Les List en java et leur performance

Dans cet article nous allons revenir aux base du langage Java. J'ai passé plusieurs entretiens d'embauches techniques et j'ai été très étonné quand on m'a posé des questions sur les bases du langage, et plus précisément sur le package Collection, les List et les implémentations.

Qu'est ce qu'une List ? Voyons la définition de la javadoc :

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements.

Une liste nous permet de stocker des éléments tout en conservant l'ordre de ces éléments.

Différentes implémentations sont proposées :

- Vector : La liste est implémentée sous forme de tableau, et l'implémentation est thread-safe
- ArrayList : La liste est implémentée sous forme de tableau
- LinkedList : La liste est consistée d'éléments liées entre eux, comme des maillons d'une chaine
- CopyOnWriteArrayList : La liste est implémentée sous forme de tableau et chaque ajout à la liste va provoquer une copie vers un nouveau tableau.

J'ai toujours eu l'habitude d'utiliser des ArrayList, mais selon le type d'utilisation de la List il sera dpeut être plus efficace d'utiliser une autre implémentation.

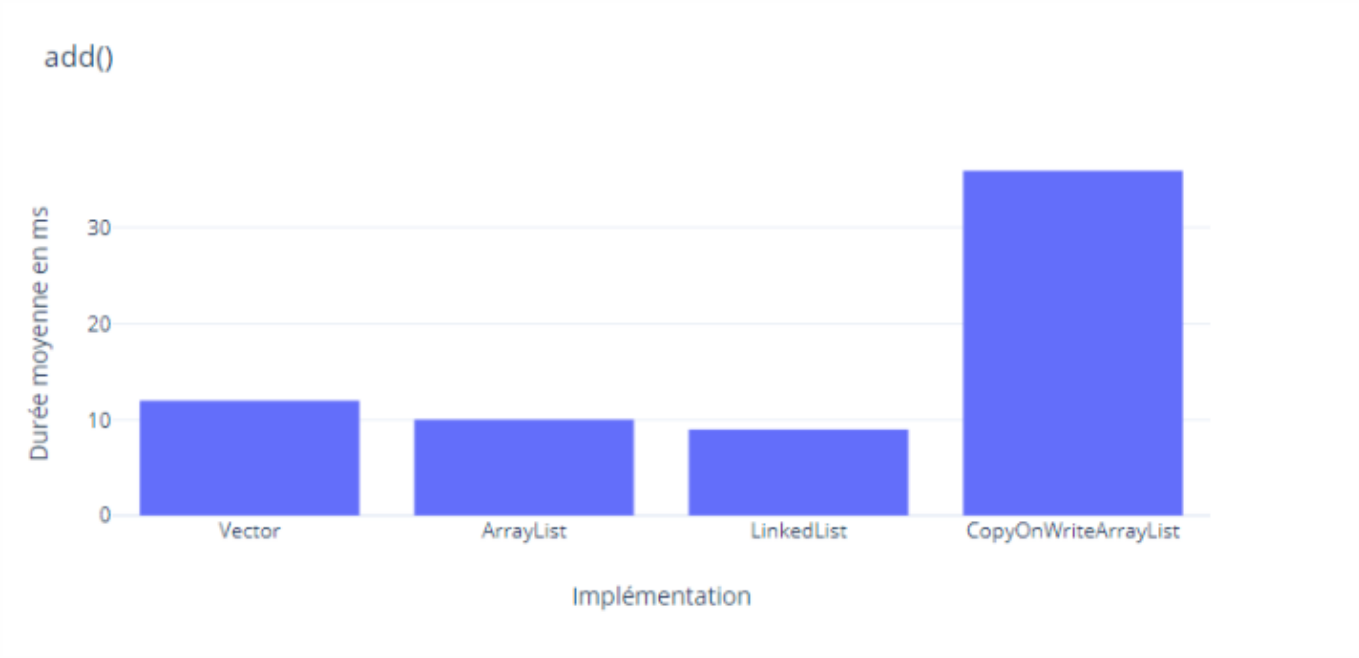
Voyons les performances de ces différentes implémentation pour les fonctions add, get, size et iterator.

Add

Nous allons ajouter 10000 éléments dans chacune de nos 4 implémentations de liste, et lancer les traitement 100 fois pour calculer une moyenne.

```
        for(int i = 0; i < 10000; i++) {
                liste.add("Ma chaine " + i);
        }
```

Voilà les résultats :



On constate que les performances sont assez identiques pour les implémentations Vector, ArrayList et LinkedList. L'implémentation Vector est synchronisée et threadsafe alors que les deux autres implémentations ne le sont pas, ce qui explique un léger impact sur les performances.

CopyOnWriteArrayList a des performances catastrophiques, comme on peut s'y attendre, cela s'explique par son implémentation :

A thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

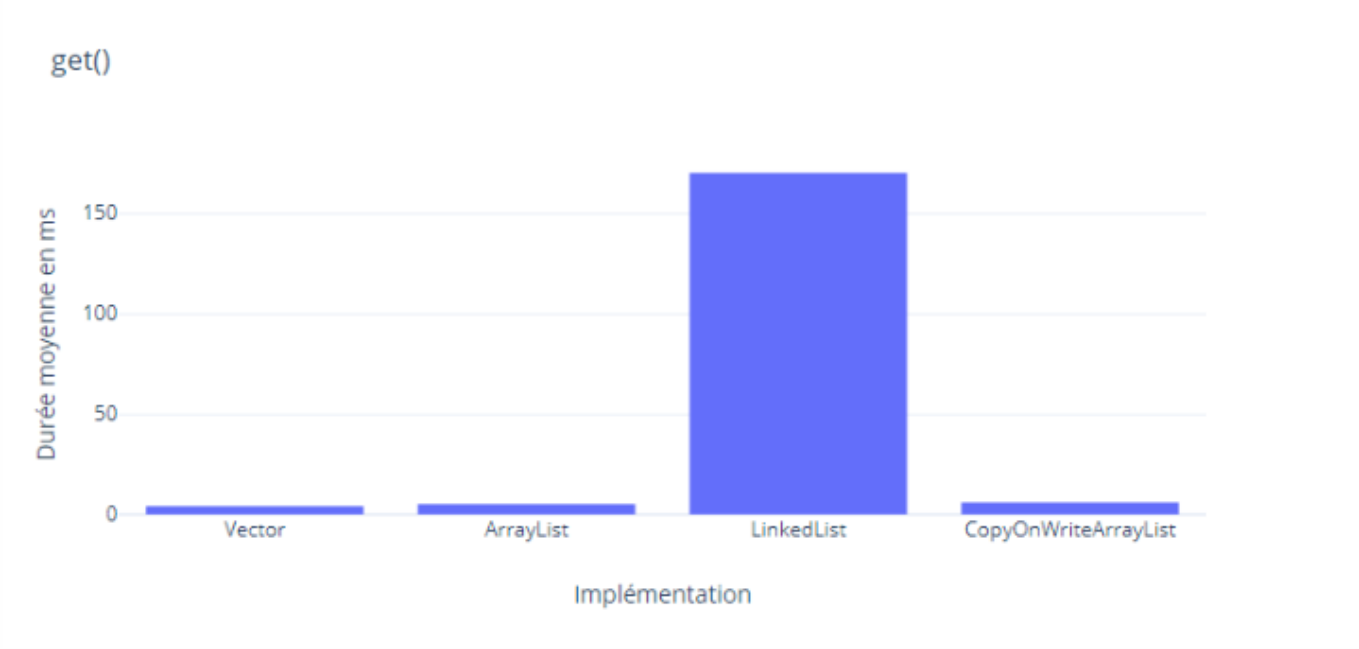
Conclusion :

- Si votre code n'est pas utilisé dans un environnement multi thread, alors autant utiliser l'ArrayList. Sinon, privilégier le Vector.
- On voit aussi que notre code est très dépendant de l'implémentation que nous allons utiliser : Un même code peut etre 3 x moins performant si on utilise une implémentation mal adaptée

Get

Voyons comment se comportent nos implémentations pour récupérer les éléments aléatoirement dans notre liste :

```
        for(int i = 0; i < 10000; i++) {
                liste.get(Random(0, TAILLE_LISTE));
        }
```



Ici nous voyons que l'implémentation la plus lente est la LinkedList, ce qui est assez logique : Pour les 3 autres implémentations, les données sont stockées sous forme de tableau et il est donc possible d'accéder directement à l'élément. Pour une LinkedList, chaque élément est lié à l'élément précédant et suivant, si on veut accedéer à l'élément 50 de la liste, on devra parcourir les 49 premiers éléments...

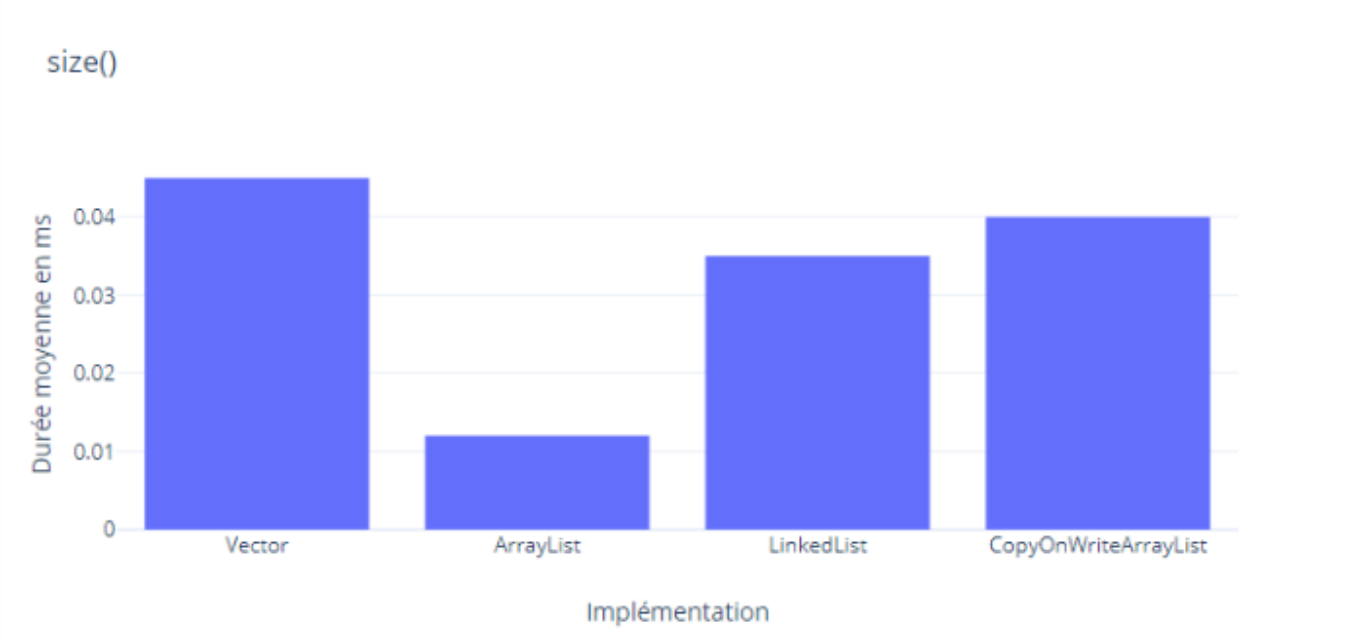
Conclusion :

- Si l'accès à notre liste se fait aléatoirement, alors il faut absolument éviter d'utiliser une LinkedList.
- Si les ajouts dans la liste sont toujours réalisés à la fin, alors privilégier le Vector ou l'ArrayList. En effet, pour une List de taille 10, si on ajoute un élément en position 5, il faudra décaller tous les éléments suivants. Dans une LinkedList il suffit d'ajouter une référence vers l'élément précédant et suivant.

Size

Dans ce test nous allons voir les performances du calcul de la taille de la List :

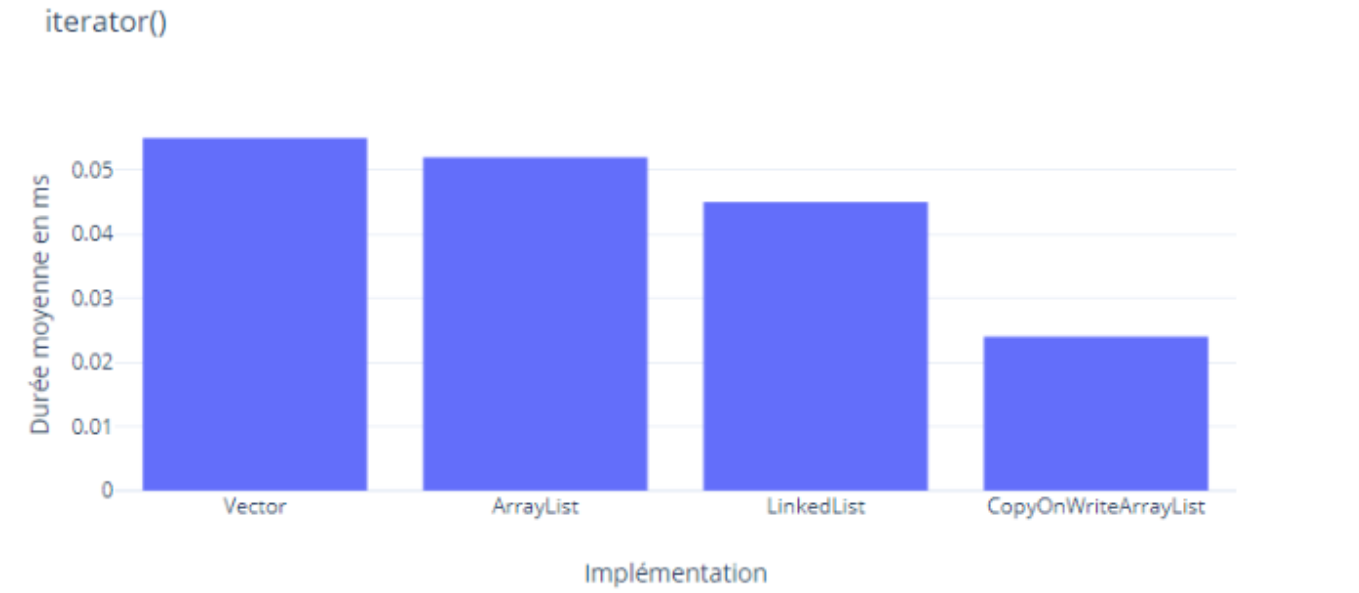
```
        for(int i = 0; i < 10000; i++) {
                liste.size(Random(0, TAILLE_LISTE));
        }
```



Ici rien à dire les temps sont négligeables, bien que la représentation graphique puisse nous faire penser le contraire les temps sont négligeables.

Iterator

Et si on parcours tous les éléments de la List ?



Idem, les temps sont sensiblement identiques.

Conclusion

Comme nous l'avons vu, il est très interessant de comprendre le fonctionnement des implémentations proposées par JAVA.

Si votre code est utilisé dans un environnement concurent, alors il faudra utiliser Vector ou CopyOnWriteArrayList. Si vous devez ajouter des éléments dans la List (pas uniquement à la fin), et que vous n'avez pas beaucoup

d'accès à la liste : On choisit la LinkedList. Si on fait autant d'ajout que de lectures aléatoires, alors choisir la LinkedList.

Comme les Vector et ArrayList stockent leur données dans un tableau, ce tableau a une taille figée. Deux inconvénients :

- Pour une liste de petite taille, le tableau sera bien trop grand et il est possible de préciser la taille de la liste lors de la création :

```
Vector(int initialCapacity)
//Constructs an empty vector with the specified initial capacity and with
its capacity increment equal to zero.
```

- Qu'est ce qu'il se passe si on dépasse la capacité initiale du tableau ? Dans ce cas l'implémentation va devoir créer un nouveau tableau ce qui est aussi couteux en performance. mais on peut aussi préciser de combien l'élément sera incrémenté le nouveau tableau :

```
Vector(int initialCapacity, int capacityIncrement)
//Constructs an empty vector with the specified initial capacity and
capacity increment.
```

La vraie conclusion de cet article :