

# Binomial Heaps and Skew Binomial Heaps

Rene Meis      Finn Nielsen      Peter Lammich

September 13, 2023

## Abstract

We implement and prove correct binomial heaps and skew binomial heaps. Both are data-structures for priority queues. While binomial heaps have logarithmic *findMin*, *deleteMin*, *insert*, and *meld* operations, skew binomial heaps have constant time *findMin*, *insert*, and *meld* operations, and only the *deleteMin*-operation is logarithmic. This is achieved by using *skew links* to avoid cascading linking on *insert*-operations, and *data-structural bootstrapping* to get constant-time *findMin* and *meld* operations. Our implementation follows the paper of Brodal and Okasaki [1].

# Contents

<b>1</b>	<b>Binomial Heaps</b>	<b>3</b>
1.1	Datatype Definition	3
1.1.1	Abstraction to Multiset	3
1.1.2	Invariant	4
1.1.3	Heap Ordering	6
1.1.4	Height and Length	7
1.2	Operations	10
1.2.1	Empty	10
1.2.2	Insert	10
1.2.3	Meld	13
1.2.4	Find Minimal Element	16
1.2.5	Delete Minimal Element	18
1.3	Hiding the Invariant	23
1.3.1	Datatype	23
1.3.2	Operations	24
1.3.3	Correctness	25
1.4	Documentation	26
<b>2</b>	<b>Skew Binomial Heaps</b>	<b>27</b>
2.1	Datatype	28
2.1.1	Abstraction to Multisets	28
2.1.2	Invariant	28
2.1.3	Heap Order	33
2.1.4	Height and Length	34
2.2	Operations	39
2.2.1	Empty Tree	39
2.2.2	Insert	39
2.2.3	meld	42
2.2.4	Find Minimal Element	47
2.2.5	Delete Minimal Element	49
2.3	Bootstrapping	59
2.3.1	Auxiliary	59
2.3.2	Datatype	60
2.3.3	Specialization Boilerplate	61
2.3.4	Bootstrapping: Phase 1	67
2.3.5	Bootstrapping: Phase 2	71
2.4	Hiding the Invariant	73
2.4.1	Datatype	73
2.4.2	Operations	73
2.4.3	Correctness	75
2.5	Documentation	76

# 1 Binomial Heaps

```
theory BinomialHeap
imports Main HOL-Library.Multiset
begin
```

```
locale BinomialHeapStruc-loc
begin
```

## 1.1 Datatype Definition

Binomial heaps are lists of binomial trees.

```
datatype ('e, 'a) BinomialTree =
  Node (val: 'e) (prio: 'a::linorder) (rank: nat) (children: ('e, 'a) BinomialTree
  list)
type-synonym ('e, 'a) BinomialQueue-inv = ('e, 'a::linorder) BinomialTree list
```

Combine two binomial trees (of rank  $r$ ) to one (of rank  $r + 1$ ).

```
fun link :: ('e, 'a::linorder) BinomialTree  $\Rightarrow$  ('e, 'a) BinomialTree  $\Rightarrow$ 
  ('e, 'a) BinomialTree where
  link (Node e1 a1 r1 ts1) (Node e2 a2 r2 ts2) =
    (if a1  $\leq$  a2
     then (Node e1 a1 (Suc r1) ((Node e2 a2 r2 ts2)#ts1))
     else (Node e2 a2 (Suc r2) ((Node e1 a1 r1 ts1)#ts2)))
```

### 1.1.1 Abstraction to Multiset

Return a multiset with all (element, priority) pairs from a queue.

```
fun tree-to-multiset
  :: ('e, 'a::linorder) BinomialTree  $\Rightarrow$  ('e  $\times$  'a) multiset
and queue-to-multiset
  :: ('e, 'a::linorder) BinomialQueue-inv  $\Rightarrow$  ('e  $\times$  'a) multiset where
  tree-to-multiset (Node e a r ts) = {#(e,a)#} + queue-to-multiset ts |
  queue-to-multiset [] = {#} |
  queue-to-multiset (t#q) = tree-to-multiset t + queue-to-multiset q
```

```
lemma qtmset-append-union[simp]: queue-to-multiset (q @ q') =
  queue-to-multiset q + queue-to-multiset q'
apply(induct q)
apply(simp)
apply(simp add: union-ac)
done
```

```
lemma qtmset-rev[simp]: queue-to-multiset (rev q) = queue-to-multiset q
apply(induct q)
apply(simp)
apply(simp add: union-ac)
done
```

### 1.1.2 Invariant

We first formulate the invariant for single binomial trees, and then extend the invariant to binomial heaps (lists of binomial trees). The invariant for trees claims that a tree labeled rank 0 has no children, and a tree labeled rank  $r + 1$  is the result of a link operation of two rank  $r$  trees.

```

function tree-invar :: ('e, 'a::linorder) BinomialTree => bool where
  tree-invar (Node e a 0 ts) = (ts = []) |
  tree-invar (Node e a (Suc r) ts) =
    (∃ e1 a1 ts1 e2 a2 ts2.
      tree-invar (Node e1 a1 r ts1) ∧
      tree-invar (Node e2 a2 r ts2) ∧
      (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2))
by pat-completeness auto
termination
  apply(relation measure (λt. rank t))
  apply auto
done

```

A queue satisfies the invariant, iff all trees inside the queue satisfy the invariant, and the queue contains only trees of distinct rank and is ordered by rank

First part: All trees of the queue satisfy the tree invariant:

```

definition queue-invar :: ('e, 'a::linorder) BinomialQueue-inv => bool where
  queue-invar q ≡ (∀ t ∈ set q. tree-invar t)

```

Second part: Trees have distinct rank, and are ordered by ascending rank:

```

fun rank-invar :: ('e, 'a::linorder) BinomialQueue-inv => bool where
  rank-invar [] = True |
  rank-invar [t] = True |
  rank-invar (t # t' # bq) = (rank t < rank t' ∧ rank-invar (t' # bq))

```

```

lemma queue-invar-simps[simp]:
  queue-invar []
  queue-invar (t#q) <=> tree-invar t ∧ queue-invar q
  queue-invar (q@q') <=> queue-invar q ∧ queue-invar q'
unfolding queue-invar-def by auto

```

Invariant for binomial queues:

```

definition invar q == queue-invar q ∧ rank-invar q

```

```

lemma mset-link[simp]: (tree-to-multiset (link t1 t2))
  = (tree-to-multiset t1) + (tree-to-multiset t2)
by(cases t1, cases t2, auto simp add: union-ac)

```

```

lemma link-tree-invar:
  [tree-invar t1; tree-invar t2; rank t1 = rank t2] ==> tree-invar (link t1 t2)

```

```

by (cases t1, cases t2, simp, blast)

lemma invar-children:
  assumes tree-invar ((Node e a r ts)::('e, 'a::linorder) BinomialTree)
  shows queue-invar ts using assms
  unfolding queue-invar-def
proof(induct r arbitrary: e a ts)
  case 0
  then show ?case by simp
next
  case (Suc r)
  from Suc(2) obtain e1 a1 ts1 e2 a2 ts2 where
    O: tree-invar (Node e1 a1 r ts1) tree-invar (Node e2 a2 r ts2)
    (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2)
  by (simp only: tree-invar.simps) blast
  from Suc(1)[OF O(1)] O(2)
  have case1: queue-invar ((Node e2 a2 r ts2) # ts1)
    unfolding queue-invar-def by simp
  from Suc(1)[OF O(2)] O(1)
  have case2: queue-invar ((Node e1 a1 r ts1) # ts2)
    unfolding queue-invar-def by simp
  from O(3) have ts = (if a1 ≤ a2
    then (Node e2 a2 r ts2) # ts1
    else (Node e1 a1 r ts1) # ts2) by auto
  with case1 case2 show ?case unfolding queue-invar-def by simp
qed

lemma invar-children': tree-invar t ⇒ queue-invar (children t)
  by (cases t) (auto simp add: invar-children)

lemma rank-link: rank t = rank t' ⇒ rank (link t t') = rank t + 1
  apply (cases t)
  apply (cases t')
  apply (auto)
  done

lemma rank-invar-not-empty-hd: [[rank-invar (t # bq); bq ≠ []]] ⇒
  rank t < rank (hd bq)
  apply (induct bq arbitrary: t)
  apply (auto)
  done

lemma rank-invar-to-set: rank-invar (t # bq) ⇒
  ∀ t' ∈ set bq. rank t < rank t'
  apply (induct bq arbitrary: t)
  apply (simp)
  apply (metis nat-less-le rank-invar.simps(3) set-ConsD xt1(7))
  done

```

```

lemma set-to-rank-invar:  $\llbracket \forall t' \in \text{set } bq. \text{rank } t < \text{rank } t'; \text{rank-invar } bq \rrbracket$ 
   $\implies \text{rank-invar } (t \# bq)$ 
apply(induct bq arbitrary: t)
apply(simp)
by (metis list.sel(1) hd-in-set list.distinct(1) rank-invar.simps(3))

```

```

lemma rank-invar-hd-cons:
   $\llbracket \text{rank-invar } bq; \text{rank } t < \text{rank } (\text{hd } bq) \rrbracket \implies \text{rank-invar } (t \# bq)$ 
apply(cases bq)
apply(auto)
done

```

```

lemma rank-invar-cons:  $\text{rank-invar } (t \# bq) \implies \text{rank-invar } bq$ 
apply(cases bq)
apply(auto)
done

```

```

lemma invar-cons-up:
   $\llbracket \text{invar } (t \# bq); \text{rank } t' < \text{rank } t; \text{tree-invar } t' \rrbracket \implies \text{invar } (t' \# t \# bq)$ 
unfolding invar-def
by (cases bq) simp-all

```

```

lemma invar-cons-down:  $\text{invar } (t \# bq) \implies \text{invar } bq$ 
unfolding invar-def
by (cases bq) simp-all

```

```

lemma invar-app-single:
   $\llbracket \text{invar } bq; \forall t \in \text{set } bq. \text{rank } t < \text{rank } t'; \text{tree-invar } t' \rrbracket$ 
   $\implies \text{invar } (bq @ [t'])$ 
proof (induct bq)
  case Nil
  then show ?case by (simp add: invar-def)
next
  case (Cons a bq)
  from  $\langle \text{invar } (a \# bq) \rangle$  have invar bq by (rule invar-cons-down)
  with Cons have invar (bq @ [t']) by simp
  with Cons show ?case by (cases bq) (simp-all add: invar-def)
qed

```

### 1.1.3 Heap Ordering

```

fun heap-ordered :: ('e, 'a::linorder) BinomialTree  $\implies$  bool where
  heap-ordered (Node e a r ts) =  $(\forall x \in \text{set-mset}(\text{queue-to-multiset } ts). a \leq \text{snd } x)$ 

```

The invariant for trees implies heap order.

```

lemma tree-invar-heap-ordered:
  assumes tree-invar t

```

```

shows heap-ordered t
proof (cases t)
  case (Node e a nat list)
  with assms show ?thesis
  proof (induct nat arbitrary: t e a list)
    case 0
    then show ?case by simp
  next
  case (Suc nat t)
  then obtain t1 e1 a1 ts1 t2 e2 a2 ts2 where
    O: tree-invar t1 tree-invar t2 t = link t1 t2
    and t1[simp]: t1 = (Node e1 a1 nat ts1)
    and t2[simp]: t2 = (Node e2 a2 nat ts2)
    by (simp only: tree-invar.simps) blast
  from O(3) have t = (if a1 ≤ a2
    then (Node e1 a1 (Suc nat) (t2 # ts1))
    else (Node e2 a2 (Suc nat) (t1 # ts2))) by simp
  with Suc(1)[OF O(1) t1] Suc(1)[OF O(2) t2]
  show ?case by (cases a1 ≤ a2) auto
qed
qed

```

#### 1.1.4 Height and Length

Although complexity of HOL-functions cannot be expressed within HOL, we can express the height and length of a binomial heap. By showing that both, height and length, are logarithmic in the number of contained elements, we give strong evidence that our functions have logarithmic complexity in the number of elements.

Height of a tree and queue

```

fun height-tree :: ('e, ('a::linorder)) BinomialTree ⇒ nat and
  height-queue :: ('e, ('a::linorder)) BinomialQueue-inv ⇒ nat
where
  height-tree (Node e a r ts) = height-queue ts |
  height-queue [] = 0 |
  height-queue (t # ts) = max (Suc (height-tree t)) (height-queue ts)

```

```

lemma link-length: size (tree-to-multiset (link t1 t2)) =
  size (tree-to-multiset t1) + size (tree-to-multiset t2)
apply(cases t1)
apply(cases t2)
apply simp
done

```

```

lemma tree-rank-estimate:
  tree-invar (Node e a r ts) ⇒
  size (tree-to-multiset (Node e a r ts)) = (2::nat) ^ r
proof (induct r arbitrary: e a ts)

```

```

case 0
then show ?case by simp
next
  case (Suc r)
  from Suc(2) obtain e1 a1 ts1 e2 a2 ts2 where link:
    (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2)
  and inv1: tree-invar (Node e1 a1 r ts1)
  and inv2: tree-invar (Node e2 a2 r ts2) by simp blast
  from link-length[of (Node e1 a1 r ts1) (Node e2 a2 r ts2)]
  Suc(1)[OF inv1] Suc(1)[OF inv2] link
  show ?case by simp
qed

```

**lemma** tree-rank-height:

tree-invar (Node e a r ts)  $\implies$  height-tree (Node e a r ts) = r

**proof** (induct r arbitrary: e a ts)

```

case 0
then show ?case by simp
next
  case (Suc r)
  from Suc(2) obtain e1 a1 ts1 e2 a2 ts2 where link:
    (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2)
  and inv1: tree-invar (Node e1 a1 r ts1)
  and inv2: tree-invar (Node e2 a2 r ts2) by simp blast
  with link Suc(1)[OF inv1] Suc(1)[OF inv2] Suc(2) show ?case
  by (cases a1  $\leq$  a2) simp-all
qed

```

A binomial tree of height  $h$  contains exactly  $2^h$  elements

**theorem** tree-height-estimate:

tree-invar t  $\implies$  size (tree-to-multiset t) = (2::nat)<sup>height-tree t</sup>

**apply** (cases t, simp only:)

**apply** (frule tree-rank-estimate)

**apply** (frule tree-rank-height)

**apply** (simp only: )

**done**

**lemma** size-mset-tree: tree-invar t  $\implies$

size (tree-to-multiset t) = (2::nat)<sup>rank t</sup>

**by** (cases t) (simp only: tree-rank-estimate BinomialTree.sel(3))

**lemma** invar-butlast: invar (bq @ [t])  $\implies$  invar bq

**unfolding** invar-def

**apply** (induct bq) **apply** simp **apply** (case-tac bq)

**by** (simp-all)



**lemma** *invar-last-max*:  $\text{invar } (bq \text{ @ } [m]) \implies \forall t \in \text{set } bq. \text{rank } t < \text{rank } m$   
**unfolding** *invar-def*  
**apply** (*induct* *bq*) **apply** *simp* **apply** (*case-tac* *bq*) **apply** *simp* **by** *simp*

**lemma** *invar-length*:  $\text{invar } bq \implies \text{length } bq \leq \text{Suc } (\text{rank } (\text{last } bq))$

**proof** (*induct* *bq* *rule*: *rev-induct*)

**case** *Nil* **thus** *?case* **by** *simp*

**next**

**case** (*snoc* *x* *xs*)

**show** *?case* **proof** (*cases* *xs*)

**case** *Nil* **thus** *?thesis* **by** *simp*

**next**

**case** [*simp*]: (*Cons* *xxs* *xx*)

**from** *snoc.hyps*[*OF invar-butlast*[*OF snoc.prem*s]] **have**

*IH*:  $\text{length } xs \leq \text{Suc } (\text{rank } (\text{last } xs))$  .

**also from** *invar-last-max*[*OF snoc.prem*s] *last-in-set*[*of* *xs*] **have**

$\text{Suc } (\text{rank } (\text{last } xs)) \leq \text{rank } (\text{last } (xs \text{ @ } [x]))$

**by** *auto*

**finally show** *?thesis* **by** *simp*

**qed**

**qed**

**lemma** *size-queue-sum-list*:

$\text{size } (\text{queue-to-multiset } bq) = \text{sum-list } (\text{map } (\text{size} \circ \text{tree-to-multiset}) \text{ } bq)$

**by** (*induct* *bq*) *simp-all*

A binomial heap of length  $l$  contains at least  $2^l - 1$  elements.

**theorem** *queue-length-estimate-lower*:

$\text{invar } bq \implies (\text{size } (\text{queue-to-multiset } bq)) \geq 2^{\text{length } bq} - 1$

**proof** (*induct* *bq* *rule*: *rev-induct*)

**case** *Nil* **thus** *?case* **by** *simp*

**next**

**case** (*snoc* *x* *xs*)

**from** *snoc.hyps*[*OF invar-butlast*[*OF snoc.prem*s]]

**have** *IH*:  $2^{\text{length } xs} \leq \text{Suc } (\text{size } (\text{queue-to-multiset } xs))$  **by** *simp*

**have** *size-q*:

$\text{size } (\text{queue-to-multiset } (xs \text{ @ } [x])) =$

$\text{size } (\text{queue-to-multiset } xs) + \text{size } (\text{tree-to-multiset } x)$

**by** (*simp* *add*: *size-queue-sum-list*)

**also**

**from** *snoc.prem*s **have** *inv-x*: *tree-invar* *x* **by** (*simp* *add*: *invar-def*)

**hence**  $\text{size } (\text{tree-to-multiset } x) = 2^{\text{rank } x}$  **by** (*simp* *add*: *size-mset-tree*)

**finally have**

*eq*:  $\text{size } (\text{queue-to-multiset } (xs \text{ @ } [x])) =$

$\text{size } (\text{queue-to-multiset } xs) + (2::\text{nat})^{\text{rank } x}$  .

**from** *invar-length*[*OF snoc.prem*s] **have**  $\text{length } xs \leq \text{rank } x$  **by** *simp*

**hence** *snd*:  $(2::\text{nat})^{\text{length } xs} \leq (2::\text{nat})^{\text{rank } x}$  **by** *simp*

**have**

$(2::\text{nat})^{\text{length } (xs \text{ @ } [x])} = (2::\text{nat})^{\text{length } xs} + (2::\text{nat})^{\text{length } xs}$

```

  by simp
with IH have
   $2^{\text{length } (xs @ [x])} \leq \text{Suc } (\text{size } (\text{queue-to-multiset } xs)) + 2^{\text{length } xs}$ 
  by simp
with snd have  $2^{\text{length } (xs @ [x])} \leq$ 
   $\text{Suc } (\text{size } (\text{queue-to-multiset } xs)) + 2^{\text{rank } x}$ 
  by arith
with eq show ?case by simp
qed

```

## 1.2 Operations

### 1.2.1 Empty

```

lemma empty-correct[simp]:
  invar Nil
  queue-to-multiset Nil = {#}
  by (simp-all add: invar-def)

```

The empty multiset is represented by exactly the empty queue

```

lemma empty-iff:  $t = \text{Nil} \longleftrightarrow \text{queue-to-multiset } t = \{\#\}$ 
  apply (cases t)
  apply auto
  apply (case-tac a)
  apply auto
  done

```

### 1.2.2 Insert

Inserts a binomial tree into a binomial queue, such that the queue does not contain two trees of same rank.

```

fun ins :: ('e, 'a::linorder) BinomialTree  $\Rightarrow$  ('e, 'a) BinomialQueue-inv  $\Rightarrow$ 
  ('e, 'a) BinomialQueue-inv where
  ins t [] = [t] |
  ins t' (t # bq) = (if (rank t') < (rank t)
    then t' # t # bq
    else (if (rank t) < (rank t')
      then t # (ins t' bq)
      else ins (link t' t) bq))

```

Inserts an element with priority into the queue.

```

definition insert :: 'e  $\Rightarrow$  'a::linorder  $\Rightarrow$  ('e, 'a) BinomialQueue-inv  $\Rightarrow$ 
  ('e, 'a) BinomialQueue-inv where
  insert e a bq = ins (Node e a 0 []) bq

```

```

lemma ins-mset:
   $\llbracket \text{tree-invar } t; \text{queue-invar } q \rrbracket \Longrightarrow \text{queue-to-multiset } (\text{ins } t \ q)$ 
  =  $\text{tree-to-multiset } t + \text{queue-to-multiset } q$ 
by (induct q arbitrary: t) (auto simp: union-ac link-tree-invar)

```

**lemma** *insert-mset*: *queue-invar*  $q \implies$   
*queue-to-multiset* (*insert*  $e$   $a$   $q$ ) = *queue-to-multiset*  $q + \{\# (e, a) \#\}$   
**by**(*simp add: ins-mset union-ac insert-def*)

**lemma** *ins-queue-invar*:  $\llbracket \text{tree-invar } t; \text{queue-invar } q \rrbracket \implies \text{queue-invar } (\text{ins } t \ q)$   
**proof** (*induct*  $q$  *arbitrary: t*)  
**case** (*Cons*  $a$   $q$ )  
**note**  $iv = \text{Cons.hyps}$   
**show** *?case*  
**proof** (*cases*  $\text{rank } t = \text{rank } a$ )  
**case** [*simp*]: *True*  
**from** *Cons.prem*s **have**  
*inv-a*: *tree-invar*  $a$  **and** *inv-q*: *queue-invar*  $q$   
**by** (*simp-all*)  
**note**  $\text{inv-link} = \text{link-tree-invar}[OF \langle \text{tree-invar } t \rangle \text{inv-a } \text{True}]$   
**from**  $iv[OF \text{inv-link } \text{inv-q}]$  **show** *?thesis* **by** *simp*  
**next**  
**case** *False*  
**with** *Cons* **show** *?thesis* **by** *auto*  
**qed**  
**qed** *simp*

**lemma** *insert-queue-invar*:  
**assumes** *queue-invar*  $q$   
**shows** *queue-invar* (*insert*  $e$   $a$   $q$ )  
**proof** –  
**have**  $\text{inv}: \text{tree-invar } (\text{Node } e \ a \ 0 \ [])$  **by** *simp*  
**from** *ins-queue-invar*[*OF inv assms*] **show** *?thesis* **by** (*simp add: insert-def*)  
**qed**

**lemma** *rank-ins*: (*rank-invar* ( $t \ \# \ bq$ )  $\implies$   
 $(\text{rank } (\text{hd } (\text{ins } t' \ (t \ \# \ bq)))) \geq \text{rank } t) \vee$   
 $(\text{rank } (\text{hd } (\text{ins } t' \ (t \ \# \ bq)))) \geq \text{rank } t')$ )  
**apply**(*auto*)  
**apply**(*induct*  $bq$  *arbitrary: t t'*)  
**apply**(*simp add: rank-link*)  
**proof** *goal-cases*  
**case** *prems*: ( $1 \ a \ bq \ t \ t'$ )  
**thus** *?case*  
**apply**(*cases*  $\text{rank } (\text{link } t' \ t) = \text{rank } a$ )  
**apply**(*auto simp add: rank-link*)  
**proof** *goal-cases*  
**case**  $1$   
**note**  $*$  = *this* **and**  $\langle \wedge t' \ t. \llbracket \text{rank-invar } (t \ \# \ bq); \text{rank } t' = \text{rank } t \rrbracket$   
 $\implies \text{rank } t \leq \text{rank } (\text{hd } (\text{ins } (\text{link } t' \ t) \ bq)) \rangle$ [*of a (link t' t)*]  
**show** *?case*  
**proof** (*cases*  $\text{rank } (\text{hd } (\text{ins } (\text{link } (\text{link } t' \ t) \ a) \ bq)) = \text{rank } a$ )  
**case** *True*

```

    with * show ?thesis by simp
  next
    case False
    with * have rank a ≤ rank (hd (ins (link (link t' t) a) bq))
      by (simp add: rank-link)
    with * show ?thesis by simp
  qed
qed
qed

lemma rank-ins2: rank-invar bq ⇒
  rank t ≤ rank (hd (ins t bq)) ∨
  (rank (hd (ins t bq)) = rank (hd bq) ∧ bq ≠ [])
  apply (induct bq arbitrary: t)
  apply (auto)
proof goal-cases
  case prems: (1 a bq t)
  hence r: rank (link t a) = rank a + 1 by (simp add: rank-link)
  from prems r and prems(1)[of (link t a)] show ?case by (cases bq) auto
qed

lemma rank-invar-ins: rank-invar bq ⇒ rank-invar (ins t bq)
  apply (induct bq arbitrary: t)
  apply (simp)
  apply (auto)
proof goal-cases
  case prems: (1 a bq t)
  hence inv: rank-invar (ins t bq) by (cases bq) simp-all
  from prems have hd: bq ≠ [] ⇒ rank a < rank (hd bq)
    by (cases bq) auto
  from prems have rank t ≤ rank (hd (ins t bq)) ∨
    (rank (hd (ins t bq)) = rank (hd bq) ∧ bq ≠ [])
    by (simp add: rank-ins2 rank-invar-cons)
  with prems have rank a < rank (hd (ins t bq)) ∨
    (rank (hd (ins t bq)) = rank (hd bq) ∧ bq ≠ []) by auto
  with prems and inv and hd show ?case by (auto simp add: rank-invar-hd-cons)
next
  case prems: (2 a bq t)
  hence inv: rank-invar bq by (cases bq) simp-all
  with prems and prems(1)[of (link t a)] show ?case by simp
qed

lemma rank-invar-insert: rank-invar bq ⇒ rank-invar (insert e a bq)
  by (simp add: rank-invar-ins insert-def)

lemma insert-correct:
  assumes I: invar q
  shows
  invar (insert e a q)

```

```

queue-to-multiset (insert e a q) = queue-to-multiset q + {# (e,a) #}
using insert-queue-invar[of q] rank-invar-insert[of q] insert-mset[of q] I
unfolding invar-def by auto

```

### 1.2.3 Meld

Melds two queues.

```

fun meld :: ('e, 'a::linorder) BinomialQueue-inv ⇒ ('e, 'a) BinomialQueue-inv
⇒ ('e, 'a) BinomialQueue-inv
where
meld [] bq = bq |
meld bq [] = bq |
meld (t1#bq1) (t2#bq2) =
  (if (rank t1) < (rank t2)
    then t1 # (meld bq1 (t2 # bq2))
    else (
      if (rank t2 < rank t1)
        then t2 # (meld (t1 # bq1) bq2)
        else ins (link t1 t2) (meld bq1 bq2)
    )
  )
)

```

**lemma** meld-queue-invar:

$\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies \text{queue-invar } (\text{meld } q \ q')$

**proof** (induct q q' rule: meld.induct)

**case** 1

**then show** ?case **by** simp

**next**

**case** 2

**then show** ?case **by** simp

**next**

**case** (3 t1 bq1 t2 bq2)

**consider** (lt) rank t1 < rank t2 | (gt) rank t1 > rank t2 | (eq) rank t1 = rank t2

**by** atomize-elim auto

**then show** ?case

**proof** cases

**case** lt

**from** 3(4) **have** inv-bq1: queue-invar bq1 **by** simp

**from** 3(4) **have** inv-t1: tree-invar t1 **by** simp

**from** 3(1)[OF lt inv-bq1 3(5)] inv-t1 lt

**show** ?thesis **by** simp

**next**

**case** gt

**from** 3(5) **have** inv-bq2: queue-invar bq2 **by** simp

**from** 3(5) **have** inv-t2: tree-invar t2 **by** simp

**from** gt **have** ¬ rank t1 < rank t2 **by** simp

**from** 3(2)[OF this gt 3(4) inv-bq2] inv-t2 gt

**show** ?thesis **by** simp

```

next
  case eq
  from 3(4) have inv-bq1: queue-invar bq1 by simp
  from 3(4) have inv-t1: tree-invar t1 by simp
  from 3(5) have inv-bq2: queue-invar bq2 by simp
  from 3(5) have inv-t2: tree-invar t2 by simp
  note inv-link = link-tree-invar[OF inv-t1 inv-t2 eq]
  from eq have *:  $\neg \text{rank } t1 < \text{rank } t2 \neg \text{rank } t2 < \text{rank } t1$  by simp-all
  note inv-meld = 3(3)[OF * inv-bq1 inv-bq2]
  from ins-queue-invar[OF inv-link inv-meld] *
  show ?thesis by simp
qed
qed

lemma rank-ins-min: rank-invar bq  $\implies$ 
  rank (hd (ins t bq))  $\geq$  min (rank t) (rank (hd bq))
  apply(induct bq arbitrary: t)
  apply(auto)
proof goal-cases
  case prems: (1 a bq t)
  hence inv: rank-invar bq by (cases bq) simp-all
  from prems have r: rank (link t a) = rank a + 1 by (simp add: rank-link)
  with prems and inv and prems(1)[of (link t a)] show ?case by (cases bq) auto
qed

lemma rank-invar-meld-strong:
  [rank-invar bq1; rank-invar bq2]  $\implies$  rank-invar (meld bq1 bq2)  $\wedge$ 
  rank (hd (meld bq1 bq2))  $\geq$  min (rank (hd bq1)) (rank (hd bq2))
proof (induct bq1 bq2 rule: meld.induct)
  case 1
  then show ?case by simp
next
  case 2
  then show ?case by simp
next
  case (3 t1 bq1 t2 bq2)
  from 3 have inv1: rank-invar bq1 by (cases bq1) simp-all
  from 3 have inv2: rank-invar bq2 by (cases bq2) simp-all

  from inv1 and inv2 and 3 show ?case
proof (auto, goal-cases)
  let ?t = t2
  let ?bq = bq2
  let ?meld = rank t2 < rank (hd (meld (t1 # bq1) bq2))
  case prems: 1
  hence ?bq  $\neq$  []  $\implies$  rank ?t < rank (hd ?bq)
  by (simp add: rank-invar-not-empty-hd)
  with prems have ne: ?bq  $\neq$  []  $\implies$  ?meld by simp
  from prems have ?bq = []  $\implies$  ?meld by simp

```

```

with ne have ?meld by (cases ?bq = [])
with prems show ?case by (simp add: rank-invar-hd-cons)
next — analog
let ?t = t1
let ?bq = bq1
let ?meld = rank t1 < rank (hd (meld bq1 (t2 # bq2)))
case prems: 2
hence ?bq ≠ [] ⇒ rank ?t < rank (hd ?bq)
  by (simp add: rank-invar-not-empty-hd)
with prems have ne: ?bq ≠ [] ⇒ ?meld by simp
from prems have ?bq = [] ⇒ ?meld by simp
with ne have ?meld by (cases ?bq = [])
with prems show ?case by (simp add: rank-invar-hd-cons)
next
case 3
thus ?case by (simp add: rank-invar-ins)
next
case prems: 4
then have r: rank (link t1 t2) = rank t2 + 1
  by (simp add: rank-link)
have m: meld bq1 [] = bq1 by (cases bq1, auto)

from inv1 and inv2 and prems
have mm: min (rank (hd bq1)) (rank (hd bq2)) ≤ rank (hd (meld bq1 bq2))
  by simp
from ⟨rank-invar (t1 # bq1)⟩ have bq1 ≠ [] ⇒ rank t1 < rank (hd bq1)
  by (simp add: rank-invar-not-empty-hd)
with prems have r1: bq1 ≠ [] ⇒ rank t2 < rank (hd bq1) by simp
from ⟨rank-invar (t2 # bq2)⟩
have r2: bq2 ≠ [] ⇒ rank t2 < rank (hd bq2)
  by (simp add: rank-invar-not-empty-hd)

from inv1 r r1 rank-ins-min[of bq1 (link t1 t2)]
have abc1: bq1 ≠ [] ⇒ rank t2 ≤ rank (hd (ins (link t1 t2) bq1))
  by simp
from inv2 r r2 rank-ins-min[of bq2 (link t1 t2)]
have abc2: bq2 ≠ [] ⇒ rank t2 ≤ rank (hd (ins (link t1 t2) bq2))
  by simp
from r1 r2 mm have
  [[bq1 ≠ []; bq2 ≠ []] ⇒ rank t2 < rank (hd (meld bq1 bq2))] by simp
with ⟨rank-invar (meld bq1 bq2)⟩
  r rank-ins-min[of meld bq1 bq2 link t1 t2]
have [[bq1 ≠ []; bq2 ≠ []] ⇒
  rank t2 < rank (hd (ins (link t1 t2) (meld bq1 bq2))] by simp
thm rank-ins-min[of meld bq1 bq2 link t1 t2]
with inv1 and inv2 and r m r1 show ?case
  apply(cases bq2 = [])
  apply(cases bq1 = [])
  apply(simp)

```

```

    apply(auto simp add: abc1)
    apply(cases bq1 = [])
    apply(simp)
    apply(auto simp add: abc2)
  done
qed
qed

```

**lemma** *rank-invar-meld*:  
 $\llbracket \text{rank-invar } bq1; \text{rank-invar } bq2 \rrbracket \implies \text{rank-invar } (\text{meld } bq1 \text{ } bq2)$   
**by** (*simp only: rank-invar-meld-strong*)

**lemma** *meld-mset*:  $\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies$   
 $\text{queue-to-multiset } (\text{meld } q \text{ } q') =$   
 $\text{queue-to-multiset } q + \text{queue-to-multiset } q'$   
**by**(*induct q q' rule: meld.induct*)  
(*auto simp add: link-tree-invar meld-queue-invar ins-mset union-ac*)

**lemma** *meld-correct*:  
**assumes** *invar q invar q'*  
**shows**  
*invar (meld q q')*  
 $\text{queue-to-multiset } (\text{meld } q \text{ } q') = \text{queue-to-multiset } q + \text{queue-to-multiset } q'$   
**using** *assms*  
**unfolding** *invar-def*  
**by** (*simp-all add: meld-queue-invar rank-invar-meld meld-mset*)

#### 1.2.4 Find Minimal Element

Finds the tree containing the minimal element.

**fun** *getMinTree* :: (*'e, 'a::linorder*) *BinomialQueue-inv*  $\Rightarrow$   
(*'e, 'a*) *BinomialTree* **where**  
*getMinTree* [t] = t |  
*getMinTree* (t#bq) = (if *prio t*  $\leq$  *prio (getMinTree bq)*  
then t else (*getMinTree bq*))

**lemma** *mintree-exists*:  $(bq \neq []) = (\text{getMinTree } bq \in \text{set } bq)$

**proof** (*induct bq*)  
**case** *Nil*  
**then show** *?case* **by** *simp*  
**next**  
**case** (*Cons - bq*)  
**then show** *?case* **by** (*cases bq simp-all*)  
**qed**

**lemma** *treehead-in-multiset*:  
 $t \in \text{set } bq \implies (\text{val } t, \text{prio } t) \in \# \text{queue-to-multiset } bq$   
**by** (*induct bq, simp, cases t, auto*)



**lemma** *heap-ordered-single*:  
*heap-ordered*  $t = (\forall x \in \text{set-mset } (\text{tree-to-multiset } t). \text{prio } t \leq \text{snd } x)$   
**by** (*cases*  $t$ ) *auto*

**lemma** *getMinTree-cons*:  
 $\text{prio } (\text{getMinTree } (y \# x \# xs)) \leq \text{prio } (\text{getMinTree } (x \# xs))$   
**by** (*induct*  $xs$  *rule: getMinTree.induct*) *simp-all*

**lemma** *getMinTree-min-tree*:  
 $t \in \text{set } bq \implies \text{prio } (\text{getMinTree } bq) \leq \text{prio } t$   
**apply** (*induct*  $bq$  *arbitrary: t rule: getMinTree.induct*)  
**apply** *simp*  
**defer**  
**apply** *simp*  
**proof** *goal-cases*  
**case** *prems*: ( $1\ t\ v\ va\ ta$ )  
**thus** *?case*  
**apply** (*cases*  $ta = t$ )  
**apply** *auto*[1]  
**apply** (*metis* *getMinTree-cons* *prems*(1) *prems*(3) *set-ConsD* *xt1*(6))  
**done**  
**qed**

**lemma** *getMinTree-min-prio*:  
**assumes** *queue-invar*  $bq$   
**and**  $y \in \text{set-mset } (\text{queue-to-multiset } bq)$   
**shows**  $\text{prio } (\text{getMinTree } bq) \leq \text{snd } y$   
**proof** –  
**from** *assms* **have**  $bq \neq []$  **by** (*cases*  $bq$ ) *simp-all*  
**with** *assms* **have**  $\exists t \in \text{set } bq. (y \in \text{set-mset } ((\text{tree-to-multiset } t)))$   
**proof** (*induct*  $bq$ )  
**case** *Nil*  
**then show** *?case* **by** *simp*  
**next**  
**case** (*Cons*  $a\ bq$ )  
**thus** *?case*  
**apply**(*cases*  $y \in \text{set-mset } (\text{tree-to-multiset } a)$ )  
**apply** *simp*  
**apply**(*cases*  $bq$ )  
**apply** *simp-all*  
**done**  
**qed**  
**from** *this* **obtain**  $t$  **where**  $O$ :  
 $t \in \text{set } bq$   
 $y \in \text{set-mset } (\text{tree-to-multiset } t)$  **by** *blast*  
**obtain**  $e\ a\ r\ ts$  **where** [*simp*]:  $t = (\text{Node } e\ a\ r\ ts)$  **by** (*cases*  $t$ ) *blast*  
**from**  $O$  *assms*(1) **have** *inv*: *tree-invar*  $t$  **by** (*simp* *add: queue-invar-def*)  
**from** *tree-invar-heap-ordered*[*OF inv*] *heap-ordered.simps*[*of e a r ts*]  $O$   
**have**  $\text{prio } t \leq \text{snd } y$  **by** *auto*

**with** *getMinTree-min-tree*[*OF O(1)*] **show** *?thesis* **by** *simp*  
**qed**

Finds the minimal Element in the queue.

**definition** *findMin* :: (*'e*, *'a::linorder*) *BinomialQueue-inv*  $\Rightarrow$  (*'e*  $\times$  *'a*) **where**  
*findMin* *bq* = (*let* *min* = *getMinTree* *bq* *in* (*val* *min*, *prio* *min*))

**lemma** *findMin-correct*:

**assumes** *I*: *invar* *q*

**assumes** *NE*: *q*  $\neq$  *Nil*

**shows**

*findMin* *q*  $\in$  # *queue-to-multiset* *q*

$\forall y \in$  *set-mset* (*queue-to-multiset* *q*). *snd* (*findMin* *q*)  $\leq$  *snd* *y*

**proof** –

**from** *NE* **have** *getMinTree* *q*  $\in$  *set* *q* **by** (*simp* *only*: *mintree-exists*)

**thus** *findMin* *q*  $\in$  # *queue-to-multiset* *q*

**by** (*simp* *add*: *treehead-in-multiset* *Let-def* *findMin-def*)

**show**  $\forall y \in$  *set-mset* (*queue-to-multiset* *q*). *snd* (*findMin* *q*)  $\leq$  *snd* *y*

**using** *I*[*unfolded* *invar-def*]

**by** (*auto* *simp* *add*: *getMinTree-min-prio* *Let-def* *findMin-def*)

**qed**

### 1.2.5 Delete Minimal Element

Removes the first tree, which has the priority *a* within his root.

**fun** *remove1Prio* :: *'a*  $\Rightarrow$  (*'e*, *'a::linorder*) *BinomialQueue-inv*  $\Rightarrow$   
(*'e*, *'a*) *BinomialQueue-inv* **where**  
*remove1Prio* *a* [] = [] |  
*remove1Prio* *a* (*t* # *bq*) =  
(*if* (*prio* *t*) = *a* *then* *bq* *else* *t* # (*remove1Prio* *a* *bq*))

Returns the queue without the minimal element.

**definition** *deleteMin* :: (*'e*, *'a::linorder*) *BinomialQueue-inv*  $\Rightarrow$   
(*'e*, *'a*) *BinomialQueue-inv* **where**  
*deleteMin* *bq*  $\equiv$  (*let* *min* = *getMinTree* *bq* *in*  
*meld* (*rev* (*children* *min*))  
(*remove1Prio* (*prio* *min*) *bq*))

**lemma** *queue-invar-rev*: *queue-invar* *q*  $\Longrightarrow$  *queue-invar* (*rev* *q*)  
**by** (*simp* *add*: *queue-invar-def*)

**lemma** *queue-invar-remove1*: *queue-invar* *q*  $\Longrightarrow$  *queue-invar* (*remove1* *t* *q*)  
**by** (*auto* *simp* *add*: *queue-invar-def*)

**lemma** *qtm-in-set-subset*: *t*  $\in$  *set* *q*  $\Longrightarrow$   
*tree-to-multiset* *t*  $\subseteq$  # *queue-to-multiset* *q*

**proof**(*induct* *q*)

**case** *Nil*

```

then show ?case by simp
next
case (Cons a q)
show ?case
proof (cases t = a)
  case True
  then show ?thesis by simp
next
case False
with Cons have t-in-q: t ∈ set q by simp
have queue-to-multiset q ⊆# queue-to-multiset (a # q)
  by simp
from subset-mset.order-trans[OF Cons(1)[OF t-in-q] this] show ?thesis .
qed
qed

```

```

lemma remove1-mset: t ∈ set q ⇒
  queue-to-multiset (remove1 t q) =
  queue-to-multiset q - tree-to-multiset t
by (induct q) (auto simp: qtm-in-set-subset)

```

```

lemma remove1Prio-remove1[simp]:
  remove1Prio (prio (getMinTree bq)) bq = remove1 (getMinTree bq) bq
proof (induct bq)
  case Nil thus ?case by simp
next
case (Cons t bq)
note iv = Cons
thus ?case
proof (cases t = getMinTree (t # bq))
  case True
  with iv show ?thesis by simp
next
case False
hence ne: bq ≠ [] by auto
with False have down: getMinTree (t # bq) = getMinTree bq
  by (induct bq rule: getMinTree.induct) auto
from ne False have prio t ≠ prio (getMinTree bq)
  by (induct bq rule: getMinTree.induct) auto
with down iv False ne show ?thesis by simp
qed
qed

```

```

lemma deleteMin-queue-invar:
  assumes INV: queue-invar q
  assumes NE: q ≠ Nil
  shows queue-invar (deleteMin q)
proof (cases q)
  case Nil

```

```

with assms show ?thesis by simp
next
  case Cons
  from NE and mintree-exists[of q] INV
  have inv-min: tree-invar (getMinTree q) by (simp add: queue-invar-def)
  note inv-children = invar-children'[OF inv-min]
  note inv-rev = queue-invar-rev[OF inv-children]
  note inv-rem = queue-invar-remove1[OF INV, of getMinTree q]
  from meld-queue-invar[OF inv-rev inv-rem] show ?thesis
    by (simp add: deleteMin-def Let-def)
qed

lemma children-rank-less:
  assumes tree-invar t
  shows  $\forall t' \in \text{set } (\text{children } t). \text{rank } t' < \text{rank } t$ 
proof (cases t)
  case (Node e a nat list)
  with assms show ?thesis
  proof (induct nat arbitrary: t e a list)
    case 0
    then show ?case by simp
  next
  case (Suc nat)
  then obtain e1 a1 ts1 e2 a2 ts2 where
    O: tree-invar (Node e1 a1 nat ts1) tree-invar (Node e2 a2 nat ts2)
    t = link (Node e1 a1 nat ts1) (Node e2 a2 nat ts2)
    by (simp only: tree-invar.simps) blast
  hence ch-id: children t =
    (if a1 ≤ a2 then (Node e2 a2 nat ts2)#ts1
     else (Node e1 a1 nat ts1)#ts2) by simp
  from O Suc(1)[of Node e1 a1 nat ts1 e1 a1 ts1]
  have p1:  $\forall t' \in \text{set } ((\text{Node } e2 \ a2 \ \text{nat } ts2) \# ts1). \text{rank } t' < \text{Suc } \text{nat}$  by auto
  from O Suc(1)[of Node e2 a2 nat ts2 e2 a2 ts2]
  have p2:  $\forall t' \in \text{set } ((\text{Node } e1 \ a1 \ \text{nat } ts1) \# ts2). \text{rank } t' < \text{Suc } \text{nat}$  by auto
  from Suc(3) p1 p2 ch-id show ?case by simp
qed
qed

lemma strong-rev-children:
  assumes tree-invar t
  shows invar (rev (children t))
  unfolding invar-def
proof (cases t)
  case (Node e a nat list)
  with assms show queue-invar (rev (children t))  $\wedge$  rank-invar (rev (children t))
  proof (induct nat arbitrary: t e a list)
    case 0
    then show ?case by simp
  next

```

```

case (Suc nat)
then obtain e1 a1 ts1 e2 a2 ts2 where
  O: tree-invar (Node e1 a1 nat ts1) tree-invar (Node e2 a2 nat ts2)
  t = link (Node e1 a1 nat ts1) (Node e2 a2 nat ts2)
by (simp only: tree-invar.simps) blast
hence ch-id: children t =
  (if a1 ≤ a2 then (Node e2 a2 nat ts2)#ts1
   else (Node e1 a1 nat ts1)#ts2) by simp
from O Suc(1)[of Node e1 a1 nat ts1 e1 a1 ts1]
have rev-ts1: invar (rev ts1) by (simp add: invar-def)
from O children-rank-less[of Node e1 a1 nat ts1]
have ∀ t∈set (rev ts1). rank t < rank (Node e2 a2 nat ts2) by simp
with O rev-ts1 invar-app-single[of rev ts1 Node e2 a2 nat ts2]
have p1: invar (rev ((Node e2 a2 nat ts2) # ts1)) by simp
from O Suc(1)[of Node e2 a2 nat ts2 e2 a2 ts2]
have rev-ts2: invar (rev ts2) by (simp add: invar-def)
from O children-rank-less[of Node e2 a2 nat ts2]
have ∀ t∈set (rev ts2). rank t < rank (Node e1 a1 nat ts1) by simp
with O rev-ts2 invar-app-single[of rev ts2 Node e1 a1 nat ts1]
have p2: invar (rev ((Node e1 a1 nat ts1) # ts2)) by simp
from p1 p2 ch-id show ?case by (simp add: invar-def)
qed
qed

```

```

lemma first-less: rank-invar (t # bq) ⇒ ∀ t' ∈ set bq. rank t < rank t'
apply (induct bq arbitrary: t)
apply (simp)
apply (metis order-le-less rank-invar.simps(3) set-ConsD xt1(7))
done

```

```

lemma strong-remove1: invar bq ⇒ invar (remove1 t bq)
proof (induct bq arbitrary: t)
  case Nil
  then show ?case by simp
next
  case (Cons a bq)
  show ?case
  proof (cases t=a)
    case True
    from Cons(2) have invar bq by (rule invar-cons-down)
    with True show ?thesis by simp
  next
  case False
  from Cons(2) have invar bq by (rule invar-cons-down)
  with Cons(1)[of t] have si1: invar (remove1 t bq) .
  from False have invar (remove1 t (a # bq)) = invar (a # (remove1 t bq))
  by simp
  show ?thesis
  proof (cases remove1 t bq)

```

```

    case Nil
    with si1 Cons(2) False show ?thesis by (simp add: invar-def)
  next
  case Cons': (Cons aa list)
  from Cons have tree-invar a by (simp add: invar-def)
  from Cons first-less[of a bq] have  $\forall t \in \text{set } (\text{remove1 } t \text{ bq}). \text{rank } a < \text{rank } t$ 
    by (metis notin-set-remove1 invar-def)
  with Cons' have rank a < rank aa by simp
  with si1 Cons(2) False Cons' invar-cons-up[of aa list a] show ?thesis
    by (simp add: invar-def)
  qed
qed
qed

```

**theorem** *deleteMin-invar*:

```

  assumes invar bq
    and bq  $\neq []$ 
  shows invar (deleteMin bq)
  proof -
    have eq: invar (deleteMin bq) =
      invar (meld (rev (children (getMinTree bq))) (remove1 (getMinTree bq) bq))
    by (simp add: deleteMin-def Let-def)
    from assms mintree-exists[of bq] have ti: tree-invar (getMinTree bq)
      by (simp add: invar-def Let-def queue-invar-def)
    with strong-rev-children[of getMinTree bq]
    have m1: invar (rev (children (getMinTree bq))) .
    from strong-remove1[of bq getMinTree bq] assms(1)
    have m2: invar (remove1 (getMinTree bq) bq) .
    from meld-correct(1)[of rev (children (getMinTree bq))
      remove1 (getMinTree bq) bq] m1 m2
    have invar (meld (rev (children (getMinTree bq))) (remove1 (getMinTree bq) bq))
    .
    with eq show ?thesis ..
  qed

```

**lemma** *children-mset*:  $\text{queue-to-multiset } (\text{children } t) =$

$\text{tree-to-multiset } t - \{\# (\text{val } t, \text{prio } t) \#\}$

**proof** (cases t)

case (Node e a nat list)

thus ?thesis by (induct list) simp-all

qed

**lemma** *deleteMin-mset*:

assumes *queue-invar* q

and  $q \neq \text{Nil}$

shows  $\text{queue-to-multiset } (\text{deleteMin } q) = \text{queue-to-multiset } q - \{\# (\text{findMin } q) \#\}$

**proof** -

from assms mintree-exists[of q] have min-in-q:  $\text{getMinTree } q \in \text{set } q$  by auto

```

with assms(1) have inv-min: tree-invar (getMinTree q)
  by (simp add: queue-invar-def)
from assms(2) have q-ne: q ≠ [] .
note inv-children = invar-children'[OF inv-min]
note inv-rev = queue-invar-rev[OF inv-children]
note inv-rem = queue-invar-remove1[OF assms(1), of getMinTree q]
note m-meld = meld-mset[OF inv-rev inv-rem]
note m-rem = remove1-mset[OF min-in-q]
note m-rev = qtmset-rev[of children (getMinTree q)]
note m-children = children-mset[of getMinTree q]
note min-subset-q = qtm-in-set-subset[OF min-in-q]
let ?Q = queue-to-multiset q
let ?MT = tree-to-multiset (getMinTree q)
from q-ne have head-subset-min:
  {# (val (getMinTree q), prio (getMinTree q)) #} ⊆ # ?MT
  by(cases getMinTree q) simp
let ?Q = queue-to-multiset q
let ?MT = tree-to-multiset (getMinTree q)
from m-meld m-rem m-rev m-children
  multiset-diff-union-assoc[OF head-subset-min, of ?Q - ?MT]
  mset-subset-eq-multiset-union-diff-commute[OF min-subset-q, of ?MT]
show ?thesis by (simp add: deleteMin-def union-ac Let-def findMin-def)
qed

```

```

lemma deleteMin-correct:
  assumes INV: invar q
  assumes NE: q ≠ Nil
  shows
    invar (deleteMin q)
    queue-to-multiset (deleteMin q) = queue-to-multiset q - {# (findMin q) #}
  using deleteMin-invar deleteMin-mset INV NE
  unfolding invar-def
  by auto

```

**end**

**interpretation** *BinomialHeapStruc*: *BinomialHeapStruc-loc* .

## 1.3 Hiding the Invariant

### 1.3.1 Datatype

```

typedef (overloaded) ('e, 'a) BinomialHeap =
  { q :: ('e, 'a)::linorder) BinomialHeapStruc.BinomialQueue-inv. BinomialHeapStruc.invar
  q }
apply (rule-tac x=Nil in exI)
apply auto
done

```

**lemma** *Rep-BinomialHeap-invar*[*simp*]:

*BinomialHeapStruc.invar* (*Rep-BinomialHeap* *x*)  
**using** *Rep-BinomialHeap*  
**by** (*auto*)

**lemma** [*simp*]:  
*BinomialHeapStruc.invar* *q*  $\implies$  *Rep-BinomialHeap* (*Abs-BinomialHeap* *q*) = *q*  
**using** *Abs-BinomialHeap-inverse* **by** *auto*

**lemma** [*simp*, *code abstype*]: *Abs-BinomialHeap* (*Rep-BinomialHeap* *q*) = *q*  
**by** (*rule Rep-BinomialHeap-inverse*)

**locale** *BinomialHeap-loc*  
**begin**

### 1.3.2 Operations

**definition** [*code*]:  
*to-mset* *t* == *BinomialHeapStruc.queue-to-multiset* (*Rep-BinomialHeap* *t*)

**definition** *empty* **where** *empty* == *Abs-BinomialHeap Nil*

**lemma** [*code abstract*, *simp*]: *Rep-BinomialHeap empty* = []  
**by** (*unfold empty-def*) *simp*

**definition** [*code*]: *isEmpty* *q* == *Rep-BinomialHeap q = Nil*

**lemma** *empty-rep*: *q=empty*  $\longleftrightarrow$  *Rep-BinomialHeap q = Nil*  
**apply** (*auto simp add: empty-def*)  
**apply** (*metis Rep-BinomialHeap-inverse*)  
**done**

**lemma** *isEmpty-correct*: *isEmpty* *q*  $\longleftrightarrow$  *q=empty*  
**by** (*simp add: empty-rep isEmpty-def*)

**definition**

*insert*  
 $:: 'e \Rightarrow ('a::\text{linorder}) \Rightarrow ('e,'a) \text{ BinomialHeap} \Rightarrow ('e,'a) \text{ BinomialHeap}$

**where** *insert* *e a q* ==  
*Abs-BinomialHeap* (*BinomialHeapStruc.insert* *e a* (*Rep-BinomialHeap* *q*))

**lemma** [*code abstract*]:  
*Rep-BinomialHeap* (*insert* *e a q*)  
= *BinomialHeapStruc.insert* *e a* (*Rep-BinomialHeap* *q*)  
**by** (*simp add: insert-def BinomialHeapStruc.insert-correct*)

**definition** [*code*]: *findMin* *q* == *BinomialHeapStruc.findMin* (*Rep-BinomialHeap* *q*)

**definition** *deleteMin* *q* ==

*if* *q=empty* *then empty*  
*else Abs-BinomialHeap* (*BinomialHeapStruc.deleteMin* (*Rep-BinomialHeap* *q*))



In this lemma, we do not use equality, but case-distinction for checking non-emptiness. That prevents the code generator from introducing an equality-class parameter for the entry type  $'a$ .

```

lemma [code abstract]: Rep-BinomialHeap (deleteMin q) =
  (case (Rep-BinomialHeap q) of [] => [] |
    - => BinomialHeapStruc.deleteMin (Rep-BinomialHeap q))
proof (cases Rep-BinomialHeap q)
  case Nil
  show ?thesis
  apply (simp add: Nil)
  apply (auto simp add: deleteMin-def BinomialHeapStruc.deleteMin-correct
    BinomialHeapStruc.empty-iff empty-rep Nil)
  done
next
  case (Cons a b)
  hence NE: Rep-BinomialHeap q ≠ [] by auto
  show ?thesis
  apply (simp add: Cons)
  apply (fold Cons)
  using NE
  by (auto simp add: deleteMin-def BinomialHeapStruc.deleteMin-correct
    BinomialHeapStruc.empty-iff empty-rep)
qed

```

```

definition meld q1 q2 ==
  Abs-BinomialHeap (BinomialHeapStruc.meld (Rep-BinomialHeap q1)
    (Rep-BinomialHeap q2))

```

```

lemma [code abstract]:
  Rep-BinomialHeap (meld q1 q2)
  = BinomialHeapStruc.meld (Rep-BinomialHeap q1) (Rep-BinomialHeap q2)
  by (simp add: meld-def BinomialHeapStruc.meld-correct)

```

### 1.3.3 Correctness

```

lemma empty-correct: to-mset q = {#} ⟷ q=empty
  by (simp add: to-mset-def BinomialHeapStruc.empty-iff empty-rep)

```

```

lemma to-mset-of-empty[simp]: to-mset empty = {#}
  by (simp add: empty-correct)

```

```

lemma insert-correct: to-mset (insert e a q) = to-mset q + {#(e,a)#}
  apply (unfold insert-def to-mset-def)
  apply (simp add: BinomialHeapStruc.insert-correct)
  done

```

```

lemma findMin-correct:
  assumes q≠empty

```

**shows**  
 $findMin\ q \in \#\ to-mset\ q$   
 $\forall y \in set-mset\ (to-mset\ q).\ snd\ (findMin\ q) \leq snd\ y$   
**using** *assms*  
**apply** (*unfold findMin-def to-mset-def*)  
**apply** (*simp-all add: empty-rep BinomialHeapStruc.findMin-correct*)  
**done**

**lemma** *deleteMin-correct*:  
**assumes**  $q \neq empty$   
**shows**  $to-mset\ (deleteMin\ q) = to-mset\ q - \{\# findMin\ q\ \# \}$   
**using** *assms*  
**apply** (*unfold findMin-def deleteMin-def to-mset-def*)  
**apply** (*simp-all add: empty-rep BinomialHeapStruc.deleteMin-correct*)  
**done**

**lemma** *meld-correct*:  
**shows**  $to-mset\ (meld\ q\ q') = to-mset\ q + to-mset\ q'$   
**apply** (*unfold to-mset-def meld-def*)  
**apply** (*simp-all add: BinomialHeapStruc.meld-correct*)  
**done**

Correctness lemmas to be used with simplifier

**lemmas** *correct = empty-correct deleteMin-correct meld-correct*

**end**

**interpretation** *BinomialHeap: BinomialHeap-loc* .

## 1.4 Documentation

*BinomialHeap.to-mset::('a, 'b) BinomialHeap  $\Rightarrow$  ('a  $\times$  'b) multiset*  
 Abstraction to multiset.

*BinomialHeap.empty::('a, 'b) BinomialHeap*

The empty heap. ( $O(1)$ )

**Spec** *BinomialHeap.empty-correct*:

$(BinomialHeap.to-mset\ q = \{\#\}) = (q = BinomialHeap.empty)$

*BinomialHeap.isEmpty::('a, 'b) BinomialHeap  $\Rightarrow$  bool*

Checks whether heap is empty. Mainly used to work around code-generation issues. ( $O(1)$ )

**Spec** *BinomialHeap.isEmpty-correct*:

$BinomialHeap.isEmpty\ q = (q = BinomialHeap.empty)$

*BinomialHeap.insert::'a  $\Rightarrow$  'b  $\Rightarrow$  ('a, 'b) BinomialHeap  $\Rightarrow$  ('a, 'b) BinomialHeap*

Inserts element ( $O(\log(n))$ )

**Spec** *BinomialHeap.insert-correct*:

$BinomialHeap.to-mset (BinomialHeap.insert\ e\ a\ q) =$   
 $BinomialHeap.to-mset\ q + \{\#(e, a)\#}$

$BinomialHeap.findMin::('a, 'b)\ BinomialHeap \Rightarrow 'a \times 'b$   
 Returns a minimal element ( $O(\log(n))$ )

**Spec**  $BinomialHeap.findMin$ -correct:

$q \neq BinomialHeap.empty \implies BinomialHeap.findMin\ q \in \# BinomialHeap.to-mset\ q$   
 $q \neq BinomialHeap.empty \implies$   
 $\forall y \in \# BinomialHeap.to-mset\ q. snd\ (BinomialHeap.findMin\ q) \leq snd\ y$

$BinomialHeap.deleteMin::('a, 'b)\ BinomialHeap \Rightarrow ('a, 'b)\ BinomialHeap$   
 Deletes the element that is returned by  $find\_min$

**Spec**  $BinomialHeap.deleteMin$ -correct:

$q \neq BinomialHeap.empty \implies$   
 $BinomialHeap.to-mset\ (BinomialHeap.deleteMin\ q) =$   
 $BinomialHeap.to-mset\ q - \{\# BinomialHeap.findMin\ q\#}$

$BinomialHeap.meld$

$BinomialHeap.meld::('a, 'b)\ BinomialHeap$   
 $\Rightarrow ('a, 'b)\ BinomialHeap \Rightarrow ('a, 'b)\ BinomialHeap$

Melds two heaps ( $O(\log(n + m))$ )

**Spec**  $BinomialHeap.meld$ -correct:

$BinomialHeap.to-mset\ (BinomialHeap.meld\ q\ q') =$   
 $BinomialHeap.to-mset\ q + BinomialHeap.to-mset\ q'$

end

## 2 Skew Binomial Heaps

**theory**  $SkewBinomialHeap$   
**imports**  $Main\ HOL-Library.Multiset$   
**begin**

Skew Binomial Queues as specified by Brodal and Okasaki [1] are a data structure for priority queues with worst case  $O(1)$   $findMin$ ,  $insert$ , and  $meld$  operations, and worst-case logarithmic  $deleteMin$  operation. They are derived from priority queues in three steps:

1. Skew binomial trees are used to eliminate the possibility of cascading links during insert operations. This reduces the complexity of an insert operation to  $O(1)$ .

2. The current minimal element is cached. This approach, known as *global root*, reduces the cost of a *findMin*-operation to  $O(1)$ .
3. By allowing skew binomial queues to contain skew binomial queues, the cost for meld-operations is reduced to  $O(1)$ . This approach is known as *data-structural bootstrapping*.

In this theory, we combine Steps 2 and 3, i.e. we first implement skew binomial queues, and then bootstrap them. The bootstrapping implicitly introduces a global root, such that we also get a constant time *findMin* operation.

**locale** *SkewBinomialHeapStruc-loc*  
**begin**

## 2.1 Datatype

**datatype** (*'e*, *'a*) *SkewBinomialTree* =  
*Node* (*val*: *'e*) (*prio*: *'a*::*linorder*) (*rank*: *nat*) (*children*: (*'e*, *'a*) *SkewBinomialTree list*)

**type-synonym** (*'e*, *'a*) *SkewBinomialQueue* = (*'e*, *'a*::*linorder*) *SkewBinomialTree list*

### 2.1.1 Abstraction to Multisets

Returns a multiset with all (element, priority) pairs from a queue

**fun** *tree-to-multiset*  
 :: (*'e*, *'a*::*linorder*) *SkewBinomialTree*  $\Rightarrow$  (*'e*  $\times$  *'a*) *multiset*  
**and** *queue-to-multiset*  
 :: (*'e*, *'a*::*linorder*) *SkewBinomialQueue*  $\Rightarrow$  (*'e*  $\times$  *'a*) *multiset* **where**  
*tree-to-multiset* (*Node e a r ts*) =  $\{\#(e,a)\# \} + \text{queue-to-multiset } ts \mid$   
*queue-to-multiset* [] =  $\{\#\}$   $\mid$   
*queue-to-multiset* (*t* $\#$ *q*) = *tree-to-multiset t* + *queue-to-multiset q*

**lemma** *tm-children*: *tree-to-multiset t* =  
 $\{\#(\text{val } t, \text{prio } t)\# \} + \text{queue-to-multiset } (\text{children } t)$   
**by** (*cases t*) *auto*

**lemma** *qtm-conc[simp]*: *queue-to-multiset (q@q')*  
 = *queue-to-multiset q* + *queue-to-multiset q'*  
**by** (*induct q*) (*auto simp add: union-ac*)

### 2.1.2 Invariant

Link two trees of rank *r* to a new tree of rank *r* + 1

```

fun link :: ('e, 'a::linorder) SkewBinomialTree ⇒ ('e, 'a) SkewBinomialTree ⇒
('e, 'a) SkewBinomialTree where
link (Node e1 a1 r1 ts1) (Node e2 a2 r2 ts2) =
  (if a1 ≤ a2
   then (Node e1 a1 (Suc r1) ((Node e2 a2 r2 ts2)#ts1))
   else (Node e2 a2 (Suc r2) ((Node e1 a1 r1 ts1)#ts2)))

```

Link two trees of rank  $r$  and a new element to a new tree of rank  $r + 1$

```

fun skewlink :: 'e ⇒ 'a::linorder ⇒ ('e, 'a) SkewBinomialTree ⇒
('e, 'a) SkewBinomialTree ⇒ ('e, 'a) SkewBinomialTree where
skewlink e a t t' = (if a ≤ (prio t) ∧ a ≤ (prio t')
 then (Node e a (Suc (rank t)) [t,t'])
 else (if (prio t) ≤ (prio t')
 then
   Node (val t) (prio t) (Suc (rank t)) (Node e a 0 [] # t' # children t)
 else
   Node (val t') (prio t') (Suc (rank t')) (Node e a 0 [] # t # children t')))

```

The invariant for trees claims that a tree labeled rank 0 has no children, and a tree labeled rank  $r + 1$  is the result of an ordinary link or a skew link of two trees with rank  $r$ .

```

function tree-invar :: ('e, 'a::linorder) SkewBinomialTree ⇒ bool where
tree-invar (Node e a 0 ts) = (ts = []) |
tree-invar (Node e a (Suc r) ts) = (∃ e1 a1 ts1 e2 a2 ts2 e' a'.
tree-invar (Node e1 a1 r ts1) ∧ tree-invar (Node e2 a2 r ts2) ∧
((Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2) ∨
(Node e a (Suc r) ts) = skewlink e' a' (Node e1 a1 r ts1) (Node e2 a2 r ts2)))
by pat-completeness auto
termination
apply(relation measure rank)
apply auto
done

```

A heap satisfies the invariant, if all contained trees satisfy the invariant, the ranks of the trees in the heap are distinct, except that the first two trees may have same rank, and the ranks are ordered in ascending order.

First part: All trees inside the queue satisfy the invariant.

```

definition queue-invar :: ('e, 'a::linorder) SkewBinomialQueue ⇒ bool where
queue-invar q ≡ (∀ t ∈ set q. tree-invar t)

```

```

lemma queue-invar-simps[simp]:
queue-invar []
queue-invar (t#q) ⟷ tree-invar t ∧ queue-invar q
queue-invar (q@q') ⟷ queue-invar q ∧ queue-invar q'
queue-invar q ⟹ t ∈ set q ⟹ tree-invar t
unfolding queue-invar-def by auto

```

Second part: The ranks of the trees in the heap are distinct, except that the

first two trees may have same rank, and the ranks are ordered in ascending order.

For tail of queue

```
fun rank-invar :: ('e, 'a::linorder) SkewBinomialQueue  $\Rightarrow$  bool where
  rank-invar [] = True |
  rank-invar [t] = True |
  rank-invar (t # t' # bq) = (rank t < rank t'  $\wedge$  rank-invar (t' # bq))
```

For whole queue: First two elements may have same rank

```
fun rank-skew-invar :: ('e, 'a::linorder) SkewBinomialQueue  $\Rightarrow$  bool where
  rank-skew-invar [] = True |
  rank-skew-invar [t] = True |
  rank-skew-invar (t # t' # bq) = ((rank t  $\leq$  rank t')  $\wedge$  rank-invar (t' # bq))
```

```
definition tail-invar :: ('e, 'a::linorder) SkewBinomialQueue  $\Rightarrow$  bool where
  tail-invar bq = (queue-invar bq  $\wedge$  rank-invar bq)
```

```
definition invar :: ('e, 'a::linorder) SkewBinomialQueue  $\Rightarrow$  bool where
  invar bq = (queue-invar bq  $\wedge$  rank-skew-invar bq)
```

```
lemma invar-empty[simp]:
  invar []
  tail-invar []
unfolding invar-def tail-invar-def by auto
```

```
lemma invar-tail-invar:
  invar (t # bq)  $\Longrightarrow$  tail-invar bq
unfolding invar-def tail-invar-def
by (cases bq) simp-all
```

```
lemma link-mset[simp]: tree-to-multiset (link t1 t2)
  = tree-to-multiset t1 + tree-to-multiset t2
by (cases t1, cases t2, auto simp add:union-ac)
```

```
lemma link-tree-invar:  $\llbracket$ tree-invar t1; tree-invar t2; rank t1 = rank t2 $\rrbracket \Longrightarrow$ 
  tree-invar (link t1 t2)
by (cases t1, cases t2, simp, blast)
```

```
lemma skewlink-mset[simp]: tree-to-multiset (skewlink e a t1 t2)
  = {# (e,a) #} + tree-to-multiset t1 + tree-to-multiset t2
by (cases t1, cases t2, auto simp add:union-ac)
```

```
lemma skewlink-tree-invar:  $\llbracket$ tree-invar t1; tree-invar t2; rank t1 = rank t2 $\rrbracket \Longrightarrow$ 
  tree-invar (skewlink e a t1 t2)
by (cases t1, cases t2, simp, blast)
```

```
lemma rank-link: rank t = rank t'  $\Longrightarrow$  rank (link t t') = rank t + 1
```

```

apply (cases t)
apply (cases t')
apply(auto)
done

```

**lemma** *rank-skew-rank-invar*:  $\text{rank-skew-invar } (t \# \text{bq}) \implies \text{rank-invar } \text{bq}$   
**by** (cases bq) simp-all

**lemma** *rank-invar-rank-skew*:  
**assumes** *rank-invar q*  
**shows** *rank-skew-invar q*  
**proof** (cases q)  
 case Nil  
 then show ?thesis by simp  
next  
 case (Cons - list)  
 with assms show ?thesis  
 by (cases list) simp-all  
**qed**

**lemma** *rank-invar-cons-up*:  
 $\llbracket \text{rank-invar } (t \# \text{bq}); \text{rank } t' < \text{rank } t \rrbracket \implies \text{rank-invar } (t' \# t \# \text{bq})$   
**by** simp

**lemma** *rank-skew-cons-up*:  
 $\llbracket \text{rank-invar } (t \# \text{bq}); \text{rank } t' \leq \text{rank } t \rrbracket \implies \text{rank-skew-invar } (t' \# t \# \text{bq})$   
**by** simp

**lemma** *rank-invar-cons-down*:  $\text{rank-invar } (t \# \text{bq}) \implies \text{rank-invar } \text{bq}$   
**by** (cases bq) simp-all

**lemma** *rank-invar-hd-cons*:  
 $\llbracket \text{rank-invar } \text{bq}; \text{rank } t < \text{rank } (\text{hd } \text{bq}) \rrbracket \implies \text{rank-invar } (t \# \text{bq})$   
**apply**(cases bq)  
**apply**(auto)  
**done**

**lemma** *tail-invar-cons-up*:  
 $\llbracket \text{tail-invar } (t \# \text{bq}); \text{rank } t' < \text{rank } t; \text{tree-invar } t' \rrbracket$   
 $\implies \text{tail-invar } (t' \# t \# \text{bq})$   
**unfolding** *tail-invar-def*  
**apply** (cases bq)  
**apply** simp-all  
**done**

**lemma** *tail-invar-cons-up-invar*:  
 $\llbracket \text{tail-invar } (t \# \text{bq}); \text{rank } t' \leq \text{rank } t; \text{tree-invar } t' \rrbracket \implies \text{invar } (t' \# t \# \text{bq})$   
**by** (cases bq) (simp-all add: invar-def tail-invar-def)

```

lemma tail-invar-cons-down:
  tail-invar (t # bq)  $\implies$  tail-invar bq
  unfolding tail-invar-def
  by (cases bq) simp-all

lemma tail-invar-app-single:
   $\llbracket$ tail-invar bq;  $\forall t \in \text{set } bq. \text{rank } t < \text{rank } t'; \text{tree-invar } t \rrbracket$ 
   $\implies$  tail-invar (bq @ [t'])
proof (induct bq)
  case Nil
  then show ?case by (simp add: tail-invar-def)
next
  case (Cons a bq)
  from  $\langle$ tail-invar (a # bq) $\rangle$  have tail-invar bq
    by (rule tail-invar-cons-down)
  with Cons have tail-invar (bq @ [t']) by simp
  with Cons show ?case
    by (cases bq) (simp-all add: tail-invar-cons-up tail-invar-def)
qed

lemma invar-app-single:
   $\llbracket$ invar bq;  $\forall t \in \text{set } bq. \text{rank } t < \text{rank } t'; \text{tree-invar } t \rrbracket$ 
   $\implies$  invar (bq @ [t'])
proof (induct bq)
  case Nil
  then show ?case by (simp add: invar-def)
next
  case (Cons a bq)
  show ?case
  proof (cases bq)
    case Nil
    with Cons show ?thesis by (simp add: invar-def)
  next
  case Cons': (Cons ta qa)
  from Cons(2) have a1: tail-invar bq by (rule invar-tail-invar)
  from Cons(3) have a2:  $\forall t \in \text{set } bq. \text{rank } t < \text{rank } t'$  by simp
  from a1 a2 Cons(4) tail-invar-app-single[of bq t']
  have tail-invar (bq @ [t']) by simp
  with Cons Cons' show ?thesis
    by (simp-all add: tail-invar-cons-up-invar invar-def tail-invar-def)
qed
qed

lemma invar-children:
  assumes tree-invar ((Node e a r ts)::(('e, 'a)::linorder) SkewBinomialTree)
  shows queue-invar ts using assms
proof (induct r arbitrary: e a ts)
  case 0

```



```

then show ?case by simp
next
case (Suc r)
from Suc(2) obtain e1 a1 ts1 e2 a2 ts2 e' a' where
  inv-t1: tree-invar (Node e1 a1 r ts1) and
  inv-t2: tree-invar (Node e2 a2 r ts2) and
  link-or-skew:
    ((Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2)
    ∨ (Node e a (Suc r) ts)
    = skewlink e' a' (Node e1 a1 r ts1) (Node e2 a2 r ts2))
  by (simp only: tree-invar.simps) blast
from Suc(1)[OF inv-t1] inv-t2
have case1: queue-invar ((Node e2 a2 r ts2) # ts1) by simp
from Suc(1)[OF inv-t2] inv-t1
have case2: queue-invar ((Node e1 a1 r ts1) # ts2) by simp
show ?case
proof (cases (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2))
  case True
  hence ts =
    (if a1 ≤ a2
     then (Node e2 a2 r ts2) # ts1
     else (Node e1 a1 r ts1) # ts2) by auto
  with case1 case2 show ?thesis by simp
next
case False
with link-or-skew
have Node e a (Suc r) ts =
  skewlink e' a' (Node e1 a1 r ts1) (Node e2 a2 r ts2) by simp
hence ts =
  (if a' ≤ a1 ∧ a' ≤ a2
   then [(Node e1 a1 r ts1), (Node e2 a2 r ts2)]
   else (if a1 ≤ a2
        then (Node e' a' 0 []) # (Node e2 a2 r ts2) # ts1
        else (Node e' a' 0 []) # (Node e1 a1 r ts1) # ts2)) by auto
  with case1 case2 show ?thesis by simp
qed
qed

```

### 2.1.3 Heap Order

```

fun heap-ordered :: ('e, 'a::linorder) SkewBinomialTree ⇒ bool where
  heap-ordered (Node e a r ts)
    = (∀ x ∈ set-mset (queue-to-multiset ts). a ≤ snd x)

```

The invariant for trees implies heap order.

```

lemma tree-invar-heap-ordered:
  fixes t :: ('e, 'a::linorder) SkewBinomialTree
  assumes tree-invar t
  shows heap-ordered t

```

```

proof (cases t)
  case (Node e a nat list)
  with assms show ?thesis
  proof (induct nat arbitrary: t e a list)
    case 0
    then show ?case by simp
  next
  case (Suc nat)
  from Suc(2,3) obtain t1 e1 a1 ts1 t2 e2 a2 ts2 e' a' where
    inv-t1: tree-invar t1 and
    inv-t2: tree-invar t2 and
    link-or-skew:  $t = \text{link } t1 \ t2 \vee t = \text{skewlink } e' \ a' \ t1 \ t2$  and
    eq-t1[simp]:  $t1 = (\text{Node } e1 \ a1 \ \text{nat } ts1)$  and
    eq-t2[simp]:  $t2 = (\text{Node } e2 \ a2 \ \text{nat } ts2)$ 
    by (simp only: tree-invar.simps) blast
  note heap-t1 = Suc(1)[OF inv-t1 eq-t1]
  note heap-t2 = Suc(1)[OF inv-t2 eq-t2]
  from link-or-skew heap-t1 heap-t2 show ?case
    by (cases t = link t1 t2) auto
  qed
qed

```

#### 2.1.4 Height and Length

Although complexity of HOL-functions cannot be expressed within HOL, we can express the height and length of a binomial heap. By showing that both, height and length, are logarithmic in the number of contained elements, we give strong evidence that our functions have logarithmic complexity in the number of elements.

Height of a tree and queue

```

fun height-tree :: ('e, ('a::linorder)) SkewBinomialTree  $\Rightarrow$  nat and
  height-queue :: ('e, ('a::linorder)) SkewBinomialQueue  $\Rightarrow$  nat
  where
    height-tree (Node e a r ts) = height-queue ts |
    height-queue [] = 0 |
    height-queue (t # ts) = max (Suc (height-tree t)) (height-queue ts)

```

```

lemma link-length: size (tree-to-multiset (link t1 t2)) =
  size (tree-to-multiset t1) + size (tree-to-multiset t2)
  apply(cases t1)
  apply(cases t2)
  apply simp
done

```

```

lemma tree-rank-estimate-upper:
  tree-invar (Node e a r ts)  $\implies$ 
  size (tree-to-multiset (Node e a r ts))  $\leq$  (2::nat)  $\wedge$  (Suc r) - 1
proof (induct r arbitrary: e a ts)

```

```

case 0
then show ?case by simp
next
case (Suc r)
from Suc(2) obtain e1 a1 ts1 e2 a2 ts2 e' a' where
  link:
    (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2) ∨
    (Node e a (Suc r) ts) = skewlink e' a' (Node e1 a1 r ts1) (Node e2 a2 r ts2)
  and inv1: tree-invar (Node e1 a1 r ts1)
  and inv2: tree-invar (Node e2 a2 r ts2)
  by simp blast
note iv1 = Suc(1)[OF inv1]
note iv2 = Suc(1)[OF inv2]
have (2::nat)^r - 1 + (2::nat)^r - 1 ≤ (2::nat)^(Suc r) - 1 by simp
with link Suc show ?case
  apply (cases Node e a (Suc r) ts = link (Node e1 a1 r ts1) (Node e2 a2 r ts2))
  using iv1 iv2 apply (simp del: link.simps)
  using iv1 iv2 apply (simp del: skewlink.simps)
  done
qed

```

**lemma** tree-rank-estimate-lower:  
 tree-invar (Node e a r ts)  $\implies$   
 size (tree-to-multiset (Node e a r ts))  $\geq$  (2::nat)<sup>^</sup>r  
**proof** (induct r arbitrary: e a ts)

```

case 0
then show ?case by simp
next
case (Suc r)
from Suc(2) obtain e1 a1 ts1 e2 a2 ts2 e' a' where
  link:
    (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2) ∨
    (Node e a (Suc r) ts) = skewlink e' a' (Node e1 a1 r ts1) (Node e2 a2 r ts2)
  and inv1: tree-invar (Node e1 a1 r ts1)
  and inv2: tree-invar (Node e2 a2 r ts2)
  by simp blast
note iv1 = Suc(1)[OF inv1]
note iv2 = Suc(1)[OF inv2]
have (2::nat)^r - 1 + (2::nat)^r - 1 ≤ (2::nat)^(Suc r) - 1 by simp
with link Suc show ?case
  apply (cases Node e a (Suc r) ts = link (Node e1 a1 r ts1) (Node e2 a2 r ts2))
  using iv1 iv2 apply (simp del: link.simps)
  using iv1 iv2 apply (simp del: skewlink.simps)
  done
qed

```

**lemma** tree-rank-height:

```

tree-invar (Node e a r ts)  $\implies$  height-tree (Node e a r ts) = r
proof (induct r arbitrary: e a ts)
  case 0
  then show ?case by simp
next
  case (Suc r)
  from Suc(2) obtain e1 a1 ts1 e2 a2 ts2 e' a' where
    link:
      (Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2)  $\vee$ 
      (Node e a (Suc r) ts) = skewlink e' a' (Node e1 a1 r ts1) (Node e2 a2 r ts2)
    and inv1: tree-invar (Node e1 a1 r ts1)
    and inv2: tree-invar (Node e2 a2 r ts2)
  by simp blast
note iv1 = Suc(1)[OF inv1]
note iv2 = Suc(1)[OF inv2]
from Suc(2) link show ?case
  apply (cases Node e a (Suc r) ts = link (Node e1 a1 r ts1) (Node e2 a2 r ts2))
  apply (cases a1  $\leq$  a2)
  using iv1 iv2 apply simp
  using iv1 iv2 apply simp
  apply (cases a'  $\leq$  a1  $\wedge$  a'  $\leq$  a2)
  apply (simp only: height-tree.simps)
  using iv1 iv2 apply simp
  apply (cases a1  $\leq$  a2)
  using iv1 iv2
  apply (simp del: tree-invar.simps link.simps)
  using iv1 iv2
  apply (simp del: tree-invar.simps link.simps)
  done
qed

```

A skew binomial tree of height  $h$  contains at most  $2^{h+1} - 1$  elements

```

theorem tree-height-estimate-upper:
  tree-invar t  $\implies$ 
  size (tree-to-multiset t)  $\leq$  (2::nat)  $\wedge$  (Suc (height-tree t)) - 1
  apply (cases t, simp only:)
  apply (frule tree-rank-estimate-upper)
  apply (frule tree-rank-height)
  apply (simp only: )
  done

```

A skew binomial tree of height  $h$  contains at least  $2^h$  elements

```

theorem tree-height-estimate-lower:
  tree-invar t  $\implies$  size (tree-to-multiset t)  $\geq$  (2::nat)  $\wedge$  (height-tree t)
  apply (cases t, simp only:)
  apply (frule tree-rank-estimate-lower)
  apply (frule tree-rank-height)
  apply (simp only: )
  done

```

**lemma** *size-mset-tree-upper*: *tree-invar t*  $\implies$   
*size (tree-to-multiset t)  $\leq$  (2::nat)<sup>^(Suc (rank t)) - (1::nat)</sup>*  
**apply** (*cases t*)  
**by** (*simp only: tree-rank-estimate-upper SkewBinomialTree.sel(3)*)

**lemma** *size-mset-tree-lower*: *tree-invar t*  $\implies$   
*size (tree-to-multiset t)  $\geq$  (2::nat)<sup>^(rank t)</sup>*  
**apply** (*cases t*)  
**by** (*simp only: tree-rank-estimate-lower SkewBinomialTree.sel(3)*)

**lemma** *invar-butlast*: *invar (bq @ [t])*  $\implies$  *invar bq*  
**unfolding** *invar-def*  
**apply** (*induct bq*)  
**apply** *simp*  
**apply** (*case-tac bq*)  
**apply** *simp*  
**apply** (*case-tac list*)  
**by** *simp-all*

**lemma** *invar-last-max*:  
*invar ((b#b'#bq) @ [m])*  $\implies \forall t \in \text{set } (b'\#bq). \text{rank } t < \text{rank } m$   
**unfolding** *invar-def*  
**apply** (*induct bq*) **apply** *simp* **apply** (*case-tac bq*) **apply** *simp* **by** *simp*

**lemma** *invar-last-max'*: *invar ((b#b'#bq) @ [m])*  $\implies \text{rank } b \leq \text{rank } b'$   
**unfolding** *invar-def* **by** *simp*

**lemma** *invar-length*: *invar bq*  $\implies \text{length } bq \leq \text{Suc } (\text{Suc } (\text{rank } (\text{last } bq)))$   
**proof** (*induct bq rule: rev-induct*)  
**case** *Nil* **thus** *?case* **by** *simp*  
**next**  
**case** (*snoc x xs*)  
**show** *?case* **proof** (*cases xs*)  
**case** *Nil* **thus** *?thesis* **by** *simp*  
**next**  
**case** [*simp*]: (*Cons xxs xx*)  
**note** *Cons' = Cons*  
**thus** *?thesis*  
**proof** (*cases xx*)  
**case** *Nil* **with** *snoc.prem*s *Cons* **show** *?thesis* **by** *simp*  
**next**  
**case** (*Cons xxxs xxx*)  
**from** *snoc.hyps*[*OF invar-butlast*[*OF snoc.prem*s]] **have**  
*IH: length xs  $\leq$  Suc (Suc (rank (last xs)))* .  
**also from** *invar-last-max*'[*OF snoc.prem*s[*unfolded Cons' Cons*]]  
*invar-last-max'*'[*OF snoc.prem*s[*unfolded Cons' Cons*]]

*last-in-set*[of *xs*] **Cons have**  
*Suc* (*rank* (*last xs*))  $\leq$  *rank* (*last* (*xs @ [x]*)) **by auto**  
**finally show** *?thesis* **by simp**  
**qed**  
**qed**  
**qed**

**lemma** *size-queue-sum-list*:  
*size* (*queue-to-multiset bq*) = *sum-list* (*map* (*size*  $\circ$  *tree-to-multiset*) *bq*)  
**by** (*induct bq*) *simp-all*

A skew binomial heap of length  $l$  contains at least  $2^{l-1} - 1$  elements.

**theorem** *queue-length-estimate-lower*:  
*invar bq*  $\implies$  (*size* (*queue-to-multiset bq*))  $\geq 2^{\wedge}(\text{length } bq - 1) - 1$   
**proof** (*induct bq rule: rev-induct*)  
**case Nil thus** *?case* **by simp**  
**next**  
**case** (*snoc x xs*) **thus** *?case*  
**proof** (*cases xs*)  
**case Nil thus** *?thesis* **by simp**  
**next**  
**case** [*simp*]: (*Cons xx xs*)  
**from** *snoc.hyps*[*OF invar-butlast*[*OF snoc.prem*s]]  
**have** *IH*:  $2^{\wedge}(\text{length } xs - 1) \leq \text{Suc}(\text{size}(\text{queue-to-multiset } xs))$  **by simp**  
**have** *size-q*:  
*size* (*queue-to-multiset* (*xs @ [x]*)) =  
*size* (*queue-to-multiset xs*) + *size* (*tree-to-multiset x*)  
**by** (*simp add: size-queue-sum-list*)  
**moreover**  
**from** *snoc.prem*s **have** *inv-x*: *tree-invar x* **by** (*simp add: invar-def*)  
**from** *size-mset-tree-lower*[*OF this*]  
**have**  $2^{\wedge}(\text{rank } x) \leq \text{size}(\text{tree-to-multiset } x)$  .  
**ultimately have**  
*eq*:  $\text{size}(\text{queue-to-multiset } xs) + (2::\text{nat})^{\wedge}(\text{rank } x) \leq$   
 $\text{size}(\text{queue-to-multiset } (xs @ [x]))$  **by simp**  
**from** *invar-length*[*OF snoc.prem*s] **have**  $\text{length } xs \leq (\text{rank } x + 1)$  **by simp**  
**hence** *snd*:  $(2::\text{nat})^{\wedge}(\text{length } xs - 1) \leq (2::\text{nat})^{\wedge}(\text{rank } x)$   
**by** (*simp del: power.simps*)  
**have**  
 $(2::\text{nat})^{\wedge}(\text{length } (xs @ [x]) - 1) =$   
 $(2::\text{nat})^{\wedge}(\text{length } xs - 1) + (2::\text{nat})^{\wedge}(\text{length } xs - 1)$   
**by auto**  
**with IH have**  
 $2^{\wedge}(\text{length } (xs @ [x]) - 1) \leq$   
 $\text{Suc}(\text{size}(\text{queue-to-multiset } xs)) + 2^{\wedge}(\text{length } xs - 1)$   
**by simp**  
**with snd have**  $2^{\wedge}(\text{length } (xs @ [x]) - 1) \leq$   
 $\text{Suc}(\text{size}(\text{queue-to-multiset } xs)) + 2^{\wedge} \text{rank } x$   
**by arith**

```

    with eq show ?thesis by simp
  qed
qed

```

## 2.2 Operations

### 2.2.1 Empty Tree

```

lemma empty-correct:  $q = Nil \longleftrightarrow \text{queue-to-multiset } q = \{\#\}$ 
  apply (cases q)
  apply simp
  apply (case-tac a)
  apply auto
  done

```

### 2.2.2 Insert

Inserts a tree into the queue, such that two trees of same rank get linked and are recursively inserted. This is the same definition as for binomial queues and is used for melding.

```

fun ins :: ('e, 'a::linorder) SkewBinomialTree  $\Rightarrow$  ('e, 'a) SkewBinomialQueue  $\Rightarrow$ 
  ('e, 'a) SkewBinomialQueue where
  ins t [] = [t] |
  ins t' (t # bq) =
    (if (rank t') < (rank t)
     then t' # t # bq
     else (if (rank t) < (rank t')
            then t # (ins t' bq)
            else ins (link t' t) bq))

```

Insert an element with priority into a queue using skewlinks.

```

fun insert :: 'e  $\Rightarrow$  'a::linorder  $\Rightarrow$  ('e, 'a) SkewBinomialQueue  $\Rightarrow$ 
  ('e, 'a) SkewBinomialQueue where
  insert e a [] = [Node e a 0 []] |
  insert e a [t] = [Node e a 0 [], t] |
  insert e a (t # t' # bq) =
    (if rank t  $\neq$  rank t'
     then (Node e a 0 []) # t # t' # bq
     else (skewlink e a t t') # bq)

```

**lemma** ins-mset:

```

[[tree-invar t; queue-invar q]]  $\implies$ 
  queue-to-multiset (ins t q) = tree-to-multiset t + queue-to-multiset q
by (induct q arbitrary: t) (auto simp: union-ac link-tree-invar)

```

**lemma** insert-mset: queue-invar q  $\implies$

```

  queue-to-multiset (insert e a q) =
  queue-to-multiset q + {# (e,a) #}
by (induct q rule: insert.induct) (auto simp add: union-ac ttm-children)

```

```

lemma ins-queue-invar:  $\llbracket \text{tree-invar } t; \text{queue-invar } q \rrbracket \implies \text{queue-invar } (\text{ins } t \ q)$ 
proof (induct q arbitrary: t)
  case Nil
  then show ?case by simp
next
  case (Cons a q)
  note iv = Cons(1)
  from Cons(2,3) show ?case
  apply (cases rank t < rank a)
  apply simp
  apply (cases rank t = rank a)
  defer
  using iv[of t] apply simp
proof goal-cases
  case prems: 1
  from prems(2) have inv-a: tree-invar a by simp
  from prems(2) have inv-q: queue-invar q by simp
  note inv-link = link-tree-invar[OF prems(1) inv-a prems(4)]
  from iv[OF inv-link inv-q] prems(4) show ?case by simp
qed
qed

```

```

lemma insert-queue-invar:  $\text{queue-invar } q \implies \text{queue-invar } (\text{insert } e \ a \ q)$ 
proof (induct q rule: insert.induct)
  case 1
  then show ?case by simp
next
  case 2
  then show ?case by simp
next
  case (3 e a t t' bq)
  show ?case
  proof (cases rank t = rank t')
  case False
  with 3 show ?thesis by simp
  next
  case True
  from 3 have inv-t: tree-invar t by simp
  from 3 have inv-t': tree-invar t' by simp
  from 3 skewlink-tree-invar[OF inv-t inv-t' True, of e a] True
  show ?thesis by simp
qed
qed

```

```

lemma rank-ins2:
  rank-invar bq  $\implies$ 
   $\text{rank } t \leq \text{rank } (\text{hd } (\text{ins } t \ bq))$ 
   $\vee (\text{rank } (\text{hd } (\text{ins } t \ bq)) = \text{rank } (\text{hd } bq) \wedge bq \neq [])$ 

```



```

apply (induct bq arbitrary: t)
apply auto
proof goal-cases
  case prems: (1 a bq t)
  hence r: rank (link t a) = rank a + 1 by (simp add: rank-link)
  with prems and prems(1)[of (link t a)] show ?case
    apply (cases bq)
    apply auto
    done
qed

lemma insert-rank-invar: rank-skew-invar q  $\implies$  rank-skew-invar (insert e a q)
proof (cases q, simp)
  fix t q'
  assume rank-skew-invar q q = t # q'
  thus rank-skew-invar (insert e a q)
  proof (cases q', (auto intro: gr0I)[1])
    fix t' q''
    assume rank-skew-invar q q = t # q' q' = t' # q''
    thus rank-skew-invar (insert e a q)
    apply(cases rank t = rank t') defer
    apply (auto intro: gr0I)[1]
    apply (simp del: skewlink.simps)
  proof goal-cases
    case prems: 1
    with rank-invar-cons-down[of t' q'] have rank-invar q' by simp
    show ?case
    proof (cases q'')
      case Nil
      then show ?thesis by simp
    next
      case (Cons t'' q''')
      with prems have rank t' < rank t'' by simp
      with prems have rank (skewlink e a t t')  $\leq$  rank t'' by simp
      with prems Cons rank-skew-cons-up[of t'' q''' skewlink e a t t']
      show ?thesis by simp
    qed
  qed
qed
qed

```

```

lemma insert-invar: invar q  $\implies$  invar (insert e a q)
  unfolding invar-def
  using insert-queue-invar[of q] insert-rank-invar[of q]
  by simp

```

```

theorem insert-correct:
  assumes I: invar q
  shows

```

```

invar (insert e a q)
queue-to-multiset (insert e a q) = queue-to-multiset q + {# (e,a) #}
using insert-mset[of q] insert-invar[of q] I
unfolding invar-def by simp-all

```

### 2.2.3 meld

Remove duplicate tree ranks by inserting the first tree of the queue into the rest of the queue.

```

fun uniqify
  :: ('e, 'a::linorder) SkewBinomialQueue ⇒ ('e, 'a) SkewBinomialQueue
where
  uniqify [] = [] |
  uniqify (t#bq) = ins t bq

```

Meld two uniquified queues using the same definition as for binomial queues.

```

fun meldUniq
  :: ('e, 'a::linorder) SkewBinomialQueue ⇒ ('e, 'a) SkewBinomialQueue ⇒
  ('e, 'a) SkewBinomialQueue where
  meldUniq [] bq = bq |
  meldUniq bq [] = bq |
  meldUniq (t1#bq1) (t2#bq2) = (if rank t1 < rank t2
    then t1 # (meldUniq bq1 (t2#bq2))
    else (if rank t2 < rank t1
      then t2 # (meldUniq (t1#bq1) bq2)
      else ins (link t1 t2) (meldUniq bq1 bq2)))

```

Meld two queues using above functions.

```

definition meld
  :: ('e, 'a::linorder) SkewBinomialQueue ⇒ ('e, 'a) SkewBinomialQueue ⇒
  ('e, 'a) SkewBinomialQueue where
  meld bq1 bq2 = meldUniq (uniqify bq1) (uniqify bq2)

```

```

lemma invar-uniqify: queue-invar q ⇒ queue-invar (uniqify q)
apply(cases q, simp)
apply(auto simp add: ins-queue-invar)
done

```

```

lemma invar-meldUniq: [queue-invar q; queue-invar q'] ⇒ queue-invar (meldUniq
q q')

```

```

proof (induct q q' rule: meldUniq.induct)
  case 1
  then show ?case by simp
next
  case 2
  then show ?case by simp
next
  case (3 t1 bq1 t2 bq2)

```

```

consider (lt) rank t1 < rank t2 | (gt) rank t1 > rank t2 | (eq) rank t1 = rank
t2
  by atomize-elim auto
then show ?case
proof cases
  case t1t2: lt
    from 3(4) have inv-bq1: queue-invar bq1 by simp
    from 3(4) have inv-t1: tree-invar t1 by simp
    from 3(1)[OF t1t2 inv-bq1 3(5)] inv-t1 t1t2
    show ?thesis by simp
  next
    case t1t2: gt
      from 3(5) have inv-bq2: queue-invar bq2 by simp
      from 3(5) have inv-t2: tree-invar t2 by simp
      from t1t2 have ¬ rank t1 < rank t2 by simp
      from 3(2) [OF this t1t2 3(4) inv-bq2] inv-t2 t1t2
      show ?thesis by simp
    next
      case t1t2: eq
        from 3(4) have inv-bq1: queue-invar bq1 by simp
        from 3(4) have inv-t1: tree-invar t1 by simp
        from 3(5) have inv-bq2: queue-invar bq2 by simp
        from 3(5) have inv-t2: tree-invar t2 by simp
        note inv-link = link-tree-invar[OF inv-t1 inv-t2 t1t2]
        from t1t2 have ¬ rank t1 < rank t2 ¬ rank t2 < rank t1 by auto
        note inv-meld = 3(3)[OF this inv-bq1 inv-bq2]
        from ins-queue-invar[OF inv-link inv-meld] t1t2
        show ?thesis by simp
      qed
    qed

```

```

lemma meld-queue-invar:
  assumes queue-invar q
    and queue-invar q'
  shows queue-invar (meld q q')
proof –
  note inv-uniq-q = invar-uniqify[OF assms(1)]
  note inv-uniq-q' = invar-uniqify[OF assms(2)]
  note inv-meldUniq = invar-meldUniq[OF inv-uniq-q inv-uniq-q']
  thus ?thesis by (simp add: meld-def)
qed

```

```

lemma uniqify-mset: queue-invar q  $\implies$  queue-to-multiset q = queue-to-multiset
(uniqify q)
  apply (cases q)
  apply simp
  apply (simp add: ins-mset)
done

```

**lemma** *meldUniq-mset*:  $\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies$   
 $\text{queue-to-multiset } (\text{meldUniq } q \ q') =$   
 $\text{queue-to-multiset } q + \text{queue-to-multiset } q'$   
**by** (*induct*  $q \ q'$  *rule*: *meldUniq.induct*)  
*(auto simp: ins-mset link-tree-invar invar-meldUniq union-ac)*

**lemma** *meld-mset*:  
 $\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies$   
 $\text{queue-to-multiset } (\text{meld } q \ q') = \text{queue-to-multiset } q + \text{queue-to-multiset } q'$   
**by** (*simp add*: *meld-def meldUniq-mset invar-uniqify uniqify-mset[symmetric]*)

Ins operation satisfies rank invariant, see binomial queues

**lemma** *rank-ins*:  $\text{rank-invar } bq \implies \text{rank-invar } (\text{ins } t \ bq)$   
**proof** (*induct*  $bq$  *arbitrary*:  $t$ )  
  **case** *Nil*  
  **then show** *?case* **by** *simp*  
**next**  
  **case** (*Cons*  $a \ bq$ )  
  **then show** *?case*  
  **apply** *auto*  
  **proof** *goal-cases*  
  **case** *prems: 1*  
  **hence** *inv*:  $\text{rank-invar } (\text{ins } t \ bq)$  **by** (*cases*  $bq$ ) *simp-all*  
  **from** *prems* **have**  $hd: bq \neq [] \implies \text{rank } a < \text{rank } (hd \ bq)$  **by** (*cases*  $bq$ ) *auto*  
  **from** *prems* **have**  $\text{rank } t \leq \text{rank } (hd \ (\text{ins } t \ bq))$   
   $\vee (\text{rank } (hd \ (\text{ins } t \ bq)) = \text{rank } (hd \ bq) \wedge bq \neq [])$   
  **by** (*metis rank-ins2 rank-invar-cons-down*)  
  **with** *prems* **have**  $\text{rank } a < \text{rank } (hd \ (\text{ins } t \ bq))$   
   $\vee (\text{rank } (hd \ (\text{ins } t \ bq)) = \text{rank } (hd \ bq) \wedge bq \neq [])$  **by** *auto*  
  **with** *prems* **and** *inv* **and** *hd* **show** *?case*  
  **by** (*auto simp add*: *rank-invar-hd-cons*)  
**next**  
  **case** *prems: 2*  
  **hence** *inv*:  $\text{rank-invar } bq$  **by** (*cases*  $bq$ ) *simp-all*  
  **with** *prems* **and** *prems(1)[of (link t a)]* **show** *?case* **by** *simp*  
**qed**  
**qed**

**lemma** *rank-uniqify*:  
  **assumes** *rank-skew-invar*  $q$   
  **shows**  $\text{rank-invar } (\text{uniqify } q)$   
**proof** (*cases*  $q$ )  
  **case** *Nil*  
  **then show** *?thesis* **by** *simp*  
**next**  
  **case** (*Cons*  $a \ list$ )  
  **with** *rank-skew-rank-invar[of a list]* *rank-ins[of list a]* *assms*  
  **show** *?thesis* **by** *simp*

**qed**

**lemma** *rank-ins-min*:  $\text{rank-invar } bq \implies \text{rank } (\text{hd } (\text{ins } t \text{ } bq)) \geq \min (\text{rank } t) (\text{rank } (\text{hd } bq))$

**proof** (*induct* *bq arbitrary*: *t*)

**case** *Nil*

**then show** *?case* **by** *simp*

**next**

**case** (*Cons a bq*)

**then show** *?case*

**apply** *auto*

**proof** *goal-cases*

**case** *prems: 1*

**hence** *inv*:  $\text{rank-invar } bq$  **by** (*cases* *bq*) *simp-all*

**from** *prems* **have** *r*:  $\text{rank } (\text{link } t \text{ } a) = \text{rank } a + 1$  **by** (*simp add*: *rank-link*)

**with** *prems* **and** *inv* **and** *prems(1)[of (link t a)]* **show** *?case*

**by** (*cases* *bq*) *auto*

**qed**

**qed**

**lemma** *rank-invar-not-empty-hd*:  $\llbracket \text{rank-invar } (t \# bq); bq \neq [] \rrbracket \implies \text{rank } t < \text{rank } (\text{hd } bq)$

**by** (*induct* *bq arbitrary*: *t*) *auto*

**lemma** *rank-invar-meldUniq-strong*:

$\llbracket \text{rank-invar } bq1; \text{rank-invar } bq2 \rrbracket \implies$

$\text{rank-invar } (\text{meldUniq } bq1 \text{ } bq2)$

$\wedge \text{rank } (\text{hd } (\text{meldUniq } bq1 \text{ } bq2)) \geq \min (\text{rank } (\text{hd } bq1)) (\text{rank } (\text{hd } bq2))$

**proof** (*induct* *bq1 bq2 rule*: *meldUniq.induct*)

**case** *1*

**then show** *?case* **by** *simp*

**next**

**case** *2*

**then show** *?case* **by** *simp*

**next**

**case** (*3 t1 bq1 t2 bq2*)

**from** *3* **have** *inv1*:  $\text{rank-invar } bq1$  **by** (*cases* *bq1*) *simp-all*

**from** *3* **have** *inv2*:  $\text{rank-invar } bq2$  **by** (*cases* *bq2*) *simp-all*

**from** *inv1* **and** *inv2* **and** *3* **show** *?case*

**apply** *auto*

**proof** *goal-cases*

**let** *?t* = *t2*

**let** *?bq* = *bq2*

**let** *?meldUniq* =  $\text{rank } t2 < \text{rank } (\text{hd } (\text{meldUniq } (t1 \# bq1) \text{ } bq2))$

**case** *prems: 1*

**hence** *?bq*  $\neq [] \implies \text{rank } ?t < \text{rank } (\text{hd } ?bq)$

**by** (*simp add*: *rank-invar-not-empty-hd*)

**with** *prems* **have** *ne*: *?bq*  $\neq [] \implies ?\text{meldUniq}$  **by** *simp*

```

from prems have  $?bq = [] \implies ?meldUniq$  by simp
with ne have  $?meldUniq$  by (cases  $?bq = []$ )
with prems show  $?case$  by (simp add: rank-invar-hd-cons)
next — analog
let  $?t = t1$ 
let  $?bq = bq1$ 
let  $?meldUniq = rank\ t1 < rank\ (hd\ (meldUniq\ bq1\ (t2\ \# \ bq2)))$ 
case prems: 2
hence  $?bq \neq [] \implies rank\ ?t < rank\ (hd\ ?bq)$ 
  by (simp add: rank-invar-not-empty-hd)
with prems have ne:  $?bq \neq [] \implies ?meldUniq$  by simp
from prems have  $?bq = [] \implies ?meldUniq$  by simp
with ne have  $?meldUniq$  by (cases  $?bq = []$ )
with prems show  $?case$  by (simp add: rank-invar-hd-cons)
next
  case 3
  thus  $?case$  by (simp add: rank-ins)
next
  case prems: 4
  then have r:  $rank\ (link\ t1\ t2) = rank\ t2 + 1$  by (simp add: rank-link)
  have m:  $meldUniq\ bq1\ [] = bq1$  by (cases  $bq1$ ) auto

from inv1 and inv2 and prems have
  mm:  $min\ (rank\ (hd\ bq1))\ (rank\ (hd\ bq2)) \leq rank\ (hd\ (meldUniq\ bq1\ bq2))$ 
  by simp
from  $\langle rank-invar\ (t1\ \# \ bq1) \rangle$  have  $bq1 \neq [] \implies rank\ t1 < rank\ (hd\ bq1)$ 
  by (simp add: rank-invar-not-empty-hd)
with prems have r1:  $bq1 \neq [] \implies rank\ t2 < rank\ (hd\ bq1)$  by simp
from  $\langle rank-invar\ (t2\ \# \ bq2) \rangle$  have r2:  $bq2 \neq [] \implies rank\ t2 < rank\ (hd\ bq2)$ 
  by (simp add: rank-invar-not-empty-hd)

from inv1 r r1 rank-ins-min[of  $bq1\ (link\ t1\ t2)$ ] have
  abc1:  $bq1 \neq [] \implies rank\ t2 \leq rank\ (hd\ (ins\ (link\ t1\ t2)\ bq1))$  by simp
from inv2 r r2 rank-ins-min[of  $bq2\ (link\ t1\ t2)$ ] have
  abc2:  $bq2 \neq [] \implies rank\ t2 \leq rank\ (hd\ (ins\ (link\ t1\ t2)\ bq2))$  by simp

from r1 r2 mm have
   $\llbracket bq1 \neq []; bq2 \neq [] \rrbracket \implies rank\ t2 < rank\ (hd\ (meldUniq\ bq1\ bq2))$ 
  by (simp)
with  $\langle rank-invar\ (meldUniq\ bq1\ bq2) \rangle$  r
  rank-ins-min[of  $meldUniq\ bq1\ bq2\ link\ t1\ t2$ ]
have  $\llbracket bq1 \neq []; bq2 \neq [] \rrbracket \implies$ 
   $rank\ t2 < rank\ (hd\ (ins\ (link\ t1\ t2)\ (meldUniq\ bq1\ bq2)))$ 
  by simp
with inv1 and inv2 and r m r1 show  $?case$ 
  apply(cases  $bq2 = []$ )
  apply(cases  $bq1 = []$ )
  apply(simp)
  apply(auto simp add: abc1)

```

```

    apply(cases bq1 = [])
    apply(simp)
    apply(auto simp add: abc2)
  done
qed
qed

```

**lemma** *rank-meldUniq*:  
 $\llbracket \text{rank-invar } bq1; \text{rank-invar } bq2 \rrbracket \implies \text{rank-invar } (\text{meldUniq } bq1 \ bq2)$   
**by** (*simp only: rank-invar-meldUniq-strong*)

**lemma** *rank-meld*:  
 $\llbracket \text{rank-skew-invar } q1; \text{rank-skew-invar } q2 \rrbracket \implies \text{rank-skew-invar } (\text{meld } q1 \ q2)$   
**by** (*simp only: meld-def rank-meldUniq rank-uniqify rank-invar-rank-skew*)

**theorem** *meld-invar*:  
 $\llbracket \text{invar } bq1; \text{invar } bq2 \rrbracket$   
 $\implies \text{invar } (\text{meld } bq1 \ bq2)$   
**by** (*metis meld-queue-invar rank-meld invar-def*)

**theorem** *meld-correct*:  
**assumes** *I*: *invar* *q invar* *q'*  
**shows**  
*invar* (*meld* *q q'*)  
*queue-to-multiset* (*meld* *q q'*) = *queue-to-multiset* *q* + *queue-to-multiset* *q'*  
**using** *meld-invar*[*of* *q q'*] *meld-mset*[*of* *q q'*] *I*  
**unfolding** *invar-def* **by** *simp-all*

## 2.2.4 Find Minimal Element

Find the tree containing the minimal element.

**fun** *getMinTree* :: ('e, 'a::linorder) *SkewBinomialQueue*  $\Rightarrow$   
('e, 'a) *SkewBinomialTree* **where**  
*getMinTree* [t] = t |  
*getMinTree* (t#bq) =  
(if *prio* t  $\leq$  *prio* (*getMinTree* bq)  
then t  
else (*getMinTree* bq))

Find the minimal Element in the queue.

**definition** *findMin* :: ('e, 'a::linorder) *SkewBinomialQueue*  $\Rightarrow$  ('e  $\times$  'a) **where**  
*findMin* bq = (let *min* = *getMinTree* bq in (val *min*, *prio* *min*))

**lemma** *mintree-exists*: (bq  $\neq$  []) = (*getMinTree* bq  $\in$  set bq)

**proof** (*induct* bq)  
**case** *Nil*  
**then show** ?*case* **by** *simp*

```

next
  case (Cons - bq)
  then show ?case by (cases bq) simp-all
qed

lemma treehead-in-multiset:
  t ∈ set bq ⇒ (val t, prio t) ∈# (queue-to-multiset bq)
  by (induct bq, simp, cases t, auto)

lemma heap-ordered-single:
  heap-ordered t = (∀ x ∈ set-mset (tree-to-multiset t). prio t ≤ snd x)
  by (cases t) auto

lemma getMinTree-cons:
  prio (getMinTree (y # x # xs)) ≤ prio (getMinTree (x # xs))
  by (induct xs rule: getMinTree.induct) simp-all

lemma getMinTree-min-tree: t ∈ set bq ⇒ prio (getMinTree bq) ≤ prio t
  by (induct bq arbitrary: t rule: getMinTree.induct) (simp, fastforce, simp)

lemma getMinTree-min-prio:
  assumes queue-invar bq
  and y ∈ set-mset (queue-to-multiset bq)
  shows prio (getMinTree bq) ≤ snd y
proof -
  from assms have bq ≠ [] by (cases bq) simp-all
  with assms have ∃ t ∈ set bq. (y ∈ set-mset (tree-to-multiset t))
  proof (induct bq)
    case Nil
    then show ?case by simp
  next
    case (Cons a bq)
    then show ?case
    apply (cases y ∈ set-mset (tree-to-multiset a))
    apply simp
    apply (cases bq)
    apply simp-all
    done
  qed
  from this obtain t where O:
    t ∈ set bq
    y ∈ set-mset ((tree-to-multiset t)) by blast
  obtain e a r ts where [simp]: t = (Node e a r ts) by (cases t) blast
  from O assms(1) have inv: tree-invar t by simp
  from tree-invar-heap-ordered[OF inv] heap-ordered.simps[of e a r ts] O
  have prio t ≤ snd y by auto
  with getMinTree-min-tree[OF O(1)] show ?thesis by simp
qed

```



**lemma** *findMin-mset*:  
**assumes** *I*: *queue-invar* *q*  
**assumes** *NE*: *q ≠ Nil*  
**shows** *findMin* *q* ∈# *queue-to-multiset* *q*  
 $\forall y \in \text{set-mset } (\text{queue-to-multiset } q). \text{snd } (\text{findMin } q) \leq \text{snd } y$   
**proof** –  
**from** *NE* **have** *getMinTree* *q* ∈ *set* *q* **by** (*simp* *only*: *mintree-exists*)  
**thus** *findMin* *q* ∈# *queue-to-multiset* *q*  
**by** (*simp* *add*: *treehead-in-multiset* *findMin-def* *Let-def*)  
**show**  $\forall y \in \text{set-mset } (\text{queue-to-multiset } q). \text{snd } (\text{findMin } q) \leq \text{snd } y$   
**by** (*simp* *add*: *getMinTree-min-prio* *findMin-def* *Let-def* *NE* *I*)  
**qed**

**theorem** *findMin-correct*:  
**assumes** *I*: *invar* *q*  
**assumes** *NE*: *q ≠ Nil*  
**shows** *findMin* *q* ∈# *queue-to-multiset* *q*  
 $\forall y \in \text{set-mset } (\text{queue-to-multiset } q). \text{snd } (\text{findMin } q) \leq \text{snd } y$   
**using** *I* *NE* *findMin-mset*  
**unfolding** *invar-def* **by** *auto*

## 2.2.5 Delete Minimal Element

Insert the roots of a given queue into an other queue.

**fun** *insertList* ::  
 $(\text{'e}, \text{'a}::\text{linorder}) \text{ SkewBinomialQueue} \Rightarrow (\text{'e}, \text{'a}) \text{ SkewBinomialQueue} \Rightarrow$   
 $(\text{'e}, \text{'a}) \text{ SkewBinomialQueue}$  **where**  
*insertList* [] *tbq* = *tbq* |  
*insertList* (*t*#*bq*) *tbq* = *insertList* *bq* (*insert* (*val* *t*) (*prio* *t*) *tbq*)

Remove the first tree, which has the priority *a* within his root.

**fun** *remove1Prio* ::  $\text{'a} \Rightarrow (\text{'e}, \text{'a}::\text{linorder}) \text{ SkewBinomialQueue} \Rightarrow$   
 $(\text{'e}, \text{'a}) \text{ SkewBinomialQueue}$  **where**  
*remove1Prio* *a* [] = [] |  
*remove1Prio* *a* (*t*#*bq*) =  
*(if* (*prio* *t*) = *a* *then* *bq* *else* *t* # (*remove1Prio* *a* *bq*))

**lemma** *remove1Prio-remove1* [*simp*]:  
*remove1Prio* (*prio* (*getMinTree* *bq*)) *bq* = *remove1* (*getMinTree* *bq*) *bq*

**proof** (*induct* *bq*)  
**case** *Nil* **thus** ?*case* **by** *simp*  
**next**  
**case** (*Cons* *t* *bq*)  
**note** *iv* = *Cons*  
**thus** ?*case*  
**proof** (*cases* *t* = *getMinTree* (*t* # *bq*))  
**case** *True*  
**with** *iv* **show** ?*thesis* **by** *simp*  
**next**

```

case False
hence ne: bq ≠ [] by auto
with False have down: getMinTree (t # bq) = getMinTree bq
  by (induct bq rule: getMinTree.induct) auto
from ne False have prio t ≠ prio (getMinTree bq)
  by (induct bq rule: getMinTree.induct) auto
with down in False ne show ?thesis by simp
qed
qed

```

Return the queue without the minimal element found by findMin

```

definition deleteMin :: ('e, 'a::linorder) SkewBinomialQueue ⇒
  ('e, 'a) SkewBinomialQueue where
  deleteMin bq = (let min = getMinTree bq in insertList
    (filter (λ t. rank t = 0) (children min))
    (meld (rev (filter (λ t. rank t > 0) (children min))))
    (remove1Prio (prio min) bq))

```

```

lemma invar-rev[simp]: queue-invar (rev q) ↔ queue-invar q
by (unfold queue-invar-def) simp

```

```

lemma invar-remove1: queue-invar q ⇒ queue-invar (remove1 t q)
by (unfold queue-invar-def) (auto)

```

```

lemma mset-rev: queue-to-multiset (rev q) = queue-to-multiset q
by (induct q) (auto simp add: union-ac)

```

```

lemma in-set-subset: t ∈ set q ⇒ tree-to-multiset t ⊆# queue-to-multiset q
proof (induct q)

```

```

  case Nil
  then show ?case by simp
next
  case (Cons a q)
  show ?case
  proof (cases t = a)
    case True
    then show ?thesis by simp
  next
  case False
  with Cons have t-in-q: t ∈ set q by simp
  have queue-to-multiset q ⊆# queue-to-multiset (a # q)
  by simp
  from subset-mset.order-trans[OF Cons(1)[OF t-in-q] this] show ?thesis .
qed
qed

```

```

lemma mset-remove1: t ∈ set q ⇒
  queue-to-multiset (remove1 t q) =
  queue-to-multiset q - tree-to-multiset t

```

**by** (*induct* *q*) (*auto simp: in-set-subset*)

**lemma** *invar-children'*:

**assumes** *tree-invar* *t*

**shows** *queue-invar* (*children* *t*)

**proof** (*cases* *t*)

**case** (*Node* *e* *a* *nat* *list*)

**with** *assms* **have** *inv: tree-invar* (*Node* *e* *a* *nat* *list*) **by** *simp*

**from** *Node invar-children*[*OF inv*] **show** *?thesis* **by** *simp*

**qed**

**lemma** *invar-filter*: *queue-invar* *q*  $\implies$  *queue-invar* (*filter* *f* *q*)

**by** (*unfold* *queue-invar-def*) *simp*

**lemma** *insertList-queue-invar*: *queue-invar* *q*  $\implies$  *queue-invar* (*insertList* *ts* *q*)

**proof** (*induct* *ts* *arbitrary: q*)

**case** *Nil*

**then** **show** *?case* **by** *simp*

**next**

**case** (*Cons* *a* *q*)

**note** *inv-insert* = *insert-queue-invar*[*OF Cons*(2), *of val a prio a*]

**from** *Cons*(1)[*OF inv-insert*] **show** *?case* **by** *simp*

**qed**

**lemma** *deleteMin-queue-invar*:

$\llbracket$ *queue-invar* *q*; *queue-to-multiset* *q*  $\neq$   $\{\#\}$  $\rrbracket \implies$

*queue-invar* (*deleteMin* *q*)

**unfolding** *deleteMin-def* *Let-def*

**proof** *goal-cases*

**case** *prems: 1*

**from** *prems*(2) **have** *q-ne*: *q*  $\neq$   $\llbracket$  $\rrbracket$  **by** *auto*

**with** *prems*(1) *mintree-exists*[*of q*]

**have** *inv-min*: *tree-invar* (*getMinTree* *q*) **by** *simp*

**note** *inv-rem* = *invar-remove1*[*OF prems*(1), *of getMinTree* *q*]

**note** *inv-children* = *invar-children'*[*OF inv-min*]

**note** *inv-filter* = *invar-filter*[*OF inv-children*, *of*  $\lambda t. 0 < \text{rank } t$ ]

**note** *inv-rev* = *iffD2*[*OF invar-rev inv-filter*]

**note** *inv-meld* = *meld-queue-invar*[*OF inv-rev inv-rem*]

**note** *inv-ins* =

*insertList-queue-invar*[*OF inv-meld*,

*of* [*t*  $\leftarrow$  *children* (*getMinTree* *q*). *rank* *t* = 0]]

**then** **show** *?case* **by** *simp*

**qed**

**lemma** *mset-children*: *queue-to-multiset* (*children* *t*) =

*tree-to-multiset* *t* -  $\{\#\}$  (*val* *t*, *prio* *t*)  $\#\}$

**by**(*cases* *t*, *auto*)

**lemma** *mset-insertList*:

$\llbracket \forall t \in \text{set } ts. \text{rank } t = 0 \wedge \text{children } t = [] ; \text{queue-invar } q \rrbracket \implies$   
 $\text{queue-to-multiset } (\text{insertList } ts \ q) =$   
 $\text{queue-to-multiset } ts + \text{queue-to-multiset } q$   
**proof** (*induct ts arbitrary: q*)  
**case** *Nil*  
**then show** *?case by simp*  
**next**  
**case** (*Cons a ts*)  
**from** *Cons(2) have ball-ts:  $\forall t \in \text{set } ts. \text{rank } t = 0 \wedge \text{children } t = []$  by simp*  
**note** *inv-insert = insert-queue-invar[OF Cons(3), of val a prio a]*  
**note** *iv = Cons(1)[OF ball-ts inv-insert]*  
**from** *Cons(2) have mset-a: tree-to-multiset a =  $\{\# (\text{val } a, \text{prio } a)\#$*   
**by** (*cases a simp*)  
**note** *insert-mset[OF Cons(3), of val a prio a]*  
**with** *mset-a iv show ?case by (simp add: union-ac)*  
**qed**

**lemma** *mset-filter: (queue-to-multiset  $[t \leftarrow q . \text{rank } t = 0]$ ) +*  
*queue-to-multiset  $[t \leftarrow q . 0 < \text{rank } t]$  =*  
*queue-to-multiset q*  
**by** (*induct q (auto simp add: union-ac)*)

**lemma** *deleteMin-mset:*  
**assumes** *queue-invar q*  
**and** *queue-to-multiset q  $\neq \{\#\}$*   
**shows** *queue-to-multiset (deleteMin q) = queue-to-multiset q -  $\{\# (\text{findMin } q)$*   
 *$\#\}$*   
**proof** –  
**from** *assms(2) have q-ne: q  $\neq []$  by auto*  
**with** *mintree-exists[of q]*  
**have** *min-in-q: getMinTree q  $\in \text{set } q$  by simp*  
**with** *assms(1) have inv-min: tree-invar (getMinTree q) by simp*  
**note** *inv-rem = invar-remove1[OF assms(1), of getMinTree q]*  
**note** *inv-children = invar-children'[OF inv-min]*  
**note** *inv-filter = invar-filter[OF inv-children, of  $\lambda t. 0 < \text{rank } t$ ]*  
**note** *inv-rev = iffD2[OF invar-rev inv-filter]*  
**note** *inv-meld = meld-queue-invar[OF inv-rev inv-rem]*  
**note** *mset-rem = mset-remove1[OF min-in-q]*  
**note** *mset-rev = mset-rev[of  $[t \leftarrow \text{children } (\text{getMinTree } q). 0 < \text{rank } t]$ ]*  
**note** *mset-meld = meld-mset[OF inv-rev inv-rem]*  
**note** *mset-children = mset-children[of getMinTree q]*  
**thm** *mset-insertList[of  $[t \leftarrow \text{children } (\text{getMinTree } q) .$*   
*rank t = 0]]*  
**have**  $\llbracket \text{tree-invar } t; \text{rank } t = 0 \rrbracket \implies \text{children } t = []$  **for** *t*  
**by** (*cases t simp*)  
**with** *inv-children*  
**have** *ball-min:  $\forall t \in \text{set } [t \leftarrow \text{children } (\text{getMinTree } q). \text{rank } t = 0].$*   
*rank t = 0  $\wedge$  children t = [] by (unfold queue-invar-def) auto*  
**note** *mset-insertList = mset-insertList[OF ball-min inv-meld]*

```

note mset-filter = mset-filter[of children (getMinTree q)]
let ?Q = queue-to-multiset q
let ?MT = tree-to-multiset (getMinTree q)
from q-ne have head-subset-min:
  {# (val (getMinTree q), prio (getMinTree q)) #}  $\subseteq$  # ?MT
  by(cases getMinTree q simp)
note min-subset-q = in-set-subset[OF min-in-q]
from mset-insertList mset-meld mset-rev mset-rem mset-filter mset-children
  multiset-diff-union-assoc[OF head-subset-min, of ?Q - ?MT]
  mset-subset-eq-multiset-union-diff-commute[OF min-subset-q, of ?MT]
show ?thesis
  by (auto simp add: deleteMin-def Let-def union-ac findMin-def)
qed

lemma rank-insertList: rank-skew-invar q  $\implies$  rank-skew-invar (insertList ts q)
  by (induct ts arbitrary: q (simp-all add: insert-rank-invar))

lemma insertList-invar: invar q  $\implies$  invar (insertList ts q)
proof (induct ts arbitrary: q)
  case Nil
  then show ?case by simp
next
  case (Cons a q)
  show ?case
    apply (unfold insertList.simps)
  proof goal-cases
    case 1
    from Cons(2) insert-rank-invar[of q val a prio a]
    have a1: rank-skew-invar (insert (val a) (prio a) q)
      by (simp add: invar-def)
    from Cons(2) insert-queue-invar[of q val a prio a]
    have a2: queue-invar (insert (val a) (prio a) q) by (simp add: invar-def)
    from a1 a2 have invar (insert (val a) (prio a) q) by (simp add: invar-def)
    with Cons(1)[of (insert (val a) (prio a) q)] show ?case .
  qed
qed

lemma children-rank-less:
  assumes tree-invar t
  shows  $\forall t' \in \text{set } (\text{children } t). \text{rank } t' < \text{rank } t$ 
proof (cases t)
  case (Node e a nat list)
  with assms show ?thesis
  proof (induct nat arbitrary: t e a list)
    case 0
    then show ?case by simp
  next
  case (Suc nat)
  then obtain e1 a1 ts1 e2 a2 ts2 e' a' where

```

$O$ :  $\text{tree-invar } (\text{Node } e1 \ a1 \ \text{nat } ts1) \ \text{tree-invar } (\text{Node } e2 \ a2 \ \text{nat } ts2)$   
 $t = \text{link } (\text{Node } e1 \ a1 \ \text{nat } ts1) \ (\text{Node } e2 \ a2 \ \text{nat } ts2)$   
 $\vee t = \text{skewlink } e' \ a' \ (\text{Node } e1 \ a1 \ \text{nat } ts1) \ (\text{Node } e2 \ a2 \ \text{nat } ts2)$   
**by** (*simp only: tree-invar.simps*) *blast*  
**hence** *ch-id*:  
 $\text{children } t = (\text{if } a1 \leq a2 \ \text{then } (\text{Node } e2 \ a2 \ \text{nat } ts2) \# ts1$   
 $\quad \text{else } (\text{Node } e1 \ a1 \ \text{nat } ts1) \# ts2) \vee$   
 $\text{children } t =$   
 $\quad (\text{if } a' \leq a1 \ \wedge \ a' \leq a2 \ \text{then } [(\text{Node } e1 \ a1 \ \text{nat } ts1), (\text{Node } e2 \ a2 \ \text{nat } ts2)]$   
 $\quad \text{else } (\text{if } a1 \leq a2 \ \text{then } (\text{Node } e' \ a' \ 0 \ []) \# (\text{Node } e2 \ a2 \ \text{nat } ts2) \# ts1$   
 $\quad \text{else } (\text{Node } e' \ a' \ 0 \ []) \# (\text{Node } e1 \ a1 \ \text{nat } ts1) \# ts2))$   
**by** *auto*  
**from**  $O \ \text{Suc}(1)$ [*of Node e1 a1 nat ts1 e1 a1 ts1*]  
**have**  $p1: \forall t' \in \text{set } ((\text{Node } e2 \ a2 \ \text{nat } ts2) \# ts1). \ \text{rank } t' < \text{Suc } \text{nat}$  **by** *auto*  
**from**  $O \ \text{Suc}(1)$ [*of Node e2 a2 nat ts2 e2 a2 ts2*]  
**have**  $p2: \forall t' \in \text{set } ((\text{Node } e1 \ a1 \ \text{nat } ts1) \# ts2). \ \text{rank } t' < \text{Suc } \text{nat}$  **by** *auto*  
**from**  $O$  **have**  
 $p3: \forall t' \in \text{set } [(\text{Node } e1 \ a1 \ \text{nat } ts1), (\text{Node } e2 \ a2 \ \text{nat } ts2)].$   
 $\quad \text{rank } t' < \text{Suc } \text{nat}$  **by** *simp*  
**from**  $O \ \text{Suc}(1)$ [*of Node e1 a1 nat ts1 e1 a1 ts1*]  
**have**  
 $p4: \forall t' \in \text{set } ((\text{Node } e' \ a' \ 0 \ []) \# (\text{Node } e2 \ a2 \ \text{nat } ts2) \# ts1).$   
 $\quad \text{rank } t' < \text{Suc } \text{nat}$  **by** *auto*  
**from**  $O \ \text{Suc}(1)$ [*of Node e2 a2 nat ts2 e2 a2 ts2*]  
**have**  $p5:$   
 $\quad \forall t' \in \text{set } ((\text{Node } e' \ a' \ 0 \ []) \# (\text{Node } e1 \ a1 \ \text{nat } ts1) \# ts2).$   
 $\quad \text{rank } t' < \text{Suc } \text{nat}$  **by** *auto*  
**from**  $\text{Suc}(3)$   $p1 \ p2 \ p3 \ p4 \ p5$  *ch-id* **show** *?case*  
**by**(*cases children t = (if a1 ≤ a2 then Node e2 a2 nat ts2 # ts1*  
 $\quad \text{else } \text{Node } e1 \ a1 \ \text{nat } ts1 \# ts2)$ ) *simp-all*  
**qed**  
**qed**

**lemma** *strong-rev-children*:  
**assumes** *tree-invar t*  
**shows** *invar (rev [t ← children t. 0 < rank t])*  
**proof** (*cases t*)  
**case** (*Node e a nat list*)  
**with** *assms* **show** *?thesis*  
**proof** (*induct nat arbitrary: t e a list*)  
**case**  $0$   
**then** **show** *?case* **by** (*simp add: invar-def*)  
**next**  
**case** (*Suc nat*)  
**show** *?case*  
**proof** (*cases nat*)  
**case**  $0$   
**with**  $\text{Suc}$  **obtain**  $e1 \ a1 \ e2 \ a2 \ e' \ a'$  **where**  
 $O: \text{tree-invar } (\text{Node } e1 \ a1 \ 0 \ []) \ \text{tree-invar } (\text{Node } e2 \ a2 \ 0 \ [])$

```

    t = link (Node e1 a1 0 []) (Node e2 a2 0 [])
    ∨ t = skewlink e' a' (Node e1 a1 0 []) (Node e2 a2 0 [])
    by (simp only: tree-invar.simps) blast
  hence [t ← children t. 0 < rank t] = [] by auto
  then show ?thesis by (simp add: invar-def)
next
case Suc': (Suc n)
from Suc obtain e1 a1 ts1 e2 a2 ts2 e' a' where
  O: tree-invar (Node e1 a1 nat ts1) tree-invar (Node e2 a2 nat ts2)
  t = link (Node e1 a1 nat ts1) (Node e2 a2 nat ts2)
  ∨ t = skewlink e' a' (Node e1 a1 nat ts1) (Node e2 a2 nat ts2)
  by (simp only: tree-invar.simps) blast
hence ch-id:
  children t = (if a1 ≤ a2 then
    (Node e2 a2 nat ts2)#ts1
  else (Node e1 a1 nat ts1)#ts2)
  ∨
  children t = (if a' ≤ a1 ∧ a' ≤ a2 then
    [(Node e1 a1 nat ts1), (Node e2 a2 nat ts2)]
  else (if a1 ≤ a2 then
    (Node e' a' 0 []) # (Node e2 a2 nat ts2) # ts1
  else (Node e' a' 0 []) # (Node e1 a1 nat ts1) # ts2))
  by auto
from O Suc(1)[of Node e1 a1 nat ts1 e1 a1 ts1] have
  rev-ts1: invar (rev [t ← ts1. 0 < rank t]) by simp
from O children-rank-less[of Node e1 a1 nat ts1] have
  ∨ t ∈ set (rev [t ← ts1. 0 < rank t]). rank t < rank (Node e2 a2 nat ts2)
  by simp
with O rev-ts1
  invar-app-single[of rev [t ← ts1. 0 < rank t]
    Node e2 a2 nat ts2]
have
  invar (rev ((Node e2 a2 nat ts2) # [t ← ts1. 0 < rank t]))
  by simp
with Suc' have p1: invar (rev [t ← ((Node e2 a2 nat ts2) # ts1). 0 < rank
t])
  by simp
from O Suc(1)[of Node e2 a2 nat ts2 e2 a2 ts2]
have rev-ts2: invar (rev [t ← ts2. 0 < rank t]) by simp
from O children-rank-less[of Node e2 a2 nat ts2]
have ∨ t ∈ set (rev [t ← ts2. 0 < rank t]).
  rank t < rank (Node e1 a1 nat ts1) by simp
with O rev-ts2 invar-app-single[of rev [t ← ts2. 0 < rank t]
  Node e1 a1 nat ts1]
have invar (rev [t ← ts2. 0 < rank t] @ [Node e1 a1 nat ts1])
  by simp
with Suc' have p2: invar (rev [t ← ((Node e1 a1 nat ts1) # ts2). 0 < rank
t])
  by simp

```

```

from  $O(1-2)$ 
have  $p3$ : invar (rev (filter ( $\lambda t. 0 < \text{rank } t$ )
  [(Node  $e1$   $a1$  nat  $ts1$ ), (Node  $e2$   $a2$  nat  $ts2$ )]))
  by (simp add: invar-def)
from  $p1$  have  $p4$ : invar (rev
  [ $t \leftarrow ((\text{Node } e' a' 0 []) \# (\text{Node } e2 a2 \text{ nat } ts2) \# ts1). 0 < \text{rank } t$ ])
  by simp
from  $p2$  have  $p5$ : invar (rev
  [ $t \leftarrow ((\text{Node } e' a' 0 []) \# (\text{Node } e1 a1 \text{ nat } ts1) \# ts2). 0 < \text{rank } t$ ])
  by simp
from  $p1$   $p2$   $p3$   $p4$   $p5$  ch-id show
  invar (rev [ $t \leftarrow \text{children } t . 0 < \text{rank } t$ ])
  by (cases children  $t = (\text{if } a1 \leq a2 \text{ then } (\text{Node } e2 a2 \text{ nat } ts2) \# ts1$ 
    else (Node  $e1 a1 \text{ nat } ts1$ ) \#  $ts2$ )) metis+

  qed
qed
qed

```

```

lemma first-less: rank-invar ( $t \# bq$ )  $\implies \forall t' \in \text{set } bq. \text{rank } t < \text{rank } t'$ 
apply (induct  $bq$  arbitrary:  $t$ )
apply (simp)
apply (metis List.set-simps(2) insert-iff not-le-imp-less
  not-less-iff-gr-or-eq order-less-le-trans rank-invar.simps(3)
  rank-invar-cons-down)
done

```

```

lemma first-less-eq:
  rank-skew-invar ( $t \# bq$ )  $\implies \forall t' \in \text{set } bq. \text{rank } t \leq \text{rank } t'$ 
apply (induct  $bq$  arbitrary:  $t$ )
apply (simp)
apply (metis List.set-simps(2) insert-iff le-trans
  rank-invar-rank-skew rank-skew-invar.simps(3) rank-skew-rank-invar)
done

```

```

lemma remove1-tail-invar: tail-invar  $bq \implies \text{tail-invar } (\text{remove1 } t \text{ } bq)$ 
proof (induct  $bq$  arbitrary:  $t$ )
  case Nil
  then show ?case by simp
next
  case (Cons  $a$   $bq$ )
  show ?case
  proof (cases  $t = a$ )
    case True
    from Cons(2) have tail-invar  $bq$  by (rule tail-invar-cons-down)
    with True show ?thesis by simp
  next
  case False
  from Cons(2) have tail-invar  $bq$  by (rule tail-invar-cons-down)
  with Cons(1)[of  $t$ ] have  $si1$ : tail-invar (remove1  $t$   $bq$ ) .

```



```

from False have tail-invar (remove1 t (a # bq)) = tail-invar (a # (remove1 t
bq))
  by simp
show ?thesis
proof (cases remove1 t bq)
  case Nil
    with si1 Cons(2) False show ?thesis by (simp add: tail-invar-def)
  next
    case Cons': (Cons aa list)
      from Cons(2) have tree-invar a by (simp add: tail-invar-def)
      from Cons(2) first-less[of a bq]
      have  $\forall t \in \text{set } (\text{remove1 } t \text{ } bq). \text{rank } a < \text{rank } t$ 
        by (metis notin-set-remove1 tail-invar-def)
      with Cons' have rank a < rank aa by simp
      with si1 Cons(2) False Cons' tail-invar-cons-up[of aa list a] show ?thesis
        by (simp add: tail-invar-def)
    qed
  qed
qed

```

```

lemma invar-cons-down: invar (t # bq)  $\implies$  invar bq
  by (metis rank-invar-rank-skew tail-invar-def
    invar-def invar-tail-invar)

```

```

lemma remove1-invar: invar bq  $\implies$  invar (remove1 t bq)
proof (induct bq arbitrary: t)
  case Nil
    then show ?case by simp
  next
    case (Cons a bq)
      show ?case
      proof (cases t = a)
        case True
          from Cons(2) have invar bq by (rule invar-cons-down)
          with True show ?thesis by simp
        next
          case False
            from Cons(2) have invar bq by (rule invar-cons-down)
            with Cons(1)[of t] have si1: invar (remove1 t bq) .
            from False have invar (remove1 t (a # bq)) = invar (a # (remove1 t bq))
              by simp
            show ?thesis
            proof (cases remove1 t bq)
              case Nil
                with si1 Cons(2) False show ?thesis by (simp add: invar-def)
              next
                case Cons': (Cons aa list)
                  from Cons(2) have ti: tree-invar a by (simp add: invar-def)
                  from Cons(2) have sbq: tail-invar bq by (metis invar-tail-invar)

```

**hence** *srm*: *tail-invar* (*remove1 t bq*) **by** (*metis remove1-tail-invar*)  
**from** *Cons*(2) *first-less-eq*[of *a bq*]  
**have**  $\forall t \in \text{set } (\text{remove1 } t \text{ } bq). \text{rank } a \leq \text{rank } t$   
**by** (*metis notin-set-remove1 invar-def*)  
**with** *Cons'* **have**  $\text{rank } a \leq \text{rank } aa$  **by** *simp*  
**with** *si1 Cons*(2) *False Cons' ti srm tail-invar-cons-up-invar*[of *aa list a*]  
**show** *?thesis* **by** *simp*  
**qed**  
**qed**  
**qed**

**lemma** *deleteMin-invar*:

**assumes** *invar bq*  
**and**  $bq \neq []$   
**shows** *invar* (*deleteMin bq*)  
**proof** –  
**have** *eq*: *invar* (*deleteMin bq*) =  
*invar* (*insertList*  
(*filter* ( $\lambda t. \text{rank } t = 0$ ) (*children* (*getMinTree bq*)))  
(*meld* (*rev* (*filter* ( $\lambda t. \text{rank } t > 0$ ) (*children* (*getMinTree bq*)))  
(*remove1* (*getMinTree bq*) *bq*)))  
**by** (*simp add: deleteMin-def Let-def*)  
**from** *assms mintree-exists*[of *bq*] **have** *ti*: *tree-invar* (*getMinTree bq*)  
**by** (*simp add: invar-def queue-invar-def del: queue-invar-simps*)  
**with** *strong-rev-children*[of *getMinTree bq*] **have**  
*m1*: *invar* (*rev* [*t*  $\leftarrow$  *children* (*getMinTree bq*).  $0 < \text{rank } t$ ]) .  
**from** *remove1-invar*[of *bq getMinTree bq*] *assms*(1)  
**have** *m2*: *invar* (*remove1* (*getMinTree bq*) *bq*) .  
**from** *meld-invar*[of *rev* [*t*  $\leftarrow$  *children* (*getMinTree bq*).  $0 < \text{rank } t$ ]  
*remove1* (*getMinTree bq*) *bq*] *m1 m2*  
**have** *invar* (*meld* (*rev* [*t*  $\leftarrow$  *children* (*getMinTree bq*).  $0 < \text{rank } t$ ]  
(*remove1* (*getMinTree bq*) *bq*)) .  
**with** *insertList-invar*[of  
(*meld* (*rev* [*t*  $\leftarrow$  *children* (*getMinTree bq*) .  $0 < \text{rank } t$ ])  
(*remove1* (*getMinTree bq*) *bq*))  
[*t*  $\leftarrow$  *children* (*getMinTree bq*) .  $\text{rank } t = 0$ ]]  
**have** *invar*  
(*insertList*  
[*t*  $\leftarrow$  *children* (*getMinTree bq*) .  $\text{rank } t = 0$ ]  
(*meld* (*rev* [*t*  $\leftarrow$  *children* (*getMinTree bq*) .  $0 < \text{rank } t$ ]  
(*remove1* (*getMinTree bq*) *bq*))) .  
**with** *eq* **show** *?thesis ..*  
**qed**

**theorem** *deleteMin-correct*:

**assumes** *I*: *invar q*  
**and** *NE*:  $q \neq \text{Nil}$   
**shows** *invar* (*deleteMin q*)  
**and** *queue-to-multiset* (*deleteMin q*) = *queue-to-multiset q* – {*#findMin q*}

```

apply (rule deleteMin-invar[OF I NE])
using deleteMin-mset[of q] I NE
unfolding invar-def
apply (auto simp add: empty-correct)
done

```

```

lemmas [simp del] = insert.simps

```

```

end

```

```

interpretation SkewBinomialHeapStruct: SkewBinomialHeapStruct-loc .

```

## 2.3 Bootstrapping

In this section, we implement datastructural bootstrapping, to reduce the complexity of meld-operations to  $O(1)$ . The bootstrapping also contains a *global root*, caching the minimal element of the queue, and thus also reducing the complexity of findMin-operations to  $O(1)$ .

Bootstrapping adds one more level of recursion: An *element* is an entry and a priority queues of elements.

In the original paper on skew binomial queues [1], higher order functors and recursive structures are used to elegantly implement bootstrapped heaps on top of ordinary heaps. However, such concepts are not supported in Isabelle/HOL, nor in Standard ML. Hence we have to use the „much less clean” [1] alternative: We manually specialize the heap datastructure, and re-implement the functions on the specialized data structure.

The correctness proofs are done by defining a mapping from teh specialized to the original data structure, and reusing the correctness statements of the original data structure.

### 2.3.1 Auxiliary

We first have to state some auxiliary lemmas and functions, mainly about multisets.

Finding the preimage of an element

```

lemma in-image-msetE:
  assumes  $x \in \# \text{image-mset } f \ M$ 
  obtains  $y$  where  $y \in \# M$   $x = f \ y$ 
  using assms
  apply (induct M)
  apply simp
  apply (force split: if-split-asm)

```

**done**

Very special lemma for images multisets of pairs, where the second component is a function of the first component

**lemma** *mset-image-fst-dep-pair-diff-split*:

$(\forall e a. (e,a) \in \# M \longrightarrow a = f e) \implies$

$image\text{-}mset\text{-}fst (M - \{\#(e, f e)\}) = image\text{-}mset\text{-}fst M - \{\#e\}$

**proof** (*induct M*)

**case empty thus** ?*case* **by** *auto*

**next**

**case** (*add x M*)

**then obtain** *e'* **where** [*simp*]:  $x = (e', f e')$

**apply** (*cases x*)

**apply** (*force*)

**done**

**from** *add.prem*s **have**  $\forall e a. (e, a) \in \# M \longrightarrow a = f e$  **by** *simp*

**with** *add.hyps* **have**

*IH*:  $image\text{-}mset\text{-}fst (M - \{\#(e, f e)\}) = image\text{-}mset\text{-}fst M - \{\#e\}$

**by** *auto*

**show** ?*case* **proof** (*cases e = e'*)

**case** *True*

**thus** ?*thesis* **by** (*simp*)

**next**

**case** *False*

**thus** ?*thesis*

**by** (*simp add: IH*)

**qed**

**qed**

**locale** *Bootstrapped*

**begin**

### 2.3.2 Datatype

We manually specialize the binomial tree to contain elements, that, in, turn, may contain trees. Note that we specify nodes without explicit priority, as the priority is contained in the elements stored in the nodes.

**datatype** (*'e, 'a*) *BsSkewBinomialTree* =

*BsNode* (*val*: (*'e, 'a*::*linorder*) *BsSkewElem*)

(*rank*: *nat*) (*children*: (*'e, 'a*) *BsSkewBinomialTree list*)

**and**

(*'e, 'a*) *BsSkewElem* =

*Element 'e* (*eprio*: *'a*) (*'e, 'a*) *BsSkewBinomialTree list*

**type-synonym** ('e,'a) *BsSkewHeap* = *unit* + ('e,'a) *BsSkewElem*  
**type-synonym** ('e,'a) *BsSkewBinomialQueue* = ('e,'a) *BsSkewBinomialTree list*

### 2.3.3 Specialization Boilerplate

In this section, we re-define the functions on the specialized priority queues, and show their correctness. This is done by defining a mapping to original priority queues, and re-using the correctness lemmas proven there.

Mapping to original binomial trees and queues

**fun** *bsmapt* **where**

*bsmapt* (*BsNode e r q*) = *SkewBinomialHeapStruc.Node e (eprio e) r (map bsmapt q)*

**abbreviation** *bsmap* **where**

*bsmap q* == *map bsmapt q*

Invariant and mapping to multiset are defined via the mapping

**abbreviation** *invar q* == *SkewBinomialHeapStruc.invar (bsmap q)*

**abbreviation** *queue-to-multiset q*

== *image-mset fst (SkewBinomialHeapStruc.queue-to-multiset (bsmap q))*

**abbreviation** *tree-to-multiset t*

== *image-mset fst (SkewBinomialHeapStruc.tree-to-multiset (bsmapt t))*

**abbreviation** *queue-to-multiset-aux q*

== (*SkewBinomialHeapStruc.queue-to-multiset (bsmap q)*)

Now starts the re-implementation of the functions

**primrec** *prio* :: ('e, 'a::linorder) *BsSkewBinomialTree*  $\Rightarrow$  'a **where**

*prio* (*BsNode e r ts*) = *eprio e*

**lemma** *proj-xlate*:

*val t* = *SkewBinomialHeapStruc.val (bsmapt t)*

*prio t* = *SkewBinomialHeapStruc.prio (bsmapt t)*

*rank t* = *SkewBinomialHeapStruc.rank (bsmapt t)*

*bsmap (children t)* = *SkewBinomialHeapStruc.children (bsmapt t)*

*eprio (SkewBinomialHeapStruc.val (bsmapt t))*

= *SkewBinomialHeapStruc.prio (bsmapt t)*

**apply** (*case-tac* [!] *t*)

**apply** *auto*

**done**

**fun** *link* :: ('e, 'a::linorder) *BsSkewBinomialTree*

$\Rightarrow$  ('e, 'a) *BsSkewBinomialTree*  $\Rightarrow$

('e, 'a) *BsSkewBinomialTree* **where**

*link* (*BsNode e1 r1 ts1*) (*BsNode e2 r2 ts2*) =

(*if eprio e1*  $\leq$  *eprio e2*)

```

then (BsNode e1 (Suc r1) ((BsNode e2 r2 ts2)#ts1))
else (BsNode e2 (Suc r2) ((BsNode e1 r1 ts1)#ts2)))

```

Link two trees of rank  $r$  and a new element to a new tree of rank  $r + 1$

```

fun skewlink :: ('e,'a::linorder) BsSkewElem  $\Rightarrow$  ('e, 'a) BsSkewBinomialTree  $\Rightarrow$ 
('e, 'a) BsSkewBinomialTree  $\Rightarrow$  ('e, 'a) BsSkewBinomialTree where
skewlink e t t' = (if eprio e  $\leq$  (prio t)  $\wedge$  eprio e  $\leq$  (prio t'))
then (BsNode e (Suc (rank t)) [t,t'])
else (if (prio t)  $\leq$  (prio t'))
then
  BsNode (val t) (Suc (rank t)) (BsNode e 0 [] # t' # children t)
else
  BsNode (val t') (Suc (rank t')) (BsNode e 0 [] # t # children t'))

```

**lemma** link-xlate:

```

bsmapt (link t t') = SkewBinomialHeapStruc.link (bsmapt t) (bsmapt t')
bsmapt (skewlink e t t') =
  SkewBinomialHeapStruc.skewlink e (eprio e) (bsmapt t) (bsmapt t')
by (case-tac [] t, case-tac [] t') auto

```

**fun** ins :: ('e, 'a::linorder) BsSkewBinomialTree  $\Rightarrow$

```

('e, 'a) BsSkewBinomialQueue  $\Rightarrow$ 
('e, 'a) BsSkewBinomialQueue where
ins t [] = [t] |
ins t' (t # bq) =
  (if (rank t') < (rank t)
   then t' # t # bq
   else (if (rank t) < (rank t')
          then t # (ins t' bq)
          else ins (link t' t) bq))

```

**lemma** ins-xlate:

```

bsmap (ins t q) = SkewBinomialHeapStruc.ins (bsmapt t) (bsmap q)
by (induct q arbitrary: t) (auto simp add: proj-xlate link-xlate)

```

Insert an element with priority into a queue using skewlinks.

**fun** insert :: ('e,'a::linorder) BsSkewElem  $\Rightarrow$

```

('e, 'a) BsSkewBinomialQueue  $\Rightarrow$ 
('e, 'a) BsSkewBinomialQueue where
insert e [] = [BsNode e 0 []] |
insert e [t] = [BsNode e 0 [],t] |
insert e (t # t' # bq) =
  (if rank t  $\neq$  rank t'
   then (BsNode e 0 []) # t # t' # bq
   else (skewlink e t t') # bq)

```

**lemma** insert-xlate:

```

bsmap (insert e q) = SkewBinomialHeapStruc.insert e (eprio e) (bsmap q)

```

```

apply (cases (e,q) rule: insert.cases)
apply (auto simp add: proj-xlate link-xlate SkewBinomialHeapStruc.insert.simps)
done

```

**lemma** *insert-correct*:

**assumes** *I*: *invar* *q*

**shows**

*invar* (*insert* *e* *q*)

*queue-to-multiset* (*insert* *e* *q*) = *queue-to-multiset* *q* + {#(*e*)#}

**by** (*simp-all* add: *I* *SkewBinomialHeapStruc.insert-correct insert-xlate*)

**fun** *uniqify*

:: ('e, 'a::linorder) *BsSkewBinomialQueue*  $\Rightarrow$  ('e, 'a) *BsSkewBinomialQueue*

**where**

*uniqify* [] = [] |

*uniqify* (t#bq) = *ins* t bq

**fun** *meldUniq*

:: ('e, 'a::linorder) *BsSkewBinomialQueue*  $\Rightarrow$  ('e, 'a) *BsSkewBinomialQueue*  $\Rightarrow$

('e, 'a) *BsSkewBinomialQueue* **where**

*meldUniq* [] bq = bq |

*meldUniq* bq [] = bq |

*meldUniq* (t1#bq1) (t2#bq2) = (if rank t1 < rank t2

then t1 # (*meldUniq* bq1 (t2#bq2))

else (if rank t2 < rank t1

then t2 # (*meldUniq* (t1#bq1) bq2)

else *ins* (*link* t1 t2) (*meldUniq* bq1 bq2)))

**definition** *meld*

:: ('e, 'a::linorder) *BsSkewBinomialQueue*  $\Rightarrow$  ('e, 'a) *BsSkewBinomialQueue*  $\Rightarrow$

('e, 'a) *BsSkewBinomialQueue* **where**

*meld* bq1 bq2 = *meldUniq* (*uniqify* bq1) (*uniqify* bq2)

**lemma** *uniqify-xlate*:

*bsmap* (*uniqify* *q*) = *SkewBinomialHeapStruc.uniqify* (*bsmap* *q*)

**by** (*cases* *q*) (*simp-all* add: *ins-xlate*)

**lemma** *meldUniq-xlate*:

*bsmap* (*meldUniq* *q* *q'*) = *SkewBinomialHeapStruc.meldUniq* (*bsmap* *q*) (*bsmap* *q'*)

**apply** (*induct* *q* *q'* rule: *meldUniq.induct*)

**apply** (*auto simp* add: *link-xlate proj-xlate uniqify-xlate ins-xlate*)

**done**

**lemma** *meld-xlate*:

*bsmap* (*meld* *q* *q'*) = *SkewBinomialHeapStruc.meld* (*bsmap* *q*) (*bsmap* *q'*)

**by** (*simp* add: *meld-def meldUniq-xlate uniqify-xlate*

*SkewBinomialHeapStruc.meld-def*)

**lemma** *meld-correct*:

**assumes**  $I$ : *invar*  $q$  *invar*  $q'$

**shows**

*invar* (*meld*  $q$   $q'$ )

*queue-to-multiset* (*meld*  $q$   $q'$ ) = *queue-to-multiset*  $q$  + *queue-to-multiset*  $q'$

**by** (*simp-all* *add*:  $I$  *SkewBinomialHeapStruc.meld-correct* *meld-xlate*)

**fun** *insertList* ::

$(e, a::\text{linorder})$  *BsSkewBinomialQueue*  $\Rightarrow$   $(e, a)$  *BsSkewBinomialQueue*  $\Rightarrow$

$(e, a)$  *BsSkewBinomialQueue* **where**

*insertList* []  $tbq$  =  $tbq$  |

*insertList* ( $t\#bq$ )  $tbq$  = *insertList*  $bq$  (*insert* (*val*  $t$ )  $tbq$ )

**fun** *remove1Prio* ::  $a$   $\Rightarrow$   $(e, a::\text{linorder})$  *BsSkewBinomialQueue*  $\Rightarrow$

$(e, a)$  *BsSkewBinomialQueue* **where**

*remove1Prio*  $a$  [] = [] |

*remove1Prio*  $a$  ( $t\#bq$ ) =

(*if* (*prio*  $t$ ) =  $a$  *then*  $bq$  *else*  $t$  # (*remove1Prio*  $a$   $bq$ ))

**fun** *getMinTree* ::  $(e, a::\text{linorder})$  *BsSkewBinomialQueue*  $\Rightarrow$

$(e, a)$  *BsSkewBinomialTree* **where**

*getMinTree* [ $t$ ] =  $t$  |

*getMinTree* ( $t\#bq$ ) =

(*if* *prio*  $t$   $\leq$  *prio* (*getMinTree*  $bq$ ))

*then*  $t$

*else* (*getMinTree*  $bq$ ))

**definition** *findMin*

::  $(e, a::\text{linorder})$  *BsSkewBinomialQueue*  $\Rightarrow$   $(e, a)$  *BsSkewElem* **where**

*findMin*  $bq$  = *val* (*getMinTree*  $bq$ )

**definition** *deleteMin* ::  $(e, a::\text{linorder})$  *BsSkewBinomialQueue*  $\Rightarrow$

$(e, a)$  *BsSkewBinomialQueue* **where**

*deleteMin*  $bq$  = (*let*  $min$  = *getMinTree*  $bq$  *in* *insertList*

(*filter* ( $\lambda t.$  *rank*  $t$  = 0) (*children*  $min$ ))

(*meld* (*rev* (*filter* ( $\lambda t.$  *rank*  $t$  > 0) (*children*  $min$ )))

(*remove1Prio* (*prio*  $min$ )  $bq$ ))

**lemma** *insertList-xlate*:

*bsmap* (*insertList*  $q$   $q'$ )

= *SkewBinomialHeapStruc.insertList* (*bsmap*  $q$ ) (*bsmap*  $q'$ )

**apply** (*induct*  $q$  *arbitrary*:  $q'$ )

**apply** (*auto simp add*: *insert-xlate* *proj-xlate*)

**done**

**lemma** *remove1Prio-xlate*:

*bsmap* (*remove1Prio*  $a$   $q$ ) = *SkewBinomialHeapStruc.remove1Prio*  $a$  (*bsmap*  $q$ )

**by** (*induct*  $q$ ) (*auto simp add*: *proj-xlate*)



**lemma** *getMinTree-xlate*:  
 $q \neq [] \implies \text{bsmapt } (\text{getMinTree } q) = \text{SkewBinomialHeapStruc.getMinTree } (\text{bsmap } q)$   
**apply** (*induct*  $q$ )  
**apply** *simp*  
**apply** (*case-tac*  $q$ )  
**apply** (*auto simp add: proj-xlate*)  
**done**

**lemma** *findMin-xlate*:  
 $q \neq [] \implies \text{findMin } q = \text{fst } (\text{SkewBinomialHeapStruc.findMin } (\text{bsmap } q))$   
**apply** (*unfold findMin-def SkewBinomialHeapStruc.findMin-def*)  
**apply** (*simp add: proj-xlate Let-def getMinTree-xlate*)  
**done**

**lemma** *findMin-xlate-aux*:  
 $q \neq [] \implies (\text{findMin } q, \text{prio } (\text{findMin } q)) = (\text{SkewBinomialHeapStruc.findMin } (\text{bsmap } q))$   
**apply** (*unfold findMin-def SkewBinomialHeapStruc.findMin-def*)  
**apply** (*simp add: proj-xlate Let-def getMinTree-xlate*)  
**apply** (*induct*  $q$ )  
**apply** *simp*  
**apply** (*case-tac*  $q$ )  
**apply** (*auto simp add: proj-xlate*)  
**done**

**lemma** *bsmap-filter-xlate*:  
 $\text{bsmap } [x \leftarrow l . P (\text{bsmapt } x)] = [x \leftarrow \text{bsmap } l . P x]$   
**by** (*induct*  $l$ ) *auto*

**lemma** *bsmap-rev-xlate*:  
 $\text{bsmap } (\text{rev } q) = \text{rev } (\text{bsmap } q)$   
**by** (*induct*  $q$ ) *auto*

**lemma** *deleteMin-xlate*:  
 $q \neq [] \implies \text{bsmap } (\text{deleteMin } q) = \text{SkewBinomialHeapStruc.deleteMin } (\text{bsmap } q)$   
**apply** (*simp add:*  
*deleteMin-def SkewBinomialHeapStruc.deleteMin-def*  
*proj-xlate getMinTree-xlate insertList-xlate meld-xlate remove1Prio-xlate*  
*Let-def bsmapt-rev-xlate, (subst bsmapt-filter-xlate) ?*)  
**done**

**lemma** *deleteMin-correct-aux*:  
**assumes**  $I$ : *invar*  $q$   
**assumes**  $NE$ :  $q \neq []$   
**shows**  
*invar* ( $\text{deleteMin } q$ )

```

queue-to-multiset-aux (deleteMin q) = queue-to-multiset-aux q -
{# (findMin q, eprio (findMin q)) #}
apply (simp-all add:
  I NE deleteMin-xlate findMin-xlate-aux
  SkewBinomialHeapStruc.deleteMin-correct)
done

```

**lemma** *bsmap-fs-dep*:

```

(e,a)∈#SkewBinomialHeapStruc.tree-to-multiset (bsmapt t) ⇒ a=eprio e
(e,a)∈#SkewBinomialHeapStruc.queue-to-multiset (bsmap q) ⇒ a=eprio e
thm SkewBinomialHeapStruc.tree-to-multiset-queue-to-multiset.induct
apply (induct bsmapt t and bsmap q arbitrary: t and q
  rule: SkewBinomialHeapStruc.tree-to-multiset-queue-to-multiset.induct)
apply auto
apply (case-tac t)
apply (auto split: if-split-asm)
done

```

**lemma** *bsmap-fs-depD*:

```

(e,a)∈#SkewBinomialHeapStruc.tree-to-multiset (bsmapt t)
⇒ e ∈# tree-to-multiset t ∧ a=eprio e
(e,a)∈#SkewBinomialHeapStruc.queue-to-multiset (bsmap q)
⇒ e ∈# queue-to-multiset q ∧ a=eprio e
by (auto dest: bsmap-fs-dep intro!: image-eqI)

```

**lemma** *findMin-correct-aux*:

```

assumes I: invar q
assumes NE: q≠[]
shows (findMin q, eprio (findMin q)) ∈# queue-to-multiset-aux q
  ∀ y∈set-mset (queue-to-multiset-aux q). snd (findMin q,eprio (findMin q)) ≤ snd
  y
apply (simp-all add:
  I NE findMin-xlate-aux
  SkewBinomialHeapStruc.findMin-correct)
done

```

**lemma** *findMin-correct*:

```

assumes I: invar q
and NE: q≠[]
shows findMin q ∈# queue-to-multiset q
and ∀ y∈set-mset (queue-to-multiset q). eprio (findMin q) ≤ eprio y
using findMin-correct-aux[OF I NE]
apply simp-all
apply (force dest: bsmap-fs-depD)
apply auto
proof goal-cases
case prems: (1 a b)

```

```

from prems( $\beta$ ) have  $(a, \text{prio } a) \in \# \text{ queue-to-multiset-aux } q$ 
  apply –
  apply (frule bsmmap-fs-dep)
  apply simp
  done
with prems( $\beta$ )[rule-format, simplified]
show ?case by auto
qed

```

```

lemma deleteMin-correct:
  assumes I: invar q
  assumes NE:  $q \neq []$ 
  shows
    invar (deleteMin q)
     $\text{queue-to-multiset } (\text{deleteMin } q) = \text{queue-to-multiset } q -$ 
     $\{\# \text{ findMin } q \# \}$ 
  using deleteMin-correct-aux[OF I NE]
  apply simp-all
  apply (rule mset-image-fst-dep-pair-diff-split)
  apply (auto dest: bsmmap-fs-dep)
  done

```

```

declare insert.simps[simp del]

```

### 2.3.4 Bootstrapping: Phase 1

In this section, we define the ticked versions of the functions, as defined in [1]. These functions work on elements, i.e. only on heaps that contain at least one entry. Additionally, we define an invariant for elements, and a mapping to multisets of entries, and prove correct the ticked functions.

```

primrec findMin' where  $\text{findMin}' (\text{Element } e \ a \ q) = (e, a)$ 
fun meld'::  $(\text{'e}, \text{'a}::\text{linorder}) \text{ BsSkewElem} \Rightarrow$ 
   $(\text{'e}, \text{'a}) \text{ BsSkewElem} \Rightarrow (\text{'e}, \text{'a}) \text{ BsSkewElem}$ 
  where  $\text{meld}' (\text{Element } e1 \ a1 \ q1) (\text{Element } e2 \ a2 \ q2) =$ 
   $(\text{if } a1 \leq a2 \ \text{then}$ 
     $\text{Element } e1 \ a1 \ (\text{insert } (\text{Element } e2 \ a2 \ q2) \ q1)$ 
   $\text{else}$ 
     $\text{Element } e2 \ a2 \ (\text{insert } (\text{Element } e1 \ a1 \ q1) \ q2)$ 
   $)$ 
fun insert' where
   $\text{insert}' e \ a \ q = \text{meld}' (\text{Element } e \ a \ []) \ q$ 
fun deleteMin' where
   $\text{deleteMin}' (\text{Element } e \ a \ q) = ($ 
   $\text{case } (\text{findMin } q) \ \text{of}$ 
     $\text{Element } ey \ ay \ q1 \Rightarrow$ 
     $\text{Element } ey \ ay \ (\text{meld } q1 \ (\text{deleteMin } q))$ 
   $)$ 

```

Size-function for termination proofs

```
fun tree-level and queue-level where
  tree-level (BsNode (Element - - qd) - q) =
  max (Suc (queue-level qd)) (queue-level q) |
  queue-level [] = (0::nat) |
  queue-level (t#q) = max (tree-level t) (queue-level q)
```

```
fun level where
  level (Element - - q) = Suc (queue-level q)
```

```
lemma level-m:
  x ∈ #tree-to-multiset t ⇒ level x < Suc (tree-level t)
  x ∈ #queue-to-multiset q ⇒ level x < Suc (queue-level q)
apply (induct t and q rule: tree-level-queue-level.induct)
apply (case-tac [!] x)
apply (auto simp add: less-max-iff-disj)
done
```

```
lemma level-measure:
  x ∈ set-mset (queue-to-multiset q) ⇒ (x,(Element e a q)) ∈ measure level
  x ∈ # (queue-to-multiset q) ⇒ (x,(Element e a q)) ∈ measure level
apply (case-tac [!] x)
apply (auto dest: level-m simp del: set-image-mset)
done
```

Invariant for elements

```
function elem-invar where
  elem-invar (Element e a q) ↔
  (∀ x. x ∈ # (queue-to-multiset q) → a ≤ eprio x ∧ elem-invar x) ∧
  invar q
by pat-completeness auto
termination
proof
  show wf (measure level) by auto
qed (rule level-measure)
```

Abstraction to multisets

```
function elem-to-mset where
  elem-to-mset (Element e a q) = {# (e,a) #}
  + sum-mset (image-mset elem-to-mset (queue-to-multiset q))
by pat-completeness auto
termination
proof
  show wf (measure level) by auto
qed (rule level-measure)
```

```
lemma insert-correct':
  assumes I: elem-invar x
  shows
```

```

elem-invar (insert' e a x)
elem-to-mset (insert' e a x) = elem-to-mset x + {#(e,a)#}
using I
apply (case-tac [!] x)
apply (auto simp add: insert-correct union-ac)
done

```

```

lemma meld-correct':
assumes I: elem-invar x elem-invar x'
shows
elem-invar (meld' x x')
elem-to-mset (meld' x x') = elem-to-mset x + elem-to-mset x'
using I
apply (case-tac [!] x)
apply (case-tac [!] x')
apply (auto simp add: insert-correct union-ac)
done

```

```

lemma findMin'-min:
[[elem-invar x; y∈#elem-to-mset x]] ==> snd (findMin' x) ≤ snd y
proof (induct n≡level x arbitrary: x rule: full-nat-induct)
case 1
note IH=1.hyps[rule-format, OF - refl]
note PREMS=1.premis
obtain e a q where [simp]: x=Element e a q by (cases x) auto

from PREMS(2) have y=(e,a) ∨
y∈#sum-mset (image-mset elem-to-mset (queue-to-multiset q))
(is ?C1 ∨ ?C2)
by (auto split: if-split-asm)
moreover {
assume y=(e,a)
with PREMS have ?case by simp
} moreover {
assume ?C2
then obtain yx where
A: yx ∈# queue-to-multiset q and
B: y ∈# elem-to-mset yx
apply (auto elim!: in-image-msetE)
done

```

```

from A PREMS have IYX: elem-invar yx by auto

```

```

from PREMS(1) A have a ≤ eprio yx by auto
hence snd (findMin' x) ≤ snd (findMin' yx)
by (cases yx) auto
also
from IH[OF - IYX B] level-m(2)[OF A]
have snd (findMin' yx) ≤ snd y by simp

```

**finally have**  $?case$  .  
**} ultimately show**  $?case$  **by** *blast*  
**qed**

**lemma** *findMin-correct'*:  
**assumes**  $I$ : *elem-invar*  $x$   
**shows**  
 $findMin' x \in \# \text{ elem-to-mset } x$   
 $\forall y \in \text{set-mset } (\text{elem-to-mset } x). \text{snd } (findMin' x) \leq \text{snd } y$   
**using**  $I$   
**apply** (*cases*  $x$ )  
**apply** *simp*  
**apply** (*simp add: findMin'-min[OF I]*)  
**done**

**lemma** *deleteMin-correct'*:  
**assumes**  $I$ : *elem-invar* (*Element*  $e$   $a$   $q$ )  
**assumes**  $NE[simp]$ :  $q \neq []$   
**shows**  
 $\text{elem-invar } (\text{deleteMin}' (\text{Element } e \text{ a } q))$   
 $\text{elem-to-mset } (\text{deleteMin}' (\text{Element } e \text{ a } q)) =$   
 $\text{elem-to-mset } (\text{Element } e \text{ a } q) - \{\# \text{ findMin}' (\text{Element } e \text{ a } q) \# \}$

**proof** –

**from**  $I$  **have**  $IQ[simp]$ : *invar*  $q$  **by** *simp*  
**from** *findMin-correct'*[*OF IQ NE*] **have**  
 $FMIQ$ :  $findMin q \in \# \text{ queue-to-multiset } q$  **and**  
 $FMIN$ :  $\forall y. y \in \# (\text{queue-to-multiset } q) \implies \text{eprio } (findMin q) \leq \text{eprio } y$   
**by** (*auto simp del: set-image-mset*)  
**from**  $FMIQ I$  **have**  $FMEI$ : *elem-invar* ( $findMin q$ ) **by** *auto*  
**from**  $I$  **have**  $FEI$ :  $\forall y. y \in \# (\text{queue-to-multiset } q) \implies \text{elem-invar } y$  **by** *auto*

**obtain**  $ey ay qy$  **where**  $[simp]$ :  $findMin q = \text{Element } ey \text{ ay } qy$   
**by** (*cases findMin q*) *auto*

**from**  $FMEI$  **have**  
 $IQY[simp]$ : *invar*  $qy$  **and**  
 $AYMIN$ :  $\forall x. x \in \# \text{ queue-to-multiset } qy \implies ay \leq \text{eprio } x$  **and**  
 $QEI$ :  $\forall x. x \in \# \text{ queue-to-multiset } qy \implies \text{elem-invar } x$   
**by** *auto*

**show** *elem-invar* ( $\text{deleteMin}' (\text{Element } e \text{ a } q)$ )  
**using**  $AYMIN QEI FMIN FEI$   
**by** (*auto simp add: deleteMin-correct meld-correct in-diff-count*)

**from**  $FMIQ$  **have**  
 $S$ :  $(\text{queue-to-multiset } q - \{\# \text{Element } ey \text{ ay } qy \# \}) + \{\# \text{Element } ey \text{ ay } qy \# \}$   
 $= \text{queue-to-multiset } q$  **by** *simp*

**show**  $\text{elem-to-mset } (\text{deleteMin}' (\text{Element } e \text{ a } q)) =$

$elem\text{-}to\text{-}mset (Element\ e\ a\ q) = \{\# findMin' (Element\ e\ a\ q)\ \#\}$   
**apply** (*simp add: deleteMin-correct meld-correct*)  
**by** (*subst S[symmetric], simp add: union-ac*)

**qed**

### 2.3.5 Bootstrapping: Phase 2

In this phase, we extend the ticked versions to also work with empty priority queues.

**definition** *bs-empty* **where** *bs-empty*  $\equiv Inl\ ()$

**primrec** *bs-findMin* **where**  
*bs-findMin* (*Inr* *x*) = *findMin'* *x*

**fun** *bs-meld*  
 $:: ('e, 'a::linorder)\ BsSkewHeap \Rightarrow ('e, 'a)\ BsSkewHeap \Rightarrow ('e, 'a)\ BsSkewHeap$   
**where**  
*bs-meld* (*Inl*  $-$ ) *x* = *x* |  
*bs-meld* *x* (*Inl*  $-$ ) = *x* |  
*bs-meld* (*Inr* *x*) (*Inr* *x'*) = *Inr* (*meld'* *x* *x'*)

**lemma** [*simp*]: *bs-meld* *x* (*Inl* *u*) = *x*  
**by** (*cases* *x*) *auto*

**primrec** *bs-insert*  
 $:: 'e \Rightarrow ('a::linorder) \Rightarrow ('e, 'a)\ BsSkewHeap \Rightarrow ('e, 'a)\ BsSkewHeap$   
**where**  
*bs-insert* *e* *a* (*Inl*  $-$ ) = *Inr* (*Element* *e* *a*  $\square$ ) |  
*bs-insert* *e* *a* (*Inr* *x*) = *Inr* (*insert'* *e* *a* *x*)

**fun** *bs-deleteMin*  
 $:: ('e, 'a::linorder)\ BsSkewHeap \Rightarrow ('e, 'a)\ BsSkewHeap$   
**where**  
*bs-deleteMin* (*Inr* (*Element* *e* *a*  $\square$ )) = *Inl*  $()$  |  
*bs-deleteMin* (*Inr* (*Element* *e* *a* *q*)) = *Inr* (*deleteMin'* (*Element* *e* *a* *q*))

**primrec** *bs-invar*  $:: ('e, 'a::linorder)\ BsSkewHeap \Rightarrow bool$   
**where**  
*bs-invar* (*Inl*  $-$ )  $\longleftrightarrow True$  |  
*bs-invar* (*Inr* *x*)  $\longleftrightarrow elem\text{-}invar\ x$

**lemma** [*simp*]: *bs-invar* *bs-empty* **by** (*simp add: bs-empty-def*)

**primrec** *bs-to-mset*  $:: ('e, 'a::linorder)\ BsSkewHeap \Rightarrow ('e \times 'a)\ multiset$   
**where**  
*bs-to-mset* (*Inl*  $-$ ) =  $\{\#\}$  |  
*bs-to-mset* (*Inr* *x*) = *elem-to-mset* *x*

**theorem** *bs-empty-correct*: *h=bs-empty*  $\longleftrightarrow bs\text{-}to\text{-}mset\ h = \{\#\}$

```

apply (unfold bs-empty-def)
apply (cases h)
apply simp
apply (case-tac b)
apply simp
done

```

```

lemma bs-mset-of-empty[simp]:
  bs-to-mset bs-empty = {#}
by (simp add: bs-empty-def)

```

```

theorem bs-findMin-correct:
  assumes I: bs-invar h
  assumes NE: h ≠ bs-empty
  shows bs-findMin h ∈# bs-to-mset h
    ∀ y ∈ set-mset (bs-to-mset h). snd (bs-findMin h) ≤ snd y
  using I NE
  apply (case-tac [!] h)
  apply (auto simp add: bs-empty-def findMin-correct')
done

```

```

theorem bs-insert-correct:
  assumes I: bs-invar h
  shows
    bs-invar (bs-insert e a h)
    bs-to-mset (bs-insert e a h) = {#(e,a)#} + bs-to-mset h
  using I
  apply (case-tac [!] h)
  apply (simp-all)
  apply (auto simp add: meld-correct')
done

```

```

theorem bs-meld-correct:
  assumes I: bs-invar h bs-invar h'
  shows
    bs-invar (bs-meld h h')
    bs-to-mset (bs-meld h h') = bs-to-mset h + bs-to-mset h'
  using I
  apply (case-tac [!] h, case-tac [!] h')
  apply (auto simp add: meld-correct')
done

```

```

theorem bs-deleteMin-correct:
  assumes I: bs-invar h
  assumes NE: h ≠ bs-empty
  shows
    bs-invar (bs-deleteMin h)
    bs-to-mset (bs-deleteMin h) = bs-to-mset h - {#bs-findMin h#}
  using I NE

```



```

apply (case-tac [!] h)
apply (simp-all add: bs-empty-def)
apply (case-tac [!] b)
apply (rename-tac [!] list)
apply (case-tac [!] list)
apply (simp-all del: elem-invar.simps deleteMin'.simps add: deleteMin-correct')
done

```

**end**

**interpretation** *BsSkewBinomialHeapStruc*: *Bootstrapped* .

## 2.4 Hiding the Invariant

### 2.4.1 Datatype

```

typedef (overloaded) ('e, 'a) SkewBinomialHeap =
  {q :: ('e, 'a)::linorder) BsSkewBinomialHeapStruc.BsSkewHeap. BsSkewBinomialHeapStruc.bs-invar q }
apply (rule-tac x=BsSkewBinomialHeapStruc.bs-empty in exI)
apply (auto)
done

```

```

lemma Rep-SkewBinomialHeap-invar[simp]:
  BsSkewBinomialHeapStruc.bs-invar (Rep-SkewBinomialHeap x)
using Rep-SkewBinomialHeap
by (auto)

```

```

lemma [simp]:
  BsSkewBinomialHeapStruc.bs-invar q
  ==> Rep-SkewBinomialHeap (Abs-SkewBinomialHeap q) = q
using Abs-SkewBinomialHeap-inverse by auto

```

```

lemma [simp, code abstype]: Abs-SkewBinomialHeap (Rep-SkewBinomialHeap q)
= q
by (rule Rep-SkewBinomialHeap-inverse)

```

```

locale SkewBinomialHeap-loc
begin

```

### 2.4.2 Operations

```

definition [code]:
  to-mset t
  == BsSkewBinomialHeapStruc.bs-to-mset (Rep-SkewBinomialHeap t)

```

```

definition empty where
  empty == Abs-SkewBinomialHeap BsSkewBinomialHeapStruc.bs-empty
lemma [code abstract, simp]:

```

*Rep-SkewBinomialHeap empty* = *BsSkewBinomialHeapStruct.bs-empty*  
 by (unfold empty-def) simp

**definition** [code]:

*isEmpty* *q* == *Rep-SkewBinomialHeap q* = *BsSkewBinomialHeapStruct.bs-empty*

**lemma** *empty-rep*:

*q=empty*  $\longleftrightarrow$  *Rep-SkewBinomialHeap q* = *BsSkewBinomialHeapStruct.bs-empty*

**apply** (auto simp add: empty-def)

**apply** (metis *Rep-SkewBinomialHeap-inverse*)

**done**

**lemma** *isEmpty-correct*: *isEmpty q*  $\longleftrightarrow$  *q=empty*

by (simp add: empty-rep isEmpty-def)

**definition**

*insert*

:: 'e  $\Rightarrow$  ('a::linorder)  $\Rightarrow$  ('e,'a) *SkewBinomialHeap*

$\Rightarrow$  ('e,'a) *SkewBinomialHeap*

**where** *insert e a q* ==

*Abs-SkewBinomialHeap* (

*BsSkewBinomialHeapStruct.bs-insert e a (Rep-SkewBinomialHeap q)*)

**lemma** [code abstract]:

*Rep-SkewBinomialHeap (insert e a q)*

= *BsSkewBinomialHeapStruct.bs-insert e a (Rep-SkewBinomialHeap q)*

by (simp add: insert-def *BsSkewBinomialHeapStruct.bs-insert-correct*)

**definition** [code]: *findMin q*

== *BsSkewBinomialHeapStruct.bs-findMin (Rep-SkewBinomialHeap q)*

**definition** *deleteMin q* ==

if *q=empty* then *empty*

else *Abs-SkewBinomialHeap* (

*BsSkewBinomialHeapStruct.bs-deleteMin (Rep-SkewBinomialHeap q)*)

We don't use equality here, to prevent the code-generator from introducing equality-class parameter for type 'a. Instead we use a case-distinction to check for emptiness.

**lemma** [code abstract]: *Rep-SkewBinomialHeap (deleteMin q)* =

(case (*Rep-SkewBinomialHeap q*) of *Inl* -  $\Rightarrow$  *BsSkewBinomialHeapStruct.bs-empty*

|

-  $\Rightarrow$  *BsSkewBinomialHeapStruct.bs-deleteMin (Rep-SkewBinomialHeap q)*)

**proof** (cases (*Rep-SkewBinomialHeap q*))

**case** [simp]: (*Inl a*)

**hence** (*Rep-SkewBinomialHeap q*) = *BsSkewBinomialHeapStruct.bs-empty*

**apply** (cases *q*)

**apply** (auto simp add: *BsSkewBinomialHeapStruct.bs-empty-def*)

**done**

**thus** ?thesis

**apply** (auto simp add: *deleteMin-def*)

```

      BsSkewBinomialHeapStruc.bs-deleteMin-correct
      BsSkewBinomialHeapStruc.bs-empty-correct empty-rep )
    done
  next
  case (Inr x)
  hence (Rep-SkewBinomialHeap q) ≠ BsSkewBinomialHeapStruc.bs-empty
  apply (cases q)
  apply (auto simp add: BsSkewBinomialHeapStruc.bs-empty-def)
  done
  thus ?thesis
  apply (simp add: Inr)
  apply (fold Inr)
  apply (auto simp add: deleteMin-def
    BsSkewBinomialHeapStruc.bs-deleteMin-correct
    BsSkewBinomialHeapStruc.bs-empty-correct empty-rep )
  done
qed

```

```

definition meld q1 q2 ==
  Abs-SkewBinomialHeap (BsSkewBinomialHeapStruc.bs-meld
    (Rep-SkewBinomialHeap q1) (Rep-SkewBinomialHeap q2))
lemma [code abstract]:
  Rep-SkewBinomialHeap (meld q1 q2)
  = BsSkewBinomialHeapStruc.bs-meld (Rep-SkewBinomialHeap q1)
    (Rep-SkewBinomialHeap q2)
  by (simp add: meld-def BsSkewBinomialHeapStruc.bs-meld-correct)

```

### 2.4.3 Correctness

```

lemma empty-correct: to-mset q = {#} ↔ q=empty
  by (simp add: to-mset-def BsSkewBinomialHeapStruc.bs-empty-correct empty-rep)

```

```

lemma to-mset-of-empty[simp]: to-mset empty = {#}
  by (simp add: empty-correct)

```

```

lemma insert-correct: to-mset (insert e a q) = to-mset q + {#(e,a)#}
  apply (unfold insert-def to-mset-def)
  apply (simp add: BsSkewBinomialHeapStruc.bs-insert-correct union-ac)
  done

```

```

lemma findMin-correct:
  assumes q≠empty
  shows
    findMin q ∈# to-mset q
    ∀ y∈set-mset (to-mset q). snd (findMin q) ≤ snd y
  using assms
  apply (unfold findMin-def to-mset-def)

```

**apply** (*simp-all add: empty-rep BsSkewBinomialHeapStruct.bs-findMin-correct*)  
**done**

**lemma** *deleteMin-correct*:

**assumes** *q≠empty*

**shows**  $to\_mset (deleteMin\ q) = to\_mset\ q - \{\# findMin\ q\ \#\}$

**using** *assms*

**apply** (*unfold findMin-def deleteMin-def to-mset-def*)

**apply** (*simp-all add: empty-rep BsSkewBinomialHeapStruct.bs-deleteMin-correct*)

**done**

**lemma** *meld-correct*:

**shows**  $to\_mset (meld\ q\ q') = to\_mset\ q + to\_mset\ q'$

**apply** (*unfold to-mset-def meld-def*)

**apply** (*simp-all add: BsSkewBinomialHeapStruct.bs-meld-correct*)

**done**

Correctness lemmas to be used with simplifier

**lemmas** *correct = empty-correct deleteMin-correct meld-correct*

**end**

**interpretation** *SkewBinomialHeap: SkewBinomialHeap-loc* .

## 2.5 Documentation

*SkewBinomialHeap.to-mset::('a, 'b) SkewBinomialHeap ⇒ ('a × 'b) multiset*  
 Abstraction to multiset.

*SkewBinomialHeap.empty::('a, 'b) SkewBinomialHeap*

The empty heap. ( $O(1)$ )

**Spec** *SkewBinomialHeap.empty-correct*:

$(SkewBinomialHeap.to\_mset\ q = \{\#\}) = (q = SkewBinomialHeap.empty)$

*SkewBinomialHeap.isEmpty::('a, 'b) SkewBinomialHeap ⇒ bool*

Checks whether heap is empty. Mainly used to work around code-generation issues. ( $O(1)$ )

**Spec** *SkewBinomialHeap.isEmpty-correct*:

$SkewBinomialHeap.isEmpty\ q = (q = SkewBinomialHeap.empty)$

*SkewBinomialHeap.insert*

$SkewBinomialHeap.insert::'a ⇒ 'b ⇒ ('a, 'b) SkewBinomialHeap$   
 $⇒ ('a, 'b) SkewBinomialHeap$

Inserts element ( $O(1)$ )

**Spec** *SkewBinomialHeap.insert-correct*:

$SkewBinomialHeap.to-mset (SkewBinomialHeap.insert\ e\ a\ q) =$   
 $SkewBinomialHeap.to-mset\ q + \{\#(e, a)\# \}$

$SkewBinomialHeap.findMin::('a, 'b)\ SkewBinomialHeap \Rightarrow 'a \times 'b$

Returns a minimal element ( $O(1)$ )

**Spec**  $SkewBinomialHeap.findMin$ -correct:

$q \neq SkewBinomialHeap.empty \Rightarrow$

$SkewBinomialHeap.findMin\ q \in\# SkewBinomialHeap.to-mset\ q$

$q \neq SkewBinomialHeap.empty \Rightarrow$

$\forall y \in\# SkewBinomialHeap.to-mset\ q. snd\ (SkewBinomialHeap.findMin\ q) \leq snd\ y$

$SkewBinomialHeap.deleteMin$

$SkewBinomialHeap.deleteMin::('a, 'b)\ SkewBinomialHeap$

$\Rightarrow ('a, 'b)\ SkewBinomialHeap$

Deletes *the* element that is returned by  $find\_min$ .  $O(\log(n))$

**Spec**  $SkewBinomialHeap.deleteMin$ -correct:

$q \neq SkewBinomialHeap.empty \Rightarrow$

$SkewBinomialHeap.to-mset (SkewBinomialHeap.deleteMin\ q) =$

$SkewBinomialHeap.to-mset\ q - \{\#SkewBinomialHeap.findMin\ q\# \}$

$SkewBinomialHeap.meld$

$SkewBinomialHeap.meld::('a, 'b)\ SkewBinomialHeap$

$\Rightarrow ('a, 'b)\ SkewBinomialHeap$

$\Rightarrow ('a, 'b)\ SkewBinomialHeap$

Melds two heaps ( $O(1)$ )

**Spec**  $SkewBinomialHeap.meld$ -correct:

$SkewBinomialHeap.to-mset (SkewBinomialHeap.meld\ q\ q') =$

$SkewBinomialHeap.to-mset\ q + SkewBinomialHeap.to-mset\ q'$

end

## References

- [1] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6:839–857, 1996.