Mise en œuvre des collections : listes

Michel Schinz

2018-05-14

1 Introduction

Les collections, dont l'utilisation a déjà été étudiée précédemment, sont mises en œuvre au moyen de techniques non triviales qu'il est bon de connaître, pour plusieurs raisons. D'une part, il s'agit de techniques fondamentales, utilisables dans de nombreux autres contextes ; d'autre part, leur connaissance permet l'utilisation optimale des différents types de collections.

Cette leçon et la suivante ont donc pour but d'examiner la mise en œuvre de versions très simplifiées des collections Java, en commençant par les listes.

2 Listes

Les listes constituent la collection la plus simple à mettre en œuvre parmi celles de la bibliothèque Java. Les deux mises en œuvre principales fournies par Java, à savoir les tableaux-listes (`ArrayList`) et les listes chaînées (`LinkedList`) sont assez différentes pour mériter d'être examinées individuellement.

La version simplifiée des listes étudiées ici est représentée par l'interface `SList` (pour *simplified list*) ci-dessous :

```
public interface SList<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  void add(int i, E e);
  void remove(int i);
  boolean contains(E e);
  E get(int i);
  E set(int i, E e);
  Iterator<E> iterator();
}
```

Trois classes implémentant cette interface sont présentées plus bas. Elles sont organisées de manière similaire aux classes Java correspondantes, c'est-à-dire qu'il existe :

- une classe héritable `SAbstractList`, qui fournit des mises en œuvre par défaut des méthodes qu'il est possible de définir en fonction d'autres méthodes, p.ex. `isEmpty` en fonction de `size`,
- une classe instanciable `SArrayList`, qui est la mise en œuvre basée sur un tableau (tableau-liste ou tableau dynamique),
- une classe instanciable `SLinkedList`, qui est la mise en œuvre basée sur des nœuds chaînés entre eux (liste chaînée).

Contrairement aux collections de la bibliothèque Java, celles développées ici ne redéfinissent ni `equals`, ni `hashCode`. Cela signifie que leurs instances sont comparées par référence et pas par structure comme les collections Java. Comme nous l'avons vu, cette solution est préférable pour les classes non immuables, étant donné les problèmes posés par des méthodes `hashCode` et `equals` dont le résultat dépend de l'état des objets auxquels on les applique.

Cela dit, si la comparaison du *contenu* de deux listes — ou autres types de collections — devait s'avérer assez utile pour justifier l'existence d'une méthode la mettant en œuvre, il ne serait pas difficile d'en ajouter une. Dans le cas des listes, cette méthode pourrait avoir la signature suivante :

```
public interface SList<E> extends Iterable<E> {
  boolean hasSameElementsAs(SList<E> that);
  // … autres méthodes (voir plus haut)
}
```

3 Liste abstraite (`SAbstractList`)

La classe héritable `SAbstractList` fournit des mises en œuvre par défaut des méthodes de l'interface `SList` qu'il est possible d'exprimer en fonction d'autres méthodes de cette même interface. Par exemple, `isEmpty` s'exprime trivialement en terme de `size` :

```
public abstract class SAbstractList<E>
    implements SList<E> {

  @Override
  public boolean isEmpty() {
    return size() == 0;
  }

  // … autre méthodes (contains, toString, …check)
}
```

3.1 Test d'appartenance

Dans le cas des listes, le test d'appartenance (`contains`) ne peut se faire de manière plus efficace que par parcours des éléments, avec une complexité de O(n). Il est dès lors possible de le mettre directement en œuvre dans la classe `AbstractList`, au moyen d'une boucle *for-each*.

```
@Override
public boolean contains(E e) {
  for (E e1: this) {
    if (e1.equals(e))
      return true;
  }
  return false;
}
```

3.2 Représentation textuelle

La classe `AbstractList` fournit une redéfinition de la méthode `toString`, qui produit une représentation textuelle de la liste à laquelle on l'applique. Celle-ci est constituée de la représentation textuelle des éléments de la liste, séparés par une virgule et entourés de crochets (`[]`).

```
@Override
public String toString() {
  StringBuilder b = new StringBuilder("[");
  Iterator<E> i = iterator();
  while (i.hasNext()) {
    b.append(i.next())
      .append(i.hasNext() ? ",␣" : "");
  }
  return b.append("]").toString();
}
```

Cette méthode permet en fait d'obtenir la représentation textuelle de n'importe quel objet de type `Iterable`.

3.3 Vérification d'index

Finalement, `AbstractList` définit deux méthodes permettant de vérifier la validité des index passés aux différentes méthodes :

- `checkElementIndex` vérifie que l'index qu'on lui passe est compris entre 0 (inclus) et la taille de la liste (exclue), et donc qu'il désigne bien un élément de celle-ci,

- `checkPositionIndex` vérifie que l'index qu'on lui passe est compris entre 0 (inclus) et la taille de la liste (incluse), et donc qu'il désigne bien une position d'insertion dans celle-ci.

Chacune de ces méthodes retourne l'index s'il est valide, et lève l'exception `IndexOutOfBoundsEx` sinon. Ces méthodes étant destinées à être utilisées exclusivement par les sous-classes concrètes de `AbstractList`, elles sont protégées.

```
protected final int checkElementIndex(int i) {
  if (! (0 <= i && i < size()))
    throw new IndexOutOfBoundsException();
  return i;
}

protected final int checkPositionIndex(int i) {
  if (! (0 <= i && i <= size()))
    throw new IndexOutOfBoundsException();
  return i;
}
```

4 Tableau-liste (`ArrayList`)

Les tableaux-listes, ou tableaux dynamiques, sont à mi-chemin entre les tableaux et les listes, d'où leur nom. Les éléments d'un tableau-liste sont stockés dans un tableau normal qui est, au besoin, « agrandi » par copie dans un nouveau tableau plus grand.

La grande force des tableaux-listes est qu'ils permettent d'accéder à un élément dont on connaît l'index en temps constant, c-à-d en O(1).

Leur faiblesse est que l'insertion d'un élément à une position quelconque implique de déplacer tous les éléments se trouvant à des index supérieurs, et la complexité de cette opération est donc O(n). Cela dit, si l'insertion se fait uniquement à la fin de la liste, elle a alors une complexité amortie de O(1).

La classe `SArrayList` représente un tableau-liste. Elle possède deux attributs : `size` est la taille de la liste, donc le nombre d'éléments qu'elle contient, tandis que `array` est le tableau contenant les éléments, appelé tableau sous-jacent. Bien entendu, le tableau sous-jacent doit toujours avoir une taille supérieure ou égale à la taille de la liste, faute de quoi il ne peut stocker la totalité des éléments. La taille du tableau sous-jacent est souvent appelée la capacité (*capacity*) du tableau-liste. La figure 1 montre un tableau-liste d'une capacité de 8 objet mais n'en contenant que 5, qui sont les noms des jours ouvrables.

Ci-dessous, la capacité initiale du tableau-liste est arbitrairement fixée à 10. La classe `ArrayList` de la bibliothèque Java offre un constructeur permettant de spécifier la capacité initiale au moment de la construction.

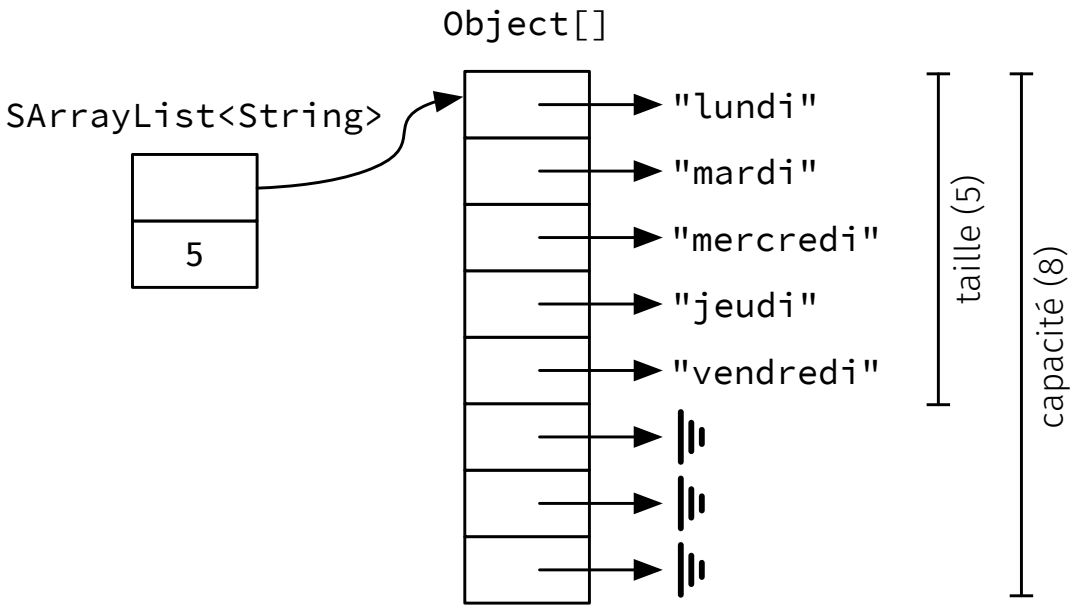

Fig. 1: Tableau-liste des jours ouvrables

```
public final class SArrayList<E> extends SAbstractList<E>{
  private int size = 0;

  @SuppressWarnings(”unchecked”)
  private E[] array = (E[]) new Object[10];

  @Override
  public int size() {
    return size;
  }

  // … autres méthodes (voir plus bas)
}
```

4.1 Ajout et suppression

Lors de l’ajout d’un élément dans un tableau-liste, deux cas sont à distinguer : soit il reste de la place dans le tableau sous-jacent pour stocker le nouvel élément, soit il n’en reste pas.

Dans le second cas, le tableau sous-jacent doit être « redimensionné ». Bien entendu, comme il n’est pas possible de redimensionner un tableau en Java, cela doit se faire par

copie dans un nouveau tableau, nommé `newArray` ci-dessous. Lorsque cela se produit, une question se pose : quelle taille donner au nouveau tableau ?

Une première idée serait de dimensionner le nouveau tableau pour qu'il puisse juste contenir le nouvel élément, c-à-d que sa taille soit celle de l'ancien augmentée d'une unité. Malheureusement, en procédant de la sorte, une nouvelle copie du tableau sous-jacent doit être faite à chaque ajout lors d'ajouts successifs, et la complexité de l'ajout est alors de $O(n)$.

Dès lors, il faut dimensionner le nouveau tableau de manière à ce que plusieurs ajouts successifs puissent être faits avant qu'un nouveau redimensionnement ne soit nécessaire. Une manière de faire consiste à doubler la taille du tableau sous-jacent à chaque redimensionnement, garantissant ainsi que l'ajout a une complexité *amortie* de $O(1)$. (La notion de complexité amortie ne sera pas examinée en détail ici.)

La méthode `add` ci-dessous met cette idée en œuvre. Elle utilise la méthode `arraycopy` pour déplacer ou copier les éléments du tableau sous-jacent. Cette méthode prend cinq paramètres qui sont, dans l'ordre : le tableau source, l'index de départ dans le tableau source, le tableau destination, l'index de départ dans le tableau destination et le nombre d'éléments à copier.

```
@Override
public void add(int i, E e) {
  checkPositionIndex(i);
  if (size < array.length) {
    System.arraycopy(array, i, array, i + 1, size - i);
  } else {
    @SuppressWarnings("unchecked")
    E[] newArray = (E[]) new Object[array.length * 2];
    System.arraycopy(array, 0, newArray, 0, i);
    System.arraycopy(array, i, newArray, i + 1, size - i);
    array = newArray;
  }
  array[i] = e;
  size += 1;
}
```

La suppression d'un élément dans un tableau-liste implique de déplacer d'une position vers le bas tous les éléments d'index supérieur. Là aussi, cela peut se faire au moyen d'une copie de ces éléments, effectuée par la méthode `arraycopy`.

Une fois les éléments déplacés, il est très important de mettre à `null` l'élément qui se trouve juste après le dernier élément appartenant à la liste. Cela garantit que le tableau sous-jacent ne référence jamais un élément qui se trouvait précédemment dans la liste mais ne s'y trouve plus, évitant ainsi de garder inutilement allouée la mémoire associée à un tel élément. Pour plus de détails à ce sujet, voir la règle 7 (*Eliminate obsolete object references*) du livre *Effective Java*.

```
@Override
public void remove(int i) {
  checkElementIndex(i);
  System.arraycopy(array, i + 1, array, i, size - i - 1);
  array[--size] = null;
}
```

A noter que le tableau sous-jacent pourrait aussi être redimensionné à la baisse lors d'une suppression, s'il est devenu trop grand par rapport au nombre d'éléments présents. Cela n'a pas été fait ici dans un souci de simplicité, et n'est pas fait non plus dans la bibliothèque Java, qui offre par contre la méthode `trimToSize` pour réduire la capacité d'un tableau-liste à sa taille actuelle.

4.2 Accès et modification

Les opérations d'accès (`get`) et de modification (`set`) sont très simple à mettre en œuvre, car elles correspondent à de simples accès au tableau sous-jacent. Il est toutefois important de bien valider les index reçus au moyen de la méthode `checkElementIndex`, car ceux-ci pourraient très bien être valides pour le tableau sous-jacent mais invalides pour le tableau-liste !

```
@Override
public E get(int i) {
  return array[checkElementIndex(i)];
}

@Override
public E set(int i, E e) {
  E oldE = array[checkElementIndex(i)];
  array[i] = e;
  return oldE;
}
```

4.3 Itération

La méthode `iterator` doit retourner un itérateur permettant de parcourir les éléments du tableau-liste, et cet itérateur doit être un objet implémentant l'interface `Iterator`. Il faut dès lors écrire une classe implémentant cette interface et capable d'itérer sur les éléments du tableau-liste.

Cette classe peut s'écrire de plusieurs manières, et trois d'entre elles sont présentées ci-dessous. Elles utilisent toutes différentes formes de classes imbriquées.

4.3.1 Itérateur imbriqué statiquement

La première manière de définir la classe de l'itérateur consiste à la définir comme une classe privée et imbriquée statiquement dans la classe `SArrayList`. Pour écrire cette classe, il convient de réfléchir à comment parcourir les éléments d'un tableau-liste, puis traduire cette technique de parcours en un itérateur.

Etant donné que les éléments d'un tableau-liste sont stockés dans un tableau, le parcours de ses éléments peut se faire très simplement au moyen d'un simple index. Cet index constitue donc l'état de l'itérateur, appelé `nextI` ci-dessous, et il est incrémenté à chaque appel de la méthode `next`. La méthode `hasNext` se contente de vérifier que cet index n'a pas atteint la taille du tableau-liste. Finalement, la méthode `remove` de l'itérateur, qui doit supprimer le dernier élément retourné par `next`, appelle simplement la méthode `remove` du tableau-liste avec la valeur précédente de l'index, qui correspond bien à la dernière valeur retournée par `next`.

La mise en œuvre de `remove` présente une petite difficulté, à savoir qu'il n'est ni valide de l'appeler avant le premier appel à `next`, ni valide de l'appeler plus d'une fois de suite, sans appeler `next` entre temps. L'attribut `canRemove` de l'itérateur, qui n'est vrai que lorsqu'un appel à `remove` est légal, est utilisé dans ce but.

Il est clair que l'itérateur doit absolument connaître le tableau-liste sur lequel il itère pour pouvoir faire son travail. Dès lors, celui-ci lui est simplement passé à la construction et stocké dans l'attribut `list`.

```
private static final class SALIterator1<E>
    implements Iterator<E> {
  private final SArrayList<E> list;
  private int nextI;
  private boolean canRemove;

  public SALIterator1(SArrayList<E> list) {
    this.list = list;
    this.nextI = 0;
    this.canRemove = false;
  }

  @Override
  public boolean hasNext() {
    return nextI < list.size;
  }

  @Override
  public E next() {
    if (! hasNext())
      throw new NoSuchElementException();
    canRemove = true;
```

```
    return list.array[nextI++];
  }

  @Override
  public void remove() {
    if (! canRemove)
      throw new IllegalStateException();
    canRemove = false;
    list.remove(--nextI);
  }
}
```

Une fois cette classe imbriquée écrite, il est très facile d'écrire la méthode `iterator` de la classe `SArrayList`, qui se contente d'en créer une nouvelle instance et de la retourner :

```
@Override
public Iterator<E> iterator() {
  return new SALIterator1<>(this);
}
```

4.3.2 Itérateur intérieur

La classe `SALIterator1` ci-dessus étant imbriquée *statiquement* dans la classe `SArrayList`, elle n'a pas accès directement aux attributs et méthodes de sa classe englobante. Pour cette raison, il est nécessaire, comme nous l'avons vu, de lui passer au moment de la construction le tableau-liste sur lequel itérer.

Java offre un autre type de classe imbriquée, nommé **classe intérieure** (*inner class*), qui n'est rien d'autre qu'une classe imbriquée *non* statique. Contrairement aux instances d'une classe imbriquée statique, les instances d'une classe intérieure sont associées à une instance de leur classe englobante, et ont donc accès à ses attributs, méthodes et paramètres de type.

Etant donné que l'itérateur de tableau-liste doit avoir accès au tableau-liste sur lequel il itère, il paraît judicieux de le définir en tant que classe intérieure. Ainsi, toute instance d'un tel itérateur est associé à l'instance du tableau-liste sur lequel il itère, et a directement accès à ses membres et paramètres de type. Cela rend cette nouvelle version de l'itérateur, nommé `SALIterator2`, passablement plus simple que la précédente.

```
private final class SALIterator2
    implements Iterator<E> {
  private int nextI = 0;
  private boolean canRemove = false;

  @Override
```

```
  public boolean hasNext() {
    return nextI < size;
  }

  @Override
  public E next() {
    if (! hasNext())
      throw new NoSuchElementException();
    canRemove = true;
    return array[nextI++];
  }

  @Override
  public void remove() {
    if (! canRemove)
      throw new IllegalStateException();
    canRemove = false;
    SArrayList.this.remove(--nextI);
  }
}
```

Il est très important de comprendre que même si cette nouvelle version est plus concise et élégante que la précédente, son principe de fonctionnement est rigoureusement identique. En particulier, la référence vers le tableau-liste sur lequel itérer, stockée dans l'attribut `list` de la première version, existe également dans la seconde, mais n'est pas visible dans le programme source. Il est possible d'y avoir néanmoins accès au moyen de la notation `this` préfixée, comme cela est fait dans la méthode `remove`.

Comme cette nouvelle version de l'itérateur ne prend plus explicitement en argument de son constructeur le tableau-liste sur lequel itérer, la définition de la version de la méthode `iterator` correspondante est légèrement plus simple :

```
@Override
public Iterator<E> iterator() {
  return new SALIterator2();
}
```

4.3.3 Itérateur intérieur anonyme

La classe `SALIterator2` ci-dessus n'est utilisée qu'à un seul endroit, dans l'énoncé `new` de la méthode `iterator`. Il paraît donc sensé d'en faire une classe intérieure anonyme, ce qui permet de récrire la méthode `iterator` ainsi :

```
@Override
public Iterator<E> iterator() {
```

```
  return new Iterator<E>() {
    private int nextI = 0;
    private boolean canRemove = false;

    @Override
    public boolean hasNext() {
      return nextI < size;
    }

    @Override
    public E next() {
      if (!hasNext())
        throw new NoSuchElementException();
      canRemove = true;
      return array[nextI++];
    }

    @Override
    public void remove() {
      if (!canRemove)
        throw new IllegalStateException();
      SArrayList.this.remove(--nextI);
      canRemove = false;
    }
  };
}
```

Là aussi, il est important de comprendre que cette nouvelle version, même si elle est encore plus concise et élégante que la précédente, se comporte exactement de la même manière.

5 Liste chaînée (`LinkedList`)

Les éléments d'une **liste chaînée** (*linked list*) ne sont pas stockés dans un tableau, comme ceux d'un tableau-liste, mais référencés par des **nœuds** (*nodes*), et ces nœuds sont chaînés entre-eux. C'est-à-dire que chaque nœud possède une référence vers au moins un de ses voisins.

Lorsque chaque nœud possède une référence vers un seul de ses voisins, généralement son successeur, on parle de **liste simplement chaînée** (*singly linked list*) ; lorsque chaque nœud possède une référence vers ses deux voisins, on parle de **liste doublement chaînée** (*doubly linked list*). Finalement, lorsque le dernier nœud et le premier nœud sont voisins — et donc liés par une ou deux références — on parle de **liste circulaire** (*circular*

list).

Contrairement aux tableaux-listes, les listes chaînées ne permettent pas d'accéder à un élément dont on connaît l'index en O(1). Avec une liste chaînée, cette même opération a une complexité de O(n), où n est le nombre d'éléments de la liste.

En contrepartie, l'insertion d'un élément à une position quelconque n'implique pas de déplacer des éléments et peut donc se faire en O(1). Mais attention : cela n'est vrai qu'en faisant l'hypothèse que l'on possède déjà une référence sur un nœud voisin de celui que l'on désire insérer, faute de quoi la simple recherche de ce nœud a une complexité de O(n), comme nous le verrons ci-dessous.

La classe `SLinkedList` représente une liste simplement chaînée. Elle possède deux attributs : `size` est la taille de la liste, tandis que `head` est le premier nœud de la liste, ou `null` si (et seulement si) la liste est vide.

La classe `Node`, privée et imbriquée statiquement dans `SLinkedList`, représente un nœud simplement chaîné. Son attribut `next` référence le prochain nœud de la liste, ou est `null` pour le dernier nœud, tandis que l'attribut `elem` référence l'élément correspondant à ce nœud.

La figure 2 montre une telle liste chaînée des jours ouvrables. Du point de vue de l'extérieur, son contenu est totalement identique à celui du tableau-liste de la figure 1, mais en interne il est organisé de manière très différente.

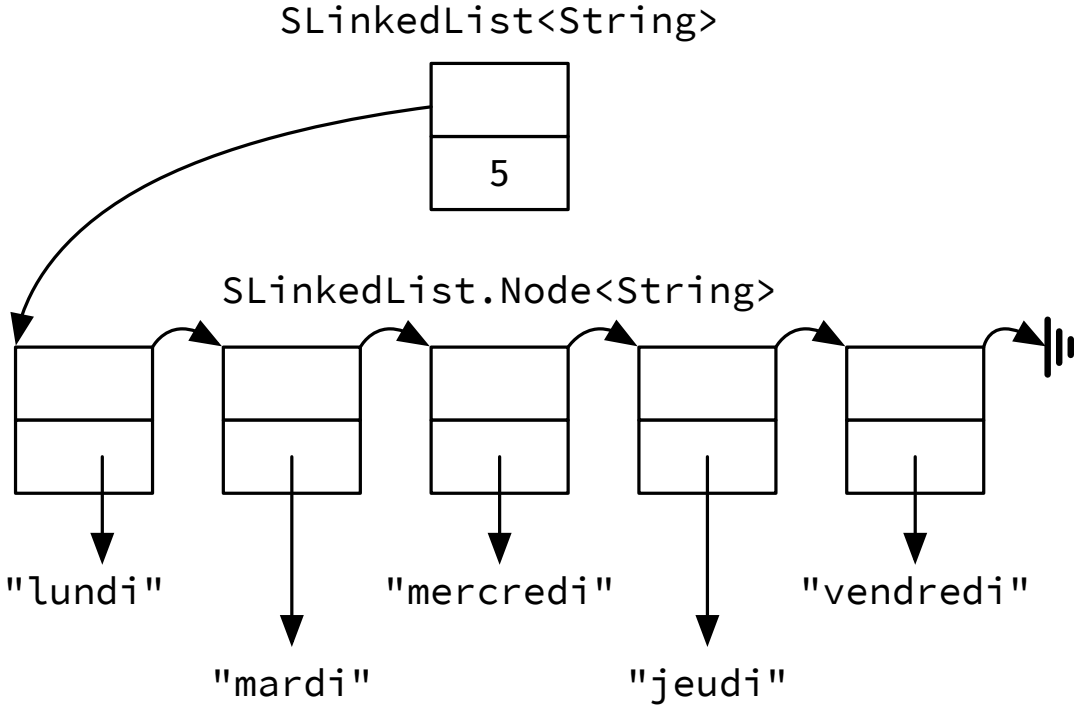

Fig. 2: Liste (simplement) chaînée des jours ouvrables

Comme nous le verrons, plusieurs méthodes publiques ont besoin d'obtenir le nœud

correspondant à un index donné, ce qui est le but de la méthode privée `getNode`. Cette méthode ne fait rien d'autre que suivre les liens liant les nœuds entre eux, jusqu'à arriver à celui d'index recherché. Cette méthode a donc, dans le cas général, une complexité de $O(n)$, et toute autre méthode l'utilisant ne peut donc avoir une complexité meilleure.

```
public final class SLinkedList<E>
    extends SAbstractList<E> {
  private int size = 0;
  private Node<E> head = null;

  @Override
  public int size() {
    return size;
  }

  // … autres méthodes

  private Node<E> getNode(int i) {
    Node<E> n = head;
    for (int j = 0; j < i; ++j)
      n = n.next;
    return n;
  }

  private static final class Node<E> {
    private Node<E> next;
    private E elem;

    public Node(Node<E> next, E elem) {
      this.next = next;
      this.elem = elem;
    }
  }
}
```

5.1 Ajout et suppression

L'ajout et la suppression d'un élément dans une liste chaînée implique la mise à jour des liens du ou des nœuds voisins de celui à insérer ou supprimer.

Pour l'ajout, deux cas sont à considérer : si le nouvel élément doit être placé en tête de liste, alors le nouveau nœud lui correspondant doit devenir la nouvelle tête de liste ; sinon, il faut tout d'abord obtenir le prédécesseur du nœud à insérer (`pred` ci-dessous)

puis ajuster son lien vers son successeur pour le faire référencer le nouveau nœud. Dans les deux cas, il faut penser à incrémenter la taille de la liste.

```
@Override
public void add(int i, E e) {
  if (checkPositionIndex(i) == 0) {
    head = new Node<>(head, e);
  } else {
    Node<E> pred = getNode(i - 1);
    pred.next = new Node<>(pred.next, e);
  }
  size += 1;
}
```

La suppression est très similaire à l'ajout, et les deux mêmes cas sont à distinguer.

```
@Override
public void remove(int i) {
  if (checkElementIndex(i) == 0) {
    head = head.next;
  } else {
    Node<E> pred = getNode(i - 1);
    pred.next = pred.next.next;
  }
  size -= 1;
}
```

5.2 Accès et modification

L'accès à l'élément à un index donné, ou sa modification, consistent à obtenir le nœud à cet index puis à extraire ou modifier son élément. Etant donné l'existence de la méthode `getNode`, cela se fait très simplement.

```
@Override
public E get(int i) {
  return getNode(checkElementIndex(i)).elem;
}

@Override
public E set(int i, E e) {
  Node<E> node = getNode(checkElementIndex(i));
  E oldE = node.elem;
  node.elem = e;
  return oldE;
```

```
}
```

5.3 Itération

Si l'on ignore dans un premier temps sa méthode `remove`, et considère simplement qu'elle lève l'exception `UnsupportedOperationException`, l'itérateur sur les listes simplement chaînées est facile à écrire. Son seul état est le nœud `next` dont l'élément doit être retourné par le prochain appel à la méthode du même nom. Un tel itérateur non modifiable est retourné par la méthode (fictive) `unmodifiableIterator` ci-dessous :

```
public Iterator<E> unmodifiableIterator() {
  return new Iterator<E>() {
    private Node<E> next = head;

    @Override
    public boolean hasNext() {
      return next != null;
    }

    @Override
    public E next() {
      if (!hasNext())
        throw new NoSuchElementException();

      E elem = next.elem;
      next = next.next;
      return elem;
    }
  };
}
```

L'ajout de la méthode `remove` complique passablement l'itérateur, car conserver uniquement une référence sur le prochain nœud (`next`) ne suffit plus. En effet, la méthode `remove` doit supprimer le dernier élément retourné par `next`, et dès lors il faut en tout temps avoir accès au prédécesseur du prédécesseur du nœud `next`!

Pour ce faire, la version complète — c-à-d avec la méthode `remove` — de l'itérateur maintient non pas une mais trois références vers des nœuds, qui sont :

- `pred`, qui désigne le nœud précédent le nœud courant,
- `curr`, qui désigne le nœud courant, c-à-d celui dont l'élément a été retourné par le dernier appel à `next`,

- `next`, qui désigne comme précédemment le nœud suivant, c-à-d celui dont l'élément sera retourné par le prochain appel à `next`.

Ces trois nœuds sont consécutifs dans la liste : `pred` précède directement `curr`, qui précède directement `next`. Dès lors, `curr` est `null` lorsque `next` désigne le premier nœud de la liste, et `pred` est `null` lorsque soit `next`, soit `curr` désignent le premier nœud de la liste.

Ces trois références doivent être mise à jour correctement dans la méthode `next`. Cela fait, la méthode `remove` est relativement simple à écrire et ressemble passablement à la méthode `remove` de la classe `SLinkedList`.

```
@Override
public Iterator<E> iterator() {

  return new Iterator<E>() {
    private Node<E> pred = null, curr = null, next = head;
    private boolean canRemove = false;

    @Override
    public boolean hasNext() {
      return next != null;
    }

    @Override
    public E next() {
      if (!hasNext())
        throw new NoSuchElementException();
      canRemove = true;

      E elem = next.elem;
      pred = curr;
      curr = next;
      next = next.next;
      return elem;
    }

    @Override
    public void remove() {
      if (!canRemove)
        throw new IllegalStateException();
      canRemove = false;

      if (pred == null)
        head = head.next;
```

```
      else
        pred.next = pred.next.next;
      curr = pred;
      pred = null;
      size -= 1;
    }
  };

}
```

Il faut noter que, dans la méthode `remove`, la mise à jour du lien du prédécesseur pourrait s'écrire de manière plus simple, en tirant parti de l'exitence de la référence `next` :

```
if (pred == null)
  head = next;       // au lieu de head.next
else
  pred.next = next;  // au lieu de pred.next.next
```

Cela n'a pas été fait plus haut afin que la similarité entre la méthode `remove` de l'itérateur et celle de la classe `SLinkedList` soit plus évidente.

6 Références

- *The Java® Language Specification*, de James Gosling et coauteurs, en particulier :
  - §8.1.3 *Inner Classes and Enclosing Instances*,
  - §15.9.5 *Anonymous Class Declarations*,
- *Effective Java (3rd ed.)* de Joshua Bloch, en particulier :
  - la règle 7, *Eliminate obsolete object references* sur la raison pour laquelle il faut éliminer les références vers les objets devenus inutiles,
  - la règle 24, *Favor static member classes over nonstatic* sur l'utilisation judicieuse des différents types de classes imbriquées.