

# Priority Queues Based on Braun Trees

Tobias Nipkow

December 14, 2021

## Abstract

This entry verifies priority queues based on Braun trees. Insertion and deletion take logarithmic time and preserve the balanced nature of Braun trees. Two implementations of deletion are provided.

## Contents

<b>1</b>	<b>Priority Queues Based on Braun Trees</b>	<b>1</b>
1.1	Introduction . . . . .	2
1.2	Get Minimum . . . . .	2
1.3	Insertion . . . . .	2
1.4	Deletion . . . . .	2
<b>2</b>	<b>Priority Queues Based on Braun Trees 2</b>	<b>5</b>
2.1	Function <i>del-min2</i> . . . . .	5
2.2	Correctness Proof . . . . .	5
<b>3</b>	<b>Sorting via Priority Queues Based on Braun Trees</b>	<b>6</b>
<b>4</b>	<b>Phase 1: List to Tree</b>	<b>7</b>
<b>5</b>	<b>Phase 2: Heap to List</b>	<b>10</b>

## 1 Priority Queues Based on Braun Trees

```
theory Priority-Queue-Braun
imports
  HOL-Library.Tree-Multiset
  HOL-Library.Pattern-Aliases
  HOL-Data-Structures.Priority-Queue-Specs
  HOL-Data-Structures.Braun-Tree
begin
```

## 1.1 Introduction

Braun, Rem and Hoogerwoord [1, 2] used specific balanced binary trees, often called Braun trees (where in each node with subtrees  $l$  and  $r$ ,  $size(r) \leq size(l) \leq size(r) + 1$ ), to implement flexible arrays. Paulson [3] (based on code supplied by Okasaki) implemented priority queues via Braun trees. This theory verifies Paulson's implementation, with small simplifications.

Direct proof of logarithmic height. Also follows from the fact that Braun trees are balanced (proved in the base theory).

**lemma** *height-size-braun*:  $braun\ t \implies 2^{\wedge}(height\ t) \leq 2 * size\ t + 1$   
*<proof>*

## 1.2 Get Minimum

**fun** *get-min* :: 'a::linorder tree  $\Rightarrow$  'a **where**  
*get-min* (Node l a r) = a

**lemma** *get-min*:  $\llbracket heap\ t; t \neq Leaf \rrbracket \implies get-min\ t = Min-mset\ (mset-tree\ t)$   
*<proof>*

## 1.3 Insertion

**hide-const** (open) *insert*

**fun** *insert* :: 'a::linorder  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree **where**  
*insert* a Leaf = Node Leaf a Leaf |  
*insert* a (Node l x r) =  
 (if a < x then Node (insert x r) a l else Node (insert a r) x l)

**lemma** *size-insert[simp]*:  $size(insert\ x\ t) = size\ t + 1$   
*<proof>*

**lemma** *mset-insert*:  $mset-tree(insert\ x\ t) = \{x\} + mset-tree\ t$   
*<proof>*

**lemma** *set-insert[simp]*:  $set-tree(insert\ x\ t) = \{x\} \cup (set-tree\ t)$   
*<proof>*

**lemma** *braun-insert*:  $braun\ t \implies braun(insert\ x\ t)$   
*<proof>*

**lemma** *heap-insert*:  $heap\ t \implies heap(insert\ x\ t)$   
*<proof>*

## 1.4 Deletion

Slightly simpler definition of *del-left* which avoids the need to appeal to the Braun invariant.

**fun** *del-left* :: 'a tree  $\Rightarrow$  'a \* 'a tree **where**  
*del-left* (Node Leaf x r) = (x,r) |  
*del-left* (Node l x r) = (let (y,l') = *del-left* l in (y,Node r x l'))

**lemma** *del-left-mset-plus*:  
*del-left* t = (x,t')  $\Longrightarrow$  t  $\neq$  Leaf  
 $\Longrightarrow$  mset-tree t = {#x#} + mset-tree t'  
 <proof>

**lemma** *del-left-mset*:  
*del-left* t = (x,t')  $\Longrightarrow$  t  $\neq$  Leaf  
 $\Longrightarrow$  x  $\in$  # mset-tree t  $\wedge$  mset-tree t' = mset-tree t - {#x#}  
 <proof>

**lemma** *del-left-set*:  
*del-left* t = (x,t')  $\Longrightarrow$  t  $\neq$  Leaf  $\Longrightarrow$  set-tree t = {x}  $\cup$  set-tree t'  
 <proof>

**lemma** *del-left-heap*:  
*del-left* t = (x,t')  $\Longrightarrow$  t  $\neq$  Leaf  $\Longrightarrow$  heap t  $\Longrightarrow$  heap t'  
 <proof>

**lemma** *del-left-size*:  
*del-left* t = (x,t')  $\Longrightarrow$  t  $\neq$  Leaf  $\Longrightarrow$  size t = size t' + 1  
 <proof>

**lemma** *del-left-braun*:  
*del-left* t = (x,t')  $\Longrightarrow$  t  $\neq$  Leaf  $\Longrightarrow$  braun t  $\Longrightarrow$  braun t'  
 <proof>

**context includes** *pattern-aliases*

**begin**

Slightly simpler definition: - instead of <> because of Braun invariant.

**function** (*sequential*) *sift-down* :: 'a::linorder tree  $\Rightarrow$  'a  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree **where**  
*sift-down* Leaf a - = Node Leaf a Leaf |  
*sift-down* (Node Leaf x -) a Leaf =  
 (if a  $\leq$  x then Node (Node Leaf x Leaf) a Leaf  
 else Node (Node Leaf a Leaf) x Leaf) |  
*sift-down* (Node l1 x1 r1 =: t1) a (Node l2 x2 r2 =: t2) =  
 (if a  $\leq$  x1  $\wedge$  a  $\leq$  x2  
 then Node t1 a t2  
 else if x1  $\leq$  x2 then Node (*sift-down* l1 a r1) x1 t2  
 else Node t1 x2 (*sift-down* l2 a r2))  
 <proof>  
**termination**  
 <proof>

**end**

**lemma** *size-sift-down*:

$braun(Node\ l\ a\ r) \implies size(sift-down\ l\ a\ r) = size\ l + size\ r + 1$   
*<proof>*

**lemma** *braun-sift-down*:

$braun(Node\ l\ a\ r) \implies braun(sift-down\ l\ a\ r)$   
*<proof>*

**lemma** *mset-sift-down*:

$braun(Node\ l\ a\ r) \implies mset-tree(sift-down\ l\ a\ r) = \{a\} + (mset-tree\ l + mset-tree\ r)$   
*<proof>*

**lemma** *set-sift-down*:  $braun(Node\ l\ a\ r)$

$\implies set-tree(sift-down\ l\ a\ r) = \{a\} \cup (set-tree\ l \cup set-tree\ r)$   
*<proof>*

**lemma** *heap-sift-down*:

$braun(Node\ l\ a\ r) \implies heap\ l \implies heap\ r \implies heap(sift-down\ l\ a\ r)$   
*<proof>*

**fun** *del-min* :: 'a::linorder tree  $\Rightarrow$  'a tree **where**

*del-min* Leaf = Leaf |

*del-min* (Node Leaf x r) = Leaf |

*del-min* (Node l x r) = (let (y,l') = del-left l in sift-down r y l')

**lemma** *braun-del-min*:  $braun\ t \implies braun(del-min\ t)$

*<proof>*

**lemma** *heap-del-min*:  $heap\ t \implies braun\ t \implies heap(del-min\ t)$

*<proof>*

**lemma** *size-del-min*: **assumes**  $braun\ t$  **shows**  $size(del-min\ t) = size\ t - 1$

*<proof>*

**lemma** *mset-del-min*: **assumes**  $braun\ t\ t \neq Leaf$

**shows**  $mset-tree(del-min\ t) = mset-tree\ t - \{\#get-min\ t\# \}$

*<proof>*

Last step: prove all axioms of the priority queue specification:

**interpretation** *braun*: Priority-Queue

**where** *empty* = Leaf **and** *is-empty* =  $\lambda h. h = Leaf$

**and** *insert* = insert **and** *del-min* = del-min

**and** *get-min* = get-min **and** *invar* =  $\lambda h. braun\ h \wedge heap\ h$

**and** *mset* = mset-tree

*<proof>*

**end**

## 2 Priority Queues Based on Braun Trees 2

```
theory Priority-Queue-Braun2
imports Priority-Queue-Braun
begin
```

This is the version verified by Jean-Christophe Filliâtre with the help of the Why3 system [http://toccata.lri.fr/gallery/braun\\_trees.en.html](http://toccata.lri.fr/gallery/braun_trees.en.html). Only the deletion function (*del-min2* below) differs from Paulson's version. But the difference turns out to be minor — see below.

### 2.1 Function *del-min2*

```
fun le-root :: 'a::linorder  $\Rightarrow$  'a tree  $\Rightarrow$  bool where
le-root a t = (t = Leaf  $\vee$  a  $\leq$  value t)
```

```
fun replace-min :: 'a::linorder  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree where
replace-min x (Node l - r) =
  (if le-root x l & le-root x r then Node l x r
   else
    let a = value l in
    if le-root a r then Node (replace-min x l) a r
    else Node l (value r) (replace-min x r))
```

```
fun merge :: 'a::linorder tree  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree where
merge l Leaf = l |
merge (Node l1 a1 r1) (Node l2 a2 r2) =
  (if a1  $\leq$  a2 then Node (Node l2 a2 r2) a1 (merge l1 r1)
   else let (x, l') = del-left (Node l1 a1 r1)
        in Node (replace-min x (Node l2 a2 r2)) a2 l')
```

```
fun del-min2 where
del-min2 Leaf = Leaf |
del-min2 (Node l x r) = merge l r
```

### 2.2 Correctness Proof

It turns out that *replace-min* is just *sift-down* in disguise:

```
lemma replace-min-sift-down: braun (Node l a r)  $\implies$  replace-min x (Node l a r)
= sift-down l x r
<proof>
```

This means that *del-min2* is merely a slight optimization of *del-min*: instead of calling *del-left* right away, *merge* can take advantage of the case where the smaller element is at the root of the left heap and can be moved up without complications. However, on average this is just the case on the first level.

Function *merge*:

**lemma** *mset-tree-merge*:

$braun (Node\ l\ x\ r) \implies mset-tree(merge\ l\ r) = mset-tree\ l + mset-tree\ r$   
*<proof>*

**lemma** *heap-merge*:

$\llbracket braun (Node\ l\ x\ r); heap\ l; heap\ r \rrbracket \implies heap(merge\ l\ r)$   
*<proof>*

**lemma** *del-left-braun-size*:

$del-left\ t = (x, t') \implies braun\ t \implies t \neq Leaf \implies braun\ t' \wedge size\ t = size\ t' + 1$   
*<proof>*

**lemma** *braun-size-merge*:

$braun (Node\ l\ x\ r) \implies braun(merge\ l\ r) \wedge size(merge\ l\ r) = size\ l + size\ r$   
*<proof>*

Last step: prove all axioms of the priority queue specification:

**interpretation** *braun: Priority-Queue*

**where** *empty* = *Leaf* **and** *is-empty* =  $\lambda h. h = Leaf$

**and** *insert* = *insert* **and** *del-min* = *del-min2*

**and** *get-min* = *get-min* **and** *invar* =  $\lambda h. braun\ h \wedge heap\ h$

**and** *mset* = *mset-tree*

*<proof>*

**end**

### 3 Sorting via Priority Queues Based on Braun Trees

**theory** *Sorting-Braun*

**imports** *Priority-Queue-Braun*

**begin**

This theory is about sorting algorithms based on heaps. Algorithm A can be found here <http://www.csse.canterbury.ac.nz/walter.guttmann/publications/0005.pdf> on p. 54. (published here <http://www.jucs.org/doi?doi=10.3217/jucs-009-02-0173>) Not really the classic heap sort but a mixture of heap sort and merge sort. The algorithm (B) in Larry's book comes closer to the classic heap sort: <https://www.cl.cam.ac.uk/~lp15/MLbook/programs/sample7.sml>.

Both algorithms have two phases: build a heap from a list, then extract the elements of the heap into a sorted list.

**abbreviation**(*input*)

$nlog2\ n == nat(ceiling(log\ 2\ n))$

## 4 Phase 1: List to Tree

Algorithm A does this naively, in  $O(n \lg n)$  fashion and generates a Braun tree:

```
fun heap-of-A :: ('a::linorder) list  $\Rightarrow$  'a tree where  
heap-of-A [] = Leaf |  
heap-of-A (a#as) = insert a (heap-of-A as)
```

```
lemma heap-heap-of-A: heap (heap-of-A xs)  
 $\langle$ proof $\rangle$ 
```

```
lemma braun-heap-of-A: braun (heap-of-A xs)  
 $\langle$ proof $\rangle$ 
```

```
lemma mset-tree-heap-of-A: mset-tree (heap-of-A xs) = mset xs  
 $\langle$ proof $\rangle$ 
```

Running time is  $n \cdot \log n$ , which we can approximate with height.

```
fun t-insert :: ('a::linorder)  $\Rightarrow$  'a tree  $\Rightarrow$  nat where  
t-insert a Leaf = 1 |  
t-insert a (Node l x r) =  
  (if a < x then 1 + t-insert x r else 1 + t-insert a r)
```

```
fun t-heap-of-A :: ('a::linorder) list  $\Rightarrow$  nat where  
t-heap-of-A [] = 0 |  
t-heap-of-A (a#as) = t-insert a (heap-of-A as) + t-heap-of-A as
```

```
lemma t-insert-height:  
t-insert x t  $\leq$  height t + 1  
 $\langle$ proof $\rangle$ 
```

```
lemma height-insert-ge:  
height t  $\leq$  height (insert x t)  
 $\langle$ proof $\rangle$ 
```

```
lemma t-heap-of-A-bound:  
t-heap-of-A xs  $\leq$  length xs * (height (heap-of-A xs) + 1)  
 $\langle$ proof $\rangle$ 
```

```
lemma size-heap-of-A:  
size (heap-of-A xs) = length xs  
 $\langle$ proof $\rangle$ 
```

```
lemma t-heap-of-A-log-bound:  
t-heap-of-A xs  $\leq$  length xs * (nlog2 (length xs + 1) + 1)  
 $\langle$ proof $\rangle$ 
```

Algorithm B mimics heap sort more closely by building heaps bottom

up in a balanced way:

```
fun heapify :: nat => ('a::linorder) list => 'a tree * 'a list where
heapify 0 xs = (Leaf, xs) |
heapify (Suc n) (x#xs) =
  (let (l, ys) = heapify (Suc n div 2) xs;
      (r, zs) = heapify (n div 2) ys
   in (sift-down l x r, zs))
```

The result should be a Braun tree:

**lemma** *heapify-snd*:

```
n ≤ length xs => snd (heapify n xs) = drop n xs
⟨proof⟩
```

**lemma** *heapify-snd-tup*:

```
heapify n xs = (t, ys) => n ≤ length xs => ys = drop n xs
⟨proof⟩
```

**lemma** *heapify-correct*:

```
n ≤ length xs => heapify n xs = (t, ys) =>
  size t = n ∧ heap t ∧ braun t ∧ mset-tree t = mset (take n xs)
⟨proof⟩
```

**lemma** *braun-heapify*:

```
n ≤ length xs => braun (fst (heapify n xs))
⟨proof⟩
```

**lemma** *heap-heapify*:

```
n ≤ length xs => heap (fst (heapify n xs))
⟨proof⟩
```

**lemma** *mset-heapify*:

```
n ≤ length xs => mset-tree (fst (heapify n xs)) = mset (take n xs)
⟨proof⟩
```

The running time of heapify is linear. (similar to [https://en.wikipedia.org/wiki/Binary\\_heap#Building\\_a\\_heap](https://en.wikipedia.org/wiki/Binary_heap#Building_a_heap))

This is an interesting result, so we embark on this exercise to prove it the hard way.

**context includes** *pattern-aliases*

**begin**

**function** (*sequential*) *t-sift-down* :: 'a::linorder tree => 'a => 'a tree => nat **where**

```
t-sift-down Leaf a Leaf = 1 |
t-sift-down (Node Leaf x Leaf) a Leaf = 2 |
t-sift-down (Node l1 x1 r1 =: t1) a (Node l2 x2 r2 =: t2) =
  (if a ≤ x1 ∧ a ≤ x2
   then 1
   else if x1 ≤ x2 then 1 + t-sift-down l1 a r1
        else 1 + t-sift-down l2 a r2)
```



*<proof>*

**termination**

*<proof>*

**end**

**fun** *t-heapify* :: *nat*  $\Rightarrow$  (*'a::linorder*) *list*  $\Rightarrow$  *nat* **where**

*t-heapify* 0 *xs* = 1 |

*t-heapify* (*Suc* *n*) (*x#xs*) =

  (*let* (*l*, *ys*) = *heapify* (*Suc* *n* *div* 2) *xs*;

*t1* = *t-heapify* (*Suc* *n* *div* 2) *xs*;

    (*r*, *zs*) = *heapify* (*n* *div* 2) *ys*;

*t2* = *t-heapify* (*n* *div* 2) *ys*

*in* 1 + *t1* + *t2* + *t-sift-down* *l* *x* *r*)

**lemma** *t-sift-down-height*:

*braun* (*Node* *l* *x* *r*)  $\Longrightarrow$  *t-sift-down* *l* *x* *r*  $\leq$  *height* (*Node* *l* *x* *r*)

*<proof>*

**lemma** *sift-down-height*:

*braun* (*Node* *l* *x* *r*)  $\Longrightarrow$  *height* (*sift-down* *l* *x* *r*)  $\leq$  *height* (*Node* *l* *x* *r*)

*<proof>*

**lemma** *braun-height-r-le*:

*braun* (*Node* *l* *x* *r*)  $\Longrightarrow$  *height* *r*  $\leq$  *height* *l*

*<proof>*

**lemma** *braun-height-l-le*:

**assumes** *b*: *braun* (*Node* *l* *x* *r*)

**shows** *height* *l*  $\leq$  *Suc* (*height* *r*)

*<proof>*

**lemma** *braun-height-node-eq*:

**assumes** *b*: *braun* (*Node* *l* *x* *r*)

**shows** *height* (*Node* *l* *x* *r*) = *Suc* (*height* *l*)

*<proof>*

**lemma** *t-heapify-induct*:

*i*  $\leq$  *length* *xs*  $\Longrightarrow$  *t-heapify* *i* *xs* + *height* (*fst* (*heapify* *i* *xs*))  $\leq$  5 \* *i* + 1

*<proof>*

**lemma** *t-heapify-bound*:

*i*  $\leq$  *length* *xs*  $\Longrightarrow$  *t-heapify* *i* *xs*  $\leq$  5 \* *i* + 1

*<proof>*

## 5 Phase 2: Heap to List

Algorithm A extracts (*list-of-A*) the list by removing the root and merging the children:

**lemma** *size-prod-measure*[*measure-function*]:  
 $is\text{-}measure\ f \implies is\text{-}measure\ g \implies is\text{-}measure\ (size\text{-}prod\ f\ g)$   
*<proof>*

**fun** *merge* :: ('a::linorder) tree  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree **where**  
*merge* Leaf t2 = t2 |  
*merge* t1 Leaf = t1 |  
*merge* (Node l1 a1 r1) (Node l2 a2 r2) =  
 (if a1  $\leq$  a2 then Node (merge l1 r1) a1 (Node l2 a2 r2)  
 else Node (Node l1 a1 r1) a2 (merge l2 r2))

**value** *merge*  $\langle \rangle, 0::int, \langle \rangle \langle \rangle, 0, \langle \rangle = \langle \rangle, 0, \langle \rangle, 0, \langle \rangle \rangle$

**lemma** *merge-size*[*termination-simp*]:  
 $size\ (merge\ l\ r) = size\ l + size\ r$   
*<proof>*

**fun** *list-of-A* :: ('a::linorder) tree  $\Rightarrow$  'a list **where**  
*list-of-A* Leaf = [] |  
*list-of-A* (Node l a r) = a # *list-of-A* (merge l r)

**value** *list-of-A* (*heap-of-A* *shuffle100*)

**lemma** *set-tree-merge*[*simp*]:  
 $set\text{-}tree\ (merge\ l\ r) = set\text{-}tree\ l \cup set\text{-}tree\ r$   
*<proof>*

**lemma** *mset-tree-merge*[*simp*]:  
 $mset\text{-}tree\ (merge\ l\ r) = mset\text{-}tree\ l + mset\text{-}tree\ r$   
*<proof>*

**lemma** *merge-heap*:  
 $heap\ l \implies heap\ r \implies heap\ (merge\ l\ r)$   
*<proof>*

**lemma** *set-list-of-A*[*simp*]:  
 $set\ (list\text{-}of\text{-}A\ t) = set\text{-}tree\ t$   
*<proof>*

**lemma** *mset-list-of-A*[*simp*]:  
 $mset\ (list\text{-}of\text{-}A\ t) = mset\text{-}tree\ t$   
*<proof>*

**lemma** *sorted-list-of-A*:

$heap\ t \implies sorted\ (list-of-A\ t)$   
 $\langle proof \rangle$

**lemma** *sortedA*:  $sorted\ (list-of-A\ (heap-of-A\ xs))$   
 $\langle proof \rangle$

**lemma** *msetA*:  $mset\ (list-of-A\ (heap-of-A\ xs)) = mset\ xs$   
 $\langle proof \rangle$

Does *list-of-A* take time  $O(n \lg n)$ ? Although *merge* does not preserve *braun*, it cannot increase the height of the heap.

**lemma** *merge-height*:  
 $height\ (merge\ l\ r) \leq Suc\ (max\ (height\ l)\ (height\ r))$   
 $\langle proof \rangle$

**corollary** *merge-height-display*:  
 $height\ (merge\ l\ r) \leq height\ (Node\ l\ x\ r)$   
 $\langle proof \rangle$

**fun** *t-merge* ::  $( 'a :: linorder )\ tree \Rightarrow 'a\ tree \Rightarrow nat$  **where**  
 $t-merge\ Leaf\ t2 = 0 \mid$   
 $t-merge\ t1\ Leaf = 0 \mid$   
 $t-merge\ (Node\ l1\ a1\ r1)\ (Node\ l2\ a2\ r2) =$   
     $(if\ a1 \leq a2\ then\ 1 + t-merge\ l1\ r1$   
       $else\ 1 + t-merge\ l2\ r2)$

**fun** *t-list-of-A* ::  $( 'a :: linorder )\ tree \Rightarrow nat$  **where**  
 $t-list-of-A\ Leaf = 0 \mid$   
 $t-list-of-A\ (Node\ l\ a\ r) = 1 + t-merge\ l\ r + t-list-of-A\ (merge\ l\ r)$

**lemma** *t-merge-height*:  
 $t-merge\ l\ r \leq max\ (height\ l)\ (height\ r)$   
 $\langle proof \rangle$

**lemma** *t-list-of-A-induct*:  
 $height\ t \leq n \implies t-list-of-A\ t \leq 2 * n * size\ t$   
 $\langle proof \rangle$

**lemma** *t-list-of-A-bound*:  
 $t-list-of-A\ t \leq 2 * height\ t * size\ t$   
 $\langle proof \rangle$

**lemma** *t-list-of-A-log-bound*:  
 $braun\ t \implies t-list-of-A\ t \leq 2 * nlog2\ (size\ t + 1) * size\ t$   
 $\langle proof \rangle$

**value** *t-list-of-A*  $(heap-of-A\ shuffle100)$

**theorem** *t-sortA*:

$t\text{-heap-of-A } xs + t\text{-list-of-A } (\text{heap-of-A } xs) \leq 3 * \text{length } xs * (\text{nlog2 } (\text{length } xs + 1) + 1)$   
 (is ?lhs ≤ -)  
 ⟨proof⟩

Running time of algorithm B:

**function** *list-of-B* :: ('a::linorder) tree ⇒ 'a list **where**  
*list-of-B* Leaf = [] |  
*list-of-B* (Node l a r) = a # *list-of-B* (del-min (Node l a r))  
 ⟨proof⟩

**lemma** *list-of-B-braun-ptermination*:  
*braun t* ⇒ *list-of-B-dom t*  
 ⟨proof⟩

**lemmas** *list-of-B-braun-simps*  
 = *list-of-B.psimps*[OF *list-of-B-braun-ptermination*]

**lemma** *mset-list-of-B*:  
*braun t* ⇒ *mset (list-of-B t) = mset-tree t*  
 ⟨proof⟩

**lemma** *set-list-of-B*:  
*braun t* ⇒ *set (list-of-B t) = set-tree t*  
 ⟨proof⟩

**lemma** *sorted-list-of-B*:  
*braun t* ⇒ *heap t* ⇒ *sorted (list-of-B t)*  
 ⟨proof⟩

**definition**  
*heap-of-B xs = fst (heapify (length xs) xs)*

**lemma** *sortedB*: *sorted (list-of-B (heap-of-B xs))*  
 ⟨proof⟩

**lemma** *msetB*: *mset (list-of-B (heap-of-B xs)) = mset xs*  
 ⟨proof⟩

**fun** *t-del-left* :: 'a tree ⇒ nat **where**  
*t-del-left* (Node Leaf x r) = 1 |  
*t-del-left* (Node l x r) = (let (y,l') = del-left l in 2 + *t-del-left* l)

**fun** *t-del-min* :: 'a::linorder tree ⇒ nat **where**  
*t-del-min* Leaf = 0 |  
*t-del-min* (Node Leaf x r) = 0 |  
*t-del-min* (Node l x r) = (let (y,l') = del-left l in *t-del-left* l + *t-sift-down* r y l')

**function** *t-list-of-B* :: ('a::linorder) tree ⇒ nat **where**

$t\text{-list-of-}B \text{ Leaf} = 0 \mid$   
 $t\text{-list-of-}B (\text{Node } l \ a \ r) = 1 + t\text{-del-min } (\text{Node } l \ a \ r) + t\text{-list-of-}B (\text{del-min } (\text{Node } l \ a \ r))$   
 ⟨proof⟩

**lemma** *t-del-left-bound*:

$t \neq \text{Leaf} \implies t\text{-del-left } t \leq 2 * \text{height } t$   
 ⟨proof⟩

**lemma** *del-left-height*:

$\text{del-left } t = (v, t') \implies t \neq \text{Leaf} \implies \text{height } t' \leq \text{height } t$   
 ⟨proof⟩

**lemma** *t-del-min-bound*:

$\text{braun } t \implies t\text{-del-min } t \leq 3 * \text{height } t$   
 ⟨proof⟩

**lemma** *t-list-of-B-braun-ptermination*:

$\text{braun } t \implies t\text{-list-of-}B\text{-dom } t$   
 ⟨proof⟩

**lemmas** *t-list-of-B-braun-simps*

$= t\text{-list-of-}B.\text{psimps}[OF \ t\text{-list-of-}B\text{-braun-ptermination}]$

**lemma** *del-min-height*:

$\text{braun } t \implies \text{height } (\text{del-min } t) \leq \text{height } t$   
 ⟨proof⟩

**lemma** *t-list-of-B-induct*:

$\text{braun } t \implies \text{height } t \leq n \implies t\text{-list-of-}B \ t \leq 3 * (n + 1) * \text{size } t$   
 ⟨proof⟩

**lemma** *t-list-of-B-bound*:

$\text{braun } t \implies t\text{-list-of-}B \ t \leq 3 * (\text{height } t + 1) * \text{size } t$   
 ⟨proof⟩

**lemma** *t-list-of-B-log-bound*:

$\text{braun } t \implies t\text{-list-of-}B \ t \leq 3 * (n \log 2 (\text{size } t + 1) + 1) * \text{size } t$   
 ⟨proof⟩

**definition**

$t\text{-heap-of-}B \ xs = \text{length } xs + t\text{-heapify } (\text{length } xs) \ xs$

**lemma** *t-heap-of-B-bound*:

$t\text{-heap-of-}B \ xs \leq 6 * \text{length } xs + 1$   
 ⟨proof⟩

**lemmas** *size-heapify = arg-cong[OF mset-heapify, where f=size, simplified]*

**theorem** *t-sortB*:

*t-heap-of-B xs + t-list-of-B (heap-of-B xs)*  
 $\leq 3 * \text{length } xs * (\text{nlog2 } (\text{length } xs + 1) + 3) + 1$   
(**is** ?lhs  $\leq$  -)  
<proof>

**end**

## References

- [1] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Technology, 1983.
- [2] R. R. Hoogerwoord. A logarithmic implementation of flexible arrays. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction, Second International Conference*, volume 669 of *LNCS*, pages 191–207. Springer, 1992.
- [3] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.