Méthodes de hachage

1. **Introduction**
2. **Hachage ouvert**
3. **Hachage fermé**
4. **Implémentation des fonctions**

Recherche dichotomique

table

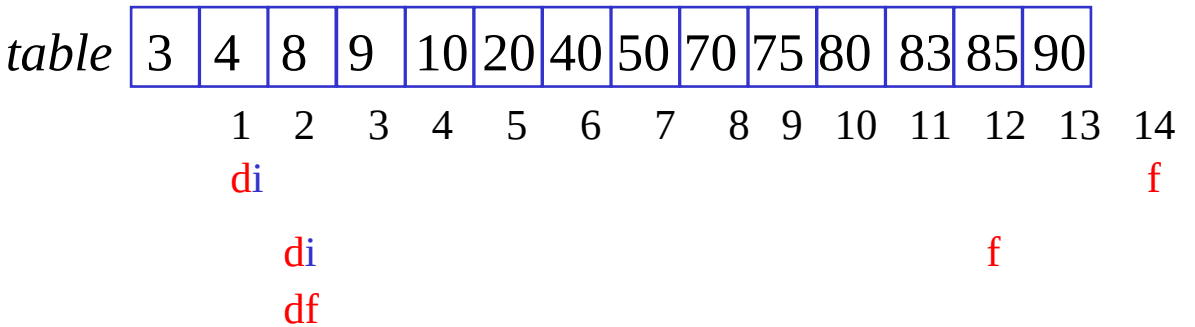
3	4	8	9	10	20	40	50	70	75	80	83	85	90
1	2	3	4	5	6	7	8	9	10	11	12	13	14
d						i							f
d			i			f							
d	i		f										
di	f												
	df												

4 ?

```
fonction  ELEMENT (x, table, d, f ) ;
début
    si  (d = f)  alors
        si  (x = table [d ])  alors  retour  (vrai)
        sinon  retour (faux)
     sinon {
        i ← ⌊(d+f ) / 2⌋ ;
        si  (x > table [i])  alors
            retour  (ELEMENT (x, table, i+1, f ))
        sinon  retour  (ELEMENT (x, table, d, i))
     }
fin
```

Temps (ELEMENT sur *table* [1 … *n*]) = O (log *n*)

Recherche par interpolation



fonction ELEMENT (*x*, *table*, *d*, *f*) ;
début
  si (*d* = *f*) **alors**
    si (*x* = *table* [*d*] **alors retour** (vrai)
    sinon retour (faux)
  sinon {

$$i \leftarrow d + \left\lfloor \frac{x - table[d]}{table[f] - table[d]} \right\rfloor * (f - d);$$

    si (*x* > *table* [*i*] **alors**
      retour (ELEMENT (*x*, *table*, *i*+1, *f*))
    sinon retour (ELEMENT (*x*, *table*, *d*, *i*))
  }
fin

Idée : Établir une relation entre un élément et l'adresse
à laquelle il est rangé en mémoire

Théorème : Si les éléments de *table* [1 … *n*] et *x* sont choisis uniformément dans un intervalle [*a*,*b*], le temps moyen d'exécution de la recherche par interpolation est O (log log *n*)

Une autre façon c’est d’utiliser
- hachage
- des tables de hachage
- des fonctionne de hachage

Une table de hachage est une structure de données qui implémente un tableau associatif (un dictionnaire)
Dans un tableau associatif, les données sont stockées sous la forme d’**une collection de paires clé-valeur**.
La position des données dans le tableau est déterminée par l’application d’un algorithme de hachage à la clé - un processus appelé **hachage**.
L’algorithme de hachage est appelé **fonction de hachage**.

- Les tables de hachage permettent une recherche très efficace.

- Dans le meilleur des cas, les données peuvent être extraites d’une table de hachage **en temps constant**.

- **La maintenance (ajout, mise à jour et suppression) de données** dans une table de hachage est également très efficace.

Comment implémenter une table de hachage ?

Pour implémenter une table de hachage :
1) Il faut **utiliser un tableau** car vous devez pouvoir accéder directement à chaque position du tableau.

2) Les positions au sein d'un tableau sont parfois appelées '**buckets** '; Chaque bucket est utilisé pour stocker des données.

3) La clé doit être stockée à côté des données

4) **La taille du tableau** doit être planifiée avec soin. Il doit être **suffisamment grand** pour stocker toutes les données, **mais pas trop grand** pour ne pas gaspiller de l'espace.

5) Une table de hachage efficace **doit toujours avoir de l'espace libre**.

des fonctionne de hachage

Une fonction de hachage est un algorithme qui convertit une clé de hachage en une valeur de hachage.

Les principales exigences d'une fonction de hachage:

1) produisent toujours la même valeur de hachage pour la même clé

2) fournir une distribution uniforme des valeurs de hachage. (chaque valeur a une probabilité égale d'être générée)

3) minimiser le clustering (la collision); Cela se produit lorsque de nombreuses clés différentes produisent la même valeur de hachage. Lorsque deux ou plusieurs clés différentes produisent la même valeur de hachage, on parle de « **collision** ».

4) être très rapide à calculer.

Exemple 1 :

supposons que les **clés sont des nombres à 5 chiffres** et que nous avons **une table de hachage avec 97 'bucket'**.

```
FUNCTION hash_integer(hash_key, number_of_slots)

// Générer la valeur de hachage à l'aide de l'opératrice modulo

hash_value = hash_key MOD number_of_slots

// Return the hash value

RETURN hash_value

ENDFUNCTION
```

Le resultat de hachage

Key	Hashing function	Hash value
12345	12345 MOD 97	26
67564	67564 MOD 97	52
34237	34237 MOD 97	93
23423	23423 MOD 97	46
00332	00332 MOD 97	41

Exemple 2 (hashage des chaîne de caractère) :

supposons que les **clés sont des chaîne de caractères comme TH5L** et que nous avons **une table de hachage avec 97 'bucket'**.

```
FUNCTION hash_string(hash_key, number_of_slots)
    // Initialise total
    total = 0

    // Repeat for every character in the hash key
    FOR i = 0 TO LEN(hash_key) - 1
       ascii_code = ASC(hash_key[i])
       total = total + ascii_code
   NEXT i

    // Generate the hash value using the modulo operator
    hash_value = sum MOD number_of_slots

    // Return the hash value
    RETURN hash_value
ENDFUNCTION
```

Le resultat de hachage

Par exemple si on considère A5RD:

ASC(A)+ASC(5)+...= 65 + 53 + 82+ 68 = 268

Et finalement: 268 MOD 97 = 74

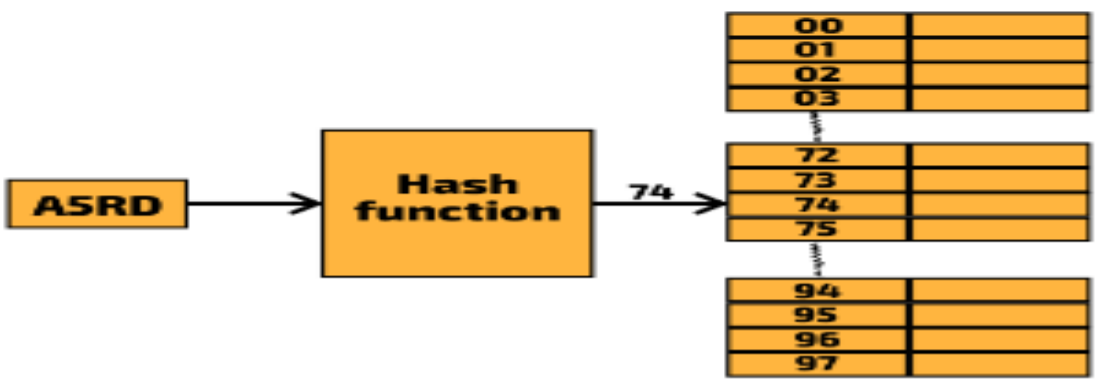
Les données associées au code A5RD seront stockées à la position 74 dans la table de hachage.

insérer des données dans une table de hachage

Il faut utiliser **la fonction de hachage pour générer l'index** de la position dans le tableau qui sera utilisé pour stocker les données.

Il faut stocker la clé à côté de données .

Hash key	Hashing function	Hash value	
A5RD	Apply hashing function	Hash value = 74	

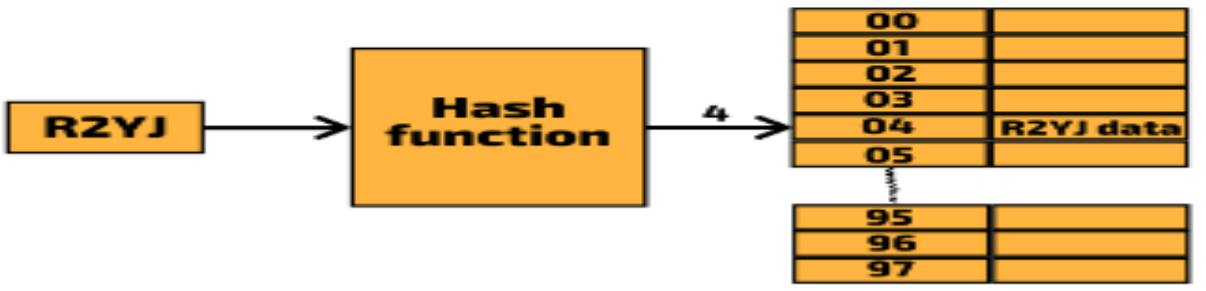


récupérer une valeur à partir d’une table de hachage

la fonction de hachage est appliquée à la clé afin de générer l’index de la position dans le tableau

c’est là que se trouveront les données associées à cette clé

Key	Hash function	Hash value	
R2YJ	Apply hash function	Hash value = 4	

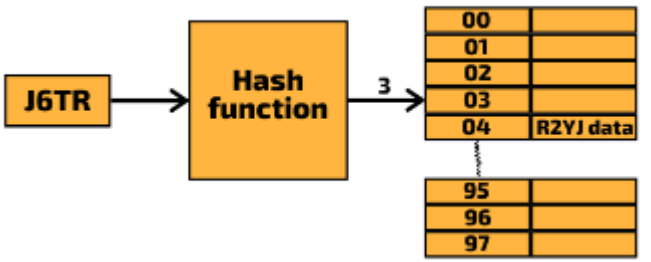


Recherche d’un élément qui n’existe pas

la fonction de hachage est appliquée à la clé afin de générer l’index de la position dans le tableau

Hash key	Hash function	Hash value	
J6TR	Apply hashing function	Hash value = 3	

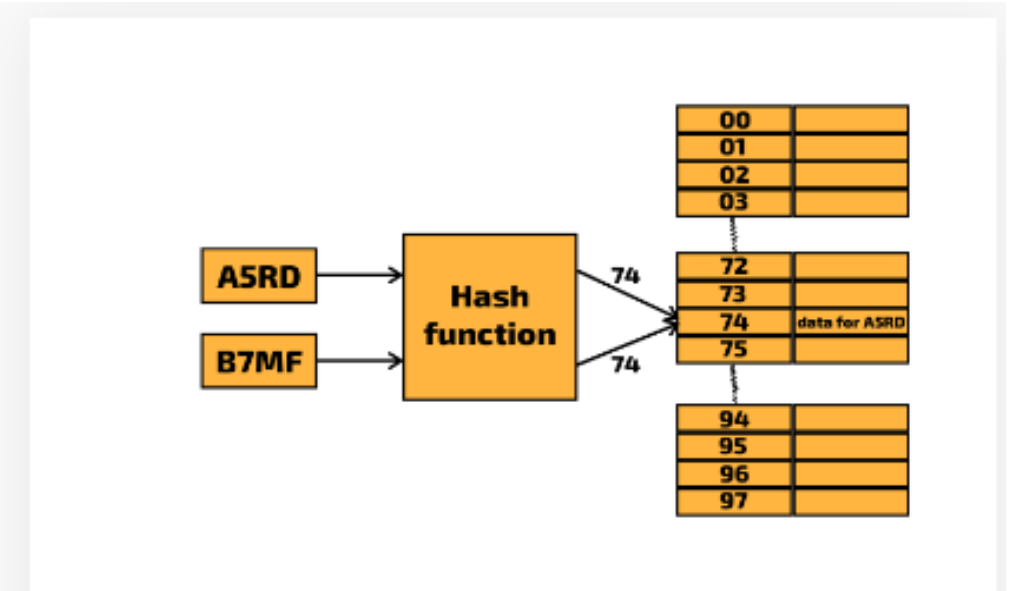
S’il n’y a pas de données à la position 3 de la table de hachage, vous savez qu’elles n’existent pas ;
Vous n’avez pas besoin de chercher dans l’ensemble du tableau pour le prouver.



collision

Une collision se produit lorsque la fonction de hachage produit la même valeur de hachage pour deux clés ou plus

Hash key	Hash function	Hash value	
J6TR	Apply hashing function	Hash value = 3	



Les problème de collision

1) vous ne pouvez pas stocker deux ensembles de données au même endroit. Vous ne pouvez pas recupérer des information

2) Quelle que soit la qualité de votre fonction de hachage, vous devrez probablement faire face à des collisions.

Résolution des collisions

Hachage ouvert : avec listes, triées ou non.

Hachage fermé : linéaire, quadratique, aléatoire, uniforme, double, ...

Le "hachage ouvert" (open hashing) ou adressage ouvert (open addressing)

- une technique de résolution de collisions utilisée dans les tables de hachage

Gestion des Collisions :

- lorsqu'une collision se produit, les éléments en collision sont stockés au même indice, formant une liste chaînée ou une autre structure de données à cet emplacement.

hachage fermé : les collisions sont résolues en stockant les éléments en collision à un emplacement différent

Opération de Recherche :

Pendant une opération de recherche, la table de hachage cherche l'éléme désiré dans la liste chaînée à l'indice correspondant.

Suppression :

La suppression implique la recherche de l'élément et sa suppression de la liste chaînée.

Hachage linéaire (linear probing)

Le sondage linéeaire

Re-hachage : $h_i(x) = (h(x) + i) \bmod B$

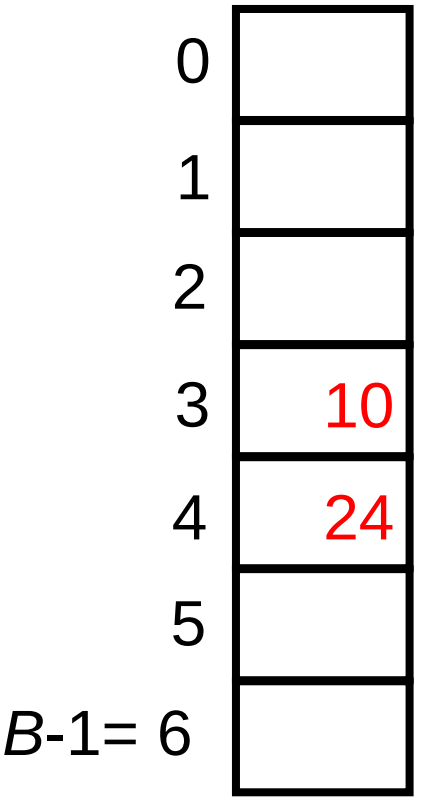
B = la taille de table de hachage

Si $h_0 = (h(x) + 0) \bmod B$ est plein on essaie h_1
Si $h_1 = (h(x) + 1) \bmod B$ est plein on essaie h_2
Etc

Clés : 10, 24, 12, 17

Pour chauque slot on a trois état : vide, plein, deleted

On insere des élément
insert(10)
$h_0(10) = (10 \bmod 7)=3$
insert(24)
$h_0(24) = (24 \bmod 7)=3$ (collosion)
$h_1(24) = ((3 +1) \bmod 7)=4$



Hachage linéaire (linear probing)

Re-hachage : $h_i(x) = (h(x) + i) \bmod B$

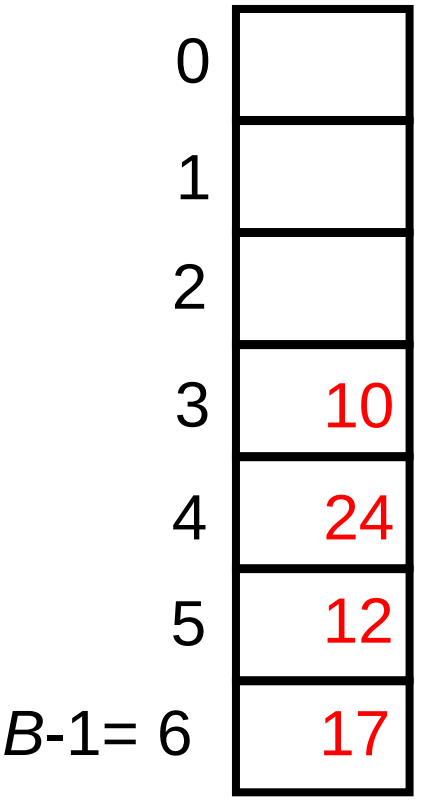
B = la taille de table de hachage

Si $h_0 = (h(x) + 0) \bmod B$ est plein on essaie h_1
Si $h_1 = (h(x) + 1) \bmod B$ est plein on essaie h_2
Etc

Clés : 10, 24, 12, 17

Pour chauque slot on a trois état : vide, plein, deleted

On insere des élément
insert(12)
$h_0(12) = (12 \bmod 7) = 5$
insert(17)
$h_0(17) = (17 \bmod 7) = 3$(collosion)
$h_1(17) = ((3 + 1) \bmod 7) = 4$ (collosion)
$h_2(17) = ((3 + 2) \bmod 7) = 5$ (collosion)
$h_3(17) = ((3 + 3) \bmod 7) = 6$



Hachage quadratique (quadratic probing)

Le sondage quadratique

Re-hachage : $h_i(x) = (h(x) + i^2) \bmod B$

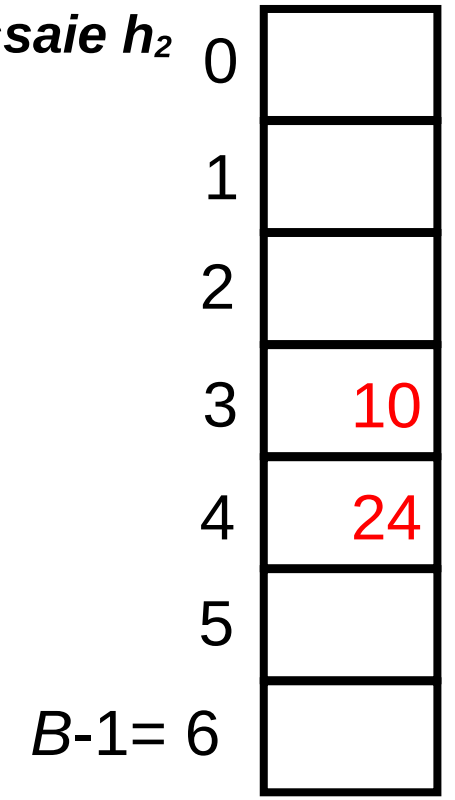
B = la taille de table de hachage

Si $h_0 = (h(x) + 0) \bmod B$ est plein on essaie h_1
Si $h_1 = (h(x) + 1) \bmod B$ est plein on essaie h_2
Si $h_1 = (h(x) + 4) \bmod B$ est plein on essaie h_2

Clés : 10, 24, 12, 17

Pour chauque slot on a trois état : vide, plein, deleted

On insere des élément
insert(10)
$h_0(10) = (10 \bmod 7) = 3$
insert(24)
$h_0(24) = (24 \bmod 7) = 3$ (collosion)
$h_1(24) = ((3 + 1) \bmod 7) = 4$



Hachage quadratique (quadratic probing)

Re-hachage : $h_i(x) = (h(x) + i) \bmod B$

B = la taille de table de hachage

Si $h_0 = (h(x) + 0) \bmod B$ est plein on essaie h_1
Si $h_1 = (h(x) + 1) \bmod B$ est plein on essaie h_2
Etc

Clés : 10, 24, 12, 17

Pour chauque slot on a trois état : vide, plein, deleted

On insere des élément
insert(12)
$h_0(12) = (12 \bmod 7) = 5$
insert(17)
$h_0(17) = (17 \bmod 7) = 3$ (collosion)
$h_1(17) = ((3 + 1) \bmod 7) = 4$ (collosion)
$h_2(17) = ((3 + 2^2) \bmod 7) = 0$

0	17
1	
2	
3	10
4	24
5	12
B-1= 6	

Le double hachage :

utilise l’idée d’appliquer une deuxième fonction de hachage à la clé lorsqu’une collision se produit dans une table de hachage.

Hachage double (double hashing)

Re-hachage : $h_i(x) = (h(x) + i*k(x)) \bmod B$

B = la taille de table de hachage

h et k sont deux fonction de hachage différentes

Si $h_0 = (h(x) + 0*k(x)) \bmod B$ est plein on essaie h_1

Si $h_1 = (h(x) + 1*k(x)) \bmod B$ est plein on essaie h_2

Si $h_1 = (h(x) + 2*k(x)) \bmod B$ est plein on essaie h_2

Clés : 10, 24, 12, 17

k(x)=3−(x mod 3)

On insere des élément

insert(10)

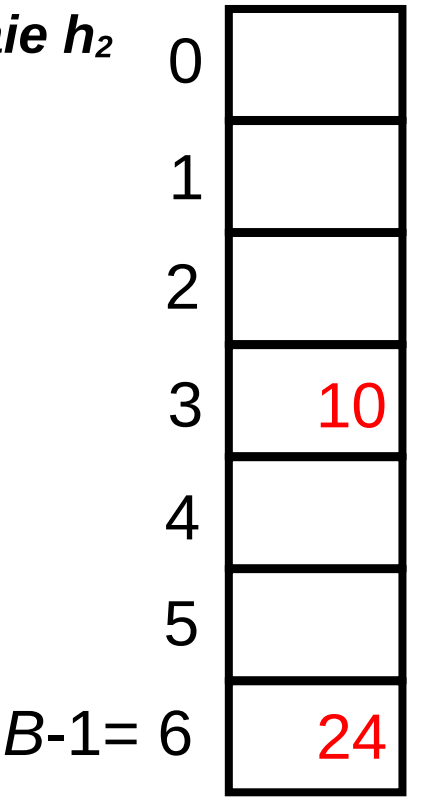
$h_0(10) = (10 \bmod 7)=3$

insert(24)

$h_0(24) = (24 \bmod 7)=3$ (collosion)

$k(24)=3-(24 \bmod 3)=3-0=3$

$h_1(24) = ((3 +1*3) \bmod 7)=6$



Hachage double (double hashing)

Re-hachage : $h_i(x) = (h(x) + i*k(x)) \bmod B$

B = la taille de table de hachage

h et k sont deux fonction de hachage différentes

Si $h_0 = (h(x) + 0*k(x)) \bmod B$ est plein on essaie h_1

Si $h_1 = (h(x) + 1*k(x)) \bmod B$ est plein on essaie h_2

Si $h_1 = (h(x) + 2*k(x)) \bmod B$ est plein on essaie h_2

Clés : 10, 24, 12, 17

k(x)=3−(x mod 3)

On insere des élément

insert(12)

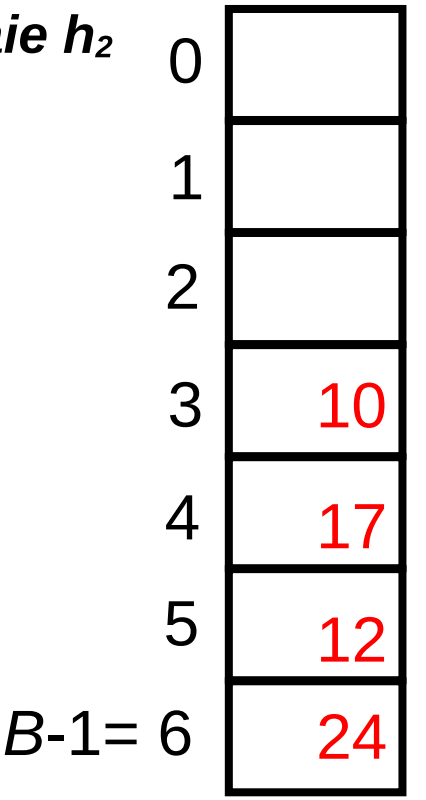
$h_0(12) = (12 \bmod 7) = 5$

insert(17)

$h_0(17) = (17 \bmod 7) = 3$ (collosion)

$k(17) = 3-(17 \bmod 3) = 3-2 = 1$

$h_1(24) = ((3 + 1*1) \bmod 7) = 4$



on explique l'algorithme seulement pour le hachage linéaire

```
const        B = { la taille de la table de hachage} ;
             vide =  {représente les positions vides dans la table   } ;
        disponible = {représente les positions disponible dans la table   } ;
                     {mais de même type        } ;
type  dictionnaire = array  [0… B-1]  of  éléments ;

fonction  VIDER  () : dictionnaire ; //  initialise une table de hachage
début
        pour  i ← 0  à  B-1  faire
                A  [ i] ←  vide ;
        retour A ;
fin

fonction  POSITION (x  élément,  A  dictionnaire) : indice ;
/* calcule la seule position possible où ajouter x dans A (une méthode général)*/
début  i ← 0 ;
  tantque  (i < B)  et  A [h_i (x)] ≠ x  et  A [h_i (x)] ≠ vide  et  A [h_i (x)] ≠ disponible
        faire  i ← i + 1 ;
  retour (h_i (x)) ;
fin
```

```
fonction  POSITIONE (x élément,  A  dictionnaire) : indice ;
//calcule la position où ajouter un élément x dans le dictionnaire A (méthode linéaire)
début  hi ← h (x) ; dernier ← (hi + B-1)  mod  B ;
        tantque  hi ≠ dernier  et   (A [hi] ∉ {x, vide})  faire
                hi ← (hi + 1)  mod  B  ;
        retour  (hi)
fin
```

```
fonction  ELEMENT (x  élément,  A  dictionnaire) : boolean ;
//Cette fonction vérifie si un élément x est présent dans le dictionnaire A
début
        si  A [POSITIONE (x, A)] = x  retour  vrai ;
        sinon  retour  faux ;
fin
```

```
fonction ENLEVER (x  élément,  A  dictionnaire) : dictionnaire ;
// Cette fonction enlève l'élément x du dictionnaire A
début
        i ← POSITIONE (x, A) ;
        si A [ i ] =  x  alors  A [ i ] ←  disponible ;
        retour A ;
fin
```

```
fonction  AJOUTER  (x  élément,  A dictionnaire) : dictionnaire ;
 // ajoute un élément x dans le dictionnaire A
début
        i ← POSITIONE(x, A) ;
        si  A [ i ] ∈ {vide, disponible}  alors
                A [ i ] ← x ;
                retour A ;
        sinon si A [ i ]  ≠ x  alors
                erreur ( " table pleine " ) ;
fin
```