Mise en œuvre des collections : ensembles

Michel Schinz

2017-05-29

1 Introduction

Pour mémoire, la bibliothèque Java offre deux mises en œuvre des ensembles :

1. `HashSet`, basée sur les tables de hachage et offrant les opérations principales en O(1),
2. `TreeSet`, basée sur les arbres binaires de recherche et offrant les opérations principales en O(log n).

Les mêmes techniques sont utilisées par les classes `HashMap` et `TreeMap` pour mettre en œuvre les tables associatives.

Etant donné que la mise en œuvre basée sur les tables de hachage est la plus simple et la plus efficace des deux, elle est le sujet de cette leçon.

2 Ensembles

La version simplifiée des ensembles Java étudiée ici est représentée par l'interface `SSet` (pour *simplified set*) suivante :

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

Un total de trois classes implémentant cette interface sont réalisées plus bas, organisées de manière similaire aux classes Java correspondantes :

- une classe héritable `SAbstractSet`, 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 `SListSet`, qui met en œuvre un ensemble au moyen d'une liste, ce qui est très inefficace mais pédagogiquement intéressant,
- une classe instanciable `SHashSet`, qui met en œuvre un ensemble au moyen d'une table de hachage.

Aucune de ces classes ne redéfinit `equals` ou `hashCode`, car l'égalité par référence est celle qui convient à des classes non immuables comme celles-ci.

3 Ensemble abstrait (`SAbstractSet`)

Tout comme pour les listes, il est intéressant de définir une classe héritable pour les ensembles, qui fournit une mise en œuvre de la méthode `isEmpty` en fonction de `size`, ainsi qu'une redéfinition de la méthode `toString`.

```
public abstract class SAbstractSet<E> implements SSet<E> {
  @Override
  public boolean isEmpty() {
    return size() == 0;
  }

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

A noter que la méthode `isEmpty` pourrait également être définie comme méthode par défaut dans l'interface `SSet`. Il n'en va toutefois pas de même de la méthode `toString`, car Java interdit les méthodes par défaut qui redéfinissent des méthodes de `Object`.

4 Ensemble par liste

Une première manière de mettre en œuvre les ensembles consiste à utiliser une liste sans doublons. C'est le but de la classe `SListSet`, sujet de cette section. Son seul attribut est la liste (ici chaînée) contenant les éléments, dans un ordre quelconque.

```
public final class SListSet<E> extends SAbstractSet<E> {
  private final SList<E> list = new SLinkedList<>();

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

  // … autres méthodes (voir ci-dessous)
}
```

Comme nous allons le voir, cette mise en œuvre est très inefficace, car les opérations principales — `add`, `remove` et `contains` — ont une complexité de $O(n)$ où n est la taille de l'ensemble. Pour cette raison, la bibliothèque Java ne fournit pas de mise en œuvre équivalente. Néanmoins, cette mise en œuvre facilite beaucoup la compréhension ultérieure de celle basée sur une table de hachage, d'où son intérêt.

4.1 Ajout

Etant donné la nécessité d'éviter les doublons, l'ajout dans la liste sous-jacente ne doit se faire que si l'élément reçu ne s'y trouve pas déjà. En conséquence, l'ajout dans un ensemble représenté par une liste a une complexité de $O(n)$, alors que l'ajout dans une liste peut, sous certaines conditions, avoir une complexité de $O(1)$ seulement !

```
@Override
public void add(E e) {
  if (! list.contains(e))
    list.add(0, e);
}
```

Si l'élément n'appartient pas encore à l'ensemble, la position à laquelle l'ajouter dans la liste est arbitraire. La mise en œuvre ci-dessus l'ajoute en tête de liste, car cela peut se faire très rapidement dans une liste chaînée.

4.2 Suppression

La suppression d'un élément de l'ensemble implique de rechercher celui-ci, puis de le supprimer si on le trouve. Cela se fait facilement au moyen d'un itérateur. Sachant

que la liste ne contient pas de doublons, la recherche peut s'arrêter dès que la première occurrence de l'élément à supprimer l'a été.

```
@Override
public void remove(E e) {
  Iterator<E> listIt = list.iterator();
  while (listIt.hasNext()) {
    E e1 = listIt.next();
    if (e1.equals(e)) {
      listIt.remove();
      return;
    }
  }
}
```

4.3 Test d'appartenance

Etant donné l'existence de la méthode `contains` dans l'interface `SList`, le test d'appartenance est trivial à mettre en œuvre :

```
@Override
public boolean contains(E e) {
  return list.contains(e);
}
```

4.4 Itération

L'itération sur les éléments de l'ensemble peut se faire simplement par itération sur les éléments de la liste. Dès lors, il est valide de simplement retourner un itérateur sur cette dernière dans la méthode `iterator`.

```
@Override
public Iterator<E> iterator() {
  return list.iterator();
}
```

5 Ensemble par table de hachage

La mise en œuvre des ensembles au moyen d'une liste, telle que présentée ci-dessus, est simple mais très inefficace. Comme nous l'avons vu, les opérations principales — ajout, suppression, test d'appartenance — ont une complexité de $O(n)$, où n est le nombre d'éléments de l'ensemble. Il est néanmoins possible de s'en inspirer pour obtenir une mise en

œuvre très efficace, dans laquelle ces opérations ont une complexité de O(1), sous certaines conditions.

L'idée est la suivante : plutôt que de stocker les n éléments de l'ensemble dans *une seule* liste de longueur n, on les stocke dans (environ) n listes distinctes, chacune de longueur (environ) 1, et placées dans un tableau. A supposer qu'il existe un moyen rapide — en O(1) — de déterminer à laquelle de ces listes un élément appartient, les opérations susmentionnées peuvent être mises en œuvre en O(1). Or ce moyen de déterminer à quelle liste appartient un élément donné existe : il s'agit du hachage.

Plus précisément, les éléments d'un ensemble de n valeurs peuvent être stockés dans un tableau de environ n listes, appelé **table de hachage** (*hash table*). L'index de la liste à laquelle un élément appartient est déterminé en ramenant sa valeur de hachage — un entier quelconque — à un index valide de ce tableau. Si la fonction de hachage répartit bien les éléments, chaque liste du tableau contient (environ) 1 élément, et dès lors les opérations sur chacune des listes ont une complexité de O(1). Si la fonction de hachage a une complexité de O(1), alors les opérations de base sur l'ensemble ont une complexité de O(1) également.

La classe `SHashSet` ci-dessous met en œuvre les ensembles au moyen de cette idée. La table de hachage est stockée dans l'attribut `table` de cette classe. La taille de l'ensemble est quant à elle stockée dans l'attribut `size`, comme d'habitude. La taille de la table de hachage, `table.length`, est appelée sa capacité (*capacity*).

La méthode `newTable` construit et retourne une table de hachage vide de capacité donnée. Initialement, cette capacité est arbitrairement fixée à 10.

```
public final class SHashSet<E> extends SAbstractSet<E> {
  private int size;
  private SList<E>[] table;

  public SHashSet() {
    this.size = 0;
    this.table = newTable(10);
  }

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

  // … autres méthodes (voir ci-dessous)

  private static <E> SList<E>[] newTable(int capacity) {
    @SuppressWarnings("unchecked")
    SList<E>[] table = new SList[capacity];
    for (int i = 0; i < capacity; ++i)
```

```
      table[i] = new SLinkedList<>();
    return table;
  }
}
```

Le rapport entre le nombre d'éléments stockés dans une table de hachage et sa capacité est nommé son **facteur de charge** (*load factor*). Idéalement, celui-ci devrait être proche de 1, afin que les listes de la table de hachage aient bien une longueur proche de 1. Nous y reviendrons.

5.1 Indexation

Comme expliqué plus haut, l'index, dans la table de hachage, de la liste correspondant à un élément s'obtient en ramenant sa valeur de hachage à un index valide. Pour ce faire, on peut simplement utiliser le reste de la division entière de la valeur de hachage par la capacité de la table[1].

En Java, il faut néanmoins prendre gardre à utiliser la bonne version de la division entière. Pour mémoire, l'opérateur « reste de la division », noté %, retourne une valeur ayant le signe du dividende. Sachant que la méthode `hashCode` peut retourner un entier `int` quelconque, y compris négatif, l'opérateur % n'est pas idéal ici. Il faut donc lui préférer la méthode `floorMod` de la classe `Math`, qui utilise une autre définition de la division entière et retourne une valeur ayant le signe du diviseur.

Etant donné qu'il est souvent nécessaire d'obtenir la liste de la table de hachage correspondant à un élément donné, il est utile de définir une méthode faisant cela :

```
private static <E> SList<E> listFor(SList<E>[] table,
                                    E elem) {
  return table[Math.floorMod(elem.hashCode(),
                             table.length)];
}
```

Il aurait pu sembler plus logique de définir une méthode *non* statique utilisant directement l'attribut `table` de la classe comme table de hachage. La version plus générale définie ci-dessus est toutefois utile lors du rehachage, comme nous le verrons plus bas.

5.2 Ajout

L'ajout d'un élément à l'ensemble se fait en deux phases :

1. Il existe des techniques plus rapides que le reste de la division pour ramener une valeur de hachage à un index valide. Par exemple, en s'assurant que la capacité de la table est toujours une puissance de deux, on peut utiliser un simple masquage. Pour une table de taille quelconque, il est également possible de combiner une multiplication sur 64 bits et un décalage à droite. Dans un soucis de simplicité, ces techniques ne sont toutefois pas explorées ici.

1. la liste correspondant à l'élément est obtenue,

2. l'élément est ajouté à cette liste, pour peu qu'il n'y soit pas déjà.

La seconde phase est exactement identique à l'ajout dans un ensemble représenté par une unique liste !

```
@Override
public void add(E e) {
  SList<E> list = listFor(table, e);
  if (! list.contains(e)) {
    list.add(0, e);
    size += 1;
  }
}
```

Notez au passage que la manière dont l'ajout est fait permet de comprendre pourquoi les méthodes `hashCode` et `equals` doivent être compatibles : la première phase ci-dessus utilise `hashCode` pour déterminer la liste correspondant à l'élément, tandis que la seconde phase utilise `equals` pour déterminer si l'élément si trouve déjà.

5.3 Suppression

La suppression d'un élément de l'ensemble se fait également en deux phases, qui sont similaires à celles de l'ajout :

1. la liste correspondant à l'élément est obtenue,

2. l'élément est supprimé de cette liste, s'il s'y trouve.

Une fois de plus, la seconde phase est en tous points identique à la suppression d'un ensemble représenté par une unique liste !

```
@Override
public void remove(E e) {
  Iterator<E> listIt = listFor(table, e).iterator();
  while (listIt.hasNext()) {
    E e1 = listIt.next();
    if (e1.equals(e)) {
      listIt.remove();
      size -= 1;
      return;
    }
  }
}
```

5.4 Test d'appartenance

Le test d'appartenance se fait, sans surprise, lui aussi en deux phase, similaires à celles de l'ajout et de la suppression :

1. la liste correspondant à l'élément est obtenue,
2. le test d'appartenance est fait dans cette liste.

```
@Override
public boolean contains(E e) {
  return listFor(table, e).contains(e);
}
```

5.5 Itération

Le parcours des éléments stockés dans la table de hachage n'est pas totalement trivial, étant donné la nature bidimensionnelle de cette table. L'itérateur doit parcourir chacune de ses deux dimensions, et possède deux attributs dans ce but :

1. `listIndex` est l'index de la liste en cours de parcours,
2. `listIt` est un itérateur parcourant la liste d'index `listIndex`.

Pour déterminer s'il reste encore des éléments à parcourir, le plus simple est d'ajouter un troisième attribut à l'itérateur, nommé `remaining` ci-dessous, qui compte le nombre d'éléments restants à parcourir.

```
@Override
public Iterator<E> iterator() {
  return new Iterator<E>() {
    int remaining = size;
    int listIndex = 0;
    Iterator<E> listIt = table[listIndex].iterator();

    @Override
    public boolean hasNext() {
      return remaining > 0;
    }

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

```
      remaining -= 1;
      while (! listIt.hasNext()) {
        listIndex += 1;
        listIt = table[listIndex].iterator();
      }
      return listIt.next();
    }

    @Override
    public void remove() {
      listIt.remove();
      size -= 1;
    }
  };
}
```

Etant donné la manière dont les éléments de l'ensemble sont placés dans les différentes listes de la table de hachage, il devrait être clair que l'ordre dans lequel l'itérateur ci-dessus parcourt ces éléments est quelconque. En particulier, il peut dépendre de l'ordre dans lequel les éléments ont été ajoutés à l'ensemble, et de la capacité de la table de hachage !

5.6 Rehachage

Jusqu'à présent, nous avons fait l'hypothèse que la table de hachage avait une capacité constante, arbitrairement fixée à 10. Bien entendu, cela ne permet pas d'obtenir les performances souhaitées : les opérations principales (ajout, suppression, test d'appartenance) n'ont une complexité de O(1) que si la longueur des listes de la table ne dépend pas du nombre d'éléments qu'elle contient.

Dès lors, il faut faire en sorte de redimensionner la table afin que sa capacité soit toujours proche de sa taille — le nombre d'éléments qu'elle contient. Cette condition peut s'exprimer succinctement au moyen du concept de facteur de charge (*load factor*) introduit plus haut. Pour mémoire, le facteur de charge l est le rapport entre la taille s de la table et sa capacité c :

$$l = \frac{s}{c}$$

Si la fonction de hachage répartit parfaitement les éléments sur les entiers $\{0, 1, \ldots, c\}$, un facteur de charge de 1 garantit que toutes les listes de la table contiennent exactement un élément. En pratique, il est très peu probable que cette condition soit satisfaite, et il est dès lors préférable que la capacité de la table soit légèrement supérieure au nombre d'éléments qu'elle stocke. En d'autres termes, il est souhaitable que le facteur de charge soit légèrement inférieur à 1.

Afin d'éviter de redimensionner la table à chaque changement du nombre d'éléments qu'elle stocke, ce qui serait très inefficace, il faut toutefois tolérer que le facteur de chage varie dans un intervalle jugé acceptable. Etant donné ce qui précède, nous fixerons cet intervalle à [0.4, 1]. Lorsque le facteur de charge sort de cette intervalle, la table est redimensionnée, et sa nouvelle capacité est déterminée pour que son nouveau facteur de charge ait une valeur considérée idéale, fixée ici à 0.7. Les constantes ci-dessous contiennent ces différentes valeurs, ainsi que la capacité minimale (et initiale) de la table :

```
private static int MIN_CAPACITY = 10;
private static double MIN_LOAD_FACTOR = 0.4;
private static double MAX_LOAD_FACTOR = 1;
private static double IDEAL_LOAD_FACTOR = 0.7;
```

Ces constantes définies, il est ensuite possible d'ajouter à la classe `SHashSet` une méthode nommée `rehashIfNeeded` qui redimensionne la table — opération nommée rehachage — si le facteur de charge n'est pas dans l'intervalle jugé acceptable. Ce rehachage consiste simplement à créer une nouvelle table, y copier la totalité des éléments de la table actuelle, et finalement remplacer l'ancienne table par la nouvelle.

```
private void rehashIfNeeded() {
  double loadFactor = size / (double)table.length;
  if (MIN_LOAD_FACTOR <= loadFactor
      && loadFactor <= MAX_LOAD_FACTOR)
    return;

  int idealCapacity = (int) (size / IDEAL_LOAD_FACTOR);
  int newCapacity = Math.max(MIN_CAPACITY, idealCapacity);
  if (newCapacity == table.length)
    return;

  SList<E>[] newTable = newTable(newCapacity);
  for (E e: this)
    listFor(newTable, e).add(0, e);

  table = newTable;
}
```

Une fois cette méthode écrite, il ne reste plus qu'à l'appeler chaque fois qu'un rehachage peut être nécessaire, c-à-d chaque fois que la taille de l'ensemble change. En cherchant dans les méthodes écrites précédemment toutes les mises à jour de l'attribut `size`, on constate qu'il y en a trois : une dans la méthode `add`, une dans la méthode `remove` et une dans la méthode `remove` de l'itérateur.

Ajouter un appel à `rehashIfNeeded` juste après la mise à jour de la taille dans les deux premiers cas ne pose aucun problème :

```
public void add(E e) {
  SList<E> list = listFor(table, e);
  if (! list.contains(e)) {
    list.add(0, e);
    size += 1;
    rehashIfNeeded();
  }
}

public void remove(E e) {
  Iterator<E> listIt = listFor(table, e).iterator();
  while (listIt.hasNext()) {
    E e1 = listIt.next();
    if (e1.equals(e)) {
      listIt.remove();
      size -= 1;
      rehashIfNeeded();
      return;
    }
  }
}
```

Par contre, effectuer un rehachage de la table alors qu'elle est en train d'être parcourue par un itérateur n'est bien entendu pas correct, car la position des éléments est affectée par le rehachage. Dès lors, aucun appel à `rehashIfNeeded` n'est fait dans la méthode `remove` de l'itérateur.