Árvores de pesquisa

Daniel Ventura
INF/UFG

2024/02

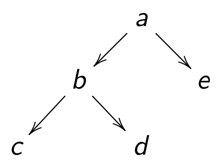
Árvores binárias I

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 entrada 1.

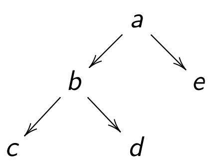
a e *b* são nós; *c*, *d* e *e* são folhas.



Árvores binárias II

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 I

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

Uma árvore é:

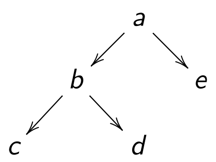
- um *nó* com duas sub-árvores; ou
- uma *folha*.

Traduzindo num tipo recursivo em Haskell:

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

Representação recursiva II

Exemplo anterior:



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

Anotações I

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

Alguns exemplos:

```
-- anotar cada nó com um inteiro
data Arv = No Int Arv Arv
         | Folha

-- anotar cada folhas com um inteiro
data Arv = No Arv Arv
         | Folha Int

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

Anotações II

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

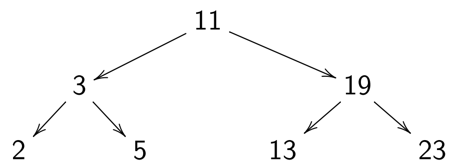
Exemplos:

```
-- 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 I

Uma árvore binária diz-se ordenada (ou de pesquisa) se o valor em cada nó for maior do que valores na sub-árvore esquerda e menor do que os valores na sub-árvore direita.

Exemplo:



Árvores de pesquisa II

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 são árvores vazias, que não têm anotações.

Listar todos os valores I

Podemos listar todos os valores de uma árvore de pesquisa listando recursivamente as sub-árvores esquerdas e direitas e colocando o valor do nó no meio.

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

Listar todos os valores II

Se a árvore estiver ordenada, então `listar` produz valores por ordem crescente; vamos usar este fato para testar se uma árvore está ordenada.

```
ordenada :: Ord a => Arv a -> Bool
ordenada arv = crescente (listar arv)
   where -- verificar se uma lista é crescente
       crescente xs = and (zipWith (<=) xs (tail xs))
```

Procurar um valor I

Para procurar um valor numa árvore ordenada, comparamos com o valor do nó e recursivamente procuramos na sub-árvore esquerda ou direita.

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

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

Inserir um valor I

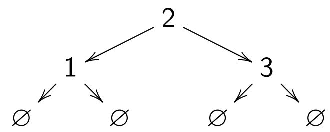
Também podemos inserir um valor numa árvore recursivamente, usando o valor em cada nó para optar por uma sub-árvore.

```
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
    | x<y  = No y (inserir x esq) dir -- insere à esquerda
    | x>y  = No y esq (inserir x dir ) -- insere à direita
```

Inserir múltiplos valores I

Podemos usar *foldr* para inserir uma lista de valores numa árvore. Em particular, começando com a árvore vazia, construimos uma árvore a partir de uma lista.

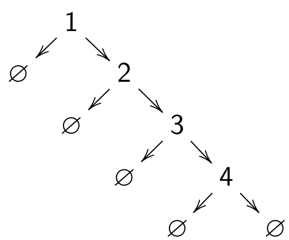
```
> foldr inserir Vazia [3,1,2]
No 2 (No 1 Vazia Vazia) (No 3 Vazia Vazia))
```



Inserir múltiplos valores II

A inserção garante a ordenação da árvore; contudo, dependendo dos valores, podemos obter árvores desequilibradas.

```
> foldr inserir Vazia [4,3,2,1]
No 1 Vazia (No 2 Vazia (No 3 Vazia (No 4 Vazia Vazia)))
```



Construir árvores equilibradas I

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

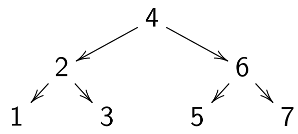
```
-- pré-condição: a lista deve estar por ordem crescente
construir :: [a] -> Arv a
construir [] = Vazia
construir xs = No x (construir xs') (construir xs'')
    where n = length xs`div`2              -- ponto médio
          xs' = take n xs              -- valores à esquerda
          x:xs'' = drop n xs     -- valores central e à direita
```

Construir árvores equilibradas II

Exemplo:

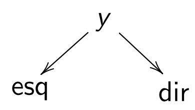
```
> construir [1,2,3,4,5,6,7]
No 4 (No 2 (No 1 Vazia Vazia) (No 3 Vazia Vazia))
     (No 6 (No 5 Vazia Vazia) (No 7 Vazia Vazia))
```

Diagrama (omitindo sub-árvores vazias):



Remover um valor I

Para remover um valor x duma árvore não-vazia



começamos por procurar o nó correto:

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

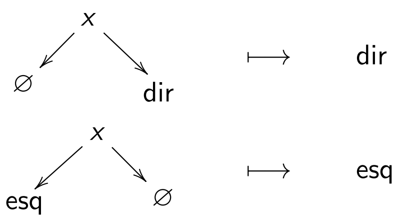
se $x > y$: procuramos em `dir`;

se $x = y$: encontramos o nó.

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

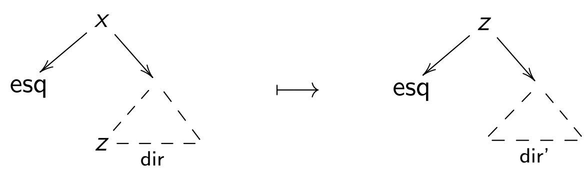
Remover um valor II

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



Remover um valor III

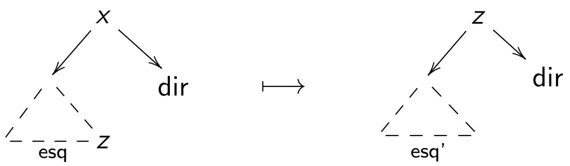
Se o nó tem dois descendentes não-vazios, então podemos substitui o seu valor pelo do *menor valor* na sub-árvore direita.



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

Remover um valor IV

Alternativamente, podemos usar o *maior valor* na sub-árvore esquerda.



Remover um valor V

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

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

Exercício: escrever uma função análoga

```
mais_dir :: Arv a -> a
```

que obtém o valor mais à direita na árvore, (i.e., o maior valor).

Remover um valor VI

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 = mais_esq dir
             in No z esq (remover z dir)
```

Remover um valor VII

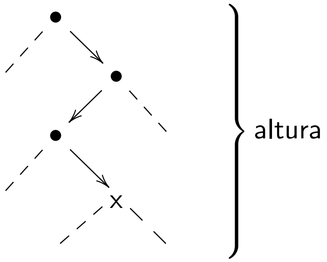
Exercício: escrever a definição alternativa

```
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.

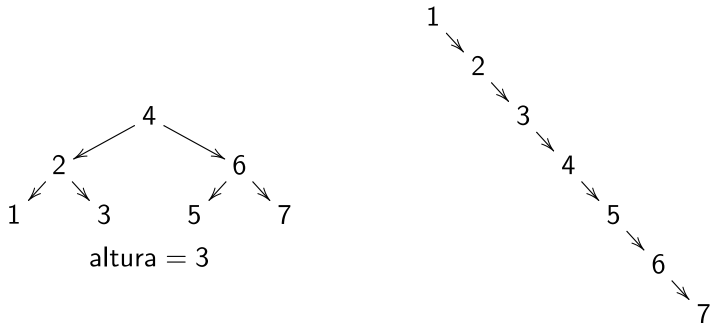
Complexidade I

Para procurar um valor numa árvore de pesquisa percorreremos um *caminho* da raiz até um nó intermédio, cujo comprimento é limitado pela altura da árvore.



Complexidade II

Para um mesmo conjunto de valores, árvores com *menor altura* (ou seja, *mais equilibradas*) permitem pesquisas mais rápidas.



altura = 3

altura = 7

Árvores equilibradas I

Uma árvore diz-se equilibrada (ou balanceada) se em cada nó a altura das sub-árvores difere no máximo de 1.

Vamos escrever uma função para testar se uma árvore é equilibrada. Começamos por definir a altura por recursão sobre a árvore:

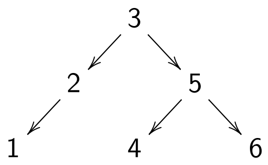
```
altura :: Arv a -> Int
altura Vazia = 0
altura (No _ esq dir) = 1 + max (altura esq) (altura dir)
```

Árvores equilibradas II

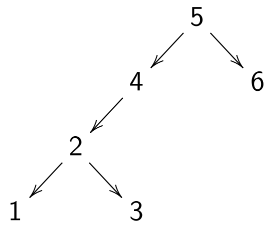
A condição de equilíbrio é também definida por recursão.

```
equilibrada :: Arv a -> Bool
equilibrada Vazia = True
equilibrada (No _ esq dir)
     = abs (altura esq - altura dir)<=1 &&
       equilibrada esq &&
       equilibrada dir
```

Exemplos



Árvore equilibrada



Árvore desequilibrada

Observações

- As árvores equilibradas permitem pesquisa mais eficiente: $O(\log n)$ operações para uma árvore com n valores
- O método de partição constroi árvores garantidamente equilibradas a partir de uma lista ordenada
- A inserção ou remoção de valores mantêm a árvore ordenada mas *podem não manter o equilíbrio*
- Na próxima aula: vamos ver *árvores AVL* que mantêm as duas condições de *ordenação* e *equilíbrio*.