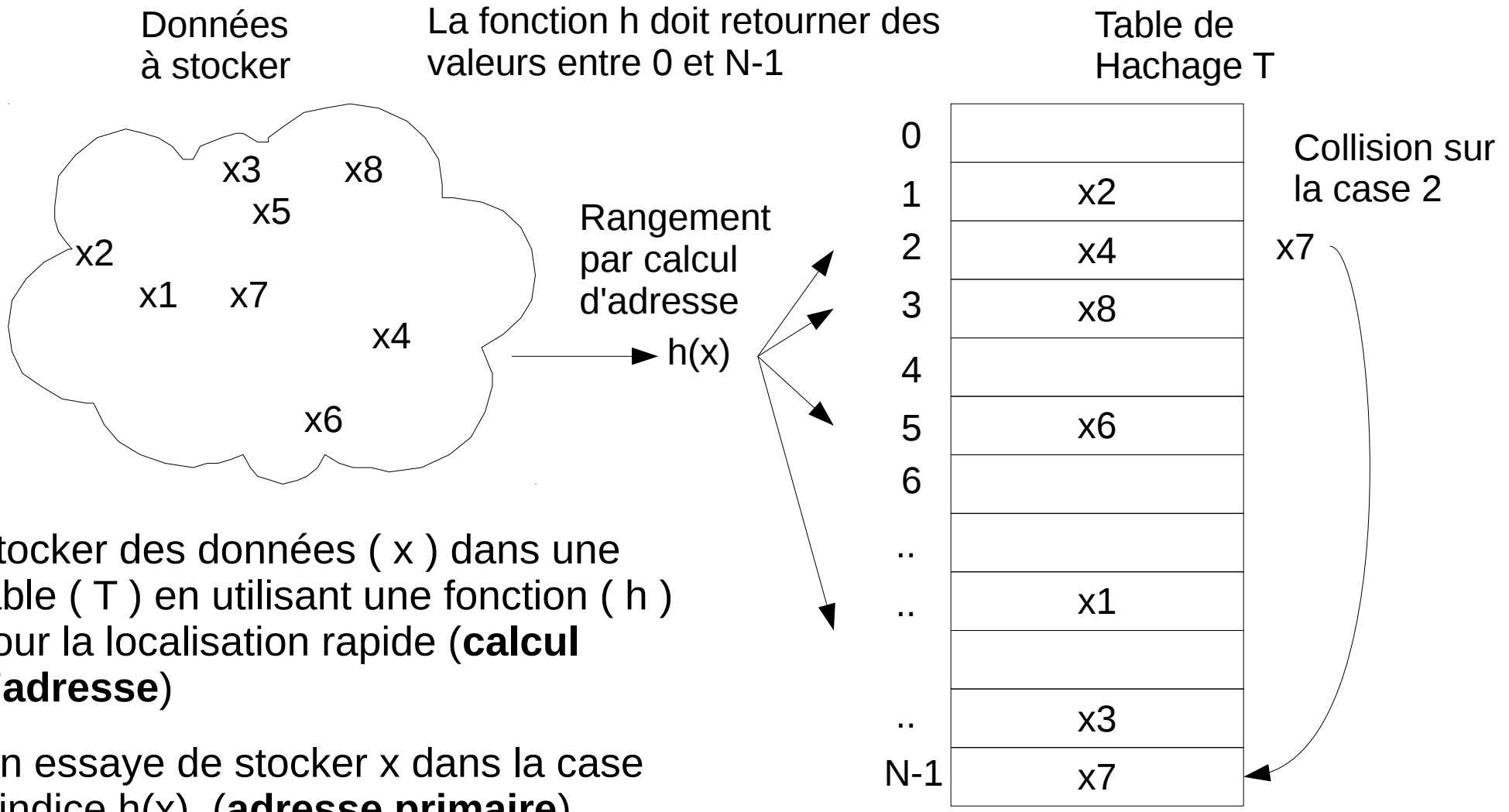
Hachage

Accès aux données d'une table (ou fichier) avec un temps constant
utilisation d'une fonction pour le calcul d'adresses

PLAN

- Définition (Rappel)
- Fonctions de Hachage (Rappel)
- Méthodes de résolution de collisions (Rappel)
- Estimation des débordements (Rappel)
- Fichier avec hachage statique
- Fichier avec hachage dynamique



- Stocker des données (x) dans une table (T) en utilisant une fonction (h) pour la localisation rapide (**calcul d'adresse**)
- On essaye de stocker x dans la case d'indice h(x) (**adresse primaire**)
- Si la case est déjà occupée (**collision**), on insère x à un autre emplacement (**adr secondaire**) déterminé par un algorithme donné (**Méthode de résolution de collisions**)

h(x4) = h(x7) = 2
x4 et x7 sont des **synonymes**
l'adr primaire de x4 et x7 est 2
x7 est inséré en **débordement**
l'adr secondaire de x7 est N-1

Quelques fonctions de hachage

- Quelques fonctions de hachage usuelles
  - La fonction de division: h(x) = x MOD N
    - Retourne le reste de la division par N (la taille de la table)
  - La fonction du « middle square »
    - Retourne les chiffres du milieu de x^2
  - La fonction du « transformation radix »
    - Change la base de numérotation de x puis applique le modulo
- Les bonnes propriétés
  - Uniforme, aléatoire, ... ==> minimise les collisions

Méthodes de résolution de collisions

- Lors de l'insertion de x, si l'adresse primaire h(x) est déjà utilisée par une autre donnée, la méthode de résolution de collision permet de trouver un autre emplacement (libre) pour x
- Quelques méthodes
  - Essai Linéaire
  - Double Hachage
  - Chaînage séparé
  - Chaînage interne

Essai Linéaire - Principe

Si la case h(x) est déjà occupée, essayer les cases qui la précèdent : h(x)-1, h(x)-2, h(x)-3,... jusqu'à trouver une case vide
=> Lors d'une recherche : **la rencontre d'une case vide indique que la donnée n'existe pas**

La **séquence** précédente est **circulaire**, c-a-d quand on arrive à 0, la prochaine case à tester est N-1

Il doit toujours rester au moins une case vide dans la table (on sacrifie une position)
=> **la séquence de test est finie**

Essai linéaire - Exemple

L'insertion des données suivantes, donne la table ci-après. Le bit 'vide' à 1, indique une case est vide.

h(a) = 5
h(b) = 1
h(c) = 3
h(d) = 3 => **collision** et insertion dans la case 2
h(e) = 0
h(f) = 2 => **collision** et insertion dans la case 10
h(g) = 8

La recherche de x (tel que h(x) = 2) s'arrête avec un échec dans la case vide d'adresse 9
=> la séquence de test est : 2,1,0,10,9

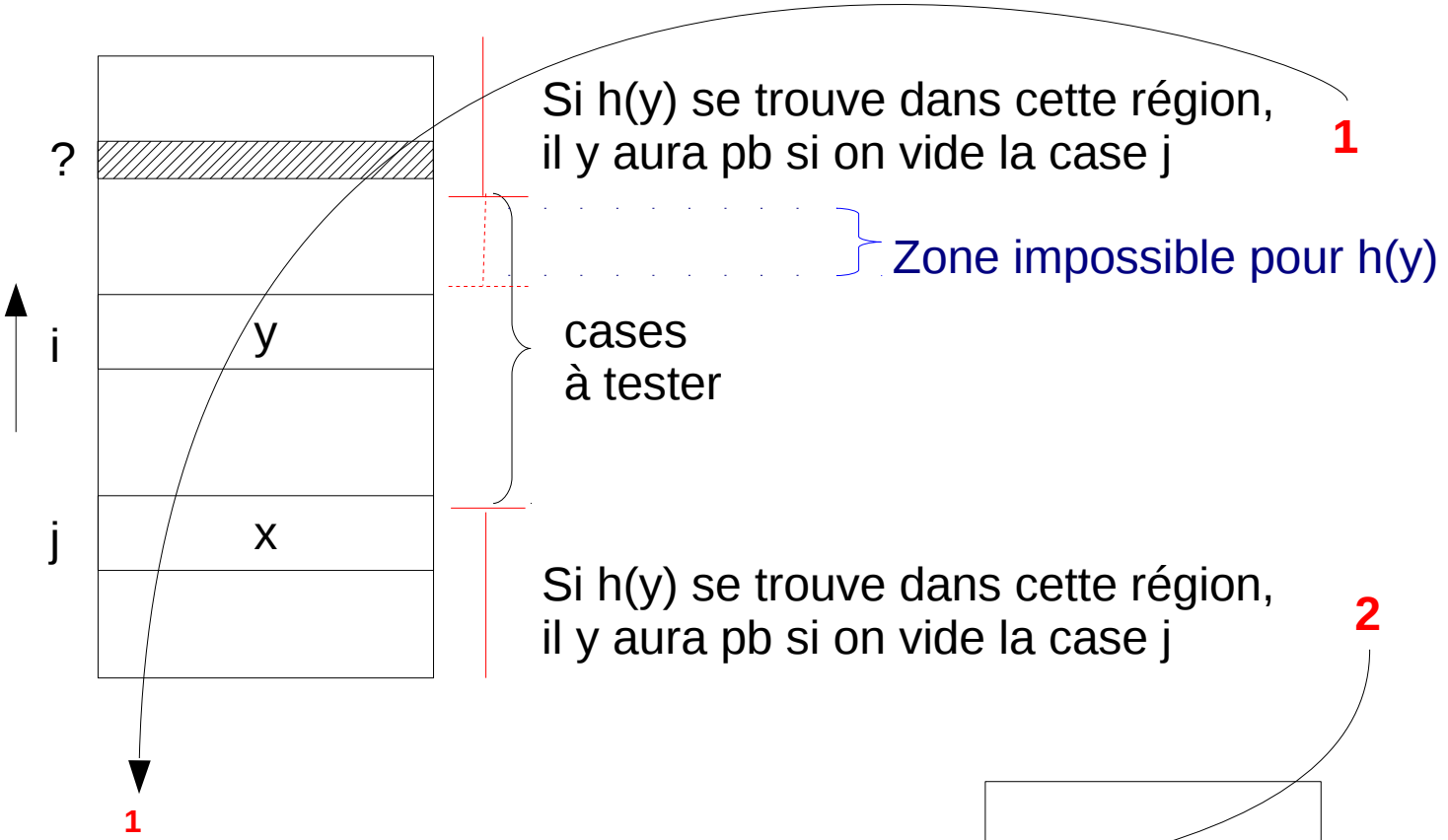
	Donnée	vide
0	e	0
1	b	0
2	d	0
3	c	0
4		1
5	a	0
6		1
7		1
8	g	0
9		1
10	f	0

Si on devait insérer x, la donnée serait affectée à la case 9 (si ce n'est pas la dernière case vide).

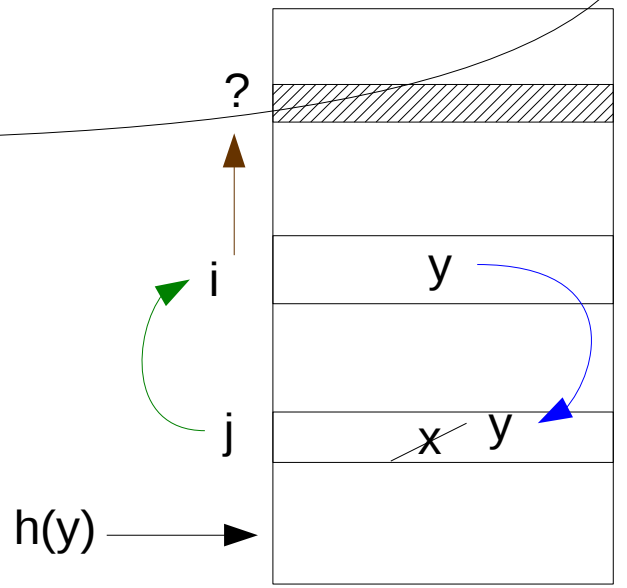
Essai linéaire - Suppression

- La **suppression** d'un élément x, **génère une case vide** => suppression physique
- Cette nouvelle case vide risque de rendre d'autres **données inaccessibles**. Dans l'exemple précédent, si on supprime b en vidant la case 1, on perd du même coup la donnée f (car elle n'est plus accessible) **==> faire des tests avant de vider une case**
- Le principe de suppression d'une donnée x est donc :
  - Rechercher l'adr j de x
  - Tester toutes les cases (i=j-1, j-2, ...) au dessus de j (circulairement) jusqu'à trouver une case vide et vérifier que les données qu'elles contiennent ne vont pas être perdues quand la case j sera vidée
  - Si c'est le cas, on vide la case j et on s'arrête
  - Sinon, dès qu'on trouve une donnée « qui pose problème » on la déplace dans la case j et on tente de vider son emplacement (en testant les cases au dessus qui n'ont pas encore été testées). C'est le même principe qu'on vient d'appliquer pour la case j.

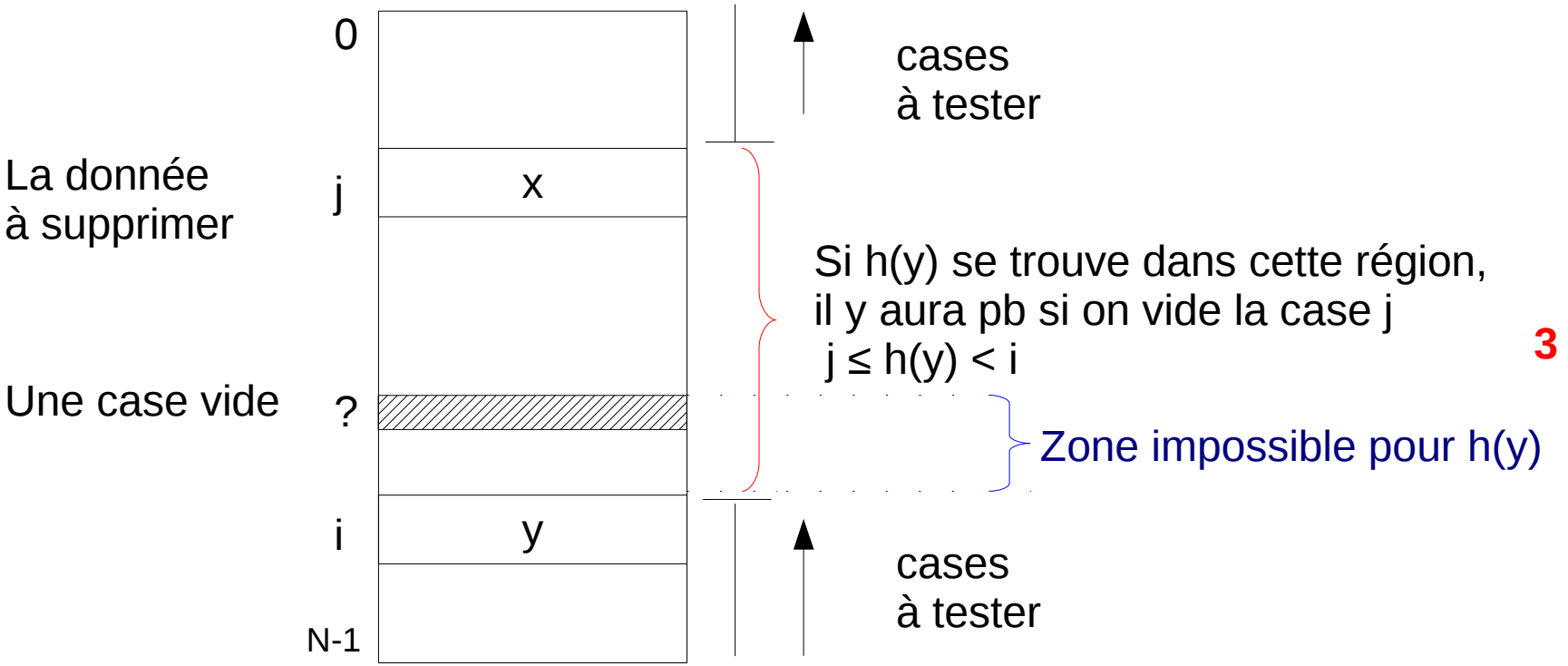
Quand est-ce qu'une donnée y se trouvant à l'adresse i pose problème si on vide la case j ?



y pose pb si (**h(y) < i** ou bien **h(y) ≥ j**)
dans ce cas :
- on déplace y dans la case j
- on modifie j pour qu'il pointe i
  (la nouvelle case à vider)
- on continue les tests sur les cases qui restent (i-1, i-2, ... jusqu'à la case vide)



Il y a un autre cas (3ème) qui pose aussi problème (lorsque la case vide se trouve en bas)



Donc en résumé, pour savoir si une donnée y se trouvant dans une case i, deviendra inaccessible quand on videra la case j, il faut vérifier si l'une des 3 conditions suivantes est vraie :

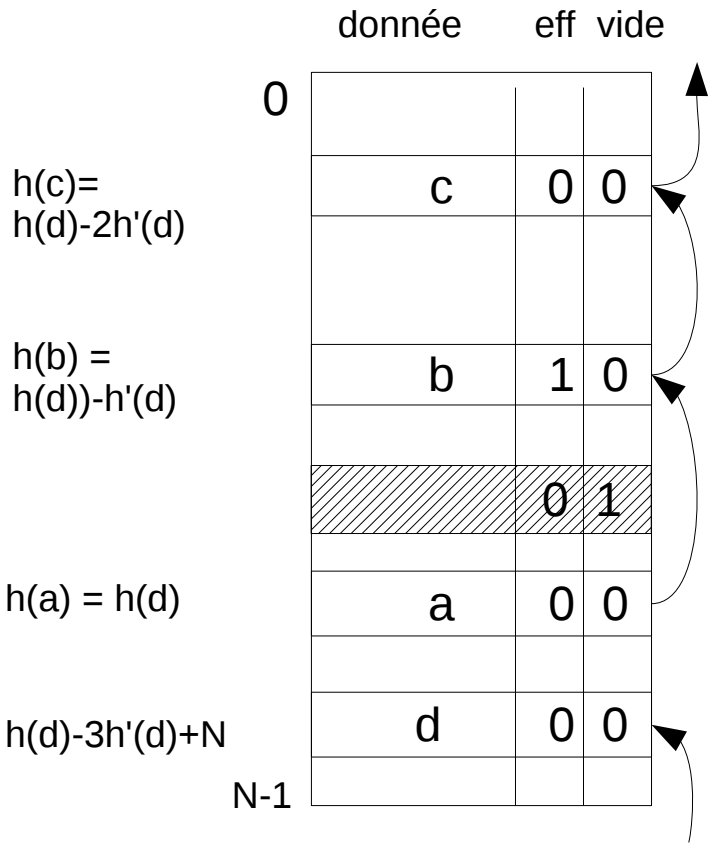
1 - (h(y) < i < j)
2 - (i < j ≤ h(y))
3 - (j ≤ h(y) < i)

Dans le cas où une de ces conditions est vraie, on déplace y dans la case j et on positionne j vers la case i

Double Hachage

- Comme l'essai linéaire sauf que la séquence cyclique des cases à tester utilise un **pas** calculé par une 2e fonction de hachage **h'(x)**
- Soient h(x) la fonction utilisée pour le calcul de l'adresse primaire et h'(x) la fonction qui calcul le pas de la séquence: h(x), h(x)-h'(x), h(x)-2h'(x), h(x)-3h'(x),...
- Pour que la séquence soit circulaire, les soustractions se font modulo N (la taille de la table)
  - C-a-d quand on calcule le nouvel indice i := i – h'(x), on rajoute juste après le test: SI (i < 0) i := i + N FSI
- Pour que la couverture soit total (passer par toutes les cases) il faut choisir comme taille de la table N, un **nombre premier**
- Pour simplifier les algorithmes, on sacrifie une case de la table
  - C-a-d il reste toujours au moins une case vide (critère d'arrêt)

Double Hachage - Fonctionnement



Recherche / Insertion

Similaire à l'essai linéaire

Suppression (Logique)

Chaque case renferme 2 bits :

vide : indiquant une case vide

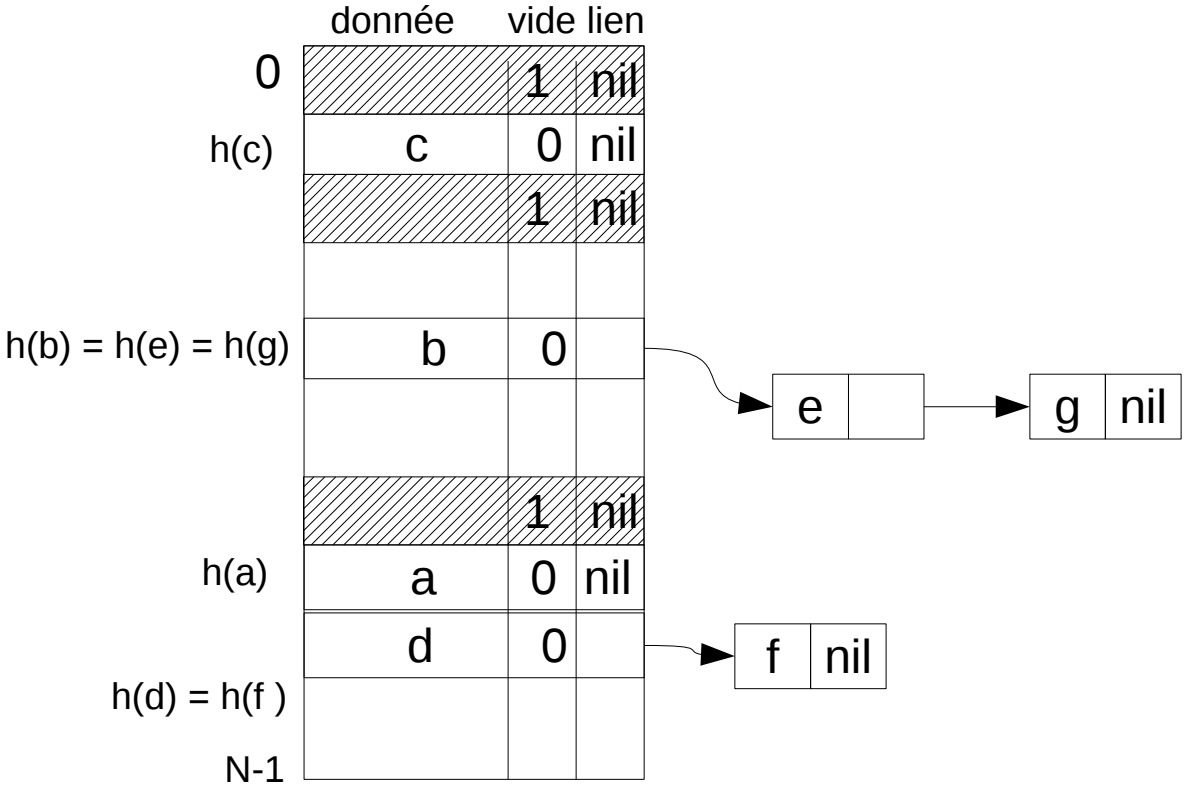
eff : indiquant un effacement logique
(**la case n'est pas vide)**
ex: b est supprimée logiquement

Réorganisations périodiques

pour récupérer l'espace perdu à cause des effacements logiques
==> réinsérer les données non effacées dans une nouvelle table

Chaînage séparé

Les données en débordement (en cas de collisions) sont stockés dans un espace non « adressable » par la fonction de hachage.
Par exemple à l'extérieur de la table sous forme de listes.



Le champ 'lien' lie les données en cases primaires avec leurs synonymes en débordement.

Le nombre de données insérées peut dépasser la taille de la table (N)

Chaînage séparé – opérations de base

Recherche de x

accès direct à la case h(x)
SI échec et non 'vide', Continuer la recherche en séquentielle dans la liste 'lien'

Insertion de x

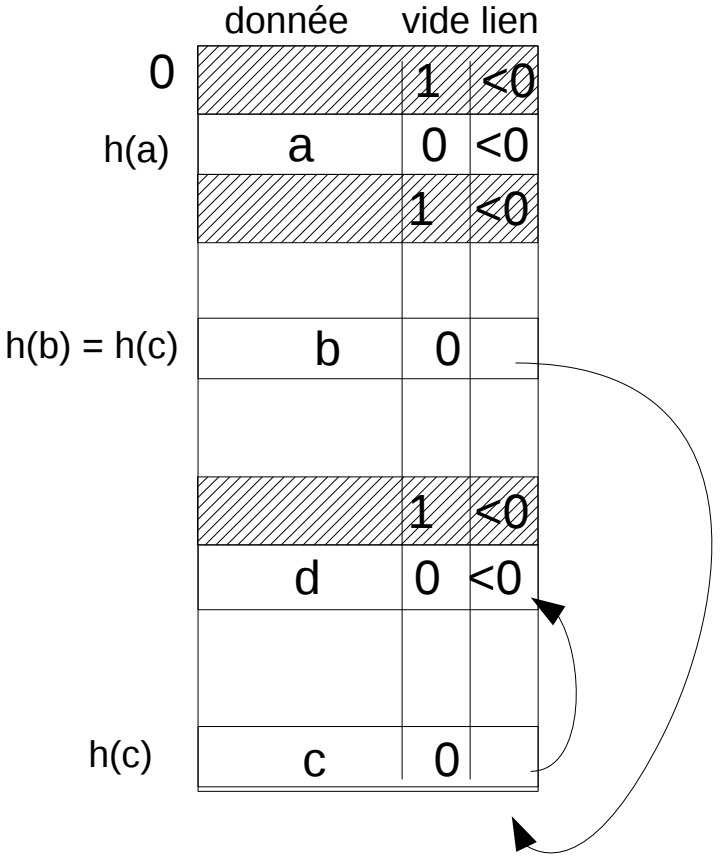
soit dans la case h(x) si elle était vide
soit dans la liste associée si h(x) n'est pas vide. (insertion en début de liste)

Suppression x (physique)

SI x se trouvait dans son adresse primaire (la donnée de la case h(x)),
    SI la liste 'lien' n'est pas vide
        déplacer le premier élément de la liste dans la case h(x) (en écrasant x)
    SI la liste 'lien' est vide
        vider la case h(x)
SI x se trouve en débordement dans une liste, la supprimer de cette liste

Chaînage interne

Les données en débordement (en cas de collisions) sont stockés dans la table (dans le même espace « adressable » par la fonction) en gardant le chaînage entre les synonymes



Les 'liens' sont des entiers (indices de cases)

<0 indique la fin de liste (nil)

Chaînage interne – opérations de base

- Recherche
  - Comme dans le chaînage séparé
- Insertion
  - Rechercher la donnée, si échec, 2 possibilités:
    la case h(x) est vide, on y insère la donnée
    la case h(x) n'est pas vide,
      soit i la dernière case visitée par la recherche, trouver une case vide
      insérer la donnée dans cette case vide et la chaîner à la suite de i
- Suppression (physique)
  - Rechercher l'adr i de la donnée (et de son prédécesseur i' dans la liste)
  - Vérifier les suivants de i pour voir s'il y en a un qui a son adr primaire = i auquel cas on le déplace vers i et on tente de vider son emplacement
  - S'il n'y a pas de pb avec les suivants, on vide la case i en mettant à jour son prédécesseur i' pour « pointer » le suivant de i
    ---> PB : est-ce que le prédécesseur existe toujours ?
    ---> une solution : listes circulaires.

Estimation des débordements

Soit une table de **N** cases, et on aimerait insérer **r** données

Le pourcentage de remplissage (la densité) est donc: **d** = r / N

Soit P(x) la probabilité que x données parmi r soient « hachées » vers la même case

$$P(x) = C_r^x \, (1 - 1/N)^{r-x} \, (1/N)^x$$

La fonction de Poisson en est une bonne approximation, en supposant une fonction de hachage uniforme

$$P(x) = (d^x \; e^{-d}) / x! \qquad \text{(avec } d = r/N\text{)}$$

N*P(x) est donc une estimation du nombre de cases ayant été choisies x fois durant l'insertion des r données dans la table

Le nombre total de données en débordement est alors estimé à :

NP(2) + 2N*P(3) + 3N*P(4) + 4N*P(5) + ... 0

Estimation débordements

- exemple numérique -

Lors de l'insertion de 1000 données dans une table de 1000 cases (**densité = 1**), on estime que :

N.P(0) = 368 cases ne recevront aucune données
N.P(1) = 368 cases auront été choisies 1 seule fois
N.P(2) = 184 cases auront été choisies 2 fois
N.P(3) = 61 cases auront été choisies 3 fois
N.P(4) = 15 cases auront été choisies 4 fois
N.P(5) = 3 cases auront été choisies 5 fois
N.P(6) = 0 cases auront été choisies 6 fois

Le nombre de données en débordement est proche de :

184 + 2*61 + 3*15 + 4*3 = 363 soit **36%** des données

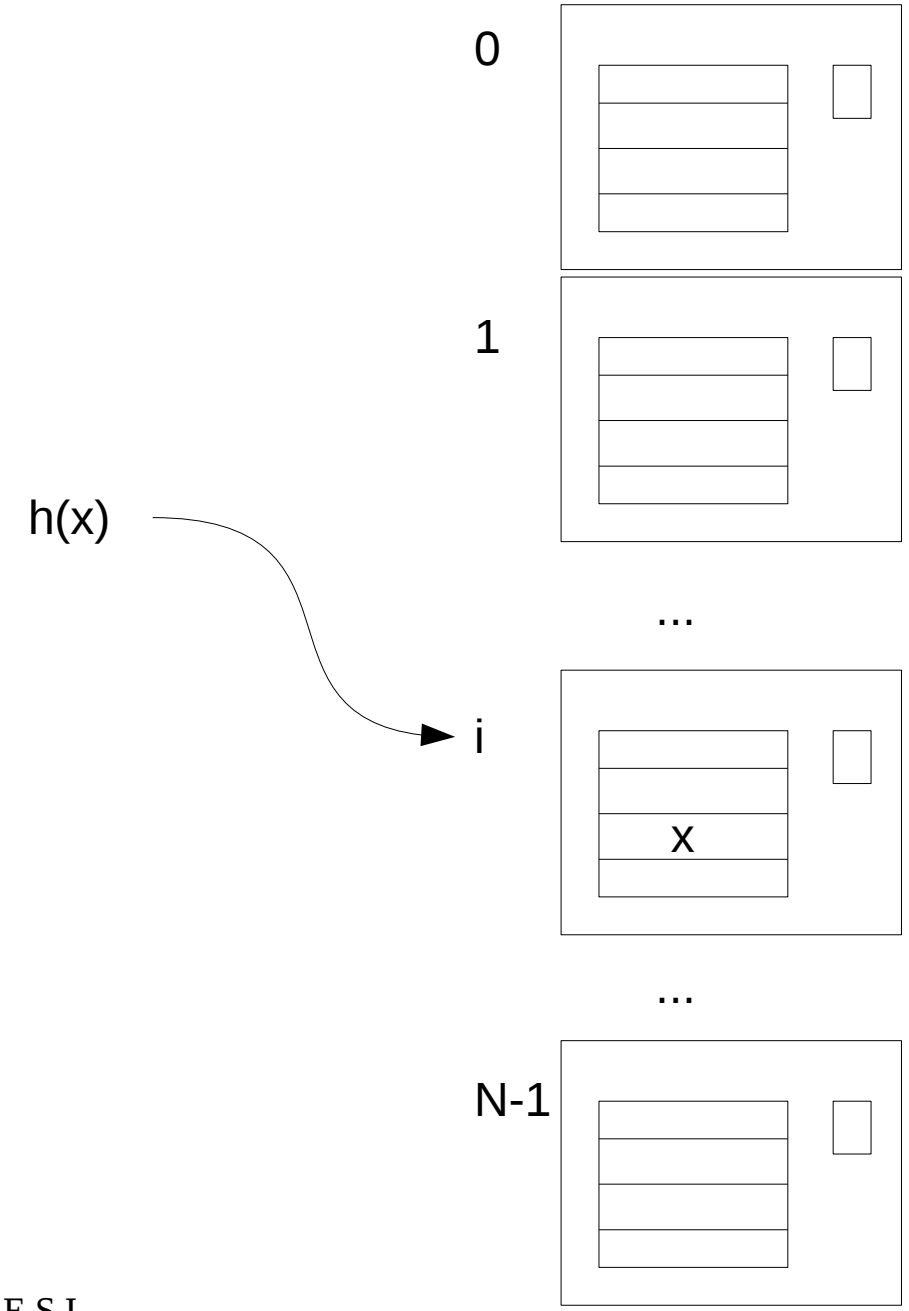
contre 631 données dans leurs adresses primaires

(368 + 184 + 61 + 15 + 3 = 631)

Pour une **densité = 0.5**, (ex r = 500 et N = 1000), on aurait eu **21%** de données en débordement

Fichier avec hachage

- L'adresse primaire de l'enregistrement de clé x est le bloc numéro h(x)
- Si le bloc est plein, on utilise l'une des méthodes de résolution de collisions.
- La capacité d'un bloc est de b enregistrements



Fichier avec hachage

- Exemple : recherche avec essai linéaire -

```
Rech(x, nomfich, var i, var j)
   Ouvrir( F, nomfichier, 'A' ) ;
   i ← h(x) ; trouv ← faux ; stop ← faux ;
   TQ  ( Non trouv && Non stop )
         LireDir( F, i, buf ) ;
         j ← 1 ;                                  // rech interne ...
         TQ  ( j <= buf.NB  &&  Non trouv )
               SI  ( x = buf.tab[ j ].cle )  trouv ← vrai
               SINON                         j ← j+1
               FSI
         FTQ
         SI ( buf.NB < b )                        // présence d'une case vide
             stop ← vrai
         FSI
   FTQ ;
   Fermer( F )
```

Fichier avec hachage

- Estimation des débordements -

N*P(x) : estimation du nombre de blocs ayant été choisis x fois durant l'insertion de r enregistrements dans un fichier contenant N blocs.

Comme la capacité maximale d'un bloc est de b enregistrements, il y aura alors collision sur un bloc i, seulement s'il a été choisi plus de b fois, durant l'insertion des r enregistrements.

Donc si un bloc aura été choisi b+1 fois, il provoquera 1 enregistrement en débordement.

De même, si un bloc aura été choisi b+2 fois, il provoquera 2 enregistrements en débordement

...

Le nombre total de données en débordement est alors estimé à :

$$NP(b+1) + 2N^*P(b+2) + 3N^*P(b+3) + 4N^*P(b+4) + \ldots 0$$

Fichier avec hachage Dynamique

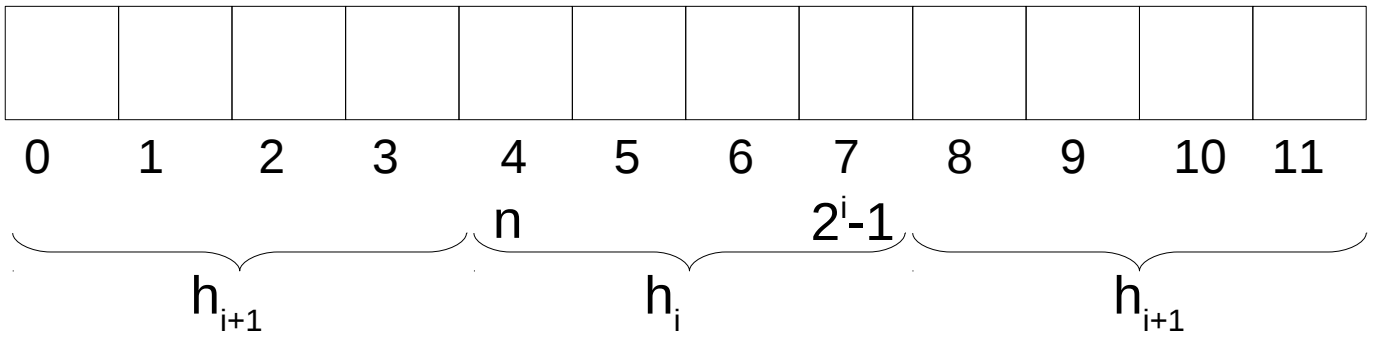
Les méthodes de hachage que l'on vient de voir sont dites « statiques », car si le nombre d'insertion devient important, les performances se dégradent à cause des nombreuses données en débordement.

Dans les méthodes du hachage dynamique, la fonction de hachage h change dynamiquement pour s'adapter à la taille du fichier, ce qui permet de garder de bonnes performances même si le nombre d'insertion (ou de suppression) augmente.

Le **Hachage Linéaire** est l'une des méthodes du hachage dynamique parmi les plus performantes

Hachage Linéaire

- Principe -



Paramètres
n : ptr d'éclatement
i : niveau du fichier

Le fichier est formé par une suite de blocs (qu'on appelle « cases »)
On utilise 2 fonctions de hachage :

$$h_i(x) = x \bmod 2^i \quad \text{et} \quad h_{i+1}(x) = x \bmod 2^{i+1}$$

Le nombre de cases augmente et diminue en fonction des insertions et des suppressions

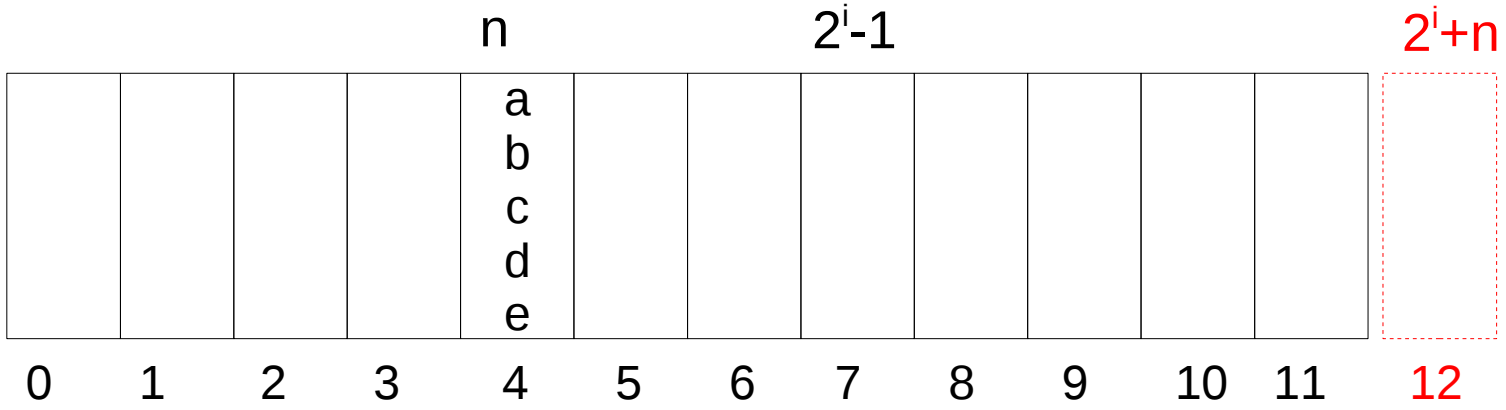
- à chaque fois que le taux de chargement dépasse un seuil, la case n éclate (une nouvelle case est rajoutée au fichier) et n est incrémenté (n++)

La fonction de hachage change dynamiquement en fonction de la taille du fichier

- quand n atteint 2^i, on fait : i++ ; n ← 0

Hachage Linéaire

- Eclatement d'une case -

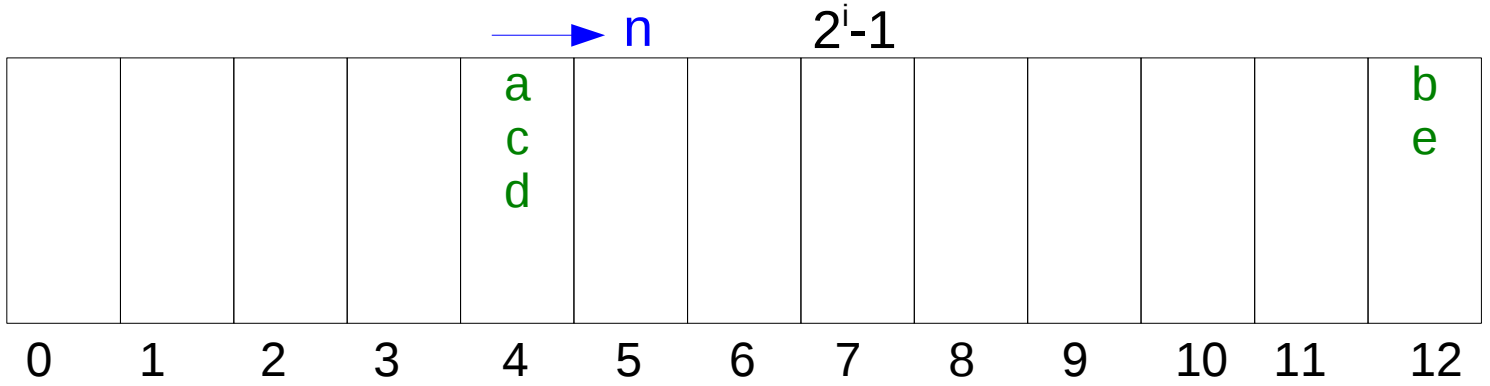


Paramètres
n : ptr d'éclatement
i : niveau du fichier

Lors d'une insertion, si le taux de chargement du fichier dépasse un certain seuil, la case n éclate : 1- Allocation d'une nouvelle case à la fin du fichier (bloc num : 2^i+n)

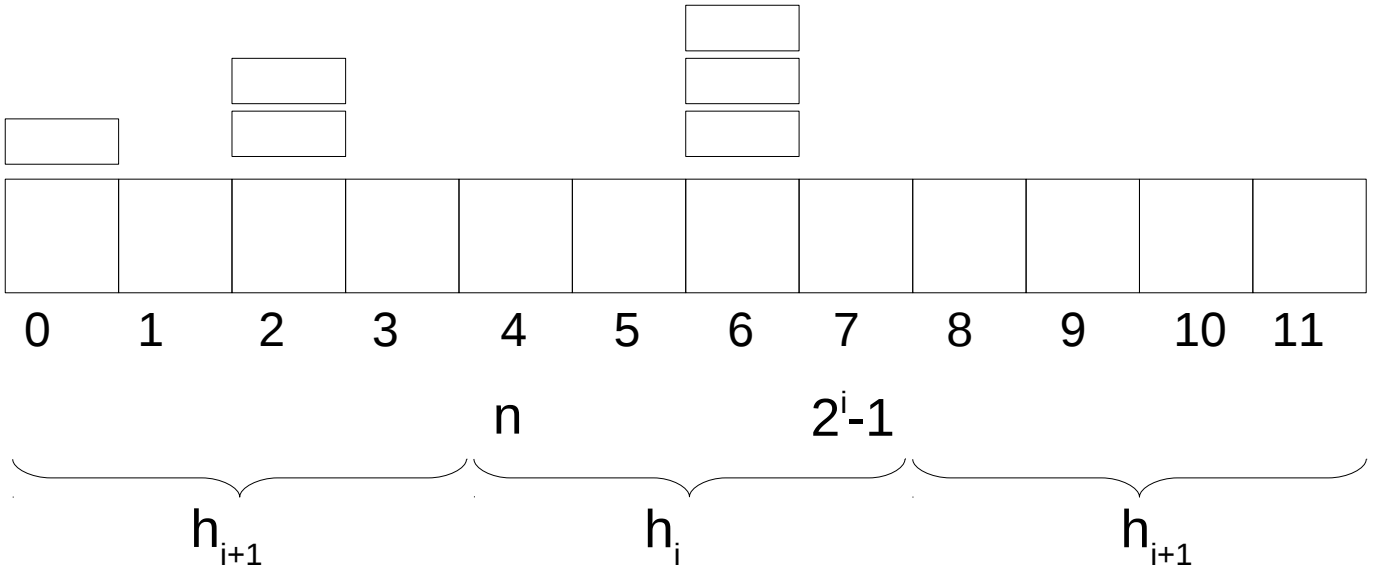
2- Rehachage des données contenues dans n avec la fonction h_{i+1}

3- Incrémentation de n



Paramètres
n : ptr d'éclatement
i : niveau du fichier

Recherche d'un élément



Paramètres
n : ptr d'éclatement
i : niveau du fichier

Rechercher x : a ← $h_i(x)$; Si (a < n) a ← $h_{i+1}(x)$ Fsi Retourner a ;	$h_i(x) = x \bmod 2^i$ $h_{i+1}(x) = x \bmod 2^{i+1}$