RAPPORT SAE COMPARAISONS D'APPROCHES ALGORITHMIQUES

(HENRY Nicolas et RENARD Tanguy)

Algo logique :

```
Fonction adjlisT (lch InOut : Liste(Chaine) , c : chaine)
Début
    p ← tête (lch)
    trouvé ← faux
        si finliste(lch,p) alors
            adjqlis (lch,c)
        sinon
            Tant que non finliste(lch,p) et non trouvé faire
                ch ← val(lch,p)
                si ch >= c alors
                    trouvé ← vrai
                sinon
                    placePré ← p
                    p ← suc(lch,p)
                fsi
            ftant
            si p = placePré alors
                adjtlis (lch,p)
            sinon
                adjlis (lch,placePré, c)
            fsi
        fsi
fin
```

Lexique :

- **lch** : Liste(Chaine), Liste triée de chaine de caractères
- **c** : Chaine , chaine de caractère a ajouté dans la liste
- **p** : entier, place courante dans la liste
- **trouvé** : booléen, retourne vrai quand la chaine de caractère a une place dans la liste
- **ch** : chaine, valeur à la place courante
- **placePré** : entier, place précédant la place courante

fonction suplisT(liste InOut : Liste(Chaine), chaîne chaîne de caractère)

```
début
    p ← tete(liste)
    arret ← faux
    tant que non finliste(liste,p) et non arret faire :
        chaîneC ← val(liste,p)
        si ( chaîneC > chaîne )
            p ← suc( liste , p)
        sinon si ( chaîneC < chaîne )
            arret ← vrai
        sinon
            suplis(liste,p)
            arret ← vrai
        fsi
    fsi
fin tq
fin
```

Lexique :

Liste : Liste(Chaine), liste trié de chaine de caractère

chaîne : chaîne de caractère : chaîne de caractère à supprimer de la liste

p : Place : place courante dans la liste

arret : booléen : une des conditions d'arrêt

chaineC : chaîne de caractère : valeur courante dans la liste

Fonction MemlisT (lch : Liste(Chaine), c : chaine) : booléen

```
Début
    P ← tête(lch)
    trouvé ← faux

        Tant que non finliste(lch,p) et non trouvé faire
            ch ← val(lch,p)
            si ch = c alors
                trouvé ← vrai
            sinon
                p ← suc(lch,p)
                si finliste(lch,p) alors
                    trouvé ← faux
                fsi
            fsi
        ftant
    retourne trouvé
fin
```

Lexique :

- **lch** : Liste(Chaine), liste triée de chaine de caractères
- **c** : chaine, élément dont on cherche l'appartenance dans la liste
- **p** : entier, place courante
- **ch** : chaine, valeur courante à la place p
- **trouvé** : booléen, retourne vrai si l'élément apparait dans la liste, faux sinon

Questions optionnelles :

Questions 7 : voir ci-dessus

Question 8 : voir ListeTriee.java

Question 12 :

- test_05_suppression_listeChainee_2_meme_chaine
- test_06_suppression_listeContigue_2_meme_chaine

→ Les tests 05 et 06 servent à voir si la liste supprime une seule des 2 chaînes de caractères identiques

- test_07_suppression_listeChainee_vide

- test_08_suppression_listeContigue_vide

→ Les tests 07 et 08 servent à voir si la méthode fonctionne bien et ne génère pas d'erreurs lorsque qu'on veut supprimer un élément alors que la liste est vide

- test_09_suppression_listeChainee_chaine_n_appartient_pas

- test_10_suppression_listeChainee_contigue_n_appartient_pas

→ Les tests 09 et 10 servent à voir si la méthode ne supprime rien si la chaîne de caractères n'est pas dans la liste

- test_11_memlisT_ListeChainee_OK

- test_12_memlisT_ListeContigue_OK

→ Les tests 11 et 12 servent à voir si la méthode détecte bien la chaîne de caractères présente dans la liste

- test_13_memlisT_ListeChainee_Non_Present

- test_14_memlisT_ListeContigue_Non_Present

→ Les tests 13 et 14 servent à voir si la méthode ne détecte pas la chaîne de caractères qui n'est pas dans la liste

Question 14 et 15:

Nous avons décidé d'utiliser le fichier texte suivant présent à l'adresse suivant comportant près de 218982 nom de famille au niveau nationale de la France :
https://www.insee.fr/fr/statistiques/3536630

Dans le fichier texte les noms de famille sont présents dans la première colonne et les colonnes sont séparées par des tabulations ('\t'). La méthode nom dans le fichier ListeGenerer.java permet d'extraire d'une ligne donnée un nom.

La méthode principale main avec un argument passer en paramètre permet de sélectionner aléatoirement X nom de famille DISTINCT tant que cet argument est inférieur à 218982.

Pour les détails du programme voir ListeGenerer.java et la liste de nom voir noms2008nat_txt.txt.

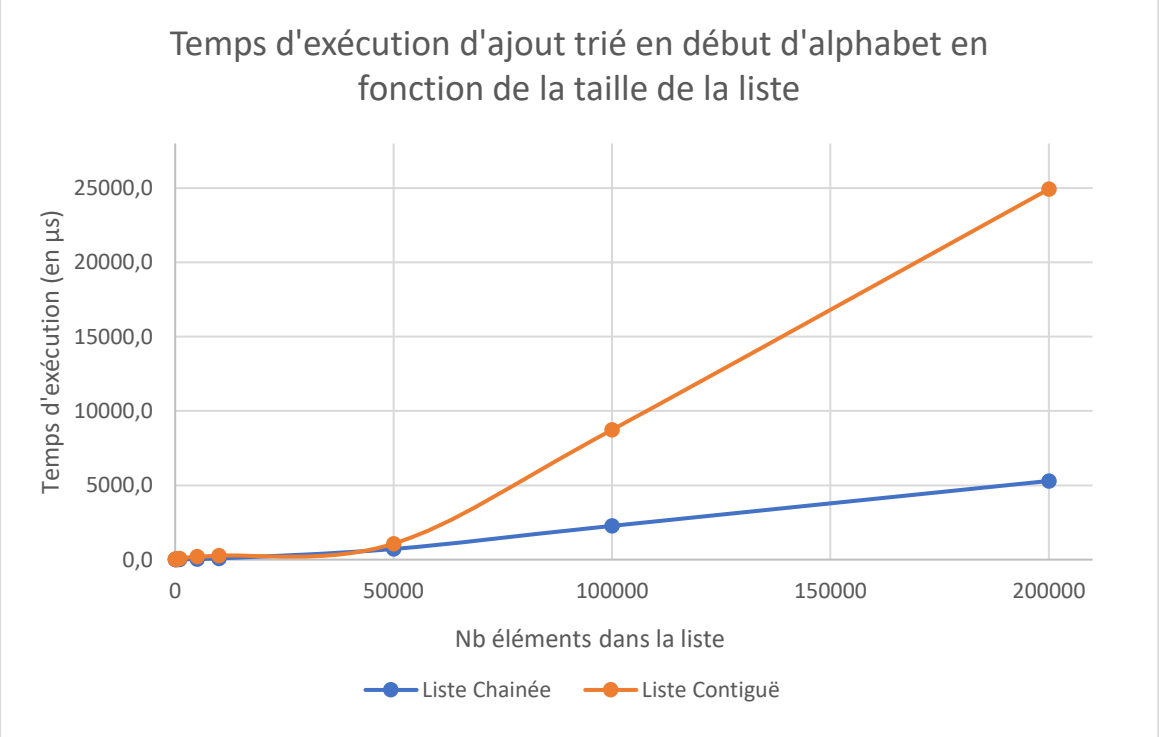
Expériences :

Les expériences réalisées consistent à mesurer le temps d'exécution d'ajout et suppression triés ainsi que de recherche d'éléments en début et fin d'alphabet dans des listes chainées et contiguës de différentes tailles. Ces mesures sont faites à l'aide d'un programme qui crée une liste puis lance un chrono qui s'arrête lorsque l'action est réalisée. Le but de ces expériences est de déterminer laquelle de ces deux types de listes est la plus efficace suivant l'action demandée. Ces mesures ont été prises sur une machine aux caractéristiques suivante :

- Processeur : Intel© Core i5-9300H CPU @2.40GHz x 8
- 8 GO de RAM
- Carte Graphique : NVIDIA GeFORCE GTX 1650

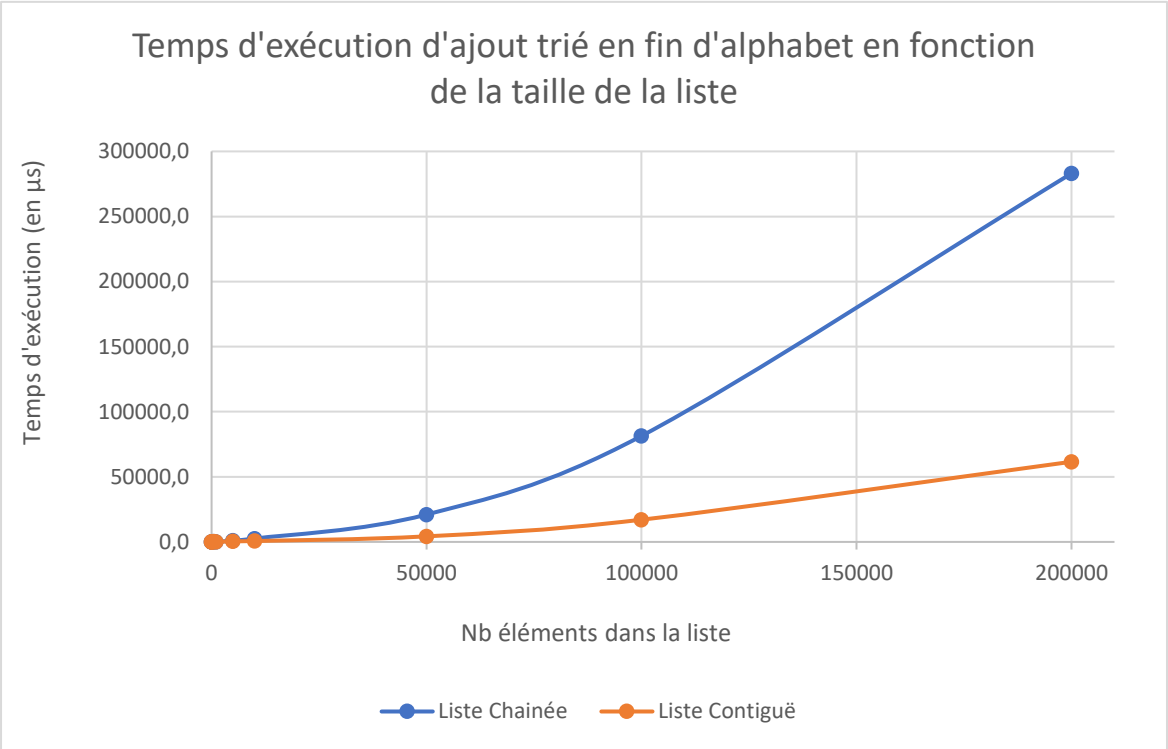
Après plusieurs mesures, nous avons fait une moyenne pour chaque action et chaque type de listes sur 100 répétitions. Nous obtenons les graphiques suivants :

Ajout trié de 10 éléments en début d'alphabet :



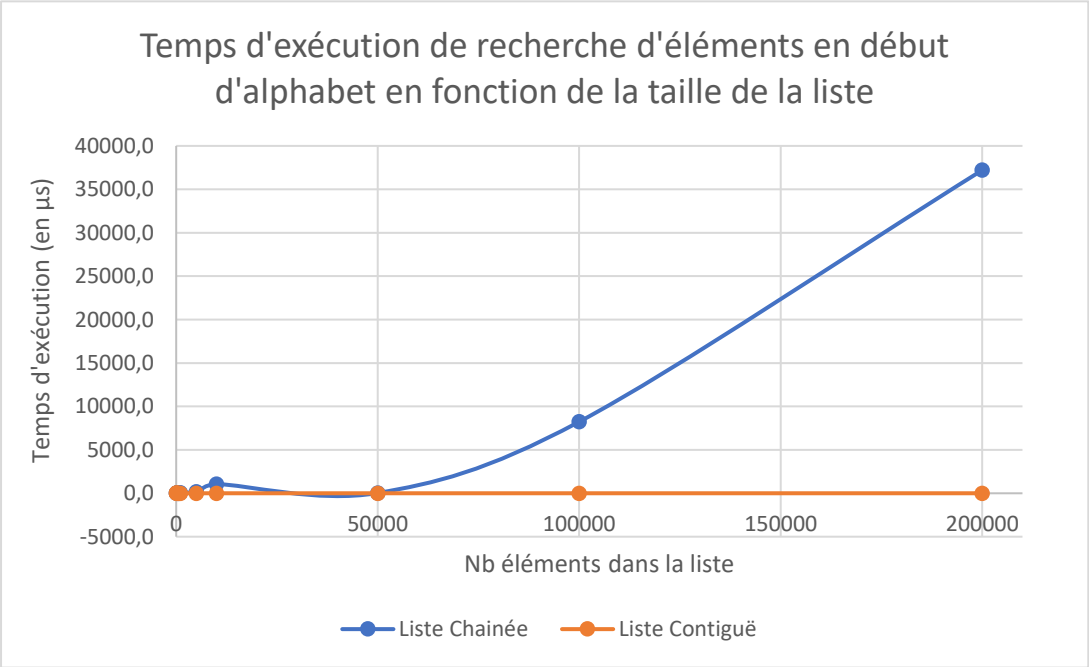
Sur ce graphique, nous remarquons que le temps d'exécution pour l'ajout trié de 1à éléments en début d'alphabet semble relativement similaire (bien qu'il y ai un écart de 30 µs dès le départ, en faveur de la liste chainée). Néanmoins, nous pouvons voir qu'à partir d'une taille de liste de 50 000 éléments, la liste contiguë prend de plus en plus de temps par rapport à la liste chainée, atteignant un temps presque 5 fois plus long pour une liste de 200 000 éléments.

Ajout trié de 10 éléments en fin d'alphabet :



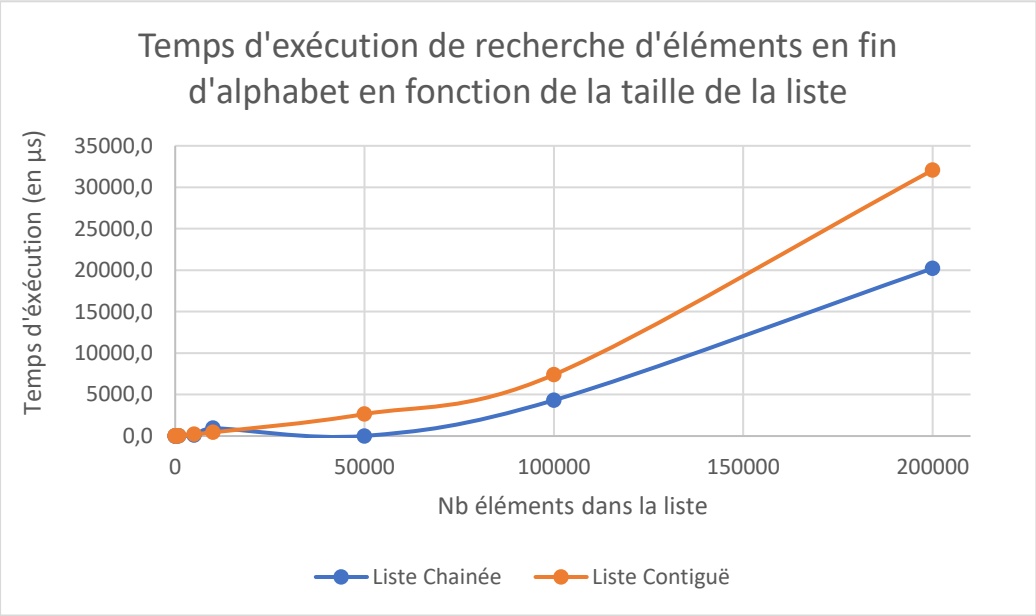
Lors de l'ajout trié de 10 éléments en fin d'alphabet, les temps d'exécution semblent encore relativement similaires pour les petites tailles de liste mais l'écart se creuse à une taille d'environ 1 000 éléments et les tendances s'inversent puisque cette fois, la liste contiguë semble plus rapide. Pour une liste de 200 000 éléments, la liste contiguë semble être près de 5 fois plus rapide.

Recherche de 10 éléments en début d'alphabet :



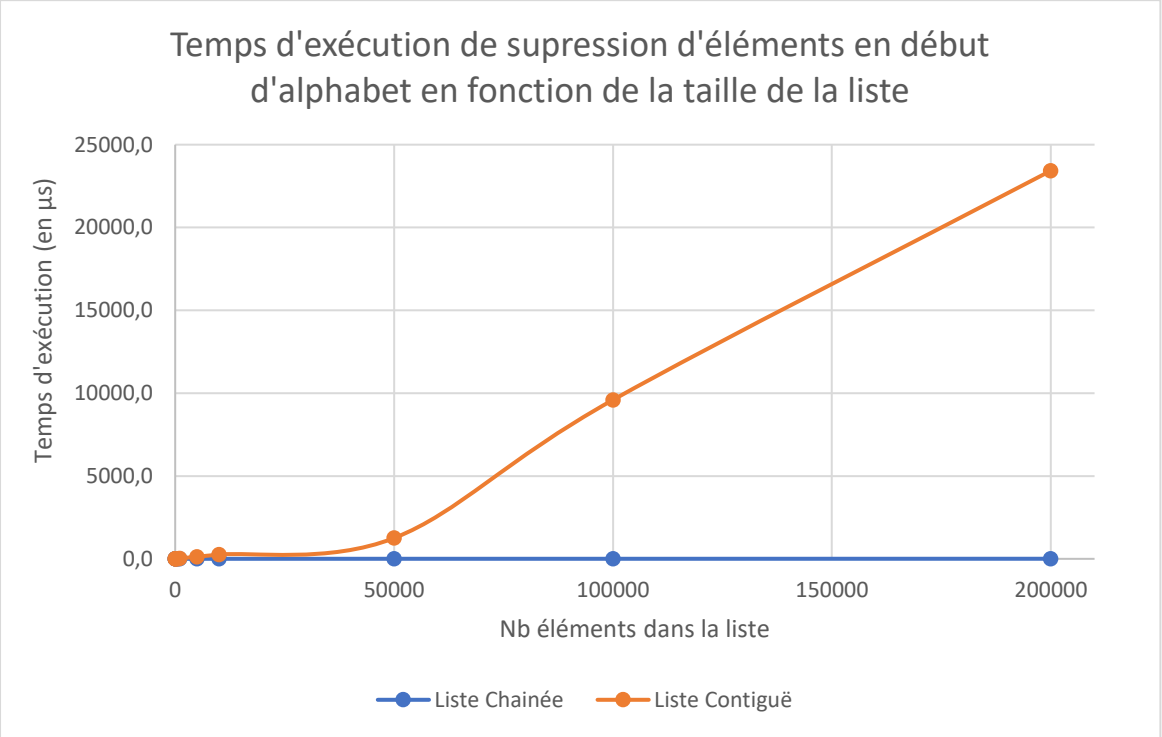
Ce graphique est très parlant car on voit tout de suite que la liste contiguë effectue une recherche d'élément en début d'alphabet très rapidement peu importe la taille de la liste. De l'autre côté la liste chainée prend de plus en plus de temps au fur et à mesure que sa taille augmente.

Recherche de 10 éléments en début d'alphabet :



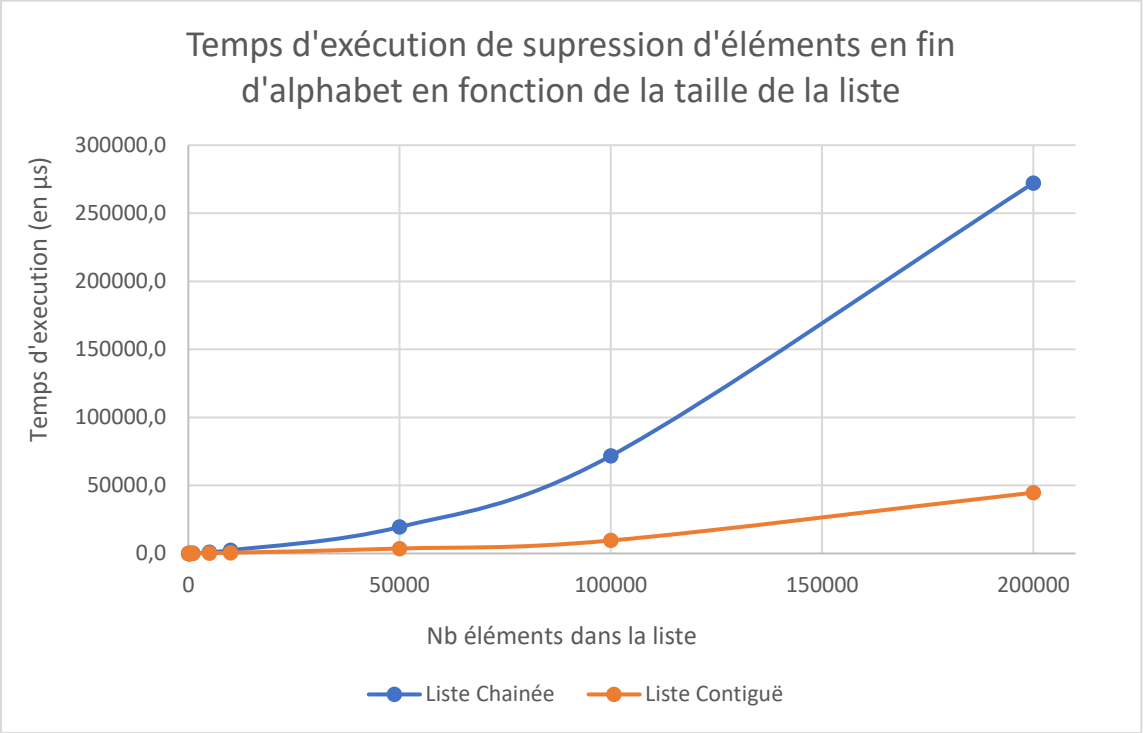
Ici, bien que la liste contiguë semble d'abord un peu plus rapide (bien que la différence ne soit pas énorme), une fois dépassé les 1 000 éléments, la liste contiguë prend de plus en plus de temps par rapport à la liste chainée. On remarque également que les deux listes augmentent leur temps d'exécution en fonction de leur taille de manière similaire bien que relative à leur structure. La liste chainée semble donc plus rapide que la liste contiguë pour la recherche d'élément en fin d'alphabet.

Suppression de 10 éléments en début d'alphabet :



On remarque sur ce graphique que la liste chainée effectue la suppression très rapidement. La liste contiguë aussi jusqu'à une liste de 50 000 éléments où son temps d'exécution commence à grandement augmenter. Ainsi, pour une liste de 200 000 éléments, l'écart de temps est colossal. La liste chainée semble clairement être la plus rapide pour cette action.

Suppression de 10 éléments en fin d'alphabet :



Pour ce dernier cas, on remarque encore une fois que les tendances s'inversent. En effet, le temps d'exécution sur une liste chainée augmente beaucoup plus rapidement que celui de la liste contiguë. Ainsi, pour une liste de 200 000 éléments, la liste contiguë est plus de 5 fois plus rapide que la liste chainée.

Conclusion de l'expérience :

Ajout trié en début d'alphabet : Les deux implémentations sont possibles pour des petites listes. Néanmoins, il est plus pertinent de passer une liste chainée, qui est plus rapide que la liste contiguë, peu importe la taille de la liste.

Ajout trié en fin d'alphabet : La liste chainée est plus rapide jusqu'à une taille de 100 éléments (selon les données recueillies). Au-dessus, la liste contiguë devient plus pertinente d'utilisation car plus rapide.

Recherche d'élément en début d'alphabet : Les résultats de l'expérience montre que la liste contiguë est le meilleur choix pour une recherche d'élément en début d'alphabet. De l'autre côté, le choix de la liste chainée reste possible pour des petites listes.

Recherche d'élément en fin d'alphabet : Bien que le choix d'une liste contiguë reste possible pour de petite liste, le meilleur choix semble être la chainée qui certes prend plus de temps en fonction de la taille de la liste, mais reste plus rapide que la liste contiguë

Suppression d'élément en début d'alphabet : La liste chainée semble être le meilleur choix au vu du graphique ayant une vitesse d'exécution nettement plus rapide que la liste contiguë. Cependant, le choix de la liste contiguë reste possible sur de petites listes.

Suppression d'élément en fin d'alphabet : La tendance s'inverse, la liste contigüe semble effectuée la suppression en fin de liste plus rapidement et semble être le meilleur choix au vu de sa vitesse d'exécution rapide.

Ainsi, si nous avons besoin de faire des modifications rapidement et régulièrement sans avoir à se soucier de l'espace disponible, il vaut mieux partir sur une liste chainée. Si peu de modifications sont à prévoir et/ou que l'espace disponible est limité, il est préférable d'utiliser une liste contiguë.

Informations supplémentaires :

Il peut y avoir des erreurs présentes dans les calculs ou dans les algorithmes ce qui peut légèrement fausser les résultats. De plus, ces résultats ne sont le résultat que d'une seule machine et ne peuvent être retrouvé que sur une machine similaire. L'autre machine utilisée pour les mesures de temps (qui ne sont pas exposées ici) retourne des résultats différents mais avec une conclusion similaire bien qu'elle n'ai pas pu dépasser une taille de liste de 50 000. Le temps d'exécution dépend donc, entre autres, de la configuration de la machine utilisée.