Antony Fourchault
MI1
Adrien Cavalieri

Rapport d'activité

TD/TP1

1) Quand nous doublons la taille du tableau, la réservation de mémoire se fait en temps constant.

$C_i = i - 1 + 1$, si $i - 1 = 2^K$ avec $K \geq 0$
$C_i = 1$, sinon

Déterminons maintenant la complexité de l'insertion de n éléments par la méthode du potentiel.

Juste avant l'extension, nous avons :
$\Phi(i) = n_i = t_i$, avec ni le nombre d'éléments et ti, la capacité du tableau

Juste après une extension, nous avons :
$\Phi(i) = 0$ car toute l'énergie a été dépensée pour copier les éléments

Ainsi, nous obtenons :
$\Phi(i) = 2n_i - t_i$

Calculons le coût amorti.

$\hat{C}_i = C_i + \Phi(i) - \Phi(i-1) = 3$

Conditions :
Avant extension, $\Phi(i) = 2n_i - t_i = (2 - 1)t_i = 0$
Après extension, $\Phi(i) = 2n_i - t_i = t_i - t_i = 0$

Or, si nous prenons $\Phi(i) = \alpha n_i - t_i$, le résultat n'est pas toujours positif donc la première condition n'est pas satisfaite.

Ainsi, avant extension, nous avons :
$\Phi(i) = \alpha n_i - t_i = 0$ car $\alpha n_i = t_i$
Puis, après extension :
$\Phi(i) = \alpha n_i - t_i = (\alpha - 1)t_i$ avec $\Phi(i)$ représentant l'énergie nécessaire pour copier

Si $\alpha > 2$, $\Phi(i)$ est trop grand (trop d'énergie).
Si $\alpha < 2$, $\Phi(i)$ est trop petit.

Donc, cette fonction marche bien uniquement avec $\alpha = 2$

Maintenant, si on prend $\Phi(i) = xn_i - yt_i$, il y a 2 conditions :
Avant extension, $xn_i - yt_i = 0$ soit $xn_i - y\alpha n_i = 0$ donc $x = y\alpha$
Après extension, $xn_i - yt_i = n_i$ soit $xn_i - yn_i = n_i$ donc $x = 1 + y$

On a donc le 2 équations suivantes :
$x = y\alpha$
$x = 1 + y$

Alors, $y + 1 = y\alpha$ soit $y = \frac{1}{\alpha - 1} \Rightarrow x = \frac{\alpha}{\alpha - 1}$

Ainsi, $\Phi(i) = \frac{\alpha}{\alpha - 1} n_i - \frac{1}{\alpha - 1} t_i$

Donc, avec $\alpha = 2$, on a :
$\Phi(i) = 2n_i - t_i$

2) Nous rappelons que $\hat{C}_i = C_i + \Phi(i) - \Phi(i-1)$

Dans le cas sans extension :

$\hat{C}_i = 1 + (\frac{\alpha}{\alpha - 1} n_i - \frac{1}{\alpha - 1} t_i) - (\frac{\alpha}{\alpha - 1} n_{i-1} - \frac{1}{\alpha - 1} t_{i-1})$
Ainsi,
$\hat{C}_i = 1 + (\frac{\alpha}{\alpha - 1}) = \Theta(\alpha)$

Dans le cas avec extension :
$\hat{C}_i = \frac{\alpha}{\alpha - 1} = \Theta(\alpha)$
Alors, si on a n éléments de coût $\Theta(\alpha)$, nous multiplions n par $\Theta(\alpha)$ et nous obtenons $\Theta(n)$.

3) a) L'ajout de données au tableau, l'extension de celui-ci (de complexité n du nombre d'éléments du tableau) et la sauvegarde de

données de l'expérience semblent prendre le plus de temps.

b) Pour le code Java, le coût amorti augmente légèrement autour des 100 000 µs, avant 200 000 µs, 400 000 µs et 800 000 µs. Quant au code C, le coût amorti augmente légèrement vers les 19 000 µs et 50 000 µs. En outre, pour le code C++, le coût amorti augmente aux alentours des 50 000 µs, 100 000 µs, 200 000 µs, 400 000 µs et juste avant 800 000 µs. Enfin, le coût amorti du code Python augmente vers les 19 000 µs.
Ces augmentations de coût correspondent plus ou moins aux moments où de la mémoire est allouée aux programmes, c'est-à-dire lorsque l'on étend le tableau par exemple (voir l'image "test1-normal-temps-langages.png" dans le répertoire "Test 1")

c) Nous prenons le cas du code Java. Pour celui-ci, le nombre de copies augmente alors que le coût amorti diminue. On remarque aussi que les augmentations de coût aux alentours de 200 000 µs, 400 000 µs et 800 000 µs correspondent aux copies de valeurs. Ainsi, quand les copies se font, l'espace mémoire gaspillé augmente.
Le coût amorti reste constant (voir les images relative au code Java dans le répertoire "Test 1").

d) Les augmentations de coût ont toujours lieu à peu près aux mêmes moments, correspondant ainsi aux copies de valeurs (voir les images dans les répertoires "Test 2" et "Test 3")

e) Le C et le C++ sont plus rapides car ils sont compilés en langage machine. En revanche, le Java est compilé en bytecode (pour être exécuté dans la machine virtuelle Java) et le Python est un langage interprété.
De plus, les autres programmes s'exécutant sur la machine, tels que les navigateurs web, peuvent influer sur la rapidité des différents langages.

f) Les 4 programmes ont besoin de moins en moins de mémoire au fil du temps. Ainsi, il y a de plus en plus d'espace mémoire inutilisé : celui-ci est gâché. Cela pourrait poser problème si un autre processus, s'exécutant en même temps qu'un des programmes, avait besoin de mémoire en quantité importante. En effet, les programmes effectuant des copies gâcheraient la mémoire mais celle-ci serait toujours réservée et indisponible pour les autre processus.

4) Nous choisissons de modifier le code Java. Pour l'instant, nous réallouons de la mémoire quand le tableau est au ¾ plein. Notons que la fonction "do_we_need_to_enlarge_capacity" se trouve dans le fichier ArrayListProxy.java.

```
private boolean do_we_need_to_enlarge_capacity() {
```

```
        return data.size() >= (capacity * 3) / 4;
}
```

Changeons ce comportement.

```
private boolean do_we_need_to_enlarge_capacity() {
        return data.size() >= capacity;
}
```

Maintenant, nous réallouons de la mémoire uniquement quand le tableau est plein. Ainsi, une fois que nous avons effectué ce changement, nous pouvons remarquer que le programme utilise plus d'espace mémoire (il y a donc moins d'espace mémoire inutilisé) que lors du 3ème test, effectué à la question précédente. A certains moments, le programme utilise toute la mémoire qui lui a été allouée (l'espace mémoire non-utilisé tombe alors à 0). Nous pouvons examiner les résultats sous la forme de graphiques, dans le sous-répertoire "do_we_need_to_enlarge_capacity-changement-facteur" du répertoire "Conditions".

5) Nous revenons sur le changement que nous avons effectué dans la fonction "do_we_need_to_enlarge_capacity" et utilisons alors le code original de cette fonction. Nous allons maintenant faire varier le facteur multiplicatif dans la fonction "enlarge_capacity". A noter que cette fonction se trouve dans le même fichier source que la fonction "do_we_need_to_enlarge_capacity".

```
private void enlarge_capacity(){
    capacity *= 2;
    data.ensureCapacity(capacity);
}
```

Nous faisons d'abord un premier test en passant le facteur à 4.

```
private void enlarge_capacity(){
    capacity *= 4;
    data.ensureCapacity(capacity);
}
```

Puis, un 2ème en le passant, cette fois, à 8.

```
private void enlarge_capacity(){
    capacity *= 8;
    data.ensureCapacity(capacity);
```

```
}
```

Ainsi, en faisant varier le facteur multiplicatif comme ceci, nous pouvons remarquer que l'espace inutilisé double si le facteur est doublé : il passe de 2ème dans le 3ème test de la question 3)f) à 4 puis à 8 alors que l'espace inutilisé maximal passe d'environ $1.3x10^6$ à $3.4x10^6$ puis à $7.5x10^6$. En outre, le coût amorti en temps ne change que très peu mais est lié à n. Ainsi, plus la valeur d'alpha est proche de 1, plus l'espace gaspillé sera réduit. Et plus elle s'éloigne de 1, plus l'espace gaspillé sera important mais en, contrepartie, l'intervalle entre chaque copie sera plus grand. Nous pouvons examiner les résultats sous la forme de graphiques, dans le sous-répertoire "enlarge_capacity-changement-facteur" du répertoire "Conditions".

6) Nous revenons sur le changement que nous avons effectué dans la fonction "enlarge_capacity" et utilisons alors le code original de cette fonction. Nous faisons varier la capacité n vers une capacité n + sqrt(n).
Nous importons alors la bibliothèque Math dans le fichier source ArrayListProxy.java.

```
import java.lang.Math;
```

Puis nous modifions à nouveau le code de la fonction "enlarge_capacity".

```
private void enlarge_capacity(){
   capacity += (int)Math.sqrt(capacity);
   data.ensureCapacity(capacity);
}
```

Nous pouvons examiner les résultats sous la forme de graphiques, dans le sous-répertoire "enlarge_capacity-changement-capacite" du répertoire "Conditions". Nous remarquons alors que l'espace mémoire inutilisé augmente linéairement au fil de l'exécution. De plus, le coût amorti augmente subitement vers les 500 000 µs. Enfin, les copies sont de plus en plus fréquente et leur nombre augmente également linéairement. Mais, l'espace mémoire inutilisé est, sur 1 million d'entrée, est 4 fois moins important que lors d'un test avec le code de base (voir les résultats dans les répertoires "Test 1", "Test 2" et "Test 3".

Ainsi, nous avons :
$t_i = t_{i-1} + sqrt(t_{i-1})$ au lieu de
$t_i = \alpha t_{i-1}$, avec α constant et $\alpha > 1$

Alors,
$t_{i-1} + sqrt(t_{i-1}) = \alpha t_{i-1}$

$\alpha = (t_{i-1} + \text{sqrt}(t_{i-1})) / t_{i-1} = 1 + (1 / \text{sqrt}(t_i))$, où α n'est pas constant

Nous pouvons alors nous demander quel est le coût amorti de insérer(T,e) sachant que :

$\Phi(i) = \frac{\alpha}{\alpha - 1} n_i - \frac{1}{\alpha - 1} t_i$

$\hat{C}_i = 1 + (\frac{\alpha}{\alpha - 1})$, dans le cas d'une extension du tableau

$\hat{C}_i = \frac{\alpha}{\alpha - 1}$, dans le cas d'absence d'extension

Alors, pour $\alpha = 1 + (1 / \text{sqrt}(t_i))$, on a :

$\hat{C}_i = \text{sqrt}(t_i) + 2$, dans le cas d'une extension

$\hat{C}_i = \text{sqrt}(t_i) + 1$, dans le cas d'absence d'extension

Conclusion

Il est intéressant de choisir un α petit si on dispose d'une machine avec une petite mémoire car il y aura moins d'espace mémoire inutilisé lors de l'exécution du programme.

TD/TP2

1) Quand $\alpha_i = n_i / t_i = 1$, on étendait le tableau (on doublait sa taille). Si $\alpha_i \leq 1 / 4$, on contractait le tableau (sa taille était divisée par 2). α_i est donc le coefficient de remplissage du tableau.

Maintenant, nouvelle stratégie : quand le tableau est rempli seulement au tiers, on le divise par 2 / 3.

Ainsi, $\alpha_i = n_i / t_i \leq 1 / 3$ et $\Phi(i) = 2n_i - t_i$

Calculons $\hat{C}_i$ de l'opération supprimer(T, e)

$\hat{C}_i = C_i + \Phi(i) - \Phi(i-1)$

Notons que $C_i = 1$ s'il n'y a pas de contraction. De plus, nous pouvons alors distinguer 2 cas.

Cas 1 : pas de contraction

$\hat{C}_i = 1 + (2n_i - t_i) - (2n_{i-1} - t_{i-1})$

On sait que $n_{i-1} = n_{i+1}$ car on a supprimé un élément et que $t_{i-1} = t_i$ car il n'y a pas eu de contraction du tableau. Alors,
$\alpha_i = n_i / t_i \geq 1 / 3$

$\alpha_{i-1} = n_{i-1} / t_{i-1} > 1 / 3 \Leftrightarrow t_i > 3(n_i + 1) \Leftrightarrow t_i > 3n_i + 3 \Leftrightarrow 3n_i + 3 - t_i < 0$

donc,

$\alpha_{i-1} = n_{i-1} / t_{i-1} > 1 / 3 \Leftrightarrow 3n_i + t_i < 0$

On en déduit alors,

$\hat{C}_i = 1 + (t_i - 2n_i) - (2(n_i + 3) - t_i) = 1 + t_i - 2n_i - (t_i - 2(n_i + 1)) = 3$

Cas 2 : contraction

$\hat{C}_i = n_{i-1} - 1 + (2n_i - t_i) - (2n_{i-1} - t_{i-1})$, avec $n_{i-1} - 1 = n_i$

On sait que $n_{i-1} = n_{i+1}$ et que $t_{i-1} = \frac{3}{2} t_i$. Alors,
$\alpha_i = n_i / t_i \leq 1 / 3 \Leftrightarrow t_{i-1} = 3n_{i-1} \Leftrightarrow \frac{3}{2} t_i = 3n_{i+1} \Leftrightarrow t_i = 2n_{i+1}$

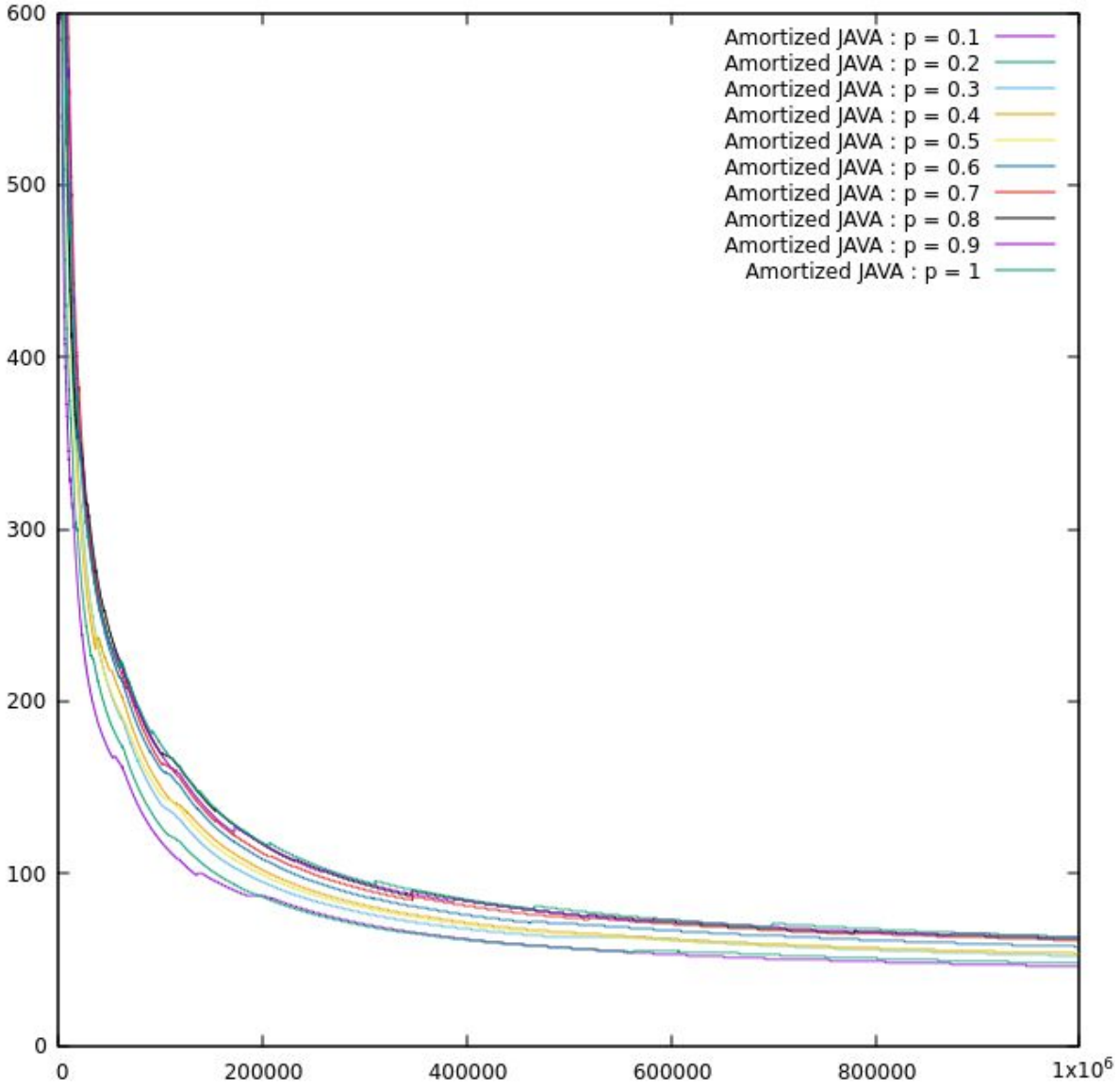
On en déduit alors,

$\hat{C}_i = n_i + (t_i - 2n_i) - (2n_{i+1} - \frac{3}{2} t_i)$
$\hat{C}_i = n_i + 2n_i + 2 - 2n_i - (2n_i + 2 - \frac{3}{2} * 2(n_i + 3))$
$\hat{C}_i = n_i + 2 - (2n_i + 2 - 3n_i - 3)$
$\hat{C}_i = n_i + 2 - (-n_i - 1) = n_i + 2 - (n_i + 1) = n_i + 2 - n_{i-1} = 1$

2) Nous modifions le code du Main.java pour effectuer un mélange de n opérations d'insertions et de suppressions (voir le fichier source Main.java).

3) Nous utilisons les fichiers gnuplots généré par le code Java pour pouvoir visualiser les coûts et l'espace mémoire non-utilisé au cours de l'exécution du programme.

4)



Coûts amortis des différentes valeurs de p (voir fichier “quest4-tous-temps-java.png” dans le sous-répertoire “Question 4” du répertoire “plots” du TP2)

Plus on augmente la valeur de p, plus le pic d’espace mémoire inutilisé est important. De plus, de p = 0.1 à p = 0.4, l’espace mémoire inutilisé varie très fréquemment. Pour p = 0.5, l’espace mémoire inutilisé, varie bien moins fréquemment mais son pic est bien plus important (550 contre 20 à p = 0.4). Puis, de p = 0.6 à p = 1, cet espace croît très lentement au fil de l’exécution du programme. En outre, plus on augmente p, moins le coût amorti baisse au cours de l’exécution du programme (voir les sous-répertoires “Question 4” dans le répertoire “plots” du TP2 pour les graphiques et les fichiers plots correspondants).

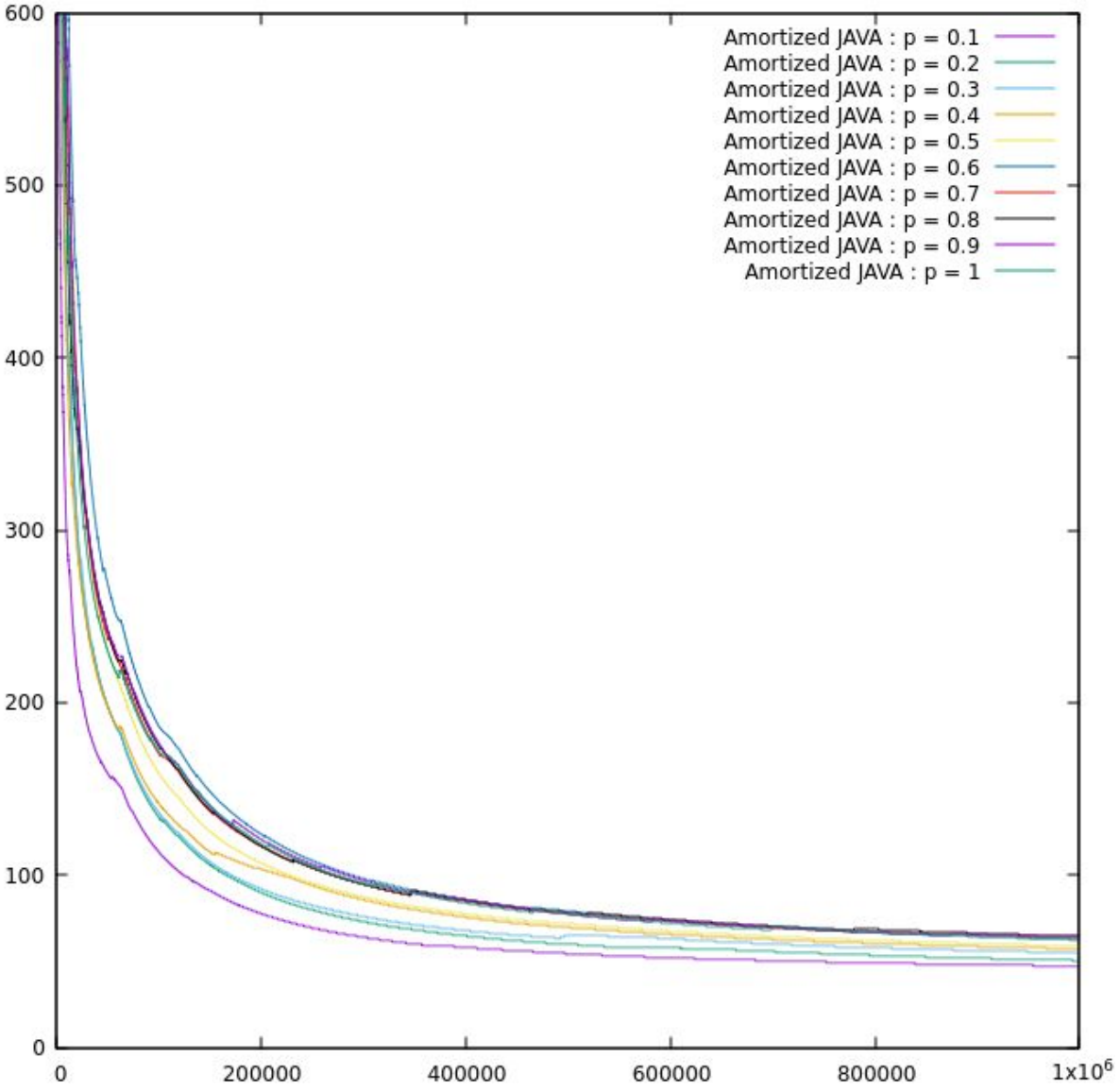
5) Nous modifions les méthodes reduce_capacity ainsi que do_we_need_to_reduce_capacity présentes dans le fichier ArrayListProxy.java, dans le répertoire “Java” du TP2:

```
private boolean do_we_need_to_reduce_capacity(){
       //return data.size() <= capacity/4 && data.size() >4;
       return data.size() == capacity - Math.sqrt(capacity);
}

void reduce_capacity(){
    //capacity /= 2;
   capacity -= Math.sqrt(capacity);
    data.ensureCapacity( capacity );
}
```

Il faut donc contracter la table quand la taille est égale à capacité - sqrt(capacité) : cette stratégie est alors optimale.

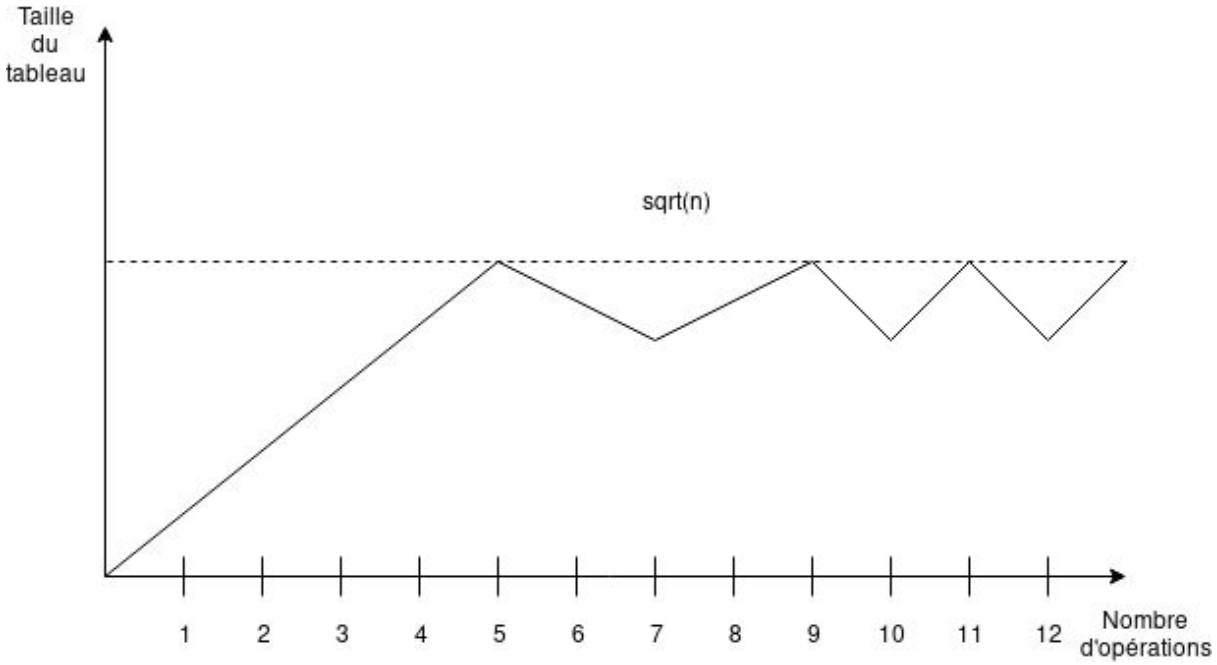
6)



Coûts amortis des différentes valeurs de p (voir fichier “quest6-tous-temps-java.png” dans le sous-répertoire “Question 6” du répertoire “plots” du TP2)

Cette stratégie est optimale quand p = 0.1. En effet, même si l’espace inutilisé varie très fréquemment, celui-ci reste peu important (6 au maximum) comparé aux autres valeurs de p. De plus, en terme de coût amorti, celui-ci est peu important et baisse très vite au cours de l’exécution du programme pour p = 0.1 (voir les sous-répertoires “Question 6” dans le répertoire “plots” du TP2 pour les graphiques et les fichiers plots correspondants)..

Nous pouvons faire un croquis de l’évolution des valeurs :



Conclusion

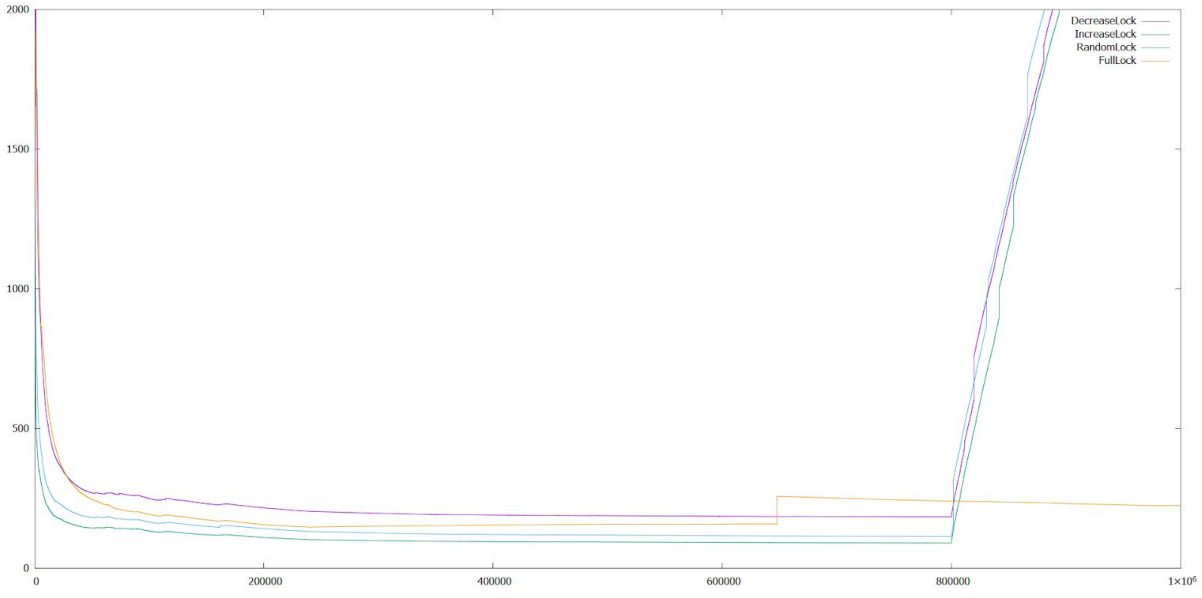
Un tableau dynamique est une structure de données utile dans le cas où l'on ignore la capacité que l'on va utiliser ou qu'elle est compliquée à calculer. Un tel tableau est également adapté dans les cas où la capacité est susceptible de changer.

TD/TP3

Tas binaire

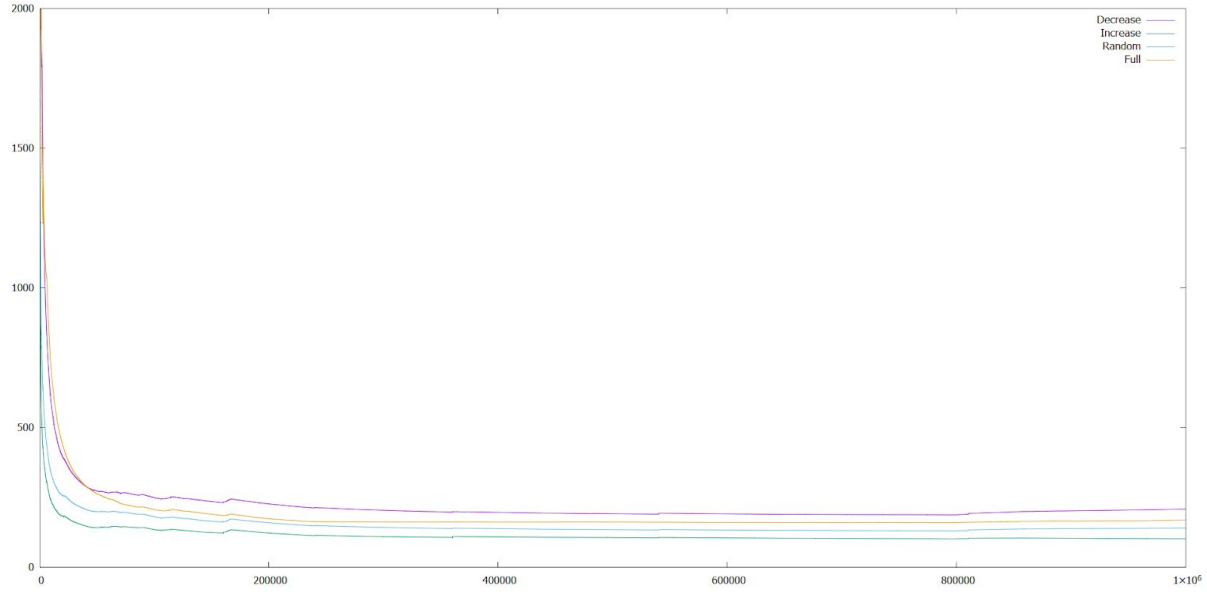
1) Voir le fichier BinaryHeapLock.java

2) On obtient alors le schéma suivant pour les temps d'exécution : (voir fichier binarylock.png) :



On voit donc ici que lorsqu'on insère des valeurs de manière décroissantes, le temps est plus long, ce qui est normal puisque qu'à chaque nouvelle valeur ajouté celle-ci doit remonter tout l'arbre pour atteindre la racine. On peut aussi voir que toutes les courbes se stabilisent vers 20 000 - 50 000 ajouts de valeurs, ce qui est logique avec la complexité des fonctions insertion et extraction du minimum qui sont de ln(n). Cette complexité est dû au fait que plus l'arbre est profond, plus il y a de noeud de même niveaux, par exemple les 32768 noeuds qui sont au niveau de profondeur 15 prennent tous le même temps pour remonter jusqu'à la racine dans le cas d'un insertion décroissante.
Quant à l'insertion croissante et random, elle presque identique puisque le temps de l'insertion croissante correspond juste au temps nécessaire pour ajouter une valeur à l'arbre, et même chose pour random avec un chance de plus en plus faible à devoir remonter plusieur branche de l'arbre, là où l'insertion décroissant devait toutes les remonter.
Pour le cas de la courbe d'insertion et d'extraction, le pic vers 650 000 peut s'expliquer par une extraction qui à nécessité plusieur réarrangement de l'arbre
Enfin nous pouvons voir une augmentation commune à l'insertion croissante, décroissante et random à partir de la 800 000 valeurs, ce qui correspond au moment où l'on a atteint la capacité maximale de l'arbre, dans le cas de "full" (extraction et insertion), cette capacité maximale à peu de chance d'être atteinte puisque l'on fait 50% d'insertion et 50% d'extraction.

3) On obtient le schéma suivant dans le cas où l'on retire la capacité maximal à notre arbre binaire:

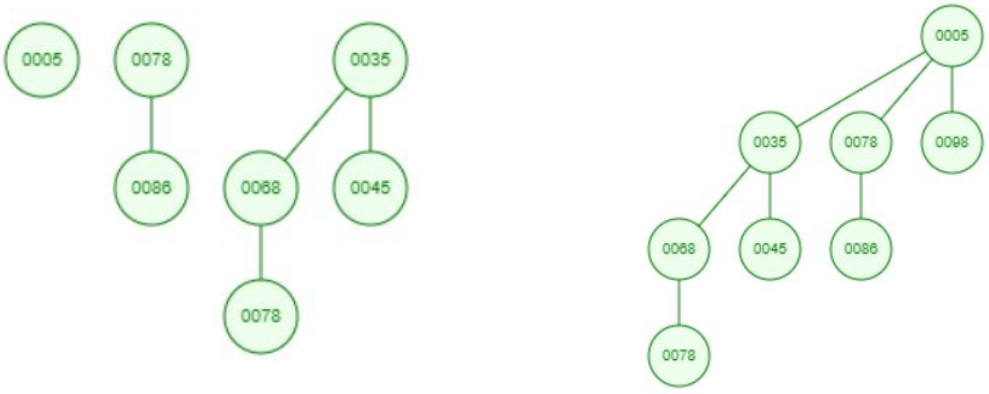


Ainsi on peut voir aucune différence avec capacité maximale ou non, voir une infime dans les premières insertions où ça doit etre un probleme avec Java qui doit fixer lui même une taille à l'arraylist et qui doit l'étendre ce qui rallonge légèrement le temps

des opérations. La non-présence de changement entre capacité max ou non s'explique par la complexité même des fonctions qui ne dépendent pas de la taille du tableau.

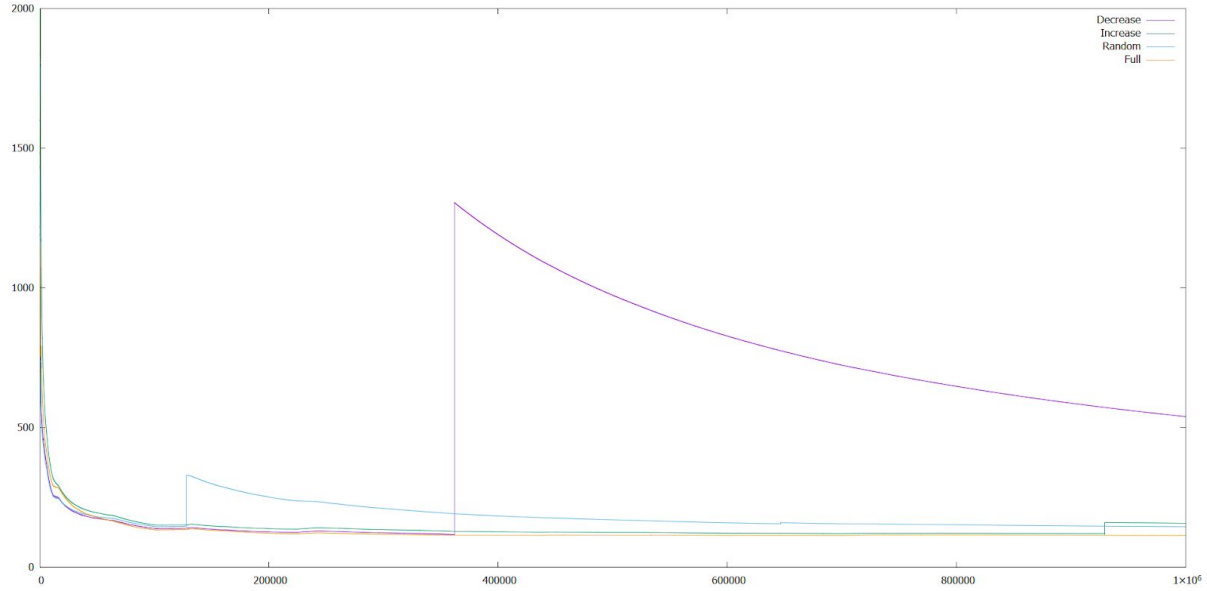
Tas binomial

1) Pour chaque chiffre binaire x, avec x > 2, sa complexité amortie d'insertion est de log2(x)+1, en effet chaque insertion amenant à un total de noeud équivalent à une chiffre binaire, tous les arbres de la forêt vont se rassembler afin de former un unique grand arbre. Par exemple cet arbre binomial composé de 7 noeuds passant à 8 noeuds après insertion de 98, il faut insérer 98 à l'arbre, fusionner 5 et 98, fusionner 5 et 78, puis fusionner 5 à 35 soit un total de log2(8)+1 = 4 opérations.



2) Voir le fichier BinaryHeap.java

3) Voici le schéma suivant sur le temps avec les mêmes expériences que le tas binomial (voir fichier binomial.png)



On peut y voir certains pics notamment dans les courbes d'insertion décroissante et random qui peut s'expliquer par plusieurs unions qui ont augmenté le temps d'exécution.
Encore la courbe de d'insertion croissante est la plus basse ce qu'il s'explique par le fait que les arbres de la forêt sont déjà triés dans l'ordre, ce qui facilité la tâche de l'exécution.

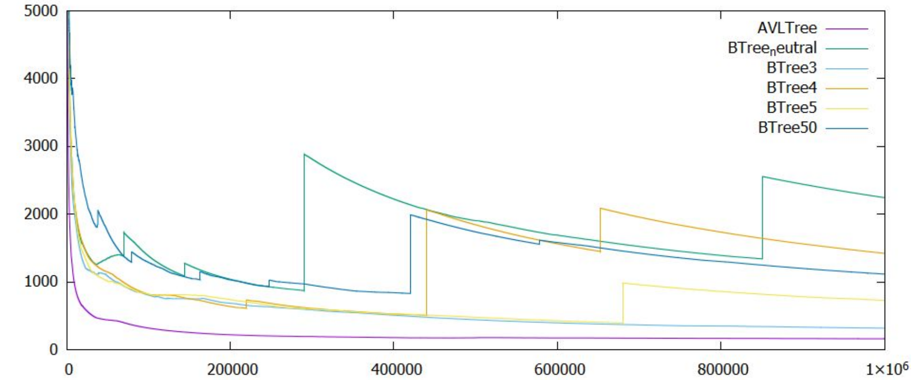
Conclusion : selon vos besoins le tas binomial est préférable aux tas binaires et inversement, par exemple dans le cas où l'on sait que l'on va avoir que des valeurs entrée dans l'ordre décroissant ou de trouver le minimum, il est préférable d'utiliser les tas binaires qui ne comportent pas de pics de temps d'exécution et qui ont une complexité de 1 pour la recherche de minimum, si vous allez faire plusieurs manipulations d'extractions et d'insertions le tas binomial est légèrement plus performant. Enfin si vous n'avez qu'à insérer des variables dans l'ordre croissant, les 2 sont équivalents.

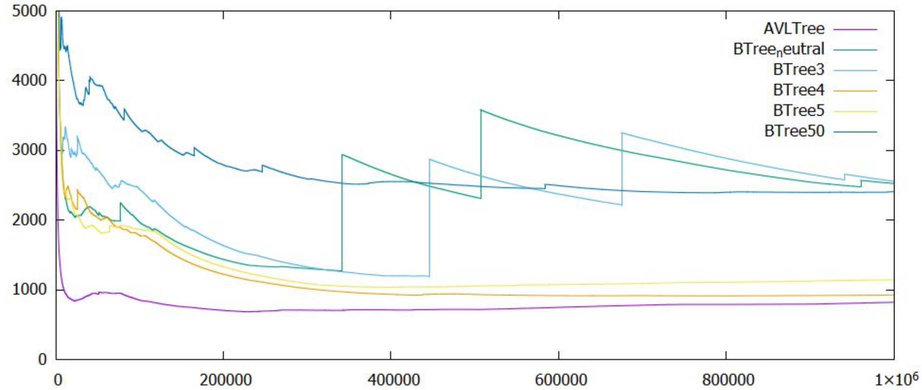
TD/TP4

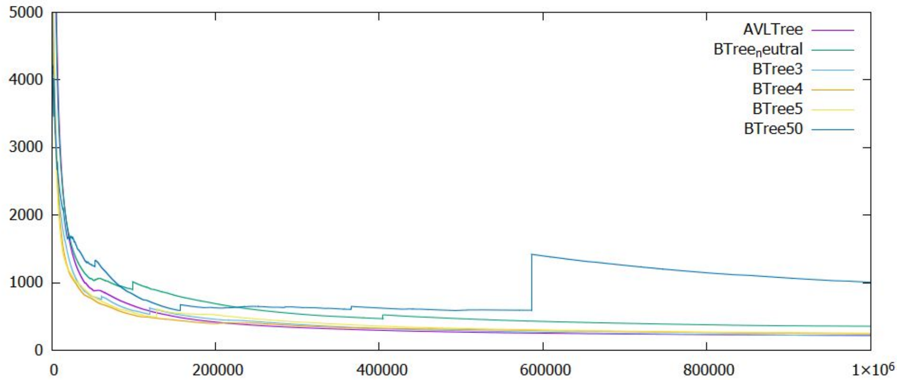
1)2)3) Voir commentaires des fichiers AVLTree.java, BinarySearchTree.java et BTree.java

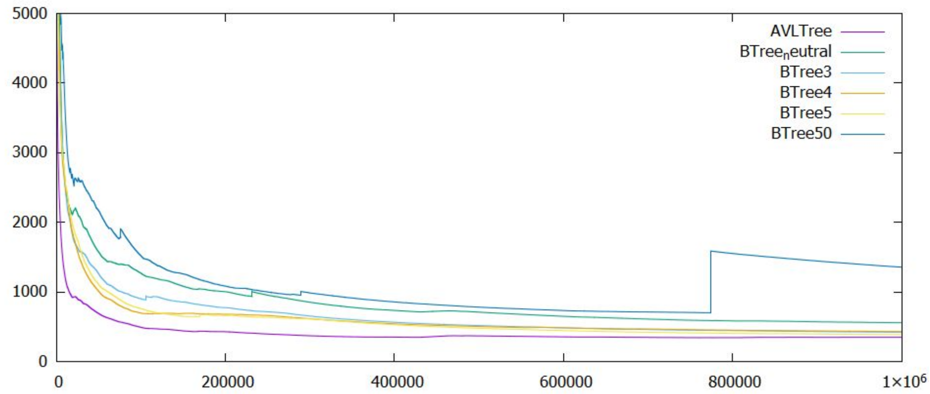
4)5) Voici le schéma que l'on obtient pour le temps d'insertion croissant, d'insertion random, d'insertion croissante et suppression, et d'insertion random et suppression

(voir fichiers i.png, r.png, ri.png et rr.png), les numéros suivant BTree-x représentent leurs ordres, c’est à dire le nombre minimal de valeurs pour noeud non-racine:









Nous pouvons voir dans un premier temps que dans tous les cas cités précédemment, l’arbre AVL est plus performant, ce qui est normal puisque la

conception des cette structure est spécialisée dans l'équilibrage ainsi à chaque nouvelle valeur sa structure se modifie, ce qui évite les divers pics que l'on peut voir dans les B-arbre, cependant si l'insertion est croissant et que l'ordre est très bas on obtient des résultats quasi-équivalent à celui des arbres AVL qui est logique puisqu'il font les même manipulations de mémoire.
Nous avons donc vu qu'en terme de temps l'arbre AVL est plus intéressant, on peut donc se poser la question de l'intérêt des B-arbres, en réalité les B-arbres sont plus utiles pour réduire coût en lecture-écriture sur disque, en effet comment un noeud contient au moins x valeurs selon l'ordre configurer, le noeud qui est enregistré en mémoire à plus de chance d'avoir la valeur que l'on recherche la où le noeud d'un arbre AVL ne possède qu'une valeur, ce qui nécessite de récupérer plusieurs noeuds et ainsi faire plusieurs recherches.