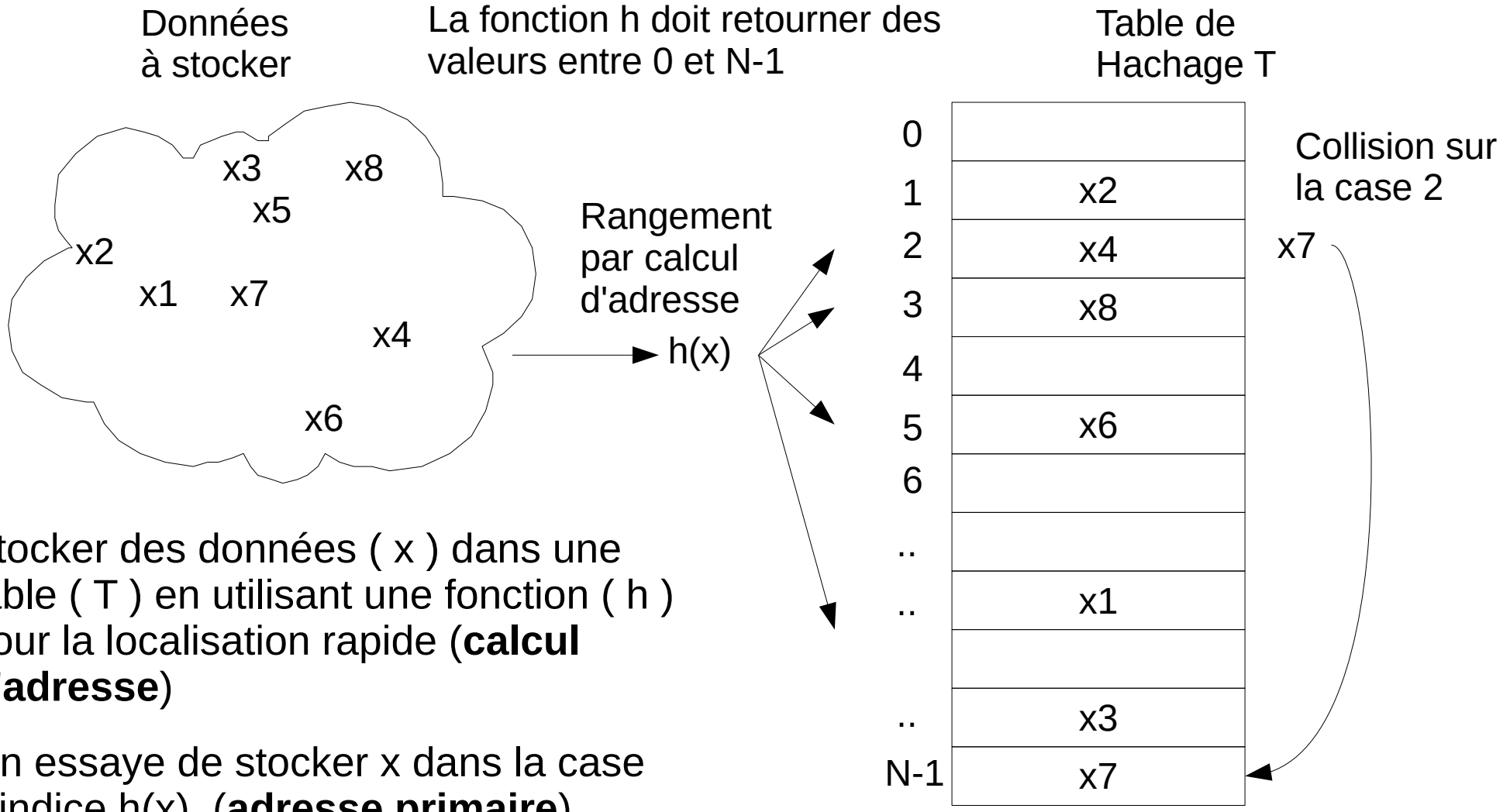
Hachage pour les Fichiers

Utilisation de méthodes de hachage
pour l'accès aux fichiers

PLAN

- Définition et Rappels
- 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

Méthodes de hachage

Combiner : une ***fonction de hachage*** avec
une ***méthode de résolution de collisions***

- Ex. de fonction de hachage: ***h(x) = x MOD N***

  L'espace adressable par la fonction est : ***[0 , N-1]***

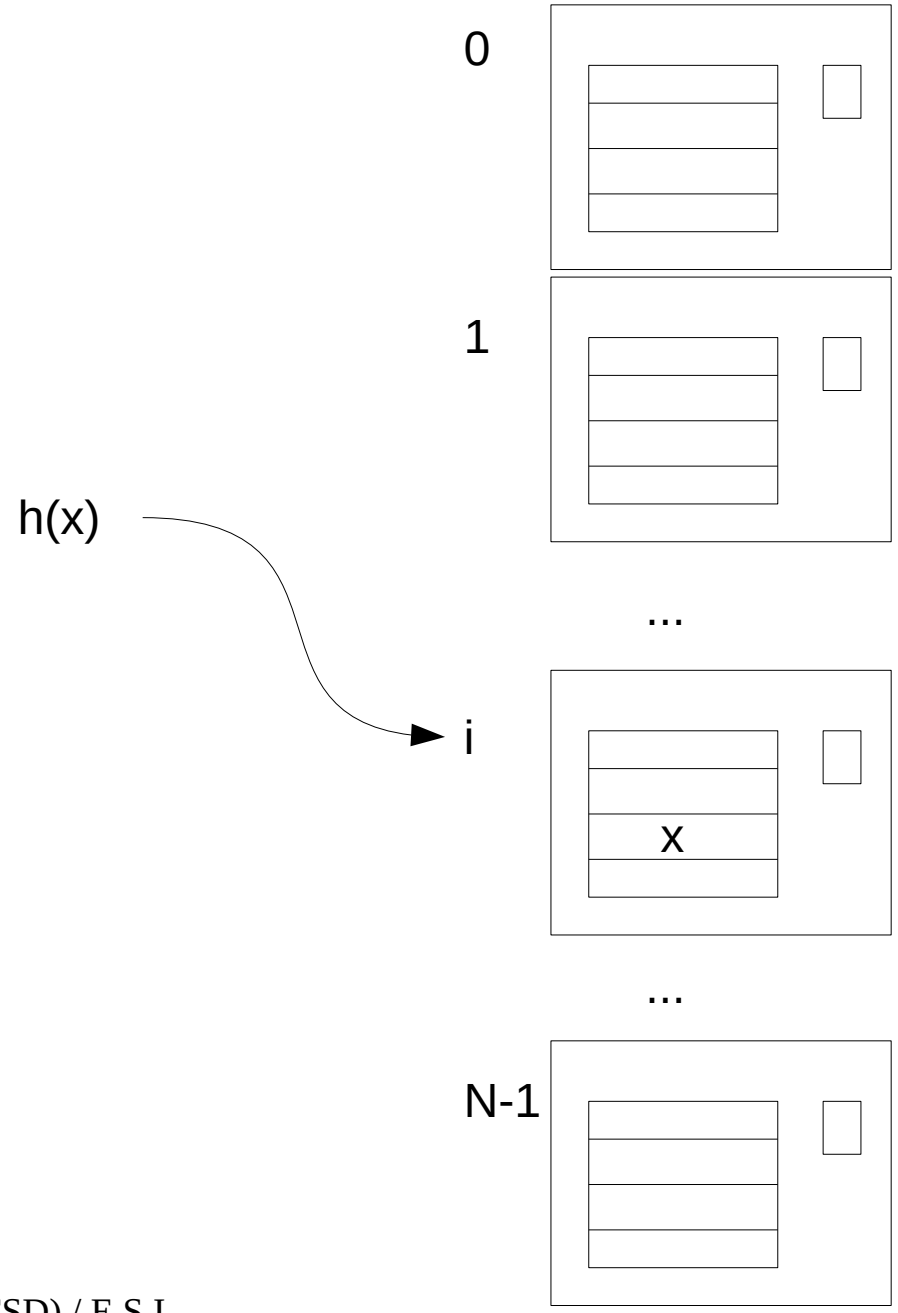
- Bonne propriété d'une fonction de hachage :
  ⇒ minimise les collisions tout en minimisant l'étendue de l'espace adressable.

- Une méthode de résolution de collision permet de gérer (rechercher, insérer et supprimer) les données ayant provoqués des collisions

  Ex. *Essai-Linéaire, Chaînage externe, Double Hachage …*

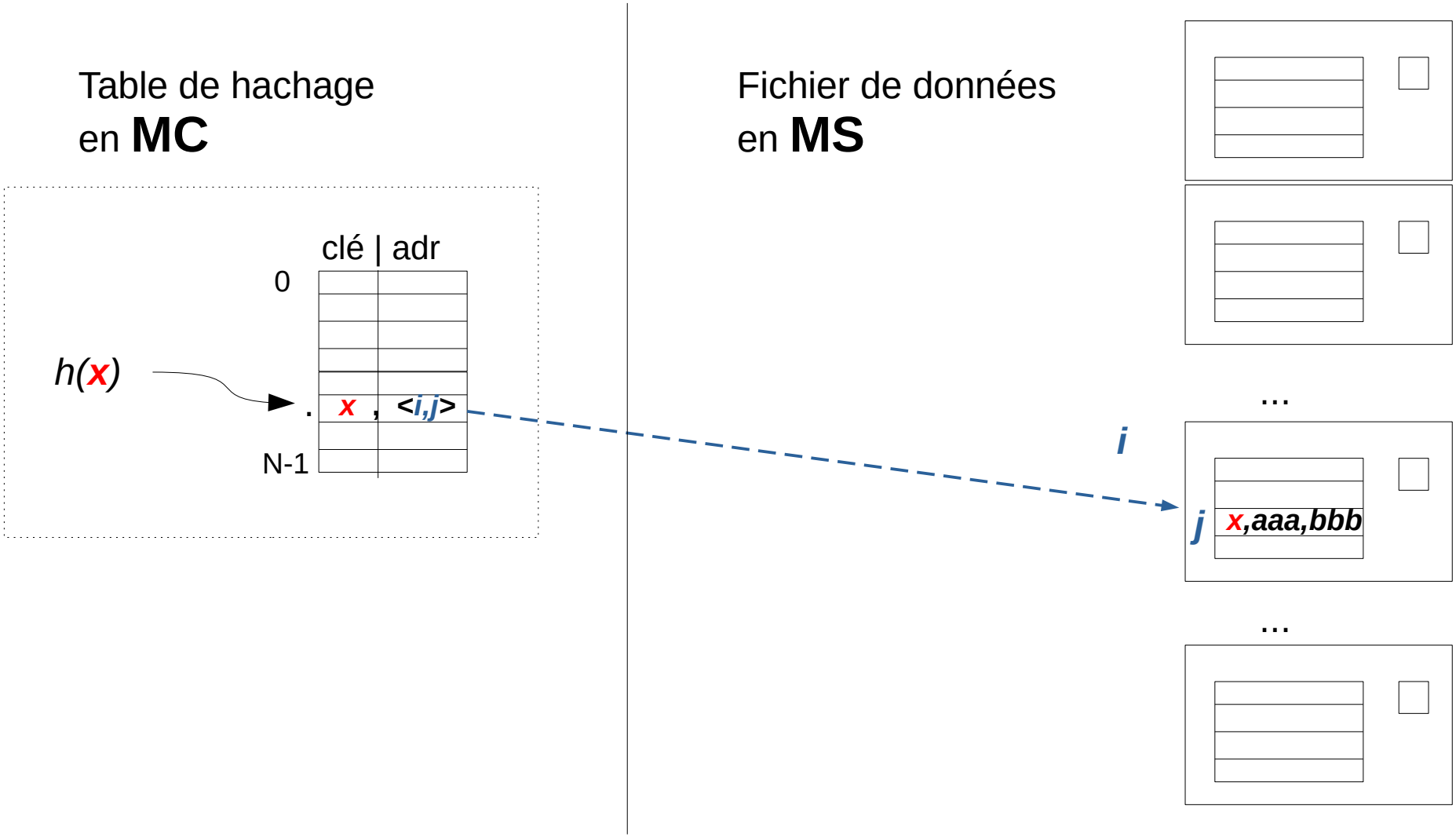
Fichier avec hachage

- 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



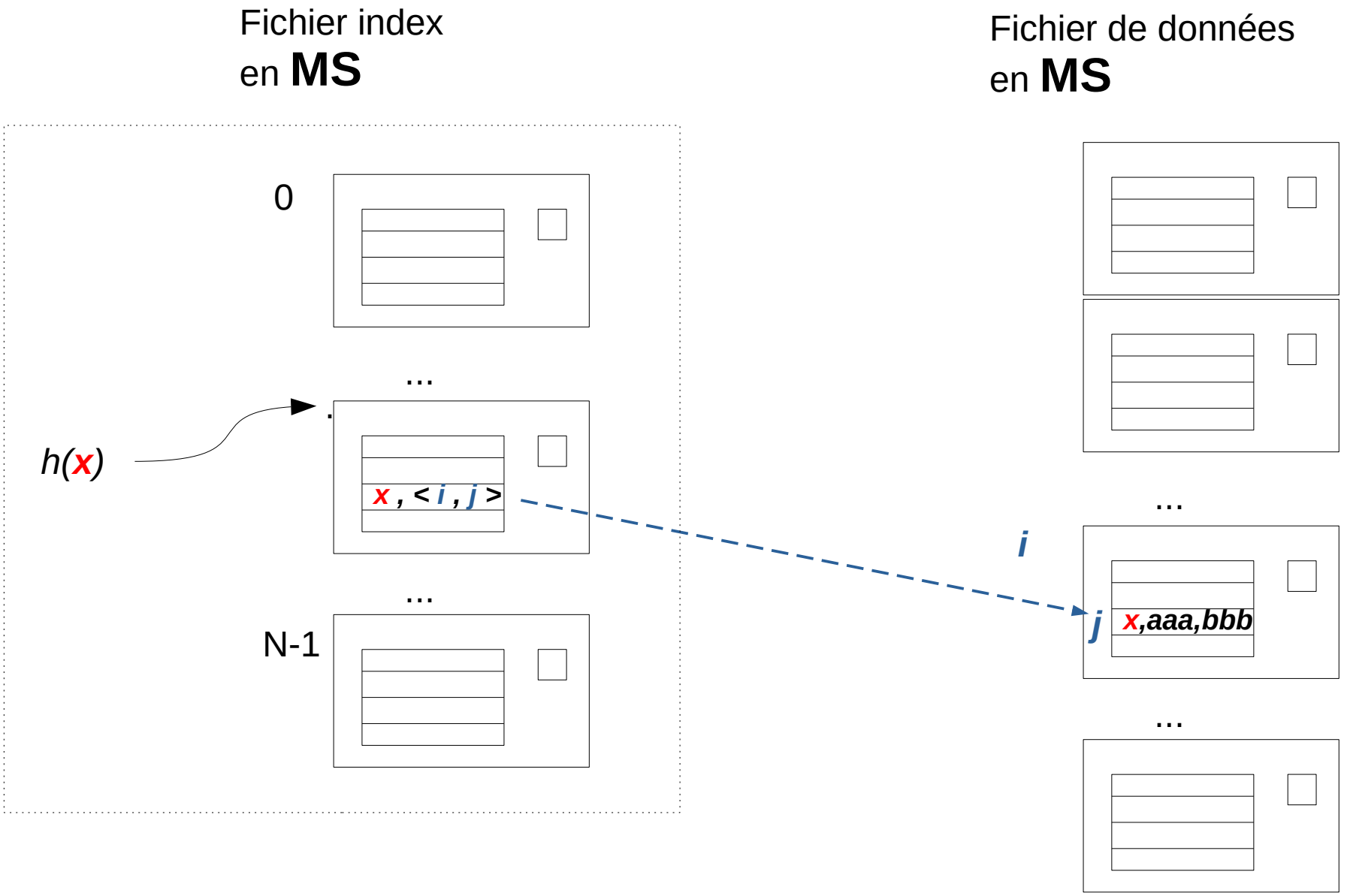
Utilisation d'une méthode de hachage comme structure de fichiers

1. Utiliser une table de hachage comme un index, en **MC**, pour accélérer les accès aux fichiers de données.



Utilisation d'une méthode de hachage comme structure de fichiers

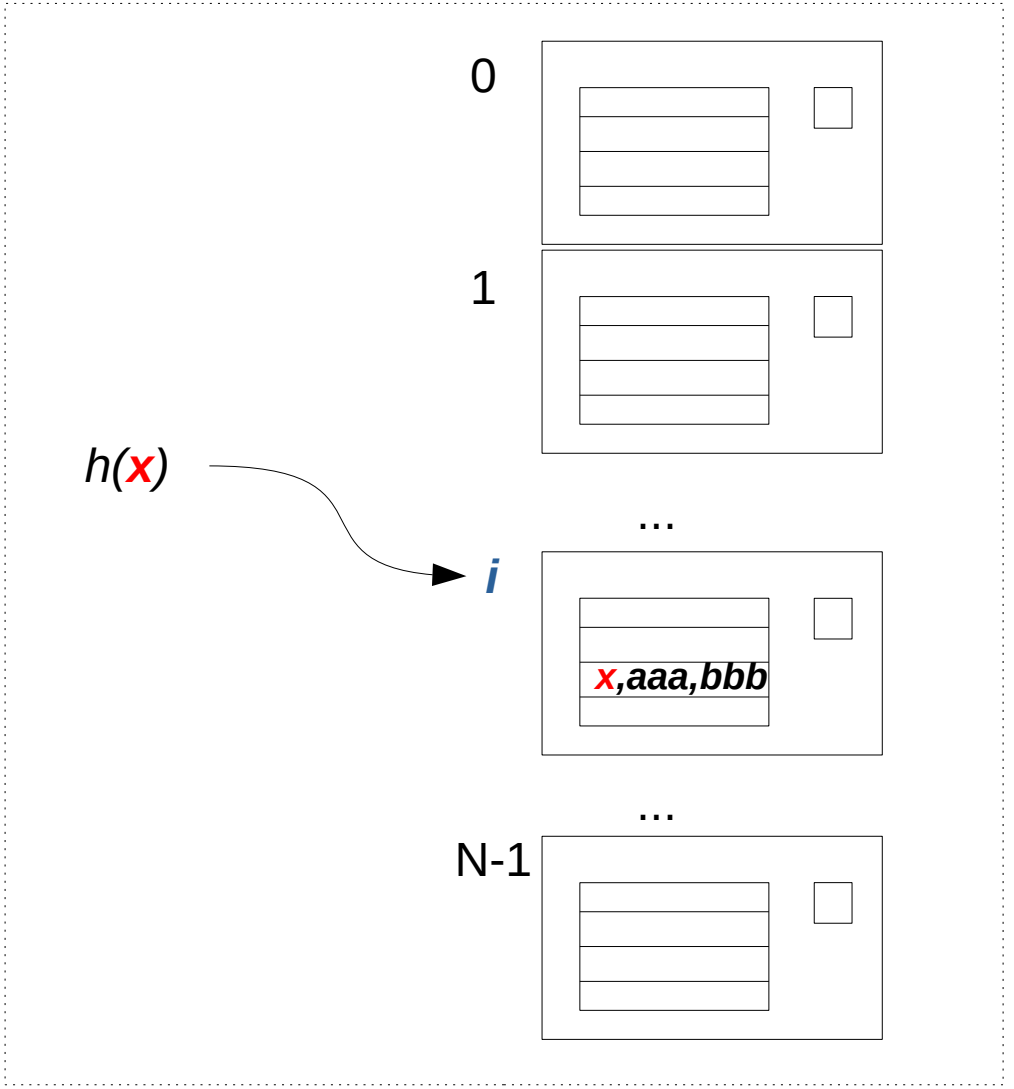
2. Utiliser un index en **MS**, géré par une méthode de hachage.



Utilisation d'une méthode de hachage comme structure de fichiers

3. Gérer le fichier de données par une méthode de hachage.

Fichier de données
en **MS**



Méthodes de résolution de collisions

- Essai Linéaire

Séquence de tests : les blocs n° *h(x) , h(x)-1 , h(x)-2 , … 0 , N-1 , … <bloc_non_plein>*

- Double Hachage

Séquence de tests : les blocs n° $h_1(x)$, $h_1(x)-h_2(x)$, $h_1(x)-2h_2(x)$, … *<bloc_non_plein>*

- Chaînage Externe

Séquence de tests : le bloc n° *h(x) et ceux de sa liste de débordement qui se situent en dehors de la zone adressable par la fonction* ***h***

- Chaînage Interne

Séquence de tests : le bloc n° *h(x) et ceux de sa liste de débordement qui se situent à l'intérieur de la zone adressable par la fonction* ***h***

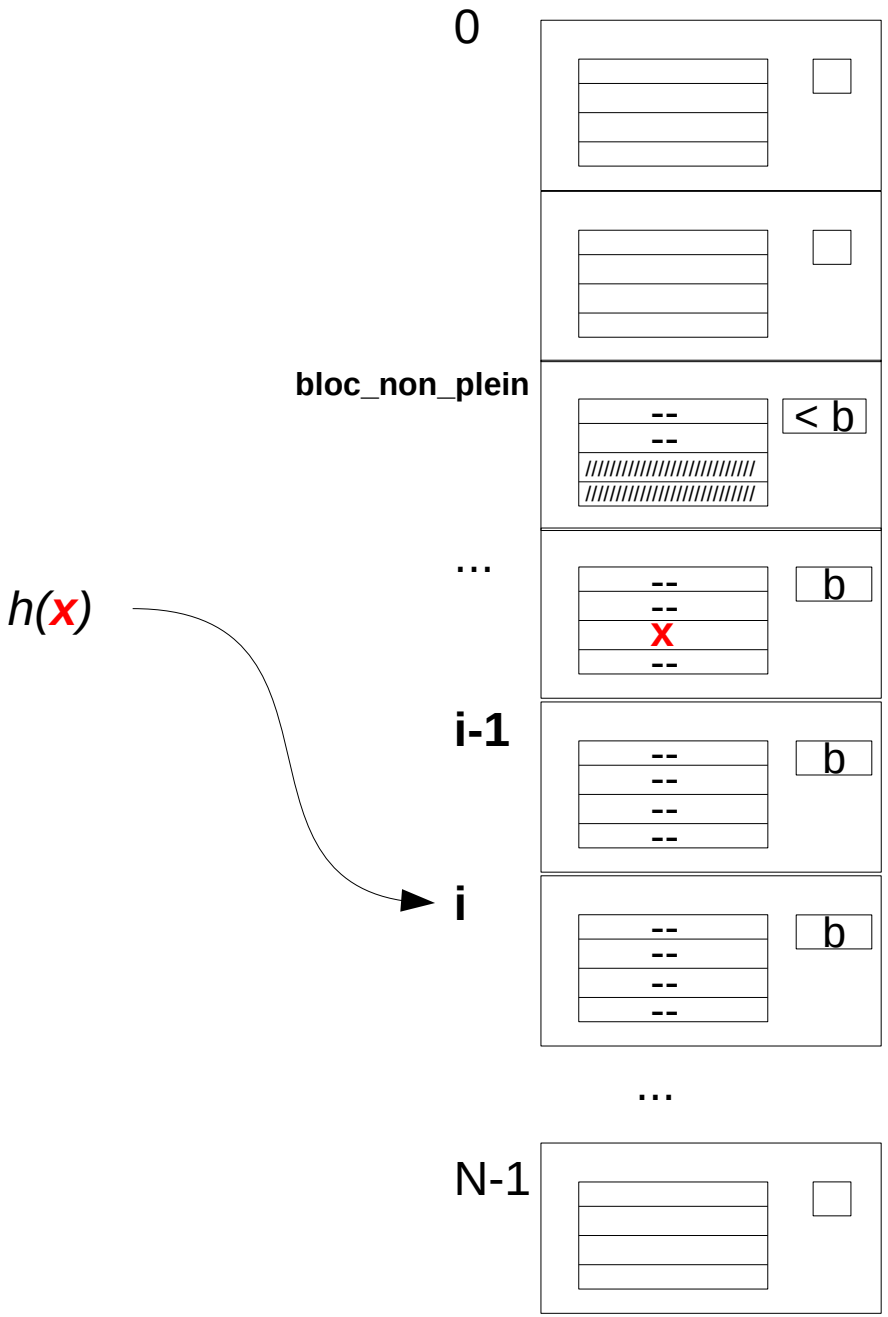
Essai Linéaire

Il doit toujours exister au moins un bloc non plein

La séquence de tests peut éventuellement être circulaire

L'insertion se fait dans le 1er bloc non plein, trouvé dans la séquence de tests

La suppression Logique ou Physique



- Algorithme de Recherche / Essai Linéaire -

Les caractéristiques :

1- N : le nombre de blocs formant le fichier **/** 2- nbIns : le nombre de données insérées

Rech(entrée : x sorties : trouv, i, j)

// on suppose que le fichier F est déjà ouvert

i ← *h(x)* ; trouv ← faux ; stop ← faux ; N ← Entete(F, 1)

TQ (Non stop)

LireDir(F, i, buf)

j ← 1 // Recherche interne dans le bloc i
TQ (j ≤ buf.NB && Non trouv)
SI (x = buf.tab[j].cle) trouv ← vrai **SINON** j ← j+1 **FSI**
FTQ

SI (trouv || buf.NB < b) // Si x existe ou présence d'un bloc non plein
stop ← vrai // Alors fin de la séquence de tests
SINON
i ← i - 1 ; **SI** (i < 0) i ← N-1 **FSI** // Sinon on continue les tests
FSI

FTQ

- Algorithme d'Insertion / Essai Linéaire -

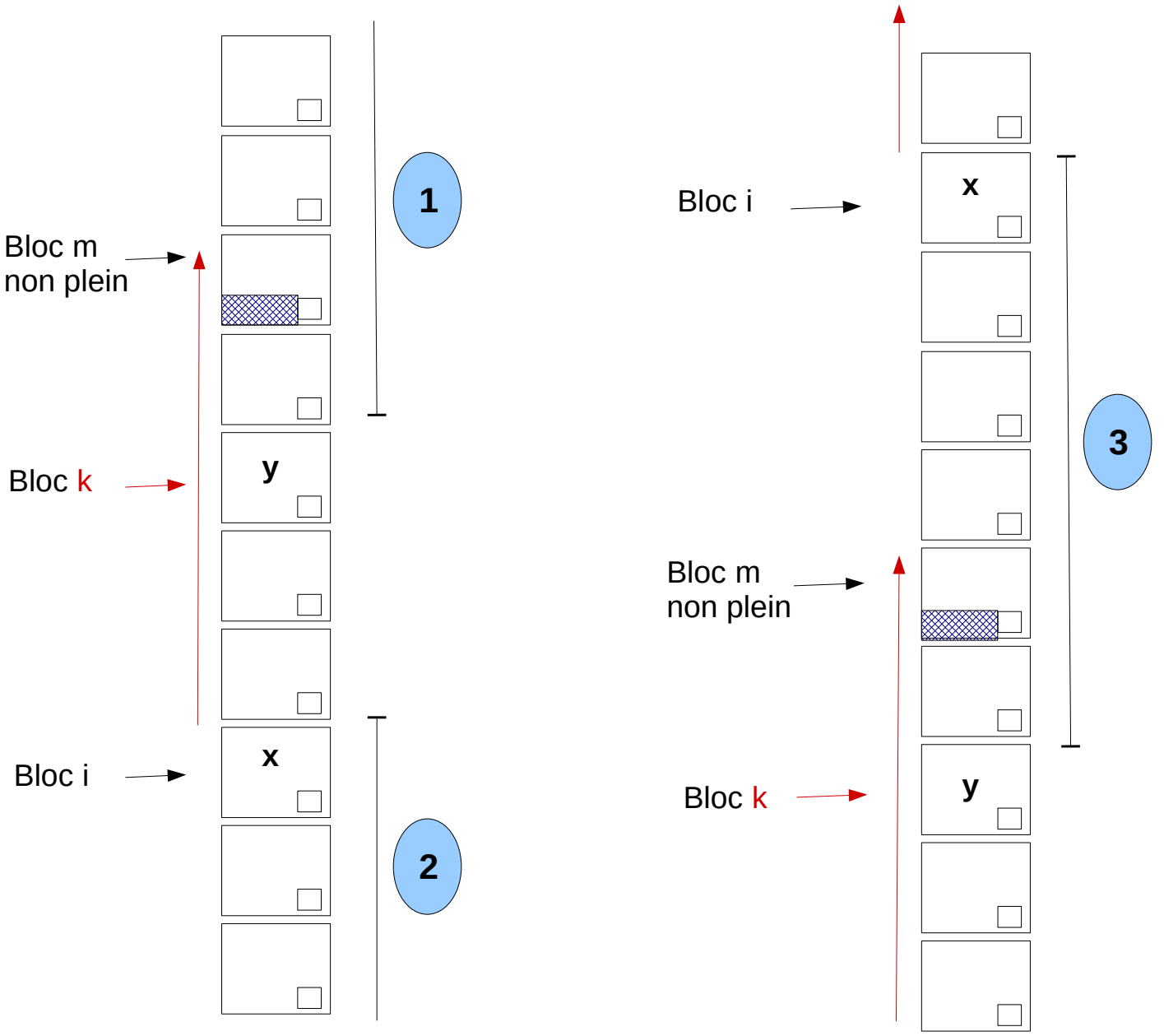
Les caractéristiques :

1- N : le nombre de blocs formant le fichier
2- nbIns : le nombre de données insérées

Ins(entrée : e)

```
// on suppose que le fichier F est déjà ouvert
N ← Entete( F, 1 ) ;  nbIns ← Entete( F, 2 )
SI ( nbIns = N * b – 1 ) Insertion impossible      // pas de place disponible
SINON
    Rech( e.clé , trouv , i , j )
    SI ( Non trouv )
        buf.NB++                                   // buf contient déjà le contenu
        buf.tab[ buf.NB ] ← e                      // du bloc i (voir Rech)
        EcrireDir( F, i, buf )
        Aff_Entete( F , 2 , nbIns+1 )
    FSI
FSI
```

- Mécanisme de Suppression (physique) / Essai Linéaire -

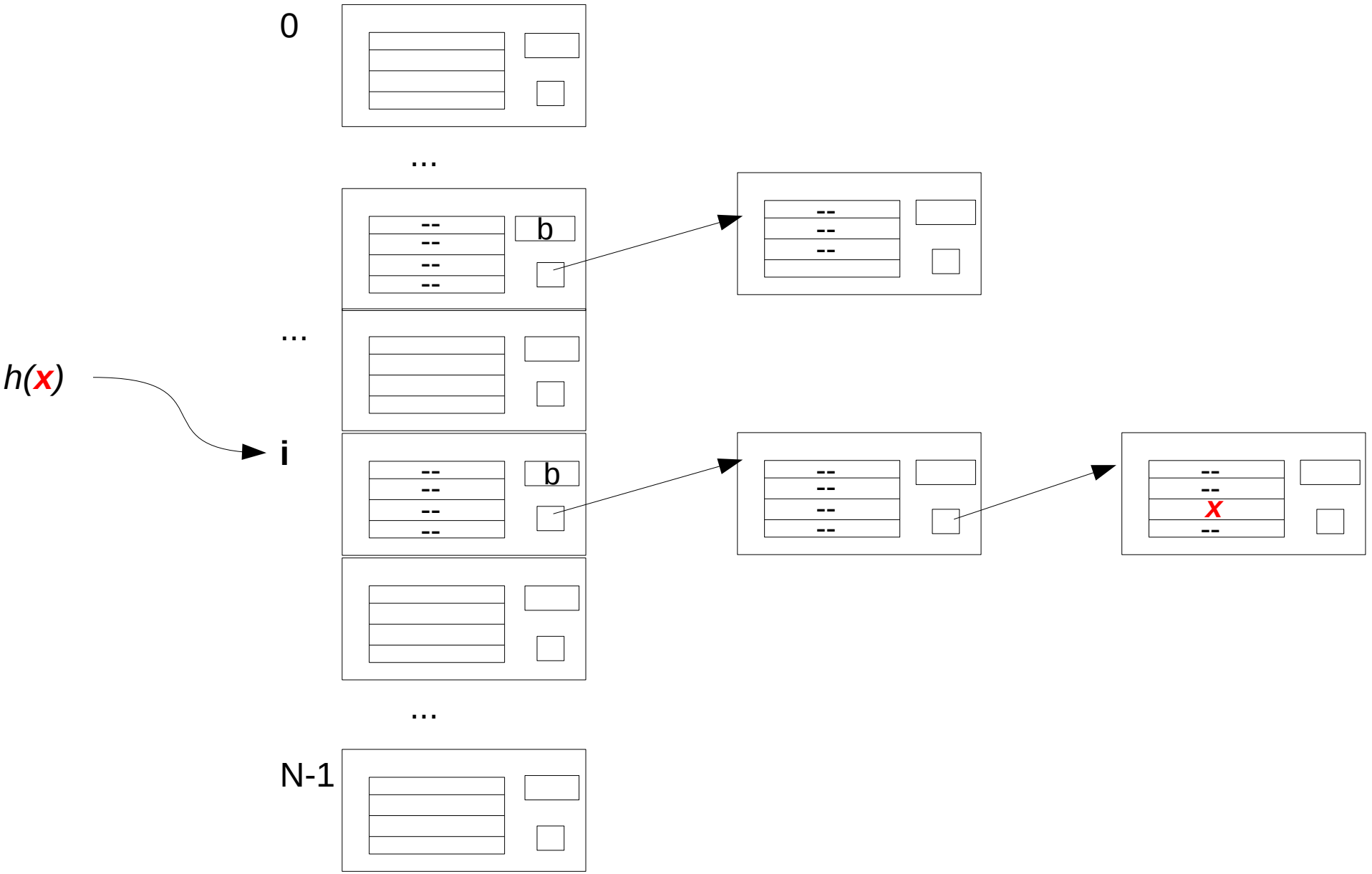


Tester tous les blocs ***k***

1ere condition : **h(y) < k < i**
2e condition : **k < i ≤ h(y)**
3e condition : **i ≤ h(y) < k**

Si une donnée ***y*** pose pb
Déplacer ***y*** à la place de ***x***

Chaînage Externe



- Algorithme de Recherche / Chaînage Externe -

On suppose que ***F*** contient la zone principale (les blocs entre ***0*** et ***N-1***) et la zone de débordement (les blocs à partir du numéro ***N*** jusqu'à la fin du fichier)

Les caractéristiques :

1- ***N*** : le nombre de blocs formant la zone principale
2- ***M*** : le nombre total de blocs dans F (zone principale + zone de débordement)
3- ***nbIns*** : le nombre de données insérées

Rech(entrée : x sorties : trouv, i, j)

```
// on suppose que le fichier F est déjà ouvert
i ← h(x) ; trouv ← faux ;  stop ← faux ;   LireDir( F, i, buf )
TQ  ( Non trouv && Non stop )
      j ← 1                                   // Recherche interne dans le bloc i
      TQ  ( j ≤ buf.NB  &&  Non trouv )
            SI  ( x = buf.tab[ j ].cle )      trouv ← vrai    SINON   j++    FSI
      FTQ
      SI ( Non trouv )
           SI ( buf.lien <> -1 )  i ← buf.lien ; LireDir( F, i, buf ) SINON  stop ← vrai  FSI
      FSI
FTQ
```

- Algorithme d'Insertion / Chaînage Externe -

Ins(entrée : e , nomFich : chaîne)

```
Ouvrir( F , nomFich , 'A' )
Rech( e.clé, trouv, i, j )
SI  ( Non trouv )
    // s'il y a de la place dans le dernier blocs visité, on y insère e
    SI ( buf.NB < b )
        buf .NB++ ; buf.tab[ buf.NB ] ← e ;  EcrireDir( F, i, buf )
    SINON
    // si le dernier blocs est déjà plein, on alloue un nouveau bloc en débordement
        nouvBloc ← Entete( F , 2 ) + 1
        buf.lien ← nouvBloc          // chaîner le nouveau bloc avec le précédent (i)
        EcrireDir( F, i, buf )
        buf .NB ← 1 ; buf.tab[ 1 ] ← e    // insérer e dans le nouveau bloc
        buf.lien ← -1
        EcrireDir( F, nouvBloc, buf )
        Aff_Entete( F , 2 , Entete( F , 2 ) + 1 )    // le nombre total de blocs
     FSI
    Aff_Entete( F , 3 , Entete( F , 3 ) + 1 )        // le nombre d'insertions
FSI
Fermer( F )
```

Fichier avec hachage Dynamique

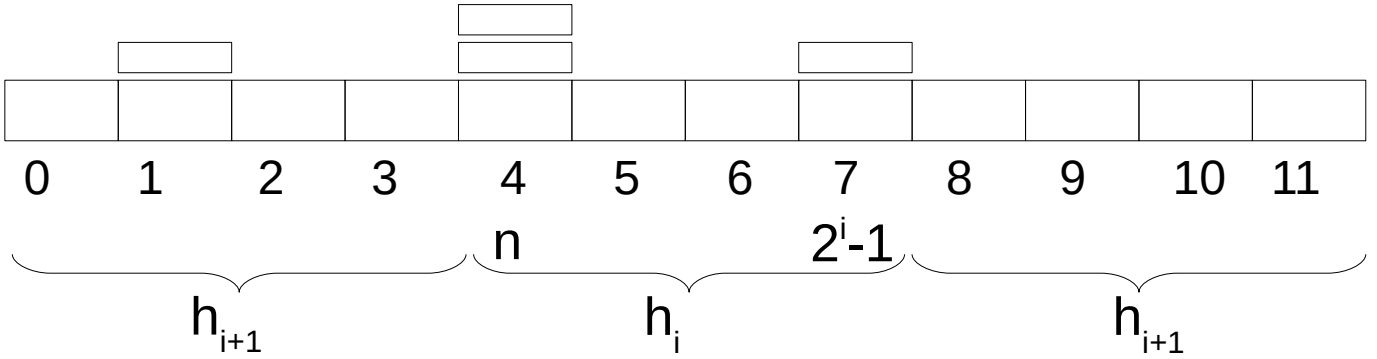
Les méthodes de hachage que l'on vient de voir sont dites « statiques », car si le nombre d'insertion devient important, les performances se dégradent à cause des nombreuses données en débordement.

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 principaux et une zone de débordement (chaque bloc avec sa liste de débordement est appelé « **case** »)

On utilise 2 fonctions de hachage : $h_i(x) = x \bmod 2^i$ et $h_{i+1}(x) = x \bmod 2^{i+1}$

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

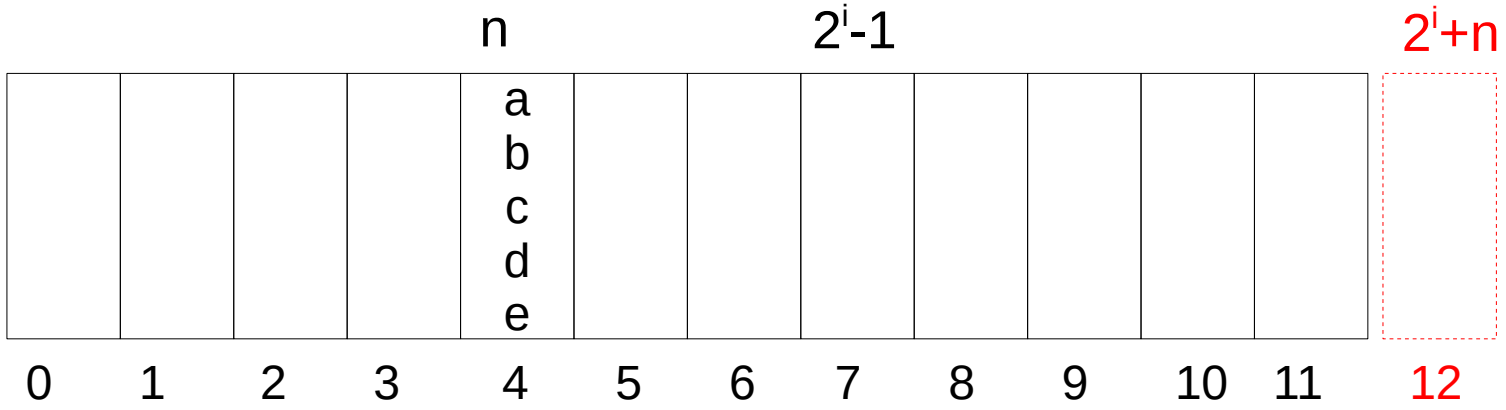
- à chaque fois que le taux de chargement dépasse un seuil, la case **n** éclate (une nouvelle case est alors rajoutée au fichier) et **n** est incrémenté modulo $\mathbf{2^i}$

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

- quand **n** atteint $\mathbf{2^i}$, il est réinitialisé à 0 et le niveau **i** augmente : **i++ ; n ← 0**

Hachage Linéaire

- Eclatement d'une case -

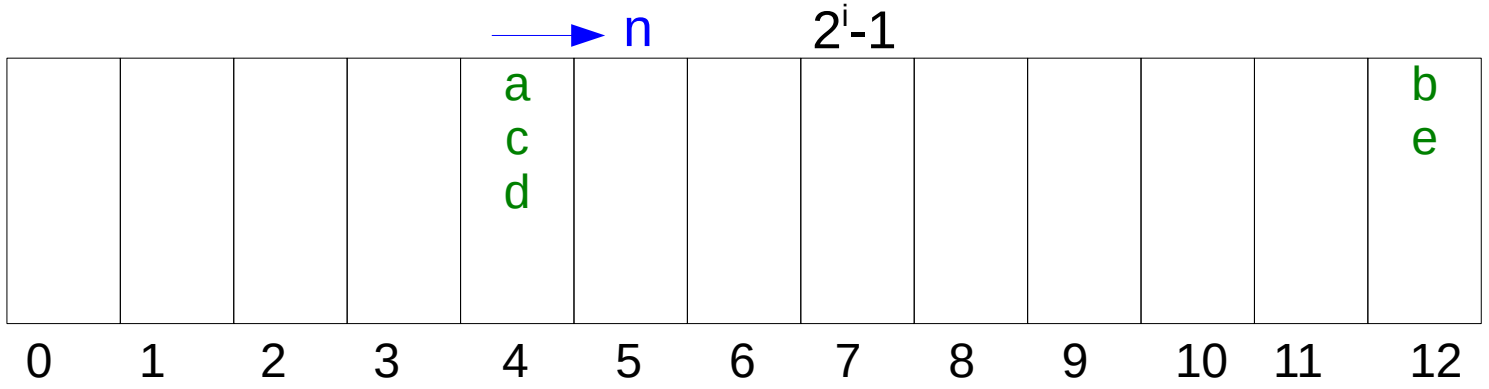


Paramètres
n : ptr d'éclatement
i : niveau du fichier

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)

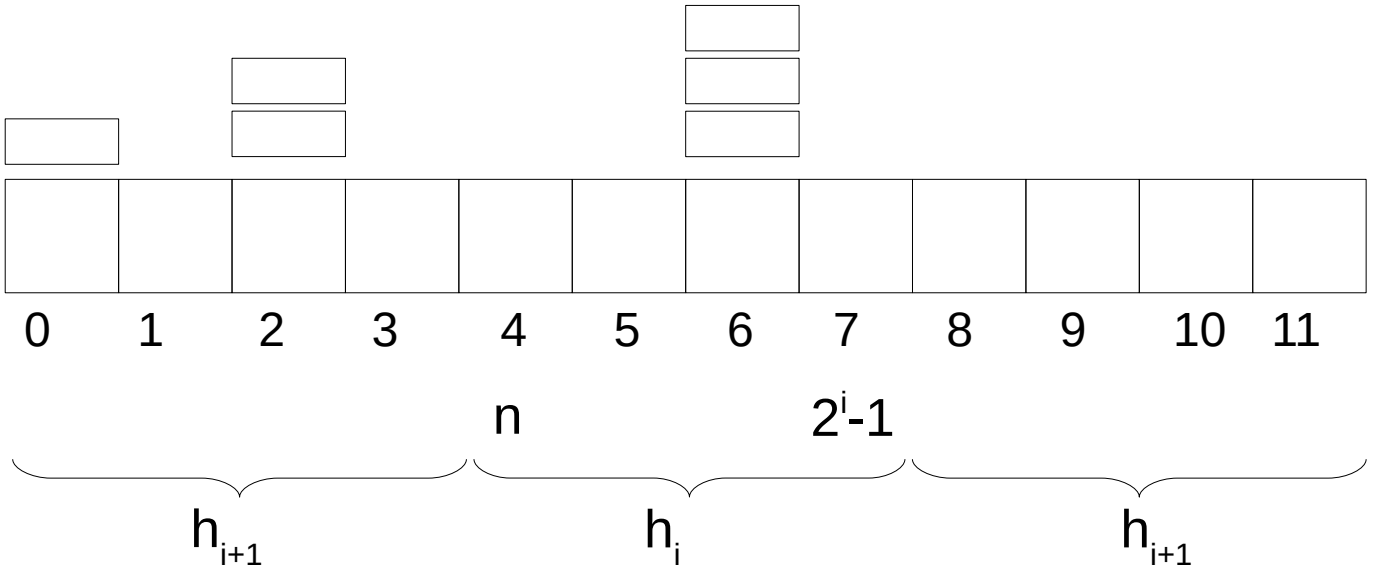
2- Rehachage des données contenues dans n avec la fonction h_{i+1}

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 ;
// num de la case où devrait se trouver x

$h_i(x) = x \bmod 2^i$

$h_{i+1}(x) = x \bmod 2^{i+1}$