



Développement Efficace

4. Hachage

BUT INFO 2A

David Auger

IUT de Vélizy - UVSQ

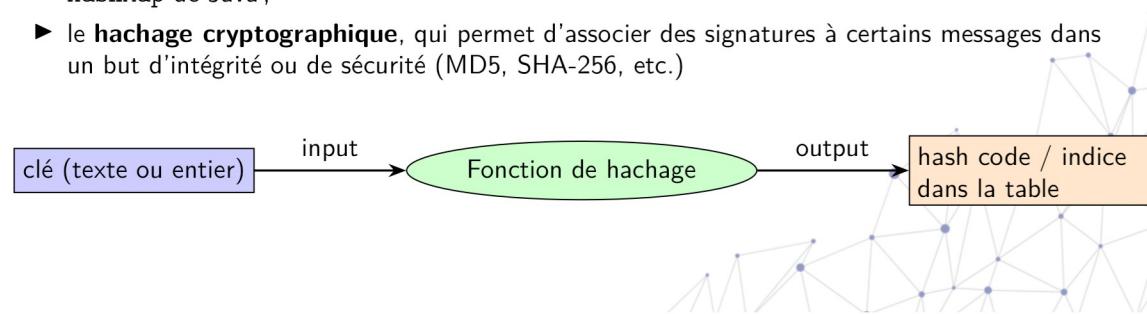


4.1- Concepts de Hachage

Le principe du hachage, qui utilise des **fonctions de hachage** (ou "tables"), est d'associer à chaque **valeur** ou **clé** possible un entier (ou un texte) de taille fixe, son **hash** ou **code/valeur de hachage**. La clé est la partie des données qui sert à les indexer (en pratique, texte ou numérique).

On distingue essentiellement deux types de hachage :

- ▶ le **hachage algorithmique**, qui permet de réaliser des structures de données efficaces pour modéliser des ensembles mathématiques, comme les set ou dict de python ou HashSet et HashMap de Java ;
- ▶ le **hachage cryptographique**, qui permet d'associer des signatures à certains messages dans un but d'intégrité ou de sécurité (MD5, SHA-256, etc.)



4.2- Hachage algorithmique : principe de base

Objectif

Stocker des clés efficacement de sorte à réaliser rapidement les opérations insertion, recherche et suppression. Contrairement au cas des Piles/Files, on veut avoir une structure analogue à un ensemble mathématique où il n'y a pas d'ordre préétabli type LIFO ou FIFO.

Pour cela, on se donne un tableau statique fixé de taille N . Pour insérer une clé c , on calcule une valeur de hachage $h(c)$ (un entier plus ou moins grand) et on place la donnée dans le tableau à l'indice $h(c) \% N$.



4.2- Hachage algorithmique : principe de base

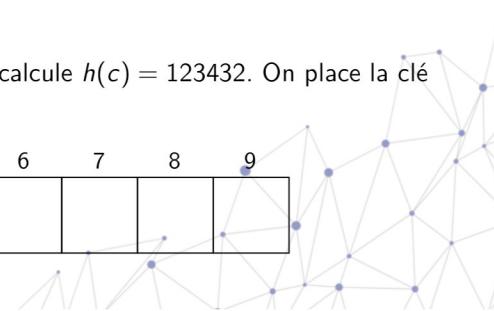
Objectif

Stocker des clés efficacement de sorte à réaliser rapidement les opérations insertion, recherche et suppression. Contrairement au cas des Piles/Files, on veut avoir une structure analogue à un ensemble mathématique où il n'y a pas d'ordre préétabli type LIFO ou FIFO.

Pour cela, on se donne un tableau statique fixé de taille N . Pour insérer une clé c , on calcule une valeur de hachage $h(c)$ (un entier plus ou moins grand) et on place la donnée dans le tableau à l'indice $h(c) \% N$.

Exemple : $N=10$. On veut ajouter la clé $c="algo"$. On calcule $h(c) = 123432$. On place la clé dans la case $h(c)\%10 = 2$.

0	1	2	3	4	5	6	7	8	9
.	.	"algo"	



4.3- Fonction de hachage

Objectif : associer un entier $k = h(c)$ à toute clé c .

Pour cela il faut définir l'univers des clés c'est-à-dire l'ensemble des clés potentielles qui peuvent se présenter, autrement dit le domaine de définition de h .

Exemples

- ▶ pour un entier n , prendre $h(n) = n$.
- ▶ pour une liste d'entiers $L = [n_1, n_2, \dots, n_k]$, prendre $h(L) = n_1 + n_2 + \dots + n_k$.
- ▶ pour une chaîne de caractères ascii s , définir $h(s)$ comme la somme des valeurs ascii des caractères (en python on peut plus généralement considérer l'entier `ord(c)` associé à chaque caractère c).

Distinguer

- ▶ la valeur de hachage (ou *hash*) qui est $h(c)$, un entier pouvant être grand
- ▶ l'indice de hachage qui est $h(c) \% N$ où N est la taille de la table.

4.4- Fonction de hachage

Un exemple de fonction de hachage récursive souvent utilisée.

N : taille du tableau

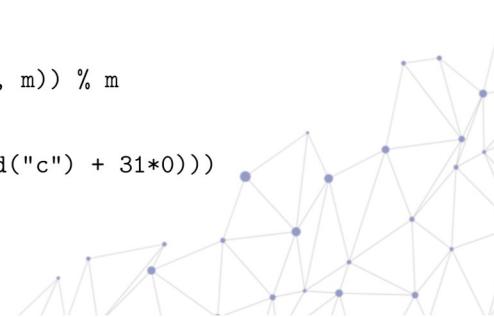
p : un petit entier premier (ex 31 ou 37)

m : un grand modulo (ex 2^{32}) qui contrôle la taille des valeurs

```
def hash_rec(s, p=31, m=2**32):
    if not s:
        return 0
    return (ord(s[0]) + p * hash_rec(s[1:], p, m)) % m
```

Exemple avec les valeurs ci-dessus

```
h("abc") = ord("a") + 31*(ord("b") + 31 * (ord("c") + 31*0)))
```



4.5- Collisions



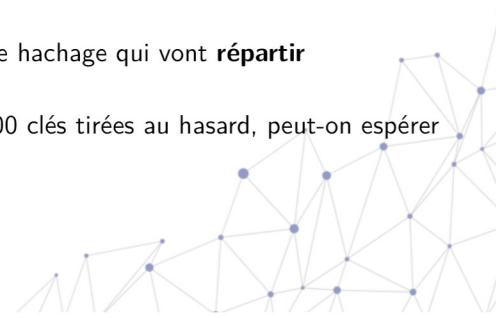
Définition

Une collision arrive quand deux clés distinctes c_1 et c_2 ont le même indice de hachage

$$h(c_1) \% N = h(c_2) \% N.$$

- ▶ Ceci arrive obligatoirement si l'univers des clés potentielles est plus grand que la taille de la table.
- ▶ Pour éviter cela on essaie de trouver des fonctions de hachage qui vont **répartir uniformément** les clés dans la table.

Exemple : si la table a une taille 10000 et qu'on insère 100 clés tirées au hasard, peut-on espérer éviter les collisions ?



4.6- Paradoxe des anniversaires

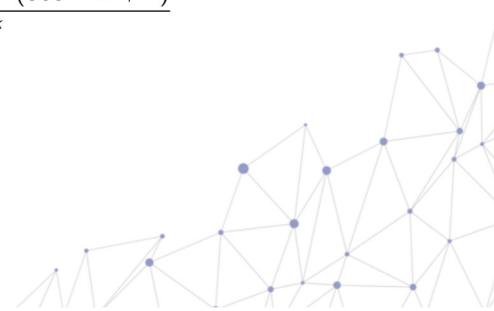
Dans un groupe de k personnes tirées uniformément au hasard (on supposera les dates de naissance équiprobables sur 365 jours), quelle est la probabilité que 2 personnes aient le même anniversaire ?

Soit D l'événement "toutes les dates sont différentes". On cherche

$$\begin{aligned} P(\bar{D}) &= 1 - P(D) \\ &= 1 - \frac{365 \cdot 364 \cdot 363 \cdots \cdot (365 - k + 1)}{365^k} \end{aligned}$$

Résultats

- ▶ pour $k=10$, $P(\bar{D}) \approx 0,12$
- ▶ pour $k=23$, $P(\bar{D}) \approx 0,5$
- ▶ pour $k=60$, $P(\bar{D}) \approx 0,99$



4.7- Paradoxe des anniversaires

Nombre moyens de collisions

Supposons de l'indice de hachage répartisse uniformément les clés tirées au hasard dans la table de taille N . On insère k clés. On veut estimer le nombre moyen de collisions, c'est à dire le nombre de couples (i, j) avec $1 \leq i < j \leq n$ tels que $h(c_i) \% N = h(c_j) \% dN$.

- ▶ chaque couple de clés (c_i, c_j) a une probabilité $\frac{1}{N}$ de collisionner.
- ▶ Il y a $\binom{k}{2} = \frac{k(k-1)}{2}$ couples.
- ▶ Ce qui donne un nombre moyen de collisions de

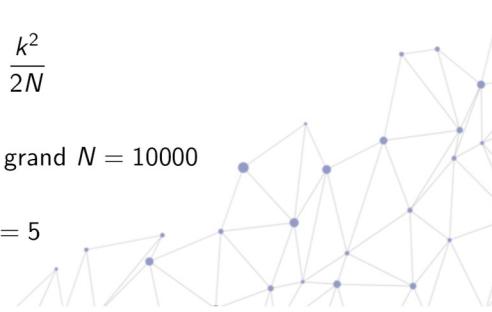
$$\frac{k(k-1)}{2} \cdot \frac{1}{N} \approx \frac{k^2}{2N}$$

application numérique

$k = 1000$ clés à stocker ; on prend le tableau 10 fois plus grand $N = 10000$

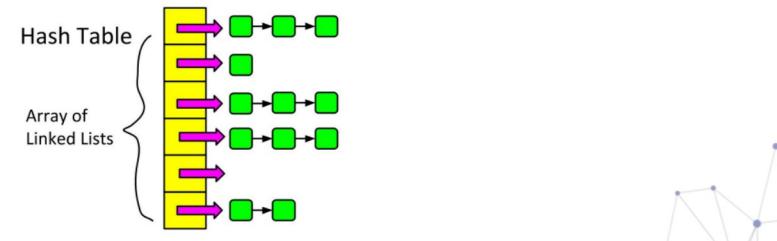
$$\frac{1000^2}{2 \cdot 10000} = \frac{10^6}{2 \cdot 10^5} = 5$$

Conclusion : il faut gérer les collisions !



4.8- Résolution par chaînage externe

Dans le **Hachage avec chaînage externe**, on ne stocke pas directement les clés dans la table. Chaque case du tableau contient en fait (un pointeur vers) une liste chaînée. Une clé c d'indice de hachage $i = (h(c) \% N)$ est insérée en début de liste à l'indice i .



- ▶ temps pour insérer : $O(1)$
- ▶ temps pour chercher/supprimer $O(\ell)$ où ℓ est la longueur de la plus longue chaîne (dans le pire des cas). Mais en moyenne il y a en moyenne $\ell = k/N$ clés par liste chaînée .

Note : la valeur k/N est appelée la **charge** de la table.

4.9- Résolution par chaînage interne ou adressage ouvert

Principe

Une seule clé est stockée dans chaque case de la table. Au lieu d'une seule fonction $h(c)$, on va construire une suite de fonctions (dite fonction d'adressage)

$$h_0(c), h_1(c), h_2(c), \dots$$

Algorithme :

- ▶ $t = 0$
- ▶ tant que la case d'indice $i_t = h_t(c) \% N$ est non vide :
 - ▶ $t = t + 1$
 - ▶ insérer dans la case $h_t(c) \% N$

Exemple du sondage linéaire : On pose

$$h_t(c) = h(c) + t$$

On essaie dans la case $h(c) \% N$, puis la case suivante, etc (de façon circulaire).



4.10- Autres types de sondage

Le sondage linéaire a tendance à créer des "grappes" de données dans la table au lieu de les répartir. La plupart des applications utilisent une fonction d'adressage qui pratique davantage le sondage de façon pseudo-aléatoire dans la table.

Adressage utilisé par python

Soit c une clé à insérer, $h(c)$ sa valeur de hachage. On suppose que la table a une longueur $N = 2^k$. La suite i_0, i_1, i_2, \dots des indices à tester est donnée par

Initialisation :

$$i_0 = h(c)\%N$$

$$p_0 = h(c)$$

Pour $t \geq 0$:

$$i_{t+1} = (5 \cdot i_t + 1 + p_t)\%N$$

$$p_{t+1} = p_t // 32$$

Le nombre p_t est un "perturbateur" qui permet que des valeurs de hachage proches donnent au final une suite différente.

4.11- Hachage interne : adresse ouverte (open addressing)

Soit (i_t) la suite des indices dans le sondage. En général $i_t = h_t(c)\%N$.

► Pour insérer : on essaie i_0 , puis i_1 , i_2 , etc. jusqu'à trouver une case disponible.

► Pour rechercher : on regarde si la clé est en i_0 . Trois possibilités :

1. la clé est trouvée
2. la cellule est vide – alors la clé n'est pas dans la table.
3. la cellule contient une autre clé : reprendre la même procédure en i_1

Et on recommence ainsi si nécessaire avec i_1, i_2, \dots .

► Pour supprimer : on fait comme pour insérer. Cependant, supprimer une clé en la remplaçant par du vide va "casser" la table et empêcher de chercher efficacement. Pour cette raison, on ne remplace pas la clé par du vide mais par un marqueur spécifique appelé "tombstone".

