Exercice 1. Permutation triable avec une pile

Dans cet exercice on s'interdit d'utiliser les traits impératifs du langage OCaml (références, tableaux, champs mutables, etc.).

On représente en OCaml une permutation σ de $[\![0, n-1]\!]$ par la liste d'entier $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$. Un arbre binaire étiqueté est soit un arbre vide, soit un nœud formé d'un sous-arbre gauche, d'une étiquette et d'un sous-arbre droit :

```ocaml
type arbre = V | N of arbre * int * arbre
```

On représente un arbre binaire non étiqueté par un arbre binaire étiqueté en ignorant simplement les étiquettes. On étiquette un arbre binaire non étiqueté à n nœuds par $[\![0; n-1]\!]$ en suivant l'ordre infixe de son parcours en profondeur. La permutation associée à cet arbre est donnée par le parcours en profondeur par ordre préfixe. La figure 1 propose un exemple (on ne dessine pas les arbres vides).

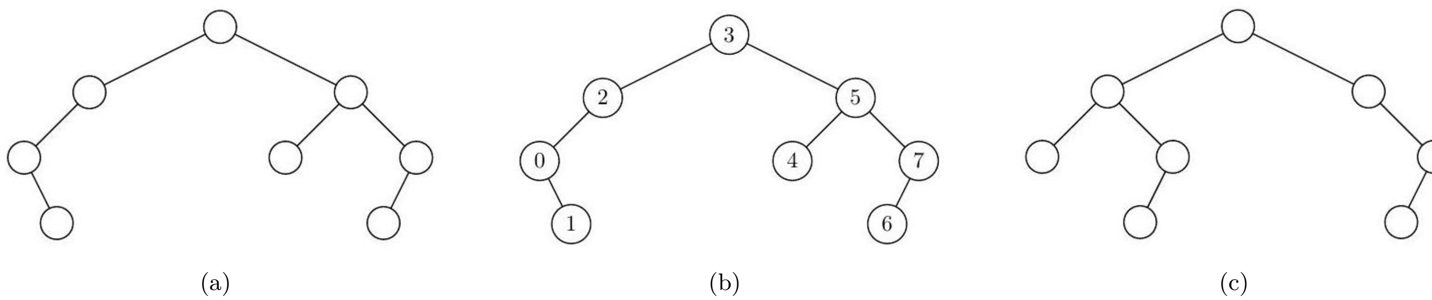

Figure 1: (a) un arbre binaire non étiqueté; (b) son étiquetage en suivant un ordre infixe, la permutation associée est $[3; 2; 0; 1; 5; 4; 7; 6]$; (c) un autre arbre binaire non étiqueté.

1. Étiqueter l'arbre (c) de la figure 1 et donner la permutation associée.
2. Écrire une fonction `parcours_prefixe : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe de son parcours en profondeur.
3. Écrire une fonction `etiquette : arbre -> arbre` qui prend en paramètre un arbre dont on ignore les étiquettes et qui renvoie un arbre identique mais étiqueté par les entiers de $[\![0, n-1]\!]$ en suivant l'ordre infixe d'un parcours en profondeur.

Une permutation σ de $[\![0, n-1]\!]$ est **triable avec une pile** s'il est possible de trier la liste $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$ en utilisant uniquement une structure de pile comme espace de stockage interne. On considère l'algorithme suivant, énoncé ici dans un style impératif :

- Initialiser une pile vide;
- Pour chaque élément en entrée :
  - Tant que l'élément est plus grand que le sommet de la pile, dépiler le sommet de la pile vers la sortie;
  - Empiler l'élément en entrée dans la pile;
- Dépiler tous les éléments restant dans la pile vers la sortie.

Par exemple, pour la permutation $[3; 2; 0; 1; 5; 4; 7; 6]$, on empile $3, 2, 0$, on dépile 0 , on empile 1 , on dépile $1, 2, 3$, on empile 5,4 , on dépile 4,5 , on empile 7,6 , on dépile 6 , 7 . On obtient la liste triée $[7; 6; 5; 4; 3; 2; 1; 0]$ en supposant avoir ajouté en sortie les éléments dans une liste. On admet qu'une permutation est triable par pile si et seulement cet algorithme permet de la trier correctement.

5. Dérouler l'exécution de cet algorithme sur la permutation associée à l'arbre (c) de la figure1 1 et vérifier qu'elle est bien triable par pile.
6. Écrire une fonction `trier : int list -> int list` qui implémente cet algorithme dans un style fonctionnel. Par exemple, trier $[3; 2; 0; 1; 5; 4; 7; 6]$ doit s'évaluer en la liste $[7; 6; 5; 4; 3; 2; 1; 0]$. On utilisera directement une liste pour implémenter une pile.
7. Montrer que s'il existe $0 \leq i < j < k \leq n-1$ tels que $\sigma_k < \sigma_i < \sigma_j$, alors σ n'est pas triable par une pile.
8. On se propose de montrer que les permutations de $[\![0, n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiquetés à n noeuds.

(a) Montrer que la permutation associée à un arbre binaire est triable par pile. On pourra remarquer le lien entre le parcours préfixe et l'opération empiler d'une part et le parcours infixe et l'opération dépiler d'autre part.

(b) Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire.

Exercice 2. Tableau autoréférent

1. Écrire une fonction `somme : int array -> int -> int` telle que l'appel `somme t i` calcule la somme partielle $\sum_{k=0}^{i} t.(k)$ des valeurs du tableau t entre les indices 0 et i inclus.

Un tableau t de $n > 0$ éléments de $[\![0, n-1]\!]$ est dit **autoréférent** si pour tout indice $0 \leq i < n$, $t.(i)$ est exactement le nombre d'occurrences de i dans t, c'est-à-dire que

$$\forall i \in [\![0, n-1]\!], \quad t.(i) = \mathrm{card}(\{k \in [\![0, n-1]\!] \mid t.(k) = i\})$$

Ainsi, par exemple, pour $n = 4$, le tableau suivant est autoréférent :

i	0	1	2	3
$t.(i)$	1	2	1	0

En effet, la valeur 0 existe en une occurrence, la valeur 1 en deux occurrences, la valeur 2 en une occurrence et la valeur 3 n'apparaît pas dans t.

3. Justifier rapidement qu'il n'existe aucun tableau autoréférent pour $n \in [\![1; 3]\!]$ et trouver un autre tableau autoréférent pour $n = 4$.

4. Écrire une fonction `est_auto : int array -> bool` qui vérifie si un tableau de taille $n > 0$ est autoréférent. On attend une complexité en $O(n)$.

5. Écrire une fonction `gen_auto : int -> unit` affichant tous les tableaux autoréferents de taille donnée. On pourra supposer qu'il existe une fonction `affiche` permettant d'afficher un tableau.

6. Quelle est la complexité de `gen_auto` ?

Pour accélérer la recherche, on peut élaguer l'arbre de recherche (repérer le plus rapidement possible qu'on se trouve dans une branche ne pouvant pas donner de solution).

7. Que peut-on dire de la somme des éléments d'un tableau autoréférent? En déduire une stratégie d'élagage pour accélérer la recherche.

8. Que peut-on dire si juste après avoir affecté la case $t.(i)$, il y a déjà strictement plus d'occurrences d'une valeur $0 \leq k \leq i$ que la valeur de $t.(k)$? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Comparer expérimentalement (sur ordinateur) le temps d'exécution avec la fonction de la question 5.

9. Après avoir affecté la case $t.(i)$, combien de cases reste-t-il à remplir? Combien de ces cases seront complétées par une valeur non nulle? À quelle condition est-on alors certain que la somme dépassera la valeur maximale possible à la fin? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 30$?

10. Montrer qu'il existe un tableau autoréférent pour tout $n \geq 7$. On pourra conjecturer la forme de ce tableau en testant empiriquement pour différentes valeurs de $n \geq 7$. On ne demande pas de montrer que cette solution est unique.