TD : Exercices sur les arbres

1 Arbres binaires

Exercice 1 : Exercice

On considère des arbres binaires de type :

```
type 'a strict =
  | F of 'a
  | N of 'a * 'a strict * 'a strict
```

1. Écrire une fonction `profondeur_min : 'a strict -> int` qui renvoie la profondeur minimale d'une feuille de l'arbre.
2. Écrire une fonction `diff_max : 'a strict -> int` qui renvoie la différence maximale entre la profondeur de deux feuilles de l'arbre.
3. Écrire une fonction `feuille_basse : 'a strict -> 'a` 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.
4. Écrire une fonction `arbre_hauteurs : 'a strict -> int strict` 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.

Exercice 2 : Exercice

On s'intéresse à des arbres ayant pour type :

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

— On définit la *taille* d'un arbre comme son nombre total de nœuds (non vides).
— Un nœud `N (x, g, d)` est dit *lourd* si la taille de d est strictement supérieure à celle de g, *léger* sinon.
— Un nœud vide `E` est considéré comme léger.
— Le *poids* d'un arbre est défini comme son nombre de nœuds lourds.

1. Écrire une fonction `taille : 'a binaire -> int`.
2. En utilisant cette fonction, écrire une fonction `poids : 'a binaire -> int` (la plus simple possible).
3. Donner un exemple d'arbre de taille n pour lequel `poids` s'exécute en temps quadratique.
4. Proposer une version plus efficace de `poids`.

Exercice 3 : Constructions d'arbres

On utilise à nouveau le type `binaire` de l'exercice précédent.
— *peigne gauche* un arbre dont tous les nœuds internes sont de la forme `N(x, gauche, V)` ;
— *peigne droit* un arbre dont tous les nœuds internes sont de la forme `N(x, V, droit)` ;
— *arbre complet* un arbre dont toutes les feuilles sont à profondeur n ou $n-1$ pour un certain n (on ne demande pas que le dernier niveau soit rempli de gauche à droite) ;
— *arbre parfait* un arbre dont toutes les feuilles sont à profondeur n pour un certain n.

1. Écrire une fonction `peigne_gauche : int -> int binaire` prenant en entrée un entier n et renvoyant un peigne gauche à n nœuds internes dont les étiquettes, lues dans l'ordre préfixe, forment la liste $[n, \dots, 1]$.
2. Écrire une fonction `peigne_droit : int -> int binaire` avec la même spécification (mais renvoyant un peigne droit).
3. Écrire une fonction `complet : int -> int binaire` ayant encore la même spécification (mais renvoyant un arbre complet).

4. Écrire une fonction `parfait : int -> int binaire` renvoyant un arbre parfait à $2^n - 1$ nœuds internes dont les nœuds situés à profondeur k portent l'étiquette $n - k$.
5. Quelle est la complexité temporelle de la fonction `parfait` ? Combien d'espace l'arbre renvoyé occupe-t-il en mémoire ?

Exercice 4 : Parcours en largeur et reconstruction

On garde toujours le type `binaire` et l'on s'intéresse cette fois au parcours en largeur.

1. Écrire une fonction `liste_largeur : 'a binaire -> 'a list` renvoyant la liste des étiquettes d'un arbre, dans l'ordre du parcours en largeur.
   Indication : on pourra utiliser le module `Queue` et définir une fonction auxiliaire de type

   `'a binaire Queue.t -> 'a list`

   réalisant le parcours en largeur d'une forêt (d'abord toutes les racines, puis tous les nœuds situés à profondeur 1 dans les différents arbres, puis...).
2. Écrire une fonction `numerote_largeur : 'a binaire -> ('a * int) arbre` renvoyant un arbre ayant exactement la même forme que celui passé en argument mais dans lequel on a adjoint à chaque étiquette le numéro du nœud correspondant dans le parcours en largeur. *Cette question est assez difficile (même si le code attendu est très court).*

Exercice 5 : Arbres binaires lourds

On fixe le type d'arbre binaire suivant :

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

1. Écrire une fonction `hauteur : 'a binary -> int` calculant la hauteur d'un arbre binaire.
2. Écrire une fonction `taille : 'a binary -> int` renvoyant le nombre de nœuds d'un arbre binaire.

On dit qu'un nœud non vide $N(x, g, d)$ d'un arbre binaire est *lourd* si la taille de d est strictement supérieure à celle de g. On considère qu'un arbre vide est léger (i.e. non lourd). Le *poids* d'un arbre est alors son nombre de nœuds lourds.

3. Déduire de la fonction `taille` une fonction calculant le poids d'un arbre binaire : `poids_naive : 'a tree -> int`.
4. *(au tableau)* Pour $n \in \mathbb{N}$, proposer un arbre de taille n tel que la fonction `poids_naive` a une complexité quadratique.
5. Améliorer la complexité.

Exercice 6 : Exercice

On considère les deux types suivants :

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

type 'a binaire_annote =
  | Va
  | Na of int * 'a * 'a binaire_annote * 'a binaire_annote
```

On définit la taille d'un arbre comme son nombre de nœuds non vide, et l'on note $|t|$ la taille de t. Un arbre de type `'a binaire_annote` sera dit *correctement annoté* s'il est :
— soit de la forme `Va` ;
— soit de la forme `Na (n, x, g, d)` avec $n = |g|$.

1. Écrire une fonction `verifie_annotation` qui détermine si un arbre est correctement annoté.

   ```
   verifie_annotation : 'a binaire_annote -> bool
   ```

2. Écrire une fonction `annote` qui annote (correctement) un arbre binaire.

   ```
   annote : 'a binaire -> 'a binaire_annote
   ```

Exercice 7 : Structures de cordes

On s'intéresse à la structure de *cordes*, qui sert à stocker de très longues chaînes de caractères à l'aide d'arbres. Ici, on se limitera en fait à des listes d'entiers, les chaînes à représenter par des cordes étant donc : `type chaine = int list`. Le type corde est alors :

```
type corde =
  | V
  | F of int * chaine
  | N of int * corde * corde
```

`V` représente un mot vide, `F (k, c)` la chaîne `c` de taille `k > 0` et `N (k, c1, c2)` la concaténation des chaînes représentées par `c1` et `c2` de longueur totale `k > 0` avec `c1 <> V` et `c2 <> V`.

1. Écrire une fonction `longueur_corde : corde -> int` calculant la longueur d'une corde.
2. Écrire une fonction `corde_of_chaine : chaine -> corde` qui prend une chaîne en entrée et renvoie une corde représentant correctement une chaîne.
3. Écrire une fonction `concatener : corde -> corde -> corde` concaténant deux cordes.
4. Écrire une fonction `corde_nth : corde -> int -> int` telle que `corde_nth c n` renvoie le `n`ième caractère de la corde `c`.
5. Déterminer la complexité de la fonction `corde_nth`.
6. Écrire une fonction `sous_corde : corde -> int -> int -> corde` telle que `sous_corde c deb fin` renvoie une corde représentant exactement la chaine représentée par `c`, pour les caractères des indices `deb` inclus à `fin` exclus.

On sent bien que pour que cette structure soit efficace pour stocker chaînes de caractères, il faut s'arranger pour que la "hauteur" de la corde soit la plus petite possible, en gardant en même temps la longueur des mots stockés dans les feuilles petite. Une première idée est de reconfigurer cet arbre pour l'"équilibrer", en insérant successivement les feuilles de la corde dans un tableau `t` initialement rempli de `V`. Supposons que l'on cherche à insérer la corde x dans la case i (pour l'appel pour insérer une feuille, on prendra $i = 1$).

On insère la feuille x_i, de taille $\text{Card}(x_i)$, en la concaténant à droite de la corde contenue dans `t.(i)` : si la longueur de la corde obtenue est toujours comprise entre F_i et F_{i+1} exclus, on la met dans `t.(i)`, sinon on insère la corde obtenue à l'indice `i+1` et on fixe `t.(i)` à `V`.

À la fin, si l'on concatène toutes les cordes contenues dans `t` par indices décroissants, on obtient une corde représentant la même chaîne que la corde initiale.

7. Écrire une fonction `fib_efficace : int -> int array` qui renvoie le tableau des n premières valeurs de la suite de Fibonacci (avec une complexité acceptable). On supposera dans la suite qu'on aura à disposition ce tableau avec $n = 50$.
8. En déduire une fonction `inserer : corde array -> corde -> int -> unit` telle que l'appel `inserer t c i` insère dans le tableau `t` à l'indice `i` la corde `c` en suivant la stratégie ci-dessus.

La corde obtenue avec cette fonction est alors de coût en $\mathrm{O}(n \log n)$, avec n sa longueur.

2 Arbres généraux

Exercice 8 : Exercice

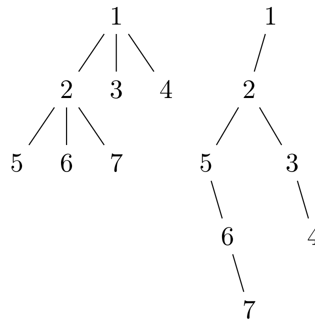
Quelques fonctions sur les arbres généraux, en utilisant le type suivant :

```
type 'a tree = N of 'a * 'a tree list
```

1. Quels arbres ne peut-on pas écrire avec `tree` ?
2. Écrire une fonction `premier_fils : 'a tree -> 'a tree` renvoyant le premier fils d'un arbre (et levant une exception s'il n'existe pas).
3. Écrire les fonctions usuelles sur les arbres : `hauteur : 'a tree -> int`, `taille : 'a tree -> int` et `max_tree : 'a t`

Exercice 9 : Exercice

On peut représenter n'importe quel arbre général par un arbre binaire en utilisant le principe LCRS (*Left child, right sibling*). L'idée est la suivante : chaque nœud contient un lien vers la liste chaînée de ses enfants, dont le premier chaînon éventuel est donc son premier fils. On transforme un arbre général en un arbre binaire en utilisant le principe suivant : pour un nœud n de l'arbre autre que la racine, il est donc présent dans une liste de fils l de son parent : on représente alors n avec comme fils gauche son premier fils, et comme fils droit le nœud suivant dans la liste l. Par exemple, pour l'arbre suivant, on a :



1. Écrire une fonction `general_to_binaire : 'a tree -> 'a binary_tree`, avec un type `binary_tree` bien choisi.
2. Écrire la fonction `binaire_to_general : 'a binary_tree -> 'a tree` faisant la transformation inverse.
3. Que peut-on dire du nombre de nœuds des arbres produits par ces fonctions ? La hauteur ?

Exercice 10 : Arbres généraux

On utilise le type suivant pour représenter des arbres généraux :

```
type 'a general =
  Node of 'a * 'a general list
```

1. Quels arbres ne peut-on pas représenter avec ce type ?
2. Écrire une fonction `nombre_noeuds : 'a general -> int` calculant
3. Écrire une fonction `est_binaire : 'a general -> bool` vérifiant si un arbre général est binaire ou non. Faire de même pour `est_binaire_strict : 'a general -> bool`.
4. Écrire une fonction `vers_binaire : 'a general -> 'a binary` qui renvoie pour un arbre binaire de type `'a general` un arbre binaire de type `'a tree` (et lève une exception si l'arbre n'est pas binaire). On choisira librement si un noeud d'arité 1 doit avoir un fils à gauche ou un fils à droite.
5. Faire de même pour `vers_binaire_strict : 'a general -> 'a strict` pour un type d'arbre binaire strict `'a strict` à choisir.

Exercice 11 : Arbres d'arité quelconque

On considère des arbres d'arité quelconque :

```
type 'a arbre =
  | N of 'a * 'a arbre list
```

1. Écrire une fonction `est_binaire_strict : 'a arbre -> bool` qui détermine si son argument est un arbre binaire strict (dont tous les nœuds sont d'arité 0 ou 2).
2. Écrire une fonction `somme : int arbre -> int` qui renvoie la somme des étiquettes de l'arbre passé en argument.