Script **generated by TTT**

Title: FDS (12.07.2019)

Date: Fri Jul 12 08:36:41 CEST 2019

Duration: 83:38 min

Pages: 97

HOL/Data_Structures/
Binomial_Heap.thy

HOL/Data_Structures/
Binomial_Heap.thy

Numerical method

Idea: only use trees t_i of size 2^i

Numerical method

Idea: only use trees t_i of size 2^i

Numerical method

Idea: only use trees t_i of size 2^i

Example

To store (in binary) 11001 elements: $[t_0,0,0,t_3,t_4]$

Numerical method

Idea: only use trees t_i of size 2^i

Example

To store (in binary) 11001 elements: $[t_0,0,0,t_3,t_4]$

Merge $\approx$ addition with carry

Numerical method

Idea: only use trees t_i of size 2^i

Example

To store (in binary) 11001 elements: $[t_0,0,0,t_3,t_4]$

Merge $\approx$ addition with carry

Needs function to combine two trees of size 2^i into one tree of size 2^{i+1}

Binomial tree

datatype $'a\ tree =$
$\quad Node\ (rank{:}\ nat)\ (root{:}\ 'a)\ ('a\ tree\ list)$

Binomial tree

datatype $'a\ tree =$
$\quad Node\ (rank{:}\ nat)\ (root{:}\ 'a)\ ('a\ tree\ list)$

Invariant: Node of rank r has children $[t_{r-1}, \dots t_0]$ of ranks $[r{-}1, \dots, 0]$

Binomial tree

datatype $'a\ tree =$
$\quad Node\ (rank{:}\ nat)\ (root{:}\ 'a)\ ('a\ tree\ list)$

Invariant: Node of rank r has children $[t_{r-1}, \dots t_0]$ of ranks $[r{-}1, \dots, 0]$

$invar\_btree\ (Node\ r\ x\ ts) =$
$((\forall\, t{\in}set\ ts.\ invar\_btree\ t) \wedge map\ rank\ ts = rev\ [0..{<}r])$

Binomial tree

datatype $'a\ tree =$
$\quad Node\ (rank{:}\ nat)\ (root{:}\ 'a)\ ('a\ tree\ list)$

Invariant: Node of rank r has children $[t_{r-1}, \dots t_0]$ of ranks $[r{-}1, \dots, 0]$

$invar\_btree\ (Node\ r\ x\ ts) =$
$((\forall\, t{\in}set\ ts.\ invar\_btree\ t) \wedge map\ rank\ ts = rev\ [0..{<}r])$

Lemma
$invar\_btree\ t \Longrightarrow |t| = 2^{rank\ t}$

Binomial tree

datatype *'a tree* =
Node (rank: nat) (root: 'a) ('a tree list)

Invariant: Node of rank r has children $[t_{r-1}, \ldots t_0]$
of ranks $[r-1, \ldots, 0]$

Numerical method

Idea: only use trees t_i of size 2^i

Example

To store (in binary) 11001 elements: $[t_0, 0, 0, t_3, t_4]$

Merge $\approx$ addition with carry
Needs function to combine two trees of size 2^i
into one tree of size 2^{i+1}

Combining two trees

How to combine two trees of rank i
into one tree of rank $i+1$

Binomial tree

datatype *'a tree* =
Node (rank: nat) (root: 'a) ('a tree list)

Binomial heap

Use sparse representation for binary numbers:
$[t_0,0,0,t_3,t_4]$ represented as $[\ (0,t_0),\ (3,t_3),(4,t_4)\ ]$

Binomial heap

Use sparse representation for binary numbers:
$[t_0,0,0,t_3,t_4]$ represented as $[\ (0,t_0),\ (3,t_3),(4,t_4)\ ]$

type_synonym $'a\ heap = {}'a\ tree\ list$

Remember: *tree* contains rank

Binomial heap

Use sparse representation for binary numbers:
$[t_0,0,0,t_3,t_4]$ represented as $[\ (0,t_0),\ (3,t_3),(4,t_4)\ ]$

type_synonym $'a\ heap = {}'a\ tree\ list$

Remember: *tree* contains rank

Invariant:

$invar\_bheap\ ts =$
$((\forall\ t{\in}set\ ts.\ invar\_btree\ t) \wedge$
$\ sorted\_wrt\ (<)\ (map\ rank\ ts))$

Inserting a tree

$ins\_tree\ t\ []\ =\ [t]$
$ins\_tree\ t_1\ (t_2\ \#\ ts)\ =$
(if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ t_2\ \#\ ts$
 else $ins\_tree\ (link\ t_1\ t_2)\ ts$)

Inserting a tree

$ins\_tree\ t\ []\ =\ [t]$
$ins\_tree\ t_1\ (t_2\ \#\ ts)\ =$
$(\text{if}\ rank\ t_1\ <\ rank\ t_2\ \text{then}\ t_1\ \#\ t_2\ \#\ ts$
$\ \text{else}\ ins\_tree\ (link\ t_1\ t_2)\ ts)$

Intuition: Handle a carry

Precondition:
Rank of inserted tree $\leq$ ranks of trees in heap

merge

$merge\ ts_1\ []\ =\ ts_1$
$merge\ []\ ts_2\ =\ ts_2$
$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2)\ =$
$(\text{if}\ rank\ t_1\ <\ rank\ t_2\ \text{then}\ t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \text{else if}\ rank\ t_2\ <\ rank\ t_1\ \text{then}\ t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad \text{else}\ ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

merge

$merge\ ts_1\ []\ =\ ts_1$
$merge\ []\ ts_2\ =\ ts_2$
$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2)\ =$
$(\text{if}\ rank\ t_1\ <\ rank\ t_2\ \text{then}\ t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \text{else if}\ rank\ t_2\ <\ rank\ t_1\ \text{then}\ t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad \text{else}\ ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Intuition: Addition of binary numbers

Note: Handling of carry *after* recursive call

Get/delete minimum element

All trees are min-heaps.

merge

$merge\ ts_1\ [] = ts_1$
$merge\ []\ ts_2 = ts_2$
$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$(\text{if } rank\ t_1 < rank\ t_2 \text{ then } t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \text{else if } rank\ t_2 < rank\ t_1 \text{ then } t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad \text{else } ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Intuition: Addition of binary numbers

Note: Handling of carry *after* recursive call

Get/delete minimum element

All trees are min-heaps.

Smallest element may be any root node:

$ts \neq [] \Longrightarrow get\_min\ ts = Min\ (set\ (map\ root\ ts))$

Get/delete minimum element

All trees are min-heaps.

Smallest element may be any root node:

$ts \neq [] \Longrightarrow get\_min\ ts = Min\ (set\ (map\ root\ ts))$

Similar:

$get\_min\_rest :: 'a\ tree\ list \Rightarrow 'a\ tree \times 'a\ tree\ list$

Returns tree with minimal root, and remaining trees

Get/delete minimum element

All trees are min-heaps.

Smallest element may be any root node:

$ts \neq [] \Longrightarrow get\_min\ ts = Min\ (set\ (map\ root\ ts))$

Similar:

$get\_min\_rest :: 'a\ tree\ list \Rightarrow 'a\ tree \times 'a\ tree\ list$

Returns tree with minimal root, and remaining trees

$del\_min\ ts =$
$(\text{case } get\_min\_rest\ ts \text{ of}$
$\quad (Node\ r\ x\ ts_1,\ ts_2) \Rightarrow merge\ (rev\ ts_1)\ ts_2)$

Why *rev*?

Get/delete minimum element

All trees are min-heaps.

Smallest element may be any root node:

$ts \neq [] \Longrightarrow get\_min\ ts = Min\ (set\ (map\ root\ ts))$

Similar:

$get\_min\_rest :: {'a}\ tree\ list \Rightarrow {'a}\ tree \times {'a}\ tree\ list$

Returns tree with minimal root, and remaining trees

$del\_min\ ts =$

(case $get\_min\_rest\ ts$ of

$(Node\ r\ x\ ts_1,\ ts_2) \Rightarrow merge\ (rev\ ts_1)\ ts_2)$

Why rev? Rank decreasing in ts_1 but increasing in ts_2

Complexity

Recall: $|t| = 2^{rank\ t}$

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ ts} \leq |ts| + 1$

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ ts} \leq |ts| + 1$

Complexity of operations: linear in length of heap

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ ts} \leq |ts| + 1$

Complexity of operations: linear in length of heap
i.e., logarithmic in number of elements

Proofs:

Complexity of *merge*

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$($if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Complexity of *merge*

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$($if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Complexity of ins_tree: $t\_ins\_tree\ t\ ts \leq length\ ts + 1$

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ ts} \leq |ts| + 1$

Complexity of operations: linear in length of heap
i.e., logarithmic in number of elements

Proofs: straightforward?

Inserting a tree

$ins\_tree\ t\ [] = [t]$
$ins\_tree\ t_1\ (t_2\ \#\ ts) =$
$(\text{if } rank\ t_1 < rank\ t_2 \text{ then } t_1\ \#\ t_2\ \#\ ts$
$\ \text{else } ins\_tree\ (link\ t_1\ t_2)\ ts)$

Intuition: Handle a carry

Complexity of *merge*

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$(\text{if } rank\ t_1 < rank\ t_2 \text{ then } t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \text{else if } rank\ t_2 < rank\ t_1 \text{ then } t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad \text{else } ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Complexity of ins_tree: $t\_ins\_tree\ t\ ts \le length\ ts + 1$

A call $merge\ t_1\ t_2$ (where $length\ t_1 = length\ t_2 = n$) can lead to calls of ins_tree on lists of length 1, ..., n.

$\sum \in O(n^2)$

Complexity of *merge*

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$(\text{if } rank\ t_1 < rank\ t_2 \text{ then } t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \text{else if } rank\ t_2 < rank\ t_1 \text{ then } t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad \text{else } ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Relate time and length of input/output:

$t\_ins\_tree\ t\ ts + length\ (ins\_tree\ t\ ts) = 2 + length\ ts$

$length\ (merge\ ts_1\ ts_2) + t\_merge\ ts_1\ ts_2$
$\le 2 * (length\ ts_1 + length\ ts_2) + 1$

Complexity of *merge*

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$(\text{if } rank\ t_1 < rank\ t_2 \text{ then } t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \text{else if } rank\ t_2 < rank\ t_1 \text{ then } t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad \text{else } ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Relate time and length of input/output:

$t\_ins\_tree\ t\ ts + length\ (ins\_tree\ t\ ts) = 2 + length\ ts$

$length\ (merge\ ts_1\ ts_2) + t\_merge\ ts_1\ ts_2$
$\le 2 * (length\ ts_1 + length\ ts_2) + 1$

220

Complexity of *merge*

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$(\text{if } rank\ t_1 < rank\ t_2 \text{ then } t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\text{else if } rank\ t_2 < rank\ t_1 \text{ then } t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\text{else } ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Relate time and length of input/output:

$t\_ins\_tree\ t\ ts + length\ (ins\_tree\ t\ ts) = 2 + length\ ts$

$length\ (merge\ ts_1\ ts_2) + t\_merge\ ts_1\ ts_2$
$\leq 2 * (length\ ts_1 + length\ ts_2) + 1$

Yields desired linear bound!

Sources

The inventor of the binomial heap:

Jean Vuillemin.
A Data Structure for Manipulating Priority Queues.
CACM, 1978.

The functional version:

Chris Okasaki. *Purely Functional Data Structures.*
Cambridge University Press, 1998.

Priority queues so far

insert, *del_min* (and *merge*)
have logarithmic complexity

Skew Binomial Heap

Similar to binomial heap, but involving also *skew binary numbers*:

Skew Binomial Heap

Similar to binomial heap, but involving also *skew binary numbers*:

$d_1 \ldots d_n$ represents $\sum_{i=1}^{n} d_i * (2^{i+1} - 1)$
where $d_i \in \{0, 1, 2\}$

Complexity

Skew binomial heap:

insert in time $O(1)$
del_min and *merge* still $O(\log n)$

Complexity

Skew binomial heap:

insert in time $O(1)$
del_min and *merge* still $O(\log n)$

Fibonacci heap (imperative!):

insert and *merge* in time $O(1)$
del_min still $O(\log n)$

Complexity

Skew binomial heap:

insert in time $O(1)$
del_min and *merge* still $O(\log n)$

Fibonacci heap (imperative!):

insert and *merge* in time $O(1)$
del_min still $O(\log n)$

Every operation in time $O(1)$?

Puzzle

Design a functional queue
with (worst case) constant time *enq* and *deq* functions

Example

n increments of a binary counter starting with 0

Example

n increments of a binary counter starting with 0

- WCC of one increment?

WCC = worst case complexity

Example

n increments of a binary counter starting with 0

- WCC of one increment? $O(\log_2 n)$

WCC = worst case complexity

Example

n increments of a binary counter starting with 0

- WCC of one increment? $O(\log_2 n)$
- WCC of n increments? $O(n * \log_2 n)$

WCC = worst case complexity

Example

n increments of a binary counter starting with 0

- WCC of one increment? $O(\log_2 n)$
- WCC of n increments? $O(n * \log_2 n)$
- $O(n * \log_2 n)$ is too pessimistic!
- Every second increment is cheap and compensates for the more expensive increments

WCC = worst case complexity

Example

n increments of a binary counter starting with 0

- WCC of one increment? $O(\log_2 n)$
- WCC of n increments? $O(n * \log_2 n)$
- $O(n * \log_2 n)$ is too pessimistic!
- Every second increment is cheap and compensates for the more expensive increments

- Fact: WCC of n increments is $O(n)$

WCC = worst case complexity

The problem

WCC of individual operations
may lead to overestimation of
WCC of sequences of operations

Amortized analysis

Idea:

Try to determine the average cost of each operation
(in the worst case!)

Use cheap operations to pay for expensive ones

Amortized analysis

Idea:

Try to determine the average cost of each operation (in the worst case!)

Use cheap operations to pay for expensive ones

Method:

- Cheap operations pay extra (into a "bank account"), making them more expensive

Amortized analysis

Idea:

Try to determine the average cost of each operation (in the worst case!)

Use cheap operations to pay for expensive ones

Method:

- Cheap operations pay extra (into a "bank account"), making them more expensive
- Expensive operations withdraw money from the account, making them cheaper

Bank account = *Potential*

Bank account = *Potential*

- The potential ("credit") is implicitly "stored" in the data structure.
- Potential Φ :: *data-structure* $\Rightarrow$ *non-neg. number* tells us how much credit is stored in a data structure

Bank account = *Potential*

- The potential ("credit") is implicitly "stored" in the data structure.
- Potential Φ :: *data-structure* $\Rightarrow$ *non-neg. number* tells us how much credit is stored in a data structure
- Increase in potential = deposit to pay for *later* expensive operation

Bank account = *Potential*

- The potential ("credit") is implicitly "stored" in the data structure.
- Potential Φ :: *data-structure* $\Rightarrow$ *non-neg. number* tells us how much credit is stored in a data structure
- Increase in potential = deposit to pay for *later* expensive operation
- Decrease in potential = withdrawal to pay for expensive operation

Bank account = *Potential*

- The potential ("credit") is implicitly "stored" in the data structure.
- Potential Φ :: *data-structure* $\Rightarrow$ *non-neg. number* tells us how much credit is stored in a data structure
- Increase in potential = deposit to pay for *later* expensive operation
- Decrease in potential = withdrawal to pay for expensive operation

Back to example: counter

Increment:

- Actual cost: 1 for each bit flip

Back to example: counter

Increment:

- Actual cost: 1 for each bit flip
- Bank transaction:
  - pay in 1 for final $0 \to 1$ flip

Back to example: counter

Increment:

- Actual cost: 1 for each bit flip
- Bank transaction:
  - pay in 1 for final $0 \to 1$ flip
  - take out 1 for each $1 \to 0$ flip

Back to example: counter

Increment:

- Actual cost: 1 for each bit flip
- Bank transaction:
  - pay in 1 for final $0 \to 1$ flip
  - take out 1 for each $1 \to 0$ flip

  $\Longrightarrow$ increment has amortized cost $2 = 1+1$

Back to example: counter

Increment:

- Actual cost: 1 for each bit flip
- Bank transaction:
  - pay in 1 for final $0 \to 1$ flip
  - take out 1 for each $1 \to 0$ flip

  $\Longrightarrow$ increment has amortized cost $2 = 1+1$

Data structure

Given an implementation:

- Type τ
- Operation(s) $f :: \tau \Rightarrow \tau$

Data structure

Given an implementation:

- Type τ
- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)
- Initial value: $init :: \tau$
  (function "empty")

Needed for complexity analysis:

- Time/cost: $t\_f :: \tau \Rightarrow num$

Data structure

Given an implementation:

- Type τ
- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)
- Initial value: $init :: \tau$
  (function "empty")

Needed for complexity analysis:

- Time/cost: $t\_f :: \tau \Rightarrow num$
  ($num =$ some numeric type
  nat may be inconvenient)
- Potential $\Phi :: \tau \Rightarrow num$

Data structure

Given an implementation:

- Type τ
- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)
- Initial value: $init :: \tau$
  (function "empty")

Needed for complexity analysis:

- Time/cost: $t\_f :: \tau \Rightarrow num$
  ($num =$ some numeric type
  nat may be inconvenient)
- Potential $\Phi :: \tau \Rightarrow num$ (creative spark!)

Need to prove: $\Phi\ s \geq 0$ and $\Phi\ init = 0$

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$

Sequence of states:

$s_0 := init,$

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$

Sequence of states:

$s_0 := init,\ s_1 := f_1\ s_0,\ \ldots,\ s_n := f_n\ s_{n-1}$

Amortized cost := real cost + potential difference

$$a_{i+1} := t\_f_{i+1}\ s_i + \Phi\ s_{i+1} - \Phi\ s_i$$

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$

Sequence of states:

$s_0 := init,\ s_1 := f_1\ s_0,\ \ldots,\ s_n := f_n\ s_{n-1}$

Amortized cost := real cost + potential difference

$$a_{i+1} := t\_f_{i+1}\ s_i + \Phi\ s_{i+1} - \Phi\ s_i$$

$$\Longrightarrow$$

Sum of amortized costs $\geq$ sum of real costs

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$
Sequence of states:

$$s_0 := init,\ s_1 := f_1\ s_0,\ \ldots,\ s_n := f_n\ s_{n-1}$$

Amortized cost := real cost + potential difference

$$a_{i+1} := t\_f_{i+1}\ s_i + \Phi\ s_{i+1} - \Phi\ s_i$$

$$\Longrightarrow$$

Sum of amortized costs ≥ sum of real costs

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t\_f_i\ s_{i-1} + \Phi\ s_i - \Phi\ s_{i-1})$$

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$
Sequence of states:

$$s_0 := init,\ s_1 := f_1\ s_0,\ \ldots,\ s_n := f_n\ s_{n-1}$$

Amortized cost := real cost + potential difference

$$a_{i+1} := t\_f_{i+1}\ s_i + \Phi\ s_{i+1} - \Phi\ s_i$$

$$\Longrightarrow$$

Sum of amortized costs ≥ sum of real costs

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} (t\_f_i\ s_{i-1} + \Phi\ s_i - \Phi\ s_{i-1}) \\ &= \left(\sum_{i=1}^{n} t\_f_i\ s_{i-1}\right) + \Phi\ s_n - \Phi\ init\end{aligned}$$

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$
Sequence of states:

$$s_0 := init,\ s_1 := f_1\ s_0,\ \ldots,\ s_n := f_n\ s_{n-1}$$

Amortized cost := real cost + potential difference

$$a_{i+1} := t\_f_{i+1}\ s_i + \Phi\ s_{i+1} - \Phi\ s_i$$

$$\Longrightarrow$$

Sum of amortized costs ≥ sum of real costs

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} (t\_f_i\ s_{i-1} + \Phi\ s_i - \Phi\ s_{i-1}) \\ &= \left(\sum_{i=1}^{n} t\_f_i\ s_{i-1}\right) + \Phi\ s_n - \Phi\ init \\ &\geq \sum_{i=1}^{n} t\_f_i\ s_{i-1}\end{aligned}$$

Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$
Sequence of states:

$$s_0 := init,\ s_1 := f_1\ s_0,\ \ldots,\ s_n := f_n\ s_{n-1}$$

Amortized cost := real cost + potential difference

$$a_{i+1} := t\_f_{i+1}\ s_i + \Phi\ s_{i+1} - \Phi\ s_i$$

$$\Longrightarrow$$

Sum of amortized costs ≥ sum of real costs

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t\_f_i\ s_{i-1} + \Phi\ s_i - \Phi\ s_{i-1})$$

Back to example: counter

$incr :: bool\ list \Rightarrow bool\ list$

Back to example: counter

$incr :: bool\ list \Rightarrow bool\ list$
$incr\ [] = [True]$
$incr\ (False\ \#\ bs) = True\ \#\ bs$
$incr\ (True\ \#\ bs) = False\ \#\ incr\ bs$

Back to example: counter

$incr :: bool\ list \Rightarrow bool\ list$
$incr\ [] = [True]$
$incr\ (False\ \#\ bs) = True\ \#\ bs$
$incr\ (True\ \#\ bs) = False\ \#\ incr\ bs$

$init = []$

$\Phi\ bs = length\ (filter\ id\ bs)$

Back to example: counter

$incr :: bool\ list \Rightarrow bool\ list$
$incr\ [] = [True]$
$incr\ (False\ \#\ bs) = True\ \#\ bs$
$incr\ (True\ \#\ bs) = False\ \#\ incr\ bs$

$init = []$

$\Phi\ bs = length\ (filter\ id\ bs)$

Lemma
$t\_incr\ bs + \Phi\ (incr\ bs) - \Phi\ bs = 2$

Back to example: counter

$incr :: bool\ list \Rightarrow bool\ list$
$incr\ [] = [True]$
$incr\ (False \# bs) = True \# bs$
$incr\ (True \# bs) = False \# incr\ bs$

$init = []$

Proof obligation summary

- $\Phi\ s \geq 0$
- $\Phi\ init = 0$
- For every operation $f :: \tau \Rightarrow ... \Rightarrow \tau$:
  $t\_f\ s\ \overline{x} + \Phi(f\ s\ \overline{x}) - \Phi\ s \leq a\_f\ s\ \overline{x}$

Proof obligation summary

- $\Phi\ s \geq 0$
- $\Phi\ init = 0$
- For every operation $f :: \tau \Rightarrow ... \Rightarrow \tau$:
  $t\_f\ s\ \overline{x} + \Phi(f\ s\ \overline{x}) - \Phi\ s \leq a\_f\ s\ \overline{x}$

If the data structure has an invariant *invar*:
assume precondition *invar s*

Proof obligation summary

- $\Phi\ s \geq 0$
- $\Phi\ init = 0$
- For every operation $f :: \tau \Rightarrow ... \Rightarrow \tau$:
  $t\_f\ s\ \overline{x} + \Phi(f\ s\ \overline{x}) - \Phi\ s \leq a\_f\ s\ \overline{x}$

If the data structure has an invariant *invar*:
assume precondition *invar s*

If f takes 2 arguments of type τ:
$t\_f\ s_1\ s_2\ \overline{x} + \Phi(f\ s_1\ s_2\ \overline{x}) - \Phi\ s_1 - \Phi\ s_2 \leq a\_f\ s_1\ s_2\ \overline{x}$

Warning: real time

Amortized analysis unsuitable for real time applications:

Warning: single threaded

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

Warning: single threaded

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

Single threaded = no value is used more than once

Otherwise:

let $counter = 0;$
$bad =$ increment $counter$ $2^n - 1$ times;

Warning: single threaded

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

Single threaded = no value is used more than once

Otherwise:

let $counter = 0;$
$bad =$ increment $counter$ $2^n - 1$ times;
$\_ = incr\ bad;$
$\_ = incr\ bad;$
$\_ = incr\ bad;$
$\vdots$

Warning: observer functions

Observer function: does not modify data structure

$\Longrightarrow$ Potential difference $= 0$

$\Longrightarrow$ amortized cost $=$ real cost

Warning: observer functions

Observer function: does not modify data structure

$\Longrightarrow$ Potential difference $= 0$

$\Longrightarrow$ amortized cost $=$ real cost

$\Longrightarrow$ Must analyze WCC of observer functions