AQL

Release 0.1

John Aoga

Nov 11, 2021

CONTENTS

CHAPTER

ONE

ORGANISATION DU COURS

L'objectif de ce cours est d'aborder la notion de la qualité d'un logiciel et de voir comment on peut s'assurer d'avoir un logiciel de qualité.

A la fin de ce cours l'étudiant doit être capable de:

- Donner une définition claire de la notion de qualité au regard d'un logiciel
- Comprendre et identifier les 06 critères de qualité de la norme ISO9126
- Etre capable d'évaluer un logiciel en terme de qualité
- Etre capable de sélectionner et/ou de créer des modèles permettant de quantifier la qualité
- Etre capable de mettre avant des aspects à prendre en compte au cours de la mise en oeuvre d'un projet pour prendre en compte la qualité.

1.1 Pédagogie

La pédagogie utilisée est mixte. Nous alternerons des:

- cours magistraux pour la clarification des conceptes
- des séances de laboratoires basé sur le modèle de l'apprentissage par soi-même et par projet
- et des travaux pratiques à faire chez soi après chaque module.

Par conséquent, les étudiants doivent impérativement travailler chez eux au jour le jour pour graduellement comprendre et développer leur projet.

1.2 Répartition du cours

Le cours est organisé en plusieurs chapitres.

Les plus important sont:

- *Chap 1*: Introduction à qualité
- *Chap 2* Chap 2: Qualité d'un logiciel
- *Chap 3* Chap 3: Théorie de la mesure
- *Chap 4* Chap 4: Mesure de la qualité d'un logiciel

Si la masse horaire le permet on peut couvrir plusieurs autres aspects liés à la qualité du logiciel.

1.3 Evaluation

Exercise sur Inginious (3) + Projet + TP exercise de maison (en 3 parties) + exposés.

Note 1: Exercise sur Inginious (3) + Projet: (3x inginious * 0.2) + proj * 0.4 Note 2: TP (3 parties) + Exposés: tp * 0.6 + exp * 0.4

1.4 Contact et communication

Les communications se ferons par whatsapp et par mail.

Tel: 97999277 Mail: John Aoga.

1.5 Cours Open-Source

Les sources de ce site web sont open-source et sur GitHub. N'hésitez pas à faire des pull request si vous voyez des erreurs ou choses à corriger.

CHAPTER

TWO

CHAPITRE 1 | INTRODUCTION À LA NOTION QUALITÉ

2.1 Objectifs

A l'issue de cette partie chaque étudiant sera capable de:

- définir clairement la notion de qualité
- faire la différence entre bon logiciel et logiciel de qualité
- situer la qualité dans le contexte d'un logiciel
- faire la différence entre assurance et test d'un logiciel

2.2 Note de théorique

2.2.1 Exercice introductif

1. Qu'est qu'un bon logiciel (on peut se référer à ce que c'est qu'un bon algorithme)? 2.1 Comment définir la qualité? 2.2 Qu'est-ce qu'un logiciel de qualité? 2.3 Quelle différence entre qualité et bon logiciel? 3. Dans vos développement de tous les jours, quels sont les éléments qui se rapportent à la qualité ? 4. La notion étant elle même subjective, comment peut on définir une référence commune à suivre?

2.2.2 Notes

- La notion de qualité
- Un bon algo est un algo correct, complet et effectif (temps et espace)
- un bon logiciel = bon algos + tests
- un bon logiciel ne signifie pas de bug, il n'est pas souvent aisé de prendre en compte tous les cas de figures

surtout quand le projet grandit (les specs même peuvent être la limitation) - AQL vs TQL - Pour s'assurer qu'un logiciel est de qualité on se base sur un processus en 4 étapes - On utilise les normes pour créer une base de référence. La norme ISO9126 définit 06 critères pour juger de la qualité d'un logiciel.

2.3 A lire / Aller plus loin

Slide du cours:

Livres de référence:

Aller plus loin:

2.4 Exercices théoriques

Note: Vous devez faire ces exercices avant S2.

2.4.1 Exercice 1

- A quoi sert la norme ISO 9126
- Comment le résumerai vous en 6 points
- Y a t'il des complément à cette norme?

CHAPTER

THREE

CHAPITRE 2 | QUALITÉ D'UN LOGICIEL ET LA NORME ISO9126

3.1 Objectifs

A l'issue de cette partie chaque étudiant sera capable:

- de définir chaque critère et sous critère de la norme ISO9126.
- de donner des exemples qui renvoie à chaque (sous) critère dans les logiciels de tous les jours
- comprendre les processus d'assurance qualité

3.2 Note de théorique

la norme ISO9126 definit 06 critères de qualité:

- la capacités fonctionnelle: attributs du logiciel à répondre aux specs (aptitude/pertinence, exactitude/conformité, sécurité)
- la fiabilité: attributs du logiciel à maintenir un niveau de service sous certaines conditions(récupération après sinistre, tolérance au fautes, maturité)
- la réutilisabilité: attributs du logiciel à présenter de

3.3 A lire

Livre de référence:

- Chapitre 1, section 1: quelques rappels de Java et la programmation en général
- Chapitre 1, section 2: Abstraction de données
- Chapitre 1, section 3: Piles, files, sacs, listes chainées
- Chapitre 1, section 4: Analyses d'algorithmes

Ainsi que ce document résumant les différentes notations de part1complexity.

Slides (keynote)

- Introduction
- Séance Intermédiaire
- Restructuration

3.4 Exercices théoriques: première partie

Note: Vous devez faire ces exercices pour le mercredi de S2.

3.4.1 Exercice 1.1.1

Définissez ce qu'est un type abstrait de données (TAD[1]). En java, est-il préférable de décrire un TAD par une classe ou une interface ? Pourquoi ?

3.4.2 Exercice 1.1.2

Comment faire pour implémenter une *pile* par une liste simplement chaînée où les opérations *push* et *pop* se font en **fin de liste** ? Cette solution est-elle efficace ? Argumentez.

3.4.3 Exercice 1.1.3

Quelles sont les implémentations possibles pour une pile? En consultant la documentation sur l'API de Java, décrivez l'implémentation d'une pile par la classe *java.util.Stack*. Aller voir le code source de l'implémentation *java.util.Stack* (crtl+B depuis IntelliJ).

Pourquoi pensez-vous que les développeurs de Java ont choisi cette implémentation (hint: argumentez au niveau de la mémoire et du garbage collector)?

3.4.4 Exercice 1.1.4

Comment faire pour implémenter le type abstrait de données *Pile* à l'aide de deux *files* ? Décrivez en particulier le fonctionnement des méthodes *push* et *pop* dans ce cas.

A titre d'exemple, précisez l'état de chacune des deux files après avoir empilé les entiers *1 2 3* à partir d'une pile initialement vide. Décrivez ce qu'il se passe ensuite lorsque l'on effectue l'opération *pop*.

Quelle est la complexité temporelle de ces méthodes si l'on suppose que chaque opération *enqueue* et *dequeue* s'exécute en temps constant?

Cette implémentation d'une pile est-elle efficace (pour n opérations) par rapport aux autres implémentations présentées dans le livre de référence ?

3.4.5 Exercice 1.1.5

- Qu'est-ce qu'un iterateur en Java (*java.util.Iterator*)?
- Pourquoi est-ce utile de définir une méthode *iterator()* sur les structures de données?
- Que pensez vous de permettre la modification d'une structure de donnée alors qu'on est en train d'itérer sur celle-ci?

[1] *abstract data type* (ADT) en anglais

Pour vous aider dans la réflexion, nous vous invitons à lire la spécification de l'API Java concernant la méthode *remove()*.

Proposez une modification du code de l'iterateur de Stack qui lance une *java.util.ConcurrentModificationException* si le client modifie la collection avec un *push()* ou *pop()* durant l'itération. Est-ce une bonne idée de laisser l'implémentation de la méthode *remove()* vide si on ne désire pas permettre cette fonctionnalité?

3.4.6 Exercice 1.1.6

La notation $\sim$ (tilde) est utilisée dans le livre de référence pour l'analyse des temps de calcul des algorithmes. En quoi cette notation diffère ou ressemble aux notations plus classiquement utilisées $\mathcal{O}$ (big Oh), $\otimes$ (big Omega) et $\times$ (big Theta)?

Expliquez précisément les liens et similitudes entre celles-ci. Que voyez-vous comme avantage à utiliser la notation $\sim$ (tilde) plutôt que $\mathcal{O}$ lorsque c'est possible?

3.4.7 Exercice 1.1.7

Expliquez comment nous pouvons extraire la caractérisation $\sim$ (tilde) de l'implémentation d'un algorithme à l'aide du test *Doubling ratio*.

- Comment fonctionne ce test?
- Quelles sont les limites et avantages de ce test?

Supposont que nous mesurons les temps d'exécutions $T(n)$ suivants (en secondes) d'un programme en fonction de la taille de l'entrée n:

n	1000	2000	4000	8000	16000	32000	64000
$T(n)$	0	0	0.1	0.3	1.3	5.1	20.5

- Comment pouvez-vous caractériser au mieux l'ordre de croissance de cette fonction ?
- Que serait le temps d'exécution pour 128000?

3.5 Exercices théoriques supplémentaires

Note: Ces exercices ne seront pas forcéments résolus en cours, ils restent néanmoins intéressants. Si vous avez des problèmes avec ceux-ci, posez votre question lors d'un TP.

3.5.1 Exercice 1.1b.1

Que signifient les paramètres -Xmx, -Xms que l'on peut passer à la JVM pour l'exécution d'un bytecode? Est-ce que ces paramètres peuvent influencer la vitesse d'exécution d'un programme Java? Pourquoi?

3.5.2 Exercice 1.1b.2

- Qu'est-ce qu'un bon ensemble de tests unitaires pour vérifier l'exactitude d'une structure de données?
- Pensez-vous aux cas limites?
- Pensez-vous à la valeur maximale des entiers, doubles, etc?
- En quoi la génération de données aléatoire peut être utile pour tester les structures de données?
- Pourquoi est-ce important de travailler avec une semence (seed) fixée?
- En quoi un outil d'analyse de couverture de code peut être utile (tel que Jacoco) pour vous aidez à concevoir des tests.
- Comment vérifier expérimentalement que l'implémentation d'une structure de données ou un algorithme a bien la complexité temporelle théorique attendue ?

3.6 Exercices sur Inginious

Note: Vous devez faire ces exercices pour le mercredi de S3.

1. Ecriture de tests unitaires pour une stack
2. Implementation d'une stack avec structure chainée
3. Implementation d'une liste chainée circulaire et d'un iterateur

3.7 Exercices théorique: deuxième partie

Note: Vous devez faire ces exercices pour le mercredi de S3.

3.7.1 Exercice 1.2.1

Dans votre implémentation d'une liste chainée circulaire ci-dessous. Quelle est la complexité de la méthode

- *public void enqueue(Item item)*?
- *public Item remove(int index)* ?
- d'une séquence d'operations qui consiste à *créer un iterateur et ensuite itérer sur les k-premiers elements* ?

```
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class CircularLinkedList<Item> implements Iterable<Item> {
 private long nOp = 0; // count the number of operations
 private int n;          // size of the stack
 private Node  last;   // trailer of the list
```

(continues on next page)

(continued from previous page)

```
// helper linked list class
private class Node {
    private Item item;
    private Node next;
}

public CircularLinkedList() {
    last = new Node(); // dummy node
    last.next = last;
    n = 1;
}

public boolean isEmpty() { return n == 1; }

public int size() { return n-1; }

private long nOp() { return nOp; }

/**
 * Append an item at the end of the list
 * @param item the item to append
 */
public void enqueue(Item item) {
    // TODO STUDENT: Implement add method
}

/**
 * Removes the element at the specified position in this list.
 * Shifts any subsequent elements to the left (subtracts one from their indices).
 * Returns the element that was removed from the list.
 */
public Item remove(int index) {
    // TODO STUDENT: Implement remove method
}

/**
 * Returns an iterator that iterates through the items in FIFO order.
 * @return an iterator that iterates through the items in FIFO order.
 */
public Iterator<Item> iterator() {
    return new ListIterator();
}

/**
 * Implementation of an iterator that iterates through the items in FIFO order.
 *
 */
private class ListIterator implements Iterator<Item> {
    // TODO STUDENT: Implement the ListIterator
}

}
```

3.7.2 Exercice 1.2.2

La notation post-fixe (ou polonaise inverse) est utilisée pour représenter des expressions algébriques. Nous ne considérons pour simplifier que des expressions post-fixes avec des entiers positifs et les opérateurs + et *. Par exemple *2 3 1 * + 9 *** dont le résultat vaut 45 et le résultat de *4 20 + 3 5 1 * * + +** est 39.

1. Ecrivez un algorithme en Java pour évaluer une expression post-fixe au départ d'une chaine de n-caractères.
2. Quelle structure de donnée utilisez vous ?
3. Quelle est la complexité de votre algorithme (temporelle et spatiale) ?

Pour rappel, voici comment on peut itérer sur les elements d'une chaine qui sont séparés par des espaces.

```
String in = "4 20 + 3 5 1 * * +";
StringTokenizer tokenizer = new StringTokenizer(in);
while (tokenizer.hasMoreTokens()) {
     String element = tokenizer.nextToken();
}
```

3.7.3 Exercice 1.2.3

La programmation fonctionnelle est un paradigme de programmation de plus en plus important. Dans ce paradigme de programmation, les structures de données sont immutables . Nous nous intéressons ici à l'implémentation d'une liste immutable appelée *FList* permettant d'être utilisée dans un cadre fonctionnel. Voici l'API d'une *FList*

```
public abstract class FList<A> implements Iterable<A> {

    // creates an empty list
    public static <A> FList<A> nil();

    // prepend a to the list and return the new list
    public final FList<A> cons(final A a);

    public final boolean isNotEmpty();

    public final boolean isEmpty();

    public final int length();

    // return the head element of the list
    public abstract A head();

    // return the tail of the list
    public abstract FList<A> tail();

    // return a list on which each element has been applied function f
    public final <B> FList<B> map(Function<A,B> f);

    // return a list on which only the elements that satisfies predicate are kept
    public final FList<A> filter(Predicate<A> f);

    // return an iterator on the element of the list
    public Iterator<A> iterator();
```

(continues on next page)

(continued from previous page)

```
}
```

Comme vous pouvez vous en rendre compte, aucune des méthodes ne permet de modifier l'état de la liste. Voici un exemple de manipulation d'une telle liste. Si vous n'êtes pas familiers avec les interfaces fonctionnelles de Java8, nous vous demandons de vous familiariser d'abord avec celles-ci.

```
FList<Integer> list = FList.nil();

for (int i = 0; i < 10; i++) {
    list = list.cons(i);
}

list = list.map(i -> i+1);
// will print 1,2,...,11
for (Integer i: list) {
    System.out.println(i);
}

list = list.filter(i -> i%2 == 0);
// will print 2,4,6,...,10
for (Integer i: list) {
    System.out.println(i);
}
```

Voici une implémentation partielle de la *FList*

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.function.Function;
import java.util.function.Predicate;

public abstract class FList<A> implements Iterable<A> {

    public final boolean isNotEmpty() {
        return this instanceof Cons;
    }

    public final boolean isEmpty() {
        return this instanceof Nil;
    }

    public final int length() {
        // TODO
    }

    public abstract A head();

    public abstract FList<A> tail();

    public static <A> FList<A> nil() {
        return (Nil<A>) Nil.INSTANCE;
```

(continues on next page)

(continued from previous page)

```
    }

    public final FList<A> cons(final A a) {
        return new Cons(a, this);
    }

    public final <B> FList<B> map(Function<A,B> f) {
        // TODO
    }

    public final FList<A> filter(Predicate<A> f) {
        // TODO
    }


    public Iterator<A> iterator() {
        return new Iterator<A>() {
            // complete this class


            public boolean hasNext() {
              // TODO
            }

            public A next() {
              // TODO
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }


    private static final class Nil<A> extends FList<A> {
        public static final Nil<Object> INSTANCE = new Nil();
        // TODO
    }

    private static final class Cons<A> extends FList<A> {
        // TODO
    }


}
```

Nous vous demandons de

- compléter cette implémentation, si possible utilisez autant que possible des méthodes récursives.
- déterminer la complexité de chacune des méthodes.

3.8 Ressources supplémentaires

Notes de bas de page

CHAPTER

FOUR

PARTIE 3 | TYPES ABSTRAITS DE DONNÉES, COMPLEXITÉ, COLLECTIONS JAVA; PILES, FILES ET LISTES LIÉES

4.1 Objectifs

A l'issue de cette partie chaque étudiant sera capable de:

- faire la disctinction entre les notations $\sim$, $\mathcal{O}$, $\times$, et $\otimes$, connaitre leurs propriétés et définitions;
- décrire avec précision les propriétés des types abstraits *pile* et *file*; ainsi que les divers types de listes chainées;
- faire la distinction entre un *type abstrait de données* et son implémentation;
- mettre en oeuvre et évaluer une implémentation d'une pile par une *liste simplement ou doublement chaînée*;
- utiliser des *tests unitaires* (avec JUnit) pour tester et prouver le bon fonctionnement d'un programme;
- utiliser les diverses collections présentes de base dans la language Java, en s'aidant de la documentation.

4.2 A lire

Livre de référence:

- Chapitre 1, section 1: quelques rappels de Java et la programmation en général
- Chapitre 1, section 2: Abstraction de données
- Chapitre 1, section 3: Piles, files, sacs, listes chainées
- Chapitre 1, section 4: Analyses d'algorithmes

Ainsi que ce document résumant les différentes notations de part1complexity.

Slides (keynote)

- Introduction
- Séance Intermédiaire
- Restructuration

4.3 Exercices théoriques: première partie

Note: Vous devez faire ces exercices pour le mercredi de S2.

4.3.1 Exercice 1.1.1

Définissez ce qu'est un type abstrait de données (TAD[1]). En java, est-il préférable de décrire un TAD par une classe ou une interface ? Pourquoi ?

4.3.2 Exercice 1.1.2

Comment faire pour implémenter une *pile* par une liste simplement chaînée où les opérations *push* et *pop* se font en **fin de liste** ? Cette solution est-elle efficace ? Argumentez.

4.3.3 Exercice 1.1.3

Quelles sont les implémentations possibles pour une pile? En consultant la documentation sur l'API de Java, décrivez l'implémentation d'une pile par la classe *java.util.Stack*. Aller voir le code source de l'implémentation *java.util.Stack* (crtl+B depuis IntelliJ).

Pourquoi pensez-vous que les développeurs de Java ont choisi cette implémentation (hint: argumentez au niveau de la mémoire et du garbage collector)?

4.3.4 Exercice 1.1.4

Comment faire pour implémenter le type abstrait de données *Pile* à l'aide de deux *files* ? Décrivez en particulier le fonctionnement des méthodes *push* et *pop* dans ce cas.

A titre d'exemple, précisez l'état de chacune des deux files après avoir empilé les entiers *1 2 3* à partir d'une pile initialement vide. Décrivez ce qu'il se passe ensuite lorsque l'on effectue l'opération *pop*.

Quelle est la complexité temporelle de ces méthodes si l'on suppose que chaque opération *enqueue* et *dequeue* s'exécute en temps constant?

Cette implémentation d'une pile est-elle efficace (pour n opérations) par rapport aux autres implémentations présentées dans le livre de référence ?

4.3.5 Exercice 1.1.5

- Qu'est-ce qu'un iterateur en Java (*java.util.Iterator*)?
- Pourquoi est-ce utile de définir une méthode *iterator()* sur les structures de données?
- Que pensez vous de permettre la modification d'une structure de donnée alors qu'on est en train d'itérer sur celle-ci?

[1] *abstract data type* (ADT) en anglais

Pour vous aider dans la réflexion, nous vous invitons à lire la spécification de l'API Java concernant la méthode *remove()*.

Proposez une modification du code de l'iterateur de Stack qui lance une *java.util.ConcurrentModificationException* si le client modifie la collection avec un *push()* ou *pop()* durant l'itération. Est-ce une bonne idée de laisser l'implémentation de la méthode *remove()* vide si on ne désire pas permettre cette fonctionnalité?

4.3.6 Exercice 1.1.6

La notation $\sim$ (tilde) est utilisée dans le livre de référence pour l'analyse des temps de calcul des algorithmes. En quoi cette notation diffère ou ressemble aux notations plus classiquement utilisées $\mathcal{O}$ (big Oh), $\otimes$ (big Omega) et $\times$ (big Theta)?

Expliquez précisément les liens et similitudes entre celles-ci. Que voyez-vous comme avantage à utiliser la notation $\sim$ (tilde) plutôt que $\mathcal{O}$ lorsque c'est possible?

4.3.7 Exercice 1.1.7

Expliquez comment nous pouvons extraire la caractérisation $\sim$ (tilde) de l'implémentation d'un algorithme à l'aide du test *Doubling ratio*.

- Comment fonctionne ce test?
- Quelles sont les limites et avantages de ce test?

Supposont que nous mesurons les temps d'exécutions $T(n)$ suivants (en secondes) d'un programme en fonction de la taille de l'entrée n:

n	1000	2000	4000	8000	16000	32000	64000
$T(n)$	0	0	0.1	0.3	1.3	5.1	20.5

- Comment pouvez-vous caractériser au mieux l'ordre de croissance de cette fonction ?
- Que serait le temps d'exécution pour 128000?

4.4 Exercices théoriques supplémentaires

Note: Ces exercices ne seront pas forcéments résolus en cours, ils restent néanmoins intéressants. Si vous avez des problèmes avec ceux-ci, posez votre question lors d'un TP.

4.4.1 Exercice 1.1b.1

Que signifient les paramètres -Xmx, -Xms que l'on peut passer à la JVM pour l'exécution d'un bytecode? Est-ce que ces paramètres peuvent influencer la vitesse d'exécution d'un programme Java? Pourquoi?

4.4.2 Exercice 1.1b.2

- Qu'est-ce qu'un bon ensemble de tests unitaires pour vérifier l'exactitude d'une structure de données?
- Pensez-vous aux cas limites?
- Pensez-vous à la valeur maximale des entiers, doubles, etc?
- En quoi la génération de données aléatoire peut être utile pour tester les structures de données?
- Pourquoi est-ce important de travailler avec une semence (seed) fixée?
- En quoi un outil d'analyse de couverture de code peut être utile (tel que Jacoco) pour vous aidez à concevoir des tests.
- Comment vérifier expérimentalement que l'implémentation d'une structure de données ou un algorithme a bien la complexité temporelle théorique attendue ?

4.5 Exercices sur Inginious

Note: Vous devez faire ces exercices pour le mercredi de S3.

1. Ecriture de tests unitaires pour une stack
2. Implementation d'une stack avec structure chainée
3. Implementation d'une liste chainée circulaire et d'un iterateur

4.6 Exercices théorique: deuxième partie

Note: Vous devez faire ces exercices pour le mercredi de S3.

4.6.1 Exercice 1.2.1

Dans votre implémentation d'une liste chainée circulaire ci-dessous. Quelle est la complexité de la méthode

- *public void enqueue(Item item)*?
- *public Item remove(int index)* ?
- d'une séquence d'operations qui consiste à *créer un iterateur et ensuite itérer sur les k-premiers elements* ?

```
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class CircularLinkedList<Item> implements Iterable<Item> {
 private long nOp = 0; // count the number of operations
 private int n;          // size of the stack
 private Node  last;   // trailer of the list
```

(continues on next page)

(continued from previous page)

```
// helper linked list class
private class Node {
    private Item item;
    private Node next;
}

public CircularLinkedList() {
    last = new Node(); // dummy node
    last.next = last;
    n = 1;
}

public boolean isEmpty() { return n == 1; }

public int size() { return n-1; }

private long nOp() { return nOp; }

/**
 * Append an item at the end of the list
 * @param item the item to append
 */
public void enqueue(Item item) {
    // TODO STUDENT: Implement add method
}

/**
 * Removes the element at the specified position in this list.
 * Shifts any subsequent elements to the left (subtracts one from their indices).
 * Returns the element that was removed from the list.
 */
public Item remove(int index) {
    // TODO STUDENT: Implement remove method
}

/**
 * Returns an iterator that iterates through the items in FIFO order.
 * @return an iterator that iterates through the items in FIFO order.
 */
public Iterator<Item> iterator() {
    return new ListIterator();
}

/**
 * Implementation of an iterator that iterates through the items in FIFO order.
 *
 */
private class ListIterator implements Iterator<Item> {
    // TODO STUDENT: Implement the ListIterator
}

}
```

4.6.2 Exercice 1.2.2

La notation post-fixe (ou polonaise inverse) est utilisée pour représenter des expressions algébriques. Nous ne considérons pour simplifier que des expressions post-fixes avec des entiers positifs et les opérateurs + et *. Par exemple *2 3 1 * + 9 *** dont le résultat vaut 45 et le résultat de *4 20 + 3 5 1 * * + +** est 39.

1. Ecrivez un algorithme en Java pour évaluer une expression post-fixe au départ d'une chaine de n-caractères.
2. Quelle structure de donnée utilisez vous ?
3. Quelle est la complexité de votre algorithme (temporelle et spatiale) ?

Pour rappel, voici comment on peut itérer sur les elements d'une chaine qui sont séparés par des espaces.

```
String in = "4 20 + 3 5 1 * * +";
StringTokenizer tokenizer = new StringTokenizer(in);
while (tokenizer.hasMoreTokens()) {
     String element = tokenizer.nextToken();
}
```

4.6.3 Exercice 1.2.3

La programmation fonctionnelle est un paradigme de programmation de plus en plus important. Dans ce paradigme de programmation, les structures de données sont immutables . Nous nous intéressons ici à l'implémentation d'une liste immutable appelée *FList* permettant d'être utilisée dans un cadre fonctionnel. Voici l'API d'une *FList*

```
public abstract class FList<A> implements Iterable<A> {

    // creates an empty list
    public static <A> FList<A> nil();

    // prepend a to the list and return the new list
    public final FList<A> cons(final A a);

    public final boolean isNotEmpty();

    public final boolean isEmpty();

    public final int length();

    // return the head element of the list
    public abstract A head();

    // return the tail of the list
    public abstract FList<A> tail();

    // return a list on which each element has been applied function f
    public final <B> FList<B> map(Function<A,B> f);

    // return a list on which only the elements that satisfies predicate are kept
    public final FList<A> filter(Predicate<A> f);

    // return an iterator on the element of the list
    public Iterator<A> iterator();
```

(continues on next page)

(continued from previous page)

```
}
```

Comme vous pouvez vous en rendre compte, aucune des méthodes ne permet de modifier l'état de la liste. Voici un exemple de manipulation d'une telle liste. Si vous n'êtes pas familiers avec les interfaces fonctionnelles de Java8, nous vous demandons de vous familiariser d'abord avec celles-ci.

```
FList<Integer> list = FList.nil();

for (int i = 0; i < 10; i++) {
    list = list.cons(i);
}

list = list.map(i -> i+1);
// will print 1,2,...,11
for (Integer i: list) {
    System.out.println(i);
}

list = list.filter(i -> i%2 == 0);
// will print 2,4,6,...,10
for (Integer i: list) {
    System.out.println(i);
}
```

Voici une implémentation partielle de la *FList*

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.function.Function;
import java.util.function.Predicate;

public abstract class FList<A> implements Iterable<A> {

    public final boolean isNotEmpty() {
        return this instanceof Cons;
    }

    public final boolean isEmpty() {
        return this instanceof Nil;
    }

    public final int length() {
        // TODO
    }

    public abstract A head();

    public abstract FList<A> tail();

    public static <A> FList<A> nil() {
        return (Nil<A>) Nil.INSTANCE;
```

(continues on next page)

(continued from previous page)

```
    }

    public final FList<A> cons(final A a) {
        return new Cons(a, this);
    }

    public final <B> FList<B> map(Function<A,B> f) {
        // TODO
    }

    public final FList<A> filter(Predicate<A> f) {
        // TODO
    }


    public Iterator<A> iterator() {
        return new Iterator<A>() {
            // complete this class


            public boolean hasNext() {
              // TODO
            }

            public A next() {
              // TODO
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }


    private static final class Nil<A> extends FList<A> {
        public static final Nil<Object> INSTANCE = new Nil();
        // TODO
    }

    private static final class Cons<A> extends FList<A> {
        // TODO
    }


}
```

Nous vous demandons de

- compléter cette implémentation, si possible utilisez autant que possible des méthodes récursives.
- déterminer la complexité de chacune des méthodes.

4.7 Ressources supplémentaires

Notes de bas de page

CHAPTER

FIVE

PARTIE 4 | TYPES ABSTRAITS DE DONNÉES, COMPLEXITÉ, COLLECTIONS JAVA; PILES, FILES ET LISTES LIÉES

5.1 Objectifs

A l'issue de cette partie chaque étudiant sera capable de:

- faire la disctinction entre les notations $\sim$, $\mathcal{O}$, $\times$, et $\otimes$, connaitre leurs propriétés et définitions;
- décrire avec précision les propriétés des types abstraits *pile* et *file*; ainsi que les divers types de listes chainées;
- faire la distinction entre un *type abstrait de données* et son implémentation;
- mettre en oeuvre et évaluer une implémentation d'une pile par une *liste simplement ou doublement chaînée*;
- utiliser des *tests unitaires* (avec JUnit) pour tester et prouver le bon fonctionnement d'un programme;
- utiliser les diverses collections présentes de base dans la language Java, en s'aidant de la documentation.

5.2 A lire

Livre de référence:

- Chapitre 1, section 1: quelques rappels de Java et la programmation en général
- Chapitre 1, section 2: Abstraction de données
- Chapitre 1, section 3: Piles, files, sacs, listes chainées
- Chapitre 1, section 4: Analyses d'algorithmes

Ainsi que ce document résumant les différentes notations de part1complexity.

Slides (keynote)

- Introduction
- Séance Intermédiaire
- Restructuration

5.3 Exercices théoriques: première partie

Note: Vous devez faire ces exercices pour le mercredi de S2.

5.3.1 Exercice 1.1.1

Définissez ce qu'est un type abstrait de données (TAD[1]). En java, est-il préférable de décrire un TAD par une classe ou une interface ? Pourquoi ?

5.3.2 Exercice 1.1.2

Comment faire pour implémenter une *pile* par une liste simplement chaînée où les opérations *push* et *pop* se font en **fin de liste** ? Cette solution est-elle efficace ? Argumentez.

5.3.3 Exercice 1.1.3

Quelles sont les implémentations possibles pour une pile? En consultant la documentation sur l'API de Java, décrivez l'implémentation d'une pile par la classe *java.util.Stack*. Aller voir le code source de l'implémentation *java.util.Stack* (crtl+B depuis IntelliJ).

Pourquoi pensez-vous que les développeurs de Java ont choisi cette implémentation (hint: argumentez au niveau de la mémoire et du garbage collector)?

5.3.4 Exercice 1.1.4

Comment faire pour implémenter le type abstrait de données *Pile* à l'aide de deux *files* ? Décrivez en particulier le fonctionnement des méthodes *push* et *pop* dans ce cas.

A titre d'exemple, précisez l'état de chacune des deux files après avoir empilé les entiers *1 2 3* à partir d'une pile initialement vide. Décrivez ce qu'il se passe ensuite lorsque l'on effectue l'opération *pop*.

Quelle est la complexité temporelle de ces méthodes si l'on suppose que chaque opération *enqueue* et *dequeue* s'exécute en temps constant?

Cette implémentation d'une pile est-elle efficace (pour n opérations) par rapport aux autres implémentations présentées dans le livre de référence ?

5.3.5 Exercice 1.1.5

- Qu'est-ce qu'un iterateur en Java (*java.util.Iterator*)?
- Pourquoi est-ce utile de définir une méthode *iterator()* sur les structures de données?
- Que pensez vous de permettre la modification d'une structure de donnée alors qu'on est en train d'itérer sur celle-ci?

[1] *abstract data type* (ADT) en anglais

Pour vous aider dans la réflexion, nous vous invitons à lire la spécification de l'API Java concernant la méthode *remove()*.

Proposez une modification du code de l'iterateur de Stack qui lance une *java.util.ConcurrentModificationException* si le client modifie la collection avec un *push()* ou *pop()* durant l'itération. Est-ce une bonne idée de laisser l'implémentation de la méthode *remove()* vide si on ne désire pas permettre cette fonctionnalité?

5.3.6 Exercice 1.1.6

La notation $\sim$ (tilde) est utilisée dans le livre de référence pour l'analyse des temps de calcul des algorithmes. En quoi cette notation diffère ou ressemble aux notations plus classiquement utilisées $\mathcal{O}$ (big Oh), $\otimes$ (big Omega) et $\times$ (big Theta)?

Expliquez précisément les liens et similitudes entre celles-ci. Que voyez-vous comme avantage à utiliser la notation $\sim$ (tilde) plutôt que $\mathcal{O}$ lorsque c'est possible?

5.3.7 Exercice 1.1.7

Expliquez comment nous pouvons extraire la caractérisation $\sim$ (tilde) de l'implémentation d'un algorithme à l'aide du test *Doubling ratio*.

- Comment fonctionne ce test?
- Quelles sont les limites et avantages de ce test?

Supposont que nous mesurons les temps d'exécutions $T(n)$ suivants (en secondes) d'un programme en fonction de la taille de l'entrée n:

n	1000	2000	4000	8000	16000	32000	64000
$T(n)$	0	0	0.1	0.3	1.3	5.1	20.5

- Comment pouvez-vous caractériser au mieux l'ordre de croissance de cette fonction ?
- Que serait le temps d'exécution pour 128000?

5.4 Exercices théoriques supplémentaires

Note: Ces exercices ne seront pas forcéments résolus en cours, ils restent néanmoins intéressants. Si vous avez des problèmes avec ceux-ci, posez votre question lors d'un TP.

5.4.1 Exercice 1.1b.1

Que signifient les paramètres -Xmx, -Xms que l'on peut passer à la JVM pour l'exécution d'un bytecode? Est-ce que ces paramètres peuvent influencer la vitesse d'exécution d'un programme Java? Pourquoi?

5.4.2 Exercice 1.1b.2

- Qu'est-ce qu'un bon ensemble de tests unitaires pour vérifier l'exactitude d'une structure de données?
- Pensez-vous aux cas limites?
- Pensez-vous à la valeur maximale des entiers, doubles, etc?
- En quoi la génération de données aléatoire peut être utile pour tester les structures de données?
- Pourquoi est-ce important de travailler avec une semence (seed) fixée?
- En quoi un outil d'analyse de couverture de code peut être utile (tel que Jacoco) pour vous aidez à concevoir des tests.
- Comment vérifier expérimentalement que l'implémentation d'une structure de données ou un algorithme a bien la complexité temporelle théorique attendue ?

5.5 Exercices sur Inginious

Note: Vous devez faire ces exercices pour le mercredi de S3.

1. Ecriture de tests unitaires pour une stack
2. Implementation d'une stack avec structure chainée
3. Implementation d'une liste chainée circulaire et d'un iterateur

5.6 Exercices théorique: deuxième partie

Note: Vous devez faire ces exercices pour le mercredi de S3.

5.6.1 Exercice 1.2.1

Dans votre implémentation d'une liste chainée circulaire ci-dessous. Quelle est la complexité de la méthode

- *public void enqueue(Item item)*?
- *public Item remove(int index)* ?
- d'une séquence d'operations qui consiste à *créer un iterateur et ensuite itérer sur les k-premiers elements* ?

```
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class CircularLinkedList<Item> implements Iterable<Item> {
 private long nOp = 0; // count the number of operations
 private int n;          // size of the stack
 private Node  last;   // trailer of the list
```

(continues on next page)

(continued from previous page)

```
// helper linked list class
private class Node {
    private Item item;
    private Node next;
}

public CircularLinkedList() {
    last = new Node(); // dummy node
    last.next = last;
    n = 1;
}

public boolean isEmpty() { return n == 1; }

public int size() { return n-1; }

private long nOp() { return nOp; }

/**
 * Append an item at the end of the list
 * @param item the item to append
 */
public void enqueue(Item item) {
    // TODO STUDENT: Implement add method
}

/**
 * Removes the element at the specified position in this list.
 * Shifts any subsequent elements to the left (subtracts one from their indices).
 * Returns the element that was removed from the list.
 */
public Item remove(int index) {
    // TODO STUDENT: Implement remove method
}

/**
 * Returns an iterator that iterates through the items in FIFO order.
 * @return an iterator that iterates through the items in FIFO order.
 */
public Iterator<Item> iterator() {
    return new ListIterator();
}

/**
 * Implementation of an iterator that iterates through the items in FIFO order.
 *
 */
private class ListIterator implements Iterator<Item> {
    // TODO STUDENT: Implement the ListIterator
}

}
```

5.6.2 Exercice 1.2.2

La notation post-fixe (ou polonaise inverse) est utilisée pour représenter des expressions algébriques. Nous ne considérons pour simplifier que des expressions post-fixes avec des entiers positifs et les opérateurs + et *. Par exemple *2 3 1 * + 9 *** dont le résultat vaut 45 et le résultat de *4 20 + 3 5 1 * * + +** est 39.

1. Ecrivez un algorithme en Java pour évaluer une expression post-fixe au départ d'une chaine de n-caractères.
2. Quelle structure de donnée utilisez vous ?
3. Quelle est la complexité de votre algorithme (temporelle et spatiale) ?

Pour rappel, voici comment on peut itérer sur les elements d'une chaine qui sont séparés par des espaces.

```
String in = "4 20 + 3 5 1 * * +";
StringTokenizer tokenizer = new StringTokenizer(in);
while (tokenizer.hasMoreTokens()) {
     String element = tokenizer.nextToken();
}
```

5.6.3 Exercice 1.2.3

La programmation fonctionnelle est un paradigme de programmation de plus en plus important. Dans ce paradigme de programmation, les structures de données sont immutables . Nous nous intéressons ici à l'implémentation d'une liste immutable appelée *FList* permettant d'être utilisée dans un cadre fonctionnel. Voici l'API d'une *FList*

```
public abstract class FList<A> implements Iterable<A> {

    // creates an empty list
    public static <A> FList<A> nil();

    // prepend a to the list and return the new list
    public final FList<A> cons(final A a);

    public final boolean isNotEmpty();

    public final boolean isEmpty();

    public final int length();

    // return the head element of the list
    public abstract A head();

    // return the tail of the list
    public abstract FList<A> tail();

    // return a list on which each element has been applied function f
    public final <B> FList<B> map(Function<A,B> f);

    // return a list on which only the elements that satisfies predicate are kept
    public final FList<A> filter(Predicate<A> f);

    // return an iterator on the element of the list
    public Iterator<A> iterator();
```

(continues on next page)

(continued from previous page)

```
}
```

Comme vous pouvez vous en rendre compte, aucune des méthodes ne permet de modifier l'état de la liste. Voici un exemple de manipulation d'une telle liste. Si vous n'êtes pas familiers avec les interfaces fonctionnelles de Java8, nous vous demandons de vous familiariser d'abord avec celles-ci.

```
FList<Integer> list = FList.nil();

for (int i = 0; i < 10; i++) {
    list = list.cons(i);
}

list = list.map(i -> i+1);
// will print 1,2,...,11
for (Integer i: list) {
    System.out.println(i);
}

list = list.filter(i -> i%2 == 0);
// will print 2,4,6,...,10
for (Integer i: list) {
    System.out.println(i);
}
```

Voici une implémentation partielle de la *FList*

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.function.Function;
import java.util.function.Predicate;

public abstract class FList<A> implements Iterable<A> {

    public final boolean isNotEmpty() {
        return this instanceof Cons;
    }

    public final boolean isEmpty() {
        return this instanceof Nil;
    }

    public final int length() {
        // TODO
    }

    public abstract A head();

    public abstract FList<A> tail();

    public static <A> FList<A> nil() {
        return (Nil<A>) Nil.INSTANCE;
```

(continues on next page)

(continued from previous page)

```
    }

    public final FList<A> cons(final A a) {
        return new Cons(a, this);
    }

    public final <B> FList<B> map(Function<A,B> f) {
        // TODO
    }

    public final FList<A> filter(Predicate<A> f) {
        // TODO
    }


    public Iterator<A> iterator() {
        return new Iterator<A>() {
            // complete this class


            public boolean hasNext() {
              // TODO
            }

            public A next() {
              // TODO
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }


    private static final class Nil<A> extends FList<A> {
        public static final Nil<Object> INSTANCE = new Nil();
        // TODO
    }

    private static final class Cons<A> extends FList<A> {
        // TODO
    }


}
```

Nous vous demandons de

- compléter cette implémentation, si possible utilisez autant que possible des méthodes récursives.
- déterminer la complexité de chacune des méthodes.

5.7 Ressources supplémentaires

Notes de bas de page

CHAPTER

SIX

PARTIE 5 | TYPES ABSTRAITS DE DONNÉES, COMPLEXITÉ, COLLECTIONS JAVA; PILES, FILES ET LISTES LIÉES

6.1 Objectifs

A l'issue de cette partie chaque étudiant sera capable de:

- faire la disctinction entre les notations $\sim$, $\mathcal{O}$, $\times$, et $\otimes$, connaitre leurs propriétés et définitions;
- décrire avec précision les propriétés des types abstraits *pile* et *file*; ainsi que les divers types de listes chainées;
- faire la distinction entre un *type abstrait de données* et son implémentation;
- mettre en oeuvre et évaluer une implémentation d'une pile par une *liste simplement ou doublement chaînée*;
- utiliser des *tests unitaires* (avec JUnit) pour tester et prouver le bon fonctionnement d'un programme;
- utiliser les diverses collections présentes de base dans la language Java, en s'aidant de la documentation.

6.2 A lire

Livre de référence:

- Chapitre 1, section 1: quelques rappels de Java et la programmation en général
- Chapitre 1, section 2: Abstraction de données
- Chapitre 1, section 3: Piles, files, sacs, listes chainées
- Chapitre 1, section 4: Analyses d'algorithmes

Ainsi que ce document résumant les différentes notations de part1complexity.

Slides (keynote)

- Introduction
- Séance Intermédiaire
- Restructuration

6.3 Exercices théoriques: première partie

Note: Vous devez faire ces exercices pour le mercredi de S2.

6.3.1 Exercice 1.1.1

Définissez ce qu'est un type abstrait de données (TAD[1]). En java, est-il préférable de décrire un TAD par une classe ou une interface ? Pourquoi ?

6.3.2 Exercice 1.1.2

Comment faire pour implémenter une *pile* par une liste simplement chaînée où les opérations *push* et *pop* se font en **fin de liste** ? Cette solution est-elle efficace ? Argumentez.

6.3.3 Exercice 1.1.3

Quelles sont les implémentations possibles pour une pile? En consultant la documentation sur l'API de Java, décrivez l'implémentation d'une pile par la classe *java.util.Stack*. Aller voir le code source de l'implémentation *java.util.Stack* (crtl+B depuis IntelliJ).

Pourquoi pensez-vous que les développeurs de Java ont choisi cette implémentation (hint: argumentez au niveau de la mémoire et du garbage collector)?

6.3.4 Exercice 1.1.4

Comment faire pour implémenter le type abstrait de données *Pile* à l'aide de deux *files* ? Décrivez en particulier le fonctionnement des méthodes *push* et *pop* dans ce cas.

A titre d'exemple, précisez l'état de chacune des deux files après avoir empilé les entiers *1 2 3* à partir d'une pile initialement vide. Décrivez ce qu'il se passe ensuite lorsque l'on effectue l'opération *pop*.

Quelle est la complexité temporelle de ces méthodes si l'on suppose que chaque opération *enqueue* et *dequeue* s'exécute en temps constant?

Cette implémentation d'une pile est-elle efficace (pour n opérations) par rapport aux autres implémentations présentées dans le livre de référence ?

6.3.5 Exercice 1.1.5

- Qu'est-ce qu'un iterateur en Java (*java.util.Iterator*)?
- Pourquoi est-ce utile de définir une méthode *iterator()* sur les structures de données?
- Que pensez vous de permettre la modification d'une structure de donnée alors qu'on est en train d'itérer sur celle-ci?

[1] *abstract data type* (ADT) en anglais

Pour vous aider dans la réflexion, nous vous invitons à lire la spécification de l'API Java concernant la méthode *remove()*.

Proposez une modification du code de l'iterateur de Stack qui lance une *java.util.ConcurrentModificationException* si le client modifie la collection avec un *push()* ou *pop()* durant l'itération. Est-ce une bonne idée de laisser l'implémentation de la méthode *remove()* vide si on ne désire pas permettre cette fonctionnalité?

6.3.6 Exercice 1.1.6

La notation $\sim$ (tilde) est utilisée dans le livre de référence pour l'analyse des temps de calcul des algorithmes. En quoi cette notation diffère ou ressemble aux notations plus classiquement utilisées $\mathcal{O}$ (big Oh), $\otimes$ (big Omega) et $\times$ (big Theta)?

Expliquez précisément les liens et similitudes entre celles-ci. Que voyez-vous comme avantage à utiliser la notation $\sim$ (tilde) plutôt que $\mathcal{O}$ lorsque c'est possible?

6.3.7 Exercice 1.1.7

Expliquez comment nous pouvons extraire la caractérisation $\sim$ (tilde) de l'implémentation d'un algorithme à l'aide du test *Doubling ratio*.

- Comment fonctionne ce test?
- Quelles sont les limites et avantages de ce test?

Supposont que nous mesurons les temps d'exécutions $T(n)$ suivants (en secondes) d'un programme en fonction de la taille de l'entrée n:

n	1000	2000	4000	8000	16000	32000	64000
$T(n)$	0	0	0.1	0.3	1.3	5.1	20.5

- Comment pouvez-vous caractériser au mieux l'ordre de croissance de cette fonction ?
- Que serait le temps d'exécution pour 128000?

6.4 Exercices théoriques supplémentaires

Note: Ces exercices ne seront pas forcéments résolus en cours, ils restent néanmoins intéressants. Si vous avez des problèmes avec ceux-ci, posez votre question lors d'un TP.

6.4.1 Exercice 1.1b.1

Que signifient les paramètres -Xmx, -Xms que l'on peut passer à la JVM pour l'exécution d'un bytecode? Est-ce que ces paramètres peuvent influencer la vitesse d'exécution d'un programme Java? Pourquoi?

6.4.2 Exercice 1.1b.2

- Qu'est-ce qu'un bon ensemble de tests unitaires pour vérifier l'exactitude d'une structure de données?
- Pensez-vous aux cas limites?
- Pensez-vous à la valeur maximale des entiers, doubles, etc?
- En quoi la génération de données aléatoire peut être utile pour tester les structures de données?
- Pourquoi est-ce important de travailler avec une semence (seed) fixée?
- En quoi un outil d'analyse de couverture de code peut être utile (tel que Jacoco) pour vous aidez à concevoir des tests.
- Comment vérifier expérimentalement que l'implémentation d'une structure de données ou un algorithme a bien la complexité temporelle théorique attendue ?

6.5 Exercices sur Inginious

Note: Vous devez faire ces exercices pour le mercredi de S3.

1. Ecriture de tests unitaires pour une stack
2. Implementation d'une stack avec structure chainée
3. Implementation d'une liste chainée circulaire et d'un iterateur

6.6 Exercices théorique: deuxième partie

Note: Vous devez faire ces exercices pour le mercredi de S3.

6.6.1 Exercice 1.2.1

Dans votre implémentation d'une liste chainée circulaire ci-dessous. Quelle est la complexité de la méthode

- *public void enqueue(Item item)*?
- *public Item remove(int index)* ?
- d'une séquence d'operations qui consiste à *créer un iterateur et ensuite itérer sur les k-premiers elements* ?

```
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class CircularLinkedList<Item> implements Iterable<Item> {
 private long nOp = 0; // count the number of operations
 private int n;          // size of the stack
 private Node  last;   // trailer of the list
```

(continues on next page)

(continued from previous page)

```
// helper linked list class
private class Node {
    private Item item;
    private Node next;
}

public CircularLinkedList() {
    last = new Node(); // dummy node
    last.next = last;
    n = 1;
}

public boolean isEmpty() { return n == 1; }

public int size() { return n-1; }

private long nOp() { return nOp; }

/**
 * Append an item at the end of the list
 * @param item the item to append
 */
public void enqueue(Item item) {
    // TODO STUDENT: Implement add method
}

/**
 * Removes the element at the specified position in this list.
 * Shifts any subsequent elements to the left (subtracts one from their indices).
 * Returns the element that was removed from the list.
 */
public Item remove(int index) {
    // TODO STUDENT: Implement remove method
}

/**
 * Returns an iterator that iterates through the items in FIFO order.
 * @return an iterator that iterates through the items in FIFO order.
 */
public Iterator<Item> iterator() {
    return new ListIterator();
}

/**
 * Implementation of an iterator that iterates through the items in FIFO order.
 *
 */
private class ListIterator implements Iterator<Item> {
    // TODO STUDENT: Implement the ListIterator
}

}
```

6.6.2 Exercice 1.2.2

La notation post-fixe (ou polonaise inverse) est utilisée pour représenter des expressions algébriques. Nous ne considérons pour simplifier que des expressions post-fixes avec des entiers positifs et les opérateurs + et *. Par exemple *2 3 1 * + 9 *** dont le résultat vaut 45 et le résultat de *4 20 + 3 5 1 * * + +** est 39.

1. Ecrivez un algorithme en Java pour évaluer une expression post-fixe au départ d'une chaine de n-caractères.
2. Quelle structure de donnée utilisez vous ?
3. Quelle est la complexité de votre algorithme (temporelle et spatiale) ?

Pour rappel, voici comment on peut itérer sur les elements d'une chaine qui sont séparés par des espaces.

```
String in = "4 20 + 3 5 1 * * +";
StringTokenizer tokenizer = new StringTokenizer(in);
while (tokenizer.hasMoreTokens()) {
     String element = tokenizer.nextToken();
}
```

6.6.3 Exercice 1.2.3

La programmation fonctionnelle est un paradigme de programmation de plus en plus important. Dans ce paradigme de programmation, les structures de données sont immutables . Nous nous intéressons ici à l'implémentation d'une liste immutable appelée *FList* permettant d'être utilisée dans un cadre fonctionnel. Voici l'API d'une *FList*

```
public abstract class FList<A> implements Iterable<A> {

    // creates an empty list
    public static <A> FList<A> nil();

    // prepend a to the list and return the new list
    public final FList<A> cons(final A a);

    public final boolean isNotEmpty();

    public final boolean isEmpty();

    public final int length();

    // return the head element of the list
    public abstract A head();

    // return the tail of the list
    public abstract FList<A> tail();

    // return a list on which each element has been applied function f
    public final <B> FList<B> map(Function<A,B> f);

    // return a list on which only the elements that satisfies predicate are kept
    public final FList<A> filter(Predicate<A> f);

    // return an iterator on the element of the list
    public Iterator<A> iterator();
```

(continues on next page)

(continued from previous page)

```
}
```

Comme vous pouvez vous en rendre compte, aucune des méthodes ne permet de modifier l'état de la liste. Voici un exemple de manipulation d'une telle liste. Si vous n'êtes pas familiers avec les interfaces fonctionnelles de Java8, nous vous demandons de vous familiariser d'abord avec celles-ci.

```
FList<Integer> list = FList.nil();

for (int i = 0; i < 10; i++) {
    list = list.cons(i);
}

list = list.map(i -> i+1);
// will print 1,2,...,11
for (Integer i: list) {
    System.out.println(i);
}

list = list.filter(i -> i%2 == 0);
// will print 2,4,6,...,10
for (Integer i: list) {
    System.out.println(i);
}
```

Voici une implémentation partielle de la *FList*

```
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.function.Function;
import java.util.function.Predicate;

public abstract class FList<A> implements Iterable<A> {

    public final boolean isNotEmpty() {
        return this instanceof Cons;
    }

    public final boolean isEmpty() {
        return this instanceof Nil;
    }

    public final int length() {
        // TODO
    }

    public abstract A head();

    public abstract FList<A> tail();

    public static <A> FList<A> nil() {
        return (Nil<A>) Nil.INSTANCE;
```

(continues on next page)

(continued from previous page)

```
    }

    public final FList<A> cons(final A a) {
        return new Cons(a, this);
    }

    public final <B> FList<B> map(Function<A,B> f) {
        // TODO
    }

    public final FList<A> filter(Predicate<A> f) {
        // TODO
    }


    public Iterator<A> iterator() {
        return new Iterator<A>() {
            // complete this class


            public boolean hasNext() {
              // TODO
            }

            public A next() {
              // TODO
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }


    private static final class Nil<A> extends FList<A> {
        public static final Nil<Object> INSTANCE = new Nil();
        // TODO
    }

    private static final class Cons<A> extends FList<A> {
        // TODO
    }


}
```

Nous vous demandons de

- compléter cette implémentation, si possible utilisez autant que possible des méthodes récursives.
- déterminer la complexité de chacune des méthodes.

6.7 Ressources supplémentaires

Notes de bas de page

CHAPTER

SEVEN

EXERCICES: RÉCAPITULATIF DES EXERCICES À FAIRE

7.1 Année 2021-2022

Master2 IFRI

7.1.1 Exercice 1

Réparti par groupe de deux, choisir un logiciel et donner un exemple de chaque sous critère de la norme ISO9126 en suivant le modèle du tableau suivant

7.1.2 Exercice 2

Reprendre l'exercice 1 et developper vos propres modèles pour évaluer chaque cas

7.1.3 Exercice 3

Reprendre l'exercice 2 n utilisant des métriques connues pour évaluer la qualité du logiciel. Si des difficultés à évaluer tous les aspects chercher et projet open-source.

7.1.4 Projet (1mois)

Conception d'un logiciel de veille citoyenne tout en tenant compte de qualité du produit final.