IFT2015 automne 2016 - Devoir1

Philippe Caron

17 novembre 2016

1 Tableaux de hachage

En premier lieu, il fallait s'assurer de pouvoir supporter `null`, pour ce faire il a suffit d'étendre la classe `LinearProbing` et de remplacer l'élément nul par un nouvel objet, nommé `VOID`. Puis il suffit de modifier les fonctions qui utilisaient `null` pour supporter le nouvel élément nul, ce qui a donner la classe `NLinearProbing` de laquelle découlent les deux prochaines classes.

1.1 Suppression paresseuse

Nommée `LazySet`, la classe qui permet la suppression paresseuse hérite de `NLinearProbing`. La différence principale est l'utilisation de la «pierre tombale» `DELETED` pour la suppression. Ceci permet d'implémenter `remove`, qui permet de retirer un élément du tableau et de réduire sa taille lorsqu'il descend sous un certain facteur. Il a également fallu modifier `add` pour accomoder le nouvel object `DELETED`. Pour se faire la nouvelle fonction `searchDel` a été créée. Elle est essentiellement identique à `search` cependant elle cherche aussi les cases supprimées ce qui permet de recycler les cases supprimées.

1.2 Suppression impatiente

La classe nommée `HastySet` implémente un tableau de hachage supportant la suppression impatiente. Tout comme la précédente, elle hérite de `NLinearProbing` afin de supporter les éléments null. Vu que cette classe déplace les éléments vers l'avant après une suppression, elle ne nécéssite pas que ses méthodes de recherche et d'ajout soient supprimées. La seule méthodes implémentées est donc `remove`. Elle fonctionne de la même manière que celle de la suppression paresseuse au nouveau du *rehash*, mais au lieu de remplacer l'élément supprimé par un marqueur, elle déplace tous les éléments subséquents vers l'avant.

2 Résultats

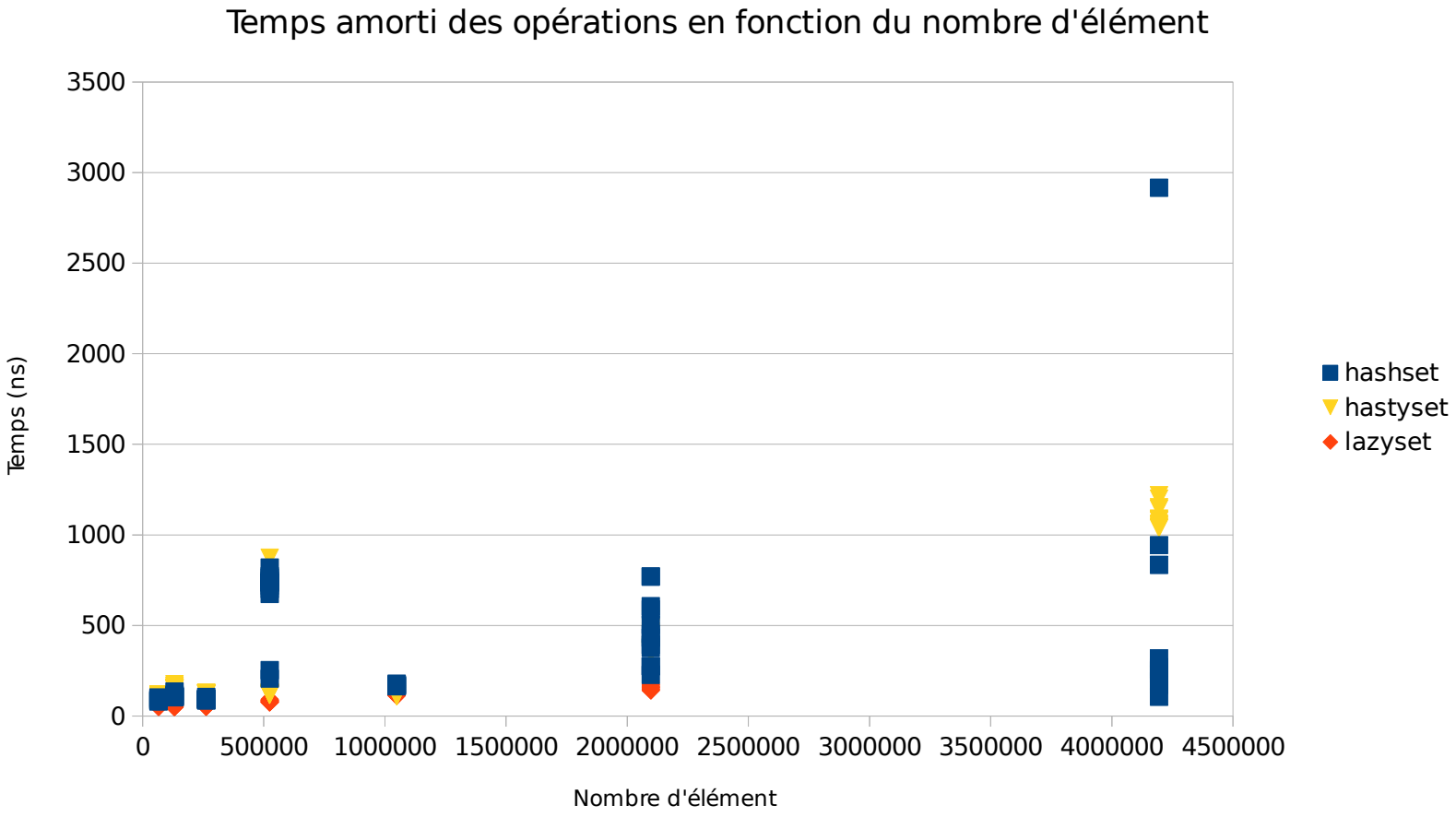
Le fichier de résultat a été produit à l'aide de la commande suivante :

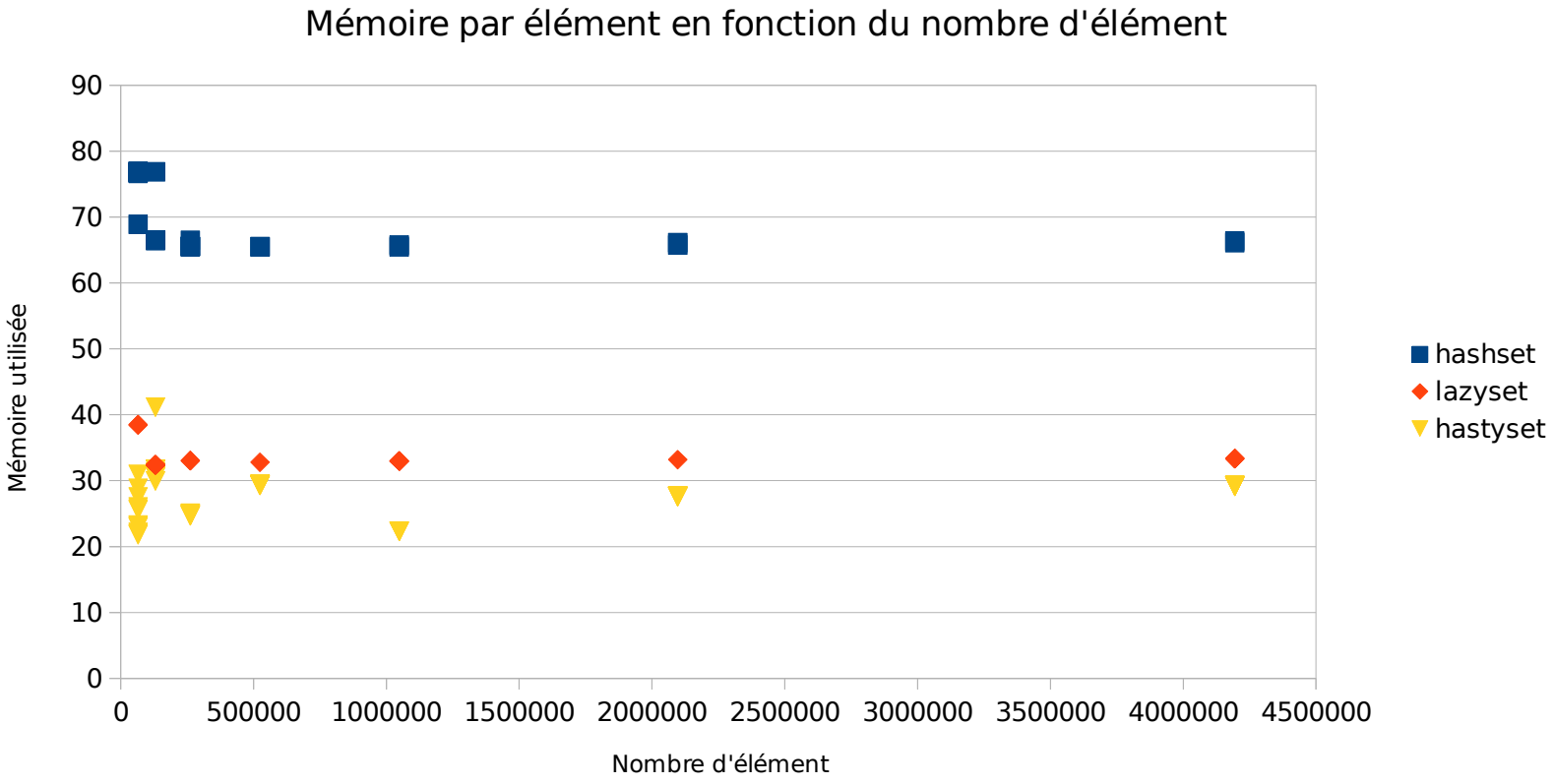
```
% bash -c 'for t in Hash Lazy Hasty; do for N in 65536 131072 262144 524288
1048576 2097152 4194304; do for rep in 1 2 3 4 5 6 7 8 9 10 11 12; do java
-Xmx2G -cp dev4.jar SetTester -type ${t} ${N}; done; done; done' > timings.log
```

Pour chaque type, toutes les tailles sont testées, et pour chaque taille le test est effectué 12 fois. Voici les spécifications de la machine de test :

```
Machine : Laptop Lenovo IdeaPad
Mémoire : 8GB
Processeur : Intel i7 @ 2.7 GHz (8 Threads / Quad Core)
OS : ARCH Linux 4.7.6-1
Java: OpenJDK 64-Bit Server VM 25.102-b14 (1.8.0_102-b14)
```

Les graphiques obtenus sont à la page suivante.


Temps amorti des opérations en fonction du nombre d'élément
Temps (ns)
3500
3000
2500
2000
1500
1000
500
0
0
500000
1000000
1500000
2000000
2500000
3000000
3500000
4000000
4500000
Nombre d'élément
hashset
hastyset
lazyset


Mémoire par élément en fonction du nombre d'élément
Mémoire utilisée
90
80
70
60
50
40
30
20
10
0
0
500000
1000000
1500000
2000000
2500000
3000000
3500000
4000000
4500000
Nombre d'élément
hashset
lazyset
hastyset

2.1 Vitesse

On peut voir qu'au niveau de la vitesse les trois implémentations sont très similaires, cependant on remarque toutefois que la suppression impatiente tend à prendre plus de temps dans les tableau qui présentent un grand nombre d'élément. La suppression paresseuse quant à elle semble légèrement plus efficace que l'implantation `HashSet` de java, mais c'est négligeable.

2.2 Mémoire

Au niveau de la mémoire, les deux implémentations qui héritent de `NLinearProbing` sont manifestement plus efficaces. Quoique relativement similaires entre elles, chacune prend significativement moins de mémoire que `HashSet`.