

LU2IN006

SDA table de hachage

“Structures de données”

Pierre-Henri Wuillemin

2022-2023

# 1. table de symboles

LU2IN006

**Question : Comment retrouver facilement les données associés à un symbole, une clé ?**

⇒ { on ne connaît pas le nombre d'éléments a priori,  
on doit accéder rapidement aux éléments.  
les clés n'ont pas à être **ordonnées**.

**Intérêts ?**

- Compilateurs [tables de variables],
- Application complexes [CAO, etc.],
- Algorithmes nécessitant des caches sophistiqués, etc.

## *Table de symboles*

structure abstraite pour effectuer rapidement :

- l'insertion
- la recherche
- la suppression

d'informations dynamiques, identifiées par une clé quelconque.

1. Table de  
symbole

2. Tables avec  
chainage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique

# Problématique des tables de hachage (1/2)

LU2IN006

1. Table de  
symbole

2. Tables avec  
chainage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique

Soit un langage (type ocaml) :

```
let aaa = 3 and aba = 4 and abb = 5 in
```

```
let abc = aaa + 2 and abd = aaa + 3 in
```

```
let acc = ...
```

- Représentation de la table des variables par une liste linéaire ordonnée ?

$aaa \rightarrow aba \rightarrow \dots \rightarrow acc.$

- Représentation de la table des variables par un table ordonné ?

aaa	aba	...	acc
-----	-----	-----	-----

$\Rightarrow$  trop de comparaisons



L'ordre est une hypothèse trop forte !

# Problématique des tables de hachage (2/2)

LU2IN006

1. Table de  
symbole

2. Tables avec  
chaînage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

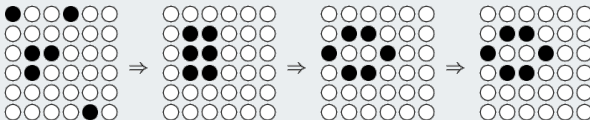
5. Hachage  
universel

6. Hachage  
cryptographique

## Game of life

- Une cellule a huit voisines.
- L'état d'une cellule à  $t + 1$  dépend de son état et de celui de ses voisines à l'instant  $t$  :

État à $t$	État à $t + 1$
Cellule morte, 3 voisines vivantes	Vivante
Cellule vivante, 2 ou 3 voisines vivantes	Vivante
Sinon	Morte



# Golly : hashlife

LU2IN006

1. Table de  
symbole

2. Tables avec  
chaînage

Résolution des  
collisions

Fonctions de  
hachage

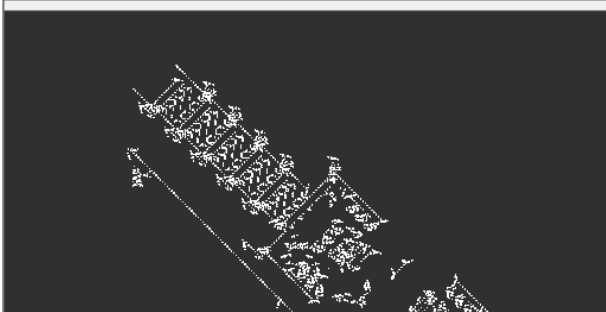
3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique

Generation = 6,366,548,773,467,669,985,195,496,000  
Population = 36,558



Utilisation d'un cache élaboré pour éliminer le plus de calculs redondants possibles : portion de plan à  $t + 1$  indexée par une autre portion de plan à  $t$ .

# Adressage direct (1/3)

LU2IN006

1. Table de  
symbole

2. Tables avec  
chaînage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique

## *Principe des tables de hachage*

- on associe à chaque information à stocker un entier
- on recherche les informations via leur entier associé

(0,aaa)	(1,aba)		(3,abb)		(5,abc)	(6,abd)	(7,acc)
---------	---------	--	---------	--	---------	---------	---------

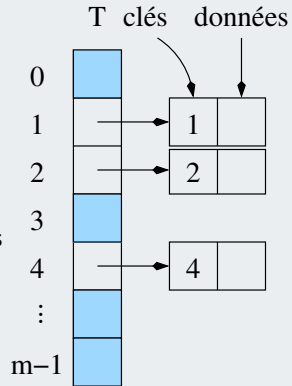
# Adressage direct (2/3)

LU2IN006

$U$  : l'univers des clés

## Tables à adressage direct

- $U = \{0, 1, \dots, M - 1\}$
- Toutes les clés ont des entiers associés différents
- un tableau  $T$  contient les données



# Adressage direct (3/3)

LU2IN006

1. Table de  
symbole

2. Tables avec  
chaînage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique

## Problèmes des tables à adressage direct :

- ❶  $U \neq \{0, 1, \dots, M-1\}$  ou, pire,  $U$  n'est pas un ensemble d'entiers positifs
- ❷  $|U|$  est très grand  $\implies T$  trop grand pour être stocké en mémoire
- ❸ plusieurs données ont la même clé

## Solutions :

- ❶ créer une fonction hash :  $U \mapsto \{0, 1, \dots, m-1\}$
- ❷ si tout l'univers des clés est utilisé ( $m=M$ ), pas de solution sinon :  
soit  $K$  l'ensemble des clés réellement utilisées ( $|K| \ll |U|$ ).  
avec  $|K|$  du même ordre (même si plus grand) que  $m$   
 $\implies$  table de hachage
- ❸ *collisions* dans une table de hachage.

Solutions dans les transparents suivants



# Tables de hachage (1/2)

LU2IN006

1. Table de  
symbole

2. Tables avec  
chaînage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique

## Définition

Table (comme pour l'adressage direct) mais au lieu de stocker l'élément de clé  $k$  à l'indice  $k$  de la table, on le stocke à l'indice  $h(k)$ , où  $h$  est une *fonction de hachage* :  $U \mapsto \{0, 1, \dots, m-1\}$ .

$|U|$  est souvent plus grand que  $m$ .

$\implies$  répond aux points 1 et 2

## Exemples

- Annuaire téléphonique :  
clé = nom des personnes, donnée = numéro de téléphone
- Nom des variables dans un compilateur :  
clé = nom de la variable, donnée = adresse en mémoire
- Gestion d'un ensemble de voitures :  
clé = numéro d'immatriculation, donnée = données sur la voiture

# Tables de hachage (2/2)

LU2IN006

## 1. Table de symbole

## 2. Tables avec chaînage

Résolution des collisions

Fonctions de hachage

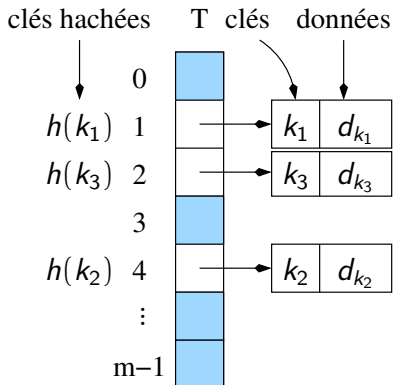
## 3. Tables avec adressage ouvert

## 4. Application : filtre de Bloom

## 5. Hachage universel

## 6. Hachage cryptographique

donnée	clé	clé hachée
$d_{k_1}$	$k_1$	$h(k_1) = 1$
$d_{k_2}$	$k_2$	$h(k_2) = 4$
$d_{k_3}$	$k_3$	$h(k_3) = 2$



# Problèmes des tables de hachage

LU2IN006

## 1. Table de symbole

## 2. Tables avec chaînage

Résolution des collisions

Fonctions de hachage

## 3. Tables avec adressage ouvert

## 4. Application : filtre de Bloom

## 5. Hachage universel

## 6. Hachage cryptographique

### ● Comment choisir la fonction $h$ ?

- le calcul de  $h(k)$  doit être rapide
- $h$  doit éviter au maximum les collisions

### ● Que faire quand il y a collision ? ( $h(k_1) = h(k_2)$ pour $k_1 \neq k_2$ )

- résolution par chaînage
- résolution par adressage ouvert

# problème des collisions

LU2IN006

1. Table de  
symbole

2. Tables avec  
chaînage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique

soit  $h(k) = 0 \forall k \in U \implies$  collision pour tout  $k_1 \neq k_2$

soit  $h(k) = k \forall k \in U \implies$  moins de collisions

$\implies$  le choix de  $h$  permet d'éviter des collisions

*Propriété* : Il est impossible d'éviter totalement les collisions

table de hachage : stocke des données ayant  $|K|$  clés possibles dans une table de longueur  $m \ll |K| \implies$  il n'y a pas bijection entre  $K$  et  $\{0, 1, \dots, m-1\}$

# Résolution par chaînage

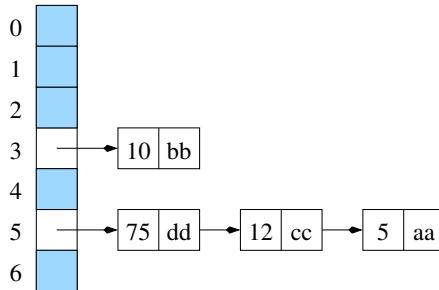
LU2IN006

## Principe

Chaque élément de la table est une liste chaînée et tous les éléments ayant la même clé hachée sont dans la même liste chaînée.

### Exemple :

- $h(k) = k \% 7$
- «aa»  $\Rightarrow k = 5 \Rightarrow h(k) = 5$
- «bb»  $\Rightarrow k = 10 \Rightarrow h(k) = 3$
- «cc»  $\Rightarrow k = 12 \Rightarrow h(k) = 5$
- «dd»  $\Rightarrow k = 75 \Rightarrow h(k) = 5$



$\Rightarrow$  nécessité de stocker les clés dans la liste chaînée

# Analyse de la résolution par chaînage (1/4)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

*Hypothèse* : le calcul de  $h(k)$  s'effectue en  $O(1)$

## *Analyse de l'insertion d'un élément*

- calcul de  $h(k) = O(1)$
- insertion dans la liste chaînée appropriée :  $O(1)$

Résultat : insertion en  $O(1)$

## *Analyse de la suppression d'un élément*

si la liste est doublement chaînée : suppression d'un élément en  $O(1)$

# Analyse de la résolution par chaînage (2/4)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

Supposons que la table  $t$  soit de taille  $m$  et contienne  $n$  éléments

## *Analyse de la recherche d'un élément*

Dans le pire des cas, tous les éléments de la table appartiennent à la même liste chaînée  $\Rightarrow$  recherche en  $\Theta(n)$ .

$\Rightarrow$  Les tables de hachage sont moins performantes que les listes chaînées dans le pire des cas

## *facteur de charge*

Le facteur de charge d'une table de hachage :  $\alpha = n/m$ .

*Hypothèse (hachage uniforme simple)* : chaque liste de la table a la même chance de recevoir un élément tiré au hasard

# Analyse de la résolution par chaînage (3/4)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *Recherche d'un élément n'appartenant pas à la table*

Sous l'hypothèse de hachage uniforme simple, la recherche d'un élément n'appartenant pas à la table est en  $\Theta(1 + \alpha)$  en moyenne.

**Démonstration :** Sous l'hypothèse de hachage uniforme simple,  $h(k)$  a une chance égale de correspondre à n'importe quelle liste de la table.

Donc le temps moyen pour la recherche d'une clé  $k$  est le temps moyen pour arriver à la fin d'une des listes chaînées.

La longueur moyenne d'une liste est  $\alpha$ . Donc la recherche s'effectue en  $1$  (calcul de  $h(k)$ )  $+ \alpha$ .



# Analyse de la résolution par chaînage (4/4)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## Recherche d'un élément appartenant à la table

Sous l'hypothèse de hachage uniforme simple, la recherche d'un élément appartenant à la table est en  $\Theta(1 + \alpha)$  en moyenne.

**Démonstration :** Hypothèse de hachage uniforme simple  $\implies$  la longueur moyenne des listes après insertion de  $i - 1$  éléments est  $(i - 1)/m$ . Supposons que les éléments sont insérés à la fin des listes. Le nombre moyen d'éléments examinés pendant la recherche du  $i$ ème élément inséré est  $1 +$  le nombre d'éléments de la liste avant d'insérer cet élément  $= 1 + (i - 1)/m$   
 $\implies$  Celui de la recherche d'un elt quelconque est en moyenne :

$$\frac{1}{n} \sum_{i=1}^n \left( 1 + \frac{i-1}{m} \right) = 1 + \frac{\alpha}{2} - \frac{1}{2m}.$$

Donc la recherche est en  $\Theta(2 + \alpha/2 - 1/2m) = \Theta(1 + \alpha)$ .

# Conclusions

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *Conclusions des analyses*

Si le nombre de cases du tableau est **au moins** proportionnel au nombre d'éléments à stocker,  $\alpha = n/m = O(m)/m = O(1)$ .

- l'ajout d'éléments est en  $O(1)$
- la suppression est en  $O(1)$  si les listes sont doublement chaînées
- la recherche d'éléments est en  $O(1)$

# Choix des fonctions de hachage (1/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

**Problème :** qu'est-ce qu'une bonne fonction de hachage ?

une fonction :

- qui se calcule rapidement (en  $O(1)$ )
- qui minimise les collisions autant que possible

Minimisation des collisions = hachage uniforme simple

$$\sum_{k:h(k)=i} P(k) = \frac{1}{m} \quad \forall i \in \{0, \dots, m-1\}$$

$P$  connue  $\implies$  on peut trouver  $h$  qui minimise les collisions

En pratique  $P$  est inconnue  $\implies$  on utilise des heuristiques

# Choix des fonctions de hachage (2/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *Principe des fonctions de hachage usuelles*

- ❶ transformer la clé  $k$  en un entier via une fonction  $f : U \mapsto \mathbb{N}$
- ❷ transformer cet entier en un entier entre 0 et  $m - 1$  via une fonction  $g : \mathbb{N} \mapsto \{0, \dots, m - 1\}$

Autrement dit,  $h(k) = g \circ f(k)$ .

## *Exemple : la fonction $f : \text{string} \rightarrow \text{int}$ d'ocaml*

```
#define Beta 19

unsigned long hash_accu = 0;
unsigned int i = strlen(k);
for (char *p = k; i > 0; i--, p++)
    hash_accu = hash_accu * Beta + *p;
```

# Hachage par division

LU2IN006

1. Table de symbole
2. Tables avec chaînage  
Résolution des collisions  
**Fonctions de hachage**
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## Définition

$$g(x) = x \% m$$

## Choix de $m$

- éviter les collisions  $\implies$  utiliser dans  $g$  tous les bits de  $x$   
 $\implies$  éviter les puissances de 2
- «bon choix» : nombre premier pas trop proche d'une puissance de 2

# Hachage par multiplication

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## Définition

$g(x) = \lfloor m(xA - \lfloor xA \rfloor) \rfloor$ , où  $A \in ]0, 1[$ .

Avantage par rapport au hachage par division : choix de  $m$  non critique

## Choix de $A$

Knuth propose d'utiliser le nombre d'or  $(-1) : \frac{\sqrt{5}-1}{2}$

**Propriété :** pour tout  $A$  irrationnel,  $g(x), g(x+1), \dots, g(x+k)$  sont éloignés les uns des autres et  $g(x+k+1)$  appartient au plus grand des segments  $[g(x+i), g(x+i+1)]$ .

# adressage ouvert (1/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

*Constat* : la résolution par chaînage peut être coûteuse à cause des pointeurs des listes chaînées

Peut-on éviter les pointeurs ?

Oui : en stockant directement les éléments dans la table et non dans des listes chaînées

*Avantage* : utilise moins de place que la résolution par chaînage  $\implies$  on peut utiliser des tables plus grandes

*Problème* : gestion des collisions

# adressage ouvert (2/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *Principe*

Pour effectuer une insertion, on scanne la table jusqu'à ce que l'on trouve une case vide dans laquelle insérer l'élément. La séquence de cases scannées dépend de la clé de l'élément à insérer.

## *fonction de hachage*

$$h : U \times \{0, \dots, m-1\} \mapsto \{0, \dots, m-1\}$$

Séquence de scans :  $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$

$\implies \langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$  doit être une permutation de  $\{0, \dots, m-1\}$



# Exemple d'insertion

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

$$h(k, i) = \left( \left\lfloor 8 \left( \frac{\pi k}{4} - \left\lfloor \frac{\pi k}{4} \right\rfloor \right) \right\rfloor + \frac{i}{2} + \frac{i^2}{2} \right) \% 8$$

0	8	dd
1		
2	3	aa
3	12	cc
4		
5	11	bb
6		
7		

$x = \text{"dd"}, k = 8$

$$h(k, 3) = 0$$

# Autres opérations

LU2IN006

- 1. Table de symbole
- 2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
- 3. Tables avec adressage ouvert
- 4. Application : filtre de Bloom
- 5. Hachage universel
- 6. Hachage cryptographique

- Recherche d'un élément,
- Suppression d'un élément.



La suppression d'élément brise la chaîne d'éléments. Il est nécessaire d'avoir une représentation de 'élément supprimé'.

Malheureusement les analyses de complexité ne dépendent plus seulement du facteur de charge  $\alpha$

⇒ en principe, on utilise plutôt la résolution par chaînage

# Analyse de l'adressage ouvert (1/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

*Hypothèse (hachage uniforme)* : chaque clé a une chance égale d'avoir n'importe laquelle des  $m!$  permutations de  $\{0, \dots, m-1\}$  comme séquence de scans

En pratique, cette hypothèse n'est jamais vérifiée : on n'en a que des approximations

## *Analyse d'une recherche infructueuse*

Soit une table de hachage dans laquelle aucune suppression n'est permise. Soit  $\alpha = n/m < 1$  le facteur de charge de la table. Alors l'espérance du nombre de scans lors de la recherche infructueuse d'un élément est au plus de  $1/(1 - \alpha)$ .

$\alpha$	50 %	80 %	90 %	99 %
scans	2	5	10	100

# Analyse de l'adressage ouvert (2/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *Analyse d'une recherche couronnée de succès*

Si  $\alpha < 1$  alors l'espérance du nombre de scans pour trouver l'élément recherché est au plus de :

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha} + \frac{1}{\alpha}.$$

$\alpha$	50 %	80 %	90 %	99 %
scans	3,396	3,26	3,67	5,66

## *Analyse d'une insertion*

Si  $\alpha < 1$  alors l'espérance du nombre de scans nécessaires à l'insertion d'un élément est au plus de  $1/(1 - \alpha)$ .

# fonctions de hachage (1/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *Probing linéaire*

$h(k, i) = (h'(k) + i) \% m$ , où  $h' : U \mapsto \{0, \dots, m-1\}$ .

**Problème** : formation de clusters de cases remplies  
 $\Rightarrow$  les scans peuvent être longs

## *Probing quadratique*

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \% m$ , où  $h' : U \mapsto \{0, \dots, m-1\}$ .

**Problème** : 2 clés  $k_1, k_2$  telles que  $h'(k_1) = h'(k_2)$  auront la même séquence de scans

# fonctions de hachage (2/2)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *double hachage*

$h(k, i) = (h_1(k) + ih_2(k)) \% m$ , où  $h_1$  et  $h_2$  sont des fonctions de hachage de  $U \mapsto \{0, \dots, m-1\}$ .

***Propriété*** : le double hachage approche l'hypothèse de hachage uniforme.

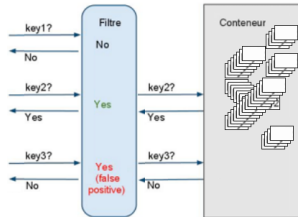
Le double hachage est plus performant que les 2 autres fonctions de hachage

# 4. Filtre de Bloom : tester si un élément n'est pas dans un ensemble.

LU2IN006

Un filtre de Bloom permet de savoir :

- avec certitude que l'élément est absent de l'ensemble  
**pas de faux négatif**
- avec une certaine probabilité que l'élément peut être présent  
**faux positifs possible**



**but** : éviter des accès inutiles à des ressources *lentes*.

- BigTable (base de données distribuées – Google)
- inspection de paquets en profondeur.
- Chrome évite des milliers d'appels réseau en stockant une blacklist.
- Correcteurs orthographiques.

# Principe du filtre de Bloom

LU2IN006

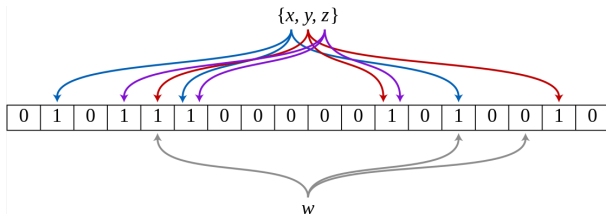
1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## Filtre de Bloom pour un sous-ensemble de $U$

- $T[0 \dots m-1]$  tableau de booléens initialisé à 0  
(très compact si le langage le permet)
- $\{h_i, 0 \leq i < k\}$   $k$  fonctions de hachage de  $U \mapsto \{0, 1, \dots, m-1\}$

Opérations :

- Insertion( $e \in U$ ) :  $\forall i < k, T[h_i(e)] \leftarrow 1$
- EstPossiblementPrésent( $e \in U$ )  $\iff \bigwedge_{i < k} (T[H_i(e)])$



vressort

un filtre de Bloom est une structure légère.

$\log_2(m)$  octets en mémoire.



# filtre de Bloom : probabilité de faux-positifs

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

*Soit un filtre de Bloom de taille  $m$ , avec  $k$  fonctions de hachage, contenant déjà les éléments  $\{e_j\}_{j < n}$ .*

En supposant les  $\{h_i\}_{i < k}$  des fonctions de hachage **uniforme**

- Proba que  $h_i(e)$  mette le bit  $b$  à 1 :
- Proba que  $h_i(e)$  laisse  $b$  à 0 :
- Proba que  $\{h_i(e)\}_{i < k}$  laisse  $b$  à 0 :
- Proba que  $\{h_i(e_j)\}_{i < k, j < n}$  laisse  $b$  à 0 :
- Proba que  $\{h_i(e_j)\}_{i < k, j < n}$  mette  $b$  à 1 :

## Proportion de faux-positifs

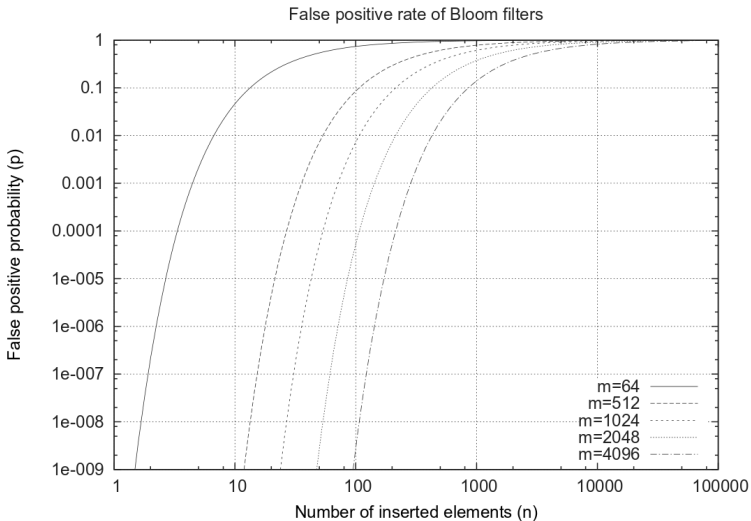
Proba. que les  $k$  bits représentant un élément  $x$  soit à 1 à cause des  $n$  clés déjà dans le filtre de Bloom :

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

# Filtre de Bloom : taux de faux-positifs

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique



hressort

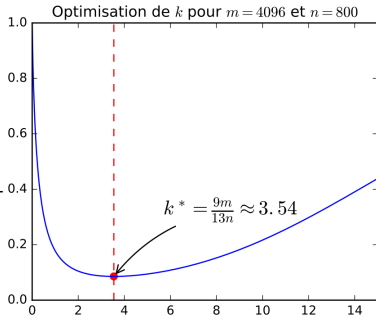
# Filtre de Bloom : Nombre de fonction de hachage

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

$$FP(n, m, k) = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

peut être optimisé pour  $k$  (en annulant la dérivée).



Optimisation de  $k$

$$k^* = \ln 2 \cdot \frac{m}{n} \approx \frac{9m}{13n}$$

$$FP(n, m, k^*) \approx 0.6185^{\frac{m}{n}}$$

Pour garder un taux constant de faux positif, il faut faire grandir linéairement le filtre de Bloom.

# Bonus : Hachage universel (1/3)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

Retour sur le hachage : *En fonction de la taille et de la fonction, le comportement de la SDA peut être très mauvais.*

**But :** éviter qu'un utilisateur mal intentionné ne choisisse que des clés ayant la même valeur hachée

**Principe :** choisir la fonction  $g$  au hasard indépendamment des clés lors de chaque exécution du programme

⇒ La vitesse change à chaque exécution  
mais en moyenne, quelque soit l'ordre d'insertion, accès en  $O(1)$ .

# Hachage universel (2/3)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## *Ensemble universel*

Soit  $\mathcal{H}$  un ensemble fini de fonctions de hachage de  $U \mapsto \{1, \dots, m-1\}$ .

$\mathcal{H}$  est universel si  $\forall k_1, k_2 \in U$  tels que  $k_1 \neq k_2$ ,  
 $|\{h \in \mathcal{H} : h(k_1) = h(k_2)\}| = |\mathcal{H}|/m$

$\implies$  pour une fonction  $h \in \mathcal{H}$  donnée, la probabilité de collision entre 2 clés est de  $1/m$

## *Théorème*

Si  $h$  est choisie dans un ensemble universel et est utilisée pour hacher  $n$  clés dans une table de taille  $m$ , avec  $n \leq m$ , alors l'espérance du nombre de collisions avec une clé  $k$  donnée est inférieure strictement à 1.

# Hachage universel (3/3)

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## Choix d'un ensemble universel

On suppose que  $m$  est un nombre premier

On définit un mot comme tout ensemble de nombres entiers de 0 à  $p$ , avec  $p < m$

On décompose une clé  $k \in U$  en une séquence de  $r + 1$  mots :

$$k = \langle k_0, k_1, \dots, k_r \rangle$$

On note  $a = \langle a_0, a_1, \dots, a_r \rangle$  une séquence où les  $a_i$  sont choisis au hasard dans  $\{0, \dots, m - 1\}$

On définit  $\mathcal{H} = \bigcup_a \{h_a\}$  avec

$$h_a(k) = \sum_{i=0}^r a_i k_i \% m.$$

Alors  $\mathcal{H}$  est un ensemble universel.

# Cryptography Hash Functions : fonctions sans tables

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

$$f = g \circ f : U \rightarrow \{0, \dots, m\}$$

Une fonction de hachage sans table associée n'a pas à **limiter fortement  $m$** .

Fonction de hachage, fonction de compression, *digest*

Une fonction de hachage avec  $m$  assez grand se comporte comme une signature (une empreinte) de l'objet hashé.

- calcul de  $h$  rapide (bien plus rapide qu'un chiffrement).
- la taille de  $h(data)$  est bien plus petite que  $data$  (compression).
- quelque soit la taille de  $data$ , la taille de  $h(data)$  est constante.

La taille de  $h(data)$  se compte en bit :  **$k$ -bits hash function**.  $k \in \{160, \dots, 512\}$ .

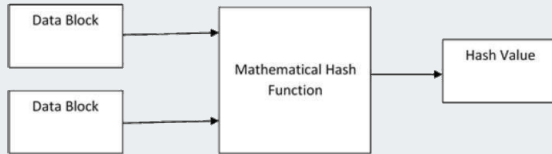
$$m = 2^k$$

# Architecture classique d'une fonction de hashage sur des données de taille quelconque

LU2IN006

## Fonction de hashage à taille fixe

La fonction de hashage opère sur 2 objets de taille  $2^k$  et retourne une valeur de taille  $2^k$ .



## Hashage en *avalanche* : améliore le 'brassage' des clés





# Cryptography Hash Functions : propriétés

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

Pour pouvoir vraiment parler de fonction de hachage *cryptographique*, il faut quelques propriétés supplémentaires :

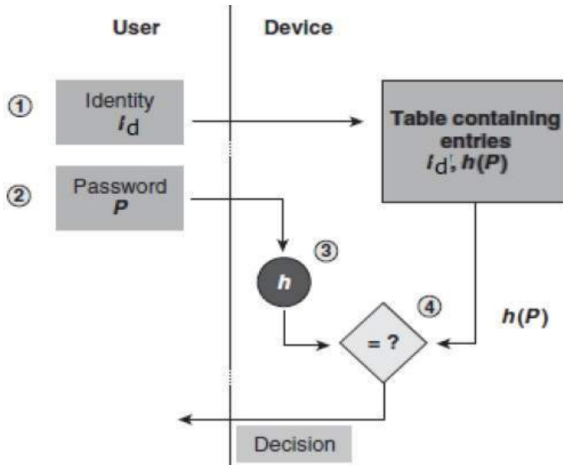
## hashage cryptographique

- **Brassage des clés**  
 $u \sim u' \Rightarrow P(h(u) \sim h(u')) \rightarrow 0.$
- **Résistance à la collision**  
Il est difficile de trouver  $u, u'$  tel que  $h(u) = h(u')$ .
- **Pre-image Resistance**  
Il est difficile de retrouver  $u$  à partir de  $h(u)$ .
- **Second Pre-image Resistance**  
Même avec  $u$  et  $h(u)$ , il est difficile de trouver  $u'$  tel que  $h(u) = h(u')$ .

# Application : garder des mots de passe dans une base de données

LU2IN006

On ne garde que les valeurs hashées des mots de passe : *Pre-image Resistance importante !*



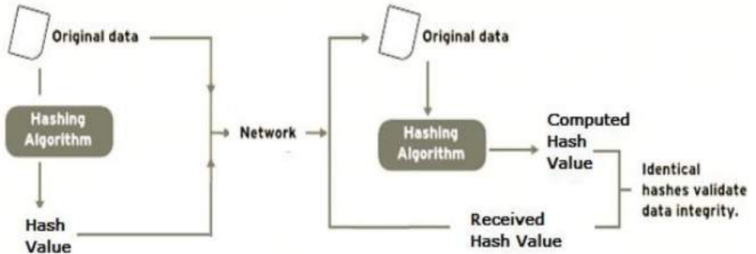
1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

# Application : *Data integrity*

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

La valeur hashée comme preuve d'intégrité (et d'authenticité) d'un fichier : *Brassage des clés importante !*



# Quelques fonctions *Digest* classiques

LU2IN006

1. Table de symbole
2. Tables avec chaînage  
Résolution des collisions  
Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

## MD5

- $k = 128\text{bits}$ .
- Très utilisé pour *Data integrity* : *MD5 checksum*.
- 2004 : *Collision attack* trouvée :  
Avec  $u$  donné,  $u'$  tel que  $h(u) = h(u')$  en 1H  $\Rightarrow$  **ne plus utiliser !**

## SHA

- SHA0 puis SHA1 (1995), utilisé par SSL (Secure Socket Layer).  
Aujourd'hui des doutes sur SHA-1  $\Rightarrow$  SHA2
- SHA2 :  $k \in 224, 256, 384, 512\text{bits}$ . Aucune attaque connue. (NSA, 2001)
- SHA3 en 2012 (SHA2 est le même algorithme que SHA1)

RIPEMD, Blake2, Bcrypt, Whirlpool (AES), etc.

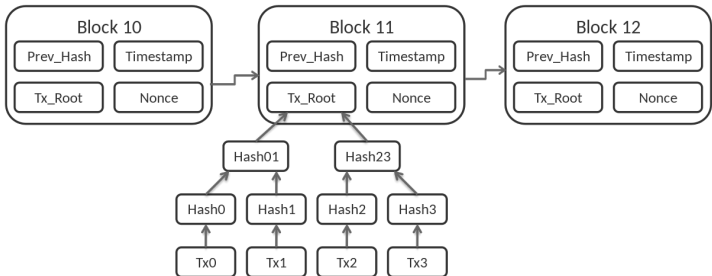
$k = 512\text{bits}$  semble être la bonne valeur (aujourd'hui).

# Application : blockchain

LU2IN006

La *blockchain* est une **structure de données distribuée** et est une liste linéaire chaînée de blocs de contenus qui utilise un *digest* pour

- Chaîner les blocs,
- Préserver l'intégrité de la chaîne,
- Indexer le contenu du block,
- **Proof of work.**



Bitcoin utilise SHA2-256.

# Proof of work

LU2IN006

1. Table de symbole
2. Tables avec chaînage
  - Résolution des collisions
  - Fonctions de hachage
3. Tables avec adressage ouvert
4. Application : filtre de Bloom
5. Hachage universel
6. Hachage cryptographique

Le problème à résoudre dans une structure de données décentralisée est comment rajouter d'une manière non ambiguë un nouveau block.

## Proof of Work (PoW) – Finney, 2004

La preuve de travail (PoW) décrit un système qui exige un effort non négligeable mais réalisable afin de décourager les utilisations frivoles ou malveillantes de la puissance informatique.

- mécanisme de consensus décentralisé qui exige que les membres d'un réseau déploient des efforts pour résoudre une énigme mathématique arbitraire.
- Grâce à la PoW, l'ajout de block peut être traité en *peer-to-peer* de manière sécurisée, sans qu'il soit nécessaire de faire appel à un tiers de confiance.

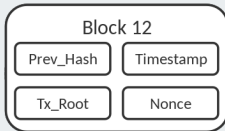


Le PoW nécessite d'énormes quantités d'énergie : beaucoup de *miners* tente de résoudre une même énigme sans trop d'enjeu à part de la résoudre en premier.

# PoW : l'énigme à résoudre (pour bitcoin)

LU2IN006

## PoW : bitcoin



Trouver Nonce tel que

$$h(block_{12}) = \underbrace{0 \dots 0}_{q \text{ fois}} \text{xxxxxxxxxx}$$

- Nonce : *Number used once*,
- La seule solution : *brute force*,
- La difficulté  $q$  est réglée en fonction du nombre de *miners* : environ une énigme résolue toutes les 10 minutes.

A transaction is requested

1

A block that represent the transaction is created

2

The block is sent to every node in the network

3

Nodes validate the transaction

4

Nodes receive a reward for the proof of work

5

The block is added to the existing blockchain

6

The transaction is complete

7

# Application : git

LU2IN006

Git identifie tout objet (fichier, tree, commit, tag, etc.) par **SHA1**.

```
1 > echo -n "Bonjour_LU2IN006!" | openssl sha1
2 (stdin)= 6445aa3608f570af5cce3d0c4f6e0c9c5d49e611
3 > echo -n "Bonjour_2i006!" | openssl sha1
4 (stdin)= d18022d087ad5a91fa7ddd8434b2c71479b6a7f6
```

tree	
.	9c435a86e664be00db0d973e981425e4a3ef3f8d

tree	
assets	7cf2a17f3345635d59e063cffdd23573b6e4a75
app.js	29bfcf9fa5824331081b31f0c307806c6f6b6f06



tree	
logo.png	aalb2fb696a831c89c53f787e03d863691d2b671
app.css	4c511f16ef2644854d04cabe9f9e9c82be0eb04f



```
.git (contents left out)
├── assets
│   ├── logo.png
│   └── app.css
└── app.js
```

1. Table de  
symbole

2. Tables avec  
chainage

Résolution des  
collisions

Fonctions de  
hachage

3. Tables avec  
adressage ouvert

4. Application :  
filtre de Bloom

5. Hachage  
universel

6. Hachage  
cryptographique