TP 3 : TDA et chainage.

1 INTRODUCTION

Pour le troisième TP, vous allez construire un type de données. Ce type de données est un contenant homogène trié à taille variable. Ce contenant va utiliser des chainages pour obtenir une taille variable. Afin d'accélérer la recherche dans le contenant, nous allons inclure un index qui permet de retrouver les éléments plus rapidement. Cette structure va avoir des méthodes (services) pour insérer, supprimer et consulter les éléments. Les techniques utilisées pour l'index vont nous permettre d'obtenir des temps $O(\sqrt{n})$, ce qui est meilleur que $O(n)$ mais moins performant que $O(\log n)$.

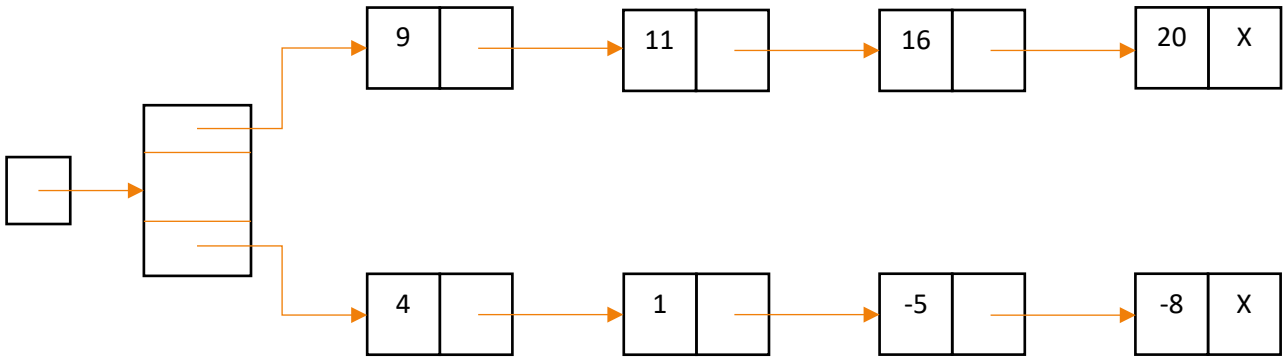
La section suivante décrit deux classes nécessaires pour la construction de cette structure. La section trois décrit des règles importantes à suivre pour ce TP et la technique qui sera utilisée pour tester votre programme. Finalement, la dernière section décrit les autres directives.

2 DESCRIPTION

Vous allez devoir construire deux structures afin de réaliser le TDA pour ce TP. La première structure `ListeMilieu` est une structure de liste chaînée dont le point d'attache est le milieu de la liste. La deuxième structure `ListeIndex` est une structure qui va gérer l'accès à plusieurs listes chaînées (`ListeMilieu`) à l'aide d'un index. Vous allez construire une classe correspondante à chacune de ces structures. Vous pouvez ajouter d'autres classes.

2.1 LISTEMILIEU

Cette structure est une liste chaînée dont les éléments seront maintenus triés. C'est-à-dire, que les éléments seront insérés au bon endroit dans la liste. Aussi, cette liste est accédée par le milieu.

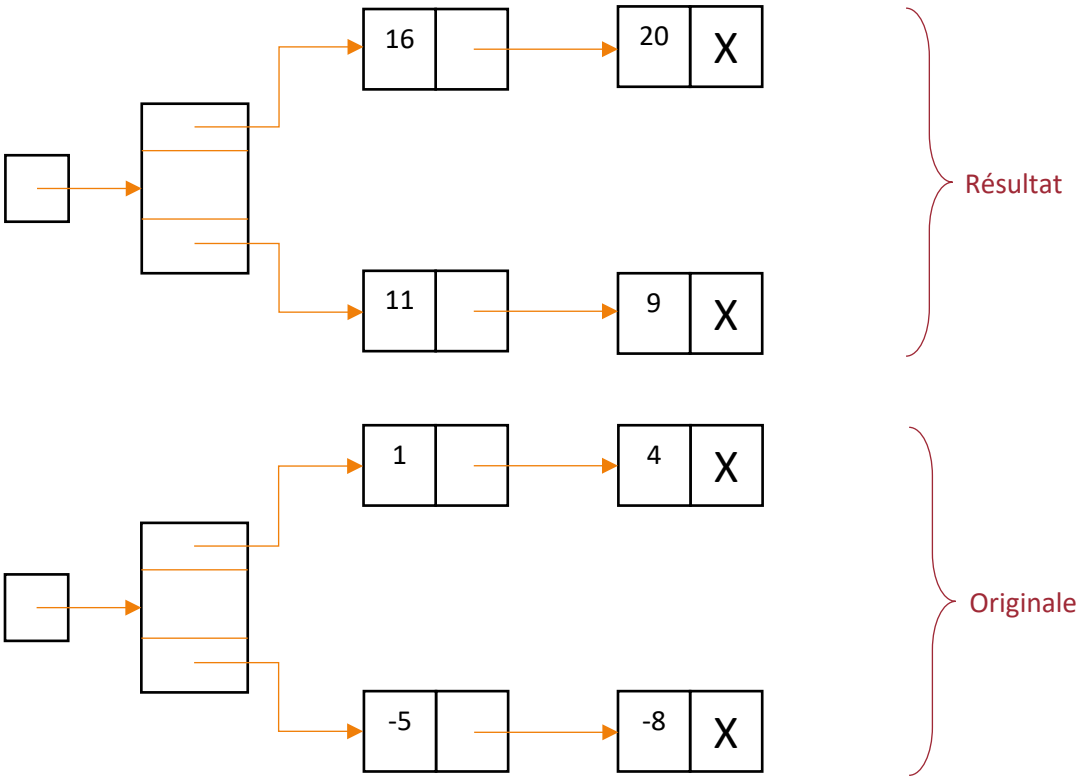


Le point milieu permet d'accéder aux valeurs inférieures (dans l'exemple : 4, 1, -5, -8) et aux valeurs supérieures (dans l'exemple : 9, 11, 16, 20). Il y a deux éléments importants dans cette structure.

1. Les éléments sont triés.
    a. Tous les éléments de la liste inférieure sont plus petits que tous les éléments de la liste supérieure.
    b. Dans la liste inférieure, les éléments sont du plus grand vers le plus petit.
    c. Dans la liste supérieure, les éléments sont du plus petit vers le plus grand.
2. Les tailles des deux listes (inférieure et supérieure) sont similaires. C'est-à-dire, la taille de la liste supérieure est égale à la taille de la liste inférieure **ou** la taille de la liste inférieure est un (1) de plus que la taille de la liste supérieure.

Vous devez vous assurer que c'est vrai en tout temps. Lors d'une insertion, vous devrez insérer l'élément au bon endroit dans la liste et ensuite rééquilibrer la liste en transférant des éléments, soit de la liste inférieure vers la liste supérieure, soit de la liste supérieure vers la liste inférieure.

Un service important offert par la `ListeMilieu` est de pouvoir se diviser en deux `ListeMilieu`s. Cette opération construit une `ListeMilieu` à partir de la liste supérieure et redistribue la liste inférieure dans la structure originale. Pour l'exemple plus haut, après division, cela nous donnerait la `ListeMilieu` résultante suivante avec la liste originale modifiée.



Ces listes vont aussi offrir un service pour consulter l'élément au milieu. L'élément au milieu est le premier élément de la liste inférieure. Il sera aussi possible d'obtenir le minima (dernier élément de la liste inférieure) et le maxima (dernier élément de la liste supérieure).

2.2 ListeIndex

La deuxième structure que vous devez construire va gérer une suite de liste (`ListeMilieu`). Cette structure construit une liste chaînée où chaque élément (e_i) contient une liste. Les éléments de cette liste seront en ordre de telle sorte que $e_i.maxima < e_{i+1}.minima$.

Mécanique d'indexage : il sera possible d'insérer des valeurs (v) dans la liste. Cette insertion devra trouver la liste (e_i) dans laquelle la valeur sera insérée : $e_i.minima \leq v \leq e_i.maxima$. Ensuite, le service d'insertion sera appelé sur la liste trouvée.

La liste d'index doit aussi gérer la taille des listes qu'elle contient. Pour décrire cette gestion, désignons par T le nombre de `ListeMilieu` indexé dans la `ListeIndex`. Chaque `ListeMilieu` (e_i) a une taille $e_i.taille$. Il est important que la taille d'une `ListeMilieu` ne dépasse pas le double du nombre de `ListeMilieu` dans la `ListeIndex` : $\forall i \in [0..T-1], e_i.taille \leq 2T$.

Donc, si une insertion dans une `ListeMilieu` rend une `ListeMilieu` plus grande que le double du nombre de `ListeMilieu` dans la `ListeIndex`, alors la `ListeMilieu` devra se diviser et la nouvelle `ListeMilieu` résultante ajoutée dans la `ListeIndex`. Cette action augmentera le nombre de `ListeMilieu` de la `ListeIndex` de un (1).

La section suivante va décrire les services offerts par ces deux structures.

3 Construction

Vos structures vont être testées par des tests unitaires (JUnit) qui seront disponibles le 23 avril. Pour que ces tests fonctionnent, il est important de respecter les noms pour les classes et les signatures des méthodes. Autre les deux classes et leurs méthodes, vous êtes libre d'ajouter des classes et des méthodes dans votre projet.

Important : pour ce projet, vous ne pouvez pas utiliser de librairies Java autres que les interfaces `Comparable`, `Comparator`, `Exception` et `RuntimeException`.

Voici la description des services que vos classes doivent implémenter.

3.1 ListeMilieu< E extends Comparable< E > >

Dans cette description, la lettre T est utilisée pour représenter la `ListeMilieu` sur laquelle le service est appelé et T^* Représente l'état de cette `ListeMilieu` après que le service ait été appelé.

Contiens :

- inférieure : une liste de valeur de type E. Dois être construite avec une liste simplement chaînée. Dans cette description, nous allons référer à la valeur à la position i de cette liste avec la notation inférieure.get(i). La première valeur est à la position i = 0. Remarquez que cela ne vous oblige pas à écrire une méthode 'get'.

- supérieure : une liste de valeurs de type E. Dois être construite avec une liste simplement chaînée. Dans cette description, nous allons référer à la valeur à la position i de cette liste avec la notation $\mathrm{supérieure.get(\ i\ )}$. La première valeur est à la position $\mathrm{i} = 0$. Remarquez que cela ne vous oblige pas à écrire une méthode 'get'.

Invariants :

1. Toutes les valeurs (v_i) de la liste inférieure sont plus petites ou égales aux valeurs (v_s) de la liste supérieure.

$$\forall \mathrm{v_i} \in \mathrm{T.inférieure}, \forall \mathrm{v_s} \in \mathrm{T.supérieure}, \mathrm{v_i} \leq \mathrm{v_s}$$

2. Les valeurs de la liste supérieure sont triées en ordre croissant.

$$\forall \mathrm{s} \in [0..\mathrm{supérieure.taille} - 2], \mathrm{supérieure.get(s)} \leq \mathrm{supérieure.get(s+1)}$$

3. Les valeurs de la liste inférieure sont triées en ordre décroissant.

$$\forall \mathrm{i} \in [0..\mathrm{inférieure.taille} - 2], \mathrm{inférieure.get(i)} \geq \mathrm{inférieure.get(i+1)}$$

4. La taille de la liste inférieure est égale à la taille de la liste supérieure ou contient une valeur de plus.

$$0 \leq T.inférieure.taille - T.supérieure.taille \leq 1$$

Ces invariants doivent être maintenus par chaque opération.

Services :

- `ListeMilieu()`
  Constructeur par défaut. Construis une liste vide.
  $$T^*.inférieure.taille = 0 \wedge T^*.supérieure.taille = 0$$
- `ListeMilieu<E> diviser()`
  Cette méthode construit une nouvelle `ListeMilieu` ($\mathrm{L^*}$). Les éléments de la liste supérieure sont déplacés dans la nouvelle `ListeMilieu`. Ensuite, une partie des éléments de la liste inférieure sont déplacés dans la liste supérieure afin de rétablir l'invariant.
  $$\forall \mathrm{v_s} \in \mathrm{T.supérieure}, \mathrm{v_s} \in \mathrm{L^*}$$
  $$\forall \mathrm{v_i} \in \mathrm{T.inférieure}, \mathrm{v_i} \in \mathrm{T^*}$$
- `void insérer( E v )`
  Cette méthode ajoute un élément dans la `ListeMilieu`. Si la liste inférieure est vide, alors v est ajouté dans la liste inférieure. Si la première valeur de la liste inférieure est plus grande ou égale à v, alors v est ajouté dans la liste inférieure, sinon v est ajouté dans la liste supérieure.
  $$T.inférieure.taille + T.supérieure.taille + 1 = T^*.inférieure.taille + T^*.supérieure.taille$$
- `E milieu()`
  Retourne la première valeur de la liste inférieure. Cela ne modifie pas la `ListeMilieu`.
- `E minima()`
  Retourne la dernière valeur de la liste inférieure. Cela ne modifie pas la `ListeMilieu`.
- `E maxima()`
  Retourne la dernière valeur de la liste supérieure. Cela ne modifie pas la `ListeMilieu`.
- `void supprimer( E valeur )`
  Trouve la première occurrence de la valeur dans la `ListeMilieu` et l'enlève de la liste.

$$T.inférieure.taille + T.supérieure.taille = T^*.inférieure.taille + T^*.supérieure.taille + 1$$

- `int taille()`
  Retourne le nombre de valeurs qu'il y a dans la `ListeMilieu`. Donc, le nombre de valeurs dans la liste inférieure additionné au nombre de valeur dans la liste supérieure.

3.2 ListeIndex< E extends Comparable< E > >

Dans cette description, la lettre U est utilisée pour représenter la `ListeIndex` sur laquelle le service est appelé et U^* représente l'état de cette `ListeIndex` après que le service ait été appelé.

Contiens :

- Index : une liste de `ListeMilieu`. Dans cette description, nous allons référer à la `ListeMilieu` à la position i de cette liste avec la notation $index.get(i)$. La première valeur est à la position $i = 0$. Nous allons utiliser la notation $index.nbrListe$ pour désigner le nombre de `ListeMilieu` qu'il y a dans l'index.

Invariants :

1. Le nombre de valeurs dans chaque `ListeMilieu` est plus petit ou égal au double du nombre de `ListeMilieu` dans l'index.
   $$\forall m_i \in U.index, m_i.taille \leq 2 \times U.index.nbrListe$$
2. Les `ListeMilieu` de l'index sont triés selon leurs minima et maxima.
   $$\forall i \in [0..U.index.nbrListe - 2], U.index.get(i).maxima < U.index.get(i+1).minima$$

Services :

- `ListeIndex()`
  Un constructeur par défaut. Construire une `ListeIndex` vide.
  $$U^*.index.nbrListe = 0$$
- `boolean contient( E valeur )`
  Retourne `true` si une des `ListeMilieu` contient la valeur en argument. `Faux` sinon.
- `int taille()`
  Retourne la somme de toutes les tailles des `ListeMilieu`s contenue dans l'index.
- `int nbrListe()`
  Retourne le nombre de `ListeMilieu` contenu dans l'index.
- `ListeMilieu< E > get( int i )`
  Retourne la `ListeMilieu` à l'indice `i` de l'index.
- `void insérer( E v )`
  Trouve la `ListeMilieu` de l'index qui peut contenir cette valeur et ajoute la valeur dans cette liste. Une `ListeMilieu` m_i, qui n'est ni la première ni la dernière `ListeMilieu` de l'index, peut contenir une valeur lorsque cette valeur est plus grande ou égale à son minima et plus petite que le minima de la `ListeMilieu` suivante.
  $$m_i.minima \leq v < m_{i+1}.minima$$
  La première `ListeMilieu` m_0 peut contenir toutes les valeurs qui sont plus petites que sont maxima.

$$v < m_0.mixima$$

La dernière `ListeMilieu` m_i peut contenir une valeur lorsque cette valeur est plus grande ou égale à son minima.

$$m_i.minima \leq v$$

Si l'ajout de l'élément dans la liste brise l'invariant 1 de la `ListeIndex` alors il faudra faire appel à la méthode diviser de la `ListeMilieu` qui brise l'invariant. La nouvelle liste est ajoutée dans l'index.

S'il n'y a pas de `ListeMilieu` dans l'index, alors une nouvelle `ListeMilieu` est ajoutée et l'élément est placé dans cette liste.

- `void supprimer( E v )`

  Trouve la `ListeMilieu` pouvant contenir la valeur (voire `inserer` plus haut) et supprime la première occurrence de la valeur dans cette liste.

4 DIRECTIVES

Les sections suivantes décrivent les attentes et les éléments d'évaluation pour le devoir.

4.1 DIRECTIVES POUR LA CONSTRUCTION DU PROJET.

- Placez vos noms au début du fichier `ListeIndex.java`.
- Lorsque vous ajoutez une méthode, vous devez l'ajouter dans la classe appropriée.
- Des tests unitaires vous seront donnés avec les résultats attendus.

4.2 DIRECTIVES POUR L'ÉCRITURE DU CODE.

1. Le TP est à faire seul ou en équipe de deux.
2. Code :
   a. Pas de `goto`, `continue`.
   b. Les `break` ne peuvent apparaitre que dans les `switch`.
   c. Un seul `return` par méthode.
   d. Additionnez le nombre de `if`, `for`, `while`, `switch` et `try`. Ce nombre ne doit pas dépasser 5 pour une méthode.
3. Indentez votre code. Assurez-vous que l'indentation est faite avec des espaces.
4. Commentaires
   - Commentez l'entête de chaque classe et méthode.
   - Une ligne contient soit un commentaire, soit du code, pas les deux.
   - Utilisez des noms d'identificateur significatif.
   - Une ligne de commentaire ou de code ne devrait pas dépasser 120 caractères. Continuez sur la ligne suivante au besoin.
   - Nous utilisons Javadoc :
     - La première ligne d'un commentaire doit contenir une description courte (1 phrase) de la méthode ou la classe.
       - Courte.

        - Complète.
        - Commencez la description avec un verbe.
        - Assurez-vous de ne pas simplement répéter le nom de la méthode, donnez plus d'information.
    - Ensuite, au besoin, une description détaillée de la méthode ou classe va suivre.
        - Indépendant du code. Les commentaires d'entêtes décrivent ce que la méthode fait, ils ne décrivent pas comment c'est fait.
        - Si vous avez besoin de mentionner l'objet courant, utilisez le mot 'this'.
    - Ensuite, avant de placer les **tags,** placez une ligne vide.
    - Placez les **tag** @param, @return et @throws au besoin.
        - @param : décris les valeurs acceptées pour la méthode. Vous devez commenter les paramètres de vos méthodes.
    - Dans les commentaires, placer les noms de variable et autre ligne de code entre les tags {@ *du code ici* }.
    - Écrivez les commentaires à la troisième personne.

4.3 REMISE

Remettre le TP par l'entremise de Moodle. Placez vos fichiers `'*.java'` dans un dossier compressé de **Windows**, vous devez remettre l'archive. Le TP est à remettre avant le 6 mai 23 :55.

4.4 ÉVALUATION

- Fonctionnalité (9 pts) : des tests partiels vous seront remis. Un test plus complet sera appliqué à votre TP. Votre projet doit compiler sans erreur pour avoir ces points.
- Structure (4 pt) : découpez votre code en classe et méthode.
- Lisibilité (2 pts) : commentaire, indentation et noms d'identificateur significatif.