## Contents

**Preface** vii

1 **Fun with binary heap trees** 1
   Chris Okasaki
   1.1 Binary heap trees 1
   1.2 Maxiphobic heaps 4
   1.3 Persistence 6
   1.4 Round-robin heaps 7
   1.5 Analysis of skew heaps 10
   1.6 Lazy evaluation 12
   1.7 Analysis of lazy skew heaps 15
   1.8 Chapter notes 16

2 **Specification-based testing with QuickCheck** 17
   Koen Claessen and John Hughes
   2.1 Introduction 17
   2.2 Properties in QuickCheck 18
   2.3 Example: Developing an abstract data type of queues 20
   2.4 Quantifying over subsets of types 25
   2.5 Test coverage 30
   2.6 A larger case study 32
   2.7 Conclusions 39
   2.8 Acknowledgements 39

3 **Origami programming** 41
   Jeremy Gibbons
   3.1 Introduction 41
   3.2 Origami with lists: sorting 42
   3.3 Origami by numbers: loops 49
   3.4 Origami with trees: traversals 52
   3.5 Other sorts of origami 56
   3.6 Chapter notes 60
4 Describing and interpreting music in Haskell
Paul Hudak
4.1 Introduction 61
4.2 Representing music 61
4.3 Operations on musical structures 67
4.4 The meaning of music 70
4.5 Discussion 78

5 Mechanising fusion
Ganesh Sittampalam and Oege de Moor
5.1 Active source 79
5.2 Fusion, rewriting and matching 85
5.3 The MAG system 89
5.4 A substantial example 98
5.5 Difficulties 101
5.6 Chapter notes 103

6 How to write a financial contract
Simon Peyton Jones and Jean-Marc Eber
6.1 Introduction 105
6.2 Getting started 106
6.3 Building contracts 108
6.4 Valuation 116
6.5 Implementation 123
6.6 Operational semantics 127
6.7 Chapter notes 128

7 Functional images
Conal Elliott
7.1 Introduction 131
7.2 What is an image? 132
7.3 Colours 135
7.4 Pointwise lifting 137
7.5 Spatial transforms 139
7.6 Animation 141
7.7 Region algebra 142
7.8 Some polar transforms 144
7.9 Strange hybrids 147
7.10 Bitmaps 148
7.11 Chapter notes 150

8 Functional hardware description in Lava
Koen Claessen, Mary Sheeran and Satnam Singh
8.1 Introduction 151
8.2 Circuits in Lava 152
8.3 Recursion over lists 153
8.4 Connection patterns 155
8.5 Properties of circuits 157
8.6 Sequential circuits 160
8.7 Describing butterfly circuits 162
8.8 Batcher’s mergers and sorters 166
8.9 Generating FPGA configurations 170
8.10 Chapter notes 175

9 Combinators for logic programming 177
Michael Spivey and Silvija Seres
9.1 Introduction 177
9.2 Lists of successes 178
9.3 Monads for searching 179
9.4 Filtering with conditions 182
9.5 Breadth-first search 184
9.6 Lifting programs to the monad level 187
9.7 Terms, substitutions and predicates 188
9.8 Combinators for logic programs 191
9.9 Recursive programs 193

10 Arrows and computation 201
Ross Paterson
10.1 Notions of computation 201
10.2 Special cases 208
10.3 Arrow notation 213
10.4 Examples 216
10.5 Chapter notes 222

11 A prettier printer 223
Philip Wadler
11.1 Introduction 223
11.2 A simple pretty printer 224
11.3 A pretty printer with alternative layouts 228
11.4 Improving efficiency 233
11.5 Examples 236
11.6 Chapter notes 238
11.7 Code 240

12 Fun with phantom types 245
Ralf Hinze
12.1 Introducing phantom types 245
12.2 Generic functions 248
12.3 Dynamic values 250
12.4 Generic traversals and queries 252
12.5 Normalisation by evaluation 255
12.6 Functional unparsing 257
The world of data structures takes on a new fascination when viewed through the lens of functional programming. Two issues in particular set functional data structures apart from typical imperative data structures: *persistence* and *laziness*. In this chapter, we will explore these and other issues in the context of priority queues implemented as binary heap trees.

### 1.1 Binary heap trees

*The world of data structures takes on a new fascination when viewed through the lens of functional programming. Two issues in particular set functional data structures apart from typical imperative data structures: *persistence* and *laziness*. In this chapter, we will explore these and other issues in the context of priority queues implemented as binary heap trees.*

**Binary heap trees** (IFPH §6.3) are close cousins of binary search trees. Like binary search trees, binary heap trees are binary trees with ordered labels. They can be represented by the datatype

\[
\text{data } (\text{Ord } \alpha) \Rightarrow \text{Tree } \alpha = \text{Null} \mid \text{Fork } \alpha (\text{Tree } \alpha) (\text{Tree } \alpha)
\]

The major difference between search trees and heap trees is the ordering invariant on labels. In a heap tree, the sequence of labels along any path from the root to a leaf must be non-decreasing. In other words, the label of every child node must be at least as big as its parent’s. One immediate consequence of this invariant is that the smallest label in a tree is always at the root. Figure 1.1 shows several examples of binary heap trees.

Notice that heap trees place no restrictions on sibling nodes. We know that the labels of both are greater than or equal to the label of the parent, but we have no idea which of the siblings is smaller. The same is true for cousin nodes, or any other pair of nodes where neither is a descendant of the other.

Notice also that heap trees are *not* required to be balanced. In fact, the more unbalanced a tree is, the faster many heap-tree algorithms run! (See Exercise 1.3.)
Priority queues

Binary heap trees are often used to represent priority queues supporting at least the following set of operations:\(^1\)

- **isEmpty** :: Tree α → Bool — is the heap empty?
- **minElem** :: Tree α → α — find the smallest element
- **deleteMin** :: Tree α → Tree α — delete the smallest element
- **insert** :: α → Tree α → Tree α — add a single element
- **merge** :: Tree α → Tree α → Tree α — combine two heaps

As is traditional, although admittedly somewhat confusing, smaller labels mean higher priorities, so the minimum element is the element with the highest priority.

The **isEmpty** and **minElem** operations are trivial to implement.

\[
\begin{align*}
\text{isEmpty } \text{Null} & = \text{True} \\
\text{isEmpty } (\text{Fork } x \ a \ b) & = \text{False} \\
\text{minElem } (\text{Fork } x \ a \ b) & = x
\end{align*}
\]

Both clearly run in \(O(1)\) time.

The **deleteMin** and **insert** operations are also trivial to implement, given an implementation of **merge**:

\[
\begin{align*}
\text{deleteMin } (\text{Fork } x \ a \ b) & = \text{merge } a \ b \\
\text{insert } x \ a & = \text{merge } (\text{Fork } \text{Null } \text{Null}) \ a
\end{align*}
\]

Assuming **merge** runs in \(O(\log n)\) time, so do **deleteMin** and **insert**. All that remains is to come up with an implementation of **merge** that runs in \(O(\log n)\) time.

\(^1\)Some implementations of priority queues omit **merge**, but we will not discuss such implementations further.
Merge

There are actually many different ways to implement merge for binary heap trees, each leading to a different kind of heap. Examples include leftist heaps [29, 80], weight-biased leftist heaps [22] (see also IFPH §8.4), and skew heaps [120].

Portions of the merge operation are straightforward. Merging any heap with an empty heap yields the original heap. Combining two non-empty heaps with root labels \( x \) and \( y \), respectively, produces a tree whose root label is the smaller of \( x \) and \( y \). These facts are enough to write part of the merge function.

\[
\begin{align*}
\text{merge } a \text{ Null } &= a \\
\text{merge } \text{Null } b &= b \\
\text{merge } a \ b \\
&\mid \text{minElem } a \leq \text{minElem } b = \text{join } a \ b \\
&\mid \text{otherwise} = \text{join } b \ a \\
\text{join } (\text{Fork } x \ a \ b) \ c &= \text{Fork } x ? ?
\end{align*}
\]

The auxiliary function join is called once we’ve decided which root label is smaller. That label becomes the root label of the result, but what should the subtrees of this new node be? We have three trees in hand—\( a \), \( b \), and \( c \)—but only two slots! The easiest way to reduce three trees to two is to merge two of them. Unfortunately, there are at least six ways to do this, depending on which trees are merged and how the resulting trees are arranged:

\[
\begin{align*}
\text{Fork } x \ a \ (\text{merge } b \ c) \\
\text{Fork } x \ b \ (\text{merge } a \ c) \\
\text{Fork } x \ c \ (\text{merge } a \ b) \\
\text{Fork } x \ (\text{merge } b \ c) \ a \\
\text{Fork } x \ (\text{merge } a \ c) \ b \\
\text{Fork } x \ (\text{merge } a \ b) \ c
\end{align*}
\]

None of these has an obvious advantage over the others, so which should we choose? If we think back to binary search trees, most techniques for balancing require some extra information in each node, such as a count or a colour. Of course, we are not trying to balance the heap trees, but perhaps extra information in each node would allow us to decide intelligently between the six possibilities above.

**Exercise 1.1** Two of the six possible strategies for join would, if followed exclusively, lead to all trees being linear in shape. Which two? □
1.2 Maxiphobic heaps

Most ravens can count to about four. If two hunters enter a hunting blind and one comes out a short time later, most birds will be fooled into thinking that the blind is now empty. But not ravens! To fool a raven, you need to send roughly six hunters into the blind and have five come out.

Our goal is to implement \texttt{merge} so that it runs in \(O(\log n)\) time. What we have so far for \texttt{merge} involves a little pattern matching, a comparison, a few trivial function calls, a recursive call to \texttt{merge}, and a constructor. Assuming comparisons take \(O(1)\) time, the whole function takes time proportional to the depth of recursion. Therefore, we want a scheme that will keep the recursion as shallow as possible.

Intuitively, we would expect \texttt{merge} to take longer for bigger trees than for smaller trees. Therefore, given a choice, we should choose to merge smaller trees rather than bigger trees. We have no control over external calls to \texttt{merge} (that is, calls by the user), nor do we have any flexibility in the internal calls to \texttt{merge} by \texttt{deleteMin} and \texttt{insert}. However, we do have a choice in the \texttt{join} function. We need to merge two of the three trees \(a, b,\) and \(c\), but we can choose which trees to merge however we wish. Following the above heuristic, we choose to merge the two smallest trees, leaving the biggest tree untouched. We dub such heaps \texttt{maxiphobic} (‘biggest avoiding’).

To implement maxiphobic heaps, we need to extend our tree datatype with a count field that keeps track of the size of each tree.

\[
\text{data } (\text{Ord } \alpha) \Rightarrow \text{Tree } \alpha = \text{Null} | \text{Fork } \text{Int} \alpha (\text{Tree } \alpha) (\text{Tree } \alpha)
\]

The original operations are nearly unaffected by this change. We add the extra count field to the \texttt{Fork} patterns in the \texttt{isEmpty}, \texttt{minElem}, and \texttt{deleteMin} operations, and an initial size of 1 to the newly created singleton node in the \texttt{insert} operation. The \texttt{merge} operation is unchanged.

\[
\begin{align*}
\text{isEmpty Null} &= \text{True} \\
\text{isEmpty (Fork } n \times a b) &= \text{False} \\
\text{minElem (Fork } n \times a b) &= x \\
\text{deleteMin (Fork } n \times a b) &= \text{merge } a b \\
\text{insert } x a &= \text{merge } (\text{Fork } 1 x \times \text{Null Null}) a \\
\text{merge } a \text{ Null} &= a \\
\text{merge } \text{Null } b &= b \\
\text{merge } a b &
\begin{cases}
| \text{minElem } a \leq \text{minElem } b &= \text{join } a b \\
| \text{otherwise} &= \text{join } b a
\end{cases}
\end{align*}
\]

The main change is in the \texttt{join} function, which must now decide which of the
three trees is biggest.

\[
\text{join (Fork n x a b) c} = \text{Fork (n + size c) x aa (merge bb cc)}
\]

where \((aa, bb, cc) = \text{orderBySize} a b c\)

\[
\text{orderBySize} a b c
\]

| size a = biggest = (a, b, c)  \\
| size b = biggest = (b, a, c)  \\
| size c = biggest = (c, a, b)  \\

\[
\text{where biggest} = \text{size a 'max' size b 'max' size c}
\]

\[
\text{size Null} = 0
\]

\[
\text{size (Fork n x a b)} = n
\]

Figure 1.2 illustrates a sequence of insertions using these rules for \text{join}.

Does this implementation run in \(O(\log n)\) time? Yes. Suppose that the combined size of the original arguments to \text{merge} was \(n\). Then the combined size of \(a, b,\) and \(c\) in \text{join} is \(n - 1\). The largest of these has size at least \(\lceil \frac{n-1}{3} \rceil\). Therefore, the combined size of the two trees passed to the recursive merge is at most \(\lceil \frac{2n-2}{3} \rceil\), or roughly two-thirds \(n\). Therefore, there can be at most \(O(\log_{1.5} n) = O(\log n)\) recursive calls.

\textbf{Exercise 1.2} Draw the trees resulting from the following operations:

1. Insert the numbers 1–7 into an initially empty heap in reverse order.
2. Insert the numbers 1, 7, 2, 6, 3, 5, 4 into an initially empty heap.
3. Delete the minimum element from the last tree of Figure 1.2.

\textbf{Exercise 1.3} Explain why an unbalanced maxiphobic heap (in the extreme, a completely linear heap) is preferable to a balanced one.
1.3 Persistence

*The Sankofa bird is a Ghanaian symbol depicted as a bird looking backwards. It symbolises the idea that one must return to the past to build for the future.*

So far, the description of maxiphobic heaps has been essentially language neutral. The code fragments have been given in Haskell, but they could just as easily have been given in, say, C or Smalltalk. However, the natural expression of this data structure in Haskell contains a subtlety that makes it characteristically functional. This subtlety occurs in a single line of the `join` function:

\[
\text{join} \ (\text{Fork} \ n \ a \ b) \ c = \text{Fork} \ (n + \text{size} \ c) \ a \ (\text{merge} \ b b \ c c)
\]

Here is the question that distinguishes functional data structures from imperative ones: Does the right-hand side of this line denote the creation of a new `Fork` node or the in-place modification of the old one? This innocent-sounding question has profound implications on the design, implementation, and analysis of data structures.

If `join` destructively modifies the existing node, as it would in most imperative implementations, then the data structure is *ephemeral*. An ephemeral data structure exists in the perpetual present—it has no appreciation for or access to the resources of the past. For example, suppose we have a heap containing the elements 1, 2, and 3. If we insert the number 4, then the heap will contain the elements 1, 2, 3, and 4. The old version of the heap—the one containing only 1, 2, and 3—no longer exists.

On the other hand, if `join` creates a new `Fork` node, as it would in most functional implementations, then the data structure is *persistent*. A persistent data structure allows equal access to any current or past version of the data structure. Suppose we again have a heap containing the elements 1, 2, and 3. If we insert the number 4, we get a new heap containing the elements 1, 2, 3, and 4, but the old heap containing 1, 2, and 3 still exists. Future heap
operations can use either heap with equal ease.

Figure 1.3 illustrates how a functional implementation of maxiphobic heaps achieves persistence. Beginning with a heap containing the numbers 1, 2, 3, 4, 5, 6, and 8, we insert the number 7. Three new nodes are created (two by join and one by insert), containing the numbers 1, 4, and 7, respectively. The resulting heap comprises those three nodes, plus two subtrees inherited from the original heap—the subtree containing 2, 3, 5, and 6 and the subtree containing 8. Because we have not modified any of the existing nodes, the original heap co-exists in memory with the new heap, with certain subtrees belonging equally to both.

This sharing of nodes between versions of a data structure is crucial to the efficiency of this style of persistence. It is safe only if we never modify a shared node. Otherwise, we might unintentionally change multiple instances of a data structure when we only wanted to change one. For example, if we were to try to change the 8 to a 9 by modifying the node in place rather than creating a new node, we would affect both the original heap (without the 7) and the new heap (with the 7).

Exercise 1.4 In the sequence of heaps presented in Figure 1.2, which node or nodes are shared among the most heaps? (Hint: It is not the node containing the number 1 because there are actually seven different nodes containing the number 1, one per heap.) □

1.4 Round-robin heaps

When geese fly in a vee formation, the lead goose works the hardest. To avoid overtaxing any single bird, the geese in the flock take turns flying in the point position.

Maxiphobic heaps are already simple and efficient, but we can still do better. In this section, we will dramatically simplify the implementation of heaps without changing the running times of the heap operations. This is not the mythical free lunch, however. The price will be a corresponding increase in the difficulty of proving the time bounds.

Imagine you are dealer in a game of bridge. You need to distribute 52 cards evenly among four players, 13 cards each. This is a social game, so you want to be able to chat with your friends while dealing. How do you deal? You might attempt to deal 13 cards to the first player, followed by 13 cards to the second player, and so forth. Unfortunately, this approach requires accurate counting, which can be rather challenging in the middle of a conversation. A simpler approach is to deal the cards in a round-robin fashion: one card to each player, followed by a second card to each player, and so forth. This task is cognitively much simpler. Rather than counting, you only need to keep track of who received the most recent card, which the position of your hands will usually tell you.
Can we apply the same intuition to heaps? Looking back at Figure 1.2, we see that, for increasing sequences at least, the end result of counting and comparing tree sizes is to distribute the incoming nodes evenly among the subtrees at any level of the heap. But surely we can achieve this result without storing an entire integer at each node! All we need is a scheme for each individual node to distribute incoming nodes evenly between its two subtrees. We dub the resulting heaps round-robin heaps.

We colour each node to indicate which of its two subtrees should receive the next incoming node. A node coloured robin’s-egg blue sends the next incoming node to its left subtree, and a node coloured robin’s-breast red sends the next incoming node to its right subtree. It does not matter which colour we use for new nodes, so we arbitrarily choose blue. The implementation is straightforward, with the \texttt{join} function using the colour to determine whether to merge \(c\) with \(a\) or \(b\) and then changing the colour for next time.

Figure 1.4 shows the code for round-robin heaps, and Figure 1.5 illustrates a sequence of insertions into a sample heap.

The \texttt{join} function for round-robin heaps is significantly shorter and simpler than the \texttt{join} function for maxiphobic heaps, but we are not finished yet. Instead of using a colour to encode which subtree gets the next incoming node, we can use the two subtree slots in the \texttt{Fork} constructor as a tiny, fixed-size FIFO queue. The left subtree is the front of the queue, and the right subtree is the back of the queue. When the front subtree gets the next incoming node,
it moves to the back of the queue, and the subtree that used to be in the back moves up to the front. This implementation, known as skew heaps [120], is shown in Figure 1.6. It is hard to imagine that any implementation could be simpler! The Tree datatype and the heap operations return to their original definitions (from Section 1.1), and join becomes simply

\[
join \left( Fork \times a \times b \right) \times c = Fork \times b \left( merge \times a \times c \right)
\]

Figure 1.7 illustrates a sequence of insertions into a sample skew heap.

**Exercise 1.5** Come up with sequences of insertions and merges that will result in the following round-robin heaps:

What do the same sequences produce using skew heaps? □

---

2Most presentations of skew heaps reverse the meaning of the left and right subtrees, treating the right as the front and the left as the back. However, this difference is insignificant.
1.5 Analysis of skew heaps

*In the daytime, when the sun is out, the owl goes deep into the hollow and sleeps. That is, they say he sleeps, but I don’t believe it. How could anyone sleep so long? I think he sits in there, part of the time at least, and thinks. And that’s why he knows so much.*


Figure 1.7 suggests an $O(\log n)$ bound for insertions into round-robin or skew heaps, but Exercise 1.5 dashes that hope. An insertion or merge involving one of those trees may take $O(n)$ time in the worst case! On the other hand, the sequences of operations leading up to the trees in Exercise 1.5 take only $O(n\log n)$ time, so even if the next insertion takes another $O(n)$ steps, we still come out ahead. If we are willing to accept amortised rather than worst-case bounds, we can show that `merge`, `insert`, and `deleteMin` still run in $O(\log n)$ time.

An *amortised* analysis [127] looks at the worst case for a sequence of operations, rather than for a single operation. The cost for the entire sequence is averaged over all the operations in the sequence. It is expected that certain individual operations will take more than the average, but this does no harm as long as enough other operations take less than the average.

We will use the accounting notion of `credits` [127] to analyse skew heaps. Each credit pays for a constant amount of work, and no work may be performed without a corresponding credit. Each operation is allocated a certain number
of credits, and any unused credits at the end of the operation are saved for use by future operations that run out of their own credits. If we can prove that we never run out of credits, then we know that our total running time is bounded by the total number of credits allocated, and we claim that each operation runs in amortised time proportional to the number of credits allocated to that operation.

As an analogy, imagine that you work for a company that allows 15 sick days per year, and that unused sick days may be saved for future years. As long as you never run out of sick days, you can be sure that you have spent no more than 15 sick days per year in an amortised sense, even if you happened to spend several years’ worth of sick days in one particularly unhealthy year.

Applying these ideas to skew heaps, we will show that allocating $O(\log n)$ credits per operation is enough to never run out, and therefore that each operation runs in $O(\log n)$ amortised time.

Define a node to be good if its right subtree is at least as big as its left, and bad if its left subtree is bigger than its right. We insist that every bad node carry a single spare credit.

Consider a merge of two trees $T_1$ and $T_2$ of sizes $m_1$ and $m_2$, respectively, and let $n = m_1 + m_2$. We will show that such a merge requires $O(\log m_1 + \log m_2) = O(\log n)$ credits. Suppose that there are a total of $k + 1$ merge steps, the first $k$ of which invoke the join function.

$$join(Fork \times a \ b) \ c = Fork \times b (merge \ a \ c)$$

There are two cases for each of the $k$ join steps:

- If $Fork \times a \ b$ is bad, then we use its credit to pay for this step. The resulting node is guaranteed to be good.
- If $Fork \times a \ b$ is good, then the resulting node may be bad or good, depending on the size of $c$. We need at most two credits: one to pay for this step and one to give to the resulting node in case it is bad.
Altogether, we allocate at most \( 2g + 1 \) credits for the entire merge, where \( g \) is the number of good nodes encountered by \( \text{join} \) and the extra credit pays for the terminal \( \text{merge} \) step (which involves a \( \text{Null} \)). All that remains is to show that \( g \leq \log m_1 + \log m_2 \).

If \( \text{Fork} x a b \) is good, then \( |a| \leq |b| \). Therefore, in the recursive call \( \text{merge} a c \), the number of elements from the tree containing \( a \) has been decreased by more than half. This can happen to \( T_1 \) no more than \( \log m_1 \) times, and to \( T_2 \) no more than \( \log m_2 \) times. Therefore, \( g \leq \log m_1 + \log m_2 \) and we conclude that \( \text{merge} \) runs in \( O(\log n) \) amortised time, from which we conclude that \( \text{insert} \) and \( \text{deleteMin} \) do also.

**Exercise 1.6** Consider a skew heap built as suggested by Exercise 1.5. How many credits does such a tree have? If you then insert the number 10, how many credits does the new tree have? □

1.6 Lazy evaluation

_Sighed Mayzie, a lazy bird hatching an egg:
I’m tired and I’m bored
And I’ve kinks in my leg
From sitting, just sitting here day after day.
It’s work! How I hate it!
I’d much rather play!_

– Dr. Seuss, _Horton Hatches the Egg_

Unfortunately, the proof in the previous section fails if we allow persistence. Inserting into a tree with lots of bad nodes uses up some of the credits on those nodes. If we then take advantage of persistence and do another insertion into the original tree, as opposed to the tree resulting from the first insertion, we may run out of credits, because the credits we need from the tree’s bad nodes have already been spent.

We still have one more trick up our sleeves, however—lazy evaluation. Haskell is a lazy language, meaning that function calls are not executed until their results are needed. As with persistence, the key to understanding lazy evaluation lies in the \( \text{join} \) function:

\[
\text{join} (\text{Fork} x a b) c = \text{Fork} x b (\text{merge} a c)
\]

Here is the question that distinguishes lazy languages such as Haskell from strict languages such as Standard ML: Is the recursive merge in the right-hand side of this line executed as part of the \( \text{join} \), or does \( \text{join} \) simply build the new \( \text{Fork} \) node and leave the merge unevaluated? Again, the answer to this question has profound implications on the design, implementation, and analysis of data structures.

If \( \text{join} \) executes the recursive merge immediately, as it would in a strict language, then data structures such as skew heaps cannot be made persistent.
in the functional style without invalidating their amortised time bounds. Intuitively, the reason is that persistence encourages us to reuse old versions of data structures, but once the credits of a particular version have been spent, they cannot be spent again.

On the other hand, if \textit{join} merely saves the information necessary to execute the recursive merge later, as it would in a lazy language, then skew heaps and many other amortised data structures can be made persistent without damaging their time bounds. We actually need one more property from lazy evaluation to achieve this goal—once a delayed merge has been executed, the resulting tree must be saved so that the delayed merge need never be executed again, even if the result is needed in different parts of the computation. Fortunately, Haskell provides exactly this behaviour.

We will prove that a lazy implementation of skew heaps retains its bounds in the next section. In the remainder of this section, we consider an example that illustrates how lazy evaluation can help skew heaps cope with persistence.

We begin with a heap with lots of bad nodes, such as those constructed in Exercise 1.5. In particular, assume we begin with the tree

Call this tree $T$. Now, assume we construct 100 new trees, $T_{100} \ldots T_{199}$, each $T_i$ the result of inserting $i$ into $T$. We now have 100 trees of the form

Each insertion takes $O(n)$ steps, so all the insertions together take $O(n^2)$ steps. Or, at least, that would be the time required in a strict language.

Using lazy evaluation, each insertion still takes $O(n)$ steps, but only one of those steps is executed immediately. In other words, inserting $i$ produces the tree
where $\oplus$ represents the postponed merge. It is not a node of the tree; rather, it is a potential node. It will not become a real node until some other operation is executed that needs information from that node. For example, if we call deleteMin on $T_i$, we get

When we delete the 1, we must compare 2 with the label of its sibling to determine which will become the new root. To answer that, we need the label of the sibling node, which means we need the actual node. Therefore, we execute the delayed computation, producing the node containing 3. However, note that we did not execute all the remaining steps of the insertion. Instead we executed just enough to produce the information we needed at the moment, and no more. In this case, we executed only the next step of the insertion, and re-postponed all the remaining steps. It would take another two deletions to execute the merge step involving 5, two more to execute the merge step involving 7, and so on.

In fact, then, the 100 insertions that created $T_{100} \ldots T_{199}$ ran in only $O(n)$ time, because each executed only a single step. The potential for the remaining steps is still there, but realising that potential would require roughly $O(n^2)$ deletions, about 100 for each $T_i$. By shifting the charge for the excess steps from the original insertions to these deletions, we can keep the amortised cost of each operation low.

**Exercise 1.7** Although the example considered in this section initially appeared to be expensive, it actually turned out to be quite cheap, once we accounted for lazy evaluation. Come up with a sequence of $n$ priority queue
operations that takes $\Theta(n \log n)$ time even after lazy evaluation is taken into account. □

1.7 Analysis of lazy skew heaps

The early bird gets the worm but the second mouse gets the cheese.

- Anonymous

The proof that lazy skew heaps support all operations in $O(\log n)$ amortised time is very similar to the proof in Section 1.5, but in terms of debits \([102]\) rather than credits. Each debit accounts for a constant amount of work that has been delayed by lazy evaluation, and must be discharged before the corresponding work can be executed. We will also assign debits to work that is not delayed, but those debits must be discharged immediately. If we can prove that no work is ever executed before the corresponding debit has been discharged, then we know that our total running time is bounded by the total number of debits discharged, and we claim that each operation runs in amortised time proportional to the number of debits discharged by that operation.

In the presence of persistence, we may accidentally discharge a debit more than once. Fortunately, this does no harm. We can be sure that the work corresponding to a particular debit will be executed at most once, no matter how many times we discharge it. Therefore, discharging a debit more than once can only result in an overestimate of the running time, which is safe.

We shall prove that $\text{merge}$ discharges $O(\log n)$ debits, from which we conclude that $\text{insert}$ and $\text{deleteMin}$ discharge $O(\log n)$ debits as well.

As in Section 1.5, define a node to be good if its right subtree is at least as big as its left, and bad if its left subtree is bigger than its right. For the purposes of this definition, we count all logical nodes of a tree, even if some of them have not yet been physically constructed because of lazy evaluation. Good nodes may carry a single debit, but bad nodes must be free of outstanding debits.

Consider a merge of two trees $T_1$ and $T_2$ of sizes $m_1$ and $m_2$, respectively, and let $n = m_1 + m_2$. We will show that such a merge discharges $O(\log m_1 + \log m_2) = O(\log n)$ debits. Suppose that there are a total of $k + 1$ $\text{merge}$ steps, the first $k$ of which invoke the $\text{join}$ function.

$\text{join} \ (\text{Fork} \ x \ a \ b) \ c = \text{Fork} \ x \ b \ (\text{merge} \ a \ c)$

There are two cases for each of the $k$ $\text{join}$ steps:

- If $\text{Fork} \ x \ a \ b$ is bad, then the resulting node is guaranteed to be good. We attach the debit for this $\text{join}$ step to the resulting node, but do not discharge it.
• If $Fork \ a \ b$ is good, then the resulting node may be bad or good. We discharge at most two debits: the existing one attached to the original good node, and the one for this join step in case the resulting node is bad.

In addition, we discharge up to two debits for the terminal merge step: one for the step itself, and one if the non-null node is good. Altogether, we discharge at most $2g + 2$ debits for the entire merge, where $g$ is the number of good nodes encountered during the merge. As in Section 1.5, the definition of good nodes guarantees that $g \leq \log m_1 + \log m_2$, so merge runs in $O(\log n)$ amortised time.

**Exercise 1.8** Re-implement skew heaps using ternary trees rather than binary trees. In other words, add a third slot to the miniature FIFO queue in each node. Adapt the proof that skew heaps run in $O(\log n)$ time to this new implementation. □

### 1.8 Chapter notes

See [99, 101] for more information on debit-based amortised analysis, and [102] for a more comprehensive look at functional data structures. The analysis of lazy skew heaps initially appeared in [47] (written in Spanish).

Maxiphobic heaps are new, and are intended as a replacement for leftist heaps [29, 80, 22]. The code for maxiphobic heaps is slightly messier, but the analysis is simpler. Round-robin heaps are also new, but are merely a pedagogical stepping-stone on the path to skew heaps. Both maxiphobic and skew heaps make good general-purpose implementations of priority queues.
Index

==⇒ (QuickCheck precondition), 19
<=⇒ (Lava equality), 158

abstract model, 20
abstraction, 41
accumulating parameter, 54, 258, 259
acquisition date, 110, 112
active source, 79
adder
  bit, see bit adder
  full, see full adder
  half, see half adder
  ripple-carry, see ripple-carry adder
adder, 154
adjunction, 208
algebra of music, 73
algebraic specification, 24
alpha-beta pruning, 98–101
amortised analysis, 10–12, 15–16
anamorphism, see unfold
apomorphism, 47, 54
arbitrage, 120
Arbitrary, 28
arbitrary, 27
arguments exhausted, 25
arrow, 201–222, 259
arrow transformer, 220
associativity, 18
asymmetric branching, 37
automaton, 203
backtracking, 202

behaviour, 110
binary heap tree, 1–3
binary search tree, 1
binary tree, 29, 79, 194, 258
birthday, 107, 108, 253
bit adder, 153
bitAdder, 154
bitonic merge, 166–168, 175, 220
breadth-first search, 184–187
breadth-first traversal, see traversal, breadth-first
bridge, 7
butterfly circuit, 162–166, 220
cat-elimination, 55, 87, 93, 258, 259
catamorphism, see fold
category, 203
Checkable, 158
choose, 26
Church numeral, 49
classify, 31
coiteration, see unfold
collect, 31
Colour, 135
combinator, 105–106, 108, 112, 128,
  155, 177, 182, 191–193, 223, 255
commutativity, 152
completeness, 36
composition
  parallel, see parallel composition
  serial, see serial composition
compression, 248–252
computer music, 61
cond, 137
connection pattern, 155
container type, 247
contract
financial, 105–128
corecursion
primitive, see primitive corecursion
counter, 161
coverage, 30
crop, 144, 146
data parallel, 202
data structure, 1–16
ephemeral, see ephemeral data structure
persistent, see persistent data structure
dataflow, 207, 222
declarative programming, 177
decorate, 102–103
deforestation, see fusion
delay, 160
dependent types, 262
deepth-first traversal, see traversal, depth-first
diagonalisation, 179
Dijkstra, 17
discount bond, 107, 110, 125
do notation, 179, 180, 183, 213, 214
domain-specific language, 18, 61, 106, 131
dynamic value, 250–252
deficit buffer, 23
elements, 34
ephemeral data structure, 6
executable specification, 20
existentially quantified type, 246, 250, 259
factorial, 48
fast reverse, 86–88
feedback, 211
Filter, 140
filter, 59
FilterC, 141
financial model, 116
fold, 41–60, 254
on general trees, 58, 59
on lists, 43, 44, 46, 56, 58–60
on numbers, 49
on rose trees, 52–54
foldr, see fold on lists
forAll, 19, 25
forest, see rose tree
formula, 32
Fourier transform, 220
FPGA cell, 170
Frac, 134
frequency, 27
full adder, 153
functional and logic programming, 177
functor, 204
fusion, 48
for fold on general trees, 58
for fold on lists, 43, 55, 57, 60
for fold on numbers, 49, 51
for fold on rose trees, 52, 53, 55
for unfold on general trees, 58
for unfold on lists, 45, 57
for unfold on numbers, 50
map, 43
mechanisation, 85–88
Gen, 25, 26
general recursion, 50
generic programming, 248–250, 252–255
genericity, 42
GHC, 18
GHCi, 18
goto, 41
half adder, 152
Hamming’s Problem, 57
Haskore, 61
heap
binary, see binary heap tree
lazy skew, see lazy skew heap
maxiphobic, see maxiphobic heap
round robin, see round-robin heap
skew, see skew heap
homogeneous function, 218, 222
Hugs, 18
hylomorphism, 48
on general trees, 58, 59
on lists, 48, 51, 55, 57
on numbers, 51
on trees, 48

ihv, 163
Image, 133
ImageC, 137
insert, 19
interpolation, 136, 137
invariant, 22
iteration, see fold

Kahn network, 207
Kleisli, 209

Lambek’s Lemma, 49
lattice, 124–126
lazy evaluation, 1, 12–15, 232
lazy skew heap, 13, 15–16
lerp, see interpolation
level-order traversal, see traversal,
level-order
lexical ordering, 58
lifting, 137
limit contract, 115
list of successes, 202
list partition, 94–95
lists, 42–48
logic programming, 177–199, 258
logical connective, 32
logical variable, 177, 188–191
long zip, 54, 56, 96, 181, 185
look-up table, 170
lzw, see long zip

map, 43
matrix, 184
maxiphobic heap, 4–5, 16
merge
bitonic, see bitonic merge
odd-even, see odd-even merge
mergeB, 168
MIDI, 78
minimisation, 50
minimum depth, 79–83
model, 33
Monad, 26
monad, 26, 179, 180, 201–222, 251
Music, 62
music, 61

natural numbers, see numbers
non-determinism, 177, 202
normal form, 225, 226, 229
numbers, 49–51

observable, 110, 112, 122
odd–even merge, 166, 169
oneof, 27
optimisation, 232, 234, 239
option, 114
American, 115, 122
European, 114
overlay, 136

paragraph formatting, 236
parallel algorithm, 202
parallel composition, 155
parallel prefix, 218
paramorphism
on lists, 44, 46
performance, 71
persistent data structure, 1, 6–7, 12
Point, 132
point-free, 201, 208, 213
point-wise, see point-free
polar coordinates, 133, 144, 145
PolarPoint, 133
precondition, 19
pretty printing, 199, 223–243, 249,
252
primitive corecursion, see apomor-
phism
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive recursion</td>
<td>50</td>
</tr>
<tr>
<td>priority queue</td>
<td>1–16</td>
</tr>
<tr>
<td>probability</td>
<td>27</td>
</tr>
<tr>
<td>probability distribution</td>
<td>28</td>
</tr>
<tr>
<td>proc notation</td>
<td>213–215</td>
</tr>
<tr>
<td>product functor</td>
<td>204</td>
</tr>
<tr>
<td>Prolog</td>
<td>177</td>
</tr>
<tr>
<td>Property</td>
<td>19</td>
</tr>
<tr>
<td>property</td>
<td>18</td>
</tr>
<tr>
<td>safety</td>
<td>157</td>
</tr>
<tr>
<td>propositional logic</td>
<td>32</td>
</tr>
<tr>
<td>queue</td>
<td>20–25, 53</td>
</tr>
<tr>
<td>QuickCheck</td>
<td>17–39</td>
</tr>
<tr>
<td>quickCheck</td>
<td>18</td>
</tr>
<tr>
<td>random</td>
<td>18</td>
</tr>
<tr>
<td>rank-2 type</td>
<td>254</td>
</tr>
<tr>
<td>ravelling lists</td>
<td>56</td>
</tr>
<tr>
<td>records</td>
<td>58</td>
</tr>
<tr>
<td>recursion</td>
<td>18</td>
</tr>
<tr>
<td>safety</td>
<td>157</td>
</tr>
<tr>
<td>general recursion</td>
<td>18</td>
</tr>
<tr>
<td>primitive recursion</td>
<td>50</td>
</tr>
<tr>
<td>Region</td>
<td>133</td>
</tr>
<tr>
<td>regions</td>
<td>142</td>
</tr>
<tr>
<td>retrieve function</td>
<td>21, 24</td>
</tr>
<tr>
<td>riffle</td>
<td>163, 218</td>
</tr>
<tr>
<td>ripple-carry adder</td>
<td>154</td>
</tr>
<tr>
<td>rose tree</td>
<td>52–56, 92, 96, 196</td>
</tr>
<tr>
<td>round-robin heap</td>
<td>7–10</td>
</tr>
<tr>
<td>row</td>
<td>155, 173</td>
</tr>
<tr>
<td>safety properties</td>
<td>157</td>
</tr>
<tr>
<td>scan</td>
<td>178</td>
</tr>
<tr>
<td>parallel prefix</td>
<td>178</td>
</tr>
<tr>
<td>serial composition</td>
<td>155</td>
</tr>
<tr>
<td>simulate</td>
<td>153</td>
</tr>
<tr>
<td>combinational</td>
<td>153</td>
</tr>
<tr>
<td>sequential</td>
<td>160</td>
</tr>
<tr>
<td>sized</td>
<td>29</td>
</tr>
<tr>
<td>skew heap</td>
<td>9, 10</td>
</tr>
<tr>
<td>lazy</td>
<td>157</td>
</tr>
<tr>
<td>sorting</td>
<td>42–48, 56–60</td>
</tr>
<tr>
<td>bitonic</td>
<td>41–60, 154, 254</td>
</tr>
<tr>
<td></td>
<td>58, 59</td>
</tr>
<tr>
<td></td>
<td>44, 47, 54–57, 59</td>
</tr>
<tr>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>52</td>
</tr>
<tr>
<td>bubble sort</td>
<td>46</td>
</tr>
<tr>
<td>bucket sort</td>
<td>58</td>
</tr>
<tr>
<td>insertion sort</td>
<td>43, 56</td>
</tr>
<tr>
<td>odd-even</td>
<td>93–94</td>
</tr>
<tr>
<td>radix sort</td>
<td>58–60</td>
</tr>
<tr>
<td>selection sort</td>
<td>46</td>
</tr>
<tr>
<td>shell sort</td>
<td>56–58</td>
</tr>
<tr>
<td>soundness</td>
<td>35</td>
</tr>
<tr>
<td>specification</td>
<td>17</td>
</tr>
<tr>
<td>stack</td>
<td>53</td>
</tr>
<tr>
<td>static typing</td>
<td>247</td>
</tr>
<tr>
<td>steep sequences</td>
<td>93–94</td>
</tr>
<tr>
<td>strictness</td>
<td>43, 51</td>
</tr>
<tr>
<td>structure clash</td>
<td>48</td>
</tr>
<tr>
<td>substitution</td>
<td>189, 190, 198</td>
</tr>
<tr>
<td>swirl</td>
<td>140</td>
</tr>
<tr>
<td>tableau method</td>
<td>33</td>
</tr>
<tr>
<td>term rewriting</td>
<td>85</td>
</tr>
<tr>
<td>test data generator</td>
<td>25, 26</td>
</tr>
<tr>
<td>testing</td>
<td>17</td>
</tr>
<tr>
<td>theorem prover</td>
<td>32</td>
</tr>
<tr>
<td>trace</td>
<td>17</td>
</tr>
<tr>
<td>Transform</td>
<td>139</td>
</tr>
<tr>
<td>transpose</td>
<td>56</td>
</tr>
<tr>
<td>traversal</td>
<td>252–255</td>
</tr>
<tr>
<td>breadth-first</td>
<td>53</td>
</tr>
<tr>
<td>depth-first</td>
<td>53</td>
</tr>
<tr>
<td>level-order</td>
<td>54, 96–97</td>
</tr>
<tr>
<td>tree</td>
<td>1</td>
</tr>
<tr>
<td>balanced</td>
<td>1</td>
</tr>
<tr>
<td>binary</td>
<td>44</td>
</tr>
<tr>
<td>binary heap</td>
<td>15</td>
</tr>
<tr>
<td>layout</td>
<td>224, 236</td>
</tr>
<tr>
<td>rose</td>
<td>52</td>
</tr>
<tr>
<td>search</td>
<td>57</td>
</tr>
<tr>
<td>traversal of</td>
<td>30</td>
</tr>
<tr>
<td>trivial</td>
<td>30</td>
</tr>
<tr>
<td>tupling</td>
<td>44</td>
</tr>
<tr>
<td>two</td>
<td>162</td>
</tr>
<tr>
<td>unfold</td>
<td>41–60, 154, 254</td>
</tr>
<tr>
<td>on general trees</td>
<td>58, 59</td>
</tr>
<tr>
<td>on lists</td>
<td>44, 47, 54–57, 59</td>
</tr>
<tr>
<td>on numbers</td>
<td>50</td>
</tr>
<tr>
<td>on rose trees</td>
<td>52</td>
</tr>
</tbody>
</table>
Index

unfoldr, see unfold on lists
unification, 190, 199
union, 32
unique prefix property, 249
universal datatype, 250
universal property
  of fold on lists, 43
  of fold on numbers, 49
  of fold on rose trees, 52
  of unfold on lists, 45, 55, 56
  of unfold on numbers, 50
universally quantified type, 245, 253, 254, 260, 261
unparsing, 257–259
unriffle, 163, 218, 220
valuation, 32, 116–123
Vector, 139
verify, 157
while, 50
XML, 237–238
zero-one principle, 167
zip
  long, see long zip