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	
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 - 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 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

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  type couple = { cle : String; val : int }
3
4  (* taille de la table (M) *)
5  let capacite = ...
6
7  (* la table *)
8  let table = couple array
9  (* utilise pour verifier si une alveole est deja remplie *)
10 let occupe = bool array
```

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

```
1  let inserer k v =
2    let tentative = ref 1
3    and adresse = ref 0
4    in
5      adresse := hprime k tentative;
6      (* recherche d'une alveole libre *)
7      while occupe.(!adresse) && (table.(!adresse).cle <> k) &&
8            (tentative < capacite) do begin
9        tentative := !tentative + 1;
10       adresse := hprime k tentative;
11     end (* while *);
12     (* 3 cas : table pleine, cle presente, cle non presente *)
13     if (tentative < CAPACITE) then begin
14       if not occupe.(!adresse) then begin
15         occupe.(!adresse) <- true;
16         table.(!adresse).cle := k;
17         table.(!adresse).val := v;
18       end else begin
19         table.(!adresse).val := v;
20       end (* if *);
21     end else begin
22       failwith "inserer: Table pleine";
23     end (* if *);
```

Code de la fonction de recherche

```
1  let rechercher k =
2    let tentative = ref 1
3    and adresse = ref 0
4    in
5       tentative := 0;
6       adresse := hprime k tentative;
7       (* recherche de l'alveole contenant la cle *)
8       while occupe.(!adresse) && (table.(!adresse).cle <> k) do begin
9          tentative := !tentative + 1;
10         adresse := hprime k tentative;
11      end (* while *);
12      if occupe.(!adresse) then
13         table.(!adresse).val
14      else
15         failwith "rechercher: Cle non trouvee";
```


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 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

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 !

Résolution des collisions par chaînage

- la table a la capacité de grandir
- si une alvéole est déjà occupée, on ajoute « dans la même alvéole » le nouveau couple `<clé,valeur>`

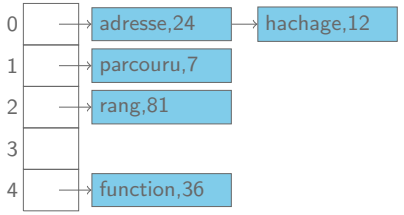
II Faisons le point

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`

Schéma de principe



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

```
1  (* la table est un tableau de listes *)
2  let table = couple list array
```

Recherche d'une clé

```
let rechercher k =
  (* acces direct a la bonne alveole *)
  let adresse = h k
  and p = ref [];
  in
    (* recherche de la cle dans la liste *)
    p := table.(!adresse);
    while (!p <> []) && ((hd !p).cle <> k) do begin
       p := tl !p;
    end (* while *);
    if !p <> [] then
       (hd p).val
    else
       failwith "recherche: Cle non trouvee";
```


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

```
let inserer k v =
  let adresse = h k
  and c = { cle = k; valeur = v }
  in
    try
       rechercher k
    with _ ->
       (* si la cle n'est pas trouvee *)
       table.(adresse) := c :: table.(adresse);
```

Note : dans cette version, si la clé est déjà présente on ne fait rien. Suivant la bibliothèque utilisée, l'insertion d'une valeur déjà présente a parfois pour effet de remplacer la valeur associée à la clé.

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)

Faisons le point

sur l'adressage par chaînage

- permet de stocker autant d'éléments qu'on veut
- la performance est liée à la fois à la fonction de hachage mais aussi à la taille de la table
- l'insertion est très efficace
- la suppression est réalisable facilement

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