SINF1121 – Algorithmique et structures de données

Mission 4

Maps et Dictionnaires : tables de hachage et autres implémentations

1 Contexte général de la mission

Durant cette mission, vous serez en charge d'implémenter un détecteur de plagiat. Le principe de la mission est simple : on vous donne un dossier contenant un ensemble de fichiers textes (`corpus`), ainsi qu'un fichier texte à part (`document`). Votre tâche est de vérifier, pour un certain entier `w`, si `document` contient des phrases d'au moins `w` caractères qui proviendraient telles quelles d'un des fichiers présents dans `corpus`. Le fil conducteur de cette mission sera bien sûr l'ADT `Map`.

2 Objectifs poursuivis

À l'issue de cette mission chaque étudiant sera capable :

— de ***décrire*** avec exactitude et précision les concepts présents dans le chapitre du livre de référence, qui traite de ***maps*** et ***dictionnaires*** (***multimaps*** dans *DSAJ-6*),

— de ***mettre en oeuvre*** des algorithmes basés sur les types abstraits `Map`, `Map Ordonné`, et `Dictionnaire`,

— de ***résoudre*** des problèmes simples sur la mémorisation d'informations et leur accès dans des dictionnaires,

— d'***évaluer*** et ***mettre en oeuvre*** des représentations classiques de dictionnaires.

3 Prérequis

Les conditions à remplir pour pouvoir aborder cette mission sont :

— avoir rempli avec succès les missions précédentes,

— se débrouiller en anglais technique et scientifique (lecture).

4 Ressources

— Livre : *Algorithms-4* : chapitre 3.4, 3.5, 5.3 (Rabin-Karp fingerprint search).

— Document : *Spécifications en Java, Préconditions et postconditions : pourquoi ? comment ?*, P. Dupont.
Sur le site Moodle, suivre `Documents et liens/pdf/specif.pdf`

— Document : *Complexité calculatoire*, P. Dupont
Sur le site Moodle, suivre `Documents et liens/pdf/complex.pdf`

5 Calendrier

— lundi 2 novembre : démarrage de la mission
— vendredi 6 novembre, avant 18h00 : envoi réponses questions (groupe) + tests Inginious (individuel)
— lundi 9 novembre : séance intermédiaire
— vendredi 13 novembre, avant 18h00 : remise des produits (groupe)
— lundi 16 novembre : séance de bilan

6 Questions

1. Citez au moins quatre implémentations différentes d'un dictionnaire (table de symboles). Précisez, dans chaque cas, quelles sont les propriétés principales de ces implémentations. Dans quel(s) cas s'avèrent-elles intéressantes ? Quelles sont les complexités calculatoires de leurs principales méthodes ?
2. Souvenez vous de la question suivante proposée en bilan sur la mission sur les tris : Étant donné un ensemble S de taille n, et un nombre x. Décrivez un algorithme efficace utilisant une HashTable pour trouver s'il existe une paire (a, b) avec $a \in S, b \in S$ telle que $a + b = x$. Quelle est la complexité de votre algorithme ? Est-elle meilleure que votre solution qui utilisait un tri ?
3. Démontrez que $(a + b)\%M$ est équivalent à $((a\%M) + b)\%M$. En quoi cette propriété peut être utile pour construire une fonction de hachage sur les String. Expliquez comment Java calcule une fonction de hachage sur les String ? Quelle est la complexité pour calculer 1 fois et N fois le hashcode d'un String.
4. Expliquez pourquoi la méthode `hash()` p461 de *Algorithms-4* retourne

   ```
   (x.hashCode() & 0x7FFFFFFF) % M
   ```

   et pas simplement

   ```
   x.hashCode() % M
   ```

   ?

   Quel nombre représente `0x7FFFFFFF` ? Quelle est sa représentation binaire ? Montrer l'exemple l'impact au niveau binaire sur un exemple où `x.hashCode()` retourne un nombre négatif. Hint : utilisez `Integer.toBinaryString(int)` pour vérifier votre réponse.
5. Java fournit la classe **`java.util.Hashtable`** comme implémentation de l'interface **`java.util.Map`**. Pouvez-vous déterminer précisément de quelle variante de table de hachage il s'agit ? Java fournit-il d'autres implémentations de l'interface **`Map`** ? Faites un diagramme qui représente les interfaces et les classes qui se rapportent à **`Map`** et précisez ce qui, dans chaque cas, les caractérise. Qu'est-ce qui peut servir de clef pour une **`Hashtable`** en Java ? Soyez précis.
6. Qu'entend-on par la notion de *collision* dans une table de hachage ? Les collisions ont-elles une influence sur la complexité des opérations ? Si oui, quelle(s) opération(s) avec quelle(s) complexité(s), sinon précisez pourquoi.

7. Est-ce que le facteur de charge d'une table de Hashage. Est-ce que le contrôle du facteur de charge est nécessaire/optionnel pour le bon fonctionnement d'une table de Hashage avec Linear Probing ou Separate Chaining ? Quelle est la stratégie utilisé par `java.util.Hashtable` pour contrôler le facteur de charge ? En quoi est-elle différente de celle proposée dans `LinearProbinHashST` ? Quel est le lien entre le facteur de charge et collision ?
8. Imaginez une nouvelle méthode `iterator()` qui retourne un itérateur sur les clefs de `LinearProbingHashST`. Votre itérateur ne devrait pas accepter de modification de la table de hashage alors qu'il est utilisée :
une `ConcurrentModificationException()` doit être lancée si c'est le cas. Que suggérez vous pour ce faire ? Hint : Inspirez vous de la stratégie de `java.util.Hashtable`.
9. Décrivez l'implémentation de la méthode `put(key)` dans une table de hachage qui utilise la technique du *linear probing* pour gérer les collisions qui utiliserait un marqueur spécial pour représenter les entrées supprimées à l'aide de la méthode `delete(key)`. En d'autres termes la méthode `delete(key)` au lieu de réarranger le contenu de la table de hachage de telle sorte qu'elle soit comme si l'entrée supprimée n'avait jamais été insérée, va simplement marquer l'entrée avec le marqueur spécial. Quelle est l'avantage ou l'inconvénient de cette approche par rapport à celle de `LinearProbingHashST` dans *Algorithms-4* ?
10. **[Question liée spécifiquement au problème posé]** Imaginez une fonction de hachage pour un string s telle que connaître sa valeur pour le sous string $s[i, ..., i+n-1]$ permettrait de calculer la fonction de hachage du string $s[i+1, ..., i+n]$ en temps constant (de manière incrémentale).
11. **[Question liée spécifiquement au problème posé]** Expliquez comment rechercher un sous string de taille M dans un long string de taille N en $O(N)$ à l'aide d'une fonction de hachage incrémentale. Comment feriez-vous pour si vous avez k strings de taille $M_1, \ldots, M_k$ à rechercher dans le long string M ? Quelle serait la complexité de votre méthode ?

Les réponses aux questions et les parties d'implémentation individuelles doivent être soumises sur INGInious ***avant*** la séance ***intermédiaire***[a]. Cela suppose une étude individuelle et une mise en commun en groupe (sans tuteur) préalablement à cette séance. Un document (au format PDF) reprenant les réponses aux questions devra être soumis sur INGInious ***au plus tard*** pour le vendredi 6 novembre à ***18h00***. Les réponses seront discutées en groupe avec le tuteur durant la séance intermédiaire. Ces réponses ne doivent pas explicitement faire partie des produits remis en fin de mission. Néanmoins, si certains éléments de réponse sont essentiels à la justification des choix d'implémentation et à l'analyse des résultats du programme, ils seront brièvement rappelés dans le *rapport de programme*.

a. à l'exception, le cas échéant, de question(s) spécifiquement liée(s) au problème traité.

7 Problème

Le but de cette mission est donc de créer une classe `Plagiarism` dont le constructeur prend en arguments deux éléments :

— Le chemin vers le `corpus` de fichiers textes,
— Le nombre `w` de caractères à partir duquel on considère qu'une phrase est plagiée.

Cette classe aura une méthode publique `detect(String doc)` qui prend en argument le document à vérifier, et qui retourne un `Set<Entry<String, Integer>>` contenant, pour chaque phrase plagiée, le nom du document plagié (dans le corpus) ainsi que la position (dans le document du corpus) du premier caractère de la phrase plagiée (en commençant à 0, comme `charAt`).

Exemple concret, avec `w` égal à 10 :

`document.txt` "Nabuchodonosor, roi de Babylone. "
`corpus/one.txt` "Mais qui était le roi de Babylone ?"
`corpus/two.txt` "Il s'appelait Nabuchodonosor. "
`output` [one.txt=23, two.txt=14, one.txt=22, two.txt=15, one.txt=21, one.txt=20, one.txt=19, one.txt=18, one.txt=17, two.txt=18, two.txt=17, two.txt=16]

Pour ce faire, vous utiliserez l'algorithme de Rabin-Karp afin de résoudre le problème du *Substring Pattern Matching* en $O(n + k)$ *expected* au lieu de $O(n \cdot k)$. Vous devrez donc implémenter votre propre HashMap afin d'avoir une fonction de hashage qui tourne en $O(1)$ *expected* (sauf pour le premier substring qui sera hashé en $O(k)$).

Comme d'habitude, vous devez pouvoir justifier vos choix d'implémentation (surtout pour la `Map`), ***pourquoi*** ils sont pertinents par rapport au problème à traiter. Parmi les critères de pertinence, vous veillerez à argumenter en fonction des complexités calculatoires associées aux structures de données mises en oeuvre et à la généricité de vos programmes.

Vous veillerez à commenter et à documenter clairement votre code pour expliciter dans tous les cas quels sont les auteurs de quelle partie de code.

Indications pour la bonne réussite de la mission

- Les réponses aux questions posées ne se trouvent pas toutes dans le chapitre du livre de référence concerné par cette mission. N'oubliez pas de consulter l'index de cet ouvrage ou toute autre ***ressource complémentaire*** mentionnée dans ce document ou sur le site WEB du cours. Veillez toujours à ***citer*** précisément ***vos sources*** lorsque vous répondez aux questions.

- Veillez à vous concentrer d'abord sur les questions avant de passer au problème à résoudre. Veillez à y répondre de manière précise et concise. Quand vous attaquez le problème, gardez à l'esprit les concepts abordés dans les questions.

- Pendant le travail d'analyse du problème et de conception d'une solution, l'écriture d'un ***diagramme de classes*** est très utile pour
  — vérifier l'exactitude, la cohérence et la complétude du découpage en plusieurs classes,
  — préciser les liens entre les classes concrètes, les classes abstraites et les interfaces,
  — concevoir de façon modulaire la solution proposée afin de pouvoir ***réutiliser*** certaines parties de code dans différents contextes (par exemple, lors d'une prochaine mission),
  — vous ***répartir*** le travail afin que chaque étudiant soit responsable de la programmation d'une ou plusieurs classe(s).

- Chaque étudiant d'un groupe doit être en charge d'une partie du travail de programmation. Vous indiquerez ***toujours*** en commentaires d'une classe, le ou les auteur(s) de la classe.

- Chaque étudiant est responsable de la bonne organisation de la mission et de l'équilibre entre son travail personnel et sa participation active au groupe.

8 Remise des produits

Les produits de la mission (code, tests et rapport) sont à soumettre sur INGInious (https://inginious.info.ucl.ac.be/course/LSINF1121-2015) pour le **vendredi 13 novembre**, à ***18h00*** **dernier délai**. Passé ce délai, il sera impossible d'effectuer une nouvelle soumission.