Estruturas de dados puramente funcionais

Dia 3

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Outline

1. Finger Trees
2. Deques
3. Catenable Deques
4. Monoides
5. Finger Tree Splitting
6. Listas indexáveis
7. Heaps
8. Outras estruturas de dados
9. Comentários finais
10. Referências

Finger Trees

Figura 1: Fonte: http://weddbook.com

UFABC

- Finger Trees são uma daquelas estruturas de dados que depois que você realmente entende, você fica boquiaberto por algum tempo (eu pelo menos fiquei):
  - Pela sua simplicidade e eficiência;
  - Pelo seu poder de adaptação.
- Finger Trees são uma (baita) variação de uma outra estrutura de dados muito conhecida no mundo imperativo chamada de Árvores 2-3.

Disclaimer

UFABC

Note

Disclaimer: Esta aula é fortemente baseada no paper que apresentou as FingerTrees ao mundo em 2006. Boa parte do código foi ”adaptada”do paper original para facilitar o entendimento. Apesar de funcional (sem trocadilhos), o código que apresentamos não é otimizado. As referências nos slides finais têm links para implementações otimizadas em Haskell e outras linguagens.

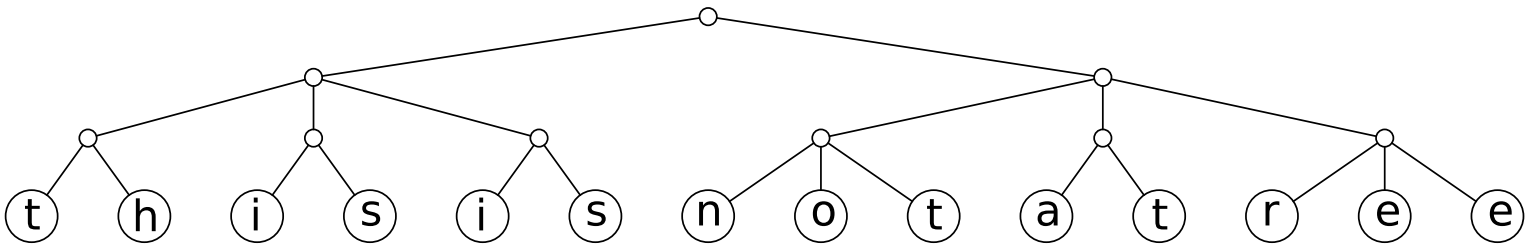
UFABC

- Árvores 2-3 são árvores balanceadas que servem de ponto de partida para várias outras:
  - Árvores rubro-negras
  - Árvores B, B+, B*
  - Árvores 2-3-4
  - E, quem diria, Finger Trees

UFABC

- Árvores 2-3 são árvores onde cada nó pode ter 2 ou 3 filhos
  - E por consequência podem guardar 2 valores por nó.
- Algumas variações guardam os elementos nos nós internos e outras apenas nas folhas
- Aqui estamos interessados na variação que guarda os elementos nas folhas.
  - Na verdade é muito mais parecido com uma árvore B+ sem os links que encadeiam as folhas

Aqui está uma foto:



Exemplo de inserção Árvores 2-3

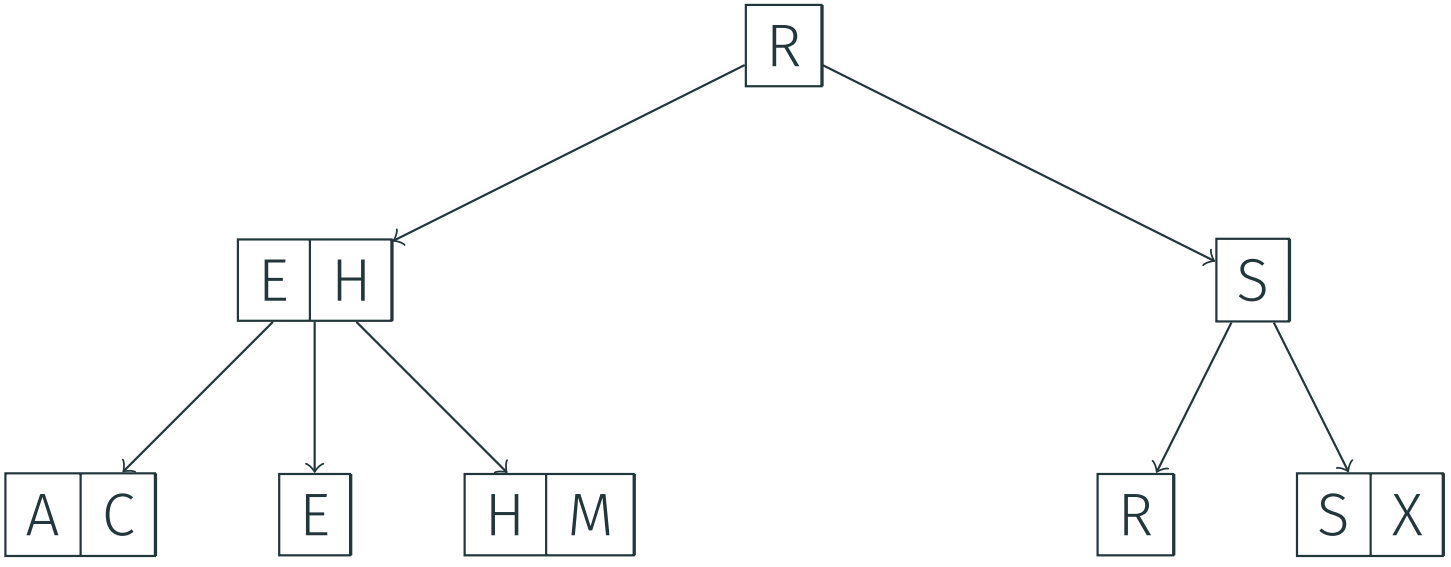
UFABC

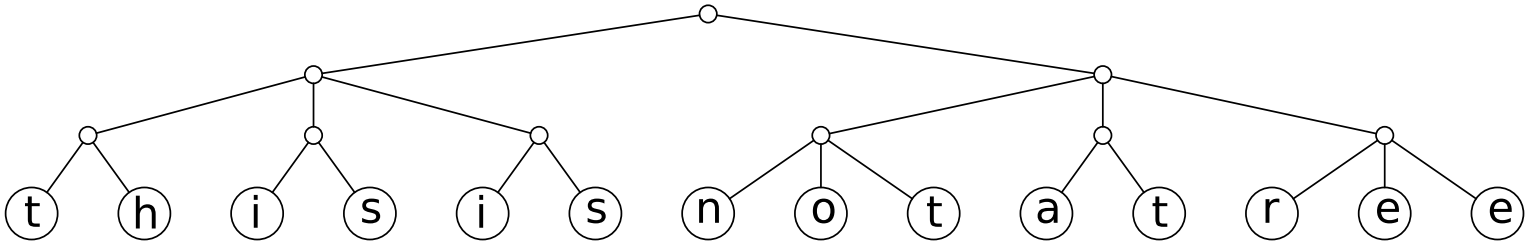
- Experimente a sequência: S, E, A, R, C, H, X, M

Exemplo de inserção Árvores 2-3

UFABC

- Experimente a sequência: S, E, A, R, C, H, X, M

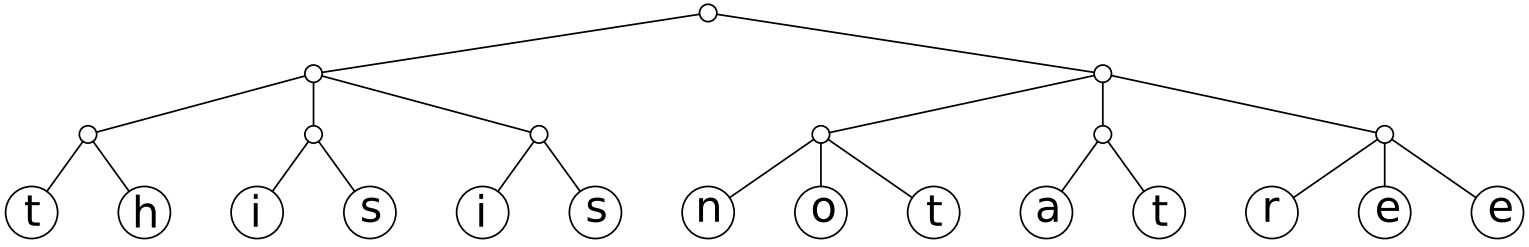




Ok, tomando novamente a nossa figura de exemplo.

- A altura é uniforme e o acesso a qualquer elemento leva tempo $O(\lg n)$ (pois a árvore é balanceada).
- Contudo para algumas aplicações, o acesso às extremidades é muito mais comum do que o acesso aos elementos centrais.
- A intenção aqui é usar a árvore para modelar ao mesmo tempo:
  - Pilhas
  - Filas
  - Deques

UFABC



Poderíamos modelar a árvore da seguinte maneira em Haskell:

```haskell
data Tree a = Zero a | Succ (Tree (Node a))
data Node a = Node2 a a | Node3 a a a
```

Warning

Preste muita atençao ao tipo `Tree`. Ele usa uma técnica chamada de tipo não regular (*non-regular type*). É essencialmente a mesma técnica que usamos para representar os números de Peano. Gaste um tempinho para realmente entender o que está acontecendo e quais são os tipos dos nós da árvore.

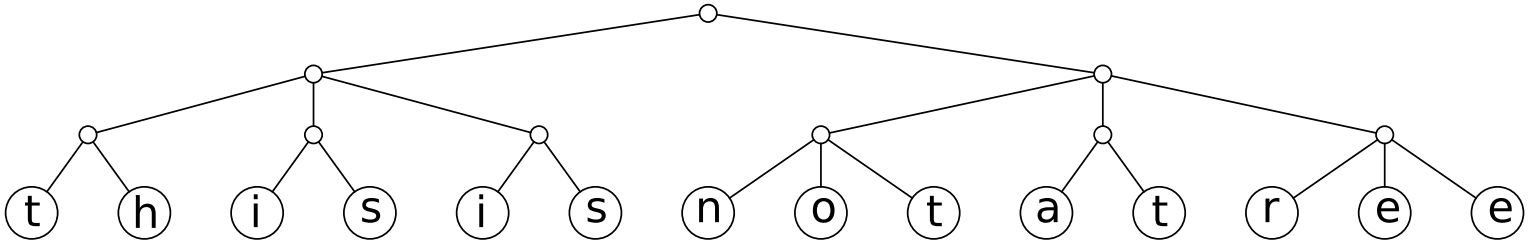
Fingers

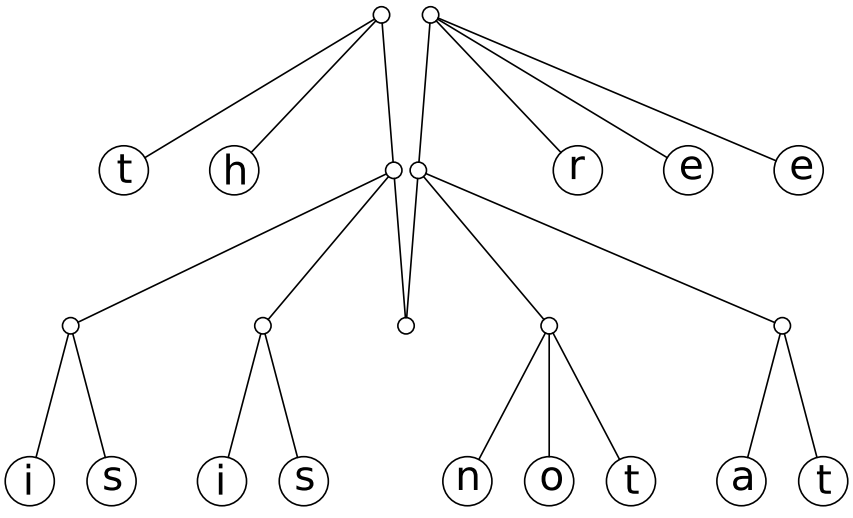
- Uma estrutura que permite acesso eficiente aos nós de uma árvore próximos de uma determinada posição é chamada de Finger.
  - Termo criado por Guibas et al. em 1997.
  - Em uma linguagem imperativa, fingers tomam a forma de um ponteiro.
  - Em uma linguagem funcional, vamos tranformar a nossa estrutura de árvore em algo parecido como que fizemos com Zippers.
- Para dar acesso rápido aos elementos no início e no fim da nossa sequência, vamos instalar fingers nas extremidades direita e esquerda da árvore.

Pendurando a árvore

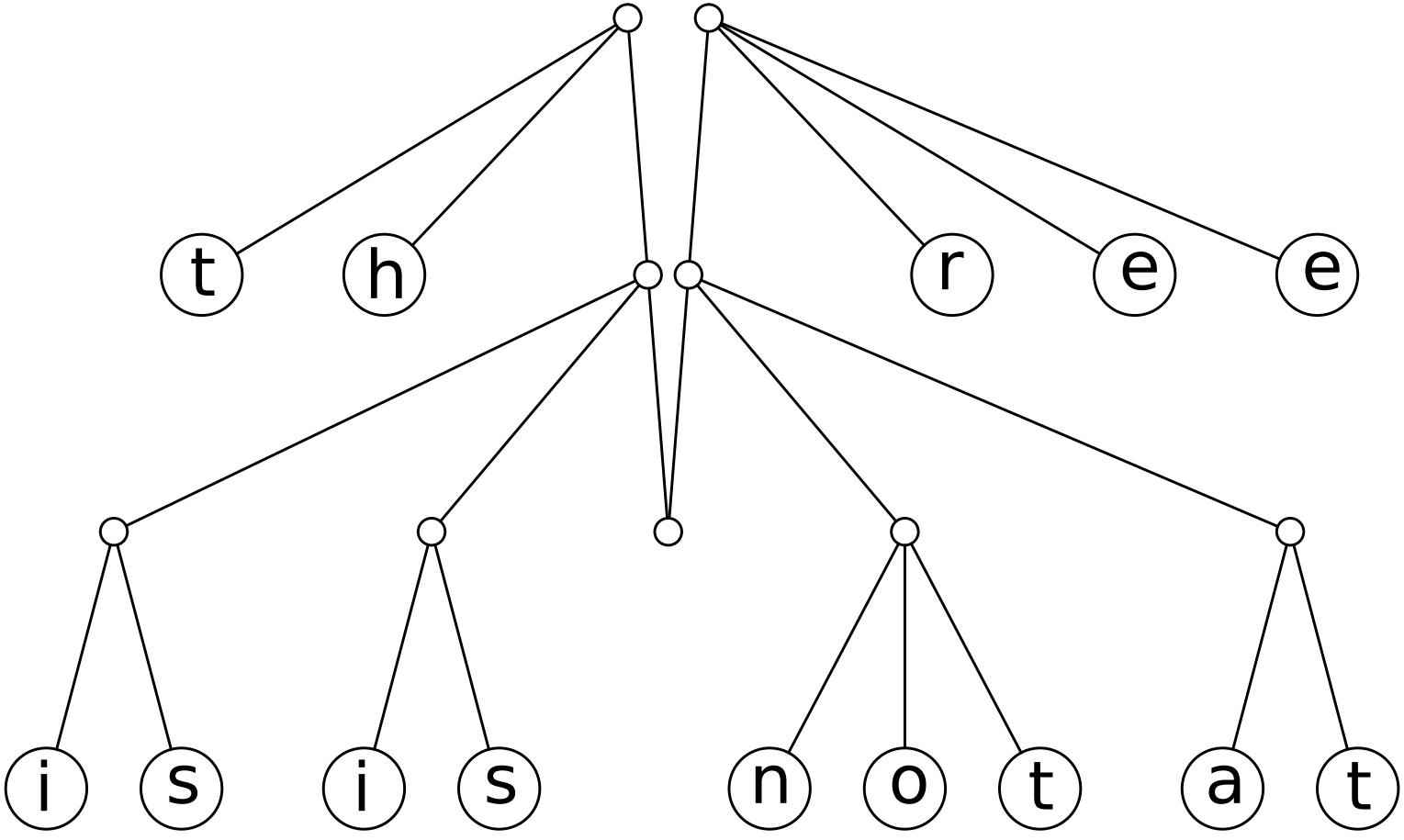
UFABC

- Imagine se pegássemos a árvore de exemplo e a ”pendurássemos”apenas pelas suas extremidades.



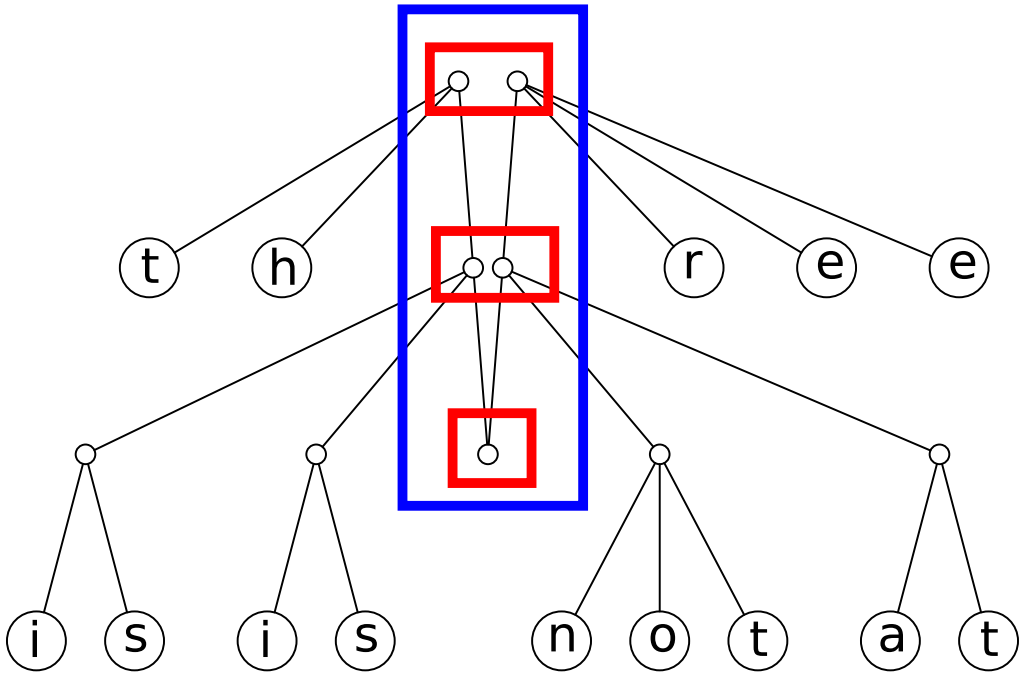


Pendurando a árvore



Note que esse tipo de transformação é possível pois a nossa árvore original era uniforme na altura (como toda árvore 2-3).

A espinha

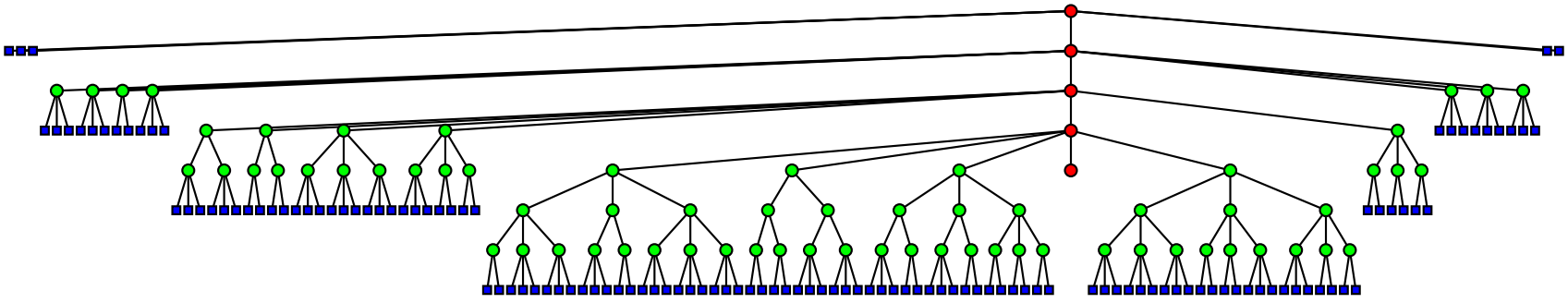


- A ideia em seguida é mesclar os nós destacados em vermelho em um só nó.
- O conjunto dos nós em vermelho forma a espinha da nossa finger tree.
- Note que conforme descemos na espinha, nós estamos na verdade indo das folhas para a raiz!

Algumas árvores maiores

UFABC

No site dos autores do paper da Finger Tree, eles têm figuras de algumas árvores maiores.



Como representar esse estrupício em Haskell?

Representando uma Finger Tree

UFABC

```haskell
data Digit a where
  One   :: a -> Digit a
  Two   :: a -> a -> Digit a
  Three :: a -> a -> a -> Digit a
  Four  :: a -> a -> a -> a -> Digit a

data Node v a where
  Node2 :: a -> a -> Node v a
  Node3 :: a -> a -> a -> Node v a

data FingerTree a where
  Empty :: FingerTree a
  -- Como digits são >0 precisamos tratar
  -- árvores de tamanho 1.
  Single :: a -> FingerTree a
  -- A cada nível da espinha, temos como raiz uma
  -- do tipo FingerTree (Node a)^n
  --      left       spine                    right
  Deep :: Digit a -> FingerTree (Node a) -> Digit a ->
  ↪  FingerTree a
```

Representando uma Finger Tree

UFABC

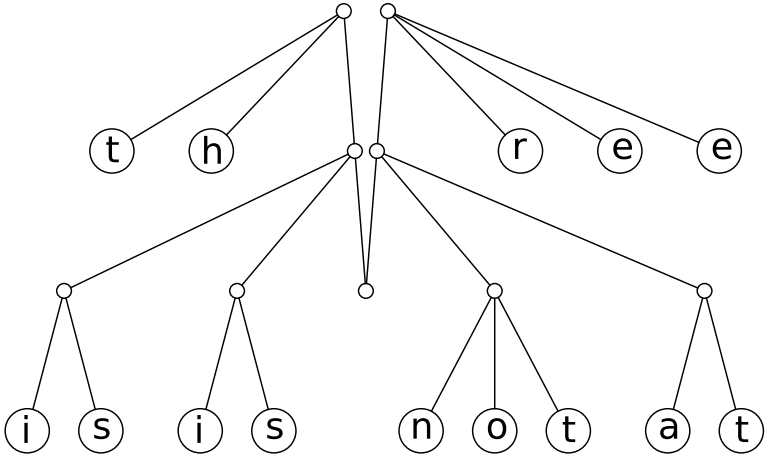
```haskell
data Digit a where
  One   :: a -> Digit a
  Two   :: a -> a -> Digit a
  Three :: a -> a -> a -> Digit a
  Four  :: a -> a -> a -> a -> Digit a
```

- Aqui percebemos a primeira diferença entre Finger Trees e árvores 2-3: é permitido que um nó armazene de 1 a 4 elementos.
- Essa modificação é importante para garantirmos os limites de complexidade amortizados que queremos alcançar.

Tip

O tipo Digit tem esse nome pois Finger Trees se encaixam na categoria de representação numérica [CO] que engloba as estrutura de dados baseadas em um sistema numérico.

Codificando a árvore de exemplo




```haskell
level3 :: FingerTree a
level3 = Empty

level2 :: FingerTree (Node Char)
level2 = Deep l level3 r
  where
    l = Two (Node2 'i' 's')
            (Node2 'i' 's')
    r = Two (Node3 'n' 'o' 't')
            (Node2 'a' 't')

level1 :: FingerTree Char
level1 = Deep l level2 r
  where
    l = Two 't' 'h'
    r = Three 'r' 'e' 'e'

hinzePatterson :: FingerTree
↪  Char
hinzePatterson = level1
```

Deques

Finger Trees como Deques Eficientes

UFABC

- Agora que temos a representação da árvore, vamos ver uma primeira aplicação de Finger Trees como *deques* (*double ended queue*).
- O que queremos é algo melhor em termos de desempenho do que já tínhamos com a implementação usando uma lista ou um par de listas.

Operações em Deques

UFABC

- Deques via de regra dão suporte as seguintes operações de consulta e modificação:
  - `head` - consulta extremidade esquerda
  - `last` - consulta extremidade direita
  - `cons` - adiciona à extremidade esquerda
  - `snoc` - adiciona à extremidade direita
  - `tail` - remove extremidade esquerda
  - `init` - remove extremidade direita

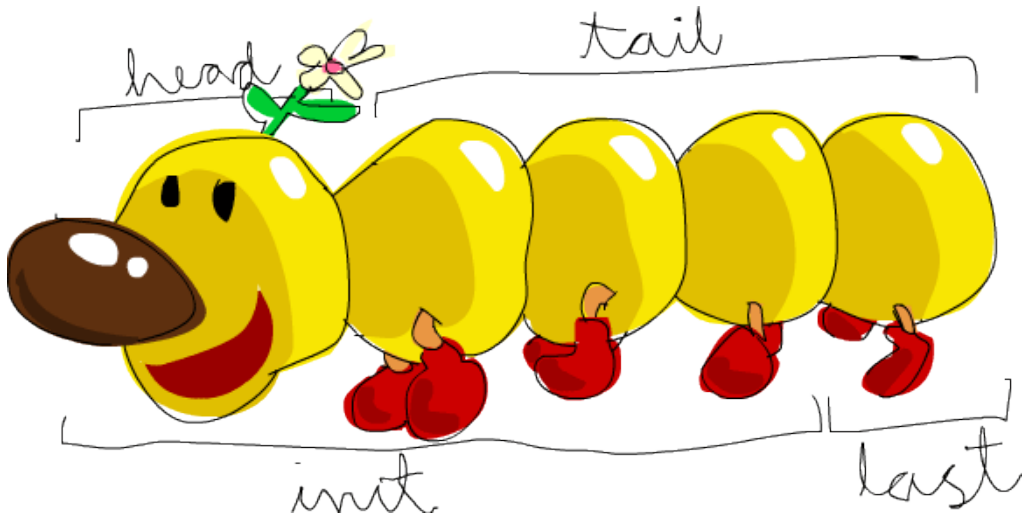


Figura 2: LYAH

cons A.K.A. <|

UFABC

```haskell
-- Imagine o símbolo <| indicando que a operação é na
-- extremidade esquerda
infixr 5 <|
(<|) :: a -> FingerTree a -> FingerTree a
-- Caso inicial e trivial: cria uma árvore single
x <| Empty = Single x
-- Já temos o suficiente para uma árvore de verdade
x <| Single y = Deep (One x) Empty (One y)
-- Adicionar um novo elemento é trivial exceto no caso
-- da árvore já conter 4 elementos. Neste caso,
-- jogamos 3 para um um novo nó e deixamos 2 para trás.
x <| Deep (Four a  b  c  d) s r =
  Deep (Two x a) (Node3 b c d <| s) r
-- Se chegou aqui então tem espaço no nó inicial
x <| (Deep l s r) = Deep (concatL l x) s r
```

Trabalhando com `Digit`

UFABC

Para facilitar a manipulação de `Digit`, criamos algumas funções auxiliares:

```haskell
   -- Equivalente ao cons para Digit
concatL :: Digit a -> a -> Digit a
concatL (One a) x   = Two x a
concatL (Two a b) x = Three x a b
concatL (Three a b c) x = Four x a b c
concatL Four{} _ = error "Digit overflow :)"

-- Equivalente ao snoc para Digit
concatR :: Digit a -> a -> Digit a
concatR (One a) x = Two a x
concatR (Two a b) x = Three a b x
concatR (Three a b c) x = Four a b c x
concatR Four{} _ = error "Digit overflow :)"
```

snoc A.K.A. |>

UFABC

```haskell
-- A implementação de |> é simétrica à de <|. Imagine o
-- símbolo |> indicando que a operação é na extremidade
-- direita. Note que por esta razão a ordem dos
-- parâmetros é invertida.
infixl 5 |>
(|>) :: FingerTree a -> a -> FingerTree a
Empty |> x = Single x
Single y |> x = Deep (One y) Empty (One x)
Deep l s (Four a b c d) |> x =
  Deep l (s |> Node3 a b c) (Two d x)
(Deep l s r) |> x =
  Deep l s (concatR r x)
```

Construindo a partir de uma lista

UFABC

Agora ficou fácil, inclusive, criar uma árvore a partir de uma lista:

```haskell
treeFromList :: [a] -> FingerTree a
treeFromList = foldr (<|) Empty
```

Na verdade, poderíamos usar qualquer `Foldable`! A implementação seria:

```haskell
fromFoldable :: Foldable f => f a -> FingerTree a
fromFoldable = foldr (<|) Empty
```

Qual a complexidade de **cons** e **snoc**?

UFABC

- Uma primeira análise pode chegar a conclusão de que o custo de **cons** e **snoc** é $O(\lg n)$. Mas a história toda é um pouco mais complicada...
- De fato o pior caso é $O(\lg n)$, mas o caso típico ficou muito mais barato!
- Vamos rever o código...

Qual a complexidade de **cons** e **snoc**?

UFABC

```haskell
(<|) :: a -> FingerTree a -> FingerTree a
x <| Empty = Single x -- Caso 1
x <| Single y = Deep (One x) Empty (One y) -- Caso 2
x <| Deep (Four a  b  c  d) s r = -- Caso 3
  Deep (Two x a) (Node3 b c d <| s) r
x <| (Deep l s r) = Deep (concatL l x) s r -- Caso 4
```

- Note que os casos 1, 2 e 4 rodam em tempo constante
- Para uma sequência de operações **cons**, por exemplo, menos de $\frac{1}{2}$ das operações caem no caso 3 (recursivo) no primeiro nível
  - O padrão dos casos para uma sequência de apenas **cons**: 1, 2, 4, 4, 4, 3, 4, 4, 3, 4, 4, 3, ...
- No segundo nível menos de $\frac{1}{4}$
- ...

Logo o custo máximo para uma sequencia de m operações vai ser:

$$T = m + \frac{1}{2}m + \frac{1}{4}m + \frac{1}{8}m + \ldots < \sum_{i=0}^{\infty} \frac{1}{2^i} m = m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

Assim o custo médio por operação é 2 e o custo amortizado por operação é $O(1)$.

Sobre nossa análise

UFABC

- A verdade é que essa análise que fizemos é muito da safada.
- Para efetivamente (e formalmente) provar o custo amortizado das operações precisamos fazer a análise usando o método do banqueiro ou do físico.
- A análise formal é bem mais complicada e é essencialmente a mesma utilizada por [CO] em sua implementação de deques implícitos.
  - A diferença entre aquela implementação e esta é que nesta são substituídos pares por nós que podem conter 2 ou 3 elementos. Isso nos dará mais liberdade para implementar algumas operações adicionais como veremos adiante.

UFABC

- Já temos implementações de **cons** e **snoc**.
- E, apesar das implementações de **head** e **last** serem triviais, vamos matar tudo junto utilizando Views.

Definimos uma view assim:

```haskell
data View s a =
    Nil -- s vazia, não há foco.
  | ConsV a (s a) -- a é o foco, (s a) o "resto"
  deriving Show
```

E criamos duas funções, uma que devolve a view à direita e outra à esquerda:

```haskell
viewL :: FingerTree v a -> View FingerTree a
viewR :: FingerTree v a -> View FingerTree a
```

Implementando a família toda...

UFABC

- Suponha que tenhamos as funções `viewL` e `viewR` prontas.
- A implementação das operações restantes do nosso deque são triviais:

```haskell
head t =
  case viewL t of
    ConsV x _ -> x
    Nil       ->
      error "head: empty"

init t =
  case viewR t of
    ConsV _ xs -> xs
    Nil        ->
      error "init: empty"
```

```haskell
last t =
  case viewR t of
    ConsV x _ -> x
    Nil       ->
      error "last: empty"

tail t =
  case viewL t of
    ConsV _ xs -> xs
    Nil        ->
      error "tail: empty"
```

Implementando `viewL`

UFABC

Precisaremos de algumas funções auxiliares em `Digit`.

```haskell
-- Primeiro elemento de um Digit
dFirst :: Digit a -> a
dFirst (One a)         = a
dFirst (Two a _)       = a
dFirst (Three a _ _)   = a
dFirst (Four a _ _ _) = a

-- Cauda de um Digit
dTail :: Digit a -> Maybe (Digit a)
dTail One{}            = Nothing
dTail (Two _ b)        = Just $ One b
dTail (Three _ b c)    = Just $ Two b c
dTail (Four _ b c d) = Just $ Three b c d
```

Implementando `viewL`

UFABC

Também precisaremos, em alguns casos, usar os elementos de um `Digit` para construir uma nova árvore.

```haskell
digitToTree :: Digit a -> FingerTree a
digitToTree (One a)         = Empty |> a
digitToTree (Two a b)       = Empty |> a |> b
digitToTree (Three a b c)   = Empty |> a |> b |> c
digitToTree (Four a b c d) = Empty |> a |> b |> c |> d
```

Tip

É um exercício interessante implementar `Digit` como uma instância de `Foldable` e utilizar a função `fromFoldable` que vimos antes no lugar de `digitToTree`!

Implementando `viewL`

UFABC

```haskell
viewL :: FingerTree a -> View FingerTree a
viewL Empty      = Nil -- Caso trivial 1
viewL (Single x) = ConsV x Empty -- Caso trivial 2
-- Caso recursivo, usamos a função auxiliar para
-- "emprestar" elementos da espinha ou de r
viewL (Deep l s r) =
  ConsV (dFirst l) (deepL (dTail l) s r)
```

Reconstruindo a cauda

UFABC

```haskell
-- Recebe o que sobrou de l (que pode ser nada) e
-- devolve uma árvore válida.
-- Empresta da direita caso necessário.
deepL :: Maybe (Digit a) -- Sobra de l
      -> FingerTree (Node a) -- Espinha
      -> Digit a -- r
      -> FingerTree a
-- Se sobrou é trivial, remonta a árvore e devolve
deepL (Just l) s r = Deep l s r
deepL Nothing s r =
  case viewL s of
    Nil -> digitToTree r
    ConsV a s'->
      case a of
        Node2 x y   -> Deep (Two x y) s' r
        Node3 x y z -> Deep (Three x y z) s' r
```

Implementando `viewR`

UFABC

A implementação de `viewR` é simétrica:

```haskell
dLast :: Digit a -> a
dLast (One z)         = z
dLast (Two _ z)       = z
dLast (Three _ _ z)   = z
dLast (Four _ _ _ z)  = z

dInit :: Digit a -> Maybe (Digit a)
dInit One{}           = Nothing
dInit (Two a _)       = Just $ One a
dInit (Three a b _)   = Just $ Two a b
dInit (Four a b c _)  = Just $ Three a b
```

Implementando `viewR`

UFABC

```haskell
viewR :: FingerTree a -> View FingerTree a
viewR Empty        = Nil
viewR (Single x)   = ConsV x Empty
viewR (Deep l s r) =
  ConsV (dLast r) (deepR l s (dInit r))
```

Reconstruindo o prefixo

UFABC

```haskell
deepR :: Digit a
      -> FingerTree (Node a)
      -> Maybe (Digit a)
      -> FingerTree a
deepR l s (Just r) = Deep l s r
deepR l s Nothing =
  case viewR s of
    Nil -> digitToTree l
    ConsV a s'->
      case a of
        Node2 x y   -> Deep l s' (Two x y)
        Node3 x y z -> Deep l s' (Three x y z)
```

Comentários sobre a eficiência de views

UFABC

```haskell
head t =
  case viewL t of
    ConsV x _ -> x
    Nil       ->
      error "head: empty"
```

```haskell
tail t =
  case viewL t of
    ConsV _ xs -> xs
    Nil        ->
      error "tail: empty"
```

- A implementação acima é apropriado apenas para linguagens com avaliação preguiçosa.
  - Em outras linguagens, pode ser interessante criar funções específicas para `head` e `tail` (`last` e `init`) além das funções `viewL/R`.
- Novamente, as operações podem levar $O(\lg n)$ no pior caso. Contudo é possível mostrar que o seu custo amortizado é $O(1)$.

Comentários sobre a eficiência de views

UFABC

- A chave para fazer a análise é classificar o estado dos `Digit` (e `Node`) em seguros ou inseguros.
  - Aqueles com 2 ou 3 elementos são seguros e aqueles com 1 ou 4 inseguros.
- Uma operação no deque propaga apenas se o valor em questão for inseguro.
- Mas após a operação o valor se torna seguro novamente!
- Isso garante que no máximo $\frac{1}{2}$ das operações precisa descer ao próximo nível, $\frac{1}{4}$ ao nível seguinte, ...
- Logo, o custo amortizado é constante.

UFABC

Tip

Para entender como isso ocorre, simule manualmente uma sequência de operações até ocorrer uma propagação.

Em seguida faça a operação oposta (`cons/tail`, `snoc/init`) à operação que desencadeou a propagação.

A árvore volta ao estado anterior? Em outras palavras, `tail (cons x t)` é exatamente igual à `t`?

Catenable Deques

Concatenação

UFABC

- Já estamos acostumados em tomar cuidado com o operador `(++)` em listas ”normais”pela sua complexidade ser $O(n)$.
- Será possível fazer melhor com Finger Trees e nossa implementação de deque?

Uma primeira tentativa

UFABC

- Considere o código abaixo para concatenar duas Finger Trees.

```haskell
-- Qual é a complexidade de naiveConcat?
naiveConcat :: FingerTree a -> FingerTree a ->
↪  FingerTree a
naiveConcat l Empty = l
naiveConcat l r =
  naiveConcat (l |> x) xs
  where
    ConsV x xs = viewL r
```

Uma primeira tentativa

UFABC

- Considere o código abaixo para concatenar duas Finger Trees.

```haskell
-- Qual é a complexidade de naiveConcat?
naiveConcat :: FingerTree a -> FingerTree a ->
↪  FingerTree a
naiveConcat l Empty = l
naiveConcat l r =
  naiveConcat (l |> x) xs
  where
    ConsV x xs = viewL r
```

Esse código não é mais rápido do que concatenar duas listas com `(++)`!

Implementando uma versão mais rápida

UFABC

- É usual em Haskell utilizar o operador `(><)` para indicar a concatenação.
  - A diferença entre `(++)` e `(><)` é que, por convenção, a implementação de `(><)` tem um desempenho melhor do que linear.
- Vamos implementar `(><)` em função de uma outra função um tanto esquisita: `concat3`.

```
1 concat3 :: FingerTree a     -- Árvore à esquerda
2         -> Maybe (Digit a) -- Elementos intermediários
3         -> FingerTree a     -- Àrvore à direita
4         -> FingerTree a
5 concat3 = ???
```

Implementando uma versão mais rápida

UFABC

Supondo que `concat3` esteja pronta e disponível, a implementação de `(><)` é trivial:

```haskell
(><) :: FingerTree a -> FingerTree a -> FingerTree a
l >< r = concat3 l Nothing r
```

- Mas só empurramos a sujeira para debaixo do tapete. Vamos ver como implementar `concat3` eficientemente.

UFABC

Primeiro os casos fáceis de `concat3`:

```haskell
concat3 :: FingerTree a -> Maybe (Digit a) -> FingerTree
↪  a -> FingerTree a
-- Casos triviais à esquerda e à direita
concat3 Empty Nothing  r = r
concat3 Empty (Just d) r =
  dFirst d <| concat3 Empty (dTail d) r
concat3 (Single y) md r =
  y <| concat3 Empty md r
concat3 l Nothing  Empty = l
concat3 l (Just d) Empty =
  concat3 l (dInit d) Empty |> dLast d
concat3 l md (Single y) =
  concat3 l md Empty |> y
-- Caso cabeludo: ambas as árvores são profundas
concat3 (Deep l0 s0 r0) md (Deep l1 s1 r1) = ???
```

concat3 - Caso cabeludo

UFABC

```haskell
...
concat3 (Deep l0 s0 r0) md (Deep l1 s1 r1) =
  Deep l0 nextLevel r1
  where
    -- Chamada recursiva para gerar o próximo nível
    nextLevel = concat3 s0 dNodes s1
    -- Pega a lista dos elementos da direita de l
    -- Pega a lista dos elementos da esquerda de r
    -- Converte esses elementos em nós para passar para
    ↪  concat3.
    dNodes = Just $ nodes r0 md l1
```

- Quantos elementos há no máximo na junção de `r0`, `md` e `l1`?

A função nodes

UFABC

```haskell
nodes :: Digit a -> Maybe (Digit a) -> Digit a -> Digit
  ↪  (Node a)
nodes l mm r =
  -- dJoin junta nds em até no máximo um Four
  let [dig] = dJoin nds in dig
  where
    (l', r') = (dToList l, dToList r)
    mm' = maybe [] dToList mm
    -- l' mm' e r' têm no máximo 12 elementos
    nds = nodes' $ l' ++ mm' ++ r'
    -- nodes' têm no máximo 4 elementos. Pq?
    nodes' []  = undefined -- l e m juntos tem com
    nodes' [_] = undefined -- certeza >= 2 elementos
    nodes' [a, b]       = [One $ node2 a b]
    nodes' [a, b, c]    = [One $ node3 a b c]
    nodes' [a, b, c, d] = [Two (node2 a b) (node2 c d)]
    nodes' (a:b:c:xs) = (One $ node3 a b c) : nodes' xs
```

Complexidade de `(><)`

UFABC

- A complexidade de `(><)` é, no final, a mesma de `concat3`
- Os casos triviais de `concat3` claramente rodam em tempo constante.
- Resta o caso cabeludo.
  - Note que no caso cabeludo *consumimos* um nível de cada árvore a cada chamada. Assim faremos $\lg(\min\{m, n\})$ chamadas recursivas (onde n e m é o número de elementos nas árvores sendo concatenadas).
  - O restante roda em tempo constante.
- Assim, a complexidade (*real*) de `(><)` é $O(\lg(\min\{m, n\}))$.

Monoides

Na matemática um semigrupo (en: *semigroup*) é um conjunto G dotado de uma operação binária $\diamond$ para a qual valem as seguintes propriedades:

- G é fechado em $\diamond$: $\forall a, b \in G \implies (a \diamond b) \in G$
- $\diamond$ é associativo: $\forall a, b, c \in G$ vale $(a \diamond b) \diamond c = a \diamond (b \diamond c)$

Em Haskell um conjunto de valores é representado por um tipo e o semigrupo é definido pela seguinte *typeclass*:

```haskell
class Semigroup a where
  -- Note que "acreditamos" que <> é associativa.
  (<>) :: a -> a -> a
```

UFABC

Um Monoide (en: *monoid*) é um conjunto de valores associados a um operador binário associativo e um elemento identidade:

- Valores inteiros com o operador `+` e o elemento `0`
- Valores inteiros com o operador `*` e o elemento `1`
- Valores String com o operador `++` e o elemento `''`

A classe **Monoid** é definida como:

```haskell
class Semigroup a => Monoid a where
  mempty  :: a -- devolve o elemento identidade
```

Monoide de Listas

UFABC

Para listas temos a seguinte instância de `Monoid`:

```haskell
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty  = []
```

UFABC

Para o tipo Maybe:

```haskell
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> b       = b
    a       <> Nothing = a
    Just a  <> Just b  = Just (a <> b)

instance Semigroup a => Monoid (Maybe a) where
    mempty = Nothing
```

Foldable

UFABC

- A importância dos monoides está na generalização em como combinar uma lista de valores de um tipo que pertença a essa classe.
- Sabendo que o tipo `a` é um `Monoid`, podemos definir:

```haskell
fold :: Monoid a => [a] -> a
fold []     = mempty
fold (x:xs) = x <> fold xs
```

Dobrando uma árvore

UFABC

Essa generalização pode ser feita para outras estruturas:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

fold :: Monoid a => Tree a -> a
fold (Leaf x)   = x
fold (Node l r) = fold l <> fold r
```

Classe `Foldable`

UFABC

Podemos então criar a classe dos ”dobráveis”:

```haskell
class Foldable t where
   fold    :: Monoid a => t a -> a
   foldMap :: Monoid b => (a -> b) -> t a -> b
   foldr   :: (a -> b -> b) -> b -> t a -> b
   foldl   :: (a -> b -> a) -> a -> t b -> a
```

Exemplo prático

UFABC

Considere os seguintes tipos:

```haskell
newtype Sum a = Sum a
   deriving (Eq, Ord, Show, Read)

newtype Prod a = Prod a
   deriving (Eq, Ord, Show, Read)

getSum :: Sum a -> a
getSum (Sum x) = x

getProd :: Prod a -> a
getProd (Prod x) = x
```

UFABC

Considere os seguintes `Monoids`:

```haskell
instance Num a => Semigroup (Sum a) where
   Sum x <> Sum y = Sum $ x + y
instance Num a => Monoid (Sum a) where
   mempty = Sum 0

instance Num a => Semigroup (Prod a) where
   Prod x <> Prod y = Prod $ x * y
instance Num a => Monoid (Prod a) where
   mempty = Prod 1
```

Exemplo prático

UFABC

Para efetuar a somatória e produtória de uma lista de números basta fazer:

```
1  > getSum (foldMap Sum [1..10])
2  55
3
4  > getProd (foldMap Prod [1..10])
5  3628800
```

Exemplo prático

UFABC

Se definirmos a instância de `Foldable` para o tipo `Tree`, bastaria fazer:

```
1 > getSum (foldMap Sum arvore)
2 > getProd (foldMap Prod arvore)
```

As funções são as mesmas!!!

UFABC

- Em Haskell, tipos que possuem elementos que podem ser comparados são da classe de tipo `Ord`.
- Entre outros define a função `compare` que devolve um elemento do tipo `Ordering` que é definida como abaixo:

```haskell
data Ordering = LT | GT | EQ
```

Monoide `Ordering`

UFABC

```haskell
data Ordering = LT | GT | EQ
```

Se definirmos o monoide para `Ordering` como abaixo podemos escrever funções de comparação muito facilmente:

```haskell
instance SemiGroup Ordering where
  EQ  <> ord = ord
  ord <> _   = ord

instance Monoid Ordering where
  mempty = EQ
```

Monoide **Ordering** - Exemplo

UFABC

```haskell
data Vestibulando = Vestibulando {
    nome :: String,
    idade :: Int,
    nota :: Int
  }

instance Ord Vestibulando where
  compare vest1 vest2 =
    compare (nota vest1) (nota vest2) <>
    compare (idade vest1) (idade vest2) <>
    compare (nome vest1) (nome vest2)
```

Outros monoides interessantes

UFABC

- `All` - `Bool` com a operação `(&&)` e elemento identidade `True`
- `Any` - `Bool` com a operação `(||)` e elemento identidade `False`
- `First` - `Maybe` com a operação que devolve o primeiro `Just` e elemento identidade `Nothing`
- `Last` - `Maybe` com a operação que devolve o último `Just` e elemento identidade `Nothing`

- O MapReduce[1] é um modelo de programação utilizado largamente em clusters.
  - Google, Amazon, Yahoo! e outros estão entre seus grandes utilizadores
- Muitas ferramentas como Apache Hadoop e Apache Spark são baseadas neste modelo
- O MapReduce nada mais é que a função **`foldMap`**
  - O *Map* é feito em paralelo e de maneira distribuída (o que é fácil já que o map é puro)
  - O *Reduce* corresponde ao *fold*
  - Como a redução ou folding é feito em uma ordem arbitrária é imprescindível que se trabalhe utilizando um monoide abeliano[2] para que o resultado seja consistente!

[1] https://en.wikipedia.org/wiki/MapReduce

[2] https://en.wikipedia.org/wiki/Monoid#MapReduce

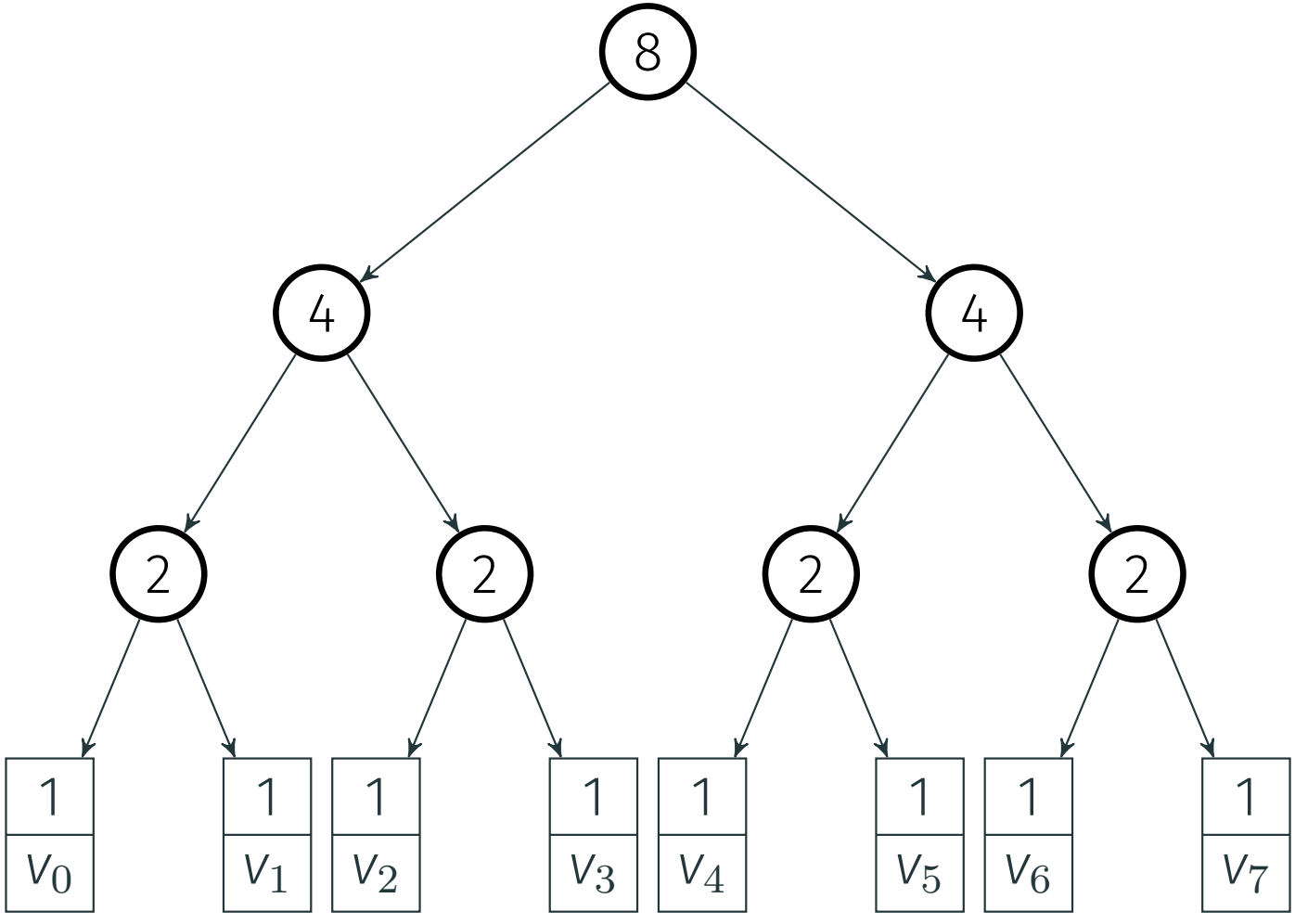
Finger Tree Splitting

Listas indexadas

UFABC

- Neste ponto já temos estruturas de dados capazes de efetuar o trabalho de deques e concatenação.
- Ainda falta algo! E a indexação de um elemento pelo seu índice?
- Listas oferecem a função `(!!)`, porém a sua complexidade é $O(n)$.
- Como resolver? É óbvio: Finger Trees!
  - Mas antes, vamos ter um cheiro da ideia...

Um cheiro da ideia...



- Como usar as anotações para acessar o elemento *i*?

UFABC

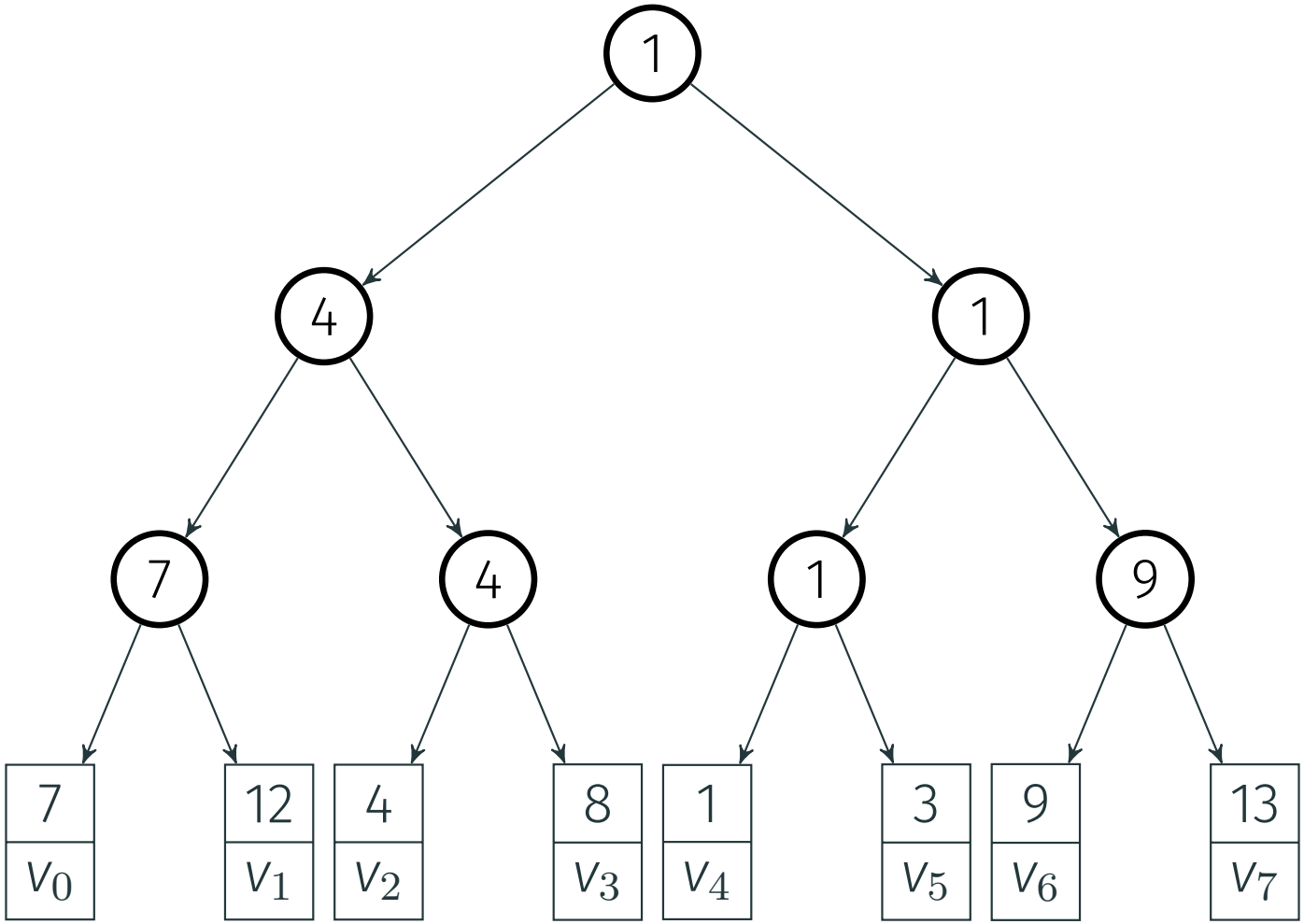
O pseudo-algoritmo ficaria com a seguinte cara:

```haskell
(!!) :: Tree a -> Int -> a
(Leaf _ a)       !! 0 = a
(Branch _ x y)   !! n
      | n < tag x      = x !! n
      | otherwise      = y !! (n - tag x)
```

Eu sinto cheiro de Heaps?

UFABC

- E se em vez de colocarmos a soma do número de elementos nós colocássemos o valor mínimo?



- Como achar o mínimo deste min-heap?

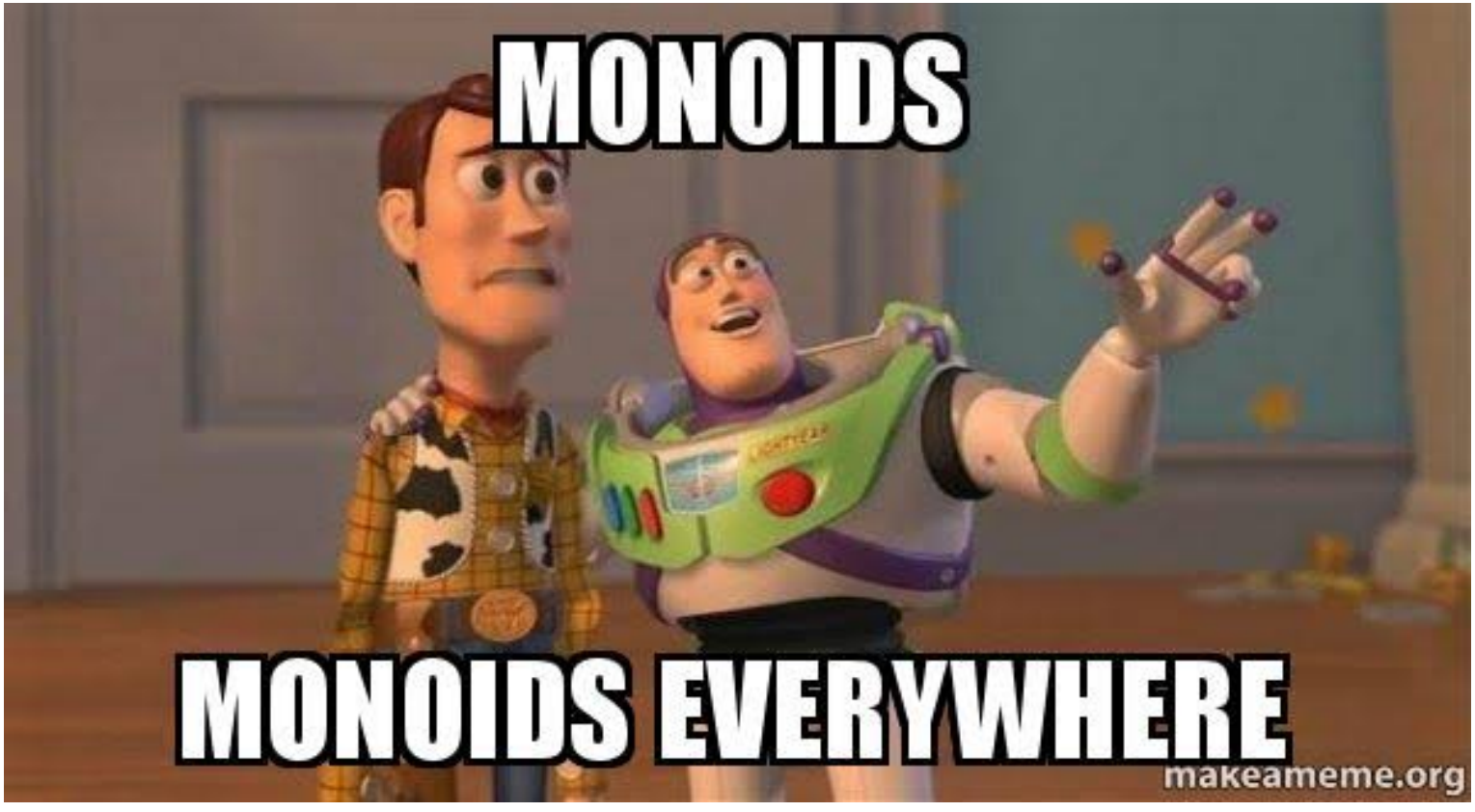
UFABC

Novamente, o pseudo-código ficaria com a seguinte cara:

```haskell
minimo :: Tree a -> a
minimo t = go t
    where
    go (Leaf _ a)          = a
    go (Branch _ x y)
        | tag x == tag t = go x
        | tag y == tag t = go y
```

Monoids to the rescue!

UFABC



UFABC

- Usamos a mesma estrutura e apenas modificando como anotamos os nós da árvore fomos capazes de criar uma sequência indexável e também um heap!
- O que as operações tem em comum é que ambas `Int` com `(+)` e `Int` com `min` formam monoides!

```haskell
-- Sequência indexável
instance Semigroup Size where
  (<>) = (+)
instance Monoid Size where
  mempty = 0
```

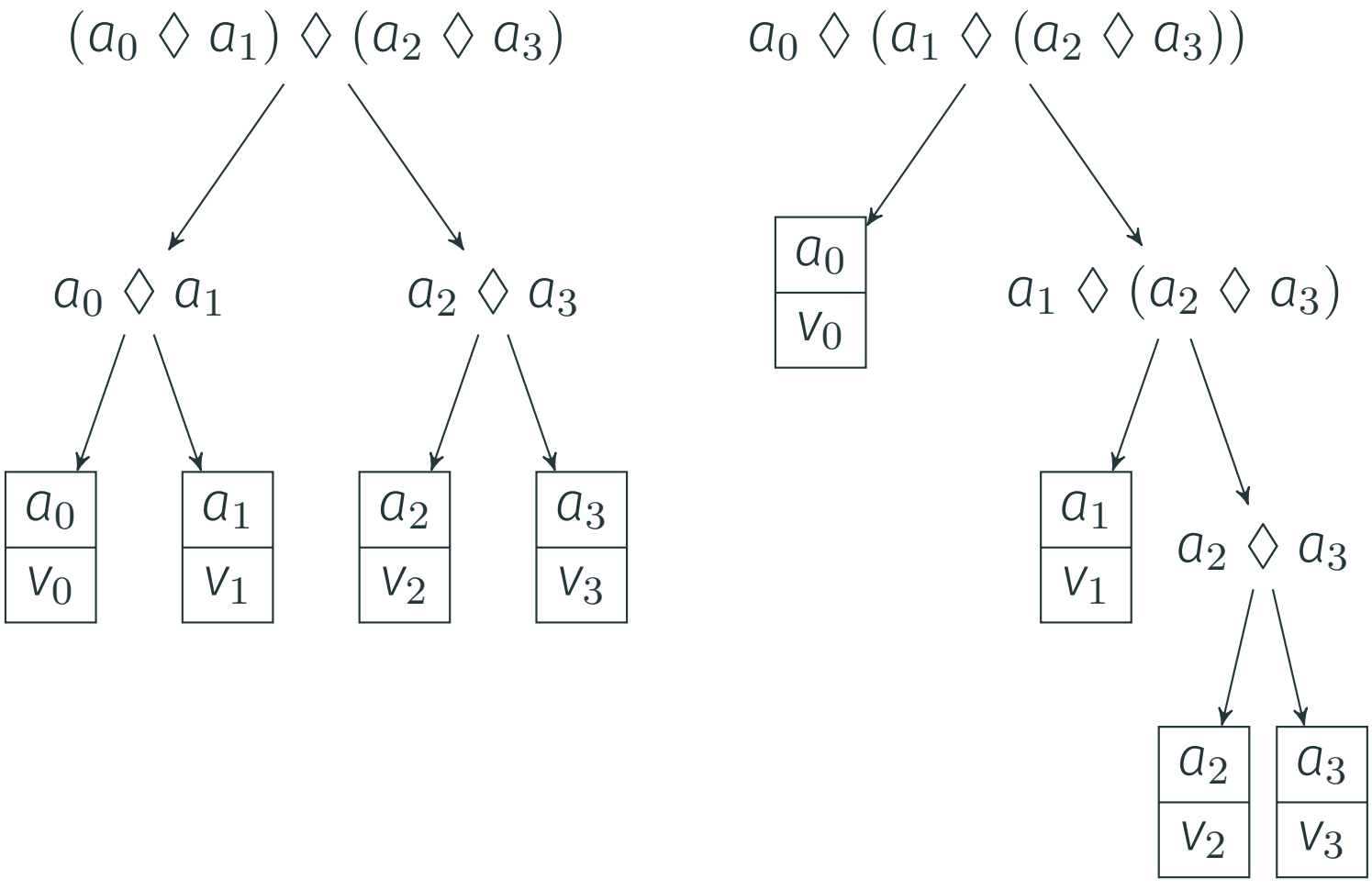
```haskell
-- Heap
instance Semigroup Priority
↪  where
  (<>) = min
instance Monoid Priority where
  mempty = maxBound
```

A vantagem do monoide

UFABC

- No caso geral, como a anotação na raiz da árvore se relaciona com os elementos até as folhas?
- De fato, pela operação `(<>)` ser associativa (garantida por nossas anotações serem monoides) pouco importa o formato da árvore.
  - Podem até ser Finger Trees!
  - Só precisamos ter cuidado pois nem todo monoide é abeliano, então temos que nos precaver para operações não comutativas.

A forma da árvore não importa



Como (<>) é associativo temos:

$$(v1 \Diamond v2) \Diamond (v3 \Diamond v4) = v1 \Diamond (v2 \Diamond (v3 \Diamond v4)) = v1 \Diamond v2 \Diamond v3 \Diamond v4$$

Implementando a busca

UFABC

- O que queremos no final é uma operação que funcione ao mesmo tempo para os dois monoides que definimos.
- Os algoritmos que descrevemos são similares no sentido que a cada nó eles descem na subárvore dependendo da anotação.
- Lembrando que $(\diamond)$ é associativo, considere a seguinte sequência de anotações:

$$a_1, a_2, a_3, a_4, \ldots, a_n$$

Indexando

UFABC

- Como podemos fazer para, por exemplo, encontrar o 3º elemento?
  - Varrermos a lista da esquerda para a direita e acumulamos 1 por cada elemento que passamos
  - Quando a contagem ultrapassar 3, sabemos que encontramos o elemento procurado

```
Elemento 0: 1                   -- > 3 False
Elemento 1: 1 + 1               -- > 3 False
Elemento 2: 1 + 1 + 1           -- > 3 False
Elemento 3: 1 + 1 + 1 + 1       -- > 3 True
...                             -- > 3 True
```

UFABC

- De maneira semelhante, como encontrar o elemento com a menor prioridade p (obtida do nó raiz)?
  - Varremos a lista da esquerda para a direita e mantemos a prioridade mínima por cada elemento que passamos
  - Quando a prioridade for igual a p sabemos que achamos o elemento

```
p1                                  -- == p False
p1 `min` p2                         -- == p False
p1 `min` p2 `min` p3                -- == p False
p1 `min` p2 `min` p3 `min` p4       -- == p True
...                                 -- == p True
```

O algoritmo genérico de busca

Em outras palavras, estamos procurando a posição onde um predicado `p` troca de `False` para `True`.

```
measure a1                                                  -- ¬ p
measure a1 <> measure a2                                    -- ¬ p
measure a1 <> measure a2 <> measure a3                      -- ¬ p
measure a1 <> measure a2 <> measure a3 <> measure a4        -- p
...                                                         -- p
```

Queremos *k* tal que:

```
p (measure a1 <> ... <> measure ak)                     -- False
p (measure a1 <> ... <> measure ak <> measure a(k+1)) -- True
```

E a árvore?

UFABC

- A chave para resolver o problema é se dar conta que não precisamos olhar elemento a elemento.
- A nossa árvore já contém o que precisamos calculado nos nós intermediários!
  - Podemos usar busca binária!

Pseudo-código:

```
1  search :: Measured a v => (v -> Bool) -> Tree v a ->
   ↪  Maybe a
2  search p t
3      | p (measure t) = Just (go mempty p t)
4      | otherwise     = Nothing
5      where
6      go i p (Leaf _ a) = a
7      go i p (Branch _ l r)
8          | p (i <> measure l) = go i p l
9          | otherwise          = go (i <> measure l) p r
```

Sobre o predicado

- Para isto tudo funcionar precisamos que o predicado `p` atenda às seguintes condições:
  - `p mempty == False`
  - Se `p x == True` então `p (x <> y)` também deve ser `True` independentemente do valor de y.
    - `p` precisa ser um predicado monótono.
    - Nosso 2 exemplos `(> 3)` e `(== mínimo)` satisfazem a propriedade juntamente com a respectiva `(<>)` de seus monoides.

```
1  t !! k    = search (> k)
2  minimo t = search (== measure t)
```

Incrementando nossa estrutura de dados

UFABC

- Pra fazer isso tudo vamos colocar um novo campo na árvore para anotar com o valor do monoid utilizado.
  - Essa informação estará presente também em todos seus nós.
- Também vamos criar uma nova *typeclass* que sabe acessar esse valor. Vamos chamar essa *typeclass* de `Measured`

```
1 class Monoid v => Measured a v | a -> v a where
2   measure :: a -> v
```

- Note que exigirmos que a medida seja um monoide!
- Isso é essencial para quando tivermos que reorganizar a árvore podermos garantir o resultado nos níveis mais altos apenas com operações locais.

Measured para FingerTrees

UFABC

```haskell
data FingerTree v a where
  Empty :: Measured a v => FingerTree v a
  Single :: Measured a v => a -> FingerTree v a
  Deep :: Measured a v => v -> Digit a -> FingerTree v
  ↪  (Node v a) -> Digit a -> FingerTree v a

instance Monoid v => Measured (FingerTree v a) v where
  measure Empty           = mempty
  measure (Single x)      = measure x
  measure (Deep v _ _ _) = v
```

Também precisamos criar instâncias de Measure para Node e Digit!

Measured para Node

UFABC

```haskell
data Node v a where
  Node2 :: Monoid v => v -> a -> a -> Node v a
  Node3 :: Monoid v => v -> a -> a -> a -> Node v a

instance Monoid v => Measured (Node v a) v where
  measure (Node2 v _ _) = v
  measure (Node3 v _ _ _) = v
```

Measured para Digit

UFABC

```haskell
instance (Monoid v, Measured a v) => Measured (Digit a)
  ↪  v where
  measure (One a) = measure a
  measure (Two a b) = measure a <> measure b
  measure (Three a b c) = measure a <> measure b <>
   ↪  measure c
  measure (Four a b c d) = measure a <> measure b <>
   ↪  measure c <> measure d
```

Tip

Defina uma instância de **Foldable** para **Digit** e implemente **measure** como um fold!

UFABC

Como agora os `Node` e `FingerTree` guardam um valor, podemos escrever funções auxiliares para a criação destes tipos que preenchem a anotação automaticamente:

```
1  node2 a b = Node2 (measure a <> measure b) a b
2
3  node3 a b c =
4    Node3 (measure a <> measure b <> measure c) a b c
5
6  deep l s r =
7    Deep (measure l <> measure s <> measure r) l s r
```

E, é claro, precisamos alterar todas as assinaturas e locais onde árvores e nós são criados para estarem conforme esta nova definição.[3]

[3] O código completo com essas modificações está na página do curso.

Tá, e daí?

UFABC

- Fizemos um monte de ginástica, mas cade o resultado?
- Toda essa trabalheira vai se pagar em breve, primeiro vamos ver como podemos implementar uma função de busca utilizando essas anotações.
- Como já deve dar para adivinhar, nossa busca vai receber um predicado que vai operar de alguma forma nas nossas anotações monoidais.
- Também precisamos que nossa função seja capaz de retirar o elemento buscado da árvore.

O conjunto de funções `split`

UFABC

- A função `split` e variações devolvem uma tupla na forma `(antes, x, depois)`.
  - `antes` é uma finger tree que contém todos os elementos da finger tree alvo da busca antes do predicado se tornar verdadeiro.
  - `x` contém o valor do elemento que tornou o predicado verdadeiro.
  - `depois` contém todos os elementos após x que estavam na árvore original.

A função `split`

UFABC

Vamos começar com o `splitNode`:

```haskell
-- A função só faz sentido se "p node" for verdadeiro
splitNode :: (Measured a v)
          => (v -> Bool) -> v -> Node v a
          -> (Maybe (Digit a), a, Maybe (Digit a))
splitNode p i (Node2 _ a b)
  | p va          = (Nothing, a, Just (One b))
  | otherwise     = (Just (One a), b, Nothing)
  where
    va       = i <> measure a
splitNode p i (Node3 _ a b c)
  | p va          = (Nothing, a, Just (Two b c))
  | p vab         = (Just (One a), b, Just (One c))
  | otherwise     = (Just (Two a b), c, Nothing)
  where
    va       = i <> measure a
    vab      = va <> measure b
```

A função `split`

UFABC

```haskell
-- A função só faz sentido se "p d" for verdadeiro
splitDigit :: Measured a v
           => (v -> Bool)
           -> v
           -> Digit a
           -> (Maybe (Digit a), a, Maybe (Digit a))
splitDigit _ _ (One d) = (Nothing, d, Nothing)
splitDigit p i d
  | p i'       = (Nothing, a, as)
  | otherwise =
    let (l, x, r) = splitDigit p i' (fromJust as)
    in  (Just (maybe (One a) (`concatL` a) l), x, r)
  where
    a  = dFirst d
    i' = i <> measure a
    as = dTail d
```

A função `split`

UFABC

```haskell
-- A função só faz sentido se "p tree" for verdadeiro
splitTree :: Measured a v
          => (v -> Bool) -> v -> FingerTree v a
          -> (FingerTree v a, a, FingerTree v a)
splitTree _ _ Empty = error "split em árvore vazia"
splitTree _ _ (Single x) = (Empty, x, Empty)
splitTree p i (Deep _ pr m sf)
  | p vpr     = let (l, x, r) = splitDigit p i pr
                in  (maybe Empty digitToTree l, x, deepL r m sf)
  | p vm      = let (ml, xs, mr) = splitTree p vpr m
                    (l, x, r) = splitNode p (vpr <> measure ml)
                    ↪  xs
                in  (deepR pr ml l, x, deepL r mr sf)
  | otherwise = let (l, x, r) = splitDigit p vm sf
                in  (deepR pr m l, x, maybe Empty digitToTree r)
  where
    vpr = i   <> measure pr
    vm  = vpr <> measure m
```

A função `split`

UFABC

A função `split` apenas combina o resultado de `splitTree` para comodidade de alguns usos.

```
1  split ::  (Measured a v)
2        => (v -> Bool)
3        -> FingerTree v a
4        -> (FingerTree v a, FingerTree v a)
5  split _ Empty  =  (Empty, Empty)
6  split p xs
7    | p (measure xs) =  (l, x <| r)
8    | otherwise    =  (xs, Empty)
9    where
10     (l, x, r) = splitTree p mempty xs
```

Complexidade de `split`

UFABC

- As funções `splitDigit` e `splitNode` são obviamente $O(1)$.
- A complexidade da função `splitTree` depende da localização do elemento buscado. Sua complexidade é $O(\lg n)$, ou mais precisamente $O(\lg \min\{i, n - i\})$ onde i é o índice do elemento buscado.

Listas indexáveis

Listas indexáveis - Tipos básicos

UFABC

```haskell
-- Representa os elementos da lista
newtype SeqV a = SeqV {getSeqV :: a}

-- Tipo monoidal que acumula o tamanho da lista
newtype Size = Size {getSize :: Int} deriving (Eq, Ord)

-- Tipo para facilitar o uso da árvore
newtype Seq a = Seq (FingerTree Size (SeqV a))
```

```haskell
-- Conforme mostramos, precisamos de um monoide
-- com a operação (+) e elemento identidade 0.
instance Semigroup Size where
  (Size x) <> (Size y) = Size $ x + y

instance Monoid Size where
  mempty = Size 0

instance Measured (SeqV a) Size where
  measure _ = Size 1
```

Listas indexáveis - Funções

UFABC

```haskell
-- O(1)
seqLength :: Seq a -> Int
seqLength (Seq ft) = getSize $ measure ft

-- O(lg(min(i, n - i)))
seqSplitAt :: Int -> Seq a -> (Seq a, Seq a)
seqSplitAt i s@(Seq ft)
  | i <= 0           = (Seq Empty, Seq ft)
  | seqLength s < i  = (Seq ft, Seq Empty)
  | otherwise        = (Seq l, Seq r)
  where
    (l, r) = split (Size i <) ft

-- O(lg(min(i, n - i)))
(!) :: Seq a -> Int -> a
Seq ft ! i = x
  where
    (_, SeqV x, _) = splitTree (Size i < ) mempty ft
```

Heaps

UFABC

```haskell
  -- A finger tree armazena os itens com suas
  ↪  prioridades
data HeapV a = HeapV {
  priority :: Int,
  item :: a
  }

-- PositiveInfinity é o elemento identidade
data Priority = Priority Int | PositiveInfinity
  deriving (Eq, Ord, Show)

newtype Heap a = Heap (FingerTree Priority (HeapV a))
```

UFABC

```haskell
instance Semigroup Priority where
  -- PositiveInfinity é tradado como o menos prioritário de todos
  PositiveInfinity <> x                = x
  x                <> PositiveInfinity = x
  (Priority x)     <> (Priority y)     = Priority $ min x y

instance Monoid Priority where
  mempty = PositiveInfinity

instance Measured (HeapV a) Priority where
  measure = Priority . priority
```

UFABC

```haskell
heapPush :: Heap a -> a -> Int -> Heap a
heapPush (Heap ft) x prio = Heap $ HeapV prio x <| ft

heapPop :: Show a => Heap a -> (a, Heap a)
heapPop (Heap ft) = (x, Heap $ l >< r)
  where
    (l, HeapV _ x, r) = splitTree (measure ft >=) mempty
    ↪  ft

heapToOrderedList :: Show a => Heap a -> [a]
heapToOrderedList (Heap Empty) = []
heapToOrderedList h = a : heapToOrderedList h'
  where
    (a, h') = heapPop h
```

Outras estruturas de dados

Outros monoides interessantes

UFABC

- Outras estruturas de dados interessantes podem ser criadas da finger tree apenas pela definição de um novo monoide e predicado.
- Exemplos:
  - Listas ordenadas - Permitem a inserção e remoção de elementos em uma lista em ordem.
    - Usa monoide que sempre devolve o segundo operando.
    - Measure é o próprio valor do elemento.
  - Árvores de intervalos - permitem a busca de intervalos que se sobrepõem a um intervalo passado como parâmetro
    - Usa monoide ”duplo”, com valores do início e do fim dos intervalos.
    - Measure varia em função de se querer buscar intervalos que intersectam no início, no fim ou ambos.

Comentários finais

Comentários finais

- Finger trees são muito interessantes pois permitem que criemos novas estruturas de dados de uma maneira muito simples.
- A complexidade das estruturas de dados criadas são boas e em alguns casos melhores do que soluções do dia a dia.
- Tome como exemplo o heap baseado em finger trees. Apesar das complexidades serem equivalentes àquelas das melhores implementações, as constantes escondidas na notação O ainda são maiores. Então se justifica usar estruturas dedicadas em alguns casos.

Referências

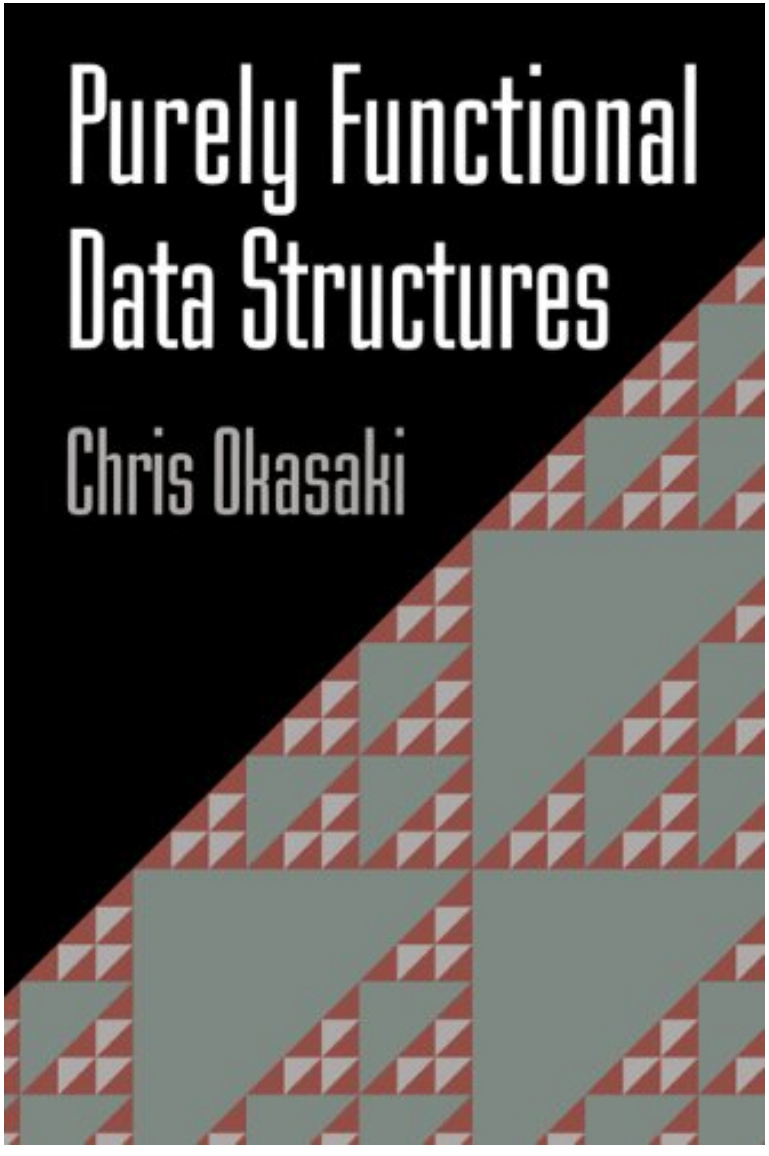
Outros

Este material foi fortemente baseado nas seguintes fontes:

- ”Finger Trees: A Simple General-purpose Data Structure”, por Ralf Hinze and Ross Paterson, Journal of Functional Programming 16(2):197–217, 2006.
- ”Finger Trees”, por Andrew Gibiansky
- ”Monoids and Finger Trees”, por Heinrich Apfelmus

Código

- O código fonte completo dos slides pode ser baixado aqui.
- A versão otimizada e ”oficial”que guarda bastante semelhança ao código didático pode ser vista aqui.



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