BUT informatique – S.A.É. S1.02

Comparaison d'approches algorithmiques

Étienne ANDRÉ

www.loria.science/andre/enseignement/R101c/

Version : 10 décembre 2021

Table des matières

Partie 1

Présentation et rendu attendu

1.1 Contexte

On considère des listes de noms de famille (**chaîne**), triées par ordre alphabétique (croissant). Par exemple :

```
['Al-Kindi','Lovelace','Turing','Yao']
```

Ces listes seront de taille « relativement importante » (on pourra considérer en pratique 10 000 éléments).

1.2 Objectif principal

L'objectif de la SAÉ est de comparer expérimentalement l'efficacité de deux types d'implémentation de listes triées de chaînes, notamment de grande taille, et ce pour les opérations d'ajout, de suppression et de test d'appartenance. Ces deux implémentations de listes seront :

1. représentation contiguë dans un tableau ;
2. représentation chaînée dans un tableau avec récupération simple des places libérées.

Objectifs

Cette SAÉ a pour objectifs de contribuer aux apprentissages critiques suivants :

AC12.01 Analyser un problème avec méthode (découpage en éléments algorithmiques simples, structure de données…)

— écrire des algorithmes de recherche, d'insertion et de suppression dans des listes triées

AC12.02 Comparer des algorithmes pour des problèmes classiques (tris simples, recherche…)

— comparer les performances des implémentations contiguës et chaînées des listes sur des opérations simples (ajout, suppression, recherche…)

1.3 Rendus

Le travail est à faire en binôme.

1.3.1 Travail à rendre

1. votre code intégral (sous forme d'archive `.zip` ou `.tar.gz`, avec uniquement les fichiers source, et un fichier `LISEZMOI.md` indiquant votre classe principale, ou un `Makefile`; aucun fichier inutile (`.class`, etc.) ne doit être inclus)
2. un rapport en PDF contenant
   (a) les trois algorithmes logiques (exercice 3.1 du poly de R101-c);
   (b) la description explicite des questions optionnelles que vous avez traitées, et comment vous les avez traitées (le cas échéant);
   (c) la description de l'environnement expérimental (machine, manière de calculer les différentes opérations...);
   (d) la description des expériences réalisées;
   (e) la conclusion sur les différents types de structures de données;
   (f) toute autre information jugée utile.

   Ne pas hésiter à intégrer des rendus graphiques (courbes, etc.) le cas échéant.

Remarque importante 1

Il vous est fortement conseillé de démarrer par réaliser l'ensemble des questions obligatoires, et seulement dans un second temps de passer aux questions optionnelles (le cas échéant).

Traiter partiellement ou entièrement les parties optionnelles donnera bien entendu lieu à une attribution de points supplémentaires.

1.4 Documents fournis par l'équipe pédagogique sur Arche

Les fichiers Java suivants vous sont fournis sur Arche :

1. le squelette de la classe `ListeTriee`;
2. un squelette vide pour la classe principale (classe `Principale`, contenant notamment le `main` à lancer);
3. la bibliothèque de test `libtest`;
4. la classe `LectureFichier`;
5. la classe `EcritureFichier` (pour la question optionnelle de création de votre propre liste);
6. divers fichiers de taille variable contenant des listes de noms de famille (qui seront interprétés comme des chaînes de caractères).

Vous pourrez par ailleurs réutiliser :

— vos implémentations des listes chaînées et contiguës programmées en TP.

Partie 2

Implémentation des listes triées

2.1 Implémentation des listes de chaînes

Question 1 : Incorporer dans votre projet l'interface `Liste` (vue au TP 1) ainsi que vos implémentations de listes chaînées (classe `ListeChainee`) et contiguës (classe `ListeContigue`), réalisées au cours des TP de R101-c. Ces deux classes implémentent l'interface `Liste`.

Question 2 : Modifier ces classes pour qu'elles deviennent des listes de chaînes de caractères (au lieu de listes de sous-titres). L'interface `Liste` devra également être modifiée.

2.2 Implémentation des listes triées

L'objectif est désormais d'implémenter une classe pour les listes triées. L'idée retenue sera d'avoir une *unique* classe, prenant comme argument du constructeur une liste vide, du bon type (chaînée ou contiguë selon le besoin). Cet argument sera un attribut (privé) de la classe :

```
private Liste liste;
```

Ainsi, la classe de liste triée est *générique*, et va appeler sur l'objet passé au constructeur les *algorithmes logiques* vus en TD, sur les listes triées.

Question 3 : Intégrer le squelette de code suivant :

```
public class ListeTriee{

  // Attribut de liste sous-jacente
  private Liste liste;

  public ListeTriee(Liste listevide){
    // Affectation de la liste vide a l'attribut prive
    liste = listevide;
  }

  /**
   * ajoute un element au bon endroit dans la liste triee
   * @param chaine element a inserer
   */
  public void adjlisT(String chaine){
    throw (new Error("not implemented"));
  }

  /**
```

```
20      * permet de supprimer un element d'une liste. Supprime le premier
         element dont la valeur est egale a "chaine" ; ne fait rien si
         "chaine" n'appartient pas a la liste.
21      * @param chaine l'element a supprimer
22      */
23     public void suplisT(String chaine){
24       throw (new Error("not implemented"));
25     }
26
27     /**
28      * Retourne vrai si au moins un element de la liste a une valeur egale
         a "chaine", et retourne faux sinon.
29      * @param chaine l'element que l'on recherche
30      */
31     public boolean memlisT(String chaine){
32       throw (new Error("not implemented"));
33     }
34
35     public String toString(){
36       // TODO (utiliser les fonctions deja ecrites dans les listes !)
37       throw (new Error("not implemented"));
38     }
39 }
```

Question 4 : Implémenter la fonction `adjlisT` (dont l'algorithme a normalement été vu en TD, question 1 de l'exercice 3.1).

Question 5 : Écrire l'algorithme logique de la fonction de suppression d'un élément dans une liste triée. Cet algorithme supprime le premier élément dont la valeur est égale à la chaîne passée en paramètre ; il ne fait rien si la chaîne passée en paramètre n'appartient pas à la liste.

Question 6 : Implémenter la méthode correspondante `suplisT`.

Question (optionnelle) 7 : Écrire l'algorithme logique de la fonction de test d'appartenance d'un élément à une liste triée. Cette fonction retourne vrai si au moins un élément de la liste a une valeur égale à la chaîne passée en paramètre, et retourne faux sinon.

Question (optionnelle) 8 : Implémenter la méthode correspondante `memlisT`.

2.3 Tests élémentaires

Tester vos méthodes.

Question 9 : Ajouter 4 éléments qui ne sont *pas* dans l'ordre alphabétique, puis vérifier que la liste est bien triée (grâce à la méthode `toString()` de `ListeTriee`). Le faire pour chacune des deux représentations (chaînée et contiguë).

Question 10 : Vérifier que la suppression d'un élément de la liste est correcte, c'est-à-dire que la suppression d'un élément de la liste donne une liste où l'élément a bien été enlevé.

Question 11 : Vérifier que la suppression d'un élément qui n'appartient pas à la liste garde la liste inchangée.

Remarque 1

Utiliser le squelette de classe `TestListeTriee.java` pour les questions de tests ci-dessus.

Question (optionnelle) 12 : Proposer d'autres tests. Les expliquer dans votre rapport.

2.4 Création d'une liste à partir d'un fichier

Question 13 : Mettre en place la création d'une liste à partir de l'un des fichiers fournis (par exemple `noms10000.txt`). Utiliser la classe Java `LectureFichier.java` à cet effet.

Question (optionnelle) 14 : Proposer votre propre méthode de création d'une liste de 10 000 éléments, en expliquant la manière dont vous avez procédé ; il peut s'agir par exemple de générer de façon aléatoire des chaînes de caractères, ou récupérer la base des 10 000 noms de famille les plus courants en France (ou ailleurs), etc.

Remarque 2

Si vous choisissez de traiter la question ci-dessus, il est possible (quoique non obligatoire) d'utiliser la classe `EcritureFichier.java` fournie sur Arche.

Remarque 3

Attention ! (à propos de la question optionnelle ci-dessus) Si vous utilisez de l'aléatoire, celui-ci ne devra pas faire partie de votre code Java (afin que les expériences soient déterministes). En d'autres termes, vous pouvez générer aléatoirement du code Java (à partir d'un autre langage, par exemple Python… ou Java !), mais *pas* utiliser dans le code Java de cette SAÉ des fonctions `Random`.

Question (optionnelle) 15 : Paramétrer la taille de la liste en fonction d'un argument passé comme paramètre de votre programme. Par exemple, l'appel pourrait être :

```
java MonProgramme 1000 # pour une liste de taille 1000
```

Partie 3

Comparaison des performances

3.1 Mesurer le temps en Java

Il est possible de mesurer le temps d'exécution d'un morceau de code Java de la façon suivante :

```java
// Debut chronometre
long date_debut = System.nanoTime();

// ici une action dont on mesure le temps

// Fin chronometre
long date_fin = System.nanoTime();
long duree = date_fin - date_debut;
```

3.2 Mesures de temps d'exécution

Question 16 : Implémenter dans votre classe principale (ou dans toute autre classe de votre choix) une méthode permettant de mesurer le temps d'exécution de l'action ajoutant 10 chaînes de caractères en début d'alphabet. Cette mesure devra être faite pour chacune des deux implémentations (listes chaînées et listes contiguës).

> **Remarque importante 2**
>
> Les temps de *création* de la liste ne doivent pas être inclus dans le chronomètre. En outre, la liste doit être recréée à *chaque* ensemble d'opérations.

Le travail demandé ressemble donc au pseudo-code suivant :

```
   /* On suppose un tableau de chaînes remplissage de taille 10 000    */
   /* On suppose un tableau de chaînes à ajouter ajout de taille 10    */
1  ℓ ← lisvide()
2  Ajouter à ℓ les 10 000 élements de remplissage
3  Lancer le chronomètre
4  pour j de 1 à 10 faire
5  |  ℓ ← adjlisT(ℓ, ajout[j])
6  fin pour
7  Arrêter le chronomètre
```

Question 17 : Même question avec des chaînes de fin d'alphabet.

Pour les chaînes de début et de fin d'alphabet (dans ces questions et dans les suivantes), il est possible (mais non obligatoire) d'utiliser les tableaux de taille 10 suivants :

```
1 private static final String[] ELEMENTS_DE_DEBUT    = {"A", "AA", "AAA",
      "AAAA", "AAAAA", "AAAAAA", "AAAAAAA", "AAAAAAAA", "AAAAAAAAA",
      "AAAAAAAAAA"};
2
3 private static final String[] ELEMENTS_DE_FIN    = {"RABIN", "RIVEST",
      "SHAMIR", "SIFAKIS", "TORVALDS", "TURING", "ULLMAN", "VALIANT",
      "WIRTH", "YAO"};
```

Question 18 : Implémenter du code permettant de mesurer le temps d'exécution de l'action supprimant 10 chaînes de caractères en début d'alphabet. Cette mesure devra être faite pour chacune des deux implémentations (listes chaînées et listes contiguës).

> **Remarque importante 3**
>
> Pour les chaînes de caractères en début d'alphabet, il est intéressant de supprimer des chaînes faisant *effectivement* partie de la liste. En effet, la suppression d'une chaîne qui ne fait *pas* partie de la liste n'entraînera aucune modification de la liste (et notamment aucun décalage dans l'implémentation contiguë).

Question 19 : Même question avec des chaînes de fin d'alphabet.

Question (optionnelle) 20 : Implémenter du code permettant de mesurer le temps d'exécution de l'action testant l'appartenance à une liste de 10 chaînes de caractères en début d'alphabet. Cette mesure devra être faite pour chacune des deux implémentations (listes chaînées et listes contiguës).

Question (optionnelle) 21 : Même question avec des chaînes de fin d'alphabet.

> **Remarque importante 4**
>
> Un effort de *factorisation* est attendu ! Une abondance de code copié-collé pour les questions précédentes n'est en aucun cas souhaitable.

3.3 Optionnel : amélioration des mesures

Mesurer un unique temps d'exécution peut être biaisé par de nombreux facteurs.

Question (optionnelle) 22 : Améliorer votre code pour que la mesure soit une *moyenne* de 100 exécutions successives.

Le travail demandé peut donc ressembler au pseudo-code suivant :

```
   /* On suppose un tableau de chaînes remplissage de taille 10 000     */
   /* On suppose un tableau de chaînes à ajouter ajout de taille 10     */
 1 pour i de 1 à 100 faire
 2 |  ℓ ← lisvide()
 3 |  Ajouter à ℓ les 10 000 élements de remplissage
 4 |  Lancer le chronomètre
 5 |  pour j de 1 à 10 faire
 6 |  |  ℓ ← adjlisT(ℓ, ajout[j])
 7 |  fin pour
 8 |  Arrêter le chronomètre
 9 fin pour
10 Calculer la durée moyenne
```

Question (optionnelle) 23 : Améliorer votre code pour que le nombre de tests consécutifs soit donné par un argument de votre programme.

Un exemple d'appel pourrait être :

```
java MonProgramme 10000 100 # pour une liste de taille 10000, où
   chaque expérience est répétée 100 fois
```

Question (optionnelle) 24 : Améliorer votre code pour que l'on mesure non seulement le temps d'exécution, mais aussi le nombre d'*opérations* en lecture et en écriture. Une opération en lecture peut être d'accéder au successeur d'un élément, ou à une case de tableau. Une opération en écriture peut être la modification d'une variable, ou la modification d'une case dans un tableau. Ajoutez à vos résultats ces nombres d'opérations. Bien préciser quelles opérations vous avez comptées.

3.4 Optionnel : variation de la taille des listes

Nous proposons ici d'étudier l'influence de la taille de la liste (par défaut 10 000 éléments, mais que l'on va justement faire varier) sur le temps de calcul, pour chacune des deux représentations.

Question (optionnelle) 25 : Effectuer plusieurs calculs de temps d'exécution pour plusieurs tailles de listes (par exemple 10, 50, 100, 500, 1 000, 5 000, 10 000, 50 000, 100 000...). On peut représenter le résultat de cette expérience par un graphique avec, en abscisse, le nombre d'éléments de la liste et, en ordonnée, le temps d'exécution d'ajout de 10 éléments en début de cette liste. Comparer les graphiques obtenus pour la représentation chaînée et la représentation contiguë.

Question (optionnelle) 26 : Même question pour les 5 autres expériences, à savoir ajout en fin de liste, suppression et appartenance en début et fin de liste.

3.5 Optionnel : implémentation de la gestion de l'espace libre

La gestion des listes chaînées est d'autant plus efficace qu'une gestion efficace des places libres est employée. Dans les TP de R101-c, nous n'avons pas implémenté de gestion de l'espace libre autre qu'une méthode indiquant une valeur spéciale (-2) dans le champ *suc* : pour

trouver une place libre, il faut parcourir toutes les places jusqu'à en trouver une de libre (voir TP 3).

 **Question (optionnelle) 27 :** Implémenter une gestion de l'espace libre pour vos listes chaînées, par exemple à l'aide d'une liste de places libres (cf. cours de R101-c).

Comparer l'efficacité des listes chaînées (pour les opérations vues ci-dessus) avec ou sans cette gestion améliorée des places libres.

3.6 Conclusions

 **Question 28 :** Proposer un récapitulatif clairement visible (par exemple sous forme de tableau ou graphique) des temps d'exécution mesurés. Proposer des conclusions sur les deux implémentations étudiées.