# 20分でわかる Purely Functional Data Structures

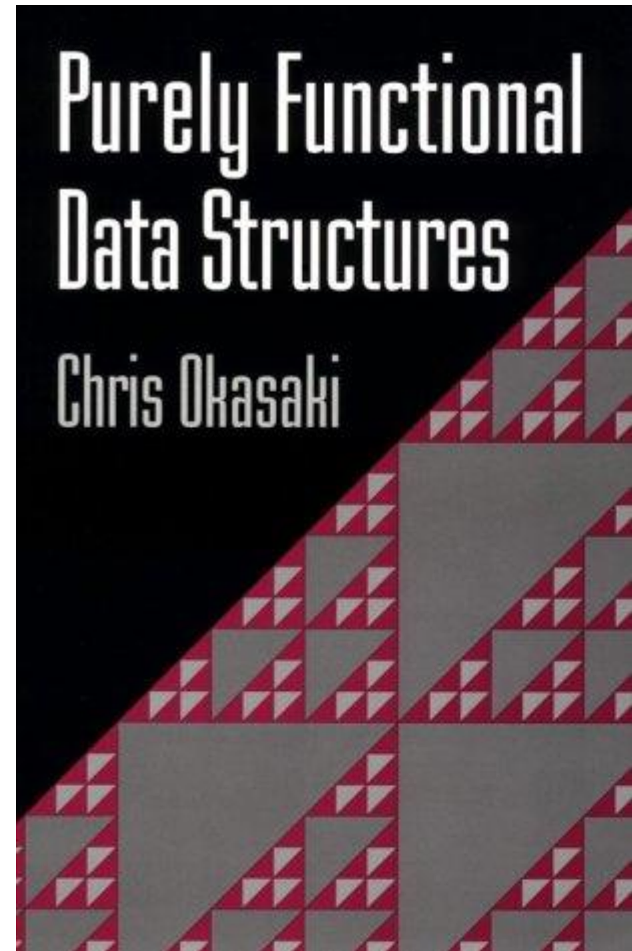
k.inaba (http://www.kmonos.net/)

Apr. 4, 2010

# あらすじ

# Immutable Object だけで作るデータ構造

この本の
内　　容を
全速力で
布教する



# お題：キュー (Queue)

- FIFO (First-In First-Out)
- pushBack(e) でデータeを入れる
- popFront() で取り出せる
- 入れた順に出てくる
- 以上

Immutable Object でない

打倒すべき目標

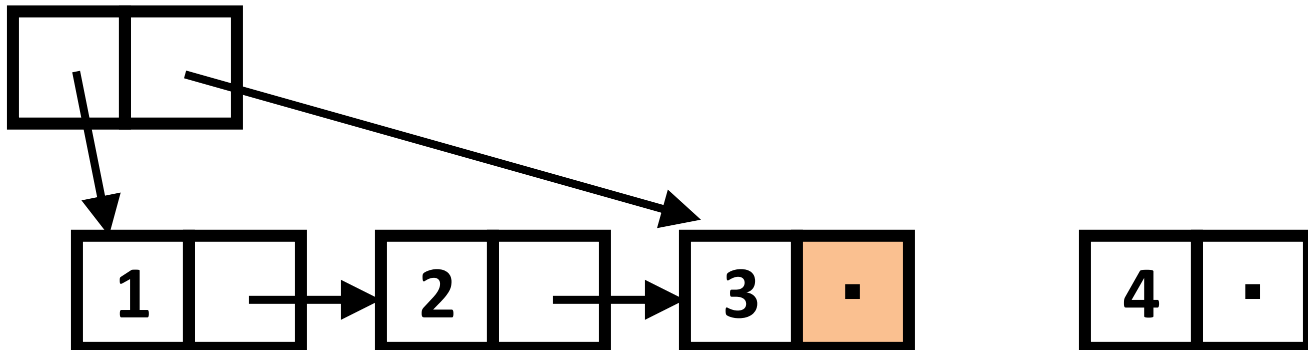
# 破壊的キュー

# 手続き型でよくある

```
interface Queue<E>
{
    void pushBack(E e);
    E    popFront();
}
```

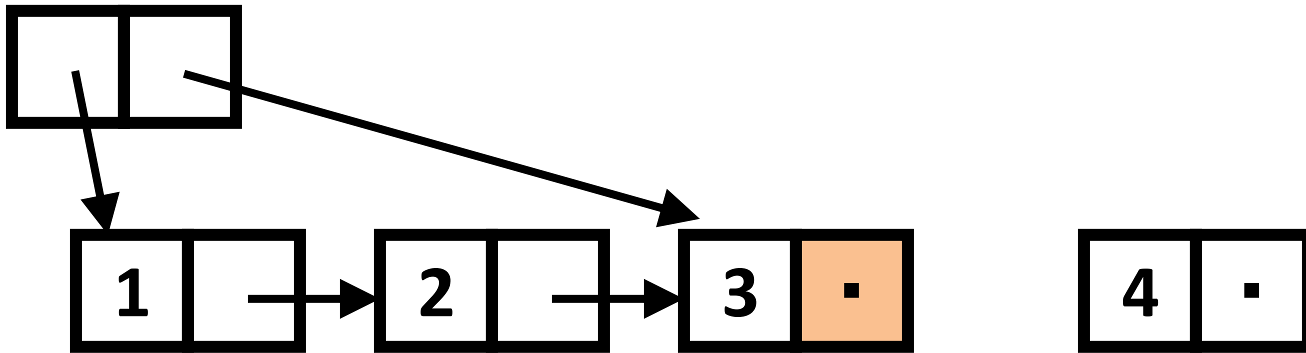
# よくある実装

```
class HakaiQueue<E> implements Queue<E> {
   class Cell { E e; Cell next; }
   Cell fst, last;

   void pushBack(E e)
     {last=last.next=new Cell(e,null);}
   E popFront()
     {E e=fst.e; fst=fst.next; return e;}
}
```



# 破壊的キューの特徴

- Mutable Object を使用
  - リスト末尾を指すポインタをもっておき末尾のセルを pushBack 時に書換




- Persistent でない
  - 操作前の状態をとっておくには全コピーしかない

```
HakaiQueue<E> q = 略;
HakaiQueue<E> p = q;
q.pushBack(e); //pも変化!
```

- pushBack, popFront の最悪実行時間は O(1)

# 計算量

Ephemeral (儚い)
使い方での計算量

| | | (比較対象)<br>破壊的 | 2リスト | 銀行家 | 実時間 |
|---|---|---|---|---|---|
| 儚 | A | O(1) | | | |
| | W | O(1) | | | |
| 永 | A | n/a | | | |
| | W | n/a | | | |

Amortized(償却)計算量

詳しくは
あとで

Worst-Case(最悪)計算量

Persistent (永続的な)
使い方での計算量

Immutable な実装

# 2リストキュー

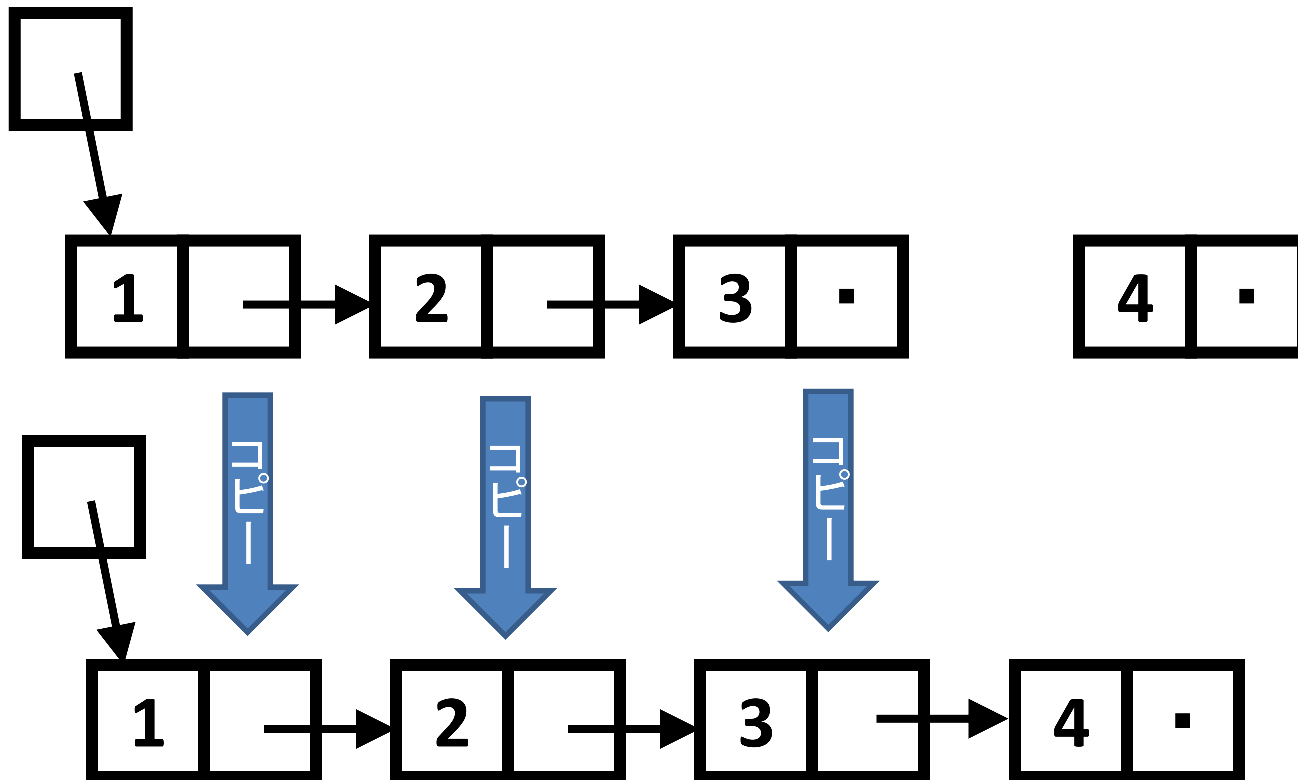
# 非破壊的キュー

- フィールドの書き換えを使わないキュー
- pushBack, popFront は「操作後のキュー」を別のオブジェクトを作って返す

```
interface ImuQueue<E>
{
   ImuQueue<E>   pushBack(E e);
   Pair<E,
    ImuQueue<E>> popFront();
}
```

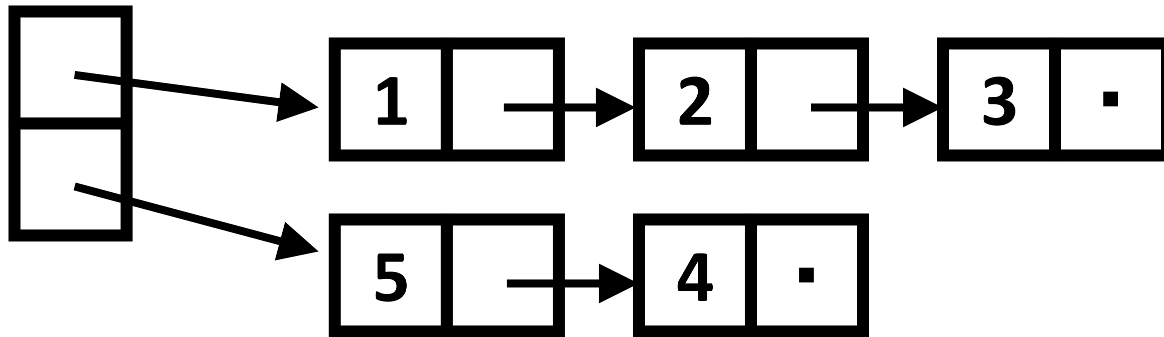
# 非破壊キュー

- 単純にやると全コピー：計算量 O(n)



# ２リストキュー

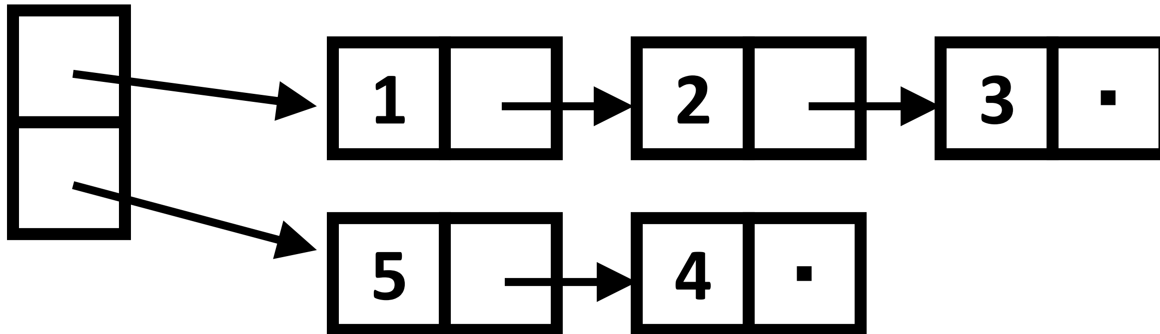
- ちょっと工夫
- キューの後ろの方は逆順で持つ
  - pushBack がリスト先頭への追加なのでO(1)に！




```
data Queue a = Q [a] [a]  --ここからコードはHaskellです

pushBack (Q front rear) e = Q front (e:rear)
popFront (Q []     r) = popFront (Q (reverse r) [])
popFront (Q (e:f) r) = (e, Q f r)
```

# ２リストキューの特徴

- front側とrear側の２つのリストで表現
- len(front) == 0 になったら rear を reverse




- Persistent である
- 最悪実行時間は、reverseが発生する瞬間 O(n)
- でも、償却実行時間は O(1)
  - なので、トータルの実行時間的に考えると計算時間は O(1) と思って問題ない！

# 償却計算量とは？

- pushBack 1 → [] [1]
- pushBack 2 → [] [2,1]
- pushBack 3 → [] [3,2,1]
- popFront → [1,2,3] [] → [2,3] []
- popFront → [3] []
- pushBack 4 → [3] [4]
- popFront → [] [4]
- popFront → [4] [] → [] []

操作列全体でコストを平均化して見たときの計算量

reverseは「たまにしか」起きない

**時間 t かかるreverseが発生する前に、必ず t 回 pushBack している**

# 償却計算量とは？

**時間 t かかるreverseが発生する前に、必ず t 回 pushBack している**

| 現実の計算量 | |
|---|---|
| pushBack | 1 |
| popFront(軽) | 1 |
| popFront (重) | t+1 |

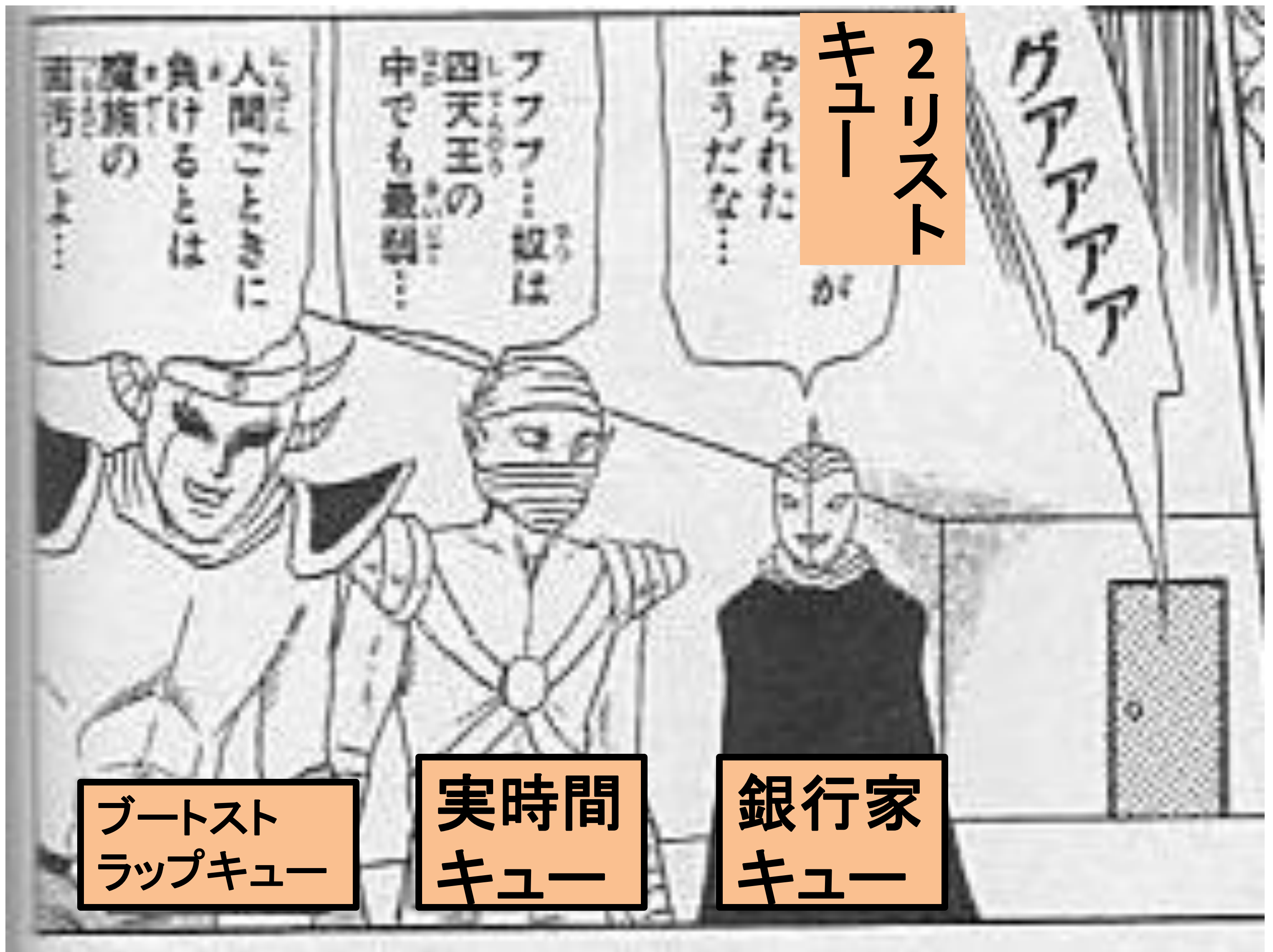
| 分担をごまかした計算量 | |
|---|---|
| pushBack | 2 |
| popFront(軽) | 1 |
| popFront (重) | 1 |

| 操作 | 状態 | 現実 | 分担 |
|---|---|---|---|
| pushBack 1 | → [] [1] | 1 | 2 |
| pushBack 2 | → [] [2,1] | 1 | 2 |
| pushBack 3 | → [] [3,2,1] | 1 | 2 |
| popFront | → [1,2,3] [] → [2,3] [] | 3+1 | 1 |
| popFront | → [3] [] | 1 | 1 |
| pushBack 4 | → [3] [4] | 1 | 2 |
| popFront | → [] [4] | 1 | 1 |
| popFront | → [4] [] → [] [] | 1+1 | 1 |
| **合計** | | **12** | **12** |

こう思えば
全部 O(1) だし
トータル計算量も
変わらないので
問題ない！

# 計算量

| | | (比較対象)<br>破壊的 | 2リスト | 銀行家 | 実時間 |
|---|---|---|---|---|---|
| 儚 | A | O(1) | O(1) | | |
| | W | O(1) | O(n) | | |
| 永 | A | n/a | O(n) | | |
| | W | n/a | O(n) | | |




# 銀行家キュー

# ２リストキューは本当に

- Persistentと言えるのか？？？
  - Persistentなら、同じバージョンを取っておいて何回も使えるはず

```
let q = loadSomeQueue () in
  … (doHoge q) … (doFuga q) …
  (doPiyo q) … (doHige q) …
```

# 破滅的な例

- reverseが起きる寸前のキューを何回も使う

```
let q = needReverseQueue () in
  … (popFront q) … (popFront q) …
  (popFront q) … (popFront q) …
```

# ごまかしきれてない

- reverseが起き る寸前のキューを何回も使う

```
let q = pushFront (pushFront (pushFront
          (Queue [] []) 1) 2) 3
in … (popFront q) … (popFront q) …
   (popFront q) … (popFront q) …
```

実際の計算時間は
1*3+4*4 = 19 ！

計算時間は 2*3+4 = 10 です！

現実の計算量

| | |
|---|---|
| pushBack | 1 |
| popFront(軽) | 1 |
| popFront (重) | 1+t |

分担をごまかした計算量

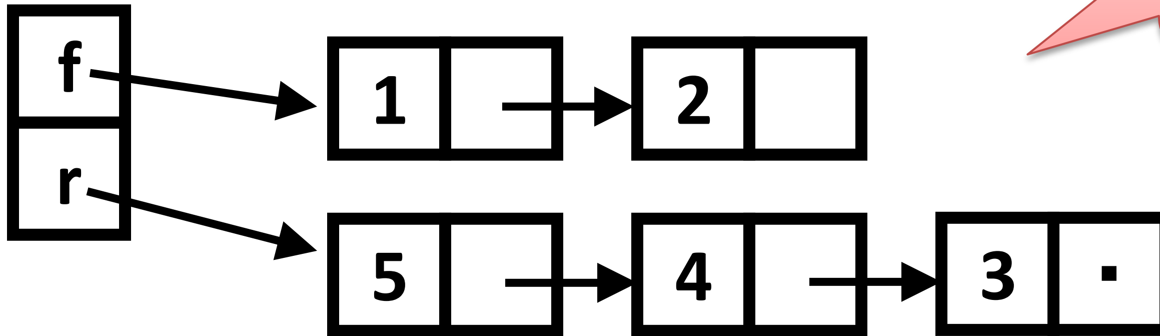
| | |
|---|---|
| pushBack | 2 |
| popFront(軽) | 1 |
| popFront (重) | 1 |

# どうしましょう

- Persistent な使い方（同じバージョンを何回も使い回す）をしても償却計算量を O(1)にするには？

# さらなる工夫

- ~~len(front)==0 になったら reverse~~
- len(front)+1 == len(rear) になったら ***遅延評価で***reverse






```
data Queue a = Q [a] Int [a] Int

pushBack(Q f fl r rl) e  = chk (Q f fl (e:r) (rl+1))
popFront(Q (e:f) fl r rl)= (e, chk (Q f (fl-1) r rl))
```

# さらなる工夫

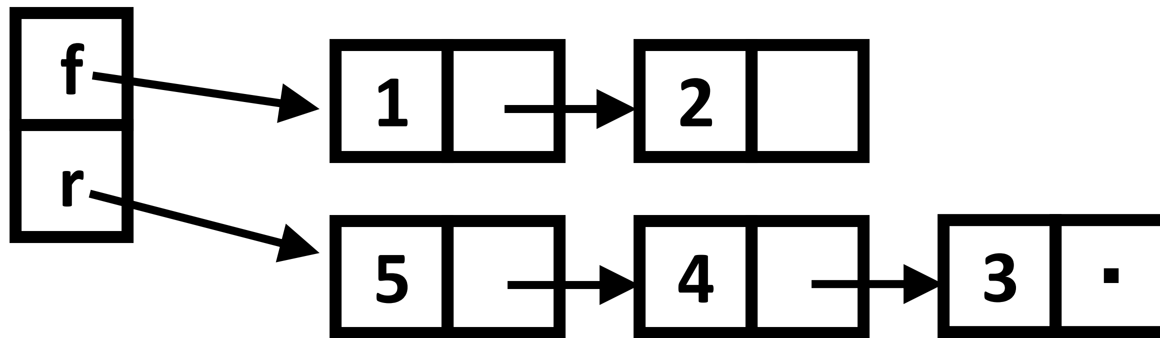
- len(front)+1 == len(rear) *になったら*
  *遅延評価で*reverse

```
data Queue a = Q [a] Int [a] Int

pushBack(Q f fl r rl) e  = chk (Q f fl (e:r) (rl+1))
popFront(Q (e:f) fl r rl)= (e, chk (Q f (fl-1) r rl))

chk (Q f fl r rl) =
  if fl+1 == rl then (Q (f++reverse r) (fl+rl) [] 0)
                else (Q f fl r rl)
```

Haskellなので
デフォルト遅延

# 特徴

- front側とrear側の２つのリストで表現
  - len(front)+1 == len(rear) になったら遅延 reverse




- Persistent である
- 最悪実行時間は、reverseが発生するとき O(n)
- 償却実行時間は (persistentな使い方でも) O(1)

# 計算量の見積もり方

## 積み立て金メソッド

「時間tかかるreverseの前に必ずt回pushBackしてる」
→ 「pushBackのコストを２にして、先にreverseの負担の分を払っておけばOK」

## から 借金メソッド へ

「早めの遅延評価reverseしておけば、値が本当に必要になるまでに時間がある」
→ 「本当に必要になる*支払期限*までに、t 回のpopFrontで負担金を用意できれば問題ない」

# 償却計算量とは？

**長さtのreverse結果が必要となるまでにt回はpopFrontするはず**

現実の計算量
pushBack 1
popFront(軽) 1
popFront (重) x+1

分担をごまかした計算量
pushBack 1
popFront 2

| 操作 | 状態 | 現実 | 償却 |
|---|---|---|---|
| pushBack 1 | → [] [1] → []r[1] [] | 1 | 1 |
| pushBack 2 | → []r[1] [2] | 1 | 1 |
| pushBack 3 | → []r[1] [3,2] → []r[1]r[3,2] [] | 1 | 1 |
| popFront | → r[1]実行 → r[3,2] [] | 1+1 | 2 |
| popFront | → r[3,2]実行 → [3] [] | 2+1 | 2 |
| pushBack 4 | → [3] [4] | 1 | 1 |
| popFront | → [] [4] → []r[4] [] | 1 | 2 |
| popFront | → r[4]実行 → [] [] | 1+1 | 2 |
| **合計** | | **12** | **12** |

# 少し大きい例：pushBack 1～15

- []　　　　　[1] → []r[1] []
- []r[1]　　　[2]
- []r[1]　　　[3,2] → []r[1]r[3,2] []
- []r[1]r[3,2] [4]
- []r[1]r[3,2] [5,4]
- []r[1]r[3,2] [6,5,4]
- []r[1]r[3,2] [7,6,5,4] → []r[1]r[3,2]r[7..4] []
- …
- []r[1]r[3,2]r[7..4] [15..8] → []r[1]r[3,2]r[7..4]r[15..8] []

# popFront

- [] [1] → []r[1] []
- []r[1] [3,2] → []r[1]r[3,2] []
- []r[1]r[3,2] [7,6,5,4] → []r[1]r[3,2]r[7..4] []
- []r[1]r[3,2]r[7..4] [15..8] → []r[1]r[3,2]r[7..4]r[15..8] []
- popFront: r[1]実行 → r[3,2]r[7..4]r[15..8] 1+1
- popFront: r[3,2]実行 → [3]r[7..4]r[15..8] 1+1
- popFront: r[7..4]r[15..8] 1+1
- popFront: r[7..4]実行 → [5,6,7]r[15..8] 1+1
- popFront: [6,7]r[15..8] 1+1
- popFront: [7]r[15..8] 1+1
- popFront: r[15..8] 1+1
- popFront: r[15..8]実行 → [9..15] 1+1

借金返済
間に合ってる

# なぜ借金メソッド？

- **積立金**は一度使うとなくなるけど

```
let q = pushFront (pushFront (pushFront
        (Queue [] []) 1) 2) 3
in … (popFront q) … (popFront q) …
   (popFront q) … (popFront q)
```

3億円貯金

revに
3億円使う

revに3億円使う

revに3億円使う

revに3億円使う

# なぜ借金メソッド？

- **借金**は、一度返せば大丈夫！
  - *遅延評価*で*メモ化*されてるから

```
let q = []r[1]r[3,2]r[7..4][15..8] []
in … (popFront q) … (popFront q) …
 (popFront q) … (popFront q) …
```

15億円借金

rev実行
返済

メモ化
されてるので
もうrev不要

メモ化
されてるので
もうrev不要

メモ化
されてるので
もうrev不要

# 計算量

| | | (比較対象)<br>破壊的 | 2リスト | 銀行家 | 実時間 |
|---|---|---|---|---|---|
| 儚 | A | O(1) | O(1) | O(1) | |
| | W | O(1) | O(n) | O(n) | |
| 永 | A | n/a | O(n) | O(1) | |
| | W | n/a | O(n) | O(n) | |

# ※注釈

- 銀行家キューという名前はなんですか
  - 償却計算量の評価の方法として（Functional データ構造に限らず一般の話として）
    - 銀行家 (Banker) の方法
    - 物理学者 (Phyisicist) の方法
  - の二つがあって、その「銀行家の方法」で設計したキューという意味だそうです。

- 本には両方の手法が紹介されています

# 実時間キュー

## …

- 償却計算量とはなんだったのか

分担をごまかした計算量

| | |
|---|---|
| pushBack | 2 |
| popFront(軽) | 1 |
| popFront (重) | 1 |

**こう思えば**

全部 O(1) だしトータル計算量も変わらないので問題ない！

- **仮想的に**「積立金」や「借金」を考え**仮想的に**返済する

# 仮想世界を現実にする

popFront で、仮想的にではなく

実際に「借金」を返す

＝

popFront のたびに、reverse 処理を

実際に「ちょっとずつ実行」する

# やりかた

１：借金ポインターを追加
（遅延評価サンクを指しておく）

```
data Queue a = Q [a] Int [a] Int [a]

pushBack(Q f fl r rl s) e
  = seq chk (Q f fl (e:r) (rl+1) s)
popFront(Q (e:f) fl r rl s)
  = (e, seq chk (Q f (fl-1) r rl s))
```

２：chk 関数はreverseチェックのついでに
無駄に遅延サンクをちょっとづつ実行
（chk自体は遅延しないようにeager実行）

# やりかた

2.1：借金ポインタに対してパターンマッチ
　＝ consセル１個分だけ実行される

```
chk (Q f fl r rl (_:s)) = (Q f fl r rl s)
chk (Q f fl r rl []) =
  -- 実は fl+1 == rl のときだけこっちに来る
  let ff = rotate f r [] in
    (Q ff (fl+rl) [] 0 ff)
```

2.2：rotate xs ys zs は xs++rev ys++zs する関数

```
rotate [] (y:_) zs         = y : zs
rotate (x:xs) (y:ys) ff =
   x : rotate xs ys (y:zs)
```

「セル1個だけ実行」
しやすいreverse

# 計算量

| | | (比較対象)<br>破壊的 | 2リスト | 銀行家 | 実時間 |
|---|---|---|---|---|---|
| 儚 | A | O(1) | O(1) | O(1) | O(1) |
| | W | O(1) | O(n) | O(n) | O(1) |
| 永 | A | n/a | O(n) | O(1) | O(1) |
| | W | n/a | O(n) | O(n) | O(1) |

# そのほかの話題

ニューメリカル表現の

多相再帰が

再帰のスローダウンが

データ構造を

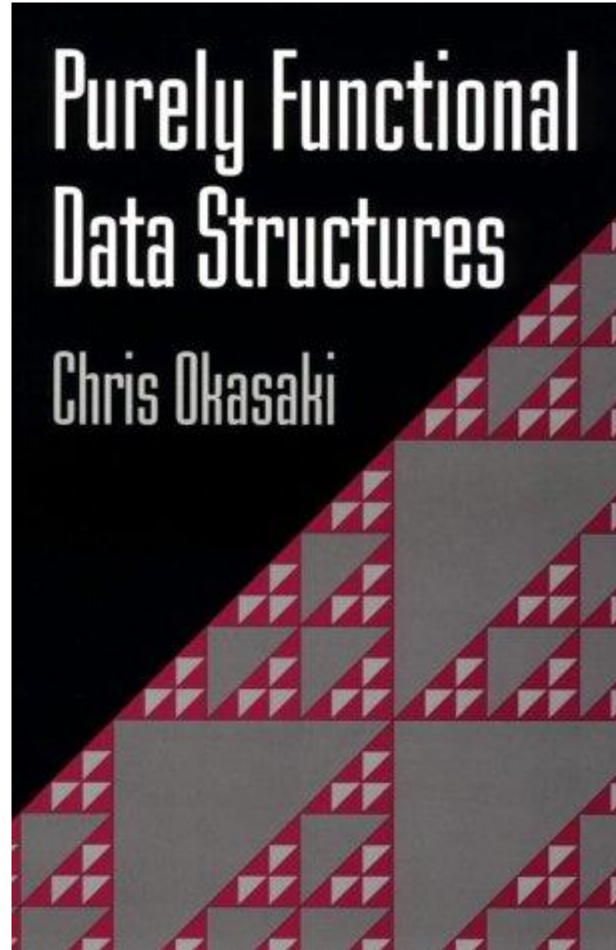
ブートストラップ

# 目次（紹介した部分）

- 2〜3章 :: 簡単な関数型データ構造の紹介
  - 2リストキュー、赤黒木、二項ヒープ、…
- 4章 :: 遅延評価とは
- 5章 :: 償却計算量とは
- 6章:: 遅延評価を駆使して、永続性と償却計算量を両立（キュー, ヒープ, …）
- 7章:: リアルタイム化（キュー, ヒープ, …）
- 8〜11章 :: 関数型データ構造汎用技法集
  - 「n進数表記に学ぶ」「ブートストラップ」「再帰スローダウン」

# あとは

みんな




を読もう！

# THANK YOU FOR LISTENING!

※スライド内の漫画はすべて
増田こうすけ「増田こうすけ劇場 ギャグマンガ日和（巻の5）」からの引用です。