Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 1

NOM : ..... PRENOM : ..... SECTION : .....

Consignes à lire impérativement !

L'examen est composé de **2 parties**. Chaque partie dure **2 heures**. Il vous est demandé de respecter les consignes suivantes.

- Commencez par écrire vos **nom**, **prénom** et **section** (math, info, ...) sur chaque feuille, y compris les feuilles de brouillon.
- Laissez vos calculatrice, téléphone portable et notes de cours dans votre sac. Leur usage n'est **pas autorisé**. Pensez à éteindre votre téléphone portable !
- Faites attention à la clarté et à l'organisation de vos réponses. Respectez les règles grammaticales et orthographiques.
- Utilisez pour vos réponses les **cadres** prévus à cet effet. Si davantage d'espace est nécessaire, utilisez le dos de la feuille ou une feuille supplémentaire et indiquez clairement où se situe le restant de la réponse.
- Vous devez terminer cette partie de l'examen avant de pouvoir sortir de la salle (pour aller à la toilette par exemple).
- Toutes les feuilles (énoncé et brouillon) doivent être remises en fin d'examen.
- Vérifiez que vous avez répondu à toutes les questions (il y a **2 questions** dans cette partie).

Question 1 – Modélisation de vecteurs (/3)

L'objectif de cette question est la conception et l'implémentation d'une classe `Vector` modélisant des vecteurs dans un espace à deux dimensions. La classe `Vector` doit être **immuable**. Elle doit permettre la **création** d'un nouveau vecteur et la conservation de ses composantes, nommées x et y. Elle doit permettre les **opérations** simples telles que

- le calcul de la norme d'un vecteur (méthode `norm`)
- l'addition de deux vecteurs (méthode `add`)
- la soustraction de deux vecteurs (méthode `sub`)
- la multiplication d'un vecteur par un scalaire (méthode `scale`)
- la rotation d'un vecteur d'un angle α, exprimé en degrés, autour de son origine (méthode `rotate`)

Il vous est demandé de fournir l'implémentation de la classe `Vector` décrite ci-dessus. Vous êtes libres du choix des signatures des méthodes. Cependant, il vous est demandé de mettre en oeuvre autant que possible les principes de programmation orienté-objet et règles de bonne pratique.

Pour rappel, la rotation d'un vecteur autour de son origine peut être réalisée en le multipliant par la matrice suivante. Attention, les fonctions trigonométriques de Java nécessitent des angles exprimés en radians.

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2019

Partie 1

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q1 **(Implémentation de la classe `Vector`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 1

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q1 (suite) **(Implémentation de la classe `Vector`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 1

NOM : ..... PRENOM : ..... SECTION : .....

Question 2 – Annuaire téléphonique (/7)

L'objectif de cette question est d'effectuer des recherches dans un fichier contenant un annuaire téléphonique. Le fichier est structuré en **enregistrements de taille fixe**, chacun contenant un nom et un numéro de téléphone. Le nom est stocké dans une chaîne d'au plus 20 octets, précédée d'un octet contenant la longueur de la chaîne. Le numéro de téléphone est enregistré dans un chaîne de 10 caractères. L'encodage des caractères est ASCII de sorte que chaque caractère tienne sur un seul octet. Un enregistrement a donc toujours une longueur de 31 octets. **Le fichier est trié** selon les noms, en utilisant l'ordre lexicographique classique défini sur les chaînes de caractères.

La Figure 1 illustre l'encodage d'une entrée de l'annuaire. Le premier octet contient la longueur du nom : 8 caractères. Les 8 caractères suivants correspondent au nom : `Bob Hard`. Même si le nom de cette entrée n'occupe que 8 caractères, 20 octets sont prévus pour stocker des noms plus longs. Les 10 derniers octets correspondent au numéro de téléphone : `555/123456`.

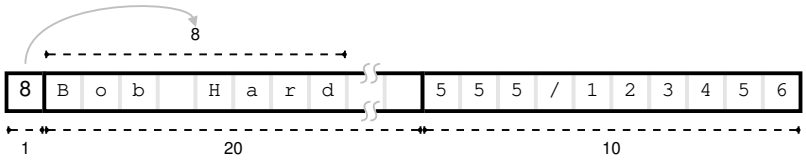

FIGURE 1 – Encodage d'une entrée dans le fichier contenant l'annuaire.

La classe `PhoneBook` illustrée à la Figure 2 permet d'interagir avec l'annuaire téléphonique. A cette fin, la classe utilise la classe `RandomAccessFile` afin d'effectuer des accès aléatoires au fichier contenant l'annuaire. La variable d'instance privée `raf` référence une instance de `RandomAccessFile` permettant de lire le fichier d'annuaire. La création de l'instance de `RandomAccessFile` par le constructeur de `PhoneBook` n'est pas illustrée. La classe interne `Entry` modélise une entrée de l'annuaire, c'est-à-dire un couple (nom, numéro de téléphone).

Il vous est demandé d'implémenter les 3 méthode suivantes de la classe `PhoneBook`.

Q2a Méthode privée `readEntry`. Cette méthode lit un enregistrement à la position actuelle du fichier (`raf`) et retourne une instance de la classe `Entry` initialisée avec le contenu de l'enregistrement. Pour faciliter l'implémentation de cette méthode, les constantes `MAX_NAME_LEN`, `PHONE_LEN` et `ENTRY_LEN` sont définies dans la classe `PhoneBook` et désignent respectivement la longueur maximum d'un nom, la longueur d'un numéro de téléphone et la taille totale d'une entrée. La méthode `readEntry` n'est pas responsable du déplacement du pointeur de fichier au début d'une entrée.

Q2b Méthode `findByPhone`. Cette méthode recherche dans le fichier un enregistrement dont le numéro correspond à celui passé en paramètre (`phone`). Si un tel enregistrement est trouvé, la méthode retourne l'instance de la classe `Entry` qui y correspond. Sinon, la valeur `null` est retournée.

Q2c Méthode `findByName`. Cette méthode recherche dans le fichier un enregistrement dont le nom correspond à celui passé en paramètre (`name`). Si un tel enregistrement est trouvé, la méthode retourne l'instance de la classe `Entry` qui y correspond. Sinon, la valeur `null` est retournée. **Le fichier étant trié selon les noms, il est important d'en tirer parti lors de la recherche!!!**

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2019

Partie 1

NOM : ..... PRENOM : ..... SECTION : .....

```
public class PhoneBook {

  public static final int MAX_NAME_LEN = 20;
  public static final int PHONE_LEN = 10;
  public static final int ENTRY_LEN = 1 + MAX_NAME_LEN + PHONE_LEN;

  private final RandomAccessFile raf;

  private class Entry {
    final String name;
    final String phone;
    public Entry(String name, String phone) {
      this.name = name;
      this.phone = phone;
    }
  }

  private Entry readEntry() { /* ... à implémenter ...                    ( Q2a) */ }

  public Entry findByPhone(String phone) { /* ... à implémenter ...       ( Q2b) */ }

  public Entry findByName(String name) { /* ... à implémenter ...         ( Q2c) */ }

}
```

FIGURE 2 – Extrait de la classe `PhoneBook`.

RandomAccessFile	
`void seek(long` pos`)`	Modifie la position du pointeur de fichier, c'est-à-dire la position où aura lieu la prochaine lecture ou écriture. La nouvelle position correspond à l'argument pos.
`long getFilePointer()`	Retourne la position du pointeur de fichier.
`int skipBytes(int` pos`)`	Effectue un déplacement relatif du pointeur de fichier. Si le pointeur de fichier vaut x, après cet appel il sera déplacé en $x + pos$.
`long length()`	Retourne la longueur du fichier, en octets.
`byte readByte()`	Lit un octet à partir du fichier.
`int read(byte [] b, int off, int len)`	Lit len octets consécutifs à partir du fichier et les stocke dans le tableau b à partir de l'index off. Le tableau doit avoir été créé auparavant.
String	
`String(byte [] b, Charset e)`	Construit une instance de `String` à partir des octets du tableau b. Le paramètre e spécifie l'encodage utilisé pour les caractères. Dans notre cas, e doit valoir `Charset.forName("ascii")`.

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 1

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q2a **(méthode `readEntry`)**

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q2b **(méthode `findByPhone`)**

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q2c **(méthode `findByName`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : ..... PRENOM : ..... SECTION : .....

Consignes à lire impérativement !

L'examen est composé de **2 parties**. Chaque partie dure **2 heures**. Il vous est demandé de respecter les consignes suivantes.

- Commencez par écrire vos **nom**, **prénom** et **section** (math, info, ...) sur chaque feuille, y compris les feuilles de brouillon.
- Laissez vos calculatrice, téléphone portable et notes de cours dans votre sac. Leur usage n'est **pas autorisé**. Pensez à éteindre votre téléphone portable !
- Faites attention à la clarté et à l'organisation de vos réponses. Respectez les règles grammaticales et orthographiques.
- Utilisez pour vos réponses les **cadres** prévus à cet effet. Si davantage d'espace est nécessaire, utilisez le dos de la feuille ou une feuille supplémentaire et indiquez clairement où se situe le restant de la réponse.
- Vous devez terminer cette partie de l'examen avant de pouvoir sortir de la salle (pour aller à la toilette par exemple).
- Toutes les feuilles (énoncé et brouillon) doivent être remises en fin d'examen.
- Vérifiez que vous avez répondu à toutes les questions (il y a **2 questions** dans cette partie).

Question 1 – Table associative (/7)

Une table associative permet de conserver des associations (k, v) entre une clé k et une valeur v. Dans une telle table, il ne peut exister simultanément deux associations de même clé. Une table de hachage est une structure de données qui implémente une table associative à l'aide d'un tableau. La cellule du tableau dans laquelle un couple (k, v) est conservé est déterminée par une fonction de hachage h appliquée à la clé. La fonction de hachage est généralement non-injective, ce qui implique que plusieurs associations clé/valeur peuvent devoir être conservées dans la même cellule. Ceci est typiquement réalisé en maintenant une liste chaînée par cellule.

Un exemple d'une telle table de hachage est illustré à la Figure 1. Le tableau contient 8 cellules. Les clés sont des chaînes de caractères et les valeurs des entiers. La Figure illustre plusieurs cas possibles. Certaines cellules telles que la cellule 2 sont vides. D'autres telles que la cellule 3 référencent une liste d'un seul élément. La cellule 5 est un exemple de collision, les clés `Bob` et `Wolfgang` menant au même index.

L'objectif de cette question est d'implémenter une table associative générique sur base des principes qui viennent d'être énoncés. La classe `HashMap<K,V>` illustrée à la Figure 3 est un embryon de cette implémentation. Elle maintient des associations entre clés de type `K` et valeurs de type `V` à l'aide d'une table de hachage. La variable d'instance `numItems` renseigne à tout moment sur le nombre d'associations actuellement présentes dans la table. Un tableau interne à la classe, nommé `tab`, référence les têtes de listes chaînées d'associations. Ces listes sont **maintenues triées selon les clés**, en utilisant l'ordre naturel du type `K`. Le tableau `tab` doit toujours être d'une **taille égale à un exposant de 2**.

Les listes chaînées reposent sur la classe interne `Node` pour représenter chaque maillon. `Node` contient la référence du maillon suivant (`next`), une clé (`key`) et une valeur (`value`).

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : ..... PRENOM : ..... SECTION : .....

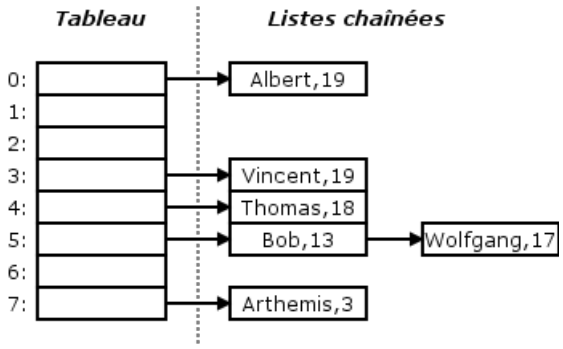

TABLE 1 – Exemple de table de hachage de taille 8 contenant 6 éléments.

La méthode `put(K k, V v)` se charge d'ajouter une association (k, v) à la table de hachage. Son code est donné à la Figure 3. Elle calcule l'index d'insertion dans le tableau grâce à la méthode `Object.hashCode`. Elle effectue ensuite l'insertion dans la liste chaînée correspondante à l'aide de la méthode `insert`. Si l'association a été ajoutée, le compteur d'associations `numItems` est mis à jour et la méthode `resize` est appelée afin d'éventuellement redimensionner la table. Si le load factor, défini comme le rapport entre le nombre d'associations et la taille du tableau, est strictement supérieur à un seuil de 75% (voir constante `MAX_LOAD_FACTOR`), alors la taille du tableau est doublée et chaque association est déplacée de l'ancien vers le nouveau tableau, après avoir recalculé son index.

Ce qui vous est demandé : Afin de compléter la classe `HashMap`, vous devez implémenter les méthodes suivantes.

Q1a Méthode privée `insert(int` $index$`, K` key`, V` $value$`)`. Cette méthode se charge d'insérer l'association (key, $value$) dans la liste chaînée référencée par la cellule $index$ du tableau. La méthode maintient les éléments de la liste ordonnés selon l'ordre naturel des clés. Si une association de même clé existe déjà, sa valeur est simplement remplacée. La méthode retourne `true` en cas d'ajout et `false` en cas de remplacement.

Q1b Méthode privée `resize()`. Cette méthode se charge de re-dimensionner le tableau de listes chaînées lorsque le nombre d'associations devient trop important. La taille du tableau est doublée et toutes les associations sont déplacées vers le nouveau tableau (leur index d'insertion est re-calculé).

Q1c Méthode privée `createNodeArray(int` $size$`)`. Cette méthode instancie un tableau de `Node` de taille $size$. Cette méthode est utilisée par le constructeur pour allouer le tableau initial et par la méthode `resize`. Attention, il y un piège potentiel...

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2019

Partie 2

NOM : ..... PRENOM : ..... SECTION : .....

```
public abstract class HashMap<K extends Comparable,V>
{

  public static final double MAX_LOAD_FACTOR = 0.75;

  /* Definition du maillon d'une liste chaînee */
  private class Node {
    final K key;
    V value;
    Node next;
    public Node(K key, V value, Node next) {
      this.key = key;
      this.value = value;
      this.next = next;
    }
  }

  /* Tableau de listes chaînees */
  private Node [] tab;

  /* Nombre d'associations présentes dans la table */
  private int numItems;

  /* Ajoute une association (key, value) */
  public void put(K key, V value)
    int hash = key.hashCode() & (tab.length - 1);
    if (insert(hash, key, value)) {
      numItems++;
      double loadFactor = (1.0 * numItems) / tab.length;
      if (loadFactor > MAX_LOAD_FACTOR)
        resize();
    }
  }

  /* Insère une association (key, value)
     dans la liste chaînée référencée par la cellule 'index' */
  private boolean insert(int index, K key, V value) { /* a implementer ...   ( Q1a) */ }

  /* Re-dimensionne la table associative */
  private void resize() { /* a implementer ...                                ( Q1b) */ }

  /* Crée un tableau de Node de taille 'size' */
  private Node[] createNodeArray(int size) { /* a implementer ...            ( Q1c) */ }

}
```

FIGURE 3 – Ebauche de la class `HashMap`.

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q1a **(méthode `insert`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q1b **(méthode `resize`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : ..... PRENOM : ..... SECTION : .....

Q1c **(méthode `createNodeArray`)**

.....

.....

.....

.....

.....

Question 2 – Now, scratch your head ! (/3)

Le programme ci-dessous est composé de plusieurs classes et interfaces dont le code source vous est fourni. Le programme principal en bas de page affiche 8 résultats à la console. Vos réponses sont les résultats affichés par le programme.

```
public interface I {
        int compute(int x);
}
```

```
public class B extends A {

        public static int y = 20;

        public int compute(int x) {
                return 2 * x;
        }

}
```

```
public abstract class A implements I {

        public int x = 2;
        public static int y = 10;

        public void setX(int value) {
                this.x = value;
        }
        public int getX() {
                return x;
        }
        public void work() {
                setX(compute(getX()));
                y = compute(y);
        }

}
```

```
public class C extends B {

        int x = 5;

        public void setX(int x) {
                this.x = x;
                y = x;
        }
        public int compute(int x) {
                try {
                        return x * x;
                } finally {
                        return x / 2;
                }
        }

}
```

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2019

Partie 2

NOM : ..... PRENOM : ..... SECTION : .....

```
A b = new B();
C c = new C();

b.work();

System.out.println(b.x);                                    /* ( Q2a) */
System.out.println(c.x);                                    /* ( Q2b) */
System.out.println(b.y);                                    /* ( Q2c) */
System.out.println(c.y);                                    /* ( Q2d) */

c.work();

System.out.println(b.x);                                    /* ( Q2e) */
System.out.println(c.x);                                    /* ( Q2f) */
System.out.println(b.y);                                    /* ( Q2g) */
System.out.println(c.y);                                    /* ( Q2h) */
```

Q2a

.....

.....

.....

.....

Q2b

.....

.....

.....

.....

Q2c

.....

.....

.....

.....

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

Q2d

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Q2e

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Q2f

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Q2g

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Q2h

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : . . . . . PRENOM : . . . . . SECTION : . . . . . .

A Solutions

A.1 Partie 1

A.1.1 Q1 – Vecteur

L'objectif de la question était de créer une classe `Vector` qui devait être **immuable** et mettait en oeuvre les principes de la programmation O.O. Une solution possible à cette question est proposée à la Figure 4.

- Pour réaliser une classe immuable, la solution ci-dessous déclare les composantes `x` et `y` du vecteur comme `final`. Ils sont également `public` de sorte qu'aucun accesseur n'est nécessaire. Une autre possibilité aurait été de déclarer `x` et `y` comme `private` et de fournir uniquement des accesseurs `getX` et `getY`. Par ailleurs, pour assurer l'immuabilité, toutes les méthodes qui "modifient" le vecteur retournent en fait une nouvelle instance de `Vector`. C'est le cas des méthodes `add`, `sub`, `scale` et `rotate`.
- La mise en oeuvre des principes O.O. consistait notamment à être capable d'écrire un constructeur ainsi que les différentes méthodes de la classe. Une méthode agissant sur un vecteur ne doit pas prendre celui-ci en paramètre, mais reposer sur l'instance sur laquelle la méthode est invoquée (référence `this`). Lorsqu'un second vecteur est nécessaire (p.ex. pour `add` et `sub`), celui-ci est fourni sour la forme d'une instance de `Vector` et non de ses composantes `x` et `y`.

Exemples de méthodes considérées comme incorrectes :

- **Non usage de `this`** : le premier argument `a` n'a pas lieu d'être ; il faut utiliser la référence vers l'instance sur laquelle la méthode est appellée (`this`).

```
public Vector add(Vector a, Vector b) {
  return new Vector(a.x + b.x, a.y + b.y);
}
```

- **Non usage d'une instance de `Vector`** : le vecteur ajouté doit l'être sous forme d'une instance de `Vector` plutôt que de ses composantes `x` et `y`.

```
public Vector add(double x, double y) {
  return new Vector(this.x + x, this.y + y);
}
```

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : ..... PRENOM : ..... SECTION : .....


```java
public class Vector {

  public final double x;
  public final double y;

  public Vector(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double norm() {
    return Math.sqrt(x * x + y * y);
  }

  public Vector add(Vector o) {
    return new Vector(x + o.x, y + o.y);
  }

  public Vector sub(Vector o) {
    return new Vector(x - o.x, y - o.y);
  }

  public Vector scale(double f) {
    return new Vector(x * f, y * f);
  }

  public Vector rotate(double aDeg) {
    double aRad = Math.PI * aDeg / 180;
    return new Vector(
      x * Math.cos(aRad) - y * Math.sin(aRad),
      x * Math.sin(aRad) + y * Math.cos(aRad)
    );
  }

}
```

FIGURE 4 – Implémentation possible de la classe `Vector`.

A.1.2 Q2 – Annuaire téléphonique

a). La **méthode `readEntry`** se charge de lire une entrée dans le fichier à la position actuelle. Elle ne déplace donc pas le pointeur de fichier avant de commencer la lecture. La méthode procède comme suit : lecture de la longueur du nom (ligne 2), lecture du nom et création d'une instance de `String` (lignes 3 à 5), lecture du numéro de téléphone et création d'une `String` (lignes 8 à 10), création d'une instance de `Entry`. La méthode `read` de `RandomAccessFile` avance le pointeur de fichier du nombre d'octets lus. Les lignes 6 et 7 avancent dans le fichier dans le cas où la longueur du nom est inférieure à la taille maximum (`MAX_NAME_LEN`).

```
1   private Entry readEntry() throws IOException {
2     byte len = raf.readByte();
3     byte [] name_bytes = new byte[len];
4     raf.read(name_bytes, 0, len);
5     String name = new String(name_bytes, Charset.forName("ascii"));
6     if (len < MAX_NAME_LEN)
7       raf.skipBytes(MAX_NAME_LEN - len);
8     byte [] number_bytes = new byte[PHONE_LEN];
9     raf.read(number_bytes, 0, PHONE_LEN);
10    String number = new String(number_bytes);
11    return new Entry(name, number);
12  }
```

b). La **méthode `findByPhone`** effectue une recherche linéaire dans le fichier. Elle commence au début du fichier, lit une entrée (tâche déléguée à la méthode `readEntry`), la compare au numéro recherché et continue ainsi tant que le numéro n'est pas trouvé et que la fin de fichier n'est pas atteinte.

```
1   public Entry findByPhone(String phone) throws IOException {
2     raf.seek(0);
3     while (raf.getFilePointer() < raf.length()) {
4       Entry e = readEntry();
5       if (e.phone.equals(phone))
6         return e;
7     }
8     return null;
9   }
```

c). La **méthode `findByName`** effectue une recherche dichotomique dans le fichier, profitant du fait que les entrées sont triées par nom. La seule difficulté de cette méthode par rapport à une recherche dichotomique classique est que les positions doivent correspondre à des multiples entiers de la taille d'une entrée (`ENTRY_LEN`). A cette fin, la longueur du fichier est divisée initialement par `ENTRY_LEN` (la longueur du fichier est un multiple entier de `ENTRY_LEN` par construction). Ensuite, à chaque fois qu'il faut se positionner sur une entrée, l'index `mid` est multiplié par `ENTRY_LEN` afin de déterminer la bonne position dans le fichier. La lecture d'une entrée est déléguée à `readEntry`.

```
1   public Entry findByName(String name) throws IOException {
2     long start = 0;
3     long end = raf.length() / ENTRY_LEN;
4     raf.seek(0);
5     while (start < end) {
6       long mid = (start + end) / 2;
7       raf.seek(mid * ENTRY_LEN);
8       Entry e = readEntry();
9       int cmp = name.compareTo(e.name);
10      if (cmp == 0)
11        return e;
12      if (cmp < 0)
13        end = mid;
14      else
15        start = mid + 1;
16    }
17    return null;
18  }
```

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2019
Partie 2

NOM : ..... PRENOM : ..... SECTION : .....

A.2 Partie 2

A.2.1 Q1 – Table associative

a). La **méthode `insert`** se charge d'insérer une association dans la liste chaînée référencée par la cellule `index` du tableau. Les listes chaînées sont triées sur les clés (`key`). La méthode est découpée en trois grandes parties.

1 Les lignes 5 à 11 insèrent un élément en tête de liste, ce qui se produit (1) si la chaîne est vide ou (2) si l'élément à insérer précède ($\leq$) la tête de liste. Dans le cas où l'association à insérer est de même clé que la tête de liste, un remplacement est effectué (seule la variable `value` de la tête de liste est changée).

2 Les lignes 13 et 14 recherchent la position d'insertion dans la liste. Le cas de l'insertion en tête de liste est déjà règlé, par conséquent, il est possible de se souvenir du prédécesseur du point d'insertion. Ce prédécesseur est conservé dans la variable `tmp`. La recherche s'arrête lorsque la fin de liste est atteinte ou lorsque le noeud suivant est de clé supérieure au nouvel élément.

3 A ce stade, il y a 3 possibilités traitées dans les lignes 16 à 21 : (1) la fin de liste est atteinte (`tmp.next` est `null`), (2) le noeud suivant est strictement supérieur (`cmp` > 0) et (3) le noeud suivant est de même clé (`cmp` $= 0$). Dans les cas (1) et (2) on procède à une insertion d'un nouveau noeud. Dans le cas (3), il s'agit d'un remplacement.

```
1   private boolean insert(int index, K key, V value) {
2     Node tmp = tab[index];
3     int cmp;
4
5     if ((tmp == null) || ((cmp = tmp.key.compareTo(key)) > 0)) {
6       tab[index] = new Node(key, value, tab[index]);
7       return true;
8     } else if (cmp == 0) {
9       tab[index].value = value;
10      return false;
11    }
12
13    while ((tmp.next != null) && ((cmp = tmp.next.key.compareTo(key)) < 0))
14      tmp = tmp.next;
15
16    if ((tmp.next == null) || (cmp > 0)) {
17      tmp.next = new Node(key, value, tmp.next);
18      return true;
19    }
20    tmp.next.value = value;
21    return false;
22  }
```

b). La **méthode `resize`** se charge de doubler la taille du tableau et de ré-insérer chaque association. En effet, l'index d'insertion dans le tableau peut changer étant donné qu'il dépend de la taille du tableau. Pour ce faire, l'ensemble des associations est parcouru à l'aide de deux boucles imbriquées. La boucle externe itère sur les cellules du tableau tandis que la boucle interne itère sur les éléments des listes chaînées. Pour chaque association trouvée, son nouvel index d'insertion est calculé et l'élément est inséré à l'aide de la méthode `insert`.

```
1  private void resize() {
2    Node [] oldTab = tab;
3    tab = buildNodeTab(oldTab.length * 2);
4    for (int i = 0; i < oldTab.length; i++) {
5      Node tmp = oldTab[i];
6      while (tmp != null) {
7        int hash = tmp.key.hashCode() & (tab.length - 1);
8        insert(hash, tmp.key, tmp.value);
9        tmp = tmp.next;
10     }
11   }
12 }
```

c). La **méthode `createNodeArray`** doit instancier un tableau dont les éléments sont de type `Node`. La difficulté est que `Node` est en fait un type générique : `HashMap<E>.Node`. Or, le langage Java refuse l'instanciation de tableaux génériques. Afin de contourner cette limitation, il suffit d'instancier un tableau dont les cellules sont de type `HashMap.Node` (non-générique) et d'effectuer un transtypage vers `HashMap<E>.Node`. Ce transtypage génère un avertissement qu'il est possible de supprimer à l'aide de l'annotation `@SuppressWarnings`.

```
@SuppressWarnings(value = "unchecked")
private Node[] createNodeArray(int size) {
  return (Node[]) new HashMap.Node[size];
}
```

A.2.2 Q2 – Scratch your head !

Vous pouvez déterminer les réponses à cette question par vous-même (en exécutant le code donné avec la JVM).