A skew heap is a self-adjusting heap (priority queue)
A skew heap is a self-adjusting heap (priority queue)

Functions insert, merge and del_min have amortized logarithmic complexity.

A skew heap is a self-adjusting heap (priority queue)

Functions insert, merge and del_min have amortized logarithmic complexity.

Functions insert and del_min are defined via merge

Implementation type

Ordinary binary trees

Invariant: heap

merge \langle \rangle \ h = h
merge \ h \ \langle \rangle = h
merge \( \langle \rangle \ h = h \)
merge \( h \ \langle \rangle \ = h \)

Swap subtrees when descending:
merge \( h_1 = \langle l_1, a, r_1 \rangle \) \( h_2 = \langle l_2, b, r_2 \rangle \) =
(if \( a \leq b \) then \( \langle \text{merge} \ h_2 \ r_1, a, l_1 \rangle \)
else \( \langle \text{merge} \ h_1 \ r_2, b, l_2 \rangle \))

Function \textit{merge} terminates because ...

---

merge \( \langle \rangle \ h = h \)
merge \( h \ \langle \rangle \ = h \)

Swap subtrees when descending:
merge \( h_1 = \langle l_1, a, r_1 \rangle \) \( h_2 = \langle l_2, b, r_2 \rangle \) =
(if \( a \leq b \) then \( \langle \text{merge} \ h_2 \ r_1, a, l_1 \rangle \)
else \( \langle \text{merge} \ h_1 \ r_2, b, l_2 \rangle \))

Function \textit{merge} terminates because ...

---

merge \( \langle \rangle \ h = h \)
merge \( h \ \langle \rangle \ = h \)

else \( \langle \text{merge} \ h_1 \ r_2, b, l_2 \rangle \))

Function \textit{merge} terminates because ...

---

Skew Heap

Insert  Remove Smallest  Clear Heap

009
merge

Very similar to leftist heap but

Towards the proof

\[ r\text{heavy } \langle l, _, r \rangle = (|l| < |r|) \]

Towards the proof

\[ r\text{heavy } \langle l, _, r \rangle = (|l| < |r|) \]

\[ lpath \langle \rangle = \[] \]

\[ lpath \langle l, a, r \rangle = \langle l, a, r \rangle \# lpath l \]
Towards the proof

\( \text{rheavy} \langle l, -, r \rangle = (|l| < |r|) \)

\( \text{lpath} \langle \rangle = \langle \rangle \)

\( \text{lpath} \langle l, a, r \rangle = \langle l, a, r \rangle \# \text{lpath} l \)

\( G \ h = \text{length} \ (\text{filter} \ \text{rheavy} \ (\text{lpath} \ h)) \)

Lemma

\( 2^{G \ h} \leq |h| + 1 \)

Towards the proof

\( \text{rheavy} \langle l, -, r \rangle = (|l| < |r|) \)

\( \text{rpath} \langle \rangle = \langle \rangle \)

\( \text{rpath} \langle l, a, r \rangle = \langle l, a, r \rangle \# \text{rpath} r \)

\( D \ h = \text{length}(\text{filter} \ (\lambda p. \neg \text{rheavy} \ p) \ (\text{rpath} \ h)) \)

Lemma

\( 2^{D \ h} \leq |h| + 1 \)

Corollary

\( G \ h \leq \log_2 |h|_1 \)

Corollary

\( D \ h \leq \log_2 |h|_1 \)
The potential is the number of *rheavy* nodes:

\[
\Phi \langle \rangle = 0 \\
\Phi \langle l, -, r \rangle = \Phi l + \Phi r + (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)
\]

*merge* descends on the right \(\implies\) *rheavy* nodes are **bad**

**Lemma**

\[
t_{\text{merge}} t_1 t_2 + \Phi (\text{merge } t_1 t_2) - \Phi t_1 - \Phi t_2 \\
\leq G (\text{merge } t_1 t_2) + D t_1 + D t_2 + 1
\]
\[ t_{\text{merge}} t_1 t_2 + \Phi (\text{merge} t_1 t_2) - \Phi t_1 - \Phi t_2 \leq G (\text{merge} t_1 t_2) + D t_1 + D t_2 + 1 \leq \log_2 |\text{merge} t_1 t_2|_1 + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \]
\[ t_{\text{merge}} \ t_1 \ t_2 + \Phi (\text{merge} \ t_1 \ t_2) - \Phi \ t_1 - \Phi \ t_2 \leq G (\text{merge} \ t_1 \ t_2) + D \ t_1 + D \ t_2 + 1 \]
\[ = \log_2 |\text{merge} \ t_1 \ t_2|_1 + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \]
\[ = \log_2 (|t_1|_1 + |t_2|_1 - 1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \]
\[ \leq \log_2 (|t_1|_1 + |t_2|_1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \]
\[ \leq \log_2 (|t_1|_1 + |t_2|_1) + 2 \ast \log_2 (|t_1|_1 + |t_2|_1) + 1 \]

**because** $\log_2 \ x + \log_2 \ y \leq 2 \ast \log_2 \ (x + y)$ if $x, y > 0$
The inventors of skew heaps:
Daniel Sleator and Robert Tarjan.
Self-adjusting Heaps.

The formalization is based on
Anne Kaldewaij and Berry Schoenmakers.

A *splay tree* is a self-adjusting binary search tree.
A splay tree is a self-adjusting binary search tree.

Functions \textit{isin}, \textit{insert} and \textit{delete} have amortized logarithmic complexity.

Definition (splay)
Become wider or more separated.

**Example**
The river splayed out into a delta.

Splay tree

Implementation type = binary tree

Key operation \textit{splay a}:

1. Search for \textit{a} ending up at \textit{x}
   where \textit{x} = \textit{a} or \textit{x} is a leaf node.
2. Move \textit{x} to the root of the tree by rotations.
Splay tree

Implementation type = binary tree

Key operation *splay* a:
1. Search for a ending up at x where x = a or x is a leaf node.
2. Move x to the root of the tree by rotations.

Derived operations *isin*/insert/delete a:
1. splay a
2. Perform *isin*/insert/delete action

Key ideas

Move to root
Double rotations

Zig-zig

Zig-zag
Zig-zig

\[
\begin{array}{ccl}
\text{z} & \Downarrow & \text{x} \\
\text{y} & \text{D} & \text{A} \\
\text{x} & \text{C} & \text{B} \\
\text{A} & \text{B} & \text{C} \end{array}
\]

Functional definition

\[
splay :: 'a \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ tree}
\]

Zig-zig and zig-zag

\[
[a < y, a < z; T \neq \langle \rangle] \\
\Rightarrow \text{splay a } \langle\langle T, y, C\rangle, z, D\rangle = \\
\text{(case splay a } T \text{ of} \\
\langle A, x, B\rangle \Rightarrow \langle A, x, \langle B, y, \langle C, z, D\rangle\rangle\rangle)
\]

Zig-zag

\[
\begin{array}{ccl}
\text{z} & \Downarrow & \text{x} \\
\text{y} & \text{D} & \text{A} \\
\text{x} & \text{D} & \text{A} \\
\text{A} & \text{B} & \text{C} \end{array}
\]
Zig-zig

\[ a < y, a < z; T \neq \langle \rangle \]
\[ \implies \text{splay } a \langle \langle T, y, C \rangle, z, D \rangle = \]
\[ (\text{case } \text{splay } a \langle T, y, C \rangle \text{ of } \]
\[ \langle A, x, B \rangle \implies \langle A, x, \langle B, y, \langle C, z, D \rangle \rangle \rangle) \]

Zig-zag

\[ a < y, a < z; T \neq \langle \rangle \]
\[ \implies \text{splay } a \langle \langle T, y, C \rangle, z, D \rangle = \]
\[ (\text{case } \text{splay } a \langle T, y, C \rangle \text{ of } \]
\[ \langle A, x, B \rangle \implies \langle A, x, \langle B, y, \langle C, z, D \rangle \rangle \rangle) \]
Some base cases

\[ a < y \implies \text{splay } a \langle\langle A, a, B\rangle, y, C\rangle = \]

\[ a < y \implies \text{splay } a \langle\langle A, a, B\rangle, y, C\rangle = \langle A, a, \langle B, y, C\rangle\rangle \]

\[ a < x \implies \text{splay } a \langle\langle\rangle, x, A\rangle, y, B\rangle = \]

\[ a < x \implies \text{splay } a \langle\langle\rangle, x, A\rangle, y, B\rangle = \langle\rangle, x, \langle A, y, B\rangle\rangle \]
Functional correctness proofs
including preservation of bst

Straightforward
Potential

Sum of logarithms of the size of all nodes:
\[ \Phi \langle \rangle = 0 \]
\[ \Phi \langle l, a, r \rangle = \Phi \ l + \Phi \ r + \varphi \langle l, a, r \rangle \]
where \( \varphi \ t = \log_2 (|t| + 1) \)

Potential

Analysis of splay

Theorem
\[[\text{bst } t; \langle l, x, r \rangle \in \text{subtrees } t] \implies a\_splay \ x \ t \leq 3 \ast (\varphi \ t - \varphi \langle l, x, r \rangle) + 1\]

Amortized complexity of splay:
\[ a\_splay \ a \ t = t\_splay \ a \ t + \Phi (splay \ a \ t) - \Phi \ t \]
Analysis of splay

Theorem
\[ \text{bst } t; \langle l, x, r \rangle \in \text{subtrees } t \]
\[ \implies a\text{-splay } x \ t \leq 3 \ast (\varphi \ t - \varphi \langle l, x, r \rangle) + 1 \]

Corollary
\[ \text{bst } t; \ a \in \text{set\_tree } t \]
\[ \implies a\text{-splay } a \ t \leq 3 \ast (\varphi \ t - 1) + 1 \]

Analysis of splay

Theorem
\[ \text{bst } t; \langle l, x, r \rangle \in \text{subtrees } t \]
\[ \implies a\text{-splay } x \ t \leq 3 \ast (\varphi \ t - \varphi \langle l, x, r \rangle) + 1 \]

Corollary
\[ \text{bst } t; \ a \in \text{set\_tree } t \]
\[ \implies a\text{-splay } a \ t \leq 3 \ast (\varphi \ t - 1) + 1 \]

Corollary
\[ \text{bst } t \implies a\text{-splay } a \ t \leq 3 \ast \varphi \ t + 1 \]

Lemma
\[ t \neq \emptyset; \text{bst } t \]
\[ \implies \exists a' \in \text{set\_tree } t. \]
\[ \text{splay } a' \ t = \text{splay } a \ t \land t\text{-splay } a' \ t = t\text{-splay } a \ t \]
Definition

\[
\text{insert } x \; t = \\
\begin{cases}
  \langle \rangle, x, \langle \rangle & \text{if } t = \langle \rangle \\
  \text{else case } \text{splay } x \; t \; \text{of} \\
  \langle l, a, r \rangle \Rightarrow \\
  \quad \text{if } x = a & \langle l, a, r \rangle \\
  \quad \text{else if } x < a & \langle l, x, \langle \rangle, a, r \rangle \\
  \quad \text{else } \langle \langle l, a, \langle \rangle \rangle, x, r \rangle 
\end{cases}
\]

Counting only the cost of splay:

Lemma

\[
\text{bst } t \implies \\
\text{t-splay } a \; t + \Phi (\text{insert } a \; t) - \Phi t \leq 4 \ast \varphi t + 2
\]
Definition
\[
delete x t = 
\begin{align*}
& \text{if } t = {} & \text{then } {} \\
\text{else case } & \text{splay } x \ t \text{ of} \\
& \langle l, a, r \rangle \Rightarrow \\
& \text{if } x = a \\
& \text{then if } l = {} & \text{then } r \\
& \text{else case } & \text{splay} \text{ max } l \text{ of} \\
& \langle l', m, r' \rangle \Rightarrow \langle l', m, r \rangle \\
& \text{else } \langle l, a, r \rangle \\
\end{align*}
\]

Lemma
\[
bst t \Rightarrow \\
t.\text{delete } a \ t + \Phi (\text{delete } a \ t) - \Phi t \leq 6 \ast \varphi t + 2
\]

\[
isin :: 'a \text{ tree } \Rightarrow 'a \Rightarrow \text{ bool}
\]

Single threaded \Rightarrow isin t a eats up t

Remember

Amortized analysis is only correct for single threaded uses of a data structure.

Otherwise:
\[
\begin{align*}
& \text{let } counter = 0; \\
& \text{bad} = \text{increment } counter \ 2^n - 1 \ \text{times}; \\
& \_ = \text{incr bad}; \\
& \_ = \text{incr bad}; \\
& \_ = \text{incr bad}; \\
& : \\
\end{align*}
\]

Solution 1:
\[
isin :: 'a \text{ tree } \Rightarrow 'a \Rightarrow \text{ bool } \times 'a \text{ tree}
\]

Observer function returns new data structure:
Solution 1:

\[ \text{isin} :: 'a \text{ tree} \Rightarrow 'a \Rightarrow \text{bool} \times 'a \text{ tree} \]

Observer function returns new data structure:

Definition

\[
\text{isin}\ t\ a = \\
(\text{let } t' = \text{splay} a\ t\ \text{in}\ (\text{case } t'\ \text{of}\\n\quad \langle \rangle \Rightarrow \text{False}\\n\quad |\langle l, x, r\rangle \Rightarrow a = x, t')\\n\)
\]

Solution 2:

\[ \text{isin} = \text{splay}; \text{is_root} \]

Client uses \text{splay} before calling \text{is_root}:

Definition

\[ \text{is_root} :: 'a \Rightarrow 'a \text{ tree} \Rightarrow \text{bool} \]

\[ \text{is_root}\ a\ t = (\text{case } t\ \text{of}\\n\quad \langle \rangle \Rightarrow \text{False}\\n\quad |\langle l, x, r\rangle \Rightarrow x = a) \]

May call \text{is_root} \_\_ t multiple times (with the same t!) because \text{is_root} takes constant time
Splay trees have an imperative flavour and are a bit awkward to use in a purely functional language.

**Sources**

The inventors of splay trees:
Daniel Sleator and Robert Tarjan.

The formalization is based on
Berry Schoenmakers. A Systematic Analysis of Splaying.

**Implementation type**

```plaintext
datatype 'a heap = Empty | Hp 'a ('a heap list)
```
Implementation type

datatype 'a heap = Empty | Hp 'a ('a heap list)

Heap invariant:
 pheap Empty = True
 pheap (Hp x hs) =
 (∀ h∈set hs. (∀ y∈set_heap h. x ≤ y) ∧ pheap h)

Also: Empty must only occur at the root

merge and insert

merge h Empty = h
merge Empty h = h
merge (hx = Hp x hxs) (hy = Hp y hsy) =
(if x < y then Hp x (hy # hxs) else Hp y (hx # hsy))

merge_pairs and del_min

merge_pairs [] = Empty
merge_pairs [h] = h
merge_pairs (h₁ # h₂ # hs) =
merge (merge h₁ h₂) (merge_pairs hs)
merge_pairs and del_min

merge_pairs [] = Empty
merge_pairs [h] = h
merge_pairs (h₁ # h₂ # hs) =
merge (merge h₁ h₂) (merge_pairs hs)

del_min Empty = Empty
del_min (Hp x hs) = merge_pairs hs
merge_pairs [] = Empty
merge_pairs [h] = h
merge_pairs (h₁, #: h₂, #: hs) =
    (merge_pairs h₁, #: h₂, #: merge_pairs hs)

Analysis
del_min

Analysis
easier (more uniform) if a pairing heap is viewed as a binary tree:

homs :: 'a heap list \rightarrow 'a tree
homs [] = \{

\}

homs (Hp x hs₁, #: hs₂) = (homs hs₁, x, homs hs₂)

hom :: 'a heap \rightarrow 'a tree
hom Empty = \{
\}

del_min (Hp x hs) = (homs hs, x, \{\})

Analysis
easier (more uniform) if a pairing heap is viewed as a binary tree:

homs :: 'a heap list \rightarrow 'a tree
homs [] = \{

\}

homs (Hp x hs₁, #: hs₂) = (homs hs₁, x, homs hs₂)

hom :: 'a heap \rightarrow 'a tree
hom Empty = \{
\}

hom (Hp x hs) = (homs hs, x, \{\})

Potential function same as for splay trees
Verified:

The functions \textit{insert}, \textit{del.min} and \textit{merge} all have \(O(\log_2 n)\) amortized complexity.

These bounds are not tight. Better amortized bounds in the literature:
\(\text{insert} \in O(1), \text{del.min} \in O(\log_2 n), \text{merge} \in O(1)\)

The exact complexity is still open.

Verified:

The functions \textit{insert}, \textit{del.min} and \textit{merge} all have \(O(\log_2 n)\) amortized complexity.

Sources

The inventors of the pairing heap:
Sources

The inventors of the pairing heap:

The Pairing Heap: A New Form of Self-Adjusting Heap.
Algorithmica, 1986.

The functional version:

Chris Okasaki. Purely Functional Data Structures.

Verified:

The functions *insert*, *del_min* and *merge* all have
$O(\log_2 n)$ amortized complexity.

These bounds are not tight.
Better amortized bounds in the literature:

*insert* ∈ $O(1)$, *del_min* ∈ $O(\log_2 n)$, *merge* ∈ $O(1)$
Thys/Pairing_Heap.thy