

# Structures de Données (2)

Cette semaine nous allons aborder d'autres structures de données : les *files*, les *files de priorité* et les *tableaux associatifs*.

## 1 Files

Les files sont une abstraction des files d'attente (aéroports, boulangeries, section dessert au self, etc.). Elles ressemblent beaucoup aux piles mais on retire les éléments dans l'ordre auquel on les a ajoutés<sup>1</sup>. Les files sont encore une fois paramétrées par le type des éléments qu'elles contiennent (on aura ainsi des files d'entiers, etc.). Le nom usuel (anglais) que la plupart des langages de programmation donnent aux files est *queue*. Cette structure est très utile en pratique pour, justement, gérer des suites d'opérations à effectuer, des mails à envoyer, des messages reçus, etc. en préservant l'ordre dans lequel ces objets sont arrivés.

### 1.1 Spécifications

Nous verrons une implémentation persistante et une implémentation impérative, dont les spécifications suivent. On voit que les interfaces diffèrent au niveau de la création et modification de la file d'attente. Les deux opérations fondamentales sont :

- ajouter un élément à la fin de la file,
- retirer un élément du début de la file.

Là encore on choisira de lancer une exception si l'on tente d'enlever un élément d'une file impérative vide. `peek` est l'équivalent de `pop` sur les piles : retourner le premier élément sans l'enlever de la file. Pour les files persistantes, `take` doit à la fois renvoyer le premier élément `x` et la nouvelle version de la file (qui ne contient plus `x`). Je donne la préférence à l'usage du type `option` pour plusieurs raisons :

1. le type de `take` montre qu'il est possible que l'opération échoue ;
2. la fonction `take` est une fonction *totale* car elle fonctionne sur n'importe quelle file, même vide.

---

1. d'où, en anglais, l'acronyme *FIFO* : « first-in, first-out »

```

type 'a queue
value is_empty : 'a queue -> bool

(* structure persistante *)
value empty : 'a queue
value add : 'a queue -> 'a -> 'a queue
value take : 'a queue -> ('a * 'a queue) option
value peek : 'a queue -> 'a option

(* structure imperative *)
exception Empty
value create : unit -> 'a queue
value add : 'a queue -> 'a -> unit
value take : 'a queue -> 'a (* peut lancer Empty *)
value peek : 'a queue -> 'a (* idem *)

```

## 1.2 Version Persistante

Voyons d'abord comment écrire une file persistante simplement. Une version naïve serait d'utiliser une liste de la manière suivante :

```

type 'a queue = 'a list;;
exception Empty;;

let empty = [];;

let add q x = q @ [x];;

let take q = match q with
| [] -> None
| x :: q' -> Some (x, q');;

let peek q = match q with
| [] -> None
| x :: _ -> Some x;;

```

mais malheureusement le coût de `add` est prohibitif (linéaire pour un simple ajout d'élément). Il faut donc éviter cette version à tout prix.

À la place, nous allons utiliser *deux* listes pour garantir une meilleure complexité. La première liste, `front`, contient les éléments du début de la file, dans l'ordre. La seconde liste, `rear`, contient les éléments de la fin de la file dans l'ordre inverse.

```

type 'a queue = {
  front : 'a list;
  rear : 'a list;
} ;;
(* invariant: si front=[] alors rear=[] aussi *)

let empty = { front=[]; rear=[]; };;

let is_empty q = q.front = [] ;;

(* construire une file qui respecte l'invariant *)
let make f r = match f with
| [] -> {front=rev r; rear=[]; }
| _ -> {front=f; rear=r; } ;;

```

```

let add q x = make q.front (x::q.rear) ;;

let take q = match q.front with
| x :: front' ->
    let q' = make front' q.rear in
    Some (x, q')
| [] -> None ;;

let peek q = match q.front with
| [] -> None
| x :: _ -> Some x;;

```

Cette structure respecte plusieurs invariants. Prenons une file  $q$ , alors nécessairement :

- si  $q.\text{front} = []$  alors  $q.\text{rear} = []$  aussi. Cet invariant est maintenu par la fonction `make`, au prix d'un renversement de liste occasionnel ;
- la liste des éléments contenus dans  $q$ , dans l'ordre, est toujours  $q.\text{front} @ \text{rev } q.\text{rear}$  (on peut prouver la préservation de cet invariant pour chaque opération).

**Exercice** : en n'utilisant que la spécification des files persistantes, écrire une fonction qui calcule la longueur d'une file.

**Exercice** : écrire une fonction qui renverse une file d'attente. On pourra utiliser une liste ou une pile.

**Exercice** : écrire une fonction `chercher : 'a -> 'a queue -> bool` qui renvoie `true` ssi l'élément apparaît dans la file.

**Exercice**<sup>2</sup> : on appelle *liste circulaire* une liste sans fin, qui boucle sur un nombre fini d'éléments.

1. définir une telle structure en vous inspirant des files persistantes.
2. écrire une fonction suivant qui décale la liste d'un cran (obtenant le premier élément et la suite de la liste).
3. écrire une fonction qui ajoute un élément en tête d'une liste circulaire.
4. écrire une fonction qui, étant donné une liste circulaire  $l$  et une valeur sentinelle  $n$  n'apparaissant pas dans  $l$ , calcule la taille de  $l$ .
5. écrire une fonction qui ajoute un élément à la fin d'une liste circulaire de manière aussi efficace que possible.

## Complexité

La file de priorité persistante présente une complexité en  $O(1)$  *amorti* sur les opérations d'ajout et de retrait. Plus précisément, un appel à `take` ou `add` peut demander  $O(n)$  opérations ( $n$  le nombre d'éléments), mais une séquence de  $m$  opérations prendra asymptotiquement  $O(m)$  opérations ; le coût de chaque opération est amorti par le reste de la séquence, d'où le  $O(1)$  amorti.

Pour une preuve de cette complexité, voir le livre de référence sur les structures persistantes, qui est sans conteste celui d'Okasaki. Il présente de nombreuses

2. merci à Jean-Loup Carré dont je me suis fortement inspiré sur ces exercices.

structures persistantes avec des exemples en ML, dont cette structure de file (page 15). Intuitivement, il s'agit de montrer que pour chaque élément on ne doit payer le coût de renversement de liste qu'une seule fois.

### 1.3 Structure Impérative

Là encore, tout comme pour les piles, nous allons traiter un cas simple où la file est de longueur bornée car la structure complète est plus difficile (même si très utile en pratique). L'idée est d'utiliser un tableau dans lequel deux indices se « poursuivent » : l'un dénote le début de la file, l'autre la fin (première case vide). Notez la similarité avec les piles implémentées par des tableaux.

```

type 'a queue = {
  tab : 'a vect;
  mutable start : int;
  mutable stop : int;
} ;;

exception Empty;;
exception Full;;

let create x = { tab=make_vect 256 x; start=0; stop=0; };;

let is_empty q = q.start = q.stop;;

(* indice suivant dans le tableau circulaire *)
let succ q i =
  if i+1 = vect_length q.tab then 0 else i+1 ;;

let is_full q =
  succ q q.stop = q.start ;;

let add q x =
  if is_full q then raise Full;
  q.tab.(q.stop) <- x;
  q.stop <- succ q q.stop ;;

let take q =
  if q.start = q.stop then raise Empty;
  let x = q.tab.(q.start) in
  q.start <- succ q q.start;
  x

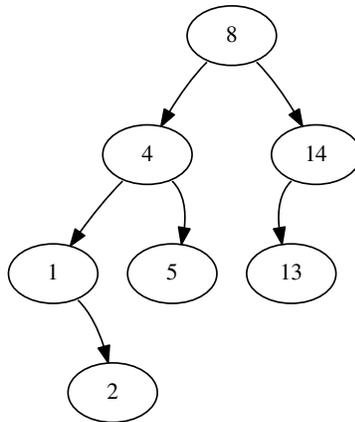
```

Là encore les opérations se font en temps constant. Dans une implémentation complète, cependant, l'insertion dans une file pleine déclenche un redimensionnement du tableau, qui se fait en temps linéaire; on obtient alors un ajout en temps constant amorti.

**Exercice** : écrire une fonction de copie de file d'attente impérative, qui préserve son argument. Là encore on veillera à n'utiliser que l'interface des files, et pas les spécificités de l'implémentation ci-dessus.

**Exercice** : reprendre, dans le TP n° 6, les fonctions de piles, et les adapter aux files impératives.

**Exercice** : On appelle *parcours en largeur d'abord* le parcours d'arbre qui explore tous les nœuds d'une profondeur donnée avant leurs enfants. Par exemple, sur l'arbre suivant :



Le parcours en largeur d'abord va renvoyer la liste [8; 4; 14; 1; 5; 13; 2]. Implémenter ce parcours à l'aide d'une file.

## 2 Files de Priorité

Les *files de priorité* ressemblent aux files d'attente mais l'ordre de sortie des éléments dépend de leur *priorité* (et non de leur ordre d'insertion). Nous choisirons là encore le terme anglais de *heap* (« tas »). Ces files de priorité sont très utiles pour l'ordonnancement de tâche par priorité, et pour certains algorithmes tels que la recherche de plus court chemin dans un graphe (algorithme de Dijkstra que vous verrez l'an prochain).

Voyons l'interface des files de priorité impératives (là encore la différence avec la version persistante concerne l'ajout et le retrait d'éléments). Une implémentation Caml existe dans la bibliothèque standard.

```
type 'a heap
exception Empty
value create : ('a -> 'a -> bool) -> 'a heap
value add : 'a heap -> 'a -> unit
value take : 'a heap -> 'a
```

La principale différence au premier coup d'œil concerne la création d'une file de priorité. En effet, on doit fournir une fonction  $f$  de comparaison d'éléments, telle que  $f\ x\ y$  vaut *true* si et seulement si la priorité de  $x$  est inférieure ou égale à celle de  $y$ . Les éléments sont enlevés de la file de priorité par priorité croissante

(c'est-à-dire que les éléments avec une petite priorité sont enlevés en premier, même s'ils ont été ajoutés plus tard). Exemple avec des entiers :

```
let h = create (fun i j -> i <= j);;

add h 10;;
add h 3;;
add h 5;;
add h 1;;

take h;; (* 1 *)
take h;; (* 3 *)
take h;; (* 5 *)
take h;; (* 10 *)
take h;; (* lance l'exception Empty *)
```

On peut, si la structure de file de priorité fournit une insertion et extraction en  $O(\ln(n))$  (avec  $n$  le nombre d'éléments du tas), en déduire un tri de liste en  $O(n \ln(n))$  :

```
let rec make_list h =
  if is_empty h then []
  else let x = take h in x :: make_list h ;;

let heap_sort f l =
  let h = create f in
  (* ajouter les elements de l dans h *)
  do_list (add h) l;
  (* construire une liste depuis h *)
  make_list h [] ;;
```

Cet algorithme (dit « tri par tas » ou « heap sort ») fonctionne en deux temps. D'abord, on crée un tas avec la fonction de comparaison servant à trier (de sorte que les éléments petits dans l'ordre auront une faible priorité) et on ajoute les éléments à trier dans ce tas ; ensuite, on extrait les éléments du tas un par un et on les ajoute à une liste dans l'ordre. Comme l'extraction d'éléments procède par priorité croissante, on obtient bien une liste triée. Pour chaque élément on paie une insertion et une extraction qui sont chacun, par hypothèse, en  $O(\ln(n))$ , donc le total est bien en  $O(n \ln(n))$ . La fonction `do_list : ('a -> unit) -> 'a list -> unit` sert à appliquer une fonction à tous les arguments d'une liste (notez encore une fois l'usage de la curryfication dans `heap_sort`). Elle est fournie dans Caml, mais peut s'implémenter ainsi si besoin :

```
let rec do_list f l = match l with
| [] -> ()
| x :: l' -> f x ; do_list f l' ;;
```

### 3 Tableaux Associatifs

Les tableaux associatifs (ou « maps », ou « dictionnaires ») sont absolument fondamentaux en informatique. Certains langages, comme JavaScript, Python ou Lua, l'utilisent comme brique de base du langage (ainsi, les objets python

sont en fait des dictionnaires. Python fournit également un type de dictionnaire très utilisé).

### 3.1 Version Persistante : Arbres de Recherche

Nous avons en fait déjà vu la version persistante des dictionnaires dans le cadre des arbres de recherche équilibrés, au cours précédent. Là encore Caml fournit le module standard `map` qui utilise des arbres AVL (arbres binaires de recherche équilibrés). L'interface persistante « usuelle »<sup>3</sup> requiert une fonction de comparaison d'éléments (ici, nous nous contenterons de la fonction de comparaison de Caml par soucis de simplicité). Voyons ici une version sans exceptions (celle de Caml utilise l'exception `Not_found`) :

```
type ('a,'b) map (* association de 'b aux cles 'a *)  
  
value empty : ('a,'b) map  
value add : 'a -> 'b -> ('a,'b) map -> ('a,'b) map  
value remove : 'a -> ('a,'b) map -> ('a,'b) map  
value find : 'a -> ('a,'b) map -> 'b option
```

Nous voyons ici que les opérations principales définissent

- une *map* vide (un dictionnaire persistant vide) ;
- des fonctions pour y ajouter ou retirer des associations clé/valeur (la clé ayant le type 'a et la valeur le type 'b) ;
- une fonction pour chercher la valeur associée à une clé (la fonction `find`) si elle existe.

Les fonctions `add` et `remove` nécessitent d'équilibrer l'arbre, nous nous contenterons de rappeler la fonction de recherche dans un arbre de recherche de type ('a,'b) map, de complexité proportionnelle à la hauteur de l'arbre au pire cas :

```
type ('a, 'b) map =  
  | Empty  
  | Node of 'a * 'b * ('a,'b) map * ('a, 'b) map;;  
  
let rec find x map = match map with  
  | Empty -> None  
  | Node (y, v, l, r) ->  
    if x=y then Some v  
    else if x<y then find x l  
    else find x r ;;
```

### 3.2 Version Impérative : Table de Hachage

La version impérative usuelle est celle des « tables de hachage » (en anglais « hash table »). Nous ne ferons ici qu'effleurer ce sujet qui est très vaste. Vous pouvez aller voir sur Wikipédia pour plus de détails sur les multiples variantes. Cette structure est disponible (avec quelques variations) dans le module `hashtbl`.

---

3. La fonction de création de dictionnaire dépend de l'algorithme sous-jacent à l'implémentation.

## Interface

L'interface impérative habituelle des tables de hachage est la suivante (elle est assez proche de celle des tableaux, mais avec un type 'a pour clé plutôt que des entiers).

```
type ('a,'b) hashtbl
exception Not_found (* predefini *)

value create : int -> ('a,'b) hashtbl
value add : ('a,'b) hashtbl -> 'a -> 'b -> unit
value remove : ('a,'b) hashtbl -> 'a -> unit
value find : ('a,'b) hashtbl -> 'a -> 'b (* ou Not_found *)
```

On note que la fonction de création de nouvelle table demande une taille initiale en argument. L'exception standard `Not_found` est lancée lorsque `find h x` est appelée sur `h` alors que `x` n'y apparaît pas.

## Algorithme

On présente ici l'implémentation basique (là encore sans redimensionnement). Le principe fondamental est d'utiliser une *fonction de hachage* qui associe chaque clé à un entier, de manière imprévisible (c'est-à-dire avec une distribution la plus uniforme possible) mais déterministe<sup>4</sup>. Cet entier est ensuite utilisé pour savoir dans quel case de la table (une table de hachage est un tableau) la valeur devrait se trouver. Pour gérer les collisions (clés différentes mais avec le même hash, ce qui peut toujours arriver : par exemple le type `string` comprend un nombre infini de valeurs, alors que la fonction de hachage renvoie un entier parmi  $[-2^{62}, 2^{62} - 1]$ ) on utilise des listes d'association. La complexité des opérations d'insertion ou de recherche (`find`) est en  $O(1)$  amorti si on suppose que la fonction de hachage est parfaite, et que les clés sont distribuées uniformément<sup>5</sup>.

```
type ('a,'b) hashtbl = ('a * 'b) list vect;;

let create n = make_vect n [];;

(* ajout dans une liste d'association *)
let rec add_list x y l = match l with
| [] -> (x,y) :: []
| (x',y') :: l' ->
    if x=x' then (x,y)::l' else (x',y') :: add_list x y l';;

(* indice de x dans la table *)
let indice h x =
  let i = hashtbl__hash x in
  i mod (vect_length h) ;;

(* deletion dans une liste d'association *)
```

---

4. Voir [https://fr.wikipedia.org/wiki/Fonction\\_de\\_hachage](https://fr.wikipedia.org/wiki/Fonction_de_hachage). Il existe aussi des fonctions de hachage dites *cryptographique* qui servent à signer des documents (on ne peut contrefaire le document sans modifier son hash, ce qui éveille la suspicion du destinataire).

5. les fonctions de hachage non parfaites, en pratique, peuvent poser de vrais problèmes, notamment de déni de service sur le web, voir ici et là.

```

let rec remove_list x l = match l with
| [] -> []
| (x',y') :: l' ->
    if x=x' then l' else (x',y') :: remove_list x l' ;;

(* ajouter x->y dans la table *)
let add h x y =
    let i = indice h x in
    h.(i) <- add_list x y h.(i) ;;

(* enlever x de la table *)
let remove h x =
    let i = indice h x in
    h.(i) <- remove_list x h.(i) ;;

(* retrouver la valeur associee a x. assoc cherche un
    element dans une liste d'association. *)
let find h x =
    let i = indice h x in
    assoc x h.(i) ;;

```

### Exemple d'utilisation

Voyons un cas simple d'utilisation d'une table de hachage : compter le nombre de fois que chaque mot apparait dans un texte (qu'on suppose découpé en une liste de mots). Pour cela, nous allons utiliser une `(string,int) hashtbl` qui associe, à chaque mot vu au moins une fois, son nombre d'occurrences. Le fichier `mots.txt` se trouve ici ; notez que le découpage en mots est très naïf au regard de la ponctuation.

```

(* compte, dans la table h, combien de fois chaque mot apparait *)
let compter texte =
    let h = create 42 in
    do_list
        (fun mot ->
            let n =
                try find h mot
                with Not_found -> 0 in
            add h mot (n+1)
        )
    texte;
    h;;

(* lit les lignes du fichier *)
let lire_fichier f =
    let i = open_in f in
    let l = ref [] in
    try
        while true do l := input_line i :: !l done; []
    with End_of_file -> rev !l ;;

let texte = lire_fichier "mots.txt" ;;
let h = compter texte ;;

find h "coeur";; (* 4 *)

```

```
find h "encensoir";; (* 2 *)  
find h "vilebrequin";; (* Not_found *)
```

**Exercice** : (*mémoïsation*) en utilisant une table de hachage pour sauvegarder les résultats intermédiaires, réécrire la fonction `fibonacci` naïve pour améliorer ses performances.

**Exercice** : écrire une fonction de complexité linéaire qui renvoie `true` ssi une liste contient des doublons.

**Exercice** : écrire une fonction `hrev h` qui à une table injective (aucune valeur n'est associée à deux clés ou plus) associe son inverse (si  $x \mapsto y \in h$ ,  $y \mapsto x \in \text{hrev}(h)$ ). On pourra utiliser la structure interne de la table pour la parcourir.