TD n°11 – Piles et files

Exercice 1 *Applications des piles*

Dans cet exercice on suppose écrite en C un type structure `pile`, représentant une pile d'entiers mutable. On dispose des fonctions `pile* creer()`, `bool est_vide(pile* p)`, `int depile(pile* p)`, `void empile(pile* p, int e)`, `void detruire(pile* p)` qui fonctionnent avec effets de bord.

1. Écrire une fonction `void echange_deux_premiers(pile* p)` qui échange les deux premiers éléments d'une pile.
2. Écrire une fonction `void nieme(pile* p, int n)` qui dépile et affiche le n-ième élément. Le reste de la pile doit rester le même. À quoi faut-il penser en terme de programmation défensive ?
3. Écrire une fonction `void premier_en_dernier(pile* p)` qui prend en entrée un pile et modifie la pile pour que le premier élément devienne le dernier, mais les autres restent dans le même ordre.
4. Écrire une fonction `void renverse(pile* p)` qui prend en entrée une pile et renverse l'ordre des éléments dans la pile
5. Écrire une fonction `pile* decoupe(pile* p)` qui prend en entrée une pile et la découpe en deux. La première moitié sera renvoyée dans une autre pile (les éléments seront dans le même ordre qu'ils étaient à l'origine) et l'autre moitié sera mise dans la pile d'origine.
6. Écrire une fonction `pile* melange(pile* p1, pile* p2)` qui prend en entrée deux piles et les "mélange" aléatoirement. On suppose qu'on dispose d'une fonction `int zero_ou_un()` qui renvoie aléatoirement 0 ou 1 avec probabilités égales.
   La notion de mélange est la suivante : si x est dans la pile 1 et y est aussi dans la pile 1 et que x est plus proche de la tête que y, alors dans le mélange on doit toujours avoir x plus proche de la tête que y. De la même manière, l'ordre relatif des éléments dans la pile 2 doit être préservé. En revanche si x est dans pile 1 et y dans pile 2 on n'impose rien sur leur ordre dans le mélange.

Exercice 2 *Nombres de Hamming et file*

Les nombres de Hamming sont les nombres de la forme $2^a3^b5^c$ pour a, b, c entiers naturels quelconques. Les premiers entiers de Hamming sont 1,2,3,4,5,6,8,9,10,12,15,16,18,20,...

Le but de cet exercice est de générer la liste des n premiers nombres de Hamming. L'apporche naïve consiste à parcourir les entiers en vérifiant s'ils sont des nombres de Hamming, jusqu'à en avoir trouvé n.

1. Écrire une fonction `est_hamming : int->bool` qui vérifie si un entier est un nombre de Hamming.
2. Écrire une fonction `hamming_naif : int->int list` qui prend en entrée n est renvoie la liste des n premiers nombres de Hamming.

Si cette approche fonctionne bien pour les premiers termes, plus n grandit et plus les nombres de Hamming sont éloignés les uns des autres (par exemple le 1999e est 8 100 000 000 et le 2000e 8 153 726 976). Il devient donc trop coûteux d'explorer tous les entiers pour trouver les nombres de Hamming.

On va plutôt générer les nombres de Hamming à partir d'autres nombres de Hamming. On utilise pour ce faire trois files f_2, f_3 et f_5, qui initialement contiennent le nombre 1 et on leur applique l'algorithme suivant, jusqu'à avoir affiché n valeurs (afficher soulageant le code des quelques lignes nécessaires à la création d'une liste) :

- on détermine le plus petit élément entre les trois têtes de files, noté k, et on l'affiche.
- on retire k des files où il est présent : k peut en effet être la tête de plusieurs files.
- on enfile sur la file f_2 l'entier $2k$, sur f_3 l'entier $3k$ et sur f_5 l'entier $5k$.

Cet algorithme repose sur le fait que tout nombre de Hamming est le produit par 2,3 ou 5 d'un autre nombre de Hamming plus petit.

Pour l'implémentation on utilisera le module Queue de Ocaml, qui propose une implémentation mutable de file. Les primitives ont les noms suivants (en anglais), le type `'a t` désigne une file :

- `Queue.create : unit -> 'a Queue.t` qui crée une file vide
- `Queue.is_empty : 'a Queue.t -> bool` qui teste si une file est vide
- `Queue.push : 'a -> 'a Queue.t -> unit` qui ajoute un élément
- `Queue.pop : 'a Queue.t -> 'a` qui retire et renvoie l'élément le plus ancien
- `Queue.peek : 'a Queue.t -> 'a` qui renvoie sans retirer l'élément le plus ancien

Les fonctions `peek` et `pop` lèvent l'exception `Empty` si la file est vide.

3. Traduire l'algorithme en Ocaml.
4. (*) L'inconvénient de la démarche précédente est que le même nombre peut se retrouver dans plusieurs des trois files. Modifier votre fonction pour que cela ne soit plus le cas.

Exercice 3 *Permutations et piles*

Une permutation de $[|1,n|]$ est une manière de réarranger les entiers de 1 à n. Par exemple pour $n = 5$, (1 2 4 5 3) est une permutation. (1 2 3 4 5) en est une aussi.

on peut aussi le voir comme les 5-uplets dont les éléments sont exactement ceux de $[|1,n|]$, sans répétitions.

On dit qu'une permutation $(a_1a_2...a_n)$ de $[|1,n|]$ peut être engendrée par une pile lorsqu'il est possible, à partir de la permutation $(12...n)$ et d'une pile (initialement vide), d'afficher la séquence de sortie $(a_1a_2...a_n)$ en utilisant uniquement les opérations suivantes :

- empiler l'élément suivant dans la permutation d'entrée.
- dépiler un élément de la pile et l'afficher

Par exemple, si E et D désignent respectivement les deux opérations permises, la permutation (231) est engendrée par la suite d'opérations EEDEDD.

1. Parmi les permutations suivantes, lesquelles peuvent être engendrées par une pile ?

$$(312), (3421), (4537216), (35768492101)$$

2. Montrer que s'il existe un triplet $(i,j,k) \in [|1,n|]^3$ tel que $i < j < k$ et $a_j < a_k < a_i$, alors la permutation $(a_1a_2...a_n)$ ne peut pas être engendrée par une pile.

Pour l'implémentation on utilisera le module Stack de Ocaml, qui propose une implémentation mutable de file. Les primitives ont les noms suivants (en anglais), le type `'a t` désigne une file :

- `Stack.create : unit -> 'a t` qui crée une file vide
- `Stack.is_empty : 'a t -> bool` qui teste si une file est vide
- `Stack.push : 'a -> 'a t -> unit` qui ajoute un élément
- `Stack.pop : 'a t -> 'a` qui retire et renvoie l'élément le plus neuf
- `Stack.top : 'a t -> 'a` qui renvoie sans retirer l'élément le plus neuf

3. Écrire une fonction Caml `est_engendrable : int list -> bool` déterminant si une permutation peut être engendrée par une pile. Dans le cas d'une réponse positive, la fonction affichera la suite d'opérations permettant de la produire. Les permutations seront représentées par le type `int list`.
4. Montrer enfin que toute permutation peut être engendrée à l'aide de deux piles, et rédiger la fonction Caml correspondante.