Institut Galilée - Université Paris 13

Structure de Données Avancées
Rapport

Travaux Pratiques - TP 01

Étudiants :
Tiziri OULD HADDA

Enseignants :
M. Olivier BODINI
M. Julien DAVID
M. Sergey DOVGAL

30 décembre 2019

Table des matières

1 Introduction

Dans ce TP, on va faire l'étude de la compléxité des tableaux dynamiques on calculant le cout réel et le cout amorti de l'algorithme d'insertion dans les tableaux dynamiques. On va tout d'abort analyser la compléxité en temps et en espace puis étudier différentes statégie d'allocation mémoire. il est aussi question de comparer aussi cette structure implémentée avec différents languages de programmation.

Le but de ce travail est d'analyser le cout amorti et d'essayer de minimiser l'espace mémoire innutilisé (dynamiquement) par cette structure et de trouver un meilleur compromis entre l'espace mémoire inutilisé et l'espace mémoire utilisé ainsi que le cout amorti.

2 Structure du programme

Afin d'analyser le programme donnée par le code source initial. On a va utiliser la structure/classe *analyzer* et programme *gnuplot* de la manière suivante :

2.1 Analyzer

Structure/classe conçu pour calculer des couts réels et amortis et permet aussi de stocker les couts de chaque opération d'insertion, de les cumuler et de les sauvegarder dans des fichiers de sortie (extension *.plot*). Ce fichier est constitué de trois colonnes (séparée par un espace) :

- numéro de l'opération.
- le cout réel
- le cout amorti

2.2 gnuplot

Une fois le fichier de sortie généré, on utilise le programme gnuplot pour tracer le contenu de ces derniers dans des graphes. On peut à partir du shell lancer un script (contanant des commandes *gnuplot*) afin d'afficher ses courbes et/ou les stoker dans un fichier image '*png*' comme suit :

- lancement du script gnuplot :
  ```
  gnuplot -c gnuplot_script --persist
  ```
- exemple de script gnuplot (définir le format, le nom et la taille de l'image puis tracer le contenu du fichier plot) :
  ```
  set terminal png size 800,600;
  set output 'fichierDeSortie.png';
  plot [0:1000000][0:2000] 'fichier.plot' using 1:3 w lines title "Titre
  du graphe";
  ```

2.3 lancement des expériences

Afin de permettre une automatisation des expériences, j'ai procéder tout le long des TP à la méthode suivante :

- modification du code source selon la question.

— lancement d'un script bash qui recompile, lance les applications, cré un repértoire propre à l'expérience en cours et lance le script gnuplot.
— le script prend un argument portant le nom de lexperience lui permettant de créer le répertoire et de renommer les fichiers plot et les fichiers image à l'intérieur du répértoire.

Les scripts sont mis dans racine de chaque TP. Voir Annexe pour l'ensemble des script du TP1. En utilisant cette méthode, il m'a suffit juste de changer le code et d'énumérer les tests à faire. Elle m'a permis de gagner énormément de temps pour faire les expériences demandées dans les TP.

3 Analyse du tableau dynamique "ArrayList"

3.1 Etude théorique

La fonction de potentielle est donnée par la relation suivante :

$a(n) = c(n) + \Phi(n)\Phi(n-1)$:

— $c(n)$: coût pour la n_{ieme} operation
— $a(n)$: coût amorti pour la n_{ieme} operation.
— $\Phi(n)$: Fonction de potentiel.
— tel que $a(n) > 0$,pour quel que soit n.

Fonction potentiel : Tableau n'est pas plein

$$\begin{aligned}
\Phi(n) &= 2.n - capacite \\
a(n) &= c(n) + \Phi(n) - \Phi(n-1) \\
&= 1 + 2.n - capacite_i - 2.(n-1) + capacite_{i-1} \\
&= 1 + 2.n - capacite_i - 2.n + 2.capacite_i \\
&= 1 + 2 + 0 \\
a(n) &= 3
\end{aligned}$$

Dans le calcul précédent on a vu que la taille ne change pas, donc elle ne dépend pas de la capacité. Comme le nombre d'éléments présents avant est égale au nombre d'éléments actuels + 1 => ce qui fait qu'elle ne dépend pas du nombre d'élément, donc le coût amorti est constant et vaut 3.

Fonction potentiel : Tableau est plein

$$\begin{aligned}
\Phi(n) &= 2.n - capacite \\
a(n) &= c(n) + \Phi(n) - \Phi(n-1) \\
&= 1 + 2.n - capacite_i - 2.(i-1) + capacite_{i-1} \\
&= n + 2.n - 2.n - 2.n + 2 + n \\
&= 4.n - 4.n + 2 \\
a(n) &= 2
\end{aligned}$$

On sait que lorsque le tableau est plein et pour ajouter un élément, il faut donc créer un autre tableau avec une taille double :

$$\begin{aligned} taille_i = 2.n = 2.taille_i - 1 \\ \#element_i = n + 1 \\ \#element_i - 1 = n \end{aligned}$$

Lorsqu'on écrit tout en fonction de n, on s'apercevoit qu'on peut simplifier les termes de l'expression précédente.

Coût amorti insertion

On augmente la taille du tableau d'une constante α, avec $\alpha \geq 1$. La fonction de potentielle proposée en TP est la suivante :

$$\Phi(n) = \frac{\alpha.n - capacite}{\alpha - 1}$$

Coût amorti insertion : tableau n'est pas plein

$$\begin{aligned} a(n) &= c(n) + \Phi(n) - \Phi(n-1) \\ &= 1 + \frac{\alpha.n - capacite}{\alpha - 1} - \frac{\alpha.(n-1) - capacite}{\alpha - 1} \\ &= 1 + \frac{\alpha.n}{\alpha - 1} - \frac{capacite}{\alpha - 1} - \frac{\alpha.n}{\alpha - 1} + \frac{\alpha}{\alpha - 1} + \frac{capacite}{\alpha - 1} \end{aligned}$$

donc

$$a(n) = 1 + \frac{\alpha}{\alpha - 1}$$

— 1. la taille ne change pas donc la capacité restera la même.
— 2. le nombre d'éléments précédent c'est le nombre d'éléments actuel moins un.

Coût amorti insertion : tableau est plein

$$\begin{aligned} a(n) &= c(n) + \Phi(n) - \Phi(n-1) \\ &= n + \frac{\alpha.n - \alpha.n}{\alpha - 1} - \frac{\alpha.(n-1) - n}{\alpha - 1} \\ &= \frac{n.(\alpha - 1) - \alpha(n-1) + n}{\alpha - 1} \\ &= \frac{\alpha.n - n - \alpha.n + \alpha + n}{\alpha - 1} \\ a(n) &= \frac{\alpha}{\alpha - 1} \end{aligned}$$

— 1. puisqu'on augmente la taille du tableau de α alors la taille après réallocation vaut $\alpha.n$.
— 2. le nombre d'élément après réallocation vaut n et après insertion vaut $n + 1$.

3.2 Analyse des expérience et comparaison entre les languages de programmation

Afin de mieux observer la différence des couts amorti il est important de connaitre la spécificité de chaque language afin de mieux comprendre les graphes suivants. .

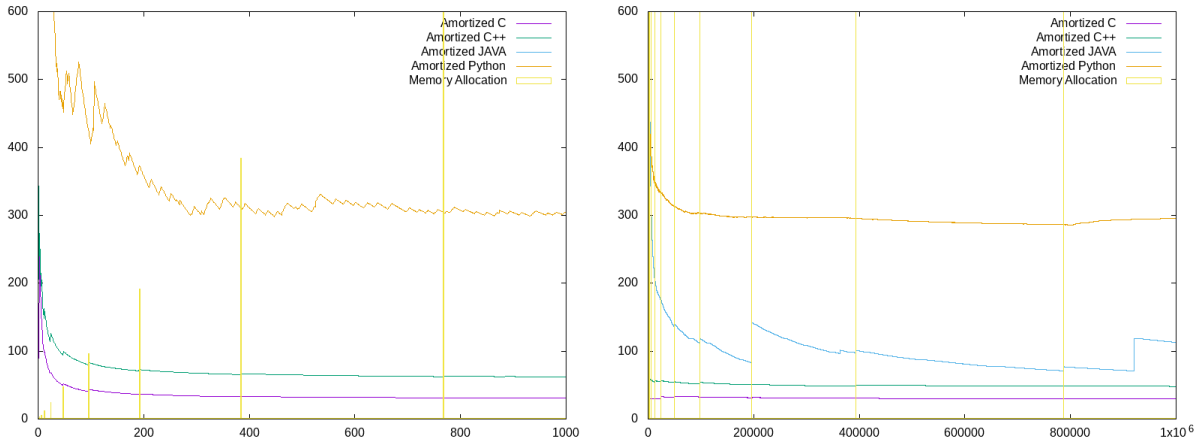

FIGURE 1 – Couts amortis avec differents languages de programmation

Selon le language utilisé on remarque sur la figure 1 que le cout amorti est grandment différent et devient de plus en plus important lorsqu'on augmente le niveau d'abstraction du language. On effet, le language le plus proche de la machine étant le "C" cela démontre le cout qui est le plus faible avec celui ci par rapport aux autres languages de programmation.

Tous les programmes testés s'exécutent sur système d'exploitation (dans mon cas *"Linux"*) et de plus pour certains code comme Python et Java sont exécutés sur une machine virtuelle et aussi à leur tour pocèdent leurs propres spécificités.

3.2.1 Différence entre C/C++ :

- Malgré la compléxité du language C++ (notions de classe, heritage, polymorphisme, exception...etc), il reste cependent très proche du C en terme de temps d'execution.
- Cette compléxité rajoute encore du code en plus par rapport au C ralangeant un peu le temps d'execution.

3.2.2 Particularité du Java :

- Le binaire JAVA obtenu après la compilation du code source est appelé bytecode (l'équivalent de l'assembleur). Le byte code est executé et traité par une machine virtuelle installé sur le système d'exploiration (2 couche d'abstraction).
- Le JAVA offre un avantage en terme de gestion de la mémoire (comme le *garbage collector*) mais cette gestion est exécuté par un autre processus surveillant les objets non pointés pour les désalloués de la mémoire. Cela en effet, ralenti le programme car il consomme des ressources CPU.

3.2.3 Particularité du Python :

— Le Python est un language interprété, il est executé par un interpreteur.
— Le Python possède comme le Java un *garbage collector*. Ce qui fait qu'il n'est pas aussi rapide que le Java.
— Lors de l'execution d'un script Python : l'interpreteur parse séquentiellement le programme (procède donc à plusieurs vérification et régles de programmation Python).
— Le typage dans Python est réalisé à l'exécution (typage dynamique) par l'inférence de type, par exemple. Cela veut dire que pour toutes expression de ce language celle ci est evaluée (à l'exécution) pour déterminer son type.

Plus le language est abstrait, plus le temps effectué pour chaque instruction de base (arithmétique/logique/affectation...etc) augmente et pouvant être aussi proche des operations un peu plus lourde comme celle de l'allocation mémoire. On observe cela, dans la figure 1 avec le language Python (pas de pic dans le cout amorti lors de l'allocation mémoire).

3.2.4 Répétabilité des expériences

On remarque des figure 2 et 3 que malgré que les paramètre de l'algorithme sont restés fixes, le résultat de chaque expérience est différent d'une autre. En effet, lorsqu'on on exécute le programme sur un système d'exploitation, ce dernier essaye d'attribuer les ressources CPU d'une manière la plus equitable possible de sorte a ce que tout les processus aient tous la chance de s'executer. C'est à cause, de cela que les résultat de nos expériences vaciés aussi, car cela dépend de l'état des autre application aussi qui pourrait demander plus de ressource a un moment donné.

De plus, la gestion de la mémoire est une partie très complexe dans les systèmes d'exploitation. Car quand un programme fait appel à malloc, il fait appel à l'appel systeme correspondant à l'allocation mémoire et reserve la mémoire en fonction de l'état de celle ci. De plus, la présence de cache dans les CPU rajoute aussi de l'incertitude (résultat variant) dans nos experiences.

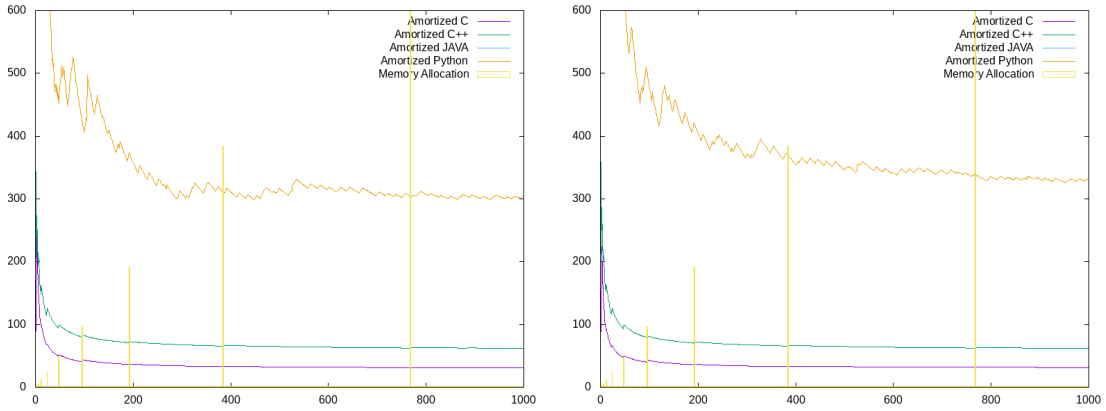

FIGURE 2 – Experience 1 et 2 avec differents languages de programmation

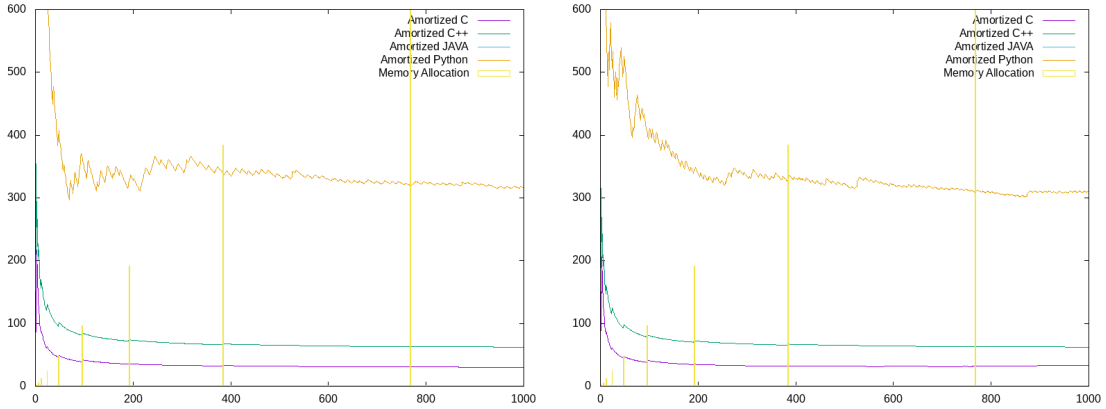

FIGURE 3 – Experience 3 et 4 avec differents languages de programmation

3.3 Analyse du programme réalisé en C

J'ai choisi de travailler tout le long des TP en language C car c'est le language le plus proche de la machine et qui offre au développeur une plus grande fléxibilité quant à la gestion de la mémoire. Afin d'analyser finement le cout de chaque instruction le C est donc le plus adapté.

3.3.1 Analyse en nombre de copie et allocation mémoire

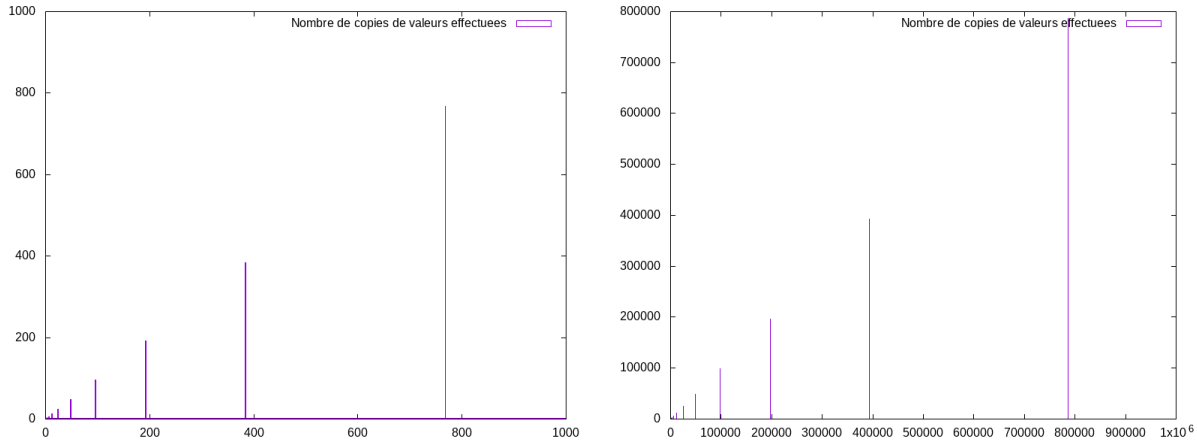

FIGURE 4 – Graphe de nombre de copies - language C

De la figure 1 et 4, on remarque que l'augmentation du cout amorti coincide exactement avec les pic de la figure 4 indiquant une allocation mémoire en vue d'augmenter la capacité du tableau dynamique. Cette allocation, d'après de la section précedente sur la difference entre les language, est moins notable pour les language C et C++ (voir figure 5)

Au début de l'expérience (de 0 à 1000 iterations) le cout amorti est plus grand. cela est du au faite que c'est dans cette interval que le programme procède beaucoup plus souvent à l'allocation dynamique et le nombre de copie à faire. Dans ce cas :

$$Cout(total) = Cout(realloc) + Cout(Insert)$$
$$Cout(total) \approx C(realloc)$$

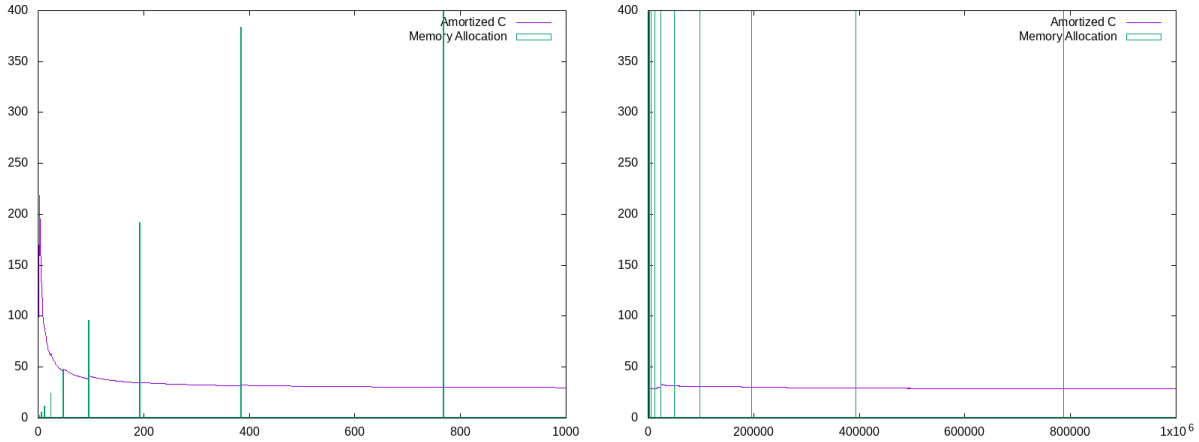

FIGURE 5 – Cout amorti - Language C - interval 0 à 1000 et interval complet

Par contre, vers la fin de l'experience (apres 1000000 iteration par exemple) ces operation arrivent moins souvent car il faudra attendre le remplissage du tableau (ou atteindre un seuil fixé par la stratégie choisie).

$$Cout(total) = Cout(realloc) + Cout(Insert)$$
$$Cout(total) \approx Cout(Insert)$$

Au début de l'expérience il apparait bien que le Python à un cout amorti inférieure à celui du Java. Cela, en effet, s'explique par le fait que le code Python est beaucoup plus abstraite et cela revient à dire que toute les instruction python (meme celle d'allocation mémoire) on un cout presque equivalent car le cout le plus important est donc celui de abstraction. Quant au Java, celui ci est moins abstrait et donc la différence de cout de chaque instruction reste notable.

3.3.2 Analyse en espace mémoire inutilisé

De la figure 6, on remarque que l'allocation mémoire survient lorsqu'on arrive à 3/4 de la capacity du tableau dynamique. cela reveint à dire que dans tout les cas 1/4 de la mémoire ne sera jamais utilisée.

Changement la condition d'extension de capacité

```
char arraylist_do_we_need_to_enlarge_capacity(arraylist_t * a){
  return ( a->size >= (a->capacity * 3)/4 )? TRUE: FALSE;
}

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

L'avantage de passer à capacité plein dans la condition d'extension de la table dynamique est que la réallocation mémoire survient moins souvent. Par contre, en augmentant la condition de 3/4 à plein cela reviens a dire que le nombre de copie passe aussi de 3/4 à Capacité complete de copie à éffectuer ce qui fait que le cout amortie augmente aussi à

son tour.

Par contre on ne sait pas quand l'utilisateur voudra arreter d'ajouter de élements à notre "ArrayList". S'il décide d'avoir au maximum un peu plus de la précédente capacité cela est très problématique car on va perdre presque le double de cette capacité en mémoire perdu. Il faut donc trouver le bon compromi selon le cas.

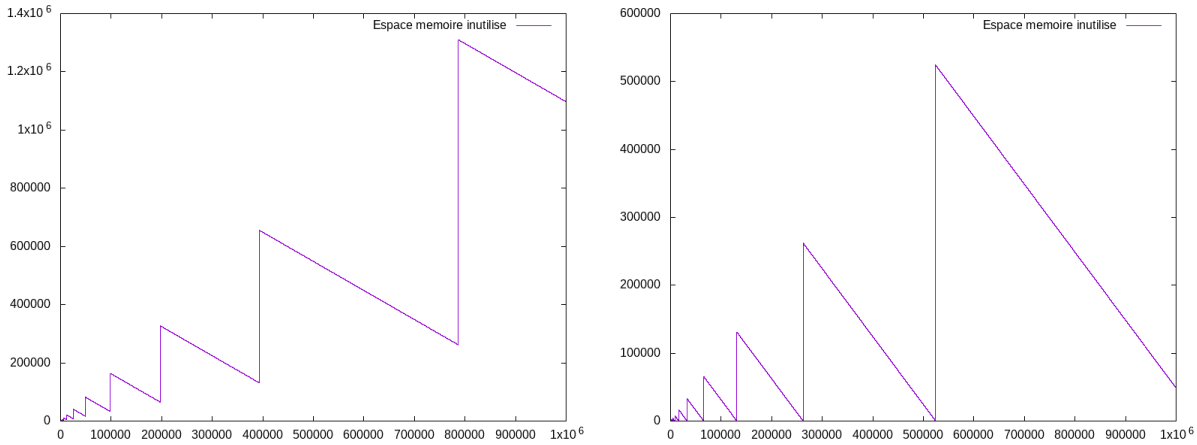

FIGURE 6 – Mémoire inutilisée - Language C - Elargissement de la capacité à 3/4 et capacité complète

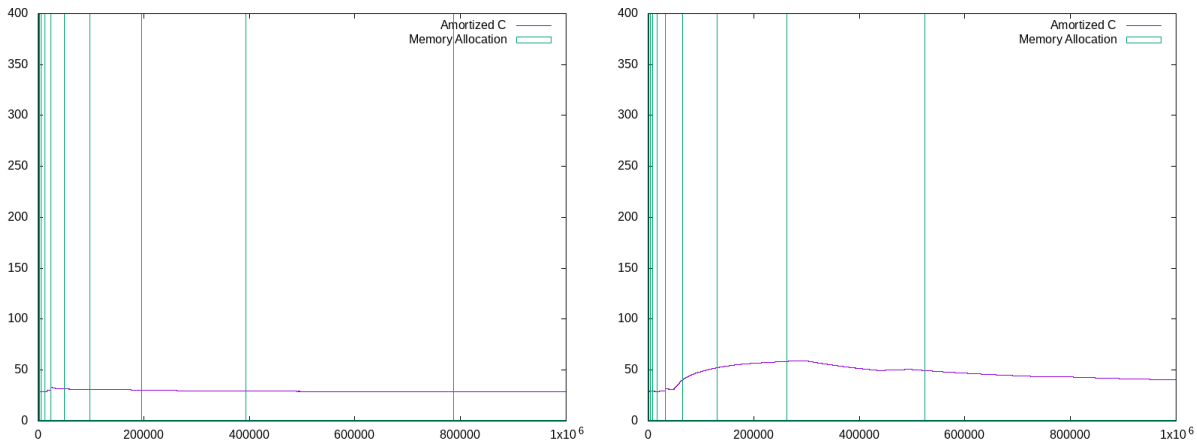

FIGURE 7 – Cout amorti - Language C - Elargissement de la capacité à 3/4 et capacité complète

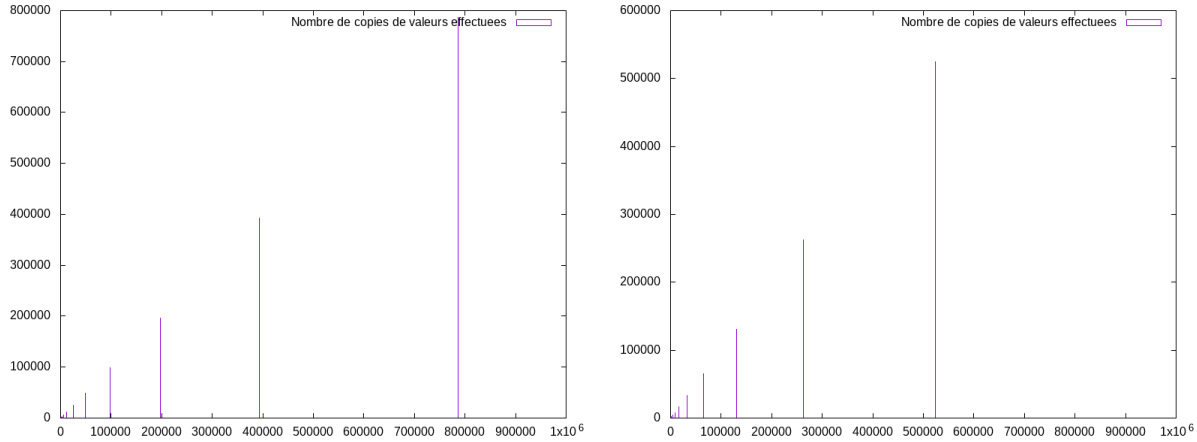

FIGURE 8 – Nombre de copies - Language C - Elargissement de la capacité à 3/4 et capacité complète

3.3.3 Elargir capacité : Changement de la valeur alpha 1.25, 2 et 3

L'avantage d'augmenter la valeur de alpha est que la réallocation mémoire survient moins souvent. Par contre, le nombre d'espace inutilisé augmente (figure 12). Le cout amortie par contre semble ne pas changer pour un nombre d'élement trop grand. Par contre pour un nombre petit d'élement cela risque d'augmenter le cout amorti.

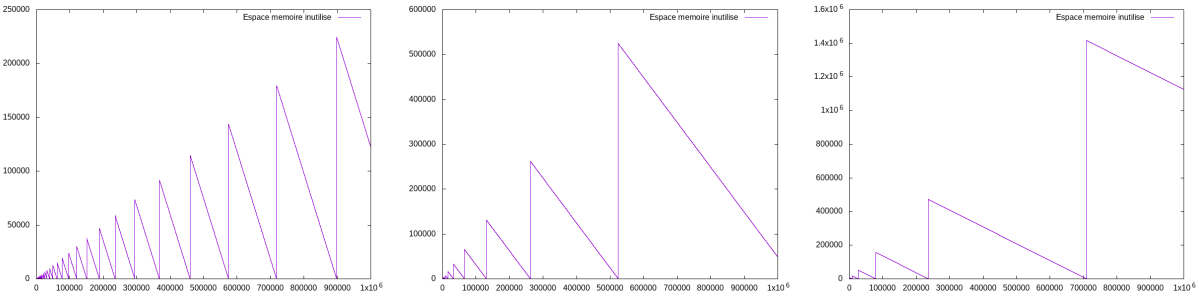

FIGURE 9 – Mémoire inutilisée - Langage C - α =1.25, 2 et 3

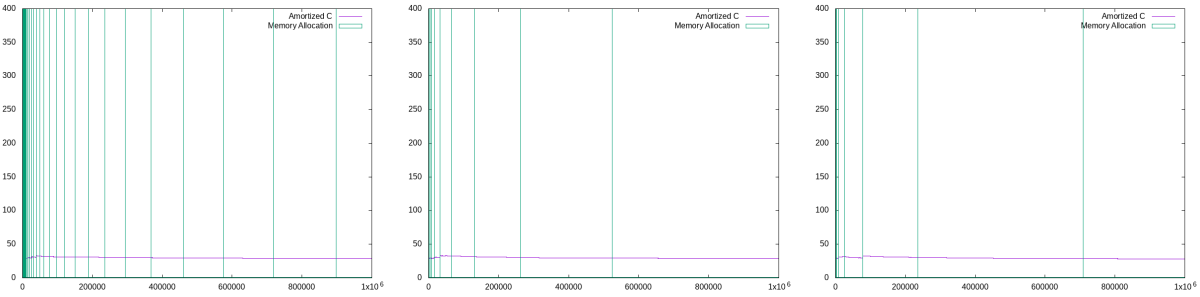

FIGURE 10 – Cout amorti - Langage C - α =1.25, 2 et 3

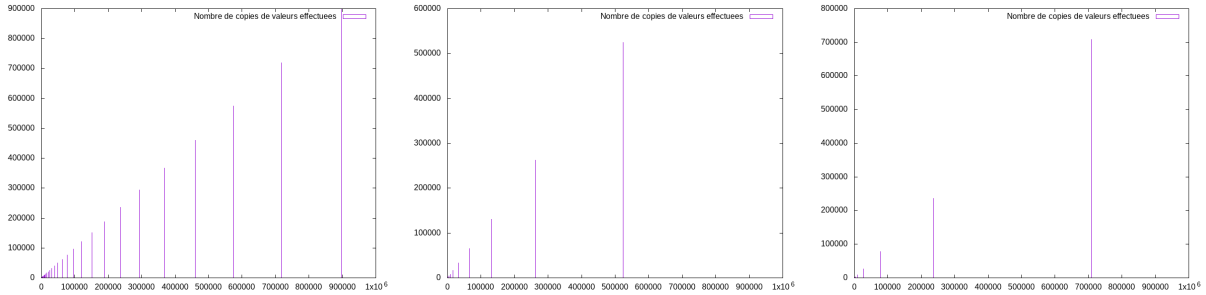

FIGURE 11 – Nombre de copie - Langage C - α =1.25, 2 et 3

3.3.4 Elargir la capacité : $n + \sqrt{n}$

Inconvénient majeur de cette méthode et que l'allocation mémoire survient trop souvent donc compléxité en espace très importante car la nombre de copie survient régulierement. Par contre la compléxité en temps semble augmenté mais très légèrement.

```
#include "arraylist.h"
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
...
char arraylist_do_we_need_to_enlarge_capacity(arraylist_t * a){
  return ( a->size >= a->capacity )? TRUE: FALSE;
}

void arraylist_enlarge_capacity(arraylist_t * a){
  a->capacity = a->capacity + sqrt(a->capacity);
  a->data = (int *) realloc(a->data, sizeof(int) * a->capacity);
}
```

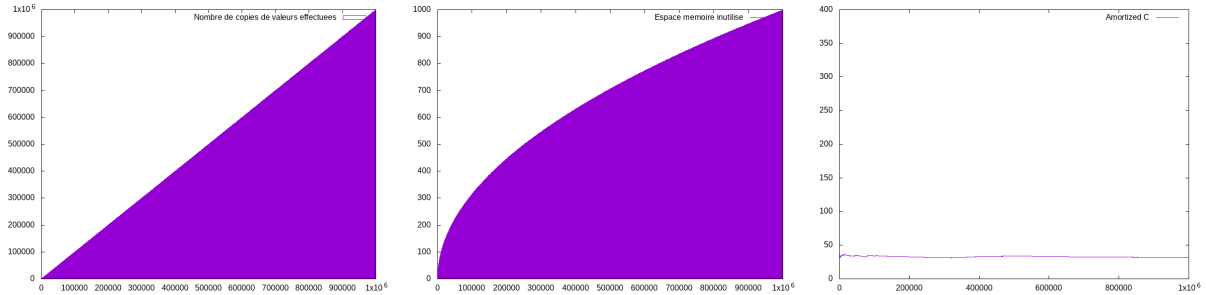

FIGURE 12 – Mémoire inutilisée - Langage C - $n + \sqrt{n}$

4 Conclusion

— On ne peux pas avoir une compléxité en temps et en espace en même temps meuilleurs. il faudra donc faire un compromis selon les besoin de l'application (but de l'application) pour fixé un paramètre alpha et un seuil d'extension de la capacité du tableau dynamique.
— On a vu dans ce TP, que fixer un seuil à capacité pleine permet de minimiser l'espace minimum perdu, contrairement à d'autre stratègie comme la capacité à 3/4 ou on a une perte minimum (dans tout les cas) à 1/4.
— Le faite de doublé, minimise le fait que la copie et l'allocation surviennent rarement par rapport à d'autre stratègie comme $n + \sqrt{n}$.

5 ANNEXE

5.1 Script Bash pour les questions sur tous les languages

Inspiré du README.md dans les source de github :

```
echo "Nom de l'experience : $1 nbfois: $2"

rm -rf  plots/*.png
rm -rf  plots/*.plot

rm -rf plots/exp_$1
mkdir  plots/exp_$1

for i in `seq 1 $2`;
do
echo ""
cd C/
make
./arraylist_analysis
make clean
cd ..

cd CPP/
make
./arraylist_analysis
make clean
cd ..

cd Java
javac *
java Main
cd ..

cd Python
python main.py
cd ..

cd plots/
gnuplot -c plot_result --persist

rm -rf exp_$1_$i
mkdir  exp_$1_$i

cp -rf *.plot exp_$1_$i/
cp -rf *.png  exp_$1_$i/

cd exp_$1_$i
```

```
for f in *
do
pre="$1_$i"
mv "$f" "../exp_$1/${pre}_$f"

done

cd ../
rm -rf exp_$1_$i/
echo "Experience exp_$1_$i OK"
cd ../

echo ""
done
```

5.2 Script gnuplot pour les questions sur tous les languages

Inspiré du fichier plot_result dans le répertoire plot et quelques documentations et forum (FAQ) sur gnuplot.

```
set terminal png size 800,600;

set output 'cout_amorti_zoom_languages_tp1.png';

plot [0:1000][0:600] 'dynamic_array_time_c.plot' using 1:3 w lines title
"Amortized C", 'dynamic_array_time_cpp.plot' using 1:3 w lines title "Amortized
C++", 'dynamic_array_time_java.plot' using 1:3 w lines title "Amortized JAVA",
'dynamic_array_time_python.plot' using 1:3 w lines title "Amortized Python",
'dynamic_array_copy_c.plot' using 1:2 w boxes title "Memory Allocation"

set output 'cout_amorti_languages_tp1.png';
plot [0:1000000][0:600] 'dynamic_array_time_c.plot' using 1:3 w lines title
"Amortized C", 'dynamic_array_time_cpp.plot' using 1:3 w lines title "Amortized
C++", 'dynamic_array_time_java.plot' using 1:3 w lines title "Amortized JAVA",
'dynamic_array_time_python.plot' using 1:3 w lines title "Amortized Python",
'dynamic_array_copy_c.plot' using 1:2 w boxes title "Memory Allocation"

set output 'cout_reel_zoom_languages_tp1.png';
plot [0:1000][0:600] 'dynamic_array_time_c.plot' using 1:2 w lines title "Real
cost C"

set output 'cout_reel_languages_tp1.png';
plot [0:1000000][0:600] 'dynamic_array_time_c.plot' using 1:2 w lines title
"Real cost C"

set output 'cout_amorti_c_mem_inutilise_tp1.png';
plot 'dynamic_array_memory_c.plot' using 1:2 w lines title "Espace memoire
```

```
inutilise"

set output 'cout_amorti_c_copy_tp1.png';
plot 'dynamic_array_copy_c.plot' using 1:2 w boxes title "Nombre de copies de
valeurs effectuees"

set output 'cout_amorti_zoom_c_copy_tp1.png';
plot [0:1000][0:1000] 'dynamic_array_copy_c.plot' using 1:2 w boxes title
"Nombre de copies de valeurs effectuees"
```

5.3 Script bash pour les questions sur le language C

```
echo "Nom de l'experience : $1"
rm -rf  plots/*.png
rm -rf  plots/*.plot
cd C/
make
./arraylist_analysis
make clean
cd ../plots/
gnuplot -c plot_result_c --persist

rm -rf exp_$1
mkdir exp_$1

cp -rf *.plot exp_$1/
cp -rf *.png  exp_$1/
cd exp_$1

for f in *
do
mv -- "$f" "$1_$f"
done
cd ../../
```

5.4 Script gnuplot pour les questions sur le language C

```
set terminal png size 800,600;

set output 'cout_amorti_c_tp1.png';
plot [0:1000000][0:400] 'dynamic_array_time_c.plot' using 1:3 w lines title
"Amortized C", 'dynamic_array_copy_c.plot' using 1:2 w boxes title "Memory
Allocation"

set output 'cout_amorti_zoom_c_tp1.png';
plot [0:1000][0:400] 'dynamic_array_time_c.plot' using 1:3 w lines title
"Amortized C", 'dynamic_array_copy_c.plot' using 1:2 w boxes title "Memory
```

```
Allocation"

set output 'cout_amorti_c_mem_inutilise_tp1.png';
plot 'dynamic_array_memory_c.plot' using 1:2 w lines title "Espace memoire
inutilise"

set output 'cout_amorti_c_copy_tp1.png';
plot 'dynamic_array_copy_c.plot' using 1:2 w boxes title "Nombre de copies de
valeurs effectuees"

set output 'cout_amorti_zoom_c_copy_tp1.png';
plot [0:1000][0:1000]'dynamic_array_copy_c.plot' using 1:2 w boxes title "Nombre
de copies de valeurs effectuees"
```