Les tables de hachage
(Hash tables)

Problématique

Avec une structure ordonnée (tableau trié ou ABR) :

- diminue le coup d'une recherche
- augmente le coup d'une insertion

Si l'emplacement en mémoire était prédictible :

- diminuerait fortement le coup d'une recherche
- diminuerait fortement le coup d'une insertion

Il serait donc intéressant d'avoir une relation direct

clé → @dresse mémoire

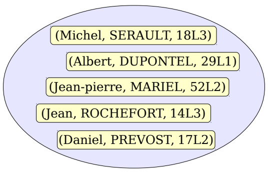
Notion de «clé»

Les données manipulées sont enregistrées dans des n-uplets.

Ex : (Nom, Prénom, N° d'étudiant)

C'est la notion de **schéma** en Base de Donnée.

Chaque enregistrement est appelé : **entité**

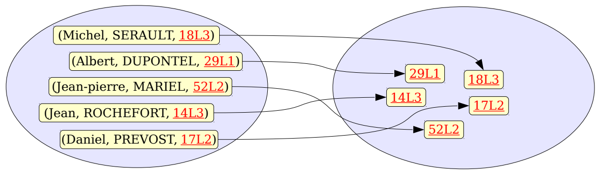


Notion de «clé»

Les entités sont distinguables les une des autres.

Parmi les champs du schéma on peut trouver un sous ensemble caractérisant chaque entité. Cet identifiant est appelé : **clé**

Ex : (Nom, Prénom, N° d'étudiant)

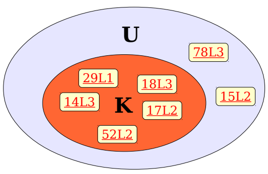


Notion de «clé»

On appelle **univers des clés** (**U**) :
l'ensemble des valeurs possibles des clés.

En général, toutes les clés ne sont pas utilisées.
On notera **K** l'ensemble des clés utilisées.

ATTENTION :
Les clés ne sont pas le résultat d'une fonction de hachage.
Elles résultent d'un choix du programmeur.



Fonctions de hachage

On appelle **Fonction de hachage** :

Une fonction qui détermine la place d'une entité uniquement d'après sa clé

H **:** clés → [0..m-1] avec [0..m-1] = Zone mémoire

C'est la composition de deux fonctions :

1) **Fonction de codage**

clés → entiers

2) **Fonction d'adressage**

entiers → [0..m-1]

Fonctions de codage

Code ASCII :

On considère la clé comme une chaîne de caractères.
Chaque caractère est remplacé par son code ASCII

Ex : H(hello) = 110 105 114 114 117
= 110105114114117

Codage des bits par un entier :

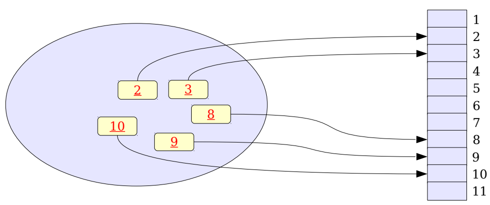
On considère la représentation machine de la clé.
On ré-interprète les bits comme le code d'un entier.

Ex : H(hello) = 110110 110000 110010 110010 110101
= 918760629

Fonctions d'adressage direct

La fonction d'adressage la plus simple c'est : l'identité ou **adressage direct**

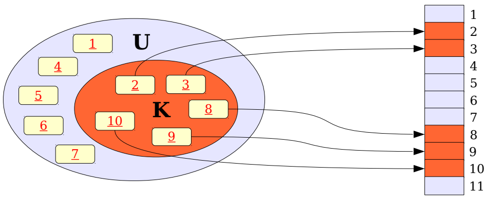
L'@dresse de l'entité correspond au code de sa clé.



Limite de l'adressage direct

Dans la pratique le nombre de clés réellement conservées est très inférieur à l'ensemble des clés. D'où un gachi de mémoire.

#(K) << #(U)

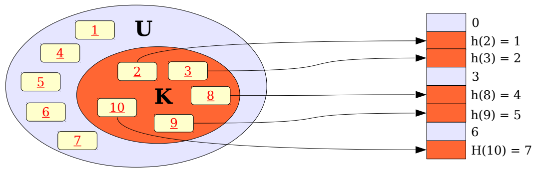


Adressage en compression

Pour économiser l'espace mémoire, la fonction d'adressage va compresser les clés

On dit que la clé *k* a été **hachée** dans l'**alvéole** *h(k)*

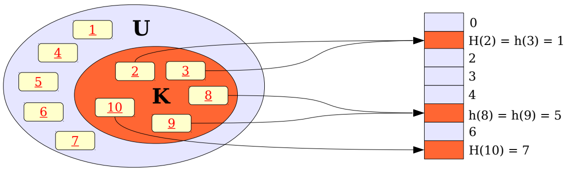
La fonction de hachage doit être déterministe : une clé est toujours hachée dans la même alvéole.



Collisions

Le problème c'est que 2 clés peuvent être hachées dans la même alvéole.

Si $h(k_A)=h(k_B)$ on dit que k_A et k_B sont en **collision**.



Fonctions uniformes injectives

La fonction de hachage doit bien repartir les clés. Elle doit être **uniforme simple** c'est-à-dire :

h: K → [0..m-1] $\forall k, \forall i, P(h(k)=i) = 1/m$

La probabilité P_{inj} que : «une fonction uniforme simple soit injective»

$P_{inj} = (1/m^n)\ m\ (m-1)\ (m-2) \dots (m-n+1) = (m!/(m-n)!)\ (1/m^n)$

Si m=365 et n=23 alors $P_{inj} < 0,5$

C'est le **problème des anniversaires** :
«Si l'on réuni 23 personnes, il y a plus d'une chance sur 2 qu'au moins 2 personnes aient leur anniversaire à la même date »

Adressage par extraction

Dans une fonction de hachage avec adressage **par extraction**, on extrait des bits de la clé pour obtenir la valeur de hachage.

Ex : avec les bits 3, 10, 18 et 23 et un codage ASCII
*h(hello)= 110110 1**1**0000 110**0**10 11**0**010 110**1**01*
= 1001 = 9

Inconvénient, la valeur de hachage ne dépend pas de l'intégralité de la clé :
h(hello) = h(Say hello)= 9

Une bonne fonction de hachage doit faire intervenir tous les bits de la clé.

Adressage par division

Hachage avec adressage **par division** :

h: K → [0..m-1]
h(k) = k modulo m

Ex : avec m=5 h(13)=3

Cette technique a une bonne répartition mais multiplie les collisions.

On la combinera avec d'autres méthodes.

Pour minimiser les collisions on doit choisir : m premier et éloigné des puissances de 2

Adressage par multiplication

Hachage avec adressage **par multiplication** :

h: K → [0..m-1]
h(k) = $\lfloor$ m (k A modulo 1)$\rfloor$ avec A=cst

Le choix de la constante de A dépend du type de clé

Knuth a montré que la valeur A = (√5 – 1)/2 avait de grande chance de bien marcher.

Ex : avec m = 10000 et A = (√5 – 1)/2 = 0,61803....
h(123456) = $\lfloor$ 10000 . ((123456 .0,61803) modulo 1) $\rfloor$
= $\lfloor$ 10000 . (76300,0041151 modulo 1) $\rfloor$
= $\lfloor$ 10000 . 0,0041151 $\rfloor$
= $\lfloor$ 41,151 $\rfloor$
= 41

Adressage par MAD

Hachage avec adressage par **M**ultiplication, **A**ddition et **D**ivision (**MAD**) :

h: K → [0..m-1]
h(k) = (ak + b) modulo m avec a>0 et b>0

Ex : avec m = 7 , a = 8 et b = 5
h(13) = (8.7 + 5) modulo 7
= (104 + 5)
= 109 modulo 7
= 4

Attention :
La constante *a* ne doit pas être un multiple de *m*.
Sinon toute valeur de hachage serait égale à *b*.

Adressage par compression

Hachage avec adressage **par compression** :

On découpe en morceaux de n bits le code de la clé

On compresse les morceaux avec une opération : addition, et, ou, ou exclusif et polynomiale

Compression par addition

Avec l'**addition** se pose le problème de la retenue :

Ex : h(47557) = 00001 + 01110 + 01110 + 00101
= (10)00010

Mais aussi de l'uniformité :
« Probabilité de tirer un 7 avec deux dés à 6 faces »

Compression par ET/OU

Avec le **ou** le résultat est :
toujours plus grand que les nombres d'origine.
Il y aura collision en fin de tableau.

Ex : h(47557) = 00001 ou 01110 ou 01110 ou 00101
= 01111
= 15

Avec le **et** le résultat est :
toujours plus petit que les nombres d'origine.
Il y aura collision en début de tableau.

Ex : h(47557) = 00001 et 01110 et 01110 et 00101
= 00000
= 0

Compression par XOR

Avec le **xor** on évite les problèmes du ET et du OU.
Les valeurs sont uniformément reparties.

Ex : h(47557) = 00001 xor 01110 xor 01110 xor 00101
= 00100
= 4

Mais on rencontre des collisions sur les permutations

Ex : h(47278) = 00001 xor 01110 xor 00101 xor 01110
= 00100
= 4
= h(47557)

Briser les sous chaînes de bits

Une bonne fonction de hachage doit :
briser sous chaînes des bits de la clé.

On garde le **xor** mais on ajoute des **décalages** :
la sous chaîne n sera décalée vers la droite de n bits

Ex : 47557 = 00001 01110 01110 00101
h(47557) = 10000 xor 10011 xor 11001 xor 01010
= 10000
= 16

Ex : 47557 = 00001 01110 00101 01110
h(47278) = 10000 xor 10011 xor 01010 xor 11100
= 10101
= 21

Compression polynomiale

Dans la **compression polynomiale** :

1) les sous chaînes ont des longueurs de 8, 16 ou 32 bits

2) chaque sous chaîne représente un coefficient

$$a_0\ a_1\ a_2\ a_3 \dots a_{n-1}$$

3) on calcule le polynôme :

$$P(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \dots + a_{n-1} z^{n-1}$$

très bonne méthode pour le type string.

Il a été démontré que le choix de $z=33$ donne au plus 6 collisions sur un ensemble de 50000 mots anglais

Résolution des collisions

On ne peut pas éviter les collisions.

Pour résoudre les collisions il y a deux méthodes :

Hachage fermé :
La fonction de hachage ne retourne qu'une valeur par clé (fermé sur cette valeur) et on range la clé à partir de cette valeur.

Hachage ouvert :
La fonction de hachage retourne un ensemble de valeurs. Cet ensemble représente les rangements possibles pour cette clé. Cet ensemble sera parcouru pour l'insertion comme pour la recherche.

Exemple général

Pour toutes les méthodes de résolution de collision :

On utilisera la fonction d'adressage : h(k) = k mod 11

On insérera les clés (→ code) suivantes :

- Rubis → 16
- Jade → 18
- Topaze → 6
- Opale → 29
- Perle → 50
- Saphir → 13
- Agate → 27
- Grenat → 31
- Onyx → 28

Résolution par chaînage

Principe de la **résolution par chaînage** :

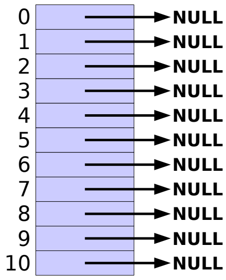
Les éléments en collisions sont chaînés entre eux à l'extérieur du tableau de hachage. La table de hachage contient le début de ces listes chaînées.

Algorithme de recherche de la *clé(k)*:

Calcul de la valeur de hachage : h(k) → @
SI** tab[@] contient la clé k **ALORS
succès
SINON
explore la liste jusqu'à trouver la clé k
FIN SI

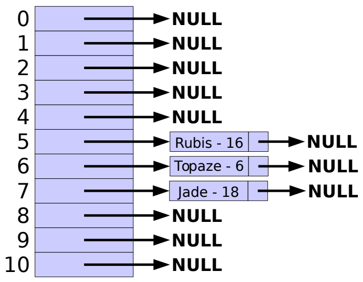
Résolution par chaînage

A l'état initial, toutes les listes chaînées sont vides.



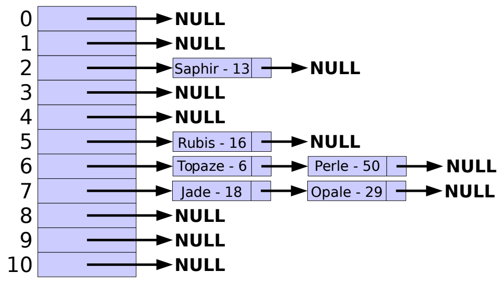
Résolution par chaînage

On insère : (Rubis → 16), (Jade → 18) et (Topaze → 6)



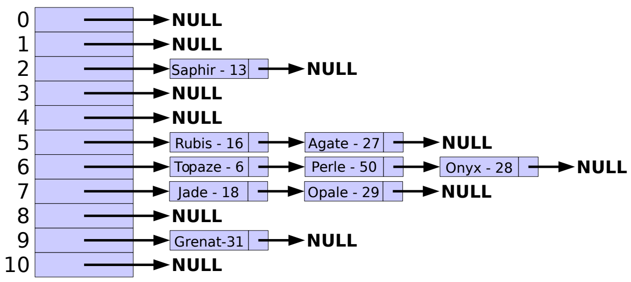
Résolution par chaînage

On insère : (Opale → 29), (Perle → 50) et (Saphir → 13)



Résolution par chaînage

On insère : (Agate → 27), (Grenat → 31) et (Onyx → 28)



Résolution par chaînage

Les avantages :

- Recherche facile
- Suppression facile
- Seul l'ordre des clés en collisions compte.

Les inconvénients :

- Une allocation mémoire à chaque enregistrement.
- Taille : un pointeur supplémentaire pour chaque clé

Complexité (sur l'exemple)

Clé	Code	H(k)	Liste ch.
Rubis	16	5	1
Jade	18	7	1
Topaze	6	6	1
Opale	29	7	2
Perle	50	6	2
Saphir	13	2	1
Agate	27	5	2
Grenat	31	9	1
Onyx	28	6	3
TOTAL :			14
MOYENNE :			1,55

Résolution par coalescence

Principe de la **résolution par coalescence** :

Lorsqu'il y a collision avec une autre clé on cherche une autre place disponible. On note cette place comme le NEXT de la première clé. Au fil des collisions, ces **liens explicites** forment une liste chaînée dans la table.

Algorithme de recherche de la *clé(k)*:

Calcul de la valeur de hachage : h(k) → @
TANQUE** tab[@] ne contient pas la clé k **FAIRE
SI** next[@] = -1 **ALORS
echec
SINON
@ ← next[@]
FIN SI
FIN TANQUE
succès

Résolution par coalescence

A l'état initial, toutes les cases sont disponibles.

0		-1
1		-1
2		-1
3		-1
4		-1
5		-1
6		-1
7		-1
8		-1
9		-1
10		-1

Disponible :
{10,9,8,7,6,5,4,3,2,1,0}

Résolution par coalescence

On insère : (Rubis ➞ 16), (Jade ➞ 18) et (Topaze ➞ 6)

0		-1
1		-1
2		-1
3		-1
4		-1
5	Rubis - 16	-1
6	Topaze - 6	-1
7	Jade - 18	-1
8		-1
9		-1
10		-1

Disponible :
{10,9,8,4,3,2,1,0}

Résolution par coalescence

On insère : (Opale ➞ 29), (Perle ➞ 50) et (Saphir ➞ 13)

0		-1
1		-1
2	Saphir - 13	-1
3		-1
4		-1
5	Rubis - 16	-1
6	Topaze - 6	9
7	Jade - 18	10
8		-1
9	Perle - 50	-1
10	Opale - 29	-1

Disponible :
{8,4,3,1,0}

Résolution par coalescence

On insère : (Agate ➞ 27), (Grenat ➞ 31) et (Onyx ➞ 28)

0		-1
1		-1
2	Saphir - 13	-1
3	Onyx - 28	-1
4	Grenat-31	3
5	Rubis - 16	8
6	Topaze - 6	9
7	Jade - 18	10
8	Agate - 27	-1
9	Perle - 50	4
10	Opale - 29	-1

Disponible :
{1,0}

Résolution par coalescence

Les avantages :

Pas d'allocation mémoire pour un enregistrement.

Les inconvénients :

Le nombre de clé est limité à la taille de la table.

Taille : un lien supplémentaire pour chaque clé.

La table dépend de l'ordre de toutes les clés.

Suppression très compliquée (voir T.D.).

Complexité (sur l'exemple)

Clé	Code	H(k)	Liste ch.	Coalesc.
Rubis	16	5	1	1
Jade	18	7	1	1
Topaze	6	6	1	1
Opale	29	7	2	2
Perle	50	6	2	2
Saphir	13	2	1	1
Agate	27	5	2	2
Grenat	31	9	1	2
Onyx	28	6	3	4
TOTAL :			14	16
MOYENNE :			1,55	1,78

Rés. par sondage linéaire

Principe **par sondage linéaire** :

Lorsqu'il y a collision avec une autre clé on parcours circulairement la table pour chercher une place disponible. On parcours donc l'ensemble suivant :

{@ | @=(h(k) + i) mod m   avec i ∈ [0..m-1] }

Algorithme de recherche de la *clé(k)*:

Calcul de la valeur de hachage : h(k) ➞ @ ➞ $@_{initiale}$
TANQUE** tab[@] ne contient pas la clé k **FAIRE
@ ← (@ + 1) mod m
SI** @ = $@_{initiale}$ **ALORS
echec
FIN TANQUE
succès

Rés. par sondage linéaire

On insère : (Rubis ➞ 16), (Jade ➞ 18) et (Topaze ➞ 6)

0	
1	
2	
3	
4	
5	Rubis - 16
6	Topaze - 6
7	Jade - 18
8	
9	
10	

h(Rubis) = (16 mod 11 + i) mod 11
avec i=0 : h(Rubis)=5

h(Jade) = (18 mod 11 + i) mod 11
avec i=0 : h(Jade)=7

h(Topaze) = (6 mod 11 + i) mod 11
avec i=0 : h(Saphir)=6

Rés. par sondage linéaire

On insère : (Opale → 29), (Perle → 50) et (Saphir → 13)

0	
1	
2	Saphir - 13
3	
4	
5	Rubis - 16
6	Topaze - 6
7	Jade - 18
8	Opale - 29
9	Perle - 50
10	

h(Opale) = (29 mod 11 + i) mod 11
avec i=0 : h(Opale)=7
avec i=1 : h(Opale)=8

h(Perle) = (50 mod 11 + i) mod 11
avec i=0 : h(Perle)=6
avec i=1 : h(Perle)=7
avec i=2 : h(Perle)=8
avec i=3 : h(Perle)=9

h(Saphir) = (13 mod 11 + i) mod 11
avec i=0 : h(Saphir)=2

Rés. par sondage linéaire

On insère : (Agate → 27), (Grenat → 31) et (Onyx → 28)

0	Grenat-31
1	Onyx - 28
2	Saphir - 13
3	
4	
5	Rubis - 16
6	Topaze - 6
7	Jade - 18
8	Opale - 29
9	Perle - 50
10	Agate - 27

h(Agate) = (27 mod 11 + i) mod 11
avec i=0 : h(Agate)=5 | avec i=3 : h(Agate)=8
avec i=1 : h(Agate)=6 | avec i=4 : h(Agate)=9
avec i=2 : h(Agate)=7 | **avec i=5 : h(Agate)=10**

h(Grenat) = (31 mod 11 + i) mod 11
avec i=0 : h(Grenat)=9
avec i=1 : h(Grenat)=10
avec i=2 : h(Grenat)= 11 mod 11 = 0

h(Onyx) = (28 mod 11 + i) mod 11
avec i=0 : h(Onyx)=6 | avec i=4 : h(Onyx)=10
avec i=1 : h(Onyx)=7 | avec i=5 : h(Onyx)=0
avec i=2 : h(Onyx)=8 | **avec i=6 : h(Onyx)=1**
avec i=3 : h(Onyx)=9 |

Rés. par sondage linéaire

Les avantages :

- Pas d'allocation mémoire pour un enregistrement.
- Recherche fructueuse facile
- Suppression facile
- Taille : réduite au minimum.

Les inconvénients :

- Recherche infructueuse en *O(n)*
- Le nombre de clé est limité à la taille de la table.
- La table dépend de l'ordre de toutes les clés.
- Formation d'amas dans la table.

Complexité (sur l'exemple)

Clé	Code	H(k)	Liste ch.	Coalesc.	Sond. Lin.
Rubis	16	5	1	1	1
Jade	18	7	1	1	1
Topaze	6	6	1	1	1
Opale	29	7	2	2	2
Perle	50	6	2	2	4
Saphir	13	2	1	1	1
Agate	27	5	2	2	6
Grenat	31	9	1	2	3
Onyx	28	6	3	4	7
TOTAL :			14	16	26
MOYENNE :			1,55	1,78	2,89

Résolution par double-hachage

Principe **par double-hachage** :

Lorsqu'il y a collision avec une autre clé on parcours circulairement la table, avec un pas donné par une autre fonction de hachage, pour chercher une place disponible. On parcours donc l'ensemble suivant :

$$\{@ \mid @ = (h_1(k) + i \times h_2(k)) \bmod m \quad \text{avec } i \in [0..m-1]\}$$

Algorithme de recherche de la *clé(k)*:

Calcul de la valeur de hachage : $h_1(k) \rightarrow @$
TANQUE** tab[@] ne contient pas la clé k **FAIRE
  $@ \leftarrow (@ + h_2(k)) \bmod m$
  SI @ déjà parcouru ALORS
    echec
FIN TANQUE
succès

Résolution par double-hachage

A l'état initial, toutes les cases de la table sont vides.

0		-1
1		-1
2		-1
3		-1
4		-1
5		-1
6		-1
7		-1
8		-1
9		-1
10		-1

Pour l'exemple on considérera :

$$h(k) = (h_1(k) + i \times h_2(k)) \bmod 11$$

Avec :

$$h_1(k) = k \bmod 11$$

$$h_2(k) = k/11 + 1$$

Résolution par double-hachage

On insère : (Rubis → 16), (Jade → 18) et (Topaze → 6)

0	
1	
2	
3	
4	
5	Rubis - 16
6	Topaze - 6
7	Jade - 18
8	
9	
10	

h_1(Rubis) = 16 mod 11 = 5
h_2(Rubis) = 16/11 + 1= 2
h(Rubis) = (5 + i x 2) mod 11
avec i=0 : h(Rubis)=5

h_1(Jade) = 18 mod 11 = 7
h_2(Jade) = 18/11 + 1= 2
h(Jade) = (7 + i x 2) mod 11
avec i=0 : h(Jade)=7

h_1(Topaze) = 6 mod 11 = 6
h_2(Topaze) = 6/11 + 1= 1
h(Topaze) = (6 + i x 1) mod 11
avec i=0 : h(Topaze)=6

Résolution par double-hachage

On insère : (Opale → 29), (Perle → 50) et (Saphir → 13)

0	Perle - 50
1	
2	
3	Saphir - 13
4	
5	Rubis - 16
6	Topaze - 6
7	Jade - 18
8	
9	
10	Opale - 29

h_1(Opale) = 7 et h_2(Opale) = 29/11 + 1= 3
h(Opale) = (7 + i x 3) mod 11
avec i=0 : h(Opale)=7
avec i=1 : h(Opale)=10

h_1(Perle) = 6 et h_2(Perle) = 50/11 + 1= 5
h(Perle) = (6 + i x 5) mod 11
avec i=0 : h(Perle)=6
avec i=1 : h(Perle)= 11 mod 11 = 0

h_1(Saphir) = 2 et h_2(Saphir) = 13/11 + 1= 2
h(Saphir) = (6 + i x 3) mod 11
avec i=0 : h(Saphir)=2

Résolution par double-hachage

On insère : (Agate → 27), (Grenat → 31) et (Onyx → 28)

0	Perle - 50
1	Onyx - 28
2	
3	Saphir - 13
4	
5	Rubis - 16
6	Topaze - 6
7	Jade - 18
8	Agate - 27
9	Grenat - 31
10	Opale - 29

h_1(Agate) = 5 et h_2(Agate) = 27/11 + 1= 3
h(Agate) = (5 + i x 3) mod 11
avec i=0 : h(Agate)=5
avec i=1 : h(Agate)=8

h_1(Grenat) = 9 et h_2(Grenat) = 31/11 + 1= 3
h(Grenat) = (9 + i x 3) mod 11
avec i=0 : h(Grenat)=9

h_1(Onyx) = 6 et h_2(Onyx) = 28/11 + 1= 3
h(Onyx) = (6 + i x 3) mod 11
avec i=0 : h(Onyx)=6
avec i=1 : h(Onyx)=9
avec i=2 : h(Onyx)= 12 mod 11 = 1

Résolution par double-hachage

Pour s'assurer de bien parcourir toute la table :
$h_2(k)$ doit être premier avec *m* (le nombre de cases)

Par exemple prenons une table de *m=6* cases :

On garde la même fonction : h_1(k) = k mod 6 et h_2(k) = k/6 + 1

On insère : Louis VI *le gros*, Jean II *le bon*, Charles IV *le bel*

0	Louis VI *le gros*
1	
2	Jean II *le bon*
3	
4	Charles IV *le bel*
5	

Avec i=0, h(Louis VI *le gros*)=0 qui est libre

Avec i=0, h(Jean II *le bon*)=2 qui est libre

Avec i=0, h(Charles IV *le bel*)=4 qui est libre

Résolution par double-hachage

Pour s'assurer de bien parcourir toute la table :
$h_2(k)$ doit être premier avec *m* (le nombre de cases)

Maintenant on veut insérer : Louis X *le hutin*

h_1(Louis X *le hutin*) = 10 mod 6 = 4

h_2(Louis X *le hutin*) = 10/6 + 1 = 2

0	Louis VI *le gros*
1	
2	Jean II *le bon*
3	
4	Charles IV *le bel*
5	

Avec i=0, h(Louis X *le hutin*)= 4
Avec i=0, h(Louis X *le hutin*)=(4 + 2)mod6=0
Avec i=0, h(Louis X *le hutin*)=(4 + 4)mod6=2
Avec i=0, h(Louis X *le hutin*)=(4 + 6)mod6=4
On boucle sur les cases déjà occupées :-(

Résolution par double-hachage

Les avantages :
- Pas d'allocation mémoire pour un enregistrement.
- Recherche fructueuse facile
- Suppression facile
- Taille : réduite au minimum.
- Évite la formation d'amas dans la table.

Les inconvénients :
- Recherche infructueuse en *O(n)*
- Le nombre de clé est limité à la taille de la table.
- La table dépend de l'ordre de toutes les clés.

Complexité (sur l'exemple)

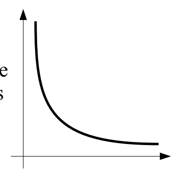
Clé	Code	H(k)	Liste ch.	Coalesc.	Sond. Lin.	Double h.
Rubis	16	5	1	1	1	1
Jade	18	7	1	1	1	1
Topaze	6	6	1	1	1	1
Opale	29	7	2	2	2	2
Perle	50	6	2	2	4	2
Saphir	13	2	1	1	1	1
Agate	27	5	2	2	6	2
Grenat	31	9	1	2	3	1
Onyx	28	6	3	4	7	3
TOTAL :			14	16	26	14
MOYENNE :			1,55	1,78	2,89	1,55

Conclusion

Les tables de hachage se résument à un compromis entre espace mémoire et collisions :

Avec une mémoire infini : on évite toutes collisions :
@ = Clé

Avec une mémoire minimum : toutes les clés sont en collision. Elles sont rangées dans une liste.



Conclusion

	Complexité moyenne		Complexité du pire cas	
	Recherche	Insertion / Retrait	Recherche	Insertion / Retrait
Sequentiel	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Dichotomie	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(log_2(n))$	$\Theta(n)$
ABR	$\Theta(log_2(n))$	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(n)$
ABR Equilibré	$\Theta(log_2(n))$	$\Theta(log_2(n))$	$\Theta(log_2(n))$	$\Theta(log_2(n))$
Hachage	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$