Cours 4 : Tables de hachage

Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@lifl.fr

On dispose de données à ranger pour lesquelles on souhaite faire les opérations

- **d'ajout,**
- **de recherche**

de manière très efficace

Des solutions ?

- utiliser un tableau, oui mais :
  - si les données ne sont pas indicées par des entiers ?
  - si les données sont éparpillées ?
    par exemple la représentation du polynôme $x + x^{200}$ dans un tableau nécessite 200 cases si `t.(i)` représente le coefficient du monôme de degré i.
- utiliser une liste, oui mais :
  - la recherche est séquentielle !
    la recherche du dernier élément nécessite un parcours de liste
- utiliser un arbre binaire de recherche, oui mais :
  - il faut une relation d'ordre sur les données !
  - le temps de recherche dépend de l'ordre dans lequel les données ont été insérées !

L'exemple de l'annuaire

- Problème : on souhaite implanter un annuaire qui permette de retrouver facilement le numéro de téléphone en fonction du nom (on suppose qu'il n'y a pas d'homonymes)
- Solution proposée : avoir un tableau indicé par les noms (chaînes de caractères) et y stocker le numéro de téléphone
- Difficulté technique : on se sait pas créer des tableaux indicés par des chaînes de caractères
- Solution proposée : transformer chaque nom en nombre et stocker le numéro de téléphone dans un tableau indicé par ces nombres

Méthode de transformation du nom (codage en base 26) :

```
1  (* CU: le nom est en majuscules *)
2  let h nom =
3    let n = (String.length nom)
4    and res = ref 0 in
5    for i = 0 to n - 1 do
6      res := !res * 26 + ((int_of_char nom.[i]) - 65 + 1)
7    done;
8    !res
```

Exemple :

VARRE $\mapsto ((((((22) * 26 + 1) * 26 + 18) * 26) + 18) * 26) + 5 = 10083689$

Remarques

- Inconvénients :
  - si la longueur des noms n'est pas bornée h(nom) peut prendre une infinité de valeurs
  - si la longueur est bornée, le tableau devra être très grand pour 10 lettres, 141,167,095,653,376 valeurs possibles ! ce qui est inutile dans la plupart des cas ... et qui de toute façon ne rentre pas dans la mémoire d'un téléphone portable
- Solution :
  - se contenter d'un tableau plus petit, ne pouvant tout contenir, mais de capacité suffisante pour l'usage ciblé
  - appliquer un modulo sur le résultat de la fonction h de manière à avoir des valeurs inférieures à la longueur tableau
- Difficulté : deux noms, même non homonymes, peuvent se voir associer le même entier : il y a collision.

Une table de hachage est une structure de données dont le cahier des charges est le suivant :

- permet l'association d'une valeur à une clé
  dans l'exemple les valeurs sont des numéros de téléphone et les clés des noms
- permet un accès rapide à la valeur à partir de la clé (comme un tableau)
- permet l'insertion rapide (comme dans une liste)

Vocabulaire

- clé : l'objet auquel est associé la valeur
- valeur : l'objet que l'on souhaite stocker
- table : la structure dans laquelle sont rangées les associations <clé,valeur> à des adresses
- alvéole : case qui se trouve à une adresse de la table
- la fonction de hachage : transforme une clé en une adresse dans la table

`http://groups.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm`

Dans l'exemple de l'annuaire :

- clé = nom
- valeur = numéro de téléphone
- fonction de hachage :

$$h(k) = \left(\sum_{i=0}^{n-1} \text{pos}(k.(n-i)) \times 26^{i-1} \right) \mod M$$

avec M la capacité de la table et pos le rang de la lettre dans l'alphabet

Fonction de hachage

La fonction de hachage permet de transformer une clé en adresse :

- adressage direct :
  - la clé = l'adresse
- adressage indirect :
  - l'adresse est une fonction de la clé
- la fonction doit être simple à calculer, pour rester efficace en temps

La fonction de hachage doit être **uniforme** pour assurer un bon comportement : chaque clé doit avoir la même chance d'être rangée dans chaque alvéole.

Méthode de la division

$$h(k) = f(k) \mod M$$

- le choix de M peut dépendre de la distribution des clés
  si on insère des clés k telles que $f(k)$ est multiple de 10 on n'aura pas intérêt à choisir un M multiple de 10
- en général, choisir un nombre premier pour M donne de bons résultats
- on pourra choisir M en fonction de la performance qu'on souhaite
  si on insère 2000 mots et que l'objectif est d'examiner en moyenne 3 alvéoles dans le cas d'une recherche infructueuse, on pourra choisir $M = 673$.

Dans ce cas, la qualité du hachage dépend de la taille de la table.

Remarque : on distingue rarement le calcul de f et de h

Traitement des chaînes de caractères

- alphabet de taille σ
- on considère un mot u, u_i est la i-ème lettre de u, i varie de 0 à $n-1$
- $\text{val}(u_i)$ est le rang de la lettre u_i dans l'alphabet

$$h(u) = \left(\sum_{i=1}^{p} \text{val}(u_i) \times \sigma^{p-i-1} \right) \mod M$$

Note

Dans certains langages de programmation qui fournissent des tables de hachage, le calcul du nombre associé à une chaîne de caractères est implicite.

Traitement d'objets quelconques

- il faut avoir une valeur unique pour chaque objet
- dans certains langages ce calcul est implicite (souvent en utilisant l'adresse mémoire où est rangé l'objet)
- mais attention, deux objets créés identiquement n'ont pas nécessairement même hash code,
  en Java par exemple il est nécessaire de redéfinir la méthode `hashCode`
- mais encore attention, il peut aussi être nécessaire de redéfinir l'égalite au sens logique des objets,
  en Java la méthode `equals`

(voir cours de POO pour les tables de hachage en Java)

En conclusion, il faut être en capacité

- de **calculer une adresse** (d'une alvéole) à partir de la clé
- de **tester l'égalité** entre deux clés

Traitement des collisions

Si deux clés (non homonymes) aboutissent à la même adresse : il y a collision.
Stratégie vis-à-vis des collisions :

1. adressage ouvert, chercher une autre alvéole de la table qui est vide :
   1. parcours itératif de la table,
   2. parcours pseudo-aléatoire
2. chaînage :
   1. augmenter la capacité de stockage,
   2. chaque alvéole de la table devient une liste

Les primitives de manipulation

Les primitives de base pour manipuler une table de hachage :

- insertion d'un couple <clé,valeur>
- suppression d'une clé (et donc de sa valeur associée)
- recherche de la valeur associée à une clé

En OCaml

Module `Hashtbl`

```
1  type ('a, 'b) t
2  (** The type of hash tables from type ['a] to type ['b]. *)
3
4  val create : int -> ('a, 'b) t
5  (** [Hashtbl.create n] creates a new, empty hash table, with
6     initial size [n]. For best results, [n] should be on the
7     order of the expected number of elements that will be in
8     the table. The table grows as needed, so [n] is just an
9     initial guess. *)
10
11 val add : ('a, 'b) t -> 'a -> 'b -> unit
12 (** [Hashtbl.add tbl x y] adds a binding of [x] to [y] in table [tbl].
13    Previous bindings for [x] are not removed, but simply
14    hidden. That is, after performing {!Hashtbl.remove}[ tbl x],
15    the previous binding for [x], if any, is restored.
16    (Same behavior as with association lists.) *)
17
18 val find : ('a, 'b) t -> 'a -> 'b
19 (** [Hashtbl.find tbl x] returns the current binding of [x] in [tbl],
20    or raises [Not_found] if no such binding exists. *)
```

En OCaml

```
val find_all : ('a, 'b) t -> 'a -> 'b list
(** [Hashtbl.find_all tbl x] returns the list of all data
   associated with [x] in [tbl].
   The current binding is returned first, then the previous
   bindings, in reverse order of introduction in the table. *)

val mem : ('a, 'b) t -> 'a -> bool
(** [Hashtbl.mem tbl x] checks if [x] is bound in [tbl]. *)

val remove : ('a, 'b) t -> 'a -> unit
(** [Hashtbl.remove tbl x] removes the current binding of [x] in [tbl],
   restoring the previous binding if it exists.
   It does nothing if [x] is not bound in [tbl]. *)

val replace : ('a, 'b) t -> 'a -> 'b -> unit
(** [Hashtbl.replace tbl x y] replaces the current binding of [x]
   in [tbl] by a binding of [x] to [y]. If [x] is unbound in [tbl],
   a binding of [x] to [y] is added to [tbl].
   This is functionally equivalent to {!Hashtbl.remove}[ tbl x]
   followed by {!Hashtbl.add}[ tbl x y]. *)
```


Exemple

```
let m = 0;;
let t = create m;;
add t "DUPONT" "Jean";;
add t "MARTIN" "Luc";;
add t "DUPONT" "Robert";;
find t "MARTIN";;
find t "DUPONT";;
find_all t "DUPONT";;
remove t "DUPONT" ;;
find_all t "DUPONT";;
add t "DUPONT" "Robert";;
find_all t "DUPONT";;
replace t "DUPONT" "Pierre" ;
find_all t "DUPONT";;
```

```
#     val m : int = 0
# val t : ('_a, '_b) Hashtbl.t = <abstr>
# - : unit = ()
# - : unit = ()
# - : unit = ()
# - : string = "Luc"
# - : string = "Robert"
# - : string list = ["Robert"; "Jean"]
# - : unit = ()
# - : string list = ["Jean"]
# - : unit = ()
# - : string list = ["Robert"; "Jean"]
# - : unit = ()
# - : string list = ["Pierre"; "Jean"]
```


Parcours de toutes les clés/valeurs de la table

- il n'y a pas moyen d'avoir la liste des clés
- au lieu de cela propose une procédure pour répéter l'application d'une procédure sur chaque couple clé/valeur

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit

Hashtbl.iter f tbl applies f to all bindings in table
tbl. f receives the key as first argument, and the
associated value as second argument. Each binding is
presented exactly once to f.
```

```
(* affiche a l'ecran k et v *)
let afficher k v =
  Printf.printf "NOM: %s NUMERO: %s\n" k v
;;

let afficher_contenu_table t =
  iter afficher t
;;
```


II **Faisons le point**

d'un point de vue pratique :

- l'utilité d'une telle structure de données
- l'implantation faite en OCaml

d'un point de vue théorique :

- le principe d'une table de hachage
- la nécessité de disposer d'une fonction de hachage
- le problème des collisions

Adressage ouvert

- toutes les valeurs sont conservées dans la table
- si une alvéole est déjà occupée, on en cherche une autre, libre : c'est le sondage

Recherche d'une alvéole libre

- toutes les valeurs sont conservées dans la table
- si une alvéole est déjà occupée, on en cherche une autre, libre,

on utilise une fonction de hachage à deux paramètres $h'(k, i)$, qui donne l'adresse de rangement d'une clé k au bout de i tentatives

- h' utilise l'adresse de l'adresse primaire donnée par h
- h' doit permettre de parcourir toutes les adresses de la table, sinon la table est sous-occupée
- lors de la recherche, il faudra parcourir successivement les emplacements pouvant contenir la clé, on aura un échec à la première alvéole vide

La fonction h' permet de réaliser des sondages successifs dans la table.

Adressage ouvert - sondage linéaire

Sondage linéaire

$h'(k, i) = (h(k) + i) \mod M$

Les clés sont des mots, la valeur le nombre d'occurrences dans un texte, l'adresse est calculée sur le rang de la dernière lettre du mot modulo 5.

0	hachage ,12
1	parcouru,7
2	rang ,81
3	adresse ,24
4	fonction ,36

k	v	$h(k)$	$h'(k,i) = h(k) + i \mod 5$ $i=0$	1	2	3
hachage	12	0	0			
fonction	36	4	4			
parcouru	7	1	1			
rang	81	2	2			
adresse	24	0	0	1	2	3

- suppression de la clé 'hachage'
- la clé 'adresse' est-elle dans la table ?
- h(adresse) = 0, la alvéole est vide ⇒ 'adresse' n'est pas dans la table