FUNCTIONAL PEARLS

Explaining Binomial Heaps

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(e-mail: ralf@uran.informatik.uni-bonn.de*)*

1 Introduction

Functional programming languages are an excellent tool for teaching algorithms and data structures. This paper explains binomial heaps, a beautiful data structure for priority queues, using the functional programming language Haskell (Peterson & Hammond, 1997). We largely follow a deductive approach: using the metaphor of a tennis tournament we show that binomial heaps arise naturally through a number of logical steps. Haskell supports the deductive style of presentation very well: new types are introduced at ease, algorithms can be expressed clearly and succinctly, and Haskell's type classes allow to capture common algorithmic patterns. The paper aims at the level of an undergraduate student who has experience in reading and writing Haskell programs, and who is familiar with the concept of a priority queue.

2 Priority queues

The abstract data type 'priority queue' provides at least the following five operations: $\emptyset$ represents the empty queue; $\wr a \int$ denotes the queue, which contains a as the single element; *insert* a q inserts a into queue q; $q_1 \uplus q_2$ denotes the union of queues q_1 and q_2 (sometimes termed 'meld'); and *splitMin* q extracts a minimal element from q. The notation has been chosen to emphasize the fact that priority queues are conceptually bags, ie unordered collections possibly with duplicates. Priority queues are the data type of choice when an efficient access to the smallest element of a varying collection of elements is required. Applications include discrete event simulation and job scheduling. Here is the class definition for priority queues.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{MinView}\ q\ a & = & \mathit{Min}\ a\ (q\ a) \mid \mathit{Infty} \\
\textbf{class}\ \mathit{PriorityQueue}\ q\ \textbf{where} & & \\
\quad \emptyset & :: & (\mathit{Ord}\ a) \Rightarrow q\ a \\
\quad \wr \cdot \int & :: & (\mathit{Ord}\ a) \Rightarrow a \rightarrow q\ a \\
\quad \mathit{insert} & :: & (\mathit{Ord}\ a) \Rightarrow a \rightarrow q\ a \rightarrow q\ a \\
\quad (\uplus) & :: & (\mathit{Ord}\ a) \Rightarrow q\ a \rightarrow q\ a \rightarrow q\ a \\
\quad \mathit{splitMin} & :: & (\mathit{Ord}\ a) \Rightarrow q\ a \rightarrow \mathit{MinView}\ q\ a \\
\quad \mathit{insert}\ a\ q & = & \wr a \int \uplus q
\end{array}
$$

Table 1. *Ladies' singles of the 1996 All England Championship*

Quarterfinal	Semifinal	Final	Champion
K Date vs. M Pierce	K Date		
	vs.	**S Graf**	
S Graf vs. J Novotna	**S Graf**		
		vs.	S Graf
J Wiesner vs. **A Sanchez Vicario**	**A Sanchez Vicario**		
	vs.	A Sanchez Vicario	
M Fernandez vs. **M McGrath**	M McGrath		

While the meaning of the first four operations should be clear, *splitMin* is in need of explanation. The call *splitMin* q has two possible outcomes: if q is empty *splitMin* q returns *Infty*, otherwise it yields *Min* a q' where a is a minimal element of q and q' consists of all elements in q except a.

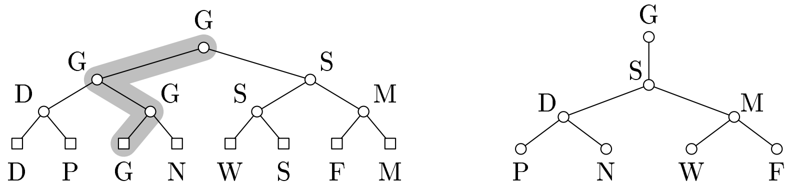
Note that *PriorityQueue* is a constructor class (Jones, 1995); the class variable q ranges over type constructors rather than types (q has kind $* \to *$). A similar comment applies to the definition of *MinView*: the parameter q is a type constructor while a is a type.

To derive an efficient implementation of priority queues we proceed in two steps. First, we consider a simple instance of the problem, namely, to determine the smallest of a *fixed* bag of elements. In a second step we drop the restriction that all elements are given in advance by making the algorithms *incremental*.

3 Tournament trees

Let us assume that we look for the best of, say, n tennis players. The first idea which probably crosses one's mind is to organize a knock-off tournament.[†] For definiteness, consider the results of the Ladies' singles of the 1996 All England Championship listed in Table 1. We see that the course of matches forms a full binary tree; each external node corresponds to a participant, and each internal node corresponds to the winner of a match. The tree representation is shown in Figure 1(a). Regarded as a data structure for priority queues tournament trees appear to be deficient because of the many repeated entries they contain. The champion, for instance, appears on every level of the tree. We may repair this defect if we label each internal node with

[†] It is important to be concrete here. If we posed the abstract problem of determining the smallest of a bag of elements we would probably provoke solutions like *foldl min* ∞.



(a) tournament tree (b) topped loser tree

Fig. 1. Different representations of the tournament listed in Table 1

the loser of the match, instead of the winner, and drop the external nodes altogether. We thus turn the tree of winners into a *tree of losers*. Since every participant—with the exception of the champion—loses exactly one match the labelling of nodes is now unique. If we place the champion additionally at the top of the tree we obtain the structure displayed in Figure 1(b).

To represent topped loser trees in Haskell we reuse the data type *MinView* building upon the standard definition of labelled binary trees.

```
type ToppedTree a  =  MinView BinTree a
data BinTree a     =  Bin a (BinTree a) (BinTree a) | Empty
```

Let us consider next how to determine the second-best player. Assuming a transitive ranking only those players who lost to the champion must be taken into account. In the example above there are three candidates for the second prize: N, D, and S. Consequently, two additional matches are required; the tree-structure of the tournament suggests to let S compete against the winner of D vs. N.

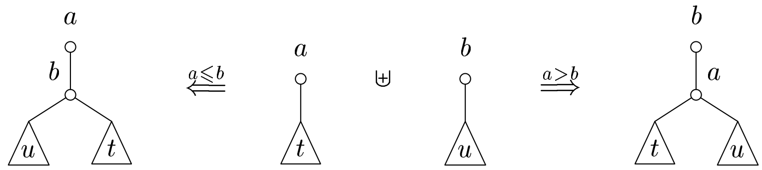
Unfortunately, we cannot set up the matches on the basis of a topped loser tree. It simply does not provide enough information to determine the champion's opponents. This can be seen if we annotate its edges with the established ranking, cf Figure 2(a). The problem is that some players dominate the left, others the

(a) original loser tree (b) left-ordered loser tree

Fig. 2. Determining the second-best player

right subtree but we cannot tell which one. A cure, however, is ready at hand: We arrange the tree such that a loser always dominates the left subtree. Then the champion's opponents are located on the right spine. The modified tree is displayed in Figure 2(b).

The left-ordering property must be taken into account whenever a match is played, ie two topped trees are melded.



Note that the subtrees, t and u, are swapped if $a \leqslant b$. Having fixed the data structure and its properties we are now ready to give the first implementation of priority queues.[‡]

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{PriorityQueue}\ \mathit{ToppedTree}\ \textbf{where}} \\
\quad \emptyset & = & \mathit{Infty} \\
\quad \wr a \int & = & \mathit{Min}\ a\ \mathit{Empty} \\
\quad \mathit{Infty} \uplus u & = & u \\
\quad t \uplus \mathit{Infty} & = & t \\
\quad \mathit{Min}\ a\ t \uplus \mathit{Min}\ b\ u & & \\
\qquad \mid a \leqslant b & = & \mathit{Min}\ a\ (\mathit{Bin}\ b\ u\ t) \\
\qquad \mid \mathit{otherwise} & = & \mathit{Min}\ b\ (\mathit{Bin}\ a\ t\ u) \\
\quad \mathit{splitMin}\ \mathit{Infty} & = & \mathit{Infty} \\
\quad \mathit{splitMin}\ (\mathit{Min}\ a\ t) & = & \mathit{Min}\ a\ (\mathit{secondBest}\ t) \\
\qquad \textbf{where}\ \mathit{secondBest}\ \mathit{Empty} & = & \mathit{Infty} \\
\qquad\qquad \mathit{secondBest}\ (\mathit{Bin}\ a'\ l\ r) & = & \mathit{Min}\ a'\ l \uplus \mathit{secondBest}\ r
\end{array}
$$

This instance has the amazing property that $(\uplus)$ can be performed in constant time. On the negative side *splitMin* exhibits $\Theta(n)$ worst-case behaviour.[§] Consider, for example, the call $\mathit{splitMin}\ (\mathit{foldr}\ \mathit{insert}\ \emptyset\ [n, n-1\,..\,1])$. The crux is that $(\uplus)$ is repeatedly applied to trees of different sizes resulting in a degenerated tree. Interestingly, the performance depends very much on the order of elements: the call $\mathit{splitMin}\ (\mathit{foldr}\ \mathit{insert}\ \emptyset\ [1\,..\,n])$, for example, takes only constant time.

4 Bottom-up tournament trees

We have seen that repeated insertions into a topped tree may result in a degenerated list-like structure degrading the performance of subsequent *splitMin* operations. This never happens in a tennis tournament because the participants are known in advance. Dropping this assumption we are faced with the following problem:

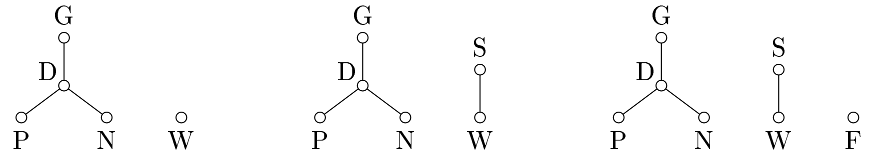
Again, we look for the best of n tennis players. But, due to prior commitments the players do not arrive at the same time; rather they join the tournament one after the other. We probably do not even know how many players are going to

[‡] The instance declaration is not legal Haskell since *ToppedTree* is not a datatype defined by **data** or by **newtype**. A datatype, however, introduces an additional data constructor which affects the readability of the code. Instead we employ **type** declarations as if they worked as **newtype** declarations.

[§] Since Haskell is a lazy language the bounds are amortized rather than worst-case bounds (Okasaki, 1996b).

participate. We only require the series of matches to be fair: opponents should always have played the same number of matches, ie only trees of equal size should be linked. Can we make arrangements to cope with this—from an organizer's point of view quite unpleasant—situation?

The answer is yes and the solution is quite simple, too. Returning to the Ladies' singles assume that the participants arrive in the following order: D, P, G, N, W, S, F, and M. Now, whenever a new player shows up we perform as many matches as possible. Here are three snapshots just after W, S, and F joined the tournament.

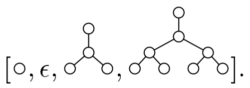


When the last player, M, arrives three pending matches can be carried out resulting in the tree of Figure 1(b). We see that the original tree is constructed in a left-to-right, bottom-up fashion. A match is performed if and only if there are two trees of equal size. This condition guarantees that all loser trees are perfectly balanced. The resulting structures, topped full binary trees, are called *pennants* by Sack and Strothotte (1990). Since a full binary tree of height h contains $2^{h+1} - 1$ nodes, a pennant of height h contains 2^h nodes. This implies that a tournament of size n contains a pennant of height i if the i-th bit in the binary representation of n is 1.

In Haskell we represent a tournament by a list of topped trees. For reasons, that will become clear later, we call the data structure 'binary binomial heap'.

$$\textbf{type}\ \mathit{BinBinomialHeap}\ a \quad = \quad [\,\mathit{ToppedTree}\ a\,]$$

A tournament of size $13 = 1 + 0 + 4 + 8$, for example, is represented by the list



The list contains an empty pennant (abbreviated by ϵ) which corresponds to the 0 in the binary representation of 13. Okasaki (1996a) discusses alternative representations.

5 Digression: Binary addition

Since the representation of a tournament is uniquely determined by the binary decomposition of the number of participants it probably comes as no surprise that the operations on heaps resemble arithmetic functions on binary numbers: inserting an element is analogous to incrementing a number, melding two tournaments is analogous to adding two numbers. For that reason we will briefly review the method of summing binary numbers. To abstract away from clerical details of representation we first introduce a class for binary digits.

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Eq}\ b) \Rightarrow \mathit{BinaryDigit}\ b\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{zero} & :: & b \\
\mathit{carry}, \mathit{sum} & :: & b \to b \to b
\end{array}
\end{array}
$$

The function *sum* calculates the sum bit of two bits, *carry* accordingly determines the carry bit. Recall from the course on computer architecture that a single step of the binary addition is performed by a full adder which calculates the sum of three bits, the two input bits and the carry bit.

$$
\begin{array}{lcl}
\mathit{fullAdder} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \rightarrow b \rightarrow b \rightarrow (b, b) \\
\mathit{fullAdder}\ c\ a\ b & = & (s_2, \mathit{sum}\ c_1\ c_2) \\
\quad \mathbf{where}\ (s_1, c_1) & = & \mathit{halfAdder}\ a\ b \\
\quad \phantom{\mathbf{where}}\ (s_2, c_2) & = & \mathit{halfAdder}\ c\ s_1
\end{array}
$$

A half adder calculates the sum of two bits.

$$
\begin{array}{lcl}
\mathit{halfAdder} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \rightarrow b \rightarrow (b, b) \\
\mathit{halfAdder}\ a\ b & = & (\mathit{sum}\ a\ b, \mathit{carry}\ a\ b)
\end{array}
$$

A binary number is represented by a list of binary digits, the least significant digit coming *first*. Keeping the digits in increasing order of weight is a natural choice since addition proceeds from the least to the most significant digit. To make the representation unique we furthermore disallow trailing zeros. The Haskell function for binary addition, *addWithCarry*, essentially corresponds to a ripple-carry addition (Cormen *et al.*, 1991, p. 661–662). Contrary to an electronic circuit we must arrange for the likely case that the numerals have unequal length. The helper function *addDigit* takes care of these cases.

$$
\begin{array}{lcl}
\mathit{add} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow [b] \rightarrow [b] \rightarrow [b] \\
\mathit{add}\ x\ y & = & \mathit{addWithCarry}\ \mathit{zero}\ x\ y
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{addWithCarry} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \rightarrow [b] \rightarrow [b] \rightarrow [b] \\
\mathit{addWithCarry}\ c\ [\,]\ y & = & \mathit{addDigit}\ c\ y \\
\mathit{addWithCarry}\ c\ x\ [\,] & = & \mathit{addDigit}\ c\ x \\
\mathit{addWithCarry}\ c\ (a : x)\ (b : y) & = & s : \mathit{addWithCarry}\ c'\ x\ y \\
\quad \mathbf{where}\ (s, c') & = & \mathit{fullAdder}\ c\ a\ b
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{addDigit} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \rightarrow [b] \rightarrow [b] \\
\mathit{addDigit}\ c\ x \mid c \equiv \mathit{zero} & = & x \\
\mathit{addDigit}\ c\ [\,] & = & [c] \\
\mathit{addDigit}\ c\ (a : x) & = & s : \mathit{addDigit}\ c'\ x \\
\quad \mathbf{where}\ (s, c') & = & \mathit{halfAdder}\ c\ a
\end{array}
$$

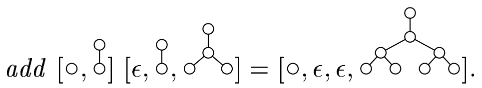
If we represent binary digits by integers,

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{BinaryDigit}\ \mathit{Int}\ \mathbf{where} \\
\quad \begin{array}{lcl}
\mathit{zero} & = & 0 \\
\mathit{carry}\ m\ n & = & (m + n)\ `\mathit{div}`\ 2 \\
\mathit{sum}\ m\ n & = & (m + n)\ `\mathit{mod}`\ 2
\end{array}
\end{array}
$$

we can try adding 3 and 6: *add* $[1,1]$ $[0,1,1] = [1,0,0,1]$. Due to the overloading we can reuse *add* to meld two lists of pennants. All we have to do is to supply the following instance declaration.

```
instance (Ord a) ⇒ BinaryDigit (ToppedTree a) where
    zero                      = Infty
    carry (Min a t) (Min b u)
        | a ⩽ b               = Min a (Bin b u t)
        | otherwise           = Min b (Bin a t u)
    carry _ _                 = Infty
    sum Infty u               = u
    sum t Infty               = t
    sum (Min _ _) (Min _ _)   = Infty
```

Note that we omit the necessary, but straightforward *Eq* instance declaration for topped trees. The function *carry* corresponds to melding two topped trees. The only difference is that *carry* returns *Infty* if one of the arguments is empty. The function *sum* determines the 'remainder' of a meld. It is instructive to see *add* in action:



The running time of *add* is determined by the length of the input lists. Since a tournament of size n comprises $\lceil \log_2(n+1) \rceil$ pennants the worst-case running time of $(\uplus)$ $(= add)$ is $\Theta(\log n)$. For *insert* one can derive tighter bounds: incrementing a binary number only takes $\Theta(1)$ amortized time (Okasaki, 1996b).

6 Binary binomial heaps

It remains to implement *splitMin* which puts our implementation on the testbench. Since we increased the cost of $(\uplus)$ from $\Theta(1)$ to $\Theta(\log n)$, we expect the running time of *splitMin* to improve. The operation proceeds in three steps. First, a pennant with a minimal root is determined and replaced by *zero*. This is most easily accomplished if we are able to compare topped trees as a whole. The following instance declaration defines a suitable ordering.

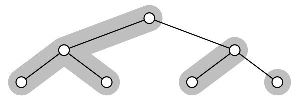
```
instance (Ord a) ⇒ Ord (ToppedTree a) where
    t ⩽ Infty             = True
    Infty ⩽ u             = False
    Min a _ ⩽ Min b _     = a ⩽ b
```

Note that the loser trees are not taken into account. This guarantees that comparing two trees has the same complexity as comparing two elements.

$$
\begin{array}{lcl}
\mathit{extractMin} & :: & (\mathit{Ord}\ t, \mathit{BinaryDigit}\ t) \Rightarrow [t] \to \mathit{MinView}\ [\,]\ t \\
\mathit{extractMin}\ [\,] & = & \mathit{Infty} \\
\mathit{extractMin}\ (a : x) & = & \mathbf{case}\ \mathit{extractMin}\ x\ \mathbf{of} \\
\quad \mathit{Infty} & \to & \mathit{Min}\ a\ [\,] \\
\quad \mathit{Min}\ b\ y \mid a \leqslant b & \to & \mathit{Min}\ a\ (\mathit{zero} : x) \\
\qquad\qquad \mid \mathit{otherwise} & \to & \mathit{Min}\ b\ (a : y)
\end{array}
$$

The type of *extractMin* is more general than actually needed: *extractMin* is not restricted to topped trees but works on binary digits. This extra generality pays off in the subsequent section when we change the representation of tournaments.

Having determined the required pennant we are left with the problem of merging the remaining list of pennants with a single full binary tree. Fortunately, we can convert a full binary tree into a tournament using the natural correspondence between binary trees and lists of topped trees. The figure on the right emphasizes the pennants hidden in a full binary tree.



This correspondence is easily coded as a Haskell program which constitutes the second step.

$$
\begin{array}{lcl}
\mathit{dismantle} & :: & \mathit{BinTree}\ a \to [\mathit{ToppedTree}\ a] \\
\mathit{dismantle}\ \mathit{Empty} & = & [\,] \\
\mathit{dismantle}\ (\mathit{Bin}\ a\ l\ r) & = & \mathit{Min}\ a\ l : \mathit{dismantle}\ r
\end{array}
$$

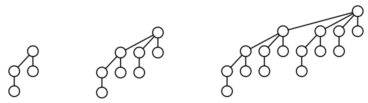
In the third and last step we simply meld the two lists. Note that the pennants resulting from the binary tree must be reversed beforehand. Since all three steps require $\Theta(\log n)$ time in the worst case, *splitMin* requires $\Theta(\log n)$ time, as well.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{PriorityQueue}\ \mathit{BinBinomialHeap}\ \mathbf{where}} \\
\quad \emptyset & = & [\,] \\
\quad \wr a \int & = & [\mathit{Min}\ a\ \mathit{Empty}] \\
\quad (\uplus) & = & \mathit{add} \\
\quad \mathit{splitMin}\ q & = & \mathbf{case}\ \mathit{extractMin}\ q\ \mathbf{of} \\
\qquad \mathit{Infty} & \to & \mathit{Infty} \\
\qquad \mathit{Min}\ (\mathit{Min}\ a\ t)\ \mathit{ts} & \to & \mathit{Min}\ a\ (\mathit{reverse}\ (\mathit{dismantle}\ t) \uplus \mathit{ts})
\end{array}
$$

This instance has the amazing feature that the running time of $(\uplus)$ is completely independent of the elements contained in the queues—quite contrary to our first implementation. The pro is also a con: binomial heaps do not adapt to the input data. You better not use binomial heaps for sorting.

7 Multiway binomial heaps

A data structure for priority queues records our partial knowledge about the ordering of its constituents. Different data structures give rise to different types of orderings: ordered lists, for example, correspond to chains, unordered lists to antichains. It is instructive to draw the underlying ordering of a pennant. The diagrams for pennants of height 2, 3, and 4 are shown on the right. A moment's reflection reveals that these *multiway trees* are obtained through the natural correspondence between binary trees and forests, described by Knuth (1968, p. 333–334). Figure 3 illustrates the special case of transforming a topped binary tree into a multiway tree.



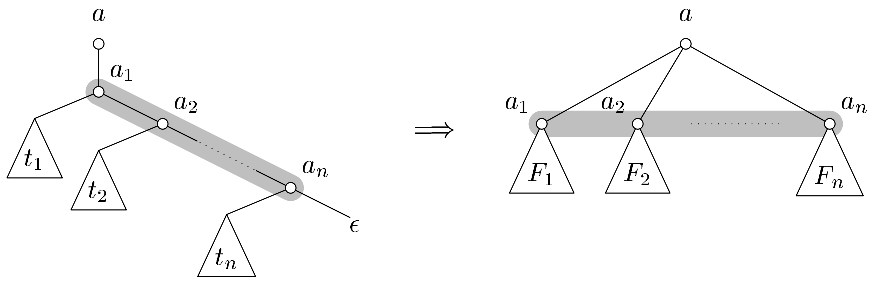


Fig. 3. Natural correspondence between topped binary trees and multiway trees (F_i is the forest corresponding to the binary tree t_i)

This correspondence suggests to implement priority queues directly by multiway trees. The data type for general trees is defined as follows.

```
data Tree a    =  Root a (Forest a) | Void
type Forest a  =  [Tree a]
```

Dictated by the application we allow that a tree is empty. We restrict, however, the use of *Void* to the top-level, ie *Void* must not appear beneath a *Root* node. The transformation pictured in Figure 3 can be rigorously defined by a Haskell function.

```
tree                ::  ToppedTree a → Tree a
tree Infty          =   Void
tree (Min a t)      =   Root a (forest t)

forest              ::  BinTree a → Forest a
forest Empty        =   []
forest (Bin a l r)  =   Root a (forest l) : forest r
```

If we apply *tree* to a pennant we obtain a *binomial tree*. This term is motivated by the fact that a binomial tree of height h contains $\binom{h}{d}$ nodes at depth d. Both, trees and coefficients, are based on a similar inductive scheme. A binomial tree of height $h+1$ consists of two trees of height h that are linked together: one is made the leftmost child of the other (see the definition of *carry* below). Binomial coefficients satisfy the recurrence $\binom{h+1}{d} = \binom{h}{d} + \binom{h}{d-1}$.

With regard to the ordering we have that a binomial tree satisfies the *heap property*: Every node is smaller or equal than any of its descendants.

Since *ToppedTree* and *Tree* are merely different representations of the same underlying structure it is not difficult to adapt the code of the preceding sections to multiway trees. We start by declaring binomial heaps.

$$\textbf{type}\ \mathit{BinomialHeap}\ a \quad = \quad \mathit{Forest}\ a$$

To be able to reuse *add* for melding we make *Tree a* an instance of *BinaryDigit*.

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Ord}\ a) \Rightarrow \mathit{BinaryDigit}\ (\mathit{Tree}\ a)\ \textbf{where}} \\
\quad \mathit{zero} & = & \mathit{Void} \\
\quad \multicolumn{2}{l}{\mathit{carry}\ t@(\mathit{Root}\ a\ \mathit{ts})\ u@(\mathit{Root}\ b\ \mathit{us})} \\
\qquad \mid a \leqslant b & = & \mathit{Root}\ a\ (u : \mathit{ts}) \\
\qquad \mid \mathit{otherwise} & = & \mathit{Root}\ b\ (t : \mathit{us}) \\
\quad \mathit{carry}\ _\ _ & = & \mathit{Void} \\
\quad \mathit{sum}\ \mathit{Void}\ u & = & u \\
\quad \mathit{sum}\ t\ \mathit{Void} & = & t \\
\quad \mathit{sum}\ (\mathit{Root}\ _\ _)\ (\mathit{Root}\ _\ _) & = & \mathit{Void}
\end{array}$$

Here is our final implementation of priority queues. For reasons of space we omit the *Eq* and *Ord* instance declarations for trees.

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{PriorityQueue}\ \mathit{BinomialHeap}\ \textbf{where}} \\
\quad \emptyset & = & [\,] \\
\quad \wr a \int & = & [\mathit{Root}\ a\ [\,]] \\
\quad (\uplus) & = & \mathit{add} \\
\quad \mathit{splitMin}\ q & = & \textbf{case}\ \mathit{extractMin}\ q\ \textbf{of} \\
\qquad \mathit{Infty} & \rightarrow & \mathit{Infty} \\
\qquad \mathit{Min}\ (\mathit{Root}\ a\ \mathit{ts})\ \mathit{us} & \rightarrow & \mathit{Min}\ a\ (\mathit{reverse}\ \mathit{ts} \uplus \mathit{us})
\end{array}$$

Having two different representations of the same structure we may now weigh up pros and cons of each. The binary coding of binomial heaps is more space economical. If we estimate the space usage of the term $C\ e_1\ \ldots\ e_k$ at $k+1$ cells, a binary heap of size $n \geqslant 1$ consumes $3 + 4(n-1)$ cells, whereas a multiway heap requires $3 + 6(n-1)$ cells. On the other hand linking and splitting is slightly faster with the second representation. Linking two pennants requires 7 cells and produces 6 cells of garbage whereas linking two binomial trees requires only 6 cells producing 3 cells of garbage. Furthermore, since binomial heaps use lists both for representing the subtrees of a tree and for the entire forest the dismantling step becomes superfluous. Practical experience shows that the advantages and disadvantages nearly counterbalance each other: binary heaps are only marginally faster. They win the race, however, due to the lower space requirements.

8 Further reading

Tournament trees and loser trees already appear in (Knuth, 1973). Binomial heaps were discovered by Vuillemin (1978). Readers interested in an average case analysis of insertion and deletion are referred to (Brown, 1978). Many variants and refinements of binomial heaps have been developed. An implicit array-based representation of binomial heaps (Carlsson *et al.*, 1988), for example, employs *heap-ordered* pennants. Binomial heaps also form the basis for an implementation of min-max priority queues (Khoong & Leong, 1993). We have seen that the loser trees in binary binomial heap are always perfectly balanced. Høyer (1994) shows that we may relax this condition and use some form of height-balancing instead.

The first functional implementation of binomial heaps is due to King (1994); (King, 1996, p. 28–42) additionally contains a simple proof of correctness. Okasaki (1996b) shows how to turn the amortized $\Theta(1)$ time bound for *insert* into a worst-case bound by scheduling delayed computations. Alternatively, one can employ a non-standard number system which avoids cascading carries: Such a variant based on skew binary numbers (Myers, 1983) is given in (Brodal & Okasaki, 1996). In fact, *loc.cit.* presents an optimal implementation of priority queues. Constant worst-case running time for $(\uplus)$ is achieved using a technique called data-structural bootstrapping (Buchsbaum, 1993). In essence $(\uplus)$ is reduced to *insert* by allowing queues to contain other queues.

9 Acknowledgements

Thanks are due to Michael Beetz, Thomas Bode, Martina Doelp, Ulrike Griefahn, Anja Hartmann, Jürgen Kalinski, and an anonymous referee for their helpful comments on a draft version of this article.

References

Brodal, Gerth Stølting, & Okasaki, Chris. (1996). Optimal purely functional priority queues. *Journal of functional programming*, **6**(6), 839–857.

Brown, Mark R. (1978). Implementation and analysis of binomial queue algorithms. *SIAM journal on computing*, **7**(3), 298–319.

Buchsbaum, Adam L. 1993 (June). *Data-structural bootstrapping and catenable deques.* Ph.d. thesis, Department of Computer Science, Princeton University.

Carlsson, Svante, Munro, J. Ian, & Poblete, Patricio V. (1988). An implicit binomial queue with constant insertion time. *Pages 1–13 of: Proceedings of 1st Scandinavian workshop on algorithm theory.* Lecture Notes in Computer Science, vol. 318. Springer-Verlag.

Cormen, Thomas H., Leiserson, Charles E., & Rivest, Ronald L. (1991). *Introduction to algorithms.* Cambridge, Massachusetts: The MIT Press.

Høyer, Peter. 1994 (October). *A general technique for implementation of efficient priority queues.* Tech. rept. PP-1994-33. Department of Mathematics and Computer Science, Odense University.

Jones, Mark P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.

Khoong, C. M., & Leong, H. W. (1993). Double-ended binomial queues. *Pages 128–137 of: Proceedings 4th international symposium ISAAC'93.* Lecture Notes in Computer Science, no. 762. Springer Verlag.

King, D. J. (1994). Functional binomial queues. Hammond, K., Turner, D. N., & Sansom, P. M. (eds), *Glasgow functional programming workshop*. Ayr, Scotland: Springer Verlag.

King, D.J. 1996 (March). *Functional programming and graph algorithms*. Ph.d. thesis, Department of Computer Science, University of Glasgow.

Knuth, Donald E. (1968). *The art of computer programming, Volume 1: Fundamental algorithms*. Addison-Wesley Publ. Comp., Inc.

Knuth, Donald E. (1973). *The art of computer programming, Volume 3: Sorting and searching*. Addison-Wesley Publ. Comp., Inc.

Myers, Eugene W. (1983). An applicative random-access stack. *Information processing letters*, **17**(5), 241–248.

Okasaki, Chris. 1996a (September). *Purely functional data structures*. Ph.d. thesis, School of Computer Science, Carnegie Mellon University.

Okasaki, Chris. 1996b (May). The role of lazy evaluation in amortized data structures. *Pages 62–72 of: ACM SIGPLAN international conference on functional programming*.

Peterson, J., & Hammond, K. 1997 (March). *Report on the programming language Haskell 1.4, a non-strict, purely functional language*. Research Report YALEU/DCS/RR-1106. Yale University, Department of Computer Science.

Sack, Jörg-Rüdiger, & Strothotte, Thomas. (1990). A characterization of heaps and its applications. *Information and computation*, **86**(1), 69–86.

Vuillemin, Jean. (1978). A data structure for manipulating priority queues. *Communications of the ACM*, **21**(4), 309–315.