Mise en œuvre des collections : introduction

Pratique de la programmation orientée-objet
Michel Schinz – 2014-03-10

Collections

Nous allons étudier la mise en œuvre des trois principales collections offertes par Java :

1. les listes,
2. les ensembles,
3. les tables associatives.

Etudier ces mises en œuvre est intéressant d'une part car cela permet de savoir laquelle est la plus adaptée à une situation donnée, et d'autre part car cela permet de voir quelques techniques algorithmiques importantes.

Listes

Listes de l'API Java

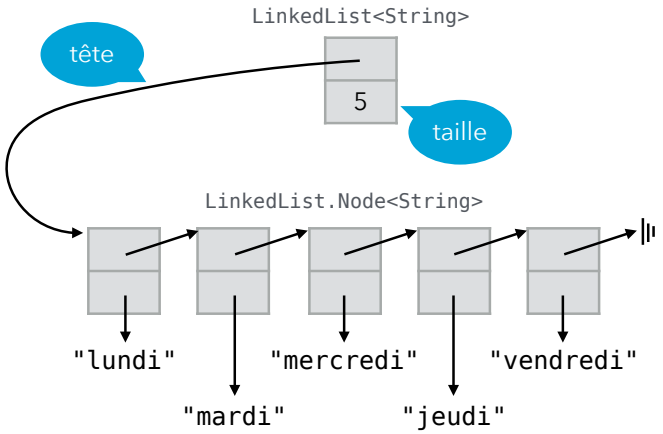
L'API Java offre deux mises en œuvre des listes :

1. la classe `LinkedList`, qui stocke les éléments dans des nœuds chaînés entre eux,
2. la classe `ArrayList`, qui stocke les éléments dans un tableau qui est « redimensionné » (par copie) au besoin.

Liste chaînée

La classe `LinkedList` représente une **liste chaînée**, constituée d'une séquence de **nœuds** liés entre eux et référençant chacun un élément. Le premier nœud de la liste s'appelle la **tête**.
Une liste chaînée est naturellement à **accès séquentiel**, c-à-d que pour accéder à un élément étant donnée sa position, il faut traverser tous les éléments qui se trouvent entre la tête et lui.

Liste chaînée



Types de chaînage

Les nœuds d'une liste peuvent être chaînés de différentes manières.

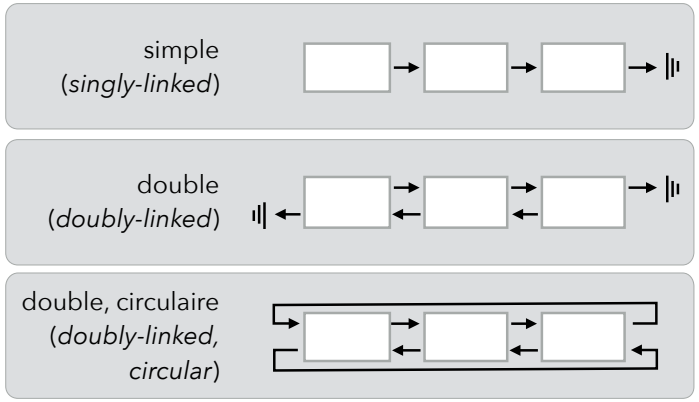
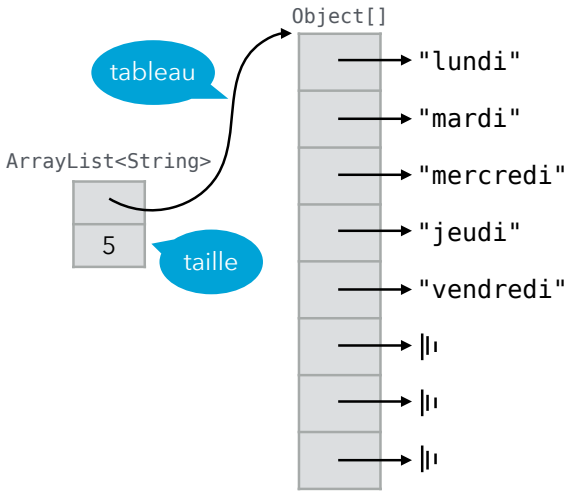


Tableau-liste

La classe `ArrayList` représente un **tableau-liste**, composé d'un tableau (de taille fixe) dans lequel les éléments de la liste sont stockés dans l'ordre. Ce tableau peut avoir une capacité plus grande que celle nécessaire au stockage des éléments de la liste, afin d'éviter les copies incessantes lors d'insertions.
Un tableau-liste est naturellement à **accès aléatoire**, c-à-d qu'on peut accéder directement à un élément étant donnée sa position.

Tableau-liste



Complexité comparée

Op. / Mise en œuvre	Liste chaînée	Tableau-liste
Ajout / suppression	O(1)	O(
Indexation	O(	O(1)
Obtention de la taille	O(1)	O(1)

* O(1) amorti pour l'ajout/suppression en fin de liste.
† O(1) pour les éléments situés aux extrémités de la liste.
‡ si la taille est mémorisée, sinon O(n)

Parcours de liste

Etant données ces deux mises en œuvre très différentes, comment peut-on parcourir les éléments d'une liste en faisant abstraction de la représentation utilisée ?
Une première – mais très mauvaise – idée consiste à utiliser une boucle `for` et la méthode `get`, un peu comme avec un tableau :

```
List<String> l = …;
for (int i = 0; i < l.size(); ++i)
  System.out.println(l.get(i));
```

Pourquoi est-ce une mauvaise idée ?

Parcours via get

Utiliser `get` est une mauvaise idée car, dans le cas des listes chaînées, cette méthode a une complexité de O(n), où n est la taille de la liste.
La boucle d'impression a alors une complexité de O(n^2), ce qui est clairement insatisfaisant pour une boucle qui n'examine chaque élément qu'une seule fois…
Comment faire mieux ?

Parcours par boucle *for-each*

Les listes – et d'autres collections – peuvent heureusement être parcourues au moyen de la boucle *for-each*. Le code d'impression peut donc se récrire ainsi :

```
List<String> l = …;
for (String s: l)
  System.out.println(s);
```

Dans le cas des listes en tout cas, cette boucle est à coup sûr en O(*n*), même avec les listes chaînées.

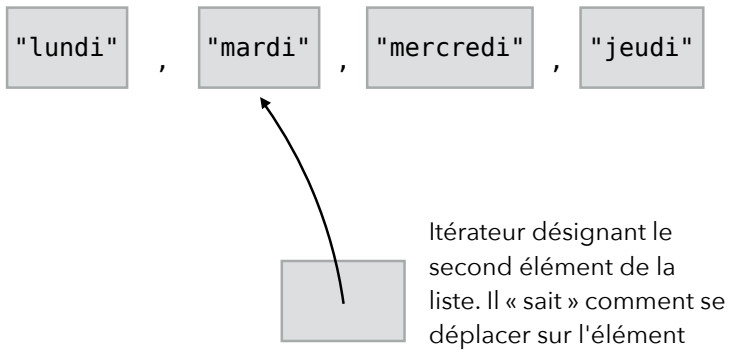
Comment est-ce possible ? Grâce à la notion d'itérateur !

Itérateur

Un **itérateur** ou **curseur** (*iterator* ou *cursor*) est un objet qui désigne un élément d'une collection.

Un itérateur permet d'une part d'obtenir l'élément qu'il désigne, et sait d'autre part se déplacer sur l'élément suivant (et parfois précédent), généralement en temps constant.

Itérateur



Itérateur désignant le second élément de la liste. Il « sait » comment se déplacer sur l'élément suivant.

Interface `Iterator`

L'interface `Iterator` du paquetage `java.util` est définie ainsi :

```
interface Iterator<E> {
  // Retourne vrai ssi il reste au moins un
  // élément (c-à-d qu'on peut appeler next).
  boolean hasNext();
  // Retourne le prochain élément et avance
  // l'itérateur sur le successeur.
  E next();
  // Supprime le dernier élément retourné
  // par next.
  void remove();
}
```

Les méthodes `next` et `remove` lèvent des exceptions en cas d'erreur (pas d'élément suivant / à supprimer).

Parcours par itérateur

La méthode `iterator` permet d'obtenir un itérateur désignant le premier élément de la liste. Une fois cet itérateur obtenu, on peut le faire passer à l'élément suivant grâce à la méthode `next` :

```
List<String> l = …;
Iterator<String> i = l.iterator();
while (i.hasNext()) {
  String s = i.next();
  System.out.println(s);
}
```

Tout comme la boucle basée sur `for`, cette boucle est toujours en O(*n*), même avec les listes chaînées. D'ailleurs, le compilateur Java récrit la boucle `for` en la boucle ci-dessus !

Ensembles

Ensembles

Pour mettre en œuvre un ensemble, on peut simplement placer ses éléments dans une liste. Malheureusement, les opérations de base – ajout, suppression et test d'appartenance d'éléments – ont alors une complexité en O(*n*), ce qui n'est pas très bon. Cela dit, si on ne sait rien sur les éléments stockés dans l'ensemble, on ne peut pas faire mieux !

Heureusement, en imposant quelques contraintes mineures sur les éléments, on peut améliorer nettement l'efficacité des opérations de base.

Ensembles de l'API Java

L'API Java offre deux mises en œuvre des ensembles :

1. la classe `HashSet`, qui stocke les éléments de l'ensemble dans une table de hachage, et
2. la classe `TreeSet`, qui stocke les éléments de l'ensemble dans un arbre binaire de recherche.

Chacune de ces deux mises en œuvre impose une contrainte différente sur les éléments de l'ensemble :

1. `HashSet` demande à ce qu'ils soient « hachables », c'est-à-dire transformables en entiers,
2. `TreeSet` demande à ce qu'ils soient comparables entre eux.

Hachage

Le **hachage** consiste à transformer un objet quelconque en un entier, qu'on appelle sa **valeur de hachage**, via une **fonction de hachage**.
Une fonction de hachage doit être une fonction au sens mathématique, c-à-d qu'appliquée plusieurs fois au même argument, elle doit toujours retourner la même valeur.
De plus, une fonction de hachage doit « bien distribuer » les valeurs, c-à-d qu'elle devrait idéalement retourner des valeurs différentes lorsqu'elle est appliquée à des arguments différents.

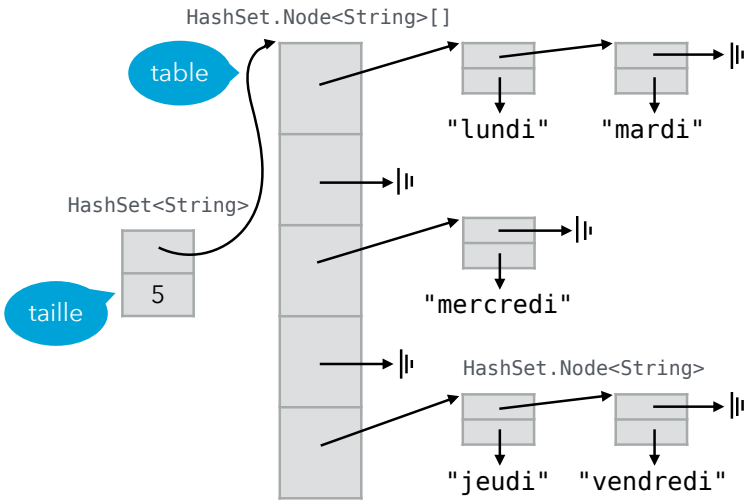
Hachage en Java

En Java, la valeur de hachage d'un objet peut être obtenue au moyen de la méthode `hashCode`, définie dans la classe `Object`.
La version par défaut de `hashCode` retourne un entier qui dépend de l'identité de l'objet, c'est-à-dire de la position qu'il occupe en mémoire. Il est bien entendu possible de redéfinir `hashCode` pour changer la fonction de hachage.
Comme nous le verrons, redéfinir `hashCode` est même obligatoire lorsque `equals` est redéfinie.

Table de hachage

Une **table de hachage** stocke un certain nombre d'éléments dans un tableau de listes chaînées. L'indice de la liste dans lequel un élément est placé dépend de sa valeur de hachage.
Si la fonction de hachage est en O(1) et répartit bien les éléments, et si le tableau a une taille proche du nombre d'éléments contenus dans la table, l'accès à l'un d'eux peut se faire en O(1) !
La classe `HashSet` représente un ensemble au moyen d'une table de hachage.

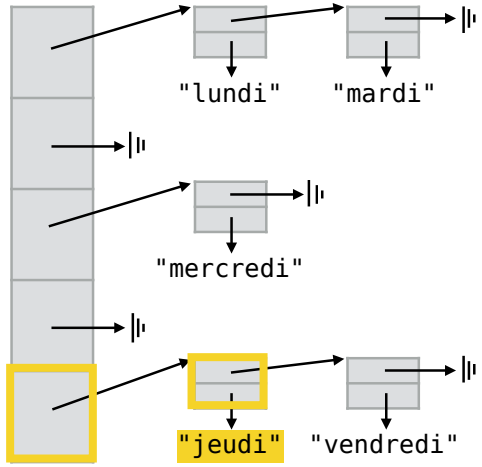
Table de hachage



Recherche dans une t.d.h.

Recherche de la chaîne `"jeudi"` dans la table de hachage ci-contre.

Note : `"jeudi".hashCode()` vaut 101'017'759, valeur que l'on peut ramener à l'intervalle [0;4] au moyen du reste de la division. On obtient alors 4.



hashCode et equals

Les méthodes `hashCode` et `equals` doivent absolument être **compatibles**, c'est-à-dire respecter la loi suivante :

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

En d'autres termes : si deux objets sont égaux au sens de `equals`, alors ils doivent avoir la même valeur de hachage.

A noter que, en général, l'implication inverse n'est pas vraie !

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

Conclusion : les méthodes `hashCode` et `equals` doivent *toujours* être redéfinies simultanément, afin de garantir leur compatibilité

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()`

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();
  }
}
```

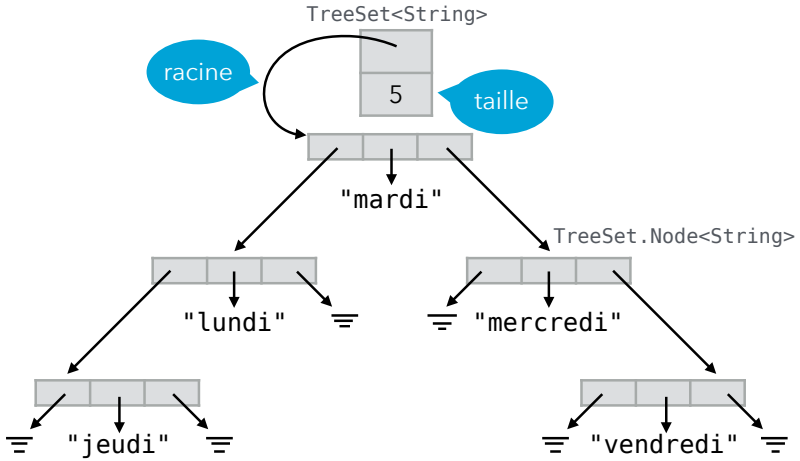

Eléments comparables

Une autre technique que le hachage pour organiser les éléments est le tri. Si les éléments d'un ensemble sont comparables entre eux, alors on peut les maintenir triés et en tirer parti pour accélérer l'accès aux éléments.
Pour ce faire, on peut placer les éléments triés dans un tableau puis utiliser la recherche dichotomique pour y accéder. Le problème de cette solution est que l'ajout et la suppression d'éléments sont en $O(n)$, car ces deux opérations impliquent, en moyenne, le déplacement de la moitié des éléments du tableau.
Une autre manière de faire consiste à stocker les éléments dans des nœuds chaînés entre eux dans un arbre.

Arbre binaire de recherche

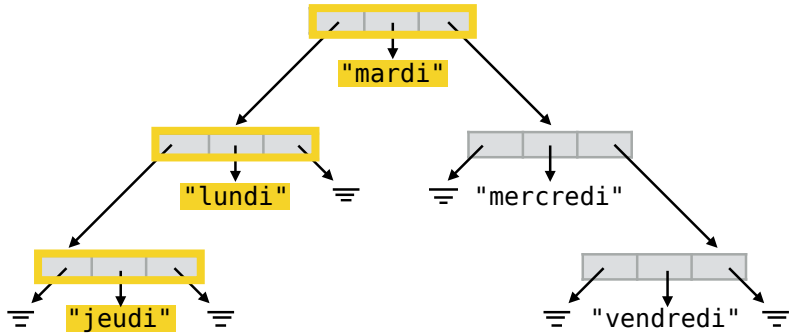
Un **arbre binaire de recherche** est un arbre dont tous les nœuds référencent un élément et deux sous-arbres, qu'on appelle ses fils.
L'arbre est de plus organisé de manière à ce que, pour chaque nœud, tous les éléments du fils gauche soient strictement plus petits que celui du nœud, et tous ceux du fils droit soient strictement plus grands que lui.
La classe `TreeSet` représente un ensemble au moyen d'un arbre binaire de recherche.

Arbre binaire de recherche



Recherche dans un a.b.r.

Recherche de la chaîne `"jeudi"` dans l'arbre binaire de recherche ci-dessous.



Comparaison en Java

Contrairement au hachage, la comparaison d'éléments n'est pas prédéfinie en Java. Au lieu de cela, toute classe dont les éléments sont comparables doit implémenter l'interface `Comparable`, qui définit une unique méthode de comparaison nommée `compareTo`.
Cette méthode retourne un entier dont le signe détermine la relation entre les deux objets. Ainsi `x.compareTo(y)` est un entier :

- négatif si `x` est strictement inférieur à `y`,
- nul si `x` et `y` sont égaux,
- positif si `x` est strictement supérieur à `y`.

Interface `Comparable`

L'interface `Comparable` du paquetage `java.lang` définit la méthode `compareTo` ainsi :

```
interface Comparable<T> {
  int compareTo(T that);
}
```

Le paramètre de type `T` permet de restreindre le type de l'argument de `compareTo`, afin de garantir que seuls des objets effectivement comparables entre eux sont comparés.
Règle : si une classe `C` implémente `Comparable`, elle doit lui passer son propre type, c-à-d `C`, en paramètre.

Exemple : dates

Pour rendre une classe `Date` comparable, il faut lui faire implémenter l'interface `Comparable<Date>` afin de garantir qu'elle ne sera comparée qu'avec d'autres dates :

```
public final class Date
    implements Comparable<Date> {
  // … autres méthodes et champs
  @Override
  public int compareTo(Date that) {
    // … compare les dates this et that.
  }
}
```

Idéalement, `equals` de `Object` devrait aussi utiliser cette technique pour restreindre le type de son argument, mais ce n'est pas le cas pour des raisons historiques.

Exemple : dates

Le paramètre de type de `Comparable` interdit la comparaison de dates avec d'autres valeurs :

```
Date d1 = …, d2 = …;
d1.compareTo(d2);       // valide
d2.compareTo(d1);       // valide
d1.compareTo("hello"); // invalide (erreur
                       // de type)
```

Comme dit, `equals` n'utilise malheureusement pas cette technique, ce qui permet des tests d'égalité clairement insensés :

```
d1.equals("hello");     // valide (mais
                        // toujours faux !)
```


`compareTo` et `equals`

De la même manière que `hashCode` et `equals` doivent être compatibles, `compareTo` et `equals` doivent également l'être.

Dans ce cas, cela signifie que `equals` doit retourner vrai si et seulement si `compareTo` retourne 0 :

`x.equals(y) ⇔ x.compareTo(y) == 0`

Conclusion : si la méthode `compareTo` est définie dans une classe, celle-ci doit également redéfinir `equals` pour garantir leur compatibilité.

Parcours d'ensemble

Etant données ces deux mises en œuvre très différentes, comment peut-on parcourir les éléments d'un ensemble en faisant abstraction de la représentation utilisée ?

Contrairement aux listes, on ne peut même pas faire une boucle qui accède aux élément en fonction de leur index, puisque les éléments d'un ensemble ne sont pas indexés…

Heureusement, la solution est la même qu'avec les listes : il faut utiliser la boucle *for-each*, ou un itérateur.

Parcours d'ensemble

Lorsqu'on parcourt les éléments d'un ensemble de type `HashSet`, ceux-ci sont fournis dans un ordre quelconque, qui peut même changer entre deux exécutions d'un même programme !
Par contre, les ensembles de type `TreeSet` étant naturellement ordonnés, leurs éléments sont toujours parcourus dans l'ordre, du plus petit au plus grand.

Complexité comparée

Op. / Mise en œuvre	Liste	Arbre binaire	Table de hachage
Ajout / suppression	O(n)	O(log	O(1)
Test d'appartenance	O(*n*)	O(log	O(1)

*Peut dégénérer en O(*n*) si l'arbre est totalement déséquilibré.
†Peut dégénérer en O(*n*) si tous les éléments ont la même valeur de hachage.

Tables associatives

Tables associatives

Même si cela peut sembler surprenant au premier abord, les tables associatives sont très similaires aux ensembles.
En effet, une table associative peut être vue comme un ensemble de paires (clef, valeur) où la valeur est ignorée lors de la recherche.
Les mises en œuvre des tables associatives peuvent donc reposer sur les mêmes techniques que les mises en œuvre des ensembles.

Tables associatives Java

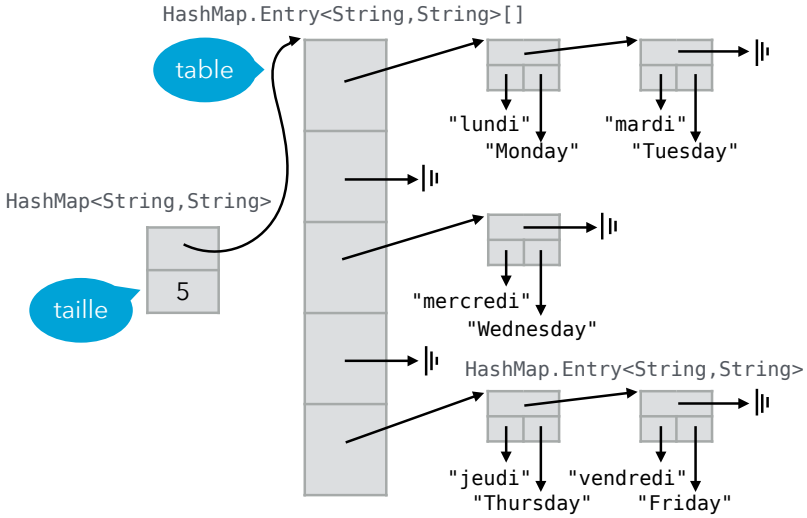
L'API Java offre deux mises en œuvre des tables associatives :

1. la classe `HashMap`, qui stocke les paires (clef, valeur) de la table dans une table de hachage, et
2. la classe `TreeMap`, qui stocke les paires (clef, valeur) de la table dans un arbre binaire de recherche.

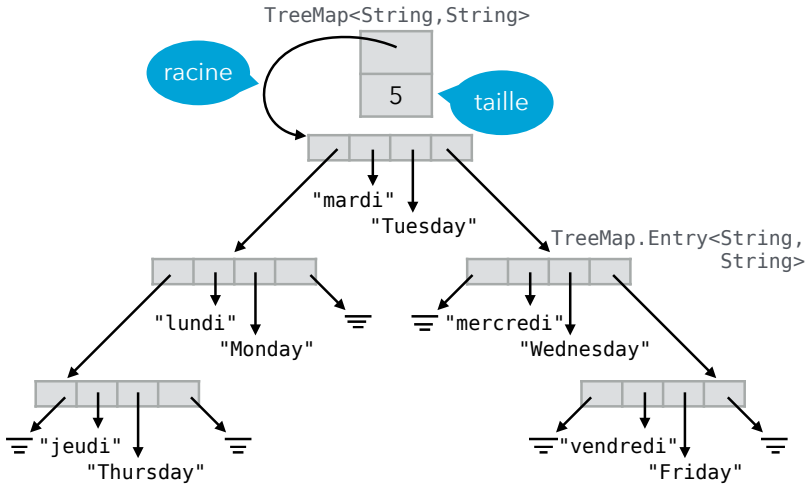
Chacune de ces deux mises en œuvre impose une contrainte différente sur les clefs :

1. `HashMap` demande à ce qu'elles soient « hachables »,
2. `TreeMap` demande à ce qu'elles soient comparables entre elles.

Table de hachage



Arbre binaire de recherche



Itérateur de table

En Java, il n'est pas possible d'itérer directement sur une table associative. Par contre, il existe des méthodes pour obtenir les collections dérivées suivantes, sur lesquelles il est ensuite possible d'itérer :

- l'ensemble des clefs,
- la collection des valeurs,
- l'ensemble des paires (clef, valeur).

Collection des clefs/valeurs

La méthode `keySet` retourne l'ensemble des clefs de la table associative à laquelle on l'applique, tandis que la méthode `values` retourne la collection des valeurs. Elles sont définies ainsi dans l'interface `Map` :

```
interface Map<K, V> {
  // … autres méthodes
  Set<K> keySet();
  Collection<V> values();
}
```


Ensemble des paires

La méthode `entrySet` retourne l'ensemble des paires (clef, valeur) de la table associative à laquelle on l'applique :

```
interface Map<K, V> {
  // … autres méthodes
  Set<Map.Entry<K, V>> entrySet();
}
```

Les paires (clef, valeur) sont des objets de type `Map.Entry`, défini par l'interface suivante, imbriquée dans l'interface `Map` :

```
interface Entry<K, V> {
  K getKey();
  V getValue();
  V setValue(V newValue);
}
```


Résumé

Les listes, les ensembles et les tables associatives sont trois types de collection parmi les plus importants en Java – et dans beaucoup d'autres langages.

Pour chacun de ces trois types de collection, il existe plusieurs mises en œuvres qui offrent la même interface mais diffèrent au point de vue de la complexité des différentes opérations et/ou des contraintes qu'elles placent sur les éléments qu'elles peuvent contenir.

Choisir la collection et sa mise en œuvre appropriée à une situation donnée requiert une bonne compréhension de ces différences.