Haskell (IR3) – Arbres Binaires

Stéphane Vialette

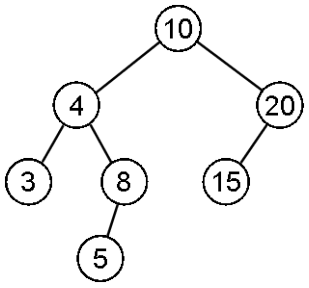
16 octobre 2019

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 permettront 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>
```

(b) Écrire la fonction

```
emptyBinIntTree :: BinIntTree
```

qui retourne un arbre binaire de recherche vide.

(c) Écrire la fonction

```
sizeBinIntTree :: Num a => BinIntTree -> a
```

qui retourne le nombre de nœuds dans un arbre binaire.

(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`.

(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`.

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

(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>
```

(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.

(i) Écrire la fonction

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

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

(j) Écrire la fonction

```
binIntTreeToList :: BinIntTree -> [Int]
```

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

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**.

(b) Écrire la fonction

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

qui insert un entier dans un arbre binaire de recherche.

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>
```

(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`.

Question 4: Pour aller plus loin

Écrire les fonctions permettant de fusionner 2 puis pusieurs arbres binaires de recherche.