

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	5
2.1	Definitions	5
2.2	Correctness Proofs	6
2.2.1	Invariants	6
2.2.2	Functional Correctness	6
3	Pairing Heap According to Okasaki (Modified)	8
3.1	Definitions	8
3.2	Correctness Proofs	9
3.2.1	Invariants	9
3.2.2	Functional Correctness	10

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  
by simp
```

```
lemma pass1-struct:  $\exists l a r.$  pass1 (Node hs1 x hs) = Node l a r  
by (cases hs) simp-all
```

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

by(*induction hs arbitrary: hs1 x rule: pass2.induct*) (*auto, metis link-struct*)

lemma *is-root-merge*:

is-root h1 \implies *is-root h2* \implies *is-root (merge h1 h2)*

by (*simp split: tree.splits*)

lemma *is-root-insert*: *is-root h* \implies *is-root (insert x h)*

by (*simp split: tree.splits*)

lemma *is-root-del-min*:

assumes *is-root h* **shows** *is-root (del-min h)*

proof (*cases h*)

case [*simp*]: (*Node lx x rx*)

have *rx = Leaf* **using** *assms* **by** *simp*

thus *?thesis*

proof (*cases lx*)

case (*Node ly y ry*)

then obtain *la a ra* **where** *pass1 lx = Node a la ra*

using *pass1-struct* **by** *blast*

moreover obtain *lb b* **where** *pass2 ... = Node b lb Leaf*

using *pass2-struct* **by** *blast*

ultimately show *?thesis* **using** *assms* **by** *simp*

qed *simp*

qed *simp*

lemma *pheap-merge*:

\llbracket *is-root h1*; *is-root h2*; *pheap h1*; *pheap h2* $\rrbracket \implies$ *pheap (merge h1 h2)*

by (*auto split: tree.splits*)

lemma *pheap-insert*: *is-root h* \implies *pheap h* \implies *pheap (insert x h)*

by (*auto split: tree.splits*)

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

by(*induction t rule: link.induct*)(*auto*)

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

by(*induction h rule: pass1.induct*) (*auto*)

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

by (*induction h*)(*auto simp: pheap-link*)

lemma *pheap-del-min*: *is-root h* \implies *pheap h* \implies *pheap (del-min h)*

by (*auto simp: pheap-pass1 pheap-pass2 split: tree.splits*)

1.2.2 Functional Correctness

lemma *get-min-in*:

h \neq *Leaf* \implies *get-min h* \in *set-tree h*

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

lemma *get-min-min*: $\llbracket \text{is-root } h; \text{pheap } h; x \in \text{set-tree } h \rrbracket \implies \text{get-min } h \leq x$
by(*auto split: tree.splits*)

lemma *mset-link*: $\text{mset-tree } (\text{link } t) = \text{mset-tree } t$
by(*cases t rule: link.cases*)(*auto simp: add-ac*)

lemma *mset-pass₁*: $\text{mset-tree } (\text{pass}_1 h) = \text{mset-tree } h$
by (*induction h rule: pass₁.induct*) *auto*

lemma *mset-pass₂*: $\text{mset-tree } (\text{pass}_2 h) = \text{mset-tree } h$
by (*induction h rule: pass₂.induct*) (*auto simp: mset-link*)

lemma *mset-merge*: $\llbracket \text{is-root } h1; \text{is-root } h2 \rrbracket$
 $\implies \text{mset-tree } (\text{merge } h1 h2) = \text{mset-tree } h1 + \text{mset-tree } h2$
by (*induction h1 h2 rule: merge.induct*) (*auto simp add: ac-simps*)

lemma *mset-del-min*: $\llbracket \text{is-root } h; t \neq \text{Leaf} \rrbracket \implies$
 $\text{mset-tree } (\text{del-min } h) = \text{mset-tree } h - \{\#\text{get-min } h\#$
by(*induction h rule: del-min.induct*)(*auto simp: mset-pass₁ mset-pass₂*)

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(*standard, goal-cases*)
 case 1 show ?case by simp
next
 case (2 q) show ?case by (cases q) auto
next
 case 3 thus ?case by (simp add: mset-merge)
next
 case 4 thus ?case by (simp add: mset-del-min)
next
 case 5 thus ?case by (simp add: eq-Min-iff get-min-in get-min-min)
next
 case 6 thus ?case by (simp)
next
 case 7 thus ?case using is-root-insert pheap-insert by blast
next
 case 8 thus ?case using is-root-del-min pheap-del-min by blast
next
 case 9 thus ?case by (simp add: mset-merge)
next
 case 10 thus ?case using is-root-merge pheap-merge by blast
qed

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 ⇒ 'a where
get-min (Hp x _) = x
```

```
hide-const (open) insert
```

```
context includes pattern-aliases
begin
```

```
fun merge :: ('a::linorder) heap ⇒ 'a heap ⇒ '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  $\Rightarrow$  '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
by (induction hs rule: merge-pairs.induct) (auto split: option.split)

```

```

declare pass12-merge-pairs[code-unfold]

```

2.2.1 Invariants

```

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

```

```

fun pheap :: ('a :: linorder) heap  $\Rightarrow$  bool where
pheap Empty = True |
pheap (Hp x hs) = ( $\forall h \in \text{set } hs. (\forall y \in \# \text{ mset-heap } h. x \leq y) \wedge \text{pheap } h$ )

```

```

lemma pheap-merge: pheap h1  $\implies$  pheap h2  $\implies$  pheap (merge h1 h2)
by (induction h1 h2 rule: merge.induct) fastforce+

```

```

lemma pheap-merge-pairs:  $\forall h \in \text{set } hs. \text{pheap } h \implies \text{pheap } (\text{merge-pairs } hs)$ 
by (induction hs rule: merge-pairs.induct)(auto simp: pheap-merge)

```

```

lemma pheap-insert: pheap h  $\implies$  pheap (insert x h)
by (auto simp: pheap-merge)

```

```

lemma pheap-del-min: pheap h  $\implies$  pheap (del-min h)
by(cases h) (auto simp: pass12-merge-pairs pheap-merge-pairs)

```

2.2.2 Functional Correctness

```

lemma mset-heap-empty-iff: mset-heap h = {#}  $\longleftrightarrow$  h = Empty
by (cases h) auto

```

```

lemma get-min-in: h  $\neq$  Empty  $\implies$  get-min h  $\in \#$  mset-heap(h)
by(induction rule: get-min.induct)(auto)

```

```

lemma get-min-min: [ h  $\neq$  Empty; pheap h; x  $\in \#$  mset-heap(h) ]  $\implies$  get-min h
 $\leq$  x
by(induction h rule: get-min.induct)(auto)

```

lemma *get-min*: $\llbracket \text{pheap } h; h \neq \text{Empty} \rrbracket \implies \text{get-min } h = \text{Min-mset } (\text{mset-heap } h)$
by (*metis Min-eqI finite-set-mset get-min-in get-min-min*)

lemma *mset-merge*: $\text{mset-heap } (\text{merge } h1 \ h2) = \text{mset-heap } h1 + \text{mset-heap } h2$
by(*induction h1 h2 rule: merge.induct*)(*auto simp: add-ac*)

lemma *mset-insert*: $\text{mset-heap } (\text{insert } a \ h) = \{\#a\# \} + \text{mset-heap } h$
by(*cases h*) (*auto simp add: mset-merge insert-def add-ac*)

lemma *mset-merge-pairs*: $\text{mset-heap } (\text{merge-pairs } hs) = \text{sum-mset}(\text{image-mset } \text{mset-heap}(\text{mset } hs))$
by(*induction hs rule: merge-pairs.induct*)(*auto simp: mset-merge*)

lemma *mset-del-min*: $h \neq \text{Empty} \implies$
 $\text{mset-heap } (\text{del-min } h) = \text{mset-heap } h - \{\#\text{get-min } h\#\}$
by(*cases h*) (*auto simp: pass12-merge-pairs mset-merge-pairs*)

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

interpretation *pairing: Priority-Queue-Merge*
where *empty* = *Empty* **and** *is-empty* = $\lambda h. h = \text{Empty}$
and *merge* = *merge* **and** *insert* = *insert*
and *del-min* = *del-min* **and** *get-min* = *get-min*
and *invar* = *pheap* **and** *mset* = *mset-heap*
proof(*standard, goal-cases*)
 case 1 **show** ?*case* **by** *simp*
next
 case (2 *q*) **show** ?*case* **by** (*cases q*) *auto*
next
 case 3 **show** ?*case* **by**(*simp add: mset-insert mset-merge*)
next
 case 4 **thus** ?*case* **by**(*simp add: mset-del-min mset-heap-empty-iff*)
next
 case 5 **thus** ?*case* **using** *get-min mset-heap.simps(1)* **by** *blast*
next
 case 6 **thus** ?*case* **by**(*simp*)
next
 case 7 **thus** ?*case* **by**(*rule pheap-insert*)
next
 case 8 **thus** ?*case* **by** (*simp add: pheap-del-min*)
next
 case 9 **thus** ?*case* **by** (*simp add: mset-merge*)
next
 case 10 **thus** ?*case* **by** (*simp add: pheap-merge*)
qed
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  $\Rightarrow$  'a where
get-min (Some(Hp x -)) = x
```

```
fun link :: ('a::linorder) hp  $\Rightarrow$  'a hp  $\Rightarrow$  '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  $\Rightarrow$  'a heap  $\Rightarrow$  '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
by(cases h)auto
```

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

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

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

```
fun merge-pairs :: ('a::linorder) hp list  $\Rightarrow$  '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 ⇒ '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
by (*induction* hs rule: *merge-pairs.induct*) (*auto split*: *option.split*)

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

3.2.1 Invariants

fun *php* :: ('a::linorder) hp ⇒ bool **where**
php (Hp x hs) = (∀ h ∈ set hs. (∀ y ∈ set-hp h. x ≤ y) ∧ *php* h)

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

lemma *php-link*: *php* h1 ⇒ *php* h2 ⇒ *php* (*link* h1 h2)
by (*induction* h1 h2 rule: *link.induct*) *fastforce*+

lemma *invar-merge*:
[[*invar* h1; *invar* h2]] ⇒ *invar* (*merge* h1 h2)
by (*auto simp*: *php-link invar-def split*: *option.splits*)

lemma *invar-insert*: *invar* h ⇒ *invar* (*insert* x h)
by (*auto simp*: *php-link invar-def split*: *option.splits*)

lemma *invar-pass1*: ∀ h ∈ set hs. *php* h ⇒ ∀ h ∈ set (*pass*₁ hs). *php* h
by(*induction* hs rule: *pass*₁.*induct*) (*auto simp*: *php-link*)

lemma *invar-pass2*: ∀ h ∈ set hs. *php* h ⇒ *invar* (*pass*₂ hs)
by (*induction* hs)(*auto simp*: *php-link invar-def split*: *option.splits*)

lemma *invar-Some*: *invar*(*Some* h) = *php* h
by(*simp add*: *invar-def*)

lemma *invar-del-min*: *invar* h ⇒ *invar* (*del-min* h)
by(*induction* h rule: *del-min.induct*)
(*auto simp*: *invar-Some intro*!: *invar-pass1 invar-pass2*)

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
by(induction h) auto

lemma *mset-hp-empty[simp]*: mset-hp hp \neq {#}
by (cases hp) auto

lemma *mset-heap-Some*: mset-heap(Some hp) = mset-hp hp
by(simp add: mset-heap-def)

lemma *mset-heap-empty*: mset-heap h = {#} \longleftrightarrow h = None
by (cases h) (auto simp add: mset-heap-def)

lemma *get-min-in*:
 $h \neq \text{None} \implies \text{get-min } h \in \text{set-hp}(\text{the } h)$
by(induction rule: get-min.induct)(auto)

lemma *get-min-min*: $\llbracket h \neq \text{None}; \text{invar } h; x \in \text{set-hp}(\text{the } h) \rrbracket \implies \text{get-min } h \leq x$
by(induction h rule: get-min.induct)(auto simp: invar-def)

lemma *mset-link*: mset-hp (link h1 h2) = mset-hp h1 + mset-hp h2
by(induction h1 h2 rule: link.induct)(auto simp: add-ac)

lemma *mset-merge*: mset-heap (merge h1 h2) = mset-heap h1 + mset-heap h2
by (induction h1 h2 rule: merge.induct)
(auto simp add: mset-heap-def mset-link ac-simps)

lemma *mset-insert*: mset-heap (insert a h) = {#a#} + mset-heap h
by(cases h) (auto simp add: mset-link mset-heap-def insert-def)

lemma *mset-merge-pairs*: mset-heap (merge-pairs hs) = sum-mset(image-mset mset-hp (mset hs))
by(induction hs rule: merge-pairs.induct)
(auto simp: mset-merge mset-link mset-heap-def Let-def split: option.split)

lemma *mset-del-min*: $h \neq \text{None} \implies$
 $\text{mset-heap } (\text{del-min } h) = \text{mset-heap } h - \{\#\text{get-min } h\#\}$
by(induction h rule: del-min.induct)
(auto simp: mset-heap-Some pass12-merge-pairs mset-merge-pairs)

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(standard, goal-cases)
  case 1 show ?case by(simp add: mset-heap-def)
next
  case (2 q) thus ?case by(auto simp add: mset-heap-def split: option.split)
next
  case 3 show ?case by(simp add: mset-insert mset-merge)
next
  case 4 thus ?case by(simp add: mset-del-min mset-heap-empty)
next
  case (5 q) thus ?case using get-min-in[of q]
    by(auto simp add: eq-Min-iff get-min-min mset-heap-empty mset-heap-Some
set-mset-mset-hp)
next
  case 6 thus ?case by (simp add: invar-def)
next
  case 7 thus ?case by(rule invar-insert)
next
  case 8 thus ?case by (simp add: invar-del-min)
next
  case 9 thus ?case by (simp add: mset-merge)
next
  case 10 thus ?case by (simp add: invar-merge)
qed

end

```

References

- [1] M. L. Fredman, R. Sedgwick, 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.