|  |
| --- |
|  |
| Structure de données  Tas binaires statiques |
|  |
|  |
| **Lynda MEDJDOUBI**  **Sabrina DJENADI** |
|  |

|  |
| --- |
|  |

**Sommaire**

**Insertion des clés dans l’ordre croissant4**

**Insertion des clés dans l’ordre décroissant6**

**Insertion des clés aléatoires** **8**

**Insertion et extraction des clés10**

**Conclusion11**

**Pour aller plus loin12**

**Choix d’implémentation :**

Pour l’implémentation des tas binaires statiques on va utiliser un tableau d’entiers.

On a commencé par définir une structure pour chaque nœud, cette structure contient un pointeur vers le tas, un pointeur vers un tableau d’entiers, qui va contenir les clés des nœuds, une capacité ainsi qu’un entier size représente le nombre d’élément dans le tableau, il est utilisé pour savoir quand est ce que le tableau est rempli, pour afficher un message d’erreur par exemple lors de l’insertion.

Ensuite on a définit une suite de fonction dont on a besoin pour faire l’analyse de cette structure.

**1.createTas** : cette fonction alloue de la mémoire pour le pointeur du tas ainsi que pour le tableau contenant les clés.

**2.tas\_insert :** cette fonction permet d’insérer un entier x dans un tas et teste si l’élément est plus petit que son père alors il va être entasser vers le haut en utilisant la fonction « reorganise\_tas ».

**3.extract\_min :** elle permet d’extraire la racine du tas qui est aussi l’élément le plus petit **et** cela en le remplacant par le dernière élément du tableau, qui sera ensuite entasser vers le bas avec la fonction « Entasser ».

dans cette Première partie du TP3 on a commencé à Développez une structure 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. Cette structure de tas nous permet d’ajouter une clé, et d’extraire la plus petite clé contenue dans le tas.

Ensuite pour effectuer des expériences sur l’efficacité en temps et en mémoire de notre structure, on lui a ajouté l’Analyzer déjà utilisé dans le TP1 en apportant quelques modifications, pour pouvoir réaliser ces expériences.

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

On a affiché le cout réel de l’opération d’insertion des valeurs successives ordonnées de façon croissante, on remarque que le coût réel rencontre des augmentations légères à chaque insertion d’une valeur et cela varie entre 100 et 200, ce qui revient au fait que les valeurs sont insérées dans un ordre croissant autrement dis on n’a pas besoin d’entasser notre tas à chaque fois ce qui nous coute moins en efficacité. La figure ci-dessous nous montre clairement les petites perturbations du graphe lors des insertions.

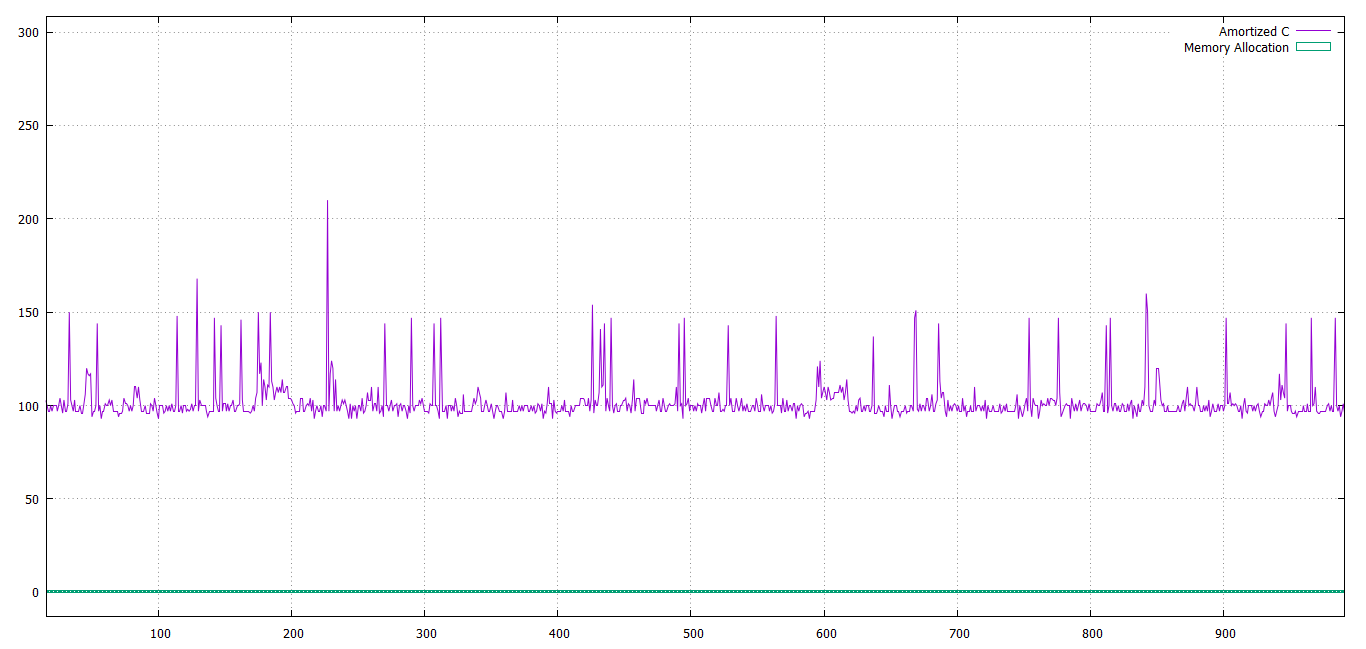
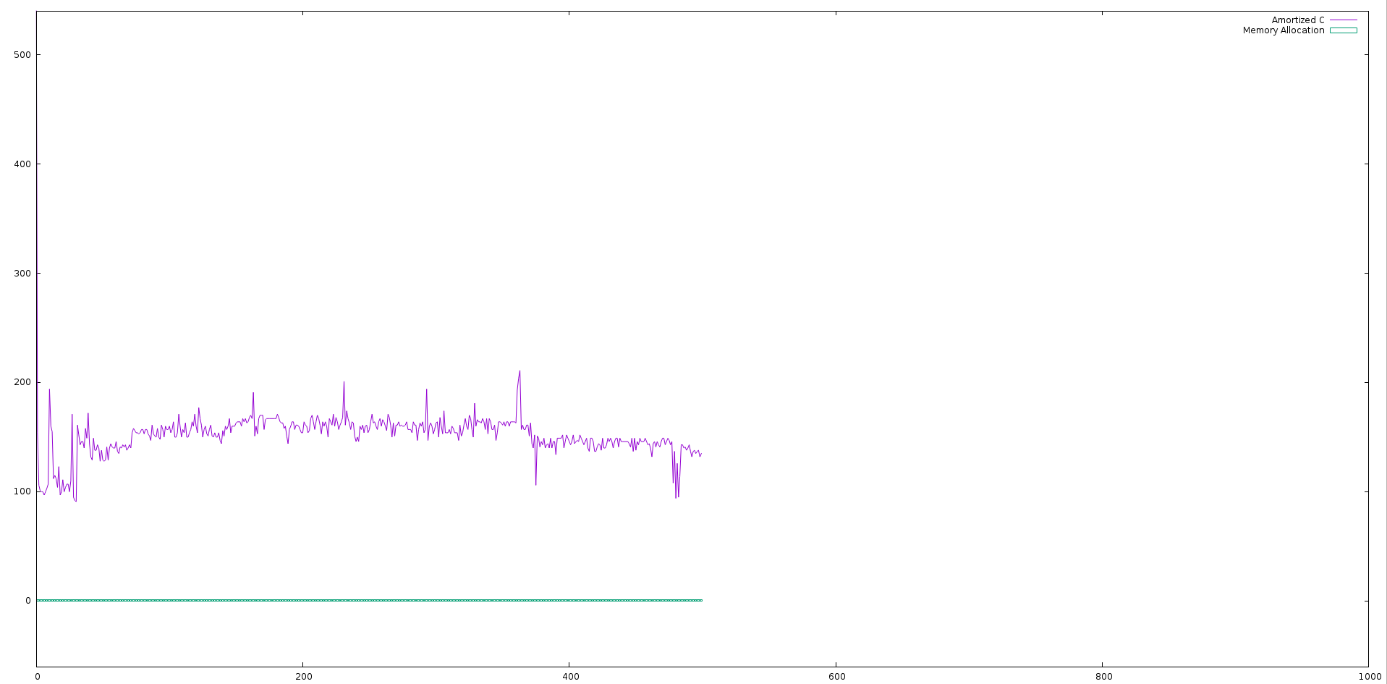


Figure  : Coût réel de l’insertion des clés dans l’ordre croissant

Figure  : coût réel de l’insertion des clés dans l’ordre croissant

Ensuite on a affiché le coût amorti pour la même opération d’insertion avec des valeurs successives croissantes, on remarque que le coût amorti au début est décroissant après avoir connu une valeur très élevée pour se stabiliser a une valeur entre 100 et 150 moins que le cout réel ce qui est normal et on remarque qu’on a moins de perturbation du graphe jusqu’à ce que on ne rencontre plus d’augmentations. Vu qu’on n’a pas besoin d’entasser notre tas à chaque fois qu’on fait une insertion ce qui nous coute moins en efficacité. La figure ci-dessous nous montre les variations du cout amorti présentées sous forme de graphe lors des insertions.

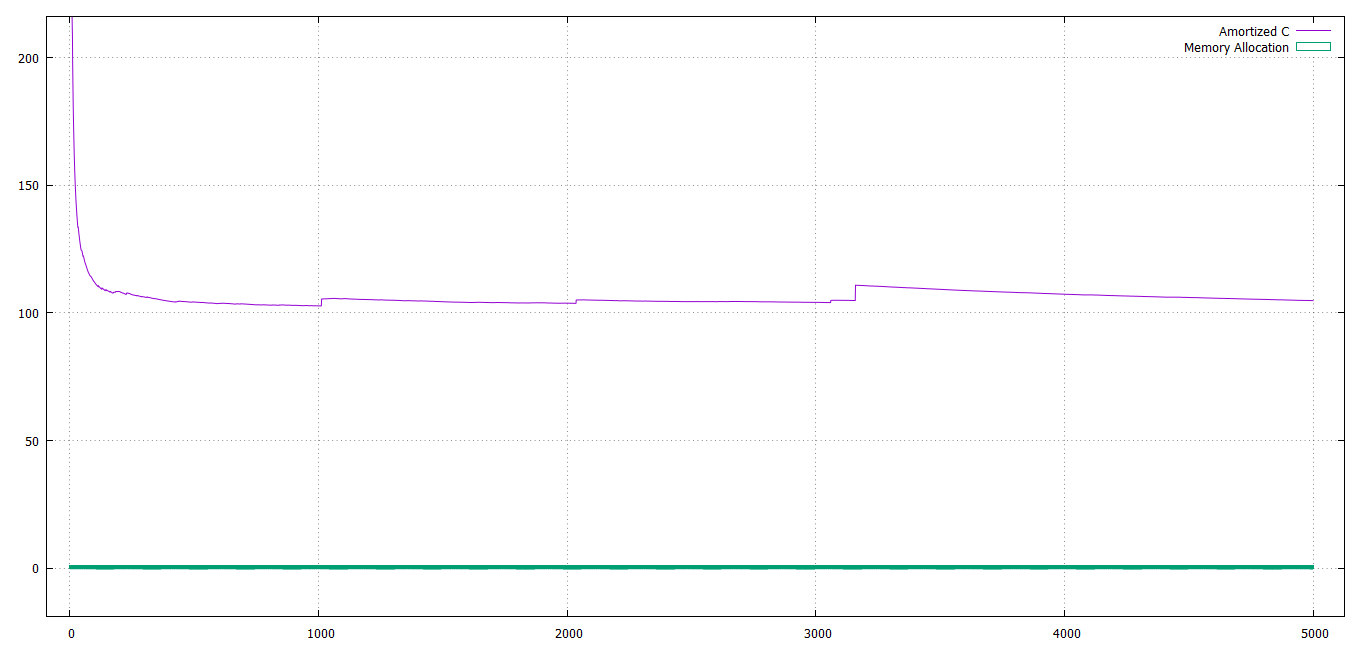


Figure  : Coût amorti de l’insertion des clés dans l’ordre croissant

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

Nous avons effectué des insertions successives des clés dans l’ordre décroissant.

Pour observer le coût amorti de ces insertions nous avons affiché les résultats de l’analyse sur gnuplot sous forme de graphe.

On remarque dans ce graphe que le coût amorti augmente au fur et à mesure que les clés sont insérées, cette augmentation est du au fait que les valeurs insérées sont décroissantes. En effet quand on insère à chaque fois une clé plus petite que la précédente alors la fonction « réorganise tas » est exécuté ce qui coûte cher car dans le pire des cas on devra parcourir toute la hauteur de l’arbre (O(logn)) pour entasser vers le haut le nouvel élément.

En ce qui concerne l’allocation mémoire, celle-ci est toujours nulle car on travaille sur des tas binaires statiques.

On pense que cette méthode d’insertion n’est pas très pratique car elle coûte chère en temps d’exécution.

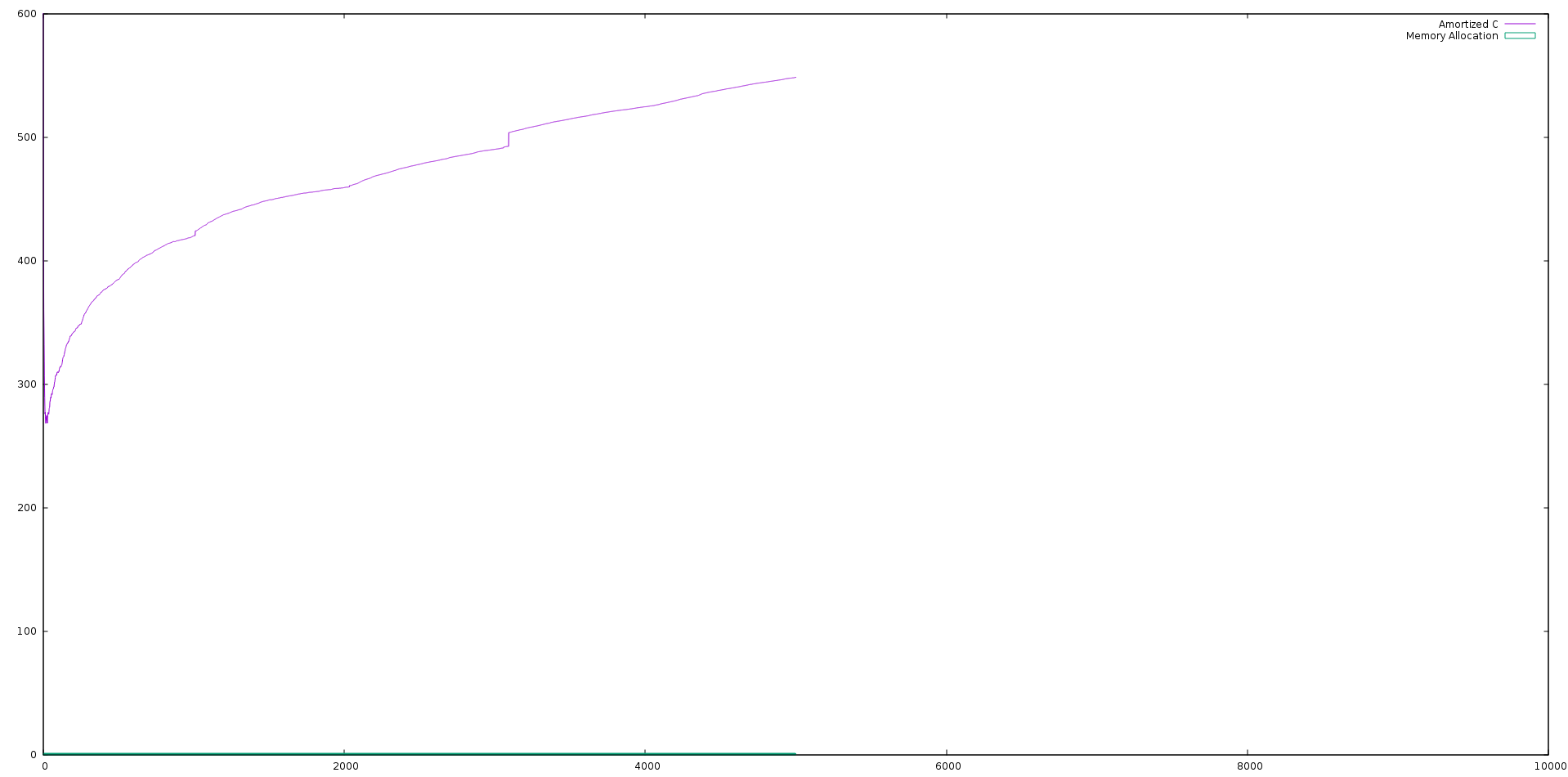


Figure 4 : Coût amorti de l'insertion des clés décroissantes

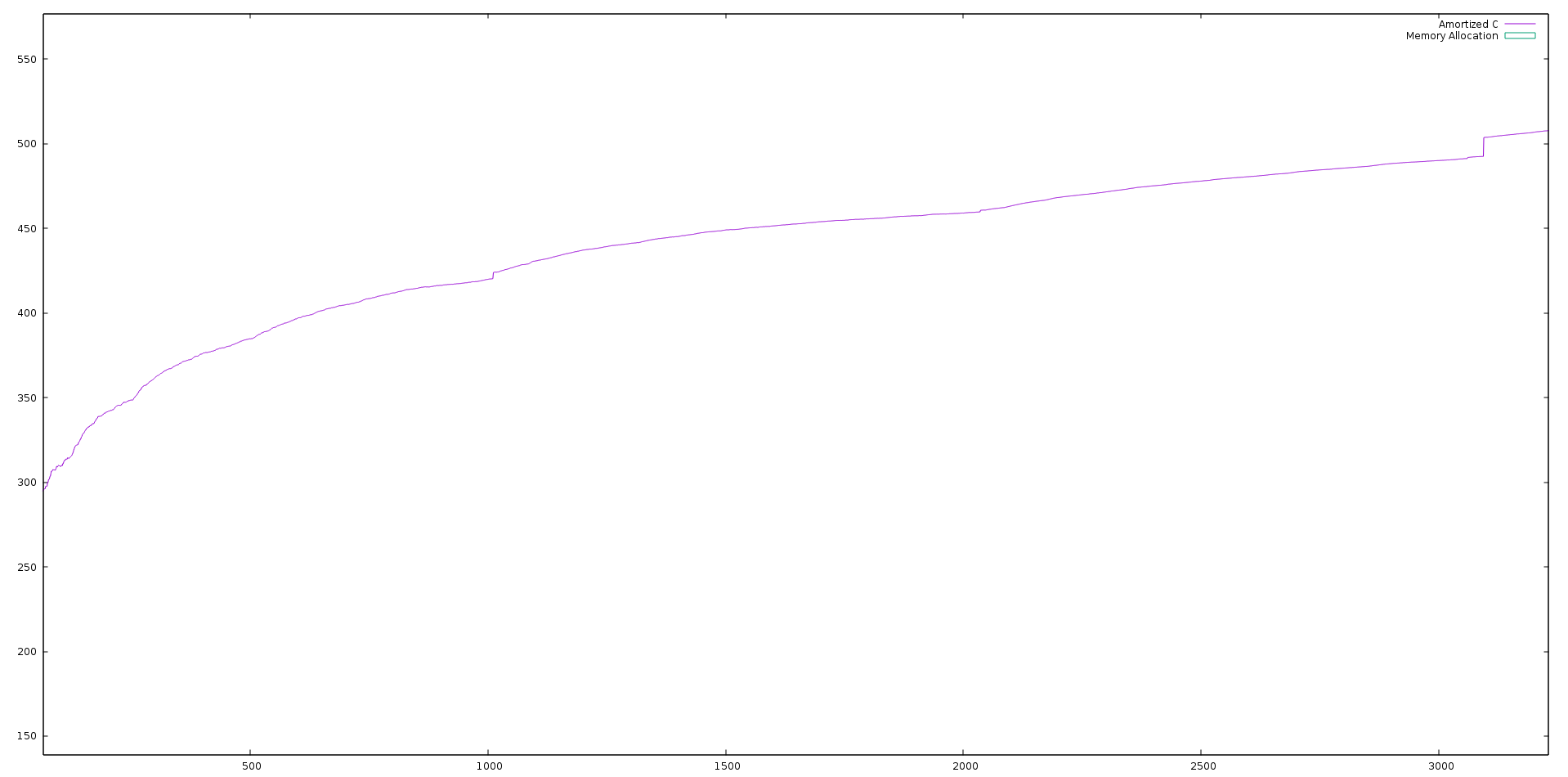


Figure 5 : coût amorti de l’insertion des clés dans l’ordre décroissant

On a aussi affiché le coût réel de ces opérations et on a obtenu le graphe suivant :

On remarque que le coût réel augmente très souvent et cela à chaque nouvelle insertion.

Le résultat est cohérent car on sait que à chaque fois qu’on insère un nouvel élément dans le tas celui-ci doit être entassé vers le haut car il est plus petit que le précédent, et on remarque aussi qu’au départ les valeurs du coût réel sont entre 200 et 300 et après plusieurs insertions les valeurs ont beaucoup augmenté et ont dépassé 600, car la hauteur du tas augmente.

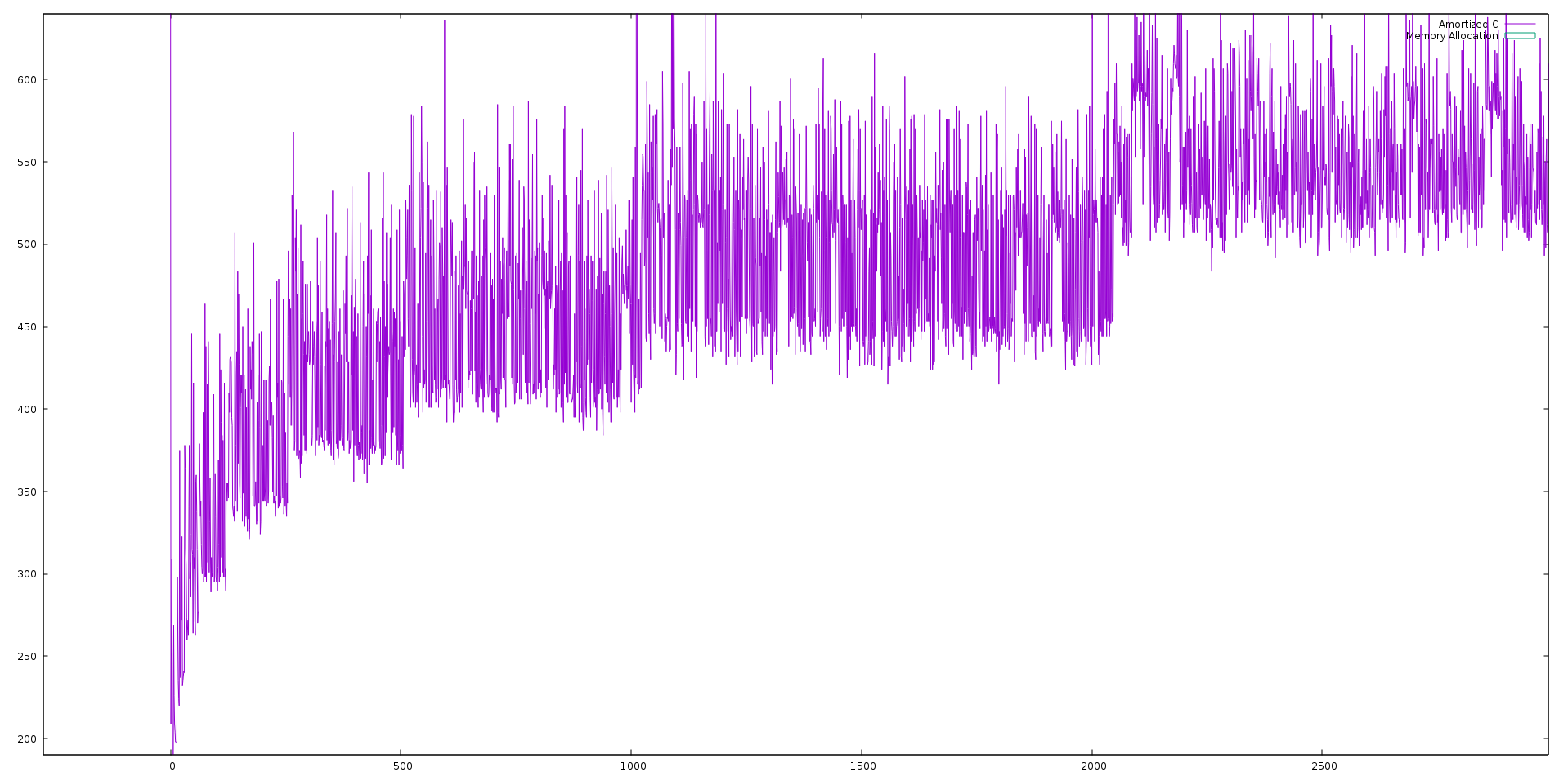


Figure  : Coût réel de l’insertion des clés dans l’ordre décroissant

**Insertion des clés d’une façon aléatoire :**

Pour effectuer cette expérience on a modifié notre main du tas binaire en utilisant randum pour en insérer des valeurs aléatoires, puis on a observé les variations du coût réel et du coût amorti de cette opération.

On remarque que le coût réel est très perturbé il augmente énormément lors de l’insertion des valeurs pour atteindre une valeur très grande, surtout au moment d’appelle à entasser, quand la valeur à insérer est plus petite que la racine alors le coût réel devient plus grand en faisant la permutation de plusieurs valeurs.

Après avoir fait plusieurs insertions la hauteur du tas devient plus grande ainsi on risque de parcourir toute la hauteur du tas pour le réorganiser, c’est ce qui fait que la valeur du coût réel après plusieurs insertions atteint une valeur qui dépasse 600 ; ce qui est trois fois plus grand que la valeur du coût réel de l’insertion des valeurs croissantes. Cette méthode n’est pas efficace comparé à l’insertion effectuée dans le cas précédent.

La figure ci-dessous nous montre clairement l’immense augmentation du cout réel de l’insertion des valeurs aléatoires :

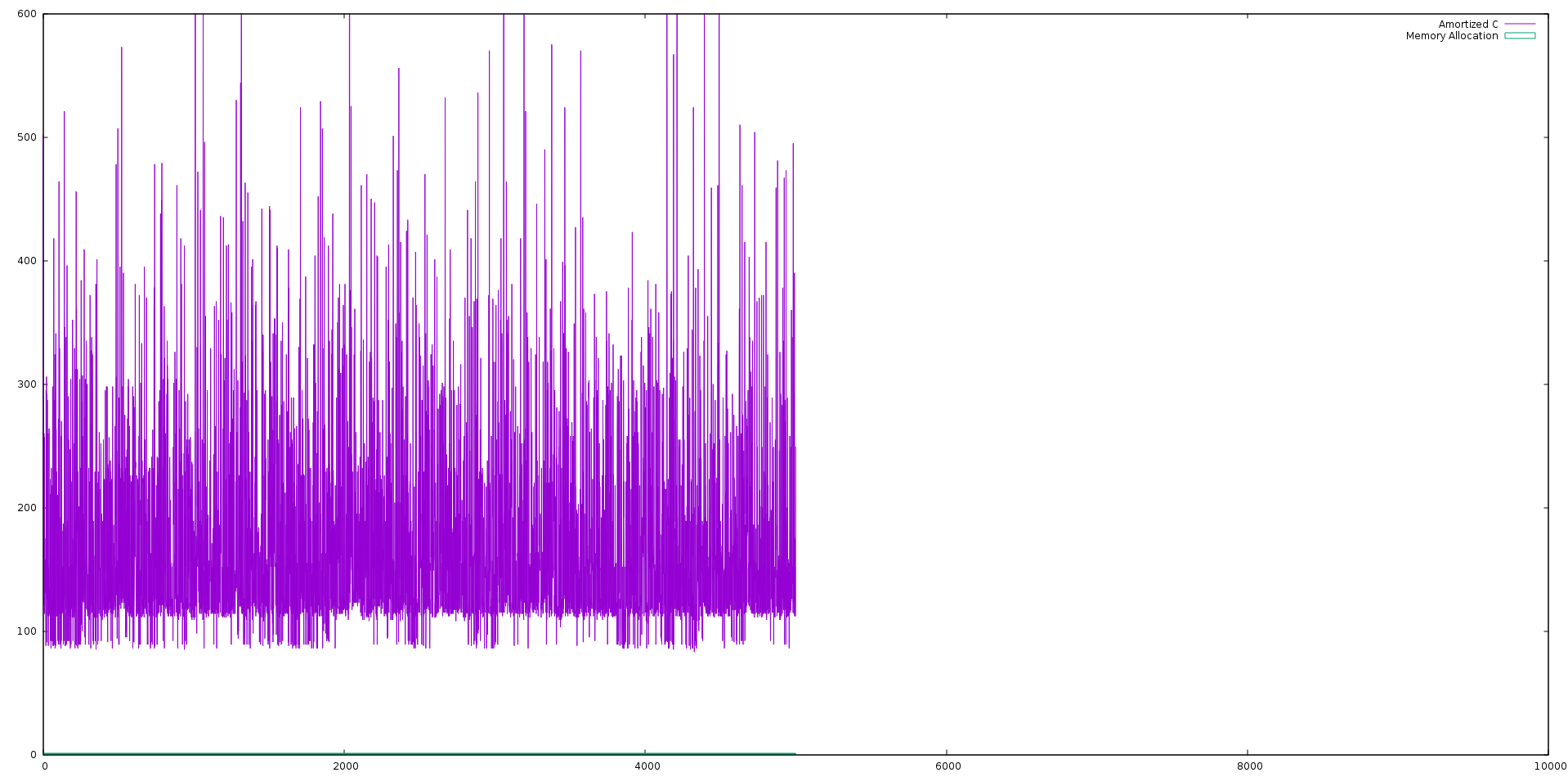
****

Figure : Coût réel de l’insertion de clés aléatoires

De la même manière on a analysé le coût amorti de l’insertion des valeurs aléatoires, on a remarqué que la valeur du coût amorti à augmenter comparée au cas où on fait des insertions des valeurs croissantes et cette fois ci elle varie entre 150 et 200.

On justifie cette différence par le fait que l’insertion des valeurs est aléatoire à chaque fois qu’on insère y a possibilité d’appeler entasser pour réorganiser le tas ce qui nous coûte plus chère. Ce qu’on remarque aussi de nouveau dans ce cas c’est qu’on rencontre des augmentations légères à long termes même si le graphe se stabilise à un certain moment dans un intervalle très petit, en parallèle de l’exécution de l’opération insérer on a des programmes qui s’exécutent aussi ce qui fait apparaitre ce phénomène de petite augmentations. Ainsi Après avoir fait plusieurs insertions la hauteur du tas devient plus grande et les permutations deviennent plus coûteuses pour réorganiser notre tas, c’est ce qui fait que la valeur du coût amorti après plusieurs insertions augmente comparé à notre précédente expérience de l’insertion des valeurs croissantes.

La figure ci-dessous nous montre l’augmentation du cout amorti de l’insertion des valeurs aléatoires :

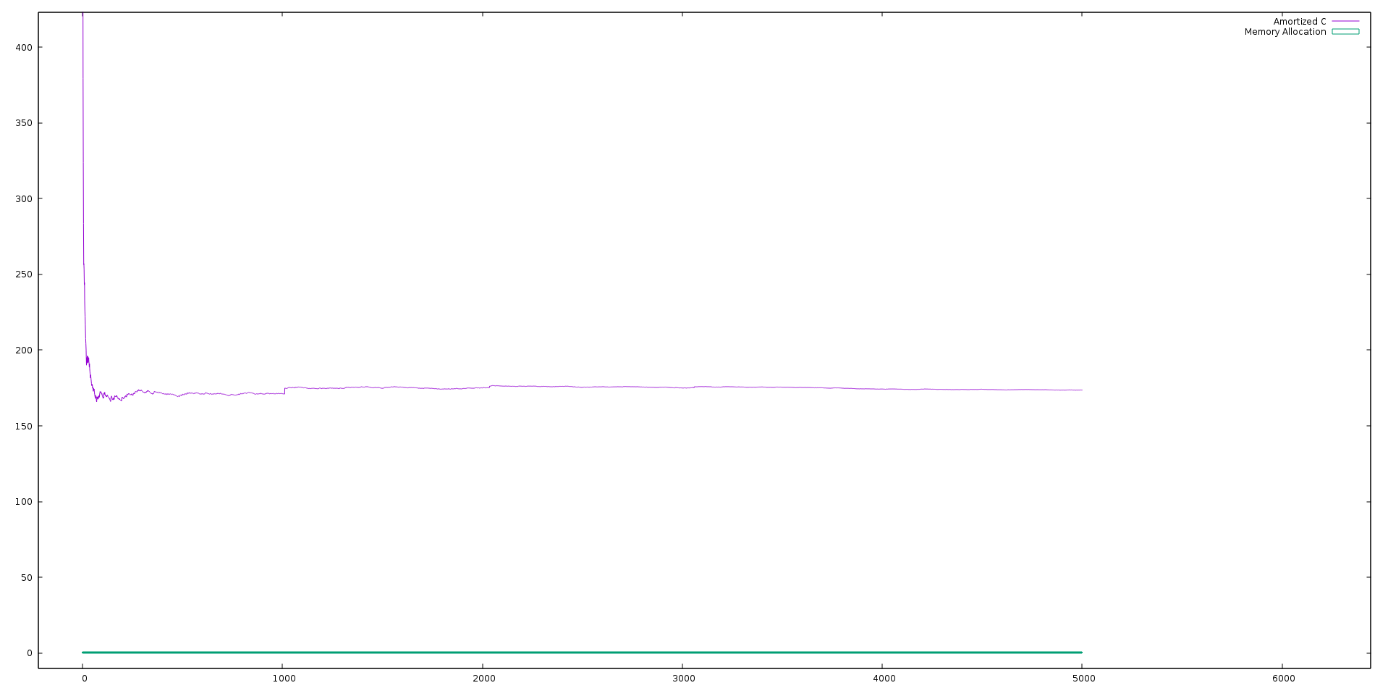
****

Figure  : Coût amorti de l’insertions des clés aléatoires

**Insertion et extraction des clés :**

Dans cette étape nous avons fait des insertions et suppressions sur le tas en utilisant un nombre aléatoire.

Pour analyser cette opération on a affiché leur coût amorti.

On remarque que le coût amorti ici n’est pas aussi élevé que dans l’expérience précédente.

Comment expliquer cela ? Tout simplement dans cette expérience on fait soit une insertion de clés dans l’ordre croissant, ce qui, comme nous l’avons vu dans la première partie, ne coûte pas très cher, soit une extraction du min, par conséquent la hauteur du tas n’augmente pas trop et coût amorti non plus.

Mais si on compare ce graphe à celui où on ne fait que des insertions de clés dans l’ordre croissant, on remarque qu’il ya une différence assez évidente. En effet le coût amorti ici est plus élevé car en faisant l’extraction du minimum on doit faire appel à la fonction « entasser », qui doit réorganiser le tas en remplaçant le min par la dernière valeur, pour ensuite l’entasser vers le bas, on doit alors dans le pire des cas parcourir toute la hauteur du tas pour faire cette opération, ce qui explique l’augmentation du coût amorti.

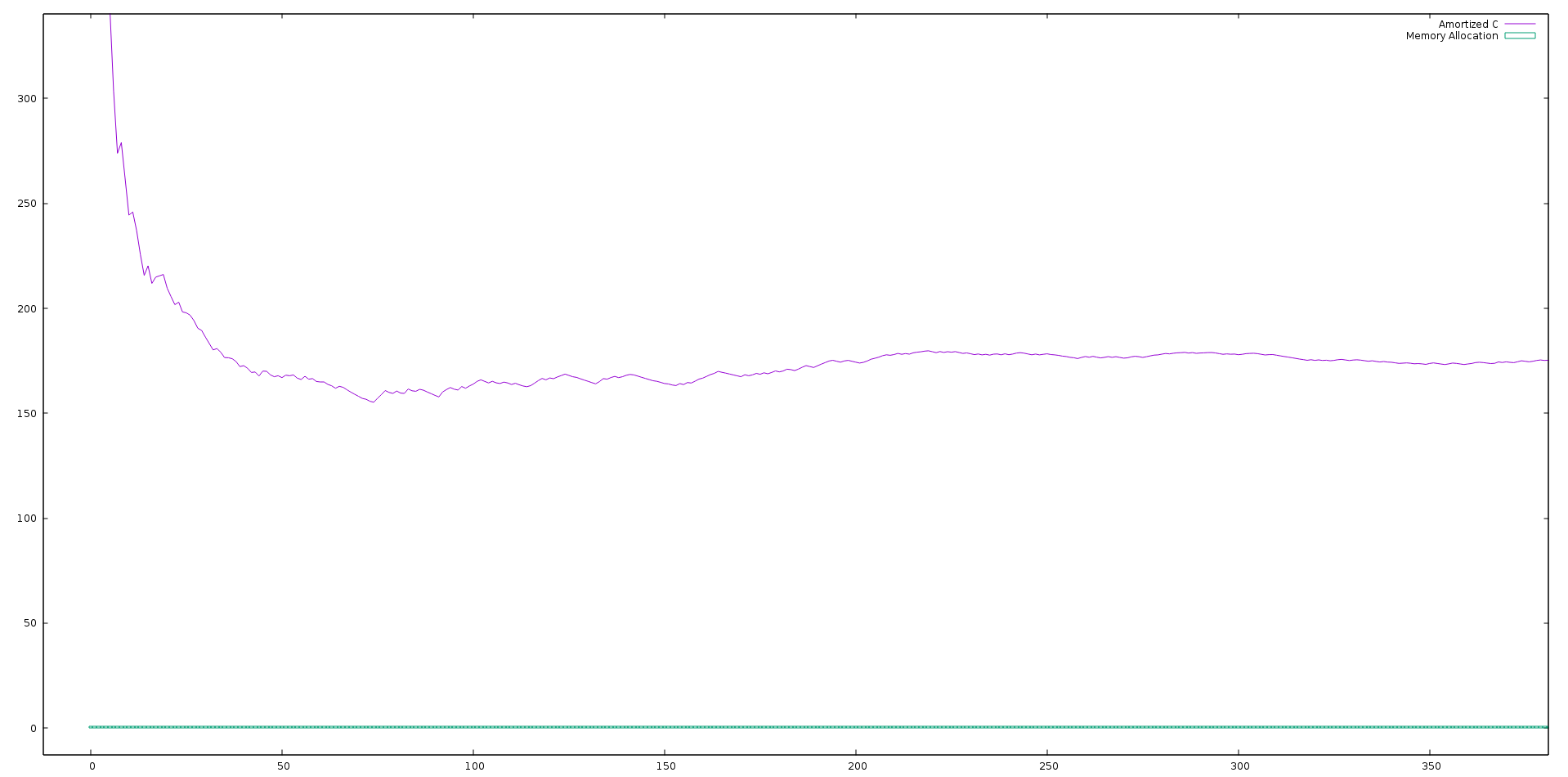
****

Figure 8 : coût amorti des insertions et extractions

Comme pour l’expérience précédente, nous avons affiché, ici aussi, le coût réel des ces insertions et suppressions. A la différence du coût amorti, le coût réel n’est pas stable (constant). En effet on remarque dans ce graphe que le coût réel augmente par fois (et cela se produit lors d’une suppression), mais il baisse aussi (et cela arrive quand on fait une insertion).

En comparant ces résultats à ceux obtenus lors de l’insertion des clés dans l’ordre décroissant, on remarque que l’insertion décroissante des clés coute bien plus cher, car dans cette expérience malgré qu’on doit parcourir toute la hauteur du tas (dans le pire des cas) pour faire une suppression , le fait qu’on fasse des insertions de clés dans l’ordre croissant fait que le coût réel ne soit pas très élevé.

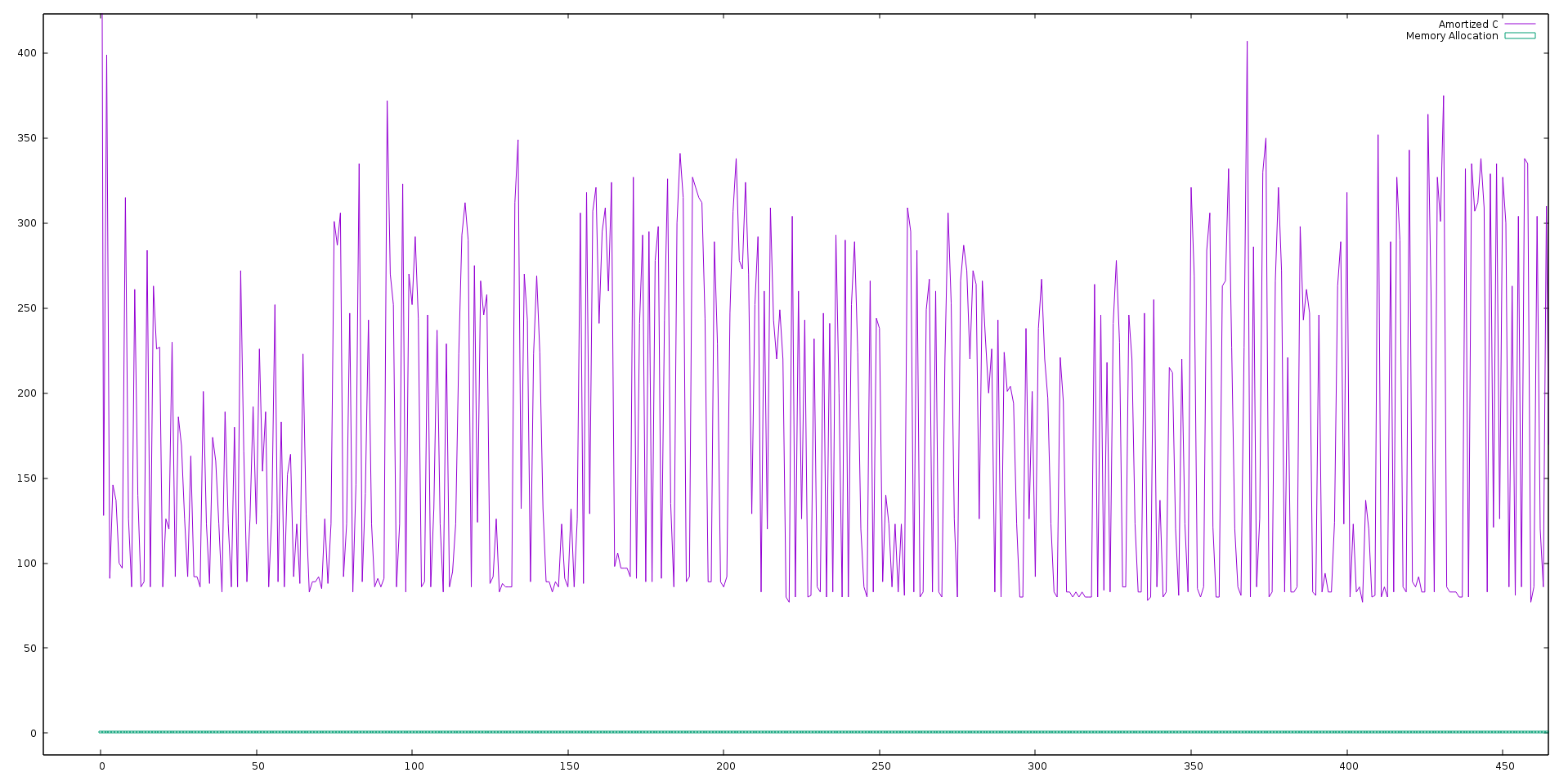


Figure 9 : coût réel des insertions et extractions

**Conclusion :**

Pour résumer on peut dire que le coût amorti et réel sont moins élevés quand on fait une insertion de clés dans l’ordre croissant, mais cela ne nous intéresse pas autant car le but principale dans l’utilisation d’un tas min c’est la simulation d’une file de priorité, or dans une file de priorité on doit pouvoir insérer des éléments dans la file mais aussi on doit extraire l’élément le plus prioritaire, qui est dans notre cas, l’élément de clé minimale. On s’intéresse donc à l’insertion et à l’extraction et c’est ce que nous avons testé dans la dernière expérience et nous avons remarqué que le coût amorti et réel ne sont pas très élevés.

En ce qui concerne l’espace mémoire utilisé il n’est pas nécessaire de l’analyser, car c’est nous qui le fixons dés le départ, c’est d’ailleurs le point embêtant de cette structure car pour pouvoir l’utiliser on doit connaitre à l’avance la taille des données qu’on va stocker, or dans la vie pratique ce n’est pas souvent le cas, une autre solution serait de réserver un espace mémoire très grand mais au risque d’avoir beaucoup d’espace mémoire inutilisé. Comment faire alors ? Bien évidemment ce problème n’est pas sans solution, et parmi les meilleures on a ce qu’on appelle les tas dynamiques, en quoi ces tas sont meilleures ? C’est ceux à quoi nous allons essayer de répondre dans notre rapport sur « Les binaires dynamiques ».

**Pour Aller plus loin**

On a parlé dans ce rapport des tas min, mais ce n’est pas l’unique type de structure de tas qui existe, il y a aussi ce qu’on appelle les tas max, c’est le même principe que pour les tas min sauf qu’ici, l’élément de la racine est le plus grand et chaque père est plus grand que les éléments de ses sous arbres.

Cette structure peut aussi être utilisée pour la simulation d’une file de priorité, mais son utilisation la plus fréquente est celle du tri par tas.

Le tri par tas peut être découpé en 2 grandes étapes :

1. Construire le tas à partir du tableau que l’on souhaite trier.

2. Utiliser le tas pour trier le tableau. Pour cela il suffit d’échanger la racine du tas (qui est la plus grande valeur) avec la dernière cellule du tableau pour que la plus grande valeur soit définitivement rangée. L’élément est ainsi à sa place mais le tas privé de sa dernière case ne respecte pas les propriétés d’un tas. Il est alors nécessaire d’appeler entasser sur le tableau privé de la dernière case. Il suffit ensuite de recommencer jusqu’à ce que le tas à considérer soit vide.

Et on obtient alors un tableau trié par ordre croissant. La complexité de cet algorithme est O(n) + n × O(log2 (n)) = O(n ln(n)) car la racine est de hauteur ⌊log2 (n)⌋ et O(n) correspond au nombres de valeurs à trier.