# Option informatique : TP 6 (Arbres et Piles)

Des fonctions d'aide sont fournies à l'adresse suivante pour certaines questions : `https://who.rocq.inria.fr/Simon.Cruanes/enseignement/tp6_helper.ml`. Le type `arbre` y est également défini. Vous pouvez télécharger le fichier et l'utiliser comme base pour le reste du TP.

## 1 Fonctions d'Arbres

On pourra utiliser la fonction d'aide `afficher_arbre : arbre -> unit` pour afficher les arbres de manière lisible. La fonction `arbre_aleatoire : unit -> arbre` sert à générer aléatoirement des arbres, à des fins de tests.
Écrire les fonctions suivantes :
**(a)** `hauteur a` qui calcule la hauteur d'un arbre
**(b)** `chercher a x` qui renvoie `true` ssi l'élément `x` apparaît dans l'arbre `a`.
**(c)** `taille a` qui calcule le nombre de nœuds de `a`.
**(d)** `infixe a` qui renvoie la liste des nœuds de `a` dans l'ordre infixe (i.e. pour l'arbre `Noeud(x,l,r)` l'élément `x` apparaît dans le résultat après les étiquettes de `l` mais avant celles de `r`).
**(e)** `prefixe a` qui renvoie la liste des nœuds de `a` dans l'ordre préfixe, en temps linéaire (càd sans utiliser l'opérateur `@`). On pourra commencer par écrire une version naïve.
**(f)** (plus difficile) écrire une fonction `infixe_egal a b` qui renvoie `true` ssi `infixe a = infixe b`. Cette fonction ne doit pas calculer explicitement `infixe a` ou `infixe b`, mais travailler petit à petit. On pourra utiliser une pile ou une liste et le type

```
type exploration =
    | Renvoyer of int
    | Explorer of arbre ;;
```

## 2 Piles

Implémenter les fonctions suivantes. Elles doivent fonctionner sur n'importe quelle implémentation de pile qui respecte la spécification du cours, que je répète ici (on peut reprendre l'implémentation par liste pour tester, mais il est interdit d'utiliser les propriétés spécifiques de l'implémentation ; seule l'interface est autorisée) :

```
type 'a stack   (* pile contenant des valeurs de type 'a *)
exception EmptyStack
```

```
value create : unit -> 'a t  (* nouvelle pile *)
value is_empty : 'a t -> bool (* pile vide? *)
value top : 'a t -> 'a  (* sommet de la pile *)
value pop : 'a t -> 'a  (* enlever sommet de la pile *)
value push : 'a t -> 'a -> unit (* pousser au sommet *)
```

**(a)** `ecrase p` qui efface complètement le contenu de `p`.
**(b)** `echange p` qui échange les deux premiers éléments du sommet de la pile (si la pile contient au moins deux éléments). On vérifiera que `echange p; echange p;` laisse la pile `p` inchangée (opération involutive).
**(c)** `rotation p` qui déplace l'élément au sommet de `p` tout au fond de `p`.
**(d)** `taille p` qui renvoie le nombre d'éléments de la pile. À la fin de son exécution, la pile doit être dans état d'origine.
**(e)** `retourner p` qui « renverse » la pile. On pourra s'aider d'une liste.
**(f)** `copier p` qui copie la pile dans une nouvelle pile. L'ancienne pile doit rester intacte à la fin de l'appel.

## 3 Expressions

On reprend le type des expressions arithmétiques et des opérations postfixes du cours. De la même manière, le fichier d'aide fournit les types `expr` et `operation` ainsi que des fonction `afficher_expr` et `expr_aleatoire` que vous pouvez utiliser.
**(a)** Écrire une fonction qui évalue une suite d'opérations postfixes (de type `operation list`) à l'aide d'une pile.
**(b)** Écrire une fonction qui transforme une suite d'opérations postfixes en une expression correspondante. Par exemple, la liste d'opérations `2 3 + 4 + 2 3 + *` correspond à l'expression `((2+3)+4) * (2+3)`.
**(c)** En déduire une fonction d'évaluation des opérations postfixes qui utilise la fonction d'évaluation des expressions.
**(d)** Comment peut on obtenir une suite d'opérations postfixes à partir d'une expression sous forme d'arbre ?