Tableaux associatifs (ou *dictionnaires*)

Pierre Karpman

Lycée Champollion MP2I
https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Définition

Un tableau associatif est une représentation d'une application partielle $\mathcal{K} \to \mathcal{V}$ d'un ensemble (fini) de clefs vers un ensemble de valeurs

Spécifications (version impérative, syntaxe OCaml)

Un type `('a, 'b) map` doté (au moins) d'opérations :

- `make : unit -> ('a, 'b) map` : création d'une structure vide
- `size : ('a, 'b) map -> int` : nombre d'éléments présents
- `add : ('a, 'b) map -> 'a -> 'b -> unit` : `add m k v` modifie `m` pour y ajouter (ou remplacer) une *association* entre la *clef* `k` et la *valeur* `v`
- `del : ('a, 'b) map -> 'a -> unit` : `del m k` modifie `m` pour en retirer (l'éventuelle) association de clef `k`
- `find_opt : ('a, 'b) map -> 'a -> 'b option` : `find_opt m k` s'évalue en une option sur l'éventuelle valeur associée à `k` dans `m`
  - Alternative : version avec exception

Spécifications *bis*

Exemples de signatures pour version fonctionnelle

- `empty : ('a, 'b) map` : structure vide
- `add : ('a, 'b) map -> 'a -> 'b -> ('a, 'b) map`
- `del : ('a, 'b) map -> 'a -> ('a, 'b) map`

Exercice

Proposez une implémentation de `add`, `del` et `find_opt` pour un tableau associatif fonctionnel par *liste d'associations*, de `type ('a, 'b) map = ('a * 'b) list`, où :

- une association entre la clef `k` et la valeur `v` est donnée par la *première* (en partant de la tête) occurrence d'un couple `k, v`
- `del` ne fait que « supprimer » l'association « la plus récente »

Correction

Par exemple :

```
type ('a, 'b) map = ('a * 'b) list
let empty = []
let add m k v = (k,v)::m
let rec del m k
    = match m with
    | [] -> []
    | (k', v')::ms -> if k' = k then
                      ms else
                      (k', v')::(del ms k)
let rec find_opt m k
    = match m with
    | [] -> None
    | (k', v')::ms -> if k' = k then
                      Some v' else
                      find_opt ms k
```

Tableau associatif par liste d'associations : commentaires

- Implémentation très simple
- Version impérative possible, et toute aussi simple
- Mauvaises performances : `del` et `find_opt` ont un coût pire-cas au moins linéaire en le nombre d'associations

Enjeu : efficacité

Applications : quelques exemples

Les tableaux associatifs sont une structure de données puissante et expressive

Recherche efficace de collision

(ou plus généralement : calcul d'histogramme)

- Notamment utile dans des algorithmes de « rencontre au milieu »

Structure d'ensemble

Les opérations élémentaires sur des ensembles se réduisent à celles sur des tableaux associatifs (`'a set` $\approx$ `('a, unit) map`)

Généralisation des tableaux à des indices non entiers

Par ex. quand les indices sont des chaînes de caractères

Table des matières

Implémentation (fonctionnelle) efficace : ABRs équilibrés

Un tableau associatif peut s'implémenter aisément à partir d'une structure de données d'arbre (binaire) de recherche **à condition que l'espace des clefs $\mathcal{K}$ soit totalement ordonnable**

- Soit $<$ un tel ordre, on utilise simplement un ABR d'éléments $\in \mathcal{K} \times \mathcal{V}$ ordonnés par $<$ projeté sur la première coordonnée
- Informatiquement :

```
let compare2 : ('a * 'b) -> ('a * 'b) -> int =
    fun x y -> compare1 (fst x) (fst y)
```

- L'ordre sert uniquement de façon interne à la structure d'ABR, il peut n'avoir « aucun sens »
  - On peut souvent introduire un ordre *ad hoc* si besoin

bis

L'efficacité des opérations de tableau associatif seront données par celles des opérations d'ABR

- Pour des ABRs auto-équilibrés : coûts pire-cas en $O(\log N)$ avec N le nombre d'associations présentes dans la structure : toujours efficace

Commentaires

- Approche naturelle pour une implémentation fonctionnelle, mais aussi envisageable dans le cas impératif
- Donne gratuitement la possibilité de parcourir les associations dans l'ordre des clefs
- Pour certaines implémentations d'ABRs, donne des opérations ensemblistes (union, intersection…) efficaces projetées sur $\mathcal{K}$

Table des matières

Une observation

Si $\mathcal{K} = [\![n]\!]$, un « bête » tableau permet de réaliser efficacement *(conditions apply)* toutes les opérations d'un tableau associatif

Exemple

Si vous avez « correctement » fait les choses lundi :

```
bool row_fine(int g[81], size_t i) {
    bool set[10] = {};
    for (size_t j = 0; j < 9; j++) {
        int v = g[i*9 + j];
        if (v > 9 || v < 0 || (v > 0 && set[v])) {
            return false;
        }
        set[v] = true;
    }
    return true;
}
```

Efficace si…

Cette approche est :

- possible si $\mathcal{K} = [\![n]\!]$
- *efficace si* le nombre d'associations stockées est $\approx n$
  - Par exemple efficace pour `row_fine`
  - Mais pas si l'on veut stocker 10 associations *a priori* quelconques de $[\![2^{64}]\!] \times \mathcal{V}$

Objectif des tables de hachage

Rendre cette approche possible et efficace sans aucune contrainte sur $\mathcal{K}$

Idée 1 : stocker mod S

On suppose toujours $\mathcal{K} = [\![n]\!]$

Principe

- Lorsque l'on crée une table de hachage vide, on définit une *taille de stockage* S qui correspondra à la longueur d'un tableau `a`
  - Idéalement, $S \approx$ le nombre maximum d'associations pouvant être présentes dans la table à un instant donné
- Une association `(k, v)` est « stockée dans la case d'indice `k % S` » du tableau

Points d'attention

- On ne peut plus se contenter de stocker `v` dans `a[k % S]` (« quel `k` ? »)
- On peut devoir stocker plusieurs couples au même indice

Idée 1 *bis*

Une stratégie de résolution

Chaque case `i` du tableau stocke elle-même un tableau associatif !

- Celui des associations $\{(k, v) \mid k \equiv i \mod S\}$

Instanciations du sous-tableau associatif

- Par liste d'association (le plus courant)
- Avec un ABR (si les clefs sont ordonnables, mais ce sera le cas ultimement)
- Avec une table de hachage…

Efficacité

Dans le pire cas, donnée par celle du sous-tableau

- *Toutes* les clefs dans le grand tableau peuvent être dans la même classe $\mod S$; dans ce cas toutes les cases de `a` sont vides sauf une…

Idée 1 *ter*

En pratique

- On utilise souvent une liste d'association pour les sous-tableaux
  - On parle alors de résolution par chaînage des collisions
- Les mauvaises performances pire-cas sont évitées de façon probabiliste (grâce à l'idée 2 à venir)

(Variante : résolution par adressage ouvert)

- On stocke au plus *une* association par case de `a`
- Si la case `k % S` est prise on en cherche une autre (par exemple `(k + 1) % S`…), et échoue si `a` est plein (ou alors on l'étend….)
- Performances pire-cas exécrables, mais aussi évitables probabilistiquement

Idée 2 : toute clef est en fait une variable aléatoire entière

On n'impose plus aucune contrainte sur $\mathcal{K}$

Principe

- Lorsque l'on crée une table de hachage vide, on définit également une *fonction de hachage* $h : \mathcal{K} \to [\![n]\!]$ (pour un certain n fixe, typiquement 2^{64})
- L'association (k, v) est stockée **comme si sa clef était** $h(k)$
  - (Typiquement) en suivant l'**idée 1** précédente

Point d'attention

- On peut avoir des *collisions* pour h : $k, k' \neq k$ t.q. $h(k) = h(k')$ (toujours possible en principe pour certains $\mathcal{K}$ par le lemme des chaussettes)
  - Mais on peut les résoudre de la même façon que précédemment

Idée 2 *bis*

Efficacité

Dans le pire cas, *toutes* les clefs utilisées ont des images par h dans la même classe $\mod S$

- Pire cas inévitable si h est définie indépendamment des clefs et $\#\mathcal{K} \geq S^2$

Résolution ?

~~On se dit que pour les clefs qu'on a ça va bien se passer~~

On *tire la fonction de hachage uniformément au hasard* dans une famille de fonctions **à chaque fois que l'on crée une nouvelle table**

- La structure de données est *probabiliste* (de type « Las-Vegas », *cf.* l'année prochaine)

Idée 1 + 2 : analyse probabiliste

Espérance du nombre de collisions

- Si la famille de fonctions de hachage est « de bonne qualité » et suffisamment grande, on a $\Pr_H[H(k) = H(k' \neq k)] \approx 1/n$
- Et alors l'espérance du nombre de collisions pour q clefs est $\approx q \times (q-1)/2n$ (*via* la linéarité de l'espérance ; admis)

Espérance du coût des opérations pour une résolution par chaînage

Pour une famille de fonctions comme ci-dessus et une résolution par chaînage dans une table de hachage de S cases où l'on a inséré q associations, on peut montrer que l'espérance du coût des opérations (`add`, `del`, `find`…) est un $O(q/S)$

Table des matières

Tables de hachage en C

- Rien d'implémenté dans la bibliothèque standard
- Mais beaucoup d'implémentations *externes* de disponibles
- « Facile » d'implémenter efficacement une fonction de hachage (le langage est bien adapté)
  - Pour nous, sera (sans-doute ?) l'occasion de pratiquer la *compilation séparée*

Tables de hachage en OCaml

Implémentées dans la bibliothèque standard dans le module `Hashtbl`

Fonctions à connaître :

- `create : int -> ('a, 'b) Hashtbl.t`

  (accepte également un paramètre caché pour décider de tirer la fonction de hachage aléatoirement ou non ; ce n'est **pas** le cas par défaut par défaut)

- `add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit`

  `replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit`

  ajoute (avec occultation possible) ou remplace une association *(clef, valeur)* dans la table

- `remove : ('a, 'b) Hashtbl.t -> 'a -> unit`

  supprime la *dernière* association faisant intervenir la *clef*, s'il y en a une

Tables de hachage en OCaml *bis*

- `mem : ('a, 'b) Hasthbl.t -> 'a -> bool`

  teste la présence d'une association faisant intervenir la *clef*

- `find : ('a, 'b) Hashtbl.t -> 'a -> 'b`

  s'évalue en la (dernière) *valeur* associée à la *clef* si une telle association est présente, et lève une exception `Not_found` sinon

- `find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option`

  version optionnelle