MPSI : TP/TD Piles et files

Piles

On utilise le module `Stack` de OCaml.

Exercice 1. Écrire une fonction **traiter** qui utilise une pile pour évaluer une expression arithmétique en notation postfixée.

```
# traiter "44  3   1 - -";;
- : int = 42
```

On suppose que l'expression est donnée sous forme de chaîne de caractères et qu'elle est bien formée. Il peut cependant y avoir un ou plusieurs caractères d'espacement entre deux items.

Pourt séparer les items de la chaîne, on peut utiliser **String.split_on_char** :

```
# String.split_on_char ' ' "44  3   1 - -" ;;
- : string list = ["44"; ""; "3"; ""; ""; "1"; "-"; "-"]
```

Enfin, on rappelle l'opérateur de conversion de type **int_of_string**.

Exercice 2. On dispose d'une pile d'assiettes bleues ou rouges numérotées disposées dans le désordre. Comment procéder pour former une pile dans laquelle les assiettes bleues sont situées sous les assiettes rouges, mais en faisant en sorte que pour chacune des deux couleurs l'ordre relatif ne soit pas modifié ? (Autrement dit, si l'assiette bleue i est située sous l'assiette bleue j dans la pile initiale, ce sera toujours le cas dans la pile finale.) On définit les types :

```
    type couleur = Bleue | Rouge and assiette = couleur * int ;;
```

Rédiger une fonction de type **assiette t -> unit** qui range une pile d'assiette en disposant les assiettes bleues sous les assiettes rouges et en respectant l'ordre initial des assiettes.

```
# let p = let p1 = Stack.create() in
  Stack.push (R,1) p1;Stack.push (R,2) p1;
  Stack.push (B,1) p1;Stack.push (R,3) p1;
  Stack.push (B,2) p1;p1 in
ranger p;
let affiche_assiette a =
Printf.printf
"(%s,%d)\n" (if fst a=R then "R" else "B") (snd a);
in Stack.iter affiche_assiette p;;
(R,3)
(R,2)
(R,1)
(B,2)
(B,1)
- : unit = ()
```

Files

Exercice 3. On utilise le module `Queue` de OCaml.

Les *nombres de Hamming* sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont 2, 3 et 5 :1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30...

Le but de cet exercice est de les générer de manière croissante. Évidemment, on peut parcourir un à un tous les entiers en testant à chaque fois si ceux-ci sont des entiers de Hamming, mais cette démarche montre vite des limites (songez que le 1999e entier de Hammingest égal à 8 100 000 000 et le 2000e à 8 153 726 976 : il faudrait tester plus de 53 millions de nombres avant d'augmenter notre liste d'un élément !).

On adopte donc la démarche suivante : on utilise trois files f_2, f_3, f_5 contenant initialement le nombre 1, et on suit la démarche suivante :

1. on détermine le plus petit des trois têtes de file, que l'on note k et que l'on imprime à l'écran
2. on retire cet élément des files où il se trouve
3. on insère en queue des files f_2, f_3 et f_5 les entiers $2k$, $3k$ et $5k$.

Vous l'avez compris : cette démarche utilise le fait que tout nombre de Hamming différent de 1 est le produit par 2, 3 ou 5 d'un nombre de Hamming plus petit.

1. Rédiger une fonction OCaml permettant l'affichage des n premiers nombres de Hamming.
2. 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.

0.1 Files et complexité amortie

Une séquence d'un algorithme ayant de bonnes propriétés de complexité amortie alterne entre :

— de (nombreuses) opérations de faible coût faisant monter progressivement le potentiel,

— et de (peu nombreuses) opérations de coût élevé faisant diminuer brusquement le potentiel.

Il y a 3 méthodes usuelles d'analyse amortie : la *méthode de l'agréga*, la *méthode comptable* et la *méthode du potentiel* (seule étudiée ici).

Méthode du potentiel : A chaque entrée possible x, on associe un nombre positif ou nul $\phi(x)$ dit *potentiel*. Il représente un coût latent dans l'entrée mais pas encore réalisé.

Remarque. Toute la difficulté consiste à identifier un bon potentiel !

Définition 1. Soit une opération *op* dont l'éxécution produit une sortie x_s à partir d'une entrée x_e. Le *coût réel* C de *op* est la complexité temporelle de son exécution. Son *coût amorti* A est la somme du coût réel et de la variation de potentiel :

$$A = C + \phi(x_s) - \phi(x_e).$$

— Dans cette définition, les *entrées* et *sorties* représentent au sens large ce qui est donné à l'algorithme (paramètres, état de la mémoire au début de l'appel) et ce qu'il produit (résultats renvoyés, état de la mémoire en sortie).

— Une *séquence d'opérations consécutives* est une suite d'appels à un algorithme de sorte que chaque sortie d'un appel soit l'entrée du suivant.

— On montre que le coût réel est toujours inférieur au coût amorti.

Théorème 0.1. *Soit une suite de n opérations*

$$x_0 \xrightarrow{op_1} x_1 \xrightarrow{op_2} x_2 \dots \xrightarrow{op_n} x_n$$

à partir d'une entrée x_0 telle que $\phi(x_0) = 0$. En notant C_i le coût réel de l'opération op_i et A_i son coût amorti, on a la relation d'amortissement suivante :

$$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} A_i.$$

Exercice 4. Montrer le théorème d'amortissment.

Corollaire 1. *Avec les notations du th. d'amortissement, si le coût amorti est borné par une constante k, alors la complexité moyenne au sein d'une* séquence arbitraire de n opérations *est bornée par k.*

Exercice 5. Montrer le corollaire du théorème d'amortissement.

Exercice 6. Nous implémentons une structure de file immuable. Celles que nous avons considérées jusqu'ici étaient mutables.

Dans un fichier d'interface **files.mli** [1] écrivons :

```
type 'a queue
val empty : 'a queue
val is_empty: 'a queue -> bool
val enqueue: 'a queue -> 'a -> 'a queue
val dequeue: 'a queue -> 'a * 'a queue
```

Il s'agit d'implanter ces primitives. Pour réaliser une file immuable on se sert de deux piles immuables (on peut efficacement utiliser des listes OCaml). Dans la première pile, on ajoute les éléments qui entrent dans la file ; dans la seconde, on retire les éléments qui sortent de la file. lorsque la seconde pile est épuisée, on y déplace tous les éléments de la première pile.

On pose donc tout d'abord :

```
type 'a queue = {
front: 'a list; (*liste où on retire les éléments*)
rear:  'a list; (*liste où on ajoute les éléments*)
}
```

La liste `front` est celle dans laquelle on retire des éléments et `rear`, celle dans laquelle on les ajoute.

Par exemple, la file qui contient les éléments **1,2** dans cet ordre (1 sera le prochain élément défilé) peut être représentée par

```
let f12 = {front = []; rear = [2;1]};;
let f12 = {front = [1]; rear = [2]};;
let f12 = {front = [1;2]; rear = []};;
```

Q.1 Avec notre structure de données `queue` de combien de façons différentes peut-on représenter une file qui contient les éléments $1, 2, \dots, n$ dans cet ordre ?

1. Les fichiers **.ml/.mli** jouent un rôle analogue aux fichiers **.c/.h** que l'on trouve en C.

Q.2 Dans une structure mutable, la création de file vide est dévolue à une fonction comme `empty_mutable : unit - > 'a mutable_queue`. En revanche pour notre type de files immuables, la file vide `empty` est une *valeur* et non pas une *fonction* (par analogie, songer qu'il n'y a qu'une seule liste vide en OCaml : la valeur `[]`).

Écrire la valeur `empty : 'a queue` qui représente une file vide.

Q.3 Écrire la fonction de complexité temporelle $O(1)$ `is_empty : 'a queue -> bool`.

En observant les types d'arrivées des fonctions `enqueue` et `dequeue`, on remarque que les opérations d'enfilement et défilement produisent une nouvelle file, contrairement à ce qu'il se passe pour une structure de file mutable où on aurait plutôt des types comme les suivants :

```
val mutable_enqueue: 'a mutable_queue -> 'a -> unit
val mutable_dequeue: 'a mutable_queue -> 'a
```

En clair dans la structure de file immuable, les arguments des fonctions `enqueue` et `dequeue` ne sont pas modifiés.

Q.4 Écrire la fonction `enqueue : 'a queue -> 'a -> 'a queue`. La complexité doit être constante.

Voici un exemple d'utilisation :

```
# let q = enqueue (enqueue empty 1) 2 in is_empty q;;
- : bool = false
```

Q.5 Écrire la fonction `dequeue : 'a queue -> 'a * 'a queue` telle que `dequeue q` renvoie un couple dont le premier élément est celui qui a été retiré à `q` et le second est une nouvelle file qui correspond à ce qu'il reste de `q` après ce retrait.

Dans le cas où la file est vide une exception (de votre choix) est soulevée. Sinon, le principe est le suivant : si la liste `front` n'est pas vide on retire simplement son premier élément, mais si elle est vide, l'inverse de la liste `rear` prend la place de `front` et c'est à cet inverse qu'on retire un élément.

Voici un exemple d'exécution :

```
# let q = enqueue (enqueue empty 1) 2 in
let x, q1 = dequeue q in let y, q2 = dequeue q1 in
x,y,is_empty q2;;
    - : int * int * bool = (1, 2, true)
```

Q.6 Soit une file `q` contenant n éléments.

(a) Quelle est la complexité au mieux de l'appel `dequeue q` ?

(b) Identifier le pire cas pour l'appel `dequeue q` et donner la complexité au pire.

(c) Calculer proprement (c'est à dire avec une espérance mathématique) la complexité en moyenne de l'appel `dequeue q`.

Comme on le voit, un appel à `enqueue` est peu coûteux. C'est en général aussi le cas pour les appels à `dequeue` sauf dans de rares situations. Tout est donc réuni pour se lancer dans un calcul de complexité amortie. On étudie donc, en partant de la file vide, le coût d'une séquence

d' opérations (enfilement ou défilement) où chaque opération s'applique à la file obtenue avec l'opération précédente.

Pour une file **q**, on définit son potentiel ainsi :

$$\Phi(\mathbf{q}) \stackrel{\text{def}}{=} \text{la longueur de la liste } \mathbf{q.rear}$$

Le *coût amorti* a d'une opération est donc la complexité réelle c de cette opération plus la différence des potentiels après et avant l'opération :

$$a = c + \Phi(\text{après}) - \Phi(\text{avant})$$

Q.7 Soit une file `q` dont la longueur `q.rear`. est ℓ.

(a) On considère une opération d'enfilement sur `q`. Montrer que le coût amorti est constant.

(b) On considère une opération de défilement sur `q`. Montrer que le coût amorti est constant.

(c) En déduire la complexité moyenne amortie d'une opération dans une séquence d'opérations (enfilement/défilement) commençant sur une file ayant un potentiel nul.