TP3 : Structures de données

Implémentation des files en OCaml à l'aide de tableaux

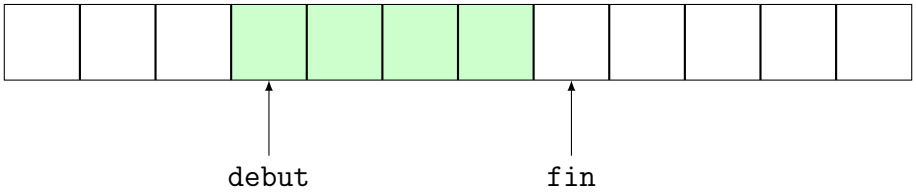
La façon la plus simple d'implémenter des files à l'aide de tableaux OCaml est de créer un type enregistrement :

```
type 'a file_array = {contenu : 'a array ; mutable debut : int ;
                      mutable fin : int} ;;
```

contenant trois champs, le nom de chaque champ étant appelé une étiquette : `contenu` qui est le tableau proprement dit, `debut` qui pointe sur le premier élément de la file et `fin` qui pointe sur la case qui suit le dernier élément (on doit avoir `debut` $<$ `fin`).

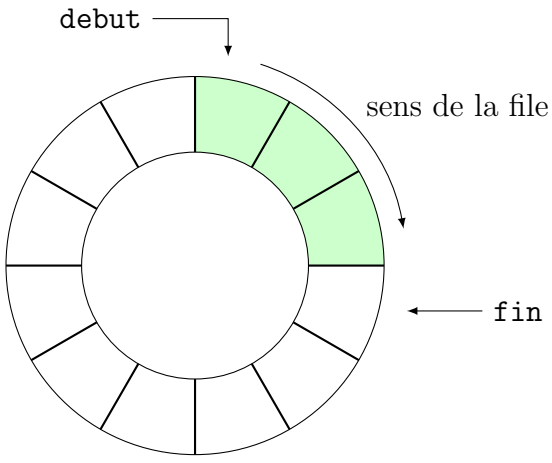
Prendre la queue d'une file revient à se placer dans la coordonnée d'indice `fin` et à incrémenter cet entier ; quitter la queue d'une file revient simplement à incrémenter `debut`.

La figure ci-dessous représente une file de quatre éléments avec cette implémentation.



Toutefois, un problème se pose : chaque action de queue déplace `debut` vers la droite du tableau jusqu'à éventuellement dépasser la taille de celui-ci et conduire à une erreur, alors qu'en fait tout le début du tableau est libre.

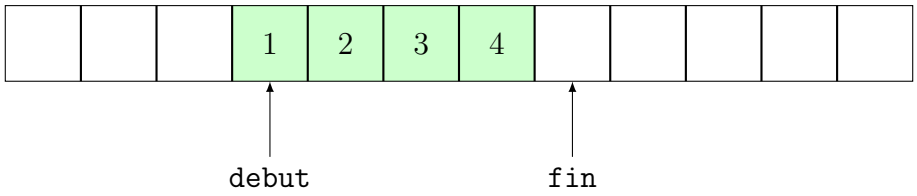
On pourrait imaginer régler ce problème par des translations à gauche du contenu de la file à des moments judicieux... On peut également envisager que le tableau n'est plus linéaire mais circulaire, les éléments étant lus par exemple, dans le sens des aiguilles d'une montre.



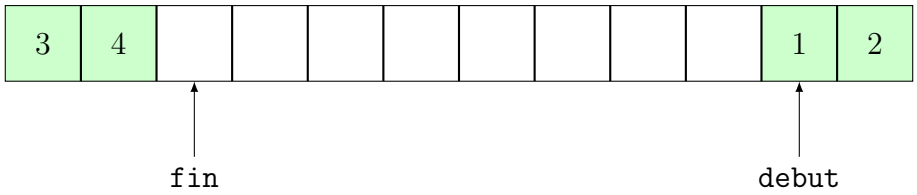
Sur le plan mathématique, cela revient à considérer les indices modulo la longueur du tableau et donc à laisser tomber la contrainte `debut` $<$ `fin`. L'indice `debut` pointe toujours

sur le début de la file et `fin` indique encore la coordonnée en laquelle on peut ajouter un nouvel élément.

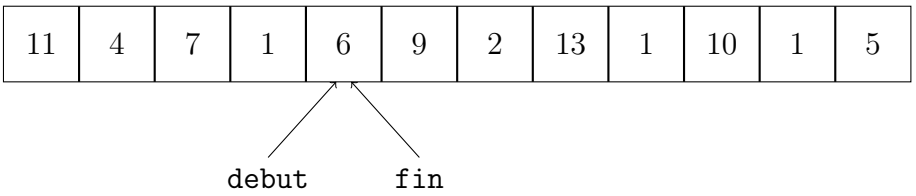
La représentation d'une file donnée n'est manifestement pas unique. Par exemple, la file (1,2,3,4) peut être représentée par :



ou par :



Mais alors, que représente la file



On ne peut pas distinguer la file pleine (6,9,2,13,1,10,1,5,11,4,7,1) de la file vide (). Pour palier à cet inconvénient, nous allons ajouter à notre type OCaml un champ booléen `vide`.

1. Modifier le type `file_array` en lui ajoutant un champ booléen `vide`.
2. Écrire une fonction en OCaml qui permet de savoir si une file est vide ou non.
3. Écrire une fonction en OCaml qui permet de savoir si une file est pleine ou non.
4. Écrire une fonction en OCaml qui ajoute un élément à la fin d'une file.
5. Écrire une fonction en OCaml qui supprime l'élément en tête d'une file.
6. Écrire une fonction qui calcule la taille d'une file.
7. Écrire une fonction qui déterminer si `x` est dans la file `f`.

Implémentation des files en OCaml à l'aide de listes

L'implémentation des files sous la forme de tableaux (rectilignes ou circulaires) impose une taille maximale fixée à la création. L'implémentation directe à l'aide d'une liste rend une des deux opérations de sélection couteuse.

Nous allons coder les files à l'aide de deux listes ou plutôt d'un couple de listes.

La première liste est constituée des éléments les plus anciens de la file, classés suivant leur ordre d'arrivée. L'élément en tête de cette liste est donc le premier élément de la file. La seconde liste est constituée des éléments les plus récents de la file, conservés dans l'ordre inverse de la première. L'ajout d'un nouvel élément se fait donc en tête de la seconde liste.

Par exemple, le couple ([1 ;2 ;3],[5 ;4]) code la file (1,2,3,4,5) (où 1 est le premier élément de la file, 5 le dernier).

Il n'y a pas unicité de cette représentation. ([1 ;2],[5 ;4 ;3]) ou ([],[5 ;4 ;3 ;2 ;1]) codent aussi la file précédente. Et comme on le voit, une file non vide peut avoir une première liste vide. On décide dans ce cas de renverser la seconde liste et d'agir sur ([1 ;2 ;3 ;4 ;5],[]). Cette opération s'appelle la normalisation de la représentation de la file.

8. Définir le type `file_list` en OCaml.
9. Écrire une fonction en OCaml qui permet de savoir si une file est vide ou non.
10. Écrire une fonction en OCaml qui ajoute un élément à la fin d'une file.
11. Écrire une fonction en OCaml qui normalise la représentation d'une file si la première liste est vide.
12. Écrire une fonction en OCaml qui supprime l'élément en tête d'une file.
13. Écrire une fonction en OCaml qui calcule la taille d'une file.
14. Écrire une fonction en OCaml qui détermine si `x` est dans la file `f`.

Implémentation des dictionnaires en OCaml

Un dictionnaire peut être modélisé en OCaml par une liste de couples (c, v) de type

```
type element = {cle : int ; valeur : string} ;;
```

mais la recherche d'un élément serait alors de complexité au pire $O(n)$.

On peut améliorer la complexité de la recherche d'un élément en utilisant les tables de hachage. Notons C l'ensemble des clés et $n = Card(C)$. L'idée est de construire un tableau T de longueur $m << n$ et une fonction simple $h : C \mapsto [\![0; m-1]\!]$ appelée fonction de hachage qui distribue les clés de manière relativement uniforme dans les différentes cases de T. Les clés étant ici des entiers, nous allons par exemple choisir la fonction de hachage suivante : $h : c \mapsto c \mod m$.

On créé ainsi un tableau T dont chaque case numéro i (appelée alvéole) est la liste des couples (c, v) telle que $h(c) = i$. Cette méthode permet de ne rechercher un élément que dans la liste de l'alvéole $h(c)$.

Pour définir une tableau de hachage, nous définissons donc le type suivant :

```
type dico = {h : int -> int ; tbl : element list array} ;;
```

15. Écrire une fonction en OCaml qui permet de définir un dictionnaire vide avec un hachage par division sur des clés entières.
16. Écrire une fonction en OCaml qui permet de chercher un élément d'un dictionnaire correspondant à une clé donnée.
17. Écrire une fonction en OCaml qui permet d'insérer un élément dans le dictionnaire.
18. Écrire une fonction en OCaml qui permet de supprimer un élément d'un dictionnaire correspondant à une clé donnée.

Avec la solution précédente, la partie coûteuse de la recherche d'une entrée dans la table est le parcours de la liste des enregistrements correspondants à la valeur de hachage de la clé considérée. C'est pourquoi d'autres solutions existent pour gérer les collisions sans recourir à des listes, autrement dit en stockant les associations (c, v) directement dans les cases du tableau.

L'une de ces méthodes consiste, en cas de collision (collision signifie que deux c ont le même hachage $h(c)$), à partir de $i = h(c)$ et à chercher une place libre dans la table (c'est ce qu'on appelle sonder la table).

On peut parcourir les cases voisines en testant successivement les cases d'indices $i + 1$ mod m, $i + 2$ mod m, $i + 3$ mod m, . . . jusqu'à trouver une place libre (on parle dans ce cas de sondage linéaire) mais ceci présente l'inconvénient de former des « agrégats » qui nuisent à la répartition uniforme recherchée.

Pour éviter cet inconvénient, on peut utiliser une deuxième fonction de hachage h' et chercher un emplacement disponible parmi les cases d'indices $i + h'(c)$ mod m, $i + 2h'(c)$ mod m, $i+3h'(c)$ mod m ..., sans pour autant résoudre complètement le problème des agrégats.

Évidemment, cette méthode (dite d'adressage ouvert) exige que le nombre k de clés soit inférieur à la taille de la table m. En outre, on imagine aisément que lorsque le rapport $\alpha = \frac{k}{m}$ se rapproche de 1, il devient de plus en plus difficile de trouver un emplacement vide : si $\alpha = 0,5$ un ajout nécessite en moyenne deux sondages, contre dix lorsque $\alpha = 0,9$. Aussi, lorsque α devient trop grand il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances.

On s'intéresse dans la suite à la méthode de résolution des collisions par adressage ouvert à l'aide d'un sondage linéaire.

19. Exécuter manuellement l'algorithme d'insertion dans une table de taille $m = 9$ des clés 5, 28, 19, 15, 20, 33, 12, 17, 10 avec la fonction de hachage $h : c \mapsto c \mod 9$.
20. On définit les types et la fonction

```
type element = {cle : int ; valeur : string} ;;
type objet = Vide | Element of element ;;
type dico = {h : int -> int ; tbl : objet array} ;;
```

Écrire une fonction en OCaml qui permet de définir un dictionnaire vide avec un hachage par division sur des clés entières.

21. Rédiger les fonctions permettant de chercher un élément et d'insérer un élément dans un dictionnaire à adressage ouvert avec sondage linéaire. On supposera la taille de la table grande devant le nombre de clés utilisées.
22. Que se passe-t-il si on supprime certaines clés de la table ? Comment peut-on résoudre ce problème ?
23. Dans une table de taille m, quelle est la probabilité qu'une clé soit placée dans une case vide précédée d'une autre case vide ? Quelle est la probabilité qu'une clé soit placée dans une case vide précédée de k cases pleines ? À votre avis, pourquoi le phénomène d'agrégat est-il mauvais pour l'efficacité des tables de hachages ?