TP 3 : Arbres Binaires de Recherche (ABR)

Un Arbre Binaire de Recherche (ABR) est un arbre binaire tel que, pour tout noeud portant une valeur x, toutes les valeurs portées par son fils gauche sont inférieures à x, et toutes les valeurs portées par son fils droit sont supérieures à x.

On définit le type suivant pour représenter les ABRs :

```
data Btree a = Nil
             | Bin a (Btree a) (Btree a)
     deriving (Show, Ord, Eq)
```

Voici quelques exemples d'arbres binaires de recherche :

```
a1 = (Bin 4 (Bin 3 (Bin 2 Nil Nil) Nil) (Bin 7 (Bin 6 Nil Nil) (Bin 8 Nil Nil)))

a2 = (Bin 4 (Bin 3 (Bin 1 Nil Nil) Nil) (Bin 8 (Bin 7 Nil Nil) (Bin 11 (Bin 9 Nil Nil) Nil)))

a3 = (Bin 40
       (Bin 30 (Bin 20 (Bin 15 Nil Nil) (Bin 25 Nil Nil)) (Bin 35 Nil Nil))
       (Bin 70 (Bin 60 (Bin 50 (Bin 45 Nil Nil) Nil) (Bin 65 Nil Nil)) (Bin 80 Nil Nil)))
```

1. **Premiers pas.**

   (a) A l'aide de la fonction `(voir t)`[1], visualiser les ABR `a1`, `a2` et `a3`.

   (b) Afficher les arbres `a4 = (Bin 10 a1 a3)` et `a5 = (Bin 17 a1 a3)`.
   Vérifier que `a4` est un ABR, mais que `a5` n'en est pas un.

   (c) Définir la fonction `(profondeur t)` qui calcule la profondeur d'un ABR `t`.

   ```
   > profondeur (Bin 2 Nil Nil)  ==> 0
   > profondeur a1  ==> 2
   ```

   (d) Définir les fonctions `prefixe, postfixe, infixe` qui effectuent respectivement un parcours en ordre `RGD` (préfixé), `GDR` (postfixé), `GRD` (infixé) d'un ABR et retournent la liste (dans l'ordre) des valeurs rencontrées.

   ```
   > prefixe a1   ==> [4,3,2,7,6,8]
   > infixe a1   ==> [2,3,4,6,7,8]
   > postfixe a1   ==> [2,3,6,8,7,4]
   ```

2. **Appartenance.** Compléter la définition de la fonction `(inBtree x t)` qui détermine l'appartenance d'un élément `x` à un ABR `t`.

   ```
   > inBtree 2 a1  ==> True
   > inBtree 20 a1 ==> False

   inBtree :: Ord a => a -> Btree a -> Bool
   inBtree x Nil =
   inBtree x (Bin y t1 t2)
        | x < y     =
        | x > y     =
        | otherwise =
   ```

[1] fichier `initTP3.hs` à télécharger

3. **Insertion.**

   (a) Définir la fonction `(insere x t)` qui insère l'élément `x` à sa place dans l'ABR `t`. *Indication : s'inspirer de la définition de la fonction* `(inBtree x t)`

   (b) En utilisant la fonction `(list2abr xs)` ci-dessous, définir la fonction `(trier xs)` qui ordonne les éléments d'une liste. *Indication : utiliser la question* `1.d`.

```
list2abr :: Ord a => [a] -> (Btree a)
list2abr [] = Nil
list2abr (x:xs) = insere x (list2abr xs)
```

   (c) Soit `x` un élément d'un ABR `t`. Définir la fonction `(path t x)` qui calcule la liste des valeurs des nœuds permettant d'aller de la racine de `t` à l'élément `x`.

```
path a3 15  ==> [40, 30, 20, 15]
path a3 20  ==> [40, 30, 20]
path a3 30  ==> [40, 30]
path a3 40  ==> [40]
```

4. **Equilibrage.** On souhaite déterminer si un ABR est équilibré.

   (a) Vérifier que la version ci-dessous est correcte.

```
estEquilibre :: Btree a -> Bool
estEquilibre Nil = True
estEquilibre (Bin _ t1 t2) = (abs (profondeur t1 - profondeur t2) <= 1)
                             && estEquilibre t1
                             && estEquilibre t2
```

   (b) Pourquoi ette définition engendre-t-elle un nombre d'appels "excessivement" élevé à la fonction `(profondeur t)` ?

5. **Dans cet exercice, les fonctions seront définies pour des ABR dont tous les éléments sont des nombres entiers.**

   (a) Définir la fonction `(sommeArbre t)` qui calcule la somme des éléments d'un ABR `t` (étiqueté par des entiers).

```
> sommeArbre a1   ==> 30
> sommeArbre a2   ==> 43
> sommeArbre a3   ==> 535
```

   (b) Définir une fonction `(minArbre t)` qui détermine le plus petit élément d'un ABR non vide (c-à-d non réduit à `Nil`). Définir de même `maxArbre`.

```
> minArbre a4   ==> 2
> maxArbre a4   ==> 80
```

(c) La fonction `(join t1 t2)` définie ci-dessous prend en arguments deux ABR `t1` et `t2` tels que le plus grand élément de `t1` soit strictement inférieur au plus petit élément de `t2` . Préciser le résultat que cette fonction retourne. Exprimer la profondeur de l'ABR `(join t1 t2)` en fonction des profondeurs de `t1` et `t2`.

```
join :: Btree Integer -> Btree Integer -> Btree Integer
join t1 Nil = t1
join Nil t2 = t2
join (Bin x u1 u2) t2 = Bin x u1 (join u2 t2)
```

(d) En utilisant la fonction `(join t1 t2)`, définir une fonction `(delete x t)` qui supprime un élément donné dans un ABR. Compléter la définition suivante:

```
delete :: Integer -> Btree Integer -> Btree Integer
delete x Nil =
delete x (Bin y t1 t2)
  | x < y     =
  | x > y     =
  | otherwise =
```

Comparer les profondeurs de `a4` et de l'arbre obtenu avec `(delete 10 a4)`.

(e) Définir une fonction `(deleteMin t)` qui supprime le plus petit élément dans un ABR non vide (c-à-d non réduit à `Nil`) de façon que l'ABR obtenu en résultat ait une profondeur inférieure ou égale à la profondeur de l'ABR donné en argument.
Définir de la même façon une fonction `(deleteMax t)`.

```
> profondeur a2   ==> 3
> profondeur (deleteMax a2)   ==> 2
> profondeur a3   ==> 4
> profondeur (deleteMin a3)   ==> 4
> profondeur a4   ==> 5
> profondeur (deleteMax (deleteMax a4)) ==> 4
```

(f) Définir une fonction `delete1` qui supprime un élément donné dans un ABR de façon que l'ABR obtenu en résultat ait une profondeur inférieure ou égale à la profondeur de l'ABR pris comme argument.

```
> profondeur a4   ==> 5
> profondeur (delete1 10 a4)   ==> 5
```

6. **Question avancée.** En quoi cette définition de l'équilibrage corrige-t-elle le défaut constaté à la question 4.b ?

```
estEquilibreBis:: Btree a -> Bool
estEquilibreBis t = fst (aux t)

aux :: (Btree a) -> (Bool, Int)
aux Nil = (True, -1)
aux (Bin _ fg fd)
   | not eqg || not eqd || abs (htg - htd) > 1 = (False, 0)
   | otherwise                                 = (True, 1 + (max htg htd))
      where (eqg, htg) = aux fg
            (eqd, htd) = aux fd
```