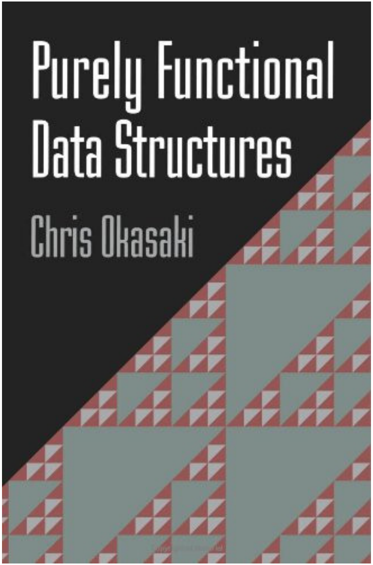


Purely Functional Data Structures

Daniel Alonso

Introduction

What is “purely functional”?

Referential transparency: substituting an expression by its value does not change the program’s behavior.

For data structures, it is essentially immutability (though there are other restrictions, e.g. not using randomization without explicitly passing a seed).

Purely functional arrays

How do we modify a purely functional array? We don't. Think copy on write.

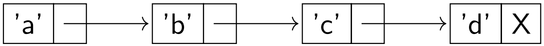
original	'a'	'b'	'c'	$\cdots$	'm'	$\cdots$	'x'	'y'	'z'

To "modify" a single element,
we must copy the whole array

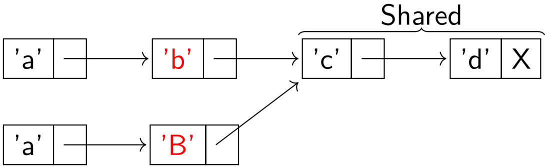
copy	'a'	'b'	'c'	$\cdots$	'M'	$\cdots$	'x'	'y'	'z'

Cost of "modifying" a single array element: $\Theta(n)$.

Lists

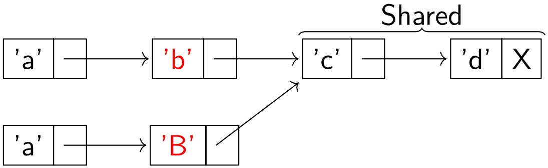


Lists



When “modifying” an element, we must copy all parts of the data structure that point (directly or indirectly) to it.

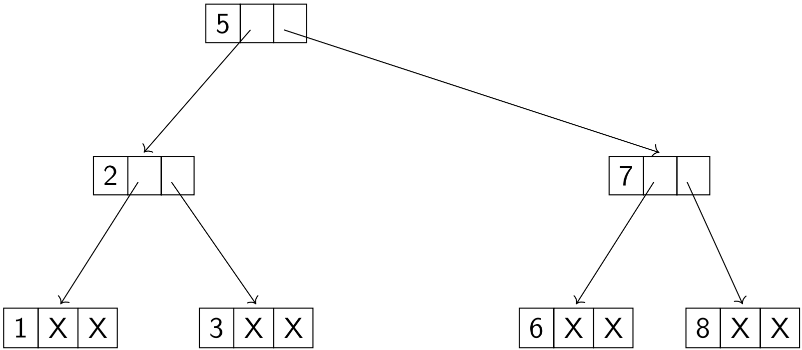
Lists



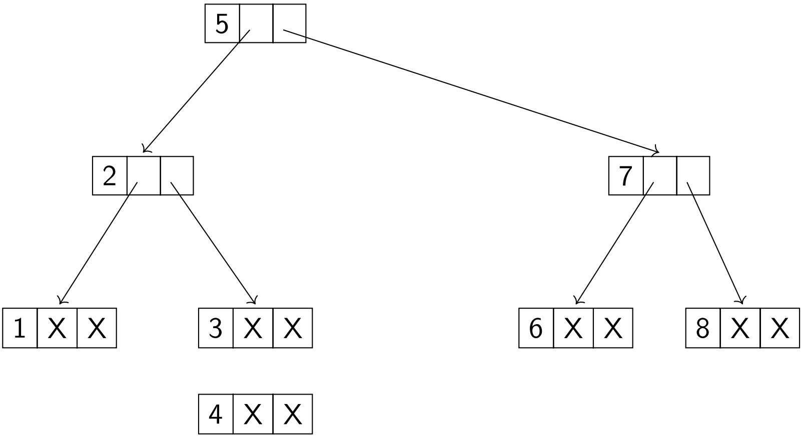
When "modifying" an element, we must copy all parts of the data structure that point (directly or indirectly) to it.

Cost of "modifying" a single list element: $O(n)$.

Binary trees

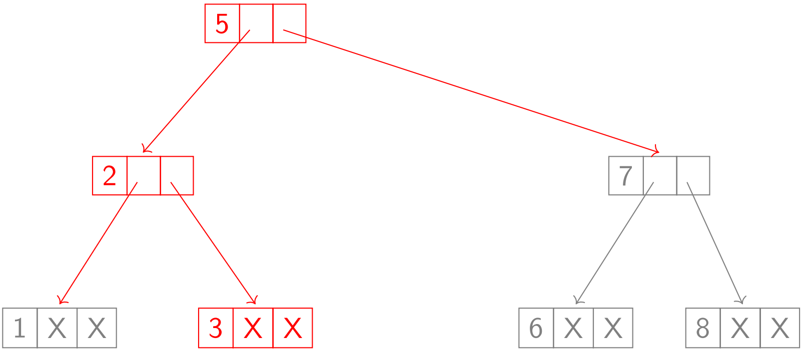


Binary trees



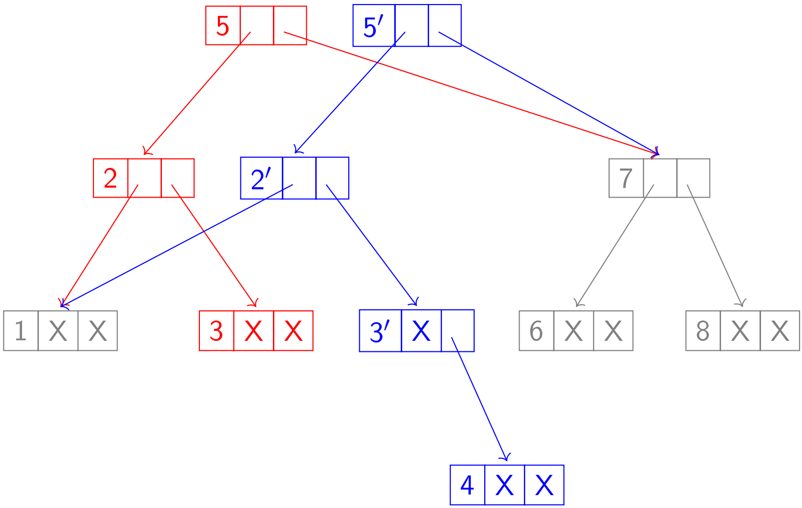
Insert 4 as a child of 3

Binary trees

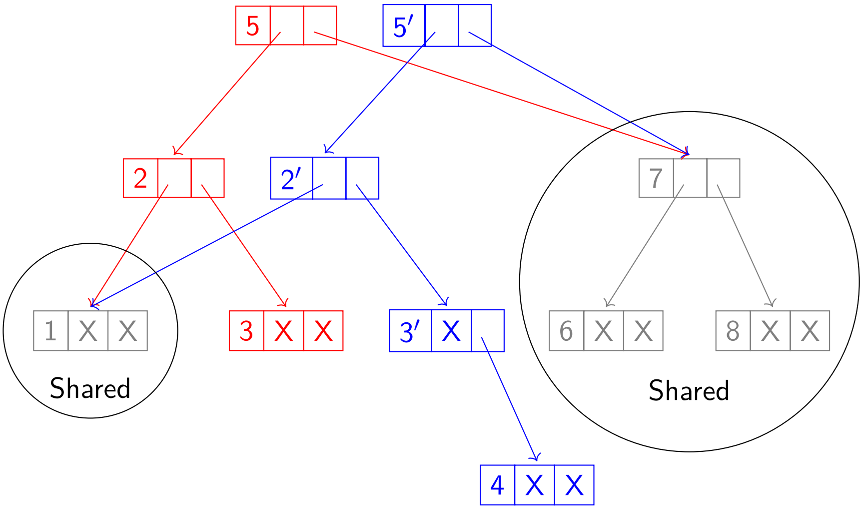


All nodes that point (directly or indirectly) to 3: path to the root.

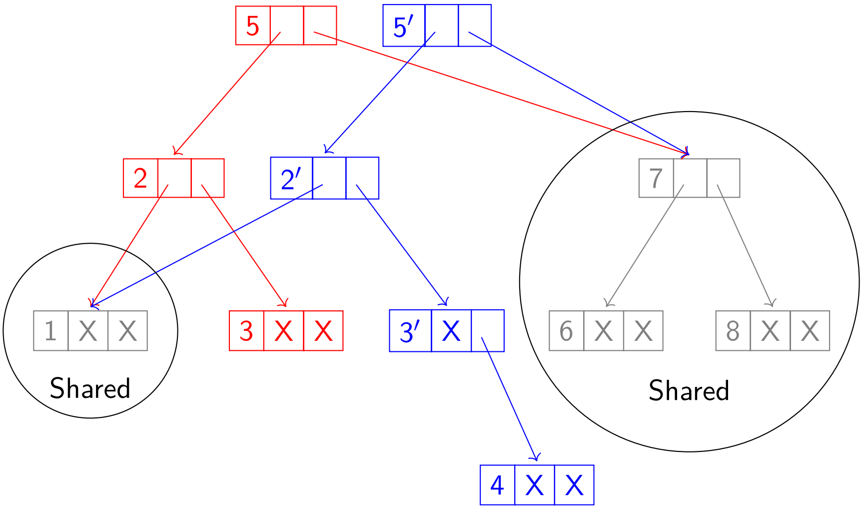
Binary trees



Binary trees



Binary trees



In a balanced tree, we will copy $O(\log n)$ nodes.

Persistence

Because we copy (instead of modify) data structures, we keep both the input and the output of every operation. All the "versions" of a data structure that have ever existed in our program remain available (unless garbage collected).

Running example: functional queue

```
class Queue q where
    empty :: q a -- trivial
    isEmpty :: q a -> Bool -- trivial
    head :: q a -> Maybe a
    snoc :: q a -> a -> q a -- snoc is rev of cons
    tail :: q a -> Maybe (q a)
```

Simplest queue: list

```
data ListQueue a = LQ [a]
instance Queue ListQueue where
```

Function	Cost
`head (LQ []) = Nothing` `head (LQ (x:_) = Just x`	$\Theta(1)$
`snoc (LQ xs) x = LQ (xs ++ [x])`	$\Theta(n)$
`tail (LQ []) = Nothing` `tail (LQ (_:xs)) = Just (LQ xs)`	$\Theta(1)$

Batched queue: code

```
data BatchedQueue a = BQ [a] [a]
```

- Invariant: `BQ fs rs` is empty $\iff$ `fs` is empty.
- `rs` contains the last elements of the queue in reverse order.

Example: we can represent | 1 | 2 | 3 | 4 | 5 | 6 | as `BQ [1,2,3] [6,5,4]`.

Batched queue: code

```
data BatchedQueue a = BQ [a] [a]
```

- Invariant: `BQ fs rs` is empty $\iff$ `fs` is empty.
- `rs` contains the last elements of the queue in reverse order.

Example: we can represent | 1 | 2 | 3 | 4 | 5 | 6 | as `BQ [1,2,3] [6,5,4]`.

Function	Cost
`head (BQ [] _) = Nothing` `head (BQ (x:_) _) = Just x`	$\Theta(1)$
`snoc (BQ fs rs) x = check fs (x:rs)`	$\Theta(1)$
`tail (BQ [] _) = Nothing` `tail (BQ (_:fs) rs) = Just (check fs rs)`	$O(n)$
`check [] rs = BQ (reverse rs) []` `check fs rs = BQ fs rs`	

Batched queue: example

```
BQ [] []
```

Batched queue: example

```
snoc (BQ fs rs) x = check fs (x:rs)
check [] rs = BQ (reverse rs) []
check fs rs = BQ fs rs
```

$$\begin{array}{l} \texttt{BQ [] []} \\ \quad \big\downarrow \texttt{snoc 1} \qquad (\texttt{check [] [1]} \rightarrow \texttt{fs}\text{ empty, reverse }\texttt{rs}) \\ \texttt{BQ [1] []} \end{array}$$

Batched queue: example

```
snoc (BQ fs rs) x = check fs (x:rs)
check [] rs = BQ (reverse rs) []
check fs rs = BQ fs rs
```

```
BQ [] []
    ↓ snoc 1    (check [] [1] → fs empty, reverse rs)
BQ [1] []
    ↓ snoc 2    (check [1] [2] → fs not empty)
BQ [1] [2]
```

Batched queue: example

```
snoc (BQ fs rs) x = check fs (x:rs)
check [] rs = BQ (reverse rs) []
check fs rs = BQ fs rs
```

`BQ [] []`

↓ `snoc 1` (`check [] [1]` → `fs` empty, reverse `rs`)

`BQ [1] []`

↓ `snoc 2` (`check [1] [2]` → `fs` not empty)

`BQ [1] [2]`

↓ `snoc 3` (same)

`BQ [1] [3,2]`

Batched queue: example

```
tail (BQ (_:fs) rs) = Just (check fs rs)
check [] rs = BQ (reverse rs) []
check fs rs = BQ fs rs
```

$$
\begin{array}{rl}
& \texttt{BQ [] []} \\
& \Big\downarrow \texttt{snoc 1} \quad (\texttt{check [] [1]} \to \texttt{fs}\text{ empty, reverse }\texttt{rs}) \\
& \texttt{BQ [1] []} \\
& \Big\downarrow \texttt{snoc 2} \quad (\texttt{check [1] [2]} \to \texttt{fs}\text{ not empty}) \\
& \texttt{BQ [1] [2]} \\
& \Big\downarrow \texttt{snoc 3} \quad (\text{same}) \\
& \texttt{BQ [1] [3,2]} \\
\color{red}{\Theta(n)} & \Big\downarrow \texttt{tail} \quad (\texttt{check [] [3,2]} \to \texttt{fs}\text{ empty, reverse }\texttt{rs}) \\
& \texttt{Just (BQ [2,3] [])}
\end{array}
$$

Batched queue: example

```
tail (BQ (_:fs) rs) = Just (check fs rs)
check [] rs = BQ (reverse rs) []
check fs rs = BQ fs rs
```

`BQ [] []`

$\downarrow$ `snoc 1` (`check [] [1]` $\to$ `fs` empty, reverse `rs`)

`BQ [1] []`

$\downarrow$ `snoc 2` (`check [1] [2]` $\to$ `fs` not empty)

`BQ [1] [2]`

$\downarrow$ `snoc 3` (same)

`BQ [1] [3,2]`

$\Theta(n)$ $\downarrow$ `tail` (`check [] [3,2]` $\to$ `fs` empty, reverse `rs`)

`Just (BQ [2,3] [])`

$\Theta(1)$ $\downarrow$ `tail` (`check [3] []` $\to$ `fs` not empty)

`Just (BQ [3] [])`

Amortization

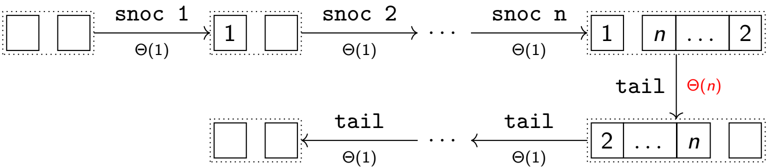
Amortized analysis

- Worst case of `tail` for `BatchedQueue` is $\Theta(n)$, though most `tail` operations will be $\Theta(1)$.
- Amortized analysis: bound the (worst case) cost of a sequence of operations, instead of each operation individually.

Amortized analysis

- Worst case of tail for `BatchedQueue` is $\Theta(n)$, though most tail operations will be $\Theta(1)$.
- Amortized analysis: bound the (worst case) cost of a sequence of operations, instead of each operation individually.

Example:



Although there are n calls to `tail`, the cost of the whole sequence of operations is $\Theta(n)$ and not $\Theta(n^2)$.

Amortized analysis (ii)

For a sequence of m operations, define:

- t_i: actual cost of operation i.
- a_i: amortized cost of operation i.

We want to prove that:

$$\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i,$$

that is, the total actual cost is bounded above by the total amortized cost.

Banker's method

- Each operation can deposit or spend some (imaginary) credits at different locations of the data structure.
  - Deposit: pay in advance to reduce the cost of a future operation.
  - Spend: take advantage of a "discount" paid for by a past operation.

- Define the amortized cost of operation i as:

$$a_i = t_i + c_i - \bar{c}_i, \quad \text{where} \begin{cases} c_i : \text{credits deposited} \\ \bar{c}_i : \text{credits spent} \end{cases}$$

- Credits can be spent only once.

Banker's method

- Each operation can deposit or spend some (imaginary) credits at different locations of the data structure.
  - Deposit: pay in advance to reduce the cost of a future operation.
  - Spend: take advantage of a "discount" paid for by a past operation.
- Define the amortized cost of operation i as:

$$a_i = t_i + c_i - \bar{c}_i, \quad \text{where} \begin{cases} c_i : \text{credits deposited} \\ \bar{c}_i : \text{credits spent} \end{cases}$$

- Credits can be spent only once.

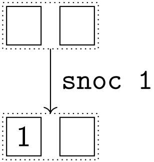
This is an analysis tool. These credits do not exist in code!

Batched queue: amortized analysis

- `head`: does not deposit nor spend any credits, $a_i = t_i = 1$

Batched queue: amortized analysis

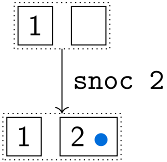
- `head`: does not deposit nor spend any credits, $a_i = t_i = 1$
- `snoc`: empty no credit deposited or spent, $a_i = t_i = 1$



Batched queue: amortized analysis

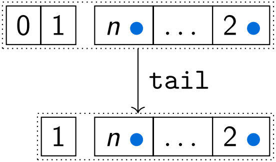
- `head`: does not deposit nor spend any credits, $a_i = t_i = 1$
- `snoc`:

empty	no credit deposited or spent, $a_i = t_i = 1$
not empty	deposit one credit to the new head of the rear list, $a_i = t_i + 1 = 2$



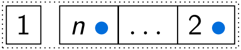
Batched queue: amortized analysis

- `head`: does not deposit nor spend any credits, $a_i = t_i = 1$
- `snoc`: empty — no credit deposited or spent, $a_i = t_i = 1$
  not empty — deposit one credit to the new head of the rear list, $a_i = t_i + 1 = 2$
- `tail`: no rotation — no credit deposited or spent, $a_i = t_i = 1$



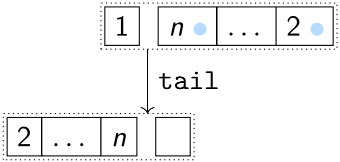
Batched queue: amortized analysis

- `head`: does not deposit nor spend any credits, $a_i = t_i = 1$
- `snoc`: empty — no credit deposited or spent, $a_i = t_i = 1$
  not empty — deposit one credit to the new head of the rear list, $a_i = t_i + 1 = 2$
- `tail`: no rotation — no credit deposited or spent, $a_i = t_i = 1$
  rotation — spend all credits in the rear list, $a_i = t_i - m = (1 + m) - m = 1$



Batched queue: amortized analysis

- `head`: does not deposit nor spend any credits, $a_i = t_i = 1$
- `snoc`: empty — no credit deposited or spent, $a_i = t_i = 1$
  not empty — deposit one credit to the new head of the rear list, $a_i = t_i + 1 = 2$
- `tail`: no rotation — no credit deposited or spent, $a_i = t_i = 1$
  rotation — spend all credits in the rear list, $a_i = t_i - m = (1 + m) - m = 1$

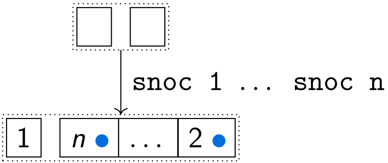


Batched queue: amortized analysis

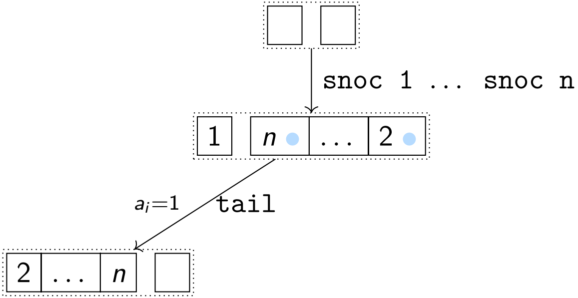
- `head`: does not deposit nor spend any credits, $a_i = t_i = 1$
- `snoc`: empty — no credit deposited or spent, $a_i = t_i = 1$
  not empty — deposit one credit to the new head of the rear list, $a_i = t_i + 1 = 2$
- `tail`: no rotation — no credit deposited or spent, $a_i = t_i = 1$
  rotation — spend all credits in the rear list, $a_i = t_i - m = (1 + m) - m = 1$

Function	Worst case	Amortized
`head`	$\Theta(1)$	$\Theta(1)$
`snoc`	$\Theta(1)$	$\Theta(1)$
`tail`	$O(n)$	$\Theta(1)$

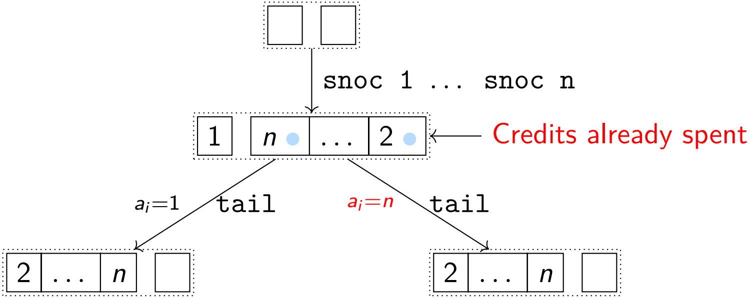
Persistence breaks amortization



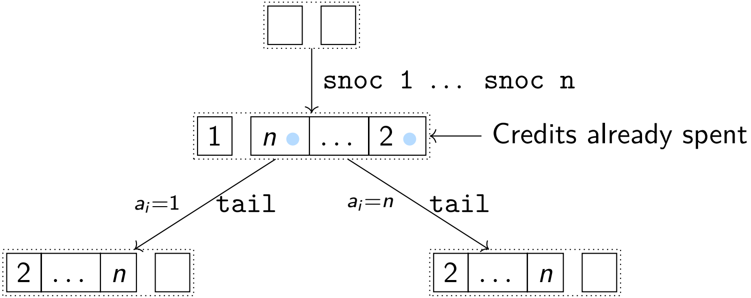
Persistence breaks amortization



Persistence breaks amortization



Persistence breaks amortization



Persistence allows performing expensive operations an arbitrary number of times. n snocs followed by n tails on [1] [n | ... | 2] have $\Theta(n^2)$ cost. The amortized worst case cost for tail with persistence must be $\Theta(n)$.

Things to remember

- Amortized analysis bounds the cost of a sequence of operations.
- In the banker's method, operations deposit (imaginary) credits to pay for the cost of other operations in advance.
- Persistence breaks standard amortized analysis.

Fixing amortized analysis with laziness

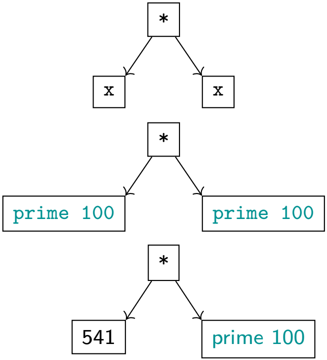
Laziness reminder

Call-by-need:

- Do not evaluate arguments to functions before evaluating the function.

Call-by-name

```
square x = x * x

square (prime 100)
```

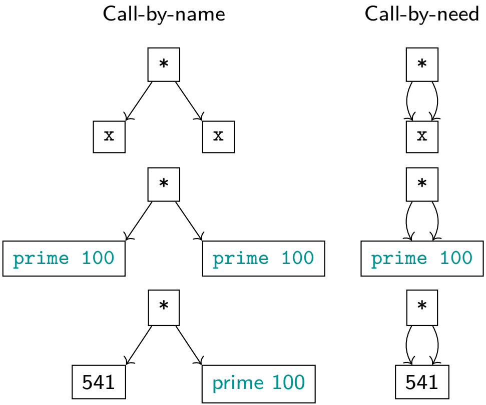


Laziness reminder

Call-by-need:

- Do not evaluate arguments to functions before evaluating the function.
- Memoize the value of named expressions (difference with call-by-name).

```
square x = x * x

square (prime 100)
```



Amortization with laziness

Divide the complete cost of operation i between:

Unshared: cost assuming every prior suspension is already memoized.

Shared: cost of forcing every suspension created but no evaluated by i.

Example:

```
     [] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
[1,2,3,4] ++ [5,6]
```

$$\underbrace{\texttt{1 :}}_{\substack{\text{unshared}\\(1)}}\ \underbrace{\texttt{([2,3,4] ++ [5,6])}}_{\substack{\text{shared}\\(3)}}$$

Amortization with laziness

Divide the complete cost of operation i between:

Unshared: cost assuming every prior suspension is already memoized.

Shared: cost of forcing every suspension created but no evaluated by i.

Given a sequence of m operations, divide the shared costs between:

Realized: costs for suspensions evaluated at some point.

Unrealized: costs for suspensions never evaluated.

Example:

```
print $ take 3 $ [1,2,3,4] ++ [5,6]
```

$$\downarrow$$

$$\underbrace{\texttt{1 :}}_{\substack{\text{unshared}\\(1)}}\ \underbrace{\texttt{2 : 3 :}}_{\substack{\text{realized}\\(2)}}\ \underbrace{\texttt{([4] ++ [5,6])}}_{\substack{\text{unrealized}\\(1)}}$$

Amortization with laziness

Divide the complete cost of operation i between:

Unshared: cost assuming every prior suspension is already memoized.

Shared: cost of forcing every suspension created but no evaluated by i.

Given a sequence of m operations, divide the shared costs between:

Realized: costs for suspensions evaluated at some point.

Unrealized: costs for suspensions never evaluated.

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (\mathrm{unshared}_i + \mathrm{realized}_i) \leq \sum_{i=1}^{m} a_i$$

Adapting the Banker's method

Replace credits with debits: created to represent the debt of the *shared cost* of an operation. Must be paid before forcing a suspended computation.

$$a_i = \text{unshared}_i + d_i$$

a_i takes into account debits discharged (d_i), but not debits created (they could belong to unrealized suspensions).

Adapting the Banker's method

Replace credits with debits: created to represent the debt of the *shared cost* of an operation. Must be paid before forcing a suspended computation.

$$a_i = \mathrm{unshared}_i + d_i$$

a_i takes into account debits discharged (d_i), but not debits created (they could belong to unrealized suspensions).

Because no suspension is executed before paying its debits, we have:

$$\sum_{i=1}^{m} (\mathrm{unshared}_i + \mathrm{realized}_i) \leq \sum_{i=1}^{m} a_i$$

Persistent queue: code

```
data PersistentQueue a = PQ Int [a] Int [a]
```

- In `PQ lenf fs lenr rs`, length of `fs` $\geq$ length of `rs`.
- `rs` contains the last elements of the queue in reverse order.

Function	Cost
`head (PQ _ [] _ _) = Nothing` `head (PQ _ (x:_) _ _) = Just x`	$\Theta(1)$
`snoc (PQ lenf fs lenr rs) x =` `    check lenf fs (lenr + 1) (x:rs)`	$O(n)$

`tail (PQ _ [] _ _) = Nothing`
`tail (PQ lenf (_:fs) lenr rs) =`
`    Just (check (lenf - 1) fs lenr rs)`
	$O(n)$

`check lenf fs lenr rs`
`    | lenr > lenf = PQ (lenf + lenr) (fs ++ reverse rs) 0 []`
`    | otherwise = PQ lenf fs lenr rs`	

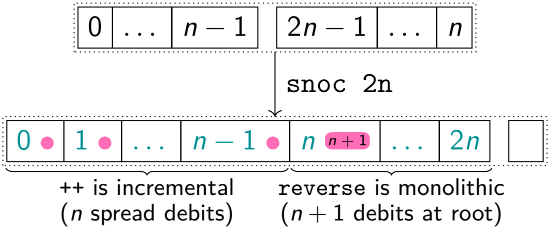
Persistent queue: amortized analysis intuition

- First location in the queue must have no (unpaid) debit at any moment (so `head` can peek at it).
- All debits must have been discharged before a rotation.

Persistent queue: amortized analysis intuition

- First location in the queue must have no (unpaid) debit at any moment (so `head` can peek at it).
- All debits must have been discharged before a rotation.

After a rotation, we create the following debits:

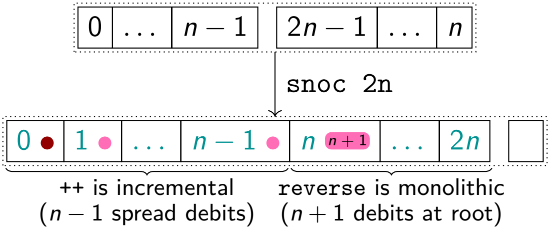


(`fs` becomes `[0..n-1] ++ (reverse [2n..n])`)

Persistent queue: amortized analysis intuition

- First location in the queue must have no (unpaid) debit at any moment (so `head` can peek at it).
- All debits must have been discharged before a rotation.

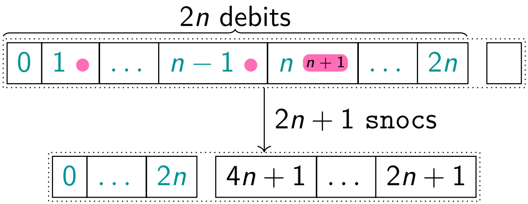
After a rotation, we create the following debits:



`snoc 2n` discharges the first debit. Remaining debits: $2n$.

Persistent queue: amortized analysis intuition

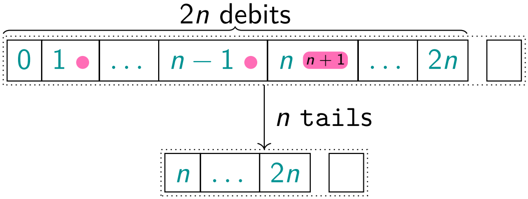
- First location in the queue must have no (unpaid) debit at any moment (so `head` can peek at it).
- All debits must have been discharged before a rotation.



After $2n+1$ `snocs` all debits must be discharged, as next `snoc` will cause a rotation. Each snoc discharges 1 debit.

Persistent queue: amortized analysis intuition

- First location in the queue must have no (unpaid) debit at any moment (so `head` can peek at it).
- All debits must have been discharged before a rotation.



After n `tails` all debits must be discharged, as head must be able to peek at the first location. Each `tail` discharges 2 debits.

Persistent queue: amortized analysis intuition

- First location in the queue must have no (unpaid) debit at any moment (so `head` can peek at it).
- All debits must have been discharged before a rotation.

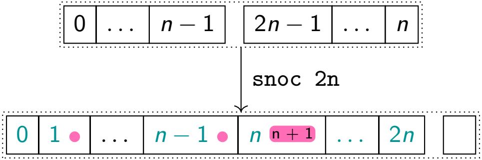
$$a_i = \text{unshared}_i + d_i$$

Function	Worst case	unshared_i	d_i	Amortized
`head`	$\Theta(1)$	$\Theta(1)$	0	$\Theta(1)$
`snoc`	$O(n)$	$\Theta(1)$	1	$\Theta(1)$
`tail`	$O(n)$	$\Theta(1)$	2	$\Theta(1)$

Persistent queue: amortized bounds hold with persistence

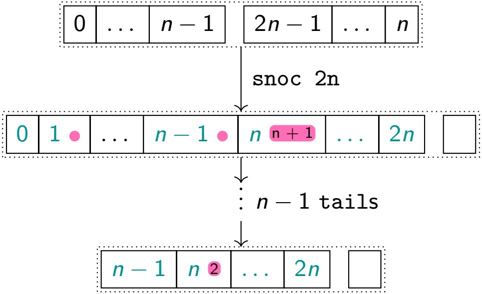
0	$\ldots$	$n-1$		$2n-1$	$\ldots$	n

Persistent queue: amortized bounds hold with persistence



`[0..n-1] ++ (reverse [2n..n])`

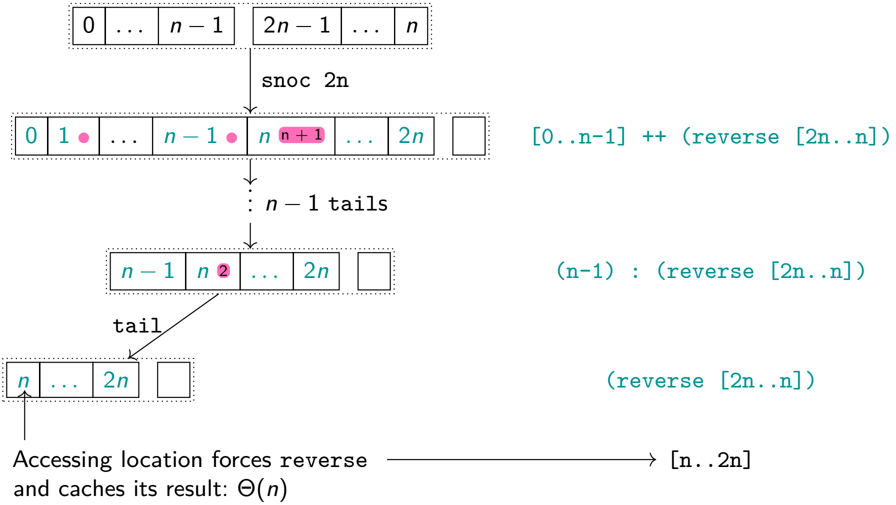
Persistent queue: amortized bounds hold with persistence



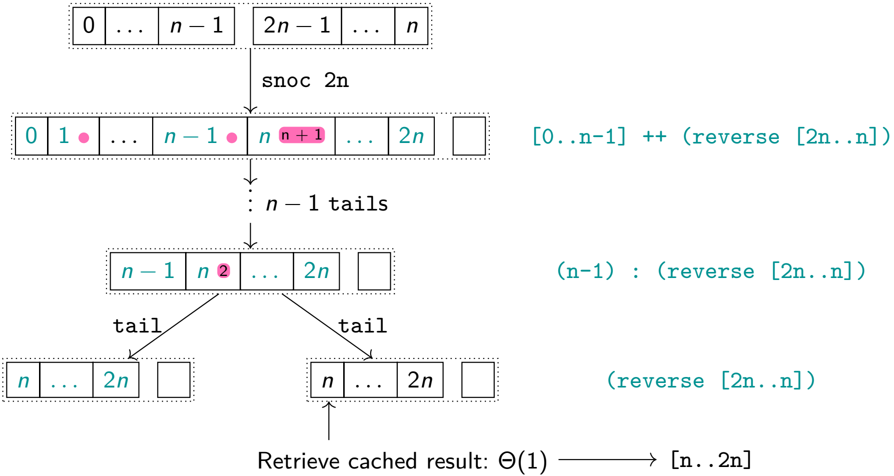
`[0..n-1] ++ (reverse [2n..n])`

`(n-1) : (reverse [2n..n])`

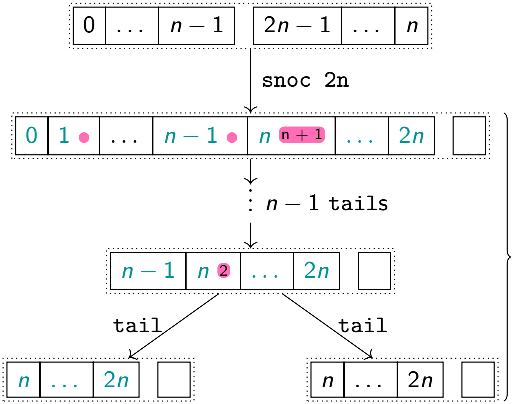
Persistent queue: amortized bounds hold with persistence



Persistent queue: amortized bounds hold with persistence

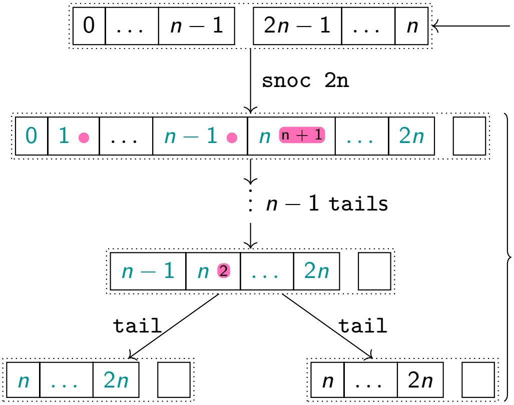


Persistent queue: amortized bounds hold with persistence



Branching from any of these queues shares the suspension

Persistent queue: amortized bounds hold with persistence



Branching from here creates a different suspension, sufficiently in advance to pay for it

Branching from any of these queues shares the suspension

Things to remember

- Credits are replaced by debits, representing the cost of suspended computations. Before forcing a suspension, all its debits must be discharged.
- Incremental suspended computations distribute its debits homogeneously, while monolithic ones accumulate all debits at the root.
- Monolithic suspended computations must be created sufficiently in advance. Their results will be shared when used persistently.

Removing amortization

Why remove amortization?

Usually data structures with amortized bounds are simpler and/or faster, but sometimes bounding the time of every operation might be necessary:

- Interactive systems.
- Parallel systems.
- Real-time systems.

Why remove amortization?

Usually data structures with amortized bounds are simpler and/or faster, but sometimes bounding the time of every operation might be necessary:

- Interactive systems.
- Parallel systems.
- Real-time systems.

Warning: non-strict languages like Haskell would usually require the use of `seq` to force specific suspensions, though we will ignore it.

Scheduling

Similar idea to "strategies" from `parallel` Haskell package: decouple the creation of a value from the way it is evaluated.

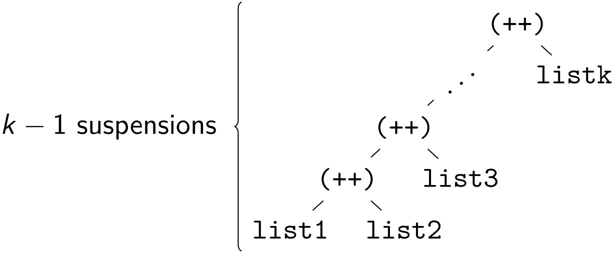
Schedule: additional component of a data structure that forces pieces of a big suspension with every operation. It may require:

- Caution with chains of unevaluated suspensions.
- Transforming monolithic computations into incremental ones.

Chains of unevaluated suspensions

```
 list1 ++ list2
(list1 ++ list2) ++ list3
              ↓
              ⋮
              ↓
(...((list1 ++ list2) ++ list3) ++ ...) ++ listk
```

Chains of unevaluated suspensions



Although each suspension produced by ++ takes $\Theta(1)$ time to force, we need to force $k - 1$ suspensions to obtain the first element of the resulting list. Required time: $\Theta(k)$.

From monolithic to incremental

When performing a rotation (`fs ++ reverse rs`), we know that $|\texttt{fs}| + 1 = |\texttt{rs}|$. Introduce an accumulating parameter (initially empty):

```
rotate    _       [r]    acc  =  y : acc
rotate  (f:fs)  (r:rs)  acc  =  f : rotate fs rs (r : acc)
```

Idea: build the list from the beginning and from the end at the same time.

From monolithic to incremental

When performing a rotation (`fs ++ reverse rs`), we know that $|\texttt{fs}| + 1 = |\texttt{rs}|$. Introduce an accumulating parameter (initially empty):

```
rotate    _       [r]    acc  =  y : acc
rotate  (f:fs)  (r:rs)   acc  =  f : rotate fs rs (r : acc)
```

Idea: build the list from the beginning and from the end at the same time.

Example:

```
rotate [1,2,3] [7,6,5,4] []
1 : (rotate [2,3] [6,5,4] [7])
1 : 2 : (rotate [3] [5,4] [6,7])
1 : 2 : 3 : (rotate [] [4] [5,6,7])
1 : 2 : 3 : 4 : [5,6,7]
```

Real-time queue

`data RealTimeQueue a = RTQ [a] [a] [a]`

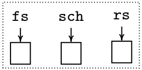
In `RTQ fs rs sch`:

- `sch` contains the last elements of `fs`.
- $|\texttt{sch}| = |\texttt{fs}| - |\texttt{rs}|$.

Function	Cost
`head (RTQ [] _ _) = Nothing` `head (RTQ (x:_) _ _) = Just x`	$\Theta(1)$
`snoc (RTQ fs rs sch) x = exec fs (x:rs) sch`	$\Theta(1)$
`tail (RTQ [] _ _) = Nothing` `tail (RTQ (_:fs) rs sch) = Just (exec fs rs sch)`	$\Theta(1)$

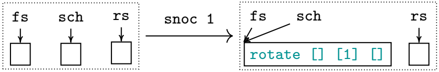
`exec fs rs (_:sch) = RTQ fs rs sch`
`exec fs rs [] = RTQ fs' [] fs'`
`    where fs' = rotate fs rs []`	

Real-time queue: example



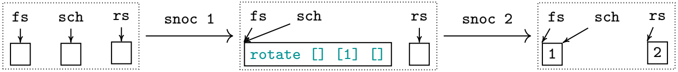
Real-time queue: example

```
snoc (RTQ fs rs sch) x = exec fs (x:rs) sch
exec fs rs (_:sch) = RTQ fs rs sch
exec fs rs [] = RTQ fs' [] fs'
    where fs' = rotate fs rs []
```



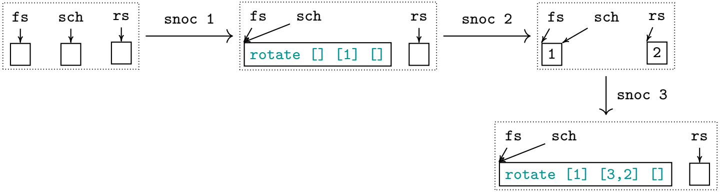
Real-time queue: example

```
snoc (RTQ fs rs sch) x = exec fs (x:rs) sch

exec fs rs (_:sch) = RTQ fs rs sch

exec fs rs [] = RTQ fs' [] fs'

     where fs' = rotate fs rs []
```



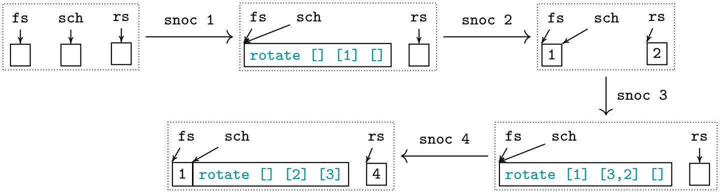
Real-time queue: example

```
snoc (RTQ fs rs sch) x = exec fs (x:rs) sch
exec fs rs (_:sch) = RTQ fs rs sch
exec fs rs [] = RTQ fs' [] fs'
    where fs' = rotate fs rs []
```



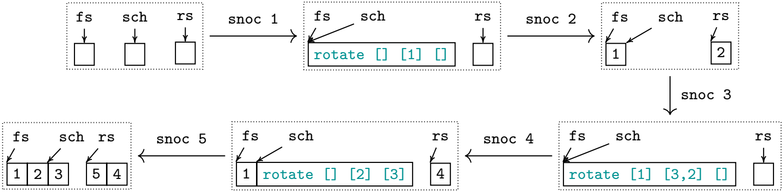
Real-time queue: example

```
snoc (RTQ fs rs sch) x = exec fs (x:rs) sch

exec fs rs (_:sch) = RTQ fs rs sch

exec fs rs [] = RTQ fs' [] fs'

     where fs' = rotate fs rs []
```



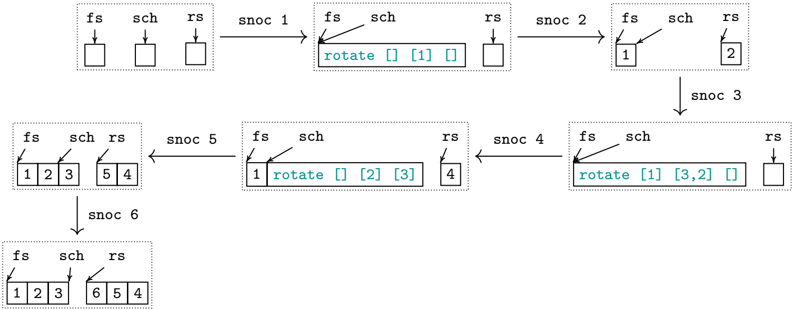
Real-time queue: example

```
snoc (RTQ fs rs sch) x = exec fs (x:rs) sch
exec fs rs (_:sch) = RTQ fs rs sch
exec fs rs [] = RTQ fs' [] fs'
    where fs' = rotate fs rs []
```



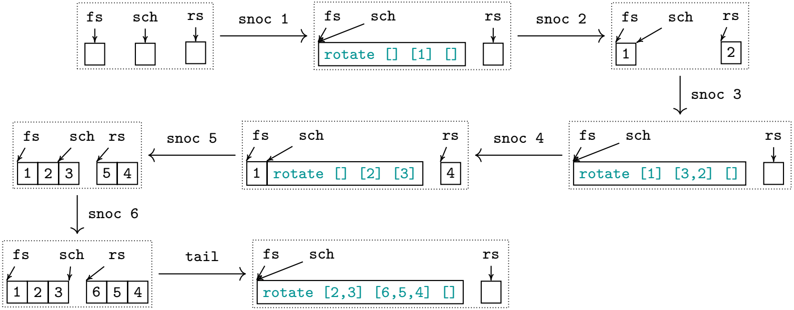
Real-time queue: example

```
snoc (RTQ fs rs sch) x = exec fs (x:rs) sch
exec fs rs (_:sch) = RTQ fs rs sch
exec fs rs [] = RTQ fs' [] fs'
    where fs' = rotate fs rs []
```



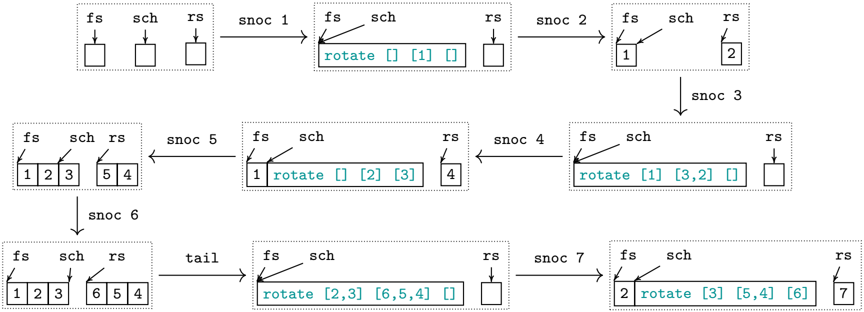
Real-time queue: example

```
tail (RTQ (_:fs) rs sch) = Just (exec fs rs sch)
exec fs rs (_:sch) = RTQ fs rs sch
exec fs rs [] = RTQ fs' [] fs'
    where fs' = rotate fs rs []
```



Real-time queue: example

```
snoc (RTQ fs rs sch) x = exec fs (x:rs) sch

exec fs rs (_:sch) = RTQ fs rs sch

exec fs rs [] = RTQ fs' [] fs'

     where fs' = rotate fs rs []
```



Things to remember

- Avoid chains of unevaluated suspensions.
- Transform monolithic suspensions into incremental ones.
- Extend data structures with a schedule that will traverse them, forcing suspensions at convenient times.

Thanks!