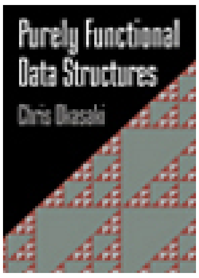


Purely Functional Data Structures

Chris Okasaki

Book DOI: http://dx.doi.org/10.1017/CBO9780511530104

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

5 - Fundamentals of Amortization pp. 39-56

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.006

Cambridge University Press

5
Fundamentals of Amortization

Over the past fifteen years, amortization has become a powerful tool in the design and analysis of data structures. Implementations with good amortized bounds are often simpler and faster than implementations with comparable worst-case bounds. In this chapter, we review the basic techniques of amortization and illustrate these ideas with a simple implementation of FIFO queues and several implementations of heaps.

Unfortunately, the simple view of amortization presented in this chapter breaks in the presence of persistence—these data structures may be extremely inefficient when used persistently. In practice, however, many applications do not require persistence, and for those applications, the implementations presented in this chapter are excellent choices. In the next chapter, we will see how to reconcile the notions of amortization and persistence using lazy evaluation.

5.1 Techniques of Amortized Analysis

The notion of amortization arises from the following observation. Given a sequence of operations, we may wish to know the running time of the entire sequence, but not care about the running time of any individual operation. For instance, given a sequence of n operations, we may wish to bound the total running time of the sequence by $O(n)$ without insisting that every individual operation run in $O(1)$ time. We might be satisfied if a few operations run in $O(\log n)$ or even $O(n)$ time, provided the total cost of the sequence is only $O(n)$. This freedom opens up a wide design space of possible solutions, and often yields new solutions that are simpler and faster than worst-case solutions with equivalent bounds.

To prove an amortized bound, one defines the amortized cost of each operation and then proves that, for any sequence of operations, the total amortized cost of the operations is an upper bound on the total actual cost, i.e.,

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

where a_i is the amortized cost of operation i, t_i is the actual cost of operation i, and m is the total number of operations. Usually, in fact, one proves a slightly stronger result: that at any intermediate stage in a sequence of operations, the accumulated amortized cost is an upper bound on the accumulated actual cost, i.e.,

$$\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_i$$

for any j. The difference between the accumulated amortized costs and the accumulated actual costs is called the *accumulated savings*. Thus, the accumulated amortized costs are an upper bound on the accumulated actual costs whenever the accumulated savings is non-negative.

Amortization allows for occasional operations to have actual costs that exceed their amortized costs. Such operations are called *expensive*. Operations whose actual costs are less than their amortized costs are called *cheap*. Expensive operations decrease the accumulated savings and cheap operations increase it. The key to proving amortized bounds is to show that expensive operations occur only when the accumulated savings are sufficient to cover the remaining cost.

Tarjan [Tar85] describes two techniques for analyzing amortized data structures: the *banker's method* and the *physicist's method*. In the banker's method, the accumulated savings are represented as *credits* that are associated with individual locations in the data structure. These credits are used to pay for future accesses to these locations. The amortized cost of any operation is defined to be the actual cost of the operation plus the credits allocated by the operation minus the credits spent by the operation, i.e.,

$$a_i = t_i + c_i - \overline{c}_i$$

where c_i is the number of credits allocated by operation i and $\overline{c}_i$ is the number of credits spent by operation i. Every credit must be allocated before it is spent, and no credit may be spent more than once. Therefore, $\sum c_i \geq \sum \overline{c}_i$, which in turn guarantees that $\sum a_i \geq \sum t_i$, as desired. Proofs using the banker's method typically define a *credit invariant* that regulates the distribution of credits in such a way that, whenever an expensive operation might occur, sufficient credits have been allocated in the right locations to cover its cost.

In the physicist's method, one describes a function Φ that maps each object d to a real number called the *potential* of d. The function Φ is typically chosen so that the potential is initially zero and is always non-negative. Then, the potential represents a lower bound on the accumulated savings.

Let d_i be the output of operation i and the input of operation $i+1$. Then, the amortized cost of operation i is defined to be the actual cost plus the change in potential between d_{i-1} and d_i, i.e.,

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

The accumulated actual costs of the sequence of operations are

$$\begin{array}{rcl} \sum_{i=1}^{j} t_i & = & \sum_{i=1}^{j}(a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ & = & \sum_{i=1}^{j} a_i + \sum_{i=1}^{j}(\Phi(d_{i-1}) - \Phi(d_i)) \\ & = & \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j) \end{array}$$

Sums such as $\sum(\Phi(d_{i-1}) - \Phi(d_i))$, where alternating positive and negative terms cancel each other out, are called *telescoping series*. Provided Φ is chosen in such a way that $\Phi(d_0)$ is zero and $\Phi(d_j)$ is non-negative, then $\Phi(d_j) \geq \Phi(d_0)$ and $\sum a_i \geq \sum t_i$, so the accumulated amortized costs are an upper bound on the accumulated actual costs, as desired.

Remark This is a somewhat simplified view of the physicist's method. In real analyses, one often encounters situations that are difficult to fit into the framework as described. For example, what about functions that take or return more than one object? However, this simplified view suffices to illustrate the relevant issues. $\diamond$

Clearly, the two methods are very similar. We can convert the banker's method to the physicist's method by ignoring locations and taking the potential to be the total number of credits in the object, as indicated by the credit invariant. Similarly, we can convert the physicist's method to the banker's method by converting potential to credits, and placing all credits on the root. It is perhaps surprising that the knowledge of locations in the banker's method offers no extra power, but the two methods are in fact equivalent [Tar85, Sch92]. The physicist's method is usually simpler, but it is occasionally convenient to take locations into account.

Note that both credits and potential are analysis tools only; neither actually appears in the program text (except maybe in comments).

```
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.

```
signature DEQUE =
sig
  type α Queue

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

  (* insert, inspect, and remove the front element *)
  val cons    : α × α Queue → 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 *)

  (* insert, inspect, and remove the rear element *)
  val snoc    : α Queue × α → α Queue
  val last    : α Queue → α          (* raises EMPTY if queue is empty *)
  val init    : α Queue → α Queue    (* raises EMPTY if queue is empty *)
end
```

Figure 5.3. Signature for double-ended queues.

5.3 Binomial Heaps

In Section 3.2, we showed that insert on binomial heaps runs in $O(\log n)$ worst-case time. Here, we prove that insert actually runs in $O(1)$ amortized time.

We use the physicist's method. Define the potential of a binomial heap to be the number of trees in the heap. Recall that this is equivalent to the number of ones in the binary representation of n, the number of elements in the heap. Now, a call to insert takes $k + 1$ steps where k is the number of calls to link. If there were initially t trees in the heap, then after the insertion, there are $t-k+1$ trees. Thus, the change in potential is $(t-k+1)-t = 1-k$ and the amortized cost of the insertion is $(k+1)+(1-k) = 2$.

Exercise 5.2 Repeat this proof using the banker's method. ◇

To be complete, we must also show that the amortized costs of merge and deleteMin are still $O(\log n)$. deleteMin poses no particular problem, but merge requires a minor extension to the physicist's method. Previously, we defined the amortized cost of an operation to be

$$a = t + \Phi(d_{out}) - \Phi(d_{in})$$

where d_{in} is the input to the operation and d_{out} is the output. However, if an operation takes or returns more than one object, then we generalize this rule to

$$a = t + \sum_{d \in Out} \Phi(d) - \sum_{d \in In} \Phi(d)$$

where *In* is the set of inputs and *Out* is the set of outputs. For the purposes of this rule, we consider only inputs and outputs of the type(s) being analyzed.

Exercise 5.3 Prove that the amortized costs of merge and deleteMin are still $O(\log n)$.

5.4 Splay Heaps

Splay trees [ST85] are perhaps the most famous and successful of all amortized data structures. Splay trees are a close relative of balanced binary search trees, but they maintain no explicit balance information. Instead, every operation blindly restructures the tree using some simple transformations that tend to increase balance. Although any individual operation can take as much as $O(n)$ time, we will show that every operation runs in $O(\log n)$ amortized time.

A major difference between splay trees and balanced binary search trees such as the red-black trees of Section 3.3 is that splay trees are restructured even during queries (e.g., member) instead of only during updates (e.g., insert). This property makes it awkward to use splay trees to implement abstractions such as sets or finite maps in a purely functional setting, because the query would have to return the new tree along with the answer.† For some abstractions, however, the queries are limited enough to avoid these problems. A good example is the heap abstraction, where the only interesting query is findMin. In fact, as we will see, splay trees make an excellent implementation of heaps.

The representation of splay trees is identical to that of unbalanced binary search trees.

```
datatype Tree = E | T of Tree × Elem.T × Tree
```

Unlike the unbalanced binary search trees of Section 2.2, however, we allow duplicate elements within a single tree. This is not a fundamental difference between splay trees and unbalanced binary search trees; rather, it reflects a difference between the set abstraction and the heap abstraction.

Consider the following strategy for implementing insert: partition the existing tree into two subtrees, one containing all the elements smaller than or equal to the new element and one containing all the elements bigger than the new element, and then construct a new node from the new element and the two subtrees. Unlike insertion into ordinary binary search trees, this procedure adds the new element at the root of the tree rather than at the leaves. The code for insert is simply

† In a language like Standard ML, it is possible to store the root of each splay tree in a ref cell, and then update the ref cell after each query, but this is not purely functional.

```
fun insert (x, t) = T (smaller (x, t), x, bigger (x, t))
```

where smaller and bigger extract the appropriate subtrees. In analogy to the partitioning phase of quicksort, we call the new element the *pivot*.

We could implement bigger naively as

```
fun bigger (pivot, E) = E
  | bigger (pivot, T (a, x, b)) =
      if x ≤ pivot then bigger (pivot, b)
      else T (bigger (pivot, a), x, b)
```

but this makes no attempt to restructure the tree to make it more balanced. Instead, we use a very simple restructuring heuristic: every time we follow two left branches in a row, we rotate those two nodes.

```
fun bigger (pivot, E) = E
  | bigger (pivot, T (a, x, b)) =
      if x ≤ pivot then bigger (pivot, b)
      else case a of
             E ⇒ T (E, x, b)
           | T (a₁, y, a₂) ⇒
               if y ≤ pivot then T (bigger (pivot, a₂), x, b)
               else T (bigger (pivot, a₁), y, T (a₂, x, b))
```

Figure 5.4 illustrates the effect of bigger on a very unbalanced tree. Although still not balanced in the usual sense, the new tree is much more balanced than the original tree; the depth of every node has been reduced by about half, from d to $\lfloor d/2 \rfloor$ or $\lfloor d/2 \rfloor + 1$. Of course, we cannot always halve the depth of every node in the tree, but we can always halve the depth of every node along the search path. In fact, this is the guiding principle of splay trees: search paths should be restructured to reduce the depth of every node in the path by about half.

Exercise 5.4 Implement smaller. Keep in mind that smaller should retain equal elements (but do not make a separate test for equality!). ◇

Notice that smaller and bigger both traverse the same search path. Rather than duplicating this traversal, we can combine smaller and bigger into a single function called partition that returns the results of both as a pair. This function is straightforward, but somewhat tedious.

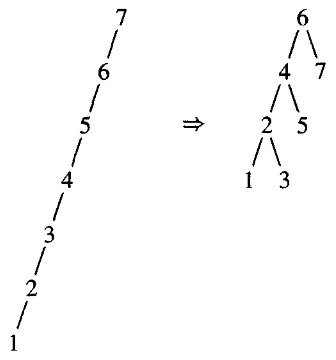


Figure 5.4. Calling bigger with a pivot element of 0.

```
fun partition (pivot, E) = (E, E)
  | partition (pivot, t as T (a, x, b)) =
    if x ≤ pivot then
      case b of
        E ⇒ (t, E)
      | T (b1, y, b2) ⇒
          if y ≤ pivot then
            let val (small, big) = partition (pivot, b2)
            in (T (T (a, x, b1), y, small), big) end
          else
            let val (small, big) = partition (pivot, b1)
            in (T (a, x, small), T (big, y, b2)) end
    else
      case a of
        E ⇒ (E, t)
      | T (a1, y, a2) ⇒
          if y ≤ pivot then
            let val (small, big) = partition (pivot, a2)
            in (T (a1, y, small), T (big, x, b)) end
          else
            let val (small, big) = partition (pivot, a1)
            in (small, T (big, y, T (a2, x, b))) end
```

Remark This function is not exactly equivalent to smaller and bigger because of phase differences: partition always processes nodes in pairs whereas smaller and bigger sometimes process only a single node. Thus, smaller and bigger sometimes rotate different pairs of nodes than partition. However, these differences are inconsequential. ◇

Next, we consider findMin and deleteMin. The minimum element in a splay tree is stored in the leftmost T node. Finding this node is trivial.

```
fun findMin (T (E, x, b)) = x
  | findMin (T (a, x, b)) = findMin a
```

deleteMin should discard the minimum node, and at the same time, restructure the tree in the same style as bigger. Since we always take the left branch, there is no need for comparisons.

```
fun deleteMin (T (E, x, b)) = b
  | deleteMin (T (T (E, x, b), y, c)) = T (b, y, c)
  | deleteMin (T (T (a, x, b), y, c)) = T (deleteMin a, x, T (b, y, c))
```

Figure 5.5 summarizes this implementation of splay trees. For completeness, we have included the merge function on splay trees even though it is rather inefficient, taking up to $O(n)$ time for many inputs.

Next, we show that insert runs in $O(\log n)$ time. Let $\#t$ denote the size of t plus one and note that if $t = \mathsf{T}(a, x, b)$ then $\#t = \#a + \#b$. Define the potential $\phi(t)$ of an individual node to be $\log(\#t)$ and the potential $\Phi(t)$ of an entire tree be the sum of potentials of all the individual nodes in the tree. We will need the following elementary fact about logarithms:

Lemma 5.1 *For all positive x, y, z such that $y + z \leq x$,*

$$1 + \log y + \log z < 2 \log x$$

Proof Without loss of generality, assume that $y \leq z$. Then $y \leq x/2$ and $z < x$, so $1 + \log y \leq \log x$ and $\log z < \log x$. □

Let $\mathcal{T}(t)$ denote the actual cost of calling partition on tree t, defined as the total number of recursive calls to partition. Let $\mathcal{A}(t)$ denote the amortized cost of calling partition on t, defined as

$$\mathcal{A}(T) = \mathcal{T}(t) + \Phi(a) + \Phi(b) - \Phi(t)$$

where a and b are the subtrees returned by *partition*.

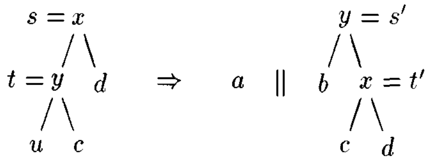
Theorem 5.2 $\mathcal{A}(t) \leq 1 + 2\phi(t) = 1 + 2\log(\sharp t)$.

Proof There are two interesting cases, called the zig-zig case and the zig-zag case, depending on whether a particular call to partition follows two left branches (symmetrically, two right branches) or a left branch followed by a right branch (symmetrically, a right branch followed by a left branch).

For the zig-zig case, assume that the original tree and the resulting trees have the shapes

```
functor SplayHeap (Element : ORDERED) : HEAP =
struct
   structure Elem = Element

   datatype Heap = E | T of Heap × Elem.T × Heap

   val empty = E
   fun isEmpty E = true | isEmpty _ = false

   fun partition (pivot, E) = (E, E)
     | partition (pivot, t as T (a, x, b)) =
         if Elem.leq (x, pivot) then
           case b of
             E ⇒ (t, E)
           | T (b1, y, b2) ⇒
               if Elem.leq (y, pivot) then
                 let val (small, big) = partition (pivot, b2)
                 in (T (T (a, x, b1), y, small), big) end
               else
                 let val (small, big) = partition (pivot, b1)
                 in (T (a, x, small), T (big, y, b2)) end
         else
           case a of
             E ⇒ (E, t)
           | T (a1, y, a2) ⇒
               if Elem.leq (y, pivot) then
                 let val (small, big) = partition (pivot, a2)
                 in (T (a1, y, small), T (big, x, b)) end
               else
                 let val (small, big) = partition (pivot, a1)
                 in (small, T (big, y, T (a2, x, b))) end

   fun insert (x, t) = let val (a, b) = partition (x, t) in T (a, x, b) end
   fun merge (E, t) = t
     | merge (T (a, x, b), t) =
         let val (ta, tb) = partition (x, t)
         in T (merge (ta, a), x, merge (tb, b)) end

   fun findMin E = raise EMPTY
     | findMin (T (E, x, b)) = x
     | findMin (T (a, x, b)) = findMin a
   fun deleteMin E = raise EMPTY
     | deleteMin (T (E, x, b)) = b
     | deleteMin (T (T (E, x, b), y, c)) = T (b, y, c)
     | deleteMin (T (T (a, x, b), y, c)) = T (deleteMin a, x, T (b, y, c))
end
```

Figure 5.5. Implementation of heaps using splay trees.



where a and b are the results of partition (*pivot*, u). Then,

$$
\begin{array}{ll}
 & \mathcal{A}(s) \\
= & \qquad \{ \text{ definition of } \mathcal{A} \; \} \\
 & \mathcal{T}(s) + \Phi(a) + \Phi(s') - \Phi(s) \\
= & \qquad \{ \; \mathcal{T}(s) = 1 + \mathcal{T}(u) \; \} \\
 & 1 + \mathcal{T}(u) + \Phi(a) + \Phi(s') - \Phi(s) \\
= & \qquad \{ \; \mathcal{T}(u) = \mathcal{A}(u) - \Phi(a) - \Phi(b) + \Phi(u) \; \} \\
 & 1 + \mathcal{A}(u) - \Phi(a) - \Phi(b) + \Phi(u) + \Phi(a) + \Phi(s') - \Phi(s) \\
= & \qquad \{ \text{ expand } \Phi(s') \text{ and } \Phi(s) \text{ and simplify } \} \\
 & 1 + \mathcal{A}(u) + \phi(s') + \phi(t') - \phi(s) - \phi(t) \\
\leq & \qquad \{ \text{ inductive hypothesis: } \mathcal{A}(u) \leq 1 + 2\phi(u) \; \} \\
 & 2 + 2\phi(u) + \phi(s') + \phi(t') - \phi(s) - \phi(t) \\
< & \qquad \{ \; \phi(u) < \phi(t) \text{ and } \phi(s') \leq \phi(s) \; \} \\
 & 2 + \phi(u) + \phi(t') \\
< & \qquad \{ \; \sharp u + \sharp t' < \sharp s \text{ and Lemma 5.1 } \} \\
 & 1 + 2\phi(s)
\end{array}
$$

The proof of the zig-zag case is left as an exercise. □

Exercise 5.5 Prove the zig-zag case. ◇

The additional cost of insert over that of partition is one actual step plus the change in potential between the two subtrees returned by partition and the final tree. This change in potential is simply ϕ of the new root. Since the amortized cost of partition is bounded by $1 + 2\log(\sharp t)$, the amortized cost of insert is bounded by $2 + 2\log(\sharp t) + \log(\sharp t + 1) \approx 2 + 3\log(\sharp t)$.

Exercise 5.6 Prove that deleteMin also runs in $O(\log n)$ time. ◇

Now, what about findMin? For a very unbalanced tree, findMin might take up to $O(n)$ time. But since findMin does not do any restructuring and therefore causes no change in potential, there is no way to amortize this cost! However, since findMin takes time proportional to deleteMin, if we double the charged cost of deleteMin then we can essentially run findMin for free once per call to deleteMin. This suffices for those applications that always call findMin and deleteMin together. However, some applications may call findMin several times per call to deleteMin. For those applications, we would not use the SplayHeap functor directly, but rather would use the SplayHeap functor in conjunction with the ExplicitMin functor of Exercise 3.7. Recall that the purpose of the ExplicitMin functor was to make findMin run in $O(1)$ time. The insert and deleteMin functions would still run in $O(\log n)$ amortized time.

Hint to Practitioners: Splay trees, perhaps in combination with the ExplicitMin functor, are the fastest known implementation of heaps for most applications that do not depend on persistence and that do not call the merge function.

A particularly pleasant feature of splay trees is that they naturally adapt to any order that happens to be present in the input data. For example, using splay heaps to sort an already sorted list takes only $O(n)$ time rather than $O(n \log n)$ time [MEP96]. Leftist heaps also share this property, but only for decreasing sequences. Splay heaps excel on both increasing and decreasing sequences, as well as on sequences that are only partially sorted.

Exercise 5.7 Write a sorting function that inserts elements into a splay tree and then performs an inorder traversal of the tree, dumping the elements into a list. Show that this function takes only $O(n)$ time on an already sorted list.

5.5 Pairing Heaps

Pairing heaps [FSST86] are one of those data structures that drive theoreticians crazy. On the one hand, pairing heaps are simple to implement and perform extremely well in practice. On the other hand, they have resisted analysis for over ten years!

Pairing heaps are heap-ordered multiway trees, as defined by the following datatype:

```
datatype Heap = E | T of Elem.T × Heap list
```

We allow only well-formed trees, in which E never occurs in the child list of a T node.

Since these trees are heap-ordered, the findMin function is trivial.

```
fun findMin (T (x, hs)) = x
```

The merge and insert functions are not much harder. merge makes the tree with the larger root the leftmost child of the tree with the smaller root. insert first creates a new singleton tree and then immediately calls merge.

```
fun merge (h, E) = h
  | merge (E, h) = h
  | merge (h1 as T (x, hs1), h2 as T (y, hs2)) =
      if Elem.leq (x, y) then T (x, h2 :: hs1) else T (y, h1 :: hs2)
fun insert (x, h) = merge (T (x, []), h)
```

Pairing heaps get their name from the deleteMin operation. deleteMin discards the root and then merges the children in two passes. The first pass merges children in pairs from left to right (i.e., the first child with the second, the third with the fourth, and so on). The second pass merges the resulting trees from right to left. These two passes can be coded concisely as

```
fun mergePairs [] = E
  | mergePairs [h] = h
  | mergePairs (h1 :: h2 :: hs) = merge (merge (h1, h2), mergePairs hs)
```

Then, deleteMin is simply

```
fun deleteMin (T (x, hs)) = mergePairs hs
```

The complete implementation appears in Figure 5.6.

Now, it is easy to see that findMin, insert, and merge all run in $O(1)$ worst-case time. However, deleteMin can take up to $O(n)$ time in the worst case. By drawing an analogy to splay trees (see Exercise 5.8), we can show that insert, merge, and deleteMin all run in $O(\log n)$ amortized time. It has been conjectured that insert and merge actually run in $O(1)$ amortized time [FSST86], but no one has yet been able to prove or disprove this claim.

Hint to Practitioners: Pairing heaps are almost as fast in practice as splay heaps for applications that do not use the merge function, and much faster for applications that do. Like splay heaps, however, they should be used only for applications that do not take advantage of persistence.

Exercise 5.8 Binary trees are often more convenient than multiway trees. Fortunately, there is an easy way to represent any multiway tree as a binary tree. Simply convert every multiway node into a binary node whose left child represents the leftmost child of the multiway node and whose right child represents the sibling to the immediate right of the multiway node. If either the leftmost child or the right sibling of the multiway node is missing, then the corresponding field in the binary node is empty. (Note that this implies that the right child of the root is always empty in the binary representation.) Applying this transformation to pairing heaps yields half-ordered binary trees in which the element at each node is no greater than any element in its left subtree.

```
functor PairingHeap (Element : ORDERED) : HEAP =
struct
  structure Elem = Element

  datatype Heap = E | T of Elem.T × Heap list

  val empty = E
  fun isEmpty E = true | isEmpty _ = false

  fun merge (h, E) = h
    | merge (E, h) = h
    | merge (h1 as T (x, hs1), h2 as T (y, hs2)) =
        if Elem.leq (x, y) then T (x, h2 :: hs1) else T (y, h1 :: hs2)
  fun insert (x, h) = merge (T (x, [ ]), h)

  fun mergePairs [ ] = E
    | mergePairs [h] = h
    | mergePairs (h1 :: h2 :: hs) = merge (merge (h1, h2), mergePairs hs)

  fun findMin E = raise EMPTY
    | findMin (T (x, hs)) = x
  fun deleteMin E = raise EMPTY
    | deleteMin (T (x, hs)) = mergePairs hs
end
```

Figure 5.6. Pairing heaps.

(a) Write a function toBinary that converts pairing heaps from the existing representation into the type

```
datatype BinTree = E' | T' of Elem.T × BinTree × BinTree
```

(b) Reimplement pairing heaps using this new representation.

(c) Adapt the analysis of splay trees to prove that deleteMin and merge run in $O(\log n)$ amortized time for this new representation (and hence for the old representation as well). Use the same potential function as for splay trees.

5.6 The Bad News

As we have seen, amortized data structures are often tremendously effective in practice. Unfortunately, the analyses in this chapter implicitly assume that the data structures in question are used ephemerally (i.e., in a single-threaded fashion). What happens if we try to use one of these data structures persistently?

Consider the queues of Section 5.2. Let q be the result of inserting n elements into an initially empty queue, so that the front list of q contains a single element and the rear list contains $n - 1$ elements. Now, suppose we use q persistently by taking its tail n times. Each call of tail q takes n actual steps. The total actual cost of this sequence of operations, including the time to build q, is $n^2 + n$. If the operations truly took $O(1)$ amortized time each, then the total actual cost would be only $O(n)$. Clearly, using these queues persistently invalidates the $O(1)$ amortized time bounds proved in Section 5.2. Where do these proofs go wrong?

In both cases, a fundamental requirement of the analysis is violated by persistent data structures. The banker's method requires that no credit be spent more than once, while the physicist's method requires that the output of one operation be the input of the next operation (or, more generally, that no output be used as input more than once). Now, consider the second call to tail q in the example above. The first call to tail q spends all the credits on the rear list of q, leaving none to pay for the second and subsequent calls, so the banker's method breaks. And the second call to tail q reuses q rather than the output of the first call, so the physicist's method breaks.

Both these failures reflect the inherent weakness of any accounting system based on accumulated savings—that savings can only be spent once. The traditional methods of amortization operate by accumulating savings (as either credits or potential) for future use. This works well in an ephemeral setting, where every operation has only a single logical future. But with persistence, an operation might have multiple logical futures, each competing to spend the same savings.

In the next chapter, we will clarify what we mean by the "logical future" of an operation, and show how to reconcile amortization and persistence using lazy evaluation.

Exercise 5.9 Give examples of sequences of operations for which binomial heaps, splay heaps, and pairing heaps take much longer than indicated by their amortized bounds.

5.7 Chapter Notes

The techniques of amortization presented in this chapter were developed by Sleator and Tarjan [ST85, ST86b] and popularized by Tarjan [Tar85]. Schoenmakers [Sch92] has shown how to systematically derive amortized bounds in a functional setting without persistence.

Gries [Gri81, pages 250–251] and Hood and Melville [HM81] first proposed the queues in Section 5.2. Burton [Bur82] proposed a similar implementation, but without the restriction that the front list be non-empty whenever the queue is non-empty. Burton combines head and tail into a single function, and so does not require this restriction to support head efficiently.

Several empirical studies have shown that splay heaps [Jon86] and pairing heaps [MS91, Lia92] are among the fastest of all heap implementations. Stasko and Vitter [SV87] have confirmed the conjectured $O(1)$ amortized bound on insert for a variant of pairing heaps.