Dictionnaires

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Dictionnaires

Une table d'association est un type de données associant un ensemble de clés à un ensemble de valeurs.
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 $v \in V$ tel que $(c, v) \in T$.

Dictionnaires

Une table d'association est un type de données associant un ensemble de clés à un ensemble de valeurs.
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 $v \in V$ 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.

Dictionnaires

Une table d'association est un type de données associant un ensemble de clés à un ensemble de valeurs.
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 $v \in V$ 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.

Exemples :

- répertoire téléphonique électronique : *nom* $\longrightarrow$ *téléphone* ;

Dictionnaires

Une table d'association est un type de données associant un ensemble de clés à un ensemble de valeurs.
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 $v \in V$ 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.

Exemples :

- répertoire téléphonique électronique : *nom* $\longrightarrow$ *téléphone* ;
- gestion des noms de domaines : *nom* $\longrightarrow$ *adresse IP* ;

Dictionnaires

Une table d'association est un type de données associant un ensemble de clés à un ensemble de valeurs.
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 $v \in V$ 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.

Exemples :

- répertoire téléphonique électronique : *nom* $\longrightarrow$ *téléphone* ;
- gestion des noms de domaines : *nom* $\longrightarrow$ *adresse IP* ;
- table de référencement des variables d'un langage de programmation : *variable* $\longrightarrow$ *adresse*.

Le 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*.

Le 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.
- La fonction **clear** de type *('a, 'b) t -> unit* vide une table d'association.

Le 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.
- 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.
- La fonction **remove** de type *('a, 'b) t -> 'a -> unit* supprime l'association liée à une clé donnée.

Le 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.
- 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.
- 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 dans la table, la fonction **find** déclenche l'exception **Not_found**.

Le module **hashtbl**

Application à la mémoïsation d'une fonction

On sait que le calcul récursif des coefficients binomiaux par la formule de Pascal est très couteux en temps :

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

Le module **hashtbl**

Application à la mémoïsation d'une fonction

On sait que le calcul récursif des coefficients binomiaux par la formule de PASCAL est très couteux en temps :

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

La mémoïsation consiste à lier à cette fonction une table d'association 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 $\binom{n}{p}$.

Le module hashtbl

Application à la mémoïsation d'une fonction

On sait que le calcul récursif des coefficients binomiaux par la formule de Pascal est très couteux en temps :

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

La mémoïsation consiste à lier à cette fonction une table d'association 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 $\binom{n}{p}$.

```
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 ;;
```

Mise en œuvre pratique

avec une liste

On définit les types :

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

Puis les fonctions :

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

Mise en œuvre pratique

avec une liste

On définit les types :

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

Puis les fonctions :

```
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 ;;
```

Mise en œuvre pratique

avec une liste

On définit les types :

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

Puis les fonctions :

```
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 ;;
```

Mise en œuvre pratique

avec une liste

On définit les types :

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

Puis les fonctions :

```
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 ;;
```

Mise en œuvre pratique

avec une liste

On définit les types :

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

Puis les fonctions :

```
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 ;;
```

L'ajout peut se faire à coût constant :

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

Dans ce cas, la fonction **remove** restaure l'association précédente, si celle-ci existe.

Mise en œuvre pratique

Comparaison des implémentations usuelles

Avec une *liste* :

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

Avec un *vecteur ordonné* :

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

Avec un *arbre de recherche équilibré* :

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

Avec une table de hachage :

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

Tables de hachage

Si $n = |C|$ est le nombre total de clé possibles, l'idéal est de représenter le table T à l'aide d'un tableau de taille n et d'une fonction bijective $h : C \to [\![0, n-1]\!]$; dans ce cas la lecture et l'ajout se font en coût constant.

Tables de hachage

Si $n = |C|$ est le nombre total de clé possibles, l'idéal est de représenter le table T à l'aide d'un tableau de taille n et d'une fonction bijective $h : C \to [\![0, n-1]\!]$; dans ce cas la lecture et l'ajout se font en coût constant.
$\Longrightarrow$ irréaliste car $n \gg 1$.

On utilise un tableau de taille $m \ll n$ et une fonction $h : C \to [\![0, m-1]\!]$ (la fonction de hachage) qui respecte les spécifications suivantes :

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

Tables de hachage

Si $n = |C|$ est le nombre total de clé possibles, l'idéal est de représenter le table T à l'aide d'un tableau de taille n et d'une fonction bijective $h : C \to [\![0, n-1]\!]$; dans ce cas la lecture et l'ajout se font en coût constant.
$\Longrightarrow$ irréaliste car $n \gg 1$.

On utilise un tableau de taille $m \ll n$ et une fonction $h : C \to [\![0, m-1]\!]$ (la fonction de hachage) qui respecte les spécifications suivantes :

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

Il y a collision lorsque deux clés distinctes c_1 et c_2 vérifient : $h(c_1) = h(c_2)$ (la même case du tableau est attribuée à deux clés distinctes).
Si $m < n$, h ne peut être injective : les collisions sont inévitables.

Tables de hachage

Si $n = |C|$ est le nombre total de clé possibles, l'idéal est de représenter le table T à l'aide d'un tableau de taille n et d'une fonction bijective $h : C \to [\![0, n-1]\!]$; dans ce cas la lecture et l'ajout se font en coût constant.
$\Longrightarrow$ irréaliste car $n \gg 1$.

On utilise un tableau de taille $m \ll n$ et une fonction $h : C \to [\![0, m-1]\!]$ (la fonction de hachage) qui respecte les spécifications suivantes :

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

Il y a collision lorsque deux clés distinctes c_1 et c_2 vérifient : $h(c_1) = h(c_2)$ (la même case du tableau est attribuée à deux clés distinctes).
Si $m < n$, h ne peut être injective : les collisions sont inévitables.

Si la répartition de k clés est uniforme, la probabilité qu'il n'y ait pas de collision est :

$$\frac{m \times (m-1) \times (m-2) \times \cdots \times (m-k+1)}{m^k} = \frac{m!}{m^k (m-k)!}.$$

Tables de hachage

Si $n = |C|$ est le nombre total de clé possibles, l'idéal est de représenter le table T à l'aide d'un tableau de taille n et d'une fonction bijective $h : C \to [\![0, n-1]\!]$; dans ce cas la lecture et l'ajout se font en coût constant.
$\Longrightarrow$ irréaliste car $n \gg 1$.

On utilise un tableau de taille $m \ll n$ et une fonction $h : C \to [\![0, m-1]\!]$ (la fonction de hachage) qui respecte les spécifications suivantes :

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

Il y a collision lorsque deux clés distinctes c_1 et c_2 vérifient : $h(c_1) = h(c_2)$ (la même case du tableau est attribuée à deux clés distinctes).
Si $m < n$, h ne peut être injective : les collisions sont inévitables.

Pour $m = 1000000$, la probabilité de collision :

- dépasse 50% lorsque le nombre de clés dépasse 1200 ;
- dépasse 95% lorsque le nombre de clés dépasse 2500.

Tables de hachage

Si $n = |C|$ est le nombre total de clé possibles, l'idéal est de représenter le table T à l'aide d'un tableau de taille n et d'une fonction bijective $h : C \to [\![0, n-1]\!]$; dans ce cas la lecture et l'ajout se font en coût constant.
$\Longrightarrow$ irréaliste car $n \gg 1$.

On utilise un tableau de taille $m \ll n$ et une fonction $h : C \to [\![0, m-1]\!]$ (la fonction de hachage) qui respecte les spécifications suivantes :

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

Il y a collision lorsque deux clés distinctes c_1 et c_2 vérifient : $h(c_1) = h(c_2)$ (la même case du tableau est attribuée à deux clés distinctes).
Si $m < n$, h ne peut être injective : les collisions sont inévitables.

Deux problèmes à résoudre :

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

Choix de la fonction de hachage

Les fonctions de hachages supposent que les clés sont des entiers naturels, quitte à utiliser une fonction de codage.

Exemple. Si les clés sont des chaînes de caractères, on note $\alpha(s)$ le code ASCII de s et on associe à la chaîne de caractères $s_0 s_1 \cdots s_{n-1}$ l'entier :

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

Choix de la fonction de hachage

Les fonctions de hachages supposent que les clés sont des entiers naturels, quitte à utiliser une fonction de codage.

Méthode de division : on choisit la fonction $h : c \mapsto c \bmod m$.

- $m = 2^p$ est un mauvais choix car $h(c)$ ne dépend que des p bits de poids les plus faibles de c, ce qui peut conduire 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.
- Il vaut mieux choisir m premier. En effet, $\left\{(a+bi) \bmod m \;\middle|\; i \in [\![0, m-1]\!]\right\} = [\![0, m-1]\!]$ ssi b est premier avec m.

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

Choix de la fonction de hachage

Les fonctions de hachages supposent que les clés sont des entiers naturels, quitte à utiliser une fonction de codage.

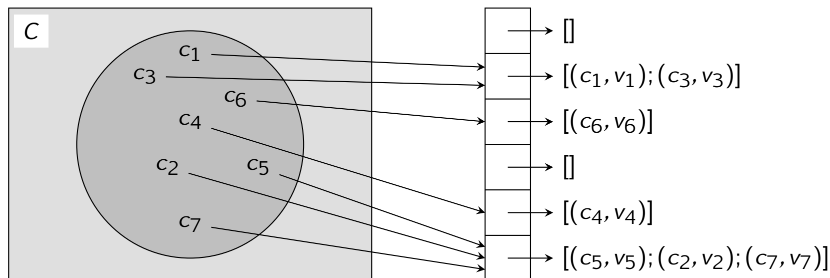
Méthode de multiplication : on choisit la fonction $h : c \mapsto \lfloor m(c\alpha \bmod 1) \rfloor$ où $\alpha \in]0,1[$.
Le calcul est plus lent que pour la méthode de division mais conduit à une bonne répartition des clés lorsque α est bien choisi.

Donald Knuth préconise de prendre $\alpha = \dfrac{\sqrt{5}-1}{2}$.
Pour cette méthode, le choix de m n'est plus critique.

Résolution des collisions

par chaînage

La façon la plus simple de résoudre ce problème consiste à utiliser un tableau dont les cases sont des listes qui servent à stocker les couples (c, v) de T associés à la même valeur de hachage.



Résolution des collisions

par chaînage

La façon la plus simple de résoudre ce problème consiste à utiliser un tableau dont les cases sont des listes qui servent à stocker les couples (c, v) de T associés à la même valeur de hachage.

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

On choisit un hachage par division :

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

Résolution des collisions

par chaînage

La façon la plus simple de résoudre ce problème consiste à utiliser un tableau dont les cases sont des listes qui servent à stocker les couples (c, v) de T associés à la même valeur de hachage.

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

On choisit un hachage par division :

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

La fonction de recherche :

```
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) ;;
```

Résolution des collisions

par chaînage

La façon la plus simple de résoudre ce problème consiste à utiliser un tableau dont les cases sont des listes qui servent à stocker les couples (c, v) de T associés à la même valeur de hachage.

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

On choisit un hachage par division :

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

La fonction d'ajout (par recouvrement) :

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

Résolution des collisions

par chaînage

La façon la plus simple de résoudre ce problème consiste à utiliser un tableau dont les cases sont des listes qui servent à stocker les couples (c, v) de T associés à la même valeur de hachage.

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

On choisit un hachage par division :

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

La fonction de suppression :

```
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) ;;
```

Résolution des collisions

par sondage

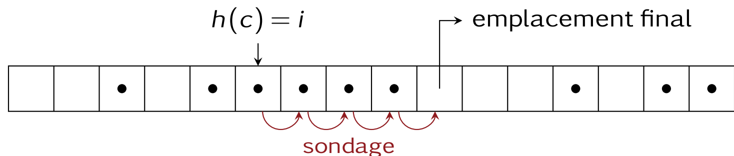
En cas de collision, une autre méthode consiste à sonder la table à la recherche d'une case vide.

Résolution des collisions

par sondage

En cas de collision, une autre méthode consiste à sonder la table à la recherche d'une case vide.

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 de sondage linéaire) mais ceci présente l'inconvénient de former des « agrégats » qui nuisent à la répartition uniforme recherchée.

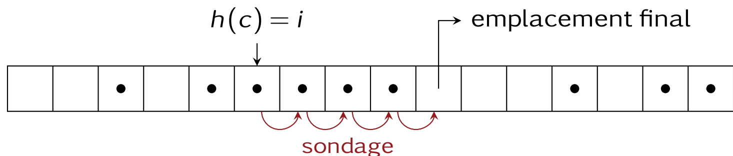


Résolution des collisions

par sondage

En cas de collision, une autre méthode consiste à sonder la table à la recherche d'une case vide.

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 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 $i+h'(c) \bmod m$, $i+2h'(c) \bmod m$, $i+3h'(c) \bmod m$ ...

Résolution des collisions

par sondage

En cas de collision, une autre méthode consiste à sonder la table à la recherche d'une case vide.

Un adressage ouvert exige que le nombre k de clés soit inférieur à la taille de la table m.

Lorsque le rapport $\alpha = \dfrac{k}{m}$ se rapproche de 1, il est de plus en plus difficile de trouver un emplacement vide :

- si $\alpha = 0,5$ un ajout nécessite en moyenne deux sondages ;
- si $\alpha = 0,9$ un ajout nécessite en moyenne dix sondages.

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.