Skew Heap
Tobias Nipkow
December 17, 2016

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

fun subtrees-mset :: 'a tree ⇒ 'a tree multiset where
  subtrees-mset Leaf = {#Leaf#} |
  subtrees-mset (Node l x r) = add-mset (Node l x r) (subtrees-mset l + subtrees-mset r)
lemma set-mset-tree[simp]: set-mset (mset-tree t) = set-tree t
⟨proof⟩

lemma size-mset-tree[simp]: size (mset-tree t) = size t
⟨proof⟩

lemma mset-map-tree: mset-tree (map-tree f t) = image-mset f (mset-tree t)
⟨proof⟩

lemma mset-iff-set-tree: x ∈# mset-tree t ⟷ x ∈ set-tree t
⟨proof⟩

lemma mset-preorder[simp]: mset (preorder t) = mset-tree t
⟨proof⟩

lemma mset-inorder[simp]: mset (inorder t) = mset-tree t
⟨proof⟩

lemma map-mirror: mset-tree (mirror t) = mset-tree t
⟨proof⟩

lemma in-subtrees-mset-iff[simp]: s ∈# subtrees-mset t ⟷ s ∈ subtrees t
⟨proof⟩

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 \implies get-min h ∈ set-tree h

**proof**

**lemma get-min-min:**

\[
\text{[\ [heap h; x ∈ set-tree h ]] \implies get-min h ≤ x}
\]

**proof**

### 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)

**proof**

**termination**

**lemma meld-code:** meld h1 h2 =

\[
\text{(case h1 of}
\text{ Leaf ⇒ h2 |}
\text{ Node l1 a1 r1 ⇒ (case h2 of}
\text{ Leaf ⇒ h1 |}
\text{ Node l2 a2 r2 ⇒}
\text{ (if a1 ≤ a2 then Node (meld h2 r1) a1 l1}
\text{ else Node (meld h1 r2) a2 l2))}
\]

**proof**

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)

**proof**

**termination**

**proof**

**lemma size-meld[simp]:** size(meld t1 t2) = size t1 + size t2

**proof**

**lemma size-meld2[simp]:** size(meld2 t1 t2) = size t1 + size t2

**proof**
lemma mset-meld: mset-tree (meld h1 h2) = mset-tree h1 + mset-tree h2
⟨proof⟩

lemma set-meld: set-tree (meld h1 h2) = set-tree h1 ∪ set-tree h2
⟨proof⟩

lemma heap-meld:
heap h1 =⇒ heap h2 =⇒ heap (meld h1 h2)
⟨proof⟩

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)
⟨proof⟩

lemma mset-insert: mset-tree (Skew-Heap.insert a h) = {#a#} + mset-tree h
⟨proof⟩

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)
⟨proof⟩

lemma mset-del-min: mset-tree (del-min h) = mset-tree h - {# get-min h #}
⟨proof⟩

end

References