Priority Queues Based on Braun Trees

Tobias Nipkow

March 17, 2025

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

theory *Priority-Queue-Braun*
imports
  HOL−Library.Tree-Multiset
  HOL−Library.Pattern-Aliases
  HOL−Data-Structures.Priority-Queue-Specs
  HOL−Data-Structures.Braun-Tree
  HOL−Data-Structures.Define-Time-Function
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 Paulsons'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*: $braun\ t \Longrightarrow 2\ \widehat{}\ (height\ t) \leq 2 * size\ t + 1$
⟨*proof*⟩

1.2 Get Minimum

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

lemma *get-min*: $[\![\ heap\ t;\ \ t \neq Leaf\ ]\!] \Longrightarrow get\text{-}min\ t = Min\text{-}mset\ (mset\text{-}tree\ t)$
⟨*proof*⟩

1.3 Insertion

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

fun *insert* :: *'a::linorder* ⇒ *'a tree* ⇒ *'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*} ∪ (*set-tree t*)
⟨*proof*⟩

lemma *braun-insert*: $braun\ t \Longrightarrow braun(insert\ x\ t)$
⟨*proof*⟩

lemma *heap-insert*: $heap\ t \Longrightarrow heap(insert\ x\ t)$
⟨*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* ⇒ *'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'*) ⟹ *t* ≠ *Leaf* ⟹ *braun t* ⟹ *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* ⇒ *'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 t1 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*:
$braun(Node\ l\ a\ r) \Longrightarrow size(sift\text{-}down\ l\ a\ r) = size\ l + size\ r + 1$
$\langle proof \rangle$

lemma *braun-sift-down*:
$braun(Node\ l\ a\ r) \Longrightarrow braun(sift\text{-}down\ l\ a\ r)$
$\langle proof \rangle$

lemma *mset-sift-down*:
$braun(Node\ l\ a\ r) \Longrightarrow mset\text{-}tree(sift\text{-}down\ l\ a\ r) = \{\#a\#\} + (mset\text{-}tree\ l + mset\text{-}tree\ r)$
$\langle proof \rangle$

lemma *set-sift-down*: $braun(Node\ l\ a\ r)$
$\Longrightarrow set\text{-}tree(sift\text{-}down\ l\ a\ r) = \{a\} \cup (set\text{-}tree\ l \cup set\text{-}tree\ r)$
$\langle proof \rangle$

lemma *heap-sift-down*:
$braun(Node\ l\ a\ r) \Longrightarrow heap\ l \Longrightarrow heap\ r \Longrightarrow heap(sift\text{-}down\ l\ a\ r)$
$\langle proof \rangle$

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*: $braun\ t \Longrightarrow braun(del\text{-}min\ t)$
$\langle proof \rangle$

lemma *heap-del-min*: $heap\ t \Longrightarrow braun\ t \Longrightarrow heap(del\text{-}min\ t)$
$\langle proof \rangle$

lemma *size-del-min*: **assumes** *braun t* **shows** $size(del\text{-}min\ t) = size\ t - 1$
$\langle proof \rangle$

lemma *mset-del-min*: **assumes** *braun t* $t \neq Leaf$
shows $mset\text{-}tree(del\text{-}min\ t) = mset\text{-}tree\ t - \{\#get\text{-}min\ t\#\}$
$\langle proof \rangle$

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*
$\langle proof \rangle$

1.5 Running Time Analysis

time-fun *insert*

lemma *T-insert*: *T-insert a t* ≤ *height t* + *1*
⟨*proof*⟩

time-fun *del-left*

lemma *T-del-left-height*: *t* ≠ *Leaf* ⟹ *T-del-left t* ≤ *height t*
⟨*proof*⟩

time-function *sift-down*
termination
⟨*proof*⟩

lemma *T-sift-down-height*: *braun*(*Node l a r*) ⟹ *T-sift-down l x r* ≤ *max*(*height l*) (*height r*) + *1*
⟨*proof*⟩

time-fun *del-min*

lemma *del-left-height*: ⟦ *del-left t* = (*x*, *t'*); *t* ≠ ⟨⟩ ⟧ ⟹ *height t'* ≤ *height t*
⟨*proof*⟩

lemma *T-del-min-neq-Leaf*: *l* ≠ *Leaf* ⟹
T-del-min (*Node l x r*) = *T-del-left l* + (*let* (*y*,*l'*) = *del-left l in T-sift-down r y l'*)
⟨*proof*⟩

lemma *T-del-min*: **assumes** *braun t* **shows** *T-del-min t* ≤ *2*∗*height t*
⟨*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. 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**

$le\text{-}root\ a\ t = (t = Leaf \vee a \leq value\ t)$

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*:
  braun (*Node l x r*) ⟹ *mset-tree*(*merge l r*) = *mset-tree l* + *mset-tree r*
⟨*proof*⟩

lemma *heap-merge*:
  ⟦ *braun* (*Node l x r*); *heap l*; *heap r* ⟧ ⟹ *heap*(*merge l r*)
⟨*proof*⟩

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

lemma *braun-size-merge*:
  braun (*Node l x r*) ⟹ *braun*(*merge l r*) ∧ *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 $O(nlgn)$ fashion and generates a Braun tree:

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*)
⟨*proof*⟩

lemma *braun-heap-of-A*: *braun* (*heap-of-A xs*)

⟨*proof*⟩

lemma *mset-tree-heap-of-A*: *mset-tree* (*heap-of-A xs*) = *mset xs*
⟨*proof*⟩

Running time is n*log n, which we can approximate with height.

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*)

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*
⟨*proof*⟩

lemma *height-insert-ge*:
height t ≤ *height* (*insert x t*)
⟨*proof*⟩

lemma *t-heap-of-A-bound*:
t-heap-of-A xs ≤ *length xs* ∗ (*height* (*heap-of-A xs*) + *1*)
⟨*proof*⟩

lemma *size-heap-of-A*:
size (*heap-of-A xs*) = *length xs*
⟨*proof*⟩

lemma *t-heap-of-A-log-bound*:
t-heap-of-A xs ≤ *length xs* ∗ (*nlog2* (*length xs* + *1*) + *1*)
⟨*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.

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* ⇒ (*'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*)
⟨*proof*⟩

lemma *sift-down-height*:
braun (*Node l x r*) ⟹ *height* (*sift-down l x r*) ≤ *height* (*Node l x r*)
⟨*proof*⟩

lemma *braun-height-r-le*:
braun (*Node l x r*) ⟹ *height r* ≤ *height l*
⟨*proof*⟩

lemma *braun-height-l-le*:
assumes *b*: *braun* (*Node l x r*)
shows *height l* ≤ *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 ≤ *length xs* ⟹ *t-heapify i xs* + *height* (*fst* (*heapify i xs*)) ≤ *5* ∗ *i* + *1*
⟨*proof*⟩

lemma *t-heapify-bound*:
i ≤ *length xs* ⟹ *t-heapify i xs* ≤ *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-measure f ⟹ *is-measure g* ⟹ *is-measure* (*size-prod f g*)
⟨*proof*⟩

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

value *merge* ⟨⟨⟩, *0*::*int*, ⟨⟩⟩ ⟨⟨⟩, *0*, ⟨⟩⟩ = ⟨⟨⟩, *0*, ⟨⟨⟩, *0*, ⟨⟩⟩⟩

lemma *merge-size*[*termination-simp*]:
size (*merge l r*) = *size l* + *size r*
⟨*proof*⟩

fun *list-of-A* :: (*'a*::*linorder*) *tree* ⇒ *'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-tree (*merge l r*) = *set-tree l* ∪ *set-tree r*
⟨*proof*⟩

lemma *mset-tree-merge*[*simp*]:
mset-tree (*merge l r*) = *mset-tree l* + *mset-tree r*
⟨*proof*⟩

lemma *merge-heap*:
heap l ⟹ *heap r* ⟹ *heap* (*merge l r*)
⟨*proof*⟩

lemma *set-list-of-A*[*simp*]:
set (*list-of-A t*) = *set-tree t*
⟨*proof*⟩

lemma *mset-list-of-A*[*simp*]:
mset (*list-of-A t*) = *mset-tree t*
⟨*proof*⟩

lemma *sorted-list-of-A*:
heap t ⟹ *sorted* (*list-of-A t*)
⟨*proof*⟩

lemma *sortedA*: *sorted* (*list-of-A* (*heap-of-A xs*))
⟨*proof*⟩

lemma *msetA*: *mset* (*list-of-A* (*heap-of-A xs*)) = *mset xs*
⟨*proof*⟩

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

lemma *merge-height*:
height (*merge l r*) ≤ *Suc* (*max* (*height l*) (*height r*))
⟨*proof*⟩

corollary *merge-height-display*:
  height (*merge l r*) ≤ *height* (*Node l x r*)
  ⟨*proof*⟩

fun *t-merge* :: (*'a::linorder*) *tree* ⇒ *'a tree* ⇒ *nat* **where**
t-merge Leaf t2 = *0* |
t-merge t1 Leaf = *0* |
t-merge (*Node l1 a1 r1*) (*Node l2 a2 r2*) =
  (*if a1* ≤ *a2 then 1* + *t-merge l1 r1*
   else 1 + *t-merge l2 r2*)

fun *t-list-of-A* :: (*'a::linorder*) *tree* ⇒ *nat* **where**
t-list-of-A Leaf = *0* |
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 ≤ *max* (*height l*) (*height r*)
  ⟨*proof*⟩

lemma *t-list-of-A-induct*:
  height t ≤ *n* ⟹ *t-list-of-A t* ≤ *2* ∗ *n* ∗ *size t*
  ⟨*proof*⟩

lemma *t-list-of-A-bound*:
  t-list-of-A t ≤ *2* ∗ *height t* ∗ *size t*
  ⟨*proof*⟩

lemma *t-list-of-A-log-bound*:
  braun t ⟹ *t-list-of-A t* ≤ *2* ∗ *nlog2* (*size t* + *1*) ∗ *size t*
  ⟨*proof*⟩

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

theorem *t-sortA*:
  t-heap-of-A xs + *t-list-of-A* (*heap-of-A xs*) ≤ *3* ∗ *length xs* ∗ (*nlog2* (*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 $\Longrightarrow$ *mset* (*list-of-B t*) = *mset-tree t*
⟨*proof*⟩

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

lemma *sorted-list-of-B*:
braun t $\Longrightarrow$ *heap t* $\Longrightarrow$ *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-list-of-B Leaf = *0* |
t-list-of-B (*Node l a r*) = *1* + *t-del-min* (*Node l a r*) + *t-list-of-B* (*del-min* (*Node l a r*))
⟨*proof*⟩

lemma *t-del-left-bound*:
$t \neq$ *Leaf* $\Longrightarrow$ *t-del-left t* $\leq$ *2* ∗ *height t*
⟨*proof*⟩

lemma *del-left-height*:
del-left t = (*v*, *t'*) $\Longrightarrow$ $t \neq$ *Leaf* $\Longrightarrow$ *height t'* $\leq$ *height t*
⟨*proof*⟩

lemma *t-del-min-bound*:
braun t $\Longrightarrow$ *t-del-min t* $\leq$ *3* ∗ *height t*

⟨*proof*⟩

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

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

lemma *del-min-height*:
braun t ⟹ *height* (*del-min t*) ≤ *height t*
⟨*proof*⟩

lemma *t-list-of-B-induct*:
braun t ⟹ *height t* ≤ *n* ⟹ *t-list-of-B t* ≤ *3* ∗ (*n* + *1*) ∗ *size t*
⟨*proof*⟩

lemma *t-list-of-B-bound*:
braun t ⟹ *t-list-of-B t* ≤ *3* ∗ (*height t* + *1*) ∗ *size t*
⟨*proof*⟩

lemma *t-list-of-B-log-bound*:
braun t ⟹ *t-list-of-B t* ≤ *3* ∗ (*nlog2* (*size t* + *1*) + *1*) ∗ *size t*
⟨*proof*⟩

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

lemma *t-heap-of-B-bound*:
t-heap-of-B xs ≤ *6* ∗ *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*)
≤ *3* ∗ *length xs* ∗ (*nlog2* (*length xs* + *1*) + *3*) + *1*
(**is** *?lhs* ≤ -)
⟨*proof*⟩

end

References

[1] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Techology, 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.