Priority Queues Based on Braun Trees

Tobias Nipkow

May 28, 2015

Abstract

This theory implements priority queues via Braun trees. Insertion and deletion take logarithmic time and preserve the balanced nature of Braun trees.

Contents

1 Priority Queues Based on Braun Trees 1
  1.1 Introduction .............................................. 1
  1.2 Multiset of tree ......................................... 2
  1.3 Braun predicate ......................................... 2
  1.4 Insertion ................................................ 3
  1.5 Deletion ................................................ 3

1 Priority Queues Based on Braun Trees

theory Priority-Queue-Braun
imports ~~/src/HOL/Library/Tree
~~/src/HOL/Library/Multiset

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, including the logarithmic bounds.

lemma size-0-iff-Leaf[simp]: size \( t = 0 \) \( \iff \) \( t = \text{Leaf} \)
by(cases \( t \)) auto

fun height :: 'a tree \Rightarrow \text{nat}


height Leaf = 0 |
height (Node l x r) = max (height l) (height r) + 1

lemma size1-height: size t + 1 ≤ 2 ^ height t
proof (induction t)
case (Node l a r)
show ?case
proof (cases height l ≤ height r)
case True
have size(Node l a r) + 1 = (size l + 1) + (size r + 1) by simp
also have size l + 1 ≤ 2 ^ height l by (rule Node.IH (1))
also have size r + 1 ≤ 2 ^ height r by (rule Node.IH (2))
also have (2 :: nat) ^ height l ≤ 2 ^ height r using True by simp
finally show ?thesis using True by (auto simp: max-def mult-2)
next
case False
have size(Node l a r) + 1 = (size l + 1) + (size r + 1) by simp
also have size l + 1 ≤ 2 ^ height l by (rule Node.IH (1))
also have size r + 1 ≤ 2 ^ height r by (rule Node.IH (2))
also have (2 :: nat) ^ height r ≤ 2 ^ height l using False by simp
finally show ?thesis using False by (auto simp: max-def mult-2)
qed
qed simp

fun heap :: 'a::linorder tree ⇒ bool where
heap Leaf = True |
heap (Node l m r) =
  (heap l ∧ heap r ∧ (∀ x ∈ set-tree l ∪ set-tree r. m ≤ x))

1.2 Multiset of tree
definition mset-tree :: 'a tree ⇒ 'a multiset where
mset-tree t = multiset-of (inorder t)

lemma mset-Leaf[simp]: mset-tree Leaf = {#}
by (simp add: mset-tree-def)

lemma mset-Node[simp]:
mset-tree (Node l x r) = {#x#} + mset-tree l + mset-tree r
by (simp add: mset-tree-def ac-simps)

lemma set-mset-tree: set-of (mset-tree t) = set-tree t
by (simp add: mset-tree-def)

lemma mset-iff-set-tree: x ∈# mset-tree t ⟷ x ∈ set-tree t
by (induction t arbitrary: x) auto

1.3 Braun predicate
fun braun :: 'a tree ⇒ bool where
braun Leaf = True | 
braun (Node l x r) = (size r ≤ size l ∧ size l ≤ Suc(size r) ∧ braun l ∧ braun r)

lemma height-size-braun: braun t ⟷ 2 ^ (height t) ≤ 2 * size t + 1
proof (induction t)
case (Node t1)
show ?case
proof (cases height t1)
case 0 thus ?thesis using Node by simp
next
case (Suc n)
hence 2 ^ n ≤ size t1 using Node by simp
thus ?thesis using Suc Node by (auto simp: max-def)
qed
qed simp

1.4 Insertion

fun insert-pq :: 'a::linorder ⇒ 'a tree ⇒ 'a tree where
insert-pq a Leaf = Node Leaf a Leaf |
insert-pq a (Node l x r) = (if a < x then Node (insert-pq x r) a l else Node (insert-pq a r) x l)

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

lemma size-insert-pq[simp]: size(insert-pq x t) = size t + 1
by (induction t arbitrary: x) auto

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

lemma set-insert-pq[simp]: set-tree (insert-pq x t) = insert x (set-tree t)
by (induction t arbitrary: x) auto

lemma braun-insert-pq: braun t ⟷ braun (insert-pq x t)
by (induction t arbitrary: x) auto

lemma heap-insert-pq: heap t ⟷ heap (insert-pq x t)
by (induction t arbitrary: x) (auto simp add: ball-Un)

1.5 Deletion

fun del-left :: 'a tree ⇒ 'a * 'a tree where
del-left (Node Leaf x Leaf) = (x,Leaf) |
del-left (Node l x r) = (let (y,l') = del-left l in (y,Node r x l'))

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

lemma del-left-braun:
  del-left t = (x,t') ===> braun t ===> t \neq Leaf ===> braun t'
apply(induction t arbitrary; x t' rule: del-left.induct)
apply(fastforce dest: del-left-size split: prod.splits)+
done

lemma del-left-elem:
  del-left t = (x,t') ===> braun t ===> t \neq Leaf
  ===> t \neq Leaf
  ===> x \in set-tree t
apply(induction t arbitrary; x t' rule: del-left.induct)
apply(fastforce split: prod.splits)+
done

lemma del-left-set:
  del-left t = (x,t') ===> braun t ===> t \neq Leaf
  ===> set-tree t = insert x (set-tree t')
apply(induction t arbitrary; x t' rule: del-left.induct)
apply(fastforce split: prod.splits)+
done

lemma del-left-mset:
  del-left t = (x,t') ===> braun t ===> t \neq Leaf
  ===> mset-tree t' = mset-tree t - {#x#}
apply(induction t arbitrary; x t' rule: del-left.induct)
  apply(auto simp: ac-simps mset-iff-set-tree[symmetric]
      dest!: del-left-elem split: prod.splits)
  apply(simp add: mset-eq-iff)
  apply(simp add: mset-eq-iff)
  apply(simp add: mset-eq-iff)
  apply(fastforce simp: mset-eq-iff)
done

lemma del-left-heap:
  del-left t = (x,t') ===> heap t ===> braun t ===> t \neq Leaf ===> heap t'
proof(induction t arbitrary; x t' rule: del-left.induct)
case (2-1 ll a br b r)
  from 2-1.prems(1) obtain l' where
    del-left (Node ll a br b r) = (x,l') and [simp]: t' = Node r b l'
    by(auto split: prod.splits)
  from del-left-set[OF this(1)] 2-1.IH[OF this(1)] 2-1.prems
  show ?case by(auto)
next
case 2-2 thus ?case by(fastforce dest: del-left-set split: prod.splits)
next
qed auto

function (sequential) sift-down :: 'a::linorder tree \Rightarrow 'a \Rightarrow 'a tree where
sift-down Leaf a Leaf = Node Leaf a Leaf |
sift-down (Node Leaf x Leaf) 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) a (Node l2 x2 r2) =
  (if a ≤ x1 ∧ a ≤ x2
   then Node (Node l1 x1 r1) a (Node l2 x2 r2)
   else if x1 ≤ x2 then Node (sift-down l1 a r1) x1 (Node l2 x2 r2)
   else Node (Node l1 x1 r1) x2 (sift-down l2 a r2))
by pat-completeness auto
termination
by (relation measure (%(l,a,r). size l + size r)) auto

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

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)

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)

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

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)

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 ⇒ braun t ⇒ 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 (3 ll b lr a r) [simp]
{ 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 heap t t ≠ Leaf
shows mset-tree t = {#val t#} + mset-tree(del-min t)
proof(cases t rule: del-min.cases)
case 1 with assms show ?thesis by simp
next
case 2 with assms show ?thesis by simp
next
case (3 ll b lr a r) [simp]
{ fix y l' assume del: del-left (Node ll b lr) = (y, l')
  have mset-tree t = {#a#} + mset-tree(sift-down r y l')
    using assms del-left-mset[OF del] del-left-size[OF del]
    del-left-braun[OF del] del-left-elem[OF del]
    by (subst mset-sift-down)
      (auto simp: ac-simps multiset-eq-iff mset-iff-set-tree[symmetric]) }
thus ?thesis by (auto split: prod.split)
qed

lemma set-del-min: [ braun t; heap t; t ≠ Leaf ]
⇒ set-tree t = insert (val t) (set-tree(del-min t))
by (drule (2) arg-cong[where f=set-of, OF mset-del-min]) (simp add: set-mset-tree)

end

References


R. Bird, C. Morgan, and J. Woodcock, editors, Mathematics of Program
Construction, Second International Conference, volume 669 of LNCS,