

Structures de Données Avancées

Rapport TP3

Tas binaire et Tas binomial

Réalisé par :

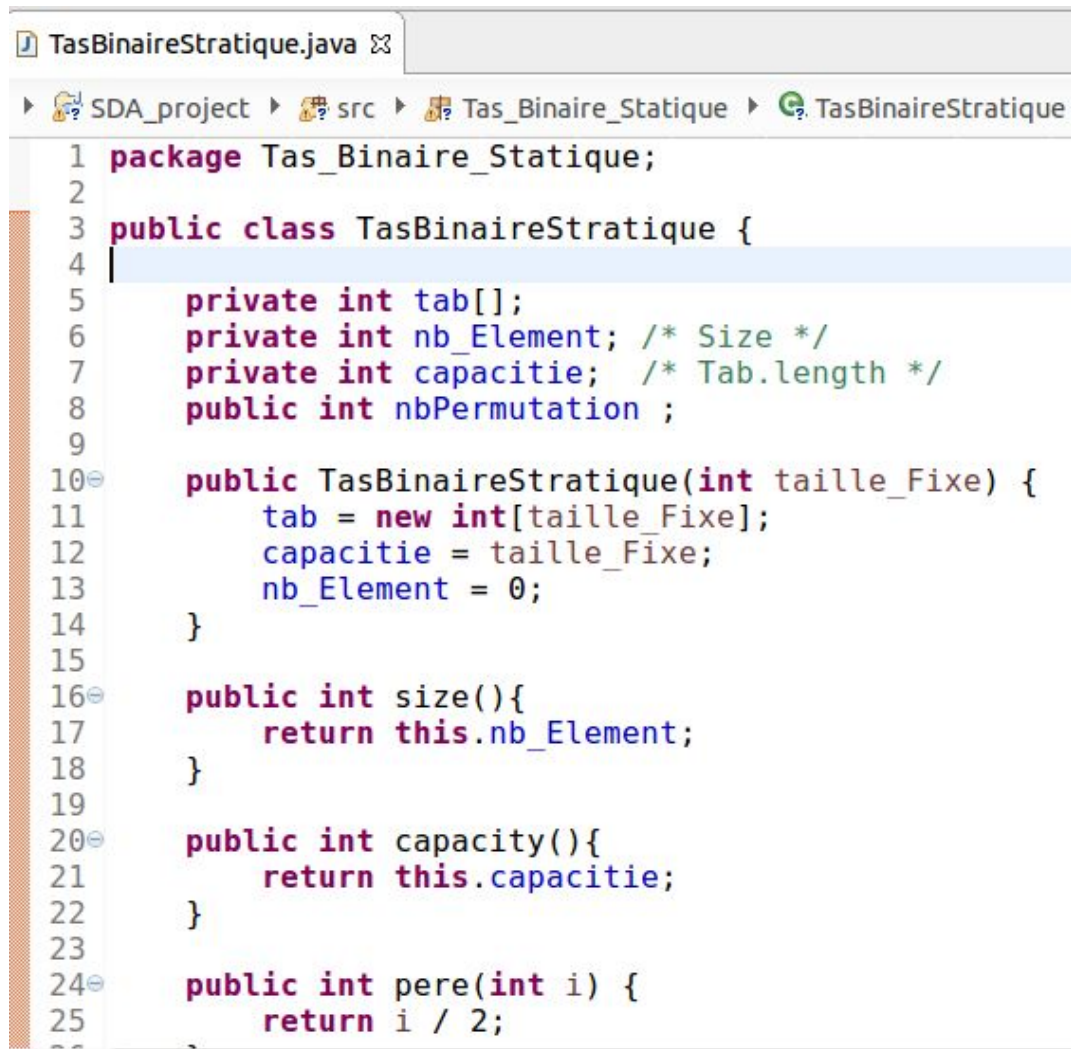
Aoudjehane Sarah

KOUACHI Abdeldjalil

I. Tas Binaire

1) Développement d'une structure/classe de tas binaire dans laquelle le tableau servant à stocker les clés est de taille fixe.

Voir le pseudo Code que nous avons implémenté avec les tableaux statiques :



```
1 package Tas_Binaire_Statique;
2
3 public class TasBinaireStratique {
4
5     private int tab[];
6     private int nb_Element; /* Size */
7     private int capacitie; /* Tab.length */
8     public int nbPermutation ;
9
10    public TasBinaireStratique(int taille_Fixe) {
11        tab = new int[taille_Fixe];
12        capacitie = taille_Fixe;
13        nb_Element = 0;
14    }
15
16    public int size(){
17        return this.nb_Element;
18    }
19
20    public int capacity(){
21        return this.capacitie;
22    }
23
24    public int pere(int i) {
25        return i / 2;
26    }
27 }
```

- Tab [] Tableau d'entiers
- la taille maximal de du Tas Binaire est déclaré par l'utilisateur lors de la création du Tas (Au niveau du constructeur)
- **Nb_Element** représente le Size ou plutôt le Nombre d'élément contenus dans le Tas Binaire.
- **capacite** représente la Taille fixe du tableau.

Les méthodes nécessaires implémentées :

```
public int pere(int i) {
    return i / 2;
}

public int gauche(int i) {
    return 2 * i;
}

public int droite(int i) {
    return (2 * i) + 1;
}

public void entasser(int i) {}

public void permuter(int x, int y) {
    nbPermutation = nbPermutation + 1 ;
    int tmp = this.tab[x];
    this.tab[x] = this.tab[y];
    this.tab[y] = tmp;
}

public int minimum(int[] tab) {
    return this.tab[0];
}

public int extraire_Min(int tab[]) throws DisplayException {

public void diminuerCle(int i, int k) throws DisplayException {

public void inserer(int k) throws DisplayException {
public void afficheTab() {
public void inserer(int k) throws DisplayException {
```

Nous nous sommes inspiré lors du développement des méthodes ci-dessus du pseudo Algorithme fourni par vous dans le cour.

Nous avons créés la Classe **DisplayException** dans le but est de renvoyer une exception Si l'utilisateur tente d'ajouter une valeur dans un tas plein.

```

DisplayException.java
SDA_project > src > Tas_Binaire_Statique > DisplayException
1 package Tas_Binaire_Statique;
2
3 public class DisplayException extends Exception{
4
5     public DisplayException(String message) {
6         super(message);
7     }
8 }

```

2) des expériences sur l'efficacité en temps et en mémoire de cette structure

cas 1 : Insertion par ordre ordre croissant :

```

Main_TBStatiqueACS.java
SDA_project > src > Tas_Binaire_Statique > Main_TBStatiqueACS
1 package Tas_Binaire_Statique;
2 import analysis.Analyzer;
3
4
5
6
7 public class Main_TBStatiqueACS {
8
9     public static void main(String[] args) throws DisplayException {
10
11         TasBinaireStratique tas_binaire = new TasBinaireStratique(10000);
12
13         // Analyse du temps pris par les opérations.
14         Analyzer time_analysis = new Analyzer();
15         // Analyse du nombre de copies faites par les opérations.
16         Analyzer copy_analysis = new Analyzer();
17         // Analyse de l'espace mémoire inutilisé.
18         Analyzer memory_analysis = new Analyzer();
19
20         long before = 0, after = 0;
21
22         for(int i = 0; i < 10000; i++)
23         {
24             before = System.nanoTime();
25             tas_binaire.inserer(i);
26             after = System.nanoTime();
27         }
28     }
29 }

```

```

Total cost : 16743167
Average cost : 1674.3167
Variance :220061273785378.58810111
Standard deviation :14834462.3692730627954006195068359375
le Tas binaire Statique est :

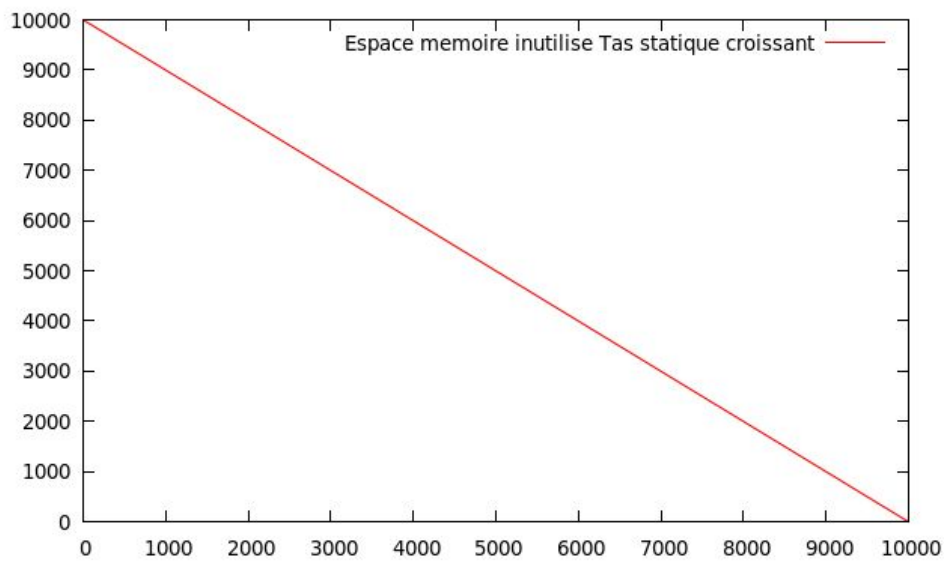
```

```

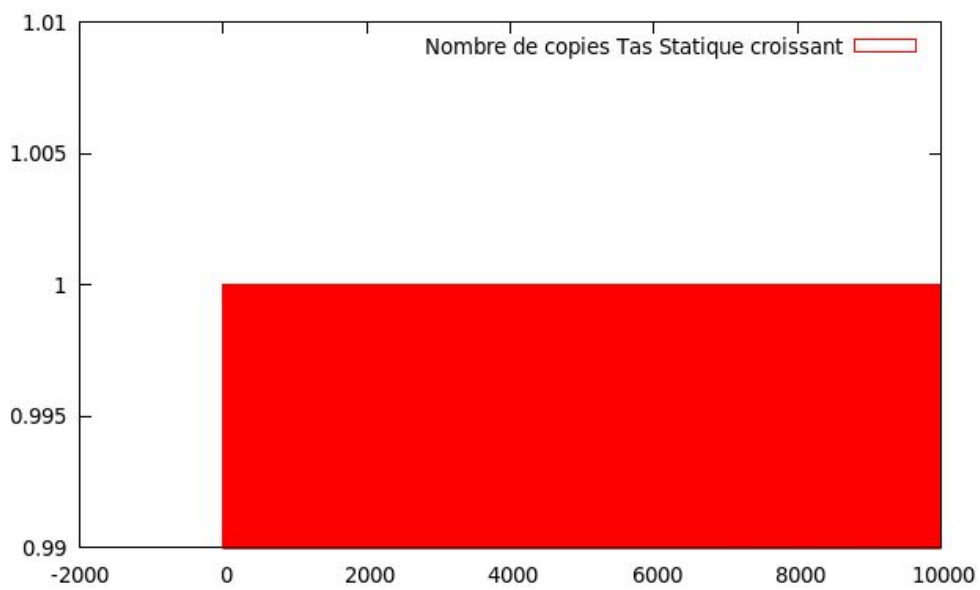
[ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9

```

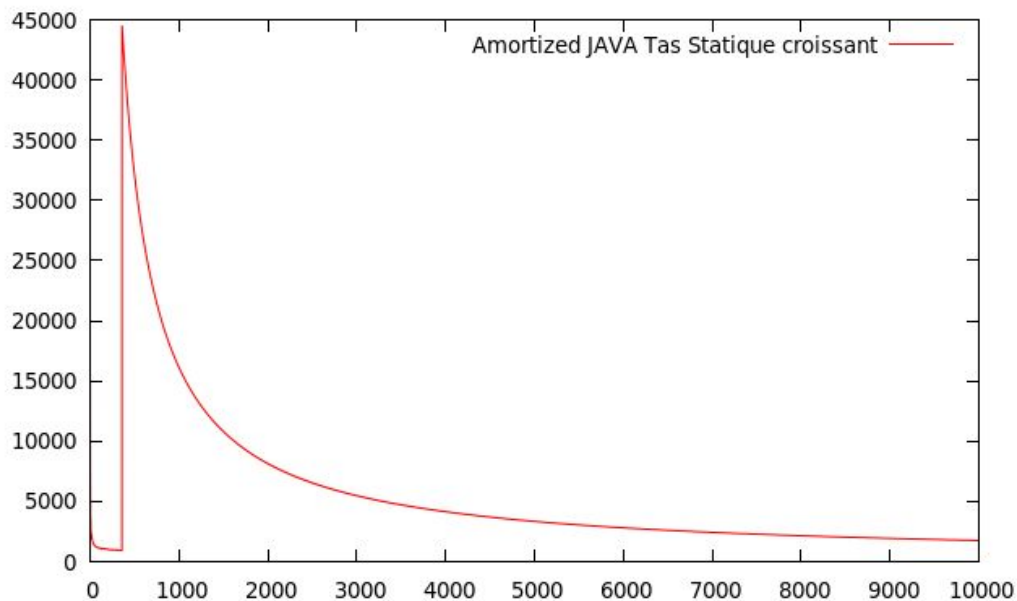
capture espace mémoire inutilisée



capture nombre de copie :



capture Coût amortie :



Insertion par Ordre Décroissant :

```

Main_TBStatiqueDESC.java
SDA_project ▸ src ▸ Tas_Binaire_Statique ▸ Main_TBStatiqueDESC ▸
1 package Tas_Binaire_Statique;
2
3 import analysis.Analyzer;
4
5 public class Main_TBStatiqueDESC {
6     public static void main(String[] args) throws DisplayException {
7
8         TasBinaireStratique tas_binaire = new TasBinaireStratique(10000);
9
10        // Analyse du temps pris par les opérations.
11        Analyzer time_analysis = new Analyzer();
12        // Analyse du nombre de copies faites par les opérations.
13        Analyzer copy_analysis = new Analyzer();
14        // Analyse de l'espace mémoire inutilisé.
15        Analyzer memory_analysis = new Analyzer();
16        long before = 0, after = 0;
17
18
19        for(int i = 10000; i > 0; i--)
20        {
21            before = System.nanoTime();
22            tas_binaire.inserer(i);
23            after = System.nanoTime();
24        }

```

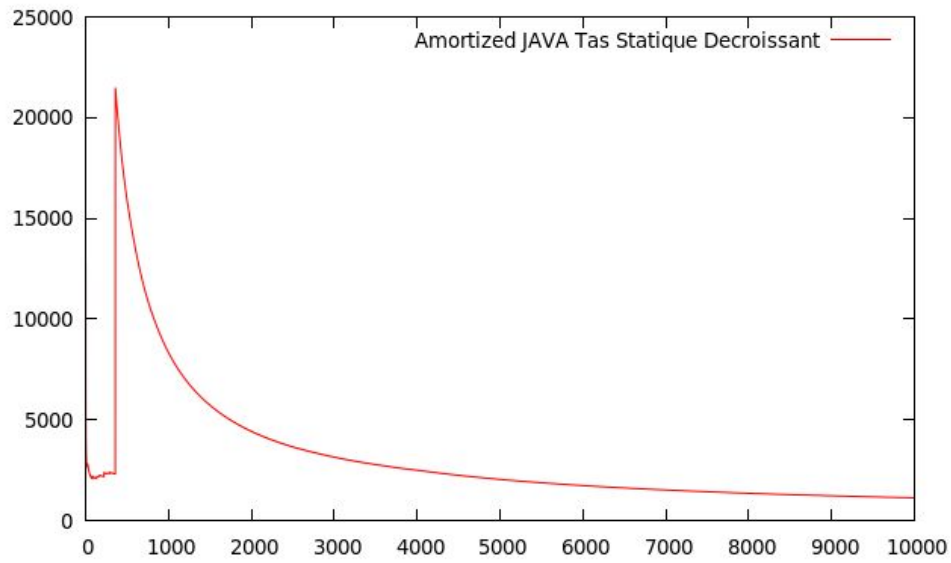
```

Total cost : 11301942
Average cost : 1130.1942
Variance :47450384981467.07028636
Standard deviation :6888423.983863585628569126129150390625
le Tas binaire Statique est :

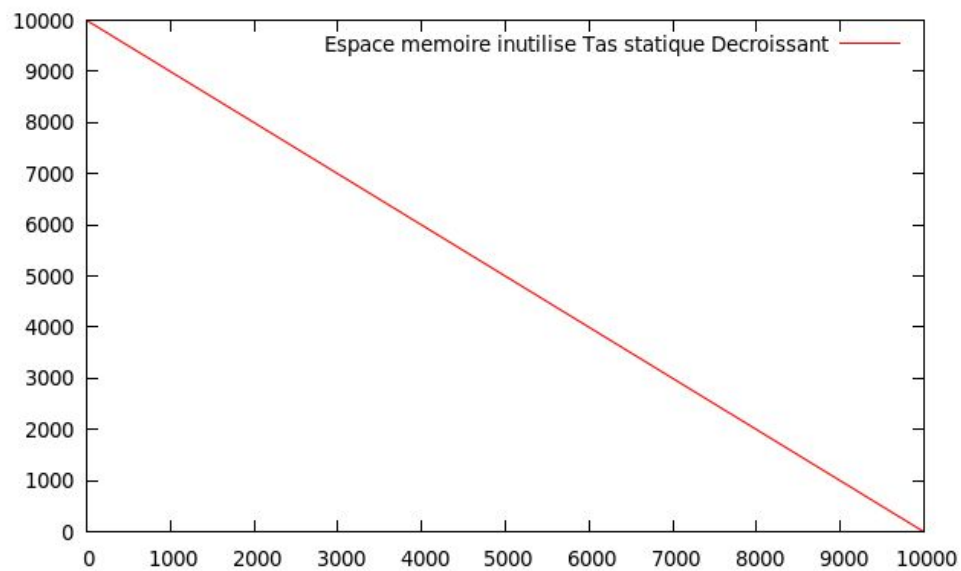
[ 1 , 2 , 3 , 1811 , 4 , 3860 , 2836 , 1812 , 789

```

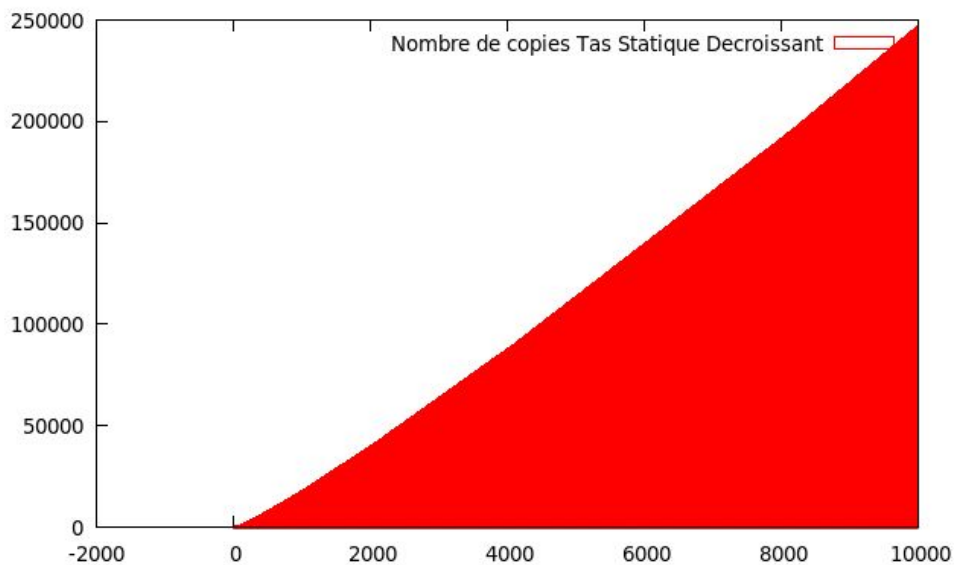
le Coût Amortie :



L'espace mémoire inutilisé :



le nombre de copie :



Conclusion :

- Lorsqu'on ajoute les clefs dans l'ordre croissant ces clés seront ajoutées en dernière position ce qui fait une complexité en temps en $O(1)$.
On constate que l'espace inutilisé au départ est élevé puis il diminue au fur à mesure qu'on insère les clefs de manière linéaire
- Lorsqu'on ajoute les clefs dans l'ordre décroissant on remarque que le coût amorti augmente légèrement par rapport au premier cas car lorsqu'on ajoute les éléments on le fait au début donc ce qui est coûteux en temps on est obligé de décaler tous les éléments déjà insérés $O(n)$. On voit bien que le nombre de copies croît de manière exponentielle contrairement à l'ordre croissant où le nombre de copie est borné par 1.

Les Tas Binaire Opération Ajout---Suppression :

3) Les Tas Binaires Dynamique :

L'implémentation : Une fois que nous avons pu implémenter le Tas Binaire en Statique, il suffisait juste de remplacer le Tab par une ArrayList<Integer>.

```
TasBinaireDynamique.java
SDA_project > src > Tas_Binaire_Dynamique > TasBinaireDynamique
1 package Tas_Binaire_Dynamique;
2
3 import java.util.ArrayList;
4
5 public class TasBinaireDynamique {
6
7     public ArrayList<Integer> tab = new ArrayList<Integer>();
8     public int nbPermutation;
9
10    public TasBinaireDynamique() {
11    }
12
13    public int pere(int i) {
14        return i / 2;
15    }
16
17    public int gauche(int i) {
18        return 2 * i;
19    }
20
21    public int droite(int i) {
22        return (2 * i) + 1;
23    }
24 }
```

Le coût amorti des opérations d'ajout et de suppression va changer car avec la table dynamique on réalloue a chaque fois de l'espace mémoire et on recopie les valeurs de l'ancien tableau vers le nouveau.

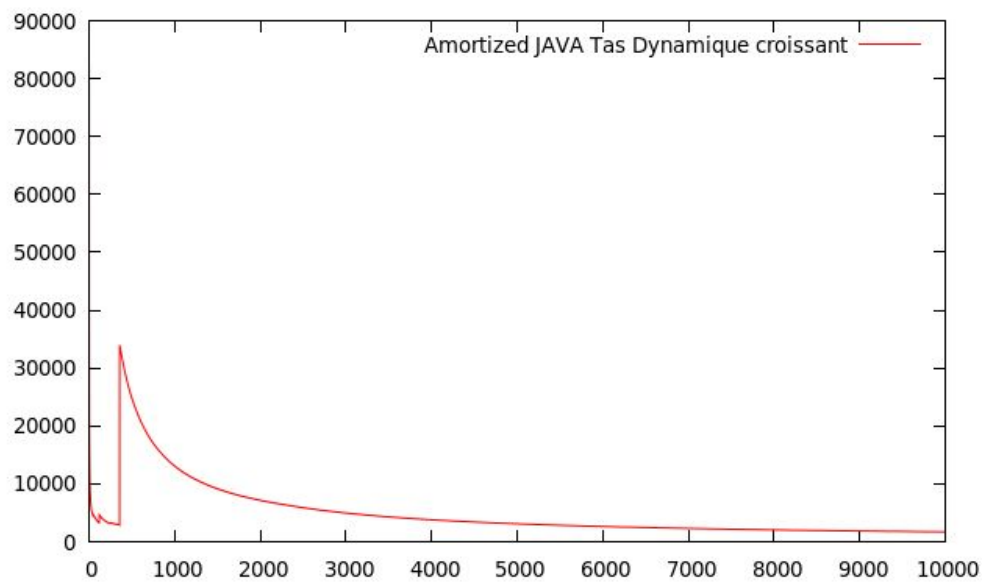
Cas 1 :Insertion par Ordre Croissant

```
Main_TBDDynamiqueASC.java
SDA_project ▸ src ▸ Tas_Binaire_Dynamique ▸ Main_TBDDynamiqueASC ▸
1 package Tas_Binaire_Dynamique;
2 import analysis.Analyzer;
3
4 public class Main_TBDDynamiqueASC {
5
6     public static void main(String[] args) throws DisplayException {
7
8         TasBinaireDynamique tasBinaireDynamique = new TasBinaireDynamique();
9
10        // Analyse du temps pris par les opérations.
11        Analyzer time_analysis = new Analyzer();
12        // Analyse du nombre de copies faites par les opérations.
13        Analyzer copy_analysis = new Analyzer();
14
15        // Analyse de l'espace mémoire inutilisé.
16        Analyzer memory_analysis = new Analyzer();
17        long before, after;
18
19
20        // insert number increased
21        for (int i = 0; i < 10000; i++) {
22            before = System.nanoTime();
23            tasBinaireDynamique.inserer(i);
24            after = System.nanoTime();
25        }
26    }
27 }
```

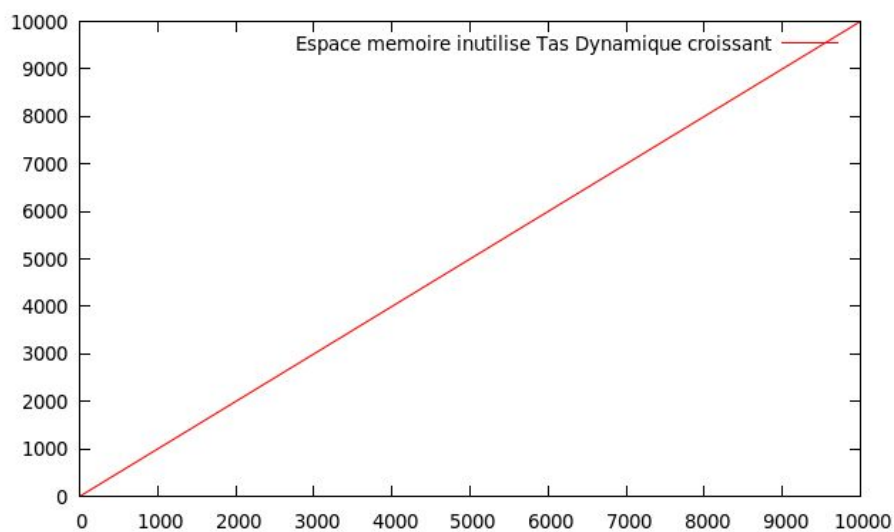
```
<terminated> Main_TBDDynamiqueASC [Java Application] /usr/lib/jvm/
Total cost : 17199741
Average cost : 1719.9741
Variance :123710632254732.09532919
Standard deviation :11122528.14133244194090366:
le Tas binaire est Dynamique est :

[ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7
```

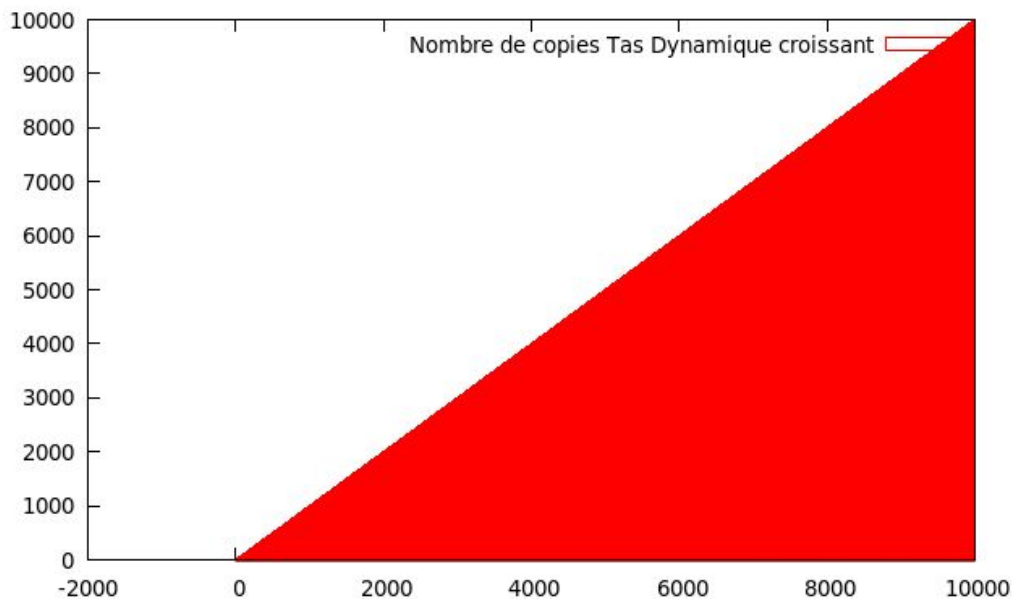
Coût Amorti :



Espace Mémoire Inutilisé :



Nombre de Copies :



cas 2 : Insertion par Ordre décroissant :

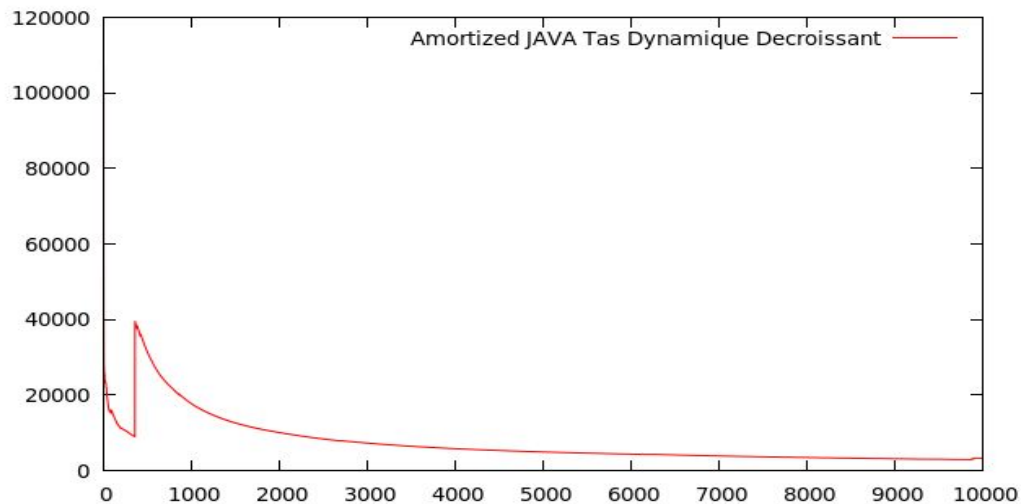
```

Main_TBDDynamiqueDESC.java
SDA_project ▸ src ▸ Tas_Binaire_Dynamique ▸ Main_TBDDynamiqueDESC ▸
1 package Tas_Binaire_Dynamique;
2 import analysis.Analyzer;
3
4 public class Main_TBDDynamiqueDESC {
5     public static void main(String[] args) throws DisplayException {
6
7         TasBinaireDynamique tasBinaireDynamique = new TasBinaireDynamique();
8
9         // Analyse du temps pris par les opérations.
10        Analyzer time_analysis = new Analyzer();
11        // Analyse du nombre de copies faites par les opérations.
12        Analyzer copy_analysis = new Analyzer();
13        // Analyse de l'espace mémoire inutilisé.
14        Analyzer memory_analysis = new Analyzer();
15        long before, after;
16
17        // insert number increased
18        for (int i = 10000; i > 0; i--) {
19            before = System.nanoTime();
20            tasBinaireDynamique.inserer(i);
21            after = System.nanoTime();
22
23            // Enregistrement du temps pris par l'opération
24            time_analysis.append(after - before);
25

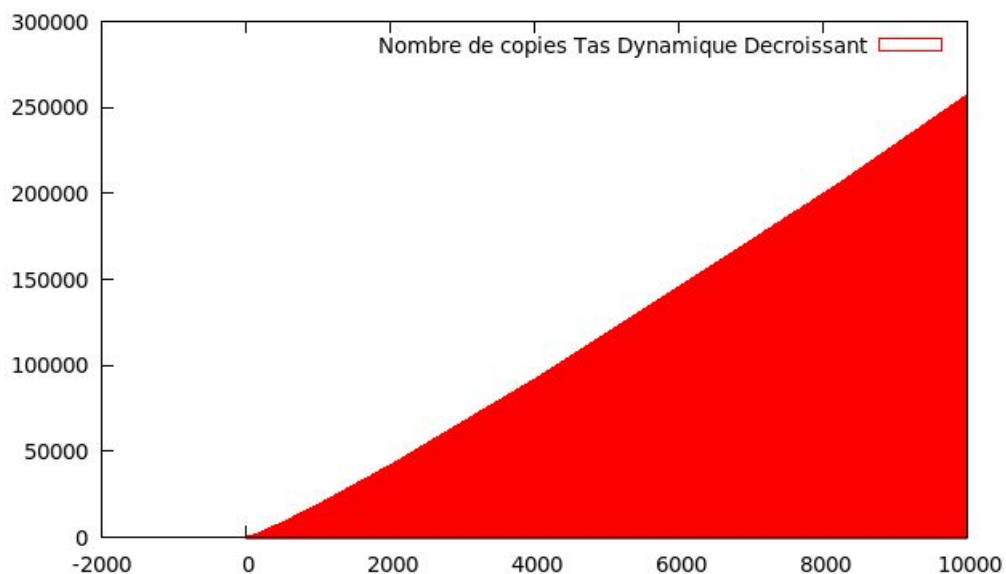
```

```
Total cost : 32956182
Average cost : 3295.6182
Variance :126421883628358.67982876
Standard deviation :11243748.646619537845253944396
le Tas binaire est Dynamique est :
[ 1 , 2 , 3 , 1811 , 4 , 3860 , 2836
```

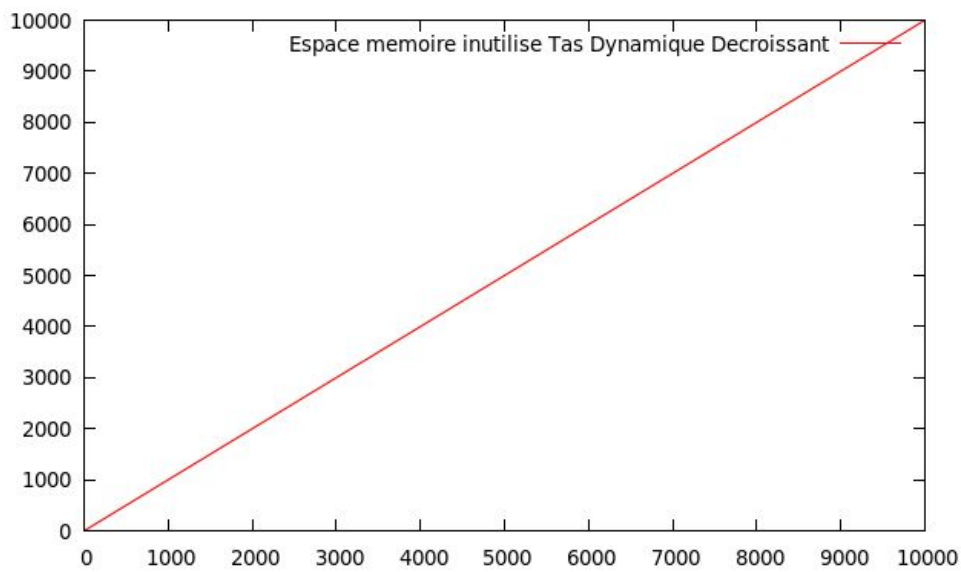
Coût Amortie :



Nombres de copies



Espace Mémoire non utilisées :



Conclusion

Dans les deux cas (ordre croissant et décroissant) on remarque que le nombre de copies et l'espace inutilisé est le même, ils augmentent au fur et à mesure qu'on ajoute des éléments. Une légère différence sur le coût amorti qui diminue dans le cas où l'ordre est décroissant.

II. Tas binomial