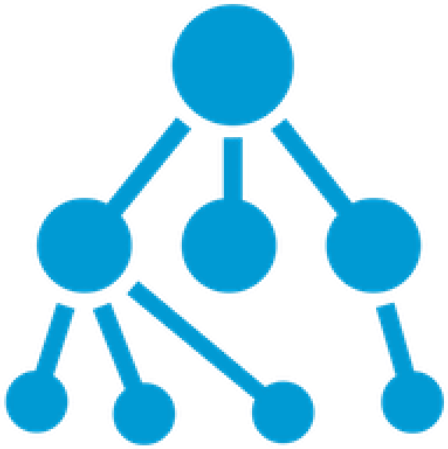
UNIVERSITÉ PARIS 13 | USPC Université Sorbonne Paris Cité

Institut Galilée

Rapport des Tps Structures de données.



Réaliser par :

-Maher BEN AYED

-Yousra REGRAGUI

Introduction

Une structure de données est un format spécial destiné à organiser, traiter, extraire et stocker des données. S'il existe plusieurs types de structures plus ou moins complexes, tous visent à organiser les données pour répondre à un besoin précis, afin de pouvoir y accéder et les traiter de façon appropriée.

Durant les tps demandées dans notre cours structures de données avancées, on a manipulé plusieurs structures de données comme les tas binomiaux, les tas binaires, les B_arbres, etc …, on a effectué aussi des expériences sur le temps, le mémoire et le cout amorti pour chaque structure après certaines opérations comme l'insertion ou suppression.

Dans ce rapport on va détailler les résultats et les diagrammes obtenus avec plot en expliquant ce qu'on a remarqué entre les différents résultats.

Tp1 : Tables Dynamiques

1- **Dans le cas où on ne double pas la taille de la table mais on en multiplie par un facteur** $\alpha \geq 1$**, la fonction potentielle s'écrit**

O(n)=2*n-capacité

Tel que : n est le nombre d'élément

Et capacité est la taille du tableau

2- **Le coût amorti de l'opération insérer-table en fonction de** α.

A(n)=C(n) +O(n)-O (n-1)

C(n) : coût pour la nième opération
A(n) : coût amorti pour la nième opération
O(n) : fonction de potentiel

On arrondit le tableau d'une constante α avec $\alpha>1$ (n-> αn)

O(n) = (αn-capacité)/ α-1

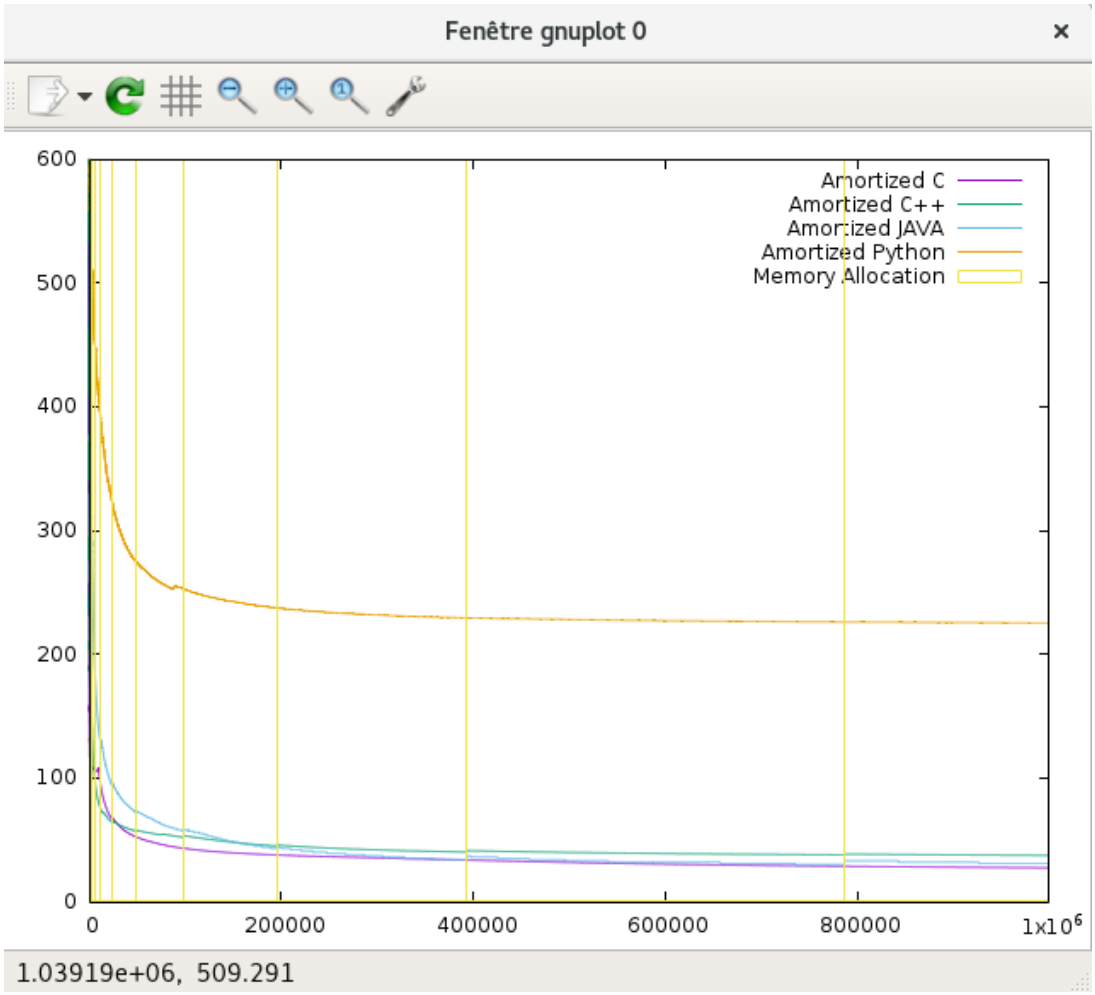
-Table n'est pas pleine :

A(n)=1+(αn-capacité/ α-1) – (α(n-1)- α(n-1)/ α-1)=1+ (α/ α-1)=O(1) (car α=O(1))

-Table est pleine :

A(n)=n+ ((αn- αn)/ α-1)-((α*(n-1)-n)/ α-1) = (αn-n- αn+ αn-n)/ α-1= (α/ α-1) =O(1)

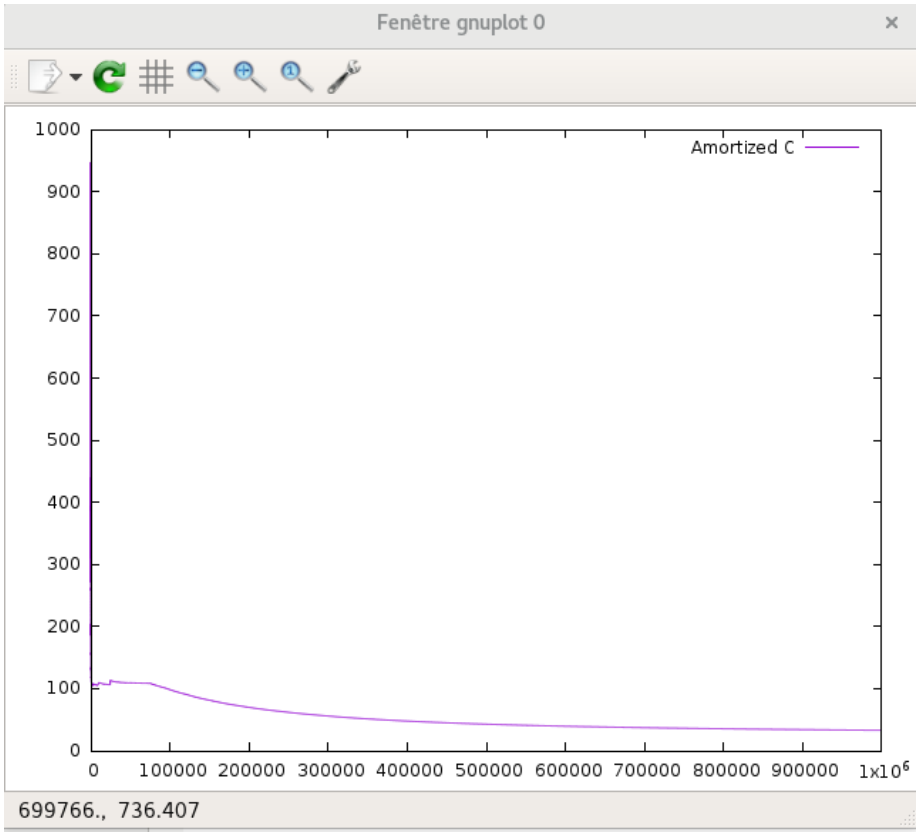
3- Après la suite des instructions dans le fichier README.md on a obtenu le schéma suivant :



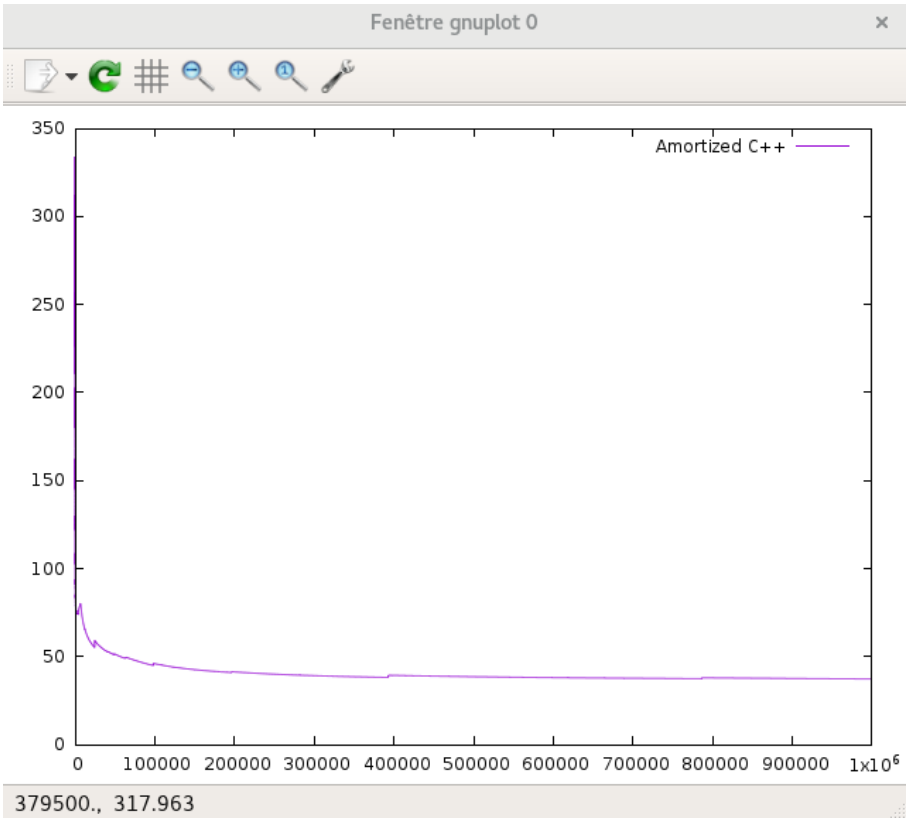
a) Le morceau de code qui prend le plus de temps d'exécution est le sauvegarde des données; Puisqu' on compile sur la mémoire et le code est écrit sur le disque dur donc cela prend du temps pour l'exécution.

b) Le coût amorti augmente quand on fait que des insertions donc nous sommes obligé d'élargir la taille de notre table.

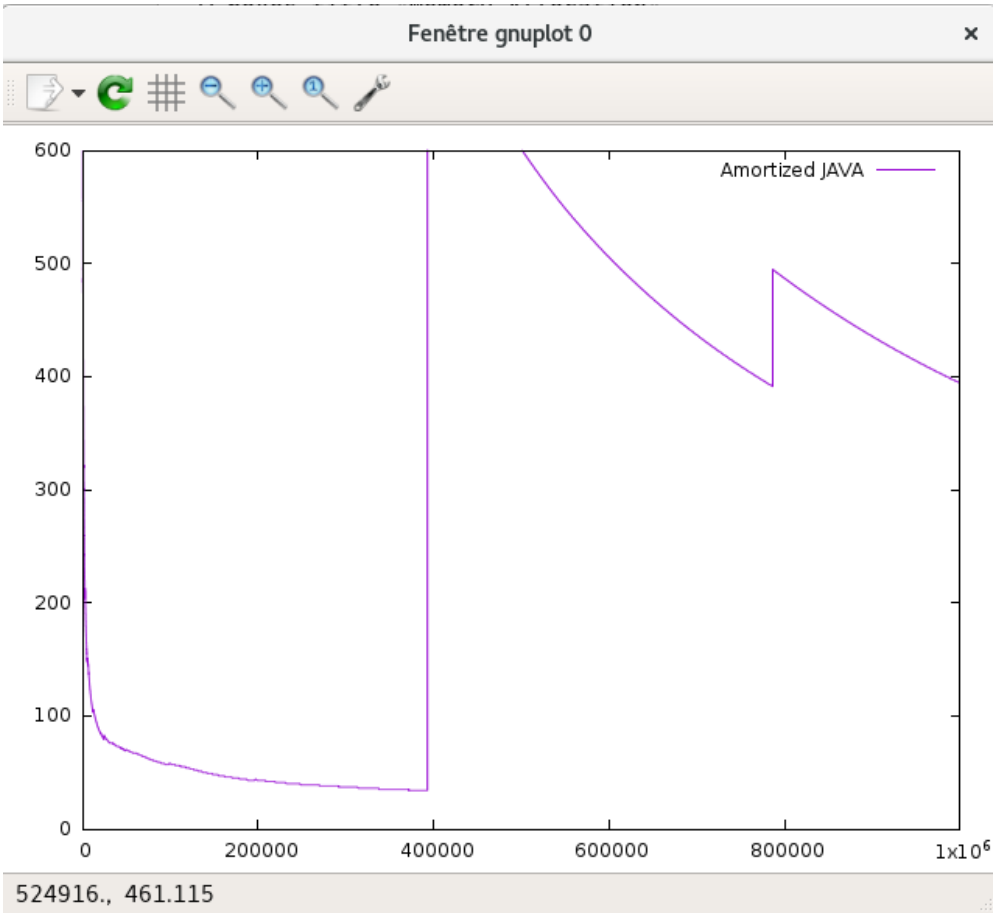
c)



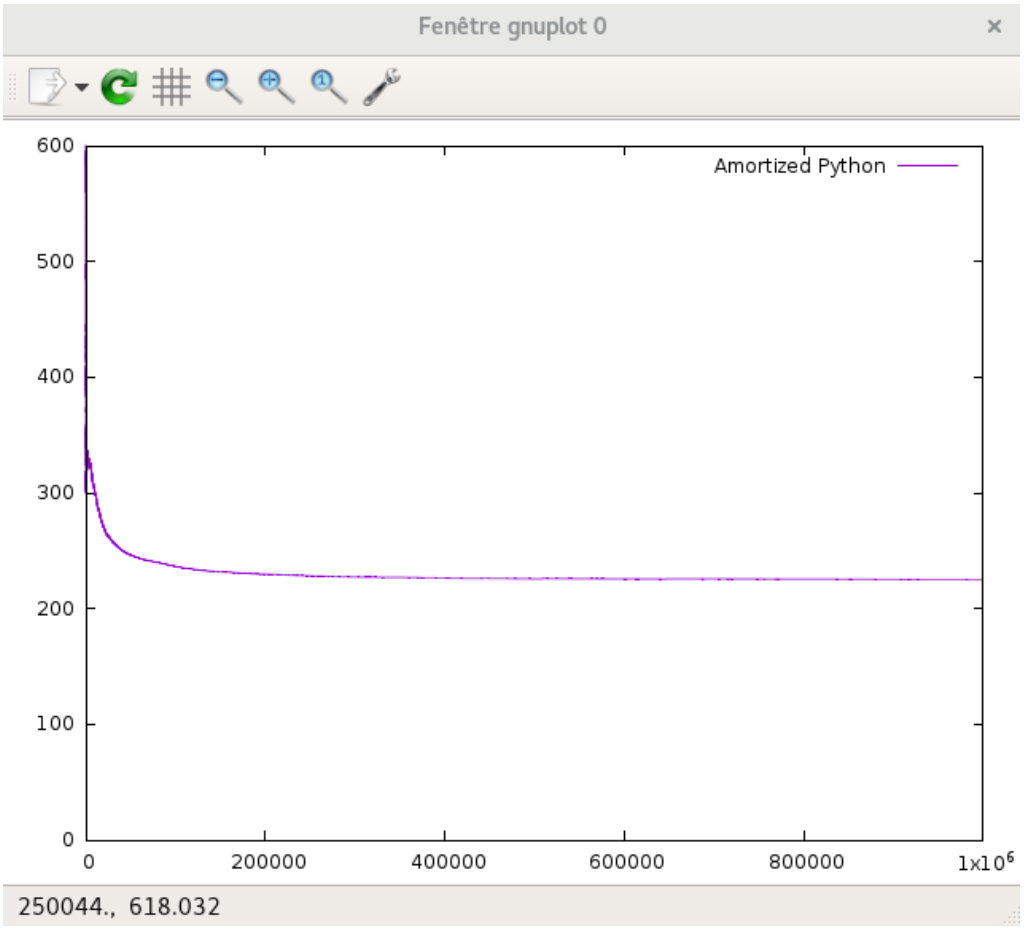
Le coût amorti pour le langage C



Le coût amorti pour le langage CPP



Le coût amorti pour le langage Java



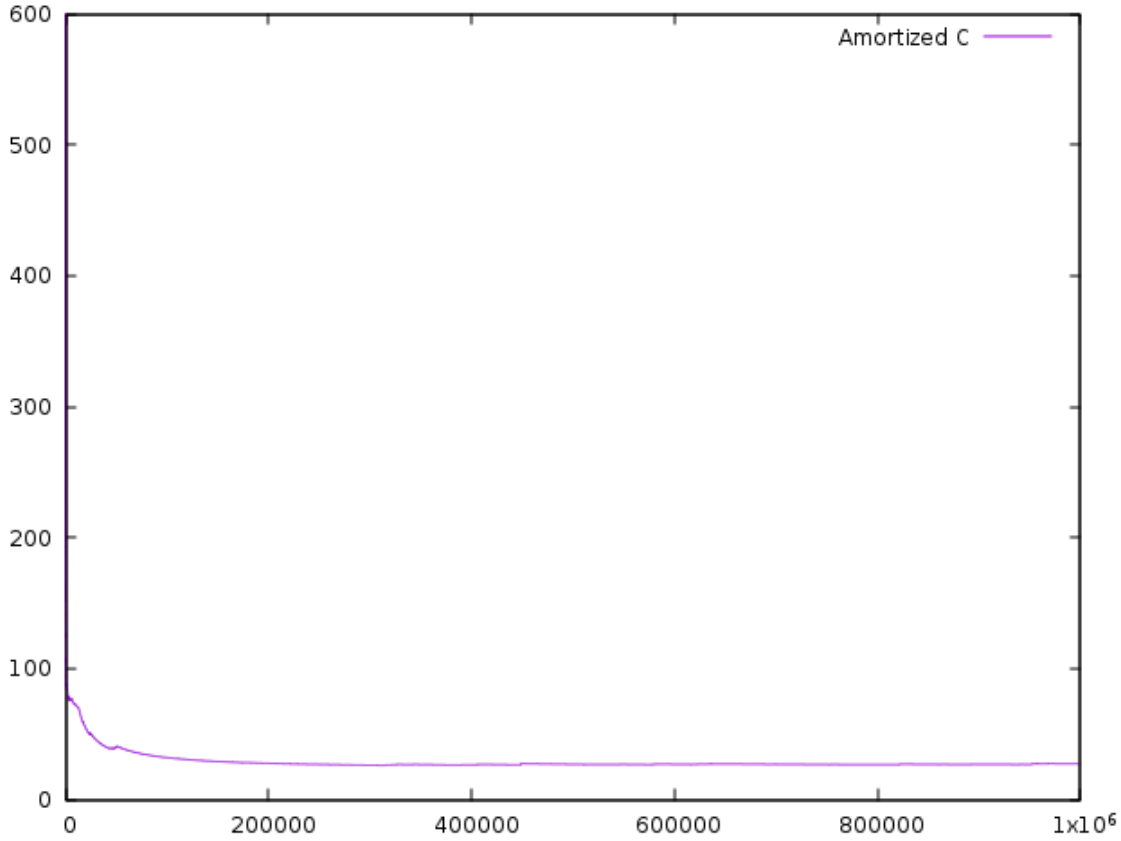
Le coût amorti pour le langage python

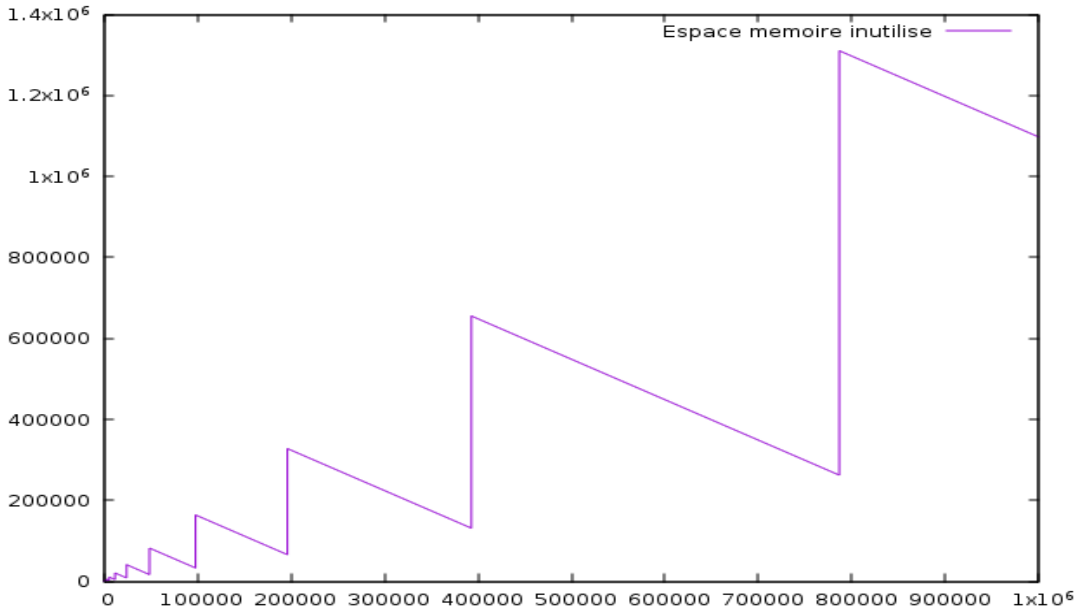
d) Ce qui change est le temps d'exécution; quand on lance le programme plusieurs fois car il n'est pas le seul à s'exécuter y'a d'autre programmes qui s'exécute avec lui.
e) Parmi les raisons lesquelles des langages de programmation sont plus rapide que d'autres, on trouve l'allocation mémoire.
f) Quand on arrive à ¼ de la taille de la table de l'ancienne table On double la taille, mais si on change le code, on met la taille=capacité le graphe commence à zéro.

Tp2 : Tables Dynamiques (Ajout et Suppression)

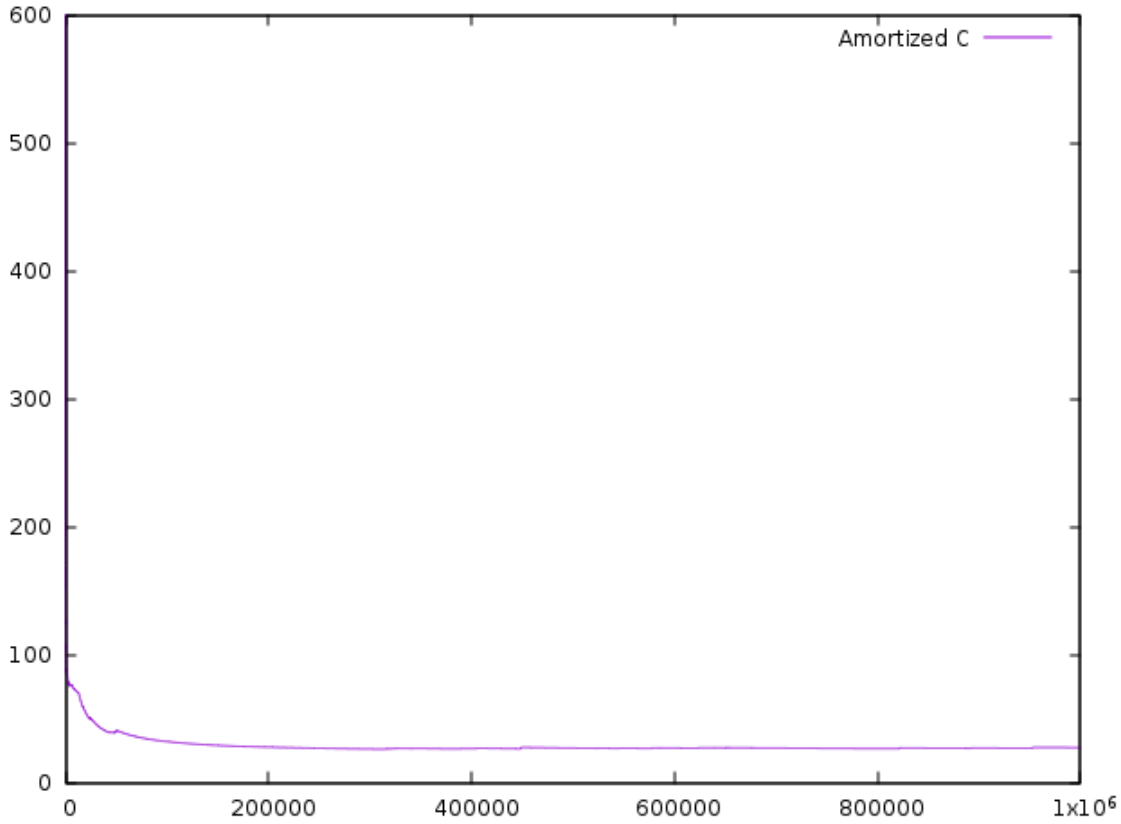
4) Tester des valeurs différentes de p ∈ {0.1, 0.2, 0.3, . . .}. Utiliser gnuplot pour afficher les coûts amortis et l'espace mémoire non-utilisé pour chacune de ces expériences. Que pensez-vous de la relation entre p, le coût en temps et le gaspillage de mémoire ?

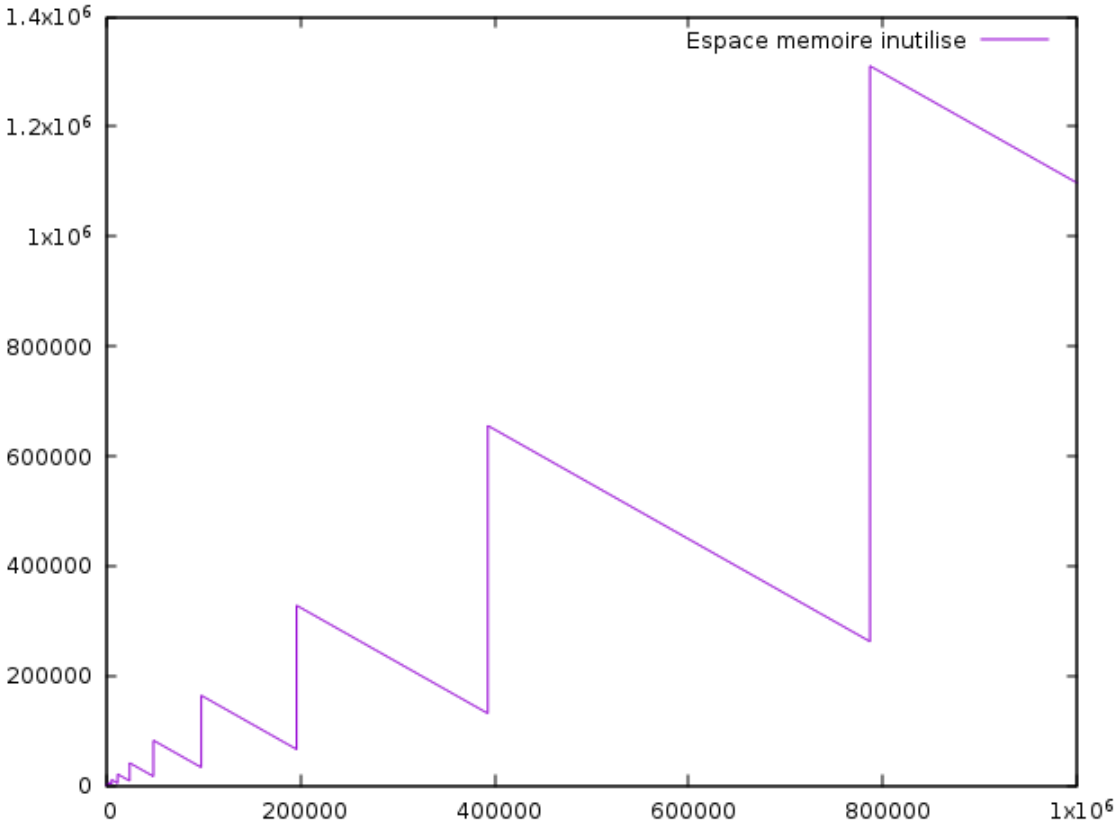
P=0.1



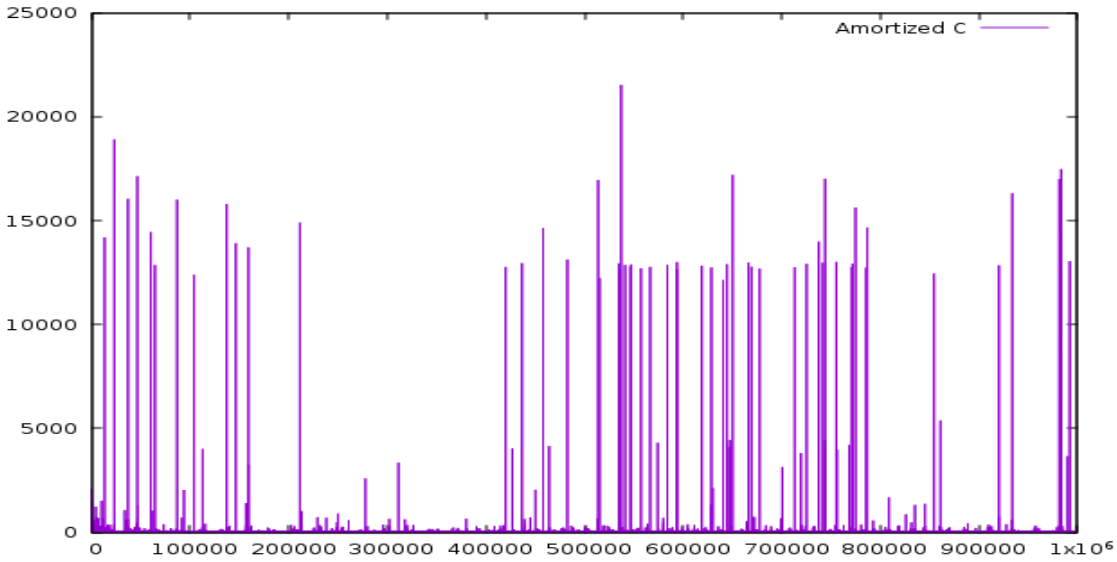

Espace memoire inutilise
1.4x10⁶
1.2x10⁶
1x10⁶
800000
600000
400000
200000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶

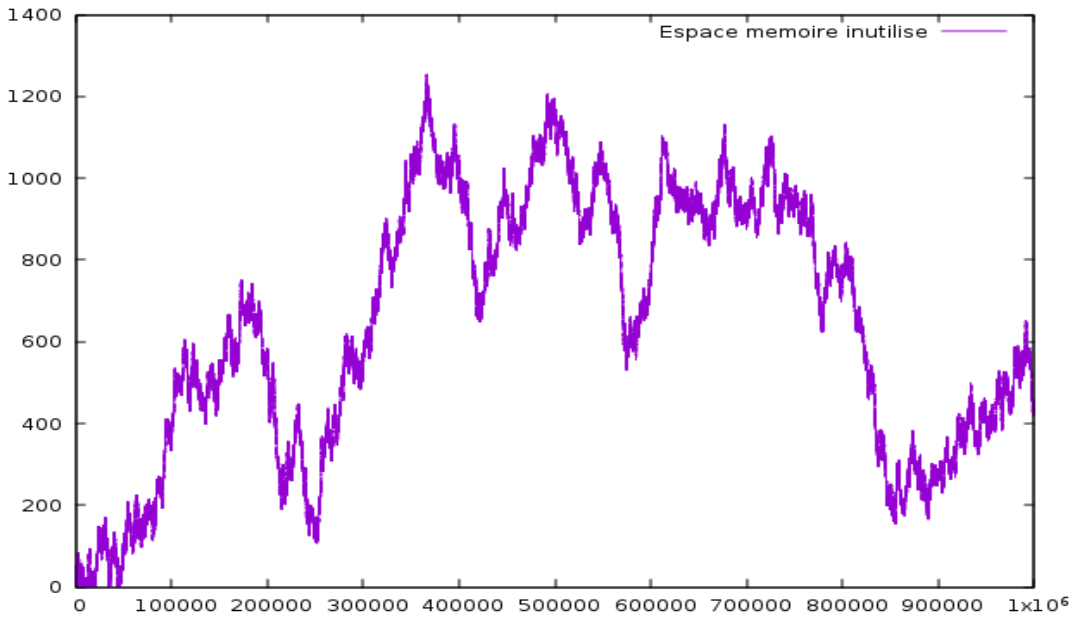
P=0.2


Amortized C
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10⁶

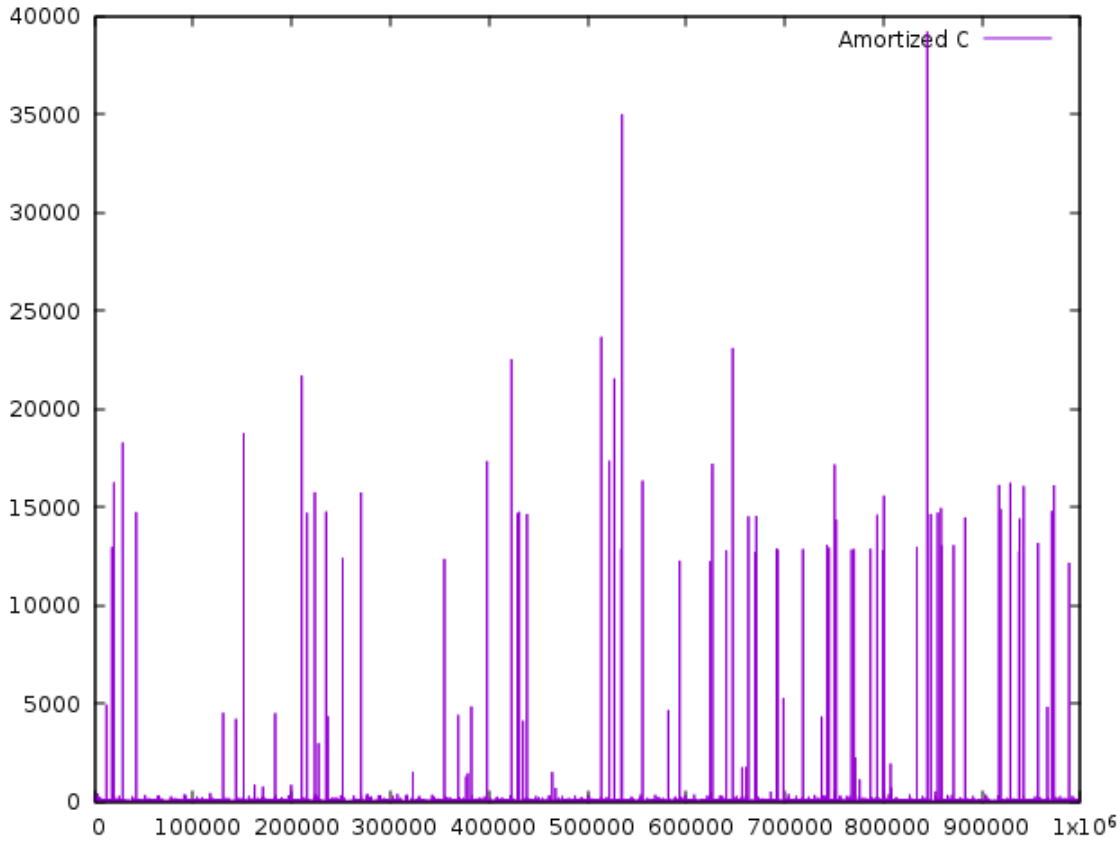

Espace memoire inutilise
1.4x10⁶
1.2x10⁶
1x10⁶
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1x10⁶

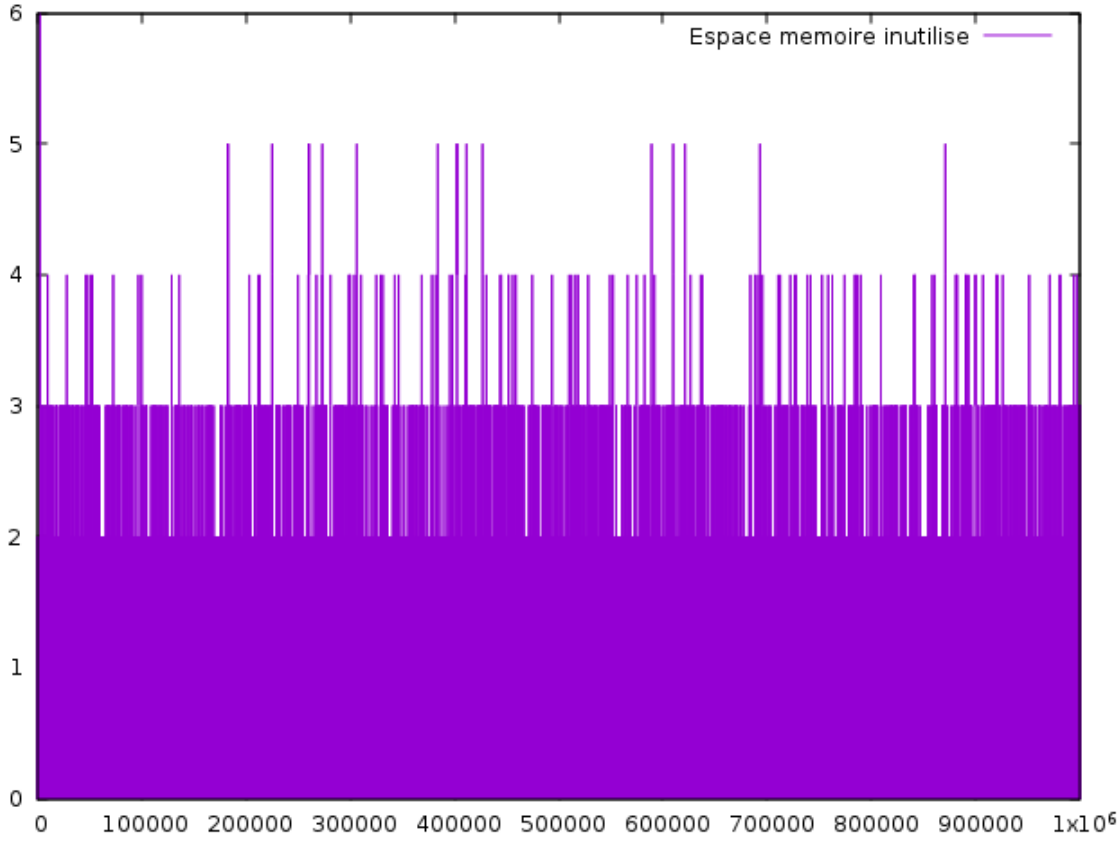
P=0.5


Amortized C
25000
20000
15000
10000
5000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶

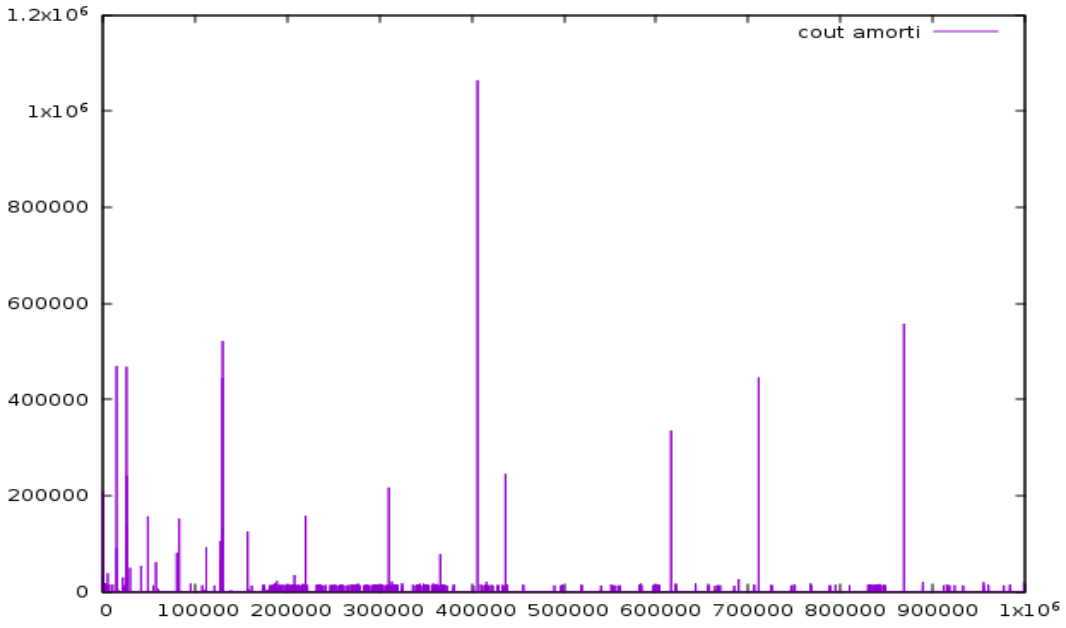

Espace memoire inutilise
1400
1200
1000
800
600
400
200
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶

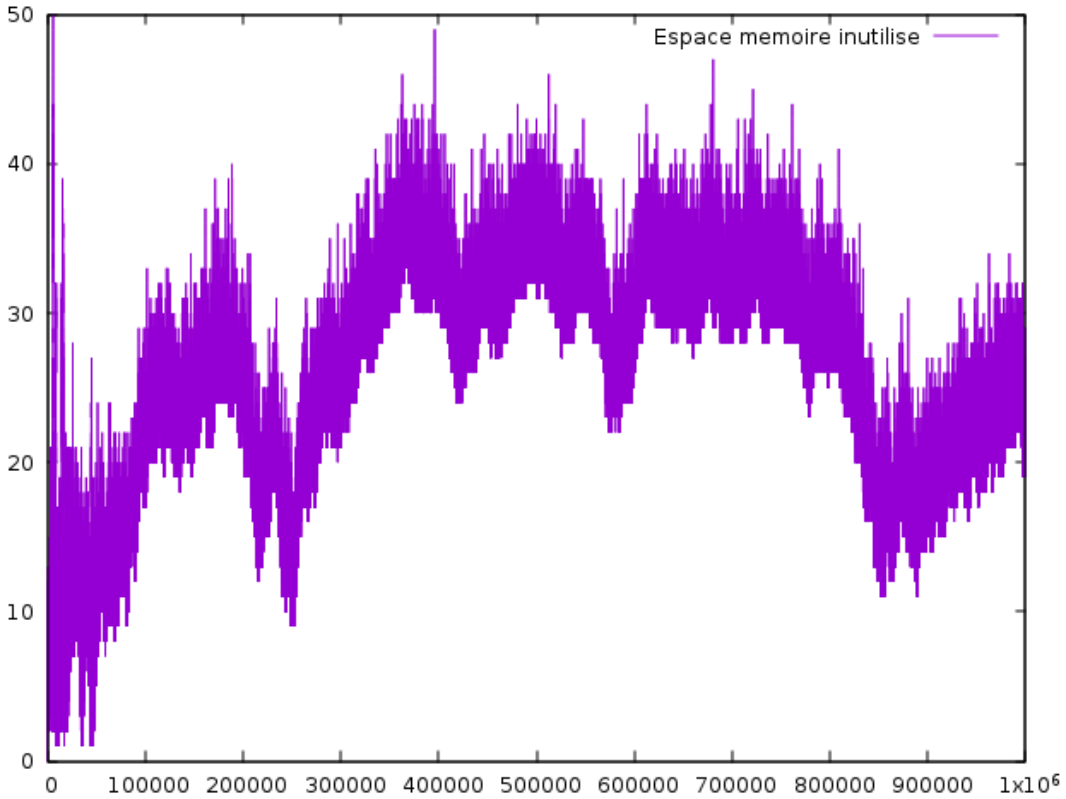
P=0.9

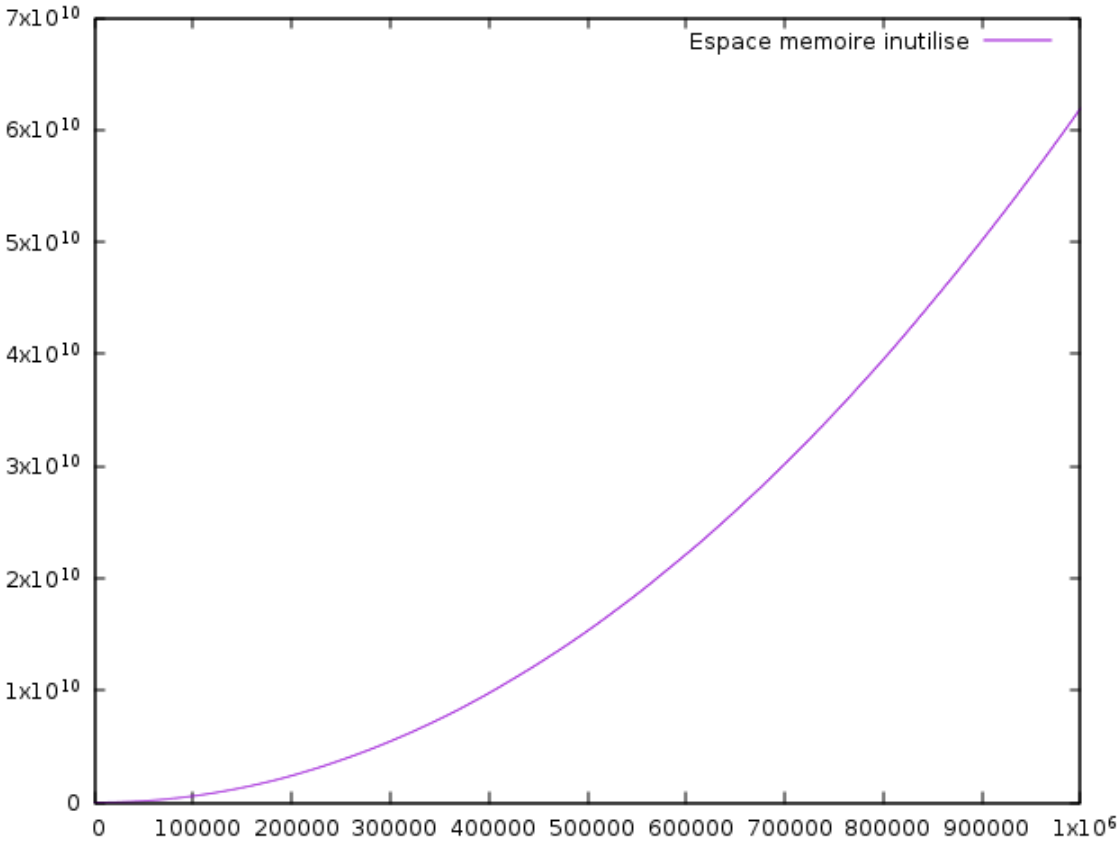

Amortized C
40000
35000
30000
25000
20000
15000
10000
5000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶

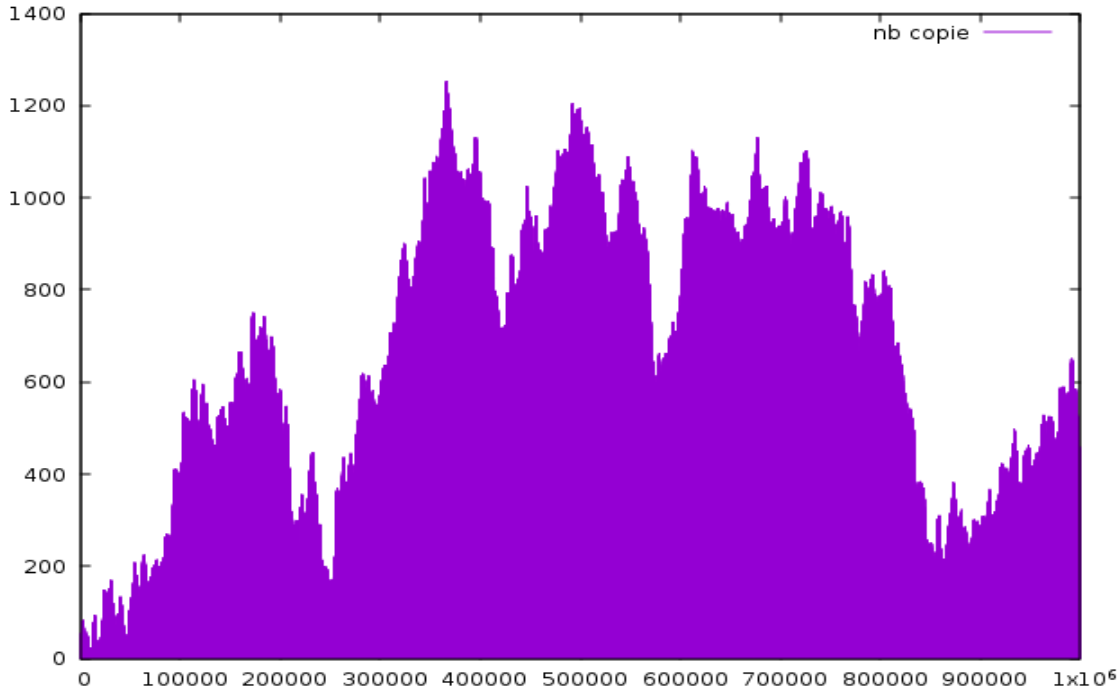


(5) Choisir p = 0.5. Modifier la stratégie de redimensionnement de la table pour utiliser taillei+1 = taillei + q taillei lors d'une extension, comme dans la dernière question du TP1 ; et taillei+1 = taillei − q taillei lors d'une contraction. On déclenche une extension au moment de l'insertion quand le facteur de remplissage de la table αi = nomi/taillei = 1, càd quand la table est pleine. Quand est-ce qu'il faut contracter la table ? Que pensez-vous de l'efficacité de cette stratégie ?


cout amorti
1.2x10⁶
1x10⁶
800000
600000
400000
200000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶


Espace memoire inutilise
50
40
30
20
10
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10⁶


Espace memoire inutilise
7x10^10
6x10^10
5x10^10
4x10^10
3x10^10
2x10^10
1x10^10
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6


nb copie
1400
1200
1000
800
600
400
200
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

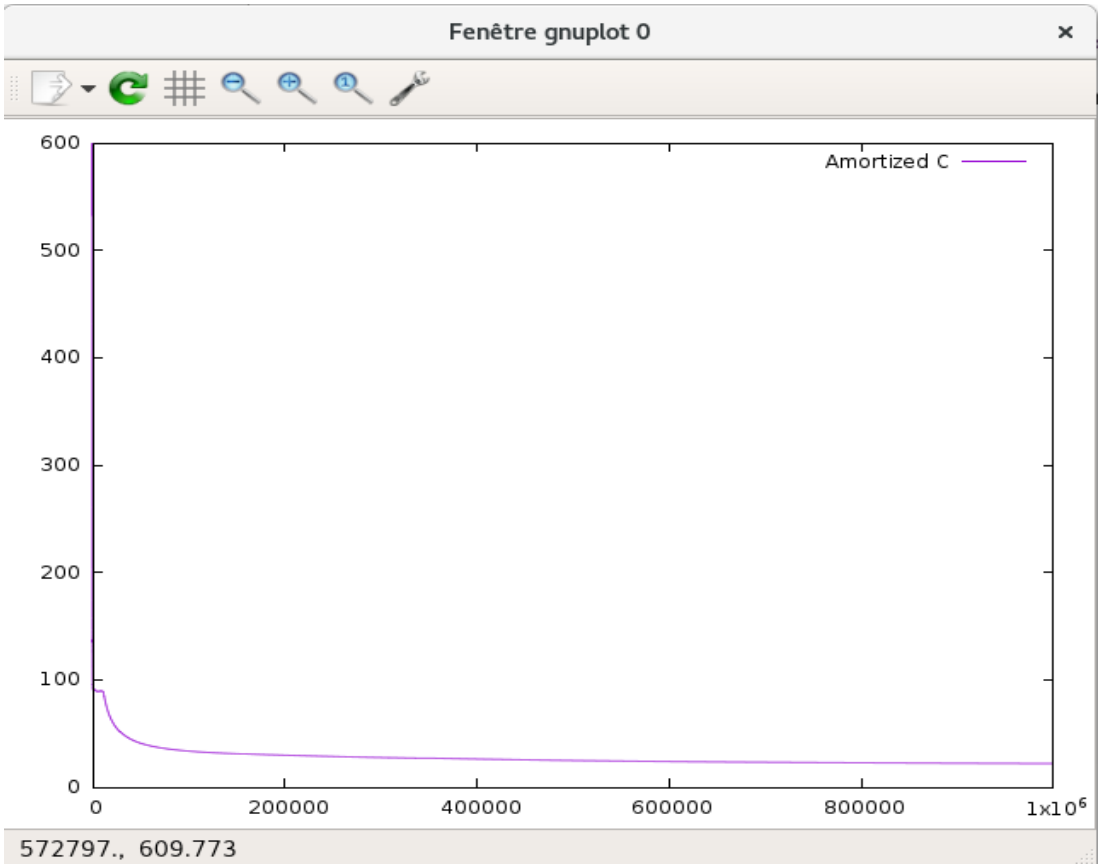
6) la valeur de p pour laquelle cette stratégie semble mieux fonctionner est p=0,5 car la probabilité que la table est remplie c’est-à-dire que des insertions est (1/2)^n.

Tp3 : Tas Binaires et Tas Binomiaux

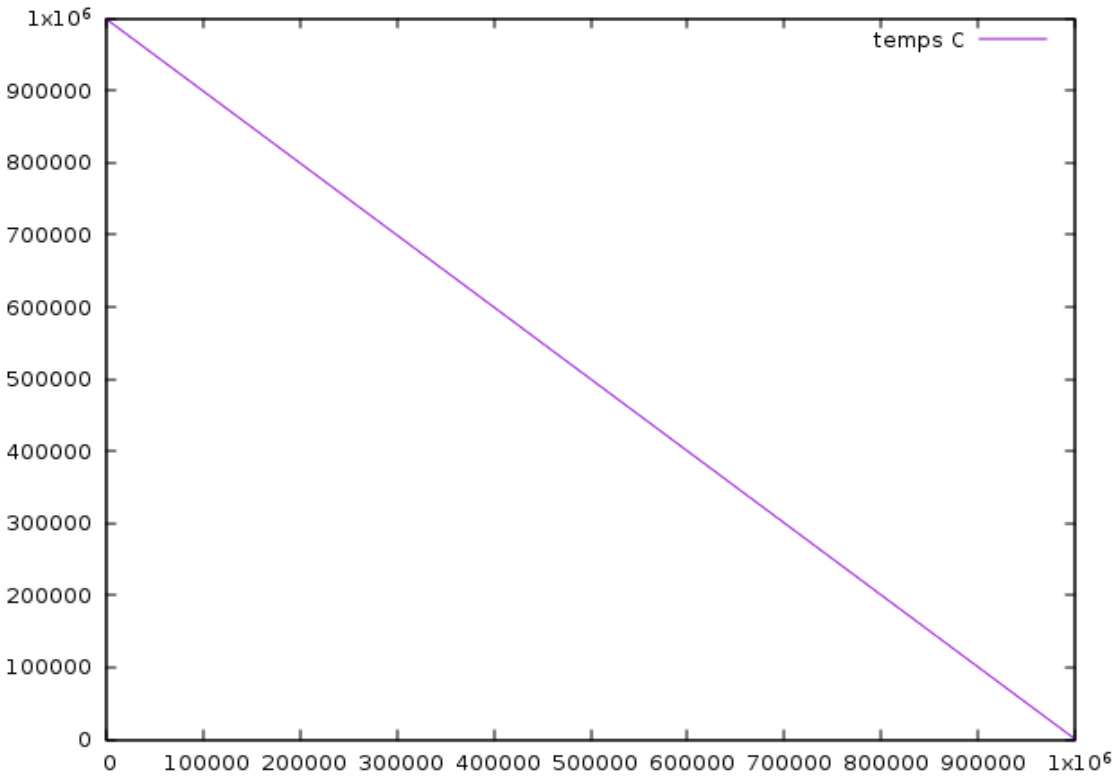
On a effectué des expériences sur l'efficacité en temps et en mémoire de cette structure :

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

En temps :

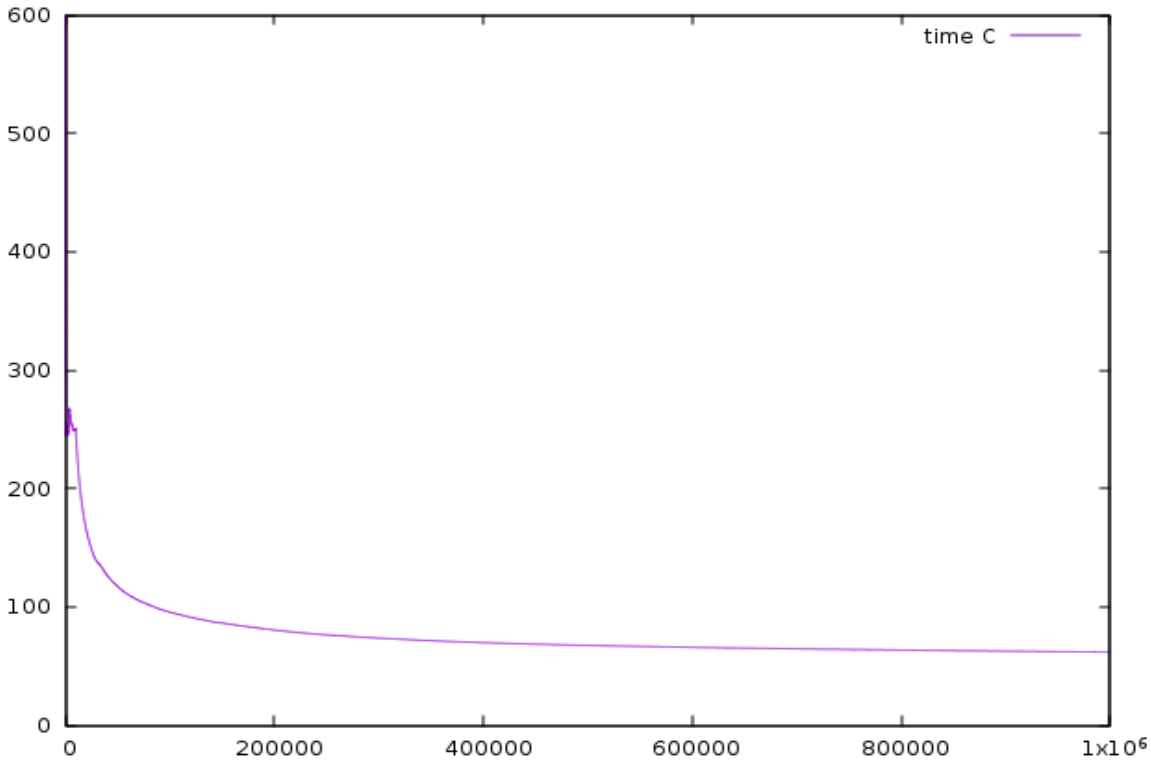


En mémoire :

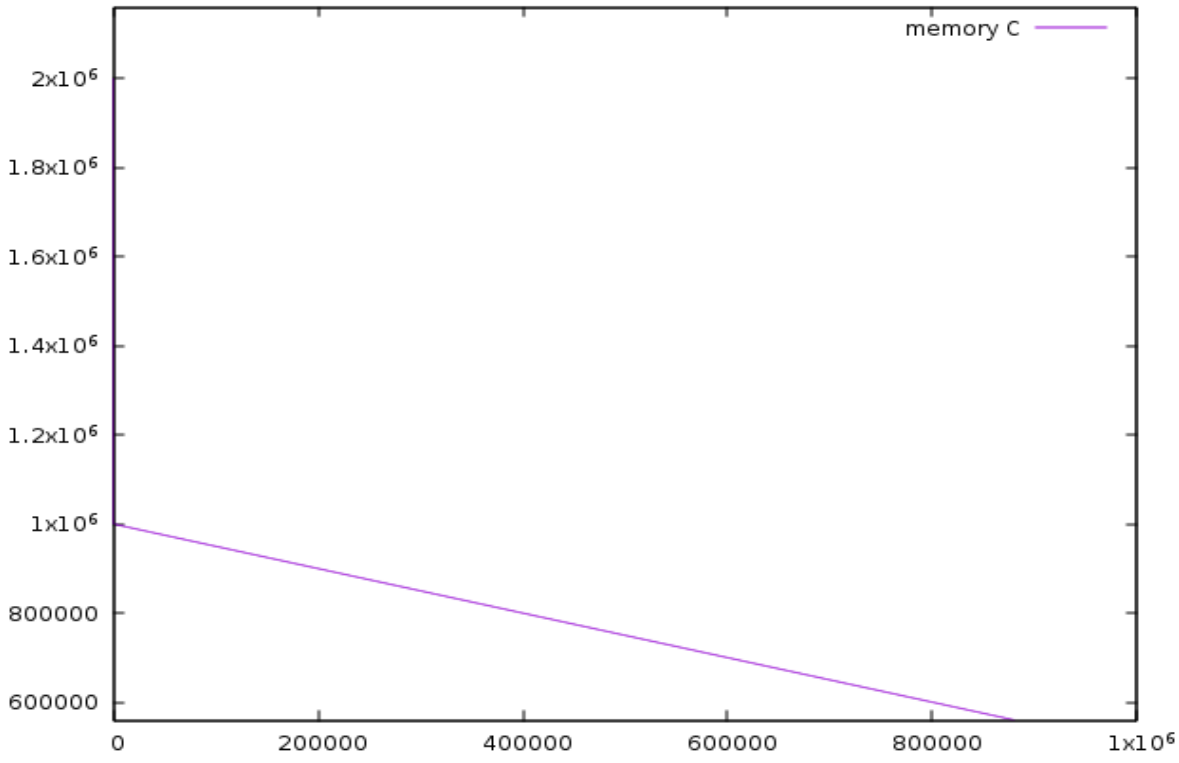


-dans le cas où l’on ne fait qu’ajouter des clés dans l’ordre décroissant

En temps :

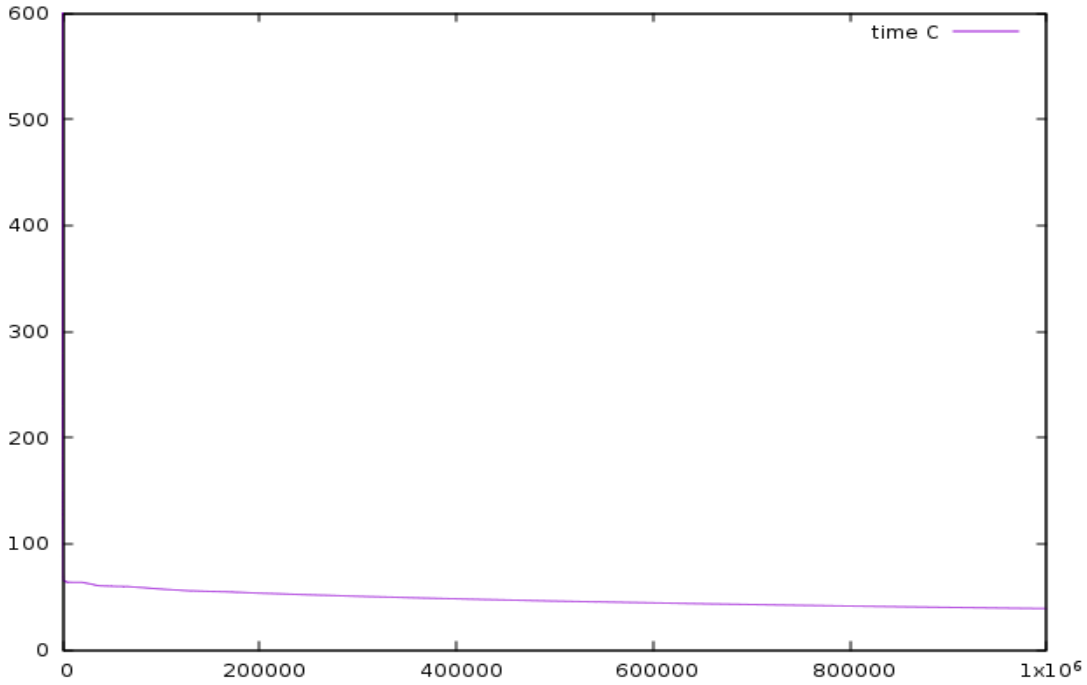

time C
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10⁶

En mémoire :

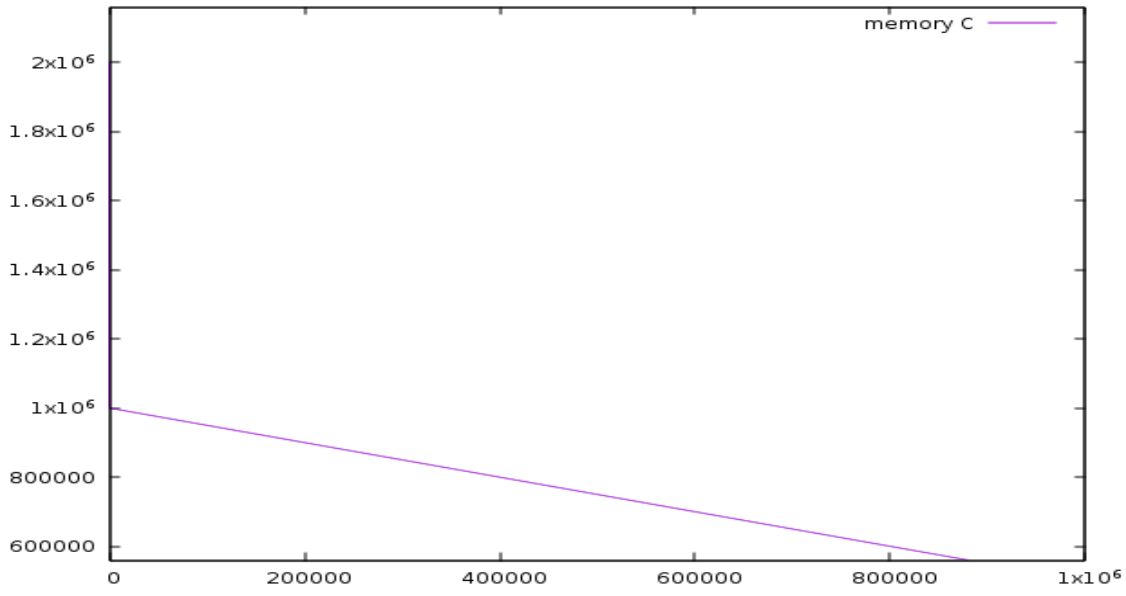

memory C
2x10⁶
1.8x10⁶
1.6x10⁶
1.4x10⁶
1.2x10⁶
1x10⁶
800000
600000
0
200000
400000
600000
800000
1x10⁶

— dans le cas où l’on ne fait qu’ajouter des clés aléatoires

En temps :



En mémoire :



Dans cettes expériences précédentes on remarque que les résultats sont presque les mêmes, dans le cas de la mémoire on constate que c’est logique puisque la courbe est décroissante et elle commence de son maximum, en effet au début le tas est vide de coup toute la mémoire est inutilisable et en insérant la mémoire inutilisable diminue de coup la courbe décroit.

Tp4 : B-arbres

L'objectif de ce tp est d'implémenter les b-arbres et les arbres AVL afin de comparer la performance de ces deux types.

1/ Pour manipuler les b-arbres, il faut premièrement implémenter une structure de données :

```
typedef struct _noeud {

int tabVal [2*Max + 1];          // tabVal[0] contient le nb de valeurs du noeud

struct _noeud* tabFils [2*Max + 1];

 struct _noeud* pere;

 int isFeuille;

 int noNoeud; //juste utilisé pour l'affichage

} Noeud;
```

On a implémenté aussi quelques fonctions comme :

// La création d'une b-arbre :

```
Noeud * b_arbre_create();
```

// Recherche la valeur v dans l'arbre r :

```
Noeud* recherche(int v, Noeud* noeud);
```

// Insertion d'une valeur dans un noeud feuille

```
int insererSimple(int v, Noeud* noeud);
```

// Suppression de valeurs dans un nœud

```
void oterSuiteElts(Noeud *noeud, int debut);
```

// Suppression de la première valeur d'un nœud

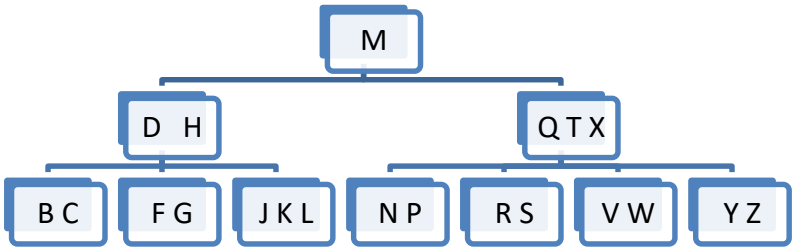
```
void oterTete(Noeud *noeud);
```

2/ Pour la structure « nœud » on a choisi comme éléments :

- tabVal : un tableau d'entier qui contient les valeurs de chaque nœud dont tab[0] contient le nombre des valeurs.
- Père : est de type struct _noeud puisque même le père il peut contenir un ensemble de valeurs.
- tabFils : un fils est lui-même une struct _ nœud qui peut contenir des valeurs.
- isFeuille : il prend comme valeur 1 ou 0 pour indiquer si ce fils est une feuille ou s'il a lui-même des fils.

En effet, chaque nœud de l'arbre contient entre t et 2t clés sauf la racine contient t-1 et 2t-1 (avec t est le degré).

Exemple : t=3



3/ Pour les arbres AVL on a comme structure :

```
struct NOEUD {

    int elt ;

    deuxbits bal ; /* compris entre –1 et +1 *//* -2 et
    +2 dans la suite */

    struct NOEUD * g, * d ;

    } noeud ;
```

- elt : c'est l'élément dans la nœud.
- G, D : sont respectivement le fils gauche et fils droit du nœud

NB : pour la partie programmation, pour chaque tp on a créé un fichier nommé avec son numéro, et vous trouvez les codes correspondants (pour tous les tps on a codé en C).

Conclusion

Durant la réalisation de nos tps, on a, fur et à mesure, essayer de trouver les bonnes résolutions, on veut dire par là qu'on a rencontré pas mal de problèmes, dans le sens où il était dur pour nous à chaque fois de mieux comprendre la situation et de définir concrètement ce qui était vraiment important dans chaque situation, notre plus grand défi était de faire les choses de façon qu'elles soient simples et correctes mais surtout compréhensibles, pour cela on est passé par trois étapes essentielles essayer, échouer, recommencer on a tourné en boucle sur ses trois étapes .

Ces tps de structures de données avancées nous ont projeté plus loin dans notre parcours, on veut dire par là que ses réalisations nous ont permis de voir un des aspects de difficulté qu'on sera peut-être, même forcement mener à réaliser au moins une fois dans notre vie future, donc ces tps étaient vraiment bénéfique pour nous, on a su mieux comprendre l'utilisation des structures de données.