Théorie et Pratique de la Programmation
Examen intermédiaire

12 avril 2013

Indications :
- l'examen dure de 12h15 à 14h00,
- utilisez un ensemble de feuilles séparé pour chaque exercice,
- indiquez votre nom, prénom et numéro SCIPER sur chaque feuille que vous rendez,
- rendez chaque exercice dans la boîte correspondante,
- placez votre carte d'étudiant sur la table,
- n'écrivez rien sur l'énoncé et ne le rendez pas à la fin de l'examen,

Bon travail !

Index inversé [10 points]

Un *index inversé* est un type de collection utilisé pour accélérer la recherche textuelle dans un ensemble de documents. Un tel index associe à chaque mot existant dans au moins un document l'ensemble des documents contenant une ou plusieurs occurrences de ce mot. La figure 1 montre trois documents simples et leur index inversé.

Document	Contenu
D_1	it is what it is
D_2	what is it
D_3	it is a banana

(a) Trois documents

Mot	Documents
a	$\{D_3\}$
banana	$\{D_3\}$
is	$\{D_1, D_2, D_3\}$
it	$\{D_1, D_2, D_3\}$
what	$\{D_1, D_2\}$

(b) Leur index inversé

FIGURE 1 – Exemple d'index inversé

Pour cet exercice, les documents sont modélisés par l'interface `Document` suivante :

```
public interface Document {
    public Iterator<String> wordsIterator();
}
```

La méthode `wordsIterator` retourne un itérateur sur les mots du document, dans leur ordre d'apparition. L'interface `Iterator` est rappelée en annexe.

Partie 1 [10 points] Ecrivez une classe représentant les index inversés et implémentant l'interface suivante :

```
public interface InvertedIndex {
    public void addDocument(Document d);
    public Set<Document> documentsWithWord(String word);
}
```

La méthode `addDocument` ajoute le document reçu en argument à l'index. La méthode `documentsWithWord` retourne l'ensemble des documents contenant le mot reçu ; si ce mot n'existe dans aucun document, elle retourne un ensemble vide.

Notez que la construction de l'index doit absolument se faire dans la méthode `addDocument`, la méthode `documentsWithWord` se devant d'être aussi rapide que possible.

Vous avez bien entendu le droit d'utiliser les collections vues au cours pour stocker le contenu de l'index. Les interfaces de ces collections sont rappelées en annexe.

Pile par morceaux [15 points]

Une *pile* est une collection similaire à une liste, si ce n'est que l'insertion et la suppression d'éléments se fait toujours à la fin. L'interface suivante décrit une simple pile générique :

```
public interface Stack<E> {
    public int size();
    public void push(E newElem);
    public E pop();
}
```

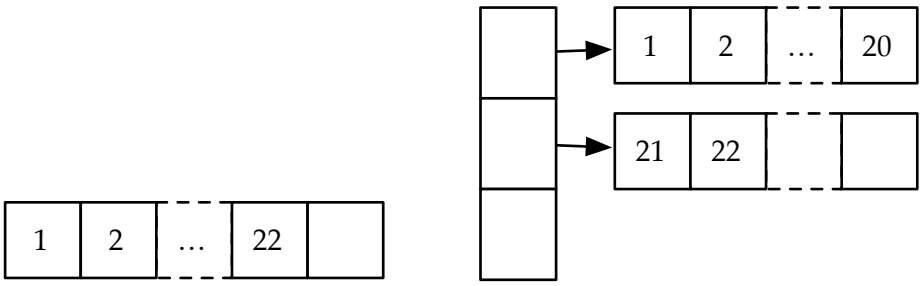
La méthode `size` retourne le nombre d'éléments stockés dans la pile ; `push` ajoute l'élément reçu en argument à la fin de la pile ; `pop` supprime et retourne le dernier élément ajouté ou lève l'exception `EmptyStackException` du paquetage `java.util` si la pile est vide.

Les piles étant une forme restreinte des listes, il est possible de les mettre en œuvre au moyen des mêmes techniques. Le but de cet exercice est d'utiliser une variante bidimensionnelle des tableaux-listes vus au cours.

Pour mémoire, la version unidimensionnelle des tableaux-listes stocke les éléments dans un tableau (dit *sous-jacent*) de taille fixe. Lorsque la capacité de ce tableau est dépassée, un nouveau tableau plus grand est créé et le contenu de l'ancien y est copié. Finalement, le nouveau tableau est utilisé en lieu et place de l'ancien.

La variante bidimensionnelle de cet exercice stocke les éléments de la pile dans une séquence de tableaux de taille fixe nommés *morceaux* (*chunks* en anglais). Les morceaux eux-même sont stockés dans un tableau-liste standard. Comparée à la version unidimensionnelle, cette organisation bidimensionnelle ralentit l'accès aux éléments en ajoutant une indirection mais accélère le redimensionnement du tableau-liste, car les morceaux eux-même n'ont pas besoin d'être copiés lorsque le tableau sous-jacent est copié.

La figure 2 montre ces deux mises en œuvre pour une pile contenant 22 éléments. La mise en œuvre bidimensionnelle utilise des morceaux d'une taille de 20 éléments.



(a) Tableau-liste unidimensionnel (b) Tableau-liste bidimensionnel

FIGURE 2 – Deux mises en œuvre d'une pile de 22 éléments

Partie 1 [15 points] Ecrivez une classe `ChunkedStack` implémentant l'interface `Stack` définie plus haut et utilisant la variante bidimensionnelle des tableaux-listes décrite ci-dessus.

Utilisez un tableau-liste de type `ArrayList` pour stocker les morceaux. L'interface `List` implémentée par `ArrayList` est rappelée en annexe. Les morceaux eux-même doivent être de simples tableaux Java, puisqu'ils ne sont pas redimensionnables. Leur taille doit être donnée par une variable privée, statique et finale de la classe que vous pouvez p.ex. initialiser à 100.

Votre mise en œuvre ne doit jamais stocker un morceau totalement vide, et tous les éléments inutilisés d'un morceau doivent toujours contenir `null`.

Graphe orienté acyclique [20 points]

Un *graphe orienté* est une structure formée d'un ensemble de *nœuds* et d'un ensemble d'*arcs* orientés reliant ces nœuds entre eux. La figure 3 présente un tel graphe composé de 7 nœuds — numérotés de 1 à 7 et représentés par des disques — et de 7 arcs — représentés par des flèches.

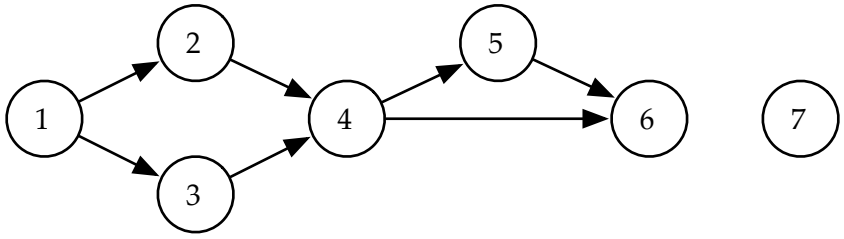


FIGURE 3 – Graphe orienté acyclique

L'arc allant d'un nœud x à un nœud y est noté $x \rightarrow y$. Il ne peut pas y avoir plus d'un arc allant d'un nœud x à un nœud y, et les arcs allant d'un nœud à lui-même (de la forme $x \rightarrow x$) sont interdits.

Dans un graphe orienté, un *chemin* est une séquence, éventuellement vide, d'arcs consécutifs. Par exemple, $[1 \rightarrow 2, 2 \rightarrow 4, 4 \rightarrow 5]$ est un chemin du graphe ci-dessus. Dans ce même graphe, il n'existe aucun chemin allant du nœud 6 au nœud 1. Il en existe par contre plusieurs menant du nœud 1 au nœud 6. Un *cycle* est un chemin non vide menant d'un nœud à lui-même.

Les *successeurs* d'un nœud x sont les nœuds atteignables depuis x via un chemin de longueur 1. Dans le graphe ci-dessus, les successeurs du nœud 1 sont les nœuds 2 et 3. Les nœuds 6 et 7 n'ont pas de successeurs.

Le graphe de la figure 3 ne contient aucun cycle et est donc dit *acyclique*. Les graphes orientés acycliques (*directed acyclic graph*, abrégé *DAG*, en anglais) sont une catégorie très importante de graphes. En Java, on peut les représenter par un ensemble de nœuds et un ensemble d'arcs. Le classe `DAG` ci-dessous, générique et paramétrisée par le type `N` des nœuds, utilise une telle représentation.

```
public class DAG<N> {
    private Set<N> nodes = new HashSet<>(); // nœuds
    private Set<Edge<N>> edges = new HashSet<>(); // arcs

    public void addNode(N n) { nodes.add(n); }
    public void removeNode(N n) { à faire }
    public void addEdge(N f, N t) { à faire }
    public boolean isReachable(N f, N t) { à faire }
    public Set<N> successors(N n) {
       code omis pour faciliter la lecture
    }
}
```

Les arcs sont modélisés par la classe `Edge` ci-dessous. Cette classe aurait pu être imbriquée statiquement dans la classe `DAG`, mais cela n'a pas été fait pour faciliter la présentation.

```
class Edge<N> {
    final N fromNode, toNode;

    public Edge(N fromNode, N toNode) {
        this.fromNode = fromNode;
        this.toNode = toNode;
    }

    @Override
    public int hashCode() { à faire }

    private boolean equalsEdge(Edge<N> that) { à faire }
    @Override
    public boolean equals(Object o) {
        return (o instanceof Edge)
            && equalsEdge((Edge<N>)o);
    }
}
```

Partie 1 [2 points] Complétez la classe `Edge` en écrivant les deux méthodes `hashCode` et `equalsEdge`. La seconde est utilisée par la méthode `equals` dans le cas où cette dernière reçoit un arc en argument.

La méthode `equalsEdge` doit retourner vrai pour deux arcs e_1 et e_2 si et seulement si le nœud de départ de e_1 est égal (au sens de `equals`) au nœud de départ de e_2 et le nœud d'arrivée de e_1 est égal au nœud d'arrivée de e_2.

La méthode `hashCode` doit être compatible avec la méthode `equals` et sa valeur doit dépendre de la valeur de hachage de chacun des deux nœuds reliés par l'arc.

Partie 2 [5 points] Ecrivez la méthode `removeNode` de la classe `DAG`. Comme son nom l'indique, elle doit supprimer du graphe le nœud reçu en argument. Si le nœud n'existe pas dans le graphe, elle ne doit rien faire.

Attention : n'oubliez pas de mettre à jour l'ensemble des arcs pour en supprimer tous ceux référençant le nœud supprimé.

Partie 3 [5 points] Ecrivez la méthode `addEdge` de la classe `DAG` qui ajoute un arc allant du premier nœud reçu au second. Elle doit lever l'exception `IllegalArgumentException` si l'ajout de cette arête introduirait un cycle dans le graphe ou si l'un des deux nœuds n'appartient pas au graphe.

Conseil : aidez-vous de la méthode `isReachable` définie à la partie 4 ci-dessous, que vous pouvez admettre exister pour cette partie.

Partie 4 [5 points] Ecrivez la méthode `isReachable` de la classe `DAG`. Cette méthode retourne vrai ssi le second nœud est atteignable depuis le premier, c-à-d s'il existe un chemin reliant le premier au second. Comme dit plus haut, un chemin peut être vide, c-à-d qu'un nœud est atteignable depuis lui-même !

Cette première mise en œuvre de `isReachable` doit être récursive et ne peut pas utiliser de méthode auxilliaire. Pour obtenir les successeurs d'un nœud, vous pouvez utiliser la méthode `successors` déclarée plus haut, sans devoir écrire son code.

Partie 5 [1 point] Combien de fois la méthode `isReachable` que vous venez de définir visite le nœud 6 dans le graphe de la figure 3 si on l'applique aux arguments 1 et 7 ?

Partie 6 [2 points] Ecrivez une nouvelle version de la méthode `isReachable` qui évite de visiter un nœud plus d'une fois. Pour ce faire, vous pouvez stocker les nœuds déjà visités dans une collection de votre choix, et vous avez cette fois l'autorisation de définir et d'utiliser des méthodes auxilliaires.

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque standard Java dont vous avez besoin pour cet examen. Notez que de nombreuses méthodes inutiles ont été omises afin de ne pas encombrer la présentation.

Interface `List`

L'interface `java.util.List` représente les listes. Elle est implémentée, entre autres, par les classes `LinkedList` (listes chaînées) et `ArrayList` (tableaux-listes).

```
interface List<E> {
    // Ajoute l'élément e à la fin de la liste et retourne vrai.
    boolean add(E e);

    // Retourne l'élément à l'index index ou lève l'exception
    // IndexOutOfBoundsException si cet index est invalide.
    E get(int index);

    // Supprime et retourne l'élément à l'index index ou lève l'exception
    // IndexOutOfBoundsException si cet index est invalide.
    E remove(int index);
}
```

Interface `Set`

L'interface `java.util.Set` représente les ensembles. Elle est implémentée, entre autres, par la classe `HashSet`.

```
interface Set<E> {
    // Ajoute l'élément e à l'ensemble et retourne vrai ssi
    // il ne s'y trouvait pas déjà.
    boolean add(E e);

    // Supprime l'élément e de l'ensemble et retourne vrai ssi
    // il s'y trouvait effectivement.
    boolean remove(E e);

    // Retourne vrai ssi l'ensemble contient l'élément e.
    boolean contains(E e);

    // Retourne un itérateur sur les éléments de l'ensemble.
    Iterator<E> iterator();
}
```

(Suite au verso)

Interface `Map`

L'interface `java.util.Map` représente les tables associatives. Elle est implémentée, entre autres, par la classe `HashMap`.

```
interface Map<K,V> {
    // Associe la valeur value à la clef key. Retourne la valeur
    // qui était associée à la clef, ou null s'il n'y en avait pas.
    V put(K key, V value);

    // Retourne la valeur associée à key, ou null s'il n'y en a aucune.
    V get(K key);

    // Retourne vrai si et seulement si la table contient la clef key.
    boolean containsKey(K key);
}
```

Interface `Iterator`

L'interface `java.util.Iterator` représente les itérateurs.

```
interface Iterator<E> {
    // Retourne vrai ssi cet itérateur peut encore livrer des éléments.
    boolean hasNext();

    // Retourne le prochain élément, ou lève l'exception
    // NoSuchElementException s'il n'y en a plus.
    E next();

    // Supprime la valeur retournée par le dernier appel à next.
    void remove();
}
```