Skew Heap

Tobias Nipkow

May 28, 2015

Abstract

Skew heaps are an amazingly simple and lightweight implementation of priority queues. They were invented by Sleator and Tarjan [1] and have logarithmic amortized complexity. This entry provides executable and verified functional skew heaps.

The amortized complexity of skew heaps is analyzed in the AFP entry Amortized Complexity.

Contents

1 Multiset of Elements of Binary Tree 1

2 Skew Heap 2
  2.1 Get Minimum .................................................. 2
  2.2 Meld ............................................................ 3
  2.3 Insert ........................................................... 4
  2.4 Delete minimum ............................................... 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-of-mset-tree[simp]: set-of (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 multiset-of-preorder[simp]: multiset-of (preorder t) = mset-tree t
by (induction t) (auto simp: ac-simps)

lemma multiset-of-inorder[simp]: multiset-of (inorder t) = mset-tree t
by (induction t) (auto simp: ac-simps)

lemma map-mirror: mset-tree (mirror t) = mset-tree t
by (induction t) (simp-all add: ac-simps)

lemma del-rightmost-mset-tree:
[ del-rightmost t = (t',x); t ≠ {} ] ⇒ mset-tree t = {#x#} + mset-tree t'
apply (induction t arbitrary: t' rule: del-rightmost.induct)
by (auto split: prod.splits) (auto simp: ac-simps)

end
theory Skew-Heap
imports ~~/src/HOL/Library/Tree-Multiset
begin

2 Skew Heap

Skew heaps [1] are possibly the simplest functional priority queues that have logarithmic (albeit amortized) complexity.

The implementation below should be generalized to separate the elements from their priorities.

type-synonym 'a heap = 'a tree

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

2.1 Get Minimum

fun get-min :: 'a::linorder heap ⇒ 'a where
get-min(Node l m r) = m

lemma get-min-in:
  h ≠ Leaf ⇒ get-min h ∈ set-tree h
by (auto simp add: neq-Leaf-iff)

lemma get-min-min:
[ heap h; h ≠ Leaf ] ⇒ ∀x ∈ set-tree h. get-min h ≤ x
by(auto simp add: neq-Leaf-iff)

2.2 Meld

function meld :: ('a::linorder) heap ⇒ 'a heap ⇒ 'a heap where
meld Leaf h = h |
meld h Leaf = h |
meld (Node l1 a1 r1) (Node l2 a2 r2) =
  (if a1 ≤ a2 then Node (meld (Node l2 a2 r2) r1) a1 l1
   else Node (meld (Node l1 a1 r1) r2) a2 l2)
by pat-completeness auto

termination
by (relation measure (λ(x, y). size x + size y)) auto

lemma meld-code: meld h1 h2 =
  (case h1 of
    Leaf ⇒ h2 |
    Node l1 a1 r1 ⇒ (case h2 of
      Leaf ⇒ h1 |
      Node l2 a2 r2 ⇒
        (if a1 ≤ a2
          then Node (meld h2 r1) a1 l1
          else Node (meld h1 r2) a2 l2)))
by(auto split: tree.split)

An alternative version that always walks to the Leaf of both heaps:

function meld2 :: ('a::linorder) heap ⇒ 'a heap ⇒ 'a heap where
meld2 Leaf Leaf = Leaf |
meld2 Leaf (Node l2 a2 r2) = Node (meld2 r2 Leaf) a2 l2 |
meld2 (Node l1 a1 r1) Leaf = Node (meld2 r1 Leaf) a1 l1 |
meld2 (Node l1 a1 r1) (Node l2 a2 r2) =
  (if a1 ≤ a2
    then Node (meld2 (Node l2 a2 r2) r1) a1 l1
    else Node (meld2 (Node l1 a1 r1) r2) a2 l2)
by pat-completeness auto

termination
by (relation measure (λ(x, y). size x + size y)) auto

lemma size-meld[simp]: size(meld t1 t2) = size t1 + size t2
by(induction t1 t2 rule: meld.induct) auto

lemma size-meld2[simp]: size(meld2 t1 t2) = size t1 + size t2
by(induction t1 t2 rule: meld2.induct) auto

lemma mset-meld: mset-tree (meld h1 h2) = mset-tree h1 + mset-tree h2
by (induction h1 h2 rule: meld.induct) (auto simp add: ac-simps)

lemma set-mset-tree: set-of(mset-tree t) = set-tree t
by(induction t) auto
lemma set-meld: set-tree (meld h1 h2) = set-tree h1 ∪ set-tree h2
by (metis mset-meld set-mset-tree set-of-union)

lemma heap-meld:
heap h1 =⇒ heap h2 =⇒ heap (meld h1 h2)
by (induction h1 h2 rule: meld.induct) (auto simp: ball-Un set-meld)

2.3 Insert
definition insert :: 'a::linorder ⇒ 'a heap ⇒ 'a heap
where
insert a t = meld (Node Leaf a Leaf) t

hide-const (open) Skew-Heap.insert

lemma heap-insert: heap h =⇒ heap (Skew-Heap.insert a h)
by (simp add: insert-def heap-meld)

lemma mset-insert: heap h =⇒ mset-tree (Skew-Heap.insert a h) = {#a#} + mset-tree h
by (auto simp: mset-meld insert-def)

2.4 Delete minimum
fun del-min :: 'a::linorder heap ⇒ 'a heap
where
del-min Leaf = Leaf |
del-min (Node l m r) = meld l r

lemma heap-del-min: heap h =⇒ heap (del-min h)
by (cases h) (auto simp: heap-meld)

lemma mset-del-min: heap h =⇒ mset-tree (del-min h) = mset-tree h − {# get-min h #}
by (cases h) (auto simp: mset-meld ac-simps)

end

References