TD #7 — Pile ou file, analyse amortie, références…

Exercice 1. *File avec deux piles*

On esquisse la stratégie suivante pour implémenter une file avec deux piles :

- La file vide correspond à deux piles vides s_1 et s_2
- On enfile les éléments en les empilant dans s_1
- Pour défiler : si s_2 est vide on dépile tout s_1 et l'empile dans s_2, et on dépile une fois s_2, sinon on dépile juste une fois s_2

1. Illustrez cette stratégie avec quelques dessins.

On souhaite implémenter une file fonctionnelle en OCaml en suivant cette stratégie ; on se dotera pour cela d'un `type 'a queue = 'a list * 'a list`.

2. Donnez une valeur `(empty_queue:'a queue)` qui correspond à une file vide.

3. Définissez une fonction `push : 'a queue -> 'a -> 'a queue` telle que `push q x` construit une nouvelle file obtenue en ajoutant `x` à la file `q`.

4. De même pour une fonction `pop : 'a queue -> 'a * 'a queue` telle que pour une file `q` non vide `pop q` s'évalue en l'élément en tête de `q` et une nouvelle file égale à la queue de celle-ci.

On souhaite maintenant prouver (ou tout du moins argumenter) la correction de notre implémentation relativement aux spécifications d'une pile fonctionnelle. Pour cela, on va montrer que *lorsqu'elles n'échouent pas* les fonctions `push` et `pop` préservent certains invariants sur la structure.

5. Montrez que les éléments de s_1 apparaissent dans l'ordre inverse d'arrivée (le sommet de la pile a été `push`ed après tous les autres éléments, récursivement).

6. Montrez que les éléments de s_2 sont plus anciens que ceux de s_1 (ont été `push`ed avant).

7. Montrez que les éléments de s_2 apparaissent dans l'ordre d'arrivée (le sommet de la pile a été `push`ed avant tous les autres éléments, récursivement).

8. Conclure

On s'intéresse enfin à l'efficacité de cette stratégie d'implémentation d'une file.

9. Montrez que le coût pire cas d'une opération `pop` sur une file contenant n éléments est un $O(n)$.

Ceci ne paraît pas mieux qu'une implémentation naïve comme celle vue en TP. On va cependant montrer que l'on a ici un bien meilleur coût *amorti*.

10. Esquissez une preuve du fait que le coût pire cas de n opérations `push` et/ou `pop` (c'est à dire, le pire cas de la somme des coûts) à partir d'une file vide est un $O(n)$.
Indication : comptez pour chaque élément le nombre d'opérations dans lesquelles il est impliqué (sa contribution au coût global) au cours de sa « présence dans la structure ».

Exercice 2. *Références et pile impérative en OCaml*

On décrit brièvement les *références* OCaml par les trois fonctions suivantes :

- `ref : 'a -> 'a ref` telle que `ref x` s'évalue en une (nouvelle) *référence* de valeur `x` ;
- `(!) : 'a ref -> 'a` opérateur unaire préfixe tel que `!r` s'évalue en la valeur associée à la référence `r` ;
- `(:=) : 'a ref -> 'a -> unit` opérateur binaire infixe tel que `r := x` *modifie* la valeur référencée par `r` en `x`. (Toute ressemblance avec la notion de pointeur n'est pas fortuite.)

On définit alors le `type 'a stack = 'a list ref` d'une pile *impérative* (ou mutable).

1. Définissez une fonction `make : unit -> 'a stack` qui construit et renvoie une *nouvelle* pile vide. Pourquoi une telle fonction n'était-elle pas nécessaire pour une pile fonctionnelle (et le devient maintenant) ?
2. Définissez une fonction `is_empty : 'a stack -> bool` qui teste si son argument est une pile vide. Pourquoi une telle fonction n'était elle pas nécessaire etc. ?
3. Définissez une fonction `push : 'a stack -> 'a -> unit` qui met à jour (modifie) la valeur référencée par son premier argument pour y ajouter son second en sommet.
4. Définissez une fonction `pop : 'a stack -> 'a` qui (si possible) met à jour (modifie) la valeur référencée par son premier argument pour en supprimer la tête, et s'évalue en cette dernière.
Indication de syntaxe : rappelez vous qu'un `let` lie vraiment un motif, et que l'on peut par exemple écrire `let () = e1 in e2` pour toute expression `(e1:unit)`.

Exercice 3. *Valores menores más cercanos*

Essayez de prouver **rigoureusement** (et en autonomie) la correction d'une de vos fonctions `vmmc2` (en OCaml ou C) du TP d'hier (indication : vous devriez avoir besoin d'au moins deux invariants).
Prouvez aussi que la version « tableau » en C est (également) de coût linéaire en la longueur de son argument (procédez par analogie avec la version utilisant explicitement une pile).