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 Skew Heap 1
  1.1 Get Minimum . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.2 Merge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.3 Insert . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.4 Delete minimum . . . . . . . . . . . . . . . . . . . . . . . . 3

1 Skew Heap

theory Skew-Heap
imports
  HOL-Library.Tree-Multiset
  HOL-Library.Pattern-Aliases
  HOL-Data-Structures.Priority-Queue-Specs
begin

unbundle pattern-aliases

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
1.1 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
(proof)

lemma get-min-min:
  [ heap h; x ∈ set-tree h ] ⇒ get-min h ≤ x
(proof)

lemma get-min:
  [ heap h; h ≠ Leaf ] ⇒ get-min h = Min-mset (mset-tree h)
(proof)

1.2 Merge

function merge :: ('a::linorder) heap ⇒ 'a heap ⇒ 'a heap where
merge Leaf h = h |
merge h Leaf = h |
merge (Node l1 a1 r1 =: h1) (Node l2 a2 r2 =: h2) =
  (if a1 ≤ a2 then Node (merge h2 r1) a1 l1 |
   else Node (merge h1 r2) a2 l2)
(proof)
termination
(proof)

lemma merge-code: merge 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 (merge h2 r1) a1 l1 |
         else Node (merge h1 r2) a2 l2)))
(proof)

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

function merge2 :: ('a::linorder) heap ⇒ 'a heap ⇒ 'a heap where
merge2 Leaf Leaf = Leaf |
merge2 Leaf (Node l2 a2 r2) = Node (merge2 r2 Leaf) a2 l2 |
merge2 (Node l1 a1 r1) Leaf = Node (merge2 r1 Leaf) a1 l1 |
merge2 (Node l1 a1 r1) (Node l2 a2 r2) =
  (if a1 ≤ a2 then Node (merge2 (Node l2 a2 r2) r1) a1 l1 |
   else Node (merge2 (Node l1 a1 r1) r2) a2 l2)
(proof)
termination
proof

lemma size-merge[simp]: size(merge t1 t2) = size t1 + size t2
⟨proof⟩

lemma size-merge2[simp]: size(merge2 t1 t2) = size t1 + size t2
⟨proof⟩

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

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

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

1.3 Insert

hide-const (open) insert

definition insert :: 'a::linorder ⇒ 'a heap ⇒ 'a heap where
insert a t = merge (Node Leaf a Leaf) t

lemma heap-insert: heap h =⇒ heap (insert a h)
⟨proof⟩

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

1.4 Delete minimum

fun del-min :: 'a::linorder heap ⇒ 'a heap where
del-min Leaf = Leaf |
del-min (Node l m r) = merge 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⟩

interpretation skew-heap: Priority-Queue-Merge
where empty = Leaf and is-empty = λh. h = Leaf
and merge = merge and insert = insert
and del-min = del-min and get-min = get-min
and invar = heap and mset = mset-tree
⟨proof⟩
end

References