Priority Queues Based on Braun Trees

Tobias Nipkow

April 20, 2020

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 Introduction .................................. 2
1.2 Get Minimum .................................. 2
1.3 Insertion ...................................... 2
1.4 Deletion ...................................... 3

2 Priority Queues Based on Braun Trees 2

2.1 Function del-min2 ............................ 6
2.2 Correctness Proof ............................ 6

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

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 \( r \)), \( \text{size}(r) \leq \text{size}(l) \leq \text{size}(r) + 1 \), 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 \( \text{height-size-braun} \): \( \text{braun } t \Rightarrow 2^\text{height } t \leq 2 \times \text{size } t + 1 \)

proof (induction \( t \))
\begin{itemize}
  \item case \( \text{(Node } t1 \) \)
    \begin{itemize}
      \item show \( ?\text{case} \)
      \begin{itemize}
        \item case \( 0 \) thus \( ?\text{thesis} \) using \( \text{Node} \) by simp
        \item next
        \begin{itemize}
          \item case \( \text{(Suc } n \) \)
          \begin{itemize}
            \item hence \( 2^n \leq \text{size } t1 \) using \( \text{Node} \) by simp
            \item thus \( ?\text{thesis} \) using \( \text{Suc Node} \) by(auto simp: \text{max-def})
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
qed simp

1.2 Get Minimum

lemma \( \text{get-min} \): \( [\text{heap } h; \ h \neq \text{Leaf} ] \Rightarrow \text{value } h = \text{Min-mset } (\text{mset-tree } h) \) by (auto simp add: \text{eq-Min-iff neq-Leaf-iff})

1.3 Insertion

hide-const (open) insert

fun insert :: \( 'a::linorder \Rightarrow 'a \text{ tree } \Rightarrow 'a \text{ tree} \) where
\begin{itemize}
  \item insert \( a \text{ Leaf} = \text{Node } \text{Leaf } a \text{ Leaf} | \)
  \item insert \( a \text{ (Node } l x r) = \\
    \text{(if } a < x \text{ then Node } (\text{insert } x r) a l \text{ else Node } (\text{insert } a r) x l) \)
\end{itemize}

value fold insert [0::int,1,2,3,−55,−5] Leaf

lemma \( \text{size-insert} \): \( \text{size } (\text{insert } x t) = \text{size } t + 1 \)
by(induction \( t \) arbitrary: \( x \) ) auto

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

lemma \( \text{set-insert} \): \( \text{set-tree } (\text{insert } x t) = \{x\} \cup (\text{set-tree } t) \)
by(induction \( t \) arbitrary: \( x \) ) auto

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

lemma heap-insert: heap t \implies 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 \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:
del-left t = (x, t') = \Rightarrow t \neq Leaf \Rightarrow 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') = \Rightarrow t \neq Leaf \Rightarrow x \in# mset-tree t \land mset-tree t' = mset-tree t - \{#x#\}
by (simp add: del-left-mset-plus)

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

lemma del-left-heap:
del-left t = (x, t') = \Rightarrow t \neq Leaf \Rightarrow heap t \implies 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') = \Rightarrow t \neq Leaf \Rightarrow 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') = \Rightarrow braun t \Rightarrow t \neq Leaf \Rightarrow 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 \Rightarrow 'a \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 simp
apply (fastforce split: prod.split intro!: braun-sift-down dest: del-left-size del-left-braun)
done

lemma heap-del-min:
  heap l ⇒ heap r ⇒ heap (del-min t)
apply simp
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]: (3 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 ≠ Leaf
shows mset-tree(del-min t) = mset-tree t - { #value 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]: (3 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 = λ h. h = Leaf
and insert = insert and del-min = del-min
and get-min = value and invar = λ h. braun h ∧ 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)
case 5 thus \texttt{?case using get-min mset-tree.simps(1) by blast}
next
case 6 thus \texttt{?case by(simp)}
next
case 7 thus \texttt{?case by(simp add: heap-insert braun-insert)}
next
case 8 thus \texttt{?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 Fillitre with the help of the Why3 system http://toccata.lri.fr/gallery/braun_trees.en.html. Only the deletion function (\texttt{del-min2} below) differs from Paulson’s version. But the difference turns out to be minor — see below.

2.1 Function \texttt{del-min2}

\texttt{fun le-root :: 'a::linorder ⇒ 'a tree ⇒ bool where}
\texttt{le-root a t = (t = Leaf ∨ a ≤ value t)}

\texttt{fun replace-min :: 'a::linorder ⇒ 'a tree ⇒ 'a tree where}
\texttt{replace-min x (Node l - r) =}
\texttt{if le-root x l & le-root x r then Node l x r}
\texttt{else}
\texttt{let a = value l in}
\texttt{if le-root a r then Node (replace-min x l) a r}
\texttt{else Node l (value r) (replace-min x r))}

\texttt{fun merge :: 'a::linorder tree ⇒ 'a tree ⇒ 'a tree where}
\texttt{merge l Leaf = l |}
\texttt{merge (Node l1 a1 r1) (Node l2 a2 r2) =}
\texttt{(if a1 ≤ a2 then Node (Node l2 a2 r2) a1 (merge l1 r1)
else let (x, l') = del-left (Node l1 a1 r1)
\texttt{in Node (replace-min x (Node l2 a2 r2)) a2 l')}}

\texttt{fun del-min2 where}
\texttt{del-min2 Leaf = Leaf |}
\texttt{del-min2 (Node l x r) = merge l r}

2.2 Correctness Proof

It turns out that \texttt{replace-min} is just \texttt{sift-down} in disguise:
lemma replace-min-sift-down: braun (Node l a r) ⇒ replace-min x (Node l a r) = sift-down l x r
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:
  braun (Node l x r) ⇒ mset-tree (merge l r) = mset-tree l + 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:
[ [ braun (Node l x r); heap l; heap r ] ] ⇒ heap (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 ≤ a2
  next
    assume ¬ a1 ≤ a2
    let ?l = Node l1 a1 r1 let ?r = Node l2 a2 r2
    have braun ?r using 2.prems (1) by auto
    obtain x' l' where dl: del-left ?l = (x, l') by (metis surj-pair)
    from del-left-heap[OF this - 2.prems (2)] have heap l' by auto
    have hr: heap (replace-min x ?r) using braun ?r: 2.prems (3)
    by (simp add: heap-sift-down neq-Leaf-iff replace-min-sift-down del: replace-min.simps)
    have 0: ∀ x ∈ set-tree ?l. a2 ≤ x using 2.prems (2) (¬ a1 ≤ a2) by (auto simp: ball-Un)
    moreover have set-tree l' ⊆ set-tree ?l x ∈ set-tree ?l
    ultimately have 1: ∀ x ∈ set-tree l'. a2 ≤ x by blast
    have ∀ x ∈ set-tree ?r. a2 ≤ x using (heap ?r) by auto
    thus ?thesis
    using (¬ a1 ≤ a2) dl (heap (replace-min x ?r)); (heap l') x ∈ set-tree ?l \ 0 1
    (braun ?r)
    by (auto simp: mset-sift-down replace-min-sift-down simp flip: set-mset-tree del: replace-min.simps)
  qed
next
lemma del-left-braun-size:
  del-left t = (x,t') \implies braun t \implies t \neq Leaf \implies 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) \implies 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:! 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 = value 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
begin

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 \(O(n\log n)\) fashion and generates a Braun tree:

\[
\begin{align*}
\text{fun heap-of-A :: ('a::linorder) list ⇒ 'a tree where} \\
\text{heap-of-A [] = Leaf} \\
\text{heap-of-A (a#as) = insert a (heap-of-A as)} \\
\end{align*}
\]

\[
\text{lemma heap-heap-of-A: heap (heap-of-A xs)} \\
\text{by (induction xs)(simp-all add: heap-insert)}
\]

\[
\text{lemma braun-heap-of-A: braun (heap-of-A xs)} \\
\text{by (induction xs)(simp-all add: braun-insert)}
\]

\[
\text{lemma mset-tree-heap-of-A: mset-tree (heap-of-A xs) = mset xs} \\
\text{by (induction xs)(simp-all add: mset-insert)}
\]

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

\[
\begin{align*}
\text{fun t-insert :: ('a::linorder ⇒ 'a tree ⇒ nat where} \\
t-insert a Leaf = 1 \\
t-insert a (Node l x r) = \\
\quad (if a < x then 1 + t-insert x r else 1 + t-insert a r) \\
\end{align*}
\]

\[
\begin{align*}
\text{fun t-heap-of-A :: ('a::linorder list ⇒ nat where} \\
t-heap-of-A [] = 0 \\
t-heap-of-A (a#as) = t-insert a (heap-of-A as) + t-heap-of-A as
\end{align*}
\]

\[
\text{lemma t-insert-height:} \\
t-insert x t ≤ height t + 1 \\
\text{apply (induct t arbitrary: x; simp)} \\
\text{apply (simp only: max-Suc-Suc[ symmetric] le-max-iff-disj, simp)} \\
\text{done}
\]
lemma height-insert-ge:
  height \( t \) \( \leq \) height (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:
  \( t \)-heap-of-A \( xs \) \( \leq \) length \( xs \) \( \times \) (height (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 \( ?lhs \leq ?lhs \) 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:
  size (heap-of-A \( xs \)) \( = \) length \( xs \)

using arg-cong[OF mset-tree-heap-of-A, of size \( xs \)]
by simp

lemma \( t \)-heap-of-A-log-bound:
  \( t \)-heap-of-A \( xs \) \( \leq \) length \( xs \) \( \times \) (nlog2 (length \( xs \) \( + \) 1) \( + \) 1)

using t-heap-of-A-bound[of \( xs \)]
balanced-if-braun[OF braun-heap-of-A, of \( xs \)]
by (simp add: height-balanced size1-size size-heap-of-A)

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

fun heapify :: nat \( \Rightarrow \) ('a::linorder) list \( \Rightarrow \) 'a tree \( \times \) 'a list
  where
heapify 0 \( xs \) = (Leaf, \( xs \)) |
heapify (Suc \( n \)) \( x \#\) \( xs \) =
  (let \( l \), \( ys \) = heapify (Suc \( n \) div 2) \( xs \);
     \( r \), \( zs \) = heapify (n div 2) \( ys \)
in (sift-down \( l \) \( x \) \( r \), \( zs \)))

The result should be a Braun tree:

lemma heapify-snd:
  \( n \leq \) length \( xs \) \( \Rightarrow \) snd (heapify \( n \) \( xs \)) \( = \) drop \( n \) \( xs \)
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=λn. drop n xs for xs])
apply simp
done

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_induct)
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 ys1 = heapify-snd-tup[of h1 le1]
from len have le2: n div 2 ≤ length ys1
  by (simp add: ys1)

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

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

from 2.prems 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 (clarsimp simp: heap-sift-down[of braun])
done
qed simp-all

lemma braun-heapify:
  n ≤ length xs ⇒ braun (fst (heapify n xs))
by (cases heapify n xs, drule (1) heapify-correct, simp)

lemma heap-heapify:
  n ≤ length xs ⇒ heap (fst (heapify n xs))
by (cases heapify n xs, drule (1) heapify-correct, simp)

lemma mset-heapify:
  n ≤ length xs ⇒ mset-tree (fst (heapify n xs)) = mset (take n 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 ⇒ 'a ⇒ 'a tree ⇒ nat where
t-sift-down Leaf a Leaf = 1 |
t-sift-down (Node Leaf x Leaf) a Leaf = 2 |
t-sift-down (Node l1 x1 r1 =: t1) a (Node l2 x2 r2 =: t2) =
  (if a ≤ x1 ∧ a ≤ x2 then 1
   else if x1 ≤ x2 then 1 + t-sift-down l1 a r1
   else 1 + t-sift-down l2 a r2)
by pat-completeness auto

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

end

fun t-heapify :: nat ⇒ ('a::linorder) list ⇒ nat where
t-heapify 0 xs = 1 |
t-heapify (Suc n) (x#xs) =
  (let (l, ys) = heapify (Suc n div 2) xs;
      t1 = t-heapify (Suc n div 2) xs;
      (r, zs) = heapify (n div 2) ys;
      t2 = t-heapify (n div 2) ys
  )
in 1 + t1 + t2 + t-sift-down l x r)

**Lemma** t-sift-down-height:

\[\text{braid} (\text{Node} \ l \times r) \implies \text{t-sift-down} l x r \leq \text{height} (\text{Node} \ l \times r)\]

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

**Lemma** sift-down-height:

\[\text{braid} (\text{Node} \ l \times r) \implies \text{height} (\text{sift-down} l x r) \leq \text{height} (\text{Node} \ l \times r)\]

**by** (induct l x r rule: sift-down.induct; auto simp: Let-def)

**Lemma** braun-height-r-le:

\[\text{braid} (\text{Node} \ l \times r) \implies \text{height} r \leq \text{height} l\]

**by** (rule balanced-optimal, auto intro: balanced-if-braun)

**Lemma** braun-height-l-le:

assumes b: braun (Node l x r)

shows height l \leq Suc (height r)

**using** b braun-height-r-le [OF b]

**by** (simp add: balanced-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:

\[i \leq \text{length} \ xs \implies \text{t-heapify} \ i \ xs + \text{height} (\text{fst} (\text{heapify} \ i \ xs)) \leq 5 * i + 1\]

**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.prems heapify-snd-tup[OF h1]

**have** le1: Suc i div 2 \leq \text{length} \ xs

**and** le2: i div 2 \leq \text{length} \ xs

**and** le4: i div 2 \leq \text{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 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 l1 Leaf = l1 |
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-heap:
  heap l ⇒ heap r ⇒ 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 ⇒ 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 $O(n \log n)$? Although merge does not preserve braun, it cannot increase the height of the heap.

lemma merge-height:
  height (merge l r) ≤ Suc (max (height l) (height r))
  by (induct rule: merge.induct, auto)
corollary merge-height-display:
  height (merge l r) ≤ height (Node l z 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 ≤ a2 then 1 + t-merge l1 r1
      else 1 + t-merge l2 r2)

fun t-list-of-A :: ('a::linorder) tree ⇒ nat where
  t-list-of-A Leaf = 0 |
  t-list-of-A (Node l a r) = 1 + t-merge l r + t-list-of-A (merge l r)

lemma t-merge-height:
  t-merge l r ≤ max (height l) (height r)
  by (induct rule: t-merge.induct, auto)

lemma t-list-of-A-induct:
  height t ≤ n ⇒ t-list-of-A t ≤ 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=t and r=r in t-merge-height)
    apply linarith
  done

lemma t-list-of-A-bound:
  t-list-of-A t ≤ 2 * height t * size t
  by (rule t-list-of-A-induct, simp)

lemma t-list-of-A-log-bound:
  braun t ⇒ t-list-of-A t ≤ 2 * nlog2 (size t + 1) * size t
  using t-list-of-A-bound[of t]
  by (simp add: height-balanced balanced-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

16
also note \( t\text{-heap-of-A-log-bound[of xs]} \)
also note \( t\text{-list-of-A-log-bound[of heap-of-A xs, OF braun-heap-of-A]} \)
finally show \( ?\text{thesis} \)
by (simp add: size-heap-of-A)
qed

Running time of algorithm B:

function \( \text{list-of-B :: ('a::linorder) tree ⇒ 'a list where} \)
\( \text{list-of-B Leaf = [] |} \)
\( \text{list-of-B (Node l a r) = a # list-of-B (del-min (Node l a r))} \)
by pat-completeness auto

lemma \( \text{list-of-B-braun-ptermination:} \)
\( \text{braun t ⇒ list-of-B-dom t} \)
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 \( \text{list-of-B-braun-simps} = \text{list-of-B-psimps[of heap-of-A-braun-ptermination]} \)

lemma \( \text{mset-list-of-B:} \)
\( \text{braun t ⇒ mset (list-of-B t) = mset-tree t} \)
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 \( \text{set-list-of-B:} \)
\( \text{braun t ⇒ set (list-of-B t) = set-tree t} \)
by (simp only: set-mset-mset[symmetric] mset-list-of-B, simp)

lemma \( \text{sorted-list-of-B:} \)
\( \text{braun t ⇒ heap t ⇒ sorted (list-of-B t)} \)
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 \( \text{heap-of-B xs = fst (heapify (length xs) xs)} \)

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

lemma \( \text{msetB: mset (list-of-B (heap-of-B xs)) = mset xs} \)
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 simp: prod.split)
  apply (frule del-left-braun, simp+)
  apply (frule del-left-size, 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)
  done
lemmas t-list-of-B-braun-simps
  = t-list-of-B.psimp[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 simp: 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 \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 (nlog2 (size t + 1) + 1) \ast size t
apply (frule t-list-of-B-bound)
apply (simp add: height-balanced balanced-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 \ast 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)
\[ \leq 3 \times \text{length } xs \times (\log_2 (\text{length } xs + 1) + 3) + 1 \]

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

proof –

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

end

References

