Script **generated by TTT**

Title: Lammich: FDS Tutorial (30.06.2017)

Date: Fri Jun 30 08:36:00 CEST 2017

Duration: 90:38 min

Pages: 72

Chapter 9

Priority Queues

12 Priority Queues

13 Leftist Heap

14 Priority Queues Based on Braun Trees

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

Priority queues are multisets

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

$\Longrightarrow$

The abstract view of a priority queue is a multiset

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 $'q$
- $empty :: 'q$
- $is\_empty :: 'q \Rightarrow bool$
- $insert :: 'a \Rightarrow 'q \Rightarrow 'q$
- $get\_min :: 'q \Rightarrow 'a$
- $del\_min :: 'q \Rightarrow 'q$

More operations

- $merge :: 'q \Rightarrow 'q \Rightarrow 'q$
  Often provided
- decrease key/priority
  Not easy 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 $invar\ q \Longrightarrow$

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

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

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

$mset\_tree\ h \neq \{\#\} \Longrightarrow$
$mset\ (del\_min\ q) = mset\ q - \{\#Min\_mset\ (mset\ q)\#\}$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

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

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

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

$mset\_tree\ h \neq \{\#\} \Longrightarrow$
$mset\ (del\_min\ q) = mset\ q - \{\#Min\_mset\ (mset\ q)\#\}$

$get\_min\ q = Min\_mset\ (mset\ q)$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

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

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

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

$mset\_tree\ h \neq \{\#\} \Longrightarrow$
$mset\ (del\_min\ q) = mset\ q - \{\#Min\_mset\ (mset\ q)\#\}$

$get\_min\ q = Min\_mset\ (mset\ q)$

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

Terminology

A tree is a heap if for every subtree
the root is $\geq$ all elements in the subtrees.

Terminology

A tree is a heap if for every subtree
the root is $\geq$ all elements in the subtrees.

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

Priority queue via heap

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:

Priority queue via heap

A naive merge:

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

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

12 Priority Queues

13 Leftist Heap

14 Priority Queues Based on Braun Trees

Leftist_Heap.thy

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

Insertions are done on the right. Thus rank bounds number of descends.

Implementation type

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

Implementation type

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

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

Implementation type

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

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

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

Leftist tree

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

Leftist tree

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

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

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

Leftist heap invariant

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

Why leftist tree?

Why leftist tree?

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

Lemma Execution time of $merge\ t_1\ t_2$ is bounded by $rank\ t_1 + rank\ t_2$

Why leftist tree?

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

Lemma Execution time of $merge\ t_1\ t_2$ is bounded by $rank\ t_1 + rank\ t_2$

$merge$

Principle: descend on the right

$merge$

Principle: descend on the right

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

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

$merge$

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

merge

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

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

merge

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

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

Logarithmic complexity

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

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

Lemma $t\_merge\ l\ r \le rank\ l + rank\ r + 1$
Lemma $[\![ltree\ l;\ ltree\ r]\!]$
$\Longrightarrow t\_merge\ l\ r \le \log_2 |l|_1 + \log_2 |r|_1 + 1$

Logarithmic complexity

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

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

Lemma $t\_merge\ l\ r \le rank\ l + rank\ r + 1$
Lemma $[\![ltree\ l;\ ltree\ r]\!]$
$\Longrightarrow t\_merge\ l\ r \le \log_2 |l|_1 + \log_2 |r|_1 + 1$

Lemma
$ltree\ t \Longrightarrow t\_insert\ x\ t \le \log_2 |t|_1 + 2$

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

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

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

Can we avoid the rank info in each node?

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

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

12 Priority Queues

13 Leftist Heap

14 Priority Queues Based on Braun Trees

Archive of Formal Proofs

https://www.isa-afp.org/entries/Priority_Queue_Braun.shtml

What is a Braun tree?

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

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

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

Idea of Invariant Maintenance

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

Idea of Invariant Maintenance

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

Add element:

Idea of Invariant Maintenance

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

Add element: to right subtree, then swap subtrees

Idea of Invariant Maintenance

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

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

Remove element:

Idea of Invariant Maintenance

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

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

Remove element: from left subtree, then swap subtrees

Idea of Invariant Maintenance

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

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

Remove element: from left subtree, then swap subtrees
Goal: $size\ l - 1 \leq size\ r \wedge size\ r \leq size\ l$ □

Priority queue implementation

Implementation type: ordinary binary trees

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

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

Correctness and preservation of invariant straightforward.

insert

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

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

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

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

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

Idea: Delete leftmost element y, put it at top, and sift it down to adequate position.

sift_down

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

sift_down

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

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

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

$sift\_down\ \langle l_1,\ x_1,\ r_1\rangle\ a\ \langle l_2,\ x_2,\ r_2\rangle =$
(if $a \le x_1 \wedge a \le x_2$ then $\langle\langle l_1,\ x_1,\ r_1\rangle,\ a,\ \langle l_2,\ x_2,\ r_2\rangle\rangle$
else if $x_1 \le x_2$ then $\langle sift\_down\ l_1\ a\ r_1,\ x_1,\ \langle l_2,\ x_2,\ r_2\rangle\rangle$
else $\langle\langle l_1,\ x_1,\ r_1\rangle,\ x_2,\ sift\_down\ l_2\ a\ r_2\rangle)$

sift_down

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

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

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

$sift\_down\ \langle l_1,\ x_1,\ r_1\rangle\ a\ \langle l_2,\ x_2,\ r_2\rangle =$
(if $a \le x_1 \wedge a \le x_2$ then $\langle\langle l_1,\ x_1,\ r_1\rangle,\ a,\ \langle l_2,\ x_2,\ r_2\rangle\rangle$
else if $x_1 \le x_2$ then $\langle sift\_down\ l_1\ a\ r_1,\ x_1,\ \langle l_2,\ x_2,\ r_2\rangle\rangle$
else $\langle\langle l_1,\ x_1,\ r_1\rangle,\ x_2,\ sift\_down\ l_2\ a\ r_2\rangle)$

Maintenance of *braun*: Preserves structure of tree.

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

Logarithmic complexity

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

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

Sorting with priority queue

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

Sorting with priority queue

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

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 $get\_min\ h\ \#\ mins\ (del\_min\ h)$)

$sort\_pq = mins \circ pq$