

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

1	Priority Queues Based on Braun Trees	1
1.1	Introduction	2
1.2	Get Minimum	2
1.3	Insertion	2
1.4	Deletion	3
2	Priority Queues Based on Braun Trees 2	6
2.1	Function <i>del-min2</i>	6
2.2	Correctness Proof	7
3	Sorting via Priority Queues Based on Braun Trees	9
4	Phase 1: List to Tree	9
5	Phase 2: Heap to List	14

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(*induction t*)

case (*Node t1*)

show *?case*

proof (*cases height t1*)

case 0 **thus** *?thesis using Node by simp*

next

case (*Suc n*)

hence $2^{\wedge}n \leq size\ t1$ **using** *Node by simp*

thus *?thesis using Suc Node by(auto simp: max-def)*

qed

qed *simp*

1.2 Get Minimum

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

get-min (*Node l a r*) = a

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

by (*auto simp add: eq-Min-iff neq-Leaf-iff*)

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]*: $size(insert\ x\ t) = size\ t + 1$

by(*induction t arbitrary: x*) *auto*

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

by(*induction t arbitrary: x*) (*auto simp: ac-simps*)

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

by(*simp add: mset-insert flip: set-mset-tree*)

lemma *braun-insert*: $\text{braun } t \implies \text{braun}(\text{insert } x \ t)$
by(*induction t arbitrary: x*) *auto*

lemma *heap-insert*: $\text{heap } t \implies \text{heap}(\text{insert } x \ t)$
by(*induction t arbitrary: x*) (*auto simp add: ball-Un*)

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*:
 $\text{del-left } t = (x,t') \implies t \neq \text{Leaf}$
 $\implies \text{mset-tree } t = \{\#x\# \} + \text{mset-tree } t'$
by (*induction t arbitrary: x t' rule: del-left.induct;*
auto split: prod.splits)

lemma *del-left-mset*:
 $\text{del-left } t = (x,t') \implies t \neq \text{Leaf}$
 $\implies x \in \# \text{mset-tree } t \wedge \text{mset-tree } t' = \text{mset-tree } t - \{\#x\# \}$
by (*simp add: del-left-mset-plus*)

lemma *del-left-set*:
 $\text{del-left } t = (x,t') \implies t \neq \text{Leaf} \implies \text{set-tree } t = \{x\} \cup \text{set-tree } t'$
by(*simp add: del-left-mset-plus flip: set-mset-tree*)

lemma *del-left-heap*:
 $\text{del-left } t = (x,t') \implies t \neq \text{Leaf} \implies \text{heap } t \implies \text{heap } t'$
by (*induction t arbitrary: x t' rule: del-left.induct;*
fastforce split: prod.splits dest: del-left-set[THEN equalityD2])

lemma *del-left-size*:
 $\text{del-left } t = (x,t') \implies t \neq \text{Leaf} \implies \text{size } t = \text{size } t' + 1$
by(*induction t arbitrary: x t' rule: del-left.induct;*
auto split: prod.splits)

lemma *del-left-braun*:
 $\text{del-left } t = (x,t') \implies t \neq \text{Leaf} \implies \text{braun } t \implies \text{braun } t'$
by(*induction t arbitrary: x t' rule: del-left.induct;*
auto split: prod.splits dest: del-left-size)

context *includes pattern-aliases*
begin

Slightly simpler definition: - instead of $\langle \rangle$ 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 ≤ 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))
by pat-completeness auto
termination
by (relation measure (%(l,a,r). size l + size r)) auto

end

```

```

lemma size-sift-down:
  braun(Node l a r) ⇒ size(sift-down l a r) = size l + size r + 1
by(induction l a r rule: sift-down.induct) (auto simp: Let-def)

```

```

lemma braun-sift-down:
  braun(Node l a r) ⇒ braun(sift-down l a r)
by(induction l a r rule: sift-down.induct) (auto simp: size-sift-down Let-def)

```

```

lemma mset-sift-down:
  braun(Node l a r) ⇒ mset-tree(sift-down l a r) = {#a#} + (mset-tree l +
mset-tree r)
by(induction l a r rule: sift-down.induct) (auto simp: ac-simps Let-def)

```

```

lemma set-sift-down: braun(Node l a r)
  ⇒ set-tree(sift-down l a r) = {a} ∪ (set-tree l ∪ set-tree r)
by(drule arg-cong[where f=set-mset, OF mset-sift-down]) (simp)

```

```

lemma heap-sift-down:
  braun(Node l a r) ⇒ heap l ⇒ heap r ⇒ heap(sift-down l a r)
by (induction l a r rule: sift-down.induct) (auto simp: set-sift-down ball-Un Let-def)

```

```

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 ⇒ braun(del-min t)
apply(cases t rule: del-min.cases)
  apply simp
  apply simp
apply (fastforce split: prod.split intro!: braun-sift-down
dest: del-left-size del-left-braun)
done

```

```

lemma heap-del-min: heap t  $\implies$  braun t  $\implies$  heap(del-min t)
apply(cases t rule: del-min.cases)
  apply simp
  apply simp
apply (fastforce split: prod.split intro!: heap-sift-down
  dest: del-left-size del-left-braun del-left-heap)
done

```

```

lemma size-del-min: assumes braun t shows size(del-min t) = size t - 1
proof(cases t rule: del-min.cases)
  case [simp]: ( $\exists$  ll b lr a r)
  { fix y l' assume del-left (Node ll b lr) = (y,l')
    hence size(sift-down r y l') = size t - 1 using assms
    by(subst size-sift-down) (auto dest: del-left-size del-left-braun) }
  thus ?thesis by(auto split: prod.split)
qed (insert assms, auto)

```

```

lemma mset-del-min: assumes braun t t  $\neq$  Leaf
shows mset-tree(del-min t) = mset-tree t - {#get-min t#}
proof(cases t rule: del-min.cases)
  case 1 with assms show ?thesis by simp
next
  case 2 with assms show ?thesis by (simp)
next
  case [simp]: ( $\exists$  ll b lr a r)
  have mset-tree(sift-down r y l') = mset-tree t - {#a#}
  if del: del-left (Node ll b lr) = (y,l') for y l'
  using assms del-left-mset[OF del] del-left-size[OF del]
  del-left-braun[OF del] del-left-mset-plus[OF del]
  apply (subst mset-sift-down)
  apply (auto simp: ac-simps del-left-mset-plus[OF del])
  done
  thus ?thesis by(auto split: prod.split)
qed

```

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-min
and get-min = get-min and invar =  $\lambda h. \text{braun } h \wedge \text{heap } h$ 
and mset = mset-tree
proof(standard, goal-cases)
  case 1 show ?case by simp
next
  case 2 show ?case by simp
next
  case 3 show ?case by(simp add: mset-insert)
next
  case 4 thus ?case by(simp add: mset-del-min)

```

```

next
  case 5 thus ?case using get-min mset-tree.simps(1) by blast
next
  case 6 thus ?case by(simp)
next
  case 7 thus ?case by(simp add: heap-insert braun-insert)
next
  case 8 thus ?case by(simp add: heap-del-min braun-del-min)
qed

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 Filiâtre 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*: $\text{braun } (\text{Node } l \ a \ r) \implies \text{replace-min } x \ (\text{Node } l \ a \ r) = \text{sift-down } l \ x \ r$
by(*induction* $l \ x \ r$ *rule*: *sift-down.induct*)(*auto*)

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*:
 $\text{braun } (\text{Node } l \ x \ r) \implies \text{mset-tree}(\text{merge } l \ r) = \text{mset-tree } l + \text{mset-tree } r$
by(*induction* $l \ r$ *rule*: *merge.induct*)
(*auto simp*: *Let-def tree.set-sel*(2) *mset-sift-down replace-min-sift-down simp del*: *replace-min.simps dest!*: *del-left-mset split!*: *prod.split*)

lemma *heap-merge*:
 $\llbracket \text{braun } (\text{Node } l \ x \ r); \text{heap } l; \text{heap } r \rrbracket \implies \text{heap}(\text{merge } l \ r)$
proof(*induction* $l \ r$ *rule*: *merge.induct*)
case 1 **thus** ?*case* **by** *simp*
next
case (2 $l1 \ a1 \ r1 \ l2 \ a2 \ r2$)
show ?*case*
proof *cases*
assume $a1 \leq a2$
thus ?*thesis* **using** 2 **by**(*auto simp*: *ball-Un mset-tree-merge simp flip*: *set-mset-tree*)
next
assume $\neg a1 \leq a2$
let ? $l = \text{Node } l1 \ a1 \ r1$ **let** ? $r = \text{Node } l2 \ a2 \ r2$
have *braun* ? r **using** 2.prem(1) **by** *auto*
obtain $x \ l'$ **where** $dl: \text{del-left } ?l = (x, l')$ **by** (*metis surj-pair*)
from *del-left-heap*[*OF* *this* - 2.prem(2)] **have** *heap* l' **by** *auto*
have $hr: \text{heap}(\text{replace-min } x \ ?r)$ **using** $\langle \text{braun } ?r \rangle$ 2.prem(3)
by(*simp add*: *heap-sift-down neq-Leaf-iff replace-min-sift-down del*: *replace-min.simps*)
have 0: $\forall x \in \text{set-tree } ?l. a2 \leq x$ **using** 2.prem(2) $\langle \neg a1 \leq a2 \rangle$ **by** (*auto simp*: *ball-Un*)
moreover **have** $\text{set-tree } l' \subseteq \text{set-tree } ?l \ x \in \text{set-tree } ?l$
using *del-left-mset*[*OF* *dl*] **by** (*auto simp flip*: *set-mset-tree dest*:*in-diffD simp*:*union-iff*)
ultimately **have** 1: $\forall x \in \text{set-tree } l'. a2 \leq x$ **by** *blast*
have $\forall x \in \text{set-tree } ?r. a2 \leq x$ **using** $\langle \text{heap } ?r \rangle$ **by** *auto*
thus ?*thesis*
using $\langle \neg a1 \leq a2 \rangle$ $dl \ \langle \text{heap}(\text{replace-min } x \ ?r) \rangle \ \langle \text{heap } l' \rangle \ \langle x \in \text{set-tree } ?l \rangle$ 0 1
 $\langle \text{braun } ?r \rangle$
by(*auto simp*: *mset-sift-down replace-min-sift-down simp flip*: *set-mset-tree*)

```

      simp del: replace-min.simps)
qed
next
  case 3 thus ?case by simp
qed

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
by (simp add: del-left-braun del-left-size)

lemma braun-size-merge:
  braun (Node l x r)  $\implies$  braun(merge l r)  $\wedge$  size(merge l r) = size l + size r
apply(induction l r rule: merge.induct)
apply(auto simp: size-sift-down braun-sift-down replace-min-sift-down
  simp del: replace-min.simps
  dest!: del-left-braun-size split!: prod.split)
done

  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 =  $\lambda h. \text{braun } h \wedge \text{heap } h$ 
and mset = mset-tree
proof(standard, goal-cases)
  case 1 show ?case by simp
next
  case 2 show ?case by simp
next
  case 3 show ?case by(simp add: mset-insert)
next
  case 4 thus ?case by(auto simp: mset-tree-merge neq-Leaf-iff)
next
  case 5 thus ?case using get-min mset-tree.simps(1) by blast
next
  case 6 thus ?case by(simp)
next
  case 7 thus ?case by(simp add: heap-insert braun-insert)
next
  case 8 thus ?case by(auto simp: heap-merge braun-size-merge neq-Leaf-iff)
qed

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  $\Rightarrow$  '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)
by(induction xs)(simp-all add: heap-insert)
```

```
lemma braun-heap-of-A: braun (heap-of-A xs)
by(induction xs)(simp-all add: braun-insert)
```

```
lemma mset-tree-heap-of-A: mset-tree (heap-of-A xs) = mset xs
by(induction xs)(simp-all add: mset-insert)
```

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*:
 $t\text{-insert } x \ t \leq \text{height } t + 1$
apply (*induct t arbitrary: x; simp*)
apply (*simp only: max-Suc-Suc[symmetric] le-max-iff-disj, simp*)
done

lemma *height-insert-ge*:
 $\text{height } t \leq \text{height } (\text{insert } x \ t)$
apply (*induct t arbitrary: x; simp add: le-max-iff-disj*)
apply (*metis less-imp-le-nat less-le-trans not-le-imp-less*)
done

lemma *t-heap-of-A-bound*:
 $t\text{-heap-of-A } xs \leq \text{length } xs * (\text{height } (\text{heap-of-A } xs) + 1)$
proof (*induct xs*)
case (*Cons x xs*)

let *?lhs = t-insert x (heap-of-A xs) + t-heap-of-A xs*

have *?lhs ≤ ?lhs*

by *simp*

also note *Cons*

also note *height-insert-ge[of heap-of-A xs x]*

also note *t-insert-height[of x heap-of-A xs]*

finally show *?case*

apply *simp*

apply (*erule order-trans*)

apply (*simp add: height-insert-ge*)

done

qed *simp-all*

lemma *size-heap-of-A*:
 $\text{size } (\text{heap-of-A } xs) = \text{length } xs$
using *arg-cong[OF mset-tree-heap-of-A, of size xs]*
by *simp*

lemma *t-heap-of-A-log-bound*:
 $t\text{-heap-of-A } xs \leq \text{length } xs * (\text{nlog2 } (\text{length } xs + 1) + 1)$
using *t-heap-of-A-bound[of xs]*
acomplete-if-braun[OF braun-heap-of-A, of xs]
by (*simp add: height-acomplete size1-size size-heap-of-A*)

Algorithm B mimics heap sort more closely by building heaps bottom up in a balanced way:

fun *heapify* :: $\text{nat} \Rightarrow ('a::\text{linorder}) \text{ list} \Rightarrow 'a \text{ tree} * 'a \text{ list}$ **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:

$n \leq \text{length } xs \implies \text{snd } (\text{heapify } n \text{ } xs) = \text{drop } n \text{ } xs$
apply (induct xs arbitrary: n rule: measure-induct[where f=length])
apply (case-tac n; simp)
apply (clarsimp simp: Suc-le-length-iff case-prod-beta)
apply (rule arg-cong[where f= $\lambda n.$ drop n xs for xs])
apply simp
done

lemma heapify-snd-tup:

$\text{heapify } n \text{ } xs = (t, ys) \implies n \leq \text{length } xs \implies ys = \text{drop } n \text{ } xs$
by (drule heapify-snd, simp)

lemma heapify-correct:

$n \leq \text{length } xs \implies \text{heapify } n \text{ } xs = (t, ys) \implies$
 $\text{size } t = n \wedge \text{heap } t \wedge \text{braun } t \wedge \text{mset-tree } t = \text{mset } (\text{take } n \text{ } xs)$
proof (induct n xs arbitrary: t ys rule: heapify.induct)
case (2 n x xs)

note len = 2.prem(1)

obtain t1 ys1 **where** h1: heapify (Suc n div 2) xs = (t1, ys1)
by (simp add: prod-eq-iff)
obtain t2 ys2 **where** h2: heapify (n div 2) ys1 = (t2, ys2)
by (simp add: prod-eq-iff)

from len **have** le1: Suc n div 2 \leq length xs
by simp
note ys1 = heapify-snd-tup[OF h1 le1]
from len **have** le2: n div 2 \leq length ys1
by (simp add: ys1)

note app-hyps = 2.hyps(1)[OF le1 h1]
 2.hyps(2)[OF refl h1[symmetric], simplified, OF le2 h2]

hence braun: braun (Node t1 x t2)
by (simp, linarith)

have eq:
 $n \text{ div } 2 + \text{Suc } n \text{ div } 2 = n$
by simp

have msets:
 $\text{mset } (\text{take } (\text{Suc } n \text{ div } 2) \text{ } xs) + \text{mset } (\text{take } (n \text{ div } 2) \text{ } ys1) = \text{mset } (\text{take } n \text{ } xs)$

```

apply (subst append-take-drop-id[symmetric, where  $n = \text{Suc } n \text{ div } 2$  and  $t = \text{take } n \text{ xs}$ ],
      subst mset-append)
apply (simp add: take-drop min-absorb1 le1 eq ys1)
done

from 2.prem1 app-hyps msets show ?case
apply (clarsimp simp: h1 h2 le2)
apply (clarsimp simp: size-sift-down[OF braun]
      braun-sift-down[OF braun]
      mset-sift-down[OF braun])
apply (simp add: heap-sift-down[OF braun])
done
qed simp-all

```

```

lemma braun-heapify:
 $n \leq \text{length } xs \implies \text{braun } (\text{fst } (\text{heapify } n \text{ xs}))$ 
by (cases heapify n xs, drule(1) heapify-correct, simp)

```

```

lemma heap-heapify:
 $n \leq \text{length } xs \implies \text{heap } (\text{fst } (\text{heapify } n \text{ xs}))$ 
by (cases heapify n xs, drule(1) heapify-correct, simp)

```

```

lemma mset-heapify:
 $n \leq \text{length } xs \implies \text{mset-tree } (\text{fst } (\text{heapify } n \text{ xs})) = \text{mset } (\text{take } n \text{ xs})$ 
by (cases heapify n xs, drule(1) heapify-correct, simp)

```

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  $\Rightarrow$  'a  $\Rightarrow$  'a tree  $\Rightarrow$  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  $a \leq x1 \wedge a \leq x2$ 
   then 1
   else if  $x1 \leq x2$  then 1 + t-sift-down l1 a r1
   else 1 + t-sift-down l2 a r2)
by pat-completeness auto

```

```

termination
by (relation measure (%(l,a,r). size l + size r)) auto

```

```

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)
by (induct l x r rule: t-sift-down.induct; auto)

```

lemma sift-down-height:

```

braun (Node l x r)  $\Longrightarrow$  height (sift-down l x r)  $\leq$  height (Node l x r)
by (induct l x r rule: sift-down.induct; auto simp: Let-def)

```

lemma braun-height-r-le:

```

braun (Node l x r)  $\Longrightarrow$  height r  $\leq$  height l
by (rule acomplete-optimal, auto intro: acomplete-if-braun)

```

lemma braun-height-l-le:

```

assumes b: braun (Node l x r)
shows height l  $\leq$  Suc (height r)
using b acomplete-if-braun[OF b] min-height-le-height[of r]
by (simp add: acomplete-def)

```

lemma braun-height-node-eq:

```

assumes b: braun (Node l x r)
shows height (Node l x r) = Suc (height l)
using b braun-height-r-le[OF b]
by (auto simp add: max-def)

```

lemma t-heapify-induct:

```

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

```

proof (induct i xs rule: t-heapify.induct)

case (1 vs)

thus ?case

by simp

next

case (2 i x xs)

obtain l ys **where** h1: heapify (Suc i div 2) xs = (l, ys)

by (simp add: prod-eq-iff)

note hyps1 = 2.hyps[OF h1[symmetric] refl, simplified]

obtain r zs **where** h2: heapify (i div 2) ys = (r, zs)

by (simp add: prod-eq-iff)

from 2.prem1 heapify-snd-tup[OF h1]

```

have le1: Suc i div 2 ≤ length xs
  and le2: i div 2 ≤ length xs
  and le4: i div 2 ≤ length ys
  by simp-all

note hyps2 = hyps1(1)[OF le1] hyps1(2)[OF refl h2[symmetric] refl le4]

note prem = add-le-mono[OF add-le-mono[OF hyps2] order-refl[where x=3]]

from heapify-correct[OF le1 h1] heapify-correct[OF le4 h2]
have braun: braun ⟨l, x, r⟩
  by auto

have t-sift-l:
  t-sift-down l x r ≤ height l + 1
  using t-sift-down-height[OF braun] braun-height-r-le[OF braun]
  by simp

from t-sift-down-height[OF braun]
have height-sift-r:
  height (sift-down l x r) ≤ height r + 2
  using sift-down-height[OF braun] braun-height-l-le[OF braun]
  by simp

from h1 h2 t-sift-l height-sift-r 2.prem
show ?case
  apply simp
  apply (rule order-trans, rule order-trans[rotated], rule prem)
  apply simp-all
  apply (simp only: mult-le-cancel1 add-mult-distrib2[symmetric])
  apply simp
  done

qed simp-all

lemma t-heapify-bound:
  i ≤ length xs ⇒ t-heapify i xs ≤ 5 * i + 1
  using t-heapify-induct[of i xs]
  by simp

```

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)
by (rule is-measure-trivial)

```

```

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 <<>, 0::int, <>> <<>, 0, <>> = <<>, 0, <<>, 0, <>>>

```

```

lemma merge-size[termination-simp]:
size (merge l r) = size l + size r
by (induct rule: merge.induct; simp)

```

```

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-tree (merge l r) = set-tree l  $\cup$  set-tree r
by (induct l r rule: merge.induct; simp)

```

```

lemma mset-tree-merge[simp]:
mset-tree (merge l r) = mset-tree l + mset-tree r
by (induct l r rule: merge.induct; simp)

```

```

lemma merge-heap:
heap l  $\Longrightarrow$  heap r  $\Longrightarrow$  heap (merge l r)
by (induct l r rule: merge.induct; auto simp: ball-Un)

```

```

lemma set-list-of-A[simp]:
set (list-of-A t) = set-tree t
by (induct t rule: list-of-A.induct; simp)

```

```

lemma mset-list-of-A[simp]:
mset (list-of-A t) = mset-tree t
by (induct t rule: list-of-A.induct; simp)

```

```

lemma sorted-list-of-A:
heap t  $\Longrightarrow$  sorted (list-of-A t)
by (induct t rule: list-of-A.induct; simp add: merge-heap)

```

```

lemma sortedA: sorted (list-of-A (heap-of-A xs))
by (simp add: heap-heap-of-A sorted-list-of-A)

```

```

lemma msetA: mset (list-of-A (heap-of-A xs)) = mset xs
by (simp add: mset-tree-heap-of-A)

```

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*:

$height (merge\ l\ r) \leq Suc\ (max\ (height\ l)\ (height\ r))$
by (*induct rule: merge.induct, auto*)

corollary *merge-height-display*:

$height (merge\ l\ r) \leq height\ (Node\ l\ x\ r)$
using *merge-height* **by** *simp*

fun *t-merge* :: ('a::linorder) tree \Rightarrow 'a tree \Rightarrow nat **where**

t-merge Leaf *t2* = 0 |
t-merge *t1* Leaf = 0 |
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 |
t-list-of-A (Node *l* *a* *r*) = 1 + *t-merge* *l* *r* + *t-list-of-A* (merge *l* *r*)

lemma *t-merge-height*:

$t\text{-merge}\ l\ r \leq max\ (height\ l)\ (height\ r)$
by (*induct rule: t-merge.induct, auto*)

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

$height\ t \leq n \implies t\text{-list-of-A}\ t \leq 2 * n * size\ t$
apply (*induct rule: t-list-of-A.induct*)
apply *simp*
apply *simp*
apply (*drule meta-mp*)
apply (*rule order-trans, rule merge-height*)
apply *simp*
apply (*simp add: merge-size*)
apply (*cut-tac l=l and r=r in t-merge-height*)
apply *linarith*
done

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

$t\text{-list-of-A}\ t \leq 2 * height\ t * size\ t$
by (*rule t-list-of-A-induct, simp*)

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

$braun\ t \implies t\text{-list-of-A}\ t \leq 2 * nlog2\ (size\ t + 1) * size\ t$
using *t-list-of-A-bound*[of *t*]
by (*simp add: height-acomplete acomplete-if-braun size1-size*)

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 -

have ?lhs ≤ ?lhs by simp

also note *t-heap-of-A-log-bound*[of xs]

also note *t-list-of-A-log-bound*[of heap-of-A xs, OF braun-heap-of-A]

finally show ?thesis

by (simp add: size-heap-of-A)

qed

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))*

by *pat-completeness auto*

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

braun t ⇒ list-of-B-dom t

apply (*induct t rule: measure-induct*[**where** *f=size*])

apply (*rule accpI, erule list-of-B-rel.cases*)

apply (*clarsimp simp: size-del-min braun-del-min*)

done

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

apply (*induct t rule: measure-induct*[**where** *f=size*])

apply (*case-tac x; simp add: list-of-B-braun-simps*)

apply (*simp add: size-del-min braun-del-min mset-del-min*)

done

lemma *set-list-of-B*:

braun t ⇒ set (list-of-B t) = set-tree t

by (*simp only: set-mset-mset*[*symmetric*] *mset-list-of-B, simp*)

lemma *sorted-list-of-B*:

braun t ⇒ heap t ⇒ sorted (list-of-B t)

apply (*induct t rule: measure-induct*[**where** *f=size*])

apply (*case-tac x; simp add: list-of-B-braun-simps*)

apply (*clarsimp simp: set-list-of-B braun-del-min size-del-min heap-del-min*)

apply (*simp add: set-mset-tree*[*symmetric*] *mset-del-min del: set-mset-tree*)

done

definition

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

```

lemma sortedB: sorted (list-of-B (heap-of-B xs))
by (simp add: heap-of-B-def braun-heapify heap-heapify sorted-list-of-B)

lemma msetB: mset (list-of-B (heap-of-B xs)) = mset xs
by (simp add: heap-of-B-def braun-heapify mset-heapify mset-list-of-B)

fun t-del-left :: 'a tree  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  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))
by pat-completeness auto

lemma t-del-left-bound:
t  $\neq$  Leaf  $\implies$  t-del-left t  $\leq$  2 * height t
apply (induct rule: t-del-left.induct; clarsimp)
apply (atomize(full); clarsimp simp: prod-eq-iff)
apply (simp add: nat-mult-max-right le-max-iff-disj)
done

lemma del-left-height:
del-left t = (v, t')  $\implies$  t  $\neq$  Leaf  $\implies$  height t'  $\leq$  height t
apply (induct t arbitrary: v t' rule: del-left.induct; simp)
apply (atomize(full), clarsimp split: prod.splits)
apply simp
done

lemma t-del-min-bound:
braun t  $\implies$  t-del-min t  $\leq$  3 * height t
apply (cases t rule: t-del-min.cases; simp)
apply (clarsimp split: prod.split)
apply (frule del-left-braun, simp+)
apply (frule del-left-size, simp+)
apply (frule del-left-height, simp)
apply (rule order-trans)
apply ((rule add-le-mono t-del-left-bound t-sift-down-height | simp)+)[1]
apply auto[1]
apply (simp add: max-def)
done

```

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

braun t \implies *t-list-of-B-dom t*

apply (*induct t rule: measure-induct*[**where** *f=size*])

apply (*rule accpI, erule t-list-of-B-rel.cases*)

apply (*clarsimp simp: size-del-min braun-del-min*)

done

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

= *t-list-of-B.psimps*[*OF t-list-of-B-braun-termination*]

lemma *del-min-height*:

braun t \implies *height (del-min t) \leq height t*

apply (*cases t rule: del-min.cases; simp*)

apply (*clarsimp split: prod.split*)

apply (*frule del-left-braun, simp+*)

apply (*frule del-left-size, simp+*)

apply (*drule del-left-height*)

apply *simp*

apply (*rule order-trans, rule sift-down-height, auto*)

done

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

braun t \implies *height t \leq n \implies t-list-of-B t \leq 3 * (n + 1) * size t*

apply (*induct t rule: measure-induct*[**where** *f=size*])

apply (*drule-tac x=del-min x in spec*)

apply (*frule del-min-height*)

apply (*case-tac x; simp add: t-list-of-B-braun-simps*)

apply (*rename-tac l x' r*)

apply (*clarsimp simp: braun-del-min size-del-min*)

apply (*rule order-trans*)

apply (*((rule add-le-mono t-del-min-bound | assumption | simp)+)*[1])

apply *simp*

done

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

braun t \implies *t-list-of-B t \leq 3 * (height t + 1) * size t*

by (*erule t-list-of-B-induct, simp*)

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

braun t \implies *t-list-of-B t \leq 3 * (nlog2 (size t + 1) + 1) * size t*

apply (*frule t-list-of-B-bound*)

apply (*simp add: height-acomplete acomplete-if-braun size1-size*)

done

definition

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

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

*t-heap-of-B xs \leq 6 * length xs + 1*

```

    by (simp add: t-heap-of-B-def order-trans[OF t-heapify-bound])

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 -
  have ?lhs ≤ ?lhs by simp
  also note t-heap-of-B-bound[of xs]
  also note t-list-of-B-log-bound[of heap-of-B xs]
  finally show ?thesis
    apply (simp add: size-heapify braun-heapify heap-of-B-def)
    apply (simp add: field-simps)
  done
qed

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.