Université de Montréal

Rapport du devoir #2 :
Gestion de stock d'une pharmacie

Par
Steven Han, 20129462
Yassine Hajouji Idrissi, 20038060
et
Chen Zong Jiang, 20122046

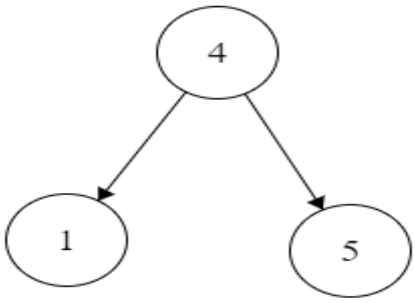
Département d'informatique et de recherche opérationnelle

Travail présenté à Samuel Ducharme
Dans le cadre du cours IFT-2015 section A
Structures de données

10 Juillet 2019

1. Autoévaluation

Nous avons effectué chaque test du fichier testsE19 et nous pouvons conclure que notre code fonctionne parfaitement. Le fichier généré par notre code était exactement identique à ceux des tests. Selon nous, il n'est pas possible d'avoir un code plus performant que le nôtre en utilisant les arbres binaires de recherche. Le code peut être plus simple, mais nous somme convaincus d'avoir le temps de complexité le plus minimale possible.

Pour une mise en contexte de notre programme, nous avons utilisé un arbre AVL pour organiser le stock de médicament. L'idée est que notre arbre est en fait un arbre de *int* représentant le numéro du médicament et que pour chaque nœud, il y a une liste de médicament associé. Cela facilite la recherche (plus facile de chercher des *int* que des objets). Par exemple, si on a trois médicaments : Médicament1, Médicament 4 et Médicament 5 et qu'on veut les insérer dans cet ordre. Alors l'arbre aura cette allure : 

. Le nœud 4 aura une liste contenant un élément (une instance de Médicament4), le nœud 5 aura une liste contenant un élément (une instance de Médicament5) et même chose pour le nœud 1. Cette façon d'organiser l'arbre sera plus facile si on veut modifier les éléments d'un certain nœud : il suffit de faire un *.add* ou un *.remove* de la liste.

Il est à noter que notre arbre ne supprime jamais de nœud. Il y a des avantages et des inconvénients, mais lorsqu'on a essayé de supprimer les nœud lorsque leur liste de médicament était vide, cela causait problème. C'est à cause du parcours de l'arbre où on utilise la récursion : si on modifie l'arbre, il se peut que les autres appels de la fonction n'aient pas la nouvelle version de l'arbre et cela peut causer des comportements étranges. Donc, pour éviter ce problème, on ne supprimera aucun nœud. L'avantage de ne pas supprimer de nœuds est que la structure est déjà là et qu'on n'a pas besoin de la changer. Par exemple, si on avait un gros stock en 2012 (donc gros arbre AVL) et qu'il est maintenant 2022 et qu'on veut refaire le stock de nos médicaments, une grosse partie de nos médicaments seront là et on à qu'a faire un *.add* (qui a un temps de complexité de $O(1)$) lorsqu'on trouve le nœud approprié. C'est beaucoup plus mieux que de faire des insertions à chaque renouvellement de médicament. L'inconvénient est que cela peut prendre un peu plus de temps, car il parcourt des nœuds avec des listes vides.

De toute façon, un arbre AVL balancé a en pire cas, pour l'insertion d'un nœud, pour la recherche d'un nœud et pour la supprimation d'un nœud un temps de complexité $O(logn)$. Ce qui est très bon.

2. Analyse de la complexité temporelle en notation grand O

Remarque : pour faciliter l'analyse, on analysera seulement les fonctions qui ont une complexité plus grande que $O(1)$, sauf si la fonction n'est pas évidente.

2.1 PRESCRIPTION

Voici le code pour le traitement de la fonction PRESCRIPTION :

```java
/* Fonction qui se fait appeler lorsqu'on lit le mot PRESCRIPTION. Il cherchera le noeud dans l'arbre et
 * dependamment de la reponse, on ajoutera un OK ou un COMMANDE.
 *
 * @param prescriptions : liste de prescriptions en string
 * @see AVLtree.rechercher(int identificateur, Node noeud, int quantiteMed, Date dateActuelle)
 */
public static void prescription(String prescriptions) throws ParseException{
    compteurPrescription++;
    // On separe notre string pour qu'il ait la forme [PRESCRIPTION, MedicamentX _ _, MedicamentY _ _, ...]
    String[] tabPrescription = prescriptions.split( regex: "\n");
    ArrayList<String> sortiePrescription = new ArrayList<>();
    sortiePrescription.add("PRESCRIPTION " + compteurPrescription);
    for (int i = 1; i < tabPrescription.length; i++) {
        /* Les champs ont beaucoup d'espace entre eux et \\s+ uniformise l'espace.
           Ex : [Medicament1, "", "", .., 5, ..., DATE] sera [Medicament1,5,DATE] Il enleve les espaces blancs
        */
        String[] med = tabPrescription[i].split( regex: "\\s+");
        try {
            int identificateur = parserNumeroMed(med[0]);
            int dose = Integer.parseInt(med[1]);
            int repetition = Integer.parseInt(med[2]);
            int quantite = dose * repetition;
            Date date = convertirStringDate(datePlusRecenteFichier);
            boolean estDansArbre = arbreAVL.rechercher(identificateur, arbreAVL.getRacine(), quantite, date);
            // Si le medicament n'est pas valide pour la prescription
            if (!estDansArbre) {
                sortiePrescription.add("Medicament" + identificateur + " " + dose + " " + repetition + " COMMANDE");
                // Si c'est un medicament qui est deja dans la commande, on doit les fusionner
                if (occurrenceCommandes.containsKey("Medicament" + identificateur)) {
                    int ancienneValeur = (int) occurrenceCommandes.get("Medicament" + identificateur);
                    occurrenceCommandes.replace("Medicament" + identificateur, ancienneValeur + quantite);
                }
                // Si c'est un nouveau medicament qui n'est pas dans la liste des commandes
                else {
                    occurrenceCommandes.put("Medicament" + identificateur, quantite);
                }
            }
            // Si le medicament est valide pour la prescription
            else {
                sortiePrescription.add("Medicament" + identificateur + " " + dose + " " + repetition + " OK");
            }
        } catch (ParseException e) {
            throw new ParseException("Erreur de format pour le " + med[0], 2);
        }
    }
    affecterSortie(sortiePrescription);
}
```

Le temps de complexité des méthodes de *hashmap* seront $O(1)$. On peut voir qu'il y a une recherche de nœud imbriqué dans une boucle, donc en pire cas, la boucle fera m prescriptions. Ensuite, pour chaque prescription, on doit chercher sa valeur dans l'arbre AVL. Comme mentionné dans la première partie, une recherche d'un nœud dans un arbre AVL balancé est en pire cas $O(logn)$. À cause de l'imbrication, le temps de complexité pour cette fonction devient $O(m * logn)$. Il n'y a pas k, car ce sera la fonction DATE qui s'en occupera.

2.2 APPROV

Voici le code pour la fonction APPROV :

```java
/* Fonction qui se fait appeler lorsqu'on lit le mot APPROV. Cette fonction ajoute dans l'arbre AVL les nouveaux
 * medicaments de la section APPROV.
 *
 * @param listeMed : liste de medicament sous forme "APPROV : \n MedicamentX _ _ \n ...
 */
public static void approv(String listeMed) throws ParseException {
    // Separe chaque medicament et les met dans un tableau
    String[] tabMed = listeMed.split( regex: "\n");
    // On ignore le premier element, car c'est APPROV
    for (int i = 1; i < tabMed.length; i++) {
        // \\s+ uniformise les espaces, comme mentionne dans la fonction prescription
        String[] champsMedicament = tabMed[i].split( regex: "\\s+");
        try {
            int identificateur = parserNumeroMed(champsMedicament[0]);
            int quantite = Integer.parseInt(champsMedicament[1]);
            String date = champsMedicament[2];
            Medicament nouveauMedicament = new Medicament( nom: "Medicament" + identificateur, quantite, date);
            // Insert le nouveau medicament dans l'arbre
            arbreAVL.setRacine(arbreAVL.checkInsert(arbreAVL.getRacine(), identificateur, nouveauMedicament));
        } catch (ParseException e) {
            throw new ParseException("Erreur de format pour le " + champsMedicament[0], 2);
        }
    }
    sortie.add("APPROV OK");
}
```

Similairement à la fonction PRESCRIPTION, il y a une boucle qui traverse chaque médicament à ajouter dans notre arbre. Donc son temps de complexité, en pire cas, sera $O(n)$. Ensuite, pour chaque médicament k, on doit l'insérer dans l'arbre. Or, comme mentionné dans la partie 1, le temps de complexité pour l'insertion d'un nœud dans un arbre AVL balancé est $O(logn)$. Donc, à cause de l'imbrication, le temps de complexité, en pire cas de la fonction APPROV sera $O(n * logn)$.

2.3 DATE

Voici le code pour la fonction DATE :

```java
/* Fonction qui se fait appeler lorsqu'on lit DATE. On genere la liste des medicaments a commander.
 *
 * @param date : date a laquelle la commande s'effectuera
 */
public static void date(String date) {
    // Si notre liste de commande est vide
    if (occurrenceCommandes.size() == 0) {
        sortie.add(date + " OK" + "\n");
    }
    // Si on a des commandes a traiter
    else {
        ArrayList<String> commandesSortie = new ArrayList<>();
        // On parcours toutes les valeurs du hashmap et on les affectes dans la liste de sortie
        Iterator it = occurrenceCommandes.entrySet().iterator();
        while (it.hasNext()) {
            Map.Entry pair = (Map.Entry)it.next();
            commandesSortie.add(pair.getKey() + " " + pair.getValue());
        }
        // On le met dans l'ordre croissant des noms des medicaments
        Collections.sort(commandesSortie);
        sortie.add(date + " COMMANDES :");
        affecterSortie(commandesSortie);
        // On reinitialise les variables qui concerne la commande pour les nouvelles commandes du futur
        occurrenceCommandes = new HashMap();
    }
}
```

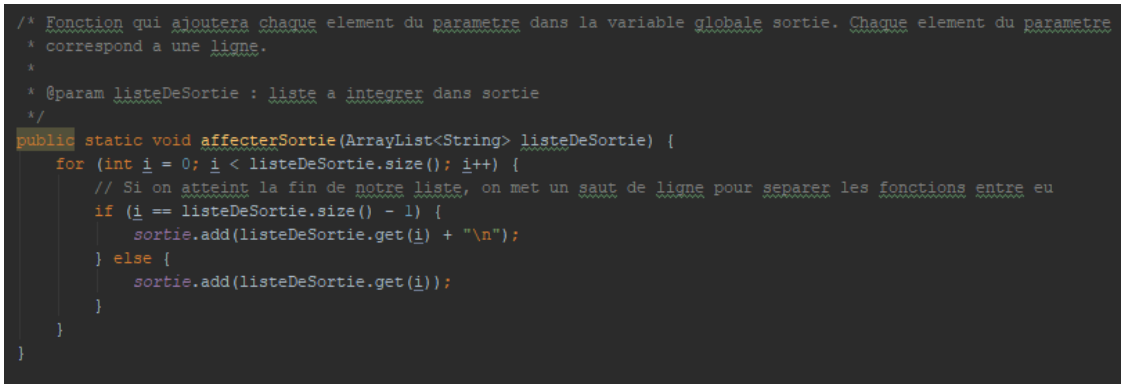
Cette boucle est une itération sur une table de hachage. Ce n'est pas $O(k)$. Ce sera plutôt $O(k+s)$ où s est la taille de la table de hachage. Sur la documentation de la fonction *hashmap* de Java[1], on peut lire ce passage: « Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). ». Pour le *.add*, son temps de complexité en pire cas est $O(1)$.

Le temps de complexité en pire cas pour les *Collections.sort()* de java est $O(ylogy)$. Ce temps peut varier à $O(y)$ si la liste est assez triée. Voici la citation provenant de la documentation de la classe *Collection*[2] : « This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays. » Dans notre cas, le temps sera $O(k\ logk)$.

Il reste qu'à traiter la fonction *affecterSortie*.

[1] https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
[2] https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

Voici le code pour la fonction *affecterSortie :*



```java
/* Fonction qui ajoutera chaque element du parametre dans la variable globale sortie. Chaque element du parametre
 * correspond a une ligne.
 *
 * @param listeDeSortie : liste a integrer dans sortie
 */
public static void affecterSortie(ArrayList<String> listeDeSortie) {
    for (int i = 0; i < listeDeSortie.size(); i++) {
        // Si on atteint la fin de notre liste, on met un saut de ligne pour separer les fonctions entre eu
        if (i == listeDeSortie.size() - 1) {
            sortie.add(listeDeSortie.get(i) + "\n");
        } else {
            sortie.add(listeDeSortie.get(i));
        }
    }
}
```

Son temps de complexité est $O(x)$ ($o\grave{u}\ x\ est\ le\ nombre\ d'\acute{e}l\acute{e}ment\ dans\ la\ liste$).

En mettant ça ensemble, on obtient, pour la fonction DATE, un temps de complexité de $O(k+s)+O(k\ logk)+O(k)$. En simplifiant, cela donne $O(k\ logk)$.

2.4 Résumé

Voici un résumé des temps de complexité en pire cas de chacun :

$$PRESCRIPTION : O(m\ logn)$$

$$APPROV : O(n\ logn)$$

$$DATE : O(k\ logk)$$

3. Bonus de complexité

Tout ce qu'on a dit est vrai, sauf qu'il ne faut pas oublier que chacune de ses fonctions se fait appeler par une boucle d'une fonction nommé *agirSelonFonction*. Voici le code :

```java
/* Fonction centrale qui lie chaque fonction du fichier (APPROV, STOCK, PRESCRIPTION, DATE)
 *
 * @param textSeparee : fichier d'entree separee par fonction. Il sera de forme [APPROV ..., PRESCRIPTION ..., ...]
 */
public static void agirSelonFonction(String[] textSeparee) throws ParseException {
    for (int i = 0; i < textSeparee.length; i++) {
        /* On separe le nom de la fonction de ses arguments. Ex [APPROV :, Medicament1...,]
         * le \\s+ fait abstraction du nombre d'espace que l'utilisateur met. Ex : "         " = " ".
         */
        String[] tabFonction = textSeparee[i].split( regex: "\\s+");
        // 0 car ce sera le nom de la fonction
        switch (tabFonction[0]) {
            case "APPROV": {
                try {
                    approv(textSeparee[i]);
                } catch (ParseException e) {
                    throw new ParseException("Erreur de format pour " + textSeparee[i], 2);
                }
            } break;
            case "DATE": {
                datePlusRecenteFichier = tabFonction[1];
                date(datePlusRecenteFichier);
            } break;
            case "PRESCRIPTION": prescription(textSeparee[i]); break;
            case "STOCK": stock(datePlusRecenteFichier); break;
            default : throw new NoSuchElementException("Erreur, mot inconnue : " + tabFonction[0]);
        }
    }
}
```

Donc, si on tient compte de tout le programme, le temps de complexité de nos 3 fonctions vont ajouter une nouvelle variable : le nombre de ligne (appelons le x). Comme c'est des imbrications, nos temps de complexité vont devenir :

$$PRESCRIPTION : O(m\ logn\ x)$$

$$APPROV : O(n\ logn\ x)$$

$$DATE : O(k\ logk\ x)$$