Rapport de laboratoire TP2

INF2010

STRUCTURES DE DONNÉES ET ALGORITHMES

17 octobre 2022

Skander Hannachi 2088988
Rayan Fellah 2147523

Introduction

But

L’objectif de ce laboratoire était de concevoir trois tables de dispersement et de comparer leur données de performance selon des tableaux de données de type *Double.* Les trois tables de hachage implémentent chacune une fonction de dispersement différente. Soit le dispersement linéaire, quadratique et double. Leur efficacité en termes de collisions, de dispersement des données et de temps d'exécution pour l’opération d’insertion sera mise à l’épreuve par trois tableaux de données de *Double* différents. Nous observerons si la taille des données entrées ainsi que le type de dispersement utilisé influence les résultats.

Méthodologie

Afin de pouvoir analyser la performance des 3 instances de tables de hachage , nous avons eu recours à l'implémentation de diverses classes Java.

D'abord, il y a la classe statique *Prime* qui implémente 3 méthodes statiques dont *nextPrime*, *findMaxPrimeBefore* et *isPrime.* La première sert à trouver le premier nombre premier suivant un paramètre entier, la deuxième sert à trouver le premier nombre premier avant un nombre entier passé en paramètre, puis la troisième est utilisée pour déterminer si un nombre est premier ou non. Cette classe a notamment servi à l'implémentation des fonctions de dispersement dans les différentes sous-classes de *HashTable*.

Ensuite, nous avons conçu une classe InputData qui prend en paramètre une Map de type *HashMap* dans son constructeur. Elle possède également trois attributs statiques qui sont les 3 tableaux de données utilisés pour les tests. La HashMap est utilisée pour faciliter les opérations d'affichage dans le main.

Finalement, pour l'analyse de la performance, nous avons eu recours à une classe *Stats* qui prend en paramètre dans son constructeur une instance de l'interface *Hash* à analyser. Elle possède une fonction privée qui sert au comptage de la taille des amas, une fonction publique qui retourne la taille moyenne des amas ainsi que deux autres fonctions publiques qui servent à déterminer la taille du plus grand et du plus petit amas de données lors de la dispersion.

Résultats

Tableau 1 : Statistiques de computation du dispersement selon une table de hachage linéaire

Taille du tableau entré	Nombre de collisions	Taille maximale d'un amas	Taille minimale d'un amas	Taille moyenne d'un amas	Temps d'exécution approximatif (ns)
15	7	3	2	2.14	176000
60	36	8	2	2.81	129100
150	92	12	2	3.24	397700

Tableau 2 : Statistiques de computation du dispersement selon une table de hachage Quadratique

Taille du tableau entré	Nombre de collisions	Taille maximale d'un amas	Taille minimale d'un amas	Taille moyenne d'un amas	Temps d'exécution approximatif (ns)
15	8	3	2	2.14	38200
60	32	8	2	2.81	100100
150	78	7	2	2.92	397700

Tableau 3 : Statistiques de computation du dispersement selon une table de hachage double

Taille du tableau entré	Nombre de collisions	Taille maximale d'un amas	Taille minimale d'un amas	Taille moyenne d'un amas	Temps d'exécution approximatif (ns)
15	8	2	2	2	65800
60	29	6	2	2.57	100100
150	77	6	2	2.5	397700

Tableau 4 : Comparaison de l'effet du facteur de compression sur le du nombre de conflits dans une table de **hachage linéaire** (nombre de collisions enregistrées)

Taille du tableau entré	Facteur de 0.25	Facteur de 0.5	Facteur de 0.75
15	3	7	24
60	14	36	102
150	35	92	347

Tableau 5 : Comparaison de l'effet du facteur de compression sur le du nombre de conflits dans une table de **hachage quadratique** (nombre de collisions enregistrées)

Taille du tableau entré	Facteur de 0.25	Facteur de 0.5	Facteur de 0.75
15	3	8	17
60	14	32	78
150	33	78	225

Tableau 6 : Comparaison de l'effet du facteur de compression sur le du nombre de conflits dans une table de **hachage double** (nombre de collisions enregistrées)

Taille du tableau entré	Facteur de 0.25	Facteur de 0.5	Facteur de 0.75
15	3	8	20
60	17	29	73
150	36	77	223

Note* À première vue, on peut facilement observer que l'augmentation du facteur de compression entraîne une nette augmentation du nombre de collisions, et ce, pour tous les types de hachage confondus. Pour un facteur de 0.25, elles ont toutes un comportement plutôt similaire. Cependant, le hachage double semble entraîner légèrement plus de collisions que les deux autres qui elles démontrent des résultats très similaires. Un facteur de 0.25 entraîne donc moins de collision que l'implémentation standard proposée de 0.5. Pour ce qui est d'un facteur de 0.75, on observe une augmentation drastique du nombre de collision comparé au facteur de 0.5. Nous remarquons également que la table de hachage linéaire se qualifie en dernière place et démontre un perte flagrante d'efficacité dans la dispersion. En effet, pour un tableau de 150 éléments, elle compte 347 collisions tandis que le hachage double en dénombre 223 et le quadratique 225.

Réponses aux questions

Section a)

1) **Explications des résultats obtenus dans les tableaux de la section a).**

Réponse: Les résultats compilés dans les tableaux ci-dessus permettent de constater que plus la taille du tableau est élevée, plus le nombre de collisions et le temps d'exécution augmentent.
De plus, on remarque qu'à petite échelle il y a peu de différence en terme de collisions entre le dispersement selon une table de hachage linéaire et selon une table de hachage quadratique. Cependant, plus le nombre d'éléments augmentent, plus la seconde table de hachage se démarque par son efficacité par rapport à la première. Le dispersement selon une table de hachage double est clairement le plus efficace pour réduire le nombre de collisions.
Pour ce qui est du temps d'exécution approximatif, le dispersement selon une table de hachage quadratique semble être le plus rapide, tandis que le dispersement selon une table de hachage linéaire semble être le plus lent.

2) **De vos observations, est-ce que la complexité temporelle de l'insertion d'un nombre N d'éléments pour chacune des trois tables est conforme à la théorie ? Expliquez brièvement les résultats.**

Réponse : Pour une fonction d'insertion dans une table de hachage linéaire, quadratique et de hachage double, la complexité temporelle est dans le meilleur des cas de O(1), mais est dans le pire des cas de O(n). En moyenne, elle est de O(1) pour une bonne table de hachage, et de O(n) pour une table moins efficace. Dans notre cas, on constate à partir des tableaux 1,2 et 3 que le nombre de collisions est en moyenne équivalent à à peu près la moitié du nombre d'éléments présents dans le tableau entré. Ainsi, on comprend qu'en moyenne, environ la moitié des éléments ont une complexité temporelle de O(1), que l'autre moitié en a une de O(2), ce qui revient à O(1). Cela est globalement conforme à la théorie. À noter tout de même le taux de variation du nombre de conflits en fonction de la taille du tableau entré est plus élevé pour une table de hachage linéaire que pour une une table de hachage quadratique, et pour une table de hachage quadratique que pour une table de hachage double.

Section b)

3) **Que constatez-vous au niveau de la relation du Load Factor et du nombre de conflits?**

Réponse : voir Note* dans la section résultats

Section c)

4) **Quelle est la complexité asymptotique de l'opération rehash?**

Réponse : Dans l'implémentation de la méthode rehash, on utilise principalement la fonction d'insertion pour effectuer une copie des éléments actifs et non nul dans un nouveau tableau alloué plus grand. Puisque l'opération de copie en général est de complexité linéaire soit O(n), la combinaison de celle-ci avec l'opération insert(), qui elle dépend de findPos() (la fonction de dispersement) pour déterminer le placement de l'élément donne une complexité combinée de O(n * log n). En effet, findPos a une complexité de O(log n) qui sera répétée n nombre de fois par l'itération et la copie du tableau dans le rehash.

5) **Suspendre le Rehash de votre code, et répéter les mêmes étapes de la partie a). Comment a varié le temps d'exécutions et le nombre de conflits par rapport au tableau de la partie a)? Pourquoi?**

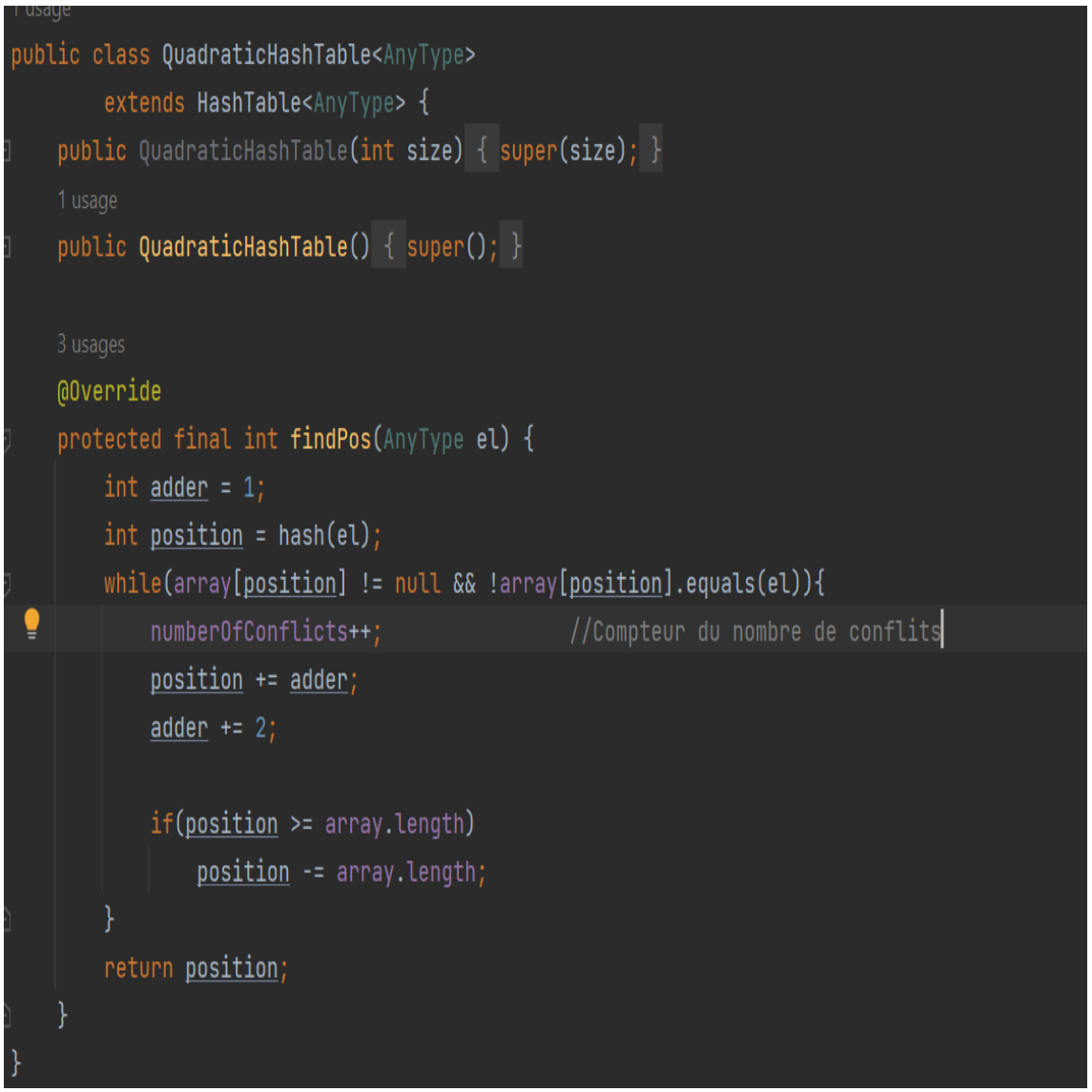
Réponse: Lorsqu'on suspend l'opération rehash, on remarque que le code n'affiche aucune valeur sur la console et ne s'exécute pas, et ce même après de longues minutes. Ce qu'il se passe est qu'on souhaite insérer un nouvel élément dans la table de hachage. Cependant, puisque la taille n'a pas été réallouée, un élément n'arrive pas à être inséré dans la table de hachage, et l'algorithme est en quelque sorte bloqué dans une série de collisions infinies. Force est de constater que par rapport à la partie a), le nombre de conflits ainsi que le temps d'exécution ont énormément augmenté, à un tel point que le code ne s'exécute même plus.

Annexes

Annexe 1 : classes et implémentations

```java
public class LinearHashTable<AnyType> extends HashTable<AnyType> {

    1 usage
    LinearHashTable() { super(); }

    LinearHashTable(int size) { super(size); }

    3 usages
    protected final int findPos(AnyType el) {
        int currentPos = hash(el);

        while(array[currentPos] != null &&
                !array[currentPos].element_.equals(el)) {
            numberOfConflicts++;                        //Compteur du nombre de conflits
            currentPos++;
            currentPos %= array.length;
            if(currentPos >= array.length) {
                currentPos -= array.length;
            }
        }

        return currentPos;
    }
```

Figure 1: *Classe "LinearHashTable" à la fin de la partie a)*.



```
1 usage
public class QuadraticHashTable<AnyType>
        extends HashTable<AnyType> {
    public QuadraticHashTable(int size) { super(size); }
    1 usage
    public QuadraticHashTable() { super(); }

    3 usages
    @Override
    protected final int findPos(AnyType el) {
        int adder = 1;
        int position = hash(el);
        while(array[position] != null && !array[position].equals(el)){
            numberOfConflicts++;                //Compteur du nombre de conflits
            position += adder;
            adder += 2;

            if(position >= array.length)
                position -= array.length;
        }
        return position;
    }
}
```

Figure 2: *Classe "QuadraticHashTable" à la fin de la section a)*

```
1 usage
public class DoubleHashedHashTable<AnyType>
        extends HashTable<AnyType>{
    1 usage
    public DoubleHashedHashTable() { super(); }
    public DoubleHashedHashTable(int size) { super(size); }

    1 usage
    private int secondHash(AnyType x ){
        return Prime.findMaxPrimeBefore(array.length) - (x.hashCode() %
                Prime.findMaxPrimeBefore(array.length));
    }
```

```
    @Override
    protected final int findPos(AnyType el) {

        int position = hash(el);
        int offset = 1;

        while(array[position] != null && !array[position].element_.equals(el) )
        {
            numberOfConflicts++;
            position += offset++ * secondHash(el) % array.length;
            if(position >= array.length)
                position -= array.length;
        }
        return position;
    }
}
```

Figures 3-4: *Classe "DoubleHashedHashTable" à la fin de la section a)*

Annexe 2 : Résultats d'affichages

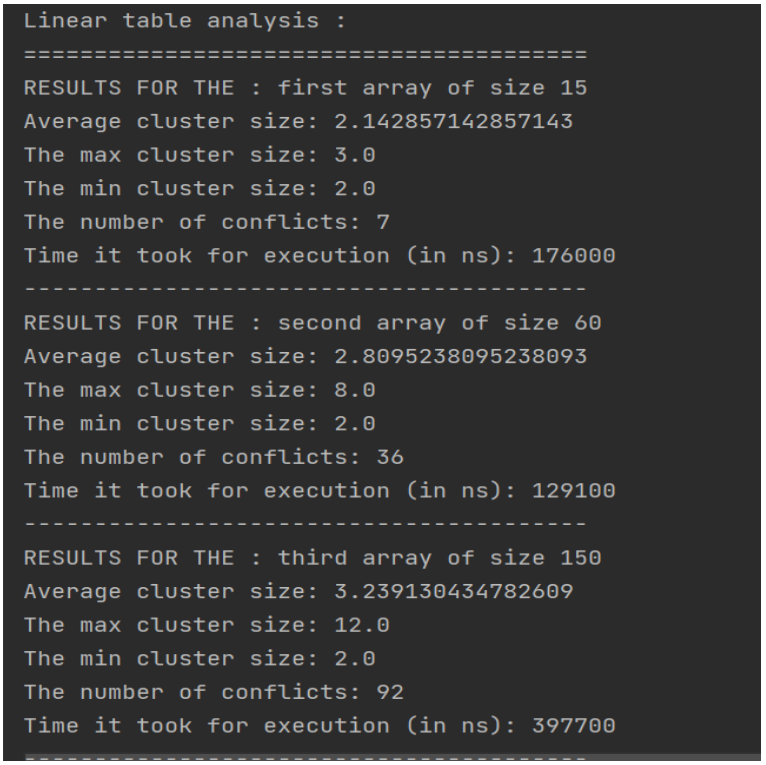


Figure 5: *Affichage de la classe "LinearHashTable" à la fin de la partie a).*

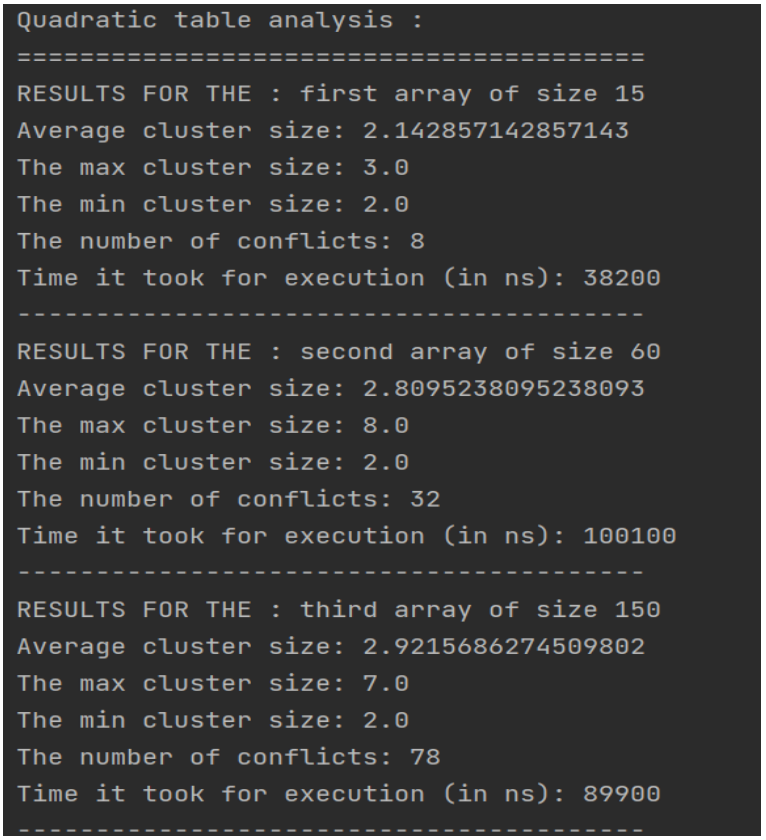


Figure 6: *Affichage de la classe "QuadraticHashTable" à la fin de la partie a).*

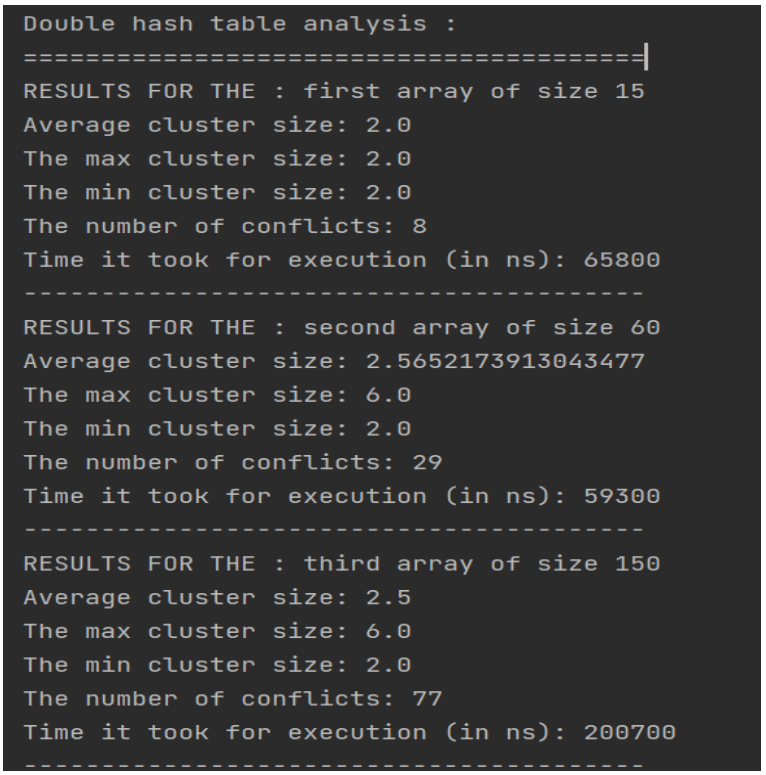


Figure 7: *Affichage de la classe" DoubleHashedTable" à la fin de la partie a).*

Annexe 3 : Modifications du rehash

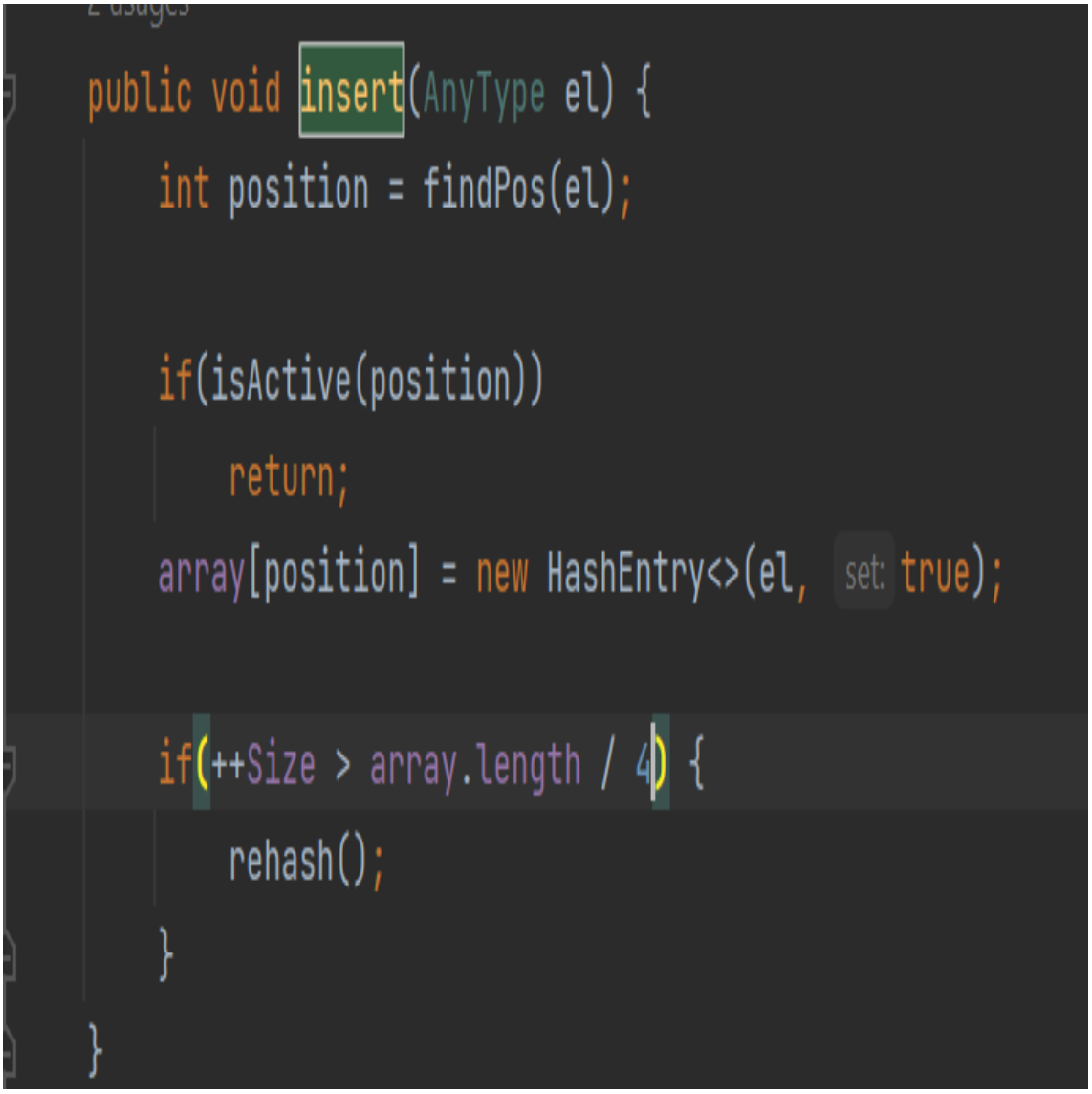


Figure 8: Modification de la méthode “insert()” afin d’avoir un facteur de compression de 0.25

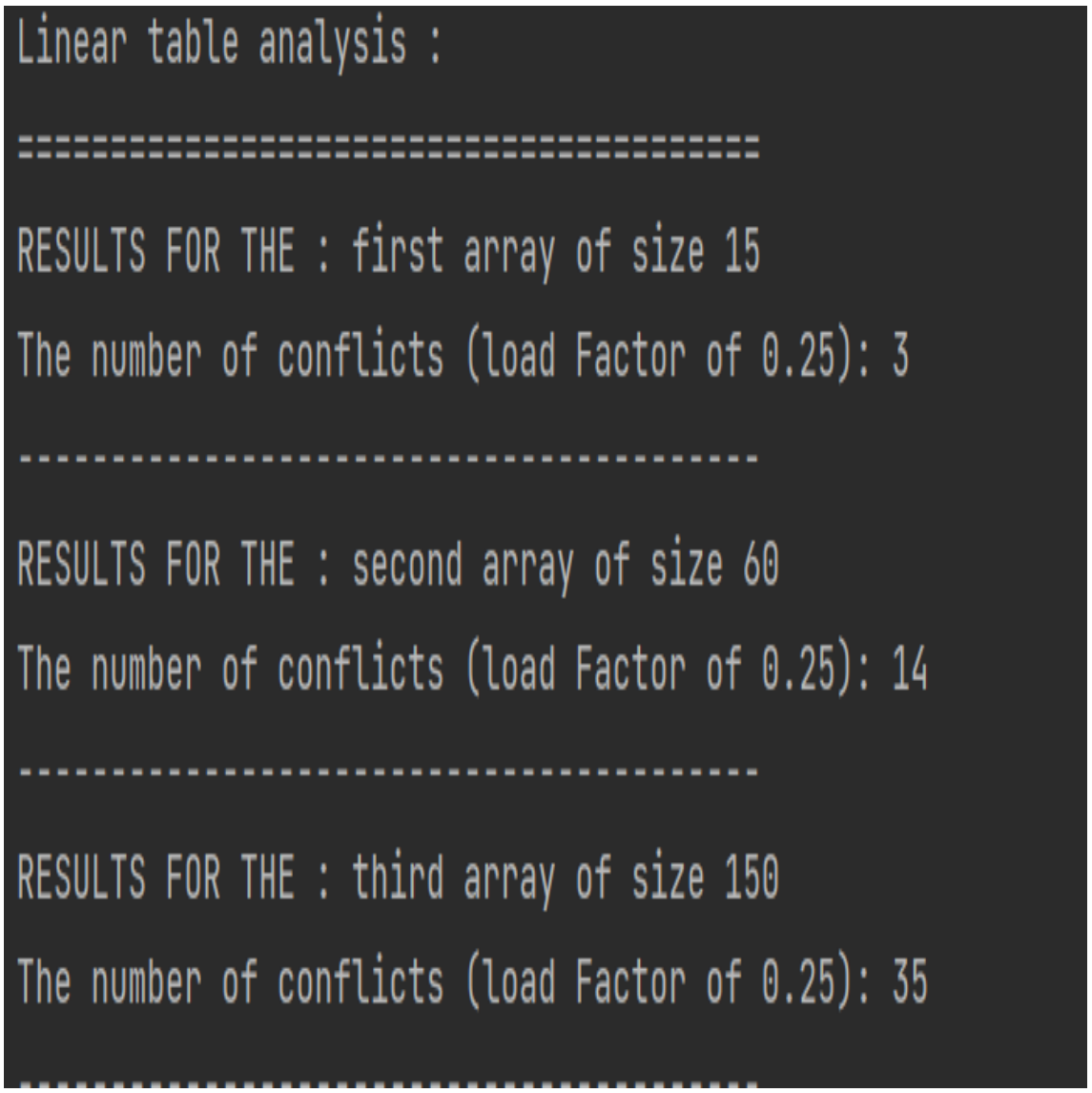


Figure 9: Affichage du nombre de collisions pour un facteur de compression de 0.25 de la classe “LinearHashTable”

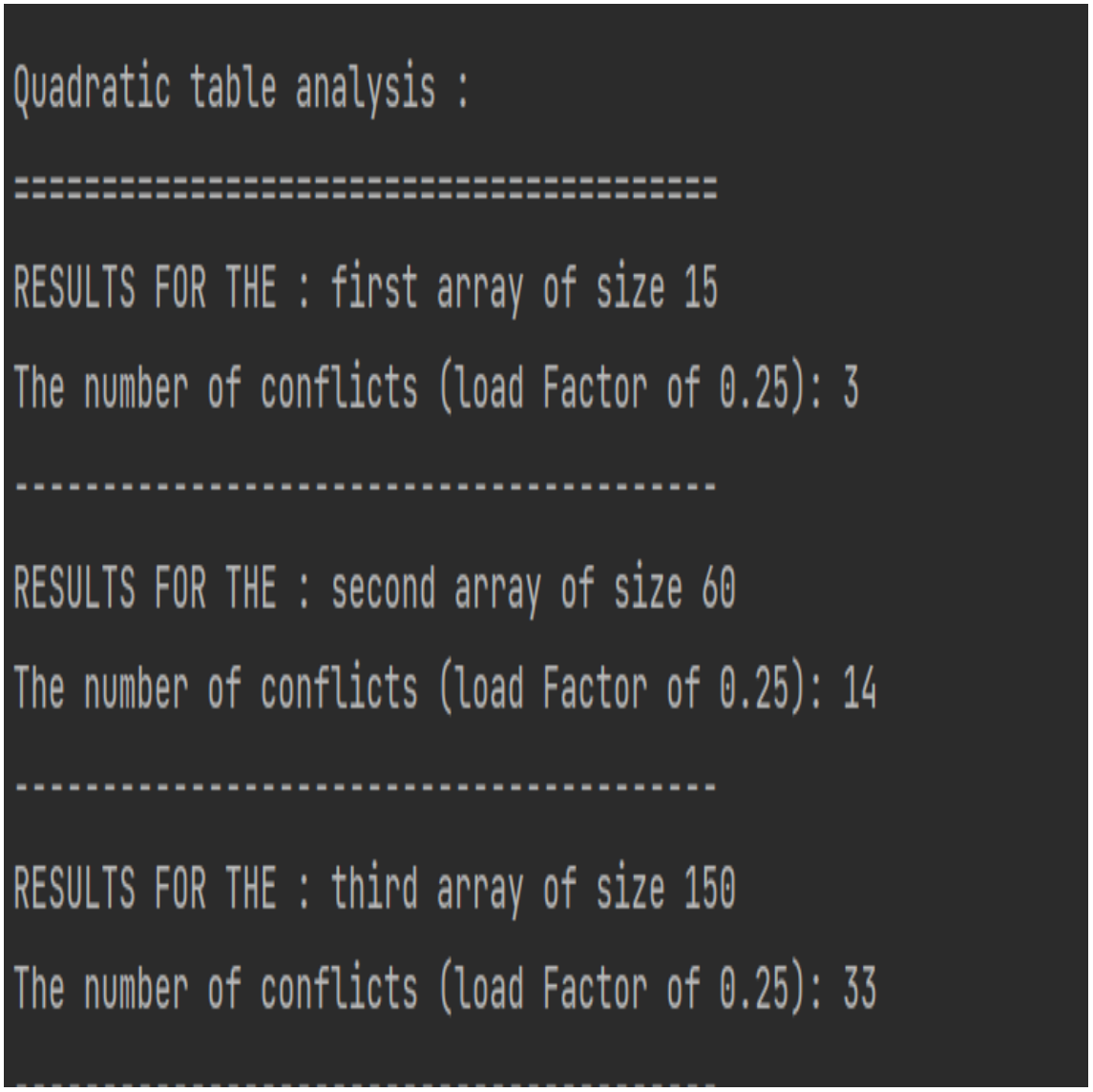


Figure 10: Affichage du nombre de collisions pour un facteur de compression de 0.25 classe “QuadraticHashTable”

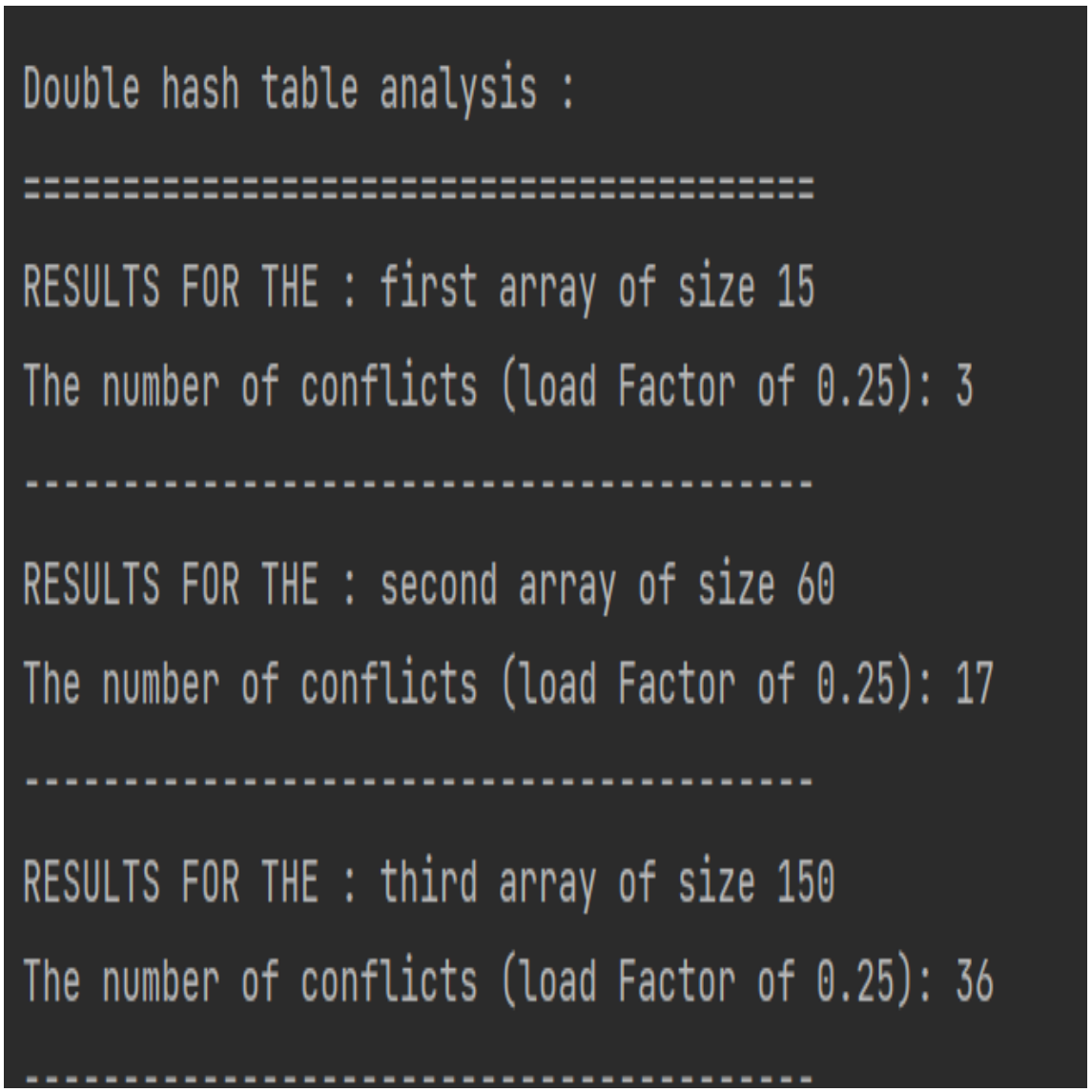


Figure 11: Affichage du nombre de collisions pour un facteur de compression de 0.25 de la classe “DoubleHashedHashTable”

Figure 12: Modification de la méthode “insert()” afin d’avoir un facteur de compression de 0.75

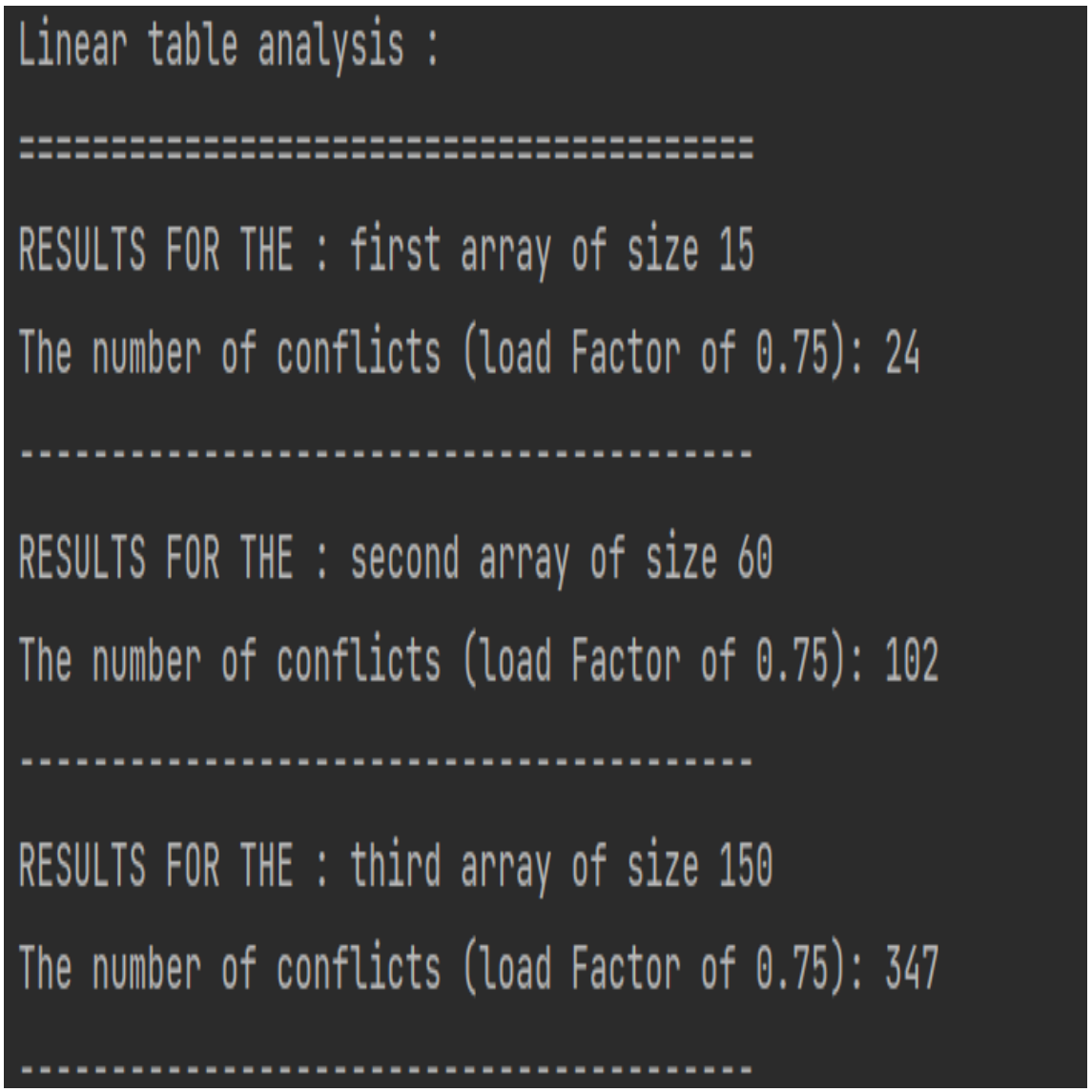


Figure 13: Affichage du nombre de collisions pour un facteur de compression de 0.75 de la classe “LinearHashTable”

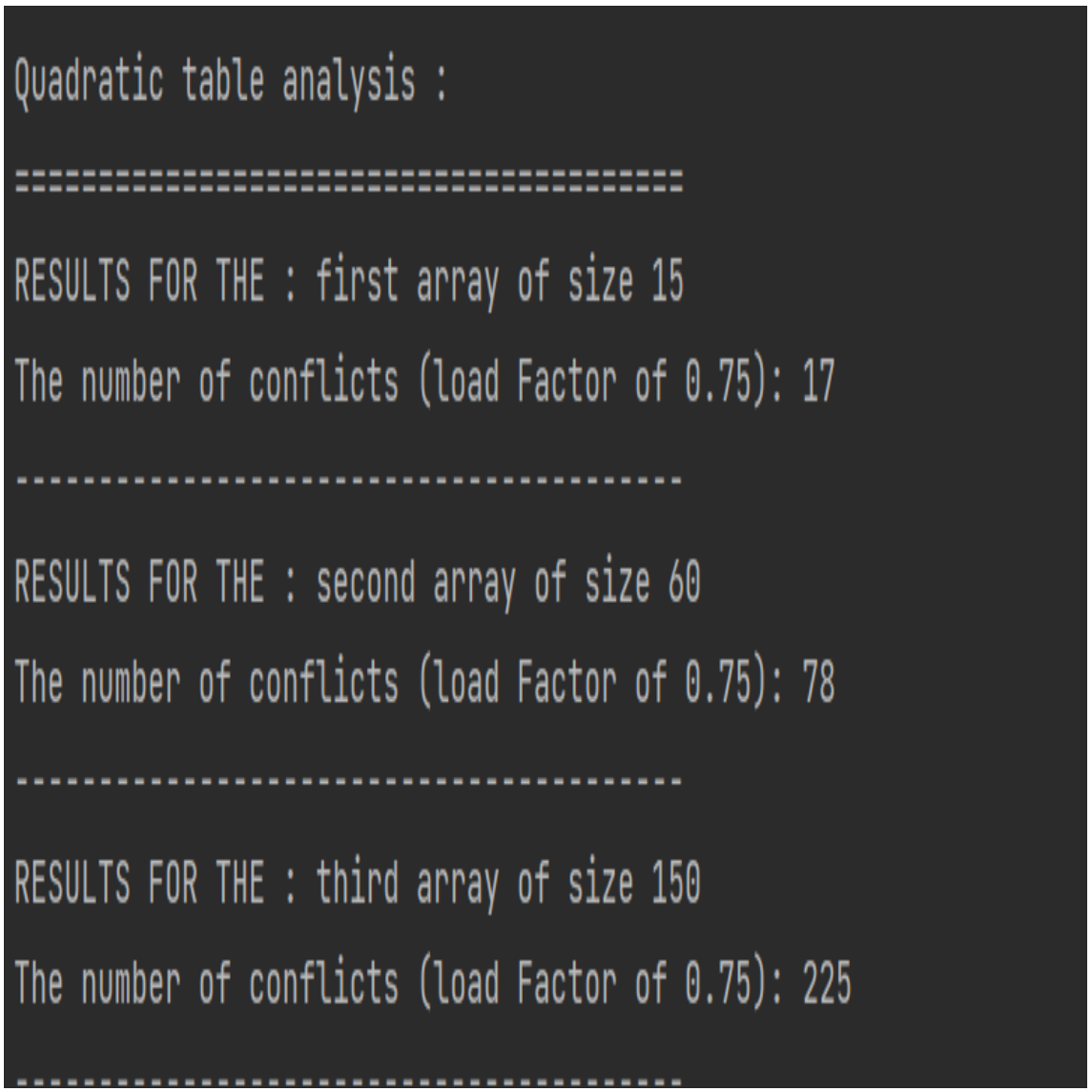


Figure 14: Affichage du nombre de collisions pour un facteur de compression de 0.75 classe “QuadraticHashTable”

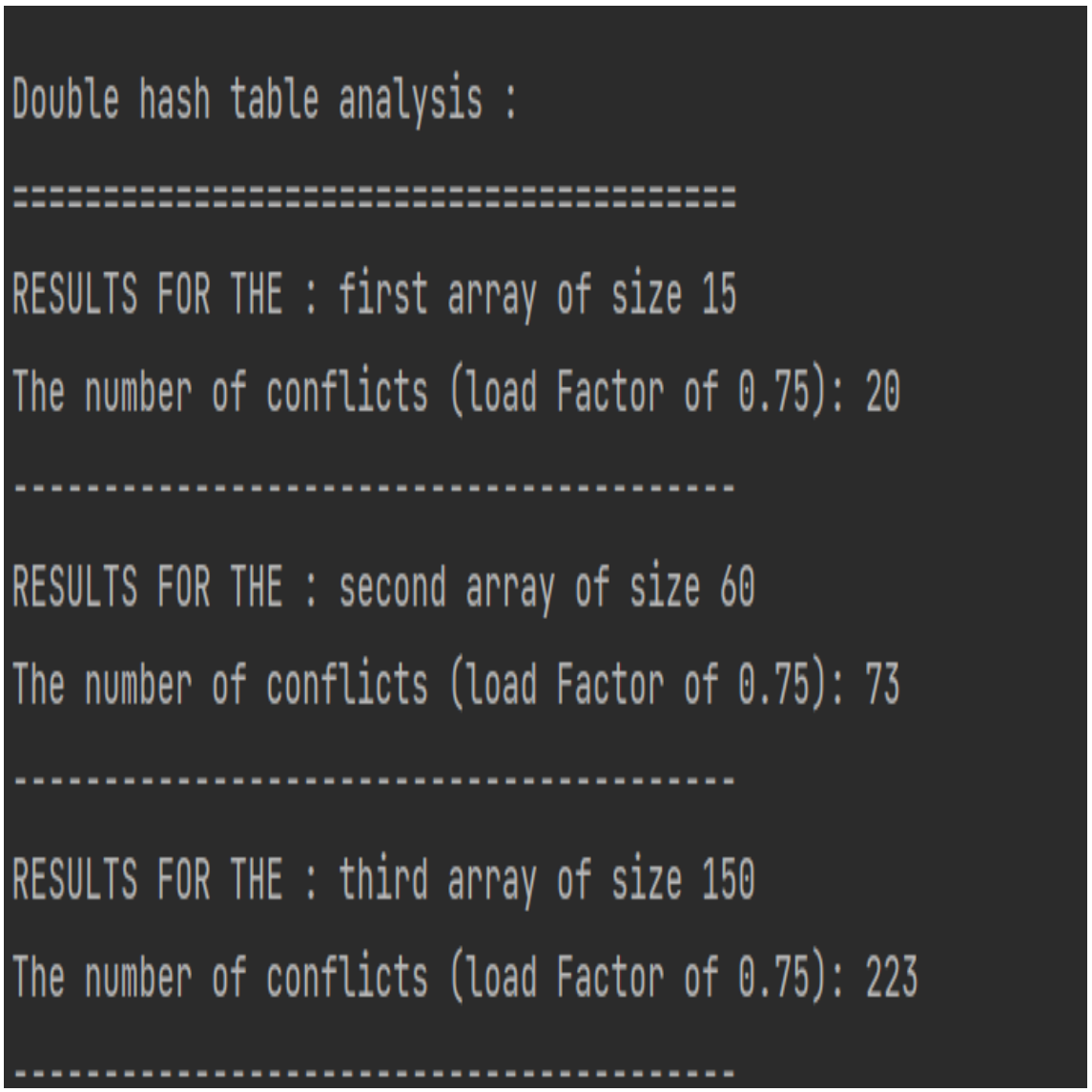
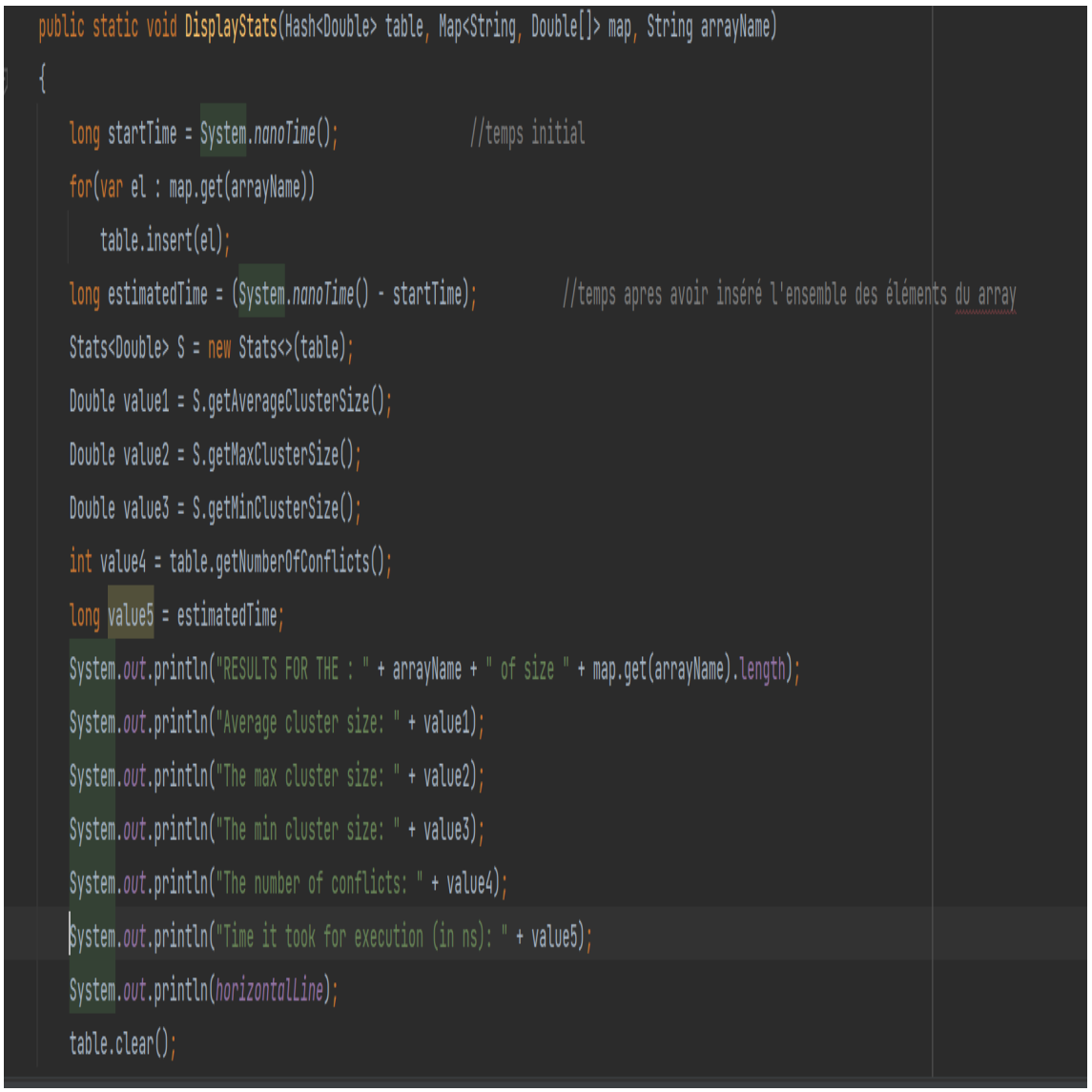


Figure 15: Affichage du nombre de collisions pour un facteur de compression de 0.75 de la classe “DoubleHashedHashTable”



```
public static void DisplayStats(Hash<Double> table, Map<String, Double[]> map, String arrayName)
{
    long startTime = System.nanoTime();              //temps initial
    for(var el : map.get(arrayName))
        table.insert(el);
    long estimatedTime = (System.nanoTime() - startTime);         //temps apres avoir inséré l'ensemble des éléments du array
    Stats<Double> S = new Stats<>(table);
    Double value1 = S.getAverageClusterSize();
    Double value2 = S.getMaxClusterSize();
    Double value3 = S.getMinClusterSize();
    int value4 = table.getNumberOfConflicts();
    long value5 = estimatedTime;
    System.out.println("RESULTS FOR THE : " + arrayName + " of size " + map.get(arrayName).length);
    System.out.println("Average cluster size: " + value1);
    System.out.println("The max cluster size: " + value2);
    System.out.println("The min cluster size: " + value3);
    System.out.println("The number of conflicts: " + value4);
    System.out.println("Time it took for execution (in ns): " + value5);
    System.out.println(horizontalLine);
    table.clear();
```

Figure 16: Fonction DisplayStats() modifiée afin d'obtenir le temps pris pour l'insertion des éléments de chaque tableau dans la table de hachage

Figure 17: Méthode insert() modifiée dans le but de suspendre l'opération de Rehash pour la partie c).

```java
public interface Hash<AnyType> {
    2 usages  1 implementation
    void insert(AnyType el);
    1 implementation
    void remove(AnyType el);

    1 implementation
    boolean contains(AnyType el);
    2 usages  1 implementation
    void clear();
    1 implementation
    int getSize();

    2 usages  1 implementation
    HashEntry<AnyType>[] getArray();
    1 usage  1 implementation
    public int getNumberOfConflicts();          //getter du nombre de conflits
}
```

Figure 18: Interface Hash à la fin de la partie a)