Estruturas de dados puramente funcionais

Dia 2

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Outline

Heaps

Heap - O que é?

UFABC

- Conjuntos ou dicionários (como o que implementamos com as árvores binárias) tipicamente permitem que tenhamos acessos a elementos arbitrários de maneira eficiente.
- Algumas vezes, contudo, queremos ter acesso apenas ao elemento mínimo (ou máximo).
- *Heaps* ou filas de prioridades são estruturas de dados especializadas neste tipo de tarefa.

Heap - Pra que serve ?

UFABC

- Heaps são utilizados, por exemplo, para:
  - Ordenar um vetor (*Heapsort*);
  - Implementação de simuladores de eventos discretos (DES);
  - Cálculo de caminhos mínimos (Algoritmo de Djikstra);
  - Cálculo de árvores geradoras de custo mínimo (Algoritmos de Kruskal e Prim);
  - Compressão de arquivos (Algoritmo de Huffman por exemplo).

Heap - Como se usa?

UFABC

- Existem diversas maneiras de se implementar um heap. Em geral, um heap deve permitir as seguintes operações (se possível eficientemente):
  - `empty` - Cria e devolve um novo heap vazio
  - `insert x h` - Insere um elemento no heap e devolve o novo heap
  - `head h` - Devolve o elemento mínimo do heap
  - `tail h` - Remove o elemento máximo do heap e devolve o novo heap
- Estamos interessados em heaps intercaláveis (*mergeable*) que além das operações anteriores também oferecem:
  - `merge h1 h2` - cria e devolve um novo heap que contém todos os elementos de h1 e h2

Uma *typeclass* para heaps i

UFABC

```haskell
class (Show h, Ord a) => Heap h a where
  empty     :: h a
  null      :: h a -> Bool

  head      :: h a -> a
  tail      :: h a -> h a

  merge     :: h a -> h a -> h a

  insert    :: a -> h a -> h a
  insert a h = merge h (singleton a)

  singleton :: a -> h a
  singleton x = insert x empty
```

Uma *typeclass* para heaps ii

UFABC

```
15
16      -- Constrói um heap binário de uma lista qualquer.
17      fromList  :: [a] -> h a
18      fromList [] = empty
19      fromList l =
20        mergeList (map singleton l)
21        where
22          -- merge a lista até que tenha um único elemento
23          mergeList [a] = a
24          mergeList x = mergeList (mergePairs x)
25          -- merge par a par da lista de heaps
26          mergePairs (a:b:c) = merge a b : mergePairs c
27          mergePairs x = x
28
29
```

Uma *typeclass* para heaps iii

UFABC

```
30    -- Quase um HeapSort
31    toList :: h a -> [a]
32    toList h
33      | null h = []
34      | otherwise = head h : toList (tail h)
```

UFABC

- Heaps são frequentemente implementados como árvores binárias, onde o valor contido por qualquer um dos seus nós não é maior que qualquer um dos valores armazenados por seus filhos.
- Em linguagens imperativas eles são tipicamente implementados com vetores.

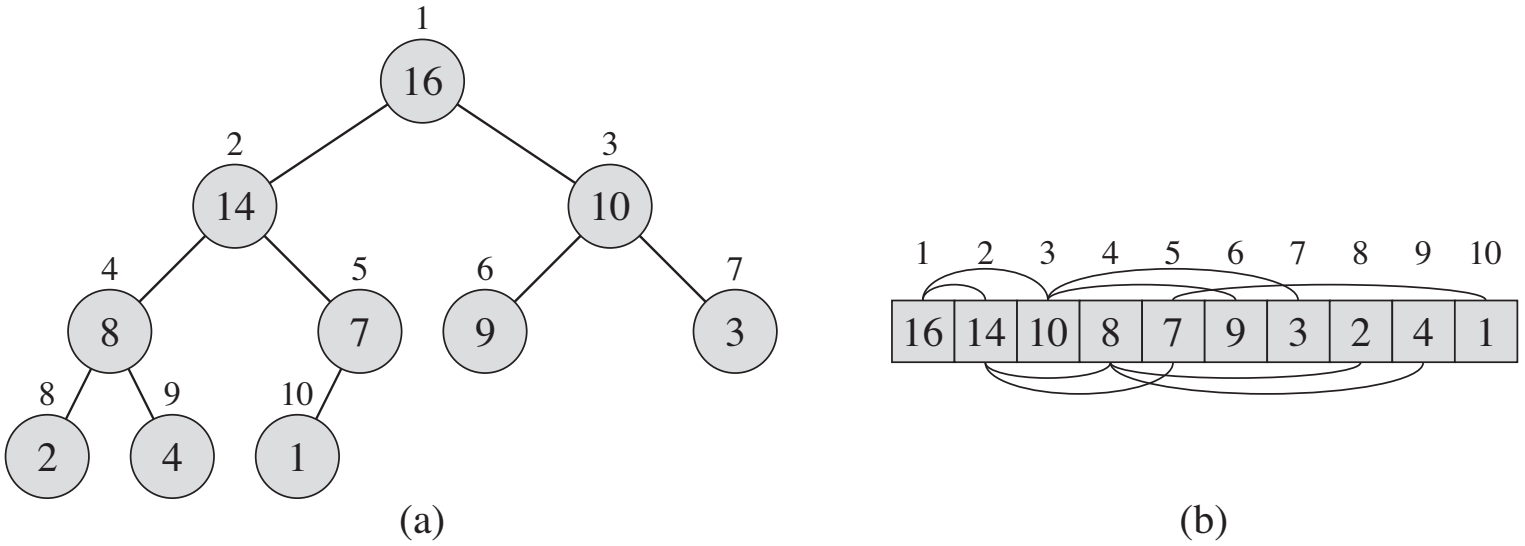


Figura 1: Fonte: [CLRS]

Heaps binários em um contexto funcional

UFABC

- Já, num contexto funcional, não temos vetores com endereçamento em tempo constante.
- A nossa definição será muito parecida com a que utilizamos em árvores binárias.
- Utilizaremos, em nossa primeira implementação o conceito de árvores esquerdistas.

Árvores esquerdistas

Árvores esquerdistas

UFABC

- Árvores esquerdistas foram criadas por Clark Allen Crane na década de 70
- São árvores binárias que obedecem a propriedade de (min) heap: um nó não é maior que qualquer um de seus filhos diretos e indiretos.
  - Não são de árvores de busca!
- Utilizar uma árvore esquerdista garante que tenhamos bons tempos de execução (constante para o mínimo e logarítmicos para as demais operações).

Árvores esquerdistas

UFABC

- Antes de definirmos o que são árvores esquerdistas, vamos definir:
  - O *rank* de um nó x é o comprimento do caminho de x até o nó não nulo mais à direita.
    - Consideramos que o *rank* de um nó nulo é -1.
    - No contexto funcional fala-se em espinha direita. O termo espinha será reutilizado quando falarmos de *finger trees*
- Árvores esquerdistas (carinhosamente chamadas de canhotinhas 😜) são árvores binárias nas quais vale a propriedade esquerdista:
  - Para todo nó x da árvore vale:

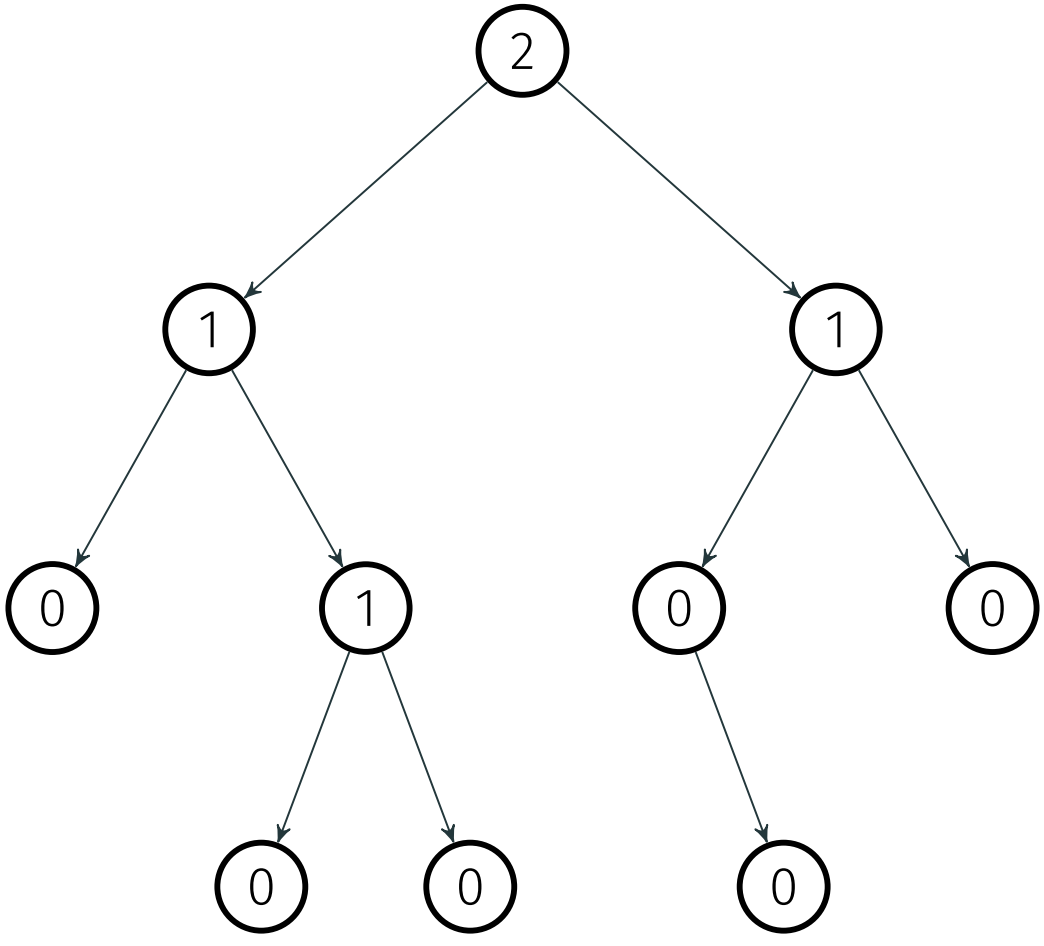
    $$\text{rank}(E(x)) >= \text{rank}(D(x)),$$

    onde D e E são, respectivamente, os filhos direito e esquerdo de x.

Identificando árvores esquerdistas

UFABC

- Os valores indicam o rank dos nós.

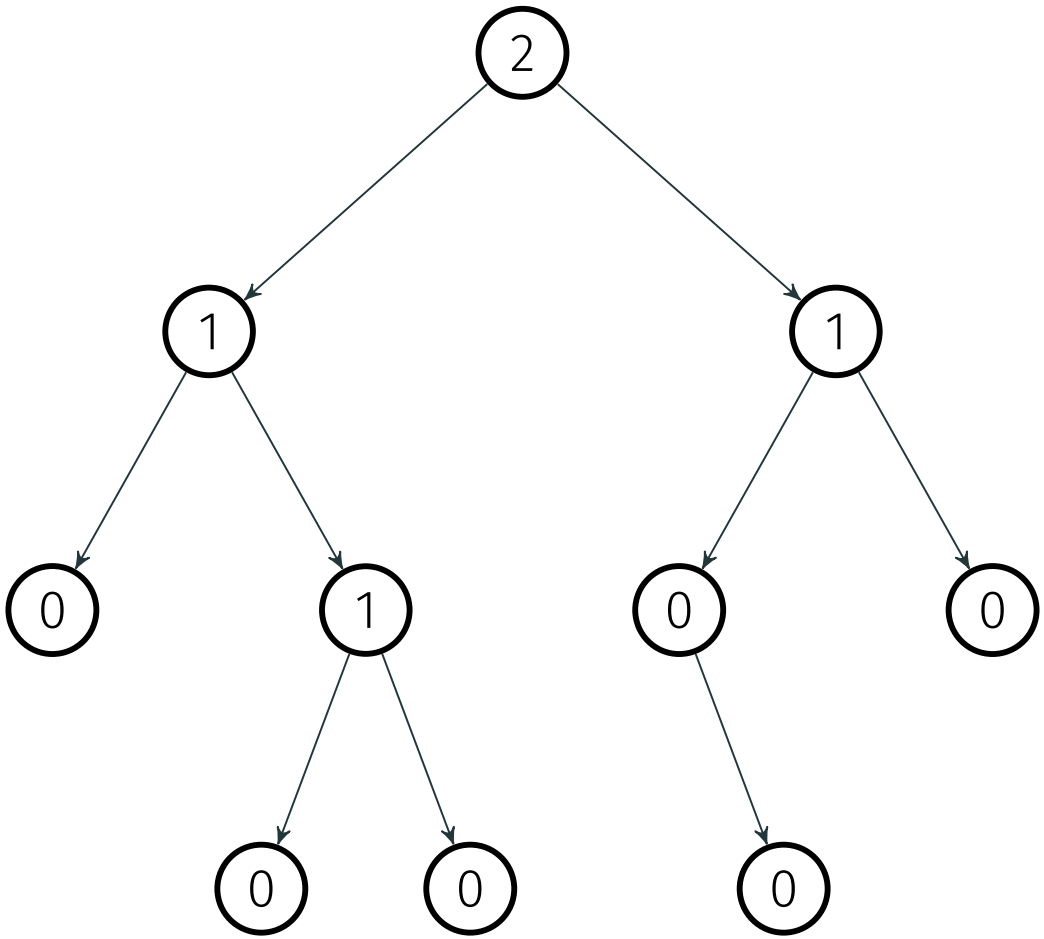


É esquerdista?

Identificando árvores esquerdistas

UFABC

- Os valores indicam o rank dos nós.

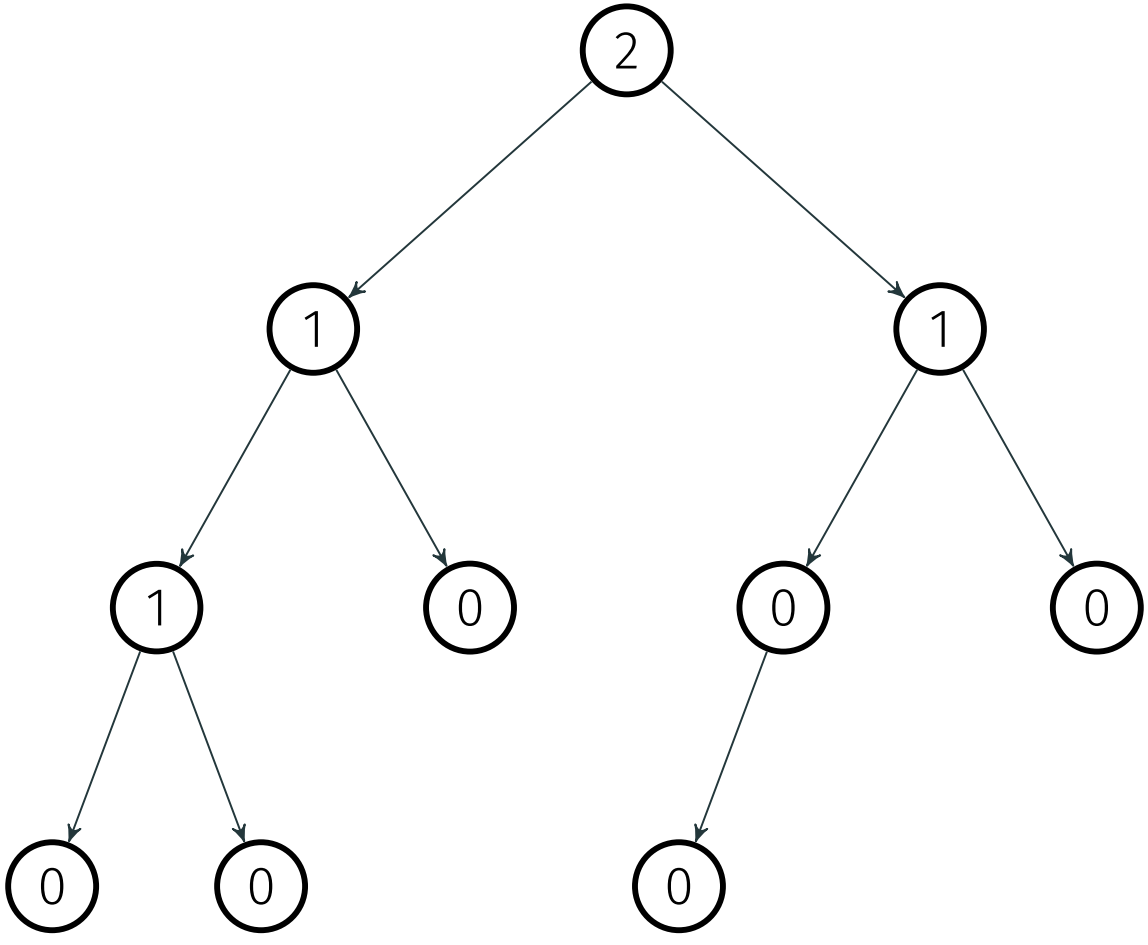


É esquerdista?

Não!

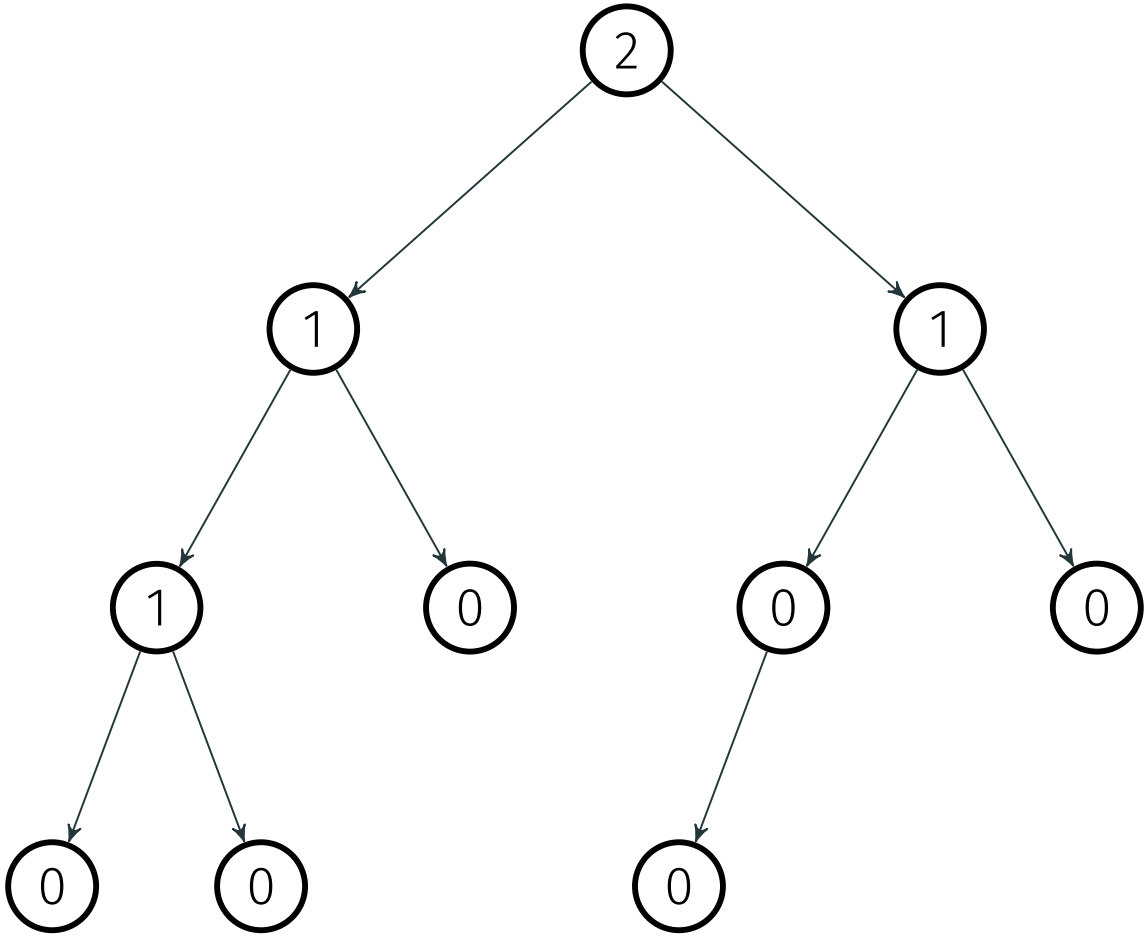
Identificando árvores esquerdistas

UFABC



É esquerdista?

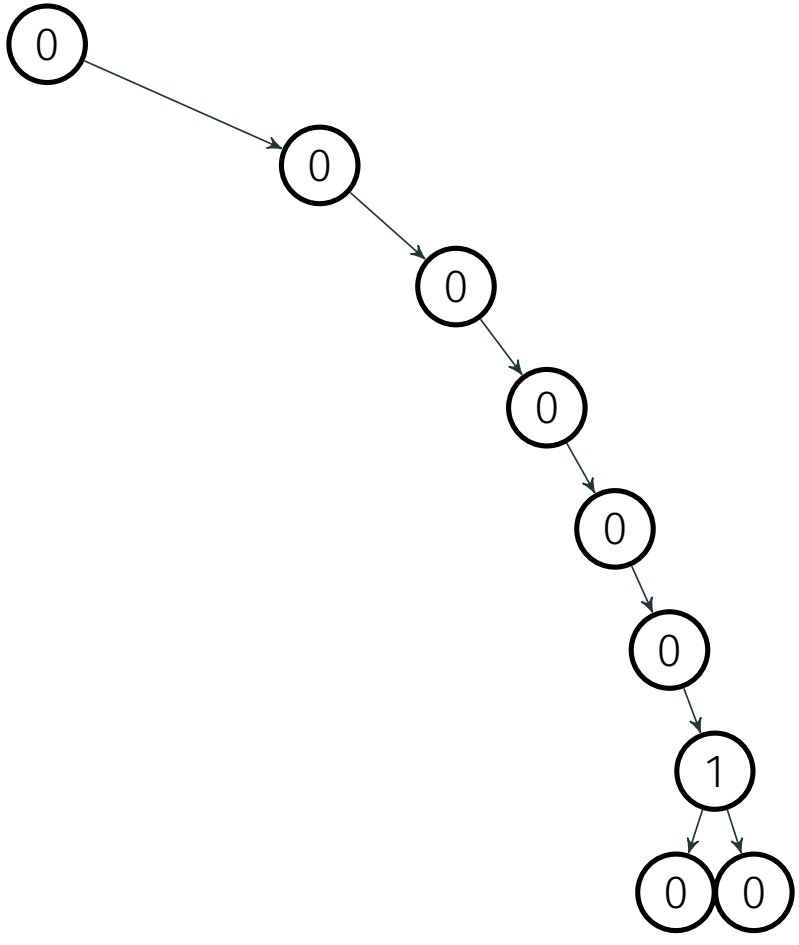
Identificando árvores esquerdistas



É esquerdista?

Sim!

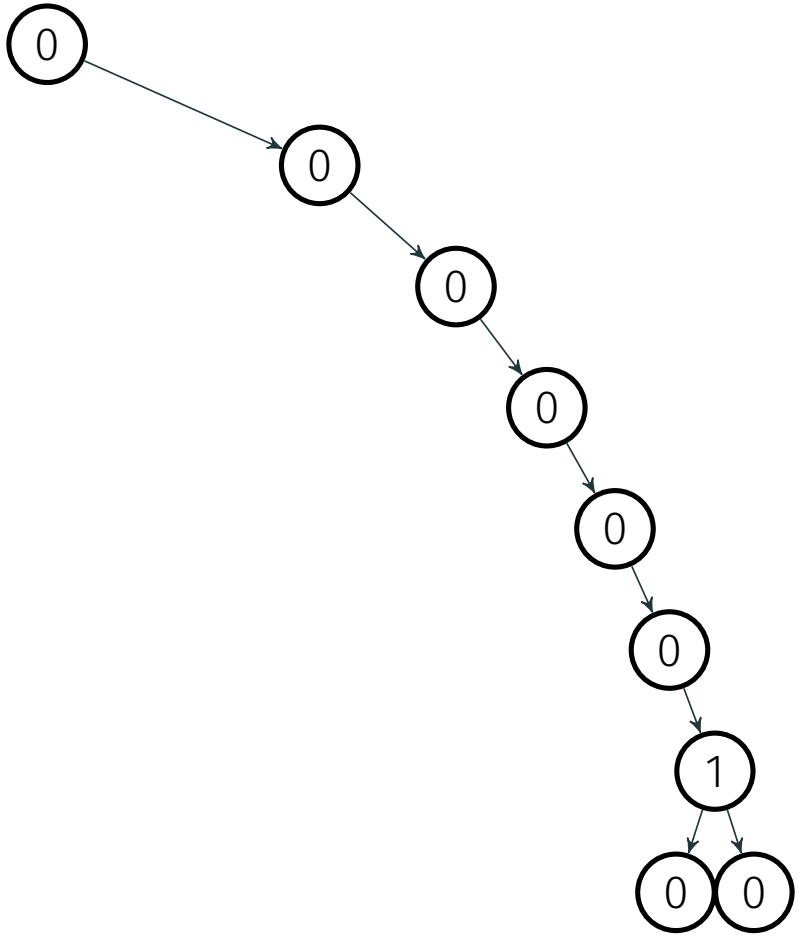
Identificando árvores esquerdistas



É esquerdista?

Identificando árvores esquerdistas

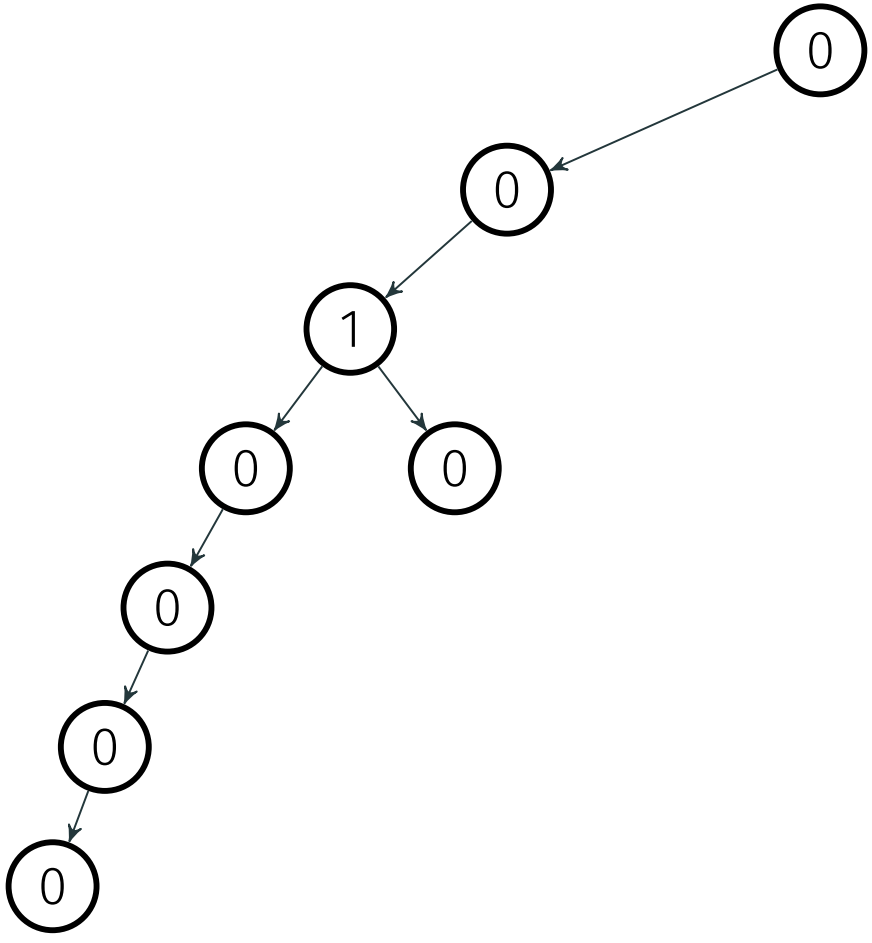
UFABC



É esquerdista?

Não!

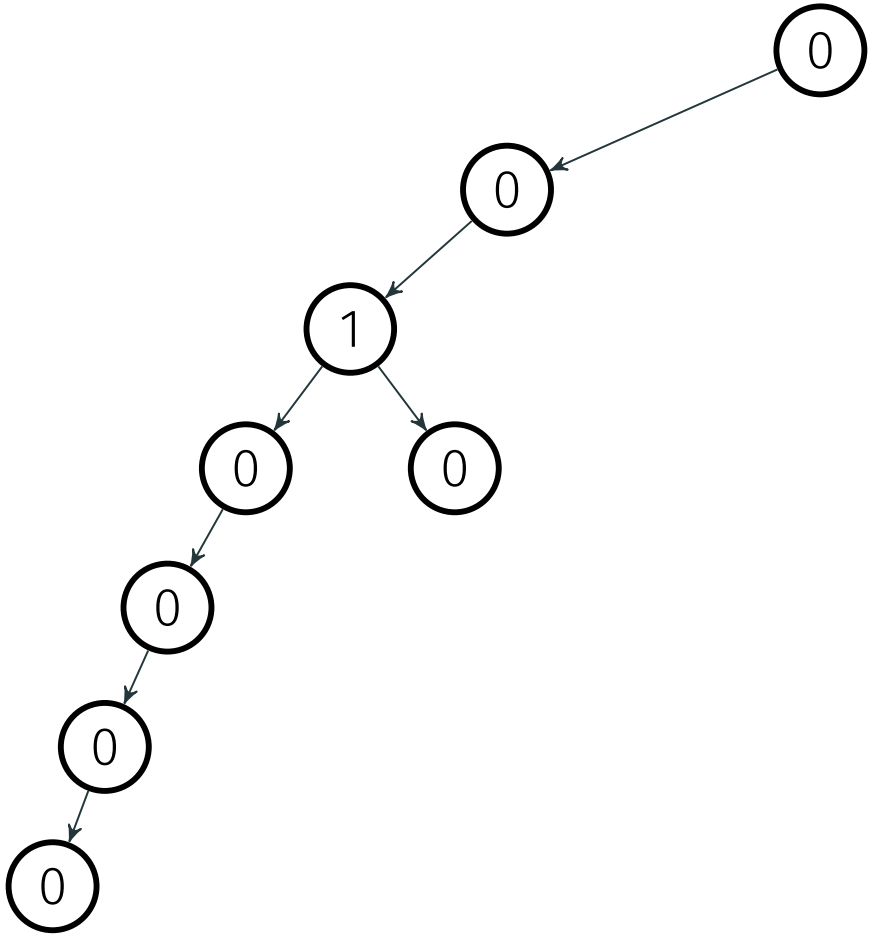
Identificando árvores esquerdistas



É esquerdista?

Identificando árvores esquerdistas

UFABC



É esquerdista?

Sim!

UFABC

- Não são balanceadas.
- Não são completas.
- Qualquer subárvore de uma árvore esquerdista é esquerdista.
- A espinha direita da árvore esquerdista é sempre o caminho mais curto até um nó vazio.
- O comprimento da espinha direita é no máximo $\lfloor \lg(n+1) \rfloor$.

UFABC

Proposição: Se a espinha direita de uma árvore esquerdista com raiz em x tem r nós, então a árvore tem ao menos $2^r - 1$ nós.

- A prova é por indução em r
  - **Caso base:** $r = 1$. A árvore tem ao menos $2^1 - 1 = 1$ nó.
  - **Passo de indução:** assuma que vale para $r - 1$, vamos provar que vale para r.
    - A espinha direita da subárvore direita de x, $D(x)$ tem $r - 1$ nós. Pela H.I. $D(x)$ tem $2^{r-1} - 1$ nós.
    - A espinha direita da subárvore esquerda de x, $E(x)$ também tem pelo menos $r - 1$ nós (Pq?), logo pela H.I. $E(x)$ tem $2^{r-1} - 1$ nós.
    - Assim, n o número total de nós da árvore esquerdista com raiz em x, é: $n \geq (2^{r-1} - 1) + (2^{r-1} - 1) + 1 = 2^r - 1$
- **Corolário:** $r = O(\lg n)$

Heaps esquerdistas

De volta ao Heap!

UFABC

- Por que árvores esquerdistas nos interessam?
  - Temos a garantia que o número de nós na espinha direita é curta em comparação com o número de nós da árvore
  - Vamos concentrar o trabalho na espinha direita!

Heaps esquerdistas

- Heaps esquerdistas são árvores esquerdistas com a propriedade de heap.
  - Ou seja, se os filhos à esquerda ($E(x)$) e à direita ($D(x)$) do nó x existirem, então $V(x) \leq V(E(x))$ e $V(x) \leq V(D(x))$
- Tudo gira em torno da operação `merge`.
- Exemplo com a sequência: $0, 28, 1, 8, 10, 4, 2, 12$

Implementando heaps esquerdistas

UFABC

```haskell
data LeftistHeap a where
  LLeaf :: Ord a => LeftistHeap a -- Nó nulo
  LNode :: Ord a => {
    rank   :: Int,
    key    :: a,
    _left  :: LeftistHeap a,
    _right :: LeftistHeap a
  } -> LeftistHeap a
```

Uma instância de `Heap` para `LeftistHeap` i

UFABC

```haskell
instance Ord a => Heap LeftistHeap a where
  -- O(1). Devolve um heap vazio.
  empty = LLeaf

  -- O(1). Devolve um heap com um único elemento.
  singleton x = LNode 0 x LLeaf LLeaf

  -- O(1). True caso heap vazio, False cc.
  null LLeaf = True
  null _     = False

  --  O(1). Devolve o elemento mínimo no heap
  head LNode {key = v} = v
  head _ = error "LeftistHeap vazio"
```

Uma instância de **Heap** para **LeftistHeap** ii

UFABC

```
15
16      --  O(lg n). Descarta a raiz e devolve o heap
        ↪  resultante.
17      tail (LNode _ _ h1 h2) = merge h1 h2
18      tail _ = error "LeftistHeap vazio"
19
20      -- Exercício: qual a complexidade de merge?
21      merge LLeaf h = h
22      merge h LLeaf = h
23      merge h1@(LNode _ v1 e1 d1) h2@(LNode _ v2 e2 d2)
24        | v1 <= v2  = makeNode v1 e1 (merge d1 h2)
25        | otherwise = makeNode v2 e2 (merge h1 d2)
26        where
27          rk LLeaf = -1
28          rk LNode {rank = r} = r
```

Uma instância de **Heap** para **LeftistHeap** iii

UFABC

```
29        makeNode x a b
30          | rk a >= rk b = LNode (rk b + 1) x a b
31          | otherwise    = LNode (rk a + 1) x b a
```

Heaps binomiais

Heaps binomiais

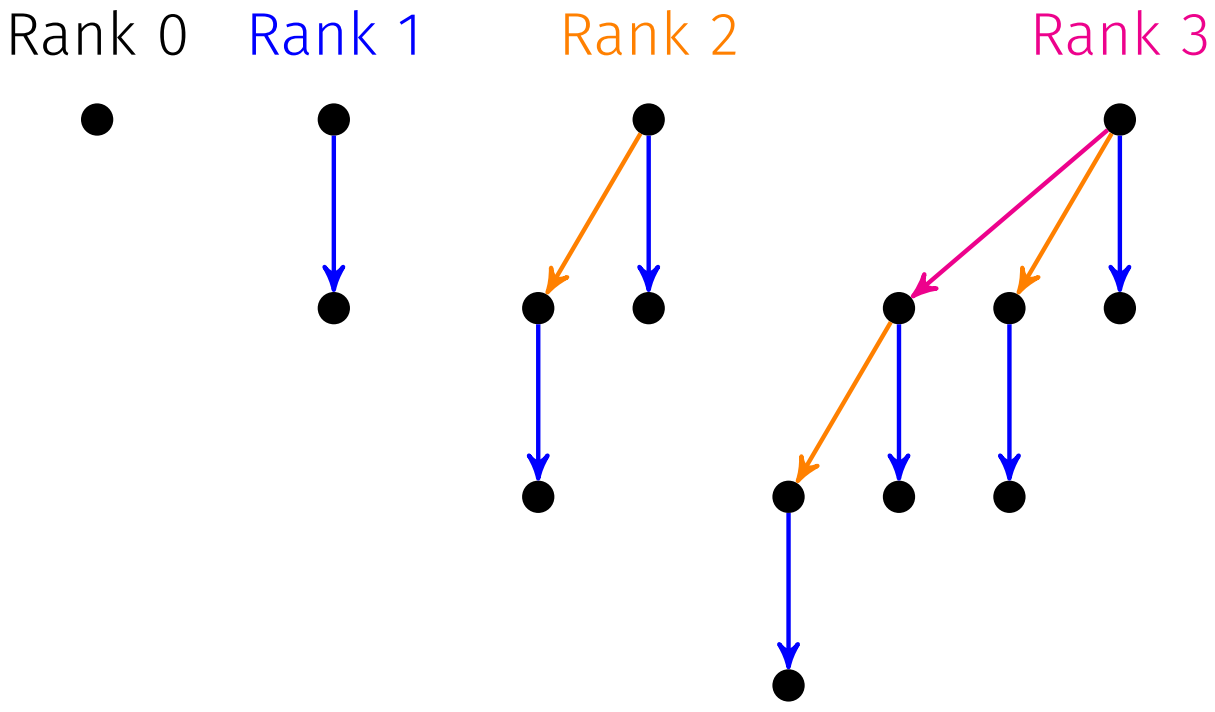
UFABC

- Criados no final da década de 70.
- Complexidades iniciais iguais às do heap esquerdistas.
  - Código mais complexo $\rightarrow$ mas é possível melhorar a inserção e merge para executarem em tempo amortizado $O(1)$.
- Assim como os heaps esquerdistas, heaps binomiais são baseados em uma estrutura mais simples chamada de árvore binomial.

UFABC

- Árvores binomiais são definidas como:
  - Uma árvore binomial de rank 0 é um único nó.
  - Uma árvore binomial de rank $r + 1$ é formada através da conexão de duas árvores binomiais de rank r, fazendo de uma árvore a filha mais à esquerda da outra.
- Uma maneira alternativa de definir as árvores binomiais é: uma árvore binomial de rank r é um nó com r filhos, $t_1 \ldots t_r$ onde cada t_i é uma árvore binomial de rank $r - i$.
- A partir da definição é fácil ver que uma árvore binomial de rank r contém exatamente 2^r nós.

Aqui está uma foto



Implementando uma árvore binomial

UFABC

```haskell
data BinomialTree a where
  BinomialTreeNode :: Ord a => {
    rnk       :: Int,
    root      :: a,
    _children :: [BinomialTree a] -- length _children ==
    ↪  rnk
    } -> BinomialTree a
```

- A lista com os filhos é mantida em ordem decrescente de rank e os elementos são armazenados satisfazem a propriedade de heap

Conectando (*linking*) duas árvores

UFABC

```haskell
linkTree :: BinomialTree a -> BinomialTree a ->
↪  BinomialTree a
linkTree t1@(BinomialTreeNode r1 x1 c1)
↪  t2@(BinomialTreeNode _ x2 c2)
  | x1 <= x2  = BinomialTreeNode (r1 + 1) x1 (t2 : c1)
  | otherwise = BinomialTreeNode (r1 + 1) x2 (t1 : c2)
```

- A propriedade de heap é mantida pois sempre conectamos árvores com raízes maiores abaixo das árvores com raízes menores.
- Apenas conectamos árvores com o mesmo rank.

Implementando o Heap Binomial

UFABC

- O heap binomial nada mais é que:
  - Uma lista de árvores binomiais que obedecem a propriedade de heap.
  - A lista é mantida em ordem crescente de ranks.
  - Não há ranks repetidos.

```haskell
newtype BinomialHeap a = BinomialHeap [BinomialTree a]
```

Sobre a representação - O pulo do gato!

UFABC

```haskell
newtype BinomialHeap a = BinomialHeap [BinomialTree a]
```

- Como cada árvore contém 2^r elementos, e nenhuma árvore tem rank repetido, os ranks das árvores de um heap de tamanho n. correspondem exatamente aos bits 1 da representação binária de n.
  - Por exemplo: heap de tamanho 21.
    - 21 em base 2: 10101
    - é armazenado como árvores de rank 0, 2, 4 (de tamanhos 1, 4 e 16 respectivamente).
- Consequentemente, assim como a representação de um número n em binário usa $\lfloor \lg(n+1) \rfloor$ bits, um heap binomial de tamanho n usa no máximo $\lfloor \lg(n+1) \rfloor$ árvores.

Operações no heap

UFABC

- Agora estamos prontos para definir as operações no heap.
- A maneira mais fácil de compreender o funcionamento das operações é imaginar que estamos fazendo operações de soma em números binários.

Inserção

UFABC

- Criamos uma árvore de rank 0 que contém o elemento que desejamos inserir.
- Em seguida, varremos as árvores que já fazem parte do heap (em ordem crescente de ranks) até encontrarmos um rank vazio (um bit zero). Durante a varredura conectamos árvores de mesmo rank r para criar uma nova árvore de rank $r + 1$. O conectar de árvores é semelhante ao "vai um"(*carry*) de uma soma binária.

```
1  insertTree :: BinomialTree a -> [BinomialTree a] ->
   ↪  [BinomialTree a]
2  insertTree t [] = [t]
3  insertTree t ts@(t':ts')
4    | rnk t < rnk t' = t:ts
5    | otherwise      = insertTree (linkTree t t') ts'
6
7  --Inserção fica: insertTree (BinomialTreeNode 0 x []) ts
```

Inserção

- O pior caso da inserção ocorre quando o heap tem um tamanho da forma $n = 2^k - 1$ (todos os bits acesos), neste caso k chamadas de `linkTree` são necessárias:
  $O(k) = O(\lg n)$
- Exemplo: 0, 28, 1, 8 , 10, 4, 2, 12

UFABC

- Para juntar dois heaps, varremos as duas listas de árvores em ordem crescente de rank e conectamos árvores de mesmo rank conforme avançamos.

```haskell
mergeTrees :: [BinomialTree a] -> [BinomialTree a] ->
↪  [BinomialTree a]
mergeTrees ts1 [] = ts1
mergeTrees [] ts2 = ts2
mergeTrees ts1@(t1:ts1') ts2@(t2:ts2')
  | rnk t1 < rnk t2 = t1 : mergeTrees ts1' ts2
  | rnk t2 < rnk t1 = t2 : mergeTrees ts1 ts2'
  | otherwise = insertTree (linkTree t1 t2) (mergeTrees
  ↪  ts1' ts2')
```

head e tail

UFABC

- Tanto `head` quanto `tail` se utilizam de uma função auxiliar (`removeMinTree`) que encontra a árvore com a raiz mínima e a retira da lista

```haskell
removeMinTree :: Ord a => [BinomialTree a] ->
↪  (BinomialTree a, [BinomialTree a])
removeMinTree [] = error "empty BinomialTree"
removeMinTree [t] = (t, [])
removeMinTree (t:ts)
  | root t < root t' = (t, ts)
  | otherwise        = (t', t:ts')
  where
    (t', ts') = removeMinTree ts
```

- Agora estamos prontos para definir uma instância de `Heap` para `BinomialHeap`

Uma instância de `Heap` para `BinomialHeap`

UFABC

```haskell
instance Ord a => Heap BinomialHeap a where

  empty = BinomialHeap []

  singleton x = BinomialHeap [BinomialTreeNode 0 x []]

  null (BinomialHeap ts) = P.null ts

  head (BinomialHeap ts) = root . fst $ removeMinTree ts

  tail (BinomialHeap ts) =
    BinomialHeap $ mergeTrees (reverse ts1) ts2
    where
      (BinomialTreeNode _ _ ts1, ts2) = removeMinTree ts

  merge (BinomialHeap ts1) (BinomialHeap ts2) =
    BinomialHeap $ mergeTrees ts1 ts2
```

Avaliação preguiçosa

Quero evitar a fadiga

UFABC

- A avaliação preguiçosa (en: *lazy evaluation*) é a estratégia de avaliação padrão de várias linguagens funcionais.
- Ela tem duas propriedades essenciais:
  - A avaliação de uma expressão é suspensa, ou atrasada (*delayed*) até que resultado seja necessário;
  - A partir do momento que o resultado tiver sido calculado ele é memoizado (en: *memoized*), *i.e. cached*, de modo que se ele for preciso novamente ele será reutilizado e não recomputado.

Quero evitar a fadiga

UFABC

- Vamos verificar como a avaliação preguiçosa funciona no Haskell.
- Para isso utilizaremos a função `sprint` no ghci que mostra o estado atual da variável.

Quero evitar a fadiga

UFABC

```
Prelude> :set -XMonomorphismRestriction
Prelude> x = 5 + 10
Prelude> :sprint x
x = _
```

Quero evitar a fadiga

UFABC

```
Prelude> x = 5 + 10
Prelude> :sprint x
x = _
Prelude> x
15
Prelude> :sprint x
x = 15
```

O valor de x é computado apenas quando requisitamos seu valor!

Quero evitar a fadiga

UFABC

```
Prelude> x = 1 + 1
Prelude> y = x * 3
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
```

Quero evitar a fadiga

UFABC

```
1  Prelude> x = 1 + 1
2  Prelude> y = x * 3
3  Prelude> :sprint x
4  x = _
5  Prelude> :sprint y
6  y = _
7  Prelude> y
8  6
9  Prelude> :sprint x
10 x = 2
```

A função `seq` recebe dois parâmetros, avalia o primeiro e retorna o segundo.

Eu quero agora!

UFABC

```
Prelude> x = 1 + 1
Prelude> y = 2 * 3
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
Prelude> seq x y
6
Prelude> :sprint x
x = 2
```

Quero evitar a fadiga

UFABC

```
Prelude> let l = map (+1) [1..10] :: [Int]
Prelude> :sprint l
l = _
Prelude> seq l ()
Prelude> :sprint l
l = _ : _
Prelude> length l
Prelude> :sprint l
l = [_,_,_,_,_,_,_,_,_,_]
Prelude> sum l
Prelude> :sprint l
l = [2,3,4,5,6,7,8,9,10,11]
```

Weak Head Normal Form (WHNF)

UFABC

Ao fazer:

```
> z = (2, 3)
> :sprint z
z = _
> z `seq` ()
()
> :sprint z
z = (_,_)
```

A função *seq* apenas forçou a avaliação da estrutura de tupla. Essa forma é conhecida como *Weak Head Normal Form*.

UFABC

Para avaliar uma expressão em sua forma normal, podemos usar a função `force` da biblioteca Control.DeepSeq:

```
1  > import Control.DeepSeq
2  > z = (2,3)
3  > force z
4  > :sprint z
5  z = (2,3)
```

UFABC

- Streams são listas similares a listas comuns. Contudo a avaliação de cada célula é feita de maneira preguiçosa.
  - Em Haskell todas as listas são assim.
  - Em outras linguagens pode ser necessário fazer alguma ginástica para implementá-las.
- Quando se fala em streams, normalmente falamos de listas infinitas ou cujo comprimento é desconhecido a priori.

```
1 [1..] -- todos os números inteiros positivos
2 [2,4..] -- todos os números pares positivos
```

Números primos

- Queremos uma lista com todos os números primos
- Aqui vamos fazer apenas uma versão simples...

Tip

Se quiser mergulhar na toca do coelho, neste link você encontra uma versão otimizada baseada em:

- "Lazy wheel sieves and spirals of primes", de Colin Runciman (1997).
- "The Genuine Sieve of Eratosthenes"de Melissa O'Neil (2009).

Primeiro exemplo - Função `ePrimo`

UFABC

```haskell
ePrimo :: Int -> Bool
ePrimo x =
  all (\e -> x `rem` e /= 0) possiveisFatores
  where
    maxDiv = ((floor . sqrt) :: Double -> Int) $
     ↪  fromIntegral x
    possiveisFatores = takeWhile (<= maxDiv) [2..]
```

E a lista de todos os primos ficaria:

```haskell
primos = filter ePrimo [2..]
```

Dá pra melhorar!

UFABC

- Não precisamos testar todos os números ≥ 2. Basta testar aqueles da forma $6k \pm 1$, $k \geq 1$ exceto 2 e 3 é claro.
- Além disso...

Ideia!

- Podemos usar a própria lista de primos que estamos gerando para verificar a primalidade!

Juntando tudo

UFABC

```haskell
primos =
  2 : 3 : filter ePrimo candidatos
  where
    candidatos =  junta [5,11..] [7,13..]
    junta ~(a:as) ~(b:bs) = a : b : junta as bs

ePrimo :: Int -> Bool
ePrimo x =
  all (\e -> x `rem` e /= 0) possiveisFatores
  where
    maxDiv = ((floor . sqrt) :: Double -> Int) $
     ↪  fromIntegral x
    possiveisFatores = takeWhile (<= maxDiv) primos
```

Segundo exemplo - Sequência de Fibonacci

UFABC

- A famosa sequência de Fibonacci:

$$F_n = F_{n-1} + F_{n-2}$$

Com $F_1 = F_2 = 1$, e, por convenção, $F_0 = 0$.

- Quero definir uma função que devolve a sequência completa.

UFABC

Tip

- **`Momento cultural 1:`** Conforme $n \to \infty$, a razão $\frac{F_n}{F_{n-1}}$ tende a ϕ (a razão áurea).

UFABC

Tip

- **`Momento cultural 1:`** Conforme $n \to \infty$, a razão $\frac{F_n}{F_{n-1}}$ tende a ϕ (a razão áurea).

- **`Momento cultural 2:`** A sequência de Fibonacci não é especial. Qualquer recorrência com essa forma e com valores iniciais arbitrários inteiros positivos serve. 😛

UFABC

Tip

- **`Momento cultural 1:`** Conforme $n \to \infty$, a razão $\frac{F_n}{F_{n-1}}$ tende a ϕ (a razão áurea).
- **`Momento cultural 2:`** A sequência de Fibonacci não é especial. Qualquer recorrência com essa forma e com valores iniciais arbitrários inteiros positivos serve. 😛
- **`Momento cultural 3:`** Em 1910 o matemático Mark Barr passou a usar a letra grega ϕ para denotar a razão áurea em homenagem ao escultor e arquiteto Fídias (em grego Φειδίας 480 a.C.—430 a.C.) que teria usado a razão áurea no projeto do Paternon.

UFABC

Tip

- **`Momento cultural 1:`** Conforme $n \to \infty$, a razão $\frac{F_n}{F_{n-1}}$ tende a ϕ (a razão áurea).
- **`Momento cultural 2:`** A sequência de Fibonacci não é especial. Qualquer recorrência com essa forma e com valores iniciais arbitrários inteiros positivos serve. 😛
- **`Momento cultural 3:`** Em 1910 o matemático Mark Barr passou a usar a letra grega ϕ para denotar a razão áurea em homenagem ao escultor e arquiteto Fídias (em grego Φειδίας 480 a.C.—430 a.C.) que teria usado a razão áurea no projeto do Paternon.
- **`Momento Cultural 4:`** Mais tarde, Barr revelou que achava improvável que Fídias tenha usado a razão áurea segundo a concepção moderna, mas já era tarde demais. O número já tinha sido batizado.

UFABC

- Note que a sequência tem a seguinte característica

```
      0  1  1  2  3  5  8  13  21 ...
+        0  1  1  2  3  5   8  13 ...
=     0  1  2  3  5  8  13  21 34 ...
```

De que isso adianta?

Lazy Fibonacci

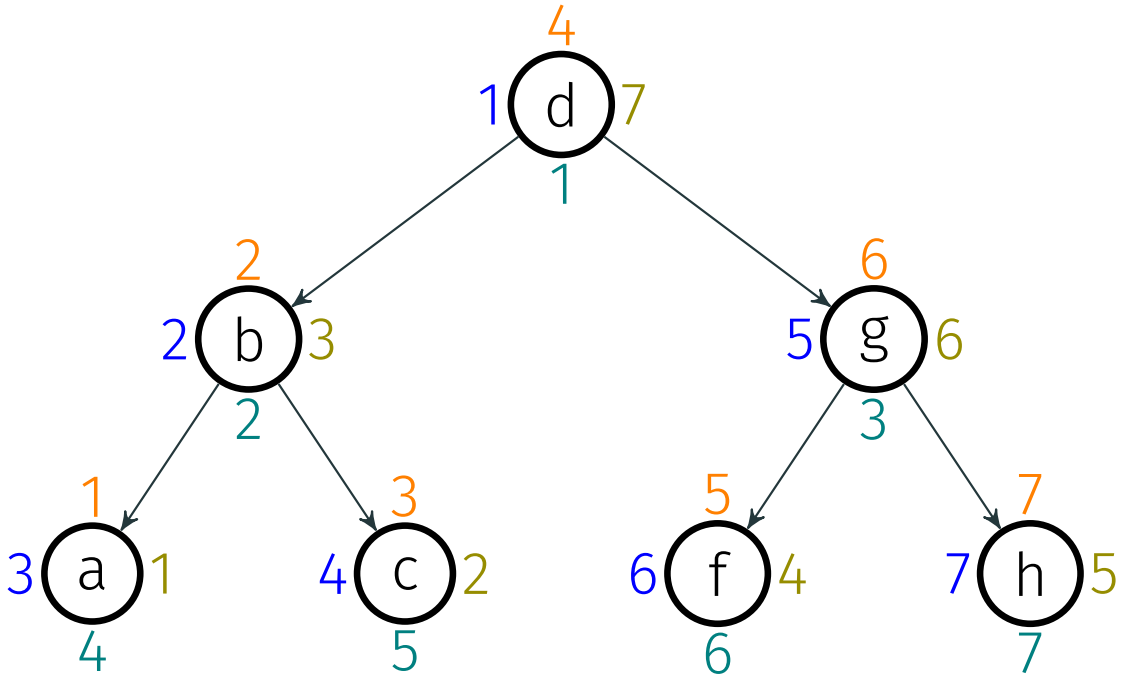
UFABC

```haskell
fibos :: [Integer]
fibos = 0 : 1 : zipWith (+) fibos (tail fibos)
```

Revisitando a varredura em largura

UFABC

- Os três percursos tradicionais de árvores, pré/in/pós-ordem podem ser aplicados de maneira trivial.



Pré-ordem: d, b, a, c, g, f, h
In-ordem: a, b, c, d, f, g, h
Pós-ordem: a, c, b, f, h, g, d
Largura: d, b, g, a, c, f, h

Varredura com fila fuleira

UFABC

```haskell
larguraQ :: Show a => Arv a -> IO ()
larguraQ arv =
  larguraFila $ enq empty arv
  where
  larguraFila q =
    if isEmpty q then return ()
    else case deq q of
      (ArvVazia, q') -> larguraFila q'
      (No e x d, q') -> do
        print x
        larguraFila $ enq (enq q' e) d
```

Largura por nível

UFABC

```haskell
larguraNivel :: Show a => Arv a -> IO ()
larguraNivel arv =
  larguraNivel' [arv]
  where
    printArv ArvVazia = return ()
    printArv (No _ x _) = print x

    filhos ArvVazia = []
    filhos (No e _ d) = [e, d]

    larguraNivel' [] = return ()
    larguraNivel' lvl = do
      mapM_ printArv lvl
      larguraNivel' $ concatMap filhos lvl
```

Numerando os nós em largura

UFABC

- O real problema ocorria quando precisávamos fazer alterações na árvore.

```haskell
bfn1 t =
  fst $ deq $ bfn' 1 (enq empty t)
  where
    bfn' i inQ
      | isEmpty inQ = empty
      | otherwise =
        case deq inQ of
          (ArvVazia, inQ1) -> enq (bfn' i inQ1) ArvVazia
          (No e x d, inQ1) ->
            let
              inQ2 = enq (enq inQ1 e) d
              outQ0 = bfn' (i + 1) inQ2
              (d', outQ1) = deq outQ0
              (e', outQ2) = deq outQ1 in
              enq outQ2 (No e' (x, i) d')
```

Solução de Jones e Gibbons

UFABC

```haskell
data Arv a = ArvVazia | No (Arv a) a (Arv a)
```

- Imagine que magicamente caia no seu colo uma lista com os índices dos primeiros elementos de cada nível.
- A função abaixo numera os nós em largura.

```haskell
bfn' :: ([Int], Arv a) -> ([Int], Arv (a, Int))
bfn' (ks, ArvVazia) = (ks, ArvVazia)
bfn' (k : ks0, No a x b) =
  (k + 1 : ks2, No a' (x, k) b')
  where
    (ks1, a') = bfn' (ks0, a)
    (ks2, b') = bfn' (ks1, b)
```

- Compare a saída de `bfn'` com a sua entrada. O que você vê?

Solução de Jones e Gibbons - Ideia!

UFABC

Solução de Jones e Gibbons - Ideia!

UFABC

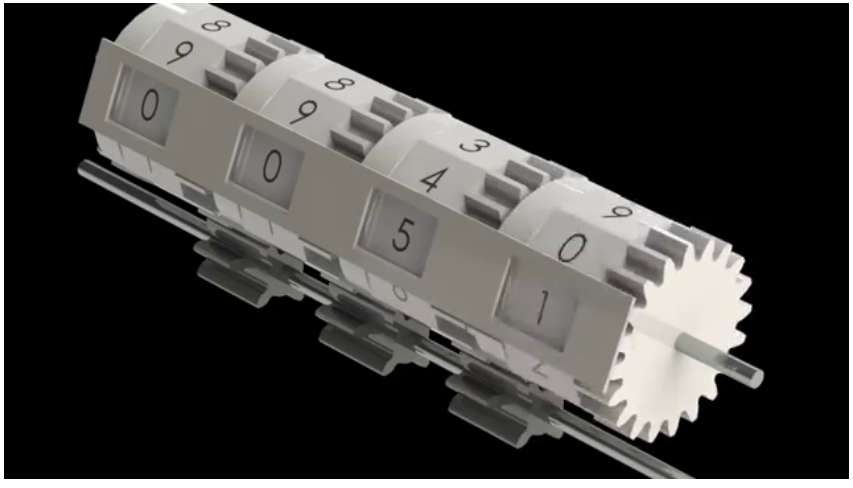
- Vamos plugar a saída na entrada!
- Precisamos apenas adicionar o primeiro nível!

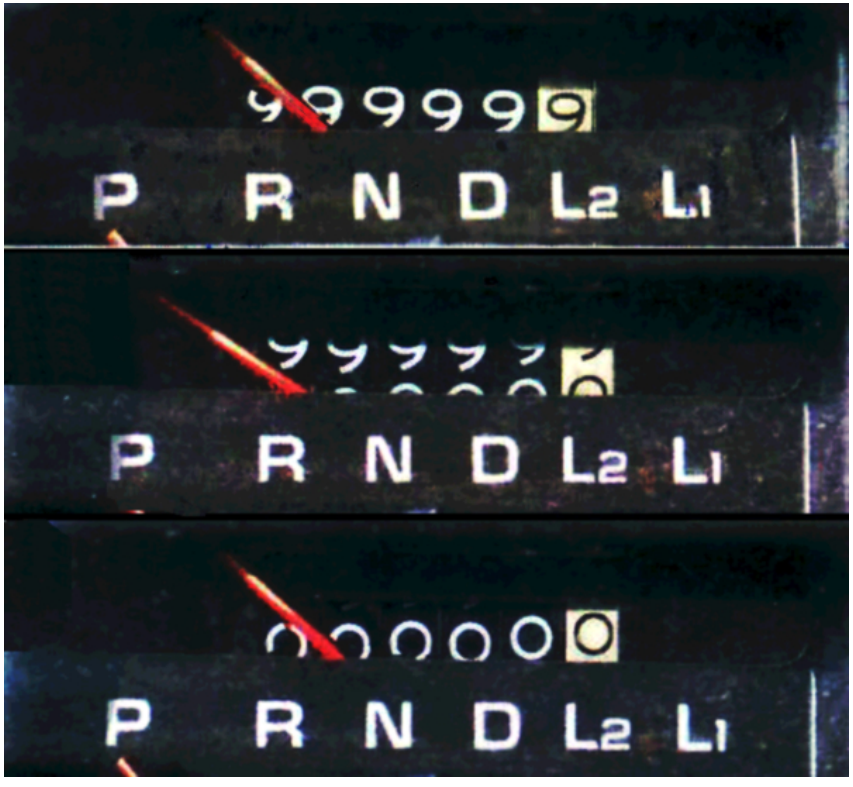
```haskell
bfn' :: ([Int], Arv a) -> ([Int], Arv (a, Int))
bfn' (ks, ArvVazia) = (ks, ArvVazia)
bfn' (k : ks0, No a x b) =
  (k + 1 : ks2, No a' (x, k) b')
  where
    (ks1, a') = bfn' (ks0, a)
    (ks2, b') = bfn' (ks1, b)
```

```haskell
bfn t = t'
  where (ks, t') = bfn' (1 : ks, t)
```

Análise amortizada

O odômetro





Figuras: trinityscsp e Wikipedia

Um odômetro binário

UFABC

- A análise do odômetro binário é muito parecida com o decimal.
- Pode ser visto como uma série de incrementos sucessivos.
- Cada mudança de dígito tem um custo real que vamos convencionar como sendo 1.

Valor do contador	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Custo total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figura: [CLRS]

Analisando o odômetro binário

- Note que o custo acumulado nunca é maior que o dobro de operações.
- Poderíamos até considerar que sendo o contador de k bits, o custo por incremento será $O(k)$ e para uma sequência de n incrementos, será $O(nk)$.
- O problema é que estamos exagerando, a maior parte das operações nem toca em todos os bits!

Valor do contador	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Custo total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figura: [CLRS]

Analisando o odômetro binário

- O bit 0 é modificado a cada incremento
- O bit 1 é modificado a cada 2 incrementos
- O bit 2 é modificado a cada 4 incrementos
- ...

Logo para uma sequência de n ($n \leq 2^k$) incrementos (com o contador a partir do 0) o total de modificações é:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Valor do contador	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Custo total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figura: [CLRS]

UFABC

- A noção da análise amortizada vem da seguinte observação:
  - Dada uma sequência de operações, estamos interessados no tempo total de execução da sequência e não de cada operação individualmente.
- Em outras palavras, queremos que a execução da sequência toda de n operações seja, por exemplo, $O(n)$ mas não nos importamos se alguma das operações não for $O(1)$.
- Em outras palavras, o custo médio de cada operação e, portanto, o custo amortizado de cada operação é $O(1)$.

Revisitando filas

- Há dois métodos consagrados para a análise amortizada:
  - O método do banqueiro ou de contabilidade;
  - O método do físico ou potencial.
- Vamos apresentar uma estrutura de dados para filas e exemplificar como podemos empregar o método do banqueiro[1].

[1] Os método do banqueiro e do físico são equivalentes, mas por motivos de tempo nos concentraremos aqui apenas no método do banqueiro. Caso queira saber mais sobre o método cheque os livros [CO] e [CLRS]. Para a mesma demonstração que fazemos aqui para filas veja o capítulo 5 de [CO].

UFABC

- A implementação funcional mais comum de filas envolve o uso de duas listas `f` e `r`.
  - `f` contém os elementos no início (*front*) da fila
    - Mantém os elementos na mesma ordem da fila
  - `r` contém os elementos do final (*rear*) da fila
    - Mantém os elementos na ordem inversa da fila

```
1 --                  f     r
2 type Fila a = ([a], [a])
```

Fila - `head` e `tail`

UFABC

- A cabeça da lista está localizada em `f`, então tanto a função `head` quanto a função `tail` operam neste ponto

```haskell
head :: Fila a -> a
head (x:f, r) = x

tail :: Fila a -> Fila a
tail (x:f, r) = (f, r)
```

UFABC

- De maneira semelhante, o último elemento da fila é o primeiro elemento de `r`, logo `snoc`[2] simplesmente adiciona um elemento à `r`.

```haskell
snoc :: Fila a -> a -> Fila a
snoc (f, r) x = (f, x : r)
```

[2] `snoc` vem de `cons` na ordem inversa. Ninguém sabe exatamente quando isso surgiu, mas há papers da década de 70 que já usam essa nomenclatura.

O invariante

UFABC

- Elementos são adicionados em `r` e retirados de `f`.
- Assim, em algum momento eles tem que ser migrados.
- Faremos a migração se após uma operação houver ao menos um elemento na fila mas `f` estiver vazio (ou seja, há elementos em `r`).
- O objetivo é manter o seguinte invariante: `f` estará vazio se e somente se `r` também estiver vazio (ou seja, a fila toda está vazia).
  - Note que se isso não fosse verdade, se `f` estivesse vazio e `r` não, então o primeiro elemento da fila seria o último elemento de `r` (acesso em tempo $O(n)$).
  - Este invariante garante que **`head`** (e **`tail`**) sempre rodam em tempo $O(1)$.

Mantendo o invariante

- Para isso precisaremos adaptar as funções que alteram a fila (`tail` e `snoc`).
- Vamos usar uma função auxiliar que faz o trabalho

```haskell
check :: Fila a -> Fila a
check ([], r) = (rev r, [])
check q = q

snoc :: Fila a -> a -> Fila a
snoc (f, r) x = checkf (f, x : r)

tail :: Fila a -> Fila a
tail (x:f, r) = checkf (f, r)
```

UFABC

- Para provar um limite amortizado, é preciso definir o custo amortizado de cada operação e provar que, para qualquer sequência de operações, o custo total amortizado das operações é um limite superior no custo total real.

$$\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} t_i$$

Onde a_i é o custo amortizado da operação i, t_i é o custo real da operação i e n é o número total de operações.

UFABC

- Na verdade na prática acabamos quase sempre provando algo mais forte que é:

$$\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_j$$

Onde $j \leq n$ indica que a qualquer momento a soma das operações amortizadas é um limite superior ao custo real acumulado.

UFABC

- A diferença entre o custo amortizado total acumulado e o custo real total acumulado é chamado de poupança acumulada (en: *accumulated savings*).
- Ocasionalmente o custo de uma operação vai ser maior que o seu custo amortizado. Neste caso, retiramos a diferença da poupança.
  - Tais operações são denominadas caras (en: *expensive*).
  - As que tem custo real igual ou inferior ao amortizado são chamadas de baratas (en: *cheap*).
- A chave é mostrar que operações caras só ocorrem quando a diferença do custo pode ser paga pela poupança!

O método do Banqueiro - Créditos

UFABC

- O método do banqueiro associa créditos a cada elemento/posição da estrutura de dados.
- Quando essas posições forem acessadas, esses créditos pagarão a diferença no custo.

O método do Banqueiro - Invariante

UFABC

- Todo crédito precisa ser alocado antes de ser usado e nenhum crédito pode ser usado mais de uma vez!
- Normalmente para usar o método do banqueiro, define-se um invariante que garante que quando uma operação cara ocorrer então, há dinheiro suficiente na poupança para cobrir o custo.

UFABC

- `head` tem custo amortizado 1 (e real também!)
- `snoc`
  - O invariante para a nossa análise é de que cada elemento em $\mathbf{r}$ está associado com 1 crédito.
  - Quando ocorre um `snoc`, gastamos um crédito para colocar o elemento em $\mathbf{r}$ e outro para deixar associado ao elemento propriamente dito.
    - Assim, uma lista $\mathbf{r}$ com n elementos tem n créditos associados.
    - O custo amortizado de uma operação `snoc` é portanto 2

UFABC

- `tail`
  - Se `tail` não precisar inverter a lista, ele custa 1 crédito
  - Se `tail` precisar inverter a lista, ele precisará fazer $n + 1$ passos, onde n é o tamanho da lista.
    - A poupança neste caso tem n créditos
    - Após a execução de `tail`, a poupança estará vazia e `tail` terá tido um custo amortizado de 1 ($n + 1 - n = 1$)

Logo, como todas as operações têm custo amortizado ≤ 2, então qualquer sequência com n operações terá custo amortizado $\leq 2n = O(n)$.

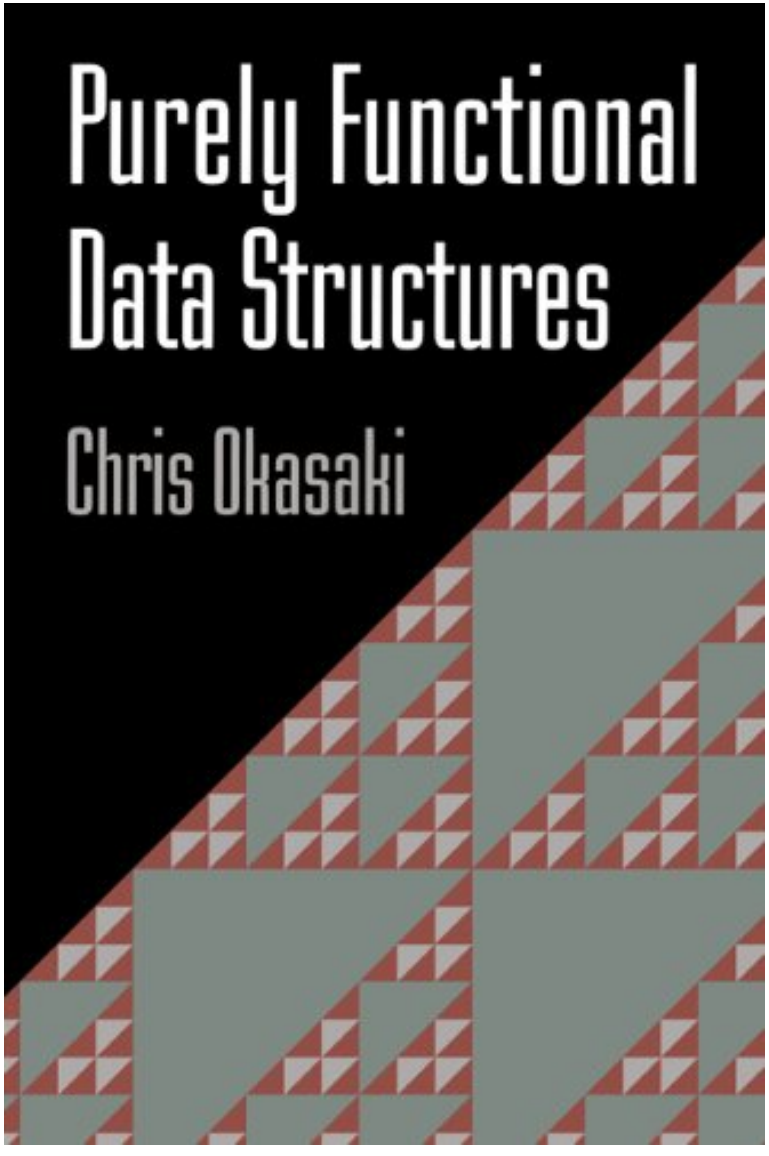
Não mostramos diretamente, mas é fácil mostrar que de fato $\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j}$ vale.

Revisitando os Heaps Binomiais

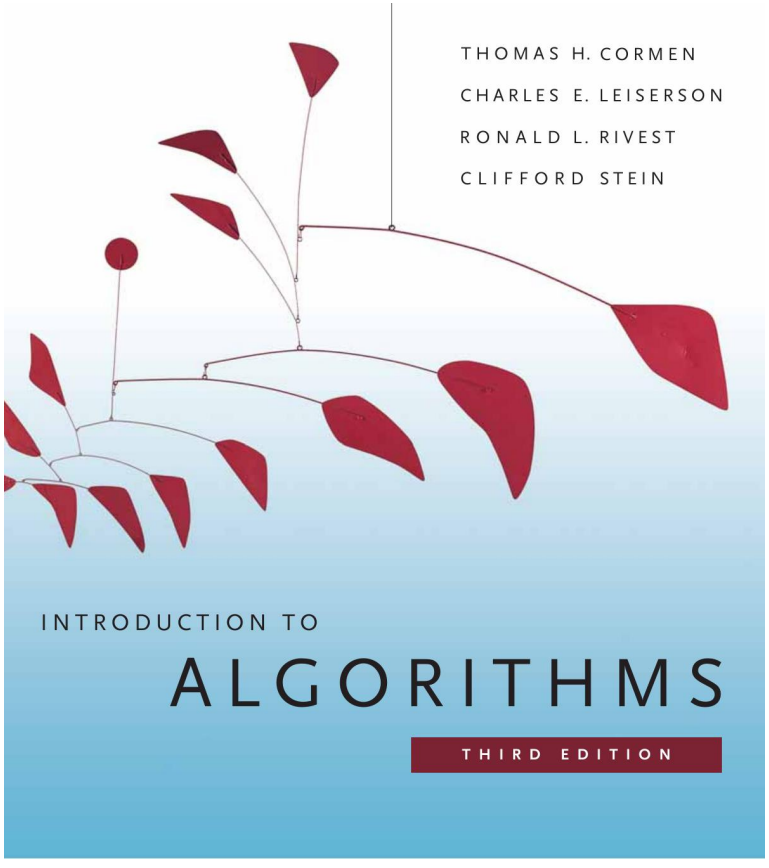
UFABC

- Mostramos que a inserção em heaps binomiais leva tempo $O(\lg n)$.
- É possível mostrar que a inserção leva tempo $O(1)$ amortizado
- Lembre-se do odômetro!
  - O número de árvores no heap é equivalente ao número de 1s na sua representação em binário
  - Inserção precisa de $k + 1$ passos onde k é o número de chamadas a função `link`.
  - Se mantivermos um crédito por árvore, o custo amortizado de `insert` (descontados os créditos da poupança) passa a ser O(1).

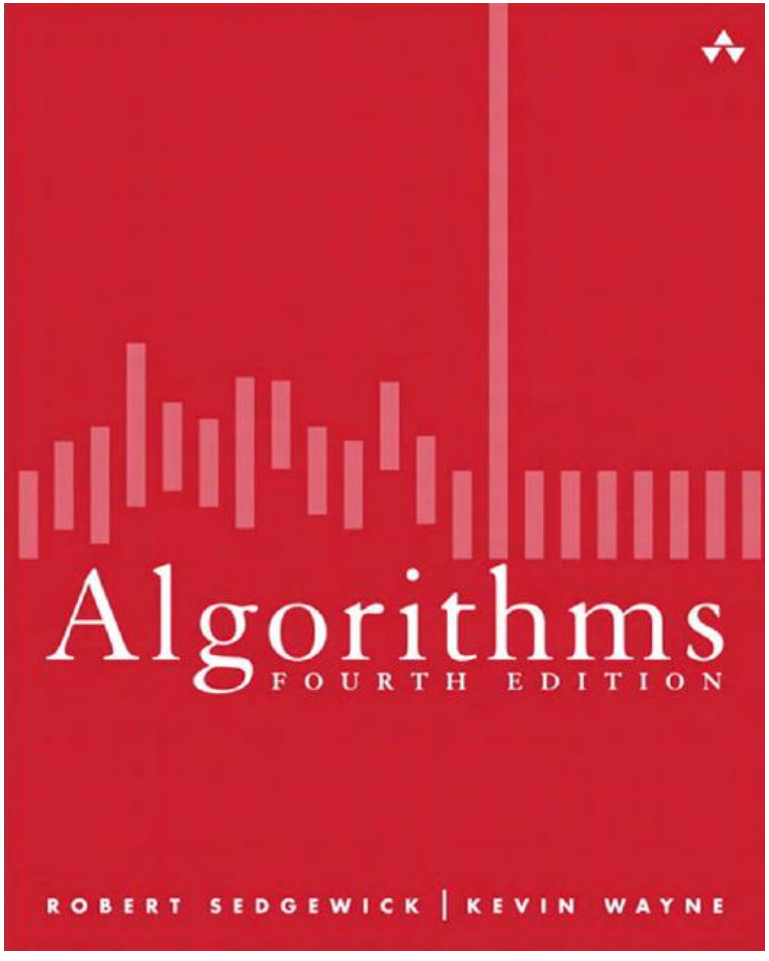
Referências



- [CO]
- Purely Functional Data Structures
  - Por *Chris Okasaki*



- [CLRS]
- Introduction to Algorithms
  - Por *Thomas H. Cormen & Charles E. Leiserson & Ronald L. Rives/t & /Clifford Stein*



- [SW]
- Algorithms
  - Por *Robert Sedgewick* & *Kevin Wayne*

UFABC

- Okasaki, Chris. ”Breadth-first numbering: lessons from a small exercise in algorithm design.”International Conference on Functional Programming: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. 2000.
- Jones, Geraint, and Jeremy Gibbons. ”Linear-time Breadth First Tree Algorithms: An exercise in the arithmetic of folds and zips”. Department of Computer Science, The University of Auckland, New Zealand, 1993.