Programação Funcional
Aula 15 — Árvores de pesquisa

Pedro Vasconcelos
DCC/FCUP

2021

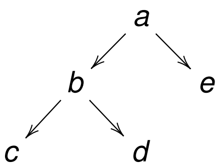
Árvores binárias

Um árvore binária é um grafo dirigido, conexo e acíclico em que cada vértice é de um de dois tipos:

nó: grau de saída 2 e grau de entrada 1 ou 0;

folha: grau de saída 0 e grau de entrada 1.

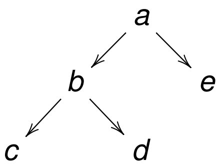
a **e** *b* **são nós;** *c*, *d* **e** *e* **são folhas.**



Árvores binárias (cont.)

Numa árvore binária existe sempre um único nó, que se designa *raiz*, com grau de entrada 0.

Exemplo: a raiz é o nó *a*



Representação recursiva

Partindo da raiz podemos decompor uma árvore binária de forma recursiva.

Uma árvore é:

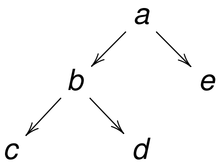
- um *nó* com duas sub-árvores; ou
- uma *folha*.

Traduzindo num tipo recursivo em Haskell:

```
data Arv = No Arv Arv -- sub-árvores esquerda e direita
         | Folha
```

Representação recursiva (cont.)

Exemplo anterior:



$$\underbrace{\texttt{No}\ \underbrace{(\texttt{No}\ \overbrace{\texttt{Folha}}^{c}\ \overbrace{\texttt{Folha}}^{d})}_{b}\ \overbrace{\texttt{Folha}}^{e}}_{a}$$

Anotações

Podemos associar informação à árvore colocando anotações nos nós, nas folhas ou em ambos.

Alguns exemplos:

```
-- anotar nós com inteiros
data Arv = No Int Arv Arv
         | Folha

-- anotar folhas com inteiros
data Arv = No Arv Arv
         | Folha Int

-- anotar nós com inteiros e folhas com boleanos
data Arv = No Int Arv Arv
         | Folha Bool
```

Anotações (cont.)

Em vez de usar tipos concretos, podemos parametrizar o tipo de árvore com os tipos de anotações.

Exemplos:

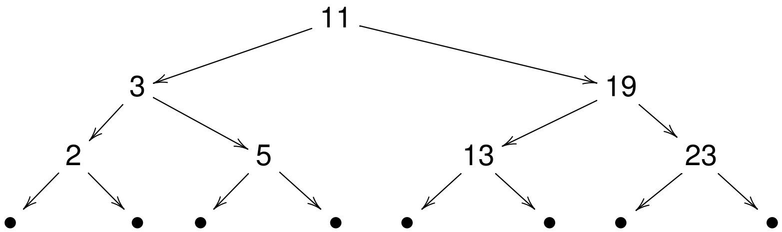
```
-- nós anotados com 'a'
data Arv a = No a (Arv a) (Arv a)
           | Folha

-- folhas anotadas com 'a'
data Arv a = No (Arv a) (Arv a)
           | Folha a

-- nós anotados com 'a' e folhas com 'b'
data Arv a b = No a (Arv a b) (Arv a b)
             | Folha b
```

Árvores de pesquisa

Uma árvore binária diz-se ordenada (ou de pesquisa) se o valor em todos os nós for:

- maior do que valores na sub-árvore esquerda;
- menor do que os valores na sub-árvore direita.

Exemplo:



Árvores de pesquisa (cont.)

Vamos representar árvores de pesquisa por um tipo recursivo parametrizado pelo tipo dos valores guardados nos nós.

```
data Arv a = No a (Arv a) (Arv a) -- nó
           | Vazia                -- folha
```

As folhas representam árvores vazias, pelo que não têm anotações.

Listar todos os valores

Para listar todos os valores de uma árvore por *ordem infixa*:

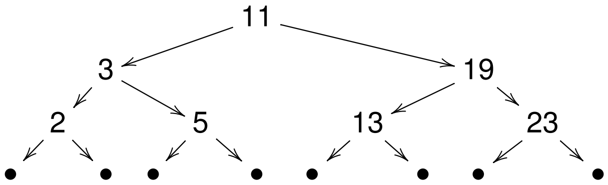
- listamos a sub-árvore esquerda;
- listamos o valor do nó;
- listamos a sub-árvore direita.

O caso base da recursão é a árvore vazia.

```
listar :: Arv a -> [a]
listar Vazia = []
listar (No x esq dir) = listar esq ++ [x] ++ listar dir
```

Listar todos os valores (cont.)

Exemplo:




```
listar
  (No 11
   (No 3 (No 2 Vazia Vazia) (No 5 Vazia Vazia))
   (No 19 (No 13 Vazia Vazia) (No 23 Vazia Vazia)))
==
 [2, 3, 5, 11, 13, 19, 23]
```

Listar todos os valores (cont.)

- Se a árvore estiver ordenada, então *listar* produz valores por ordem ascendente
- Podemos usar este facto para escrever uma função que testa se a árvore está ordenada

```
ordenada :: Ord a => Arv a -> Bool
ordenada arv = ascendente (listar arv)
  where
  ascendente []         = True
  ascendente [_]        = True
  ascendente (x:y:xs) = x<y && ascendente (y:xs)
```

Procurar um valor

Para procurar um valor numa árvore ordenada:

- comparamos com o valor do nó;
- recursivamente procuramos na sub-árvore esquerda ou direita.

```
procurar :: Ord a => a -> Arv a -> Bool
procurar x Vazia = False                    -- não ocorre
procurar x (No y esq dir)
    | x==y = True                           -- encontrou
    | x<y  = procurar x esq         -- procura à esquerda
    | x>y  = procura x dir           -- procura à direita
```

A restrição de classe "`Ord a =>`" indica que necessitamos de operações de comparação entre valores.

Inserir um valor

Também podemos inserir um valor numa árvore recursivamente, usando o valor em cada nó para optar por uma sub-árvore.

```
inserir :: Ord a => a -> Arv a -> Arv a
inserir x Vazia = No x Vazia Vazia
inserir x (No y esq dir)
    | x==y = No y esq dir           -- já ocorre; não insere
    | x<y  = No y (inserir x esq) dir -- insere à esquerda
    | x>y  = No y esq (inserir x dir) -- insere à direita
```

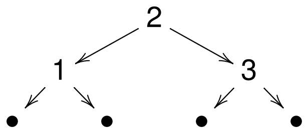
Construir a partir duma lista

Podemos inserir valores numa lista um-a-um recursivamente a partir da árvore vazia.

```
construir :: Ord a => [a] -> Tree a
construir [] = Vazia
construir (x:xs) = insert x (construir xs)
```

Exemplo:

```
construir [3,1,2]
== No 2 (No 1 Vazia Vazia) (No 3 Vazia Vazia)
```



Construir a partir duma lista (cont.)

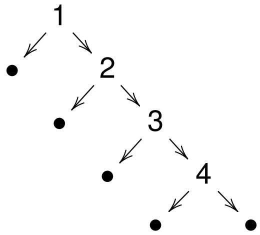
Alternativa: podemos usar *foldr* em vez da recusão.

```
construir xs =  foldr inserir Vazia xs
```

Construir a partir duma lista (cont.)

- A inserção garante que a árvore fica ordenada
- Mas podemos obter uma árvore desequilibrada

```
construir [4,3,2,1] ==
No 1 Vazia (No 2 Vazia (No 3 Vazia (No 4 Vazia Vazia)))
```



Construir árvores equilibradas

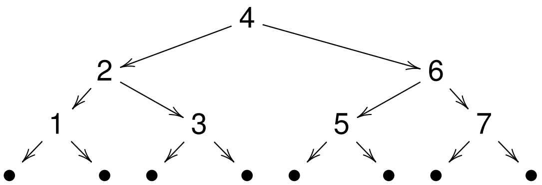
Partindo de uma lista ordenada, podemos construir uma árvore equilibrada usando partições sucessivas.

```
-- Construir uma árvore equilibrada
-- pré-condição: a lista de valores deve estar
-- por ordem crescente
construir :: [a] -> Arv a
construir [] = Vazia
construir xs = No x (construir xs') (construir xs'')
    where n = length xs`div`2                -- ponto médio
          xs' = take n xs                 -- partir a lista
          x:xs'' = drop n xs
```

Construir árvores equilibradas (cont.)

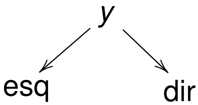
Exemplo:

```
 construir [1,2,3,4,5,6,7]
==
 No 4 (No 2 (No 1 Vazia Vazia) (No 3 Vazia Vazia))
      (No 6 (No 5 Vazia Vazia) (No 7 Vazia Vazia))
```



Remover um valor

Para remover um valor x duma árvore não-vazia



começamos por procurar o nó correcto:

se $x < y$**:** procuramos em *esq*;

se $x > y$**:** procuramos em *dir*;

se $x = y$**:** encontramos o nó.

Se chegarmos à árvore vazia: o valor x não ocorre.

Remover um valor (cont.)

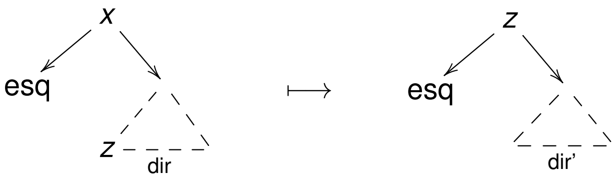
Podemos fácilmente remover um nó duma árvore com um só descendente não-vazio.

$$\begin{tikzcd}[column sep=small] & x \arrow[dl] \arrow[dr] & & & \\ \bullet & & \mathrm{dir} & \longmapsto & \mathrm{dir} \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] & x \arrow[dl] \arrow[dr] & & & \\ \mathrm{esq} & & \bullet & \longmapsto & \mathrm{esq} \end{tikzcd}$$

Remover um valor (cont.)

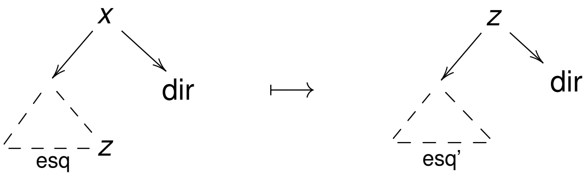
Se o nó tem dois descendentes não-vazios, então podemos substitui o seu valor pelo do *menor valor* na sub-árvore direita.



Note que temos ainda que remover z da sub-árvore direita.

Remover um valor (cont.)

Em alternativa, poderiamos usar o *maior valor* na sub-árvore esquerda.



Remover um valor (cont.)

Usamos uma função auxiliar para obter o o valor mais à esquerda duma árvore de pesquisa não vazia (isto é, o *menor valor*).

```
maisEsq :: Arv a -> a
maisEsq (No x Vazia _) = x
maisEsq (No _ esq _)   = maisEsq esq
```

Remover um valor (cont.)

Podemos agora definir a remoção considerando os diferentes casos.

```
remover :: Ord a => a -> Arv a -> Arv a
remover x Vazia  = Vazia                     -- não ocorre
remover x (No y Vazia dir)                 -- um descendente
     | x==y = dir
remover x (No y esq Vazia)                 -- um descendente
     | x==y = esq
remover x (No y esq dir)                -- dois descendentes
     | x<y  = No y (remover x esq) dir
     | x>y  = No y esq (remover x dir)
     | x==y = let z = maisEsq dir
              in No z esq (remover z dir)
```

Exercício

Escrever a definição alternativa

```
remover' :: Ord a => a -> Arv a -> Arv a
```

que usa o valor mais à direita da sub-árvore esquerda no caso dos dois descendentes não-vazios

Sugestão: escrever uma função auxiliar

```
maisDir :: Arv a -> a
```

que obtém o valor mais à direita de uma árvore não-vazia, i.e., o maior valor.