



BUT informatique – S.A.É. S1.02

Comparaison d'approches algorithmiques

Étienne André, Sergueï Lenglet



Version : 9 janvier 2023

Table des matières

1	Présentation et rendu attendu	3
2	Listes chaînées avec liste de places libres	5
3	Listes triées	6
4	Comparaison des performances	8

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 SAE est de comparer expérimentalement l'efficacité de trois types d'implémentation de listes triées de chaînes, notamment de grande taille, et ce pour les opérations d'ajout et de suppression. Ces implémentations de listes seront :

1. représentation contiguë dans un tableau;
2. représentation chaînée dans un tableau avec récupération des places libérées (partie 2.3.2.5 du polycopié);
3. représentation chaînée dans un tableau avec gestion de l'espace libre à l'aide d'une liste ("liste libre" - partie 2.3.2.6 du polycopié).

Objectifs

Cette SAE 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.

Travail à rendre

1. Votre code intégral sous forme d'archive S1X_Nom1_Nom2.zip, avec uniquement les fichiers source; aucun fichier inutile (.class, etc.) ne doit être inclus.
2. un rapport en PDF contenant
 - (a) une introduction rappelant l'objectif de la SAE et décrivant l'environnement expérimental : processeur, mémoire et système d'exploitation de la machine utilisée;
 - (b) une partie donnant les algorithmes logiques (exercice 4.1 du poly de R101-c);
 - (c) une partie décrivant les expériences réalisées et leurs résultats;
 - (d) une conclusion sur l'efficacité des implémentations des listes considérées dans cette SAE;
 - (e) toute autre information jugée utile.

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

1.4 Documents fournis par l'équipe pédagogique sur Arche

Les fichiers Java suivants vous sont fournis sur le cours en ligne :

1. les fichiers des implémentations contiguës et chaînées vues en TP (Liste, ListeContigue, ListeChaine et MaillonChaine);
2. la classe ListeChainePlacesLibres partiellement complétée, avec sa classe de tests TestListeChainePlacesLibres;
3. le squelette de la classe ListeTrie;
4. la classe de test partielle TestListeTrie;
5. le squelette de la classe Principale, contenant notamment le main à lancer;
6. la bibliothèque de test libtest;
7. les classes LectureFichier et la classe EcritureFichier;
8. divers fichiers de taille variable contenant des listes de noms de famille (qui seront interprétés comme des chaînes de caractères).

Partie 2

Listes chaînées avec liste de places libres

L'implémentation des listes chaînées dans laquelle les places libres sont maintenues dans une liste est décrite dans le polycopié de cours (partie 2.3.2.6). Dans cette implémentation, le tableau contient deux listes représentées de manière chaînée :

- la liste des éléments à stocker;
- la liste des places libres (“liste libre”).

Les opérations d'ajout et de suppression mettent à jour simultanément les deux listes : voir le polycopié de cours pour plus de détails.

Le code fourni sur le cours en ligne contient une classe `ListeChainePlacesLibres` reprenant les méthodes écrites pour l'implémentation de la liste chaînée (TP 3). Les méthodes de gestion de l'espace libre sont à compléter. Un attribut privé `teteLibre` a été ajouté pour indiquer la tête de la liste libre.



Question 1 : Compléter le constructeur de la classe `ListeChainePlacesLibres` ainsi que les méthodes `libererPlace` et `retournerPlaceLibre`. Tester votre classe avec la classe `TestListeChainePlacesLibres` fournie.

Partie 3

Listes triées

3.1 Principe et méthodes élémentaires


L'objectif est 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.

Le squelette de la classe `ListeTrie` est disponible sur le cours en ligne, ainsi que les implémentations des listes contiguës (`ListeContigue.java`) et chaînées (`ListeChaine`.java et `MaillonChaine.java`) qui reprennent ce qui a été fait dans les TP 2 et 3.

Les méthodes suivantes permettent d'écrire plus facilement les tests sur les listes triées.


 **Question 2 :** Implémenter les méthodes suivantes de la classe `ListeTrie` : `tete`, `val`, `suc` et `finListe`.

Remarque 1


Chaque méthode de la question ci-dessus s'écrit en une seule ligne.

3.2 Implémentation et tests

Les questions suivantes demandent d'implémenter les méthodes correspondantes aux fonctions d'ajout et de suppression vues en TD (exercice 4.1). Une classe de tests partielle `TestListeTrie` est fournie sur le cours en ligne : les tests pour certaines sont à écrire.

 **Question 3 :** Implémenter la méthode `adjlist`. Utiliser les tests fournis pour tester votre méthode.

 **Question 4 :** Implémenter la méthode `suplist`.

 **Question 5 :** En vous inspirant des tests pour `adjlist`, écrire les tests pour `suplist`, et tester votre méthode.

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



Question 6 : Dans la classe Principale, 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` à cet effet.

Partie 4

Comparaison des performances

4.1 Principe

L'objectif est de mesurer le temps pris en répétant 10 opérations faites sur une liste triée, en faisant varier plusieurs paramètres :

- le type d'opération (ajout ou suppression) ;
- où l'opération a lieu (en début ou fin de liste) ;
- l'implémentation de la liste (contiguë ou chaînée, avec ou sans liste libre).

Pour chaque mesure, une nouvelle liste est créée, et on mesure le temps pris par 10 opérations successives. En supposant que la liste triée contient initialement 10 000 chaînes, le travail demandé ressemble donc au pseudo-code suivant :

```
/* On suppose un tableau de chaînes remplissage de taille 10 000 */
1  $\ell \leftarrow \text{lisvide}()$ 
2 Ajouter à  $\ell$  les 10000 éléments de remplissage
3 Lancer le chronomètre
4 pour  $j$  de 1 à 10 faire
  | /* Opération d'ajout ou de suppression
5 fin pour
6 Arrêter le chronomètre
```

Remarque importante 1

Le temps de *création* de la liste n'est pas inclus dans le chronomètre.

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


```
1 // Debut chronometre
2 long date_debut = System.nanoTime();
3
4 // ici une action dont on mesure le temps
5
6 // Fin chronometre
7 long date_fin = System.nanoTime();
8 long duree = date_fin - date_debut;
```



4.2 Mesures de temps d'exécution

Dans cette partie, la liste triée est construite à partir du fichier de 10 000 noms fourni sur le cours en ligne.

Remarque importante 2


Pour les questions de cette partie, le copié-collé doit être réduit au minimum. La factorisation du code sera évaluée.

 **Question 7 :** Implémenter dans votre classe principale du code permettant de mesurer le temps d'exécution de la méthode `adjlist` avec 10 chaînes de caractères en début d'alphabet. Cette mesure devra être faite pour chacune des trois implémentations (listes chaînées avec et sans liste libre, et listes contiguës).


 **Question 8 :** Même question avec des chaînes de fin d'alphabet.


Remarque importante 3

Pour la question suivante, aucun code n'est attendu, seulement une réponse dans le rapport.

 **Question 9 :** Que se passe-t-il dans le cas d'un appel à `suplist` avec une chaîne qui n'appartient pas à la liste? Est-il intéressant de répéter cette opération 10 fois pour une implémentation de liste donnée? Justifier votre réponse.

Pour mesurer le temps d'exécution de `suplist` (questions suivantes), il est possible (mais non obligatoire) d'utiliser les tableaux de début et fin d'alphabet définis dans le fichier `Principale.java` fourni sur le cours en ligne.

 **Question 10 :** Implémenter dans votre classe principale du code permettant de mesurer le temps d'exécution de la méthode `suplist` avec 10 chaînes de caractères **appartenant à la liste** en début d'alphabet. Cette mesure devra être faite pour chacune des trois implémentations.

 **Question 11 :** Même question avec des chaînes de fin d'alphabet.

Remarque importante 4

Pour factoriser le code entre les questions 7, 8, 10 et 11, on pourra écrire une fonction qui exécute le test en prenant en paramètres ce qui varie entre les différentes questions (le type de liste, le type d'opération, les chaînes à ajouter ou supprimer).

4.3 Écriture des résultats dans un fichier


Écrire les temps d'exécution dans un fichier lisible par un tableur (comme LibreOffice ou Excel) permet de générer plus facilement un tableau ou un graphique. Le format le plus simple à utiliser est le `csv`, dans lequel les données sont séparés par un symbole, comme par

exemple le point-virgule. Le fichier généré peut ressembler au format suivant (les valeurs données ne sont pas forcément cohérentes) :

```


1  liste;operation;emplacement;duree(ns)
2  contigue;ajout;debut;106474
3  chaine;ajout;debut;65096
4  contigue;ajout;fin;34706
5  chaine;ajout;fin;397279
6  contigue;suppression;debut;9999
7  chaine;suppression;debut;14361
8  ...

```

 **Question 12 :** Modifier votre code pour que les mesures soient écrites dans un fichier `resultats.csv`. Utiliser la classe `EcritureFichier` fournie sur le cours en ligne.

4.4 Amélioration des mesures

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

 **Question 13 :** Modifier la classe `Principale` pour que les mesures soient des *moyennes* 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 */
1  pour i de 1 à 100 faire
2    ℓ ← lisvide()
3    Ajouter à ℓ les 10 000 éléments de remplissage
4    Lancer le chronomètre
5    pour j de 1 à 10 faire
6      /* Opération d'ajout ou de suppression */
7    fin pour
8  fin pour
9  Calculer la durée moyenne

```

4.5 Optionnel : variation de la taille des listes

Nous proposons ici d'étudier l'influence de la taille de la liste sur le temps de calcul pour chaque représentation.

 **Question (optionnelle) 14 :** Copier la classe `Principale` dans une nouvelle classe `PrincipaleVariationTaille`. Modifier `PrincipaleVariationTaille` pour que celle-ci effectue plusieurs calculs de temps d'exécution pour plusieurs tailles de listes (par exemple 10, 100, 1 000, 10 000, ...). 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 chaque représentation (conitguë, chaînée avec ou sans liste libre).

 **Question (optionnelle) 15 :** Même question pour les 3 autres expériences, à savoir ajout en fin de liste, suppression en début et fin de liste.

4.6 Conclusions



Question 16 : Proposer un récapitulatif clairement visible (par exemple sous forme de tableau ou graphique) des temps d'exécution mesurés. Proposer des conclusions sur les trois implémentations étudiées.