**Analyse de l’exercice #2**

**Résultats :**

**Remplissage**

|  |  |  |
| --- | --- | --- |
| **Type** | **Temps pour remplissage en ms** | **Nombre d’éléments** |
| ArrayList | 2383.0461 | 70,000,000 |
| LinkedList | 10209.6985 |
| Array | 718.6416 |
| ArrayList | 259.4325 | 9,000,000 |
| LinkedList | 910.2177 |
| Array | 80.527501 |
| ArrayList | 15.491099 | 1,000,000 |
| LinkedList | 10.2494 |
| Array | 4.9168 |

**Addition**

|  |  |  |
| --- | --- | --- |
| **Type** | **Temps pour addition des éléments du tableau en ms** | **Nombre d’éléments** |
| ArrayList | 0.6297358 | 10,000,000 |
| LinkedList | 2.1200321 |
| Array | 0.0167136 |
| ArrayList | 2.6925624 | 50,000,000 |
| LinkedList | 13.3766082 |
| Array | 0.5210043 |

**Analyse :**

En analysant ce tableau ci-dessus, nous pouvons, au premier regard, remarquer que l’utilisation de LinkedList est définitivement moins efficace si nous le comparons avec Array ou même avec ArrayList. En effet, les résultats portent à conclure que l’utilisation de LinkedList va toujours engendrer un temps plus long dans le remplissage et ceci va nécessiter plus de mémoire pour obtention des données. Cette comparaison est faite en tenant compte des deux autres types mentionnés auparavant. En rentrant plus en détails, on observe que cette conclusion est totalement justifiable du fait que le type de structure de donnée d’un tableau simple n’est pas efficace si nous voulons remplir un tableau par un nombre d’éléments indéfinis, il faudrait ainsi déclarer la taille du tableau avant de commencer à le remplir. Le temps de recherche linéaire est plus lent dans LinkedList en raison de la localité de référence.

Pour ce qui en est avec ArrayList, si nous le comparons avec Array, l’utilisation de Array restera la plus efficace. Lorsque nous rajoutons un élément à ArrayList et qu'il se déborde, dans ce cas, l’implémentation de la classe ArrayList permettra ce débordement en créant un nouveau tableau et en copiant chaque élément de l'ancien tableau vers le nouveau. En faisant ce processus, on obtiendra ainsi une structure de donnée dynamique qui prend plus de temps qu’un simple tableau statique. De plus, nous pouvons aussi faire le lien avec le temps de complexité soit pour Array, le temps de complexité d’insertion c’est O(1), tandis que pour ArrayList c’est O(n) au cas de débordement, ce qui affirme la certitude de nos résultats.

**Conclusions et recommandations:**

Nous pouvons en tirer la conclusion que l’utilisation de Array, si on a un nombre fixe d’éléments à sauvegarder dans un tableau, est évidemment la plus efficace en comparant avec LinkedList et ArrayList, elle nécessite moins d’espaces mémoire et moins de temps dans l’addition de tous les éléments ensemble contenus dans le tableau.

Cependant, ArrayList est essentiellement un tableau dynamique qui commence initialement avec une taille raisonnable mais petite, si on travaille à l’ordre de millions d’éléments, ça serait mieux si on implémentait ArrayList avec une grande taille initiale, pour minimaliser les instructions de créer un nouveau tableau et copier les anciens éléments dans un nouveau tableau. En résumé, si on a un nombre d’éléments indéfinis à sauvegarder que nous ne pouvons pas déterminer à l’avance, nous devrions opter pour l’utilisation d’ArrayList, puisqu’avec l’ajout des éléments au fur et à mesure, cette implémentation de liste dynamique devrait être plus efficace qu’un tableau statique.

Par contre, LinkedList pour effectuer la recherche d’un élément aura une complexité de O(n), sa méthode get() n’as pas le même temps d’exécution que ArrayList qui peut chercher un élément quelconque dans le tableau placé aléatoirement avec une complexité de O(1) , c’est pour cela pendant l’addition la différence entre ArrayList et un vecteur statique n’est pas perceptible.