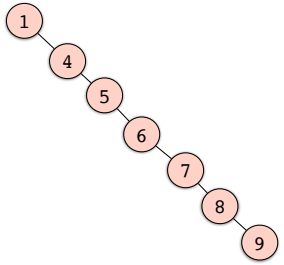
Árvores binárias de procura

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

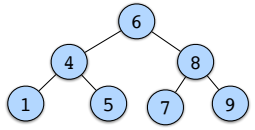
Exemplo: Considere a seguinte função que converte uma lista numa árvore, inserindo os elementos pela a ordem em que estes aparecem na lista.

```
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] =
```

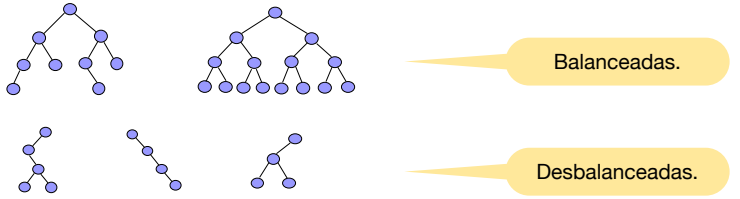


Árvores balanceadas

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

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

Exemplos:



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.

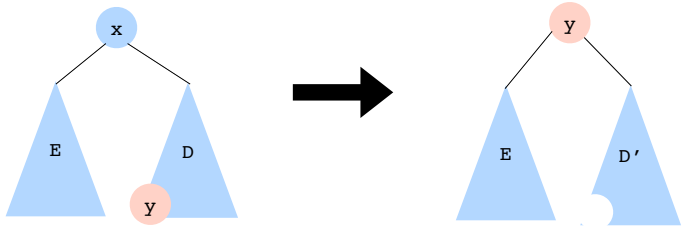
```
balance :: BTree a -> BTree a
balance t = constroi (inorder t)

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. (Sugestão: trabalhe com um par que tem a lista e o seu comprimento.)

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 de 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)*

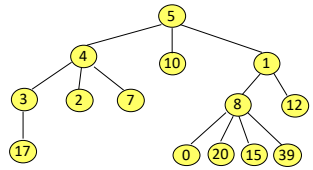
Nas **árvores irregulares** cada nodo pode ter um número variável de descendentes. O seguinte tipo de dados é uma implementação de árvores irregulares, não vazias.

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

O único construtor da árvore 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.*