UEF4.3. Programmation Orientée Objet

EXAMEN FINAL

02 heures-Documents interdits

Questions (6 pts)

1. Expliquez le concept de polymorphisme et ses différents types.
2. Quelle est la différence entre un programme correcte et un programme robuste ? Quel est le moyen le plus efficace et le plus "propre" pour rendre un programme java plus robuste ?
3. En java, comment compiler et exécuter un programme en mode console ? donnez un exemple.
4. Expliquez chacun des mots dans la ligne suivante

```
Public static void main (String [] args)
```

Exercice (14 pts) Modélisation d'un graphe non orienté pondéré

Un graphe non orienté G = (S, A) est défini par un ensemble de sommets S et un ensemble d'arêtes A. Un sommet est un nœud du graphe tandis qu'une arête est une ligne reliant deux sommets du graphe. Une seule arête peut relier deux sommets et deux sommets reliés par une arête sont dits voisins (ou adjacents).

Dans certaines applications, on associe à chaque arête un poids (le graphe est dit, alors, pondéré). Dans un réseau routier, par exemple, les sommets peuvent représenter les villes, une arête correspond au tronçon routier entre deux villes voisines et le poids de l'arête correspond à la longueur du tronçon..

Dans notre modélisation, nous associons à chaque sommet s une étiquette et l'ensemble de ses voisins. Un voisin est un couple (s', p) où s' est un sommet et p est un entier correspondant au poids de l'arête reliant s et s'.

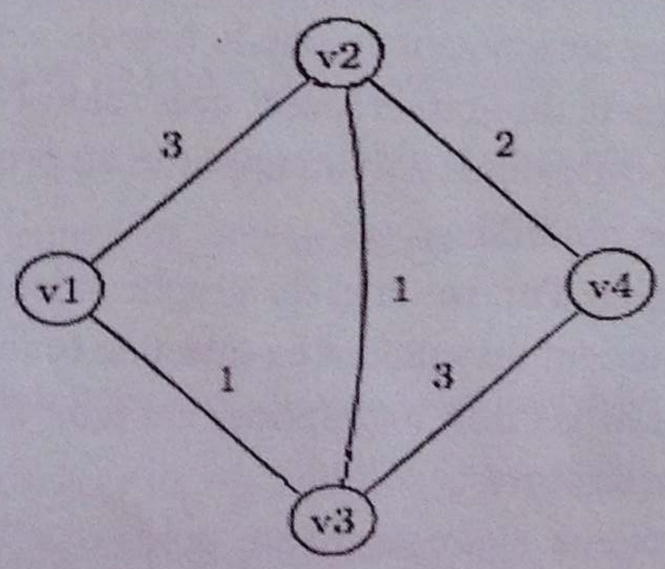

Figure 1. Exemple d'un graphe non orienté pondéré

Dans notre modélisation, nous utiliserons trois classes : la classe Voisin, la classe Sommet et la classe Graphe.

A. Ecrire, en respectant le principe d'encapsulation, le code java des trois classes en sachant que :

1. La classe `Voisin` doit comporter un constructeur initialisant ses attributs.
2. La classe `Sommet` a comme attributs une étiquette (de type String) et une collection non triée contenant ses voisins (choisissez le type de collections le plus adéquat). Cette classe doit fournir les méthodes publiques suivantes :
   - Un constructeur `Sommet (String etiquette)`
   - Une méthode permettant d'ajouter un voisin en donnant comme arguments un sommet et un poids
   - Une méthode permettant de vérifier si un sommet donné est un voisin
   - Une méthode permettant de parcourir la liste des voisins à l'aide d'un `iterator` et d'afficher leurs étiquettes
   - Une méthode qui retourne le degré d'un sommet qui est égal à la somme des poids des arêtes dont le sommet en question est l'une des extrémités. Dans l'exemple donné en figure 1, le degré du sommet v1 est égal à (3+1) = 4 et celui du sommet v3 est égal à (1+1+3) = 5
3. La classes Graphe comporte une table associative non triée ayant comme valeurs les sommets du graphe et comme clés les étiquettes correspondantes. Elle offre les méthodes publiques suivantes :
   - Une méthode permettant de chercher un sommet par son étiquette (elle renvoie null si le sommet n'existe pas)
   - Une méthode permettant d'ajouter une arête en donnant le sommet de départ, le sommet d'arrivée et le poids de l'arête. Cette méthode lance une exception de type `SommetException` si au moins un des deux sommets n'existe pas.
   - Une méthode permettant d'afficher les étiquettes des voisins de tous les sommets du graphe

B. Pour tester votre programme, écrivez une classe TestGraphe comportant une méthode main qui :

- Crée un graphe identique à celui donné en figure 1comme suit : d'abord les sommets v1, v2, v3 et v4 sont créés puis les arêtes reliant les sommets sont ajoutées.
- Affiche pour chaque sommet du graphe la liste de ses voisins.

C. Si l'on veut que les sommets du graphe soient triés selon l'ordre alphabétique des étiquettes des sommets, quelles modifications doit on apporter au programme ? Expliquez en donnant les parties du programme modifiées.

D. Si l'on veut que les voisins d'un sommet du graphe soient triés selon l'ordre croissant des poids des arêtes qui les relient au sommet en question (exemple, les voisins de v2 sont : v3, v4, v1), quelles modifications doit on apporter au programme ? expliquez en donnant les parties du programme nécessaires.

Remarque : pour les voisins ayant le même poids d'arête, c'est l'ordre alphabétique des étiquettes qui sera respecté. Exemple : les voisins du sommet v3 sont : v1, v2, v4.

NB : la lisibilité de votre code sera prise en considération. Une copie non soignée ne sera pas corrigée.

Annexe

A. Opérations basiques de l'interface java.util.Collection<E> :

- size: retourne le nombre d'éléments dans la collection
- isEmpty(): retourne true si la collection est vide
- contains (Object o) : retourne true si la collection l'objet o (basé sur la méthode equals de la classe E)
- add (E o): ajoute l'élément o à la collection. Retourne false si doubles interdits
- remove(Object object): supprime l'objet o
- iterator(): retourne l'itérateur de la collection

B. Opérations basiques de l'interface java.util.Iterator<E>

- boolean hasNext(): retourne true s'il reste au moins un élément à parcourir dans la collection
- E next(): renvoie l'élément courant dans le parcours et passe l'itérateur à l'élément suivant
- void remove(): supprime le dernier élément parcouru (next doit être appelé avant)

C. Opération basiques de l'interface Map<K,V>

- V put(K k, V v) : ajouter v avec la clé k : retourne l'ancienne valeur associée à la clé si la clé existait déjà
- V get(Object k) : retourne la valeur associée à la clé ou null
- V remove(Object k) : supprimer l'entrée associée à k
- boolean containsKey(Object k) : retourne true si la clé k est utilisée
- boolean containsValue(Object v) : retourne true si v existe dans la map
- Set<Map.Entry<K, V>> entrySet() Récupère les entrées (paires clé-valeur) sous forme de Set<Map.Entry<K,V>>
- Collection<V> values() Récupère les valeurs sous forme de Collection<V>
- Set<K> keySet() Récupère les clés sous forme de Set<K>
- int size() : retourne le nombre d'entrées dans la table
- boolean isEmpty() : retourne true si la map est vide
- void clear() : vide la map