Priority Queues Based on Braun Trees

Tobias Nipkow

September 19, 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 Multiset of Elements of Binary Tree 1

2 Priority Queues Based on Braun Trees 2
  2.1 Introduction .............................................. 2
  2.2 Braun predicate ......................................... 3
  2.3 Insertion .................................................. 3
  2.4 Deletion ................................................... 4

1 Multiset of Elements of Binary Tree

theory Tree-Multiset
imports Multiset Tree
begin

  Kept separate from theory Tree to avoid importing all of theory Multiset into Tree. Should be merged if Multiset ever becomes part of Main.

  fun mset-tree :: 'a tree ⇒ 'a multiset where
  mset-tree Leaf = {#} |
  mset-tree (Node l a r) = {#a#} + mset-tree l + mset-tree r

  lemma set-mset-tree[simp]: set-mset (mset-tree t) = set-tree t
  by(induction t) auto

  lemma size-mset-tree[simp]: size(mset-tree t) = size t
  by(induction t) auto

  lemma mset-map-tree: mset-tree (map-tree f t) = image-mset f (mset-tree t)
  by (induction t) auto
Lemma mset-iff-set-tree: \( x \in \# mset\text{-}tree t \longleftrightarrow x \in \text{set}\text{-}tree t \)
by (induction t arbitrary: \( x \)) auto

Lemma mset-preorder[simp]: \( \text{mset (preorder } t) = \text{mset}\text{-}tree t \)
by (induction t) (auto simp: ac-simps)

Lemma mset-inorder[simp]: \( \text{mset (inorder } t) = \text{mset}\text{-}tree t \)
by (induction t) (auto simp: ac-simps)

Lemma map-mirror: \( \text{mset}\text{-}tree (\text{mirror } t) = \text{mset}\text{-}tree t \)
by (induction t) (simp-all add: ac-simps)

end

2 Priority Queues Based on Braun Trees

case study Priority-Queue-Braun imports
 ~/src/HOL/Library/Tree-Multiset begin

2.1 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.

fun height :: 'a tree ⇒ nat where
height Leaf = 0 |
height (Node l x r) = max (height l) (height r) + 1

Lemma size1-height: \( \text{size } t + 1 \leq 2 \cdot \text{height } t \)
proof (induction t)
case (Node l a r)
show ?case
proof (cases height \( l \leq \text{height } r \))
case True
  have size(Node l a r) + 1 = (size \( l + 1 \) + (size \( r + 1 \)) by simp
also have size \( l + 1 \leq 2 \cdot \text{height } l \) by (rule Node.IH(1))
also have size \( r + 1 \leq 2 \cdot \text{height } r \) by (rule Node.IH(2))
also have (2::nat) \( \cdot \text{height } l \leq 2 \cdot \text{height } r \) 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 $size_l + 1 \leq 2 \cdot height_l$ by (rule Node.IH(1))
also have $size_r + 1 \leq 2 \cdot height_r$ by (rule Node.IH(2))
also have $(2\cdot\text{nat}) \cdot height_r \leq 2 \cdot height_l$ using False by simp
finally show ?thesis using False by (auto simp: max-def mult-2)
qed
qed simp

2.2 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 =⇒ $2 \cdot (height t) \leq 2 \cdot size t + 1$
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 $2 \cdot n \leq size t1$ using Node by simp
thus ?thesis using Suc Node (auto simp: max-def)
qed
qed simp

2.3 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)
2.4 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 ≠ 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 ≠ Leaf ⇒ x ∈ 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 ≠ 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 ≠ 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: multiset-eq-iff)
apply(simp add: multiset-eq-iff)
apply(simp add: multiset-eq-iff)
apply(fastforce simp: multiset-eq-iff)
done

lemma del-left-heap:
  del-left t = (x, t') ⇒ heap t ⇒ braun t ⇒ t ≠ 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) = (x, l') and [simp]: t' = Node r b l'
apply(auto split: prod.splits)
from del-left-set[OH this(1)] 2-1.IH[OH 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 ⇒ 'a ⇒ '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-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)

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 heap 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 add: size-0-iff-Leaf)
next
  case [simp]: (3 ll b lr a r)
  { fix y l' assume 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)
    } thus ?thesis by (auto split: prod.split)
qed (insert assms, auto)

lemma mset-del-min: assumes braun t heap t t \neq 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 add: size-0-iff-Leaf)
next
  case [simp]: (3 ll b lr a r)
  { 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)
    } thus ?thesis by (auto simp add: ac-simps multiset-eq-iff mset-iff-set-tree[ symmetric])
qed

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

end
References

