


This assignment will be closed on May 22, 2019.

On May 14, 2019 (14:07:10), we had submissions for the following questions: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25 — [Details](#)  ([/agns/INF371/TD03/2018/uploads/:my/](https://x.strub.nu/agns/INF371/TD03/2018/uploads/:my/))

# Le 36<sup>e</sup> chapitre de «Madame Bovary»

Jean-Marc Notin, Luc Maranget

- Description de l'algorithme
  - Construction de la table associative
  - Génération de texte
  - Structures de données
- Les listes chaînées
  - Premières fonctions sur les maillons
  - Maillons vs liste chaînée
  - Opérations sur les listes chaînées
- Manipulation des préfixes
  - Égalités
- Tables de hachage
  - Fonction de hachage
  - Définition de la table associative
    - Gestion des entrées dans la table
    - Redimensionnement du tableau d'indice
- Madame Bovary
  - Lecture du texte
  - Construction de la table associative
  - Génération de texte
  - Le 36<sup>e</sup> chapitre de Mme Bovary

Le but de ce TD est d'écrire un programme Java qui va générer un pseudo texte à partir des 35 chapitres du livre de Gustave Flaubert. La méthode employée pour la génération du texte est basée sur les chaînes de Markov.

Le texte généré aura les propriétés suivantes : Étant donné un entier  $n \geq 1$

- Toutes les sous-suites de  $n + 1$  mots du texte généré sont des sous-suites de  $n + 1$  mots du texte de *Madame Bovary*.
- Les  $n$  premiers mots du texte sont les  $n$  premiers mots d'un chapitre de *Madame Bovary*.
- Les  $n$  derniers mots du texte sont aussi les  $n$  derniers mots d'un chapitre de *Madame Bovary*.

L'entier  $n$  est donné en argument de la ligne de commande (ou, dans Eclipse, dans le menu *Run > Run Configurations...* > *Java Application*, onglet *Arguments*). Ainsi, la commande

```
% java Bovary 3
```

produit un texte plutôt étrange, par exemple :

Un soir que la fenêtre était ouverte, et que, assise au bord, elle venait de s'échapper dans le jardin. Emma, tout exprès, avait retiré la clef de sol, et s'occupait volontiers de littérature après son dîner, quand il ne jouait pas aux cartes. M. Homais le considérait pour son instruction ; madame Homais l'affectionnait pour sa complaisance, car souvent il accompagnait au jardin les petits Homais, marmots toujours barbouillés, fort mal élevés et quelque peu lymphatiques, comme leur mère. Ils avaient pour les soigner, outre la bonne, Justin, l'élève en pharmacie, un arrière-cousin de M. Homais et peut-être les pesanteurs du déjeuner, restait indécis et comme sous la fascination du pharmacien qui répétait : [...]

Le texte ci-dessus est de toute évidence dérivé de l'œuvre de Flaubert. « Dérivé » est d'ailleurs le mot juste, on perçoit un certain flottement dans l'écriture.

On notera par exemple, que le chapitre numéro 28 commence par « *Un soir que [Charles l'écoutait,...]* ». La phrase « *Emma, tout exprès, avait retiré la clef de sol, [...]* » n'est pas de Flaubert, mais Flaubert a écrit :

- « *Emma, tout exprès, avait retiré la clef de la barrière, que Charles crut perdue.* » Une manœuvre d'Emma pour faciliter ses rencontres avec Rodolphe, chap. 19.
- « *Puis il possédait des talents, il peignait à l'aquarelle, savait lire la clef de sol, et s'occupait volontiers de littérature après son dîner, quand il ne jouait pas aux cartes.* » Une description de Léon, le clerc de notaire, chap. 12.

On remarquera que « *retiré la clef de* » et « *la clef de sol,* » sont deux suites de  $n + 1 = 4$  mots qui permettent de relier la clef de la barrière à la clef de sol.

Ce petit exercice pseudo-littéraire sera le prétexte de découvrir en Java quelques unes des structures de données classiques utilisées en informatique.

## Description de l'algorithme

L'algorithme emploie une table associative dont les clés sont des séquences de  $n$  mots et les valeurs associées des multi-ensembles (ensembles avec répétitions possibles des éléments) de mots. La première phase de l'algorithme construit la table à partir du texte de *Madame Bovary*; la seconde phase utilise la table pour produire un nouveau texte.

### Construction de la table associative

Le principe de cette première phase est de construire une table associant à chaque sous-séquence  $P$  de  $n$  mots du texte, le multi-ensemble des mots qui suivent immédiatement  $P$ . L'algorithme doit donc parcourir les 35 chapitres de *Mme Bovary*, lire chaque mot et l'associer à la séquence des  $n$  mots précédemment lue. Une telle séquence est appelée *préfixe* et est notée  $w_1, w_2, \dots, w_{n-1}, w_n$ .

Pour construire la table associative, nous utiliserons l'algorithme suivant :

Pour chaque chapitre :

- Le préfixe  $P$  est initialisé à  $\langle \text{START} \rangle, \langle \text{START} \rangle, \dots, \langle \text{START} \rangle$ .
- Pour chaque mot  $w$  du chapitre en cours :
  - on ajoute  $w$  aux mots associés à  $P$ ,
  - le préfixe  $P$  devient  $w_2, w_3, \dots, w_n, w$ .
- À la fin, on ajoute  $\langle \text{END} \rangle$  aux mots associés à  $P$ .

Les chaînes de caractères  $\langle \text{START} \rangle$  et  $\langle \text{END} \rangle$  sont des marqueurs spéciaux servant à représenter respectivement le début et la fin de chaque chapitre. Ils n'apparaissent explicitement pas dans le texte.

### Génération de texte

Une fois la table associative construite, nous utiliserons l'algorithme suivant pour générer un nouveau texte :

- Le préfixe  $P$  est initialisé à  $\langle \text{START} \rangle, \langle \text{START} \rangle, \dots, \langle \text{START} \rangle$ .
- Répéter :
  - choisir  $w$  parmi les mots associés à  $P$ , selon une loi uniforme,
  - si  $w$  est  $\langle \text{END} \rangle$ , alors terminer,
  - sinon :
    - on affiche  $w$ ,
    - le préfixe  $P$  devient  $w_2, w_3, \dots, w_n, w$ .

## Structures de données

Dans cet algorithme, il est nécessaire de manipuler plusieurs structures de données.

- D'une part, nous devons représenter des multiensembles (<https://fr.wikipedia.org/wiki/Multiensemble>) de mots (c'est-à-dire des ensembles de mots dans lesquels il peut y avoir plusieurs fois le même mot). La taille de ceux-ci ne pouvant être connue à l'avance, il est judicieux de choisir une structure de données facilement extensible. Nous utiliserons donc une structure de liste chaînée ([https://fr.wikipedia.org/wiki/Liste\\_cha%C3%A9n%C3%A9e](https://fr.wikipedia.org/wiki/Liste_cha%C3%A9n%C3%A9e)) pour implémenter un multi-ensemble de mots. *De fait, les premières questions sont classiques et destinées à vous familiariser avec la manière dont les listes chaînées sont implémentées en Java.*
- D'autre part, nous utilisons une table associative qui, à la manière d'un dictionnaire, associe une *valeur* à une *clé* donnée. Pour une telle structure de données, il est essentiel de garantir une recherche rapide de la clé. Nous implémenterons donc la table associative à l'aide d'une table de hachage ([https://fr.wikipedia.org/wiki/Table\\_de\\_hachage](https://fr.wikipedia.org/wiki/Table_de_hachage)).

Si vous travaillez sous Eclipse, commencez par créer un nouveau projet Java pour le TD.

## Les listes chaînées

Une liste chaînée est implémentée par une suite de maillons chaînés les uns aux autres : chaque maillon contient la référence du maillon suivant. Ainsi, avec la référence vers le premier maillon, on peut accéder successivement à tous les maillons de la chaîne. À la fin de la chaîne, on met la référence du maillon suivant à `null`.

Concrètement, les maillons d'une liste chaînée sont déclarés de la manière suivante :

```
class Node {
    String head;
    Node next;

    Node(String head, Node next) {
        this.head = head;
        this.next = next;
    }
}
```

Le champ `head` permet de stocker la valeur associée à un maillon et le champ `next` permet d'accéder au maillon suivant dans la chaîne. La classe `Node` contient également la définition d'un constructeur permettant la création d'un nouveau maillon à partir d'une valeur et d'une référence à un maillon.

Pour créer en Java la chaîne correspondant à la liste `["foo", "bar", "baz"]`, on écrira le code suivant :

```
Node foobar = new Node("foo", new Node("bar", new Node("baz", null)));
```

Rappel : `null` est une valeur spécifique en Java qui représente une référence ne correspondant à aucun objet en mémoire. C'est la valeur par défaut de toute variable de type non primitif.

### Premières fonctions sur les maillons

Créez un nouveau fichier `Node.java` qui implémente les liste chaînées telles que décrites ci-dessus.

### Question 1 : longueur d'une chaîne

Dans la classe `Node`, définissez la fonction récursive (statique) `length_rec`, qui permet de calculer la longueur d'une chaîne. Le profil de la fonction `length_rec` sera le suivant :

```
static int length_rec(Node l) {
    ...
}
```

Vous pourrez tester votre fonction `length_rec` en ajoutant une fonction `main` dans la classe `Node` et affichant la longueur de la chaîne `foobar` donnée en exemple.

Last submission: May 07, 2019 (14:19:31)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java

### Question 2 : longueur d'une chaîne (bis)

Dans les langages impératifs tels que Java, on évite de recourir systématiquement aux fonctions récursives. Comme vu en cours, l'appel de fonction est coûteux en temps et en espace mémoire. La pile ayant une taille limitée, un trop grand nombre d'appels imbriqués se terminent par une erreur de *stack overflow* :

```
Exception in thread "main" java.lang.StackOverflowError
```

Nous allons donc réécrire le calcul de la longueur d'une chaîne en utilisant un parcours itératif :

```
for (Node cur = l; cur != null; cur = cur.next) {
    ...
}
```

Dans la classe `Node`, ajoutez une fonction `length` permettant de calculer la longueur d'une chaîne de manière itérative.

Last submission: May 07, 2019 (14:19:46)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java

### Question 3 : affichage d'une chaîne

Toujours dans la classe `Node`, définissez la fonction :

```
static String printNodes(Node l) {
    ...
}
```

qui renvoie la chaîne désignée par `l`. Sur la chaîne `foobar` définie précédemment, la fonction `printNodes` doit renvoyer la chaîne suivante :

```
"[foo, bar, baz]"
```

Last submission: May 10, 2019 (23:45:07)

0.17 / 2  
Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java

**Question 4 : ajout dans une chaîne**

Définissez la fonction `static void addLast(String s, Node l)` qui ajoute `s` à la fin de la chaîne `l`. Pour cette fonction, nous supposons que `l` ne vaut pas `null`.

Testez votre fonction `addLast` en ajoutant la chaîne `"qux"` à la suite de la chaîne `foobar` donnée en exemple.

Vous remarquerez que l'argument `l` de la fonction `addLast` n'est pas modifié directement, mais que le résultat (attendu) de cette fonction, est de modifier, dans la mémoire, le chaînage entre les différents enregistrements de type `Node`.

Last submission: May 10, 2019 (23:45:41)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java

**Question 5 : chaînes triées**

Écrivez une fonction `Node copy()` qui renvoie une copie d'une chaîne.

Il est possible de comparer deux `String s1` et `s2` pour voir laquelle est avant l'autre dans l'ordre alphabétique. On utilise pour cela la méthode `compareTo`. Plus précisément, `s1.compareTo(s2)` rend :

- une valeur strictement positive si `s2` est avant `s1` dans l'ordre lexicographique,
- une valeur strictement négative si `s1` est avant `s2`,
- la valeur 0 si les deux sont identiques (donc si `s1.equals(s2)` vaut `true`).

Écrivez une fonction `static Node insert(String s, Node l)` qui suppose que la chaîne `l` est triée pour fabriquer la chaîne dans laquelle on aura inséré `s` (on ne devra pas modifier `l`, mais on pourra utiliser `copy`).

Utilisez cette fonction pour écrire une fonction `static Node insertionSort(Node l)` qui fabrique la version triée de `l`. Cette fonction doit être simple et peut avoir une complexité quadratique.

Last submission: May 11, 2019 (10:18:58)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java

**Maillons vs liste chaînée**

Telle quelle, la classe `Node` permet bien d'implémenter la structure de liste chaînée. Néanmoins, comme le montre la question précédente, des problèmes se posent pour l'écriture des opérations sur les listes, notamment l'ajout et la suppression des éléments. Par exemple, dans la question précédente, comment implémenter l'ajout d'un élément dans une chaîne vide ?

Pour écrire facilement les opérations sur les listes, une solution est d'envelopper cette structure de maillons dans un nouvel enregistrement `WordList`, défini de la manière suivante :

```
class WordList {
    Node content;

    WordList() {
        content = null;
    }
}
```

Avec cette implémentation, comment est représentée la liste vide ? Comparez cette représentation avec celle de la chaîne de `Node`.

### Question 6

Dans la classe `WordList`, écrivez la définition d'une variable `foobar`, de type `WordList`, correspondant à la liste `["foo", "bar", "baz"]`.

Last submission: May 11, 2019 (10:44:48)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

## Opérations sur les listes chaînées

Contrairement à la partie précédente, nous implémenterons les opérations sur les listes de mots directement dans la classe `WordList`. Nous abandonnerons donc l'emploi du mot clé `static` et parlerons de *méthode* au lieu de fonction. Une méthode s'utilise sur un objet existant en mémoire et a accès directement aux champs de données.

### Question 7

Ajoutez dans la classe `WordList` une méthode `int length()` calculant la longueur de la liste.

Last submission: May 11, 2019 (10:50:24)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

### Question 8

Vous ajouterez également la méthode `print()`, qui retourne la chaîne de caractère correspondante. Cette méthode respectera le même format que la fonction `printNodes` écrite précédemment.

Last submission: May 11, 2019 (10:53:29)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

### Question 9

Implémentez dans la classe `WordList` les opérations suivantes :

- `void addFirst(String w)`, qui ajoute un mot en tête de la liste;
- `void addLast(String w)`, qui ajoute un mot à la fin de la liste;
- `String removeFirst()`, qui enlève le premier mot de la liste, et le rend en résultat;

0.17 / 2

- `String removeLast()` , qui enlève le dernier mot de la liste, et le rend en résultat.

Dans le cas où la liste est vide, les fonctions `removeFirst` et `removeLast` retourneront la valeur `null` .

Last submission: May 11, 2019 (11:39:18)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

### Question 10

Ajoutez à la classe `WordList` les méthodes `void insert(String s)` et `void insertionSort()` correspondant aux fonctions statiques homonymes de la classe `Node` .

Last submission: May 11, 2019 (11:52:19)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

### Question 11 (cette question est facultative)

Ajoutez à la classe `Node` une fonction `static Node merge(Node l1, Node l2)` qui fusionne deux chaînes triées en une seule chaîne triée.

Utilisez cette fonction pour ajouter à `WordList` une méthode de tri efficace `void mergeSort()` .

Last submission: May 11, 2019 (12:11:10)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

### Question 12

Ajoutez dans la classe `WordList` un constructeur permettant de créer une liste à partir d'un tableau de `String` . Ce constructeur devra avoir le profil suivant :

```
WordList(String[] t) {
    ...
}
```

Ce constructeur devra respecter l'ordre des éléments contenus dans `t` .

Last submission: May 12, 2019 (00:44:36)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

### Question 13

Toujours dans la classe `WordList` , écrivez une méthode `String[] toArray()` qui permet de construire un tableau de chaînes de caractères à partir d'une liste chaînée.

0.17 / 2

Last submission: May 12, 2019 (00:49:41)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Node.java, WordList.java

## Manipulation des préfixes

Les clés de notre table associative sont des séquences de  $n$  mots. Nous allons donc avoir besoin de créer une classe spécifique `Prefix` permettant de manipuler ces séquences :

```
class Prefix {
    String[] t;

    final static String start = "<START>", end = "<END>", par = "<PAR>";
}
```

### Question 14

Ajouter dans la classe `Prefix` un constructeur prenant en argument un entier  $n$  et construisant le préfixe `<START>`, ..., `<START>` de taille  $n$ . Ce préfixe servira à amorcer la construction de la table associative et la génération de texte.

Last submission: May 12, 2019 (01:03:39)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Prefix.java

## Égalités

Considérons le programme suivant :

```
class EqString {
    public static void main(String[] args) {
        String foo1 = new String("foo");
        String foo2 = new String("foo");
        System.out.println(foo1 == foo2);
    }
}
```

Étonnamment (ou pas), le résultat affiché est `false`. En effet, l'opérateur `==` compare les valeurs des variables, ce qui correspond, dans le cas des enregistrements, à comparer les références en mémoire. Or, dans l'exemple `foo1` et `foo2` sont des références à deux enregistrements distincts en mémoire.

Heureusement, la classe `String` fournit une méthode `equals` renvoyant `true` si et seulement si les valeurs des deux enregistrements de type `String` sont égales. Ainsi, l'instruction suivante :

```
System.out.println(foo1.equals(foo2));
```

affiche bien `true` sur la sortie standard.

### Question 15

Dans la classe `Prefix`, ajoutez une méthode statique `eq(Prefix p1, Prefix p2)` qui renvoie `true` si et seulement si les préfixes `p1` et `p2` sont identiques (au sens de `equals`).

0.17 / 2



Last submission: May 12, 2019 (01:14:15)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Prefix.java

**Question 16**

Dans la classe `Prefix`, ajoutez une méthode `Prefix addShift(String w)` qui à partir du préfixe  $w_1, w_2, \dots, w_n$  va construire le préfixe  $w_2, \dots, w_n, w$ .

**Attention**, cette méthode ne doit pas modifier l'objet `this` et construire un nouveau tableau.

Last submission: May 12, 2019 (23:19:59)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Prefix.java

## Tables de hachage

Une table de hachage est une collection de données où chaque entrée est un couple  $(clef, valeur)$ . L'intérêt d'une table de hachage, par rapport à une simple liste, est que si l'on connaît la clef, on peut savoir si la collection contient cette clef et récupérer la valeur associée en temps moyen constant. Par exemple, si l'on déclare un type enregistrement `Eleve` avec comme champs : `nom`, `prénom`, `matricule`, etc. et que l'on range des objets de ce type dans une table de hachage en prenant le matricule comme clef, il sera alors possible d'obtenir la fiche d'un élève à l'aide du matricule. Si l'on ajoute une seconde table pour laquelle la clef est le nom, il sera aussi possible de chercher la fiche d'un élève par son nom.

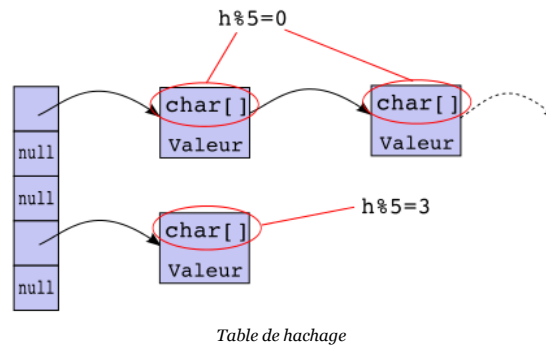
Une seule valeur est associée à une clef et associer une nouvelle valeur à une clef déjà présente dans la table écrase l'ancienne valeur.

Implémenter une table de hachage est trivial dans le cas où les clefs sont des entiers pris dans un intervalle  $[0, n]$  quand  $n$  n'est pas trop grand. Dans ce cas la table de hachage est un tableau et la case  $i$  contient la valeur associée à la clef  $i$  et `null` si aucune valeur ne lui est associée.

Dans le cas général, les tables de hachage se ramènent au cas précédent : on utilise une fonction de hachage qui calcule un entier à partir de la clef. La fonction de hachage doit être telle que si l'on prend deux valeurs égales (au sens sémantique, mais cela peut être deux références différentes), leurs valeurs de hachage doivent aussi être égales. En pratique, pour que la structure soit efficace, on demande beaucoup plus à la fonction de hachage et en particulier, on veut que les valeurs qu'elle donne pour les valeurs rencontrées pendant l'exécution du programme soient les plus éloignées possible les unes des autres.

Une fois cette bonne fonction de hachage `h` trouvée, on procède de la manière suivante : On commence par créer un tableau à  $n$  cases et si l'on veut ajouter l'entrée  $(clef, valeur)$ , on stocke le couple  $(clef, valeur)$  dans la case  $h(clef) \bmod n$  du tableau. Comme il peut y avoir plusieurs valeurs dans la même case, chaque case contient une liste de couples  $(clef, valeur)$ .

La structure est illustrée par le dessin suivant :



En général, les opérations sur la table de hachage sont donc les suivantes :

- **add(key, value)** : considérer la liste présente dans la case  $h(\text{key}) \bmod n$ , chercher si **key** est dans la liste et dans ce cas lui associer la nouvelle valeur; sinon ajouter le couple (**key**, **value**) à la liste.
- **find(key)** : considérer la liste présente dans la case  $h(\text{key}) \bmod n$ , chercher si **key** est dans la liste et dans ce cas retourner la valeur associée; sinon retourner **null**.

On notera que dans ce cas, plus la table est pleine, plus les listes se remplissent et moins le temps de recherche est constant, même avec un bon hachage. Ainsi, si le tableau a pour taille  $n$ , mais que la table contient  $n^2$  entrées, chaque liste a, au mieux,  $n$  éléments et les opérations ci-dessus sont en temps  $n$ . Ainsi, quand la table commence à être trop remplie, il est nécessaire de faire un « rehachage » des entrées : on agrandit le tableau et on répartit à nouveau les données à l'aide de la fonction de hachage.

### Fonction de hachage

La classe `String` de la librairie standard de Java fournit une fonction de hachage sur les chaînes de caractères. Nous allons utiliser celle-ci pour écrire une fonction de hachage pour les préfixes.

#### Question 17 : Fonction de hachage

Ajoutez dans la classe `Prefix` une fonction `public int hashCode()` qui calcule la valeur de hachage pour un préfixe donné. Cette fonction procédera au mélange des valeurs de hachage de chacune des chaînes de caractères qui constituent le préfixe de la manière suivante :

```
h = 37*h + t[i].hashCode();
```

avec `i` parcourant le tableau des mots du préfixe.

Pour travailler avec un tableau de taille  $n$  il faut ramener la valeur de hachage à un entier entre  $0$  et  $n - 1$ . Pour cela, ajoutez également une méthode `int hashCode(int n)` qui fait cela. On pourra utiliser le reste de la division entière de  $a$  et  $b$  qui est noté  $a \% b$  en Java.

**Attention !** La méthode `hashCode()` est susceptible de rendre des valeurs négatives (pour être précis, `%` n'implémente pas le modulo habituel mais le reste de la division entière). Du coup  $a \% b$  est également susceptible d'être négatif. Il faut en tenir compte en écrivant `int hashCode(int n)`.

Last submission: May 13, 2019 (00:29:32)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Prefix.java

### Définition de la table associative

#### Gestion des entrées dans la table

Les entrées dans la table de hachage sont des couples (*clef*, *valeur*) où *clef* est un objet de type `Prefix` et *valeur* un objet de type

WordList .

Créez un nouvel enregistrement `Entry` permettant d'implémenter ces couples (*clef*, *valeur*).

```
class Entry {
    Prefix key;
    WordList value;

    Entry (Prefix key, WordList value) {
        this.key = key;
        this.value = value;
    }
}
```

On a vu que la table de hachage utilise un tableau de taille fixe  $N$  pour stocker les couples (*clef*, *valeur*). Dans le cas général, il peut arriver que 2 entrées dans la table aient le même indice dans la table. Pour implémenter la table de hachage, il faut prendre en compte ces collisions.

Pour cela, nous allons ajouter une classe `EntryList`, implémentant une chaîne d'objets de type `Entry` :

```
class EntryList {
    Entry head;
    EntryList next;

    EntryList(Entry head, EntryList next) {
        this.head = head;
        this.next = next;
    }
}
```

### Question 18 : définition de la table

Créez une nouvelle classe `HMap` contenant la définition de la table de hachage. Cette classe doit contenir 2 attributs :

- `t` : le tableau d'entrées;
- `size` : le nombre d'entrées stockées dans la table.

Vous ajouterez dans cette classe deux constructeurs :

- `HMap(int n)` permettant d'initialiser une table de hachage avec un tableau d'entrées de taille  $n$ ;
- `HMap()` faisant la même chose, mais pour une taille par défaut (par exemple 20).

Last submission: May 13, 2019 (00:43:19)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java

### Question 19 : recherche d'une entrée dans la table

Écrivez dans la classe `HMap` la fonction `WordList find(Prefix key)` qui renvoie la liste de mots associée à un préfixe donné. Si `key` ne peut être trouvé dans la table, `find` renverra la valeur `null`.

Last submission: May 13, 2019 (01:43:58)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java

### Question 20 : ajout d'une entrée dans la table

Ajoutez dans la classe `HMap`, une fonction `void addSimple(Prefix key, String w)` qui va ajouter le mot `w` à la liste de mots associés à `key` dans la table. S'il n'y a pas encore d'entrée associée à `key` alors on crée cette entrée en associant `key` à la liste contenant uniquement `w`.

Remarquez que :

- On a ici une version des tables de hachage spécifique à notre problème. On n'efface jamais une entrée et on peut juste modifier les entrées existantes en ajoutant des mots à la valeur.
- Vous devrez faire attention à bien faire évoluer le champ `size`. Il augmente lorsqu'on crée une nouvelle entrée, mais pas lorsqu'on ajoute un mot à une entrée existante.

Last submission: May 13, 2019 (02:08:42)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java

### Redimensionnement du tableau d'indice

#### Question 21 : rehachage

Comme expliqué précédemment, l'efficacité de la table de hachage dépend du choix de la taille du tableau d'entrées. Il est donc nécessaire d'implémenter une méthode `rehash(int n)` qui fixe le tableau d'indice à la taille `n` et re-répartit l'ensemble des entrées dans la nouvelle table.

Last submission: May 13, 2019 (22:36:23)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java

#### Question 22 : ajout amélioré

Ecrivez une méthode `add` similaire à `addSimple` mais améliorée pour provoquer un rehachage automatique quand la table a plus d'entrées que trois-quarts de la taille du tableau. On choisira dans ce cas de doubler la taille du tableau d'entrées.

Last submission: May 13, 2019 (22:43:47)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java

## Madame Bovary

### Lecture du texte

Le texte de Madame Bovary est donné sous la forme de 35 fichiers (un par chapitre de `01.txt` à `35.txt`) dans l'archive [bovary.zip](#) ([resources/bovary/bovary.zip](#)) à désarchiver ainsi :

0.17 / 2

```
% unzip bovary.zip
```

Vous disposez maintenant d'un sous-répertoire `bovary` qui contient les 35 chapitres. Le texte est soumis à [cette licence](http://abu.cnam.fr/cgi-bin/donner_licence) ([http://abu.cnam.fr/cgi-bin/donner\\_licence](http://abu.cnam.fr/cgi-bin/donner_licence)).

Vous devez également récupérer le fichier `WordReader.java` ([resources/src/WordReader.java](#)) contenant la classe `WordReader` permettant de lire les chapitres.

Cette classe comprend entre autres, un constructeur `WordReader (String filename)` qui prend un nom de fichier en argument; ainsi qu'une méthode `main` d'essai :

```
% java WordReader bovary/01.txt
[Nous]
[étions]
[à]
[l'Etude,]
[quand]
[le]
[Proviseur]
[entra]
[suivi]
[d'un]
...
[<PAR>]
...
```

Outre les mots du premier chapitre écrit par Flaubert, vous voyez apparaître des mots `<PAR>` qui indiquent simplement les paragraphes. Ces mots `<PAR>` sont à traiter comme des mots ordinaires.

La classe `WordReader` définit une méthode `String read()` qui lit un mot dans le chapitre en cours. Si le chapitre a été entièrement lu, la méthode `read()` renvoie `null`.

## Construction de la table associative

### Question 23 : buildTable

Créez une nouvelle classe `Bovary`, dans laquelle vous définirez la fonction `static HMap buildTable(String[] files, int n)` qui construit la table associative à partir des fichiers dont le nom se trouve dans le tableau `files`, avec des préfixes de taille `n`.

Last submission: May 14, 2019 (00:10:05)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Bovary.java, Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java

## Génération de texte

### Question 24 : generate

Ajoutez dans la classe `Bovary` une fonction `void generate(HMap t, int n)` qui affiche sur la sortie standard un pseudo texte à partir de la table `t`, en utilisant des préfixes de taille `n`.

Sans chercher à produire une présentation parfaite, l'affichage devra être lisible (espaces entre les mots, sauts de lignes avant et après le mot `<PAR>`).

Last submission: May 14, 2019 (00:38:42)

0.17 / 2  
Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Bovary.java, Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java

## Le 36<sup>e</sup> chapitre de Mme Bovary

### Question 25

Ajoutez une fonction `main` qui combine les fonctions `buildTable` et `generate` pour produire sur la sortie standard le 36<sup>e</sup> chapitre de *Mme Bovary*.

Last submission: May 14, 2019 (01:01:55)

Choose

Choose one or more files...

Submit

☐ Reuse files from previous submissions ?

Expected files: Bovary.java, Entry.java, EntryList.java, HMap.java, Node.java, Prefix.java, WordList.java