Chapter 4

Linear Abstract Data Types

Over the years, computer scientists have invented a number of ADTs that are extremely frequently used when constructing software systems. The `dictionary` ADT discussed in chapter 1 is an example of such an ADT. It is used so often by programmers that it deserves to be studied in great detail. This is the topic of chapter 6 and chapter 7. This chapter discusses three other ADTs that occur over and over in software applications. The ADTs studied — stacks, queues and priority queues — belong together in one chapter because they all exhibit *linear behavior*. By this we mean that the operations of the ADT *suggest* that the elements of the corresponding data structures are organized in a linear way, pretty much in the sense studied in the previous chapter. By simply looking at the definitions of these ADTs, straightforward reasoning leads us to the conclusion that they are just special cases of lists and that they are naturally implemented as such. Therefore, a large part of this chapter consists of applying the knowledge of the previous chapter to implement them. However, we will see that it is sometimes more beneficial to implement ADTs that exhibit linear behavior by non-linear data structures. An example of this is the priority queue ADT for which a non-linear implementation using heaps is the most optimal one. Heaps are also studied in this chapter. They illustrate that linear ADTs and linear data structures are not quite the same.

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

4.1 Stacks

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

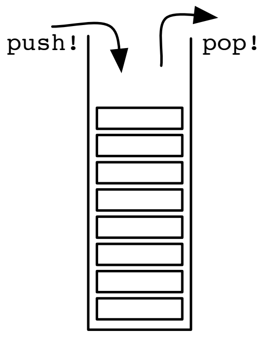

Figure 4.1: The Behaviour of a Stack

stack makes the stack shift all plates one position up. The essence of this behavior is that there are two major operations that one can apply on a stack: *push* an element on top of the stack and *pop* an element from the stack. The order in which elements are pushed onto the stack and popped from the stack is governed by the LIFO principle: last in, first out. At any point in the lifetime of a stack, the only element that can be popped from the stack is the element that was pushed most recently. Other additions and deletions are not allowed. The behavior of stacks is depicted in figure 4.1 This prescribed behavior of stacks suggests that they are best implemented by means of a linear data structure that grows and shrinks “at one end”.

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

Internet Browsers When browsing the internet, every time one clicks a link, that link is pushed on the internal stack of your browser. As such, the stack contains the navigational history of the user. Every time a link takes the user “deeper” into the net, the link is stored on the stack. At any time, the user can decide to go back in his or her navigational history by clicking the “Back” button. When clicking this button, the link which resides at the top of the stack is popped from the stack and used to reload and display the corresponding page.

Undo Modern applications usually feature an “undo” menu option. At any stage during the execution of the program, the user can select “undo” which reverses the most recent action taken by the user. Whereas older applications only used to remember the most recent action, modern applications are much “smarter” in the sense that they can unwind a huge number of user actions. This unwinding is to be done in reverse order. To implement this feature, the application maintains a stack of user actions. Every time the user selects an action in the application, that action is pushed onto the stack (e.g. “user typed a t”, “user selected ’save”’or “user dragged a diagram to a new location”) so that selecting “undo” always finds the most recent action on the top of the stack. By popping that action from the stack, the application can rewind the action, after which it finds the next to last action on the top of the stack again.

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

These are just two examples. Many other examples of stacks in computer science exist and this is the main reason for including stacks in a book on data structures. The following sections formally specify the ADT and study two alternative implementation strategies.

4.1.1 The Stack ADT

The `stack` ADT is shown below. A stack can contain values of any Scheme data type. We do not require any operations that will be applied on the values stored in the stack. Hence, there is no need to parametrize the ADT with `V`. This is in contrast with most of the ADTs presented in chapter 3 which assume the presence of an operator `==?` the procedural type of which is (`V V` → `boolean`) where `V` is the data type of the values stored. Since the `stack` ADT does not put any restrictions on the data type, it can be used with any Scheme data type. Hence, when creating “a” stack, we are actually dealing with a stack that can store `any` data type.

ADT stack

new
    ($\emptyset$ → **stack**)
stack?
    (**any** → **boolean**)
push!
    (**stack any** → **stack**)
top
    (**stack** → **any**)
pop!
    (**stack** → **any**)
empty?
    (**stack** → **boolean**)
full?
    (**stack** → **boolean**)

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

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

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

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

4.1.2 Vector Implementation

The first implementation of the `stack` ADT uses vectors. The Scheme definitions needed to represent a vector-based stack are given below.

```
(define stack-size 10)

(define-record-type stack
  (make f v)
  stack?
  (f first-free first-free!)
  (v storage))

(define (new)
  (make 0 (make-vector stack-size ())))
```

A stack is represented as a headed vector which keeps a reference to its actual vector as well as the first available position in that vector. Initially this number is 0. It is incremented on every push. `storage` is used to select the actual vector from the headed vector. `first-free` returns the first position available. Given this representation, the implementation of `empty?` and `full?` is not at all thrilling. `empty?` returns #`t` whenever the first position equals 0. `full?` returns #`t` as soon as the first free position is equal to the length of the vector. Remember that vector indices vary from 0 to the length of the vector. Both operations are obviously in $O(1)$.

```
(define (empty? stack)
  (= (first-free stack) 0))

(define (full? stack)
  (= (first-free stack)
     (vector-length (storage stack))))
```

The implementation for `push!`, `pop!` and `top` is given below. The stack is represented by a vector that "is filled from left to right". This is accomplished by storing new elements at the index returned by `first-free` and by incrementing that number (using `first-free!`) on every call to `push!`. Popping elements from the stack is realized by "emptying the vector from right to left". This is accomplished by peeking at the last meaningful element of the vector which resides at the first free position minus 1. Deleting that element is just a matter of decrementing the number returned by `first-free`. `top` merely peeks at the last meaningful position of the vector without changing anything.

```
(define (push! stack val)
  (define vector (storage stack))
  (define ff (first-free stack))
  (if (= ff (vector-length vector))
    (error "stack full (push!)" stack))
  (vector-set! vector ff val)
  (first-free! stack (+ ff 1))
  stack)

(define (top stack)
  (define vector (storage stack))
  (define ff (first-free stack))
  (if (= ff 0)
    (error "stack empty (top)" stack))
  (vector-ref vector (- ff 1)))

(define (pop! stack)
  (define vector (storage stack))
  (define ff (first-free stack))
  (if (= ff 0)
    (error "stack empty (pop!)" stack))
  (let ((val (vector-ref vector (- ff 1))))
    (first-free! stack (- ff 1))
    val))
```

What can we say about the performance characteristics of these procedures? One merely has to scrutinize their body to notice that none of them contains any recursion or iteration. Since all procedures called from the bodies are in $O(1)$, we are allowed to conclude that they are in $O(1)$ as well. In contrast to the vectorial implementation of positional lists, no storage move "to the right" or "to the left" are needed.

4.1.3 Linked Implementation

The linked implementation of the `stack` ADT is similar to the single linked implementation of the positional list ADT presented in section 3.2.6. Again, stacks are represented by headed lists that hold a reference to a regular single linked Scheme list. This is accomplished in the following code excerpt. The excerpt shows the representation. The operations are discussed below.

```
(define-record-type stack
  (make l)
  stack?
  (l scheme-list scheme-list!))

(define (new)
  (make ()))
```

The verification operations for stacks look as follows. `empty?` merely accesses the underlying Scheme list to check whether or not it is the empty list. Furthermore, a linked list is never full. Both operations are obviously in $O(1)$.

```
(define (empty? stack)
  (define slst (scheme-list stack))
  (null? slst))

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

Finally, we describe `push!`, `pop!` and `top`. For the linked implementation, we use the exact opposite strategy as the one used in the vectorial implementation: the stack grows "to the left" and shrinks "to the right". This is accomplished by implementing `push!` using a combination of `cons` and `scheme-list!`. Accordingly, `pop!` is conceived as a combination of `car` and `scheme-list!`. `top` is similar to `pop!` except that it does not use `scheme-list!` to destructively modify the underlying Scheme list.

```
(define (push! stack val)
  (define slst (scheme-list stack))
```

```
    (scheme-list! stack (cons val slst))
    stack)

  (define (top stack)
    (define slst (scheme-list stack))
    (if (null? slst)
      (error "stack empty (top)" stack))
    (car slst))

  (define (pop! stack)
    (define slst (scheme-list stack))
    (if (null? slst)
      (error "stack empty (pop!)" stack))
    (let ((val (car slst)))
      (scheme-list! stack (cdr slst))
      val))
```

In order to come up with the performance characteristics, we merely observe that none of the procedures use recursion or iteration. Hence, they are all in $O(1)$.

4.1.4 Discussion

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

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

Figure 4.2: Comparative Stack Performance Characteristics

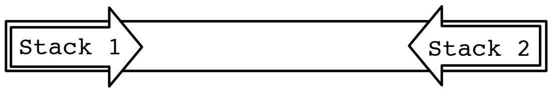

Figure 4.3: A pair of stacks in a vector

One variation of the `stack` ADT that occurs quite frequently in computer science is the `stack-pair` ADT. A `stack-pair` is a data structure that manages two stacks at the same time. The reason why this ADT is attractive is that it has an efficient vectorial implementation. The vectorial implementation of regular stacks needs an estimate of the capacity at creation time (c.f. `stack-size` in the vectorial implementation) which can result in painful wastes of memory. This pain is mitigated by the `stack-pair` ADT. The idea of the ADT consists of using the wasted memory to host a second stack. The vector contains one stack that "grows to the right" in the vector, and a second stack that "grows to the left" in the same vector. The principle is shown in figure 4.3. The specification of the ADT as well as its implementation is left as an exercise.

4.2 Queues

A second frequently occurring ADT that exhibits linear behavior is the `queue` ADT. There are many examples of queues in our daily lives. For example, at the cash register of a supermarket, customers queue up to pay for their groceries. A pile of documents that are to be processed by an administration is another example of a queue. Every time new documents arrive, they are put on the pile of documents to be processed and every time a secretary is ready to process a new document, he or she takes a document from the bottom of the pile. The defining property of a queue is that items appear at one end of the queue and disappear at the other end of the queue. In other words, queues exhibit a FIFO behavior: first in, first out. This kind of behavior is depicted in figure 4.4. The operation that adds a value to the rear of the queue is called *enqueue*. The operation that reads a value at the front of the queue (also called the *head* of the queue) is called *serve*.

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

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

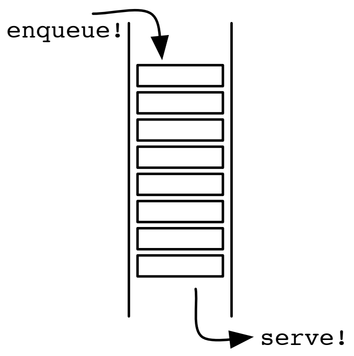

Figure 4.4: The behaviour of a queue

queue and prints it.

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

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

4.2.1 The Queue ADT

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

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

ADT queue

new

    ($\emptyset \rightarrow$ **queue**)
queue?
    (**any** $\rightarrow$ **boolean**)
enqueue!
    (**queue any** $\rightarrow$ **queue**)
peek
    (**queue** $\rightarrow$ **any**)
serve!
    (**queue** $\rightarrow$ **any**)
empty?
    (**queue** $\rightarrow$ **boolean**)
full?
    (**queue** $\rightarrow$ **boolean**)

4.2.2 Implementation Strategies

Unfortunately, implementing the `queue` ADT is not as trivial as implementing the `stack` ADT. The main reason is that the underlying linear data structure has to grow and shrink at different ends.

- In the vectorial implementation this means that the implementation for `enqueue!` can be implemented in $O(1)$ by storing the data elements in the "leftmost locations" of the vector and by allowing the queue to grow "to the right". However, this means that the implementation for `serve!` has to remove elements "from the left". In order to prevent the queue from "bumping" against the rightmost end of the vector, this requires us to do a storage move "to the left" on every occasion of `serve!`. Hence, serve would be in $O(n)$. Conversely, `serve!` is in $O(1)$ and `enqueue!` is in $O(n)$ if we decide to store the queue's elements in the "rightmost locations" of the underlying vector. Fortunately, there is a way out.

  There exist an extremely efficient vectorial implementation for queues that has an $O(1)$ performance characteristics for all the operations. It is based on what is often referred to as *circular vectors*. The principle behind the implementation is shown in figure 4.5. The key insight is that we do not have to keep all queue elements sitting at one end of the vector. The idea is to represent the queue in a vector by maintaining two indices in the vector: `head` designates the head of the queue and `rear` refers to the end of the queue. Whenever `enqueue!` is called, an element is added at the end of the queue, i.e. at `rear`. `serve!` removes an element from the beginning of the queue, i.e. at `head`. As a result, the queue "crawls" in the vector "from left to right". Whenever it bumps into the last position of the vector, it restarts at the very first position of the vector. As such, the queue crawls in a circular way. The queue is considered full whenever the beginning of the queue (circularly) bumps into its end. The implementation is presented in section 4.2.4.

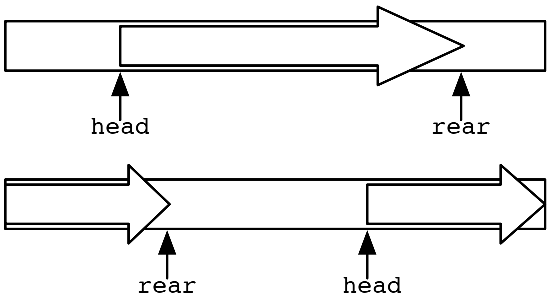

Figure 4.5: A circular vector implementation for queues

- In the single linked implementation, adding an element to the start of the list is in $O(1)$ but removing an element from the end of the list is in $O(n)$. Conversely, adding an element to the end is in $O(n)$ while removing it from the start of the list is in $O(1)$. Again, choosing `enqueue!` to be in $O(1)$ causes `serve!` to be in $O(n)$ and the other way around. However we know from our study of the `position-list` ADT that we can alleviate this problem by storing an additional reference to the last element of the list. Hence, the optimal linked implementation for queues consists of a single linked implementation which stores an explicit reference to the last node of the queue. This is the implementation presented in section 4.2.3.

4.2.3 Linked Implementation

Let us start by discussing the linked implementation. It is contained by the Scheme code shown below. As usual, we have a double layered constructor consisting of the procedures `new` and `make`. Again, we use ellipsis to replace the implementation of the procedures explained further below.

```
(define-record-type queue
  (make h r)
  queue?
  (h head head!)
  (r rear rear!))

(define (new)
  (make () ()))
```

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

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


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

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

```
(define (enqueue! q val)
  (define last (rear q))
  (define node (cons val '()))
  (if (null? (head q))
    (head! q node)
    (set-cdr! last node))
  (rear! q node)
  q)


(define (peek q)
  (if (null? (head q))
    (error "empty queue (peek)" q)
    (car (head q))))


(define (serve! q)
  (define first (head q))
  (if (null? first)
    (error "empty queue (serve!)" q))
  (head! q (cdr first))
  (if (null? (head q))
    (rear! q '()))
  (car first))
```

4.2.4 Vector Implementation

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

```
(define default-size 5)
(define-record-type queue
  (make s h r)
  queue?
  (s storage)
  (h head head!)
  (r rear rear!))

(define (new)
  (make (make-vector default-size) 0 0))
```

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

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

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

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

Checking whether or not a queue is empty or full is a bit tricky. A queue is considered full when its rear bumps into its head. The problem is that a queue is also considered empty if the `head` index is equal to the `rear` index. In order to be able to make a distinction between empty queues and full queues, we therefore "waste" one vector entry by considering the queue full whenever the successor of the rear bumps into the head.

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

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

All procedures are simple combinations of arithmetic and vector indexing. As a result, they are all in $O(1)$.

4.2.5 Discussion

Since both the vector implementation and the linked implementation have nothing but operations in $O(1)$, we can conclude that the choice to be made will depend on the amount of flexibility that is required. In an application where the size of the queue is easily established upfront, the vector implementation outperforms the linked implementation because it is more efficient regarding memory consumption (it does not have to store cdr pointers). If it is impossible to estimate the queue's size upfront, then a linked implementation is preferred. They are both equally fast.

4.3 Priority Queues

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

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

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

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

Emergencies Most hospitals have an emergency service that is open day and night and which has a limited staff that has to take care of all emergencies arriving at the hospital. Because of the limitation in manpower (e.g. at night), a choice must be made in order to determine which patients require faster treatment. Clearly, someone with a headache will have to wait if an ambulance arrives with a victim of a serious car crash. To formalize this (and to avoid giving people the impression that they are being treated unfairly), every arriving patient is assigned a number that indicates the order in which he or she will be treated. However, depending on how serious his or her symptoms are, the number is printed on a ticket with a different color. For example, three different colors — say red, orange and green — might be used to discriminate between "urgent", "serious" and "not urgent" symptoms. Hence, at any moment in time, the list of patients is ordered according to the number they are assigned and according to the color of the ticket. Clearly, $n+1$ is of higher priority than n when they are printed on a ticket of the same color. However, n has a higher priority than k when the color of the ticket on which n is printed has a higher priority than the color on which k is printed, even if n is a bigger number than k. Such rules can be used to implement an operator `>>?` on priorities.

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

4.3.1 The Priority Queue ADT

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

```
ADT priority−queue < P >

new
   ( ( P P → boolean ) → priority−queue < P > )
priority−queue?
   ( any → boolean )
enqueue!
   ( priority−queue < P > any P → priority−queue < P > )
peek
   ( priority−queue < P > → any )
serve!
   ( priority−queue < P > → any )
full?
   ( priority−queue < P > → boolean )
empty?
   ( priority−queue < P > → boolean )
```

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

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

4.3.2 Implementation with Sorted Lists

The first implementation is based on the `sorted-list` ADT presented in section 3.4.2. It is based on the fact that a single linked implementation of sorted lists has been selected. This will become clear later on. In the code shown below, we prefix all `sorted-list` operations with `slist:`. The principle of the implementation is simple: a priority queue is nothing but a linear data structure in which the elements are sorted by priority. Serving an element from the priority queue is then simply achieved by deleting the very first element from the sorted list. Enqueueing is achieved by using `add!` to add an element along with its priority to the sorted list.

The `sorted-list` ADT presented in section 3.4.2 is unaware of priorities. It just sorts "values". We therefore have to create a new kind of values that corresponds to the actual values to be stored in the priority queue, paired with their priority. This new kind of values is called a *priority queue item*. In other words, priority queue items are pairs consisting of "an ordinary value" and its associated priority.

```
(define pq-item-make cons)
(define pq-item-val car)
(define pq-item-priority cdr)
(define (lift func)
  (lambda (item1 item2)
    (func (pq-item-priority item1)
          (pq-item-priority item2))))

(define-record-type priority-queue
  (make s)
  priority-queue?
  (s slist))

(define (new >>?)
  (make (slist:new (lift >>?)
                   (lift eq?))))
```

Let us study the notion of a priority queue item first. Our library shows a procedure `pq-item-make` to create a priority queue item, a procedure `pq-item-val` to select the value from a priority queue item and a procedure `pq-item-priority` to access the priority of a priority queue item. Such a priority queue item will be created each time a value and its associated priority are enqueued. In other words, a priority queue is represented as an headed list that refers to a value of type `sorted-list<pq-item>`.

The sorted list implementation needs a "smaller than" operator and the constructor for priority queues receives a "higher priority than" operator. We therefore consider one priority queue item smaller than another priority queue item whenever the priority of the first item is higher than the priority of the second item. The procedure `lift` is a higher order procedure that takes care of this. It takes a procedure `func` that works on priorities and returns the "lifted" version of the procedure than can work on corresponding priority queue items. The lifted version of the procedure takes two priority queue items, selects their priorities and applies the original procedure to these priorities. This procedure is used in our implementation to lift the equality operator and the "higher priority than" >>? operator to priority queue items. The results of the calls (`lift` >>?) and (`lift eq?`) is thus used to order priority queue items in our sorted list implementation.

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

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

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

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

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

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

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

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

We know from section 3.4.2 that both `add!` and `delete!` are in $O(n)$ since we have to find the correct whereabouts of the newly added element in the list. Therefore, the performance characteristic of `enqueue!` is in $O(n)$. This is true for both implementations of sorted lists.

For `delete!`, section 3.4.2 displays $O(n)$ as well. However, this is because we have used a vectorial implementation for the sorted list ADT in section 3.4.2. Given a single linked list, this operation is in $O(1)$ when deleting the very first element of the list: all we have to do is forget the original first element of the list and make the list's header refer to the second element in the list. In other words, the best-case analysis for `delete!` in the single linked implementation is in $O(1)$. It occurs whenever we remove the very first element from the list. Since our priority queue implementation *always* removes the first element from the sorted list, we always find ourselves in the best-case of `delete!` for single linked sorted lists.

We conclude that, the sorted list implementation for priority queues has a performance characteristic that is in $O(n)$ for `enqueue!` and in $O(1)$ for `serve!`, provided that we select a linked implementation for the `sorted-list` ADT.

4.3.3 Implementation With Positional Lists

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

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

```
(define-record-type priority-queue
  (make l g)
  priority-queue?
  (l plist)
  (g greater))


(define (new >>?)
  (make (plist:new eq?) (lift >>?)))
```

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

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

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

The code for `enqueue!`, `serve!` and `peek` is more interesting. The costly `enqueue!` operation is replaced by an `enqueue!` operation that has a performance characteristic that is in $O(1)$: the element is just added to the beginning of the list, right before all other elements of the positional list. This operation is in $O(1)$ in all linked implementations. However, there is a price to pay for this fast implementation of `enqueue!`. Since items are always added to the priority queue without paying any attention to their priorities, a search process is needed in `serve!` and `peek` in order to determine the priority queue item with the highest priority. The `serve!` procedure shown below shows a loop that traverses the entire positional list using alternating applications of `has-next?` and `next`. While doing this, it computes the *position* of the highest priority and stores it in the variable `maximum-pos`. After finishing the loop, this variable contains the position of the priority queue element with the highest priority. We finish the procedure by calling `delete!` on that position. In order to come up with a performance characteristic for this version of `serve!` we might conclude that we need the double linked list implementation because `delete!` is in $O(n)$ for all other implementations. But even if we do use the double linked implementation, the `loop` is used to traverse the entire priority queue. Hence, `serve!` is in $O(n)$ (in fact we can even say that it is in $\Theta(n)$!). We omit the implementation of `peek` because it is entirely similar to `serve!`. The only difference is that it lacks the call to `delete!`.

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

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

4.3.4 Priority Queue Performance Characteristics

Figure 4.6 summarizes the performance characteristics for the two main operations of the `priority-queue` ADT. The important design decision to be made is whether or not the elements of the priority queue are *stored* in a way that takes the priorities into account. If this is the case, we end up with a cheap `serve!` but with an expensive `enqueue!`. If the organization in memory does not take the priorities into account as is the case with a simple positional list, then the `enqueue!` operation becomes cheap. However, the price to pay is a search process to determine the element with the highest priority. As a result, `serve!` gets expensive.

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

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

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

Figure 4.6: Comparative Priority Queue Performance Characteristics

4.4 Heaps

In this section, we present an auxiliary data structure — called *a heap* — that provides us with a much faster implementation for priority queues. Using heaps,

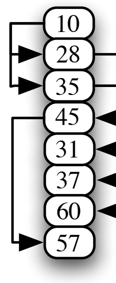

Figure 4.7: A Heap

both `serve!` and `enqueue!` exhibit a performance that is in $O(log(n))$. One might say that the workload is somehow spread over both operations.

4.4.1 What is a Heap?

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

Conceptually, a heap is a sequence of data elements e_1, e_2, ..., e_n that are *ordered* according to what is known as *the heap condition*: for all $i > 0$ we have $e_i < e_{2i}$ and $e_i < e_{2i+1}$. Figure 4.7 shows an example of a heap. In the figure, we have drawn an arrow from e_i to e_j whenever $e_i < e_j$. Surely, the order $<$ that is used in this definition depends on the data type of the elements stored. If we create a heap that contains numbers, then we should use Scheme's normal $<$ operator that compares numbers. If we create a heap of strings, then `string<?` might be used as the ordering needed to satisfy the heap condition.

It is important to understand that the elements in a heap are not necessarily sorted, even though any sorted sequence is also a valid heap since the elements of a sorted sequence automatically satisfy the heap condition . Given a number of data elements, there are many possible arrangements for those elements to form a valid heap. In other words, there does not exist a unique heap for a given set of data elements.

Caution is required when storing heaps in Scheme vectors. Since Scheme vectors are indexed starting from 0, this would cause us to satisfy the heap condition $e_0 < e_{2.0}$ which is impossible. Hence, *conceptually* heaps start counting from 1 even when their underlying storage vector starts counting from 0. This will require us to do the necessary index conversions in our implementation of heaps.

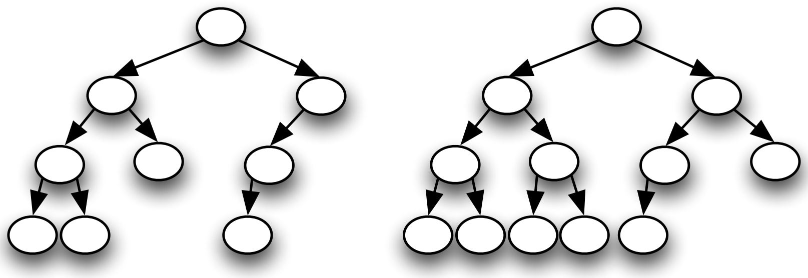

Figure 4.8: A Non-complete Binary Tree and its Completion

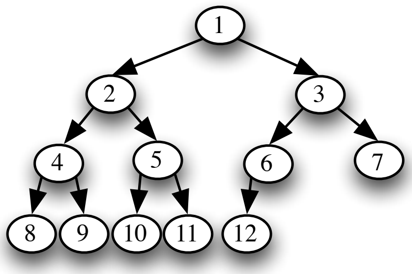

Figure 4.9: Indexing a Complete Binary Tree

Even though a heap is actually just a sequence of elements that happen to be ordered in some clever way, it pays off to think of a heap as if it were a *complete binary tree*. As you probably already know, a tree is a structure in which every element has a number of *children*. The element that refers to those children is called the *parent*. All elements in the tree are said to reside in *nodes*. The unique node that is not a child of any other node is called the *root node* of the tree. Nodes that have no children are called *leaf nodes*. *Binary trees* are trees in which every node has two (or less) children. Trees do not necessarily need to be *complete* as is illustrated in figure 4.8 which shows a non-complete tree on the left hand side and its completed version on the right hand side. A complete tree is a tree in which all levels are either entirely filled, or partially filled but in such a way that all the positions which are left of a certain node, also contain nodes. A complete tree is a tree that shows no gaps when "read" from left to right, level by level.

The reason why completeness of trees is such an important property is that it forms the basis for seeing the link between a tree and its representation as

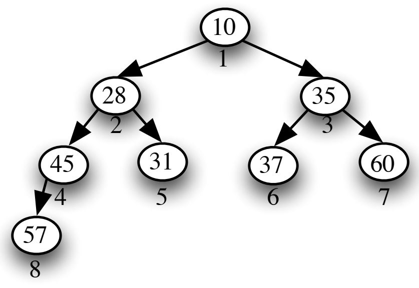

Figure 4.10: A Heap Drawn as a Complete Binary Tree

a sequence. Indeed, since a complete tree does not contain any gaps, we can assign numbers (called *indices*) to its nodes from left to right, level by level. Figure 4.9 shows how this indexing scheme is applied to the completed tree of figure 4.8. If the tree were to contain gaps, this would not be possible in an unambiguous way. It *would* be possible to take an incomplete tree and assign numbers to its nodes. However, given a series of numbered nodes, there is no way to reconstruct the tree since the numbers do not suffice to know where the gaps should be. Hence, the interchangeability of a tree and its representation as a numbered sequence relies on the fact that the tree is complete.

The indexing scheme allows a complete tree to be stored in a vector. The children of a node residing in index i in the vector are to be found at indices $2i$ and $2i+1$ in the vector (check this!). Similarly, given any index i in the vector, then the parent node of the node residing at that index resides in vector entry $\lfloor \frac{i}{2} \rfloor$.

Now that we know precisely how to think of heaps as complete binary trees, we can use this equivalence to draw the heap displayed in figure 4.7 as a tree. It is shown in figure 4.10. Using our indexing scheme, we can reformulate the heap condition as the requirement which states that every element in the heap has to be smaller than the elements residing in both of its subtrees. As a consequence, the root node of the heap always contains the smallest element of the heap.

For any node in the tree we define the *height of the node* as the length of the longest path from that node to a leaf node. The length of a path is defined as the number of arrows in the path. The *height of the heap* is the height of the root node, i.e. the length of the longest path in the heap. As an example, the height of the heap depicted in figure 4.10 is 3.

4.4.2 Properties of Heaps

Thanks to the correspondence between heaps and complete binary trees, it is possible to derive three useful mathematical properties for heaps. These

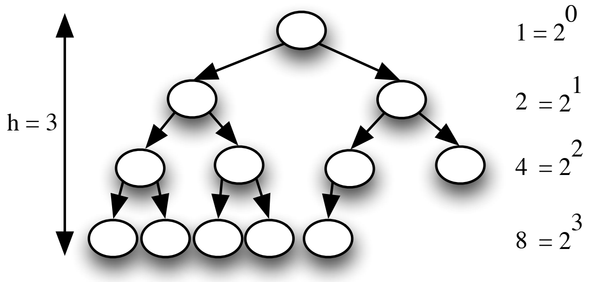

Figure 4.11: Number of Elements in a Heap

properties will turn out to be essential when deriving performance characteristics for heap-based algorithms.

The **first property** is a relation between the number of elements in a heap, say n, and the height of the heap, say h. In figure 4.11 we show a heap of height 3. As can be observed from the drawing, every level[1] — except the last — is full: the i^{th} full level contains 2^i elements. The deepest level residing at the level h contains between 1 and 2^h nodes. This means that a heap of height h has minimum $\sum_{i=0}^{h-1} 2^i + 1$ nodes and maximum $\sum_{i=0}^{h} 2^i$ nodes: $\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i$. Using the fact that for a geometric series $\sum_{i=0}^{h} a^i = \dfrac{a^{h+1}-1}{a-1}$, we get $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$. Therefore $h \leq log_2(n) < h+1$ and thus $h = \lfloor log_2(n) \rfloor$. This relation between the number of elements in a heap and the height of a heap is frequently used when establishing the performance characteristics for the heap operations.

A **second useful property** is that a heap with n elements has $\lceil \frac{n}{2} \rceil$ leaves. The proof for this uses contradiction. In a heap, all non-leaves sit in the leftmost locations of the vector. The leaves sit in the rightmost locations. Suppose that a heap contains less than $\lceil \frac{n}{2} \rceil$ leaves. Consider the very last element that is not a leaf. Its index $i > \lceil \frac{n}{2} \rceil$. But since it is not a leaf, it needs to have at least one child residing at $2.i > n$ which is impossible. Suppose that a heap contains more than $\lceil \frac{n}{2} \rceil$ leaves (i.e. fewer than $\lfloor \frac{n}{2} \rfloor$ non-leaves). Then the very last element of our heap (sitting at location n) has a parent that is also a leaf (because it sits at location $\lfloor \frac{n}{2} \rfloor$) . This is impossible since leaves cannot have children by definition.

A third useful property is that, *in a heap with n nodes there are maximum* $\lceil \frac{n}{2^{h+1}} \rceil$ *elements residing at height h.* This is not hard to see. At height 0 we

[1]Notice that we start counting “levels” from the root whereas the “height” is counted from a leaf.

only find leaf nodes and we know that a heap has $\lceil \frac{n}{2} \rceil$ leaves. One level higher in the heap, at height 1, we have half the number of nodes we have in the lowest layer, i.e. $\lceil \frac{n}{4} \rceil$. This is because we have 1 parent for every 2 nodes (except maybe for one). Hence, at height h, we find $\lceil \frac{n}{2^{h+1}} \rceil$. This property also turns out to be useful to establish performance characteristics.

4.4.3 The Heap ADT

.

Below we specify heaps in the form of a `heap` ADT.

```
ADT heap < V >

from−scheme−vector
  ( vector ( V V → boolean ) → heap < V > )
new
  ( number ( V V → boolean ) → heap < V > )
full?
  ( heap < V > → boolean )
empty?
  ( heap < V > → boolean )
insert!
  ( heap < V > V → heap < V > )
delete!
  ( heap < V > → V )
peek
  ( heap < V > → V )
length
  ( heap < V > → number )
```

The ADT specifies two ways to create a new heap. `new` takes a number indicating the capacity of the newly created (empty) heap. `from-vector` takes an existing Scheme vector and stores the vector as a heap. The elements of the vector are rearranged by the constructor in order to make them satisfy the heap condition. Both constructors take a procedure `<<?` the procedural type of which is (`V V` → `boolean`). This determines the ordering that is used to arrange the elements in the heap and to keep the heap condition satisfied at all times. `empty?` should be self-explaining. `full?` returns `#t` whenever the heap stores as many elements as indicated by the capacity.

`insert!` takes a (non-full) heap and a data element. It adds the element to the heap thereby rearranging the heap so that its elements meet the heap condition again. Because of the heap condition, it is always the case that the element sitting in the very first position of the heap is the smallest element. One can think of that element as the root of the complete binary tree that corresponds to the heap. It is smaller than its two children, which are in their turn smaller than their children, etc. This is the element returned by `peek`. `peek` does not remove the element from the heap. `delete!` on the other hand, returns the smallest element from the heap, removes it from the heap and rearranges the heap in order for its remaining elements to satisfy the heap condition again.

4.4.4 The Heap Representation

The following code excerpt shows the representation. A heap is represented by an headed vector containing a vector, the heap size (i.e. the number of elements that sit in the heap at a certain moment in time) and the comparison operator `lesser` that will be used to organize the heap in order for it to satisfy the heap condition.

```
(define-record-type heap
  (make v s l)
  heap?
  (v storage storage!)
  (s size size!)
  (l lesser))

(define (new capacity <<?)
  (make (make-vector capacity) 0 <<?))
```

The implementations of `length`, `full?` and `empty?` are trivial. As explained, satisfying the heap condition implies that the very first element of the heap is always the smallest element in the heap. Hence, `peek` simply requires us to peek into the first position of the underlying vector.

```
(define (peek heap)
  (if (empty? heap)
    (error "heap empty" heap)
    (vector-ref (storage heap) 0)))
```

The implementations of the operations `insert!` and `delete!` are a bit more contrived. After all, we cannot just insert or delete elements anyplace in the heap. This is most likely to violate the heap condition. We therefore restrict insertion and deletion to some very specific cases. The idea is to guarantee that the very first element of the heap is the only element that is ever deleted from a heap. Conversely, we only allow an element to be added to the rear of the heap. But even with this restricted insertion and deletion scheme, the heap condition is easily violated. Two potential problems can arise:

- First, when removing the very first element of the heap, we have to replace it by another element for otherwise the heap does not have a first element which makes the resulting data structure violate the heap condition. It would correspond to a complete binary tree that has no root. To resolve this situation, we replace the very first element by the very last element of the heap. A potential problem then is that the element that appears in the very first location like this, is too big and violates the heap condition (remember that it has to be smaller than the elements sitting at its child nodes). In other words — if we think of the heap as a complete binary tree — the element resides in "too high a level" in the heap. It therefore has to be *sifted down* a few levels. This is the task of the private procedure `sift-down` explained in section 4.4.5.

- Second, when adding a new element to the rear of the heap, the element might be too small in the sense that it resides in "too low a level" in the heap. The element belongs in a higher level if we think about the heap as a complete binary tree. *Sifting up* the element a few levels is the responsibility of the procedure `sift-up` that is explained in section 4.4.5 as well. Just like `sift-down`, it will be private to our library.

Given the procedures `sift-down` and `sift-up`, then the implementation of `insert!` and `delete!` looks as follows:

```
(define (insert! heap item)
  (if (full? heap)
    (error "heap full" heap))
  (let∗ ((vector (storage heap))
         (size (size heap)))
    (vector-set! vector size item)
    (if (> size 0)
      (sift-up heap (+ size 1)))
    (size! heap (+ size 1))))

(define (delete! heap)
  (if (empty? heap)
    (error "heap empty" heap))
  (let∗ ((vector (storage heap))
         (size (size heap))
         (first (vector-ref vector 0))
         (last (vector-ref vector (− size 1))))
    (size! heap (− size 1))
    (if (> size 1)
      (begin
        (vector-set! vector 0 last)
        (sift-down heap 1)))
    first))
```

`insert!` adds the new element to the last location of the heap and then sifts it up. This will rearrange the heap so that it is guaranteed to satisfy the heap condition again. Similarly, `delete!` replaces the first element of the heap by the very last one. Subsequently, the new element residing in the first position is sifted down in the heap in order to make the entire vector satisfy the heap condition. The original first element is returned from `delete!`.

4.4.5 Maintaining the Heap Condition

Let us now have a look at how the heap condition is restored by the sift-down and sift-up procedures. Remember from section 4.4.1 that the indexing scheme for heaps starts at 1 even though the indexing for vectors in Scheme starts counting from 0. To bridge this difference, both sift-up and sift-down have their own local version of vector-ref and vector-set! that perform the necessary index conversions. This can be observed in the let expression that constitutes the body of both procedures.

sift-up takes an index idx and assumes that idx is the last position of the heap. sift-up takes the element residing in that last position and percolates it to a higher location in the heap (i.e. to a location closer to the root of the heap). sift-up is an iterative process that reads the element at the idx position and moves it up the heap by shifting down all elements it encounters in the iterative process. This process continues until the element to be sifted up has reached its destination or until the root of the heap is reached. At that point, the element is stored in the vector. In every step of the iteration, the loop sift-iter computes the vector index of the parent using the expression (div child 2). In the body of the loop, we observe a conditional. The first branch checks whether we have arrived at the root. If this is the case, the element is stored in the root of the heap. The second branch checks whether the heap condition would be violated if we were to store the element at the current location. If this is the case, we store the element at the current location one level down and we continue the sifting process. The third branch is only reached if storing the element at the current location does not violate the heap condition. In that case, the element is simply stored in the vector entry that was freed by the previous iteration of the loop.

```
(define (sift-up heap idx)
  (let
      ((vector-ref
        (lambda (v i)
          (vector-ref v (- i 1))))
       (vector-set!
        (lambda (v i a)
          (vector-set! v (- i 1) a)))
       (vector (storage heap))
       (size (size heap))
       (<<? (lesser heap)))
    (let sift-iter
      ((child idx)
       (element (vector-ref vector idx)))
      (let ((parent (div child 2)))
        (cond ((= parent 0)
               (vector-set! vector child element))
              ((<<? element (vector-ref vector parent))
               (vector-set! vector child (vector-ref vector parent))
```

```
                    (sift-iter parent element))
                   (else
                    (vector-set! vector child element)))))))
```

`sift-down` assumes that an `element` sitting at position `idx` is the root of a heap for which all other elements are guaranteed to satisfy the heap condition. Hence, if we think of the heap as a complete binary tree, then `sift-down` assumes that the `element` sitting at the root may be violating the heap condition when it is compared with its children. However it assumes that the two subheaps that start at the children of the root are correct heaps. `sift-down` percolates the `element` down the heap (by shifting other elements up) until it reaches a location where it can be stored such that the overal data structure satisfies the heap condition.

```
(define (sift-down heap idx)
  (let
      ((vector-ref
        (lambda (v i)
          (vector-ref v (- i 1))))
       (vector-set!
        (lambda (v i a)
          (vector-set! v (- i 1) a)))
       (vector (storage heap))
       (size (size heap))
       (<<? (lesser heap)))
    (let sift-iter
      ((parent idx)
       (element (vector-ref vector idx)))
      (let*
          ((childL (* 2 parent))
           (childR (+ (* 2 parent) 1))
           (smallest
            (cond
              ((< childL size)
               (if (<<? (vector-ref vector childL)
                        (vector-ref vector childR))
                   (if (<<? element (vector-ref vector childL))
                     parent
                     childL)
                   (if (<<? element (vector-ref vector childR))
                     parent
                     childR)))
              ((= childL size)
               (if (<<? element (vector-ref vector childL))
                 parent
                 childL))
              (else parent))))
        (if (not (= smallest parent))
          (begin (vector-set! vector parent (vector-ref vector smallest))
                 (sift-iter smallest element))
          (vector-set! vector parent element))))))
```

At every level in the iteration, `sift-down` compares the `element` residing at the current `parent` with the elements residing at the (two or less) children. If the index `smallest` of the smallest element of this 3-way comparison resides in one of the children (i.e. `(not (= smallest parent))`, then that element of the child is copied into the parent and the iteration is continued with the child. Once the smallest element is the one sitting in the current parent, that parent is replaced by our element. This is safe since the element sitting in the current parent was moved up one level in the previous iteration anyhow.

Notice that calculating the index `smallest` is a bit tricky. When `(< childL size)`, we are sure that we have encountered a parent that effectively has two children. This requires a 3-way comparison to find the index of the smallest element. However, if `(= childL size)`, we have a parent that only has a single (left) child. Determining the smallest element thus requires a 2-way comparison of the child with the parent. In the remaining case, we have reached a parent that has no children which is therefore automatically the smallest element.

`sift-down` can be considered as an operation that merges two heaps. The `element` residing at location `idx` can be thought of as the root of a complete binary tree that is not a heap yet. However, both subtrees are valid heaps. By sifting the `element` down (into one of the subtrees), the entire tree with root `idx` becomes a heap as well.

4.4.6 Heap Performance Characteristics

Now that we have presented an implementation for all the heap operations, it is time to establish their performance characteristics. Looking back at the implementations for `insert!` and `delete!`, we observe that their entire body is $O(1)$ except for the call to `sift-down` or `sift-up`. The implementation of `sift-down` and `sift-up` consist of expressions that are in $O(1)$ except for the loops `sift-iter` that traverse the heap. Hence, in order to come up with a performance characteristic for `insert!` and `delete!`, we have to find out how often the `sift-iter` loop is executed in both cases (i.e. $r(n)$). In the `sift-iter` loop of `sift-up`, we notice that the number `child` is divided by 2 in every step of the iteration. Since we start with n (i.e. the size of the heap) the question becomes how often we can divide a number (using integer division) before we reach zero. In other words, for which k is $\lfloor \frac{n}{2^k} \rfloor = 0$? Clearly the answer is $k = \lfloor log_2(n) \rfloor$. Hence $r(n) \in O(log(n))$. Similarly, the `sift-iter` loop of `sift-down` starts from 1 and multiplies its iteration variable `parent` by two until the correct position in the heap is reached. Clearly, the last possible position that can be reached this way is n, the size of the heap. Hence, the question becomes how often we can double 1 before we reach n. In other words, for which k is $2^k = n$? Again the answer is $k = \lfloor log_2(n) \rfloor$ and thus $r(n) \in O(log(n))$. Hence, both `sift-down` and `sift-up` are in $O(log(n))$. Another way to understand this result is to notice that — in the worst case — both `sift-down` and `sift-up` traverse the complete binary tree that corresponds to the heap. `sift-down` starts at the root and moves its way down to a leaf of the tree (in the worst-case). Conversely, `sift-up` starts at the last position (i.e. a leaf of the tree) and moves its way up to the root (in the worst-case). In both cases, the number of iterative steps is bound by the height of the tree which is $h = \lfloor log_2(n) \rfloor$ as explained in section 4.4.2.

Hence, `insert!` and `delete!` are in $O(log(n))$.

4.4.7 Building a heap

Now that we have presented the basic operations of heaps as well as a way to maintain the heap condition, we show how to build a heap given a randomly ordered vector. The implementation of `from-vector` takes a vector and a comparator <<?. It turns the vector into a heap by packing it into a headed list and by reorganizing its elements. This process is usually referred to as the *heapification* of a vector.

```
(define (from-scheme-vector vector <<?)
  (define size (vector-length vector))
  (define heap (make vector size <<?))
  (define (iter index)
    (sift-down heap index)
    (if (> index 1)
      (iter (− index 1))))
  (iter (div size 2))
  heap)
```

At this point, it is useful to think about the heap as a complete binary tree again. If we consider a vector containing n elements as a complete binary tree, then the $\frac{n}{2}$ last elements of the vector form the bottom level (called the *yield*) of the tree. All the elements residing in the yield are 1-element heaps by definition. Hence, these do not need to be considered in order to build the heap such that the iteration to build the heap can start in the middle of the vector (i.e. we start at (`div size` 2)). The heap construction process counts backward from the middle of the vector down to the first element of the vector. In every phase of the iteration, an element is considered as the root of a new heap that has to be built on top of two smaller subheaps that result from the previous step of the iteration. Since the new root might violate the heap condition, it has to be sifted down the new heap. In terms of complete binary trees, a new complete binary tree is made based on two previously constructed complete binary subtrees.

The implementation of `from-vector` calls `iter` for all $\frac{n}{2}$ elements that are not leaves in the newly built heap. `iter` calls `sift-down` for every element and `sift-down` is $O(log(n))$. Hence, $O(n.log(n))$ is a worst-case performance characteristic for `from-vector`. Although this naive analysis is correct, we get significantly better results by delving a bit deeper into the mathematical properties of a heap.

Remember from section 4.4.2 that a heap with n elements has at most $\lceil \frac{n}{2^{h+1}} \rceil$ elements residing at height h. When adding an element to the heap at height h, this causes `sift-up` to do $O(h)$ work. Since this has to be done for all nodes residing at height h, constructing the heaps at height h requires $\lceil \frac{n}{2^{h+1}} \rceil O(h)$ computational steps. This has to be done for all "heights" going from 0 to the height $\lfloor log(n) \rfloor$ of the entire heap. Hence, the total amount of work is $\displaystyle\sum_{h=0}^{\lfloor log(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^h})$. Since $\displaystyle\sum_{k=0}^{\infty} k.x^k = \frac{x}{(1-x)^2}$, we can use this result for $x = \frac{1}{2}$ which yields $\displaystyle\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. Hence we get $O(n \displaystyle\sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^h})$ $= O(n \displaystyle\sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$ which is a better result than the $O(n.log(n))$ given by the naive analysis presented above. In other words, `from-vector` builds a heap from any vector in linear time.

The performance characteristics for our entire `heap` ADT implementation is summarized in figure 4.12. Notice that our implementation of `from-vector` destructively modifies the argument vector. If this behavior for `from-vector` should be unwanted, we have to make a new vector and copy the argument vector. This operation is $O(n)$ as well such that it does not affect the performance characteristic for `from-vector`.

Operation	Performance
`new`	$O(1)$
`empty?`	$O(1)$
`full?`	$O(1)$
`from-vector`	$O(n)$
`insert!`	$O(log(n))$
`delete!`	$O(log(n))$
`peek`	$O(1)$
`length`	$O(1)$

Figure 4.12: Heap Performance Characteristics

4.4.8 Priority Queues and Heaps

Remember that the main reason for studying heaps is that they provide us with an extremely efficient implementation for priority queues. Below we present the heap implementation of the `priority-queue` ADT that was presented in section 4.3. Representing a priority queue by means of a heap is not very different from the representation that uses sorted lists or positional lists. A priority queue is represented by a header that maintains a reference to a heap. Just like in the sorted list implementation and the positional list implementation, the heap implementation for priority queues actually stores priority queue items which are pairs that consist of an actual value and its associated priority. The "higher priority than" operator of the constructor >>? is used as the "smaller than" operator that is needed by the heap ADT. In other words, one priority queue item is smaller than another priority queue item whenever the first has a higher priority than the second.

```
(define-record-type priority-queue
  (make h)
  priority-queue?
  (h heap))

 (define default-size 50)

(define (new >>?)
  (make (heap:new default-size (lift >>?))))
```

Given the abstractions provided by the `heap` ADT, the implementations of `enqueue!`, `serve!` and `peek` are quite simple. `enqueue!` creates a new priority queue item and inserts the item in the heap. The heap does the rest as it will sift the item to some position needed to satisfy the heap condition. `serve!` deletes a priority queue item from the heap (by removing its smallest element) and retrieves the value of that item. Again, the heap does the sifting necessary to make a new smallest element appear in the root of the heap. `peek` is entirely equivalent. `empty?` and `full?` are trivial and are therefore omitted.

```
(define (serve! pq)
  (if (empty? pq)
    (error "empty priority queue (serve!)" pq)
    (pq-item-val (heap:delete! (heap pq)))))

(define (peek pq)
  (if (empty? pq)
    (error "empty priority queue (peek)" pq)
    (pq-item-val (heap:peek (heap pq)))))

(define (enqueue! pq value pty)
  (heap:insert! (heap pq) (pq-item-make value pty))
  pq)
```

The implementation uses the fact that — because of the heap conditions — the smallest element (i.e. the element with highest priority) always resides at the root of the heap. Hence, the heap implementation for priority queues is similar to the sorted list implementation: serving an element from the priority queue merely requires us to remove the first element. In a sorted list implementation this had no further implications since the rest of the list is sorted as well. In a heap implementation, removing the first elements requires us to do the sifting necessary to restore the heap property for the remaining elements. This process takes $O(log(n))$ work. Figure 4.13 is a repetition of figure 4.6. We invite the reader to look back at figure 1.5 in order to understand that this is an extremely powerful result. E.g., for enqueueing an element in a priority queue that contains one million elements, only twenty computational steps are required.

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

Figure 4.13: Comparative Priority Queue Performance Characteristics

4.5 Exercises

1. Implement a procedure `postfix-eval` that evaluates a Scheme list representing expressions in postfix notation. For example, (`postfix-eval` '(5 6 +)) should return 11 and (`postfix-eval` '(5 6 + 7 −)) should return 4. You can use the predicate `number?` to test whether or not a Scheme value is a number.

2. XML is a language that allows one to represent documents by including data it in arbitrarily deep nestings of "parentheses". Instead of using real parentheses like (and) or [and], XML allows us to define our own parentheses. Every string that is included in angular brackets < and > is considered to be an "opening" parenthesis. The corresponding closing parenthesis uses an additional slash in front of the string. For example, <open> is an opening parenthesis. Its corresponding closing parenthesis is </open>. For example, the list '(<html> <head> This is the head </head> <body> And this is the body </body></html>) could be a valid XML document. Notice that we can nest these "parentheses" in an arbitrarily deep way. Write a procedure (`valid? lst`) that takes a list of Scheme symbols and that checks whether or not the list constitutes a valid XML document. You will need `symbol->string` to convert the symbols to strings which you can further investigate using `string-length` and `string-ref`. Write auxiliary procedures `opening-parenthesis?` and `closing-parenthesis?` that check whether or not a given symbol is an opening or closing parenthesis. Also write a procedure `matches?` that takes two symbols and that checks whether they both represent an opening parenthesis and its matching closing parenthesis. The `substring` explained in chapter 2 procedure may simplify your procedures.

3. The *Josephus Problem* for a given number m is a mathematical problem where n people, numbered 1 to n sit in a circle. Starting at person 1, we

count m people in a circular way. The last person in the count is removed from the circle[2] after which we start counting m people again starting at the person sitting next to the person that was removed. And so on. The circle is getting smaller and smaller and the person that remains wins[3]. It is possible to solve the Josephus problem in a mathematical way. However, in this exercise we will write a simulation procedure `josephus` that takes n and m and which sets up an iterative process to simulate the flow of events and which returns the number of the winning person. Use the `queue` ADT to formulate the procedure.

4. Consider implementing the `stack` and `queue` ADTs on top of our `positional-list` ADT of chapter 3. For both ADTs, consider implementations based on all four implementations of the `positional-list`. What are the performance characteristics for the four basic operations (namely `push!` and `pop!` for stacks and `enqueue!` and `serve!` for queues) ?

5. Design and implement the `stack-pair` ADT discussed in section 4.1.4. How can you ensure that all operations are in $O(1)$ for both stacks?

6. A deque (or double ended queue) is a mixture of queues and stacks that allows removing elements at both ends. Formulate the `deque` ADT and provide both a vectorial and a linked implementation for which all operations are in $O(1)$.

7. Implement the >>? operator for the hospital emergency service discussed at the beginning of section 4.3.

8. Consider the heap created using the expression (`from-scheme-vector` (`vector` 25 2 17 20 84 5 7 12) <).

   - What is the parent of the element sitting at index 3?
   - Which element in the heap does not have a parent?
   - Which element in the heap does not have a left child? Which element only has a left child?
   - What is the height of the heap? Implement an operation `height` that returns the height of any heap. Give the procedural type of the operation.
   - Is the following statement true or false? "The value sitting at the root of a subheap of a heap is always the smallest element of all values contained by that subheap."

9. What can you say about the location of the greatest element of a heap?

10. Assume you have an empty `heap` with comparator <. Using `insert!`, we add the elements $5, 2, 3, 1, 2, 1$ in that order. Draw every phase of the

[2]In the original formulation of the problem, this person was killed.
[3]In the original formulation, this was the person released.

heap during the construction. Now remove two elements from the heap and redraw the result. In all phases of the exercise, draw the heap as a complete binary tree and draw the underlying vector as well.

11. An n-ary heap is a heap where all nodes have n children instead of just 2.

    - Consider representing n-ary heaps using vectors. How do you determine the children of a node? How do you determine the parent of a node?
    - What is the height of an n-ary heap that contains N elements?
    - What is the performance characteristic of `insert!` and `delete!` ?
    - Implement n-ary heaps.

12. Read about Huffman encodings. Write a procedure (`huffman-tree freqs`) that takes a list of leaves that consist of a symbol and the frequency with which it occurs in a text. The procedure returns the Huffman tree in an iterative way by using a priority queue. It starts out by enqueueing all the leaves in a priority queue. The frequency of the leaf is used as the priority. Small frequencies are considered as high priority (i.e. they are served first). In every phase of the iteration, we are done when the priority queue contains a single tree. In that case, we serve it and return it as the result. In case it contains more than one tree, we serve two trees and we enqueue a new tree that combines both trees. The priority of the new tree is the sum of the weights of the two original trees. You can use the following abstractions (that are taken from SICP [AS96]).

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))

(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))

(define (leaf? object)
  (eq? (car object) 'leaf))
(define (symbol-leaf x) (cadr x))
(define (weight-leaf x) (caddr x))
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```

Here is how your procedure should be used.

```
(define freqs (list (make-leaf 'c 10) (make-leaf 'g 4) (make-leaf 'd 8)
                    (make-leaf 'a 40) (make-leaf 'e 8) (make-leaf 'b 20)
                    (make-leaf 'f 6)  (make-leaf 'h 4)))
(define ht (huffman-tree freqs))
```

4.6 Further Reading

The initial material presented in this chapter (stacks and queues) is often ignored by more mathematical books on algorithms and data structures. Apart from the circular queue implementation, the implementation is fairly boring from an algorithmic point of view. We nevertheless consider it a useful exercises to expose students to the difference between a linked implementation and a vectorial implementation of these ADTs. Concerning the heap implementation, one should realize that we have only considered one particular kind of heaps, namely *binary heaps*. More advanced types of heaps such as *binomial heaps* and *fibonacci heaps* are e.g. presented in [CLRS01]. These heaps support more useful operations but are also more complicated to represent and implement.