TD #13 — ABRs

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.

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.
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 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.
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.
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.

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

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.
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.
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.
4. Écrivez une fonction OCaml `of_list2 : 'a list -> 'a tree` qui implémente cette approche.
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.
6. Commentez le coût dans le cas où la liste n'est pas initialement triée.