Verification of Functional Data Structures

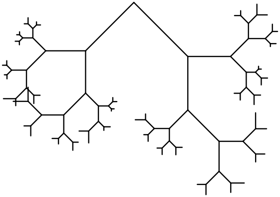
Correctness and Complexity

Tobias Nipkow

Technical University of Munich

Tobias Nipkow, Jasmin Blanchette,
Manuel Eberl, Alejandro Gómez-Londoño,
Peter Lammich, Christian Sternagel,
Simon Wimmer, Bohua Zhan

Functional Algorithms, Verified!



datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

$$T_{insert}\ a\ t + \Phi(insert\ a\ t) - \Phi(t) \le 3 \cdot \lg(|t| + 3) + 2$$

Quod Erat Demostrandum
QED
Isabelle/HOL

A compendium of functional data structures and algorithms

A compendium of
functional data structures and algorithms

Formally verified (in Isabelle/HOL)

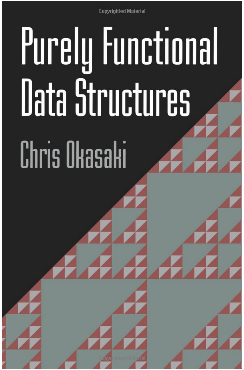
A compendium of
functional data structures and algorithms

Formally verified (in Isabelle/HOL)

Both functional correctness and (amortized) runnig time

Inspired by ...

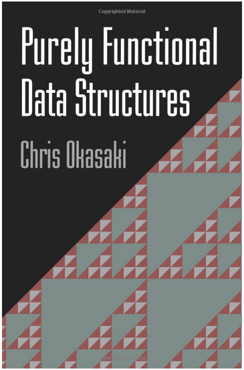
Inspired by ...

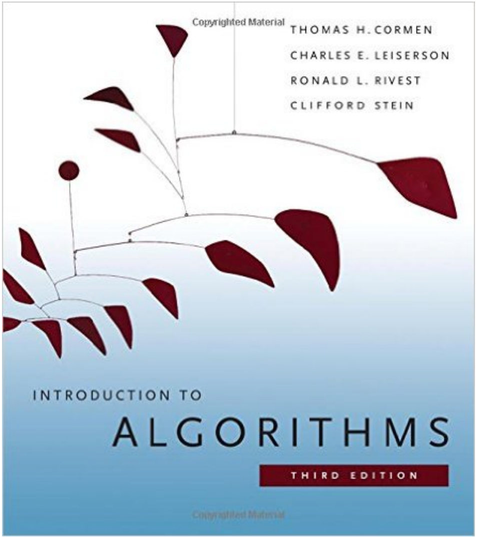

Copyrighted Material

Purely Functional Data Structures

Chris Okasaki

Inspired by ...


Copyrighted Material
Purely Functional Data Structures
Chris Okasaki


Copyrighted Material
THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION
Copyrighted Material

… but with both textual
and (online) machine checked proofs

Topics

Sorting, Selection, Binary Trees, Binary Search Trees, Abstract Data Types, 2-3 Trees, Red-Black Trees, AVL Trees, Just Join, Braun Trees, Tries, Huffman's Algorithm, Priority Queues, Leftist Heaps, Leftist Heaps, Dynamic Programming, Amortized Analysis, Queues, Splay Trees, Skew Heaps, Pairing Heaps

Topics

Sorting, Selection, Binary Trees, Binary Search Trees, Abstract Data Types, 2-3 Trees, Red-Black Trees, AVL Trees, Just Join, Braun Trees, Tries, Huffman's Algorithm, Priority Queues, Leftist Heaps, Leftist Heaps, Dynamic Programming, Amortized Analysis, Queues, Splay Trees, Skew Heaps, Pairing Heaps

Graph Algorithms,
$\alpha\beta$-Search, Quadtrees,
Burrows-Wheeler Transformation
...

❶ Time

❷ Real Time Queue

❸ Real Time Double-Ended Queue

❹ Skew Heap

1. Time
2. Real Time Queue
3. Real Time Double-Ended Queue
4. Skew Heap

The idea

Running time complexity $\approx$ number of function calls

The idea

Running time complexity $\approx$ number of function calls

For each $f :: \tau \to \tau'$

The idea

Running time complexity $\approx$ number of function calls

For each $f :: \tau \to \tau'$
there is a $T_f :: \tau \to nat$

The idea

Running time complexity $\approx$ number of function calls

For each $f :: \tau \to \tau'$
there is a $T_f :: \tau \to nat$ that counts function calls

The idea

Running time complexity $\approx$ number of function calls

For each $f :: \tau \to \tau'$
there is a $T_f :: \tau \to nat$ that counts function calls

Proofs about both f and T_f follow the same principles:
induction, case analyses, equational reasoning, logic, ...

The idea

Running time complexity $\approx$ number of function calls

For each $f :: \tau \to \tau'$
there is a $T_f :: \tau \to nat$ that counts function calls

Proofs about both f and T_f follow the same principles:
induction, case analyses, equational reasoning, logic, ...

Where does T_f come from?

Example

$f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [\,] \Rightarrow ys \mid x\#xs \Rightarrow x\ \#\ f\ xs\ ys$

Example

$$f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow ys \mid x\#xs \Rightarrow x\ \#\ f\ xs\ ys$$

$$T_f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow 1 \mid x\#xs \Rightarrow 1 + 1 + T_f\ xs\ ys$$

Example

$f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow ys \mid x\#xs \Rightarrow x\ \#\ f\ xs\ ys$

$T_f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow 1 \mid x\#xs \Rightarrow 1 + 1 + T_f\ xs\ ys$

Principle: T_f is abstract interpretation of f

Example

$f\ xs\ ys =$ case xs of $[] \Rightarrow ys \mid x\#xs \Rightarrow x\ \#\ f\ xs\ ys$

$T_f\ xs\ ys =$ case xs of $[] \Rightarrow 1 \mid x\#xs \Rightarrow 1 + 1 + T_f\ xs\ ys$

Principle: T_f is abstract interpretation of f

Can be automated

Example

$f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow ys \mid x\#xs \Rightarrow x\ \#\ f\ xs\ ys$

$T_f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow 1 \mid x\#xs \Rightarrow 1 + 1 + T_f\ xs\ ys$

Principle: T_f is abstract interpretation of f

Can be automated (easily for call-by-value)

Example

$f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow ys \mid x\#xs \Rightarrow x\ \#\ f\ xs\ ys$

$T_f\ xs\ ys = \textsf{case}\ xs\ \textsf{of}\ [] \Rightarrow 1 \mid x\#xs \Rightarrow 1 + 1 + T_f\ xs\ ys$

Principle: T_f is abstract interpretation of f

Can be automated (easily for call-by-value)

Additive constants can be reduced to 1

Alternative: Monadic approach

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Basic combinators:

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Basic combinators:

$return\ v = (v, 1)$

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Basic combinators:

$return\ v = (v, 1)$
$bind\ (a, m)\ f = (\text{let } (b, n) = f\ a \text{ in } (b, m + n))$

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Basic combinators:

$return\ v = (v, 1)$
$bind\ (a, m)\ f = (\mathsf{let}\ (b, n) = f\ a\ \mathsf{in}\ (b, m + n)$
Notation: $\{x \leftarrow e_1; e_2\} = bind\ e_1\ (\lambda x.\ e_2)$

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Basic combinators:

$return\ v = (v, 1)$

$bind\ (a, m)\ f = (\text{let } (b, n) = f\ a \text{ in } (b, m + n))$

Notation: $\{x \leftarrow e_1; e_2\} = bind\ e_1\ (\lambda x.\ e_2)$

How to define your algorithms:

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Basic combinators:

$return\ v = (v, 1)$
$bind\ (a, m)\ f = (\text{let}\ (b, n) = f\ a\ \text{in}\ (b, m + n))$
Notation: $\{x \leftarrow e_1; e_2\} = bind\ e_1\ (\lambda x.\ e_2)$

How to define your algorithms:

Define monadic $fm :: \cdots \rightarrow (\tau, nat)$

Alternative: Monadic approach

- Purpose: define f and T_f simultaneously
- Implementation: define function on $(value, time)$ pairs
- Monadic notation hides time

Basic combinators:

$return\ v = (v, 1)$
$bind\ (a, m)\ f = (\textsf{let}\ (b, n) = f\ a\ \textsf{in}\ (b, m + n))$
Notation: $\{x \leftarrow e_1; e_2\} = bind\ e_1\ (\lambda x.\ e_2)$

How to define your algorithms:

Define monadic $fm :: \cdots \to (\tau, nat)$
Then define $f = value \circ fm$ and $T_f = time \circ fm$

Example

$$fm\ []\ ys = return\ []$$

Example

$$fm\ []\ ys = return\ []$$

$$fm\ (x\#xs)\ ys = \{xys \leftarrow fm\ xs\ ys; return(x\ \#\ xys)$$

Example

$$fm\ []\ ys = return\ []$$
$$fm\ (x\#xs)\ ys = \{xys \leftarrow fm\ xs\ ys; return(x\ \#\ xys)$$

$$f\ xs\ ys = val(fm\ xs\ ys) \qquad T_f\ xs\ ys = time(fm\ xs\ ys)$$

Example

$fm\ []\ ys = return\ []$

$fm\ (x\#xs)\ ys = \{xys \leftarrow fm\ xs\ ys; return(x\ \#\ xys)$

$f\ xs\ ys = val(fm\ xs\ ys) \qquad T_f\ xs\ ys = time(fm\ xs\ ys)$

For proving properties of f and T_f:

Derive original recursive definitions of f and T_f
by automatic inductive proof

The rest of the presentation, mostly

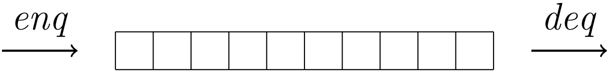
- Focus on persistence and constant time access

The rest of the presentation, mostly

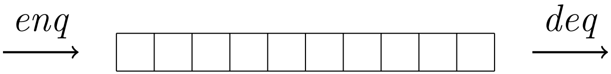
- Focus on persistence and constant time access
- No need to analyze time
  because all functions non-recursive

1. Time
2. Real Time Queue
3. Real Time Double-Ended Queue
4. Skew Heap

Queue

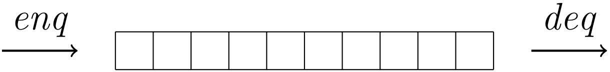


Queue



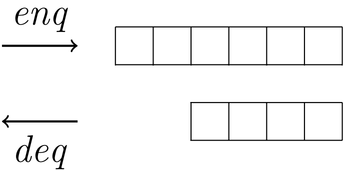
How to implement a functional queue efficiently?

Queue

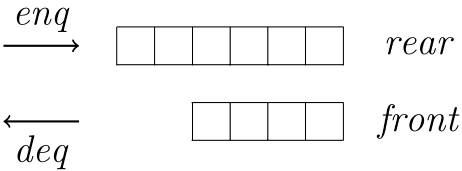


How to implement a functional queue efficiently?
As a list: either enq or deq take linear time

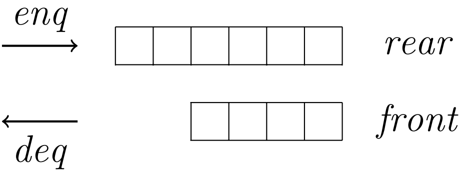
Two stacks



Two stacks

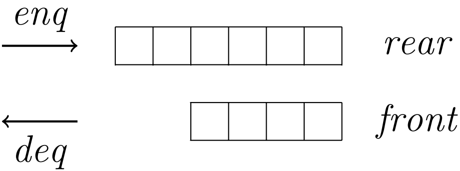


Two stacks



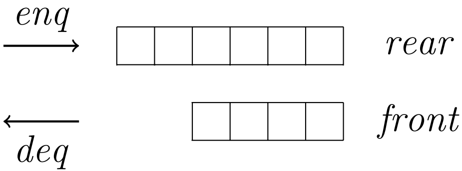
Problem: what if $front$ becomes empty?

Two stacks



Problem: what if $front$ becomes empty?
Need to reverse $rear$ — linear time!

Two stacks



enq $\longrightarrow$ $rear$

$\longleftarrow$ deq $front$

Problem: what if $front$ becomes empty?
Need to reverse $rear$ — linear time!

However: *amortized* running time of each operation (averaged over a sequnce of operations) is constant

Challenge: *Real Time Queue*

All operations have *worst-case* constant running time

One solution: *laziness*

One solution: *laziness*

Implementation with eager/call-by-value evaluation?

Real Time Queue

with call-by-value

- Do not wait for $front = []$

Real Time Queue

with call-by-value

- Do not wait for $front = []$
- Compute new front $front\ @\ rev\ rear$
  early and incrementally

Real Time Queue

with call-by-value

- Do not wait for $front = []$
- Compute new front $front \mathbin{@} rev\ rear$
  early and incrementally
- Incremental reversal by pair of stacks:
  $([a, b, c], [])$

Real Time Queue

with call-by-value

- Do not wait for $front = []$
- Compute new front $front\ @\ rev\ rear$ early and incrementally
- Incremental reversal by pair of stacks: $([a, b, c], []) \to ([b, c], [a])$

Real Time Queue

with call-by-value

- Do not wait for $\mathit{front} = []$
- Compute new front $\mathit{front} \mathbin{@} \mathit{rev}\ \mathit{rear}$
  early and incrementally
- Incremental reversal by pair of stacks:
  $([a, b, c], []) \to ([b, c], [a]) \to ([c], [b, a])$

Real Time Queue

with call-by-value

- Do not wait for $\mathit{front} = []$
- Compute new front $\mathit{front} \mathbin{@} \mathit{rev}\ \mathit{rear}$
  early and incrementally
- Incremental reversal by pair of stacks:
  $([a, b, c], []) \to ([b, c], [a]) \to ([c], [b, a]) \to ([], [c, b, a])$

Real Time Queue

with call-by-value

- Do not wait for $front = []$
- Compute new front $front \mathbin{@} rev\ rear$
  early and incrementally
- Incremental reversal by pair of stacks:
  $([a, b, c], []) \to ([b, c], [a]) \to ([c], [b, a]) \to ([], [c, b, a])$
- Using a ‘copy’ of $front$ and $rear$

Real Time Queue

with call-by-value

- Do not wait for $front = []$
- Compute new front $front \mathbin{@} rev\ rear$
  early and incrementally
- Incremental reversal by pair of stacks:
  $([a, b, c], []) \rightarrow ([b, c], [a]) \rightarrow ([c], [b, a]) \rightarrow ([], [c, b, a])$
- Using a ‘copy’ of $front$ and $rear$
  “shadow queue”

Real Time Queue

with call-by-value

- Do not wait for $front = []$
- Compute new front $front \mathbin{@} rev\ rear$ early and incrementally
- Incremental reversal by pair of stacks: $([a, b, c], []) \rightarrow ([b, c], [a]) \rightarrow ([c], [b, a]) \rightarrow ([], [c, b, a])$
- Using a ‘copy’ of $front$ and $rear$ “shadow queue”
- In parallel with enq and deq calls

Reversal strategy

Aim: $(r, f) \rightarrow^* ([], f \ @ \ rev \ r)$

Reversal strategy

Aim: $(r, f) \to^* ([], f \mathbin{@} \mathit{rev}\ r)$

In two phases:

Reversal strategy

Aim: $(r, f) \to^* ([], f \mathbin{@} rev\ r)$

In two phases:

1. Reverse $\quad r = [b_1, \ldots, b_m] \to^m [b_m, \ldots, b_1] =: r'$

Reversal strategy

Aim: $(r, f) \rightarrow^{*} ([], f \mathbin{@} rev\ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \rightarrow^{m} [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \rightarrow^{n} [a_n, \ldots, a_1] =: f'$

Reversal strategy

Aim: $(r, f) \rightarrow^* ([], f \mathbin{@} \mathit{rev}\ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \rightarrow^m [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \rightarrow^n [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \rightarrow^n ([], \mathit{rev}\ f' \mathbin{@} r')$

Reversal strategy

Aim: $(r, f) \to^* ([], f \mathbin{@} rev\ r)$

In two phases:

1. Reverse $\quad r = [b_1, \ldots, b_m] \to^m [b_m, \ldots, b_1] =: r'$
   and $\quad f = [a_1, \ldots, a_n] \to^n [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \to^n \ ([], rev\ f' \mathbin{@} r') = ([], f \mathbin{@} rev\ r)$

Reversal strategy

Aim: $(r, f) \rightarrow^* ([], f \mathbin{@} rev\ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \rightarrow^m [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \rightarrow^n [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \rightarrow^n \ ([], rev\ f' \mathbin{@} r') = ([], f \mathbin{@} rev\ r)$

When to start?

Reversal strategy

Aim: $(r, f) \rightarrow^{*} ([], f \text{ @ } rev\ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \rightarrow^{m} [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \rightarrow^{n} [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \rightarrow^{n} ([], rev\ f' \text{ @ } r') = ([], f \text{ @ } rev\ r)$

When to start? When $m = n + 1$!

Reversal strategy

Aim: $(r, f) \to^{*} ([], f \mathbin{@} \mathit{rev}\ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \to^{m} [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \to^{n} [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \to^{n} ([], \mathit{rev}\ f' \mathbin{@} r') = ([], f \mathbin{@} \mathit{rev}\ r)$

When to start? When $m = n + 1$!

- Requires $n + 1 + n$ steps

Reversal strategy

Aim: $(r, f) \rightarrow^* ([], f \ @ \ rev \ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \rightarrow^m [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \rightarrow^n [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \rightarrow^n \ ([], rev \ f' \ @ \ r') = ([], f \ @ \ rev \ r)$

When to start? When $m = n + 1$!

- Requires $n + 1 + n$ steps
- Need to finish before original front becomes empty

Reversal strategy

Aim: $(r, f) \rightarrow^* ([], f \mathbin{@} rev\ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \rightarrow^m [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \rightarrow^n [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \rightarrow^n ([], rev\ f' \mathbin{@} r') = ([], f \mathbin{@} rev\ r)$

When to start? When $m = n + 1$!

- Requires $n + 1 + n$ steps
- Need to finish before original front becomes empty
- Need to perform 2 steps per enq/deq

Reversal strategy

Aim: $(r, f) \rightarrow^* ([], f \mathbin{@} rev\ r)$

In two phases:

1. Reverse $r = [b_1, \ldots, b_m] \rightarrow^m [b_m, \ldots, b_1] =: r'$
   and $f = [a_1, \ldots, a_n] \rightarrow^n [a_n, \ldots, a_1] =: f'$
2. Reverse f' onto r':
   $(f', r') \rightarrow^n ([], rev\ f' \mathbin{@} r') = ([], f \mathbin{@} rev\ r)$

When to start? When $m = n + 1$!

- Requires $n + 1 + n$ steps
- Need to finish before original front becomes empty
- Need to perform 2 steps per enq/deq
- $+1$ initial step

Complication

Complication

deq from the original $front$

Complication

deq from the original $front$

Cannot easily remove them from the shadow queue

Complication

deq from the original $front$

Cannot easily remove them from the shadow queue

Solution:

Complication

deq from the original $front$

Cannot easily remove them from the shadow queue

Solution:

- Remember how many elements have been removed

Complication

deq from the original $front$

Cannot easily remove them from the shadow queue

Solution:

- Remember how many elements have been removed
- Better: how many elements are still valid

Complication

deq from the original $front$

Cannot easily remove them from the shadow queue

Solution:

- Remember how many elements have been removed
- Better: how many elements are still valid

enq into new (initially empty) $rear$.

Complication

deq from the original $front$

Cannot easily remove them from the shadow queue

Solution:

- Remember how many elements have been removed
- Better: how many elements are still valid

enq into new (initially empty) $rear$.
Reversal fast enough to ensure $|new\ rear| \leq |new\ front|$ at the end

Implementation

The shadow queue

datatype $'a\ status =$

$\quad Idle\ |$

$\quad Rev\ (nat)\ ('a\ list)\ ('a\ list)\ ('a\ list)\ ('a\ list)\ |$

$\quad App\ (nat)\ ('a\ list)\ ('a\ list)\ |$

$\quad Done\ ('a\ list)$

Shadow step

$$exec :: {}'a\ status \Rightarrow {}'a\ status$$

$$exec\ Idle = Idle$$

$$\begin{aligned}&exec\ (Rev\ ok\ (x \# f)\ f'\ (y \# r)\ r')\\ &= Rev\ (ok + 1)\ f\ (x \# f')\ r\ (y \# r')\end{aligned}$$

$$exec\ (Rev\ ok\ []\ f'\ [y]\ r') = App\ ok\ f'\ (y \# r')$$

$$exec\ (App\ (ok + 1)\ (x \# f')\ r') = App\ ok\ f'\ (x \# r')$$

$$exec\ (App\ 0\ f'\ r') = Done\ r'$$

$$exec\ (Done\ v) = Done\ v$$

Dequeue from shadow queue

$$invalidate :: {}'a\ status \Rightarrow {}'a\ status$$

$$invalidate\ Idle = Idle$$

$$invalidate\ (Rev\ ok\ f\ f'\ r\ r') = Rev\ (ok - 1)\ f\ f'\ r\ r'$$

$$invalidate\ (App\ (ok + 1)\ f'\ r') = App\ ok\ f'\ r'$$

$$invalidate\ (App\ 0\ f'\ (x \# r')) = Done\ r'$$

$$invalidate\ (Done\ v) = Done\ v$$

The whole queue

$$\textbf{record}\ 'a\ queue = \begin{array}{ll} front & :: 'a\ list \\ lenf & :: nat \\ rear & :: 'a\ list \\ lenr & :: nat \\ status & :: 'a\ status \end{array}$$

enq and deq

$$enq\ x\ q =$$
$$check\ (q(\!|rear := x\ \#\ rear\ q,\ lenr := lenr\ q + 1|\!))$$

$$deq\ q =$$
$$check$$
$$(q(\!|lenf := lenf\ q - 1,\ front := tl\ (front\ q),$$
$$status := invalidate\ (status\ q)|\!))$$

$$\begin{array}{l}
check\ q = \\
(\mathsf{if}\ lenr\ q \le lenf\ q\ \mathsf{then}\ exec2\ q \\
\quad \mathsf{else\ let}\ newstate = \\
\qquad\qquad\qquad Rev\ 0\ (front\ q)\ []\ (rear\ q)\ [] \\
\qquad\quad \mathsf{in}\ exec2 \\
\qquad\qquad\quad (q(\!|lenf := lenf\ q + lenr\ q, \\
\qquad\qquad\qquad\quad status := newstate, \\
\qquad\qquad\qquad\quad rear := [],\ lenr := 0|\!)))
\end{array}$$

$$\begin{array}{l}
exec2\ q = \ (case\ exec\ (exec\ q)\ of \\
\qquad\qquad\qquad\quad Done\ fr \Rightarrow q(\!|status = Idle,\ front = fr|\!)\ | \\
\qquad\qquad\qquad\quad newstatus \Rightarrow q(\!|status = newstatus|\!))
\end{array}$$

Model oriented specification

Model oriented specification

Model data structure by existing mathematical types

Model oriented specification

Model data structure by existing mathematical types

Example: $queue$ by $list$

Model oriented specification

Model data structure by existing mathematical types

Example: $queue$ by $list$

Assume abstraction function (α) from $queue$ to $list$

Model oriented specification

Model data structure by existing mathematical types

Example: $queue$ by $list$

Assume abstraction function (α) from $queue$ to $list$

Specify each $queue$ function by a corresponding $list$ function

Model oriented specification

Model data structure by existing mathematical types

Example: $queue$ by $list$

Assume abstraction function (α) from $queue$ to $list$

Specify each $queue$ function by a corresponding $list$ function

Formally: require that α is a homomorphism

Model oriented specification

Model data structure by existing mathematical types

Example: $queue$ by $list$

Assume abstraction function (α) from $queue$ to $list$

Specify each $queue$ function by a corresponding $list$ function

Formally: require that α is a homomorphism

Correctness proof of an implementation:

Model oriented specification

Model data structure by existing mathematical types

Example: $queue$ by $list$

Assume abstraction function (α) from $queue$ to $list$

Specify each $queue$ function by a corresponding $list$ function

Formally: require that α is a homomorphism

Correctness proof of an implementation:
define α and prove Spec

Queue specification

interface $empty :: {'a}\ queue$

$enq :: {'a} \Rightarrow {'a}\ queue \Rightarrow {'a}\ queue$

$deq :: {'a}\ queue \Rightarrow {'a}\ queue$

$first :: {'a}\ queue \Rightarrow {'a}$

Queue specification

interface $empty :: {'a}\ queue$

$enq :: {'a} \Rightarrow {'a}\ queue \Rightarrow {'a}\ queue$

$deq :: {'a}\ queue \Rightarrow {'a}\ queue$

$first :: {'a}\ queue \Rightarrow {'a}$

abstraction $list :: {'a}\ queue \Rightarrow {'a}\ list$

Queue specification

interface $empty :: 'a\ queue$

$enq :: 'a \Rightarrow 'a\ queue \Rightarrow 'a\ queue$

$deq :: 'a\ queue \Rightarrow 'a\ queue$

$first :: 'a\ queue \Rightarrow 'a$

abstraction $list :: 'a\ queue \Rightarrow 'a\ list$

invariant $invar :: 'a\ queue \Rightarrow bool$

Queue specification

interface $empty :: {}'a\ queue$

$enq :: {}'a \Rightarrow {}'a\ queue \Rightarrow {}'a\ queue$

$deq :: {}'a\ queue \Rightarrow {}'a\ queue$

$first :: {}'a\ queue \Rightarrow {}'a$

abstraction $list :: {}'a\ queue \Rightarrow {}'a\ list$

invariant $invar :: {}'a\ queue \Rightarrow bool$

specification

$$invar\ q \Longrightarrow list\ (enq\ x\ q) = list\ q\ @\ [x]$$

Queue specification

interface $\mathit{empty} :: {}'a\ \mathit{queue}$

$\mathit{enq} :: {}'a \Rightarrow {}'a\ \mathit{queue} \Rightarrow {}'a\ \mathit{queue}$

$\mathit{deq} :: {}'a\ \mathit{queue} \Rightarrow {}'a\ \mathit{queue}$

$\mathit{first} :: {}'a\ \mathit{queue} \Rightarrow {}'a$

abstraction $\mathit{list} :: {}'a\ \mathit{queue} \Rightarrow {}'a\ \mathit{list}$

invariant $\mathit{invar} :: {}'a\ \mathit{queue} \Rightarrow \mathit{bool}$

specification

$\mathit{invar}\ q \Longrightarrow \mathit{list}\ (\mathit{enq}\ x\ q) = \mathit{list}\ q\ @\ [x]$

$\mathit{invar}\ q \Longrightarrow \mathit{list}\ (\mathit{deq}\ q) = \mathit{tail}\ (\mathit{list}\ q)$

$\mathit{invar}\ q \wedge \mathit{list}\ q \neq [] \Longrightarrow \mathit{first}\ q = \mathit{head}\ (\mathit{list}\ q)$

$\vdots$

Correctness

Correctness

The proof is

Correctness

The proof is

- easy because all functions are non-recursive

Correctness

The proof is

- easy because all functions are non-recursive
  ($\Longrightarrow$ constant running time!)

Correctness

The proof is

- easy because all functions are non-recursive ($\Longrightarrow$ constant running time!)
- tricky because of invariant and abstraction function

Correctness

The proof is

- easy because all functions are non-recursive
  ($\Longrightarrow$ constant running time!)
- tricky because of invariant and abstraction function
  700 lines of Isabelle (by Alejandro Gómez-Londoño)

$status$ invariant

$$\begin{aligned}
&inv\_st\ (Rev\ ok\ f\ f'\ r\ r') = \\
&(|f| + 1 = |r| \wedge |f'| = |r'| \wedge ok \leq |f'|) \\
&inv\_st\ (App\ ok\ f'\ r') = (ok \leq |f'| \wedge |f'| < |r'|) \\
&inv\_st\ Idle = True \\
&inv\_st\ (Done\ \_) = True
\end{aligned}$$

Queue invariant

$$
\begin{array}{l}
\mathit{invar}\ q = \\
(\mathit{lenf}\ q = |\mathit{front\_list}\ q| \wedge \\
\ \mathit{lenr}\ q = |\mathit{rev}\ (\mathit{rear}\ q)| \wedge \\
\ \mathit{lenr}\ q \leq \mathit{lenf}\ q \wedge \\
\ (\textsf{case}\ \mathit{status}\ q\ \textsf{of} \\
\qquad \mathit{Rev}\ \mathit{ok}\ f\ f'\ r\ r' \Rightarrow \\
\qquad\quad 2 * \mathit{lenr}\ q \leq |f'| \wedge \\
\qquad\quad \mathit{ok} \neq 0 \wedge 2 * |f| + \mathit{ok} + 2 \leq 2 * |\mathit{front}\ q| \\
\quad |\ \mathit{App}\ \mathit{ok}\ f\ r \Rightarrow \\
\qquad\quad 2 * \mathit{lenr}\ q \leq |r| \wedge \mathit{ok} + 1 \leq 2 * |\mathit{front}\ q| \\
\quad |\ \_ \Rightarrow \mathit{True}) \wedge \\
\ (\exists\, \mathit{rest}.\ \mathit{front\_list}\ q = \mathit{front}\ q\ @\ \mathit{rest}) \wedge \\
\ (\nexists\, \mathit{fr}.\ \mathit{status}\ q = \mathit{Done}\ \mathit{fr}) \wedge \mathit{inv\_st}\ (\mathit{status}\ q))
\end{array}
$$

Abstraction function

$$
\begin{array}{l}
\mathit{list}\ q = \mathit{front\_list}\ q\ @\ \mathit{rear\_list}\ q \\
\mathit{front\_list}\ q = \\
(\textsf{case}\ \mathit{status}\ q\ \textsf{of} \\
\quad \mathit{Idle} \Rightarrow \mathit{front}\ q \\
\ \mid \mathit{Rev}\ \mathit{ok}\ f\ f'\ r\ r' \Rightarrow \mathit{rev}\ (\mathit{take}\ \mathit{ok}\ f')\ @\ f\ @\ \mathit{rev}\ r\ @\ r' \\
\ \mid \mathit{App}\ \mathit{ok}\ f'\ x \Rightarrow \mathit{rev}\ (\mathit{take}\ \mathit{ok}\ f')\ @\ x \\
\ \mid \mathit{Done}\ f \Rightarrow f)
\end{array}
$$

The inventors

Robert Hood and Robert Melville.
Real-Time Queue Operation in Pure LISP.
Information Processing Letters, 1981.

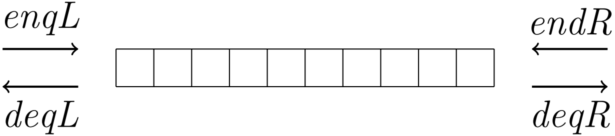
1. Time

2. Real Time Queue

3. Real Time Double-Ended Queue

4. Skew Heap

Double-Ended Queue ("Deque")



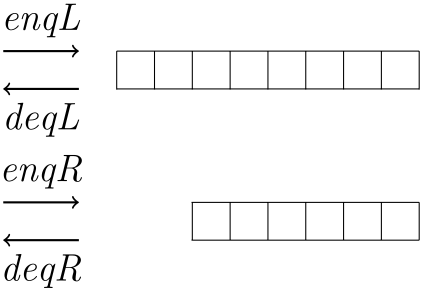
$enqL$

$deqL$

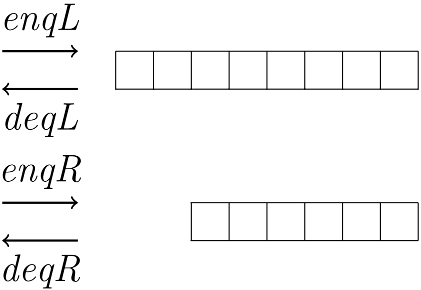
$endR$

$deqR$

Two stacks



Two stacks



$\xrightarrow{enqL}$

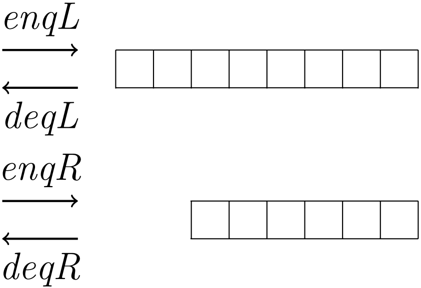
$\underset{deqL}{\xleftarrow{\hspace{2em}}}$

$\xrightarrow{enqR}$

$\underset{deqR}{\xleftarrow{\hspace{2em}}}$

Amortized constant time enq/deq:

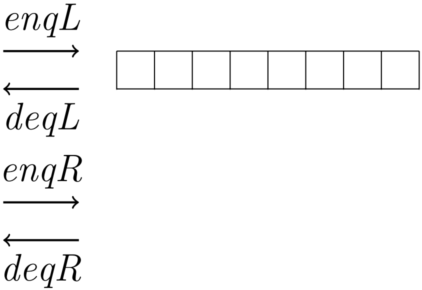
Two stacks



Amortized constant time enq/deq:

If one stack becomes empty,

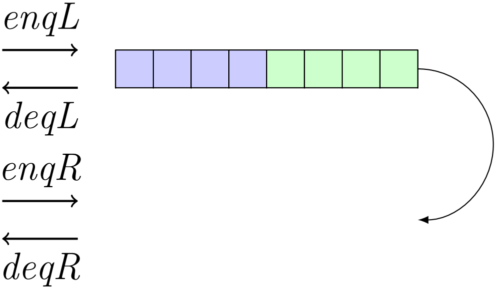
Two stacks



Amortized constant time enq/deq:

If one stack becomes empty,

Two stacks



Amortized constant time enq/deq:

If one stack becomes empty,
reverse *the botttom half* of the other one

Real Time Deque

One solution: *laziness*

Real Time Deque

One solution: *laziness*

Implementation with eager/call-by-value evaluation?

Real Time Deque

Call-by-value

- Do not wait for $[\,]$

Real Time Deque

Call-by-value

- Do not wait for []
- When the stacks become “too unbalanced”:

Real Time Deque

Call-by-value

- Do not wait for $[]$
- When the stacks become “too unbalanced”:
  Move part of bigger stack to smaller stack

Real Time Deque

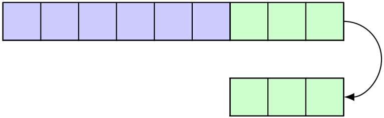
Call-by-value

- Do not wait for []
- When the stacks become “too unbalanced”:
  Move part of bigger stack to smaller stack
- Aim for equal size of both stacks after reversal:

Real Time Deque

Call-by-value

- Do not wait for []
- When the stacks become “too unbalanced”:
  Move part of bigger stack to smaller stack
- Aim for equal size of both stacks after reversal:



Main invariant

S is smaller stack, B bigger stack, $m = |S|$, $n = |B|$.

Main invariant

S is smaller stack, B bigger stack, $m = |S|$, $n = |B|$.

$$3m \geq n$$

Main invariant

S is smaller stack, B bigger stack, $m = |S|$, $n = |B|$.

$$3m \geq n$$

When is $3m \geq n$ destroyed by enq or deq?

Main invariant

S is smaller stack, B bigger stack, $m = |S|$, $n = |B|$.

$$3m \geq n$$

When is $3m \geq n$ destroyed by enq or deq?
When $3m \approx n$

Main invariant

S is smaller stack, B bigger stack, $m = |S|$, $n = |B|$.

$$3m \geq n$$

When is $3m \geq n$ destroyed by enq or deq?
When $3m \approx n$ ($\approx$ means we ignore the fine details)

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.

Aim: $B_{12} @ B_3, S \ \rightsquigarrow\ B_{12}, S @ \overleftarrow{B_3}$

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.

Aim: $B_{12} @ B_3, S \ \rightsquigarrow \ B_{12}, S @ \overleftarrow{B_3}$

$B_{12} @ B_3$

S

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.

Aim: $B_{12} @ B_3, S \ \rightsquigarrow\ B_{12}, S @ \overleftarrow{B}_3$

$$B_{12} @ B_3 \quad \rightarrow^{2m} \quad \overleftarrow{B}_{12}$$

$$B_3$$

$$S$$

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.
Aim: $B_{12} @ B_3, S \rightsquigarrow B_{12}, S @ \overleftarrow{B_3}$

$$\begin{array}{lll}
B_{12} @ B_3 & \rightarrow^{2m} & \overleftarrow{B_{12}} \\
 & & B_3 \\
S & \rightarrow^{m} & \overleftarrow{S}
\end{array}$$

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.

Aim: $B_{12} @ B_3, S \ \rightsquigarrow\ B_{12}, S @ \overleftarrow{B}_3$

$$\begin{array}{llll}
B_{12} @ B_3 & \rightarrow^{2m} & \overleftarrow{B}_{12} & \rightarrow^{2m} & B_{12} \\
 & & B_3 & & \\
S & \rightarrow^{m} & \overleftarrow{S} & &
\end{array}$$

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.

Aim: $B_{12} @ B_3, S \rightsquigarrow B_{12}, S @ \overleftarrow{B_3}$

$$\begin{tikzcd}
B_{12} @ B_3 \arrow[r, "2m"] & \overleftarrow{B_{12}} \arrow[rr, "2m"] & & B_{12} \\
& B_3 \arrow[r, "m"] & \overleftarrow{B_3} & \\
S \arrow[r, "m"] & \overleftarrow{S} & &
\end{tikzcd}$$

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.
Aim: $B_{12} @ B_3, S \ \rightsquigarrow\ B_{12}, S @ \overleftarrow{B_3}$

$$
\begin{array}{lllllll}
B_{12} @ B_3 & \rightarrow^{2m} & \overleftarrow{B_{12}} & & & \rightarrow^{2m} & B_{12} \\
 & & B_3 & \rightarrow^{m} & \overleftarrow{B_3} & \rightarrow^{m} & S @ \overleftarrow{B_3} \\
S & \rightarrow^{m} & \overleftarrow{S} & & & &
\end{array}
$$

Rebalancing strategy

Start: $B = B_{12} @ B_3$ where $|B_{12}| = 2m$ and $|B_3| = m$.

Aim: $B_{12} @ B_3, S \;\rightsquigarrow\; B_{12}, S @ \overleftarrow{B_3}$

$$\begin{array}{lllllll}
B_{12} @ B_3 & \rightarrow^{2m} & \overleftarrow{B_{12}} & & \rightarrow^{2m} & & B_{12} \\
 & & B_3 & \rightarrow^{m} & \overleftarrow{B_3} & \rightarrow^{m} & S @ \overleftarrow{B_3} \\
S & \rightarrow^{m} & \overleftarrow{S} & & & &
\end{array}$$

Requires $4m$ micro-steps, 4 per enq/deq step

Two deques

Rebalancing happens on shadow deque

Two deques

Rebalancing	happens on	shadow deque
enq/deq	happens on	current deque

Another complication

At the end of rebalancing:

Another complication

At the end of rebalancing:

Need to combine results of rebalancing
and newly enq'ed elements,

Another complication

At the end of rebalancing:

Need to combine results of rebalancing
and newly enq'ed elements, without using $@$!

Another complication

At the end of rebalancing:

Need to combine results of rebalancing
and newly enq'ed elements, without using @ !

$\Longrightarrow$ New stacks pair of lists

Another complication

At the end of rebalancing:

Need to combine results of rebalancing
and newly enq'ed elements, without using @ !

$\Longrightarrow$ New stacks pair of lists

(No need for triples etc)

Another complication

At the end of rebalancing:

Need to combine results of rebalancing
and newly enq'ed elements, without using @ !

$\Longrightarrow$ New stacks pair of lists
(No need for triples etc)

(Why not a problem with real time queue?)

Another detail

Deques of size ≤ 3 are represented as normal lists

No problems

No problems

Rebalancing needs m steps
and yields two stacks of size $2m$ each.

No problems

Rebalancing needs m steps
and yields two stacks of size $2m$ each.

Two extremes during rebalancing:

- $m \times enq$ at one end:

No problems

Rebalancing needs m steps
and yields two stacks of size $2m$ each.

Two extremes during rebalancing:

- $m \times enq$ at one end:
  in the end the stacks have size $2m$ and $3m$ ✓

No problems

Rebalancing needs m steps
and yields two stacks of size $2m$ each.

Two extremes during rebalancing:

- $m \times enq$ at one end:
  in the end the stacks have size $2m$ and $3m$ ✓
- $m \times deq$ of S:

No problems

Rebalancing needs m steps
and yields two stacks of size $2m$ each.

Two extremes during rebalancing:

- $m \times enq$ at one end:
  in the end the stacks have size $2m$ and $3m$ ✓
- $m \times deq$ of S: S has m elements ✓

No problems

Rebalancing needs m steps
and yields two stacks of size $2m$ each.

Two extremes during rebalancing:

- $m \times enq$ at one end:
  in the end the stacks have size $2m$ and $3m$ ✓
- $m \times deq$ of S: S has m elements ✓
  in the end the stacks have size m and $2m$ ✓

The full story

500 lines of code

3900 lines of invariants, abstraction functions and proofs

(by Balazs Toth)

The full story

500 lines of code

3900 lines of invariants, abstraction functions and proofs

(by Balazs Toth)

Based on

Chuang and Goldberg.
Real-time deques, multihead turing machines, and purely functional programming. In *FPCA* 1993.

The full story

500 lines of code

3900 lines of invariants, abstraction functions and proofs

(by Balazs Toth)

Based on

Chuang and Goldberg.
Real-time deques, multihead turing machines, and purely functional programming. In *FPCA* 1993.

Already sketched in Hood's PhD thesis 1982

1. Time

2. Real Time Queue

3. Real Time Double-Ended Queue

4. **Skew Heap**

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

Implementation type

Ordinary binary trees

Invariant: $heap$

$merge$

$$merge\ \langle\rangle\ t = t$$
$$merge\ h\ \langle\rangle = h$$

$merge$

$merge\ \langle\rangle\ t = t$
$merge\ h\ \langle\rangle = h$

Swap subtrees when descending:

$merge$

$merge\ \langle\rangle\ t = t$
$merge\ h\ \langle\rangle = h$

Swap subtrees when descending:

$merge\ (\langle l_1,\ a_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ r_2\rangle =: t_2) =$
$(\textsf{if}\ a_1 \le a_2\ \textsf{then}\ \langle merge\ t_2\ r_1,\ a_1,\ l_1\rangle$
$\ \textsf{else}\ \langle merge\ t_1\ r_2,\ a_2,\ l_2\rangle)$

Functional correctness proofs

Straightforward

Logarithmic amortized complexity

Theorem

$$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$$
$$\leq 3 * \log_2\ (|t_1|_1 + |t_2|_1) + 1$$

Towards the proof

Towards the proof

Right heavy:

$rh\ l\ r = (\textsf{if}\ |l| < |r|\ \textsf{then}\ 1\ \textsf{else}\ 0)$

Towards the proof

Right heavy:
$rh\ l\ r = (\textsf{if } |l| < |r| \textsf{ then } 1 \textsf{ else } 0)$

Number of right heavy nodes on left spine:

$lrh\ \langle\rangle = 0$
$lrh\ \langle l,\ \_,\ r\rangle = rh\ l\ r + lrh\ l$

Towards the proof

Right heavy:
$rh\ l\ r = (\textsf{if}\ |l| < |r|\ \textsf{then}\ 1\ \textsf{else}\ 0)$

Number of right heavy nodes on left spine:

$lrh\ \langle\rangle = 0$
$lrh\ \langle l,\ \_,\ r\rangle = rh\ l\ r + lrh\ l$

Lemma

$2^{lrh\ t} \leq |t| + 1$

Towards the proof

Right heavy:
$rh\ l\ r = (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)$

Number of right heavy nodes on left spine:

$lrh\ \langle\rangle = 0$
$lrh\ \langle l,\ \_,\ r\rangle = rh\ l\ r + lrh\ l$

Lemma

$2^{lrh\ t} \le |t| + 1$

Corollary

$lrh\ t \le \log_2\ |t|_1$

Towards the proof

Right heavy:

$rh\ l\ r = (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)$

Number of not right heavy nodes on right spine:

$rlh\ \langle\rangle = 0$

$rlh\ \langle l,\ \_,\ r\rangle = 1 - rh\ l\ r + rlh\ r$

Lemma

$2^{rlh\ t} \leq |t| + 1$

Corollary

$rlh\ t \leq \log_2\ |t|_1$

Potential

The potential is the number of right heavy nodes:

Potential

The potential is the number of right heavy nodes:

$$\Phi\ \langle\rangle = 0$$
$$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + rh\ l\ r$$

Potential

The potential is the number of right heavy nodes:

$\Phi\ \langle\rangle = 0$

$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + rh\ l\ r$

$merge$ descends on the right

$\Longrightarrow$ right heavy nodes are bad

Potential

The potential is the number of right heavy nodes:

$\Phi\ \langle\rangle = 0$
$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + rh\ l\ r$

$merge$ descends on the right
$\Longrightarrow$ right heavy nodes are bad

Lemma

$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$
$\leq lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1$

Potential

The potential is the number of right heavy nodes:

$\Phi\ \langle\rangle = 0$
$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + rh\ l\ r$

$merge$ descends on the right
$\Longrightarrow$ right heavy nodes are bad

Lemma

$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$
$\le lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1$

```
by(induction t1 t2 rule: merge.induct)(auto)
```

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1\rangle$, $t_2 = \langle l_2,\ a_2,\ r_2\rangle$.

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1 \rangle$, $t_2 = \langle l_2,\ a_2,\ r_2 \rangle$.
Case $a_1 \leq a_2$.

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1 \rangle$, $t_2 = \langle l_2,\ a_2,\ r_2 \rangle$.
Case $a_1 \leq a_2$. Let $m = merge\ t_2\ r_1$

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1 \rangle$, $t_2 = \langle l_2,\ a_2,\ r_2 \rangle$.
Case $a_1 \leq a_2$. Let $m = merge\ t_2\ r_1$

$$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$$

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1\rangle$, $t_2 = \langle l_2,\ a_2,\ r_2\rangle$.
Case $a_1 \leq a_2$. Let $m = merge\ t_2\ r_1$

$$
\begin{aligned}
&T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2\\
&= T\_merge\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1\\
&\quad - \Phi\ t_1 - \Phi\ t_2
\end{aligned}
$$

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1\rangle$, $t_2 = \langle l_2,\ a_2,\ r_2\rangle$.
Case $a_1 \le a_2$. Let $m = merge\ t_2\ r_1$

$$\begin{aligned}
&T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2\\
= {}&T\_merge\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1\\
&\quad - \Phi\ t_1 - \Phi\ t_2\\
= {}&T\_merge\ t_2\ r_1 + 1 + \Phi\ m + rh\ m\ l_1\\
&\quad - \Phi\ r_1 - rh\ l_1\ r_1 - \Phi\ t_2
\end{aligned}$$

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1 \rangle$, $t_2 = \langle l_2,\ a_2,\ r_2 \rangle$.
Case $a_1 \leq a_2$. Let $m = merge\ t_2\ r_1$

$$\begin{aligned}
& T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
= {} & T\_merge\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1 \\
& - \Phi\ t_1 - \Phi\ t_2 \\
= {} & T\_merge\ t_2\ r_1 + 1 + \Phi\ m + rh\ m\ l_1 \\
& - \Phi\ r_1 - rh\ l_1\ r_1 - \Phi\ t_2 \\
\leq {} & lrh\ m + rlh\ t_2 + rlh\ r_1 + rh\ m\ l_1 + 2 - rh\ l_1\ r_1
\end{aligned}$$

by IH

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1\rangle$, $t_2 = \langle l_2,\ a_2,\ r_2\rangle$.
Case $a_1 \le a_2$. Let $m = merge\ t_2\ r_1$

$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$
$= T\_merge\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1$
$\quad - \Phi\ t_1 - \Phi\ t_2$
$= T\_merge\ t_2\ r_1 + 1 + \Phi\ m + rh\ m\ l_1$
$\quad - \Phi\ r_1 - rh\ l_1\ r_1 - \Phi\ t_2$
$\le lrh\ m + rlh\ t_2 + rlh\ r_1 + rh\ m\ l_1 + 2 - rh\ l_1\ r_1$
by IH
$= lrh\ m + rlh\ t_2 + rlh\ t_1 + rh\ m\ l_1 + 1$

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1\rangle$, $t_2 = \langle l_2,\ a_2,\ r_2\rangle$.
Case $a_1 \le a_2$. Let $m = merge\ t_2\ r_1$

$$
\begin{aligned}
& T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
= {} & T\_merge\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1 \\
& - \Phi\ t_1 - \Phi\ t_2 \\
= {} & T\_merge\ t_2\ r_1 + 1 + \Phi\ m + rh\ m\ l_1 \\
& - \Phi\ r_1 - rh\ l_1\ r_1 - \Phi\ t_2 \\
\le {} & lrh\ m + rlh\ t_2 + rlh\ r_1 + rh\ m\ l_1 + 2 - rh\ l_1\ r_1 \\
& \text{by IH} \\
= {} & lrh\ m + rlh\ t_2 + rlh\ t_1 + rh\ m\ l_1 + 1 \\
= {} & lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1
\end{aligned}
$$

Main proof

$$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$$

Main proof

$$
\begin{aligned}
&T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
&\le lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1
\end{aligned}
$$

Main proof

$$
\begin{aligned}
&T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2\\
&\le lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1\\
&\le \log_2\ |merge\ t_1\ t_2|_1 + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1
\end{aligned}
$$

Main proof

$$
\begin{aligned}
& T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
& \leq lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 \\
& \leq \log_2\ |merge\ t_1\ t_2|_1 + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& = \log_2\ (|t_1|_1 + |t_2|_1 - 1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1
\end{aligned}
$$

Main proof

$$
\begin{aligned}
& T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
& \leq lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 \\
& \leq \log_2\ |merge\ t_1\ t_2|_1 + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& = \log_2\ (|t_1|_1 + |t_2|_1 - 1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& \leq \log_2\ (|t_1|_1 + |t_2|_1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1
\end{aligned}
$$

Main proof

$$
\begin{aligned}
& T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
& \le lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 \\
& \le \log_2\ |merge\ t_1\ t_2|_1 + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& = \log_2\ (|t_1|_1 + |t_2|_1 - 1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& \le \log_2\ (|t_1|_1 + |t_2|_1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& \le \log_2\ (|t_1|_1 + |t_2|_1) + 2 * \log_2\ (|t_1|_1 + |t_2|_1) + 1
\end{aligned}
$$

because $\log_2\ x + \log_2\ y \le 2 * \log_2\ (x + y)$ if $x,y > 0$

Main proof

$$
\begin{aligned}
& T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
\le\ & lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 \\
\le\ & \log_2\ |merge\ t_1\ t_2|_1 + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
=\ & \log_2\ (|t_1|_1 + |t_2|_1 - 1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
\le\ & \log_2\ (|t_1|_1 + |t_2|_1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
\le\ & \log_2\ (|t_1|_1 + |t_2|_1) + 2 * \log_2\ (|t_1|_1 + |t_2|_1) + 1 \\
& \text{ because } \log_2\ x + \log_2\ y \le 2 * \log_2\ (x + y) \text{ if } x,y > 0 \\
=\ & 3 * \log_2\ (|t_1|_1 + |t_2|_1) + 1
\end{aligned}
$$

Sources

The inventors of skew heaps:

Daniel Sleator and Robert Tarjan.
Self-adjusting Heaps.
SIAM J. Computing, 1986.

Sources

The inventors of skew heaps:

Daniel Sleator and Robert Tarjan.
Self-adjusting Heaps.
SIAM J. Computing, 1986.

The formalization is based on

Anne Kaldewaij and Berry Schoenmakers.
The Derivation of a Tighter Bound for Top-down Skew Heaps. *Information Processing Letters*, 1991.

Sources

The inventors of skew heaps:

Daniel Sleator and Robert Tarjan.
Self-adjusting Heaps.
SIAM J. Computing, 1986.

The formalization is based on

Anne Kaldewaij and Berry Schoenmakers.
The Derivation of a Tighter Bound for Top-down Skew Heaps. *Information Processing Letters*, 1991.

Formalisation: TN

Conclusion

The Verification Perspective

Invariants and abstract functions are key

Conclusion

The Verification Perspective

Invariants and abstract functions are key

Main invariants are good for intuition

Conclusion

The Verification Perspective

Invariants and abstract functions are key

Main invariants are good for intuition

Formal proof needs much more

Conclusion

The Verification Perspective

Invariants and abstract functions are key

Main invariants are good for intuition

Formal proof needs much more

Often unsuitable for presentation in seminar, paper or even book

Conclusion

The Verification Perspective

Invariants and abstract functions are key

Main invariants are good for intuition

Formal proof needs much more

Often unsuitable for presentation in seminar, paper or even book

Can the queue verifications be automated more?

Conclusion

The Verification Perspective

Invariants and abstract functions are key

Main invariants are good for intuition

Formal proof needs much more

Often unsuitable for presentation in seminar, paper or even book

Can the queue verifications be automated more?

Verification of lazy versions?