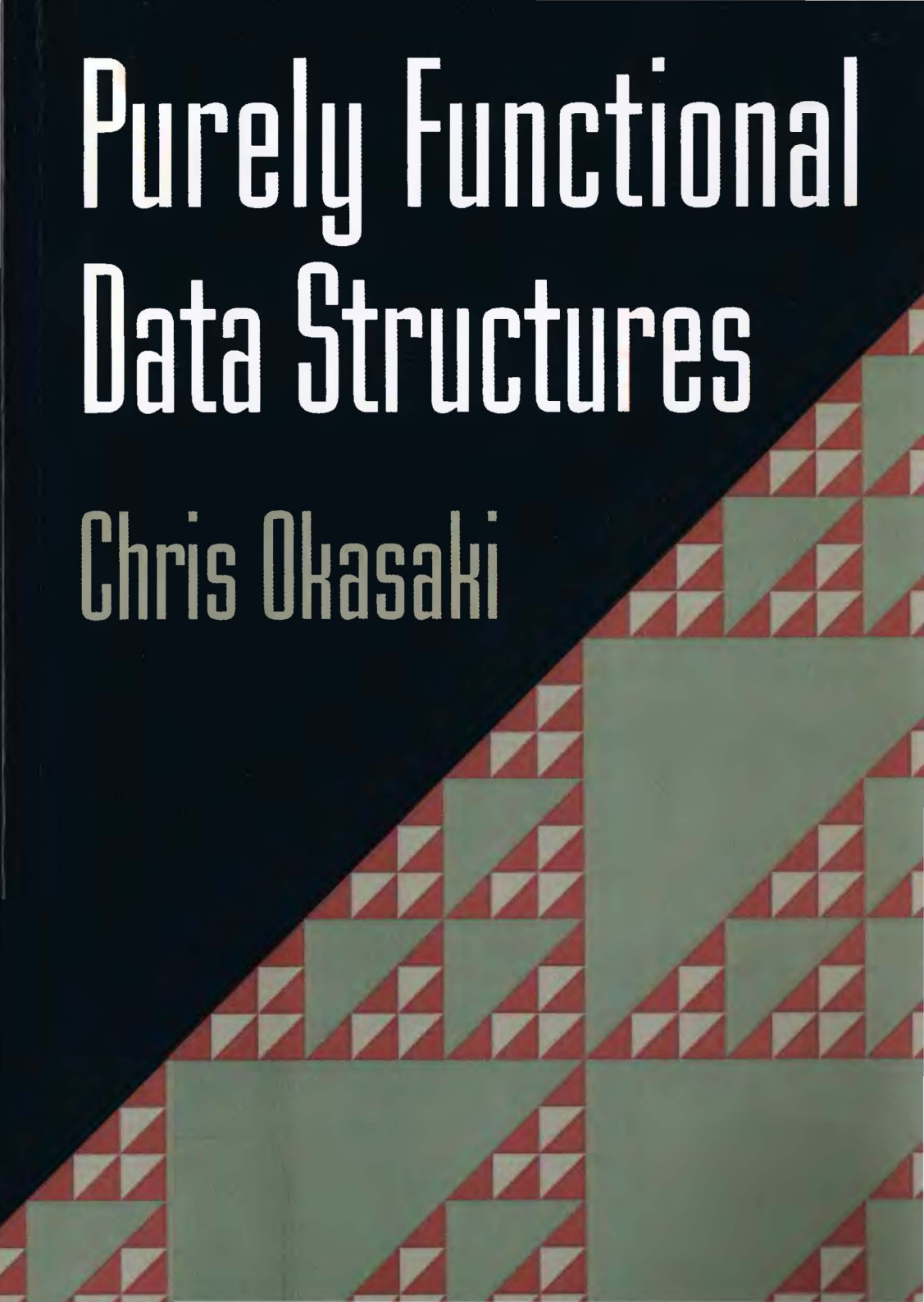
Purely Functional Data Structures

Chris Okasaki



PURELY FUNCTIONAL DATA STRUCTURES

Most books on data structures assume an imperative language like C or C++. However, data structures for these languages do not always translate well to functional languages such as Standard ML, Haskell, or Scheme. This book describes data structures from the point of view of functional languages, with examples, and presents design techniques so that programmers can develop their own functional data structures. It includes both classical data structures, such as red-black trees and binomial queues, and a host of new data structures developed exclusively for functional languages. All source code is given in Standard ML and Haskell, and most of the programs can easily be adapted to other functional languages.

This handy reference for professional programmers working with functional languages can also be used as a tutorial or for self-study.

PURELY FUNCTIONAL DATA STRUCTURES

CHRIS OKASAKI
COLUMBIA UNIVERSITY

CAMBRIDGE
UNIVERSITY PRESS

Contents

Preface *page* ix

1 Introduction **1**
- 1.1 Functional vs. Imperative Data Structures 1
- 1.2 Strict vs. Lazy Evaluation 2
- 1.3 Terminology 3
- 1.4 Approach 4
- 1.5 Overview 4

2 Persistence **7**
- 2.1 Lists 7
- 2.2 Binary Search Trees 11
- 2.3 Chapter Notes 15

3 Some Familiar Data Structures in a Functional Setting **17**
- 3.1 Leftist Heaps 17
- 3.2 Binomial Heaps 20
- 3.3 Red-Black Trees 24
- 3.4 Chapter Notes 29

4 Lazy Evaluation **31**
- 4.1 $-notation 31
- 4.2 Streams 34
- 4.3 Chapter Notes 37

5 Fundamentals of Amortization **39**
- 5.1 Techniques of Amortized Analysis 39
- 5.2 Queues 42
- 5.3 Binomial Heaps 45
- 5.4 Splay Heaps 46
- 5.5 Pairing Heaps 52

5.6 The Bad News 54
5.7 Chapter Notes 55

6 Amortization and Persistence via Lazy Evaluation 57
6.1 Execution Traces and Logical Time 57
6.2 Reconciling Amortization and Persistence 58
6.2.1 The Role of Lazy Evaluation 59
6.2.2 A Framework for Analyzing Lazy Data Structures 59
6.3 The Banker's Method 61
6.3.1 Justifying the Banker's Method 62
6.3.2 Example: Queues 64
6.3.3 Debit Inheritance 67
6.4 The Physicist's Method 68
6.4.1 Example: Binomial Heaps 70
6.4.2 Example: Queues 72
6.4.3 Example: Bottom-Up Mergesort with Sharing 74
6.5 Lazy Pairing Heaps 79
6.6 Chapter Notes 81

7 Eliminating Amortization 83
7.1 Scheduling 84
7.2 Real-Time Queues 86
7.3 Binomial Heaps 89
7.4 Bottom-Up Mergesort with Sharing 94
7.5 Chapter Notes 97

8 Lazy Rebuilding 99
8.1 Batched Rebuilding 99
8.2 Global Rebuilding 101
8.2.1 Example: Hood–Melville Real-Time Queues 102
8.3 Lazy Rebuilding 104
8.4 Double-Ended Queues 106
8.4.1 Output-Restricted Deques 107
8.4.2 Banker's Deques 108
8.4.3 Real-Time Deques 111
8.5 Chapter Notes 113

9 Numerical Representations 115
9.1 Positional Number Systems 116
9.2 Binary Numbers 116
9.2.1 Binary Random-Access Lists 119
9.2.2 Zeroless Representations 122

9.2.3 Lazy Representations 125
9.2.4 Segmented Representations 127
9.3 Skew Binary Numbers 130
9.3.1 Skew Binary Random-Access Lists 132
9.3.2 Skew Binomial Heaps 134
9.4 Trinary and Quaternary Numbers 138
9.5 Chapter Notes 140

10 Data-Structural Bootstrapping **141**
10.1 Structural Decomposition 142
10.1.1 Non-Uniform Recursion and Standard ML 143
10.1.2 Binary Random-Access Lists Revisited 144
10.1.3 Bootstrapped Queues 146
10.2 Structural Abstraction 151
10.2.1 Lists With Efficient Catenation 153
10.2.2 Heaps With Efficient Merging 158
10.3 Bootstrapping To Aggregate Types 163
10.3.1 Tries 163
10.3.2 Generalized Tries 166
10.4 Chapter Notes 169

11 Implicit Recursive Slowdown **171**
11.1 Queues and Deques 171
11.2 Catenable Double-Ended Queues 175
11.3 Chapter Notes 184

A Haskell Source Code **185**
Bibliography 207
Index 217

Preface

I first began programming in Standard ML in 1989. I had always enjoyed implementing efficient data structures, so I immediately set about translating some of my favorites into Standard ML. For some data structures, this was quite easy, and to my great delight, the resulting code was often both much clearer and much more concise than previous versions I had written in C or Pascal or Ada. However, the experience was not always so pleasant. Time after time, I found myself wanting to use destructive updates, which are discouraged in Standard ML and forbidden in many other functional languages. I sought advice in the existing literature, but found only a handful of papers. Gradually, I realized that this was unexplored territory, and began to search for new ways of doing things.

Eight years later, I am still searching. There are still many examples of data structures that I just do not know how to implement efficiently in a functional language. But along the way, I have learned many lessons about what *does* work in functional languages. This book is an attempt to codify these lessons. I hope that it will serve as both a reference for functional programmers and as a text for those wanting to learn more about data structures in a functional setting.

Standard ML Although the data structures in this book can be implemented in practically any functional language, I will use Standard ML for all my examples. The main advantages of Standard ML, at least for presentational purposes, are (1) that it is a strict language, which greatly simplifies reasoning about how much time a given algorithm will take, and (2) that it has an excellent module system that is ideally suited for describing these kinds of abstract data types. However, users of other languages, such as Haskell or Lisp, should find it quite easy to adapt these examples to their particular environments. (I provide Haskell translations of most of the examples in an appendix.) Even

C or Java programmers should find it relatively straightforward to implement these data structures, although C's lack of automatic garbage collection can sometimes prove painful.

For those readers who are not familiar with Standard ML, I recommend Paulson's *ML for the Working Programmer* [Pau96] or Ullman's *Elements of ML Programming* [Ull94] as introductions to the language.

Other Prerequisites This book is not intended as a first introduction to data structures in general. I assume that the reader is reasonably familiar with basic abstract data types such as stacks, queues, heaps (priority queues), and finite maps (dictionaries). I also assume familiarity with the basics of algorithm analysis, especially "big-Oh" notation (e.g., $O(n \log n)$). These topics are frequently taught in the second course for computer science majors.

Acknowledgments My understanding of functional data structures has been greatly enriched by discussions with many people over the years. I would particularly like to thank Peter Lee, Henry Baker, Gerth Brodal, Bob Harper, Haim Kaplan, Graeme Moss, Simon Peyton Jones, and Bob Tarjan.

1

Introduction

When a C programmer needs an efficient data structure for a particular problem, he or she can often simply look one up in any of a number of good textbooks or handbooks. Unfortunately, programmers in functional languages such as Standard ML or Haskell do not have this luxury. Although most of these books purport to be language-independent, they are unfortunately language-independent only in the sense of Henry Ford: Programmers can use any language they want, as long as it's imperative.† To rectify this imbalance, this book describes data structures from a functional point of view. We use Standard ML for all our examples, but the programs are easily translated into other functional languages such as Haskell or Lisp. We include Haskell versions of our programs in Appendix A.

1.1 Functional vs. Imperative Data Structures

The methodological benefits of functional languages are well known [Bac78, Hug89, HJ94], but still the vast majority of programs are written in imperative languages such as C. This apparent contradiction is easily explained by the fact that functional languages have historically been slower than their more traditional cousins, but this gap is narrowing. Impressive advances have been made across a wide front, from basic compiler technology to sophisticated analyses and optimizations. However, there is one aspect of functional programming that no amount of cleverness on the part of the compiler writer is likely to mitigate — the use of inferior or inappropriate data structures. Unfortunately, the existing literature has relatively little advice to offer on this subject.

Why should functional data structures be any more difficult to design and implement than imperative ones? There are two basic problems. First, from

† Henry Ford once said of the available colors for his Model T automobile, "[Customers] can have any color they want, as long as it's black."

the point of view of designing and implementing efficient data structures, functional programming's stricture against destructive updates (i.e., assignments) is a staggering handicap, tantamount to confiscating a master chef's knives. Like knives, destructive updates can be dangerous when misused, but tremendously effective when used properly. Imperative data structures often rely on assignments in crucial ways, and so different solutions must be found for functional programs.

The second difficulty is that functional data structures are expected to be more flexible than their imperative counterparts. In particular, when we update an imperative data structure we typically accept that the old version of the data structure will no longer be available, but, when we update a functional data structure, we expect that both the old and new versions of the data structure will be available for further processing. A data structure that supports multiple versions is called *persistent* while a data structure that allows only a single version at a time is called *ephemeral* [DSST89]. Functional programming languages have the curious property that *all* data structures are automatically persistent. Imperative data structures are typically ephemeral, but when a persistent data structure is required, imperative programmers are not surprised if the persistent data structure is more complicated and perhaps even asymptotically less efficient than an equivalent ephemeral data structure.

Furthermore, theoreticians have established lower bounds suggesting that functional programming languages may be fundamentally less efficient than imperative languages in some situations [BAG92, Pip96]. In light of all these points, functional data structures sometimes seem like the dancing bear, of whom it is said, "the amazing thing is not that [he] dances so well, but that [he] dances at all!" In practice, however, the situation is not nearly so bleak. As we shall see, it is often possible to devise functional data structures that are asymptotically as efficient as the best imperative solutions.

1.2 Strict vs. Lazy Evaluation

Most (sequential) functional programming languages can be classified as either *strict* or *lazy*, according to their order of evaluation. Which is superior is a topic debated with sometimes religious fervor by functional programmers. The difference between the two evaluation orders is most apparent in their treatment of arguments to functions. In strict languages, the arguments to a function are evaluated before the body of the function. In lazy languages, arguments are evaluated in a demand-driven fashion; they are initially passed in unevaluated form and are evaluated only when (and if!) the computation needs the results to continue. Furthermore, once a given argument is evaluated, the value of that

argument is cached so that, if it is ever needed again, it can be looked up rather than recomputed. This caching is known as *memoization* [Mic68].

Each evaluation order has its advantages and disadvantages, but strict evaluation is clearly superior in at least one area: ease of reasoning about asymptotic complexity. In strict languages, exactly which subexpressions will be evaluated, and when, is for the most part syntactically apparent. Thus, reasoning about the running time of a given program is relatively straightforward. However, in lazy languages, even experts frequently have difficulty when, or even if, a given subexpression will be evaluated. Programmers in such languages are often reduced to pretending the language is actually strict to make even gross estimates of running time!

Both evaluation orders have implications for the design and analysis of data structures. As we shall see, strict languages can describe worst-case data structures, but not amortized ones, and lazy languages can describe amortized data structures, but not worst-case ones. To be able to describe both kinds of data structures, we need a programming language that supports both evaluation orders. We achieve this by extending Standard ML with lazy evaluation primitives as described in Chapter 4.

1.3 Terminology

Any discussion of data structures is fraught with the potential for confusion, because the term *data structure* has at least four distinct, but related, meanings.

- *An abstract data type (that is, a type and a collection of functions on that type).* We will refer to this as an *abstraction.*
- *A concrete realization of an abstract data type.* We will refer to this as an *implementation*, but note that an implementation need not be actualized as code — a concrete design is sufficient.
- *An instance of a data type, such as a particular list or tree.* We will refer to such an instance generically as an *object* or a *version.* However, particular data types often have their own nomenclature. For example, we will refer to stack or queue objects simply as stacks or queues.
- *A unique identity that is invariant under updates.* For example, in a stack-based interpreter, we often speak informally about "the stack" as if there were only one stack, rather than different versions at different times. We will refer to this identity as a *persistent identity.* This issue mainly arises in the context of persistent data structures; when we speak of different versions of the same data structure, we mean that the different versions share a common persistent identity.

Roughly speaking, abstractions correspond to signatures in Standard ML, implementations to structures or functors, and objects or versions to values. There is no good analogue for persistent identities in Standard ML.†

The term *operation* is similarly overloaded, meaning both the functions supplied by an abstract data type and applications of those functions. We reserve the term *operation* for the latter meaning, and use the terms *function* or *operator* for the former.

1.4 Approach

Rather than attempting to catalog efficient data structures for every purpose (a hopeless task!), we instead concentrate on a handful of general techniques for designing efficient functional data structures and illustrate each technique with one or more implementations of fundamental abstractions such as sequences, heaps (priority queues), and search structures. Once you understand the techniques involved, you can easily adapt existing data structures to your particular needs, or even design new data structures from scratch.

1.5 Overview

This book is structured in three parts. The first part (Chapters 2 and 3) serves as an introduction to functional data structures.

- Chapter 2 describes how functional data structures achieve persistence.
- Chapter 3 examines three familiar data structures—leftist heaps, binomial heaps, and red-black trees—and shows how they can be implemented in Standard ML.

The second part (Chapters 4–7) concerns the relationship between lazy evaluation and amortization.

- Chapter 4 sets the stage by briefly reviewing the basic concepts of lazy evaluation and introducing the notation we use for describing lazy computations in Standard ML.
- Chapter 5 reviews the basic techniques of amortization and explains why these techniques are not appropriate for analyzing persistent data structures.

† The persistent identity of an ephemeral data structure can be reified as a reference cell, but this approach is insufficient for modelling the persistent identity of a persistent data structure.

- Chapter 6 describes the mediating role lazy evaluation plays in combining amortization and persistence, and gives two methods for analyzing the amortized cost of data structures implemented with lazy evaluation.
- Chapter 7 illustrates the power of combining strict and lazy evaluation in a single language. It describes how one can often derive a worst-case data structure from an amortized data structure by systematically scheduling the premature execution of lazy components.

The third part of the book (Chapters 8–11) explores a handful of general techniques for designing functional data structures.

- Chapter 8 describes *lazy rebuilding*, a lazy variant of *global rebuilding* [Ove83]. Lazy rebuilding is significantly simpler than global rebuilding, but yields amortized rather than worst-case bounds. Combining lazy rebuilding with the scheduling techniques of Chapter 7 often restores the worst-case bounds.
- Chapter 9 explores *numerical representations*, which are implementations designed in analogy to representations of numbers (typically binary numbers). In this model, designing efficient insertion and deletion routines corresponds to choosing variants of binary numbers in which adding or subtracting one take constant time.
- Chapter 10 examines *data-structural bootstrapping* [Buc93]. This technique comes in three flavors: *structural decomposition*, in which unbounded solutions are bootstrapped from bounded solutions; *structural abstraction*, in which efficient solutions are bootstrapped from inefficient solutions; and *bootstrapping to aggregate types*, in which implementations with atomic elements are bootstrapped to implementations with aggregate elements.
- Chapter 11 describes *implicit recursive slowdown*, a lazy variant of the *recursive-slowdown* technique of Kaplan and Tarjan [KT95]. As with lazy rebuilding, implicit recursive slowdown is significantly simpler than recursive slowdown, but yields amortized rather than worst-case bounds. Again, we can often recover the worst-case bounds using scheduling.

Finally, Appendix A includes Haskell translations of most of the implementations in this book.

2

Persistence

A distinctive property of functional data structures is that they are always *persistent*—updating a functional data structure does not destroy the existing version, but rather creates a new version that coexists with the old one. Persistence is achieved by *copying* the affected nodes of a data structure and making all changes in the copy rather than in the original. Because nodes are never modified directly, all nodes that are unaffected by an update can be *shared* between the old and new versions of the data structure without worrying that a change in one version will inadvertently be visible to the other.

In this chapter, we examine the details of copying and sharing for two simple data structures: lists and binary search trees.

2.1 Lists

We begin with simple linked lists, which are common in imperative programming and ubiquitous in functional programming. The core functions supported by lists are essentially those of the stack abstraction, which is described as a Standard ML signature in Figure 2.1. Lists and stacks can be implemented trivially using either the built-in type of lists (Figure 2.2) or a custom datatype (Figure 2.3).

Remark The signature in Figure 2.1 uses list nomenclature (cons, head, tail) rather than stack nomenclature (push, top, pop), because we regard stacks as an instance of the general class of sequences. Other instances include *queues*, *double-ended queues*, and *catenable lists*. We use consistent naming conventions for functions in all of these abstractions, so that different implementations can be substituted for each other with a minimum of fuss. ◇

Another common function on lists that we might consider adding to this signature is ++, which catenates (i.e., appends) two lists. In an imperative setting,

```
signature STACK =
sig
   type α Stack

   val empty   : α Stack
   val isEmpty : α Stack → bool

   val cons    : α × α Stack → α Stack
   val head    : α Stack → α          (* raises EMPTY if stack is empty *)
   val tail    : α Stack → α Stack    (* raises EMPTY if stack is empty *)
end
```

Figure 2.1. Signature for stacks.

```
structure List : STACK =
struct
   type α Stack = α list

   val empty = []
   fun isEmpty s = null s

   fun cons (x, s) = x :: s
   fun head s = hd s
   fun tail s = tl s
end
```

Figure 2.2. Implementation of stacks using the built-in type of lists.

```
structure CustomStack : STACK =
struct
   datatype α Stack = NIL | CONS of α × α Stack

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

   fun cons (x, s) = CONS (x, s)
   fun head NIL = raise EMPTY
     | head (CONS (x, s)) = x
   fun tail NIL = raise EMPTY
     | tail (CONS (x, s)) = s
end
```

Figure 2.3. Implementation of stacks using a custom datatype.

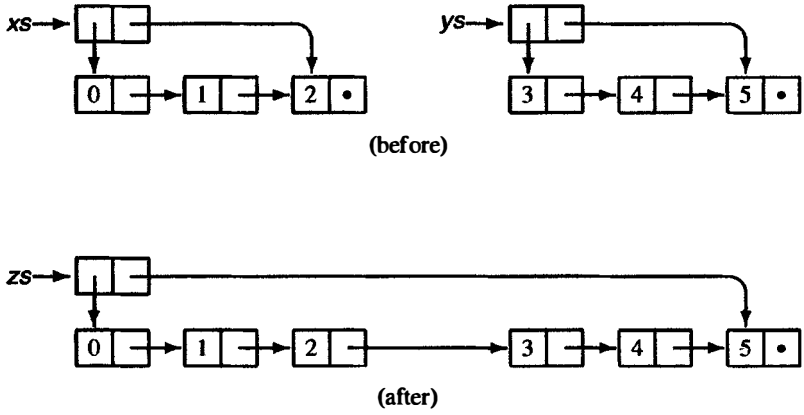


Figure 2.4. Executing *zs* = *xs* ++ *ys* in an imperative setting. Note that this operation destroys the argument lists, *xs* and *ys*.

this function can easily be supported in $O(1)$ time by maintaining pointers to both the first and last cell in each list. Then ++ simply modifies the last cell of the first list to point to the first cell of the second list. The result of this operation is shown pictorially in Figure 2.4. Note that this operation *destroys* both of its arguments—after executing *zs* = *xs* ++ *ys*, neither *xs* nor *ys* can be used again.

In a functional setting, we cannot destructively modify the last cell of the first list in this way. Instead, we *copy* the cell and modify the tail pointer of the copy. Then we copy the second-to-last cell and modify its tail to point to the copy of the last cell. We continue in this fashion until we have copied the entire list. This process can be implemented generically as

```
fun xs ++ ys = if isEmpty xs then ys else cons (head xs, tail xs ++ ys)
```

If we have access to the underlying representation (say, Standard ML's built-in lists), then we can rewrite this function using pattern matching as

```
fun [] ++ ys = ys
  | (x :: xs) ++ ys = x :: (xs ++ ys)
```

Figure 2.5 illustrates the result of catenating two lists. Note that after the oper-

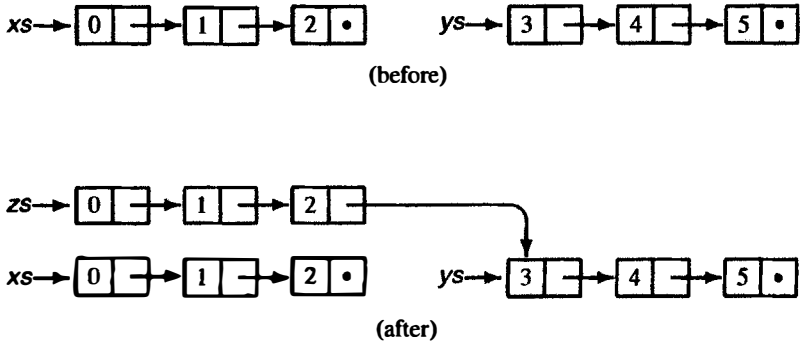


Figure 2.5. Executing *zs* = *xs* ++ *ys* in a functional setting. Notice that the argument lists, *xs* and *ys*, are unaffected by the operation.

ation, we are free to continue using the old lists, *xs* and *ys*, as well as the new list, *zs*. Thus, we get persistence, but at the cost of $O(n)$ copying.†

Although this is undeniably a lot of copying, notice that we did not have to copy the second list, *ys*. Instead, these nodes are shared between *ys* and *zs*. Another function that illustrates these twin concepts of copying and sharing is update, which changes the value of a node at a given index in the list. This function can be implemented as

```
fun update ([], i, y) = raise SUBSCRIPT
  | update (x :: xs, 0, y) = y :: xs
  | update (x :: xs, i, y) = x :: update (xs, i-1, y)
```

Here we do not copy the entire argument list. Rather, we copy only the node to be modified (node *i*) and all those nodes that contain direct or indirect pointers to node *i*. In other words, to modify a single node, we copy all the nodes on the path from the root to the node in question. All nodes that are not on this path are shared between the original version and the updated version. Figure 2.6 shows the results of updating the third node of a five-node list; the first three nodes are copied and the last two nodes are shared.

Remark This style of programming is greatly simplified by automatic garbage collection. It is crucial to reclaim the space of copies that are no longer needed, but the pervasive sharing of nodes makes manual garbage collection awkward.

† In Chapters 10 and 11, we will see how to support ++ in $O(1)$ time without sacrificing persistence.

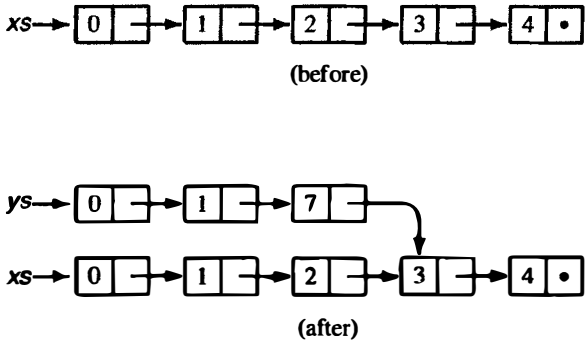


Figure 2.6. Executing *ys* = update(*xs*, 2, 7). Note the sharing between *xs* and *ys*.

Exercise 2.1 Write a function suffixes of type α list $\rightarrow$ α list list that takes a list *xs* and returns a list of all the suffixes of *xs* in decreasing order of length. For example,

suffixes [1,2,3,4] = [[1,2,3,4], [2,3,4], [3,4], [4], []]

Show that the resulting list of suffixes can be generated in $O(n)$ time and represented in $O(n)$ space.

2.2 Binary Search Trees

More complicated patterns of sharing are possible when there is more than one pointer field per node. Binary search trees provide a good example of this kind of sharing.

Binary search trees are binary trees with elements stored at the interior nodes in *symmetric order*, meaning that the element at any given node is greater than each element in its left subtree and less than each element in its right subtree. We represent binary search trees in Standard ML with the following type:

datatype Tree = E | T **of** Tree $\times$ Elem $\times$ Tree

where Elem is some fixed type of totally-ordered elements.

Remark Binary search trees are not polymorphic in the type of elements because they cannot accept arbitrary types as elements—only types that are equipped with a total ordering relation are suitable. However, this does not mean that we must re-implement binary search trees for each different element

```
signature SET =
sig
  type Elem
  type Set

  val empty  : Set
  val insert : Elem × Set → Set
  val member : Elem × Set → bool
end
```

Figure 2.7. Signature for sets.

type. Instead, we make the type of elements and its attendant comparison functions parameters of the *functor* that implements binary search trees (see Figure 2.9). ◇

We will use this representation to implement sets. However, it can easily be adapted to support other abstractions (e.g., finite maps) or fancier functions (e.g., find the ith smallest element) by augmenting the T constructor with extra fields.

Figure 2.7 describes a minimal signature for sets. This signature contains a value for the empty set and functions for inserting a new element and testing for membership. A more realistic implementation would probably include many additional functions, such as deleting an element or enumerating all elements.

The member function searches a tree by comparing the query element with the element at the root. If the query element is smaller than the root element, then we recursively search the left subtree. If the query element is larger than the root element, then we recursively search the right subtree. Otherwise the query element is equal to the element at the root, so we return true. If we ever reach the empty node, then the query element is not an element of the set, so we return false. This strategy is implemented as follows:

```
fun member (x, E) = false
  | member (x, T (a, y, b)) =
      if x < y then member (x, a)
      else if x > y then member (x, b)
      else true
```

Remark For simplicity, we have assumed that the comparison functions are named < and >. However, when these functions are passed as parameters to a functor, as they will be in Figure 2.9, it is often more convenient to use names such as lt or leq, and reserve < and > for comparing integers and other primitive types. ◇

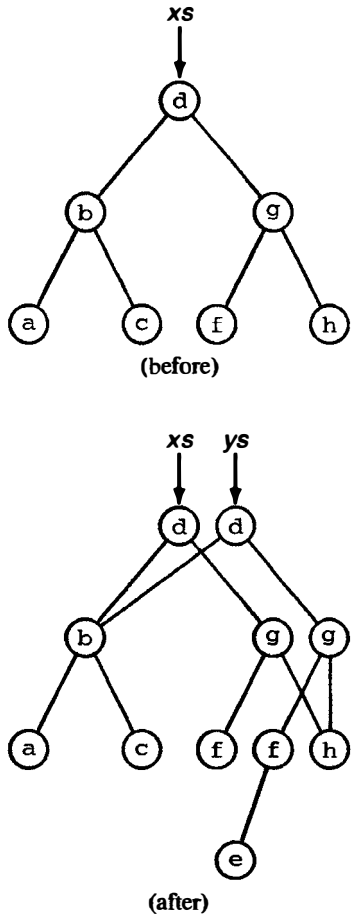


Figure 2.8. Execution of *ys* = insert ("e", *xs*). Once again, notice the sharing between *xs* and *ys*.

The insert function searches the tree using the same strategy as member, except that it copies every node along the way. When it finally reaches an empty node, it replaces the empty node with a node containing the new element.

```
fun insert (x, E) = T (E, x, E)
  | insert (x, s as T (a, y, b)) =
      if x < y then T (insert (x, a), y, b)
      else if x > y then T (a, y, insert (x, b))
      else s
```

Figure 2.8 illustrates a typical insertion. Every node that is copied shares one

```
signature ORDERED =
  (* a totally ordered type and its comparison functions *)
sig
  type T

  val eq : T × T → bool
  val lt : T × T → bool
  val leq : T × T → bool
end

functor UnbalancedSet (Element : ORDERED) : SET =
struct
  type Elem = Element.T
  datatype Tree = E | T of Tree × Elem × Tree
  type Set = Tree

  val empty = E

  fun member (x, E) = false
    | member (x, T (a, y, b)) =
        if Element.lt (x, y) then member (x, a)
        else if Element.lt (y, x) then member (x, b)
        else true

  fun insert (x, E) = T (E, x, E)
    | insert (x, s as T (a, y, b)) =
        if Element.lt (x, y) then T (insert (x, a), y, b)
        else if Element.lt (y, x) then T (a, y, insert (x, b))
        else s
end
```

Figure 2.9. Implementation of binary search trees as a Standard ML functor.

subtree with the original tree—the subtree that was not on the search path. For most trees, this search path contains only a tiny fraction of the nodes in the tree. The vast majority of nodes reside in the shared subtrees.

Figure 2.9 shows how binary search trees might be implemented as a Standard ML functor. This functor takes the element type and its associated comparison functions as parameters. Because these same parameters will often be used by other functors as well (see, for example, Exercise 2.6), we package them in a structure matching the ORDERED signature.

Exercise 2.2 (Andersson [And91]) In the worst case, member performs approximately $2d$ comparisons, where d is the depth of the tree. Rewrite member to take no more than $d + 1$ comparisons by keeping track of a candidate element that *might* be equal to the query element (say, the last element for which

$<$ returned false or $\leq$ returned true) and checking for equality only when you hit the bottom of the tree.

Exercise 2.3 Inserting an existing element into a binary search tree copies the entire search path even though the copied nodes are indistinguishable from the originals. Rewrite insert using exceptions to avoid this copying. Establish only one handler per insertion rather than one handler per iteration.

Exercise 2.4 Combine the ideas of the previous two exercises to obtain a version of insert that performs no unnecessary copying and uses no more than $d + 1$ comparisons.

Exercise 2.5 Sharing can also be useful within a single object, not just between objects. For example, if the two subtrees of a given node are identical, then they can be represented by the same tree.

(a) Using this idea, write a function complete of type Elem $\times$ int $\rightarrow$ Tree where complete (*x*, *d*) creates a complete binary tree of depth *d* with *x* stored in every node. (Of course, this function makes no sense for the set abstraction, but it can be useful as an auxiliary function for other abstractions, such as bags.) This function should run in $O(d)$ time.

(b) Extend this function to create balanced trees of arbitrary size. These trees will not always be complete binary trees, but should be as balanced as possible: for any given node, the two subtrees should differ in size by at most one. This function should run in $O(\log n)$ time. (Hint: use a helper function create2 that, given a size *m*, creates a pair of trees, one of size *m* and one of size *m*+1.)

Exercise 2.6 Adapt the UnbalancedSet functor to support finite maps rather than sets. Figure 2.10 gives a minimal signature for finite maps. (Note that the NOTFOUND exception is not predefined in Standard ML—you will have to define it yourself. Although this exception could be made part of the FINITEMAP signature, with every implementation defining its own NOTFOUND exception, it is convenient for all finite maps to use the same exception.)

2.3 Chapter Notes

Myers [Mye82, Mye84] used copying and sharing to implement persistent binary search trees (in his case, AVL trees). Sarnak and Tarjan [ST86a] coined the term *path copying* for the general technique of implementing persistent

```
signature FINITEMAP =
sig
  type Key
  type α Map
  val empty : α Map
  val bind  : Key × α × α Map → α Map
  val lookup : Key × α Map → α  (* raise NOTFOUND if key is not found *)
end
```

Figure 2.10. Signature for finite maps.

data structures by copying all affected nodes. Other general techniques for implementing persistent data structures have been proposed by Driscoll, Sarnak, Sleator, and Tarjan [DSST89] and Dietz [Die89], but these techniques are not purely functional.

3

Some Familiar Data Structures in a Functional Setting

Although many imperative data structures are difficult or impossible to adapt to a functional setting, some can be adapted quite easily. In this chapter, we review three data structures that are commonly taught in an imperative setting. The first, leftist heaps, is quite simple in either setting, but the other two, binomial queues and red-black trees, have a reputation for being rather complicated because imperative implementations of these data structures often degenerate into nightmares of pointer manipulations. In contrast, functional implementations of these data structures abstract away from troublesome pointer manipulations and directly reflect the high-level ideas. A bonus of implementing these data structures functionally is that we get persistence for free.

3.1 Leftist Heaps

Sets and finite maps typically support efficient access to arbitrary elements. But sometimes we need efficient access only to the *minimum* element. A data structure supporting this kind of access is called a *priority queue* or a *heap*. To avoid confusion with FIFO queues, we use the latter name. Figure 3.1 presents a simple signature for heaps.

Remark In comparing the signature for heaps with the signature for sets (Figure 2.7), we see that in the former the ordering relation on elements is included in the signature while in the latter it is not. This discrepancy is because the ordering relation is crucial to the semantics of heaps but not to the semantics of sets. On the other hand, one could justifiably argue that an *equality* relation is crucial to the semantics of sets and should be included in the signature. ◇

Heaps are often implemented as *heap-ordered* trees, in which the element at each node is no larger than the elements at its children. Under this ordering, the minimum element in a tree is always at the root.

```
signature HEAP =
sig
  structure Elem : ORDERED

  type Heap

  val empty     : Heap
  val isEmpty   : Heap → bool

  val insert    : Elem.T × Heap → Heap
  val merge     : Heap × Heap → Heap

  val findMin   : Heap → Elem.T   (* raises EMPTY if heap is empty *)
  val deleteMin : Heap → Heap     (* raises EMPTY if heap is empty *)
end
```

Figure 3.1. Signature for heaps (priority queues).

Leftist heaps [Cra72, Knu73a] are heap-ordered binary trees that satisfy the *leftist property*: the rank of any left child is at least as large as the rank of its right sibling. The rank of a node is defined to be the length of its *right spine* (i.e., the rightmost path from the node in question to an empty node). A simple consequence of the leftist property is that the right spine of any node is always the shortest path to an empty node.

Exercise 3.1 Prove that the right spine of a leftist heap of size n contains at most $\lfloor \log(n+1) \rfloor$ elements. (All logarithms in this book are base 2 unless otherwise indicated.) ◇

Given some structure Elem of ordered elements, we represent leftist heaps as binary trees decorated with rank information.

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

Note that the elements along the right spine of a leftist heap (in fact, along any path through a heap-ordered tree) are stored in sorted order. The key insight behind leftist heaps is that two heaps can be merged by merging their right spines as you would merge two sorted lists, and then swapping the children of nodes along this path as necessary to restore the leftist property. This can be implemented as follows:

```
fun merge (h, E) = h
  | merge (E, h) = h
  | merge (h1 as T (_, x, a1, b1), h2 as T (_, y, a2, b2)) =
      if Elem.leq (x, y) then makeT (x, a1, merge (b1, h2))
      else makeT (y, a2, merge (h1, b2))
```

where makeT is a helper function that calculates the rank of a T node and swaps its children if necessary.

```
fun rank E = 0
  | rank (T (r, _, _, _)) = r
fun makeT (x, a, b) = if rank a ≥ rank b then T (rank b + 1, x, a, b)
                      else T (rank a + 1, x, b, a)
```

Because the length of each right spine is at most logarithmic, merge runs in $O(\log n)$ time.

Now that we have an efficient merge function, the remaining functions are trivial: insert creates a new singleton tree and merges it with the existing heap, findMin returns the root element, and deleteMin discards the root element and merges its children.

```
fun insert (x, h) = merge (T (1, x, E, E), h)
fun findMin (T (_, x, a, b)) = x
fun deleteMin (T (_, x, a, b)) = merge (a, b)
```

Since merge takes $O(\log n)$ time, so do insert and deleteMin. findMin clearly runs in $O(1)$ time. The complete implementation of leftist heaps is given in Figure 3.2 as a functor that takes the structure of ordered elements as a parameter.

Remark To avoid cluttering our examples with minor details, we usually ignore error cases when presenting code fragments. For example, the above code fragments do not describe the behavior of findMin or deleteMin on empty heaps. We always include the error cases when presenting complete implementations, as in Figure 3.2.

Exercise 3.2 Define insert directly rather than via a call to merge.

Exercise 3.3 Implement a function fromList of type Elem.T list → Heap that produces a leftist heap from an unordered list of elements by first converting each element into a singleton heap and then merging the heaps until only one heap remains. Instead of merging the heaps in one right-to-left or left-to-right pass using foldr or foldl, merge the heaps in $\lceil \log n \rceil$ passes, where each pass merges adjacent pairs of heaps. Show that fromList takes only $O(n)$ time.

Exercise 3.4 (Cho and Sahni [CS96]) Weight-biased leftist heaps are an alternative to leftist heaps that replace the leftist property with the *weight-biased leftist property*: the size of any left child is at least as large as the size of its right sibling.

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

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

  fun rank E = 0
    | rank (T (r, _, _, _)) = r
  fun makeT (x, a, b) = if rank a ≥ rank b then T (rank b + 1, x, a, b)
                        else T (rank a + 1, x, b, a)

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

  fun merge (h, E) = h
    | merge (E, h) = h
    | merge (h1 as T (_, x, a1, b1), h2 as T (_, y, a2, b2)) =
        if Elem.leq (x, y) then makeT (x, a1, merge (b1, h2))
        else makeT (y, a2, merge (h1, b2))
  fun insert (x, h) = merge (T (1, x, E, E), h)

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

Figure 3.2. Leftist heaps.

(a) Prove that the right spine of a weight-biased leftist heap contains at most $\lfloor \log(n+1) \rfloor$ elements.

(b) Modify the implementation in Figure 3.2 to obtain weight-biased leftist heaps.

(c) Currently, merge operates in two passes: a top-down pass consisting of calls to merge, and a bottom-up pass consisting of calls to the helper function makeT. Modify merge for weight-biased leftist heaps to operate in a single, top-down pass.

(d) What advantages would the top-down version of merge have in a lazy environment? In a concurrent environment?

3.2 Binomial Heaps

Another common implementation of heaps is binomial queues [Vui78, Bro78], which we call *binomial heaps* to avoid confusion with FIFO queues. Binomial heaps are more complicated than leftist heaps, and at first appear to offer no compensatory advantages. However, in later chapters, we will see ways in

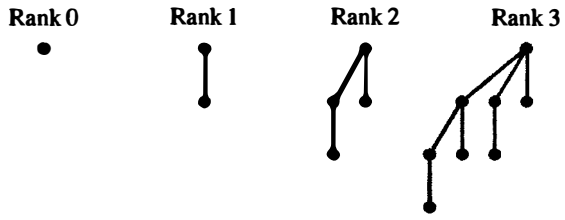


Figure 3.3. Binomial trees of ranks 0–3.

which insert and merge can be made to run in $O(1)$ time for various flavors of binomial heaps.

Binomial heaps are composed of more primitive objects known as binomial trees. Binomial trees are inductively defined as follows:

- A binomial tree of rank 0 is a singleton node.
- A binomial tree of rank $r+1$ is formed by *linking* two binomial trees of rank r, making one tree the leftmost child of the other.

From this definition, it is easy to see that a binomial tree of rank r contains exactly 2^r nodes. There is a second, equivalent definition of binomial trees that is sometimes more convenient: a binomial tree of rank r is a node with r children $t_1 \ldots t_r$, where each t_i is a binomial tree of rank $r-i$. Figure 3.3 illustrates binomial trees of ranks 0 through 3.

We represent a node in a binomial tree as an element and a list of children. For convenience, we also annotate each node with its rank.

```
datatype Tree = Node of int × Elem.T × Tree list
```

Each list of children is maintained in decreasing order of rank, and elements are stored in heap order. We maintain heap order by always linking trees with larger roots under trees with smaller roots.

```
fun link (t1 as Node (r, x1, c1), t2 as Node (_, x2, c2)) =
      if Elem.leq (x1, x2) then Node (r+1, x1, t2 :: c1)
      else Node (r+1, x2, t1 :: c2)
```

We always link trees of equal rank.

Now, a binomial heap is a collection of heap-ordered binomial trees in which no two trees have the same rank. This collection is represented as a list of trees in increasing order of rank.

```
type Heap = Tree list
```

Because each binomial tree contains 2^r elements and no two trees have the same rank, the trees in a binomial heap of size n correspond exactly to the ones in the binary representation of n. For example, the binary representation of 21 is `10101` so a binomial heap of size 21 would contain one tree of rank 0, one tree of rank 2, and one tree of rank 4 (of sizes 1, 4, and 16, respectively). Note that, just as the binary representation of n contains at most $\lfloor \log(n+1) \rfloor$ ones, a binomial heap of size n contains at most $\lfloor \log(n+1) \rfloor$ trees.

We are now ready to describe the functions on binomial heaps. We begin with insert and merge, which are defined in loose analogy to incrementing or adding binary numbers. (We will tighten this analogy in Chapter 9.) To insert a new element into a heap, we first create a new singleton tree (i.e., a binomial tree of rank 0). We then step through the existing trees in increasing order of rank until we find a missing rank, linking trees of equal rank as we go. Each link corresponds to a carry in binary arithmetic.

```
fun rank (Node (r, x, c)) = r
fun insTree (t, []) = [t]
  | insTree (t, ts as t' :: ts') =
      if rank t < rank t' then t :: ts else insTree (link (t, t'), ts')
fun insert (x, ts) = insTree (Node (0, x, []), ts)
```

The worst case is insertion into a heap of size $n = 2^k - 1$, requiring a total of k links and $O(k) = O(\log n)$ time.

To merge two heaps, we step through both lists of trees in increasing order of rank, linking trees of equal rank as we go. Again, each link corresponds to a carry in binary arithmetic.

```
fun merge (ts1, []) = ts1
  | merge ([], ts2) = ts2
  | merge (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
      if rank t1 < rank t2 then t1 :: merge (ts1', ts2)
      else if rank t2 < rank t1 then t2 :: merge (ts1, ts2')
      else insTree (link (t1, t2), merge (ts1', ts2'))
```

Both findMin and deleteMin call an auxiliary function removeMinTree that finds the tree with the minimum root and removes it from the list, returning both the tree and the remaining list.

```
fun removeMinTree [t] = (t, [])
  | removeMinTree (t :: ts) =
      let val (t', ts') = removeMinTree ts
      in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
```

Now, findMin simply returns the root of the extracted tree.

```
fun findMin ts = let val (t, _) = removeMinTree ts in root t end
```

The deleteMin function is a little trickier. After discarding the root of the extracted tree, we must somehow return the children of the discarded node to the remaining list of trees. Note that each list of children is *almost* a valid binomial heap. Each is a collection of heap-ordered binomial trees of unique rank, but in decreasing rather than increasing order of rank. Thus, we convert the list of children into a valid binomial heap by reversing it and then merge this list with the remaining trees.

```
fun deleteMin ts = let val (Node (_, x, ts1), ts2) = removeMinTree ts
                   in merge (rev ts1, ts2) end
```

The complete implementation of binomial heaps is shown in Figure 3.4. All four major operations require $O(\log n)$ time in the worst case.

Exercise 3.5 Define findMin directly rather than via a call to removeMinTree.

Exercise 3.6 Most of the rank annotations in this representation of binomial heaps are redundant because we know that the children of a node of rank r have ranks $r-1, \ldots, 0$. Thus, we can remove the rank annotations from each node and instead pair each tree at the top-level with its rank, i.e.,

```
datatype Tree = Node of k
```

...

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

  datatype Tree = Node of int × Elem.T × Tree list
  type Heap = Tree list

  val empty = [ ]
  fun isEmpty ts = null ts

  fun rank (Node (r, x, c)) = r
  fun root (Node (r, x, c)) = x
  fun link (t1 as Node (r, x1, c1), t2 as Node (_, x2, c2)) =
        if Elem.leq (x1, x2) then Node (r+1, x1, t2 :: c1)
        else Node (r+1, x2, t1 :: c2)
  fun insTree (t, [ ]) = [t]
    | insTree (t, ts as t' :: ts') =
        if rank t < rank t' then t :: ts else insTree (link (t, t'), ts')

  fun insert (x, ts) = insTree (Node (0, x, [ ]), ts)
  fun merge (ts1, [ ]) = ts1
    | merge ([ ], ts2) = ts2
    | merge (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
        if rank t1 < rank t2 then t1 :: merge (ts1', ts2)
        else if rank t2 < rank t1 then t2 :: merge (ts1, ts2')
        else insTree (link (t1, t2), merge (ts1', ts2'))

  fun removeMinTree [ ] = raise EMPTY
    | removeMinTree [t] = (t, [ ])
    | removeMinTree (t :: ts) =
        let val (t', ts') = removeMinTree ts
        in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end

  fun findMin ts = let val (t, _) = removeMinTree ts in root t end
  fun deleteMin ts =
        let val (Node (_, x, ts1), ts2) = removeMinTree ts
        in merge (rev ts1, ts2) end
end
```

Figure 3.4. Binomial heaps.

3.3 Red-Black Trees

In Section 2.2, we introduced binary search trees. Although these trees work very well on random or unordered data, they perform very poorly on ordered data, for which any individual operation might take up to $O(n)$ time. The solution to this problem is to keep each tree approximately balanced. Then no individual operation takes more than $O(\log n)$ time. Red-black trees [GS78] are one of the most popular families of balanced binary search trees.

A red-black tree is a binary search tree in which every node is colored either

red or black. We augment the type of binary search trees from Section 2.2 with a color field.

```
datatype Color = R | B
datatype Tree = E | T of Color × Tree × Elem × Tree
```

All empty nodes are considered to be black, so the empty constructor E does not need a color field.

We insist that every red-black tree satisfy the following two balance invariants:

Invariant 1. No red node has a red child.

Invariant 2. Every path from the root to an empty node contains the same number of black nodes.

Taken together, these two invariants guarantee that the longest possible path in a red-black tree, one with alternating black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only.

Exercise 3.8 Prove that the maximum depth of a node in a red-black tree of size n is at most $2\lfloor \log(n+1) \rfloor$. ◇

The member function on red-black trees ignores the color fields. Except for a wildcard in the T case, it is identical to the member function on unbalanced search trees.

```
fun member (x, E) = false
  | member (x, T (_, a, y, b)) =
      if x < y then member (x, a)
      else if x > y then member (x, b)
      else true
```

The insert function is more interesting because it must maintain the two balance invariants.

```
fun insert (x, s) =
    let fun ins E = T (R, E, x, E)
          | ins (s as T (color, a, y, b)) =
              if x < y then balance (color, ins a, y, b)
              else if x > y then balance (color, a, y, ins b)
              else s
        val T (_, a, y, b) = ins s   (* guaranteed to be non-empty *)
    in T (B, a, y, b) end
```

This function extends the insert function for unbalanced search trees in three significant ways. First, when we create a new node in the ins E case, we initially color it red. Second, we force the final root to be black, regardless of the color returned by ins. Finally, we replace the calls to the T constructor in the $x < y$ and $x > y$ cases with calls to the balance function. The balance

function acts just like the T constructor except that it massages its arguments as necessary to enforce the balance invariants.

Coloring the new node red maintains Invariant 2, but violates Invariant 1 whenever the parent of the new node is red. We allow a single red-red violation at a time, and percolate this violation up the search path toward the root during rebalancing. The balance function detects and repairs each red-red violation when it processes the black parent of the red node with a red child. This black-red-red path can occur in any of four configurations, depending on whether each red node is a left or right child. However, the solution is the same in every case: rewrite the black-red-red path as a red node with two black children, as illustrated in Figure 3.5. This transformation can be coded as follows:

```
fun balance (B,T (R,T (R,a,x,b),y,c),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,T (R,a,x,T (R,b,y,c)),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,a,x,T (R,T (R,b,y,c),z,d)) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,a,x,T (R,b,y,T (R,c,z,d))) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance body = T body
```

It is routine to verify that the red-black balance invariants both hold for the resulting (sub)tree.

Remark Notice that the right-hand sides of the first four clauses are identical. Some implementations of Standard ML, notably Standard ML of New Jersey, support a feature known as *or-patterns* that allows multiple clauses with identical right-hand sides to be collapsed into a single clause [FB97]. Using or-patterns, the balance function might be rewritten

```
fun balance ( (B,T (R,T (R,a,x,b),y,c),z,d)
            | (B,T (R,a,x,T (R,b,y,c)),z,d)
            | (B,a,x,T (R,T (R,b,y,c),z,d))
            | (B,a,x,T (R,b,y,T (R,c,z,d))) ) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance body = T body
```

◇

After balancing a given subtree, the red root of that subtree might now be the child of another red node. Thus, we continue balancing all the way to the top of the tree. At the very top of the tree, we might end up with a red node with a red child, but with no black parent. We handle this case by always recoloring the root to be black.

This implementation of red-black trees is summarized in Figure 3.6.

Hint to Practitioners: Even without optimization, this implementation of balanced binary search trees is one of the fastest around. With appropriate optimizations, such as Exercises 2.2 and 3.10, it really flies!

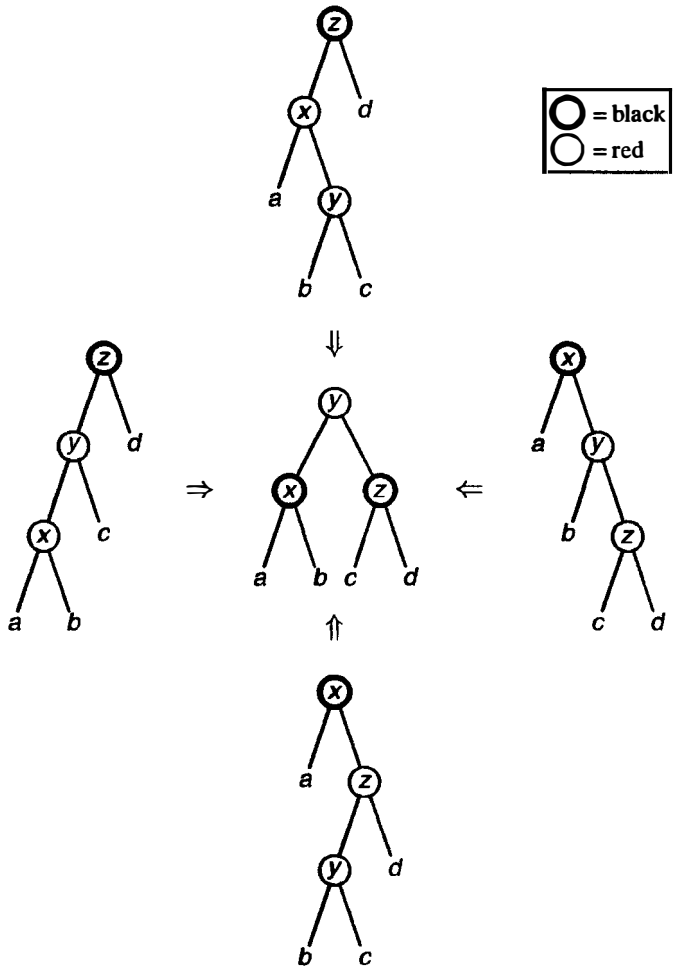


Figure 3.5. Eliminating red nodes with red parents.

Remark One of the reasons this implementation is so much simpler than typical presentations of red-black trees (e.g., Chapter 14 of [CLR90]) is that it uses subtly different rebalancing transformations. Imperative implementations typically split the four dangerous cases considered here into eight cases, according to the color of the sibling of the red node with a red child. Knowing the color of the red parent's sibling allows the transformations to use fewer assignments in some cases and to terminate rebalancing early in others. However, in a func-

```
functor RedBlackSet (Element : ORDERED) : SET =
struct
  type Elem = Element.T

  datatype Color = R | B
  datatype Tree = E | T of Color × Tree × Elem × Tree
  type Set = Tree

  val empty = E

  fun member (x, E) = false
    | member (x, T (_, a, y, b)) =
        if Element.lt (x, y) then member (x, a)
        else if Element.lt (y, x) then member (x, b)
        else true

  fun balance (B,T (R,T (R,a,x,b),y,c),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance (B,T (R,a,x,T (R,b,y,c)),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance (B,a,x,T (R,T (R,b,y,c),z,d)) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance (B,a,x,T (R,b,y,T (R,c,z,d))) = T (R,T (B,a,x,b),y,T (B,c,z,d))
    | balance body = T body

  fun insert (x, s) =
        letfun ins E = T (R, E, x, E)
             | ins (s as T (color, a, y, b)) =
                 if Element.lt (x, y) then balance (color, ins a, y, b)
                 else if Element.lt (y, x) then balance (color, a, y, ins b)
                 else s
          val T (_, a, y, b) = ins s   (* guaranteed to be non-empty *)
        in T (B, a, y, b) end
end
```

Figure 3.6. Red black trees.

tional setting, where we are copying the nodes in question anyway, we cannot reduce the number of assignments in this fashion, nor can we terminate copying early, so there is no point is using the more complicated transformations.

Exercise 3.9 Write a function fromOrdList of type Elem list → Tree that converts a sorted list with no duplicates into a red-black tree. Your function should run in $O(n)$ time.

Exercise 3.10 The balance function currently performs several unnecessary tests. For example, when the ins function recurses on the left child, there is no need for balance to test for red-red violations involving the right child.

(a) Split balance into two functions, lbalance and rbalance, that test for vio-

lations involving the left child and right child, respectively. Replace the calls to balance in ins with calls to either lbalance or rbalance.

(b) Extending the same logic one step further, one of the remaining tests on the grandchildren is also unnecessary. Rewrite ins so that it never tests the color of nodes not on the search path.

3.4 Chapter Notes

Núñez, Palao, and Peña [NPP95] and King [Kin94] describe similar implementations in Haskell of leftist heaps and binomial heaps, respectively. Red-black trees have not previously appeared in the functional programming literature, but several other kinds of balanced binary search trees have, including AVL trees [Mye82, Mye84, BW88, NPP95], 2-3 trees [Rea92], and weight-balanced trees [Ada93].

Knuth [Knu73a] originally introduced leftist heaps as a simplification of a data structure by Crane [Cra72]. Vuillemin [Vui78] invented binomial heaps; Brown [Bro78] examined many of the properties of this elegant data structure. Guibas and Sedgewick [GS78] proposed red-black trees as a general framework for describing many other kinds of balanced trees.

4

Lazy Evaluation

Lazy evaluation is the default evaluation strategy of many functional programming languages (although not of Standard ML). This strategy has two essential properties. First, the evaluation of a given expression is delayed, or *suspended*, until its result is needed. Second, the first time a suspended expression is evaluated, the result is *memoized* (i.e., cached) so that, if it is ever needed again, it can be looked up rather than recomputed. Both aspects of lazy evaluation are algorithmically useful.

In this chapter, we introduce a convenient notation for lazy evaluation and illustrate this notation by developing a simple streams package. We will use both lazy evaluation and streams extensively in later chapters.

4.1 $-notation

Unfortunately, the definition of Standard ML [MTHM97] does not include support for lazy evaluation, so each compiler is free to provide its own set of lazy evaluation primitives. We present here one such set of primitives, called $-notation. Translating programs written with $-notation into other notations for lazy evaluation should be straightforward.

In $-notation, we introduce a new type α **susp** to represent suspensions. This type has a single, unary constructor called **$**. To a first approximation, α **susp** and **$** behave as if defined by the ordinary datatype declaration

datatype α **susp** = **$ of** α

We create a new suspension of type τ **susp** by writing $e, where e is an expression of type τ. Similarly, we extract the contents of an existing suspension by matching against the pattern $p. If the pattern p matches values of type τ, then $p matches suspensions of type τ **susp**.

The main difference between **$** and ordinary constructors is that **$** does not

immediately evaluate its argument. Instead, it saves whatever information it will need to resume the evaluation of the argument expression at a later time. (Typically, this information consists of a code pointer together with the values of the free variables in the expression.) The argument expression is not evaluated until and unless the suspension is matched against a pattern of the form $\$p$. At that time, the argument expression is evaluated and the result is memoized. Then the result is matched against the pattern p. If the suspension is later matched against another pattern of the form $\$p'$, the memoized value of the suspension is looked up and matched against p'.

The \$ constructor is also parsed differently from ordinary constructors. First, the scope of the \$ constructor extends as far to the right as possible. Thus, for example, the expression **\$***f x* parses as **\$(***f x***)** rather than **(\$***f***)** *x*, and the pattern **\$CONS (***x***,** *xs***)** parses as **\$(CONS (***x***,** *xs***))** rather than **(\$CONS) (***x***,** *xs***)**. Second, \$ does not constitute a valid expression by itself—it must always be accompanied by an argument.

As an example of \$-notation, consider the following program fragment:

```
val s = $primes 1000000      (* fast *)
...
val $x = s                   (* slow *)
...
val $y = s                   (* fast *)
...
```

This program computes the one millionth prime. The first line, which simply creates a new suspension, runs very quickly. The second line actually computes the prime by evaluating the suspension. Depending on the algorithm for computing primes, it might take a long time. The third line looks up the memoized value and also runs very quickly.

As a second example, consider the fragment

```
let val s = $primes 1000000
in 15 end
```

This program never demands the contents of the suspension, and so never evaluates primes 1000000.

Although we can program all the examples of lazy evaluation in this book using only \$-expressions and \$-patterns, two forms of syntactic sugar will be convenient. The first is the force operator, defined as

```
fun force ($x) = x
```

This is useful for extracting the contents of a suspension in the middle of an expression, where it might be awkward to introduce a pattern matching construct.

The second form of syntactic sugar is useful for writing certain kinds of lazy functions. For example, consider the following function for addition of suspended integers:

fun plus ($*m*, $*n*) = $*m*+*n*

Although this function seems perfectly reasonable, it is in fact not the function that we probably intended. The problem is that it forces both of its arguments too early. In particular, it forces its arguments when plus is applied, rather than when the suspension created by plus is forced. One way to get the desired behavior is to explicitly delay the pattern matching, as in

fun plus (*x*, *y*) = **$case** (*x*, *y*) **of** ($*m*, $*n*) ⇒ *m*+*n*

However, this idiom is common enough that we provide syntactic sugar for it, writing

fun lazy *f p* = *e*

instead of

fun *f x* = **$case** *x* **of** *p* ⇒ force *e*

The extra force ensures that the **lazy** keyword has no effect on the type of a function (assuming that the result was a susp type to begin with), so we can add or remove the annotation without changing the function text in any other way. Now we can write the desired function for addition of suspended integers simply as

fun lazy plus ($*m*, $*n*) = $*m*+*n*

Expanding this syntactic sugar yields

fun plus (*x*, *y*) = **$case** (*x*, *y*) **of** ($*m*, $*n*) ⇒ force ($*m*+*n*)

which is exactly same as the hand-written version above except for the extra force and $ around the *m*+*n*. This force and $ would be optimized away by a good compiler since force ($*e*) is equivalent to *e* for any *e*.

The plus function uses the **lazy** annotation to delay pattern matching, so that $-patterns are not matched prematurely. However, the **lazy** annotation is also useful when the right-hand side of the function is an expression that returns a suspension as the result of a possibly long and involved computation. Using the **lazy** annotation in this situation delays the execution of the expensive computation from the time the function is applied until the time that its resulting suspension is forced. We will see several functions in the next section that use the **lazy** annotation in this fashion.

The syntax and semantics of $-notation are formally defined in [Oka96a].

4.2 Streams

As an extended example of lazy evaluation and $-notation in Standard ML, we next develop a small streams package. These streams will be used in several of the data structures in subsequent chapters.

Streams (also known as lazy lists) are similar to ordinary lists, except that every cell is systematically suspended. The type of streams is

```
datatype α StreamCell = NIL | CONS of α × α Stream
withtype α Stream = α StreamCell susp
```

A simple stream containing the elements 1, 2, and 3 could be written

```
$CONS (1, $CONS (2, $CONS (3, $NIL)))
```

It is illuminating to contrast streams with suspended lists of type α list susp. The computations represented by the latter type are inherently *monolithic*—once begun by forcing the suspended list, they run to completion. The computations represented by streams, on the other hand, are often *incremental*—forcing a stream executes only enough of the computation to produce the outermost cell and suspends the rest. This behavior is common among datatypes such as streams that contain nested suspensions.

To see this difference in behavior more clearly, consider the append function, written $s +\!\!+ t$. On suspended lists, this function might be written

```
fun s ++ t = $(force s @ force t)
```

or, equivalently,

```
fun lazy ($xs) ++ ($ys) = $(xs @ ys)
```

The suspension produced by this function forces both arguments and then appends the two lists, producing the entire result. Hence, this suspension is monolithic. We also say that the function is monolithic. On streams, the append function is written

```
fun lazy ($NIL) ++ t = t
       | ($CONS (x, s)) ++ t = $CONS (x, s ++ t)
```

This function immediately returns a suspension, which, when forced, demands the first cell of the left stream by matching against a $-pattern. If this cell is a CONS, then we construct the result from x and $s +\!\!+ t$. Because of the **lazy** annotation, the recursive call simply creates another suspension, without doing any extra work. Hence, the computation described by this function is

incremental; it produces the first cell of the result and delays the rest. We also say that the function is incremental.

Another incremental function is take, which extracts the first n elements of a stream.

```
fun lazy take (0, s) = $NIL
        | take (n, $NIL) = $NIL
        | take (n, $CONS (x, s)) = $CONS (x, take (n-1, s))
```

As with ++, the recursive call to take $(n-1, s)$ immediately returns a suspension, rather than executing the rest of the function.

However, consider the function to delete the first n elements of a stream, which could be written

```
fun lazy drop (0, s) = s
        | drop (n, $NIL) = $NIL
        | drop (n, $CONS (x, s)) = drop (n-1, s)
```

or more efficiently as

```
fun lazy drop (n, s) = let fun drop' (0, s) = s
                           | drop' (n, $NIL) = $NIL
                           | drop' (n, $CONS (x, s)) = drop' (n-1, s)
                       in drop' (n, s) end
```

This function is monolithic because the recursive calls to drop′ are never delayed—calculating the first cell of the result requires executing the entire function. Here we use the **lazy** annotation to delay the initial call to drop′ rather than to delay pattern matching.

Exercise 4.1 Use the fact that force ($\$e$) is equivalent to e to show that these two definitions of drop are equivalent. ◇

Another common monolithic stream function is reverse.

```
fun lazy reverse s =
    let fun reverse' ($NIL, r) = r
          | reverse' ($CONS (x, s), r) = reverse' (s, $CONS (x, r))
    in reverse' (s, $NIL) end
```

Here the recursive calls to reverse′ are never delayed, but note that each recursive call creates a new suspension of the form $\$\text{CONS}\ (x, r)$. It might seem then that reverse does not in fact do all of its work at once. However, suspensions such as these, whose bodies contain only a few constructors and variables, with no function applications, are called *trivial*. Trivial suspensions are delayed, not for any algorithmic purpose, but rather to make the types work out. We can consider the body of a trivial suspension to be executed at the time the

```
signature STREAM =
sig
  datatype α StreamCell = NIL | CONS of α × α Stream
  withtype α Stream = α StreamCell susp

  val ++      : α Stream × α Stream → α Stream   (* stream append *)
  val take    : int × α Stream → α Stream
  val drop    : int × α Stream → α Stream
  val reverse : α Stream → α Stream
end

structure Stream : STREAM =
struct
  datatype α StreamCell = NIL | CONS of α × α Stream
  withtype α Stream = α StreamCell susp

  fun lazy ($NIL) ++ t = t
         | ($CONS (x, s)) ++ t = $CONS (x, s ++ t)
  fun lazy take (0, s) = $NIL
         | take (n, $NIL) = $NIL
         | take (n, $CONS (x, s)) = $CONS (x, take (n-1, s))
  fun lazy drop (n, s) =
      let fun drop' (0, s) = s
            | drop' (n, $NIL) = $NIL
            | drop' (n, $CONS (x, s)) = drop' (n-1, s)
      in drop' (n, s) end
  fun lazy reverse s =
      let fun reverse' ($NIL, r) = r
            | reverse' ($CONS (x, s), r) = reverse' (s, $CONS (x, r))
      in reverse' (s, $NIL) end
end
```

Figure 4.1. A small streams package.

suspension is created. In fact, a reasonable compiler optimization is to create such suspensions in already-memoized form. Either way, forcing a trivial suspension never takes more than $O(1)$ time.

Although monolithic stream functions such as drop and reverse are common, incremental functions such as ++ and take are the *raison d'être* of streams. Each suspension carries a small but significant overhead, so for maximum efficiency laziness should be used only when there is a good reason to do so. If the only uses of lazy lists in a given application are monolithic, then that application should use simple suspended lists rather than streams.

Figure 4.1 summarizes these stream functions as a Standard ML module. Note that this module does not export functions such as isEmpty and cons, as one might expect. Instead, we deliberately expose the internal representation in order to support pattern matching on streams.

Exercise 4.2 Implement insertion sort on streams. Show that extracting the first k elements of sort *xs* takes only $O(n \cdot k)$ time, where n is the length of *xs*, rather than $O(n^2)$ time, as might be expected of insertion sort.

4.3 Chapter Notes

Lazy Evaluation Wadsworth [Wad71] introduced lazy evaluation as an optimization of normal-order reduction in the lambda calculus. Vuillemin [Vui74] later showed that, under certain restricted conditions, lazy evaluation is an optimal evaluation strategy. The formal semantics of lazy evaluation has been studied extensively [Jos89, Lau93, OLT94, AFM+95].

Streams Landin introduced streams in [Lan65], but without memoization. Friedman and Wise [FW76] and Henderson and Morris [HM76] extended Landin's streams with memoization.

Memoization Michie [Mic68] coined the term memoization to denote the augmentation of functions with a cache of argument–result pairs. The argument field is dropped when memoizing suspensions by regarding suspensions as nullary functions—that is, functions with zero arguments. Hughes [Hug85] later applied memoization, in the original sense of Michie, to functional programs.

Algorithmics Both components of lazy evaluation—delaying computations and memoizing the results—have a long history in algorithm design, although not always in combination. The idea of delaying the execution of potentially expensive computations (often deletions) is used to good effect in hash tables [WV86], priority queues [ST86b, FT87], and search trees [DSST89]. Memoization, on the other hand, is the basic principle of such techniques as dynamic programming [Bel57] and path compression [HU73, TvL84].

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.,

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

where c_i is the number of credits allocated by operation i and $\bar{c}_i$ 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, $\sum c_i \geq \sum \bar{c}_i$, 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

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

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. ◇

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.

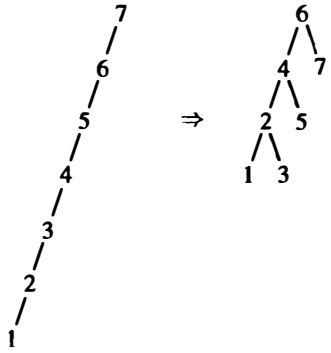


Figure 5.4. Calling bigger with a pivot element of 0.

```
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*.

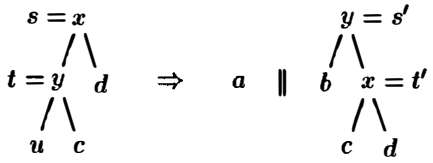
Theorem 5.2 $\mathcal{A}(t) \leq 1 + 2\phi(t) = 1 + 2\log(\#t)$.

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,

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

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.

6

Amortization and Persistence via Lazy Evaluation

The previous chapter introduced the idea of amortization and gave several examples of data structures with good amortized bounds. However, for each these data structures, the amortized bounds break in the presence of persistence. In this chapter, we demonstrate how lazy evaluation can mediate the conflict between amortization and persistence, and adapt both the banker's and physicist's methods to account for lazy evaluation. We then illustrate the use of these new methods on several amortized data structures that use lazy evaluation internally.

6.1 Execution Traces and Logical Time

In the previous chapter, we saw that traditional methods of amortization break in the presence of persistence because they assume a unique future, in which the accumulated savings will be spent at most once. However, with persistence, multiple logical futures might all try to spend the same savings. But what exactly do we mean by the "logical future" of an operation?

We model logical time with *execution traces*, which give an abstract view of the history of a computation. An execution trace is a directed graph whose nodes represent operations of interest, usually just update operations on the data type in question. An edge from v to v' indicates that operation v' uses some result of operation v. The *logical history* of operation v, denoted $\hat{v}$, is the set of all operations on which the result of v depends (including v itself). In other words, $\hat{v}$ is the set of all nodes w such that there exists a path (possibly of length 0) from w to v. A *logical future* of a node v is any path from v to a terminal node (i.e., a node with out-degree zero). If there is more than one such path, then node v has multiple logical futures. We sometimes refer to the logical history or logical future of an object, meaning the logical history or logical future of the operation that created the object.

Exercise 6.1 Draw an execution trace for the following set of operations. Annotate each node in the trace with the number of logical futures at that node.

```
val a = snoc (empty, 0)
val b = snoc (a, 1)
val c = tail b
val d = snoc (b, 2)
val e = c ++ d
val f = tail c
val g = snoc (d, 3)
```

◇

Execution traces generalize the notion of *version graphs* [DSST89], which are often used to model the histories of persistent data structures. In a version graph, nodes represent the various versions of a single persistent identity and edges represent dependencies between versions. Thus, version graphs model the results of operations and execution traces model the operations themselves. Execution traces are often more convenient for combining the histories of several persistent identities (perhaps not even of the same type) or for reasoning about operations that do not return a new version (e.g., queries) or that return several results (e.g., splitting a list into two sublists).

For ephemeral data structures, the out-degree of every node in a version graph or execution trace is typically restricted to be at most one, reflecting the limitation that objects can be updated at most once. To model various flavors of persistence, version graphs allow the out-degree of every node to be unbounded, but make other restrictions. For instance, version graphs are often limited to be trees (forests) by restricting the in-degree of every node to be at most one. Other version graphs allow in-degrees of greater than one, but forbid cycles, making every graph a dag. We make none of these restrictions on execution traces for persistent data structures. Nodes with in-degree greater than one correspond to operations that take more than one argument, such as list catenation or set union. Cycles arise from recursively defined objects, which are supported by many lazy languages. We even allow multiple edges between a single pair of nodes, as might occur if a list is catenated with itself.

We will use execution traces in Section 6.3.1 when we extend the banker's method to cope with persistence.

6.2 Reconciling Amortization and Persistence

In this section, we show how the banker's and physicist's methods can be repaired by replacing the notion of accumulated savings with accumulated debt, where debt measures the cost of unevaluated lazy computations. The intuition

is that, although savings can only be spent once, it does no harm to pay off debt more than once.

6.2.1 The Role of Lazy Evaluation

Recall that an *expensive* operation is one whose actual costs are greater than its (desired) amortized costs. For example, suppose some application $f\ x$ is expensive. With persistence, a malicious adversary might call $f\ x$ arbitrarily often. (Note that each operation is a new logical future of x.) If each operation takes the same amount of time, then the amortized bounds degrade to the worst-case bounds. Hence, we must find a way to guarantee that if the first application of f to x is expensive, then subsequent applications of f to x will not be.

Without side-effects, this is impossible under call-by-value (i.e., strict evaluation) or call-by-name (i.e., lazy evaluation without memoization), because every application of f to x takes exactly the same amount of time. Therefore, amortization cannot be usefully combined with persistence in languages supporting only these evaluation orders.

But now consider call-by-need (i.e., lazy evaluation with memoization). If x contains some suspended component that is needed by f, then the first application of f to x forces the (potentially expensive) evaluation of that component and memoizes the result. Subsequent operations may then access the memoized result directly. This is exactly the desired behavior!

Remark In retrospect, the relationship between lazy evaluation and amortization is not surprising. Lazy evaluation can be viewed as a form of self-modification, and amortization often involves self-modification [ST85, ST86b]. However, lazy evaluation is a particularly disciplined form of self-modification — not all forms of self-modification typically used in amortized ephemeral data structures can be encoded as lazy evaluation. In particular, splay trees do not appear to be amenable to this technique.

6.2.2 A Framework for Analyzing Lazy Data Structures

We have just shown that lazy evaluation is necessary to implement amortized data structures purely functionally. Unfortunately, analyzing the running times of programs involving lazy evaluation is notoriously difficult. Historically, the most common technique for analyzing lazy programs has been to pretend that they are actually strict. However, this technique is completely inadequate

for analyzing lazy amortized data structures. We next describe a basic framework to support such analyses. In the remainder of this chapter, we adapt the banker's and physicist's methods to this framework, yielding both the first techniques for analyzing persistent amortized data structures and the first practical techniques for analyzing non-trivial lazy programs.

We classify the costs of any given operation into several categories. First, the *unshared cost* of an operation is the actual time it would take to execute the operation under the assumption that every suspension in the system at the beginning of the operation has already been forced and memoized (i.e., under the assumption that force always takes $O(1)$ time, except for those suspensions that are created and forced within the same operation). The *shared cost* of an operation is the time that it would take to execute every suspension created but not evaluated by the operation (under the same assumption as above). The *complete cost* of an operation is the sum of its shared and unshared costs. Note that the complete cost is what the actual cost of the operation would be if lazy evaluation were replaced with strict evaluation.

We further partition the total shared costs of a sequence of operations into realized and unrealized costs. ***Realized costs*** are the shared costs for suspensions that are executed during the overall computation. *Unrealized costs* are the shared costs for suspensions that are never executed. The *total actual cost* of a sequence of operations is the sum of the unshared costs and the realized shared costs—unrealized costs do not contribute to the actual cost. Note that the amount that any particular operation contributes to the total actual cost is at least its unshared cost, and at most its complete cost, depending on how much of its shared cost is realized.

We account for shared costs using the notion of *accumulated debt*. Initially, the accumulated debt is zero, but every time a suspension is created, we increase the accumulated debt by the shared cost of the suspension (and any nested suspensions). Each operation then pays off a portion of the accumulated debt. The *amortized cost* of an operation is the unshared cost of the operation plus the amount of accumulated debt paid off by the operation. We are not allowed to force a suspension until the debt associated with the suspension is entirely paid off.

Remark An amortized analysis based on the notion of accumulated debt works a lot like a *layaway plan*. In a layaway plan, you find something—a diamond ring, say—that you want to buy, but that you can't afford to pay for yet. You agree on a price with the jewelry store and ask them to set the ring aside in your name. You then make regular payments, and receive the ring only when it is entirely paid off.

In analyzing a lazy data structure, you find a computation that you can't afford to execute yet. You create a suspension for the computation, and assign the suspension an amount of debt proportional to its shared cost. You then pay off the debt a little at a time. Finally, when the debt is entirely paid off, you are allowed to execute the suspension. ◇

There are three important moments in the life cycle of a suspension: when it is created, when it is entirely paid off, and when it is executed. The proof obligation is to show that the second moment precedes the third. If every suspension is paid off before it is forced, then the total amount of debt that has been paid off is an upper bound on the realized shared costs, and therefore the total amortized cost (i.e., the total unshared cost plus the total amount of debt that has been paid off) is an upper bound on the total actual cost (i.e., the total unshared cost plus the realized shared costs). We will formalize this argument in Section 6.3.1.

One of the most difficult problems in analyzing the running time of lazy programs is reasoning about the interactions of multiple logical futures. We avoid this problem by reasoning about each logical future *as if it were the only one*. From the point of view of the operation that creates a suspension, any logical future that forces the suspension must itself pay for the suspension. If two logical futures wish to force the same suspension, then both must pay for the suspension individually—they may not cooperate and each pay only a portion of the debt. An alternative view of this restriction is that we are allowed to force a suspension *only when the debt for that suspension has been paid off within the logical history of the current operation*. Using this method, we sometimes pay off a debt more than once, thereby overestimating the total time required for a particular computation, but this does no harm and is a small price to pay for the simplicity of the resulting analyses.

6.3 The Banker's Method

We adapt the banker's method to account for accumulated debt rather than accumulated savings by replacing credits with debits. Each debit represents a constant amount of suspended work. When we initially suspend a given computation, we create a number of debits proportional to its shared cost and associate each debit with a location in the object. The choice of location for each debit depends on the nature of the computation. If the computation is *monolithic* (i.e., once begun, it runs to completion), then all debits are usually assigned to the root of the result. On the other hand, if the computation is *incremental* (i.e., decomposable into fragments that may be executed inde-

Most books on data structures assume an imperative language like C or C++. However, data structures for these languages do not always translate well to functional languages such as Standard ML, Haskell, or Scheme. This book describes data structures from the point of view of functional languages, with examples, and presents design techniques so that programmers can develop their own functional data structures. It includes both classical data structures, such as red-black trees and binomial queues, and a host of new data structures developed exclusively for functional languages. All source code is given in Standard ML and Haskell, and most of the programs can easily be adapted to other functional languages.

This handy reference for professional programmers working with functional languages can also be used as a tutorial or for self-study.

CAMBRIDGE
UNIVERSITY PRESS