Skew Heap

Tobias Nipkow

September 13, 2023

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 Merge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1 Skew Heap

theory Skew-Heap

imports

HOL-Library.Tree-Multiset
HOL-Library.Pattern-Aliases
HOL-Data-Structures.Heaps

begin

unbundle pattern-aliases

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

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

1.1 Merge

function merge :: ('a::linorder) tree ⇒ 'a tree ⇒ 'a tree where
merge Leaf t = t |
merge t Leaf = t |
merge (Node l1 a1 r1 =: t1) (Node l2 a2 r2 =: t2) =

1
(if \( a_1 \leq a_2 \) then Node (merge \( t_2 \) \( r_1 \)) \( a_1 \) \( l_1 \) \\
else Node (merge \( t_1 \) \( r_2 \)) \( a_2 \) \( l_2 \))

\langle proof \rangle

termination
\langle proof \rangle

\textbf{lemma} merge-code: merge \( t_1 \) \( t_2 \) =

\( \langle \text{case } t_1 \text{ of} \rangle \)
\( \text{Leaf } \Rightarrow \text{Leaf} \ |
\)
\( \text{Node } l_1 a_1 r_1 \Rightarrow \langle \text{case } t_2 \text{ of} \rangle \)
\( \text{Leaf } \Rightarrow \text{t}_1 \ |
\)
\( \text{Node } l_2 a_2 r_2 \Rightarrow \)
\( \text{(if } a_1 \leq a_2 \text{ then Node (merge } t_2 r_1 \text{) } a_1 \ l_1 \)
\( \text{else Node (merge } t_1 r_2 \text{) } a_2 \ l_2 ))\)

\langle proof \rangle

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

\textbf{function} merge2 :: (\('a::linorder\) \text{ tree } \Rightarrow \text{'a tree} \Rightarrow \text{'a tree}) \text{ where}

\( \text{merge2 Leaf Leaf } = \text{Leaf} \ |
\)
\( \text{merge2 Leaf (Node } l_2 a_2 r_2 \text{) } = \text{Node (merge2 } r_2 \text{ Leaf) } a_2 \ l_2 \ |
\)
\( \text{merge2 (Node } l_1 a_1 r_1 \text{) Leaf } = \text{Node (merge2 } r_1 \text{ Leaf) } a_1 \ l_1 \ |
\)
\( \text{merge2 (Node } l_1 a_1 r_1 \text{) (Node } l_2 a_2 r_2 \text{) =}
\)
\( \text{(if } a_1 \leq a_2 \text{ then Node (merge2 (Node } l_2 a_2 r_2 \text{) } r_1 \text{) } a_1 \ l_1 \)
\( \text{else Node (merge2 (Node } l_1 a_1 r_1 \text{) } r_2 \text{) } a_2 \ l_2 ))\)

\langle proof \rangle

termination
\langle proof \rangle

\textbf{lemma} size-merge: size(merge \( t_1 \) \( t_2 \)) = size \( t_1 \) + size \( t_2 \)

\langle proof \rangle

\textbf{lemma} size-merge2: size(merge2 \( t_1 \) \( t_2 \)) = size \( t_1 \) + size \( t_2 \)

\langle proof \rangle

\textbf{lemma} mset-merge: mset-tree (merge \( t_1 \) \( t_2 \)) = mset-tree \( t_1 \) + mset-tree \( t_2 \)

\langle proof \rangle

\textbf{lemma} set-merge: set-tree (merge \( t_1 \) \( t_2 \)) = set-tree \( t_1 \) \( \cup \) set-tree \( t_2 \)

\langle proof \rangle

\textbf{lemma} heap-merge:
\[ \llbracket \text{heap } t_1; \text{heap } t_2 \rrbracket \Rightarrow \text{heap (merge } t_1 t_2 \)

\langle proof \rangle

\textbf{interpretation} skew-heap: Heap-Merge
\textbf{where} merge = merge

\langle proof \rangle
References