Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2019-2020

Le hachage

I. Principe général

Différentes représentations des ensembles

Pour rappel, les cours précédents ont montré qu'on pouvait obtenir les complexités suivantes pour les opérations usuelles sur les 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	$\Theta(n)$	$\Theta(n)$	$+\,\Theta(1)$	$+\,\Theta(1)$	$\Theta(\log n)$

Question : est-ce vraiment optimal, si on ne demande *que* ces trois opérations ou si on accepte d'être moins efficace pour les autres (union, intersection, sélection...) ?

Soyons fous :

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace, et/ou la complexité en temps pour les autres opérations) ?

Solution simple : l'adressage direct

Réponse (naïve) fréquente : oui, évidemment ! Il suffit :

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

Comme trop de réponses naïves, celle-ci ne résiste pas à un examen attentif :

- *il faut* que les éléments de l'ensemble soient des nombres, et même des entiers, sinon parler de `T[i]` n'a tout simplement pas de sens.
  On peut en revanche autoriser les entiers négatifs, en décalant suffisamment les éléments si on sait qu'il sont tous supérieurs à une certaine borne `min`*. On peut même autoriser des fractions à dénominateur borné... mais pas n'importe quels nombres, et en tout cas pas des éléments non numériques.*
- *il faut* que le nombre de valeurs possibles soit raisonnable : la complexité en espace est $\Theta($`max-min`$)$ où `min` et `max` sont les valeurs extrémales possibles, *indépendamment* de la taille `n` de l'ensemble.
  Par exemple, la place nécessaire pour stocker un ensemble d'entiers entre -10^9 et -10^9 est de 2 Go (si un booléen est stocké sur un octet, ce qui n'est certes pas optimal ; 250 Mo au mieux), même si l'ensemble ne contient que 5 éléments... Pas terrible. Et *lister* les éléments de l'ensemble nécessite de parcourir les 2 milliards de cases...

Pourtant, en Python par exemple...

...il existe deux types de données qui sont « vendus » pour assurer une complexité en $O(1)$ pour les accès (ajout/recherche/suppression) (et ce n'est pas spécifique à Python bien sûr) : les *ensembles* et les *dictionnaires*.

Exemple :

```
>>> S = { 'a', 2, 4, (1,1) } # exemple d'ensemble
>>> 'a' in S; 'b' in S
True
False
>>> S.add('b'); 'b' in S
True
>>> S.remove(2); S.remove(4); S.add('b'); S
{'b', 'a', (1, 1)}
>>>
>>> D = { 'a' : 2, 'b' : 5, (1,2) : 'toto' } # exemple de dictionnaire
>>> 'a' in D; 'c' in D
True
False
>>> D['a']
2
>>> D.pop('a'); D
2
{(1, 2): 'toto', 'b': 5}
>>> D['c'] = 'coucou'; D
{(1, 2): 'toto', 'c': 'coucou', 'b': 5}
```

une petite différence...

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

pour beaucoup de points communs :

- les clés peuvent être des entiers, des réels, des chaînes de caractères, des tuples (mais pas des listes, par exemple : seuls les types *non mutables* sont autorisés) ;
- ... ce qui fait que l'ensemble de clés possibles est infini ;
- il n'y a pas de contrainte d'homogénéité de type entre les clés ;
- la recherche est réputée coûter $O(1)$.

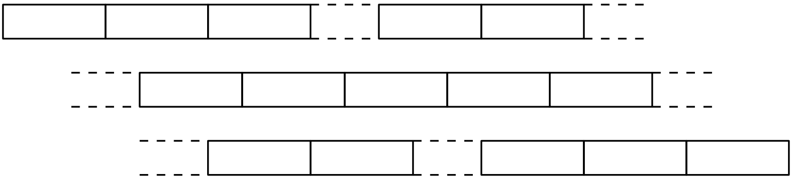
en fait, les dictionnaires sont essentiellement des ensembles de couples (`clé, valeur`), rangés en tenant uniquement compte de la clé

Principe du hachage

Ensembles et dictionnaires sont en fait des *tables de hachage*, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier...

- allouer un (grand) tableau `T` de taille `m` ;
- transformer n'importe quelle clé en un entier plus petit que `m` 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*).

Exemple : (en utilisant pour valeur de hachage d'un mot le rang de son initiale dans l'alphabet)

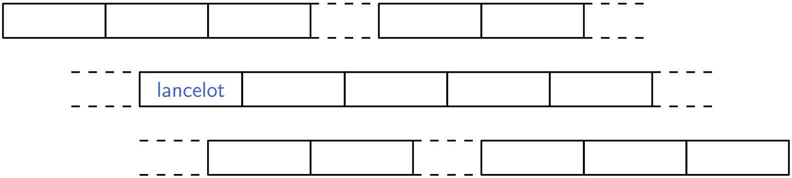


Principe du hachage

Ensembles et dictionnaires sont en fait des *tables de hachage*, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier…

- allouer un (grand) tableau `T` de taille `m` ;
- transformer n'importe quelle clé en un entier plus petit que `m` 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*).

Exemple : (en utilisant pour valeur de hachage d'un mot le rang de son initiale dans l'alphabet)



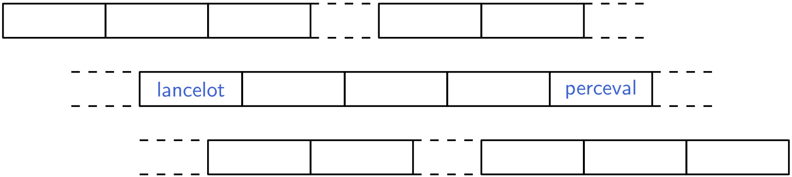
$$\mathsf{h}(\mathsf{lancelot}) = 12$$

Principe du hachage

Ensembles et dictionnaires sont en fait des *tables de hachage*, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier…

- allouer un (grand) tableau `T` de taille `m` ;
- transformer n'importe quelle clé en un entier plus petit que `m` 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*).

Exemple : (en utilisant pour valeur de hachage d'un mot le rang de son initiale dans l'alphabet)



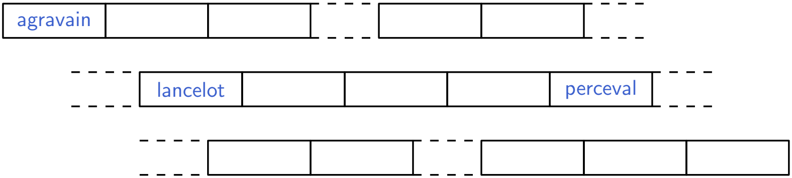
$$\mathrm{h}(\mathrm{perceval}) = 16$$

Principe du hachage

Ensembles et dictionnaires sont en fait des *tables de hachage*, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier...

- allouer un (grand) tableau `T` de taille `m` ;
- transformer n'importe quelle clé en un entier plus petit que `m` 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*).

Exemple : (en utilisant pour valeur de hachage d'un mot le rang de son initiale dans l'alphabet)



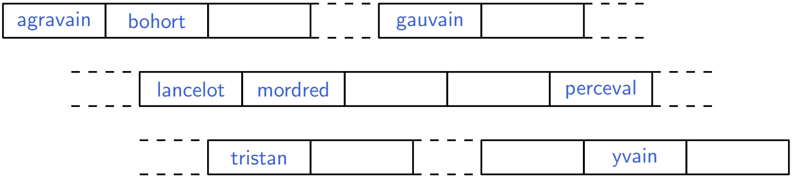
$$\mathsf{h}(\mathsf{agravain}) = 1$$

Principe du hachage

Ensembles et dictionnaires sont en fait des *tables de hachage*, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier…

- allouer un (grand) tableau `T` de taille `m` ;
- transformer n'importe quelle clé en un entier plus petit que `m` 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*).

Exemple : (en utilisant pour valeur de hachage d'un mot le rang de son initiale dans l'alphabet)



Principe du hachage

Donc, en gros, les opérations d'ajout, suppression et recherche correspondent aux fonctions suivantes :

attention, version grossière – donc (vraiment très) très fausse

Pour les ensembles :

```
def ajouter(table, elt) :
  table[h(elt)] = True
  # ou : table[h(elt)] = elt

def supprimer(table, elt) :
  table[h(elt)] = False
  # ou : table[h(elt)] = None

def chercher(table, elt) :
  return table[(h(elt)]
  # ou : return table[h(elt)] == elt
```

Pour les dictionnaires :

```
def ajouter(table, cle, valeur) :
  table[h(elt)] = valeur
  # ou : table[h(cle)] = (cle, valeur)

def supprimer(table, cle) :
  table[h(cle)] = None

def chercher(table, cle) :
  return table[(h(cle)]
  # ou : return table[(h(cle)][1]
```

attention, version grossière – donc (vraiment très) très fausse

La version commentée est un tout petit peu moins naïve que l'autre, car elle tient compte du fait que chaque case peut correspondre à plusieurs clés différentes ; mais cela reste trop naïf pour fonctionner.

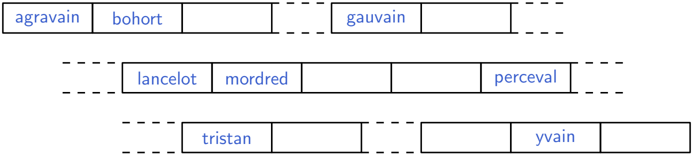
Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très *gros problème* :

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

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

Dès qu'on cherche à insérer deux telles clés, il se produit ce qu'on appelle une ***collision*** : deux éléments doivent être placés dans la même boîte.

Exemple :



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

Problème des collisions

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très *gros problème* :

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

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

Dès qu'on cherche à insérer deux telles clés, il se produit ce qu'on appelle une ***collision*** : deux éléments doivent être placés dans la même boîte.

Le principe du hachage ne peut donc pas fonctionner sans prévoir un mécanisme additionnel de ***résolution des collisions***.

Pour cela, il y a deux grandes méthodes :

- la résolution des collisions par ***chaînage*** ;
- la résolution des collisions par ***sondage***.