Rapport TP SDA Structures de données Avancée

Master 1 Informatique

Cherfa Abderaouf et Yahouni Amine

30 décembre 2019

Table des matières

Chapitre 1

tp1

1.0.1 le morceau de code qui semble prendre le plus de temps à s'exécuter :

la partie du code qui semble prendre le plus de temps a s'excuter est en effet la fonction arraylist_append Dans la figure 1.1 page 2, ...

FIGURE 1.1 – arraylist_append

```
char arraylist_append(arraylist_t * a, int x){
  char memory_allocation = FALSE;
  if( a!=NULL ){
    if( arraylist_do_we_need_to_enlarge_capacity(a) ){
      memory_allocation = TRUE;
      arraylist_enlarge_capacity(a);
    }
    a->data[a->size++] = x;
  }
  return memory_allocation;
}
```

...

Cette fonction fait appel 2 autres fonction qui vont vérifier dans un premier temps si il y'a besoin d'agrandir la capacite de notre table puis dans un 2eme temps si c'est le cas l'agrandir en faisant appel a enlarge_capacity qui va elle-même faire appel a la fonction realloc pour réallouer un nouvel espace mémoire ce qui justifie la lenteur à l'exécution

1.0.2 Le coût amorti en temps dans les différents langages

— Le temps d'exécution avec le langage C :

FIGURE 1.2 – Language C

```
votre@votre-VirtualBox:~/Bureau/SDA/sda/C$ ./arraylist_analysis
Total cost: 49834254.000000
Average cost: 49.834254
Variance: 122815043516.547128
Standard deviation: 350449.773172
```

— Temps d'exécution avec le langage CPP :

FIGURE 1.3 – Cpp

```
votre@votre-VirtualBox:~/Bureau/SDA/sda/C$ ./arraylist_analysis
Total cost: 49834254.000000
Average cost: 49.834254
Variance: 122815043516.547128
Standard deviation: 350449.773172
```

— Temps d'exécution avec le langage JAVA :

FIGURE 1.4 – java

```
votre@votre-VirtualBox:~/Bureau/SDA/sda/Java$ java Main
Total cost : 325518615
Average cost : 325.518615
Variance :331400599630361 98.631288481775
Standard deviation :182044115.43094766139984130859375
```

— Temps d'exécution avec le langage Python :

FIGURE 1.5 – python

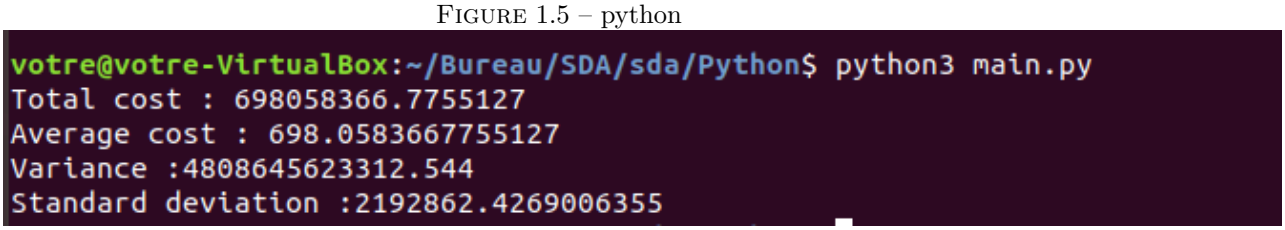
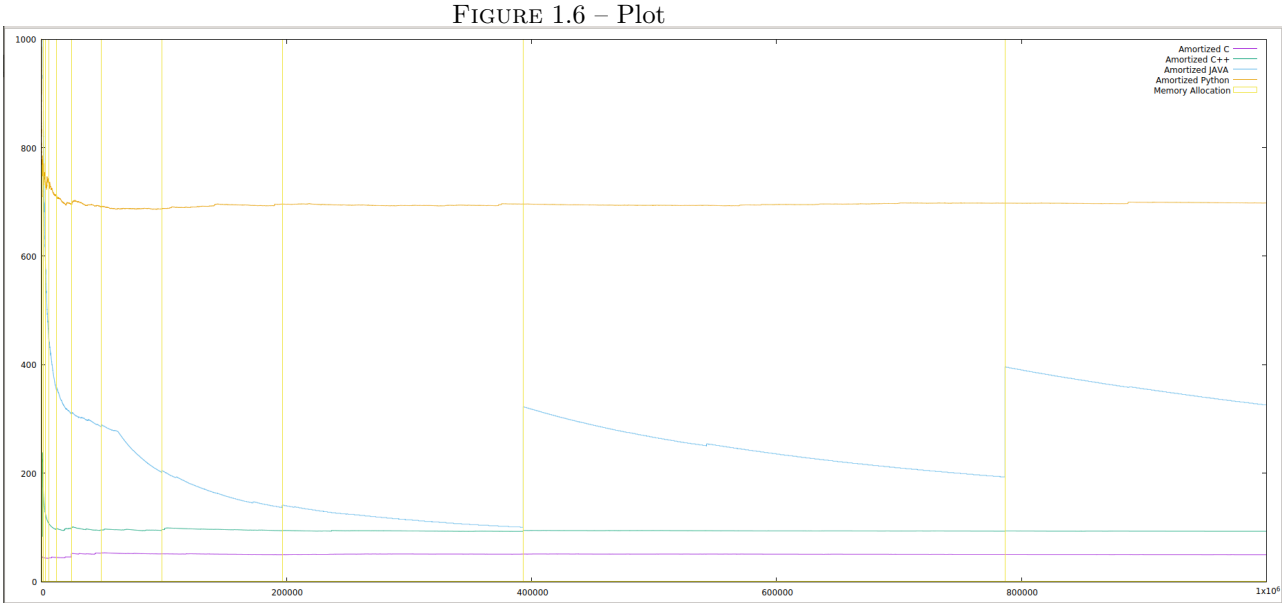


FIGURE 1.6 – Plot

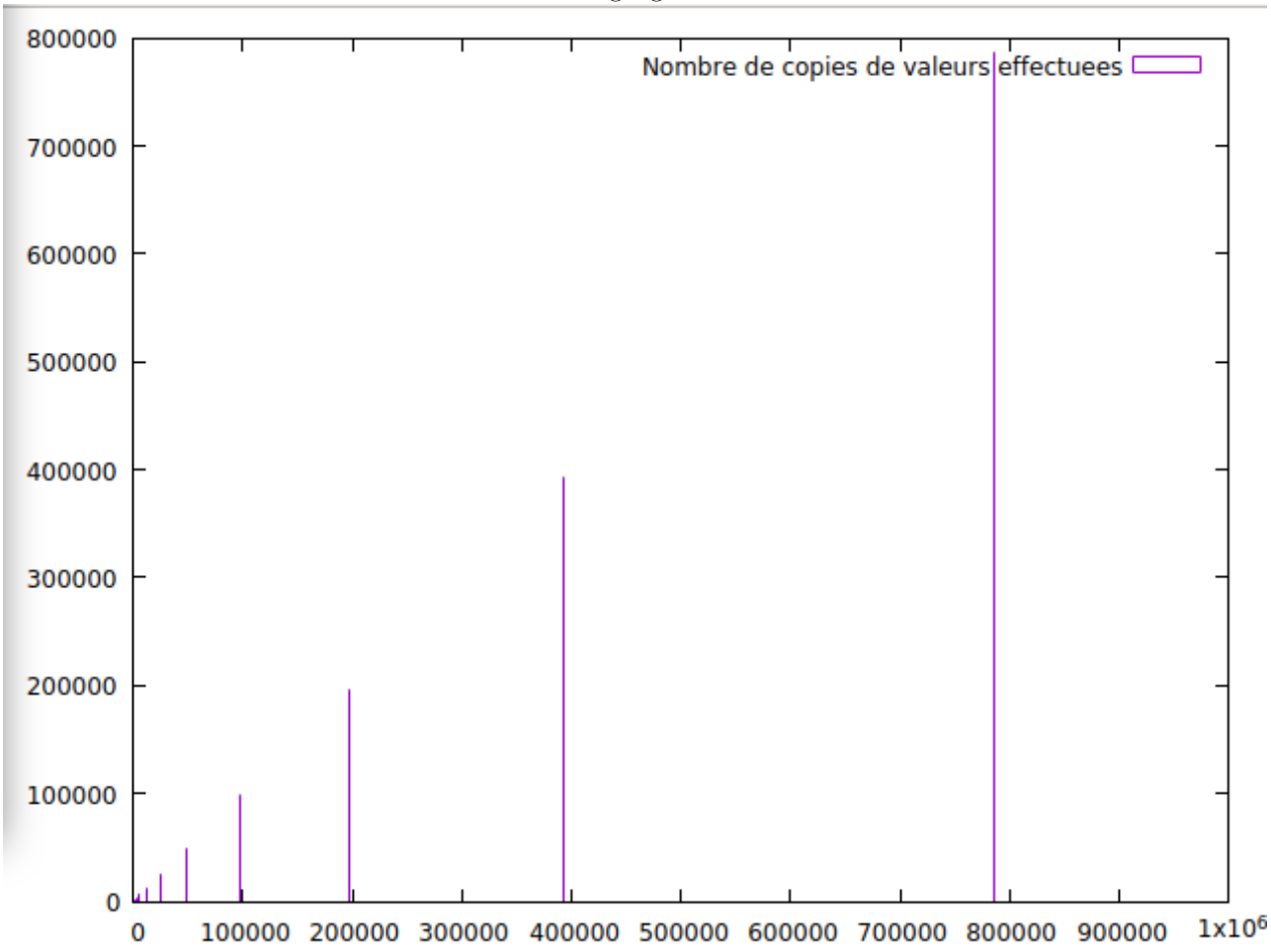


On peut remarquer que le cout amorti en temps est a son plus haut niveau au lancement des programmes pour les 3 langage C++,Java et Python car ce sont des langage orienté objet du coup au lancement le programme doit charger toute les class et les objets et les initialiser ce qui prend plus de temps que l'initialisation d'une simple structure de donnée en C ce qui fait que le C démarre assez bas et monte au fur et à mesure des opérations

1.0.3 Le nombre de copies effectués par chaque opération (C/C++ ou Java)

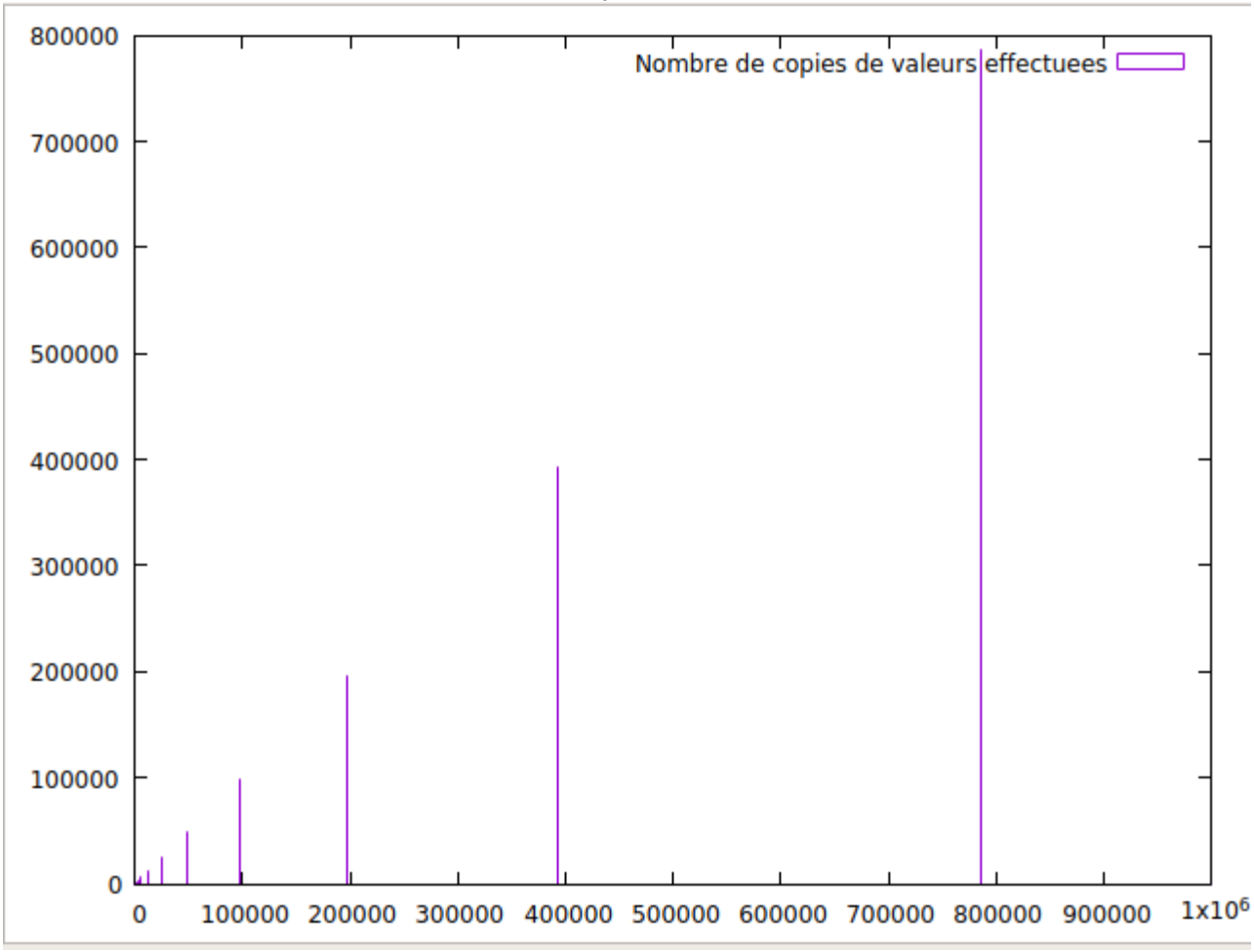
— Nombre de copie de effectuées C :

FIGURE 1.7 – Language C



— Nombre de copie de effectuées JAVA :
On peut remarquer que le nombre de copie est identique avec les deux langage

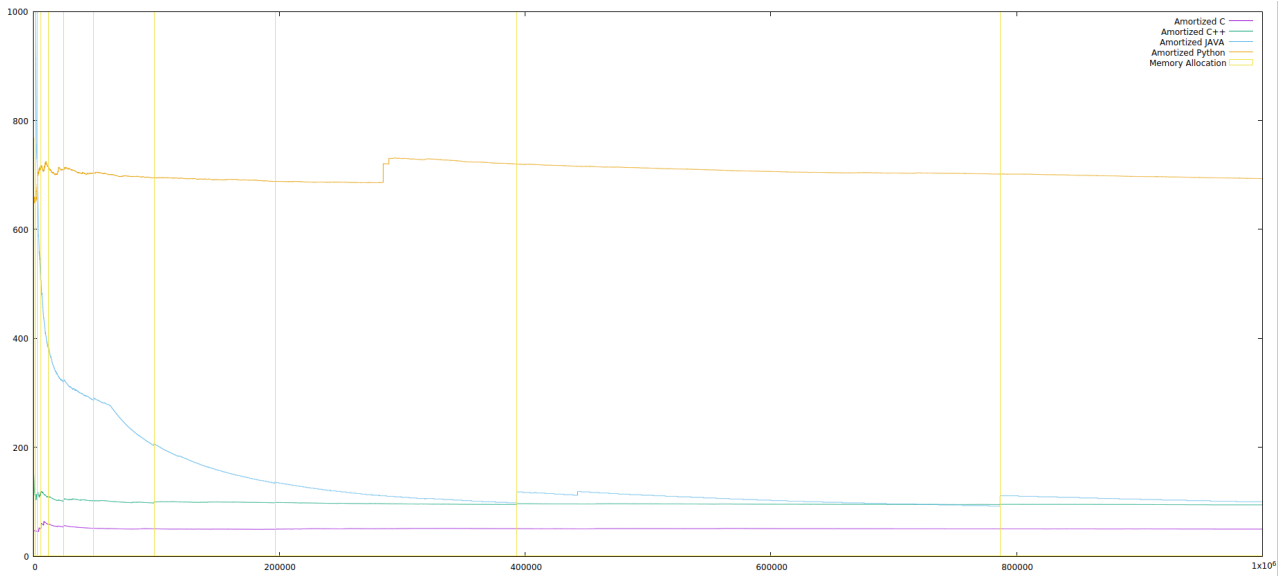
FIGURE 1.8 – java



1.0.4 Recommencement des experiences

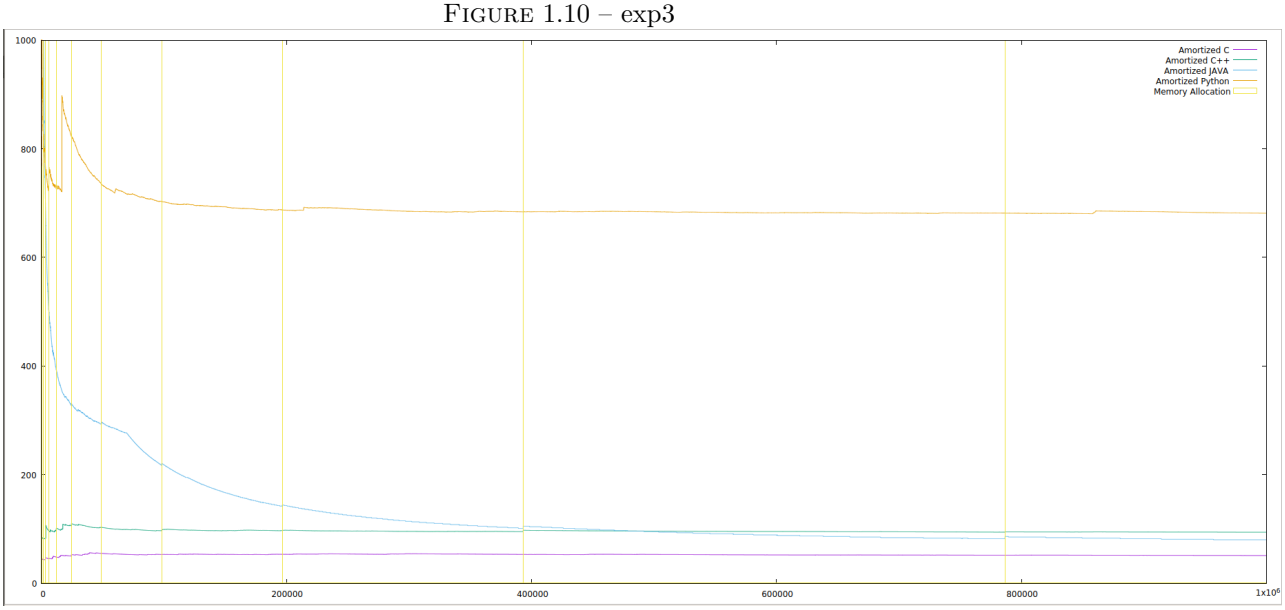
— cout amortie dans une nouvelle expérience :

FIGURE 1.9 –



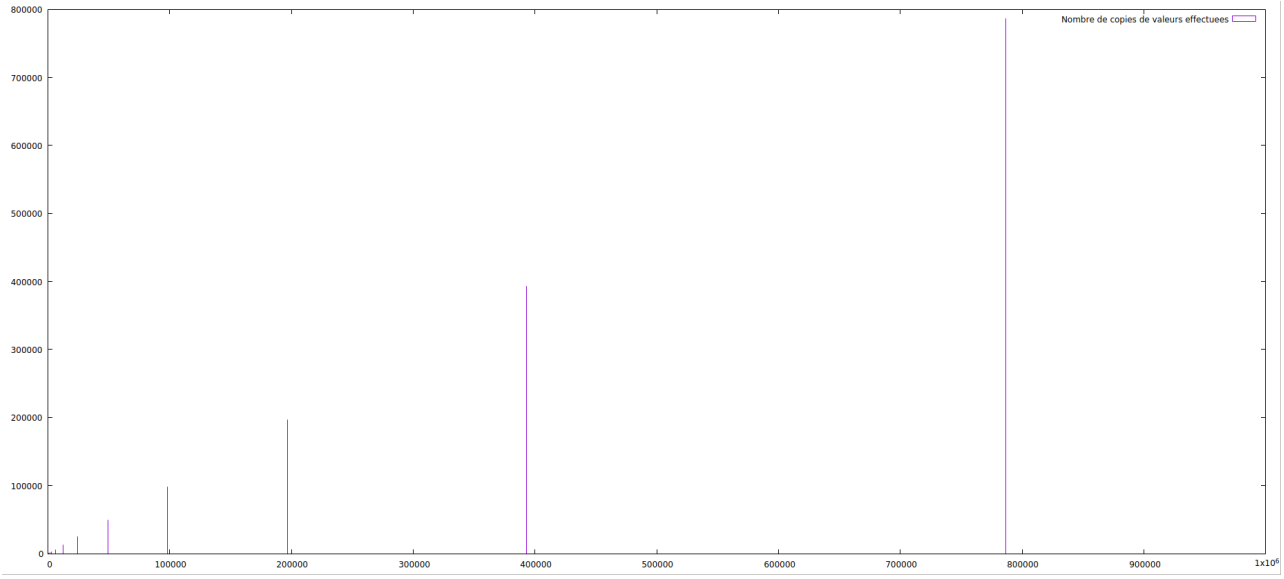
— Expérience 3

FIGURE 1.10 – exp3



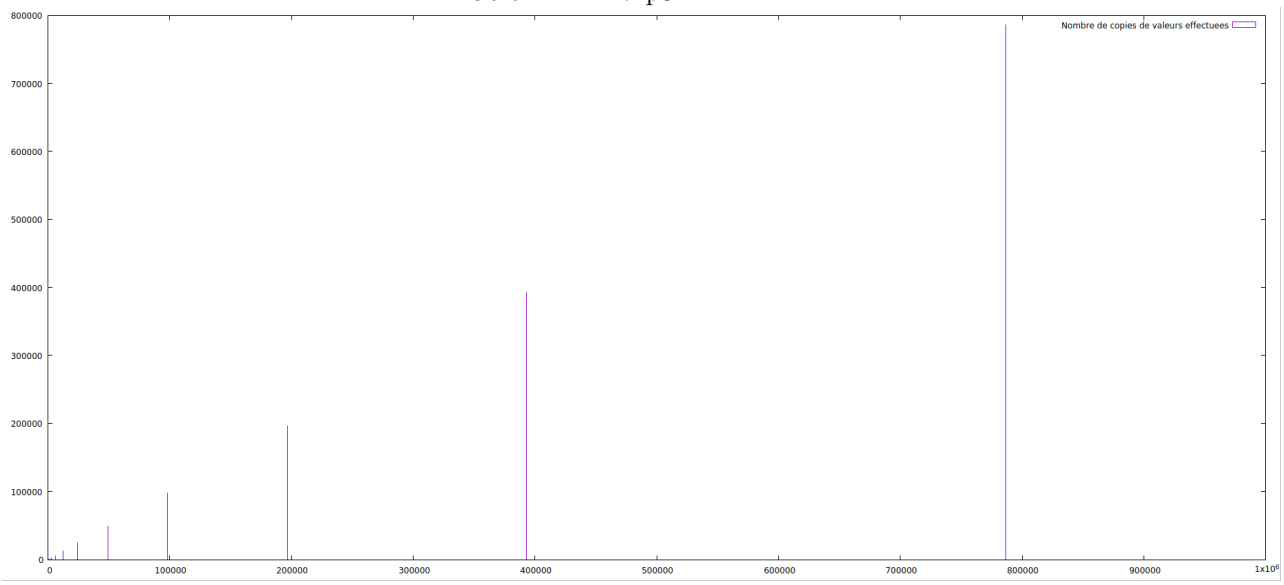
— On remarque que le cout amortie en temps change d'une expérience a une autre comme l'indique les plots ce dessus mais pas le reste comme l'indique les plots ci-dessous

FIGURE 1.11 –



— Expérience 3

FIGURE 1.12 – exp3

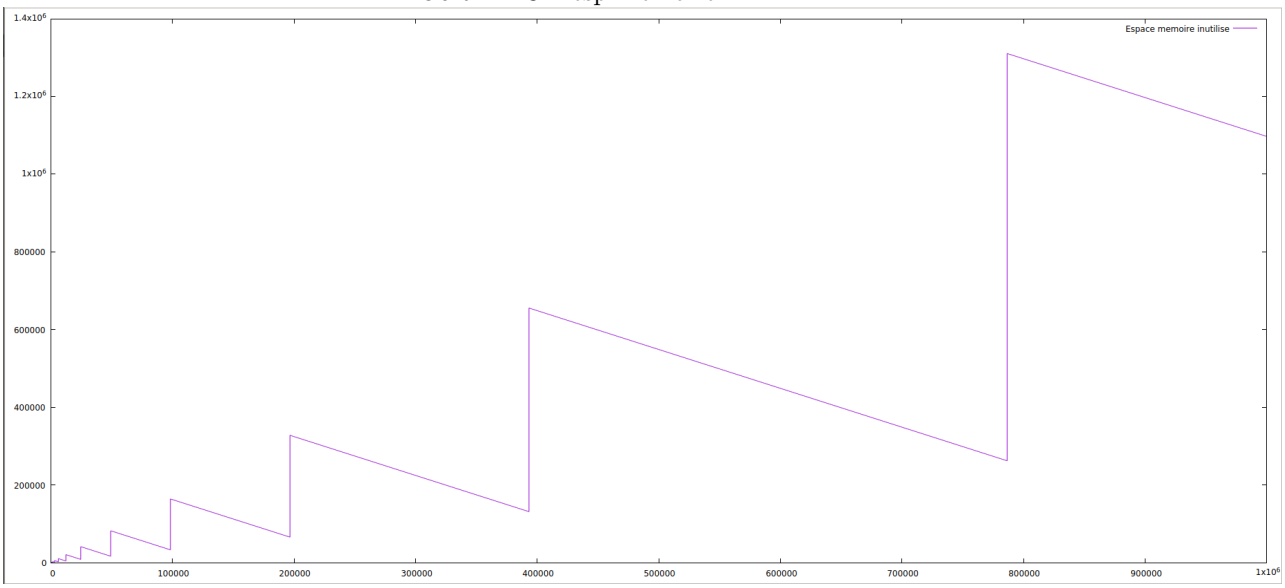


1.0.5 Explication :

Certains langages sont plus rapides que d'autres pour différente raison Premièrement leurs paradigme par exemple le langage C étant un langage fonctionnel ce qui fait qu'il n'a pas de class et d'objets à gérer comme le CPP ou encore le JAVA Mais aussi pour une autre raison par exemple le Java exécute son code sur une JVM (Java virtuel machine) ce qui fait que sa ralentie l'exécution de ces programme par rapport aux autres Dans les langage dit orienté objet l'appel d'un objet enclenche un système de recherche dans l'arborescence du programme pour retrouver la class adéquate et par la suite l'attribut ou la méthode en question ce qui demande du temps et affecte de ce fait les résultats sur le temps d'exécution

1.0.6 L'espace mémoire inutilisé :

FIGURE 1.13 – esp mémoire



On peut remarqué qu'il est ascendant cela est du a notre méthode enlarge_capacity qui utilise comme méthode pour agrandir la taille la duplication de la taille, le scénario dans le quel cela pourrait poser problème est si on agrandis notre tableau jusqu'au moment ou l'on ne dispose plus d'espace libre

1.0.7 Modification de la fonction do_we_need_to_enlarge_capacity

Pour la suite de ce rapport on a choisi de travaillé avec le langage C

Notre nouvelle fonction si size+1 (car on commence a compter de 0) = capacity cela veut dire que notre tableau est plein Du coup les résultat de notre nouvel expérience

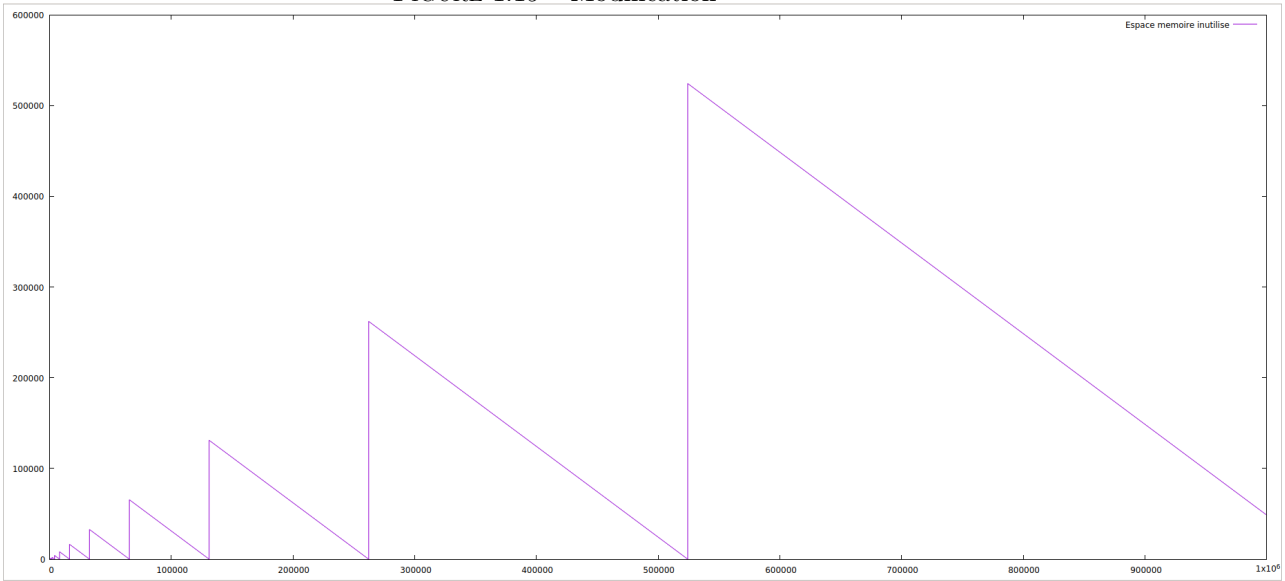
On termine avec quasiment plus d'espace mémoire inutilisé ce qui est plus optimal que la première méthode d'agrandissement

h- En faisant varier le facteur Alpha dans la méthode enlarge_capacity on remarque que plus Alpha est grand on obtient un meilleurs temps amortie mais a coté on a beaucoup plus de

FIGURE 1.14 – Modification

```
char arraylist_do_we_need_to_enlarge_capacity(arraylist_t * a){
  return ( a->size+1 == a->capacity )? TRUE: FALSE;
}
```

FIGURE 1.15 – Modification



mémoire inutilisé cela s'explique par le fait que plus Alpha est grand, moins on aura a copier toute les valeurs dans la nouvelle table Les plots avec diffèrent Alpha

- alpha 10 :
- alpha 3
- i- On paramétrant enlarge_capacity avec n= n+sqrt(n) on obtient un graph de memoir inutilisé ressemblant a celui de la fonction racine comme affiché ci-dessous

FIGURE 1.16 –

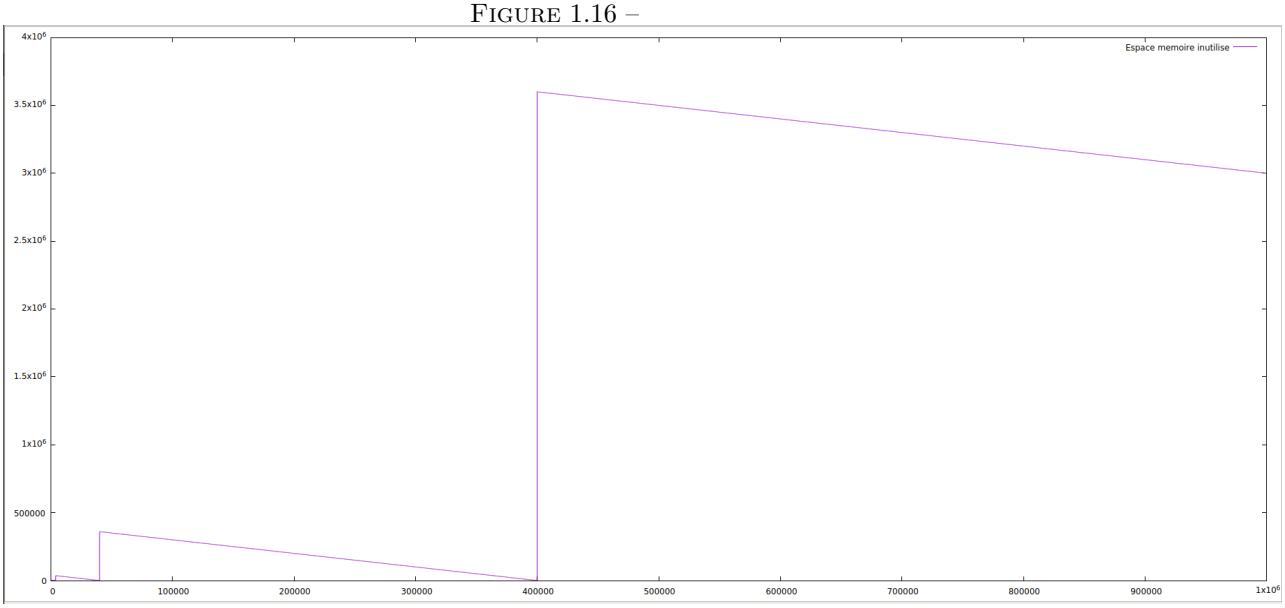


FIGURE 1.17 –

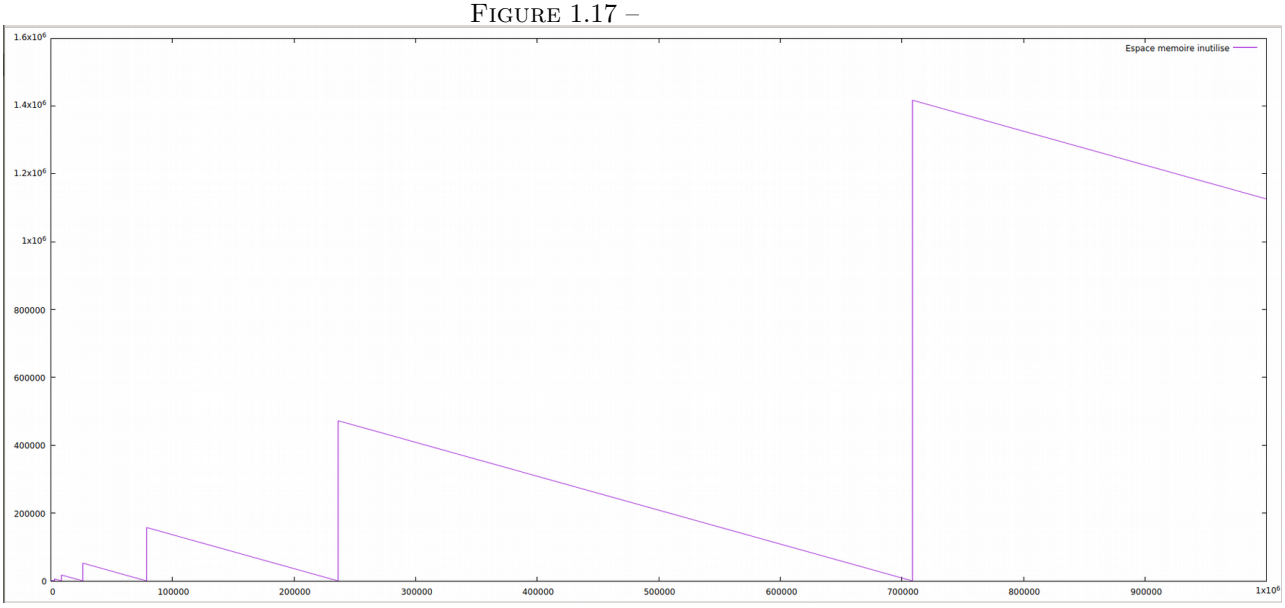
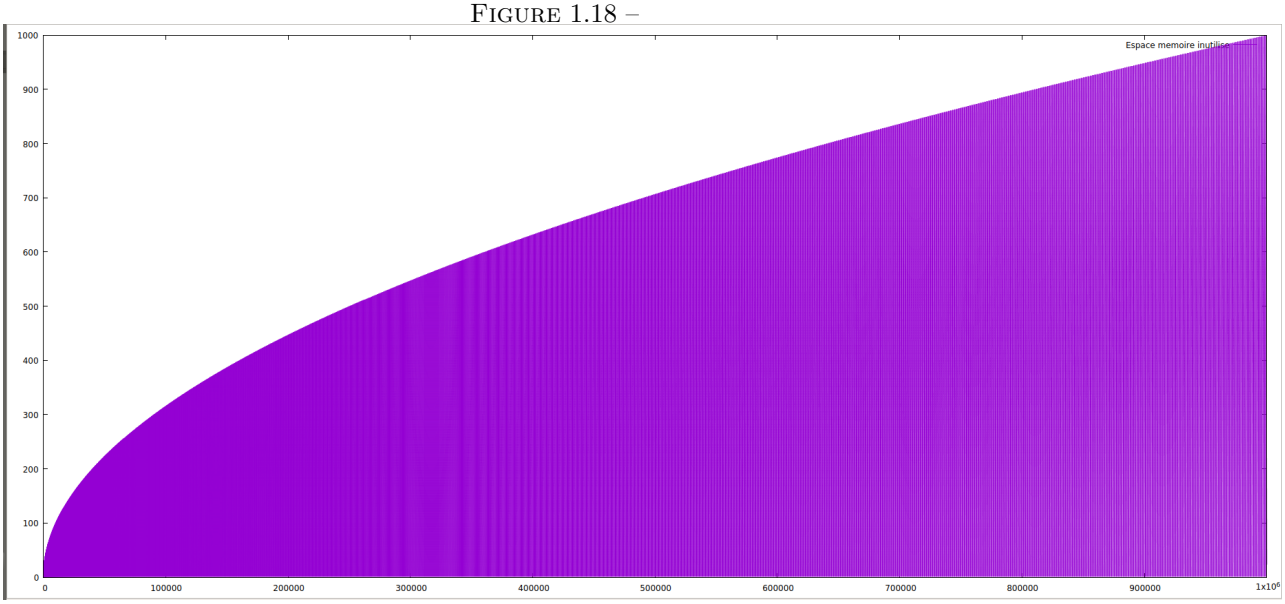


FIGURE 1.18 –


Espace memoire inutilisé
1000
900
800
700
600
500
400
300
200
100
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

Chapitre 2

Corrige TP2

2.1 Code source

```
#include<stdio.h>
#include <time.h>
#include<stdlib.h>
#include "arraylist.h"
#include "analyzer.h"



int main(int argc, char ** argv){
  int i;
  float nombre;
  // Tableau dynamique.
  arraylist_t * a = arraylist_create();
  // Analyse du temps pris par les op rations.
  analyzer_t * time_analysis = analyzer_create();
  // Analyse du nombre de copies faites par les op rations.
  analyzer_t * copy_analysis = analyzer_create();
  // Analyse de l'espace m moire inutilis .
  analyzer_t * memory_analysis = analyzer_create();
  // Mesure de la dur e d'une op ration.
  struct timespec before, after;
  clockid_t clk_id = CLOCK_REALTIME;
  // utilis  comme bool en pour savoir si une allocation a  t  effectu e.
  char memory_allocation;

  for(i = 0; i < 1000000 ; i++){
    // la probabilit  p de faire une insertion
    float p = 0.5;
    srand(11607015);
    nombre = rand()/(RAND_MAX+1.0);

        if (nombre >= p)
```

```
{
    clock_gettime(clk_id, &before);
    // Ajout d'un    lment    et mesure du temps pris par l'op ration.
    memory_allocation = arraylist_append(a, i);
    clock_gettime(clk_id, &after);

}

else

{
        if (a->size > 0 )
{

    clock_gettime(clk_id, &before);
    memory_allocation = arraylist_pop_back(a);
    clock_gettime(clk_id, &after);
}
}
    // Enregistrement du temps pris par l'op ration
    analyzer_append(time_analysis, after.tv_nsec - before.tv_nsec);
    // Enregistrement du nombre de copies eff ctu es par l'op ration.
    // S'il y a eu r allocation de m moire, il a fallu recopier tout le tableau.
    analyzer_append(copy_analysis, (memory_allocation)? i:1 );
    // Enregistrement de l'espace m moire non-utilis .
    analyzer_append(memory_analysis,arraylist_capacity(a)-arraylist_size(a));
}

  // Affichage de quelques statistiques sur l'exp rience.
  fprintf(stderr, "Total cost: %Lf\n", get_total_cost(time_analysis));
  fprintf(stderr, "Average cost: %Lf\n", get_average_cost(time_analysis));
  fprintf(stderr, "Variance: %Lf\n", get_variance(time_analysis));
  fprintf(stderr, "Standard deviation: %Lf\n", get_standard_deviation(time_analysis))


  // Sauvegarde les donn es de l'exp rience.
  save_values(time_analysis, "../plots/dynamic_array_time_c.plot");
  save_values(copy_analysis, "../plots/dynamic_array_copy_c.plot");
  save_values(memory_analysis, "../plots/dynamic_array_memory_c.plot");

  // Nettoyage de la m moire avant la sortie du programme
  arraylist_destroy(a);
  analyzer_destroy(time_analysis);
  analyzer_destroy(copy_analysis);
  analyzer_destroy(memory_analysis);
  return EXIT_SUCCESS;
}
```

2.2 Cout amorti

FIGURE 2.1 –

Calculer le coût amorti c de l'opération **Supprimer (T,x)**:

on c =ci +Φ(Di) – Φ (Di-1).

Cas1: pas de contraction

On sait $n_i-1 = n_i+1$ et $t_i-1=t_i$

$x_i-1=(n_i-1/t_i-1) > 1/3$ donc

$(n_i-1/t_i) > 1/3$, car $t_i-1 =t_i$

$t_i>3*(n_i+1)$,car $n_i-1=n_i+1$

$t_i>3*n_i+3$

$3*n_i+3-t_i < 0$

$2* n_i-t_i <0$

Alors $C= 1+ |2n_i-t_i| - | 2n_i-1 -t_i-1|$

$C= 1+ |-2n_i+t_i| - | 2(n_i+1)-t_i|$

$C =1 + t_i- 2n_i -(t_i-2(n_i+1))$

$C =1+ t_i- 2n_i +2n_i +2 +t_i$

$C = 3$

Cas 2: avec Contractions

On sait que $n_i-1 = n_i+1$ et $t_i =(3/2)\ t_i$

et $\alpha_i+1 =(n_i-1/t_i-1)=1/3$

$t_i-1 =3* n_i-1$

$(3/2)\ t_i=3* (n_i+1)$

$t_i=2*(n_i+1)$

Donc

$C = n_i-1 + |2 n_i- t_i| -|2 n_i- t_i| - |2ni-1 - t_i-1|$

$C = n_i+ (2ni + 2 -2n_i - | 2n_i+2 - (3/2)(2* (n_i+1))|$

$C = n_i+2-|- n_i -1|$

$C= n_i+2- n_i-1$

$C=1$

Chapitre 3

Corrigé Tp3 :

3.1 Tas Binnaire

3.1.1 Implémentation des tas avec un tableau de taille fixe

On suppose pour notre expérience que notre table est de taille 100

FIGURE 3.1 – Code C

```c
Heap* createHeap(void)
{
    Heap *heap = malloc(sizeof(Heap));
    if(!heap) return NULL;
    heap->tab = malloc(100*sizeof(int));    /* Prévoir l'échec de malloc */
    heap->capacity = 100;
    heap->size = 0;
    return heap;
}
```

b-On a ici les plots des couts en temps pour les 3 premiers cas (Aléatoire, ascendant, descendant)

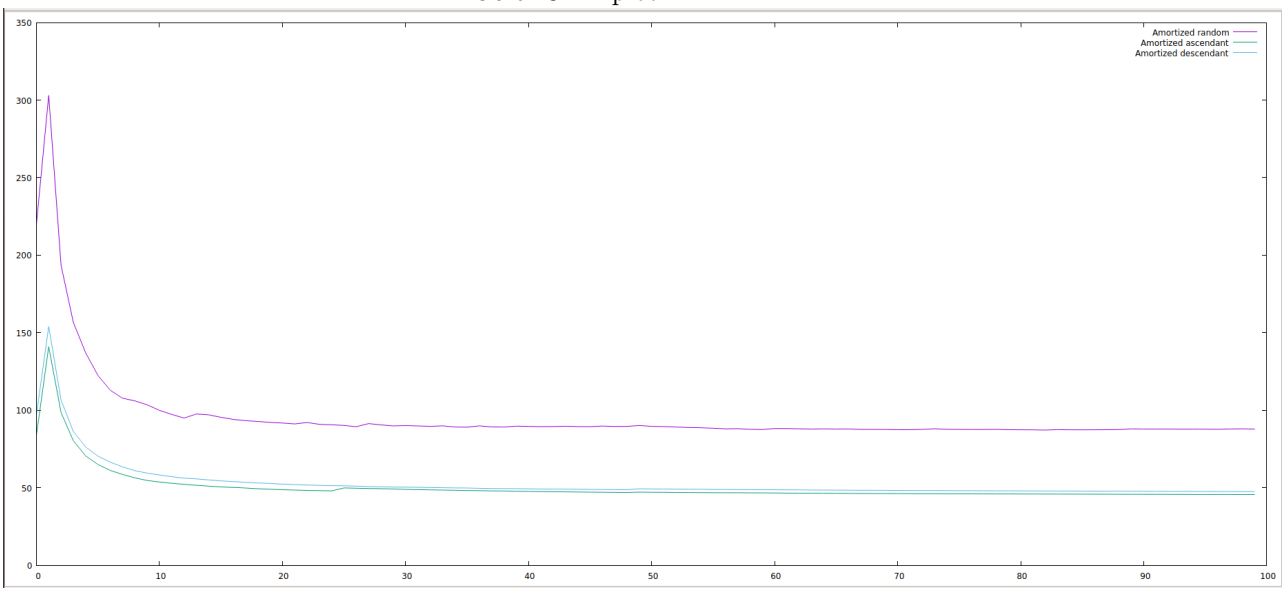
On y remarque que le cout d'insertion de valeurs aléatoire est nettement supérieur a celui des valeurs ascendante ou descendante et cela est du au fait que la fonction pushnode doit faire appel a reorgonizeheap a chaque insertion au pire des cas(que des valeurs non ordonnée) pour ce qui est des insertions en ascendant ou descendant c'est quasi similaire avec un coup un peu plus élevé pour l'insertion descendant car on doit parcourir tout l'arbre pour l'insertion Pour ce qui est des cout en mémoire pour les 3 premiers cas

On remarque que pour les 3cas le cout en mémoire est similaire car nous avons ici un tableau de taille fixe

4eme cas on alterne entre des insertion et des suppression de nœud

On remarque que la différence en termes de temps est très importante cela est dû au fait que la suppression effectue plusieurs changement sur l'arbre avant de retirer l'élément

FIGURE 3.2 – plot2



3.1.2 Remplacement du tableau taille fix par un tableau dynamique

On remplace notre tableau de taille fixe par un tableau dynamique et en relance le test

On a ici le cout en termes de mémoire qui est linéaire avec un tableau static et qui double de taille avec les tableau dynamique

Pour ce qui est du coup en temps

En vert on alterne entre suppression et insertion on remarque parfois des pique qui se traduit par une insertion ou une suppression d'un élément qui nécessite le parcours de tout l'arbre

FIGURE 3.3 – plot

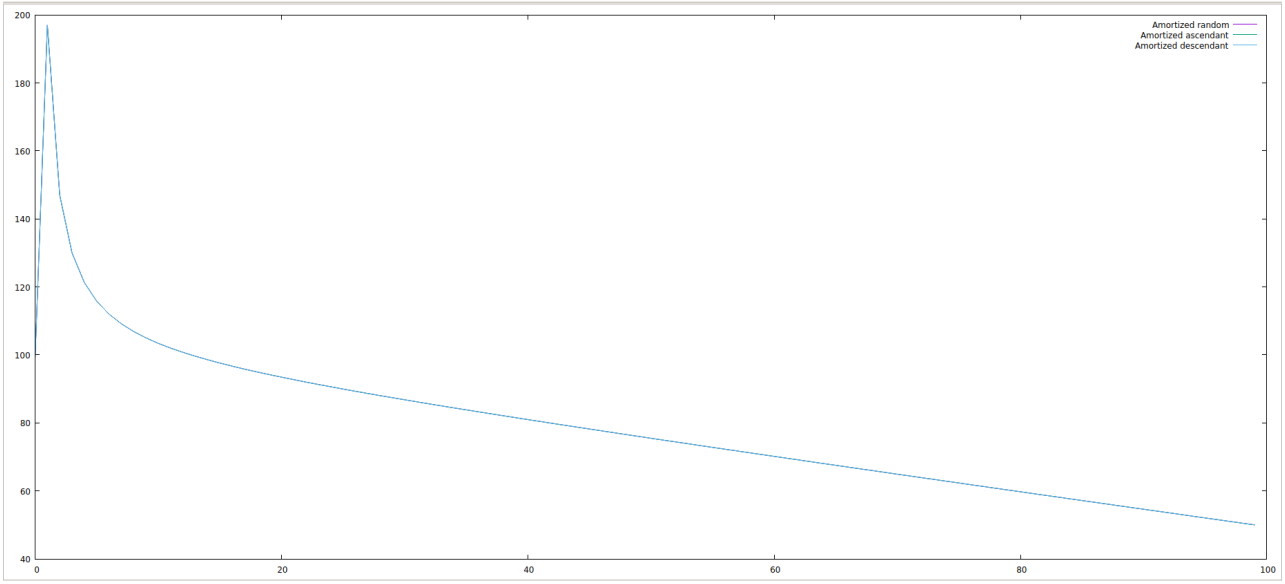


FIGURE 3.4 – plot3

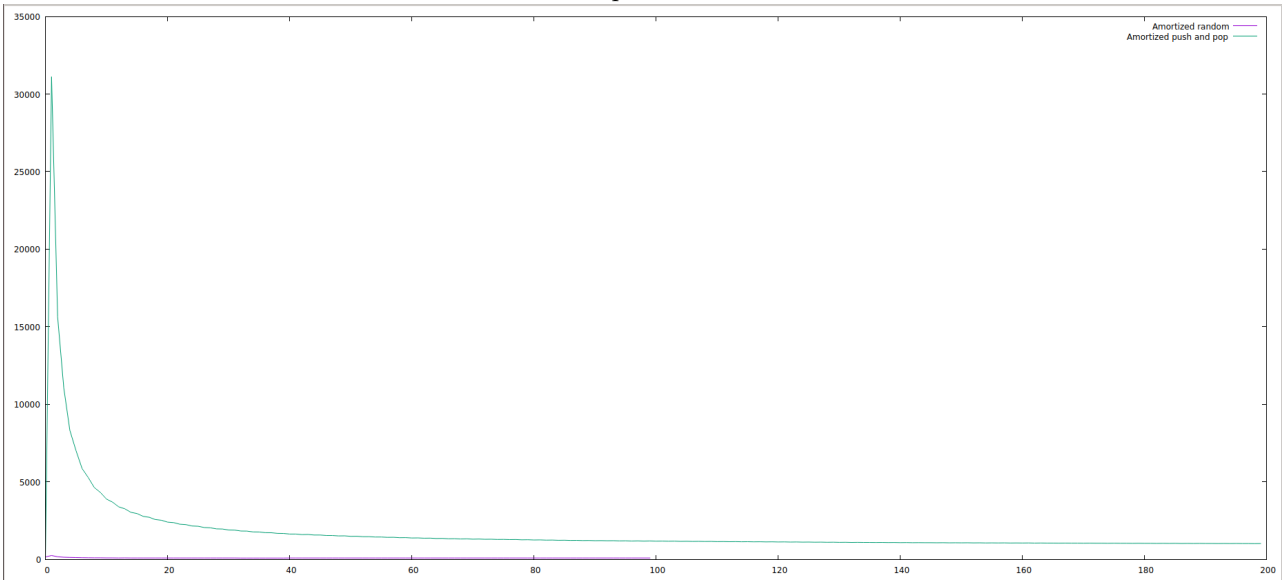


FIGURE 3.5 – plot4

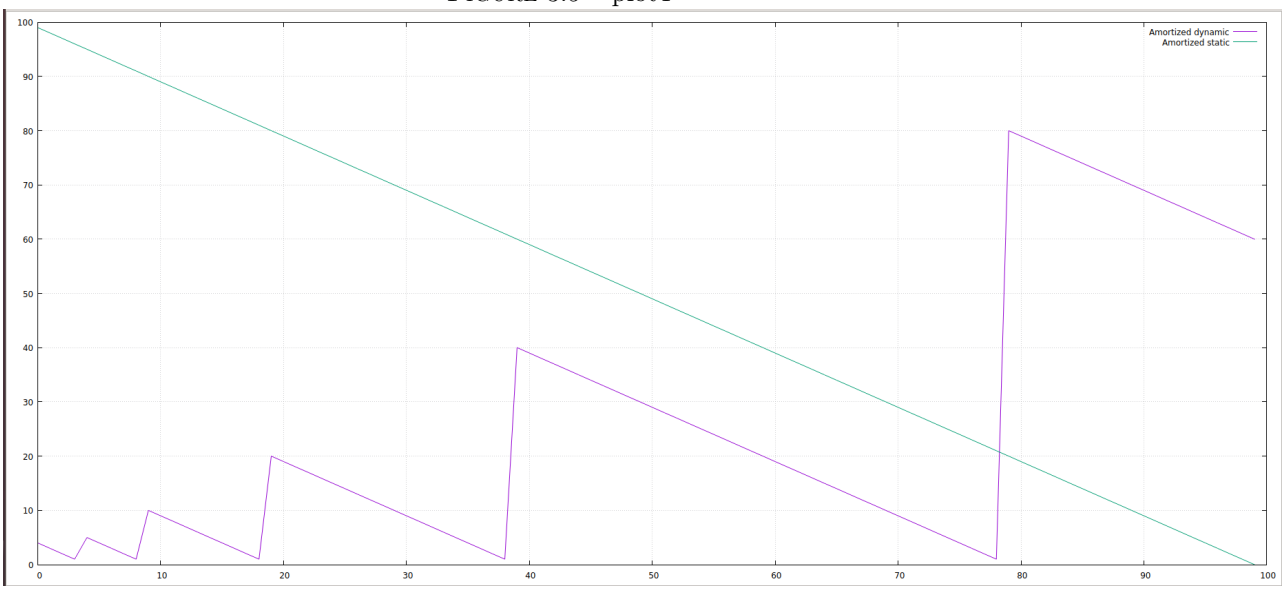
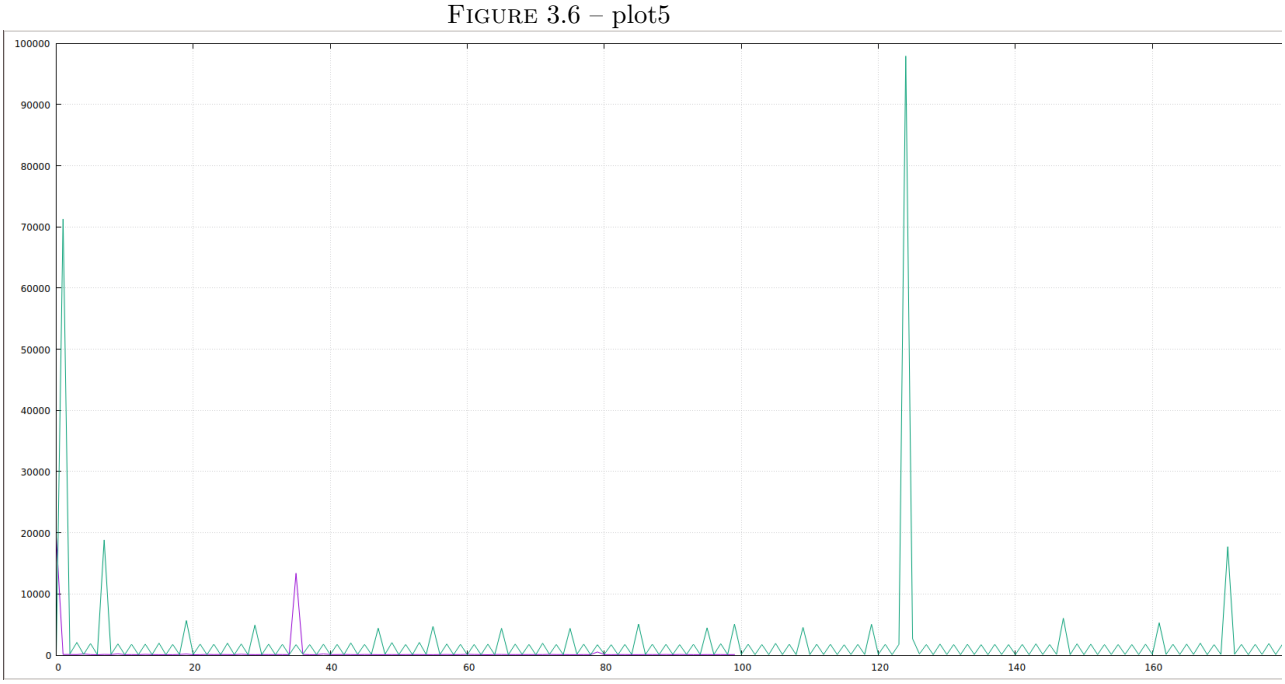


FIGURE 3.6 – plot5



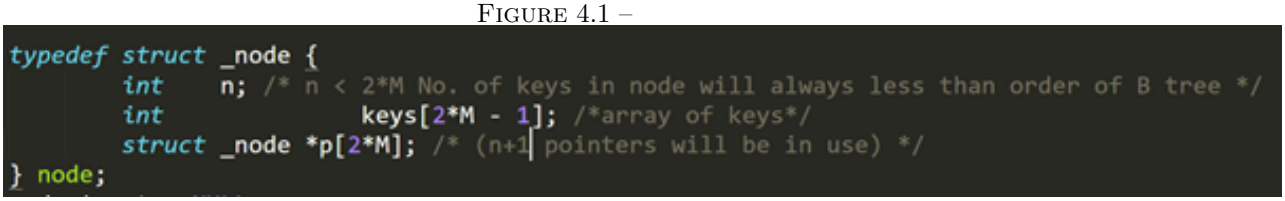
Chapitre 4

corrigé TP4 :

4.1 la création d'un B-arbre

Pour cette partie on a implémenté une structure de donnée pour représenter la structure B-arbre

FIGURE 4.1 –



On a un entier n qui stockera la numéro de clés qui sera inferieur a 2*M On a un tableau d'entier de taille 2*m-1 qui contiendra les clés Et une table de pointeur vers les différents fils Pour ce qui est des opération scindage et fusion on a définie une fonction ins qui retourne une énumération

Notre fonction commence par vérifier est ce que la clé en question existe déjà si c'est le cas elle retourne duplicate, sinon elle effectue les shift nécessaire pour l'insertion de notre élément

— cas 1 Arbre vide elle effectue l'insertion directement a la racine

FIGURE 4.2 –

```
typedef enum KeyStatus {
        Duplicate,
        SearchFailure,
        Success,
        InsertIt,
        LessKeys,
} KeyStatus;
```

FIGURE 4.3 –

```
KeyStatus ins(node *ptr, int key, int *upKey, node **newnode) {
        node *newPtr, *lastPtr;
        int pos, i, n, splitPos;
        int newKey, lastKey;
        KeyStatus value;
        if (ptr == NULL) {
                *newnode = NULL;
                *upKey = key;
                return InsertIt;
        }
```

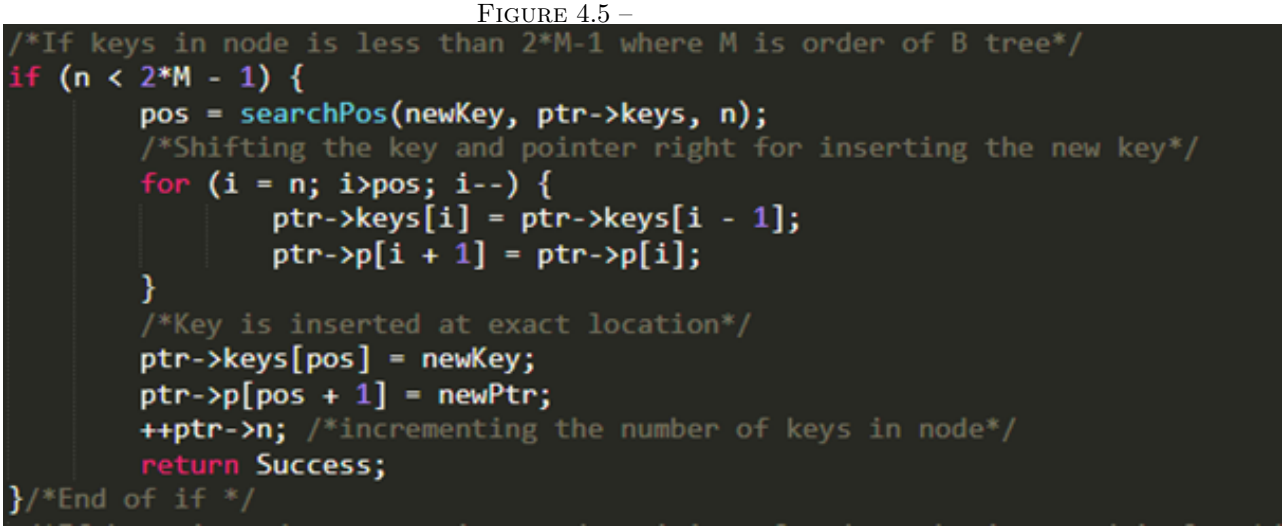
— cas 2 Notre clé existe déjà dans l'arbre

FIGURE 4.4 –

```
n = ptr->n;
pos = searchPos(key, ptr->keys, n);
if (pos < n && key == ptr->keys[pos])
        return Duplicate;
```

— cas 3 Si le nombre de clés dans le nœud est inferieur a 2*m-1 alors on décale la clés a

FIGURE 4.5 –



```
/*If keys in node is less than 2*M-1 where M is order of B tree*/
if (n < 2*M - 1) {
        pos = searchPos(newKey, ptr->keys, n);
        /*Shifting the key and pointer right for inserting the new key*/
        for (i = n; i>pos; i--) {
                ptr->keys[i] = ptr->keys[i - 1];
                ptr->p[i + 1] = ptr->p[i];
        }
        /*Key is inserted at exact location*/
        ptr->keys[pos] = newKey;
        ptr->p[pos + 1] = newPtr;
        ++ptr->n; /*incrementing the number of keys in node*/
        return Success;
}/*End of if */
```

droite pour insérer notre clés, on insère notre clé et on incrémente le nombre de clés dans le nœud puis on retourne Success pour notifier la réussite de l'insertion

— cas 4 i le nombre de clés est au maximum et que le nœud a insérer ce trouve naturellement

FIGURE 4.6 –

```
 /*If keys in nodes are maximum and position of node to be inserted is last*/
if (pos == 2*M - 1) {
        lastKey = newKey;
        lastPtr = newPtr;
}
else { /*If keys in node are maximum and position of node to be inserted is not last*/
        lastKey = ptr->keys[2*M - 2];
        lastPtr = ptr->p[2*M - 1];
        for (i = 2*M - 2; i>pos; i--) {
                ptr->keys[i] = ptr->keys[i - 1];
                ptr->p[i + 1] = ptr->p[i];
        }
        ptr->keys[pos] = newKey;
        ptr->p[pos + 1] = newPtr;
}
splitPos = (2*M - 1) / 2;
(*upKey) = ptr->keys[splitPos];

(*newnode) = (node*)malloc(sizeof(node));/*Right node after split*/
ptr->n = splitPos; /*No. of keys for left splitted node*/
(*newnode)->n = 2*M - 1 - splitPos;/*No. of keys for right splitted node*/
for (i = 0; i < (*newnode)->n; i++) {
        (*newnode)->p[i] = ptr->p[i + splitPos + 1];
        if (i < (*newnode)->n - 1)
                (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
        else
                (*newnode)->keys[i] = lastKey;
}
(*newnode)->p[(*newnode)->n] = lastPtr;
return InsertIt;
```

a la fin si ce n'est pas le cas alors on cherche cette position en question pour l'insertion, on effectue par la suite un scindage des nœud pour obtenir la forme souhaité et insérer notre élément en retournant INSERTIT a la méthode insert

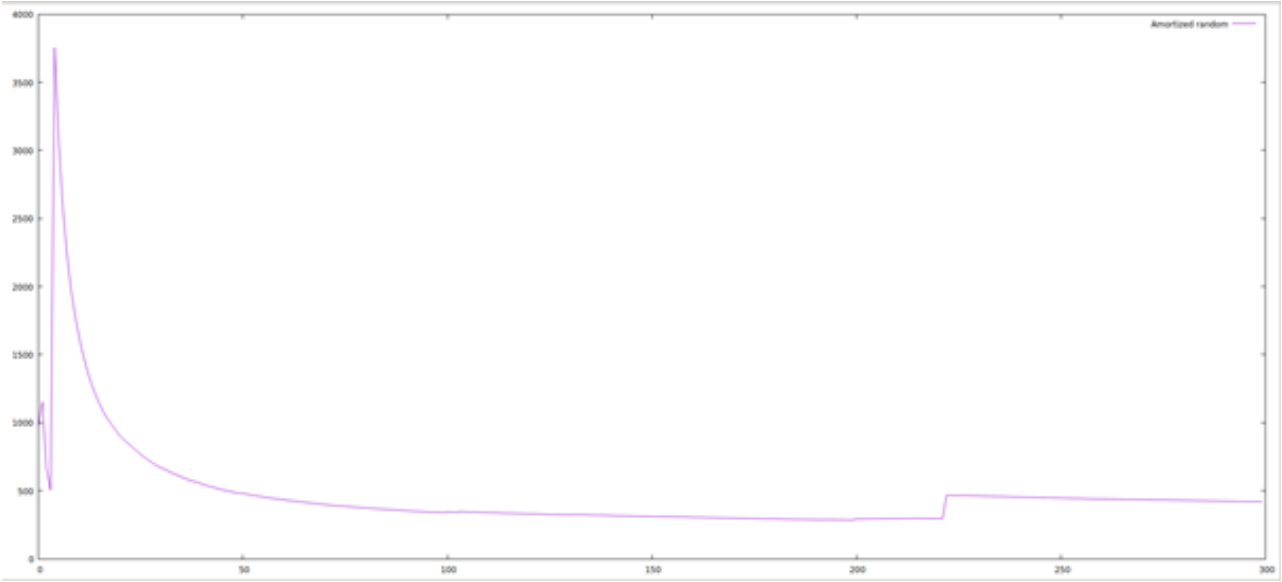
FIGURE 4.7 –

```
void insert(int key) {
        node *newnode;
        int upKey;
        KeyStatus value;
        value = ins(root, key, &upKey, &newnode);
        if (value == Duplicate)
                printf("Key already available\n");
        if (value == InsertIt) {
                node *uproot = root;
                root = (node*)malloc(sizeof(node));
                root->n = 1;
                root->keys[0] = upKey;
                root->p[0] = uproot;
                root->p[1] = newnode;
        }/*End of if */
}/*End of insert()*/
```

Puis cette dernière va allouer l'espace mémoire nécessaire et effectue l'insertion

Le cout en temps pour une insertion de 300 valeurs aléatoire dans notre Barbre on remarque quelque pique dû au fusionnement et scindement de nos nœud dans certaines insertions

FIGURE 4.8 –


Amortized random
4000
3500
3000
2500
2000
1500
1000
500
0
0
50
100
150
200
250
300

4.2 Implémentation AVL

FIGURE 4.9 –

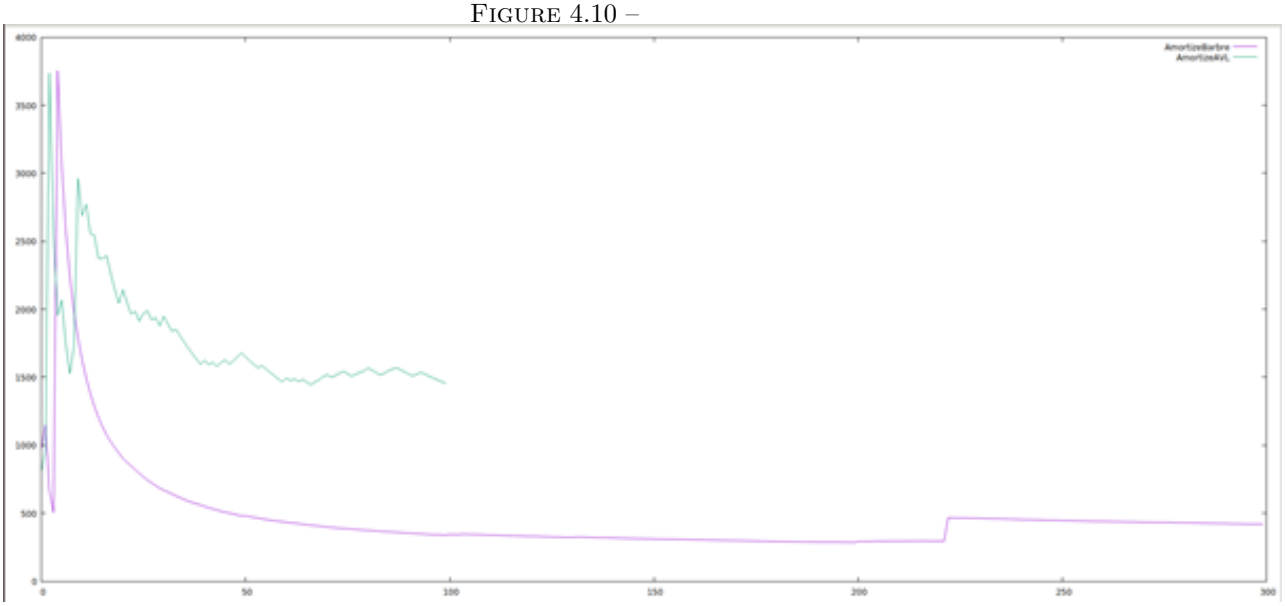
```
struct Noeud
{
 int Info;
 int Signe;
 struct  Noeud *fils_gauche; /* fils gauche (inferieur au noeud) */
 struct  Noeud *fils_droit;  /* fils droit (superieur au noeud) */
};
```

On a une structure avec une clés un fils gauche qui est inferieure a son père et un fils droit qui est supérieur à son père et une un signe qui nous permettra de savoir quelle action entreprendre par la suite dans nos fonction

Test d'insertion de valeurs aléatoire

4.3 Implémentation AVL

FIGURE 4.10 –



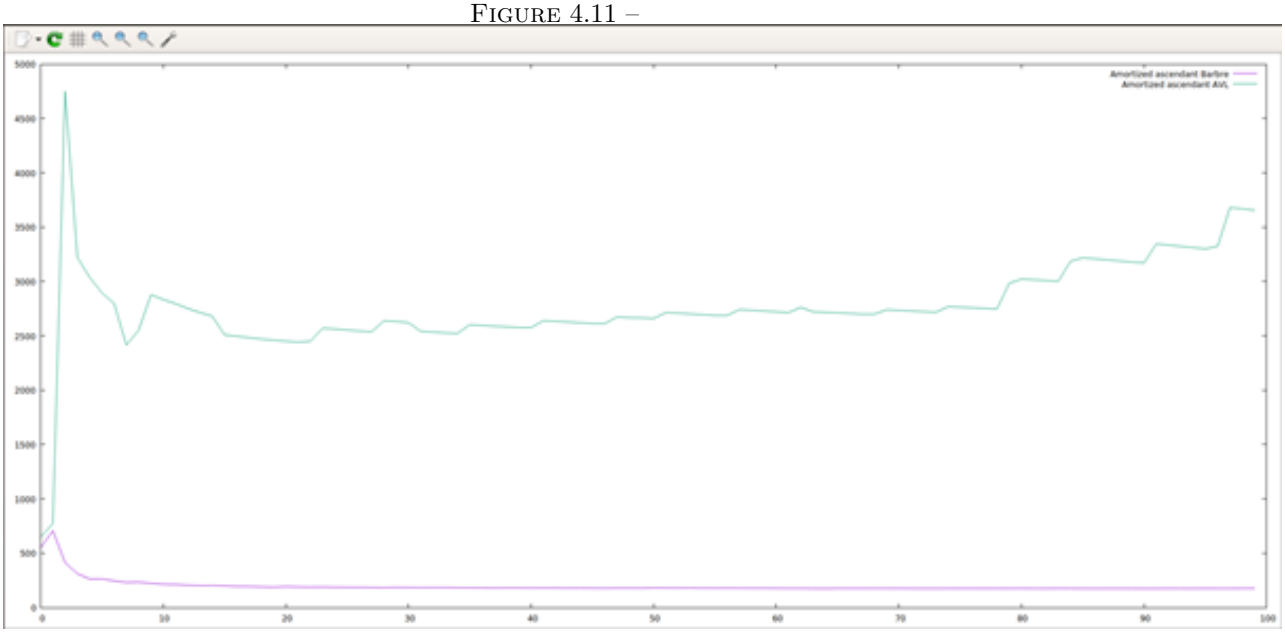
On remarque ici que le cout en temps et plus important sur les avl que sur les Barbre a notre avis cela est dû au rotation multiple effectué à chaque insertion afin de rééquilibré l'arbre

Test d'insertion de valeur ascendante

4.4 Implémentation AVL

On remarque une nette amélioration sur les Barbre mais ce ne fut pas le cas pour AVL car il est dans ce qu'on appel le pire cas car il est obligé d'effectuer des rotation à chaque insertion

FIGURE 4.11 –


Amortized ascendant Barbre
Amortized ascendant AVL
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90
100