Audinot Noah
Arthur Hollebeque
Groupe E

Comparaison d'approches algorithmiques
SAE S1.03

1) Introduction

L'objectif de la SAE est d'explorer l'efficacité de trois approches distinctes pour la mise en œuvre de listes triées de chaînes, pour différents ensembles de données de taille variable. Les opérations se concentrent spécifiquement sur l'ajout et la suppression d'éléments au sein de ces listes.

Trois méthodes d'implémentation sont envisagées :

- **Représentation contiguë dans un tableau** Cette approche consiste à stocker les chaînes de manière contiguë dans un tableau.

- **Représentation chaînée dans un tableau avec récupération des places libérées** : Dans cette configuration, les chaînes sont gérées sous forme de listes chaînées dans un tableau, avec la récupération des emplacements libérés.

- **Représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste** : Ici, les chaînes sont également organisées de manière chaînée dans un tableau, mais avec une gestion de l'espace libre par le biais d'une liste dédiée.

À travers cette démarche comparative, nous chercherons à dégager des conclusions éclairées sur la performance relative de ces trois méthodes pour les opérations d'ajout et de suppression au sein de listes triées de chaînes.

Pour cette SAE il y a eu 2 environnements expérimentaux. L'un sur un des pc de l'iut sous WIndows et l'autre sur une machine aussi Windows avec 16GB DDR4 de mémoire et un processeur AMD Ryzen 5 5600G avec 6 coeurs, 12 threads et une fréquence de 3.9GHz. Les résultats sont effectués depuis la deuxième machine.

2) Question 9

SI la chaîne n'est pas présente et qu'on utilise suplisT, l'algo va parcourir toute la liste, il est donc pas nécessaire de répéter 10 fois pour connaître le temps vu que l'exécution devrait être quasi similaire et les actions effectuées seront les mêmes.

3) Algo logiques

```
fonction adjlisT (l:liste(chaîne), c:chaîne)
        début
        p <- tête(l)
        insérée <- faux
        pPre <- p
        tant que non finliste(l,p) et insérée = faux faire
           si val(l,p) > c alors
                si p = tête(l) alors
                        adjtlis(l,c)
                sinon
                        adjlis(l,pPre,c)
                fin si
                insére <- vrai
           sinon
                pPre <- p
                p <- suc(l,p)
           finsi
        fintantque
        si non insérée alors
          adjlis(l,p,c)
        finsi
        fin
```

Lexique:
p : Place, place dans la liste
pPre : Place, place avant p
insérée : booléen, vrai si la chaîne a été insérée

```
fonction suplisT (l:liste(chaîne), c:chaîne)
        début
        p <- tête(l)
        supprime <- faux
        tant que non finliste(l,p) et supprime = faux faire
            si val(l,p) = c alors
                suplis(l,p)
                supprime <- vrai
            sinon
                p <- suc(l,p)
```

```
        finsi
    fintantque
    fin
```

Lexique:
p : Place, place dans la liste
supprime : booléen, vrai si la chaîne a été supprimée

```
fonction memlisT (l:liste(chaîne), c:chaîne) : booléen
    début
    p <- tête(l)
    meme <- faux
    tant que non finliste(l,p) et meme = faux faire
        si val(l,p) = c alors
            meme <- vrai
        sinon
            p <- suc(l,p)
        finsi
    fintantque
    retourne meme
    fin
```

Lexique:
p : Place, place dans la liste
meme : booléen, vrai si la chaîne a été trouvée

4) Expériences réalisées

Il y eut pour chacune des 3 méthodes d'implémentation, 4 tests (Ajouter au début, Ajouter à la fin, Supprimer au début et Supprimer à la fin) avec 3 listes de tailles différentes (100, 1000, 10000). Soit 36 tests.
Chacun des test est effectués de la même manière :

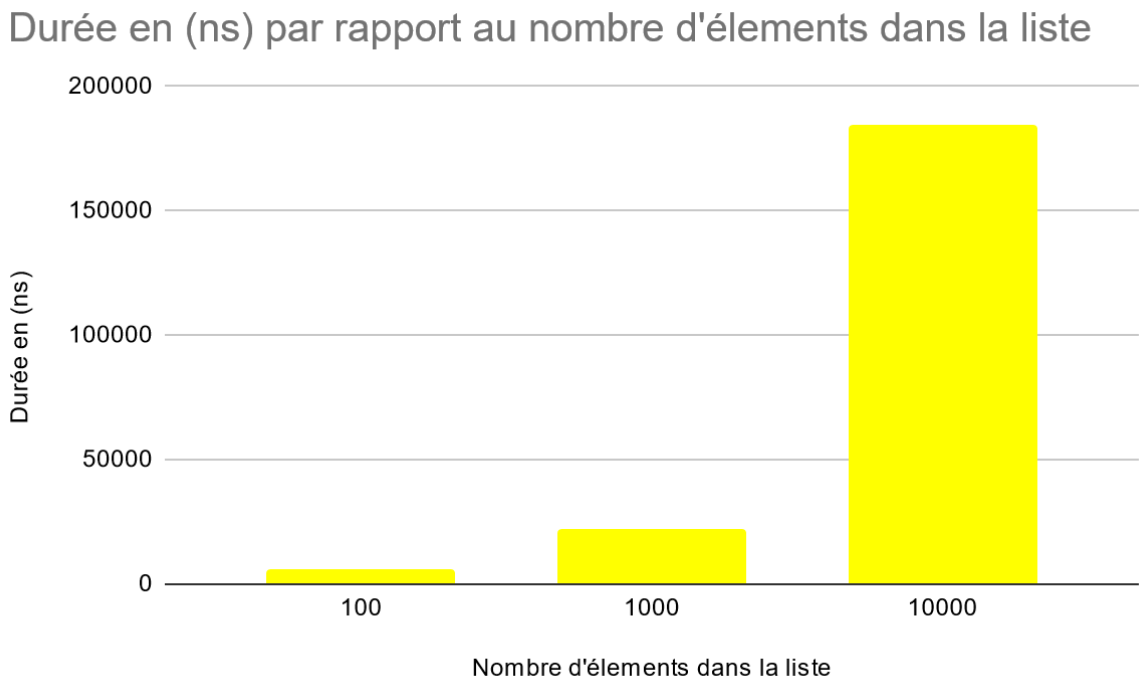
- Création de la liste avec l'algo choisis et la taille
- Ajout du nombres d'éléments définis selon le test (100, 1000, 10000)
- Démarrage du chronomètre
- Ajout ou Suppression de 10 éléments sur la liste
- Arrêt du chronomètre

Tous les résultats sont en nanosecondes.

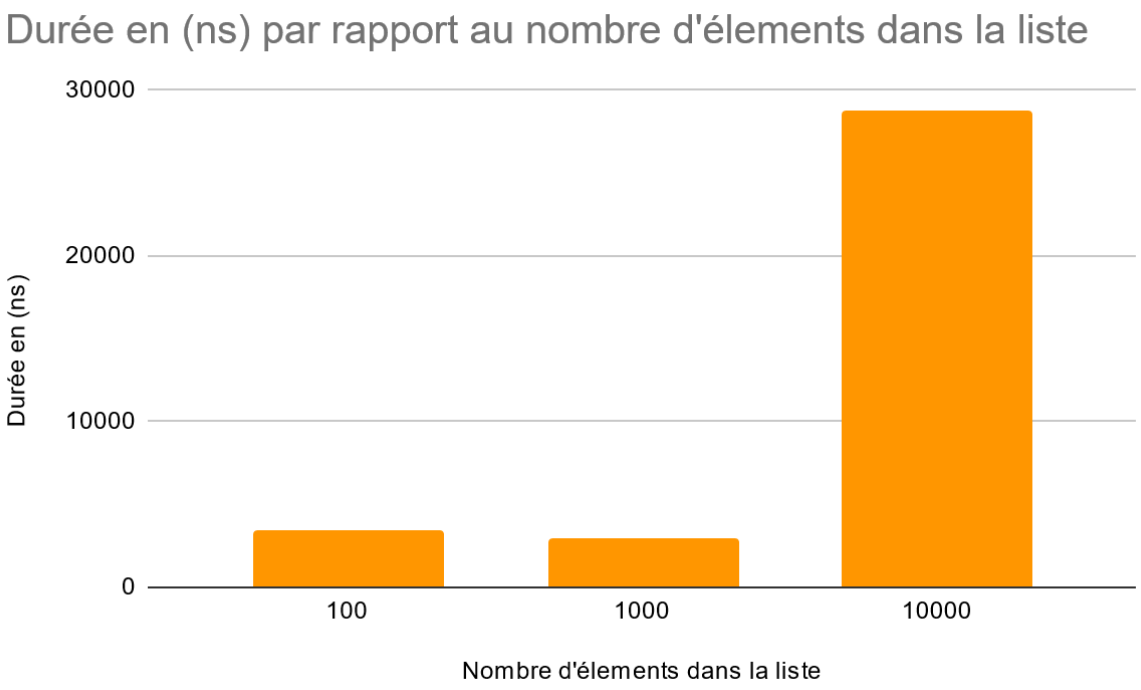
Ajouter au début :

Type d'algo / Nombre d'elements dans la liste	100	1000	10000
contigue	*5725*	*22407*	*184626*
chainee	*3410*	*2933*	*28724*
chainee libre	*1264*	*466*	*256*

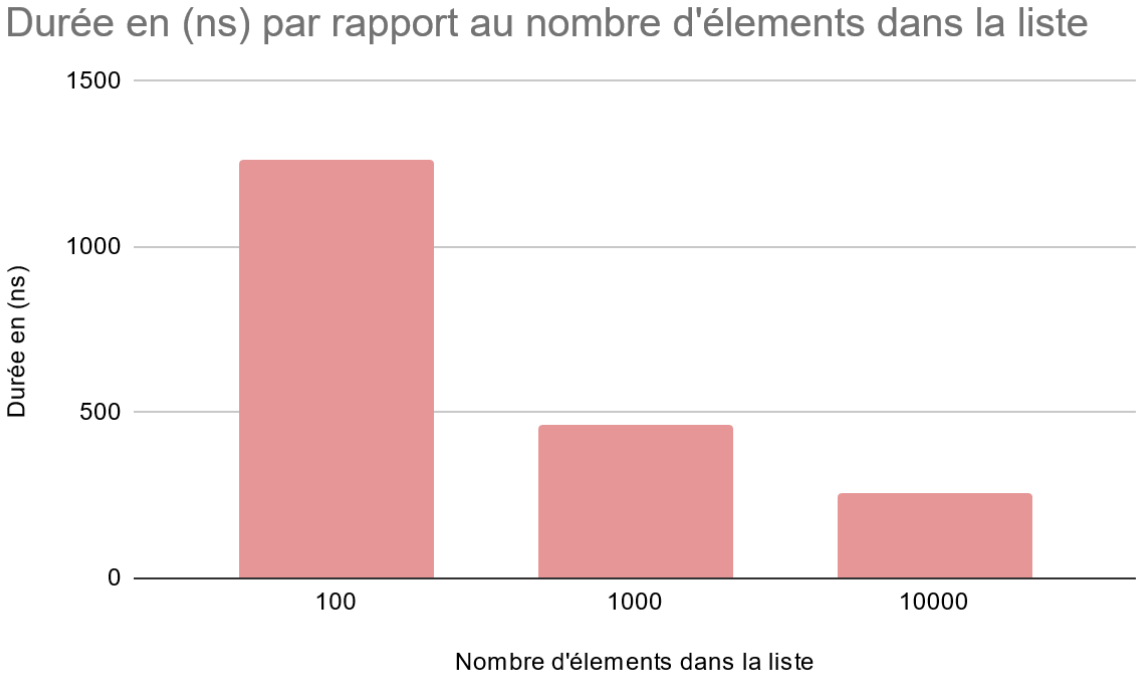
- Pour la représentation contiguë dans un tableau :



- Pour la représentation chaînée dans un tableau avec récupération des places libérées :



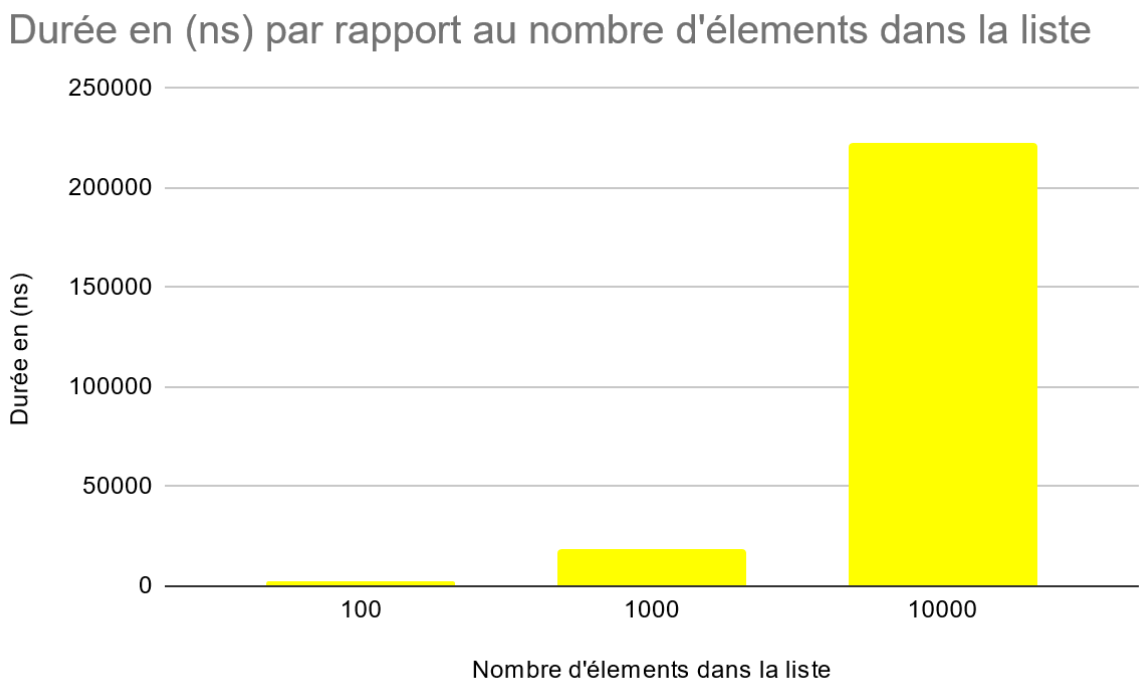
- Pour la représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste :



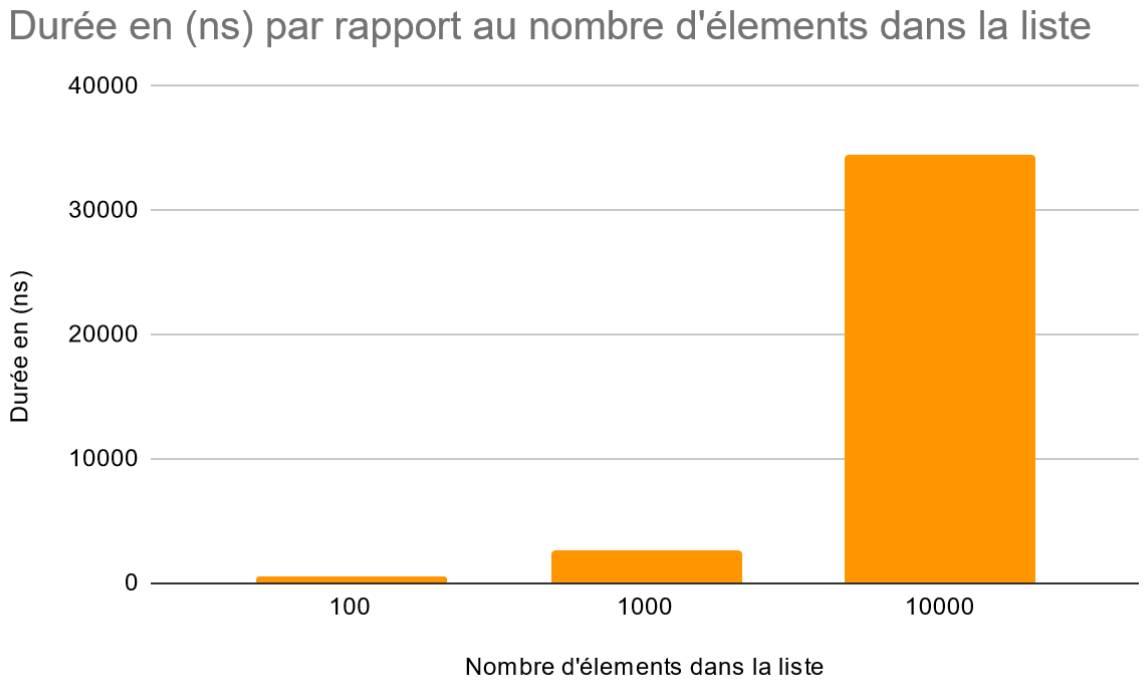
Ajouter à la fin :

Type d'algo / Nombre d'elements dans la liste	100	1000	10000
contigue	*2199*	*18979*	*222741*
chainee	*571*	*2717*	*34524*
chainee libre	*285*	*274*	*278*

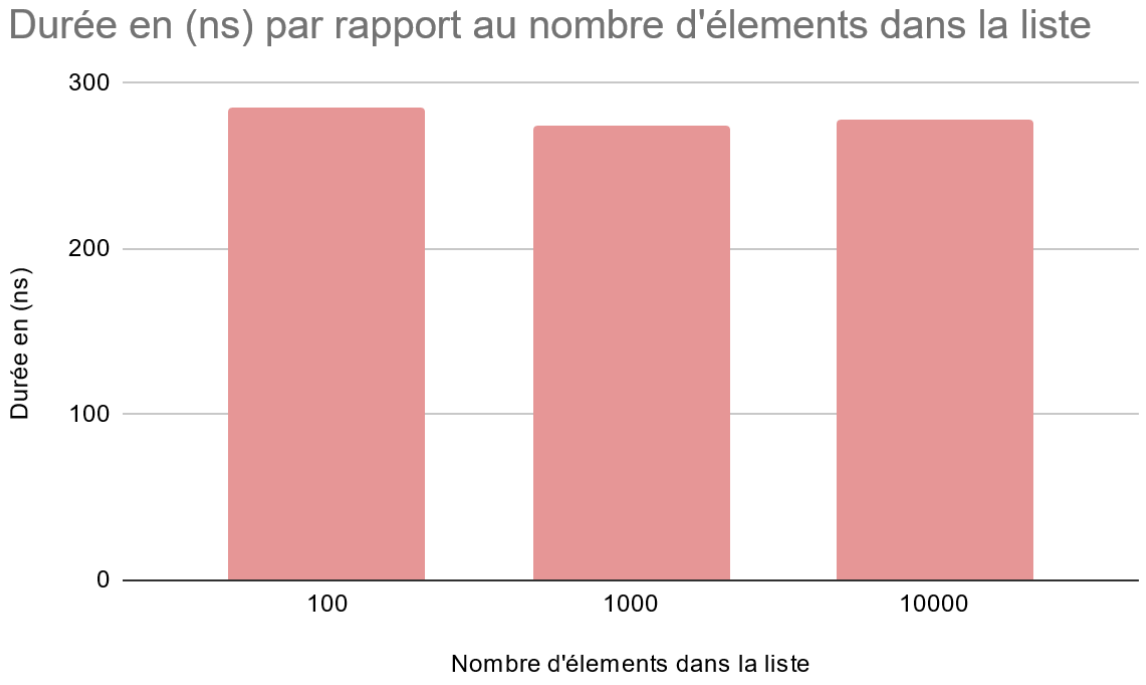
- Pour la représentation contiguë dans un tableau :



- Pour la représentation chaînée dans un tableau avec récupération des places libérées :



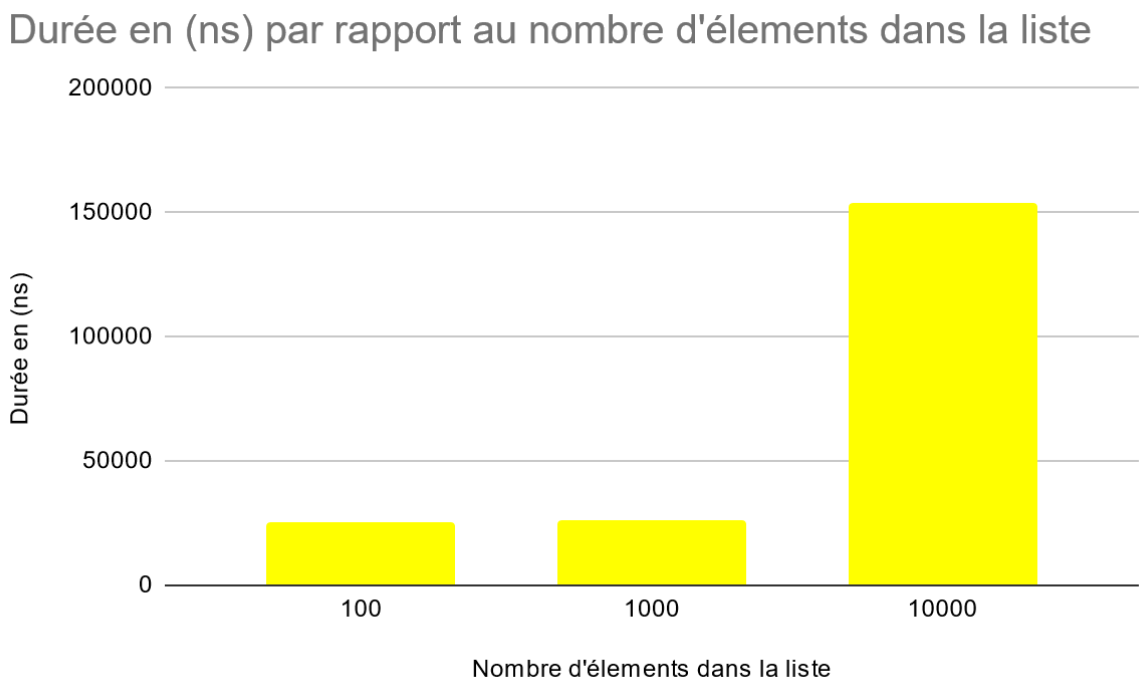
- Pour la représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste :



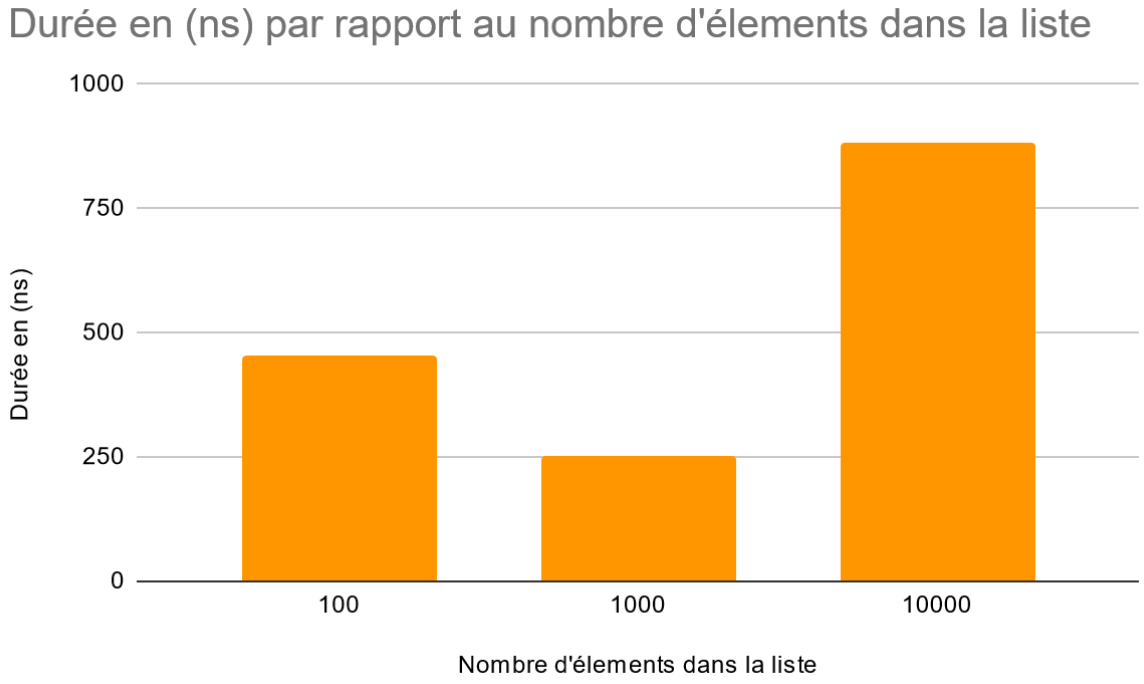
Supprimer au début :

Type d'algo / Nombre d'elements dans la liste	100	1000	10000
contigue	*25547*	*26223*	*153740*
chainee	*453*	*254*	*882*
chainee libre	*369*	*234*	*291*

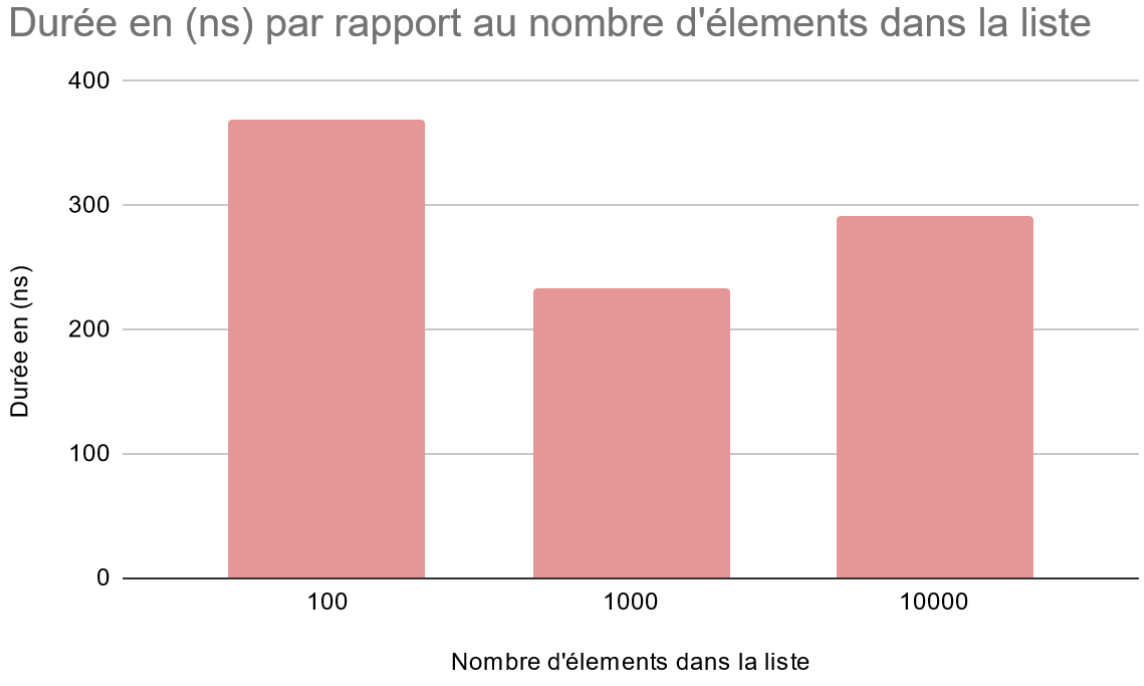
- Pour la représentation contiguë dans un tableau :



- Pour la représentation chaînée dans un tableau avec récupération des places libérées :



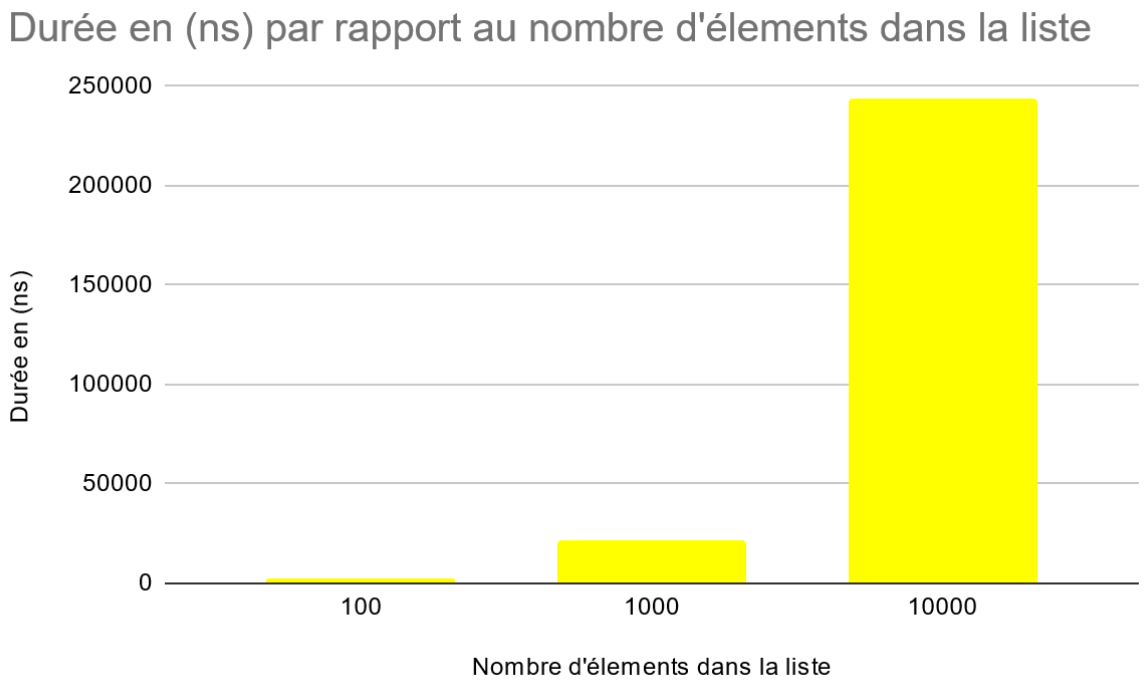
- Pour la représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste :



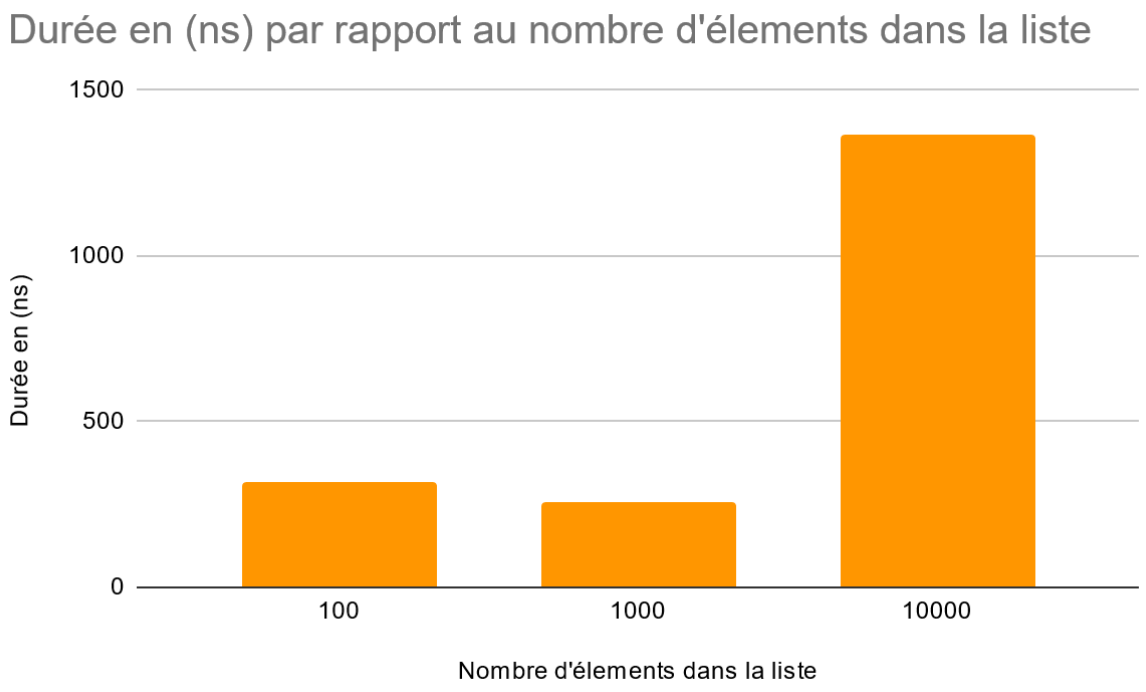
Supprimer à la fin :

Type d'algo / Nombre d'elements dans la liste	100	1000	10000
contigue	*1972*	*21218*	*243974*
chainee	*316*	*257*	*1369*
chainee libre	*371*	*154*	*148*

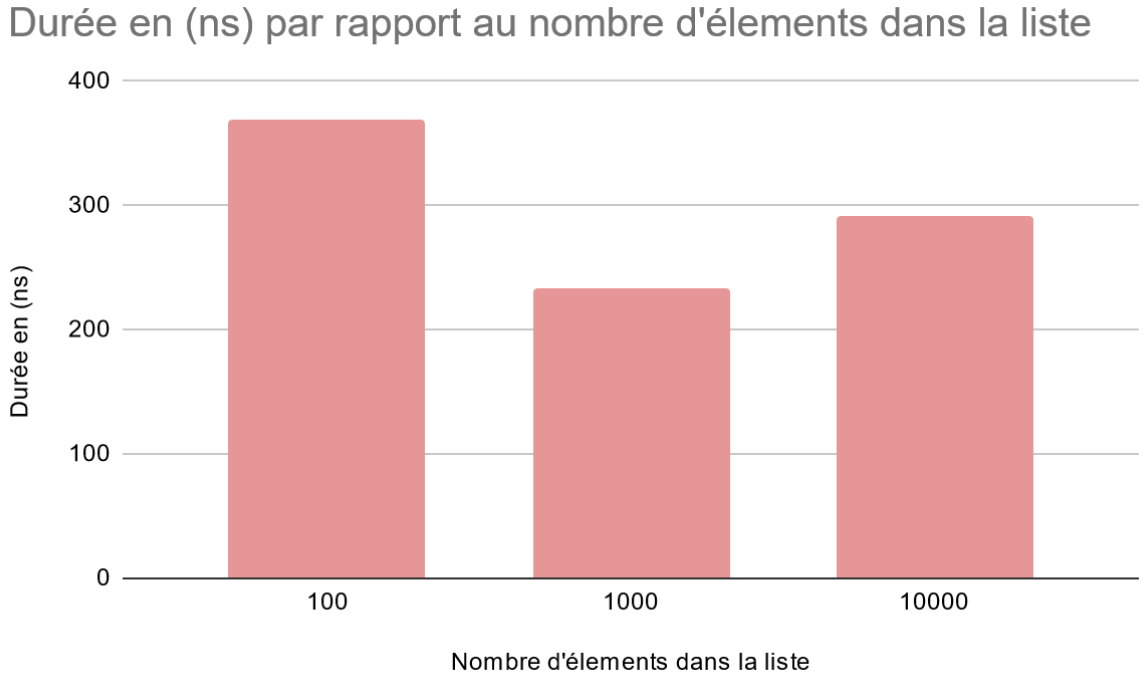
- Pour la représentation contiguë dans un tableau :



- Pour la représentation chaînée dans un tableau avec récupération des places libérées :



- Pour la représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste :



5) Conclusion de l’efficacité

En conclusion, la meilleure méthode d'implémentation est la représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste pour les 4 expériences.
On remarque que la différence est assez faible entre les 3 méthodes pour 100 éléments dans la liste, mais avec 1000 ou 10000 la liste contiguë performe très mal comparé aux 2 autres .
La liste chaînée libre performe un peu près pareil tout le temps alors que la non libre commence perd de son efficacité à >1000 éléments

La raison de pourquoi une implémentation chaînée est plus rapide qu’une implémentation contiguë, est que la chaînée ne nécessite pas de devoir décaler tous les éléments suivants quand on ajoute et supprime. La rendant plus rapide lors de ses actions par contre elle sera plus longue pour chercher un élément car l’accès indirect aux éléments, nécessite souvent un parcours de la liste pour le trouver.

La raison pour laquelle la représentation libre est plus rapide s’explique car il permet une meilleure utilisation de l'espace mémoire disponible donc est parfait pour ajouter et supprimer mais peut être encore plus coûteux lors d’une recherche.

Alors quelles options choisir ?

Chaque méthode d'implémentation à ses avantages et ses inconvénient mais en général :

- La représentation contiguë dans un tableau sera meilleure si vous comptez faire beaucoup de recherche dans votre liste et que vous ne cherchez pas à ajouter ou supprimer des éléments.
- La représentation chaînée dans un tableau avec récupération des places libérées permet de faire les deux, si vous avez souvent besoin de rechercher et d'ajouter et supprimer des éléments, elle sera la plus adaptée.
- La représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste est parfaite si vous comptez utiliser des listes très remplis ou si vous comptez surtout ajouter et supprimer des éléments.