


TP n° 7


Tables de hachage

L'objectif de cette séance de travaux pratiques est de se familiariser avec l'utilisation d'une table de hachage, en l'occurrence la classe `HashMap` ([documentation](#) en ligne).

Exercice 1 – Traduction


Dans cet exercice, on utilise un dictionnaire pour traduire un texte mot à mot.


 Dans un nouveau paquetage `traduction`, télécharger la classe `Traduction_EN_FR` qui contient un dictionnaire (implémenté par une table de hachage) pré-rempli et une phrase en anglais à traduire.

 Écrire une méthode `traduire` qui prend en paramètre une chaîne de caractères (un texte en anglais) et qui retourne une chaîne de caractère (texte traduit en français). Pour construire la chaîne renvoyée par la méthode, on pourra découper¹ la chaîne originale et utiliser le dictionnaire pour traduire chaque mot. Lorsqu'un mot n'existe pas dans le dictionnaire, on conservera le mot original anglais qu'on mettra entre parenthèses.

Exercice 2 – Analyse fréquentielle

Dans cet exercice, nous utilisons une table de hachage pour compter la fréquence des lettres dans un texte et comprendre comment la langue du texte peut être détectée automatiquement.


 Dans un nouveau paquetage `analyse_freq`, télécharger la classe `AnalyseFrequentielle` qui contient trois textes, en français, anglais et tchèque.


 Écrire la fonction `analyser` qui prend en paramètre un texte sous forme de chaîne de caractères et retourne une liste ordonnée des 6 caractères les plus utilisés dans le texte fourni en paramètre. L'algorithme utilise une table de hachage dans laquelle les clés sont les caractères et les valeurs le nombre de fois où chaque caractère apparaît. Il fonctionne en deux phases :


1. Parcourir l'intégralité du texte caractère par caractère et mettre à jour les données dans la table de hachage, puis

1. On s'intéressera à la méthode `split()` de la classe `String`

2. (répéter 6 fois) Parcourir les valeurs de la table de hachage à la recherche de la valeur la plus élevée. À la fin de chacun des 6 parcours, ajouter dans une liste le caractère correspondant à la valeur la plus élevée et supprimer de la table le couple correspondant afin de chercher le caractère suivant.


 Appliquer la fonction aux trois textes et comparer les résultats.


 Pour améliorer les capacités de détection automatique de la langue, on désire maintenant compter les digrammes (un digramme est un couple de 2 lettres de l'alphabet) dans le texte. Dans une nouvelle fonction `analyserDigrammes`, adapter l'algorithme ci-dessus pour retourner la liste des 6 digrammes les plus utilisés dans un texte passé en paramètre.


 Appliquer cette nouvelle fonction et comparer les résultats. Comment pourrait-on utiliser cet algorithme pour détecter de manière automatique la langue d'un texte? Quel type de cryptage de texte cet algorithme est-il en mesure de résoudre?


Exercice 3 – Index d'un document


Dans cet exercice, nous utilisons une table de hachage pour mettre en œuvre la construction automatique de l'index d'un document découpé en sections. L'index d'un document consiste en la liste d'un ensemble d'entrées, généralement des mots apparaissant dans le document, avec les numéros des pages où ils apparaissent (voir une illustration d'index en fin de document).


 Dans un nouveau paquetage `index`, télécharger la classe `Index.java`. Assurez-vous également de copier à la racine de votre projet l'ensemble des sections (fichiers textes à télécharger sous forme d'archive zip). Lancer le projet et vérifier l'affichage dans la console du contenu du fichier `abstract.txt`.

 Créer la classe `Entrée` permettant de représenter une entrée de l'index. Une entrée est composée d'un nom de section (chaîne de caractères) et d'un numéro de ligne (entier).


 Modifier la classe `Index` afin qu'elle dérive désormais de la classe Java `HashTable`. La table de hachage doit être paramétrée de la manière suivante : la clé est une chaîne de caractères (i.e. le mot à indexer) et la valeur est une liste d'entrées (i.e. classe `Entrée`, représentant une section et une ligne auxquelles le mot apparaît).

 Écrire dans la classe `Index` la méthode `scan` qui prend en paramètre une section et un mot et qui permet de "scanner" le fichier correspondant à la section et ajouter une entrée dans l'index à chaque fois qu'une ligne du fichier contient le mot en paramètre.

 Dans le programme principal, écrire le code qui permet de créer un index puis de le remplir en 'scannant' dans toutes les sections de l'article tous les mots suivants : "Cortexionist", "neural network", "knowledge", "pattern", "simulation", "intelligence", "memory", "behaviour", "learning".

 À la fin du programme principal, afficher la table de hachage dans la console pour vérifier que toutes les entrées ont correctement été ajoutées. Il sera sans doute nécessaire de modifier les méthodes `toString` des classes `Index` et `Entrée`. Pour comparer, on pourra se référer à l'annexe page suivante.

On remarque que plusieurs lignes de la même section apparaissent dans notre index comme plusieurs entrées différentes.

 Apporter les modifications nécessaires au projet de manière à ce que chaque section ne soit noté qu'une seule fois par mot de l'index, quel que soit le nombre de fois où le mot apparaît dans la section. S'il apparaît plusieurs fois, alors l'index affiche une liste de pages pour le couple (mot, section).

Index de l'article scientifique

L'index ci-dessous illustre les sections et numéros de page dans lesquels apparaissent les mots de l'index.

behaviour

- abstract 1,7
- intro 9,10
- related 1,4,8
- contribution 4,5,8
- foundations 15
- model 3,4,8,14
- experiment 9
- conclusion 5,6

memory

- abstract 6
- intro 8
- related 4,8,10
- contribution 3,7,8
- foundations 16
- model 16,19,30,33,34
- discussion 1,2,3,7,8,9
- conclusion 1,5

simulation

- related 1
- foundations 5,6
- experiment 10

learning

- foundations 11
- model 8,32,33,35
- discussion 2

pattern

- contribution 4

- foundations 16

- model 24,27,29,30

- experiment 11

intelligence

- abstract 4

- intro 2,3,4,6,8,10

- related 9

- contribution 8

- foundations 9,15,18

- model 4

neural network

- foundations 2

- model 15,34

- conclusion 10

knowledge

- abstract 3

- related 4,5,8

- contribution 4

- model 16,17,20,22–24,26–29,32–35

- conclusion 3,5,6

Cortexionist

- abstract 7

- foundations 1

- model 1

- discussion 1,9

- conclusion 10