Rapport
IFT 2015 : Devoir 2

Absalon Marion (20211423) et Sooben Vennila (20235256)

10 juillet 2023

1 Auto-évaluation

Évaluation du fonctionnement du programme: Correct

Notre code fonctionne correctement sauf preuve du contraire.

1.1 Description du code

Nous avons implémenté une manière de gérer un système de stockage de médicaments pour une pharmacie en *Java* en utilisant **TreeSet**, **HashMap** et **ArrayList** en ayant en tête la complexité spatiale et temporelle.

Analyse Amortie : Nous avons pris la décision de s'autoriser à faire certaines opérations "coûteuses" afin d'ameliorer l'efficacité. En effet, pour pouvoir une meilleure complexité temporelle grâce à l'utilisation de TreeSet, nous avons fait le choix de rendre notre complexité spatiale "moins" bonne notamment en rajoutant un attribut UUID qui permettra de rendre chaque médicaments uniques dans notre TreeSet. Ceci permet de faciliter les opérations sur les médicaments ayant le même nom mais pas forcément la même date d'expiration.

Voici une explication de ce que fait notre code:

1. Lire le fichier passé en paramètre et d'utiliser des techniques de *string manipulation* et *parsing* afin d'en extraire les données.
2. La première étape est de chercher les commandes en ordre dans le fichier txt et de procéder avec leurs méthodes respectives dès l'obtention de la commande.
3. Ces données sont ensuites ajoutés à un *TreeSet* si la commande est **APPROV**. Nous faisons une vérification sur la date d'expiration et le nom pour déterminer s'il faut modifier le stock ou juste rajouter une nouvelle instance de *Medicament*.

4. Si la commande est **PRESCRIPTION**, nous cherchons le médicament dans l'arbre puis nous vérifions le stock et une décision *(COMMANDE ou OK)* est prise. Si la décision est *OK*, nous changeons le stock de l'arbre.

5. Si la commande est **DATE**, nous procédons avec une sortie des commandes à passer et nous actualisons la nouvelle date au besoin.

6. Si la commande est **STOCK**, nous procédons avec une sortie du stock donc tous les éléments de l'arbre.

7. A chaques itérations de la boucle, notre fichier de sortie est modifié.

1.2 Tests effectués

Nous avons testés avec les fichiers mis à notre disposition sur StudiuM.

2 Analyse de la complexité temporelle (pire cas) théorique en notation grand O

2.1 Analyse pire cas PseudoJava

- $O(n)$ pour la boucle for qui dépendra de la taille de l'arbre.
- Le reste en O(1)
- Max $\Rightarrow$ O(n)

```
public static Medicament findClosest(String nom, LocalDate date, int
    total) {
    long closestDiff = Long.MAX_VALUE;
    for (Medicament M : tree) {
        long diff = ChronoUnit.DAYS.between(M.getDateExpi(), date);
        if (M.getStock()>=total && Math.abs(diff)<=closestDiff &&
            M.getNom().equals( nom)) {
            // reste du code ici
        }
    // reste du code ici
}
```

Figure 1: Méthode findClosest() en partie pour trouver la date la plus proche dans *BST*

- $O(nlogn)$ pour Collections.sort
- $O(n)$ pour parcourir l'arbre dans le for loop
- $O(1)$ pour le reste

```
public static ArrayList<String> outputStock(){

        ArrayList<String> medsStock =new ArrayList<String>();
        for (Medicament med:tree) {
            medsStock.add(med.getNom() + "\t" + med.getStock() + "\t"
                + med.getDateExpi());
        }

        Collections.sort(medsStock);

        // va permettre de gerer le cas ou on a le meme nom
        Collections.sort(medsStock, (med1, med2) -> {
        String[] med1Parts = med1.split("\t");
        String[] med2Parts = med2.split("\t");

        // Compare names
        int nameComparison = med1Parts[0].compareTo(med2Parts[0]);
        if (nameComparison != 0) {
            return nameComparison;
        }

        // If names are the same, compare dates
        return med1Parts[2].compareTo(med2Parts[2]);
    });

    return medsStock;
}
```

Figure 2: Méthode outputStock() en partie dans *BST*

- $O(logn)$ pour add
- $O(logn)$ pour remove

```
// enleve un medicament de l'arbre
public static void removeMed(Medicament med){
    tree.remove(med);
    setTree(tree);
}

// ajoute un medicament a l'arbre
public static void addMed(Medicament med){
    tree.add(med);
    setTree(tree);
}
```

Figure 3: Add et Remove de l'arbre dans *BST*

- $O(n)$ d'apres a methode de TreeSet

```
public static void removeAllExpired(LocalDate date){

    tree.removeIf(mmm -> mmm.getDateExpi().isBefore(date));
    tree.removeIf(mmm -> mmm.getDateExpi().isEqual(date));

 }
```

Figure 4: Methode qui va enlever les medicaments expires dans *BST*

- $O(n)$ pour la boucle

```
public static Medicament searchMed(Medicament med) {

for (Medicament medicament:tree) {

    if (med.getNom().equals(medicament.getNom()) &&
        med.getDateExpi().equals(medicament.getDateExpi())){

return medicament;

    }
}
}
```

Figure 5: Methode permettant de chercher un medicament dans *BST*

- $O(n)$ pour une boucle (pas imbrique = $O(n)$
- $O(nlogn)$ pour Collections.sort
- $O(1)$ le reste

$\Rightarrow O(nlogn)$

```
    public static String outputCommande(){
       String output ="";

       Map<String, Integer> num = new HashMap<>();

       for (Record record : recordCommande) {
           String name = record.name;
           int number = record.number;
           num.put(name, num.getOrDefault(name, 0) + number);
       }

       List<Record> uniqueRecords = new ArrayList<>();
       for (Map.Entry<String, Integer> entry : num.entrySet()) {
           String name = entry.getKey();
           int numb = entry.getValue();
           uniqueRecords.add(new Record(name, numb));
       }

       Collections.sort(recordCommande, Comparator.comparing(r ->
            r.name));
       for (Record record : recordCommande) {
           output = output + (record.name() + "\t" + record.number() +
               "\n");
       }
   }
```

Figure 6: Code dans *Tp2* permettant d'envoyer à une methode un string qui permettra d'ecrire les commandes dans le fichier

- $O(n)$ pour la boucle
- $O(1)$ pour le reste

$\Rightarrow O(n)$

```
      public static void readTheThing() {

     while (line != null) {

     // code ici ....

      if (instruction == "STOCK") {

         writer.write("STOCK " + getCurrentDate() + "\n");
         ArrayList<String> stock = BST.outputStock();

         for (int j=0; j<stock.size(); j++){
           writer.write(stock.get(j) + "\n");
         }

         writer.write("\n");
         line = reader.readLine();

       }
       else if (instruction == "DATE"){
         continue;

       }
     }
 }
```

Figure 7: Methode dans *Tp2* qui permet de lire, ecrire dans le fichier

- $O(1)$ remove
- $O(n)$ pour la boucle
  $\Rightarrow O(n)$

```
        public static String methodPrescription(String line, LocalDate
            date){

           for (int i = 0; i < parts.length; i++) {
             parts[i] = parts[i].trim();
           }

           BST.outputStock();
           Medicament foundMed = BST.findClosest(med,date);
            //reste du code

           if (foundMed != null && ((foundMed.getStock()<total))){
               outputstring=(med + "\t" + num1 + "\t"+ num2 +"\t"+
                   "COMMANDE");
           } else if (foundMed !=null && foundMed.getStock()>=total){

           BST.removeMed(foundMed);
           foundMed.setStock(foundMed.getStock()-total);
           BST.addMed(foundMed);
           // reste du code
  }
 }
}
```

Figure 8: Methode permettant de gerer la commande PRESCRIPTION dans *Tp2*

- $O(n)$ pour parse
- $O(n)$ pour itérations de boucle
- $O(1)$ pour le reste

$\Rightarrow O(n)$

```
Medicament medicament = new
    Medicament(med,UUID.randomUUID(),date,num);
Medicament medoc = BST.searchMed(medicament);
if (medoc != null) {
   medoc.setStock(medoc.getStock()+ num);
   BST.addMed(medoc);
}
else{
   BST.addMed(medicament);
}
```

Figure 9: Bout de code de la methode *stringToMed* dans *Tp2*

3 Analyse de la complexité temporelle empirique

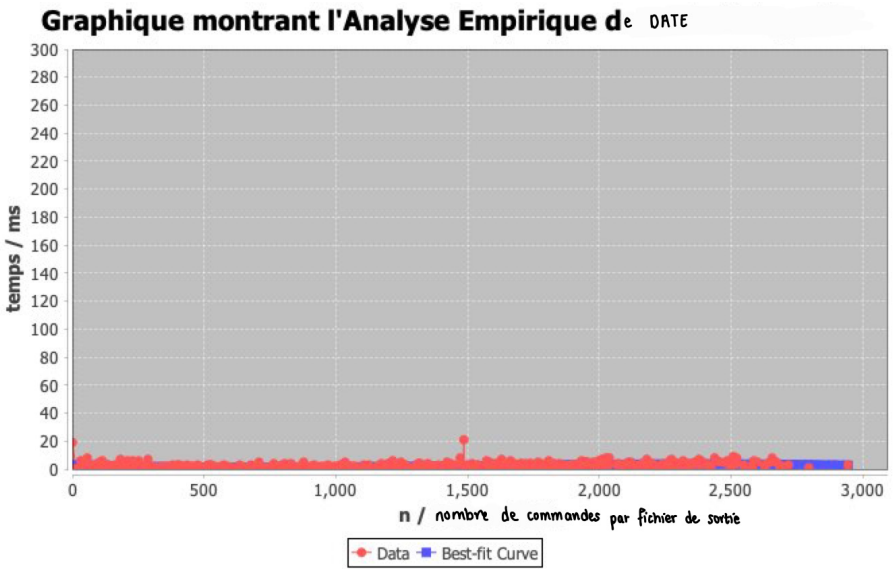
3.1 Graphique de temps d'exécution en fonction de n

L'analyse a été effectuée avec le temps ainsi que les données suivantes dans 3 diagrammes différents:
nombre de types de médicaments différents, nombre d'items sur la prescription, nombre d'items sur la liste de demande et tout le code dans son intégralité dans un 4ème diagramme;

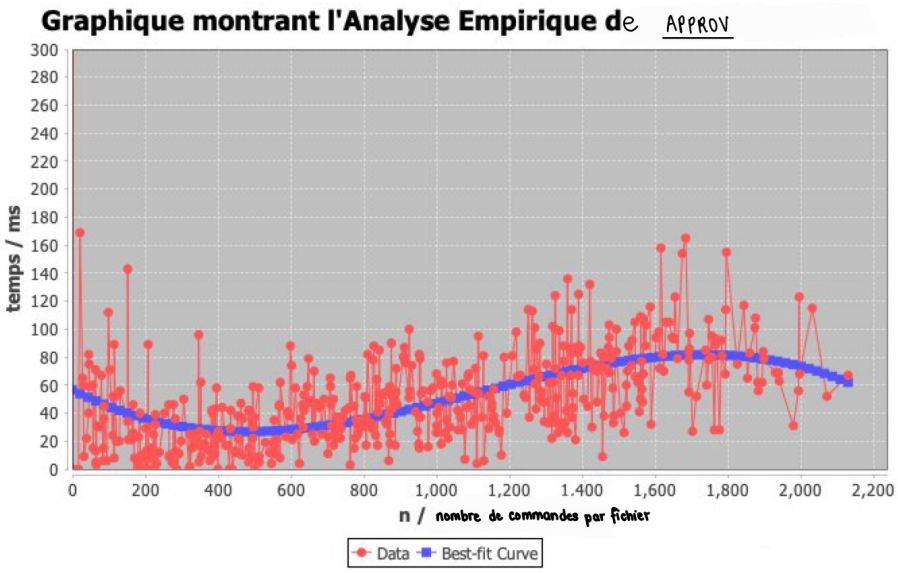
```
long startTime = System.currentTimeMillis();
long endTime = System.currentTimeMillis();
long time = (endTime - startTime);
```

Nous avons testés avec les fichiers mis à notre disposition sur StudiuM mais afin d'augmenter notre n, nous avons également fait des fusions de fichiers grâce à. JFreeChart et un "script" Java pour générer des fichiers aléatoires.

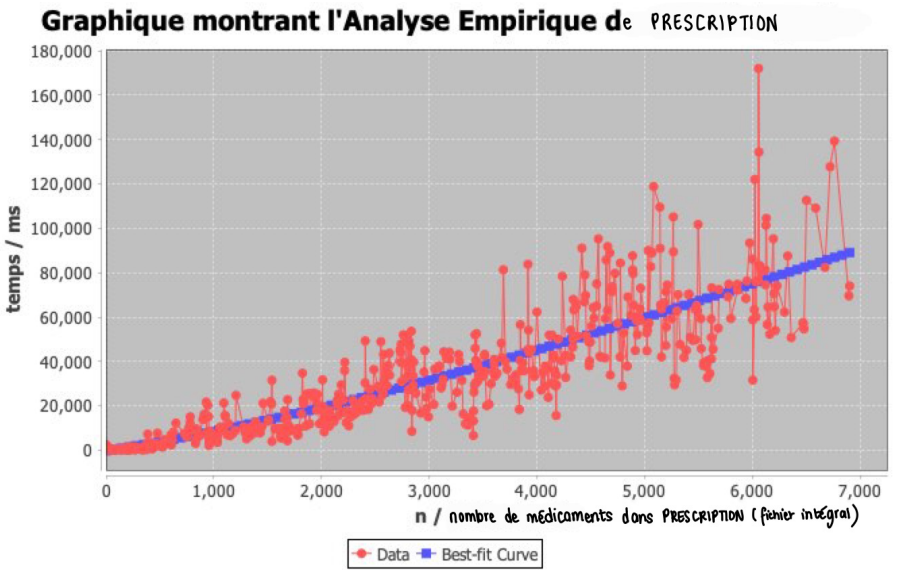
- **DATE** : Utilisé avec $0 < n < 3000$
- On peut apercevoir la ligne de $O(1)$



- **APPROV**: Utilisé avec 0 < n < 2200
- On peut apercevoir la boucle de $O(lgn)$ encore une fois.



- **PRESCRIPTION** : Utilisé avec 0 < n < 7000
- On peut apercevoir la courbe de $O(N/10)$ et donc notre code roule en $O(nlgn)$



- **CODE INTEGRALE** : Utilisé avec 0 < n < 19000
- On peut apercevoir la ligne de $O(nlgn)$