Chapter 3

Linear Data Structures

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

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

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

We start out by defining exactly what we mean by linearity. Based hereupon, we present a number of ADTs that define linear data structures abstractly and we discuss a number of implementation strategies for these ADTs. We discuss the performance characteristics for all implemented operations and whenever different from $O(1)$, we seek for ways to speed them up. The most notorious operation is `find`. It is — as explained in section 1.2.5 — one of the central operations studied in this book. We will analyze its performance characteristic

for almost every data structure studied. Section 3.4 studies the different versions of `find` for linear data structures and reveals some clever tricks that can speed up `find` considerably.

3.1 Using Scheme's Linear Data Structures

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

3.1.1 "Naked" Vectors

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

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

- Another important problem arising from the fact that the length of a vector is always fixed, is that we have to decide what is to happen whenever

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

Although these considerations make us conclude that naked Scheme vectors are not very practicable, vectors remain an interesting data structure. As the rest of this chapter shows, vectors remain an extremely useful compound data type that has attractive performance characteristics provided that we build the right abstractions "on top of" vectors.

3.1.2 Scheme's Built-in Lists

In Scheme, lists (built using *pairs*) form an alternative that circumvent the main disadvantages of vectors: when building a list, it is not necessary to know its capacity upfront. Using `cons`, one can always create a new pair to store an element and link that pair to an existing list.

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

- Scheme's lists only allow one to "cons something upfront" since a Scheme list is actually nothing more than a reference to its first pair. There is no direct reference to the last pair in the list. This means that operations that need to add a data value to or change something about the list at any position which is not the first one, turn out to be slow. Examples are `add-to-end` that adds an element to the end of the list and `append` which concatenates two lists. Both operations are in $O(n)$ since they need to traverse the entire list before doing any interesting work: starting at the first pair, they need to recursively work their way to the end of the list (following "cdr pointers") in order to reach the final pair. This loss of efficiency is the price to pay for the flexibility offered by lists. However, as we will see, "dressing up" lists with additional information (e.g. maintaining a direct reference to the very last pair of the list) can speed up things considerably.

- Another drawback that comes with naked Scheme lists is a consequence of the way arguments are passed to procedures. Whenever a procedure is called with pairs, Scheme passes the pairs *themselves* to that procedure instead of any variables containing those pairs. Manipulating parameters of a procedure will thus not change any variables external to that procedure.

  This behavior of Scheme is particularly problematic when storing data in naked Scheme lists. Indeed, suppose we implement a procedure `add-to-first!` as follows:

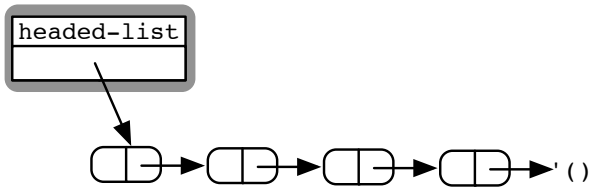

Figure 3.1: A typical headed list

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

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

- A final drawback of naked Scheme lists is that they are not generic in the way genericity was introduced in section 1.2.4. Scheme features several built-in operations (such as `member`, `memv` and `memq`) to find out whether or not a data element belongs to a list. However, these procedures explicitly rely on `equal?`, `eqv?` and `eq?`. It is impossible to alter these operations if one wishes to use a different equality. For instance, suppose one has a list containing persons (i.e. data values beloning to some ADT `person`, the details of which are not relevant here) and suppose one wishes to search the list for a person given his or her name. None of the aforementioned equality operators will do the job because we actually want the membership test to use our own (`person-equal? p1 p2`) procedure which is e.g. designed to return `#t` when `p1` and `p2` have an identical first name. In other words, we would actually want to call one of the built-in membership tests in such a way that it uses *our* equality procedure. Unfortunately, this kind of genericity is not built into the standard list processing procedures that come with Scheme. Scheme's lists are not generic.

3.1.3 Headed Lists and Headed Vectors

The standard technique that is used to solve the problems with naked Scheme lists is to use *headed Scheme lists*. Headed Scheme lists are lists that have an additional level of indirection that refers to the actual list. The additional level of indirection is called *the header* of the headed Scheme list. Figure 3.1 shows a headed list containing four elements. Its header consists of a record that is displayed on a grey background. Technically spoken, headed Scheme lists

consist of a data structure for the header (e.g. a list of pairs or a record) that holds a reference to the first pair of the actual Scheme list. The code below shows a headed list. Apart from storing a reference to a Scheme list, it stores an additional "info" field which is not further specified.

```
(define-record-type headed-list
  (make l i)
  headed-list?
  (l scheme-list scheme-list!)
  (i info info!)))
```

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

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

```
(define (add-to-front! hl e)
  (scheme-list! hl (cons e (scheme-list hl))))
```

The solution to the third problem of section 3.1.2 lies in using higher order procedures as the constructors for headed lists. Remember from section 1.2.4 that turning a constructor into a higher order procedure is the standard recipe to achieve genericity in Scheme. Every time we create a new list, we pass a comparator function to the constructor. The idea is to store that comparator function in the header. For example, if `new` is the name of the constructor, then we might use the call (`new eq?`) to create a headed list that uses Scheme's `eq?` whenever it needs to compare data values. For instance, the ADT's procedure `find` can use that equality operator by reading it from the list's header.

We finish this section by mentioning that it is also meaningful to talk about *headed vectors*. Similarly, these are data structures that consist of a header that maintains a reference to the actual vector in which the actual data elements reside. The contents of the header can be used to speed up some operations or to make the information in the vector more meaningful, e.g. by storing the counter that designates the last position in the vector that contains meaningful information.

In what follows, we present a number of useful linear ADTs and we describe how to implement them in Scheme using headed vectors and headed lists. As

we will see, depending on what is stored in the header, different performance characteristics are obtained for the operations of the ADTs

3.2 Positional Lists

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

3.2.1 Definitions

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

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

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

Actually, studying positional lists is not our real goal. Our actual point is to use the positional list ADT in order to study strategies for implementing linear data structures in general. These implementation techniques can be used to implement a wide variety of linear data structures. Positional lists are but one example.

3.2.2 The Positional List ADT

In the definition of the ADT shown below, we deliberately choose *not* to specify what we mean exactly by a position. As we will see, positions can be indices in a vector (i.e. numbers) as well as references to some Scheme pair or record value. The actual data types that are used as positions is not really important for users of the ADT. The only thing that matters is that every position in a

positional list (apart from the first position and the last position) has a next position and a previous position.

It is our goal to make storage data structures as generic as possible such that they can be used to store different types of data elements. That is why we have parameterized the positional list ADT with the data type `V` of the value elements stored. Moreover, since the actual type of the positions is not hardcoded in the definition of the ADT, the ADT `positional-list` is also parameterized by the position data type `P`. As we will see, different implementations of the ADT use different data types for `P`. E.g., in a vector implementation, positions correspond to vector indices which means that the role of `P` is concretized by the `number` data type. In other implementations, `P` is concretized by `pair` such that positions correspond to dotted pairs. Yet others will represent `P`'s values as references to records.

The generic ADT `positional-list< V P >` specification is shown below. For a particular implementation of this ADT and for a particular usage, we have to think of concrete data types for `V` and `P`. For example, suppose that we choose the vectorial implementation that represents positions as numbers. If we use this implementation to store strings, then the resulting positional lists are of type `positional-list<string number>`. Any concrete positional list that we use in our programs has a data type like this. Notice that `P` is chosen by the implementor whereas `V` is determined by the user of the ADT.

ADT positional−list < V P >

new
((V V → **boolean**) → **positional−list** < V P >)
from−scheme−list
(**pair** (V V → **boolean**) → **positional−list** < V P >))
positional−list?
(**any** → **boolean**)
length
(**positional−list** < V P > → **number**)
full?
(**positional−list** < V P > → **boolean**)
empty?
(**positional−list** < V P > → **boolean**)
map
(**positional−list** < V P >
(V → V')
(V' V' → **boolean**) → **positional−list** < V' P >)
for−each
(**positional−list** < V P > (V → **any**) → **positional−list** < V P >)
first
(**positional−list** < V P > → P)
last
(**positional−list** < V P > → P)

has–next?
(**positional–list** < V P > P → **boolean**)
has–previous?
(**positional–list** < V P > P → **boolean**)
next
(**positional–list** < V P > P → P)
previous
(**positional–list** < V P > P → P)
find
(**positional–list** < V P > V → P ∪ { #f })
update!
(**positional–list** < V P > P V → **positional–list** < V P >)
delete!
(**positional–list** < V P > P → **positional–list** < V P >)
peek
(**positional–list** < V P > P → V)
add–before!
(**positional–list** < V P > V . P → **positional–list** < V P >)
add–after!
(**positional–list** < V P > V . P → **positional–list** < V P >)

The constructor `new` takes a comparator function that will be used to compare any two values in the positional list. The procedural type of such comparators is (`V V` → `boolean`). As explained in section 1.2.4, turning the constructor into a higher order procedure is our technique to implement generic data structures. The comparator will be used by `find` during its search process. The comparator's job is to return `#t` whenever `find`'s argument matches the data values that are being investigated (one by one) during the search process. Apart form `new`, we have `from-scheme-list` which is an alternative constructor that returns a positional list given an ordinary Scheme list of data values (which is technically just a pair) and a comparator that works for those values. E.g., (`from-scheme-list` '(`1 2 3 4`) =) creates a positional list (containing the four numbers contained by the Scheme list) that uses Scheme's = operator for comparing elements.

Given any Scheme value[1], then `positional-list?` can be used to check whether or not that value is a positional list. Given a positional list, the operation `length` returns the number of data elements sitting in the list and `full?` (resp. `empty?`) can be used as a predicate to check whether or not the list is full (resp. empty).

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

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

the position that follows the position `p`. Similarly, (`previous l p`) returns the position that precedes `p`.

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

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

Figure 3.2 shows a screenshot of Apple's Numbers spreadsheet application (which is similar to Microsoft's Excell). One might implement the columns as a positional list in which the positions correspond to some column data type. In this application, the user can click a column after which a popup menu appears. Two of the menu options that appear have to do with adding columns. We can see from the figure how they correspond to the `add-before!` and the `add-after!` operations just described.

Caution is required with `add-before!` and `add-after!`. By default, both operations take a value and a position. The value is added right before or right after the given position. But what shall we do when there are no positions in the list? This occurs whenever we try to add elements to an empty positional list. That is why both `add-before!` and `add-after!` take the position as an *optional* argument. When omitting the argument in the case of an empty list, both `add-before!` and `add-after!` simply add the value as the first and only

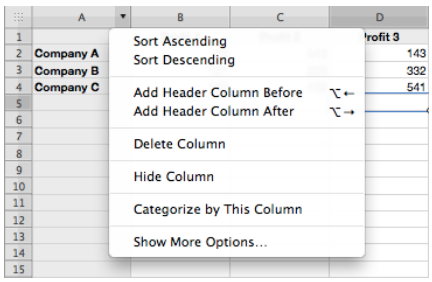

Figure 3.2: Possible use of a positional list

element of the list. But what is the meaning of `add-before!` and `add-after!` when the position argument is omitted for non-empty lists? In these cases, we use the convention that `add-before!` adds the element before *all* positions. In other words, we add the element to the front of the list. Similarly, calling `add-after!` on a non-empty list without providing a position adds the value after all positions, i.e., to the rear of the list.

3.2.3 An Example

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

```
(define-record-type event
  (make-event d m n)
  event?
  (d day)
  (m month)
  (n note))
```

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

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

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

This procedure is used to create a new positional list that can be used to store such events.

```
(define event-eq? (lambda (event1 event2)
                    (and (eq? (day event1) (day event2))
                         (eq? (month event1) (month event2)))))
(define todo-list (new event-eq?))
```

The following calls show us how to compose a todo-list by adding our three events to the end of the list, one after the other.

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

Now suppose that our professor decides that he has to prepare his class on linearity before teaching it. `lecture-2` is the position associated to the lecture of 12 October. A new event right before that lecture is added, namely the preparation of the lecture.

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

Now suppose our professor decides to have a resting period after preparing his lecture. A reference to the position of to the preparation is obtained and stored in the variable `prepare-lecture`. Subsequently, an event for the rest is scheduled after the preparation.

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

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

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

3.2.4 The ADT Implementation

In what follows, we discuss four implementations of the `positional-list` ADT. All four implementations use a different representation for the data structure. As a consequence, most of the ADT's operations are implemented differently in all four cases. However, this is not the case for all procedures. A number of procedures can be implemented on top of the other — lower level — procedures and therefore do not explicitly depend on the representation. We therefore group these procedures in a Scheme library that imports the lower level procedures which do rely on the representation. Those lower level procedures are put in four libraries; one per implementation strategy. By selecting a different library to import, a different implementation strategy is chosen. This is graphically

shown in figure 3.3. The library shown on the top is the one that users have to import when using positional lists. This library imports *one* of the four lower level implementation libraries in its turn. Before we move on to the first implementation of the ADT in section 3.2.5, we first discuss these procedures.

It may come as no surprise that `map` and `for-each` do not depend on any particular representation. The following code excerpt shows their implementations. It is very instructive to study these procedures in detail because they reveal how the abstractions provided by the `positional-list` ADT are used.

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

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

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

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

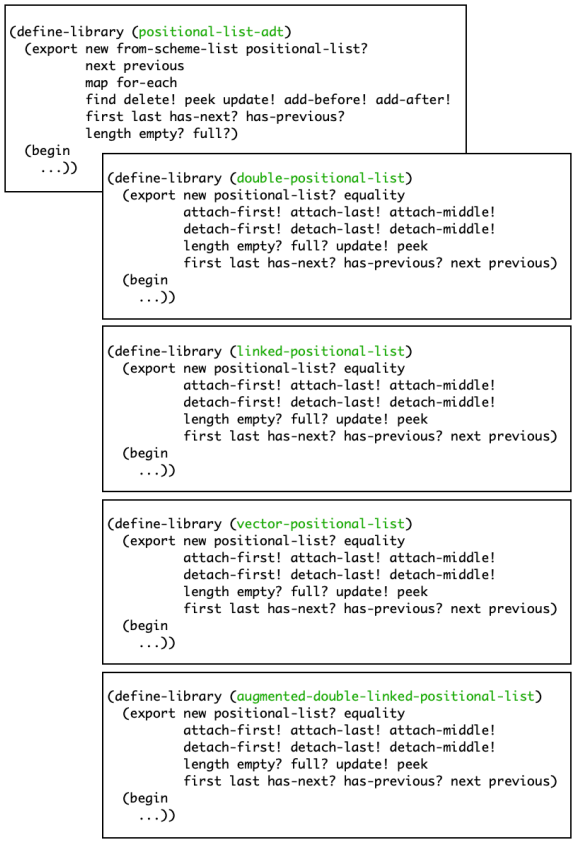

Figure 3.3: The structure of positional list implementations

`curr`. `next` is used to determine the next position in both the original list and in the result.

Another procedure that can be built on top of the more primitive procedures of the ADT, is the "secondary" constructor `from-scheme-list` which constructs a positional list based on a plain Scheme list. Its implementation is shown below. It simply creates a new positional list using `new` and then traverses the Scheme list. During that traversal, every element of the Scheme list is added to the positional list using `add-after!`. Notice that the performance characteristic of `from-scheme-list` heavily depends on the performance characteristic of the operations used. Especially the efficiency of `add-after!` will be crucial to keep the performance of `from-scheme-list` within reasonable bounds.

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

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

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

These procedures are not exported by the final implementation of the ADT! However, provided that all four concrete implementations implement (and export) these procedures, we can implement the ADT procedures `delete!`, `add-before!` and `add-after!`.

Deleting an element from a positional list depends on the position the element occupies. For the first position and for the last position (which is detected by the fact that calling `has-next?` yields #f) we use the dedicated procedures

`detach-first!` and `detach-last!`. In all other cases, we use `detach-middle!`. At this point it is impossible to establish a performance characteristic for `delete!` since it clearly depends on the three representation-dependent detachment procedures.

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

`add-before!` inserts a value before a given position. `add-after!` inserts a value right after the position provided. When `add-before!` does not receive an optional position argument, then the element is added to the front of the list. Similarly, omitting the optional position argument for `add-after` adds the element to the rear of the list. Their implementation looks as follows.

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

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

Both procedures use `attach-middle!` to add the new element to the list. `add-after!` uses its argument position to call `attach-middle!`. `add-before!` uses the previous position of the argument position. Notice that this position first has to be determined using `previous`. When the optional position is omitted, `add-after!` uses `attach-last!`. Similarly, `add-before!` uses

`attach-first!`. As for `delete!`, it is not possible to come with a performance characteristic without have a better look at the performance characteristic of the procedures `attach-first!`, `attach-middle!` and `attach-last!` provided by the concrete implementations.

Sequential Search

The final procedure that can be implemented without explicitly relying on a particular representation is `find`. The goal of `find` is to search the positional list for a given key. For comparing keys, `find` uses the positional list's own comparator that is accessed using the procedure `equality` that is exported by all four concrete implementations. `find` starts from the first position of the list and then considers every single element of the list by comparing it with the key. Successive applications of `has-next?` and `next` are used to traverse the positional list. In the body of the `sequential-search` loop, we observe two stop conditions to end the iteration: either the key is found, or the end of the list has been reached.

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

This algorithm is known as *the sequential searching algorithm* as is searches the positional list by considering the list's elements one after the other, in a sequence. Clearly the sequential searching algorithm exhibits an $O(n)$ behavior in the worst case. The average amount of work is in $O(\frac{n}{2})$ but that is $O(n)$ as well. In section 3.4 we study several techniques to speed up `find` for positional lists.

Some readers may find it strange that `find` does a lot of work to search for a value which we already have at our fingertips when calling `find`: isn't it a bit weird to execute the call (`find a-list a-key`) in order to ask `find` to search for something we already have (namely the object bound to the variable `a-key`)? In order to understand this, have another look at the example in section 3.2.3. In the construction of the positional list, a comparator `event-eq?` is passed. Upon closer inspection, this comparator does not compare *all* the fields of the `event`'s. Instead, two `event`'s are considered equal whenever their day and their month are equal. Remember from section 1.2.5 that we make a distinction

between key fields and satellite fields. In our example, the day and the month are considered to be the key fields and the note is considered to be a satellite field. The example clearly shows that the comparator used in the positional list's constructor only compares key fields and ignores satellite fields. When calling `find`, we use a key `event` that only bears meaningful values for the key fields. When the corresponding data element is found, we can use `peek` to read it from the positional list and subsequently access the satellite fields as well. In our example this is done using the accessors `day`, `month` and `note`.

Towards Performance Characteristics

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

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

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

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

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

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

Figure 3.4: Performance Characteristics for Shared Procedures

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

3.2.5 The Vectorial Implementation

In the vectorial implementation, the position abstraction `P` of the ADT is filled in by plain Scheme numbers. Hence `P` = `number`. In other words, a position is an index in the underlying vector. This is important to know when we find ourselves in the role of the *implementor* of the ADT. However, this should never be relied upon when *using* the ADT. It is not a part of the ADT specification. Positions may only be manipulated through the abstract operations `first`, `last`, `has-next?`, `has-previous?`, `next` and `previous`.

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

(`define` (`storage-move-right vector i j`)

```
  (do ((idx j (- idx 1)))
    ((< idx i))
    (vector-set! vector (+ idx 1) (vector-ref vector idx))))

(define (storage-move-left vector i j)
  (do ((idx i (+ idx 1)))
    ((> idx j))
    (vector-set! vector (- idx 1) (vector-ref vector idx))))
```

Representation

In the vector implementation of the `positional-list` ADT, we represent a positional list by an headed vector. Its header contains the vector `v` that stores the positional list's elements, a comparator procedure `e` and a variable `s` which is a counter indicating the next free position in the vector. It is initialized to 0. By convention we store the elements of the positional list "in the leftmost positions of the vector". Hence `s` is the first location in the vector (considered from left to right) that does not contain any meaningful data.

```
(define positional-list-size 10)

(define-record-type positional-list
  (make v s e)
  positional-list?
  (v storage storage!)
  (s size size!)
  (e equality))

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

Remember that the size of a vector needs to be known upon creation. In our implementation, the size is a predefined constant (namely 10). By adjusting the definition of `positional-list-size`, the implementation generates positional lists of different sizes. Notice that we make a distinction between the ADT's constructor `new` and the private constructor `make`. Remember from section 1.1.1 that the constructor's job is to reserve computer memory and to initialize the memory with meaningful values. These two roles are being taken care of separately by `new` and `make`: `make` creates the data structure in memory and `new` subsequently initializes it.

Verification

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

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

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

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

Navigation

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

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

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

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

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

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

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

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

Manipulation

Finally, we study the procedures to manipulate positions and the data elements associated with them. This includes the ADT operations `peek` and

`update!` as well as the 6 private procedures that are used to attach and detach positions in a positional list. Their implementation is shown below.

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

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

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

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

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

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

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

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

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

(define (detach-middle! plst pos)
```

```
(define vect (storage plst))
(define free (size plst))
(storage-move-left vect (+ pos 1) (- free 1))
(size! plst (- free 1)))
```

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

Performance

The first column of the table shown in figure 3.7 summarizes the performance characteristics for the vector implementation of positional lists. The table is a completion of the table in figure 3.4.

3.2.6 The Single Linked Implementation

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

Figure 3.5 shows a typical linked positional list. It consists of a header and the pairs that store data values. They are referred to as the *nodes* of the linked list. In the implementation presented here, these nodes are linked up in only one direction: every node stores a reference to its successor. We therefore refer to the implementation as a *single linked implementation*. In section 3.2.7 we present an alternative linked implementation in which every node stores a reference to its successor as well as a reference to its predecessor. This will be called a *double linked list*. The example shown in figure 3.5 is a single linked list with four nodes.

Representation

Let us start with the representational issues. Again, a distinction is made between the constructor `new` and the unexported private procedure `make`. The latter merely allocates memory to store the positional list. The role of the former is to call the latter and provide additional arguments to properly initialize the newly created positional list. The procedures `head`, `head!` and `equality` are used to read and write the elements of the header.

```
(define-record-type positional-list
  (make h e)
  positional-list?
  (h head head!)
```

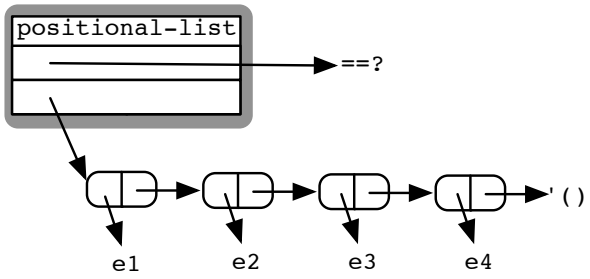

Figure 3.5: A typical linked positional list

```
  (e equality))

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

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

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

Verification

The implementations of `empty?` and `full?` are not that interesting. In the linked implementation, `full?` always returns `#f` since the implementation can always create a new node to add to the list[2]. This is in contrast with the vector implementation which requires the storage capacity of the list to be fixed when calling the constructor. The `length` operation is interesting though. In order to determine the length of the linked list, we need a loop `length-iter` that traverses all the nodes of the list in order to count them. In contrast to the vector implementation, this gives us an $O(n)$ implementation for `length`.

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

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

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

(define (full? plst)
  #f)

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

Navigation

The operations `first`, `last`, `has-next?`, `has-previous?`, `next` and `previous` are used to navigate through a positional list using the abstract notion of positions. In our pair representation of positions, the "cdr" stores a reference to the next position. This means that operations such as `next` and `has-next?` are readily implemented with an $O(1)$ performance characteristic. The same goes for `first` which merely accesses the first pair stored in the header. `has-previous?` is an $O(1)$ operation as well since it only has to verify whether or not the given position is the first position.

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

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

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

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

Unfortunately, the same cannot be said about the two remaining operations `previous` and `last`. Since the nodes of the list do not store a reference to their previous position, the only way to access the previous position is by starting at the first position and iterating through the list until the node is reached of which the previous position is required. The same goes for `last`: since the header only stores a reference to the very first node, the only way to reach the final node is to traverse the entire list starting from the header. The following higher-order procedure `iter-from-head-until` will be used in both procedures. It takes a positional list and starts iterating from the head of the list until the predicate

`stop?` returns #t for some node. As soon as this is the case, the preceding node is returned. Obviously, any operation that calls `iter-from-head-until` will exhibit a worst-case $O(n)$ performance characteristic.

```
(define (iter-from-head-until plst stop?)
  (define frst (head plst))
  (let chasing-pointers
    ((prev '())
     (next frst))
    (if (stop? next)
        prev
        (chasing-pointers
         next
         (list-node-next next)))))
```

`iter-from-head-until` uses an iteration technique that uses two variables `prev` and `next` representing two consecutive positions in the list. The `next` variable is the actual node we are inspecting in each step of the iteration and the action taken in the last step of the iteration is to return `prev`. `prev` and `next` systematically "follow each other". They are referred to as *chasing pointers*.

Based on `iter-from-head-until`, we can implement `previous` and `last` as follows. Both procedures call the higher-order procedure with a different `stop?` parameter. Hence, both `previous` and `last` are operations with performance characteristic in $O(n)$.

```
(define (previous plst pos)
  (if (not (has-previous? plst pos))
      (error "list has no previous (previous)" plst)
      (iter-from-head-until plst (lambda (node) (eq? pos node)))))

(define (last plst)
  (if (null? (head plst))
      (error "list empty (last)" plst)
      (iter-from-head-until plst null?)))
```

Manipulation

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

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

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

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

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

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

(define (attach-last! plst val)
  (define last (iter-from-head-until plst null?))
  (define node (make-list-node val '()))
  (define frst (head plst))
  (if (null? frst)
      (head! plst node) ; last is also first
      (list-node-next! last node)))
```

Detaching nodes is only simple for the very first node. All we have to do is make the header of the positional list refer to the second node. Hence, `detach-first!` is simple and in $O(1)$. `detach-middle!` and `detach-last!` are much more complicated. The reason is that the node to be removed stores a reference to its next node but not to its previous node. The previous node is needed since we have to make sure it refers to the next node of the node to be removed. Therefore, both `detach-middle!` and `detach-last!` will need to call `iter-from-head-until` to get a reference to the previous node of the node to be removed. Hence, both `detach-middle!` and `detach-last!` are in $O(n)$. Notice that `detach-last!` also has to cover the special case when the last node is the only node in the list. This means that we have to make sure that the header of the list no longer refers to that node as its first node. It is therefore set to '().

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

```
  (define frst (head plst))
  (define scnd (list-node-next frst))
  (head! plst scnd))

(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (iter-from-head-until
                plst
                (lambda (node) (eq? pos node))))
  (list-node-next! prev next))

(define (detach-last! plst pos)
  (define frst (head plst))
  (define scnd (list-node-next frst))
  (if (null? scnd) ; last is also first
      (head! plst '())
      (list-node-next! (iter-from-head-until
                        plst
                        (lambda (last) (not (has-next? plst last))))
                       '())))
```

Performance

The performance characteristics for the single linked implementation of the positional list ADT are summarized in the second column of figure 3.7. Remember from figure 3.4 that the $O(n)$ behavior of `attach-last!` only occurs if we call `add-after!` without the optional position parameter. Hence, `add-after!` is in $O(1)$ provided that we call it on a position.

3.2.7 A Double Linked Implementation

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

In this and the next section, we present two linked list implementations that perform significantly better than the vector implementation and the single linked list implementation. However, as is often the case in computer science, an improvement in speed is to be payed for with additional memory consumption. We distinguish two improvements:

- First, a number of operations can be sped up by allowing linked list nodes to store an additional reference to the node corresponding to its previous position. The resulting data structure is called a *double linked list* and the extra reference is known as a *back pointer*. A double linked version of the

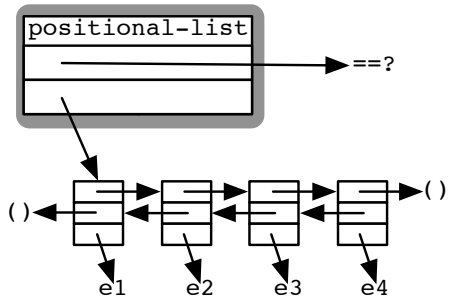

Figure 3.6: A double linked positional list

positional list depicted in figure 3.5 is depicted in figure 3.6. The code for this is explained in this section.

- Second, a number of operations can be sped up by storing additional information in the header of the positional list. E.g., instead of *computing* the length of the list in the implementation of the `length` operation, we can *store* the length of the list in the header of the list. This changes the $O(n)$ performance characteristic of `length` into $O(1)$. An implementation that applies this to a number of things is given in section 3.2.8.

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

Representation

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

```
(define-record-type list-node
  (make-list-node v p n)
  list-node?
  (v list-node-val list-node-val!)
  (p list-node-prev list-node-prev!)
  (n list-node-next list-node-next!))
```

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

```
(define-record-type positional-list
  (make h e)
```

```
  positional-list?
  (h head head!)
  (e equality))

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

Verification

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

Navigation

As can be expected, most of the navigation operations have an identical implementation as well. The only exception is the `previous` operation. In the single linked implementation, this operation was implemented by calling `iter-from-head-until` to iterate from the first node to the node of which the previous node is required. In the double linked implementation, all we have to do is follow the back pointer which brings `previous` into $O(1)$.

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

Manipulation

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

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

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

```
  (list-node-next! pos node)
  (if (not (null? next))
      (list-node-prev! next node)))

(define (attach-last! plst val)
  (define last (iter-from-head-until plst null?))
  (define node (make-list-node val last '()))
  (define frst (head plst))
  (if (null? frst)
      (head! plst node) ; last is also first
      (list-node-next! last node)))
```

Detaching nodes is very similar to the single linked implementation. The only notable difference is the implementation of `detach-middle!`. In the single linked implementation, this was an $O(n)$ operation since `iter-from-head-until` was needed in order to find the previous node of the detached node. In the double linked implementation, the previous node is found by simply following the back pointer. Hence, the operation is in $O(1)$ now.

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

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

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

Performance

The performance characteristics for the double linked implementation are summarized in the third column of figure 3.7. Here too, we note that `add-after!` only exhibits $O(n)$ behavior when omitting the position argument. When called on a concrete position, `add-after!` is in $O(1)$. The most important improvement of the double linked implementation w.r.t. the single linked implementation is that the implementation of the operations `previous`, `delete!` and

`add-before!` are in $O(1)$ whereas they used to be in $O(n)$ for the single linked implementation. This is because it is precisely these operations that require the previous position of a position. In the single linked implementation this required a call to `iter-from-head-until`. In the double linked implementation, we just have to follow the back pointer of a node.

3.2.8 An Augmented Double Linked Implementation

From the third column in figure 3.7, we observe that `last` and `add-after!` still exhibit an $O(n)$ behavior. The latter is due to the exceptional case that occurs when we omit the position argument. `add-after!` has to iterate in order to obtain the last position in that case. Our fourth and final implementation fixes this by storing an additional reference in the header that explicitly refers to the last node of the list. `length` is also turned into an $O(1)$ operation by simply storing the length of the list instead of computing it every time again.

Representation

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

```
(define-record-type positional-list
  (make h t s e)
  positional-list?
  (h head head!)
  (t tail tail!)
  (s size size!)
  (e equality))

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

Verification

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

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

Navigation

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

```
(define (last plst)
```

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

Manipulation

Since we maintain an explicit reference to the last node of the list, we no longer need expensive iterations to get to that node. However, storing an explicit reference to the last node and storing the length of the list has its price in the sense that all procedures that possibly affect these values need to take them into account. That is the reason why we have to reimplement all other manipulation procedures as well. Every time a node is added, the size has to be incremented and every time a node is removed, the size has to be decremented accordingly. Furthermore every operation that potentially modifies the last node, has to ensure that the header refers to the correct last node at all times. We illustrate this using `attach-middle!`; the other 5 procedures are left as an exercise.

```
(define (attach-middle! plst val pos)
  (define next (list-node-next pos))
  (define node (make-list-node val next pos))
  (list-node-next! pos node)
  (if (not (null? next))
      (list-node-prev! next node)
      (tail! plst node)); next null =¿ new last
  (size! plst (+ 1 (size plst))))
```

3.2.9 Comparing List Implementations

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

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

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

Figure 3.7 basically shows us that we can "buy time with space". By storing extra information in the data structure, we can speed up most of the operations significantly. But how much space is needed? Since a single linked list explicitly needs to store all the next pointers, a positional list of n elements typically requires $\Theta(2.n)$ references in memory: n to store the actual elements and n to store the next pointers. Similarly, a double linked list requires $\Theta(3.n)$ memory locations to store a list of n elements. For small memories, this cost can be significant. For example, embedded systems like household equipment (e.g. a microwave oven) typically have to deal with limited amounts of memory. In such cases, the vector implementation outperforms the linked ones.

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

Figure 3.7: Comparative List Performance Characteristics

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

3.3 Variations on Positional Lists

The implementation strategies studied in the previous sections not only apply to positional lists. We now present two alternative list ADTs — `list-with-current` and `ranked-list`. These ADTs solve a number of *conceptual* problems of positional lists. Hence, they can be considered as the result of an exercise in ADT design instead of implementation techniques: their implementations will be similar to those of positional lists; their abstractions will not.

3.3.1 The Problem

We have defined a linear data structure in section 3.2.1 as a collection of data elements that are associated with positions. Every position (except for the first and the last one) has a next position and a previous position. We have tried to be as abstract as possible in our conception of positions. In our implementations, positions have been represented by numeric vector indices, by pairs and by a dedicated record type.

Let us pick up our example of section 3.2.3 again. Remember that our professor has created a todo-list containing five entries: a lecture on strings, the preparation of the lecture on linearity, a rest period, a lecture on linearity and a lecture on sorting. In order to print a schedule for his students, our professor creates a plain Scheme list containing the positions of all three lectures:

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

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

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

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

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

Depending on the concrete implementation we use, this code gives rise to some serious problems. In the linked implementation, there is no problem. Even though the positional list has changed by adding the event for the night out, the values contained by the `lectures` list are still referring to the correct

positions in the positional list. However, in the vector implementation we get the following strange result on the screen:

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

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

The essence of the problem is that, in the *specification* of our positional list ADT, positions indicate *relative* positions in a particular list at a particular time. Positions are defined in terms of their neighbors. However, in a concrete *implementation*, these positions are implemented by a concrete data value in Scheme that directly refers to some pair or index in an array. Given the fact that the list evolves over time, "externalized" conceptually relative positions are no longer synchronized with the *absolute* technical ones that reside in the positional list. The reason is that the externalized positions do not evolve along when the list evolves. In order to mend the problem, it is necessary to render the positions *relative* to the representation of the list at *all* times. There are two ways to do this:

- The first option is not to solve the problem, but rather to turn it into a feature of the linear data structure. The main idea consist of putting the entire responsibility with the user of the ADT. In this option, *all* positions are *by definition* relative positions, i.e. they are positions that refer to a list element at a *single* moment in the time. Manipulating the list can change the list in such ways that the semantics of a whole bunch of positions changes in one strike. Lists that have this property will be called *ranked lists*. They are the topic of section 3.3.3.
- The second way to solve our problem is to prevent the problem from happening by *internalizing* positions in the list data structure. The idea is to store one position inside the list itself and then make all operations work relative to that special position. We shall call this position the *current position*. The result of this redesign is a new list ADT in which the notion of "externalized positions" has been replaced by that current position. All list operations are expressed relatively to that current position. This ADT is presented in the next section.

.

3.3.2 Relative Positions: Lists with a Current

The `list-with-current` ADT is shown below. The ADT has many similarities with the `position-list` ADT presented in section 3.2.2. Note that the abstract

notion of a position no longer appears in the definition of the ADT. Hence, the ADT is not parametrized by a data type `P` of positions. Furthermore, there are no operations (such as `first`) that "externalize" positions towards users. All navigation through the list is done using a "current position" that is maintained inside every list and that is hidden for the user of the ADT. There are several operations to manipulate that current position. For the sake of simplicity, we have omitted "non-essential" operations such as `map` and `for-each` from the ADT. Interestingly, `find!` becomes a destructive operation that makes the current of a list refer to the value found.

ADT **list–with–current** < V >

new
  ((V V → **boolean**) → **list–with–current** < V >)
from–scheme–list
  (**pair** (V V → **boolean**) → **list–with–current** < V >)
list–with–current?
  (**any** → **boolean**)
length
  (**list–with–current** < V > → **number**)
full?
  (**list–with–current** < V > → **boolean**)
empty?
  (**list–with–current** < V > → **boolean**)
set–current–to–first!
  (**list–with–current** < V > → **list–with–current** < V >)
set–current–to–last!
  (**list–with–current** < V > → **list–with–current** < V >)
current–has–next?
  (**list–with–current** < V > → **boolean**)
current–has–previous?
  (**list–with–current** < V > → **boolean**)
set–current–to–next!
  (**list–with–current** < V > → **list–with–current** < V >)
set–current–to–previous!
  (**list–with–current** < V > → **list–with–current** < V >)
has–current?
  (**list–with–current** < V > → **boolean**)
find!
  (**list–with–current** < V > V → **list–with–current** < V >)
update!
  (**list–with–current** < V > V → **list–with–current** < V >)
peek
  (**list–with–current** < V > → V)
delete!
  (**list–with–current** < V > → **list–with–current** < V >)

add−before!
(**list−with−current** < V > V → **list−with−current** < V >)
add−after!
(**list−with−current** < V > V → **list−with−current** < V >)

Not every operation leaves the data structure with a meaningful current position. For example, when launching a search using `find!` it might be the case that the target key was not found. In that case the current has no meaning. It is said to be *invalidated*. The current is also invalidated when a list is empty (e.g. right after creation) or after deleting the element associated with the current position (using `delete!`). To be able to deal with this, the ADT features an operation `has-current?` which can be used to test whether or not the current is referring to a meaningful data value in the list at a certain moment in time. Apart from this difference, the ADT is very similar to the positional list ADT. However, it does not suffer from the problem with externalized positions. As already said, we can apply our four implementation strategies to this ADT. We leave them as a programming exercise to the reader.

3.3.3 Relative Positions: Ranked Lists

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

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

ADT ranked−list < V >

new
  ((V V → **boolean**) → **ranked—list** < V >)
from—scheme—list
  (**pair** (V V → **boolean**) → **ranked—list** < V >)
ranked—list?
  (**any** → **boolean**)
length
  (**ranked—list** < V > → **number**)
full?
  (**ranked—list** < V > → **boolean**)
empty?
  (**ranked—list** < V > → **boolean**)
find
  (**ranked—list** < V > V → **number**)
peek—at—rank
  (**ranked—list** < V > **number** → V)
update—at—rank!
  (**ranked—list** < V > **number** V → **ranked—list** < V >)
delete—at—rank!
  (**ranked—list** < V > **number** → **ranked—list** < V >)
add—at—rank!
  (**ranked—list** < V > V . **number** → **ranked—list** < V >)

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

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

3.4 Searching in Linear Data Structures

Let us refer back to the table in figure 3.7. At first sight, the augmented double linked list implementation — shown in the fourth column — seems to be the

fastest implementation that we can achieve. All operations are in $O(1)$ except for the operations that *have* to process the entire list like `map`. They are in $O(n)$ "by definition". One might think that the same is true for `find`: since we have to compare the key with the data elements residing in the data structure, we have to traverse the entire data structure. Hence, $O(n)$ really seems the best we can do. This is true if our `find` is solely based on comparing elements that are linearly stored without any additional organization. In sections 3.4.2 and 3.4.3, we show that cleverly organizing a list's elements can speed up things considerably. Indeed, just keeping the elements in the list *sorted* will already allow us to speed up `find` up to the level of $O(log(n))$ for some implementations. Lists for which the data elements are always sorted are called *sorted lists*.

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

3.4.1 Sentinel Searching

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

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

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

```
                pos
                #f)))
    (detach-last! plst (last plst))
    res))))
```

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

Surely, the sentinel search algorithm is still $O(n)$, but the resulting code is "k times as fast" (for some constant k) as the naive sequential searching algorithm since only one test has to be executed in each step of the iteration. In the code, this is reflected by the fact that the `cond` expression has been replaced by a plain `if` expression. Since we have eliminated one of two tests, k will be close to 2 if our Scheme runs all tests equally fast. The advantage has its price though: the `attach-last!` and `detach-last!` operations have to be in $O(1)$. In the vector implementation this is indeed the case. In linked implementations, sentinels only make sense if the list's header stores an explicit reference to the last node of the list. All other implementations need to traverse the entire list to add the sentinel before the search and to remove the sentinel again after having searched the list. Clearly this cost is higher than the speedup we obtain from the avoided test. Notice too that the vector implementation also has to keep in mind never to fill the vector entirely: whenever a vector of n entries is allocated in computer memory, the list it represents has to be considered full once it contains $n - 1$ elements. The last entry has to be reserved to store a sentinel when a `find` is launched on a full list.

3.4.2 Sorted Lists

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

The idea of imposing additional organization on a linear data structure consists of making sure that the elements of the linear data structure are *always* stored in sorted order. Such lists are known as a *sorted lists* and their specification results in a new ADT called `sorted-list`. Because the elements of the list are always stored in sorted order, `find` gets more efficient because we can use the order of the elements to stop the search process earlier in this chapter. Indeed, given a sequential search, we know for sure that the key won't show up anymore in the search process as soon as the key is "smaller" than the element in the list being visited, since all elements that are visited from that point

onwards will turn out to be greater than the key[3].

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

sorted—list < V >

new
((V V → **boolean**)
(V V → **boolean**) → **sorted—list** < V >)
from—scheme—list
(**pair**
(V V → **boolean**)
(V V → **boolean**) → **sorted—list** < V >))
sorted—list?
(**any** → **boolean**)
length
(**sorted—list** < V > → **number**)
empty?
(**sorted—list** < V > → **boolean**)
full?
(**sorted—list** < V > → **boolean**)
find!
(**sorted—list** < V > V → **sorted—list** < V >)
delete!
(**sorted—list** < V > → **sorted—list** < V >)
peek
(**sorted—list** < V > → V)
add!
(**sorted—list** < V > V → **sorted—list** < V >)
set—current—to—first!
(**sorted—list** < V > → **sorted—list** < V >)
set—current—to—next!
(**sorted—list** < V > → **sorted—list** < V >)
has—current?
(**sorted—list** < V > → **boolean**)
current—has—next?
(**sorted—list** < V > → **boolean**)

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

For reasons of brevity, generic operations like `map` have been omitted. Also, the number of navigational operations that have to do with "the" current have been kept to a minimum: the ADT specifies the presence of a current but it can only be used to traverse the list in ascending order.

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

In what follows, we discuss a vectorial implementation. The linked implementations are left as a programming exercise. We invite the reader to compare various implementations of the `sorted-list` ADT from the point of view of their performance characteristics and the representation of the data structures. The vectorial representation is presented below.

The representation resembles the representation of vectorial positional list (presented in section 3.2.5). Basically, a sorted list is represented as a headed vector that stores the number of elements (i.e. `size`), the current, the actual vector (i.e. `storage`) and both the equality and the ordering procedures (i.e. `equality` and `lesser`). A sorted list is created using one of the constructors `new` and `from-scheme-list`.

```
(define default-size 20)

(define-record-type sorted-list
  (make-sorted-list s c v l e)
  sorted-list?
  (s size size!)
  (c current current!)
  (v storage)
  (l lesser)
  (e equality))

(define (make len <<? ==?)
  (make-sorted-list 0 -1 (make-vector (max default-size len)) <<? ==?))
```

The constructors `new` and `from-scheme-list` are implemented in two phases again: the private procedure `make` takes care of the memory allocation while

`new` and `from-scheme-list` take care of the proper initialization of the newly created sorted list.

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

(define (from-scheme-list slst <<? ==?)
  (let loop
    ((lst slst)
     (idx 0))
    (if (null? lst)
        (make idx <<? ==?)
        (add! (loop (cdr lst) (+ idx 1)) (car lst)))))
```

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

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

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

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

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

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

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

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

```
  (size! slst (- last 1))
  (current! slst -1)
  slst)
```

The focus of our attention is the implementation of `find!`. It takes a sorted list and a key. It traverses the list until the data element containing the matching key is found and makes the current refer to that data element. Since the list is sorted, `find!` can be optimized in the sense that it will not look any further if the element encountered during the traverse is greater than the key it is looking for. When this is the case, the test (`<<? (vector-ref vector idx) key`) returns #f and there is no point in continuing the search. This can speed up the implementation of `find!` considerably in the case of a negative answer. However, the result is still in $O(n)$ in the worst case. As we will see in section 3.4.3, the true benefit of using sorted lists stems from the fact that we are using a vector implementation for the `sorted-list` ADT.

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vect (storage slst))
  (define leng (size slst))
  (let sequential-search
    ((curr 0))
    (cond ((>= curr leng)
           (current! slst -1))
          ((==? key (vector-ref vect curr))
           (current! slst curr))
          ((<<? (vector-ref vect curr) key)
           (sequential-search (+ curr 1)))
          (else
           (current! slst -1))))
  slst)
```

The price to pay for the gain in speed for `find!` is a slower insertion of data elements in the list: the operation `add!` has to traverse the list in order to find the correct position of the element it is to insert. The worst-case performance characteristic is in $O(n)$ since we might end up traversing the entire list. Furthermore, in our vector implementation of the ADT, one also has to perform a storage move in order to shift the remaining elements "one entry to the right". Hence, sorted lists are not a very good choice when the composition of the list is unstable, especially when many insertions of new elements are expected.

```
(define (add! slst val)
  (define <<? (lesser slst))
  (define vect (storage slst))
  (define leng (size slst))
  (if (= leng (vector-length vect))
      (error "list full (add!)" slst))
```

```
(let vector-iter
  ((idx leng))
  (cond
    ((= idx 0)
     (vector-set! vect idx val))
    ((<<? val (vector-ref vect (- idx 1)))
     (vector-set! vect idx (vector-ref vect (- idx 1)))
     (vector-iter (- idx 1)))
    (else
     (vector-set! vect idx val))))
(size! slst (+ leng 1))
slst)
```

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

3.4.3 Binary Search

Although we have already presented two optimization techniques for `find`, the resulting procedures still have a performance characteristic in $O(n)$. An excellent searching algorithm — known as *binary search* — beats this performance characteristic up to $O(log(n))$. The algorithm is shown below. It explicitly relies on the fact that we have chosen a *vector* implementation for our *sorted* lists. Conform with the vector representation of sorted lists, the algorithm uses the value `-1` to invalidate the current when the key being searched for does not occur in the list.

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vect (storage slst))
  (define leng (size slst))
  (let binary-search
    ((left 0)
     (right (- leng 1)))
    (if (<= left right)
        (let ((mid (quotient (+ left right 1) 2)))
          (cond
```

```
          ((==? (vector-ref vect mid) key)
           (current! slst mid))
          ((<<? (vector-ref vect mid) key)
           (binary-search (+ mid 1) right))
          (else
           (binary-search left (- mid 1)))))))
      (current! slst -1)))
  slst)
```

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

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

3.5 Rings

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

ADT ring

new
($\emptyset \rightarrow$ **ring**)
from–scheme–list
(**pair** $\rightarrow$ **ring**)
ring?

(**any** → **boolean**)
add−after!
(**ring any** → **ring**)
add−before!
(**ring any** → **ring**)
shift−forward!
(**ring** → **ring**)
shift−backward!
(**ring** → **ring**)
delete!
(**ring** → **ring**)
update!
(**ring any** → **ring**)
peek
(**ring** → **any**)
length
(**ring** → **number**)

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

The following code shows a single linked implementation of the `ring` ADT. We start with the representation. Rings are represented as headed lists that simply maintain a reference to the ring's current element.

```
(define-record-type ring
  (make-ring c)
  ring?
  (c current current!))

(define (new)
  (make-ring '()))

(define (from-scheme-list slst)
  (let loop
    ((scml slst)
     (ring (new)))
```

```
    (if (null? scml)
        ring
        (loop (cdr scml) (add-after! ring (car scml))))))
```

Ring nodes are identical to the nodes used by the single linked implementation of positional lists discussed in section 3.2.6:

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

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

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

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

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

(define (iter-to-previous node)
  (let chasing-pointers
```

```
    ((prev node)
     (next (ring-node-next node)))
    (if (eq? node next)
        prev
        (chasing-pointers next (ring-node-next next)))))

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

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

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

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

The mutators for rings are implemented below. `add-after!` merely inserts a new node after the current. The implementation is $O(1)$ since we only manipulate next pointers. The implementation of `add-before!` is in $O(n)$ though. The reason is that we need a call to `iter-to-previous` because our single linked implementation does not provide an explicit back pointer to the previous node of the current node. This previous node is needed since we have to make it refer to the new one. The next node of the newly added node is the current.

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

(define (add-before! ring val)
```

```
  (define curr (current ring))
  (define node (make-ring-node val curr))
  (ring-node-next!
   (if (null? curr)
       node
       (iter-to-previous curr))
   node)
  (current! ring node)
  ring)
```

Likewise, `delete!` is in $O(n)$. In order for the result of `delete!` to be a valid ring, we have to make sure that the previous node of the current refers to the next node of the current. As such, we to call `iter-to-previous` again. Once again, a double linked implementation would change `add-before!` and `delete!` into $O(1)$ operations.

```
(define (delete! ring)
  (define curr (current ring))
  (if (null? curr)
      (error "empty ring (delete!)" ring))
  (ring-node-next!
   (iter-to-previous curr)
   (ring-node-next curr))
  (if (eq? curr (ring-node-next curr))
      (current! ring '())
      (current! ring (ring-node-next curr)))
  ring)
```

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

Examples

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

- A famous application of rings is the notion of a *round-robin task scheduler*. A scheduler is a program that maintains a ring of tasks. One can think of a task as a Scheme expression. A scheduler handles the tasks one by one by executing (a limited number of steps of) the task and by frequently moving on to the next task in the ring. E.g., a scheduler might choose to evaluate a number of subexpressions of a Scheme expression and then move on to the next Scheme expression in the scheduler. As a result all tasks have the impression of being executed simultaneously given the fact that they receive a fair amount of time by the scheduler. The resulting

behavior is called *time sharing* because all tasks have the impression to be executed on a separate computer even though they share the same computer. Schedulers are at the heart of operating systems such as Mac OS X, Linux, Windows and Unix since these are all task-based. Hence, the notion of a ring is buried deep down in almost every computer system.

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

3.6 Exercises

1. Write Scheme expressions for the headed lists and headed vectors shown in figure 3.8. Based on the drawings, try to distill a meaningful semantics for the extra information stored.

2. - Pick any implementation of the `positional-list` ADT (i.e. deciding on `P` is up to you) and use the operations of the ADT to construct the list '("hello" "world" "and" "goodday" "to" "me") by adding the words in the following order: "and", "me", "to", "goodday", "hello", "world".
   - Write a procedure which runs over the elements of the list and which counts the number of words that contain an #\e. Use a pattern matching algorithm of chapter 2.

3. Use your positional list of the previous exercise. Use `map` to generate a positional list of pairs (i.e. `V=pair`) that consists of a string and its length. In order to do so, define your own procedure `pair-eq?` that declares two pairs equal whenever they store the same number in their cdr. Subsequently , apply `find` in order to locate the word whose length is 7.

4. Change the single linked list implementation of the `positional-list` ADT by representing the nodes by vectors of size 2 instead of pairs.

5. Extend the `positional-list` ADT by an operation `accumulate` that takes a positional list, a null value and a combiner. Specify the procedural type of the operation and implement the operation. Can you implement the operation on top of the existing operations, or do you need access to the implementation details of the ADT?

6. Write a procedure which takes two `positional-list` arguments and which returns a `positional-list` that only contains the data elements contained by both argument lists (i.e. the intersection of both input lists).

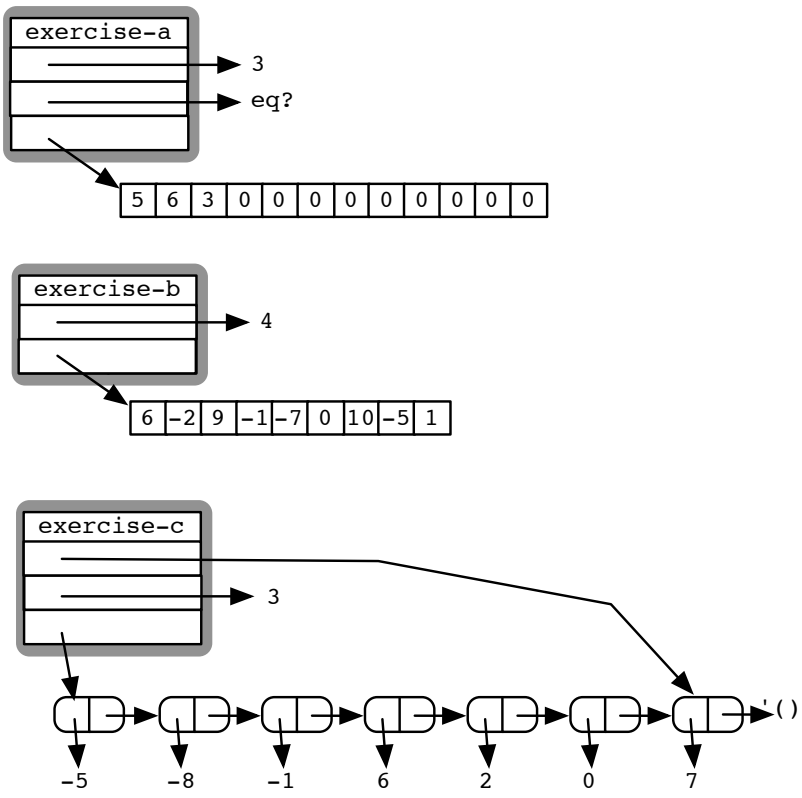

Figure 3.8: Exercise 1

7. Implement the `ranked-list` ADT using at least two of the four implementation strategies discussed. Can you imagine practical situations to defend each of the four implementations?

8. Implement a procedure `ternary-search` that resembles binary search except that it divides a sorted list in three instead of two parts in every phase of the iteration. What is the worst-case performance characteristic of your procedure? Can you identify the price to pay for the improvement? (*hint:* what does the implementation look like for 4-ary, 5-ary, 6-ary, ... searching?)

9. Write a procedure `sort` which takes a plain Scheme list and which returns a new list that consists of the same elements but in sorted order. Use the `sorted-list` ADT to implement your procedure. What is the worst-case performance characteristic of your implementation?

10. Rewrite the implementation of the `ring` ADT to get a maximal number of operations in $O(1)$.

11. Why is it impossible to get all `ring` ADT operations in $O(1)$ when opting for a vectorial implementation?

3.7 Further Reading

The material presented in this chapter can be found in any good book on algorithms and data structures. A more mathematical treatise of the performance characteristic of binary search can e.g. be found in [Lev02]. Much of the tradition of list processing comes from functional programming. Any good book on functional programming (such as [Bir98]) devotes a good deal of space to the implementation of list processing functions.