A Simple Implementation Technique for Priority Search Queues

RALF HINZE

Institute of Information and Computing Sciences
Utrecht University
Email: ralf@cs.uu.nl
Homepage: http://www.cs.uu.nl/~ralf/

April, 2001

(Pick the slides at .../~ralf/talks.html#T22.)

Aim of the talk

- ☞ advertize *priority search queues*
- ☞ describe a new implementation technique for priority search queues
- ☞ promote *views*

Recap: views

A *view* allows any type to be viewed as a free data type. The following view (minimum view) allows any list to be viewed as an ordered list.

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

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.

Recap: views (continued)

The *view constructors*, $Empty$ and Min, can now be used to pattern match elements of type $[a]$ (where a is an instance of Ord).

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

Priority search queues: signature

Priority search queues are conceptually finite maps that support 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.

Bindings are represented by the following data type:

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

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

-- constructors

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

-- destructors

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

-- observers

$$
\begin{array}{lcl}
lookup & :: & k \to PSQ\ k\ p \to Maybe\ p \\
to\text{-}ord\text{-}list & :: & PSQ\ k\ p \to [k \mapsto p]
\end{array}
$$

-- modifier

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

Application: single-source shortest path

Dijkstra's algorithm maintains a queue that maps each vertex to its estimated distance from the source and works by repeatedly removing the vertex with minimal distance and updating the distances of its adjacent vertices.

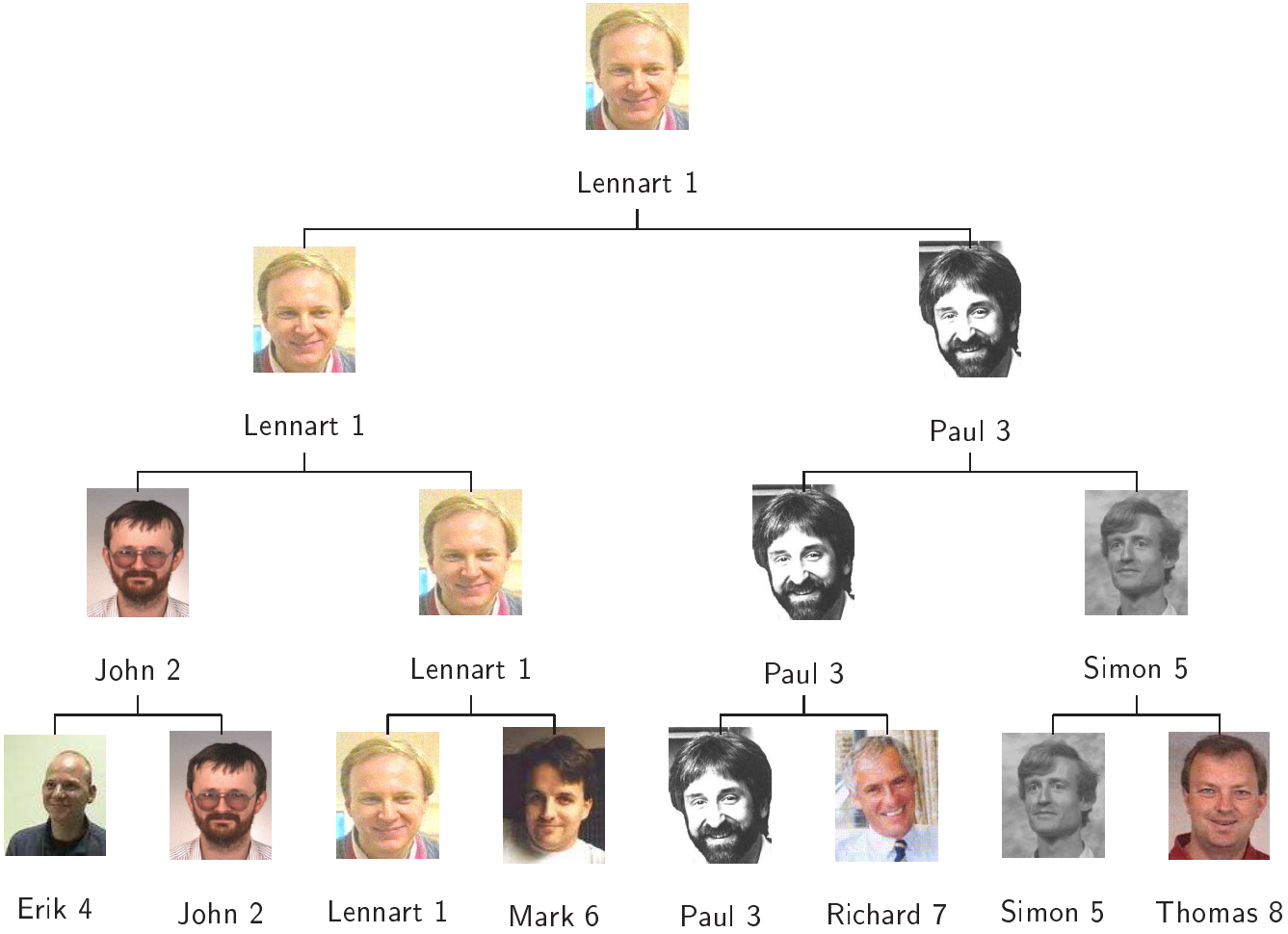
The update operation is typically called $decrease$:

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

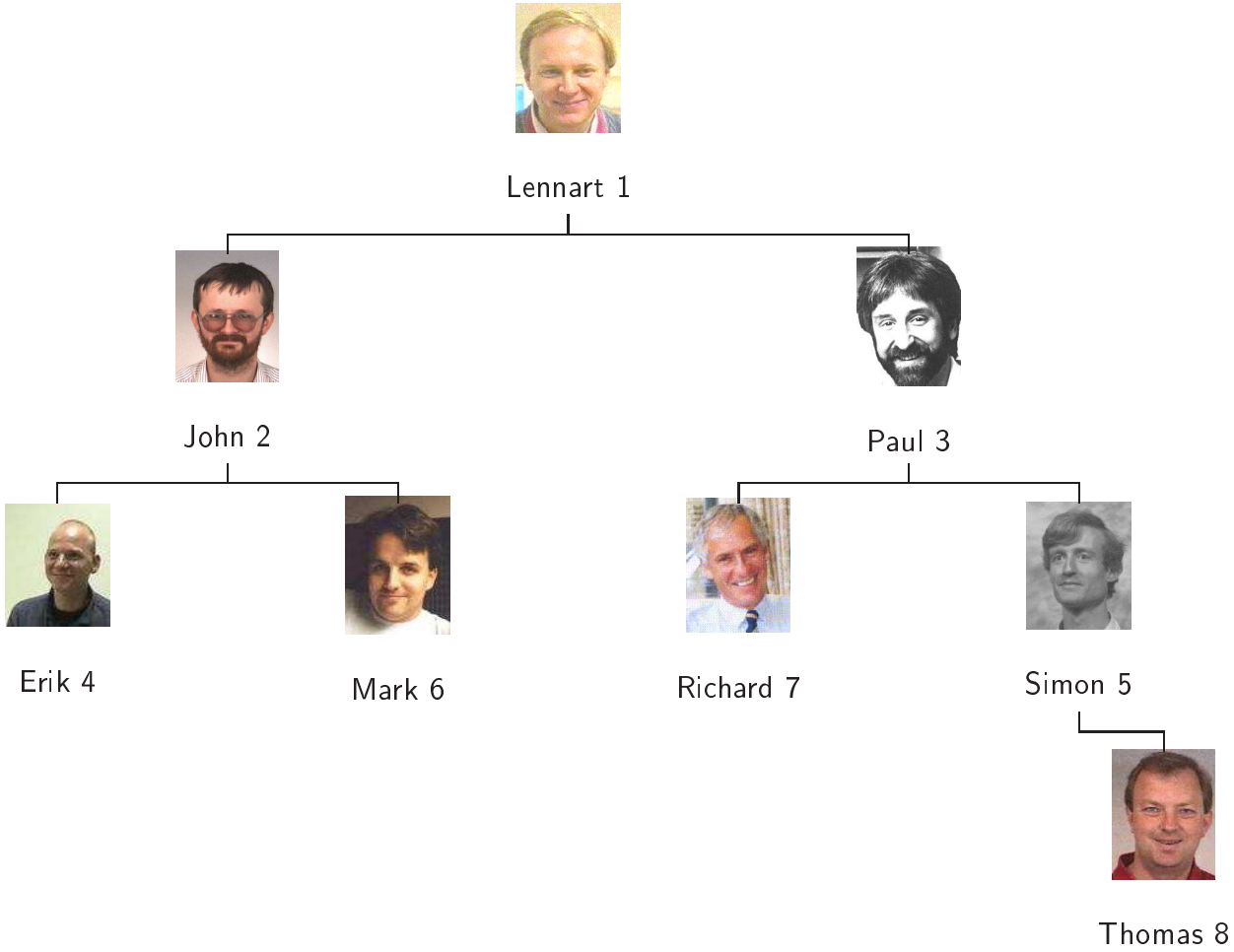
Application: single-source shortest path (continued)

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

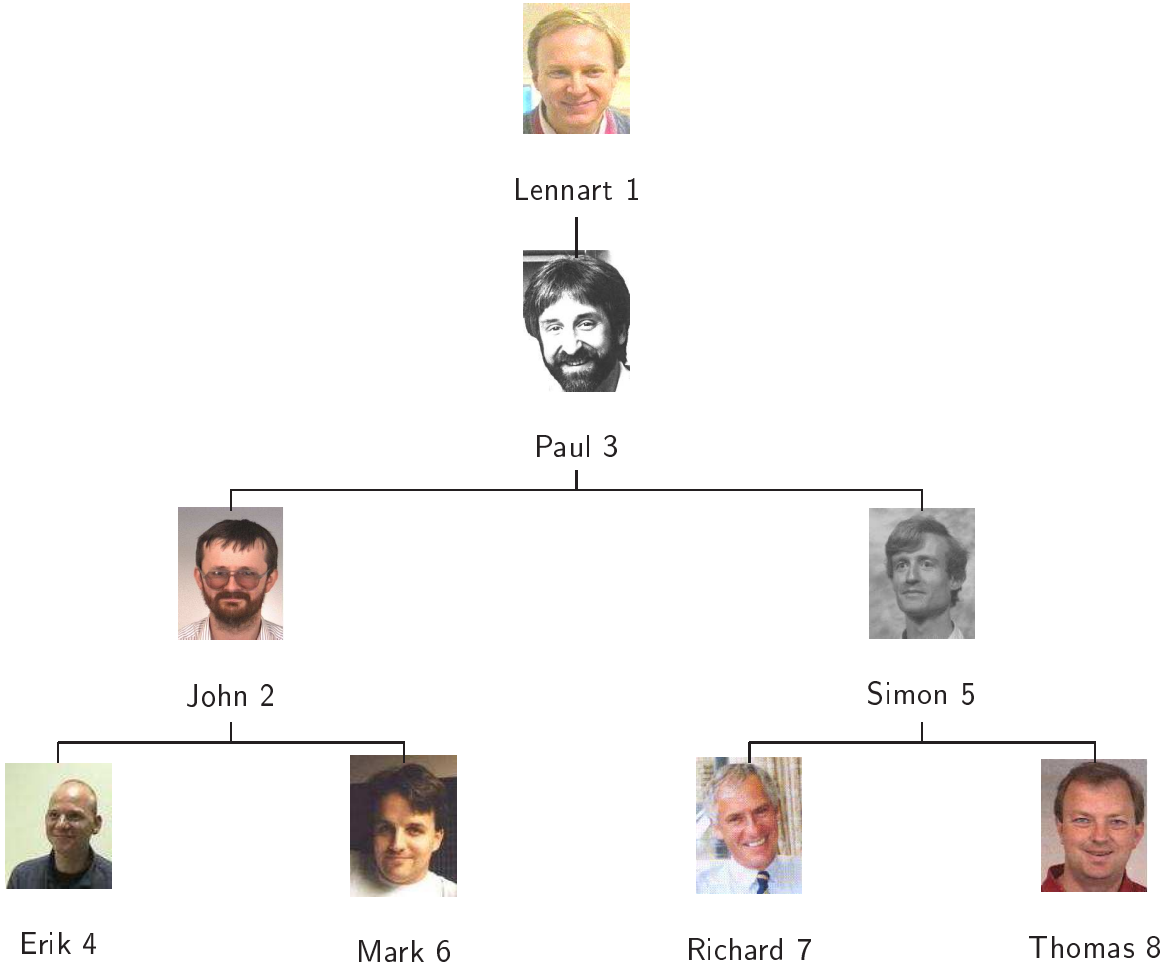
Implementation: tournament trees



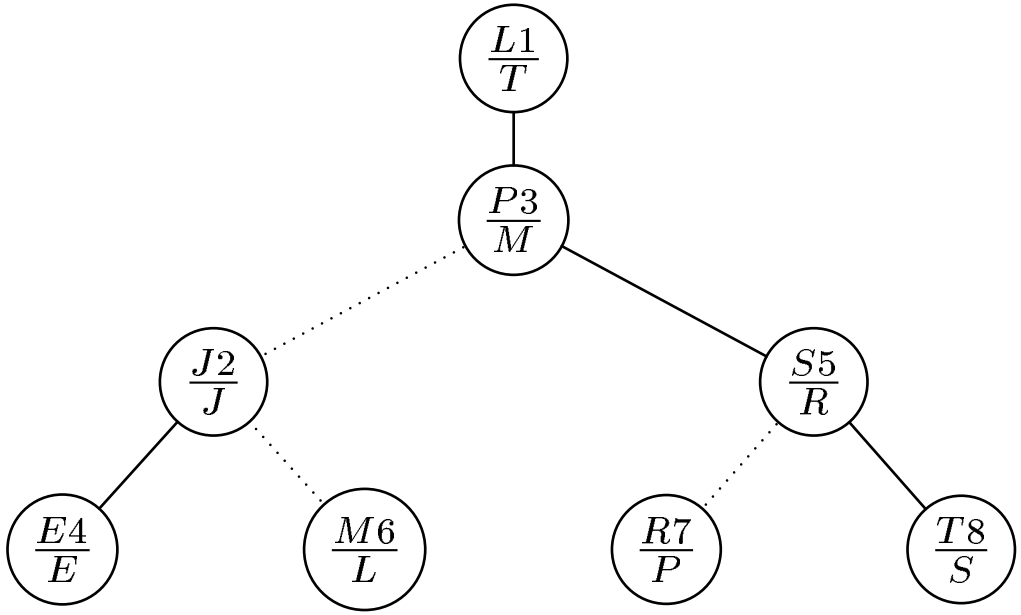
Heaps — priority search trees



Semi-heaps — priority search pennants



Priority search pennants: adding split keys



Priority search pennants: data types

The Haskell data type for priority search pennants is a direct implementation of these ideas.

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

NB. $\mathit{Winner}\ b\ t\ m \cong \mathit{Loser}\ b\ t\ m\ \mathit{Start}$.

The maximum key is accessed using the function $\mathit{max\text{-}key}$.

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

Priority search pennants: invariants

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.

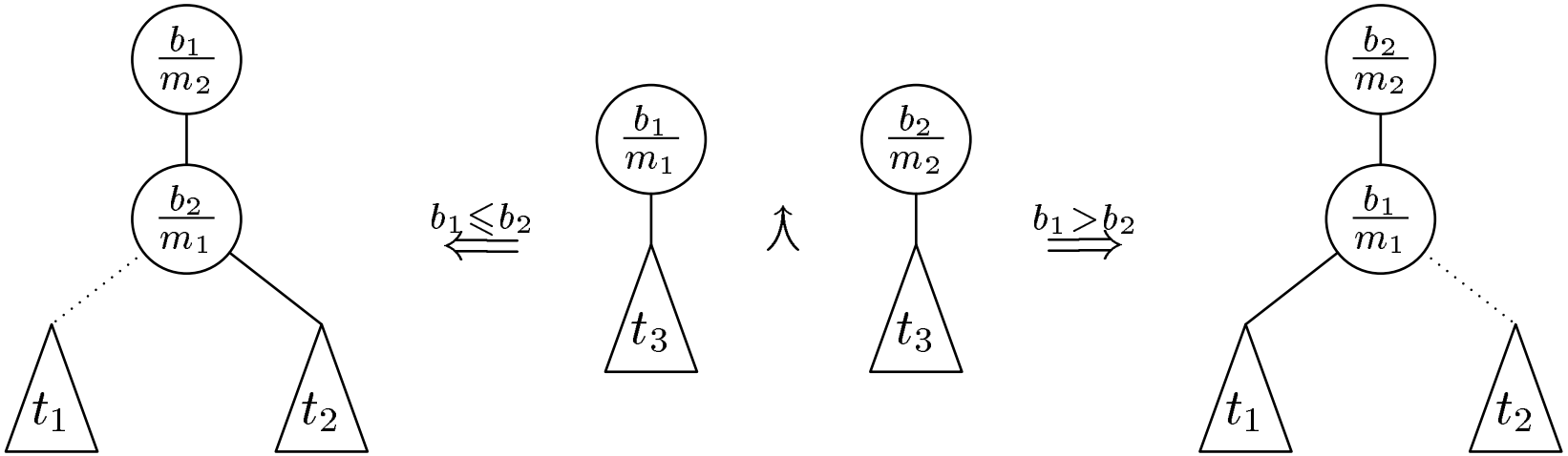
Key condition: The maximum key and the split keys must also occur as keys of bindings.

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

Constructors: $\emptyset$ and $\{\cdot\}$

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

Playing a match



NB. $b_1 \leqslant b_2$ is shorthand for $prio\ b_1 \leqslant prio\ b_2$.

Playing a match (continued)

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

Constructors: *from-ord-list*

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

NB. *foldm* folds a list in a binary-sub-division fashion.

Destructors

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

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

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

A second view: priority search pennants as tournaments

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

NB. We have taken the liberty of using $\emptyset$, $\{\cdot\}$ and '$\curlywedge$' also as constructors.

Observers: *to-ord-list*

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

Observers: *lookup*

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

Modifier: *adjust*

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

Constructors: *insert*

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

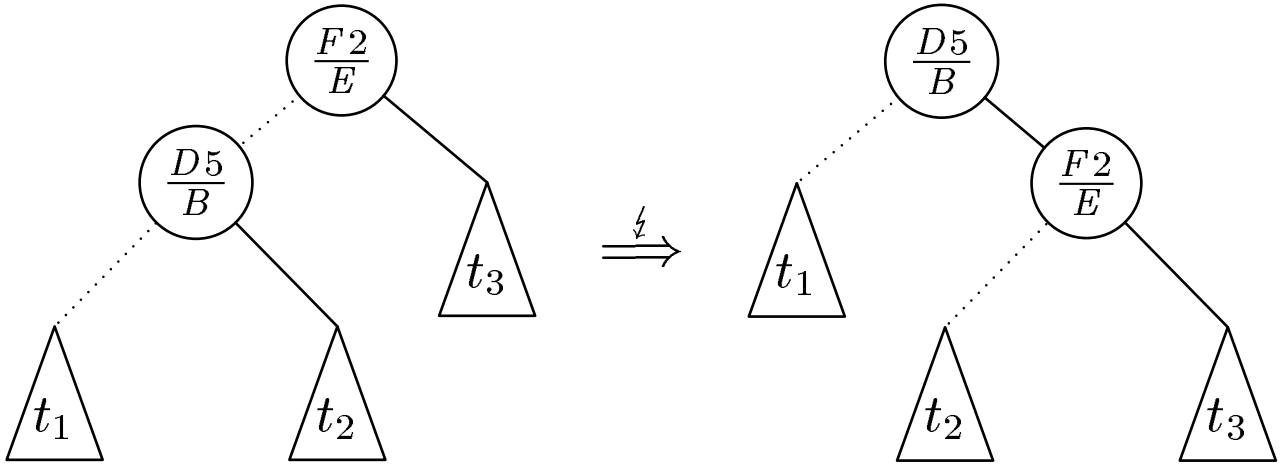
Destructors: *delete*

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

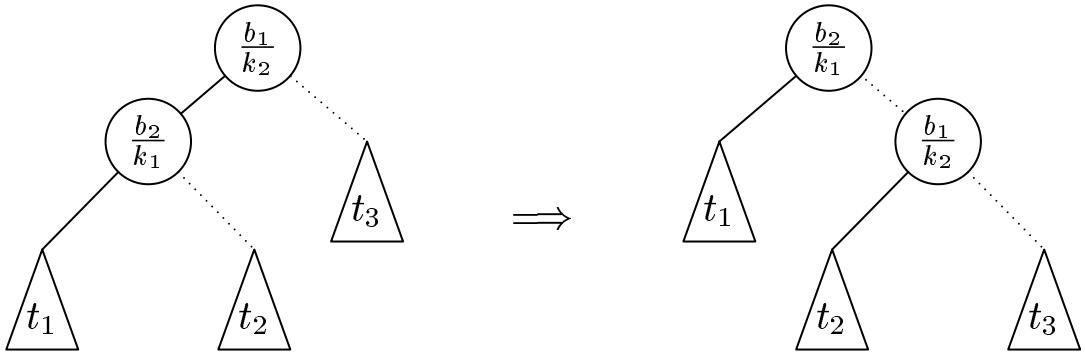
Adding a balancing scheme

One of the strengths of priority search pennants as compared to priority search trees is that a balancing scheme can be easily added.

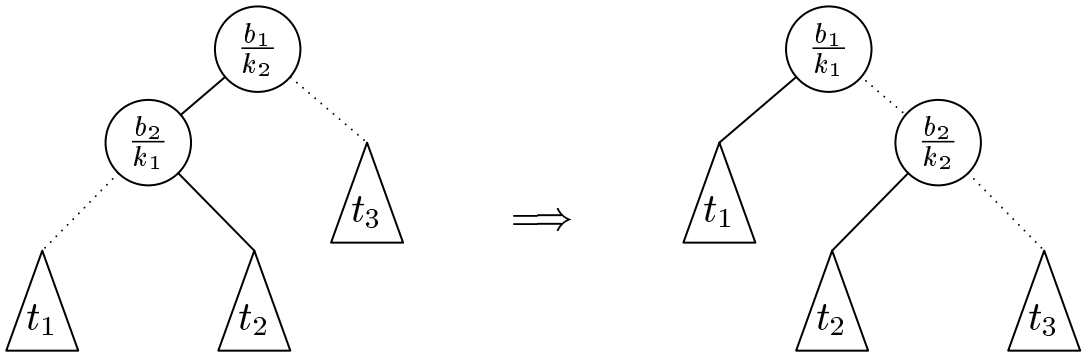
Most balancing schemes use rotations to restore balancing invariants. However, rotations do not preserve the semi-heap property:



Single rotation

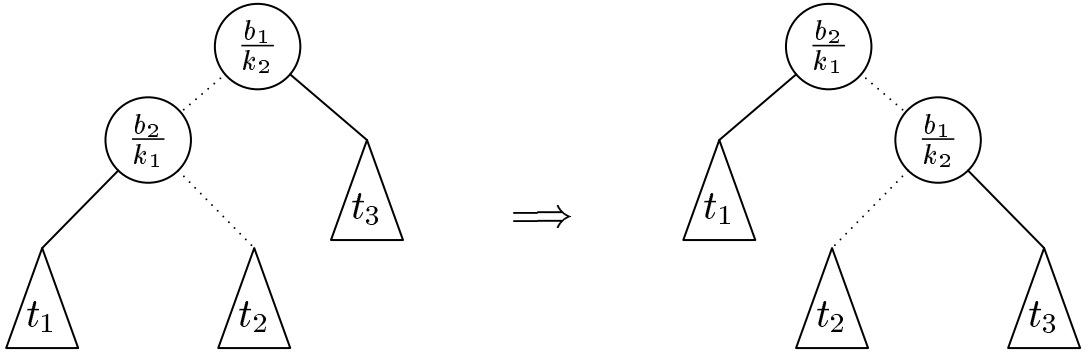


$(b_2 \curlywedge b_1) \curlywedge -\infty$ $\qquad$ $b_2 \curlywedge (b_1 \curlywedge -\infty)$

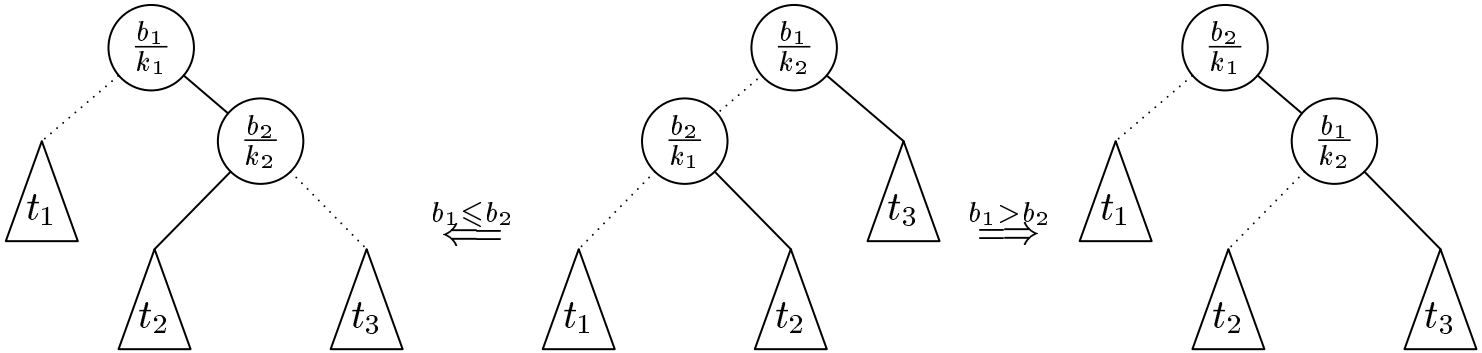


$(b_1 \curlywedge b_2) \curlywedge -\infty$ $\qquad$ $b_1 \curlywedge (b_2 \curlywedge -\infty)$

Single rotation (continued)



$(b_2 \curlywedge -\infty) \curlywedge b_1$ $\qquad$ $b_2 \curlywedge (-\infty \curlywedge b_1)$



$-\infty \curlywedge (b_2 \curlywedge b_1)$ $\qquad$ $(-\infty \curlywedge b_2) \curlywedge b_1$ $\qquad$ $-\infty \curlywedge (b_2 \curlywedge b_1)$

Summary

☞ Priority search queues are a versatile ADT.

☞ They can be easily implemented by priority search pennants—using an arbitrary balancing scheme.

☞ Views were very helpful:

- they provide a convenient interface to the ADT and
- they enhance both the readability and the modularity of the code.

Appendix

$$
\begin{array}{lcl}
\mathit{foldm} & :: & (a \to a \to a) \to a \to [\,a\,] \to a \\
\mathit{foldm}\ (*)\ e\ \mathit{as} & & \\
\quad \mid \mathit{null}\ \mathit{as} & = & e \\
\quad \mid \mathit{otherwise} & = & \mathit{fst}\ (\mathit{rec}\ (\mathit{length}\ \mathit{as})\ \mathit{as}) \\
\quad \mathbf{where}\ \mathit{rec}\ 1\ (a : \mathit{as}) & = & (a, \mathit{as}) \\
\qquad\qquad \mathit{rec}\ n\ \mathit{as} & = & (a_1 * a_2, \mathit{as}_2) \\
\qquad\qquad\quad \mathbf{where}\ m & = & n\ `\mathit{div}`\ 2 \\
\qquad\qquad\qquad\qquad (a_1, \mathit{as}_1) & = & \mathit{rec}\ (n - m)\ \mathit{as} \\
\qquad\qquad\qquad\qquad (a_2, \mathit{as}_2) & = & \mathit{rec}\ m\ \mathit{as}_1
\end{array}
$$

A programming challenge

Improve the following code which implements the first-fit heuristics for the bin packing problem.

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

Solution to the programming challenge

Priority search queues not only support dictionary and priority queue operations. As a little extra they also allow for so-called *range queries*:

$$\begin{array}{lcl} \mathit{at\text{-}most} & :: & p \to PSQ\ k\ p \to [k \mapsto p] \\ \mathit{at\text{-}most\text{-}range} & :: & p \to (k,k) \to PSQ\ k\ p \to [k \mapsto p]. \end{array}$$

The call $\mathit{at\text{-}most}\ p_t\ q$ returns a list of bindings ordered by key whose priorities are at most p_t; $\mathit{at\text{-}most\text{-}range}\ p_t\ (k_l, k_r)\ q$ 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.

Priority search pennants support both operations in $\Theta(r(\log n - \log r))$ time where r is the length of the output list.

Solution to the programming challenge (continued)

Using *at-most* we can quickly determine the first bin that can accommodate a given item (the bins are numbered consecutively).

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