Purely Functional Data Structures

Chris Okasaki

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

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

9 - Numerical Representations pp. 115-140

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.010

Cambridge University Press

9

Numerical Representations

Consider the usual representations of lists and natural numbers, along with several typical functions on each data type.

```
datatype α List =
    NIL
  | CONS of α × α List

fun tail (CONS (x, xs)) = xs

fun append (NIL, ys) = ys
  | append (CONS (x, xs), ys) =
      CONS (x, append (xs, ys))
```

```
datatype Nat =
    ZERO
  | SUCC of Nat

fun pred (SUCC n) = n

fun plus (ZERO, n) = n
  | plus (SUCC m, n) =
      SUCC (plus (m, n))
```

Other than the fact that lists contain elements and natural numbers do not, these implementations are virtually identical. Binomial heaps exhibit a similar relationship with binary numbers. These examples suggest a strong analogy between representations of the number n and representations of container objects of size n. Functions on the container strongly resemble arithmetic functions on the number. For example, inserting an element resembles incrementing a number, deleting an element resembles decrementing a number, and combining two containers resembles adding two numbers. This analogy can be exploited to design new implementations of container abstractions — simply choose a representation of natural numbers with certain desired properties and define the functions on the container objects accordingly. Call an implementation designed in this fashion a *numerical representation.*

In this chapter, we explore a host of numerical representations for two different abstractions: *heaps* and *random-access lists* (also known as *flexible arrays*). These abstractions stress different sets of arithmetic operations. Heaps require efficient increment and addition functions, whereas random-access lists require efficient increment and decrement functions.

9.1 Positional Number Systems

A *positional number system* [Knu73b] is a notation for writing a number as a sequence of digits $b_0 \ldots b_{m-1}$. The digit b_0 is called the *least significant digit* and the digit b_{m-1} is called the *most significant digit*. Except when writing ordinary, decimal numbers, we will always write sequences of digits from least significant to most significant.

Each digit b_i has weight w_i, so the value of the sequence $b_0 \ldots b_{m-1}$ is $\sum_{i=0}^{m-1} b_i w_i$. For any given positional number system, the sequence of weights is fixed, as is the set of digits D_i from which each b_i is chosen. For unary numbers, $w_i = 1$ and $D_i = \{1\}$ for all i, and for binary numbers $w_i = 2^i$ and $D_i = \{0, 1\}$. (By convention, we write all digits in typewriter font except for ordinary, decimal digits.) A number is said to be written in base B if $w_i = B^i$ and $D_i = \{0, \ldots, B-1\}$. Usually, but not always, weights are increasing sequences of powers, and the set D_i is the same for every digit.

A number system is said to be *redundant* if there is more than one way to represent some numbers. For example, we can obtain a redundant system of binary numbers by taking $w_i = 2^i$ and $D_i = \{0, 1, 2\}$. Then the decimal number 13 can be written `1011`, or `1201`, or `122`. By convention, we disallow trailing zeros, since otherwise almost all number systems are trivially redundant.

Computer representations of positional number systems can be *dense* or *sparse*. A dense representation is simply a list (or some other kind of sequence) of digits, including those digits that happen to be zero. A sparse representation, on the other hand, elides the zeros. It must then include information on either the rank (i.e., the index) or the weight of each non-zero digit. Figure 9.1 shows two different representations of binary numbers in Standard ML—one dense and one sparse—along with increment, decrement, and addition functions on each. Among the numerical representations that we have already seen, scheduled binomial heaps (Section 7.3) use a dense representation, while binomial heaps (Section 3.2) and lazy binomial heaps (Section 6.4.1) use sparse representations.

9.2 Binary Numbers

Given a positional number system, we can implement a numerical representation based on that number system as a sequence of trees. The number and sizes of the trees representing a collection of size n are governed by the representation of n in the positional number system. For each weight w_i, there are b_i trees of that size. For example, the binary representation of 73 is `1001001`,

```
structure Dense =
struct
  datatype Digit = ZERO | ONE
  type Nat = Digit list    (* increasing order of significance *)

  fun inc [] = [ONE]
    | inc (ZERO :: ds) = ONE :: ds
    | inc (ONE :: ds) = ZERO :: inc ds          (* carry *)

  fun dec [ONE] = []
    | dec (ONE :: ds) = ZERO :: ds
    | dec (ZERO :: ds) = ONE :: dec ds          (* borrow *)

  fun add (ds, []) = ds
    | add ([], ds) = ds
    | add (d :: ds1, ZERO :: ds2) = d :: add (ds1, ds2)
    | add (ZERO :: ds1, d :: ds2) = d :: add (ds1, ds2)
    | add (ONE :: ds1, ONE :: ds2) =
        ZERO :: inc (add (ds1, ds2))            (* carry *)
end
```

```
structure SparseByWeight =
struct
  type Nat = int list    (* increasing list of weights, each a power of two *)

  fun carry (w, []) = [w]
    | carry (w, ws as w' :: ws') =
        if w < w' then w :: ws else carry (2*w, ws')

  fun borrow (w, ws as w' :: ws') =
        if w = w' then ws' else w :: borrow (2*w, ws)

  fun inc ws = carry (1, ws)
  fun dec ws = borrow (1, ws)

  fun add (ws, []) = ws
    | add ([], ws) = ws
    | add (m as w1 :: ws1, n as w2 :: ws2) =
        if w1 < w2 then w1 :: add (ws1, n)
        else if w2 < w1 then w2 :: add (m, ws2)
        else carry (2*w1, add (ws1, ws2))
end
```

Figure 9.1. Two implementations of binary numbers.

so a collection of size 73 in a binary numerical representation would contain three trees, of sizes 1, 8, and 64, respectively.

Trees in numerical representations typically exhibit a very regular structure. For example, in binary numerical representations, all trees have sizes that are

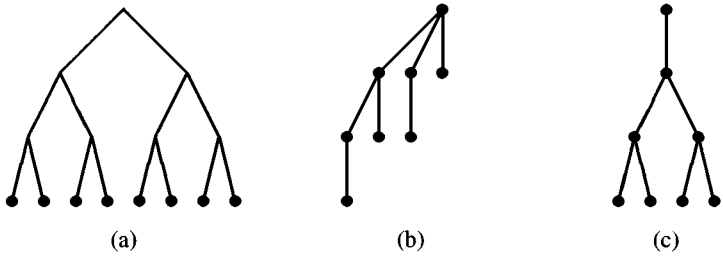

Figure 9.2. Three trees of rank 3: (a) a complete binary leaf tree, (b) a binomial tree, and (c) a pennant.

powers of two. Three common kinds of trees that exhibit this structure are *complete binary leaf trees* [KD96], *binomial trees* [Vui78], and *pennants* [SS90].

Definition 9.1 (Complete binary leaf trees) A complete binary tree of rank 0 is a leaf and a complete binary tree of rank $r > 0$ is a node with two children, each of which is a complete binary tree of rank $r - 1$. A leaf tree is a tree that contains elements only at the leaves, unlike ordinary trees that contain elements at every node. A complete binary tree of rank r has $2^{r+1} - 1$ nodes, but only 2^r leaves. Hence, a complete binary leaf tree of rank r contains 2^r elements.

Definition 9.2 (Binomial trees) A binomial tree of rank r is a node with r children $c_1 \ldots c_r$, where c_i is a binomial tree of rank $r - i$. Alternatively, a binomial tree of rank $r > 0$ is a binomial tree of rank $r - 1$ to which another binomial tree of rank $r - 1$ has been added as the leftmost child. From the second definition, it is easy to see that a binomial tree of rank r contains 2^r nodes.

Definition 9.3 (Pennants) A pennant of rank 0 is a single node and a pennant of rank $r > 0$ is a node with a single child that is a complete binary tree of rank $r - 1$. The complete binary tree contains $2^r - 1$ elements, so the pennant contains 2^r elements.

Figure 9.2 illustrates the three kinds of trees. Which kind of tree is superior for a given data structure depends on the properties the data structure must maintain, such as the order in which elements should be stored in the trees. A key factor in the suitability of a particular kind of tree for a given data structure is how easily the tree supports functions analogous to carries and borrows in binary arithmetic. When simulating a carry, we *link* two trees of rank r to form a tree of rank $r + 1$. Symmetrically, when simulating a borrow, we *unlink* a

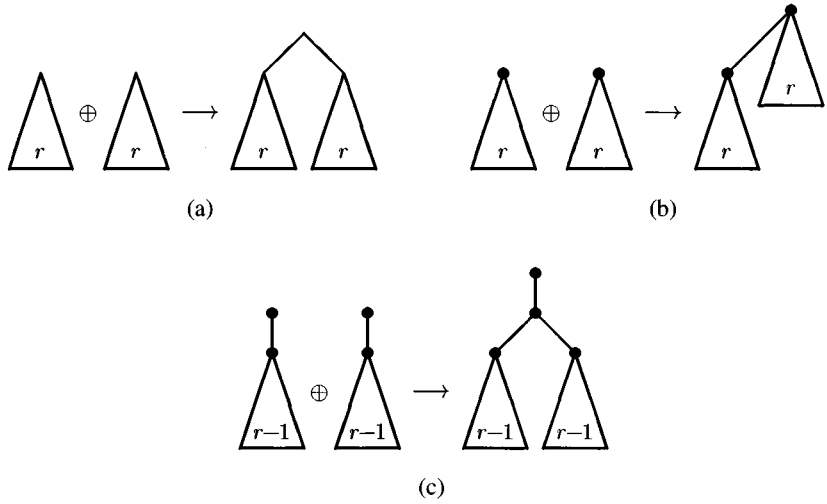

Figure 9.3. Linking two trees of rank r to obtain a tree of rank $r + 1$ for (a) complete binary leaf trees, (b) binomial trees, and (c) pennants.

tree of rank $r > 0$ to obtain two trees of rank $r - 1$. Figure 9.3 illustrates the link operation (denoted $\oplus$) on each of the three kinds of trees. Assuming that elements are not rearranged, each of the three kinds of trees can be linked or unlinked in $O(1)$ time.

We have already seen several variations of heaps based on binary arithmetic and binomial trees. We next explore a simple numerical representation for random-access lists. Then we discuss several variations of binary arithmetic that yield improved asymptotic bounds.

9.2.1 Binary Random-Access Lists

A *random-access list*, also called a one-sided flexible array, is a data structure that supports array-like lookup and update functions, as well as the usual cons, head, and tail functions on lists. A signature for random-access lists is shown in Figure 9.4.

We implement random-access lists using a binary numerical representation. A binary random-access list of size n contains a tree for each one in the binary representation of n. The rank of each tree corresponds to the rank of the corresponding digit; if the ith bit of n is one, then the random-access list contains a tree of size 2^i. We can use any of the three kinds of trees, and either a dense or

```
signature RANDOMACCESSLIST =
sig
  type α RList

  val empty   : α RList
  val isEmpty : α RList → bool

  val cons    : α × α RList → α RList
  val head    : α RList → α
  val tail    : α RList → α RList
        (* head and tail raise EMPTY if list is empty *)

  val lookup  : int × α RList → α
  val update  : int × α × α RList → α RList
        (* lookup and update raise SUBSCRIPT if index is out of bounds *)
end
```

Figure 9.4. Signature for random-access lists.

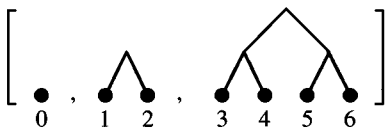

Figure 9.5. A binary random-access list containing the elements 0...6.

a sparse representation. For this example, we choose the simplest combination of features: complete binary leaf trees and a dense representation. The type α RList is thus

```
datatype α Tree = LEAF of α | NODE of int × α Tree × α Tree
datatype α Digit = ZERO | ONE of α Tree
type α RList = α Digit list
```

The integer in each node is the size of the tree. This number is redundant since the size of every tree is completely determined by the size of its parent or by its position in the list of digits, but we include it anyway for convenience. Trees are stored in increasing order of size, and the order of elements is left-to-right, both within and between trees. Thus, the head of the random-access list is the leftmost leaf of the smallest tree. Figure 9.5 shows a binary random-access list of size 7. Note that the maximum number of trees in a list of size n is $\lfloor \log(n+1) \rfloor$ and the maximum depth of any tree is $\lfloor \log n \rfloor$.

Inserting an element into a binary random-access list (using cons) is analogous to incrementing a binary number. Recall the increment function on dense binary numbers:

```
fun inc [] = [ONE]
  | inc (ZERO :: ds) = ONE :: ds
  | inc (ONE :: ds) = ZERO :: inc ds
```

To add a new element to the front of the list, we first convert the element into a leaf, and then insert the leaf into the list of trees using a helper function consTree that follows the rules of inc.

```
fun cons (x, ts) = consTree (LEAF x, ts)

fun consTree (t, []) = [ONE t]
  | consTree (t, ZERO :: ts) = ONE t :: ts
  | consTree (t1, ONE t2 :: ts) = ZERO :: consTree (link (t1, t2), ts)
```

The link helper function constructs a new tree from two equal-sized subtrees and automatically calculates the size of the new tree.

Deleting an element from a binary random-access list (using tail) is analogous to decrementing a binary number. Recall the decrement function on dense binary numbers:

```
fun dec [ONE] = []
  | dec (ONE :: ds) = ZERO :: ds
  | dec (ZERO :: ds) = ONE :: dec ds
```

The corresponding function on lists of trees is unconsTree. When applied to a list whose first digit has rank r, unconsTree returns a pair containing a tree of rank r, and the new list without that tree.

```
fun unconsTree [ONE t] = (t, [])
  | unconsTree (ONE t :: ts) = (t, ZERO :: ts)
  | unconsTree (ZERO :: ts) =
      let val (NODE (_, t1, t2), ts') = unconsTree ts
      in (t1, ONE t2 :: ts') end
```

The head and tail functions remove the leftmost leaf using unconsTree and then either return its element or discard it, respectively.

```
fun head ts = let val (LEAF x, _) = unconsTree ts in x end
fun tail ts = let val (_, ts') = unconsTree ts in ts' end
```

The lookup and update functions do not have analogous arithmetic operations. Rather, they take advantage of the organization of binary random-access lists as logarithmic-length lists of logarithmic-depth trees. Looking up an element is a two-stage process. We first search the list for the correct tree, and then search the tree for the correct element. The helper function lookupTree uses the size field in each node to determine whether the ith element is in the left subtree or the right subtree.

```
fun lookup (i, ZERO :: ts) = lookup (i, ts)
  | lookup (i, ONE t :: ts) =
      if i < size t then lookupTree (i, t) else lookup (i − size t, ts)

fun lookupTree (0, LEAF x) = x
  | lookupTree (i, NODE (w, t1, t2)) =
      if i < w div 2 then lookupTree (i, t1)
      else lookupTree (i − w div 2, t2)
```

update works in same way but also copies the path from the root to the updated leaf.

```
fun update (i, y, ZERO :: ts) = ZERO :: update (i, y, ts)
  | update (i, y, ONE t :: ts) =
      if i < size t then ONE (updateTree (i, y, t)) :: ts
      else ONE t :: update (i − size t, y, ts)

fun updateTree (0, y, LEAF x) = LEAF y
  | updateTree (i, y, NODE (w, t1, t2)) =
      if i < w div 2 then NODE (w, updateTree (i, y, t1), t2)
      else NODE (w, t1, updateTree (i − w div 2, y, t2))
```

The complete code for this implementation is shown in Figure 9.6.

cons, head, and tail perform at most $O(1)$ work per digit and so run in $O(\log n)$ worst-case time. lookup and update take at most $O(\log n)$ time to find the right tree, and then at most $O(\log n)$ time to find the right element in that tree, for a total of $O(\log n)$ worst-case time.

Exercise 9.1 Write a function drop of type int $\times$ α RList $\rightarrow$ α RList that deletes the first k elements of a binary random-access list. Your function should run in $O(\log n)$ time.

Exercise 9.2 Write a function create of type int $\times$ α $\rightarrow$ α RList that creates a binary random-access list containing n copies of some value x. This function should also run in $O(\log n)$ time. (You may find it helpful to review Exercise 2.5.)

Exercise 9.3 Reimplement BinaryRandomAccessList using a sparse representation such as

```
datatype α Tree = LEAF of α | NODE of int × α Tree × α Tree
type α RList = α Tree list
```

9.2.2 Zeroless Representations

One disappointing aspect of binary random-access lists is that the list functions cons, head, and tail run in $O(\log n)$ time instead of $O(1)$ time. Over the next three sections, we study variations of binary numbers that improve the running

```
structure BinaryRandomAccessList : RANDOMACCESSLIST =
struct
  datatype α Tree = LEAF of α | NODE of int × α Tree × α Tree
  datatype α Digit = ZERO | ONE of α Tree
  type α RList = α Digit list

  val empty = []
  fun isEmpty ts = null ts

  fun size (LEAF x) = 1
    | size (NODE (w, t1, t2)) = w
  fun link (t1, t2) = NODE (size t1+size t2, t1, t2)
  fun consTree (t, []) = [ONE t]
    | consTree (t, ZERO :: ts) = ONE t :: ts
    | consTree (t1, ONE t2 :: ts) = ZERO :: consTree (link (t1, t2), ts)
  fun unconsTree [] = raise EMPTY
    | unconsTree [ONE t] = (t, [])
    | unconsTree (ONE t :: ts) = (t, ZERO :: ts)
    | unconsTree (ZERO :: ts) =
        let val (NODE (_, t1, t2), ts') = unconsTree ts
        in (t1, ONE t2 :: ts') end

  fun cons (x, ts) = consTree (LEAF x, ts)
  fun head ts = let val (LEAF x, _) = unconsTree ts in x end
  fun tail ts = let val (_, ts') = unconsTree ts in ts' end

  fun lookupTree (0, LEAF x) = x
    | lookupTree (i, LEAF x) = raise SUBSCRIPT
    | lookupTree (i, NODE (w, t1, t2)) =
        if i < w div 2 then lookupTree (i, t1)
        else lookupTree (i − w div 2, t2)
  fun updateTree (0, y, LEAF x) = LEAF y
    | updateTree (i, y, LEAF x) = raise SUBSCRIPT
    | updateTree (i, y, NODE (w, t1, t2)) =
        if i < w div 2 then NODE (w, updateTree (i, y, t1), t2)
        else NODE (w, t1, updateTree (i − w div 2, y, t2))

  fun lookup (i, []) = raise SUBSCRIPT
    | lookup (i, ZERO :: ts) = lookup (i, ts)
    | lookup (i, ONE t :: ts) =
        if i < size t then lookupTree (i, t) else lookup (i − size t, ts)
  fun update (i, y, []) = raise SUBSCRIPT
    | update (i, y, ZERO :: ts) = ZERO :: update (i, y, ts)
    | update (i, y, ONE t :: ts) =
        if i < size t then ONE (updateTree (i, y, t)) :: ts
        else ONE t :: update (i − size t, y, ts)
end
```

Figure 9.6. Binary random-access lists.

times of all three functions to $O(1)$. We begin in this section with the head function.

Remark An obvious approach to making head run in $O(1)$ time is to store the first element separately from the rest of the list, à la the ExplicitMin functor of Exercise 3.7. Another solution is to use a sparse representation and either binomial trees or pennants, so that the head of the list is the root of the first tree. The solution we explore in this section has the advantage that it also improves the running times of lookup and update slightly. ◇

Currently, head is implemented via a call to unconsTree, which extracts the first element and rebuilds the list without that element. This approach yields compact code since unconsTree supports both head and tail, but wastes time building lists that are immediately discarded by head. For greater efficiency, we should implement head directly. As a special case, head can easily be made to run in $O(1)$ time whenever the first digit is non-zero.

```
fun head (ONE (LEAF x) :: _) = x
```

Inspired by this rule, we seek to arrange that the first digit is *always* non-zero. There are quite a few ad hoc solutions that satisfy this criterion, but a more principled solution is to use a *zeroless* representation, in which every digit is non-zero.

Zeroless binary numbers are constructed from ones and twos instead of zeros and ones. The weight of the ith digit is still 2^i. Thus, for example, the decimal number 16 can be written `2111` instead of `00001`. We can implement the increment function on zeroless binary numbers as follows:

```
datatype Digit = ONE | TWO
type Nat = Digit list

fun inc [] = [ONE]
  | inc (ONE :: ds) = TWO :: ds
  | inc (TWO :: ds) = ONE :: inc ds
```

Exercise 9.4 Write decrement and addition functions for zeroless binary numbers. Note that carries during additions can involve either ones or twos. ◇

Now, if we replace the type of digits in binary random-access lists with

```
datatype α Digit = ONE of α Tree | TWO of α Tree × α Tree
```

then we can implement head as

```
fun head (ONE (LEAF x) :: _) = x
  | head (TWO(LEAF x, LEAF y) :: _) = x
```

which clearly runs in $O(1)$ worst-case time.

Exercise 9.5 Implement the remaining functions for this type.

Exercise 9.6 Show that lookup and update on element i now run in $O(\log i)$ time.

Exercise 9.7 Under certain conditions, red-black trees (Section 3.3) can be viewed as a numerical representation. Compare and contrast zeroless binary random-access lists to red-black trees in which insertions are restricted to the leftmost position. Focus on the cons and insert functions and on the shape invariants of the structures produced by these functions.

9.2.3 Lazy Representations

Suppose we represent binary numbers as digit streams rather than digit lists. Then, the increment function becomes

```
fun lazy inc ($NIL) = $CONS (ONE, $NIL)
       | inc ($CONS (ZERO, ds)) = $CONS (ONE, ds)
       | inc ($CONS (ONE, ds)) = $CONS (ZERO, inc ds)
```

Note that this function is incremental.

In Section 6.4.1, we saw how lazy evaluation could be used to make insertions into binomial heaps run in $O(1)$ amortized time, so it should be no surprise that this version of inc also runs in $O(1)$ amortized time. We can prove this using the banker's method.

Proof Allow one debit on each ZERO and zero debits on each ONE. Suppose *ds* begins with k ONEs followed by a ZERO. Then inc *ds* changes each of these ONEs to a ZERO and the ZERO to a ONE. Allocate a new debit for each of these steps. Now, each of the ZEROs has a single debit, but the ONE has two debits: the debit inherited from the original suspension at that location plus the newly created debit. Discharging both debits restores the invariant. Since the amortized cost of a function is its unshared cost (in this case $O(1)$) plus the number of debits it discharges (in this case two), inc runs in $O(1)$ amortized time. □

Now, consider the decrement function.

```
fun lazy dec ($CONS (ONE, $NIL)) = $NIL
       | dec ($CONS (ONE, ds)) = $CONS (ZERO, ds)
       | dec ($CONS (ZERO, ds)) = $CONS (ONE, dec ds)
```

Since this function follows the same pattern as inc, but with the roles of the digits reversed, we would expect that a similar proof would yield a similar bound. And, in fact, it does provided we do not use *both* increments and decrements. However, if we use both functions, that at least one must be charged $O(\log n)$ amortized time. To see why, consider a sequence of increments and decrements that cycle between $2^k - 1$ and 2^k. In that case, every operation touches every digit, taking $O(n \log n)$ time altogether.

But didn't we prove that both functions run in $O(1)$ amortized time? What went wrong? The problem is that the two proofs require contradictory debit invariants. To prove that inc runs in $O(1)$ amortized time, we require that each ZERO has one debit and each ONE has zero debits. To prove that dec runs in $O(1)$ amortized time, we require that each ONE has one debit and each ZERO has zero debits.

The critical property that inc and dec both satisfy when used without the other is that at least half the operations that reach a given position in the stream terminate at that position. In particular, every inc or dec processes the first digit, but only every other operation processes the second digit. Similarly, every fourth operation processes the third digit, and so on. Intuitively, then, the amortized cost of a single operation is approximately

$$O(1 + 1/2 + 1/4 + 1/8 + \cdots) = O(1).$$

Classify the possible values of a digit as either *safe* or *dangerous* such that a function that reaches a safe digit always terminates there, but a function that reaches a dangerous digit might continue on to the next digit. To achieve the property that no two successive operations at a given index both proceed to the next index, we must guarantee that, whenever an operation processes a dangerous digit and continues on, it transforms the dangerous digit into a safe digit. Then, the next operation that reaches this digit is guaranteed not to continue. We can formally prove that every operation runs in $O(1)$ amortized time using a debit invariant in which a safe digit is allowed one debit, but a dangerous digit is allowed zero debits.

Now, the increment function requires that the largest digit be classified as dangerous, and the decrement function requires that the smallest digit be classified as dangerous. To support both functions simultaneously, we need a third digit to be the safe digit. Therefore, we switch to *redundant* binary numbers, in which each digit can be zero, one, or two. We can then implement inc and dec as follows:

```
datatype Digit = ZERO | ONE | TWO
type Nat = Digit Stream
```

```
fun lazy inc ($NIL) = $CONS (ONE, $NIL)
       | inc ($CONS (ZERO, ds)) = $CONS (ONE, ds)
       | inc ($CONS (ONE, ds)) = $CONS (TWO, ds)
       | inc ($CONS (TWO, ds)) = $CONS (ONE, inc ds)

fun lazy dec ($CONS (ONE, $NIL)) = $NIL
       | dec ($CONS (ONE, ds)) = $CONS (ZERO, ds)
       | dec ($CONS (TWO, ds)) = $CONS (ONE, ds)
       | dec ($CONS (ZERO, ds)) = $CONS (ONE, dec ds)
```

Note that the recursive cases of inc and dec—on TWO and ZERO, respectively—both produce ONEs. ONE is classified as safe, and ZERO and TWO are classified as dangerous. To see how redundancy helps us, consider incrementing the redundant binary number `222222` to get `1111111`. This operation takes seven steps. However, decrementing this value does not return to `222222`. Instead, it yields `0111111` in only one step. Thus, alternating increments and decrements no longer pose a problem.

Lazy binary numbers can serve as template for many other data structures. In Chapter 11, we will generalize this template into a design technique called *implicit recursive slowdown*.

Exercise 9.8 Prove that inc and dec both run in $O(1)$ amortized time using a debit invariant that allows one debit per ONE and zero debits per ZERO or TWO.

Exercise 9.9 Implement cons, head, and tail for random-access lists based on zeroless redundant binary numbers, using the type

```
datatype α Digit =
       ONE of α Tree
     | TWO of α Tree × α Tree
     | THREE of α Tree × α Tree × α Tree
type α RList = Digit Stream
```

Show that all three functions run in $O(1)$ amortized time.

Exercise 9.10 As demonstrated by scheduled binomial heaps in Section 7.3, we can apply scheduling to lazy binary numbers to achieve $O(1)$ worst-case bounds. Reimplement cons, head, and tail from the preceding exercise so that each runs in $O(1)$ worst-case time. You may find it helpful to have two distinct TWO constructors (say, TWO and TWO′) so that you can distinguish between recursive and non-recursive cases of cons and tail.

9.2.4 Segmented Representations

Another variation of binary numbers that yields $O(1)$ worst-case bounds is *segmented* binary numbers. The problem with ordinary binary numbers is that

carries and borrows can cascade. For example, incrementing $2^k - 1$ causes k carries in binary arithmetic. Symmetrically, decrementing 2^k causes k borrows. Segmented binary numbers solve this problem by allowing multiple carries or borrows to be executed in a single step.

Notice that incrementing a binary number takes k steps whenever the number begins with a block of k ones. Similarly, decrementing a binary number takes k steps whenever the number begins with a block of k zeros. Segmented binary numbers group contiguous sequences of identical digits into blocks so that we can execute a carry or borrow on an entire block in a single step. We represent segmented binary numbers as alternating blocks of zeros and ones using the following datatype:

```
datatype DigitBlock = ZEROS of int | ONES of int
type Nat = DigitBlock list
```

The integer in each DigitBlock represents the block's length.

We use the helper functions zeros and ones to add new blocks to the front of a list of blocks. These functions merge adjacent blocks of the same digit and discard empty blocks. In addition, zeros discards any trailing zeros.

```
fun zeros (i, []) = []
  | zeros (0, blks) = blks
  | zeros (i, ZEROS j :: blks) = ZEROS (i+j) :: blks
  | zeros (i, blks) = ZEROS i :: blks

fun ones (0, blks) = blks
  | ones (i, ONES j :: blks) = ONES (i+j) :: blks
  | ones (i, blks) = ONES i :: blks
```

Now, to increment a segmented binary number, we inspect the first block of digits (if any). If the first block contains zeros, then we replace the first zero with a one, creating a new singleton block of ones and shrinking the block of zeros by one. If the first block contains i ones, then we perform i carries in a single step by changing the ones to zeros and incrementing the next digit.

```
fun inc [] = [ONES 1]
  | inc (ZEROS i :: blks) = ones (1, zeros (i-1, blks))
  | inc (ONES i :: blks) = ZEROS i :: inc blks
```

In the third line, we know the recursive call to inc cannot loop because the next block, if any, must contain zeros. In the second line, the helper functions deal gracefully with the special case that the leading block contains a single zero.

Decrementing a segmented binary number is almost exactly the same, but with the roles of zeros and ones reversed.

```
fun dec (ONES i :: blks) = zeros (1, ones (i-1, blks))
  | dec (ZEROS i :: blks) = ONES i :: dec blks
```

Again, we know that the recursive call cannot loop because the next block must contain ones.

Unfortunately, although segmented binary numbers support inc and dec in $O(1)$ worst-case time, numerical representations based on segmented binary numbers end up being too complicated to be practical. The problem is that the idea of changing an entire block of ones to zeros, or vice versa, does not translate well to the realm of trees. More practical solutions can be obtained by combining segmentation with redundant binary numbers. Then we can return to processing digits (and therefore trees) one at a time. What segmentation gives us is the ability to process a digit in the middle of a sequence, rather than only at the front.

For example, consider a redundant representation in which blocks of ones are represented as a segment.

```
datatype Digits = ZERO | ONES of int | TWO
type Nat = Digits list
```

We define a helper function ones to handle the details of merging adjacent blocks and deleting empty blocks.

```
fun ones (0, ds) = ds
  | ones (i, ONES j :: ds) = ONES (i+j) :: ds
  | ones (i, ds) = ONES i :: ds
```

Think of a TWO as representing a carry in progress. To prevent cascades of carries, we must guarantee that we never have more than one TWO in a row. We maintain the invariant that the last non-one digit before each TWO is a ZERO. This invariant can be characterized by either the regular expression $(0 \mid 1 \mid 01^*2)^*$ or, if we also take into account the lack of trailing zeros, the regular expression $(0^*1 \mid 0^+1^*2)^*$. Note that the first digit is never a TWO. Thus, we can increment a number in $O(1)$ worst-case time by blindly incrementing the first digit.

```
fun simpleInc [] = [ONES 1]
  | simpleInc (ZERO :: ds) = ones (1, ds)
  | simpleInc (ONES i :: ds) = TWO :: one (i-1,ds)
```

The third line obviously violates the invariant by producing a leading TWO, but the second line might also violate the invariant if the first non-one digit in *ds* is a TWO. We restore the invariant with a function fixup that checks whether the first non-one digit is a TWO. If so, fixup replaces the TWO with a ZERO and increments the following digit, which is guaranteed not to be TWO.

```
fun fixup (TWO :: ds) = ZERO :: simpleInc ds
  | fixup (ONES i :: TWO :: ds) = ONES i :: ZERO :: simpleInc ds
  | fixup ds = ds
```

The second line of fixup is where we take advantage of segmentation, by skipping over a block of ones to check whether the next digit is a TWO. Finally, inc calls simpleInc, followed by fixup.

```
fun inc ds = fixup (simpleInc ds)
```

This implementation can also serve as template for many other data structures. Such a data structure comprises a sequence of levels, where each level can be classified as *green*, *yellow*, or *red*. Each color corresponds to a digit in the above implementation. Green corresponds to ZERO, yellow to ONE, and red to TWO. An operation on any given object may degrade the color of the first level from green to yellow, or from yellow to red, but never from green to red. The invariant is that the last non-yellow level before a red level is always green. A fixup procedure maintains the invariant by checking if the first non-yellow level is red. If so, the fixup procedure changes the color of the level from red to green, possibly degrading the color of the following level from green to yellow, or from yellow to red. Consecutive yellow levels are grouped in a block to support efficient access to the first non-yellow level. Kaplan and Tarjan [KT95] call this general technique *recursive slowdown*.

Exercise 9.11 Extend binomial heaps with segmentation so that insert runs in $O(1)$ worst-case time. Use the type

```
datatype Tree = NODE of Elem.T × Tree list
datatype Digit = ZERO | ONES of Tree list | TWO of Tree × Tree
type Heap = Digit list
```

Restore the invariant after a merge by eliminating all TWOs.

Exercise 9.12 The example implementation of binary numbers based on recursive slowdown supports inc in $O(1)$ worst-case time, but might require up to $O(\log n)$ for dec. Reimplement segmented, redundant binary numbers to support both inc and dec in $O(1)$ worst-case time by allowing each digit to be `0`, `1`, `2`, `3`, or `4`, where `0` and `4` are red, `1` and `3` are yellow, and `2` is green.

Exercise 9.13 Implement cons, head, tail, and lookup for a numerical representation of random-access lists based on the number system of the previous exercise. Your implementation should support cons, head, and tail in $O(1)$ worst-case time, and lookup in $O(\log i)$ worst-case time.

9.3 Skew Binary Numbers

In lazy binary numbers and segmented binary numbers, we have seen two methods for improving the asymptotic behavior of the increment and decrement functions from $O(\log n)$ to $O(1)$. In this section, we consider a third

method, which is usually simpler and faster in practice, but which involves a more radical departure from ordinary binary numbers.

In *skew binary numbers* [Mye83, Oka95b], the weight w_i of the ith digit is $2^{i+1} - 1$, rather than 2^i as in ordinary binary numbers. Digits may be zero, one, or two (i.e., $D_i = \{0, 1, 2\}$). For example, the decimal number 92 could be written `002101` (least-significant digit first).

This number system is redundant, but, if we add the further constraint that only the lowest non-zero digit may be two, then we regain unique representations. Such a number is said to be in *canonical form*. Henceforth, we will assume that all skew binary numbers are in canonical form.

Theorem 9.1 (Myers [Mye83]) *Every natural number has a unique skew binary canonical form.*

Recall that the weight of digit i is $2^{i+1} - 1$ and note that $1 + 2(2^{i+1} - 1) = 2^{i+2} - 1$. This implies that we can increment a skew binary number whose lowest non-zero digit is two by resetting the two to zero and incrementing the next digit from zero to one or from one to two. (The next digit cannot already be two.) Incrementing a skew binary number that does not contain a two is even easier — simply increment the lowest digit from zero to one or from one to two. In both cases, the result is still in canonical form. And, assuming we can find the lowest non-zero digit in $O(1)$ time, both cases take only $O(1)$ time!

We cannot use a dense representation for skew binary numbers since scanning for the lowest non-zero digit would take more than $O(1)$ time. Instead, we choose a sparse representation, so that we always have immediate access to the lowest non-zero digit.

```
type Nat = int list
```

The integers represent either the rank or weight of each non-zero digit. For now, we use weights. The weights are stored in increasing order, except that the smallest two weights may be identical, indicating that the lowest non-zero digit is two. Given this representation, we implement inc as follows:

```
fun inc (ws as w1 :: w2 :: rest) =
      if w1 = w2 then (1+w1+w2) :: rest else 1 :: ws
  | inc ws = 1 :: ws
```

The first clause checks whether the first two weights are equal and then either combines the weights into the next larger weight (incrementing the next digit) or adds a new weight of 1 (incrementing the smallest digit). The second clause

handles the case that *ws* is empty or contains only a single weight. Clearly, inc runs in only $O(1)$ worst-case time.

Decrementing a skew binary number is just as easy as incrementing a number. If the lowest digit is non-zero, then we simply decrement that digit from two to one or from one to zero. Otherwise, we decrement the lowest non-zero digit and reset the previous zero to two. This can be implemented as follows:

```
fun dec (1 :: ws) = ws
  | dec (w :: ws) = (w div 2) :: (w div 2) :: ws
```

In the second line, note that if $w = 2^{k+1} - 1$, then $\lfloor w/2 \rfloor = 2^k - 1$. Clearly, dec also runs in only $O(1)$ worst-case time.

9.3.1 Skew Binary Random-Access Lists

We next design a numerical representation for random-access lists, based on skew binary numbers. The basic representation is a list of trees, with one tree for each one digit and two trees for each two digit. The trees are maintained in increasing order of size, except that the smallest two trees are the same size when the lowest non-zero digit is two.

The sizes of the trees correspond to the weights in skew binary numbers, so a tree representing the ith digit has size $2^{i+1} - 1$. Up until now, we have mainly considered trees whose sizes are powers of two, but we have also encountered a kind of tree whose sizes have the desired form: complete binary trees. Therefore, we represent skew binary random-access lists as lists of complete binary trees.

To support head efficiently, the first element in the random-access list should be the root of the first tree, so we store the elements within each tree in left-to-right preorder and with the elements in each tree preceding the elements in the next tree.

In previous examples, we have stored a size or rank in every node, even when that information was redundant. For this example, we adopt the more realistic approach of maintaining size information only for the root of each tree in the list, and not for every subtree as well. The type of skew binary random-access lists is therefore

```
datatype α Tree = LEAF of α | NODE of α × α Tree × α Tree
type α RList = (int × α Tree) list
```

Now, we can define cons in analogy to inc.

```
fun cons (x, ts as (w1, t1) :: (w2, t2) :: rest) =
      if w1 = w2 then (1+w1+w2, NODE (x, t1, t2)) :: rest)
      else (1, LEAF x) :: ts
  | cons (x, ts) = (1, LEAF x) :: ts
```

head and tail inspect and remove the root of the first tree. tail returns the children of the root (if any) back to the front of the list, where they represent a new two digit.

```
fun head ((1, LEAF x) :: ts) = x
  | head ((w, NODE (x, t1, t2)) :: ts) = x
fun tail ((1, LEAF x) :: ts) = ts
  | tail ((w, NODE (x, t1, t2)) :: ts) = (w div 2, t1) :: (w div 2, t2) :: ts
```

To lookup an element, we first search the list for the right tree, and then search the tree for the right element. When searching a tree, we keep track of the size of the current tree.

```
fun lookup (i, (w, t) :: ts) =
      if i < w then lookupTree (w, i, t)
      else lookup (i-w, ts)

fun lookupTree (1, 0, LEAF x) = x
  | lookupTree (w, 0, NODE (x, t1, t2)) = x
  | lookupTree (w, i, NODE (x, t1, t2)) =
      if i < w div 2 then lookupTree (w div 2, i-1, t1)
      else lookupTree (w div 2, i - 1 - w div 2, t2)
```

Note that in the penultimate line, we subtract one from i because we have skipped over x. In the last line, we subtract $1 + \lfloor w/2 \rfloor$ from i because we have skipped over x and all the elements in t_1. update and updateTree are defined similarly, and are shown in Figure 9.7, which contains the complete implementation.

It is easy to verify that cons, head, and tail run in $O(1)$ worst-case time. Like binary random-access lists, skew binary random-access lists are logarithmic-length lists of logarithmic-depth trees, so lookup and update run in $O(\log n)$ worst-case time. In fact, every unsuccessful step of lookup or update discards at least one element, so this bound can be reduced slightly to $O(\min(i, \log n))$.

Hint to Practitioners: Skew binary random-access lists are a good choice for applications that take advantage of both the list-like aspects and the array-like aspects of random-access lists. Although there are better implementations of lists, and better implementations of (persistent) arrays, none are better at both [Oka95b].

Exercise 9.14 Rewrite the HoodMelvilleQueue structure from Section 8.2.1 to use skew binary random-access lists instead of regular lists. Implement lookup and update functions on these queues.

```
structure SkewBinaryRandomAccessList : RANDOMACCESSLIST =
struct
  datatype α Tree = LEAF of α | NODE of α × α Tree × α Tree
  type α RList = (int × α Tree) list      (* integer is the weight of the tree *)

  val empty = [ ]
  fun isEmpty ts = null ts

  fun cons (x, ts as (w1, t1) :: (w2, t2) :: ts') =
        if w1 = w2 then (1+w1+w2, NODE (x, t1, t2)) :: ts'
        else (1, LEAF x) :: ts
    | cons (x, ts) = (1, LEAF x) :: ts
  fun head [ ] = raise EMPTY
    | head ((1, LEAF x) :: ts) = x
    | head ((w, NODE (x, t1, t2)) :: ts) = x
  fun tail [ ] = raise EMPTY
    | tail ((1, LEAF x) :: ts) = ts
    | tail ((w, NODE (x, t1, t2)) :: ts) = (w div 2, t1) :: (w div 2, t2) :: ts

  fun lookupTree (1, 0, LEAF x) = x
    | lookupTree (1, i, LEAF x) = raise SUBSCRIPT
    | lookupTree (w, 0, NODE (x, t1, t2)) = x
    | lookupTree (w, i, NODE (x, t1, t2)) =
        if i ≤ w div 2 then lookupTree (w div 2, i−1, t1)
        else lookupTree (w div 2, i − 1 − w div 2, t2)
  fun updateTree (1, 0, y, LEAF x) = LEAF y
    | updateTree (1, i, y, LEAF x) = raise SUBSCRIPT
    | updateTree (w, 0, y, NODE (x, t1, t2)) = NODE (y, t1, t2)
    | updateTree (w, i, y, NODE (x, t1, t2)) =
        if i ≤ w div 2 then NODE (x, updateTree (w div 2, i−1, y, t1), t2)
        else NODE (x, t1, updateTree (w div 2, i − 1 − w div 2, y, t2))

  fun lookup (i, [ ]) = raise SUBSCRIPT
    | lookup (i, (w, t) :: ts) =
        if i < w then lookupTree (w, i, t)
        else lookup (i−w, ts)
  fun update (i, y, [ ]) = raise SUBSCRIPT
    | update (i, y, (w, t) :: ts) =
        if i < w then (w, updateTree (w, i, y, t)) :: ts
        else (w, t) :: update (i−w, y, ts)
end
```

Figure 9.7. Skew binary random-access lists.

9.3.2 Skew Binomial Heaps

Finally, we consider a hybrid numerical representation for heaps based on both skew binary numbers and ordinary binary numbers. Incrementing a skew binary number is both quick and simple, and serves admirably as a template for the insert function. Unfortunately, addition of two arbitrary skew binary num-

bers is awkward. We therefore base the merge function on ordinary binary addition, rather than skew binary addition.

A *skew binomial tree* is a binomial tree in which every node is augmented with a list of up to r elements, where r is the rank of the node in question.

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

Unlike ordinary binomial trees, the size of a skew binomial tree is not completely determined by its rank; rather the rank of a skew binomial tree determines a range of possible sizes.

Lemma 9.2 *If t is a skew binomial tree of rank r, then* $2^r \leq |t| \leq 2^{r+1} - 1$.

Exercise 9.15 Prove Lemma 9.2. ◇

Skew binomial trees may be *linked* or *skew linked*. The link function combines two trees of rank r to form a tree of rank $r + 1$ by making the tree with the larger root a child of the tree with the smaller root.

```
fun link (t1 as NODE (r, x1, xs1, c1), t2 as NODE (_, x2, xs2, c2)) =
      if Elem.leq (x1, x2) then NODE (r+1, x1, xs1, t2 :: c1)
      else NODE (r+1, x2, xs2, t1 :: c2)
```

The skewLink function combines two trees of rank r with an additional element to form a tree of rank $r + 1$ by first linking the two trees, and then comparing the root of the resulting tree with the additional element. The smaller of the two elements remains as the root, and the larger is added to the auxiliary list of elements.

```
fun skewLink (x, t1, t2) =
      let val NODE (r, y, ys, c) = link (t1, t2)
      in
         if Elem.leq (x, y) then NODE (r, x, y :: ys, c)
         else NODE (r, y, x :: ys, c)
      end
```

A skew binomial heap is represented as a list of heap-ordered skew binomial trees of increasing rank, except that the first two trees may share the same rank. Since skew binomial trees of the same rank may have different sizes, there is no longer a direct correspondence between the trees in the heap and the digits in the skew binary number representing the size of the heap. For example, even though the skew binary representation of 4 is `11`, a skew binomial heap of size 4 may contain one rank 2 tree of size 4; two rank 1 trees, each of size 2; a rank 1 tree of size 3 and a rank 0 tree; or a rank 1 tree of size 2 and two rank 0 trees. However, the maximum number of trees in a heap is still $O(\log n)$.

The big advantage of skew binomial heaps is that we can insert a new element in $O(1)$ time. We first compare the ranks of the two smallest trees. If they are the same, we skew link the new element with these two trees. Otherwise, we make a new singleton tree and add it to the front of the list.

```
fun insert (x, ts as t1 :: t2 :: rest) =
      if rank t1 = rank t2 then skewLink (x, t1, t2) :: rest
      else NODE (0, x, [], []) :: ts
  | insert (x, ts) = NODE (0, x, [], []) :: ts
```

The remaining functions are nearly identical to their counterparts from ordinary binomial heaps. We change the name of the old merge function to mergeTrees. It still walks through both lists of trees, performing a regular link (not a skew link!) whenever it finds two trees of equal rank. Since both mergeTrees and its helper function insTree expect lists of strictly increasing rank, merge normalizes its two arguments to remove any leading duplicates before calling mergeTrees.

```
fun normalize [] = []
  | normalize (t :: ts) = insTree (t, ts)
fun merge (ts1, ts2) = mergeTrees (normalize ts1, normalize ts2)
```

findMin and removeMinTree are completely unaffected by the switch to skew binomial heaps since they both ignore ranks, being concerned only with the root of each tree. deleteMin is only slightly changed. It begins the same by removing the tree with the minimum root, reversing the list of children, and merging the reversed children with the remaining trees. But then it reinserts each of the elements from the auxiliary list attached to discarded root.

```
fun deleteMin ts =
      let val (NODE (_, x, xs, ts1), ts2) = removeMinTree ts
          fun insertAll ([], ts) = ts
            | insertAll (x :: xs, ts) = insertAll (xs, insert (x, ts))
      in insertAll (xs, merge (rev ts1, ts2)) end
```

Figure 9.8 presents the complete implementation of skew binomial heaps.

insert runs in $O(1)$ worst-case time, while merge, findMin, and deleteMin run in the same time as their counterparts for ordinary binomial queues, i.e., $O(\log n)$ worst-case time each. Note that the various phases of deleteMin — finding the tree with the minimum root, reversing the children, merging the children with the remaining trees, and reinserting the auxiliary elements — take $O(\log n)$ time each.

If desired, we can improve the running time of findMin to $O(1)$ using the ExplicitMin functor of Exercise 3.7. In Section 10.2.2, we will see how to improve the running time of merge to $O(1)$ as well.

```
functor SkewBinomialHeap (Element : ORDERED) : HEAP =
struct
  structure Elem = Element
  datatype Tree = NODE of int × Elem.T × Elem.T list × Tree list
  type Heap = Tree list
  val empty = []
  fun isEmpty ts = null ts
  fun rank (NODE (r, x, xs, c)) = r
  fun root (NODE (r, x, xs, c)) = x
  fun link (t1 as NODE (r, x1, xs1, c1), t2 as NODE (_, x2, xs2, c2)) =
        if Elem.leq (x1, x2) then NODE (r+1, x1, xs1, t2 :: c1)
        else NODE (r+1, x2, xs2, t1 :: c2)
  fun skewLink (x, t1, t2) =
        let val NODE (r, y, ys, c) = link (t1, t2)
        in
          if Elem.leq (x, y) then NODE (r, x, y :: ys, c)
          else NODE (r, y, x :: ys, c)
        end
  fun insTree (t, []) = [t]
    | insTree (t1, t2 :: ts) =
        if rank t1 < rank t2 then t1 :: t2 :: ts else insTree (link (t1, t2), ts)
  fun mergeTrees (ts1, []) = ts1
    | mergeTrees ([], ts2) = ts2
    | mergeTrees (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
        if rank t1 < rank t2 then t1 :: mergeTrees (ts1', ts2)
        else if rank t2 < rank t1 then t2 :: mergeTrees (ts1,ts2')
        else insTree (link (t1, t2), mergeTrees (ts1', ts2'))
  fun normalize [] = []
    | normalize (t :: ts) = insTree (t, ts)
  fun insert (x, ts as t1 :: t2 :: rest) =
        if rank t1 = rank t2 then skewLink (x, t1, t2) :: rest
        else NODE (0, x, [], []) :: ts
    | insert (x, ts) = NODE (0, x, [], []) :: ts
  fun merge (ts1, ts2) = mergeTrees (normalize ts1, normalize 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, xs, ts1), ts2) = removeMinTree ts
            fun insertAll ([], ts) = ts
              | insertAll (x :: xs, ts) = insertAll (xs, insert (x, ts))
        in insertAll (xs, merge (rev ts1, ts2)) end
end
```

Figure 9.8. Skew binomial heaps.

Exercise 9.16 Suppose we want a delete function of type Elem.T $\times$ Heap $\rightarrow$ Heap. Write a functor that takes an implementation H of heaps and produces an implementation of heaps that supports delete as well as all the other usual heap functions. Use the type

```
type Heap = H.Heap × H.Heap
```

where one of the primitive heaps represents positive occurrences of elements and the other represents negative occurrences. A negative occurrence of an element means that that element has been deleted, but not yet physically removed from the heap. Positive and negative occurrences of the same element cancel each other out and are physically removed when both become the minimum elements of their respective heaps. Maintain the invariant that the minimum element of the positive heap is strictly smaller than the minimum element of the negative heap. (This implementation has the curious property that an element can be deleted before it has been inserted, but this is acceptable for many applications.)

9.4 Trinary and Quaternary Numbers

In computer science, we are so accustomed to thinking about binary numbers, that we sometimes forget that other bases are possible. In this section, we consider uses of base 3 and base 4 arithmetic in numerical representations.

The weight of each digit in base k is k^r, so we need families of trees with sizes of this form. We can generalize each of the families of trees used in binary numerical representations as follows.

Definition 9.4 (Complete k-ary leaf trees) A complete k-ary tree of rank 0 is a leaf and a complete k-ary tree of rank $r > 0$ is a node with k children, each of which is a complete k-ary tree of rank $r - 1$. A complete k-ary tree of rank r has $(k^{r+1} - 1)/(k - 1)$ nodes and k^r leaves. A complete k-ary leaf tree is a complete k-ary tree that contains elements only at the leaves.

Definition 9.5 (k-nomial trees) A k-nomial tree of rank r is a node with $k - 1$ children of each rank from $r - 1$ to 0. Alternatively, a k-nomial tree of rank $r > 0$ is a k-nomial tree of rank $r-1$ to which $k-1$ other k-nomial trees of rank $r - 1$ have been added as the leftmost children. From the second definition, it is easy to see that a k-nomial tree of rank r contains k^r nodes.

Definition 9.6 (k-ary pennants) A k-ary pennant of rank 0 is a single node and a k-ary pennant of rank $r > 0$ is a node with $k - 1$ children, each of

which is a complete k-ary tree of rank $r-1$. Each of the subtrees contains $(k^r-1)/(k-1)$ nodes, so the entire tree contains k^r nodes.

The advantage of choosing bases larger than 2 is that fewer digits are needed to represent each number. Whereas a number in base 2 contains approximately $\log_2 n$ digits, a number in base k contains approximately $\log_k n = \log_2 n / \log_2 k$ digits. For example, base 4 uses approximately half as many digits as base 2. On the other hand, there are now more possible values for each digit, so processing each digit might take longer. In numerical representations, processing a digit in base k often takes about $k+1$ steps, so an operation that processes every digit should take about $(k+1)\log_k n = \frac{k+1}{\log_2 k}\log n$ steps altogether. The following table displays values of $(k+1)/\log_2 k$ for $k = 2, \ldots, 8$.

k	2	3	4	5	6	7	8
$(k+1)/\log_2 k$	3.00	2.52	2.50	2.58	2.71	2.85	3.0

This table suggests that numerical representations based on trinary or quaternary numbers might be as much as 16% faster than numerical representations based on binary numbers. Other factors, such as increased code size, tend to make larger bases less effective as k increases, so one rarely observes speedups that large in practice. In fact, trinary and quaternary representations often run slower than binary representations on small data sets. However, for large data sets, trinary and quaternary representations often yield speedups of 5 to 10%.

Exercise 9.17 Implement trinomial heaps using the type

```
datatype Tree = NODE of Elem.T × (Tree × Tree) list
datatype Digit = ZERO | ONE of Tree | TWO of Tree × Tree
type Heap = Digit list
```

Exercise 9.18 Implement zeroless quaternary random-access lists using the type

```
datatype α Tree = LEAF of α | NODE of α Tree vector
datatype α RList = α Tree vector list
```

where each vector in a NODE contains four trees, and each vector in a list contains one to four trees.

Exercise 9.19 We can also adapt the notion of skew binary numbers to arbitrary bases. In skew k-ary numbers, the ith digit has weight $(k^{i+1}-1)/(k-1)$. Each digit is chosen from $\{0, \ldots, k-1\}$ except that the lowest non-zero digit may be k. Implement skew trinary random-access lists using the type

```
datatype α Tree = LEAF of α | NODE of α × α Tree × α Tree × α Tree
type α RList = (int × α Tree) list
```

9.5 Chapter Notes

Data structures that can be cast as numerical representations are surprisingly common, but only rarely is the connection to a variant number system noted explicitly [GMPR77, Mye83, CMP88, KT96b]. Skew binary random-access lists originally appeared in [Oka95b]. Skew binomial heaps originally appeared in [BO96].