Script **generated by TTT**

Title: FDS (05.07.2019)

Date: Fri Jul 05 08:36:46 CEST 2019

Duration: 76:19 min

Pages: 99

Chapter 9

Priority Queues

Priority queue informally

Collection of elements with priorities

Priority queue informally

Collection of elements with priorities

Operations:

Priority queue informally

Collection of elements with priorities

Operations:

- empty
- emptiness test
- insert
- get element with minimal priority
- delete element with minimal priority

We focus on the priorities:
element = priority

Priority queues are multisets

The same element can be contained multiple times in a priority queue

Interface of implementation

The type of elements (= priorities) $'a$ is a linear order

Interface of implementation

The type of elements (= priorities) $'a$ is a linear order

An implementation of a priority queue of elements of type $'a$ must provide

Interface of implementation

The type of elements (= priorities) $'a$ is a linear order

An implementation of a priority queue of elements of type $'a$ must provide

- An implementation type $'q$

More operations

- $merge :: 'q \Rightarrow 'q \Rightarrow 'q$
  Often provided

More operations

- $merge :: 'q \Rightarrow 'q \Rightarrow 'q$
  Often provided
- decrease key/priority
  A bit tricky in functional setting

Correctness of implementation

A priority queue represents a multiset of priorities.
Correctness proof requires:

Abstraction function: $mset :: 'q \Rightarrow 'a\ multiset$

Correctness of implementation

A priority queue represents a multiset of priorities.
Correctness proof requires:

Abstraction function: $mset :: 'q \Rightarrow 'a\ multiset$
Invariant: $invar :: 'q \Rightarrow bool$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

$mset\ empty = \{\#\}$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

$mset\ empty = \{\#\}$

$is\_empty\ q = (mset\ q = \{\#\})$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

$mset\ empty = \{\#\}$

$is\_empty\ q = (mset\ q = \{\#\})$

$mset\ (insert\ x\ q) = mset\ q + \{\#x\#\}$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

$mset\ empty = \{\#\}$

$is\_empty\ q = (mset\ q = \{\#\})$

$mset\ (insert\ x\ q) = mset\ q + \{\#x\#\}$

$mset\ q \neq \{\#\} \Longrightarrow get\_min\ q = Min\_mset\ (mset\ q)$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

$mset\ empty = \{\#\}$

$is\_empty\ q = (mset\ q = \{\#\})$

$mset\ (insert\ x\ q) = mset\ q + \{\#x\#\}$

$mset\ q \neq \{\#\} \Longrightarrow get\_min\ q = Min\_mset\ (mset\ q)$

$mset\ q \neq \{\#\} \Longrightarrow$
$mset\ (del\_min\ q) = mset\ q - \{\#get\_min\ q\#\}$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

$mset\ empty = \{\#\}$
$is\_empty\ q = (mset\ q = \{\#\})$
$mset\ (insert\ x\ q) = mset\ q + \{\#x\#\}$
$mset\ q \neq \{\#\} \Longrightarrow get\_min\ q = Min\_mset\ (mset\ q)$
$mset\ q \neq \{\#\} \Longrightarrow$
$mset\ (del\_min\ q) = mset\ q - \{\#get\_min\ q\#\}$

$invar\ empty$
$invar\ (insert\ x\ q)$
$invar\ (del\_min\ q)$

Terminology

A binary tree is a *heap* if for every subtree the root is $\leq$ all elements in that subtree.

Terminology

A binary tree is a *heap* if for every subtree the root is $\leq$ all elements in that subtree.

$heap\ \langle\rangle = True$
$heap\ \langle l,\ m,\ r\rangle =$
$(heap\ l \wedge heap\ r \wedge (\forall x{\in}set\_tree\ l \cup set\_tree\ r.\ m \leq x))$

Terminology

A binary tree is a *heap* if for every subtree the root is $\leq$ all elements in that subtree.

$heap\ \langle\rangle = True$
$heap\ \langle l,\ m,\ r\rangle =$
$(heap\ l \wedge heap\ r \wedge (\forall x{\in}set\_tree\ l \cup set\_tree\ r.\ m \leq x))$

The term "heap" is frequently used synonymously with "priority queue".

Priority queue via heap

- $empty = \langle\rangle$

Priority queue via heap

- $empty = \langle\rangle$
- $is\_empty\ h = (h = \langle\rangle)$
- $get\_min\ \langle\_,\ a,\ \_\rangle = a$

Priority queue via heap

- $empty = \langle\rangle$
- $is\_empty\ h = (h = \langle\rangle)$
- $get\_min\ \langle\_,\ a,\ \_\rangle = a$
- Assume we have $merge$

Priority queue via heap

- $empty = \langle\rangle$
- $is\_empty\ h = (h = \langle\rangle)$
- $get\_min\ \langle\_,\ a,\ \_\rangle = a$
- Assume we have $merge$
- $insert\ a\ t = merge\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ t$

Priority queue via heap

- $empty = \langle\rangle$
- $is\_empty\ h = (h = \langle\rangle)$
- $get\_min\ \langle\_,\ a,\ \_\rangle = a$
- Assume we have $merge$
- $insert\ a\ t = merge\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ t$
- $del\_min\ \langle l,\ a,\ r\rangle = merge\ l\ r$

Priority queue via heap

A naive merge:

$merge\ t_1\ t_2 =$ (case (t_1,t_2) of
$\quad(\langle\rangle,\ \_) \Rightarrow t_2\ |$
$\quad(\_,\ \langle\rangle) \Rightarrow t_1\ |$

Priority queue via heap

A naive merge:

$merge\ t_1\ t_2 =$ (case (t_1,t_2) of
$\quad(\langle\rangle,\ \_) \Rightarrow t_2\ |$
$\quad(\_,\ \langle\rangle) \Rightarrow t_1\ |$
$\quad(\langle l_1,a_1,r_1\rangle,\ \langle l_2,a_2,r_2\rangle) \Rightarrow$
$\quad\quad$ if $a_1 \leq a_2$ then $\langle merge\ l_1\ r_1,\ a_1,\ t_2\rangle$
$\quad\quad$ else $\langle t_1,\ a_2,\ merge\ l_2\ r_2\rangle$

Priority queue via heap

A naive merge:

$merge\ t_1\ t_2 =$ (case (t_1,t_2) of
$\quad(\langle\rangle,\ \_) \Rightarrow t_2\ |$
$\quad(\_,\ \langle\rangle) \Rightarrow t_1\ |$
$\quad(\langle l_1,a_1,r_1\rangle,\ \langle l_2,a_2,r_2\rangle) \Rightarrow$
$\quad\quad$ if $a_1 \leq a_2$ then $\langle merge\ l_1\ r_1,\ a_1,\ t_2\rangle$
$\quad\quad$ else $\langle t_1,\ a_2,\ merge\ l_2\ r_2\rangle$

Challenge: how to maintain some kind of balance

Priority queue via heap

- $empty = \langle\rangle$
- $is\_empty\ h = (h = \langle\rangle)$
- $get\_min\ \langle\_,\ a,\ \_\rangle = a$
- Assume we have $merge$
- $insert\ a\ t = merge\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ t$
- $del\_min\ \langle l,\ a,\ r\rangle = merge\ l\ r$

Priority queue via heap

A naive merge:

$merge\ t_1\ t_2 =$ (case (t_1,t_2) of
  $(\langle\rangle,\ \_) \Rightarrow t_2$ |
  $(\_,\ \langle\rangle) \Rightarrow t_1$ |
  $(\langle l_1,a_1,r_1\rangle,\ \langle l_2,a_2,r_2\rangle) \Rightarrow$
    if $a_1 \le a_2$ then $\langle merge\ l_1\ r_1,\ a_1,\ t_2\rangle$
    else $\langle t_1,\ a_2,\ merge\ l_2\ r_2\rangle$

Priority queue via heap

A naive merge:

$merge\ t_1\ t_2 =$ (case (t_1,t_2) of
  $(\langle\rangle,\ \_) \Rightarrow t_2$ |
  $(\_,\ \langle\rangle) \Rightarrow t_1$ |
  $(\langle l_1,a_1,r_1\rangle,\ \langle l_2,a_2,r_2\rangle) \Rightarrow$
    if $a_1 \le a_2$ then $\langle merge\ l_1\ r_1,\ a_1,\ t_2\rangle$
    else $\langle t_1,\ a_2,\ merge\ l_2\ r_2\rangle$

Challenge: how to maintain some kind of balance

Priority queue via heap

A naive merge:

$merge\ t_1\ t_2 = $ (case (t_1,t_2) of
  $(\langle\rangle,\ \_) \Rightarrow t_2\ |$
  $(\_,\ \langle\rangle) \Rightarrow t_1\ |$
  $(\langle l_1,a_1,r_1\rangle,\ \langle l_2,a_2,r_2\rangle) \Rightarrow$
    if $a_1 \leq a_2$ then $\langle merge\ l_1\ r_1,\ a_1,\ t_2\rangle$
    else $\langle t_1,\ a_2,\ merge\ l_2\ r_2\rangle$

Challenge: how to maintain some kind of balance

Leftist tree informally

The *rank* of a tree is the depth of the rightmost leaf.

Leftist tree informally

The *rank* of a tree is the depth of the rightmost leaf.

In a *leftist tree*, the rank of every left child is $\geq$ the rank of its right sibling.

Leftist tree informally

The *rank* of a tree is the depth of the rightmost leaf.

In a *leftist tree*, the rank of every left child is $\geq$ the rank of its right sibling.

Merge descends along the right spine.
Thus rank bounds number of steps.

Leftist tree informally

The *rank* of a tree is the depth of the rightmost leaf.

In a *leftist tree*, the rank of every left child is $\geq$ the rank of its right sibling.

Merge descends along the right spine.
Thus rank bounds number of steps.

If rank of right child gets too large: swap with left child.

Implementation type

datatype
$'a\ lheap = Leaf \mid Node\ ('a\ tree)\ 'a\ nat\ ('a\ tree)$

Implementation type

datatype
$'a\ lheap = Leaf \mid Node\ ('a\ tree)\ 'a\ nat\ ('a\ tree)$

Abbreviations $\langle\rangle$ and $\langle l,\ a,\ n,\ r\rangle$ as usual

Abstraction function:
$mset\_tree :: 'a\ lheap \Rightarrow 'a\ multiset$
$mset\_tree\ \langle\rangle = \{\#\}$
$mset\_tree\ \langle l,\ a,\ \_,\ r\rangle =$
$\{\#a\#\} + mset\_tree\ l + mset\_tree\ r$

Leftist tree

$rank :: 'a\ lheap \Rightarrow nat$

Leftist tree

$rank :: 'a\ lheap \Rightarrow nat$

$rank\ \langle\rangle = 0$
$rank\ \langle\_,\ \_,\ \_,\ r\rangle = rank\ r + 1$

Leftist tree

$rank :: 'a\ lheap \Rightarrow nat$

$rank\ \langle\rangle = 0$
$rank\ \langle\_,\ \_,\ \_,\ r\rangle = rank\ r + 1$

Node $\langle l,\ a,\ n,\ r\rangle$: $n =$ rank of node

Leftist tree

$rank :: 'a\ lheap \Rightarrow nat$

$rank\ \langle\rangle = 0$
$rank\ \langle\_,\ \_,\ \_,\ r\rangle = rank\ r + 1$

Node $\langle l,\ a,\ n,\ r\rangle$: $n =$ rank of node

$ltree :: 'a\ lheap \Rightarrow bool$

$ltree\ \langle\rangle = True$
$ltree\ \langle l,\ \_,\ n,\ r\rangle =$
$(n = rank\ r + 1 \wedge rank\ r \leq rank\ l \wedge ltree\ l \wedge ltree\ r)$

Leftist heap invariant

$invar\ h = (heap\ h \wedge ltree\ h)$

merge

Principle: descend on the right

merge

Principle: descend on the right

$merge\ \langle\rangle\ t_2 = t_2$
$merge\ t_1\ \langle\rangle = t_1$

merge

Principle: descend on the right

$merge\ \langle\rangle\ t_2 = t_2$
$merge\ t_1\ \langle\rangle = t_1$

$merge\ (\langle l_1,\ a_1,\ n_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ n_2,\ r_2\rangle =: t_2) =$
$(\textsf{if}\ a_1 \leq a_2\ \textsf{then}\ node\ l_1\ a_1\ (merge\ r_1\ t_2)$
$\ \textsf{else}\ node\ l_2\ a_2\ (merge\ t_1\ r_2))$

merge

Principle: descend on the right

15 Priority Queues

16 Leftist Heap

17 Priority Queue via Braun Tree

18 Binomial Heap

19 Skew Binomial Heap

merge

Principle: descend on the right

$merge\ \langle\rangle\ t_2 = t_2$
$merge\ t_1\ \langle\rangle = t_1$
$merge\ (\langle l_1,\ a_1,\ n_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ n_2,\ r_2\rangle =: t_2) =$
(if $a_1 \leq a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ t_2)$
else $node\ l_2\ a_2\ (merge\ t_1\ r_2)$)

$node :: 'a\ lheap \Rightarrow 'a \Rightarrow 'a\ lheap \Rightarrow 'a\ lheap$

$node\ l\ a\ r =$
(let $rl = rk\ l$; $rr = rk\ r$
in if $rr \leq rl$ then $\langle l,\ a,\ rr + 1,\ r\rangle$ else $\langle r,\ a,\ rl + 1,\ l\rangle$)
where $rk\ \langle\_,\ \_,\ n,\ \_\rangle = n$

merge

$merge\ (\langle l_1,\ a_1,\ n_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ n_2,\ r_2\rangle =: t_2) =$
(if $a_1 \leq a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ t_2)$
else $node\ l_2\ a_2\ (merge\ t_1\ r_2)$)

merge

$merge\ (\langle l_1,\ a_1,\ n_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ n_2,\ r_2\rangle =: t_2) =$
(if $a_1 \leq a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ t_2)$
else $node\ l_2\ a_2\ (merge\ t_1\ r_2)$)

Function $merge$ terminates because ?
decreases with every recursive call.

merge

$merge\ (\langle l_1,\ a_1,\ n_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ n_2,\ r_2\rangle =: t_2) =$
(if $a_1 \le a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ t_2)$
else $node\ l_2\ a_2\ (merge\ t_1\ r_2)$)

Function *merge* terminates because $size\ t_1 + size\ t_2$ decreases with every recursive call.

Logarithmic complexity

Correlation of rank and size:
Lemma $ltree\ t \Longrightarrow 2^{rank\ t} \le |t|_1$

merge

$merge\ (\langle l_1,\ a_1,\ n_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ n_2,\ r_2\rangle =: t_2) =$
(if $a_1 \le a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ t_2)$
else $node\ l_2\ a_2\ (merge\ t_1\ r_2)$)

Function *merge* terminates because ? decreases with every recursive call.

Implementation type

datatype
$'a\ lheap = Leaf \mid Node\ ('a\ tree)\ 'a\ nat\ ('a\ tree)$

Abbreviations $\langle\rangle$ and $\langle l,\ a,\ n,\ r\rangle$ as usual

Abstraction function:
$mset\_tree :: 'a\ lheap \Rightarrow 'a\ multiset$

Logarithmic complexity

Correlation of rank and size:

Lemma $ltree\ t \Longrightarrow 2^{rank\ t} \leq |t|_1$

Complexity measures t_merge, t_insert t_del_min: count calls of $merge$.

Logarithmic complexity

Correlation of rank and size:

Lemma $ltree\ t \Longrightarrow 2^{rank\ t} \leq |t|_1$

Complexity measures t_merge, t_insert t_del_min: count calls of $merge$.

Lemma $t\_merge\ l\ r \leq rank\ l + rank\ r + 1$

$merge$

$merge\ (\langle l_1, a_1, n_1, r_1\rangle =: t_1)\ (\langle l_2, a_2, n_2, r_2\rangle =: t_2) =$
(if $a_1 \leq a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ t_2)$
else $node\ l_2\ a_2\ (merge\ t_1\ r_2)$)

Function $merge$ terminates because $size\ t_1 + size\ t_2$ decreases with every recursive call.

Functional correctness proofs

including preservation of $invar$

Straightforward

Logarithmic complexity

Correlation of rank and size:

Lemma $ltree\ t \Longrightarrow 2^{rank\ t} \leq |t|_1$

Complexity measures t_merge, $t\_insert\ t\_del\_min$: count calls of $merge$.

Lemma $t\_merge\ l\ r \leq rank\ l + rank\ r + 1$

Corollary $[\![ltree\ l;\ ltree\ r]\!]$
$\Longrightarrow t\_merge\ l\ r \leq \log_2 |l|_1 + \log_2 |r|_1 + 1$

Corollary
$ltree\ t \Longrightarrow t\_insert\ x\ t \leq \log_2 |t|_1 + 2$

Logarithmic complexity

Correlation of rank and size:

Lemma $ltree\ t \Longrightarrow 2^{rank\ t} \leq |t|_1$

Complexity measures t_merge, $t\_insert\ t\_del\_min$: count calls of $merge$.

Lemma $t\_merge\ l\ r \leq rank\ l + rank\ r + 1$

Corollary $[\![ltree\ l;\ ltree\ r]\!]$
$\Longrightarrow t\_merge\ l\ r \leq \log_2 |l|_1 + \log_2 |r|_1 + 1$

Corollary
$ltree\ t \Longrightarrow t\_insert\ x\ t \leq \log_2 |t|_1 + 2$

Corollary
$ltree\ t \Longrightarrow t\_del\_min\ t \leq 2 * \log_2 |t|_1 + 1$

What is a Braun tree?

$braun :: 'a\ tree \Rightarrow bool$

What is a Braun tree?

$braun :: 'a\ tree \Rightarrow bool$

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$

What is a Braun tree?

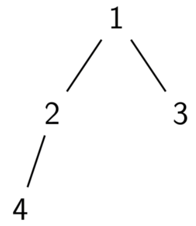
$braun :: {'a}\ tree \Rightarrow bool$

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \le |l| \wedge |l| \le |r| + 1 \wedge braun\ l \wedge braun\ r)$

What is a Braun tree?

$braun :: {'a}\ tree \Rightarrow bool$

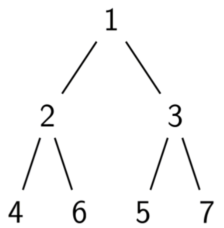
$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \le |l| \wedge |l| \le |r| + 1 \wedge braun\ l \wedge braun\ r)$



What is a Braun tree?

$braun :: {'a}\ tree \Rightarrow bool$

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \le |l| \wedge |l| \le |r| + 1 \wedge braun\ l \wedge braun\ r)$



Idea of invariant maintenance

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \le |l| \wedge |l| \le |r| + 1 \wedge braun\ l \wedge braun\ r)$

Idea of invariant maintenance

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$

Add element:

Idea of invariant maintenance

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$

Add element: to right subtree, then swap subtrees

Idea of invariant maintenance

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$

Add element: to right subtree, then swap subtrees
Goal: $|l| \leq |r| + 1 \wedge |r| + 1 \leq |l| + 1$

Idea of invariant maintenance

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$

Add element: to right subtree, then swap subtrees
Goal: $|l| \leq |r| + 1 \wedge |r| + 1 \leq |l| + 1$ $\square$

Priority queue implementation

Implementation type: $'a\ tree$

Invariants: $heap$ and $braun$

Priority queue implementation

Implementation type: $'a\ tree$

Invariants: $heap$ and $braun$

No $merge$

$insert$

$insert :: 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$

$insert$

$insert :: 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$

$insert\ a\ \langle\rangle = \langle\langle\rangle,\ a,\ \langle\rangle\rangle$

$insert\ a\ \langle l,\ x,\ r\rangle =$

(if $a < x$ then $\langle insert\ x\ r,\ a,\ l\rangle$ else $\langle insert\ a\ r,\ x,\ l\rangle$)

del_min

$del\_min :: 'a\ tree \Rightarrow 'a\ tree$

del_min

$del\_min :: 'a\ tree \Rightarrow 'a\ tree$

$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle\langle\rangle, x, r\rangle = \langle\rangle$
$del\_min\ \langle l, x, r\rangle =$
$(\text{let } (y, l') = del\_left\ l \text{ in } sift\_down\ r\ y\ l')$

del_min

$del\_min :: 'a\ tree \Rightarrow 'a\ tree$

$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle\langle\rangle, x, r\rangle = \langle\rangle$
$del\_min\ \langle l, x, r\rangle =$
$(\text{let } (y, l') = del\_left\ l \text{ in } sift\_down\ r\ y\ l')$

1. Delete leftmost element y
2. Sift y from the root down

Reminiscent of heapsort, but not quite . . .

del_left

$del\_left :: 'a\ tree \Rightarrow 'a \times 'a\ tree$

sift_down

$sift\_down :: 'a\ tree \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$

del_left

$del\_left :: 'a\ tree \Rightarrow 'a \times 'a\ tree$

sift_down

$sift\_down :: 'a\ tree \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$

$sift\_down\ \langle\rangle\ a\ \langle\rangle = \langle\langle\rangle,\ a,\ \langle\rangle\rangle$

$sift\_down\ \langle\langle\rangle,\ x,\ \langle\rangle\rangle\ a\ \langle\rangle =$
$(\text{if } a \le x \text{ then } \langle\langle\langle\rangle,\ x,\ \langle\rangle\rangle,\ a,\ \langle\rangle\rangle$
$\ \text{else } \langle\langle\langle\rangle,\ a,\ \langle\rangle\rangle,\ x,\ \langle\rangle\rangle)$

$sift\_down\ (\langle l_1,\ x_1,\ r_1\rangle =: t_1)\ a\ (\langle l_2,\ x_2,\ r_2\rangle =: t_2) =$
$\text{if } a \le x_1 \wedge a \le x_2 \text{ then } \langle t_1,\ a,\ t_2\rangle$
$\text{else if } x_1 \le x_2 \text{ then } \langle sift\_down\ l_1\ a\ r_1,\ x_1,\ t_2\rangle$
$\quad \text{else } \langle t_1,\ x_2,\ sift\_down\ l_2\ a\ r_2\rangle$

sift_down

$sift\_down :: 'a\ tree \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$

$sift\_down\ \langle\rangle\ a\ \langle\rangle = \langle\langle\rangle,\ a,\ \langle\rangle\rangle$

$sift\_down\ \langle\langle\rangle,\ x,\ \langle\rangle\rangle\ a\ \langle\rangle =$
$(\text{if } a \le x \text{ then } \langle\langle\langle\rangle,\ x,\ \langle\rangle\rangle,\ a,\ \langle\rangle\rangle$
$\ \text{else } \langle\langle\langle\rangle,\ a,\ \langle\rangle\rangle,\ x,\ \langle\rangle\rangle)$

del_left

$del\_left :: 'a\ tree \Rightarrow 'a \times 'a\ tree$

Functional correctness proofs for *del_min*

Many lemmas, mostly straightforward

Logarithmic complexity

Running time of *insert*, *del_left* and *sift_down* (and therefore *del_min*) bounded by height

Logarithmic complexity

Running time of *insert*, *del_left* and *sift_down* (and therefore *del_min*) bounded by height

Remember: $braun\ t \Longrightarrow 2^{h(t)} \leq 2 * |t| + 1$

Source of code

Based on code from
L.C. Paulson. *ML for the Working Programmer.* 1996
based on code from Chris Okasaki.

Sorting with priority queue

Sorting with priority queue

$pq\ [] = empty$
$pq\ (x\#xs) = insert\ x\ (pq\ xs)$

Sorting with priority queue

$pq\ [] = empty$
$pq\ (x\#xs) = insert\ x\ (pq\ xs)$

$mins\ q =$
(if $is\_empty\ q$ then $[]$
else $get\_min\ h\ \#\ mins\ (del\_min\ h)$)

Sorting with priority queue

$pq\ [] = empty$
$pq\ (x\#xs) = insert\ x\ (pq\ xs)$

$mins\ q =$
(if $is\_empty\ q$ then $[]$
else $get\_min\ h \mathbin{\#} mins\ (del\_min\ h))$

$sort\_pq = mins \circ pq$

Sorting with priority queue

$pq\ [] = empty$
$pq\ (x\#xs) = insert\ x\ (pq\ xs)$

$mins\ q =$
(if $is\_empty\ q$ then $[]$
else $get\_min\ h \mathbin{\#} mins\ (del\_min\ h))$

$sort\_pq = mins \circ pq$

Complexity of $sort$: $O(n \log n)$
if all priority queue functions have complexity $O(\log n)$

15 Priority Queues

16 Leftist Heap

17 Priority Queue via Braun Tree

18 Binomial Heap

19 Skew Binomial Heap