Advanced FP

Functional Data Structures

Daniil Berezun

2023

Us; Contacts

Daniil Berezun

Ekaterina Verbitskaia

- telegram: @DaniilBerezun
- danya.berezun@gmail.com

- telegram: @kajigor
- kajigor@gmail.com

JetBrains Research, Programming Languages and Tools Lab

Telegram: Advanced Functional Programming, CUB fall 2023

Syllabus

- Functional data structures, persistency (C. Okasaki):
  - RB-trees in functional setting
  - Queues in functional setting:
    - Pure Functional Queue
    - Banker's queue
    - Physicist's queue
    - Real-time Queue
- Quality assurance; Property-based testing
- Lazy programming; Profiling and debugging of Haskell programs
- Functional concepts and pearls (R. Bird):
  - Smallest missing number
  - Remove repeats
  - Burrows-Wheeler transform
  - All prefixes
  - Knuth-Morris-Pratt algorithm
  - Puzzels: rush hour and sudoku
  - Hylomorphisms and nexyses
  - Code with no cycles
- Effect systems;Using them for logging, error catching, mutability
- Zippers, type algebra, comonads, and Pearl: Scrap Your Zippers
- New Pearls: More Fixpoints!, monoids and 'vector reverse'
- Functional Design Patterns. Design and implementation of DSLs:
  - GADTs
  - Existential type
  - Rank N types
  - DSL
- Type level programming in Haskell and beyond. Type families, tagless-final
- Template Haskell, Lens
- Interfacing with the real world: working with databases, IORef, concurrency

Today: Reminder about Persistency

Main today's concepts

- *Immutable* data structures
- *Persistent* data structures

Remarks

- We can use *old* nodes (*share*) in new version of the data structure
- Non-persistent data structures are called *ephemeral*

Important remark

During **this lecture** we **do not** assume our language to be **lazy**

Linked List

Definition (Linked List)

Who knows?

Linked List

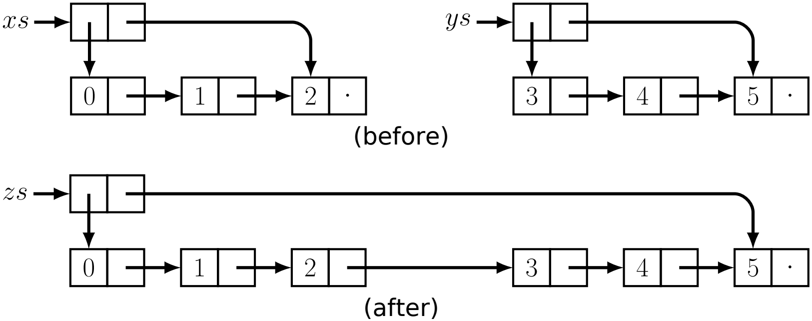
Definition (Linked List)

Who knows?

Definition (List) [One of possible definitions]

A data structure such that from some predefined side (for example, list head) deletion and insertion of element has complexity $O(1)$

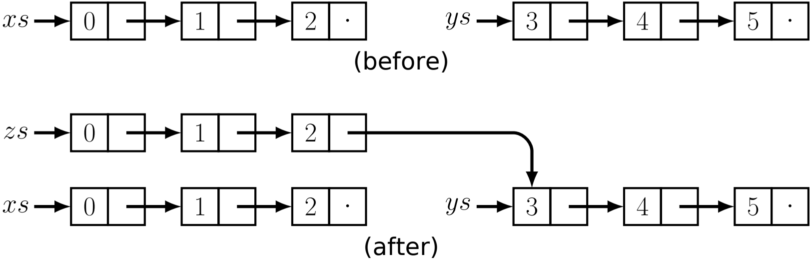
List Concatenation in the Imperative Paradigm



Concatenation of lists xs and ys in the imperative paradigm

- Destroys argument lists xs and ys (one can't use them further)
- Complexity: $O(1)$

Pure Functional Lists Concatenation



Execution of `zs = xs ++ ys` in functional world

- xs and ys remain intact
- we copied **a lot** but the first list only
  - i.e. *persistency* through copying (memory)
  - *shared* parts

How to implement concatenation ++ of lists xs and ys?

- If xs is empty then ys is the answer
- Otherwise xs consists of h as a head and tl as a tail then the answer is a list with head h and tail tl++ys.

Complexity: $O(length(xs))$

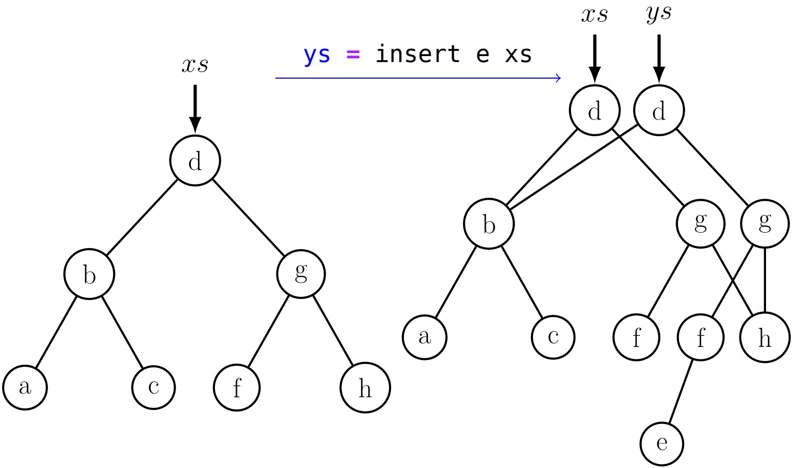
```
(++) []     ys = ys
(++) (h:tl) ys = h : (tl ++ ys)
```

How to update the n-th list element?

```
update []   i y = error "i is greater than list length"
update x:xs 0 y = y : xs
update x:xs i y = x : update xs (i-1) y
```

- $O(n)$ ... very sad ;(
- We copy the element being modified **and** all elements that have direct or indirect pointers to it

Another example: Trees



- Usually, the number of nodes to be copied is at most $\log_2 n$

On Concatenation Associativity

In theory list concatenation is associative

$$(((a_1 +\!\!+ a_2) +\!\!+ a_3) +\!\!+ \ldots +\!\!+ a_n) \equiv (a_1 +\!\!+ (a_2 +\!\!+ (a_3 +\!\!+ (\ldots +\!\!+ a_n))))$$

In practise left-had side is much slower than right-hand side

Note for developers

Sometimes, for an efficient implementation one need to redesign algorithms in a way such that shorter lists are concatenated with longer lists; Ideally, always concatenate one element with a list

Pearl: RB-Trees: definition

```haskell
data Colour = R | B
data Tree a = E | T Colour (Tree a) a (Tree a)
```

Invariants

- ⓿ `T _ a x b` $\Rightarrow \forall i \in a, j \in b. \quad i < x \leq j$
- ❶ No red node has a red parent
- ❷ Every path from the root to an empty node contains the same number of black nodes

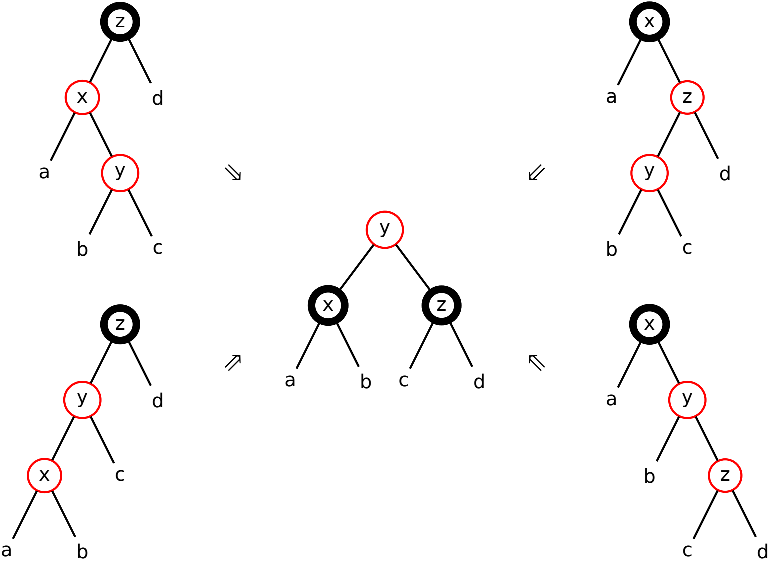
Simple set operations

```haskell
empty = E -- we assume them to be black

member x E = False
member x (T _ a y b)
  | x < y     = member x a
  | x > y     = member x b
  | otherwise = True
```

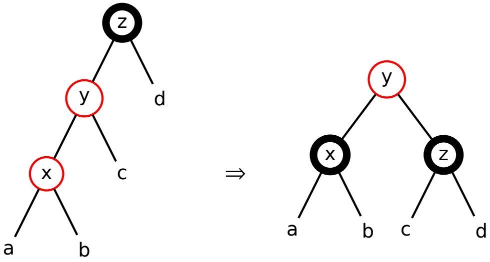
```haskell
insert :: Ord a => a -> Tree a -> Tree a
insert x s = makeBlack (ins s) -- root is always black
  where
    ins E = T R E x E           -- new node is red
    ins s@(T colour a y b)
      | x < y     = balance colour (ins a) y b
      | x > y     = balance colour a       y (ins b)
      | otherwise = s
    makeBlack (T _ a y b) = T B a y b
```

Pearl: RB-Trees: Possible Invariant Violations

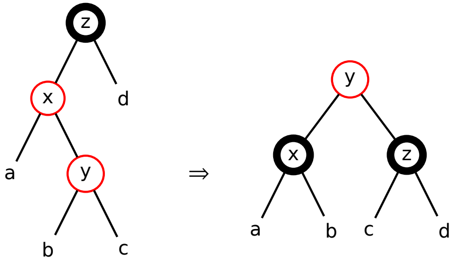


RB-Trees: Balance (1/4)

```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
  = T R (T B a x b) y (T B c z d)
```

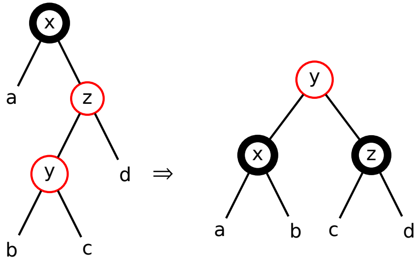


```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R a x (T R b y c)) z d
  = T R (T B a x b) y (T B c z d)
```



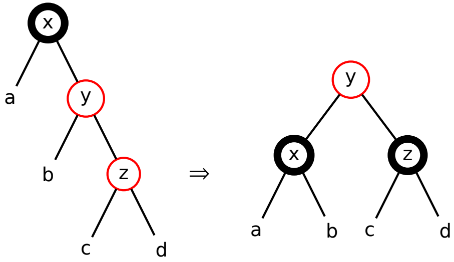
RB-Trees: Balance (3/4)

```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B a x (T R (T R b y c) z d)
  = T R (T B a x b) y (T B c z d)
```



RB-Trees: Balance (4/4)

```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B a x (T R b y (T R c z d))
  = T R (T B a x b) y (T B c z d)
```



Pearl: RB-Trees: All together

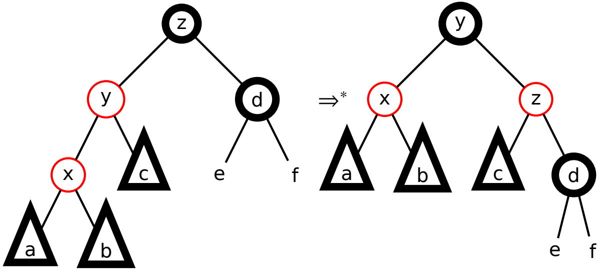
```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
    = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
    = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
    = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
    = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```

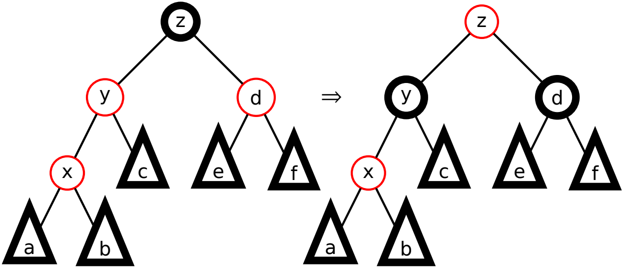
Pearl: RB-Trees: All together

```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
    = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
    = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
    = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
    = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```

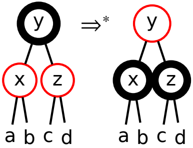
```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
     || B (T R a x (T R b y c)) z d
     || B a x (T R (T R b y c) z d)
     || B a x (T R b y (T R c z d))
    = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```

In imperative world we usually consider more cases (like uncle's colour) trying to *minimize assignes*
While in FP we *construct new tree anyway*

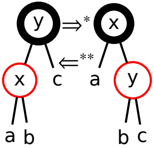




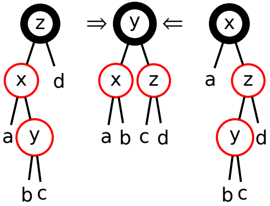
Pearl: RB-Trees: Alternative imperative-like



Colour flip;
* one of $a - d$ red



Single Rotation;
* a red; ** c red



Double Rotation

```haskell
-- color flips
balance B (T R a@(T R _ _ _) x b) y (T R c z d)
     || B (T R a x b@(T R _ _ _)) y (T R c z d)
     || B (T R a x b) y (T R c@(T R _ _ _) z d)
     || B (T R a x b) y (T R c z d@(T R _ _ _))
    = T R (T B a x b) y (T B c z d)
-- single rotations
balance B (T R a@(T R _ _ _) x b) y c = T B a x (T R b y c)
balance B a x (T R b y c@(T R _ _ _)) = T B (T R a x b) y c
-- double rotations
balance B (T R a x (T R b y c)) z d
     || B a x (T R (T R b y c) z d)
    = T B (T R a x b) y (T R c z d)
-- no balancing necessary
balance color a x b = T color a x b
```

Next for Today

- Reminder about *amortized* methods

- Persistent + amortization + laziness on FIFO queue example

Standard complexity notation $O(\cdot)$ -- worst case estimation

But actually, we may have more freedom:

- Let's perform $n+1$ action
- Most of actions will be ``cheap'': $O(1)$
- One ``expensive'' action: for example, $O(n)$
- Standard assymptotic compexity: $O(n)$
- Average complexity of performing n actions (*amortized time complexity*) can be $O(1)$ for an action

$$a = \frac{\sum_{i=1}^{n} t_i}{n}$$

This additional freedom degree sometimes allows a simpler and more efficient implementation to be designed

Definition (*Accumulated Savings*)

A difference between total current amortized cost and total current fair value

- **NB:** accumulated savings must be **non-negative**
- I.e. ``expensive'' operations may take place iff accumulated savings are enough to cover theis additional cost

$$a_i = t_i + c_i - \bar{c}_i$$

where t_i --- fair cost, c_i --- credit amount allocated by action i, $\bar{c}_i$ --- amount of credit spent by action i

- Each credit unit must be allocated before being spent
- Credit cannot be used twice
- $\sum c_i \geq \sum \bar{c}_i \Rightarrow \sum a_i \geq \sum t_i$
- Amortized compexity is $n * O(f(n,m)) \Leftrightarrow \forall n. a_i = O(f(n,m))$ $\Rightarrow a = \frac{\sum_{i=1}^{n} a_i}{n} = \frac{n * O(f(n,m))}{n} = O(f(n,m))$

$$\Phi : Object(d) \rightarrow Potential$$

Usually, initial potential is zero and is always non-negative

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

$$\begin{aligned} \sum_{i=1}^{j} t_i &= \sum_{i=1}^{j}(a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \sum_{i=1}^{j}(\Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j) \end{aligned}$$

TODO: diff between methods

Pure Functional Queues

Interface:

- `empty: queue -> bool`
- `enqueue: queue * int -> queue`
- `head: queue -> int`
- `tail: queue -> queue`

Pure Functional Queues

Interface:

- `empty: queue -> bool`
- `enqueue: queue * int -> queue`
- `head: queue -> int`
- `tail: queue -> queue`

Simplest implementation

Via a pair of lists, `f` and `r`

- `f` (front) contains the head elements of the queue in the initial (correct) order,
- `r` (reversed) consist of tail elements in reverse order

For example, queue `=[1;2;3;4;5;6]` can be represented as two lists `f=[1;2;3]` and `r=[6;5;4]`

Question: When to move elements from the front to reversed list?

Batched (Naïve) Queue

Definition (Queue Invariant)

List f may become empty iff list r is also empty (i.e., the queue is empty)

```
enqueue ([], _) x = ([x], [] )   --- O(1)
enqueue (f , r) x = (f   , x:r)  --- O(1)
tail ([x], r) = (rev r, [])      --- O(n); O*(1)
tail (x:f, r) = (f     , r )     --- O(1)
```

Thus, head and enqueue are $O(1)$ instead of $O(n)$,
but tail is still $O(n)$

Definition (Invariant)

Each element in the *tail* list is associated with one credit unit

- Each `enqueue` call performs the only real computational step and emits *additional* credit unit for an element in the tail list
  amortized complexity is 2

- `tail`, if no list inversion happend, preforms one step and spends no credit units
  amortized complexity is 1

- `tail`, if list reverse happends, performs $(m + 1)$ steps, where m is a tail list length, and spends m credit units
  amortized complexity is $m + 1 - m = 1$

Definition (Invariant)

Each element in the *tail* list is associated with one credit unit

- Each `enqueue` call performs the only real computational step and emits *additional* credit unit for an element in the tail list
  amortized complexity is 2
- `tail`, if no list inversion happend, preforms one step and spends no credit units
  amortized complexity is 1
- `tail`, if list reverse happends, performs $(m+1)$ steps, where m is a tail list length, and spends m credit units
  amortized complexity is $m+1-m=1$

Physicist's method: just let Φ be rear list length

Conclusion

- In case of purely functional queue, function `tail` worst case complexity is $O(n)$ and amortized --- $O(1)$
- Good if one do not need persistency, and amortized performance is good enough for the problem
- Bad news: we have implicitly assumed that we use queues *ephemerally* i.e. in *persistent setting* still $O(n)$
- (Next) Lazy evaluations $+$ amortized compexity $=$ persistent queues with a very good amortized complexity

Lazy Evaluations

Delays the evaluation of an expression until its value is needed (*non-strict evaluation*)

ML

```
type α sup
val delay : (unit -> α) -> α sup
val force : α sup -> α
```

We will use $ denote that evaluation is suspended

Lazy Evaluations

Delays the evaluation of an expression until its value is needed (*non-strict evaluation*)

ML

```
type α sup
val delay : (unit -> α) -> α sup
val force : α sup -> α
```

We will use $ denote that evaluation is suspended

Memoization of lazy evaluations

Ones the value of expression is needed, evaluate it and *memoize* (remember, *sharing*) the result; If it will be needed further, just return the memoized result

Definition (Stream)

is a list but evaluations of sublists are delayed

Example: Stream of all possible natural numbers

Lazy Lists (Streams)

Definition (Stream)

is a list but evaluations of sublists are delayed

Example: Stream of all possible natural numbers

Notation

Add an element x to the tail xs: $\$Cons\ x\ xs$
Empty stream: $\$Nil$
Delay f: $\$f$

Remark

Stream may be both finite and infinite;
One never knows until the end appears

```
datatype α SteamCell = Nil | Cons of α × α Stream
withtype α Stream = α StreamCell susp

(* Stream example *)
$(Cons(1, $Cons(2, $Cons(3, $Nil))))

(* monolitic append aka. suspended list *)
fun s ++ t = $(force s @ force t)

(* incremental append aka. stream *)
fun s ++ t = $case s of
    $Nil => force t
  | $Cons (x, s') => Cons (x, s' ++ t)
```

Example: Fibonacci Numbers

Consider function `zip` $: \mathit{stream} \times \mathit{stream} \rightarrow \mathit{stream}$, which sums streams element by element

A stream of Fibonacco numbers:

$$\mathit{fibs} \equiv \texttt{\$Cons(1,\$Cons(1,zip(fibs,tail(fibs))))}$$

Example: Fibonacci Numbers

Consider function `zip` : *stream* × *stream* → *stream*, which sums streams element by element

A stream of Fibonacco numbers:

fibs ≡ `$Cons(1,$Cons(1,zip(fibs,tail(fibs))))`

1	1			

Example: Fibonacci Numbers

Consider function `zip` : *stream* × *stream* → *stream*, which sums streams element by element

A stream of Fibonacco numbers:

$$fibs \equiv \texttt{\$Cons(1,\$Cons(1,zip(fibs,tail(fibs))))}$$

1	1			
1				

Example: Fibonacci Numbers

Consider function `zip` : *stream* $\times$ *stream* $\to$ *stream*, which sums streams element by element

A stream of Fibonacco numbers:

$$\mathit{fibs} \equiv \texttt{\$Cons(1,\$Cons(1,zip(fibs,tail(fibs))))}$$

1	1	2		
1				

Consider function `zip` $: \mathit{stream} \times \mathit{stream} \to \mathit{stream}$, which sums streams element by element

A stream of Fibonacco numbers:

$$\mathit{fibs} \equiv \texttt{\$Cons(1,\$Cons(1,zip(fibs,tail(fibs))))}$$

1	1	2		
1	2			

Example: Fibonacci Numbers

Consider function `zip` : *stream* × *stream* → *stream*, which sums streams element by element

A stream of Fibonacco numbers:

fibs ≡ `$Cons(1,$Cons(1,zip(fibs,tail(fibs))))`

1	1	2	3	
1	2			

and so on

Remark

This implementation has amortized complexity $O(1)$ and is persistent

1. Use streams instead of lists
2. Store stream lengths explicitly
3. Invariant: $|\mathtt{f}| > |\mathtt{r}|$

If streams f and r have the same length, define f as $\mathtt{f} \mathbin{+\!\!+} \mathit{reverse}(\mathtt{r})$

Why rotate when $|f| \simeq |r|$?

- *reverse* is monolithic, hence, it needs $|r|$ steps
- suspended *reverse* can be forced only after $|f|$ calls of *tail*
- *debits* instead credits:

each *tail* call discharges one debit

Why is it persistent?

- Lazy (suspended) evaluation ⇒ delayed until needed
- Memoization ⇒ each suspension computes only ones

Persistent Physicist's Queue

- Ψ maps object with potential representing an upper bound on the accumulated debt $\Psi = min(2|W|, |F| - |R|)$
- We need only *monolithic suspensions*: use suspended list instead of stream
- Front devided into two parts:
  - strict non-empty prefix
  - suspended front itself --- the only suspended part

```
datatype α Queue = Queue of
  {W : α list, F : α list susp, LenF : int, R : α list, LenR : int}
```

```
(* invariante 1: W is no-empty whenever F is non-empty *)
fun checkW {W = [], F = f, LenF = lenF, R = r, LenR = lenR}) =
    Queue {W = force f, F = f, LenF = lenF, R = r, LenR = lenR})
  | checkW q = Queue q
(* invariante 2: R is no longer than F *)
fun checkR (q as {W = w, F = f, LenF = lenF, R = r, LenR = lenR}) =
  if lenR <= lenF then q
  else let val w' = force f
      in {W = w', F= $(w' @ rev r), LenF = lenF+lenR, R=[], LenR=0} end
(* check invariants *)
fun inv-queue q = checkW (checkR q)
```

```
fun enqueue (Queue {W = w, F = f, LenF = lenF, R = r, LenR = lenR}, x) =
  inv-queue {W = w, F = f, LenF = lenF, R = x::r, LenR = lenR+1}
fun head (Queue {W = x::w, ...}) = x
fun tail (Queue {W = x::w, F = f, LenF = lenF, R = r, LenR = lenR}) =
  inv-queue {W = w,F = $tl (force f), LenF=lenF-1, R=r, LenR=lenR})
```

Amortization no more: Scheduling

Problem Statement

- We produce n ``cheap'' steps
- Then, one ``expensive'' step $O(n)$
- Thus, we can only state amortized complexity

An Idea: *scheduling*

Instead of one ``expensive'' step let's perform $n$ smaller steps with constant complexity. Performing each ``cheap'' step, we will also perform one of this ``smaller'' steps.

Reminder: banker's queue: we relied on calculation of
$\texttt{f} +\!\!+ \textit{reverse}(\texttt{r})$

Now let's instead use a special function `rotate`

$$\texttt{rotate}(\texttt{f}, \texttt{r}, a) = \texttt{f} +\!\!+ \textit{reverse}(\texttt{r}) +\!\!+ a$$

Third parameter is an accomulator which stores partially computed result of $\textit{reverse}(\texttt{r})$

Obviously

$$\textit{rotate}(\texttt{f}, \texttt{r}, \$\textit{Nil}) = \texttt{f} +\!\!+ \textit{reverse}(\texttt{r})$$

Let's reorder queue when $|\mathtt{r}| = |\mathtt{f}| + 1$
This ratio will be maintained throughout the rebuilding

Let's prove it by induction on the length of front $|\mathtt{f}|$

Base:

$$\begin{aligned} \mathit{rotate}(\texttt{\$Nil}, \texttt{\$Cons(y,\$Nil)}, a) &\equiv \texttt{\$Nil} +\!\!+ \mathit{reverse}(\texttt{\$Cons(y,\$Nil)}) +\!\!+ a \\ &\equiv \texttt{\$Cons(y,a)} \end{aligned}$$

Induction step:

$$\begin{aligned} &\mathit{rotate}(\texttt{\$Cons(x,f)}, \texttt{\$Cons(y,r)}, a) \\ &\quad \equiv \texttt{\$Cons(x,f)} +\!\!+ \mathit{reverse}(\texttt{\$Cons(y,r)}) +\!\!+ a \\ &\quad \equiv \texttt{\$Cons(x,f} +\!\!+ \texttt{reverse(\$Cons(y,r))} +\!\!+ \texttt{a)} \\ &\quad \equiv \texttt{\$Cons(x,f} +\!\!+ \texttt{reverse(r)} +\!\!+ \texttt{\$Cons(y,a))} \\ &\quad \equiv \texttt{\$Cons(x,rotate (f, r, \$Cons(y,a)))} \end{aligned}$$

Real-time Queue: code

```
(* S is a schedule -- suffix s.t. all nodes before S in F have already
been forced and !memoized! *)
datatype Queue = Queue of {F : α stream, R : α list, S : α stream}

val empty = Queue {F = $Nil, R = [], S = $Nil}
fun isEmpty (Queue {F = f , ... }) = null f

fun rotate (f, r, a) = $case (f, r) of
  ($Nil          , $Cons (y, _ )) => Cons (y, a)
| ($Cons (x, f'), $Cons (y, r')) => Cons (x, rotate (f',r',$Cons (y, a)))

(* Invariant: |S| = |F| - |R| *)
fun inv {F = f, R = r, S = $Cons (x, s)} = Queue {F = f, R = r, S = s}
  | inv {F = f, R = r, S = $Nil         } =
    let val f' = rotate (f, r, $Nil)      (* create a suspension      *)
    in Queue {F = f', R = [], S = f'} end (* store it in both F and S *)

fun enqueue (Queue {F=f, R=r, S=s}, x) = inv {F = f, R = x::r, S = s}

fun head (Queue {F = $Cons (x, f), ...}) = x

fun tail (Queue {F = $Cons (x, f), R=r , S=s}) = inv {F=f, R=r, S=s}
```

Conclusion

Queue\ Operation	enqueue	head	tail
Banker's	O(1)*	O(1)*	O(1)*
Physicist's	O(1)*	O(1)*	O(1)*
Real-time	O(1)	O(1)	O(1)

Amortized estimations are marked with *

Questions?