Árvores binárias

As árvores binárias são estruturas de dados muito úteis para organizar a informação.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
   deriving (Show)
```

Os construtores da árvores são:

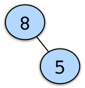
```
Empty :: BTree a

Node :: a -> (BTree a) -> (BTree a) -> (BTree a)
```

`Empty` representa a árvore vazia.

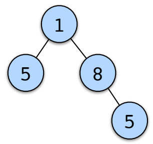
`Node` recebe um elemento e duas árvores, e constrói a árvore com esse elemento na raiz, uma árvore do lado esquerdo e outra do lado direito.

```
arv1 = Node 5 Empty Empty
```


```
arv2 = Node 8 Empty arv1
```

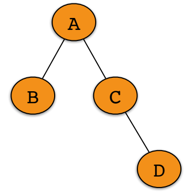



```
arv3 = Node 1 arv1 arv2
```



Árvores binárias

Terminologia



- O nodo A é a **raiz** da árvore.
- Os nodos B e C são **filhos** (ou **descendentes**) de A.
- O nodo C é **pai** de D.
- B e C são **folhas** da árvore.
- O **caminho** (*path*) de um nodo é a sequência de nodos da raiz até esse nodo. Por exemplo, A,C,D é o caminho para o modo D.
- A **altura** da árvore é o comprimento do caminho mais longo. Esta árvore tem altura 3.

Árvores binárias

As funções definidas sobre tipos de dados recursivos, são geralmente funções recursivas, com padrões de recursividade semelhantes aos dos tipos de dados.

Exemplo: Calcular o número de nodos que tem uma árvore.

```
conta :: BTree a -> Int
conta Empty = 0
conta (Node x e d) = 1 + conta e + conta d
```

Exemplo: Somar todos de nodos de uma árvore de números .

```
sumBT :: Num a =>  BTree a -> a
sumBT Empty = 0
sumBT (Node x e d) = x + sumBT e + sumBT d
```

```
> sumBT (Node 2 Empty (Node 7 Empty Empty))
          = 2 + (sumBT Empty) + sumBT (Node 7 Empty Empty)
          = 2 + 0 + (7 + sumBT Empty + sumBT Empty)
          = 2 + 0 + (7 + 0 + 0)
          = 9
```

Árvores binárias

Exemplo: Calcular a altura de uma árvore.

```
altura :: BTree a -> Int
altura Empty = 0
altura (Node _ e d) = 1 + max (altura e) (altura d)
```

Exemplos: As funções map e zip para árvores binárias.

```
mapBT :: (a -> b) -> BTree a -> BTree b
mapBT f Empty = Empty
mapBT f (Node x e d) = Node (f x) (mapBT f e) (mapBT f d)
```

```
zipBT :: BTree a -> BTree b -> BTree (a,b)
zipBT (Node x1 e1 d1) (Node x2 e2 d2) =
                         Node (x1,x2) (zipBT e1 e2) (zipBT d1 d2)
zipBT _ _ = Empty
```

Travessias de árvores binárias

Uma árvore pode ser percorrida de várias formas. As principais estratégias são:

Travessia preorder: visitar a raiz, depois a árvore esquerda e a seguir a árvore direita.

```
preorder :: BTree a -> [a]
preorder Empty = []
preorder (Node x e d) = [x] ++ (preorder e) ++ (preorder d)
```

Travessia inorder: visitar árvore esquerda, depois a raiz e a seguir a árvore direita.

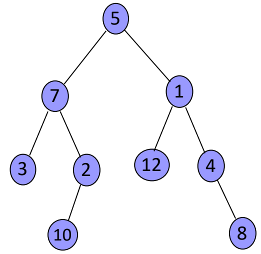
```
inorder :: BTree a -> [a]
inorder Empty = []
inorder (Node x e d) = (inorder e) ++ [x] ++ (inorder d)
```

Travessia postorder: visitar árvore esquerda, depois árvore direita, e a seguir a raiz..

```
postorder :: BTree a -> [a]
postorder Empty = []
postorder (Node x e d) = (postorder e) ++ (postorder d) ++ [x]
```

Travessias de árvores binárias

```
arv = (Node 5 (Node 7 (Node 3 Empty Empty)
                      (Node 2 (Node 10 Empty Empty) Empty)
              )
              (Node 1 (Node 12 Empty Empty)
                      (Node 4 Empty (Node 8 Empty Empty))
              )
      )
```




```
preorder arv  = [5,7,3,2,10,1,12,4,8]

inorder arv   = [3,7,10,2,5,12,1,4,8]

postorder arv = [3,10,2,7,12,8,4,1,5]
```

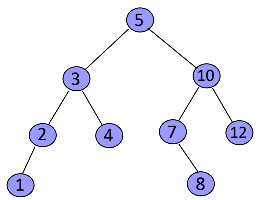
Árvores binárias de procura

Uma árvore binária em que o valor de cada nodo é maior do que os nodos à sua esquerda, e meor do que os nodos à sua direita diz-se uma árvore binária de procura (ou de pesquisa)

Uma **árvore binária de procura** é uma árvore binária que verifica as seguinte condição:

- a raiz da árvore é maior do que todos os elementos que estão na sub-árvore esquerda;
- a raiz da árvore é menor do que todos os elementos que estão na sub-árvore direita;
- ambas as sub-árvores são árvores binárias de procura.

Exemplo: Esta é uma árvore binária de procura de procura



Árvores binárias de procura

Exemplo: Testar se um elemento pertence a uma árvore.

```
elemBT :: Ord a => a -> BTree a -> Bool
elemBT x Empty = False
elemBT x (Node y e d)
           | x < y  = elemBT x e
           | x > y  = elemBT x d
           | x == y = True
```

Exemplo: Inserir um elemento numa árvores binária de procura.

```
insertBT :: Ord a => a -> BTree a -> BTree a
insertBT x Empty = Node x Empty Empty
insertBT x (Node y e d)
           | x < y  = Node y (insertBT x e) d
           | x > y  = Node y e (insertBT x d)
           | x == y = Node y e d
```

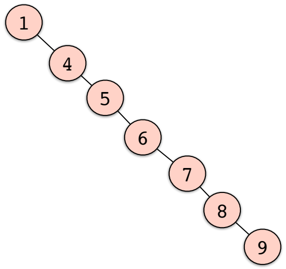
Árvores binárias de procura

O formato de uma árvore depende da ordem pela qual os elementos vão sendo inseridos.

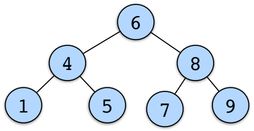
Exemplo: Considere a seguinte função que converte uma lista numa árvore.

```
listToBT :: Ord a => [a] -> BTree a
listToBT l = foldl (flip insertBT) Empty l
```

```
listToBT [1,4,5,6,7,8,9] =
```



```
listToBT [6,4,1,8,9,5,7] =
```

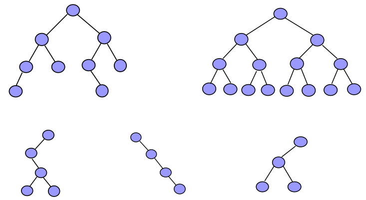


Árvores balanceadas

Uma árvore binária diz-se **balanceada** (ou **equilibrada**) se é vazia, ou se verifica as seguintes condições:

- as alturas da sub-árvores esquerda e direita diferem no máximo em uma unidade;
- ambas as sub-árvores são balanceadas.

Exemplos:



Balanceadas.

Desbalanceadas.

Exercício: Defina uma função que testa se uma árvore é balanceada.

Árvores binárias de procura

As árvores binárias de procura possibilitam pesquisas potencialmente mais eficientes do que as listas.

Exemplo:

Pesquisa numa lista não ordenada.

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup x [] = Nothing
lookup x ((y,z):t) | x == y = Just z
                   | x /= y = lookup x t
```

Pesquisa numa lista ordenada.

```
lookupSL :: Ord a => a -> [(a,b)] -> Maybe b
lookupSL x [] = Nothing
lookupSL x ((y,z):t) | x < y  = Nothing
                     | x == y = Just z
                     | x > y  = lookupSL x t
```

O número de comparações de chaves é **no máximo igual ao comprimento da lista.**

Pesquisa numa árvore binária de procura.

```
lookupBT :: Ord a => a -> BTree (a,b) -> Maybe b
lookupBT x Empty = Nothing
lookupBT x (Node (y,z) e d) | x < y  = lookupBT x e
                            | x > y  = lookupBT x d
                            | x == y = Just z
```

O número de comparações de chaves é **no máximo igual à altura da árvore.**

Árvores binárias de procura

A pesquisa em árvores binárias de procura são especialmente mais eficientes se as árvores forem balanceadas.

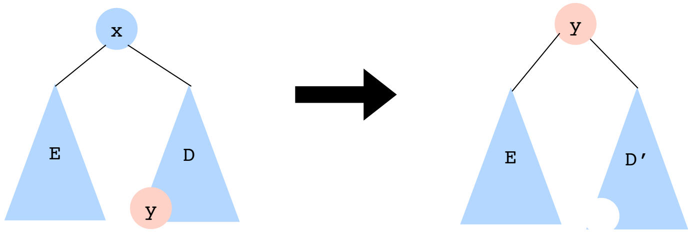
Uma forma de balancear uma árvore de procura consiste em: primeiro gerar uma lista ordenada com os seus elementos e depois, a partir dessa lista, gerar a árvore.

```
balance :: BTree a -> BTree a
balance t = constroi (inorder t)

constroi :: [a] -> BTree a
constroi [] = Empty
constroi l = let n = length l
                 (l1, x:l2) = splitAt (n `div` 2) l
             in Node x (constroi l1) (constroi l2)
```

Exercício: Defina uma versão mais eficiente desta função que não esteja a calcular sempre o comprimento da lista.

Árvores binárias de procura

A remoção do elemento que está na raiz de uma árvore de procura pode ser feita indo buscar o menor elemento da sub-árvore direita (ou, em alternativa, o maior elemento da sub-árvore esquerda) para tomar o seu lugar.



Exercício: Com base nesta ideia, defina uma função que remove um elemento uma árvore de procura. Comece por definir uma função que devolve um par com o mínimo de uma árvore não vazia e a árvore sem o mínimo.

Árvores irregulares *(rose trees)*

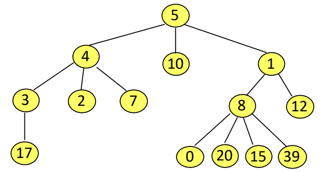
Nas **árvores irregulares** cada nodo pode ter um número variável de descendentes. O seguinte tipo de dados é uma implementação de árvores irregulares, não vazias.

```
data RTree a = R a [RTree a]
   deriving (Show)
```

O único construtor da árvores é `R :: a -> [RTree a] -> RTree a`

R recebe o elemento que fica na raiz da árvore e a lista das sub-árvores com que vai construir a árvore.

```
R 5 [ R 4 [ R 3 [R 17 []], R 2 [], R 7 []],
      R 10 [],
      R 1 [ R 8 [ R 0 [], R 20 [], R 15 [],  R 39 [] ],
            R 12 [] ]
    ]
```



Árvores irregulares *(rose trees)*

Como é de esperar, as funções definidas sobre rose trees seguem um padrão de recursividade compatível com sua definição indutiva.

Exemplo: Contar os nodos de uma árvore.

```
contaRT :: RTree a -> Int
contaRT (R x l) = 1 + sum (map contaRT l)
```

Exemplo: Calcular a altura de uma árvore.

```
alturaRT :: RTree a -> Int
alturaRT (R x []) = 1
alturaRT (R x l) = 1 + maximum (map alturaRT l)
```

Árvores irregulares *(rose trees)*

Exemplo: Testar se um elemento pertence a uma árvore.

```
pertenceRT :: Eq a => a -> RTree a -> Bool
pertenceRT x (R y l) = x==y || or (map (pertenceRT x) l)
```

(pertenceRT x) :: RTree a -> Bool

Exemplo: Fazer uma travessia *preorder* uma árvore.

```
preorderRT :: RTree a -> [a]
preorderRT (R x l) = x : concat (map preorderRT l)
```

Exercício: Defina uma função que converte uma árvore binária numa *rose tree*.