

Module EA4 – Éléments d’Algorithmique

Dominique Poulalhon

dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot

L2 Informatique

Année universitaire 2014-2015

EXAMEN – SESSION 1

mercredi 27 mai 15h30-18h30

Documents autorisés

uniquement une feuille A4 manuscrite
(recto-verso)

mercredi 27 mai 15h30-18h30

Documents autorisés

uniquement une feuille A4 manuscrite
(recto-verso)

Rappel : *manuscrit* vient de *manus, manus, f.* : la main

mercredi 27 mai 15h30-18h30

Documents autorisés

uniquement une feuille A4 manuscrite
(recto-verso)

Rappel : *manuscrit* vient de *manus*, *manus*, *f.* : la main

Autrement dit :

- tout autre document (manuscrit, imprimé ou électronique) *interdit*
- matériel électronique *éteint et rangé*

DIFFÉRENTES REPRÉSENTATIONS DES ENSEMBLES

	tableau		liste chaînée		ABR (en moyenne)
	non trié	trié	non triée	triée	
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
insertion	$+ \Theta(1)$	$\Theta(n)$	$+ \Theta(1)$	$+ \Theta(1)$	$\Theta(\log n)$
suppression	$\Theta(n)$	$\Theta(n)$	$+ \Theta(1)$	$+ \Theta(1)$	$\Theta(\log n)$

DIFFÉRENTES REPRÉSENTATIONS DES ENSEMBLES

	tableau		liste chaînée		ABR
	non trié	trié	non triée	triée	(en moyenne)
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
insertion	$+ \Theta(1)$	$\Theta(n)$	$+ \Theta(1)$	$+ \Theta(1)$	$\Theta(\log n)$
suppression	$\Theta(n)$	$\Theta(n)$	$+ \Theta(1)$	$+ \Theta(1)$	$\Theta(\log n)$

Est-ce qu'on ne peut vraiment pas faire mieux, si on ne demande *que* ces trois opérations ?

SOLUTION SIMPLE : L'ADRESSAGE DIRECT

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace) ?

SOLUTION SIMPLE : L'ADRESSAGE DIRECT

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace) ?

réponse : oui

- allouer un tableau `T` suffisamment grand
- stocker `True` ou `False` dans `T[i]` si `i` est dans l'ensemble

SOLUTION SIMPLE : L'ADRESSAGE DIRECT

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace) ?

réponse : oui

- allouer un tableau T suffisamment grand
- stocker `True` ou `False` dans $T[i]$ si i est dans l'ensemble

une solution, vraiment ???

- si les éléments de l'ensemble sont des entiers
- si le nombre de valeurs possibles est raisonnable : la complexité en espace est $\Theta(\max)$ où \max est la plus grande valeur possible, indépendamment de la taille n de l'ensemble

LES ENSEMBLES EN PYTHON

```
>>> S = { 'a', 2, 4, (1,1) }
>>> 'a' in S
True
>>> 'b' in S
False
>>> S.add('b')
>>> 'b' in S
True
>>> S.remove(2)
>>> S.remove(4)
>>> S
{'a', 'b', (1, 1)}
>>> S.add('b')
>>> S
{'a', 'b', (1, 1)}
```

LES DICTIONNAIRES EN PYTHON

```
>>> D = { 'a' : 2, 'b' : 5, (1,2) : 'toto' }
>>> 'c' in D
False
>>> 'a' in D
True
>>> D['a']
2
>>> D.pop('a')
2
>>> D
{(1, 2): 'toto', 'b': 5}
>>> D['c'] = 'coucou'
>>> D
{(1, 2): 'toto', 'c': 'coucou', 'b': 5}
```

ENSEMBLES *vs* DICTIONNAIRES

une différence

- les ensembles contiennent seulement des éléments (**clés**)
- dans les dictionnaires, des **valeurs** sont attachées aux **clés**
(on parle parfois de **données satellites**)

ENSEMBLES *vs* DICTIONNAIRES

une différence

- les ensembles contiennent seulement des éléments (**clés**)
- dans les dictionnaires, des **valeurs** sont attachées aux **clés** (on parle parfois de **données satellites**)

des points communs

- les clés peuvent être des **entiers**, des **réels**, des **chaînes de caractères**, des **tuples** (mais **pas des listes**, par exemple)
- ce qui fait que l'ensemble de clés possibles est **infini**
- il n'y a pas de contrainte d'**homogénéité**

ENSEMBLES *vs* DICTIONNAIRES

une différence

- les ensembles contiennent seulement des éléments (**clés**)
- dans les dictionnaires, des **valeurs** sont attachées aux **clés**
(on parle parfois de **données satellites**)

des points communs

- les clés peuvent être des **entiers**, des **réels**, des **chaînes de caractères**, des **tuples** (mais **pas des listes**, par exemple)
- ce qui fait que l'ensemble de clés possibles est **infini**
- il n'y a pas de contrainte d'**homogénéité**

les dictionnaires sont des ensembles de couples (**clé, valeur**), rangés en tenant compte seulement de la clé

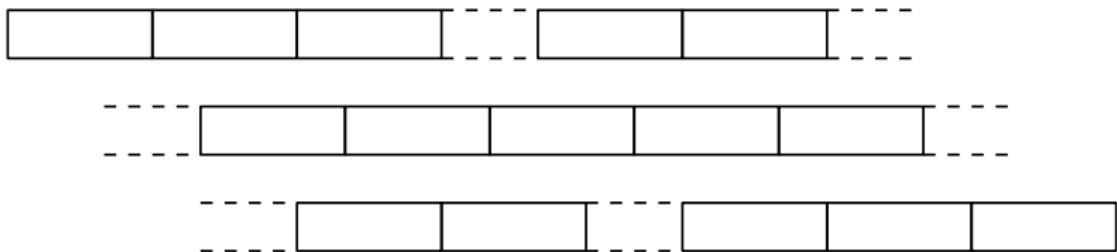
PRINCIPE DU HACHAGE

- allouer un (grand) tableau T de taille max



PRINCIPE DU HACHAGE

- allouer un (grand) tableau **T** de taille **max**
- transformer n'importe quelle clé en **entier** plus petit que **max** à l'aide d'une **fonction de hachage h**



$$h(\text{lancelot}) = 12$$

PRINCIPE DU HACHAGE

- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en `entier` plus petit que `max` à l'aide d'une `fonction de hachage h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



PRINCIPE DU HACHAGE

- allouer un (grand) tableau T de taille \max
- transformer n'importe quelle clé en entier plus petit que \max à l'aide d'une **fonction de hachage h**
- stocker chaque élément elt dans la case $T[h(\text{elt})]$ (on parle de *boîte* ou *bucket*)



PRINCIPE DU HACHAGE

- allouer un (grand) tableau T de taille \max
- transformer n'importe quelle clé en entier plus petit que \max à l'aide d'une **fonction de hachage h**
- stocker chaque élément elt dans la case $T[h(\text{elt})]$ (on parle de *boîte* ou *bucket*)



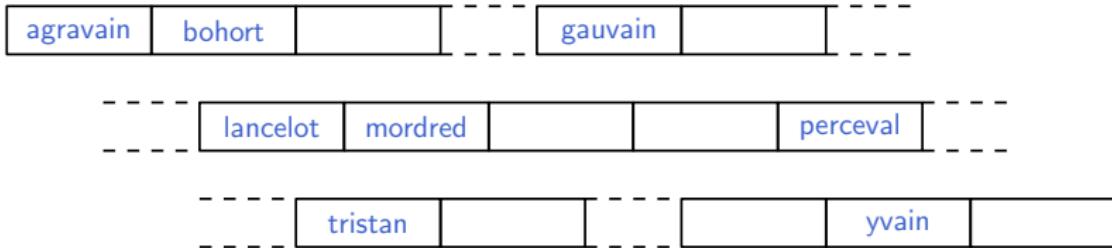
PRINCIPE DU HACHAGE

- allouer un (grand) tableau T de taille max
- transformer n'importe quelle clé en entier plus petit que max à l'aide d'une **fonction de hachage h**
- stocker chaque élément elt dans la case $T[h(\text{elt})]$ (on parle de *boîte* ou *bucket*)



PRINCIPE DU HACHAGE

- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en `entier` plus petit que `max` à l'aide d'une `fonction de hachage h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



PRINCIPE DU HACHAGE

```
def ajouter(table, elt) :  
    table[h(elt)] = True  
  
def supprimer(table, elt) :  
    table[h(elt)] = False  
  
def chercher(table, elt) :  
    return table[(h(elt))]
```

PRINCIPE DU HACHAGE

```
def ajouter(table, cle, valeur) :  
    table[h(cle)] = valeur  
  
def supprimer(table, cle) :  
    table[h(cle)] = None  
  
def chercher(table, cle) :  
    return table[(h(cle))]
```

COLLISIONS

Problème : l'ensemble des clés possibles est infini, et l'ensemble des valeurs hachées est fini...

COLLISIONS

Problème : l'ensemble des clés possibles est infini, et l'ensemble des valeurs hachées est fini...

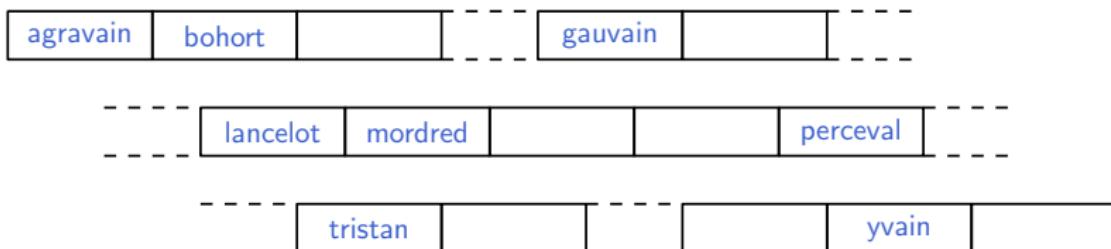
nécessairement, il existe des clés `cle1` et `cle2` telles que
`h(cle1) = h(cle2)`

COLLISIONS

Problème : l'ensemble des clés possibles est infini, et l'ensemble des valeurs hachées est fini...

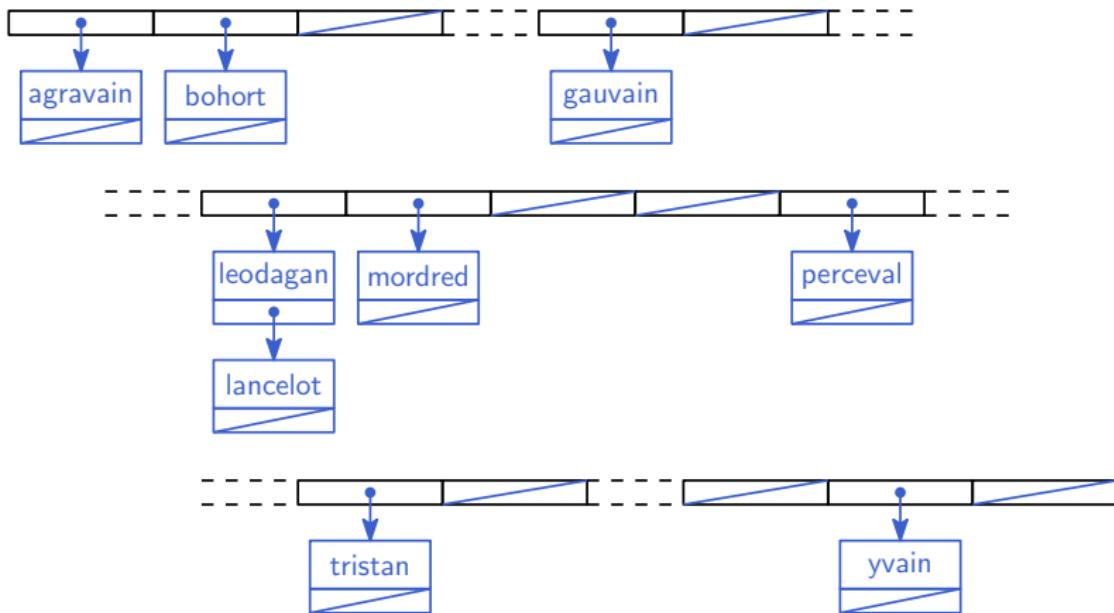
nécessairement, il existe des clés `cle1` et `cle2` telles que
`h(cle1) = h(cle2)`

si on cherche à insérer deux telles clés, il se produit une collision : deux éléments à placer dans la même boîte



$$h(\text{leodagan}) = 12 = h(\text{lancelot})$$

RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE OU « HACHAGE OUVERT »



RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE OU « HACHAGE OUVERT »

Implémentation avec des listes PYTHON :

```
def ajouter(table, elt) :  
    table[h(elt)].append(elt)
```

```
def supprimer(table, elt) :  
    table[h(elt)].remove(elt)
```

```
def chercher(table, elt) :  
    return elt in table[h(elt)]
```

RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE OU « HACHAGE OUVERT »

Implémentation avec des listes PYTHON :

```
def ajouter(table, elt) :  
    table[h(elt)].append(elt)  
  
# quoi ?! sans même vérifier si elt est déjà dans table ???  
  
def supprimer(table, elt) :  
    table[h(elt)].remove(elt)  
  
  
def chercher(table, elt) :  
    return elt in table[h(elt)]
```

RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE OU « HACHAGE OUVERT »

Implémentation avec des listes PYTHON :

```
def ajouter(table, elt) :  
    table[h(elt)].append(elt)  
  
# quoi ?! sans même vérifier si elt est déjà dans table ???  
  
def supprimer(table, elt) :  
    table[h(elt)].remove(elt)  
  
# admettons, mais alors remove(elt) ne suffit pas  
  
def chercher(table, elt) :  
    return elt in table[h(elt)]
```

RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE OU « HACHAGE OUVERT »

Implémentation avec des listes PYTHON :

```
def ajouter(table, elt) :  
    table[h(elt)].append(elt)  
  
# quoi ?! sans même vérifier si elt est déjà dans table ???  
  
def supprimer(table, elt) :  
    table[h(elt)].remove(elt)  
  
# admettons, mais alors remove(elt) ne suffit pas  
  
def chercher(table, elt) :  
    return elt in table[h(elt)]  
  
# là en revanche, OK
```

RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE OU « HACHAGE OUVERT »

Alternative :

```
def ajouter(table, elt) :  
    hache = h(elt)  
    if elt not in table[hache] :  
        table[hache].append(elt)  
  
def supprimer(table, elt) :  
    table[h(elt)].remove(elt)  
  
def chercher(table, elt) :  
    return elt in table[h(elt)]
```

RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE OU « HACHAGE OUVERT »

Variante pour les dictionnaires :

```
def ajouter(table, cle, valeur) :  
    table[h(cle)].append((cle, valeur)) # mais bien sûr...  
  
def chercher(table, cle) :  
    for key, val in table[h(cle)][::-1] : # attention,  
        if key == cle : return val          # parcours à l'envers !  
    return None                          # pourquoi, au fait ?  
  
def supprimer(table, cle) :  
    for key, val in table[h(cle)] :  
        if key == cle :  
            table[h(cle)].remove((key, val)) # ben voyons !  
    return
```

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()`, `rechercher()` et `supprimer()` ont un coût linéaire en la taille de la boîte où se trouve l'élément

(mais si on y tient, `ajouter()` peut avoir un coût constant)

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()`, `rechercher()` et `supprimer()` ont un coût linéaire en la taille de la boîte où se trouve l'élément

(mais si on y tient, `ajouter()` peut avoir un coût constant)

Que vaut cette taille (en moyenne sur les boîtes occupées) ?

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()`, `rechercher()` et `supprimer()` ont un coût linéaire en la taille de la boîte où se trouve l'élément

(mais si on y tient, `ajouter()` peut avoir un coût constant)

Que vaut cette taille (en moyenne sur les boîtes occupées) ?

elle dépend du taux de charge $\alpha = \frac{n}{\max}$ de la table et de la façon dont les données sont bien (ou mal) réparties dans la table, donc de la fonction de hachage

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()`, `rechercher()` et `supprimer()` ont un coût linéaire en la taille de la boîte où se trouve l'élément

(mais si on y tient, `ajouter()` peut avoir un coût constant)

Que vaut cette taille (en moyenne sur les boîtes occupées) ?

elle dépend du taux de charge $\alpha = \frac{n}{\max}$ de la table et de la façon dont les données sont bien (ou mal) réparties dans la table, donc de la fonction de hachage

Théorème

si la répartition des éléments est uniforme dans la table, le coût moyen d'une recherche (réussie ou non) est $O(1 + \frac{n}{\max})$.

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

Théorème

si la répartition des éléments est uniforme dans la table, le coût moyen d'une recherche est $O(1 + \frac{n}{m_{\max}})$.

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

Théorème

si la répartition des éléments est uniforme dans la table, le coût moyen d'une recherche est $O(1 + \frac{n}{\max})$.

Corollaire

si la longueur \max de la table est choisie supérieure à αn pour un α fixé, alors le coût moyen d'une recherche (ou d'un ajout, ou d'une suppression) est $O(1 + \alpha) = O(1)$.

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

Théorème

si la répartition des éléments est uniforme dans la table, le coût moyen d'une recherche est $O(1 + \frac{n}{\max})$.

Corollaire

si la longueur \max de la table est choisie supérieure à αn pour un α fixé, alors le coût moyen d'une recherche (ou d'un ajout, ou d'une suppression) est $O(1 + \alpha) = O(1)$.

mais si n n'est pas connu à l'avance ?

REDIMENSIONNEMENT DE TABLE DE HACHAGE

Soit α le taux de remplissage à *ne pas dépasser*

initialisation : choisir un `max` arbitraire, allouer un tableau `T1` de longueur `max` et choisir une fonction de hachage `h1` à valeurs dans $[1, \text{max}]$

évolution : au fur et à mesure des ajouts ou suppressions, calculer le taux de remplissage effectif β

redimensionnement : si β atteint α :

- créer une nouvelle table `T2` de longueur 2 max et choisir une nouvelle fonction de hachage `h2` à valeurs dans $[1, 2 \text{ max}]$
- parcourir `T1` pour transférer tous ses éléments dans `T2`
- faire : `max, T1, h1 = 2 * max, T2, h2`

COMPLEXITÉ DU REDIMENSIONNEMENT

nichée dans le parcours de **T1** : si une table de longueur m contient n éléments, le parcours a une complexité $\Theta(m + n)$

COMPLEXITÉ DU REDIMENSIONNEMENT

nichée dans le parcours de T_1 : si une table de longueur m contient n éléments, le parcours a une complexité $\Theta(m + n)$

ici, $m = \alpha n$ avec α fixé, donc $\Theta(m + n) = \Theta(n)$: le redimensionnement a un coût linéaire

COMPLEXITÉ DU REDIMENSIONNEMENT

nichée dans le parcours de **T1** : si une table de longueur m contient n éléments, le parcours a une complexité $\Theta(m + n)$

ici, $m = \alpha n$ avec α fixé, donc $\Theta(m + n) = \Theta(n)$: le redimensionnement a un coût linéaire

... mais ce redimensionnement a lieu après *au moins* n ajouts

COMPLEXITÉ DU REDIMENSIONNEMENT

nichée dans le parcours de T_1 : si une table de longueur m contient n éléments, le parcours a une complexité $\Theta(m + n)$

ici, $m = \alpha n$ avec α fixé, donc $\Theta(m + n) = \Theta(n)$: le redimensionnement a un coût linéaire

... mais ce redimensionnement a lieu après *au moins* n ajouts

on dit que le coût *amorti* du redimensionnement est constant

COMPLEXITÉ DU REDIMENSIONNEMENT

nichée dans le parcours de T1 : si une table de longueur m contient n éléments, le parcours a une complexité $\Theta(m + n)$

ici, $m = \alpha n$ avec α fixé, donc $\Theta(m + n) = \Theta(n)$: le redimensionnement a un coût linéaire

... mais ce redimensionnement a lieu après *au moins* n ajouts

on dit que le coût *amorti* du redimensionnement est constant

Théorème

si la répartition des éléments est uniforme dans une table à taux de remplissage borné, le coût moyen amorti des opérations de recherche, ajout et suppression est $\Theta(1)$.

RÉSOLUTION PAR ADRESSAGE OUVERT OU « HACHAGE FERMÉ » (*sic*)

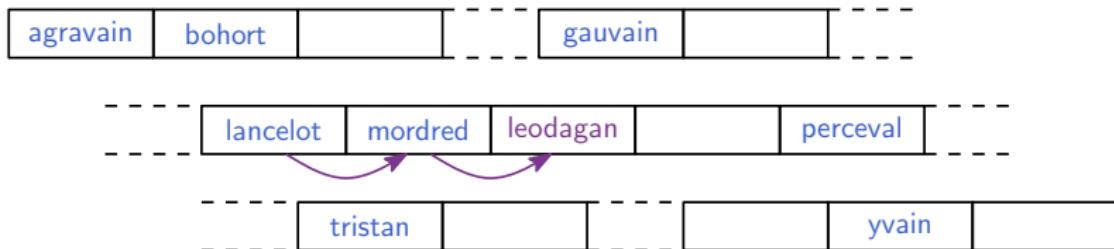
en utilisant directement la table (*espace d'adressage*) pour stocker les données : si une cellule est occupée, essayer ailleurs !

problème : comment retrouver ensuite cet « ailleurs » ?

sondage linéaire : si $T[h(\text{cle})]$ est occupée, tester itérativement $T[h(\text{cle}) + 1]$, $T[h(\text{cle}) + 2]$, etc.

HACHAGE (PAR SONDAGE) LINÉAIRE

si $T[h(\text{cle})]$ est occupée, tester itérativement $T[h(\text{cle}) + 1]$,
 $T[h(\text{cle}) + 2]$, etc.



$$h(\text{leodagan}) = 12 = h(\text{lancelot})$$

HACHAGE (PAR SONDAGE) LINÉAIRE

```
def ajouter(table, cle, valeur) :  
    for i in range(h(cle), len(table)) :  
        if table[i] == None or table[i][0] == cle :  
            break  
    else : # si on atteint la fin, on recommence au début  
    for i in range(h(cle)) :  
        if table[i] == None or table[i][0] == cle :  
            break  
    table[i] = (cle, valeur)
```

HACHAGE (PAR SONDAGE) LINÉAIRE

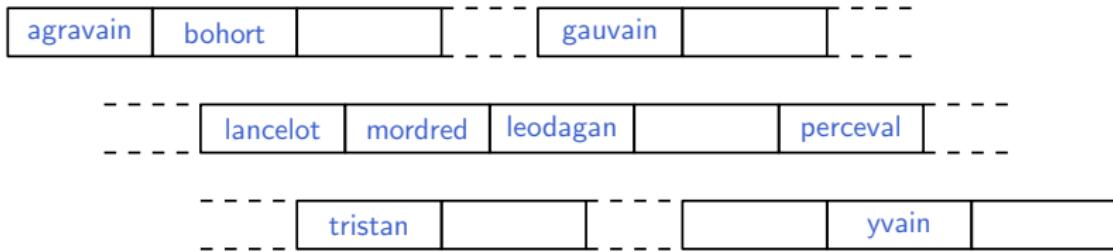
```
def chercher(table, cle) :  
    for i in range(h(cle), len(table)) :  
        if table[i] == None : return  
        if table[i][0] == cle : return table[i][1]  
    # au cas où, on recommence au début  
    ...
```

HACHAGE (PAR SONDAGE) LINÉAIRE

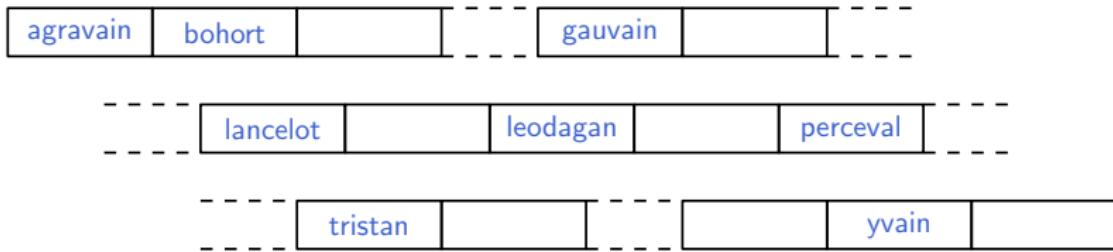
```
def chercher(table, cle) :
    for i in range(h(cle), len(table)) :
        if table[i] == None : return
        if table[i][0] == cle : return table[i][1]
    # au cas où, on recommence au début
    ...

def supprimer(table, cle) :
    for i in range(h(cle), len(table)) :
        if table[i] == None : return
        if table[i][0] == cle :
            table[i] = None
    return
    # au cas où, on recommence au début
    ...
```

HACHAGE (PAR SONDAGE) LINÉAIRE



HACHAGE (PAR SONDAGE) LINÉAIRE



HACHAGE (PAR SONDAGE) LINÉAIRE



GLOUPS!!!

HACHAGE (PAR SONDAGE) LINÉAIRE

```
def supprimer(table, cle) :
    for i in range(h(cle), len(table)) :
        if table[i] == None : return
        if table[i][0] == cle :
            # la case n'est pas vidée, mais libérée
            table[i][1] = None
        return
    # au cas où, on recommence au début
    ...
```

HACHAGE (PAR SONDAGE) LINÉAIRE

```
def chercher(table, cle) :
    for i in range(h(cle), len(table)) :
        if table[i] == None : return
        if table[i][0] == cle : return table[i][1]
    # au cas où, on recommence au début
    ...

def ajouter(table, cle, valeur) :
    for i in range(h(cle), len(table)) :
        if table[i] == None or table[i][1] == None :
            # pas tout à fait suffisant (pb si cle est déjà dans table)
            table[i] = (cle, valeur)
            return
    # au cas où, on recommence au début
    ...
```

QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?

QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?



QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?



QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?



QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?

pour une bonne efficacité (en temps), il faut :

- trouver rapidement la bonne boîte
- fouiller rapidement la boîte

et bien sûr, il faut éviter le gâchis en espace.

QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?

pour une bonne efficacité (en temps), il faut :

- trouver rapidement la bonne boîte
- fouiller rapidement la boîte

et bien sûr, il faut éviter le gâchis en espace.

la fonction de hachage doit donc

- être facile à calculer
- idéalement, être sans collision – et s'il y en a, les éléments concernés doivent être facilement discernables
- remplir la table *uniformément* : éviter d'avoir de grandes zones vides et de grandes zones pleines
- pour cela, il faut disperser les données similaires

QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?

pour répondre, il faut savoir **à quoi ressemblent les données**

QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?

pour répondre, il faut savoir **à quoi ressemblent les données**

Exemple

toutes les chaînes de moins de 25 caractères ne peuvent pas être des entrées d'un dictionnaire (un vrai)

QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?

pour répondre, il faut savoir **à quoi ressemblent les données**

Exemple

un compilateur maintient une **table des symboles** référençant les identificateurs du programme en cours de compilation
or les programmeurs ont tendance à utiliser des identificateurs qui se ressemblent : **tmp, tmp1, tmp2...**

QU'EST-CE-QU'UNE BONNE FONCTION DE HACHAGE ?

pour répondre, il faut savoir **à quoi ressemblent les données**

Exemple

un compilateur maintient une **table des symboles** référençant les identificateurs du programme en cours de compilation
or les programmeurs ont tendance à utiliser des identificateurs qui se ressemblent : **tmp, tmp1, tmp2...**

en général, les données ne sont pas réparties uniformément dans l'univers des données possibles

CONSTRUCTION DE FONCTIONS DE HACHAGE

une fonction de hachage doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

CONSTRUCTION DE FONCTIONS DE HACHAGE

une fonction de hachage doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

deux étapes

- transformer toute donnée en valeur numérique (entière)
- hacher les nombres

CONSTRUCTION DE FONCTIONS DE HACHAGE

une fonction de hachage doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

deux étapes

- transformer toute donnée en valeur numérique (entière)
- hacher les nombres

pour du texte, le plus simple : remplacer chaque caractère par son code ASCII, et considérer le texte $t_0 \dots t_m$ comme l'entier

$$h(t_0t_1\dots t_m) = t_0b^m + t_1b^{m-1} + \dots + t_{m-1}b + t_m$$

(en Java : $b = 31$)

CONSTRUCTION DE FONCTIONS DE HACHAGE

une fonction de hachage doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

méthode *par division*

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

CONSTRUCTION DE FONCTIONS DE HACHAGE

une fonction de hachage doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

méthode *par division*

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

méthode *par multiplication*

$$h(x) = \lfloor m \times \{Ax\} \rfloor$$

avec $\{x\} = x - \lfloor x \rfloor$

- m a peu d'importance, par exemple une puissance de 2
- une bonne valeur pour A est $\frac{\sqrt{5}-1}{2}$ (ou une approximation fractionnaire)