Estudo de caso - Árvores binárias

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```haskell
module Aula09 where

main :: IO ()
main = return ()
```

Objetivos

- Implementar algoritmos clássicos sobre árvores binárias de busca.
- Implementar funções de ordem superior sobre árvores binárias de busca.

Árvores Binárias

- Definição do tipo de dados de árvores binárias.

```
data Tree a
  = Leaf
  | Node a (Tree a) (Tree a)
  deriving (Eq, Ord, Show)
```

Árvores Binárias

- Exemplo de uma árvore.

```haskell
ex1 :: Tree Int
ex1 = Node 2 (Node 1 Leaf Leaf)
             (Node 3 Leaf Leaf)
```

Busca

- Primeira funcionalidade: Buscar um elemento em uma árvore de busca.

1. Definir o tipo da função

```
memberTree :: Ord a => a -> Tree a -> Bool
```

Busca

2. Enumerar os casos

```
memberTree :: Ord a => a -> Tree a -> Bool
memberTree _ Leaf = _
memberTree x (Node y l r) = _
```

Busca

3. Implementar os casos base.

```
memberTree :: Ord a => a -> Tree a -> Bool
memberTree _ Leaf = False
memberTree x (Node y l r) = _
```

Busca

4. Implementar os casos recursivos.

```
memberTree :: Ord a => a -> Tree a -> Bool
memberTree _ Leaf = False
memberTree x (Node y l r)
      = case compare x y of
          LT -> memberTree x l
          GT -> memberTree x r
          _  -> True
```

Inserção

- Inserção de árvores binárias.

1. Definir o tipo da função

```
insertTree :: Ord a => a -> Tree a -> Tree a
```

Inserção

- Inserção de árvores binárias.

2. Enumerar os casos

```
insertTree :: Ord a => a -> Tree a -> Tree a
insertTree x Leaf = _
insertTree x (Node y l r) = _
```

Inserção

3. Definir o caso base.

```
insertTree :: Ord a => a -> Tree a -> Tree a
insertTree x Leaf = Node x Leaf Leaf
insertTree x (Node y l r) = _
```

Inserção

4. Definir os casos recursivos

```
insertTree :: Ord a => a -> Tree a -> Tree a
insertTree x Leaf = Node x Leaf Leaf
insertTree x (Node y l r)
  = case compare x y of
      LT -> Node y (insertTree x l) r
      GT -> Node y l (insertTree x r)
      _  -> Node y l r
```

Conversão em listas

- Convertendo em uma lista ordenada.
- Passo 1. Definindo o tipo.

```
toList :: Tree a -> [a]
```

Conversão em listas

- Convertendo em uma lista ordenada.
- Passo 2. Enumerando os casos

```
toList :: Tree a -> [a]
toList Leaf = _
toList (Node x l r) = _
```

Conversão em listas

- Convertendo em uma lista ordenada.
- Passo 3. Definindo o caso base.

```
toList :: Tree a -> [a]
toList Leaf = []
toList (Node x l r) = _
```

Conversão em listas

- Convertendo em uma lista ordenada.
- Passo 4. Definindo o caso recursivo.

```
toList :: Tree a -> [a]
toList Leaf = []
toList (Node x l r) = toList l ++ [x] ++ toList r
```

Conversão em listas

- Ineficiente... chamadas excessivas à concatenação de listas.
- Idéia melhor: uso de um acumulador.

Conversão em listas

- Definindo os casos.

```
toList :: Tree a -> [a]
toList t = toList' t []
   where
     toList' Leaf ac = _
     toList' (Node x l r) ac = _
```

Conversão em listas

- Definindo o caso base: retornar o acumulador.

```
toList :: Tree a -> [a]
toList t = toList' t []
   where
     toList' Leaf ac = ac
     toList' (Node x l r) ac = _
```

Conversão em listas

- Definindo o caso recursivo.
- Acumulador da árvore esquerda, deve possuir o resultado da árvore direita e o valor do nó atual.

Conversão em listas

- Solução:

```
toList :: Tree a -> [a]
toList t = toList' t []
   where
     toList' Leaf ac = ac
     toList' (Node x l r) ac
        = toList' l (x : toList' r ac)
```

Convertendo para árvores

- Simples:

```
fromList :: Ord a => [a] -> Tree a
fromList = foldr insertTree Leaf
```

Sort for free!

- Conversão de listas/árvores fornece um algoritmo de ordenação.

```
treeSort :: Ord a => [a] -> [a]
treeSort = toList . fromList
```

Remoção

- Mostramos como inserir e procurar elementos em uma árvore.
- Porém, como remover um elemento preservando os invariantes da árvore?

Remoção

- Passo 1. Definir o tipo

```
remove :: Ord a => a -> Tree a -> Tree a
```

Remoção

- Passo 2. Enumerar os casos

```
remove :: Ord a => a -> Tree a -> Tree a
remove _ Leaf = _
remove v (Node x l r) = _
```

Remoção

- Passo 3. Definir o caso base.

```
remove :: Ord a => a -> Tree a -> Tree a
remove _ Leaf = Leaf
remove v (Node x l r) = _
```

Remoção

- Passo 4. Definir o caso recursivo.

```
remove :: Ord a => a -> Tree a -> Tree a
remove _ Leaf = Leaf
remove v (Node x l r)
   | v == x = _
   | v < x = Node x (remove v l) r
   | otherwise = Node x l (remove v r)
```

Remoção

- Só podemos remover valores em uma "folha", i.e., sem subárvores.
- Mas como remover um valor presente em um nó interno?

Remoção

- Removendo valor de um nó interno.
  - Substituir um do nó interno por um que preserva os invariantes da árvore.
- Qual valor preserva o invariante?
  - Possibilidades: Maior valor da subárvore esquerda ou menor da subárvore direita.

Remoção

- Obtendo menor valor e removendo da árvore.
- Passo 1. Definindo o tipo.

```
removeMin :: Ord a => Tree a -> Maybe (a,Tree a)
```

Remoção

- Obtendo menor valor e removendo da árvore.
- Passo 2. Enumerando os casos.

```
removeMin :: Ord a => Tree a -> Maybe (a,Tree a)
removeMin Leaf = _
removeMin (Node x l r) = _
```

Remoção

- Obtendo menor valor e removendo da árvore.
- Passo 3. Definindo o caso base.

```
removeMin :: Ord a => Tree a -> Maybe (a,Tree a)
removeMin Leaf = Nothing
removeMin (Node x l r) = _
```

Remoção

- Obtendo menor valor e o removendo da árvore.
- Passo 4. Definindo o caso recursivo.

```
removeMin :: Ord a => Tree a -> Maybe (a,Tree a)
removeMin Leaf = Nothing
removeMin (Node x Leaf r) = Just (x, r)
removeMin (Node x l r)
   = case removeMin l of
        Nothing -> Nothing
        Just (y,l') -> Just (y, Node x l' r)
```

Remoção

- Removendo de um nó interno.
  - Se o nó não possui uma das subárvores, o resultado é a outra árvore.
  - Caso contrário, devemos substituir o valor atual por um elemento já presente na árvore.

Remoção

- Implementarem essa lógica em outra função

```haskell
removeEq :: Ord a => Tree a -> Tree a -> Tree a
removeEq Leaf r = r
removeEq l Leaf = l
removeEq l r
    = case removeMin l of
        Nothing -> error "Impossible!"
        Just (x,l') -> Node x l' r
```

Remoção

- Implementação final.

```
remove :: Ord a => a -> Tree a -> Tree a
remove _ Leaf = Leaf
remove v (Node x l r)
  = case compare v x of
      EQ -> removeEq l r
      LT -> Node x (remove v l) r
      GT -> Node x l (remove v r)
```

Função map

1. Definir o tipo da função

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

Função map

2. Enumerar os casos

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Leaf = _
mapTree f (Node x l r) = _
```

Função map

3. Definir o caso base

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Leaf = Leaf
mapTree f (Node x l r) = _
```

Função map

4. Definir o caso recursivo

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Leaf = Leaf
mapTree f (Node x l r)
      = Node (f x) (mapTree f l)
                   (mapTree f r)
```

Função fold

1. Definir o tipo da função

```
foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b
```

- Porquê uma função de 3 parâmetros?
  - 1o parâmetro: elemento da árvore.
  - 2o parâmetro: acumulador da sub-árvore esquerda.
  - 3o parâmetro: acumulador da sub-árvore direita.

Função fold

2. Enumerar os casos

```
foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b
foldTree _ v Leaf = _
foldTree f v (Node x l r) = _
```

Função fold

3. Definir o caso base.

```
foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b
foldTree _ v Leaf = v
foldTree f v (Node x l r) = _
```

Função fold

4. Definir o caso recursivo.

```
foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b
foldTree _ v Leaf = v
foldTree f v (Node x l r)
     =  f x (foldTree f v l)
            (foldTree f v r)
```

Altura

- Definindo a altura em termos de foldTree.

```
height :: Tree a -> Int
height = foldTree (\ _ acl acr -> 1 + max acl acr) 0
```

Exercícios

1. Implemente a função mapTree usando foldTree.
2. Implemente uma função que calcula o número de elementos presente em uma árvore usando foldTree.