Haskell (Info 3)
Arbres Binaires

Stéphane Vialette (stephane.vialette@univ-eiffel.fr)

29 novembre 2022

Question 1: Arbres binaires

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

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

(a) Examiner les types de **Empty** et **Branch**.

(b) Écrire dans l'interpréteur une expression, de type **BTree**, qui représente l'arbre suivant.

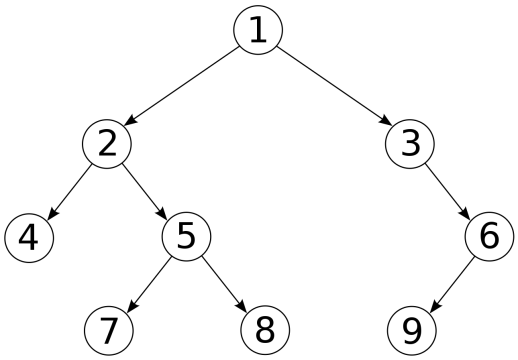

FIGURE 1 – Un arbre binaire.

Pour simplifier, écrire la fonction `exampleBT :: BTree Int` qui retourne l'arbre donnée en Fig. **??**.

Attention, si vous laissez haskell inférérer le typage de la fonction, vous aller sans doute obtenir `exampleBT :: BTree Integer`. Mais le type **Integer** n'implémente pas la classe **Bounded** (c'est bien normal pour des entiers arbitrairement longs) alors que cette contrainte de classe est nécéssaire dans la suite du TP.

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

```
indent :: Int -> String
indent = flip L.replicate '.'

str :: (Show a) => BTree a -> String
str = aux 0
```

```
  where
    aux k Empty             = indent k ++ "⊥\n"
    aux k (Branch lt x rt) = indent k ++ show x ++ "\n" ++
                             aux (k+1) lt ++ aux (k+1) rt
```

Nous pouvons les utiliser de la façon suivante :

```
ghci> let s = str exampleBT in putStrLn s
1
.2
..4
...⊥
...⊥
..5
...7
....⊥
....⊥
...8
....⊥
....⊥
.3
..⊥
..6
...9
....⊥
....⊥
...⊥
```

(c) Écrire la fonction

```
emptyBT :: BTree a
```

qui retourne un arbre binaire vide.

(d) Écrire la fonction

```
size :: Num b => BTree a -> b
```

qui retourne le nombre de sommets dans un arbre binaire.

(e) Écrire les fonctions

```
minBT :: (Ord a, Bounded a) => BTree a -> Maybe a
maxBT :: (Ord a, Bounded a) => BTree a -> Maybe a
```

La fonction `minBT` (resp. `maxBT`) retourne l'étiquette minmale (resp. maximale) dans un arbre binaire. Si l'arbre binaire est vide, la fonction retourne **`Nothing`**.

(f) Écrire la fonction

```
height :: (Ord b, Num b) => BTree a -> b
```

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

(g) Écrire la fonction

```
searchBT :: Eq a => BTree a -> a -> Bool
```

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

(h) Écrire la fonction

```
toList :: BTree a -> [a]
```

qui retourne la liste des éléments qui apparaissent dans un abre binaire. L'ordre des éléments n'est pas contraint ici (mais certains ordres sont plus faciles que d'autres !). Par exemple,

```
ghci> toList exampleBT
[1,2,4,5,7,8,3,6,9]
```

(i) Écrire les fonctions

```
preVisit :: BTree a -> [a]
inVisit :: BTree a -> [a]
postVisit :: BTree a -> [a]
```

qui retournent les éléments d'un arbre binaire par un parcours préfixe, infixe et suffixe. On pourra en profiter pour réécrire la fonction `toList`.

```
λ: preVisit exampleBT
[1,2,4,5,7,8,3,6,9]
λ: inVisit exampleBT
[4,2,7,5,8,1,3,9,6]
λ: postVisit exampleBT
[4,7,8,5,2,9,6,3,1]
```

(j) Écrire la fonction

```
filterBT :: (a -> Bool) -> BTree a -> [a]
```

qui retourne la liste des éléments d'un arbre binaire filtrée par un prédicat. Encore une fois, l'ordre des éléments n'est pas contraint.

```
λ: filterBT even exampleBT
[2,4,8,6]
λ: filterBT odd exampleBT
[1,5,7,3,9]
λ: filterBT (> 5) exampleBT
[7,8,6,9]
λ: filterBT (< 0) exampleBT
[]
```

(k) Écrire la fonction

```
mapBT :: (a -> b) -> BTree a -> BTree b
```

qui retourne un **nouvel** arbre binaire obtenu en appliquant une fonction sur chaque élément d'un arbre binaire. Par exemple,

```
λ: putStr $ str $ mapBT (+10) exampleBT
11
.12
..14
...⊥
...⊥
..15
...17
....⊥
....⊥
...18
....⊥
....⊥
.13
..⊥
..16
...19
....⊥
....⊥
...⊥
```

Question 2: Arbres binaires de recherche

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 strictement supérieurs.

(a) Écrire les fonctions

```
minBST :: (Ord a) => BTree a -> Maybe a
maxBST :: (Ord a) => BTree a -> Maybe a
searchBST :: (Ord a) => BTree a -> a -> Bool
```

(b) Écrire la fonction

```
insertBST :: (Ord a) => BTree a -> a -> BTree a
```

qui insert un élément dans un arbre binaire de recherche (les doublons sont autorisés).

(c) Écrire la fonction

```
deleteLargestBST :: BTree a -> Maybe (a, BTree a)
```

qui retourne une paire contenant l'élément maximum dans un arbre binaire de recherche et l'arbre binaire de recherche obtenu en supprimant cet élément. La fonction retourne `Nothing` si l'arbre est vide.

(d) Écrire la fonction

```
deleteBST :: (Ord a) => BTree a -> a -> BTree a
```

qui supprime - s'il existe - un élément dans un arbre binaire de recherche. La fonction retourne l'arbre non modifié si l'élément recherché ne se trouve pas dans l'arbre binaire de recherche. Pensez à utiliser `deleteLargestBST` dans la conception de votre algorithme.

(e) Écrire la fonction

```
deleteBST' :: (Ord a) => BTree a -> a -> Maybe (BTree a)
```

qui supprime - s'il existe - un élément dans un arbre binaire de recherche. La fonction retourne `Nothing` si l'élément recherché ne se trouve pas dans l'arbre binaire de recherche. Encore une fois, pensez à utiliser `deleteLargestBST`.

Question 3: (*) Pour aller (un peu) plus loin

(a) Ecrire la fonction

```
mkBST :: [a] -> BTree a
```

qui construit un arbre binaire de recherche à partir d'une liste d'éléments.

(b) Ecrire la fonction

```
levelVisit :: BTree a -> [a]
```

qui retourne la liste des éléments d'un arbre binaire par niveau.

```
ghci> levelVisit exampleBT
[10,4,20,3,8,15,5]
```