A Simple Implementation Technique for Priority Search Queues

März 2001

RALF HINZE
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* `ralf@cs.uu.nl`)

Abstract

This paper presents a new implementation technique for priority search queues. This abstract data type is an amazing blend of finite maps and priority queues. Our implementation supports logarithmic access to a binding with a given key and constant access to a binding with the minimum value. Priority search queues can be used, for instance, to give a simple, purely functional implementation of Dijkstra's single source shortest-path algorithm.

A non-technical concern of the paper is to foster abstract data types and views. Priority search queues have been largely ignored by the functional programming community and we believe that they deserve to be known better. Views prove their worth both in defining a convenient interface to the abstract data type and in providing a readable implementation.

1 Introduction

The aim of this paper is threefold:

First, we would like to advertise *priority search queues*, a useful abstract data type that has been largely ignored by the functional programming community and that deserves to be known better. Priority search queues are an amazing blend of finite maps (or dictionaries) and priority queues, that is, they support both dictionary operations (for instance, accessing a binding with a given key) and priority queue operations (for instance, accessing a binding with the minimum value). We give two simple applications that demonstrate their usefulness: a purely functional implementation of Dijkstra's single-source shortest path algorithm and an efficient implementation of the first-fit heuristics for the bin packing problem.

Second, we describe a simple implementation technique for the abstract data type. The standard implementation of priority search queues, McCreight's priority search trees (1985), combines binary search trees and heaps. Unfortunately, balanced search trees and heaps do not go well together. Rotations that are typically used to maintain balance destroy the heap property and restoring the property takes $\Theta(h)$ time where h is the height of the tree. Consequently, in order to attain overall logarithmic time bounds the underlying balancing scheme must guarantee that the number of rotations per update is bounded by a constant. We show that it is possible to weaken the heap property so that rotations become constant time operations without sacrificing the running time of the priority queue methods. Thus, we can freely choose an underlying balancing scheme—we illustrate our approach using Adams's weight-balanced trees (1993).

Third, we would like to promote the use of *views*. Views have been introduced by Wadler (1987) to relieve the tension between pattern matching and abstraction. Briefly, views allow any type (in particular, any abstract data type) to be viewed as a free data type. We have found views not only useful for providing a convenient interface to an abstract data type but also extremely helpful in the implementation itself. The use of views made the code substantially clearer.

The remainder of this paper is structured as follows. Section 2 briefly reviews the concept of views. Section 3 introduces the abstract data type *priority search queue* and Section 4 illustrates its use. Section 5 provides a simple implementation based on unbalanced trees. Section 6 then shows how to augment the basic implementation by a balancing scheme. Section 7 analyses the running time of so-called range queries. Finally, Section 8 reviews related work and Section 9 concludes. Auxiliary types and functions that are used in the implementation are listed in Appendix A.

2 Preliminaries: views

The code in this paper is given in Haskell 98 (Peyton Jones & Hughes, 1999) augmented by the concept of views (Burton *et al.*, 1996; Okasaki, 1998b). This section briefly reviews Okasaki's proposal for views (1998b).

A view allows any type to be viewed as a free data type. A *view declaration* for a type T consists of an anonymous data type, the *view type*, and an anonymous function, the *view transformation*, that shows how to map elements of T to the view type. Here is a simple example that defines a *minimum view* on lists:

$$
\begin{array}{lcl}
\textbf{view}\ (\mathit{Ord}\ a) \Rightarrow [\,a\,] & = & \mathit{Empty} \mid \mathit{Min}\ a\ [\,a\,]\ \textbf{where} \\
\quad [\,] & \to & \mathit{Empty} \\
\quad a_1 : \mathit{Empty} & \to & \mathit{Min}\ a_1\ [\,] \\
\quad a_1 : \mathit{Min}\ a_2\ \mathit{as} & & \\
\quad\quad \mid a_1 \leqslant a_2 & \to & \mathit{Min}\ a_1\ (a_2 : \mathit{as}) \\
\quad\quad \mid \mathit{otherwise} & \to & \mathit{Min}\ a_2\ (a_1 : \mathit{as}).
\end{array}
$$

This declaration introduces two constructors, *Empty* and *Min*, that henceforth can be used to pattern match elements of type $[\,a\,]$, where the context '$(\mathit{Ord}\ a) \Rightarrow$' restricts a to instances of *Ord*. The minimum view allows any list to be viewed as an ordered list. The following definition of selection sort nicely illustrates the use of views:

$$
\begin{array}{lcl}
\mathit{selection\text{-}sort} & :: & (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to [\,a\,] \\
\mathit{selection\text{-}sort}\ \mathit{Empty} & = & [\,] \\
\mathit{selection\text{-}sort}\ (\mathit{Min}\ a\ \mathit{as}) & = & a : \mathit{selection\text{-}sort}\ \mathit{as}.
\end{array}
$$

The view constructors can be freely mixed with ordinary data type constructors. In fact, the view transformation of the minimum view already illustrates nested patterns. A type can even have multiple views. However, view constructors may only appear in patterns—with the notable exception of the view transformation itself.

View declarations can be implemented by a simple source to source translation: each view is expanded into a data type and a function. For the minimum view we obtain:

$$
\begin{array}{lll}
\mathbf{data}\ \mathit{Min\text{-}View}\ a & = & \mathit{Empty} \mid \mathit{Min}\ a\ [a] \\
\mathit{min\text{-}view} & :: & (\mathit{Ord}\ a) \Rightarrow [a] \to \mathit{Min\text{-}View}\ a \\
\mathit{min\text{-}view}\ x_1 & = & \mathbf{case}\ x_1\ \mathbf{of} \\
\quad [\,] & \to & \mathit{Empty} \\
\quad a_1 : x_2 & \to & \mathbf{case}\ \mathit{min\text{-}view}\ x_2\ \mathbf{of} \\
\qquad \mathit{Empty} & \to & \mathit{Min}\ a_1\ [\,] \\
\qquad \mathit{Min}\ a_2\ \mathit{as} & & \\
\qquad\quad \mid a_1 \leqslant a_2 & \to & \mathit{Min}\ a_1\ (a_2 : \mathit{as}) \\
\qquad\quad \mid \mathit{otherwise} & \to & \mathit{Min}\ a_2\ (a_1 : \mathit{as}).
\end{array}
$$

The function is invoked whenever constructors of the view appear in patterns. In our example, the view constructors appear in the view transformation itself. Consequently, it is expanded into a recursive function. Selection sort becomes:

$$
\begin{array}{lll}
\mathit{selection\text{-}sort}\ x & = & \mathbf{case}\ \mathit{minView}\ x\ \mathbf{of} \\
\quad \mathit{Empty} & \to & [\,] \\
\quad \mathit{Min}\ a\ \mathit{as} & \to & a : \mathit{selection\text{-}sort}\ \mathit{as}.
\end{array}
$$

For a precise definition of the semantics we refer the interested reader to Okasaki's paper (1998b)—the proposal is for Standard ML but it can be easily adapted to Haskell 98.

3 Priority search queues

The abstract data type *priority search queue* is conceptually a finite map that supports efficient access to the binding with the minimum value, where a *binding* is an argument-value pair and a *finite map* is a finite set of bindings. For emphasis, we call the arguments *keys* and the associated values *priorities*. Bindings are represented by the following data type:

$$
\begin{array}{lll}
\mathbf{data}\ k \mapsto p & = & k \mapsto p \\
\mathit{key} & :: & (k \mapsto p) \to k \\
\mathit{key}\ (k \mapsto p) & = & k \\
\mathit{prio} & :: & (k \mapsto p) \to p \\
\mathit{prio}\ (k \mapsto p) & = & p.
\end{array}
$$

Note that we use '$\mapsto$' both as a type constructor and as a value constructor. The functions *key* and *prio* provide access to the key and to the priority of a binding. Like the type of bindings, the abstract data type of priority search queues is parametric in the types of keys and priorities:

$$\mathbf{data}\ \mathit{PSQ}\ k\ p.$$

Most operations on priority search queues require that both k and p are totally ordered. This condition is expressed in Haskell by the context '$(Ord\ k, Ord\ p) \Rightarrow$'. However, in the rest of this section we will omit the context to reduce clutter.

Priority search queues support both finite map and priority queue operations plus so-called *range queries*.

Constructors and insertion

$$\begin{array}{lll}
\emptyset & :: & PSQ\ k\ p \\
\{\cdot\} & :: & (k \mapsto p) \to PSQ\ k\ p \\
insert & :: & (k \mapsto p) \to PSQ\ k\ p \to PSQ\ k\ p \\
from\text{-}ord\text{-}list & :: & [k \mapsto p] \to PSQ\ k\ p
\end{array}$$

The constructor $\emptyset$ represents the empty queue; $\{b\}$ creates a queue that contains b as the single binding; $insert\ b\ q$ inserts binding b into q (if the queue contains a binding with the same key, then the old binding is overwritten); and *from-ord-list* converts a list of bindings into a queue with the precondition that the list is sorted into increasing order by key.

Destructors and deletion

$$\begin{array}{lll}
\mathbf{view}\ PSQ\ k\ p & = & Empty \mid Min\ (k \mapsto p)\ (PSQ\ k\ p) \\
delete & :: & k \to PSQ\ k\ p \to PSQ\ k\ p
\end{array}$$

A queue is destructed using the patterns *Empty* and $Min\ b\ q$ introduced by the view declaration. The function *delete* removes a binding with the given key (the queue is left unchanged if it does not contain a binding with the key). The constructors of the *minimum view* have the following meaning: if a queue pattern matches *Empty*, then it is empty; otherwise it matches $Min\ b\ q$ where b is the binding with the minimum priority and q is the remaining queue. Thus, using the view we can effectively treat a priority search queue as a list of bindings ordered by priority.

Observers

$$\begin{array}{lll}
lookup & :: & k \to PSQ\ k\ p \to Maybe\ p \\
to\text{-}ord\text{-}list & :: & PSQ\ k\ p \to [k \mapsto p] \\
at\text{-}most & :: & p \to PSQ\ k\ p \to [k \mapsto p] \\
at\text{-}most\text{-}range & :: & p \to (k, k) \to PSQ\ k\ p \to [k \mapsto p]
\end{array}$$

The function *lookup* finds the priority associated with a given key: the call $lookup\ k\ q$ returns *Nothing* if the queue does not contain the key k; otherwise it yields $Just\ p$ where p is the priority associated with k. The function *to-ord-list* converts a queue into a list of bindings ordered by key. Priority search queues not only support dictionary and priority queue operations. As a little extra they also allow for so-called *range queries*: $at\text{-}most\ p_t\ q$ returns a list of bindings ordered by key whose priorities are at most p_t; $at\text{-}most\text{-}range\ p_t\ (k_l, k_r)\ q$ returns a list of bindings ordered by key whose priorities are at most p_t and whose keys lie between k_l and k_r. If we interpret keys and priorities as x- and y-coordinates, then *at-most-range* implements a $1\frac{1}{2}$-dimensional query. It is only $1\frac{1}{2}$-dimensional, because we can only specify semi-infinite ranges for the y-coordinate.

Modifier

$$adjust :: (p \to p) \to k \to PSQ\ k\ p \to PSQ\ k\ p$$

The function *adjust* changes a binding for the given key by applying the function to its priority (the queue is left unchanged if it does not contain a binding with the key).

4 Applications

4.1 Single-source shortest path

Dijkstra's algorithm for the single-source shortest-paths problem serves as a nice example for the use of priority search queues. The algorithm maintains a queue that maps each vertex to its estimated distance from the source. The algorithm works by repeatedly removing the vertex with minimal distance and updating the distances of its adjacent vertices. Priority search queues support both operations equally well. The update operation is typically called *decrease*:

$$\begin{array}{lcl}
decrease & :: & (Ord\ k, Ord\ p) \Rightarrow (k \mapsto p) \to PSQ\ k\ p \to PSQ\ k\ p \\
decrease\ (k \mapsto p)\ q & = & adjust\ (min\ p)\ k\ q \\
decrease\text{-}list & :: & (Ord\ k, Ord\ p) \Rightarrow [k \mapsto p] \to PSQ\ k\ p \to PSQ\ k\ p \\
decrease\text{-}list\ bs\ q & = & foldr\ decrease\ q\ bs.
\end{array}$$

Note that $decrease\ (k \mapsto p)\ q$ has no effect if k's priority in q is less than p.

To keep the presentation terse we assume that the following functions on graphs are provided from somewhere.

$$\begin{array}{lcl}
vertices & :: & Graph \to [\,Vertex\,] \\
adjacent & :: & Graph \to Vertex \to [\,Vertex\,]
\end{array}$$

The function *vertices* returns an ordered list of all vertices of a graph; *adjacent* produces a list of vertices adjacent to the given one.

The function *dijkstra* defined below takes three arguments: a directed graph, a weight function, and a source vertex. It returns a list of vertex-distance bindings that determine the minimal distance of each vertex from the source.

$$\begin{array}{lcl}
\textbf{type}\ Weight & = & Vertex \to Vertex \to Double \\
dijkstra & :: & Graph \to Weight \to Vertex \to [\,Vertex \mapsto Double\,] \\
dijkstra\ g\ w\ s & = & loop\ (decrease\ (s \mapsto 0)\ q_0) \\
\quad \textbf{where} & & \\
\quad q_0 & = & from\text{-}ord\text{-}list\ [\,v \mapsto +\infty \mid v \leftarrow vertices\ g\,] \\
\quad loop\ Empty & = & [\,] \\
\quad loop\ (Min\ (u \mapsto d)\ q) & = & (u \mapsto d) : loop\ (decrease\text{-}list\ bs\ q) \\
\qquad \textbf{where}\ bs & = & [\,v \mapsto d + w\ u\ v \mid v \leftarrow adjacent\ g\ u\,]
\end{array}$$

The helper function *loop* uses the minimum view to process the queue. Note that the computed list of vertex-distance bindings may contain bindings with priority $+\infty$, which indicates that the given graph was not strongly connected. Now, if we assume that the computation of the view and the *decrease* operation each take $\Theta(\log V)$ time, then the algorithm has a worst-case running time of $\Theta((V+E)\log V)$, which is the best known running time for purely functional implementations.

Remark 1
If we modify the computation of the new distances as follows

$$\ldots\ \textbf{where}\ bs\ =\ [v \mapsto w\ u\ v \mid v \leftarrow adjacent\ g\ u],$$

we obtain Prim's algorithm for computing a minimum spanning tree. □

4.2 One-dimensional bin packing

As the second example we employ priority search queues to implement the first-fit heuristics for the bin packing problem. Recall that the standard list-based implementation shown below has a worst-case running time of $\Theta(n^2)$ where n is the number of items.

$$
\begin{array}{lcl}
\textit{pack-first-fit} & :: & [\textit{Item}] \to [\textit{Bin}] \\
\textit{pack-first-fit} & = & \textit{foldl first-fit}\,[\,] \\
\textit{first-fit} & :: & [\textit{Bin}] \to \textit{Item} \to [\textit{Bin}] \\
\textit{first-fit}\,[\,]\ i & = & [i] \\
\textit{first-fit}\,(b : bs)\ i & & \\
\quad \mid b + i \leqslant 1 & = & b + i : bs \\
\quad \mid \textit{otherwise} & = & b : \textit{first-fit}\ bs\ i
\end{array}
$$

The function *pack-first-fit* takes a list of items, each of a certain size, and returns a list of bins that contain the input items. For simplicity, we represent an item by its size and a bin by its total size (each bin has a capacity of 1).

Using priority search queues we can improve the running time of the naïve implementation to $\Theta(n \log n)$. The central idea is to use the function *at-most* to quickly determine the first bin that can accommodate a given item (the bins are numbered consecutively).

$$
\begin{array}{lcl}
\textbf{type}\ \textit{No} & = & \textit{Int} \\
\textit{pack-first-fit} & :: & [\textit{Item}] \to [\textit{Bin}] \\
\textit{pack-first-fit is} & = & [\textit{prio}\ b \mid b \leftarrow \textit{to-ord-list}\ q] \\
\quad \textbf{where}\ (q, \_) & = & \textit{foldl first-fit}\,(\emptyset, 0)\ \textit{is} \\
\textit{first-fit} & :: & (\textit{PSQ No Bin}, \textit{No}) \to \textit{Item} \to (\textit{PSQ No Bin}, \textit{No}) \\
\textit{first-fit}\,(q, n)\ i & = & \textbf{case}\ \textit{at-most}\,(1 - i)\ q\ \textbf{of} \\
 & & \quad [\,] \to (\textit{insert}\ (n \mapsto i)\ q, n + 1) \\
 & & \quad (k \mapsto \_) : \_ \to (\textit{adjust}\ (+i)\ k\ q, n)
\end{array}
$$

This is the only place where essential use is made of Haskell's non-strict semantics as we merely require the first element of the list returned by *at-most*. In a strict language, we would be forced to define a specialized version of *at-most* that computes the first binding only (if any).

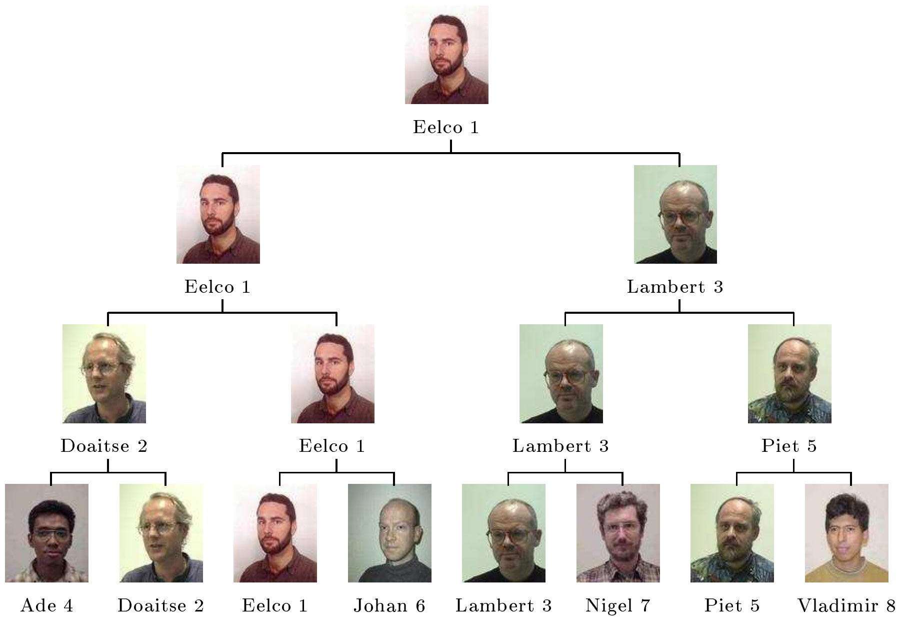


Fig. 1. A tournament tree.

5 Priority search pennants

This section describes an implementation of priority search queues based on unbalanced search trees. Great care has been taken to modularize the code so that a balancing scheme can be added later with ease (Section 6 discusses the necessary amendments). It should be noted, however, that the implementation in this section is perfectly suitable for Dijkstra's or Prim's algorithm since both do not require insertions.

The underlying idea of the implementation is best explained using the metaphor of a knockout tournament. Consider the tournament depicted in Figure 1. We have eight participants, so the course of matches forms a complete binary tree. Each external node corresponds to a participant; each internal node corresponds to a winner of a match. To facilitate searching the participants are arranged from left to right in increasing order by name. Tournament trees are almost a suitable data structure for priority search queues if it were not for the many repeated entries. The champion, for instance, appears on every level of the tree. Now, there are at least two ways to repair this defect.

One possibility is to promote losers up the tree turning the tournament tree of Figure 1 into the heap-structured tree of Figure 2. This transformation usually involves additional matches. In our example, Doaitse has to play with Johan to determine the second-best player of the first half of the tournament. Pursuing this idea further leads to a data structure known as a *priority search tree* (McCreight, 1985). We will come back to this data structure in Sections 7 and 8.

An alternative possibility, which we will investigate in this section, is to label each internal node with the loser of the match, instead of the winner, and to drop the external nodes altogether. If we additionally place the champion on top of the tree, we obtain the topped loser tree of Figure 3. We call the resulting data structure *priority search pennant*. Since every participant—with the notable exception of the champion—loses exactly one match, the pennant does not contain repeated entries. It is important to note, however, that the loser tree is not heap-structured. Since the nodes are labelled with losers, they dominate, in general, only one subtree. The node labelled Lambert, for instance, dominates its right but not its left subtree. Thus the loser tree constitutes only a so-called *semi-heap*.

The Haskell data type for priority search pennants is a direct implementation of the above ideas except that we additionally introduce split keys to support searching.

$$\begin{aligned}
&\mathbf{data}\ \mathit{PSQ}\ k\ p && = \mathit{Void} \mid \mathit{Winner}\ (k \mapsto p)\ (\mathit{LTree}\ k\ p)\ k \\
&\mathbf{data}\ \mathit{LTree}\ k\ p && = \mathit{Start} \mid \mathit{Loser}\ (k \mapsto p)\ (\mathit{LTree}\ k\ p)\ k\ (\mathit{LTree}\ k\ p)
\end{aligned}$$

Here, *Void* represents the empty tournament; $\mathit{Winner}\ b\ t\ m$ represents a tournament that b has won, t is the associated loser tree and m is the maximum key. Likewise, *Start* is the empty loser tree; $\mathit{Loser}\ b\ t_l\ k\ t_r$ represents a subtournament that b has lost, t_l is the left subtree, k is the split key, and t_r is the right subtree. The maximum key is usually accessed using the function *max-key*.

$$\begin{aligned}
&\mathit{max\text{-}key} && :: \mathit{PSQ}\ k\ p \to k \\
&\mathit{max\text{-}key}\ (\mathit{Winner}\ b\ t\ m) && = m
\end{aligned}$$

We will see in Section 5.1 why it is useful to keep track of the maximum key.

Priority search pennants combine the features of search trees and semi-heaps. To formulate the invariants, it is convenient to view the top node $\mathit{Winner}\ b\ t\ m$ as a binary node with an empty right subtree so that the maximum key becomes an ordinary split key ($\mathit{Winner}\ b\ t\ m \cong \mathit{Loser}\ b\ t\ m\ \mathit{Start}$).

Semi-heap conditions: 1) Every priority in the pennant must be less than or equal to the priority of the winner. 2) For all nodes in the loser tree, the priority of the loser's binding must be less than or equal to the priorities of the bindings of the subtree, from which the loser originates. The loser *originates* from the left subtree if its key is less than or equal to the split key, otherwise it originates from the right subtree.

Search-tree condition: For all nodes, the keys in the left subtree must be less than or equal to the split key and the keys in the right subtree must be greater than the split key.

Key condition: The maximum key and the split keys must also occur as keys of bindings.

Finite map condition: The pennant must not contain two bindings with the same key.

Two remarks are in order. First, the second semi-heap condition shows that a priority search pennant contains enough information to reconstruct the original tournament tree. This ability is crucial for implementing the priority queue operations.

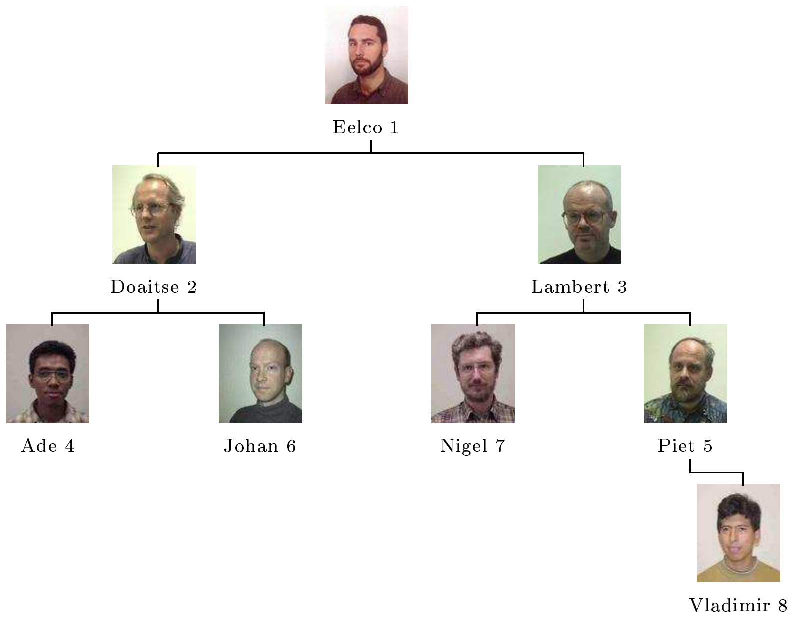


Fig. 2. The heap corresponding to the tournament of Figure 1.

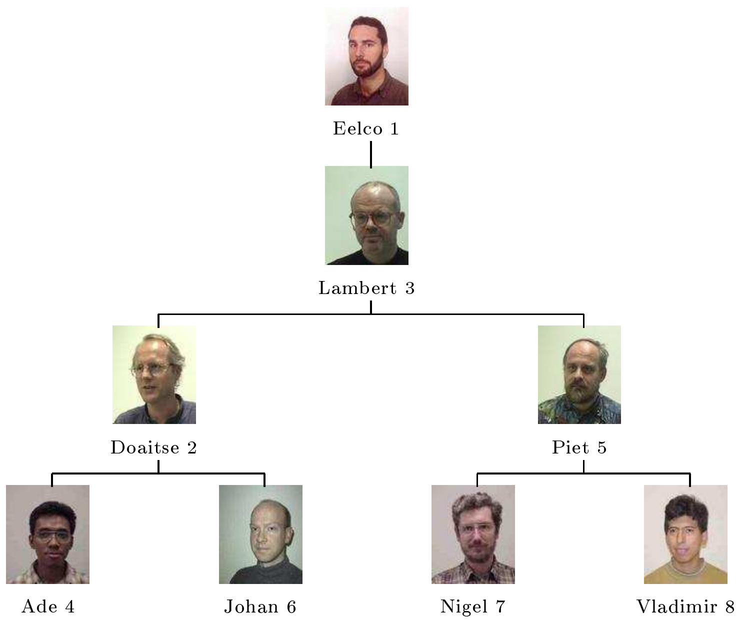


Fig. 3. The semi-heap corresponding to the tournament of Figure 1.

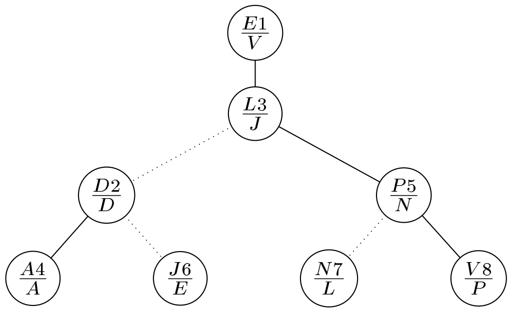


Fig. 4. The priority search pennant corresponding to the tree of Figure 3.

Second, the key condition ensures that every search key originates from a binding in the tree. This means, in particular, that if we delete a binding from a tree, we must also delete the key's second occurrence as a search key. We will see that it is relatively easy to maintain this invariant.

Let us consider an example. If we augment the tree of Figure 3 by split keys, we obtain the priority search pennant depicted in Figure 4. Note that the dotted lines mark the subtrees that are *not* dominated by the loser. As we have remarked before, the semi-heap structure can also be determined by comparing the loser's key to the split key: the node labelled Lambert, for instance, dominates its right subtree since $L > J$; the node labelled Doaitse on the other hand dominates its left subtree since $D \leqslant D$. The pennant can quite easily be expressed as a Haskell term:

```
Winner (E ↦ 1) (
    Loser (L ↦ 3) (
        Loser (D ↦ 2) (
            Loser (A ↦ 4) Start A Start)
            D (
            Loser (J ↦ 6) Start E Start))
        J (
        Loser (P ↦ 5) (
            Loser (N ↦ 7) Start L Start)
            N (
            Loser (V ↦ 8) Start P Start)))
    V.
```

Note that if we list the search keys from left to right, we obtain the keys of the participants in increasing order.

Remark 2
The nodes are decorated with bindings of type $k \mapsto p$. While this is convenient for the presentation, it comes at a small run-time cost since every access involves one extra level of indirection. In the production code, which is available from `http://www.cs.uu.nl/~ralf/software`, we speed up the access by storing the keys and the priorities directly in the nodes. □

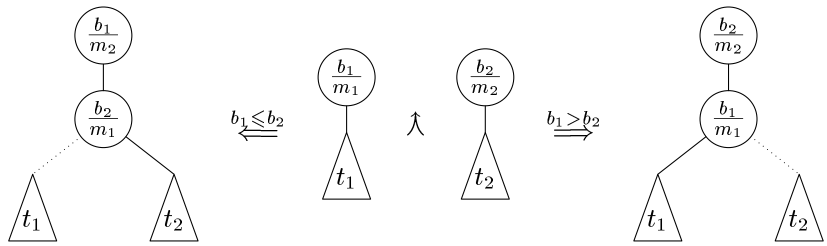


Fig. 5. Playing a match ($b_1 \leqslant b_2$ is shorthand for *prio* $b_1 \leqslant$ *prio* b_2).

5.1 Constructors

The empty queue and the singleton queue are defined as follows:

$$\begin{array}{lcl}
\emptyset & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p \\
\emptyset & = & \mathit{Void} \\
\{\cdot\} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow (k \mapsto p) \to \mathit{PSQ}\ k\ p \\
\{b\} & = & \mathit{Winner}\ b\ \mathit{Start}\ (\mathit{key}\ b).
\end{array}$$

The data types *PSQ* and *LTree* have been designed to efficiently support the binary operation ($\curlywedge$), which corresponds to playing a match. This operation, which is used by most of the remaining functions, takes two pennants and returns a new pennant that is the union of the two with the precondition that the keys in the first tree are strictly smaller than the keys in the second tree. The operation is illustrated in Figure 5.

$$\begin{array}{lcl}
(\curlywedge) & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p \to \mathit{PSQ}\ k\ p \to \mathit{PSQ}\ k\ p \\
\mathit{Void} \curlywedge t' & = & t' \\
t \curlywedge \mathit{Void} & = & t \\
\multicolumn{3}{l}{\mathit{Winner}\ b\ t\ m \curlywedge \mathit{Winner}\ b'\ t'\ m'} \\
\quad \mid \mathit{prio}\ b \leqslant \mathit{prio}\ b' & = & \mathit{Winner}\ b\ (\mathit{Loser}\ b'\ t\ m\ t')\ m' \\
\quad \mid \mathit{otherwise} & = & \mathit{Winner}\ b'\ (\mathit{Loser}\ b\ t\ m\ t')\ m'
\end{array}$$

Note that in order to construct the loser tree we require a split key, which is why we keep track of the maximum key in the top node. This makes '$\curlywedge$' a constant-time operation. It is not hard to see that '$\curlywedge$' preserves the invariants of priority search pennants. Using '$\curlywedge$' we can easily define *from-ord-list*.

$$\begin{array}{lcl}
\mathit{from\text{-}ord\text{-}list} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow [k \mapsto p] \to \mathit{PSQ}\ k\ p \\
\mathit{from\text{-}ord\text{-}list} & = & \mathit{foldm}\ (\curlywedge)\ \emptyset \cdot \mathit{map}\ (\lambda b \to \{b\})
\end{array}$$

The helper function *foldm*, which is listed in the Appendix, folds a list in a binary-sub-division fashion. For instance,

$$\mathit{from\text{-}ord\text{-}list}\,[A \mapsto 4, D \mapsto 2, E \mapsto 1, J \mapsto 6, L \mapsto 3, N \mapsto 7, P \mapsto 5, V \mapsto 8]$$

reduces to

$$\begin{array}{l}
((\{A \mapsto 4\} \curlywedge \{D \mapsto 2\}) \curlywedge (\{E \mapsto 1\} \curlywedge \{J \mapsto 6\})) \\
\quad \curlywedge ((\{L \mapsto 3\} \curlywedge \{N \mapsto 7\}) \curlywedge (\{P \mapsto 5\} \curlywedge \{V \mapsto 8\})),
\end{array}$$

which in turn evaluates to the tree of Figure 4. In general, the expression tree generated by *foldm* takes the form of a Braun tree (Braun & Rem, 1983). Since '$⅄$' preserves the shape of the expression tree, the priority search pennant produced by *from-ord-list* corresponds to a topped Braun tree. This means, in particular, that the shape is solely determined by the total number of participants (and not by their priorities).

5.2 Destructors

The minimum view is implemented as follows:

$$
\begin{array}{lcl}
\textbf{view}\ (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p & = & \mathit{Empty} \mid \mathit{Min}\ (k \mapsto p)\ (\mathit{PSQ}\ k\ p)\ \textbf{where} \\
\quad \mathit{Void} & \rightarrow & \mathit{Empty} \\
\quad \mathit{Winner}\ b\ t\ m & \rightarrow & \mathit{Min}\ b\ (\mathit{second\text{-}best}\ t\ m).
\end{array}
$$

The function *second-best* used in the second clause determines the second-best player by replaying the tournament without the champion.

$$
\begin{array}{lcl}
\mathit{second\text{-}best} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{LTree}\ k\ p \rightarrow k \rightarrow \mathit{PSQ}\ k\ p \\
\mathit{second\text{-}best}\ \mathit{Start}\ m & = & \mathit{Void} \\
\mathit{second\text{-}best}\ (\mathit{Loser}\ b\ t\ k\ u)\ m & & \\
\quad \mid \mathit{key}\ b \leqslant k & = & \mathit{Winner}\ b\ t\ k \curlywedge \mathit{second\text{-}best}\ u\ m \\
\quad \mid \mathit{otherwise} & = & \mathit{second\text{-}best}\ t\ k \curlywedge \mathit{Winner}\ b\ u\ m
\end{array}
$$

Note that only those players who lost to the champion are taken into account. The origin of the champion is determined by comparing the loser's key to the split key.

Again, it is straightforward to see that *second-best* preserves the invariants except perhaps for the key condition: does *second-best* also remove the search key of the champion? This is most easily shown if we define *second-best* on pennants instead of loser trees (we call this variant *delete-min*).

$$
\begin{array}{lcl}
\mathit{delete\text{-}min} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p \rightarrow \mathit{PSQ}\ k\ p \\
\mathit{delete\text{-}min}\ \mathit{Void} & = & \mathit{Void} \\
\mathit{delete\text{-}min}\ (\mathit{Winner}\ b\ \mathit{Start}\ m) & & \\
 & = & \mathit{Void} \\
\mathit{delete\text{-}min}\ (\mathit{Winner}\ b\ (\mathit{Loser}\ b'\ t\ k\ u)\ m) & & \\
\quad \mid \mathit{key}\ b' \leqslant k & = & \mathit{Winner}\ b'\ t\ k \curlywedge \mathit{delete\text{-}min}\ (\mathit{Winner}\ b\ u\ m) \\
\quad \mid \mathit{otherwise} & = & \mathit{delete\text{-}min}\ (\mathit{Winner}\ b\ t\ k) \curlywedge \mathit{Winner}\ b'\ u\ m
\end{array}
$$

Since the argument of *delete-min* is always a legal pennant, m must equal $\mathit{key}\ b$ in the second equation by virtue of the key condition. Furthermore, we know that b is the champion, since the champion is passed unchanged to the recursive calls. The function *second-best* can now be seen as a simple optimization: we have

$$\mathit{delete\text{-}min}\ (\mathit{Winner}\ b\ t\ m) \ = \ \mathit{second\text{-}best}\ t\ m.$$

Remark 3

When we replay a tournament we determine the origin of a loser by comparing the loser's key to the split key ($\mathit{key}\ b \leqslant k$). Instead of using this perhaps costly comparison, we can alternatively code the information into the constructors when building the tree:

$$\begin{array}{lcl}\textbf{data}\ \mathit{LTree}\ k\ p & = & \mathit{Start} \\ & \mid & \mathit{LLoser}\ (k \mapsto p)\ (\mathit{LTree}\ k\ p)\ k\ (\mathit{LTree}\ k\ p) \\ & \mid & \mathit{RLoser}\ (k \mapsto p)\ (\mathit{LTree}\ k\ p)\ k\ (\mathit{LTree}\ k\ p).\end{array}$$

This is, in fact, the representation we use in the production code. The original representation, however, is slightly easier to augment by a balancing scheme. □

5.3 Observers

Views are not only convenient for the client of an abstract data type. They can also be tremendously helpful when implementing an abstract data type. The following declaration allows us to view a pennant as a tournament tree.

$$\begin{array}{l}\textbf{view}\ (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p\ =\ \emptyset \mid \{k \mapsto p\} \mid \mathit{PSQ}\ k\ p \curlywedge \mathit{PSQ}\ k\ p \\ \quad \textbf{where} \\ \quad \begin{array}{lcl}\mathit{Void} & \rightarrow & \emptyset \\ \mathit{Winner}\ b\ \mathit{Start}\ m & \rightarrow & \{b\} \\ \mathit{Winner}\ b\ (\mathit{Loser}\ b'\ t_l\ k\ t_r)\ m & & \\ \quad \mid \mathit{key}\ b' \leqslant k & \rightarrow & \mathit{Winner}\ b'\ t_l\ k \curlywedge \mathit{Winner}\ b\ t_r\ m \\ \quad \mid \mathit{otherwise} & \rightarrow & \mathit{Winner}\ b\ t_l\ k \curlywedge \mathit{Winner}\ b'\ t_r\ m\end{array}\end{array}$$

Note that we have taken the liberty of using $\emptyset$, $\{\cdot\}$ and '$\curlywedge$' also as constructors. There is little danger of confusion since the constructors of the view may only appear in patterns—with the notable exception of the view transformation itself—while the functions of the same name may only appear in expressions. The view transformation is essentially the inverse of the '$\curlywedge$' operation. In particular, if a winner tree matches $t_l \curlywedge t_r$, then it is guaranteed that the keys in t_l are strictly smaller than the keys in t_r. Furthermore, both t_l and t_r are non-empty.

The function *to-ord-list*, which converts a queue into a list of bindings ordered by key, nicely illustrates the use of the tournament view.[1]

$$\begin{array}{lcl}\mathit{to\text{-}ord\text{-}list} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p \rightarrow [k \mapsto p] \\ \mathit{to\text{-}ord\text{-}list}\ \emptyset & = & [\,] \\ \mathit{to\text{-}ord\text{-}list}\ \{b\} & = & [b] \\ \mathit{to\text{-}ord\text{-}list}\ (t_l \curlywedge t_r) & = & \mathit{to\text{-}ord\text{-}list}\ t_l \mathbin{+\!\!+} \mathit{to\text{-}ord\text{-}list}\ t_r\end{array}$$

In the last clause we rely on the fact that the keys in t_l precede the keys in t_r.

It is instructive to rewrite the definition of *to-ord-list* into a form that does not make use of views. We will see that the resulting code is much harder to read. On the other hand, the rewrite opens the possibility of small improvements (which

[1] Due to the use of $(+\!\!+)$ the definition of *to-ord-list* exhibits $\Theta(n^2)$ worst-case behaviour. This is, however, easily remedied using standard techniques.

a good optimizing compiler might be able to perform automatically). As the first step, we fuse the view transformation and the original function:

$$
\begin{array}{lll}
\mathit{to\text{-}ord\text{-}list} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p \to [k \mapsto p] \\
\mathit{to\text{-}ord\text{-}list}\ \mathit{Void} & = & [\,] \\
\mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b\ \mathit{Start}\ m) & & \\
 & = & [b] \\
\mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b\ (\mathit{Loser}\ b'\ t_l\ k\ t_r)\ m) & & \\
\quad \mid \mathit{key}\ b' \leqslant k & = & \mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b'\ t_l\ k) +\!\!+\ \mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b\ t_r\ m) \\
\quad \mid \mathit{otherwise} & = & \mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b\ t_l\ k) +\!\!+\ \mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b'\ t_r\ m).
\end{array}
$$

Note that in each of the recursive calls *to-ord-list* is passed a non-empty winner tree. Furthermore, the maximum key and the split keys are never used. This suggests specializing $\mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b\ t\ m)$ to $\mathit{traverse}\ b\ t$:

$$
\begin{array}{lll}
\mathit{to\text{-}ord\text{-}list} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow \mathit{PSQ}\ k\ p \to [k \mapsto p] \\
\mathit{to\text{-}ord\text{-}list}\ \mathit{Void} & = & [\,] \\
\mathit{to\text{-}ord\text{-}list}\ (\mathit{Winner}\ b\ t\ m) & = & \mathit{traverse}\ b\ t \\
\mathit{traverse} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow (k \mapsto p) \to \mathit{LTree}\ k\ p \to [k \mapsto p] \\
\mathit{traverse}\ b\ \mathit{Start} & = & [b] \\
\mathit{traverse}\ b\ (\mathit{Loser}\ b'\ t_l\ k\ t_r) & & \\
\quad \mid \mathit{key}\ b' \leqslant k & = & \mathit{traverse}\ b'\ t_l +\!\!+\ \mathit{traverse}\ b\ t_r \\
\quad \mid \mathit{otherwise} & = & \mathit{traverse}\ b\ t_l +\!\!+\ \mathit{traverse}\ b'\ t_r.
\end{array}
$$

Most of the following functions can be optimized along these lines.

The look-up function is very similar to the look-up function for binary search trees. Again, the tournament view allows for a very natural implementation.

$$
\begin{array}{lll}
\mathit{lookup} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow k \to \mathit{PSQ}\ k\ p \to \mathit{Maybe}\ p \\
\mathit{lookup}\ k\ \emptyset & = & \mathit{Nothing} \\
\mathit{lookup}\ k\ \{b\} & & \\
\quad \mid k == \mathit{key}\ b & = & \mathit{Just}\ (\mathit{prio}\ b) \\
\quad \mid \mathit{otherwise} & = & \mathit{Nothing} \\
\mathit{lookup}\ k\ (t_l \curlywedge t_r) & & \\
\quad \mid k \leqslant \mathit{max\text{-}key}\ t_l & = & \mathit{lookup}\ k\ t_l \\
\quad \mid \mathit{otherwise} & = & \mathit{lookup}\ k\ t_r
\end{array}
$$

The running time of *lookup* is proportional to the height of the tree even if we search for a binding that is high up in the tree. This observation suggests to additionally test the bindings on the search path at the cost of one additional comparison per recursive call. Of course, this change neither affects the worst-case nor the average-case running time.

$$
\begin{array}{lcl}
\mathit{lookup}' & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow k \rightarrow \mathit{PSQ}\ k\ p \rightarrow \mathit{Maybe}\ p \\
\mathit{lookup}'\ k\ (\mathit{Min}\ b\ q) & & \\
\quad \mid k == \mathit{key}\ b & = & \mathit{Just}\ (\mathit{prio}\ b) \\
\mathit{lookup}'\ k\ \emptyset & = & \mathit{Nothing} \\
\mathit{lookup}'\ k\ \{b\} & = & \mathit{Nothing} \quad \text{-- we know that } k \neq \mathit{key}\ b \\
\mathit{lookup}'\ k\ (t_l \curlywedge t_r) & & \\
\quad \mid k \leqslant \mathit{max\text{-}key}\ t_l & = & \mathit{lookup}'\ k\ t_l \\
\quad \mid \mathit{otherwise} & = & \mathit{lookup}'\ k\ t_r
\end{array}
$$

Note that this version of the look-up function uses both the minimum and the tournament view.

5.4 Modifier, insertion, and deletion

The dictionary functions *adjust*, *insert*, and *delete* can be most easily implemented using the tournament view.

$$
\begin{array}{lcl}
\mathit{adjust} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow (p \rightarrow p) \rightarrow k \rightarrow \mathit{PSQ}\ k\ p \rightarrow \mathit{PSQ}\ k\ p \\
\mathit{adjust}\ f\ k\ \emptyset & = & \emptyset \\
\mathit{adjust}\ f\ k\ \{b\} & & \\
\quad \mid k == \mathit{key}\ b & = & \{k \mapsto f\ (\mathit{prio}\ b)\} \\
\quad \mid \mathit{otherwise} & = & \{b\} \\
\mathit{adjust}\ f\ k\ (t_l \curlywedge t_r) & & \\
\quad \mid k \leqslant \mathit{max\text{-}key}\ t_l & = & \mathit{adjust}\ f\ k\ t_l \curlywedge t_r \\
\quad \mid \mathit{otherwise} & = & t_l \curlywedge \mathit{adjust}\ f\ k\ t_r
\end{array}
$$

The modifier *adjust* does not change the shape of the pennant. By contrast, *insert* possibly increases the height of the tree. Since the loser trees are not balanced, there is the annoying possibility that repeated insertions may produce a degenerated tree.

$$
\begin{array}{lcl}
\mathit{insert} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow (k \mapsto p) \rightarrow \mathit{PSQ}\ k\ p \rightarrow \mathit{PSQ}\ k\ p \\
\mathit{insert}\ b\ \emptyset & = & \{b\} \\
\mathit{insert}\ b\ \{b'\} & & \\
\quad \mid \mathit{key}\ b < \mathit{key}\ b' & = & \{b\} \curlywedge \{b'\} \\
\quad \mid \mathit{key}\ b == \mathit{key}\ b' & = & \{b\} \qquad\qquad \text{-- update} \\
\quad \mid \mathit{key}\ b > \mathit{key}\ b' & = & \{b'\} \curlywedge \{b\} \\
\mathit{insert}\ b\ (t_l \curlywedge t_r) & & \\
\quad \mid \mathit{key}\ b \leqslant \mathit{max\text{-}key}\ t_l & = & \mathit{insert}\ b\ t_l \curlywedge t_r \\
\quad \mid \mathit{otherwise} & = & t_l \curlywedge \mathit{insert}\ b\ t_r
\end{array}
$$

In the case of search trees deletion is notoriously more difficult to handle than insertion. Perhaps surprisingly, this does not hold for priority search pennants. The reason is simply that using the tournament view all modifications take place at the fringe of the tree.

$$
\begin{array}{lcl}
\mathit{delete} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow k \to \mathit{PSQ}\ k\ p \to \mathit{PSQ}\ k\ p \\
\mathit{delete}\ k\ \emptyset & = & \emptyset \\
\mathit{delete}\ k\ \{b\} & & \\
\quad \mid k == \mathit{key}\ b & = & \emptyset \\
\quad \mid \mathit{otherwise} & = & \{b\} \\
\mathit{delete}\ k\ (t_l \curlywedge t_r) & & \\
\quad \mid k \leqslant \mathit{max\text{-}key}\ t_l & = & \mathit{delete}\ k\ t_l \curlywedge t_r \\
\quad \mid \mathit{otherwise} & = & t_l \curlywedge \mathit{delete}\ k\ t_r
\end{array}
$$

Given the hybrid nature of priority search pennants the definition of *delete* is surprisingly attractive.

5.5 Range queries

Like the second version of the look-up function, the query function *at-most* employs two views simultaneously. The minimum view is used to prune the search if a node is encountered whose priority is greater than the given one.

$$
\begin{array}{lcl}
\mathit{at\text{-}most} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow p \to \mathit{PSQ}\ k\ p \to [k \mapsto p] \\
\mathit{at\text{-}most}\ p_t\ (\mathit{Min}\ b\ q) & & \\
\quad \mid \mathit{prio}\ b > p_t & = & [\,] \\
\mathit{at\text{-}most}\ p_t\ \emptyset & = & [\,] \\
\mathit{at\text{-}most}\ p_t\ \{b\} & = & [b] \quad \text{-- we know that } \mathit{prio}\ b \leqslant p_t \\
\mathit{at\text{-}most}\ p_t\ (t_l \curlywedge t_r) & = & \mathit{at\text{-}most}\ p_t\ t_l +\!\!+\ \mathit{at\text{-}most}\ p_t\ t_r
\end{array}
$$

The second query function *at-most-range* additionally tests the relation of the split key to the given range to prune the search space.

$$
\begin{array}{lcl}
\mathit{at\text{-}most\text{-}range} & :: & (\mathit{Ord}\ k, \mathit{Ord}\ p) \Rightarrow p \to (k, k) \to \mathit{PSQ}\ k\ p \to [k \mapsto p] \\
\mathit{at\text{-}most\text{-}range}\ p_t\ (k_l, k_r)\ (\mathit{Min}\ b\ q) & & \\
\quad \mid \mathit{prio}\ b > p_t & = & [\,] \\
\mathit{at\text{-}most\text{-}range}\ p_t\ (k_l, k_r)\ \emptyset & = & [\,] \\
\mathit{at\text{-}most\text{-}range}\ p_t\ (k_l, k_r)\ \{b\} & & \\
\quad \mid k_l \leqslant \mathit{key}\ b \leqslant k_r & = & [b] \quad \text{-- we know that } \mathit{prio}\ b \leqslant p_t \\
\quad \mid \mathit{otherwise} & = & [\,] \\
\mathit{at\text{-}most\text{-}range}\ p_t\ (k_l, k_r)\ (t_l \curlywedge t_r) & & \\
\qquad = \mathit{guard}\ (k_l \leqslant \mathit{max\text{-}key}\ t_l)\ (\mathit{at\text{-}most\text{-}range}\ p_t\ (k_l, k_r)\ t_l) & & \\
\qquad +\!\!+\ \mathit{guard}\ (\mathit{max\text{-}key}\ t_l \leqslant k_r)\ (\mathit{at\text{-}most\text{-}range}\ p_t\ (k_l, k_r)\ t_r) & &
\end{array}
$$

The helper function *guard* is defined

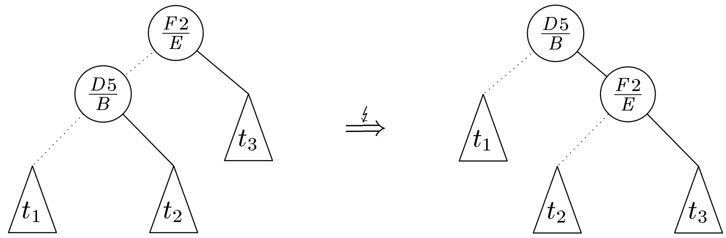
$$
\begin{array}{lcl}
\mathit{guard} & :: & \mathit{Bool} \to [a] \to [a] \\
\mathit{guard}\ \mathit{False}\ \mathit{as} & = & [\,] \\
\mathit{guard}\ \mathit{True}\ \mathit{as} & = & \mathit{as}.
\end{array}
$$

Note that the implementation can be improved by specializing *at-most-range* to semi-infinite key ranges, see, for instance (Wood, 1993).

The query functions are analysed in Section 7.

6 A balanced scheme

One of the strengths of priority search pennants as compared to priority search trees is that the basic implementation can be easily extended by a balancing scheme. Most schemes use rotations to restore balancing invariants. Now, while rotations preserve the search-tree property, they do not preserve the semi-heap property as the following example shows.



In the original tree, both losers, D and F, dominate their right subtree. This implies that they have not played against each other and that the winner stems from the leftmost subtree t_1. Now, if we rotate the loser tree to the right, the new root should dominate its right subtree but it does not. To restore the semi-heap property we have to exchange $D5$ and $F2$. We will see that, in general, at most one exchange at the cost of at most one additional comparison is required. In other words, rotations are constant time operations for priority search pennants.

By contrast, in the case of priority search trees we have to preserve the *heap property*, which takes $\Theta(h)$ time where h is the height of the tree. This means, in particular, that in order to ensure an overall logarithmic time bound, the number of rotations per update must be bounded by a constant. Red-black trees (Guibas & Sedgewick, 1978) or 2-3-4 trees (Huddleston & Mehlhorn, 1982) satisfy this constraint. On the other hand, AVL trees (Adel'son-Vel'skiĭ & Landis, 1962) or weight-balanced trees (Adams, 1993) do not guarantee such a bound. Ironically, Okasaki's elegant functional implementation of red-black trees (1999) also fails to meet this condition.

However, for priority search pennants we can freely chose an underlying balancing scheme. We pick Adams's weight-balanced trees (1993) since they support insertions and deletions equally well. A tree is *weight-balanced* if for all nodes *either* both subtrees have at most one element *or* one subtree does not have more than ω times as many elements as the opposite subtree, where ω is some constant > 3.75. To check and to maintain the invariant, each node in a loser tree is augmented by a size field:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Size} &&= \mathit{Int}\\
&\textbf{data}\ \mathit{Tree}\ k\ p &&= \mathit{Lf} \mid \mathit{Nd}\ \mathit{Size}\ (k \mapsto p)\ (\mathit{Tree}\ k\ p)\ k\ (\mathit{Tree}\ k\ p).
\end{aligned}
$$

Using views and *smart constructors* we can make the computation of the size field totally transparent.

$$
\begin{array}{lcl}
\mathit{leaf} & = & \mathit{Lf} \\
\mathit{node}\ b\ l\ k\ r & = & \mathit{Nd}\ (1 + \mathit{size}\ l + \mathit{size}\ r)\ b\ l\ k\ r \\
\mathbf{view}\ \mathit{Tree}\ k\ p & = & \mathit{Leaf} \mid \mathit{Node}\ (k \mapsto p)\ (\mathit{Tree}\ k\ p)\ k\ (\mathit{Tree}\ k\ p) \\
\quad \mathbf{where} & & \\
\quad \mathit{Lf} & \to & \mathit{Leaf} \\
\quad \mathit{Nd}\ \_\ b\ l\ k\ r & \to & \mathit{Node}\ b\ l\ k\ r \\
\mathit{size} & :: & \mathit{Tree}\ k\ p \to \mathit{Size} \\
\mathit{size}\ \mathit{Lf} & = & 0 \\
\mathit{size}\ (\mathit{Nd}\ s\ \_\ \_\ \_\ \_) & = & s
\end{array}
$$

In the sequel we will use the smart constructors *leaf* and *node* to construct weight-balanced trees, the view constructors *Leaf* and *Node* to pattern match weight-balanced trees, and the function *size* to query the size field.

The *balance* function defined below maintains weight-balance using single and double rotations under the precondition that at most one subtree has changed size by at most one element and the original tree was in balance. The algorithm is described in more detail in Adams (1993).

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{balance}\ b\ l\ k\ r} \\
\quad \mid \mathit{size}\ l + \mathit{size}\ r < 2 & = & \mathit{node}\ b\ l\ k\ r \\
\quad \mid \mathit{size}\ r > \omega * \mathit{size}\ l & = & \mathit{balance\text{-}left}\ b\ l\ k\ r \\
\quad \mid \mathit{size}\ l > \omega * \mathit{size}\ r & = & \mathit{balance\text{-}right}\ b\ l\ k\ r \\
\quad \mid \mathit{otherwise} & = & \mathit{node}\ b\ l\ k\ r \\
\multicolumn{3}{l}{\mathit{balance\text{-}left}\ b\ l\ k\ r@(\mathit{Node}\ \_\ \mathit{rl}\ \_\ \mathit{rr})} \\
\quad \mid \mathit{size}\ \mathit{rl} < \mathit{size}\ \mathit{rr} & = & \mathit{single\text{-}left}\ b\ l\ k\ r \\
\quad \mid \mathit{otherwise} & = & \mathit{double\text{-}left}\ b\ l\ k\ r \\
\multicolumn{3}{l}{\mathit{balance\text{-}right}\ b\ l@(\mathit{Node}\ \_\ \mathit{ll}\ \_\ \mathit{lr})\ k\ r} \\
\quad \mid \mathit{size}\ \mathit{lr} < \mathit{size}\ \mathit{ll} & = & \mathit{single\text{-}right}\ b\ l\ k\ r \\
\quad \mid \mathit{otherwise} & = & \mathit{double\text{-}right}\ b\ l\ k\ r
\end{array}
$$

The *balance* operation is essentially the same as for search trees. Only the implementation of the rotations is more elaborate since they have to maintain the semi-heap property. Figure 6 displays the possible cases for a single rotation to the right. Since a single rotation involves two nodes and since each node may dominate one of two subtrees, we must distinguish four different cases. The only problematic case is the last one, where we have to perform one additional match to determine the top

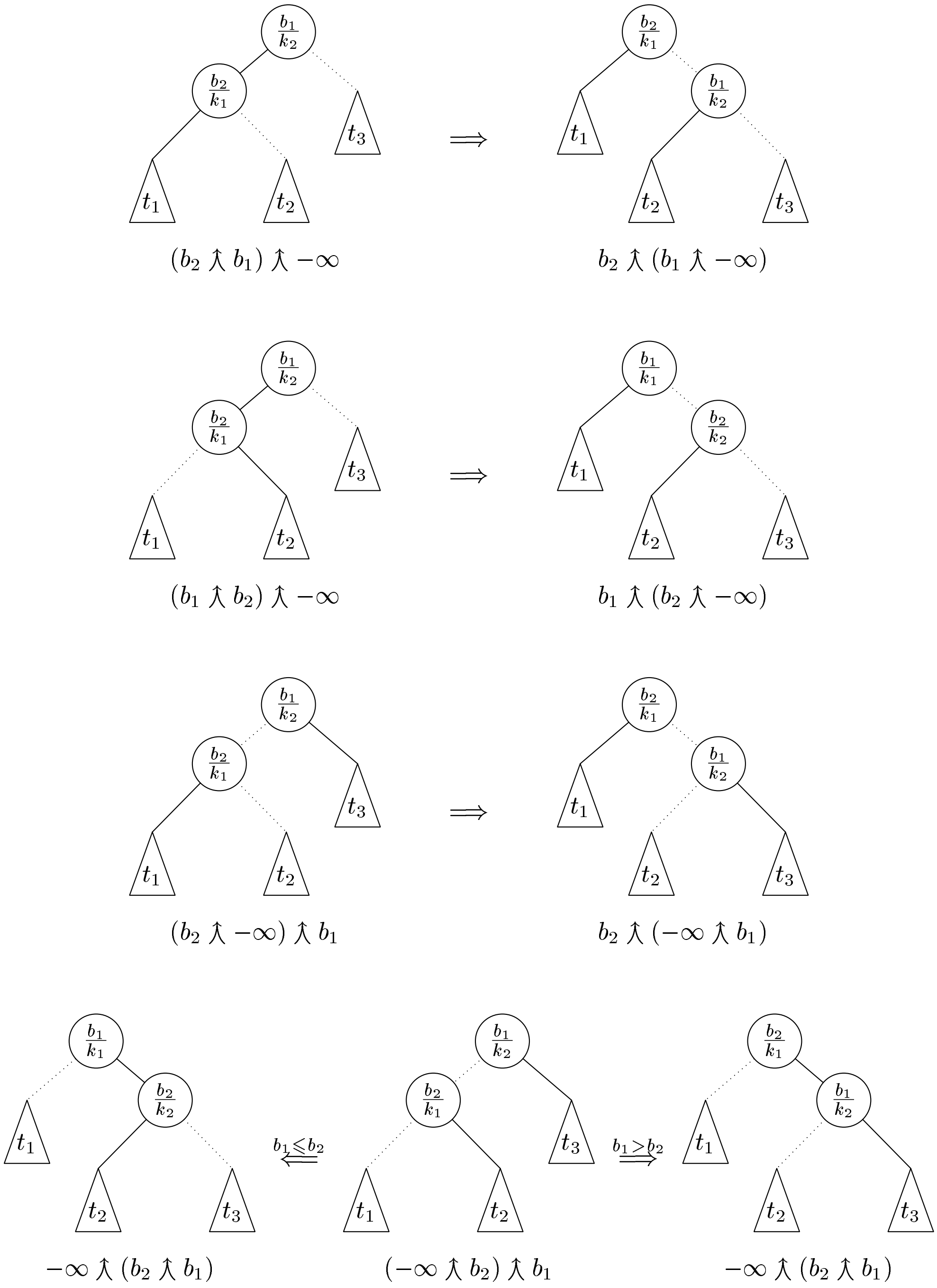


Fig. 6. A single rotation to the right ($-\infty$ represents the winner; $b_1 \leqslant b_2$ is shorthand for *prio* $b_1 \leqslant$ *prio* b_2).

binding. In general, b_1 is the new top binding iff $key\ b_2 > k_1$ and $prio\ b_1 \leqslant prio\ b_2$. The four cases for the left rotation are symmetric.

$$
\begin{array}{l}
\mathit{single\text{-}left}\ b_1\ t_1\ k_1\ (\mathit{Node}\ b_2\ t_2\ k_2\ t_3) \\
\quad \mid \mathit{key}\ b_2 \leqslant k_2 \wedge \mathit{prio}\ b_1 \leqslant \mathit{prio}\ b_2 \;=\; \mathit{node}\ b_1\ (\mathit{node}\ b_2\ t_1\ k_1\ t_2)\ k_2\ t_3 \\
\quad \mid \mathit{otherwise} \qquad\qquad\qquad\qquad\;\; =\; \mathit{node}\ b_2\ (\mathit{node}\ b_1\ t_1\ k_1\ t_2)\ k_2\ t_3 \\
\mathit{single\text{-}right}\ b_1\ (\mathit{Node}\ b_2\ t_1\ k_1\ t_2)\ k_2\ t_3 \\
\quad \mid \mathit{key}\ b_2 > k_1 \wedge \mathit{prio}\ b_1 \leqslant \mathit{prio}\ b_2 \;=\; \mathit{node}\ b_1\ t_1\ k_1\ (\mathit{node}\ b_2\ t_2\ k_2\ t_3) \\
\quad \mid \mathit{otherwise} \qquad\qquad\qquad\qquad\;\; =\; \mathit{node}\ b_2\ t_1\ k_1\ (\mathit{node}\ b_1\ t_2\ k_2\ t_3)
\end{array}
$$

Double rotations are implemented in terms of single rotations.

$$
\begin{array}{l}
\mathit{double\text{-}left}\ b_1\ t_1\ k_1\ (\mathit{Node}\ b_2\ t_2\ k_2\ t_3) \\
\;=\; \mathit{single\text{-}left}\ b_1\ t_1\ k_1\ (\mathit{single\text{-}right}\ b_2\ t_2\ k_2\ t_3) \\
\mathit{double\text{-}right}\ b_1\ (\mathit{Node}\ b_2\ t_1\ k_1\ t_2)\ k_2\ t_3 \\
\;=\; \mathit{single\text{-}right}\ b_1\ (\mathit{single\text{-}left}\ b_2\ t_1\ k_1\ t_2)\ k_2\ t_3
\end{array}
$$

Remark 4

Since a double rotation is defined in terms of two single rotations, at most two additional matches are required. Perhaps surprisingly, one can show that only one additional match suffices. A direct implementation of the double rotations is left as an exercise to the reader. □

It remains to adapt the implementation of Section 5 to balanced trees. This can be done by a simple renaming: occurrences of the constructors *Start* and *Loser* in patterns must be replaced by *Leaf* and *Node*; occurrences in expressions must be replaced by *leaf* and *balance*. The smart constructor *node* can be used instead of *balance* if the shape of the tree has not changed (as in the case of *adjust*) or if the tree is known to be balanced (as in the case of *from-ord-list*).

Let us conclude the section with a brief discussion of the running times of the various operations. For simplicity, we assume that we are working in a strict setting. Weight-balanced trees have a height that is logarithmic in the number of elements. Consequently, the dictionary operations (*lookup*, *insert*, and *delete*) and the priority queue operations (*Min*) have a worst-case running time of $\Theta(\log n)$. The conversion functions *from-ord-list* and *to-ord-list* are both linear in the number of bindings. Finally, the range queries take $\Theta(r(\log n - \log r))$ time where r is the length of the output list—the next section contains a detailed analysis. The following table summarizes the running times:

Constructors and insertion		*Destructors and deletion*	
$\emptyset$	$\Theta(1)$	*Empty*	$\Theta(1)$
$\{\cdot\}$	$\Theta(1)$	*Min*	$\Theta(\log n)$
insert	$\Theta(\log n)$	*delete*	$\Theta(\log n)$
from-ord-list	$\Theta(n)$		
Observers		*Modifier*	
lookup	$\Theta(\log n)$	*adjust*	$\Theta(\log n)$
to-ord-list	$\Theta(n)$		
at-most	$\Theta(r(\log n - \log r))$		
at-most-range	$\Theta(r(\log n - \log r))$.		

7 Analysis of *at-most* and *at-most-range*

The range queries *at-most* and *at-most-range* are so-called *output-sensitive* algorithms, that is, their running time is not only governed by the total number of bindings in the tree but also by the number of bindings they return as a result. To estimate their running time we have to determine the number of nodes that must be inspected to return r outputs. A general observation is that whenever a player enters the output list, we must additionally check all the players who have lost to this particular player. Consider the pennant of Figure 3. If Eelco is selected, we must check Lambert, Doaitse, and Johan. If Lambert is also selected, we must additionally check Piet and Nigel.

The structure becomes more apparent if we turn the binary semi-heap into a *multiway heap*. The dominated subtrees become children and the non-dominated subtrees become siblings. Figure 7 displays the tree thus obtained. This transformation is an instance of what is known as the *natural correspondence* between binary trees and forests, see Knuth (1997).

To simplify the analysis let us assume that the original trees are perfectly balanced as in our example, so that we have a total number of $n = 2^h$ bindings. In this special case we obtain as the result of the transformation a so-called *binomial heap* (Vuillemin, 1978). Now, in a binomial heap with $n = 2^h$ elements, we have one node with h subtrees (namely the root), 2^0 nodes with $h-1$ subtrees, 2^1 nodes with $h-2$ subtrees, …, 2^{h-2} nodes with 1 subtree, and 2^{h-1} nodes with 0 subtrees. Summing up and and adding one for the root we obtain a total of n nodes:

$$n = 1 + h + 2^0 \cdot (h-1) + 2^1 \cdot (h-2) + \cdots + 2^{h-2} \cdot 1 + 2^{h-1} \cdot 0.$$

Using the binary logarithm we can rewrite the above identity into the following form:

$$n = 1 + h + \sum_{k=1}^{n-1} h - 1 - \lfloor \lg k \rfloor.$$

On the right-hand side we have a sum with $n+1$ summands. Now, if we only sum

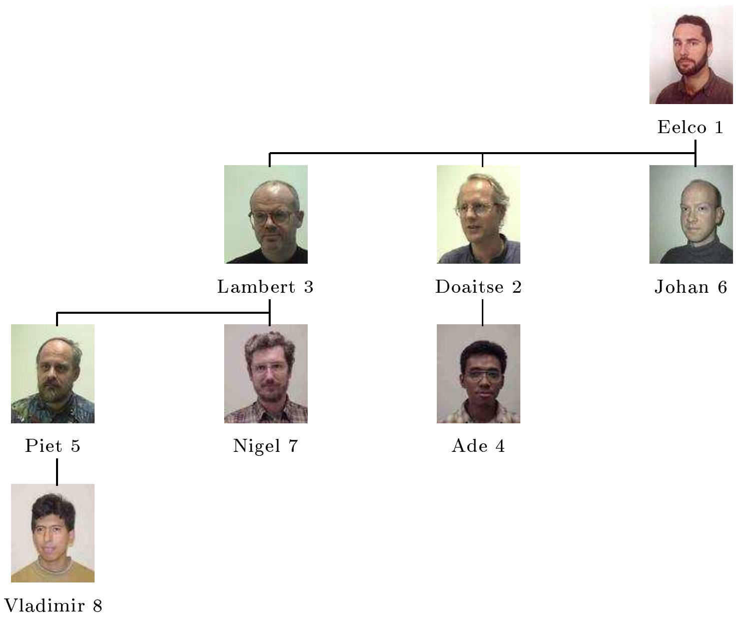


Fig. 7. The multi-way heap corresponding to the binary semi-heap of Figure 3.

up the first $r+1$ summands, we obtain the desired maximum number of successors of r nodes. Consequently, the worst-case running time of *at-most* is proportional to

$$1 + h + \sum_{k=1}^{r-1} h - 1 - \lfloor \lg k \rfloor,$$

for $1 < r \leqslant n$. To estimate the asymptotic growth of this function we use the formula

$$\sum_{k=1}^{m} \lfloor \lg k \rfloor = (m+1)\lfloor \lg(m+1) \rfloor - 2^{\lfloor \lg(m+1) \rfloor + 1} + 2$$

and calculate

$$\begin{aligned}
& 1 + h + \sum_{k=1}^{r-1} h - 1 - \lfloor \lg k \rfloor \\
= \quad & 1 + h + (r-1)(h-1) - (r\lfloor \lg r \rfloor - 2^{\lfloor \lg r \rfloor + 1} + 2) \\
= \quad & r \lg n - r - r\lfloor \lg r \rfloor + 2^{\lfloor \lg r \rfloor + 1} \\
= \quad & r(\lg n - \lg r) + O(r).
\end{aligned}$$

Thus, if r is small, we have a logarithmic running time. The running time eventually becomes linear as r approaches n.

The second query function *at-most-range* enjoys the same asymptotic bound since the test of the priority dominates the running time. Note that a range query that considers only the keys requires merely $\Theta(\log n + r)$ time.

Let us conclude the section by noting that priority search pennants answer range queries less efficiently than priority search trees, which support them in $\Theta(\log n + r)$ time (Fries *et al.*, 1987). The reason is simply that the heap property is stronger than the semi-heap property: in the case of binary heaps at most two additional elements must be checked for every element that enters the output list. As an aside, this also shows that binomial heaps, which are essentially sequences of semi-heaps (Hinze, 1999b), are less well-suited for answering range queries.

8 Related work

Priority search queues We have already commented on the relationship between priority search pennants and McCreight's priority search trees (1985). Let us briefly summarize the main points. Priority search trees are restricted to balancing schemes where the number of rotations per update is bounded by a constant. By contrast, our methods works with arbitrary balancing schemes. The asymptotic running times of the finite map and the priority queue operations are the same for both approaches. However, priority search trees support range queries more efficiently.

As an aside, priority search trees should not be confused with *cartesian trees* or *treaps*, which are also a combination of search trees and priority queues (Vuillemin, 1980). In a priority search tree each node is labelled with two keys, the key of the binding and an additional split key, whereas in a treap the key of the binding serves as the split key, which completely determines the structure of the treap.

Tournament trees and pennants Tournament trees and loser trees already appear in Knuth's TAOCP series (1998). The term *pennant* was coined by Sack and Strothotte (1990) to denote topped, perfectly balanced trees (we do not require the trees to be perfectly balanced though). Pennants are widespread: Sack and Strothotte employ them to design algorithms for splitting and merging heaps in the form of *left-complete* binary trees, Okasaki (1998a) uses pennants as a fundamental building block for data structures modelled after number systems, pennants underly *binomial heaps* (Hinze, 1999b), and they are useful for analysing *red-black trees* (Hinze, 1999a).

Dijkstra's algorithm Using priority search queues we were able to implement Dijkstra's shortest-path algorithm in a purely functional way. Previous formulations like that of King (1996) relied in an essential way on stateful computations. King writes:

> ... if a purely function solution exists for these algorithms [Dijkstra's and Kruskal's] it will probably involve using a state-encapsulating combinator.

Perhaps surprisingly, by using a different abstract data type—priority search queues instead of priority queues—we obviate the need for state. We feel that the resulting code is much clearer than the state-based formulation.

Views Views have originally been introduced by Wadler (1987). Later the idea was fleshed out into a proposal for an extension to Haskell (Burton *et al.*, 1996). Okasaki slightly simplified the proposal and adapted it to Standard ML (1998b). A recent paper by the same author (2000), where Okasaki strongly advocates the use of views, revived the author's interest in this language feature.

9 Conclusion

Priority search queues are an amazing combination of finite maps and priority queues in that they support both dictionary and priority queue operations. Building upon the metaphor of a knockout tournament we have developed a simple, yet efficient implementation technique for this abstract data type. In developing the code the concept of views was tremendously helpful: views enhanced both the readability and the modularity of the code. We have presented two applications of priority search queues: a purely functional implementation of Dijkstra's single-source shortest path algorithm and an efficient implementation of the first-fit heuristics for the bin packing problem. We hope to see further applications in the future.

References

Adams, S. (1993) Functional Pearls: Efficient sets—a balancing act. *Journal of Functional Programming* **3**(4):553–561.

Adel'son-Vel'skiĭ, G. and Landis, Y. (1962) An algorithm for the organization of information. *Doklady Akademiia Nauk SSSR* **146**:263–266. English translation in Soviet Math. Dokl. 3, pp. 1259–1263.

Braun, W. and Rem, M. (1983) *A logarithmic implementation of flexible arrays.* Memorandum MR83/4, Eindhoven University of Technology.

Burton, W., Meijer, E., Sansom, P., Thompson, S. and Wadler, P. (1996) *Views: An Extension to Haskell Pattern Matching.* available from `http://www.haskell.org/development/views.html`.

Fries, O., Mehlhorn, K., Näher, S. and Tsakalidis, A. (1987) A $\log\log n$ data structure for three-sided range queries. *Information Processing Letters* **25**(4):269–273.

Guibas, L. J. and Sedgewick, R. (1978) A diochromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science* pp. 8–21. IEEE Computer Society.

Hinze, R. (1999a) Constructing red-black trees. Okasaki, C. (ed), *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France* pp. 89–99. The proceedings appeared as a technical report of Columbia University, CUCS-023-99, also available from `http://www.cs.columbia.edu/~cdo/waaapl.html`.

Hinze, R. (1999b) Functional Pearl: Explaining binomial heaps. *Journal of Functional Programming* **9**(1):93–104.

Huddleston, S. and Mehlhorn, K. (1982) A new data structure for representing sorted lists. *Acta Informatica* **17**:157–184.

King, D. (1996) *Functional Programming and Graph Algorithms.* Ph.d. thesis, Department of Computer Science, University of Glasgow.

Knuth, D. E. (1997) *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* 3rd edn. Addison-Wesley Publishing Company.

Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching.* 2nd edn. Addison-Wesley Publishing Company.

McCreight, E. M. (1985) Priority search trees. *SIAM Journal on Computing* **14**(2):257–276.

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA.* Lecture Notes in Computer Science 523, pp. 124–144. Springer-Verlag.

Okasaki, C. (1998a) *Purely Functional Data Structures.* Cambridge University Press.

Okasaki, C. (1998b) Views for Standard ML. *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland* pp. 14–23.

Okasaki, C. (1999) Functional Pearl: Red-Black trees in a functional setting. *Journal of Functional Programming* **9**(4):471–477.

Okasaki, C. (2000) Breadth-first numbering: lessons from a small exercise in algorithm design. *ACM SIGPLAN Notices* **35**(9):131–136.

Peyton Jones, S. and Hughes, J. (eds). (1999) *Haskell 98 — A Non-strict, Purely Functional Language.* Available from `http://www.haskell.org/definition/`.

Sack, J.-R. and Strothotte, T. (1990) A characterization of heaps and its applications. *Information and Computation* **86**(1):69–86.

Sheard, T. and Fegaras, L. (1993) A fold for all seasons. *Proceedings 6th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark* pp. 233–242. ACM-Press.

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

Vuillemin, J. (1980) A unifying look at data structures. *Communications of the ACM* **23**:229–239.

Wadler, P. (1987) Views: a way for pattern matching to cohabit with data abstraction. ACM (ed), *POPL '87. Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages, January 21–23, 1987, Munich, W. Germany* pp. 307–313. ACM Press.

Wood, D. (1993) *Data Structures, Algorithms, and Performance.* Addison-Wesley Publishing Company.

A Auxiliary types and functions

This appendix lists auxiliary types and functions used in the paper. All of them with the notable exception of *foldm* are predefined in Haskell.

The type *Maybe a* represents optional values of type a.

$$\textbf{data}\ \mathit{Maybe}\ a \ = \ \mathit{Nothing} \mid \mathit{Just}\ a$$

The ubiquitous data type of parametric lists is given by

$$\textbf{data}\ [a] \ = \ [\,] \mid a : [a].$$

The function *foldr* captures a common pattern of recursion on lists—it is also known as the *fold-functional* (Sheard & Fegaras, 1993) or as a *catamorphism* (Meijer *et al.*, 1991).

$$\begin{array}{lcl}
\mathit{foldr} & :: & (a \to b \to b) \to b \to [a] \to b \\
\mathit{foldr}\ (\star)\ b\ [\,] & = & b \\
\mathit{foldr}\ (\star)\ b\ (a : as) & = & a \star \mathit{foldr}\ (\star)\ b\ as
\end{array}$$

Thus, *foldr* $(\star)$ b applied to the list $a_1 : a_2 : \cdots : a_n : [\,]$ yields $a_1 \star (a_2 \star (\cdots \star (a_n \star b) \cdots))$. Note that the parentheses group from the right. The dual computation, where the parentheses group from the left, is performed by *foldl*.

$$
\begin{array}{lcl}
\mathit{foldl} & :: & (a \to b \to a) \to a \to [\,b\,] \to a \\
\mathit{foldl}\ (\star)\ a\ [\,] & = & a \\
\mathit{foldl}\ (\star)\ a\ (b : bs) & = & \mathit{foldl}\ (\star)\ (a \star b)\ bs
\end{array}
$$

Informally, we have *foldl* $(\star)$ a $(b_1 : b_2 : \cdots : b_n : [\,]) = (\cdots\ ((a \star b_1) \star b_2) \star \cdots) \star b_n$. Both, *foldr* and *foldl*, produce a degenerated expression tree. The third fold-operator, called *foldm*, folds a list in a binary-sub-division fashion producing an almost balanced tree.

$$
\begin{array}{lcl}
\mathit{foldm} & :: & (a \to a \to a) \to a \to [\,a\,] \to a \\
\mathit{foldm}\ (\star)\ e\ as & & \\
\quad \mid \mathit{null}\ as & = & e \\
\quad \mid \mathit{otherwise} & = & \mathit{fst}\ (\mathit{rec}\ (\mathit{length}\ as)\ as) \\
\quad \mathbf{where}\ \mathit{rec}\ 1\ (a : as) & = & (a, as) \\
\qquad\qquad \mathit{rec}\ n\ as & = & (a_1 \star a_2, as_2) \\
\qquad\qquad\quad \mathbf{where}\ m & = & n\ \text{`}\mathit{div}\text{`}\ 2 \\
\qquad\qquad\qquad (a_1, as_1) & = & \mathit{rec}\ (n - m)\ as \\
\qquad\qquad\qquad (a_2, as_2) & = & \mathit{rec}\ m\ as_1
\end{array}
$$

The expression tree generated by *foldm* takes the form of a leaf-oriented Braun tree: for any given subexpression $l \star r$, the left part l has either the same number of leaves as the right part, or one leaf more.