TD #14 — Arbres n-aires, re:ABRs

Exercice 1. *Fonctions de base sur arbres n-aires*

Dans tout cet exercice, on utilisera le :
```
type 'a ntree = N of 'a * 'a ntree list
```
pour représenter des arbres n-aires en OCaml.

1. Écrivez une fonction OCaml `size : 'a ntree -> int` qui s'évalue en la taille de son argument, définie comme le nombre de nœuds construits avec `N`.
2. Écrivez une fonction OCaml `height : 'a ntree -> int` qui s'évalue en la hauteur de son argument, définie récursivement par le fait que la hauteur d'un nœud vaut un de plus que la hauteur maximale de ses enfants (où l'on prendra la convention que le maximum sur l'ensemble vide vaut zéro).
3. Écrivez une fonction OCaml :
   ```
   dfs_ante : ('a -> unit) -> 'a ntree -> unit
   ```
   qui applique son premier argument aux étiquettes des nœuds de son second dans un parcours en profondeur préfixe.
4. Écrivez une fonction OCaml
   ```
   dfs_post : ('a -> unit) -> 'a ntree -> unit
   ```
   qui fait de même pour un parcours en profondeur postfixe.

Exercice 2. *Algorithmes d'ABRs à base de* join

Dans tout le début de cet exercice, on utilisera le :
```
type 'a tree = N of 'a tree * 'a * 'a tree | E
```
pour représenter des arbres binaires en OCaml. Sauf mention du contraire, ces arbres binaires seront supposés être « de recherche », et ne jamais contenir plusieurs nœuds de même étiquette.

Le but de cet exercice est de concevoir des algorithmes pour ABR équilibrés, dont l'opération d'équilibrage est abstraite par une unique fonction *join*. Celle ci satisfait les spécifications suivantes : `join t1 k t2` construit et s'évalue en un arbre dont le parcours en profondeur infixe énumère les valeurs de `t1` dans l'ordre de son parcours en profondeur infixe, puis `k`, puis les valeurs de `t2` dans l'ordre etc. De plus, si `t1` et `t2` sont équilibrés au sens d'un certain schéma d'équilibrage, alors le résultat est équilibré pour le même schéma.

1. En utilisant uniquement la fonction `join` et les manipulations de base du type `'a tree`, écrivez une fonction OCaml :
   ```
   split : 'a -> 'a tree ->  'a tree * bool * 'a tree
   ```
   telle que `split k t` s'évalue en `less, b, more` où `b` vaut `true` ssi. `k` est

présent dans `t`, et `less` (resp. `more`) est un ABR dont les éléments sont ceux de `t` strictement inférieurs (resp. supérieurs) à `k`.

2. Toujours avec les mêmes contraintes, écrivez une fonction OCaml
   `splitlast : 'a tree -> 'a tree * 'a`
   telle que pour un argument non vide, `splitlast t` s'évalue en `t', k` où `k` est l'étiquette du nœud « le plus à droite » de `t` (qui est donc de valeur maximale pour les éléments de `t`), et `t'` un arbre d'éléments de mêmes étiquettes que `t`, moins `k`.
3. Déduisez de ce qui précède une fonction OCaml :
   `join2 : 'a tree -> 'a tree -> 'a tree`
   identique à `join` si ce n'est pour l'absence du paramètre `k:'a`.
4. Toujours avec les mêmes contraintes, écrivez une fonction OCaml `add : 'a -> 'a tree -> 'a tree` d'insertion dans un ABR. Le résultat devra être équilibré relativement au schéma implémenté par `join`.
5. De même pour une fonction OCaml `del : 'a -> 'a tree -> 'a tree` de suppression dans un ABR.

On s'intéresse maintenant à une implémentation de `join` pour l'équilibrage AVL. Pour cela, on représente maintenant les arbres binaires par le :

```
type 'a tree = N of int * 'a tree * 'a * 'a tree | E
```

où le constructeur `N` prend un premier argument supplémentaire désignant la hauteur de l'arbre qu'il construit. On supposera également implémenté un constructeur `node : 'a tree -> 'a -> 'a tree -> 'a tree` semi-intelligent qui maintient à jour les informations de hauteur.

6. Écrivez deux fonctions OCaml `rotate_left` et `rotate_right` toutes deux de type `'a tree -> 'a tree` qui effectuent respectivement une rotation à gauche et à droite de leur argument.

La fonction `join` repose sur deux fonctions symétriques `join_r` et `join_l`, qui effectuent les équilibrages nécessaires quand `t1` ou `t2` est trop haut par rapport à l'autre. On va se concentrer sur l'écriture de `join_r` pour d'équilibrage AVL, qui est de mêmes spécifications que `join` si ce n'est que `t1` et `t2` sont équilibrés AVL t.q. les hauteurs h_1 et h_2 de `t1` et `t2` satisfont $h_2 < h_1 - 1$, et que le résultat est de hauteur $\in [\![h_1, h_1 + 1]\!]$. Le principe de `join_r` sur des arguments t_1, k, t_2 est le suivant : on déconstruit t_1 (nécessairement non nul) comme $\langle A, b, C\rangle$, et 1) si C est d'une hauteur t.q. $C' := \langle C, k, t_2\rangle$ soit équilibré, on renvoie $\rho\langle A, b, C'\rangle$, où ρ effectue au besoin une (double) rotation quand C' est trop haut par rapport à A (on peut montrer que ce dernier cas se produit uniquement si $h_C = h_A + 1$) ; 2) sinon on appelle `join_r` récursivement sur C, k et t_2 obtenant C', et l'on renvoie $\rho\langle A, b, C'\rangle$, où ρ effectue au besoin une unique rotation.

7. Écrivez une fonction OCaml :
   `join_r : 'a tree -> 'a -> 'a tree -> 'a tree`
   suivant le principe ci-dessus.