FUNCTIONAL DATA STRUCTURES
The challenge and beauty of purity

RALF HINZE

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

May, 2001

(Pick the slides at `.../~ralf/talks.html#T25`.)

Overview

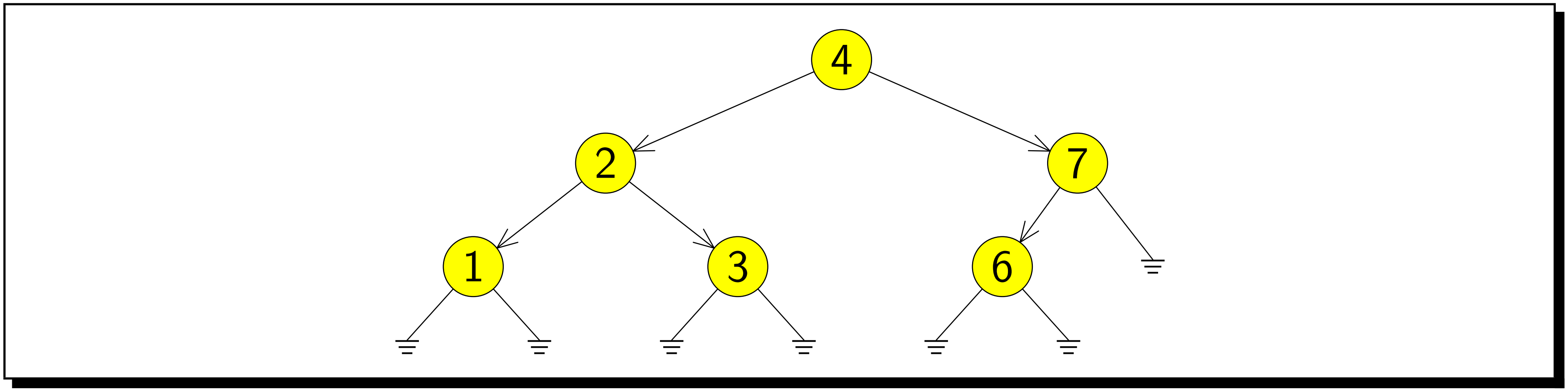
- Search trees
- Priority search queues

Search trees—Learning targets

- ✗ Persistence
- ✗ Red-black trees
- ✗ Smart constructors
- ✗ Number systems

Unbalanced binary search trees

Elements in the internal nodes are stored in *symmetric order*.



Unbalanced binary search trees—Haskell

In Haskell we represent binary trees with the following data type.

$$\textbf{data}\ \mathit{STree}\ a \quad = \quad \mathit{Leaf} \mid \mathit{Node}\ (\mathit{STree}\ a)\ a\ (\mathit{STree}\ a)$$

NB. The type STree is parameterized with the type of elements.

$$\begin{array}{lcl} \mathit{stree} & = & \mathit{Node}\ (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf})\ 1\ (\mathit{Leaf})) \\ & & \qquad\qquad\qquad 2 \\ & & \qquad\qquad\qquad (\mathit{Node}\ (\mathit{Leaf})\ 3\ (\mathit{Leaf}))) \\ & & \qquad\quad 4 \\ & & \qquad\quad (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf})\ 6\ (\mathit{Leaf})) \\ & & \qquad\qquad\qquad 7 \\ & & \qquad\qquad\qquad \mathit{Leaf}) \end{array}$$

Unbalanced binary search trees—Insertion

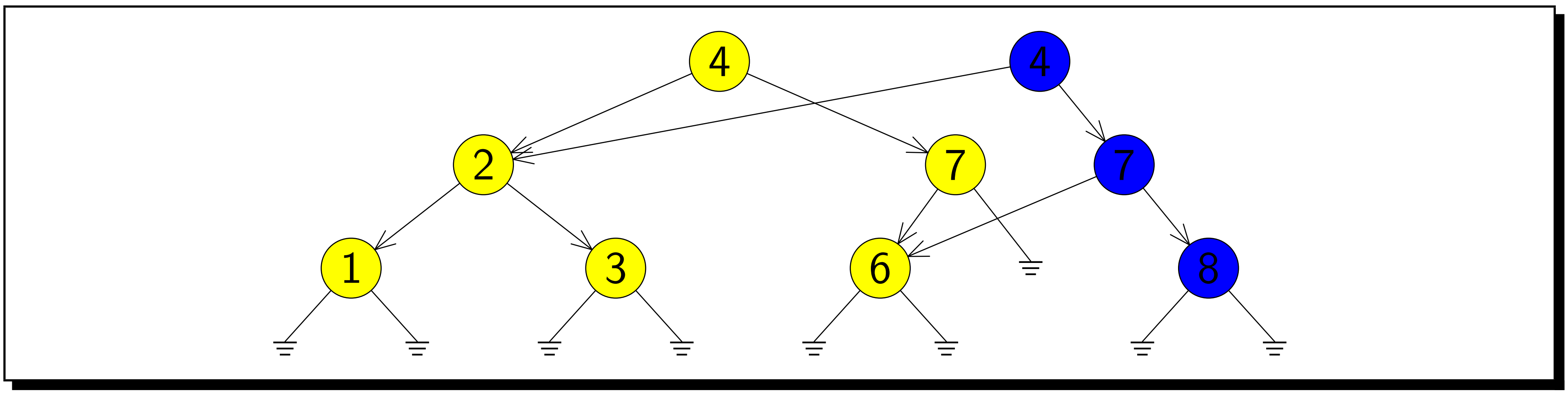
$$
\begin{array}{lcl}
insert & :: & (Ord\ a) \Rightarrow a \rightarrow STree\ a \rightarrow STree\ a \\
insert\ k\ t & = & ins\ t \\
\quad \textbf{where} & & \\
\quad ins\ Leaf & = & Node\ Leaf\ k\ Leaf \\
\quad ins\ (Node\ l\ a\ r) & & \\
\qquad \mid k < a & = & Node\ (ins\ l)\ a\ r \\
\qquad \mid k == a & = & Node\ l\ k\ r \\
\qquad \mid k > a & = & Node\ l\ a\ (ins\ r)
\end{array}
$$

NB. The type of elements must be an instance of Ord.

Persistence

Functional data structures are always *persistent*: an update creates a new structure that coexists with the old one.

Persistence is achieved via *path copying*. Situation after $insert\ 8\ stree$:



Note that the subtrees rooted at 2 and 6 are *shared*.

Persistence

Making use of persistence:

- ✘ Arbitrary "undo" (text editor, image manipulation program).
- ✘ Nested declarations with static scoping. *Idea*: use a stack of environments (only the "topmost" is active).

`{`	$\Leftarrow$ duplicate top of stack
`    int j; ...`	$\Leftarrow$ insert `j`
`    {`	$\Leftarrow$ duplicate top of stack
`        int i;`	$\Leftarrow$ insert `i`
`        int j; ...`	$\Leftarrow$ insert `j`
`    } ...`	$\Leftarrow$ pop top of stack
`}`	$\Leftarrow$ pop top of stack

Balanced search trees: Red-black trees

Unbalanced search trees may degenerate. Red-black trees are among the simplest balancing schemes.

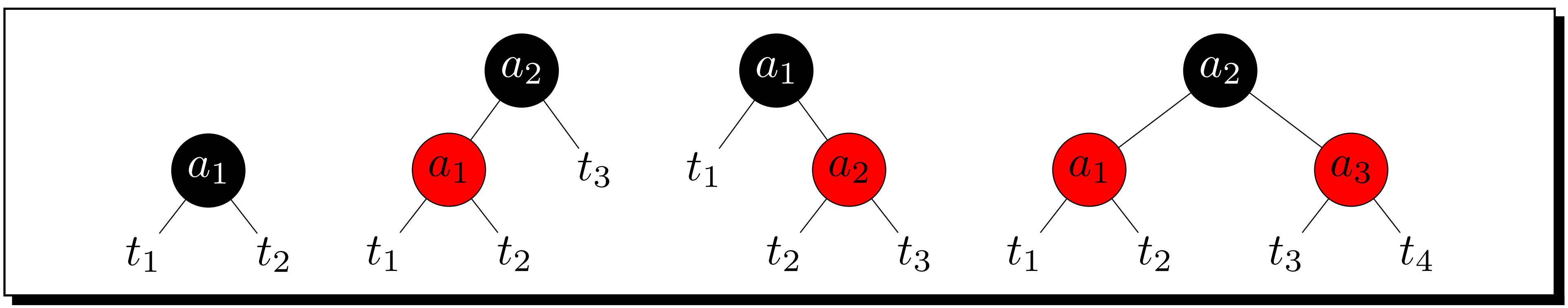
A red-black tree is a binary tree whose nodes are coloured either red or black (leaves are, by definition, black).

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Colour} & = & R \mid B \\
\mathbf{data}\ \mathit{RBTree}\ a & = & L \mid N\ \mathit{Colour}\ (\mathit{RBTree}\ a)\ a\ (\mathit{RBTree}\ a)
\end{array}
$$

Historical roots

Red-black trees were developed by R. Bayer under the name *symmetric binary B-trees* as binary tree representations of *2-3-4 trees* (a 2-3-4 tree consists of 2-, 3- and 4-nodes and satisfies the invariant that all leaves appear on the same level).

The idea of red-black trees is to represent 3- and 4-nodes by small binary trees, which consist of a black root and one or two auxiliary red children.



Balance conditions

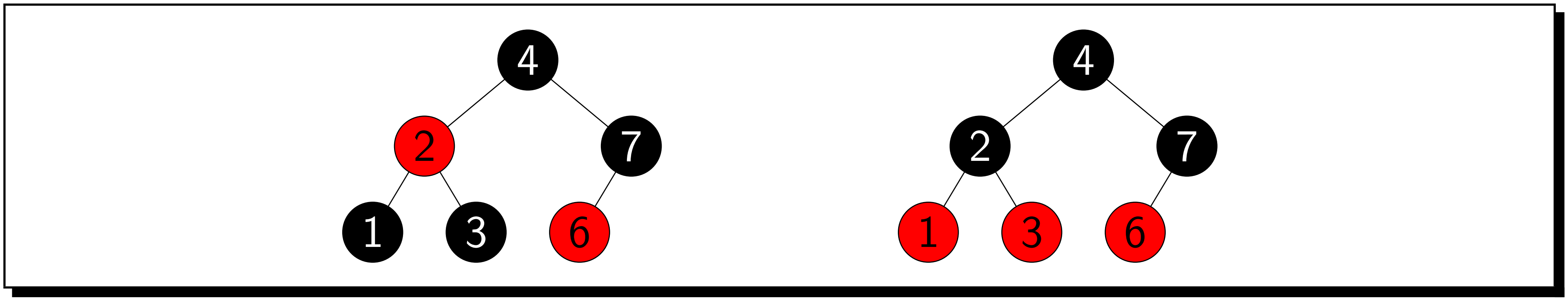
This explains the following two balance conditions.

Red condition: Each red node has a black parent.

Black condition: Each path from the root to an empty node contains exactly the same number of black nodes (this number is called the tree's *black height*).

Example red-black trees

There are two ways to color the above tree.



Properties of red-black trees

The balance conditions imply the following properties—recall that $\sum_{k=0}^{n} x^k = (1 - x^{n+1})/(1 - x)$.

$$\begin{aligned} \mathit{black\text{-}depth}\ t \leqslant \mathit{depth}\ t &\leqslant 2 \cdot \mathit{black\text{-}depth}\ t \\ 2 \uparrow \mathit{black\text{-}depth}\ t - 1 \leqslant \mathit{size}\ t &\leqslant 4 \uparrow \mathit{black\text{-}depth}\ t - 1 \\ \mathit{depth}\ t &\leqslant 2 \cdot \mathit{lg}\ (\mathit{size}\ t + 1) \end{aligned}$$

In other words, red-black trees guarantee $O(\log n)$ worst-case running time of basic dynamic-set operations.

Red-black trees: Insertion

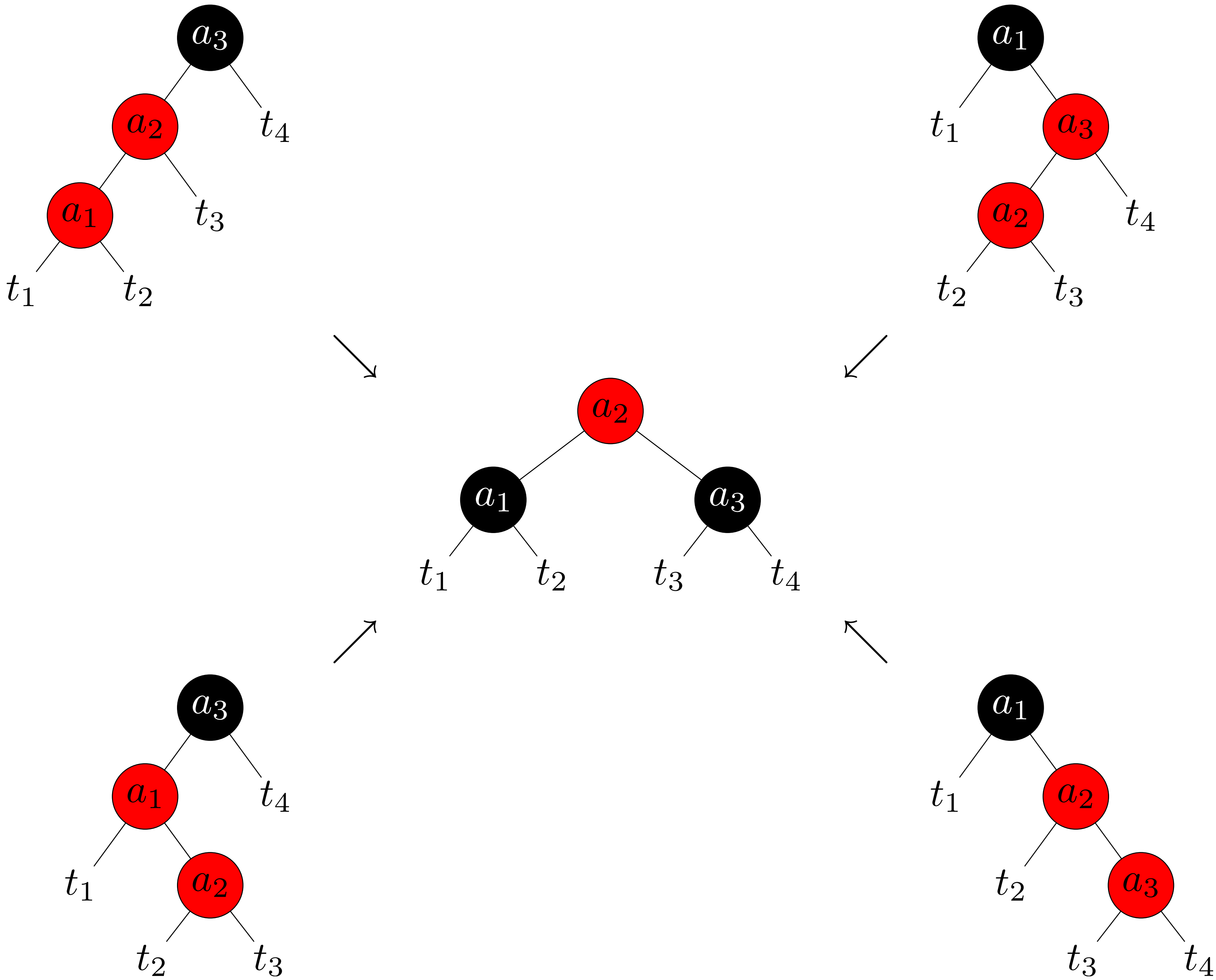
$$
\begin{array}{lcl}
insert & :: & (Ord\ a) \Rightarrow a \rightarrow RBTree\ a \rightarrow RBTree\ a \\
insert\ a\ t & = & blacken\ (ins\ t) \\
\quad \textbf{where} & & \\
\quad ins\ L & = & N\ R\ L\ a\ L \\
\quad ins\ (N\ c\ l\ b\ r) & & \\
\qquad \mid a < b & = & bal\ c\ (ins\ l)\ b\ r \\
\qquad \mid a == b & = & N\ c\ l\ a\ r \\
\qquad \mid a > b & = & bal\ c\ l\ b\ (ins\ r) \\
blacken\ (N\ \_\ l\ a\ r) & = & N\ B\ l\ a\ r
\end{array}
$$

NB. bal is a so-called *smart constructor*.

Red-black trees: Balancing

Since a new node is colored red, only the red condition is possibly violated. The smart constructor bal detects and repairs such violations.

$$
\begin{array}{lcl}
bal\ B\ (N\ R\ (N\ R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4 & & \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
bal\ B\ (N\ R\ t_1\ a_1\ (N\ R\ t_2\ a_2\ t_3))\ a_3\ t_4 & & \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
bal\ B\ t_1\ a_1\ (N\ R\ (N\ R\ t_2\ a_2\ t_3)\ a_3\ t_4) & & \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
bal\ B\ t_1\ a_1\ (N\ R\ t_2\ a_2\ (N\ R\ t_3\ a_3\ t_4)) & & \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
bal\ c\ l\ a\ r & = & N\ c\ l\ a\ r
\end{array}
$$


a_3 a_2 t_4 a_1 t_3 t_1 t_2
a_1 t_1 a_3 a_2 t_4 t_2 t_3
a_2 a_1 a_3 t_1 t_2 t_3 t_4
a_3 a_1 t_4 t_1 a_2 t_2 t_3
a_1 t_1 a_2 t_2 a_3 t_3 t_4

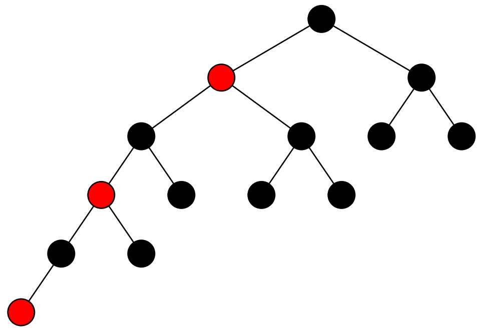
Building red-black trees

We can build a red-black tree by repeatedly inserting elements into an empty tree.

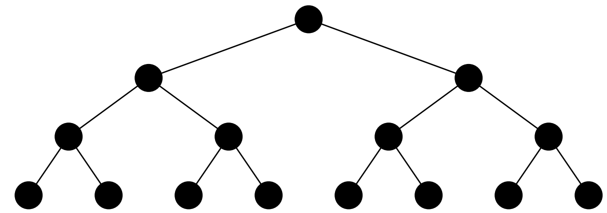
$$
\begin{array}{lcl}
\mathit{top\text{-}down} & :: & (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow \mathit{RBTree}\ a \\
\mathit{top\text{-}down} & = & \mathit{foldr}\ \mathit{insert}\ L
\end{array}
$$

NB. The elements are inserted from right to left.

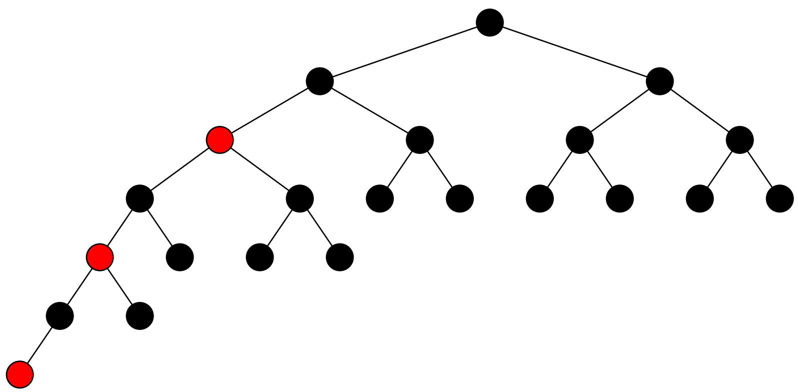
Now, assume that the elements are given in increasing order. Can we improve $\mathit{top\text{-}down}$, which has a running time of $O(n \log n)$, for this special case?



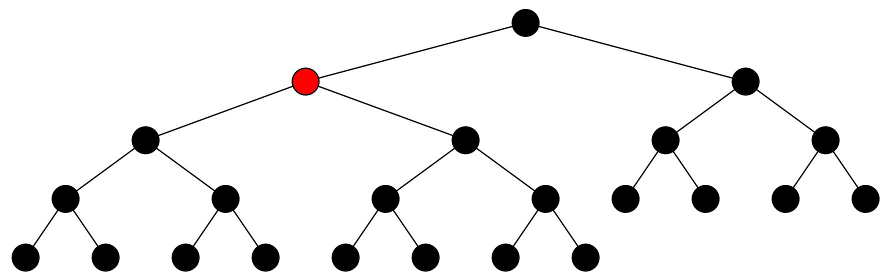
$n = 14$



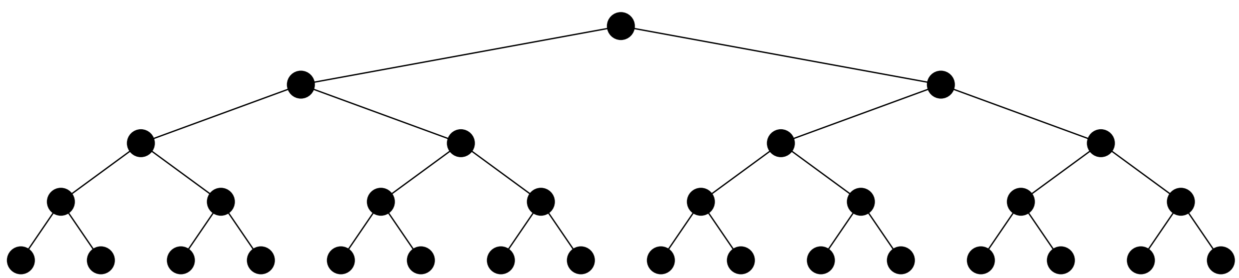
$n = 15$



$n = 22$



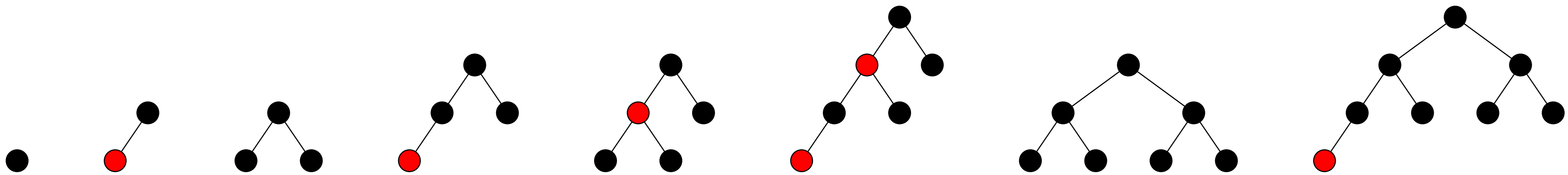
$n = 23$



$n = 31$

A closer look at $\mathit{top\text{-}down}$

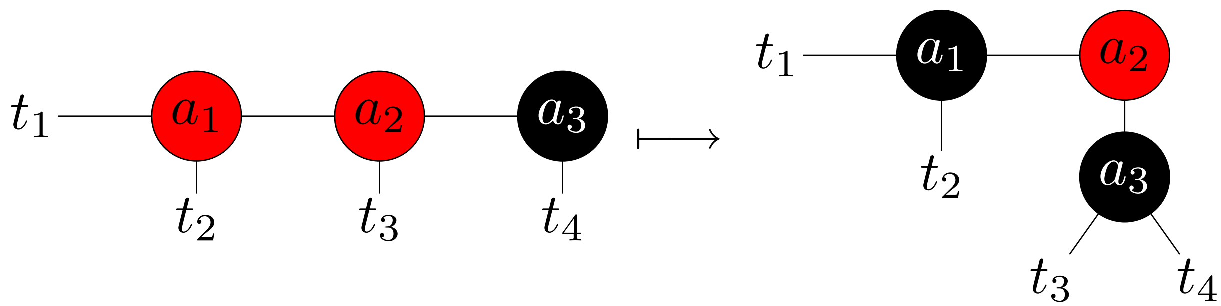
The following trees are generated by $\mathit{top\text{-}down}\,[1\,..\,i]$ for $1 \leqslant i \leqslant 8$.



NB. ins always traverses the *left spine* of the tree to the leftmost leaf.

A closer look at *top-down*

If we draw the left spine horizontally, the balancing operation (first equation of bal) takes on the following form.

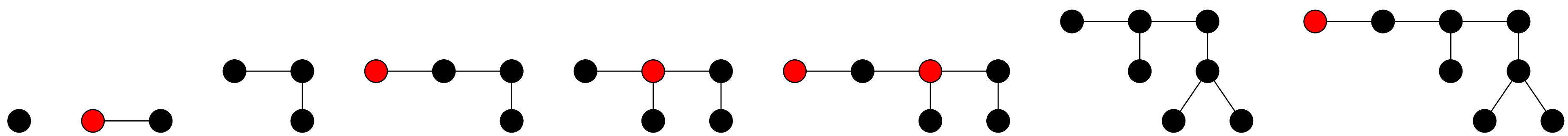


☞ The trees below the left spine (t_2, t_3 and t_4) must be perfectly balanced binary trees (perfect trees for short). Thus, the generated red-black trees correspond to sequences of *topped perfect trees* or *pennants*.

☞ A pennant of rank r contains exactly 2^r nodes.

A closer look at $top\text{-}down$

It is helpful to redraw the examples according to the *left-spine view*.



Let r be the rank of the rightmost pennant; the black condition implies that a pennant of rank i appears either once or twice for all $0 \leqslant i \leqslant r$.

☞ The red-black trees generated by $top\text{-}down$ correspond to 'binary numbers' composed of the digits 1 and 2.

The 1-2 number system

Recall that the value of a radix-2 number is given by

$$(b_{n-1} \ldots b_0)_2 = \sum_{i=0}^{n-1} b_i 2^i.$$

Each natural number has a unique representation in the 1-2 number system.

$$()_2, (1)_2, (2)_2, (11)_2, (12)_2, (21)_2, (22)_2, (111)_2, (112)_2 \ldots$$

The 1-2 number system—Haskell

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Digit} & = & \mathit{One} \mid \mathit{Two} \\
\textbf{type}\ \mathit{Nat} & = & [\,\mathit{Digit}\,]
\end{array}
$$

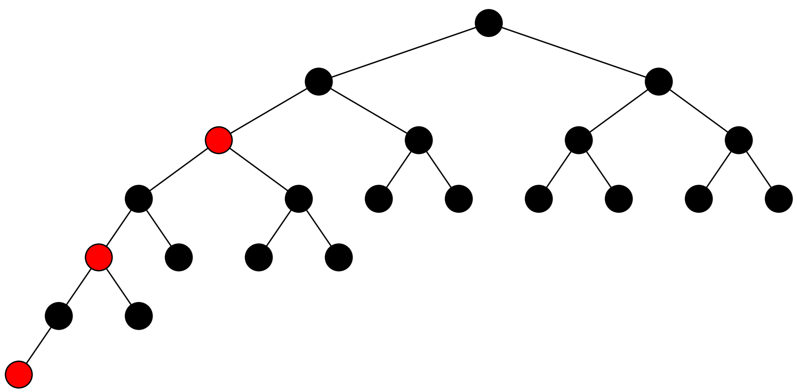
Incrementing a 1-2 number:

$$
\begin{array}{lcl}
\mathit{incr} & :: & \mathit{Nat} \to \mathit{Nat} \\
\mathit{incr}\ n & = & \mathit{add}\ \mathit{One}\ n \\
\mathit{add} & :: & \mathit{Digit} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{add}\ \mathit{One}\ [\,] & = & [\,\mathit{One}\,] \\
\mathit{add}\ \mathit{One}\ (\mathit{One} : \mathit{ds}) & = & \mathit{Two} : \mathit{ds} \\
\mathit{add}\ \mathit{One}\ (\mathit{Two} : \mathit{ds}) & = & \mathit{One} : \mathit{add}\ \mathit{One}\ \mathit{ds}
\end{array}
$$

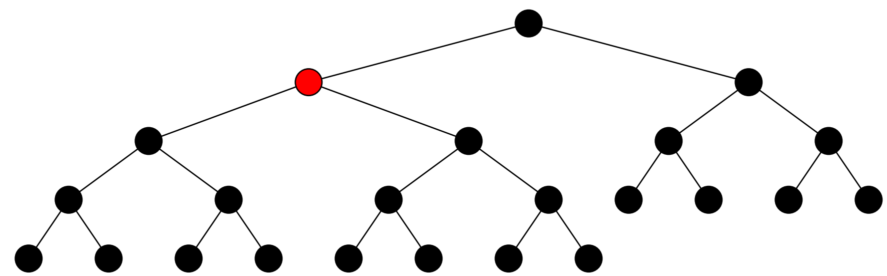
NB. The carry is made explicit.

Corollaries

The trees corresponding to $(1^{\{n\}})_{\text{1-2}}$ are perfectly balanced; the trees corresponding to $(2^{\{n\}})_{\text{1-2}}$ and $(12^{\{n\}})_{\text{1-2}}$ are *skinny trees* (a skinny tree is a tree of smallest possible size for a given height); the trees corresponding to $(1^{\{n\}}2)_{\text{1-2}}$ and $(21^{\{n\}})_{\text{1-2}}$ are *left-complete trees*.



$(1222)_{\text{1-2}}$



$(2111)_{\text{1-2}}$

Improving *top-down*

The analogy to the 1-2 number system can be exploited to improve the implementation of *top-down* for the special case that the elements appear in ascending order. The digits become containers for pennants:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Digit}\ a \quad &= \quad \mathit{One}\ a\ (\mathit{RBTree}\ a) \\
&\mid \quad \mathit{Two}\ a\ (\mathit{RBTree}\ a)\ a\ (\mathit{RBTree}\ a)\ .
\end{aligned}
$$

A red-black tree under the left-spine view is represented as a list of digits.

$$
\mathbf{type}\ \mathit{RBTree}'\ a \quad = \quad [\mathit{Digit}\ a]\ .
$$

Improving *top-down*

Inserting an element corresponds to incrementing a 1-2 number.

$$
\begin{aligned}
\mathit{insert'} &:: a \rightarrow \mathit{RBTree'}\ a \rightarrow \mathit{RBTree'}\ a \\
\mathit{insert'}\ a\ \mathit{ps} &= \mathit{add}\ (\mathit{One}\ a\ L)\ \mathit{ps}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{add}\ (\mathit{One}\ a\ t)\ [\,] && = [\mathit{One}\ a\ t] \\
&\mathit{add}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{One}\ a_2\ t_2 : \mathit{ps}) && = \mathit{Two}\ a_1\ t_1\ a_2\ t_2 : \mathit{ps} \\
&\mathit{add}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{Two}\ a_2\ t_2\ a_3\ t_3 : \mathit{ps}) \\
&\qquad = \mathit{One}\ a_1\ t_1 : \mathit{add}\ (\mathit{One}\ a_2\ (N\ B\ t_2\ a_3\ t_3))\ \mathit{ps}
\end{aligned}
$$

Improving *top-down*

$$
\begin{array}{lcl}
\mathit{bottom\text{-}up} & :: & [\,a\,] \rightarrow \mathit{RBTree}\ a \\
\mathit{bottom\text{-}up} & = & \mathit{foldl}\ \mathit{link}\ L \cdot \mathit{foldr}\ \mathit{insert}'\ [\,] \\
\mathit{link} & :: & \mathit{RBTree}\ a \rightarrow \mathit{Digit}\ a \rightarrow \mathit{RBTree}\ a \\
\mathit{link}\ l\ (\mathit{One}\ a\ t) & = & N\ B\ l\ a\ t \\
\mathit{link}\ l\ (\mathit{Two}\ a_1\ t_1\ a_2\ t_2) & = & N\ B\ (N\ R\ l\ a_1\ t_1)\ a_2\ t_2
\end{array}
$$

If as is ordered, we have $\mathit{top\text{-}down}\ as = \mathit{bottom\text{-}up}\ as$.

☞ A standard amortization argument shows that *bottom-up* runs in linear time.

☞ *top-down* and *bottom-up* construct trees with a minimal number of red nodes among all trees of that size.

Overview

✔ Search trees

✘ Priority search queues

Priority search queues—Learning targets

- ✘ Views
- ✘ Tournament trees
- ✘ Priority search pennants

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.

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

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.

Views

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

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

However, the view constructors Empty and Min must not be used in expressions—with the notable exception of the view transformation itself.

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.

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

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

-- constructors

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

-- destructors

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

-- observers

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

-- modifier

$$
\begin{array}{lcl}
adjust & :: & (p \to p) \to k \to PSQ\ k\ p \to PSQ\ k\ p
\end{array}
$$

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.

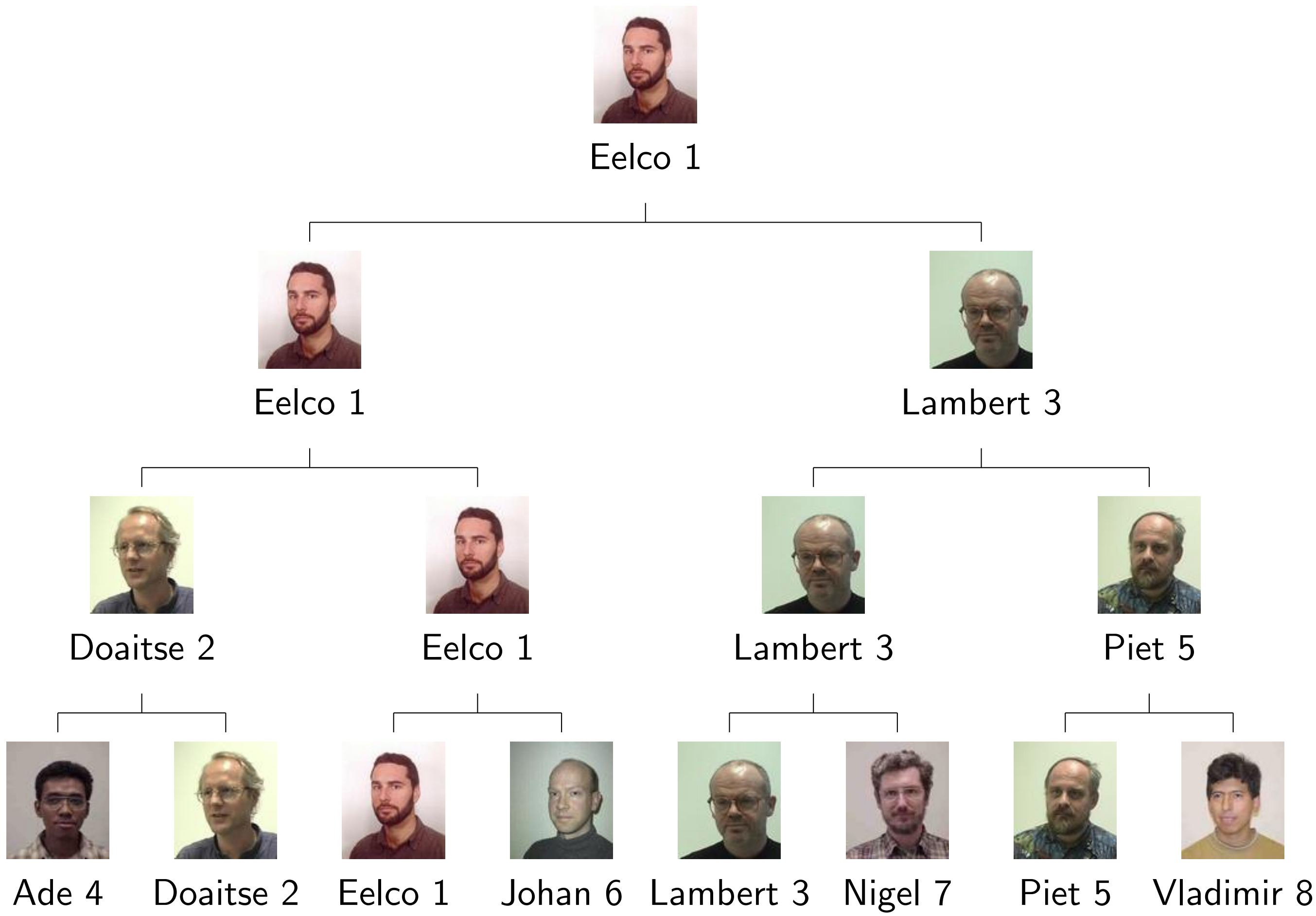
The update operation is typically called *decrease*:

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

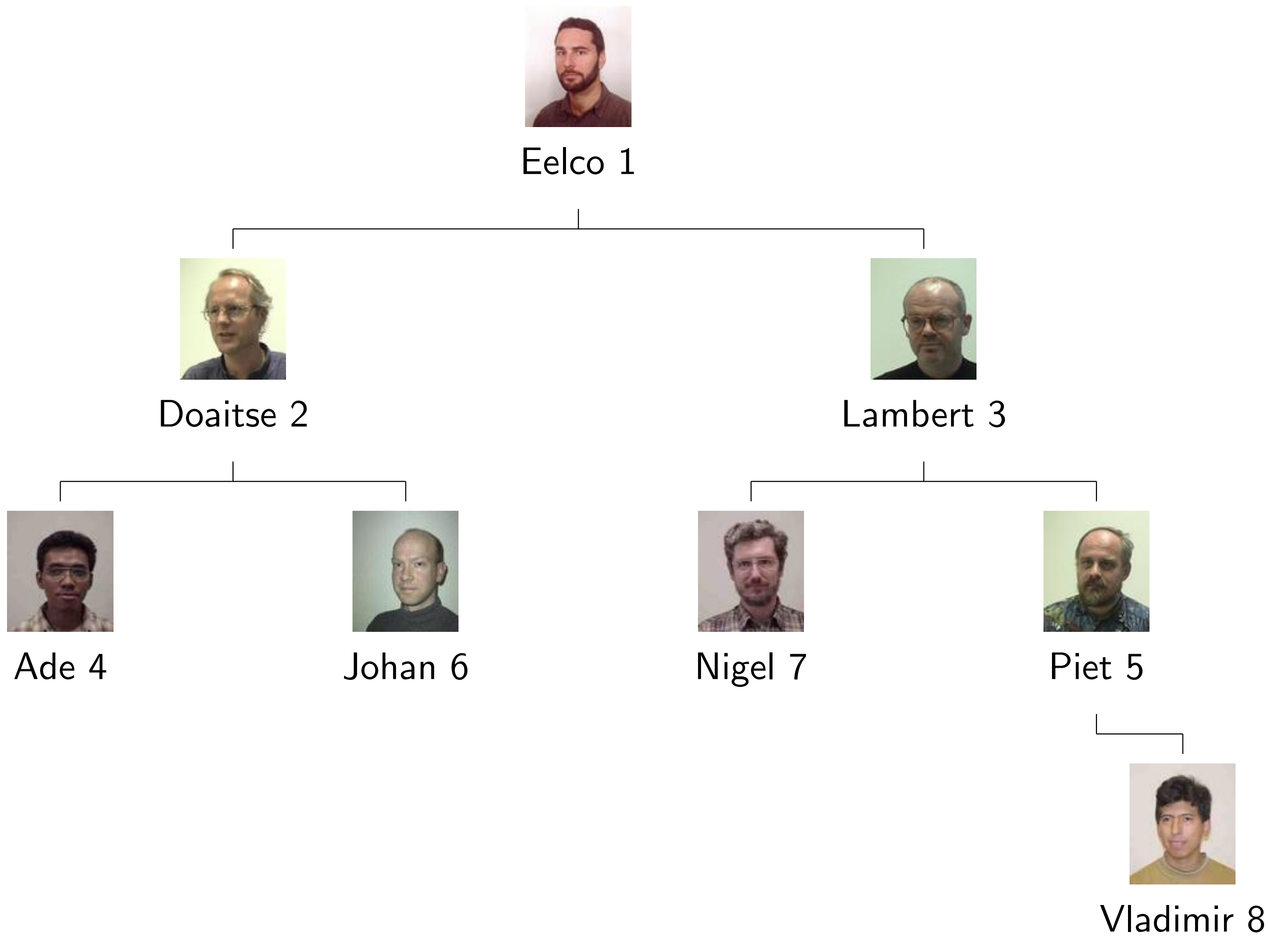
Application: single-source shortest path

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

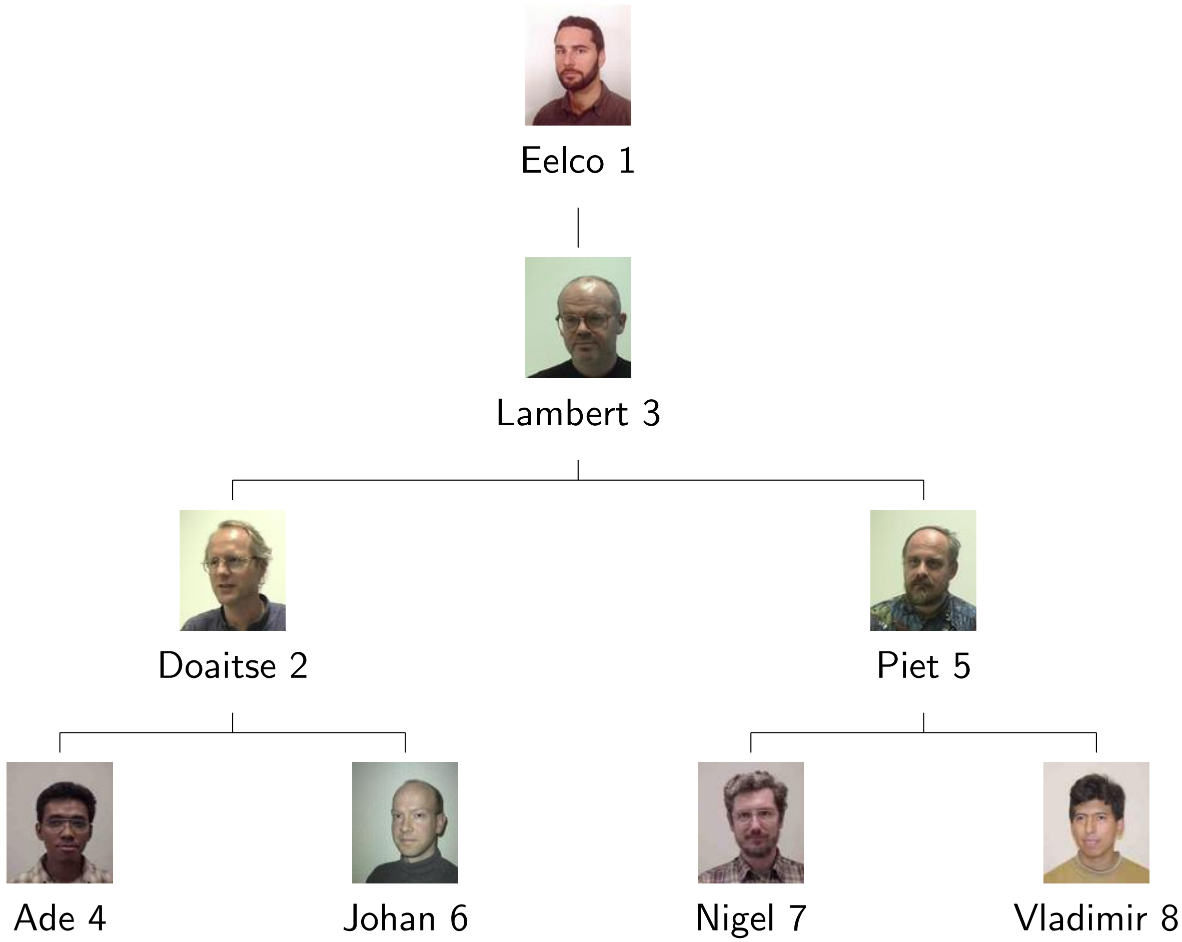
Implementation: tournament trees



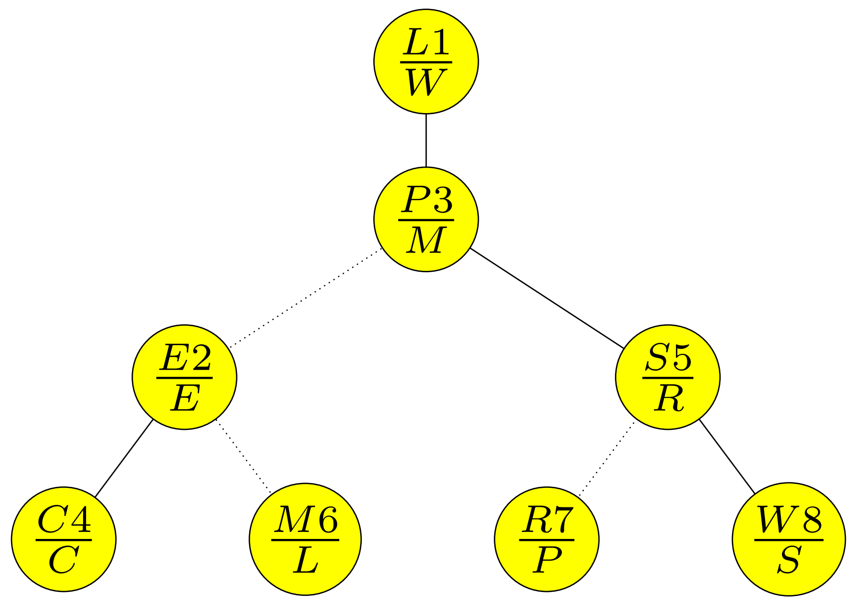
Heaps — priority search trees



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.

$$
\begin{array}{lcl}
\mathbf{data}\ PSQ\ k\ p & = & Void \\
 & | & Winner\ (k, p)\ (LTree\ k\ p)\ k \\
\mathbf{data}\ LTree\ k\ p & = & Start \\
 & | & Loser\ (k, p)\ (LTree\ k\ p)\ k\ (LTree\ k\ p)
\end{array}
$$

NB. $Winner\ b\ t\ m \cong Loser\ b\ t\ m\ Start$.

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

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

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.

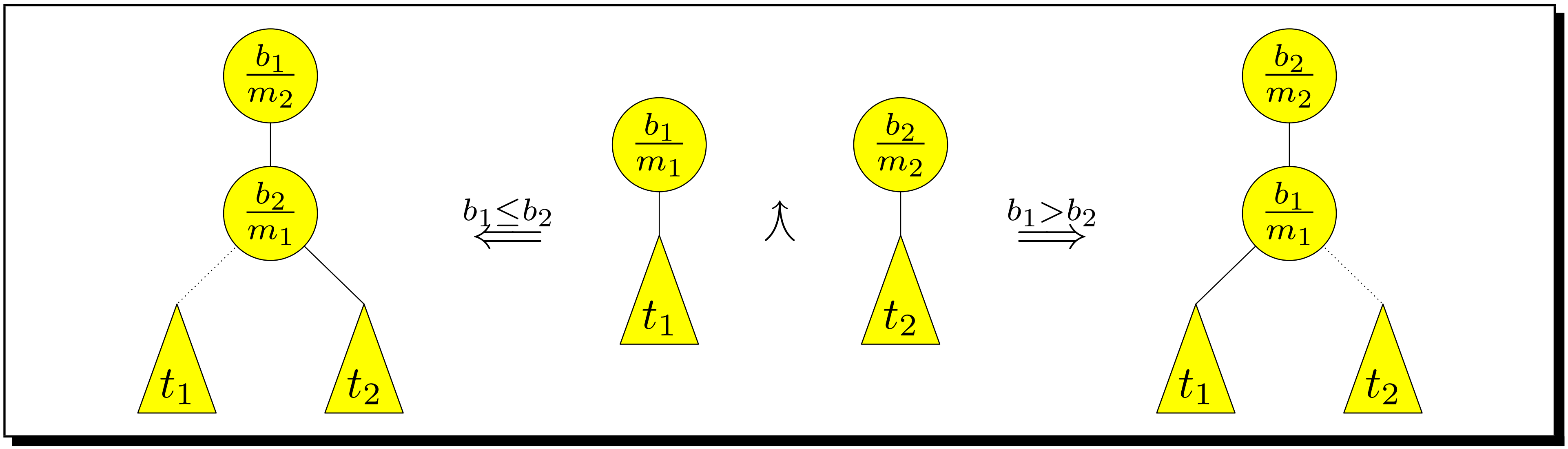
Key condition: The maximum key and the split keys must also occur as keys of bindings.

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

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

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

Playing a match



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

Playing a match

$$
\begin{array}{lcl}
(\curlywedgeuparrow) & :: & PSQ\ k\ p \rightarrow PSQ\ k\ p \rightarrow PSQ\ k\ p \\
Void \curlywedgeuparrow t' & = & t' \\
t \curlywedgeuparrow Void & = & t \\
\multicolumn{3}{l}{Winner\ b\ t\ m \curlywedgeuparrow Winner\ b'\ t'\ m'} \\
\quad \mid prio\ b \leqslant prio\ b' & = & Winner\ b\ (Loser\ b'\ t\ m\ t')\ m' \\
\quad \mid otherwise & = & Winner\ b'\ (Loser\ b\ t\ m\ t')\ m'
\end{array}
$$

Constructors: *from-ord-list*

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

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

Destructors

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

The function $\mathit{second\text{-}best}$ 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 tournament trees

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

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

Observers: *to-ord-list*

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

Observers: *lookup*

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

Modifier: *adjust*

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

Constructors: $insert$

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

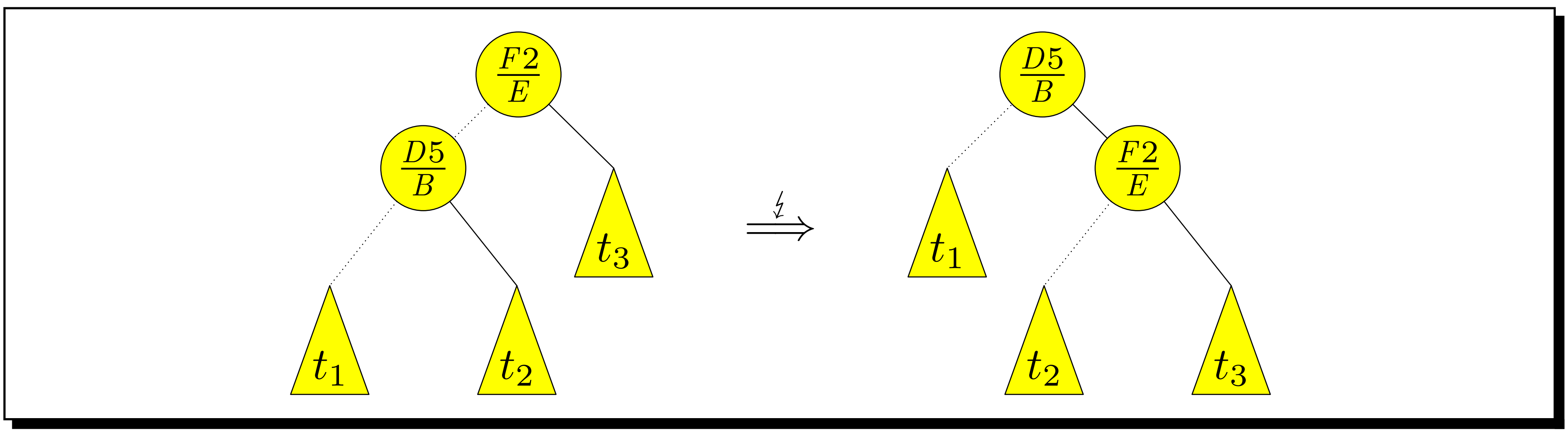
Destructors: $delete$

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

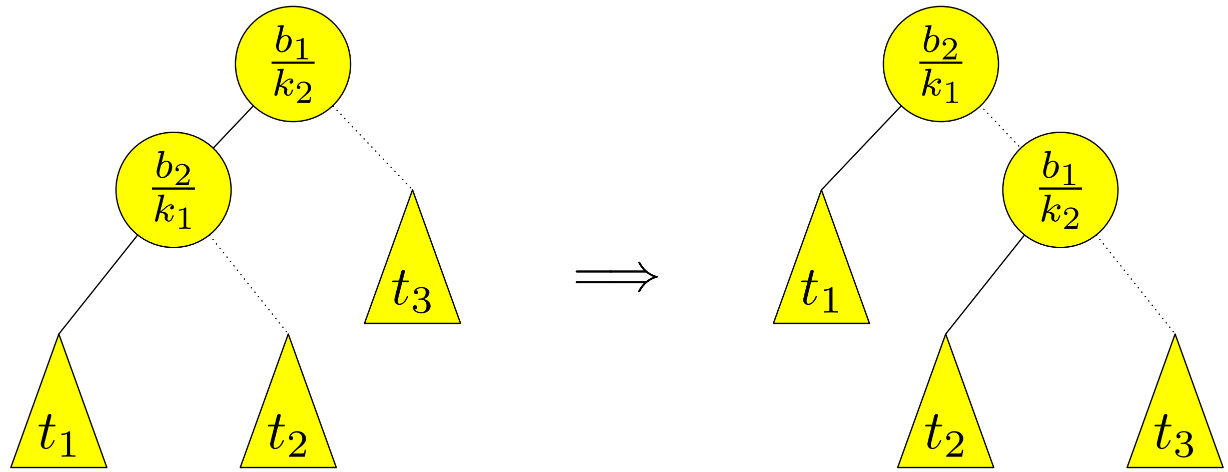
Adding a balancing scheme

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

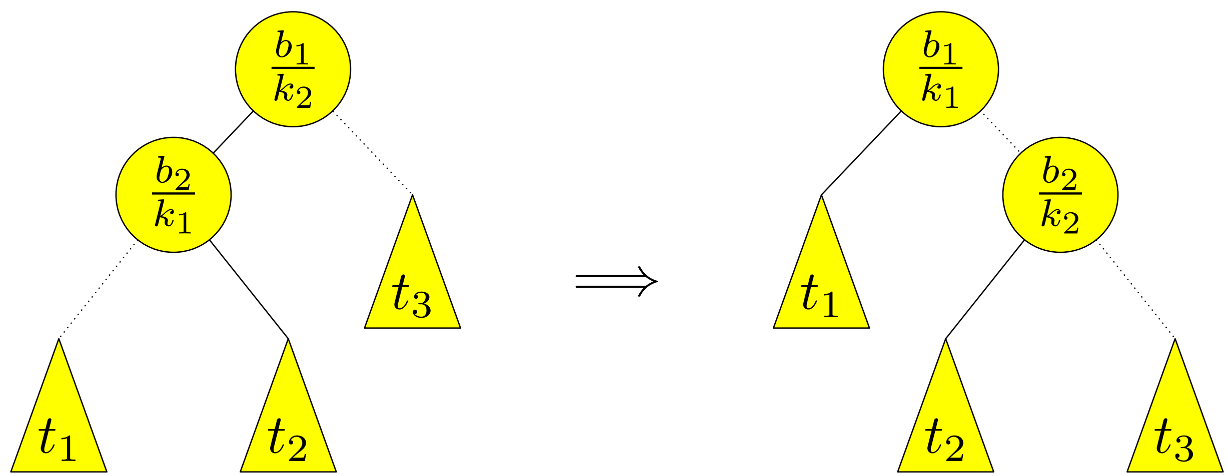
Most balancing schemes such as red-black trees use rotations to restore balancing invariants. However, rotations do not preserve the semi-heap property:



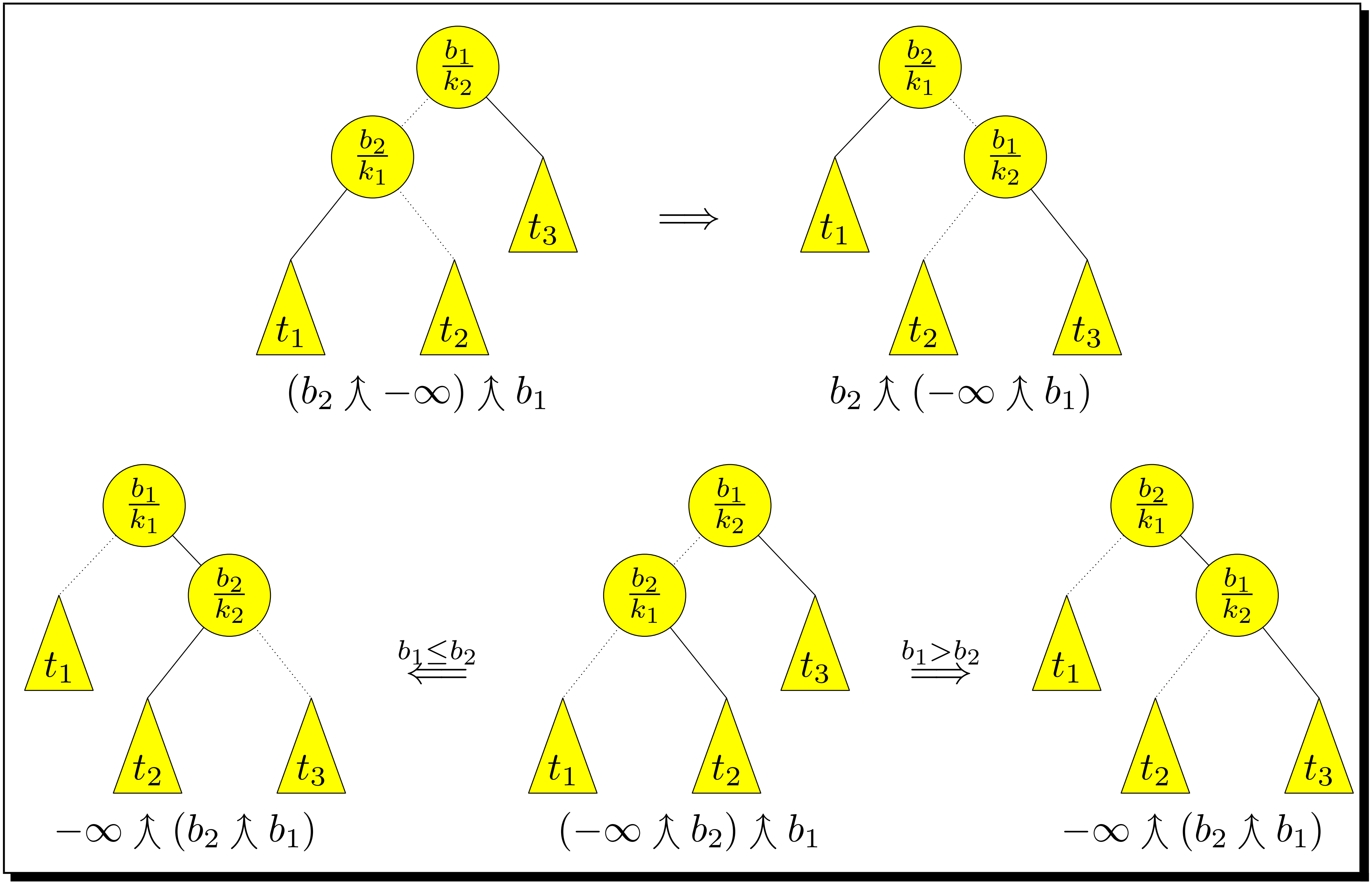
Single rotation



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



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



$\frac{b_1}{k_2}$
$\frac{b_2}{k_1}$
t_1
t_2
t_3
$\Longrightarrow$
$(b_2 \curlywedge -\infty) \curlywedge b_1$
$b_2 \curlywedge (-\infty \curlywedge b_1)$
$\xLeftarrow{b_1 \leq b_2}$
$\xRightarrow{b_1 > b_2}$
$-\infty \curlywedge (b_2 \curlywedge b_1)$
$(-\infty \curlywedge b_2) \curlywedge b_1$
$-\infty \curlywedge (b_2 \curlywedge b_1)$

Learning targets

- ✘ Persistence
- ✘ Red-black trees
- ✘ Smart constructors
- ✘ Number systems
- ✘ Views
- ✘ Tournament trees
- ✘ Priority search pennants

Appendix: $foldr$

The function $foldr$ captures a common pattern of recursion on lists (it is a so-called *catamorphism*; the greak preposition $\kappa\alpha\tau\alpha$ means "downwards").

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

For example,

$$
foldr\ (\star)\ e\ (a_1 : a_2 : \cdots : a_n : [\,]) \quad = \quad a_1 \star (a_2 \star (\cdots \star (a_n \star e)\ \cdots)) \ .
$$

Appendix: $foldl$

The function $foldl$ is similar to $foldr$, except that the parentheses group from the left.

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

For example,

$$foldl\ (\star)\ e\ (a_1 : a_2 : \cdots : a_n : [\,]) \;=\; (\cdots\ ((e \star a_1) \star a_2) \star \cdots) \star a_n\ .$$

Appendix: *foldm*

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

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