Alex Guimbeau S3A

# SAE 3.8 Algorithmique
# Projet – Filtre de Bloom

# Sommaire

# Introduction

Un **filtre de Bloom** est un type de filtre de données utilisé pour identifier rapidement si un élément est présent ou absent dans un ensemble de données. Il fonctionne en utilisant une **table de hachage** pour **mapper** les éléments à des emplacements dans la table, ce qui permet de les retrouver rapidement.

Il est souvent utilisé pour vérifier si un élément a déjà été vu ou non, ce qui est utile dans de nombreux contextes, tels que la détection de spam, la vérification de l'unicité de données dans une base de données, etc.

Il a l'avantage d'être rapide et de nécessiter peu de mémoire, mais il a également l'inconvénient de pouvoir produire des **faux positifs**, c'est-à-dire indiquer qu'un élément est présent alors qu'il ne l'est pas réellement. Pour minimiser ce problème, il est important de choisir une table de hachage de grande taille et de bons algorithmes de hachage.

# Présentation mes implémentations

Le code joint représente une classe Java appelée **FBTab**, **FBArrayList** et **FBLinkedList** qui implémente un **filtre de Bloom** utilisant respectivement un **tableau**, une **ArrayList** et une **LinkedList** de **booléens**. Les classes comprennent une variable d'instance *tab, Alist et Llist*, qui sont les structures utilisées pour stocker les données du filtre, ainsi que deux variables d'instance *TAILLE* et *NB_HASHAGE* qui stockent respectivement la taille du tableau et le nombre de fonctions de hachage utilisées.

Les classes comprennent également un constructeur qui prend en entrée la **taille** de la structure et le **nombre de fonctions de hachage** à utiliser. Les constructeurs initialisent les variables d'instance en utilisant les valeurs passées en paramètre et créent le **tableau** ainsi que **l'ArrayList** et la **LinkedList** de booléens en utilisant cette taille.

Elles comprennent une méthode **add** qui prend en entrée un **entier** et ajoute cet entier au filtre en utilisant les **fonctions de hachage**. Pour chaque fonction de hachage, la méthode **calcule l'indice de la structure** correspondant à l'élément en utilisant la méthode **hash** et met à “true” la case correspondante.

Les classes comprennent également une méthode **contains** qui prend en entrée un **entier** et vérifie si cet entier est présent dans le filtre. Pour cela, la méthode **calcule les indices** de la structure correspondants à l'élément en utilisant les fonctions de **hachage** et vérifie que toutes les cases correspondantes contiennent la valeur “true”. Si une seule case ne contient pas “true”, la méthode renvoie false, sinon elle renvoie true.

Enfin, elles comprennent une méthode **hash** qui calcule les différents **hachages** d'un entier en utilisant une formule spécifique. Cette méthode prend en

entrée **l'élément** à hacher et le **numéro de hachage** à effectuer et renvoie **l'indice** correspondant.

# Le benchmark

## Qu’est-ce qu’un benchmark

Un **benchmark** est une mesure de performance d'un système, d'un logiciel ou d'une méthode dans un contexte donné. Il peut être utilisé pour comparer les performances de différents systèmes ou logiciels, ou pour évaluer l'évolution des performances d'un système au fil du temps.

Il est important de noter que les résultats d'un benchmark ne sont valables que pour le contexte dans lequel ils ont été obtenus. Par exemple, un **benchmark** exécuté sur un ordinateur de bureau peut ne pas être représentatif des performances d'un appareil mobile, même si le matériel est similaire. De même, un **benchmark** exécuté sur un système avec une configuration particulière ne peut pas être généralisé à tous les systèmes de la même manière.

## Implémentation du Benchmark

Le **Benchmark** implémente un banc d'essai pour comparer les différentes implémentations d'un **filtre de Bloom**. La classe comprend trois variables d'instance qui sont des instances des différentes implémentations du filtre de Bloom (**FBTab, FBArrayList et FBLinkedList**) ainsi qu'une variable d'instance *TAILLE* qui représente la **taille** des filtres et une variable d'instance *NB_HASHAGE* qui représente le nombre de **fonctions de hachage** utilisées.

La classe comprend un constructeur qui prend en entrée le nombre de **fonctions de hachage** à utiliser et initialise la variable d'instance *NB_HASHAGE* avec cette valeur. Le constructeur initialise également les variables d'instance *tab, alist et llist* avec de nouvelles instances des implémentations du **filtre de Bloom** en utilisant la taille *TAILLE* et le nombre de **fonctions de hachage** *NB_HASHAGE*.

Elle comprend plusieurs méthodes qui permettent de calculer le temps d'exécution de la méthode **contains** pour chaque implémentation du filtre de Bloom. Les méthodes **tmpExecFBTab**, **tmpExecFBArrayList** et **tmpExecFBLinkedList** prennent en entrée un élément et ajoutent cet élément au filtre avant de mesurer le temps d'exécution de la méthode contains pour vérifier si l'élément est présent dans le filtre.

Elle comprend également une méthode **tauxFauxPosTab** qui prend en entrée le nombre de **fonctions de hachage k** à utiliser et le **pourcentage n de la taille du filtre** qui doit être rempli avec des éléments. Cette méthode initialise un nouveau filtre de Bloom en utilisant la classe **FBTab** et ajoute un certain nombre

d'éléments au filtre en utilisant la boucle for. Ensuite, la méthode calcule le **taux de faux positifs** en vérifiant si des éléments qui ne sont pas présents dans le filtre sont considérés comme étant présents en utilisant la méthode **contains**.

# Analyse des données

## Les configurations testées

Ces tests sont conçus pour évaluer les performances des trois implémentations du filtre: l'une utilisant un **tableau (FBTab)**, l'une utilisant une **ArrayList (FBArrayList)** et l'une utilisant une **LinkedList (FBLinkedList)**. Les tests sont effectués par les méthodes tempExec et **fauxPos**, qui sont appelées par la méthode main.

La méthode **tempExec** effectue deux tests :

- Le premier test mesure le **temps d'exécution** de la méthode **contains** pour chacune des trois implémentations du **filtre de Bloom**, **sans éléments** présents dans le filtre.

- Le deuxième test mesure le **temps d'exécution** de la méthode **contains** pour chacune des trois implémentations du **filtre Bloom**, avec **un élément** présent dans le filtre.

La méthode **fauxPos** effectue trois séries de tests :

- Le premier ensemble de tests mesure le taux de faux positifs pour chacune des trois implémentations, en utilisant **1 fonction de hachage**. Les tests sont effectués avec **1%, 5% et 10%** des éléments présents dans le filtre.

- La deuxième série de tests mesure le taux de faux positifs pour chacune des trois implémentations, en utilisant **3 fonctions de hachage**. Les tests sont effectués avec **1%, 5% et 10%** des éléments présents dans le filtre.

- La troisième série de tests mesure le taux de faux positifs pour chacune des trois implémentations, en utilisant **5 fonctions de hachage**. Les tests sont effectués avec **1%, 5% et 10%** des éléments présents dans le filtre.

# Résultats obtenue

## Tests de vitesse d'exécution

Voici une analyse des résultats :

- Pour le **temps d'exécution** de la méthode **contains** avec **0 élément** dans le filtre :
  - Le filtre basé sur un **tableau** prend environ 3200 nanosecondes pour exécuter la méthode **contains**.
  - Le filtre basé sur une **ArrayList** prend environ 8200 nanosecondes.
  - Le filtre basé sur une **LinkedList** prend environ 3183400 nanosecondes.
- Pour le **temps d'exécution** de la méthode **contains** avec **un élément** dans le filtre :
  - Le filtre basé sur un **tableau** prend environ 1300 nanosecondes.
  - Le filtre basé sur une **ArrayList** prend environ 4600 nanosecondes.
  - Le filtre basé sur une **LinkedList** prend environ 82047800 nanoseconde.

## Calculs du taux de faux positif théorique

- Pour le **taux de faux positifs théorique** avec **1 fonction de hachage** :
  - **Avec 1% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,00995 soit 0,995%.
  - **Avec 5% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,04877 soit 4,877%.
  - **Avec 10% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,09516 soit 9,516%.
- Pour le **taux de faux positifs théorique** avec **3 fonctions de hachage** :
  - **Avec 1% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,00003 soit 0,003%.
  - **Avec 5% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,00270 soit 0,27%.
  - **Avec 10% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,01741 soit 1,741%.
- Pour le **taux de faux positifs théorique** avec **5 fonctions de hachage** :
  - **Avec 1% d'éléments** présents dans le filtre, le taux de faux positifs est proche de 0%.
  - **Avec 5% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,00053 soit 0,053%.
  - **Avec 10% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,00943 soit 0,943%.

## Calculs du taux de faux positif observer

- Pour le **taux de faux positifs observés** avec **1 fonction de hachage** :
    - **Avec 1% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,01005 soit 1,005%.
    - **Avec 5% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,042886 soit 4,2886%.
    - **Avec 10% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,081369 soit 8,1369%.
- Pour le **taux de faux positifs observés** avec **3 fonctions de hachage** :
    - **Avec 1% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,00699 soit 0,699%.
    - **Avec 5% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,046506 soit 4,6506%.
    - **Avec 10% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,097714 soit 9,7714%.
- Pour le **taux de faux positifs observés** avec **5 fonctions de hachage** :
    - **Avec 1% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,00985 soit 0,985%.
    - **Avec 5% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,0265 soit 2,65%.
    - **Avec 10% d'éléments** présents dans le filtre, le taux de faux positifs est d'environ 0,129258 soit 12,9258%.

## Interprétation

On remarque que :

- Le filtre basé sur un **tableau** a des **temps d'exécution** relativement **faibles** pour la méthode **contains**, comparé aux autres implémentations.
- Le filtre basé sur une **LinkedList** a des **temps d'exécution** beaucoup plus **élevés** que les autres implémentations pour la méthode **contains**, mais contrairement aux autres implémentations, il est “beaucoup” plus rapide lorsqu'il est vide que lorsqu’un élément est présent.

- **Plus le nombre de fonctions de hachage est élevé**, **moins le taux de faux positifs est élevé**. Cependant, cela peut avoir un coût en termes de temps d'exécution pour la méthode **add**, qui doit effectuer plus de calculs de hachage.
- **Plus le pourcentage d'éléments présents dans le filtre est élevé, plus le taux de faux positifs est élevé**. Cela est attendu, car plus il y a d'éléments dans le filtre, plus il y a de chances que des **collisions** (c'est-à-dire qu'elles peuvent produire le même résultat pour des éléments différents) se produisent lors des calculs de hachage, entraînant ainsi **des faux positifs**.

- Le **taux de faux positif observé** est plus **élevé** que le **taux théorique**. Il y a plusieurs raisons pour lesquelles le **taux de faux positif observé** peut être **supérieur** au **taux de faux positif théorique** :
    - Les **fonctions de hashage** utilisées ne sont pas totalement **indépendantes** et peuvent produire des **collisions**. Cela peut entraîner un **taux de faux positifs** plus élevé que le taux de faux positifs théorique.
    - Les **éléments** utilisés pour les tests peuvent avoir des caractéristiques qui les rendent plus susceptibles de produire des **faux positifs**. Par exemple, si les éléments sont choisis de manière aléatoire mais ont une distribution non uniforme.

Il est important de noter que le **taux de faux positif théorique** est une **indication** et que le taux de **faux positif observé** peut être différent en fonction des caractéristiques de l'ensemble de tests utilisés.

**Rappel:**
**Le taux de faux positif théorique est le taux de faux positifs que l'on peut attendre pour un filtre de Bloom dans des conditions idéales**. Il est calculé en utilisant la formule suivante :
taux de faux positifs théorique = (1 - e^(-kn/m))^k
où :

- **k** est le nombre de **fonctions de hashage** utilisées par le filtre
- **n** est le nombre **d'éléments présents** dans le filtre
- **m** est la **taille** du filtre

# Conclusion

Le **filtre de Bloom** est une structure de données qui permet de vérifier rapidement si un élément appartient ou non à un ensemble. Il est particulièrement utile lorsqu'il est nécessaire de **traiter de grandes quantités de données** et que la **vérification de l'appartenance** d'un élément à un ensemble doit être effectuée de manière efficace.

Il existe plusieurs implémentations possibles d'un **filtre de Bloom**, notamment en utilisant un **tableau, une ArrayList ou une LinkedList.** Chaque implémentation a ses propres avantages et inconvénients en termes de performance.

Les résultats des tests réalisés montrent que l'implémentation du **filtre de Bloom** en utilisant un **tableau** est la **plus rapide** pour les éléments testés, suivie de l'implémentation en utilisant une **ArrayList** et enfin l'implémentation en utilisant une **LinkedList**. En ce qui concerne le **taux de faux positifs**, les résultats indiquent que le taux de faux positifs des implémentations en utilisant un **tableau** est relativement proche du **taux de faux positifs théorique** pour chaque configuration de **k et n.**