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 Python - primitives

```
1  d[key] = value
2  # Set d[key] to value.
3
4  del d[key]
5  # Remove d[key] from d. Raises a KeyError if key is not in the map.
6
7  key in d
8  # Return True if d has a key key, else False.
9
10 items()
11 # Return a new view of the dictionarys items ((key, value) pairs).
12
13 keys()
14 # Return a new view of the dictionarys keys.
15
16 values()
17 # Return a new view of the dictionarys values.
18
19 ...
```


En Python - implantation de la SD

```
1  typedef struct {
2      PyObject_HEAD
3      Py_ssize_t ma_used;
4      PyDictKeysObject *ma_keys;
5      PyObject **ma_values;
6  } PyDictObject;
7
8  typedef struct {
9      /* Cached hash code of me_key. */
10     Py_hash_t me_hash;
11     PyObject *me_key;
12     PyObject *me_value; /* This field is only meaningful for combined tables */
13 } PyDictKeyEntry;
```


En Python - le hash code

- il existe une fonction `hash` disponible sur les objets Python (qui retourne un entier, pas une adresse dans une table)

```
1  >>> hash('abracadabra')
2  >>> 967464175714002499
```

- pour les entiers : la valeur de l'entier
- pout les chaînes de caractères :

> *By default, the `__hash__()` values of str, bytes and datetime objects are "salted" with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.*

II Faisons le point

d'un point de vue pratique :

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

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

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

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

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

Remarques 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

Sondage quadratique

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

Prenons $a = 1$ et $b = 2$. $h'(k,i) = h(k) + i + 2i^2 \mod 5$.

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

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

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 - double hachage

On utilise deux fonctions de hachage primaire.

Double hachage

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

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	$h_1(k)$	$h_2(k)$	$h'(k,i)$ $i=0$	1	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 peu 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

Meilleur des cas

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

Et en moyenne ?

Les types et variables utilisés

```
1  # association <cle,valeur>
2  { "cle" : string, "val" : int }
3
4  # taille de la table (M)
5  capacite = ...
6
7  # la table
8  table = ...
9  # utilise pour verifier si une alveole est deja remplie
10 occupe = ...
```

CU : l'insertion d'une clé déjà présente aura pour effet de remplacer l'ancienne valeur associée par la nouvelle

Code de la fonction d'insertion

```
Require: k, v key and value to insert
Require: t the hash table
  function ADD(t, k, v)
    n ← 0
    a ← h'(k, v, n) (*  hashCode  *)
    (*  look for an empty slot if any  *)
    while t[a] is not empty && t[a].key ≠ k && n ≤ M do (* equals *)
      n ← n + 1
      a ← h'(k, v, n)
    end while
    if n ≤ M then
      if t[a] is not empty then (* insert the new entry *)
        t[a].key ← k
        t[a].value ← v
      else(* change the entry *)
        t[a].value ← v
      end if
    else(*  table is full  *)
      raise Error
    end if
```

Code de la fonction de recherche

```
Require: t a hashtable
Require: k the key
  function LOOKUP(t, k)
      n ← 0
      a ← h'(k, v, n)
      (*  look for an empty slot if any  *)
      while  t[a] is not empty && t[a].key ≠ k do
          n ← n + 1
          a ← h'(k, v, n)
      end while
      if  t[a] is not empty  then
          return t[a].value
      else
          raise NotFound
      end if
  end function
```


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 au moment de l'ajout.

- il y a un accès à l'alvéole d'adresse $h'(k,0)$
  avec une probabilité 1
- l'accès à $h'(k,1)$ se fait ssi $h'(k,0)$ était occupée
  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 occupées
  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 :

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

τ	nb moyen accès	τ	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
- le taux de remplissage avant l'insertion de la i-ème clé est $\frac{i-1}{M}$
- on peut majorer le nombre moyen d'accès a_i :

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

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 :

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

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

En Python - sondage et hash code

The first half of collision resolution is to visit table indices via this recurrence :

```
1    j = ((5*j) + 1) mod 2**i
```

```
0 -> 1 -> 6 -> 7 -> 4 -> 5 -> 2 -> 3 -> 0 [and here it's repeating]
```

The other half of the strategy is to get the other bits of the hash code into play. This is done by initializing a (unsigned) vrbl "perturb" to the full hash code, and changing the recurrence to :

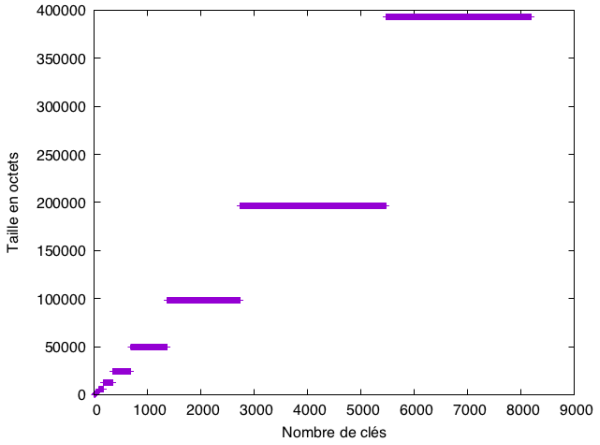
```
1    j = (5*j) + 1 + perturb;
2    perturb >>= PERTURB_SHIFT;
3    use j % 2**i as the next table index;
```

```
3 -> 11 -> 19 -> 29 -> 5 -> 6 -> 16 -> 31 -> 28 -> 13 -> 2
```

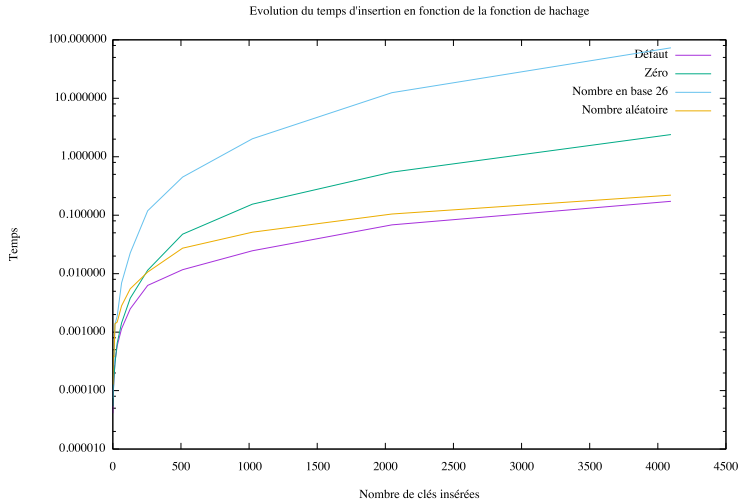
Note that because perturb is unsigned, if the recurrence is executed often enough perturb eventually becomes and remains 0. At that point (very rarely reached) the recurrence is on (just) 5*j+1 again, and that's certain to find an empty slot eventually (since it generates every int in range(2**i), and we make sure there's always at least one empty slot).

En Python - taille du dictionnaire

En Python, la taille de la table n'est pas fixe. Un dictionnaire vide possède 8 alvéoles.



En Python - influence de la fonction de hachage



Gestion des suppressions

- tout ce qui a été énoncé auparavant est vrai si il n'y a pas eu de suppression d'élements dans la table
- comment gérer les suppressions ?
  1. marquer l'alvéole d'un état spécial 'supprimé'
     - complexité : c'est le même coût qu'une recherche fructueuse.
  2. décaler les alvéoles 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 !

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

Gestion des suppressions

- tout ce qui a été énoncé auparavant est vrai si il n'y a pas eu de suppression d'élements dans la table
- comment gérer les suppressions ?
  1. marquer l'alvéole d'un état spécial 'supprimé'
     - complexité : c'est le même coût qu'une recherche fructueuse.
  2. décaler les alvéoles 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 !

Faisons le point

sur la résolution des collisions par adressage ouvert :

- la taille de la table est fixe
- on utilise des stratégies de hachage permettant de balayer les alvéoles de manières uniforme
  - inutile de réinventer la roue : il existe déjà de très nombreuses stratégies performantes
  - ces stratégies sont déjà implantées dans les bibliothèques fournissant ces structures de données
- la performance de l'insertion et de la recherche est excellente
- la suppression est délicate : on réservera l'utilisation de ces tables à l'indexation d'informations pérennes

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