S.A.É. S1.02

GRAFF Mathieu
LOUNICI Ilyès

Le but de cette SAE est de comparer l'efficacité de différentes implémentations de listes en termes de temps d'exécution et de consommation de mémoire On va donc tester 3 types d'implémentation de listes et les comparer afin de trouver les avantages et les inconvénients de chacune d'entre elles.

Les tests seront effectués sur un ordinateur portable ayant un processeur Intel i5-7300HQ 2.500GHz, 12GB de RAM, avec Arch Linux comme système d'exploitation.

Dans le cadre de notre exploration des structures de données, particulièrement des listes, nous avons développé une fonction d'ajout qui insère de manière ordonnée une chaîne de caractères dans la liste. Cette fonction se nomme adjlisT et son programme en algorithme est présenté comme ci-contre :

```
fonction adjlisT(chaine: Chaîne)
début
   pos ← tete()
   prec ← pos
   Si finliste(tete()) OU (prec = tete() ET val(pos).compareTo(chaine) > 0) Alors
         adjtlis(chaine)
   Sinon
      Tant Que Non finliste(pos) ET val(pos).compareTo(chaine) < 0 Faire
         prec ← pos
         pos ← suc(pos)
      Fin Tant Que
      adjlis(prec, chaine)
   Fin Si
Fin
```

Lexique : pos = entier naturel qui indique la position dans la liste
prec = entier naturel qui indique la position précédant la position actuelle.

Au-delà de adjilisT, nous avons également conçu des opérations de suppression.
L'algorithme suivant illustre la méthode de suppression utilisée dans notre implémentation de la liste

```
Fonction suplisT(chaine: Chaîne)
   pos ← 0
   trouve ← faux

   Si Non finliste(tete()) Alors
      Tant Que Non finliste(pos) Faire
         Si val(pos).equals(chaine) ET Non trouve Alors
```

```
            suplis(pos)
            trouve ← vrai
          Fin Si
          pos ← suc(pos)
        Fin Tant Que
    Fin Si
Fin
```

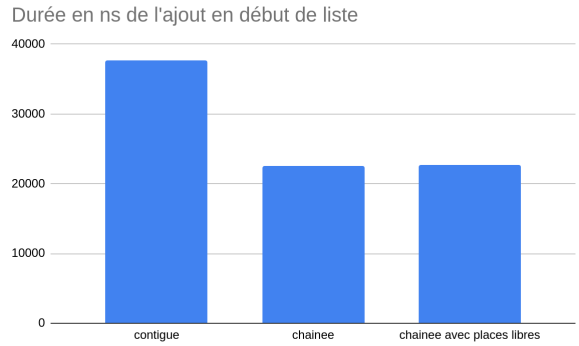
Lexique : pos = entier naturel qui indique la position dans la liste
trouve = booléen qui arrête la boucle tant que si il trouve la chaine à supprimer en question
Enfin, notre étude sur les listes inclut aussi la conception d'une fonction de recherche qui vérifie l'existence d'une chaîne de caractères dans la liste. C'est ce à quoi l'algorithme memlisT à été implémenté.

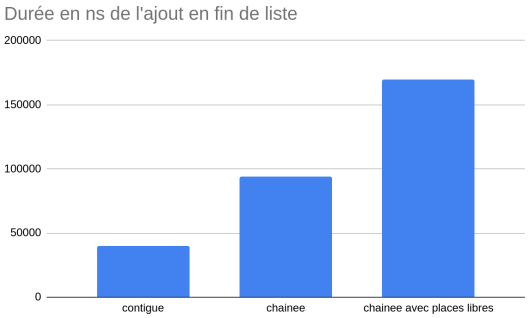
```
Fonction memlisT(liste, chaine)
Début
  courant ← tete(liste)
  Tant Que Non finliste(courant) Faire
    Si val(courant).equals(chaine) Alors
      Retourner vrai
    Sinon Si val(courant).compareTo(chaine) > 0 Alors
      Retourner faux
    Fin Si
    courant ← suc(courant)
  Fin Tant Que
  Retourner faux
Fin
```

Question 9)
La méthode suplisT parcourra l'entièreté de la liste sans trouver aucune occurrence. Ainsi, il est intéressant de répéter cette opération plusieurs fois pour savoir laquelle de ses listes est parcourue le plus vite par la méthode.

Après avoir effectué des tests d'insertion et de suppression les trois types de listes avec un volume de données croissant, nous avons mesuré le temps d'exécution et l'utilisation de la mémoire pour chaque type de liste. Voici les résultats graphique pour le temps d'insertion :

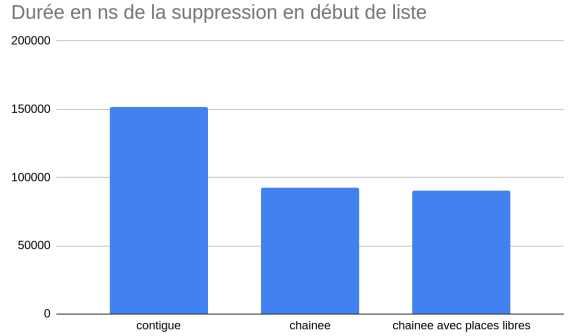
Ajouts en début de liste :

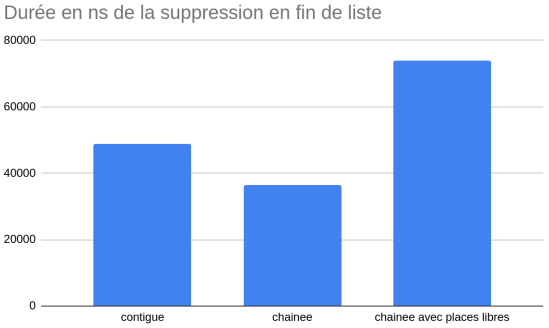



On observe donc que la liste chaînée et la liste chaînée avec places libres montrent de meilleures performances en temps d'insertion lorsque les ajouts se font en début de liste. D'un autre côté, la liste contiguë est plus rapide pour les ajouts en fin de liste.

Voyons maintenant le graphique pour la suppression :




Il semble que la liste chaînée soit plus performante que ses semblables pour supprimer au début et à la fin d'une liste.

Pour conclure sur l'efficacité des implémentations des différentes listes, il semblerait que les listes chaînées soient plus efficaces en moyenne dans tous les domaines tandis que les listes contiguës sont surtout efficaces en suppression en fin de liste et les listes chaînées avec places libres ne sont jamais les plus rapides car elles nécessitent plus d'opérations.