Cours 4 : Tables de hachage

Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@lifl.fr

Problématique

- on dispose de données à ranger pour lesquelles on souhaite faire les opérations d'ajout, de recherche et de suppression de manière très efficace
- 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

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 et y stocker le numéro de téléphone
- Difficulté technique : on se sait pas créer des tableaux indicés par les noms
- 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) :

$$h(\text{nom}) = \sum_{i=1}^{n} \text{val}(\text{nom}(n-i)) \times 26^{i-1}$$

où nom(i) est la i-ème lettre du nom et val représente le rang de la lettre (A=0, Z=25).

Exemple

$$\begin{aligned} h(VARRE) &= val(V) \times 26^4 + val(A) \times 26^3 + val(R) \times 26^2 + val(R) \times 26 + val(E) \\ &= 21 \times 26^4 + 0 \times 26^3 + 17 \times 26^2 + 17 \times 26 + 4 \\ &= 9608009 \end{aligned}$$

Remarques

- Inconvénients :
  - si la longueur des noms n'est pas bornée, k(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
- 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 k de manière à avoir des valeurs entre 1 et la longueur du tableau
- Difficulté : deux noms, même non homonymes, peuvent se voir associer le même entier : il y a collision.

Table de hachage

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

Dans l'exemple de l'annuaire :

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

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

avec M la capacité de la table

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

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

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

- le choix de M peut dépendre de la distribution des clés
  Si on insère des clés k multiples 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 $M = 701$.

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

Note

On distingue rarement le calcul de h et de h modulo la longueur du tableau.

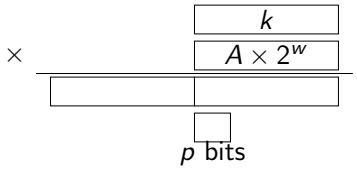
Méthode de la multiplication

$$h(k) = \lfloor M(kA \mod 1) \rfloor, \quad 0 < A < 1$$

- $kA \mod 1 = kA - \lfloor kA \rfloor$ (on ne travaille que sur la partie entière)
- fonctionne sur des clés qui sont des entiers

Pour le calcul :

- on suppose que la clé k est représentée sur un mot machine de w bits
- la taille de la table est une puissance de 2 : $M = 2^p$



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 1 à p
- 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 caractère est implicite.

Traitement des collisions

Si deux clés (non homonymes) aboutissent à la même adresse : il y a collision.

Comportement vis-à-vis des collisions :

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

Les primitives de manipulation d'une table de hachage

Les primitives de base pour manipuler une table :

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

Adressage ouvert

Caractéristiques :

- toutes les valeurs sont conservées dans la table
- si une case 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

- la fonction h' utilise la valeur de l'adresse primaire donnée par h
- la fonction 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 case vide

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

Adressage ouvert

Sondage linéaire

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

Insertion

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 : $h(k) = \text{val}(k[\text{high}(k)]) \mod 5$

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

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

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

Remarque sur le sondage linéaire

Inconvénients :

- deux clés ayant même adresse primaire auront la même suite d'adresses
- formation de groupements : l'insertion successive de clés avec la même adresse primaire remplit une zone contigüe de la table

une collision en adresse primaire ⇒ une collision sur les autres adresses

Avantages :

- très simple à calculer
- fonctionne correctement pour un taux de remplissage moyen

Taux de remplissage

C'est le rapport entre le nombre de clés effectivement dans la table et sa capacité :

$$\tau = \frac{n}{M}$$

Adressage ouvert

Sondage quadratique

$h'(k,i) = (h(k) + a \times i + b \times i^2) \mod M$, a et b des constantes non nulles

Insertion

Les clés sont des mots, l'adresse est calculée sur le rang de la dernière lettre du mot modulo 5. Prenons $a = 1$ et $b = 2$.

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

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

Note: si on avait choisi $a = b = 1$ on aurait eu la séquence
$0 \to 2 \to 1 \to 2 \to 0 \to 0 \to 2 \to 1 \to 2 \to 0 \cdots$

Remarques sur le sondage quadratique

Inconvénients :

- difficulté de choisir les constantes a et b et la taille de la table pour s'assurer de parcourir toute la table

Exemple de parcours (a=b=1)

i	0	1	2	3	4	5	6
$i + i^2$	0	2	5	12	20	30	42
$M = 10$	0	2	5	2	0	0	2
$M = 11$	0	2	5	1	9	8	9
$M = 13$	0	2	5	12	7	4	3

- comme pour le sondage linéaire, deux clés ayant même adresse primaire auront la même suite d'adresses

Avantages :

- plus efficace que le sondage linéaire
- moins d'effet de formation de groupements

Adressage ouvert

On utilise deux fonctions de hachage primaire.

Double hachage

$h'(k,i) = (h_1(k) + i \times h_2(k)) \mod M$

Insertion

Les clés sont des mots, h_1 est calculée sur le rang de la dernière lettre du mot modulo 5 alors que h_2 est calculée sur le rang de l'avant-dernière lettre.

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

k	v	$h_1(k)$	$h_2(k)$	$h'(k,i) = (h_1(k) + ih_2(k)) \mod 5$ $i=0$	$i=1$	$i=2$
hachage	12	0	2	0		
fonction	36	4	0	4		
parcouru	7	1	3	1		
rang	81	2	4	2		
adresse	24	0	4	0	4	3

Remarques sur le double hachage

Inconvénients :

- choix des deux fonctions de hachage

Pour être sûr de parcourir toute la table, il faut que $h_2(k)$ soit premier avec M. Un schéma classique :

$$M = 2^p,$$
$$h_1(k) = k \mod M,$$
$$h_2(k) = 1 + (k \mod M')$$

avec M' un peut plus petit que M.

Avantages :

- contrairement aux précédents sondages, les suites d'adresses ne sont pas nécessairement linéaires : fourni de l'ordre de M^2 séquences de sondage au lieu de M
  Comme dans l'exemple, les deux fonctions de hachage n'utilisent pas la même donnée de la clé
- bien meilleur que les sondages précédents

Pire des cas et meilleur des cas

Analyse de la complexité de l'insertion et de la recherche

Meilleur des cas

La case d'insertion est vide (insertion), il existe une seule clé k dans la table ayant pour adresse $h'(k,0)$ (recherche), on est en $\Theta(1)$.

Pire des cas

Il ne reste plus de case libre, on aura parcouru toute la table, on est en $\Omega(M)$.
Si la fonction h' est correctement réalisée, chaque case est parcourue un nombre fini constant de fois, on est alors en $\Theta(M)$.

Et en moyenne ?

Les types et variables utilisés

```
type
  // association <cle,valeur>
  COUPLE = record
    cle : CLE;
    val : VALEUR
  end {record};

var
   // la table
   table       : array[NUMERO_CASE] of COUPLE;
   // utilise pour verifier si une alveole est deja remplie
   occupe      : array[NUMERO_CASE] of BOOLEAN;
```


Code de la fonction d'insertion

```
procedure inserer (k : CLE; v : VALEUR);
var
   tentative : CARDINAL;
   adresse : NUMERO_CASE;
begin
   tentative := 0;
   adresse := hprime(k,tentative);
   // recherche d'une alveole libre
   while occupe[adresse] and (table[adresse].cle <> k) and (tentative < CAPACITE) do
      inc(tentative);
      adresse := hprime(k,tentative);
   end {while};
   // 3 cas : table pleine, cle presente, cle non presente
   if (tentative < CAPACITE) then begin
      if not occupe[adresse] then begin
         occupe[adresse] := True;
         table[adresse].cle := k;
         table[adresse].val := v;
      end else begin
         table[adresse].val := v;
      end {if};
   end else begin
      raise TablePleine.create('Table pleine');
   end {if};
end {inserer};
```


Code de la fonction de recherche

```
   function rechercher (k : CLE) : VALEUR;
   var
      tentative : CARDINAL;
      adresse : NUMERO_CASE;
   begin
      tentative := 0;
      adresse := hprime(k,tentative);
      // recherche de l'alveole contenant la cle
      while occupe[adresse] and (table[adresse].cle <> k) do begin
         inc(tentative);
         adresse := hprime(k,tentative);
      end {while};
      if occupe[adresse] then
         rechercher := table[adresse].val
      else
         raise CleAbsente.create('Cle non trouvee');
   end {valeur};
```

Complexité en moyenne de la recherche

Deux cas :

- recherche infructueuse :
  - on recherche une clé qui n'est pas dans la table
  - il faudra parcourir la suite des adresses donnée par h'
  - dans le pire des cas on parcoura les n alvéoles remplies de la table
- recherche fructueuse :
  - on recherche une des n clés insérées dans la table

Recherche infructueuse 1/2

On considère qu'il y a n éléments dans la table.

- il y a un accès à $h'(k,0)$
  avec une probabilité 1
- l'accès à $h'(k,1)$ se fait ssi $h'(k,0)$ était occuppée
  avec une probabilité de $p_1 = \frac{n}{M}$
- l'accès à $h'(k,2)$ se fait ssi $h'(k,1)$ et $h'(k,0)$ étaient occuppées
  avec une probabilité de $p_2 = \frac{n}{M} \times \frac{n-1}{M-1}$
- on accède à $h'(k,i)$ ssi les i adresses précédentes étaient occupées
  avec une probabilité de $p_i = \frac{n}{M} \times \frac{n-1}{M-1} \times \cdots \times \frac{n-i+1}{M-i+1}$

On peut montrer que :

$$p_i \leq \left(\frac{n}{M}\right)^i = \tau^i$$

Recherche infructueuse 2/2

- le nombre moyen d'accès à l'adresse $h'(k,0)$ est 1×1
  1 accès avec une probabilité de 1
- le nombre moyen d'accès à l'adresse $h'(k,1)$ est $1 \times p_1$
  1 accès avec une probabilité de p_1
- ...
- le nombre d'accès moyen s'exprime ainsi :

$$\sum_{i=0}^{n-1} p_i \leq \sum_{i=0}^{n-1} \tau^i \leq \frac{1}{1-\tau}$$

taux	nb moyen accès	taux	nb moyen accès
0.5	2	0.8	5
0.75	4	0.9	10
		0.99	100

Analyse de l'insertion en moyenne

- insérer une clé c'est rechercher une alvéole libre
- insérer la i-ème clé, c'est donc une recherche infructueuse dans une table contenant $i-1$ éléments
- on peut majorer le nombre moyen d'accès a_i :

$$a_i \leq \frac{1}{1-\frac{i-1}{M}} = \frac{M}{M-i+1}$$

$\frac{i-1}{M}$ c'est la taux de remplissage avant l'insertion de la i-ème clé

Recherche fructueuse

- on dispose de n clés dans la table, on suppose que chaque clé a la même probabilité d'être recherchée
- la recherche du i-ème élément se fait avec un parcours identique à l'insertion
- le nombre moyen d'accès s'exprime ainsi :

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} a_i &\leq \frac{1}{n}\sum_{i=1}^{n}\frac{M}{M-i+1}\\ &\leq \frac{M}{n}\sum_{i=0}^{n-1}\frac{1}{M-i} = \frac{M}{n}\sum_{i=M-n+1}^{M}\frac{1}{i}\\ &\leq \frac{M}{n}\ln\left(\frac{M}{M-n}\right)\end{aligned}$$

taux	nb. moy. accès	taux	nb. moy. accès
0.5	1.386	0.8	2.012
0.75	1.848	0.9	2.558
		0.99	4.652

Gestion des suppressions

- tout ce qui a été énoncé auparavant est vrai si il n'y a pas d'élements supprimés dans la table
- comment gérer les suppressions ?
  - marquer l'alvéole d'un état spécial 'supprimé'
  - décaler les cases de la série des $h'(k,i)$
    - ⋆ réalisable avec un sondage dont la suite d'adresses est déterminée par $h'(k,0)$
    - ⋆ impossible dans les autres cas (cela reviendrait à réordonner toute la table)
- complexité (dans le cas du marquage) : c'est le même coût qu'une recherche fructueuse.

Adressage ouvert

Sondage linéaire

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

Insertion

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 : $h(k) = \text{val}(k[\text{high}(k)]) \mod 5$

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

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

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

Gestion des suppressions

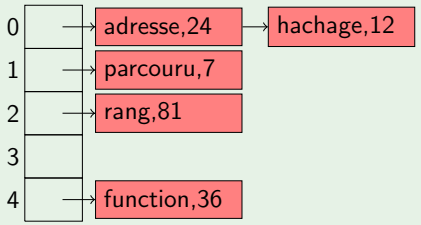
- tout ce qui a été énoncé auparavant est vrai si il n'y a pas d'élements supprimés dans la table
- comment gérer les suppressions ?
  - marquer l'alvéole d'un état spécial 'supprimé'
  - décaler les cases de la série des $h'(k,i)$
    - ⋆ réalisable avec un sondage dont la suite d'adresses est déterminée par $h'(k,0)$
    - ⋆ impossible dans les autres cas (cela reviendrait à réordonner toute la table)
- complexité (dans le cas du marquage) : c'est le même coût qu'une recherche fructueuse.

Résolution des collisions par chaînage

Schéma de principe

Insertion

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 :
$h(k) = \text{val}(k[\text{high}(k)]) \mod 5$



k	v	$h(k)$
hachage	12	0
fonction	36	4
parcouru	7	1
rang	81	2
adresse	24	0

Recherche d'une clé

```
function rechercher (k : CLE) : VALEUR;
var
   adresse : NUMERO_CASE;
   p       : LISTE;
begin
   // acces direct a la bonne alveole
   adresse := h(k,0);
   // recherche de la cle dans la liste
   p := table[adresse];
   while not estListeVide(p) and (tete(p).cle <> k) do begin
      p := reste(p);
   end {while};
   if not estListeVide(p) then
      rechercher := p.tete.val
   else
      raise CleAbsente.create('Cle non trouvee');
end {valeur};
```


Coût de la recherche infructueuse d'un élément

- l'accès à la bonne alvéole est en $\Theta(1)$
- la recherche de l'élément dans la liste va dépendre de la longueur de la liste :
  - $\Theta(1)$ pour le meilleur des cas (où aucune clé ayant même adresse n'a été insérée dans la table)
  - $\Theta(n)$ pour le pire des cas (où toutes les clés insérées ont même adresse, donc dans une seule liste, et la clé recherchée a même adresse)

En moyenne :

- si on suppose que la fonction de hachage est uniforme, les n clés auront été hachées équitablement dans les alvéoles
- les listes auront donc une longueur de l'ordre de $\frac{n}{M} = \tau$
- le temps moyen de la recherche est en $\Theta(1 + \tau)$ (le 1 est pour l'accès à l'alvéole)

Coût de la recherche fructueuse d'un élément

- l'accès à la bonne alvéole est en $\Theta(1)$
- la recherche de l'élément dans la liste va dépendre de la longueur de la liste :
  - $\Theta(1)$ pour le meilleur des cas (clé en première position de la liste)
  - $\Theta(n)$ pour le pire des cas (où toutes les clés insérées ont même adresse, donc dans une seule liste, et la clé recherchée est en fin de liste)

En moyenne :

- si on suppose que la fonction de hachage est uniforme, les n clés auront été hachées équitablement dans les alvéoles
- les listes auront donc une longueur de l'ordre de $\frac{n}{M} = \tau$
- le temps moyen de la recherche est aussi en $\Theta(1 + \tau)$

Insertion d'une clé

```
procedure inserer (k : CLE; v : VALEUR);
var
   adresse : NUMERO_CASE;
   c       : COUPLE;
begin
   adresse := h(k);
   c.cle := k;
   c.val := v;
   try
      rechercher(k);
   except
      // si la cle n'est pas trouvee
      table[adresse] := ajouterEnTete(table[adresse],c);
   end;
end {inserer};
```

Note: dans cette version, si la clé est déjà présente on ne fait rien.

Coût de l'insertion et de la suppression

- insertion :
  - trouver l'alvéole : $\Theta(1)$
  - tester l'existence : $\Theta(\frac{n}{M})$
  - ajouter en tête de liste : $\Theta(1)$
- suppression :
  - trouver l'alvéole : $\Theta(1)$
  - trouver la position dans la liste : $\Theta(\frac{n}{M})$
  - supprimer : $\Theta(1)$

on se souvient que la suppression d'un élément dans une liste, une fois l'élément trouvé, est en temps constant

A retenir

- la fonction de hachage doit être uniforme pour permettre une bonne répartition des clés et donc une bonne performance
- pour ranger des couples <clé,valeur>, il est impératif :
  - de pouvoir calculer une adresse à partir de la clé
  - d'être capable de tester l'égalité entre deux clés

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