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
   1.1 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 del-min2 ........................................... 6
   2.2 Correctness Proof ......................................... 7

3 Sorting via Priority Queues Based on Braun Trees 8

4 Phase 1: List to Tree 9

5 Phase 2: Heap to List 14

1 Priority Queues Based on Braun Trees
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).

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

proof (induction \( t \))
  case (Node \( t1 \))
  show ?case
  proof (cases \( \text{height } t1 \))
    case 0 thus ?thesis using Node by simp
  next
    case (Suc \( n \))
    hence $2 ^ n \leq \text{size } t1$ 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 ⇒ ‘a where
get-min (Node l a r) = a

lemma get-min: \[ \text{heap } t; \ t \neq \text{Leaf } \] ⇒ 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 ⇒ ‘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]: \( \text{size } (\text{insert } x \ t) = \text{size } t + 1 \)
by (induction \( t \) arbitrary: \( x \)) auto

lemma mset-insert: \( \text{mset-tree } (\text{insert } x \ t) = \{ \#x\# \} + \text{mset-tree } t \)
by (induction \( t \) arbitrary: \( x \)) (auto simp: ac-simps)

lemma set-insert[simp]: \( \text{set-tree } (\text{insert } x \ t) = \{ x \} \cup (\text{set-tree } t) \)
by (simp add: mset-insert flip: set-mset-tree)
lemma braun-insert: braun t ⟷ braun(insert x t)
by(induction t arbitrary: x) auto

lemma heap-insert: heap t ⟷ heap(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 ⇒ 'a ✤ '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'
by (induction t arbitrary: x t' rule: del-left.induct;
    auto split: prod.splits)

lemma del-left-mset:
  del-left t = (x, t') ⟷ t ≠ Leaf
  ⇒ x ∈# mset-tree t ∧ mset-tree t' = mset-tree t − {#x#}
by (simp add: del-left-mset-plus flip: set-mset-tree)

lemma del-left-set:
  del-left t = (x, t') ⟷ t ≠ Leaf ⟷ set-tree t = {x} ∪ set-tree t'
by(simp add: del-left-mset-plus flip: set-mset-tree)

lemma del-left-heap:
  del-left t = (x, t') ⟷ t ≠ Leaf ⟷ heap t ⟷ 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:
  del-left t = (x, t') ⟷ t ≠ Leaf ⟷ size t = size t' + 1
by(induction t arbitrary: x t' rule: del-left.induct;
    auto split: prod.splits)

lemma del-left-braun:
  del-left t = (x, t') ⟷ braun t ⟷ t ≠ Leaf ⟷ 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 ⟨⟩ 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))
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) ⇒
  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 \rightarrow braun t \rightarrow 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 1 with assms show ?thesis by simp
next
case 2 with assms show ?thesis by (simp)
next
case 3 with assms show ?thesis by (simp add: mset-insert)
next
case 4 thus ?case by (simp add: mset-del-min)

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 \land 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)
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-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 (merge 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 \textit{replace-min} is just \textit{sift-down} in disguise:

\textbf{lemma} replace-min-sift-down: \textit{braun (Node l a r) \implies replace-min x (Node l a r) = sift-down l x r}
\textbf{by}(induction l x r rule: sift-down.induct)(auto)

This means that \textit{del-min2} is merely a slight optimization of \textit{del-min}: instead of calling \textit{del-left} right away, \textit{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 \textit{merge}:

\textbf{lemma} mset-tree-merge:
\textit{braun (Node l x r) \implies mset-tree (merge l r) = mset-tree l + mset-tree r}
\textbf{by}(induction l r rule: merge.induct)
\text{(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)}

\textbf{lemma} heap-merge:
\[ [\text{\textit{braun (Node l x r); heap l; heap r}}] \implies \text{heap (merge l r)} \]
\textbf{proof}(induction l r rule: merge.induct)
\text{case 1 thus ?case by simp}\next
\text{case (2 l1 a1 r1 l2 a2 r2)}
\text{show ?case}
\textbf{proof cases}
\text{assume a1 \leq a2}
\text{thus ?thesis using 2 by(auto simp: ball-Un mset-tree-merge simp flip: set-mset-tree)}
\next
\text{assume \neg a1 \leq a2}
\text{let "l = Node l1 a1 r1 let "r = Node l2 a2 r2}
\text{have "braun "r using 2.prems(1) by auto}
\text{obtain x l' where "dl: del-left "l = (x, l') by (metis surj-pair)}
\text{from del-left-heap[OF this - 2.prems(2)] have heap l' by auto}
\text{have hr: heap (replace-min x "r) using (braun "r) 2.prems(3)}
\text{by(simp add: heap-sift-down neq-Leaf-iff replace-min-sift-down del: replace-min.simps) have 0: \forall x \in set-tree "l. a2 \leq x using 2.prems(2) \neg a1 \leq a2 by (auto simp: ball-Un)}
\text{moreover have set-tree l' \subseteq set-tree "l x \in set-tree "l}
\text{using del-left-mset[OF dl] by (auto simp flip: set-mset-tree dest:in-diffD simp: union-iff)}
\text{ultimately have 1: \forall x \in set-tree l'. a2 \leq x by blast}
\text{have \forall x \in set-tree "r. a2 \leq x using \heap "r) by auto}
\text{thus ?thesis}
\text{using \neg a1 \leq a2 dl (heap (replace-min x "r)) (heap l') (x \in set-tree "l) 0 1 (braun "r)}
\text{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') \Rightarrow braun t \Rightarrow t \neq Leaf \Rightarrow braun t' \land size t = size t' + 1
by (simp add: del-left-braun del-left-size)

lemma braun-size-merge:
  braun (Node l x r) \Rightarrow braun (merge l r) \land 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
  dest!: 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 = Leaf
and insert = insert and del-min = del-min2
and get-min = get-min and invar = \lambda h. braun h \land 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

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

**4 Phase 1: List to Tree**

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

```latex
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)**
**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 $n \log n$, which we can approximate with height.

```latex
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)
```

```latex
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**
**apply (induct t arbitrary: x; simp)**
**apply (simp only: max-Suc-Suc[symmetric] le-max-iff-disj, simp)**
done

**Lemma** height-insert-ge:
\[ \text{height } t \leq \text{height } \left( \text{insert } x \ t \right) \]
**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:
\[ \text{t-heap-of-} A \ \text{xs} \leq \text{length } \text{xs} \ast \left( \text{height } \left( \text{heap-of-} A \ \text{xs} \right) + 1 \right) \]
**proof** (induct \( \text{xs} \))
**case** (\( \text{Cons } x \ \text{xs} \))

\[ \text{let } \tilde{\text{lhs}} = \text{t-insert } x \left( \text{heap-of-} A \ \text{xs} \right) + \text{t-heap-of-} A \ \text{xs} \]
**have** \( \tilde{\text{lhs}} \leq \tilde{\text{lhs}} \)
**by** simp
**also note** Cons
**also note** height-insert-ge[of \( \text{heap-of-} A \ \text{xs} \) \( x \)]
**also note** t-insert-height[of \( x \) \( \text{heap-of-} A \ \text{xs} \)]

**finally show** \( ?\text{case} \)
**apply** simp
**apply** (erule order-trans)
**apply** (simp add: height-insert-ge)
done
**qed** simp-all

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

**Lemma** \( t \)-heap-of-\( A \)-log-bound:
\[ \text{t-heap-of-} A \ \text{xs} \leq \text{length } \text{xs} \ast \left( \text{nlog}_2 \left( \text{length } \text{xs} + 1 \right) + 1 \right) \]
**using** t-heap-of-\( A \)-bound[of \( \text{xs} \)]
**acomplicate-if-braun[OF braun-heap-of-\( A \), of \( \text{xs} \)]
**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 :: \( \text{nat} \Rightarrow (\text{a::linorder}) \ \text{list} \Rightarrow \text{a tree} \ast \text{a list} \)**

**where**

heapify \( 0 \) \( \text{xs} \) = (Leaf, \( \text{xs} \)) | heapify \( \text{Suc } n \) \( x \# \text{xs} \) =

(let \( \text{l} \), \( y s \) = heapify \( \text{Suc } n \ \text{div } 2 \) \( x s \);
\( r \), \( z s \) = heapify \( n \ \text{div } 2 \) \( y s \)
in (sift-down \( l \ x \ r \), \( z s \)))

The result should be a Braun tree:

**Lemma** heapify-snd:
\begin{verbatim}
lemma heapify-snd-tup:
heapify n xs = (t, ys) ⇒ n ≤ length xs ⇒ ys = drop n xs
by (drule heapify-snd, simp)

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 (induct n xs arbitrary: t ys rule: heapify)
case (2 n x xs)

  note len = 2.prems(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 ≤ length xs
    by simp

  note app-hyps = 2.hyps(1)[OF h1 le1]

  from len have le2: n div 2 ≤ length ys1
    by (simp add: ys1)

  note app-hyps = 2.hyps(2)[OF refl h1] simplicity, OF le2 le2]

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

  have eq:
    n div 2 + Suc n div 2 = n
    by simp

  have msets:
    mset (take (Suc n div 2) xs) + mset (take (n div 2) ys1) = mset (take n xs)
    apply (subst append-take-drop-id[symmetric, where n=Suc n div 2 and t=take n xs],
           subst mset-append)
    apply (simp add: take-drop min-absorb1 le1 eq ys1)
    done
\end{verbatim}
from 2.prems app-hyps msets show \textit{case}
\textbf{apply (clarsimp simp: h1 h2 le2)}
\textbf{apply (clarsimp simp: size-sift-down[OF braun]}
\textbf{braun-sift-down[OF braun]}
\textbf{mset-sift-down[OF braun])}
\textbf{apply (simp add: heap-sift-down[OF braun])}
done
qed simp-all

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

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

\textbf{lemma mset-heapify:}
\textit{n \leq \text{length} \, xs \Rightarrow mset-tree (\text{fst} (heapify \, n \, xs)) = mset (\text{take} \, n \, xs)}
\textbf{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

\textbf{function (sequential) t-sift-down :: 'a::linorder \text{tree} \Rightarrow 'a \Rightarrow 'a \text{tree} \Rightarrow nat where}
t-sift-down \text{Leaf} \, a \, \text{Leaf} = 1 |
t-sift-down \text{(Node \, Leaf \, x \, Leaf)} \, a \, \text{Leaf} = 2 |
t-sift-down \text{(Node \, l1 \, x1 \, r1 =: t1)} \, a \, \text{(Node \, l2 \, x2 \, r2 =: t2)} =
\text{(if a \leq x1 \land a \leq x2}
\text{then 1}
\text{else if x1 \leq x2 \, then 1 + t-sift-down \, l1 \, a \, r1}
\text{else 1 + t-sift-down \, l2 \, a \, r2)}
\textbf{by pat-completeness auto}

\textbf{termination}
\textbf{by (relation measure (%(l,a,r). size \, l + size \, r)) auto}

end

\textbf{fun t-heapify :: nat \Rightarrow ('a::linorder) \text{list} \Rightarrow nat where}
t-heapify 0 \, xs = 1 |
t-heapify (Suc \, n) \, (x#xs) =
(\text{let} \, (l, ys) = \text{heapify} \, (Suc \, n \, \text{div} \, 2) \, xs;
\text{t1} = t-heapify \, (Suc \, n \, \text{div} \, 2) \, xs;
\text{(r, zs) = heapify (n \, \text{div} \, 2) \, ys;}}
\[ t_2 = t\text{-heapify}(n \text{ div } 2)\ ys \]
\[in 1 + t_1 + t_2 + t\text{-sift-down } l\ x\ r) \]

**Lemma** \( t\text{-sift-down-height}: \)
\[ \text{brawn}(\text{Node } l\ x\ r) \implies t\text{-sift-down } l\ x\ r \leq \text{height}(\text{Node } l\ x\ r) \]
by \((\text{induct } l\ x\ r \text{ rule: } t\text{-sift-down.induct; } \text{auto})\)

**Lemma** \( \text{sift-down-height}: \)
\[ \text{brawn}(\text{Node } l\ x\ r) \implies \text{height}(\text{sift-down } l\ x\ r) \leq \text{height}(\text{Node } l\ x\ r) \]
by \((\text{induct } l\ x\ r \text{ rule: } \text{sift-down.induct; } \text{auto simp: Let-def})\)

**Lemma** \( \text{brawn-height-r-le}: \)
\[ \text{brawn}(\text{Node } l\ x\ r) \implies \text{height } r \leq \text{height } l \]
by \(\text{rule acomplete-optimal, auto intro: acomplete-if-braun}\)

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

**Lemma** \( \text{brawn-height-node-eq}: \)
assumes \( b: \text{brawn}(\text{Node } l\ x\ r) \)
shows \( \text{height}(\text{Node } l\ x\ r) = \text{Suc}(\text{height } l) \)
using \( b\ \text{braun-height-r-le}[\text{OF } b] \)
by \((\text{auto simp add: max-def})\)

**Lemma** \( t\text{-heapify-induct}: \)
\[ i \leq \text{length } xs \implies t\text{-heapify } i\ xs + \text{height}(\text{fst}(\text{heapify } i\ xs)) \leq 5 \times i + 1 \]
**Proof** \((\text{induct } i\ xs \text{ rule: } t\text{-heapify.induct})\)
\[ \text{case } (1\ vs) \]
thus \(?\text{case}\)
by \(\text{simp}\)
**Next**
\[ \text{case } (2\ i\ xs) \]

\( \text{obtain } l\ ys \text{ where } h1: \text{heapify}(\text{Suc } i\ \text{div } 2)\ xs = (l,\ ys) \)
by \((\text{simp add: prod-eq-iff})\)
\(\text{note hyps1 = 2.hyps[OF h1[\text{symmetric}]}\ refl, \text{simplified}]\)

\( \text{obtain } r\ zs \text{ where } h2: \text{heapify}(i\ \text{div } 2)\ ys = (r,\ zs) \)
by \((\text{simp add: prod-eq-iff})\)

from \(2.\text{prems heapify-snd-tup[OF h1]}\)
\(\text{have le1: Suc } i\ \text{div } 2 \leq \text{length } xs\)
and \(le2: i\ \text{div } 2 \leq \text{length } xs\)
and \(le4: i\ \text{div } 2 \leq \text{length } ys\)
by \(\text{simp-all}\)
\(\text{note hyps2 = hyps1(1)[OF le1] hyps1(2)[OF refl h2[\text{symmetric}]}\ refl le4]\)
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.prems
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 ⇒ '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
by (induct rule: merge.induct; simp)

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
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-height:
height (merge l r) ≤ 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 ⇒ '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 \leq 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 \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-list-of-A t \leq 2 * n * size t
apply (induct rule: t-list-of-A.induct)
  apply simp
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-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-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) \leq 3 * length xs * (nlog2 (length xs + 1) + 1)
(is ?lhs \leq -)
proof –
have \( \text{lhs} \leq \text{lhs} \) 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 \( \text{thesis} \)
  by (simp add; size-heap-of-A)
qed

Running time of algorithm B:

function list-of-B :: ('a::linorder) tree \Rightarrow '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 \Rightarrow list-of-B-dom t
apply (induct t rule: measure-induct[where \( f = \text{size} \)])
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 \Rightarrow mset (list-of-B t) = mset-tree t
apply (induct t rule: measure-induct[where \( f = \text{size} \)])
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 \Rightarrow 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 \Rightarrow heap t \Rightarrow sorted (list-of-B t)
apply (induct t rule: measure-induct[where \( f = \text{size} \)])
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 ⇒ 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))

by pat-completeness auto

lemma t-del-left-bound:
t ≠ Leaf ⇒ t-del-left t ≤ 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') ⇒ t ≠ Leaf ⇒ height t' ≤ 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 ⇒ t-del-min t ≤ 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-ptermination:
braun t ⇒ 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)

18
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 \Longrightarrow 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 (drule del-left-height)
apply simp
apply (rule order-trans, rule sift-down-height, auto)
done

lemma t-list-of-B-induct:
braun t \Longrightarrow height t \leq n \Longrightarrow t-list-of-B t \leq 3 \ast (n + 1) \ast 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 \Longrightarrow t-list-of-B t \leq 3 \ast (height t + 1) \ast size t
by (erule t-list-of-B-induct, simp)

lemma t-list-of-B-log-bound:
braun t \Longrightarrow t-list-of-B t \leq 3 \ast (n\log2 (size t + 1) + 1) \ast size t
apply (frule t-list-of-B-bound)
apply (simp add: height-acomplete acomplete-if-braun size1-size)
done

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

lemma t-heap-of-B-bound:
\[ t\text{-heap-of-B } xs \leq 6 \ast \text{length } xs + 1 \]
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:
\[\text{t-heap-of-B } xs + \text{t-list-of-B } (\text{heap-of-B } xs)\]
\[\leq 3 \times \text{length } xs \times (\text{\(n\log2\)} (\text{length } xs + 1) + 3) + 1\]

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

proof

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

qed

end

References

