Chapitre 10

Dictionnaires

Ce chapitre a pour objet l'étude d'une structure de données qui répond à un problème très fréquemment rencontré en informatique : la recherche d'information dans un ensemble géré de façon dynamique (c'est à dire dont le contenu est susceptible d'évoluer au cours du temps). Un dictionnaire (bien que statique sous format papier) en constitue une assez bonne image : il permet d'accéder à l'information recherchée (la définition d'un mot) à partir d'une clé (ici le mot lui-même), ces clés étant classées suivant leur première lettre pour permettre une recherche plus rapide.

1. Tables d'association

1.1 Description

Un *dictionnaire* (ou mieux une *table d'association*) est un type de données associant un ensemble de clés à un ensemble de valeurs. Plus formellement, si C désigne l'ensemble des clés et V l'ensemble des valeurs, une table d'association est un sous-ensemble T de $C \times V$ tel que pour toute clé $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$.

Une table d'association supporte en général les opérations suivantes :

- *ajout* d'une nouvelle paire $(c, v) \in C \times V$ dans T ;
- *suppression* d'une paire (c, v) de T ;
- *lecture* de la valeur associée à une clé dans T.

La gestion des emprunts d'une bibliothèque constitue une bonne illustration de ce qu'est une table d'association : ici les clés sont les livres de la bibliothèque et les valeurs les abonnés de celle-ci. La table d'association est alors l'ensemble des couples (livre, emprunteur) des livres sortis de la bibliothèque. Chaque livre ne peut être emprunté que par une personne à la fois, mais chaque personne peut emprunter plusieurs livres.
À chaque fois qu'une personne emprunte un nouveau livre, une opération d'ajout est effectuée ; à chaque fois qu'une personne rend un livre, une opération de suppression est effectuée. Enfin, le bibliothécaire a la possibilité de consulter le nom de l'emprunteur d'un livre donné.

Les tables d'associations sont couramment utilisées en informatique : c'est par exemple la structure de données utilisée pour gérer les systèmes de noms de domaine (DNS, pour *Domain Name System*). Les ordinateurs connectés à un réseau comme Internet possèdent une adresse numérique (en IPv4 par exemple, celles-ci sont représentées sous la forme xxx.xxx.xxx.xxx, où xxx est un nombre hexadécimal variant entre 0 et 255). Pour faciliter l'accès aux systèmes qui disposent de ces adresses, un mécanisme a été mis en place pour associer un nom (plus facile à retenir) à une adresse IP. Ce mécanisme utilise une table d'association dans laquelle les clés sont les noms de domaine et les valeurs les adresses IP.

1.2 Le module `hashtbl`

Le module **`hashtbl`** de la bibliothèque standard permet d'utiliser des tables d'association en CAML. Rappelons qu'avant d'utiliser les fonctions de ce module, il faudra utiliser la commande : **`#open "hashtbl"`** ou préfixer chacune des fonctions de ce module par le préfixe **`hashtbl__`**.

• Les primitives du module `hashtbl`

Les tables d'association dont les clés sont de type *'a* et les valeurs de type *'b* ont pour type *('a, 'b) t*.
La fonction **`new`** de type *int -> ('a, 'b) t* crée une nouvelle table d'association vide.

Le paramètre entier donne la taille initiale de la table (au moins égale à 1) ; puisque cette dernière est susceptible d'augmenter, cette valeur ne constitue qu'une évaluation initiale. Le manuel conseille de choisir un nombre premier pour cette valeur [1].

1. Ces spécifications sont liées au choix de l'implémentation des dictionnaires par des tables de hachage ; la raison en sera expliqué plus loin dans le cours.

La fonction **clear** de type *('a, 'b) t -> unit* vide une table d'association.
La fonction **add** de type *('a, 'b) t -> 'a -> 'b -> unit* ajoute une nouvelle association à la table.
Si la clé est déjà présente, la nouvelle association remplace la précédente.
La fonction **remove** de type *('a, 'b) t -> 'a -> unit* supprime l'association liée à une clé donnée.
La fonction **find** de type *('a, 'b) t -> 'a -> 'b* renvoie la valeur associée à une clé donnée.
Si la clé n'est pas présente, la fonction déclenche l'exception **Not_found**.

Pour illustrer l'utilisation de ce module, nous allons revenir sur le problème qui nous a intéressé dans le chapitre précédent : comment rendre efficace une fonction récursive dont les appels sont interdépendants ?
Considérons de nouveau le calcul des coefficients binomiaux par la formule de Pascal :

```
let rec binom = fun
  | n 0 -> 1
  | n p when n = p -> 1
  | n p -> binom (n-1) (p-1) + binom (n-1) p ;;
```

Nous savons que cette définition est très peu efficace car les mêmes calculs sont effectués de nombreuses fois.
La *mémoïsation* consiste à lier à cette fonction une table d'association initialement vide, qui sera remplie au fur et à mesure par des paires (c, v) où la clé c est un couple d'entiers (n, p) et la valeur associée v le coefficient $\binom{n}{p}$.
Ainsi, la fonction commencera par regarder si la valeur du coefficient binomial n'est pas déjà présente dans la table avant d'effectuer les appels récursifs si ce n'est pas le cas.
Par exemple :

```
let binom =
  let tbl = new 997 in
  let rec b n p =
    try find tbl (n, p) with
      | Not_found -> let v = (if p = 0 or n = p then 1
                              else b (n-1) (p-1) + b (n-1) p) in
                     add tbl (n, p) v ;
                     v
  in b ;;
```

Cette fonction a maintenant des performances temporelles tout à fait semblables à la version utilisant la programmation dynamique, tout en restant très proche de la définition mathématique initiale.

1.3 Mise en œuvre pratique d'une table d'association

Évaluer la complexité temporelle de la fonction **binom** nous est pour l'instant impossible car nous ignorons le coût de la lecture et de l'ajout dans une table d'association. Pour le connaître il va être nécessaire de s'interroger sur l'implémentation de ces structures de données, en gardant à l'esprit que pour que ces dictionnaires soient utiles il est nécessaire que ces coûts soient les plus faibles possibles.

• Insuffisance des listes et des vecteurs

Une première idée serait d'utiliser une liste pour stocker les couples (c, v) de T. Ceci conduirait à la définition de type suivante :

```
type ('a, 'b) data = {Key : 'a ; Value : 'b} ;;
type ('a, 'b) t = {mutable Tbl : ('a, 'b) data list} ;;
```

Puis aux définitions des fonctions :

```
let new () = {Tbl = []} ;;
```

```
let find t c =
  let rec aux = function
    | []                  -> raise Not_found
    | d::_ when d.Key = c -> d.Value
    | _::q                -> aux q
  in aux t.Tbl ;;
```

```
let add t c v =
  let rec aux = function
    | []                  -> [{Key = c ; Value = v}]
    | d::q when d.Key = c -> {Key = c ; Value = v}::q
    | d::q                -> d::(aux q)
  in t.Tbl <- aux t.Tbl ;;
```

```
let remove t c =
  let rec aux = function
    | [] -> []
    | d::q when d.Key = c -> q
    | d::q -> d::(aux q)
  in t.Tbl <- aux t.Tbl ;;
```

Cette implémentation n'est donnée qu'à titre d'exemple car pas assez efficace : les coûts des fonctions d'ajout et de lecture sont dans le pire des cas comme en moyenne linéaires vis à vis de la taille du dictionnaire.

pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

On peut améliorer l'efficacité de l'ajout en insérant systématiquement en tête de liste le nouveau couple (c,v) :

```
let add t c v = t.Tbl <- {Key = c ; Value = v}::t.Tbl ;;
```

Le coût de l'ajout devient un $\Theta(1)$ mais en contrepartie, plusieurs clés identiques associées à des valeurs différentes peuvent se trouver dans la table et la fonction de suppression se contente de restaurer l'association précédente, si celle-ci existe [2].

Lorsque l'ensemble des clés C est totalement ordonné, il est possible de stocker les paires (c,v) dans un vecteur maintenu trié. L'inconvénient de cette démarche est qu'il faut prévoir au départ la taille maximale du dictionnaire ; l'avantage est que la lecture peut se faire par recherche dichotomique avec un coût logarithmique. En revanche, ajout et suppression ont toujours un coût linéaire (il faut décaler une partie des éléments du tableau vers la gauche ou vers la droite pour maintenir le tableau trié).

pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$

On l'aura compris, ce ne sont pas ces structures de données qui sont utilisées pour représenter les tables d'association. Celles-ci sont principalement représentées par les *arbres de recherche équilibrés* (étudiés en seconde année), dont les coûts en moyenne comme dans le pire des cas sont logarithmiques :

pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

et surtout par les *tables de hachage* encore plus efficaces, qui seront l'objet de la suite de notre étude :

pire des cas		**en moyenne**	
lecture	ajout	lecture	ajout
$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

1.4 Tables de hachage

Si $n = |\mathrm{C}|$, l'idéal serait de représenter le table T à l'aide d'un tableau de taille n et d'une fonction bijective $h : \mathrm{C} \to [\![0, n-1]\!]$; dans ce cas la lecture et l'ajout se feraient en coût constant. Ce n'est bien évidemment pas réaliste, l'ensemble C étant le plus souvent de taille bien trop importante, et en tout cas bien plus grande que le nombre de clés utilisées.
C'est pourquoi on se contente d'un tableau de taille $m \ll n$ et d'une fonction $h : \mathrm{C} \to [\![0, m-1]\!]$ qui porte de nom de *fonction de hachage* et qui doit respecter autant que faire ce peut les spécifications suivantes :

2. On peut noter que c'est de cette façon qu'agit le module **hashtbl**.

– être facile à calculer ;
– avoir une distribution d'apparence uniforme.

Cette dernière condition est motivée par le souhait de minimiser le nombre de *collisions*, lorsque deux clés distinctes c_1 et c_2 vérifient $h(c_1) = h(c_2)$ (notons que celles-ci deviennent inévitables dès lors que $m < n$). Il est donc utile que cette fonction assure une bonne répartition des clés dans les différents emplacements du tableau c'est à dire, de manière informelle, qu'étant donné un entier $i \in [\![0, m-1]\!]$, la probabilité que $h(c)$ soit égale à i soit proche de $1/m$. Dans ces conditions, si k désigne le nombre de clés distinctes de T, la probabilité pour qu'il y ait au moins une collision est égale à :

$$1 - \frac{m!}{m^k (m-k)!}$$

Pour $m = 1000000$, la probabilité pour qu'il y ait collision dépasse 50% lorsque le nombre de clés dépasse 1200 ; pour $k = 2500$ la probabilité qu'il y ait collision dépasse 95% (c'est le fameux paradoxe des anniversaires).

Nous avons donc à résoudre deux problèmes :

– quelle fonction de hachage choisir ?
– que faire en cas de collision ?

• Choix de la fonction de hachage

Si toutes les clés sont connues à l'avance, il existe des algorithmes pour construire une fonction de hachage parfaite, c'est à dire sans collision, mais la plus-part du temps on se contente d'utiliser une heuristique basée sur la nature attendue des clés.

Les fonctions de hachages supposent que les clés sont des entiers naturels ; si ce n'est pas le cas, il faut préalablement utiliser une fonction de codage. Par exemple, si les clés sont des chaînes de caractères, à chaque caractère s est associé son code ASCII $\alpha(s)$ (un entier compris entre 0 et 127) et à la chaîne de caractères $s_0 s_1 \cdots s_{n-1}$ est associée l'entier :

$$\sum_{k=0}^{n-1} \alpha(s_k) \times 127^k.$$

Il faut ensuite choisir la fonction de hachage $h : \mathbb{N} \to [\![0, m-1]\!]$. Deux types de fonctions sont principalement utilisées : par *division* ou par *multiplication*.

Méthode de division

On choisit la fonction $\boxed{h : c \mapsto c \bmod m}$
C'est une fonction simple et rapide, mais qui demande de choisir judicieusement m car certaines valeurs conduisent à de très mauvais hachages :

– $m = 2^p$ est un mauvais choix car $h(c)$ ne dépend que des p bits de poids les plus faibles de c. Dans le cas des chaînes de caractères par exemple, si $m = 128$ seul le premier caractère de la chaîne sert au hachage, ce qui conduit en général à une répartition non uniforme.
– $m = 2^p - 1$ est aussi considéré comme un mauvais choix car une permutation des paquets de p bits consécutifs ne modifie pas le hachage (dans le codage des chaînes de caractères par exemple, si $m = 127$ la permutation des lettres d'une chaîne ne modifie pas la valeur du hachage).
– Enfin, si les clés forment des séquences périodiques, il vaut mieux choisir m premier. En effet, si b n'est pas premier avec m, alors $\left\{(a + bi) \bmod m \;\middle|\; i \in [\![0, m-1]\!]\right\} \subsetneq [\![0, m-1]\!]$ alors que dans le cas contraire il y a égalité.

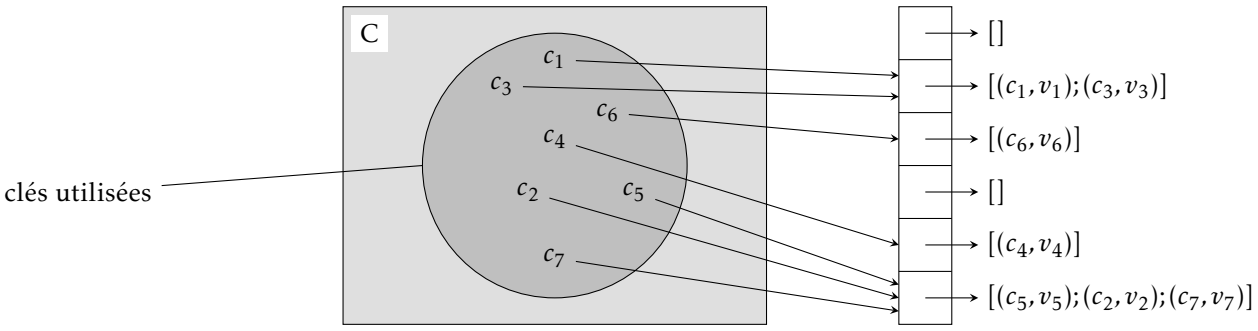
Toutes ces considérations conduisent en général à préconiser de choisir pour m un nombre premier pas trop proche d'une puissance de 2.

Méthode de multiplication

On choisit la fonction $\boxed{h : c \mapsto \lfloor m \times (c\alpha \bmod 1) \rfloor}$ où α est un nombre réel dans $]0, 1[$.
Le calcul est plus lent que pour la méthode de division mais conduit à une bonne répartition des clés lorsque α est bien choisi (D. Knuth préconise de prendre $\alpha = \dfrac{\sqrt{5}-1}{2}$). De plus, le choix de m n'est plus critique.

• Résolution des collisions par chaînage

On l'a dit, malgré tout le soin pris à choisir une bonne fonction de répartition, les collisions restent inévitables. La façon la plus simple de résoudre ce problème consiste à utiliser un tableau dont les cases sont des listes qui serviront à stocker les couples (c, v) de T associés à la même valeur de hachage.



Ceci conduit à définir le type suivant :

```
type ('a, 'b) t = {H : 'a -> int ; Tbl : ('a, 'b) data list vect} ;;
```

Choisissons un hachage par division sur des clés entières en définissant :

```
let new m = {H = (function x -> x mod m) ; Tbl = make_vect m []};;
```

On définit alors les fonctions attachées à la table d'association de la façon suivante (indépendamment du choix de la fonction de hachage) :

```
let find t c =
  let rec aux = function
    | []                  -> raise Not_found
    | d::_ when d.Key = c -> d.Value
    | _::q                -> aux q
  in aux t.Tbl.(t.H c) ;;
```

```
let add t c v =
  let i = t.H c in
  t.Tbl.(i) <- {Key = c ;  Value = v}::t.Tbl.(i) ;;
```

(à l'instar du module **hashtbl** la fonction d'ajout ne supprime pas une éventuelle association précédente mais se contente de la recouvrir).

```
let remove t c =
  let i = t.H c in
  let rec aux = function
    | []                  -> []
    | d::q when d.Key = c -> q
    | d::q                -> d::(aux q)
  in t.Tbl.(i) <- aux t.Tbl.(i) ;;
```

En guise d'illustration, calculons de nouveau les coefficients binomiaux par mémoïsation. Puisque les clés sont ici des couples (n, p), nous avons besoin d'une fonction de codage de $\mathbb{N}^2$ dans $\mathbb{N}$. On utilise le polynôme de Cantor qui réalise une bijection entre ces deux ensembles :

```
let code n p = ((n + p) * (n + p) + 3 * n + p) / 2 ;;
```

puis :

```
let binom =
  let tbl = new 997 in
  let rec b n p =
    try find tbl (code n p) with
      | Not_found -> let v = (if p = 0 or n = p then 1
                              else b (n-1) (p-1) + b (n-1) p) in
                     add tbl (code n p) v ;
                     v
  in b ;;
```

- **Adressage ouvert**

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, 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 $i+1 \bmod m$, $i+2 \bmod m$, $i+3 \bmod m$, ... 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 $i + h'(c) \bmod m$, $i + 2h'(c) \bmod m$, $i + 3h'(c) \bmod m$ ..., sans pour autant résoudre complètement le problème des agrégats.

Évidemment, un 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 $\alpha = \dfrac{k}{m}$ 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.
Ces questions sont abordées dans les exercices 3 et 4.

2. Exercices

Exercice 1 On considère k clés distinctes réparties uniformément dans un tableau de taille m. Montrer que si $k \geqslant \dfrac{1+\sqrt{1+8m\ln 2}}{2}$, la probabilité qu'il y ait au moins une collision est supérieure ou égale à 50%.

Exercice 2 On dispose d'une table de hachage de taille m dont les collisions sont résolues par listes chaînées, et on suppose la fonction de hachage h uniforme.

a) Exécuter manuellement l'algorithme d'insertion dans la table pour $m = 9$ et $h : c \mapsto c \bmod m$ des clés suivantes : 5, 28, 19, 15, 20, 33, 12, 17, 10.

b) Si k clés sont présentes dans la table, quelle est la durée moyenne (en nombre de clés testées) de recherche d'une clé non présente dans la table ?

c) On admet que si $r(c)$ est le rang d'insertion de la clé c présente dans la table, les k valeurs possibles de $r(c)$ sont équiprobables. Déterminer la durée moyenne de recherche d'une clé présente dans la table.

d) Que peut-on conclure des questions précédentes lorsque $k = O(m)$?

Exercice 3 Dans cet exercice, on s'intéresse à la méthode de résolution des collisions par adressage ouvert à l'aide d'un sondage linéaire.

a) 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 $h(c) = c \bmod 9$.

b) On définit les types et la fonction :

```
type ('a, 'b) data = {Key : 'a ; Value : 'b} ;;
type ('a, 'b) item = Nil | Data of ('a, 'b) data ;;
type ('a, 'b) t = {H : 'a -> int ; Tbl : ('a, 'b) item vect} ;;

let new m = {H = (function x -> x mod m) ; Tbl = make_vect m Nil};;
```

Rédiger les fonctions de lecture et d'ajout associées dans une table de hachage à adressage ouvert avec sondage linéaire (on supposera la taille de la table grande devant le nombre de clés utilisées).

c) Que ce passe-t-il si on supprime certaines clés de la table ? Comment peut-on résoudre ce problème ?

d) 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 ?

Exercice 4 Que ce soit par chaînage ou par adressage ouvert, le coût du sondage augmente dès lors que la proportion du nombre de clés dans le tableau augmente. D'où l'idée d'utiliser une table de hachage de taille dynamique, le principe étant d'augmenter la taille de la table dès lors que la proportion de clé atteint un certain seuil.
Ceci nous amène à définir :

```
type ('a, 'b) t = {mutable Size : int ; mutable H : 'a -> int ;
                   mutable Tbl :  ('a, 'b) data list vect} ;;
```

le champ **`Size`** décrivant le nombre de clés dans la table de hachage.

a) Définir les fonctions de création **`new`** et de recherche **`find`**. On utilisera un hachage par division et une résolution des collisions par chaînage.

b) On utilise le principe de redimensionnement suivant : lors de l'ajout d'une entrée, si le nombre de clés dépasse le double de la taille du tableau on double la taille de ce dernier avant de procéder à l'ajout.
Définir la fonction **`reorder`** qui réalise ce redimensionnement, et en déduire la fonction d'ajout **`add`**.

c) En revanche, la suppression d'une clé ne conduit pas à un redimensionnement de la table, puisqu'une table de hachage peu remplie ne constitue pas un inconvénient.
Définir pour terminer la fonction **`remove`**.