Chapter 14

External Storage: Basic Concepts

All the algorithms and data structures that we have presented so far have been designed with the implicit assumption that they are to be executed on a computer that has enough memory in order to contain all the data. In this chapter, we begin our study of data structures and algorithms that are about data, the size of which is too big to fit in our computer's central memory. Instead, the data is stored on "external" storage mediums such as disks, solid state drives, tapes or CDs.

Storing data on external storage mediums is not only useful when the data is too big to fit in the computer's central memory. We all know that, for instance in a text processing program, one regularly has to "save" ones work before logging off or switching off the computer. This means that the text processing program must store its data (i.e. your text) on a storage medium which provides "persistent" memory; i.e. memory that does not lose its contents when power is shut off. As we will see, the algorithms and data structures studied so far are not trivially applicable in the context of these external storage mediums. The particular hardware characteristics of these storage mediums will force us to rethink some of the solutions that have been presented. Data structures that support persistency are therefore fundamentally different from the ones we have seen so far. They form the contents of this and the following chapters.

14.1 What is "Data" Anyway?

Until now, we have been using Scheme's built-in data values. In chapter 1, we have presented a list of data values and their corresponding data type. But as we all know, a computer is a machine that relies on electrical current in order to represent data. Even though it is an oversimplification, we can state that a computer actually only knows two kinds of data: current and no current. Abstractly spoken, we think of these as 0 and 1. Hence, deep down, a computer

is a machine that only knows how to handle digits that can take on two different values, namely 0 and 1. They are called binary digits or *bits*.

14.1.1 Bits and bytes

The bit is the smallest amount of information that can be manipulated and stored by a digital computer. In order to represent more complex information forms, we have to combine multiple bits. Historically, computer scientists have been grouping bits "by eight". According to the urban legend[1], this is the origin of the word *byte*. Hence a byte is just a group of eight bits. Digital computers just happen to be built in such a way that they can easily perform actions on groups of eight bits.

Using two bits, we can represent four different pieces of information (namely 00, 01, 10 and 11). Using three bits, we can represent eight pieces of information (namely 000, 001, 010, 011, 100, 101, 110 and 111). In general, using n bits, 2^n different pieces of information can be represented. Since eight bits allow us to represent $2^8 = 256$ different pieces of information, we can represent any number between 0 and 255.

It is fairly easy to associate a bit sequence with its corresponding decimal number. Suppose that we have a byte $b_7b_6b_5b_4b_3b_2b_1b_0$. The corresponding decimal number is then given by $2^0.b_0 + 2^1.b_1 + 2^2.b_2 + 2^3.b_3 + 2^4.b_4 + 2^5.b_5 + 2^6.b_6 + 2^7.b_7$. E.g., we can easily verify that the byte 11011001 corresponds to the number 217. Notice that whenever we consider a bit sequence like this, we start counting the bits from right to left. The zeroth bit is the rightmost bit.

As soon as we want to represent numbers bigger than 256, more than one byte is needed. Since 2^{16} equals 65536, we can represent all numbers between 0 and 65535 using 16 bits, i.e. two bytes. All we need to do is divide a number by 256. The outcome of the division is known as the *hi-byte* and the remainder of the division is the *lo-byte*. This idea generalizes to numbers beyond 65536 as well. We just need more bytes. In order to convert a number to a sequence of bytes we have to keep on dividing that number by 256. In every iteration, the remainder of the division (i.e. what we get from `mod`) is a byte because it is a number between 0 and 255. The quotient is the number that needs to be converted by the next iteration. Take for example the number 1234567. If we divide this number we get remainder 135 and quotient 4822. Dividing 4822 by 256 yields remainder 214 and quotient 18. Dividing 18 by 256 yields remainder 18 and quotient 0. In other words $1234567 = 18.256^2 + 214.256^1 + 135.256^0$. Hence, we can represent 1234567 by the bytes 18, 214, 135. This order is called the *big endian representation* since the most significant byte is listed first. Should we prefer the inverse order 135, 214, 18 then we are said to have chosen the *little endian representation*. The chosen representation does not really matter. What matters is that we reconstruct the number from its bytes in the same order as the order used to deconstruct the number into bytes.

[1] This story is most probably not true since early machines had bytes consisting of 7 bits instead of 8.

Notice that we have only talked about positive integer numbers so far. We will explain negative numbers below. Explaining how real numbers (also called *floating point numbers*) are represented as a sequence of bytes is fairly complicated. It falls beyond the scope of this book; we refer the reader to any good book on computer systems or numerical analysis. What is useful to know is that there are two methods. The *single precision* method represents floating point numbers using 4 bytes. The *double precision* method represents floating point numbers using 8 bytes. Both methods were defined by the IEEE[2] (pronounced I-triple-E).

The point of this section is that *any* piece of information, varying from plain numbers, characters, strings, drawings, music, pictures and even Scheme programs are actually represented as long sequences of bytes. The more bytes we have at our disposal, the more information we can represent. Let us consider a final example to show the power of this idea. Fig. 14.1 shows how a very simple operating system (say, one that was written in the early eighties) deals with fonts. The letter "a" is drawn using pixels in an 8 by 8 grid that is subsequently considered as a group of eight bytes. Every row in the grid corresponds to one byte. We invite the reader to carefully verify that this letter "a" therefore corresponds to the following sequence of eight bytes: 60, 66, 2, 30, 98, 66, 70, 59. This means that we can store a font of say 128 characters using $128 \times 8 = 1024 (= 2^{10})$ bytes. This example shows us that we can represent just about any piece of information in a digital computer, provided that we have enough bytes and provided that we can come up with a simple encoding (and inverse decoding) to convert that piece of information to an appropriate sequence of bytes.

The latter number (i.e. 1024 bytes) is also known as *a kilobyte* or $1K$. Similarly, $1024K$ is known as a *megabyte* (or $1M$) and $1024M$ is known as a *gigabyte* (or $1G$). The row is completed by the terms *terabyte* (or $1T$), *petabyte* (or $1P$), *exabyte* (or $1E$), *zettabyte* (or $1Z$) and finally *yottabyte* (or $1Y$). Modern digital computers have memories that can store several gigabytes. Modern disk drives can store several terabytes.

14.1.2 Byte Vectors

Sometimes, it is useful to manipulate *raw* data. By this we mean data that has no internal structure but which merely consists of a sequence of bytes. Let us for example reconsider the drawing that corresponds to letter "a". As explained, a computer might store this drawing by means of 8 bytes in memory. In Scheme, we may have the reflex of using a vector of length 8 to store those bytes. However, an ordinary Scheme vector is far too general for this. Scheme vectors can store any data type and thus have to be general enough to accommodate for all the machinery that is needed for storing different data types (e.g. numbers, strings, vectors, procedures, and so on). But in our example, we know that we want to store 8 consecutive *bytes* in our computer memory and we do

[2]Institute of Electrical and Electronics Engineers.

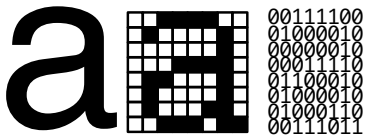

Figure 14.1: Binary representation of the letter "a"

not want the machine to waste memory. This is where Scheme's *byte vectors* come into play. A byte vector is very similar to an ordinary vector of bytes. However, in a byte vector, the Scheme interpreter guarantees that every vector entry is effectively a number between 0 and 255 and that all those bytes are stored consecutively in computer memory. Hence, if we store the drawing of our letter "a" in a byte vector of length 8, we are guaranteed to store the drawing in exactly 8 bytes of computer memory!

Byte vectors are made available in Scheme by importing the library (`rnrs bytevectors`). A byte vector can be created using the `make-bytevector` procedure. E.g., the expression (`make-bytevector` 10 2) creates a byte vector that consists of 10 bytes that are all initialized to the byte 2. `bytevector-u8-ref` and `bytevector-u8-set!` can be used to read a byte from a byte vector and to change a byte at a given index in a byte vector. These procedure are completely analogous to the `vector-ref` and `vector-set!` procedures for ordinary vectors. However, a runtime error is generated should the argument not be a positive number between 0 and 255. As an example, consider our letter "a" again (Fig. 14.1). Its drawing can be represented in memory by the following definitions:

```
(define letter-a
  (let ((v (make-bytevector 8)))
    (bytevector-u8-set! v 0 60)
    (bytevector-u8-set! v 1 66)
    (bytevector-u8-set! v 2 2)
    (bytevector-u8-set! v 3 30)
    (bytevector-u8-set! v 4 98)
    (bytevector-u8-set! v 5 66)
    (bytevector-u8-set! v 6 70)
    (bytevector-u8-set! v 7 59)
    v))
```

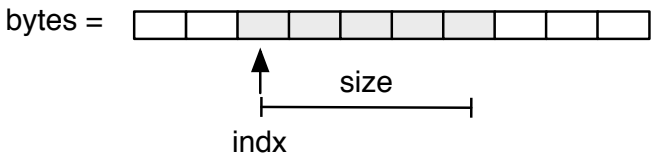

Figure 14.2: Encoding a number in a byte vector

14.1.3 Storing Data in Bytevectors

Although we can use the above two procedures to manually store individual bytes in a byte vector, we will usually want to encode richer Scheme values our byte vectors. This is the raison d'être for Scheme's `bytevectors` library. In what follows, we will use the following operations for storing different kinds of Scheme numbers in a bytevector.

```
(bytevector-sint-set! bytes indx nmbr 'big size)
(bytevector-sint-ref  bytes indx 'big size)
(bytevector-uint-set! bytes indx nmbr 'big size)
(bytevector-uint-ref  bytes indx 'big size)
(bytevector-ieee-double-set! bytes indx nmbr 'big)
(bytevector-ieee-single-set! bytes indx nmbr 'big)
(bytevector-ieee-double-ref  bytes indx 'big)
(bytevector-ieee-single-ref  bytes indx 'big)
```

These procedures all store Scheme numbers in a byte vector `bytes` as illustrated in Fig. 14.2. The `*-set!` procedures take a Scheme number `nmbr` and deconstruct that number into a sequence of bytes (which are indicated in grey). The `*-ref` procedures will read those bytes from the bytevector and reconstruct the original Scheme number. The 'big parameter means that we choose to store the most significant bytes first. This corresponds to the aforementioned "endianness" of the representation.

Let us think about **integer numbers** for a while. An integer number can be positive or negative. Obviously, deep down inside your computer, the minus sign of a negative number somehow will have to be represented by a bit. There are several techniques to do this[3] and we will not go into detail in this book. Whatever the technique used, we have to be aware of it when interpreting a sequence of bytes sitting in a byte vector. For instance, suppose that we read a byte (say, 200) from a byte vector. If we interpret this byte as an unsigned integer (i.e. a natural number), it obviously corresponds to the number 200. However, if we interprete one of the bits as a sign bit, then same byte 200 is suddenly interpreted as the number -56 (again, the exact details of this result are not important in the context of this book).

[3]The most famous one is called the two's-complement representation.

Hence, when reading a sequence of bytes, we need to know whether we are interpreting its bits as a signed number (i.e. one of the bits is used for representing the sign) or as an unsigned number (i.e. all bits contribute to the actual number). In Scheme's `bytevector` procedures, both ways of interpreting the bytes are supported. Procedures that do not interpret one of the bits as a sign bit are prefixed by `u`. These can be used whenever we are absolutely sure that we are storing natural numbers in a byte vector. Procedures that properly treat one bit as sign information are prefixed by `s`. These have to be used when the encoded numbers can be negative or positive.

How can we determine the correct number of bytes needed to store a natural number or an integer? The answer depends on whether we want to store a natural number or an integer number. Here is a procedure to calculate how many bytes are needed to store a **natural number**. It basically determines how often we can divide the number by 256. This is the number of bytes needed to store that number. An exception is needed to cover the values 0 and 1 because their log does not exist or equals 0. In that case we also want 1 to be the result. Hence we take the maximum of the input number and 2.

```
(define (natural-bytes nmbr)
  (exact (ceiling (log (max (+ nmbr 1) 2) 256))))
```

Once we have this number `size`, `bytevector-uint-set!` takes the natural (i.e. unsigned) number `nmbr` and stores its bytes into the given bytevector `bytes`, starting at index `indx`. An error is generated in case the foreseen size does not suffice to store all the bytes of the natural number. `bytevector-uint-get` reads and reconstructs the number from the bytes sitting in the byte vector.

Similarly, `integer-bytes` takes an **integer number** and returns the number of bytes needed to store it in a byte vector. We leave it up to the more technically inspired student to figure out how it works. The others can ignore its implementation. The only thing we need to remember is that `integer-bytes` takes a (signed) integer number and returns the number of bytes that will be needed to represent that number.

```
(define (integer-bytes nmbr)
  (exact (ceiling (log (max (abs (+ (* 2 nmbr) 1)) 2) 256))))
```

Thus, `bytevector-sint-set!` takes an integer (i.e. potentially signed) number `nmbr` of arbitrary size (computed by `integer-bytes`) and stores its bytes into the given bytevector `bytes`, starting at index `indx`. `size` is the number of bytes and this should be sufficient to accommodate for all the bytes that make up the integer number. Again, an error is generated in case the foreseen size does not suffice to store all bytes that constitute the given integer number.

What can we say about **real numbers**? As explained before, using the IEEE standards, a distinction is made between single and double precision. Single precision techniques typically use 4 bytes to represent a real number whereas double precision techniques use 8. This explains the meaning of procedures such as `bytevector-ieee-single-set!` and `bytevector-ieee-double-ref`:

for both precisions, there is a procedure to encode a real number in a bytevector and an equivalent procedure to decode it back from the bytevector.

We finish the section by investigating how **strings** are represented in bytevectors. This used to be really simple in the days when strings were just sequences of characters from the ASCII alphabet. This is because the ASCII alphabet was designed under the assumption that every character corresponds to just one single byte. However, this restricts the number of characters to 256 which is largely insufficient to accommodate for all possible alphabets in the world (think of Russian and Chinese for a moment). That is why computer scientists have come up with extensions of the ASCII alphabet. One such extension is the Unicode standard.

Unicode is a standard in which not all characters are represented by the same number of bytes. This means that extremely frequently occurring characters (such as the English alphabet) are still represented by a single byte. Extremely rare characters require more than one byte, up to 4 bytes per character. The Unicode standard is used by R6RS. The R6RS libraries provide two primitive procedures for converting unicode strings to byte vectors and the other way around. (`string->utf8 str`) converts a given string `str` to a bytevector. (`utf8->string bytes`) does the exact opposite. This means that we will not have to deal ourselves with the way Scheme uses the Unicode standard to represent characters and strings.

In the following chapters, we will sometimes need the greatest string (i.e. the one that always yields #t when compared with the string comparison procedures discussed in section 2.1) that can be created given a number of bytes (between 1 and 4). E.g., (`utf8-sentinel-for 3`) generates the string of 3 bytes that will be greater than any other string that can be represented with 3 or less bytes. This *sentinel string* (similar to $+\infty$ for numbers) is constructed using the following procedure. Its implementation is not relevant for the rest of our study and we leave it up to the more technically inspired student to decipher it after reading about the Unicode standard on the internet.

```
(define (utf8-sentinel-for nmbr-byts)
  (define byts (make-bytevector nmbr-byts))
  (define (fill! offset rem)
    (cond ((= rem 1)
           (bytevector-u8-set! byts offset 127))
          ((= rem 2)
           (bytevector-u8-set! byts offset 223)
           (bytevector-u8-set! byts (+ offset 1) 191))
          (else
           (bytevector-u8-set! byts offset 239)
           (bytevector-u8-set! byts (+ offset 1) 191)
           (bytevector-u8-set! byts (+ offset 2) 191)
           (if (> rem 3)
               (fill! (+ offset 3) (- rem 3))))))
  (fill! 0 nmbr-byts)
```

(`utf8->string byts`))

14.2 The Memory Hierarchy

In the previous section, we have analyzed the composition of some data types down to their most basic constituents. Computers are built to represent bits. Bits are grouped into bytes and all data types are subsequently constructed by combining various bytes. Modern programming languages (like Scheme) do this for us automatically. However, Scheme does provide us with enough low level libraries to manipulate data as sequences of bytes should we need this for some reason. In what follows, this is exactly what we will do.

Bytes are stored in memories. Different types of memory exist. They are constructed using various technologies. Some memories are *volatile*. This means that their content is gone as soon as the electrical power is lost. Other memories are *non-volatile* and allow for persistent data storage, even when the memory is disconnected from its electricity source. Disks, solid state drives, tapes and CD are all examples of non-volatile memory. The reason for this multitude of memory types is that the various kinds of memory exhibit different properties, which all have their advantages and disadvantages.

The different types of memories are are usually taxonomized into categories known as *primary memory*, *secondary memory* and *tertiary memory*. These memories differ in the way they are organized, in their access speed and in their cost per byte. Primary memory is considered to be the "main memory" of a computer. All the algorithms and data structures discussed until now operate on primary memory. Secondary memory and tertiary memory is also known as *peripherical* or *external* memory. These memories reside in devices (such as a hard disk, a tape drive, a CD drive, a memory stick, ...) that are not essential to the basic operation of the computer. This chapter is the start of our study of algorithms and data structures for external memories. First, we continue our journey in the taxonomy of memory for a while.

14.2.1 Primary Memory

The main memory of a computer system is actually just one big sea of bytes. This is also known as the *central* memory or *random access memory (RAM)*. Each byte has a number (called the *address*) and the only basic operations that a computer can undertake consist of storing and retrieving a byte that resides at a certain address. Hence, abstractly spoken, the central memory of a computer system is nothing but a huge byte vector. As explained before, any data type can be mapped onto sequences of bytes. In older programming languages, programmers really had to manipulate the individual bytes of the computer by reading them and storing them in central memory. In modern languages like Scheme, this is done automatically for us by the interpreter of the language.

Roughly spoken, a computer system consists of such a byte vector and a chip that does the actual computational work. The chip reads bytes from the byte vector, manipulates them (e.g. adds them or multiplies them) and stores the resulting bytes back in the byte vector. In order to optimize the speed of this process, the chip has a small amount of on-board memory locations that are extremely fast. These locations are called *registers*. Registers can be thought of as local variables that contain short-lived data during some computation. Thanks to registers, the chip does not need to store every single intermediate result in the computer's byte vector. Unfortunately, registers are very expensive to manufacture. That is why they are limited in number. This means that the computer will usually have to store its register contents into the main memory, which is already much slower. Nevertheless, central memory is extremely fast as well. In modern computers, accessing a byte takes time that is in the order of one hundred nanoseconds[4].

The central memory and the registers[5] together form the *primary* memory of a computer system. It is the fastest memory in the computer system. Unfortunately, primary memory is quite expensive, at least compared to secondary and tertiary memory. This together with some technical limitations result in the fact that modern computers only contain a few gigabytes of memory. Another property of all primary memory is that it is volatile. As soon as the computer's power source is removed, all the data is gone.

14.2.2 Secondary Memory

Secondary storage was invented to overcome this problem. Examples of secondary memory are disk drives, solid state drives, USB-sticks and so on. It is made of different technologies which guarantee that the data is remembered even in the absence of electrical current. This makes secondary memory extremely useful for saving the user's work and for storing large databases that have to survive the execution of an application. Secondary storage is several orders of magnitude slower than primary storage. On the positive side, secondary storage is several orders of magnitude cheaper. Hard disks with a few terabytes can be bought for less than 100 euro nowadays.

Fig. 14.3 shows the design of a hard disk schematically. A hard disk is a device that consists of several rotating *platters* that are covered with a magnetic substance. The principle behind the hard disk is that electricity and magnetism are somehow physically interchangeable quantities. Using an electric current, we can generate a magnetic field. Conversely, using a magnetic field, we can generate an electric current. This exchange happens when the disk's platters rotate in the physical proximity of the *disk heads*. The disk heads consist of a reading head per platter and are connected by a mechanical arm. By moving the arm, the heads are positioned over a different part of the platters which allows it to read and write information as the platters rotate under the heads.

[4] A nanosecond is one billionth of a second, i.e. 10^{-9} sec.

[5] In practice, these two are connected by an intermediate form of memory called the *processor cache*.

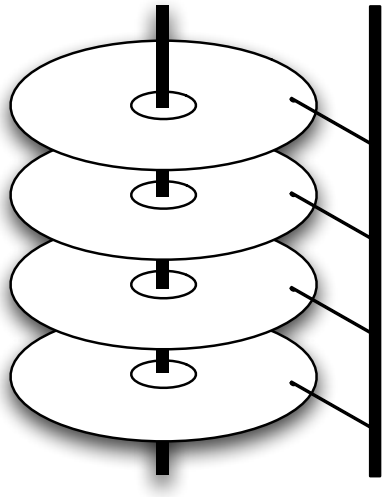

Figure 14.3: Principle of a Hard disk

By sending an electrical current through the head, a magnetic field is created that affects the magnetic substance on the disk. Ultra small magnetic north poles and south poles are created like this. These represent the bits that are written on the disk. Conversely, by keeping the disk heads close to the (rotating) sequence of north poles and south poles on the platters, an electrical current is generated that represents the bits inside the computer system.

Remember that the central memory can be thought of as one giant byte vector in which all entries can be accessed equally fast[6]. This is no longer true for a disk. The disk is organized into concentric circles called *tracks*. Each tracks is given a number. Furthermore, every track is divided into chunks, called *sectors* which are also given numbers. The combination of the track number and the sector number suffice to identify a *block* of data on the disk. Because of technological restrictions, a block is the smallest chunk of data that can be read or written by the disk. **This means that we cannot just read or write a single byte: we have to read or write all bytes that are in the same block**. Although the technology for solid state drives differs substantially from hard disk technology, they also work with blocks as the main unit of data transfer.

If we want to read a block of data, our computer system will have to fetch the bytes of the entire block and store them somewhere in central memory for further processing. This location is known as the *buffer*. Hence, the buffer is a chunk of central memory that has the same size as the size of the disk block.

[6]E.g., a computer with 4G of memory can be thought of as a byte vector of 2^{30} bytes.

Writing a block of data happens by reading the buffer and transferring its bytes from the buffer to the corresponding block on the disk. Conversely, reading a block causes all its bytes to be transferred to the buffer.

In order to read or write a block, the disk has to move the mechanical arm to the right track. Subsequently, it has to wait for the right sector to appear (as a consequence or the rotating disk) under the disk head. As soon as the right sector appears, the head can read or write the bytes of the corresponding block.

The time needed to read or write a block of data can therefore be divided into three phases. First, it takes some time to move the disk arm to the right track position. This time is known as the *seek time*. On modern drives, this takes about 10 milliseconds. After having moved the disk arm, we have to wait for the right sector to appear. The time it takes to do so is known as the *rotational latency*. This also takes about 5 milliseconds. Finally it takes a few milliseconds to read or write the block to or from the buffer. This is known as the *transfer time*. The total time for accessing a block is called the *access time*. It is the sum of the three components just described. Symbolically, we write $T = T_{seek} + T_{latency} + T_{transfer}$. Notice that the numbers just described are very sensitive to variations as technology progresses. However, also notice that we are talking about milliseconds here whereas the time to read and write central memory was expressed in nanoseconds. This means that accessing a disk takes more than 10.000 times as much time as accessing the central memory!

In this and the following chapters, we present a number of algorithms and data structures for secondary storage. Because of the tremendous differences in speed between primary memory and secondary memory, the performance of an algorithm that accesses secondary memory will be entirely dominated by the amount of data that is transferred to and from secondary memory. For some algorithms, the amount of block transfers is so high that the processing time of the algorithm itself becomes completely negligible compared to the time spent transferring data! **It will therefore be our primary goal to keep the number of block accesses as small as possible.** This will be the driving force behind our designs.

14.2.3 Tertiary Memory

A completely different type of memory arises as soon as human or robotic operators are needed to mount disks or tapes into a storage device before they can be read. This is true for so-called tape silos, optical juke-boxes or similar forms of mass storage. Before a byte can be accessed, several seconds may be needed before the correct disk or tape is put into the device. Obviously, these types of storage are extremely slow. They are quite cheap solutions and they allow for petabytes of information to be stored. Examples where such solutions can be useful include the data generated by satellites, data generated by the LHC particle accelerometer (more than 10^6 CDs per year filled with information) or digitalized audiovisual archives (e.g. all TV programs ever broadcasted by the BBC). We will not consider tertiary memory in this book.

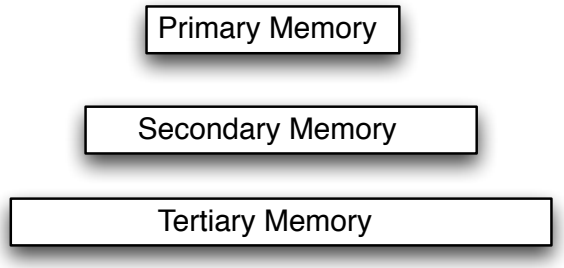

Figure 14.4: The Memory Hierarchy

14.2.4 The Memory Hierarchy

The three types of memory are summarized in figure 14.4. This taxonomy is also known as the *memory hierarchy*.

The memory hierarchy is relevant for designers of algorithms and data structures. The more we go up in the hierarchy, the faster memory becomes. However, the more we go up, the more expensive memory becomes and the more volatile it is. The difference in speed is expressed in orders of magnitude: primary memory is accessed in $O(10^{-9}sec)$ whereas secondary memory requires $O(10^{-3}sec)$ and tertiary memory $O(10^{2}sec)$. The cost per byte is also orders of magnitude different. Primary memory costs about 50 euro per gigabyte (i.e. $O(10^{-7})$ euro per byte) whereas a terabyte hard disk can be bought for 100 euro (i.e. $O(10^{-10})$ euro per byte). Needless to say, these numbers change about every day. Still, the differences remain big enough to be taken into account on the scientific level of algorithms and data structures. They will not get smaller anywhere near soon.

14.2.5 Caches

An important concept from the memory hierarchy which we did not talk about is the notion of a *cache*. A cache is a dedicated piece of fast memory that can help in bridging the difference in speed between a fast memory and a slow memory. The cache acts like a mailbox through which the fast memory and the slow memory can communicate with one another. Like this the fast memory can be fast and the slow memory can be slow at their own pace. Needless to say, this requires a sophisticated protocol to make sure that data that was logically updated in the fast memory is also effectively updated in the slow memory. Otherwise, data would be lost. There exists processor caches and disk caches. The processor cache acts like a fast bridge between the registers and the central memory. The disk cache acts like a bridge between the central memory and the disk. We will discuss the details of a disk cache in section 14.5.

14.3 A Model for Secondary Storage

In the following chapters, we present a number of algorithms and data structures that work on secondary storage. Recall that a byte vector serves as a software model for central memory. In a similar vein, we now present a software model for secondary storage. The abstraction corresponds to the way hard disks really work. Furthermore, the differences with an accurate model for more modern solid state drives are marginal.

14.3.1 The Disk ADT

We start with the `disk` ADT. It is presented below in the familiar format.

```
1  ADT disk
2
3  block−size
4      number
5  disk−size
6      number
7  block−ptr−size
8      number
9  block−idx−size
10     number
11 new
12     ( string → disk )
13 mount
14     ( string → disk )
15 unmount
16     ( disk → ∅ )
17 disk?
18     ( any → boolean )
19 name
20     ( disk → string )
21 read−block
22     ( disk number → block )
```

Some readers may be puzzled by the operation `new`. How can we "make" a new disk given the fact that this is a device made out of real material? The idea is that we will be working with a *virtual* disk, also known as a *disk image*. This is a file on a computer that acts like a normal file (you can copy it, you can throw it away, ...) but which is just a long sequence of bytes that happens to be organized internally like a real disk. For some applications on your computer, opening this file is like mounting a real disk. On Mac OS X for example, there exist `.dmg` (short for disk image) files. You can download these files, copy them, throw them in the trash, etc. However, as soon as you double click such a file, it mounts itself as a disk. From that point on, Mac OS X treats it like any other disk. It is said to be a virtual disk.

`new` creates such a disk image on your computer's drive. `mount` opens a previously created disk image without changing its internal organization. Both take a string argument which serves as the name of the disk. Obviously, the name is also used as the file name of the disk image as it appears on the real disk. `name` returns the name of the disk. `unmount` closes the disk image. Information about the disk image that still resides in Scheme's working memory and that has not been written back to the real disk is then properly written back to that real disk. As such, the representation of the disk image in Scheme's working memory can be safely garbage collected without losing any data.

The most interesting operation of the ADT is probably `read-block`. It takes a disk and a number identifying a certain block on the disk. This number will be called a *block pointer* as it can be used to refer to the block without actually transferring the block into main memory. `read-block` effectively reads the corresponding block into main memory. It does so by reading the correct amount of bytes from the disk and returning them as a `block`. This ADT is discussed in the following section.

Notice that the `disk` ADT also defines 4 constants: `block-size`, `disk-size`, `block-ptr-size`, `block-idx-size` have to be defined in an implementation of the ADT. `block-size` is the number of bytes that fit in one block. As indicated before, for real disks, this often varies from 512 bytes to $4K$, depending on the manufacturer. `disk-size` is the number of blocks that fit on our virtual disk. We leave it as an exercise to calculate the number of blocks found on a $4G$ disk when the block size is 512 bytes.

Suppose that we have a disk whose storage capacity is 512 $(= 2^9)M$ and suppose that its blocks can store 1024 bytes (i.e. $1K$). This means that it will have a total of 524288 blocks whose pointer are numbers that vary between 0 and 524287. In order to store any such pointer, we need 3 bytes. This is the role of `block-ptr-size`. It is the number of bytes needed to store a block pointer. Let us continue with this example. Suppose that we have an application that needs a string the bytes of which are located in byte numbers 468 to 566 in a disk block. The amount of bytes needed to encode these offsets will depend on the size of the block. For blocks that maximally store 256 bytes, we only need 1 byte to encode the offset needed to refer to particular a byte inside a block. For blocks the size of which is 512 bytes (such as in our example) we need two bytes for representing an offset. This is the meaning of the number `block-idx-size`. It is the number of bytes needed to correctly encode an index inside a block.

14.3.2 The Block ADT

The `block` ADT models a block of bytes that lives on a disk. It represents a block of bytes after they have been read into Scheme's working memory using `read-block`. This means that we actually have a *copy* of the actual disk block in our Scheme's working (i.e. in central) memory. As soon as we have a `block` in our Scheme memory, we can ask for its `position` (i.e. its block pointer on the disk) and the `disk` on which it is stored. After modifying the block, we have to write it back to the disk using `write-block!`. This is important to

keep in mind: applying an operation on a block does not automatically apply that operation on the actual block that lives on the disk. The user of the `block` ADT has to manually write modified blocks back to the disk!

```
ADT block

write-block!
    ( block  → ∅ )
block?
    ( any  → boolean )
disk
    ( block  → disk )
position
    ( block  → number )
decode-byte
    ( block number  → byte )
encode-byte!
    ( block number byte  → ∅ )
decode-string
    ( block number number  → string )
encode-string!
    ( block number number string  → ∅ )
decode-fixed-natural
    ( block number number  → natural )
encode-fixed-natural!
    ( block number number natural  → ∅ )
decode-arbitrary-integer
    ( block number  → integer × number )
encode-arbitrary-integer!
    ( block number integer  → number )
decode-real
    ( block number number → real )
encode-real!
    ( block number number real  → number )
decode-bytes
    ( block bytevector number number number  → ∅ )
encode-bytes!
    ( block bytevector number number number  → ∅ )
```

Modifying a block means changing the bytes of the block. This is the role of the other operations, which come in pairs. For every Scheme data type supported by the ADT, we have provided an encoder procedure to convert a value into a sequence of bytes and store those bytes in the block. Conversely, we have a decoder procedure to read a given number of bytes from a block and convert them back to the Scheme data value which they represent.

Every `encode-*` procedure takes a block as its first parameter, an offset (i.e. a byte number) in that block as a second parameter and some additional parameters depending on the particular encoder. E.g., `encode-byte!` just takes the byte that has to be encoded in the block. `encode-string!` takes a string and the number of bytes that it is allowed to occupy in the block (additional characters are discarded). `encode-fixed-natural!` takes a natural number and the number of bytes that have to be used to encode that natural number. `encode-arbitrary-integer!` takes an arbitrarily long Scheme integer number that will be entirely encoded in the block. `encode-real!` takes a real number and the number of bytes that are needed to encode that number (i.e. 4 or 8 depending on whether we are using single precision or double precision).

Every `decode-*` procedure takes a block as its first parameter and an offset in that block as a second parameter. `decode-fixed-natural` will decode a natural (i.e. positive) number given the number of bytes to be considered as a third parameter. Similarly, `decode-string` takes the maximum number of bytes that will be read from the block. The string in the block should be shorter than this number. `decode-real` takes the number of bytes (i.e. 4 or 8) and decodes the bytes by reconstructing the corresponding real number. Notice that `decode-integer!` returns two numbers in a pair. The first number is the decoded integer. The second number is the number of bytes that where occupied by that number.

`encode-bytes!` and `decode-bytes` can be used to copy a raw byte vector into a given block and vice versa. The last parameter is the number of bytes that need to be copied. The third parameter is the offset in the block and the fourth parameter is the offset in the byte vector.

14.3.3 Representation and Implementation

Obviously, both ADTs are fairly intertwined. The `disk` ADT produces blocks by virtue of the `read-block` operation and the `disk` operation defined on blocks returns the disk on which a block is stored. The implementation of both ADTs are therefore best presented simultaneously.

We start by presenting the four constants that belong to the `disk` specification. In our current implementation, we are managing a disk of 200 blocks that can store 50 bytes each. The `natural-bytes` procedure presented in section 14.1.3 is used to calculate the number of bytes that are needed to store a block pointer and an offset referring to a particular byte in a block.

```
(define block-size 50)
(define disk-size  200)
(define block-ptr-size (natural-bytes disk-size))
(define block-idx-size (natural-bytes block-size))
```

Creating, opening and closing a disk is presented below. In order to create a virtual disk (on the real disk of your computer!), we use the output ports of R6RS. `open-file-output-port` opens an output port that is connected to a file. The exact details of this procedure are not really important here. We just

have to know that it opens a new (or existing) file on your hard disk for output. We then create a byte vector `zeroes` of 50 zeroes and we write this byte vector 200 times to the output port. The output port is properly closed and a pair is returned that represents the disk in Scheme's central memory. Like this, we have created a file of 10000 bytes that will represent the virtual disk. "Opening" an existing disk is called *mounting* and "closing" is called *unmounting*. In our current implementation these operations do not really do anything. The cached implementation of the ADT however will use these operations to manipulate the underlying cache properly. This is explained in section 14.5.

```
(define-record-type disk
  (make-disk n)
  disk?
  (n name))

(define (new name)
  (define port (open-file-output-port name
                                      (file-options no-truncate no-fail)
                                      (buffer-mode block)))
  (define zeroes (make-bytevector block-size 0))
  (let low-level-format
    ((block-nr 0))
    (put-bytevector port zeroes)
    (if (< (+ 1 block-nr) disk-size)
        (low-level-format (+ 1 block-nr))))
  (close-port port)
  (make-disk name))

(define (mount name)
  (make-disk name))

(define (unmount dsk)
  ()) ; default is to do nothing ; cached version does more
```

Once we have a disk, we can read its blocks given some block pointer `bptr`. The `read-block` procedure shown below opens the file for input (using R6RS's `open-file-input-port`), sets the file's reading position to the first byte of the block to be read. This is byte number (∗ `bptr block-size`) since block pointers start counting from zero. Then `get-bytevector-n` reads 50 bytes from the disk which are returned from the procedure in the form of a block constructed with `make-block` (which is explained further below). Before returning the newly read block, the input port is properly closed.

```
(define (read-block dsk bptr)
  (define port (open-file-input-port (name dsk) (file-options no-fail)))
  (set-port-position! port (∗ bptr block-size))
  (let ((byts (get-bytevector-n port block-size)))
```

```
    (close-port port)
    (make-block dsk bptr byts)))
```

`write-block!` is the inverse of this operation. It receives a block `blck` and uses the block's block pointer `bptr` and the block's byte vector `data-byts`. This byte vector is written to the disk image after setting the port's position to the right byte number. We use R6RS's `put-bytevector` to accomplish the actual write.

```
(define (write-block! blck)
  (define bptr (position blck))
  (define data-byts (bytes blck))
  (define port (open-file-output-port (name (disk blck))
                                      (file-options no-truncate no-fail)))
  (set-port-position! port (* bptr block-size))
  (put-bytevector port data-byts)
  (close-port port))
```

Here is how a block looks like. It is a simple combination of a disk, a block pointer and a byte vector that was read from that disk. We want to stress once again that this is actually a copy of the block that is stored on the disk. Modifying the bytes in central memory will have no effect, unless we write the block back to the disk. We can think of a block as a data structure with a *dual representation*: the "real" representation lives on the disk and a temporal copy of that representation is stored in Scheme's memory. We have to manual make sure that both versions are kept consistent.

```
(define-record-type block
  (make-block d p b)
  block?
  (d disk)
  (p position)
  (b bytes))
```

The rest of the implementation consists of the encoders and decoders that can be used to manipulate the bytes of a block in a more or less comfortable way.

Bytes can be encoded in a block `blck` at a certain offset `offs`. `encode-byte!` pokes a `byte` in a block at the given offset and `decode-byte` peeks a byte from a block at the offset.

```
(define (encode-byte! blck offs byte)
  (bytevector-u8-set! (bytes blck) offs byte))

(define (decode-byte blck offs)
  (bytevector-u8-ref (bytes blck) offs))
```

Scheme integer numbers of any size are encoded and decoded as follows. In order to encode an arbitrary length integer, we first encode its length (i.e.

the number of bytes it requires) as a byte. Then, we use `bytevector-sint-set!` to encode the integer as a sequence of bytes in the block. The number of bytes occupied by the integer is returned from the procedure. Decoding does the exact opposite. A pair is returned that consists of the decoded number as well as the offset that follows the encoded integer. This is needed by the caller of this procedure in order to know the offset of the next meaningful value sitting in the block.

```
(define (encode-arbitrary-integer! blck offs nmbr)
  (define size (integer-bytes nmbr))
  (encode-byte! blck offs size)
  (bytevector-sint-set! (bytes blck) (+ offs 1) nmbr 'big size)
  (+ size 1))

(define (decode-arbitrary-integer blck offs)
  (define size (decode-byte blck offs))
  (define nmbr (bytevector-sint-ref (bytes blck) (+ offs 1) 'big size))
  (cons nmbr (+ offs size 1)))
```

Natural numbers can be encoded more efficiently. The procedures below take a positive `nmbr` and a chosen number of bytes `size`. The natural number is encoded as an unsigned integer in the block. Decoding does the exact opposite.

```
(define (encode-fixed-natural! blck offs size nmbr)
  (bytevector-uint-set! (bytes blck) offs nmbr 'big size))

(define (decode-fixed-natural blck offs size)
  (bytevector-uint-ref (bytes blck) offs 'big size))
```

Real numbers (also known as floating-point numbers) can be represented using 4 bytes or 8 bytes. The IEEE single precision and double precision standards are used to encode and decode the floating point number accordingly.

```
(define real32 4)
(define real64 8)

(define (encode-real! blck offs size nmbr)
  (cond ((= size real64)
         (bytevector-ieee-double-set! (bytes blck) offs nmbr 'big)
         size)
        ((= size real32)
         (bytevector-ieee-single-set! (bytes blck) offs nmbr 'big)
         size)
        (else
         (error "illegal real size" size))))
```

```
(define (decode-real blck offs size)
  (cond ((= size real64)
         (bytevector-ieee-double-ref (bytes blck) offs 'big))
        ((= size real32)
         (bytevector-ieee-single-ref (bytes blck) offs 'big))
        (else
         (error "illegal real size" size))))
```

Strings are encoded using the procedures presented below. `encode-string!` takes a block, the offset inside the block, the maximal number of bytes that can be used and a string. The string is converted to a byte vector using `string->utf8` which is then copied to the byte vector of the block. The bytes of the string that exceed the previously provided `size` are ignored. Should the length of the string be smaller than `size`, then we use byte 0 to fill up the unused bytes. This is known as *padding*. Decoding a string consists of reading a maximum of `size` bytes but may end earlier as soon as a 0 is encountered.

```
(define str-end-byte 0)

(define (encode-string! blck offs size strg)
  (set! strg (string->utf8 strg))
  (do ((indx 0 (+ indx 1)))
    ((= indx size))
    (let ((byte (if (< indx (bytevector-length strg))
                    (bytevector-u8-ref strg indx)
                    str-end-byte)))
      (encode-byte! blck (+ offs indx) byte))))

(define (decode-string blck offs size)
  (let ((bytevector-u8-set!
         (lambda (bv indx val)
           (bytevector-u8-set! bv indx val)
           bv)))
    (utf8->string
     (let loop
       ((indx 0))
       (if (< indx size)
           (let ((byte (decode-byte blck (+ offs indx))))
             (if (eq? byte str-end-byte)
                 (make-bytevector indx 0)
                 (bytevector-u8-set! (loop (+ indx 1)) indx byte)))
           (make-bytevector indx 0))))))
```

Raw byte vectors are encoded and decoded using simple loops that copy the bytes one by one.

```
(define (decode-bytes blck byts blck-offs u8-offs size)
  (do ((indx 0 (+ indx 1)))
    ((= indx size))
    (bytevector-u8-set! byts
                        (+ u8-offs indx)
                        (decode-byte blck (+ blck-offs indx)))))

(define (encode-bytes! blck byts blck-offs u8-offs size)
  (do ((indx 0 (+ indx 1)))
    ((= indx size))
    (encode-byte! blck
                  (+ blck-offs indx)
                  (bytevector-u8-ref byts (+ u8-offs indx)))))
```

14.4 Files

Now that we have the basic infrastructure to manipulate blocks on a (virtual) disk, we can start using those blocks to store data on our disk. Just as we have used `cons` and `make-vector` as "data glue" to build data structures in central memory, we can use blocks as glue to build data structures on a disk. In order to do so, we will have to link up blocks by storing block pointers in blocks. Like this, we link up blocks to build complicated storage data structures. It is such a collection of linked blocks that is known as *a file*.

When we want to create a new data structure in central memory, it is the job of `cons` (resp. `make-vector`) to find the amount of bytes necessary to store the car and the cdr of a newly created pair (resp. vector). Fortunately, this is done for us by our Scheme interpreter and we do not have to worry about this when writing Scheme programs. Unfortunately, no such mechanism exists on our disk. So how can we know where to find a block that is not occupied (i.e. does not already contain meaningful data) when we need one to create a data structure on disk? Another issue is to know where to find a data structure on the disk. Since the disk is just a flat space of blocks, we must somehow know how to find the first block of each data structure that is stored on the disk. These two questions are taken care of by the *file system*. In what follows, we assume that the `disk` ADT has been imported using `disk:` as a prefix for all its identifiers.

14.4.1 The File System

The file system is a software abstraction that is built on top of the disks. It manages the space of blocks by keeping track of free and occupied blocks. The idea is to give the users of the file system a procedure to ask for fresh unoccupied blocks and as well as a procedure to release an occupied block and thus give it back to the file system. Hence, implementations of data structures can never directly use `read-block` and `write-block!` without going through the

file system. The file system also maintains a list of disk blocks that are the first block of the data structures. This list is called a *directory* of the disk. The directory of a disk pretty much corresponds to the global environment of a Scheme system: it is the place where a reference can be found to all other data structures.

Structure of the File System

In our file system, we will reserve one particular block (called the *meta block*) to store the information used by the file system itself. This means that its block pointer, `meta-bptr`, is never used by any other data structure on the disk. Therefore, we can use that block pointer to represent a "null pointer" on the disk. This is captured by the following declarations. `read-meta-block` and `write-meta-block!` are two procedures that read and write the meta block from and to the disk.

```
(define null-block 0)
(define meta-bptr  null-block)

(define (null-block? bptr)
  (= bptr null-block))

(define (read-meta-block disk)
  (disk:read-block disk meta-bptr))

(define (write-meta-block! meta)
  (disk:write-block! meta))
```

The meta block contains three useful pieces of information. First, it stores the head of the list of free blocks (called the *freelist*). Second, it contains the disk position of the first block of the directory. Third, it stores the amount of blocks that are still available. The following definitions provide us with the abstractions to encode and decode these three components into and from the meta block. Notice that all three fields require `disk:block-ptr-size` bytes to be encoded. The three `*-offset` constants correspond to the location of the three components inside the meta block.

```
(define directory-offset 0)
(define freelist-offset  disk:block-ptr-size)
(define available-offset (* 2 disk:block-ptr-size))

(define (directory meta)
  (disk:decode-fixed-natural  meta directory-offset disk:block-ptr-size))
(define (directory! meta blck)
  (disk:encode-fixed-natural! meta directory-offset disk:block-ptr-size blck))
(define (freelist meta)
  (disk:decode-fixed-natural  meta freelist-offset  disk:block-ptr-size))
(define (freelist! meta flst)
```

```
  (disk:encode-fixed-natural! meta freelist-offset  disk:block-ptr-size flst))
(define (blocks-free meta)
  (disk:decode-fixed-natural  meta available-offset disk:block-ptr-size))
(define (blocks-free! meta free)
  (disk:encode-fixed-natural! meta available-offset disk:block-ptr-size free))
```

Both the freelist and the directory are conceived as a single linked list of disk blocks. In order to link up blocks with each other, we provide the following abstractions. The position of the next block is encoded in the first few bytes of a block. Of course, the exact number of bytes depends on the number of different blocks that we have.

```
(define next-offset 0)

(define (next-bptr blck)
  (disk:decode-fixed-natural blck next-offset disk:block-ptr-size))
(define (next-bptr! blck bptr)
  (disk:encode-fixed-natural! blck next-offset disk:block-ptr-size bptr))
```

Allocating and Deallocating Blocks

The following procedure `format!` shows how the file system is initialized. Its job is to read the meta block, encode the file system information in that meta block and write it back to the disk. The information encoded is the initially "empty" directory (indicated by `null-block`), the head of the freelist (i.e. block number 1) and the number of blocks available (i.e. the size of disk, minus the meta block). Subsequently, the initial freelist is constructed in a `high-level-format` loop which traverses every possible block pointer `bptr` from 1 to `disk-size`. For every such block pointer, we read the corresponding block, we encode the address of the next block and we write the block back to the disk. Hence, initially, the freelist is a list in which every block points to its direct neighbor.

```
(define (format! disk)
  (define meta (read-meta-block disk))
  (directory!  meta null-block)
  (freelist!   meta (+ meta-bptr 1))
  (blocks-free! meta (- disk:disk-size 1)); the metablock itself is not free
  (write-meta-block! meta)
  (let high-level-format
    ((bptr (+ meta-bptr 1)))
    (let ((block (disk:read-block disk bptr)))
      (cond ((< (+ bptr 1) disk:disk-size)
             (next-bptr! block (+ bptr 1))
             (disk:write-block! block)
             (high-level-format (+ bptr 1)))
            (else
             (next-bptr! block null-block)
             (disk:write-block! block))))))
```

```
disk)
```

This regular organization of the freelist will not last very long. The following procedures will be used to construct and remove data structures on our disk.

`new-block` returns a new block by removing it from the freelist. It does this by reading the head of the freelist from the meta block and making the meta block refer to the next block in the freelist. The number of free blocks in the meta block needs to be decremented. The modified meta block is written back to the disk.

```
(define (new-block disk)
  (define meta (read-meta-block disk))
  (define flst (freelist meta))
  (if (null-block? flst)
      (error "disk full! (new-block)" disk)
      (let* ((blck (disk:read-block disk flst)))
        (blocks-free! meta (- (blocks-free meta) 1))
        (freelist!    meta (next-bptr blck))
        (write-meta-block! meta)
        blck)))
```

Conversely, `delete-block` has to be called to give a block back to the file system (for future usage). Again we read the head of the freelist from the meta block and we insert the block in the freelist by making it refer to the head of the freelist and by replacing the freelist by the position of our block. This time, the number of free blocks needs to be incremented. Again, the meta block is written back to the disk.

```
(define (delete-block blck)
  (define disk (disk:disk blck))
  (define meta (read-meta-block disk))
  (next-bptr! blck (freelist meta))
  (disk:write-block!  blck)
  (freelist!    meta (disk:position blck))
  (blocks-free! meta (+ (blocks-free meta) 1))
  (write-meta-block! meta))
```

By having alternating calls of these two procedures, the freelist will become quite an irregular chain of blocks that are dispersed all over the disk. What is important is that both procedures are in $O(1)$. Since all free blocks are of equal size, it does not matter which block is allocated. Therefore, no complicated searching is required to find a free block on the disk.

The following procedure `delete-chain!` can be used to clean up entire lists of blocks. It keeps on reading a block (in order to read its next pointer), deleting the block and calling itself in order to continue deleting the next block.

```
(define (delete-chain! disk bptr)
  (unless (null-block? bptr)
    (let* ((blck (disk:read-block disk bptr))
```

```
      (next (next-bptr blck)))
    (delete-block blck)
    (delete-chain! disk next))))
```

The Directory

The freelist is just a list of blocks that do not contain meaningful information for anyone. A directory is also conceived as a list of blocks (that starts in the meta block). Every such block contains a collection of *slots* that store one string and one block pointer. The string is a file name and the block pointer corresponds to the pointer of the first block of the corresponding file. Since the length of the file names is limited, we will have to cap names that are too long (i.e. longer than the allowed `filename-size`). Empty slots in the directory are indicated by the $+\infty$ string which can be determined with the `utf8-sentinel-for` procedure shown in section 14.1.3.

```
(define filename-size     10)
(define sentinel-filename (utf8-sentinel-for filename-size))

(define (cap-name name)
  (if (> (string-length name) filename-size)
      (substring name 0 filename-size)
      name))
```

The following abstractions allow us to see a block as a directory block.

```
(define slot-size        (+ filename-size disk:block-pointer-size))
(define nr-of-dir-slots (div (- disk:block-size disk:block-pointer-size)
                             slot-size))

(define (dir-name/bptr! blck slot name bptr)
  (define offn (+ disk:block-ptr-size (* slot slot-size)) )
  (define offp (+ disk:block-ptr-size (* slot slot-size) filename-size))
  (disk:encode-string! blck offn filename-size name)
  (disk:encode-fixed-natural! blck offp disk:block-ptr-size bptr))
(define (dir-name blck slot)
  (define offn (+ disk:block-ptr-size (* slot slot-size)))
  (disk:decode-string blck offn filename-size))
(define (dir-bptr blck slot)
  (define offp (+ disk:block-ptr-size (* slot slot-size) filename-size))
  (disk:decode-fixed-natural blck offp disk:block-ptr-size))
```

Three procedures are implemented to store a file name and its corresponding block pointer in a given slot number (`dir-name/bptr!`), to read the file name of a given slot (`dir-name`) and to read the block pointer belonging to that file name (`dir-bptr`). The offset used to store a slot is easily computed by multiplying the number of the slot with the size of a slot. The size of a slot is the number of bytes needed to store a file name and the corresponding block

pointer. Notice that all these procedures have to add `disk:block-ptr-size` to the offsets in order not to overwrite the next pointer information that was previously explained.

In what follows, we will be building data structures on our disk. Every time we create a new data structure, we will register it in the directory under a certain file name. Likewise, every time we remove a data structure from the disk, we have to remove its file name from the directory of the disk. This means that we have to think about the way slots are occupied and released. We initialize all slots in a directory block as empty by means of a sentinel (indicated by a `sentinel-filename`). Whenever we have to add a slot to the directory, we search for the first slot containing that sentinel value. We extend the directory with a fresh block should this be necessary. We also make the directory smart enough to recuperate a disk block as soon as all its slots store the sentinel. Here are some abstractions that will render the code more readable.

```
(define (has-next? blck)
  (not (null-block? (next-bptr blck))))

(define (empty-slot? blck slot)
  (string=? sentinel-filename (dir-name blck slot)))

(define (at-end? blck slot)
  (= slot nr-of-dir-slots))
```

The following auxiliary procedure is used to create a freshly initialized directory block and link it up to some other block by calling the parameter procedure `next!`. `fresh-block!` asks the file system for a new block and initializes the new block by setting its next pointer to `null-block` and by initializing all its slots to the sentinel file name and the null block pointer. Finally, the fresh block is linked up to its predecessor by calling `next!` which is a procedure that has to be provided by the caller of `fresh-block!`.

```
(define (fresh-block! disk next!)
  (define next (new-block disk))
  (next-bptr! next null-block)
  (do ((slot 0 (+ slot 1)))
    ((at-end? next slot)
     (next! next)
     next)
    (dir-name/bptr! next slot sentinel-filename null-block)))
```

File System Operations: Some Examples

Now that we have implemented the representation of the file system on the disk, we can put it to work. Several useful operations can be implemented that give us access to the file system. They are sometimes hidden from an end user behind a graphical user interface (such as e.g. Apple's Lion), and sometimes made available in the form of *shell commands* (e.g. on top of unix or DOS). It

is not our intention to implement a complete operating system here. We merely show the inner workings of a few key operations. The names of our operations follow the unix tradition and are therefore slightly cryptic.

Our first operation is `mk` ("make"). `mk` takes a disk, a file name and the block pointer `bptr` that corresponds to the very first block of some data structure (i.e. a file) on the disk. `mk` is called to register the file under the given file name with in the file system's directory.

```
(define (mk disk name bptr)
  (define meta (read-meta-block disk))
  (let loop-dir
    ((dptr (directory meta))
     (new! (lambda (newb)
             (let ((meta (read-meta-block disk)))
               (directory! meta (disk:position newb))
               (write-meta-block! meta)))))
    (let ((blck (if (null-block? dptr)
                    (fresh-block! disk new!)
                    (disk:read-block disk dptr))))
      (let loop-block
        ((slot 0))
        (cond ((at-end? blck slot)
               (loop-dir (next-bptr blck)
                         (lambda (newb)
                           (next-bptr! blck (disk:position newb))
                           (disk:write-block! blck))))
              ((empty-slot? blck slot)
               (dir-name/bptr! blck slot name bptr)
               (disk:write-block! blck))
              (else
               (loop-block (+ slot 1))))))))
```

`mk` consists of an outer loop that governs a "current" disk pointer `dptr` and a procedure `new!` that will be used to link a possibly created fresh block `newb` to the rest of the directory. Initially, this is a procedure that will link `newb` to the meta block. However, as soon as we have one iteration of `loop-dir`, the procedure is replaced by another procedure that will link `newb` to the previous block. `loop-dir` hops from block to block searching for a block that contains at least one unoccupied slot. The search within each block is taken care of by `loop-block`. It investigates every single `slot`. As soon as an empty slot is found, it is occupied by the newly added information. When all slots are investigated, `loop-dir` is called again in order to proceed with the next block. If we bump into a null block pointer, we call `fresh-block!` in order to create a fresh block and link it up to the previous block using `new!`.

The inverse operation of `mk` is called `rm` which is short for "remove". It looks as follows:

```
(define (rm disk name)
```

```
(define meta (read-meta-block disk))
(set! name (cap-name name))
(let loop-dir
  ((bptr (directory meta))
   (nxt! (lambda (next)
           (directory! meta next)
           (write-meta-block! meta))))
  (let ((blck (if (null-block? bptr)
                  (error "file not found (rm)" name)
                  (disk:read-block disk bptr))))
    (let loop-block
      ((slot 0)
       (seen #f))
      (cond ((at-end? blck slot)
             (loop-dir (next-bptr blck) (lambda (next)
                                          (next-bptr! blck next)
                                          (disk:write-block! blck))))
            ((empty-slot? blck slot)
             (loop-block (+ slot 1) seen))
            ((string=? name (dir-name blck slot))
             (dir-name/bptr! blck slot sentinel-filename null-block)
             (disk:write-block! blck)
             (if (not seen)
                 (maybe-delete-block! blck slot nxt!)))
            (else
             (loop-block (+ slot 1) #t)))))))
```

Again, a system of nested loops is used. This time, we are searching the directory entry to be deleted. The outer loop is used to hop from block to block in the directory list. In every step, a procedure `nxt!` is constructed that can be used to link the previous block of the current block to the successor of the current block. This happens when the current block becomes "empty" if all its slots are removed.

In the code, the inner loop investigates all the slots one by one. If all slots are considered, the outer loop is called again with the next block and a procedure to possibly link its next block to the current block. Empty slots are skipped. If a slot is non-empty and its name corresponds to the file name of the file to be removed, we fill the slot with the sentinel value. The `seen` variable is transported throughout the inner loop. It is initially #f and becomes #t as soon as the inner loop has seen at least one *non*-sentinel entry. If this has not happened after erasing the slot, then the removal of a slot may mean that we have to cleanup the block in which the slot resides. However, in order to be really sure, we need to investigate the rest of the block as well in order to see whether or not it contains any meaningful slot. This is the job of `maybe-delete-block!`. Its implementation is as follows.

```
(define (maybe-delete-block! blck slot next!)
```

```
(cond
  ((at-end? blck slot)      ; block appears to be completely empty
   (next! (next-bptr blck)) ; hence, delete it
   (delete-block blck))
  ((empty-slot? blck slot)  ; continue checking
   (maybe-delete-block! blck (+ slot 1) next!))))
```

The procedure keeps on going as long as it encounters empty slots. The iteration stops if a slot is non-empty. In that case we do not do anything since the block has to survive. However, if the iteration encounters the end of the block, we have investigated the entire block without having encountered any meaningful file name. As a consequence, we should delete the block. We do so by linking the previous block of that block to the next block of that block (by calling `next!`) and by releasing the block itself by calling `delete-block`.

One of the most important operations provided by our file system is `whereis`. This operation takes a file name and returns its associated block pointer. `whereis` searches the directory sequentially. A loop `traverse-dir` hops from block to block and checks each and every slot inside every block. As soon as the file name stored in the slot corresponds to the name we are searching, the associated pointer is returned by calling (`dir-bptr blck slot`).

```
(define (whereis disk name)
  (define meta (read-meta-block disk))
  (define bptr (directory meta))
  (set!  name (cap-name name))
  (if (null-block? bptr)
      0
      (let traverse-dir
        ((blck (disk:read-block disk bptr))
         (slot 0))
        (cond ((at-end? blck slot)
               (if (has-next? blck)
                   (traverse-dir (disk:read-block disk (next-bptr blck)) 0)
                   null-block))
              ((string=? name (dir-name blck slot))
               (dir-bptr blck slot))
              (else
               (traverse-dir blck (+ slot 1)))))))
```

More operations can be implemented. The `df` ("disk free") procedure retrieves the number of free blocks from the super block. It can be used by applications for knowing how much disk space is still available (counted in blocks). Implementing it is a simple programming exercise because the number of free blocks is readily available from the super block. `ls` ("list segments") is used in unix-like systems to list the contents of a directory. The implementation is straightforward again: we traverse the entire directory, block by block. Empty slots are skipped. Meaningful slots are recursively accumulated in a list.

Summary

This section has presented a minimalistic file system on top of the previously presented `disk` abstraction. This exercise has a double goal. On the one hand, it has shown how blocks can be linked up in order to create data structures. The first such data structure is the directory which was conceived as a linked list of blocks. Again, the file system presented is extremely simplistic. In real file systems, we can create nested directories (a.k.a. folders), we can put access rights on files, etc.

The exercise is an illustration of the fact that we will have to rethink most of the data structures studied until now. E.g., removing a slot from the list of slots is far more complicated than in central memory. We have to keep in mind that successive elements of our list are encoded in different blocks on the disk. Removing an element can be achieved by allowing some slots to be "empty". This is the strategy we have chosen in our implementation. Another option would be to perform a storage move in order to fill up holes emerging from deleting a slot. This would result in an implementation that is even worse than the storage move discussed in section 3.1.1 because the storage move requires us to read and write the consecutive blocks of the directory. If the directory has N entries with block size B, then a worst-case of $O(N/B)$ blocks have to be transferred.

14.4.2 Sequential Files

We can think of a file as an intelligently organized sequence (or cluster) of disk blocks. This can be accomplished by using some of a block's bytes to serve as pointers to "next" blocks. Like this, any type of file structure such as a linked lists of blocks and trees of blocks can be built. In what follows, we will see several examples. In this section we start with the most common among all file types, namely *sequential files*.

A sequential file is an external data structure that supports writing and reading data values one after the other. We will implement two kinds of sequential files. Sequential *output files* are sequential files that support a `write!` operation. Internally they maintain a "current" that perpetually refers to the location immediately following the last data value written. Writing a new data value causes that value to be added after that current. Sequential *input files* support the operations `peek` and `read`. Input files also maintain a "current" that points to the last data value read. Reading a data value using `read` causes the current to advance whereas reading it using `peek` does not. Hence, one can peek the same value multiple times.

The ADTs for output files and input files are presented below.

```
1 ADT output−file
2
3 new
4     ( disk string → output−file )
5 sequential−file?
```

```
    ( any → boolean )
name
    ( output−file → string )
open−write!
    ( disk string → output−file )
reread!
    ( output−file → input−file )
write!
    ( output−file any → ∅ )
close−write!
    ( output−file → ∅ )
delete!
    ( output−file → ∅ )
```

`new` allows an output file to be created on a disk given a file name. It will foresee the necessary bookkeeping blocks and it will also register the file name in the directory of the disk. `delete!` does the exact opposite: the output file and its bookkeeping information are removed from the disk. All the file's blocks are recovered by the file system and the file name is deleted from the directory. As explained, `write!` is used to append data values to the file. After writing data values, `close-write!` has to make sure the file is properly closed. This means that the file's bookkeeping information is properly updated on the disk. Finally, `open-write!` opens an existing file. It will position its current at the beginning of the existing file such that all subsequent invocations of `write!` will reuse the existing blocks by overwriting the information that already sits in those blocks. In case less is written than already present in that file, `close-write!` will then chop off the file and reclaim the blocks that have not been overwritten. Hence, `open-write!` allows us to overwrite a sequential file without having to delete it first. This is much more efficient since deleting a file corresponds to adding all its blocks to the freelist, which requires reading and writing every single block.

Input files are the companion of output files. Notice that we cannot create an input file from scratch since the file would be empty (and thus there would be nothing to read). Therefore, input files lack a `new` operation. They can only come into existence by calling `open-read!` given a disk and a file name. This opens a previously written file and prepare it for reading.

```
ADT input−file

sequential−file?
    ( any → boolean )
name
    ( input−file → string )
open−read!
    ( disk string → input−file )
rewrite!
    ( input−file → output−file )
```

```
read
    ( input−file → any )
peek
    ( input−file → any )
has−more?
    ( input−file → boolean )
close−read!
    ( input−file → ∅ )
delete!
    ( input−file → ∅ )
```

As expected, `delete!` is used to remove an input file from the disk and erase its file name from the directory. `close-read!` properly closes an input file after reading. This allows the file to update its bookkeeping information on the disk should this be necessary. `read` and `peek` are the workhorses of this ADT: as long as the predicate `has-more?` yields #t, these procedures can be called to read data from the file.

Notice the operations `reread!` in the first ADT and `rewrite!` in the latter. `reread!` allows one to close an output file and start using it as an input file. `rewrite!` does the exact opposite. These operations will be heavily relied upon by our external sorting algorithms presented in chapter 15. Functionally, `rewrite!` is exactly the same as closing the file with `close-read!` and subsequently calling `open-write!` to open it again. Similarly, `reread!` is just a combination of `close-write!` followed by an immediate `open-read!`. Hence these operations are not really essential to the ADT. Nevertheless, adding the operations to the ADT will turn out to be slightly more convenient in chapter 15 and will allow us to implement them saving a few block transfers.

Example

Before we present the implementation of the ADTs, let us first show how to use them by means of an example. Here is a small test program that creates a new output file and which writes three values (a real number, a natural number and a string) to the file. After closing the file, the file is reopened as an input file and `read` is called three times. Obviously, this will print the three values on the screen in the same order as the order in which they were written to the output file. Notice that we first have to create a formatted disk in order for our file system to work correctly.

```
(define d (disk:new "My Computer"))
(fs:format! d)

(define f (out:new d "TestFile"))

(out:write! f 3.14)
(out:write! f 42)
(out:write! f "Done!")
```

```
(out:close-write! f)

(set! f (in:open-read! d "TestFile"))
(display (in:read f))(newline)
(display (in:read f))(newline)
(display (in:read f))(newline)

(in:close-read! f)
```

Representation

Let us now implement these ADTs on top of our `disk` abstraction and using the file system procedures of the previous section. We assume that the file system library is imported with prefix `fs:`. We start with the representation. Obviously, the file itself is entirely represented on the disk. However, part of the file is also represented in central memory. Otherwise, every single read and write operation would cause a block transfer and we want to avoid as many block transfers as possible. Therefore, the internal representation of the file keeps one "current" block in central memory. That block is referred to as the *buffer* of the file. Moreover, we also maintain a copy of a block that is known as the *header* of the file. This is a block that stores some bookkeeping information of the file. It is this header that will contain a reference to the very first block of the actual file.

The internal representation of a file consists of a record value that keeps a reference to the file's disk, the file name, a copy of the header block and the current buffer. This explains the following representation.

```
(define-record-type sequential-file
  (make d n h b)
  sequential-file?
  (d disk)
  (n name)
  (h header header!)
  (b buffer buffer!))
```

Instead of manipulating the header's bytes directly, we use the following abstraction layer. A header stores the block pointer of the very first data block at position `first-offset`. When writing to a file, we have to maintain a "current offset" in order to know where (i.e. the offset in a block) to start writing the next data element. This current offset is also encoded in the header. In the future, we may add extra information to the header such as the file size (i.e. the number of disk blocks it occupies), graphical information to store the file's icon in a windowing system and so on. Here are the procedures that map the logical header operations onto naked byte manipulations.

```
(define frst-offs 0)
(define curr-offs disk:block-ptr-size)
```

```
(define (first hder)
  (disk:decode-fixed-natural  hder frst-offs disk:block-ptr-size))
(define (first! hder bptr)
  (disk:encode-fixed-natural! hder frst-offs disk:block-ptr-size bptr))
(define (current hder)
  (disk:decode-fixed-natural  hder curr-offs disk:block-idx-size))
(define (current! hder offs)
  (disk:encode-fixed-natural! hder curr-offs disk:block-idx-size offs))
```

Using this information, we can already implement the `delete!` operation of both ADTs. Here it is:

```
(define (delete! file)
  (define fnam (name file))
  (define hder (header file))
  (define fdsk (disk file))
  (fs:delete-chain! fdsk (first hder))
  (fs:delete-block hder)
  (fs:rm fdsk fnam))
```

All we need to do is grab the disk pointer that corresponds to the first disk block. We use the `delete-chain!` operation of the file system to clean up an entire linked list of blocks. Finally we give back the header block to the file system (by calling `delete-block`) and we remove the file name from the directory structure using `fs:rm`.

Opening and Closing Output Files

Let us now have a look at the procedures that implement the `output-file` ADT. Creating a new file or opening an existing file for writing are extremely similar. In the first case, we have to create a new header block and a very first (empty) buffer block using `new-block`. The header's block number is registered in the directory by means of `mk`. Inside the header, we make the reference to the block pointer of the first buffer block. In the second case, we locate the header by looking it up in the directory. We then use `read-block` to read the header and the first buffer block from the disk. In both cases, we initialize the current offset to `block-ptr-size` since we have to start writing in the byte that follows the bytes that contain the next pointer of the block.

```
(define (new disk name)
  (define hder (fs:new-block disk))
  (define bffr (fs:new-block disk))
  (define file (make disk name hder bffr))
  (fs:mk disk name (disk:position hder))
  (first!  hder (disk:position bffr))
  (fs:next-bptr! bffr fs:null-block)
  (current! hder disk:block-ptr-size)
  (disk:write-block! hder)
```

```
  file)

(define (open-write! disk name)
  (define bptr (fs:whereis disk name))
  (define hder (disk:read-block disk bptr))
  (define fptr (first hder))
  (define bffr (disk:read-block disk fptr))
  (define file (make disk name hder bffr))
  (current! hder disk:block-ptr-size)
  file)
```

Remember that there is a third way of ending up with an output file, namely by calling `rewrite!` on an input file. Here is how the code looks like. It closes the input file using `close-read!` and reinitializes the buffer block to the first block of the file. Again, the current offset is set to `block-ptr-size` in order to skip the next pointer of the block.

```
(define (rewrite! file)
  (define fdsk (disk file))
  (define hder (header file))
  (define fptr (first hder))
  (close-read! file)
  (buffer! file (disk:read-block fdsk fptr))
  (current! hder disk:block-ptr-size)
  file)
```

The entire idea of `open-write!` and `rewrite!` is to create an output file as an "overlay" on some existing sequential file. Hence, every time we write something to the file, we will reuse the existing blocks to store the data (instead of asking the file manager for new blocks all the time). When closing such a file with `close-write!`, the unused blocks of the original file have to be cleaned up. This is accomplished by the implementation of `close-write!` which takes the `rest-list` from the buffer and cleans up all its subsequent blocks using `delete-chain!`. `close-write!` also updates the header to the disk and properly terminates the buffer block by encoding a final `eof-tag`. This is needed since the buffer block is not necessarily full when closing the file. The unused bytes of the block have no meaning and `eof-tag` (defined further below) will preclude `read` from accidentally reading these meaningless bytes from the last block of a file.

```
(define (block-bytes-free file)
  (define hder (header file))
  (define curr (current hder))
  (- disk:block-size curr))

(define (close-write! file)
  (define hder (header file))
  (define bffr (buffer file))
```

```
(define fdsk (disk file))
(disk:write-block! hder)
(if (> (block-bytes-free file) 0)
    (disk:encode-byte! bffr (current hder) eof-tag))
(let ((rest-list (fs:next-bptr bffr)))
  (fs:next-bptr! bffr fs:null-block)
  (fs:delete-chain! fdsk rest-list))
(disk:write-block! bffr))
```

Writing Data to Output Files

Looking back at the sequential file ADTs, we observe that the `write!` and `read` operations do not specify the data type of the value to be written or read. *Any* Scheme value can be written and read. Hence, whenever the file reads a number of bytes, it will have to be smart enough to know the meaning of those bytes. A different number of bytes has to be read when we are reading a real number compared to reading an integer. Therefore, `write!` always first writes a 1-byte tag which indicates *the type* of the bytes that follow. Similarly, `read` always first reads a type tag after which it will be able to read the correct amount of bytes in order to read the data value. In our current implementation, we discern 6 different tags, 4 of which are type tags for identifying natural numbers, integer numbers, real numbers (called decimals for historical reasons) and string values. Other data types are not supported by our current implementation but this is just a matter of hard work: all we need to do is add more tags and extend the writing and reading procedures presented below. The `eof-tag` is used to encode the end of a (partially filled) block and the `eob-tag` will be explained further below.

```
(define natural-tag 0)
(define integer-tag 1)
(define decimal-tag 2)
(define string-tag  3)
(define eof-tag 255)
(define eob-tag 254)
```

In what follows, we implement `write!`. `write!` first encodes the data type of the object to be written (in one byte) and subsequently puts the bytes representing the actual data value on the file. Hence, after writing the tag, the code further dispatches to helper procedures (such as `write-natural`) that do the actual encoding of `sval`.

```
(define (write! file sval)
  (cond ((natural? sval)
         (write-type-tag file natural-tag)
         (write-natural file (exact sval))) ; (natural 2.0) = #t
        ((integer? sval)
         (write-type-tag file integer-tag)
         (write-integer  file (exact sval))) ; (integer? 2.0) = #t
```

```
((real? sval)
 (write-type-tag file decimal-tag)
 (write-real    file sval))
((string? sval)
 (write-type-tag file string-tag)
 (write-string  file sval))
(error "unsupported type (write!)" sval)))
```

Writing the tag is very simple: we merely encode the tag as a byte (using `encode-byte!`) and we advance the current by 1. `write-type-tag` already illustrates a central problem that is faced by all writing procedures: we need enough space in the buffer to encode the object. Instead of polluting all procedures with this code, a general procedure `claim-bytes!` is called with the number of bytes that will be necessary. Obviously, only one byte is necessary to write the tag. That is why the following procedure claims 1 byte before doing the actual write of the tag.

```
(define (write-type-tag file ttag)
  (claim-bytes! file 1)
  (let* ((bffr (buffer file))
         (hder (header file))
         (curr (current hder)))
    (disk:encode-byte! bffr curr ttag)
    (current! hder (+ curr 1))))
```

Claiming bytes is shown below. We verify whether there is still room left in the current block. If this is not the case, we properly terminate the block and we make sure that a next block is provided that will have enough room. Properly terminating a block is done by writing the `eob-tag` (i.e. end of block). It means that some of the very last bytes of the block do not contain useful information, but that the file continues in the next block.

```
(define (claim-bytes! file nmbr)
  (define bffr (buffer file))
  (define hder (header file))
  (define curr (current hder))
  (when (< (block-bytes-free file) nmbr)
    (if (not (= curr disk:block-size))
        (disk:encode-byte! bffr curr eob-tag))
    (provide-next-block! file)))
```

`provide-next-block!` is the engine that extends the file, either by asking the file system for a new block or by following next pointers in existing blocks. The latter happens when we are overwriting an existing file. After having found the `next` block, it is made the current buffer (after writing the current buffer back to the disk) and the current offset is set to point to the very first meaningful byte of the block (by skipping the size of the next-pointer of the block).

```
(define (provide-next-block! file)
```

```
(define fdsk (disk file))
(define hder (header file))
(define bffr (buffer file))
(define next (cond ((fs:null-block? (fs:next-bptr bffr))
                    (let ((newb (fs:new-block fdsk)))
                      (fs:next-bptr! newb fs:null-block)
                      (fs:next-bptr! bffr (disk:position newb))
                      (disk:write-block! bffr)
                      newb))
                   (else
                    (disk:write-block! bffr)
                    (disk:read-block fdsk (fs:next-bptr bffr)))))
(buffer!  file next)
(current! hder disk:block-ptr-size))
```

Given this low level machinery, the data writing procedures become extremely simple. The ones for numbers are shown below. We have provided procedures for (unsigned) naturals, (signed) integers and for real numbers. Of course, nothing precludes us from implementing more procedures for encoding the multitude of numeric types in Scheme's numeric tower.

```
(define (write-natural file nmbr)
 (claim-bytes! file (+ (disk:natural-bytes nmbr) 1)) ; size byte
 (let* ((hder (header file))
        (curr (current hder))
        (bffr (buffer file)))
   (disk:encode-byte! bffr curr (disk:natural-bytes nmbr))
   (disk:encode-fixed-natural! bffr (+ curr 1) (disk:natural-bytes nmbr) nmbr)
   (current! hder (+ curr 1 (disk:natural-bytes nmbr)))))

(define (write-integer file nmbr)
  (claim-bytes! file (+ (disk:integer-bytes nmbr) 1))
  (let* ((hder (header file))
         (curr (current hder))
         (bffr (buffer file)))
    (current! hder (+ curr (disk:encode-arbitrary-integer! bffr curr nmbr)))))

(define (write-real file nmbr)
  (claim-bytes! file disk:real64)
  (let* ((hder (header file))
         (curr (current hder))
         (bffr (buffer file)))
    (current! hder (+ curr (disk:encode-real! bffr curr disk:real64 nmbr)))))
```

All three procedures have exactly the same structure: enough bytes are claimed for representing the number and then the correct `encode` procedure of the `disk` ADT is invoked to actually encode the number in the current buffer.

Remember that the encoder for arbitrary sized integers returns the number of bytes used. This result is used here in order to increment the current accordingly.

Encoding strings in a sequential output file is a bit more complicated. This is because strings can span multiple blocks. Strings are encoded by first writing a 1 byte number to indicate the length of a string. This means that our current implementation excludes strings of more than 255 bytes but this is easily adapted. After claiming a byte for encoding the string length, the string is converted to a byte vector using `string->utf8` which is subsequently rolled out in the current buffer block. Should this block be too small to contain the entire string, then the string is chopped into chunks that stored in consecutive blocks. This is the role of `rollout-bytes`. This procedure keeps on calling itself recursively thereby claiming bytes each time and thus causing blocks to be provided.

```
(define (write-string file strg)
  (claim-bytes! file 1)
  (let* ((hder (header file))
         (curr (current hder))
         (bffr (buffer file))
         (byts (string->utf8 strg)))
    (disk:encode-byte! bffr curr (bytevector-length byts))
    (current! hder (+ curr 1))
    (rollout-bytes file byts 0)))

(define (rollout-bytes file byts indx)
  (when (< indx (bytevector-length byts))
    (claim-bytes! file 1)
    (let* ((hder (header file))
           (bffr (buffer file))
           (curr (current hder))
           (free (block-bytes-free file)))
      (cond ((< (- (bytevector-length byts) indx) free)
             (disk:encode-bytes! bffr byts curr indx (- (bytevector-length byts) indx) )
             (current! hder (+ curr (- (bytevector-length byts) indx))))
            (else
             (disk:encode-bytes! bffr byts curr indx free)
             (current! hder disk:block-size)
             (rollout-bytes file byts (+ indx free)))))))
```

Opening and Closing Input Files

Now that we know how output files encode their data values, we can implement the inverse operations for input files. We start by the machinery needed to open and close input files. An input file is opened either by calling `open-read!` in the hope that a file corresponding to the name is found on disk, or by converting an existing output file to an input file using `reread!`. The latter corresponds to "rewinding" an output file in order to start reading its contents. In the current

representation, closing an input file does not require anything to be written to the disk. In future implementations (e.g. implementations that store the current after partially reading the file) this may change.

```
(define (open-read! disk name)
  (define bptr (fs:whereis disk name))
  (define hder (disk:read-block disk bptr))
  (define file (make disk name hder))
  (define fptr (first hder))
  (buffer! file (disk:read-block disk fptr))
  (current-offset! hder disk:block-pointer-size)
  file)

(define (reread! file)
  (define dsk (disk file))
  (define hder (header file))
  (define fptr (first hder))
  (close-write! file)
  (buffer! file (disk:read-block dsk fptr))
  (current-offset! hder disk:block-pointer-size)
  file)

(define (close-read! file)
  ())
```

The basic functionality of `open-read!` and `reread!` is the same. The latter closes the output file and reinitializes the buffer with the first block of the file. The former looks up the file name in the directory structure (using `whereis`), reads the file's header and uses this header in order to access the first actual block of the file. In both cases, the current offset is initialized by refering to the very first meaningful byte in the buffer, i.e. by skipping the next pointer of the block.

Reading Data from Output Files

Having opened an input file, we can use the operations `read`, `peek` and `has-more?`. The implementation for the latter looks as follows. A file has meaningful bytes left provided that its buffer is not yet entirely consumed (implemented in `more-on-buffer?`) or provided that its buffer has a next block with more data on it.

```
(define (more-on-buffer? file)
  (define bffr (buffer file))
  (define hder (header file))
  (define offs (current hder))
  (and (< offs disk:block-size)
       (not (or (= (disk:decode-byte bffr offs) eob-tag)
                (= (disk:decode-byte bffr offs) eof-tag)))))
```

```
(define (has-more? file)
  (define bffr (buffer file))
  (or (more-on-buffer? file)
      (not (fs:null-block? (fs:next-bptr bffr)))))
```

Reading and peeking from the file is explained below. The structure of both procedures is very similar. The only difference is that `peek` restores the original situation (i.e. the buffer and the current offset) whereas `read` does not. All detailed peeking procedures (such as `peek-real`, `peek-integer`, `peek-natural` and `peek-string`) actually return a pair with two results. The car of the pair is the actual return value. The cdr of the pair is the amount of bytes that were already read from the block after reading the car (or the last part of the car if it is a string). Notice that the last used block is not necessarily the same as the current buffer block because peeking a value may have caused a new buffer block to be read. `read` returns the car of the result and installs the returned offset as the current position in the buffer.

```
(define (read file)
  (define hder (header file))
  (define curr (current hder))
  (let* ((ttag (read-type-tag file))
         (vcur (cond ((= ttag natural-tag)
                      (peek-natural file))
                     ((= ttag integer-tag)
                      (peek-integer file))
                     ((= ttag decimal-tag)
                      (peek-real file))
                     ((= ttag string-tag)
                      (peek-string file))
                     ((= ttag eof-tag)
                      (cons () curr))
                     (else
                      (error "unsupported type on file (read)" ttag)))))
    (current! hder (cdr vcur))
    (car vcur)))
```

`peek` on the other hand returns the car of the result but restores the original buffer and its current offset.

```
(define (peek file)
  (define hder (header file))
  (let* ((bffr (buffer file))
         (curr-offs (current hder))
         (ttag (read-type-tag file))
         (res (car (cond ((= ttag natural-tag)
                          (peek-natural file))
                         ((= ttag integer-tag)
                          (peek-integer file))
```

```
                    ((= ttag decimal-tag)
                     (peek-real file))
                    ((= ttag string-tag)
                     (peek-string file))
                    ((= ttag eof-tag)
                     (cons () curr-offs))
                    (else
                     (error "unsupported type on file (peek)" ttag))))))
  (current! hder curr-offs)
  (buffer!  file bffr)
  res))
```

Both procedures first read a 1-byte tag from the file. After having read the tag, they have enough information about the type of the value to be read from the file. This information is used to dispatch the work by calling one of the more dedicated peeking procedures (such as `peek-real` and `peek-integer`). The tag is read as follows:

```
(define (read-type-tag file)
  (supply-bytes! file)
  (let* ((hder (header file))
         (bffr (buffer file))
         (curr (current hder))
         (ttag (disk:decode-byte bffr curr)))
    (current! hder (+ curr 1))
    ttag))
```

Just like we had the couple `claim-bytes!` and `provide-next-block!`, we have a couple `supply-bytes!` and `read-next-block!` to generate more blocks should this be necessary.

```
(define (supply-bytes! file)
  (define bffr (buffer file))
  (define hder (header file))
  (define curr (current hder))
  (if (not (more-on-buffer? file))
    (read-next-block! file)))
```

The idea is that all peeking procedures make a claim for new bytes to be supplied. `supply-bytes!` requires the next buffer block if either the current exceeds the block size or if the current refers to an `eob-tag`. This means that the current buffer no longer holds meaningful information and that the next buffer block has to be read. Reading the next buffer block is the job of `read-next-block!`. It reads the next pointer from the current buffer block and loads the corresponding block from the disk. Again, the current offset is reinitialized to the very first meaningful byte of the block.

```
(define (read-next-block! file)
  (define fdsk (disk file))
```

```
(define hder (header file))
(define bffr (buffer file))
(define next-bptr (fs:next-bptr bffr))
(define next-blck (disk:read-block fdsk next-bptr))
(buffer! file next-blck)
(current! hder disk:block-ptr-size))
```

The procedures for reading the actual data from the disk are pretty straightforward. We merely present `peek-real` for the sake of completeness. It decodes a real number and returns that number as well as the updated offset (which is the current offset plus 8).

```
(define (peek-real file)
  (supply-bytes! file)
  (let* ((hder (header file))
         (bffr (buffer file))
         (curr (current hder)))
    (cons (disk:decode-real bffr curr disk:real64) (+ curr disk:real64))))
```

Summary

In this section, we have presented two ADTs (`output-file` and `input-file`) for representing sequential files on a disk. Sequential files can be considered the analogue of Scheme lists on a disk. Sequential files are easy to implement and — as we will see — form one of the most important external data structures. Needless to say, sequential files are not very well suited for storing information that has to be accessed in a non-sequential way (e.g. for implementing dictionaries on disk). Searching a sequential file of n data elements causes an average of $O(n/2.B)$ blocks of size B to be transferred, which is bad news. Therefore, we will study richer data structures (e.g. $B^+ - Trees$) that generate far fewer block transfers. Nevertheless, sequential files remain very useful for storing information on a disk (that does not require fast indexing) and for sorting information on a disk. This will be the topic of chapter 15.

14.5 Caching

Remember from section 14.2 that there is a difference of about 6 orders of magnitude between the time needed for accessing a byte on the disk and accessing a byte in plain good old Scheme memory. As a consequence, it will be our perpetual goal to keep the number of block transfers as small as possible. This can be done by designing our algorithms and data structures in a clever way. This is pretty much what we will do in the forthcoming chapters. However, even in the face of such smart algorithms, we can still optimize the number of block transfers by using a *cache*.

The idea of a cache is to reserve part of the central memory to store the most heavily used blocks. In Scheme, this can be done with a plain vector storing

such blocks. The big advantage of this is that the invocations of `read-block` for those heavily used blocks then become really cheap. Instead of reading the blocks from the disk, we merely have to read them from our cache. Suppose that an algorithm executes a `read-block` operation that is implemented according to this scheme. If we are lucky and the cache effectively contains the requested block, then we speak of a *cache-hit*. If the block is not present in the cache, we speak of a *cache-miss*. In this case, we have to read the block from the disk and store it in the cache before handing it over to the algorithm.

A number of problems immediately show up:

- Obviously, after a certain number of `read-block` invocations, the cache will be completely full. What should be our strategy when this happens?

- When an algorithm uses one of the encoders (see section 14.3.3) to modify a block that resides in the cache, the cache has to guarantee that those changes are written to the disk eventually.

As a solution to the first problem, the idea is to identify a block in the cache that is most suitable to be removed from the cache in order to free space for the newly read block. In technical terms, it is said that we *evict* a block from the cache. Clearly, evicting a block means that modifications made to the block (by one of the encoders) first need to be written back to the disk. There are several strategies to choose a block for eviction:

First-in, first-out (FIFO) In this strategy, the cache is organized like a queue. The block that resides longest in the cache is evicted first. At first sight, this seems to be a reasonable choice. However, it is not always the optimal choice: a block may be the oldest block but at the same time one of the most frequently used blocks. If this is the case, it is better to keep the block in the cache and evict a younger block that is used less frequently. It can be shown that the FIFO strategy performs poorly in practice: it is even possible that the number of cache-misses grows as the cache gets larger. This is known as *Belady's anomaly*, named after the hungarian computer scientist Laszlo Belady.

Least-Frequently Used (LFU) Another strategy is to evict the block that is the least frequently used. By "used" we mean that the block has been subject to an invocation of one of the encoders or decoders of data. In order to implement this strategy, the encoders and decoders will have to maintain a counter for all cached blocks and increment that counter upon every usage of a block. Evicting a block then means that we have to search the cache for the block with the smallest counter.

Least-Recently Used (LRU) Instead of registering the number of times that a block is used, we can also register the *time* at which the block was used. In Scheme, this can be accomplished by calling the procedure

(`current-time`) that can be imported from the SRFI-19[7]. Each and every time the contents of a block is used, its time stamp is updated.

Random In this strategy, we randomly select a block from the cache.

Clearly, the randomized strategy is the easiest to implement in $O(1)$ for it requires no search process in the cache. The FIFO strategy can also be realized in $O(1)$ since all we need is a simple vectorial queue (see section 4.2). The LFU and LRU strategies are clearly better from a qualitative point of view. However, they will require a search process which will take $O(n)$ computational steps given a cache of size n. But remember that this is a small price to pay if it helps us to reduce the number of block transfers.

The fact that the search may need to investigate all locations of the cache stems from the fact that every cache location has the same probability for receiving a block. Such caching architectures are called *fully associative caches*. Other strategies are possible. E.g., support that we decide that blocks with an even block number always end up in the first half of the cache and that blocks with an odd block number end up in the second half of the cache. With a strategy like this (others are possible as well), we halve the amount of cache locations to be traversed. The drawback is that some locations of the cache may be underused: whenever we have to read a block whose predestined locations are all taken, then we have to proceed to evict one of the blocks even if other locations in the cache are still free. Hence, there is a trade-off between optimal usage of all the cache's locations on the one hand and a fast lookup algorithm for finding a block's whereabouts on the other hand.

Let us now have a look at the second problem. If we avoid the `write-block!` operations as much as possible for improving the speed, then how can we make sure that the information on the disk is consistent with what a running algorithm is expecting? There are two main strategies. In the **write-through** strategy, this happens immediately: every time the application calls the `write-block!` operation, the block is effectively written to the disk. Of course this does not mean that the block is removed from the cache. The block stays in the cache and serves as an efficient solution for all subsequent applications of `read-block` as long as the block is not evicted. In the **write-back** strategy, the block is not written to the disk when executing `write-block!`. Instead, the block is "released" by the application which means that the cache system is allowed to evict the block from the cache. However as long as this does not happen, the block stays in the cache. As soon as the block is read again by an invocation of `read-block`, it is "locked" again. Hence, alternating calls to `read-block` and `write-block!` toggle the block's status between "locked" and "released". While locked, algorithms are allowed to modify the block using one of the encoders. When released, the block is a potential candidate for eviction. When this happens, the block has to be written back to the disk (hence the name of this strategy). A "dirty-bit" decides whether or not this writing effectively

[7]SRFI or Scheme Request For Implementation is a set of non-standardized libraries that is implemented and supported by an internet community.

takes place. The dirty bit is set to 1 as soon as one of the encoders modifies the block.

14.5.1 Implementation of the Cache

The cache itself is conceived as a simple vector. `cache:new` creates the vector and `cache:get` and `cache:put!` are just synonyms for the corresponding operations on vectors.

```
(define cache-size 10)

(define (cache:new)
  (make-vector cache-size ()))

(define (cache:get cche indx)
  (vector-ref cche indx))

(define (cache:put! cche indx blck)
  (vector-set! cche indx blck))
```

Here is the implementation of `cache:find-free-index`. This procedure will be called by `read-block` when the requested block is not in the cache. In that case, the block will be read from the disk after a cache slot is found in which the block will be stored. The procedure below traverses the entire cache by calling (`traverse 0`). It does two things at the same time. If a free slot is found, the index of that slot is returned from `traverse`. However, if `traverse` reaches the end of the cache, it returns the value of `oldest-indx` provided that this value is positive. The variable `oldest-indx` is initialized to -1 and is updated as soon as a block is found that is currently not locked. The oldest of all such blocks is remembered in `oldest-index`.

Hence, `traverse` either returns the index of the first free slot or the index of the oldest unlocked block should no free slot exist.

```
(define (cache:find-free-index cche)
  (define oldest-time (current-time))
  (define oldest-indx -1)
  (define (traverse indx)
    (if (< indx cache-size)
      (let
          ((blck (cache:get cche indx)))
        (if (null? blck)
          indx
          (let ((lckd (locked? blck)))
            (if (not lckd)
              (let
                  ((stmp (time-stamp blck)))
                (when (time<? stmp oldest-time)
                  (set! oldest-time stmp)
```

```
                (set! oldest-indx indx))))
            (traverse (+ indx 1)))))
      (if (negative? oldest-indx)
        (error "cache full" cche)
        oldest-indx)))
  (traverse 0))
```

Whenever `read-block` is called to read a block from the disk, it will first consult the cache in order to check whether or not the block already resides in the cache. This is the job of `find-block`. Given a block pointer `bptr`, it traverses the cache in order to see whether it contains the corresponding block. This block is returned from the procedure if it exists.

```
(define (cache:find-block cche bptr)
  (define (position-matches? blck)
    (and (not (null? blck))
         (= (position blck) bptr)))
  (let traverse
    ((indx 0)
     (blck (cache:get cche 0)))
    (cond ((position-matches? blck)
           blck)
          ((< (+ indx 1) cache-size)
           (traverse (+ indx 1)
                     (cache:get cche (+ indx 1))))
          (else
           ()))))
```

14.5.2 Reimplementing the disk ADT

We are now ready to present a cached implementation of the `disk` ADT presented in section 14.3.1. The cached implementation of the ADT relies on our previous implementation of section 14.3.3. We assume that this implementation has been imported using the prefix `disk:`. A cached disk is simply represented as a pair consisting of a cache (i.e. a vector) and a non-cached disk:

```
(define-record-type cdisk
  (make-cdisk v d)
  disk?
  (v disk-cache)
  (d real-disk))

(define (new name)
  (make-cdisk (cache:new) (disk:new name)))

(define (mount name)
  (make-cdisk (cache:new) (disk:new name)))
```

```
(define (name cdsk)
  (disk:name (real-disk cdsk)))
```

In our original implementation presented in section 14.3.3, unmounting a disk was an empty operation since blocks where at all times properly written back to the disk. This is no longer the case for a cached disk. The cache of the disk can still contain blocks whose dirty bit is set (i.e. that have been modified by one of the encoders) but which have not yet been written back to the disk. It is our duty to consider all such blocks and write them back to the disk before unmounting the disk. Otherwise data would be lost. In the code below, `traverse` considers every cache slot and writes all dirty blocks to the disk.

```
(define (unmount cdsk)
  (define vctr (disk-cache cdsk))
  (define (traverse indx)
    (if (< indx cache-size)
      (let ((blck (cache:get vctr indx)))
        (cond
          ((not (null? blck))
           (if (dirty? blck)
             (disk:write-block! (block blck)))
           (invalidate! blck)))
        (traverse (+ indx 1)))))
  (traverse 0)
  (disk:unmount (real-disk cdsk)))
```

This code illustrates another property of cached blocks. Remember that disk blocks are actually Scheme data structures that reside in our Scheme's central memory. During the execution of our Scheme algorithms, different parts of our Scheme system can get a reference to a block. Surely, when evicting a block from the cache, we need a way to notify all parts of our algorithms that use the block that the block is no longer valid (unless it is read back from the disk first). This is managed by an extra bit in every block: as soon as a block is evicted from main memory, it is *invalidated*. All operations (both encoders and decoders) applied to an invalidated block will generate a runtime error.

14.5.3 Reimplementing the block-ADT

We are now ready to study the representation of cached disk blocks. Again, the representation relies on the ordinary disk blocks presented in section 14.3.3. A cached disk block consists of a (non-cached) disk block that is enhanced with a time stamp, a dirty bit and a bit indicating the aforementioned locked/released status of the block. The Scheme code for the representation is straightforward:

```
(define-record-type block
  (make d l t i b)
  block?
```

```
  (d dirty? dirty-set!)
  (l locked? locked!)
  (t time-stamp time-stamp!)
  (i disk)
  (b block block!))

(define (make-block cdsk blck)
  (make #f #t (current-time) cdsk blck))

(define (dirty! blck)
  (dirty-set! blck #t))
```

Invalidating a block is achieved by setting the underlying disk block to (). Like this, the block is logically removed from central memory and all subsequent calls of the `block` ADT operations (such as encoders and decoders) will henceforth generate an error.

```
(define (invalidate! blck)
  (block! blck ()))

(define (valid? blck)
  (not (null? (block blck))))
```

Now we have to present a reimplementation of the `block` ADT's operations. We start with `position`: the block pointer of a cached block is just the block pointer of the underlying disk block.

```
(define (position blck)
  (if (not (valid? blck))
    (error "invalidated cblock(position)" blck))
  (disk:position (block blck)))
```

In a similar vein, we have to implement a new version of all encoders and decoders. All these operations can be considered as "lifted" versions of the original operations. They need to check whether the block is still valid and call the original operation if this is indeed the case. Moreover, they have to update the time stamp of the cached block. Instead of manually writing this code for all decoders, we implement a higher order procedure `make-cached-decoder` that transforms the original decoders by lifting them in a new lambda that performs these actions and then calls the original decoder.

```
(define (make-cached-decoder proc)
  (lambda args
    (define blck (car args))
    (if (not (valid? blck))
      (error "invalidated cblock(cached version of decoder)" blck))
    (time-stamp! blck (current-time))
    (apply proc (cons (block blck)  (cdr args)))))

(define decode-byte (make-cached-decoder disk:decode-byte))
```

```
(define decode-fixed-natural (make-cached-decoder disk:decode-fixed-natural))
(define decode-arbitrary-integer (make-cached-decoder disk:decode-arbitrary-integer))
(define decode-real (make-cached-decoder disk:decode-real))
(define decode-string (make-cached-decoder disk:decode-string))
(define decode-bytes (make-cached-decoder disk:decode-bytes))
```

Lifting the encoders is entirely similar but also requires the lifted operation to call `dirty!` in order to set the modified block's dirty bit. This is because we are modifying the block.

```
(define (make-cached-encoder proc)
  (lambda args
    (define blck (car args))
    (if (not (valid? blck))
      (error "invalidated cblock(cached version of encoder)" blck))
    (time-stamp! blck (current-time))
    (dirty! blck)
    (apply proc (cons (block blck)  (cdr args)))))

(define encode-byte! (make-cached-encoder disk:encode-byte!))
(define encode-fixed-natural! (make-cached-encoder disk:encode-fixed-natural!))
(define encode-arbitrary-integer! (make-cached-encoder disk:encode-arbitrary-integer!))
(define encode-real! (make-cached-encoder disk:encode-real!))
(define encode-string! (make-cached-encoder disk:encode-string!))
(define encode-bytes! (make-cached-encoder disk:encode-bytes!))
```

14.5.4 Reading and writing blocks

We now have all the building blocks to implement a cached version of `read-block` and `write-block!`. `write-block!` does not do anything but logically releasing the block by calling `locked!` with #f. This means that the block stays in the cache but that it becomes a potential candidate for eviction should space be needed.

```
(define (write-block! blck)
  (define cche (disk-cache (disk blck)))
  (if (not (valid? blck))
    (error "invalidated block(write-block!)" blck))
  (locked! blck #f)) ; no write, that's the whole point!
```

`read-block` first calls `cache:find-block` in order to check whether or not the block is in the cache. If this is the case, the block's status is set to locked again and the operation finishes. However, if the block does not appear to reside in the case, then we call `find-free-index` in order to find a cache slot in which we can store the newly created cached block. If this slot happens to contain a block (this is the `when`-test shown below), then this block is evicted. The evicted block is invalidated after it has been properly written back to the disk.

```
(define (read-block cdsk bptr)
```

```
(define cche (disk-cache cdsk))
(define blck (cache:find-block cche bptr))
(if (null? blck)
  (let*
      ((indx (cache:find-free-index cche))
       (blck (cache:get cche indx)))
    (when (not (null? blck))
      (if (dirty? blck)
        (disk:write-block! (block blck)))
      (invalidate! blck))
    (set! blck (make-block cdsk (disk:read-block (real-disk cdsk) bptr)))
    (cache:put! cche indx blck)))
(locked! blck #t)
blck)
```

14.5.5 Locality Properties of Algorithms

What is caching going to buy us in terms of efficiency? How many block transfers are really going to be avoided thanks to the presence of the cache? This is a question that cannot be answered unless we have an idea of the algorithm that is invoking all those `read-block`s and `write-block!`s that are intercepted by our caching machinery.

The basis for caching to work properly is so-called **locality** of algorithms. Algorithms can exhibit two kinds of locality:

Temporal locality is a property of an algorithm which states that the most recently used data will probably be used again in the near future. E.g., a stack has excellent temporal locality while a queue has very poor temporal locality.

Spatial locality means that the algorithm has a high likelihood of using data values that are stored in memory locations that are close to the most recently used data values. Bubble sort is an algorithm that has excellent spatial locality since it only considers adjacent locations in a vector. Quicksort has two localities, one for each index that is used in the partitioning phase. Radix sort has very poor locality properties: the `spread` operation operates on very distinct locations in the vector depending on the value of the digit currently under investigation.

These concepts immediately explain why the algorithms studied so far cannot be trivially applied on top of a disk architecture. Consider e.g. sorting a huge amount of data that is stored on some disk. In order to apply Quicksort to a disk filled with data, one might e.g. naively consider the disk as a huge vector (by mapping indexes in `vector-ref` and `vector-set!` onto the correct block numbers). However, given the fact that Quicksort exhibits poor caching behavior (because it has poor locality properties) this solution will engender many useless block transfers. As we will see in the next chapter, merge sort is about

the only advanced sorting algorithm that has acceptable caching behavior. As a consequence, all external sorting algorithms of chapter 15 will be variations on merge sort.

14.6 Exercises

1. What is the number of blocks needed on a $4G$ disk when the block size is 512 bytes?

2. Writing a block with unchanged contents back to the disk is a useless and costly operation. We can optimize our `disk`/`block` ADTs by avoiding such useless writes. Concretely, we change the `write-block!` operation so that it only actually writes the block to the disk if it was modified by one of the encoder operations. To know whether a block was modified we can use a "dirty" flag. Add such a flag to the `block` ADT implementation. Make sure the flag is set whenever the block is modified by means of an encoder. Finally, modify `write-block!` so that only dirty blocks are effectively written to the disk.

3. By looking at the `encode-fixed-natural!` and `decode-fixed-natural` operations we observe that the true complexity of encoding natural numbers into bytes is hidden in `bytevector-uint-set!` and `bytevector-uint-ref` functions. To better understand what is going on we will reimplement both operations without relying on functions provided by R6RS.

   Remember that the largest natural number we can represent in a single byte is 255 ($= 2^8 - 1 = 256 - 1$). For 2 bytes this is 65535 ($= 2^{16} - 1 = 256^2 - 1$), ..., and for n bytes this is $2^{n \times 8} - 1 = 256^n - 1$.. To encode a natural number as a series of bytes we therefore we need to iteratively divide the number by 256 and encode the remainder of the division (`mod val 256`) into a single byte (using the `encode-byte!` operation), while passing the result of the division (`div val 256`) to the next iteration.

   (a) Reimplement `encode-fixed-natural!` using this algorithm. You can choose to use either the big endian or the little endian byte order (or both). Test your implementation using the original `decode-fixed-natural` operation. If it decodes the same number you had encoded this means your implementation of the encoder is correct. Be sure to try numbers that require more than one byte.

   (b) In `encode-fixed-natural!` the number of bytes to use is passed as the `size` parameter. In the original implementation an error was thrown (by `bytevector-uint-set!`) whenever the number being encoded was too big to fit in the number of bytes specified. Can you add this check to your own implementation?

(c) Reimplement `decode-fixed-natural`, using the same byte order as the one chosen for your implementation of `encode-fixed-natural!`. Test your code.

4. The `encode-fixed-natural!` operation only allows us to store natural numbers ($\in \mathbb{N}$). For storing integers ($\in \mathbb{Z}$) we can use `encode-arbitrary-integer!`. This operation reserves a whole byte to store the size of the number being stored. This relieves the programmer from having to specify the number of bytes to be used, but of course this makes the operation less efficient in terms of space. It would thus be practical if we also had a `encode-fixed-integer!` operation (and a corresponding `decode-fixed-integer` operation) which lets the programmer specify the number of bytes to be used, such as in `encode-fixed-natural!` and `decode-fixed-natural`.

   To do this we will need a binary representation for both negative and positive numbers. There are different ways to do this and we consider two options.

   Sign-and-magnitude representation This is the most naive way to represent signed numbers. As the name suggests this representation simply allocates one bit to store the sign (0 = positive and 1 = negative) and uses the remaining bits to store the magnitude (the absolute value). The bit which is used to store the sign is the first bit of the first byte (the most significant bit). The magnitude is stored as an unsigned number in the remaining 7 bits of the first byte and any bytes that may follow.

   With this representation we can use one byte to store values in the interval $[-127, 127]$. Notice however that there are two ways to store the value 0 (i.e. we have $+0$ and -0) which clearly does not make any mathematical sense.

   Two's complement representation In this representation positive values are stored as we would store unsigned numbers. For negative values we first add 256^n, where n is the number of bytes used, and then store them as we would store unsigned numbers. In both cases we use all the available bits. However, by adding the 256^n to negative numbers we are setting the most significant bit to 1. Consequently this bit will indicate the sign again (0 = positive and 1 = negative).

   With this representation we can use one byte to store values form the $[-128, 127]$ interval. Notice that there is only one way to store 0. The two's complement representation is clearly better and consequently this representation is being used in more or less all of today's software and hardware.

   (a) Do you understand why the sign-and-magnitude representation is such a bad choice? There are at least two reasons.

(b) Choose one of both representations and implement the new `encode-fixed-integer!` and `decode-fixed-integer` operations accordingly. You can choose to use either the little or big endian byte order. Test to see if your implementation can correctly encode/decode all values in the interval and be sure to try numbers that require more than one byte.

5. On most computer systems the user can specify a personal name to disks or USB memories. This name is stored as part of the file system. We will add this feature to our filesystem by reserving a field in the meta block (at a predefined offset `label-offset` and of a predefined length `label-size`) to store the label as a string. Implement a `label!` procedure that sets the label and a `label` procedure that gets it. Also adapt the `format!` procedure so that it takes the label as an argument and stores the label on the newly formatted disk. Additionally adapt the `df` procedure so that it returns both the label and the remaining free blocks.

6. The `input-file` and `output-file` ADTs allow us to copy sequential files in a very straightforward fashion: we read Scheme values (numbers, strings, etc.) one at the time from an existing `input-file` and subsequently write them directly to a new `output-file`.

   (a) Implement this "high-level" `copy` operation for sequential files by relying on the existing `input-file` and `output-file` ADTs.

   (b) Implement a "low-level" version of `copy` which directly operates on the level of blocks and bytes instead of Scheme values.

7. In the header block of our sequential files we keep a pointer to the first "content" block of the file and an offset (`current`) that points to the next byte for reading or writing in the buffer. In our output file ADT we can only write data at the end of a (thereby constantly growing) file. This means that `current` is always pointing to a position in the last block of the file. However, in our implementation it is not possible to use this information to *extend* an output file once it has been closed: the `open-write!` operation reopens an existing file but overwrites any existing contents. Therefore we will add a `open-extend-write!` operation which reopens a file in order to extend it, starting at the last block of the file. In principle we could find the last block of the file by following all next-pointers starting from the first block. But then we would have to read every block of the file one-by-one from the disk into central memory which is tremendously inefficient. Therefore we will add a "last" pointer to the header of the sequential file representation, in order to enable us to always find the last block of a file in $O(1)$. Implementation steps:

   (a) Add `last!` and `last` operations which respectively set and get the `last-block` pointer field in the header (at a predefined offset `last-offset` and of the correct size);

(b) adapt the `output-file` ADT such that the last pointer is correctly initialised and changed as the file grows;

(c) implement the `open-extend-write!` operation which opens an existing output file and positions and prepares the buffer for writing at the end of the file (using the last block pointer and the current offset).

8. In real world file systems much more information about files is stored. We will now add some of the most basic examples of such meta data: the size of the file (approximated by the number of blocks), the time at which it was created and the time of the last modification. Again we will store this information by adding new fields to the header block of our sequential files: `block-count`, `created`, `modified`. As always the fields start at a specific offset and have a specific length. Also add (and export) an `info` operation to the sequential file ADTs which returns a list containing the file name, the block count and the creation and modification times of a file.

9. To find out how many disk access operations we can avoid by using a cached disk we can add some bookkeeping to the implementation. For any particular disk we want to keep track of the total number of read and write operations that have been requested by applications and the portion of those which resulted in an effective read or write operation on the underlying disk. Implementation steps:

   (a) Extend the cached `disk` representation such that it can hold the following counters: `cache:requested-reads`, `cache:effective-reads`, `cache:requested-writes`, `cache:effective-writes`. Upon creation of a cached disk all counters should be set to zero.

   (b) Ensure that the counters are updated at the correct places in the implementation of the cached disk.

   (c) Add and export a `cache-info` operation which returns all counters (grouped in a list)

   (d) Test you solution.

10. Our implementation of the cached disk follows a *fully associative* caching architecture. However there are alternative strategies that can be used. A straightforward example is given in section 14.5: we could split the cache in two and use one half for blocks when an even block number and the other half for blocks with an uneven block number. The goal of this exercise is the implement (and test) a generalised version of this strategy in which the cache is split up in n parts. This requires only a surprisingly small number of changes.

11. Our implementation of the cached disk follow a "least recently used" (LRU) eviction strategy. This means that the block which has not been

used for the longest amount of time is evicted from the cache (provided that it is not locked). An alternative strategy is to evict the "least frequently used" (LFU) block. Implement this variant.

12. Finding a suitable spot to store a new block in the cache (and thus select which block must be evicted) is handled by the `cache:find-free-index` procedure. Since the cache is an unsorted vector, this procedure must traverse the cache linearly, resulting an a $O(n)$ performance characteristic. One way to improve upon this is to keep the the vector sorted. Implement it.

13. Discuss the locality properties of heapsort.

Chapter 15

External Sorting

In chapter 5 we have extensively studied a number of widely known sorting algorithms. A common characteristic of all these algorithms is that they are *internal* sorting algorithms. They assume that all the data can be kept in central memory (more specifically in a list or vector) and that the entire sorting process takes place in the central memory of our computer. This assumption does not always hold. Whenever the data to be sorted is too large to fit in the few gigabytes of central memory that we have in our computer, we have to resort to so-called *external* sorting algorithms; i.e. sorting algorithms that (partially) operate on disk.

Most of the repertoire of sorting techniques that we have studied is completely useless when applied on a disk. As we know, it is impossible to read individual data elements from a disk without reading all other data elements that reside in the same disk block. An algorithm that has to operate on a disk should therefore better exploit the fact that the entire block is in memory before writing it back to the disk or allowing it to be garbage collected by Scheme. In chapter 14, such algorithms have been said to exhibit good *caching behaviour*.

In chapter 5 we presented three advanced sorting algorithms: Quicksort, heapsort and merge sort. As we have seen in chapter 14, the caching behavior of an algorithm depends entirely on its locality properties. Quicksort has two localities in its partitionning phase and each such phase only places one single element in its right position. Heapsort heavily depends on the sifting operations which continuously poke around in the vector at positions i, $2i$ and $2i+1$. This obviously results in poor locality properties. Merge sort is the only algorithm that has good locality properties: every time we read a value from a file, we either have to read a fresh buffer block, or we continue consuming all other values from the current buffer block. That is why most external sorting algorithms are modifications of merge sort.

15.1 Introduction by Example

We will explain the basic principle of external merge sorts by means of an example. In what follows, we assume that we have a disk that contains a huge sequential file storing data values, (as implemented in the previous chapter). We also assume that the amount of data sitting on that file largely exceeds the storage capacity of our central memory several times. Concretely, we could think of trying to sort a 54 gigabyte file using a computer with only 2 gigabytes of internal memory. These numbers will be used to explain the general idea behind all external merge sorts:

The distribution phase reads the original file and constructs a number of auxiliary files. This is achieved by continuously filling a buffer in central memory, sorting that buffer (e.g. using quicksort) and writing the sorted buffer back to one of the auxiliary files that can be merged together. As such, the data of the input file becomes distributed over several auxiliary files. This basic idea is depicted schematically in figure 15.1. Notice that this figure suggests that all data on the original file can be distributed on p (here: 3) *sorted* files which are then merged into one file again. It assumes that each and everyone of the p files was small enough to be sorted in central memory. But, how do we proceed when the amount of data is even larger? In that case, the p files will consist of multiple sorted "chunks". In other words, the auxiliary files will consist of several (logically separated) sequences of values that are internally sorted, but which are further unrelated with one another. Such sorted sequences are called *runs*. Hence, the general idea of the distribution phase consists of distributing the data over several auxiliary files (which may live on different disks) which each consist of multiple runs. This is shown on the left hand side of figure 15.2. The figure shows how a file of 54 gigabytes is read into central memory in 27 turns. Each time, a 2 gigabyte chunk is read into central memory, sorted and subsequently written to one of the 3 auxiliary files. After the distribution phase, every auxiliary file consists of 18 gigabytes of data that is internally organised as 9 consecutive runs of 2 gigabytes of sorted data.

There are two ways in which a sequential file can consist of runs. In a **file with counted runs**, we basically just have a sequential file about which we *know* that it consists of consecutive runs of a given length. One could e.g. consider the sequence 11, 50, 65, 8, 12, 76, 7, 23, 33, 45, 53, 98 as a file (whose length is 12) with counted runs of length 3. Indeed, we can verify that every sequence of length 3 in this sequence effectively consists of numbers that appear in sorted order. In a file with counted runs, we do not need to write a special "end of run marker" since all the runs are of the same (fixed) length. This is no longer the case in a **file with varying runs**. In our most advanced sorting algorithm (i.e. p-polyphase sort), the runs sitting in a file will no longer have a fixed length. Instead, we will use a special sentinel value that is added after every run in order to mark

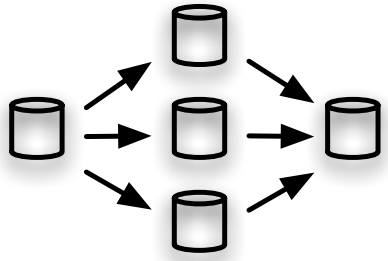

Figure 15.1: The basic idea of external merging

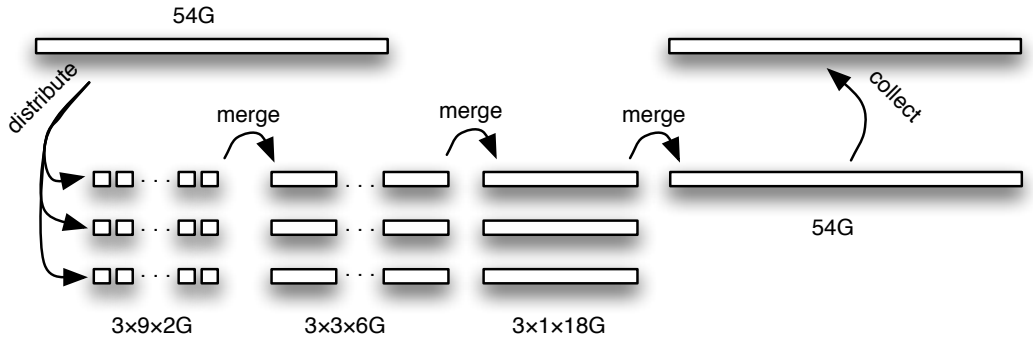

Figure 15.2: Balanced Multiway Merge Sort

the end of the run. E.g., if we consider $+\infty$ as our sentinel value, then the sequence 8, 11, 12, 50, 65, $+\infty$, 7, 23, 76, $+\infty$, 33, 45, 53, 98, $+\infty$ might be a file that consists of 3 runs of length 5, 3 and 4 respectively. In the example shown in figure 15.2 we have used a scheme with 27 counted runs of length 2 gigabytes.

The merging phase merges the runs on the auxiliary files in order to generate longer runs on yet other auxiliary files. The process is further illustrated in figure 15.2. In the example, 3 additional auxiliary files are used which brings the total to 6. The merging process takes the first run from every auxiliary file and merges this into one run (that is 3 times as long). Every newly generated run is written to one of the additional auxiliary files and this process is repeated until the original auxiliary files are all empty. As soon as the merge is finished, the 3 input files become 3 output files and vice versa. Again, 3 runs are merged into 1 run and so on. This process

is repeated until the auxiliary files are merged into a file containing one single sorted run. Every merge divides the overall number of runs by 3 while it multiplies the length of the runs by 3. The more often the process needs to be repeated, the slower the algorithm will be. As we will see, merging runs can be a very complex process and various techniques exist that all have their advantages and disadvantages.

The collection phase collects the sorted data from the final auxiliary file and writes it back to the original file. This is shown on the right hand side of figure 15.2. In a truly efficient implementation, the collection phase is integrated with the very last step of the merging phase. In our implementations, we have chosen not to do this for didactic purposes since such an integration tends to clutter up the code of the merging phase with additional `if` tests.

Remember from chapter 14 that an open file only requires 2 disk blocks that have to be stored in central memory[1]: one to keep a reference to the header of the file and another one to store its "current buffer block". Every time an input buffer block is entirely consumed, the next block is read and the previous block can be safely garbage collected by Scheme. *Therefore, the key insight for understanding all merge sorts of this chapter, is that we only need* $2 \times p$ *buffer blocks of central memory to merge* p *files because this is the number of blocks needed to keep* p *files open at the same time. This does not depend on the length of those files.* It is really crucial to understand this!

15.2 Balanced Multiway Merge Sort

The first algorithm that we discuss is the balanced multiway merge sort. This is the algorithm that was exemplified in figure 15.2. The idea of the algorithm is to work with $2p$ auxiliary files, for some constant p. The distribution phase distributes the original data over p auxiliary files. In every iteration of the merge phase, the data on p auxiliary files is merged onto the other p auxiliary files (thereby emptying the first p auxiliary files). Hence, the algorithm has p input files and merges these onto p output files. When this is done, both groups of p files are swapped: the p input files become the output files and the other way around. This process is repeated until a final merge of p auxiliary files merges all the data onto 1 auxiliary file.

Suppose that the length of the runs on all p input files is r at a certain moment in time. In the next merge, we will thus continuously consider p runs of length r from the input files and merge them into one run of length $r.p$ that is written to one of the p output files. By selecting the 'next' output file in every iteration of the merge, the new runs of length $r.p$ are evenly distributed

[1] We could avoid storing the header in central memory since it is only really needed when opening and closing the file. Hence, the central memory required for keeping a reference to an open sequential file could be reduced to 1 block. Opening p files for merging then still requires p blocks of central memory. But no more than p!

on the output files. After swapping input and output files, the next merge can take place, during which p runs of length $r.p$ will be merged into runs of length $r.p.p$. And so on. The process stops as soon as the run length equals the length of the original file. Computing the next file onto which a run is written is easily achieved with the following Scheme procedure:

```
(define (next-file indx p)
  (mod (+ indx 1) p))
```

15.2.1 Files with Counted Runs

In what follows, we will be writing (resp. reading) data to auxiliary output (resp. input) files thereby keeping track of the *number* of data value that have already been written (reps. read). This is possible because, in balanced multiway merge sort, all runs are of equal length (An exception to this rule is the very last run which may be slightly shorter). This fact allows us to keep track of the runs by simply counting the number of data values that are written (resp. read). No special "end of run" marker needs to be written to a file.

Instead of cluttering up the main code of our sorting algorithm with the run counting logic, we hide it in a simple ADT: `output-file-with-counted-runs` (resp. `input-file-with-counted-runs`) represents an output file (resp. input file) that acts like a regular sequential file. The only difference is that its constructor `new` takes a Scheme number corresponding the desired run length. The `write!` (resp. `read`) operation keeps track of the number of data values written (resp. read). As soon as more data values are written (resp. read), an error is generated. In order to prevent this from happening, the main algorithm needs to start a new run by calling `new-run!`. This resets the internal counter to zero. Hence, the `output-file-with-counted-runs` and `input-file-with-counted-runs` ADTs protect us from making mistakes in our main sorting logic. Only an erroneous usage of `write!`, `read` and `new-run!` will cause their implementation to generate a runtime error. In a bug free version of the sorting algorithm, this is not supposed to happen!

```
1  ADT output–file–with–counted–runs
2
3  new
4      ( disk string number → output–file–with–counted–runs )
5  reread!
6      ( output–file–with–counted–runs number → input–file–with–counted–runs )
7  close−write!
8      ( output–file–with–counted–runs → ∅ )
9  file
10     ( output–file–with–counted–runs → output–file )
11 run−length
12     ( output–file–with–counted–runs → number )
13 name
14     ( output–file–with–counted–runs → string )
```

```
15 delete!
16     ( output–file–with–counted–runs → ∅ )
17 new–run!
18     ( output–file–with–counted–runs → ∅ )
19 write!
20     ( output–file–with–counted–runs any → ∅ )
```

`input-file-with-counted-runs` has one additional operation that can be called to check whether or not "the current run" still contains unread data: `run-has-more?` returns #f if and only if all the data on the current run has been read. Again, `new-run!` has to be called to start a new run whenever this is the case. Erroneously reading data from an empty run generates an error.

We omit the implementation of both ADTs because they are fairly trivial. They can be represented by a record that keeps a reference to a sequential file (as presented in the previous chapter), a number indicating the run length and a counter that represents the number of data values written (resp. read). All operations simply call the corresponding operations on the actual "underlying" sequential file. `write!` and `read` adjust the counters and perform the necessary tests to generate an error should this be needed.

```
1  ADT input–file–with–counted–runs
2
3  rewrite!
4      ( input–file–with–counted–runs number → output–file–with–counted–runs )
5  close–read!
6      ( input–file–with–counted–runs → ∅ )
7  file
8      ( input–file–with–counted–runs → input–file )
9  run–length
10     ( input–file–with–counted–runs → number )
11 name
12     ( input–file–with–counted–runs → string )
13 delete!
14     ( input–file–with–counted–runs → ∅ )
15 new–run!
16     ( input–file–with–counted–runs → ∅ )
17 has–more?
18     ( input–file–with–counted–runs → boolean )
19 run–has–more?
20     ( input–file–with–counted–runs → boolean )
21 read
22     ( input–file–with–counted–runs → any )
23 peek
24     ( input–file–with–counted–runs → any )
```

In the rest of this chapter, we consistently prefix the operations of `output-file`

by `out`, the ones of `input-file` by `in`, the ones of `output-file-with-counted-runs` by `ofcr` and the ones of `input-file-with-counted-runs` by `ifcr`.

15.2.2 The Initial Runs Buffer

The distribution phase generates the initial runs. It consists of a loop that reads part of the input file in a vector, sorts that vector by means of an internal sorting algorithm (such as quick sort) and writes the sorted vector to one of the p auxiliary files as a freshly created run. Hence, the initial run length on the auxiliary files is equal to the size of the vector. This vector is called the *buffer* which is slightly confusing since it has very little in common with the buffer block of sequential files as presented in the previous chapter. The vector is filled, sorted and emptied in a loop until the entire input file has been distributed. In a realistic implementation, this buffer is chosen to be as large as possible in order to make the initial run length as long as possible. After all, the longer the initial run length, the less merging will be needed. The only limitation on the initial run length is the size of our computer's central memory.

Below we show a simple implementation of the buffer. `read-run!` fills the buffer as much as possible and returns the number of data values that were effectively read. This will be `rlen` except for the very last time when the input file does not contain enough data anymore to fill the buffer entirely. E.g., if we have an input file containing 183 numbers and if the size of the buffer is 10, then the procedure will return '10' 18 times[2]. The very last application will return '3'. The number returned can be used as the second argument to `write-run!` in order to flush the correct number of data values to an output file.

```
(define rlen 10)
(define irun (make-vector rlen))

(define (read-run! file)
  (let loop
    ((indx 0))
    (cond ((or (= indx rlen) (not (in:has-more? file)))
           indx)
          (else
           (vector-set! irun indx (in:read file))
           (loop (+ indx 1))))))

(define (write-run! ofcr imax)
  (let loop
    ((indx 0))
    (ofcr:write! ofcr (vector-ref irun indx))
    (if (< (+ indx 1) imax)
        (loop (+ indx 1)))))
```

[2]Notice that we have chosen a buffer of size 10 for didactic reasons and for reasons of debugging. In a realistic implementation, the buffer is several megabytes or even gigabytes large.

Notice that the buffer is filled from an ordinary input file (as described in section 14.4.2) while it is emptied to an output file with counted runs. Remember that we assume that the `input-file` ADT has been imported using the prefix `in:` and that the `output-file-with-counted-runs` ADT has been imported with prefix `ofcr:`.

15.2.3 The Bundle of Auxiliary Files

In what follows, we will use a little data structure to bundle the $2p$ auxiliary files in order to easily transport them through our Scheme procedures. It consists of the "order" of the sort (i.e. the number p), a vector of p input files and a vector of p output files. Here is the code to construct that data structure along with its accessors.

```
(define (make-bundle p in out)
  (cons p (cons in out)))

(define (order files)
  (car files))

(define (input files indx)
  (vector-ref (cadr files) indx))

(define (output files indx)
  (vector-ref (cddr files) indx))
```

Given such a bundle `b`, then (`for-each-input b (lambda (p i) ...)`) can be used to apply a procedure `p` to every input file. Similarly, `for-each-output` is used to apply some procedure to every output file. The second argument of the procedure (i.e. `i`) is bound to the index of the file in the bundle of auxiliary files.

```
(define (for-each-input files proc)
  (define nrfs (order files))
  (do ((indx 0 (+ indx 1)))
    ((= indx nrfs))
    (proc (input files indx) indx)))

(define (for-each-output files proc)
  (define nrfs (order files))
  (do ((indx 0 (+ indx 1)))
    ((= indx nrfs))
    (proc (output files indx) indx)))
```

The bundle is created by `make-aux-bundle` as shown below. It is called by the main sorting procedure and receives a vector of disks onto which the auxiliary files will be created. The procedure calculates the order of the sort `p` and constructs the vector holding the input files `in` as well as the vector holding

the output files `out`. `make-aux-bundle` creates the p input files and the p output files on the formatted disks. All these files have a file name of the form `aux-x` where `x` is the index of the corresponding disk in the vector of disks. Notice that the initial run length of all auxiliary files is initialized to `rlen`; the size of the buffer. In the distribution phase, only the output files will be used. However, as soon as the merging phase takes over, we have to make sure that both vectors contain the right type of sequential files. The input files are therefore created by creating an empty output file and immediately applying `reread!` on them.

```
(define (make-aux-bundle disks)
  (define p (floor (/ (vector-length disks) 2)))
  (define in  (make-vector p))
  (define out (make-vector p))
  (define name "aux-")
  (do ((i 0 (+ i 1)))
    ((= i p))
    (vector-set! out i (ofcr:new (vector-ref disks i)
                                 (string-append name (number->string i))
                                 rlen))
    (vector-set! in i (ofcr:new (vector-ref disks (+ p i))
                                (string-append name (number->string (+ p i)))
                                rlen))
    (ofcr:reread! (vector-ref in i) rlen))
  (make-bundle p in out))
```

`delete-aux-bundle!` does the exact opposite of `make-aux-bundle`. It deletes all $2p$ files from their corresponding disk.

```
(define (delete-aux-bundle! files)
  (for-each-input files
                  (lambda (file indx)
                    (ifcr:delete! file)))
  (for-each-output files
                   (lambda (file indx)
                     (ofcr:delete! file))))
```

Finally, we implement the code for adjusting the bundle of auxiliary files in such a way that the output files become input files and the other way around. `swap-files!?` takes care of this. It swaps the references to the vectors storing the input and output files. It uses `reread!` to turn the output files into input files with the same run length. It uses `rewrite!` in order to turn input files into output files. The run length of the output files is set to p times their old run length.

```
(define (swap-files!? files)
  (define (switch-refs)
    (define tmp input)
    (set! input  output)
```

```
    (set! output tmp))
  (define p (order files))
  (define old-run-length (ofcr:run-length (output files 0)))
  (define new-run-length (* p old-run-length))
  (for-each-output files (lambda (outp indx)
                           (ofcr:reread! outp old-run-length)))
  (for-each-input files (lambda (inpt indx)
                          (ifcr:rewrite! inpt new-run-length)))
  (switch-refs)
  (ifcr:has-more? (input files 1)))
```

Notice that the procedure returns a boolean that indicates whether or not there is more data left on input file number 1. In the very last iteration of the merging phase, all the data will be merged to a single run that sits on output file 0. No more data is written to the other output files. Hence, when those output files become input files, we terminate the merging phase as soon as input file 0 is the only input file containing data. All other input files (numbered 1 until $p-1$) will be empty. The merge process thus continues as long as `swap-files!?` keeps on returning #t.

15.2.4 The Algorithm

The general architecture of the balanced multiway merge sort is presented next. `sort!` takes a sequential input file `file` and a Scheme vector `asks` containing formatted disks. The predicate `<<?` makes the procedure generic such that it can be applied to input files storying values of whatever data type.

```
(define (sort! file dsks <<?)
  (define files (make-aux-bundle dsks))
  (distribute! file files <<?)
  (merge!     files <<?)
  (collect!   files file)
  (delete-aux-bundle! files))
```

`sort!` first creates the aforementioned bundle of auxiliary files. `distribute!` is called to distribute the input file `file` to the output files in that bundle and to turn those output files into input files. Then `merge!` is called to keep on merging the data from the p input files to the p output files of the bundle until all data is merged onto one single file (i.e. the zeroth output file). As soon as this happens, `collect!` finishes the sorting process by copying the data from that final file back to the original input file. We now discuss the three phases in detail.

Distribution

The distribution process is explained below. It perpetually reads a buffer from the input file. If at least one value was read, the buffer is sorted (using quicksort) and emptied to the current output file. The current output file is represented

by the index `indx` (which varies from 0 to $p-1$ in a cyclic way). We start by outputting runs to output file number 0. `distribute!` writes the sorted runs to the output files in a cyclic way by incrementing the index of the output file using `next-file`.

```
(define (distribute! inpt files <<?)
  (define p (order files))
  (let loop
    ((indx 0))
    (let ((nmbr (read-run! inpt)))
      (when (not (= nmbr 0))
        (quicksort irun nmbr <<?)
        (write-run! (output files indx) nmbr)
        (ofcr:new-run! (output files indx))
        (loop (next-file indx p)))))
  (swap-files!? files))
```

As an example, consider an initial file that contains 2457 numbers. Let us assume that our buffer size is 10 and that $p = 4$. In such a setup, `distribute!` will generate a total of 245 runs of length 10 and a final run of length 7. The runs will be written to the output files 0, 1, 2, 3, 0, 1, 2, 3, 0, and so on. As a consequence, file 0 will contain 62 runs of length 10. File 1 will contain 61 runs of length 10 and one final run of length 7. Files 2 and 4 will contain 61 runs of length 10. $(62 \times 10) + (61 \times 10 + 7) + (61 \times 10) + (61 \times 10) = 2457$.

Collection

`collect!` does the exact opposite of `distribute!`. It assumes that all the sorted data was left on the first input file (i.e. the input file sitting at index 0 in the bundle). `collect!` reads that file (called `last` in the code shown below) entirely and copies its contents back to the original file `inpt` which is turned into an output file by applying `rewrite!` first. Notice that `last` contains one single run. Hence, `run-has-more?` is equivalent to `has-more?` in this case.

```
(define (collect! files inpt)
  (define last (input files 0))
  (in:rewrite! inpt)
  (let loop
    ((rcrd (ifcr:read last)))
    (out:write! inpt rcrd)
    (if (ifcr:run-has-more? last)
        (loop (ifcr:read last))))
  (out:close-write! inpt))
```

Merging

We now still have to describe the actual merging process. It is implemented by the procedure `merge!` shown below. The procedure is conceived as an outer

loop `merge-files` and an inner loop `merge-runs`. The goal of the outer loop is to merge p auxiliary files to p auxiliary files. The inner loop will eat one run from each auxiliary file and generate one run (that is p times as long) on the current output auxiliary file. After every such merge, the current output file is incremented cyclically using `next-file`. As a consequence, the runs become evenly distributed on the p output files.

```
(define (merge! files <<?)
  (define heap (heap:new (order files)
                         (lambda (c1 c2)
                           (<<? (cdr c1) (cdr c2)))))
  (let merge-files
    ((out-idx 0))
    (cond ((read-from-files? heap files)
           (let merge-p-runs
             ((rcrd (serve heap files)))
             (ofcr:write! (output files out-idx) rcrd)
             (if (not (heap:empty? heap))
                 (merge-p-runs (serve heap files))))
           (ofcr:new-run! (output files out-idx))
           (merge-files (next-file out-idx (order files))))
          ((swap-files!? files)
           (merge-files 0)))))
```

In chapter 5 we have explained merge sort. This algorithm basically consists of continuously comparing 2 values using some `<<?` procedure and by selecting the smallest such value. In the merging algorithm presented here, we have to compare p values and select the smallest of the p values in order to write it to the output file. Fortunately, we already know a very efficient way of determining the smallest value of p values. The answer lies in the `heap` ADT explained in section 4.4. Hence, the above `merge!` procedure creates a heap of size p. Every iteration of the outer loop starts with an empty heap and with p input files. It tries to fill the heap by reading the first data value from all p input files. This is accomplished by the `read-from-files?` procedure which considers all input files and which inserts the head of each non-empty input file in the heap.

- If at least one element was read, `read-from-files?` returns #t and `merge!`'s inner loop starts the merging process. The p runs will be merged after which `read-from-files?` is called again to refill the heap with the heads of the next p runs. The inner loop serves an element from the heap (i.e. the smallest element of the p input files) and writes it back to the current output file. This process continues until the heap becomes empty which means that all data of the p runs on the input files has been merged. At that point, a new run is demarcated on the output file and the outer loop is entered again with the next output file index.
- If `read-from-files?` returns #f, this will cause `swap-files!?` to swap the p input files with the p output files. As long as more than one input

file (namely the first file) contains data, `swap-files!?` returns #t and the outer loop continues again by calling `read-from-files?`.

The inner loop applies `serve` to extract the smallest value from the p values sitting in the heap. After writing that value to the current output file, we have to read the next value from the same input file as the one from which that smallest value was initially read (as long as the run on that file contains more values). But how can we know which file that was ? In order to solve this, every value that sits in the heap is paired with the index of the input file from which that value was originally read. This information is used by the implementation of `serve`: when removing the smallest value from the heap, we make sure to read a new value from the same file. Hence, our heap stores pairs. The car of each pair is the index of the file from which the cdr of the pair was read. Obviously, the heap only has to take the actual value into account when comparing values. This explains the lambda from in the construction of the heap (i.e. `heap:new`).

`serve` now looks as follows. It deletes a pair from the heap. As long as the file's run contains more values, a new value is read from the same file (the file number is indicated by the cdr) and inserted in the heap with the same index.

```
(define (serve heap files)
  (define el (heap:delete! heap))
  (define indx (car el))
  (define rcrd (cdr el))
  (when (ifcr:run-has-more? (input files indx))
    (heap:insert! heap (cons indx (ifcr:read (input files indx)))))
  rcrd)
```

`read-from-files?` is also quite simple to implement. Every input file is considered. If an input file has not been completely emptied, a new run is started and the first value of that run (together with the index of the input file) is inserted in the heap. If at least one input file has caused a value to be inserted, #t is returned from the procedure.

```
(define (read-from-files? heap files)
  (for-each-input
   files
   (lambda (file indx)
     (when (ifcr:has-more? file)
       (ifcr:new-run! file)
       (heap:insert! heap (cons indx (ifcr:read file))))))
  (not (heap:empty? heap)))
```

15.2.5 Performance

Remember that it is our goal to keep the number of block transfers as small as possible. In our merge sort algorithm, this number is entirely determined by the perpetual reading and writing of all the data that sits on the auxiliary files. Every time we merge the p input files into p output files, we access all the data.

We therefore speak of a *pass* through the data. Hence, the question really boils down to how many passes the algorithm performs.

If n is the total number of data values that sit on the input file and if m is the size of the input buffer in main memory (i.e. 10 in our example), then the initial run length is m and we will generate a total of $\lceil \frac{n}{m} \rceil$ runs. In every pass, we continuously merge a maximum[3] of p runs into runs that are p times as long. Hence, in every pass, the length of the runs is multiplied by p and the total number of runs is divided by p. The process continues until we are left with one single run. Hence, the question becomes: how often can we divide $\lceil \frac{n}{m} \rceil$ by p before we reach 1. Clearly the answer is $\lceil log_p(\lceil \frac{n}{m} \rceil) \rceil$. Hence, multiway balanced merge sort performs a total of $O(log_p(\frac{n}{m}))$ passes through the data.

If we have a closer look at this formula, we see that n and m are not really variables that can be changed in order to speedup the process. n is the amount of data and m is the buffer size that we have at our disposal in central memory. Hence, the only way to increase the performance of our balanced multiway merge sort algorithm is by increasing p such that the basis of the logarithm increases. In other words, we have to provide more auxiliary files (possibly on the same disks).

In practice, there is a second technique for making balanced multiway merge sort considerably faster. The idea is to come up with some clever trickery to drastically increase the initial run length generated by the distribution phase. A standard technique is known as *replacement selection*. We keep its implementation as an exercise.

15.3 Polyphase Sort

The biggest disadvantage of balanced multiway merge sort is that we need a relatively high number of auxiliary files. In cases with huge amounts of data, we can think of one auxiliary file per disk. Since the number of available disks is usually fixed, this effectively limits the number of auxiliary files that we can use. More importantly, balanced multiway merge sort cannot be used with less than 4 disks and any odd number of disks leaves at least one disk unused.

A solution to this problem is given by the polyphase sorting algorithm. In polyphase sort, we use $p + 1$ auxiliary files. Every phase of the merging process merges the contents of p input files onto 1 empty output file. However, instead of performing a balanced merge, polyphase sort uses an *unbalanced* merge: only one of the input files is effectively emptied during the merge. Enough data is kept on the other $p-1$ files such that it can be merged with the newly generated file in the next phase of the merging algorithm. This is achieved by starting the merge process with an unbalanced distribution.

Polyphase is quite a complicated algorithm. We therefore split its presentation into two sections. In this section, we explain the case $p + 1 = 3$ because

[3]Normally, p runs are merged, except for the last couple of runs on each file in case the file does not give rise to a perfect distribution.

this makes the algorithm easy to explain. In section 15.4, we then generalize the algorithm to any value for p.

Polyphase sort (with $p = 2$) works with 3 auxiliary files. The distribution phase puts the data on 2 auxiliary files in an unbalanced way: one auxiliary file will contain considerably more runs than the other one. The merge phase can then eat runs from both auxiliary files (in a pairwise fashion) and writes the merged runs to the third auxiliary file. This process continues until the shortest auxiliary file is empty. Then the polyphase process continues by merging the contents of the newly generated third file with the runs that were left on the longer input file of the previous phase. Again, we merge runs from both files until the shortest file is empty and, once again, this will leave some runs on the longest file. The goal of course is to end up with two input files that contain *one single* run that can be merged after which the collect phase can take over.

The question then becomes how to do the unbalanced initial distribution: how many runs go on the first and the second auxiliary file? Let us call the auxiliary files f_0, f_1 and f_2. We now refer to the table in figure 15.3. The goal of that table is to determine the initial distribution so that we are sure that f_0 contains one single sorted run containing all the data after the final merge. This corresponds to the last row in the table. This can only happen if the previous phase left f_0 empty (so that it can serve as an output file) and both f_1 and f_2 with one single run. This explains row $last - 1$. But how did we end up with this row based on the contents of the files in phase $last - 2$? This situation must have come into existence by merging runs from f_0 and f_2 to f_1 thereby keeping one run on f_2 for usage in phase $last - 1$. Hence, in phase $last - 2$, f_1 must have been empty, f_0 must have had one run and f_1 must have had 2 runs. Let us do the reasoning one more time. The situation in row $last - 2$ emerged from merging the situation in $last - 3$. In going from $last - 3$ to $last - 2$, the data was this time merged to f_2 since every file is used as an empty output file in its turn. This means that f_2 must have had 2 runs in phase $last - 3$. Hence, the merging process that led us from $last - 3$ to $last - 2$ merged 2 runs from the two input files. As a consequence, f_1 must have had 2 runs that are both consumed and f_0 must have had 3 runs, two of which are consumed and one of which stays on f_0 for going from $last - 2$ to $last - 1$.

Phase	f_0	f_1	f_2	Total
distribute	?	?	?	original file size
...				
$last - 3$	3	2	0	5
$last - 2$	1	0	2	3
$last - 1$	0	1	1	2
$last$	1	0	0	1

Figure 15.3: Initial Run Distribution

The more data we have on our original input file, the higher the row in the table that we need for our initial distribution. In the rightmost column, we have counted the total number of runs that can be distributed according to the unbalanced distribution in that particular row. Let us consider some examples. If we have a file with only 2 runs, the distribution file should generate two files both containing a run. If we have a file with 3 runs, the distribution should generate a file containing 1 runs and another file containing 2 runs. If we have a file with 5 runs, the distribution should generate a file containing 3 runs and another file containing 2 runs. These distributions all correctly lead to a final merge. Now we see a pattern emerging. In every row of the table, the number of runs on the two auxiliary files are two consecutive Fibonacci numbers F_{n-1} and F_{n-2}. The merge then consumes all the runs from the smallest file (i.e. the file with the smallest number of runs) and the exact same number of runs from the larger file. Hence, in every merge phase, the file that contained F_{n-1} runs will now contain $F_{n-1} - F_{n-2}$ runs. The file that contained F_{n-2} runs will be empty. The file that was empty will contain F_{n-2} merged runs. Since $F_{n-1} - F_{n-2} = F_{n-3}$ we clearly see the Fibonacci-pattern showing up.

Summarized, we need to distribute the data on two files f_0 and f_1 in such a way that the total number of runs on both initial auxiliary files are two consecutive Fibonacci numbers. If we can accomplish this, the merging process will do the rest and correctly end up at the bottom row in the table. But what if the initial data cannot be distributed into a Fibonacci number of runs? This is solved by adding a number of "dummy" runs that contain the equivalent of $+\infty$; a value that is greater than all other values that can possibly occur in the file. Such a value was called a *sentinel* in chapter 3. Dummy runs do not affect the merging process because they appear at the end of a file.

15.3.1 The Initial Runs Buffer & Dummy Runs

The following Scheme procedure calls the original `read-run!` presented in section 15.2.2. It fills the buffer as before. However, should the input file contain insufficient data to fill the buffer entirely, then the buffer is *padded* with the sentinel value `sent`. In the worst case, no values at all were read from the file. In that case, the buffer is entirely filled with sentinel values. The procedure returns the number `bsiz` which corresponds to the amount of "actual" values that were read.

```
(define (padded-read-run! file sent)
  (define bsiz (read-run! file))
  (if (and (not (= bsiz rlen))
           (not (in:has-more? file)))
      (do ((pad bsiz (+ pad 1)))
        ((= pad rlen) bsiz)
        (vector-set! irun pad sent))
      bsiz))
```

15.3.2 The Bundle of Auxiliary Files

Creating the bundle of auxiliary files is similar to the multiway merge sort case, but a bit simpler. We create three files, one on each disk sitting in the argument vector `disks`. Remember that our `merge!` procedure will be merging two runs from input files f_0 and f_1 into a run on the file f_2. Therefore, the initial run length of f_2 is equal to the sum of the run lengths of f_0 and f_1. All created files are output files. `distribute!` will write its initial runs to f_0 and f_1 and will subsequently turn these files into input files using `reread!`.

```
(define (make-aux-bundle disks)
  (define files (make-vector 3))
  (vector-set! files 0 (ofcr:new (vector-ref disks 0)
                                 "aux-0" rlen))
  (vector-set! files 1 (ofcr:new (vector-ref disks 1)
                                 "aux-1" rlen))
  (vector-set! files 2 (ofcr:new (vector-ref disks 2)
                                 "aux-2" (+ rlen rlen)))
  files)
```

Notice that the bundle of auxiliary files created by `make-aux-bundle` is just a vector. During the merge phase, the files sitting in positions 0 and 1 are the input files whereas the one sitting in position 2 is the output file. We use the following accessors to make our final code a bit more readable.

```
(define (output files)
  (vector-ref files 2))


(define (input files i)
  (vector-ref files i))
```

Deleting the files from their disk is trivial:

```
(define (delete-aux-bundle! files)
  (fwrs:delete! (vector-ref files 0))
  (fwrs:delete! (vector-ref files 1))
  (fwrs:delete! (vector-ref files 2)))
```

Finally, we discuss the procedure `next-phase!?` presented below. In every phase, we merge two files to an empty file thereby leaving one of the input files empty. This empty file is subsequently used as the target for merging in the next phase. For the sake of simplicity, we *always* merge the runs from files f_0 and f_1 onto file f_2. This means that we have to make sure that after each phase, we rotate the files in such a way that f_2 becomes the empty file and that f_1 contains the smallest number of runs. Notice that, after merging from f_0 and f_1 to f_2, f_1 becomes empty and f_2 becomes the file with the highest number of runs. We therefore rotate the three files "one position to the right" in order to have the larger file sitting at index 0, the shorter one sitting at index 1 and the empty one sitting at index 2 again. This is exactly what `next-phase!?` does.

```
(define (next-phase!? files)
  (define irln (ifcr:run-length (input files 0)))
  (define orln (ofcr:run-length (output files)))
  (define last (vector-ref files 2))
  (vector-set! files 2 (vector-ref files 1))
  (vector-set! files 1 (vector-ref files 0))
  (vector-set! files 0 last)
  (ifcr:rewrite! (output files) (+ irln orln))
  (ofcr:reread! (input files 0) orln)
  (ifcr:has-more? (input files 1)))
```

The original output file becomes an input file and therefore requires a `reread!`. Likewise, the empty input file becomes an output file and requires a `rewrite!`. What can we say about the new length of the runs? In the very beginning, both f_0 and f_1 contain runs of equal length (e.g. 10). This means that runs of length 20 will be generated on f_2. Hence, after one application of `next-phase!?`, we have two new input files: one with runs the length of which is still 10 and the other one with runs whose length is 20. After merging both files to an output file whose run length is 30, the file with run length 10 is empty. This time we are left with two files, one containing runs of length 30 and another one containing runs of length 20. These will be merged to a file whose run length will be 50. And so on. This explains why the run length of the new output file is set to (+ `irln orln`) in the implementation of `next-phase!?`.

Example Suppose that f_0 contains 13 runs of length 10 and suppose that file f_1 contains 8 runs of length 10. In other words, we have a total of 21 runs of length 10 sitting on our files after the initial distribution. The first merging phase merges 8 runs of length 20 to f_2. This empties f_1 and leaves 5 runs on f_0. Then the files are rotated. After the rotation, f_0 contains 8 runs of length 20 and f_1 still contains 5 runs of length 10. This time, we can merge 5 runs of length 20 and 5 runs of length 10 into 5 runs of length 30. f_1 is emptied and f_0 still contains 3 runs of length 20. Again we rotate the files leaving 5 runs of length 30 on f_0 and 3 runs of length 20 on f_1. This time the merge generates 3 runs of length 50 emptying f_1 and leaving 2 runs of length 30 on f_0. Another rotation allows us to merge 2 runs of length 50 and 2 runs of length 30 into 2 runs of length 80. 1 single run is kept on f_0. The next rotation sets the stage to merge 1 run of length 80 with 1 run of length 50 into 1 run of length 130. The final rotation allows us to merge 1 run of length 130 and one run of length 80 into one file run of length 210. This time both f_0 and f_1 are empty.

Notice that `next-phase!?` returns a boolean. If, after the rotation, f_1 is empty, then this means that the longest input file (before the rotation) sitting in position 0 got emptied. This is only possible if *all* the data got merged onto f_2 which only occurs in the very last merge. Hence, `next-phase!?` returns `#t` if and only if more merging is required.

15.3.3 The Algorithm

The polyphase sort algorithm has exactly the same structure as the multiway merge sort algorithm. The only difference is that the algorithm is parametrized with an additional sentinel value `sent`. This value corresponds to the $+\infty$ for the data type to be sorted. It is used in the distribution phase in order to pad the file with dummy runs.

```
(define (sort! file dsks <<? sent)
  (define files (make-aux-bundle dsks))
  (distribute! file files <<? sent)
  (merge!     files <<?)
  (collect!   files file sent)
  (delete-aux-bundle! files))
```

Distribution

The distribution phase has to make sure that f_0 and f_1 are filled with an amount of runs that corresponds to two consecutive Fibonacci numbers (e.g. 13 and 8). Whatever the initial volume of data is, we have to make sure to end up in one of the rows presented in figure 15.3. The more initial data we have, the higher in the table we will end up. Hence, the idea of the distribution is to write runs to f_0 and switch files f_0 and f_1 as soon as f_0 is entirely "full" in the sense that the Fibonacci number of runs is reached. Then the files are swapped and we start filling the other file until the next Fibonacci number is reached. And so on.

```
(define (distribute! inpt files <<? sent)
  (define (swap-input files)
    (define temp (vector-ref files 0))
    (vector-set! files 0 (vector-ref files 1))
    (vector-set! files 1 temp))
  (let loop
    ((fib1 1)
     (fib2 0)
     (out-ctr 0)
     (nmbr (padded-read-run! inpt sent)))
    (cond ((< out-ctr fib1)
           (cond ((= nmbr 0)
                  (write-run! (input files 0) rlen)
                  (ofcr:new-run! (input files 0))
                  (loop fib1 fib2 (+ out-ctr 1) nmbr))
                 (else
                  (quicksort irun nmbr <<?)
                  (write-run! (input files 0) rlen)
                  (ofcr:new-run! (input files 0))
                  (loop fib1 fib2 (+ out-ctr 1) (padded-read-run!
                                                 inpt sent)))))
```

```
        ((in:has-more? inpt)
         (swap-input files)
         (loop (+ fib1 fib2) fib1 fib2 nmbr))))
  (ofcr:reread! (input files 0) (ifcr:run-length (input files 0)))
  (ofcr:reread! (input files 1) (ifcr:run-length (input files 1))))
```

The `distribute!` procedure consists of a loop that manages four variables: `fib1`, `fib2`, `out-ctr` and `nmbr`. `fib1` and `fib2` are two consecutive Fibonacci numbers, the largest one of which is `fib1`. At any moment in time during the execution of the loop, we assume that f_1 is full because it already contains `fib2` runs. `out-ctr` is initialized to 0 and counts the number of sorted runs that is written to f_0. As long as (< `out-ctr fib1`), we keep on writing runs to f_0. However, as soon as `out-ctr` is equal to `fib1` (and more input is available), we swap both files and we compute the next pair of Fibonacci numbers to enter the loop: `fib1` is replaced by (+ `fib1 fib2`) and `fib2` is replaced by `fib1`. Notice that the new value of `out-ctr` becomes `fib2` since we keep on writing runs to f_0. However, that file already contains `fib2` runs from the previous iteration.

The job of the `loop`'s body is to fill the buffer using `padded-read-run!`. If the number of actual values on the run is 0, we just empty the (dummy) buffer and keep on looping with the same unmodified buffer (which only contains sentinel values). If at least one value was encountered, we quicksort the buffer and write it to f_0. In that case, we call `padded-read-run!` again and re-enter the loop.

The `distribute!` procedure finishes by turning both f_0 and f_1 into input files. This is where `merge!` takes over.

Collection

The collection phase is almost identical to the one of balanced multiway merge sort. The only difference is that we stop collecting as soon as the sentinel value is encountered. Remember that we used sentinel values to (a) pad the last incomplete run and to (b) generate the dummy runs. Since the sentinel value corresponds to $+\infty$ for our ordering relation <<?, it is guaranteed to be located at the end of the sorted file. Hence, we are guaranteed not to lose any data if we break off the collection phase as soon as we encounter the first sentinel value.

```
(define (collect! files inpt sent)
  (define last (input files 0))
  (in:rewrite! inpt)
  (let loop
    ((rcrd (ifcr:read last)))
    (out:write! inpt rcrd)
    (if (ifcr:run-has-more? last)
        (let ((rcrd (ifcr:read last)))
          (if (not (eq? rcrd sent))
              (loop rcrd)))))
  (out:close-write! inpt))
```

Merging

The structure of the merge code is very similar to the one of multiway merge sort. The procedure `read-from-files?` is used in conjunction with `next-phase!?` to do the merging. In an outer loop (called `merge-files`), `read-from-files?` is called and as long as it returns `#t` (i.e. two new runs have started because data was still available on the shortest input file), we merge those runs in an inner loop (called `merge-2-runs`). The newly generated run on the output file is properly marked after merging the two input runs. As soon as `read-from-files?` returns `#f` because f_1 is exhausted, the files are rotated. If this results in `#t` because the (new) shortest input file still contains data, the outer loop is entered again. Otherwise, only f_0 has data which means that the sorting process is done.

```
(define (merge! files <<?)
  (define heap (heap:new 2
                         (lambda (c1 c2)
                           (<<? (cdr c1) (cdr c2)))))
  (let merge-files
    ()
    (cond ((read-from-files? heap files)
           (let merge-2-runs
             ((rcrd (serve heap files)))
             (ofcr:write! (output files) rcrd)
             (if (not (heap:empty? heap))
                 (merge-2-runs (serve heap files))))
           (ofcr:new-run! (output files))
           (merge-files))
          ((next-phase!? files)
           (merge-files)))))
```

The `serve` procedure is identical to the one for balanced multiway merge sort: it deletes the smallest element from the heap and refills the heap with a value from the same file as the file from which the smallest element was originally read before it was inserted in the heap.

`read-from-files?` is shown below. It has the same semantics as its balanced multiway merge sort counterpart. Its implementation is slightly different since we only have two files to read from. A new run is started on both input files and a refill of the heap happens only if the shortest file (i.e. `ifcr2` sitting in position 1) still contains data. A boolean is returned that tells `merge!` whether or not the procedure succeeded in starting two new runs.

```
(define (read-from-files? heap files)
  (define ifcr1 (input files 0))
  (define ifcr2 (input files 1))
  (ifcr:new-run! (input files 1))
  (ifcr:new-run! (input files 0))
  (when (ifcr:has-more? ifcr2)
    (heap:insert! heap (cons 0 (ifcr:read ifcr1)))
```

```
    (heap:insert! heap (cons 1 (ifcr:read ifcr2))))
  (not (heap:empty? heap)))
```

15.3.4 Performance

In the next section, we present a generalization of this algorithm to other values of p. The performance of the general algorithm is studied afterwards.

15.4 p-Polyphase Sort

Most textbooks that treat polyphase sort immediately present the version for p auxiliary files. As we will see, the general version uses Fibonacci numbers of the p'th order. From a didactic point of view, however, it is easier to understand the algorithm after having studied the version that uses "ordinary" Fibonacci numbers (i.e. the Fibonacci numbers of order 2). This was the algorithm presented in the previous section.

15.4.1 The Bundle of Auxiliary Files

Our bundle of files now contains the $p + 1$ auxiliary files along with some important runtime information about those files.

- As we will see in a moment, we no longer actually write the dummy runs to the files. Instead, we maintain one counter per auxiliary file that *counts* the number of dummies that sit on that file. E.g., if a file can host 44 runs and we just wrote 32, then we know that the file 'still' has 12 dummy runs left. Every time we write a run, the number of dummies for that file is decremented by 1. This information is stored in the vector `dummies`. The vector `runs` stores the total number of runs (including the dummy runs) that are allowed to sit on each auxiliary file.
- As we will see, not every input file will participate in every phase of the merge process. Depending on the number of dummies on a file, we will determine whether or not to consume data from that file into our heap. If that is the indeed the case, the file is *selected* and its corresponding entry in the `selected` vector will be #t.

The following record definition defines the bundle of auxiliary files for our generalised polyphase sort. It keeps a reference to a vector of booleans (`selecteds`), a vector of files (`files`), a vector of numbers indicating the runs that can sit on those files (`runs`) and a vector of numbers indicating the dummies sitting on those files (`dummies`).

```
(define-record-type bundle
  (make-bundle s d r f)
  bundle?
  (s selecteds)
```

```
  (d dummies)
  (r runs)
  (f files))
```

We can access the vectors' entries using the following accessors:

```
(define (dummy fils indx)
  (vector-ref (dummies fils) indx))
(define (dummy! fils indx cnt)
  (vector-set! (dummies fils) indx cnt))
(define (run fils indx)
  (vector-ref (runs fils) indx))
(define (run! fils indx cnt)
  (vector-set! (runs fils) indx cnt))
(define (file fils indx)
  (vector-ref (files fils) indx))
(define (file! fils indx file)
  (vector-set! (files fils) indx file))
(define (selected fils indx)
  (vector-ref (selecteds fils) indx))
(define (selected! fils indx sltd)
  (vector-set! (selecteds fils) indx sltd))
```

Finally, we provide abstractions that allows us to refer to one of the $p + 1$ files as input files or as the output file:

```
(define (output fils)
  (vector-ref (files fils) (order fils)))

(define (input fils i)
  (vector-ref (files fils) i))

(define (order fils)
  (- (vector-length (files fils)) 1))
```

The bundle of auxiliary files and their information is created by `make-aux-bundle` (Its counterpart, `delete-aux-bundle`, is omitted). It receives a vector of disks and creates an auxiliary file on each disk. The very last disk contains the output file. All other disks contain input files. As explained below, the length of the runs sitting on these files is longer be fixed. Consecutive runs sitting on one single file can vary. Such files are called `input-file-with-varying-runs` and `output-file-with-varying-runs`. We discuss these ADTs later in this chapter. We assume that they are imported using the prefixes `ofvr:` and `ifvr:`.

```
(define (make-aux-bundle disks)
  (define p   (- (vector-length disks) 1))
  (define rslt (make-bundle (make-vector (+ p 1))
                            (make-vector (+ p 1))
                            (make-vector (+ p 1))
```

```
                              (make-vector (+ p 1))))
(define name "aux-")
(do ((indx 0 (+ indx 1)))
  ((= indx p))
  (file! rslt indx  (ofvr:new (vector-ref disks indx)
                              (string-append name (number->string indx))
                              +inf.0))
  (run!   rslt indx 1)
  (dummy! rslt indx 1)
  (selected! rslt indx #f))
(file! rslt p (ofvr:new (vector-ref disks p)
                        (string-append name (number->string p))
                        +inf.0))
(run!   rslt p 0)
(dummy! rslt p 0)
rslt)
```

The `do` loop creates p output files (f_0 to f_{p-1}) that are used by `distributed!` for the initial run distribution. The $(p+1)$-th file f_p is the first output file that will be used as an output file by `merge!`. The initial values for `selected`, `runs` and `dummies` will become clear later.

15.4.2 The Algorithm

The algorithm follows exactly the same structure as the previous two algorithms. In what follows, we discuss the distribution phase, the merging phase and the collection phase.

Distribution

The goal of the distribution phase is to take an input file `inpt` and a bundle of auxiliary files `fils` onto which we have to organize the initial distribution. The question is how to distribute the (sorted) runs on the p output files. Let us do the exercise with $p = 4$. In figure 15.4, we show a table with possible distributions. The longer the original input file, the higher in the table we find a row that can accommodate enough runs in order to distribute our file.

As in the case for $p = 2$, the goal is to end up (after successive merging) with one single run in the last row. The only way to end up with this row is if the previous phase had an empty last file and one run on all the other files. Let us chose file f_0 for this. This explains row $last - 1$. Let us now derive row $last - 2$. In order to get one empty file on row $last - 1$ all the runs have to be read from some file, say f_1. In going from $last - 2$ to $last - 1$, this means that 1 run was taken from f_1. But then one run must have been taken from all other files as well. This 1 is added to the number of runs that have to stay on the files for phase $last - 1$. This explains the numbers in the row $last - 2$. Let us derive the numbers for row $last - 3$. One file needs to be empty in order to merge towards $last - 2$. This time we select f_2. Hence, we see that 2 runs were

Phase	f_0	f_1	f_2	f_3	f_4	Total
distribute	?	?	?	?	?	input file size
...						
$last-6$	0	29	27	23	15	94
$last-5$	15	14	12	8	0	49
$last-4$	7	6	4	0	8	25
$last-3$	3	2	0	4	4	13
$last-2$	1	0	2	2	2	7
$last-1$	0	1	1	1	1	4
$last$	1	0	0	0	0	1
collect	0	0	0	0	0	0

Figure 15.4: Initial Run Distribution

taken from f_2 when going from $last-3$ to $last-2$. But then 2 runs must have been taken from all other files as well. Hence, we add 2 to what is expected to stay on the files (i.e. the numbers in row $last-2$. This explains row $last-3$. In the previous phase (i.e. $last-4$), f_3 was the file that was empty to do the merge towards $last-3$. Hence, 4 runs were written to f_3. But then 4 runs must have been eaten from all other files as well. Together with the number of runs that have to stay on those files for phase $last-3$, this gives us row $last-4$: all we need to do is add 4 to the information in row $last-3$ in order to obtain the numbers for row $last-4$. And so on. The rightmost column shows the total number of runs that can be distributed with the distribution of every row.

Figure 15.5 displays the same table. However, instead of determining the index of the file that is to remain empty, we rotate the files themselves. Like this, f_4 is always empty before the start of the merging phase. We always merge to f_4 and then we rotate all the files "one position to the right", thus turning f_4 into f_0. The old f_3 becomes f_4. This means that — in every phase — f_3 is the file that needs to be entirely emptied in order to prepare it for becoming f_4 in the next phase.

Let us now try to come up with a general formula to generate a new row from any given row. This is what the distribution algorithm does: given an input file, it tries to distribute its runs in order to end up with one of the ideal distributions shown in one of the rows. From that point on, the merging phase takes us downward in the table and we end up with one run in the first file (which is then collected).

The general idea of the merge is exactly the same as with the simplified polyphase sort of section 15.3: all the runs of the shortest file are consumed and some runs are kept on the other files for merging in the next phase. After the merge, the files are rotated "to the right" which means that the resulting runs end up in file number f_0 in the next phase. This means that we can calculate the ideal distribution of a row by considering the number of runs on file f_0 of

Phase	f_0	f_1	f_2	f_3	f_4	Total
...						
$last-6$	29	27	23	15	0	94
$last-5$	15	14	12	8	0	49
$last-4$	8	7	6	4	0	25
$last-3$	4	4	3	2	0	13
$last-2$	2	2	2	1	0	7
$last-1$	1	1	1	1	0	4
$last$	1	0	0	0	0	1
$collect$	0	0	0	0	0	0

Figure 15.5: Initial Run Distribution (Rotated)

the row right underneath, and by adding this number to all other numbers in the row right underneath. This is because those numbers indicate the number of runs that stay on those files after the merge on that level. In other words,

$$f_i^{n+1} = f_0^n + f_{i+1}^n$$

Hence, the idea of the distribution phase will be to generate the $(n+1)$-th row (i.e. $\forall\ i : f_i^{n+1}$) based on the numbers of the n-th row (i.e. the f_i^n). Of course, each such row assumes an *ideal* situation where the actual number of runs on the input file is indeed the sum of all the runs after following the ideal distribution on that particular row. Below we explain how a system of dummies is used to cover the situations where the number of actual runs does not correspond to one of the ideal rows in the table.

Let us fist try to understand the meaning of those numbers from a mathematical point of view. Let us focus on the first column and try to understand the meaning of f_0^n for any n. We just keep on applying the above formula and substitute the result recursively. Since $f_0^{n+1} = f_0^n + f_1^n$, and $f_1^n = f_0^{n-1} + f_2^{n-1}$, we get $f_0^{n+1} = f_0^n + f_0^{n-1} + f_2^{n-1}$. We apply the same substitution, this time using $f_2^{n+1} = f_0^n + f_3^n$. This gives us $f_0^{n+1} = f_0^n + f_0^{n-1} + f_0^{n-2} + f_3^{n-2}$. Finally, since $f_3^{n+1} = f_0^n + f_4^n$ and $f_4^n = 0$, we get $f_0^{n+1} = f_0^n + f_0^{n-1} + f_0^{n-2} + f_0^{n-3}$. Hence, if we trace the origin of the greatest number in every row, it turns out to be the sum of the greatest numbers in the p previous rows!

In general, the sequence of numbers in which every number is the sum of the previous p numbers is known as the *sequence of Fibonacci-numbers or order p.*

$$F_{(p)}^n = F_{(p)}^n + F_{(p)}^{n-1} + F_{(p)}^{n-2} + ... + F_{(p)}^{n-p}$$

The sequence starts with $p-1$ zeroes followed by a 1. The "ordinary" Fibonacci-numbers perfectly fit this definition for $p = 2$. This insight corresponds to the distribution we have seen in section 15.3: the perfect distribution in every row consisted of two consecutive Fibonacci numbers of order 2. This finishes our explanation of how to interpret the initial distribution mathematically.

Let us now have a look on *how* to distribute the input data given the numbers for an ideal distribution. The difficulty lies in what to do when our initial number of runs is not a pth order Fibonacci number. In that case, we have to take the dummy runs into account. Suppose that we have an initial file with 63 runs and suppose that we are working with $p = 4$. From the table in figure 15.5, we can see that at some point, we will be considering the ideal distribution $(15, 14, 12, 8)$. Since we have 63 runs, this ideal distribution will be entirely filled because that distribution can only accommodate $15 + 14 + 12 + 8 = 49$ runs. Computing the next *ideal* row gives us $(39, 27, 23, 15)$. Since the previous row was filled entirely, we have to subtract the runs that have already been filled from the next ideal situation. Hence, in the new situation, we start with the following *number* of dummy runs on each file: $(39-15, 27-14, 23-12, 15-8) = (14, 13, 9, 7)$. Every time we write a run on one of these files, the corresponding dummy count is decremented. As soon as we reach $(0, 0, 0, 0)$, we have to compute the next row because the current distribution has been entirely filled with actual runs (i.e. there are no more dummy runs left).

Let us now focus on $(14, 13, 9, 7)$. We have $63-49 = 14$ runs that still need to be distributed. If we generalize the algorithm presented in section 15.3, we write those 14 runs to the first file, leaving room for $(0, 13, 11, 7)$ dummies. This leads to suboptimal merging because all those dummy runs will be merged with one single actual run (i.e. the ones on the first file). This simply causes the real runs to be copied without doing any useful work. Obviously, a much more efficient sorting is possible if we try to spread out the runs and the dummy runs as evenly as possible. In that case, the merge algorithm merges actual runs with actual runs and merges dummy runs with dummy runs. Unfortunately, there does not seem to exist a single best distribution. The following procedure generates *an* initial distribution. The method is due to D. Knuth [Knu98] and it seems to be provide fairly good results given its simplicity. The Knuth distribution rule writes runs from left to right until the amount of dummies on a file is equal to the amount of dummies on the file sitting immediately to its right. Figure 15.6 shows the resulting evolution of the number of dummies on each file. As we can see, this algorithm never excessively fills one file in detriment of another file. Alls files are evenly filled with data.

As soon as all the dummies have been replaced by actual data, we have to compute the next ideal row in the table. As explained above, the runs are computed following the rule:

$$f_i^{n+1} = f_0^n + f_{i+1}^n$$

The dummies are calculated by subtracting the previous value f_i^n from the new value f_i^{n+1}, i.e. $d_i^{n+1} = f_i^{n+1} - f_i^n$. Therefore,

$$d_i^{n+1} = f_0^n + f_{i+1}^n - f_i^n$$

This explains the `do` loop in the `distribute` procedure shown below. The procedure itself consists of a `loop` that governs two iteration variables: `nmbr` is

f_0	f_1	f_2	f_3
14	13	9	7
13	13	9	7
12	13	9	7
12	12	9	7
11	12	9	7
11	11	9	7
10	11	9	7
10	10	9	7
9	10	9	7
9	9	9	7
...	...	...	...

Figure 15.6: The Knuth Distribution Rule

the number of values that were effectively read from the input file and `indx` is the index of the current output file. When at least one data value was read (i.e. (> `nmbr` 0)), we quicksort the buffer and we write it as a run to the current output file (i.e. (`input fils indx`)). The algorithm keeps track of the amount of dummies sitting on every file. Hence, every time we write a sorted run to a file, we decrement its number of dummies by 1.

```
(define (distribute! inpt fils <<?)
  (define p (order fils))
  (let loop
    ((indx 0)
     (nmbr (read-run! inpt)))
    (when (> nmbr 0)
      (quicksort irun nmbr <<?)
      (write-run! (input fils indx) nmbr)
      (ofvr:new-run! (file fils indx))
      (dummy! fils indx (- (dummy fils indx) 1))
      (cond ((< (dummy fils indx) (dummy fils (+ indx 1)))
             (loop (+ indx 1) (read-run! inpt)))
            ((= (dummy fils indx) 0)
             (let ((rmax (run fils 0)))
               (do ((i 0 (+ i 1)))
                   ((= i p))
                   (dummy! fils i (+ rmax (run fils (+ i 1)) (- (run fils i))))
                   (run!   fils i (+ rmax (run fils (+ i 1)))))
               (loop 0 (read-run! inpt))))
            (else
             (loop 0 (read-run! inpt))))))
  (do ((indx 0 (+ indx 1)))
    ((= indx p))
    (ofvr:reread! (input fils indx))))
```

```
(eat-dummies fils))
```

The penultimate `do` expression of the procedure turns the filled files into output files by applying `reread!` on them. What can we say about the very last expression of the `distribute!` procedure? Remember that it is our goal to distribute the dummies as evenly as possible. Suppose for example, that we end up with a situation that generates the dummy distribution $(6, 6, 4, 3)$ (given $p = 4$). In other words, f_0 and f_1 contain 6 dummies, f_2 contains 4 dummies and f_3 contains 3 dummies. Clearly, without doing any real work, we can already eat 3 dummy runs from each file and "merge" those 3 dummy runs into 1 dummy run on f_4. This means that we have to add 1 to the dummy count of f_4 and that we have to subtract 3 from all other dummy counts. This is precisely what `eat-dummies` does. `skip` is the number of dummies that can be skipped like this (i.e. 3 in our example). It corresponds to the smallest number of dummies sitting on any of the input files.

```
(define (eat-dummies fils)
  (define p (order fils))
  (define skip
    (let ((dmin +inf.0))
      (do ((indx 0 (+ indx 1)))
        ((= indx p)
         dmin)
        (if (< (dummy fils indx) dmin)
            (set! dmin (dummy fils indx))))))
  (dummy! fils p (+ (dummy fils p) skip))
  (do ((indx 0 (+ indx 1)))
    ((= indx p))
    (dummy! fils indx (- (dummy fils indx) skip))
    (run! fils indx (- (run fils indx) skip)))
  (run! fils p (+ (run fils p) skip)))
```

Merging

The `merge!` procedure is identical to the one explained in section 15.3. As in the previous two algorithms, `merge!` starts by filling its heap with the first value of a suite of new runs and then merges these runs into a run on the output file by repeatedly serving a value from the heap and refilling the heap from the same file (if still possible). This is again accomplished through the procedures `read-from-files?` and `serve`. We do not repeat `serve` since it is identical to the version already presented. `read-from-files?` is shown below. After all runs of a phase are consumed, `read-from-files?` is called to refill the heap with the first values of p new runs. However, not all files participate in every merge: when a file contains at least one dummy run (even after eating dummies using the above procedure), we consume that dummy run (by decrementing its dummy count) and the file does not participate in the merge. If a file contains no dummies, it is selected by the merge. An auxiliary procedure `start-new-runs!`

therefore first decides which files participate in the merge and which ones do not. It writes its conclusions into the boolean vector `selected`. `read-from-files?` refills the heap with the first data value of the selected files. It returns #f if no file got selected at all.

```
(define (read-from-files? heap fils)
  (define p (order fils))
  (cond ((> (run fils (- p 1)) 0)
         (start-new-runs! fils)
         (do ((indx 0 (+ indx 1)))
           ((= indx p) #t)
           (if (selected fils indx)
               (heap:insert! heap (cons indx (ifvr:read (file fils indx))))))))
        (else #f)))
```

`start-new-runs!` decrements the number of runs on every file and increments the number of runs on the target file of the merge (i.e. the p-th file). Then all the input files are considered in a second `do` loop. If the number of dummies happens to be zero, there must be actual data on the file and we select it for merging. If the number of dummies is greater than zero, we do not select the file for participation in the merge process. Instead, we consume a dummy run by decrementing the file's dummy count.

```
(define (start-new-runs! fils)
  (define p (order fils))
  (do ((indx 0 (+ indx 1)))
    ((= indx p)
     (run! fils p (+ (run fils p) 1)))
    (run! fils indx (- (run fils indx) 1)))
  (do ((indx 0 (+ indx 1)))
    ((= indx p))
    (cond ((and (ifvr:has-more? (file fils indx))
                (= (dummy fils indx) 0))
           (if (selected fils indx)
               (ifvr:new-run! (file fils indx)))
           (selected! fils indx #t))
          (else
           (selected! fils indx #f)
           (if (> (dummy fils indx) 0)
               (dummy! fils indx (- (dummy fils indx) 1)))))))
```

Since it is dynamically decided which files will and which files will not be selected for merging, this means that the runs will no longer be of the same length. That is why we have to replace our `file-with-counted-runs` ADT by a new `file-with-varying-runs` ADT. This ADT is nearly identical except that the constructor (used in `distribute!`) no longer takes a number indicating the run length. `write!` will never fail since runs no longer have a predetermined length. `new-run!` will thus mark the end of a run by writing the sentinel value specified

in the constructor. When used as input, `run-has-more?` checks whether or not the sentinel value is detected. `read` keeps on reading from the file and produces an error should the sentinel be detected. In our code, we use the prefixes `ifvr` and `ofvr` to refer to input files and output files with varying runs. We consider the specification and implementation of the ADTs trivial.

Just like in the algorithm presented in section 15.3, `merge!` rotates the files one position "to the right" as soon as `read-from-files?` returns #f. This is the task of `next-phase!?`. It consistently rotates all the information in the auxiliary file bundle: the run counts, the dummy counts and the files. Moreover, it resets all slots of the `selected` vector to #f. It performs a `reread!` on the new zeroth file and a `rewrite!` on the new last file. #t is returned as long as the $(p-1)$-th input file still contains data. This means that all data has not been merged into f_0 yet and that the merging process has to keep on going.

```
(define (next-phase!? fils)
  (define p (order fils))
  (define last-fil (file fils p))
  (define last-run (run fils p))
  (define last-dum (dummy fils p))
  (do ((indx p (- indx 1)))
    ((= indx 0))
    (file! fils indx (file fils (- indx 1)))
    (dummy! fils indx (dummy fils (- indx 1)))
    (run! fils indx (run fils (- indx 1)))
    (selected! fils indx (selected fils (- indx 1))))
  (file!  fils 0 last-fil)
  (dummy! fils 0 last-dum)
  (run!   fils 0 last-run)
  (selected! fils 0 #f)
  (ofvr:reread! (input fils 0))
  (ifvr:rewrite! (output fils))
  (eat-dummies fils)
  (> (run fils (- p 1)) 0))
```

Collection

Since we did not actually write the sentinel values to the auxiliary files, the data that is left on f_0 is the entire sorted file. Therefore, the collection phase of p-polyphase sort is exactly the same as the collection phase of balanced multiway merge sort. We do not repeat the code here.

Performance

Unfortunately, a thorough analysis of p-polyphase sort requires some fairly intricate mathematics which falls beyond the scope of this book. We thus give informal arguments to illustrate the performance improvement compared to balanced multiway merge sort.

A concrete example

Let us look at an example with 6 auxiliary files. Suppose that we have to sort 1921 runs. We pick this number because it results in a perfect distribution when using 5-polyphase sort. In a balanced multiway merge sort, this gives us about 640 runs on each of the 3 files. This means that we have to merge the data $log_3(640) = 6$ times before we end up with one single run. Figure 15.7 shows a perfect polyphase distribution with 6 auxiliary files. At first sight, this requires 10 passes through the data, which is a worse result than the 6 passes of our balanced multiway merge sort. However, the key point is that every pass of balanced multiway merge sort copies *all* the data while polyphase sort only copies part of the data in each phase (leaving data on some of the files).

Let us look at the polyphase distribution shown in figure 15.7. In the first phase, we merge 236 runs. In the second phase, 120 runs are merged. And so on. If we add all the numbers, we arrive at 236+120+61+31+16+8+4+2+1+1 = 433 runs that are merged during the entire sorting process. This is much less than the number of runs copied by the balanced multiway merge sort, which is approximately given by: 3×640+3×213+3×71+3×23+3×8+3×3+3×1 = 2877.

Phase	f_0	f_1	f_2	f_3	f_4	f_5
$last - 9$	464	448	417	356	236	0
$last - 8$	236	228	212	181	120	0
$last - 7$	120	116	108	92	61	0
$last - 6$	61	59	55	47	31	0
$last - 5$	31	30	28	24	16	0
$last - 4$	16	15	14	12	8	0
$last - 3$	8	8	7	6	4	0
$last - 2$	4	4	4	3	2	0
$last - 1$	2	2	2	2	1	0
$last$	1	1	1	1	1	0
$collect$	1	0	0	0	0	0

Figure 15.7: Initial Run Distribution with 6 files

Roughly spoken, polyphase sort is more efficient than balanced multiway merge sort because, given p auxiliary files, it always results in an $p-1$- way merge instead of an $\frac{p}{2}$-way merge. As the number of required passes is approximately $log_p(n)$, polyphase merging promises a significant improvement over balanced merging.

Quantitative Data

Figure 15.8 shows the quantitative results when we apply p-polyphase sort to files that consist of a number of initial runs S varying from 1 to 5000. Notice that the figure uses a logarithmic scale. This means that the linear graph actually has to be interpreted as a logarithmic graph when shown on a linear scale. In the graphs, it is shown how often every data value is written by the sorting algorithm. We have graphs[4] for $p = 3$ up to and including $p = 10$. The graph

[4]The graph is a scan from [Knu98]. We plan to make our own copy in a later version of

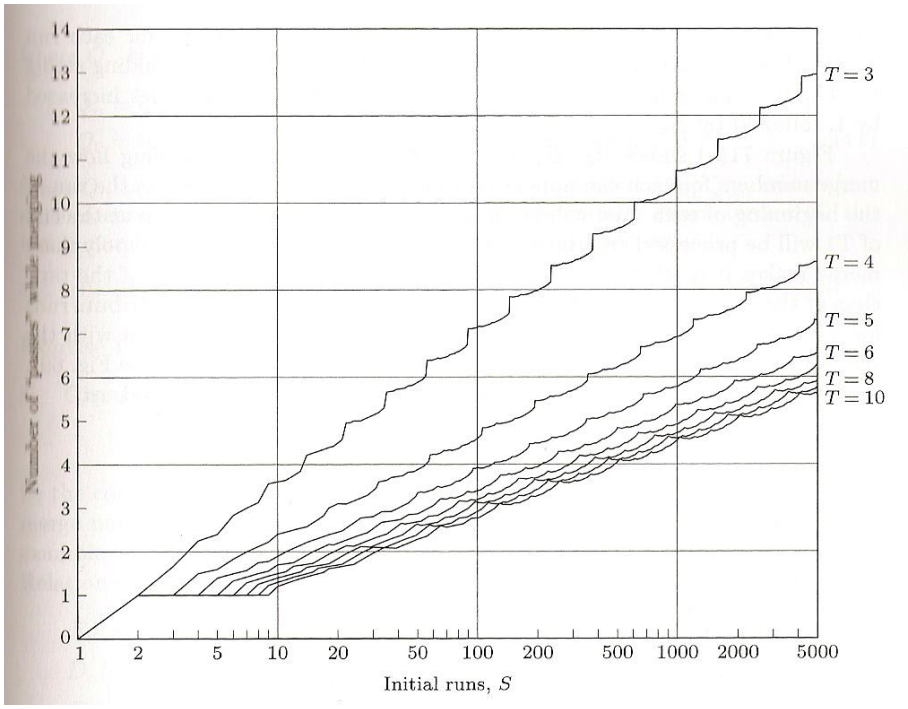

Figure 15.8: Knuth's Quantitative Data

clearly shows that there exists a sweet spot at $p = 7$. Using more auxiliary files barely has an effect on the number of times that our values are written.

15.5 Limitations of External Sorting

In section 5.4 we have proved a theoretical result stating that it is impossible to design a general internal sorting algorithm that works in less than $O(n.log(n))$ computational steps. A similar result exists for external sorting. We mention it without a formal proof:

Theorem: Let m be the size of our central memory and let b be the size of a disk block. Sorting n values requires at least $\Omega(\frac{n}{b}.log_{\frac{m}{b}}(\frac{n}{b}))$ block transfers.

Informally, $\frac{n}{b}$ is the number of block transfers that are required to consider every data value exactly once. $\frac{m}{b}$ is the number of disk blocks that fit in central memory. Hence $log_{\frac{m}{b}}$ is the number of times that we have to divide the data into central memory chunks.

this book. The graph uses T instead of p as the notation for the number of auxiliary files.

15.6 Exercises

1. Currently `distribute!` fills the buffer vector, sorts it with quicksort and then writes the buffer to an output file as a single fixed-length run. If, instead of using a plain vector, we use a heap then the data will be sorted by the heap itself (instead of requiring us to explicitly call quicksort). Implement a variation of multiway balanced merge sort that uses heapsort instead of quicksort in the `distribute!` phase. The heap will be of the same size (i.e. `rlen`) as the buffer vector currently used. Keep on using fixed-length runs.

2. Apart from increasing the number of auxiliary disks/files p (which is often impossible) we can improve the performance of multiway balanced merge sort by increasing the length of the initial runs. Longer runs means fewer runs which implies less merging work. In our implementation the run length is constrained by the amount of available central memory (i.e. the `rlen` variable). A well-known technique for generating longer runs is known as **replacement selection**. It involves using a second heap (instead of a plain vector) as the buffer. In this exercise, we implement a (naive) version of replacement selection, starting with the solution of the previous exercise.

   Instead of writing all the values sitting in the heap in one go, we write them one by one and we **replace** every written value by inserting a new value from the input file into the heap. The key insight here is that, every time a value is deleted from the heap and written to the disk, we effectively free up space in central memory which can be used immediately to store the next value of the input file. Because the heap automatically restores its ordering after every insertion this allows us to extend runs beyond the limit imposed by the size of the heap or buffer. However, because the runs will have to end at some point, we should be careful not to insert values which are smaller than those already written. This means that the length of our runs will no longer be predictable or fixed. Luckily we can reuse the in/outputfile-with-**varying**-runs ADTs (as used by p-polyphase sort) for this. Replace the use of the in/outputfile-with-**counted**-runs ADTs with those. Be sure to specify a value to mark new runs (e.g. $+\infty$). Add the replacement selection behaviour in the `distribute!` phase. End the run when a value which is smaller than those already written appears at the front of the heap.

3. The solution of the previous exercise is naive because we stop a run as soon as a value is encountered that is smaller than the values already written. In fact, this does not result in longer initial runs at all[5]. What we need is a way to temporarily store values which are too small while filling the current run. A clever way to do this is to store all values in the same heap

[5] In fact, if you add some statistics code to your solution you will find that on average it produces runs which are much shorter than `rlen`.

but to keep the ones which are too small "in the back". Consequently any value which is fit for writing in the current run will sit at the front of the heap. This can be achieved by modifying the ordering relation which governs the heap.

An important thing to remember is that the "problematic" values also use up space in central memory. Because the size of the heap remains fixed and only the values fit for writing to the current run are deleted, eventually the heap will only contain those values which were too small. When this happens there is nothing else we can do but to end the current run and start a new one in which the problematic values are written. We can start by adapting the solution of the previous exercise by making the necessary changes to `distribute!`. Keep track of the last-written value in the current run. Change the order relation of the heap such that it stores those values which are smaller than the last written value of the current run in the back of the heap (in order). The others are stored at the front (also in order).

It can be shown mathematically that this improved replacement selection approach yields initial runs which are — on average — twice the size of the fixed-length runs generated by the original distribution technique[6].

[6] Again you may want to verify this by adding some statistics code to your solution

Chapter 16

Data Files

In this chapter we begin our study of external data structures, i.e. data structures that live on a disk. Just like internal data structures are constructed by using `cons` and `make-vector` as "data glue", we will be using blocks and block pointers as glue in order to group together values as an external data structure. In section 1.2.4 we have defined storage data structures as data structures the raison d'être of which it is to store data for later retrieval. In section 1.2.5 we have defined the `dictionary` ADT as the common abstraction for all such storage data structures. We have explained that dictionaries consist of a collection of *fields* some of which are key fields and others of which are satellite fields. In the terminology used by external data structures these collections of fields are known as *tuples* and tuples are said to *populate* an external data structure. In Scheme, a tuple is just a list of atomic data values. E.g., in an external data structure that has to store people's names along with their age and sex, (`list` "Allysa P." "Hacker" 23 'female) might constitute a valid tuple. Tuples have their origin in the relational database model. We briefly review the model in section 16.1. We start out by reviewing the relational model in section 16.1. We describe a tiny little subset of the relational model. It will be our goal to implement that subset in Scheme.

Our working horse will be the `table` ADT which forms the central abstraction for our implementing collections of tuples. This chapter discusses techniques to implement this ADT as a particular type of file, known as a *data file*. Remember from chapter 14 that a file is actually just a bunch of disk blocks that are linked up in some way. In section 16.2, we present the auxiliary ADTs that are needed to support our implementation. Section 16.3 discusses one particular implementation of the `table` ADT in great detail: it will turn out to be the case that a double linked list of blocks is extremely well-suited for representing data files. In section 16.4, we present performance characteristics for the implementation and we discuss some alternative representations together with their performance characteristics.

In our study of internal data structures (such as the trees of chapter 6 and the hash tables of chapter 7), `find!` was one of the central foci of attention.

This is not the case for data files. The reason is that the relational model stores tuples instead of key-value pairs. This means that there is no tuple field that is predefined as *the* key of a tuple. The field on which searching is done depends on the specific usage of `find!`. E.g., if we store people with their age and sex, we can search a data file given a name, given a numeric value (i.e. the age) or given a sex. In order to keep the search as fast as possible, techniques similar to trees and hashing will need to be developed. However, since various search criteria (i.e. name, age or sex) can be used, it is easier to store the information needed for executing the search in separate files (e.g. one per criterion), called *index files*. Index files are the topic of chapter 17.

16.1 Introduction: The Relational Model

External data structures are typically aligned with relational databases. In the relational model, we think of a *database* as a number of *tables*. Every table contains a number of columns and the name of the column is called an *attribute*. The attributes together form the *schema* of the table. As soon as we start filling up a table with rows, we are said to populate the table with *tuples*. Every tuple consists of a number of *fields*. In this section, we explain how we plan to implement these concepts in our tiny subset of the relational model.

16.1.1 The Schema

Let us start with an example. Our goal is to make a small database `solar-system` that contains two tables: `planets` and `moons`. In what follows, we assume that we have a formatted disk (see section 14.4) called `my-computer`. Our first step consists of creating the actual database. This is accomplished by calling the constructor `new` of the `database` ADT that is presented in section 16.1.4. We assume that the ADT was imported with the prefix `db`. The constructor is called with a formatted disk and a string that indicates the name of the database.

```
(define solar-system (db:new my-computer "Solar System"))
```

Next we add the two tables to the database. The first table is the `planets` table in which we store information about the planets of our solar system. Before we can start populating that table by adding tuples representing the planets, we first need to construct the table using the `create-table` procedure. It takes a database, the name of the newly created table and the schema of that table. The schema of the `planets` table is depicted in figure 16.1. In Scheme, we represent it by means of a simple association list `planets-schema` which enumerates the data *type* of every field (i.e. `decimal`, `integer`, `natural` or `string`) as well as the number of bytes that will be used to store the corresponding values. The type `natural` corresponds to positive integer numbers. `integer` refers to signed integer values. `decimal` is the type used for real numbers. Just like in chapter 14, we use a default of 8 bytes to store real numbers.

The `planets-schema` shown below stipulates that every planet tuple consists of 6 fields: a string of 9 characters, two decimal values, a natural number that

uses 3 bytes and two more decimals. In comments, we indicate the meaning of the corresponding tuple fields.

```
(define planets (db:create-table solar-system "Planets" planets-schema))

(define planets-schema '((string  9) ; name of the planet
                         (decimal)   ; distance to the sun
                         (decimal)   ; earth-masses
                         (natural 3); diameter
                         (decimal)   ; orbital time in earth years
                         (decimal)   ; rotational time in earth days
                         ))
```

planet-name	dist-to-sun	earth-mass	planet-diameter	orb-time	rot-time

Figure 16.1: The "planets" Schema

In what follows, we refer to the columns of the `planets` table using the following Scheme identifiers. Using `:earth-mass:` for instance, we can say "the third column of the table".

```
(define :planet-name:     0)
(define :distance-to-sun: 1)
(define :earth-mass:      2)
(define :planet-diameter: 3)
(define :orbital-time:    4)
(define :rotation-time:   5)
```

Similarly, we can define the schema for the second table in our database that stores the moons of the planets. It is depicted in figure 16.2 and it is defined in Scheme as follows. After executing this code, our `solar-system` database consists of two tables. For the moment, these tables are still empty because no tuples have been added yet.

```
(define moons (db:create-table solar-system "Moons" moons-schema))

(define moons-schema  '((string  9) ; name of the moon
                        (string  9) ; name of its planet
                        (natural 2) ; diameter
                        (natural 2) ; year-of-discovery
                        (string 10) ; discoverer
                        ))
```

Again, we can define 5 Scheme constants (such as `:diameter:`) that allow us to refer to the individual columns of the table. We omit the definitions because of space limitations.

moon-name	planet	moon-diameter	year-of-discovery	discoverer

Figure 16.2: The "moons" Schema

16.1.2 Tuples

Now that we have defined the schemas and the tables that go with these schemas, we can start populating the tables. This is achieved by inserting *tuples* in the tables. Tuples form the individual rows of a populated table. Each tuple can be thought of as a list of atomic Scheme values. Every value of the tuple goes into one column. Notice that the schema of a table is fixed during the entire lifetime of a table[1]. However, the population of a table naturally varies over time: tuples can be added and deleted from the table while a database system is running. Figure 16.3 shows a population for our `planets` table.

planet-name	dist-to-sun	earth-mass	planet-diameter	orb-time	rot-time
Mercury	0.3871	0.053	4840	0.241	+58.79
Venus	0.7233	0.815	12200	0.615	-243.68
Earth	1.0000	1.000	12756	1.000	+1.00
Mars	1.5237	0.109	6790	1.881	+1.03
Jupiter	5.2028	317.900	142800	11.862	+0.41
Saturn	9.5388	95.100	119300	29.458	+0.43
Uranus	19.1819	14.500	47100	84.013	-0.45
Neptune	30.0578	17.500	44800	164.793	+0.63

Figure 16.3: The "planets" table

In our Scheme implementation of the `database` ADT, we use the `insert-into-table!` procedure in order to accomplish this.

```
(db:insert-into-table! solar-system planets
                       (list "Mercury"  0.3871   0.053   4840   0.241  +58.79))
(db:insert-into-table! solar-system planets
                       (list "Venus"    0.7233   0.815  12200   0.615 −243.68))
(db:insert-into-table! solar-system planets
                       (list "Earth"    1.0000   1.000  12756   1.000   +1.00))
(db:insert-into-table! solar-system planets
                       (list "Mars"     1.5237   0.109   6790   1.881   +1.03))
(db:insert-into-table! solar-system planets
                       (list "Jupiter"  5.2028 317.900 142800  11.862   +0.41))
(db:insert-into-table! solar-system planets
```

[1] Changing the schema of a populated table is a very challenging problem in the database research community and we explicitly do not support it in this book.

```
                    (list "Saturn"   9.5388  95.100 119300  29.458   +0.43))
(db:insert-into-table! solar-system planets
                    (list "Uranus"  19.1819  14.500  47100  84.013   −0.45))
(db:insert-into-table! solar-system planets
                    (list "Neptune" 30.0578  17.500  44800 164.793   +0.63))
(db:insert-into-table! solar-system planets
                    (list "Pluto"   39.2975   1.000   5000 248.430   +0.26))
```

Obviously, we can do a similar exercise for our `moons` table.

Deleting a tuple from a table is achieved by means of the `delete-where!` procedure. The following code snipped exemplifies its usage. Notice that executing this procedure involves two phases. First the tuple has to be searched for in the data file representing the table. Second the located tuple needs to be removed from that file.

```
(db:delete-where! solar-system moons :planet: "Earth")
```

16.1.3 Queries

Apart from defining the machinery needed to define tables, their schema and their population, the relational model also provides us with a way for retrieving information from a (number of) table(s). This is accomplished by SQL's famous "SELECT" statement which we have also implemented in our Scheme subset in the form of a `select-from/eq` procedure. It is used as follows:

```
(db:select-from/eq solar-system moons :discoverer: "Cassini"))
```

This kind of query is known as a *equality query* because it queries the database (more precisely, the table `moons` of the database) for all the tuples for which the value in the `:discoverer:` column (i.e. the 5th field of the tuple) is equal to the string `"Cassini"`. You probably remember from your course on databases that it is also possible to query the database for all tuples that have some column value that is greater or smaller than a given value. Such queries are called *range queries* since they return all the tuples for which that column value fall in a certain range. We discuss range queries later.

16.1.4 The Database ADT

We finish section 16.1 by summarising the SQL-like operations that have been exemplified in the previous paragraphs. Together they form the `database` ADT shown below. Its implementation is the goal of this chapter and chapter 17.

```
1 ADT database
2
3 new
4     ( disk string → database )
5 delete!
6     ( database → ∅ )
```

```
7  create−table
8      ( database string pair → table )
9  create−index!
10     ( database table string number → ∅ )
11 drop−table!
12     ( database table → ∅ )
13 insert−into−table!
14     ( database table pair → ∅ )
15 delete−where!
16     ( database table number any → ∅ )
17 select−from/eq
18     ( database table number any → pair )
```

`new` creates a new database, given a disk and a file name. `delete!` removes an entire database from its disk. `create-table!` creates a new table, given a database, a file name and a list containing the schema for the newly created table (e.g. `moons-schema` explained above). `create-index!` can be ignored for now; it will be explained in chapter 17. `drop-table` removes a table (and all its tuples) from a database.

Tuples can be manipulated using `insert-into-table!`, `delete-where!` and `select-from/eq`. The former two have been explained before. `select-from/eq` takes a database, a table of that database, an attribute number and a value (representing the key to be searched for). It returns a list of tuples for which the attribute value is equal to that value.

16.2 Abstractions

A database implementation has to store the tuples of each table in some file which is — by definition — a collection of blocks that happens to be linked up in some clever way. Let us think about the `select-from/eq` procedure for a while. It allows us to query a table for all tuples that meet the search criterion. Surely, we want to avoid searching the entire data file because this requires us to read every single disk block of the file into central memory. In order to limit the number of blocks to be read upon querying a table, we will have to invent a clever system of pointers that can lead us through the blocks to the desired tuples in an efficiently manner. One approach could be to store those pointers in the data file itself, i.e. in the same disk blocks as the ones that contain the actual tuples. However, this soon leads to very complex data structures since we want to be able to search efficiently on *any* attribute value of the table. In other words, very different ways of navigating through the tuples would be mingled into the same data file.

The typical solution to this problem consists of storing a database as a *collection* of separate files. For every table, we have one *data file* that stores the actual tuples and a set of *index files* that store the navigational pointers to quickly locate the tuples, given a search key. Instead of creating such an index

file only for columns of a table, the idea will be to create an index file for every column for which fast searching is needed. Hence, an index file will be created for every attribute that is subject to frequent applications of `select-from/eq`. In this chapter, we study the organization of data files. Chapter 17 discusses index files in great detail.

16.2.1 Record Identifiers

Let us consider the organization of the data file. Every disk block in the file will typically store several[2] tuples, one after the other. The number of tuples that fit in one block depends on the size of the tuples: roughly spoken, the number of tuples equals the block size divided by the tuple size. We say that the blocks are organized into *slots*. A slot is either available or stores a tuple. In order to locate a tuple in a table, we thus need the block number of the block that stores the tuple, as well as the slot number of the tuple inside the block (e.g. 0 for the first tuple, 1 for the second tuple and so on). In other words, the whereabouts of a tuple are completely determined by means of two numbers. We will call such a couple a *record identifier*, or rcid for short. Obviously the slot numbers will need to be mapped onto an offset (i.e. a particular byte number) in the disk block. However, instead of polluting our algorithms with these low level details, we prefer to think in terms of slot numbers and hide that mapping behind an abstraction.

Record identifiers are implemented using the following Scheme definitions. We typically import them using the `rcid` prefix. `new` creates an rcid given a block number and the slot number that designates a tuple inside that block. `bptr` and `slot` are accessors to access both constituents.

```
(define (new bptr slot)
  (cons bptr slot))

(define bptr car)

(define slot cdr)
```

A rcid is represented as a Scheme pair. In our database implementation, we will be encoding an rcid in blocks because we want index files to refer to a tuple sitting in a data file. Hence, we need the machinery to encode and decode an rcid in a disk block as a number that can be further encoded into a series of bytes. This is the role of `rcid->fixed` and `fixed->rcid`. `rcid->fixed` converts a pair into a positive number. `fixed->rcid` does the exact opposite. `block-idx-size` is a constant of chapter 14 that contains the number of bytes that are required to encode an index (and thus also a slot number) inside a block. That is why we take the block pointer, multiply it by $256^{\texttt{block-idx-size}}$ and then add the slot number. `div` and `mod` are used to decode a "flat" positive

[2] We assume that all tuples of a table are equally big. Moreover, we assume that the block size exceeds the tuple size. Adapting the algorithms for tuples that can span multiple blocks is a non-trivial programming exercise.

number into two numbers in order to reconstruct the original rcid. `size` is the number of bytes needed to encode one rcid in a block.

```
(define size (+ disk:block-ptr-size disk:block-idx-size))

(define (rcid->fixed rcid)
  (+ (* (expt 256 disk:block-idx-size) (bptr rcid)) (slot rcid)))

(define (fixed->rcid num)
  (define radx (expt 256 disk:block-idx-size))
  (new (div num radx) (mod num radx)))
```

An `rcid` can be considered as a two dimensional pointer that consists of two pointers, one pointing to a disk block and another one pointing to a slot inside that block. The following definitions define a corresponding `null` pointer (i.e. $(0, 0)$) and a `null?` test that allows us to verify whether or not a given `rcid` is `null`.

```
(define null (new fs:null-block 0))

(define (null? rcid)
  (and (fs:null-block? (bptr rcid))
       (= (slot rcid) 0)))
```

In brief, rcids constitute a type of pointers that allows an index file to refer to a tuple in a data file. In general, multiple index files can refer to a tuple in a data file. Obviously, all of them will be using the same rcid. A corollary of this architecture is that it is very inconvenient to move tuples between blocks in a data file. Since this changes the whereabouts of a tuple, the tuple needs to be designated by an updated rcid. But then we have to scan all the index files in order to update the corresponding rcids. Obviously, this is a very costly operation.

16.2.2 The Table ADT

We have made a distinction between data files storing tuples and index files storing `rcid`s. We therefore study two ADTs. The `table` ADT represents a data file. In chapter 17 we study the `b-tree` ADT which represents the index files. Hence, every database consists of a number of `table`s and a number of `b-tree`s that live on some disk. The operations of the `table` ADT and the `b-tree` ADT are later used to implement the `database` operations such as `create-table`, `insert-into-table!` and `select-from/eq`.

Here is the `table` ADT. As explained, a table is just an unordered collection of tuples. Tuples are identified by their `rcid`. Thus, `insert!` takes a tuple (i.e. the `pair` argument) and it returns an `rcid`. The rcid indicates the whereabouts of the newly added tuple. `delete!` takes such an `rcid` and removes the associated tuple from the data file. We discuss the other operations in section 16.3 after we have introduced some additional terminology.

```
1  ADT table
2
3  new
4      ( disk string pair node → table )
5  open
6      ( disk string → table )
7  close!
8      ( table → ∅ )
9  table?
10     ( any → boolean )
11 drop!
12     ( table → ∅ )
13 schema
14     ( table → schema )
15 disk
16     ( table → disk )
17 set−current−to−first!
18     ( table → status )
19 set−current−to−next!
20     ( table → status )
21 current
22     ( table → rcid ∪{no-current} )
23 current!
24     ( table rcid → ∅ )
25 insert!
26     ( table pair → rcid )
27 delete!
28     ( table rcid → ∅ )
29 peek
30     ( table → pair )
```

Notice that some of the operations return a value of type `status`. Its constant values are enumerated below. For the time being, we only use `done` and `no-current`. The other values will become clear in chapter 17.

`status` = { `done`, `no-current`, `next-higher`, `duplicate`, `not-found` }

16.2.3 The Schema ADT

Figure 16.4 illustrates the file architecture of `table`s. The implementation of the `table` ADT relies on two auxiliary ADTs: the `schema` ADT and the `node` ADT. The idea is to implement a `table` as one single disk block (called a *header*) that refers to another disk block corresponding to a `schema`. Remember that a table's schema defines the type of the attributes as well as the size of each attribute. The disk block storing the schema information thus allow us to correctly interpret the tuples from the data file. Apart from referring to a `schema`, the disk block

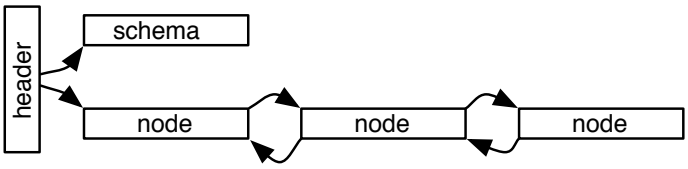

Figure 16.4: Representation of a Data File

corresponding to a `table` also refers to a number of `node`s that contain the actual tuples. A `node` is just an abstraction built atop disk blocks. It allows us to reason in terms of slots instead of raw byte offsets. As we will see in a moment, the nodes are linked up as a double linked list.

The `schema` ADT is presented below. It is used by the implementation of the `table` ADT. The schema information of a table is passed on to the constructor of the `schema` ADT. It is exemplified by the aforementioned `planets-schema`. This explains the second argument of the `new` constructor of the `schema` ADT. After constructing a schema, we can use `disk` and `position` to return the block number of the schema and the disk on which it is stored. `open` opens an existing `schema` given its block number on the disk. `delete!` removes the disk block that represents the schema by returning it to the file system.

```
1  ADT schema
2
3  new
4      ( disk pair → schema )
5  open
6      ( disk number → schema )
7  schema?
8      ( any → boolean )
9  delete!
10     ( schema → ∅ )
11 nr−of−attributes
12     ( schema → number )
13 nr−of−occupancy−bytes
14     ( schema → number )
15 record−size
16     ( schema → number )
17 type
18     ( schema number → byte )
19 size
20     ( schema number → byte )
21 capacity
22     ( schema → number )
```

```
23 disk
24     ( schema → disk )
25 position
26     ( schema → number )
```

The implementation of the `schema` ADT has to compute the total number of bytes that are needed to store a tuple that satisfies the schema. It does this by adding all the size information that is contained in the association list of the schema. After doing so, it can be retrieved from the `schema` using the `record-size` accessor. Based on this, we can derive the total number of tuples that fit in one disk node. This explains the `capacity` accessor. `nr-of-attributes` is the number of columns of the schema. In other words, it is the number of associations in the association list that was originally given to `new`. `nr-of-occupancy-bytes` is explained later.

The following code excerpt shows a transcript with some experiments that illustrate the `schema` ADT. We use the `moons-schema` of section 16.1. The example also illustrates the meaning of the `type` and `size` accessors. The type of the 4th attribute is 3 which will correspond to the `string` type and the size of the 4th attribute is 10 which is the number specified in the schema. The ADT has calculated the sum of the sizes of all the attributes and that sum turns out to be 32. Hence, every tuple in our `planets` table will require 32 bytes of disk storage. Based on this sum, we can calculate the capacity of a disk block. With an arbitrarily chosen `block-size` (see chapter 14) of 500, this happens to be 15.

```
> (define d (disk:new "Harddisk"))
> (fs:format! d)
> (define scma (new d moons-schema))
> (position scma)
1
> (disk scma)
#<disk>
> (capacity scma)
15
> (type scma 4)
3
> (size scma 4)
10
> (nr-of-attributes scma)
5
> (nr-of-occupancy-bytes scma)
2
> (record-size scma)
32
```

16.2.4 The Node ADT

As mentioned before, every data file internally consists of a number of disk blocks that are linked together as a double linked list. Every such block stores a number of tuples depending on the capacity calculated by the `schema` of the data file. Instead of implementing all the operations of the `table` ADT directly on top of the low level machinery of disk blocks, we can simplify our life by first defining a new `node` abstraction on top of the `block` ADT of chapter 14.

```
1  ADT node
2
3  new
4      ( schema node → node )
5  node?
6      ( any → boolean )
7  delete!
8      ( node → ∅ )
9  read
10     ( schema number → node )
11 write!
12     ( node → ∅ )
13 schema
14     ( node → schema )
15 position
16     ( node → number )
17 record!
18     ( node number pair → ∅ )
19 record
20     ( node number → pair )
21 occupy−slot!
22     ( node number → ∅ )
23 clear−slot!
24     ( node number → ∅ )
25 slot−occupied?
26     ( node number → boolean )
27 all−free?
28     ( node  → boolean )
29 all−occupied?
30     ( node  → boolean )
31 next
32     ( node → number )
33 next!
34     ( node number → ∅ )
35 previous
36     ( node → number )
37 previous!
38     ( node number → ∅ )
```

A node is conceived as a double linked disk block. i.e. a disk block in which some of the bytes are used to refer to a "next" disk block and in which some are used to refer to a "previous" disk block. The constructor `new` takes a schema and an existing node. The schema will be used to correctly calculate the offset of the tuple slots in the associated disk block. However, when using `write!` to write the node to the disk, the schema is not saved along with the node: a node on the disk *is* a disk block. That is because the schema is a disk block is shared by all nodes in the table. It is therefore saved only once as part of the table and not as part of every single node. The second argument of `new` is the next node of the newly created node. For the first call, () serves as an empty node. `read` is an alternative constructor: given a schema and a block number, it reads the corresponding block from the disk and it returns that block disguised as a node. `delete!` simply deletes the underlying disk block by returning it to the file system. `position` returns the block number of the node and `schema` returns the schema that was passed at construction time.

The other operations are more interesting. `next`, `next!`, `previous` and `previous!` can be used to manipulate the node as a member of a double linked list of nodes. Conceptually, a node is just a collection of numbered slots (which are obviously mapped onto some offset in the underlying byte vector). Every slot can store one tuple. The number of slots that a node can host is determined by the schema that was passed to the constructor. `record!` encodes a tuple (i.e. a list of values) in a node at a given slot number (counting from zero). `record` is its inverse. A node maintains the administrative information that is necessary to know which slots are occupied by a tuple and which are not. Basically, it consists of a bit for every slot. `occupy-slot!`, `clear-slot!` and `slot-occupied?` can be used to access and mutate this administrative bit.

16.2.5 Summary

In this section, we have presented four auxiliary abstractions that lie at the heart of our implementation of in the relational model. A `table` is the ADT that represents a data file storing tuples. Every tuple resides in some disk block. `node` is the abstraction that allows us to reason about disk blocks as double linked collections of slots. Hence, every tuple resides in a certain slot number in a node that corresponds to a disk block. This two dimensional pointer information is grouped together in a so-called `rcid`. Nodes can be thought of as disk blocks in disguise: every operation is translated into lower level operations that directly manipulate the bytes of the underlying block. In order to correctly interpret those bytes, we need a `schema` which stores the information of the table's columns. A schema holds the key to the correct interpretation of a disk block as a node.

16.3 Implementions

Let us now implement these abstractions. We start by looking at how the `schema` information is stored in a disk block. Then we explain the implementation of the `node` ADT because it is the fundamental building block for constructing the double linked list that makes up the `table` implementation.

16.3.1 Duality of Representation

It is important to understand that all the constituents that we discuss actually have a dual life. E.g., when implementing the `schema` ADT, we will be representing a schema as a Scheme data structure that resides in central memory. At the same time however, there exists a "copy" of the data structure on the disk. In the case of the `schema` ADT, this corresponds to just one disk block. The point however is that some of our ADTs only have a *partial* representation in central memory. For instance, the `table` implementation will maintain only one single disk block in central memory (the header disk block). All other blocks that make up the data structure only exist on the disk unless they are explicitly read from the disk and temporarily is copied in main memory. Needless to say, it is our job as the implementors of the ADTs to perform the necessary applications of `read` and `write!` in order to make sure that both "copies" of the data structure are consistent.

16.3.2 The Schema

So let us start by implementing the `schema` ADT. We begin by defining some constants.

```
(define schema-nr-size 2)

(define fixed-header-size (* 2 disk:block-ptr-size))
```

`schema-nr-size` is the number of bytes that are needed to encode numerical values in the schema block (such as the capacity of a block). `fixed-header-size` is the number of bytes that we need for encoding the next pointer and the previous pointer of a `node`. This number has to be subtracted from the block size in order to compute the number of bytes available for storing the actual tuples in a `node`.

On the disk, the schema is represented as a single disk block that stores the number of tuples that fit in a node (called the `capacity`), the size of one single tuple (called the record size `rsz`), the number of components that every tuple consists of (called the `nr-of-attributes`) and the type as well as the size (i.e. the number of bytes) of every single attribute. The outline of a `schema` is illustrated in figure 16.5. The figure illustrates that the characteristics of the attributes are split up in a pairwise fashion: all the types are located on the left, and their corresponding sizes are stored on the right. We use a single byte to encode the type of the data: at the time of writing, we support decimal

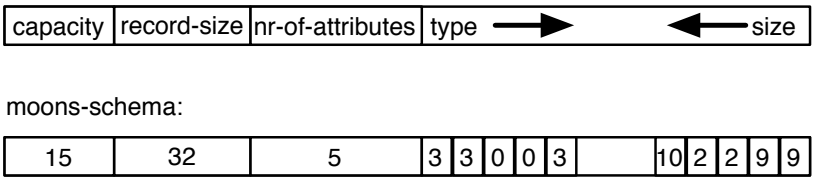

Figure 16.5: The Schema Block Layout Exemplified by `moons-schema`

numbers, natural numbers, integer numbers and strings. Notice that the type tags used are identical to the ones we used in section 14.4.2 for encoding data on a sequential file.

```
(define natural-tag 0)
(define integer-tag 1)
(define decimal-tag 2)
(define string-tag  3)
```

String lengths are restricted to 255 which allows us to encode the size of the data in a single byte as well.

As explained, the `schema` ADT has a dual representation. The following definitions are used to construct a Scheme record data structure in central memory that corresponds to the block layout described further below. `block` is an accessor to retrieve the single disk block that contains the schema. `disk` and `position` are operations on the schema that can be directly relayed to the `disk` ADT.

```
(define-record-type schema
  (make b)
  schema?
  (b block))

(define (disk scma)
  (disk:disk (block scma)))

(define (position scma)
  (disk:position (block scma)))
```

The following definitions define the disk block layout corresponding to such a schema. All the procedures accept a schema `scma` as the first parameter. All the procedures encode or decode the information in the schema's disk block by using the low level encoder and decoder procedures. The schema stores the capacity of nodes, the record size and the type and size of every attribute of the schema. Of particular interest are `type`, `type!`, `size` and `size!`. These procedures calculate the offset in the block and subsequently encode or decode the information at that particular offset. Notice that the type information is

stored on the left hand side (indicated by addition) whereas the size information is stored on the right hand side (indicated by subtraction).

```
(define cap-offset       0)
(define rsz-offset       (+ cap-offset schema-nr-size))
(define schema-size-offset (+ rsz-offset schema-nr-size))
(define schema-info-offset (+ schema-size-offset 1)) ; size is only one byte

(define (capacity! scma nmbr)
  (define blck (block scma))
  (disk:encode-fixed-natural! blck cap-offset schema-nr-size nmbr))
(define (capacity scma)
  (define blck (block scma))
  (disk:decode-fixed-natural blck cap-offset schema-nr-size))

(define (record-size! scma nmbr)
  (define blck (block scma))
  (disk:encode-fixed-natural! blck rsz-offset schema-nr-size nmbr))
(define (record-size scma)
  (define blck (block scma))
  (disk:decode-fixed-natural blck rsz-offset schema-nr-size))

(define (nr-of-attributes scma)
  (define blck (block scma))
  (disk:decode-byte blck schema-size-offset))
(define (nr-of-attributes! scma nmbr)
  (define blck (block scma))
  (disk:encode-byte! blck schema-size-offset nmbr))

(define (type! scma indx type)
  (define blck (block scma))
  (disk:encode-byte! blck (+ schema-info-offset indx)  type))
(define (type scma indx)
  (define blck (block scma))
  (disk:decode-byte blck (+ schema-info-offset indx)))

(define (size! scma indx nmbr)
  (define blck (block scma))
  (disk:encode-byte! blck (- disk:block-size 1 indx)  nmbr))
(define (size scma indx)
  (define blck (block scma))
  (disk:decode-byte blck (- disk:block-size 1 indx)))
```

Let us have a look at the implementation of `new` shown below. It receives a disk and an association list. The association list is the schema information that was exemplified by the `planets-schema` definition from section 16.1.

```
(define (new dsk atts)
```

```
(define rsz (sum-of-sizes atts))
(define cap (div (* (- disk:block-size fixed-header-size) 8)
                 (+ (* rsz 8) 1)))
(if (< cap 1)
  (error "tuples must fit in a block (new schema)" rsz))
(let* ((blck (fs:new-block dsk))
       (scma (make blck)))
  (capacity!       scma cap)
  (record-size!    scma rsz)
  (nr-of-attributes! scma (length atts))
  (types/sizes!    scma atts)
  (disk:write-block! blck)
  scma))
```

The first step is to add all the sizes of the attributes in order to calculate `rsz`, the total number of bytes needed to store a tuple. This is taken care of by the `sum-of-sizes` procedure. It traverses the association list in order to add up all the field sizes of every single attribute. The latter is extracted from each attribute by `field-size`.

```
(define (sum-of-sizes atts)
  (let loop
    ((atts atts)
     (sizs 0))
    (if (null? atts)
        sizs
        (loop (cdr atts) (+ (field-size (car atts)) sizs)))))

(define (field-size fild)
  (cond ((eq? (car fild) 'natural)
         (cadr fild))
        ((eq? (car fild) 'integer)
         (cadr fild))
        ((eq? (car fild) 'decimal)
         disk:real64)
        ((eq? (car fild) 'string)
         (cadr fild))
        (else (error "unsported field type" (car fild)))))
```

We continue explaining the constructor. The number of records that fit in a block (called the capacity `cap`) is roughly calculated by dividing the size of a block (minus the bytes that are occupied to store the next and previous pointers) by the total record size. However, remember from the `node` ADT that every node stores one administrative bit per slot. These bits allow us to keep track of which slots are occupied by a tuple and which are not. The number of bits needed equals the number of tuples that fit in a block. E.g., if we have 7 tuples in a block, then we will need 7 bits which can be hosted by 1 byte.

If we have 13 tuples per block, we need 13 bits which requires 2 bytes. This means that a certain number of bytes of a node are reserved for storing these *occupancy bits* which may affect the number of tuples that fit in a block in its turn. In order to resolve this circular dependency, `new` makes the overall calculation in terms of bits. We need `rsz` times 8 bits to store a tuple, plus 1 bit for storing its occupancy bit. We have a total of (∗ (− `disk:block-size` `fixed-header-size`) 8) bits at our disposal (after subtracting the space needed for storing the next pointer and the previous pointer of a node). Hence, this means that we have to divide those numbers in order to calculate the total number of tuples that fit in one block. This explains the binding for the `cap` variable.

Given the capacity, it is trivial to calculate the number of bytes that are needed to store the occupancy bits. Here is the code:

```
(define (nr-of-occupancy-bytes schema)
  (define bits (capacity schema))
  (exact (ceiling (/ bits 8))))
```

The rest of the implementation of `new` is straightforward. A fresh block is allocated from the file system. Subsequently, all quantitative information is encoded in the block and the block is written back to the disk. Below, we include the implementation of the remaining procedures needed for the encoding. `types/sizes!` takes the entire table schema as an association list. It traverses the list and encodes the types and sizes of the schema in a pairwise fashion using `field-type` and `field-size`.

```
(define (field-type fild)
  (cond ((eq? (car fild) 'natural)
         natural-tag)
        ((eq? (car fild) 'decimal)
         decimal-tag)
        ((eq? (car fild) 'string)
         string-tag)
        (else (error "unsported field type" (car fild)))))

(define (types/sizes! scma atts)
  (let loop
    ((indx 0)
     (atts atts))
    (type! scma indx (field-type (car atts)))
    (size! scma indx (field-size (car atts)))
    (if (not (null? (cdr atts)))
        (loop (+ indx 1) (cdr atts)))))
```

The remaining `schema` operations are straightforward translations of the underlying `disk` operations: a schema is read from disk by reading its block and constructing its dual in-memory representation. Deleting a schema simply consists of giving its disk block back to the file system. The Scheme garbage

collector is responsible for reclaiming the copy that lives in central memory. The code is trivial:

```
(define (open dsk bptr)
  (define blck (disk:read-block dsk bptr))
  (define scma (make blck))
  scma)

(define (delete! scma)
  (fs:delete-block (block scma)))
```

16.3.3 The Nodes

Nodes are the basic building blocks of a data file. Again, every node is a dually represented data structure. We start by discussing the representation of a node as a disk block. Having discussed quite a number of block layouts by now, the following code should be fairly easy to understand. Irrespective of the record size, every node contains a next pointer, a previous pointer and a number of bytes that hold the occupancy bits for the node's slots. These constituents are accessed and mutated by the following procedures.

```
(define next-offset 0)
(define prev-offset (+ next-offset disk:block-ptr-size))
(define bits-offset (+ prev-offset disk:block-ptr-size))

(define (next node)
  (define blck (block node))
  (disk:decode-fixed-natural blck next-offset disk:block-ptr-size))
(define (next! node next)
  (define blck (block node))
  (disk:encode-fixed-natural! blck next-offset disk:block-ptr-size next))
(define (previous node)
  (define blck (block node))
  (disk:decode-fixed-natural blck prev-offset disk:block-ptr-size))
(define (previous! node prev)
  (define blck (block node))
  (disk:encode-fixed-natural! blck prev-offset disk:block-ptr-size prev))
(define (occupancy-bits node)
  (define blck (block node))
  (define scma (schema node))
  (define byts (scma:nr-of-occupancy-bytes scma))
  (disk:decode-fixed-natural blck bits-offset byts))
(define (occupancy-bits! node bits)
  (define blck (block node))
  (define scma (schema node))
  (define byts (scma:nr-of-occupancy-bytes scma))
  (disk:encode-fixed-natural! blck bits-offset byts bits))
```

The dual representation of a node in central memory follows. Every node is represented as a record that stores a reference to a disk block and a `schema` that is needed to correctly interpret the bytes of the disk block. The constructor `new` takes two parameters, namely the schema and the next node of the node in the double linked list. All occupancy bits are set to 0 which means that none of the slots contains a meaningful tuple.

```
(define-record-type node
  (make s b)
  node?
  (s schema)
  (b block))

(define (new scma next)
  (define blck (fs:new-block (scma:disk scma)))
  (define node (make scma blck))
  (next!     node next)
  (previous! node fs:null-block)
  (occupancy-bits! node 0)
  node)
```

The relation between both representations is established by the following procedures. `read` takes a schema and block pointer. It reads the block from the disk and returns that block disguised as a `node`. `write!` takes a node and effectively writes its underlying block to the disk. The `position` of a node is the position of its underlying block. Deletion of a node merely consists of returning its block to the file system. The dual representation in central memory is garbaged collected by Scheme.

```
(define (read scma bptr)
  (define disk (scma:disk scma))
  (define blck (disk:read-block disk bptr))
  (make scma blck))

(define (write! node)
  (define blck (block node))
  (disk:write-block! blck))

(define (position node)
  (disk:position (block node)))

(define (delete! node)
  (fs:delete-block (block node)))
```

Next we look at the code for encoding and decoding tuples in nodes. Both the procedures `record` and `record!` take a node and a slot number `slot`. `record` decodes the tuple sitting in that particular slot of the node. `record!` encodes

a tuple in the slot. Remember that a tuple is just a list of values. Both procedures are implemented as a loop. `record!` iterates over the list of values `vals` and encodes them one by one. `record` does the exact opposite. Both start encoding (resp. decoding) the values starting at an offset that is calculated by `calc-record-offset`. That offset is found by skipping the fixed header (i.e. the next and the previous pointers) and the bytes that host the occupancy bits, and by multiplying the slot number with the size of a record (thereby skipping all preceding records).

```
(define (calc-record-offset node slot)
  (define scma (schema node))
  (define skip (+ scma:fixed-header-size (scma:nr-of-occupancy-bytes scma)))
  (define rsiz (scma:record-size scma))
  (+ skip (* rsiz slot)))

(define (record! node slot tupl)
  (define scma (schema node))
  (define blck (block node))
  (let loop
    ((cntr 0)
     (offs (calc-record-offset node slot))
     (vals tupl))
    (cond ((null? vals)
           (if (< cntr (scma:nr-of-attributes scma))
               (error "too few values in tuple" vals)))
          ((>= cntr (scma:nr-of-attributes scma))
           (if (not (null? vals))
               (error "too many values in tuple" vals)))
          (else ((vector-ref encoders (scma:type scma cntr))
                 blck offs (scma:size scma cntr) (car vals))
                (loop (+ cntr 1) (+ offs (scma:size scma cntr)) (cdr vals)))))
  (occupy-slot! node slot))

(define (record node slot)
  (define scma (schema node))
  (define blck (block node))
  (let loop
    ((cntr 0)
     (offs (calc-record-offset node slot)))
    (if (= cntr (scma:nr-of-attributes scma))
        ()
        (cons ((vector-ref decoders (scma:type scma cntr))
               blck offs (scma:size scma cntr))
              (loop (+ cntr 1) (+ offs (scma:size scma cntr)))))))
```

The magic of `record!` and `record` happens in the `else`-branch of their conditional. Both the encoding and decoding process is steered by two vectors

`encoders` and `decoders` that contain the encoding and decoding procedures for every supported type.

```
(define encoders  (vector disk:encode-fixed-natural!
                          disk:encode-arbitrary-integer!
                          disk:encode-real!
                          disk:encode-string!))
(define decoders  (vector disk:decode-fixed-natural
                          disk:decode-arbitrary-integer
                          disk:decode-real
                          disk:decode-string))
```

While iterating over the fields of the tuple to be encoded or decoded, the right procedures are selected from these vectors by using the type of the `cntr`'th attribute of the schema as index. This is achieved by the expression (`scma:type scma cntr`). The number of bytes used during the encoding or decoding is determined by the size (`scma:size scma cntr`) of the attribute.

Finally, we discuss the machinery that supports the management of free slots in a block. Remember that every record slot is assigned an administrative bit that allows us to verify whether or not the slot is occupied. `occupy-slot!` sets the bit corresponding to a slot number to 1. `clear-slot!` sets the bit to 0. `slot-occupied?` verifies whether or not the bit is set. `all-free?` checks whether or not *all* occupancy bits are cleared. If that is the case, the node is completely empty. Its counterpart is called `all-occupied?`.

```
(define (occupy-slot! node slot)
  (define bits (occupancy-bits node))
  (occupancy-bits! node (bitwise-ior bits (expt 2 slot))))


(define (clear-slot! node slot)
  (define bits (occupancy-bits node))
  (occupancy-bits! node (bitwise-and bits (bitwise-not (expt 2 slot)))))


(define (slot-occupied? node slot)
  (define bits (occupancy-bits node))
  (not (= (bitwise-and bits (expt 2 slot)) 0)))


(define (all-free? node)
  (define bits (occupancy-bits node))
  (= bits 0))


(define (all-occupied? node)
  (define bits (occupancy-bits node))
  (define all1 (- (expt 2 (scma:capacity (schema node))) 1))
  (= (bitwise-and bits all1) all1))
```

16.3.4 The Table

Now that we know how the schema and the individual nodes operate, we can tie everything together in an implementation of the `table` ADT. Again, it is a data structure that has a dual representation (albeit a partial one this time). We start by presenting the representation that resides in central memory. The following code shows how a table looks like in central memory. `make` constructs the record data structure which consists of a name of the corresponding file in the directory, a reference to a schema, a header block and a couple $(buffer, slot)$ that defines a "current". The couple consists of an in-memory buffer `node` and a slot number in that buffer. These two components will be used when implementing `set-current-to-first!` and `set-current-to-next!`. The in-memory buffer block prevents us from reading a disk node every time `set-current-to-next!` is called: as long as the buffer contains unvisited tuples, it stays in central memory and the counter is set to refer to the next tuple slot in the block.

```
(define-record-type table
  (make n h s b l)
  table?
  (n name)
  (h header header!)
  (s schema schema!)
  (b buffer buffer!)
  (l slot  slot!))
```

The dual representation of the `table` ADT basically consists of the header block since we have already discussed the schema. The header block is the actual "begin" of the data file. It is the block that is registered in the directory under the table's file name. In the header block, the pointers are encoded that refer to the schema and to the nodes that make up the double linked list for storing the tuples.

Architecture of the File

The double linked list of nodes is a bit special. It is depicted in figure 16.6 which is a more refined version of figure 16.4. The architecture is driven by the fact that we want to reconcile the following requirements: (1) we want one continuous data structure in order to ease the implementation of `set-current-to-next!`, (2) we want fast access to a node that has unoccupied slots in order to have a fast insertion and deletion algorithm. We therefore have two double linked lists. The `full` pointer refers to blocks that are completely full (i.e. nodes that have no unoccupied slots left). In the picture, these are the grey nodes. The `part` pointer refers to nodes that are only partially full, i.e. nodes that have some unoccupied slots. These are the nodes in white. Nodes are moved from the `part` list to the `full` list as soon as they become full after inserting a tuple. Nodes are moved from the `full` list to the `part` list when a tuple is deleted from a node that was entirely full. Notice that the nodes can be extracted from the list(s) in $O(1)$ because of the fact that it is double linked. We need a reference to the last node of the `full` list in order to make the connection to the first node of the

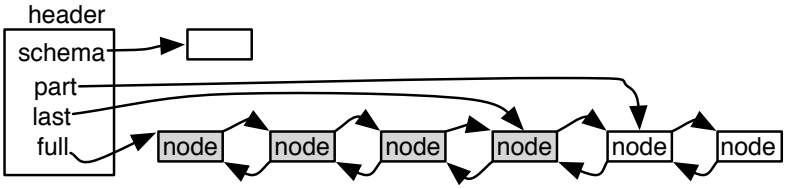

Figure 16.6: Detailed Representation of a Data File

`part` list. In the end, we have one double linked list that can be traversed with a very simple implementation of `set-current-to-next!` as explained below.

Here is how the header block looks like. It basically stores a reference to the head of both double linked lists, a reference to the last node of the linked list of full nodes and a `schema-ptr` that refers to the disk block containing the schema information of the table.

```
(define full-offset  0)
(define last-offset  (+ full-offset disk:block-ptr-size))
(define part-offset  (+ last-offset disk:block-ptr-size))
(define schema-offset (+ part-offset disk:block-ptr-size))

(define (full tble)
  (define hder (header tble))
  (disk:decode-fixed-natural hder full-offset disk:block-ptr-size))
(define (full! tble bptr)
  (define hder (header tble))
  (disk:encode-fixed-natural! hder full-offset disk:block-ptr-size bptr))

(define (last tble)
  (define hder (header tble))
  (disk:decode-fixed-natural hder last-offset disk:block-ptr-size))
(define (last! tble bptr)
  (define hder (header tble))
  (disk:encode-fixed-natural! hder last-offset disk:block-ptr-size bptr))

(define (part tble)
  (define hder (header tble))
  (disk:decode-fixed-natural hder part-offset disk:block-ptr-size))
(define (part! tble bptr)
  (define hder (header tble))
  (disk:encode-fixed-natural! hder part-offset disk:block-ptr-size bptr))

(define (schema-bptr tble)
  (define hder (header tble))
```

```
  (disk:decode-fixed-natural hder schema-offset disk:block-ptr-size))
(define (schema-bptr! tble bptr)
  (define hder (header tble))
  (disk:encode-fixed-natural! hder schema-offset disk:block-ptr-size bptr))
```

Creation and Destruction

Now that we have both representations of the `table` ADT, we move on to the implementation of its operations. Creating a table is the task of `new`. It receives a reference to a disk, the name of the newly created table and an association list that represents the schema of the table to be constructed. `new` calls `new` of the `scma` ADT in order to create the schema on the disk. Then the file system is asked a disk block `hder`. All its constituents are initialized to "null" except for the pointer that refers to the schema block. Then the header block is written back to the disk and registered in the directory.

```
(define (new disk name atts)
  (define scma (scma:new disk atts))
  (define hder (fs:new-block disk))
  (define tble (make name hder scma))
  (full! tble fs:null-block)
  (last! tble fs:null-block)
  (part! tble fs:null-block)
  (schema-ptr! tble (scma:position scma))
  (disk:write-block! hder)
  (fs:mk disk name (disk:position hder))
  tble)
```

`open!` locates the header in the directory. It reads the header from the disk and it also reads the schema from the disk. Just like in the implementation of `new`, an in-memory data structure is constructed using `make`.

```
(define (open disk name)
  (define hptr (fs:whereis disk name))
  (define hder (disk:read-block disk hptr))
  (define tble (make name hder ()))
  (define sptr (schema-ptr tble))
  (define scma (scma:open disk sptr))
  (schema! tble scma)
  tble)
```

`close!` closes a table by writing its header block back to the disk. After this, the Scheme data structure can be safely garbage collected.

```
(define (close! tble)
  (disk:write-block! (header tble)))
```

`drop!` deletes an entire table from the disk: the nodes are read one after the other and deleted (i.e. returned to the file system). After cleaning up all nodes, the schema and the header are deleted and the table is removed from the directory of its disk.

```
(define (drop! tble)
  (define scma (schema tble))
  (define hder (header tble))
  (define disk (scma:disk scma))
  (define (delete-nodes bptr)
    (if (not (fs:null-block? bptr))
        (let*
             ((node (node:read scma bptr))
              (next (node:next node)))
          (node:delete! node)
          (delete-nodes next))))
  (if (fs:null-block? (full tble))
      (delete-nodes (part tble))
      (delete-nodes (full tble)))
  (scma:delete! scma)
  (fs:delete-block hder)
  (fs:rm disk (name tble)))
```

Double Linked List Manipulation Procedures

In order to simplify the insertion and deletion of tuples, we implement a number of auxiliary procedures that allows us to insert and extract nodes from our double linked lists. `extract-node!` removes a node from the double linked list. It has to be called with `full` and `full!` (resp. `part` and `part!`) to correctly remove the node from the list of full (resp. partially full) nodes. The steps needed to remove a node from a doubly linked that lives on disk are entirely similar to what is needed in the implementation of a doubly linked list in central memory (see section 3.2.7).

```
(define (extract-node! tble first first! node)
  (define next-bptr (node:next node))
  (define prev-bptr (node:previous node))
  (node:next!     node fs:null-block)
  (node:previous! node fs:null-block)
  (if (not (fs:null-block? next-bptr))
      (let ((next (node:read (schema tble) next-bptr)))
        (node:previous! next prev-bptr)
        (node:write! next)))
  (if (not (fs:null-block? prev-bptr))
      (let ((prev (node:read (schema tble) prev-bptr)))
        (node:next! prev next-bptr)
        (node:write! prev)))
  (if (= (first tble) (node:position node))
      (first! tble next-bptr))
  (if (= (last tble) (node:position node))
      (last! tble prev-bptr)))
```

`insert-node!` inserts a node in the double linked list. It is called by `insert-part!` and `insert-full!` which make sure that the `last` pointer always

correctly refers to the last node of the `full` list.

```
(define (insert-node! tble first first! node)
  (define frst-bptr (first tble))
  (if (not (fs:null-block? frst-bptr))
      (let ((next (node:read (schema tble) frst-bptr)))
        (node:previous! next (node:position node))
        (node:next! node frst-bptr)
        (node:write! next)))
  (first! tble (node:position node)))

(define (insert-part! tble node)
  (define last-bptr (last tble))
  (insert-node! tble part part! node)
  (if (not (fs:null-block? last-bptr))
      (let ((prev (node:read (schema tble) last-bptr)))
        (node:next! prev (node:position node))
        (node:previous! node last-bptr)
        (node:write! prev))))

(define (insert-full! tble node)
  (define last-bptr (last tble))
  (insert-node! tble full full! node)
  (if (fs:null-block? last-bptr)
      (last! tble (node:position node))))
```

Insertion and Deletion

Given these auxiliary procedures, the following two procedures allow us to insert a tuple in a table and delete a tuple from a table. Remember that a tuple is identified by its `rcid`. `insert!` takes a tuple, inserts it in some free slot in some node of the data file. It returns the rcid that corresponds to the whereabouts of the tuple. `delete!` takes such an rcid as its argument. It erases the corresponding tuple from the corresponding node.

`insert!` checks whether the list of partially filled nodes still contains a node. If this is the case, that node is read into main memory. Otherwise, a new node is created and added to the list of partially filled nodes. Subsequently, the first free slot in the node is located and the tuple is encoded at that slot using `record!`. If encoding the tuple renders the node entirely full, then it is removed from the list of partially filled nodes and inserted in the list of nodes that are entirely full. The node and its slot number become the current using `buffer!` and `slot!`. The whereabouts of the newly added tuple (i.e. the rcid of the tuple) is returned from the procedure.

```
(define (insert! tble tupl)
  (define scma (schema tble))
  (define room (part tble))
  (define node (if (fs:null-block? room)
```

```
                    (let ((new (node:new scma fs:null-block)))
                      (insert-part! tble new)
                      new)
                    (node:read scma room)))
  (define free (find-free-slot node -1))
  (node:record! node free tupl)
  (when (node:all-occupied? node)
    (extract-node! tble part part! node)
    (insert-full!  tble node))
  (node:write!  node)
  (buffer!     tble node)
  (slot!      tble free)
  (rcid:new (node:position node) free))

(define (find-free-slot node strt)
  (define scma (node:schema node))
  (let loop
    ((cntr (+ strt 1)))
    (cond ((= (scma:capacity scma) cntr)
           -1)
          ((node:slot-occupied? node cntr)
           (loop (+ cntr 1)))
          (else
           cntr))))
```

`delete!` removes a tuple pointed to by its argument `rcid`. The `node` is read into main memory and the slot is cleared. If clearing the slots means that all the tuples now become free (which is verified by `all-free?`), then the node is removed from the right double linked lists and returned to the file system using `node:delete!`. Otherwise the updated node is simply written back to the disk. However, if the node was entirely full before the deletion, it has to be extracted from the list of entirely filled nodes and added to the list of partially filled nodes. `buffer!` and `slot!` are used to invalidate the current.

```
(define (delete! tble rcid)
  (define scma (schema tble))
  (define node (node:read scma (rcid:bptr rcid)))
  (define was-full? (node:all-occupied? node))
  (node:clear-slot! node (rcid:slot rcid))
  (cond ((node:all-free? node)
         (if was-full?
           (extract-node! tble full full! node)
           (extract-node! tble part part! node))
         (node:delete! node))
        (else
         (when was-full?
           (extract-node! tble full full! node)
```

```
          (insert-part! tble node))
        (node:write! node)))
  (buffer! tble ())
  (slot!  tble -1))
```

Managing the Current

The current is manipulated using `set-current-to-first!` and `set-current-to-next!`. `set-current-to-first!` reads the first node of the table (from either list) into central memory and makes it the current block. The current slot is set to point to the first slot in the block that is not empty. This slot is searched for using `find-occupied-slot` which starts testing the occupancy bits at `strt` and which searches the block using the good old linear searching algorithm. Notice that the $O(n)$ behavior of linear searching inside a node is negligible compared to the act of reading the node from the disk. `set-current-to-next!` tries to advance the current in the same buffer node. Should this fail (indicated by the fact that `find-occupied-slot` returns -1), then the next node is read into central memory and the current slot is set to point to the first occupied slot in the new block. The current is invalidated as soon as the end of the data file is reached.

```
(define (find-occupied-slot node strt)
  (define scma (node:schema node))
  (let loop
    ((cntr (+ strt 1)))
    (cond ((= (scma:capacity scma) cntr)
           -1)
          ((node:slot-occupied? node cntr)
           cntr)
          (else
           (loop (+ cntr 1))))))

(define (set-current-to-first! tble)
  (define scma (schema tble))
  (if (and (fs:null-block? (full tble))
           (fs:null-block? (part tble)))
    no-current
    (let* ((fptr (if (fs:null-block? (full tble))
                     (part tble)
                     (full tble)))
           (bffr (node:read scma fptr))
           (curr (find-occupied-slot bffr -1)))
      (buffer! tble bffr)
      (slot!  tble curr)
      done)))

(define (set-current-to-next! tble)
  (define scma (schema tble))
```

```
(define bffr (buffer tble))
(define curr (slot  tble))
(if (null? bffr)
    no-current
    (let ((indx (find-occupied-slot bffr curr)))
      (cond ((not (= indx -1))
             (buffer! tble bffr)
             (slot!  tble indx)
             done)
            ((not (fs:null-block? (node:next bffr)))
             (let* ((next (node:read scma (node:next bffr)))
                    (indx  (find-occupied-slot next -1)))
               (buffer! tble next)
               (slot!  tble indx)
               done))
            (else
             (buffer! tble ())
             (slot!  tble -1)
             no-current)))))
```

Notice that the `table` ADT specifies that `set-current-to-first!` and `set-current-to-next!` return a `status` value. In our implementations presented here, we observe that this can take the form of `done` or `no-current`. In chapter 17 we will further extend the list of possible `status` values.

Using the Current

The following operations are related to the current. `peek` returns the tuple that is being pointed to by the current. `current` returns the current rcid and `current!` sets the current to point to a given rcid. The corresponding block is read into central memory and the current slot is made to refer to the slot number comprised in the `rcid`. These operations are heavily used when implementing our SQL subset in chapter 17.

```
(define (peek tble)
  (define bffr (buffer tble))
  (define curr (slot tble))
  (if (null? bffr)
      no-current
      (node:record bffr curr)))

(define (current tble)
  (define bffr (buffer tble))
  (define curr (slot  tble))
  (if (null? curr)
      no-current
      (rcid:new (node:position bffr) curr)))

(define (current! tble rcid)
```

```
(define bffr (buffer tble))
(define curr (rcid:slot rcid))
(if (or (null? bffr)
        (not (= (rcid:bptr rcid) (node:position bffr))))
  (set! bffr (node:read (schema tble) (rcid:bptr rcid))))
(buffer! tble bffr)
(slot!   tble curr))
```

16.4 Performance Considerations

Surely, a double linked list of disk blocks is not the only possible way to organise a data file. Let us consider some alternatives and think of their performance characteristics. Before we start investigating the alternatives, we have to know the criteria that have to be used in order to compare them. As explained in section 16.1.4, we are interested in the following operations on a database. They are the key operations of the relational model.

Insertion and deletion Obviously, inserting a tuple in a database is an operation that we want to keep as efficient as possible. In practice, tuples are often inserted into databases by users that are filling out some kind of web form. By clicking 'ok', the information of the form is sent to the web server and inserted in a database. Even in large datasets (like e.g. the amazon.com database), we want this to happen in a reasonable amount of time. It can be argued that deletion happens less frequently than insertion. A slightly less efficient implementation of deletion is therefore slightly more acceptable than an inefficient insert.

Equality Queries Equality queries are those "find" operations that search a database for all tuples that have some column value that is *equal* to some given search value. In SQL, this can be done by using the "SELECT ... FROM ... WHERE ... = ..." statement with an equality operator. This is the equivalent of the `select-from/eq` operation explained in section 16.1.3.

Range Queries Range queries query the database for all tuples that have field that is greater than (resp. smaller than) some given value, or a field whose value lies between a couple of values. In SQL, range queries typically take the form "SELECT ... FROM ... WHERE ... > ..." or "SELECT ... FROM ... WHERE ... BETWEEN ... AND ...".

Let us have a look at how we might implement these SQL operations given the current implementation of our `table` ADT. Insertion and deletion is trivial. All we have to do is call the `insert!` and `delete!` operations as they appear in the ADT. Obviously, this requires $O(1)$ disk block transfers since the data file is implemented by means of a double linked list. Sometimes, a node is "moved" between two different linked lists. However, we never have to physically move

(i.e. copy) blocks or tuples across different blocks. Because of our system with occupancy bits, occupied and unoccupied slots can freely coexist without expensive storage moves. Equality queries and range queries are problematic. Suppose that we need B disk blocks to store all n tuples in a given application. Since our double linked list is not constructed according to any particular order, we have to traverse the entire table in order to collect *all* tuples that meet the condition of the query. Hence, in the worst case, both equality queries as well as range queries require a total of $O(B)$ block transfers. This explains the first row of the table in figure 16.7.

Representation	`insert!`	`delete!`	Equality Queries	Ranges Queries
Double Linked List	$O(1)$	$O(1)$	$O(B)$	$O(B)$
Sorted List	$O(B)$	$O(1)$ or $O(B)$	$O(B)$	$O(B)$
Hashing	$O(\alpha)$	$O(\alpha)$	$\Theta(1.3B)$	$\Theta(1.3B)$

Figure 16.7: Comparative Data File Organizations

An alternative implementation of the `table` ADT would be to store the tuples in sorted order. This does not really lead to anything useful. Respecting the order would require `insert!` to be in $O(B)$. Even if we could implement the actual insert in a fast way, we have to traverse the blocks to locate the right position. `delete!` could stay in $O(1)$ if we allow "holes" in the data file. However, if we require that the data occupies the least possible number of disk blocks, we have to move the tuples which brings `delete!` into $O(B)$ as well. An equality query is resolved by launching a `find` operation on the database. This makes the "current" of the table point to the first tuple that satisfies the equality test. From that point on, `set-current-to-next!` is used to keep on visiting the tuples as long as the search criterion is satisfied. Notice that — even though the data is sorted — we cannot use binary searching. This method is only applicable if we can calculate the middle element of the n elements in $O(1)$. Unfortunately, this is not the case: the data is scattered over B disk blocks and there is no guarantee whatsoever that these blocks are adjacent on our disk. Hence, we have to resort to plain linear searching which brings the queries in $O(B)$ as well. This explains the second row in figure 16.7.

Yet another alternative would be to use a hashing strategy in order to *calculate* the block location of a tuple and use some kind of collision resolution strategy if the block appears full. This brings `insert!` and `delete!` in the order of the load factor $O(\alpha)$ (which is fairly close to 1). It also brings `find!` in $O(\alpha)$. However, hashing has two drawbacks. First of all, we have to keep the load factor α below 75% which requires $O(1.3B)$ disk blocks. Second, we have to traverse the entire data file when resolving queries: since hashing stores the tuples in an unordered way, both equality queries and range queries require a total of $\Theta(1.3B)$ block transfers. These considerations are summarized in the last row of figure 16.7.

Fortunately there is a way out. In chapter 17 we present a special type of tree that are used to bring these numbers back into the realm of logarithms. However, instead of storing the tuples in a tree, we store their rcids in the tree. The double linked data file model presented in this chapter is still used in order to store the actual tuples. This model has very attractive performance characteristics for insertion and deletion. The tree takes care of searching.

16.5 Exercises

1. Answer the following questions.

   (a) What is the biggest natural number that can be stored in the `nr-of-attributes` field of the schema block?

   (b) What is the maximum number of different data types that we can potentially use for the attributes described by the schema? (The current implementation uses only 4.)

   (c) What it the maximum size (in bytes) for any tuple field described by the schema?

   The fact that we use only a single block to encode the schema of a table imposes a number of constraints on which schemas. What this in mind, try to answer the following questions:

   (d) What is the effective maximum number of attributes that a schema can contain and still fit in the schema block?

   (e) What will happen if a user tries to use a schema with more attributes than we can handle in the schema block (i.e. more than your answer for (d))?

   (f) Change the `new` procedure to make sure that this problem is avoided (by only accepting schemas with a number of attributes that fit).

2. Consider the implementations of `set-current-to-first!` and `set-current-to-next!`. These can be used to traverse the tuples of a data file. Image that we would like to be able to traverse the tuples in the reverse order[3]. In order to do so we will need a `set-current-to-last!` and a `set-current-to-previous!` procedure.

   `set-current-to-previous!` procedure is easily implemented because the nodes form a double-linked list. However, to implement `set-current-to-last!` we need to be able to find the last node in the list. To do this efficiently (i.e. without traversing the entire list of nodes) we need a pointer that always points to the end of the list. In cases where all nodes are full we can use the last pointer for this. There is however no pointer to the last

[3]Strictly speaking this is not useful because the order in which tuples are stored does not have any meaning or significance, so reversing the order is pointless.

node of the partially filled nodes. Therefore we will first to add this new pointer to the header block.

Follow the following implementation steps:

- Write the `set-current-to-previous!` procedure. Base your solution on `set-current-to-next!`. It is recommended that you implement a `find-last-occupied-slot` auxiliary procedure analogous to `find-occupied-slot`.
- Add a `lpar-offset` variable and `last2` and `last2!` procedures that write and read the pointer (stored in the header block) to the last partially filled node.
- To initialise and update the value of the new pointer we use the existing `insert-node!` and `extract-node!` procedures. Add `last` and `last!` parameters to both procedures. These will be used to pass either `last` and `last!` or `last` and `last2!` depending on whether a node is inserted into or extracted from the sublist of full nodes or the sublist of partially filled nodes. In the `extract-node!` procedure the code already deals with the new parameters. In `insert-node!` you will need to add this code (which updates the pointer to the last node of the sublist when needed) yourself. This means that the existing code for this purpose in `insert-full!` can be removed. Of course all calls to `insert-node!` or `extract-node!` will need to be updated in order to pass the right pair of procedures.
- Now you can implement `set-current-to-last!`. Base your solution on `set-current-to-first!`. Depending on whether or not the sublist of partially filled nodes contains nodes you will either use the `last2` or the `last` pointer to find the last node. You can reuse the `find-last-occupied-slot` auxiliary procedure.

Chapter 17

Index Files: B^+-Trees

Remember from the previous chapter that we store tuples in data files and that the whereabouts of a tuple is determined by its rcid. An `rcid` is just a couple of indices, the first one referring to a block number and the second one referring to a slot number within the corresponding disk block. Unfortunately, determining the rcid of a tuple is a time consuming operation. With the machinery of the previous chapter, we can only use the current to traverse the entire data file. This leads to an unacceptably high number of block transfers. Consider e.g. the query[1]:

```
(db:select-from/eq solar-system moons :discoverer: "Cassini"))
```

Remember from chapter 16 that a table's tuples are stored in no particular order. That is why we can keep insertion and deletion of tuples in $O(1)$. However, this also implies that the above `select-from/eq` query needs to traverse the entire table file in order to find all tuples whose `:discoverer:` attribute indeed equals `"Cassini"`.

The solution for this problem consist of coming up with a set of additional *index files* that go with a data file. An index file is a file that contains enough information and enough structure so that we can navigate our way to the desired tuples with a limited number of block transfers. Much can be said about index files. We advise the reader to consult a good book on database implementations for taxonomies and overviews of different kinds of indexes. A good start is [GMUW09]. In a database system such as SQL, index files are created by means of the "CREATE INDEX" command. In our `database` ADT, it takes the following form:

```
(db:create-index! solar-system moons "Discoverer-IDX" :discoverer:)
```

This means that we create an index file called `"Discoverer-IDX"` on the `:discoverer:` part of the `moons` table of our `solar-system` database. The index file contains information to make sure that queries, such as the one just shown, are executed much more efficiently. In this chapter, we present one of the most sophisticated

[1] We assume that the `database` ADT is imported using prefix `db:`

kinds of index files, namely *B-trees*. As we will see, B-trees allow us to determine the rcid of a tuple with only a logarithmic number of block transfers. Moreover, the base of the logarithm will be fairly big which results in a very fast access times, even when we are searching huge amounts of tuples.

We start the chapter by revising some of our knowledge about trees. Needless to say, the terminology developed in chapter 6 still holds and we will try to reuse as many insights as possible in the realm of external memories that we are currently investigating. For instance, it will always remain a good idea to keep our trees as balanced as possible. After all, unbalanced trees run the risk of becoming entirely degenerated which leads to horrible performance characteristics. In section 17.2 we introduce B-trees by means of an example. We finish the section with the development of a more formal definition. In section 17.3 we present a full-fledged implementation of B^+-trees in Scheme. B^+-trees form a very attractive variant of B-trees. It is this variant that lies at the basis of index files in the relational model. In section 17.4 we tie the knowledge of chapters 16 and 17 together by implementing the `database` ADT that was described in section 16.1.4.

B-trees where first published in 1972 by R. Bayer and E. McCreight who were working at Boeing Research Labs at the time. Unfortunately, the etymology of the "B" was never explained in their article. Some authors have speculated about the meaning of the "B" by thinking of "Boeing", "balanced", "Bayer" or even far-fetched proposals such as "broad" and "bushy".

17.1 Deploying Trees on a Disk

Since the goal is to come up with a tree data structure that fits the architecture of a disk, we need to come up with clever ways of storing tree nodes in disk blocks and linking up tree nodes by means of block-transcending pointers. Let us think about the implications of this for a moment and let us try to reason about what the effect of this will be on the knowledge we have developed in chapter 6.

17.1.1 Plain Balanced Binary Trees

The most optimal trees that we know so far are the AVL-trees discussed at the end of chapter 6. An AVL-tree is a binary search tree that is nearly balanced: at all levels in the tree, the difference in height between the left and right subtrees is smaller than or equal to 1. This guarantees a $O(log_2(n))$ performance characteristic for searching.

In the realm of disks, AVL-trees have an important drawback however. If we assume that the search keys are relatively small (e.g. a string of a few dozen bytes maximally), then it is reasonable to assume that a disk block can host multiple AVL-tree nodes. However, there is nothing in the algorithms of an AVL tree which guarantees that nearby nodes (e.g. both children of a node) reside in the same disk block. As a consequence, every time we follow a node pointer,

that node can potentially sit in a different disk block! Using the terminology developed in chapter 14, we say that the AVL-tree code exhibits very poor locality properties. As we know, this results in poor caching behavior which possibly results in an excessively high number of actual block transfers.

A reaction might be to store "adjacent" nodes in the same disk block. E.g., we could decide to store a node, its two children and its 4 grandchildren in one disk block. But when following this scenario, there is no longer the need to link up those children: we might as well "flatten" these 7 nodes into one node that has 7 keys and references to all 8 grand-grandchildren. In other words, we have come up with a k-ary tree (here: $k = 8$) in a natural way.

17.1.2 Do k-ary Search Trees Make Sense?

A *k-ary search tree* is a search tree in which every node contains $k - 1$ sorted keys. The keys are separated and surrounded by k pointers. Upon visiting a node, the search algorithm compares the search key with the keys that are stored in the node. Depending on the outcome of the $(k - 1)$-way comparison, one of the k pointers is followed and recursively investigated. If a k-ary tree is perfectly balanced, its height is in $O(log_k(n))$ where n is the number of values sitting in the tree. How can we perform the $(k - 1)$-way comparison efficiently? If we store the $k-1$ keys in a sorted order, then we can make the decision about which pointer to follow using binary search. This which requires $O(log_2(k))$ computational steps. However, this has to be done at *every* level in the search tree. Hence, searching a k-ary search tree requires $(O(log_k(n).log_2(k))$ steps. Applying the arithmetical properties of the logarithmic function, this boils down to $O(log_2(n))$ again.

Hence, in central memory, k-ary search trees give us no speedup over ordinary binary search trees!

Fortunately, this reasoning does not hold for k-ary trees when deploying them on a disk. The work to be done at every level of the tree consist of reading the k-ary node into central memory, searching it using binary search and following the corresponding pointer. This takes $T + t.log_2(k)$ time for some small constant t and where $T = T_{seek} + T_{latency} + T_{transfer}$ as we saw in chapter 14. We have to do this operation at every level of the tree which gives us $log_k(n).(T + t.log_2(k))$. This cannot be reduced to $log_2(n)$ because T is such a big constant when compared to t. On modern computer systems, T will even be much greater than $t.log_2(k)$ because processors are *much* faster than disks. We can thus ignore the $t.log_2(k)$ factor altogether which indeed gives us an $O(log_k(n))$ performance characteristic!

Hence, on a disk it certainly does pay off to replace binary trees by k-ary trees!

Obviously, we get the best performance if we align the node of a k-ary tree with one disk block. This is one of the most fundamental assumptions of B-trees and B^+-trees.

17.2 B^+-trees

In this section, we explain B-trees and a variant called B^+-trees. These are currently used as index files in commercial database systems. We start out by explaining the general principle of these trees without dealing with the technicalities. After we have understood how these trees are created, deleted and searched, we study a full-fledged implementation of B^+-Trees on disk in section 17.3.

From now on we make a clear distinction between n and N. n represents the number of elements stored in our external data structure. In an index file, n thus corresponds to the number of rcids stored in the index file. $N+1$ is the arity of the trees that are used to index those rcids. This basically means that every node[2] stores N keys that are surrounded by $N+1$ pointers referring to $N+1$ children.

One of the fundamental laws that governs the composition of B-trees and B^+-trees is that the keys are stored in the nodes in sorted order. This means that the binary search algorithm can be applied when searching a node. $N+1$ pointers separate the N keys. Hence, we can organize the tree in such a way that all data values that lie between two consecutive values in a node reside in a subtree that is pointed to by the pointer separating those two consecutive values. The leftmost pointer of a node refers to a subtree that contains all elements that are smaller than the smallest element of that node. The rightmost pointer of a node refers to a subtree that contains all elements that are greater than the greatest element of that node.

17.2.1 An Example

The technical definition of B-trees and B^+-trees is best given after understanding its insertion and deletion procedures by means of an example. Let us start with B-trees. We assume that $N = 4$ and that we store plain numbers in the B-tree. Hence, every node will be able to store 4 sorted data values and 5 pointers referring to children. Figure 17.1 illustrates the insertion algorithm. We start with an empty B-tree and we successively insert 50, 10, 20, 90, 13, 17, 11, 58 and 12 in that order.

We start out by inserting the first 4 values. This is a fairly straightforward evolution of the data structure. The values are inserted in a sorted list causing a storage move in order to shift values "to the right" should this be necessary. This explains the first 4 steps. Things get more interesting when the node is entirely full. This is the case after 50, 10, 20 and 90 have been inserted. When inserting the next value (i.e. 13), the B-tree tries to apply the same logic. Since the node is entirely full, the node is *split* in two. Half of the keys stay in the original node and half of the keys are copied into a new sibling. The value that separates both halves (in this case this is 20) is propagated upwards. This gives rise to the creation of a new root node. The separating key is inserted in the

[2]For now, we assume that every node stores the same number of keys. In our actual implementation, leaf nodes will have to be treated differently from internal nodes.

new root node and the pointers that lead to both siblings are installed to the left and to the right of the separating key. Next, we insert 17. This causes the B-tree to search for the node in which to insert 17. The search algorithm starts at the root of the B-tree and works its way down in the expected way: nodes use the binary search technique and the separating pointers are followed to eventually find the leaf node that hast to host the new value. This causes 17 to be inserted in the leftmost leaf. Exactly the same reasoning applies when inserting 11. Notice that at this stage the leftmost leaf node has become full again. Next we try to insert 58. This triggers the default technique again: this time the search leads us to the rightmost leaf node. Since there is still room left in that node, 58 is inserted and we are done. Finally, we insert 12. The search technique leads us to the leftmost leaf node. However, this time we discover that the node is full and we therefore decide to split the node. Half the values (i.e. 10 and 11) stay in the original node and half the values (i.e. 13 and 17) are moved into a new sibling node. The separating value (12 in our case) is recursively propagated upwards. It is therefore inserted in the root node and appears before 20. The 3 pointers that surround 12 and 20 in the root are updated accordingly.

Figure 17.2 applies the same logic but with a slightly different algorithm for splitting nodes. The result is a variant of B-trees, known as B^+-trees. The difference lies in the fact that, upon splitting a node, the separating key is not only propagated upwards. It also stays in the original node. This means that *every* value residing in a B^+ tree always resides in one of the leaves! Some of these values also get propagated upward and therefore provide the internal nodes with information about the values in the leaves. The internal nodes therefore become pure navigational nodes while the leaves store the actual data. One can verify that a key in an internal node corresponds to the biggest key among all keys that sit in the leaves of its left subtrees.

The fact that internal nodes only store keys for navigation does not really seem to matter in our case because our sample trees store plain numbers. However, as soon as "richer" values are stored in a B-tree, the difference between a B-tree and a B^+ becomes clearer: the internal nodes only store search keys. They help the search process navigate its way down to the actual key-values[3] which are stored in leaf nodes. The big advantage of this is that the internal nodes can contain more keys than the leaf nodes. This leads to trees with a higher branching factor which, by consequence, allows for faster navigation since the tree will be less deep.

Obviously, no node in a B-tree (or B^+-tree) can contain more than N keys. As soon as we try to insert an $(N+1)$'th key, the node gets split in two nodes that store $N/2$ keys. In a B^+-tree we get one node storing $N/2+1$ keys and another one storing $N/2$ keys).

Deletion is illustrated in figure 17.3. We focus on the final B^+-tree of figure 17.2 and we delete the values in the following order: 11, 17, 10, 58, 20 and

[3]In a database implementation, these values will be the rcids that refer to the tuples in a data file.

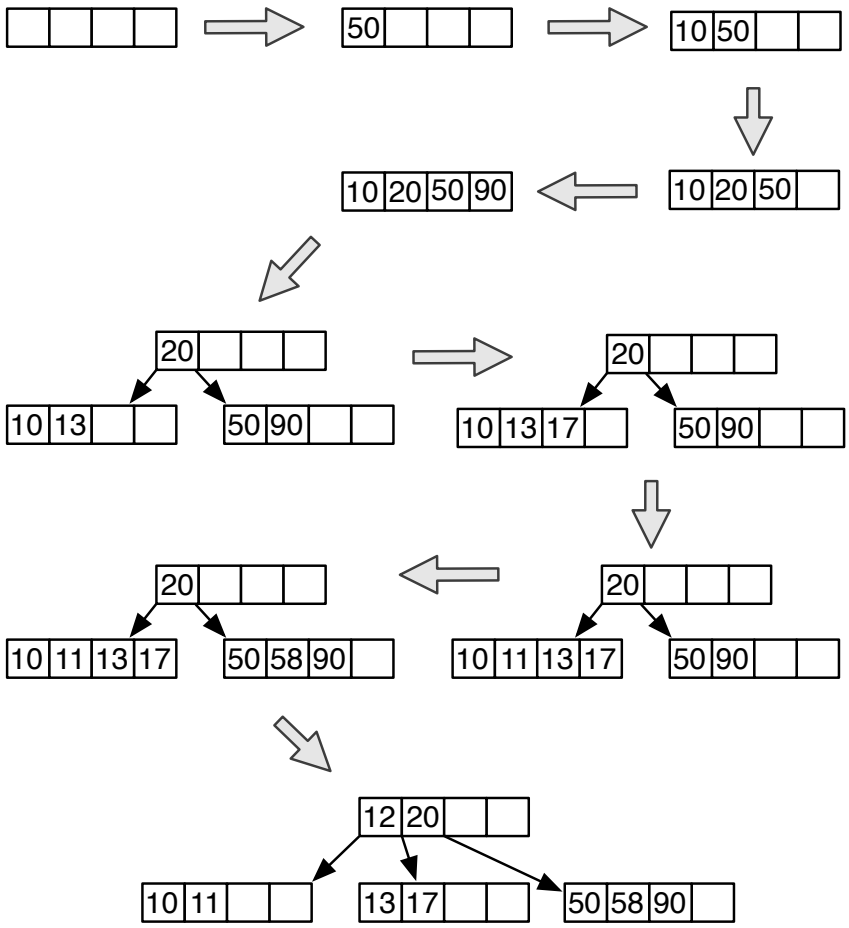

Figure 17.1: B-tree Insertion Algorithm

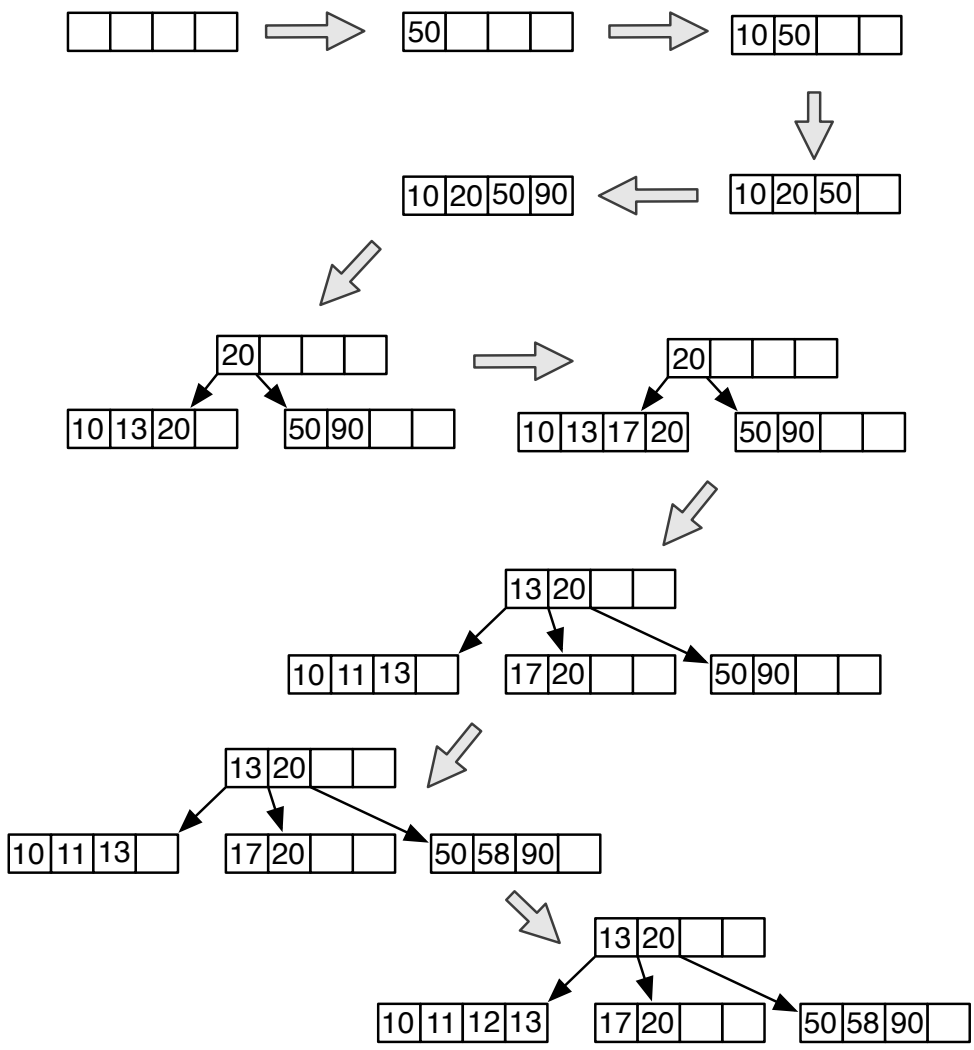

Figure 17.2: B^+-tree Insertion Algorithm

13. We start by deleting 11. The first part of the algorithm consists of looking up 11. This leads us to the leftmost leaf node. Then we remove 11 from that leaf node and we are done. Naively spoken, we could use this algorithm all the time. However, it soon leads to a tree that has many empty nodes. Removing those empty nodes would soon result in a B-tree that is no longer balanced. We therefore apply a clever deletion algorithm which guarantees a balanced tree. The actual deletion algorithm prescribes that a tree node should always be at least half full. As soon as the number of elements in a node drops below $N/2$, we have to do something about this underflow. Our first option will be to borrow an element in the left or right sibling of the node suffering from underflow. This is possible as long as one of those siblings contain at least $N/2 + 1$ values. Consider the deletion of 17. We can borrow 13 from the leftmost node and this neatly rebalances the tree again. Of course, when borrowing a value from one of the siblings, we have to make sure that the corresponding navigational information in the parent of both siblings is updated accordingly. Therefore, in our example, 13 is replaced by 12 in the parent node. Let us now delete 10 from the tree. This time the search process leads us to the leftmost leaf node. Again, removing 10 from that node would cause the node to suffer from an underflow. However, this time, borrowing a value from its sibling is impossible since this would just shift the problem to the sibling. The deletion algorithm therefore prescribes that we merge two sibling nodes into one node as soon as it is impossible to borrow a value from any of the two siblings of a node. Notice that merging is always possible: the node from which the key is deleted suffers from underflow and thus maximally has $N/2 - 1$ values. The siblings cannot lend a value and thus maximally contain $N/2$ values as well. Hence, merging the node with a sibling never gives rise to more than N values. Obviously, when merging two nodes, the navigational information in their (shared) parent needs to be removed. This is where the deletion algorithm calls itself recursively. This is why 12 gets removed from the parent node.

17.2.2 Definitions

We are now ready to define B-trees and their insertion and deletion algorithm a bit more precisely. We give the definitions for "ordinary" B-trees. A full-fledged implementation of (the slightly more sophisticated) B^+-trees is presented in the following sections however.

A B-tree is an $(N + 1)$-ary tree such that every node contains (a) at most N sorted values and (b) at least $N/2$ sorted values. The root node is the only exception to this rule. It is allowed to contain between 0 and N values; i.e. it is allowed to contain less than $N/2$ values. Figure 17.4 shows a typical B-tree node (containing $T < N$ keys). p_0 is a pointer that refers to a subtree containing all keys that are smaller than k_1. p_T refers to a subtree containing all keys that are bigger than k_T. p_2 refers to a subtree containing all the keys k such that $k_2 < k \leq k_3$. And so on.

Searching Searching a B-tree starts at the root. The binary search algorithm

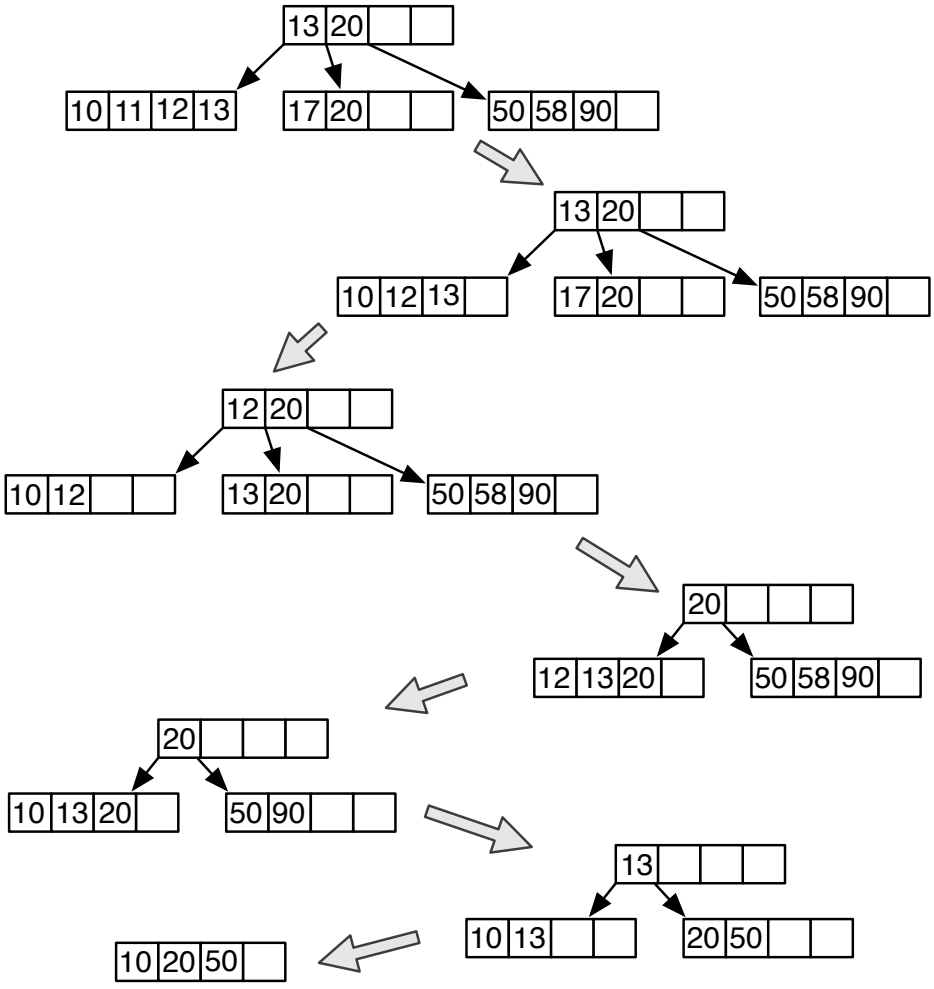

Figure 17.3: B-tree Deletion Algorithm

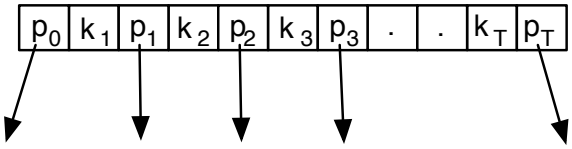

Figure 17.4: The Structure of an (internal) B-tree node

is applied in every node. If the search key is smaller than the leftmost key in the node, then the search follows the leftmost pointer of the node. If the search key is greater than the rightmost key of the node, then the search follows the rightmost meaningful pointer. Otherwise, the search key lies between two consecutive keys. In that case, the search continues by following the pointer that separates those two keys.

Insertion The insertion procedure first applies the search procedure in order to locate the leaf node that is *supposed* to host the element to be inserted. In case the node has a free slot, the element is inserted and the procedure terminates. However, if the node appears to be full, the node is split and the keys are evenly distributed over both nodes. The key that separates the keys of both nodes is recursively inserted in the parent of the nodes (thereby correctly attaching the pointers to the both nodes resulting from the split). In case a split of the root node is required (because it is full), a new root node is created that contains one single key (i.e. the key that separates the values in the split nodes).

Deletion The deletion procedure first applies the search procedure in order to locate the B-tree node that contains the key to be deleted. Then the key is actually erased from the node. If the resulting node still contains at least $N/2$ data values, the procedure terminates. Otherwise, the siblings of the node are inspected and we try to borrow a data value from one of those siblings. Should this be impossible (because both siblings contains exactly $N/2$ values), then one of the siblings is randomly selected and the node is merged with that sibling. This is possible since none of those siblings contain more than $N/2$ values (since borrowing was impossible). The key that separates the node from its sibling is then recursively deleted from the parent. In case this causes the root to become empty, it is deleted and replaced by the node resulting from the merge of its only two descendants.

17.3 An Implementation in Scheme

Now that we have a fairly accurate idea of how B-trees (and B^+-trees) look like, we study a full-fledged implementation in Scheme. From now on, we focus

on B^+-trees. The implementation is built on top of the `disk` ADT of chapter 14 and assumes the tuple storage model of chapter 16. I.e., we assume the existence of a data file (i.e. a file of the ADT `table`) that stores a collection of tuples that are identified by means of their rcid. A B^+-tree is an index file whose internal nodes are organized according to some key type (e.g. a `string` (= 3) indicating the name of a planet in our example presented in section 16.1) and whose leaf nodes associate an rcid with every key stored. The rcid refers to a position in the data file which contains the actual tuple.

The implementation consists of the following ADTs:

- `b-tree` is the actual ADT that represents a B^+-tree on disk as well as in central memory. In the terminology developed in chapter 16, `b-tree`s have a dual representation. Obviously, because of the size of the tree, only parts are stored in central memory at any given moment in time.

- `node` is an ADT that can be considered as an abstraction layer build on top of the `block` ADT of chapter 14. It offers us procedures to reason about a disk block in terms of splitting, merging, inserting keys etc. Like this, we do not have to be bothered with the low level details of encoders, decoders and byte offsets in our B^+-tree algorithms. The purpose of the `node` ADT studied in this chapter is the same as the purpose of the `node` ADT studied in chapter 16.

- `node-type` is an ADT that allows us to bundle a number of quantitative characteristics about the B^+-tree nodes. E.g., if we use strings as search keys, we will need the exact length of the strings. This has an impact on the number of strings that can be stored in a node. Such information is contained in a `node-type`. It can be compared to the `schema` ADT of chapter 16: without a `node-type`, a disk block is just a flat sea of raw bytes. Given a disk block, it is impossible to know whether it is a 5-ary node with string keys of size 9, whether it is a 10-ary node with integer keys of length 4, or any other organisation that one might think of. A `node-type` provides us with the information needed to correctly interpret those bytes as B^+-tree nodes. It stores the key type and the key size that have to be used in order to correctly decipher the bytes of a node.

Consider the example of the introduction again:

```
(db:create-index! solar-system moons "Discoverer-IDX" :discoverer:)
```

This operation creates a B^+-tree whose file name is `"Discoverer-IDX"`. The tree will consist of a collection of nodes, the exact number of which depends on the amount of tuples in the `moons` table. The leaf nodes of the tree store rcids that refer to the actual tuples sitting in the `moons` table. Furthermore, the tree will reserve one single disk block in order to store the node type of the nodes. This node type contains the knowledge that the tree is used to search according to discoverers, i.e. strings of size 10 (see section 16.1.1). This information is represented by two bytes, to wit 3 (type: string) and 10 (the size).

17.3.1 Node Types

Our B^+-tree algorithms are responsible for organizing a B^+-tree built atop a collection of raw disk blocks. A central design decision is that every node of the tree corresponds to precisely one block on the disk. But given a block of raw bytes, how can the B^+-tree algorithm know the meaning of these bytes? How can we know the length of the keys stored in a node? How can we know the number of keys and the number of pointers stored in the tree node? This is where the `node-type` abstraction comes in.

```
1  ADT node–type
2
3  new
4      ( disk byte byte → node–type )
5  node–type?
6      ( any → boolean )
7  disk
8      ( node–type → disk )
9  key–type
10     ( node–type → byte )
11 key–size
12     ( node–type → byte )
13 key–sent
14     ( node–type → any )
15 leaf–capacity
16     ( node–type → number )
17 internal–capacity
18     ( node–type → number )
```

Think back of our implementation of the `block` ADT that represents blocks as sequences of `block-size` consecutive bytes on a disk. Suppose that we want to encode keys of size `key-size` (e.g. suppose that we have strings of length 10) and suppose that a pointer occupies p bytes. For internal nodes, p is equal to `disk:block-ptr-size` since the internal B^+-nodes refer to yet other B^+-nodes (remember that disk blocks are referred to by block pointers). For leaf nodes, p is `rcid:size` since leaf nodes refer to tuples that sit in some data file (remember that tuples are referred to by rcids). If N_{cap} is the number of keys that can be stored in a node — called the capacity of the node — then the following equation must hold for all nodes:

$$\texttt{block-size} = \texttt{block-ptr-size} + N_{cap} \times (\texttt{key-size} + p)$$

because every node is required to store N_{cap} pointers and N_{cap} keys, plus one block pointer in the leftmost (i.e. zeroth) location of the node. We use a small trick here. Internal nodes "really" use the zeroth pointer in order to refer to a disk block that is the leftmost descendant of the internal node. In other words, internal nodes actually contain N_{cap} keys and $N_{cap} + 1$ block pointers. Leaf

nodes do not use the zeroth pointer because they store exactly one rcid per key, i.e. they store N_{cap} keys and N_{cap} pointers. Hence, leaf nodes need one pointer less than internal nodes. We can therefore use this pointer exactly for the purpose of recognising a leaf node: a leaf node is a node whose leftmost block pointer refers to `null-block`. Still, it is a block pointer that must be stored. The difference between leaf nodes and internal nodes is depicted in figure 17.5. We explain the details further below.

Solving this equation for N_{cap}, yields

$$N_{cap} = \frac{\texttt{block-size} - \texttt{block-ptr-size}}{\texttt{key-size} + p}$$

as the number of keys that can be stored in a given disk block. Notice that except for the `key-size`, all constants in this equation depend on the hardware. Also notice that this expression needs to be calculated twice: once for internal nodes (where $p =$ `disk:block-ptr-size`) and once for leaf nodes (where $p =$ `rcid:size`). This explains the following constructor for the `node-type` ADT. In our code, N_{cap} is represented by the local variables `leaf-node-cpty` and `internal-node-cpty`. In what follows, we use $N_{cap}^{internal}$ and N_{cap}^{leaf} as slightly more precise notations for N_{cap}. The constructor code relies on the traditional `make` procedure that allocates the record necessary to bundle the constituents.

```
(define-record-type node-type
  (make d t s l i)
  node-type?
  (d disk)
  (t key-type)
  (s key-size)
  (l leaf-capacity)
  (i internal-capacity))

(define (new disk key-type key-size)
  (define leaf-node-cpty
    (div (- disk:block-size disk:block-ptr-size)
         (+ key-size rcid:size)))
  (define internal-node-cpty
    (div (- disk:block-size disk:block-ptr-size)
         (+ key-size disk:block-ptr-size)))
  (if (or (< leaf-node-cpty 2)
          (< internal-node-cpty 2))
      (error "key too large (new)" key-size)
      (make disk key-type key-size leaf-node-cpty internal-node-cpty)))
```

The node type prescribes that we can interpret disk blocks as nodes storing keys of type `key-type` (which can be 0, 1, 2 or 3 depending on whether we are using natural numbers, integer numbers, floating point decimals or strings) and whose length is `key-size`. Notice that we do not allow nodes with less than 2 keys for otherwise the idea of splitting a node does not make sense.

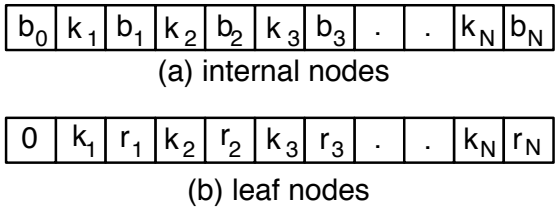

Figure 17.5: Node layout for B^+-trees

17.3.2 Node Layout

As already explained, the layout of nodes in disk blocks is slightly different in the case of leaf nodes and internal nodes. The layout for both types of nodes is displayed in figure 17.5. The precise technical way in which that layout is mapped onto raw disk blocks is explained in section 17.3.3.

- An internal B^+-tree node consists of $N_{cap}^{internal}$ keys k_i which are surrounded by $N_{cap}^{internal} + 1$ pointers b_i. This is illustrated in figure 17.5(a). The idea is that the keys split up the range of keys into disjoint intervals: all keys in the interval $]k_{i-1},\ k_i]$ can be found in the subtree that is pointed to by block pointer b_{i-1} that separates those keys.
- A leaf node consists of N_{cap}^{leaf} keys together with their associated rcid. This means that one slot is unused (i.e. the zeroth slot) compared with internal nodes. As explained, this slot can be filled with `fs:null-block` in order to encode the very fact that the node is a leaf node. This is illustrated in figure 17.5(b). This node layout allows us to distinguish leaf nodes from internal nodes merely by accessing the zeroth slot.

An important decision to be made for internal nodes is what happens for search keys that are exactly equal to one of the keys sitting in an internal node. Is the rcid belonging to the search key to be found in the subtree pointed to by the pointer on the left hand side of the key or is it to be found in the subtree pointed to by the pointer on the right hand side of the key? This is a matter of convention. In our implementation we taken the convention that search keys equal to a key k_i have to be found in the node that is referred to by the block pointer sitting to the left of that key. In other words, a pointer b_i leads to a node that contains all keys that are strictly greater than k_{i-1} and smaller than or equal to k_i. Hence the interval $]k_{i-1},\ k_i]$.

17.3.3 The Node ADT

This section presents the `node` ADT which is an abstraction built on top of the `block` ADT. That abstraction allows our B^+-tree algorithms to be liberated

from low level operations that operate directly on the level of disk blocks and the bytes they store. For convenience, we use a unified pointer type `pntr` = `number` ∪ `rcid` to indicate block pointers *or* record identifiers.

Here is the formal definition of the `node` ADT:

```
1  ADT node
2
3  new
4      ( node–type boolean → node )
5  read
6      ( node–type number  → node )
7  write!
8      ( node → ∅ )
9  delete!
10     ( node → ∅ )
11 position
12     ( node → number )
13 node?
14     ( any → boolean )
15 type
16     ( node → node–type )
17 capacity
18     ( node → number )
19 size
20     ( node → number )
21 meaningless?
22     ( node number → boolean )
23 leaf?
24     ( node → boolean )
25 locate–leftmost
26     ( node any → number )
27 key
28     ( node number → any )
29 key!
30     ( node number any → ∅ )
31 pointer
32     ( node number → pntr )
33 pointer!
34     ( node number pntr → ∅ )
35 key–pointer!
36     ( node number any pntr → ∅ )
37 key–pointer–insert!
38     ( node number any pntr → ∅ )
39 key–pointer–delete!
40     ( node number → ∅ )
41 key–pointer–insert–split!
```

```
    ( node node number any
      pntr boolean          → any )
borrow−from−left?
    ( node node any → any ∪ { #f } )
borrow−from−right?
    ( node node any → any ∪ #f } )
merge!
    ( node node any → ∅ )
```

A node is constructed with `new` given a `node-type` and a `boolean`. The `node-type` knows about the disk on which the node's block is to be allocated and contains the parameters needed to organize the `node` (i.e. the keys and the pointers) on that disk. The boolean value indicates whether or not we are creating a leaf node. The node is created by asking the file system for a new block and by filling the slots of that block with sentinel keys and null pointers. Remember that the pointer type differs for leaf nodes (rcids) and internal nodes (ordinary block pointers). Once again, we are dealing with a data structure with a dual representation: in central memory, a node consists of a disk block paired with the node type that is needed to correctly interpret the block.

```
(define-record-type node
  (make t b)
  node?
  (t type)
  (b block))

(define (new ntyp leaf)
  (define disk (ntype:disk ntyp))
  (define ktyp (ntype:key-type ntyp))
  (define ksiz (ntype:key-size ntyp))
  (define blck (fs:new-block disk))
  (define node (make ntyp blck))
  (define sent (sentinel-for ktyp ksiz))
  (pointer! node 0 (if (not leaf) 1 fs:null-block))
  (do ((null (null-ptr-for node))
       (slot 1 (+ slot 1 )))
    ((> slot (capacity node)))
    (key-pointer! node slot sent null))
  node)

(define (null-ptr-for node)
  (if (leaf? node)
      rcid:null
      fs:null-block))

(define (sentinel-for ktyp ksiz)
  (cond ((= ktyp natural-tag)
```

```
 (- (expt 256 ksiz) 1))
((= ktyp integer-tag)
 (- (div (expt 256 ksiz) 2) 1))
((= ktyp decimal-tag)
 +inf.0)
((= ktyp string-tag)
 (utf8-sentinel-for ksiz))))
```

Notice that both the kind of null pointer as well as the kind of sentinel depends on the type of keys and the type of pointers that we are storing. The technical details of these procedures are not that important in order to understand the rest of the algorithms. However, we do encourage the technically inclined reader to understand them in detail. We refer to section 14.1.3 for the implementation of `utf8-sentinel-for`.

An alternative constructor is `read!`. It takes a block number and a node type. Depending on the information in the zeroth slot that is found in the disk block designated by the number, the block is read as a leaf node or as an internal node. The implementations of `read!` looks as follows.

```
(define (read ntyp bptr)
  (define disk (ntype:disk ntyp))
  (define blck (disk:read-block disk bptr))
  (make ntyp blck))
```

The operations on nodes that do not require the node's type are directly relayed to their underling disk block:

```
(define (delete! node)
  (fs:delete-block (block node)))

(define (position node)
  (disk:position (block node)))

(define (write! node)
  (disk:write-block! (block node)))
```

Slot Accessors and Mutators

The following procedures form the raison d'être of the `node` ADT. They will allow our B^+-tree algorithms to be written in terms of nodes, keys and pointers instead of raw bytes stored in disk blocks.

Conceptually, an internal B^+-tree node consists of $N_{cap}^{internal}$ keys that are surrounded by $N_{cap}^{internal} + 1$ pointers. A leaf node consists of N_{leaf} keys that are paired with N_{leaf} rcids. However, for the sake of simplicity we will not *actually* store the keys and the pointers in this alternating fashion. Instead, we store the keys in the rightmost bytes of the node and we store the pointers in the leftmost ones. This explains the implementations for `key`, `key!`, `pointer` and `pointer!` presented further below. Note the strong conceptual difference between the `slot` number argument (which is B^+-tree-speak) and the calculated

`offs` (which is `block`-speak). The `slot` number lies between 0 and N_{cap}. The `offs` lies between 0 and `disk:block-size` -1.

```
(define (key node slot)
  (define ksiz (ntype:key-size (type node)))
  (define ktyp (ntype:key-type (type node)))
  (define blck (block node))
  (define offs (- disk:block-size (* ksiz slot)))
  (define decoder (vector-ref decoders ktyp))
  (decoder blck offs ksiz))

(define (key! node slot skey)
  (define ksiz (ntype:key-size (type node)))
  (define ktyp (ntype:key-type (type node)))
  (define blck (block node))
  (define offs (- disk:block-size (* ksiz slot)))
  (define encoder! (vector-ref encoders ktyp))
  (encoder! blck offs ksiz skey))
```

`key` takes a `node` and a `slot` number. It uses the node's `key-size` (available from the node's type) in order to calculate the offset in the node's block that corresponds to the first byte of the given key. This byte number is the rightmost byte (i.e. `file:block-size`) minus the amount of slots to be skipped (i.e. (* `ksize slot`)). `key!` performs the exact same calculation to store the key `skey` in a given `slot` number in the `node`. The `encoders` and `decoders` vector were explained in section 16.3.3 of chapter 16: they contain encoding and decoding procedures for every supported key type.

The implementations of `pointer` and `pointer!` follow a similar reasoning. However, since we store pointers in the leftmost positions of a block, their `offs` is calculated as $0 +$ (* `pntr-size slot`) where `pntr-size` depends on whether we are manipulating the type of pointers stored in a leaf node or the type of pointers stored in an internal node.

```
(define (pointer node slot)
  (define blck (block node))
  (define rid? (and (not (= slot 0)) (leaf? node)))
  (define pntr-size (if rid?
                        rcid:size
                        disk:block-pointer-size))
  (define offs (* pntr-size slot))
  (define bptr (disk:decode-fixed-natural blck offs pntr-size))
  (if rid? (rcid:fixed->rcid bptr) bptr))

(define (pointer! node slot pntr)
  (define blck (block node))
  (define rid? (and (not (= slot 0)) (leaf? node)))
  (define pntr-size (if rid?
```

```
                    rcid:size
                    disk:block-pointer-size))
  (define offs (* pntr-size slot))
  (disk:encode-fixed-natural! blck offs pntr-size (if rid?
                                                      (rcid:rcid->fixed pntr)
                                                      pntr)))
```

Notice that pointers are encoded as fixed sized natural numbers. In the case of block pointers this is trivial. However, when encoding and decoding rcids, we are actually dealing with pairs. We therefore have to covert these pairs to fixed sized natural numbers and vice versa. This is the role of `rcid:rcid->fixed` and `rcid:fixed->rcid` whose implementation was presented in section 16.2.1.

Some More Mutators

The four procedures just described liberate us from the level of individual bytes stored in disk blocks. We can now use them to further enrich our vocabulary with additional procedures for manipulating nodes. All of them will make out B^+-tree implementation more readable.

`key-pointer!` is just a combination of `key!` and `pointer!` which stores a key `skey` and a pointer `pntr` in a node in one shot.

```
(define (key-pointer! node slot skey pntr)
  (key!     node slot skey)
  (pointer! node slot pntr))
```

`key-pointer-insert!` also stores a key `skey` and a pointer `pntr` in a given `slot` number. However, the key-pointer pair is inserted in sorted order by conceptually moving the "bigger" key-pointer pairs one position to the right. `move` is a local procedure that takes care of the actual storage move.

```
(define (key-pointer-insert! node slot skey pntr)
  (define nsiz (ntype:capacity (type node)))
  (define (move index)
    (if (> index slot)
        (let ((previous-index (- index 1)))
          (key-pointer! node index
                        (key node previous-index)
                        (pointer node previous-index))
          (move previous-index))))
  (move nsiz)
  (key-pointer! node slot skey pntr))
```

`key-pointer-delete!` does the exact opposite: it erases a key-pointer pair from a node and fills up the resulting "hole" by shifting all bigger pairs one position to the left. Again, `move` does the storage move.

```
(define (key-pointer-delete! node slot)
  (define nsiz (ntype:capacity (type node)))
  (define ktyp (ntype:key-type (type node)))
  (define ksiz (ntype:key-size (type node)))
```

```
(define sent (sentinel-for ktyp ksiz))
(define nulp (null-ptr-for node))
(define (move index)
  (if (< index nsiz)
      (let ((next-index (+ index 1)))
        (key-pointer! node index
                      (key node next-index)
                      (pointer node next-index))
        (move next-index))))
(move slot)
(key-pointer! node nsiz sent nulp))
```

Some More Accessors

`capacity` returns the total number of keys that fit in a node. Notice that this is different for leaf nodes and internal nodes because the pointer size used by both types of nodes is different.

```
(define (capacity node)
  (define ntyp (type node))
  (if (leaf? node)
      (ntype:leaf-capacity ntyp)
      (ntype:internal-capacity ntyp)))
```

The following procedure will prove handy when implementing the B^+-tree algorithms. It allows us to check whether or not a slot contains meaningful data. The procedure returns true if the slot number is beyond the capacity of the node (remember that we start counting from 0 to $N_{cap} - 1$) or if the slot is taken by a sentinel key.

```
(define equals  (vector = = = string=?))


(define (meaningless? node slot)
  (define ntyp (type node))
  (define ktyp (ntype:key-type ntyp))
  (define sent (ntype:key-sent ntyp))
  (define ===?  (vector-ref equals ktyp))
  (if (= slot (capacity node))
      #t
      (===? (key node (+ slot 1)) sent)))
```

Searching Nodes

The `locate-leftmost` procedure can be used to search some search key `skey` in a given `node`. It is used by the `find!`, the `insert!` and the `delete!` operations of the `btree` ADT. The procedure implements a binary search algorithm (see section 3.4.3) which is possible because the keys are stored in one block that obviously allows $O(1)$ access. However, `locate-leftmost` uses a slightly more sophisticated version of binary search because it gracefully handles duplicates in the sense that it always searches for the leftmost occurrence

of the search key in a node. This is e.g. needed for implementing SQL-queries correctly because we want to return *all* key-rcid pairs that satisfy the query's search criterion. We will accomplish this by locating the leftmost key first and then use `set-current-to-next!` to find the others. Notice that the procedure is general enough to search for any type of key. The key type of the node is used to select the correct comparison from one of the vectors `smaller` and `greater`.

```
(define smaller (vector < < < string<?))
(define greater (vector > > > string>?))

(define (locate-leftmost node skey)
  (define ntyp (type node))
  (define ktyp (ntype:key-type ntyp))
  (define <<<? (vector-ref smaller ktyp))
  (define >>>? (vector-ref greater ktyp))
  (define (search first last)
    (if (> first last)
        last
        (let*
            ((mid (div (+ first last) 2))
             (mid-key (key node mid)))
          (cond
            ((>>>? skey mid-key)
             (search (+ mid 1) last))
            ((<<<? skey mid-key)
             (search first (- mid 1)))
            (else
             (let ((try (search first (- mid 1))))
               (if (negative? try)
                   try
                   (- mid)))))))))
  (search 1 (ntype:capacity ntyp)))
```

Given a `node` and a `key`, we try to search for that key between slots 1 and (`ntype:capacity ntyp`) by recursively investigating the range of keys between slots `first` and `last`. We have two possible outcomes:

- Either the key is *not* found. This is indicated by returning a *positive* number. `last` is the number of the slot where the key was expected to be found. This will be useful when calling `locate-leftmost` during the insertion procedure of our B^+-tree implementation.
- Either the key is *found*. In that case, `mid` is the success slot and this slot is returned as a *negative* number in order to indicate that the key was found.

After obtaining a result form this procedure, the procedure `complement` needs to be applied. The return value of `complement` corresponds to the so called

"two's complement" of its argument. It leaves positive numbers untouched and returns the predecessor of the absolute value of any negative number. This is because finding a key at slot s requires us to follow the pointer immediately to the left of s (i.e. the pointer corresponding to slot $s-1$). Remember that the pointer to the left of a key leads to a node that stores all keys smaller than or *equal* to the key. Hence, when returning a success slot s from `locate-leftmost`, we have to descend by following the pointer residing in slot number $s-1$. In summary, if a call to `locate-leftmost` returns a positive number s, then the search key was found in slot number s of the node. If that call returns a negative number s, then the key was not found and we have to follow the pointer residing in slot number $|s|-1$. This is captured by the following procedure:

```
(define (complement slot)
  (if (negative? slot)
    (- -1 slot)
    slot))
```

Now we are ready to calculate the size of a node. The size of a node is the number of slots that it actually contains at a particular moment in time. This is different from the capacity of a node. The size is calculated by looking up the leftmost sentinel key in a node. If found, the preceding slot must have been the last slot containing an actual key. Otherwise, the size is just the capacity because the node does not store any sentinel. Hence the following implementation of `size`:

```
(define (size node)
  (define ncap (capacity node))
  (define sent (ntype:key-sent (type node)))
  (define indx (locate-leftmost node sent))
  (if (negative? indx)
      (complement indx)
      ncap))
```

17.3.4 Insertion Machinery: Splitting Nodes

Finally, we move on to discussing the more difficult procedures of the `node` ADT. These procedures contain the hairy details of splitting and merging nodes and will therefore make our insertion and deletion procedures much more readable. We start with `key-pointer-insert-split!` which is invoked by the B^+-tree insertion algorithm whenever a key-pointer pair has to be added to a node that is already full. When this happens, the B^+-tree insertion procedure creates a fresh `new-node` on the disk and invokes `key-pointer-insert-split!` with the full `node`, the empty `new-node`, the key and pointer values that have to be inserted and the `slot` number that corresponds to the position of the key-pointer pair to be inserted (pretending as if the node was not full).

```
(define (key-pointer-insert-split! node new-node slot skey pntr leaf)
  (define nsiz (ntype:capacity (type node)))
```

```
(define ktyp (ntype:key-type (type node)))
(define ksiz (ntype:key-size (type node)))
(define sent-skey (sentinel-for ktyp ksiz))
(define nptr (null-ptr-for node))
(define split-slot (+ (div (+ nsiz 1) 2) 1))
(define at-end (> slot nsiz))
(define hold-key (if at-end skey (key node nsiz)))
(define hold-datum (if at-end pntr (pointer node nsiz)))
(define (move slot new-slot)
  (cond
    ((<= slot nsiz)
     (key-pointer! new-node new-slot
                   (key node slot)
                   (pointer node slot))
     (move (+ slot 1) (+ new-slot 1)))
    (else
     new-slot)))
(define (clear slot)
  (when (<= slot nsiz)
    (key-pointer! node slot sent-skey nptr)
    (clear (+ slot 1))))
(if (not at-end) (key-pointer-insert! node slot skey pntr))
(let*
    ((prop-key (key node (if leaf (- split-slot 1) split-slot)))
     (insert-slot
      (cond
        (leaf
         (pointer! new-node 0 fs:null-block)
         (move split-slot 1))
        (else
         (pointer! new-node 0 (pointer node split-slot))
         (move (+ split-slot 1) 1)))))
  (key-pointer! new-node insert-slot hold-key hold-datum)
  (clear split-slot)
  prop-key))
```

The procedure contains two local procedures `move` and `clear`. `clear` stores the sentinel key-pointer pair in the new node starting at some slot number. `move` copies a number of slots (from left to right) from the `node` into the `new-node`. It returns the first free index in the new node. This is where the very last key-pointer pair is to be stored. This pair consists of `skey` and `pntr` if the `slot` to be inserted falls outside the boundaries of the original node. Otherwise, it consists of the key-value pair that can be found at position `nsiz` in the original node. In both cases, the key-pointer pair is temporarily stored in the variables `hold-key` and `hold-datum` and is added to the end of the `new-node` after performing the actual split.

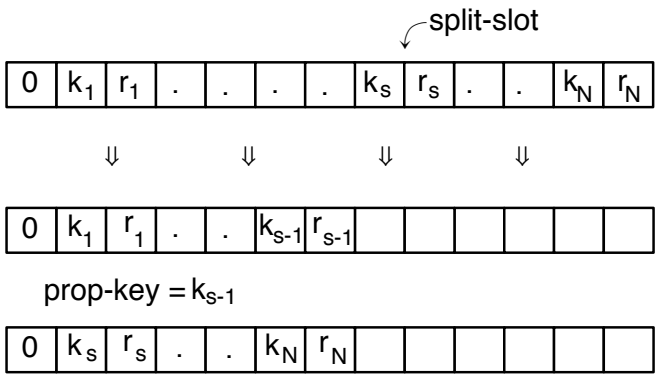

Figure 17.6: Splitting a leaf node

The body of the procedure determines the `insert-slot` as the first free index in the new node for adding the `hold-key`/`hold-datum` pair. It also determines the `prop-key` to be propagated upwards (i.e. returned by the procedure to the B^+-tree insertion algorithm). This is the key that will separate both nodes after the split (in their parent).

In order to understand the `leaf` argument of the procedure, it is important to understand that internal nodes and leaf nodes are split in a different way. At this point, it is important to keep the node layout presented in figure 17.5 in mind.

- Let us first have a look at leaf nodes. The situation is schematically depicted in figure 17.6. Let s be the number of the slot where the split is to be done, i.e. the middle slot number. In the code it is covered by the local variable `split-slot`. Since the resulting node must be a leaf node too, `fs:null-block` is stored in slot 0 of the `new-node`. Furthermore, all key-pointer pairs starting at k_s have to be copied to the new node. k_{s-1} is the key that has to be propagated upwards. It demarcates the keys of the original node in the parent node of the node to be split.
- Splitting an internal node is different and is depicted in figure 17.7. This time, k_s is the key propagated upwards. This key separates all the keys to the left of (i.e. smaller than or equal to) k_s in the original node and the keys to the right of (i.e. greater than) k_s in the new node.

17.3.5 Deletion Machinery: Borrowing and Merging Nodes

Remember from our introductory example that the B^+-tree deletion algorithm first tries to borrow a key-pointer pair from one of the node's siblings whenever

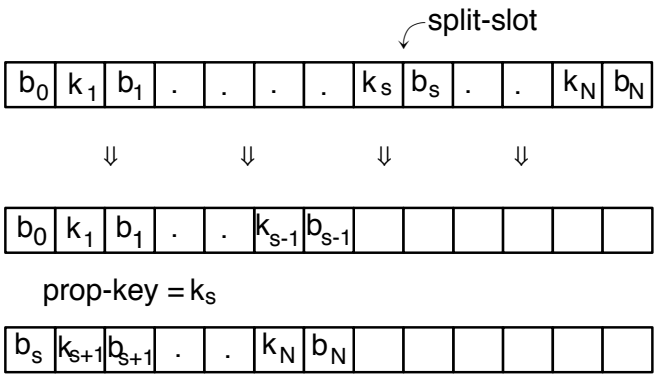

Figure 17.7: Splitting an internal node

the node runs the risk of containing less than $N/2$ entries. Only when borrowing a key from both siblings is impossible (because they both contain $N/2$ entries and thus cannot miss a pair), we resort to merging a node with one of its siblings.

Borrowing from Siblings

Borrowing a key-pointer pair is taken care of by `borrow-from-left?` and `borrow-from-right?`. We only discuss `borrow-from-right?`. `borrow-from-left?` is very similar.

```
(define (borrow-from-right? node right pull-skey)
  (define rsiz (size right))
  (define cpty (ntype:capacity (type right)))
  (define nsiz (size node))
  (if (< (- rsiz 1) (div cpty 2))
      #f
      (let* ((rkey (key right 1))
             (rptr (pointer right 1))
             (lkey (if (leaf? node)
                       rkey
                       pull-skey))
             (lptr (if (leaf? node)
                       rptr
                       (pointer right 0))))
        (key-pointer-insert! node (+ nsiz 1) lkey lptr)
        (key-pointer-delete! right 1)
        (unless (leaf? node)
          (pointer! right 0 rptr))
        rkey)))
```

`borrow-from-right?` takes a `node` and its right sibling `right`. The pro-

cedure returns #f if the right sibling runs the risk of getting underpopulated and thus cannot lend a key-pointer pair. If the sibling has enough key-pointer pairs, the borrowing process proceeds. `pull-key` is the key that separates both nodes in their parent. The procedure "pulls" this key down into the node. The procedure returns a new key `rkey` that separates the keys of both nodes *after* the borrowing process. After calling `borrow-from-right?`, it is this key that will have to be stored in the parent node as the new key that separates both nodes. In `borrow-from-right?`, the first key-pointer pair of the right sibling is added as the last key-pointer pair in the node. Notice that there is a different meaning of the phrase "first key" in the case of leaf nodes and internal nodes. This is the reason for the `if`-test in the code for binding `lkey` and `lptr`.

Merging Nodes

Finally we present `merge!`. This is the procedure that is applied by the deletion algorithm whenever borrowing an element from either sibling is impossible because both siblings are exactly half full. In that case, the contents of an `accu-node` and a `node` are merged into the `accu-node`. The `pull-skey` argument is the key that separates both nodes in their parent. Again, a subtle distinction has to be made between merging leaf nodes and merging internal nodes. We invite the reader to make the drawings for both cases.

```
(define (merge! accu-node node pull-skey)
  (define cpty (ntype:capacity (type node)))
  (define asiz (size accu-node))
  (define nsiz (size node))
  (define strt (if (leaf? node)
                   (+ asiz 1)
                   (+ asiz 2)))
  (if (not (leaf? node))
      (key-pointer! accu-node (+ asiz 1) pull-skey (pointer node 0)))
  (do ((indx strt (+ indx 1)))
    ((= (- indx strt -1) (+ nsiz 1)))
    (key-pointer! accu-node indx (key node 1) (pointer node 1))
    (key-pointer-delete! node 1)))
```

17.3.6 Paths

Remember that the actual key-pointer pairs sit in the leaf nodes of the B^+-tree. The "current" of a B^+-tree thus refers to some key-pointer pair sitting in some leaf node. In order to provide an implementation for `set-current-to-next!` we have to be able to obtain the "next" key-pointer pair of the current key-pointer pair. This is easy as long as the current key-pointer pair has a "next" key-pointer pair in the *same* node. But how can we technically find the next key-pointer pair of the last key-pointer pair of a node? It means that we have to advance the current to the leaf node immediately to the right of the current node. This is not necessary a direct sibling of the current node!

Since we do not store back pointers from nodes to their parent node (see

section 6.1.4), this would require us to search the entire tree! To avoid this, the representation of a B^+-tree in central memory maintains the path from the root of the tree down to the "current" key-pointer pair. The current key-pointer pair is identified by the last entry on that path. The "next" key-pointer pair can be either found in the same node or by switching to the next leaf node by traversing the tree in a pre-order way. This can be achieved by following the path in a LIFO-order as explained in section 6.2.1. In other words, the path of a B^+-tree is the stack that is built while descending the B^+-tree during the search process.

The following Scheme code relies on the linked implementation of the `stack` ADT presented in section 4.1. We assume it is imported using prefix `stck:`. The code is used when traversing a tree starting at the root down to the current key-pointer pair. Every time we follow a pointer in a node, we store that node and the slot chosen number on the stack. `push!` pushes the node and the slot number on the stack. `slot` and `node` are used for reading the top from the stack. `pop!` zaps the top of the stack. Finally, `clear!` forgets the entire stack by popping all its elements.

```
(define new   stck:new)

(define empty? stck:empty?)

(define pop!  stck:pop!)

(define (push! stck node slot)
  (stck:push! stck (cons node slot)))

(define (node stck)
  (car (stck:top stck)))

(define (slot stck)
  (cdr (stck:top stck)))

(define (clear! stck)
  (let loop
    ()
    (when (not (empty? stck))
      (pop! stck)
      (loop))))
```

17.3.7 B^+-Tree Algorithms

Now that we have an implementation for the `node-type`, the `node` and the `path` ADTs, we are ready to define, represent and implement the `b-tree` ADT. A `b-tree` is created by calling the following `new` constructor given a disk, a file name and two bytes. Remember that we use a `b-tree` for searching a `table` in a faster way. This means that we will use one of the table's attributes as

the key type for organising the tree. This type is the meaning of the first byte argument. Its corresponds to one of the aforementioned key types (i.e. 0, 1, 2 or 3 corresponding to `natural`, `integer`, `decimal` and `string`). The fourth argument of `new` is a byte that indicates the key size, i.e. the number of bytes needed to store a key. For example, if we have a tree that uses an attribute of type (`string` 12) then 12 will be passed as the fourth argument to the constructor. `open` opens an existing B^+-tree stored on the disk. It will read some administrative data into central memory and thus recreate a dual representation for part of the B^+-tree. `flush!` closes a B^+-tree thereby bringing the corresponding disk blocks up to date with the changes made on the dual representation that sits in central memory. `drop!` removes the entire B^+-tree from the disk.

```
1  ADT b–tree
2
3  new
4      ( disk string byte byte → b–tree )
5  open
6      ( disk string → b–tree )
7  b–tree?
8      ( any → boolean )
9  flush!
10     ( b–tree → ∅ )
11 drop!
12     ( b–tree → ∅ )
13 insert!
14     ( b–tree any rcid → status )
15 find!
16     ( b–tree any → status )
17 set–current–to–first!
18     ( b–tree → status )
19 set–current–to–next!
20     ( b–tree → status )
21 delete!
22     ( b–tree → status )
23 peek
24     ( b–tree → any × rcid )
25 update!
26     ( b–tree rcid → status )
```

`insert!` adds a new key-rcid pair to a given B^+-tree. The rcid is expected to refer to some tuple in a data file (see section 16.2.2). The B^+-tree is equipped with a "current" that is assigned a meaningful value by `find!`, by `set-current-to-first!` or by `set-current-to-next!`. The operations `peek`, `update!` and `delete!` operate w.r.t. that current. `peek` reads the key and the rcid that is pointed to by the current. `update!` overwrites the rcid associated

with the key pointed to by the current. `delete!` removes the key-rcid pair pointed to by the current.

Notice that some of these operations return a `status` value. This can be one of the following:

1 **status** = { done, no−current, next−higher, not−found }

We implement them as follows:

```
(define done 'done)
(define no-current 'no-current)
(define next-higher 'next-higher)
(define not-found 'not-found)
```

The rationale behind these status values is as follows. `no-current` is returned from operations like `set-current-to-next!`, `update!`, `peek` and `delete!` if an attempt is made to manipulate an invalidated current. In the case of `find!`, the returned status is either `not-found`, `done` or `next-higher`. The former two should be self-explanatory. The latter is returned if the search key *itself* is not found but if another key is found that is greater than the search key, i.e. if the search process did not "fall off the end" of the B^+-tree. The idea behind returning `next-higher` from `find!` is to support the implementation of range queries. For instance, the SQL-query `SELECT * FROM table WHERE attribute > 3` is implemented by using `find!` with `3` (possibly returning `next-higher` if `3` itself does not occur in the tree) and then using `set-current-to-next!` until `no-current` is returned.

Construction and Destruction

The implementation of the `b-tree` ADT follows. In central memory, a B^+-tree is represented as a Scheme record value that consists of a `disk`, the file `name` of the B^+-tree, a path containing a reference to the nodes leading to the "currrent" key-pointer pair and the `node-type` that comprise the information needed to decode the disk blocks as `node`s.

```
(define-record-type b-tree
  (make n h p t)
  b-tree?
  (n name name!)
  (h header header!)
  (p path path!)
  (t node-type node-type!))
```

On the disk, the tree is represented by a header block that is registered under the tree's file name in the directory. This header block is also kept in central memory while working with the B^+-tree. In the current implementation, the header block stores the type of the keys used in the B^+-tree (i.e. a byte that is equal to 0, 1, 2 or 3), the size of the keys (another byte) and the block number of the root node of the B^+-tree. Storing the root's disk block number is necessary since a B^+-tree grows by adding a new root node every now and then. Hence,

the block number of the root varies over time and thus needs to be encoded in the header block. Here is the Scheme code necessary to encode the header's constituents in a disk block:

```
(define keytype-size 1) ; one byte to encode the key's type
(define keysize-size 2) ; two bytes to encode the key's size

(define key-type-offset  0)
(define key-size-offset  (+ key-type-offset keytype-size))
(define root-block-offset (+ key-size-offset keysize-size))

(define (key-type indx)
  (define hder (header indx))
  (disk:decode-fixed-natural hder key-type-offset keytype-size))
(define (key-type! indx ktyp)
  (define hder (header indx))
  (disk:encode-fixed-natural! hder key-type-offset keytype-size ktyp))
(define (key-size indx)
  (define hder (header indx))
  (disk:decode-fixed-natural hder key-size-offset keysize-size))
(define (key-size! indx ksiz)
  (define hder (header indx))
  (disk:encode-fixed-natural! hder key-size-offset keysize-size ksiz))
(define (tree-root indx)
  (define hder (header indx))
  (disk:decode-fixed-natural hder root-block-offset disk:block-ptr-size))
(define (tree-root! indx root)
  (define hder (header indx))
  (disk:encode-fixed-natural! hder root-block-offset disk:block-ptr-size root))
```

Creating a new B^+-tree is accomplished by creating a new header block `hder` on the disk and by registering that header block under the file name in the directory using `fs:mk`. The rest of the code of `new` should be straightforward. We construct a node type given the key type and the key size, we construct the representation of the B^+-tree in central memory (i.e. `tree`) and we initialize the header block and write it back to the disk. We omit the implementation of `open` because it is entirely similar. Flushing a B^+-tree merely updates the header by writing the header back to the disk. This code is omitted as well.

```
(define (new disk name ktyp ksiz)
  (define ntyp (ntype:new disk ktyp ksiz))
  (define stck (path:new))
  (define hder (fs:new-block disk))
  (define tree (make name hder stck ntyp))
  (key-size! tree ksiz)
  (key-type! tree ktyp)
  (tree-root! tree fs:null-block)
```

```
(fs:mk disk name (disk:position hder))
(disk:write-block! hder)
tree)
```

Here is how to remove a B^+-tree from the disk. The code is fairly simple because it is written in a recursive way. It would be a fairly challenging exercise to implement an iterative version of this procedure. The process is accomplished by the `rec-delete` which simply deletes leaf nodes and which deletes internal nodes after having deleted all children. The process terminates by removing the header from the disk and by removing the file name from the directory. After all this has been done, Scheme is responsible for garbage collecting the dual representation in central memory.

```
(define (drop! tree)
  (define ntyp (node-type tree))
  (define root (tree-root tree))
  (define (rec-delete bptr)
    (define node (node:read ntyp bptr))
    (define head (node:pointer node 0))
    (cond ((node:leaf? node)
           (node:delete! node))
          (else
           (rec-delete head)
           (do ((slot 1 (+ slot 1)))
             ((fs:null-block? (node:pointer node slot))
              (node:delete! node))
             (rec-delete (node:pointer node slot))))))
  (if (not (fs:null-block? (tree-root tree)))
      (rec-delete (tree-root tree)))
  (fs:delete-block (header tree))
  (fs:rm (ntype:disk ntyp) (name tree)))
```

Searching a B^+-tree

We are now ready to discuss the first real B^+-tree procedure, namely `find!`. It returns one of the aforementioned `status` descriptors but it also modifies the representation of the B^+-tree in central memory by adjusting its internal path. Hence the exclamation mark in its name. `find!` has a local procedure `build-path` which iteratively constructs the path leading to the search key. `find!` starts out by clearing the path and calling `build-path` to construct the new path. In every iteration, `build-path` takes the block number of a node as an argument and reads the corresponding node from the disk. It uses the `locate-leftmost` argument to perform a binary search on the node. `complement` is used to turn the result into a positive number. If the node is not a leaf node, the node and the slot number resulting from the binary search are pushed on the path stack and the pointer is used to enter `build-path` again. If the node is a leaf node, there are three possibilities. Either the slot was found in the leaf node (i.e. a negative value was returned by `locate-leftmost`).

Then the slot number is pushed on the stack and `done` is returned. If the slot was not found in the node (i.e. a positive number ≥ 0 was returned from `locate-leftmost`) we distinguish between two cases. Either our search has lead us beyond the last meaningful key of that leaf node. In that case, the B^+-tree has guided us to the rightmost leaf node and there does not seem to exist a corresponding key. We then clear the path and return `not-found`. If the slot number resulting from the binary search *does* correspond to a slot number somewhere in the middle of the node, then the exact key does not exist in the B^+-tree, but a successor of the key was found. In that case, we return `next-higher`.

```
(define (find! tree key)
  (define stck (path tree))
  (define (build-path bptr)
    (define node (node:read (node-type tree) bptr))
    (define slot (node:locate-leftmost node key))
    (define actual-slot (node:complement slot))
    (cond
      ((node:leaf? node)
       (cond
         ((negative? slot)
          (path:push! stck node (+ actual-slot 1))
          done)
         ((node:meaningless? node actual-slot)
          (path:clear! stck)
          not-found)
         (else
          (path:push! stck node (+ actual-slot 1))
          next-higher)))
      (else
       (path:push! stck node actual-slot)
       (build-path (node:pointer node actual-slot)))))
  (path:clear! stck)
  (if (fs:null-block? (tree-root tree))
      not-found
      (build-path (tree-root tree))))
```

Notice that `find!` has a performance characteristic that is in $O(log_N(n))$ because we are mainly interested in the number of block transfers. The $O(log_2(N))$ resulting from the binary search process performed by `locate-leftmost`. The time required so search a relatively small number of keys using binary search is *much* smaller than the time required to read a block from the disk.

Manipulating the Current

Notice that `find!` stores the entire path leading to the "current" key-rcid pointer on a stack. A completely different way to manipulate the current consists of making the current refer to the very first key-rcid pair in the B^+-tree (i.e. the one that sits in the leftmost slot of the leftmost leaf node) using

`set-current-to-first!`. In both cases, we can use `set-current-to-next!` to hop from one key-rcid pair to the next.

The implementation of `set-current-to-first!` is fairly simple. It can be seen as a simplified version of `find!` that perpetually selects the zeroth block pointer when descending down the tree. All intermediate nodes (together with 0 as the slot number) are pushed on the stack. The very last time, 1 is pushed on the stack because we are dealing with a leaf node.

```
(define (set-current-to-first! tree)
  (define ntyp (node-type tree))
  (define stck (path tree))
  (define (build-path bptr)
    (define node (node:read ntyp bptr))
    (cond ((node:leaf? node)
           (path:push! stck node 1)
           done)
          (else
           (path:push! stck node 0)
           (build-path (node:pointer node 0)))))
  (path:clear! stck)
  (if (fs:null-block? (tree-root tree))
      no-current
      (build-path (tree-root tree))))
```

`set-current-to-next!` is a bit more involved. It contains two local procedures `backtrack` and `advance`. The traversal from the current key-rcid pair to its successor key-rcid pair *may* require us to hop from one leaf node to another. It is initiated by the call (`backtrack` 0). The procedure pops the top node and its corresponding slot from the stack. If this was not the last meaningful slot in that node, we advance to the right by calling (`advance node` (+ `slot` 1) `level`). Otherwise, we have visited the entire node and we backtrack one level higher in the tree by calling (`backtrack` (+ `level` 1)). Like this, several calls to `backtrack` will cause us to climb the tree until a node is found in which we *can* advance one slot to the right. As soon as `advance` is called, we descend to the level of leaves again by pushing the node and its associated slot on the stack and by keeping calling `advance` until the `level` parameter reaches 0. Notice how `advance` is called differently the very last time (i.e. at level 1): in leaf nodes slots start counting from 1 instead of 0.

```
(define (set-current-to-next! tree)
  (define stck (path tree))
  (define ntyp (node-type tree))
  (define (backtrack level)
    (define node (path:node stck))
    (define slot (path:slot stck))
    (path:pop! stck)
    (if (node:meaningless? node slot)
```

```
      (if (path:empty? stck)
          not-found
          (backtrack (+ level 1)))
      (advance node (+ slot 1) level)))
  (define (advance node slot level)
    (path:push! stck node slot)
    (if (= 0 level)
        done
        (let*
            ((bptr (node:pointer node slot))
             (next-node (node:read ntyp bptr)))
          (if (= level 1)
              (advance next-node 1 0)
              (advance next-node 0 (- level 1))))))
  (if (path:empty? stck)
      no-current
      (backtrack 0)))
```

The following two procedures operate w.r.t. the current that is defined by the path stored in the B^+-tree. They both check whether or not the B^+-tree contains a non-empty path. Should this not be the case, then `no-current` is returned as a complaint value. Otherwise, the leaf `node` and the current `slot` value are accessed from the top of the stack. `peek` reads the key-rcid pair from the slot and returns it as a Scheme pair. `update!` updates the pointer part with the given rcid and writes the updated node back to the disk. `done` is returned to indicate success.

```
(define (peek tree)
  (define stck (path tree))
  (if (path:empty? stck)
      no-current
      (let ((node (path:node stck))
            (slot (path:slot stck)))
        (cons (node:key node slot) (node:pointer node slot)))))

(define (update! tree rcid)
  (define stck (path tree))
  (if (path:empty? stck)
      no-current
      (let ((node (path:node stck))
            (slot (path:slot stck)))
        (node:pointer! node slot rcid)
        (node:write! node)
        done)))
```

Insertion

Next we describe the insertion algorithm. The `insert!` procedure contains two local procedures. `build-path` tries to locate the leaf node in which the

key-rcid pair is supposed to be inserted, thereby constructing the path from the root of the tree down to the node of the actual insertion. `build-path` keeps on descending the tree by reading a node from the disk, searching the key to be inserted in that node and following the corresponding pointer as long as the node is an internal one. As soon as the path from the root to the node has been entirely constructed, `traverse-path` traverse this path in reverse order. It inserts the key in a node and keeps on doing this — level after level — until a node is encountered in which this does *not* engender a split of the node in which the key is inserted. Detecting whether or not a node has room left to store an additional key-pointer pair is achieved by looking up the sentinel key. Remember that `locate-leftmost` returns a negative value in the case of success. If there is no more room left, a `new-node` is created and the old node is split using the new node. Both nodes are written to the disk and the `prop-key` resulting from the split is propagated upward for insertion in the node that sits one level higher in the B^+-tree (i.e. the parent of the current node). At the highest level — when the path is empty — a new root node is created that stores one key and two surrounding pointers pointing to the two nodes resulting from the split at the previous level.

```
(define (insert! tree key rcid)
  (define ntyp (node-type tree))
  (define ktyp (ntype:key-type ntyp))
  (define sent (ntype:key-sent ntyp))
  (define stck (path tree))
  (define root (tree-root tree))
  (define (build-path bptr)
    (define node (node:read ntyp bptr))
    (define slot (node:locate-leftmost node key))
    (path:push! stck node (node:complement slot))
    (if (not (node:leaf? node))
        (build-path (node:pointer node (node:complement slot)))))
  (define (traverse-path key pointer leaf?)
    (if (path:empty? stck)
      (let
          ((new-root (node:new ntyp leaf?)))
        (node:key-pointer! new-root 1 key pointer)
        (node:pointer! new-root 0 root)
        (node:write! new-root)
        (tree-root! tree (node:position new-root)))
      (let*
          ((node (path:node stck))
           (slot (path:slot stck))
           (boundary (node:locate-leftmost node sent)))
        (path:pop! stck)
        (cond
          ((negative? boundary)
```

```
          (node:key-pointer-insert! node (+ slot 1) key pointer)
          (node:write! node))
         (else
          (let*
              ((new-node (node:new ntyp leaf?))
               (prop-key
                (node:key-pointer-insert-split!
                 node new-node (+ slot 1) key pointer leaf?)))
            (node:write! node)
            (node:write! new-node)
            (traverse-path prop-key (node:position new-node) #f)))))))
  (path:clear! stck)
  (if (not (fs:null-block? root))
    (build-path root))
  (traverse-path key rcid #t)
  (path:clear! stck)
  done)
```

Since `insert!` first constructs a path (requiring at most $O(log_N(n))$ block transfers) and then traverses the path in reverse order, we can expect a total of $O(log_N(n))$ block transfers.

Deletion

Finally, we study the deletion procedure. The procedure assumes that a B^+-tree has a valid "current" in the form of a path. This means that `delete!` can only be called after calling `find!`, `set-current-to-first!` or `set-current-to-next!`. `delete!` calls `do-delete-key-pointer!` which removes the key-rcid pair pointed to by the current. If this leaves the leaf node with more than $N/2$ key-rcid pairs, the process terminates after writing the node back to the disk. However, if the leaf node suffers from an underflow after deleting the pair, then `traverse-path` is called to rebalance the B^+-tree.

```
(define (delete! tree)
  (define ntyp (node-type tree))
  (define stck (path tree))
  (define (traverse-path node)
    (if (path:empty? stck)
        (cond ((= (node:size node) 0)
               (tree-root! tree (node:pointer node 0))
               (node:delete! node))
              (else
               (node:write! node)))
        (let* ((prnt-node (path:node stck))
               (prnt-slot (path:slot stck))
               (left-sibl (and (< 0 prnt-slot)
                               (node:read ntyp
                                          (node:pointer prnt-node (- prnt-slot 1)))))
               (push-lkey (and left-sibl
```

```
                        (node:borrow-from-left?
                         left-sibl node (node:key prnt-node prnt-slot))))
            (rght-sibl (and (not push-lkey)
                        (< prnt-slot (node:size prnt-node))
                        (node:read ntyp
                                   (node:pointer prnt-node (+ prnt-slot 1)))))
            (push-rkey (and rght-sibl
                        (node:borrow-from-right?
                         node rght-sibl (node:key prnt-node (+ prnt-slot 1))))))
        (path:pop! stck)
        (cond ((or push-lkey push-rkey) ; we managed to borrow a slot =¿ finish up
               (cond (push-lkey
                      (node:write! left-sibl)
                      (node:key! prnt-node prnt-slot push-lkey))
                     (else
                      (node:write! rght-sibl)
                      (node:key! prnt-node (+ prnt-slot 1) push-rkey)))
               (node:write! node)
               (node:write! prnt-node))
              (else                ; no borrowing from siblings =¿ merge + recurse
               (cond (left-sibl
                      (node:merge!
                       left-sibl node (node:key prnt-node prnt-slot))
                      (node:write! left-sibl)
                      (node:delete! node)
                      (do-delete-key-pointer! prnt-node prnt-slot))
                     (rght-sibl
                      (node:merge!
                       node rght-sibl (node:key prnt-node (+ prnt-slot 1)))
                      (node:write! node)
                      (node:delete! rght-sibl)
                      (do-delete-key-pointer! prnt-node (+ prnt-slot 1)))))))))
(define (do-delete-key-pointer! node slot)
  (node:key-pointer-delete! node slot)
  (if (< (node:size node) (div (node:capacity node) 2))
    (traverse-path node)
    (node:write! node)))
(if (path:empty? stck)
    no-current
    (let ((node (path:node stck))
          (slot (path:slot stck)))
      (path:pop! stck)
      (do-delete-key-pointer! node slot)
      (path:clear! stck)
      done)))
```

`traverse-path` perpetually pops the parent node `prnt-node` and the corresponding slot `prnt-slot` from the stack. It checks wether the node suffering from underflow has a left sibling. If this is the case, the left sibling is read from the disk and an attempt is made to borrow a key-pointer pair from the left sibling. If the node has no left sibling or if the borrowing failed (indicated by the fact that `push-lkey` is #f), then an attempt is made to borrow a key-pointer pair from the right sibling if such a right sibling exists. If this fails as well, `push-rkey` is also #f. If either borrowing from the left sibling or right sibling was successful, the sibling and the node are written back to the disk and the parent node is written back to the disk after having updated the key that separates the keys in the node from the keys in the sibling. If the borrowing attempt failed twice, the node is merged with the left sibling or with the right sibling should there be no left sibling (because the node itself was the leftmost child of its parent node). After the merge, one of the nodes is written back to the disk and the other node is returned to the file system. In that case however, we have to recursively delete the key that separates the node and the sibling from the parent node. This is why `do-delete-pointer!` is called again. `delete!` requires $O(log_N(n))$ block transfers for exactly the same reason as `insert!` and `find!`.

17.4 The Database ADT

We have started chapter 16 with a summary of the relational model. We have discussed the usage of scheme procedures `create-table`, `create-index!`, `insert-into-table!` and so on. All such procedures operate on a database which is just a collection of tables together with some index files that can speed up the access to the tuples of the tables. More technically, a database is a collection of `table`s and a collection of `b-tree`s. Obviously we also need a way to group these together on a disk so that we know which index files belong to which data files. This is why we need a `database` ADT. It was presented in section 16.1.4. A first reaction may consist of creating yet another external data structure to implement the `database` ADT; a data structure that groups together a table with its indexes.

A key insight is that we can store that information in a suite of tables as well! Two such *meta tables* are used to store the information about all tables in a database. One table stores the file name of each table in our database, together with some automatically generated identification number for that table. The second table stores the file names of all the indexes that are defined on a table that goes with a particular identification number. Both tables are depicted in figure 17.8. The figures shows a population of the meta tables that corresponds to the example given in section 16.1. We have two relational tables, one is called "Planets" and the other is called "Moons". The system has assigned them the ID-numbers 0 and 1 upon calling `create-table`. The database also has two index files, both defined on the "Moons" table (indicated by ID-number 1). The first index is called "Name-Idx" and is defined on column number 0. The second

index is called "Orbital-Idx" and is defined on column number 5.

table-name	id-number
Planets	0
Moons	1

id-number	index-name	column-number
1	Name-Idx	0
1	Orbital-Idx	5

Figure 17.8: The Meta Tables' Schema

17.4.1 The Meta Schema

In order to create the meta tables, we need a schema as well. That schema is known as the *meta schema* of a database. The tables depicted in figure 17.8 are defined using the following meta schemas. `meta-schema:table` shown below consists of a string attribute and a natural number attribute (i.e. the identification number). `meta-schema:indexes` consists of a natural number attribute (the identification), a string attribute (an index file name) and another natural number attribute (the column number on which the index is defined). Remember that `fs:` is the prefix using which we usually import the file system. The code uses quasiquoting in Scheme: a back-quote is exactly the same as an ordinary quote except that we can unquote certain identifiers using a comma. The unquoted identifiers are ordinary Scheme identifiers that have to visible in the scope of the quasiquoted expression.

```
(define meta-schema:table `((string ,fs:filename-size)
                            (natural 2)))
(define table:table-name   0)
(define table:table-id     1)

(define meta-schema:indexes `((natural 2)
                              (string ,fs:filename-size)
                              (natural 2)))
(define indexes:tble-idty 0)
(define indexes:index-name 1)
(define indexes:key-att   2)
```

17.4.2 Dual Representation

Let us now implement the `database` ADT of section 16.1.4. Once again, a `database` is a data structure that has a dual representation. In our central memory it is represented by a Scheme record that contains two `table`s, namely the aforementioned meta tables. Since these tables are themselves data structures with a dual representation, it follows that the database is automatically represented by means of two tables that effectively exist on the disk.

```
(define-record-type database
  (make t i)
  database?
  (t tables)
  (i indexes))
```

The Scheme definitions for creating a new and opening an existing `database` look as follows. All we need to do is create (or open) the meta tables using the schema information just presented. `name` is the name of the database.

```
(define (new disk name)
  (define tbls (tbl:new disk (string-append "TBL" name) meta-schema:table))
  (define idxs (tbl:new disk (string-append "IDX" name) meta-schema:indexes))
  (make tbls idxs))

(define (open disk name)
  (define tbls (tbl:open disk name))
  (define idxs (tbl:open disk name))
  (make tbls idxs))
```

17.4.3 Meta Machinery

Now we need a bootstrapping phase. Our ultimate goal is to implement procedures such as `select-from/eq` using the information stored in the meta tables. However, in order to extract that information, we need exactly the same operation! We will therefore need to implement some elementary meta machinery (which is hard coded in Scheme) in order to bootstrap the implementation.

We start by implementing `find-id-in-meta-table` which takes a database and a table that belongs to that database. It traverses the first meta table of the database in a linear way (using `set-current-to-first!` and `set-current-to-next!` until the name of the argument table is encountered. The associated identification number `tble-idty` is returned.

```
(define (find-id-in-meta-table dbse tabl)
  (define name (tbl:name tabl))
  (define tbls (tables dbse))
  (tbl:set-current-to-first! tbls)
  (let loop
    ((tuple (tbl:peek tbls)))
    (let ((tble-name (car tuple))
          (tble-idty (cadr tuple)))
      (cond ((string=? tble-name name)
             tble-idty)
            ((not (eq? (tbl:set-current-to-next! tbls) no-current))
             (loop (tbl:peek tbls)))
            (else
             not-found)))))
```

`for-all-tables` takes a database schemedbse and a Scheme procedure `proc`. It traverses the first meta table and it applies the procedure to all the tables that are registered in that meta table. It keeps on traversing the meta table as long as the `proc` does not return #f. Notice that `and` is a lazy special form.

```
(define (for-all-tables dbse proc)
  (define tbls (tables dbse))
  (define disk (tbl:disk tbls))
  (when (not (eq? (tbl:set-current-to-first! tbls) no-current))
    (let loop
      ((tuple (tbl:peek tbls)))
      (let ((tabl (tbl:open disk (list-ref tuple table:table-name))))
        (if (and (proc tabl)
                 (not (eq? (tbl:set-current-to-next! tbls) no-current)))
            (loop (tbl:peek tbls)))))))
```

`for-all-indices` has a similar effect on the indexes that are defined on a table. These indexes sit in the second meta table and can be identified by the identification number of their table (which is retrieved from the first meta table using `find-id-in-meta-table`). The procedure argument `proc` is applied to all indexes as long as the procedure does not return #t. Obviously, this is a very inefficient procedure if we are dealing with a large meta table that occupied many disk blocks. Fortunately, meta tables are usually small.

```
(define (for-all-indices dbse tble proc)
  (define idxs (indexes dbse))
  (define disk (tbl:disk idxs))
  (define idty (find-id-in-meta-table dbse tble))
  (when (not (eq? (tbl:set-current-to-first! idxs) no-current))
    (let loop
      ((tuple (tbl:peek idxs)))
      (cond ((= (list-ref tuple indexes:tble-idty) idty) ; the index belongs to the table idty
             (let ((indx (btree:open disk (list-ref tuple indexes:index-name))))
               (if (and (proc indx (list-ref tuple indexes:key-att))
                        (not (eq? (tbl:set-current-to-next! idxs) no-current)))
                   (loop (tbl:peek idxs)))))
            ((not (eq? (tbl:set-current-to-next! idxs) no-current))
             (loop (tbl:peek idxs)))))))
```

Finally, we present `for-all-tuples` which takes a table and a procedure `proc`. It reads the tuples from the table one by one, and it keeps on applying the procedure as long as it does not return #f.

```
(define (for-all-tuples table proc)
  (if (not (eq? (tbl:set-current-to-first! table) no-current))
      (let loop
        ((tuple (tbl:peek table)))
        (let ((curr (tbl:current table)))
```

```
      (if (and (proc tuple curr)
               (not (eq? (tbl:set-current-to-next! table) no-current)))
          (loop (tbl:peek table)))))))
```

17.4.4 Table Creation and Index Creation

Now we are ready to present the implementation of the `database` ADT. Here is `create-table`. It takes a database, a table name and an association list (such as the `planets-schema` presented in section 16.1) that represents the schema of the newly created table. `create-table` creates the table using the `table` ADT and registers the newly created table in the first meta table under a newly generated identification number[4].

```
(define (create-table dbse name scma)
  (define tbls (tables dbse))
  (define disk (tbl:disk tbls))
  (define tble (tbl:new disk name scma))
  (define idty (gennum))
  (tbl:insert! tbls (list name idty))
  tble)
```

`create-index!` takes a database, a table, the name for a new index file and the attribute number (i.e. the column number) for the table on which the index is supposed to operate. It looks up the table in the first meta table, it creates the index (using the `new` constructor of the `b-tree` ADT) and it registers the index in the second meta table. Notice that we have to traverse all tuples of the table in order to insert them in the newly created index! If we need B blocks to store the data file that represents the table, then `create-index!` uses $O(B)$ block transfers.

```
(define (create-index! dbse tabl name attribute)
  (define disk (tbl:disk tabl))
  (define tbls (tables dbse))
  (define idxs (indexes dbse))
  (define idty (find-id-in-meta-table dbse tabl))
  (define scma (tbl:schema tabl))
  (define indx (btree:new disk name
                          (scma:type scma attribute)
                          (scma:size scma attribute)))
  (tbl:insert! idxs (list idty name attribute))
  (for-all-tuples
   tabl
   (lambda (tuple rid)
     (btree:insert! indx (list-ref tuple attribute) rid)))
  (tbl:close! idxs)
  (btree:flush! indx))
```

[4] We use `gennum` for this.

17.4.5 Insertion and Deletion

Now we are ready to populate the tables of a database. `insert-into-table!` takes a table belonging to a database as well as a tuple. It inserts the tuple in the table (which requires $O(1)$ block transfers) and in all index files that are defined on that table in the database (which all require $O(log(B))$ block transfers). Hence, the operation is in $O(log(B))$. Strictly spoken, it would be more correct to speak about $O(I \times log(B))$ where I is the number of index files defined on the table.

```
(define (insert-into-table! dbse tble tuple)
  (define rcid  (tbl:insert! tble tuple))
  (tbl:close! tble)
  (for-all-indices dbse tble
                   (lambda (indx att)
                     (btree:insert! indx (list-ref tuple att) rcid)
                     (btree:flush! indx))))
```

We omit the implementation of `delete-where!` because it is very similar to `select-from/eq`. After all, deleting (a) tuple(s) consists of efficiently locating the tuple(s) after which it has (resp. they have) to be deleted from the data file and from all index files that are defined on that data file. The bulk of the work is done in locating the tuple(s).

17.4.6 Equality Queries

Finally, we present the long awaited `select-from/eq`. It takes a table that belongs to a database. It also takes a column number `attr` and a value `valu`. It returns a Scheme list containing *all* the tuples for which the column value is indeed equal to `valu`.

```
(define (for-all-identical-keys indx eqls valu proc)
  (let loop
    ((cur? (eq? (btree:find! indx valu) done)))
    (if cur?
      (loop (and (proc (cdr (btree:peek indx)))
                 (eq? (btree:set-current-to-next! indx) done)
                 (eqls (car (btree:peek indx)) valu))))))

(define (select-from/eq dbse tble attr valu)
  (define scma (tbl:schema tble))
  (define type (scma:type scma attr))
  (define eqls (vector-ref equals type))          ;right equality procedure
  (define indx ())
  (define rslt ())
  (for-all-indices dbse tble (lambda (idx att)     ;first try to find an index on 'attr'
                               (when (= att attr)
                                 (set! indx idx)
```

```
                                        #f)))
  (if (null? indx)        ; index on 'attr' found, or search the tuple file sequentially
    (for-all-tuples tble (lambda (tple rcid)
                           (if (eqls (list-ref tple attr) valu)
                             (set! rslt (cons (tbl:peek tble) rslt)))))
    (for-all-identical-keys indx eqls valu
                            (lambda (rcid)
                              (tbl:current! tble (cdr (btree:peek indx)))
                              (set! rslt (cons (tbl:peek tble) rslt)))))
  rslt)
```

The implementation first tries to find an index file on the specified column. It does so by enumerating all the indexes that are defined for that particular table. As soon as an index on the specified `attr` is encountered, it is registered in the variable `indx`. If no such index is found, we use `for-all-tuples` to traverse the entire table and gather all tuples that meet the requirement in the variable `rslt`. This is bound to take $\Theta(B)$ amount of work. Otherwise we execute a `find!` operation on the index file (which requires $O(log(B))$ work and we use the current to traverse the tuples as long as their column value is indeed equal to the specified `valu`. This takes at most $O(B)$ block transfers. This is taken care of by `for-all-identical-keys`.

Notice that we need to be careful since every attribute value type needs a different equality test. This is taken care of by the `equals` vector that was already shown before:

```
(define equals   (vector = = string=?))
```

17.5 Summary

In this and the previous chapter, we have studied the implementation of a small relational database system. The previous chapter has implemented the `table` ADT as an abstraction for storing tuples (that fit some particular schema) in a collection of (double linked) disk blocks. This implementation has very attractive performance characteristics for insertion and deletion. Unfortunately this is not the case for data retrieval which requires $O(B)$ disk block transfers on the average. To alleviate this problem, this chapter has studied the `b-tree` ADT which defines a data structure which is "external" to a data file and which allows locating a data value in just a logarithmic number of disk block transfers. Moreover, the base of the logarithm is roughly equal to the number of keys that fit in a disk block. For example, consider a realistic setting where disk blocks contain $4Kbytes$, where we use strings of 20 characters as keys and where 4 bytes are needed to store a disk pointer. This allows us to store more than 150 key-pointer pairs in one disk block which results in a B^+-tree that is less than 5 levels deep even if we have to index 10^{10} rcids. This is a truly amazing property of B^+-trees!

17.6 Exercises

1. (a) The current implementation of B^+-trees allows us to directly find the tuple with the smallest key by means of the `set-current-to-first!` operation. However, it is currently not possible to find the tuple with biggest key in a similar fashion. Make this possible by adding a `set-current-to-last!` operation to the B-tree ADT.

   (b) Using `set-current-to-first!` and `set-current-to-last!` we can quickly find the lowest or highest value of an indexed attribute in a table (e.g. SQL queries such as "SELECT MIN(score) FROM students" or "SELECT MAX(score) FROM students"). To answer such queries for non-indexed attributes we have to traverse *all* tuples in order to compare their values for the attribute. Extend the `database` ADT with two new query procedures (`select-from/min` and `select-from/max`) that represent such queries. Make sure that the queries also work for non-indexed attributes, albeit inefficiently.

2. (a) The current implementation of B-trees allows us to navigate through an index from lower to higher keys using `set-current-to-next!`. However, it is currently not possible to navigate in reversed order (i.e., from higher to lower keys). Make this possible by adding a `set-current-to-previous!` operation to the B-tree ADT.

   (b) Using the combination of either `set-current-to-first!` and `set-current-to-next!` or `set-current-to-last!` and `set-current-to-previous!` we can extend our database system with support for queries which return all tuples of a table in sorted order (ascending or descending) by the values of an indexed attribute (e.g. SQL queries such as "SELECT * FROM students ORDER BY lastname ASC" or "SELECT * FROM students ORDER BY score DESC"). Add and implement a new query operation (`select-from/all/ordered`) to the `database` ADT. Allow the choice between ascending or descending order to the user by making the procedure expect a boolean parameter (`asc?`). You only have to make this operation work for indexed attributes.

3. The `database` ADT allows us to select tuples based on the equality of an particular attribute to a specified value by using the `select-from/eq` operation. A type of query which is currently not supported by our database system is one which selects tuples based on a value range for a particular attribute. Such a range could be specified using a lower bound (e.g. "SELECT * FROM students WHERE score $>$ 10"), an upper bound (e.g. "SELECT * FROM students WHERE score $<$ 10") or both (e.g. "SELECT * FROM students WHERE score $>$ 10 AND score $<$ 16").

   (a) As a first step, extend the `database` ADT with a `select-from/range/incl` operation which takes a lower (`lova`) and an upper bound (`hiva`) – either one of which can be () to indicate the fact that there is no

bound – and treats them as **inclusive** bounds ($\leq$ / $\geq$). Make sure that the range query can be applied to both indexed and as well as non-indexed attributes.

(b) Extend your solution (renamed as `select-from/range`) with support for **exclusive** bounds ($<$ / $>$). Allow the user to choose between the inclusiveness or exclusiveness of the bounds by adding two boolean parameters (`loi?` and `hii?`).

Chapter 18

Automatic Garbage Collection Algorithms

The previous chapter has presented a number of techniques that can be used to organise a memory consisting of a continuous row of numbered cells into a set of free and occupied chunks (i.e. vectors). The occupied chunks somehow make sense to an application. The memory managers presented in the previous chapter start from the assumption that the application is written such that it will *manually* notify the memory manager whenever it has a reference to a chunk of memory that is no longer needed. When this happens, the memory manager can reclaim that chunk and incorporate it back into the space of free memory.

Unfortunately the latter is easier said than done. Whenever the application program forgets (as a result of a bug in its application logic) to release a chunk of memory that it no longer needs, the chunk is said to be *leaked*. When this is the case, we speak of a *memory leak*. It has been shown that more than half of the bugs in applications have to do with memory leaks. It is therefore not a luxury problem to seek for methods that avoid memory leaks. This can be done by augmenting the memory manager with a suite of techniques that *automatically* determine whether or not a certain chunk is still in use by the application. Whenever it can determine that a chunk is no longer needed, the chunk is called *garbage* and can be automatically reclaimed by the memory manager. The procedures of the memory manager that are responsible for this matter are referred to as *automatic storage reclamation systems* or as *garbage collectors*.

A central concept in garbage collection is the *root set*. The root set of an application program is the set of addresses that are directly referred to by that program. This is not necessarily the same as the set of memory locations that are used by the application. E.g., in a Scheme interpreter, at any time during the execution, the root set is the set of memory locations that are referred to from within "the current environment". However, these memory locations might contain references to pairs and vectors that refer to other pairs and vectors in

their turn. Since the latter are still *reachable* from the root set, they are not garbage. In brief, the memory that is still in use by an application is precisely the memory that can be directly or indirectly reached by following pointers starting at the root set. Determining whether a chunk of memory is garbage then boils down to checking whether that chunk is (indirectly) reachable from the root set. Whenever this is *not* the case, the chunk can be safely reclaimed since there is no way the application can ever re-establish a reference to it.

Hence, determining or keeping track of reachability (i.e. "which part of the program refers to what") is an essential ingredient of any garbage collector. A crucial assumption to be made is that the application program doesn't fiddle with the memory without the automatic memory manager being aware of this. In other words, manually manipulating pointers (such as is done in languages like C++, C and Pascal) is not reconcilable with automatic storage reclamation[1].

As is the case with non-automatic memory management techniques, there is a substantial difference in complexity between memory management techniques for memory chunks of fixed size and techniques for managing variable length chunks. We will therefore present every garbage collection technique by first looking at a variant for Scheme pairs and later on generalise the technique Scheme vectors. We start by presenting the two ADTs that will be implemented several times, to wit `pair` and `vector`.

18.1 Two Abstract Models of Computer Memory

As was the case for manual memory management, a taxonomical distinction exists between managing blocks of fixed size and managing blocks of variable size. This distinction is epitomised by Scheme's pairs and vectors. Hence, the rest of the chapter "implements" pairs and vectors as an ADT in Scheme. In this section we present both ADTs together with a layer of Scheme code that implements the ADTs in terms of more primitive operations. Various implementations of those primitive operations constitutes the remainder of the chapter.

18.1.1 A Memory of Pairs: The `pair` ADT

The following ADT definition presents the `pair` ADT. Even though the ADT is built into Scheme, we present several implementations in Scheme. In order to do so, we use the meta-circular approach. Hence, we rely on Scheme's memory management in order to implement Scheme's memory management. This allows us to focus on the essence of the algorithms without having to "descend" down to the level of bits and bytes. Since we are implementing Scheme functions such

[1] There *is* the famous Boehm - Demers - Weiser garbage collector which simply treats everything it encounters as a potential address. Such *conservative* collection is guaranteed not to reclaim any memory that is still in use. The catch is that not all garbage is necessarily reclaimed.

as `cons`, `car` and `cdr` in Scheme itself, we must make sure to cleanly separate the `cons` we are implementing from the built-in `cons` that we are using for the implementation. This is accomplished by prefixing the latter with `scheme:` giving rise to identifiers such as `scheme:cons`, `scheme:pair?`, etc.

```
ADT pair

null
    pair
cons
    ( any any → pair )
pair?
    ( any → boolean )
car
    ( pair → any )
cdr
    ( pair → any )
set−car!
    ( pair any → ∅ )
set−cdr!
    ( pair any → ∅ )
root!
    ( pair → ∅ )
```

We present no less than three implementations of this ADT. However, some of the code is shared by all three implementations. We continue discussing this code here. In all three cases, our pairs will represented as *tagged addresses*. The addresses correspond to locations in the "car memory" and "cdr memory". The tags are used to distinguish pairs from other (e.g. atomic) data. In a real low-level implementation, tagged addresses have to be implemented by means of adding some bit pattern to an address. In Scheme, we conceive them by pairing the address with a symbol.

```
(define pair-tag 'pair)

(define (tag addr)
  (scheme:cons pair-tag addr))

(define (untag pair)
  (if (scheme:pair? pair)
      (scheme:cdr pair)
      (error "pair expected" pair)))
```

The procedures `tag` and `untag` are used to convert naked addresses (i.e. numbers) into pairs by pairing them with the 'pair tag and the other way around. Given an appropriate implementation for `car-peek`, `cdr-peek`, `car-poke!`, `cdr-poke!`, `out-of-memory?`, `gc` and `allocate`, the ADT can now be implemented as follows:

```
(define (pair? exp)
  (and (scheme:pair? exp)
       (eq? (scheme:car exp) pair-tag)))

(define (car pair)
  (car-peek (untag pair)))

(define (cdr pair)
  (cdr-peek (untag pair)))

(define (set-car! pair car)
  (car-poke! (untag pair) car))

(define (set-cdr! pair cdr)
  (cdr-poke! (untag pair) cdr))

(define (cons car cdr)
  (if (out-of-memory?)
    (gc))
  (if (out-of-memory?)
    (error "storage overflow" cons))
  (let ((addr (allocate)))
    (car-poke! addr car)
    (cdr-poke! addr cdr)
    (tag addr)))
```

Our memory managers for pairs will store the pairs in two "memory banks", i.e. two vectors one of which contains all the cars and the the other of which contains all the cdrs. A pair is then identified by an vector index that corresponds to the address of the pair. The vectors representing the banks are called `car-mem` and `cdr-mem`. They are not shown in the memory manager code below because the various garbage collection algorithms require them to be initialised in different ways. The above implementation of `car`, `cdr`, `set-car!` and `set-cdr!` show how cars and cdrs are accessed and mutated by using their untagged address in order to access the corresponding memory bank.

The most interesting procedure is `cons`. It checks whether we have run out of memory by calling `out-of-memory` which is not shown as it also depends on the particular memory management algorithm used. When running out of memory, `gc` is called which is an implementation of the garbage collection algorithm at hand. If no pairs have been reclaimed after a call to the garbage collector, we have definitively run out of memory and therefore an error message is generated. In case memory *is* available, a new pair is allocated and properly initialised. The pair is represented as a tagged address.

18.1.2 A Memory of Vectors: The `vector` ADT

Managing pairs will be quite simple since they all have the same length. Indeed, traversing a structure of homogeneous components can be done by (recursively

or iteratively) performing the same (simple) operation in every component. Think for example about traversing a binary tree using inorder, preorder or postorder traversal.

As we will see, the algorithms are much more complex as soon as the various components that make up our memory can be of different sizes. This is the type of memory that is managed e.g. by a Java virtual machine or by a JavaScript interpreter since all the living objects in such a system typically have a different size[2]. In Scheme, memory chunks of various sizes can be thought of as vectors. That is why we focus on implementing an automatic memory management system for the following `vector` ADT.

```
1  ADT vector
2
3  make-vector
4      ( number → vector )
5  vector-set!
6      ( vector number any → ∅ )
7  vector-ref
8      ( vector number → any )
9  vector-length
10     ( vector → number )
11 vector?
12     ( any → boolean )
13 root!
14     ( vector × ( vector → ) → ∅ )
```

`make-vector` is the constructor of the ADT. It takes a number (the size of the vector to be built) and it creates a new vector of the requested size. The semantics of `vector-set`, `vector-ref` and `vector-length` should be self-explaining. The ADT allows us to allocate several vectors, one after the other. Surely, vectors can refer to other vectors. They can even refer to themselves. As a result a myriad of vectors arises. The `root!` procedure allows us to register one vector as the root from which all meaningful vectors can be accessed indirectly. Vectors that cannot be accessed (directly or indirectly) starting at that root, are considered garbage and will be reclaimed. In a typical Scheme implementation, the evaluator itself is responsible for registering "its" root with the garbage collector. The procedural type of `root!` is explained further below.

As with pairs, the above abstractions can be implemented independently of the garbage collection algorithm used. In other words, the following code is shared among all approaches that will be studied. It relies on lower level procedures that will be foreseen by all three implementations.

The memory managers manage a soup of address that are represented as a Scheme vector. In the meta-circular approach, it represents the memory being managed. A vector is thus nothing more but a (tagged) index in this soup. The

[2]In Java, the size of the objects can be calculated statically by inspecting the classes. In JavaScript this is not possible because JavaScript does not have classes.

machinery for constructing tagged addresses is common to all three implementations and is trivial:

```
(define vector-tag 'vector)

(define (tag addr)
  (cons vector-tag addr))

(define (untag vctr)
  (if (vector? vctr)
    (scheme:cdr vctr)
    (error "vector expected" vctr)))
```

Now we are ready to present the ADT as a small layer of veneer on top of the more low level memory management procedures `allocate`, `peek`, `poke!` and the constant `overhead`:

```
(define (vector? any)
  (and (scheme:pair? any)
       (eq? (car any) vector-tag)))

(define (make-vector leng)
  (define addr (allocate (+ leng overhead)))
  (do ((index overhead (+ index 1)))
    ((>= index (+ leng overhead)))
    (poke! (+ addr index) null))
  (tag addr))

(define (vector-set! vctr indx any)
  (define addr (untag vctr))
  (define leng (- (peek addr) overhead))
  (if (or (< indx 0) (>= indx leng))
    (error "illegal index" vector-set!))
  (poke! (+ addr indx overhead) any))

(define (vector-ref vctr indx)
  (define addr (untag vctr))
  (define leng (- (peek addr) overhead))
  (if (or (< indx 0) (>= indx leng))
    (error "illegal index" vector-ref))
  (peek (+ addr indx overhead)))

(define (vector-length vector)
  (define addr (untag vector))
  (- (peek addr) overhead))
```

The constructor `make-vector` has the same semantics as its built-in equivalent. It relies on `allocate` and `overhead`, the implementation of which depends

on the particular garbage collection technique used. `overhead` is a constant (that depends on the implementation) which indicated the number of bookkeeping memory locations that are required by a particular garbage collection algorithm. E.g., we will typically store the length of each vector in a "hidden" slot of each vector. However, some implementations may store additional hidden bookkeeping information in a vector. The implementations of `vector-length`, `vector-ref` and `vector-set!` should be self explaining. They use the untagged address of a vector and take into account that the first `overhead` slots should be semantically skipped because they store the bookkeeping information.

18.1.3 Summary

In this section, we have presented two ADTs, namely `pair` and `vector` which epitomise the more general problem of managing memory chunks of fixed length and variable length. We will now present three automatic memory management techniques and illustrate them by means of implementations for both ADTs. In all three cases, the implementation for `pair` will be a simplified (and thus didactic preparation) for the implementation for `vector`.

18.2 Reference Counting

Reference counting is a technique which we will not study in great technical detail. Remember that the central question to be answered by a garbage collector is whether or not the application program is (indirectly) referring to a given chunk of memory. The idea of reference counting is to equip every chunk of memory with a hidden counter (known as *the reference count*) which counts the number of references that point to that chunk. This is accomplished as follows:

- When the chunk is allocated by a request from the application program, the memory manager initializes its reference count to 1 since only one reference to the new chunk exists.
- Whenever the application program executes an assignment

  ```
  (set! q p)
  ```

  it needs to increment `p`'s reference count by one since after executing the expression, `q` will refer to the same chunk of memory `p` is referring to. Furthermore, it needs to decrement `q`'s reference count by one, since the chunk is no longer referred to be the variable `q`. Subsequently, the memory manager will check wether the chunk `q` originally referred to has a zero reference count. When this is the case, no one is referring to the chunk anymore such that it can be reclaimed (e.g. added to the freelist).
- Adding a chunk to the freelist is a bit more complicated than one might expect. Since the chunk itself can refer to a bunch of other chunks, one must make sure that the chunk that is added to the freelist doesn't refer to

those chunks anymore. Removing these references will cause the reference counts of those chunks to decrement by one. If their reference count reaches zero, than these chunks must be (recursively) added to the freelist in their turn. In the worst case, this can cause the entire memory to be added to the freelist. Obviously, this can be a costly operation.

The effort of recursively deleting chunks can be spread over the entire computation by using a technique called *lazy deletion*. The idea is to add the freed chunk to the freelist without removing the references it still holds. Whenever the chunk is re-allocated later on (and removed from the freelist), the references inside the chunk are inspected and the same technique is used as in standard deletion. Remember from section **??** that the freelist is built by using one of the chunk's fields in order to store the next pointer. Obviously, this cannot be combined with lazy deletion since this would ruin one of the references the chunk was referring to. This is easily fixed by using the reference count in order to store the next pointer. This remains zero while the chunk remains in the freelist anyhow.

The reference counting technique is quite simple to implement and used to be popular with Lisp and Smalltalk systems. Another major advantage of reference counting is that the cost of automatic memory management is uniformly spread over the entire computation. Reference counting is said to be an *incremental garbage collection* technique. As we will see below, other garbage collection techniques centralise the cost to a single moment (usually allocation of a chunk) which causes the application program to be temporarily suspended while the garbage collector is working.

The main drawback of reference counting is that it cannot handle cycles. Given two chunks of memory that refer to each other but which are not referred to by any other chunk in memory. Since they are not referred to by any other chunk in memory, they should be collected. However, since they still refer to each other, their reference count will be 1 which will prevent them from being collected. This problem is unsolvable in general. Nevertheless, reference counting is still useful in systems that avoid cyclic data structures (such as certain embedded systems) and in systems that exhibit a very disciplined creation of cycles (such as functional programming languages).

18.3 Stop-and-Copy Collectors

A second simple garbage collection technique is known as *stop-and-copy garbage collection* or *scavenging*. The idea of this technique is to split the available memory into two *semi-spaces* and to select one of them as the working memory. We now study this idea for pairs as well as for vectors. Both implementations allocate chunks in a "leftmost" way. Upon every allocation, a `next-free` pointer shifts to the right. As soon as the `next-free` pointer has reached `memory-size` (i.e. the upper limit of the memory), the working memory is considered exhausted and garbage collection is required. We use the root set to traverse

all reachable memory chunks. While doing this, the memory is copied to the other semi-space. Then the roles of the semi-spaces are swapped. Whenever the original working memory contained some garbage chunks (i.e. whenever not everything was reachable from the root set), this implies that not everything will be copied. Since less is copied, this will probably leave us with enough free memory in the target semi-space.

18.3.1 Implementation for pairs

The following code shows the declaration of a number of constants to be used by the memory manager. `memory-size` is the total number of pairs that can be used by the application program. It indicates the size of each semi-space. `next-free` indicates the separation of the free pairs (sitting to its right) from the ones that are occupied or garbage (sitting to its left). `root` is the access to the root that will be used by the garbage collection algorithms. When set using the procedure `root!`, it defines the set of reachable cells.

```
(define memory-size 10)
(define null ())

(define car-memory (make-vector memory-size null))
(define cdr-memory (make-vector memory-size null))

(define car-memory-2 (make-vector memory-size null))
(define cdr-memory-2 (make-vector memory-size null))

(define (car-poke! address val)
  (vector-set! car-memory address val))
(define (cdr-poke! address val)
  (vector-set! cdr-memory address val))
(define (car-peek address)
  (vector-ref car-memory address))
(define (cdr-peek address)
  (vector-ref cdr-memory address))

(define root null)

(define (root! r)
  (set! root r))
```

In total, we have four memory banks. A bank for cars and a bank for cdrs is required for both semi-spaces. `car-memory` and `cdr-memory` indicates the working semi-space. `car-memory-2` and `cdr-memory-2` indicates the unused (and thus empty) semi-space that take over the role of the working semi-space as soon as the other semi-space is exhausted.

As indicated before, allocating a pair just means moving the `next-free` pointer one memory location to the right. We run out of memory as soon as that pointer reaches the upper limit of the supported address range.

```
(define next-free 0)

(define (out-of-memory?)
  (eq? next-free memory-size))

(define (allocate)
  (define addr next-free)
  (set! next-free (+ 1 next-free))
  addr)
```

At that point, the implementation of `cons` will try to reclaim storage by calling `gc`. The stop-and-copy version of this procedure is shown below:

```
(define forward 'forward)

(define (gc)
  (define old-car-memory car-memory)
  (define old-cdr-memory cdr-memory)

  (define (move old-pair)
    (if (scheme:pair? old-pair)
      (let*
          ((old-addr (untag old-pair))
           (old-car (vector-ref old-car-memory old-addr))
           (old-cdr (vector-ref old-cdr-memory old-addr)))
        (if (eq? old-car forward)
          old-cdr
          (let*
              ((new-addr next-free)
               (new-pair (tag new-addr)))
            (set! next-free (+ next-free 1))
            (vector-set! old-car-memory old-addr forward)
            (vector-set! old-cdr-memory old-addr new-pair)
            (vector-set! car-memory new-addr old-car)
            (vector-set! cdr-memory new-addr old-cdr)
            new-pair)))
      old-pair))
  (define (scan addr)
    (if (< addr next-free)
      (let
          ((old-car (vector-ref car-memory addr))
           (old-cdr (vector-ref cdr-memory addr)))
        (vector-set! car-memory addr (move old-car))
        (vector-set! cdr-memory addr (move old-cdr))
        (scan (+ addr 1)))))
  (set! car-memory car-memory-2)
  (set! cdr-memory cdr-memory-2)
```

```
(set! next-free 0)
(set! root (move root))
(scan 0)
(set! car-memory-2 old-car-memory)
(set! cdr-memory-2 old-cdr-memory))
```

Whenever we detect that the working memory has become exhausted, we traverse all reachable memory starting at the root. While doing the traverse, the traversed pairs are copied to the other semi-space, in a leftmost fashion. After the copy, all reachable pairs have been copied. The unreachable ones have been omitted and this should give rise to less memory used in the second semi-space than the memory that was used in the original semi-space.

The algorithm is initialised by copying the root pair to the new semi-space and calling `scan` from the root pair (i.e. scan with index 0). The heart of the copying process is the `scan` procedure shown. It moves from left to right *in the new semi-space* and investigates every pair it encounters. Since this pair resides in the new semi-space, it is a pair that was previously moved to that semi-space by the same `scan`. However, copying that pair did *not* copy its components. Hence, whenever `scan` encounters a pair in the new semi-space, its car and cdr still refer to the old semi-space. Therefore, `scan` simply calls `move` for both the car and the cdr of the pair. Now there are two possibilities:

- Either the content encountered is not a pair (this is checked by (`pair? old-pair`)). This means that the pair in the new semi-space contains an atomic.

- If the content *is* a pair, then it has to be copied from the old semi-space to the new one. This is the main purpose of `move`. However, caution is required when copying pairs. Whenever two distinct pairs refer to a (third) pair in the original semi-space, then the copied versions of those two pairs should also refer to the *same* copied version of the third pair. In order to ensure this, a pair should *not* be copied when it is encountered for the second time during the scanning process. To ensure this, after copying the third pair as a result of following the reference from the first pair we have to install a marker in the third pair. When accessing the third pair from within the second pair, we will find the marker which will prevent us from copying the pair again. The marker is a combination of the `forward` symbol stored in the car of the third pair and a reference in the cdr to the location of its copy in the new semi-space. When accessing the third pair from within the second pair, we will find the marker telling us the pair has already been copied. All we need to do then is to consult the cdr in the old semi-space to obtain the location of the third pair in the new bank. The `forward` symbol is sometimes called a *broken heart* in garbage collection literature.

This concludes our presentation of the stop-and-copy algorithm for pair memories.

18.3.2 Implementation for vectors

The stop-and-copy algorithm is easily generalised towards memories of variable-length chunks, i.e. Scheme vector memories.

We start by defining the memory model which consists of two semi-spaces called `memory` and `old-memory`. At any point in time, `memory` is the working semi-space. Vectors contain only 1 hidden slot. Hence `overhead` is 1.

```
(define null ())
(define overhead 1)
(define memory-size 25)

(define memory (scheme:make-vector memory-size null))

(define (poke! addr value)
  (scheme:vector-set! memory addr value))

(define (peek addr)
  (scheme:vector-ref memory addr))

(define old-memory (scheme:make-vector memory-size null))

(define (poke-old! addr exp)
  (scheme:vector-set! old-memory addr exp))

(define (peek-old addr)
  (scheme:vector-ref old-memory addr))
```

Again, the root is the memory chunk that is considered to be the start of the graph of vectors that are still in use. The root needs to be registered by calling `root!` with a given vector. The second argument to `root!` is a callback procedure that will be called every time the root is moved. It allows the application to take the appropriate action.

```
(define root null)
(define new-root! ())
(define (root! r notify)
  (set! root r)
  (set! new-root! notify))
```

The implementation of `allocate` is fairly simple. At any point in time, `next-free` refers to the leftmost memory address such that all memory that is located to the right of that address is free. First, the procedure checks whether a chunk big enough is still available and calls the garbage collector whenever this is not the case. In case enough memory is available, the head of free memory zone `next-free` is shifted to the right and the original value of that head is returned after having stored the size of the allocated chunk in the first memory cell beloning to the chunk.

```
(define next-free 0)

(define (allocate size)
  (define boundary (- memory-size size))
  (if (> next-free boundary)
    (gc))
  (if (> next-free boundary)
    (error "storage overflow" size))
  (let ((addr next-free))
    (set! next-free (+ next-free size))
    (poke! addr size)
    addr))
```

The garbage collection procedure `gc` is a straightforward generalisation of the version for pairs we saw in section 18.3.1. The semi-spaces are swapped and the vector that corresponds to the root is copied to the new semi-space. The heart of the algorithm is the `scan` function which scans vectors that have already been copied into the *new* semi-space from left to right. Every time a pointer refers to a chunk of memory in the old semi-space that is not a forward reference, it means that the vector corresponding to that pointer was still not copied so far. In that case we copy that vector to the new semi-space as well. In the location of the original vector (in the old semi-space) we put a forward reference to the location of the vector in the new semi-space. This way, vectors are copied only once. Whenever we encounter a second reference to the same position in the old semi-space, we are bound to find a forward reference. Instead of copying, the address of the vector in the new semi-space is simple returned. By using this technique, consistency of pointers pointing to the same vector is guaranteed.

```
(define (gc)
  (define hold-memory memory)

  (define (move old-vector)
    (if (vector? old-vector)
      (let*
          ((old-addr (untag old-vector))
           (old-size (peek-old old-addr)))
        (if (forward? old-size)
          (tag (forward-address old-size))
          (let*
              ((addr next-free)
               (newv (tag addr)))
            (set! next-free (+ next-free old-size))
            (poke-old! old-addr (make-forward addr))
            (poke! addr old-size)
            (do ((index 1 (+ index 1)))
              ((>= index old-size) newv)
```

```
                (poke! (+ addr index) (peek-old (+ old-addr index)))))))
      old-vector))

  (define (scan addr)
    (if (< addr next-free)
      (let
          ((size (peek addr)))
        (do ((index 1 (+ index 1)))
          ((>= index size))
          (poke! (+ addr index) (move (peek (+ addr index)))))
        (scan (+ addr size)))))
  (set! memory old-memory)
  (set! old-memory hold-memory)
  (set! next-free 0)
  (set! root (move root))
  (new-root! root)
  (scan 0))
```

The implementation of the broken heart is slightly more complicated than the version or pairs. It is a tagged address that refers to an address in the new semi-space. Here is the machinery.

```
(define forward-tag 'forward)

(define (forward? exp)
  (and (scheme:pair? exp)
       (eq? (car exp) forward-tag)))
(define (make-forward addr)
  (cons forward-tag addr))
(define (forward-address fwd)
  (scheme:cdr fwd))
```

18.4 Mark-and-Sweep Collection for Pairs

The previous sections presented a two simple garbage collection schemes. Both techniques are simple to program but have a substantial drawback. The reference counting algorithms cannot handle cyclic data structures and the stop-and-copy algorithms waste half of the available memory. Notice that cyclic data structures are obtained more easily than one might expect. E.g. in a double linked list implementation as presented in section 3.2.7, every node has a reference to its successor and every such successor has a back pointer to the node. Hence, having cyclic data structures is quite common.

In this section we present a family of algorithms that avoids both problems. We present a so-called *mark and sweep* algorithm. The idea of this approach is to traverse all the memory still in use by transitively following all the pointers accessible from the root. During this process, we "mark" the fact that we have

seen a chunk. Afterwards, the entire memory is traversed again. Those chunks that are still left unmarked can be safely reclaimed since they are not reachable from the root. This is the "sweep" phase of the algorithm.

In this section, we present the version for pairs. We split our presentation in two. We first explain a "naive" recursive version in order to explain the basic idea. In our second version, we remove the recursion from the algorithm.

18.4.1 Recursive Version

This time, the memory model is much simpler as we do not need the two semi-spaces anymore. Our memory just consists of two banks, one containing the car of each pair and the other containing the corresponding cdr. The initialisation phase simply builds up a freelist in the car parts. Notice that we did not need a free list in the stop-and-copy collector: since we copy the pairs from the old semi-space to the new semi-space, we always copy the pairs in a leftmost fashion. Hence free memory always sits "to the right" of the `next-free` pointer. In the mark-and-sweep approach, pairs are never copied or moved. We just need to visit those pairs that are still relevant. The others need are collected in a freelist. Here is the code for this memory model:

```
(define memory-size 20)

(define car-memory
  (do ((memory (make-vector memory-size null))
       (addr 1 (+ addr 1)))
    ((= addr memory-size ) memory)
    (vector-set! memory (- addr 1) addr)))

(define cdr-memory
  (make-vector memory-size null))

(define (car-poke! address val)
  (vector-set! car-memory address val))
(define (cdr-poke! address val)
  (vector-set! cdr-memory address val))
(define (car-peek address)
  (vector-ref car-memory address))
(define (cdr-peek address)
  (vector-ref cdr-memory address))
```

Managing the cells is extremely simple. Allocation just boils down to popping a pair from the freelist. We run out of cells as soon as the freelist reaches `null`.

```
(define next-free 0)

(define (allocate)
  (define addr next-free)
```

```
  (set! next-free (vector-ref car-memory addr))
  addr)

(define (out-of-memory?)
  (eq? next-free null))
```

Let us now look at the collector. The recursive version for pairs is extremely simple. The `gc` procedure contains two local procedures `mark` and `sweep` that are called one after the other. `mark` foresees one bit per pair and uses that bit in order not to go into an infinite recursion: whenever the argument pair is a previously unvisited pair, we mark it as visited and we subsequently visit its car and its cdr. The `sweep` phase scans the entire memory in a brute-force manner and pushes every unvisited pair in the freelist. Notice that the `mark` process actually corresponds to the depth-first traversal algorithm of graphs explained in section 10.2.1.

```
(define (gc)
  (define free-bits (make-vector memory-size #t))
  (define (mark pair)
    (if (scheme:pair? pair)
        (let ((addr (untag pair)))
          (when (vector-ref free-bits addr)
            (vector-set! free-bits addr #f)
            (mark (vector-ref car-memory addr))
            (mark (vector-ref cdr-memory addr))))))
  (define (sweep)
    (do ((addr 0 (+ addr 1)))
      ((= addr memory-size ))
      (when (vector-ref free-bits addr)
        (vector-set! car-memory addr next-free)
        (set! next-free addr))))
  (mark root)
  (sweep))
```

Unfortunately the above collector is extremely naive. Indeed, the recursion is a tree-recursion and this inevitably requires memory in order to build up the runtime stack. In a real world implementation, we cannot afford this because we just ran out of memory. That was the reason for calling `gc` in the first place! Hence, we have to convert the algorithm into an iterative version. In Scheme this means turning the recursion into tail recursion. The resulting algorithm is presented in the following section.

18.4.2 Iterative Version: The Deutsch-Shorr-Waite algorithm

In the original algorithm, we rely on the tree recursion to steer the traversal. It is the history of the tree recursive process that "knows" whether we have never

seen a pair, whether we have just returned to a pair after visiting its car or whether we have just returned to a pair after visiting its cdr. In the iterative version we will have to encode this information manually for every pair. In our version of the algorithm, we foresee a counter *per* cell. That counter will be 0 in the beginning and will become 1 and 2 after visiting a pair once (descending into its car) and twice (descending into its cdr) respectively. In a real world implementation, these three states can be encoded in only two bits.

```
(define count-mem (make-vector memory-size 0))
(define (count! pair cnt)
  (vector-set! count-mem (untag pair) cnt))
(define (count pair)
  (vector-ref count-mem (untag pair)))
```

Whenever we read the car or the cdr of a pair, the resulting item is to be visited if (a) it is a pointer to a pair that (b) was previously unvisited. Hence, if it is a pair whose counter is still equal to 0:

```
(define (descend? item)
  (and (scheme:pair? item)
       (zero? (count item))))
```

Next we discuss the `gc` procedure. Again it consists of two local procedures called `mark` and `sweep`. The `sweep` procedure is nearly identical to the previous version. It scans the entire memory and collects the unvisited pairs (i.e. the pairs whose counter is still 0) into the freelist.

The big difference lies in the implementation of `mark`. This time, the algorithm is iterative (i.e. all recursive calls of `mark` are in tail position!) and it only relies on two explicit parameters called `prev` and `cure` which have a reference to the current pair and the previous pair in the graph. This means that `prev` is the pair who's car *or* who's cdr is `curr`. If the algorithm descended into the car of `prev`, then `curr` will be that car. If the algorithm descended into the cdr of `prev`, then `curr` will be that cdr.

```
(define (gc)
  (define (mark prev curr)
    (case (count curr)
      ((0)
       (count! curr 1)
       (let ((hold (car-peek (untag curr))))
         (cond
           ((descend? hold)
            (car-poke! (untag curr) prev)
            (mark curr hold))
           (else
            (mark prev curr)))))
      ((1)
       (count! curr 2)
```

```
      (let ((hold (cdr-peek (untag curr))))
        (cond
          ((descend? hold)
           (cdr-poke! (untag curr) prev)
           (mark curr hold))
          (else
           (mark prev curr)))))
     ((2)
      (if (not (null? prev))
        (case (count prev)
          ((1)
           (let ((hold (car-peek (untag prev))))
             (car-poke! (untag prev) curr)
             (mark hold prev)))
          ((2)
           (let ((hold (cdr-peek (untag prev))))
             (cdr-poke! (untag prev) curr)
             (mark hold prev)))))))))
(define (sweep)
  (do ((addr 0 (+ addr 1)))
    ((= addr memory-size))
    (when (zero? (vector-ref count-mem addr))
      (vector-set! car-memory addr next-free)
      (set! next-free addr))
    (vector-set! count-mem addr 0)))
(unless (null? root)
  (mark () root))
(sweep))
```

Upon visiting `curr` the algorithm tests whether this is the first time, the second time or the third time that this pair is visited. This is done by inspecting the counter of the pair. Notice that this counter is updated *immediately* (i.e. before anything else happens) in the case 0 and 1. This precludes the algorithm from walking in cycles. As soon as the counter reaches 2, we backtrack from this pair and we will never visit it again.

The central idea is to encode the runtime stack (needed to properly backtrack from the recursion) in the pairs themselves! Whenever we reach a pair who's count is 0, we decide to descend into the car of that pair. We first change its count to 1 an then we descend into the car if needed. This means that in the next iteration, the current pair will become the `prev` and that the current pair's car will become the `curr`. But since we have a proper reference to the car (sitting in the variable `curr`), we can safely encode the old `prev` (i.e. the previous of the previous) into the car of the pair. Likewise, if we reach a pair for the third time, we decide to descend into its cdr (if it is necessary). But before doing so, we turns its count into 2 and we register the previous of `curr` into the cdr slot of the pair. Notice that in both cases, we stay in the same pair by

calling `mark` with the exact same arguments. However, since we have updated the counter this will not result in an infinite recession.

The final insight for understanding the algorithm is to understand what happens during the backtracking phase. As soon as we reach a pair whose count is 2, we have visited its car as well as its cdr. In that case, we "feel" the `prev` by inspecting its count. If this pair's count is 1 we have previously descended from that `prev` following its car pointer. Hence, we know that we can backtrack to that `prev` by reading *its* previous form its car slot. Likewise, if we find the count to be 2, we know that the `prev` encoded its previous in its cdr slot before descending. Hence, in both cases, we "climb out of the recursion" by making the `prev` the new `curr` and by reading the new `prev` from the car or the cdr of the current `prev`.

18.5 Mark-and-sweep Collection for Vectors

The mark-and-seep collectors for vectors are non-trivial generalisations of the ones for pairs. Again, we proceed in two steps. In the first version, we present the recursive implementation. This will illustrate an important principle that is also relevant in the iterative version. However, the code of the iterative version is further complicated by the fact that we have to manually encode the backtracking process in the same vein as what we did in the case for pairs.

18.5.1 Crunching

The mark-and-sweep collectors for pairs simply link up the collected pairs in a freelist. This makes sense because all pairs are identical anyhow. Hence there is no need to move pairs. However, when collecting vectors in a free list we might end up with the undesirable situation where there is still enough free memory (in terms of the number of free locations) to host a vector of some particular size but in such a way that the components of that vector cannot be stored in adjacent memory locations. This is problematic because part of the definition of vectors is that `vector-ref` and `vector-set!` are in $O(1)$. Obviously, this is impossible to guarantee if we do not store vectors in adjacent memory locations. If this happens, we say that the memory is *fragmented*: the available memory locations are scattered all over the memory, in between vectors that are in use.

The solution to this problem consists of combining the garbage collection phase with a *crunching* algorithm. The idea is to identify those vectors that are still in use and to move them to the leftmost addresses in memory. As a result, after the collect, all available memory will consist of one big chunk that is situated to the right of the vectors that are still in use.

However we cannot just bluntly move vectors "to the left. If vectors refer to one another (or to themselves), we have to make sure that the references are modified accordingly during the crunch. Just moving vectors would leave many *dangling references*.

18.5.2 Recursive Version

Let us begin with the recursive implementation of the garbage collection technique. The memory model is extremely simple. In the meta-circular style, the memory to be managed is represented as a Scheme vector again. Notice that every chunk (both allocated vectors as well as free chunks of memory) will store its size in its very first component. Hence, every vector generates an overhead of 1 that is needed by the memory management technique. In the following section, we will see that the iterative version requires an overhead of 2.

```
(define null ())
(define overhead 1)
(define memory-size 250)

(define memory
  (let
      ((m (scheme:make-vector memory-size null)))
    (scheme:vector-set! m 0 memory-size)
    m))

(define (poke! addr value)
  (scheme:vector-set! memory addr value))

(define (peek addr)
  (scheme:vector-ref memory addr))
```

In contrast to the garbage collector for pairs, the one for vectors will perform a crunch and thus relocate vectors. As long as these vectors are referred to by other vectors residing in that same memory, the references are taken care of by the algorithm. However, if we have "external" Scheme variables referring to those vectors, then we need to update these variables every time data is moved by the garbage collector. This is especially true for the root. Therefore, registering a root consist of designating a root vector *and* registering a callback procedure (called `new-root!`) that will be called by the garbage collector every time the root is moved about. In this callback, the application programmer can update his variables such that they correctly refer to the vectors they were originally referring to.

```
(define root null)

(define new-root! ())
(define (root! r notify)
  (set! root r)
  (set! new-root! notify))
```

Allocation is really simple. `next-free` refers to the address in memory that separates useful vectors and garbage (left of `next-free`) from free memory (right of `next-free`). The allocator first checks whether there is still enough space

sitting on the right of `next-free`. If this is not the case, garbage is collected and the allocation is given a second try. In case it succeed, `next-free` corresponds to the home address of the newly allocated vector. (+ `next-free size`) becomes the new value for `next-free`. Remember that the very first slot of any chunk (be it a vector or the beginning of the free memory) always contains the length of the chunk.

```
(define next-free 0)

(define (allocate size)
  (define boundary (- memory-size size))
  (if (> next-free boundary)
    (gc))
  (if (> next-free boundary)
    (error "storage overflow" size))
  (let ((addr next-free))
    (set! next-free (+ next-free size))
    (poke! addr size)
    (if (< next-free memory-size)
      (poke! next-free (- memory-size next-free)))
    addr))
```

Let us now study the recursive version of the collector. This time, the algorithm consists of three consecutive phases, the latter two of which correspond to the sweep phase shown in the collector for pairs.

- The first phase still corresponds to marking all memory that is transitively accessible from the root. This is the role of the procedure `mark`
- The second phase consist of traversing all vectors that are still in use hereby updating all pointers such that they refer to the *new* location of the vectors they originally point to. This is the role of the procedure `sweep-prepare`.
- The third phase actually executes the crunch by moving vectors to the left. This is the role of the procedure `sweep-crunch`.

Here is the structure of the code:

```
(define (gc)
  (define (mark curr)
    ...)
  (define (sweep-prepare)
    ...)
  (define (sweep-crunch)
    ...)
  (if (vector? root)
    (mark root))
```

```
(sweep-prepare)
(sweep-crunch)
(new-root! root))
```

A central problem to be solved when moving vectors is that *all* the pointers that refer to some vector before the move should be updated in order for them to move to the same moved vector after the move. In order to make this possible, the `mark` phase will build up a linked list of vectors that refer to a vector. First, `mark` uses the size slot of a vector to indicate whether or not the vector was previously visited. We only visit those vectors that have a size slot that (a) is a number which is (b) positive. In this is the case, we immediately turn the number negative in order not to walk in cycles. Next, we start a loop in order to consider all components `comp` of the vector from left to right. Those components that are vectors (i.e. not atomic data) are recursively marked. So far, the algorithm is a straightforward extension of the recursive algorithm for marking pairs.

```
(define (mark curr)
  (define addr (untag curr))
  (define size (peek addr))
  (when (and (number? size) (> size 0))
    (poke! addr (- size))
    (do ((indx 1 (+ indx 1)))
      ((= indx size))
      (let*
          ((comp-addr (+ addr indx))
           (comp (peek comp-addr)))
        (when (vector? comp)
          (mark comp)
          (let*
              ((head-addr (untag comp))
               (head (peek head-addr)))
            (poke! comp-addr head)
            (poke! head-addr (tag comp-addr)))))))))
```

But notice however that some interesting code is still executed after each (`mark comp`) call. Since we have just returned from recursively visiting `comp`, we now that `curr` is one of the vectors that refers to `comp`. We therefore insert `curr` in the linked list of vectors that refer to comp. We do this by making the size slot of `comp` refer to the current slot of `curr` and by making that slot refer to the old contents of the size slot of `comp`. After executing this code, `curr` has "added itself" to the linked list of vectors referring to `comp`. By doing this for every reference to `comp`, after the mark phase, we can safely move `comp` while still being able to find all vectors that need to be notified about this move.

The next phase is `sweep-prepare`. This is a so-called "2 finger algorithm". The two fingers `addr` and `new-addr` hop from vector to vector. `addr` is always slightly ahead and "feels" which vectors can be skipped (because they were never

marked) and which vectors need to be copied "to the left". The target location for that copy is `new-addr`. The loop updates *both* fingers if we encounter a vector that needs to be copied whereas it only updates `addr` if we encounter a vector that turns out to be garbage. In every vector encountered, we traverse the list whose head begins in the size slot. This may be a "trivial list" that just consists of a positive number or a real list that ends in a negative number. In the latter case, the body of the named let `traverse` will encounter all the vectors that used to refer to the vector encountered. All we need to do is traverse that list (which ends in a negative number indicating the size of the vector) and make all the encountered locations refer to the new location of that vector.

```
(define (sweep-prepare)
  (let loop
    ((addr 0)
     (new-addr 0))
    (if (< addr memory-size)
      (let*
          ((size (let traverse
                   ((curr (peek addr)))
                   (if (number? curr)
                     curr
                     (let*
                         ((curr-addr (untag curr))
                          (next (peek curr-addr)))
                       (poke! curr-addr (tag new-addr))
                       (traverse next))))))
        (cond
          ((negative? size)
           (poke! addr size)
           (loop (- addr size) (- new-addr size)))
          (else
           (loop (+ addr size) new-addr)))))))
```

Notice that `sweep-prepare` cannot copy the vectors immediately. If it would do so, it might erroneously copy a vector that will still be encountered in one of the other linked lists later on. In brief, it cannot move any vector as long as the old version of the vector may still be needed. Since `sweep-prepare` is scanning all memory from left to right, any vector can still be a part of any of the lists that we still need to traverse. It would be an error to move that vector since this would result in a corrupted memory. This is why `sweep-crunch` traverses the memory one more time *after* all vector pointers have been updated to point to the new location of the corresponding vector. Once again, `sweep-crunch` hops from vector to vector with a 2-finger algorithm. Vectors with a positive size slot are skipped since they where never reached by the `mark` phase. Vectors with a negative size slot are copied to the left. The algorithm terminates when the entire memory is scanned. All that remains to be done is poke the amount of available memory in the size slot of the rightmost memory chunk and make

`next-free` refer to the corresponding address.

```
(define (sweep-crunch)
  (let loop
    ((addr 0)
     (new-addr 0))
    (cond
      ((< addr memory-size)
       (let ((size (- (peek addr))))
         (cond
           ((> size 0)
            (poke! new-addr size)
            (do ((index 1 (+ index 1)))
              ((= index size))
              (poke! (+ new-addr index)
                     (peek (+ addr index))))
            (if (= addr (untag root))
              (set! root (tag new-addr)))
            (loop (+ addr size) (+ new-addr size)))
           (else
            (loop (- addr size) new-addr)))))
      (else
       (if (< new-addr memory-size)
         (poke! new-addr (- memory-size new-addr)))
       (set! next-free new-addr)))))
```

Notice that `sweep-crunch` registers the new location of the root as soon as that location is known. The garbage collector `gc` finishes by notifying the application about the new location of the root.

18.5.3 Iterative Version: The Deutsch-Shorr-Waite algorithm

Let us now combine the two most intriguing ideas of the mark-and-sweep collectors seen so far. First, we will remove the recursion from the algorithm by encoding the backtracking path in the data structure we are traversing. Second we construct a list of vectors that refer to a vector in order to prepare the crunch phase. We focus on the `mark` phase as the code for `sweep-prepare` and `sweep-crunch` is virtually left untouched.

Remember from section 18.4.2 that every pair uses two bits in order to encode the three possible states of the pair during the `mark` phase: "unvisited", "now visiting the car" and "now visiting the cdr". If we want to generalise this towards vectors that can consist of N components, every vector will need to remember how many of these components have already been visited such that the next component can be chosen as soon as the iterative algorithm arrives at the vector again. Since this number needs to be remembered on a per-vector basis, the algorithm has an overhead of two slots per vector: one to store the

size of the vector and the other one to store the number of the component that has to be visited next time the marking algorithm "passes by".

```
(define overhead 2)
```

Let us discuss the `mark` procedure shown below. We begin with the helper procedure `descend?` item which only returns #t provided that the argument is a pointer to a vector that was never visited before. If this is the case, the size slot of the vector is made negative in order to prevent cyclic behaviour.

```
(define (descend? item)
  (if (vector? item)
    (let*
        ((addr (untag item))
         (size (peek addr)))
      (if (number? size)
        (cond
          ((> size 0)
           (poke! addr (- size))
           #t)
          (else #f))
        #f))
    #f))
```

`mark` itself takes two parameters, namely a reference `curr` to the currently visited vector and a reference `prev` to the "parent" of that vector in the vector graph. Notice that `mark` is an iterative algorithm since all the recursive calls are in tail position.

```
(define (mark prev curr)
  (define addr (untag curr))
  (define size (peek addr))
  (define indx (peek (+ addr 1)))
  (cond
    ((> indx 2)
     (let*
         ((comp-addr (+ addr indx -1))
          (comp (peek comp-addr)))
       (poke! (+ addr 1) (- indx 1))
       (cond
         ((descend? comp)
          (poke! comp-addr prev)
          (mark curr comp))
         ((vector? comp)
          (let*
              ((head-addr (untag comp))
               (head (peek head-addr)))
            (poke! comp-addr head)
```

```
            (poke! head-addr (tag comp-addr))
            (mark prev curr)))
         (else
          (mark prev curr)))))
   ((vector? prev)
    (let*
        ((prev-addr (untag prev))
         (prev-size (peek prev-addr))
         (prev-indx (peek (+ prev-addr 1)))
         (comp-addr (+ prev-addr prev-indx)) ; prev-indx already -1
         (prev-prev (peek comp-addr)))
      (poke! comp-addr size)
      (poke! addr (tag comp-addr))
      (mark prev-prev prev)))))
```

Upon entering `mark`, we read the `indx` from the second overhead slot of the current vector. This number indicates the number of the vector component that is to be visited this time. Naively spoken, this number should vary from 2 (the memory offset that corresponds to the 0th component of the vector) to `size`−2 (the memory offset that corresponds to the last component of the vector). However, in order to test this in the code, we would need the numeric value of `size`. This is not easy to determine since the size slot may contain the head of the list of vectors that refer to this vector (as in the recursive version of the algorithm). Surely we could decide to traversing that list in order to find (the negation of) the size at the end of the list, but this would not be very efficient. That is why we chose to let `indx` vary from the `size` of the vector *down to* 2. The latter end condition *can* be tested in code without needing the size of the current vector. The last time we visit the vector in `curr`, its `indx` will hold the value 2. At that moment we decide to backtrack to the `prev` vector.

As long as `indx` has not reached 2, we update its value (for the next visit) to (− `indx` 1). The next (i.e. previous in terms of vector entries) component `comp` is considered by calling (`mark curr comp`) in case `comp` is a component that requires descending. However, before doing so, we encode the previous `prev` in the slot of the `curr` that referred to that component. This is the direct generalisation of remembering the previous of a pair in the car or cdr or that pair. Only this time there are "many cars" since we are dealing with vectors. If the component `comp` is a vector that does not require descending (i.e. we have already been there), all we do is registering the current vector in the linked list of the vectors that refer to `comp` and we enter (`make prev curr`) again in order to consider the next vector component (remember that the `indx` has already been decremented by 1). Reentering the (`mark prev curr`) is also what needs to be done if the component is not a vector at all (i.e. an atom).

Finally, let us discuss the backtracking machinery of this algorithm. This is the second part of the `cond` which is triggered when `indx` is equal to 2 (i.e. all components from right to left have been considered). In that case, we backtrack (provided `prev` is still a vector — otherwise we have reached the root). All

we need to do is read the previous of `prev` from the corresponding slot in (+ `prev-addr prev-indx`). However, instead of making that slot refer to the address of the current again (which is what we did in the pair version), we make that slot refer to the head of the linked list of vectors that refer to the current (which resides in the `size` variable) and we make the size slot of the current refer to slot of the previous. In brief, we insert the previous in the linked list of blocks that refer to the current.

18.6 Summary

In this chapter we have briefly studied automatic storage reclamation systems. We have studied three algorithms in detail, both for memory chunks of fixed size (epitomised by Scheme pairs) and for chunks of variable size (epitomised by Scheme vectors). Remember however that the memory structure of a web of vectors referring to one another is exactly the same as the memory structure of a web of (Java) objects referring to each other. Hence, the techniques studied here are much more widely applicable than just Scheme.

Notice that we have only touched the tip of the iceberg. Memory management really has become an entire domain of computer science. We refer the interested student to [JHM11] for an excellent introduction to the field.

Bibliography

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2 edition, May 1998.

[CL04] Christian Charras and Thierry Lecroq. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, Cambridge, MA, USA, 2001.

[GMUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[JP04] Neil C. Jones and Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms*. The MIT Press, 2004.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Professional, second edition, May 1998.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[Lev02] Anany V. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[MT07] Hayden Melton and Ewan D. Tempero. Jooj: Real-time support for avoiding cyclic dependencies. In Gillian Dobbie, editor, *ACSC*, volume 62 of *CRPIT*, pages 87–95. Australian Computer Society, 2007.

[SDF+09] Michael Sperber, R. kent Dybvig, Matthew Flatt, Anton Van straaten, Robby Findler, and Jacob Matthews. Revised6 report on the algorithmic language scheme. *J. Funct. Program.*, 19:1–301, August 2009.