Collections : Tables associatives

Théorie et pratique de la programmation
Michel Schinz – 2013-03-18

Table associative

Rappel : une **table associative** ou **dictionnaire** (*map* ou *dictionary* en anglais) est une collection associant des valeurs à des clefs.

Les opérations principales d'une table associative sont :

- l'ajout d'une association clef→valeur,
- le remplacement d'une valeur associée à une clef,
- la suppression d'une clef, et donc de la valeur associée,
- la recherche de la valeur associée à une clef.

Exemples

Quelques exemples de tables associatives :

	Clef	Valeur
Encyclopédie	Terme	Définition
Annuaire téléphonique	Nom	Numéro de téléphone
Fonction mathématique	Valeur du domaine	Valeur du codomaine
Tableau (p.ex. en Java)	Index	Elément du tableau

Interface

type des clefs type des valeurs

```
public interface Map<K, V> {
  boolean isEmpty();
  int size();
  void put(K key, V value);
  void remove(K key);
  V get(K key);
  boolean containsKey(K key);
}
```

Utilisation

En faisant l'hypothèse qu'il existe une classe **PN** modélisant les numéros de téléphone (*phone number*), on peut utiliser une table associative pour gérer un annuaire ainsi :

```
Map<String, PN> pBook = …;
pBook.put("Jean", new PN("078 123 45 69"));
pBook.put("Marie", new PN("079 157 78 89"));
…
pBook.put("Ursule", new PN("026 688 87 98"));
System.out.println("Le numéro de Jean est "
                   + pBook.get("Jean"));
pBook.put("Marie", new PN("077 554 12 55"));
pBook.remove("Ursule");
```


Mises en œuvre

Ensembles et tables

Les ensembles et les tables associatives sont très similaires :

- Un ensemble peut être vu comme une table associative dans laquelle aucune information n'est associée aux clefs. C-à-d que seule la présence ou l'absence d'une clef importe.
- A l'inverse, une table associative peut être vue comme un ensemble de paires clef/valeur.

Dès lors, les techniques de mise en œuvre examinées pour les ensembles peuvent être également utilisées pour les tables associatives.

Mises en œuvres

Pour les tables associatives, nous examinerons trois des quatre mises en œuvres examinées pour les ensembles, à savoir :

- les listes,
- les arbres binaires de recherche,
- les tables de hachage.

Paires clef/valeur

Dans chacune des mises en œuvres des tables associatives examinées, les paires clef/valeur jouent un rôle très important.

Pour respecter la terminologie utilisée dans l'API Java, nous appellerons de telles paires des **entrées** (*entries*). Les classes les mettant en œuvre auront toutes l'aspect suivant :

```
class Entry<K, V> {
  public final K key;
  public V value;
  …
}
```

Elles comporteront en plus des liens vers d'autres entrées (successeur pour les listes, enfants pour les arbres).

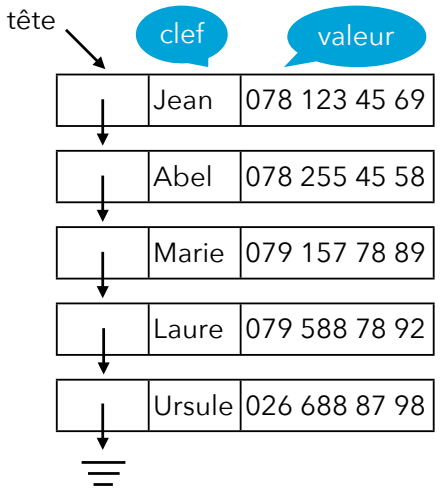
Mise en œuvre 1 : liste associative

Liste associative

Une **liste associative** est une liste d'entrées – composées chacune d'une clef et d'une valeur.

Les entrées d'une liste associative ne sont en général pas triées. Les opérations de base – insertion, suppression, recherche – sont donc réalisées par parcours linéaire de la liste. Leur complexité est ainsi en O(n).

Exemple



Mise en œuvre en Java

Les listes associatives sont mises en œuvre par la classe `AList`, très proche de la classe `LinkedList` présentée plus tôt.

Les entrées, modélisées par la classe imbriquée statique `Entry`, remplacent les nœuds de la liste.

Code (1)

```
class AList<K, V> implements Map<K, V> {
  private Entry<K, V> head = null;
  private int size = 0;
  …
  private static class Entry<K, V> {
    public Entry<K, V> next;
    public final K key;
    public V value;
    public Entry(Entry<K, V> next,
                 K key,
                 V value) { … }
  }
}
```


Code (2)

```
class AList<K, V> implements Map<K, V> {
  private Entry<K, V> head = null;
  private int size = 0;
  …
  public void put(K k, V v) {
    Entry<K, V> e = entryForKey(k);
    if (e != null) {
      e.value = v;
    } else {
      head = new Entry<>(head, k, v);
      ++size;
    }
  }
}
```

remplacement de la valeur existante !

Code (3)

```
class AList<K, V> implements Map<K, V> {
  private Entry<K, V> head = null;
  private int size = 0;
  …
  public V get(K key) {
    Entry<K, V> e = entryForKey(key);
    return (e == null ? null : e.value);
  }
  public boolean containsKey(K key) {
    return entryForKey(key) != null;
  }
  private Entry<K, V> entryForKey(K key) {
    // ???
  }
}
```

Code (4)

```
class AList<K, V> implements Map<K, V> {
  private Entry<K, V> head = null;
  private int size = 0;
  …
  public void remove(K key) {
    // ???
  }
}
```


Mise en œuvre 2 : arbre de recherche

Arbres de recherche

Utiliser un arbre binaire de recherche (a.b.r.) pour mettre en œuvre une table associative est très simple : il suffit de ne prendre que les clefs en considération – et d'ignorer les valeurs – pour la comparaison !

Dans un tel arbre de recherche, les clefs de tous les éléments du fils gauche sont strictement plus petites que la clef de l'élément à la racine ; les clefs de tous les éléments du fils droit sont strictement plus grandes que la clef de l'élément à la racine.

Table associative (a.b.r.)

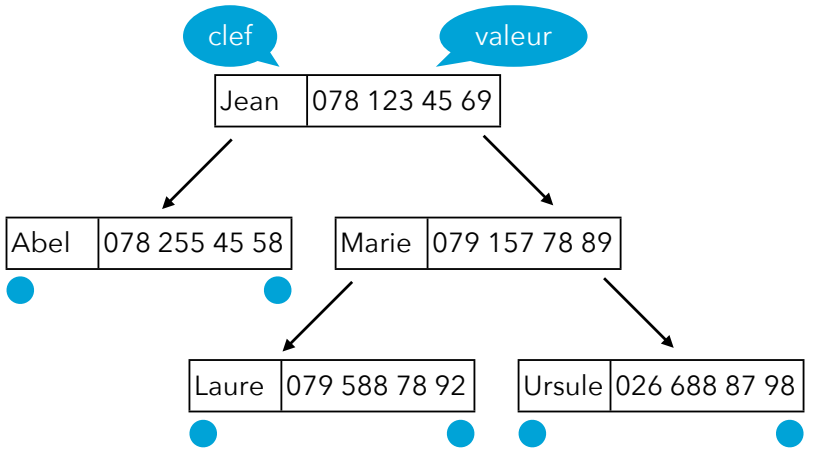
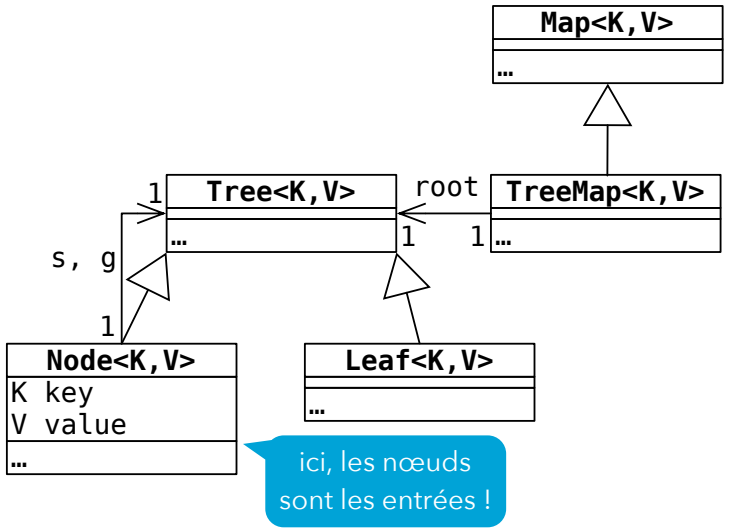


Diagramme de classes



Code (1)

seules les clefs doivent être comparables !

```
class TreeMap<K extends Comparable<K>, V>
    implements Map<K, V> {
  private Tree<K, V> root = new Leaf<K, V>();
  …
  public boolean containsKey(K key) {
    return root.nodeForKey(key) != null;
  }
  …
  // interface Tree, classes Node et Leaf
  // imbriquées statiquement
}
```


Code (2) : arbres

```
private static interface
    Tree<K extends Comparable<K>, V> {
  …
  public Node<K, V> nodeForKey(K key);
}
```


Code (3) : nœuds

```
private static class
    Node<K extends Comparable<K>, V>
      implements Tree<K, V> {
  private final K key;
  private V value;
  private Tree<K, V> s, g;
  …
  public Node<K, V> nodeForKey(K k) {
    int c = k.compareTo(key);
    if (c < 0) return s.nodeForKey(k);
    else if (c > 0) return g.nodeForKey(k);
    else return this;
  }
}
```

Code (4) : feuilles

```
private static class
    Leaf<K extends Comparable<K>, V>
      implements Tree<K, V> {
  …
  public Node<K, V> nodeForKey(K k) {
    return null;
  }
}
```


Mise en œuvre 3 : table de hachage

Table de hachage

Utiliser des tables de hachage pour mettre en œuvre des tables associatives est très simple : il suffit de ne prendre que les clefs en considération – et d'ignorer les valeurs – lors du hachage et de la comparaison.

Table associative (hachage)

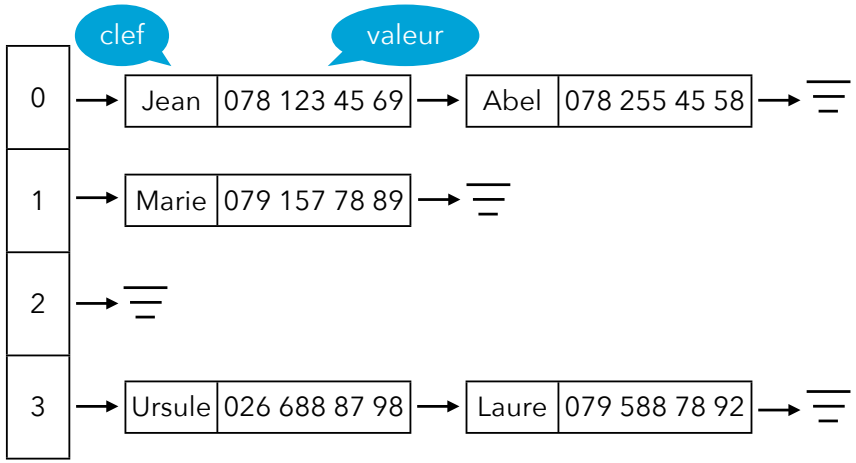
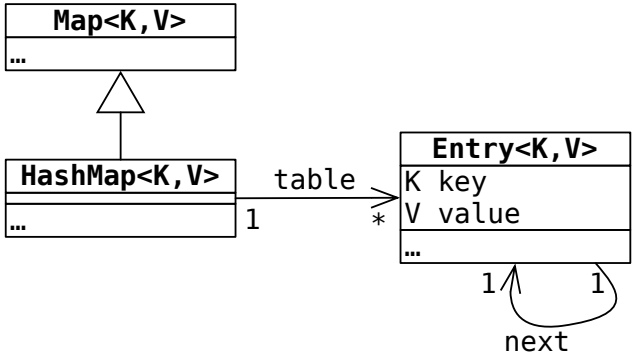


Diagramme de classes



Code (1)

```
class HashMap<K, V> implements Map<K, V> {
  private Entry<K, V>[] table = …;
  private int size = 0;
  …
  private static class Entry<K, V> {
    public Entry<K, V> next;
    public final K key;
    public V value;
    public Entry(Entry<K, V> next,
                 K key,
                 V value) { … }
  }
}
```


Code (2)

```
class HashMap<K, V> implements Map<K, V> {
  private Entry<K, V>[] table = …;
  private int size = 0;
  …
  public void put(K k, V v) {
    int i = Math.abs(k.hashCode())%table.length;
    Entry<K, V> e = entryForKey(k, i);
    if (e != null) {
      e.value = v;
    } else {
      table[i] = new Entry<>(table[i], k, v);
      ++size;
    }
  }
}
```

quasi-identique à la version des listes associatives.

Tables associatives dans l'API Java

Tables associatives de l'API

Le paquetage `java.util` contient la définition d'une interface `Map<K, V>` pour les tables associatives, ainsi que les mises en œuvre suivantes :

- `TreeMap<K, V>` qui utilise des arbres binaires de recherche auto-équilibrants (arbres rouge-noir),
- `HashMap<K, V>` qui utilise la technique du hachage,
- quelques autres que nous n'examinerons pas ici.

Set et Map

Les ensembles et les tables associatives sont tellement similaires que les ensembles de l'API Java sont généralement mis en œuvre au moyen des tables associatives :

- `HashSet<E>` utilise en interne une table de type `HashMap<E, Object>` et associe à chaque élément présent dans l'ensemble un objet sans signification propre.
- `TreeSet<E>` utilise en interne une table de type `TreeMap<E, Object>`.

Itération

Question : pourquoi ni notre interface `Map` ni celle de Java n'étendent l'interface `Iterable` ?

Réponse : parce qu'il n'est pas évident de savoir si on désire parcourir les clefs, les valeurs ou les entrées. En d'autres termes, on ne sait pas forcément choisir entre les trois options suivantes :

- `Map<K, V> extends Iterable<K>`
- `Map<K, V> extends Iterable<V>`
- `Map<K, V> extends Iterable<Entry<K,V>>`

(même si la dernière option est clairement la plus générale et serait la plus logique si on devait faire un choix).

entrySet, keySet, values

Plutôt que d'étendre directement l'interface `Iterable`, l'interface `Map` de Java offre des méthodes pour obtenir des **vues** sur l'ensemble des entrées, des valeurs ou la collection des clefs :

```
interface Map<K, V> {
  …
  Set<Map.Entry<K, V>> entrySet();
  Set<K> keySet();
  Collection<V> values();
}
```

Ces vues permettent d'une part de parcourir les éléments de la table, mais aussi de les modifier.

Parcours via les vues

Au moyen de la vue fournie par la méthode `entrySet` il est facile d'afficher le contenu de notre annuaire téléphonique :

```
Map<String, PN> pBook = …;
pBook.put("Jean", new PN("078 123 45 69"));
pBook.put("Marie",new PN("079 157 78 89"));
…
for (Map.Entry<String, PN> e:
       pBook.entrySet()) {
  System.out.println("Le numéro de "+
    e.getKey() +" est "+ e.getValue());
}
```


Modification via les vues

Les trois vues fournies par les tables associatives Java offrent la possibilité de supprimer des éléments dans la table elle-même, via les méthodes des vues ou de leurs itérateurs.

Par exemple, pour supprimer toutes les personnes dont le nom commence par J de notre annuaire, on peut écrire :

```
Iterator<String> it =
  pBook.keySet().iterator();
while (it.hasNext()) {
  String n = it.next();
  if (n.startsWith("J")) {
    it.remove();
  }
}
```


Clefs modifiables

Eléments/clefs modifiables

Stocker dans un ensemble des éléments modifiables dont les méthodes `hashCode`, `equals` et/ou `compareTo` dépendent de l'état est une très très mauvaise idée...

Pour la même raison, utiliser de tels éléments comme clefs de tables associatives doit être évité à tout prix.

Points modifiables

Pour illustrer le problème, utilisons une classe de points modifiables (une très mauvaise idée en soi !):

```
class MutablePoint {
  public double x, y;
  public boolean equals(Object that) {
    return (that instanceof MutablePoint)
     && (this.x == ((MutablePoint)that).x)
     && (this.y == ((MutablePoint)that).y);
  }
  public int hashCode() {
    return (int)x * 3 + (int)y;
  }
}
```


Eléments modifiables

Question : qu'affiche le programme ci-dessous et pourquoi ?

```
Set<MutablePoint> s = new HashSet<>();
MutablePoint p = new MutablePoint(0, 0);
s.add(p);
System.out.println(s.contains(p));
p.x = 0.1;
System.out.println(s.contains(p));
p.x = 1;
System.out.println(s.contains(p));
p.x = 0;
System.out.println(s.contains(p));
```


Eléments/clefs modifiables

Règle : ne jamais stocker dans un ensemble (ou utiliser comme clef de table associative) des objets dont la valeur de hachage (méthode `hashCode`) ou la notion d'égalité (méthodes `equals` ou `compareTo`) dépend de données modifiables.

A vrai dire, de tels objets sont mal conçus au départ : ni la valeur de hachage ni la notion d'égalité ne devraient dépendre de l'état des objets.

Malheureusement, les exemple de tels objets sont nombreux, à commencer par les instances de `HashMap`, `HashSet`, `TreeMap` et `TreeSet` de Java eux-même !

Résumé

Une table associative est une collection associant une valeur à une clef.

Une table associative peut être mise en œuvre au moyen des mêmes techniques qu'un ensemble, à savoir une liste (associative), un tableau trié, un arbre binaire de recherche ou une table de hachage.