ABR en OCaml

1 Fonctions élémentaires

On considère le type suivant :

```
type 'a abr =
  | V
  | N of 'a abr * 'a * 'a abr
```

1.1 Exercice

Écrire les fonctions suivantes (les spécifications devraient être évidentes) :

```
insere : 'a abr -> 'a -> 'a abr
appartient : 'a abr -> 'a -> bool
cardinal : 'a abr -> int
```

1.2 Exercice

1. Écrire la fonction `construit` qui prend en entrée une liste d'objets de type `'a` et renvoie l'arbre binaire de recherche obtenu en insérant successivement tous les éléments de la liste, dans l'ordre, dans un arbre initialement vide.
2. Écrire la fonction `elements` qui renvoie la liste des éléments d'un arbre binaire de recherche, dans l'ordre croissant. On exige une complexité en $O(|t|)$.

```
construit : 'a list -> 'a abr
elements : 'a abr -> 'a list
```

1.3 Exercice

1. Écrire une fonction `extrait_min` qui prend en entrée un ABR `t`, supposé non vide, et renvoie le couple (m, t'), où :
   — m est le minimum de t ;
   — t' est l'arbre binaire de recherche obtenu en supprimant m de t.
2. Écrire la fonction `supprime` qui supprime un élément d'un arbre binaire de recherche. Si l'élément fourni n'appartient pas à l'arbre, ce dernier sera renvoyé inchangé.

```
extrait_min : 'a abr -> 'a * 'a abr
supprime : 'a abr -> 'a -> 'a abr
```

2 Fonctions supplémentaires

2.1 Séparation d'un ABR

Écrire une fonction `separe` telle que l'appel `separe t x` renvoie un couple (inf, sup) d'ABR vérifiant :
— tous les éléments de inf sont inférieurs ou égaux à x ;
— tous les éléments de sup sont strictement supérieurs à x ;
— la réunion des éléments de inf et de ceux de sup est égal à l'ensemble des éléments de t.
On demande une complexité en $O(h(t))$.

2.2 Exercice

1. Écrire une fonction `verifie_abr` qui détermine si l'arbre passé en argument vérifie la condition d'ordre des ABR. On n'hésitera pas à utiliser les fonctions préalablement définies, et l'on précisera les complexités en temps et en espace de `verifie_abr`.
2. Écrire une fonction `tab_elements` qui convertit un ABR en un tableau trié.
3. Ré-écrire les fonctions `verifie_abr` et `tab_elements` pour que leur complexité en espace (sans compter la taille du résultat pour `tab_elements`) soit en $O(h(t))$[1]. Pour la la fonction `verifie_abr`, on pourra se limiter au cas des arbres à étiquettes entières (et supposer qu'aucun nœud ne porte l'étiquette `min_int`).

3 Structure de multi-ensemble ordonné

On considère un type totalement ordonné `'a`, et l'on souhaite représenter des *multi-ensembles* d'éléments de `'a` de manière à pouvoir réaliser un certain nombre d'opérations de manière efficace. On rappelle que dans un multi-ensemble, chaque élément possède une *multiplicité* (ou nombre d'occurrences).

La liste des opérations qui nous intéressent :

— `get_occurrences : 'a multiset -> 'a -> int` qui renvoie le nombre d'occurrences (éventuellement nul) d'un objet de type `'a` dans un `'a multiset` ;

— `add_occurrence : 'a multiset -> 'a -> 'a multiset` qui ajoute une occurrence ;

— `rem_occurrence : 'a multiset -> 'a -> 'a multiset` qui enlève une occurrence ;

— `size : 'a multiset -> int` qui renvoie le nombre total d'éléments dans un multi-ensemble, en tenant compte de la multiplicité ;

— `select : 'a multiset -> int -> 'a` qui renvoie x_i, où $x_0 < x_1 < \cdots < x_{size(t)}$ sont les éléments de t, avec multiplicité (on lèvera une exception si i n'est pas un indice valide).

3.1 Utilisation d'un dictionnaire

Une première idée serait de remplacer la structure (g, x, r) d'un ABR par (g, x, mul, r), où `mul` est un entier (strictement positif) indiquant la multiplicité de `x`. Cette idée fonctionne, et correspond en fait à un cas particulier de dictionnaire à clé de type `'a` et valeur de type `int`.

3.2 Exercice

On définit le type suivant :

```
type ('k, 'v) dict =
  | Empty
  | Node of ('k, 'v) dict * 'k * 'v * ('k, 'v) dict
```

Écrire les fonctions suivantes (vues en cours) :

1. `get` qui renvoie `Some v` si la clé fournie est associée à la valueur `v`, `None` sinon ;
2. `set` qui crée une association, ou remplace la valeur associée à une clé s'il y en avait déjà une ;
3. `remove` qui supprime l'association correspondant à la clé fournie s'il y en avait une, et ne fait rien sinon.

```
get : ('k, 'v) dict -> 'k -> 'v option
set : ('k, 'v) dict -> 'k -> 'v -> ('k, v') dict
remove : ('k, 'v) dict -> 'k -> ('k, v') dict
```

3.3 Exercice

1. Écrire les foncions `get_occurrences`, `add_occurrence` et `rem_occurrence` à l'aide des fonctions `get`, `set` et `remove`.
2. Donner la complexité de ces trois fonctions.

```
get_occurrences : ('a, int) dict -> 'a -> int
add_occurrence : ('a, int) dict -> 'a -> ('a, int) dict
rem_occurrence : ('a, int) dict -> 'a -> ('a, int) dict
```

1. On réfléchira aussi à la question suivante : pourquoi l'énoncé demande-t-il $O(h(t))$ et non $O(1)$?

3.4 Exercice

1. Écrire la fonction `size`, et déterminer sa complexité.
2. Proposer un algorithme pour la fonction `select` (on ne demande pas de l'implémenter en OCaml).
3. Quelle est la complexité de cet algorithme ?

```
size : ('a, int) dict -> int
select : ('a, int) dict -> int -> 'a
```

3.5 Enrichissement de la structure

Pour obtenir des fonctions `select` et `size` plus efficaces, on décide d'enrichir la structure en ajoutant dans chaque nœud (non vide) un entier indiquant la taille (nombre d'éléments avec multiplicité) du sous-arbre correspondant.

```
type 'a multiset =
  | Empty
  | Node of int * 'a multiset * 'a * int * 'a multiset
```

On maintiendra les invariants suivants :
— en considérant uniquement les étiquettes de type `'a`, on a un ABR ;
— dans un nœud `t = Node (n, left, x, i, right)`, on a $i > 0$ et n égal à la taille de t (avec multiplicité).

3.6 Exercice

1. Écrire les fonctions `get_occurrences`, `add_occurrence` et `rem_occurrence`.
2. Écrire les fonctions `size` et `select`, et déterminer leur complexité.

4 Un problème pour finir

Ce problème est adapté du *Projet Euler* (`projecteuler.net`), site qui contient des centaines de problèmes intéressants sur lesquels vous pouvez travailler. On note $(p_n)_{k\geqslant 1}$ la suite des nombres premiers :

$$p_1 = 2, \quad p_2 = 3, \quad p_3 = 5, \quad \text{etc.}$$

On définit deux suites $(u_n)_{n\geqslant 1}$ et $(v_n)_{n\geqslant 1}$ par :

$$\forall n \in \mathbb{N}^*, \qquad \begin{aligned} u_n &:= p_n^n \bmod 10\,007 \\ v_n &:= u_n + u_{\lfloor n/10\,000 \rfloor + 1} \end{aligned}$$

On définit ensuite $M(i, j)$ pour $i \leqslant j$ comme la médiane des éléments $v_i, \dots, v_j$, en convenant que la médiane d'une série de longueur paire est la moyenne des deux éléments centraux. On a alors $M(1, 10) = 2021.5$ et $M(10^2, 10^3) = 4715$. Finalement, on pose

$$F(n, k) := \sum_{i=1}^{n-k+1} M(i, i + k - 1).$$

On a alors $F(100, 10) = 433628.5$.

1. En utilisant ce que l'on a fait depuis le début du sujet, déterminer $F(10^5, 10^4)$.
2. En essayant de garder une consommation mémoire raisonnable, c'est-à-dire quelques centaines de méga-octets, mais pas quelques giga-octets, déterminer $F(10^7, 10^5)$.
3. Proposer une solution plus simple en utilisant le fait que le modulo utilisé est petit.