TD #13 — ABRs (avec solutions)

Exercice 1. *Cas de tests*

On se dote du `type 'a tree = E | N of 'a tree * 'a * 'a tree` pour représenter des arbres binaires en OCaml.

1. Construisez quatre valeurs `t1`, `t2`, `t3`, `t4` de type `'a tree`, et indiquez pour chacune s'il s'agit d'un arbre binaire de recherche ou non. Vous devez avoir au moins une valeur qui est un ABR et une qui ne l'est pas, et au moins un arbre de hauteur deux.

Par exemple :

```
let t0 = N (E, 0, E)    (* abr *)
let t3 = N (E, 3, 1)    (* abr *)
let t2 = N (t0, 2, t3) (* abr *)
let t4 = N (t3, 4, t0) (* pas abr *)
```

Exercice 2. `mem` & `find`*s*

1. Rappelez le principe de la recherche d'élément dans un arbre binaire de recherche, et en quoi elle peut être significativement plus efficace que pour un arbre binaire quelconque.

Pour un arbre binaire quelconque, la recherche d'un élément se fait typiquement par un parcours, par exemple en profondeur. À chaque nœud, ce parcours doit (dans le pire cas) explorer les deux sous-arbres. Si l'arbre est un ABR, on peut en revanche s'économiser le parcours d'un des deux sous-arbres : en effet, si l'élément recherché est plus grand (resp. plus petit) que l'étiquette du nœud courant, la structure d'ABR garantit qu'elle ne se trouvera pas dans le sous-arbre de gauche (resp. droite) de ce nœud, puisque celui-ci ne contient que des éléments plus petits (resp. plus grands). Ainsi, le nombre de nœuds visités est majoré par la hauteur de l'arbre plutôt que sa taille comme dans le cas non structuré. La hauteur d'un arbre pouvant être logarithmique en sa taille, le gain en performance peut être exponentiel, ce qui est significatif.

2. Écrivez une telle fonction OCaml `mem : 'a -> 'a tree -> bool` pour un ABR représenté par le même type qu'à l'exercice précédent.

On propose :

```
let rec mem x
    = function
    | E -> false
    | N (_A, b, _C)
        -> if b < x then mem x _C else
           if b > x then mem x _A else
           true
```

On souhaite maintenant écrire des fonctions `find` pour des ABRs qui se distinguent de `mem` en ce qu'elles ne testent pas la présence d'un élément dans mais recherchent un ou plusieurs éléments particuliers.

3. Écrivez une fonction OCaml `find_min : 'a tree -> 'a option` qui s'évalue en `None` sur un argument `E`, et sinon en son élément minimum.

On propose :

```
let rec find_min
    = function
    | E -> None
    | N (E, b, _) -> Some b
    | N (_A, _, _) -> find_min _A
```

4. Écrivez une fonction OCaml :
   `find_nearest_greater : 'a -> 'a tree -> 'a option`

telle que `find_nearest_greater x t` s'évalue à `None` si `x` est plus grand ou égal (dans le cas d'un ABR représentant un multiensemble) à l'élément maximal se trouvant dans `t`, et sinon à `Some v` avec `v` la plus petite valeur présente dans `t` qui soit strictement supérieure à `x` (pour l'ordre usuel). Son coût devra être linéaire en la hauteur de l'arbre.
Remarque : cette fonction n'est pas très longue mais un peu fine. Réfléchissez bien.

Une approche qui fonctionne bien est de propager une solution candidate en descendant l'arbre toujours d'un seul côté. Si le nœud courant est plus grand que `x` c'est une solution candidate, mais il pourrait y en avoir une autre meilleure dans son sous-arbre gauche. Si elle est trop petite, ce n'est pas une solution candidate mais on en trouvera peut-être une dans son sous-arbre droit. Ceci donne par exemple :

```
let find_nearest_greater x t =
    let rec fng cand
        = function
        | E -> cand
        | N (_A, b, _C)
            -> if b > x then
               fng (Some b) _A else
               fng cand _C
    in fng None t
```

5. Écrivez de même une fonction OCaml :
`find_in_between : 'a -> 'a -> 'a tree -> 'a list`
telle que `find_in_between lo hi t` s'évalue en la liste ordonnée croissante des valeurs de `t` se trouvant dans l'intervalle $[\![\texttt{lo}, \texttt{hi}]\!]$. Son coût devra être un $O(h+\ell)$, avec h la hauteur de t et ℓ la longueur de son résultat.
Indice : c'est essentiellement une combinaison de la logique de la fonction précédente avec celle d'un parcours en profondeur infixe.

On propose :

```
let find_in_between lo hi t =
    let rec fib acc
        = function
        | E -> acc
        | N (_A, b, _C)
            -> if b > hi then fib acc _A else
               if b < lo then fib acc _C else
               fib (b::(fib acc _C)) _A
    in fib [] t
```

Exercice 3. *Correction de* `del`

On donne la fonction OCaml suivante :

```
let rec pop_min
    = function
    | E -> invalid_arg "pop_min: empty tree"
    | N (E, b, _C)  -> b, _C
    | N (_A, b, _C) -> let m, _A' = pop_min _A in
                       m, N (_A', b, _C)
```

1. Montrez que si `t` est un ABR non vide, `pop_min t` termine et s'évalue en `m, t'` où `m` est l'élément de `t` de valeur (d'étiquette) minimale et `t'` un ABR de mêmes éléments que `t` moins celui de valeur (d'étiquette) `m`. On considère ici des ABRs représentant des ensembles ; e toute rigueur il faudrait formaliser ce que l'on entend par là, typiquement en définissant une fonction donnant les éléments présents dans un ABR.

On prouve la terminaison en montrant que la hauteur de l'argument est un variant récursif.
Minoration. Par sa définition, la hauteur d'un argument est minorée par -1 (et celle d'un argument valide est même minorée par 0).
Stricte décroissance. Un appel `pop_min t` engendre un appel récursif uniquement sur `_A` quand `t` est de la forme `N (_A, _, _)`, c'est à dire sur un sous-arbre de `t`. Par définition de la hauteur, celle de`_A` est au plus celle de `t` moins un.

Par hypothèse l'argument `t` est non vide et est donc égal à `N (_A, b, _C)` pour certains `_A`, `b`, `_C`. On va prouver la correction partielle par récurrence sur la hauteur gauche de l'argument, définie comme :

```
let rec lefth = function E -> -1 | N (_L, _, _) -> 1 + lefth _L
```

Cas de base. On considère le cas d'une hauteur gauche de 0. Dans ce cas `_A = E`, et par la ligne 4 la fonction s'évalue en `b, _C`. Par hypothèse que `t` est un ABR, `b` en est l'élément de valeur minimale et `_C` est un ABR. De plus, les éléments de `t` sont donnés par `b` et ceux de `_C`, et `_C` est donc bien un ABR de mêmes éléments que `t` moins `b`, et la fonction est correcte.
Cas récursif. On suppose `pop_min` correcte pour les arbres de hauteur gauche $h > 0$, et l'on montre qu'elle est correcte pour ceux de hauteur gauche $h+1$. Par définition de cette hauteur, un tel arbre est égal à `N (_A, b, _C)` pour `_A` un arbre de hauteur gauche $h > 0$; notamment, `_A` n'est pas vide.
Puisque `_A` n'est pas vide, alors par la ligne 5 la fonction s'évalue en `m, N (_A', b, _C)` avec `m, _A'` le résultat de `pop_min _A`.
Par hypothèse que `t` est un ABR, les éléments de `_A` sont tous strictement inférieurs à `b` et à ceux de `_C`, et puisque `_A` est non vide son élément minimum est l'élément minimum de `_A`. De plus, ses éléments sont donnés par l'union de ceux de `_A`, ceux de `_C` et `b`.
Par hypothèse de récurrence, `m` et `_A'` sont respectivement l'élément minimum de `_A` et un ABR d'éléments égaux à ceux de `_A` privé de `m`, qui (par la remarque ci-dessus) sont (toujours) strictement inférieurs à `b` et ceux de `_C`. Ainsi, `m` est bien l'élément minimum de `t`, et `N (_A', b, C)` un ABR de mêmes éléments que `t`, privés de `m`, et la propriété est conservée.

2. Écrivez une fonction OCaml `del : 'a -> 'a tree -> 'a tree` telle que pour un ABR `t`, `del x t` s'évalue en un ABR de mêmes éléments que `t` moins celui de valeur (d'étiquette) `x` si `t` en possède un, et `t` sinon.

On propose :

```
1  let rec del x
2      = function
3      | E -> E
4      | N (_A, b, _C)
5          -> if b < x then N (_A, b, (del x _C)) else
6             if b > x then N ((del x _A), b, _C) else
7             (* b = x *)
8             if _C = E then _A else
9             if _A = E then _C else
10            let b', _C' = pop_min _C in N (_A, b', _C')
```

3. Prouvez la correction partielle de votre fonction `del`.

On procède de façon similaire à la preuve de correction partielle de `pop_min`, mais en procédant cette fois par récurrence forte sur la hauteur de l'argument `t`.
Cas de base. Il y a quatre cas à considérer, qui correspondent aux lignes 3, 8, 9, 10 de la proposition de correction ci-dessus :

- `t = E`. Alors `x` n'apparaît pas dans `t` et il est correct de s'évaluer en `E = t`.
- `t = N (_A, x, E)` pour un certain `_A`. Alors par hypothèse que `t` est un ABR `_A` en est un également. De plus, les éléments de `t` sont donnés par l'union de ceux de `_A` et `x`, et donc il est correct de s'évaluer en `_A`.
- `t = N (E, x, _C)` pour un certain `_C`. Symétrique du cas précédent.
- `t = N (_A, x, _C)` pour certains `_A`, `_C` non vides. Alors par correction de `pop_min`, soit `b', _C' = pop_min _C`, on a que `N (_A, b', _C')` est un arbre binaire dont les éléments sont les mêmes que `t` moins `x` ; il reste à montrer que c'est aussi un ABR. Par hypothèse que `t` est un ABR, les éléments de `_C` sont tous strictement supérieurs à ceux de `_A`. Par correction de `pop_min`, `b'` est strictement supérieur à tous les éléments de `_A` (puisqu'élément de `_C`), et strictement inférieur à tous ceux de `_C'`, et `N (_A, b', _C')` est bien un ABR.

Cas récursifs. On suppose `del` correcte pour tous les arbres de hauteur $\leq h$, et l'on va montrer qu'elle est également correcte pour les arbres de hauteur $h+1$. Il y a deux cas à considérer, qui correspondent aux lignes 5 et 6 de la proposition de correction ci-dessus.

- `t = N (_A, b, _C)` pour certains `_A`, `b`, `_C` avec `b < x`. Alors par le fait que `t` est un ABR, si `x` y apparaît il se trouve nécessairement dans le sous-arbre droit `_C`. Par définition de la hauteur, celle de `_C` est strictement inférieure à celle de `t`, et donc au plus égale à h. Par hypothèse de récurrence forte, on a donc que `del x _C` est un ABR de mêmes éléments que `_C` dont on a éventuellement retiré `x`, et il s'ensuit que

`N (_A, b, (del x _C))` est bien un ABR de mêmes éléments que `t` dont on a éventuellement retiré `x`. La propriété est conservée.

— `t = N (_A, b, _C)` pour certains `_A`, `b`, `_C` avec `b > x`. Symétrique du cas précédent.

Exercice 4. `of_list`

Dans cet exercice, on souhaite construire des ABRs à partir de `'a list` d'éléments supposés tous distincts. On utilise un `type 'a tree = E | N of 'a tree * 'a * 'a tree` pour représenter des ABRs fonctionnels, et suppose implémentée une fonction `add : 'a -> 'a tree -> 'a tree` d'insertion dans un ABR.

1. Écrivez une fonction OCaml `of_list : 'a list -> 'a tree` la plus simple possible, qui construit un ABR de mêmes éléments que son argument.

On propose :
```
let rec of_list
    = function
    | [] -> E
    | x::xs -> add x (of_list xs)
```
ou
```
let of_list' x = List.fold_right add x E
```
ou
```
let of_list'' x = List.fold_left (Fun.flip add) E x
```

2. Dessinez l'ABR construit par une telle fonction `of_list` pour l'argument `[0; 1; 2; 3; 4]`, en supposant que `add` est appelée sur les éléments de la liste dans l'ordre (de la tête vers la queue ou inversement), et qu'aucun équilibrage n'est effectué. Déduisez-en un coût pire-cas pour une telle fonction `of_list`, et un majorant de la hauteur des arbres qu'elle construit.

L'arbre obtenu est un peigne de hauteur égale à la longueur de la liste (éventuellement moins un), ce qui donne un coût pour `of_list` quadratique en la longueur de son argument. Ce dernier coût est bien un pire cas, puisque la suite des hauteurs des arbres intermédiaires construits est elle-même un pire cas (la hauteur augmente de un à chaque appel à `add`, et il n'est pas possible de faire plus). (La fonction atteint donc ici le pire cas possible atteignable par *n'importe quelle* fonction `of_list`.)

3. Proposez une approche diviser-pour-régner qui permette de construire un ABR que l'on argumentera être équilibré depuis une liste de valeur triée (pour le même ordre que celui de l'ABR). Analysez son coût.

On propose l'approche suivante : pour construire l'ABR correspondant à une certaine liste `x` (non vide) de longueur n, on décompose `x` comme `xhi @ [xmid] @ xlo` avec `xhi` et `xlo` de taille $\lceil (n-1)/2 \rfloor$, puis l'on construit récursivement des ABRs `bhi` et `blo` d'éléments respectivement `xhi` et `xlo`, et l'on construit notre résultat comme `N (bhi, xmid, blo)`. Celui ci est bien un ABR par hypothèse sur `bhi` et `blo` et que `x` est triée, et il est équilibré si `bhi` et `blo` le sont, ce que l'on garantit récursivement. L'argument ici est que l'on garantit récursivement qu'à chaque nœud `bhi` et `blo` ont (essentiellement) la même *taille*. On obtient alors des arbres « de poids équilibré » (*weight-balanced*), qui sont équilibrés (admis).
Le coût de cette fonction est donné par la somme de celui de la décomposition de `x` et des appels récursifs, et satisfait donc (à plus-ou-moins un) la récurrence $T(n) = 2T(n/2) + O(n)$; c'est donc un $O(n \log n)$.
Ce coût n'a aucune raison d'être optimal (le minorant trivial étant $\Omega(n)$), mais nous nous en contenteront ici.

4. Écrivez une fonction OCaml `of_list2 : 'a list -> 'a tree` qui implémente cette approche.

On propose :
```
let split x =
    let rec _split n xhi
        = function
        | [] -> assert false
        | x::xs -> if n = 0 then
                    xhi, x, xs else
                    _split (n - 1) (x::xhi) xs
    in
```

```
    let n = List.length x in
    let mid = (n - 1) / 2 in
    let xhi, xmid, xlo = _split mid [] x in
    List.rev xhi, xmid, xlo

let rec of_list2 x =
    if x = [] then E else
    let xhi, xmid, xlo = split x in
    let bhi = of_list2 xhi in
    let blo = of_list2 xlo in
    N (bhi, xmid, blo)
```

5. Commentez l'intérêt éventuel de cette fonction `of_list2` construisant un ABR équilibré depuis une liste triée `x` si l'on s'attend ensuite à devoir de nombreuses fois rechercher si certains éléments sont présents dans `x`, en fonction de si l'on dispose également ou non d'une structure de données de tableau efficace.

C'est assez inutile dans le premier cas où l'on disposerait d'une structure de données de tableaux efficaces, puisqu'il suffirait alors de convertir la liste en un tableau et d'ensuite effectuer des recherches dichotomiques classiques. Dans le second cas, la structure d'ABR est intéressante puisqu'elle permet justement une recherche dichotomique efficace, impossible à réaliser sur des listes. Cette approche est alors une alternative simple à l'utilisation d'ABRs auto-équilibrés, notamment bien adaptée au cas où l'arbre n'est pas modifié après sa construction.

6. Commentez le coût dans le cas où la liste n'est pas initialement triée.

Si la liste n'est pas initialement triée, on peut simplement se ramener à ce cas en commençant par la trier… On peut faire cela pour un coût $O(n \log n)$ (avec n la longueur de la liste) en utilisant un tri par comparaison asymptotiquement optimal, ce qui n'augmente pas le coût global de la construction puisque c'est déjà le coût de `of_list2`. On peut remarquer que réciproquement, il n'est pas *essentiel* d'avoir une variante de `of_list` de coût linéaire en la longueur de son argument si sa seule utilisation l'est comme ci-dessus.