

Module EA4 – Éléments d’Algorithmique

Dominique Poulalhon

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

Université Paris Diderot
L2 Informatique, Math-Info et EIDD
Année universitaire 2013-2014

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	$\Theta(n)$	$\Theta(n)$	$+ \Theta(1)$	$+ \Theta(1)$	$\Theta(\log n)$
minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$

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)}
```

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 $T[h(\text{elt})]$ (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 $\text{fonction de hachage } h$
- 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*)



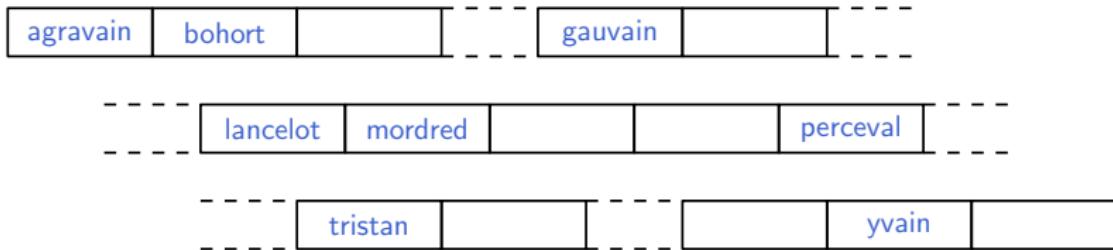
PRINCIPE DU HACHAGE

- 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 $T[h(\text{elt})]$ (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...

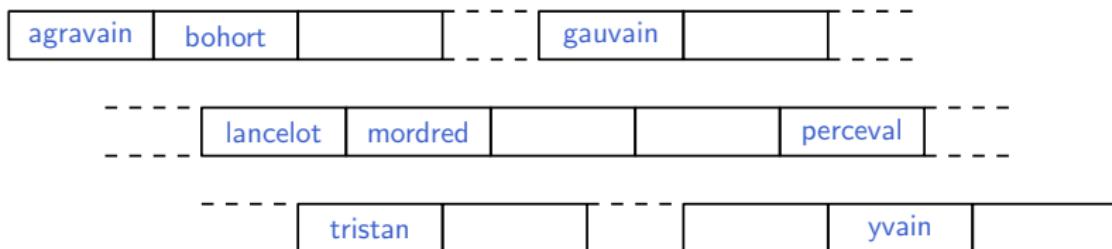
nécessairement, il existe des clés `cle1` et `cle2` telles que
`h(cle1) = h(cle2)`

COLLISIONS

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

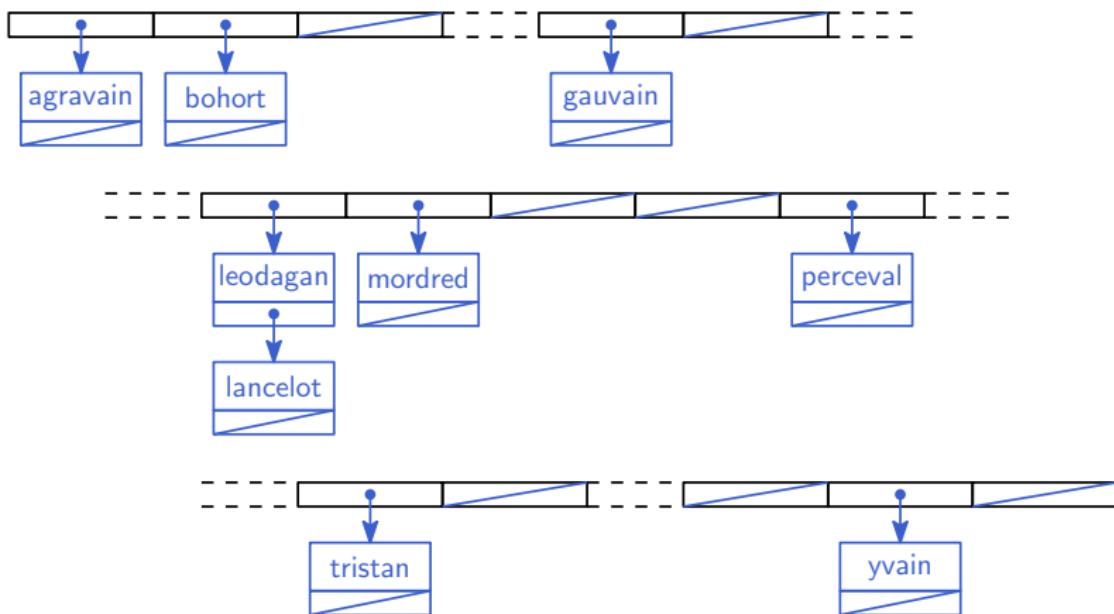
nécessairement, il existe des clés `cle1` et `cle2` telles que
`h(cle1) = h(cle2)`

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, cle, valeur) :  
    table[h(cle)].append((cle, valeur))  
  
def chercher(table, cle) :  
    for key, val in table[h(cle)] :  
        if key == cle : return val  
    return None  
  
def supprimer(table, cle) :  
    for key, val in table[h(cle)] :  
        if key == cle :  
            table[h(cle)].remove((key, val))  
    return
```

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

ajouter() a un coût constant

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

ajouter() a un coût constant

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()` a un coût constant

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

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()` a un coût constant

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

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()` a un coût constant

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

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

COMPLEXITÉ DU HACHAGE PAR CHAÎNAGE

`ajouter()` a un coût constant

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

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

elle dépend du taux de charge $\frac{n}{\max}$ 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()` a un coût constant

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

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

elle dépend du taux de charge $\frac{n}{\max}$ 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 (admis)

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

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

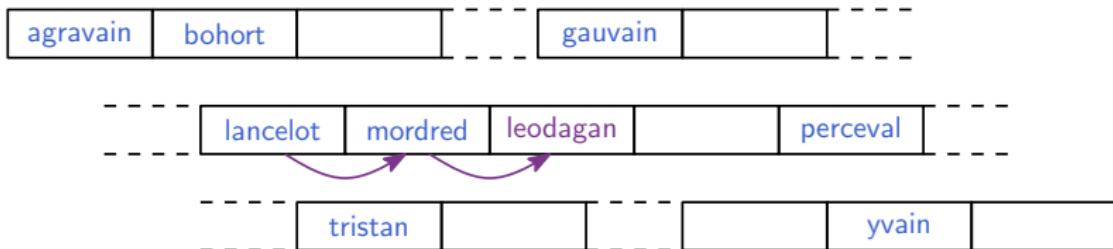
en utilisant directement la table (*espace d'adressage*) pour stocker les données : si une cellule est occupée, essayer ailleurs !

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 :  
            table[i] = (cle, valeur)  
            return  
# au cas où, on recommence au début  
    i = 0  
    while table[i] != None : i = i + 1  
    table[i] = (cle, valeur)
```

HACHAGE (PAR SONDAGE) LINÉAIRE

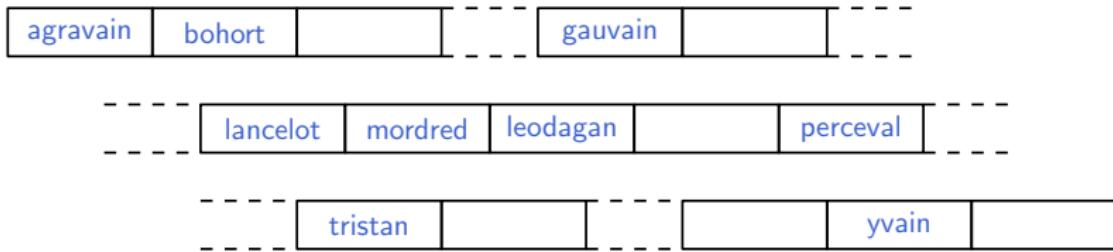
```
def chercher(table, cle) :  
    for i in range(h(cle), len(table)) :  
        if table[i] == None : return  
        if table[i][0] == cle : return table[i][1]  
    # au cas où, on recommence au début  
    ...
```

HACHAGE (PAR SONDAGE) LINÉAIRE

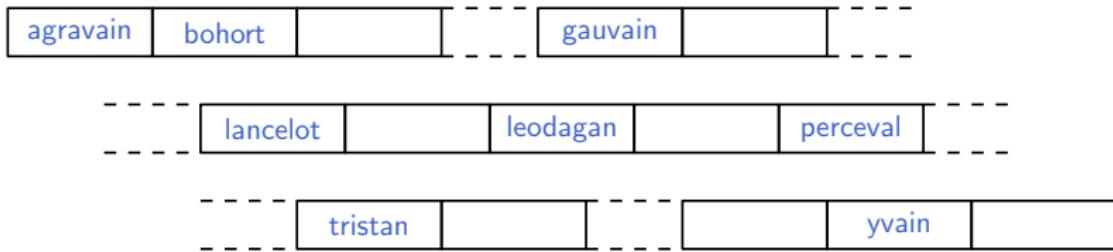
```
def chercher(table, cle) :
    for i in range(h(cle), len(table)) :
        if table[i] == None : return
        if table[i][0] == cle : return table[i][1]
    # au cas où, on recommence au début
    ...

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 ajouter(table, cle, valeur) :
    for i in range(h(cle), len(table)) :
        if table[i] == None or table[i][1] == None :
            table[i] = (cle, valeur)
    return

# au cas où, on recommence au début
...
...

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]
    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 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

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

pour une bonne efficacité, il faut :

- trouver rapidement la bonne boîte
- fouiller rapidement la boîte

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

pour une bonne efficacité, il faut :

- trouver rapidement la bonne boîte
- fouiller rapidement la boîte

la fonction de hachage doit donc

- être facile à calculer
- idéalement, être sans collision
- remplir la table *uniformément* : éviter d'avoir de grandes zones vides et de grandes zones pleines
- pour cela, il faut disperser les données similaires