TD XXVII

ARBRES BINAIRES DE RECHERCHE EN OCAML

1 Fonctions élémentaires

On considère le type suivant :

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

Exercice XXVII.1

p. 5

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

```
insere : 'a abr -> 'a -> 'a abr

appartient : 'a abr -> 'a -> bool

cardinal : 'a abr -> int
```

Exercice XXVII.2

p. 5

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
```

Exercice XXVII.3

p. 6

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

Exercice XXVII.4 – Séparation d'un ABR

p. 6

Écrire une fonction `separe` telle que l'appel `separe t x` renvoie un couple $(\mathrm{inf}, \mathrm{sup})$ 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 $O(h(t))$.

Exercice XXVII.5

p. 6

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 $O(h(t))$[a]. 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`).

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

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 $x_0 < x_1 < \cdots < x_{\mathtt{size}(t)}$ 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`.

Exercice XXVII.6

p. 8

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 valeur `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
```

Exercice XXVII.7

p. 9

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
```

Exercice XXVII.8

p. 9

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
```

3.2 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).

Exercice XXVII.9 p. 10

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 (problèmes beaucoup plus mathématiques en moyenne que ceux de France-IOI).

On définit deux suites $(u_k)_{k\geqslant 1}$ et $(v_k)_{k\geqslant 1}$ par :

- $u_k = (p_k)^k \text{mod } 10007$, où p_k est le k-ème nombre premier (avec donc $p_1 = 2$) ;
- $v_k = u_k + u_{\lfloor k/10000 \rfloor + 1}$

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 $M(1,10) = 2\,021{,}5$ et $M(10^2, 10^3) = 4\,715$.

Finalement, on pose $F(n,k) = \sum_{i=1}^{n-k+1} M(i, i+k-1)$. On a alors $F(100,10) = 433\,628{,}5$.

Exercice XXVII.10

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 (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.

Solutions

Correction de l'exercice XXVII.1 page 1

```
let rec insere t x =
  match t with
  | V -> N (V, x, V)
  | N (l, y, r) ->
     if x = y then t
     else if x < y then N (insere l x, y, r)
     else N (l, y, insere r x)

let rec appartient t x =
  match t with
  | V -> false
  | N (l, y, r) ->
     (x = y) || (x < y && appartient l x) || (x > y && appartient r x)

let rec cardinal t =
  match t with
  | V -> 0
  | N (gauche, _, droite) -> 1 + cardinal gauche + cardinal droite
```

Correction de l'exercice XXVII.2 page 1

```
let construit u =
  let rec aux v acc =
    match v with
    | [] -> acc
    | x :: xs -> aux xs (insere acc x) in
  aux u V

(* Version efficace (en O(|t|)). *)
let elements t =
  let rec aux arbre acc =
    match arbre with
    | V -> acc
    | N (g, x, d) ->
      let avec_d = aux d acc in
      aux g (x :: avec_d) in
  aux t []
```

Correction de l'exercice XXVII.3 page 1

```
let rec extrait_min t =
  match t with
  | V -> failwith "vide"
  | N (V, x, d) -> (x, d)
  | N (g, x, d) ->
    let m, g' = extrait_min g in
    (m, N (g', x, d))

let rec supprime t x =
  match t with
  | V -> V
  | N (g, y, d) when x < y -> N (supprime g x, y, d)
  | N (g, y, d) when x > y -> N (g, y, supprime d x)
  | N (V, y, d) (* y = x *) -> d
  | N (g, y, V) (* y = x *) -> g
  | N (g, y, d) (* y = x *) ->
    let successeur, d' = extrait_min d in
    N (g, successeur, d')
```

Correction de l'exercice XXVII.4 page 2

On descend le long d'une branche de l'arbre jusqu'à trouver x ou arriver à un nœud vide, en faisant des opérations en temps constant à chaque niveau : la complexité est bien en $O(h(t))$.

```
let rec separe t x =
  match t with
  | V -> V, V
  | N (l, y, r) ->
     if x = y then N (l, y, V), r
     else if x < y then
       let lo, hi = separe l x in
       lo, N (hi, y, r)
     else
       let lo, hi = separe r x in
       N (l, y, lo), hi
```

Correction de l'exercice XXVII.5 page 2

1. Un arbre est un ABR si et seulement si ses étiquettes lues dans l'ordre infixe forment une suite (strictement) croissante :

```
let verifie_abr t =
  let rec croissant u =
    match u with
    | x :: y :: xs -> (x < y) && croissant (y :: xs)
    | _ -> true in
  croissant (elements t)
```

La fonction `elements` a une complexité en temps en $O(|t|)$, et la fonction `croissant` en $O(|u|) = O(|t|)$. On obtient donc une complexité en temps en $O(|u|)$.
Pour la complexité en espace, il y a deux choses à considérére :

- le stockage auxiliaire est constitué de la liste `elements t`, de longueur $|t|$;

- il faut aussi prendre en compte l'espace consommé sur la pile d'appels : la fonction auxiliaire de `elements` a une profondeur d'appel en $O(h(t))$ et la fonction `croissant` en $O(|t|)$ dans le pire cas (qui sera atteint dès que `t` est effectivement un ABR).

On obtient donc une complexité en espace en $O(|t|)$.

2. Avec tout ce que l'on a déjà écrit, le plus simple est clairement :

```
let tab_elements t =
  Array.of_list (elements t)
```

3. Le parcours de l'arbre (qui est clairement nécessaire) nécessite un espace $O(h(t))$, que ce soit pour la pile d'appel si on l'effectue récursivement ou pour la pile « tout court » si on choisit une version itérative (ou récursive terminale). Pour ne pas dépenser plus que cela, il faut se débarrasser de la liste `elements t`.

Pour `verifie_abr`, plusieurs solutions sont envisageables : on en présente deux ici.

```
let verifie_abr_bis t =
  let courant = ref min_int in
  let rec aux t =
    match t with
    | V -> true
    | N (g, x, d) ->
      aux g && x > !courant && (courant := x; aux d) in
  aux t

exception Faux

let verifie_abr_exception t =
  let courant = ref min_int in
  let rec aux t =
    match t with
    | V -> ()
    | N (g, x, d) ->
      aux g;
      if x <= !courant then raise Faux;
      courant := x;
      aux d in
  try
    aux t;
    true
  with
  | Faux -> false
```

Pour `tab_elements`, on peut aussi procéder de plusieurs manières mais le plus simple est sans doute :

```ocaml
let tab_elements_bis t =
  match t with
  | V -> [| |]
  | N (_, x, _) ->
    let n = cardinal t in
    let arr = Array.make n x in
    let indice = ref 0 in
    let rec aux arbre =
      match arbre with
      | V -> ()
      | N (g, x, d) ->
        aux g;
        arr.(!indice) <- x;
        incr indice;
        aux d in
    aux t;
    arr
```

On prend garde à gérer correctement le cas où l'arbre est vide, et à ne pas se limiter aux arbres à étiquettes entières (ce qui nécessite de récupérer une étiquette pour initialiser le tableau).

Correction de l'exercice XXVII.6 page 2

```ocaml
let rec get dict key =
  match dict with
  | Empty -> None
  | Node (left, k, v, right) ->
    if key = k then Some v
    else if key < k then get left key
    else get right key

let rec set dict key value =
  match dict with
  | Empty -> Node (Empty, key, value, Empty)
  | Node (left, k, v, right) ->
    if key = k then Node (left, k, value, right)
    else if key < k then Node (set left key value, k, v, right)
    else Node (left, k, v, set right key value)

let rec extract_min = function
  | Empty -> failwith "empty"
  | Node (Empty, k, v, right) ->
    (k, v, right)
  | Node (left, k, v, right) ->
    let km, vm, left' = extract_min left in
    (km, vm, Node (left', k, v, right))
```

```
let rec remove dict key =
  match dict with
  | Empty -> Empty
  | Node (left, k, v, right) when key < k ->
    Node (remove left key, k, v, right)
  | Node (left, k, v, right) when key > k ->
    Node (left, k, v, remove right key)
  | Node (Empty, k, v, child) | Node (child, k, v, Empty) ->
    child
  | Node (left, k, v, right) ->
    let km, vm, right' = extract_min right in
    Node (left, km, vm, right')
```

Correction de l'exercice XXVII.7 page 3

1. C'est très simple avec ce que l'on a écrit :

```
let get_occurrences ms x =
  match get ms x with
  | None -> 0
  | Some i -> i

let add_occurence ms x =
  let i = get_occurrences ms x in
  set ms x (i + 1)

let rem_occurrence ms x =
  let i = get_occurrences ms x in
  if i = 1 then remove ms x
  else set ms x (i - 1)
```

2. `get_occurrences` a la même complexité que `get`, c'est-à-dire $O(h)$. `add_occurrence` fait un appel à `get_occurrence` et un appel à `set`, tous deux en $O(h)$, donc est à nouveau en $O(h)$. Finalement, `rem_occurrence` fait un appel à `get_occurrence` puis soit un appel à `remove` soit un appel à `set`, donc toujours du $O(h)$.

Correction de l'exercice XXVII.8 page 3

1. Il suffit de parcourir l'arbre en sommant les nombres d'occurrences, pour une complexité en $O(n)$:

```
let rec size = function
  | Empty -> 0
  | Node (left, _, i, right) ->
    i + size left + size right
```

2. Il n'y a rien de très satisfaisant pour `select` avec cette structure de données. Une possibilité est de commencer par convertir l'arbre en une liste de couple $(x, occ(x))$ classée par x croissants (grâce à un parcours infixe), en un temps $O(n)$. Ensuite, on parcourt cette liste en sommant les occurrences jusqu'à dépasser l'indice souhaité, ce qui se fait à nouveau en $O(n)$ (dans le pire des cas).

Correction de l'exercice XXVII.9 page 4

1. Pour `get_occurrences` et `add_occurrence`, on adapte `get` et `set` en mettant à jour les tailles :

```
let rec get_occurrences ms x =
  match ms with
  | E -> 0
  | N (_, left, y, i, right) ->
    if x = y then i
    else if x < y then get_occurrences left x
    else get_occurrences right x

let rec add_occurrence ms x =
  match ms with
  | E -> N (1, E, x, 1, E)
  | N (n, left, y, i, right) ->
    if x = y then N (n + 1, left, x, i + 1, right)
    else if x < y then N (n + 1, add_occurrence left x, y, i, right)
    else N (n + 1, left, y, i, add_occurrence right x)
```

Pour `rem_occurrence`, il faut faire attention : on ne peut pas savoir si les tailles doivent être modifiées avant de savoir s'il y a une occurrence à supprimer. Le plus simple est de faire deux parcours : un pour savoir si l'élément est présent, et, au besoin, un pour supprimer une occurrence (en sachant que les tailles de tous les sous-arbres rencontrés doivent être diminuées de une unité).

```
let rec extract_min ms =
  match ms with
  | E -> failwith "minimum of empty multiset"
  | N (_, E, x, i, right) ->
    (x, i, right)
  | N (n, left, x, i, right) ->
    let m, i_m, left' = extract_min left in
    (m, i_m, N (n - i_m, left', x, i, right))


let rec rem_occurrence_aux ms x =
  match ms with
  | E -> E
  | N (n, left, y, i, right) when x < y ->
    N (n - 1, rem_occurrence_aux left x, y, i, right)
  | N (n, left, y, i, right) when x > y ->
    N (n - 1, left, y, i, rem_occurrence_aux right x)
  | N (_, E, y, 1, ms') | N (_, ms', y, 1, E) -> ms'
  | N (n, left, y, 1, right) ->
    let m, i_m, right' = extract_min right in
    N (n - 1, left, m, i_m, right')
  | N (n, left, y, i, right) -> N (n - 1, left, y, i - 1, right)

let rem_occurrence ms x =
  if get_occurrences ms x = 0 then ms
  else rem_occurrence_aux ms x
```

2. Les fonctions `size` et `select` s'écrivent très facilement :

```ocaml
let size = function
  | E -> 0
  | N (n, _, _, _, _) -> n

let rec select ms index =
  match ms with
  | E -> failwith "invalid index"
  | N (n, left, x, i, right) ->
    if index < size left then select left index
    else if index < size left + i then x
    else select right (index - size left - i)
```

`size` est bien évidemment en $O(1)$, et pour `select` on ne parcourt qu'une branche de l'arbre, avec des opérations en temps constant (y compris les appels à `size`) à chaque nœud traversé : la complexité est en $O(h)$).