Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2021

Partie 1

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

Consignes à lire impérativement !

L'examen est composé de **2 parties**. Chaque partie dure **1h30**. 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 **3 questions** dans cette partie).

Question 1 – Analyseur de texte (/4)

L'objectif de cette question est l'extraction des mots d'un texte. Dans le contexte de cette question, un texte ne comprend que des lettres, des symboles de ponctuation, des tirets et des espaces. Un mot est composé d'une suite de lettres consécutives. Les mots sont séparés par un ou plusieurs espaces, symboles de ponctuation ou tirets. Un mot peut contenir un tiret unique à condition que celui-ci ne soit situé ni au début du mot ni à sa fin. Si un mot contient deux tirets consécutifs ou plus, alors ces tirets ne font pas partie du mot et séparent le mot en deux parties.

Pour répondre à cette question, vous devez écrire une classe Java nommée `Tokenizer`. Le constructeur de la classe prend en argument la chaîne de caractères à analyser. La méthode `next` permet ensuite d'extraire chacun des mots, un par appel. Elle retourne `null` lorsqu'il n'y a plus de mot à extraire. Cette classe est donc typiquement utilisée comme dans l'exemple ci-dessous. La suite de 5 mots produite par cet exemple est : `cette`, `super-string`, `dé-montre`, `le`, `tokenizer`.

```
Tokenizer tk = new Tokenizer("cette- super-string dé-montre-le--tokenizer");
String s;
while ((s = tk.next()) != null)
    System.out.println(s);
```

Dans votre implémentation, **vous ne pouvez** pas utiliser les méthodes de découpage de chaînes de caractères déjà fournies par une librairie Java (telles que p.ex. `String.split`).

A la page suivante, vous trouverez une ébauche de la classe `Tokenizer` ainsi que la documentation de plusieurs méthodes qui vous seront utiles, telles que `String.charAt` et `String.substring` ou `Character.isLetter`.

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2021

Partie 1

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

```
public class Tokenizer {

    private String s;
    /* à compléter */

    public Tokenizer(String s) {
        this.s = s;
    }

    public String next() {
        /* à compléter */
    }

}
```

String

char charAt(**int** i)	Donne le caractère d'index i de la chaîne.
String substring(**int** b, **int** e)	Retourne la sous-chaîne commençant à l'index b et s'étendant jusqu'à l'index $e-1$. Les index doivent respecter la contrainte suivante : $0 \leq b \leq e \leq N$ où N est la longueur de la chaîne.
String **toLowerCase()**	Retourne une copie de la chaîne, en minuscules.

Character

boolean isLetter(**char** c)	Retourne **true** si c est une lettre, **false** sinon.

Q1 **(classe `Tokenizer`)**

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 1

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

Q1 (suite) **(classe `Tokenizer`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 1

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

Question 2 – Fréquence des mots (/2)

En utilisant la classe `Tokenizer` documentée à la question précédente, écrivez un algorithme qui détermine la fréquence de chaque mot dans un texte. Par exemple, pour le texte ci-dessous à gauche, l'analyse de texte devrait retourner le résultat montré dans le tableau ci-dessous à droite.

≪ *Le type a surgi sur le boulevard,*
sur sa grosse moto super-chouette. ≫
(R. Séchan)

Mot	Fréquence
`le`	2
`type`	1
`a`	1
`surgi`	1
`sur`	2
`boulevard`	1
`sa`	1
`grosse`	1
`moto`	1
`super-chouette`	1

Pour répondre à cette question, vous devez écrire une méthode `frequencyAnalysis` qui prend un argument unique de type `String`. En utilisant `Tokenizer`, la méthode extrait chaque mot de la chaîne et enregistre dans une structure de données de votre choix le nombre de fois que chaque mot est apparu dans la chaîne (sa fréquence). La méthode retourne ensuite cette structure de données à l'appelant.

Q2 **(méthode `frequencyAnalysis`)**

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 1

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

Q2 **(méthode `frequencyAnalysis`)**

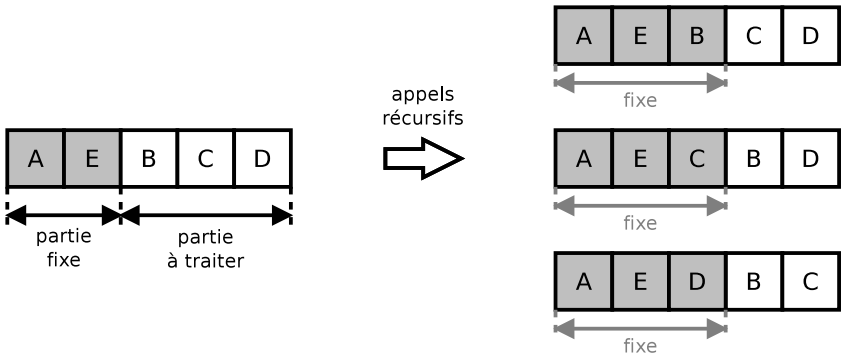
Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 1

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

Question 3 – Permutations (/4)

L'objectif de cette question est la conception et l'implémentation en Java d'un algorithme permettant de générer toutes les permutations d'une séquence de N caractères **différents**. Prenons, par exemple, la séquence de 3 caractères `ABC`. Il existe 6 permutations différentes de cette séquence : `ABC`, `ACB`, `BAC`, `BCA`, `CAB` et `CBA`.

Voici comment un tel algorithme pourrait procéder récursivement. L'algorithme considère que la séquence de caractères à traiter est composée de deux parties : la partie de gauche qui ne peut plus être changée (déjà traitée) et la partie de droite qui doit encore être permutée. Cette situation est illustrée ci-dessous. La chaîne initiale est `ABCDE`. Lors de l'appel actuel de l'algorithme, les deux premiers caractères de la séquence sont fixés. Il reste donc à générer les permutations qui commencent par `AE` et qui se terminent par toutes les permutations possibles de `BCD`.



L'algorithme doit récursivement générer toutes les permutations de la partie de droite. Pour cela, il prend un caractère restant dans la partie de droite et il ajoute ce caractère à la partie fixe de gauche. Il s'appelle ensuite récursivement pour la partie de droite diminuée du caractère fixé. L'exemple de la figure illustre le fait que 3 appels récursifs sont effectués. Par exemple, le premier de ces appels va générer les chaînes commençant par `AEB` et se terminant par les permutations de `CD`.

Ce qui vous est demandé : Implémenter dans une méthode nommée `permutations` l'algorithme récursif génerant toutes les permutations d'une chaîne de N caractères **différents**. Cette méthode prend les arguments suivants : str la chaîne de caractères et i la longueur de la partie fixe. La méthode retourne une collection contenant toutes les permutations générées.

Le code suivant donne un exemple d'invocation de la méthode `permutations`.

```
for (String s: permutations("ABCDE", 0))
    System.out.println(s);
```

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

Q3 **(implémentation de la méthode `permutations`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 2

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

Consignes à lire impérativement !

L'examen est composé de **2 parties**. Chaque partie dure **1h30**. 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 – LinkedHashMap (/7)

La bibliothèque Java fournit trois implémentations différentes de l'ADT Association : `HashMap`, `TreeMap` et `LinkedHashMap`. Ces trois classes fournissent les mêmes services mais diffèrent dans leur implémentation. `HashMap` utilise une table de hachage, `TreeMap` utilise un arbre binaire de recherche tandis que `LinkedHashMap` utilise à la fois une table de hachage et une liste chaînée. Nous nous focalisons sur cette dernière classe dans cette question.

Pourquoi utiliser à la fois une table de hachage et une liste chaînée ? De la même manière que dans `HashMap`, la table de hachage permet le stockage des paires (clé, valeur). La liste chaînée permet d'itérer sur tous les éléments de `LinkedHashMap` selon leur ordre d'ajout, une fonctionnalité qui n'existe pas avec `HashMap`.

La Figure 1 illustre l'organisation d'une `LinkedHashMap`. Le coeur de la structure est un tableau qui référence des paires (clé, valeur). Ces paires sont stockées au sein d'instances de la classe `Entry` (représentées par des rectangles grisés). Les instances d'`Entry` peuvent être liées entre elles grâce à deux variables références : `next` et `after`. La variable `next`, illustrée avec une flèche en trait plein, désigne le noeud suivant dans la liste de collision, de la même manière que dans une table de hachage classique. La variable `after`, illustrée avec un trait pointillé, permet de créer une liste chaînée globale de toutes les paires de la `LinkedHashMap`. Dans l'exemple de la Figure 1, trois paires sont stockées dans la table. Elles ont été ajoutées dans l'ordre suivant : ("`aa`", 42), ("`bB`", 17) et ("`toto`", 123). Le parcours de la liste globale donnera ces paires dans l'ordre inverse de leur ajout : ("`toto`", 123), ("`bB`", 17) et ("`aa`", 42). Les clés "`aa`" et "`bB`" ont la même valeur de hachage et sont donc conservées dans une liste de collision.

Une implémentation partielle de la classe `LinkedHashMap` est donnée à la Figure 2. Elle contient une référence `tab` vers le tableau interne ainsi qu'une référence `head` vers la tête de liste globale. Elle contient également la définition de la classe interne `Entry` contenant une paire clé (`k`) et valeur (`v`) ainsi que les références `next` et `after` vers, respectivement, le noeud suivant dans la liste de collision et dans la liste globale.

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2021

Partie 2

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

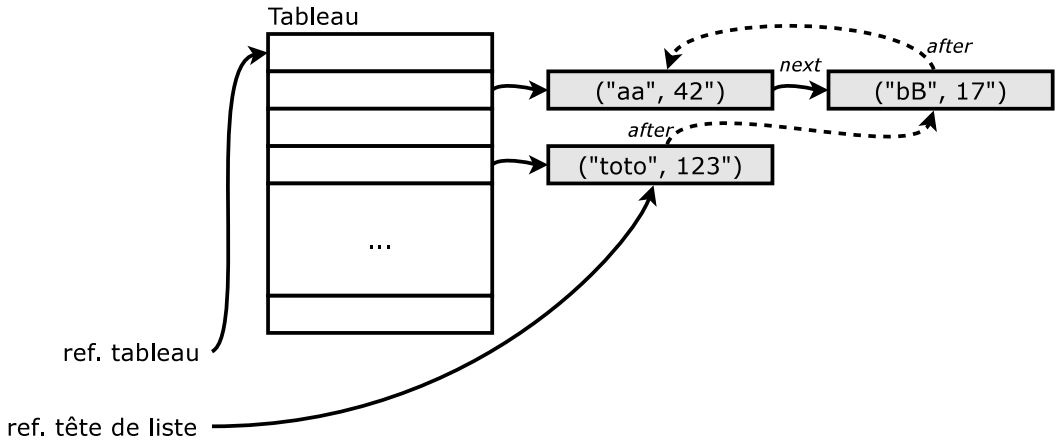


FIGURE 1 – Illustration d'une `LinkedHashMap`. Les rectangles gris correspondent aux instances de la classe interne `Entry` et contiennent les paires (clé, valeur). Les flèches en trait plein ("`next`") désignent les noeuds suivants dans une liste de collisions tandis que celles en trait pointillé ("`after`") désignent les noeuds suivants dans la liste globale.

```
public class LinkedHashMap {

    private class Entry {
        public Object k, v;
        public Entry next;  // suivant dans liste collisions
        public Entry after; // suivant dans liste globale
    }

    public static final int TAB_SIZE = 1024; // exposant de 2
    private Entry [] tab; // ref. tableau
    private Entry head;   // ref. tete liste globale

    public void put(Object k, Object v) { /* à compléter */ }                ( Q1a)
    public Object get(Object k) { /* à compléter */ }                        ( Q1b)

}
```

FIGURE 2 – Implémentation partielle de la classe `LinkedHashMap`.

Pour répondre à cette question, il vous est demandé de compléter les méthodes `put` et `get`.

Q1a méthode `put` : ajoute la paire (clé, valeur) à la table de hachage et en tête de la liste globale. La position dans la table de hachage dépend du `hashcode` de la clé. Si la clé existe déjà, la valeur associée est mise à jour, mais aucun ajout à la liste globale n'est réalisé. Si plusieurs paires ont le même index dans le tableau, elles sont chaînées au sein d'une liste de collisions.

Note : l'ajout des paires en tête de liste globale, entraîne que le parcours de cette liste donne les paires dans l'ordre inverse de leur ajout.

Q1b méthode `get` : permet de récupérer la valeur associée à une clé si elle existe ou `null` dans le cas contraire.

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 2

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

Q1a **(méthode `put`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 2

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

Q1a (suite) **(méthode `put`)**

Q1b **(méthode `get`)**

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 2

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

Question 2 – Itérateur (/3)

L'interface `Iterator` permet d'itérer sur un ensemble d'éléments, par exemple toutes les valeurs d'une instance d'une `Map`, d'une `Collection` ou encore d'un tableau. Cette interface spécifie deux méthodes. La méthode `hasNext` retourne `true` s'il reste au moins un élément sur lequel itérer, `false` sinon. La méthode `next` retourne l'élément suivant à récupérer et fait avancer l'itérateur.

```
public interface Iterator {
    boolean hasNext();
    Object  next();
}
```

Le langage Java permet d'utiliser la boucle `for` (à gauche) comme sucre syntaxique pour une boucle utilisant un itérateur et ses méthodes `hasNext` et `next`. L'exemple ci-dessous montre que la boucle `for` (à gauche) est en fait traduite par le compilateur en une boucle utilisant un itérateur (à droite).

```
Map m = /* ... */
for (Object v: m)
    System.out.println(v);
```

```
Map m = /* ... */
Iterator iter = m.iterator();
while (iter.hasNext()) {
    Object v = iter.next();
    System.out.println(v);
}
```

Votre objectif est de concevoir une classe nommée `InternalIterator` qui soit interne à la classe `LinkedHashMap` et qui implémente l'interface `Iterator`. Cette classe doit permettre d'itérer sur toutes les valeurs d'une instance de `LinkedHashMap`. A cet effet, la classe `InternalIterator` a un accès direct à la variable `head` qui référence la tête de liste globale de `LinkedHashMap`.

Pour implémenter `InternalIterator`, il vous est conseillé de d'abord écrire le code permettant d'itérer sur tous les éléments de la liste globale de `LinkedHashMap`. Identifiez quel état est maintenu dans cette boucle pour déterminer la position de l'itération. Cet état doit être conservé dans la classe `InternalIterator` et manipulé par les méthodes `hasNext` et `next`.

Q2 **(classe `InternalIterator`)**

.....

.....

.....

.....

.....

.....

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 2

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

Q2 (suite) **(classe `InternalIterator`)**

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2021

Partie 2

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

```
public class Tokenizer {

    private String s;
    private int pos;

    public Tokenizer(String s) {
        this.s = s;
    }

    public String next() {
        while ((pos < s.length()) && !Character.isLetter(s.charAt(pos)))
            pos++;
        if (pos == s.length())
            return null;
        var lastPos = pos;
        while ((pos < s.length()) && Character.isLetter(s.charAt(pos)))
            pos++;
        if ((pos == s.length()) || (s.charAt(pos) != '-'))
            return s.substring(lastPos, pos);
        pos++;
        if ((pos == s.length()) || !Character.isLetter(s.charAt(pos)))
            return s.substring(lastPos, pos-1);
        while ((pos < s.length()) && Character.isLetter(s.charAt(pos)))
            pos++;
        return s.substring(lastPos, pos);
    }

    public static void main(String [] args) {
        Tokenizer tk = new Tokenizer("cette-␣super-string␣dé-montre-le--tokenizer");
        String s;
        while ((s = tk.next()) != null)
            System.out.println(s);
    }

}
```

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2021

Partie 2

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

```
import java.util.Map;
import java.util.LinkedHashMap;

public class FrequencyAnalysis {

    public static Map<String,Integer> frequencyAnalysis(String s) {
        Map<String,Integer> m = new LinkedHashMap<>();
        Tokenizer tk = new Tokenizer(s);
        String w;
        while ((w = tk.next()) != null) {
            w = w.toLowerCase();
            if (m.containsKey(w))
                m.put(w, m.get(w) + 1);
            else
                m.put(w, 1);
        }
        return m;
    }

    public static void main(String [] args) {
        var s = "Le␣type␣a␣surgi␣sur␣le␣boulevard," +
                "sur␣sa␣grosse␣moto␣super-chouette.";
        var m = frequencyAnalysis(s);
        for (Map.Entry<String,Integer> e: m.entrySet())
            System.out.println(e.getKey() + "=" + e.getValue());
    }

}
```

Examen du cours de Programmation et Algorithmique II

1ère Session, Juin 2021

Partie 2

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

```java
import java.util.List;
import java.util.ArrayList;

public class Permutations {

    public static List<String> permutations(String s, int fix) {
        List<String> al = new ArrayList<>();
        if (fix == s.length()) {
            al.add(s);
        } else {
            for (int i = 0; i < s.length()-fix; i++) {
                String subS = s.substring(0, fix) + s.charAt(fix + i) +
                    s.substring(fix, fix + i ) + s.substring(fix+i+1, s.length());
                al.addAll( permutations(subS, fix + 1) );
            }
        }
        return al;
    }

    public static void main(String [] args) {
        for (String s: permutations("ABCD", 0))
            System.out.println(s);
    }

}
```

Examen du cours de Programmation et Algorithmique II
1ère Session, Juin 2021
Partie 2

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

```
import java.util.Iterator;
import java.util.Iterable;

public class LinkedHashMap implements Iterable<Object> {

    public static final int TAB_SIZE = 1024;

    private Entry [] tab;
    private Entry list;
    private Entry listLast;

    private class Entry {
        Object k, v;
        Entry next;  // dans la liste de collision
        Entry after; // dans la liste d'itération
        public Entry(Object k, Object v, Entry next) {
            this.k = k;
            this.v = v;
            this.next = next;
        }
        public Entry(Object k, Object v) {
            this(k, v, null);
        }
    }

    public LinkedHashMap() {
        tab = new Entry[TAB_SIZE];
        list = null;
    }

    private static int hash(Object k) {
        return k.hashCode() & (TAB_SIZE - 1);
    }

    public void put(Object k, Object v) {
        int h = hash(k);
        if (tab[h] == null) {
            tab[h] = new Entry(k, v);
            tab[h].after = list;
            list = tab[h];
        } else {
            Entry tmp = tab[h];
            while (tmp != null) {
                if (tmp.k.equals(k)) {
                    tmp.v = v;
                    return;
                }
                tmp = tmp.next;
            }
            tab[h] = new Entry(k, v, tab[h]);
            tab[h].after = list;
```

```
            list = tab[h];
        }

    }

    public Object get(Object k) {
        int h = hash(k);
        if (tab[h] == null)
            return null;
        Entry tmp = tab[h];
        while (tmp != null) {
            if (tmp.k.equals(k))
                return tmp.v;
            tmp = tmp.next;
        }
        return null;
    }

    public Iterator<Object> iterator() {
        return new Iterator<Object>() {
            Entry after = list;
            public boolean hasNext() {
                return (after != null);
            }
            public Object next() {
                Object v = after.v;
                after = after.after;
                return v;
            }
        };
    }

    public static void main(String [] args) {
        LinkedHashMap m = new LinkedHashMap();
        m.put("toto", 42);
        m.put("lulu", 22);
        m.put("aa", 17);
        m.put("bB", 23);

        System.out.println(m.get("toto"));
        System.out.println(m.get("lulu"));
        System.out.println(m.get("aa"));
        System.out.println(m.get("bB"));

        System.out.println("***␣1st␣iteration␣***");
            for (Object o: m)
        System.out.println(o);

        m.put("toto", 43);

        System.out.println("***␣2nd␣iteration␣***");
        for (Object o: m)
```

```
            System.out.println(o);
    }

}
```