IFT 2015 – Structures des donnée
Devoir #2

Manuela Girotti

1 Auto-évaluation

Le programme fonctionne correctement.

2 Code en Java

Voir le fichier `GIROTTI_Tp2.java` joint [1].

2.1 Structures des donnée utilisées

Pour le codage on a utilisé les structures des donnée suivantes :

— pour le catalogue de la pharmacie `Apotheek_Catalogue`, on implément un arbre binaire de recherche où chaque noeud est identifié par le nom du médicament et contient un arbre binaire de recherche avec noeuds qui contiennent une date d'expiration et le nombre des boîtes du médicament considéré qui expirent à la date indiquée :

```
TreeMap<String , TreeMap<LocalDate , Integer>> Apotheek_Catalogue
        = new TreeMap<String , TreeMap<LocalDate , Integer >>();
```

— pour la liste des commandes `orders`, on implément un arbre binaire de recherche où chaque noeud contient le nom du médicament et la quantité des boîtes à commander :

```
TreeMap<String , Integer> orders = new TreeMap<String , Integer >();
```

3 Analyse temporelle théorique (pire cas)

Soit :

— N le nombre de types de médicaments différents disponible dans la pharmacie ; si on assume que notre pharmacie est bien fournie et possède un espace infini de stockage, N sera le nombre dominant dans l'analyse de complexité ;

— p_i le nombre des dates d'expiration pour chaque médicament i qu'on a en stock ; on peux estimer que $p_i \ll N$: il existe un nombre énorme des médicaments différents dans le monde, mais pour chaque médicament il est suffisant d'avoir quelque stock avec des dates d'expiration raisonnables pour avoir une pharmacie bien fournie. Par conséquent, on peux assumer que toutes les valeurs p_i sont bornées par une constant positive κ :

$$\max\{p_i\} \leq \kappa, \qquad \kappa \in \mathbb{R}_+;$$

— n le nombre de médicaments reçus à chaque livraison ;
— M le nombre d'items sur la liste de demande (dans le pire cas, si on n'a pas assez de stock pour aucun médicament et on a beaucoup de clients : $M \approx N$) ;
— K le nombre d'items sur la prescription (sauf pour les cas extrêmement hypocondriaques $K \ll N$).

On assume que toutes les opérations élémentaires (opérations algébriques, initialisation des variables, etc.) prennent un temps d'exécution constant $\mathcal{O}(1)$.

On assume aussi que chaque appel à `Scanner` et `BufferedWriter` (lecture et écriture d'un fichier) prends un temps d'exécution constant ($\mathcal{O}(1)$), puisque chaque ligne de texte dans les fichiers de input et output contiennent un nombre limité des caractères.

`APPROV` Pour chaque médicament en livraison, le parsing prend un temps constant (donc, $\mathcal{O}(n)$ pour une livraison de n médicaments) et ensuite on appelle la méthode `putToMap` pour chaque médicament livré.

Dans cette méthode on cherche si le médicament est déjà présent dans le catalogue : cela prend un temps

$$\mathcal{O}(\log N),$$

puisque `Apotheek_Catalogue` est un `TreeMap`, i.e. arbre rouge-noir. Si le médicament n'est pas présent, on l'insère dans un nouveau noeud ($\mathcal{O}(\log N)$).

Si le médicament i est présent, on vérifie si la date d'expiration coincide avec une des dates d'expiration déjà en stock et soit on met à jour le noeud correspondant, soit on ajoute un nouveau noeud dans l'arbre du médicament i. La complexité dans l'arbre du médicament i est $\mathcal{O}(\log p_i)\ \forall\ i$ (cette complexité est borné par une constante : $\mathcal{O}(\log p_i) \subseteq \mathcal{O}(1)$).

En conclusion, la complexité globale de cette partie du code est

$$\mathcal{O}(n \log N).$$

`DATE` Le parsing pour la date courante prend un temps constant $\mathcal{O}(1)$. Soit M le nombre d'items sur la liste de demande `orders` (TreeMap). Si $M = 0$, on imprime seulement `"OK"` : $\mathcal{O}(1)$. Si $M \geq 1$, on parcourt l'arbre et on affiche tous les commandes :

$$\mathcal{O}\left(M\right)$$

Enfin, on vide la liste des commandes ($\mathcal{O}(1)$).

`STOCK` Soit N le nombre de types de médicaments différents disponible dans la pharmacie, la méthode `updateStock` parcourt l'arbre `Apotheek_Catalogue` et pour chaque noeud (i.e. pour chaque médicament) elle parcourt l'arbre de stock en vérifiant que la date d'expiration soit après la date courante ($\mathcal{O}(p_i) \subseteq \mathcal{O}(1)$ une fois que le médicament i a été visionné dans le catalogue, $\forall\, i$).

Donc, la méthode `updateStock` a une complexité de l'ordre de

$$\mathcal{O}\left(N\right).$$

Pour la même raison, l'affichage du stock sur le fichier de sortie prends un temps de l'ordre

$$\mathcal{O}\left(N\right).$$

`PRESCRIPTION` soit K le nombre d'items sur une prescription, alors le parsing prend prends un temps $\mathcal{O}(K)$ pour cette prescription.

Ensuite, pour chaque item dans la prescription, on appelle la méthode `isAvailable` pour verifier si on a assez de boîtes de médicament à vendre au client. Cette méthode parcourt le catalogue `Apotheek_Catalogue` pour trouver le médicament requis ($\mathcal{O}(\log N)$) et contrôle la date d'expiration ($\mathcal{O}(p_i) \subseteq \mathcal{O}(1)$ dans le pire cas).

Si on possède la quantité exacte pour remplir la prescription, on l'affiche sur le fichier d'output et on met à jour le catalogue ($\mathcal{O}(1)$; la mise-à-jour a lieu pendant l'exécution de `isAvailable`). Sinon, on appelle la méthode `addToOrders` : on parcourt la liste de commande pour vérifier si le médicament a déjà été ajouté à la liste ($\mathcal{O}(\log M)$) et soit on met à jour l'ordre soit on ajoute un nouveau ordre pour le médicament.

En total, pour chaque médicament dans la prescription, cette partie du code prends un temps de l'ordre $\mathcal{O}(\log M + \log N)$. Pour chaque prescription avec K items, la complexité est

$$\mathcal{O}(K \log M + K \log N).$$

Références

[1] plusieurs consultations pendant l'écriture du programme : https://docs.oracle.com/en/