Colle 8 : Autour des files et des piles

1 Pile

Reprenez l'implémentation de pile réalisée dans le TP précédent.

▷ **Question 1.** 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
type assiette = couleur * int
```

Rédiger une fonction `ranger : assiette pile -> unit` qui range une pile d'assiettes en disposant les assiettes bleues sous les assiettes rouges. ◁

▷ **Question 2.** Soit $P = (a_1, \ldots, a_n)$ une pile non vide de n éléments où a_1 est le sommet de la pile. On désire modifier la pile en $P = (a_2, \ldots, a_n, a_1)$. Cette opération s'appelle rotation d'une pile. Programmer la fonction rotation d'une pile. La fonction sera de type `'a pile->unit`. ◁

2 File

2.1 Implémentation des files

On va maintenant implémenter la structure de file dont on rappelle que les fonctions de base sont : `file_vide`, `est_vide`, `enfile`, `defile`, `premier`.

- On suppose pour cette première version que la taille de la file ne dépassera jamais un certain entier N et on implémente la file à l'aide d'un tableau de taille N :

  ```
  type 'a file = {mutable debut : int; mutable fin : int; elts : 'a array; taille : int};;
  ```

  Cette définition du type file signifie qu'une file est définie par quatre champs d'information : le tableau elts qui contient les éléments de la file, les indices de début et de fin de la file ainsi qu'un entier qui garde en mémoire la taille maximum N de la file ainsi implémentée.

  Attention, on utilise un tableau circulaire, il est donc tout à fait possible d'avoir fin < debut. Une conséquence de ce choix est que l'égalité `fin = debut` ne permet pas de distinguer entre la file vide et la file pleine ; pour palier à ce problème, on s'interdit de remplir complètement le tableau : ainsi, on décrète qu'une file qui contient $taille - 1$ éléments est pleine, situation caractérisée par l'égalité :
  (fin + 1) mod taille = debut.

  Ceci nous conduit à définir deux exceptions :

  ```
  exception Empty ;; exception Full ;;
  ```

  ▷ **Question 3.** Réaliser le type file avec cette implémentation. ◁

- Un inconvénient de l'implémentation précédente est qu'elle nécessite d'avoir une borne supérieure sur la taille de la file à gérer. Nous allons maintenant voir une autre implémentation de la structure de file qui permet d'éviter ce problème. On peut réaliser les opérations de file avec le type liste fourni en standard par Caml, mais l'opération "insertion en queue" a une mauvaise complexité sur ce type de listes, aussi utilise-t-on des structures de données mieux adaptées aux fonctionnalités des files. On implémente maintenant une file par un couple de listes : l'avant de la file, classée par ordre d'arrivée et l'arrière, classée par ordre inverse d'arrivée.

  ```
  type 'a file = { mutable avant : 'a list; mutable arriere : 'a list};;
  ```

  Les nouveaux arrivants seront placés en tête de la liste arrière, les premiers arrivés seront extraits en tête de la liste avant. Lorsque la liste avant est épuisée on retourne la liste arrière, on la place à l'avant et on réinitialise la liste arrière à `[ ]`. Cette opération de retournement a une complexité linéaire en la taille de la liste arrière, donc le temps d'extraction du premier élément d'une file ainsi implémentée n'est pas constant, mais comme un élément de la file n'est retourné qu'au plus une fois, le temps cumulé de n extractions en tête de la file est linéaire en n (complexité amortie constante).

  ▷ **Question 4.** Programmer les fonctions de la structure de donnée file à l'aide de cette structure concrète. ◁

2.2 La suite de Hamming

Un entier de Hamming est un entier naturel non nul dont les seuls facteurs premiers éventuels sont $2, 3, 5$. Le *problème de Hamming* consiste à énumérer les n premiers entiers de Hamming par ordre croissant. Pour celà, on remarque que le premier entier de Hamming est 1 et que tout autre entier de Hamming est le double, le triple ou le quintuple d'un entier de Hamming plus petit (ces cas n'étant pas exclusifs). Il suffit donc d'utiliser trois files d'attente, h_2, h_3 et h_5 contenant initialement le seul nombre 1 puis d'appliquer l'algorithme suivant :

Déterminer le plus petit des trois nombres en tête des files d'attente, soit x. Imprimer x, le retirer de chacune des files le contenant et insérer en queue de h_2, h_3 et h_5 respectivement $2x$, $3x$ et $5x$.

1. Programmer cet algorithme et faire afficher les n premiers entiers de Hamming. Observer l'évolution des files d'attente à chaque étape.
2. L'algorithme précédent est très dispendieux car il place la plupart des entiers de Hamming dans les trois files alors qu'une seule suffirait. En effet, si x est un entier de Hamming divisible à la fois par 2, 3 et 5 alors x a été placé dans h_2 au moment où l'on extrayait $x/2$, dans h_3 au moment où l'on extrayait $x/3$ et dans h_5 au moment où l'on extrait $x/5$. Modifier votre programme de sorte qu'un même entier de Hamming ne soit inséré que dans une seule des files d'attente.
3. Les entiers Caml sont limités ce qui ne permet pas d'aller très loin dans la suite de Hamming. Pour pouvoir traiter des grands nombres on convient de représenter un entier de Hamming x par le triplet (a, b, c) tel que $x = 2^a 3^b 5^c$. Reprendre votre programme avec cette convention et calculer le millionnième entier de Hamming. Pour comparer deux entiers de Hamming x et y connus par leurs exposants il suffit de comparer les réels $\ln x$ et $\ln y$ On admettra que la précision des calculs sur les flottants est suffisante pour ne pas induire de comparaison erronée.

2.3 Implémentation d'une file avec une liste chainée :

- Définissons le type file à l'aide d'une liste chainée pour laquelle on garde un accès à la fois au premier mais aussi au dernier élément. :

  ```
  type 'a cell =
    | Nil
    | Cons of { content: 'a; mutable next: 'a cell }

  type 'a t = {
    mutable length: int;
    mutable first: 'a cell;
    mutable last: 'a cell
  }
  ```

▷ **Question 5.** Ecrire une réalisation de la structure de file avec ce type. ◁

- Considérons maintenant une liste circulaire définie ainsi :

```
type 'a cellule = {
  element : 'a;
  mutable suivante : 'a cellule
}

type 'a file =
  | Vide
  | Dernier of 'a cellule
```

▷ **Question 6.** Ecrire une réalisation de la structure de file avec ce type (on pourra tester avec la commande `a==b` si deux objets `a` et `b` sont les mêmes en mémoire). ◁