RAPPORT STRUCTURE DE DONNEES AVANCEES

TP3 TABLE BINAIRE

Réalisé par :

LE Minh Hao

KEITA Aïssata

Table des matières

Tas binaire

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

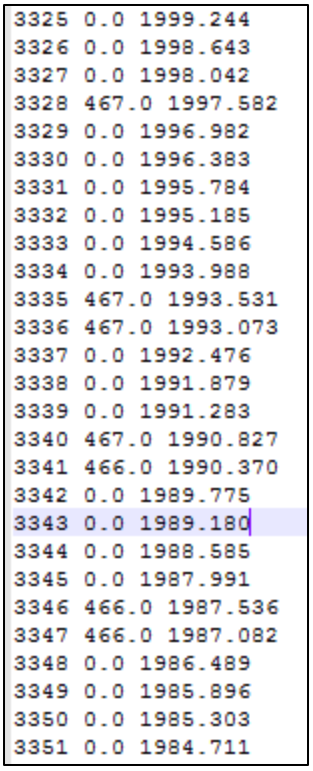
Cette taille est fixée à la création du tas. Si l'utilisateur tente d'ajouter une valeur dans un tas plein, un programme en C affichera une erreur et un programme en C++ ou Java jettera une exception. La structure de tas permettra au moins d'ajouter une clé, et d'extraire la plus petite clé contenu dans le tas.

On développe une classe de tas binaire avec un tableau fixe. En d'autres termes, l'allocation de mémoire est allouée une seule fois et cette taille ne changera pas dans le futur. L'espace mémoire inutilisé est diminué avec le temps, bien sûr.

Cette classe permet d'ajouter une clé et d'extraire la plus petite clé contenu dans le tas. Ce tas binaire a la propriété **min-heap** (la plus petite clé est la racine de ce tas binaire). Dans cette classe, on va compter une affectation de tas binaire **(assignment statement)** comme une opération de copier. Par exemple, la methode **« swap »** est compté comme deux opérations de copier :

```
107    private void swap(int index1, int index2) {
108        countSwap++;
109        int temp = data[index1];
110        data[index1] = data[index2];
111        data[index2] = temp;
112    }
```

De plus, il y a des valeurs 0 dans le fichier de cout réel en temps puisque l'opération est exécutée très vite (before = after). En d'autres termes, l'ordinateur n'a peut-être pas une résolution d'horloge suffisante, il peut donc y avoir une bonne partie du temps pendant laquelle **nanoTime** renvoie le même nombre.

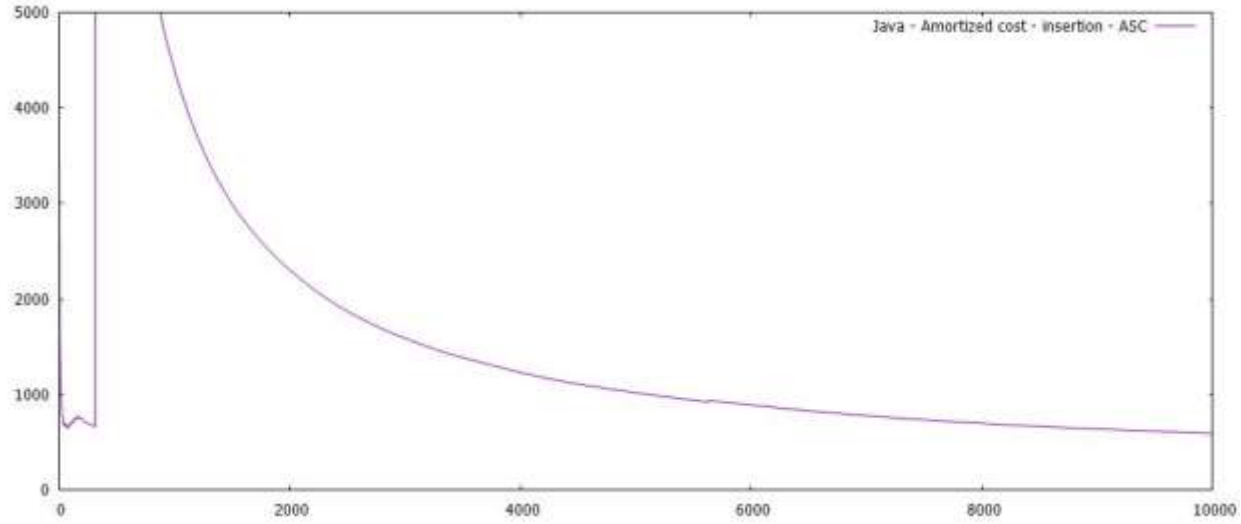


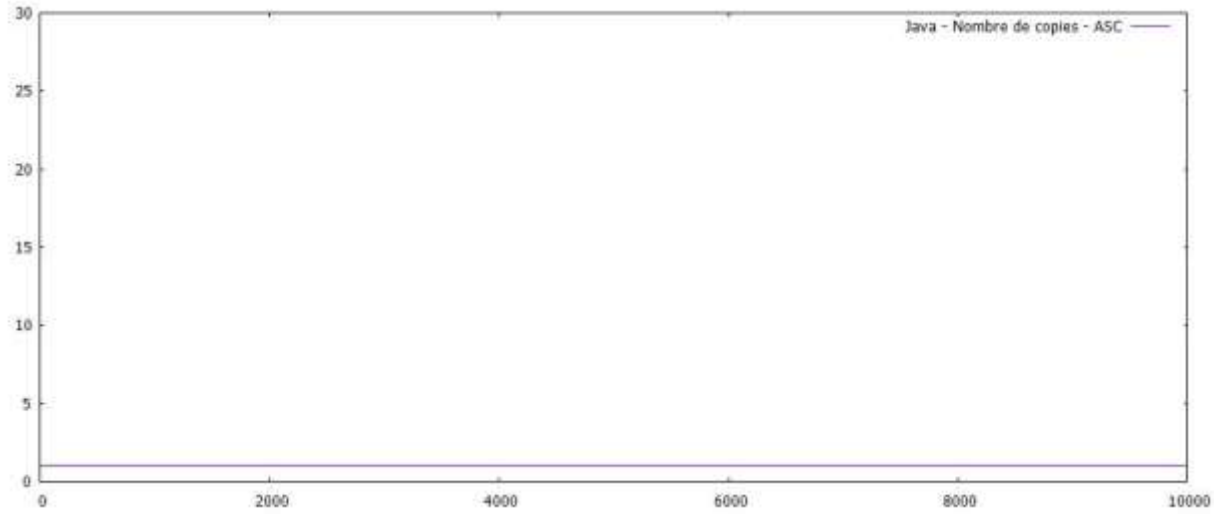
2. Effectuez des expériences sur l'efficacité en temps et en mémoire de cette structure.

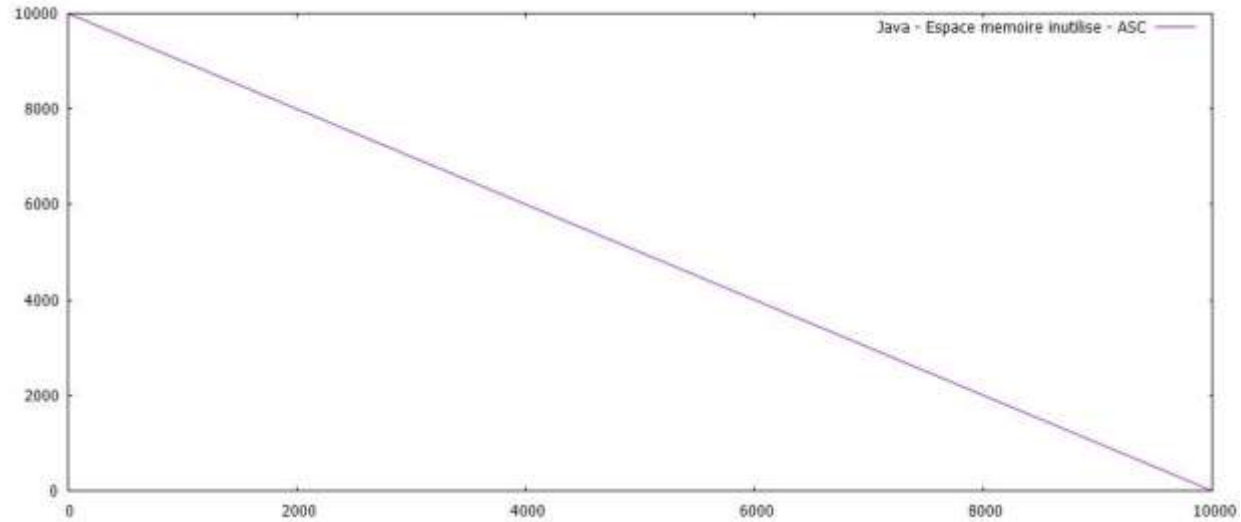
On va faire avec 10000 opérations et la hauteur est donc environs 12. Alor on a normalement le plus grand nombre de copier égale à25 (1 affectation + 2 * 12 swap).

- dans le cas où l'on ne fait qu'ajouter des clés dans l'ordre croissant

Dans ce cas, cette classe ajoute automatiquement les clés à la dernière place dans l'ordre croissant sans arranger puisque ce tas binaire a la propriété min-heap. C'est-à-dire, l'efficacité en temps de ce cas est constant 1.

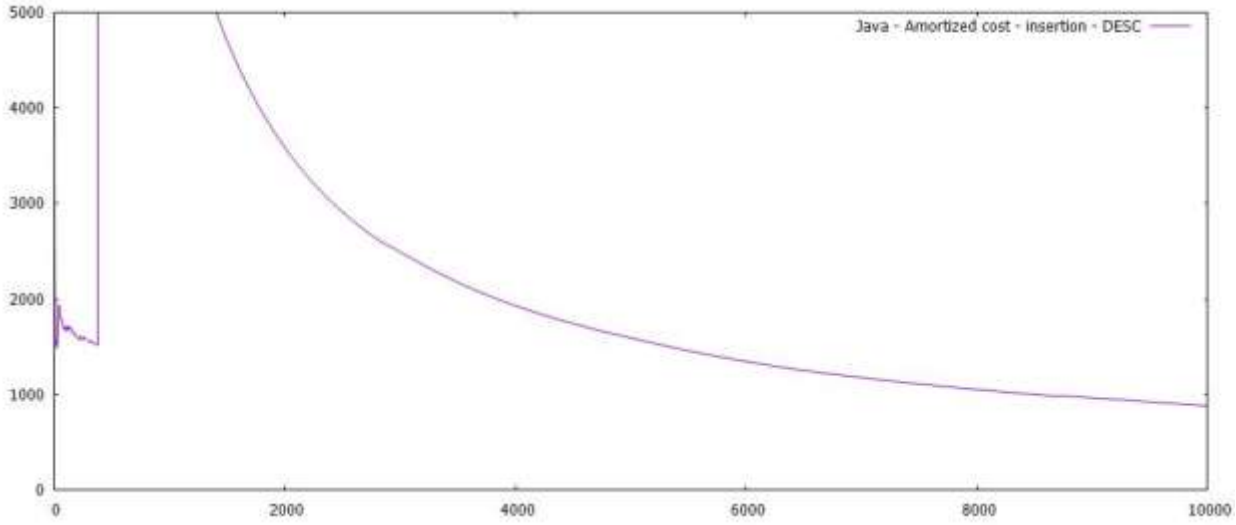


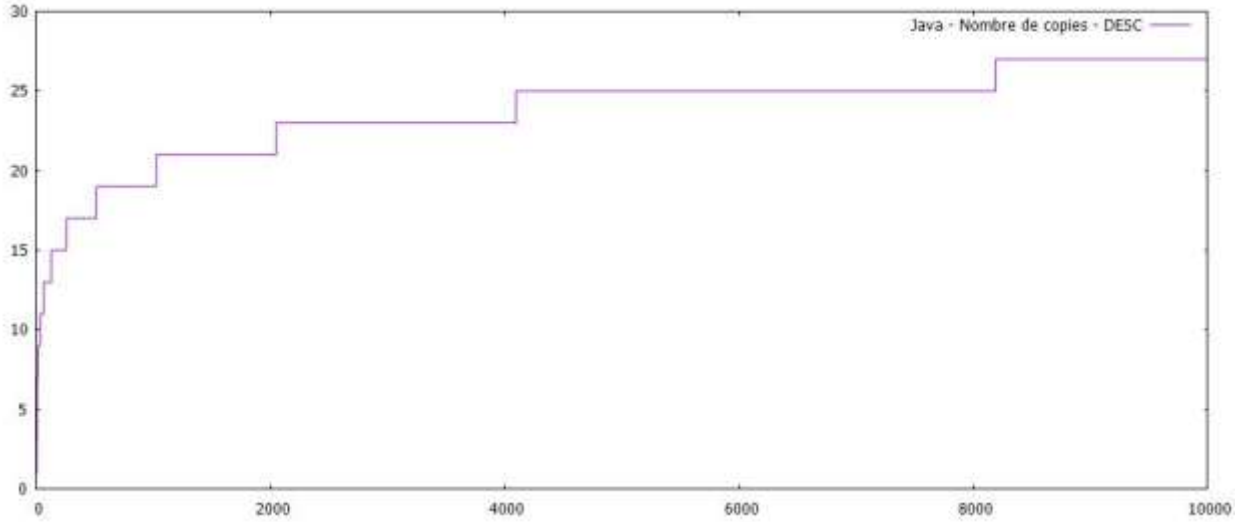




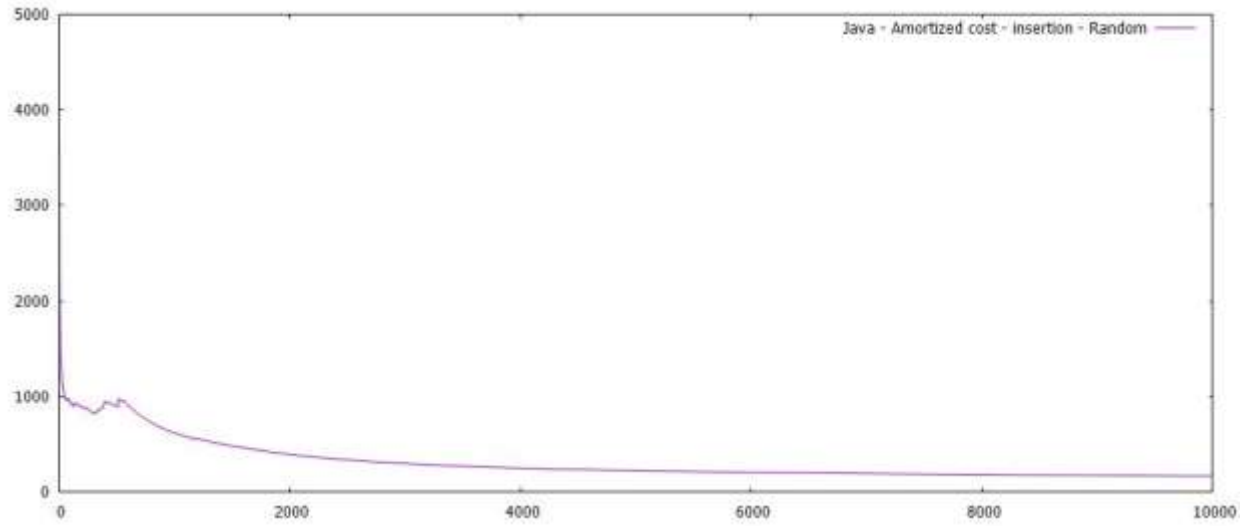
- dans le cas où l'on ne fait qu'ajouter des clés dans l'ordre décroissant

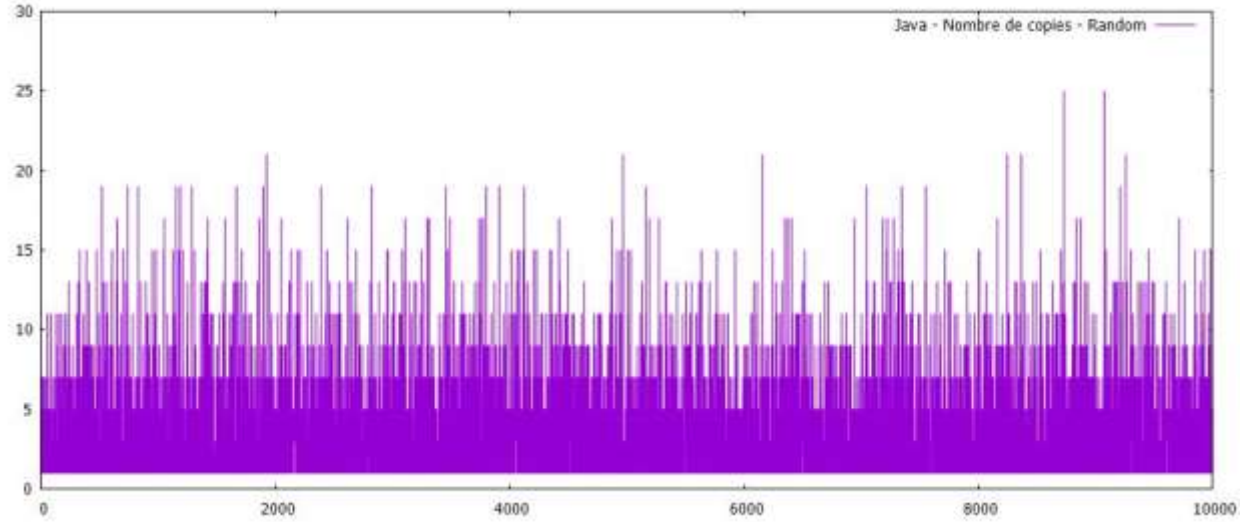
Dans ce cas, cette classe ajoute les clés à la dernière place et arrange et l'efficacité en temps de ce cas est augmenté avec la taille du tableau (avec la hauteur du tas).



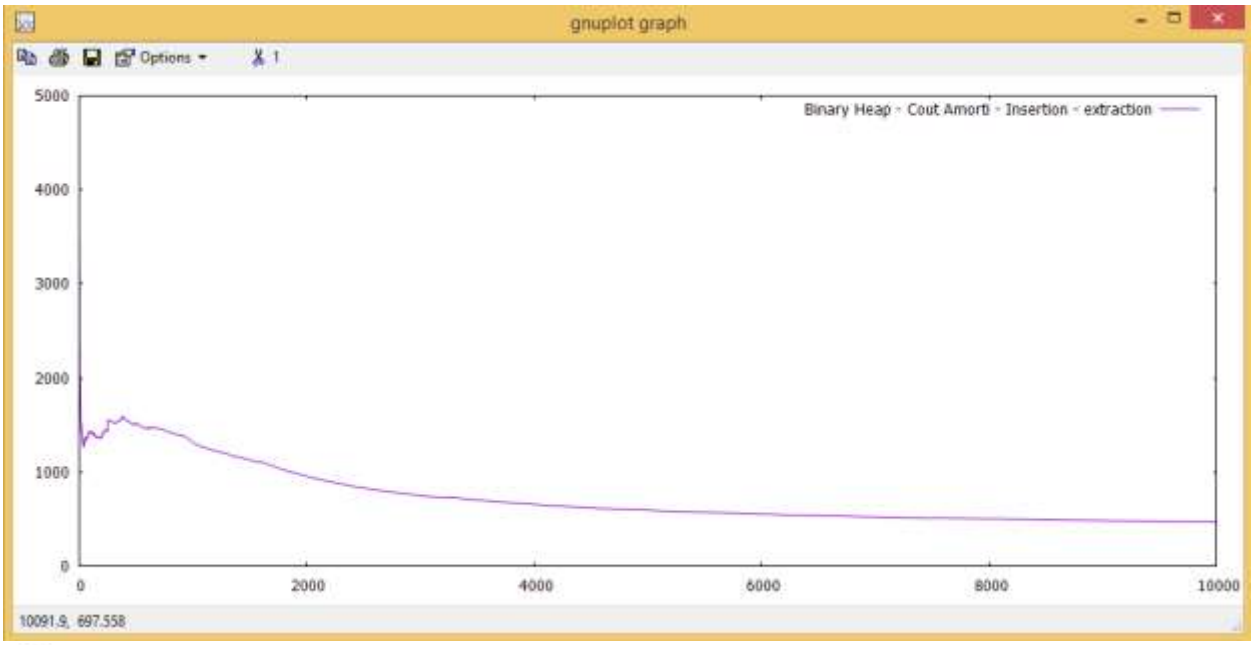


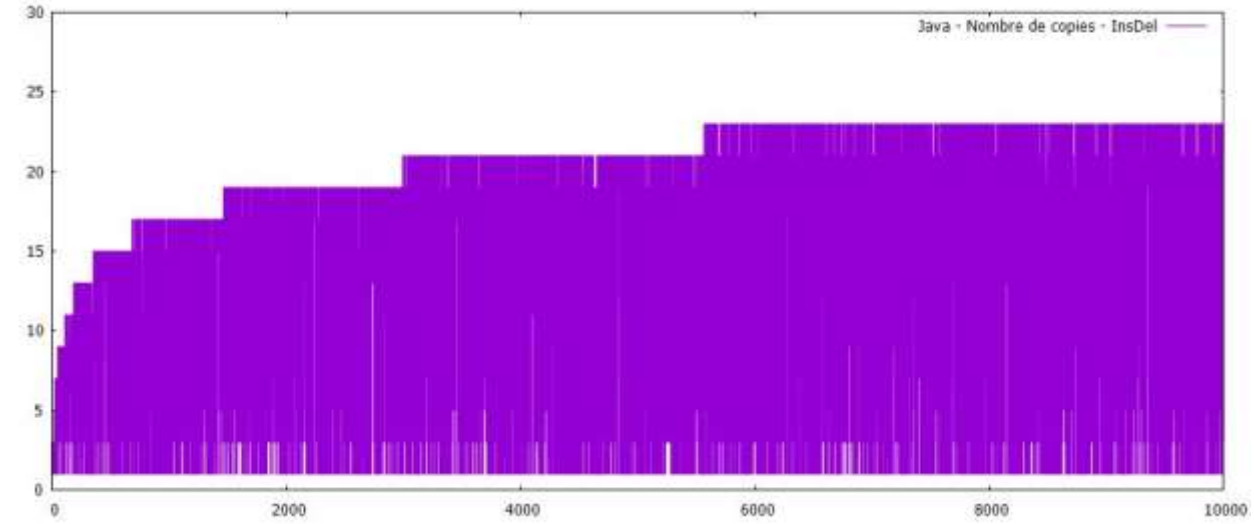
- dans le cas où l'on ne fait qu'ajouter des clés aléatoires
  Cette classe ajoute les clés et dans le meilleur cas (cas 1 - la clé ajoutée est la plus grande), on ne doit pas arranger cette clé. Dans le pire cas (cas 2 - la clé ajoutée est la plus petite), on doit échanger cette clé avec les places des clés en avant.





- puis dans le cas où l'on ajoute et on extrait des éléments.
  Dans ce cas, on va utiliser une probabilité p = 0.7 (70% insertion, 30% extraction).





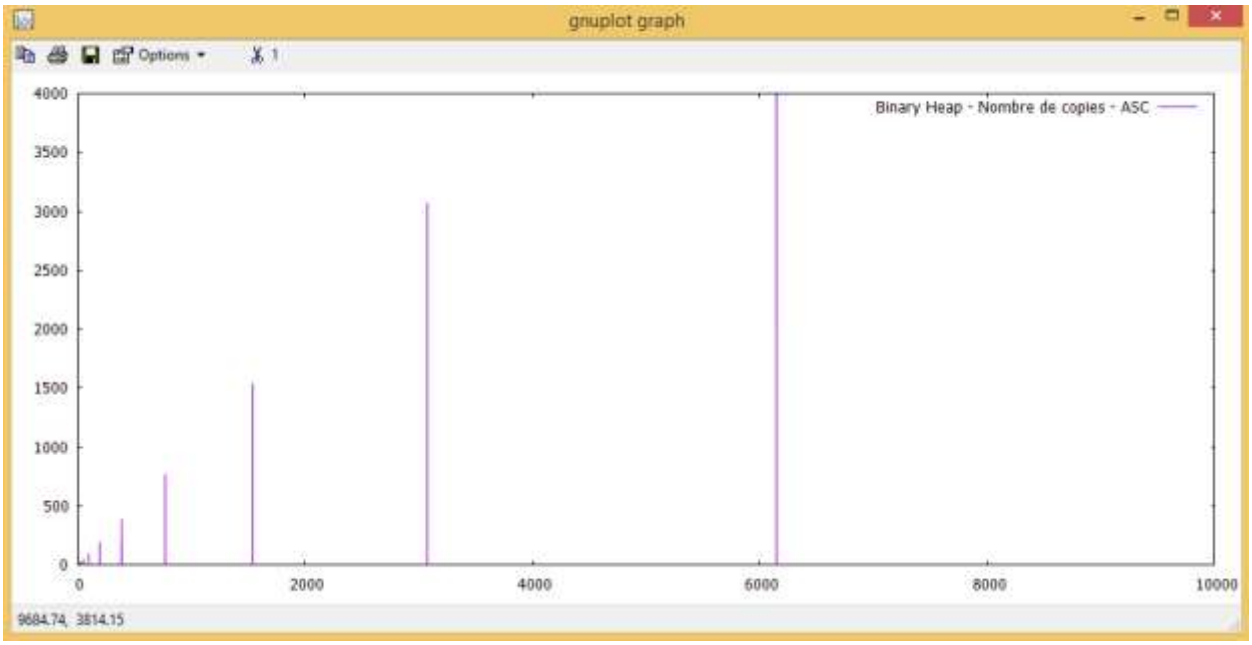
3. Remplacez le tableau de taille fixe par un tableau dynamique

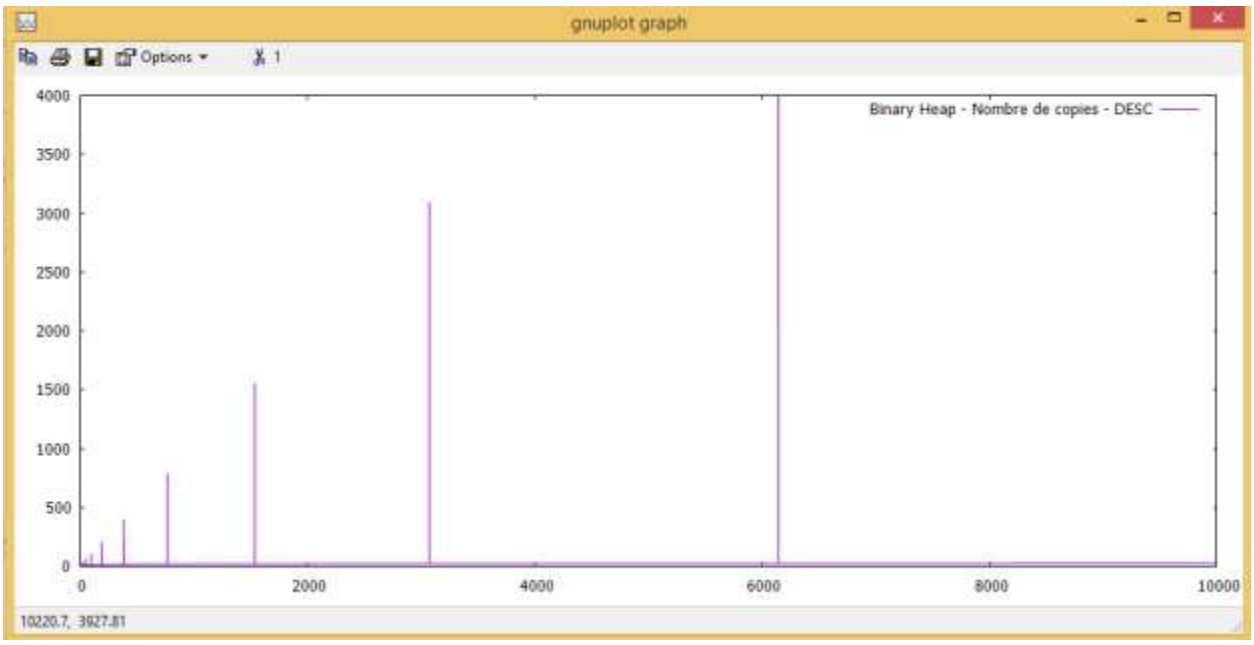
Le tas binaire n'a donc plus de taille maximale. Recommencez les expériences. La complexité amortie des opérations d'ajout et de suppression/extraction a-t-elle changé ? Pourquoi ?

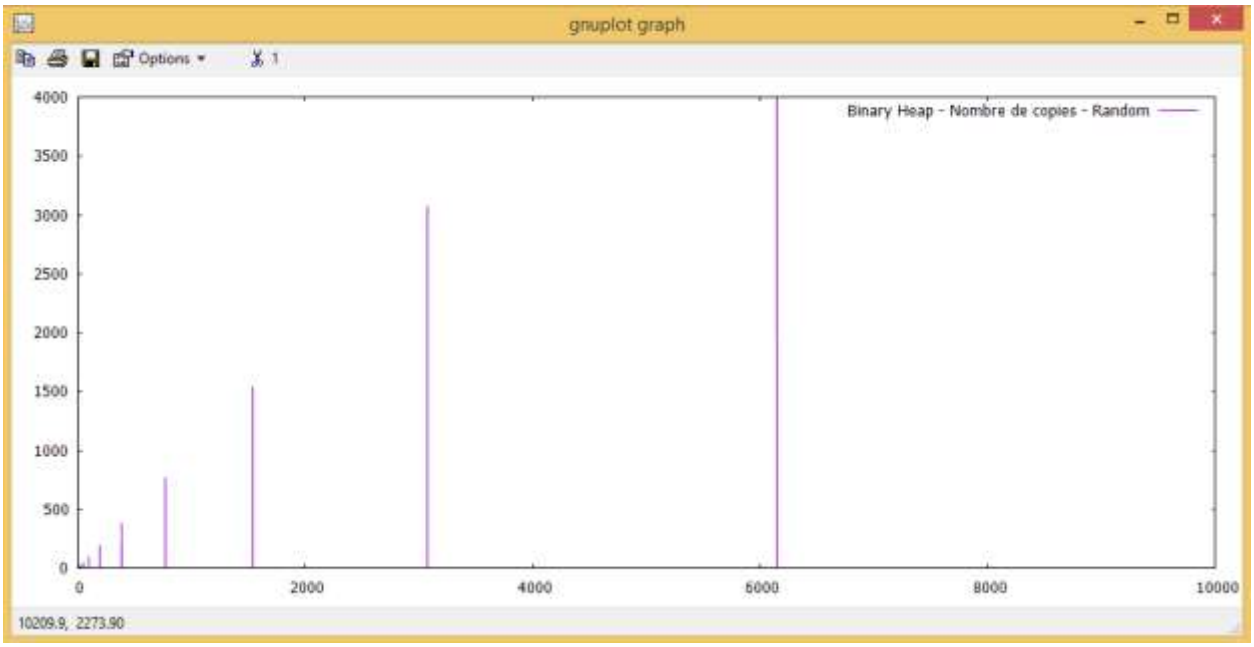
Le cout amorti des opérations d'ajout et de suppression a changé puisque avec un tableau dynamique, on doit allouer le nouveau tableau et copier les valeurs d'ancien tableau. Donc, on change la formule de compter comme suivant :

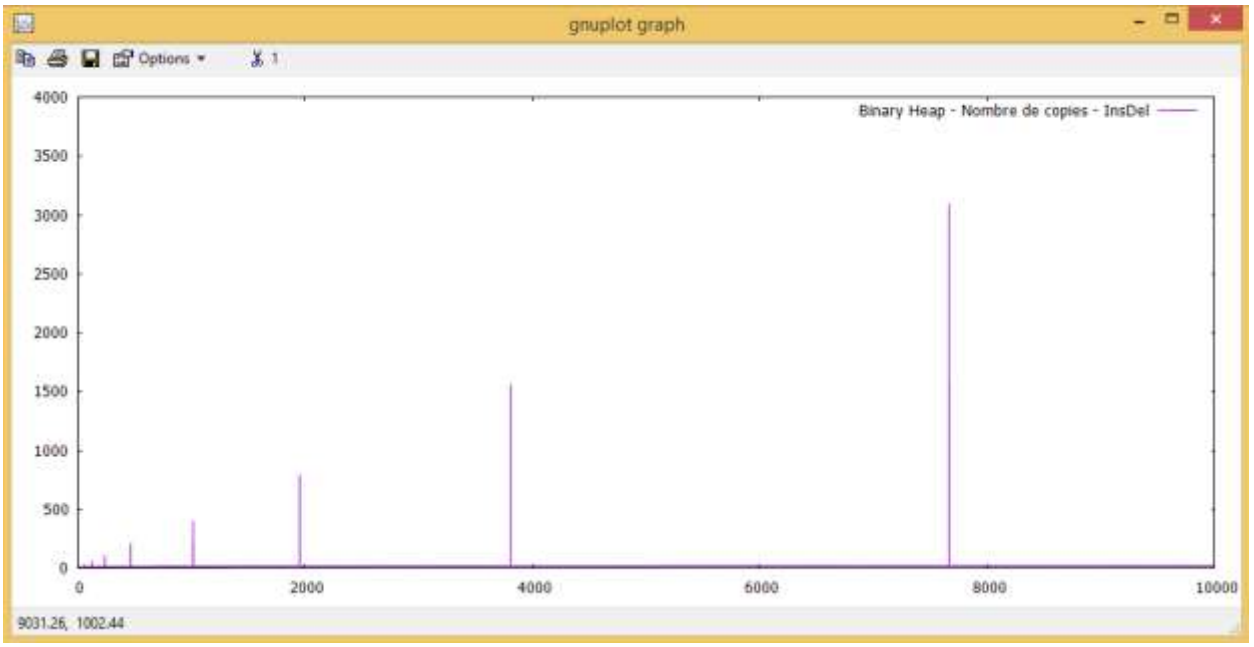
```
27              // Enregistrement du nombre de copies efféctuées par l'opération.
28              // S'il y a eu réallocation de mémoire, il a fallu recopier tout le tableau.
29              if (isAllocated) {
30                  copy_analysis.append(heap.countSwap * 2 + heap.size());
31              } else {
32                  copy_analysis.append(1 + heap.countSwap * 2);
33              }
34
```

Le cout principal est pour gérer la dynamique et le cout de comparer et d'échanger n'est pas grand. Ce sont certains résultats de plot.

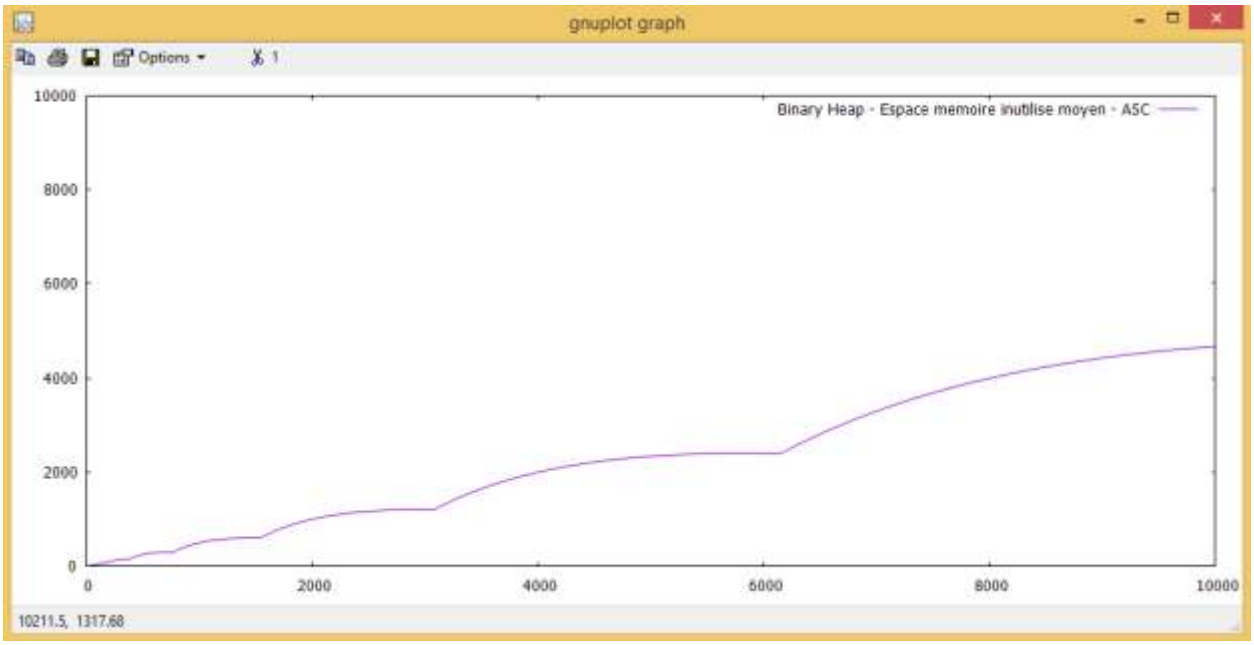


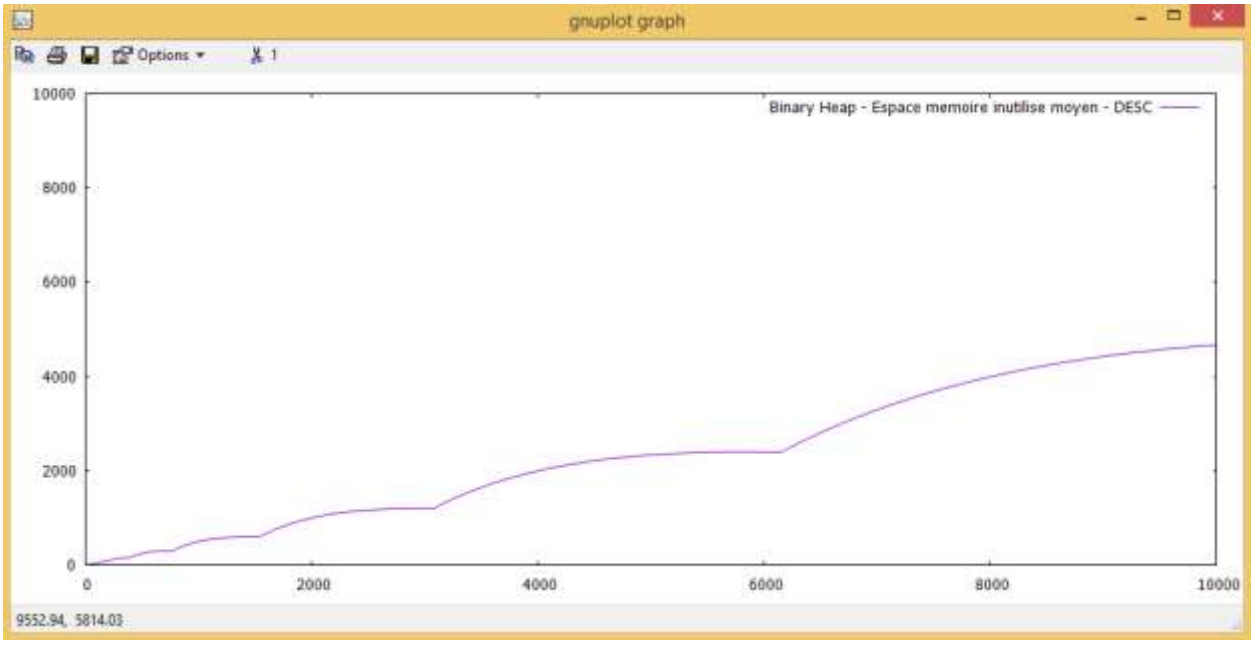


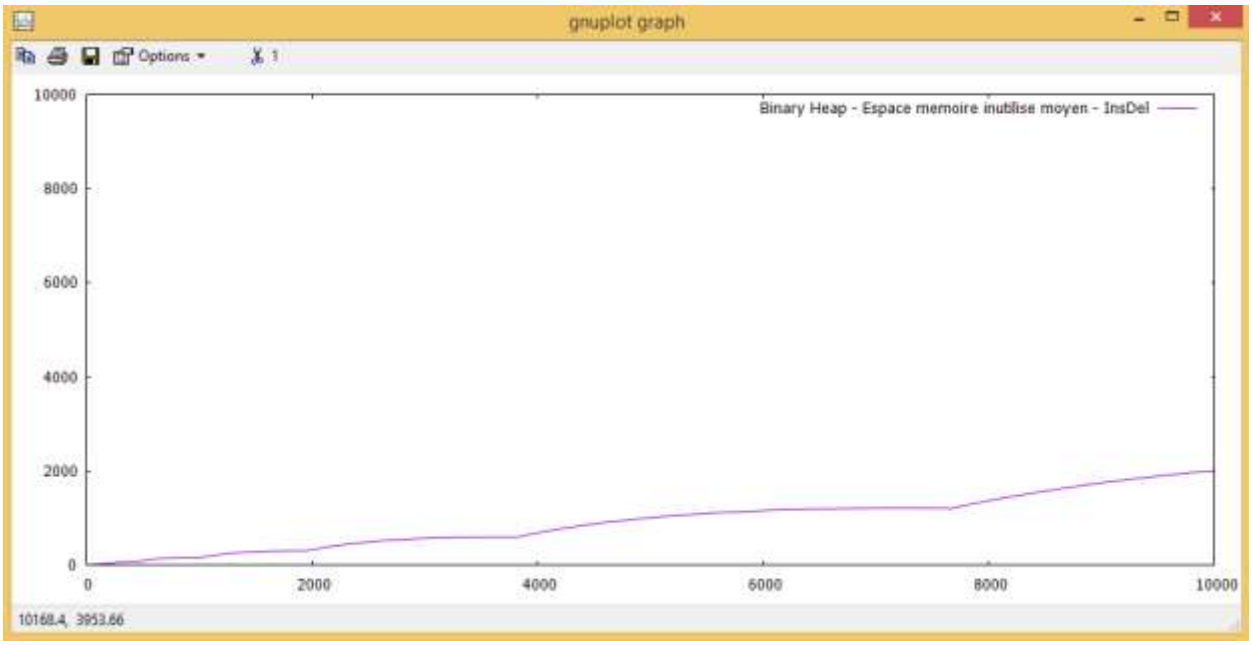




Dans trois premiers cas, les espaces mémoires inutilisés sont pareils puisque ils font seulement des opérations d'insérer et ils ont une même stratégie. En général, si on utilise le tableau dynamique dont le cout amorti est plus que le cout amorti d'un tableau fixe.


gnuplot graph
Options
1
Binary Heap - Espace memoire inutilise moyen - ASC
10000
8000
6000
4000
2000
0
0
2000
4000
6000
8000
10000
10211.5, 1317.68


gnuplot graph
Options
1
Binary Heap - Espace memoire inutilise moyen - DESC
10000
8000
6000
4000
2000
0
0
2000
4000
6000
8000
10000
9552.94, 5814.03



Tas binomial

1. Quelle est la complexité amortie de l'opération "incrémenter" sur un nombre binaire ?

Utilisez ce résultat pour démontrer la complexité amortie de l'opération d'ajout d'une clé dans un tas binomial.

La complexité amortie de l'opération "incrémenter" sur un nombre binaire : Si l'appel de INCRÉMENTER a reinitialise t_i bits, le cout reel de l'operation est au plus de $1 + t_i$ (car il met t_i bit à 0 et 1 bit à 1). Si $b_i = 0$ alors $b_{i-1} = t_i = k$. Si $b_i > 0$ alors $b_i = b_{i-1} - t_i + 1$. On vérifie que $b_i \leq b_{i-1} - t_i + 1$ dans tous les cas. La différence de potentiel est :

$$\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1}$$

$$\leq b_{i-1} - t_i + 1 - b_{i-1}$$

$$= 1 - t_i$$

.

Le cout amorti vaut alors :

$$= c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$\leq (t_i + 1) + (1 - t_i)$$

$$= 2$$

La complexité amortie de l'operation d'ajout d'une clé dans un tas binomial est pareil avec un nombre binaire.

Par exemple, un tas binomial contenant 63 éléments contiendra des arbres binomiaux d'ordre 1, 2, 3, 4, 5, et 6 puisqu'il faut six chiffres pour écrire le nombre décimal 63 en binaire. 63 en binaire est 111111. Si on ajoute d'une clé, le cout reel d'operation est 1 + 6 (car il union 6 l'arbre binomial). Enfin, le tas binomial a un seul arbre binomial d'ordre 6 ($2^6 = 64$) et alors 64 nœud. 64 en binaire est « 1000000 ».

2. Développez une structure/classe de tas binomial

Vous détaillerez vos choix d'implantation. Les trois opérations que l'on attend de vous au minimum sont la fusion de deux tas, l'ajout d'une clé et l'extraction de la plus petite clé.

On a développé une classe de tas binomial comme suivant :

- Un hachage stocke les arbres binomials comme des noeurs de tas binomial. Puisque un tas binomial a un seul arbre binomial k-ordre dans la structure. Alors un hachage avec key = integer (l'ordre) et value = BinomialTree. On peut facilement accéder et valider l'existe d'un arbre binomial k-ordre dans la structure. (Ou on peut utiliser un tableau dynamique au lieu d'un hachage)
- Un nombre entier « capacity » pour gérer la taille de hachage. Parce que l'on ne peut pas accéder la taille de hachage mais seulement le nombre d'éléments dans Java.
- Un nombre entier « countUnion » pour compter le nombre d'union d'une opération. Comme un simulateur pour regarder le cout en temps avec un petit nombre d'opérations. Une fois d'union a environs 5 affectations ***reference*** comme (node.sibling = …, node.child = …, etc)
- Pourtant le hachage a le facteur multiplicatif et le facteur de contraction, la taille de hachage dépend aussi de le plus grand clé. Par exemple, il y a seulement un arbre binomial 10-ordre dans tas binomial. Alors le nombre d'element est 1 mais la taille de hachage est 16.

```
 7  public class BinomialHeap {
 8
 9      // public List<BinomialTree> list;
10      public HashMap<Integer, BinomialTree> nodes;
11      int capacity;
12      public int countUnion;
13      public float loadFactor = 0.75f;
14      public float reduceFactor = 0.25f;
15      public boolean isReduced;
16
17      // Java document: Constructs an empty HashMap with the default initi
18      // (16)
19      // and the default load factor (0.75).
20      public BinomialHeap() {
21          nodes = new HashMap<Integer, BinomialTree>(4, loadFactor);
22          capacity = 4;
23      }
```

- On va utiliser une structure d'un arbre binomial comme un nœud de tas binomial. Ce n'est pas peut-être optimal mais c'est facile à comprendre.

```
5  public class BinomialTree {
6
7      public BinomialTreeNode root;
8      public int height;
9
10     public BinomialTree(BinomialTreeNode root) {
11         super();
12         this.root = root;
13         this.height = root.height;
14     }
```

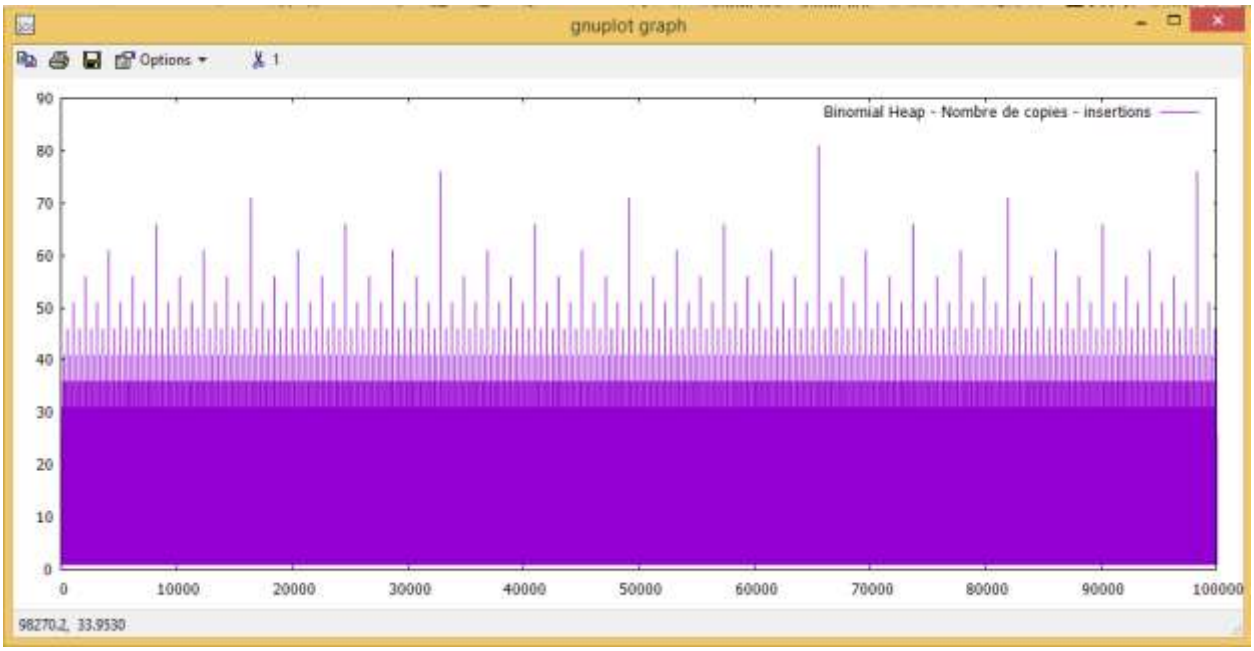
3. Effectuez des expériences sur l'efficacité en temps et en mémoire de cette structure

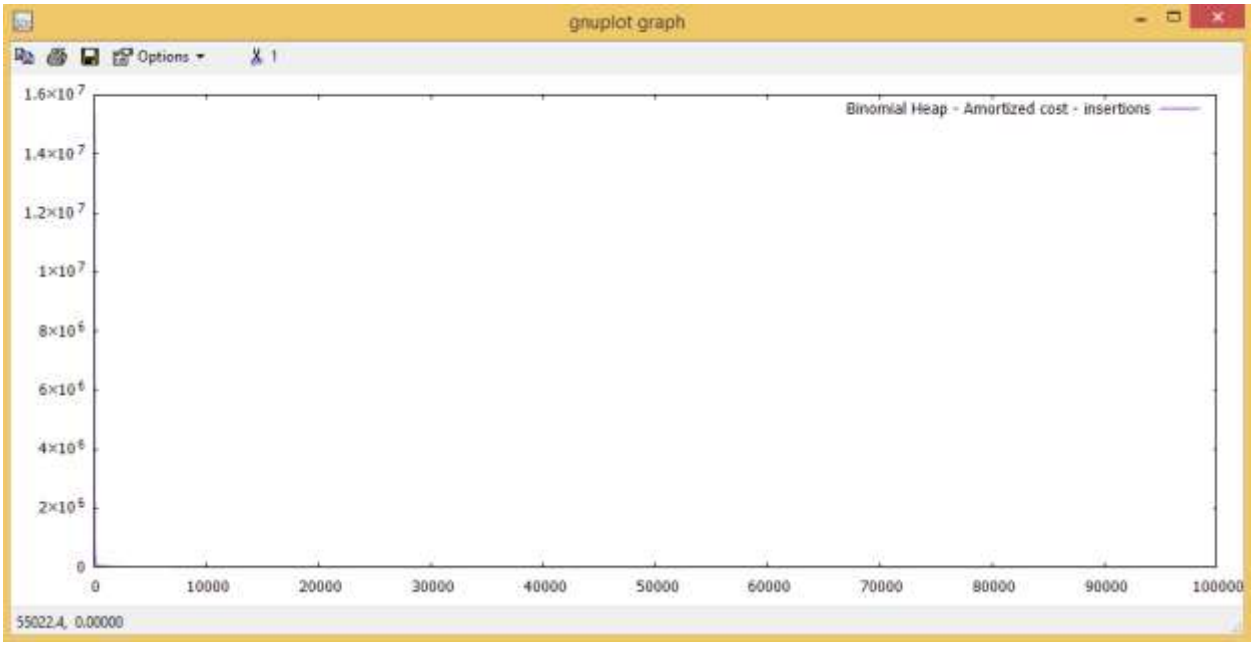
Dans un premier temps, vous ferez uniquement des ajouts dans le tas, puis des ajouts et des suppressions. Enfin vous imaginerez une expérience permettant de tester l'ajout et la suppression de clés dans plusieurs tas et, dans de rares moments, vous fusionnerez deux tas. Commentez ces expériences.

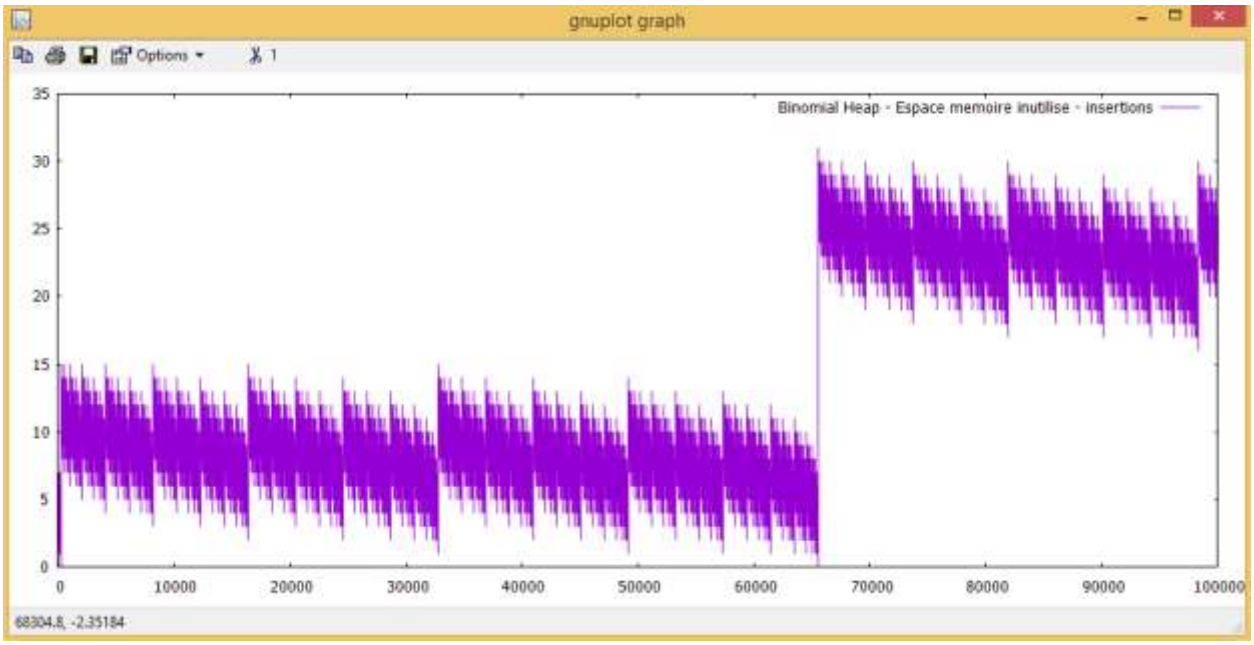
D'abord, on utilise une formule pour compter le nombre de copier. C'est le nombre d'affectation d'une opération (le nombre d'opération Union) :

```
30              // Enregistrement du nombre de copies efféctuées par l'opération.
31              // S'il y a eu réallocation de mémoire, il a fallu recopier tout le tableau.
32              if (isAllocated) {
33                  copy_analysis.append(binomialHeap.size() + binomialHeap.countUnion * 5);
34              } else {
35                  copy_analysis.append(1 + binomialHeap.countUnion * 5);
36              }
37
```

- Dans le cas où on exécute des ajoutes des clés aléatoires :
  - Le nombre de copies ne change pas beaucoup. Puisque la complexité d'ajoute de tas binomial est constant (1). De plus, la taille de hachage n'est pas grande avec 100000 opérations et la plupart de copie est des affectations de nœud.
  - L'espace mémoire inutilisé est basé sur la mémoire de hachage. En première fois, on a compté les valeurs NULL des nœuds comme un mémoire inutilisé mais on a après réalisé que dans Java, il y a une valeur NULL pour tous les variables. C'est-à-dire, si un variable est NULL, la référence de ce variable est la NULL pointeur. Donc on compte finalement sur la mémoire de hachage.
  - Comme tu vois la graphie d'espace mémoire inutilisé, quand le nombre d'élément égale à la taille, on va grandir la taille du hachage dans la prochaine fois pourtant le nombre d'élément de prochaine fois égale à 1.


gnuplot graph
Options
1
Binomial Heap - Nombre de copies - insertions
90
80
70
60
50
40
30
20
10
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
98270.2, 33.9530

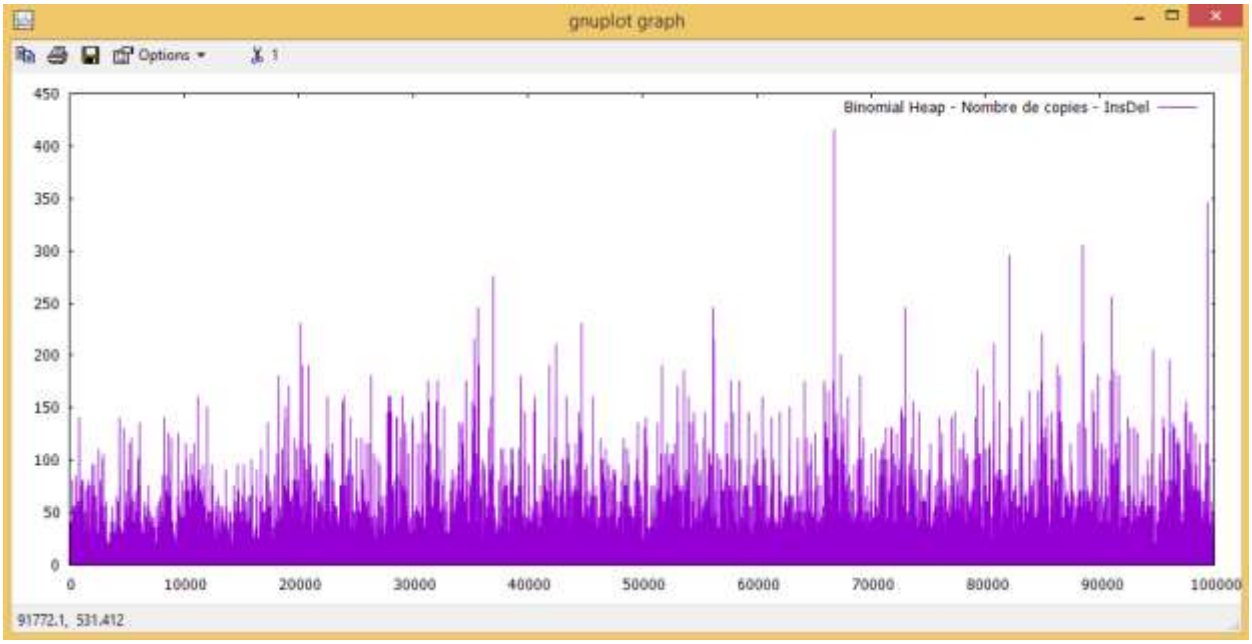

gnuplot graph
Options
1
Binomial Heap - Amortized cost - insertions
1.6×10^7
1.4×10^7
1.2×10^7
1×10^7
8×10^6
6×10^6
4×10^6
2×10^6
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
55022.4, 0.00000

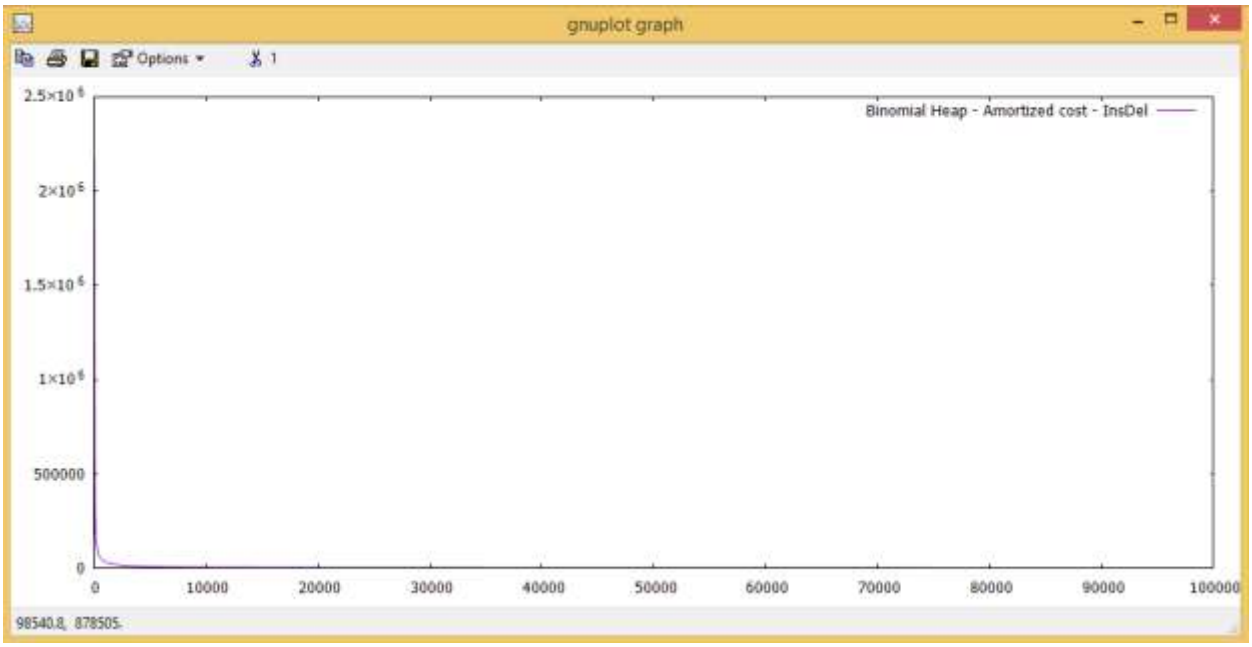


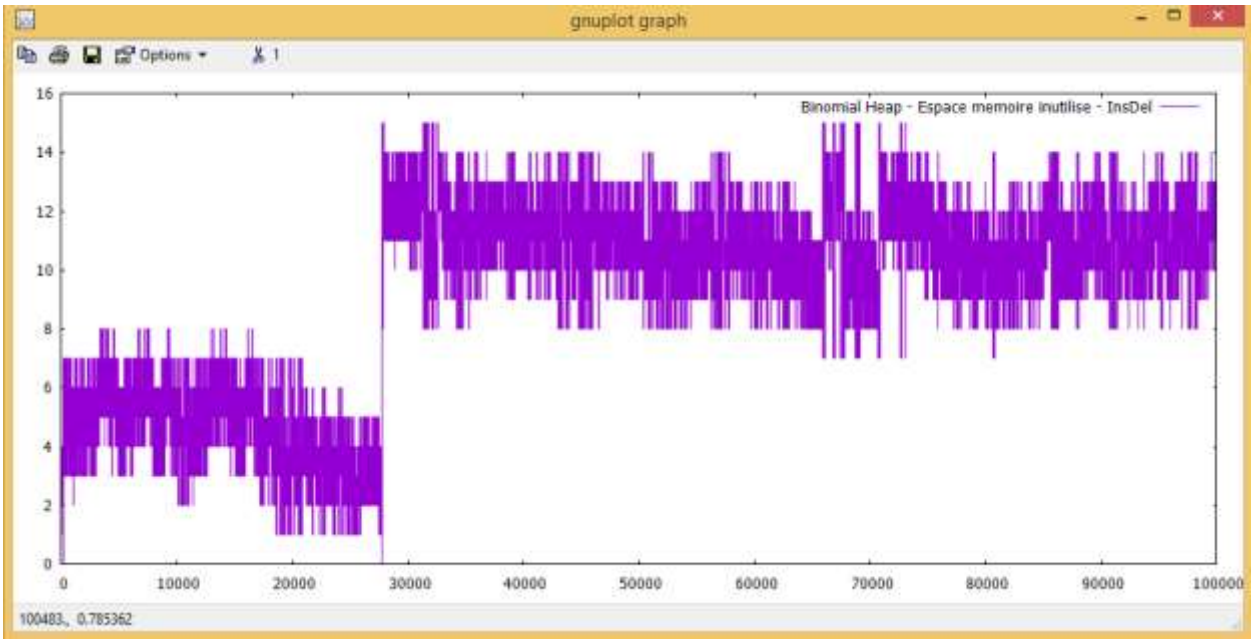
Dans le cas où on exécute des ajoutes et des suppressions :

- Dans ce cas, on utilise la probabilité d'ajoute est p = 0.5 (50% insertion, 50% suppression)

- La complexité de suppression (extraction la plus petite clé) est O(log *n*). Dans ce cas, le nombre de copies est plus grand que le nombre de copies de premier cas (toutes les ajoutes).



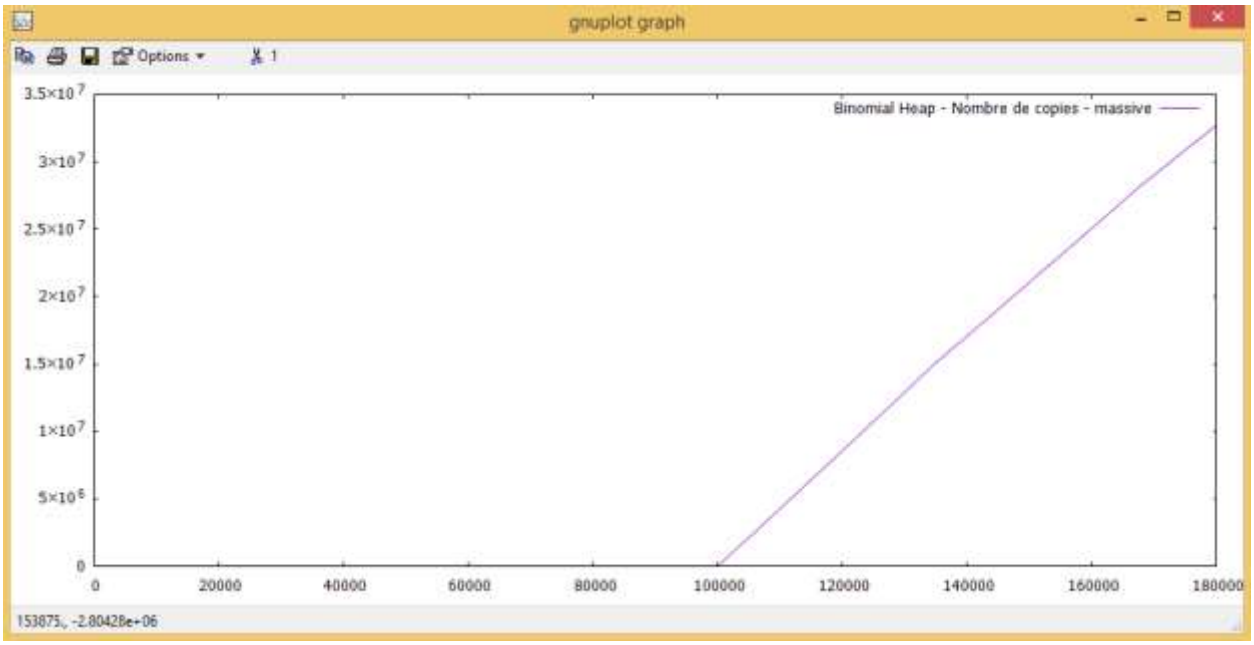


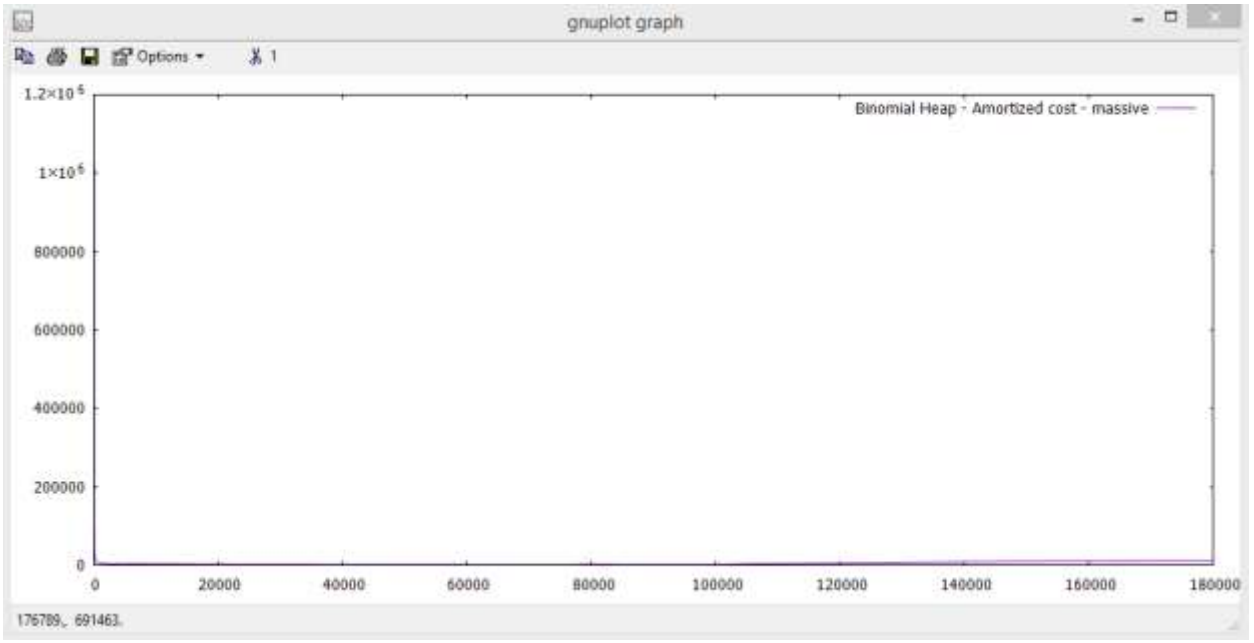


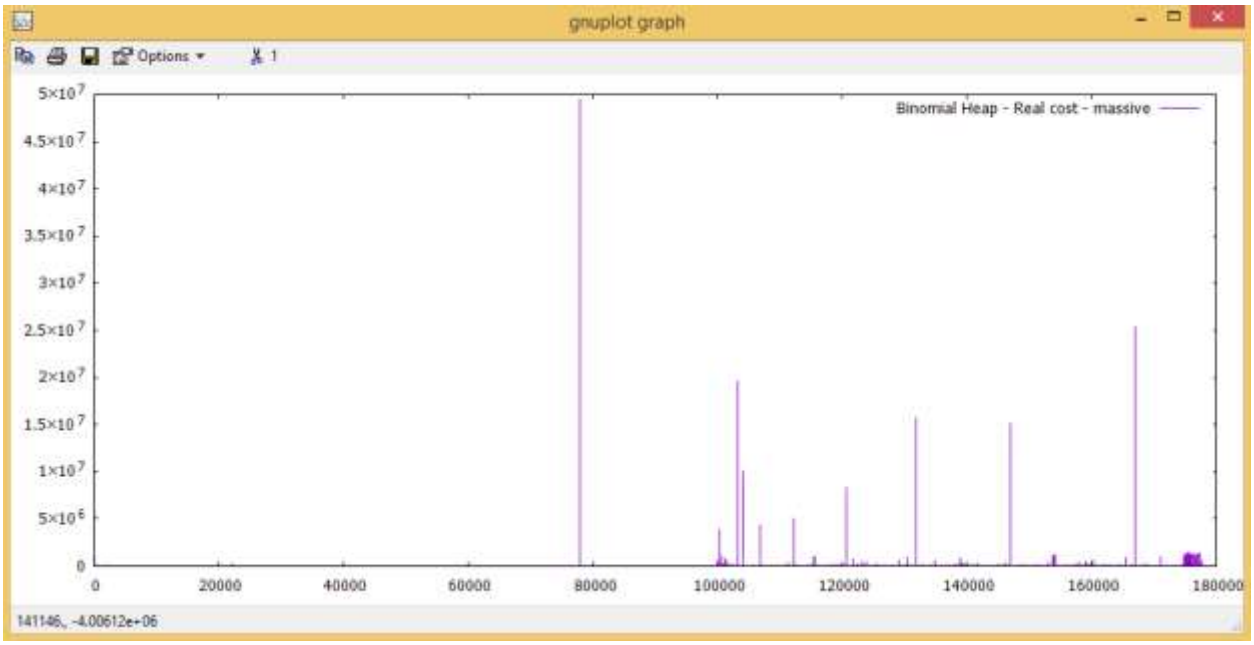
- Dans le cas où on exécute l'ajoute et la suppression de clés dans plusieurs tas (6 tas binomial), on va faire 100 milles d'ajoute et 80 milles de suppression après :
  - Le nombre de copies est augmenté dans les suppressions puisque l'on doit faire plusieurs des opérations Union après une suppression.
  - L'espace mémoire inutilisé est stable dans les ajoutes ou les suppressions. Parce que la taille du hachage dépend du plus grand arbre binomial dans le tas binomial.

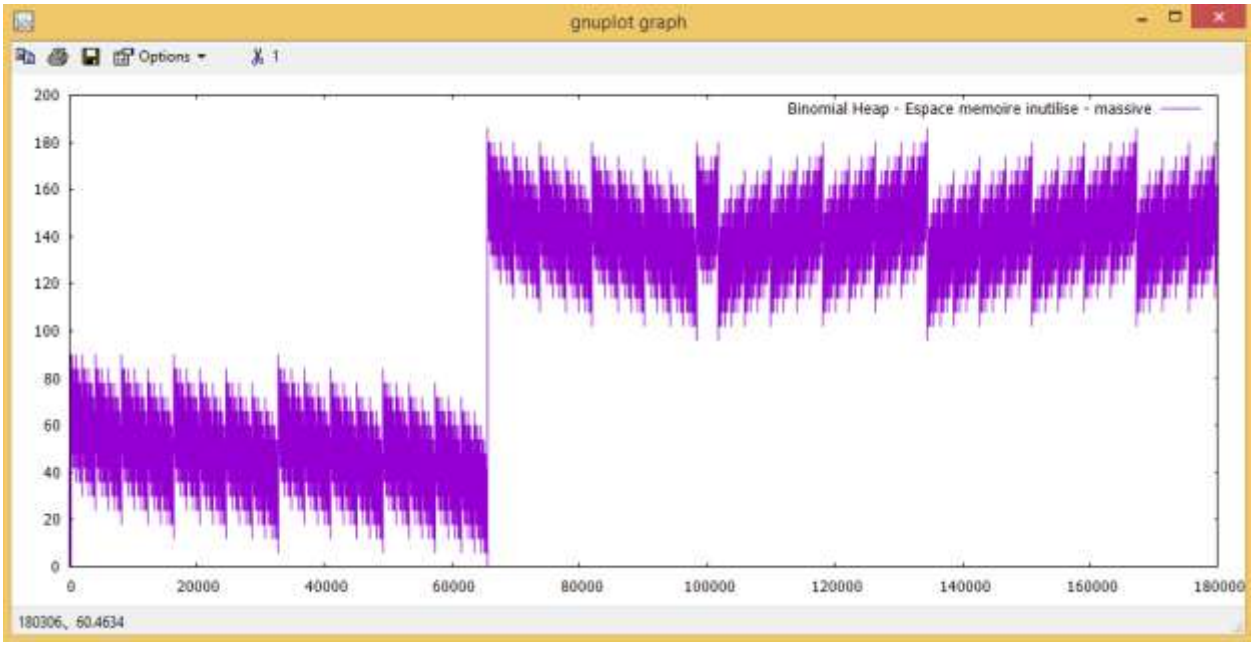
```
27          boolean isAllocated = false;
28          for (int i = 0; i < 100000; i++) {
29              element = randomGen.nextInt();
30              before = System.nanoTime();
31              heap1.add(element);
32              heap2.add(element);
33              heap3.add(element);
34              heap4.add(element);
35              heap5.add(element);
36              isAllocated = heap6.add(element);
37              after = System.nanoTime();
38              time_analysis.append(after - before);
39
40              // Enregistrement du nombre de copies efféctuées par l'opération.
41              // S'il y a eu réallocation de mémoire, il a fallu recopier tout le tableau.
42              if (isAllocated) {
43                  copy_analysis.append((heap1.size() + heap1.countUnion * 5) * 6);
44              } else {
45                  copy_analysis.append((1 + heap1.countUnion * 5) * 6);
46              }
47              //
48              // Enregistrement de l'espace mémoire non-utilisé.
49              memory_analysis.append(heap1.getUnusedMemory() * 6);
50          }
51
```

```
52          for (int i = 0; i < 80000; i++) {
53              before = System.nanoTime();
54              heap1.extractMin();
55              heap2.extractMin();
56              heap3.extractMin();
57              heap4.extractMin();
58              heap5.extractMin();
59              heap6.extractMin();
60              after = System.nanoTime();
61              isAllocated = heap1.isReduced;
62              time_analysis.append(after - before);
63
64              // Enregistrement du nombre de copies efféctuées par l'opération.
65              // S'il y a eu réallocation de mémoire, il a fallu recopier tout le tableau.
66              if (isAllocated) {
67                  copy_analysis.append((heap1.size() + heap1.countUnion * 5) * 6);
68              } else {
69                  copy_analysis.append((1 + heap1.countUnion * 5) * 6);
70              }
71              //
72              // Enregistrement de l'espace mémoire non-utilisé.
73              memory_analysis.append(heap1.getUnusedMemory() * 6);
74          }
75
```


gnuplot graph
Options
1
Binomial Heap - Nombre de copies - massive
3.5×10^7
3×10^7
2.5×10^7
2×10^7
1.5×10^7
1×10^7
5×10^6
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
153875, -2.80428e+06


gnuplot graph
Options
1
Binomial Heap - Amortized cost - massive
1.2×10^6
1×10^6
800000
600000
400000
200000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
176789, 691463.


gnuplot graph
Options
1
Binomial Heap - Real cost - massive
5×10^7
4.5×10^7
4×10^7
3.5×10^7
3×10^7
2.5×10^7
2×10^7
1.5×10^7
1×10^7
5×10^6
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
141146, -4.00612e+06


gnuplot graph
Options
1
Binomial Heap - Espace memoire inutilise - massive
200
180
160
140
120
100
80
60
40
20
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
180306, 60.4634