Priority Queues Based on Braun Trees

Tobias Nipkow

February 23, 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

1 Priority Queues Based on Braun Trees
   1.1 Introduction .............................................. 2
   1.2 Get Minimum ........................................... 2
   1.3 Insertion .................................................. 2
   1.4 Deletion ................................................... 2

2 Priority Queues Based on Braun Trees 2
   2.1 Function del-min2 ....................................... 5
   2.2 Correctness Proof ....................................... 5

3 Sorting via Priority Queues Based on Braun Trees 6

4 Phase 1: List to Tree 7

5 Phase 2: Heap to List 10

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 \( r \), \( \text{size}(r) \leq \text{size}(l) \leq \text{size}(r) + 1 \)), 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).

\textbf{lemma} \( \text{height-size-braun} \): \( \text{braun } t \implies 2 ^ {\text{height } t} \leq 2 \ast \text{size } t + 1 \)

\langle proof \rangle

1.2 Get Minimum

\textbf{fun} get-min :: `'a::linorder tree ⇒ 'a where
get-min (Node l a r) = a

\textbf{lemma} get-min: \[ \text{heap } t; \ t \neq \text{Leaf } \] ⇒ get-min \( t \) = \( \text{Min-mset (mset-tree } t) \)

\langle proof \rangle

1.3 Insertion

\textbf{hide-const (open) insert}

\textbf{fun} insert :: `'a::linorder ⇒ 'a tree ⇒ 'a tree where
insert a Leaf = Node Leaf a Leaf |
insert a (Node l x r) =
  \( \text{if } a < x \text{ then } \text{Node (insert } x \text{ r) } a \text{ l else Node (insert } a \text{ r) } x \text{ l} \)

\textbf{lemma} size-insert[simp]: \( \text{size(insert } x \text{ t) } = \text{size } t + 1 \)

\langle proof \rangle

\textbf{lemma} mset-insert: \( \text{mset-tree (insert } x \text{ t) } = \{ \# x \# \} + \text{mset-tree } t \)

\langle proof \rangle

\textbf{lemma} set-insert[simp]: \( \text{set-tree (insert } x \text{ t) } = \{ x \} \cup \text{(set-tree } t) \)

\langle proof \rangle

\textbf{lemma} braun-insert: \( \text{braun } t \implies \text{braun(insert } x \text{ t) } \)

\langle proof \rangle

\textbf{lemma} heap-insert: \( \text{heap } t \implies \text{heap(insert } x \text{ t) } \)

\langle proof \rangle

1.4 Deletion

Slightly simpler definition of \textit{del-left} which avoids the need to appeal to the Braun invariant.
fun del-left :: 'a tree ⇒ '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') ⇒ t ≠ Leaf
⇒ mset-tree t = {#x#} + mset-tree t'
⟨proof⟩

lemma del-left-mset:
del-left t = (x,t') ⇒ t ≠ Leaf
⇒ x ∈# mset-tree t ∧ mset-tree t' = mset-tree t - {#x#}
⟨proof⟩

lemma del-left-set:
del-left t = (x,t') ⇒ t ≠ Leaf ⇒ set-tree t = {x} ∪ set-tree t'
⟨proof⟩

lemma del-left-heap:
del-left t = (x,t') ⇒ t ≠ Leaf ⇒ heap t ⇒ heap t'
⟨proof⟩

lemma del-left-size:
del-left t = (x,t') ⇒ t ≠ Leaf ⇒ size t = size t' + 1
⟨proof⟩

lemma del-left-braun:
del-left t = (x,t') ⇒ braun t ⇒ t ≠ Leaf ⇒ braun t'
⟨proof⟩

context includes pattern-aliases
begin

Slightly simpler definition: - instead of {} because of Braun invariant.

function (sequential) sift-down :: 'a:linorder tree ⇒ 'a ⇒ 'a tree where
sift-down Leaf a = - = Node Leaf a Leaf |
sift-down (Node Leaf x -) a Leaf =
  (if a ≤ 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 ≤ x1 ∧ a ≤ x2
   then Node l1 a t2
   else if x1 ≤ 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:
\[ \text{braun}(\text{Node } l \ a \ r) \implies \text{size}(\text{sift-down } l \ a \ r) = \text{size } l + \text{size } r + 1 \]
⟨proof⟩

lemma braun-sift-down:
\[ \text{braun}(\text{Node } l \ a \ r) \implies \text{braun}(\text{sift-down } l \ a \ r) \]
⟨proof⟩

lemma mset-sift-down:
\[ \text{braun}(\text{Node } l \ a \ r) \implies \text{mset-tree}(\text{sift-down } l \ a \ r) = \{a\} + (\text{mset-tree } l + \text{mset-tree } r) \]
⟨proof⟩

lemma set-sift-down:
\[ \text{braun}(\text{Node } l \ a \ r) \implies \text{set-tree}(\text{sift-down } l \ a \ r) = \{a\} \cup (\text{set-tree } l \cup \text{set-tree } r) \]
⟨proof⟩

lemma heap-sift-down:
\[ \text{braun}(\text{Node } l \ a \ r) \implies \text{heap } l \implies \text{heap } r \implies \text{heap}(\text{sift-down } l \ a \ r) \]
⟨proof⟩

fun del-min :: 'a::linorder tree ⇒ '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 ≠ 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 = λh. h = Leaf
and insert = insert and del-min = del-min
and get-min = get-min and invar = λh. braun h ∧ heap h
and mset = mset-tree
⟨proof⟩
end
2 Priority Queues Based on Braun Trees

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. 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 ⇒ 'a tree ⇒ bool where
le-root a t = (t = Leaf ∨ a ≤ value t)

fun replace-min :: 'a::linorder ⇒ 'a tree ⇒ '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 ⇒ 'a tree ⇒ 'a tree where
merge l Leaf = l |
merge (Node l1 a1 r1) (Node l2 a2 r2) = (if a1 ≤ 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) ⇒ 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:
\[ \text{braun}(\text{Node } l \times r) \implies \text{mset-tree}(\text{merge } l \times r) = \text{mset-tree } l + \text{mset-tree } r \]
(\proof)

lemma heap-merge:
\[ [\text{braun}(\text{Node } l \times r); \text{heap } l; \text{heap } r] \implies \text{heap}(\text{merge } l \times r) \]
(\proof)

lemma del-left-braun-size:
\[ \text{del-left } t = (x,t') \implies \text{braun } t \implies t \neq \text{Leaf} \implies \text{braun } t' \land \text{size } t = \text{size } t' + 1 \]
(\proof)

lemma braun-size-merge:
\[ \text{braun}(\text{Node } l \times r) \implies \text{braun}(\text{merge } l \times r) \land \text{size}(\text{merge } l \times r) = \text{size } l + \text{size } r \]
(\proof)

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

interpretation braun: Priority-Queue
where empty = Leaf and is-empty = \( \lambda h. h = \text{Leaf} \)
and insert = insert and del-min = del-min2
and get-min = get-min and invar = \( \lambda h. \text{braun } h \land \text{heap } h \)
and mset = mset-tree
(\proof)

end

3 Sorting via Priority Queues Based on Braun Trees

theory Sorting-Braun
imports Priority-Queue-Braun
begin


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

abbreviation (input)
\[ n \text{log2 } n == \text{nat}(\text{ceiling}(\log 2 n)) \]
4 Phase 1: List to Tree

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

```haskell
fun heap-of-A :: （'a::linorder）list ⇒ '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)  
(\textit{proof})

**lemma** braun-heap-of-A: braun (heap-of-A xs)  
(\textit{proof})

**lemma** mset-tree-heap-of-A: mset-tree (heap-of-A xs) = mset xs  
(\textit{proof})

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

```haskell
fun t-insert :: （'a::linorder）⇒ 'a tree ⇒ 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)
```

```haskell
fun t-heap-of-A :: （'a::linorder）list ⇒ 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 ≤ height t + 1  
(\textit{proof})

**lemma** height-insert-ge:  
height t ≤ height (insert x t)  
(\textit{proof})

**lemma** t-heap-of-A-bound:  
t-heap-of-A xs ≤ length xs * (height (heap-of-A xs) + 1)  
(\textit{proof})

**lemma** size-heap-of-A:  
size (heap-of-A xs) = length xs  
(\textit{proof})

**lemma** t-heap-of-A-log-bound:  
t-heap-of-A xs ≤ length xs * (nlog2 (length xs + 1) + 1)  
(\textit{proof})

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)

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

corpus includes pattern-aliases

corpus begins

corpus 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)
termination

fun t-heapify :: nat ⇒ ('a::linorder) list ⇒ 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) ⇒ t-sift-down l x r ≤ height (Node l x r)

lemma sift-down-height:
braun (Node l x r) ⇒ height (sift-down l x r) ≤ height (Node l x r)

lemma braun-height-r-le:
braun (Node l x r) ⇒ height r ≤ height l

lemma braun-height-l-le:
assumes b: braun (Node l x r)
shows height l ≤ Suc (height r)

lemma braun-height-node-eq:
assumes b: braun (Node l x r)
shows height (Node l x r) = Suc (height l)

lemma t-heapify-induct:
i ≤ length xs ⇒ t-heapify i xs + height (fst (heapify i xs)) ≤ 5 * i + 1

lemma t-heapify-bound:
i ≤ length xs ⇒ t-heapify i xs ≤ 5 * i + 1
5 Phase 2: Heap to List

Algorithm A extracts \((\text{list-of-A})\) the list by removing the root and merging the children:

**Lemma** \(\text{size-prod-measure}[\text{measure-function}]:\)

\[
is\text{-measure } f \implies is\text{-measure } g \implies is\text{-measure } (size\text{-prod } f \; g)
\]

\(\langle \text{proof} \rangle\)

**fun** \(\text{merge} :: ('a::linorder) \text{ tree } \Rightarrow 'a \text{ tree } \Rightarrow 'a \text{ tree} \text{ where}\)

\[
\text{merge } \text{Leaf } t2 = t2 \\
\text{merge } t1 \text{ Leaf } = t1 \\
\text{merge } (\text{Node } l1 \; a1 \; r1) \; (\text{Node } l2 \; a2 \; r2) = \\
\quad \text{ (if } a1 \leq a2 \text{ then } \text{Node } (\text{merge } l1 \; r1) \; a1 \; (\text{Node } l2 \; a2 \; r2) \\
\quad \quad \text{ else } \text{Node } (\text{Node } l1 \; a1 \; r1) \; a2 \; (\text{merge } l2 \; r2))
\]

\(\text{value} \; \text{merge } \langle\langle\rangle, \; 0::\text{int}, \; \langle\rangle\rangle \; \langle\langle\rangle, \; 0, \; \langle\rangle\rangle = \langle\langle\rangle, \; 0, \; \langle\rangle, \; \langle\rangle\rangle\rangle\)

**Lemma** \(\text{merge-size}[\text{termination-simp}]:\)

\[
\text{size } (\text{merge } l \; r) = \text{size } l + \text{size } r
\]

\(\langle \text{proof} \rangle\)

**fun** \(\text{list-of-A} :: ('a::linorder) \text{ tree } \Rightarrow 'a \text{ list} \text{ where}\)

\[
\text{list-of-A } \text{Leaf } = \langle\rangle \\
\text{list-of-A } (\text{Node } l \; a \; r) = a \; # \text{list-of-A } (\text{merge } l \; r)
\]

\(\text{value} \; \text{list-of-A } (\text{heap-of-A } \text{shuffle100})\)

**Lemma** \(\text{set-tree-merge}[\text{simp}]:\)

\[
\text{set-tree } (\text{merge } l \; r) = \text{set-tree } l \cup \text{set-tree } r
\]

\(\langle \text{proof} \rangle\)

**Lemma** \(\text{mset-tree-merge}[\text{simp}]:\)

\[
\text{mset-tree } (\text{merge } l \; r) = \text{mset-tree } l + \text{mset-tree } r
\]

\(\langle \text{proof} \rangle\)

**Lemma** \(\text{merge-heap}:\)

\[
\text{heap } l \implies \text{heap } r \implies \text{heap } (\text{merge } l \; r)
\]

\(\langle \text{proof} \rangle\)

**Lemma** \(\text{set-list-of-A}[\text{simp}]:\)

\[
\text{set } (\text{list-of-A } t) = \text{set-tree } t
\]

\(\langle \text{proof} \rangle\)

**Lemma** \(\text{mset-list-of-A}[\text{simp}]:\)

\[
\text{mset } (\text{list-of-A } t) = \text{mset-tree } t
\]

\(\langle \text{proof} \rangle\)

**Lemma** \(\text{sorted-list-of-A}:\)
heap $t \rightarrow \text{sorted (list-of-A } t) \langle \text{proof} \rangle$

\textbf{lemma sortedA:} \text{sorted (list-of-A (heap-of-A } xs)) \langle \text{proof} \rangle

\textbf{lemma msetA:} \text{mset (list-of-A (heap-of-A } xs)) = \text{mset } xs \langle \text{proof} \rangle

Does \text{list-of-A} take time $O(nlgn)$? Although \text{merge} does not preserve \text{braun}, it cannot increase the height of the heap.

\textbf{lemma merge-height:}
\text{height (merge } l r) \leq \text{Suc (max (height } l) (height } r)) \langle \text{proof} \rangle

\textbf{corollary merge-height-display:}
\text{height (merge } l r) \leq \text{height (Node } l x r) \langle \text{proof} \rangle

\textbf{fun} \text{t-merge ::} \ ('a::linorder) \text{ tree } \Rightarrow \ 'a \text{ tree } \Rightarrow \ \text{nat where}
\text{t-merge Leaf } t 2 = 0 |
\text{t-merge } t l \text{ Leaf } = 0 |
\text{t-merge (Node } l l a 1 r l) \text{ (Node } l 2 a 2 r 2) =
\text{ (if } a 1 \leq a 2 \text{ then } 1 + \text{t-merge } l l r 1 \\
\text{ else } 1 + \text{t-merge } l 2 r 2)

\textbf{fun} \text{t-list-of-A ::} \ ('a::linorder) \text{ tree } \Rightarrow \ \text{nat where}
\text{t-list-of-A Leaf } = 0 |
\text{t-list-of-A (Node } l a r) = 1 + \text{t-merge } l r + \text{t-list-of-A (merge } l r)

\textbf{lemma t-merge-height:}
\text{t-merge } l r \leq \text{max (height } l) (height r) \langle \text{proof} \rangle

\textbf{lemma t-list-of-A-induct:}
\text{height } t \leq n \Rightarrow \text{t-list-of-A } t \leq 2 * n * \text{size } t \langle \text{proof} \rangle

\textbf{lemma t-list-of-A-bound:}
\text{t-list-of-A } t \leq 2 * \text{height } t * \text{size } t \langle \text{proof} \rangle

\textbf{lemma t-list-of-A-log-bound:}
\text{braun } t \Rightarrow \text{t-list-of-A } t \leq 2 * \text{nlog2 (size } t + 1) * \text{size } t \langle \text{proof} \rangle

\textbf{value} \text{t-list-of-A (heap-of-A shuffle100)}

\textbf{theorem t-sortA:}
\[ t\text{-}heap\text{-}of\text{-}A\ \text{xs} + t\text{-}list\text{-}of\text{-}A\ (\text{heap}\text{-}of\text{-}A\ \text{xs}) \leq 3 \ast \text{length}\ \text{xs} \ast (n\log_2 (\text{length}\ \text{xs} + 1)) + 1 \]

(\text{is } ?\text{lhs} \leq -)

(\text{proof})

Running time of algorithm B:

\textbf{function} list-of-B :: ('a::linorder) tree \Rightarrow 'a list
\textbf{where}

\begin{align*}
\text{list-of-B Leaf} &= [] \\
\text{list-of-B (Node l a r)} &= a \# \text{list-of-B (del-min (Node l a r))}
\end{align*}

(\text{proof})

\textbf{lemma} list-of-B-braun-termination:

\begin{align*}
\text{braun t} &\implies \text{list-of-B-dom t} \\
\end{align*}

(\text{proof})

\textbf{lemmas} list-of-B-braun-simps

\begin{align*}
&= \text{list-of-B.psimps[OF list-of-B-braun-termination]}
\end{align*}

\textbf{lemma} mset-list-of-B:

\begin{align*}
\text{braun t} &\implies \text{mset (list-of-B t)} = \text{mset-tree t} \\
\end{align*}

(\text{proof})

\textbf{lemma} set-list-of-B:

\begin{align*}
\text{braun t} &\implies \text{set (list-of-B t)} = \text{set-tree t} \\
\end{align*}

(\text{proof})

\textbf{lemma} sorted-list-of-B:

\begin{align*}
\text{braun t} &\implies \text{heap t} \implies \text{sorted (list-of-B t)} \\
\end{align*}

(\text{proof})

\textbf{definition}

\text{heap-of-B xs} = \text{fst (heapify (length xs) xs)}

\textbf{lemma} sortedB: \text{sorted (list-of-B (heap-of-B xs))}

(\text{proof})

\textbf{lemma} msetB: \text{mset (list-of-B (heap-of-B xs))} = \text{mset xs}

(\text{proof})

\textbf{fun} t-del-left :: 'a tree \Rightarrow nat
\textbf{where}

\begin{align*}
\text{t-del-left (Node Leaf x r)} &= 1 \\
\text{t-del-left (Node l x r)} &= (\text{let } (y,l') = \text{del-left l in } 2 + \text{t-del-left l})
\end{align*}

\textbf{fun} t-del-min :: 'a::linorder tree \Rightarrow nat
\textbf{where}

\begin{align*}
\text{t-del-min Leaf} &= 0 \\
\text{t-del-min (Node Leaf x r)} &= 0 \\
\text{t-del-min (Node l x r)} &= (\text{let } (y,l') = \text{del-left l in } \text{t-del-left l} + \text{t-sift-down r y l'})
\end{align*}

\textbf{function} t-list-of-B :: ('a::linorder) tree \Rightarrow nat
\textbf{where}
\begin{align*}
\text{t-list-of-B Leaf} &= 0 \\
\text{t-list-of-B (Node l a r)} &= 1 + \text{t-del-min (Node l a r)} + \text{t-list-of-B (del-min (Node l a r))}
\end{align*}

\textit{proof}

\textbf{lemma} \texttt{t-del-left-bound}:
\[ t \neq \text{Leaf} \implies \text{t-del-left } t \leq 2 \ast \text{height } t \]

\textit{proof}

\textbf{lemma} \texttt{del-left-height}:
\[ \text{del-left } t = (v, t') \implies t \neq \text{Leaf} \implies \text{height } t' \leq \text{height } t \]

\textit{proof}

\textbf{lemma} \texttt{t-del-min-bound}:
\[ \text{braun } t \implies \text{t-del-min } t \leq 3 \ast \text{height } t \]

\textit{proof}

\textbf{lemma} \texttt{t-list-of-B-braun-ptermination}:
\[ \text{braun } t \implies \text{t-list-of-B-dom } t \]

\textit{proof}

\textbf{lemmas} \texttt{t-list-of-B-braun-simps}
\[ = \text{t-list-of-B-psimps[OF t-list-of-B-braun-ptermination]} \]

\textbf{lemma} \texttt{del-min-height}:
\[ \text{braun } t \implies \text{height (del-min } t \leq \text{height } t \]

\textit{proof}

\textbf{lemma} \texttt{t-list-of-B-induct}:
\[ \text{braun } t \implies \text{height } t \leq n \implies \text{t-list-of-B } t \leq 3 \ast (n + 1) \ast \text{size } t \]

\textit{proof}

\textbf{lemma} \texttt{t-list-of-B-bound}:
\[ \text{braun } t \implies \text{t-list-of-B } t \leq 3 \ast (\text{height } t + 1) \ast \text{size } t \]

\textit{proof}

\textbf{lemma} \texttt{t-list-of-B-log-bound}:
\[ \text{braun } t \implies \text{t-list-of-B } t \leq 3 \ast (\text{nlog2 (size } t + 1) + 1) \ast \text{size } t \]

\textit{proof}

\textbf{definition}
\[ \text{t-heap-of-B } xs = \text{length } xs + \text{t-heapify (length } xs) \]

\textbf{lemma} \texttt{t-heap-of-B-bound}:
\[ \text{t-heap-of-B } xs \leq 6 \ast \text{length } xs + 1 \]

\textit{proof}

\textbf{lemmas} \texttt{size-heapify = arg-cong[OF mset-heapify, where f=size, simplified]}
\textbf{theorem} \textit{t-sortB}: \\
\textit{t-heap-of-B} \textit{xs} + \textit{t-list-of-B} (\textit{heap-of-B} \textit{xs}) \\
\leq 3 \times \textit{length} \textit{xs} \times (n\log_2 (\textit{length} \textit{xs} + 1) + 3) + 1 \\
(is \ ?\textit{lhs} \leq \_) \\
(proof) \\
end

\textbf{References}

