# Mise en œuvre des ensembles (I)

Pratique de la programmation orientée-objet

Michel Schinz – 2014-03-24

# Mises en œuvre

Nous allons examiner trois mises en œuvre des ensembles, les deux dernières étant des versions simplifiées de celles offertes par la bibliothèque Java :

1. les « listes-ensembles » (classe `ListSet`),
2. les tables de hachage (classe `HashSet`),
3. les arbres binaires de recherche (classe `TreeSet`).

Les classes `ListSet` et `HashSet` sont le sujet de cette leçon tandis que `TreeSet` sera celui de la prochaine.

# Interface Set

L'interface `Set` implémentée par nos mises en œuvre est une simplification de celle de la bibliothèque Java. Elle contient toutefois les méthodes les plus importantes.

```
public interface Set<E> extends Iterable<E>{
  boolean isEmpty();
  int size();
  void add(E elem);
  void remove(E elem);
  boolean contains(E elem);
  Iterator<E> iterator();
}
```

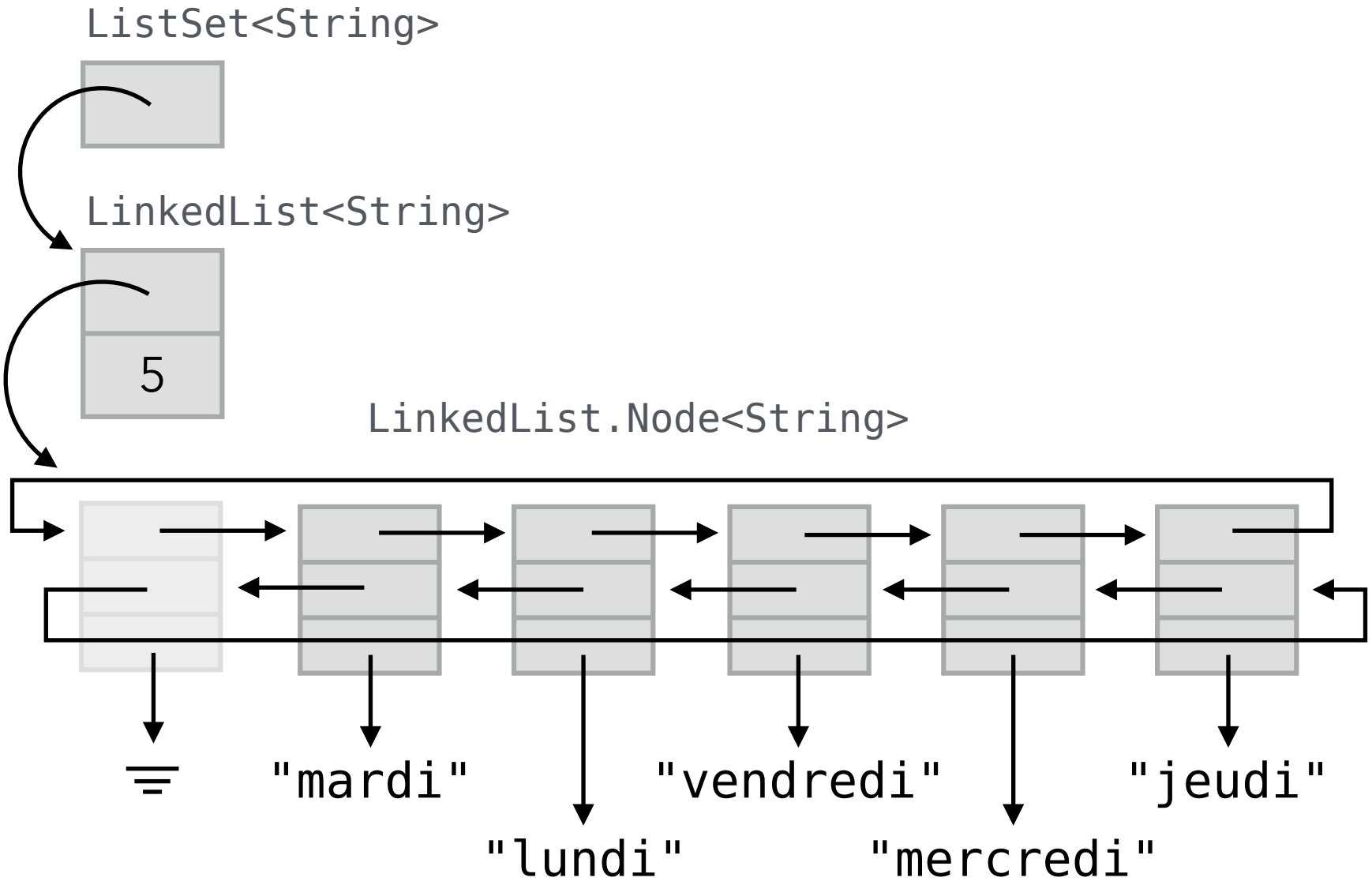
# Mise en œuvre n°1 : listes-ensembles

# Liste-ensemble

Une mise en œuvre simple mais très inefficace des ensembles consiste à stocker les éléments dans une liste (chaînée ou autre). Nous l'appellerons **liste-ensemble**.
Les opérations sur une liste-ensemble sont très faciles à mettre en œuvre au moyen des opérations offertes par la liste sous-jacente.

# Liste-ensemble

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



# Classe `ListSet`

La classe `ListSet` ne contient qu'un champ, la liste sous-jacente. On utilise ici une liste chaînée, mais un tableau-liste conviendrait également.

```
public class ListSet<E> implements Set<E> {
  private List<E> list = new LinkedList<>();

  public boolean isEmpty() {
    return list.isEmpty();
  }
  public int size() {
    return list.size();
  }
  // … autres méthodes
}
```

# Ajout

L'ajout d'un élément à une liste-ensemble se fait en deux étapes :

1. la liste est parcourue pour voir si l'élément à ajouter s'y trouve déjà [complexité : $O(n)$],
2. si tel n'est pas le cas, l'élément est ajouté à la liste, à une position quelconque mais généralement choisie pour que l'ajout puisse se faire en temps constant [complexité : $O(1)$].

Complexité finale : $O(n)$.

# Méthode add

```
public class ListSet<E> implements Set<E> {
  private List<E> list = new LinkedList<>();

  public void add(E newElem) {
    for (E elem: list) {
      if (elem.equals(newElem))
        return;
    }
    list.add(newElem);
  }
  // … autres méthodes
}
```

# Suppression

La suppression d'un élément d'une liste-ensemble se fait par parcours de la liste sous-jacente. Si l'élément est trouvé, il est supprimé, sinon la liste est laissée telle quelle.
Complexité : $O(n)$.

# Méthode remove

```
public class ListSet<E> implements Set<E> {
  private List<E> list = new LinkedList<>();

  public void remove(E elem) {
    Iterator<E> it = list.iterator();
    while (it.hasNext()) {
      if (it.next().equals(elem)) {
        it.remove();
        return;
      }
    }
  }
  // … autres méthodes
}
```

# Test d'appartenance

Le test d'appartenance d'un élément à une liste-ensemble se fait par parcours de la liste sous-jacente. Si l'élément y figure, alors il appartient à l'ensemble, sinon il n'y appartient pas.
Complexité : $O(n)$

# Méthode contains

```
public class ListSet<E> implements Set<E> {
  private List<E> list = new LinkedList<>();

  public boolean contains(E elem) {
    // ???
  }
  // … autres méthodes
}
```

# Itération

L'itération peut se faire directement sur la liste sous-jacente, donc la méthode `iterator` est triviale :

```
public class ListSet<E> implements Set<E> {
  private List<E> list = new LinkedList<>();

  public Iterator<E> iterator() {
    return list.iterator();
  }
  // … autres méthodes
}
```

# Inefficacité des listes

La mise en œuvre des ensembles au moyen des listes a l'avantage d'être très simple, mais le gros inconvénient d'être très peu efficace, les principales opérations ayant une complexité de $O(n)$.
La source de cette inefficacité est la longueur de la liste. S'il était possible de la découper en plusieurs listes contenant chacune un petit nombre d'éléments, et s'il était de plus possible de savoir rapidement à laquelle de ces listes un élément appartient, l'efficacité pourrait être nettement améliorée.
C'est l'idée des tables de hachage.

# Mise en œuvre n°2 : tables de hachage

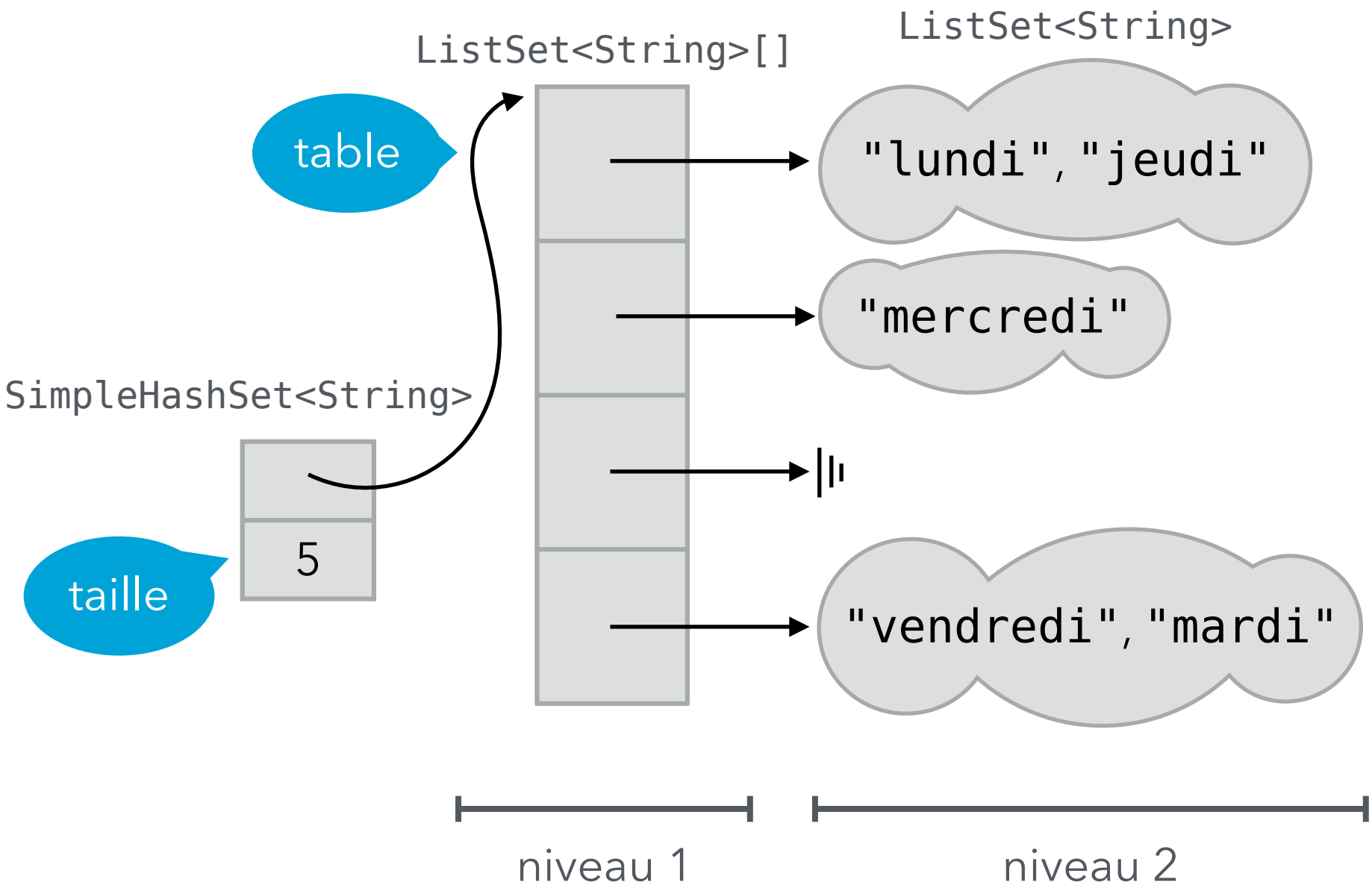
# Table de hachage

Un ensemble peut être mis en œuvre au moyen d'une **table de hachage** qui stocke les éléments de l'ensemble dans un tableau de listes.
Chaque élément de l'ensemble est placé dans la liste dont la position dans le tableau est dérivée de la **valeur de hachage** de cet élément. Cette dernière est calculée par une **fonction de hachage**, qui fait correspondre un entier à un élément.
Une bonne fonction de hachage devrait bien répartir les éléments, c-à-d associer un entier différent à chaque élément. En pratique, cet idéal n'est que rarement atteignable.

# Hachage en Java

En Java, tout objet possède une méthode `hashCode` permettant d'obtenir sa valeur de hachage.
Comme nous l'avons vu, la méthode `hashCode` doit toujours être **compatible** avec la méthode `equals`, c-à-d que deux objets considérés comme égaux (par `equals`) doivent avoir la même valeur de hachage (donnée par `hashCode`).
Les définitions par défaut de `hashCode` et `equals`, héritées de `Object`, sont basées sur l'identité des objets, c-à-d en gros leur position mémoire.
Ces définitions sont compatibles entre-elles mais toute redéfinition doit absolument maintenir cette compatibilité !

# Table de hachage



# Classe SimpleHashSet

```
public class SimpleHashSet<E>
    implements Set<E> {
  private int size;
  private ListSet<E>[] table;

  public SimpleHashSet() {
    ListSet<E>[] table = new ListSet[10];
    for (int i = 0; i < table.length; ++i)
      table[i] = new ListSet<>();
    this.size = 0;
    this.table = table;
  }
  // … autres méthodes
}
```

# Ajout, suppression, test

Les opérations d'ajout, de suppression et de test d'appartenance d'un élément à une table de hachage se font toutes en deux étapes, qui reflètent l'organisation en deux niveaux de la table :

1. l'index de la liste à laquelle l'élément appartient est déterminé en fonction de sa valeur de hachage,
2. l'opération (ajout, suppression, test d'appartenance) est faite dans la liste déterminée par la première étape.

Si la fonction de hachage est en O(1) et répartit assez bien les éléments pour que toutes les listes soient très courtes, toutes ces opérations sont en O(1).

# Etape 1 : indexation

La première étape, qui consiste à déterminer la liste dans laquelle effectuer l'opération, peut être placée dans une méthode privée, nommée p.ex. `listFor` :

```
public class SimpleHashSet<E>
    implements Set<E> {
  private int size;
  private ListSet<E>[] table;
  // … autres méthodes
  private ListSet<E> listFor(E elem) {
    return table[Math.abs(elem.hashCode()
                          % table.length)];
  }
}
```

# Méthode `contains`

La méthode `contains` pour les tables de hachage se définit simplement par composition de la méthode d'indexation `listFor` (étape 1) et de la méthode `contains` sur les listes-ensembles (étape 2).

```
public class SimpleHashSet<E>
    implements Set<E> {

  public boolean contains(E elem) {
    return listFor(elem).contains(elem);
  }
  // … autres méthodes
}
```

# Méthode add

La méthode `add` – et la méthode `remove`, similaire – est légèrement plus complexe car elle doit garantir que la taille stockée dans le champ `size` reste correcte.

```
public class SimpleHashSet<E>
    implements Set<E> {

  public void add(E newElem) {
    ListSet<E> list = listFor(newElem);
    size -= list.size();
    list.add(newElem);
    size += list.size();
  }
  // … autres méthodes
}
```

# Itération

L'itération sur les éléments d'une table de hachage est, en quelques sortes, bidimensionnelle. C'est-à-dire qu'il faut à la fois parcourir le tableau de listes chaînées (1ère dimension), et chacune des listes (2e dimension).
L'itération est réalisée par la classe d'itérateur `SHSIterator`, imbriquée statiquement dans la classe `SimpleHashSet`. Son constructeur prend en paramètre la taille de l'ensemble et la table de hachage. Au moyen de ces deux éléments, il n'est pas très difficile de définir les méthodes `hasNext` et `next` (laissé en exercice).

# Classe SHSIterator

```
public class SimpleHashSet<E>
    implements Set<E> {
  public Iterator<E> iterator() {
    return new SHSIterator<>(table, size);
  }
  // … autres méthodes
  private static final class SHSIterator<E>
      implements Iterator<E> {
    private int remaining; // … autres champs
    public SHSIterator(ListSet<E>[] table,
                       int size)
      { this.remaining = size; … }
    // … méthodes hasNext, next et remove
  }
}
```

# Exercice

Comme on l'a vu précédemment, les méthodes `hashCode` et `equals` doivent être compatibles en Java, c-à-d que pour toute paire d'objets `x` et `y`, l'implication suivante doit être vraie :

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

Notre classe `SimpleHashSet` fonctionne-t-elle également si cette implication n'est pas vraie ? Qu'en est-il de la classe `ListSet` ?

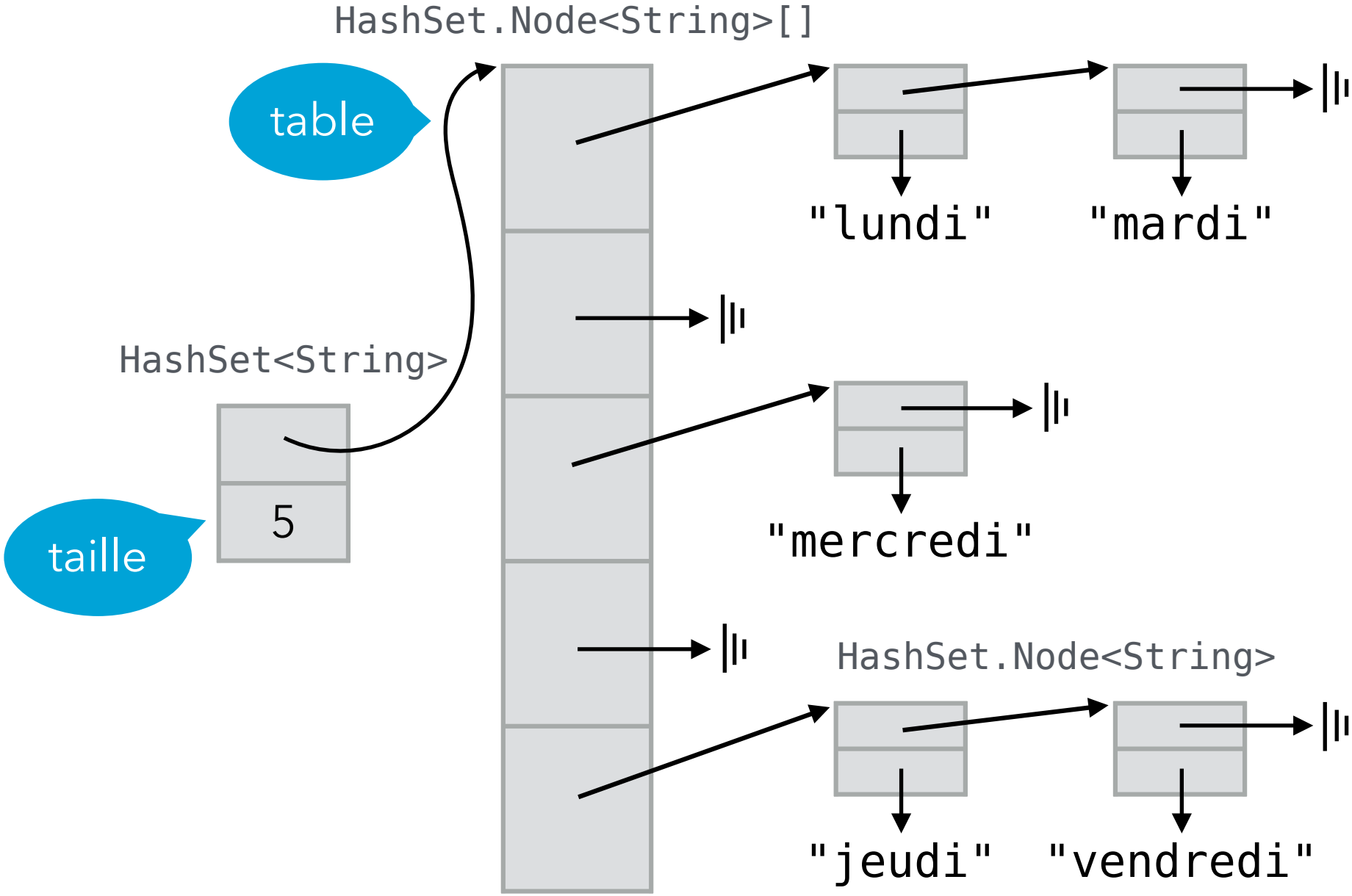
# Classe HashSet

La classe `SimpleHashSet` utilise la classe des listes-ensembles (`ListSet`) 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.

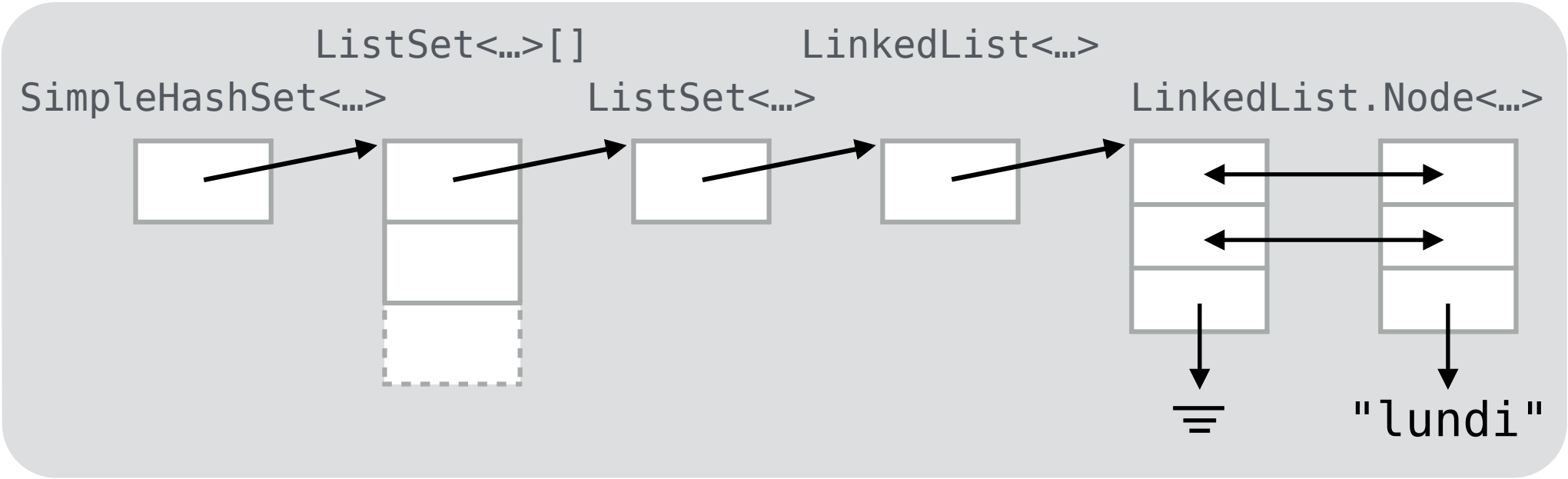
Une mise en œuvre réaliste des ensembles par table de hachage utilisera plutôt des listes simplement chaînées sans en-tête et dont le premier nœud est accessible directement depuis le tableau. Ces listes sont gérées par la classe mettant en œuvre les ensembles.

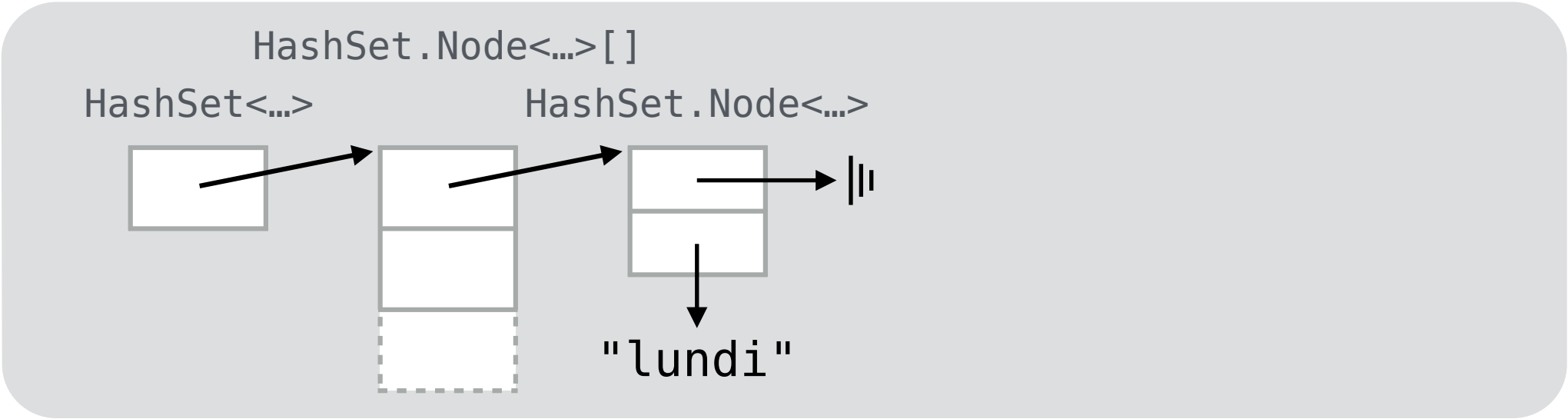
La classe `HashSet` utilise ce principe.

# Classe HashSet



# SimpleHashSet/HashSet



# 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 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.

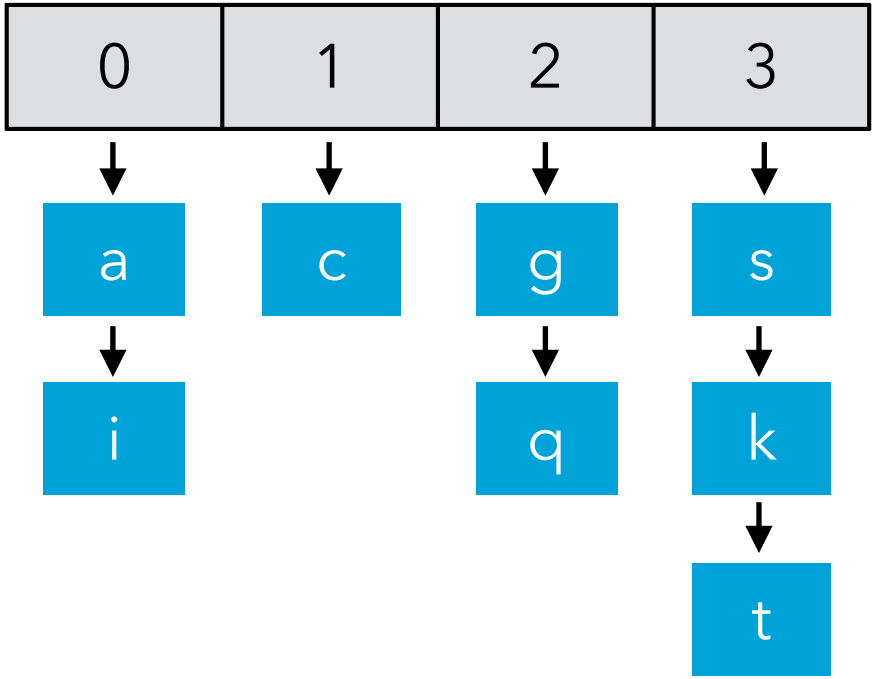
# Rehachage

Un tableau de taille fixe tel que nous l'avons utilisé jusqu'à présent ne saurait convenir !
Lorsque le nombre d'éléments présents dans l'ensemble atteint un certain seuil, il faut le redimensionner et redistribuer les éléments, afin de s'assurer que les listes gardent une taille raisonnable, c-à-d proche de 1.
Cette opération s'appelle **rehachage**.

# Facteur de charge

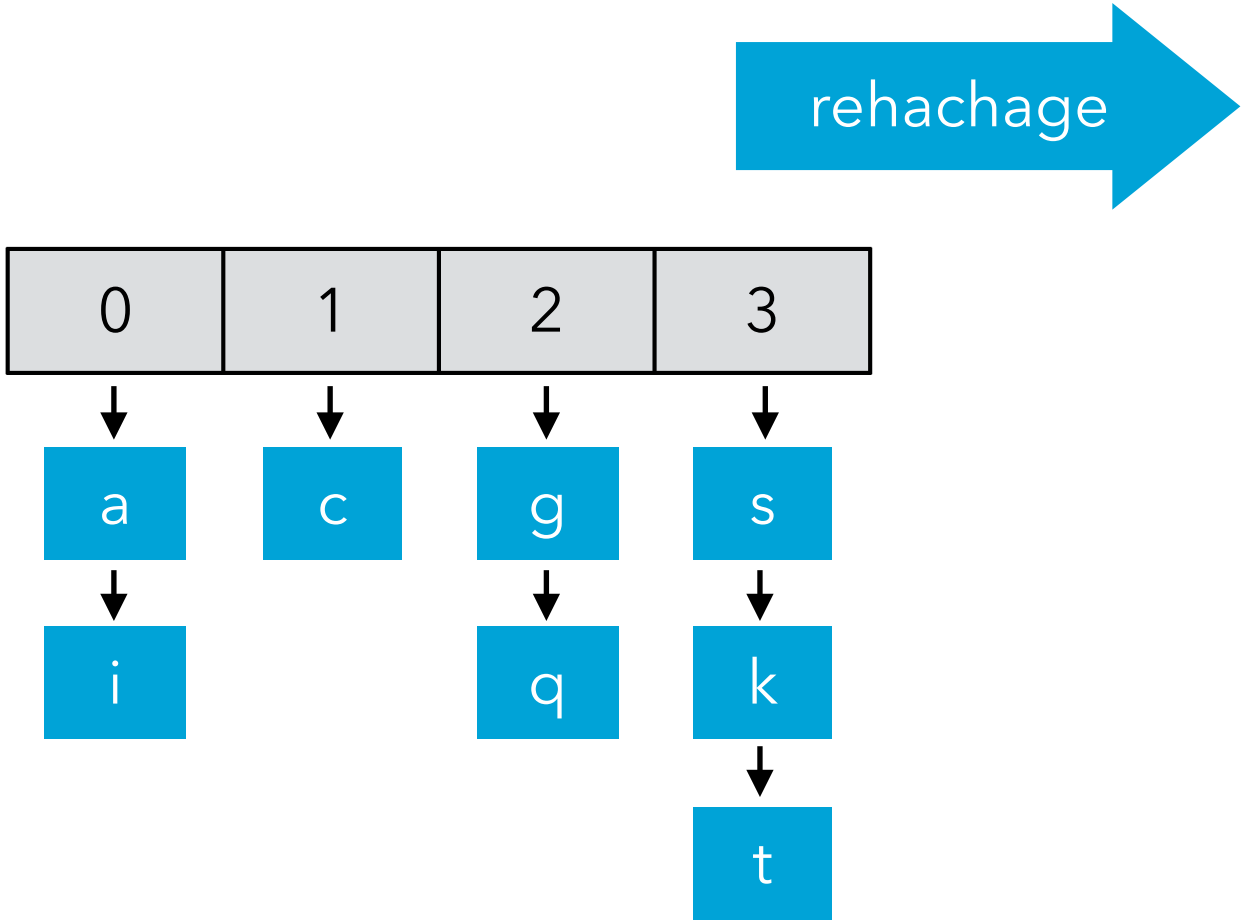
Pour savoir quand re-hacher, on définit la notion de **facteur de charge** (*load factor*).
Le facteur de charge d'une table de hachage est le rapport entre le nombre d'éléments qu'elle contient et sa **capacité**, c-à-d la taille du tableau sous-jacent.
Par exemple, une table ayant une capacité de 100 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



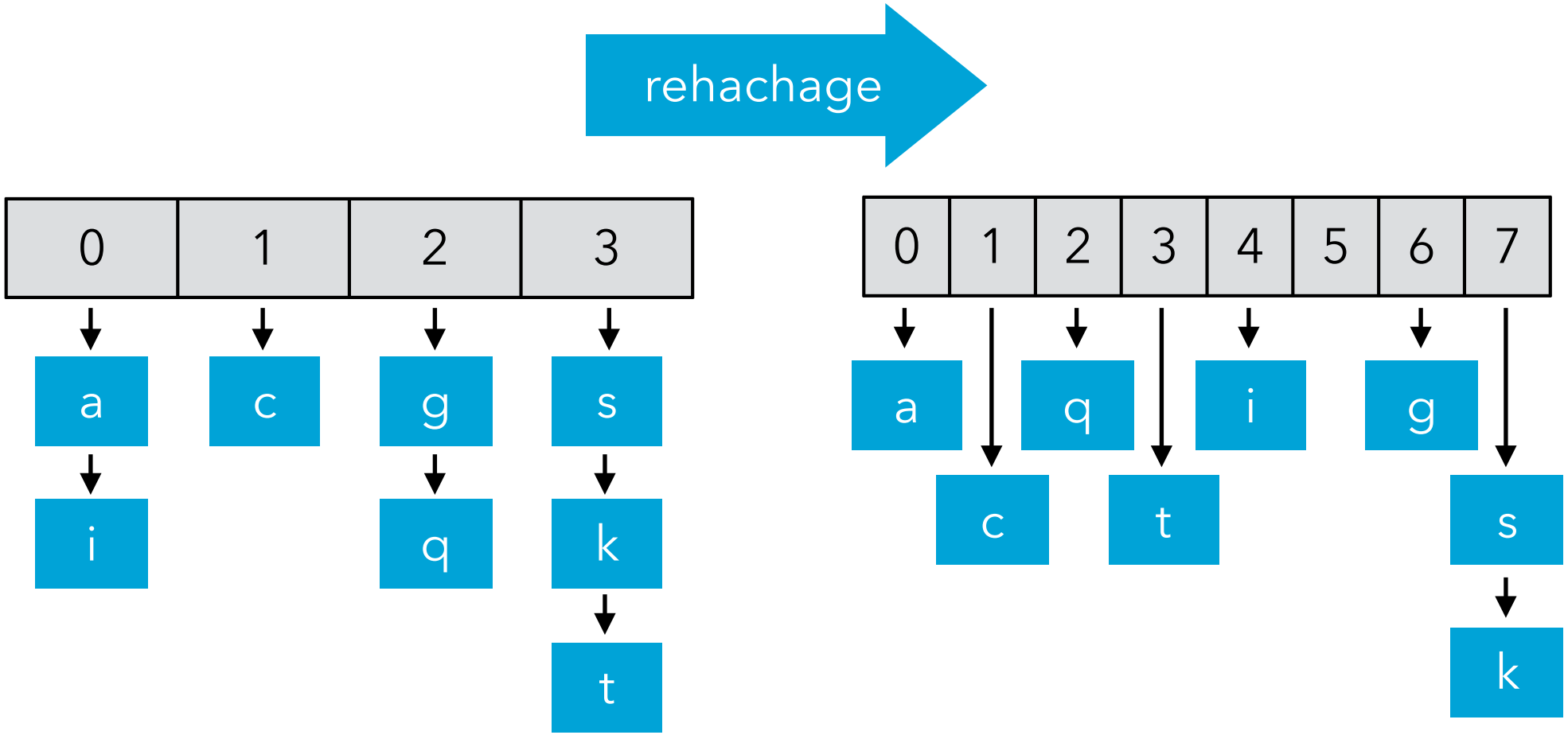

Facteur de charge : 2

# Exemple de rehachage




Facteur de charge : 2

# Exemple de rehachage




Facteur de charge : 2

Facteur de charge : 1

# Classes imbriquées non statiques (digression)

# Itérateur SHSIterator

L'itérateur `SHSIterator`, comme la plupart des itérateurs, doit connaître certaines informations internes de la collection qu'il parcourt – ici le nombre d'éléments de l'ensemble et la table de hachage contenant ces éléments. Ces informations sont passées à son constructeur.
Le code pourrait être simplifié si la classe `SHSIterator` avait directement accès à ces informations. Cela est possible en Java, en déclarant la classe des itérateurs comme non statique.

# Itérateur non statique

En enlevant le modificateur `static` de la classe `SHSIterator`, celle-ci devient une **classe imbriquée** (tout court, et non pas statique).

En tant que classe imbriquée (non statique), cette nouvelle version de la classe `SHSIterator` a accès à :

- tous les membres – champs et méthodes – de sa classe englobante, y compris ceux qui ne sont pas statiques ou qui sont privés,
- tous les paramètres de type de sa classe englobante.

# Itérateur non statique

```
public class SimpleHashSet<E>
    implements Set<E> {
  private int size;
  private ListSet<E>[] table;

  public Iterator<E> iterator() {
    return new SHSIterator();
  }
  // … autres méthodes
  private final class SHSIterator
      implements Iterator<E> {
    private int remaining = size;
    // … méthodes hasNext, next et remove
  }
}
```

plus de paramètres

plus de paramètre de type E

accès direct au champ `size`

# Itérateur anonyme

Java permet même d'aller plus loin en ne nommant pas la classe `SHSIterator`. Son corps est alors placé dans la méthode `iterator` elle-même, dans l'énoncé `new` de création de l'itérateur.

Une telle **classe imbriquée anonyme** est similaire à une classe imbriquée si ce n'est qu'elle n'a pas de nom et que, dès lors, il n'est possible d'en créer des instances qu'à un seul endroit dans le programme.

# Itérateur anonyme

```
public class SimpleHashSet<E>
    implements Set<E> {
  private int size;
  // … autres méthodes
  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private int remaining = size;
      // … méthodes hasNext, next et remove
    };
  }
}
```

# Constructeur anonyme

Les classes anonymes ne peuvent pas avoir de constructeur à proprement parler, mais elles peuvent avoir du code qui est exécuté lors de leur initialisation. Celui-ci doit être placé dans le corps de la classe anonyme, entouré d'accolades.
Exemple :

```
public Iterator<E> iterator() {
  return new Iterator<E>() {
    private int remaining;
    {
      remaining = size;
    }
    // … méthodes hasNext, next et remove
  };
}
```

# Accès aux variables locales

Les classes anonymes ont même accès aux variables locales de la méthode dans laquelle elles sont déclarées, pour peu que celles-ci soient déclarées finales – ce qui les rend non modifiables. Exemple :

```
public Iterator<E> iterator() {
  final int initiallyRemaining = size;
  return new Iterator<E>() {
    private int remaining =
      initiallyRemaining;
   // … méthodes hasNext, next et remove
  };
}
```