Estruturas de dados puramente funcionais

Dia 1

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Outline

Introdução

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE StandaloneDeriving #-}

{-# OPTIONS_GHC -Wno-unticked-promoted-constructors #-}

module Dia01 where

import Prelude hiding ((++))
import Data.Maybe
```

Quando programadores de C, C++, Java, Python, ... precisam de uma estrutura de dados específica:

- Usam uma biblioteca.
  - Legal! Também funciona em funcional! (*sic*)
- Abrem um livro como o [CLRS] e copiam o código.
  - Infelizmente, programadores de linguagens funcionais como Haskell, e ML não podem se dar a esse luxo.

Apesar desses livros se dizerem ”independentes da linguagem de programação”, eles são independentes apenas no sentido do Henry Ford:

> *Programadores podem usar a linguagem que quiserem, contanto que ela seja imperativa*[1].

Queremos ajudar a resolver este problema!

[1] Henry Ford certa vez disse sobre as cores disponíveis para o Ford T: ”[Os clientes] podem escolher a cor que quiserem, contanto que seja preta”

Objetivos

UFABC

- Mostraremos como implementar e manipular algumas estruturas de dados clássicas no contexto funcional.
- Além disto também apresentaremos algumas estruturas nascidas no contexto funcional para lidar com as suas peculiaridades como imutabilidade, persistência, transparência referencial, avaliação preguiçosa, ...

Estruturas funcionais *vs.* estruturas imperativas

Linguagens funcionais

UFABC

- Apesar de muitos reconhecerem os benefícios das linguagens funcionais, até recentemente elas eram pouco utilizadas e até mesmo pouco conhecidas.
- Isso explica, em parte, porque os códigos existentes atualmente são majoritariamente imperativos.
- Além disto, historicamente, linguagens funcionais tinham um desempenho bem inferior ao de linguagens tradicionais. Isso tem mudado bastante com a evolução de compiladores que fazem análises sofisticadas de código.

- De qualquer modo, mesmo com compiladores espetaculares é improvável que eles sejam capazes de resolver problemas de uso de estruturas de dados com desempenhos ruins.

Então basta usarmos boas estruturas!

Mais fácil falar do que fazer...

UFABC

Mas porque desenvolver estruturas de dados funcionais é mais difícil que estruturas imperativas?

UFABC

- Motivo 1 - Inexistência de atualizações destrutivas

Atualizações destrutivas exigem cuidado em seu uso, porém podem ser extremamente poderosas e eficientes se bem utilizadas.

UFABC

- Motivo 2 - Aumento de expectativa

Em uma estrutura imperativa, raramente esperamos que ela seja persistente (*en: persistent* ↔ *ephemeral*).

Note

- Estruturas de dados persistentes são aquelas que permitem a existência simultânea de diversas versões da estrutura (tipicamente com compartilhamento de dados).
- Estruturas de dados efêmeras são aquelas que permitem a existência de apenas uma versão da estrutura de dados simultaneamente.

Paradigma funcional *vs.* imperativo

UFABC

- Em linguagens de programação funcional, estruturas de dados funcionais são sempre persistentes.
- Em linguagens imperativas, estruturas de dados são tipicamente efêmeras e, quando programadores destas linguagens precisam de estruturas persistentes, eles não se surpreendem e aceitam estruturas cuja facilidade de uso, implementação e complexidade computacional são superiores às suas versões efêmeras.

Expectativas aumentadas

UFABC

Queremos, então:

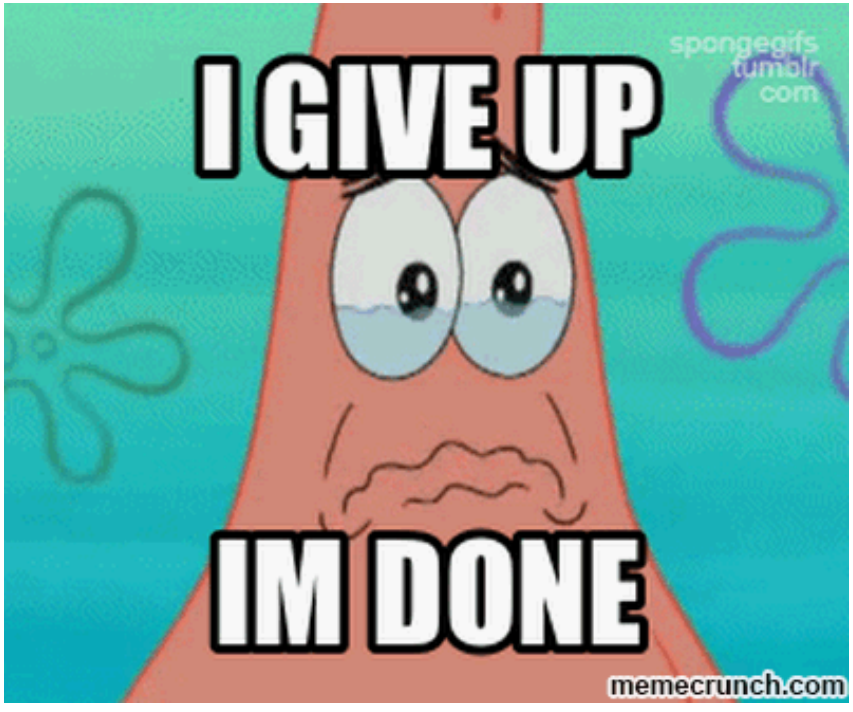
- Persistência.
- Mesma complexidade computacional de EDs efêmeras.

Não nos ajuda o fato de que é possível mostrar que, em alguns casos, linguagens funcionais são intrinsecamente menos eficientes que suas colegas imperativas.

- Era de se esperar, já que o hardware é imperativo.

Jogamos a toalha?

UFABC



Ainda há esperança!

UFABC

- A verdade é que, na prática, a coisa não é tão assustadora assim!
  - Para praticamente todas as estruturas de dados cobertas em um curso de graduação de computação, já se conhecem implementações funcionais cuja complexidade assintótica é a mesma das versões imperativas.
  - Neste curso vamos explorar algumas estruturas de dados funcionais mais comuns. Mas não vamos dar um catálogo, vamos, na verdade, mostrar as técnicas usadas para que você possa desenvolver a sua própria ED quando precisar.

Estruturas de dados persistentes

Uma propriedade que se torna evidente em estruturas de dados (EDs) funcionais é que elas são sempre persistentes

- Atualizações em uma ED não destroem a versão antiga.
- Elementos *afetados* são copiados.
- Elementos não afetados são compartilhados (*en: shared*) entre as diversas versões da ED.

Um primeiro exemplo: Listas

UFABC

Listas são onipresentes em linguages funcionais e são expressas naturalmente em cálculo-λ. As operações que elas dão suporte são normalmente as mesmas daquelas oferecidas por pilhas (*en: stacks*)

- `nova :: Pilha a` Cria uma nova pilha vazia
- `vazia :: Pilha a -> Bool` Verdadeiro se pilha vazia
- `empilha :: a -> Pilha a -> Pilha a` Empilha um valor
- `topo :: Pilha a -> a` Recupera o valor no topo da pilha
- `desempilha :: Pilha a -> Pilha a` Desempilha um valor

Implementação 1 - Na Unha

UFABC

```haskell
data Pilha a = Vazia | Cons a (Pilha a)

nova :: Pilha a
nova = Vazia

vazia :: Pilha a -> Bool
vazia Vazia = True
vazia _     = False

empilha :: a -> Pilha a -> Pilha a
empilha  = Cons

topo :: Pilha a -> a
topo Vazia = error "Pilha vazia"
topo (Cons v _) = v

desempilha :: Pilha a -> Pilha a
desempilha Vazia = error "Pilha vazia"
desempilha (Cons _ p) = p
```

Implementação 2 - Listas padrão do Haskell

UFABC

Utilizando as listas presentes por padrão em Haskell teríamos:

```haskell
type Pilha' a = [a]

nova' :: Pilha' a
nova' = []

vazia' :: Pilha' a -> Bool
vazia' = null

empilha' :: a -> Pilha' a -> Pilha' a
empilha' = (:)

topo' :: Pilha' a -> a
topo' = head

desempilha' :: Pilha' a -> Pilha' a
desempilha' = tail
```

A partir de agora, seja uma lista ou uma pilha, utilizaremos a nomenclatura tradicional de listas.

- Pilhas, de fato, são uma instância do caso mais geral de sequências. Outras instâncias que veremos durante o curso incluem: filas, filas de duas extremidades e listas catenárias.

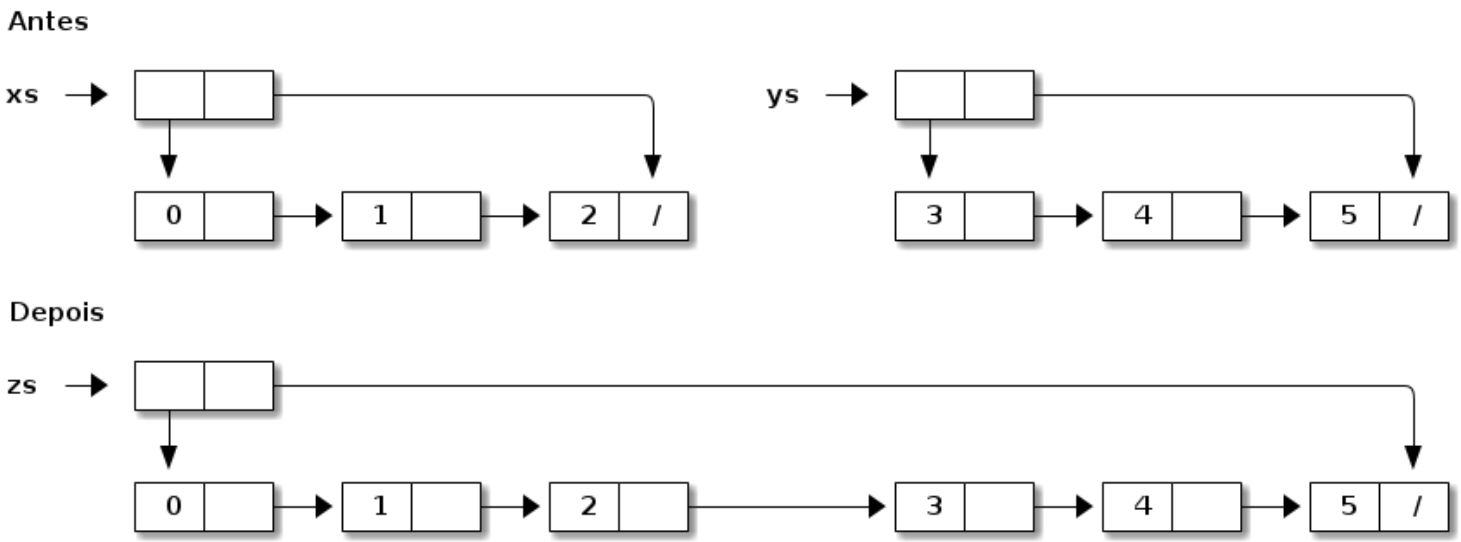
Tip

Momento cultural

Concatenar tem a mesma raiz etimológica de *catenária* e ambas vêm do latim *catena*: cadeia, corrente... Em inglês há inclusive o verbo *catenate* equivalente ao nosso concatenar.

Catenárias...

- Falando em catenária, outra operação comum em listas é a concatenação, normalmente representada pelo operador `(++)`.
- A concatenação de listas pode ser implementada facilmente em linguagens imperativas em tempo constante ($O(1)$).



Concatenação efêmera

UFABC

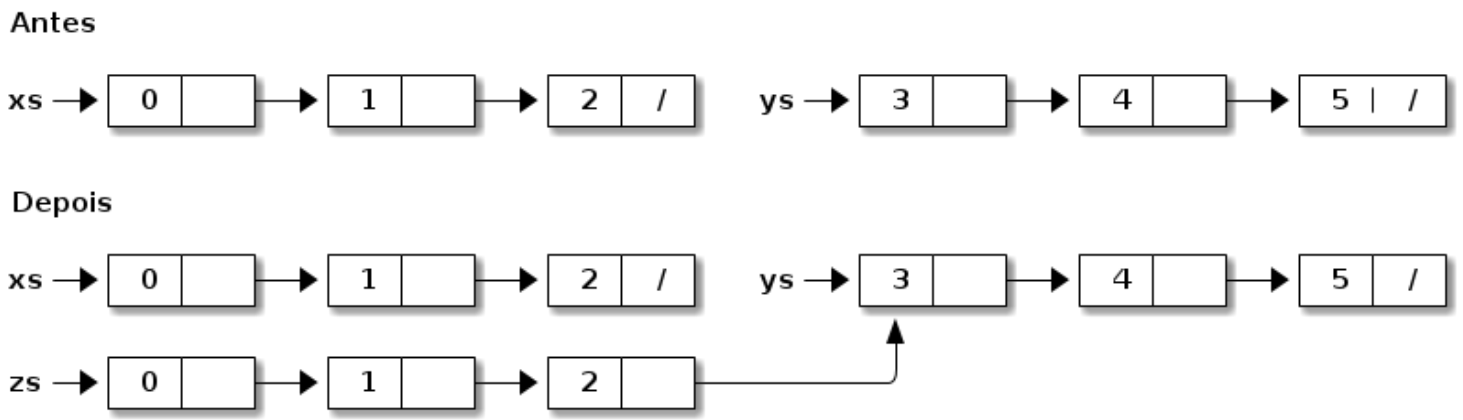
- Porém, essa implementação imperativa é destrutiva em seus parâmetros. Após a execução de `(++)`, nem `xs` nem `ys` podem ser usadas novamente.
  - Não possui persistência.
- Em uma linguagem funcional não há a possibilidade de fazer operações destrutivas, então a saída é fazer uma implementação que copia células, mantendo os valores e trocando o valor do apontador para a próxima célula.

Concatenação persistente

- A implementação da operação (++) fica:

```haskell
(++) :: [a] -> [a] -> [a]
(++) [] ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

A figura abaixo representa essa operação



Concatenação persistente

UFABC

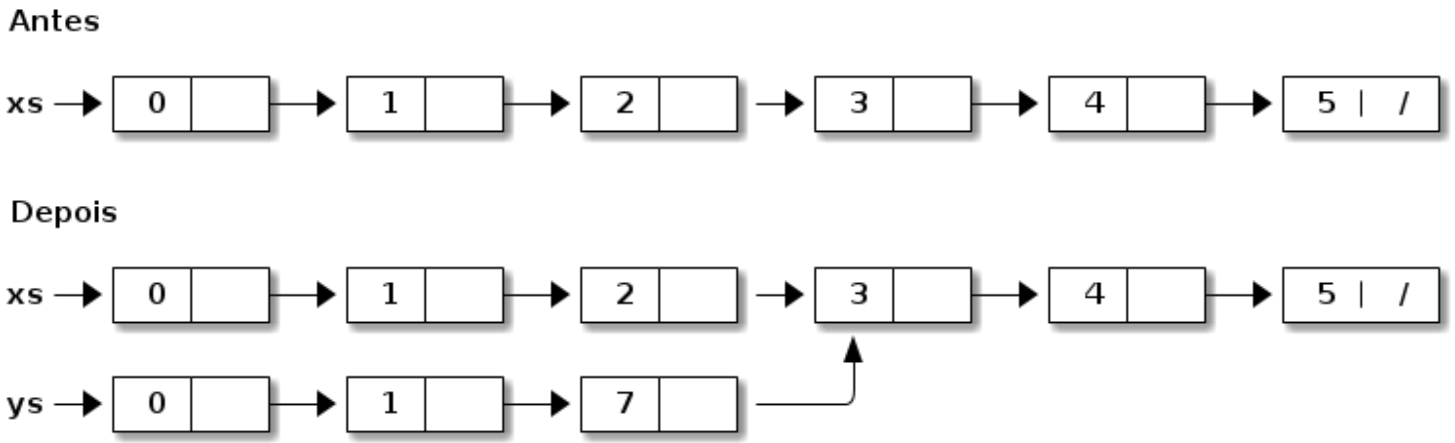
- Ganhamos persistência, mas agora o custo passou a ser $O(n)$ onde n, é o comprimento de `xs`.
  - Veremos mais adiante como reduzir esse custo para $O(1)$.
- Um caso parecido ocorre quando queremos fazer a atualização de um elemento com índice i da lista. Neste caso, precisamos copiar as células da lista ligada até o elemento desejado.

UFABC

O código fica:

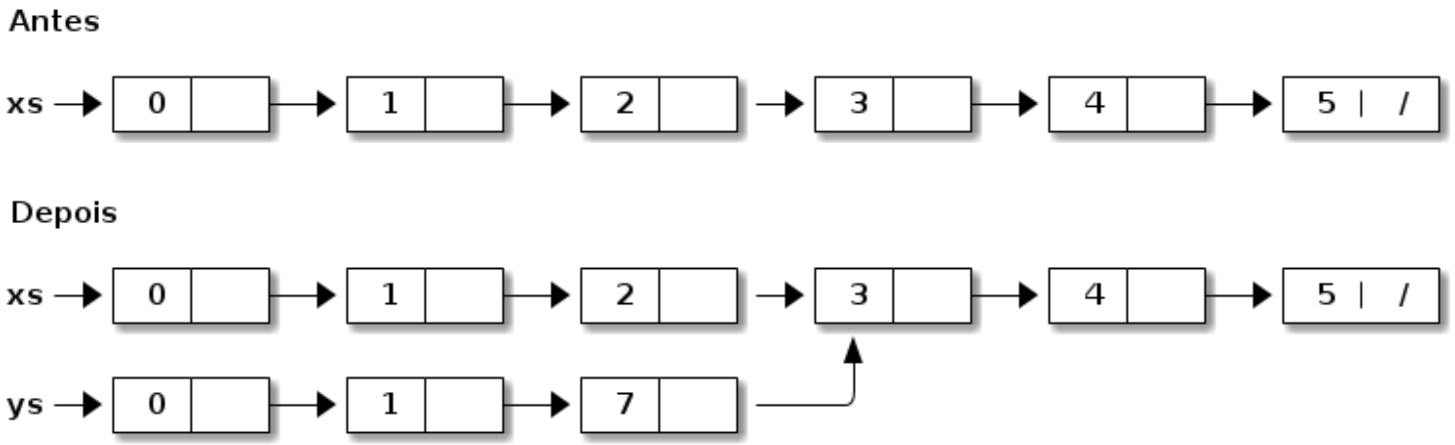
```haskell
update :: [a] -> Int -> a -> [a]
update [] _ _ = error "Tentando atualizar lista vazia"
update (_:xs) 0 v = v:xs
update (x:xs) i v = x : update xs (i - 1) v
```

Graficamente `ys = update xs 2 7`:



UFABC

Apenas para reforçar: copiamos apenas o caminho até o elemento desejado.



Warning

Essa implementação se baseia fortemente no coletor de lixo (*en: garbage collector*). Implementar algo dessa maneira com controle de memória manual ficaria rapidamente complicadíssimo por conta dos compartilhamentos.

UFABC

1. Escreva uma função `sufixos :: [a] -> [[a]]` que recebe uma lista `xs` e devolve uma lista com todos os sufixos da lista `xs` em ordem decrescente de comprimento. Por exemplo:

```
1  sufixos [1, 2, 3, 4] =
2      [[1,2,3,4],[2,3,4],[3,4],[4],[]]
```

2. Mostre que o resultado pode ser gerado em tempo $O(n)$ e representado em espaço $O(n)$.

Percorrendo listas

UFABC

- Percorrer listas do início em direção ao fim é fácil e eficiente.
- Percorrer listas no sentido contrário é um pouco mais complicado.
- Uma lista em um contexto funcional é muito parecida com uma lista ligada com encadeamento simples utilizada corriqueiramente em linguagens imperativas.

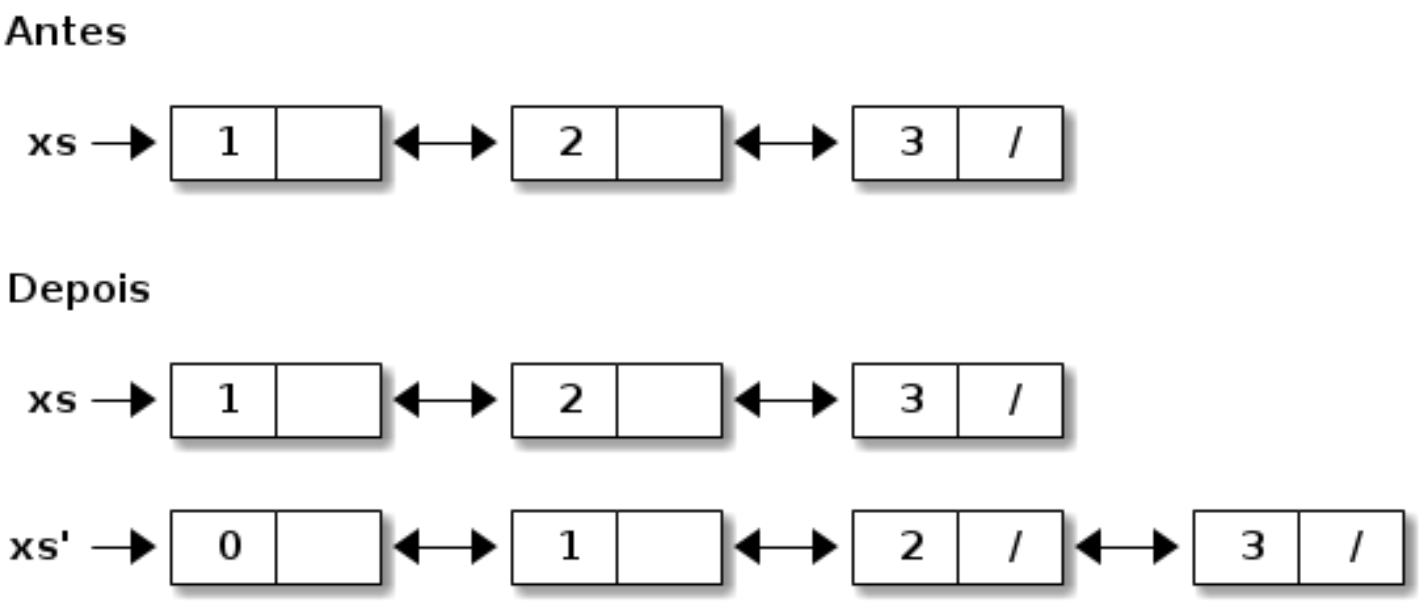
Como fazer uma lista duplamente encadeada caso queiramos percorrê-la em ambas as direções?

```haskell
--                                  esquerda  valor direita
data Lista2 a = Vazia2 | Cons2 (Lista2 a) a (Lista2 a)

null2 :: Lista2 a -> Bool
null2 Vazia2 = True
null2 _      = False

head2 :: Lista2 a -> a
head2 Vazia2 = error "Lista vazia"
head2 (Cons2 _ x _) = x

tail2 :: Lista2 a -> Lista2 a
tail2 Vazia2 = Vazia2
tail2 (Cons2 _ _ d) = d
```

Parece bom! Mas e agora, como implementar a função **cons2**?

```haskell
cons2 :: a -> Lista2 a -> Lista2 a
cons2 = undefined -- ??
```

- O problema é que, agora, o nó seguinte da nossa lista aponta para o anterior. Modificar o nó inicial da lista causa uma alteração em cascata até o final da lista!



Listas duplamente encadeadas - Tentativa de implementação

UFABC

- Toda a lista precisa ser copiada! Não há compartilhamento!
- Colocar algo no fim da lista ou concatenar têm o mesmo problema.
- Temos o que queríamos, mas a que custo?

UFABC

- Zipper é um *idiom* (do inglês: expressão idiomática) de linguagens funcionais muito utilizado para manipular estruturas de dados persistentes.
- Zippers podem ser usados não apenas para percorrer estruturas de dados como também para auxiliar em sua modificação.

Note

Aqui veremos o uso aplicado de Zippers, mas existem maneiras formais de se definir um zipper para uma estrutura de dados qualquer. Na verdade um zipper é definido como a derivada da ADT[2].

[2] http://pesquisa.ufabc.edu.br/haskell/posts/categorias/06-ADT.html

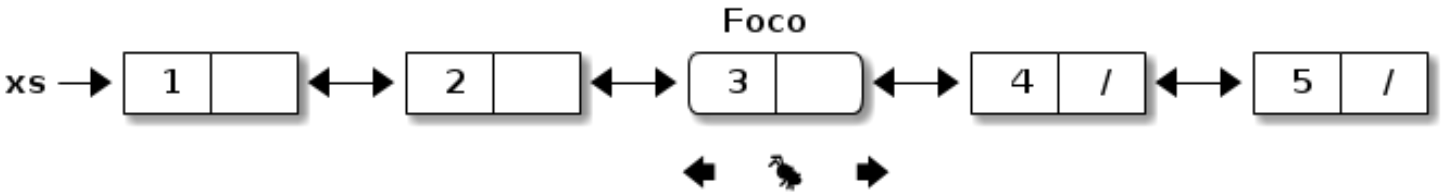
- Mas afinal, o que é um zipper?

Zippers funcionam de uma maneira parecida com aquela pela qual uma formiga explora o mundo.

- A localização atual da formiga é o foco do zipper.
- A formiga pode se deslocar para explorar o mundo seguindo os caminhos disponíveis.
- Para poder se orientar, ela deixa por cada local por onde passou uma trilha de feromônio que pode ser utilizada para retornar pelo mesmo caminho por onde ela veio.

UFABC

- Esta última característica permite que implementemos, por exemplo, backtracking em linguagens funcionais de maneira eficiente.
- No caso específico de uma lista, a formiga/zipper, que mora em uma lista, explora um mundo unidimensional: ela só pode avançar ou retroceder.



- Como implementar tal abstração?

UFABC

- Simples! 2 listas!
  - Uma contém o que ainda não foi percorrido da lista. Tudo à direita do foco.
  - A outra contém uma trilha de migalhas, a trilha de feromônio da nossa formiga, para que ela possa voltar por onde veio. Tudo à esquerda do foco.

```haskell
type ListZipper a =  ([a], [a])
```

List Zippers

UFABC

```haskell
toListZipper :: [a] -> ListZipper a
toListZipper xs = ([], xs)

fromListZipper :: ListZipper a -> [a]
fromListZipper (es, ds) = reverse es ++ ds

lzFoco :: ListZipper a -> Maybe a
lzFoco (_, []) = Nothing
lzFoco (_, x:_) = Just x

lzDir :: ListZipper a -> Maybe (ListZipper a)
lzDir (_, []) = Nothing
lzDir (es, d:ds) = Just (d:es, ds)

lzEsq :: ListZipper a -> Maybe (ListZipper a)
lzEsq ([], _) = Nothing
lzEsq (e:es, ds) = Just (es, e:ds)
```

UFABC

- Tudo muito bem e muito bonito, mas além de percorrer a lista como uso isso para modificá-la?
- Imagine que temos uma lista de inteiros e queremos trocar todos os números 5 por 42.
  - Vamos definir, primeiramente, uma função para trocar o valor do elemento em foco.

```haskell
lzTrocaFoco :: a -> ListZipper a -> Maybe (ListZipper a)
lzTrocaFoco _ (_, []) = Nothing
lzTrocaFoco y (es, _:ds) = Just (es, y:ds)
```

Agora podemos definir a função que troca os cincos:

```haskell
trocaCincos :: [Int] -> [Int]
trocaCincos [] = []
trocaCincos xs =
  fromListZipper $ trocaZip $ toListZipper xs
  where
    trocaZip z@(_, []) = z
    trocaZip z@(_, d:_) =
      let nz = if d == 5
          then fromJust $ lzTrocaFoco 42 z
          else z in
      maybe nz trocaZip (lzDir nz)
```

```haskell
-- Mas pra que tudo isso?
trocaCincos' :: [Int] -> [Int]
trocaCincos' xs = [if x == 5 then 42 else x | x <- xs]
```

Mas pra quê tudo isso?

UFABC

- Listas são estruturas de dados relativamente simples. Por isso a manipulação de um elemento é fácil...
  - ... e existem mecanismos como compreensão de listas que nos ajudam a fazer o que queremos.

Mas pra quê tudo isso 2?

UFABC

- Pense agora em estruturas um pouco mais elaboradas, como árvores ou grafos.
  - O fato de estarmos lidando com uma linguagem funcional (e EDs persistentes) impede que simplesmente alteremos um valor de um nó na árvore através de um ponteiro!
  - Precisamos substituir o nó que queremos modificar por um novo e "consertar" todo o caminho por onde passamos para que ele inclua as novas versões dos dados.

O caminho por onde passamos é justamente o caminho armazenado no Zipper!!

Exercício 2

UFABC

Estude a função `trocaCincos` e as funções que ela utiliza. Em seguida reimplemente a função `fromListZipper :: ListZipper a -> [a]`. Você não pode utilizar as funções `(++)` ou `reverse` (nem reimplementá-las!).

```haskell
trocaCincos :: [Int] -> [Int]
trocaCincos [] = []
trocaCincos xs =
  fromListZipper $ trocaZip $ toListZipper xs
  where
    trocaZip z@(_, []) = z
    trocaZip z@(_, d:_) =
      let nz = if d == 5
          then fromJust $ lzTrocaFoco 42 z
          else z in
      maybe nz trocaZip (lzDir nz)
```

Árvores

Árvores binárias

UFABC

- Padrões de compartilhamento mais interessantes podem aparecer quando temos mais de um campo por nó.
- Árvores binárias apresentam esses padrões.

UFABC

Árvores binárias de busca são árvores binárias cujos elementos internos são organizados em ordem simétrica.

- Ou seja, o valor contido por um nó qualquer é maior que todos os valores armazenados em sua sub-árvore esquerda e menor que todos os elementos armazenados em sua sub-árvore direita.
- Nesta definição assumimos que a árvore só conterá elementos que possuem uma ordem total. Ou, em outras palavras, não admitimos elementos repetidos.

UFABC

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

- Vamos usar essa representação para implementar *Conjuntos*.
- Contudo, é fácil de adaptá-la para outras abstrações como *Mapas* ou para outras operações (como encontrar o i-ésimo menor elemento) acrescentando alguns valores na definição da nossa árvore.

Uma classe de tipo para conjuntos

UFABC

O código abaixo define uma *typeclass* para um conjunto:

```haskell
class Conjunto t where
   membro :: Ord a => a -> t a -> Bool
   insere :: Ord a => a -> t a -> t a
```

Note

Uma *typeclass* (classe de tipos) é semelhante a uma interface que define algum comportamento. A nossa typeclass `Conjunto` define dois métodos: `membro` e `insere`. Qualquer tipo `x` que deseje "*participar*"desta typeclass precisa fornecer implementações de funções para estes dois métodos. Quando isto é feito dizemos que declaramos uma instância da typeclass para o tipo `x`.

UFABC

⚠ Warning

Não caia na confusão de achar que os termos comuns em orientação a objeto *classes*, *instâncias* e *métodos* têm qualquer coisa a ver com os termos homônimos em Haskell! Eles não tem relação!

UFABC

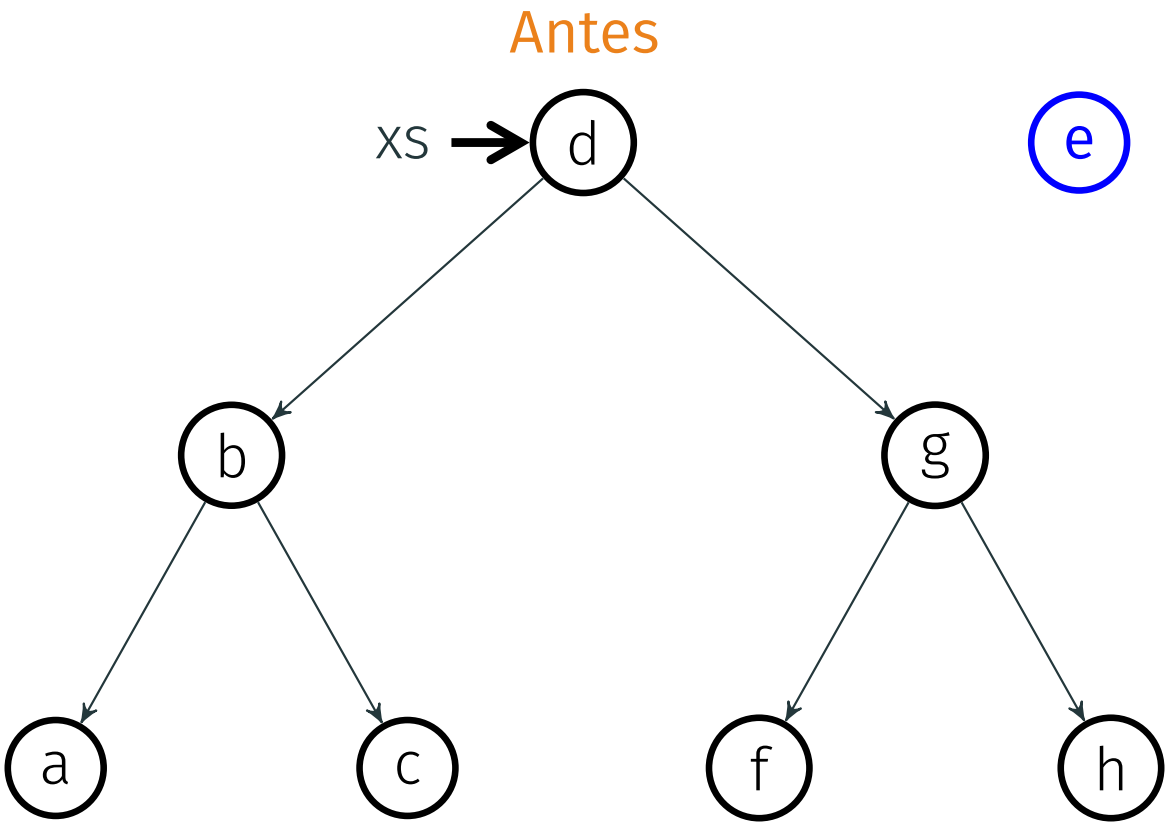
```haskell
data Arv a = ArvVazia | No (Arv a) a (Arv a)
```

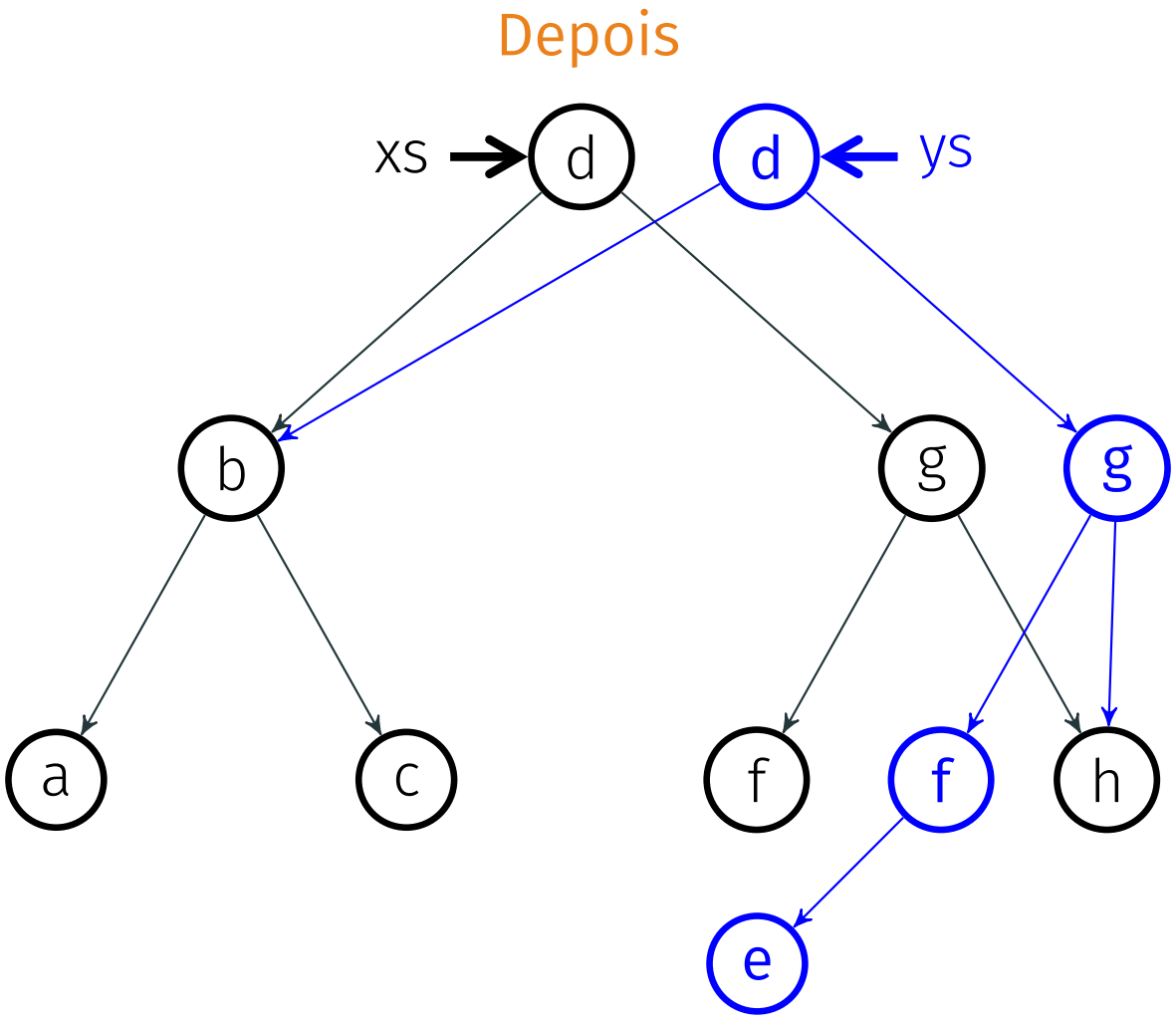
E para tornar nossa árvore um conjunto podemos fazer:

```haskell
instance Conjunto Arv where

    membro _ ArvVazia = False
    membro x (No e v d)
        | x < v     = membro x e
        | x > v     = membro x d
        | otherwise = True

    insere x ArvVazia = No ArvVazia x ArvVazia
    insere x t@(No e v d)
       | x < v     = No (insere x e) v d
       | x > v     = No e v (insere x d)
       | otherwise = t
```

UFABC

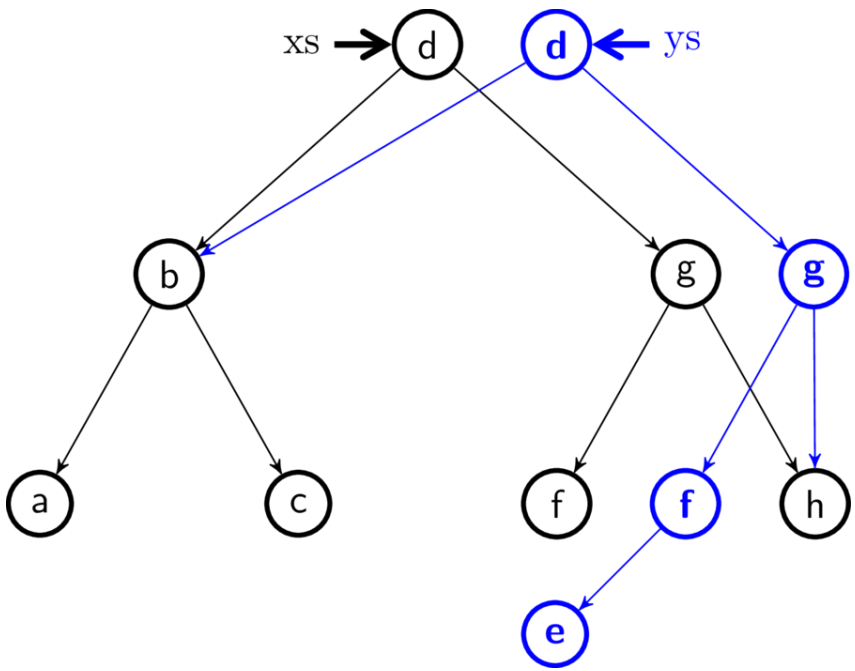
A execução da função **`membro`** é trivial. Vamos verificar com mais cuidado a execução de **`insere`** com o elemento **`e`**:




Depois
xs
d
d
ys
b
g
g
a
c
f
f
h
e

UFABC

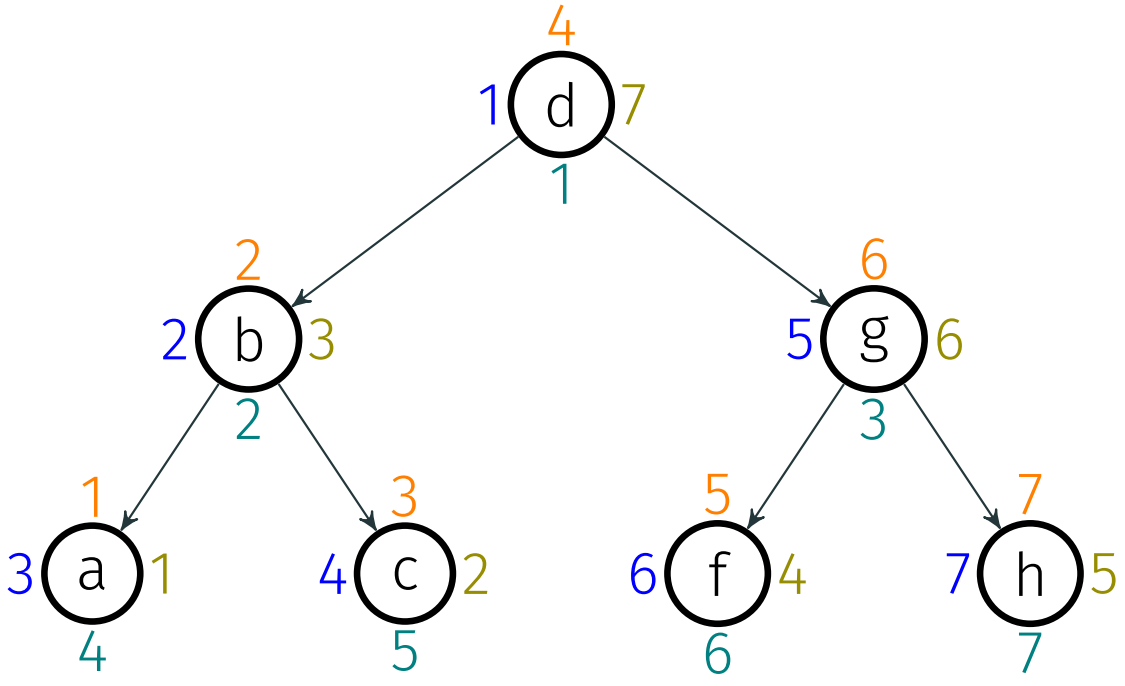
- Cada nó que foi copiado, ou seja aqueles que foram afetados pela inserção, tem ao menos uma sub-árvore compartilhada com a árvore inicial. Na verdade, são copiados apenas os nós no caminho da inserção que, para uma árvore que for balanceada, é proporcional a $O(\log n)$.
- Em um sistema real a maior parte dos nós reside nas sub-árvores compartilhadas.



Percorrendo árvores

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

Ou em código que imprime o conteúdo dos nós:

```haskell
preOrdem, inOrdem, posOrdem :: Show a => Arv a -> IO ()
preOrdem ArvVazia = return ()
preOrdem (No e x d) = do
  print x
  preOrdem e
  preOrdem d
inOrdem ArvVazia = return ()
inOrdem (No e x d) = do
  inOrdem e
  print x
  inOrdem d
posOrdem ArvVazia = return ()
posOrdem (No e x d) = do
  posOrdem e
  posOrdem d
  print x
```

Percorrendo em largura

UFABC

- Mas e se quisermos uma busca em largura? As implementações tradicionais para percorrer uma árvore lançam mão de uma fila.
- Vamos usar a implementação de fila fuleira® baseada em listas.

```haskell
-- Fila "fuleira"®
type Queue a = [a]

-- O(1)
empty :: Queue a
empty = []

-- O(1)
isEmpty :: Queue a -> Bool
isEmpty = null
```

Operações na fila fuleira®

UFABC

```haskell
-- O(1)
enq :: Queue a -> a -> Queue a
enq = flip (:)

-- O(n)
deq :: Queue a -> (a, Queue a)
deq xs = (last xs, init xs)
```

BFS - Versão 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
```

- Com as estruturas de dados que temos até agora (apenas listas), apesar de possível, ainda não é trivial implementar uma estrutura de dados para fila que seja eficiente.

Tip

Já é possível, contudo, implementar uma ED com complexidade amortizada $O(1)$ (falaremos mais sobre isso mais adiante no curso).

Quem não tem cão, caça com gato!

UFABC

- Uma outra abordagem (sem usar filas) é utilizar um algoritmo baseado em níveis:

```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

- Vamos considerar agora um problema um pouco diferente: numeração dos nós em largura (en: *breadth-first numbering*, BFN).
- Considere que queremos uma função:

```haskell
bfn :: Arv a -> Arv (a, Int)
```

onde o segundo elemento da tupla é a ordem em que este elemento foi visitado pela varredura.

- Se fosse para fazer os casos tradicionais (*a.k.a.* fáceis) pré/in/pós-ordem...

Numerando os nós

UFABC

```haskell
numeraPreOrdem :: Arv a -> Arv (a, Int)
numeraPreOrdem arv =
  snd $ numeraPreOrdem' 1 arv
  where
    numeraPreOrdem' i ArvVazia = (i, ArvVazia)
    numeraPreOrdem' i (No e x d) =
      (i3, No e' (x, i) d')
      where
        (i2, e') = numeraPreOrdem' (i + 1) e
        (i3, d') = numeraPreOrdem' i2 d
```

Versão consulta:

```haskell
preOrdem ArvVazia = return ()
preOrdem (No e x d) = do
  print x
  preOrdem e
  preOrdem d
```

Numerando os nós

UFABC

```haskell
numeraInOrdem :: Arv a -> Arv (a, Int)
numeraInOrdem arv =
   snd $ numeraInOrdem' 1 arv
  where
    numeraInOrdem' i ArvVazia = (i, ArvVazia)
    numeraInOrdem' i (No e x d) =
      (i3, No e' (x, i2) d')
      where
        (i2, e') = numeraInOrdem' i e
        (i3, d') = numeraInOrdem' (i2 + 1) d
```

Versão consulta:

```haskell
inOrdem ArvVazia = return ()
inOrdem (No e x d) = do
  inOrdem e
  print x
  inOrdem d
```

Numerando os nós

UFABC

```haskell
numeraPosOrdem :: Arv a -> Arv (a, Int)
numeraPosOrdem arv =
   snd $ numeraPosOrdem' 1 arv
  where
    numeraPosOrdem' i ArvVazia = (i, ArvVazia)
    numeraPosOrdem' i (No e x d) =
      (i3 + 1, No e' (x, i3)  d')
      where
        (i2, e') = numeraPosOrdem' i e
        (i3, d') = numeraPosOrdem' i2 d
```

Versão consulta:

```haskell
posOrdem ArvVazia = return ()
posOrdem (No e x d) = do
  posOrdem e
  posOrdem d
  print x
```

UFABC

- Já a versão em largura é um pouco mais trabalhosa...

```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')
```

UFABC

- Apesar de, a princípio, esta versão parecer complicada ela é bem eficiente.
- Se a fila utilizada tiver complexidade $O(1)$ para inserção e remoção, então o algoritmo roda em tempo $O(n)$.

BFN - Versão pavorosa (*a.k.a.* por níveis)

UFABC

```haskell
bfn2 arv = head $ bfn' 1 [arv]
  where
    filhos ArvVazia = []
    filhos (No a _ b) = [a, b]

    recons _ [] [] = []
    recons i (ArvVazia:ts) cs= ArvVazia : recons i ts cs
    recons i ~(No _ x _ : ts) ~(a : b : cs) =
      No a (x, i) b : recons (i + 1) ts cs

    bfn' _ [] = []
    bfn' i lvl =
      let
        proxNivel = concatMap filhos lvl
        j = i + length proxNivel `div` 2
        proxNivelNum = bfn' j proxNivel in
        recons i lvl proxNivelNum
```

BFN - Versão pavorosa (*a.k.a.* por níveis)

UFABC

- Além de pavorosa, essa versão não é nem um pouco eficiente. Ela varre cada nível 3 vezes!
  - Uma vez para pegar os filhos.
  - Uma segunda vez para computar o comprimento.
  - Uma terceira para reconstruir o nível.
- Veremos mais adiante como fazer isto de maneira mais eficiente e simples explorando a avaliação preguiçosa disponível em praticamente todas as linguagens de programação funcional.

Um zipper para árvores

UFABC

Figura 1: Fonte: Street Art Utopia

Um zipper para árvores

UFABC

- Diferentemente do caso de listas, zippers para árvores não são unidimensionais.
- Para percorrer uma árvore podemos:
  - ir em direção às folhas (no caso de uma árvore binária seguindo pelos filhos à esquerda ou à direita).
  - ir em direção à raiz (para cima).

Assim, nosso zipper terá as seguintes opções de deslocamento:

- `esq` $\rightarrow$ caminha em direção às folhas pelo filho esquerdo
- `dir` $\rightarrow$ caminha em direção às folhas pelo filho direito
- `cima` $\rightarrow$ caminha em direção à raiz

Implementando um zipper para árvores

UFABC

Assim como o zipper de listas, o nosso zipper de árvores precisará guardar pelo menos o foco e o caminho percorrido.

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

```haskell
type ZipperArv a =
    (Arv a, [Either (a, Arv a) (a, Arv a)])
```

Implementando um zipper para árvores

UFABC

```haskell
toArvZipper :: Arv a -> ZipperArv a
toArvZipper arv = (arv, [])

fromArvZipper :: ZipperArv a -> Arv a
fromArvZipper (arv, []) = arv
fromArvZipper z = fromJust $ fromArvZipper <$> arvCima z

arvFoco :: ZipperArv a -> Maybe a
arvFoco (ArvVazia, _) = Nothing
arvFoco (No _ x _, _) = Just x

arvTrocaFoco :: a -> ZipperArv a -> Maybe (ZipperArv a)
arvTrocaFoco _ (ArvVazia, _) = Nothing
arvTrocaFoco x (No a _ b, rastro) =
    Just (No a x b, rastro)
```

Movimentando o zipper

UFABC

```haskell
arvDir :: ZipperArv a -> Maybe (ZipperArv a)
arvDir (ArvVazia, _) = Nothing
arvDir (No e x d, rastro) =
    Just (d, Right (x, e):rastro)

arvEsq :: ZipperArv a -> Maybe (ZipperArv a)
arvEsq (ArvVazia, _) = Nothing
arvEsq (No e x d, rastro) =
    Just (e, Left (x, d):rastro)

arvCima :: ZipperArv a -> Maybe (ZipperArv a)
arvCima (_, []) = Nothing
arvCima (arv, Left (x, d):rastro)  =
    Just (No arv x d, rastro)
arvCima (arv, Right (x, e):rastro) =
    Just (No e x arv, rastro)
```

Sobre o uso de `Maybe`

UFABC

- Note que como no caso do zipper para listas, todas as operações que podem falhar devolvem um `Maybe a`.
- Isto torna as nossas funções totais evitando, em tempo de compilação, uma série de erros que normalmente ocorreriam apenas em tempo de execução.

Note

- Uma função é total se ela for definida para todos os valores possíveis do seu tipo de entrada.
- Uma função é parcial se existem algumas entradas para as quais ela não tem valor de saída definido.

```
1  > head []
2  *** Exception: Prelude.head: empty list
```

UFABC

Tal qual fizemos com as listas, vamos trocar um valor por outro em nossa BST na unha e em seguida usando zippers!

```haskell
trocaArv :: Ord a => a -> a -> Arv a -> Arv a
trocaArv _ _ ArvVazia = ArvVazia
trocaArv velho novo (No e x d)
  | x < velho = No e x $ trocaArv velho novo d
  | x > velho = No (trocaArv velho novo e) x d
  | otherwise = No e novo d
```

Trocando valores - Versão com zipper

UFABC

```haskell
trocaArvZ :: Ord a => a -> a -> Arv a -> Arv a
trocaArvZ velho novo arv =
  fromArvZipper $ trocaArvZ' $ toArvZipper arv
  where
    trocaArvZ' z@(ArvVazia, _) = z
    trocaArvZ' z
      | f < velho = go arvDir
      | f > velho = go arvEsq
      | otherwise = fromJust $ arvTrocaFoco novo z
      where
        f = fromJust $ arvFoco z
        go d = maybe z trocaArvZ' (d z)
```

Comentários finais

UFABC

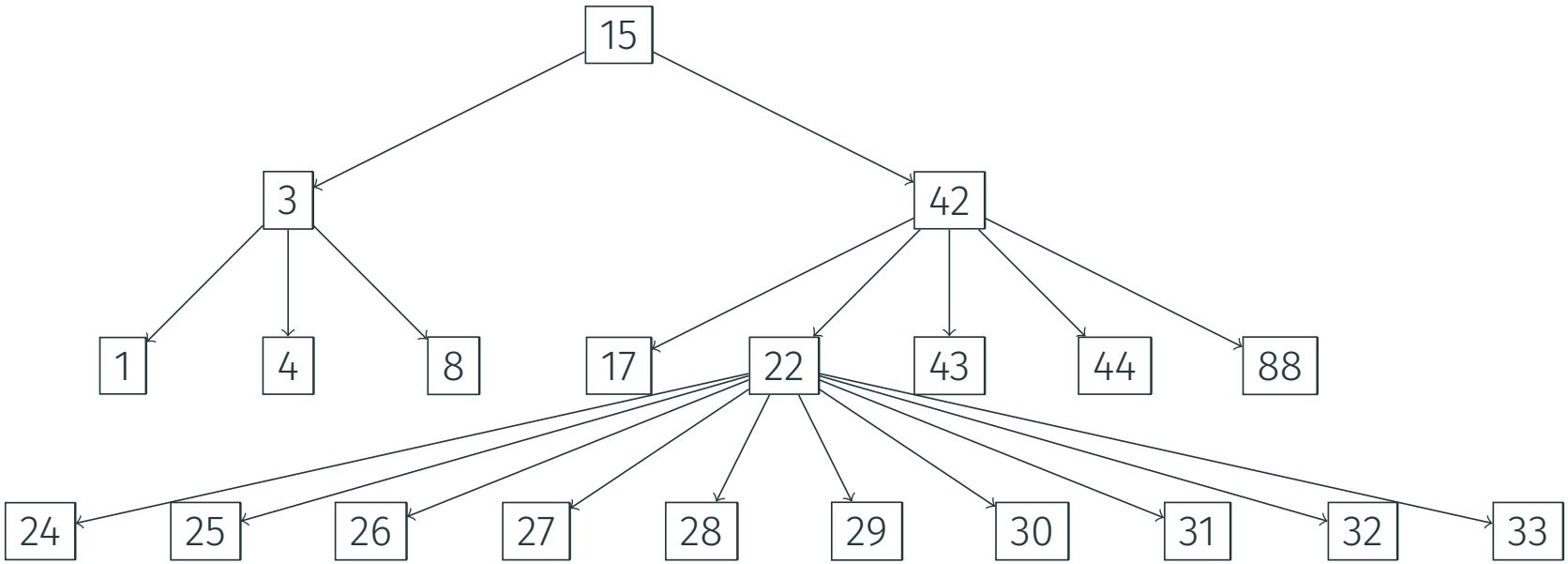
- Zippers foram propostos originalmente por Gérard Huet em 1997.
- Certamente muitos que trabalharam com árvores durante os mais de 50 anos de linguagens funcionais fizeram buscas em largura.
  - Impressionantemente, um dos primeiros papers que descrevem de maneira organizada e elegante como isto pode ser feito é de 1993!
    - Veremos esta solução na próxima aula.
  - Em 2000 Chris Okasaki fez uma enquete com diversos usuários (em uma conferência da área) e descobriu que havia um bloqueio mental generalizado. Em outras palavras, ninguém sabia fazer isso direito em um contexto funcional!

Roseiras

UFABC

- Roseiras (en: *rose trees*) são árvores que têm um número variável, potencialmente infinito[3], de ramificações por nó.

[3] Veremos mais adiante como a avaliação preguiçosa nos permite trabalhar com EDs infinitas.


15
3
42
1
4
8
17
22
43
44
88
24
25
26
27
28
29
30
31
32
33

Implementando uma roseira (plantando? 😛)

UFABC

- Em Haskell podemos representar tal estrutura da seguinte maneira

```haskell
data RoseTree a = EmptyRose | RoseTree a [RoseTree a]
```

- Contudo, a implementação acima permite que criemos coisas assim:

```haskell
arv = RoseTree 42 [EmptyRose]
```

- Não queremos isso! Temos algumas opções para evitar o problema...

Implementando uma roseira

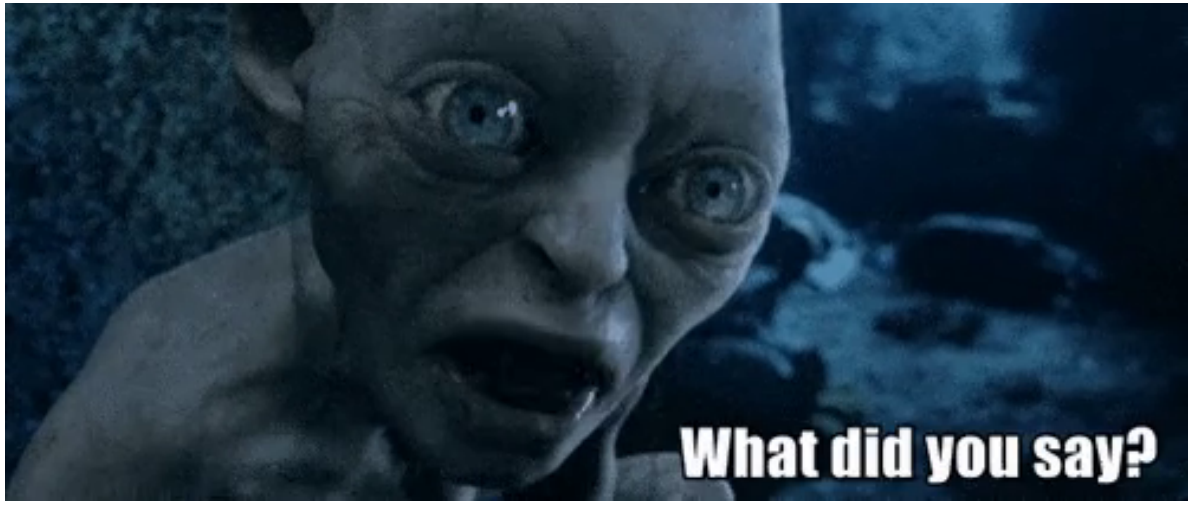
UFABC

```haskell
data RoseTree a = EmptyRose | RoseTree a [RoseTree a]
```

1. Algoritmo do avestruz[4]: assumimos que o problema não vai ocorrer pois a nossa implementação, assim como os usuários da nossa ED que são "gente boa", nunca vão fazer a besteira de colocar uma árvore vazia na lista.
2. Função intermediária[5]: para acesso ao construtor `RoseTree` que verifica, em tempo de execução, que não estamos recebendo árvores vazias para incluir na lista.
3. Tipos fantasmas[6]: para garantir que tudo está correto em tempo de compilação!

[4] https://pt.wikipedia.org/wiki/Algoritmo_do_avestruz
[5] https://wiki.haskell.org/Smart_constructors
[6] https://wiki.haskell.org/Phantom_type

- Um tipo fantasma (en: *phantom type*) é simplesmente um tipo paramétrico que não utiliza pelo menos um dos seus tipos parâmetros em sua definição.



Talvez um exemplo seja ser mais esclarecedor...

UFABC

```haskell
data FormData a = FormData String
```

O exemplo parece estranho... Porque um tipo `a` se ele não é usado? De fato é possível usar o tipo `FormData` da seguinte maneira:

```haskell
changeType :: FormData a -> FormData b
changeType (FormData str) = FormData str
```

Que troca o tipo simplesmente chamando o mesmo construtor dos dois lados da equação!

Usando tipos fantasmas

UFABC

```haskell
data Validated
data Unvalidated
-- Se escondermos os construtores dos usuários (basta
-- não exportá-los) essa  função é a única que pode
-- ser usada pelos usuários. Ou seja, o usuário só
-- pode criar FormData que não são validados.
formData :: String -> FormData Unvalidated
formData str = FormData str

-- Podemos criar uma função que valida os dados
validate :: FormData Unvalidated -> Maybe (FormData
↪  Validated)
validate (FormData str) = ...

-- E assim GARANTIR que funções consumam apenas dados
-- que foram validados!
useData :: FormData Validated -> IO ()
useData (FormData str) = ...
```

Dá pra melhorar!

UFABC

```haskell
data FormData a = FormData String
```

- Contudo, essa implementação não garante que só existam os tipos que façam sentido como `FormData Validated` e `FormData Unvalidated`.
- Tal como está, o código permitiria a criação de um `FormData Int`.
  - Claro, como só exportamos a função auxiliar `formData`, apenas a nossa própria biblioteca conseguiria tal feito. Mas nem isso quero deixar em aberto!

Dá pra melhorar!

UFABC

Para resolver, podemos fazer[7]:

```haskell
data ValidationStatus = Validated | Unvalidated

data FormData (status :: ValidationStatus) where
  FormData :: String -> FormData status

formData :: String -> FormData Unvalidated
formData str = FormData str

validate :: FormData Unvalidated -> Maybe (FormData
↪  Validated)
validate (FormData str) = ..
```

[7] Esse código exige o uso das extensões `DataKinds`, `KindSignatures`, `GADTs`

E a roseira?

UFABC

Ah sim! Estávamos falando de roseiras!

Utilizando tipos fantasmas, o código pode ser melhorado:

```haskell
data Emptiness = Empty | NonEmpty

data RoseTree (e :: Emptiness) a where
  EmptyRose :: RoseTree Empty a
  RoseTree  :: a -> [RoseTree NonEmpty a] -> RoseTree
  ↪  NonEmpty a
```

Tip

Aqui vamos apenas arranhar a superfície do que é possível fazer com *type-safe programming* e *dependent types*. Apesar de haver várias propostas em andamento para avançar a linguagem Haskell nesta direção, ela é atualmente incompleta. Caso queira explorar além do que falaremos por aqui, procure pelas linguagens: Idris, Agda e Coq.

Um zipper para roseiras

UFABC

Figura 2: craft-craft.net

- Não faz mais sentido falar em direita ou esquerda para ir em direção às folhas.
- Mais ainda, o que significa mudar o foco de uma roseira?
- Temos uma mistura de um zipper de árvores com um zipper de listas!

UFABC

```haskell
data RoseTreeZipper a = RoseTreeZipper
  a -- Valor do foco
  (ListZipper (RoseTree NonEmpty a)) -- Subzipper da
  ↪ lista de filhos
  [(a, ListZipper (RoseTree NonEmpty a))] -- Rastro
```

Vamos comparar[8] com o zipper de árvores e listas:

```haskell
type ListZipper a =  ([a], [a])
```

```haskell
type ZipperArv a =
    (Arv a, [Either (a, Arv a) (a, Arv a)])
```

[8] Essa implementação não é minimal, mas facilita o entendimento.

Implementando um zipper para roseiras

UFABC

```haskell
toRoseZipper :: RoseTree e a -> Maybe (RoseTreeZipper a)
toRoseZipper EmptyRose = Nothing
toRoseZipper (RoseTree v bs) =
  Just $ RoseTreeZipper v (toListZipper bs) []
```

```haskell
roseFoco :: RoseTreeZipper a -> a
roseFoco (RoseTreeZipper v _ _) = v
```

```haskell
roseTrocaFoco :: RoseTreeZipper a-> a-> RoseTreeZipper a
roseTrocaFoco (RoseTreeZipper _ lz ps) x =
  RoseTreeZipper x lz ps
```

Implementando um zipper para roseiras

UFABC

```haskell
roseDir :: RoseTreeZipper a -> Maybe (RoseTreeZipper a)
roseDir (RoseTreeZipper v lz ps) = do
  lz' <- lzDir lz
  Just $ RoseTreeZipper v lz' ps

roseEsq :: RoseTreeZipper a -> Maybe (RoseTreeZipper a)
roseEsq (RoseTreeZipper v lz ps) = do
  lz' <- lzEsq lz
  Just $ RoseTreeZipper v lz' ps
```

Implementando um zipper para roseiras

UFABC

```haskell
roseBaixo:: RoseTreeZipper a -> Maybe (RoseTreeZipper a)
roseBaixo (RoseTreeZipper v lz ps) = do
  (RoseTree v' bs') <- lzFoco lz
  Just $ RoseTreeZipper v' (toListZipper bs') ((v, lz) :
  ↪  ps)
```

```haskell
roseCima :: RoseTreeZipper a -> Maybe (RoseTreeZipper a)
roseCima (RoseTreeZipper _ _ []) = Nothing
roseCima (RoseTreeZipper _ _ ((v',lz'):ps)) =
  Just $ RoseTreeZipper v' lz' ps
```

Comentários finais

UFABC

- Roseiras foram apresentadas por Lambert Meertens em 1988.
- Sua variação preguiçosa para buscas vinculadas a otimização talvez seja a sua aplicação mais comum quando vinculada a zippers.

Exercício 3

UFABC

A combinação das implementações das funções `trocaFoco` e `roseCima`, tal como está, descarta eventuais alterações feitas no valor em foco. Corrija a implementação.

```haskell
data RoseTreeZipper a = RoseTreeZipper
  a -- Valor do foco
  (ListZipper (RoseTree NonEmpty a)) -- Subzipper da
  ↪  lista de filhos
  [(a, ListZipper (RoseTree NonEmpty a))] -- Rastro
```

```haskell
roseTrocaFoco :: RoseTreeZipper a-> a-> RoseTreeZipper a
roseTrocaFoco (RoseTreeZipper _ lz ps) x =
  RoseTreeZipper x lz ps
```

```haskell
roseCima :: RoseTreeZipper a -> Maybe (RoseTreeZipper a)
roseCima (RoseTreeZipper _ _ []) = Nothing
roseCima (RoseTreeZipper _ _ ((v',lz'):ps)) =
  Just $ RoseTreeZipper v' lz' ps
```

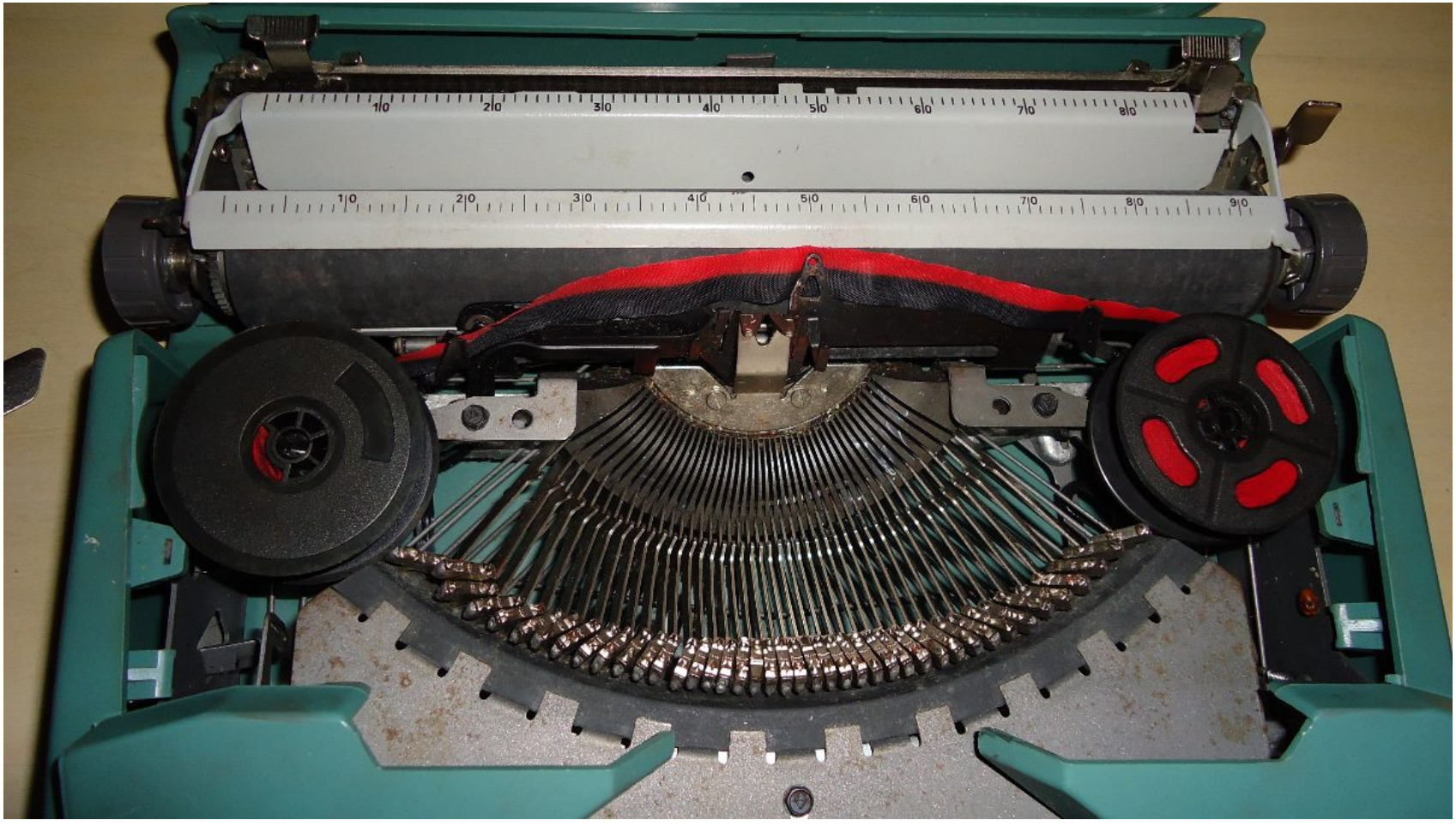
Árvores Rubro-Negras

UFABC

- Árvores de busca com garantias de altura máxima.
- Em outras palavras, árvores balanceadas em geral têm a garantia de que a sua altura é no máximo $O(\lg n)$.
- Árvores AVL e árvores rubro-negras são exemplos de árvores balanceadas de busca.

UFABC

- Árvores Rubro-Negras são árvores de busca balanceadas.
- Também chamadas de árvores vermelho-preto (en: *red-black trees*).
- Receberam este nome em um artigo de Guibas e Sedgewick em 1978.
  - Supostamente pois a impressora apenas era capaz de imprimir vermelho e preto...



Características

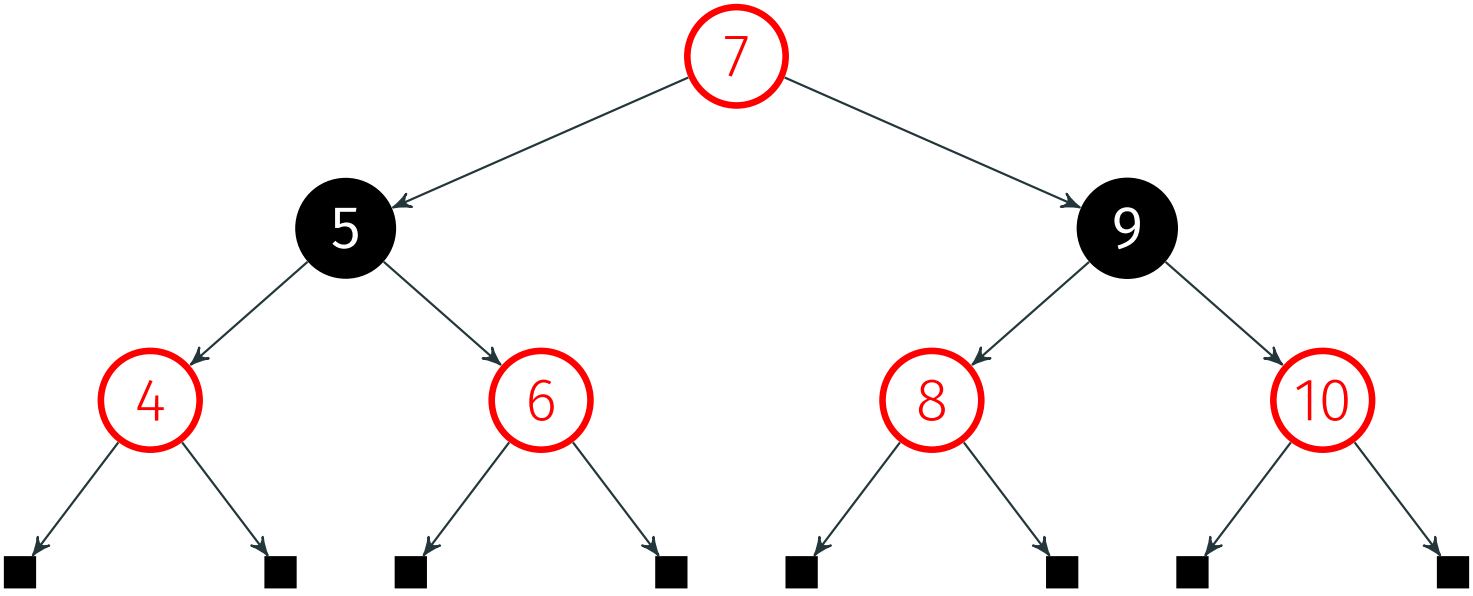
- Uma árvore rubro-negra é uma árvore de busca binária, logo segue todas as regras:
  - todo nó da sub-árvore esquerda de um nó p tem chave menor que a chave de p;
  - todo nó da sub-árvore direita de um nó p tem chave maior que a chave de p.
- Além disto, cada nó de uma árvore rubro negra tem as seguintes características:
  - Cor – vermelho ou preto.
  - Chave (ou valor) – Conteúdo do nó.
  - Dir, Esq – Sub-árvores direita e esquerda.

Regras de uma árvore rubro-negra

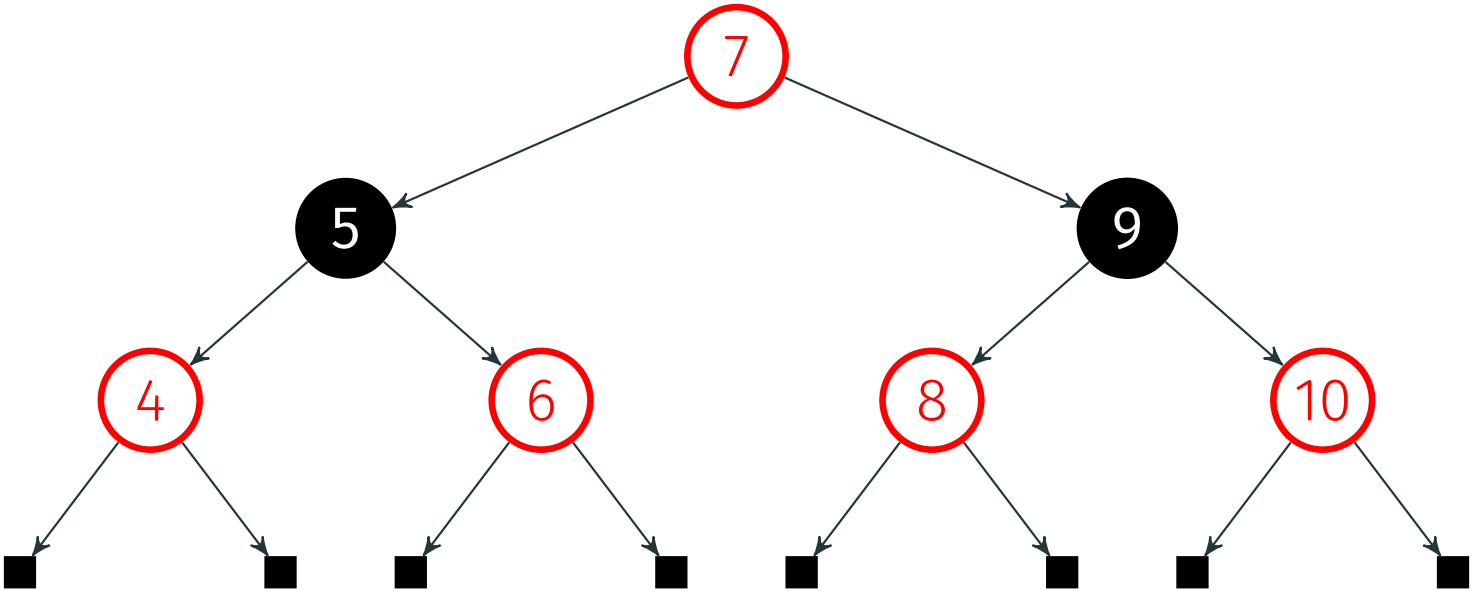
UFABC

- Regra -1: É uma árvore binária de busca.
- Regra 0: Os nós são vermelhos ou pretos.
- Regra 1: A raiz é sempre preta.
- Regra 2: Nenhum nó vermelho tem filhos vermelhos.
- Regra 3: Os nós *"nulos"* são considerados pretos.
- Regra 4: Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós pretos.

UFABC

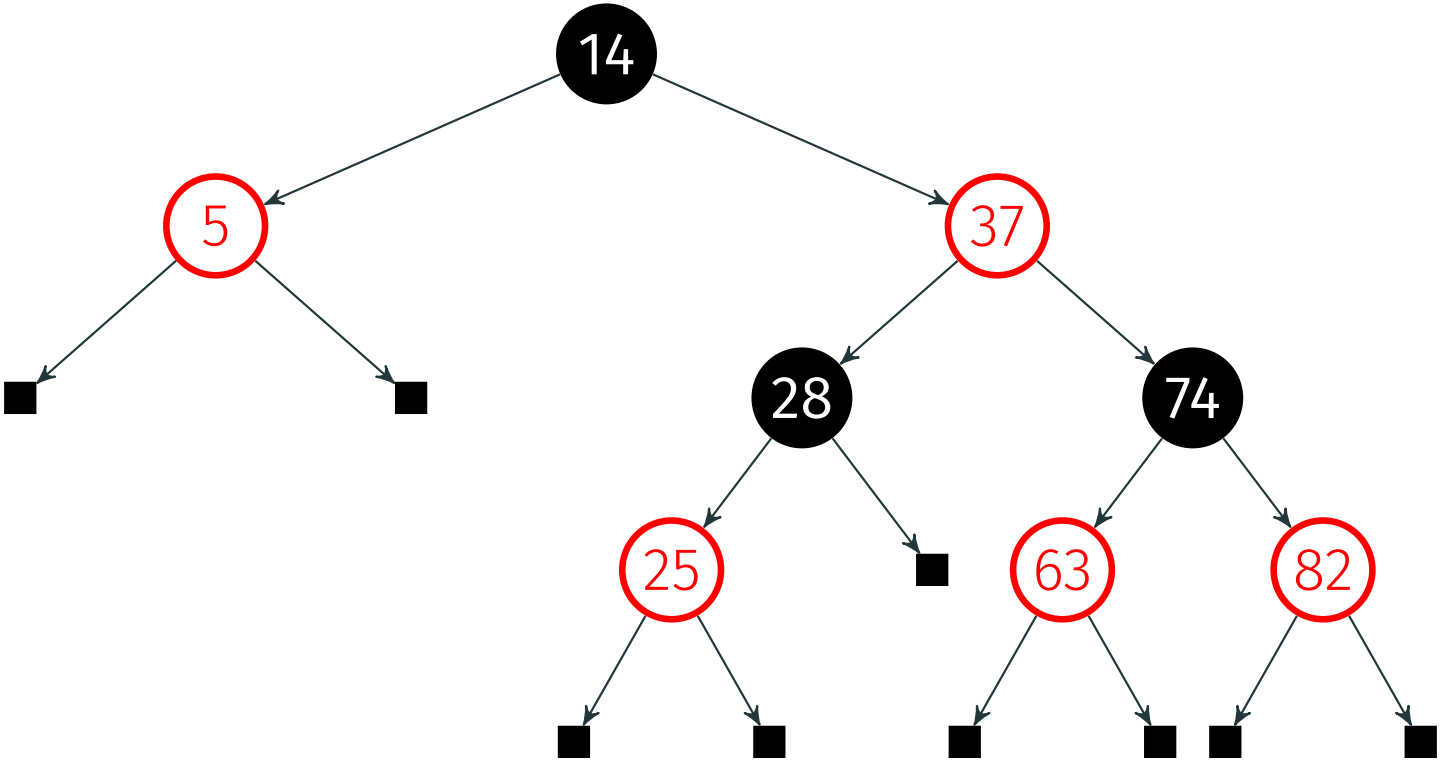


É uma árvore rubro-negra?

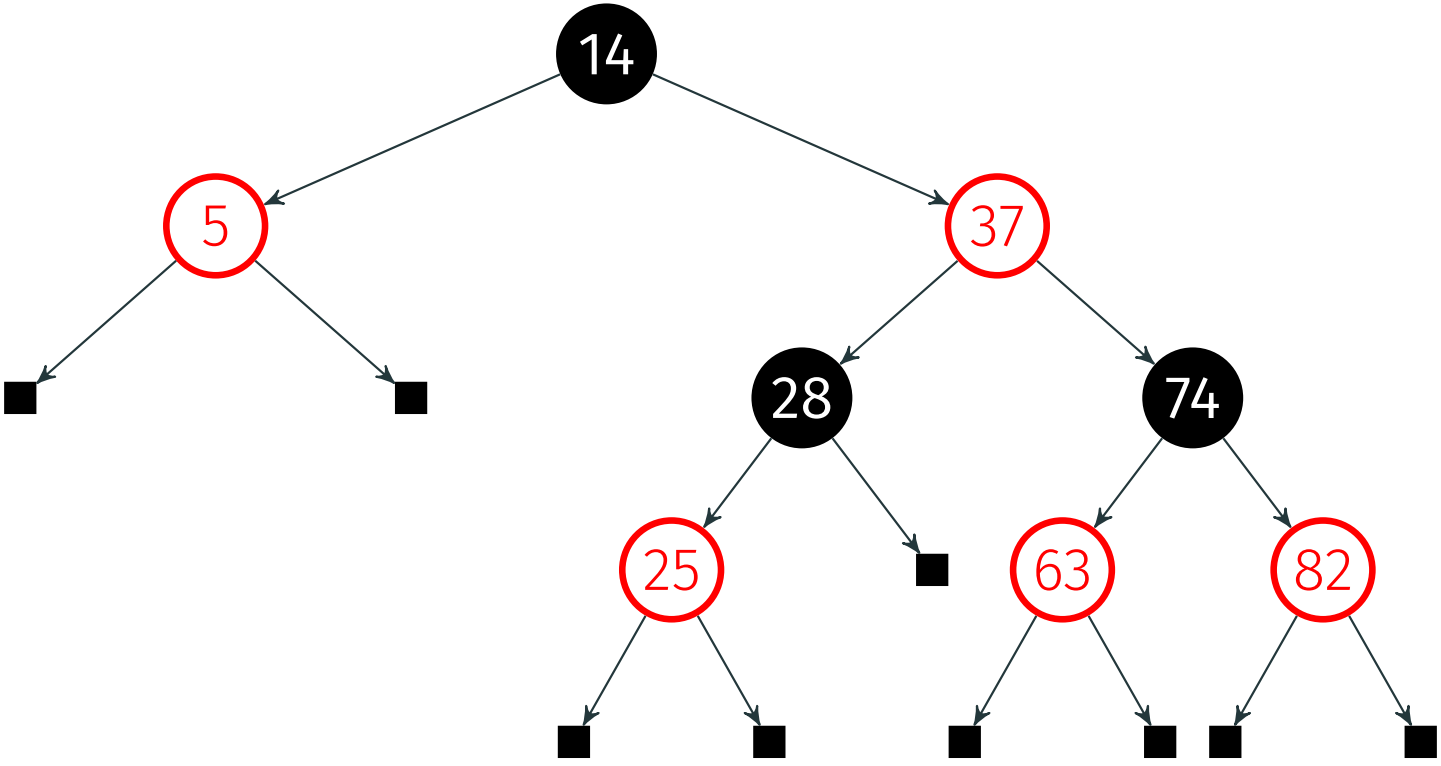


É uma árvore rubro-negra?

Não! Viola a Regra 1 (A raiz é sempre preta).



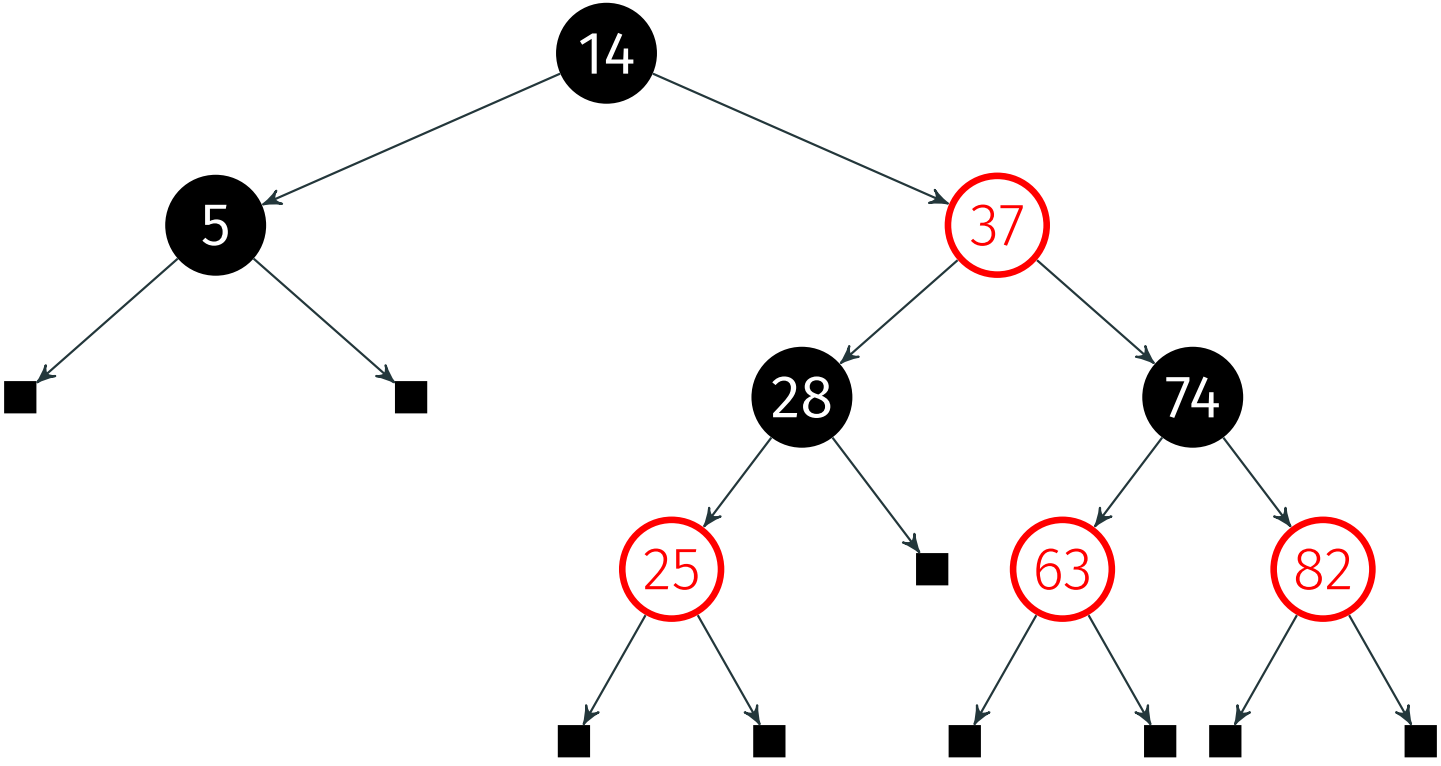
É uma árvore rubro-negra?



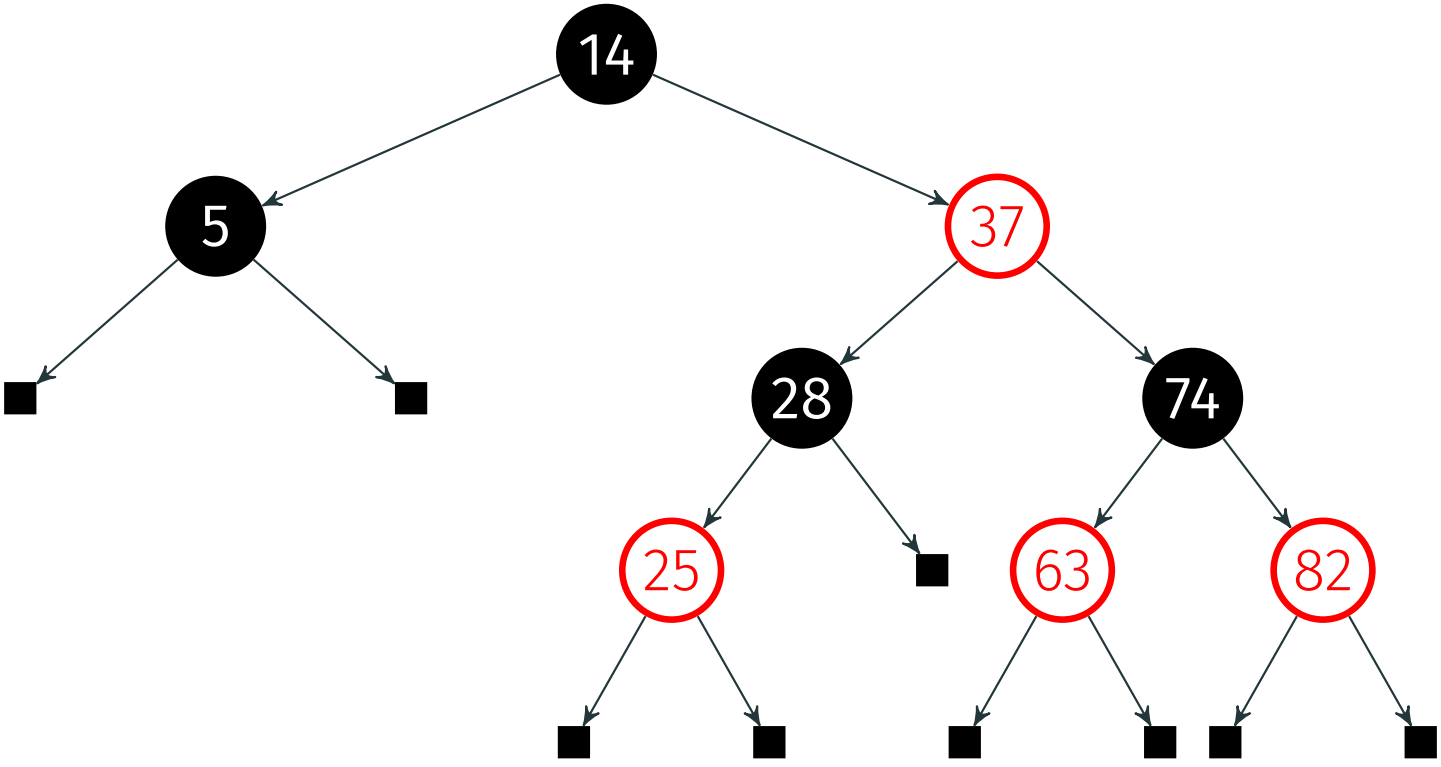
É uma árvore rubro-negra?

Não! Viola a Regra 4 (Para cada nó p, todos os caminhos desde p até as folhas contêm o mesmo número de nós pretos).

UFABC

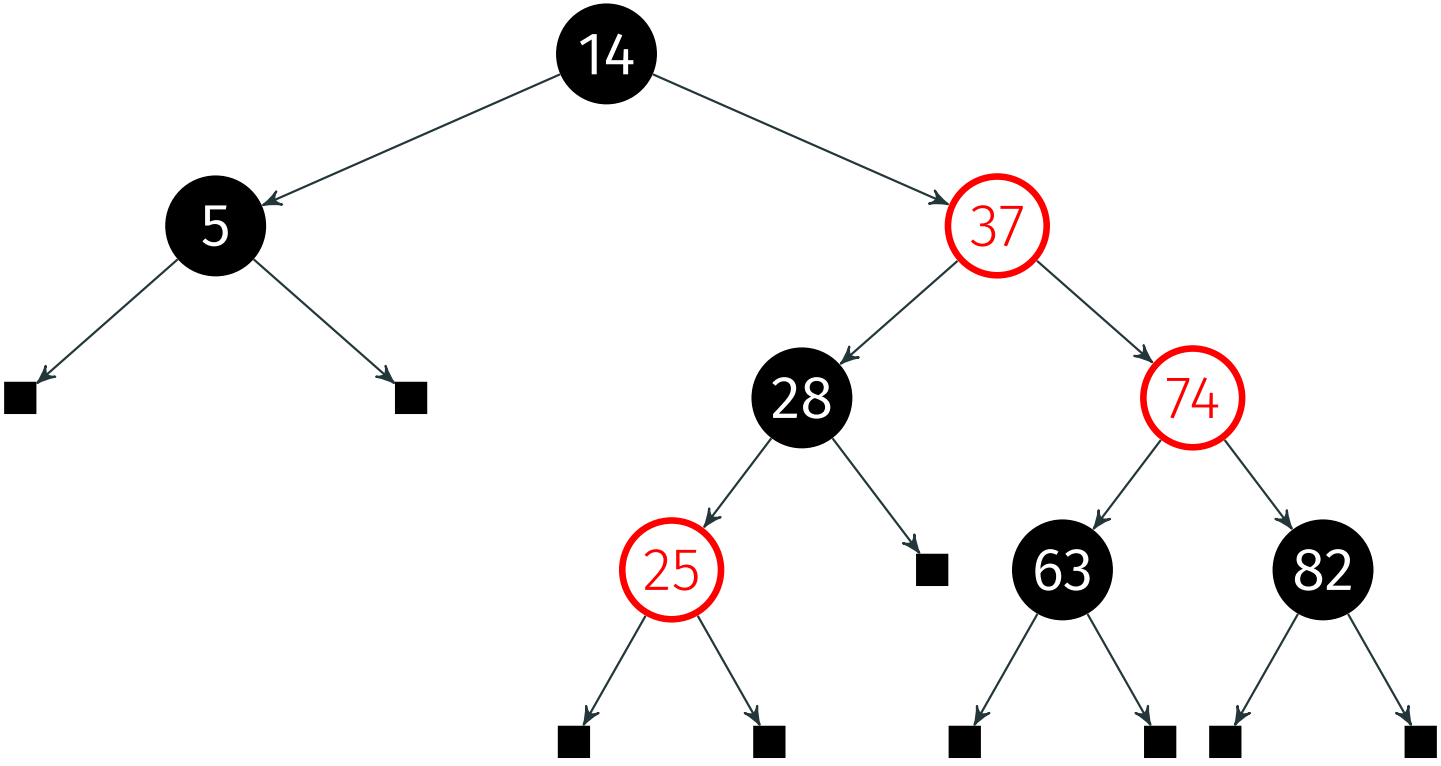


É uma árvore rubro-negra?

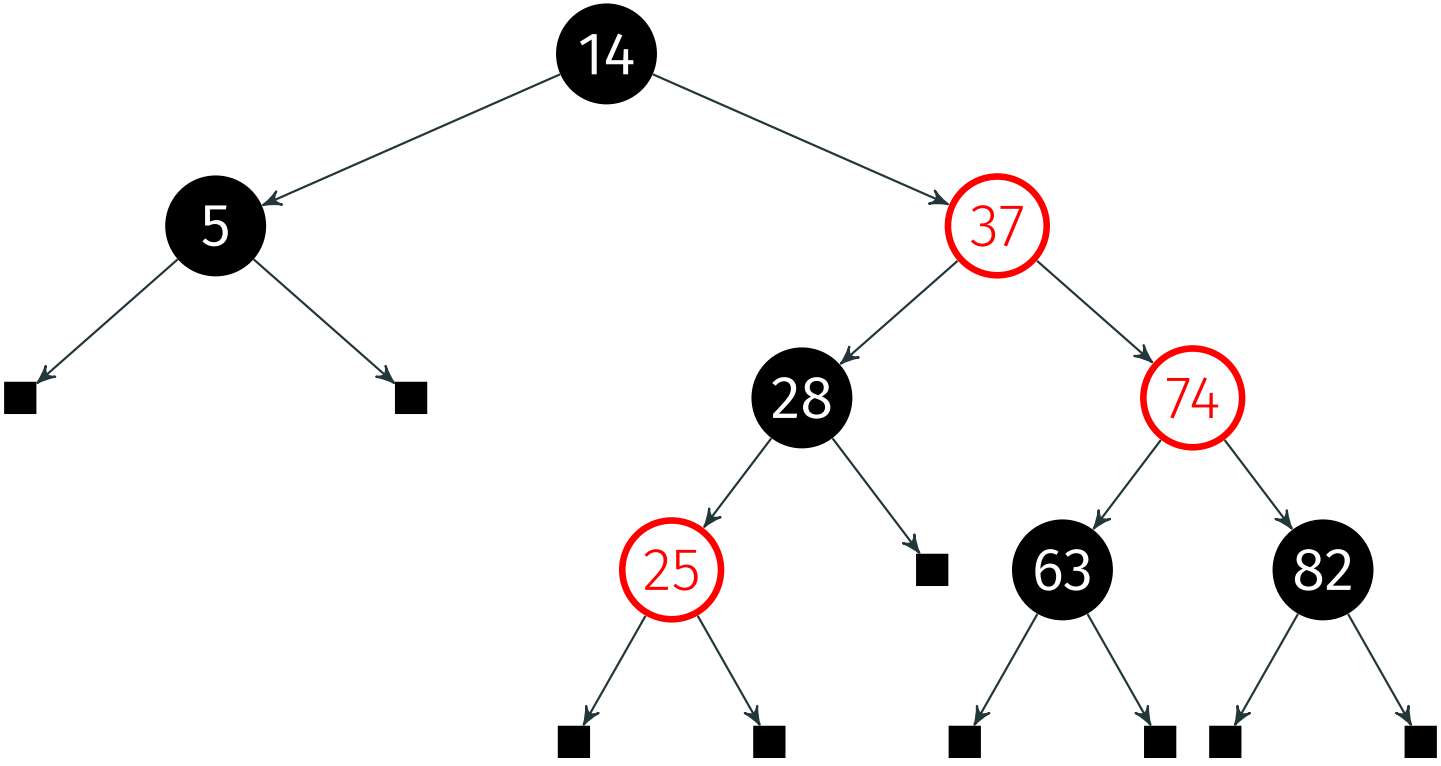


É uma árvore rubro-negra?

Sim!



É uma árvore rubro-negra?

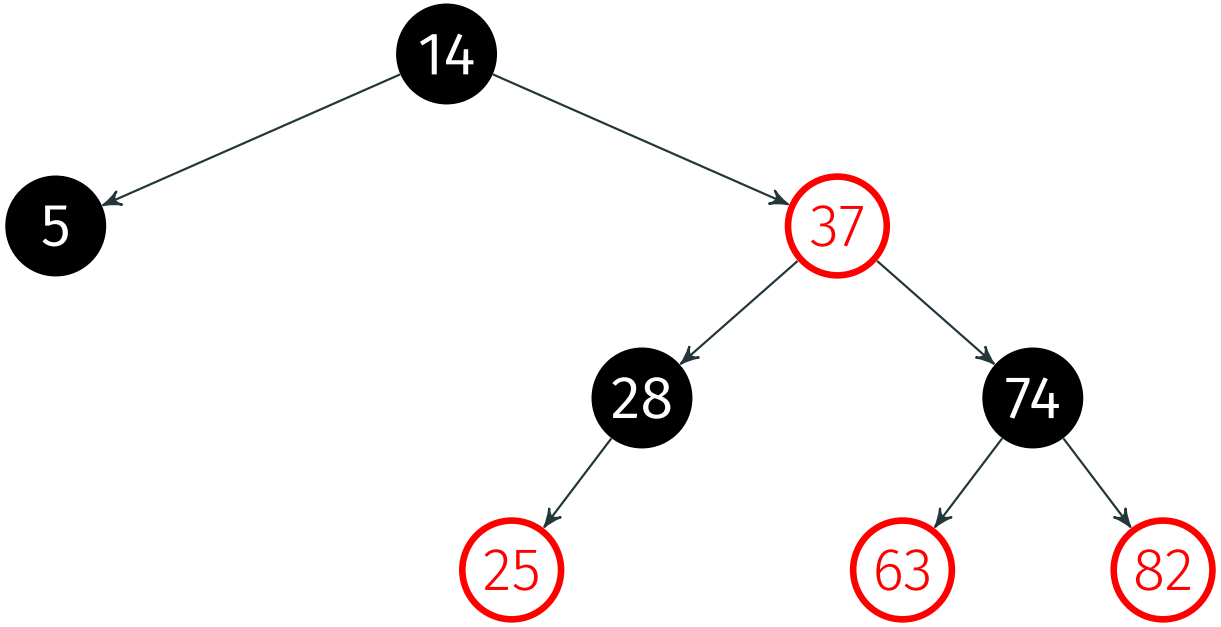


É uma árvore rubro-negra?

Não! Viola a Regra 2 (Nenhum nó vermelho tem filhos vermelhos).

UFABC

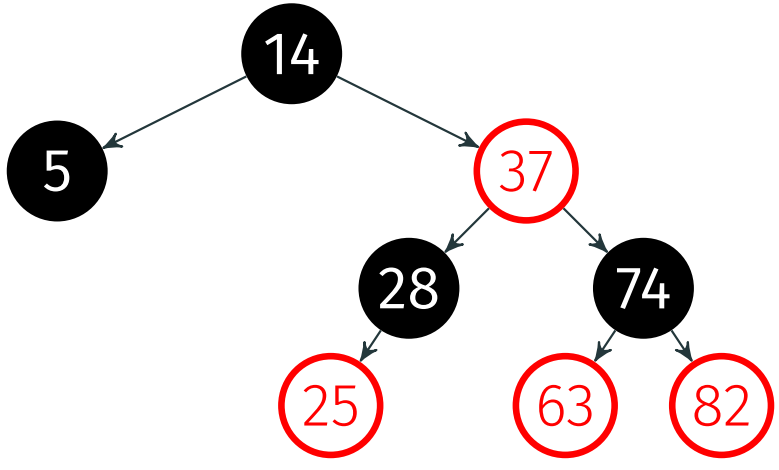
Por simplicidade, no restante das figuras vamos representar as árvores omitindo as folhas nulas (pretas). Uma árvore válida seria então:



Ok, pra quê tudo isso?

UFABC

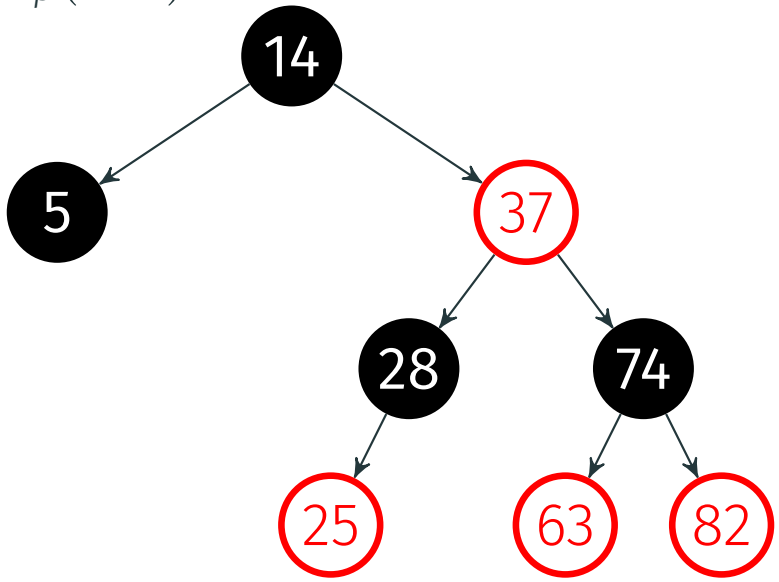
- Restringindo a maneira que os nós podem ser coloridos do caminho da raiz até qualquer uma das suas folhas, as árvores rubro-negras garantem:
  - que nenhum dos caminhos será maior que 2x o comprimento de qualquer outro.
  - que a árvore é aproximadamente balanceada.



UFABC

A altura negra de um nó n é definida como o número de nós pretos (sem incluir o próprio n) visitados em qualquer caminho de n até as folhas.

- A altura negra do nó n é denotada por $H_p(n)$.
- Pela Regra 4, $H_p(n)$ é bem definida para todos os nós da árvore.
  - A altura negra da árvore rubro-negra é definida como sendo a $H_p(\text{raiz})$.



Lema 1 - Altura máxima

UFABC

- Lema 1: A altura máxima de uma árvore rubro-negra com n nós internos é de $2\lg(n+1)$.
  - A prova pode ser feita por indução utilizando a H_p dos nós da árvore. Veja o Lema 13.1 do [CLRS] para a prova completa.
- Corolário: As operações de *Busca*, *Mínimo*, *Máximo*, *Sucessor e Predecessor* podem ser efetuadas em tempo $O(\lg(n))$.

UFABC

- Busca – A busca que estamos acostumados funciona sem modificações.
- Inserção e Remoção – Se feitas sem qualquer cuidado, apesar de manter as propriedades de árvores binárias de busca, podem ferir as propriedades rubro-negras.

  Veja animação de operações em:
  `http://tommikaikkonen.github.io/rbtree`

Primeiro definimos as cores:

```haskell
data RBColor = R | B
```

E em seguida, a árvore:

```haskell
data RBTree a where
  RBEmpty :: Ord a => RBTree a
  RBT :: Ord a => RBColor -> RBTree a -> a -> RBTree a
  ↪  -> RBTree a
```

Operações básicas

UFABC

```haskell
data RBTree a where
  RBEmpty :: Ord a => RBTree a
  RBT :: Ord a => RBColor -> RBTree a -> a -> RBTree a
  ↪  -> RBTree a
```

```haskell
empty :: Ord a => RBTree a
empty = RBEmpty

singleton :: Ord a => a -> RBTree a
singleton x = RBT B empty x empty

null :: RBTree a -> Bool
null RBEmpty = True
null _       = False

head :: RBTree p -> p
head (RBT _ _ x _) = x
head _ = error "head: empty Red Black Tree"
```

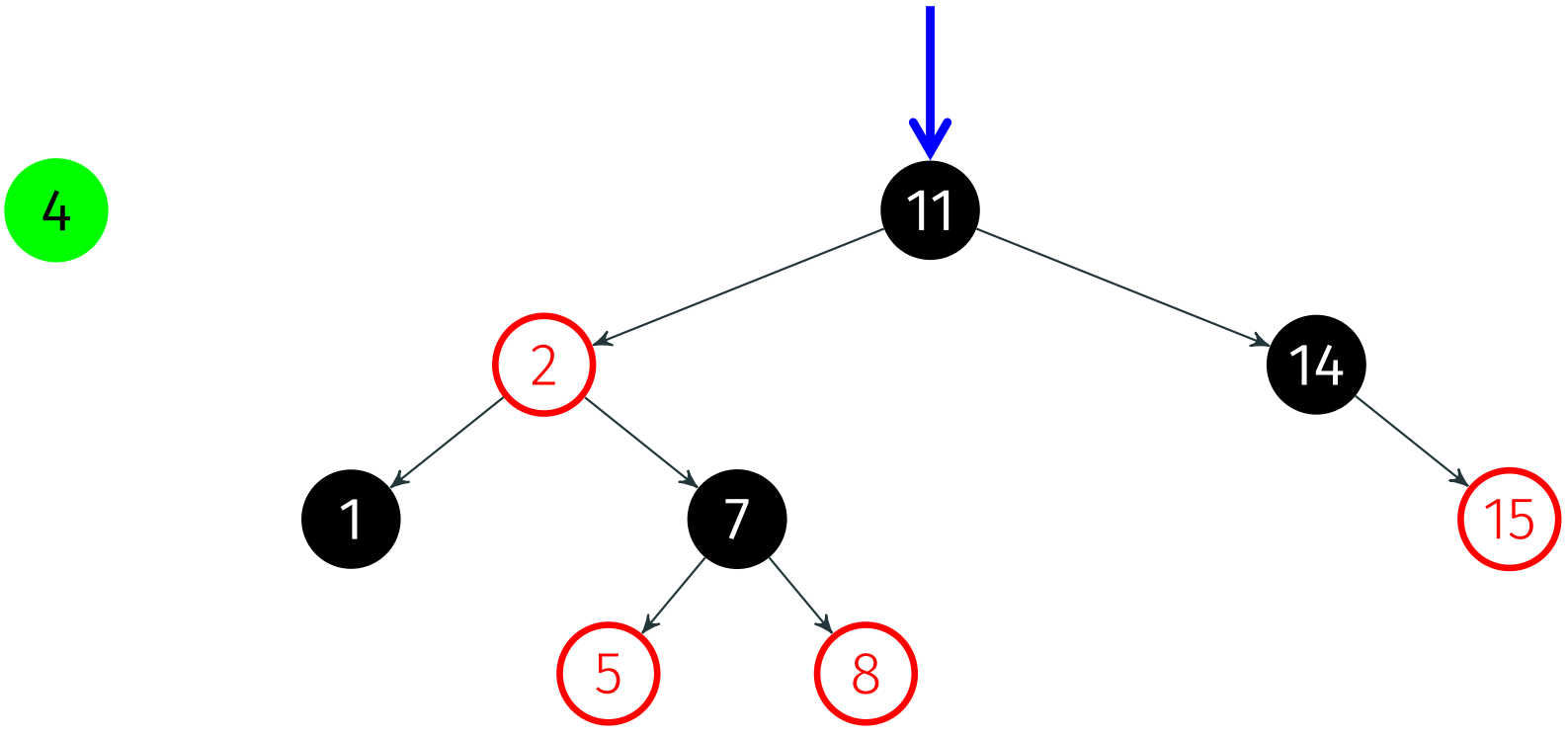
Busca e percurso ficam iguais...

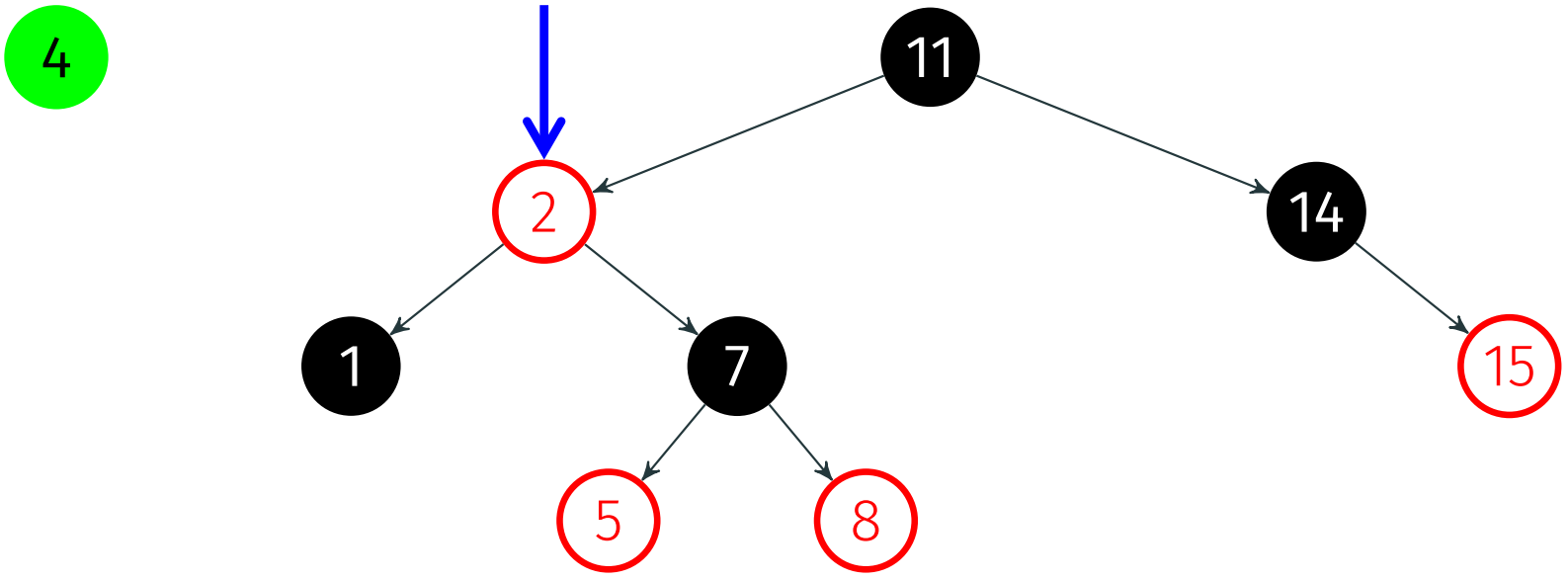
UFABC

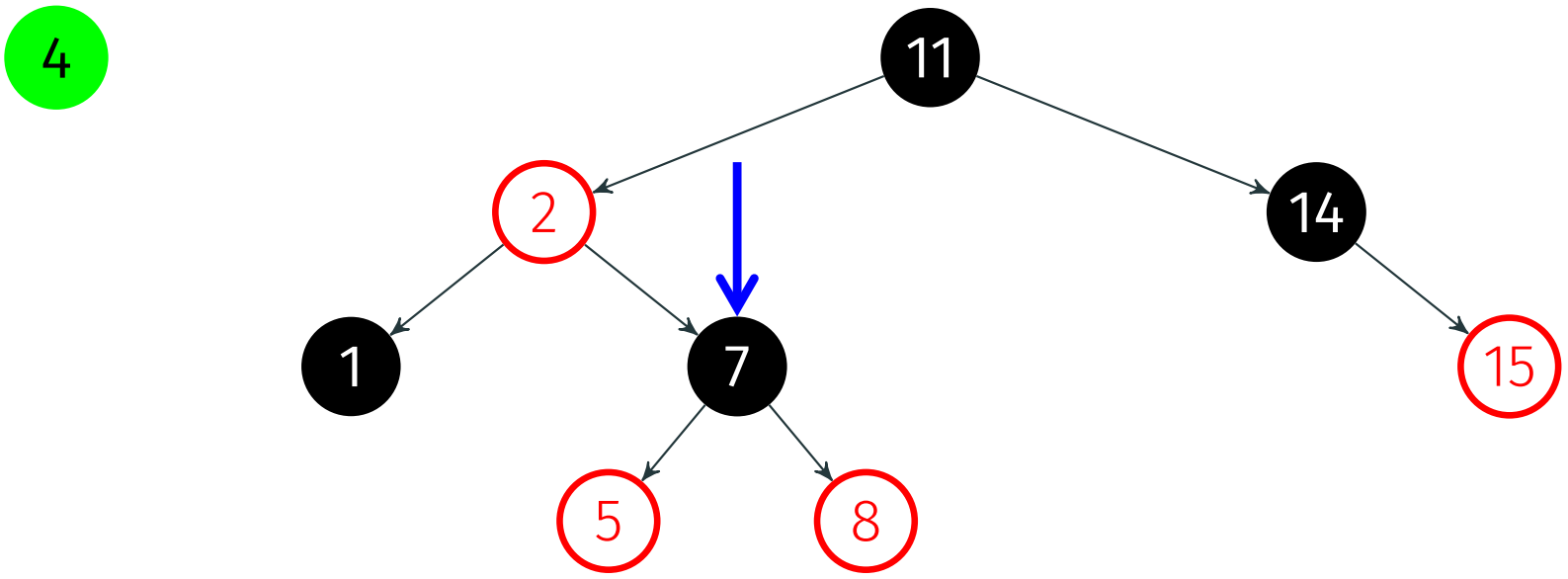
```haskell
data RBTree a where
  RBEmpty :: Ord a => RBTree a
  RBT :: Ord a => RBColor -> RBTree a -> a -> RBTree a
  ↪  -> RBTree a
```

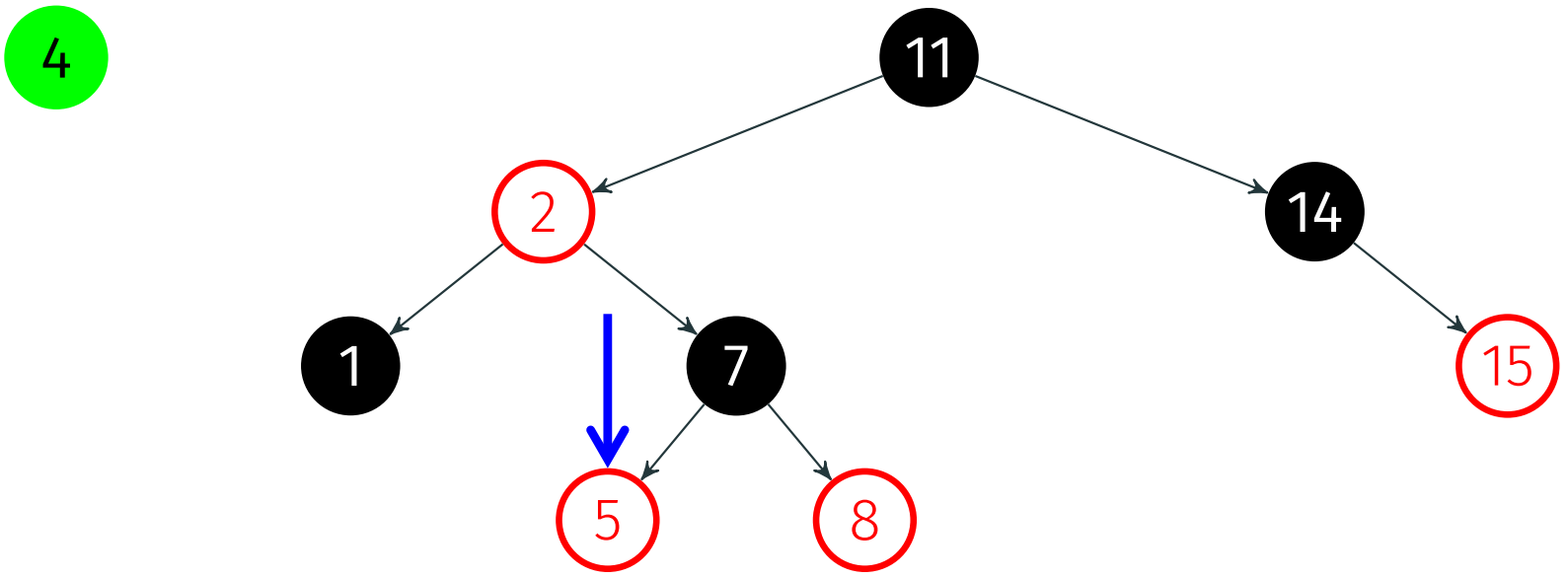
```haskell
elem :: a -> RBTree a -> Bool
elem _ RBEmpty = False
elem x (RBT _ l y r)
  | x < y = elem x l
  | x > y = elem x r
  | otherwise = True

-- Percurso in-ordem
toList :: RBTree a -> [a]
toList RBEmpty = []
toList (RBT _ l v r) =
  toList l ++  v : toList r
```

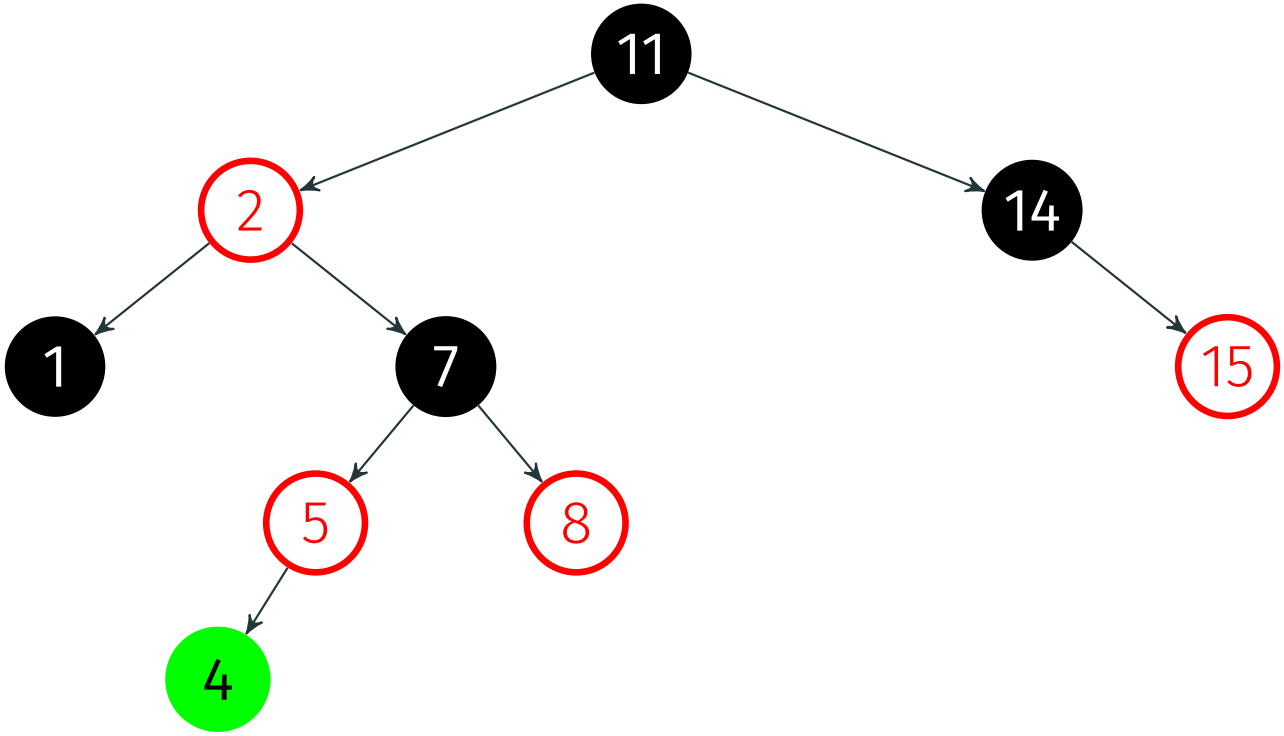
UFABC

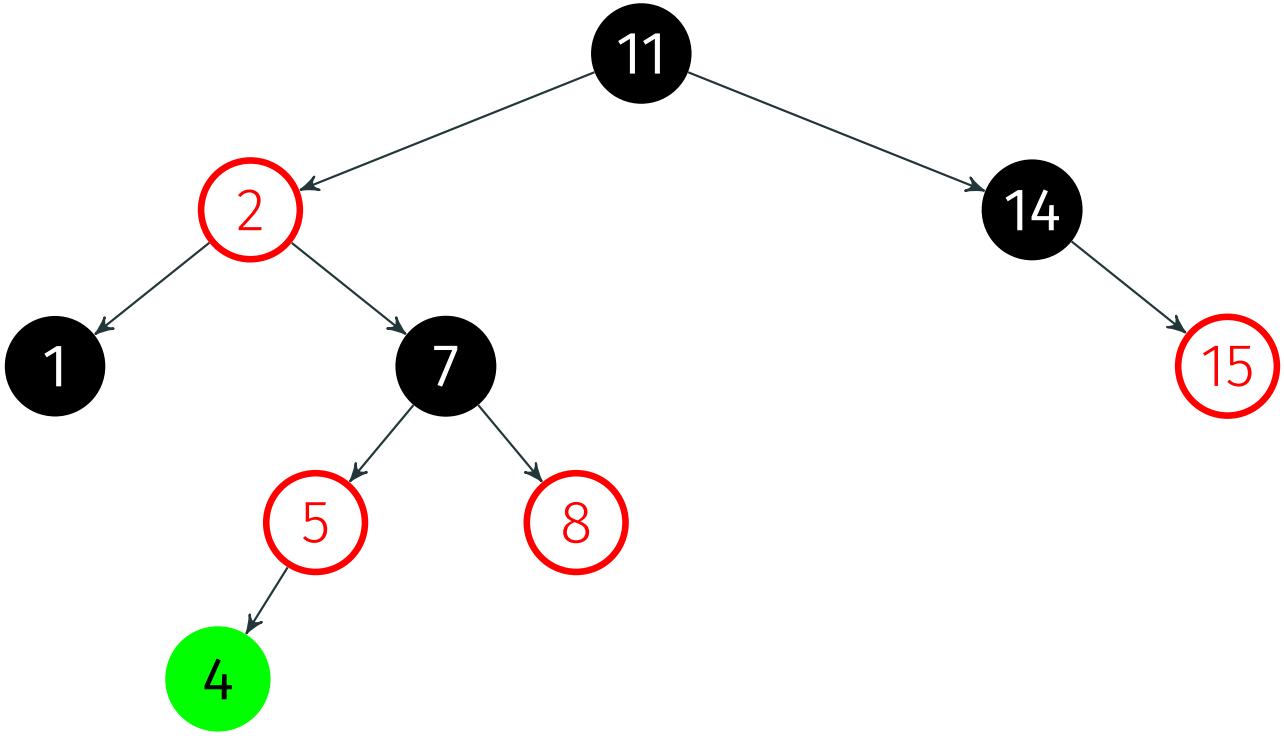



4
11
2
14
1
7
15
5
8


4
11
2
14
1
7
15
5
8


4
11
2
14
1
7
15
5
8


11
2
14
1
7
15
5
8
4

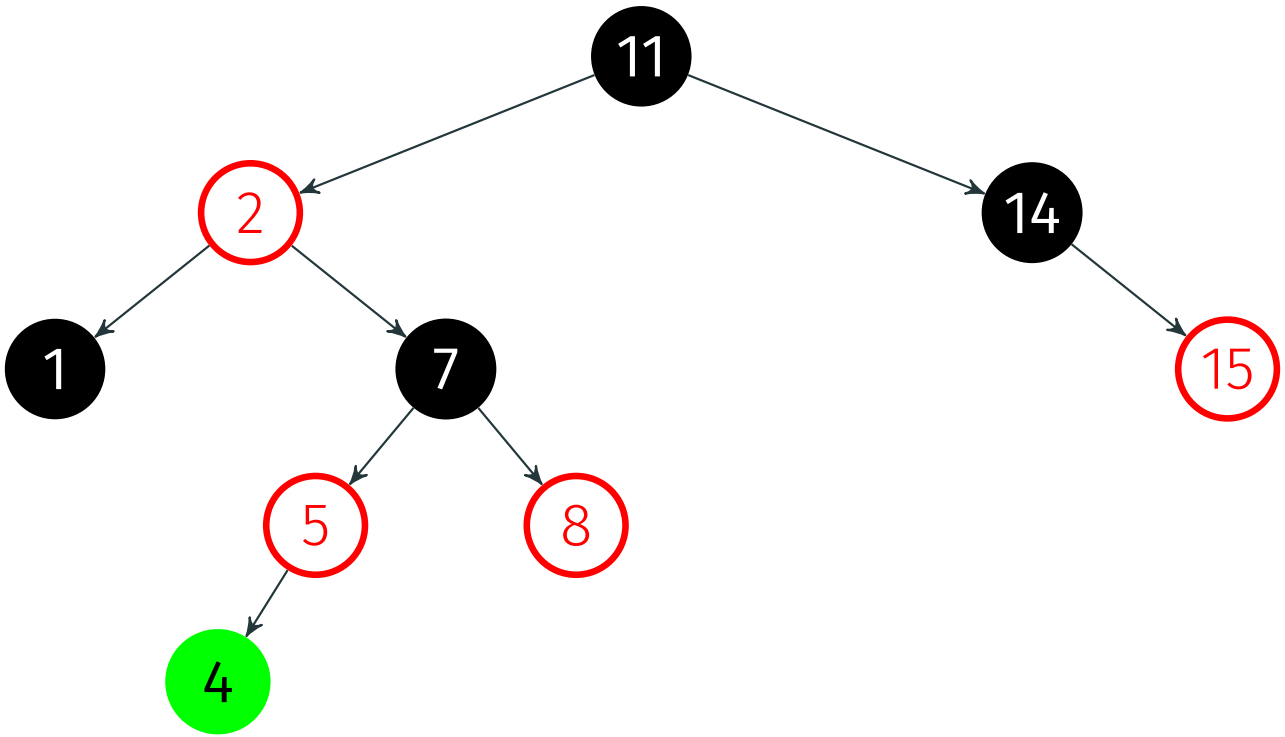


Quebra a Regra 0 – Todo nó deve ser vermelho ou preto.

Consertando a Regra 0

UFABC

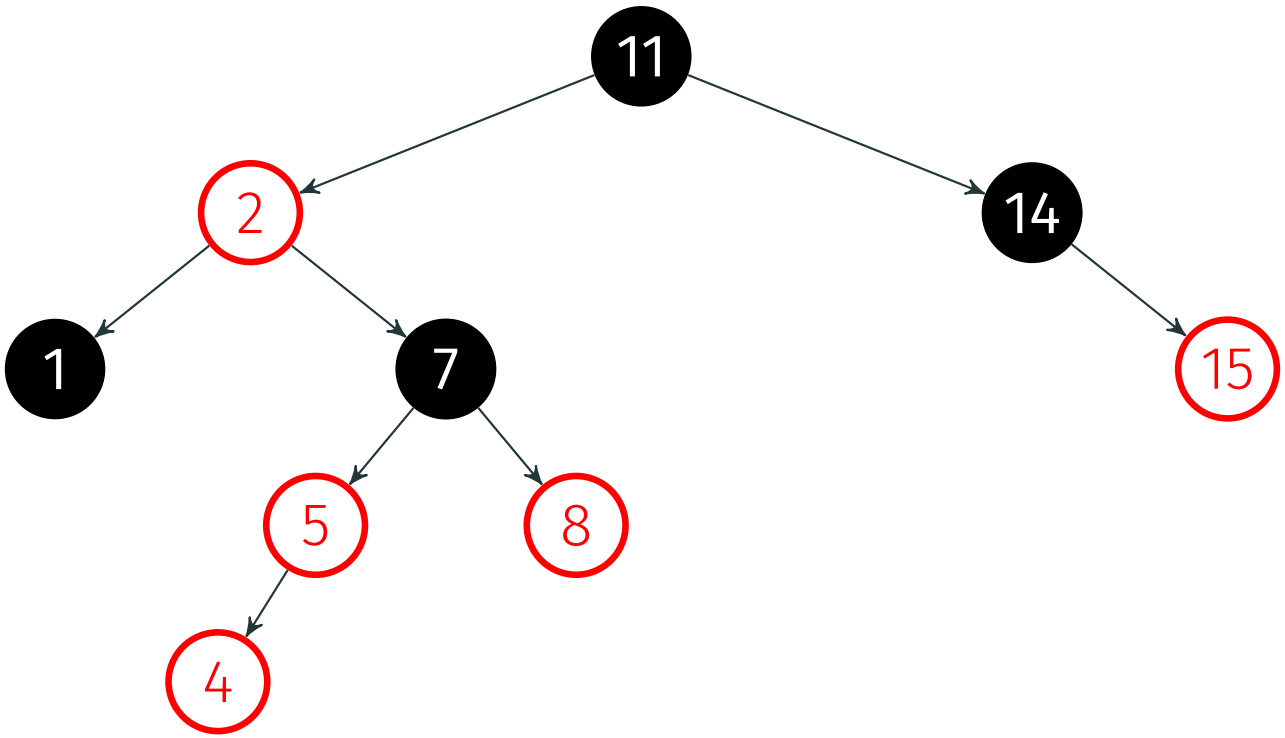
- É preciso decidir, qual faz menos mal, colocar um nó vermelho ou um preto?
  - O vermelho pode não quebrar nada.
  - O preto vai desequilibrar a altura negra da raiz, com certeza.



Consertando a Regra 0

UFABC

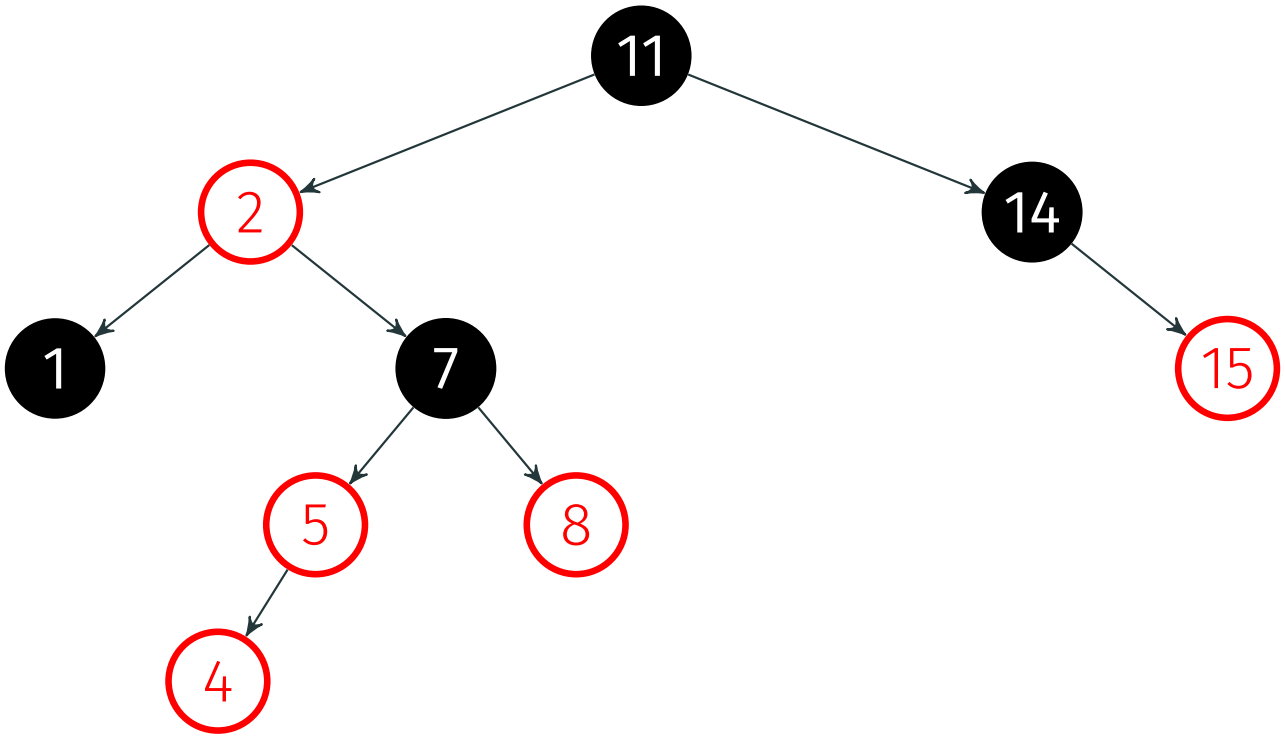
- É preciso decidir, qual faz menos mal, colocar um nó vermelho ou um preto?
  - O vermelho pode não quebrar nada.
  - O preto vai desequilibrar a altura negra da raiz, com certeza.



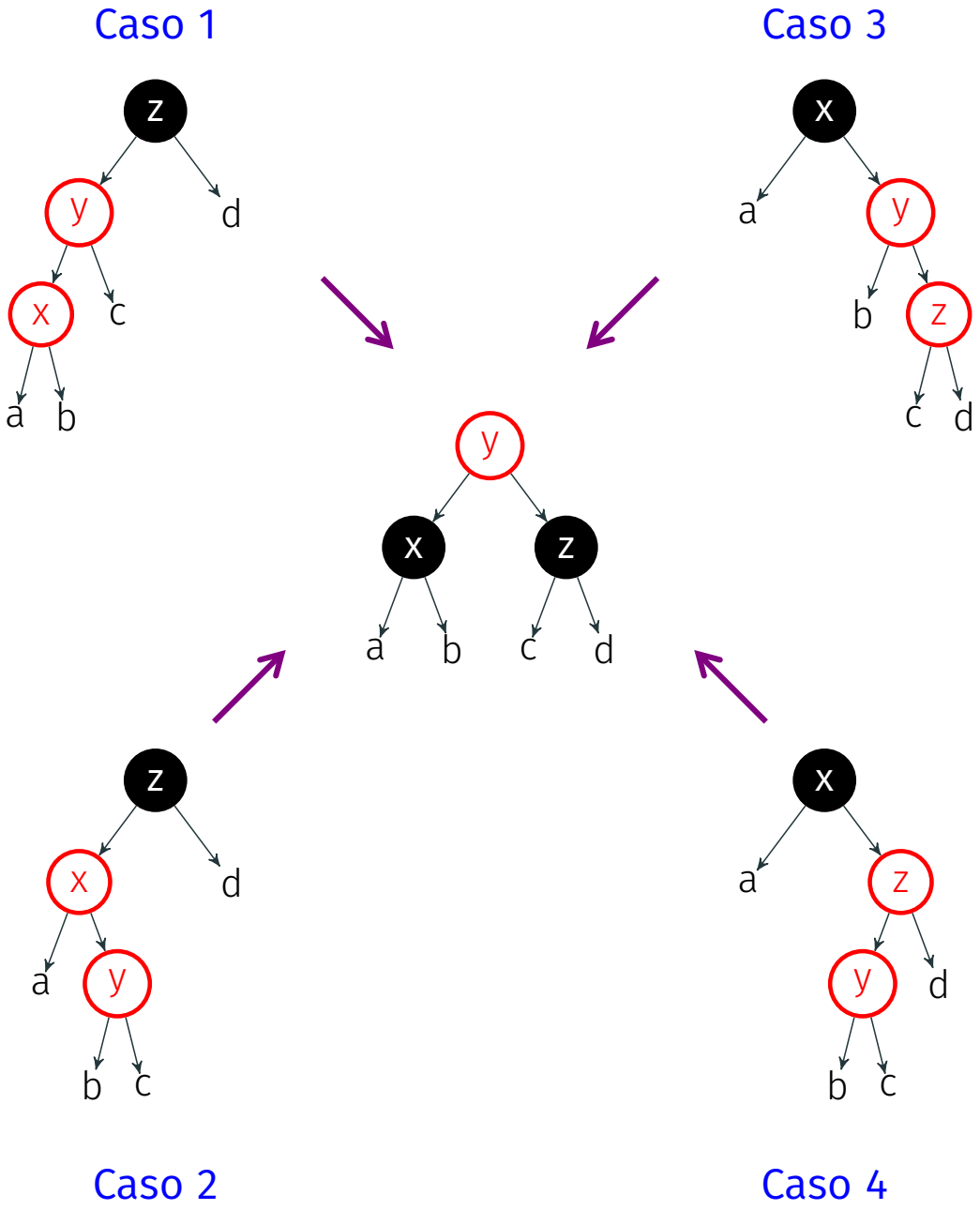
Mantendo as regras válidas

UFABC

- Regra 0 resolvida, sempre insiro um nó com a cor vemelha.
- E agora, qual regra eu quebrei?
  - Melhor ainda, quais regras eu poderia ter quebrado?

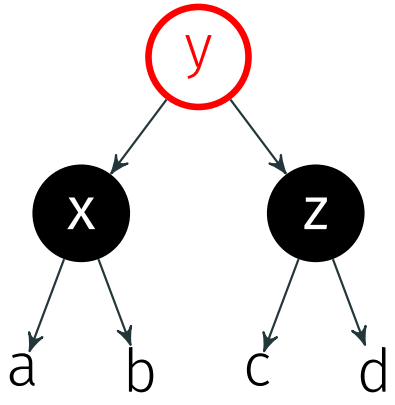


Rebalanceamento



UFABC

Primeiramente, criamos uma função que cria a árvore alvo após o balanceamento:

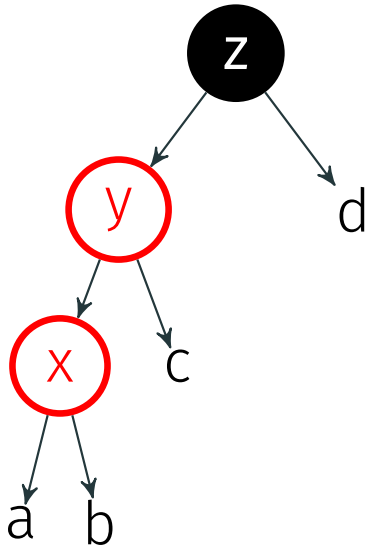



```haskell
buildRed :: Ord a => RBTree a -> a -> RBTree a -> a ->
  RBTree a -> a -> RBTree a -> RBTree a
buildRed a x b y c z d =
  RBT R (RBT B a x b) y (RBT B c z d)
```

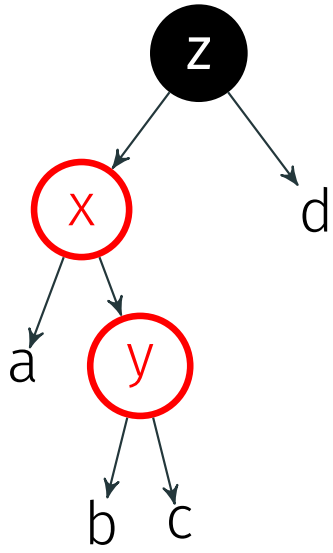
UFABC

Agora basta descrever as transições dos 4 casos:

Caso 1



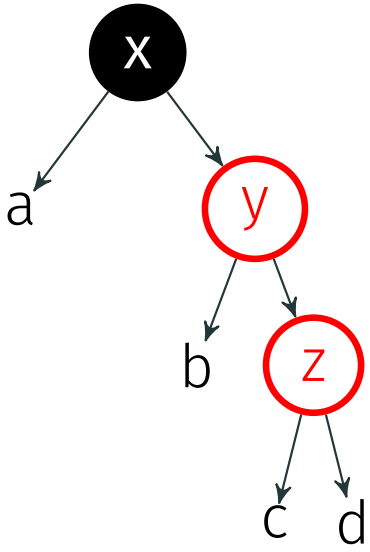
Caso 2



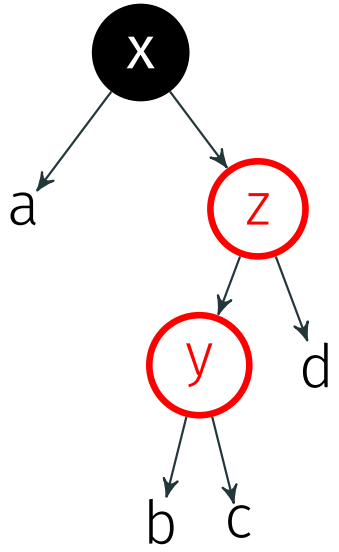

```haskell
balance :: Ord a => RBColor -> RBTree a -> a -> RBTree a
  -> RBTree a
balance B (RBT R (RBT R a x b) y c) z d = -- Caso 1
  buildRed a x b y c z d
balance B (RBT R a x (RBT R b y c)) z d = -- Caso 2
  buildRed a x b y c z d
```

UFABC

Caso 3



Caso 4




```haskell
balance B a x (RBT R b y (RBT R c z d)) = -- Caso 3
  buildRed a x b y c z d
balance B a x (RBT R (RBT R b y c) z d) = -- Caso 4
  buildRed a x b y c z d
balance color a x b =
  RBT color a x b
```

Inserção

UFABC

- Fazemos a inserção como sempre.
- Na linha 2, garantimos que a raiz continuará preta.
- As chamadas à `balance` garantem que as propriedades da árvore serão mantidas

```
1  insert :: Ord a => a -> RBTree a -> RBTree a
2  insert x t = makeBlack $ ins t
3    where
4      ins RBEmpty = singleton x
5      ins t2@(RBT color l y r)
6        | x < y = balance color (ins l) y r
7        | x > y = balance color l y (ins r)
8        | otherwise = t2
9
10     makeBlack ~(RBT _ a y b) = RBT B a y b
```

Sobre a implementação

UFABC

- Mesmo sem otimizações, esta é uma das implementações funcionais mais rápidas.
  - Os exercícios para casa tratam de possíveis otimizações que fazem essa implemtação voar!

Comparação com a versão imperativa

UFABC

Note

- Se comparada à implementação imperativa ([CLRS], [SW]), a implementação funcional é bem mais simples pois usa algumas transformações um pouco diferentes.
- Normalmente implementações imperativas dividem os 4 casos apresentados aqui em 8 casos de acordo com a cor do irmão do nó vermelho com filho vermelho.
- Saber a cor do ”tio”do nó vermelho permite, em alguns casos, utilizar menos atribuições ou terminar o rebalanceamento sem precisar propagá-lo até a raiz.
- Tais otimizações em uma implementação persistente são irrelevantes! Precisamos de qualquer forma copiar todo o caminho até a raiz, então não há razão de utilizar as transformações mais complicadas!

Danger

A implementação da inserção que apresentamos tem um erro grave! Você é capaz de apontá-lo?

```haskell
insert :: Ord a => a -> RBTree a -> RBTree a
insert x t = makeBlack $ ins t
  where
    ins RBEmpty = singleton x
    ins t2@(RBT color l y r)
      | x < y = balance color (ins l) y r
      | x > y = balance color l y (ins r)
      | otherwise = t2

    makeBlack ~(RBT _ a y b) = RBT B a y b
```

Corrigindo a inserção

UFABC

- A correção é simples, bastando fazer com que o nó inserido seja vermelho.

```haskell
insert :: Ord a => a -> RBTree a -> RBTree a
insert x t = makeBlack $ ins t
  where
    ins RBEmpty = RBT R empty x empty
    ins t2@(RBT color l y r)
      | x < y = balance color (ins l) y r
      | x > y = balance color l y (ins r)
      | otherwise = t2

    makeBlack ~(RBT _ a y b) = RBT B a y b
```

- Quais outros erros poderiam estar escondidos na nossa implementação? O compilador não poderia nos ajudar a detectá-los?

Comentários finais

- A versão de árvore rubro-negra que apresentamos aqui é devida a Chris Okasaki e foi apresentada em 1999. Antes disso as implementações funcionais eram uma adaptação forçada das implementações imperativas.
- Aqueles mais atentos vão perceber que houve a omissão de um tópico essencial: remoção funcional de nós.
  - Essa omissão também ocorre no livro do Okasaki [CO].
  - Assim como a inserção, a versão funcional ”redonda”de árvores rubro-negras demorou para aparecer. Em particular, a remoção só apareceu em 2014 em um paper apropriamente chamado (tradução minha): ”Remoções: a maldição das árvores rubro negras”.

Type-safe Red-Black Trees

UFABC

- Assim como fizemos com as Roseiras, vamos melhorar o nosso código para que erros como o que cometemos (e alguns outros) não sejam possíveis.
- Para isto vamos precisar relembrar um pouco de matemática.

Axiomas de Peano

UFABC

- Em 1879, Giuseppe Peano apresentou uma fundamentação (utilizando a linguagem da época) que admite três conceitos primitivos (número natural, zero e sucessor) relacionados entre si por cinco axiomas.
- Este conjunto de axiomas é a base para a formalização ordinal de números naturais.
- Mais detalhes em [MC].

Figura 3: Giuseppe Peano

Os axiomas de Peano

UFABC

- Indicaremos por $\sigma(n)$ o sucessor de n e, como usual, 0 para denotar o valor zero.
- Os cinco axiomas são:
  1. 0 é um número natural.
  2. Todo número natural n tem um sucessor $\sigma(n)$.
  3. 0 não é sucessor de nenhum número.
  4. Se $\sigma(n) = \sigma(m)$ então $n = m$.
  5. Princípio da indução completa: Seja S um conjunto de números naturais tal que: (i) $0 \in S$; e (ii) Se $n \in S$ então $\sigma(n) \in S$; então S é o conjunto de todos os números naturais.

- Criado por Alonzo Church na década de 1930.
  - Apresentado em 1932 e refinado até 1940 quando apresentou a sua versão tipada.
  - Church foi o orientador de doutorado do Alan Turing, que publicou o paper descrevendo máquinas de Turing em 1936.
  - Para saber mais sobre Cálculo-λ: aqui e [SK] Cap. 5.

Figura 4: Alonzo Church

Cálculo λ e Axiomas de Peano

- Church apresentou uma maneira de representar inteiros em Cálculo-λ utilizando funções anônimas.
- Essa representação ficou conhecida como Números de Church.
- Um número **n** é codificado como a chamada de uma função **n** vezes:

```
1  ZERO = \f x -> x
2  UM   = \f x -> f x
3  DOIS = \f x -> f (f x)
4  TRES = \f x -> f (f (f x))
```

- Que nada mais é que a ideia da função σ apresentada por Peano!

Saindo de valores e indo para tipos

UFABC

- A implementação sugerida anteriormente é baseada em valores disponíveis apenas em tempo de execução.
- Queremos usar essa garantia em tempo de compilação![9].

```haskell
-- Inteiros de Peano
data Peano = Zero | Succ Peano
```

E os números naturais passam a ser...

```haskell
type One = Succ Zero
type Two = Succ One
type Three = Succ Two
...
```

[9] Essa implementação usa a extensão `DataKinds`

Pronto! Já temos o que precisamos!

UFABC

- Vamos relembrar as regras de uma árvore rubro-negra:
  - Regra 0: Os nós são vermelhos ou pretos ✗
  - Regra 1: A raiz sempre é preta ✗
  - Regra 2: Nenhum nó vermelho tem filhos vermelhos ✗
  - Regra 3: Os nós nulos são pretos ✗
  - Regra 4: Altura negra à esquerda e à direita iguais ✗

... e nossa implementação atual:

```haskell
data RBColor = R | B
```

```haskell
data RBTree a where
  RBEmpty :: Ord a => RBTree a
  RBT :: Ord a => RBColor -> RBTree a -> a -> RBTree a
  ↪  -> RBTree a
```

Vamos começar pelo mais fácil... Regras 0 e 3

UFABC

- Regra 0: Os nós são vermelhos ou pretos ✓
- Regra 1: A raiz sempre é preta ✗
- Regra 2: Nenhum nó vermelho tem filhos vermelhos ✗
- Regra 3: Os nós nulos são pretos ✓
- Regra 4: Altura negra à esquerda e à direita iguais ✗

```haskell
data Color = R | B
type Black = Node B -- Apenas um atalho
type Red   = Node R

data Node (c :: Color) a where -- regra 0
  Null :: Ord a => Black a -- regra 3
  RBT :: Ord a => c -> Node c0 a -> a -> Node c1 a ->
  ↪  Node c a
```

Regra 1

UFABC

- Regra 0: Os nós são vermelhos ou pretos ✓
- Regra 1: A raiz sempre é preta ✓
- Regra 2: Nenhum nó vermelho tem filhos vermelhos ✗
- Regra 3: Os nós nulos são pretos ✓
- Regra 4: Altura negra à esquerda e à direita iguais ✗

```haskell
data Color = R | B
type Black = Node B
type Red   = Node R

data RBTree a = Black a -- regra 1

data Node (c :: Color) a where -- regra 0
  Null :: Ord a => Black a -- regra 3
  RBT :: Ord a => c -> Node c0 a -> a -> Node c1 a ->
  ↪  Node c a
```

Regra 2

UFABC

- Regra 0: Os nós são vermelhos ou pretos ✓
- Regra 1: A raiz sempre é preta ✓
- Regra 2: Nenhum nó vermelho tem filhos vermelhos ✓
- Regra 3: Os nós nulos são pretos ✓
- Regra 4: Altura negra à esquerda e à direita iguais ✗

```haskell
data Color = R | B
type Black = Node B
type Red   = Node R

data RBTree a = Black a -- regra 1

data Node (c :: Color) a where -- regra 0
  Null :: Ord a => Black a -- regra 3
  Black::Ord a => Node c0 a -> a -> Node c1 a -> Black a
  -- regra 2
  Red :: Ord a => Black a -> a -> Black a -> Red a
```

Regra 4

UFABC

- Regra 0: Os nós são vermelhos ou pretos ✓
- Regra 1: A raiz sempre é preta ✓
- Regra 2: Nenhum nó vermelho tem filhos vermelhos ✓
- Regra 3: Os nós nulos são pretos ✓
- Regra 4: Altura negra à esquerda e à direita iguais ✓

```haskell
data Peano = Zero | Succ Peano
type One = Succ Zero

data RBTree a = forall n. T (Black n a) -- regra 1

data Node (c :: Color) (n :: Peano) a where -- regra 0
  Null  :: Ord a => Black One a -- regra 3, 4
  Black :: Ord a => Node c0 n a -> a -> Node c1 n a ->
  ↪  Black (Succ n) a -- regra 4
  -- regra 2 e 4
  Red :: Ord a => Black n a -> a -> Black n a -> Red n a
```

gRReat success!

#SQN

UFABC

A busca precisa apenas de algumas pequenas alterações para funcionar na nova ED:

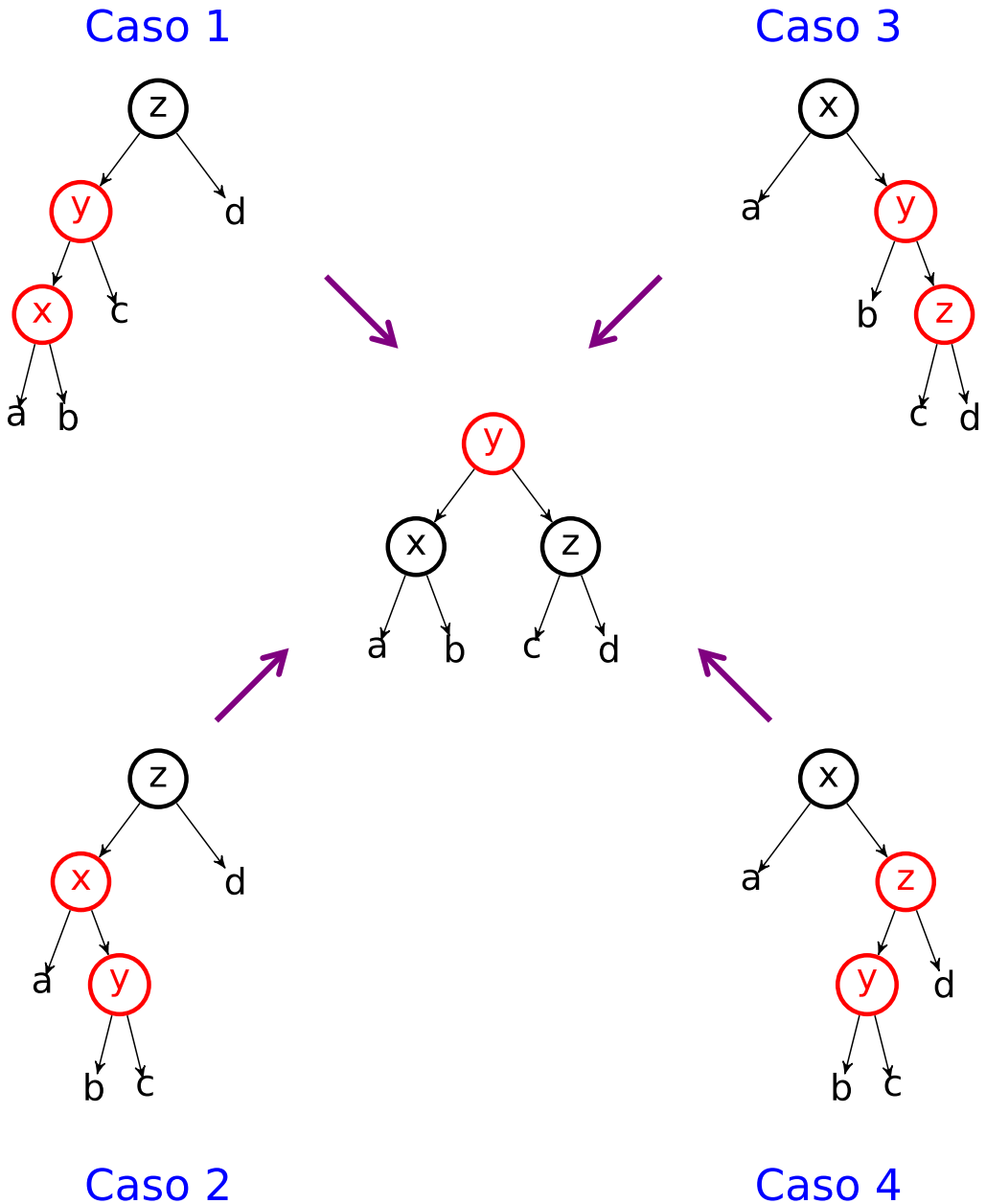
```haskell
elem :: Ord a => a -> RBTree a -> Bool
elem x (T node) = elemNode x node

elemNode :: a -> Node c n a -> Bool
elemNode x node =
  case node of
     Null -> False
     (Black l y r) -> elem' x l y r
     (Red l y r) -> elem' x l y r
  where
    elem' e l y r
      | e < y = elemNode e l
      | e > y = elemNode e r
      | otherwise = True
```

UFABC

- A estratégia que utilizamos antes de:
  - Inserir um novo nó vermelho.
  - Verificar se há um problema e propagar as correções até a raiz...
- ... não funciona mais!
- Os tipos proibem que criemos uma árvore inválida mesmo que temporariamente!
- Precisamos montar uma árvore válida diretamente.

Relembrando o rebalanceamento

UFABC



Consertando a inserção

UFABC

- A ideia da implementação se baseia na seguinte observação: a inserção de um novo nó vermelho abaixo de...
  - ... um nó vermelho pode causar uma violação das regras.
  - ... um nó preto não gera uma violação das regras.
- Assim, dividimos a implementação da inserção em cores!

O tipo `Violation`

UFABC

```haskell
data Violation (n :: Peano) a where
  Case14 -- Nós vermelhos à esq. esq.
    :: Ord a =>    -- Caso 1 Caso 4
       a           -- Red y  Red z
    -> a           -- Red x  Red y
    -> Black n a   -- a      b
    -> Black n a   -- b      c
    -> Black n a   -- c      d
    -> Violation n a
  Case23 -- Nós vermelhos à dir. dir.
    :: Ord a =>    -- Caso 2 Caso 3
       a           -- Red x  Red y
    -> a           -- Red y  Red z
    -> Black n a   -- a      b
    -> Black n a   -- b      c
    -> Black n a   -- c      d
    -> Violation n a
```

Inserção abaixo de um nó vermelho

UFABC

```haskell
-- insere o elemento x0 na árvore de raiz vermelha n
-- Fica sob responsabilidade do pai a correção de
-- eventuais violações.
insR :: a -> Red n a -> Either (Violation n a) (Red n a)
insR x0 n@(Red l0 y0 r0) -- l0 e r0 são pretos
  | x0 < y0 =
    case insB x0 l0 of
      (Left black)          -> mkRed black y0 r0
      (Right (Red a x b)) -> Left $ Case14 y0 x a b r0
  | x0 > y0 =
    case insB x0 r0 of
      (Left black)          -> mkRed l0 y0 black
      (Right (Red b y c)) -> Left $ Case23 y0 y l0 b c
  | otherwise = Right n
  where
    mkRed a v c = Right $ Red a v c
```

Inserção abaixo de um nó preto i

UFABC

```haskell
-- insere o elemento x0 na árvore de raiz negra e
-- devolve um novo no pode ser vermelho ou negro
insB :: a -> Black n a -> Either (Black n a) (Red n a)
insB x0 Null = Right $ Red Null x0 Null
insB x0 n@(Black l0 y0 r0)
  | x0 < y0 =
    case l0 of
      Null -> eitherInsBL x0 Null y0 r0
      black@Black{} -> eitherInsBL x0 black y0 r0
      red@Red{} ->
        case insR x0 red of
          (Left (Case14 y x a b c)) -> -- Caso 1
            balance a x b y c y0 r0
          (Left (Case23 x y a b c)) -> -- Caso 2
            balance a x b y c y0 r0
          (Right r) ->
            mkBlack y0 r r0
```

Inserção abaixo de um nó preto ii

UFABC

```
18
19      -- O caso de x0 > y0 é semelhante, porém pode causar
20      -- as violações 3 ou 4
21      | x0 > y0 =
22        case r0 of
23          Null -> eitherInsBR x0 l0 y0 Null
24          black@Black{} -> eitherInsBR x0 l0 y0 black
25          red@Red{} ->
26            case insR x0 red of
27              (Left (Case14 z y b c d)) -> -- Caso 4
28                balance l0 y0 b y c z d
29              (Left (Case23 y z b c d)) -> -- Caso 3
30                balance l0 y0 b y c z d
31              (Right r) ->
32                mkBlack y0 l0 r
33
34      -- Neste caso o valor já estava na árvore
35      | otherwise = Left n
```

Inserção abaixo de um nó preto iii

UFABC

```
36      where
37        mkBlack x a b = Left $ Black a x b
38        eitherInsBR x l y r = either (mkBlack y l) (mkBlack y l)
          ↪  (insB x r)
39        eitherInsBL x l y r = either (flip (mkBlack y) r) (flip
          ↪  (mkBlack y) r) (insB x l)
40        balance a x b y c z d =  Right $ Red (Black a x b) y (Black
          ↪  c z d)
```

UFABC

E finalmente a inserção na árvore propriamente dita

```haskell
-- insere o elemento x na árvore t
insert :: a -> RBTree a -> RBTree a
insert x (T node) =
  -- pela regra 1 (e o tipo de RBTree) sabemos que node
  -- é preto, então insB type-checks
  case insB x node of
    (Left b) -> T b
    (Right (Red l v r)) -> T $ Black l v r
```

Agora sim!

Quase!

Ainda falta uma regra

UFABC

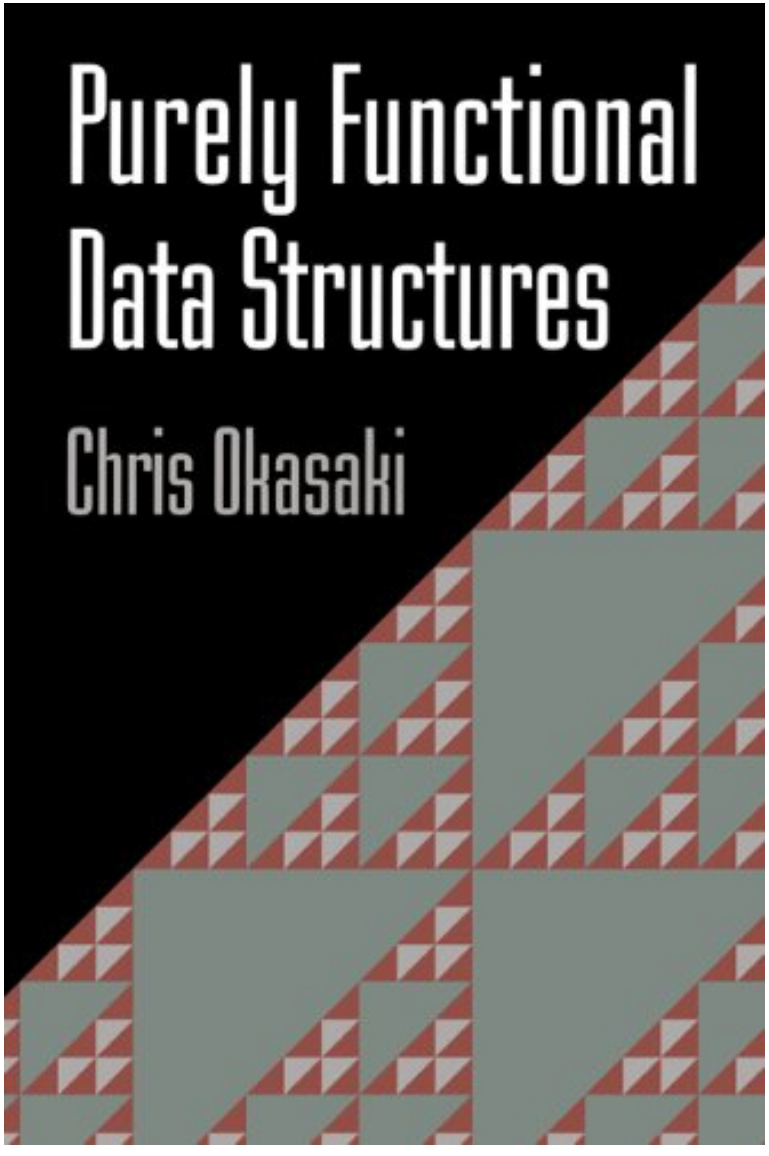
- Regra -1: A árvore é de busca ✗
- Infelizmente, para fazer a verificação em tempo de compilação, precisamos de uma linguagem com tipos dependentes (en: *dependent types*) como Idris.
  - Reveja o comentário quando falamos sobre este assunto relacionado às Roseiras.
- O máximo que podemos fazer, enquanto as propostas de tipos dependentes para Haskell ainda não estão prontas, é uma verificação em tempo execução 😵

Comentários finais

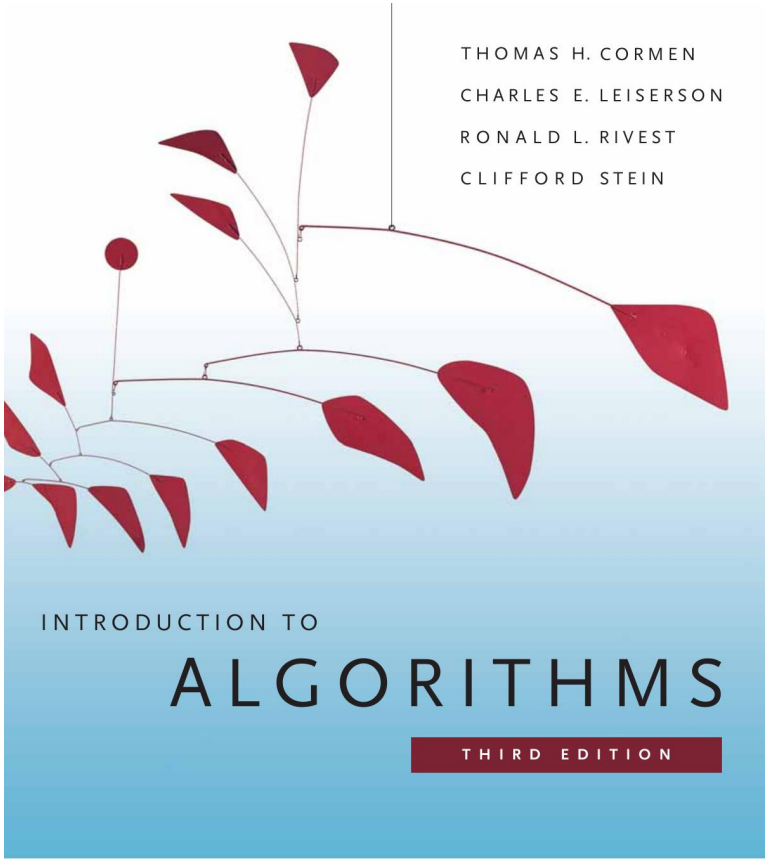
UFABC

- Implementações *type-safe* (com diversos graus de *safety* 😛) de árvores rubro-negras abundam na Internet.
  - Inclusive em linguagens como Java.
- Dentre as implementações em linguagens funcionais, as seguintes características de implementação comuns à que mostramos aqui estão presentes (em maior ou menor grau) em muitas delas:
  - Uso de `GADTs` e `DataKinds` para garantir as cores.
  - Números de Peano para assegurar que a altura negra está consistente.
  - Divisão entre inserções vermelhas e inserções pretas.
- Até onde pude averiguar, contudo, o uso de um tipo `Violation` como o que fizemos aqui para melhor organização e entendimento do código é minha jaboticaba.

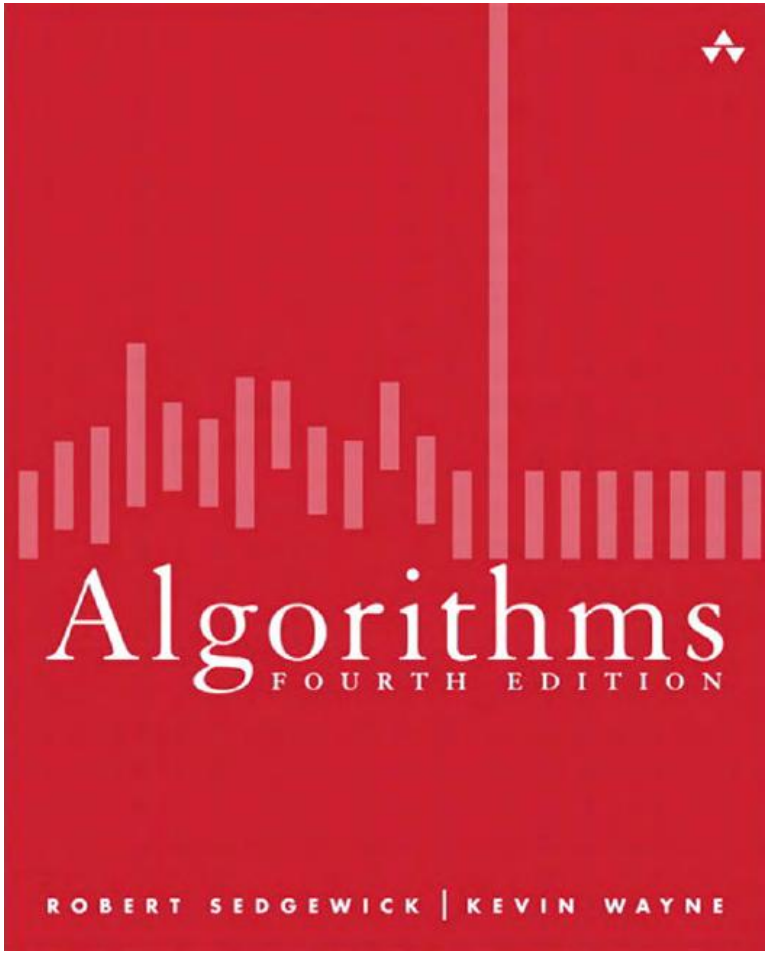
Referências



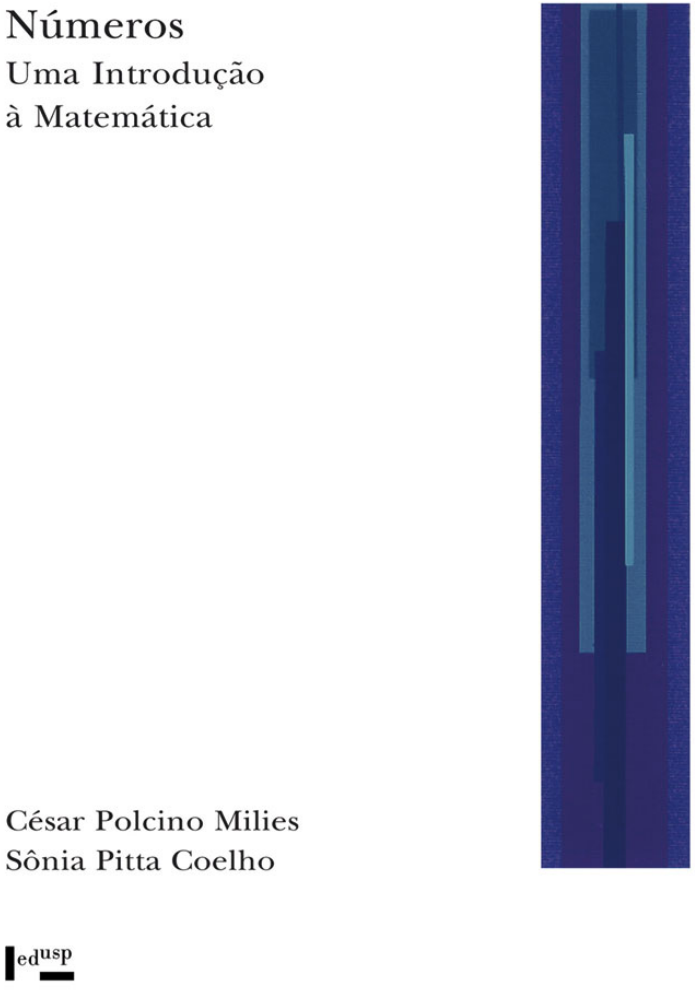
- [CO]
- Purely Functional Data Structures
  - Por *Chris Okasaki*



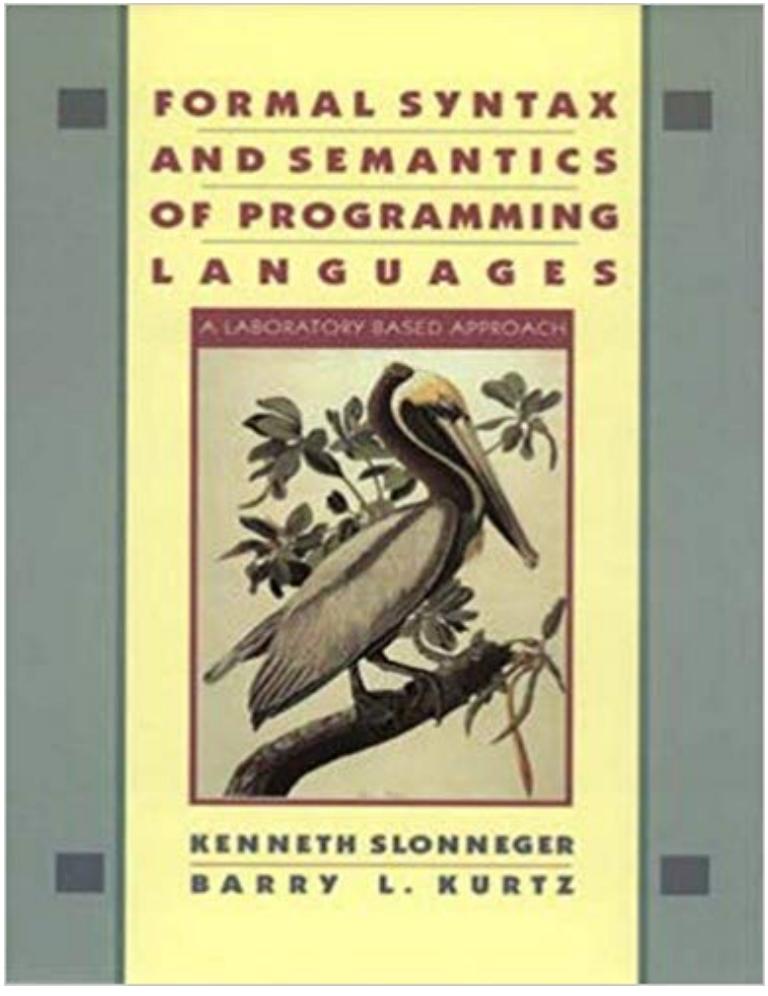
- [CLRS]
- Introduction to Algorithms
  - Por *Thomas H. Cormen & Charles E. Leiserson & Ronald L. Rives/t & /Clifford Stein*



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



- [MC]
- Números - Uma introdução à Matemática
  - Por *César Polcino Milies & Sônia Pitta Coelho*



- [SK]
- Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach
  - Por *Kenneth Slonneger & Barry L. Kurtz*
- Disponível gratuitamente aqui: `http://homepage.divms.uiowa.edu/~slonnegr/plf/Book/`

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.

UFABC

- Okasaki, Chris. "Red-black trees in a functional setting." Journal of functional programming 9.4 (1999): 471-477.
- Germane, Kimball, and Matthew Might. "Deletion: The curse of the red-black tree." Journal of Functional Programming 24.4 (2014): 423-433.
- Kahrs, Stefan. "Type-Safe Red-Black Trees with Java Generics." Specification Transformation Navigation (2009): 168.

- Huet, Gérard. ”The zipper.” Journal of functional programming 7.5 (1997): 549-554.
- Meertens, Lambert. ”First steps towards the theory of rose trees.”CWI, Amsterdam (1988).
- Eisenberg, Richard A. ”Dependent types in haskell: Theory and practice.”. PhD Thesis, University of Pensylvania 2016.