Haskell (IR3) – Arbres Binaires

Stéphane Vialette

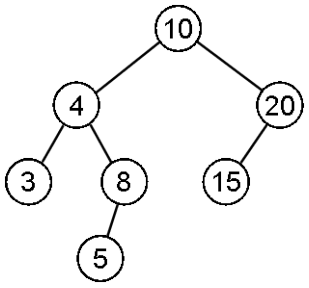
21 novembre 2018

Question 1: Arbres binaires sur le entiers

Considérons la variante suivante du type récursif des arbres binaires sur les entiers, défini par :

```
data BinIntTree = Empty | Branch BinIntTree Int BinIntTree
                  deriving (Show)
```

(a) Écrire une expression Haskell, de type `BinIntTree`, qui représente l'arbre suivant.



Dans la suite, nous supposerons que cet exemple est retourné par la fonction : `binIntTreeExample :: BinIntTree`.

Les fonctions haskell suivantes vous permettrons de visualiser plus facilement vos arbres :

```
indent :: [String] -> [String]
indent = map ("  " ++)

layoutTree :: BinIntTree -> [String]
layoutTree Empty = []
layoutTree (Branch left x right) = indent (layoutTree right) ++
                                   [show x]                  ++
                                   indent (layoutTree left)

prettyTree :: BinIntTree -> String
prettyTree = unlines . layoutTree
```

Nous pouvons les utiliser de la façon suivante :

```
*Main> putStrLn $ prettyTree binIntTreeExample
  20
    15
10
    8
      5
  4
    3

*Main>
```

Solution:
On peut définir l'arbre pas à pas dans `ghci` :

```
ghci: let bt3  = Branch Empty 3 Empty
ghci: let bt5  = Branch Empty 5 Empty
ghci: let bt8  = Branch bt5 8 Empty
ghci: let bt4  = Branch bt4 4 bt8
ghci: let bt15 = Branch Empty 15 Empty
ghci: let bt20 = Branch bt15 20 Empty
ghci: let bt10 = Branch bt4 10 bt20
ghci:
```

Le mieux est de définir cet example via une fonction pour tester rapidement les fonctions que vous devez écrire :

```
binIntTree = bt10
  where
    bt3  = Branch Empty 3 Empty
    bt5  = Branch Empty 5 Empty
    bt8  = Branch bt5 8 Empty
    bt4  = Branch bt3 4 bt8
    bt15 = Branch Empty 15 Empty
    bt20 = Branch bt15 20 Empty
    bt10 = Branch bt4 10 bt20
```

(b) Écrire la fonction
`emptyBinIntTree :: BinIntTree`
qui retourne un arbre binaire de recherche vide.

Solution:

```
emptyBinIntTree :: BinIntTree
emptyBinIntTree = Empty
```

(c) Écrire la fonction
`sizeBinIntTree :: Num a => BinIntTree -> a`
qui retourne le nombre de nœuds dans un arbre binaire.

Solution:

```
sizeBinIntTree :: Num a => BinIntTree -> a
sizeBinIntTree Empty = 0
sizeBinIntTree (Branch left _ right) = sizeLeft + 1 + sizeRight
```

```
  where
    sizeLeft  = sizeBinIntTree left
    sizeRight = sizeBinIntTree right
```

(d) Écrire la fonction

```
maxBinIntTree :: BinIntTree -> Int
```

qui retourne l'étiquette maximale. Dans un premier temps (et pour simplifier), nous supposerons que l'étiquette de l'arbre vide est `minBound :: Int`.

Solution:

```
maxBinIntTree :: BinIntTree -> Int
maxBinIntTree Empty = minBound
maxBinIntTree (Branch left x right) = maximum [maxLeft, x, maxRight]
  where
    maxLeft  = maxBinIntTree left
    maxRight = maxBinIntTree right
```

(e) Écrire la fonction

```
minBinIntTree :: BinIntTree -> Int
```

qui retourne l'étiquette minimale. Dans un premier temps (et pour simplifier), nous supposerons que l'étiquette de l'arbre vide est `maxBound :: Int`.

Solution:

```
minBinIntTree :: BinIntTree -> Int
minBinIntTree Empty = minBound
minBinIntTree (Branch left x right) = minimum [minLeft, x, minRight]
  where
    minLeft  = minBinIntTree left
    minRight = minBinIntTree right
```

(f) (⋆) Pouvez vous extraire la logique des fonctions `maxBinIntTree` et `minBinIntTree` pour factoriser le code commun ?

Solution:

```
applyBinIntTree :: ([Int] -> Int) -> Int -> BinIntTree -> Int
applyBinIntTree _ b Empty = n
applyBinIntTree f b (Branch left x right) = f [left', x, right']
  where
    left'  = applyBinIntTree f b left
    right' = applyBinIntTree f b right

maxBinIntTree' :: BinIntTree -> Int
maxBinIntTree' = applyBinIntTree maximum (minBound :: Int)

minBinIntTree' :: BinIntTree -> Int
minBinIntTree' = applyBinIntTree minimum (maxBound :: Int)
```

(g) (⋆) Écrire maintenant les fonctions

```
maxBinIntTree'' :: BinIntTree -> Maybe Int
minBinIntTree'' :: BinIntTree -> Maybe Int
```

qui retournent les étiquettes maximales et minimales (respectivement) d'un arbre binaire. Cette fois ci, l'étiquette de l'arbre vide est `Nothing`.

```
*Main> maxBinIntTree'' emptyBinIntTree
Nothing
*Main> maxBinIntTree'' binIntTreeExample
Just 20
*Main> minBinIntTree'' emptyBinIntTree
Nothing
*Main> minBinIntTree'' binIntTreeExample
Just 3
*Main>
```

Solution:

On remarque que :

```
ghci> maximum [Nothing]
Nothing
ghci> maximum [Nothing, Just 1]
Just 1
ghci> maximum [Just 2, Nothing, Just 1]
Just 2
ghci> maximum [Just 2, Nothing, Just 1, Nothing, Just 3]
Just 3
```

Essayez d'en déduire la logique. Il semble que maximum peut s'écrire de la façon suivante :

```
maximum []             = error "*** Exception: maximum: empty list"
maximum [x]            = x
maximum (Nothing : xs) = maximum xs
maximum (Just x : xs)  = case maximum xs of
                           Nothing -> Just x
                           Just x' -> if x > x' then Just x else Just x'
```

On peut alors en déduire facilement la fonction `maxBinIntTree''` :

```
import qualified Data.Maybe as Maybe

maxBinIntTree'' :: BinIntTree -> Maybe Int
maxBinIntTree'' Empty = Nothing
maxBinIntTree'' (Branch left x right) = maximum ms
  where
    maxLeft  = maxBinIntTree'' left
    maxRight = maxBinIntTree'' right
    ms       = [maxLeft, Just x, maxRight]

minBinIntTree'' :: BinIntTree -> Maybe Int
minBinIntTree'' Empty = Nothing
minBinIntTree'' (Branch left x right) = minimum ms
  where
    minLeft  = minBinIntTree'' left
    minRight = minBinIntTree'' right
    ms       = filter (isJust) [minLeft, Just x, minRight]
```

(h) Écrire la fonction

`heightBinIntTree :: (Ord a, Num a) => BinIntTree -> a`

qui retourne la hauteur de l'abre binaire. La hauteur de l'arbre vide est 0.

Solution:

```
heightBinIntTree :: (Ord a, Num a) => BinIntTree -> a
heightBinIntTree Empty = 0
heightBinIntTree (Branch left _ right) = 1 + max heightLeft heightRight
  where
    heightLeft  = heightBinIntTree left
    heightRight = heightBinIntTree right
```

(i) Écrire la fonction

`searchBinIntTree :: BinIntTree -> Int -> Bool`

qui retourne `True` si un entier donné apparaît dans un abre binaire.

Solution:

```
searchBinIntTree :: BinIntTree -> Int -> Bool
searchBinIntTree Empty _ = False
searchBinIntTree (Branch left x right) y = x == y     ||
                                           searchleft ||
                                           searchRight
  where
    searchleft  = searchBinIntTree left y
    searchRight = searchBinIntTree right y
```

(j) Écrire la fonction

`binIntTreeToList :: BinIntTree -> [Int]`

qui retournela liste des entiers qui apparaissent dans un abre binaire.

Solution:

```
binIntTreeToList :: BinIntTree -> [Int]
binIntTreeToList Empty = []
binIntTreeToList (Branch left x right) = [x]      ++
                                         leftList ++
                                         rightList
  where
    leftList  = binIntTreeToList left
    rightList = binIntTreeToList right
```

Question 2: Arbres binaires de recherche sur les entiers

Nous nous intéressons maintenant aux arbres binaires de recherche : les éléments dans le sous-arbre gauche sont inférieurs ou égaux à la racine, et ceux du sous-arbre droit sont supérieurs.

(a) Écrire la fonction

`searchBinIntTree' :: BinIntTree -> Int -> Bool`

qui retourne `True` si un entier donné apparaît dans un abre binaire **de recherche**.

Solution:

```
searchBinIntTree' :: BinIntTree -> Int -> Bool
searchBinIntTree' Empty _ = False
searchBinIntTree' (Branch left x right) y
  | x == y    = True
  | x >= y    = searchBinIntTree' left y
  | otherwise = searchBinIntTree' right y
```

(b) Écrire la fonction

```
insertBinIntTree :: BinIntTree -> Int -> BinIntTree
```

qui insert un entier dans un arbre binaire de recherche.

Solution:

```
insertBinIntTree :: BinIntTree -> Int -> BinIntTree
insertBinIntTree Empty y = Branch Empty y Empty
insertBinIntTree (Branch left x right) y
  | y <= x    = Branch (insertBinIntTree left y) x right
  | otherwise = Branch left x (insertBinIntTree right y)
```

Il est très important de remarquer que l'on reconstruit l'arbre binaire à chaque niveau.

(c) Écrire la fonction

```
deleteLargestInBinIntTree :: BinIntTree -> Maybe (Int, BinIntTree)
```

qui retourne une paire contenant l'élément maximum et l'arbre binaire de recherche obtenu en supprimant cet élément.

```
*Main> deleteLargestInBinIntTree emptyBinIntTree
Nothing
*Main> binIntTree
Branch (Branch (Branch Empty 3 Empty) 4 (Branch (Branch Empty 5 Empty) 8
Empty)) 10 (Branch (Branch Empty 15 Empty) 20 Empty)
*Main> deleteLargestInBinIntTree binIntTree
Just (20,Branch (Branch (Branch Empty 3 Empty) 4 (Branch (Branch Empty
5 Empty) 8 Empty)) 10 (Branch Empty 15 Empty))
*Main>
```

Solution:

```
deleteLargestInBinIntTree :: BinIntTree -> Maybe (Int, BinIntTree)
deleteLargestInBinIntTree Empty = Nothing
deleteLargestInBinIntTree (Branch left x Empty)  = Just (x, left)
deleteLargestInBinIntTree (Branch left x right) =
  case deleteLargestInBinIntTree right of
          Nothing          -> Nothing
          Just (y, right') -> Just (y, Branch left x right')
```

(d) Écrire la fonction

```
deleteBinIntTree :: BinIntTree -> Int -> BinIntTree
```

qui supprime - s'il existe - un entier dans un arbre binaire de recherche.

```
*Main> putStrLn $ prettyTree $ deleteBinIntTree binIntTreeExample 5
  20
    15
10
    8
  4
    3

*Main> putStrLn $ prettyTree $ deleteBinIntTree binIntTreeExample 8
  20
    15
10
    5
  4
    3

*Main> putStrLn $ prettyTree $ deleteBinIntTree binIntTreeExample 3
  20
    15
10
    8
      5
  4

*Main> putStrLn $ prettyTree $ deleteBinIntTree binIntTreeExample 10
  20
    15
8
    5
  4
    3

*Main>
```

Question 3: Arbres binaires de recherche

Mofifier votre code pour obtenir des arbres de recherche polymorphes. En même temps, organiser votre code sous forme d'un module **`BinTree`**.