

Pairing Heap

Hauke Brinkop and Tobias Nipkow

December 14, 2021

Abstract

This library defines three different versions of pairing heaps: a functional version of the original design based on binary trees [1], the version by Okasaki [2] and a modified version of the latter that is free of structural invariants.

The amortized complexities of these implementations are analyzed in the AFP article [Amortized Complexity](#).

Contents

1	Pairing Heap in Binary Tree Representation	1
1.1	Definitions	2
1.2	Correctness Proofs	2
1.2.1	Invariants	2
1.2.2	Functional Correctness	3
2	Pairing Heap According to Okasaki	4
2.1	Definitions	4
2.2	Correctness Proofs	5
2.2.1	Invariants	5
2.2.2	Functional Correctness	6
3	Pairing Heap According to Okasaki (Modified)	6
3.1	Definitions	7
3.2	Correctness Proofs	8
3.2.1	Invariants	8
3.2.2	Functional Correctness	8

1 Pairing Heap in Binary Tree Representation

```
theory Pairing-Heap-Tree
imports
  HOL-Library.Tree-Multiset
  HOL-Data-Structures.Priority-Queue-Specs
begin
```

1.1 Definitions

Pairing heaps [1] in their original representation as binary trees.

```
fun get-min :: 'a :: linorder tree  $\Rightarrow$  'a where  
get-min (Node - x -) = x
```

```
fun link :: ('a::linorder) tree  $\Rightarrow$  'a tree where  
link (Node hsx x (Node hsy y hs)) =  
  (if x < y then Node (Node hsy y hsx) x hs else Node (Node hsx x hsy) y hs) |  
link t = t
```

```
fun pass1 :: ('a::linorder) tree  $\Rightarrow$  'a tree where  
pass1 (Node hsx x (Node hsy y hs)) = link (Node hsx x (Node hsy y (pass1 hs))) |  
pass1 hs = hs
```

```
fun pass2 :: ('a::linorder) tree  $\Rightarrow$  'a tree where  
pass2 (Node hsx x hs) = link(Node hsx x (pass2 hs)) |  
pass2 Leaf = Leaf
```

```
fun del-min :: ('a::linorder) tree  $\Rightarrow$  'a tree where  
del-min Leaf = Leaf  
| del-min (Node hs - Leaf) = pass2 (pass1 hs)
```

```
fun merge :: ('a::linorder) tree  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree where  
merge Leaf hp = hp  
| merge hp Leaf = hp  
| merge (Node hsx x Leaf) (Node hsy y Leaf) = link (Node hsx x (Node hsy y Leaf))
```

```
fun insert :: ('a::linorder)  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree where  
insert x hp = merge (Node Leaf x Leaf) hp
```

The invariant is the conjunction of *is-root* and *pheap*:

```
fun is-root :: 'a tree  $\Rightarrow$  bool where  
is-root hp = (case hp of Leaf  $\Rightarrow$  True | Node l x r  $\Rightarrow$  r = Leaf)
```

```
fun pheap :: ('a :: linorder) tree  $\Rightarrow$  bool where  
pheap Leaf = True |  
pheap (Node l x r) = (( $\forall y \in$  set-tree l. x  $\leq$  y)  $\wedge$  pheap l  $\wedge$  pheap r)
```

1.2 Correctness Proofs

1.2.1 Invariants

```
lemma link-struct:  $\exists l a.$  link (Node hsx x (Node hsy y hs)) = Node l a hs  
<proof>
```

```
lemma pass1-struct:  $\exists l a r.$  pass1 (Node hs1 x hs) = Node l a r  
<proof>
```

```
lemma pass2-struct:  $\exists l a.$  pass2 (Node hs1 x hs) = Node l a Leaf
```

<proof>

lemma *is-root-merge*:

$is\text{-root } h1 \implies is\text{-root } h2 \implies is\text{-root } (merge\ h1\ h2)$

<proof>

lemma *is-root-insert*: $is\text{-root } h \implies is\text{-root } (insert\ x\ h)$

<proof>

lemma *is-root-del-min*:

assumes $is\text{-root } h$ **shows** $is\text{-root } (del\text{-min } h)$

<proof>

lemma *pheap-merge*:

$\llbracket is\text{-root } h1; is\text{-root } h2; pheap\ h1; pheap\ h2 \rrbracket \implies pheap\ (merge\ h1\ h2)$

<proof>

lemma *pheap-insert*: $is\text{-root } h \implies pheap\ h \implies pheap\ (insert\ x\ h)$

<proof>

lemma *pheap-link*: $t \neq Leaf \implies pheap\ t \implies pheap\ (link\ t)$

<proof>

lemma *pheap-pass1*: $pheap\ h \implies pheap\ (pass_1\ h)$

<proof>

lemma *pheap-pass2*: $pheap\ h \implies pheap\ (pass_2\ h)$

<proof>

lemma *pheap-del-min*: $is\text{-root } h \implies pheap\ h \implies pheap\ (del\text{-min } h)$

<proof>

1.2.2 Functional Correctness

lemma *get-min-in*:

$h \neq Leaf \implies get\text{-min } h \in set\text{-tree } h$

<proof>

lemma *get-min-min*: $\llbracket is\text{-root } h; pheap\ h; x \in set\text{-tree } h \rrbracket \implies get\text{-min } h \leq x$

<proof>

lemma *mset-link*: $mset\text{-tree } (link\ t) = mset\text{-tree } t$

<proof>

lemma *mset-pass1*: $mset\text{-tree } (pass_1\ h) = mset\text{-tree } h$

<proof>

lemma *mset-pass2*: $mset\text{-tree } (pass_2\ h) = mset\text{-tree } h$

<proof>

lemma *mset-merge*: $\llbracket \text{is-root } h1; \text{is-root } h2 \rrbracket$
 $\implies \text{mset-tree } (\text{merge } h1 \ h2) = \text{mset-tree } h1 + \text{mset-tree } h2$
<proof>

lemma *mset-del-min*: $\llbracket \text{is-root } h; t \neq \text{Leaf} \rrbracket \implies$
 $\text{mset-tree } (\text{del-min } h) = \text{mset-tree } h - \{\#\text{get-min } h\#\}$
<proof>

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

interpretation *pairing*: *Priority-Queue-Merge*
where *empty* = *Leaf* **and** *is-empty* = $\lambda h. h = \text{Leaf}$
and *merge* = *merge* **and** *insert* = *insert*
and *del-min* = *del-min* **and** *get-min* = *get-min*
and *invar* = $\lambda h. \text{is-root } h \wedge \text{pheap } h$ **and** *mset* = *mset-tree*
<proof>

end

2 Pairing Heap According to Okasaki

theory *Pairing-Heap-List1*
imports
 HOL-Library.Multiset
 HOL-Library.Pattern-Aliases
 HOL-Data-Structures.Priority-Queue-Specs
begin

2.1 Definitions

This implementation follows Okasaki [2]. It satisfies the invariant that *Empty* only occurs at the root of a pairing heap. The functional correctness proof does not require the invariant but the amortized analysis (elsewhere) makes use of it.

datatype *'a heap* = *Empty* | *Hp 'a 'a heap list*

fun *get-min* :: *'a heap* \Rightarrow *'a* **where**
get-min (*Hp* *x* *-*) = *x*

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

context **includes** *pattern-aliases*
begin

fun *merge* :: (*'a::linorder*) *heap* \Rightarrow *'a heap* \Rightarrow *'a heap* **where**
merge *h* *Empty* = *h* |
merge *Empty* *h* = *h* |

```
merge (Hp x hsx =: hx) (Hp y hsy =: hy) =
  (if x < y then Hp x (hy # hsx) else Hp y (hx # hsy))
```

end

```
fun insert :: ('a::linorder) => 'a heap => 'a heap where
insert x h = merge (Hp x []) h
```

```
fun pass1 :: ('a::linorder) heap list => 'a heap list where
pass1 (h1#h2#hs) = merge h1 h2 # pass1 hs |
pass1 hs = hs
```

```
fun pass2 :: ('a::linorder) heap list => 'a heap where
pass2 [] = Empty
| pass2 (h#hs) = merge h (pass2 hs)
```

```
fun merge-pairs :: ('a::linorder) heap list => 'a heap where
merge-pairs [] = Empty
| merge-pairs [h] = h
| merge-pairs (h1 # h2 # hs) = merge (merge h1 h2) (merge-pairs hs)
```

```
fun del-min :: ('a::linorder) heap => 'a heap where
del-min Empty = Empty
| del-min (Hp x hs) = pass2 (pass1 hs)
```

2.2 Correctness Proofs

An optimization:

```
lemma pass12-merge-pairs: pass2 (pass1 hs) = merge-pairs hs
⟨proof⟩
```

```
declare pass12-merge-pairs[code-unfold]
```

2.2.1 Invariants

```
fun mset-heap :: 'a heap => 'a multiset where
mset-heap Empty = {#} |
mset-heap (Hp x hs) = {#x#} + sum-mset(mset(map mset-heap hs))
```

```
fun pheap :: ('a :: linorder) heap => bool where
pheap Empty = True |
pheap (Hp x hs) = (∀ h ∈ set hs. (∀ y ∈# mset-heap h. x ≤ y) ∧ pheap h)
```

```
lemma pheap-merge: pheap h1 ==> pheap h2 ==> pheap (merge h1 h2)
⟨proof⟩
```

```
lemma pheap-merge-pairs: ∀ h ∈ set hs. pheap h ==> pheap (merge-pairs hs)
⟨proof⟩
```

lemma *pheap-insert*: $pheap\ h \implies pheap\ (insert\ x\ h)$
(*proof*)

lemma *pheap-del-min*: $pheap\ h \implies pheap\ (del-min\ h)$
(*proof*)

2.2.2 Functional Correctness

lemma *mset-heap-empty-iff*: $mset-heap\ h = \{\#\} \longleftrightarrow h = Empty$
(*proof*)

lemma *get-min-in*: $h \neq Empty \implies get-min\ h \in\# mset-heap(h)$
(*proof*)

lemma *get-min-min*: $\llbracket h \neq Empty; pheap\ h; x \in\# mset-heap(h) \rrbracket \implies get-min\ h \leq x$
(*proof*)

lemma *get-min*: $\llbracket pheap\ h; h \neq Empty \rrbracket \implies get-min\ h = Min-mset\ (mset-heap\ h)$
(*proof*)

lemma *mset-merge*: $mset-heap\ (merge\ h1\ h2) = mset-heap\ h1 + mset-heap\ h2$
(*proof*)

lemma *mset-insert*: $mset-heap\ (insert\ a\ h) = \{\#a\#\} + mset-heap\ h$
(*proof*)

lemma *mset-merge-pairs*: $mset-heap\ (merge-pairs\ hs) = sum-mset(image-mset\ mset-heap(mset\ hs))$
(*proof*)

lemma *mset-del-min*: $h \neq Empty \implies mset-heap\ (del-min\ h) = mset-heap\ h - \{\#get-min\ h\#\}$
(*proof*)

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

interpretation *pairing*: *Priority-Queue-Merge*
where *empty* = *Empty* **and** *is-empty* = $\lambda h. h = Empty$
and *merge* = *merge* **and** *insert* = *insert*
and *del-min* = *del-min* **and** *get-min* = *get-min*
and *invar* = *pheap* **and** *mset* = *mset-heap*
(*proof*)

end

3 Pairing Heap According to Oksaki (Modified)

theory *Pairing-Heap-List2*

```

imports
  HOL-Library.Multiset
  HOL-Data-Structures.Priority-Queue-Specs
begin

```

3.1 Definitions

This version of pairing heaps is a modified version of the one by Okasaki [2] that avoids structural invariants.

```

datatype 'a hp = Hp 'a (hps: 'a hp list)

```

```

type-synonym 'a heap = 'a hp option

```

```

hide-const (open) insert

```

```

fun get-min :: 'a heap ⇒ 'a where
  get-min (Some(Hp x -)) = x

```

```

fun link :: ('a::linorder) hp ⇒ 'a hp ⇒ 'a hp where
  link (Hp x lx) (Hp y ly) =
    (if x < y then Hp x (Hp y ly # lx) else Hp y (Hp x lx # ly))

```

```

fun merge :: ('a::linorder) heap ⇒ 'a heap ⇒ 'a heap where
  merge h None = h |
  merge None h = h |
  merge (Some h1) (Some h2) = Some(link h1 h2)

```

```

lemma merge-None[simp]: merge None h = h
<proof>

```

```

fun insert :: ('a::linorder) ⇒ 'a heap ⇒ 'a heap where
  insert x None = Some(Hp x []) |
  insert x (Some h) = Some(link (Hp x []) h)

```

```

fun pass1 :: ('a::linorder) hp list ⇒ 'a hp list where
  pass1 [] = []
| pass1 [h] = [h]
| pass1 (h1#h2#hs) = link h1 h2 # pass1 hs

```

```

fun pass2 :: ('a::linorder) hp list ⇒ 'a heap where
  pass2 [] = None
| pass2 (h#hs) = Some(case pass2 hs of None ⇒ h | Some h' ⇒ link h h')

```

```

fun merge-pairs :: ('a::linorder) hp list ⇒ 'a heap where
  merge-pairs [] = None
| merge-pairs [h] = Some h
| merge-pairs (h1 # h2 # hs) =
  Some(let h12 = link h1 h2 in case merge-pairs hs of None ⇒ h12 | Some h ⇒

```

link h12 h)

fun *del-min* :: ('a::linorder) heap \Rightarrow 'a heap **where**
 del-min None = None
 | *del-min* (Some(Hp x hs)) = *pass*₂ (*pass*₁ hs)

3.2 Correctness Proofs

An optimization:

lemma *pass12-merge-pairs*: *pass*₂ (*pass*₁ hs) = *merge-pairs* hs
<proof>

declare *pass12-merge-pairs*[*code-unfold*]

3.2.1 Invariants

fun *php* :: ('a::linorder) hp \Rightarrow bool **where**
php (Hp x hs) = ($\forall h \in \text{set } hs. (\forall y \in \text{set-hp } h. x \leq y) \wedge \text{php } h$)

definition *invar* :: ('a::linorder) heap \Rightarrow bool **where**
invar ho = (case ho of None \Rightarrow True | Some h \Rightarrow *php* h)

lemma *php-link*: *php* h1 \Longrightarrow *php* h2 \Longrightarrow *php* (*link* h1 h2)
<proof>

lemma *invar-merge*:
 [[*invar* h1; *invar* h2]] \Longrightarrow *invar* (*merge* h1 h2)
<proof>

lemma *invar-insert*: *invar* h \Longrightarrow *invar* (*insert* x h)
<proof>

lemma *invar-pass1*: $\forall h \in \text{set } hs. \text{php } h \Longrightarrow \forall h \in \text{set } (\text{pass}_1 \text{ } hs). \text{php } h$
<proof>

lemma *invar-pass2*: $\forall h \in \text{set } hs. \text{php } h \Longrightarrow \text{invar } (\text{pass}_2 \text{ } hs)$
<proof>

lemma *invar-Some*: *invar*(Some h) = *php* h
<proof>

lemma *invar-del-min*: *invar* h \Longrightarrow *invar* (*del-min* h)
<proof>

3.2.2 Functional Correctness

fun *mset-hp* :: 'a hp \Rightarrow 'a multiset **where**
mset-hp (Hp x hs) = {#x#} + *sum-mset*(*mset*(*map* *mset-hp* hs))

definition *mset-heap* :: 'a heap \Rightarrow 'a multiset **where**
mset-heap ho = (case ho of None \Rightarrow {#} | Some h \Rightarrow *mset-hp* h)

lemma *set-mset-mset-hp*: *set-mset* (*mset-hp* h) = *set-hp* h
 <proof>

lemma *mset-hp-empty[simp]*: *mset-hp* hp \neq {#}
 <proof>

lemma *mset-heap-Some*: *mset-heap*(Some hp) = *mset-hp* hp
 <proof>

lemma *mset-heap-empty*: *mset-heap* h = {#} \longleftrightarrow h = None
 <proof>

lemma *get-min-in*:
 h \neq None \implies *get-min* h \in *set-hp*(the h)
 <proof>

lemma *get-min-min*: \llbracket h \neq None; *invar* h; x \in *set-hp*(the h) $\rrbracket \implies$ *get-min* h \leq x
 <proof>

lemma *mset-link*: *mset-hp* (*link* h1 h2) = *mset-hp* h1 + *mset-hp* h2
 <proof>

lemma *mset-merge*: *mset-heap* (*merge* h1 h2) = *mset-heap* h1 + *mset-heap* h2
 <proof>

lemma *mset-insert*: *mset-heap* (*insert* a h) = {#a#} + *mset-heap* h
 <proof>

lemma *mset-merge-pairs*: *mset-heap* (*merge-pairs* hs) = *sum-mset*(*image-mset* *mset-hp*
 (*mset* hs))
 <proof>

lemma *mset-del-min*: h \neq None \implies
mset-heap (*del-min* h) = *mset-heap* h - {#*get-min* h#}
 <proof>

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

interpretation *pairing*: *Priority-Queue-Merge*
where *empty* = None **and** *is-empty* = $\lambda h. h = \text{None}$
and *merge* = *merge* **and** *insert* = *insert*
and *del-min* = *del-min* **and** *get-min* = *get-min*
and *invar* = *invar* **and** *mset* = *mset-heap*
 <proof>

end

References

- [1] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
- [2] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.