Université de Bordeaux Licence S&TS Mention Info et MI, S4 2018/2019

Algorithmique des structures de données arborescentes

TP Noté — 15 avril 2019 — durée 1h20

Dans ce TP noté, on utilise le type `'a tree` suivant :

```
type 'a tree = Empty | Bin of 'a * 'a tree * 'a tree
```

On s'intéresse aux arbres binaires *enrichis* : un arbre est dit enrichi si chacun de ses nœuds est étiqueté par un couple (x, y), où y est le **nombre de nœuds** du sous-arbre enraciné en ce nœud. Ainsi, un arbre enrichi est représenté par un objet de type `('a * int) tree`.

Un exemple est donné sur la Figure 1. L'arbre de gauche est un arbre dont les nœuds portent juste une clé entière. Par exemple, la clé de la racine est 42. Le dessin de droite montre l'arbre enrichi correspondant. En plus de la clé originale, chaque nœud porte, en seconde composante, le **nombre de nœuds** de son sous-arbre. Par exemple, le sous-arbre du nœud de clé 84 a **3** nœuds (le nœud de clé 84 lui-même, celui de clé 0 et celui de clé 99).

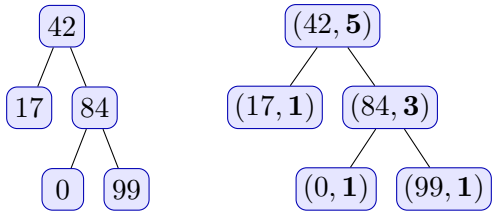


FIG. 1 : Un arbre et l'arbre enrichi correspondant

Il faut noter que dans un arbre enrichi, lorsqu'un nœud a une étiquette (x, y), la première composante du couple, x, est arbitraire. Elle peut donc elle-même être un couple de la forme (c, m), où, par exemple, m représente une multiplicité. Ainsi, l'arbre de droite de la figure suivante représente l'enrichissement de l'arbre de gauche.

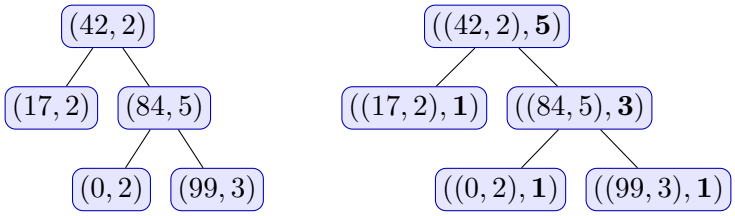


FIG. 2 : Un autre arbre et l'arbre enrichi correspondant

Exercice 1 Écrire une fonction `size : ('a * int) tree -> int` retournant le nombre de nœuds de l'arbre enrichi passé en argument.

Remarque importante. Cette fonction est simple et non récursive. On demande une complexité $O(1)$. Elle pourra être utilisée dans les exercices suivants.

Exercice 2 Écrire une fonction `enrich : 'a tree -> ('a * int) tree` qui renvoie l'arbre enrichi correspondant à son argument. Par exemple, si on lui passe l'arbre de gauche de la Figure 1, la fonction retourne l'arbre de droite (idem avec les arbres de la Figure 2).

On demande une fonction de **complexité linéaire** en la taille de l'arbre en entrée ; il est conseillé de parcourir l'arbre une seule fois.

Exercice 3 On représente des multiensembles comme des arbres binaires de recherche enrichis. Chaque nœud contient donc un élément de la forme `((cle, mult), size)`, où `cle` est un élément du multiensemble, `mult` sa multiplicité, et `size` la taille du sous-arbre enraciné en ce nœud. Ainsi, un multiensemble est représenté par un objet de type `(('a * int) * int) tree`. C'est précisément la situation de la Figure 2.

Écrire une fonction

```
insert : 'a -> (('a * int) * int) tree -> (('a * int) * int) tree
```

prenant en entrée une clé k et un arbre enrichi représentant un multiensemble t et qui renvoie l'arbre enrichi représentant le multiensemble obtenu en insérant k dans t.

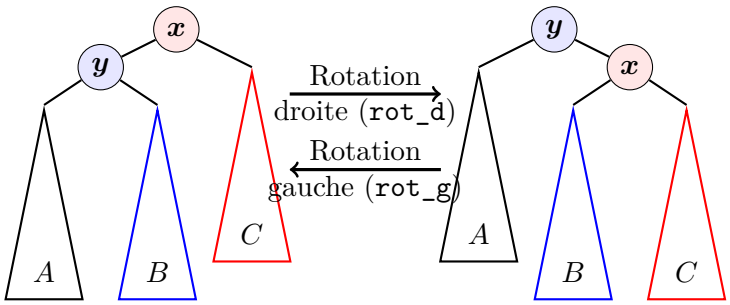
La complexité de votre fonction doit être la même que celle de l'insertion dans un arbre binaire de recherche classique.

Remarques. L'arbre d'entrée comme l'arbre construit doivent être des ABR. Par ailleurs, pour obtenir la bonne complexité, vous devrez produire un arbre enrichi en ne recalculant **que certaines** des tailles attachées aux nœuds.

Exercice 4 Écrire deux fonctions `rot_g` et `rot_d` de type

```
('a * int) tree -> ('a * int) tree
```

effectuant les transformations suivantes sur l'arbre de recherche enrichi pris en argument :



Le résultat doit rester un arbre de recherche enrichi. Si l'arbre n'a pas la forme requise, vous renverrez l'arbre donné en argument.

Remarque. Les fonctions sont simples et non récursives. N'oubliez pas d'effectuer les changements nécessaires sur la composante "taille" des nœuds.

Exercice 5 À l'aide des deux fonctions de l'exercice 5, écrire une fonction `remove` de type

```
'a -> (('a * int) * int) tree -> (('a * int) * int) tree
```

qui prend en paramètre une clé k et un arbre enrichi t qui représente un multiensemble, et qui retourne l'arbre représentant le multiensemble obtenu en supprimant une occurrence de k dans t (s'il y en a), et en utilisant la stratégie décrite ci-dessous.

Pour supprimer un nœud de clé k dont les deux sous-arbres G et D sont non vides, on utilisera une rotation pour faire **descendre** le nœud vers les feuilles en préservant la propriété d'arbre de recherche enrichi. On utilisera une rotation droite quand G a au moins autant de nœuds que D, et une rotation gauche sinon.

Pour supprimer un nœud dont au moins un des deux sous-arbres est vide, on utilisera le même algorithme que celui vu pour les arbres binaires de recherche classiques – en maintenant la propriété d'être un arbre enrichi.