**Sommaire**

**Coût amorti3**

**Choix d’implémentation4**

**Insertion des clés dans l’ordre croissant5**

**Insertion des clés dans l’ordre décroissant7**

**Conclusion9**

**Pour aller plus loin10**

**Quelle est la complexité amortie de l’opération "incrémenter" sur un nombre binaire ?**

Un nombre (compteur binaire est représenté par un tableau (ou liste) de **0** et **1** de taille fixe **K.**

Algorithme d’incrémentation :

**Incrémentation(B)**

i=0

Tant que i < log B et B[i] =1 Faire

B[i] =0

i = i+&

Si i < log B Faire

B[i] =1

La complexité d’Incrémentation(B) au pire cas est :

O(K) = O(log n)

**Quelle est la complexité de n opération Incrémentation(B) ?**

i= 0

Tant que i < n faire

Incrémentation(B)

Coût total de n opérations d’Incrémentation(B) est :

n/2 + n/4 + n/8 + … +n/n

= n ∑ 1/2i < n ∑ 1/2i = 2n

= O(n) => le cout amorti de Incrémentation(B) = O(n) / n = O(1) = Ĉ i

Analyse par méthode du potentiel

Ĉ i= Ci + Φ (Bi) – Φ(Bi-1)

Utilisez ce résultat pour démontrer la complexité amortie de l’opération

D’ajout d’une clé dans un tas binomial.

**Choix d’implémentation :**

Les tas binomiaux sont une collection d’arbres binomiaux, pour les implémenter on va utiliser les listes chainées, les racines des arbres binomiaux d’un tas binomial sont organisées en liste chainée par ordre de degrés croissant, appelée liste des racines.

Un tas binomial est constitué d’arbres binomiaux, un arbre binomial est quant à lui constitué de nœuds, pour définir une structure pour ces arbres on a défini une structure pour chaque nœud, cette structure est constituée de 3 pointeurs, un vers les enfants du nœud, un vers le parent et un autre vers le frère, on a aussi la clé du nœud et son degré.

Après avoir défini cette structure on peut alors créer toutes les fonctions dont on a besoin :

**1. Create\_noeud** : elle permet de créer un nouveau nœud en lui allouant de la mémoire, puis renvoie un objet p où tete[p] = NULL et cela en temps constant O(1).

**2.Lien\_Binomial :** elle prend en paramètre deux nœuds (ayant la structure décrite en haut) de même degré et fais de l’un le parent de l’autre (le parent c’est celui ayant la plus petite clé), cette opération se fait en temps constant O(1) car on ne fait que changer le pointeur du parent.

**3 .Fusion\_Listes\_Tas :** elle prend en paramètre deux tas binomiaux et fusionne les listes de leurs racines pour en faire une seule liste ordonnée de manière croissante en fonction du degré des arbres binomiaux.

**4.Union\_Binomial :** elle prend deux tas binomiaux en paramètre aussi, cette procédure se déroule en deux étapes, elle commence par faire appel à la fonction « Fusion\_Listes\_Tas » pour fusionner les racines des deux tas par ordre de degrés croissant, puis elle fait appel à « Lien\_Binomial » pour fais l’union entre chaque deux arbres ayant le même degré. L’union de tas binomiaux se fait en O(logn).

**5.Inserer\_Binomial :** elle prend en paramètre un tas et un nœud et créer en temps constant O(1) un tas binomial au nœud avant de le fusionner avec le tas en faisant appel à la fonction « Union\_Binomial » et cela en temps 0(logn).

**6.DISPLAY :** cette fonction nous affiche la liste des racines du tas, on l’utilise pour tester que les fonctions en haut fonctionnent bien.

Après avoir fait l’analyse des tas binaires statiques et dynamiques, on passe maintenant à une autre structure d’arbre qui est « les tas binomiaux » aussi connues sous le nom de tas fusionables et qui sont représentés par une collection d’arbre binomiaux.

Dans ce rapport on va donc analyser l’efficacité de cette structure en temps et en espace et surtout la comparer à notre structure précédente qui est les tas binaire.

**Insertion des valeurs croissantes :**

On a effectué des expériences sur l’efficacité en temps et en mémoire de cette structure.

Dans un premier temps, on a fait uniquement des ajouts dans le tas, on a affiché le coût réel de l’opération insérer des valeurs croissantes et la mémoire utilisée pour cette opération. On remarque que le graphe du coût réel est très perturbé, sa valeur varie entre 150 et 600 un intervalle assez grand et cela revient au fait qu’on fait l’union de deux tas très souvent, qui elle même fait appel à fusion.

Effectivement l'opération de fusion de deux tas est souvent utilisée et est réutilisée dans la plupart des autres opérations tel qu’union, qui est appelée à son tour dans l’opération insérer. L’appel à UNION-TAS-BINOMIAUX se charge de libérer le tas binomial temporaire. Les listes de racines des deux tas sont parcourues simultanément. Si seul un des deux tas contient un arbre d'ordre j, celui-ci est ajouté au tas fusionné. Si les deux tas contiennent un arbre d'ordre j, les deux arbres sont fusionnés en un arbre d'ordre j+1 en respectant la structure de tas. Sachant que l'on peut avoir besoin de fusionner cet arbre avec un arbre d'ordre j+1 présent dans un des deux tas initiaux.

On remarque aussi la mémoire utilisée pour l’insertion augmente de plus en plus ce qui est logique ; on alloue à chaque fois qu’on insère donc en insérant un grand nombre de valeurs on utilise beaucoup de mémoire.

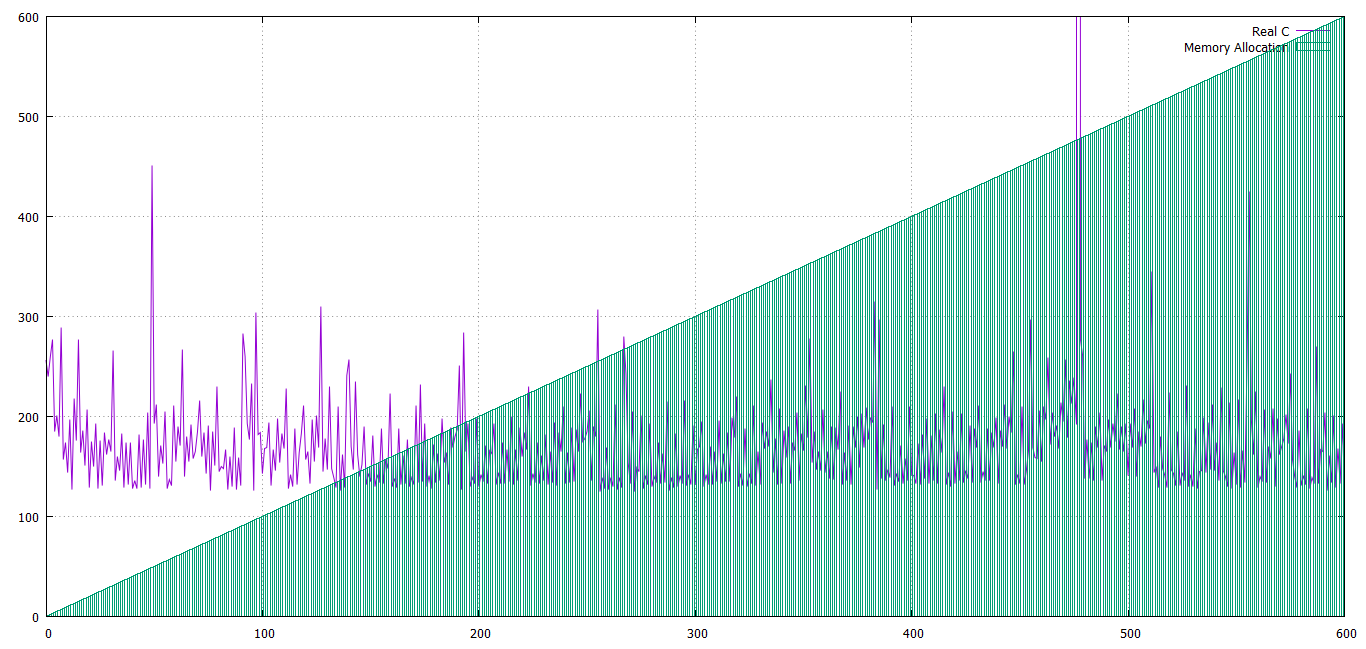


Figure  : Coût réel et espace mémoire utilisé pour l’insertion des clés croissantes

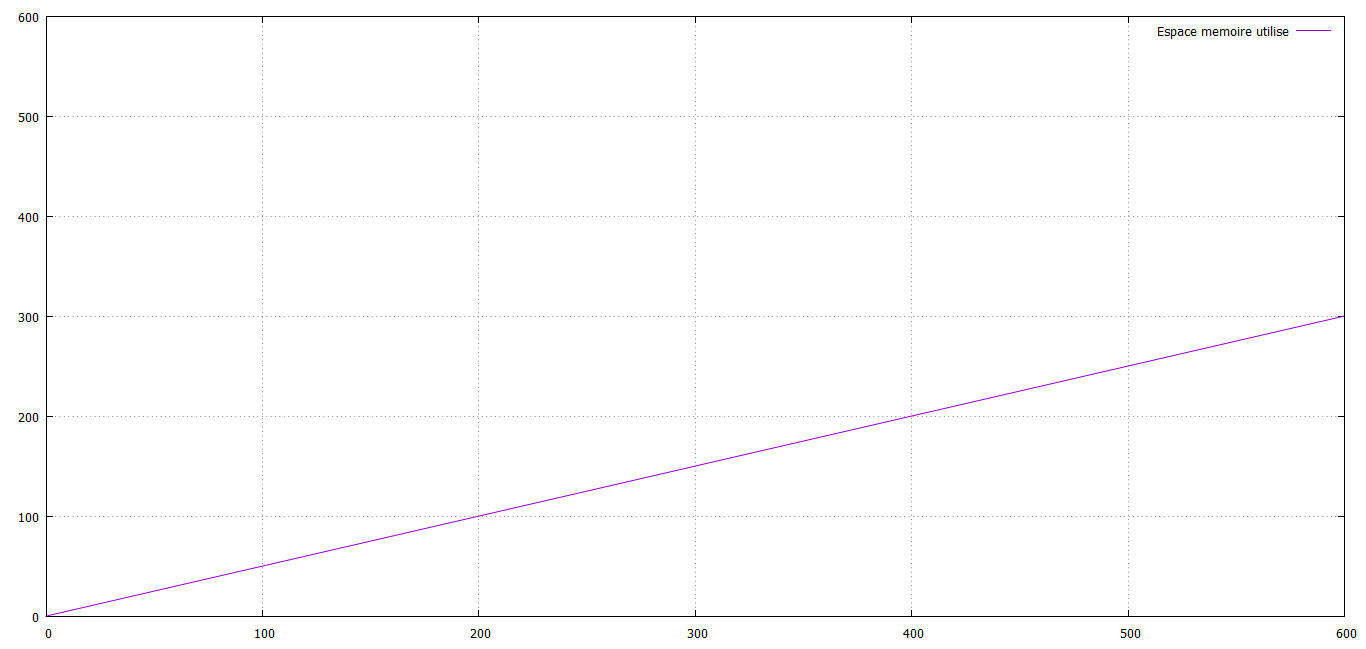


Figure  : Espace mémoire utilisé pour l’insertion des clés croissantes

Puis on a affiché le coût amorti et la mémoire utilisée pour cette opération. On remarque que le graphe décroit d’une valeur de 500 pour se stabiliser dans un intervalle entre 180 et 220, le graphe rencontre certaines perturbations dans cet intervalle et cela revient à la même raison mentionnée précédemment pour le cout réel. On remarque aussi la même chose concernant la mémoire utilisée.

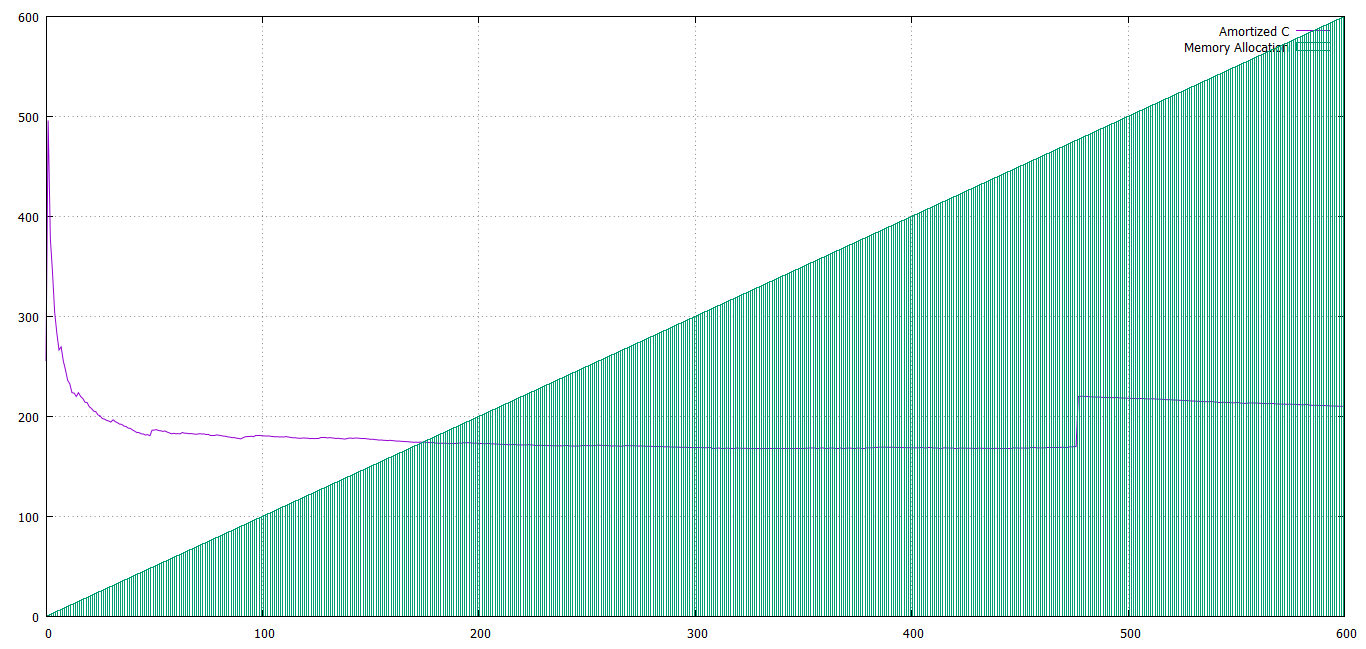


Figure  : Coût amorti et mémoire utilisée pour l’insertion croissante des clés

**Insertion des clés dans l’ordre décroissant :**

Nous avons effectuées des insertions de clés dans l’ordre décroissant et pour analyser nos résultats nous avons affiché le graphe suivant.

On remarque que le coût amorti n’est pas très élevé, mais il augmente jusqu’à 500 au début et cela peut s’expliquer par le fait qu’en insérant on fait appel à la fonction union. Puis il devient constant au fur et à mesure des insertions.

En le comparant au coût amorti de l’insertion croissante on remarque qu’il n’y a pas beaucoup de différence, ils augmentent tous les deux jusqu’à 500 au début. On pense que la fonction d’insertion se comporte de la même manière, peu importe la manière dont les éléments sont insérés celle-ci fait toujours appel à la fonction Fusion\_Listes\_Tas et Link\_Binomial, ce qui fait que le coût amorti ne dépend pas tellement de la façon d’insertion.

Contrairement aux tas binaires, ici on n’a pas analysé l’espace mémoire inutilisée car on ne réalloue pas de mémoire mais on utilise des listes chainées, le seule moment où on alloue de la mémoire est au moment de la création des nœuds.

On a donc analysé l’espace mémoire utilisé durant ces insertions, qui peut aussi être considéré comme le nombre de nœuds qui ont était créé ou inséré. On voit qu’à l’insertion numéro 100 par exemple, l’espace mémoire utilisé atteint 100, ce qui est normal car on a inséré 100 éléments. Cet espace mémoire augmente donc au fur et à mesure de l’insertion.

Mais, comme dit précédemment, dans notre structure on utilise des listes chainées de nœuds, on alloue donc de la mémoire quand on en a besoin, ce qui veut dire qu’on n’en gaspille jamais.

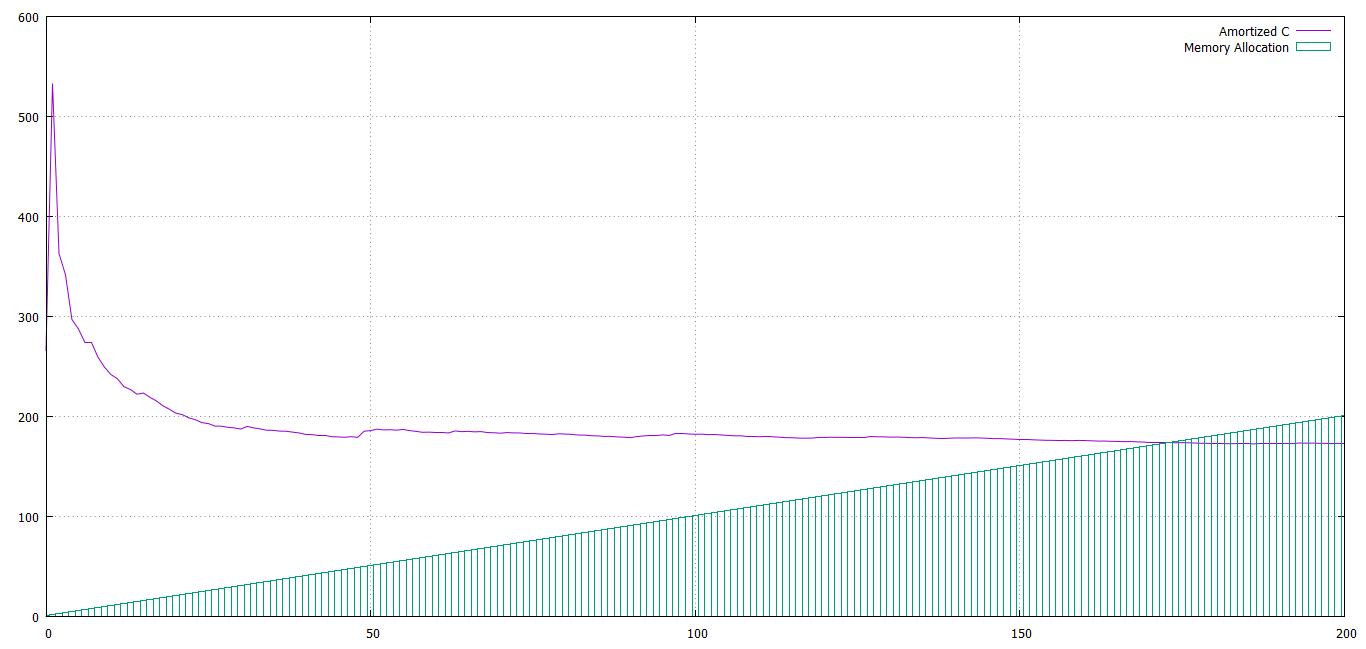


Figure  : Coût amorti et mémoire utilisée pour l’insertion des clés dans l’ordre décroissant

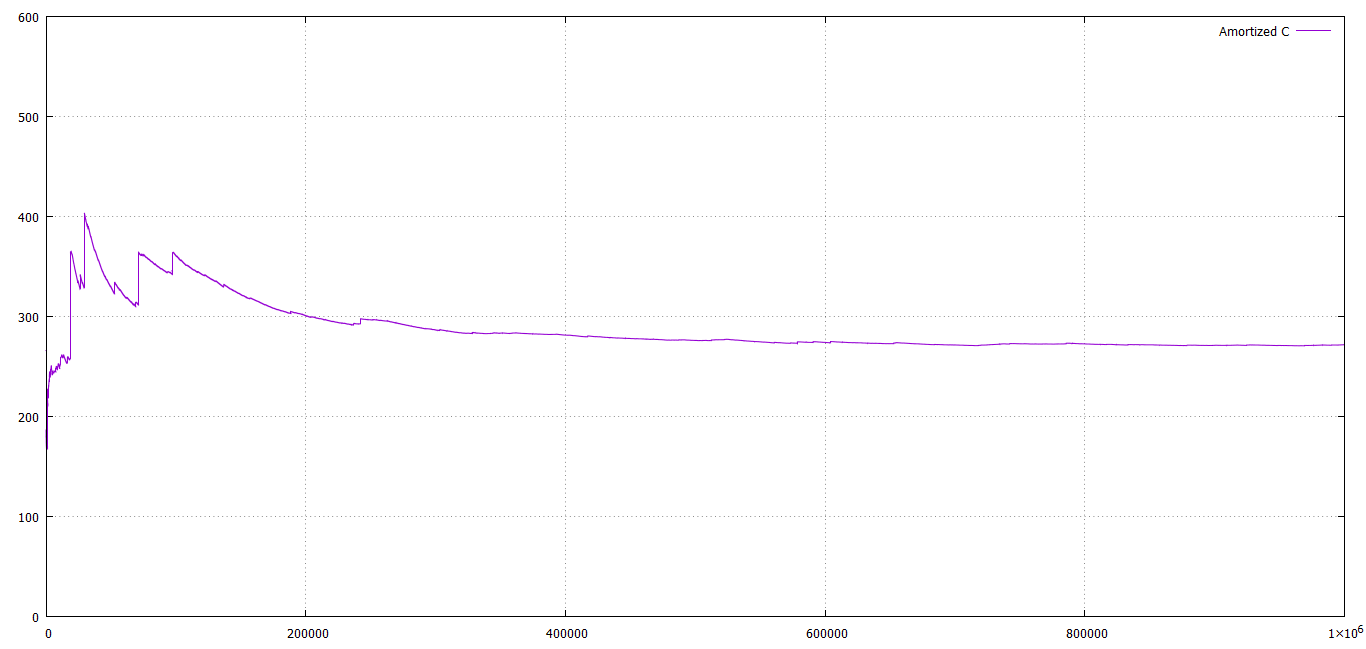
****

Figure  : Coût amorti de l’insertion des clés dans l’ordre décroissant

On a aussi affiché le coût réel des insertions, on remarque d’après ces graphes qu’il est bien plus perturbé que le coût amorti, ce qui est normal car en théorie le coût amorti est constant, quant au coût réel il dépend de l’état de la structure et change d’une opération à une autre.

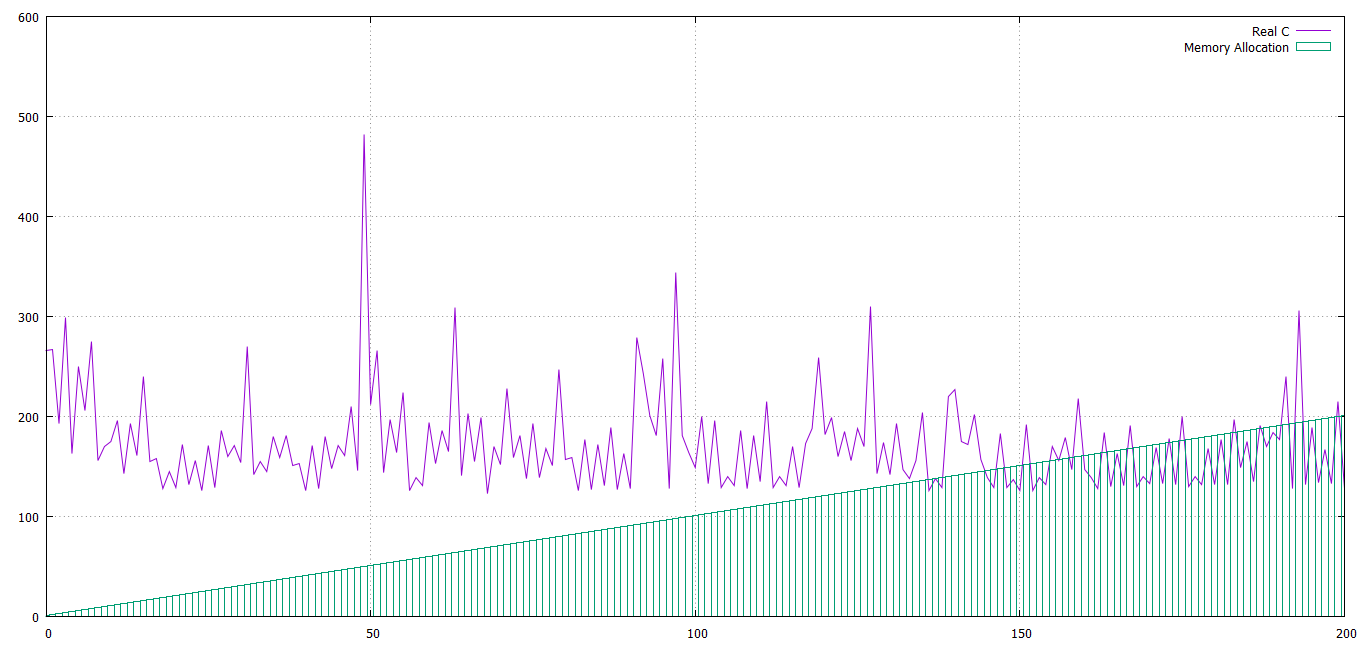


Figure  : Coût réel et mémoire utilisée pour l’insertion des clés dans l’ordre décroissant

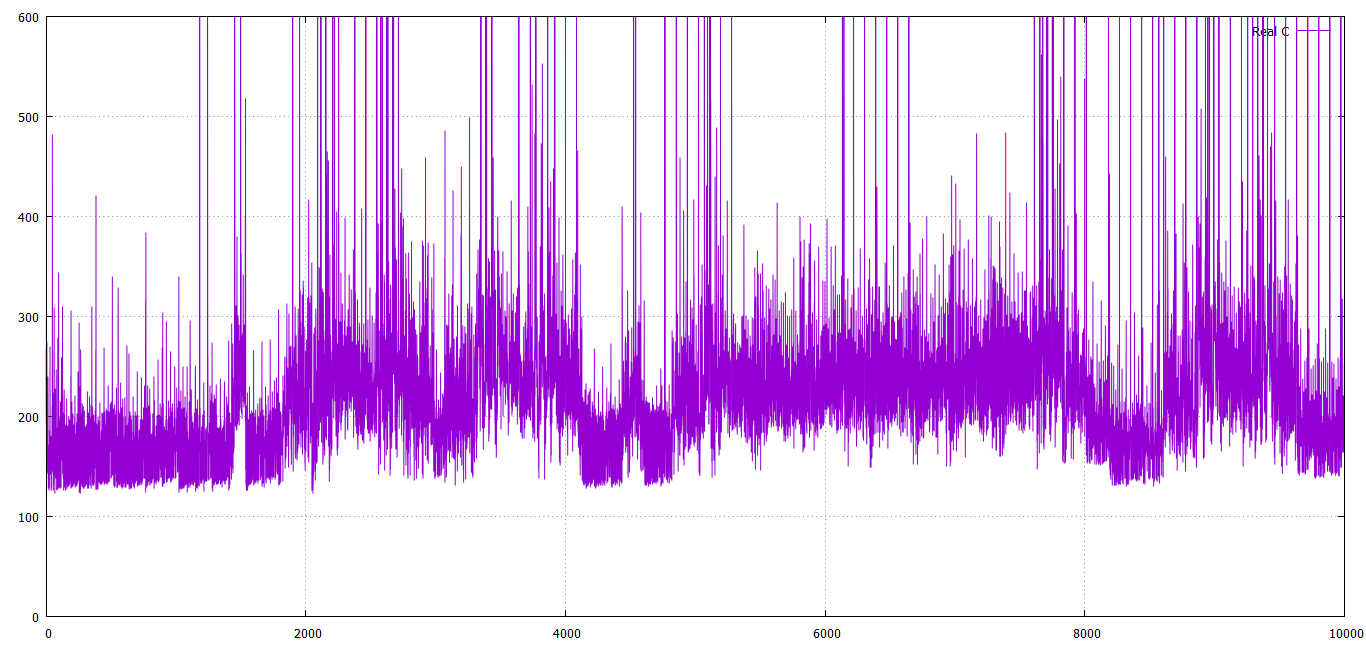


Figure  : Coût réel de l’insertion des clés dans l’ordre décroissant

**Conclusion :**

En analysant le coût amorti de ces différentes expériences et en le comparant au coût amorti obtenu dans l’analyse des tas binaires, on voit que le coût amorti de l’insertion sur les tas binomiaux ne dépasse pas celui de l’insertion sur les tas binaires, soit il est en dessous soit ils sont pareils  et pourtant on pourrait penser que les tas binomiaux sont plus couteux car pour insérer on fait appel à « Union\_Binomial » qui elle même fait appel à deux autres fonctions. Mais ce n’est pas le cas car justement c’est le point fort de cette structure, les tas binomiaux, comme on l’a dit au début, sont aussi appelés tas fusionnables car on peut faire la fusion de deux tas en seulement O(logn) alors que les tas binaires sont peu efficaces car la concaténation des deux tableaux contenant les tas binaires à fusionner, puis l’exécution de ENTASSER font que UNION prend un temps Q(n) dans le cas le plus défavorable.

Mais en ce qui concerne la recherche du minimum, les tas binaires sont bien plus efficaces car il suffit de renvoyer la racine, c’est donc en O(1), par contre pour les tas binomiaux il faut parcourir la liste des racines pour trouver la minimale ce qui se fait en O(logn).

Les tas binomiaux sont donc meilleures que les tas binaires dans le cas de l’union seulement, on les utilise donc quand on veut faire de l’union, par exemple quand on veut fusionner deux processus pour en faire un (car les tâches des processus sont souvent gérées pas des files de priorité et donc par des tas min), les tas binomiaux sont aussi utilisés pour résoudre le problème de l’arbre couvrant de poids minimal dans un graphe non orienté.

Par contre, tous comme les tas binaires, les tas binomiaux sont inefficaces en ce qui concerne l’opération RECHERCHER.

**Pour aller plus loin :**

Les tas binomiaux sont dis avantageux car ils sont utilisés comme élément constitutif dans d'autres structures de données (tas de Fibonacci, tas mous, etc.)

Les tas de Fibonacci sont parfaitement adaptés lorsque le nombre d’opérations EXTRAIRE\_MIN et SUPPRIMER est petit par rapport aux autres opérations.

En effet les tas de Fibonacci, supportent les mêmes opérations que les tas binomiaux mais avec l’avantage supplémentaire que les opérations qui ne nécessitent pas la suppression d’un élément s’exécutent avec un temps amorti O(1).