Collections : Ensembles (II)

Théorie et pratique de la programmation
Michel Schinz – 2013-03-11

Mise en œuvre 2 : table de hachage

Exemple : gestion d'amis

On désire écrire un programme de « gestion d'amis », qui permette de dire si un nom donné fait partie de nos amis ou non.

On pourrait bien entendu utiliser un arbre de recherche, mais essayons une autre solution…

La solution magique

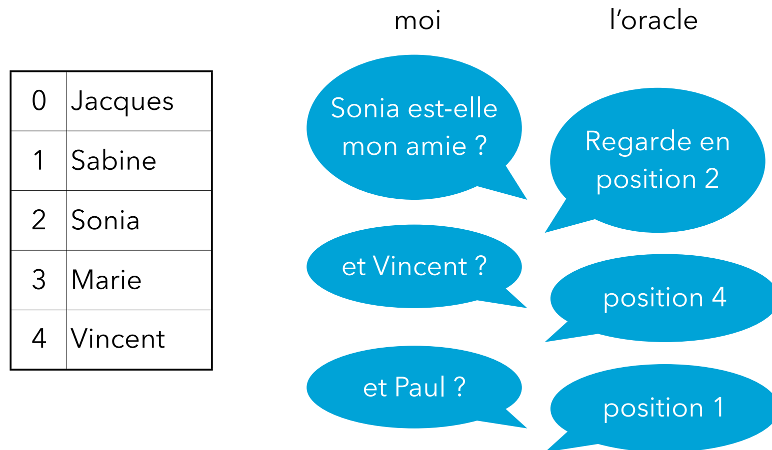
On garde l'ensemble de nos n amis dans un tableau de taille n.

Pour savoir si une personne p en fait partie, on donne le nom de p à un oracle, qui nous fournit un nombre i compris entre 0 et n.

Si la i^e entrée de notre tableau contient le nom de p, alors p est notre ami ; sinon, il ne l'est pas.

Notez que si l'oracle est O(1), la recherche l'est aussi !

La solution magique



L'oracle en Java

Aussi surprenant que cela puisse paraître, il est facile de programmer cet oracle-là en Java :

```
int oracle(String s) {
  int res = 0;
  for (int i = 0; i < s.length(); ++i)
    res = res * 3 + code(s.charAt(i));
  return res % 5;
}
```

donne la position du caractère dans l'alphabet (A/a→0, B/b→1, etc.)

Utilisation de l'oracle

```
int oracle(String s) {
  int res = 0;
  for (int i = 0; i < s.length(); ++i)
    res = res * 3 + code(s.charAt(i));
  return res % 5;
}
```

s		S	o	n	i	a	
code		18	14	13	8	0	
res	0	18	68	217	659	1977	% 5 = 2

Fonction de hachage

En informatique, on appelle un tel oracle une **fonction de hachage** (*hashing function*).

Le but d'une fonction de hachage est de transformer une valeur d'un certain type - ici une chaîne de caractères - en un entier.

En se basant sur une telle fonction, on peut stocker des ensembles de valeurs dans des tableaux - appelés tables de hachage - comme nous l'avons vu.

Table de hachage

Une **table de hachage** est un tableau contenant un ensembles d'éléments, dont la position est déterminée au moyen d'une fonction de hachage.

La possibilité de déterminer la position d'un élément en lui appliquant la fonction de hachage permet – sous certaines conditions – de fournir les opérations de base sur l'ensemble de valeurs en O(1) !

Hachage parfait

Une fonction de hachage h(x) est dite **parfaite** si elle retourne un entier différent pour tous les éléments auxquels on peut l'appliquer.

La fonction `oracle` vue précédemment était parfaite pour l'ensemble des valeurs qui nous intéressait, à savoir les chaînes « Jacques », « Sabine », « Sonia », « Marie » et « Vincent ».

Collisions de hachage

Dans les – très rares – cas où l'on connaît à l'avance la totalité des valeurs à hacher, il est possible de trouver et d'utiliser une fonction de hachage parfaite.

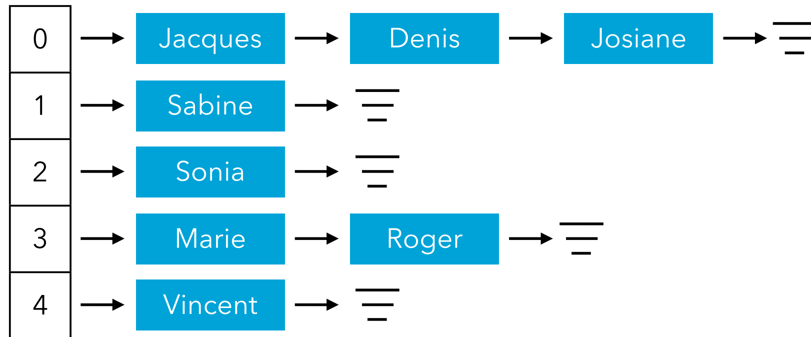
Dans les autres cas, il faut trouver un moyen pour gérer les **collisions de hachage**, c'est-à-dire les cas où plusieurs éléments différents ont la même valeur de hachage.

Gestion des collisions

Si on désire ajouter les noms « Josiane », « Denis » et « Roger » à notre ensemble d'amis, plusieurs collisions se produisent. Par exemple, la fonction de hachage `oracle` utilisée jusqu'à présent produit la valeur 0 pour « Josiane », comme pour « Jacques ».

Une solution simple pour gérer ces collisions consiste à stocker des listes de valeurs dans la table, p.ex. des listes chaînées.

Gestion par chaînage



Propriétés du hachage

Une fonction de hachage h(x) doit absolument satisfaire les deux propriétés suivantes pour être correcte :

- deux éléments égaux doivent avoir la même valeur de hachage [$x = y \Rightarrow h(x) = h(y)$],
- elle doit « bien répartir » les éléments – une notion très difficile à spécifier formellement.

Note : deux éléments différents peuvent avoir la même valeur de hachage ! [$h(x) = h(y) \not\Rightarrow x = y$]

Dégénérescence

Si la fonction de hachage utilisée ne répartit pas bien les éléments, il est possible que la table de hachage dégénère en liste chaînée !

Exemple de très mauvaise fonction de hachage, à éviter absolument : toute fonction constante $h(x) = c$

Tables de hachage en Java

Vue d'ensemble

La mise en œuvre des tables de hachage est simple :

- on représente la table par un tableau de listes chaînées,
- pour rechercher, ajouter ou supprimer un élément, on calcule sa valeur de hachage h puis on recherche, ajoute ou supprime l'élément dans la h^e liste chaînée.

Cela implique de pouvoir effectuer les opérations suivantes sur les éléments placés dans la table :

1. obtenir la valeur de hachage d'un élément,
2. tester si deux éléments sont égaux.

Cela est très simple en Java puisque la classe `Object` possède les méthodes `hashCode` et `equals` qui permettent précisément cela.

La méthode equals

La classe `Object` définit une méthode `equals` qui teste si l'objet auquel on l'applique est égal à celui passé en argument :

```
public boolean equals(Object that)
```

Par défaut, cette méthode vérifie si les deux objets sont physiquement les mêmes – c-à-d s'ils résultent du même `new`.

Il est bien entendu possible de redéfinir cette méthode dans une sous-classe, afin d'offrir une autre notion d'égalité.

La méthode hashCode

La classe `Object` définit une méthode `hashCode` qui fournit une valeur de hachage pour l'objet auquel on l'applique :

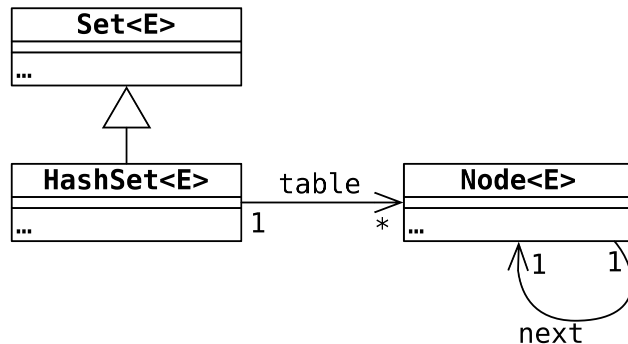
```
public int hashCode()
```

Par défaut, cette méthode fournit une valeur qui dépend de l'identité de l'objet. Cela signifie que deux objets résultant de deux `new` différents auront en général une valeur de hachage différente, même si leur contenu est identique !

Vrai, faux, indéterminé ?

1. `(new Object()).equals(new Object())`
2. `(new Object()).hashCode()`
   `== (new Object()).hashCode()`
3. `Object o = new Object();`
   `o.hashCode() == o.hashCode()`
4. `Object o1 = …, o2 = …;`
   `(o1.hashCode() == o2.hashCode()) ⇒ o1.equals(o2)`
5. `Object o1 = …, o2 = …;`
   `o1.equals(o2) ⇒ (o1.hashCode() == o2.hashCode())`
6. `"bonjour".equals("bon" + "jour")`
7. `"bonjour".hashCode() == ("bon" + "jour").hashCode()`

Diagramme de classes



Code (1)

```
public class HashSet<E> implements Set<E> {
  private Node<E>[] table =
    (Node<E>[])new Node[20];
  private int size = 0;
  …
  private static class Node<E> {
    public Node<E> next;
    public final E elem;
    public Node(Node<E> next, E elem) { … }
  }
}
```


Code (2)

```
class HashSet<E> implements Set<E> {
  private Node<E>[] table = …;
  private int size = 0;
  …
  public void add(E e) {
    int h = Math.abs(e.hashCode())
            % table.length;
    Node<E> n = table[h];
    while (n != null && !n.elem.equals(e))
      n = n.next;
    if (n == null) {
      table[h] = new Node<E>(table[h], e);
      ++size;
    }
  }}
```


Code (3)

```
class HashSet<E> implements Set<E> {
  private Node<E>[] table = …;
  private int size = 0;
  …
  public void contains(E e) {
    // ???
  }
}
```

Rehachage

Taille du tableau

Une table de hachage est efficace si et seulement si :

- le tableau n'est pas « plein de trous », sans quoi de la mémoire est gaspillée,
- les listes chaînées sont aussi courtes que possible – idéalement de longueur 1 – sans quoi les opérations deviennent inefficaces.

Pour assurer ces propriétés, il faut s'assurer que le tableau soit toujours de « bonne » dimension.

Rehachage

Un tableau de taille fixe tel que nous l'avons utilisé jusqu'à présent ne saurait convenir !

Lorsque le nombre d'éléments présents dans la table atteint un certain seuil, il faut le redimensionner et redistribuer les éléments, afin de s'assurer que les listes chaînées gardent une taille raisonnable.

Cette opération s'appelle **rehachage**.

Quand rehacher

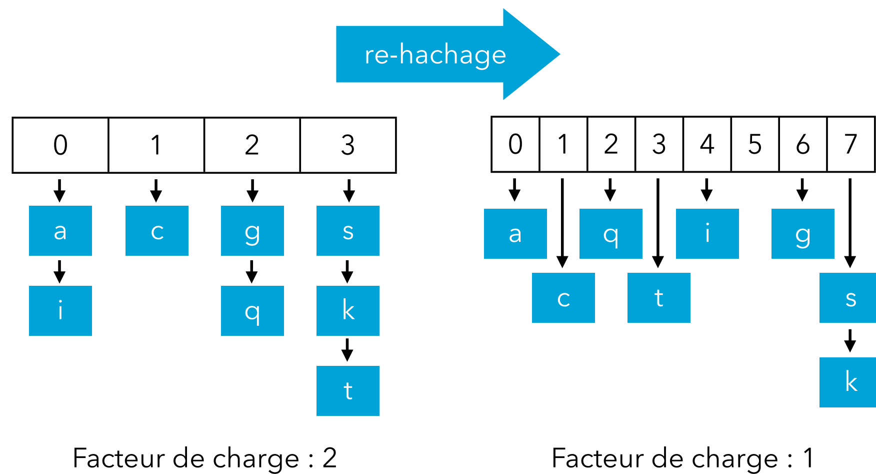
Pour savoir quand re-hacher, on définit la notion de **facteur de charge** (*load factor*).

Le facteur de charge d'une table de hachage est le rapport entre le nombre d'éléments qu'elle contient et sa **capacité**, c-à-d la taille du tableau sous-jacent.

Par exemple, une table ayant une capacité de 100 et contenant 75 éléments a un facteur de charge de 0.75.

Notez que le facteur de charge peut très bien dépasser 1.

Exemple de rehachage



Facteur de charge : 2

Facteur de charge : 1

Tables de hachage dans l'API Java

La classe HashSet

La classe `java.util.HashSet<E>` représente des ensembles au moyen d'une table de hachage.
Pour déterminer la valeur de hachage d'un élément, elle utilise la méthode `hashCode` de la classe `Object` ; pour déterminer si deux éléments sont égaux, elle utilise la méthode `equals`. Pour que la classe `HashSet` se comporte correctement, il est donc impératif que les deux méthodes soient compatibles !

Compatibilité

Nous avons vu qu'une des propriétés qu'une fonction de hachage doit satisfaire est :

$x = y \Rightarrow h(x) = h(y)$

Dans l'API Java, cela signifie que l'implication suivante doit être vraie pour toute paire d'objets x et y :

`x.equals(y)` ⇒ `x.hashCode() == y.hashCode()`

Conseil : redéfinissez *toujours* `equals` et `hashCode` en même temps pour garantir cela.

Compatibles ? (1)

Rappel : compatibles si et seulement si
`x.equals(y) ⇒ x.hashCode() == y.hashCode()`

```
class Employee {
  final String fName, lName;
  int salary;
  public boolean equals(Object that) {
    return (that instanceof Employee)
      && ((Employee)that).fName.equals(fName)
      && ((Employee)that).lName.equals(lName);
  }
  // méthode hashCode héritée de Object
}
```


Compatibles ? (2)

Rappel : compatibles si et seulement si
`x.equals(y) ⇒ x.hashCode() == y.hashCode()`

```
class Employee {
  final String fName, lName;
  int salary;
  public boolean equals(Object that) {
    return (that instanceof Employee)
      && ((Employee)that).fName.equals(fName)
      && ((Employee)that).lName.equals(lName);
  }
  public int hashCode() {
    return fName.hashCode() + lName.hashCode();
  }
}
```


Compatibles ? (3)

Rappel : compatibles si et seulement si
`x.equals(y) ⇒ x.hashCode() == y.hashCode()`

```
class Employee {
  final String fName, lName;
  int salary;
  public boolean equals(Object that) {
    return (that instanceof Employee)
      && ((Employee)that).fName.equals(fName)
      && ((Employee)that).lName.equals(lName);
  }
  public int hashCode() {
    return fName.hashCode();
  }
}
```


Compatibles ? (4)

Rappel : compatibles si et seulement si
`x.equals(y) ⇒ x.hashCode() == y.hashCode()`

```
class Employee {
  final String fName, lName;
  int salary;
  public boolean equals(Object that) {
    return (that instanceof Employee)
      && ((Employee)that).fName.equals(fName)
      && ((Employee)that).lName.equals(lName);
  }
  public int hashCode() {
    return salary;
  }
}
```

Annotations Java (digression)

Annotations

Il est souvent intéressant d'attacher de l'information aux entités qui composent un programme (classes, interfaces, méthodes, champs, etc.).
Par exemple, on peut vouloir dire qu'une méthode d'une classe est obsolète et ne devrait plus être utilisée. Placer un commentaire dans la documentation de la méthode est bien, mais idéalement il faudrait que le compilateur détecte et signale toute utilisation d'une telle méthode.
Java offre pour cela la notion d'**annotation**.

Annotations en Java

En Java, les annotations ont un nom et, éventuellement, des paramètres nommés.
Une annotation est toujours attachée à une déclaration (de classe, d'interface, de méthode, de champ, de variable, etc.). Le nom de l'annotation est précédé d'une arobase (@) et suivi des éventuels paramètres, entre parenthèses.
Exemples :

```
public class LinkedListTest {
  @Test
  public void testGet() { ... }

  @Test(expected = Exception.class)
  public void testGetEmpty() { ... }
}
```


Définition d'annotations

Il est possible de définir de nouvelles annotations, au moyen d'une syntaxe proche de celle utilisée pour définir des interfaces.
Nous n'examinerons pas cette syntaxe ici.
Une annotation est toujours définie dans un paquetage, comme une classe ou une interface. Avant de pouvoir utiliser une annotation, il faut donc importer son nom !
(Exception : les annotations définies dans le paquetage `java.lang`, puisque celui-ci est importé par défaut. On les appelle **annotations prédéfinies**).

@Deprecated

L'annotation prédéfinie `Deprecated` s'attache aux entités (p.ex. les méthodes) qui sont obsolètes et qui ne sont gardées que pour garantir la compatibilité ascendante.

Un avertissement est produit lors de l'utilisation d'une telle entité. Exemple :

```
public class Thread { …
  @Deprecated
  void destroy() { … }
}
public class ThreadUse {
  public void m(Thread t) {
    t.destroy();
  }
}
```

signale l'utilisation d'une méthode obsolète (Eclipse)

@Override

L'annotation prédéfinie `Override` s'attache à une méthode et déclare que celle-ci redéfinit (ou implémente) une méthode héritée. Elle est très utile pour détecter les erreurs bêtes, p.ex.:

```
public class Employee {
  private final String name;
  …
  @Override
  public boolean equals(Employee that) {…}
  @Override
  public int hashCode() {…}
}
```

erreur…

(Conseil : utilisez *toujours* `Override` lors d'une redéfinition !)

@SuppressWarnings

L'annotation `SuppressWarnings` permet de supprimer un avertissement produit par le compilateur.

Exemple :

```
class ArrayList<E> implements List<E> {
  @SuppressWarnings("unchecked")
  private E[] array = (E[])new Object[1];
  …
}
```

Avant de supprimer un avertissement, il faut bien entendu comprendre pourquoi il apparaît, et pourquoi il peut être ignoré !

(Conseil : insérez toujours ces annotations via la commande *Quick Fix* d'Eclipse, après avoir bien réfléchi).

@Test (JUnit)

La bibliothèque JUnit définit l'annotation `@Test`, qui s'attache aux méthodes et permet à JUnit de distinguer celles qui représentent des tests (et qui doivent donc être exécutées) des autres.

Elle accepte un paramètre optionnel, nommé `expected`, qui contient la classe de l'exception que le test devrait lever.

Exemple :

```
public class LinkedListTest {
  @Test
  public void testGet() { ... }

  @Test(expected = Exception.class)
  public void testGetEmpty() { ... }
}
```

Résumé

Comme les arbres de recherche, les tables de hachage permettent de représenter les ensembles d'éléments.
Contrairement aux arbres de recherche, ces tables ne requièrent pas un ordre total sur les éléments. On doit toutefois pouvoir tester l'égalité de deux éléments, et les hacher.
Si la fonction de hachage utilisée est bonne, les opérations de base ont une complexité en O(1) !

* * *

Les annotations permettent d'attacher de l'information aux déclarations du programme.