Travessias de árvores binárias

Para converter uma árvore binária numa lista podemos usar diversas estratégias, como por exempo:

Preorder:	**R E D**	**R** – visitar a raiz
Inorder:	**E R D**	**E** – atravessar a sub-árvore esquerda
Postorder:	**E D R**	**D** – atravessar a sub-árvore direita

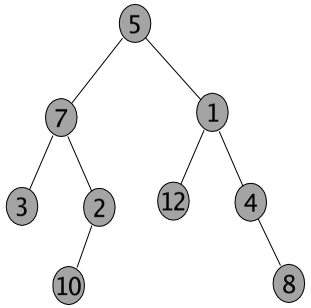
```
preorder :: ArvBin a -> [a]
preorder Vazia = []
preorder (Node x e d) = [x] ++ (preorder e) ++ (preorder d)
```

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

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



```
arv = (Nodo 5 (Nodo 7 (Nodo 3 Vazia Vazia)
                      (Nodo 2 (Nodo 10 Vazia Vazia) Vazia)
              )
              (Nodo 1 (Nodo 12 Vazia Vazia)
                      (Nodo 4 Vazia (Nodo 8 Vazia Vazia))
              )
      )
```




```
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 diz-se de **procura,** se é *vazia*, ou se verifica todas as seguintes condições:

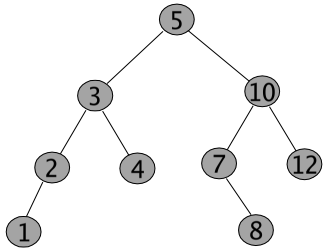
- a raiz da árvore é maior do que todos os elementos da sub-árvore esquerda;
- a raiz da árvore é menor do que todos os elementos da sub-árvore direita;
- ambas as sub-árvores são árvores binárias de procura.

Exemplo: Predicado para testar se uma dada árvore binária é de procura.

```
arvBinProcura Vazia = True
arvBinProcura (Nodo x e d) =
     (x > maximum (preorder e)) &&  (x < minimum (preorder d))
           && (arvBinProcura e) && (arvBinProcura d)
```

Exemplo: A árvore seguinte é uma árvore binária de procura.

Qual é o termo que a representa ?



Exemplo: Acrescentar um elemento à árvore binária de procura.

```
insABProc x Vazia = (Nodo x Vazia Vazia)
insABProc x (Nodo y e d )
          | x < y  = Nodo y (insABProc x e) d
          | y < x  = Nodo y e (insABProc x d)
          | x == y = Nodo y e d
```

Note que os elementos repetidos não estão a ser acrescentados à árvore de procura.

O que alteraria para, relaxando a noção de árvore binária de procura, aceitar elementos repetidos na árvore ?

Exercício: Qual é a função de travessia que aplicada a uma árvore binária de procura retorna uma lista ordenada com os elementos da árvore ?

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

Exercício: Desenhe as árvores resultantes das seguintes sequências de inserção numa árvore inicialmente vazia.

a) 7, 4, 9, 6, 1, 8, 5
b) 1, 4, 5, 6, 7, 8, 9
c) 6, 4, 1, 8, 9, 5, 7

Exercício: Defina uma função que recebe uma lista e constoi uma árvore binária de procura com os elementos da lista.

Árvores Balanceadas

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

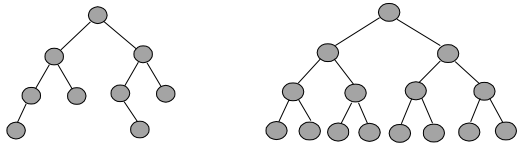
- as alturas da sub-árvores esquerda e direita diferem no máximo em uma unidade;
- ambas as sub-árvores são árvores balancedas.

Exemplo: Predicado para testar se uma dada árvore binária é balanceada.

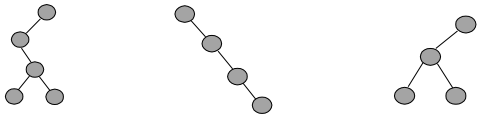
```
balanceada Vazia = True
balanceada (Nodo _ e d) = (abs ((altura e)-(altura d))) <= 1
                          && (balanceada e) && (balanceada d)
```

Exemplos:

Balanceadas:



Não balanceadas:



As árvores binárias de procura são estruturas de dados que possibilitam pesquisas potencialmente mais eficientes da informação, do que as pesquisas em listas.

Exemplo:

A tabela de associações BI – Nome, pode ser guardada numa árvore binária de procura com o tipo `ArvBin (BI,Nome)`.

A função de pesquisa nesta árvores binária de procura organizada por BI pode ser definida por

```
pesquisaABProc :: BI -> ArvBin (BI,Nome) -> Maybe Nome
pesquisaABProc n Vazia = Nothing
pesquisaABProc n (Nodo (x,y) e d)
                     | n == x = Just y
                     | n < x  = pesquisaABProc n e
                     | n > x  = pesquisaABProc n d
```


Chama-se **chave** ao componente de informação que é único para cada entidade. Por exemplo: o nº de BI é chave para cada cidadão; nº de aluno é chave para cada estudante universitário; nº de contribuinte é chave para cada empresa.

Uma medida da eficiência de uma pesquisa é a o número de comparações de chaves que são feitas até que se encontre o elemento a pesquisar. É claro que isso depende da posição da chave na estrutura de dados.

O número de comparações de chaves numa pesquisa:

- *numa lista*, é no máximo igual ao comprimento da lista;
- *numa árvore binária de procura*, é no máximo igual à altura da árvore.

Assim, a pesquisa em árvores binárias de procura são especialmente mais eficientes se as árvores forem balanceadas.

Porquê ?

Existem algoritmos de inserção que mantêm o equilibrio das árvores (mas não serão apresentados nesta disciplina).

Exemplo: A partir de uma lista ordenada por ordem crescente de chaves podemos construir uma árvore binária de procura balanceada, através da função

```
constroiArvBal [] = Vazia
constroiArvBal xs = Nodo x (constroiArvBal xs1) (constroiArvBal xs2)
   where
         k = (length xs) `div` 2
         xs1 = take k xs
         (x:xs2) = drop k xs
```

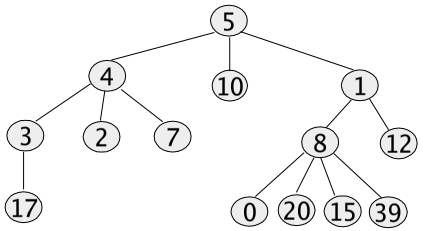
Exercícios:

- Defina uma função que dada uma árvore binária de procura, devolve o seu valor mínimo.
- Defina uma função que dada uma árvore binária de procura, devolve o seu valor máximo.
- Como poderá ser feita a remoção de um nodo de uma árvore binária de procura, de modo a que a árvore resultante continue a ser de procura ?
  Defina uma função que implemente a estratégia que indicou.

Outras Árvores

Árvores Irregulares
(finitely branching trees)

```
data Tree a = Node a [Tree a]
```



Esta árvore do tipo **(Tree Int)** é representada pelo termo:

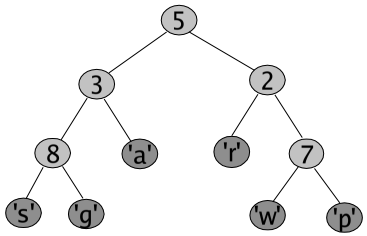
```
Node 5 [ Node 4 [ Node 3 [Node 17 []],
                  Node 2 [], Node 7 []
                ],
         Node 10 [],
         Node 1 [ Node 8 [ Node 0 [], Node 20 [],
                           Node 15 [],  Node 39 []
                         ],
                  Node 12 []
                ]
  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


Outras Árvores

Full Trees Árvores com *nós* (intermédios) do tipo a e *folhas* do tipo b.

```
data ABin a b = Folha b
              | No a (ABin a b) (ABin a b)
```

Esta árvore do tipo **(ABin Int Char)** é representada pelo termo:




```
(No 5 (No 3 (No 8 (Folha 's') (Folha 'g'))
            (Folha 'a')
      )
      (No 2 (Folha 'r')
            (No 7 (Folha 'w') (Folha 'p'))
      )
)
```


"Records"

Numa declaração de um tipo algébrico os construtores podem ser declarados associando a cada um dos seus parâmetros um nome (uma etiqueta).

Exemplo:

```
data PontoC = Pt {xx :: Float, yy :: Float, cor :: Cor}
```

desta forma, para além do construtor de dados

```
Pt :: Float -> Float -> Cor -> PontoC
```

também ficam definidos os nome dos *campos* xx, yy e cor, e 3 ***selectores*** com o mesmo nome:

```
xx  :: PontoC -> Float
yy  :: PontoC -> Float
cor :: PontoC -> Cor
```

Os valores do novo tipo PontoC podem ser construidos da forma usual, por aplicação do construtor aos seus argumentos.

```
p1 = (Pt 3.2 5.5 Azul)   :: PontoC
```

Além disso, o nome dos campos podem agora também ser usados na construção de valores do novo tipo.

```
p2 = Pt {xx=3.1, yy=8.0, cor=Vermelho}   :: PontoC
p3 = Pt {cor=Verde, yy=2.2, xx=7.1}      :: PontoC
```


"Records"

Note que
```
(Pt 3.2 5.5 Azul)
Pt {xx=3.2, yy=5.5, cor=Azul}
Pt {yy=5.5, cor=Azul, xx=3.2}
```
são exactamente o mesmo valor.

Aos tipos com um único construtor e com os campos etiquetados dá-se o nome de ***records***.

Os padrões podem também usar o nome dos campos (todos ou alguns, por qualquer ordem).

Exemplo: Três versões equivalentes da função que calcula a distância de um ponto à origem.

```
dist0 :: PontoC -> Float
dist0 p = sqrt ((xx p)^2 * (yy p)^2)
```

```
dist0' :: PontoC -> Float
dist0' Pt {xx=x, yy=y} = sqrt (x^2 * y^2)
```

```
dist0'' :: PontoC -> Float
dist0'' (Pt x y c) = sqrt (x^2 * y^2)
```