```
signature QUEUE =
sig
  type α Queue

  val empty   : α Queue
  val isEmpty : α Queue → bool

  val snoc    : α Queue × α → α Queue
  val head    : α Queue → α         (* raises EMPTY if queue is empty *)
  val tail    : α Queue → α Queue (* raises EMPTY if queue is empty *)
end
```

Figure 5.1. Signature for queues.

(Etymological note: snoc is cons spelled backward and means "cons on the right".)

5.2 Queues

We next illustrate the banker's and physicist's methods by analyzing a simple functional implementation of the FIFO queue abstraction, as specified by the signature in Figure 5.1.

The most common implementation of queues in a purely functional setting is as a pair of lists, *f* and *r*, where *f* contains the front elements of the queue in the correct order and *r* contains the rear elements of the queue in reverse order. For example, a queue containing the integers 1...6 might be represented by the lists *f* = [1,2,3] and *r* = [6,5,4]. This representation is described by the following type:

```
type α Queue = α list × α list
```

In this representation, the head of the queue is the first element of *f*, so head and tail return and remove this element, respectively.

```
fun head (x :: f, r) = x
fun tail (x :: f, r) = (f, r)
```

Similarly, the last element of the queue is the first element of *r*, so snoc simply adds a new element to *r*.

```
fun snoc ((f, r), x) = (f, x :: r)
```

Elements are added to *r* and removed from *f*, so they must somehow migrate from one list to the other. This migration is accomplished by reversing *r* and installing the result as the new *f* whenever *f* would otherwise become empty, simultaneously setting the new *r* to [ ]. The goal is to maintain the invariant that *f* is empty only if *r* is also empty (i.e., the entire queue is empty). Note that, if *f* were empty when *r* was not, then the first element of the queue would be

```
structure BatchedQueue : QUEUE =
struct
  type α Queue = α list × α list

  val empty = ([ ], [ ])
  fun isEmpty (f, r) = null f

  fun checkf ([ ], r) = (rev r, [ ])
    | checkf q = q

  fun snoc ((f, r), x) = checkf (f, x :: r)

  fun head ([ ], _) = raise EMPTY
    | head (x :: f, r) = x
  fun tail ([ ], _) = raise EMPTY
    | tail (x :: f, r) = checkf (f, r)
end
```

Figure 5.2. A common implementation of purely functional queues.

the last element of *r*, which would take $O(n)$ time to access. By maintaining this invariant, we guarantee that head can always find the first element in $O(1)$ time.

snoc and tail must now detect those cases that would otherwise result in a violation of the invariant, and change their behavior accordingly.

```
fun snoc (([ ], _), x) = ([x], [ ])
  | snoc ((f, r), x) = (f, x :: r)
fun tail ([x], r) = (rev r, [ ])
  | tail (x :: f, r) = (f, r)
```

Note the use of the wildcard in the first clause of snoc. In this case, the *r* field is irrelevant because we know by the invariant that if *f* is [ ], then so is *r*.

A slightly cleaner way to write these functions is to consolidate into a single function checkf those parts of snoc and tail that are devoted to maintaining the invariant. checkf replaces *f* with rev *r* when *f* is empty, and otherwise does nothing.

```
fun checkf ([ ], r) = (rev r, [ ])
  | checkf q = q

fun snoc ((f, r), x) = checkf (f, x :: r)
fun tail (x :: f, r) = checkf (f, r)
```

The complete code for this implementation is shown in Figure 5.2. snoc and head run in $O(1)$ worst-case time, but tail takes $O(n)$ time in the worst-case. However, we can show that snoc and tail both take $O(1)$ amortized time using either the banker's method or the physicist's method.

Using the banker's method, we maintain a credit invariant that every element in the rear list is associated with a single credit. Every snoc into a non-empty queue takes one actual step and allocates a credit to the new element of the rear list, for an amortized cost of two. Every tail that does not reverse the rear list takes one actual step and neither allocates nor spends any credits, for an amortized cost of one. Finally, every tail that does reverse the rear list takes $m + 1$ actual steps, where m is the length of the rear list, and spends the m credits contained by that list, for an amortized cost of $m + 1 - m = 1$.

Using the physicist's method, we define the potential function Φ to be the length of the rear list. Then every snoc into a non-empty queue takes one actual step and increases the potential by one, for an amortized cost of two. Every tail that does not reverse the rear list takes one actual step and leaves the potential unchanged, for an amortized cost of one. Finally, every tail that does reverse the rear list takes $m + 1$ actual steps and sets the new rear list to [ ], decreasing the potential by m, for an amortized cost of $m + 1 - m = 1$.

In this simple example, the proofs are virtually identical. Even so, the physicist's method is slightly simpler for the following reason. Using the banker's method, we must first choose a credit invariant, and then decide for each function when to allocate or spend credits. The credit invariant provides guidance in this decision, but does not make it automatic. For instance, should snoc allocate one credit and spend none, or allocate two credits and spend one? The net effect is the same, so this freedom is just one more potential source of confusion. On the other hand, using the physicist's method, we have only one decision to make—the choice of the potential function. After that, the analysis is mere calculation, with no more freedom of choice.

Hint to Practitioners: These queues cannot be beat for applications that do not require persistence and for which amortized bounds are acceptable.

Exercise 5.1 (Hoogerwoord [Hoo92]) This design can easily be extended to support the *double-ended queue*, or *deque*, abstraction, which allows reads and writes to both ends of the queue (see Figure 5.3). The invariant is updated to be symmetric in its treatment of f and r: both are required to be non-empty whenever the deque contains two or more elements. When one list becomes empty, we split the other list in half and reverse one of the halves.

(a) Implement this version of deques.

(b) Prove that each operation takes $O(1)$ amortized time using the potential function $\Phi(f, r) = abs(|f| - |r|)$, where abs is the absolute value function.