# Anagrammes [20 points]

Une *anagramme* est un mot obtenu par permutation des lettres d'un autre mot. Par exemple, les mots anglais *parental*, *prenatal* et *paternal* sont des anagrammes les uns des autres, car ils sont tous formés des mêmes huit lettres : *a* (2 occurences), *e*, *l*, *n*, *p*, *r* et *t*.

Admettons que l'on dispose d'une liste de mots anglais et qu'on désire y rechercher *toutes* les anagrammes. Une manière élégante et relativement efficace de procéder consiste à utiliser une table de hachage et à y insérer les mots. Si la fonction de hachage est telle que tous les mots composés des mêmes lettres possèdent la même valeur de hachage, alors deux anagrammes se trouveront forcément dans la même liste chaînée de la table. Une fois les mots insérés, il suffira alors de parcourir les listes chaînées de la table de hachage et d'y rechercher les vraies anagrammes.

La classe `Word` ci-dessous, encore incomplète, représente un mot de la langue anglaise. On désire l'utiliser pour rechercher les anagrammes selon la technique décrite ci-dessus.

```
public class Word {
    private final String word;
    public Word(String word) { this.word = word; }
    public static int code(char c) {
        switch (c) {
            case 'a': return 0;
            case 'b': return 1;
            case 'c': return 2;
            // ... et ainsi de suite
            case 'z': return 25;
        }
    }
}
```

**Partie 1 [6 points]** Complétez la classe `Word` afin de lui faire redéfinir les méthodes `equals` et `hashCode` héritées de la classe `Object`.

La méthode `equals` doit simplement comparer les mots lettre à lettre, exactement comme le fait la méthode `equals` de la classe `String`.

La méthode `hashCode` doit produire la même valeur de hachage pour deux mots qui sont des anagrammes. Elle ne peut de plus pas être triviale, dans le sens où la valeur qu'elle retourne doit dépendre des lettres composant le mot auquel on l'applique.

Pour écrire ces méthodes, vous pouvez faire l'hypothèse que les chaînes passées au constructeur de la classe `Word` comportent exclusivement des caractères minuscules et sans accents, tirés de l'alphabet latin (de *a* à *z*).

Pour cette partie comme pour les suivantes, vous avez le droit, au besoin, d'ajouter des champs à la classe `Word` et/ou de modifier son constructeur. De plus, vous pouvez bien entendu faire usage de la méthode statique `code` présentée plus haut.

**Partie 2 [2 points]** Les méthodes `equals` et `hashCode` que vous avez définies dans la partie précédente sont-elles compatibles au sens de Java ? Justifiez votre réponse.

*(Suite au verso)*

**Partie 3 [12 points]** Ajoutez à la classe `Word` une méthode `isAnagramOf` testant si un mot est une anagramme d'un autre, et ayant le profil suivant :

```
public boolean isAnagramOf(Word that)
```

Cette méthode doit retourner vrai si et seulement si le mot stocké dans le récepteur (`this`) est une anagramme du mot stocké dans l'argument `that`. Attention, un mot donné ne doit pas être considéré comme une anagramme de lui-même !

## Itérateurs [20 points]

Les classes `IntSet` et `Interval` ci-dessous permettent de représenter les ensembles d'entiers. La classe `Interval` représente un intervalle par ses bornes (qui sont incluses), tandis que la classe `IntSet` représente un ensemble d'entiers au moyen d'une liste de tels intervalles.

La liste d'intervalles représentant l'ensemble est toujours triée et optimale, dans le sens où les intervalles successifs ne se touchent pas. Par exemple, l'ensemble $\{1, 2, 3, 7, 9, 10, 11\}$ est représenté par la liste des trois intervalles $[1; 3]$, $[7; 7]$ et $[9; 11]$, dans cet ordre. Il n'existe aucune autre liste d'intervalles triée et optimale permettant de représenter cet ensemble.

```
public class IntSet {
    private LinkedList<Interval> intervals;
    public IntSet(int begin, int end) {
        intervals = new LinkedList<Interval>();
        intervals.add(new Interval(begin, end));
    }
    public int size() {
        int size = 0;
        for (Interval i: intervals) size = size + i.size();
        return size;
    }
    public boolean contains(int i) {
        for (Interval interval: intervals)
            if (interval.contains(i))
                return true;
        return false;
    }
    public void union(IntSet that) {
      // ... code omis pour faciliter la lecture
    }
}
public class Interval {
    public final int begin, end;
    public Interval(int begin, int end) {
        this.begin = begin; this.end = end;
    }
    public int size() { return this.end - this.begin + 1; }
    public boolean contains(int x) {
        return (begin <= x) && (x <= end);
    }
}
```

*(Suite au verso)*

**Partie 1 [18 points]** Ajoutez à la classe `IntSet` une méthode `iterator` ayant le profil suivant :

```
public Iterator<Integer> iterator() { ... }
```

où `Iterator` désigne l'interface `java.util.Iterator`, présentée en annexe. Cette méthode doit retourner un itérateur permettant de parcourir les éléments de l'ensemble, du plus petit au plus grand.

L'itérateur retourné par la méthode `iterator` doit posséder une méthode `remove`, sans quoi il ne peut implémenter l'interface `Iterator`. Toutefois, cette méthode doit lever l'exception `UnsupportedOperationException`.

Vous avez bien entendu le droit de définir des classes auxilliaires au besoin.

**Partie 2 [2 points]** Montrez comment il est possible de modifier la déclaration de la classe `IntSet` afin que ses éléments puissent être parcourus au moyen de la boucle *for each* de Java.

## Arbres binaires de recherche [20 points]

Les quatre classes ci-dessous permettent de représenter les ensembles à l'aide d'arbres binaires de recherche. La classe `MyTreeSet` est la classe des ensembles, tandis que les classes `Tree`, `TreeNode` et `TreeLeaf` représentent respectivement les arbres, les nœuds et les feuilles.

```
public class MyTreeSet<E extends Comparable<E>> {
    private Tree<E> root = new TreeLeaf<E>();
    public void add(E newElem) { root = root.add(newElem); }
    public void addAll(E[] newElements) {
        for (E e: newElements)
            add(e);
    }
}
abstract class Tree<E extends Comparable<E>> {
    abstract public Tree<E> add(E newElem);
}
class TreeLeaf<E extends Comparable<E>> extends Tree<E> {
    @Override public Tree<E> add(E newElem) {
        return new TreeNode<E>(newElem,
                               new TreeLeaf<E>(),
                               new TreeLeaf<E>());
    }
}
class TreeNode<E extends Comparable<E>> extends Tree<E> {
    public final E v;
    public Tree<E> s, g;
    public TreeNode(E e, Tree<E> s, Tree<E> g)
        { this.s = s; this.v = e; this.g = g; }
    @Override public Tree<E> add(E newElem) {
        int c = newElem.compareTo(v);
        if (c < 0) s = s.add(newElem);
        else if (c > 0) g = g.add(newElem);
        return this;
    }
}
```

**Partie 1 [6 points]** Dessinez l'arbre contenant les éléments de l'ensemble `set` à la fin du morceau de programme ci-dessous :

```
MyTreeSet<Integer> set = new MyTreeSet<Integer>();
set.addAll(new Integer[] { 7,6,5,4,3,2,1 });
```

Pour dessiner l'arbre, utilisez la même représentation que dans le cours : un nœud est représenté par une boîte contenant l'élément du nœud, tandis qu'une feuille est représentée par un disque plein. De plus, un nœud se trouve toujours au-dessus de ses deux fils, et le fils gauche contient les éléments plus petits que celui du nœud tandis que le fils droit contient les éléments plus grands.

**Partie 2 [4 points]** Quel est le problème de l'arbre contenant les éléments de l'ensemble `set`, que vous avez représenté ci-dessus ?

*(Suite au verso)*

**Partie 3 [10 points]** Afin de résoudre le problème identifié à la partie 2, ajoutez à la classe `MyTreeSet` une méthode nommée `addAllSorted`.

Tout comme `addAll`, cette méthode reçoit un tableau d'éléments, et elle les ajoute tous à l'ensemble via la méthode `add`. Toutefois, elle n'est censée être appelée que avec des tableaux dont les éléments sont triés selon leur ordre naturel, soit de manière croissante soit de manière décroissante. Elle doit tirer parti de cette connaissance pour éviter le problème mentionné à la partie 2.

Note : vous avez le droit d'ajouter une ou plusieurs méthodes privées à la classe `MyTreeSet`, si besoin est. Toutefois, vous ne devez pas modifier les autres classes.

# Formulaire

Ce formulaire présente toutes les parties de la bibliothèque standard Java dont vous avez besoin pour cet examen. Notez que de nombreuses méthodes inutiles ont été omises pour ne pas encombrer inutilement la présentation.

## Classe `String`

La classe `java.lang.String` représente les chaînes de caractères.

```
class String {
    // Retourne la longueur de la chaîne.
    int length();

    // Retourne le caractère à la position index ou lève l'exception
    // IndexOutOfBoundsException si celui-ci est hors des bornes.
    // Le premier caractère de la chaîne est à l'index 0.
    char charAt(int index);

    // Retourne vrai ssi that est également une chaîne de caractères
    // et tous ses caractères sont égaux à ceux du récepteur (this).
    boolean equals(Object that);

    // Retourne une valeur de hachage telle que si deux chaînes sont composées
    // des mêmes caractères dans le même ordre, alors leurs valeurs de hachage
    // sont égales.
    int hashCode();
}
```

## Interface `List`

L'interface `java.util.List` représente les listes. Elle par exemple implémentée par les classes `LinkedList` (listes chaînées) et `ArrayList` (tableaux-listes).

```
interface List<E> {
    // Retourne un itérateur permettant de parcourir les éléments de la liste.
    Iterator<E> iterator();
}
```

## Interface `Iterator`

L'interface `java.util.Iterator` représente les itérateurs.

```
interface Iterator<E> {
    // Retourne vrai ssi cet itérateur peut encore livrer des éléments.
    boolean hasNext();

    // Retourne le prochain élément, ou lève l'exception
    // NoSuchElementException s'il n'y en a plus.
    E next();

    // Supprime la valeur désignée par cet itérateur (opération optionnelle,
    // peut lever l'exception UnsupportedOperationException).
    void remove();
}
```