TD 7 : Data – Arbres Binaires de Recherche et Listes Infinies

1 Arbres Binaires de Recherche

On vous rappelle la notion des **Arbres Binaires de Recherche** : selon Wikipedia "Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud possède une clé, telle que chaque nœud du sous–arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous–arbre droit possède une clé supérieure à celle-ci. (Lien : `https://fr.wikipedia.org/wiki/Arbre_binaire_de_recherche`

Pour cet exercice on va implémenter les arbres binaires de recherche en Haskell en utilisant la définition suivante :

```
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show
```

Notez bien que le type `Tree` est un type paramétrique : on peut utiliser un `Tree Int`, `Tree String`, `Tree Float`, ... Quelles sont les contraintes qui s'imposent sur le type `a` ?

Programmer les fonctions suivantes (et précisez leurs types) :

1. `list2tree` : étant donné une liste d'éléments retourne un arbre binaire de recherche avec les mêmes éléments.
2. `tree2list` : fait la conversion inverse, mais retourne une liste triée en ordre croissant.
3. `find` : étant donné un élément et un arbre retourne `True` si l'élément paraît dans l'arbre. Évidemment, cette fonction est censée utiliser la structure de l'arbre pour accélérer la recherche.

Exemples :

```
*Main> list2Tree "hello"
Node (Node Leaf 'e' Leaf) 'h' (Node (Node Leaf 'l' Leaf) 'l' (Node Leaf 'o' Leaf))
*Main> list2Tree [1..5]
Node Leaf 1 (Node Leaf 2 (Node Leaf 3 (Node Leaf 4 (Node Leaf 5 Leaf))))
*Main> tree2list  (list2Tree "abracadabra")
"aaaaabbcdrr"
*Main> map (\x -> find x $ list2Tree "abracadabra") ['a'..'e']
[True,True,True,True,False]
*Main> map (\x -> find x $ list2Tree [1,3..10]) [1..10]
[True,False,True,False,True,False,True,False,True,False]
```

1.1 Efficacité

La motivation principale pour utiliser les arbres binaires de recherche est que la recherche d'un élément peut être beaucoup plus efficace dans un arbre que dans une liste non–triée. Plus précisément, la complexité de rechercher un élément dans une liste est $O(n)$ (où n est la taille de la liste), alors que dans un arbre binaire de recherche elle est $O(h)$ où h est la hauteur de l'arbre (la distance maximum entre la racine et une feuille).

Avec une implémentation naïve de la fonction `list2tree` la hauteur de l'arbre produit pourrait être $O(n)$ au pire des cas, et donc on ne gagne rien par rapport à l'utilisation d'une liste. Cependant, pour n'importe quelle liste, il existe une façon de construire un arbre binaire de recherche de hauteur $O(\log n)$, si on prend soin de diviser les éléments de la liste dans deux sous–listes de la même taille.

Programmer une fonction `height` qui, étant donné un arbre calcule son hauteur. Puis, donner une implémentation de `list2tree` qui produit toujours un arbre de hauteur $O(\log n)$.

Exemples :

```
*Main> list2Tree [1..8] -- BAD!
Node Leaf 1
   (Node Leaf 2
     (Node Leaf 3
        (Node Leaf 4
          (Node Leaf 5 (Node Leaf 6 (Node Leaf 7 (Node Leaf 8 Leaf)))))))
*Main> list2Tree' [1..8] -- GOOD!
Node (Node (Node (Node Leaf 1 Leaf) 2 Leaf) 3 (Node Leaf 4 Leaf)) 5
     (Node (Node Leaf 6 Leaf) 7 (Node Leaf 8 Leaf))
*Main> height $ list2Tree [1..127]
127
*Main> height $ list2Tree' [1..127]
7
```

Hint : pour partitionner la liste en deux sous–listes de la même taille vous aurez besoin de calculer l'élément médian (et le mettre dans la racine de l'arbre produit). La façon la plus simple pour le faire (même si ce n'est pas la façon la plus efficace) est de trier la liste et regarder l'élément en position $n/2$. Vous pouvez trier la liste en utilisant l'implémentation (naïve) des arbres binaires de recherche qu'on a déjà faite.

1.2 Bonus : Map et Arbres Binaires

Un arbre binaire peut être vu comme un conteneur, c'est-à-dire, une structure qui contient des éléments. Cela nous rappelle les listes, qui sont les conteneurs les plus basiques de Haskell. On peut donc poser la question suivante : est-ce possible de programmer une nouvelle version de quelques fonctions basiques qui manipulent les listes pour traiter les arbres. Dans cette partie on regarde la fonction `map`

1. Écrire une fonction `treeMap :: (a->b) -> Tree a -> Tree b` qui prend comme arguments une fonction `f` et un arbre et applique `f` sur chaque élément de l'arbre.
2. Si `t` est un arbre binaire de recherche, l'expression `treeMap f t` ne retourne pas forcément un arbre binaire de recherche valide (pourquoi ?). Écrire une fonction `safeMap` qui retourne un `Maybe Tree`. Cette fonction applique la fonction `f` sur les éléments de l'arbre `t` et si le résultat est un arbre binaire de recherche valide `t'` elle retourne `Just t'`, sinon, elle retourne `Nothing`.

NB : comme on l'a vu pendant le cours, le fait que `map` peut être programmé pour les arbres (et en principe pour d'autres types de conteneurs) n'est pas un hasard. Ce point commun forme la base de la très importante classe `Functor`.

2 Listes Infinies

On a déjà vu que Haskell nous permet d'utiliser des listes de taille infinie, grâce à l'évaluation paresseuse. On a aussi vu pendant le cours une implémentation alternative des listes en utilisant `Data`, qui nous permet elle aussi de construire des listes infinies. Pour cet exercice on va aller plus loin et définir une variation de cette structure où **toutes** les listes sont obligatoirement infinies. Malgré ça, on va écrire quelques fonctions simples pour utiliser de telles listes.

On commence avec la définition de type suivante :

```
data InfList a = Cons a (InfList a) deriving Show
```

Rappel : pendant le cours on avait fait `data List a = Nil | Cons a (List a)`. Donc, on a éliminé l'option de construire une liste vide, et avec elle la possibilité de construire une liste de taille finie. Cependant, le compilateur accepte cette définition de type.

Programmer les fonctions suivantes (et préciser leurs types !) :

1. `makelist` : prend un élément `x` et retourne la liste `[x,x,x,x,x,....]`.
2. `mytake` : prend un `Int` n et une `InfList` l et retourne une liste (classique) avec les n premiers éléments de l.
3. `myfilter` : fonctionne comme `filter` mais pour les `InfList` (donc retourne une `InfList` ayant supprimé les éléments qui ne satisfont pas la condition donnée)
4. `suiteArithm` : prend deux `Int` x, y et retourne la `InfList` $x, x+y, x+2y, x+3y, x+4y, \ldots$.
5. (**Bonus** :) Donner une version plus générique de `suiteArithm` qui prend un x de type `Enum a => a` et un `Int` y. Pour rappel, la classe `Enum` contient les type énumerables, c'est-à-dire les types dont les valeurs peuvent être ordonnées et numérotées (par exemple le type `Char`). Voir ci-dessous pour un exemple d'utilisation de cette version. Vous devrez ici utiliser la fonction `succ` de la classe `Enum`.

Exemples :

```
*Main> mytake 20 $ makelist 'a'
"aaaaaaaaaaaaaaaaaaaa"
*Main> suitear 5 7
Cons 5 (Cons 12 (Cons 19 (Cons 26 (Cons 33 (Cons 40 (Cons 47 (Cons 54 (Cons 61...
*Main> mytake 10 $ suitear 'a' 2
"acegikmoqs"
*Main> mytake 20 $ myfilter (\x -> x `mod` 3 == 1) $ suitear 1 4
[1,13,25,37,49,61,73,85,97,109,121,133,145,157,169,181,193,205,217,229]
```