Mise en œuvre des tables associatives

Pratique de la programmation orientée-objet

Michel Schinz – 2014-04-07

Ensembles/tables associatives

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'est-à-dire 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.

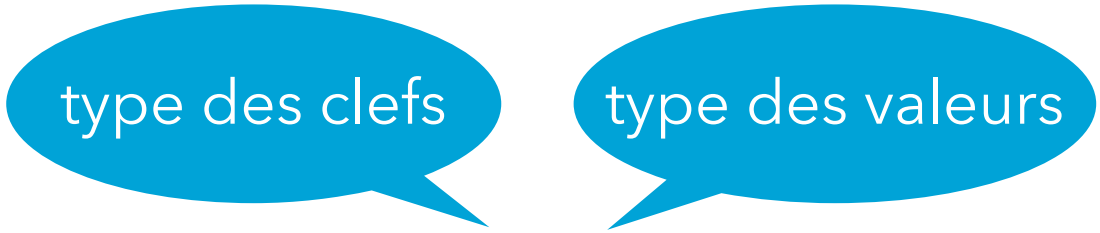
Dès lors, comme nous allons le voir, les techniques de mise en œuvre utilisées pour les ensembles peuvent l'être pour les tables associatives.

Mises en œuvre

Comme pour les ensembles, nous examinerons trois mises en œuvre des tables associatives, les deux dernières étant des versions simplifiées de celles offertes par la bibliothèque Java :

- les listes associatives (classe `ListMap`),
- les tables de hachage (classe `HashMap`),
- les arbres binaires de recherche (classe `TreeMap`).

Interface Map




```
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);
}
```

Note : dans un premier temps, à l'image de l'interface `Map` de la bibliothèque Java, la nôtre n'implémente pas l'interface `Iterable`.

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 – généralement imbriquées statiquement et souvent privées – les mettant en œuvre ont l'aspect suivant :

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

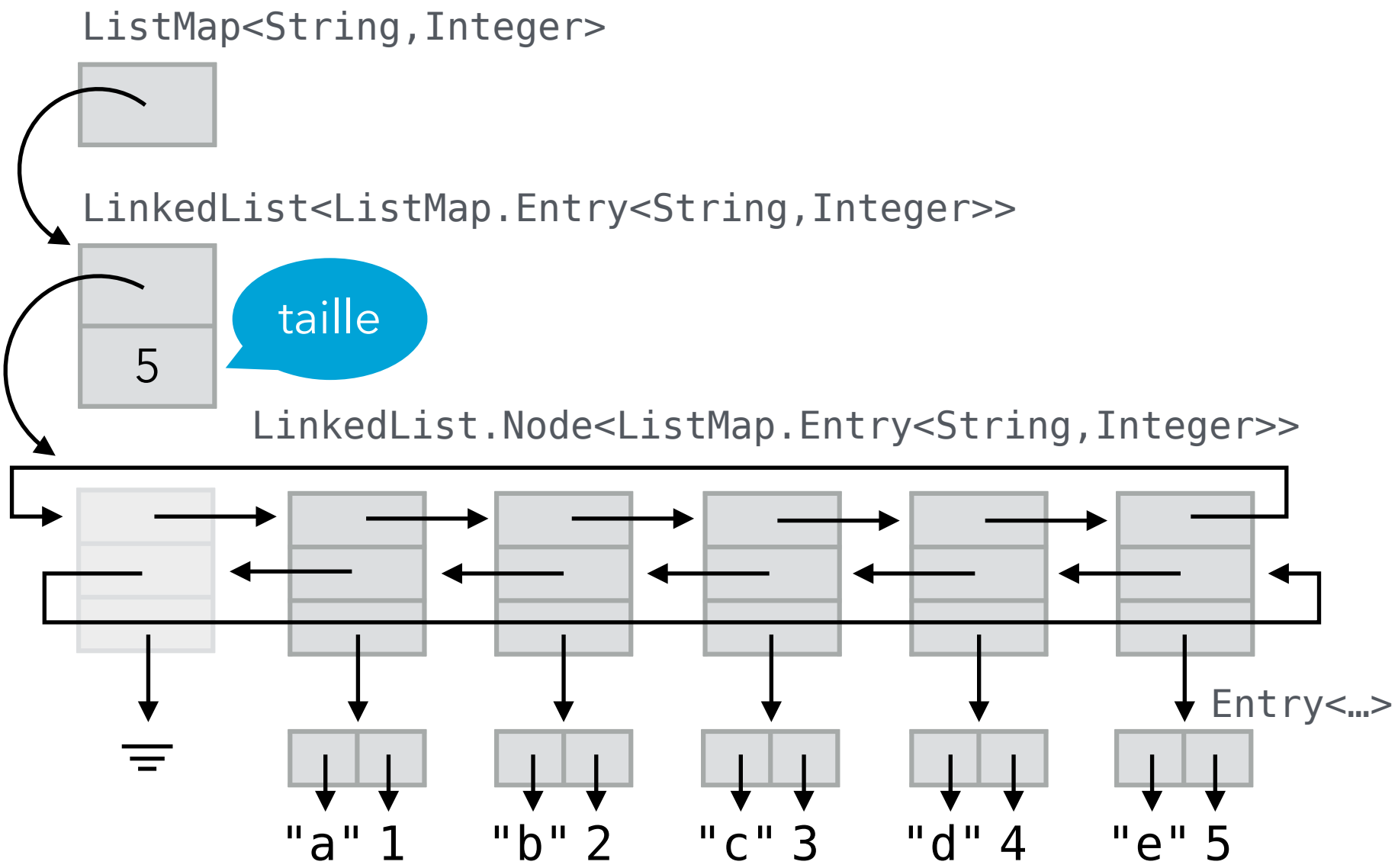
Mise en œuvre n°1 : listes associatives

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 $O(n)$.

Liste associative

(Utilisant une liste doublement chaînée circulaire avec en-tête)



Classe `ListMap`

La classe `ListMap` est très similaire à la classe `ListSet` mais prend deux arguments de type et stocke une liste d'entrées et pas de simples éléments.

```
public class ListMap<K, V>
    implements Map<K, V> {
  private List<Entry<K, V>> entries =
    new LinkedList<>();
  // … méthodes
  private final static class Entry<K, V> {
    public final K key;
    public V value;
    public Entry(K key, V value) { … }
  }
}
```

Méthode `entryFor`

Plusieurs méthode de `ListMap` doivent obtenir, si elle existe, l'entrée possédant une clef donnée. Il est donc utile de définir la méthode auxiliaire `entryFor` dans ce but :

```
public class ListMap<K, V>
    implements Map<K, V> {
  private List<Entry<K, V>> entries = …;
  // … autres méthodes
  private Entry<K, V> entryFor(K key) {
    for (Entry<K, V> e: entries) {
      if (e.key.equals(key))
        return e;
    }
    return null;
  }
}
```

Ajout d'association

L'ajout/remplacement d'une association clef→valeur à une liste associative commence par une recherche, dans la liste des entrées, d'une entrée ayant la même clef que celle de l'association à ajouter.
Si une telle entrée existe, sa valeur est remplacée par celle de l'association à ajouter.
Si une telle entrée n'existe pas, une nouvelle entrée est créée et ajoutée à la liste.

Méthode put

```
public class ListMap<K, V>
    implements Map<K, V> {
  private List<Entry<K, V>> entries = …;
  // … autres méthodes
  public void put(K key, V value) {
    // ???
  }
}
```

Suppression de clef

La suppression d'une clef – et de la valeur qui lui est associée – se fait par parcours de la liste des entrées à la recherche d'une entrée ayant la clef en question.
Si une telle entrée existe, elle est supprimée de la liste.
Sinon, la liste est laissée telle quelle.

Méthode remove

```
public class ListMap<K, V>
    implements Map<K, V> {
  private List<Entry<K, V>> entries = …;
  // … autres méthodes
  public void remove(K key) {
    Iterator<Entry<K, V>> it =
      entries.iterator();
    while (it.hasNext()) {
      if (it.next().key.equals(key)) {
        it.remove();
        return;
      }
    }
  }
}
```

Obtention de valeur

L'obtention de la valeur associée à une clef se fait par recherche, dans la liste des entrées, d'une entrée ayant la clef en question.
Si une telle entrée existe, sa valeur est retournée.
Si une telle entrée n'existe pas, `null` est retourné.

Méthode get

```
public class ListMap<K, V>
    implements Map<K, V> {
  private List<Entry<K, V>> entries = …;
  // … autres méthodes
  public V get(K key) {
    Entry<K, V> e = entryFor(key);
    return e != null ? e.value : null;
  }
}
```

Test de présence de clef

Le test de présence d'une association pour une clef donnée se fait par recherche, dans la liste des entrées, d'une entrée ayant la clef en question.

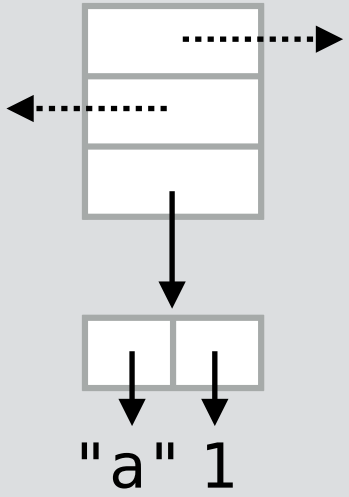
Méthode containsKey

```
public class ListMap<K, V>
    implements Map<K, V> {
  private List<Entry<K, V>> entries = …;
  // … autres méthodes
  public boolean containsKey(K key) {
    return entryFor(key) != null;
  }
}
```

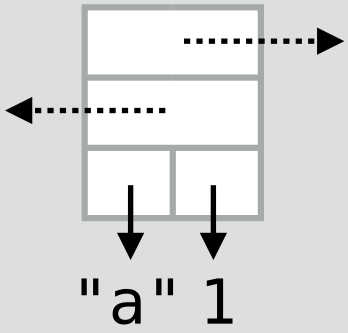
Fusion nœuds / entrées

Pour économiser de la mémoire et du temps, il peut être judicieux de fusionner les nœuds et les entrées, étant donné qu'à un nœud correspond exactement une entrée :

nœuds et entrées séparés



nœuds et entrées fusionnés



Fusion nœuds / entrées

Bien que désirable, la fusion des nœuds et des entrées peut être fastidieuse à mettre en œuvre, car elle implique soit que les nœuds soient extensibles par héritage, soit une réécriture du code de gestion des listes.
En pratique (p.ex. dans la bibliothèque Java), la fusion est généralement mise en œuvre par réécriture du code de gestion des listes, p.ex. dans la classe `HashMap` comme nous allons le voir.

Inefficacité des listes

La mise en œuvre d'une table associative par liste associatives est très simple mais peu efficace, les principales opérations ayant une complexité de $O(n)$.

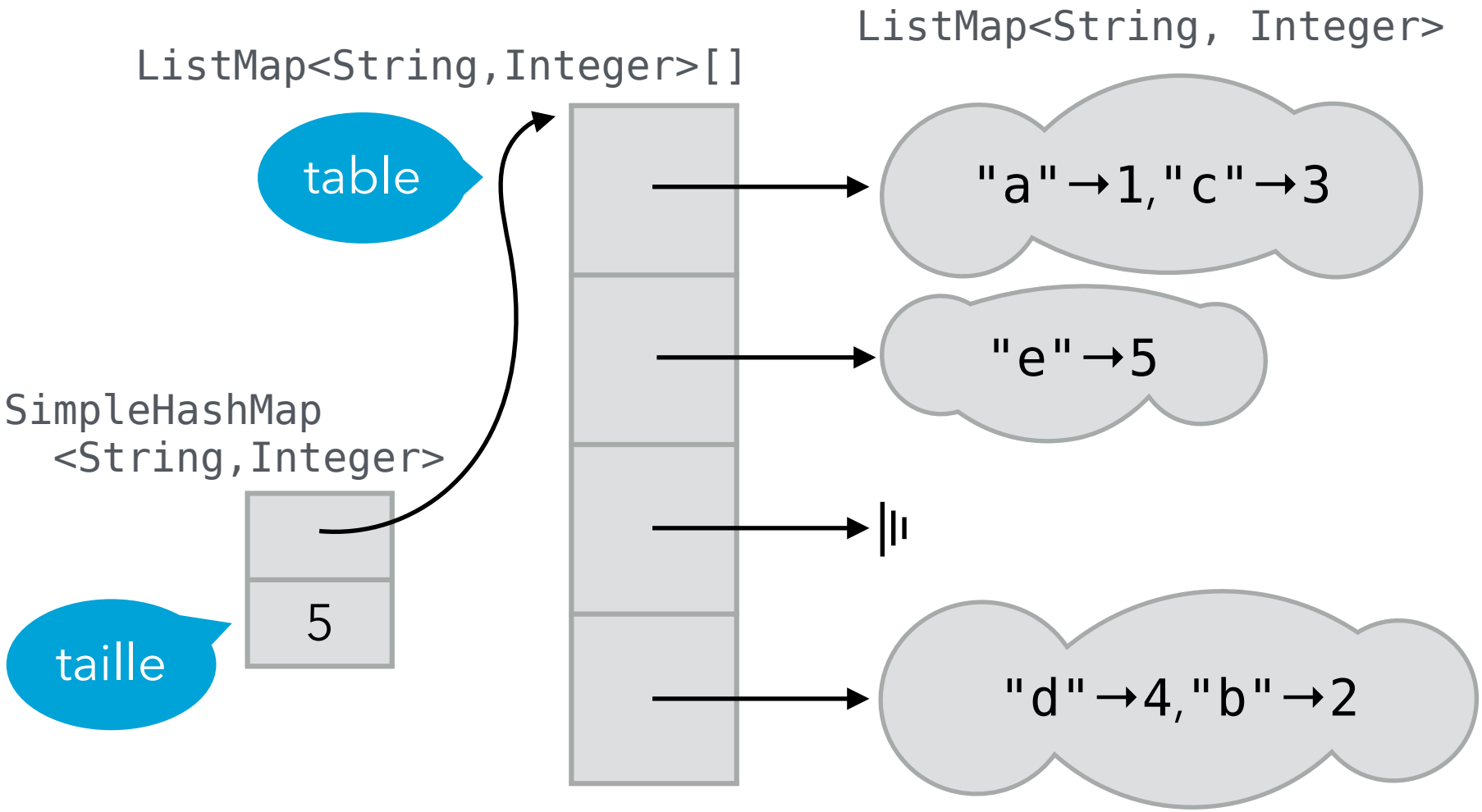
Tout comme pour les ensembles, il est possible de faire nettement mieux en utilisant une table de hachage ou un arbre binaire de recherche.

Mise en œuvre n°2: table de hachage

Table de hachage

Utiliser une table de hachage pour mettre en œuvre une 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 de hachage



Classe `SimpleHashMap`

La classe `SimpleHashMap` ressemble beaucoup à `SimpleHashSet`. Une différence importante est que la méthode `listFor` prend uniquement la clef en argument!

```
public class SimpleHashMap<K, V>
    implements Map<K, V> {
  private int size = …;
  private ListMap<K, V>[] table = …;
  // … méthodes isEmpty, size, put, get, …
  private static <K, V> ListMap<K, V>
      listFor(ListMap<K, V>[] table, K key){
    return table[Math.abs(key.hashCode()
                          % table.length)];
  }
}
```

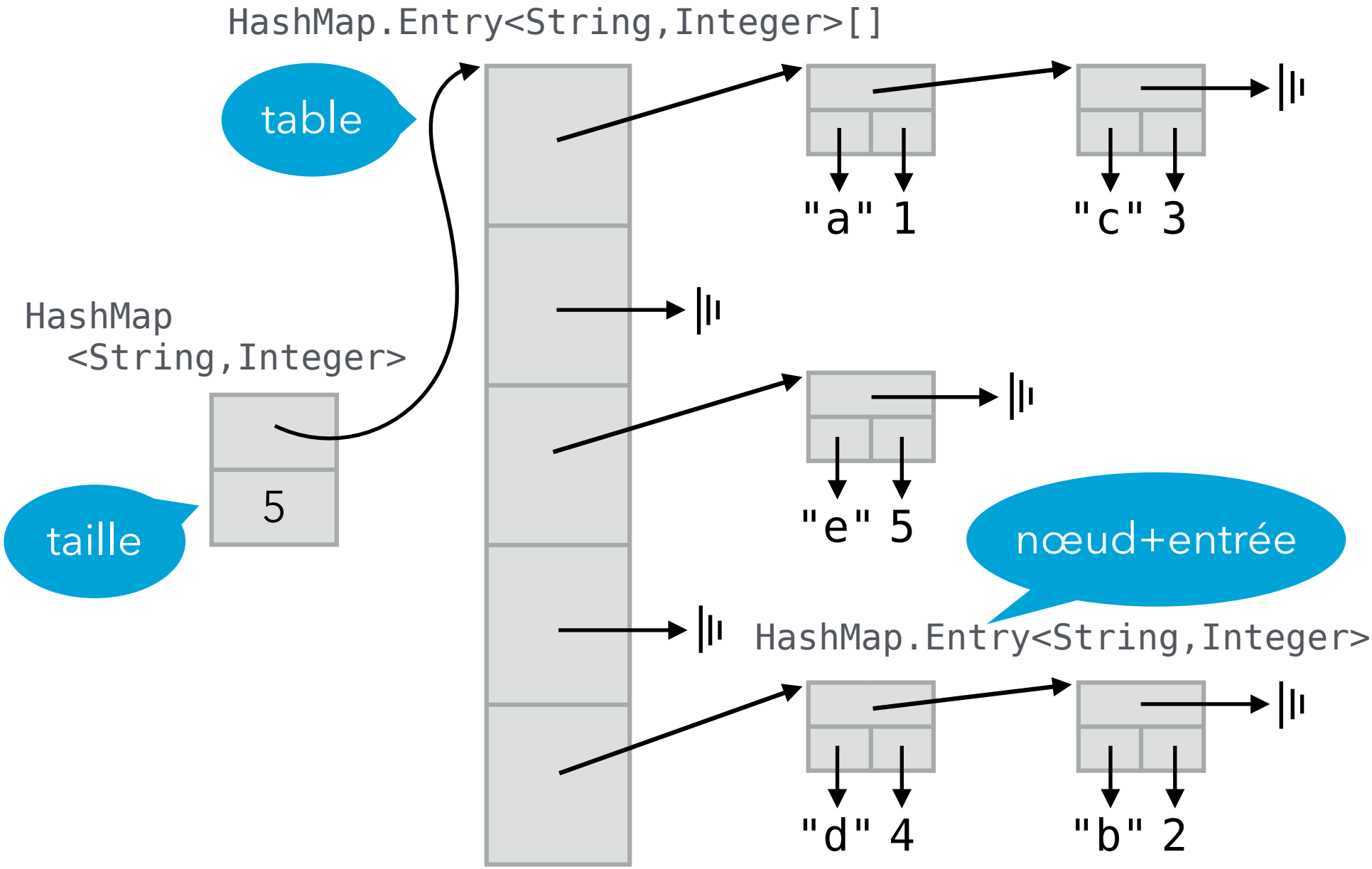
Classe HashMap

La classe `SimpleHashMap` utilise la classe des listes associatives (`ListMap`) pour représenter les listes de la table de hachage. Cette solution est simple mais aussi gourmande en mémoire et – dès lors – plus lente que nécessaire.

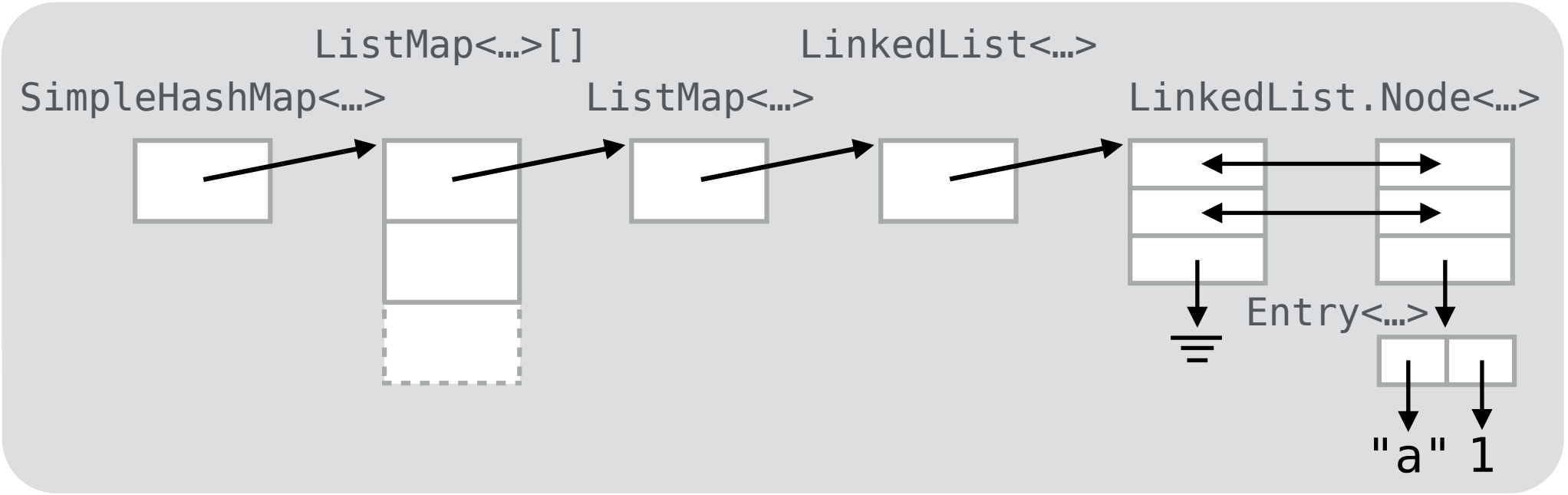
Une mise en œuvre réaliste des tables associatives par table de hachage utilisera plutôt des listes simplement chaînées sans en-tête, dont les nœuds et entrées sont fusionnés et dont le premier nœud est accessible directement depuis le tableau. Ces listes sont gérées par la classe mettant en œuvre les tables associatives.

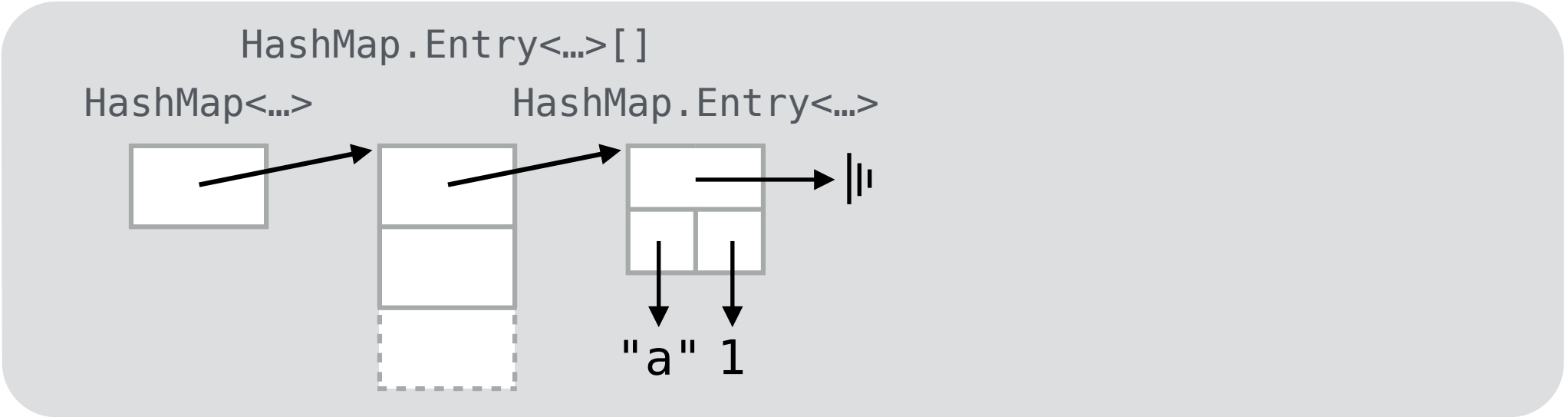
La classe `HashMap` de l'API Java utilise ce principe.

Classe HashMap



SimpleHashMap/HashMap





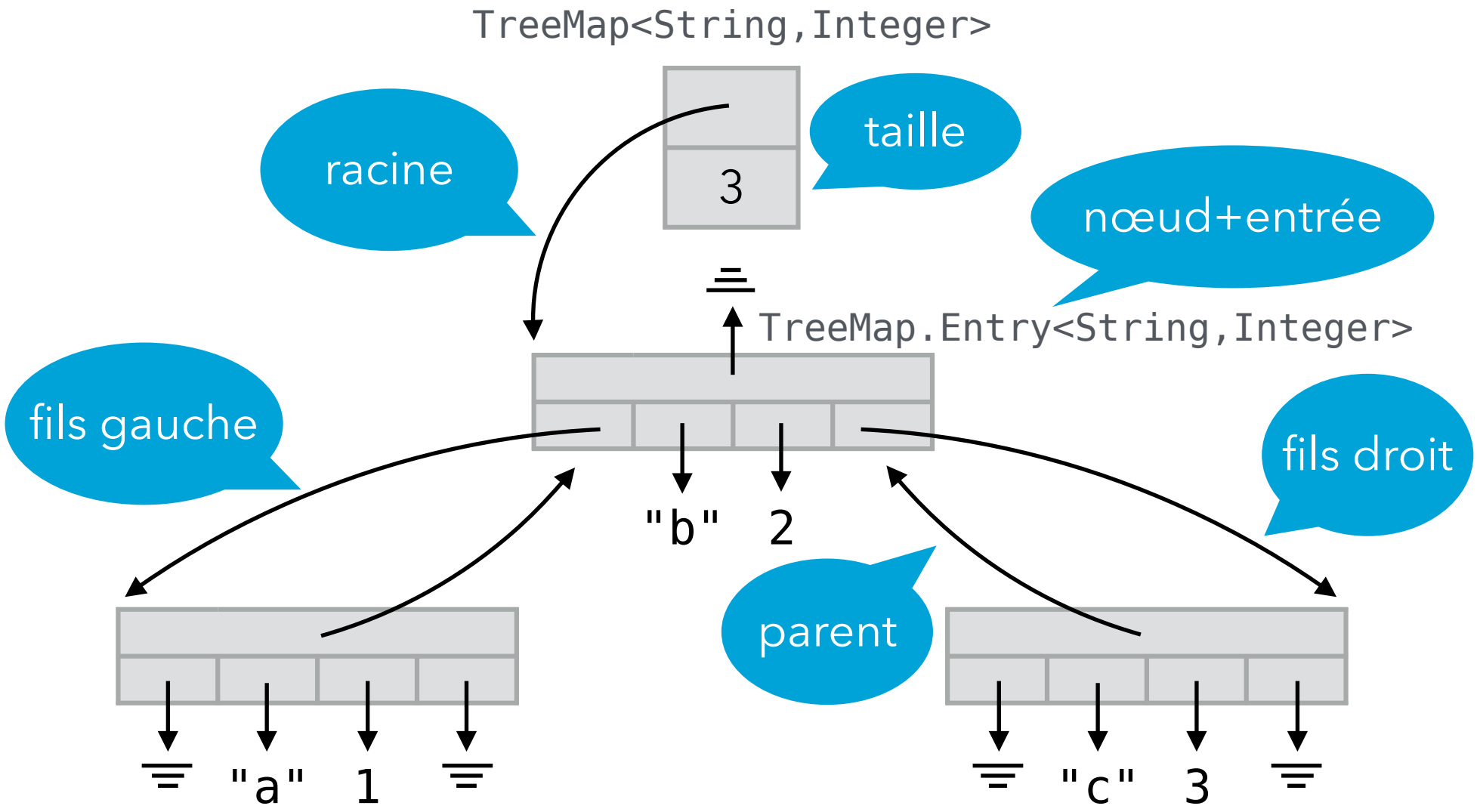
Mise en œuvre n°3: arbre de recherche

Arbre 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 par a.b.r.

(Nœuds « doublement » chaînés, fusionnés avec les entrées)



Classe TreeMap

```
public final class
  TreeMap<K extends Comparable<K>, V>
    implements Map<K, V> {
  private int size = 0;
  private Entry<K, V> root =
    null;
  // … méthodes
  private final static class Entry<K, V> {
    public K key;
    public V value;
    public Entry<K, V> parent;
    public Entry<K, V> smaller, greater;
    // … constructeurs
  }
}
```

seules les clefs doivent être comparables

entrées et nœuds fusionnés

Tables associatives et itération

`Map` et `Iterable`

Au même titre que l'interface Map de la bibliothèque Java, la première version de notre interface `Map` n'étend pas l'interface `Iterable`. La raison en est qu'il n'est pas aussi évident de savoir sur quoi itérer que dans le cas des listes et des ensembles, puisqu'une table associative contient à la fois des clefs et des valeurs.

Cela dit, une table associative est une collection d'associations (ou paires) clef→valeur – les entrées – et il semble donc raisonnable d'offrir l'itération sur ces entrées !

Table associative itérable

Afin de rendre les tables associatives itérables, il faut au préalable définir une interface pour les entrées, que l'on imbrique logiquement dans l'interface des tables :

```
public interface Map<K, V>
    extends Iterable<Map.Entry<K, V>> {
  // … méthodes
  public interface Entry<K, V> {
    K key();
    V value();
  }
}
```

`ListMap` itérable

Une fois `Map` changée, il faut adapter les classes qui l'implémentent en leur ajoutant une méthode `iterator` et en faisant implémenter l'interface `Entry` à leurs entrées.

Exemple pour `ListMap` :

```
public class ListMap<K, V>
    implements Map<K, V> {
  // … méthodes (y compris iterator())
  private final static class Entry<K, V>
      implements Map.Entry<K, V> {
    // … champs et constructeur
    public K key() { return key; }
    public V value() { return value; }
  }
}
```

Itérateur pour `ListMap`

La méthode `iterator` pour `ListMap` devrait être aussi simple à définir que celle pour `ListSet` :

```
public class ListMap<K, V>
    implements Map<K, V> {
  private List<Entry<K, V>> entries =
    new LinkedList<>();
  // … autres méthodes
  public Iterator<Map.Entry<K, V>> iterator(){
    return entries.iterator(); // faux!
  }
}
```

Malheureusement, ce code n'est pas correct du point de vue des types. Pourquoi ?

Itérateur pour `ListMap`

La définition simple de la méthode iterator de la classe `ListMap` est incorrecte car :

- `entries` a le type `List<Entry>` où `Entry` est le type de la classe imbriquée (pas l'interface !),
- dès lors, `entries.iterator` retourne une valeur de type `Iterator<Entry>`,
- or la méthode `iterator` de `ListMap` doit retourner une valeur de type `Iterator<Map.Entry>`.

Les types `Iterator<Entry>` et `Iterator<Map.Entry>` sont incompatibles en Java, malgré le fait que `Entry` implémente `Map.Entry`. Nous verrons plus tard pourquoi…

Itérateur pour `ListMap`

La méthode `iterator` de `ListMap` doit donc retourner un nouvel itérateur dont la définition est triviale :

```
public Iterator<Map.Entry<K, V>> iterator(){
  return new Iterator<Map.Entry<K,V>>() {
    Iterator<Entry<K, V>> it =
      entries.iterator();
    public boolean hasNext() {
      return it.hasNext();
    }
    public Map.Entry<K, V> next() {
      return it.next();
    }
    // … idem pour remove
  };
}
```

correct car `Entry<K,V>` a le type `Map.Entry<K,V>`

Vues de l'API Java

Les concepteurs de la bibliothèque Java ont fait le choix de ne pas rendre les tables associatives itérables. Au lieu de cela, ces tables sont équipées de trois méthodes permettant d'obtenir des **vues** sur l'ensemble des entrées, l'ensemble des clefs ou la collection des valeurs :

```
package java.util;
interface Map<K, V> {
  // … autres méthodes
  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.

Vue `entrySet`

La vue fournie par `entrySet` permet de voir une table associative comme un ensemble d'entrées. Cette vue est identique à notre version itérable de `Map`.

Par exemple, pour itérer sur les entrées d'une table associative de la bibliothèque Java, on écrit :

```
for (Map.Entry<String, Integer> e:
        someMap.entrySet()) {
  // e.getKey() donne la clef,
  // e.getValue() donne la valeur,
  // e.setValue(…) modifie la valeur.
}
```

Modification des vues

Les vues fournies par `entrySet`, `keySet` et `values` sont modifiables, et les modifications sont répercutées sur la table associative.

Par exemple, pour supprimer toutes les entrées dont la clef commence par la lettre `A` d'une table associative `m`, on écrit :

```
Map<String, Integer> m = …;
Iterator<String> it = m.keySet().iterator();
while (it.hasNext()) {
  String k = it.next();
  if (k.startsWith("A"))
    it.remove();
}
```