Script generated by TTT

Title: Lammich: FDS (22.06.2018)

Date: Fri Jun 22 08:37:24 CEST 2018

Duration: 99:55 min

Pages: 80

Chapter 9

Priority Queues

Priority queue informally

Collection of elements with priorities

Operations:

- empty

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

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 $'q$
- $empty :: 'q$

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 $'q$
- $insert :: 'a \Rightarrow 'q \Rightarrow 'q$
- $del\_min :: 'q \Rightarrow 'q$

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 $invar\ q \Longrightarrow$

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

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

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 $merge$
- $insert\ a\ t = merge\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ t$
- $del\_min\ \langle l,\ a,\ r\rangle = merge\ l\ r$

181

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
$(\langle\rangle,\ \_) \Rightarrow t_2\ |$
$(\_,\ \langle\rangle) \Rightarrow t_1\ |$

182

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.

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

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

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

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

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

$ltree :: 'a\ lheap \Rightarrow bool$
$ltree\ \langle\rangle = True$
$ltree\ \langle n, l, \_, r\rangle =$
$(n = rank\ r + 1 \wedge rank\ r \leq rank\ l \wedge ltree\ l \wedge ltree\ r)$

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

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

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

merge

Principle: descend on the right

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

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

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

merge

Principle: descend on the right

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

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

merge

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

merge

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

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

190

Functional correctness proofs

including preservation of $invar$

Straightforward

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 $merge$.

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

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

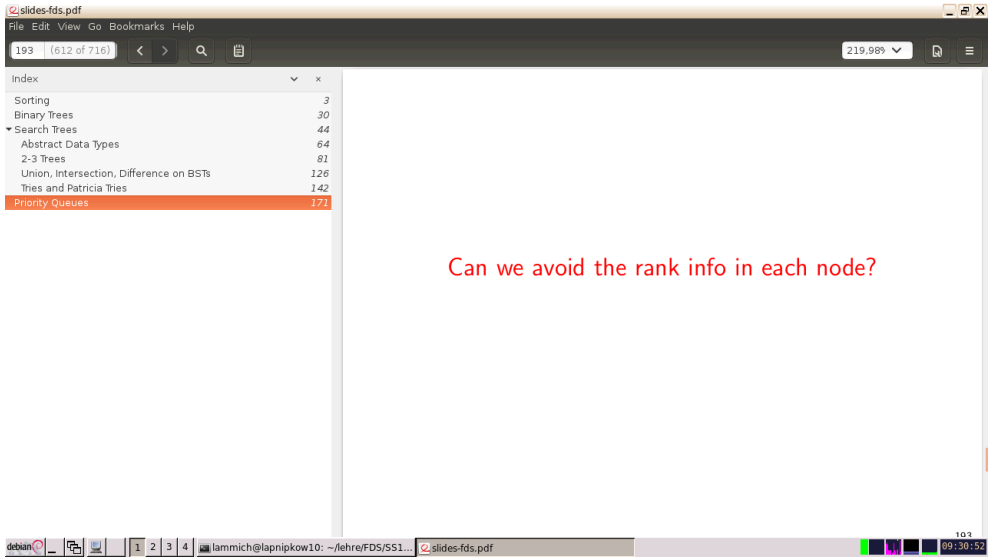
Logarithmic complexity

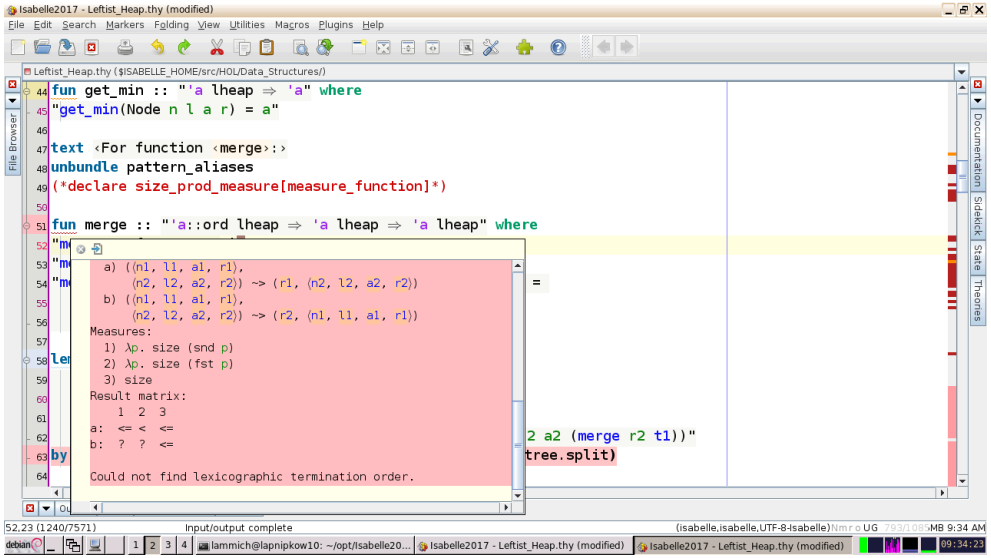
Correlation of rank and size:
Lemma $ltree\ t \Longrightarrow 2^{rank\ t} \leq |t|_1$

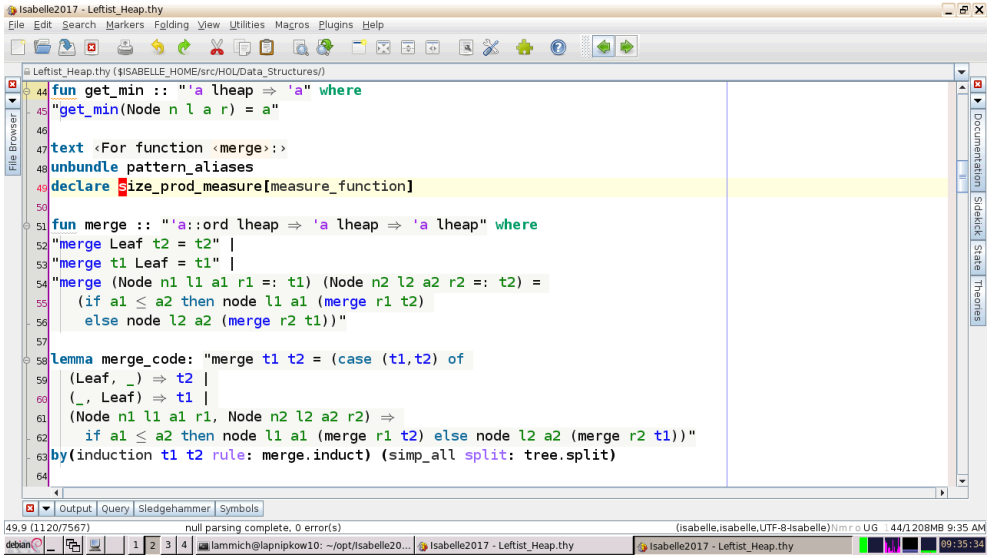
Complexity measures t_merge, t_insert t_del_min: count calls of $merge$.

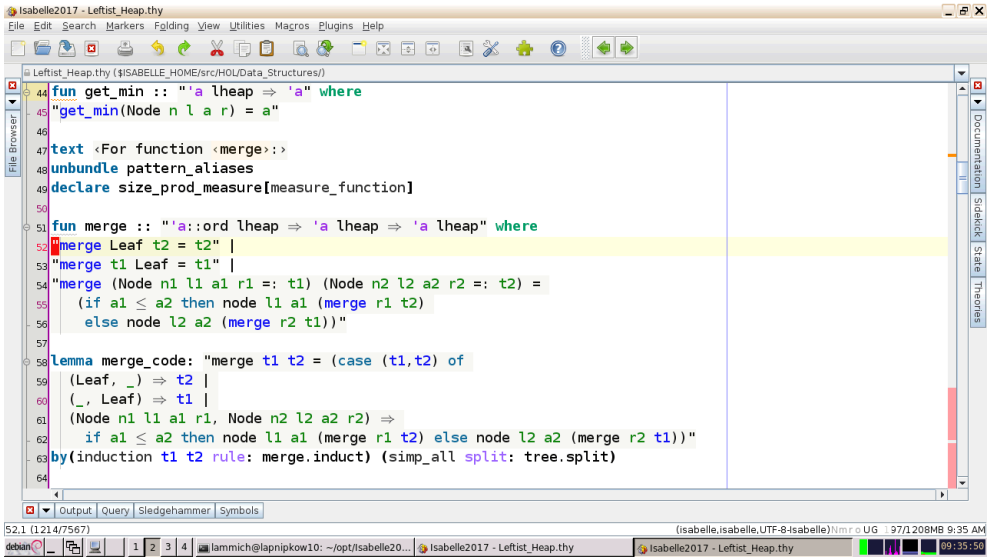
Lemma $t\_merge\ l\ r \leq rank\ l + rank\ r + 1$
Corollary $[\![ltree\ l;\ ltree\ r]\!]$
$\Longrightarrow t\_merge\ l\ r \leq \log_2 |l|_1 + \log_2 |r|_1 + 1$

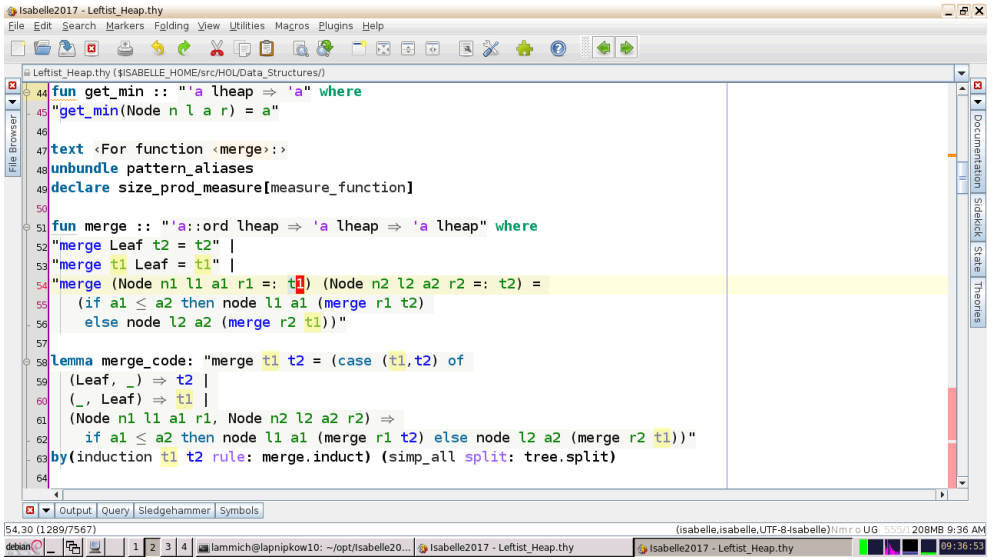
Corollary
$ltree\ t \Longrightarrow t\_insert\ x\ t \leq \log_2 |t|_1 + 2$

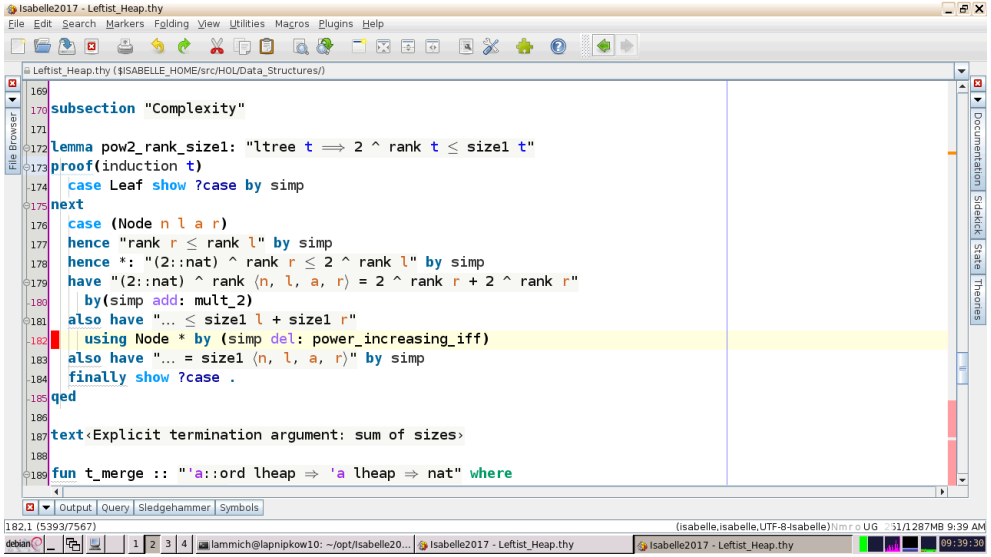


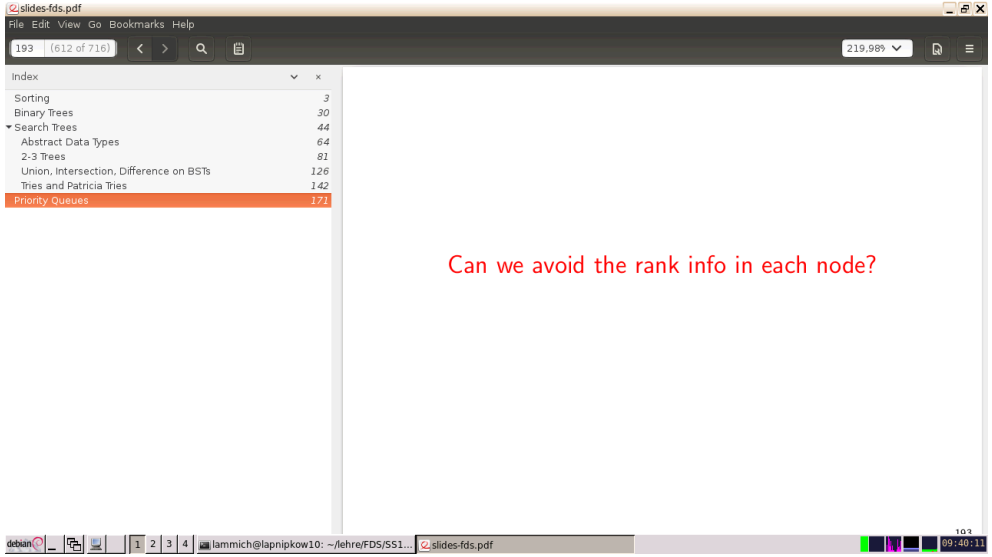

Isabelle2017 - Leftist_Heap.thy (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
Leftist_Heap.thy ($ISABELLE_HOME/src/HOL/Data_Structures/)
fun get_min :: "'a lheap ⇒ 'a" where
"get_min(Node n l a r) = a"
text ‹For function ‹merge›:›
unbundle pattern_aliases
(*declare size_prod_measure[measure_function]*)
fun merge :: "'a::ord lheap ⇒ 'a lheap ⇒ 'a lheap" where
a) ((n1, l1, a1, r1),
(n2, l2, a2, r2)) ~> (r1, ⟨n2, l2, a2, r2⟩)
b) ((n1, l1, a1, r1),
(n2, l2, a2, r2)) ~> (r2, ⟨n1, l1, a1, r1⟩)
Measures:
1) λp. size (snd p)
2) λp. size (fst p)
3) size
Result matrix:
1 2 3
a: <= < <=
b: ? ? <=
Could not find lexicographic termination order.
52,23 (1240/7571)
Input/output complete
(isabelle,isabelle,UTF-8-Isabelle)
09:34:23

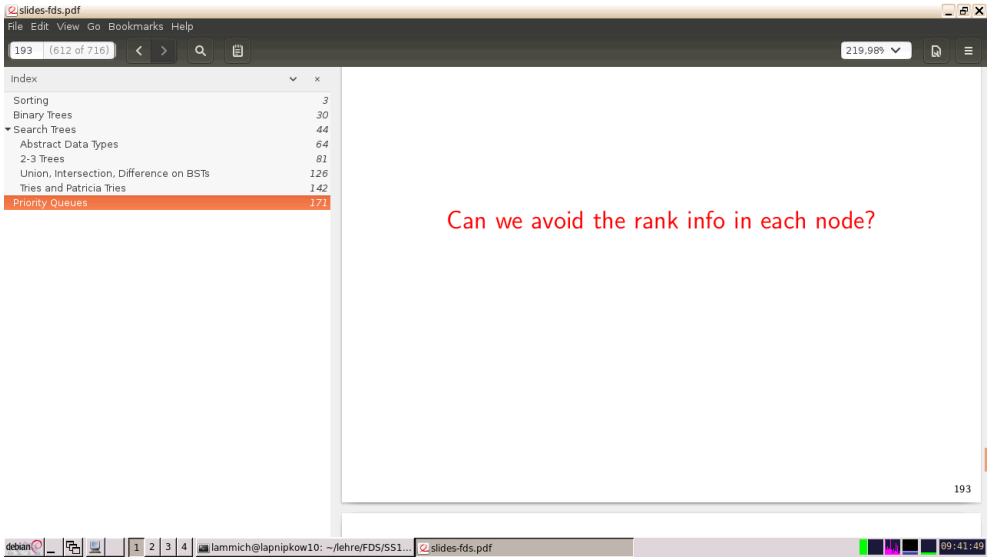

Isabelle2017 - Leftist_Heap.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Leftist_Heap.thy ($ISABELLE_HOME/src/HOL/Data_Structures/)
fun get_min :: "'a lheap ⇒ 'a" where
"get_min(Node n l a r) = a"
text ‹For function ‹merge›:›
unbundle pattern_aliases
declare size_prod_measure[measure_function]
fun merge :: "'a::ord lheap ⇒ 'a lheap ⇒ 'a lheap" where
"merge Leaf t2 = t2" |
"merge t1 Leaf = t1" |
"merge (Node n1 l1 a1 r1 =: t1) (Node n2 l2 a2 r2 =: t2) =
(if a1 ≤ a2 then node l1 a1 (merge r1 t2)
else node l2 a2 (merge r2 t1))"
lemma merge_code: "merge t1 t2 = (case (t1,t2) of
(Leaf, _) ⇒ t2 |
(_, Leaf) ⇒ t1 |
(Node n1 l1 a1 r1, Node n2 l2 a2 r2) ⇒
if a1 ≤ a2 then node l1 a1 (merge r1 t2) else node l2 a2 (merge r2 t1))"
by(induction t1 t2 rule: merge.induct) (simp_all split: tree.split)
Output Query Sledgehammer Symbols
49,9 (1120/7567)
null parsing complete, 0 error(s)
(isabelle,isabelle,UTF-8-Isabelle)
09:35:34


Isabelle2017 - Leftist_Heap.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Leftist_Heap.thy ($ISABELLE_HOME/src/HOL/Data_Structures/)
fun get_min :: "'a lheap ⇒ 'a" where
"get_min(Node n l a r) = a"
text ‹For function ‹merge›:›
unbundle pattern_aliases
declare size_prod_measure[measure_function]
fun merge :: "'a::ord lheap ⇒ 'a lheap ⇒ 'a lheap" where
"merge Leaf t2 = t2" |
"merge t1 Leaf = t1" |
"merge (Node n1 l1 a1 r1 =: t1) (Node n2 l2 a2 r2 =: t2) =
(if a1 ≤ a2 then node l1 a1 (merge r1 t2)
else node l2 a2 (merge r2 t1))"
lemma merge_code: "merge t1 t2 = (case (t1,t2) of
(Leaf, _) ⇒ t2 |
(_, Leaf) ⇒ t1 |
(Node n1 l1 a1 r1, Node n2 l2 a2 r2) ⇒
if a1 ≤ a2 then node l1 a1 (merge r1 t2) else node l2 a2 (merge r2 t1))"
by(induction t1 t2 rule: merge.induct) (simp_all split: tree.split)
Output Query Sledgehammer Symbols
52,1 (1214/7567)
(isabelle,isabelle,UTF-8-Isabelle)
09:35:50


Isabelle2017 - Leftist_Heap.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Leftist_Heap.thy ($ISABELLE_HOME/src/HOL/Data_Structures/)
fun get_min :: "'a lheap ⇒ 'a" where
"get_min(Node n l a r) = a"
text ‹For function ‹merge›:›
unbundle pattern_aliases
declare size_prod_measure[measure_function]
fun merge :: "'a::ord lheap ⇒ 'a lheap ⇒ 'a lheap" where
"merge Leaf t2 = t2" |
"merge t1 Leaf = t1" |
"merge (Node n1 l1 a1 r1 =: t1) (Node n2 l2 a2 r2 =: t2) =
(if a1 ≤ a2 then node l1 a1 (merge r1 t2)
else node l2 a2 (merge r2 t1))"
lemma merge_code: "merge t1 t2 = (case (t1,t2) of
(Leaf, _) ⇒ t2 |
(_, Leaf) ⇒ t1 |
(Node n1 l1 a1 r1, Node n2 l2 a2 r2) ⇒
if a1 ≤ a2 then node l1 a1 (merge r1 t2) else node l2 a2 (merge r2 t1))"
by(induction t1 t2 rule: merge.induct) (simp_all split: tree.split)
Output Query Sledgehammer Symbols
54,30 (1289/7567)
(isabelle,isabelle,UTF-8-Isabelle)
09:36:53







What is a Braun tree?

$$braun :: 'a\ tree \Rightarrow bool$$
$$braun\ \langle\rangle = True$$
$$braun\ \langle l, x, r\rangle =$$
$$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$$

1

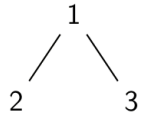
What is a Braun tree?

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

$braun\ \langle\rangle = True$

$braun\ \langle l, x, r\rangle =$

$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$



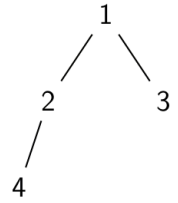
What is a Braun tree?

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

$braun\ \langle\rangle = True$

$braun\ \langle l, x, r\rangle =$

$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$



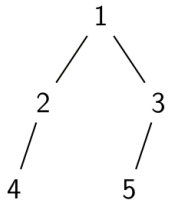
What is a Braun tree?

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

$braun\ \langle\rangle = True$

$braun\ \langle l, x, r\rangle =$

$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$



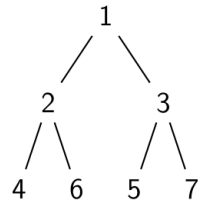
What is a Braun tree?

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

$braun\ \langle\rangle = True$

$braun\ \langle l, x, r\rangle =$

$(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)$



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

Idea of invariant maintenance

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

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
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: $|l| \leq |r| + 1 \wedge |r| + 1 \leq |l| + 1$ □

Remove 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
Goal: $|l| \leq |r| + 1 \wedge |r| + 1 \leq |l| + 1$ □

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

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: $|l| \leq |r| + 1 \wedge |r| + 1 \leq |l| + 1$ □

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

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: $|l| \leq |r| + 1 \wedge |r| + 1 \leq |l| + 1$ □

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

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.

del_min

$del\_min :: 'a\ tree \Rightarrow 'a\ tree$
$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle\langle\rangle, x, r\rangle = \langle\rangle$
$del\_min\ \langle l, x, r\rangle =$
(let $(y, l') = del\_left\ l$ in $sift\_down\ r\ y\ l'$)

del_min

$del\_min :: 'a\ tree \Rightarrow 'a\ tree$
$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle\langle\rangle, x, r\rangle = \langle\rangle$
$del\_min\ \langle l, x, r\rangle =$
(let $(y, l') = del\_left\ l$ in $sift\_down\ r\ y\ l'$)

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

sift_down

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

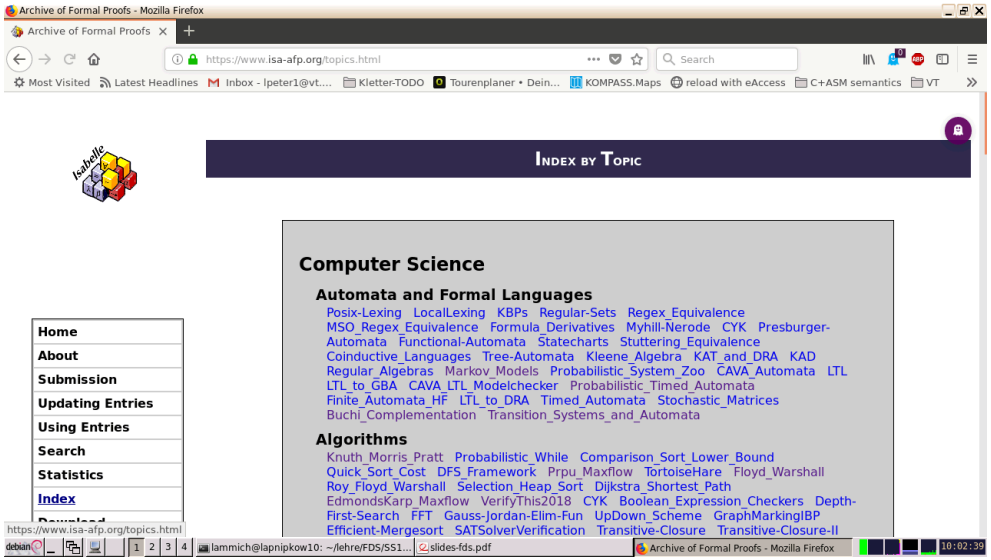
sift_down

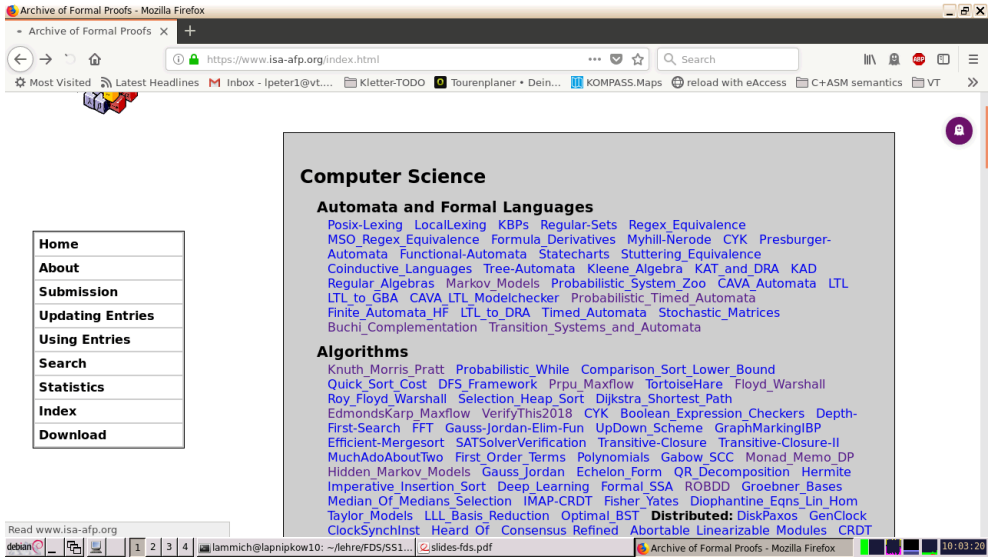
$sift\_down :: 'a\ tree \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$
$sift\_down\ \langle\rangle\ a\ \langle\rangle = \langle\langle\rangle, a, \langle\rangle\rangle$
$sift\_down\ \langle\langle\rangle, x, \langle\rangle\rangle\ a\ \langle\rangle =$
(if $a \leq x$ then $\langle\langle\langle\rangle, x, \langle\rangle\rangle, a, \langle\rangle\rangle$
else $\langle\langle\langle\rangle, a, \langle\rangle\rangle, x, \langle\rangle\rangle$)
$sift\_down\ (\langle l_1, x_1, r_1\rangle =: t_1)\ a\ (\langle l_2, x_2, r_2\rangle =: t_2) =$
if $a \leq x_1 \wedge a \leq x_2$ then $\langle t_1, a, t_2\rangle$
else if $x_1 \leq x_2$ then $\langle sift\_down\ l_1\ a\ r_1, x_1, t_2\rangle$
else $\langle t_1, x_2, sift\_down\ l_2\ a\ r_2\rangle$

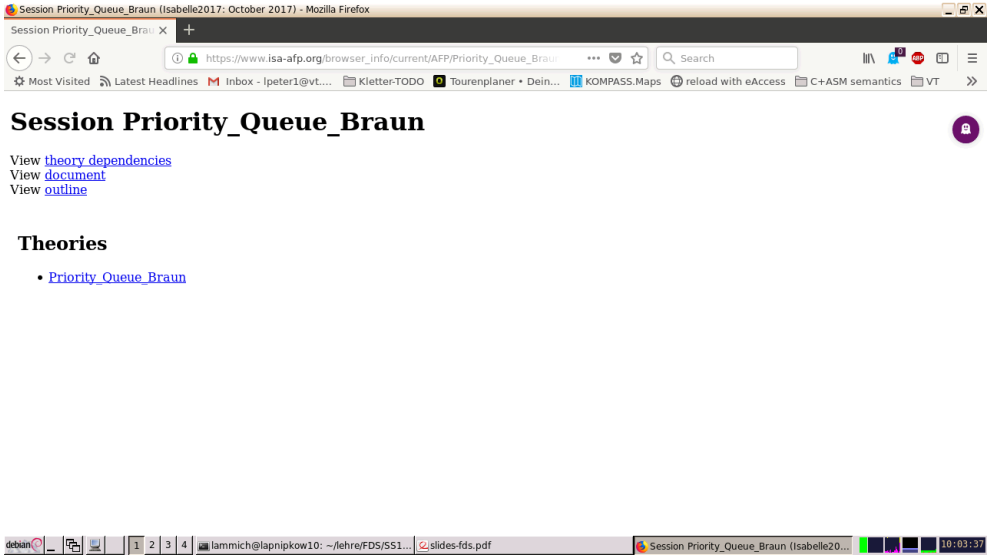
Maintains *braun*

Logarithmic complexity

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






Session Priority_Queue_Braun (Isabelle2017: October 2017) - Mozilla Firefox
Session Priority_Queue_Brau
https://www.isa-afp.org/browser_info/current/AFP/Priority_Queue_Braun
Search
Most Visited
Latest Headlines
Inbox - lpeter1@vt....
Kletter-TODO
Tourenplaner • Dein...
KOMPASS.Maps
reload with eAccess
C+ASM semantics
VT
Session Priority_Queue_Braun
View theory dependencies
View document
View outline
Theories
Priority_Queue_Braun
lammich@lapnipkow10: ~/lehre/FDS/SS1...
slides-fds.pdf
Session Priority_Queue_Braun (Isabelle20...
10:03:37