Structure séquentielle

Table des matières

1 Type abstrait et implémentation

1.1 Types abstraits liste et vecteur

Un type de donnée est avant tout spécifié par une liste d'opérations élémentaires permettant de créer, et éventuellement de modifier, un objet du type en question ainsi que d'accéder aux éléments qu'il contient. Par exemple, on appellera usuellement « liste » et « vecteur » une structure de données fournissant les opérations élémentaires suivantes :

Opération	Type	Effet
`[]`	`'a list`	Liste vide
`cons x m`	`'a -> 'a list -> 'a list`	Rajoute un élément en tête
`tail m`	`'a list -> 'a list`	Renvoie la liste sauf son premier élément
`head m`	`'a list -> 'a`	Renvoie le premier élément de la liste

FIGURE 1 – Signature d'un type « liste ».

Opération	Type	Effet
`create n x`	`int -> 'a -> 'a array`	Renvoie un vecteur de taille n initialisé à x
`get t i`	`'a array -> int -> 'a`	Renvoie l'élément d'indice i
`set t i x`	`'a array -> int -> 'a -> unit`	Modifie l'élément d'indice i en place

FIGURE 2 – Signature d'un type « vecteur ».

Ces « signatures » ne précisent pas la manière dont est réalisée la structure, ce qui est voulu : tant que la spécification est respectée, la réalisation concrète peut être changée sans que le code utilisant la structure n'ait à être réécrit. Cependant, on précise normalement la complexité des opérations élémentaires : une structure qui ne permet pas l'accès à un élément arbitraire en temps constant ne sera généralement pas considérée comme un vecteur. On obtiendrait alors :

Opération	Coût
`cons x m`	$O(1)$
`tail m`	$O(1)$
`head m`	$O(1)$

FIGURE 3 – Coût des opérations sur une « liste ».

Opération	Coût
`create n x`	$O(n)$
`get t i`	$O(1)$
`set t i x`	$O(1)$

FIGURE 4 – Coût des opérations sur un « vecteur ».

Si l'on connaît les opérations élémentaires fournies et leur coût, l'implémentation sous-jacente n'a que peu d'importance pour l'« utilisateur » de la structure de donnée : c'est tout l'intérêt du concept de type abstrait de données. Cependant, vous êtes censés savoir comment ces structures sont réalisées en OCaml (listes simplement chaînées et tableaux, respectivement).

1.2 Définition en OCaml

Il n'est pas possible de redéfinir le type `array` en OCaml (il faut avoir accès à la représentation mémoire des données). En revanche, le type `list` se définit très simplement :

```
type 'a liste =
  | Vide
  | Cons of 'a * 'a liste

(* Pour définir l'équivalent de [1; 3; 7] : *)
let u = Cons(1, Cons(3, Cons (7, Vide)))

let rec somme u =
  match u with
  | Vide -> 0
  | Cons (x, xs) -> x + somme xs
```

```
# somme u;;
- : int = 11
```

C'est un type *récursif* : il apparaît des deux côtés du signe `=` dans sa définition. Ce sera le cas à chaque fois que l'on définira un nouveau type de données (permettant de stocker un nombre arbitraire d'éléments) sans réutiliser les types `list` ou `array`.

Remarque

⇨ Le type `'a liste` défini ci-dessus est *rigoureusement* équivalent au type `'a list` prédéfini : après compilation, on obtient exactement les mêmes instructions machine.

1.3 Modèle mémoire

On peut voir la mémoire (vive) d'un ordinateur comme un immense tableau dont les cases sont numérotées de 0 à $N-1$, où N est de l'ordre de quelques milliards (ou nettement plus).

— Le numéro d'une case est appelé *adresse mémoire*. Il s'agit d'un entier de taille fixée. Cette taille est celle d'un *mot machine*, et correspond également à la taille des *entiers machine*, c'est-à-dire des entiers que le processeur est capable de traiter en une opération (pour faire une addition par exemple). La taille d'un mot machine fait 64 bits sur la grande majorité des ordinateurs actuels (32 bits parfois).

— Les cases ont toutes la même taille (elles peuvent toutes contenir la même quantité de données) : un octet, c'est-à-dire 8 bits. Cependant, nous considérerons pour simplifier que chaque case fait un mot machine – sur un ordinateur actuel, cela revient à regrouper les cases par paquets de 8. Ainsi, une case peut contenir l'adresse d'une autre case (on parle de « pointeur »).

— Accéder (en lecture ou en écriture) à une case d'adresse donnée est une opération élémentaire qui se fait en temps constant.

1.4 Réalisation concrète en OCaml

Tableau

— Un `'a array` est représenté en OCaml par une série de cases mémoire consécutives. La « valeur » d'un tableau `u` est en fait l'adresse de la première de ces cases :
  — quand on passe un `t : 'a array` comme argument à une fonction, on passe en fait l'*adresse* du tableau `t` ;
  — si l'on fait `let v = t in ...`, alors le nom `v` est associé à la même adresse que le nom `t`.

— Pour obtenir l'adresse de la case `t.(i)`, il suffit de prendre l'adresse de `t` et de lui ajouter `i` (à quelques détails près) : on peut donc bien accéder à une case quelconque en temps constant (une addition et une lecture mémoire).

— Pour que cette méthode fonctionne, il est indispensable que toutes les cases du tableau fassent la même taille : en OCaml, ce sera systématiquement un mot machine.

— Si un objet de type `'a` ne tient pas dans un mot machine, les cases d'un `'a array` contiendront donc en fait des pointeurs vers les éléments du tableau.

— La taille du tableau est stockée juste avant les données ; cela permet d'avoir une fonction `Array.length` (en temps constant, de plus) et de détecter quand un accès à `t.(i)` est « hors bornes » (en comparant `i` à la longueur de `t`).

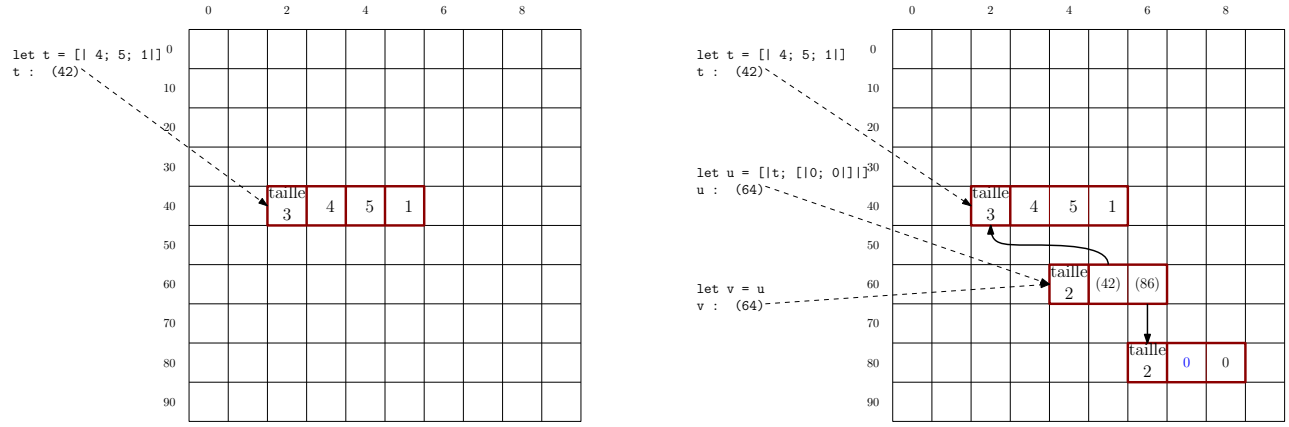


FIGURE 5 – Représentation mémoire des array en OCaml.

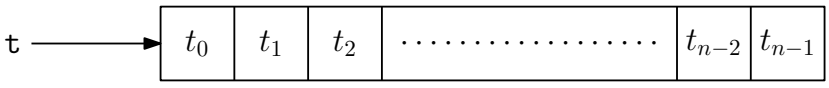


FIGURE 6 – Représentation simplifiée d'un array.

Remarques

⇨ On peut être plus précis sur le calcul de l'adresse de la i-ème case d'un tableau `t` : sachant que chaque case d'un tableau OCaml occupe 8 octets et qu'il y a un bloc de 8 octets au début du tableau contenant (entre autres) la taille du tableau, on a

$$\text{adresse}(t_i) = \text{adresse}(t) + 8(i+1).$$

⇨ En C, on ne sera pas limité à des « cases » de 8 octets : cela ne pose pas de problème (il faut juste remplacer le 8 dans le calcul ci-dessus) mais il est indispensable que toutes les cases d'un tableau donné fassent la même taille.

⇨ Être capable de détecter les accès « hors bornes » à un tableau est très important et il est nécessaire que le tableau « connaisse sa taille » si l'on veut pouvoir *garantir* cette détection. Si on ne le fait pas, alors un accès à la $10^{\text{ème}}$ case d'un tableau de taille 5 résulte en un accès mémoire à une case n'ayant rien à voir avec le tableau. Plusieurs choses peuvent alors se produire :
  — Si l'on a de la chance, cet accès est interdit par le système, car la zone mémoire en question n'appartient pas au programme : on a donc un plantage immédiat avec une *segmentation fault*.
  — Si l'on a moins de chance, la lecture ou l'écriture a lieu et l'on a donc une corruption silencieuse du programme : il finira sans doute par planter on se sait trop quand, après avoir fait on ne sait trop quoi.
  — Si l'on a *vraiment* pas de chance, cet accès illégal n'a pas été fait par hasard, mais bien pour accéder à des données qui devraient être protégées : on peut ainsi par exemple modifier l'adresse de retour d'une fonction pour exécuter du code arbitraire. C'est l'exemple le plus classique de faille de sécurité : on parle de *buffer overflow*.

Liste simplement chaînée

— Une liste est constituée de *cellules*, et chaque cellule est constituée de deux cases : l'une contenant l'élément de la liste, l'autre l'adresse de la cellule suivante.
— S'il n'y a pas de cellule suivante (parce qu'on est au bout de la liste), on mettra une valeur particulière dans la deuxième case (zéro, en pratique).
— La « valeur » de la liste est l'adresse de la cellule de tête.
— Si `u` est une `'a list` et `x : 'a`, la liste `x :: u` est obtenue en créant une nouvelle cellule constituée de l'élément `x` et d'un pointeur vers la liste `u` (c'est-à-dire vers la cellule de tête de `u`).

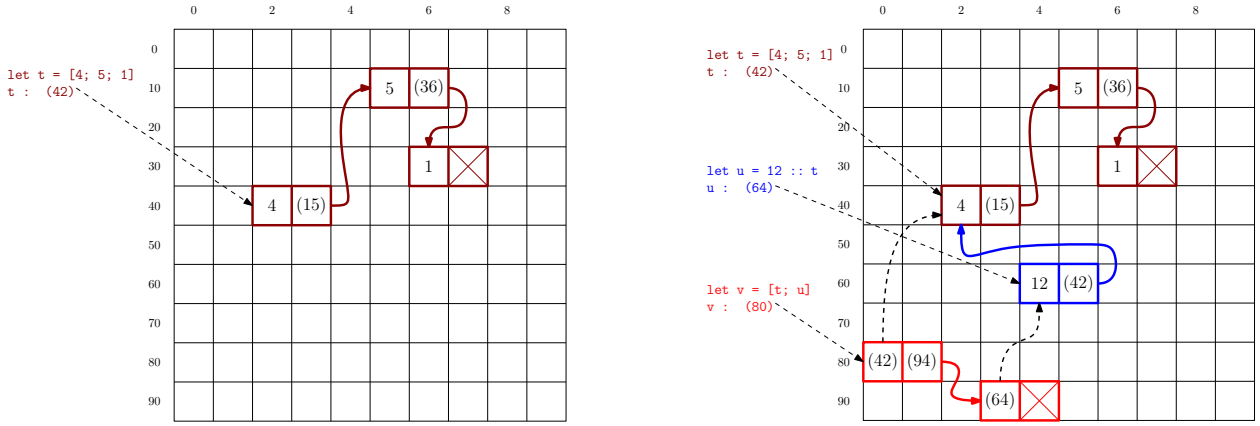


FIGURE 7 – Représentation mémoire des list en OCaml.

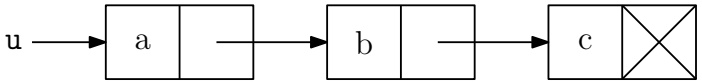


FIGURE 8 – Représentation simplifiée d'une liste u = [a ; b ; c].

Remarques

⇨ Connaître une liste `u`, c'est simplement connaître l'adresse de sa première cellule : par conséquent, le seul moyen d'accéder au i-ème élément d'une liste est de suivre i pointeurs, ce qui prendra un temps proportionnel à i.

⇨ De même, le calcul de la longueur prend un temps proportionnel à la longueur de la liste : il faut suivre les pointeurs jusqu'à arriver a la liste vide.

⇨ Une liste consomme plus d'espace qu'un tableau contenant les mêmes éléments puisqu'il faut stocker un pointeur supplémentaire par élément. Cependant, cela ne changera rien aux complexités spatiales puisqu'on négligera les facteurs constants.

⇨ En réalité, une cellule de liste occupe *trois* cases en OCaml, la troisième étant à usage interne : elle sert au *garbage collector* (*ramasse-miettes* en français) pour libérer la mémoire quand elle n'est plus utilisée.

Une remarque sur les « listes » de Python : La réalisation de ces deux structures en OCaml est très classique, mais c'est loin d'être la seule possible. En particulier, le type `list` de Python ressemble à la fois à ce que l'on a ici appelé « liste » et « vecteur », mais son implémentation est significativement différente de celle des `array` de OCaml (et n'a rien à voir avec celle des `list`). On parle généralement de *tableau dynamique* pour désigner le type de données abstrait correspondant aux `list` de Python (et aussi aux `std::vector` du C++, entre autres).

1.5 Structure de données fonctionnelle et impérative

Une structure de donnée est dite *fonctionnelle* (ou *persistante*) si elle ne peut pas être modifiée *en place* ; dans le cas contraire, elle est dite *impérative*, ou *mutable*. La spécification donnée plus haut pour les listes est fonctionnelle : les opérations `cons` et `tail` ne modifient pas la liste passée en entrée, mais renvoient de nouvelles listes. C'est tout le contraire pour la spécification « vecteur ».

Remarque

⇨ Le type `list` de Python satisfait simultanément les spécifications « liste » et « vecteur », à ceci près qu'il est impératif, et que l'ajout des éléments se fait en queue : si on exécute `m.append(x)`, `m` est modifiée.

Les structures de données impératives ont un intérêt majeur : il est souvent plus facile d'écrire des algorithmes *efficaces* en utilisant ces structures. En particulier, il n'existe pas de réalisation fonctionnelle de la structure de « vecteur » permettant l'accès à une case arbitraire en temps constant, ce qui impose parfois un facteur $\log n$ supplémentaire pour la complexité si l'on ne souhaite utiliser que des structures fonctionnelles.

L'intérêt des structures fonctionnelles est double :

— Il est souvent plus facile d'écrire des programmes *corrects* si on limite au maximum les effets de bord (en particulier, on n'a plus de problème d'*aliasing*).
— Plusieurs objets peuvent partager une partie de leur représentation mémoire.
— En particulier, on peut garder une trace des précédentes versions de la structure pour un coût moindre que dans le cas mutable : si l'on fait un `append` en Python, l'ancienne version de la liste est perdue. Le seul moyen d'en garder une trace (ce qui est assez souvent utile, comme nous aurons l'occasion de le voir) est d'en faire une copie, pour un coût en $O(n)$ tant en temps qu'en espace. En Caml, un `let u = x :: v` laisse v accessible tout en ayant un coût unitaire. L'équivalent Python, `u = v + [x]`, a un coût linéaire en la taille de v.

Exercices 1

⇨ On considère deux listes `u = [1; 2 ; 3]` et `v = [21; 22; 23]`. Faire les schémas mémoire (simplifiés) correspondant à :

1. `let w = 0 :: u;`
2. `let a = w @ v;`
3. `let b = u @ a;`
4. `let c = [a; b].`

⇨ On considère la fonction suivante :

```
let rec f u =
  match u with
  | [] -> [[]]
  | x :: xs -> u :: f xs
```

1. Quel est le type de `f` ?
2. Que renvoie cette fonction si on l'appelle sur `[1; 2; 3]` ?
3. Quelle est la complexité en temps de `f`, en fonction de $|u|$?
4. Faire un schéma mémoire correspondant au résultat renvoyé pour `u = [1; 2; 3]`.
5. Combien d'espace le résultat renvoyé occupe-t-il en mémoire en fonction de la longueur $|u|$ de l'argument (on supposera que chaque élément de la liste de départ u occupe un espace unitaire) ? En quoi cela peut-il être surprenant ?

Dans le cas d'une structure de données fonctionnelle, le fait que plusieurs objets partagent tout ou partie de leur représentation mémoire ne pose jamais de problème. En revanche, pour une structure impérative, il convient d'être très prudent !

Exercice 2

⇨ On souhaite écrire une fonction `nulle : int -> int -> float array array` permettant de créer une matrice nulle dont les dimensions sont passées en argument.

1. On propose le code suivant :

```
let nulle n p =
  let ligne = Array.make p 0. in
  Array.make n ligne
```

   (a) Quelle est la complexité de cette fonction ?
   (b) Faire un schéma de la représentation mémoire de `nulle 2 3`.
   (c) Quel est le problème ?

2. Reprendre la question précédente dans chacun des cas suivants :

```
let nulle_2 n p =
  Array.make n (Array.make p 0.)

let nulle_3 n p =
  let t = Array.make n [| |] in
  for i = 0 to n - 1 do
    t.(i) <- Array.make p 0.
  done;
  t

let nulle_4 n p =
  Array.init n (fun i -> Array.make p 0.)
```

Ce problème se présentant assez souvent, OCaml propose une fonction prédéfinie `Array.make_matrix : int -> int -> 'a -> 'a array array` permettant de créer et initialiser proprement une « matrice ».

2 Pile

2.1 Principe

Une pile (*stack* en anglais) est une structure de données dite LIFO (*last in, first out*, ce qu'on traduit le plus souvent par *premier arrivé, dernier servi*), qui doit son nom à l'analogie avec une pile d'assiettes. Les deux opérations élémentaires sont :

— rajouter une assiette au sommet de la pile ;
— récupérer l'assiette se trouvant actuellement au sommet (qui est donc la dernière à avoir été empilée). Bien évidemment, cette opération échouera si la pile est vide.

Dans la pile ci-dessous, l'élément au sommet (3) est le dernier ajouté, et sera le prochain à être retiré (à condition qu'on n'en ajoute pas d'autre avant de faire une extraction).

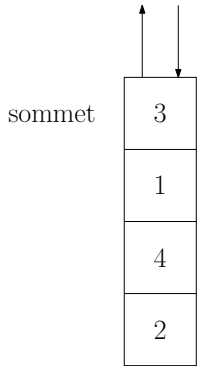


FIGURE 9 – Un exemple de pile contenant quatre entiers.

2.2 Pile fonctionnelle

Opération	Type	Signification
`empty_stack`	`'a stack`	Pile vide
`is_empty`	`'a stack -> bool`	Teste si la pile est vide
`push`	`'a -> 'a stack -> 'a stack`	Rajoute un élément au sommet
`pop`	`'a stack -> 'a * 'a stack`	Récupère le sommet et le reste de la pile

Remarque

⇨ On se convainc facilement que cette interface est parfaitement équivalente à celle du type abstrait *liste* : `push` a la même spécification que `cons` et `pop u` renvoie le couple `(List.head u, List.tail u)`. Il n'y a donc rien de nouveau à dire, et l'on utilisera simplement des `'a list` OCaml pour réaliser cette structure.

2.3 Pile impérative

2.3.1 Interface

Opération	Type	Signification
`empty_stack`	`unit -> 'a stack`	Crée une nouvelle pile vide
`is_empty`	`'a stack -> bool`	Teste si la pile est vide
`push`	`'a -> 'a stack -> unit`	Empile un élément (modifie la pile)
`pop`	`'a stack -> 'a`	Dépile un élément (modifie la pile)

Remarques

⇨ Attention à la fonction `pop` : son type de retour est `'a` (et pas `unit`) mais elle a bien un effet de bord (l'élément est retiré de la pile). Ainsi, deux appels successifs `pop s` renverront en général deux valeurs différentes.

⇨ Dans le cas d'une pile impérative, `empty_stack` est une fonction de type `unit -> 'a stack`, et pas juste une constant de type `'a stack` comme dans le cas fonctionnel. Pourquoi ?

On peut alors écrire une fonction `somme` :

```
let somme pile =
  let s = ref 0 in
  while not (is_empty pile) do
    let x = pop pile in
    s := !s + x
  done;
  !s
```

Remarque

⇨ Cette fonction a un *énorme* inconvénient. Lequel ? En réalité, on n'utilisera que très rarement les piles de cette manière (parcours de toute la pile). Il sera bien plus fréquent d'avoir des insertions et des extractions entrelacées, comme nous aurons l'occasion de le voir dans les chapitres suivants.

2.3.2 Implémentation

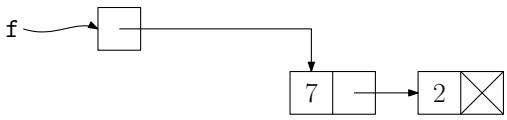
Version à base de liste On peut très facilement réaliser une pile impérative en utilisant une `'a list ref`.

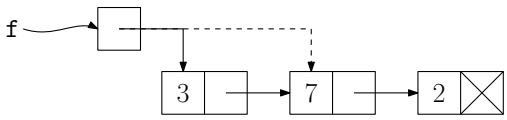
```
type 'a pile = 'a list ref
```

```
let empty_stack () = ref []

let push x s = (s := x :: !s)
```

```
let pop s =
 match !s with
 | [] -> failwith "pop from empty stack"
 | x :: xs ->  s := xs; x
```

Si l'on souhaite un schéma-mémoire, il faut juste comprendre qu'une `'a list ref` est simplement une case mémoire qui contient un pointeur *mutable* vers une liste :





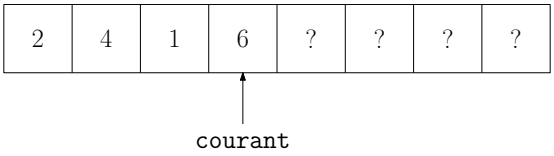
Pile dans un tableau Pour la deuxième version, il faut, à la création de la pile, définir une capacité maximale. On crée alors un tableau ayant cette taille, et l'on maintient à jour l'indice `courant` du sommet actuel :

— Pour ajouter un élément, on vérifie que la pile n'est pas pleine, on incrémente `courant` et l'on écrit le nouvel élément.

— Pour extraire un élément, on vérifie que la pile n'est pas vide, on récupère l'élément à l'indice `courant` et l'on décrémente `courant`.

En OCaml, on pourrait utiliser le type suivant :

```
type 'a pile = {donnees : 'a array; mutable courant : int; capacite : int}
```



Remarques

⇨ Les valeurs des cases situées à droite de `courant` n'ont aucune importance.

⇨ Si l'on utilise le type proposé plus haut, il faut disposer d'un élément de type `'a` au moment de la création car il est nécessaire d'initialiser le tableau. On parle de *témoin de type*. Une solution à ce problème est d'utiliser un tableau de `'a option`, que l'on initialise à `None`.

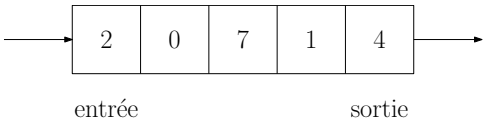
⇨ Il est possible de se libérer de la contrainte de capacité bornée en utilisant un tableau *dynamique*, c'est-à-dire redimensionnable. C'est ce qui est fait dans le type `list` de Python.

```
let empty_stack (x : 'a) (r : int) : 'a pile =
  {donnees = Array.make r x; courant = -1; capacite = r}

let is_empty (p : 'a pile) : bool = (p.courant = -1)

let push (a : 'a) (p : 'a pile) : unit =
  if p.courant + 1 = p.capacite then failwith "La pile est pleine"
  else begin
    p.courant <- p.courant + 1;
    p.donnees.(p.courant) <- a
  end

let pop (p : 'a pile) : 'a =
  if p.courant = -1 then failwith "La pile est vide"
  else
    let t = p.donnees.(p.courant) in
    p.courant <- p.courant - 1;
    t
```

3 File

3.1 Principe et interface

Une file (*queue* en anglais) est une structure FIFO (*first in, first out*, qu'on pourrait traduire par *premier arrivé, premier servi*). On peut l'imaginer horizontale : on rajoute les éléments par la gauche, on les enlève par la droite. Cette fois, l'analogie est avec une file d'attente : les clients arrivent par la gauche, le guichet est à droite. Le prochain client servi sera celui qui attend depuis le plus longtemps.



Voici l'interface d'une file fonctionnelle.

Opération	Type	Signification
`empty_queue`	`'a queue`	File vide
`is_empty`	`'a queue -> bool`	Teste si la file est vide
`push`	`'a -> 'a queue -> 'a queue`	Rajoute un élément
`pop`	`'a queue -> 'a * 'a queue`	Récupère l'élément le plus ancien et le reste de la file

Voici celle d'une file impérative.

Opération	Type	Signification
`empty_queue`	`unit -> 'a queue`	Crée une file vide
`is_empty`	`'a queue -> bool`	Teste si la file est vide
`push`	`'a -> 'a queue -> unit`	Rajoute un élément
`pop`	`'a queue -> 'a`	Récupère l'élément le plus ancient (et l'enlève)

Remarques

⇨ Les noms des opérations sont moins fixés que pour les piles : on peut par exemple trouver `enqueue` pour celle que j'ai appelée `push` et `dequeue` pour celle que j'ai appelée `pop`.

⇨ À nouveau, attention à la version impérative de `pop` qui renvoie une valeur *et* a un effet de bord.

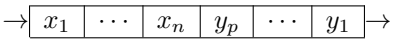
⇨ Sauf mention contraire, on écrira $(x_1, \dots, x_n)$ pour une file dont l'élément le plus ancien est x_n.

3.2 File fonctionnelle

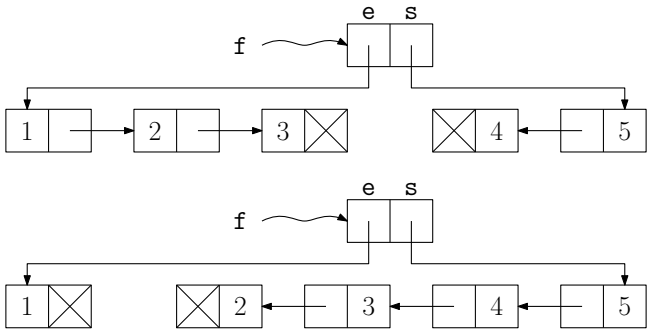
On peut implémenter naïvement une file à l'aide d'une liste, mais soit `push` soit `pop` sera alors en $O(n)$, suivant si l'on fait les insertions en queue ou en tête de liste. La solution classique est d'utiliser deux listes.

— Une que l'on appellera e pour « entrant » et qui correspondra à la partie « gauche » de la file.
— Une que l'on appellera s pour « sortant » et qui correspondra à la partie « droite » de la file, *à l'envers*.

Le couple $(e, s) = ([x_1; \dots; x_n], [y_1; \dots; y_p])$ correspondra alors à la file



— Pour ajouter un élément à la file, il suffit de le rajouter en tête de e : `push x (e, s) = (x :: e, s)`
— Pour extraire un élément :
  — si s est non vide, on extrait l'élément de tête de s : `pop (e, x :: xs) = (x, (e, xs))`
  — si s est vide et e non vide, on se ramène au cas précédent en remplaçant s par le miroir de e et e par la liste vide : `pop ([x_1;...;x_n], []) = pop ([], [x_n;...;x_1]))`
  — si les deux listes sont vides, l'opération échoue.



Exercice 3

⇨ 1. Donner une série d'opérations permettant d'aboutir à situation de la première des figures précédentes en partant d'une file vide.
2. Donner une série d'opérations permettant d'aboutir à situation de la seconde des figures précédentes en partant d'une file vide.
3. Compléter le tableau suivant.

Instruction	f		
	e	s	x
	`[1; 2; 3]`	`[5; 4]`	
`let x, f = pop f`			
`let x, f = pop f`			
`let f = push 6 f`			
`let x, f = pop f`			
`let x, f = pop f`			
`let f = push 7 f`			
`let x, f = pop f`			

L'intérêt de cette réalisation est qu'une opération coûteuse (l'extraction d'un élément alors que la liste s est vide et que la liste e contient n éléments) est compensée par une série d'opérations peu coûteuses. C'est le principe de la *complexité amortie* : au lieu de s'intéresser au coût maximal d'une opération, qui peut ici être proportionnel à la taille de la file, on majore le coût total d'une série de n opérations.

Proposition 3.1

Avec la réalisation décrite ci-dessus, le coût total d'une série de n opérations (insertions et extractions) sur une file initialement vide est en $O(n)$. Autrement dit, lors d'une série de n opérations, le coût moyen d'une opération est en $O(1)$.

Remarque

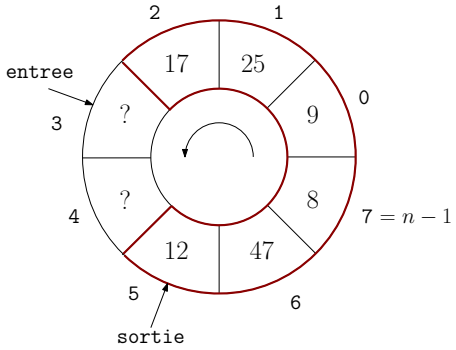
⇨ Autrement dit, lors d'une série de n opérations, le coût moyen d'une opération est en $O(1)$.

3.3 File impérative

Tableau circulaire On fixe une capacité maximale pour la file à la création et l'on adapte la réalisation vue pour les piles. Il y a deux modifications à apporter :

— Les *deux* extrémités de la file sont « actives ». Au lieu d'avoir un indice `courant` indiquant le sommet de la pile, on aura donc deux indices, disons `entree` et `sortie`, pour indiquer les deux extrémités.

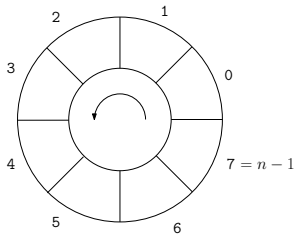
— Si l'on fait une série de « insertion + extraction » le nombre d'éléments dans la file va rester constant mais la zone dans laquelle ils sont stockés va se déplacer dans le tableau. Pour éviter de se retrouver bloqué, on rend le tableau *circulaire* en considérant les indices modulo sa taille.



Exercices 4

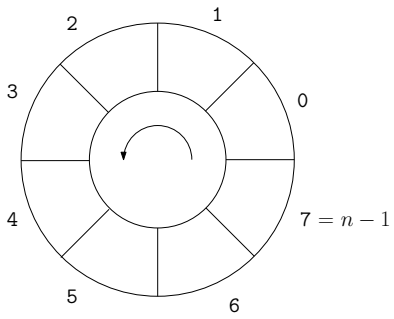
⇨ On exécute les instructions suivantes ; compléter le schéma.

```
push 10 f; print_int (pop f);
print_int (pop f); push 20 f
```



⇨ Compléter le schéma ci-dessous pour qu'il corresponde à l'état de f après les opérations suivantes :

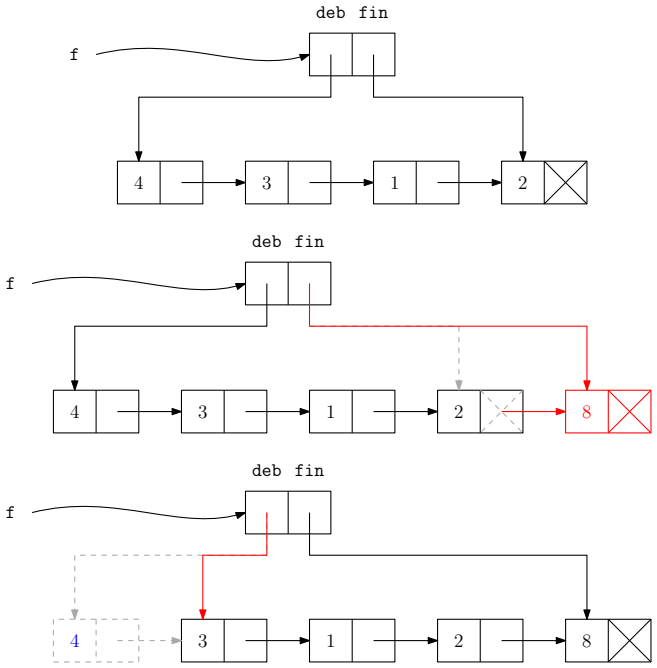
```
# push_left 3 f;;
# push_left 12 f;;
# pop_right
```



Liste mutable On adapte la réalisation des piles impératives avec des `'a list ref`. Comme les deux extrémités de la file sont « actives », la file contiendra deux pointeurs mutables : un vers la première cellule et un vers la dernière. Les cellules seront très similaires à celles d'une liste (un élément et un pointeur vers la cellule suivante), à ceci près que ce pointeur `next` sera *mutable*.

— Pour l'opération `pop`, on suit le pointeur `deb` pour accéder à la première cellule `c`, puis :
  — on récupère la valeur de l'élément `c.elt` ;
  — on modifie le pointeur `deb` de la file pour qu'il pointe vers la cible de `c.next`.

— Pour l'opération `push`, on suit le pointeur `fin` pour accéder à la dernière cellule `d`, puis :
  — on crée une nouvelle cellule avec l'élément voulu et une valeur particulière « vide » pour `next` ;
  — on modifie le pointeur `next` de la cellule `d` pour qu'il pointe vers cette nouvelle cellule ;
  — on modifie le pointeur `fin` de la file pour qu'il pointe vers cette nouvelle cellule.



Remarques

⇨ Cette réalisation de la structure de file est celle utilisée par le module `Queue` de OCaml, à quelques détails près.

⇨ On a passé sous silence quelques difficultés (file vide, file ne contenant qu'un seul élément, choix des types OCaml).

Exercice 5

⇨ 1. Expliquer pourquoi on pourrait facilement modifier la structure de donnée de manière à disposer d'une fonction `length` en temps constant, et pourquoi cette solution ne s'applique pas vraiment au type `list` de OCaml.

2. On a choisi de faire les extractions en « tête de liste » et les insertions en « queue ». Expliquer pourquoi c'est en fait la seule possibilité.

3.4 File de priorité

Une file de priorité est une structure dans laquelle chaque élément est inséré avec une certaine *priorité* (typiquement un entier ou un flottant, mais en tout cas un élément d'un ensemble totalement ordonné). Quand on extrait un élément, on ne récupère pas le plus ancien ou le plus récent mais *celui ayant la priorité la plus forte* (ou celle qui a la priorité la plus forte, selon les variantes).

Si l'on note `('a, 'b) pq` le type des files de priorité dont les éléments sont de type `'a` et les priorités de type `'b`, on obtient l'interface suivante pour une file de priorité impérative.

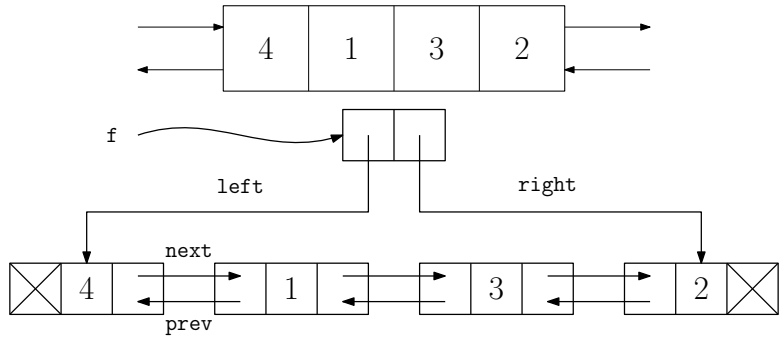
Opération	Type	Signification
`empty_pq`	`unit -> ('a, 'b) pq`	Crée une file vide
`insert`	`('a * 'b) -> ('a, 'b) pq -> unit`	Insère un élément avec une priorité
`extract_max`	`('a, 'b) pq -> ('a * 'b)`	Récupère l'élément de priorité maximale et l'enlève de la file

Il existe de nombreuses réalisations (fonctionnelles ou impératives) d'une file de priorité offrant des complexités en $O(\log n)$ (ou mieux) pour l'insertion et l'extraction du maximum. Nous aurons l'occasion d'utiliser cette structure, et d'en étudier des réalisations possibles, cette année.

3.5 Deque

Les *deques* (*Double ended queues*) sont une généralisation des files dans lesquelles on peut insérer et extraire par les deux extrémités :

Opération	Type	Signification
`empty_deque`	`unit -> 'a deque`	Crée une deque vide
`is_empty`	`'a deque -> bool`	Teste si la deque est vide
`push_left`	`'a -> 'a deque -> unit`	Rajoute un élément à gauche
`push_right`	`'a -> 'a deque -> unit`	Rajoute un élément à droite
`pop_left`	`'a deque -> 'a`	Récupère l'élément de gauche et l'enlève
`pop_right`	`'a deque -> 'a`	Récupère l'élément de droite et l'enlève



— Pour réaliser une deque impérative, il y a deux possibilités :
  — Adapter la version « tableau circulaire » des files impératives.
  — Adapter la version « liste mutable » des files impératives, ce qui demande d'utiliser des listes *doublement chaînées* (où chaque cellule contient un pointeur vers la cellule suivante et un vers la cellule précédente). L'idée est simple, mais il faut être soigneux pour la mettre en pratique.
— On peut réaliser une deque fonctionnelle efficace en adaptant la réalisation fonctionnelle des files à l'aide de deux listes, mais c'est très loin d'être évident.

4 Ensemble, dictionnaire

4.1 Ensemble

La structure de données abstraite Set correspond à la notion mathématique d'ensemble : il n'y a pas de répétition et les éléments ne sont pas ordonnés. Une signature possible est donnée ci-dessous, pour des ensembles *fonctionnels* :

Opération	Type	Commentaire
Opérations caractéristiques		
`member`	`'a -> 'a set -> bool`	Test d'appartenance
`add`	`'a -> 'a set -> 'a set`	
`remove`	`'a -> 'a set -> 'a set`	
Opérations complémentaires		
`empty_set`	`'a set`	
`is_empty`	`'a set -> bool`	
`iter`	`('a -> unit) -> 'a set -> unit`	« Boucle `for` »
`equal`	`'a set -> 'a set -> bool`	Égalité **ensembliste**

Signature minimale pour un type abstrait Set fonctionnel

Remarque

⇨ Cette signature est suffisante, mais il manque bien sûr de nombreuses fonctions très utiles que l'on ajouterait si l'on écrivait une « vraie » bibliothèque.

La spécification des différentes fonctions est intuitivement évidente. Cependant, il n'est jamais inutile de formaliser notre intuition. Pour ce faire, on associe à tout objet `s : 'a set` un ensemble, au sens mathématique du terme, $\varphi(\texttt{s})$ et l'on exige :
— $\varphi(\texttt{empty\_set}) = \varnothing$
— $\varphi(\texttt{add x s}) = \varphi(\texttt{s}) \cup \{\texttt{x}\}$
— $\varphi(\texttt{remove x s}) = \varphi(\texttt{s}) \setminus \{\texttt{x}\}$
— `member x s` si et seulement si $\texttt{x} \in \varphi(\texttt{s})$
— `equal s s'` si et seulement si $\varphi(\texttt{s}) = \varphi(\texttt{s'})$
— Si $\varphi(\texttt{s}) = \{x_1, \ldots, x_n\}$, alors `iter f s` a le même effet que `List.iter f [x_1; ... ; x_n]`. Attention, l'ordre des éléments est arbitraire.

Exercice 6

⇨ *Nombre d'éléments distincts* : On suppose que l'on dispose d'un type `'a set` doté des opérations définies ci-dessus. Écrire une fonction `cardinal_liste : 'a list -> int` qui renvoie le nombre d'éléments *distincts* de son argument.

4.2 Dictionnaire

Le type abstrait MAP (ou DICT) correspond à un dictionnaire, c'est-à-dire à une application partielle d'un ensemble A dans un ensemble B. Les éléments de A sont appelés *clés* et ceux de B *valeurs*. On parle généralement de *dictionnaire* ou de *tableau associatif*.

Ici, on présente une signature fonctionnelle :

Opération	Type	Commentaire
Opérations caractéristiques		
`get`	`'a -> ('a, 'b) map -> 'b option`	`None` si la clé est absente
`set`	`'a -> 'b -> ('a, 'b) map -> ('a, 'b) map`	Ajout ou remplacement
`remove`	`'a -> ('a, 'b) map -> ('a, 'b) map`	
Opérations complémentaires		
`empty_map`	`('a, 'b) map`	
`iter`	`('a * 'b -> unit) -> ('a, 'b) map -> unit`	

Signature minimale pour un type abstrait MAP fonctionnel

Formellement, on associe à tout `d : ('a, 'b) map` une relation fonctionnelle $\varphi(d) \subset A \times B$, c'est-à-dire un ensemble de couples vérifiant $((x,y) \in \varphi(d) \quad \text{et} \quad (x,y') \in \varphi(d)) \Longrightarrow y = y'$.
On demande alors :

— $\varphi($`empty_map`$) = \varnothing$
— `get x m = Some y` si $(x,y) \in \varphi($`m`$)$ (y est nécessairement unique)
— `get x m = None` s'il n'y a pas de couples de la forme (x,y) dans $\varphi($`m`$)$
— $\varphi($`remove x m`$) = \varphi($`m`$) \setminus \{(x,y) \mid y \in B\}$
— $\varphi($`set x y m`$) = \varphi($`remove x m`$) \cup \{(x,y)\}$

Exercices 7

⇨ *Listes d'associations* : La manière la plus simple de réaliser le type abstrait DICT est d'utiliser une liste de couples *(clé, valeur)* telles que les clés soient deux à deux distinctes. On suppose (uniquement dans cet exercice) que l'on choisit cette réalisation :

```
type ('a, 'b) dict = ('a * 'b) list
```

Écrire en OCaml des fonctions `get` et `set` respectant la spécification donnée ci-dessus, et donner leur complexité.

⇨ *Fonctions de conversion* : En général, on choisira des réalisations plus efficaces que les listes d'associations. Cependant, il est toujours utile de pouvoir convertir entre un objet de type DICT et une liste d'associations, et les fonctions présentes dans la signature suffisent pour réaliser cette conversion. Écrire en OCaml les deux fonctions suivantes (en supposant que l'on dispose de toutes les fonctions de la signature pour le type `('a, 'b) dict`) :

1. `of_list : ('a * 'b) list -> ('a, 'b) map` (on pourra supposer que la liste fournie ne contient pas deux associations différentes pour la même clé) ;
2. `to_list : ('a, 'b) map -> ('a * b') list`

Dans cet exercice, on ne sait *absolument pas* comment est réalisé le type `('a, 'b) dict` (en particulier, il n'est pas *a priori* égal à `('a * 'b) list`).

⇨ *Listes d'antécédents* : On considère un `'a array` de longueur n que l'on voit comme une application $f : [0 \dots n[\to A$.
Écrire une fonction `antecedents : 'a array -> ('a, int list) map` qui renvoie un dictionnaire m dont les clés sont les $y \in f([0 \dots n-1])$ et où la valeur associée à un y est la liste des $x \in [0 \dots n[$ tels que $f(x) = y$.

4.3 Table de hachage

Les tables de hachage fournissent une réalisation *fondamentalement impérative* des structures abstraites de dictionnaire et d'ensemble. Pour des raisons partiellement mathématiques et partiellement liées aux spécificités des machines actuelles, elles sont souvent plus rapides que les variantes d'arbres binaires de recherche, mais un peu moins flexibles.

4.3.1 Un cas simple

Supposons que l'on souhaite stocker un ensemble d'associations *(clé, valeur)*, et supposons de plus que l'on sache que :

— il y aura environ 5000 clés ;

— toutes les clés seront des entiers entre 0 et 9999.

Dans ce cas, la solution la plus efficace (et de loin !) est d'utiliser un tableau de longueur 10000, et de stocker l'association (k, v), si elle existe, dans la case d'indice k du tableau. En OCaml, on pourrait par exemple prendre le type suivant :

```
type 'v table = 'v option array
```

Les fonctions `get`, `set` et `remove` sont alors extrêmement simples, et s'exécutent, de manière immédiate, en temps constant :

```
let get table k = table.(k)
let set table k v = table.(k) <- Some v
let rem table k = table.(k) <- None
```

Exercice 8

⇨ 1. Pourquoi l'information « il y aura environ 5000 clés » est-elle importante ?

2. Supposons que l'on souhaite en fait réaliser une structure d'ensemble, avec les mêmes informations sur les clés. Quelle structure serait appropriée ?

4.3.2 Principe

Le principe d'une table de hachage est de conserver une solution proche de celle décrite ci-dessus tout en s'affranchissant des contraintes sur les clés, qui sont désormais librement choisies dans un ensemble K. On va donc fixer une valeur N pour la taille de la table (la longueur du tableau sous-jacent) et ramener les clés dans l'ensemble $[0 \dots N-1]$. Typiquement, il y aura deux étapes pour cela :

— on se dote d'une fonction $h : K \to \mathbb{N}$, dite *fonction de hachage* ;

— pour obtenir un indice dans le tableau à partir d'une clé k, on calcule $h_N(k) \overset{\text{déf.}}{=} h(k) \pmod N$.

Remarques

⇨ Quand on parle de fonction de K dans $\mathbb{N}$, il faut comprendre que l'on souhaite en réalité obtenir un entier de taille borné : typiquement, un `uint32_t`, un `uint64_t` ou un `int` OCaml positif.

⇨ L'entier $h(k)$ est appelé *empreinte* de la clé k (ou, en bon français informatique, *hash* de k).

⇨ Dans toute la suite, la notation $a \pmod b$ désigne le reste de la division euclidienne de a par b.

Exemple

⇨ Nous verrons plus loin les propriétés que l'on recherche pour une « bonne » fonction de hachage. Cependant, nous pouvons dès à présent donner quelques exemples de fonctions de hachage *possibles* (sans préjuger de leur qualité) :

—si $K = \mathbb{N}$ (ou $K = \mathbb{Z}$), on peut tout simplement prendre l'identité pour h, et l'on a alors $h_N(k) = k \pmod N$;

—si K est l'ensemble des chaînes de caractères, on peut voir chaque octet de la chaîne comme un entier (entre 0 et 255) et définir $h(k)$ comme la somme de ces entiers ;

—si K est l'ensemble des nombres flottants (disons sur 64 bits), on peut juste interpréter la représentation binaire de k comme un entier et se ramener au premier cas.

L'idée est alors de créer un tableau un tableau de taille N et de stocker l'association (k, v) dans la case numéro $h_N(k)$ de ce tableau. Si i est l'indice d'une case du tableau, on aura alors trois cas :

— aucune association ne vérifie $h_N(k) = i$: la case du tableau est « vide » ;

— une seule association vérifie $h_N(k) = i$: la case du tableau contient cette association ;

— on a une *collision* : plusieurs associations vérifient $h_N(k) = i$. On a alors un problème : les différents types de tables de hachage correspondent aux différentes manières de le régler.

L'idée fondamentale est que la recherche d'une association devient beaucoup plus rapide : si l'on cherche une clé k, il est inutile de parcourir tout le tableau. On calcule $h_N(k)$ et l'on regarde dans la case correspondante du tableau : si la clé n'y apparaît pas, c'est qu'elle n'apparaît nulle part dans le tableau.

Remarque

⇨ Il est important de bien comprendre que l'endroit où l'on stocke une association (k, v) ne dépend *que de la clé* k et pas du tout de la valeur v.

4.3.3 Résolution par chaînage

Une idée simple et couramment utilisée pour gérer les collisions est celle du *chaînage*. Au lieu d'utiliser un tableau de couples (k, v), on utilise un *tableau de listes de couples* (k, v). La case i du tableau contiendra la liste (éventuellement vide) des associations (k, v) vérifiant $h_N(k) = i$: en anglais, on appelle cette liste un *bucket* (*seau*, littéralement, mais on parle plutôt d'*alvéole* en français).

On pourrait donc imaginer le type suivant en OCaml :

```
type ('k, 'v) dict =
  {hash : 'k -> int;
   table : ('k * 'v) list array}
```

— Le champ `hash` contient la fonction de hachage h.
— Le champ `table` contient le tableau à proprement parler.

À l'intérieur d'une alvéole, les opérations de recherche, ajout, remplacement se font comme dans la réalisation naïve de l'exercice **??**. Globalement, on a donc deux étapes pour chaque opération :

— quand on veut ajouter un couple (k, v), on calcule $h_N(k)$ et l'on ajoute (k, v) à la liste se trouvant dans la case $h_N(k)$ du tableau (ou l'on remplace la valeur associée à k, s'il y en avait déjà une) ;
— quand on cherche la valeur associée à une clé k, on calcule $h_N(k)$ et on la cherche dans la liste contenue dans la case $h_N(k)$ du tableau (et uniquement dans cette liste).

Exercice 9

⇨ 1. Écrire trois fonctions `get_list`, `set_list` et `remove_list` agissant sur une liste d'associations. On pourra supposer que la liste contient au plus une association pour une clé donnée (et on maintiendra cet invariant).

```
get_list : ('k * 'v) list -> 'k -> 'v option
set_list : ('k * 'v) list -> 'k -> 'v -> ('k * 'v) list
remove_list : ('k * 'v) list -> 'k -> ('k * 'v) list
```

2. Écrire les fonctions `get`, `set` et `remove` agissant sur un dictionnaire.

```
get : ('k * 'v) dict -> 'k -> 'v option
set : ('k * 'v) dict -> 'k -> 'v -> unit
remove : ('k * 'v) dict -> 'k -> unit
```

En suivant cette méthode, la complexité de la recherche, de l'ajout ou de la suppression d'une clé k ne dépend plus du nombre total n d'associations du dictionnaire, mais seulement du nombre de clés en collision avec k. Plus précisément, on a :

— une évaluation de $h_N(k)$, dont le coût peut dépendre de k (si notre clé est par exemple une chaîne de caractères arbitrairement longue), mais pas de n ;
— un nombre de comparaisons entre clés égal à la longueur de la chaîne présente dans l'alvéole.

Idéalement, on souhaiterait répartir équitablement les valeurs de h_N, pour qu'il y ait (environ) le même nombre d'associations dans chaque alvéole.

4.3.4 Critères pour une bonne fonction de hachage

Une fonction de hachage « idéale » ne provoquerait jamais de collisions : c'est bien évidemment impossible, puisque l'ensemble K des clés possibles est en général beaucoup plus grand que $[0 \dots N-1]$ (voire infini). En étant un peu plus raisonnable, on souhaiterait que h_N se comporte comme un tirage aléatoire uniforme dans l'ensemble $[0 \dots N-1]$. Il n'est pas très difficile de construire des générateurs de nombre pseudo-aléatoires approchant raisonnablement bien cette propriété, mais il y a une difficulté supplémentaire : il est indispensable que deux appels $h_N(k)$ avec le même k renvoient la même valeur ! Si l'on tire au hasard une empreinte à chaque fois, ce ne sera bien sûr pas le cas.

Quand on analysera le coût des opérations sur les tables de hachage, on supposera toujours que la fonction de hachage répartit les clés uniformément et indépendamment dans l'ensemble $[0 \dots n-1]$.

Remarques

⇨ Autrement dit, on supposera :
— Pour tout $0 \leqslant i < N$, pour une clé k choisie aléatoirement, $\mathbb{P}(h_N(k) = i) = 1/N$.
— Si k et k' sont choisies aléatoirement, alors $\mathbb{P}(h_N(k) = h_N(k')) = 1/N$.

Ces propriétés ne sont pas réellement vérifiées par les fonctions de hachage utilisées en pratique (elles n'ont même pas forcément de sens), et de plus les clés n'ont aucune raison d'être choisies au hasard (ce qui invalidera la plus

grande partie de notre analyse). Cependant :

— il est possible de rendre cela rigoureux (en tirant la fonction de hachage au hasard dans une famille de fonctions bien choisies) ;

— les performances pratiques sont conformes à l'analyse simplifiée, pour peu que la fonction de hachage soit raisonnablement bien choisie.

⇨ On peut montrer que si l'on répartit aléatoirement n boules dans n urnes, le nombre de boules dans l'urne la plus « chargée » sera en moyenne $(\ln n)/(\ln(\ln n))$ (environ). Cela signifie que dans une table de hachage chaînée de taille n avec un facteur de charge de 1, le *pire cas* pour la recherche d'une clé sera en $\mathrm{O}\left((\ln n)/(\ln(\ln n))\right)$ (en supposant que la hachage se fasse uniformément).

Une fonction de hachage f est une fonction définie sur un ensemble E de cardinal très grand (voire infini) et à valeurs dans un ensemble F de taille fixée (grande, mais moins...). Un exemple typique : à une chaîne de caractère de taille quelconque, on associe un entier machine sur 32 bits. On appellera $f(x)$ l'*empreinte* de x (ou en bon français informatique le *hash* de x).

La propriété recherchée pour ce type de fonctions est la *pseudo-injectivité* : on souhaite que tout se passe comme si la valeur de $f(x)$ avait été tirée au hasard, uniformément, dans l'ensemble F (sauf bien sûr que l'on veut obtenir le même résultat si l'on calcule plusieurs fois $f(x)$ pour le même x). Ainsi, on pourra faire comme si $\mathbb{P}(f(x) = f(y))$ valait $\frac{1}{\operatorname{Card} F}$ dès que $x \neq y$.

Aucune fonction n'a réellement cette propriété (en fait, cette propriété n'a pas vraiment de sens mathématiquement). Cependant, il est possible de formaliser plus précisément ce qui nous intéresse réellement dans la pseudo-injectivité, et de trouver des fonctions convenables. C'est cependant difficile : on utilisera donc `Hashtbl.hash :'a -> int`, qui à un objet quelconque associe un entier positif ou nul de manière « pseudo-uniforme ».