# Tp2

## 1)montrer que le coût amorti d'une opération Supprimer(T, x) qui fait appel à cette stratégie est majoré par une constante

Quand $\alpha_i = (n_i/taille_i) = 1$, on fait l'extension *2
Et $\alpha_i <= 1/4$, on fait la contraction *1/2
Et $\alpha_i <= 1/3$ , on fait la contraction *2/3

On a $\alpha_i = (n_i/taille_i) <= 1/3$ , $t_i = 2/3*t_{i-1}$
$\Phi(i) = |2*nom_i - taille_i|$

Calculer le cout amorti c de l'opération **supprimer (T,x):**
On $c = ci + \Phi(Di) - \Phi(Di-1)$ .

**Cas1: pas de contraction**
On sait $n_{i-1} = n_i + 1$ et $t_{i-1} = t_i$
$\alpha_{i-1} = (n_{i-1}/t_{i-1}) > 1/3$ 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 $C = 1 + |2n_i - t_i| - |2n_{i-1} - t_{i-1}|$
$C = 1 + |-2n_i + t_i| - |2(n_i + 1) - t_i|$
$C = 1 + t_i - 2n_i - (t_i - 2(n_i + 1))$
$C = 1 + t_i - 2n_i + 2n_i + 2 - t_i$
$C=3$

**Cas2: avec contractions**
On sait que: $n_{i-1} = n_i + 1$ et $t_{i-1} = (3/2)\, t_i$
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 $C = n_{i-1} - 1 + |2n_i - t_i| - |2n_{i-1} - t_{i-1}|$

$C = n_i + (2n_i - ti|) - |2(n_i + 1) - (3/2)\, t_i|$
$C = n_i + 2n_i + 2 - 2n_i - |2n_i + 2 - (3/2)(2*\, (n_i + 1))|$
$C = n_i + 2 - |2n_i + 2 - 3n_i - 3|$
$C = n_i + 2 - |-n_i - 1|$
$C = n_i + 2 - n_i - 1$
$C = 1$

## 2)

On fait pas que des insertions dans un tableau, on fait aussi des suppressions.

Dans tous les cas la stratégie est de multiplier la taille de tableau *2 quand il est plein et diviser par 2 lorsque le tableau est plein à 1/4.

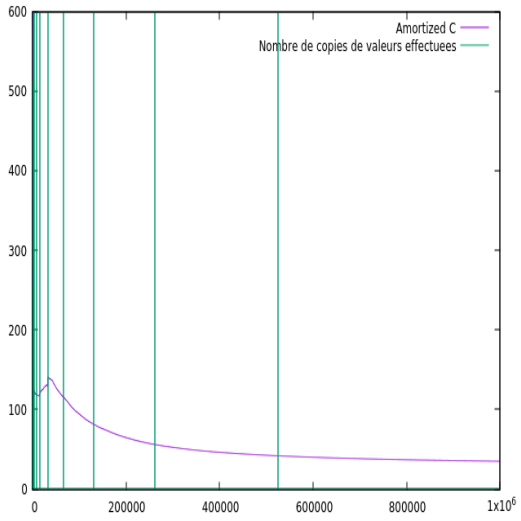
Dans tous les cas le coût amorti reste constant.

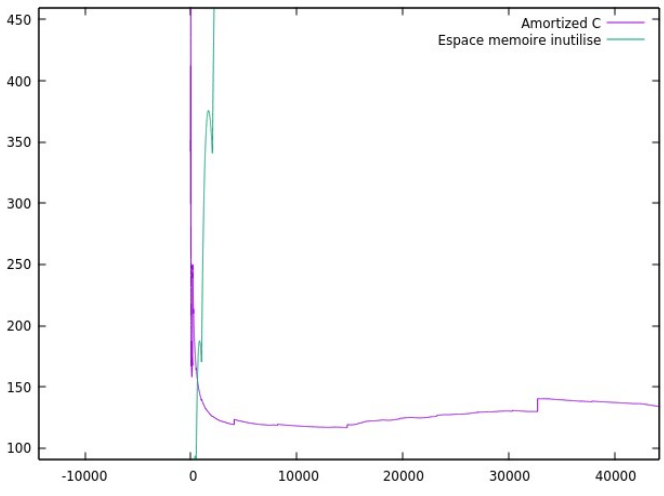
```
char memory_allocation;
float nbr=0;
 srand(time(NULL));
for(i = 0; i < 1000000 ; i++){
      nbr=rand();
  // Ajout d'un élément et mesure du temps pris par l'opération.
      if(nbr>=0.3 || a->size == 0){
        clock_gettime(clk_id, &before);

// Ajout d'un élément et mesure du temps pris par l'opération.
          memory_allocation = arraylist_append(a, i);
          clock_gettime(clk_id, &after);
  }
      else{
              clock_gettime(clk_id, &before);
              memory_allocation = arraylist_pop_back(a);
              clock_gettime(clk_id, &after);
  }
```

## 3) Utiliser les outils développés pour le TP1 afin d'enregistrer les coûts réels et amortis des opérations, ainsi que l'espace mémoire non-utilisé.

**P=0.3**



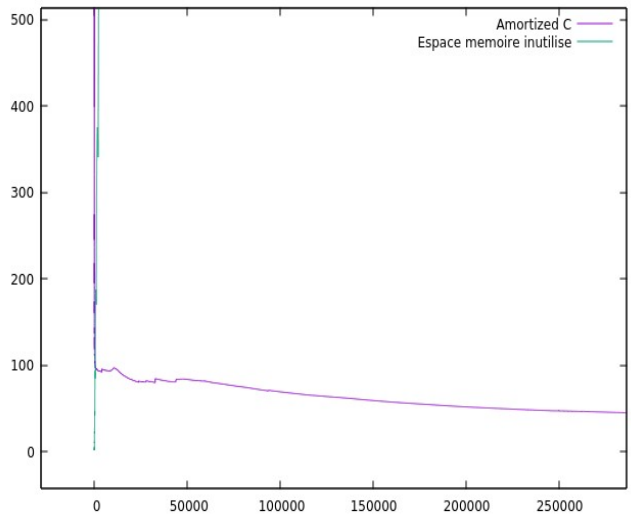

On remarque que le coût amorti augmente parce que on fait beaucoup de copies.
Si on fait que des insertions dans le tableau dès que ce dernier est plein on doit créer un nouveau plus grand et recopier tous les éléments dans ce nouveau , voila donc pourquoi on remarque que l'espace mémoire inutilisé décroît .

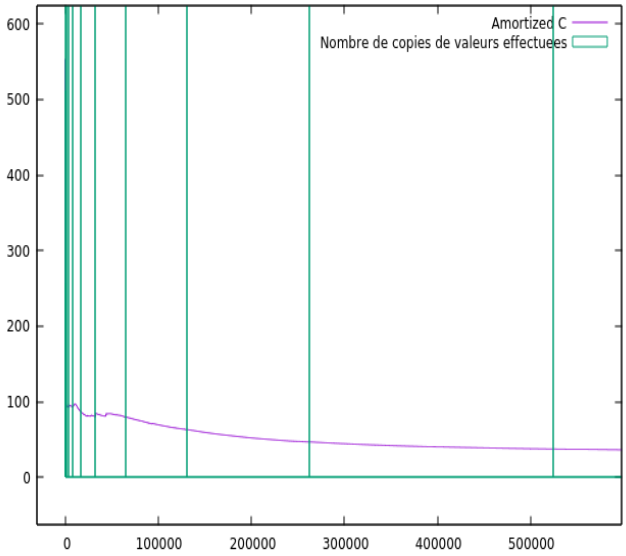
Le coût amorti commence à diminuer ainsi que le nombre de copies et ceci est due à la suppression des éléments dans le tableau donc on aura pas besoin d'augmenter la taille du tableau.
On remarque une augmentation de l'espace inutilisé à cause de la libération des cases de tableau après l'opération suppression.

En choisissant la probabilité de p=0.5  on fait autant d'insertion que de suppression dans le tableau.

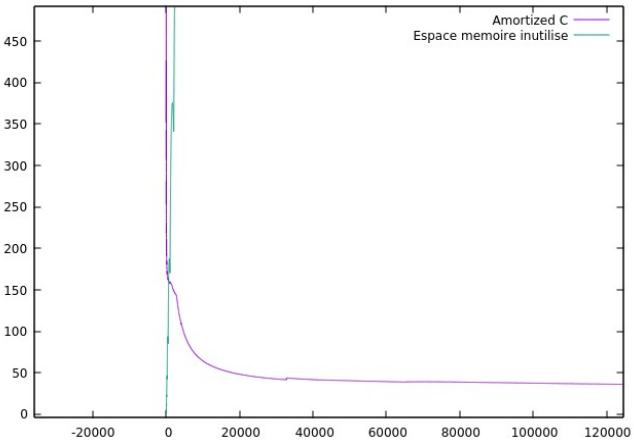
**4) Tester des valeurs différentes de p ∈ {0.1, 0.2, 0.3, . . .}. Utiliser gnuplot pour afficher les coûts amortis et l'espace mémoire non-utilisé pour chacune de ces expériences. Que pensez-vous de la relation entre p, le coût en temps et le gaspillage de mémoire ?**

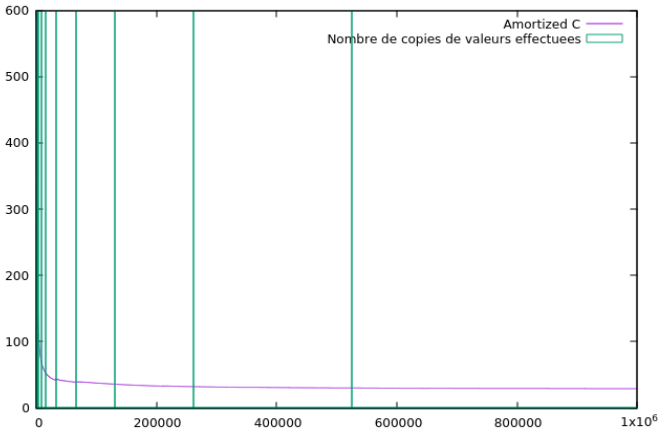
**P=0.1**



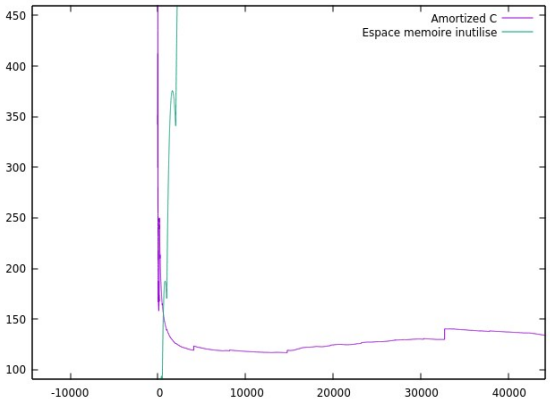



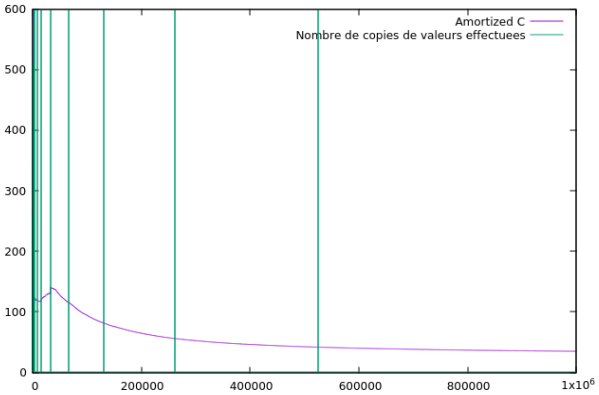

**p=0.2**



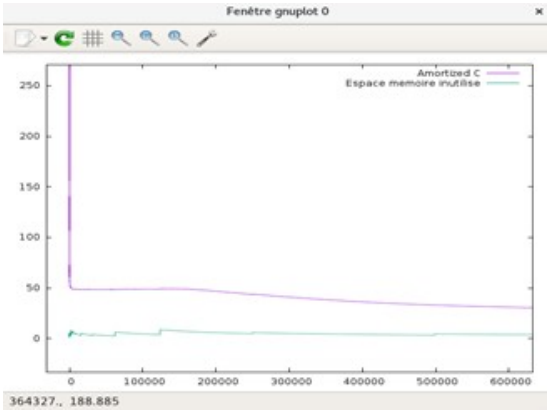



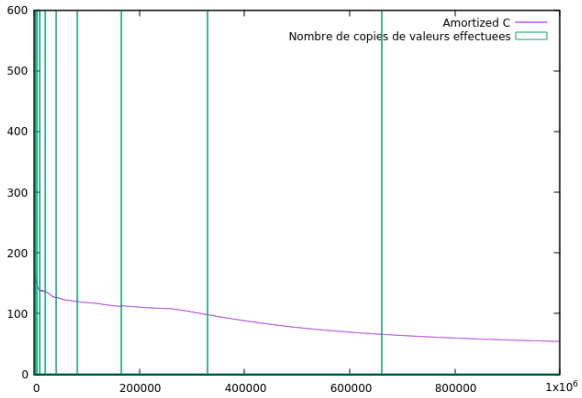
## p=0.3


Amortized C
Espace memoire inutilise
450
400
350
300
250
200
150
100
-10000
0
10000
20000
30000
40000

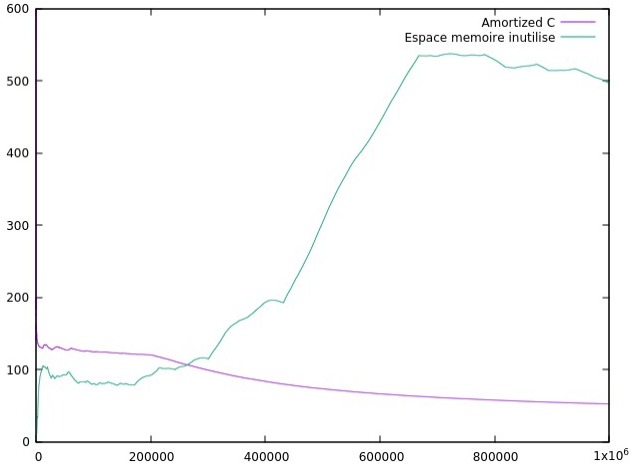

Amortized C
Nombre de copies de valeurs effectuees
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

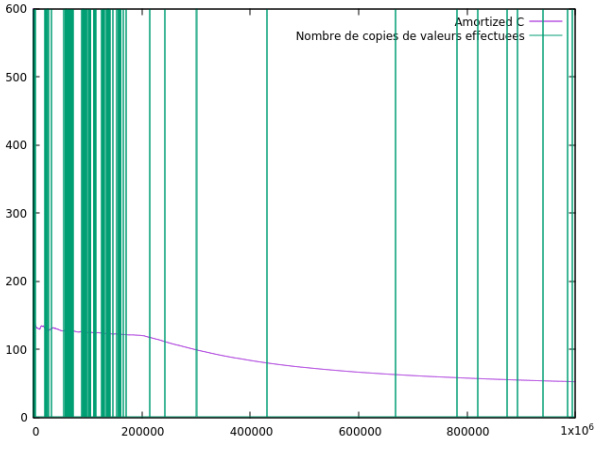
## p=0.4


Fenêtre gnuplot 0
Amortized C
Espace memoire inutilise
250
200
150
100
50
0
0
100000
200000
300000
400000
500000
600000
364327., 188.885

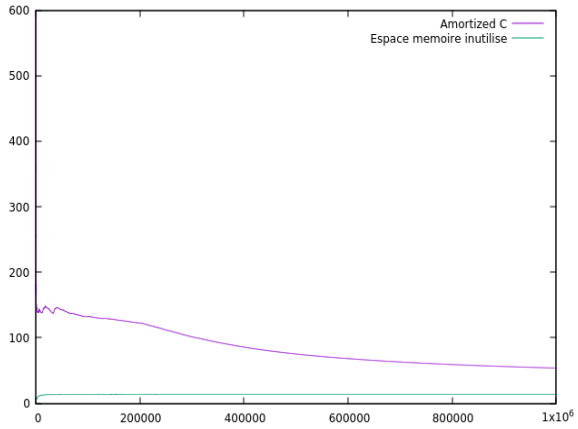

Amortized C
Nombre de copies de valeurs effectuees
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

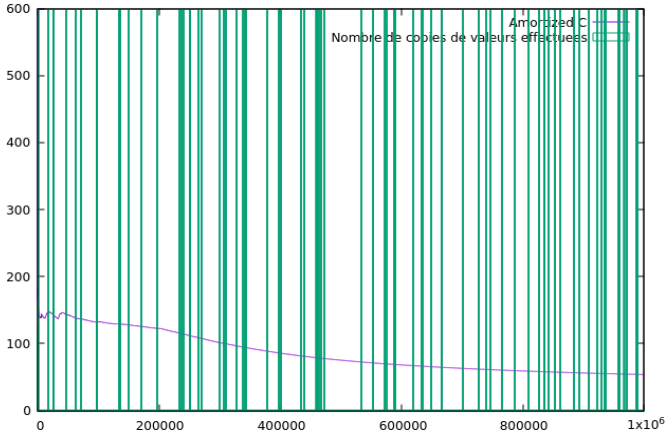
## p=0.5


Amortized C
Espace memoire inutilise
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

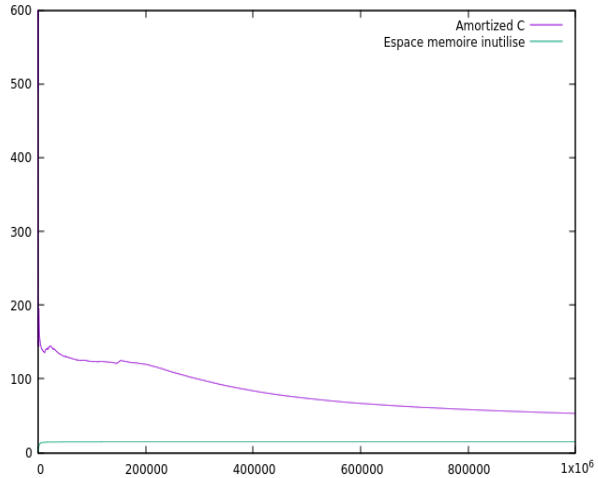

Amortized C
Nombre de copies de valeurs effectuees
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

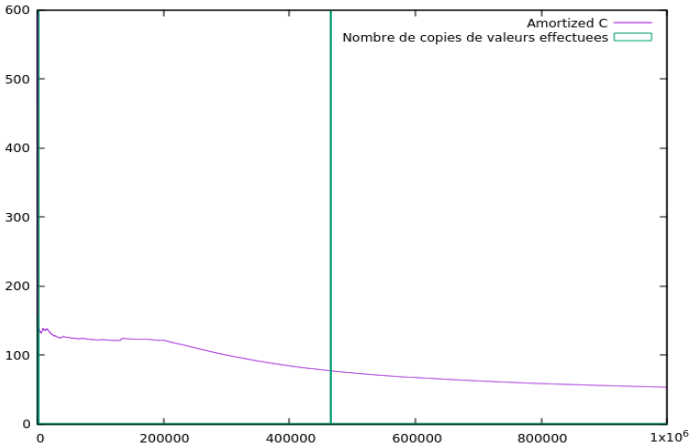
## p=0.6


Amortized C
Espace memoire inutilise
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

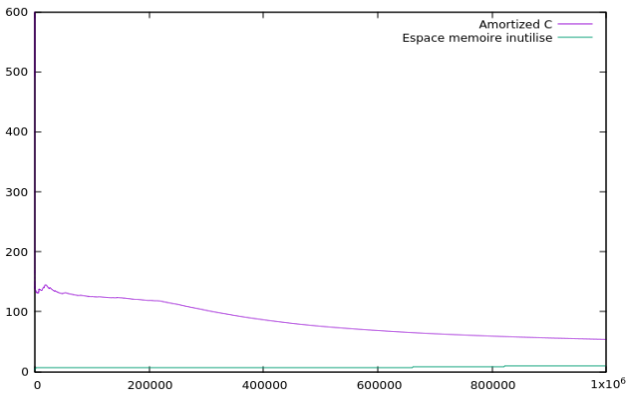

Amortized C
Nombre de copies de valeurs effectuees
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

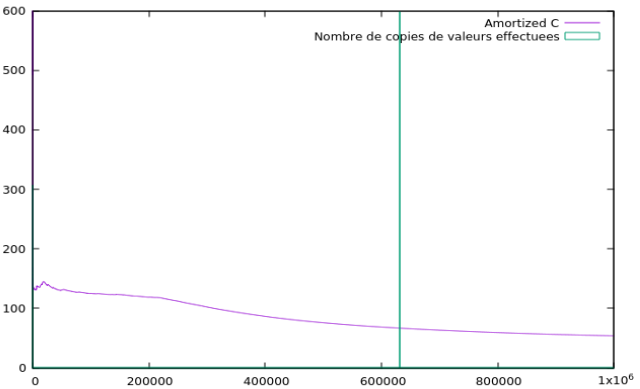
## p=0.7


Amortized C
Espace memoire inutilise
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

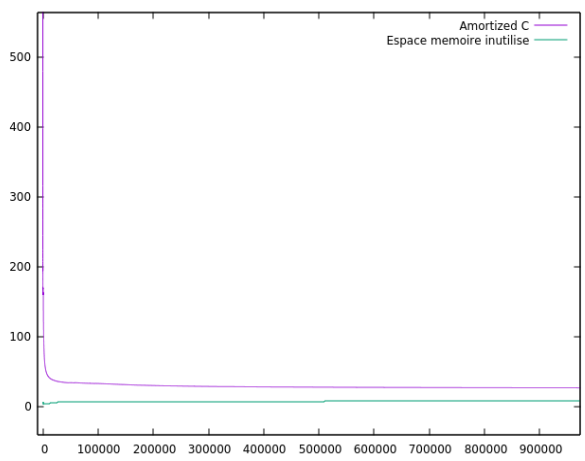

Amortized C
Nombre de copies de valeurs effectuees
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

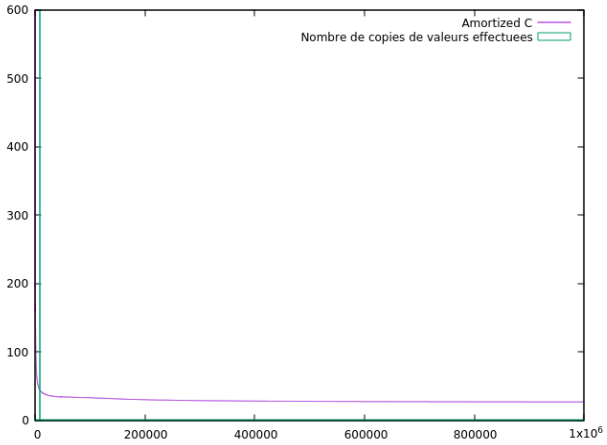
## p=0.8


Amortized C
Espace memoire inutilise
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6


Amortized C
Nombre de copies de valeurs effectuees
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

**p=0.9**




Quand la probabilité d'insertion est plus grande que la probabilité de la suppression, on remarque que le coût amorti augmente parce que on fait beaucoup de copies donc on insert beaucoup d'élément dans le tableau, dès que le tableau est plein on doit élargir la taille de ce dernière, par contre on voit que le gaspillage de mémoire décroît car on fait beaucoup plus d'insertions que de la suppression dons on n'aura pas beaucoup de cas vides donc on gagne en mémoire.

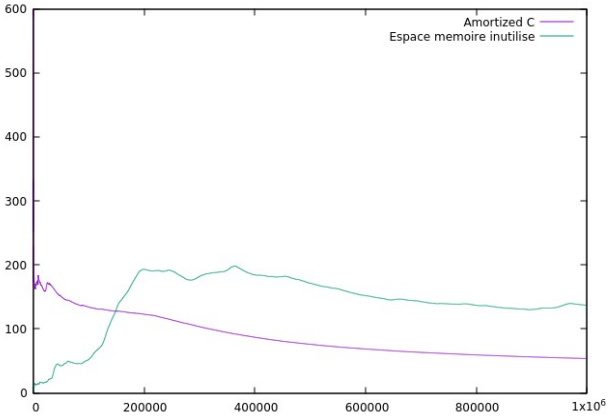
Quand la probabilité de la suppression est plus grande que la probabilité d'insertion on voit que le coût amorti diminue et on n'a pas besoin de faire beaucoup de copies, on remarque que le gaspillage de mémoire croit parce que on a fait que de la suppression des éléments dans le tableau par contre on a gagné en temps car on n'a pas besoin de faire beaucoup de copies.

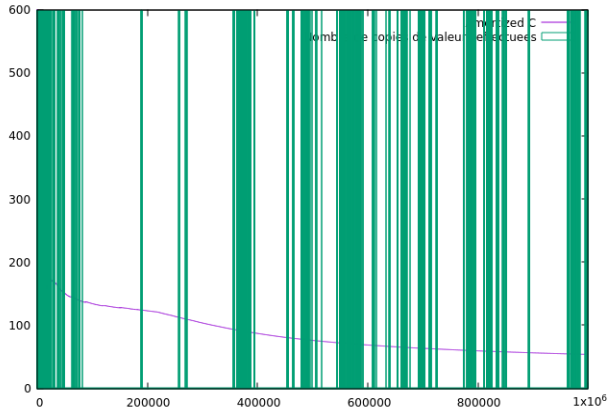
## 5) Choisir p = 0.5. Modifier la stratégie de redimensionnement de la table pour utiliser $taille_{i+1} = taille_i + \sqrt{taille_i}$ :

```
arraylist_enlarge_capacity(arraylist_t * a){
a->capacity *= 2;
a->capacity += sqrt(a->capacity);// la question 6 de tp1
capacity += sqrt(a->capacity); //la queston 5 de tp 2
>data = (int *) realloc(a->data, sizeof(int) * a->capacity);


char arraylist_do_we_need_to_reduce_capacity(arraylist_t * a){
  return ( a->size <= a->capacity/4 && a->size >4 )? TRUE: FALSE;
}

void arraylist_reduce_capacity(arraylist_t * a){
 // a->capacity /= 2;
 a->capacity -= sqrt(a->capacity); // la question 5 tp2
  a->data = (int *) realloc(a->data, sizeof(int) * a->capacity);
}
```



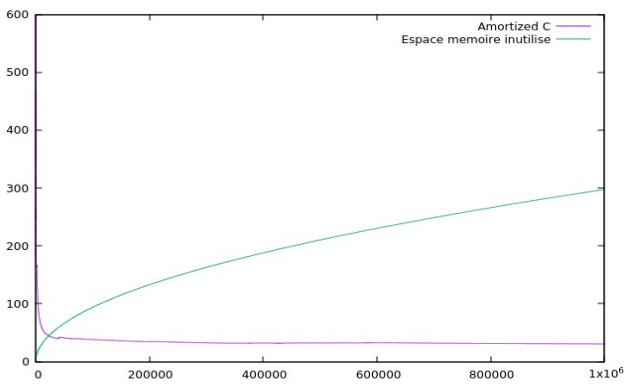



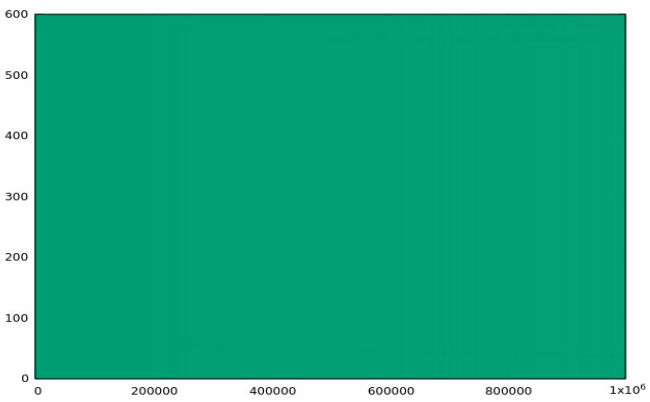

La contraction d'une table intervienne quand la table est rempli à 1/4 de sa taille, on divise sa taille sur deux pour libérer de l'espace mémoire non-utilisé.

Cette stratégie nous semble pertinente dans le cas ou le nombre de suppression est largement supérieur au nombre d'ajout, ce qui permet de gagner en mémoire. Par contre, elle est inefficace dans le cas contraire en effet, on fait énormément de copies.

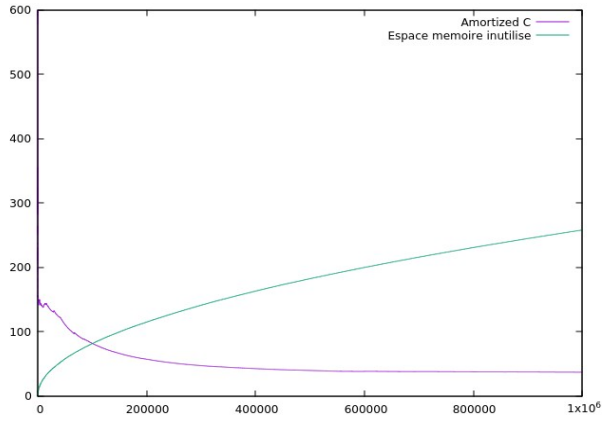
## 6) Tester à nouveau les différentes valeurs de p. Qu'en pensez-vous ? Quelle est la valeur de p pour laquelle cette stratégie semble mieux fonctionner ? Pourquoi ?

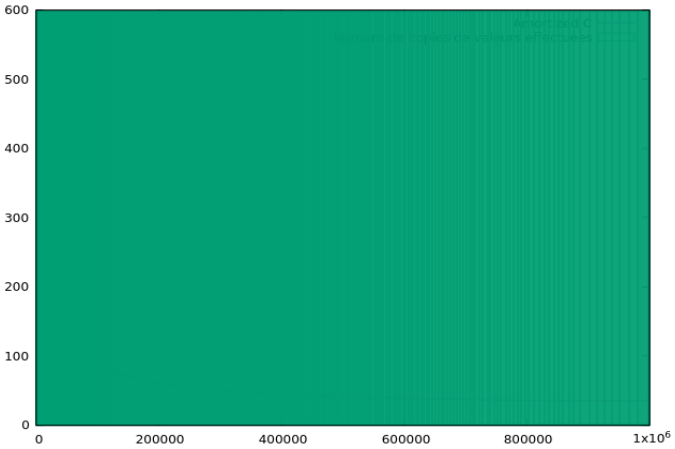
**P=0.1**



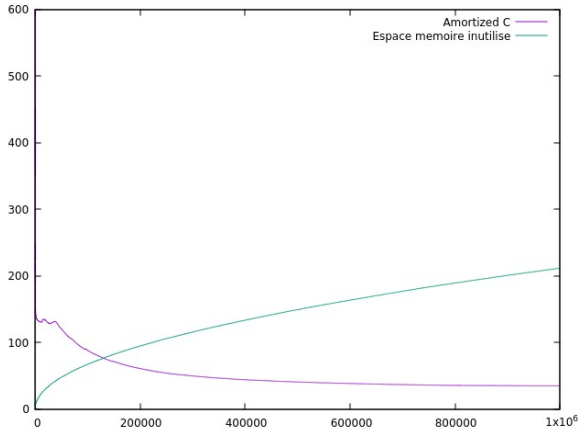



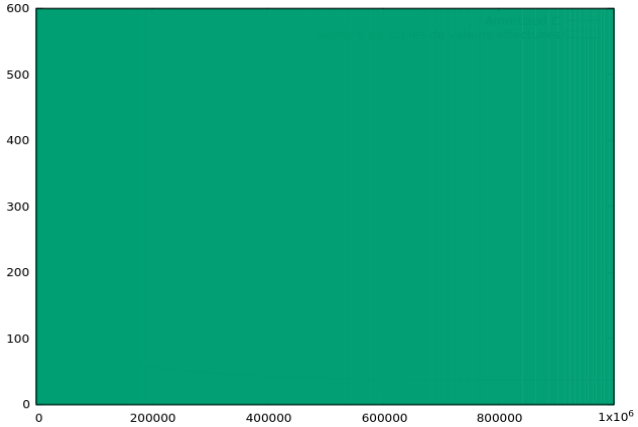
## p=0.2


Amortized C
Espace memoire inutilise
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

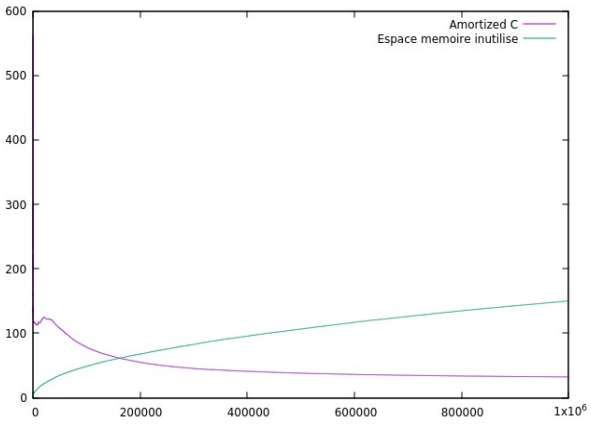

600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

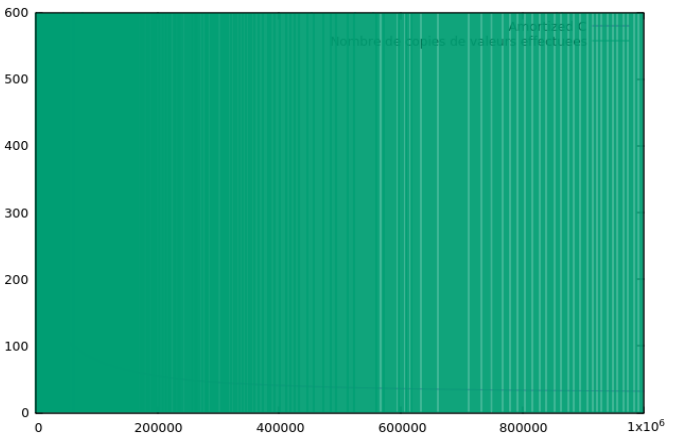
## p=0.3


Amortized C
Espace memoire inutilise
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

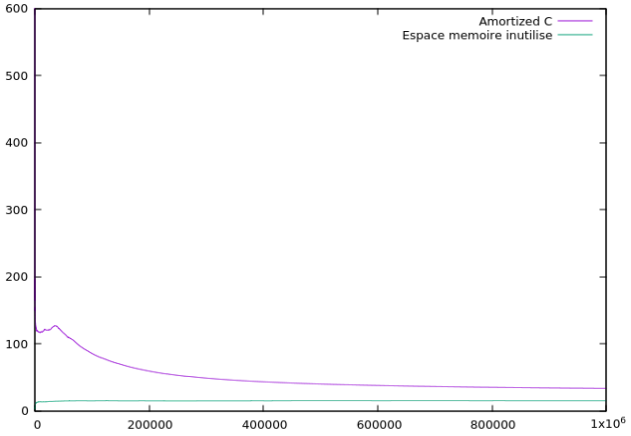

600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

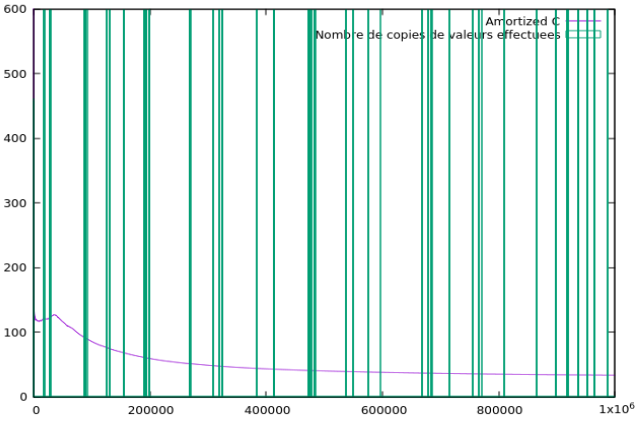
## p=0.4


Amortized C
Espace memoire inutilise
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

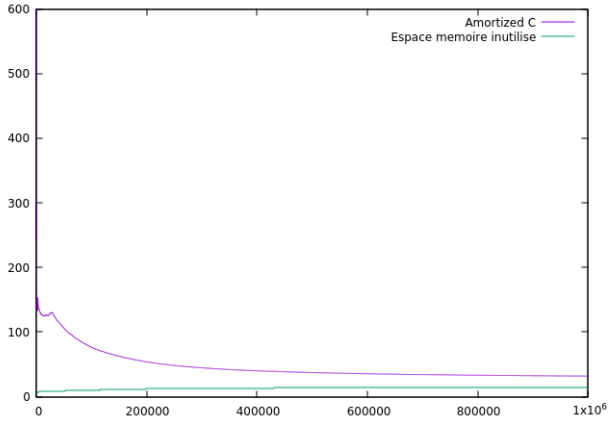

600
500
400
300
200
100
0
0
200000
400000
600000
800000
1x10^6

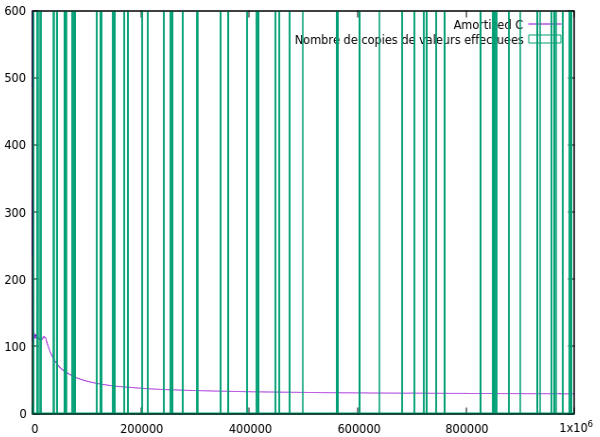
## p=0.6



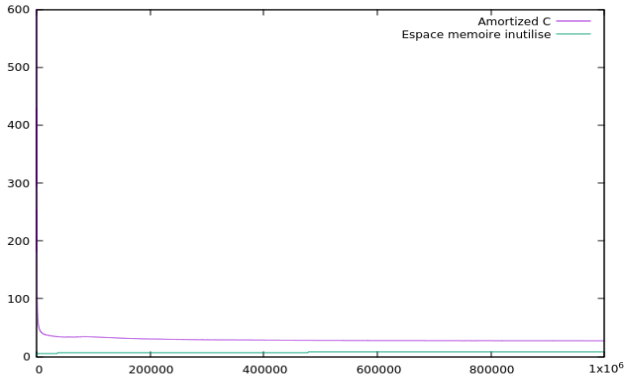



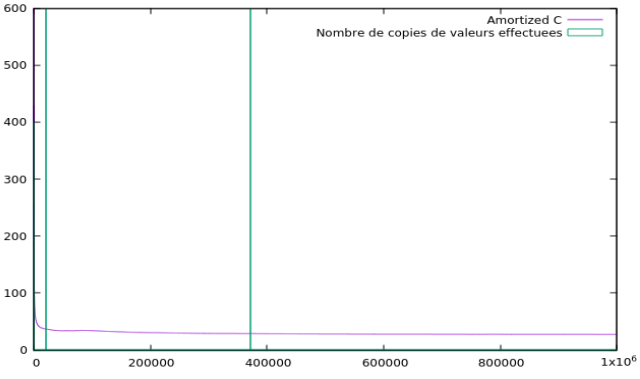

## p=0.7



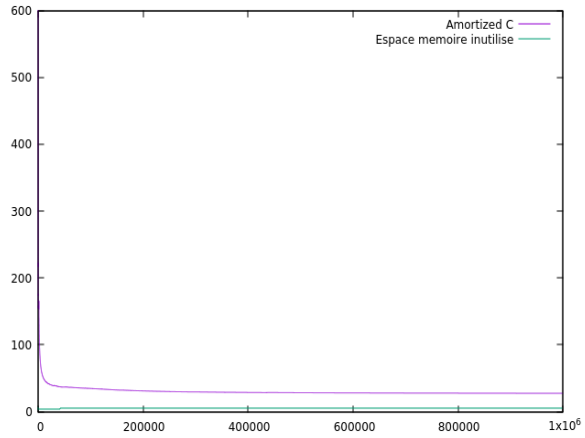



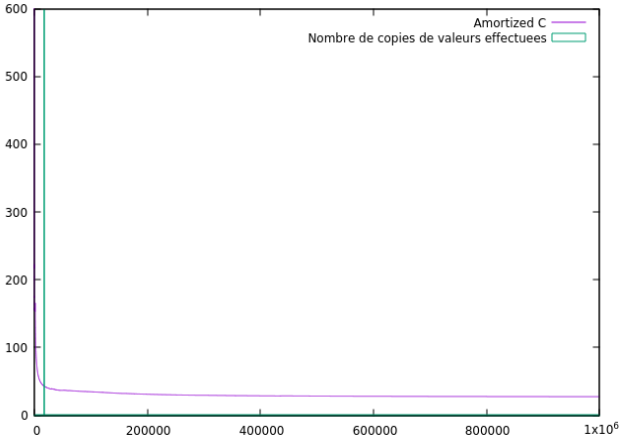

## p=0.8




## p=0.9




La contraction d’une table intervienne quand la table est remplie à 1/4 de sa taille, on divise sa taille sur deux pour libérer de l’espace mémoire non-utilisé.

Cette stratégie nous semble pertinente dans le cas où le nombre de suppression est largement supérieur au nombre d’ajout, ce qui permet de gagner en mémoire. Par contre, elle est inefficace dans le cas contraire en effet, on fait énormément de copies.