Table et fonctions de hachage

•

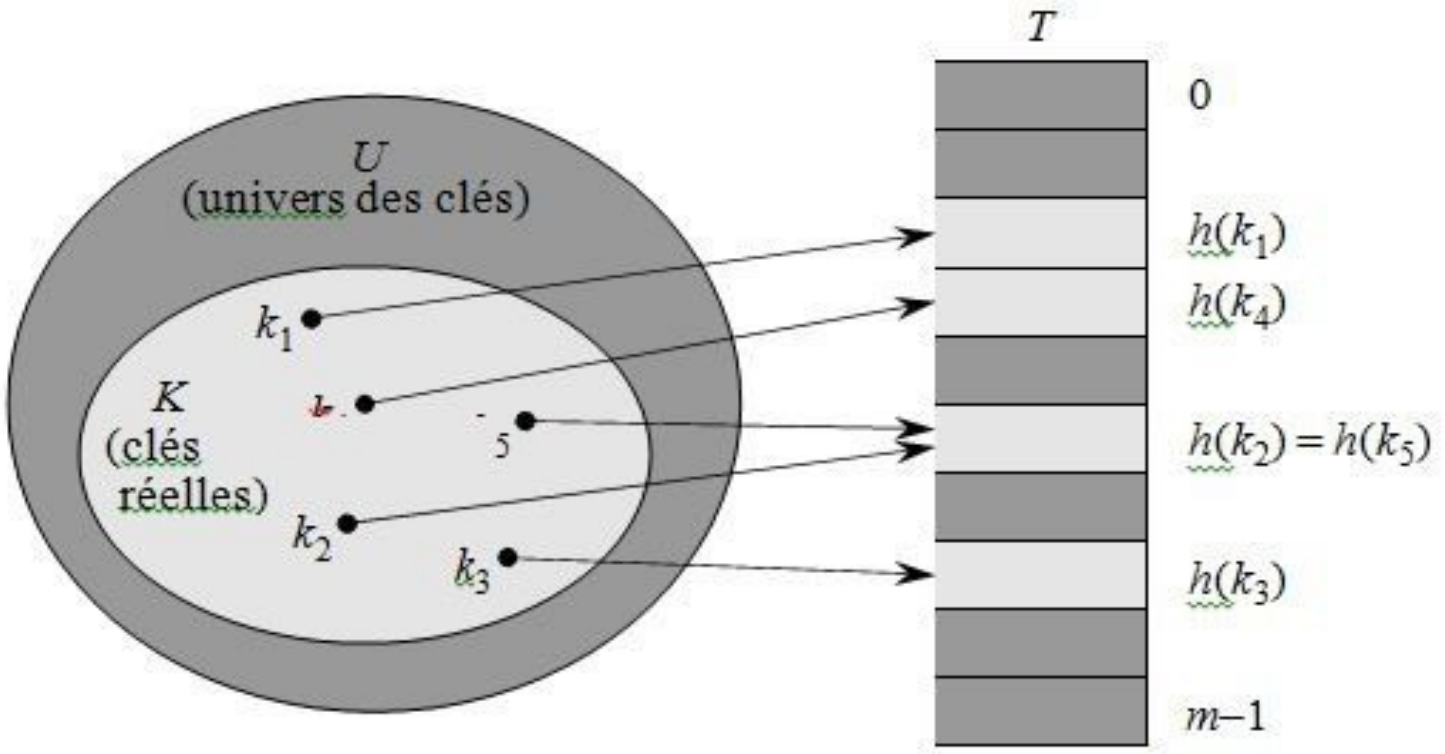


Table et fonctions de hachage

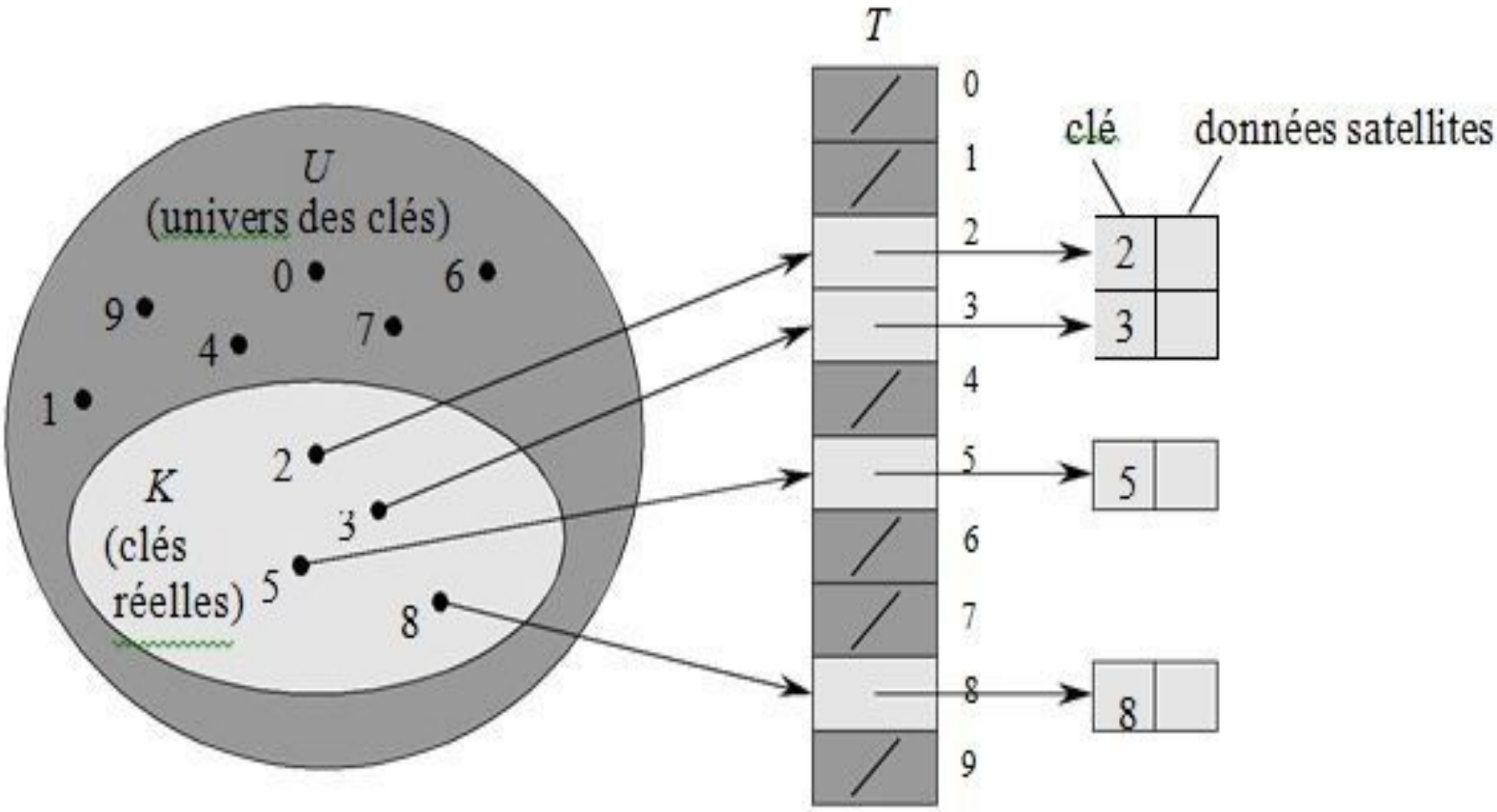


Table et fonctions de hachage

- Introduction
- Notion de clé
- Fonction de hachage
- Collisions
- Exemple de fonction de hachage
- Exemple de table de hachage

Recherche dichotomique

```
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 ← d + ((x-table[d])/(table[f])-table[d]))*(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)

HACHAGE

Type abstrait « dictionnaire »

Ensembles avec les opérations principales

ELEMENT (x, A)

AJOUTER (x, A) temps constant en moyenne

ENLEVER (x, A)

Implémentation non adaptée au classement.

Table de hachage

table dont les indices sont dans [0 .. B-1]

Fonction de hachage

h : éléments → [0 .. B-1] non injective en général

Résolution des collisions

Hachage ouvert : avec listes, triées ou non.

Hachage fermé : linéaire, quadratique, aléatoire,

uniforme, double, ...

Tables de hachage : accès à une valeur appartenant à un ensemble à partir d'une chaîne de caractères.

Ex : t["pierre"]=36 ;

On ne pourra pas écrire sous cette forme, on utilisera une fonction permettant de récupérer la donnée à partir de la chaîne de caractères :

Accéder Donnee(t,"pierre") ;

t sera toujours un tableau : il faut définir comment trouver l'indice de la donnée à partir de la chaîne de caractères...

On s'intéresse à des structures de données donnant un accès aussi direct que possible au contenu. Une donnée est retrouvée à partir d'une clé.

La clé permet de retrouver la donnée dans le tableau.

Tableau à adressage direct : la clé est l'indice dans le tableau.

→ meilleure solution, accès en O(1)

Rappel :

- Une application est surjective si
– tout élément de F possède au moins un antécédent
- Une application est injective si
– tout élément de F possède au plus un antécédent
- Une application est bijective si
– tout élément de F possède exactement un antécédent

Objectif : trouver une structure de données et les fonctions associées permettant de stocker des données avec une clé quelconque.

Contexte : l'intervalle des valeurs possibles pour les clés est beaucoup plus grand que le nombre de valeurs que l'on souhaite stocker.

Conclusion : on est prêt à renoncer à la bijection emplacement-clé.

Afin de réduire la taille du tableau, on accepte que plusieurs clés différentes pointent sur le même emplacement.

Fonction de correspondance :

h(k) et T[0, taille − 1]

h() est la fonction de correspondance.

k est la clé utilisée,

taille est la taille de la table de hachage.

Principe : associer une valeur représentant un indice dans un tableau à une valeur de type prédéfini quelconque (chaîne de caractères, par exemple).

Objectif : restreindre l'espace des valeurs possibles. Une seule case de la table correspond à plusieurs clés...

Table de Hachage

- Structure de données reposant sur des tableaux
  - Structure de données statique ??? (voir TP)
  - Algorithme de recherche très performant
- Comment ?
  - La position de l'élément dans le tableau est fonction de l'élément lui-même

Notion de clé

- Définition :

– Une clé est une partie d'un élément qui permet de désigner le contenu de cet élément de manière non ambiguë

- Exemples :

– élément : étudiant (nom, prénom, …)

- Clé : numéro d'étudiant

– élément : abonné téléphonique

- Clé : numéro de téléphone

Fonction de hachage

- But :

– Ranger les N éléments dans un tableau de taille M afin d'optimiser la recherche d'un élément donné

- Connaissant la clé d'un élément du tableau, on cherche

– un algorithme efficace pour trouver l'élément dans le tableau (de l'ordre de O(1))

Fonction de hachage

- Définition : Soit E l'ensemble des clés possibles, et F l'ensemble des indices du tableau
- Une fonction de hachage H est une fonction qui associe à toute clé K, un indice dans le tableau

– H : E → F

– H(K) = i

Fonction de hachage

- Exemple : Annuaire inversé
  - Classe Abonné :
  - Attributs :
    - string nom
    - string prenom
    - long numeroTel

Fonction de hachage

- Soit un abonné Ai de clé Ki
- La position de Ai dans le tableau sera la valeur de H(Ki)

→tab[H(Ki)] = Ai

- Rappel :
  - Ai : l'élément
  - Ki : sa clé
  - H : la fonction de hachage
  - tab : le tableau contenant les éléments

Fonction de hachage

	Indice	
H(0381111144) = 0	0	Pierre Durand - 03.81.11.11.44
	1	
H(0381333333) = 2	2	Paul Dupond - 03.81.33.33.33
H(0381222222) = 2	3	Yvette Bon 03.81.22.22.22
	4	
	5	
H(0381123456) = 6	6	Guillaume Dupont 03.81.12.34.56

Fonction de hachage

- La recherche dans une telle table est immédiate
- Connaissant K (le numéro de téléphone), l'indice dans le tableau est donné par H(K)

- En pratique, il est difficile de trouver une 'bonne' fonction de hachage

Fonction de hachage

- Une fonction de hachage doit être injective

sinon on a H(K1) = K(K2) = i

C-a-d 2 éléments sont stockés au même indice

→ Quel élément placer à l'indice i ?

→ Que faire de l'autre élément ?

Fonction de hachage

- En pratique :

- Il est souvent difficile de trouver une fonction injective

- Dans certains cas, une telle fonction n'existe pas (M < N)

- Quand elle existe, elle est alors parfois complexe et le calcul H(K) peut être coûteux

Collisions

- En pratique, on utilise souvent des fonctions non injectives
- Conséquences :

– On a H(K1) = H(K2) = i

– 2 clés différentes donnent le même indice dans le tableau. On dit qu'il y a collision

Collisions

	Indice	
H(0381111144) = 0	0	Pierre Durand - 03.81.11.11.44
	1	
H(0381333333) = 2 H(0381222222) = 2	2	Paul Dupond - 03.81.33.33.33 Yvette Bon 03.81.22.22.22
	3	
	4	
	5	
H(0381123456) = 6	6	Guillaume Dupont 03.81.12.34.56

Traitement des collisions

- 1ère méthode : chaînage externe

– Déclarer un tableau de pointeurs (au lieu du tableau d'éléments)

– tab[i] contiendra la liste des éléments dont les clés K ont la même image par H

Traitement des collisions

- Soient K1, K2, .. Kj les clés des éléments E1, E2, …Ej telles que

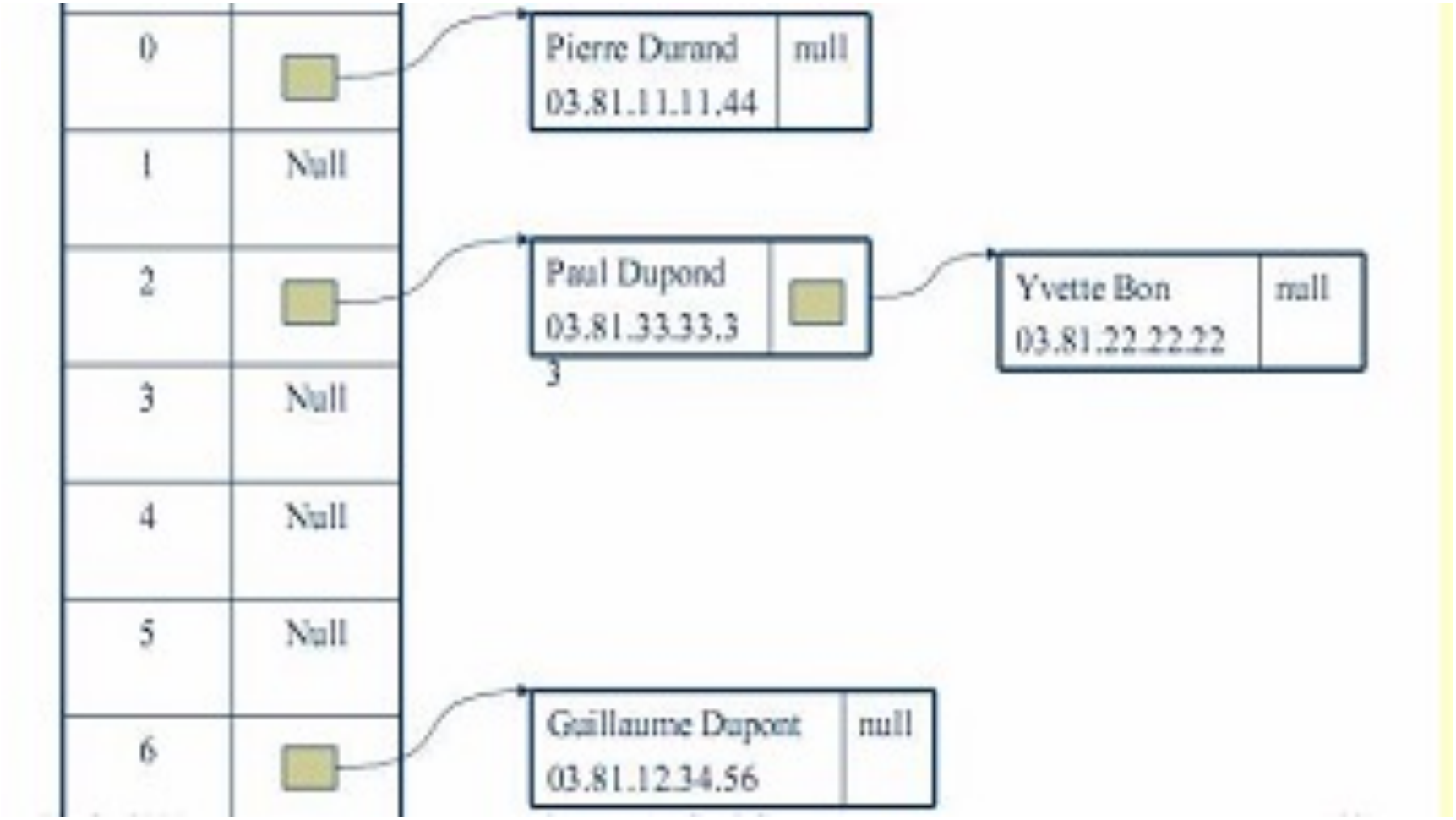
$$H(K1) = H(K2) = \ldots = H(Kj) = i$$

- Alors les éléments E1, E2, … Ej seront chaînés à partir de tab[i]

Traitement des Collisions

	Indice	
H(0381111144) = 0	0	Pierre Durand - 03.81.11.11.44
	1	
H(0381333333) = 2 H(0381222222) = 2	2	Paul Dupond - 03.81.33.33.33 Yvette Bon 03.81.22.22.22
	3	
	4	
	5	
H(0381123456) = 6	6	Guillaume Dupont 03.81.12.34.56

Traitement des collisions



Traitement des collisions

- Avantages de cette méthode :
  - Un seul tableau (de pointeurs)
- Inconvénients :
  - liste chaînée
  - la recherche d'un élément n'est plus immédiate
  - (voir TP)

Traitement des collisions
2ème méthode

- Agir sur le tableau de collisions

– Augmenter la taille du tableau tab : M’ > M Les emplacements de M à M’ serviront à stocker les éléments en collisions

– Créer un tableau supplémentaire (col) en parallèle du tableau tab pour permettre de gérer les collisions

Traitement des collisions

- Si 2 éléments E1 et E2 sont en collisions
  - (H(E1)=H(E2)=i)
- Alors
  - tab[i] = E1 et col[i] = i'
  - i' est l'indice tel que tab[i'] = E2
  - Avec i' > M
- Sinon tab[i] = E1 et col[i] = -1

H(0381333333) = 2

H(0381222222) = 2

Indice	tab
0	Pierre Durand - 03.81.11.11.44
1	
2	Paul Dupond – 03.81.33.33.33 Yvette Bon – 03.81.22.22.22
3	
4	
5	
6	Guillaume Dupond – 03.81.12.34.56

N = 4

M = 7

M' = 8

Indice	tab	col
0	Pierre Durand – 03.81.11.11.44	-1
1		
2	Paul Dupond – 03.81.33.33.33	7
3		
4		
5		
6	Guillaume Dupont – 03.81.12.34.56	-1
7	Yvette Bon – 03.81.22.22.22	-1

hachage linéaire

- Principe : en cas de collision, on incrémente de 1 (modulo taille)
- l'indice trouvé par la fonction de hachage.

Si la case correspondante est occupée, on continue jusqu'`a tomber sur une case vide.

METHODE DE HACHAGE LINEAIRE

- On suppose que K est un nombre entier
- Prendre comme indice dans le tableau le reste de la division de la clé (qui doit être un entier !) par la taille du tableau H(K) = K mod M
- Le choix de M est alors primordial pour éviter un trop grand nombre de collisions (M : nombre premier, …)

Exemple de fonction de hachage

Annuaire inversé :

On a 500 000 abonnés à ranger dans une table de taille 1 000 003 (nombre premier)

H(K) = K mod 1 000 003

H(03 81 12 34 56) = 122 313

- tab[122313] = « Guillaume Dupond …»

H(03 81 22 22 22) = 221 079

- tab[221079] = « Yvette Bon …»

Exemple de table de hachage: Dictionnaire

Rappel: Le but est de savoir si un mot est présent dans un dictionnaire et de le trouver rapidement

Ex 1 : dictionnaire de français :

Élément : mot + définition

Clé : mot

(voir(TP)

Structure de donnée C pour stocker une valeur

```
typedef struct
_donneeTableDeHash *PDonneeTableDeHash ;

typedef struct _donneeTableDeHash {
char * cle ;
char * data ;
} DonneeTableDeHash ;
DonneeTableDeHash tableau[taille] ;
```

Ajouter une entrée dans la table :

```
PDonneeTableDeHash c=tableau+h(cle,taille);
while (c->cle!=NULL) {
  if (!strcmp(cle,c->cle)) {
    printf("Cle deja utilisee: %s\n",cle);
    return;
  }
  printf("Collision lors de l'ecriture");
  if (c-tableau == taille-1)
    c=tableau;
  else
    c++;
}

/* c pointe sur une case vide... */
```

Récupérer une entrée dans la table :

```
PDonneeTableDeHash c=tableau+h(cle,taille);
if (c->cle==NULL) {
  printf("Donnee absente, cle : %s\n",cle);
  return NULL;
}
while (strcmp(c->cle,cle)) {
  if (c-t.tableau == t.taille)
    c=t.tableau;
  else
    c++;
  if (c->cle==NULL) {
    printf("Donnée absente, clé : %s\n",cle);
    return NULL;
  }
}
/* c pointe sur la donnee recherchee */
```

Exemple de table de hachage :

compilateur :

Élément : variable

Clé : nom de variable

On souhaite détecter les erreurs suivantes :

- déclaration d’une variable déjà déclarée
- utilisation d’une variable non déclarée

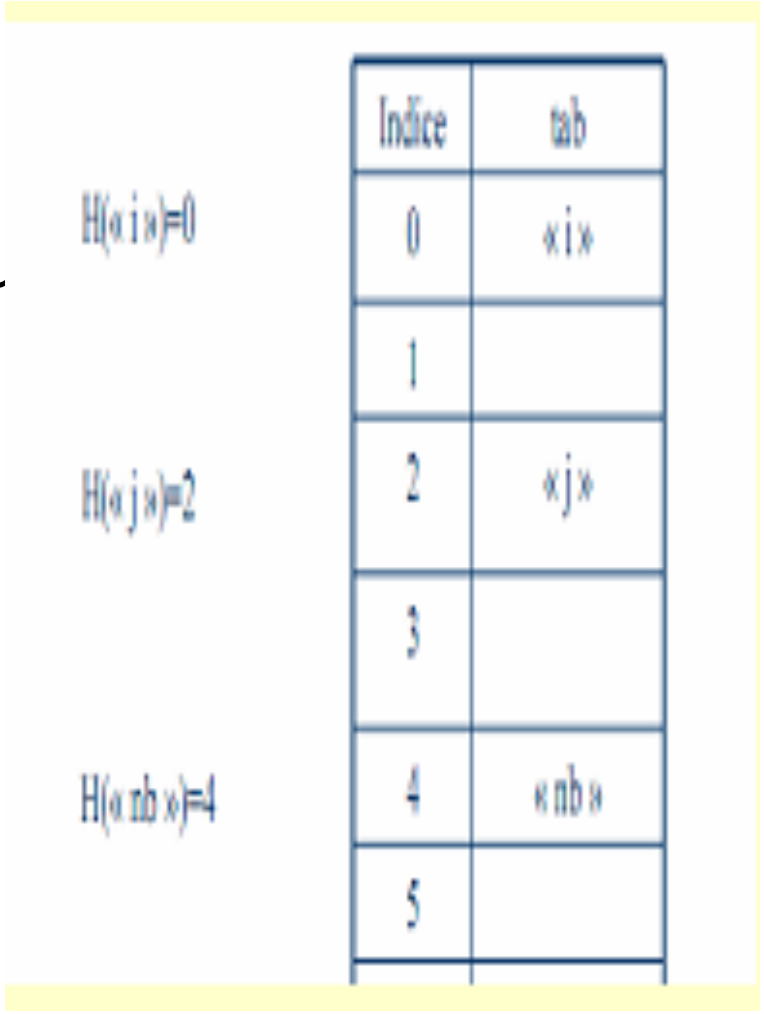
Exemple de table de hachage

Exemple :

```
int i, j, nb;
i=j;
nb=x;   // x : non déclaré
int j;    // j : déjà déclarée
```

1ère étape :

à chaque déclaration, on range la variable déclarée dans le tableau



```
int i, j, nb;
i=j;
nb=x;   // x : non déclaré
int j;    // j : déjà déclarée
```

Tab[H(« j »)] est déjà occupée : j est déjà déclaré !

- Détection des erreurs « variable déjà déclarée »

2ème étape

à chaque instruction, on vérifie que la variable est dans le tableau

int i, j, nb;

i=j;

nb=x; // x : non déclaré

int j; // j : déjà déclarée

On vérifie que tab[H(« i »)] et tab[H(« j »)]

sont occupés

```
int i, j, nb;
i=j;
nb=x;   // x : non déclaré
int j;    // j : déjà déclarée
```

Tab[H(«x »)] est vide : x est non déclaré !

- Détection des erreurs « variable non déclarée »

- En utilisant les tables de hachage, on est capable de détecter des erreurs de compilation (erreur de déclaration) très efficacement.

- Note : Avec d'autres structures (listes, arbres, …), l'algorithme de recherche d'un élément a une complexité au mieux en log(N)

- Ici, c'est immédiat (sous réserve que l'on trouve une « bonne » fonction de hachage)

Gestion des collisions : hachage linéaire

- Problème : “Clustering”
- en cas de conflits fréquents, certaines zones vont se remplir plus vite que d’autres...
- Plus il y a de conflits pour une même case, plus il sera long de trouver un espace disponible. (Voir TP)

Problème :

Si on souhaite utiliser la méthode linéaire

« de la division » comme fonction de hachage, il faut une clé numérique (au lieu d'une chaîne de caractères)

Question : comment passer d'une clé «chaîne de caractère » à une clé numérique ?

Soit L la longueur de la chaîne « nom »

Soit ascii(nom[i]) le code ascii du (i+1) éme caractère de nom.

On peut choisir :

H(nom) = $\sum_{i=0}^{L-1}(ascii(nom[i] \times t^i)$

Avec t bien choisi (ex: le nombre de caractères différents)

Il s’agit d’écrire le « nom » dans un système à base 256

Ainsi tout « nom » a une valeur différente dans cette base

ascii(i) = 105 ascii(j) = 106

ascii(n) = 110 ascii(b) = 98

En prenant t=256:

$H(«i») = 105 * 256^0 = 105$

$H(«j») = 106 * 256^0 = 106$

$H(«nb») = 110* 256^0 + 98*256^1 = 110+25088 = 25198$

$H(«bn») = 98* 256^0 + 110*256^1 = 98+28160 = 28258$

Gestion des collisions : Chaînage

Exemple : Table de hachage avec chaînage.

Structure de donnée correspondante ?

```
typedef struct donneeTableDeHash
*PDonneeTableDeHash ;
typedef struct donneeTableDeHash {
char * cle ;
char * data ;
PDonneeTableDeHash suivant ;
} DonneeTableDeHash ;
```

Ajout d'un élément dans la table :

```
PDonneeTableDeHash c=t a b l e a u+h ( c l e , t . t a i l l e ) ;
i f ( c−>c l e !=NULL) {
wh i l e ( c−>s u i v a n t !=NULL) {
i f ( ! s t rcmp ( c l e , c−>c l e ) ) {
p r i n t f ( " Cl e  d e j a  u t i l i s e e : %s \n" , c l e ) ;
r e t u r n ;
}
c=c−>s u i v a n t ;
}
c−>s u i v a n t=ma l l o c ( s i z e o f ( Donnee Table DeHash ) ) ;
c=c−>s u i v a n t ;
}
/∗ c p o i n t e s u r l ' e l ement a mo d i f i e r ∗/
```

Accéder à une donnée :

```
PDonneeTableDeHash c=t a b l e a u + h ( c l e , t . t a i l l e ) ;
i f ( c−>c l e==NULL) {
p r i n t f ( ”Donnee abs ent e , c l e : %s \n” , c l e ) ;
r e t u r n NULL ;
}
wh i l e ( s t rcmp ( c−>c l e , c l e ) ) {
c=c−>s u i v a n t ;

i f ( c==NULL | | c−>c l e==NULL) {
p r i n t f ( ”Donnee abs ent e , c l e : %s \n” , c l e ) ;
r e t u r n NULL ;
}
}
r e t u r n c−>data ;
```

Comparaison entre les avantages et inconvénients des deux stratégies :

Type	Avantages	Inconvénients
Linéaire	simple	limite de taille, “clustering”
Chaînage	pas de limite de taille	occupation mémoire, pb d’accés

Fonction de correspondance : objectifs

→ calculer un indice à partir d'une clé.

h(k) e [0, taille − 1]

h() =?

Cas d'une clé de type chaîne de caractères

On transforme la chaîne de caractères en un entier.

Exemple : ajout des codes ASCII des caractères composant la chaîne.

“UE C avancé” : ’U’=85, ’E’=69, ’ ’=32, ’C’=67, ’ ’=32, ’a’=97, ’v’=118, ’a’=97, ’n’=110, ’c’=99, ’´e’=233 → total=1039

Remarque : en faisant cela tous les anagrammes d’un mot iront dans la même case.

Fonction de correspondance :

solution intuitive

- Si toutes les clés sont équiprobables et

repartie dans un intervalle[0, r] :

- h(k) = floor (((taille − 1) ∗ k)/r)

Fonction de correspondance : modulo

- On calcule le modulo :
  - **h(k) = k mod taille**
- taille doit être choisie convenablement.

Généralement, on choisit un nombre premier proche d'une puissance de 2, exemple : 4093.

Fonction de correspondance : produit

- h(k) =

  floor (taille $*$ (coeff $*$ k – floor (coeff $*$ k)))

- coeff est un flottant entre 0 et 1

Hachage Ouvert

Liens explicites

Taille variable

fonction h (x : mot) : indice ;

début

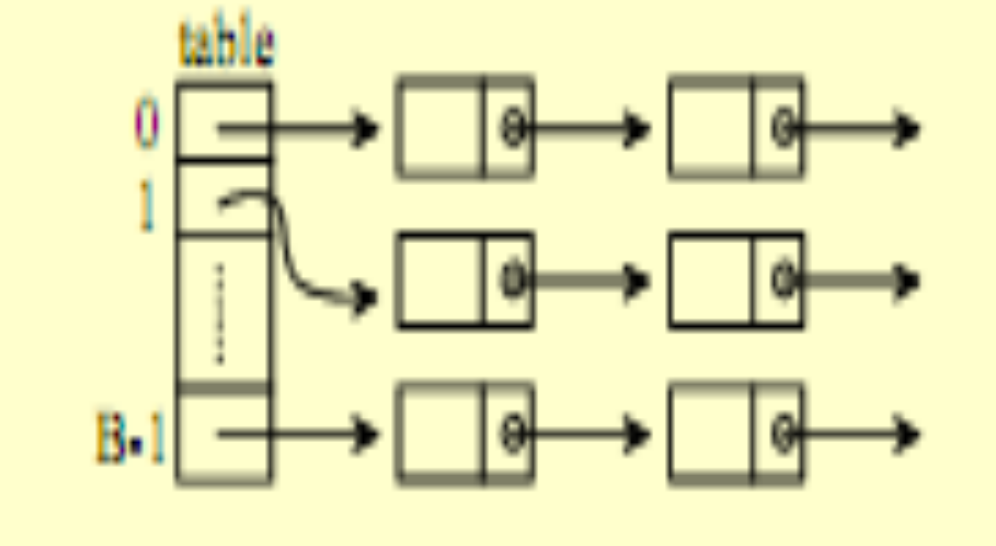
somme : = 0 ;

pour i ← 1 à longueurmaxi faire

somme ← somme + ord (x [i]) ;

retour (somme mod B) ;

fin



```
int hash (char * s) {
   char * p ;
   unsigned h = 0, g ;
   for (p = s ;  * p ;   p++) {
            h = (h<< 4) + (* p) ;
            if (g = h  &  0xf0000000) {
                     h =  h ^ (g >> 24) ;
                     h =  h ^ g ;
            }
   }
return (h % PRIME) ;
}
```

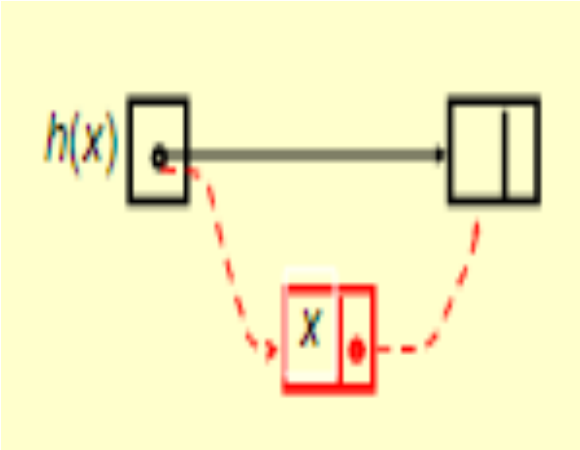
Voir Aho, Sethi, Ullman, Compilers, Addison-Wesley, 1986

```
const B = {constante ad hoc} ;
type liste = ↑ cellule ;
cellule = record
    elt : élément ;
    suivant : liste
end ;
dictionnaire = array [0 .. B-1] of liste ;

fonction  VIDER  () : dictionnaire ;
début
pour i ← 0 à B-1  faire A [ i ] ← NULL ;
retour A ;
Fin

fonction  ELEMENT (x élément,  A dictionnaire) : booléen ;
début p ←  A [ h(x) ] ;
Tant que p ≠ NULL  faire
si p->elt = x  alors retour vrai
sinon p ← p->suivant ;
retour  faux ;
fin
```

```
fonction AJOUTER ( x élément,  A dictionnaire) ;
début
AJOUTERLISTE ( x, h(x) ) ;
retour A ;
fin
```




```
fonction AJOUTER (x  élément,  A dictionnaire) ;
début
si  non ELEMENT (x, A)  alors {
i ← h (x) ;
p ←  A [ i ] ;
allouer  A [ i ]  ;
A [ i ]->elt ←  x ;
A [ i ]->suivant ←  p ;
}
retour A ;
fin
```

Temps des opérations(compléter!!)

Hachage ouvert

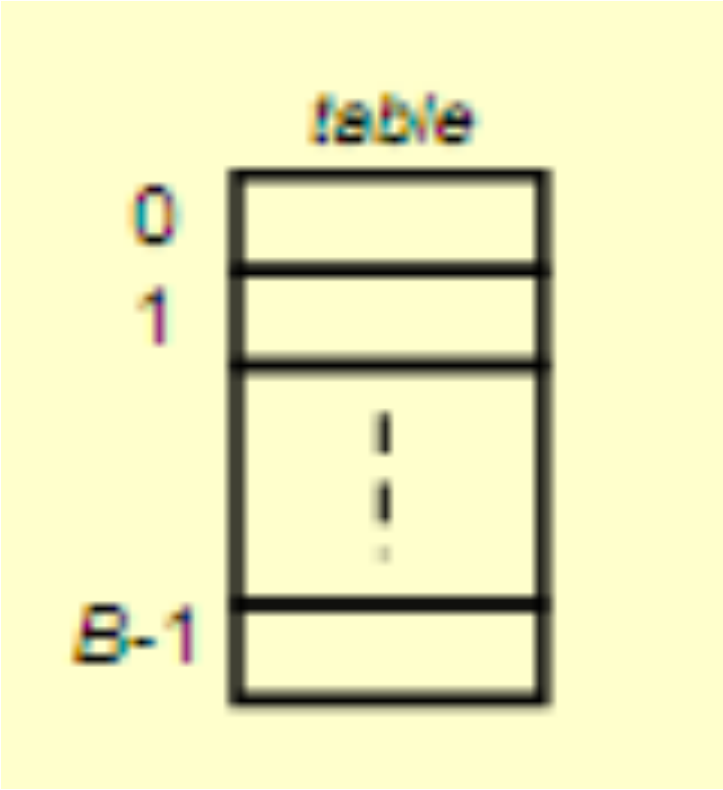
- initialisation (VIDER) : O (B)
- ajout : temps constant (après test d ’appartenance)
- appartenance
- suppression

si les événements "h (x) = i" sont équiprobables

Création d'une table de n éléments :

$$\mathbf{O(n(1+n/B))=O(1+n/B)}$$

Hachage fermé



- liens implicites : gain de place
- taille limitée
- ré-allocation en cas de débordement

Re hachage

h0 (x) = h (x), h1 (x), … hi(x), …

où hi

(x) dépend généralement de x et de i

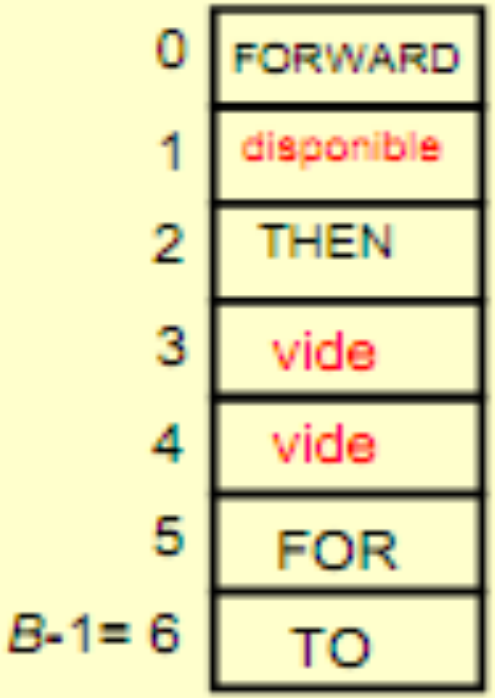
Suppression d ’un élément

distinction entre « vide » et «disponible »

facteur de charge : n / B où n = nombre d ’éléments

Hachage Linéaire

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



EXEMPLE

$h(x) = (\text{ord}(c) - \text{ord}('a')) \bmod B$
où c première lettre de x.

AJOUTER ('BEGIN')
AJOUTER ('FOR')
AJOUTER ('FUNCTION')
AJOUTER ('FORWARD')
AJOUTER ('THEN')
ENLEVER ('FUNCTION')
ENLEVER ('BEGIN')
AJOUTER ('TO')

```
const B = {constante ad hoc} ;
vide =  {constantes particulières  } ;
disponible = {distinctes des éléments   } ;
{mais de même type        } ;

type dictionnaire = array [0… B-1]  of
    éléments ;

fonction VIDER  () : dictionnaire ;
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 */

début i  ← 0 ;
tantque (i < B  et
A [hi (x)] ≠ x et A [hi(x)] ≠ vide  et
A [hi (x)] ≠ disponible)

faire i  ← i + 1 ;
retour (hi(x)) ;
fin
```

```
fonction POSITIONE (x élément,  A
    dictionnaire) : indice ;
Début

 hi  ← h (x) ;
 dernier ←  (hi + B-1)  mod B ;

Tant que hi ≠ dernier et
 (A [hi] ∉  {x, vide})  faire

hi  ← (hi + 1)  mod B ;

retour (hi)

fin
```

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

```
fonction ELEMENT (x élément,  A
    dictionnaire) : boolean ;
début
si A [POSITIONE (x, A)] = x
retour  vrai ;
sinon  retour faux ;
fin
```

```
fonction POSITION A (x
   élément,  A dictionnaire) :
   indice ;
début
hi  ← h (x) ;
dernier := (hi + B-1)  mod B;
Tant que hi ≠ dernier et
(A [hi]  ∉ {x, vide disponible})
   faire
hi  ← (hi + 1)  mod B ;
retour hi ;
fin
```

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

Hachage Quadratique

Re-hachage : hi (x) = (h (x) + i ^{2}) mod B

fonction POSITION (x élément, A dictionnaire) : indice ;
début
hi ← h (x) ; inc ← 1 ;
Tant que (inc < B) et (A [hi] ∉ {x, vide, ? }) faire
hi ← (hi + inc) mod B ;
inc ← inc + 2 ;
retour hi ;
Fin

- seule la moitié de la table est examinée par re-hachage

→ utiliser la suite :
h (x), h (x) + 1, h (x) - 1, h (x) + 4, h (x) - 4, …
avec B premier.

Hachage Double

Re-hachage :

$$\mathbf{hi(x) = (h(x) + i\ g\ (x))\ mod\ B}$$

- B premier et $1 \leq g\ (x) \leq B - 1$
- ou B premier avec chacun des g (x) pour examen de toute la table par re-hachage.

fonction POSITION (x élément, A dictionnaire) : indice ;
début
hi $\leftarrow$ h (x) ;
inc $\leftarrow$ g (x) ;
dernier $\leftarrow$ (hi + (B-1)* inc) mod B ;
tantque (hi $\neq$ dernier) et (A [hi] $\notin$ {x, vide, ? }) faire
hi $\leftarrow$ (hi + inc) mod B ;
retour hi ;
fin

Hachage aléatoire

Re-hachage : hi(x) = (h (x) + di) mod B,

d1, d2, …, dB-1 permutation aléatoire de (1,..., B-1)

Génération des di par « décalage ».

- choix de k ∈ (1,…, B-1)

- di+ 1 = {2

Exemple B = 8 k = 3 = 11_2

d1 = 5_{10} = 1012; d2 = 1_{10} = 0012

d3 = 2_{10} = 0102; d4 = 4_{10} = 1002

d5 = 310 = 0112; d6 = 6_{10} = 1102;

d7 = 7_{10} = 1112

Fin…

Rappel :

- Une application est surjective si
  – tout élément de F possède au moins un antécédent
- Une application est injective si
  – tout élément de F possède au plus un antécédent
- Une application est bijective si
  – tout élément de F possède exactement un antécédent

Tableau associatif

F tableau associatif : nombre fini d'indices de type quelconque

⇔ F fonction de domaine fini

⇔ F représentable par l'ensemble

$$E = \{ (x, F[x]) \ / \ x \in \text{domaine de } F \}$$

Opérations

- INIT (F) rendre F [x] non défini pour tout x
- DEFINIR (F, x, y) poser F [x] = y
- CALCULER (F, x) = y si F [x] = y; nul sinon

Implémentation

- représenter E par hachage sur le domaine de F