Rapport | IFT-2015

Tarik Hireche : 202 301 89

18. October 2025

1

1.1 Problème 1 (5pts)

a)

```java
// Données de l'énoncé (Q1 - distribution de livres)
int[] books        = {2, 1, 3, 4, 3};     // livre demandé par chaque participant i
int[] distribution = {1, 3, 3, 2, 4};     // ordre réel de remise des livres
```

Dans ce premier problème, j'avais à simuler la distribution de livres et à calculer la « colère » des participants.

Je me suis rendu compte rapidement qu'il s'agissait d'un cas classique de files (queues). Chaque livre a sa propre file d'attente et le premier de la file obtient son exemplaire.

Ce qui était plus délicat, c'était de gérer la colère:

- comment compter ceux qui se fâchent chaque fois qu'un participant derrière eux reçoit son livre ?

J'ai choisi de le faire simplement : chaque fois que je sers quelqu'un, j'ajoute +1 à la colère de tous ceux avant lui qui attendent encore.

b)

```java
// Structure de données : une file par livre
List<Queue<Integer>> waitlist = new ArrayList<>();
for (int i = 0; i <= books.length; i++) waitlist.add(new LinkedList<>());

// Peupler les files d'attente par livre demandé
for (int i = 0; i < books.length; i++) {
    int livre = books[i];
    int participant = i + 1;
```

```
9       waitlist.get(livre).add(participant);
10  }
```

Ici, j'ai opté pour la simplicité. `waitlist[l]` représente la file des gens qui veulent le livre numéro `l`. Ce choix rend le code très intuitif : on sait tout de suite qui attend quoi.

c)

```java
// Simulation + colère
boolean[] served = new boolean[books.length + 1];
int[] anger      = new int[books.length + 1];
List<Integer> servedOrder = new ArrayList<>();

for (int livre : distribution) {
    Queue<Integer> q = waitlist.get(livre);
    if (q.isEmpty()) continue;

    int p = q.poll();
    served[p] = true;
    servedOrder.add(p);

    // Bon là, je gère la frustration :
    // tous ceux avant lui (numéro plus petit) non servis prennent +1.
    for (int i = 1; i < p; i++) {
        if (!served[i]) anger[i]++;
    }
}
```

Au début, j'ai essayé de modéliser la colère autrement (avec un compteur global par livre) mais ça devenait trop flou.

Cette approche, bien que naïve, est claire et solide. Elle m'a forcé à bien visualiser les tours de distribution comme une vraie scène : des gens qui attendent, certains qui bougonnent, d'autres qui partent contents.

d)

```java
// Affichage final
System.out.print("participant served: ");
for (int p : servedOrder) System.out.print(p + " ");
System.out.println();

int totalAnger = 0;
for (int i = 1; i < anger.length; i++) totalAnger += anger[i];
System.out.println("total anger: " + totalAnger);
```

Complexité :

- Construction des files : $O(n)$
- Distribution : au pire $O(n^2)$ à cause de la mise à jour des colères.

Je sais que techniquement on pourrait faire mieux mais cette implémentation fonctionne très bien pour ce tp.

e)

```text
participant served: 2 3 5 1 4
total anger: 4
```

Tout correspond à l'exemple du sujet. J'ai aussi testé quelques cas limites :

- tous demandent le même livre → colère = 0 ;
- ordre parfait → colère = 0 ;
- ordre inversé → colère maximale mais toujours cohérente.

1.2 Problème 2 (5pts)

Ce problème-là m'a beaucoup plu : on travaille avec une **hiérarchie** d'employés, ce qui revient à gérer un arbre. Chaque employé connaît son boss et ses subordonnés. Les opérations à coder : ajouter, déplacer, retirer, afficher, et même trouver le boss commun le plus bas.

```java
// Node interne : un employé
class Node {
    String name;
    Node boss;
    int level;
    List<Node> reports = new ArrayList<>();
    Node(String name, int level) { this.name = name; this.level = level; }
}
```

J'ai construit une `HashMap<String, Node>` pour retrouver n'importe quel employé instantanément, et un `Node root` pour le big boss (Claude dans l'exemple). Le plus gros défi : ne jamais briser les liens entre boss et subordonnés, surtout quand on déplace ou supprime un employé.

a)

- **addEmployee(name, level, bossName)** : si `level == 1`, c'est la racine ; sinon, je le rattache au boss avec `boss.reports.add(newEmp)` et `newEmp.boss = boss`.
- **printChart(root)** : j'ai utilisé une récursion avec indentation pour avoir un rendu clair comme un organigramme.
- **move(employee, boss)** : bon là, il faut enlever la personne de son ancien boss et la rattacher au nouveau. Faut pas oublier de mettre à jour la référence `boss`.
- **remove(employee, boss)** : probablement la plus délicate. J'ai dû transférer tous les subordonnés de la personne supprimée vers un nouveau boss avant de la retirer. Une erreur ici, et l'arbre devient incohérent.
- **printAllBoss(employee)** : on remonte la chaîne de supervision jusqu'à la racine. C'est simple, mais ça aide à visualiser la hiérarchie.
- **lowestCommonBoss(e1, e2)** : je construis la liste des ancêtres du premier employé, puis je remonte le deuxième jusqu'à trouver un boss en commun. Une belle petite recherche ascendante, efficace et élégante.

b)

Résultat typique après les tests :

```text
=== Structure initiale ===
Claude
|  Bob
|  |  Alice
|  Elaine
|  |  David

=== Déplacement d'Alice sous Elaine ===
Claude
|  Bob
|  Elaine
|  |  David
|  |  Alice

=== Suppression de Bob (on transfère à Elaine) ===
Claude
|  Elaine
|  |  David
|  |  Alice
```

J'ai vérifié les cas d'usage : rien ne casse. Les références restent valides et les impressions sont bien structurées.

1.3 Conclusion

Ce TP m'a demandé pas mal de rigueur, surtout sur le deuxième problème. Au début, je trouvais la hiérarchie un peu abstraite : beaucoup de références, des listes de listes… Mais plus je le codais, plus ça faisait sens. C'est aussi le TP où j'ai compris à quel point une bonne représentation mentale du problème vaut mille diagrammes UML.

Côté difficultés :

- comprendre comment gérer les liens sans créer de cycles (quand on bouge un employé, il ne doit pas devenir son propre boss !);
- faire en sorte que le `printChart` reste propre avec plusieurs niveaux d'imbrication;
- et tester tous les cas de suppression sans tout casser.

2 Sources utilisées pour ce devoir:

Outre les notes de cours et ChatGPT pour clarifier certains points logiques, j'ai aussi consulté :

- source I – Documentation officielle Java (java.util.*)
- source II – GeeksForGeeks : rappels sur les arbres et BFS/DFS
- source III – StackOverflow : retirer proprement d'une ArrayList en itération
- source IV – Programiz : structures de données en Java

Pour finir j'ai aussi regardé quelques tutoriel sur youtube pour mieux comprendre certaines choses.