MP2I : TP/TD Piles et files 2023

1 Piles

Exercice 1. Voir ici comment récupérer les arguments en ligne de commande ne OCaml. Relire également ce fichier qui rappelle les commandes d'entrées/sorties en OCaml.

On utilise le type **Stack** de OCaml.

On veut écrire un programme qui lit des lignes de textes dans un fichier et les réimprime en ordre inverse dans un autre.

Ce programme existe sous le nom de **tac** sous Linux et nous 'appellerons **tacos** notre version. Lancé depuis un terminal, il prend deux arguments **source** et **but** qui sont deux noms de fichiers. Il parcourt donc la source pour écrire dans le but.

Si dans le fichier **toto**, on trouve :

```
toto
gogo
```

Voici un exemple de déroulement :

```
$ ./tacos toto titi
$ cat titi
gogo
toto
```

Exercice 2. 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 3. 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 = ()
```

Exercice 4. Dans cet exercice, les *permutations* sont implantées par des listes.

On dit qu'une permutation $(a_1, a_2, \dots a_n)$ de $(1, 2, \dots n)$ peut être *engendrée par une pile* lorsque il est possible, à partir de la séquence d'entrée $(1, 2, \dots, n)$ et d'une pile (initialement vide), de produire la séquence de sortie $(a_1, a_2, \dots a_n)$ en utilisant les opérations suivantes :

— empiler l'élément suivant de la séquence d'entrée

— ou dépiler le sommet de la pile et l'imprimer à l'écran.

Par exemple, si E et D désignent respectivement chacune des deux opérations permises, la permutation $(2, 3, 1)$ est engendrée par la suite d'opérations : $EEDEDD$.

1. Parmi les permutations suivantes, lesquelles peuvent être engendrées par une pile ?
   $(3, 1, 2)$; $(3, 4, 2, 1)$; $(4, 5, 3, 7, 2, 1, 6)$ et $(3, 5, 7, 6, 8, 4, 9, 2, 10, 1)$

2. Soit une permutation $(a_1, a_2, \dots a_n)$ de $[1, \dots, n]$.
   Montrer que s'il existe un triplet $(i, j, k) \in [1, n]^3_{\mathbb{N}}$ tel que $i < j < k$ et $a_j < a_k < a_i$, alors la permutation $(a_1, a_2, \dots a_n)$ n'est pas engendrable par une pile.

3. Utilisant ce qui précède, écrire une fonction **encodable** qui prend en paramètre une liste d'entiers de 1 à n représentant une permutation (supposée bien formée) à afficher et déterminant si cette dernière peut être engendrée par une pile.
   Cette fonction gère une pile contenant des entiers et retourne une liste :
   — la liste vide si la permutation n'est pas encodable
   — sinon, une liste de $2n$ caractères E et D indiquant la séquence d'encodage.

```
# encoder [3;4;2;1];;
- : char list = ['E'; 'E'; 'E'; 'D'; 'E'; 'D'; 'D'; 'D']
# encoder[3;1;2];;
- : char list = []
```

4. Expliquer comment toute permutation peut être engendrée à l'aide de deux piles (mais toujours en respectant l'ordre de la séquence d'entrée), et rédiger la fonction **encoder2(p)** correspondante.

Il s'agit donc d'introduire une seconde pile pour stocker les éléments qui ne doivent pas être affiché tout de suite. On utilise les notations E, D pour l'empilement et le dépilement dans la pile principale et e, d pour la pile auxiliaire.

2 Files

Une *file de priorité* est un type abstrait élémentaire sur lequelle on peut effectuer trois opérations principales :

— insérer un élément ;

— extraire l'élément ayant la plus petite clé ;

— tester si la file de priorité est vide ou pas.

Une manière efficace d'implanter les files de priorité est la gestion d'un *tas* (*heap* en anglais). Nous verrons cette structure un peu plus tard. Pour le moment, nous nous contentons d'une implémentation basique.

Exercice 5. Nous choisissons d'implanter les files de priorité avec une liste doublement chaînée de maillons et une structure de contrôle à deux pointeurs et un indicateur de taille.

On commence par définir le type **elt** qui désigne le type des maillons :

```
typedef struct _element{
  int v;// value
  int p;// priority
}elt;

```

La priorité est le second champ. Plus elle est petite, plus le maillon est prioritaire.

Q1 Proposer un type **maillon** pour les maillons (3 champs) et la structure de contrôle **grip** (3 champs aussi).

Q2 Ecrire les fonctions

```
grip * init ();// crée une file vide

bool estVide(grip * p);// test de file vide

/*ajouter en fin de file sans tenir compte des priorités :*/
void ajoute(grip *pt, elt x);

// supprimer la tête de file
elt supprime(grip *pg);

// affichage d'un maillon
void printMaillon(maillon *pm);

// affichage d'une file
void printFile(grip* pg);

```

La fonction d'ajout met le nouvel élément en fin de file sans se préoccuper des priorités.
Voici un test

```
grip * pg = init();
printFile(pg);
elt e = {1,4};
ajoute(pg,e);
elt e2 = {2,5};
ajoute(pg,e2);
elt e3 = {3,1};
ajoute(pg,e3);
printFile(pg);

printf("\nSuppressions\n");
int n = pg->size;;
for (int i = 0; i<n; i++){
  printf("Suppression de ");
   _printElement(supprime(pg));
     printf("; pg->size=%d\n", pg->size);
}
printf("\nestVide(pg)=%d\n",estVide(pg));
free(pg);

```

et son rendu

```
(1,4)<->(2,5)<->(3,1)<->_

Suppressions
Suppression de (1,4); pg->size=2
Suppression de (2,5); pg->size=1
Suppression de (3,1); pg->size=0

estVide(pg)=1
```

Q3 Ecrire la fonction **void inserer(grip *pg, elt x)** qui met l'élément **elt** à sa bonne place dans une file de priorité déjà triée par ordre de priorité croissante.
A l'issue de tests consistant à faire grossir la file par des ajouts successifs suivis d'une remise en ordre, on souhaite obtenir un rendu comme celui-ci

```
None
--------------
ajouter(10,5)Après insertion on obtient :
(10,5)<->None
--------------
ajouter(20,7)Après insertion on obtient :
(10,5)<->(20,7)<->None
--------------
ajouter(30,1)Après insertion on obtient :
(30,1)<->(10,5)<->(20,7)<->None
--------------
ajouter(40,2)Après insertion on obtient :
(30,1)<->(40,2)<->(10,5)<->(20,7)<->None
--------------
ajouter(50,0)Après insertion on obtient :
(50,0)<->(30,1)<->(40,2)<->(10,5)<->(20,7)<->None
--------------
ajouter(60,3)Après insertion on obtient :
(50,0)<->(30,1)<->(40,2)<->(60,3)<->(10,5)<->(20,7)<->None
```

Exercice 6. 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.

Exercice 7. 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 = []};;
```

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

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

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 de complexité temporelle $O(1)$ `enqueue : 'a queue -> 'a -> 'a queue`.

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.

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 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.