Option Informatique

TP $n°3$: structures de données

Deux implémentations du type File.

On étudie trois implémentations possibles de la structure de données `file`.

Dans tout calcul de complexité de cette partie, c'est bien le nombre d'opérations standard que l'on demande d'estimer (comparaisons, opérations arithmétiques, opérations structurelles des autres structures de données...).

Exercice 1. *Implémentation naïve.*

Dans cet exercice, on implémente la version immuable du type file comme un simple alias du type liste :

```
type 'a file = 'a list;;
```

1. Implémenter les opérations structuelles sur les files pour cette implémentation des files, et donner sans démonstration leur complexité.
   Il y a deux solutions possibles... n'en donner qu'une !
2. Quel est le problème ?

Exercice 2. *Implémentation par couples de listes.*

Dans cet exercice, on implémente la version immuable type file à l'aide d'un couple de listes.

```
type 'a file = ('a list) * ('a list);;
```

La première liste est constituée des premiers éléments de la file, classés par ordre de priorité. L'élément en tête de cette première liste est donc le premier élément de la file, etc. La seconde liste est constituée des derniers éléments de la file, conservés *dans l'ordre inverse* de leur priorité. L'ajout d'un nouvel élément se fait donc par un `cons` en tête de la seconde liste.
Il n'y a pas unicité de la représentation : par exemple, la file $1, 2, 3$ (où 1 est le premier élément de la file et 3 le dernier) peut aussi bien être représentée par `([1;2;3],[])`, `([1;2],[3])`, `([1],[3;2])` et `([],[3;2;1])`. Seule la file vide a une unique représentation : `([],[])`. Ainsi, l'égalité entre deux files n'est pas l'égalité naturelle de Caml, sauf l'égalité à la file vide.

3. Implémenter les opérations structuelles sur les files pour cette implémentation des files.
4. Donner sans démonstration la complexité de ces opérations dans le pire des cas.
5. On s'intéresse maintenant à la complexité *amortie*, c'est-à-dire à la complexité moyenne des opérations pour n applications sucessives des opérations. Que vaut-elle ?

On obtient ici une bonne complexité amortie en utilisant la même stratégie qu'utilise Python pour ses "listes", qui ont en fait des tableaux redimensionnables.

Exercice 3. *Implémentation avec un tableau circulaire.*

Dans cet exercice, on implémente la version mutable du type file à l'aide d'une structure impérative. On va faire émerger pas-à-pas sa description.

On commence par une restriction : en pratique, lorsque l'on utilise des files, la longueur de la file est bornée. Par

exemple, le parcours en largeur d'un arbre binaire (complet) de profondeur n nécessite [1] une file de longueur au plus 2^{n+1}. On décide donc que la longueur d'une file ne pourra pas excéder une longueur maximale `fmax` que l'on affecte une bonne fois pour toute à une valeur grande (ou moins grande suivant nos besoins), par exemple :

```
let fmax = 4194303;;
```

On considèrera uniquement des files de longueur au plus `fmax`.

On commence par définir une file comme un tableau de longueur `fmax`, accompagné de deux indices remarquables `debut` et `fin` : les éléments de la file, par ordre de priorité, seront alors les éléments de ce tableau compris entre `debut` et `fin-1`. Ajouter un élément à la file revient donc à affecter la coordonnée d'indice `fin` du tableau (puis à incrémenter `fin`), alors que la queue de la file est simplement obtenue en incrémentant `debut`.
Sur l'illustration suivante, on a `fmax=12`.

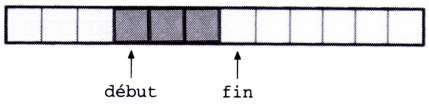

Figure 1 : *Une file de trois éléments avec un tableau rectiligne.*

L'intérêt d'une telle implémentation est de permettre que **toutes** les opérations structurelles soient **toujours** en temps constant.

Le problème de cette implémentation est que la longueur maximale de la file ne reste pas `fmax` : elle diminue à chaque décalage de `debut`. On propose alors l'astuce suivante, qui consiste à imaginer que le tableau n'est pas rectiligne mais circulaire, ce qui revient à considérer les indices du tableau modulo `fmax`. Une fois encore, illustration pour `fmax=12`.

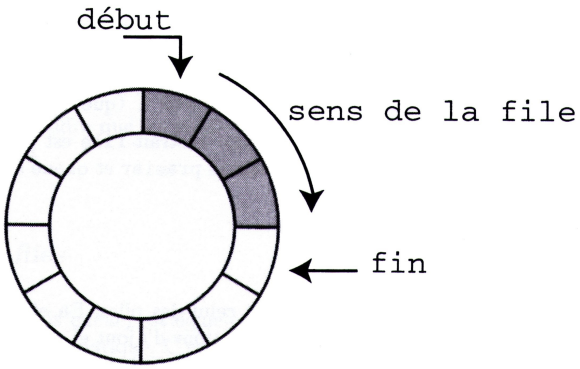

Figure 2 : *Une file de trois éléments avec un tableau circulaire.*

Comme dans l'exercice précédent, la représentation n'est pas unique. Ainsi, la file $1, 2, 3$ (où 1 est le premier élément de la file et 3 le dernier) peut aussi bien être représentée par `[|1;2;3;0;0;0;0;0;0;0;0;0|]` avec `debut=0` et `fin=3`, que par `[|0;0;1;2;3;0;0;0;0;0;0;0|]` avec `debut=2` et `fin=5`, ou par `[|2;3;0;0;0;0;0;0;0;0;0;1|]` avec `debut=11` et `fin=2`, ou même par `[|2;3;42;42;42;42;666;42;42;43;42;1|]` avec `debut=11` et `fin=2`.

Mais il reste un problème : pour `debut=fin=3`, quelle file représente `[|1;2;3;4;5;6;7;8;9;10;11;12|]` ? S'agit-il de la file vide ou de la file $4, 5, 6, 7, 9, 9, 10, 11, 12, 1, 2, 3$? Une solution pour régler le problème est d'indiquer non seulement les indices `debut` et `fin`, mais aussi si la file est vide ou pas.

Et... il y a encore le problème de l'initialisation (que met-on dans le tableau au moment de la création de la file ?). On garde la solution habituelle : le type `'a option` !

Ouf ! Finalement, on en est venu à l'implémentation suivante :

```
type 'a file = {contenu : 'a option array;  mutable debut : int;  mutable fin : int;  mutable vide : bool};;
```

6. Implémenter les opérations structuelles sur les files pour cette implémentation des files. Vous devrez faire en sorte qu'elles soient toutes de complexité constante.

1. Pourquoi, au fait ?