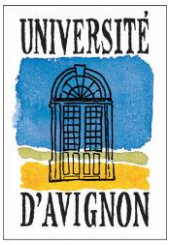

UNIVERSITÉ D'AVIGNON
ET DES PAYS DE VAUCLUSE

M2 ILSEN – 2017/18

UE Ingénierie du document et de l'information

UCE Indexation & recherche

Vincent Labatut

TP 1 | définition d'un fichier inverse

L'objectif de cette série de TP est d'implémenter des fonctionnalités de base nécessaires à l'indexation. Nous allons donc nous attacher à programmer des versions simples de ces structures et algorithmes, afin de bien comprendre comment ils fonctionnent.

1 Organisation

Outils. Nous utiliserons le langage Java et l'IDE Eclipse. Le fichier Objets Java pour la RI récapitule les différentes classes Java permettant de représenter des collections d'objets. Il est recommandé d'y jeter un œil avant de commencer le TP.

Évaluation. L'évaluation des TP est réalisée lors d'une séance d'examen qui a lieu en fin de semestre. Cette séance s'organise comme un TP normal, lors duquel on implémente une fonctionnalité vue en cours (mais pas déjà traitée lors des TP précédents).

Chaque étudiant doit se présenter au TP noté avec le code source correspondant aux exercices des TP précédents. En effet, le TP noté consiste à partir de l'une des versions de l'outil développée lors des séances non-évaluées, et à l'enrichir. Un étudiant qui se présenterait sans cette ressource (ou même avec une version incomplète) n'aurait vraisemblablement pas le temps matériel de finir le TP noté.

Corpus. Le TP est fourni avec un corpus (collection de textes à traiter) prenant deux formes distinctes : `corpus` qui est la version complète (2871 textes), et `test` qui est un sous-ensemble (200 textes). Dans un premier temps, il est conseillé de déboguer votre code source sur `test`, pour des raisons de rapidité. Lorsque votre programme semble fonctionner correctement, alors vous pouvez l'appliquer à `corpus`.

Pour information, le corpus est composé d'articles de la version française de Wikipedia. Ceux-ci ont été obtenus en crawlant ce site à partir de l'article Recherche d'information, de façon interne (i.e. en considérant seulement les liens vers d'autres pages françaises de Wikipedia) et en limitant la distance parcourue à deux hops. Les articles sont nettoyés de manière à ne contenir que les caractères suivants : lettres, chiffres, espace, `'\n'`, et la ponctuation `'():,-!?."';&@%+`.

Attention : les consignes suivantes sont valides pour toute la série de TP :

- Il est obligatoire de respecter les contraintes d'implémentations spécifiées dans les sujets : identificateurs (noms de méthodes, paramètres, variables, etc.), types de paramètres et de retour, etc.
- Vous ne devez pas créer d'autres classes et méthodes que celles demandées.
- Vos classes et méthodes de test doivent forcément traiter *au moins* les exemples proposés dans les sujets (mais il est préférable de les étendre).

Chaque sujet est prévu pour une durée d'1h30. Si vous ne parvenez pas à les terminer dans le temps imparti, ils sont à finir chez vous avant la séance suivante.

2 Préparation

L'archive disponible sur e-uapv contient deux projets Eclipse. Le projet `Common` contient les deux versions du corpus, placés respectivement dans les dossiers `corpus` et `test`. Le projet `V1` contient le code source. Il s'agit du squelette de notre outil d'indexation et de recherche. Il contient des classes dans lesquelles les champs sont manquants et les méthodes sont vides. Il vous faut donc les compléter, en suivant les instructions des sujets. Notez que 3 versions du logiciel, appelées `V1`, `V2` et `V3` seront successivement développées et comparées au cours des TP.

Exercice 1

Vous devez d'abord importer les projets dans Eclipse. Pour cela :

- Dans Eclipse, allez dans *File* > *Import* > *Existing Projects into Workspace.*
- Dans *Select archive file*, sélectionnez l'archive téléchargée depuis e-uapv.
- Dans *Projects*, sélectionnez les deux projets disponibles.
- Vérifiez que l'option *Copy projects into workspace* est bien cochée.
- Cliquez sur *Finish* pour démarrer l'importation.

Les deux projets vont alors apparaître dans votre espace de travail.

Notez que sous Eclipse, les chemins *relatifs* sont interprétés par rapport au projet en cours d'exécution. Si vous voulez accéder au corpus situé dans le dossier `corpus` du projet `Common`, à partir du code source situé dans le projet `V1`, vous devez donc utiliser le chemin relatif suivant : `../Common/corpus`.

Attention : pour des raisons de portabilité, il est interdit d'utiliser des chemins *absolus* dans votre code source.

3 Tokénisation basique

Le but du TP n'est pas la tokénisation, néanmoins nous avons besoin de réaliser ce traitement, donc dans un premier temps nous allons en implémenter une version très simple.

Exercice 2

Dans le package `indexation.content`, la classe `Token` représente un token basique, à savoir : une chaîne de caractères `type` correspondant au type associé au token et un entier `docId` représentant le docID du document contenant le token.

Créez les deux champs nécessaires et complétez le constructeur public `Token(String type, int docId)` de manière à les initialiser grâce aux paramètres reçus.

Exercice 3

La classe `Token` doit implémenter l'interface `Comparable<Token>`. Cela signifie que vous devez compléter la méthode publique `int compareTo(Token token)` qui renvoie un entier négatif, nul ou positif si le token considéré (i.e. l'objet `this`) est respectivement plus petit, égal ou plus grand que le token `token` passé en paramètre. On comparera les tokens en utilisant le type comme critère primaire, et en cas d'égalité, le docID comme critère secondaire.

Exercice 4

Dans `Token`, vous devez aussi surcharger plusieurs méthodes publiques héritées de la classe `Object` :

- `boolean equals(Object o)` : renvoie `true` si l'objet passé en paramètre est le même que l'objet considéré (utilisez `compareTo`) ;

- `String toString()` : renvoie une chaîne de caractères représentant le token considéré.

Testez vos méthodes directement en complétant la méthode `main` de la classe.

exemple : pour un token `maison` contenu dans le document 5, il faudra afficher :

```
(maison, 5)
```

Exercice 5

Dans le package `indexation.processing`, la classe `Tokenizer` est chargée d'effectuer la tokénisation. Dans cette classe, complétez la méthode publique `List<String> tokenizeString(String string)` qui tokénise la chaîne de caractères `string` passée en paramètre, et renvoie une liste correspondant à cette décomposition.

On considérera toute suite de caractères alphanumériques comme un token. Autrement dit, tout ce qui n'est ni un chiffre ni une lettre sera considéré comme un séparateur de mots. Une chaîne de caractères vide (i.e. `""`) ne sera pas considérée comme un token et devra donc être ignorée.

Comme précédemment, testez votre méthode via le `main` de la classe. Appliquez `tokenizeString` à une chaîne de caractères quelconque et affichez le résultat renvoyé.

Remarque : utilisez la méthode `String.split` avec une expression régulière appropriée. Pour information, en Regex Java, `\pL` et `\pN` permettent respectivement de faire référence à n'importe quelle lettre et à n'importe quel chiffre. Vous avez également besoin de la notion de classe Regex de caractères, dont un certain nombre sont prédéfinies en Java.

Exercice 6

Toujours dans la classe `Tokenizer`, complétez la méthode privée `void tokenizeDocument(File document, int docId, List<Token> tokens)`, qui est chargée d'effectuer la tokénisation du document passé en paramètre. Les tokens obtenus doivent être renvoyés en complétant la liste `tokens` passée en paramètre. Cette méthode doit bien sûr utiliser `tokenizeString`. Comme précédemment, testez votre méthode via le `main` de la classe.

exemple : l'affichage des tokens obtenus pour le tout premier fichier de `corpus` (`001f1107-8e72-4250-8b83-ef02eeb4d4a4.txt`) doit avoir la forme :

```
[(En, 0),
(mathématiques, 0),
(le, 0),
(théorème, 0),
(de, 0),
(la, 0),
(dimension, 0),
(pour, 0),
(les, 0),
(espaces, 0),
(vectoriels, 0),
...]
```

Rappel : pour ouvrir un fichier texte unicode représenté par une variable `file`, en Java :

```
FileInputStream fis = new FileInputStream(file);
InputStreamReader isr = new InputStreamReader(fis,"UTF-8");
Scanner scanner = new Scanner(isr);
```

Pour en lire le contenu ligne par ligne :

```
while(scanner.hasNextLine())
{  String line = scanner.nextLine();
   // traitement ici
}
```

N'oubliez pas de refermer le flux à la fin :

```
scanner.close();
```

Exercice 7

Pour en finir avec la classe `Tokenizer`, complétez la méthode publique `int tokenizeCorpus (String folder, List<Token> tokens)`, qui s'applique au corpus entier, et qui utilise la méthode privée `tokenizeDocument`. Elle prend en paramètre

le chemin d'un dossier (contenant les fichiers texte qui constituent un corpus) et une liste vide de tokens. Elle effectue la segmentation des fichiers texte constituant le corpus, et place les tokens résultants dans cette liste. Elle renvoie un entier correspondant au nombre de documents dans le corpus.

Testez votre méthode via le `main`, en appliquant `tokenizeCorpus` au corpus de taille réduite contenu dans le dossier `test` du projet `Common`. Vous devez obtenir une liste de 631 928 tokens.

Attention : les noms des fichiers du corpus ne sont pas des entiers, donc vous devez vous-même générer les docIds de manière à avoir une séquence d'entiers consécutifs (en partant de zéro).

Remarque : pour obtenir un tableau des noms des fichiers contenus dans un dossier, chacun étant représenté par un objet de classe `String`, utilisez `File.list()`. Notez que Java ne garantit pas d'ordre particulier pour ces fichiers : en effet, cet ordre peut varier en fonction du système d'exploitation/de fichiers. Il est donc nécessaire d'appliquer `Arrays.sort()` à ce tableau, afin de trier les noms. Si vous ne faites pas cette opération, vous ne pourrez pas comparer vos propres résultats à ceux donnés en exemples dans les sujets de TP.

De plus, quand vous construisez un chemin de fichier, utilisez la constante `File.separator` à la place du littéral `'/'` ou `'\'`, afin de produire un code source indépendant du système d'exploitation.

4 Normalisation basique

Comme pour la tokénisation, nous allons réaliser une normalisation linguistique très simple, avant d'approfondir ce point plus tard.

Exercice 8

Allez dans le package `processing` et ouvrez la classe `Normalizer`. Dans cette classe, complétez la méthode publique `String normalizeType(String type)`, qui prend en paramètre une chaîne de caractères `type` représentant le type associé à un token, et renvoie le terme résultant de sa normalisation, sous la forme d'une chaîne de caractères.

La normalisation doit être effectuée simplement en transformant toute lettre majuscule en lettre minuscule, et en supprimant tous les signes diacritiques. Si le résultat obtenu n'est pas un terme (par exemple la chaîne vide `""`), la méthode doit renvoyer `null`.

Remarque : la version présente de cette méthode n'est pas supposée renvoyer `null`, par contre les versions ultérieures le feront, c'est pourquoi on inclut ce test dès maintenant.

Testez votre méthode directement en complétant la méthode `main` de la classe, comme pour la tokénisation.

Remarque : `String.toLowerCase` permet de passer une chaîne de caractères en minuscules. Pour enlever les signes diacritiques d'une chaine `string`, on peut utiliser l'instruction suivante :

```
string = java.text.Normalizer.normalize(string, Form.NFD)
            .replaceAll("\\p{InCombiningDiacriticalMarks}+", "");
```

Exercice 9

Toujours dans la classe `Normalizer`, complétez la méthode `void normalizeTokens(List<Token> tokens)`, qui s'applique à une liste de tokens et les normalise individuellement. Pour cela, elle doit nécessairement utiliser la méthode `normalizeType`, qui s'applique, elle, à une simple chaîne de caractères. N'oubliez pas que `normalizeType` est susceptible de renvoyer `null`, donc vous devez traiter ce cas dans `normalizeToken`. Testez votre méthode via le `main`, comme précédemment.

exemple : sur le tout premier fichier de `corpus`, vous devez obtenir un affichage du type :

```
[(en, 0),
(mathematiques, 0),
(le, 0),
(theoreme, 0),
(de, 0),
(la, 0),
(dimension, 0),
(pour, 0),
(les, 0),
(espaces, 0),
(vectoriels, 0),
...]
```

Attention : vous devez utiliser la classe `Iterator` pour parcourir la liste tout en ayant la possibilité d'en supprimer des éléments.

5 Fichier inverse simple

On veut utiliser un fichier inverse comme index. Comme on l'a vu en cours, il s'agit d'une séquence ordonnée de termes, chacun étant lui-même associé à une liste ordonnée de postings.

Exercice 10

Dans le package `content`, la classe `Posting` représente un posting. Dans un premier temps, un posting correspondra simplement à un entier (mais cela évoluera dans les TP prochains). Définissez donc le champ entier appelé `docId` et représentant cette information. Puis, complétez le constructeur public `Posting(int docId)` chargé d'initialiser ce champ avec le paramètre reçu.

Comme `Token`, la classe `Posting` doit implémenter l'interface `Comparable<Posting>`. Vous devez donc là aussi écrire la méthode publique `int compareTo(Posting posting)`. Les deux postings ne seront comparés qu'à travers leur champ `docId`.

De plus, encore à l'instar de `Token`, vous devez également surcharger dans `Posting` les méthodes publiques `boolean equals(Object o)` et `String toString()` héritées de `Object`. Pour `toString`, on se contentera d'afficher le champ `docId`. Pour `equals`, on utilisera simplement `compareTo`, comme on l'avait déjà fait dans `Token`.

Testez vos méthodes directement en complétant la méthode `main` de la classe.

Exercice 11

Dans le package `content`, la classe `IndexEntry` représente une entrée de l'index, à savoir : une chaîne de caractères `term` correspondant au terme concerné et une liste de `Postings` appelée `postings` et représentant les postings associés à ce terme. Créez les deux champs nécessaires.

Complétez le constructeur `IndexEntry(String term)`, qui crée une instance contenant le terme `term` et une liste de postings *vide*. La liste doit être accessible pour pouvoir être complétée par la suite, avec les différents ids des documents contenant le terme.

Comme pour `Token` et `Posting`, `IndexEntry` doit implémenter l'interface `Comparable<IndexEntry>` et vous devez surcharger les méthodes `toString` et `equals`. Pour comparer deux entrées de l'index, on utilisera seulement le champ `term` (et on ignorera donc la liste de postings).

Testez vos méthodes directement en complétant la méthode `main` de la classe.

exemple : pour une entrée contenant le terme *bateau* et les postings (1,5,99,694,702), on affiche :

```
<bateau ( 1 5 99 694 702 )>
```

Exercice 12

Dans le package `indexation`, la classe `Index` est chargée de représenter le fichier inverse. Pour stocker le lexique et les listes de postings associées aux termes, on utilisera un champ public `data` correspondant à un tableau d'entrées `IndexEntry[]`. Deux autres

champs `normalizer` et `tokenizer` seront utilisés pour représenter les objets utilisés lors de l'indexation. Enfin, un dernier champ entier `docNbr` permettra de stocker le nombre de documents présents dans le corpus traité. Créez les 4 champs correspondants.

La classe a besoin d'un constructeur public qui prend en paramètre un entier `size` représentant la taille (en nombre d'entrées) de l'index à créer. Cette valeur doit être utilisée pour initialiser le champ `data` lors de la construction de l'objet.

Exercice 13

Dans `Index`, complétez la méthode publique `void print()` qui affiche le contenu de l'index dans la console Vous devez bien sûr exploiter la méthode `IndexEntry.toString()`.

exemple : l'affichage d'un index devrait prendre la forme suivante

```
<barque ( 1 3 5 6 1023 1024 1268 1989 )>
<bateau ( 1 5 99 694 702 )>
<coquille (199 723 )>
...
```

Remarque : nous n'avons pas encore implémenté les méthodes nécessaires à l'initialisation de l'index (c'est l'objet du TP suivant), donc on ne peut pas les utiliser pour tester cette méthode. Pour faire vos tests, devez donc initialiser *manuellement* l'index, à l'aide de quelques entrées bidon.

Exercice 14

Dans la classe `Index`, complétez la méthode publique `IndexEntry getEntry(String term)`, qui renvoie l'entrée de l'index associée au terme `term` passé en paramètre. Vous devez utiliser le principe de la recherche dichotomique. On supposera pour cela que les entrées de l'index sont triées dans l'ordre alphanumérique.

Comme dans l'exercice précédent, on ne peut pas tester cette fonction avec un index réel : il faut en initialiser un (tout petit) manuellement.

Remarque : pour la recherche dichotomique, utilisez la méthode `Arrays.binarySearch`, qui prend en paramètres un tableau trié et un objet susceptible de lui appartenir, et qui renvoie la position de l'objet dans le tableau, ou une valeur négative si l'objet n'apparaît pas dans le tableau.