TRAVAUX PRATIQUES XV

Introduction aux arbres

Dans ce TP, les questions marquées *(Bonus)* sont à réserver pour la fin du TP ou si vous êtes en avance. Sinon, priorisez de passer à la suite pour avoir le temps d'explorer un peu chaque partie.

Les questions ou indications (en italique) marquées d'un *(Aide)* sont là pour vous aider et alléger la difficulté. Si vous voulez vous entraîner, sautez ces questions pour faire directement les suivantes, ou ne lisez pas ces indications.

A En OCaml

A.1 Arbres binaires (non stricts) :

Dans cette partie, on représente les arbres binaires (non stricts) avec le type OCaml suivant :

OCaml
```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree
```

Voici un arbre exemple pour vous aider à tester :

OCaml
```
let arbre_exemple = Node(42, Node(3, Node(5, Nil, Nil),Node(7, Nil, Nil)), Node(24, Nil, Nil))
```

Il correspond à l'arbre suivant :

Terminal
```
arbre_exemple :
      42
     / \
    3  24
   / \
  5  7
```

Bien entendu, vous devez faire **vos propres tests** entre chaque fonction, comme d'habitude !

A.1.i : Généralités

1. Écrire une fonction `hauteur : 'a tree -> int` qui calcule la hauteur d'un arbre.
2. Écrire une fonction `taille : 'a tree -> int` qui calcule le nombre total de noeuds d'un arbre.
3. Écrire une fonction `somme : int tree -> int` qui calcule la somme des étiquettes des noeuds d'un arbre étiquetté par des entiers.

A.1.ii : Affichage bien dans un sens précis

4. Écrire une fonction `affiche_prefixe` qui affiche toutes les étyiquettes des noeuds d'un arbre d'entiers de la façon suivante : d'abord il affiche l'étiquette du noeud en cours, ensuite tous les noeuds de son sous-arbre gauche, enf tous ceux de son sous-arbre droit.
5. Écrire une fonction `affiche_prefixe` qui fait de même mais dans l'ordre gauche, en cours, droite.
6. Écrire une fonction `affiche_postfixe` qui fait de même mais dans l'ordre gauche, droite, en cours.

A.1.iii : Arbres particuliers

7. Écrire une fonction `peigne_gauche : int -> int tree` prenant en entrée un entier n et renvoyant un peigne gauche à n nœuds internes dont les étiquettes, lues dans l'ordre préfixe (c'est à dire ici de gauche à droite, ou encore de bas en haut), forment la liste `[n; . . . ; 1]`. *La racine vaut n ici.*
8. *(Bonus)* Écrire une fonction `peigne_droit : int -> int tree` prenant en entrée un entier n et renvoyant un peigne droit à n nœuds internes dont les étiquettes, lues dans l'ordre préfixe (c'est à dire ici de gauche à droite, ou encore de haut en bas), forment la liste `[1; . . . ; n]`. *Attention, la racine vaut 1 ici.*
9. Écrire une fonction `parfait : int -> int tree` prenant en entrée un entier h et renvoyant un arbre parfait à $2^{h+1} - 1$ nœuds dont les nœuds situés à profondeur k portent l'étiquette $h - k$.
10. Écrire une fonction `est_strict : 'a tree -> bool` qui teste si un arbre binaire est strict.

A.2 Arbres binaires stricts

Dans cette partie, on manipule une définition légèrement différente, adaptée aux arbres stricts. On suppose ici que les feuilles et les noeuds internes sont étiquetés par le même ensemble.

OCaml
```
type 'a strict = Feuille of 'a | Noeud of 'a * 'a strict * 'a strict
```

11. Rappeler ce qu'est un arbre strict. Cette définition est-elle correcte ?
12. Recoder la fonction `hauteur` sur ces arbres. On appelera cette fonction `hauteur_stricte` .

A.2.i : Autour du nombre de noeuds

13. Écrire deux fonctions `nb_feuilles` et `nb_noeuds_internes : 'a strict -> int` qui calculent le nombre de feuilles de l'arbre et son nombre de noeuds internes, respectivement.
14. *(Bonus)* Vérifier sur quelques exemples que le nombre de feuilles `f` d'un arbre binaire strict vérifie $f = n_i + 1$ où n_i est le nombre de noeuds internes de l'arbre.

A.2.ii : Autour de la hauteur

15. Écrire une fonction `profondeur_min : 'a strict -> int` qui calcule la profondeur minimale d'une feuille de l'arbre.
16. Écrire une fonction `diff_max : 'a strict -> int` qui calcule la différence maximale entre la profondeur de deux feuilles de l'arbre.
17. *(Bonus)* Écrire une fonction `feuille_basse : 'a strict -> int` qui renvoie l'étiquette de la feuille de l'arbre située le plus à gauche, parmi celles de profondeur maximale. *On essaiera d'écrire une fonction efficace.*
18. *(Bonus)* Écrire une fonction `arbre_hauteurs : 'a strict -> int tree` qui prend en entrée un arbre t et renvoie un arbre ayant exactement la même forme que t, mais dans lequel l'étiquette de chaque nœud a été remplacée par la hauteur du sous-arbre correspondant. *On essaiera d'écrire une fonction efficace.*

B Pour aller plus loin : Arbres d'arité quelconque

Dans cette partie, on étudie des arbres dont les noeuds peuvent avoir un nombre arbitraire d'enfants :

OCaml
```
type 'a arbre = Nil_k | Node_k of 'a * 'a arbre list
```

Si l'on omet le « _k », OCaml ne saurait pas comment typer `Nil`. Plus précisément, il utiliserait le dernier `Nil` défini. Il y a plusieurs façons d'éviter ce problème ; ici on fait au plus simple : au lieu de créer un nouveau `Nil` on crée `Nil_k`.

19. En utilisant `List.map` , écrire une fonction qui prend en argument une liste d'arbres non vides et renvoie la liste des racines des arbres.
    Pour vous aider, vous pouvez définir à part une fonction `racine` qui prend en argument un arbre et renvoie sa racine.
20. Écrire une fonction `hauteur_arbre : 'a arbre -> int` qui calcule la hauteur d'un arbre d'arité quelconque.
    (Aide) : On définira une fonction auxiliaire `max_liste` qui calcule le maximum des éléments d'une liste, et on pourra faire appel à la fonction `List.map` pour construire une liste bien choisie.

21. *(Bonus)* Reprendre la question précédente en utilisant un `fold_left` pour coder `max_liste` .

22. Écrire une fonction `transfo_LCRS : 'a arbre -> 'a tree` qui applique la transformation LCRS à un arbre d'arité quelconque
(Aide) : Vous pourrez faire une fonction auxiliaire récursive qui garde en argument le noeud visité et ses frères droits.