notice that all elements of the set $\{0, 25, 50, 75, 100\}$ end up at $f(k) = 0$. All elements of the set $\{5, 30, 55, 80\}$ end up at $f(k) = 5$. As a result, *all* elements of S end up at the entries 0, 5, 10, 15 and 20. All other entries (i.e. 1, 2, 3, 4, 6 ,...) of the table remain unused. This phenomenon is known as *funneling*: (a subset of) the input ends up in a subset of the table entries. It is exactly the opposite of what we try to achieve with hashing, namely to uniformly *distribute* the key-value pairs as much as possible over the entire table in order to make collisions as unlikely as possible.

In order to understand the funneling phenomenon, consider the situation in which the numbers $h(k)$ and M have a multiplicative factor γ in common. In other words, let us assume that $h(k) = \gamma.r_{h(k)}$ and $M = \gamma.r_M$. Recall that, by definition, $a = b \ mod N$ if and only if $a = \beta.N + b$ for some β (which can be negative!). Then $f(k) = \beta.M + h(k) = \beta.\gamma.r_M + \gamma.r_{h(k)} = \gamma.X$. Hence, all hash values $h(k)$ that have a factor γ in common with M end up at some multiple of γ. In the above example, all hashed keys that have the factor 5 in common with 25 end up at some multiple of 5. Hence, how do we choose an M that has *no* factors in common with any hash value? The answer is to select a prime number for M. The best values for M turn out to be primes that are not too close to exact powers of 2. Another possibility to make sure that M and $h(k)$ have no factors is common is to make sure that 2 is the only factor of M (i.e. we choose M to be a perfect power of two) and that $h(k)$ does not have two as a factor (i.e. is an odd number).

7.2.3 Open Addressing Methods

Whereas external chaining collects colliding elements in linked lists, open addressing methods store such elements elsewhere in the same hash table. Upon encountering a collision for a certain key-value pair, the idea of open addressing methods consists of searching the hash table in order to find a free entry in which we can store that key-value pair. There are several techniques to accomplish this. They are the topic of this section. Resolving collisions with open addressing methods means that we necessarily have $\alpha \leq 1$ since we cannot store more key-value pairs in the table than the number of entries it provides. Open addressing methods were first proposed by G.M. Amdahl in the early 1950s.

In order to be able to store elements elsewhere in the same table, keys are *rehashed* whenever their home address turns out to be a location in the hash table that is already occupied. When the new location that is obtained after rehashing is occupied as well, the rehashing algorithm is repeated, until a location is found that is still available. Every such trial in the table is called *a probe*. Several such probes may be necessary. The result is called a *probe sequence*. Clearly, generating the probe sequence by subsequent rehashing is in

fact a search process in a linear list. Hence, the longer the probe sequences, the worse the performance of the hashtable's operations.

We will see several variants of the open addressing collision resolution strategies.

Linear Probing

The first open addressing method for resolving collisions is called *linear rehashing* or *linear probing*. If $h(k)$ is the original hash function, then linear probing causes the following entries in the hash table to be visited:

$$h'(k,i) = (h(k) + i.c) \ mod \ M \text{where } i \in \{1,2,3,4,...\}$$

for some constant c. In every repetition of the rehashing algorithm (i.e. $\forall i$), c is added to the home address in order to come up with the next probe. Very often c is simply chosen to be 1. This is called *single-slot stepping*. We can ask ourselves whether any value for c is acceptable. In fact, this is not the case. Suppose that c and M have a factor γ in common, i.e. $c = \gamma.r_c$ and $M = \gamma.r_M$. Because of the definition of *mod*, there is a (possibly negative) constant β such that

$$\begin{aligned} h'(k,i) &= (h(k) + i.c) \ mod \ M \\ &= \beta.M + (h(k) + i.c) \\ &= \beta.\gamma.r_M + h(k) + i.\gamma.r_c \\ &= h(k) + \gamma(\beta.r_M + i.r_c) \end{aligned}$$

This means, that for all i, $h'(k,i)$ will end up at the home addres $h(k)$ plus a $\gamma -$ fold. In other words, all table entries which are unreachable by adding $\gamma -$ folds to the home address (i.e. most of the table) remain unvisited. Hence, the table is not completely covered by subsequent probes; we seem to be repetitively jumping around at the same locations in the table. This funneling phenomenon is avoided if we make sure that c and M have no factors in common. It is said that c and M are *relatively prime*. For example, even though 6 and 35 are not prime numbers, they are relatively prime. The only factors of 6 are 2 and 3 whereas the factors of 35 are 5 and 7. When choosing c and M such that they are relatively prime, we get a *nonrepetitious complete coverage* of the table.

We present two implementations of linear probing. The difference lies in the way they deal with deletions from a table. Deleting is problematic because we cannot delete an entry by simply marking it as 'empty again. This would cause any probe sequence that contains the deleted entry to end at that entry. Indeed, suppose we have three keys k_0, k_1 and k_2 that are added to a hash table, one after the other. Suppose that k_1 collides with k_0 and has to be rehashed. Suppose that k_2 first has to be rehashed because of a collision with k_0 and suppose that this rehashing is not enough because of another collision with k_1.

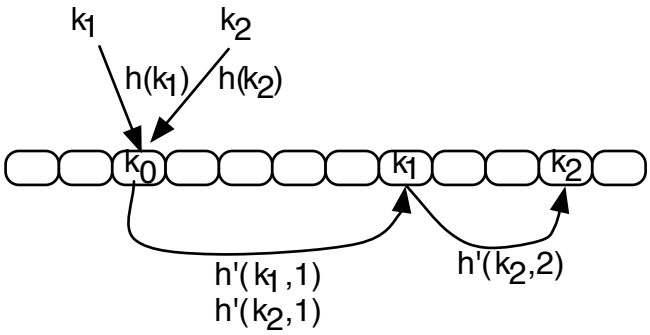

Figure 7.2: Deleting from Open Addressing Tables

This would result in a hash table whose structure is shown in figure 7.2. If we simply delete k_0 by marking the entry as 'empty, then k_1 and k_2 would no longer be found by `find`.

The first solution to this problem is the most popular one. It consist of replacing a deleted association by a special 'deleted marker that is known as *a tombstone*. For the insertion procedure, this marker has the same meaning as the initial 'empty tag that is used to mark empty table entries. When trying to insert an element that hashes to an entry containing 'deleted, we simply store the element (as if the entry would be marked 'empty). However, the interpretation of 'deleted is different from the interpretation of 'empty when searching the table (as in `find`)in a probe sequence. Any occurrence of 'deleted along this sequence is ignored and causes a rehash to generate the rest of the probe sequence as well. This tombstone technique is adopted in the following implementation.

The representation of the hashtable as an enhanced list is shown below. This part of the code is entire analogous to the code that uses external chaining.

```
(define dictionary-tag 'hashtable)
(define (make ==? M h)
  (list dictionary-tag ==? (make-vector M 'empty) h))
(define (equality table)
  (cadr table))
(define (storage table)
  (caddr table))
(define (hash-function table)
  (cadddr table))

(define (new ==? M h)
  (make ==? M (lambda (x) (mod (h x) M))))
```

Linear rehashing is implemented using the following definitions. We define the constant c and we define the function h' that computes $h'(k, i)$ given the

value of $h'(k, i-1)$.

```
(define c 1)
(define (rehash address M)
  (mod (+ address c) M))
```

The implementation of `insert!` follows. It creates a new association that represents the key-value pair to be stored. The `rehash-iter` iteration begins by initializing the `address` iteration variable to the home address of the key. When the corresponding table entry is 'empty or 'deleted, we use the entry to store the association. If the table already contains an association with the same key, we simply replace that association by the new association. The `else` branch of the conditional means that we have found a table entry that already contains an association that has a different key. This is a collision. Therefore, we continue the iteration with a rehashed value for `address`. That value is computed by calling the aforementioned `rehash` rehashing procedure.

```
(define (insert! table key val)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (define new-assoc (make-assoc key val))
  (let rehash-iter
    ((address (h key)))
    (let ((assoc (vector-ref vector address)))
      (cond ((or (eq? assoc 'empty)
                 (eq? assoc 'deleted))
             (vector-set! vector address new-assoc))
            ((==? (assoc-key assoc) key)
             (vector-set! vector address new-assoc))
            (else
             (rehash-iter (rehash address M))))))
  table)
```

`find` is very similar to `insert!`. We setup an iteration that starts at the home address of the key and that iterates by rehashing the key time and time again. Having found an association with the given key terminates `find` by returning the value that goes with the key. Once 'empty is encountered, it means that the entire probe sequence has been tried and that the key does not occur in the table. However, upon encountering 'delete we have to continue the iteration since the semantics of 'delete precisely prescribes that an element has been deleted that used to be part of a probe sequence.

```
(define (find table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
```

```
(define ==? (equality table))
(let rehash-iter
  ((address (h key)))
  (let ((assoc (vector-ref vector address)))
    (cond
      ((eq? assoc 'empty)
       #f)
      ((eq? assoc 'deleted)
       (rehash-iter (rehash address M)))
      ((==? (assoc-key assoc) key)
       (assoc-value assoc))
      (else
       (rehash-iter (rehash address M)))))))
```

`delete!` has to implement the correct semantics of the 'deleted tag. Again, we setup an iteration by starting at the home address and continue by calling the `rehash` function. Upon encountering an association with the key to be deleted, we replace the association by 'deleted. When the iteration encounters 'empty, the key to be deleted does not occur in the table.

```
(define (delete! table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (let rehash-iter
    ((address (h key)))
    (let ((assoc (vector-ref vector address)))
      (cond
        ((eq? assoc 'empty)
         #f)
        ((eq? assoc 'deleted)
         (rehash-iter (rehash address M)))
        ((==? (assoc-key assoc) key)
         (vector-set! vector address 'deleted))
        (else
         (rehash-iter (rehash address M))))))
  table)
```

This implementation of linear rehashing has a very important drawback: it makes `find` no longer dependent on the load factor itself but on previous values of the load factor. When deletions occur frequently, the hash table will be heavily polluted with 'deleted tombstones. This will have an effect on the efficiency. As a result it might be advisable to clean up the table now and then by considering all keys in the table and by inserting them in a newly created hash table. Clearly, this is an $O(n)$ operation.

The second implementation of linear rehashing properly deletes the element

the element to be deleted from the hash table. Unfortunately, this deletion method is only possible for linear probing with single-slot stepping (i.e. `c` = 1).

This proper deletion algorithm is implemented by the `delete!` procedure shown below. The idea is to iterate towards the address of the association to be deleted and to replace the association by the next association of the probe sequence that begins at that association. This process is repeated and causes all associations of that probe sequence to be moved "to the left". This process is very similar to the storage move process that was discussed in section 3.2.5. In other words, we traverse the entire probe sequence that starts at that home address in order to copy the elements of the probe sequence one position "backward" in the probe sequence. This is accomplised by the `storage-move` procedure shown below. It returns the address of the last association of the probe sequence which is subsequently filled with 'empty. Caution is required when copying associations: elements encountered in the probe sequence whose *home* address is in the probe sequence do not have to be copied. They belong to a different probe sequence that happens to align with the probe sequence under consideration. Hence, the iteration computes the home address `k` for every element encountered in the probe sequence. If the home address itself (circularly) lies between `prev` and `next`, then the element is skipped and we "follow the next pointer". If the home address is outside the probe sequence then the association must have arrived in the probe sequence by rehashing. Hence, it has to be copied "to the left" and both the `prev` and `next` pointers are shifted forward.

The implementations for `find` and `insert!` are not repeated. They only differ from the ones shown above in that they use 'empty instead of both 'empty and 'deleted to stop the iterations.

```
(define (delete! table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (define (between x <<1 y <<2 z)
    (and (<<1 x y)
         (<<2 y z)))
  (define (storage-move prev next)
    (if (eq? (vector-ref vector next) 'empty)
      prev
      (let ((home (h (assoc-key (vector-ref vector next)))))
        (if (or (between prev < home <= next)
                (between home <= next < prev)
                (between next < prev < home))
          (storage-move prev (rehash next M))
          (begin (vector-set! vector prev (vector-ref vector next))
                 (storage-move next (rehash next M)))))))
  (let rehash-iter
```

```
    ((address (h key)))
    (let ((assoc (vector-ref vector address)))
      (cond
        ((eq? assoc 'empty)
         #f)
        ((==? (assoc-key assoc) key)
         (vector-set!
          vector
          (storage-move address (rehash address M)) 'empty))
        (else
         (rehash-iter (rehash address M))))))
  table)
```

The Clustering Phenomenon

In our discussion of the proper deletion procedure we have already explained that elements occuring in one probe sequence can also be part of other probe sequence. This is called *clustering*. We distinguish between two kinds of clustering:

Primary Clustering means that once two keys k_1 and k_2 (with a *distinct* home address) rehash to the same address h somewhere along their probe sequences, then the rest of their probe sequences will be identical as well. Hence, primary clustering means that two different probe sequences "stick together" as soon as they have "met" because they happen to have one single rehashing address in common. Linear probing suffers a lot from primary clustering.

Secondary Clustering is a second phenomenon that makes probes sequences merge. Given two keys k_1 and k_2 that have an *identical* home address $h(k_1) = h(k_2)$. We speak about secondary clustering when the probe sequences of the keys cluster: for all i, we have $h'(k_1, i) = h(k_2, i)$. In other words, secondary clustering means that the two probe sequences of keys have joined together and never bifurcate again. Linear probing suffers from secondary clustering as well.

The consequence of primary and secondary clustering is that probe sequences tend to get longer and longer. Moreover, the probe sequences are no longer depending on α but also on the number of collisions that have occurred in the past.

Quadratic Probing

The second open addressing collision resolution method is called *quadratic rehashing* or *quadratic probing*. Quadratic probing is an improvement over linear probing that eliminates the problem of primary clustering. Instead of taking a constant c to bridge the gap between two consecutive probes, we make the step

size increase for every rehash. This can be done by allowing the step size to grow quadratically:

$$h'(k,i) = (h(k) + c_1.i + c_2.i^2) \ mod \ M$$

In order to make sure that all the entries in the hash table are visited (in other words, in order to achieve nonrepetitious complete coverage), we have to select c_1, c_2 and M carefully. There is no general rule for doing this. A good choice appears to be $c_1 = 0$, $c_2 = 1$ and M a prime number of the form $4k + 3$. Other possibilities exist as well, but most of the time c_1 is chosen to be 0.

Quadratic probing solves the problem of primary clustering: two keys with a distinct home address that — by accident — end up in the same location somewhere along their probe sequences will rehash to *different* locations in the next iteration of the rehashing algorithm. In other words, the probe sequences bifurcate again after the accidental merge. The reason is that the square used to rehash the first key is not the same as the square used to rehash the second one. In other words, both keys are rehashed using a different step size. The only exception to this is when both keys hash to the same location from their very first hash. In other words, quadratic probing does not solve the problem of secondary clustering.

The following code excerpt shows an implementation of quadratic rehashing. We only present the implementation for `find`. Just like with linear rehashing, `insert!` and `delete!` are based on the same algorithm. The only difference between `find` and `insert!` is that `insert!` doesn't distinguish between 'empty and 'deleted whereas `find` does. Just as with lineair rehashing, `delete!` replaces the association to be deleted by the 'deleted tombstone.

The algorithm starts an iteration by initializing the `address` variable to the home address of the key to be found. If `address` appears to be empty, then the key doesn't occur in the table. If the key to be found is equal to the key of the association sitting in `address`, then the iteration is successfully terminated by returning the value of the association. If the key is not equal to the key to be found or if `address` contains a tombstone, we continue the iteration by invoking the rehashing algorithm.

Rehashing is accomplished by adding the next square to the home address. However, instead of calculating the square in every iteration, we notice that two consecutive square numbers are always separated by an odd number. More precisely, the distance between two consecutive square numbers is an odd number that is the "next odd number" than the one that determines the distance between the previous two consecutive squares. In other words if we start from 0, we get consecutive squares by constantly adding the "next odd number". $0 \rightarrow (+1) \rightarrow 1 = 1^2 \rightarrow (+3) \rightarrow 4 = 2^2 \rightarrow (+5) \rightarrow 9 = 3^2 \rightarrow (+7) \rightarrow 16 = 4^2$....

```
(define (find table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
```

```
(let rehash-iter
  ((address (h key))
   (odd 1))
  (let ((assoc (vector-ref vector address)))
    (cond
      ((eq? assoc 'empty)
       #f)
      ((eq? assoc 'deleted)
       (rehash-iter (rehash address odd M) (+ odd 2)))
      ((==? (assoc-key assoc) key)
       (assoc-value assoc))
      (else
       (rehash-iter (rehash address odd M) (+ odd 2)))))))
```

The rehashing algorithm that is used in this procedure looks as follows. It simply adds the next odd number `odd` to the address to be rehashed. The odd number is updated in every iteration of `rehash-iter` shown above.

```
(define (rehash address j M)
  (mod (+ address j) M))
```

Double Hashing

The final open addressing method discussed is called *double hashing*. Double hashing avoids both primary clustering and secondary clustering. In order to avoid secondary clustering, we need to make sure that a rehash results in different step sizes even for keys that end up at the same home address. Like this, $h'(k_1, 0) = h'(k_2, 0)$ does not necessarily means that $h'(k_1, i) = h'(k_2, i)$ for future values of i.

Double hashing can be considered as a variant of linear rehasing, but for which the step size is recomputed in every iteration of the rehashing phase. In other words, the constant c of the lineair probing method is replaced by a value that changes in every iteration. To achieve this, two separate hash functions h_1 and h_2 are used:

$$h'(k, i) = (h_1(k) + i.h_2(k)) \ mod \ M \text{ where } i \in \{1, 2, 3, 4, ...\}$$

Double hashing avoids both primary clustering and secondary clustering since the step-size to be taken is recomputed again and again for every distinct key, whereas the step size was identical for all keys in linear rehashing. Again, we have to ask ourselves the question which are the good values for this technique to cover the entire hash table.

From the discussion of section 7.2.2, we know that we have to make sure that M and $h_2(k)$ are relative prime, for if they share a factor γ, then subsequent rehashes tend to funnel in the γ-folds. We can assure that M and $h_2(k)$ are relative prime by:

- taking M to be a power of two (such that 2 is its only factor) and taking h such that it always returns an odd number (such that 2 is not a factor).

- taking M to be prime and $h_2(k)$ to be smaller than M (such that it does not share a factor with M). A possibility for prime M might be $h_1(k) = k \ mod \ M$ and $h_2(k) = 1 + (k \ mod \ M')$ where $M' = M - 1$ or $M = M - 2$.

The code for double hashing is shown below. The representation of the data structure is slightly different from the representation used in the lineair probing and quadratic probing methods. The reason is that we have to store two hash functions in the hash table instead of just one. The representation looks as follows.

```
(define dictionary-tag 'hashtable)
(define (make ==? M h1 h2)
  (list dictionary-tag ==? (make-vector M 'empty) (lambda (x) (mod (h1 x) M)) h2))
(define (equality table)
  (cadr table))
(define (storage table)
  (caddr table))
(define (hash-function1 table)
  (cadddr table))
(define (hash-function2 table)
  (caddr (cddr table)))

(define (new ==? size h1 h2)
  (make ==? size h1 h2))
```

In contrast to the linear probing method and the quadratic probing method, double rehashing uses the second hash function to calculate a probe given an address that results in a collision:

```
(define (rehash key address h2 M)
  (mod (+ address (h2 key)) M))
```

Below, the implementation of `find` is presented. Again, `insert!` and `delete!` are omitted because of the fact that they are almost identical to `find`. The algorithm is very similar to the `find` algorithms presented above. The only difference is that the `rehash` function needs access to the second access function stored in the hash table.

```
(define (find table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h1 (hash-function1 table))
  (define h2 (hash-function2 table))
  (define ==? (equality table))
  (let rehash-iter
    ((address (h1 key)))
    (let ((assoc (vector-ref vector address)))
      (cond
```

```
((eq? assoc 'empty)
 #f)
((eq? assoc 'deleted)
 (rehash-iter (rehash key address h2 M)))
((==? (assoc-key assoc) key)
 (assoc-value assoc))
(else
 (rehash-iter (rehash key address h2 M)))))))
```

Double hashing performs *much* better than linear probing or quadratic probing. It can be shown that linear probing and quadratic probing use $\Theta(M)$ different probe sequences, whereas double hashing generates about $\Theta(M^2)$ different probe sequences. The more different probe sequences the technique generates, the less clustering we have.

Open Addressing Performance

Let us now have a look at the efficiency of the three open addressing methods discussed. Instead of making the analysis for `find`, `insert!` and `delete!`, the analysis focusses on *succesful* and for *unsuccesful* searches. This obviously covers the performance characteristics for `find` and `delete!`. It also covers the behaviour of `insert!` since an insertion corresponds to an unsuccesful search that is followed by the creation of a new association which is stored at the free location that was discovered by the unsuccessful search.

Unfortunately, analysing the behaviour exhibited by the processes for successful and unsuccessful table searches is much more complicated than the simple analysis that can be done for the external chaining technique. Quite a lot of probabilistic mathematics is needed in order to derive the exact average-case performance characteristic expressions. Therefore, we don't discuss these calculations but merely present the results in figure 7.3. For the sake of completeness, the bottom row repeats the results from figure 7.1 for external chaining.

	Unsuccesful	Succesful
Linear rehashing	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$
Double rehashing	$\frac{1}{1-\alpha}$	$-\left(\frac{1}{\alpha}\right)\times log(1-\alpha)$
External Chaining	$1+\alpha$	$1+\frac{1}{2}\alpha$

Figure 7.3: Hashtable Performance Characteristics

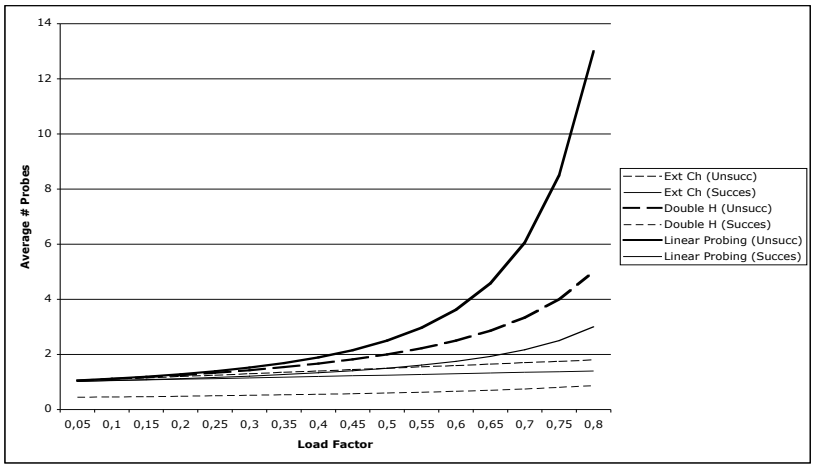

Figure 7.4: Performance Characteristics of Resolution Strategies

Since some of these formulas are not very meaningful, we have chosen to represent them graphically in figure 7.4. As we can see from the drawing, it is wise to keep the load factor below 75% for open addressing methods.

The table shows that external chaining is quite an attractive alternative. External chaining allows α to exceed 1 which is impossible for open addressing methods. Moreover, external chaining has the advantage that insertion and deletion are simple operations that never really require elements to be rehashed. This is in contrast to the open addressing methods. First, a table that uses an open addressing method can be full. If this happens, we need to allocate a bigger table and rehash all the elements from the old table into the new table. In fact, this already happens as soon as the load factor becomes greater than 75 % because open addressing methods start to exhibit unacceptable performance beyond this treshold. External chaining allows table resizes to be postponed much longer since the performance decreases only gradually (i.e. linearly). Second, deletion in open addressing methods can result in a table that has many occurrences that are marked with the 'deleted tombstone. After several deletions, open addressing methods get slow because the 'deleted tombstones do not make probe sequences shorter. In other words, after many deletions, open addressing methods get slow even for small values of α. The solution consist of "cleaning up the table" by removing all the 'deleted tombstones and rehashing all the elements still sitting in the table.

A drawback of external chaining is that the overhead of searching linked lists can become significant as the size of the lists grows. One might wonder whether we can replace the linked lists in the buckets by more efficient data structures such as AVL trees. Unfortunately, this only turns out to be cost-effective when a large number of elements end up in one bucket; i.e. when α is much bigger

than 1. But then again, it is probably more cost-effective to resize the table and rehash everything from scratch. Another drawback of external chaining is that the amount of memory needed is not known upfront. This can be a burden if the technique is to be applied for devices that have limited amounts of memory.

7.3 Hash Functions

In section 7.1 we have said that collisions are a fact of life and that it was more instructive to study the collision resolution strategies before studying hash functions. The phenomena encountered during our study of collision resolution strategies have to be avoided when designing a hash function. So what makes a good hash function? In this section, we first show that perfect hash functions are so rare that they are often not worthwhile looking for. Subsequently, we discuss a number of properties that good hash functions should satisfy. Finally, a number of popular hash functions are briefly explained.

7.3.1 Perfect Hash Functions

A *perfect hash function* is a hash function that yields no collisions. In other words it guarantees that $h(k_1) = h(k_2)$ implies $k_1 = k_2$.

In probability theory, the *birthday paradox* states that given a group of 23 (or more) randomly chosen people, the probability is more than 50% that at least two of them will have the same birthday. For 60 or more people, the probability is greater than 99%, although it cannot actually be 100% unless there are at least 366 people. This is not a paradox in the sense that it leads to a logical contradiction; it is described as a paradox because it is a mathematical truth that contradicts our naive intuition: most people would estimate that the chance is much lower than 50%. Because of the birthday paradox, it is quite difficult to come up with a realistic number of home addresses that do not collide sooner or later, even if the function that generates them is uniformly distributed (i.e. every home address is equally likely to occur).

7.3.2 Good Hash Functions

Two phenomena make it hard to come up with good hash functions:

Funelling is the phenomenon which causes entire parts of the table never to be considered for (some subset of) the keys. Whenever hashing a key, the possibility arises that we keep on stepping in circles in the table even though there is plenty of free space in the table.

Clustering is the phenomenon which causes probe sequences of distinct keys to merge in some cases. We have made a distinction between primary clustering and secondary clustering.

It is not easy to come up with good hash functions that avoid clustering and funnelling. A lot depends on the particular nature and amount of the keys that have to be stored in the table. As a consequence, creating good hash functions is craftsmanship rather than science. In general, hash functions have to be

Simple: The more complex a hash function, the harder it gets to derive its funnelling and clustering characteristics. Good hash functions are typically simple arithmetical computations.

Fast: A good hash function is to be fast since the entire idea of hashing is to mimik the $O(1)$ direct access performance characteristic of vectors. Speed is often a result of simplicity.

Strong: A strong hash function is a hash function that uniformly distributes the keys over the entries of the table. In other words, given a randomly selected key, then the probability that it ends up at a given entry should be $1/M$ if M is the number of entries in the hash table.

In what follows, we discuss a number of hash functions. Without loss of generality, we can allow ourselves to focus on hashing Scheme's integer numbers. All other Scheme data types can be mapped onto numbers in a fairly straightforward manner. For example, a string can be converted to a number by converting its characters to a number one by one (using `char->integer`) and by combining these numbers such that every character has an equal share in the final result. If the numbers associated with characters can vary between 0 and 255 (as is the case in programming languages that use the ASCII-standard), then a string "abc" can be converted into the number $v_a \times 256 \times 256 + v_b \times 256 + v_c$ if v_a is the number associated with "a", v_b is the number associate with "b" and v_c is the number associated with "c".

So let's assume that keys consists of a (possibly large) integer number. The following hash functions provide us with ways to convert them into smaller numbers that can be used as indexes in the hash table.

The Folding Method

An extremely popular hash function is the *folding method*. The idea is to take all the digits of a key and to combine them, simply by adding them or by applying `bitwise-xor` to them (e.g. (`bitwise-xor 9 4 3 7 5`)). In the case of addition, adding 5 digits yields a number between 0 and 45 that can be used immediately as the index in a hash table. The general pattern for folding a key $k = d_1 d_2 d_3 ... d_n$ is $h(k) = d_1 + d_2 + d_3 + ... + d_n$. In practice, folding performs quite poorly. One of the reasons is that the relative order between the digits plays no role whatsoever in the final value returned by the hash function. E.g., $h(12345) = h(32145) = h(54123) = ...$ All keys the digits of which are permutations of each other hash to the same home address.

The Digit Selection Method

Digit selection is a popular hashing method as well. The idea is to take a key that consist of n digits, $k = d_1d_2d_3...d_n$ and to select a combination of digits $h(k) = d_id_jd_k$ that results in a good uniform distribution. Care must be taken with digit selection. If we were to take role numbers of students (which typically consist of five digits), then selecting the last three digits will cause all recently enrolled students to hash to the same locations. This is because the first two digits only change after 1000 students have been enrolled. Similarly, when taking phone numbers of a city, then selecting the first few digits will be a bad choice since these digits are identical for (most of) the numbers of that city.

In practice it is wise to perform a *digit analysis* for a large number of concrete keys that can occur in a given situation. If this analysis reveals that there is a strong dependency between digit d_3 and digit d_9 then it is not wise to include both digits in the digit selection since their presence together reduces the number of different hash values that can be obtained. In other words, a high degree of funnelling is the result. The digits with the lowest interdigit correlations are to be selected.

The ultimate goal of a digit analysis is to select digits which *achieve avalanche*. Avalanche means that a difference of one single bit in the input guarantees that half of the bits in the output are different. If avalanche is achieved, it means that every bit (which can takes two different states, namely 0 or 1) in the input is used to its full extent and has a large effect on the output, irrespective of the effect of the other bits in the input.

The Division Method

The idea of the division method is to take any key k and "truncate" it by considering the remainder of division by M, the size of the hash table. In other words, we obtain $h(k) = k \ mod \ M$. This is one of the most popular hash functions since it is simple and performs quite well. We have seen that this requires k and M to have no common factors, which is most easily obtained by keeping M a prime number. We can also choose M to be a perfect power of 2 since this will neatly divide the keys into the even and odd entries of the table. This option has the benefit that it allows the hash table to be resized in extremely simple a way: just double the table size in order to obtain the next perfect power of two. However, this can be a dangerous choice as well. Since all computer data is — eventually — represented in binary format, dividing by a perfect power of two, say $M = 2^a$ (and taking the remainder) will only select the a least significant bits of the key. Unless we are sure that all a-bit patterns are equally likely to occur, it can cause serious funnelling. In practice, primes yield the best results. Their most important drawback is that prime numbers are not easily computed. Hence, we need to keep track of the "next prime number" every time the table needs to be resized. This requires us to store a predefined number of prime numbers.

The Multiplication Method

The idea of the multiplication method is to multiply the key k with some constant C (with $0 < C < 1$). The fractional part of this real number is then multiplied by M. In other words, $h(k) = \lfloor M.(kC \ mod \ 1) \rfloor$ (where the mod 1 operation takes the fractional part of a number). A popular value for C is $C = (\sqrt{5} - 1)/2 \approx 0.618034$. It generates a fairly good distribution. As an example for $k = 123456$, $M = 10000$ and $C = 0.618034$, then $h(k) = \lfloor 1000 \times (123456 \times 0.618034) \ mod \ 1 \rfloor = \lfloor 10000 \times 0.005504 \rfloor = 55$. The benefit of the multiplication method is that the distribution is being taken care of by C which makes the choice of M less critical. The method works well when M is chosen to be a perfect power of 2.

7.4 A Taxonomy of Dictionaries

Hash tables trivially serve to implement the `dictionary` ADT specified in section 1.4.3. At this point we like to draw the reader's attention to compare the specification of the ADT with the search-based implementations of chapter 6 and the hash-based implementations presented here. The focus of our attention is the constructor `new`.

Search-based implementations store the dictionary's key-value pair in a (usually linked) data structure. When launching `find`, a search-based implementation traverses the dictionary and *compares* the search key with the keys that the algorithm encounters during the traversal. The idea of search-based dictionary implementations is to organise the elements of a dictionary in a way as clever as possible in order to speed up the search. In chapter 3 we have seen that ordered data structures perform substantially better than unordered ones. This is confirmed in the tree versions of the dictionary ADT represented chapter 6.

The implementation of the constructor for search-based dictionary implementations not only requires an `==?` operator (as prescribed by the `dictionary` ADT definition) but also an additional ordering operator `<<?`. This was the case in all three search-based implementations of the ADT (the `sorted-list` implementation, the `BST` implementation and the `AVL` implementation). Dictionaries that rely on an order between their keys are called **ordered dictionaries**. Although we did not use the ordering relation between the keys in the ADT description, this leads to interesting operations. E.g., we might consider extending the `dictionary` ADT with a set of navigational operations such as `first`, `has-next?` and `next!`. Implementing these operations by using the order `<<?` and relying on a "current" stored in the enhanced list is a trivial programming exercises.

Hashed implementations try to avoid searching for keys by requiring the keys to map themselves to their home address in the storage data struc-

ture. Whenever searching for a given key with `find`, the hash function is applied to the key to be searched for. In hashed dictionary implementations the constructor `new` does not need the ordering that is needed by ordered dictionaries. Dictionaries that don't rely on an ordering relation between the keys are called **unordered dictionaries**. They are easily implemented by means of a hash table. Conversely, it is hard to implement ordered dictionaries by means of hash tables since hash table entries don't know the notion of the "next element". The whole idea of hashing is to distribute the keys as much as possible in order to avoid collisions. Surely we can store additional "next" pointers in the hash table entries. But then insertion and deletion need to take these into account which brings us back to linked lists.