

Cours n° 9 : les tables de hachage

Christophe Gonzales



HUGO3 — Algorithmique

Problématique

structure pour effectuer rapidement :

- ▶ l'insertion
- ▶ la recherche
- ▶ la suppression

d'informations dans 1 ensemble géré dynamiquement

Problématique

structure pour effectuer rapidement :

- ▶ l'insertion
- ▶ la recherche
- ▶ la suppression

d'informations dans 1 ensemble géré dynamiquement

Exemples : dictionnaire, compilateur

Problématique

structure pour effectuer rapidement :

- ▶ l'insertion
- ▶ la recherche
- ▶ la suppression

d'informations dans 1 ensemble géré dynamiquement

Exemples : dictionnaire, compilateur

⇒ { on ne connaît pas le nombre d'éléments *a priori*
on doit accéder rapidement aux éléments

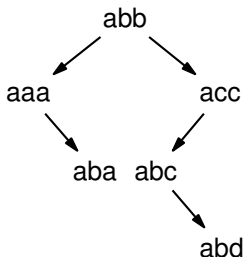
► Exemple d'un compilateur :

```
aaa = 3;  
aba = 4;  
abb = 5;  
abc = aaa + 2;  
abd = aaa + 3;  
acc = abc + abd;
```

► Exemple d'un compilateur :

```
aaa = 3;  
aba = 4;  
abb = 5;  
abc = aaa + 2;  
abd = aaa + 3;  
acc = abc + abd;
```

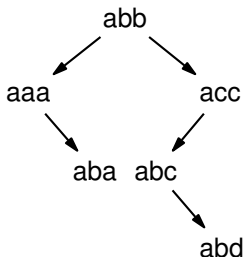
► Stockage des variables par arbre binaire :



► Exemple d'un compilateur :

```
aaa = 3;  
aba = 4;  
abb = 5;  
abc = aaa + 2;  
abd = aaa + 3;  
acc = abc + abd;
```

► Stockage des variables par arbre binaire :



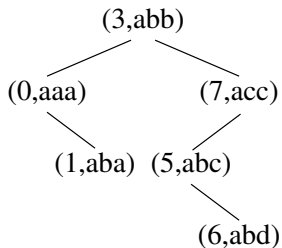
⇒ trop de comparaisons

Principe des tables de hachage

- ▶ on associe à chaque information à stocker une clé
- ▶ on recherche les informations via leur clé

Principe des tables de hachage

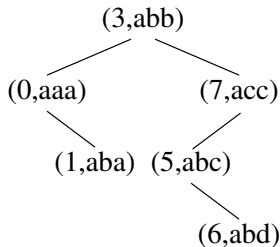
- ▶ on associe à chaque information à stocker une clé
- ▶ on recherche les informations via leur clé



Solution : table de hachage (1/3)

Principe des tables de hachage

- ▶ on associe à chaque information à stocker une clé
- ▶ on recherche les informations via leur clé



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

```
typedef struct {  
    int    cle;  
    data_t donnee;  
} bucket_t;
```

Tables à adressage direct

- ▶ U : l'univers des clés
- ▶ $U = \{0, 1, \dots, m - 1\}$

Tables à adressage direct

- ▶ U : l'univers des clés
- ▶ $U = \{0, 1, \dots, m - 1\}$
- ▶ Toutes les données ont des clés différentes

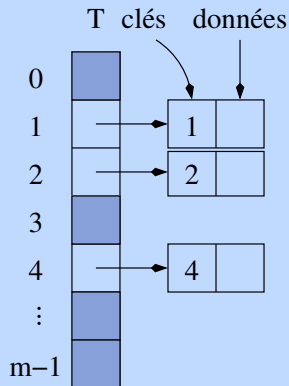
Solution : table de hachage (2/3)

Tables à adressage direct

- ▶ U : l'univers des clés
- ▶ $U = \{0, 1, \dots, m-1\}$
- ▶ Toutes les données ont des clés différentes

```
bucket_t* tab[m];
```

```
data_t search(tab, cle) {  
    if (tab[cle] == NULL)  
        throw std::exception ();  
    return tab[cle]->donnee;  
}
```



Solution : table de hachage (2/3)

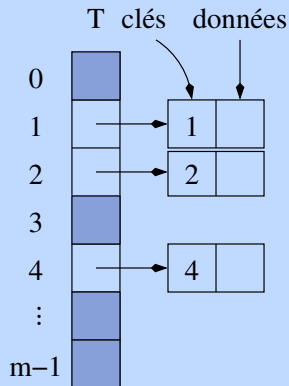
Tables à adressage direct

- ▶ U : l'univers des clés
- ▶ $U = \{0, 1, \dots, m-1\}$
- ▶ Toutes les données ont des clés différentes

```
bucket_t* tab[m];
```

```
data_t search(tab, cle) {  
    if (tab[cle] == NULL)  
        throw std::exception ();  
    return tab[cle]->donnee;  
}
```

```
void insert(tab, donnee) {  
    int cle = get_cle(donnee);  
    bucket_t* bucket = (bucket_t*)  
        malloc (sizeof(bucket_t));  
    if (bucket == NULL)  
        throw std::exception ();  
    bucket->cle = cle;  
    bucket->donnee = donnee;  
    tab[cle] = bucket;  
}
```



Problèmes des tables à adressage direct :

- ① $U \neq \{0, 1, \dots, m - 1\}$ ou, pire, U n'est pas un ensemble d'entiers positifs

Problèmes des tables à adressage direct :

- 1 $U \neq \{0, 1, \dots, m - 1\}$ ou, pire, U n'est pas un ensemble d'entiers positifs
- 2 $|U|$ est très grand \implies t trop grand pour être stocké en mémoire

Problèmes des tables à adressage direct :

- 1 $U \neq \{0, 1, \dots, m - 1\}$ ou, pire, U n'est pas un ensemble d'entiers positifs
- 2 $|U|$ est très grand \implies t trop grand pour être stocké en mémoire
- 3 plusieurs données ont la même clé

Problèmes des tables à adressage direct :

- 1 $U \neq \{0, 1, \dots, m - 1\}$ ou, pire, U n'est pas un ensemble d'entiers positifs
- 2 $|U|$ est très grand \implies t trop grand pour être stocké en mémoire
- 3 plusieurs données ont la même clé

Solutions :

- 1 créer une fonction $\text{hash} : U \mapsto \{0, 1, \dots, m - 1\}$

Problèmes des tables à adressage direct :

- 1 $U \neq \{0, 1, \dots, m - 1\}$ ou, pire, U n'est pas un ensemble d'entiers positifs
- 2 $|U|$ est très grand \implies t trop grand pour être stocké en mémoire
- 3 plusieurs données ont la même clé

Solutions :

- 1 créer une fonction $\text{hash} : U \mapsto \{0, 1, \dots, m - 1\}$
- 2 si tout l'univers des clés est utilisé, pas de solution

Problèmes des tables à adressage direct :

- 1 $U \neq \{0, 1, \dots, m - 1\}$ ou, pire, U n'est pas un ensemble d'entiers positifs
- 2 $|U|$ est très grand \implies t trop grand pour être stocké en mémoire
- 3 plusieurs données ont la même clé

Solutions :

- 1 créer une fonction $\text{hash} : U \mapsto \{0, 1, \dots, m - 1\}$
- 2 si tout l'univers des clés est utilisé, pas de solution
sinon : soit K l'ensemble des clés réellement utilisées ($|K| \ll |U|$). créer une fonction $\text{hash} : U \mapsto \{0, 1, \dots, |K| - 1\}$
 \implies table de hachage

Problèmes des tables à adressage direct :

- 1 $U \neq \{0, 1, \dots, m - 1\}$ ou, pire, U n'est pas un ensemble d'entiers positifs
- 2 $|U|$ est très grand \implies t trop grand pour être stocké en mémoire
- 3 plusieurs données ont la même clé

Solutions :

- 1 créer une fonction $\text{hash} : U \mapsto \{0, 1, \dots, m - 1\}$
- 2 si tout l'univers des clés est utilisé, pas de solution
sinon : soit K l'ensemble des clés réellement utilisées ($|K| \ll |U|$). créer une fonction $\text{hash} : U \mapsto \{0, 1, \dots, |K| - 1\}$
 \implies table de hachage
- 3 *collisions* dans une table de hachage. Solutions dans les transparents suivants

Tables de hachage (1/2)

Définition d'une table de hash

- ▶ Tableau (comme pour l'adressage direct)

Définition d'une table de hash

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

Tables de hachage (1/2)

Définition d'une table de hash

- ▶ Tableau (comme pour l'adressage direct)
- ▶ mais au lieu de stocker l'élément de clé k à l'indice k : 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 beaucoup plus grand que m .

Tables de hachage (1/2)

Définition d'une table de hash

- ▶ Tableau (comme pour l'adressage direct)
- ▶ mais au lieu de stocker l'élément de clé k à l'indice k : 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 beaucoup plus grand que m .

⇒ répond aux points 1 et 2

Définition d'une table de hash

- ▶ Tableau (comme pour l'adressage direct)
- ▶ mais au lieu de stocker l'élément de clé k à l'indice k : 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 beaucoup plus grand que m .

⇒ répond aux points 1 et 2

Exemples :

- ▶ Annuaire téléphonique :
clé = nom des personnes, donnée = numéro de téléphone

Définition d'une table de hash

- ▶ Tableau (comme pour l'adressage direct)
- ▶ mais au lieu de stocker l'élément de clé k à l'indice k : 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 beaucoup plus grand que m .

⇒ 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

Définition d'une table de hash

- ▶ Tableau (comme pour l'adressage direct)
- ▶ mais au lieu de stocker l'élément de clé k à l'indice k : 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 beaucoup plus grand que m .

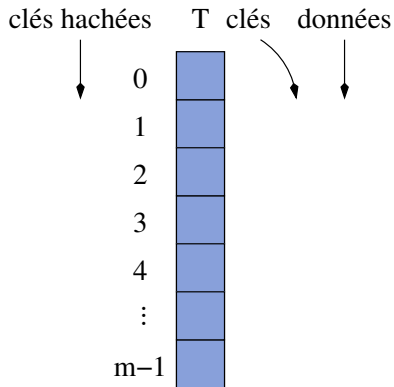
⇒ répond aux points 1 et 2

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

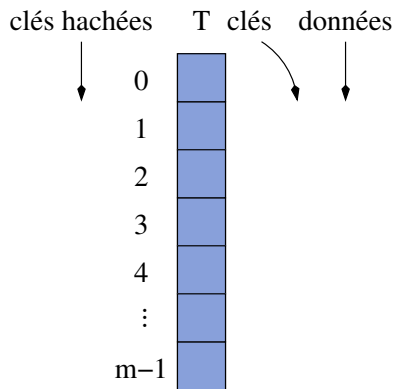
Tables de hachage (2/2)

donnée	clé	clé hachée



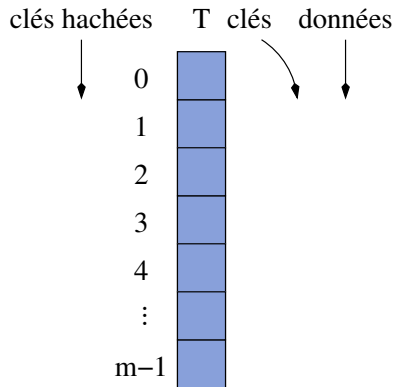
Tables de hachage (2/2)

donnée	clé	clé hachée
d_{k_1}		



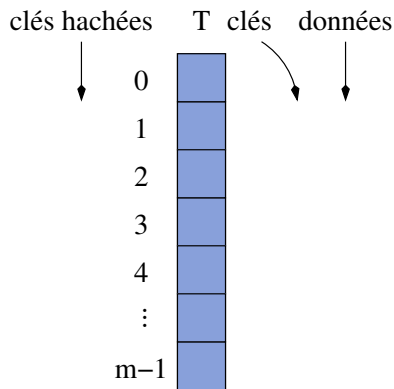
Tables de hachage (2/2)

donnée	clé	clé hachée
d_{k_1}	k_1	



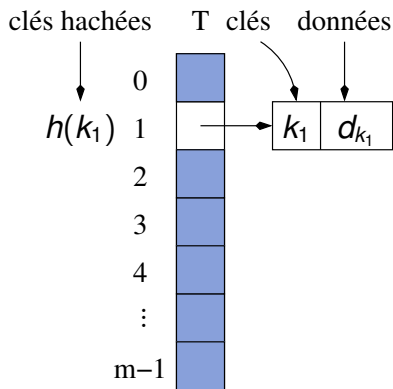
Tables de hachage (2/2)

donnée	clé	clé hachée
d_{k_1}	k_1	$h(k_1) = 1$



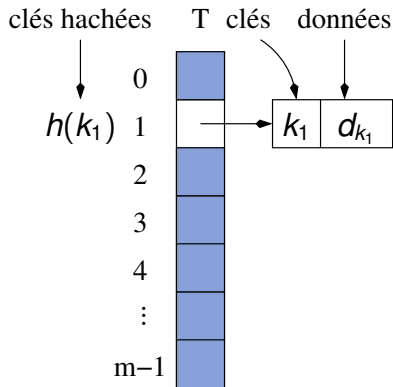
Tables de hachage (2/2)

donnée	clé	clé hachée
d_{k_1}	k_1	$h(k_1) = 1$



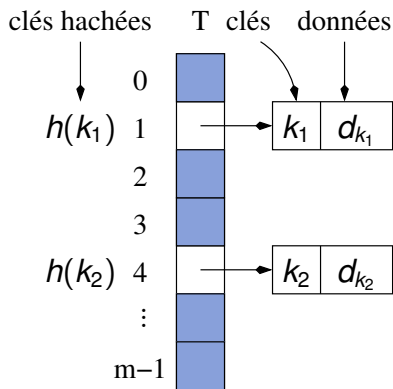
Tables de hachage (2/2)

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



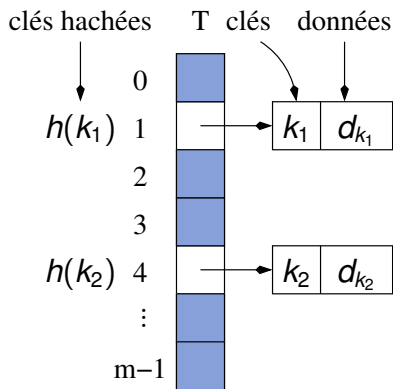
Tables de hachage (2/2)

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



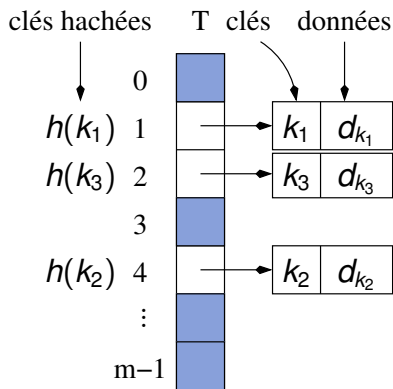
Tables de hachage (2/2)

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	$h(k_3) = 2$



Tables de hachage (2/2)

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	$h(k_3) = 2$



- ▶ **Comment choisir la fonction h ?**

- ▶ **Comment choisir la fonction h ?**
 - ▶ le calcul de $h(k)$ doit être rapide

- ▶ **Comment choisir la fonction h ?**
 - ▶ le calcul de $h(k)$ doit être rapide
 - ▶ h doit éviter au maximum les collisions

► **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 $k_1 \neq k_2$)

- ▶ **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

▶ 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

soit $h(k) = 0 \forall k \in U$

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

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

soit $h(k) = k \forall k \in U$

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

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

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

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 $|U|$ clés possibles dans une table de longueur $m \ll |U| \implies$ il n'y a pas bijection entre U et $\{0, 1, \dots, m - 1\}$

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.

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 :

► $h(k) = k \bmod 7$



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 :

► $h(k) = k \bmod 7$

► « aa » $\Rightarrow k = 5 \Rightarrow h(k) = 5$



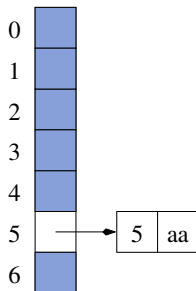
Principe

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 :

► $h(k) = k \bmod 7$

► « aa » $\Rightarrow k = 5 \Rightarrow h(k) = 5$

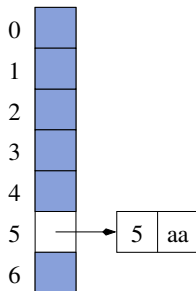


Principe

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 :

- ▶ $h(k) = k \bmod 7$
- ▶ « aa » $\Rightarrow k = 5 \Rightarrow h(k) = 5$
- ▶ « bb » $\Rightarrow k = 10 \Rightarrow h(k) = 3$



Principe

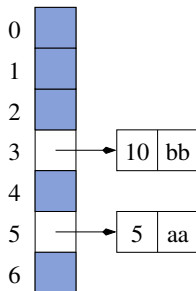
Chaque élément de la table est une liste chaînée et tous les éléments ayant la même clé hachée sont dans la même liste chaînée.

Exemple :

► $h(k) = k \bmod 7$

► « aa » $\Rightarrow k = 5 \Rightarrow h(k) = 5$

► « bb » $\Rightarrow k = 10 \Rightarrow h(k) = 3$

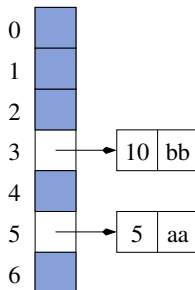


Principe

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 :

- ▶ $h(k) = k \bmod 7$
- ▶ « aa » $\Rightarrow k = 5 \Rightarrow h(k) = 5$
- ▶ « bb » $\Rightarrow k = 10 \Rightarrow h(k) = 3$
- ▶ « cc » $\Rightarrow k = 12 \Rightarrow h(k) = 5$

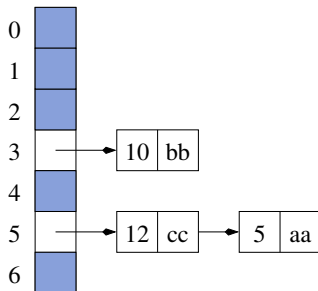


Principe

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 :

- ▶ $h(k) = k \bmod 7$
- ▶ « aa » $\Rightarrow k = 5 \Rightarrow h(k) = 5$
- ▶ « bb » $\Rightarrow k = 10 \Rightarrow h(k) = 3$
- ▶ « cc » $\Rightarrow k = 12 \Rightarrow h(k) = 5$

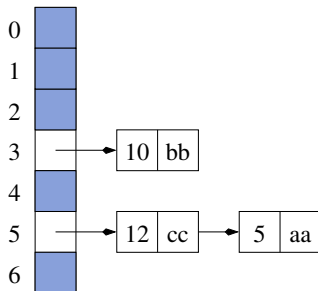


Principe

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 :

- ▶ $h(k) = k \bmod 7$
- ▶ « 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 » $\Rightarrow k = 75 \Rightarrow h(k) = 5$



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 :

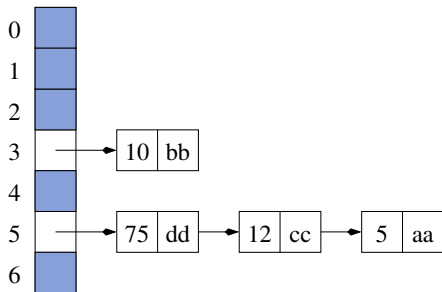
▶ $h(k) = k \bmod 7$

▶ « 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 » $\Rightarrow k = 75 \Rightarrow h(k) = 5$



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 :

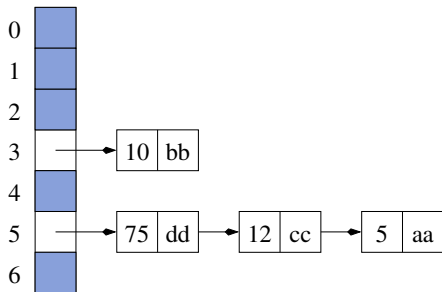
▶ $h(k) = k \bmod 7$

▶ « 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

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

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 $O(1)$

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 $O(1)$

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

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 $\Theta(n)$.

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 $\Theta(n)$.

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

Analyse de la résolution par chaînage (2/4)

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 $\Theta(n)$.

\implies 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 : $\alpha = n/m$.

Analyse de la résolution par chaînage (2/4)

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 $\Theta(n)$.

\implies 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

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.

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.

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.

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 α . Donc la recherche s'effectue en 1 (calcul de $h(k)$) + α .

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.

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 $(i - 1)/m$.

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.

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 $= 1 + (i - 1)/m$

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 :

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

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 des analyses

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

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 $O(1)$

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

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 $O(1)$

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

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

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

une fonction :

- ▶ qui se calcule rapidement (en $O(1)$)

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

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

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

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

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

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

Principe des fonctions de hachage usuelles

- 1 transformer la clé k en un entier via une fonction $f : U \mapsto \mathbb{N}$

Principe des fonctions de hachage usuelles

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

Principe des fonctions de hachage usuelles

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

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

Principe des fonctions de hachage usuelles

- 1 transformer la clé k en un entier via une fonction $f : U \mapsto \mathbb{N}$
- 2 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 : fonction f pour hacher des chaînes de caractères

```
#define Beta 19

int f(char str[]) {
    int hash_accu = 0;
    while ( *str != '\0' ) {
        hash_accu = hash_accu * Beta + *str;
        str++;
    }
    return hash_accu;
}
```

Définition

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

Définition

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

Choix de m

- ▶ éviter les collisions \implies utiliser dans g tous les bits de x
 \implies éviter les puissances de 2

Définition

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

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

Définition

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

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

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 $(-1) : \frac{\sqrt{5} - 1}{2}$

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 $[g(x + i), g(x + i + 1)]$.

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

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

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 accès en $O(1)$

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 $k_1 \neq k_2$,
 $|\{h \in \mathcal{H} : h(k_1) = h(k_2)\}| = |\mathcal{H}|/m$

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 $1/m$

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.

Choix d'un ensemble universel

- ▶ On suppose que m est un nombre premier

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 $p < m$

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

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 $\{0, \dots, m - 1\}$

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

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

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

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

- ▶ Alors \mathcal{H} est un ensemble universel.

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

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

Peut-on éviter les pointeurs ?

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

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

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

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.

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

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

```
void hash_insert(bucket_t hashtable[],
                int m,
                char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```



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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```



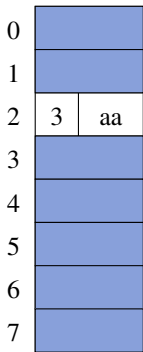
x = "aa", k = 3
 $h(k, 0) = 2$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```



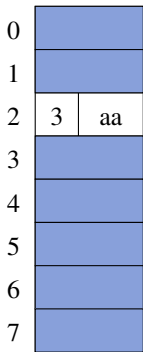
x = "aa", k = 3
 $h(k, 0) = 2$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```



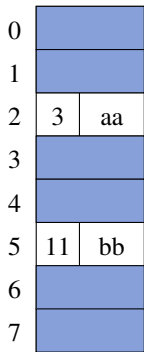
$x = \text{"bb"}, k = 11$
 $h(k, 0) = 5$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```



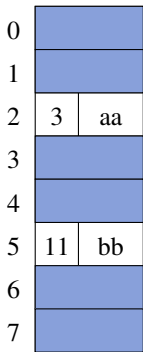
$x = \text{"bb"}, k = 11$
 $h(k, 0) = 5$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```



x = "cc", k = 12
 $h(k, 0) = 3$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```

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

x = "cc", k = 12
 $h(k, 0) = 3$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```

0	[]	
1	[]	
2	3	aa
3	12	cc
4	[]	
5	11	bb
6	[]	
7	[]	

x = "dd", k = 8
 $h(k, 0) = 2$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```

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

x = "dd", k = 8
 $h(k, 1) = 3$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```

0	[]	
1	[]	
2	3	aa
3	12	cc
4	[]	
5	11	bb
6	[]	
7	[]	

x = "dd", k = 8
 $h(k, 2) = 5$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                int m,
                char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```

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

x = "dd", k = 8
 $h(k, 3) = 0$

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

Exemple d'insertion

```
void hash_insert(bucket_t hashtable[],
                 int m,
                 char elt[]) {
    // hachage de elt
    int k = get_cle(elt);

    // on essaye tous les h(k,i)
    // jusqu'à ce qu'une case de
    // hashtable soit vide
    for (int i = 0; i < m; i++) {
        int j = h(k,i);
        if (hashtable[j].cle == -1) {
            // ici, on a trouvé une case vide
            hashtable[j].cle = k;
            hashtable[j].donnee = str;
            return;
        }
    }
    throw std::exception ();
}
```

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

x = "dd", k = 8
 $h(k, 3) = 0$

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

```
data_t hash_search(bucket_t hashtable[],
                  int m,
                  int cle) {
    // on essaye tous les h(k,i) jusqu'à ce qu'on
    // trouve l'élément que l'on recherche
    for (int i = 0; i < m; i++) {
        int j = h(cle, i);
        // si on rencontre une case vide, c'est que
        // l'élément n'est pas dans la table
        if (hashtable[j].cle == -1)
            throw std::exception ();

        // si la cle de la case est cle, on
        // a trouvé l'élément qu'on cherchait
        if (hashtable[j].cle == cle) {
            return hashtable[j].donnee;
        }
    }

    // on a parcouru la table sans trouver
    // l'élément qu'on cherchait
    throw std::exception ();
}
```

- ▶ Suppression \implies « clé = -1 » pour indiquer une case vide ne suffit plus

- ▶ Suppression \implies « clé = -1 » pour indiquer une case vide ne suffit plus
- \implies on définit « clé = -2 » pour indiquer une case qui a été affectée puis supprimée

- ▶ Suppression \implies « clé = -1 » pour indiquer une case vide ne suffit plus
 - \implies on définit « clé = -2 » pour indiquer une case qui a été affectée puis supprimée
 - \implies elle est libre pour des insertions
 - elle compte comme case « non vide » pour les recherches

- ▶ Suppression \implies « clé = -1 » pour indiquer une case vide ne suffit plus
 - \implies on définit « clé = -2 » pour indiquer une case qui a été affectée puis supprimée
 - \implies elle est libre pour des insertions
 - elle compte comme case « non vide » pour les recherches
- ▶ Malheureusement les analyses de complexité ne dépendent plus seulement du facteur de charge α
 - \implies en principe, on utilise plutôt la résolution par chaînage

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

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

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 $1/(1 - \alpha)$.

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 $1/(1 - \alpha)$.

α	50%	80%	90%	99%
scans	2	5	10	100

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 :

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

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 :

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

α	50%	80%	90%	99%
scans	3,396	3,26	3,67	5,66

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 :

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

α	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 $1/(1-\alpha)$.

Probing linéaire

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

Probing linéaire

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

Problème : formation de clusters de cases remplies
 \implies les scans peuvent être longs

Probing linéaire

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

Problème : formation de clusters de cases remplies
 \implies les scans peuvent être longs

Probing quadratique

$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$, où $h' : U \mapsto \{0, \dots, m - 1\}$.

Probing linéaire

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

Problème : formation de clusters de cases remplies
 \implies les scans peuvent être longs

Probing quadratique

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, 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

double hachage

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

double hachage

$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$, 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.

double hachage

$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$, 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

Opération	Complexité	Intérêt
Insertion d'un élément	$O(1)$	✓
Suppression d'un élément	$O(1)$	✓
Accéder à un élément donné	$O(1)$	✓
Accéder à l'élément le plus petit	$O(n)$	✗
Accéder à l'élément le plus grand	$O(n)$	✗
Chercher si un élément existe	$O(1)$	✓