TP1 - Création d'une classe Graphe

Dans ce premier TP, on souhaite créer une classe Graphe que l'on pourra utiliser pour des graphes orientés, sans coût associés aux arcs.
Pour créer cet objet, on peut procéder en deux temps.
Tout d'abord on va créer une classe Sommet, dont les attributs seront :

- **nom :** Le nom du sommet qui pourra être mis en paramètre lors de la création de l'objet Sommet.
- **successeurs :** La liste de ses successeurs.

Les listes utilisées seront des listes chaînées de la classe LinkedList afin de pourvoir rajouter et supprimer des éléments.
Nous verront dans un second temps, comment gérer les coûts des arcs.
Voici les principales méthodes associées a la classe LinkedList :
(https ://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) :

- **addFirst() :** Ajoute un élément au début de la liste.
- **addLast() :** Ajoute un élément a la fin de la liste.
- **add(Objet o) :** Ajoute l'élément o a la fin de la liste.
- **removeFirst() :** Supprime le premier élément de la liste.
- **removeLast() :** Supprime le premier élément de la liste.
- **remove(Objet o) :** supprime la première occurrence de l'objet o.
- **getFirst() :** Renvoie le premier élément de la liste.
- **getLast() :** Renvoie le dernier élément de la liste.
- **contain(Objet o) :** Renvoie le booléen "true" si la liste contient l'objet o.
- **size() :** Renvoie un entier correspondant à la taille de la liste
- ....

Compléter la structure suivante pour la classe Sommet permettant de créer des objets de type Sommet :

```
1  import java.util.LinkedList;
2
3  public class Sommet {
4      private String ..... ;
5      private LinkedList<Sommet> ..... ;
6
7      //Constructeur
8      public Sommet(String nom) {
9          this.nom = ..... ;
10         this.successeurs = ..... ;
11     }
```

Notre classe ayant été muni de son constructeur, il nous reste à lui ajouter les méthodes permettant d'accéder, de rajouter, de modifier .... les attributs.
Tout d'abord,la méthode ajouterSuccesseur aura en paramètre un sommet de la classe Sommet. Cette méthode ne renverra rien et ne fera qu'ajouter l'élément Sommet à la listes chaînées correspondantes.

```
1     public void ajouterSuccesseur(........) {
2         ................... ;
3     }
```

On va ensuite rajouter une méthode permettant directement de rajouter plusieurs successeurs à partir d'une liste de sommets qui sera en variable de la méthode.

```
1     public void ajouterListeSuccesseurs(Sommet[] listeSommet) {
2         for(............ : ............) {
3             ........................ ;
4         }
5     }
```

On va maintenant rajouter 2 méthodes à notre classe.

- **getNom() :** permet de récupérer le nom du sommet.
- **getSuccesseur() :** permet de récupérer la liste des successeurs du sommet associé à l'objet.

Vous trouverez ci dessous la structure de la méthode **getNom()**/

```
1     public String getNom() {
2         return .......... ;
3     }
```

Une dernière méthode permettra d'afficher l'objet, avec son nom et la liste de ses successeurs.

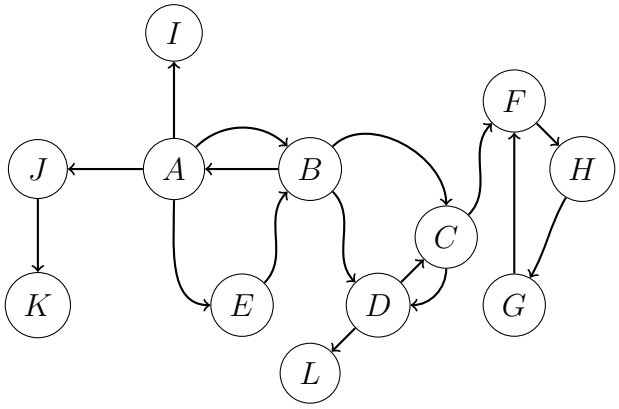
On peut maintenant créer une classe Graphe permettant de créer un objet de type Graphe dont l'attribut sera une liste d'éléments de type Sommet.
La encore, on choisira une liste chaînée de la classe LinkedList d'éléments de type Sommet comme dans le code ci-dessous.
Cette méthode devra être munie des méthodes suivantes :

- **ajouter(Sommet s) :** Permet d'ajouter un sommet s au graphe.
- **ajouterListeSommet(Sommet[] liste) :** Permet d'ajouter directement une liste de sommets s au graphe.
- **get(int index) :** Retourne le sommet dont l'index est "index" dans la liste des sommets.
- **getSommetParNom(String s) :** Retourne la première occurrence du sommet ayant pour nom s.

- **getIndex(Sommet s) :** Retourne l'index du sommet s.
- **size() :** Retourne la taille de la liste, c'est à dire l'ordre du graphe -1.
- **afficher() :** Affiche la liste des sommets du graphe avec leur successeurs.

```
1   import java.util.LinkedList;
2
3   public class Graphe {
4       ..............
5       ..............
6   }
```

Vous testerez vos méthodes sur le graphe suivant :



TP2 - Exploration d'un graphe Graphe

L'objectif de ce TP est de compléter les méthodes de la classe Graphe, vue dans le TP1, qui permettront d'explorer un graphe, puis d'afficher le résultat de l'exploration.

1. Détermination de la liste des descendants $\widehat{\Gamma}^+(s)$
   (a) Pour cela, on va compléter la classe Sommet avec un attribut booléen "marque", qui par défaut sera initialisé à "false".
   Cet attribut permettra de marquer les sommets visités, comme on l'a vu dans les algorithmes du cours.
   Il faudra aussi ajouter les méthodes permettant de récupérer l'état (marqué ou non) du sommet ainsi que celle permettant de modifier ce dernier ("setMarque()" et "getMarque()").
   (b) Dans un second temps, écrire une méthode itérative de la classe graphe permettant d'explorer le graphe en profondeur a partir d'un Sommet s et d'afficher la liste des sommets visités, c'est à dire la liste des descendants de s, $\left(\widehat{\Gamma}^+(s)\right)$
   (c) Réaliser une seconde méthode itérative permettant une exploration du graphe en largeur pour obtenir $\widehat{\Gamma}^+(s)$.
   (d) Ecrire une dernière méthode permettant l'exploration en profondeur d'un graphe et l'affichage du résultat de manière récursive.
2. Les explorations précédentes ne permettent pas de déterminer la composantes fortement connexe $\widehat{\Gamma}(s)$ d'un graphe orienté. Pour cela, il faudrait comme on l'a vu dans le cours, définir le graphe à l'aide des sommets et de la liste de leurs prédécesseurs. Cette liste peut être directement déterminée dans la classe Sommet créée dans le TP1.
   (a) Créer une nouvelle liste LinkedList<Sommet> ajoutée en attribut de la classe sommet.
   (b) Modifier la méthode AjouterSuccesseurs(Sommet s) pour ajouter le sommet créé comme prédesseur du sommet s.
   (c) Créer une méthode getPredesseurs() pour récupérer la liste des prédécesseurs.
   (d) Modifier la méthode afficher() pour permettre d'afficher la liste des prédécesseurs en plus de l'affichage des successeurs.
   (e) Modifier la classe graphe en ajoutant une méthode permettant d'obtenir et d'afficher la composante fortement connexe associée à un sommet s d'un graphe.

TP3 - Coloration d'un graphe Graphe

L'objectif de ce TP est créer des méthodes permettant la coloration d'un graphe. Pour cela, vous pourrez le faire en deux temps.

- Modifier la classe Sommet pour rajouter les attributs couleur et Dsat qui donneront la couleur du sommet et son niveau de saturation qui correspond au nombre de ses sommets déjà coloriés. Il faudra bien sur ajouter les méthodes "getCouleur()" et "getDsat()" pour pouvoit récupérer les attributs couleur et Dsat, ainsi que les méthodes "setCouleur()" et "setDsat()" pour les mettre a jour.
- Dans un second temps, vous aller créer trois méthodes dans la classe graphe pour de colorer les sommets du graphe en mettant à jour l'attribut "couleur" des sommets.
  - La première méthode sera privée et renverra le sommet à colorier.
  - La seconde méthode sera aussi privée et aura en variable le sommet à colorier. Elle renverra la couleur à utiliser pour le colorer.
  - La dernière méthode sera public. Elle appellera les deux méthodes précédentes, coloriera le graphe et affichera le résultat.

  Il ne faudra pas oublier de mettre à jours les attributs Dsat des sommets après la coloration.

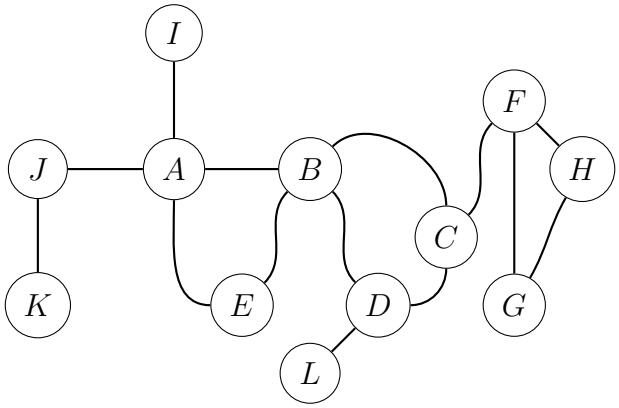
Vous utiliserez l'algorithme DSATUR qui est plus efficace que Welsh-Powell, car **adaptatif** : il considère, en plus de degré des sommets, leur "degré de saturation", qui évolue donc au fil de l'exécution de l'algorithme.

Algorithme 1 : Algorithme DSATUR.

tant que *il existe un sommet non colorié* **faire**
  trouver un sommet non colorié s avec $DSAT(s)$ maximal (en cas d'égalité, choisir un sommet avec $deg(s)$ maximal);
  colorier s avec la plus petite couleur possible k;
  pour chaque x *voisin non colorié de* s **faire**
    actualiser $DSAT(x)$;
  fin
fin

Vous testerez votre programme sur le graphe suivant :

8



TP4 - Graphe Cordal

L'objectif de ce TP est créer des méthodes permettant de déterminer si un graphe est cordal ou non. Pour cela vous pourrez créer deux méthodes de la classe graphe avec les objectifs suivant :

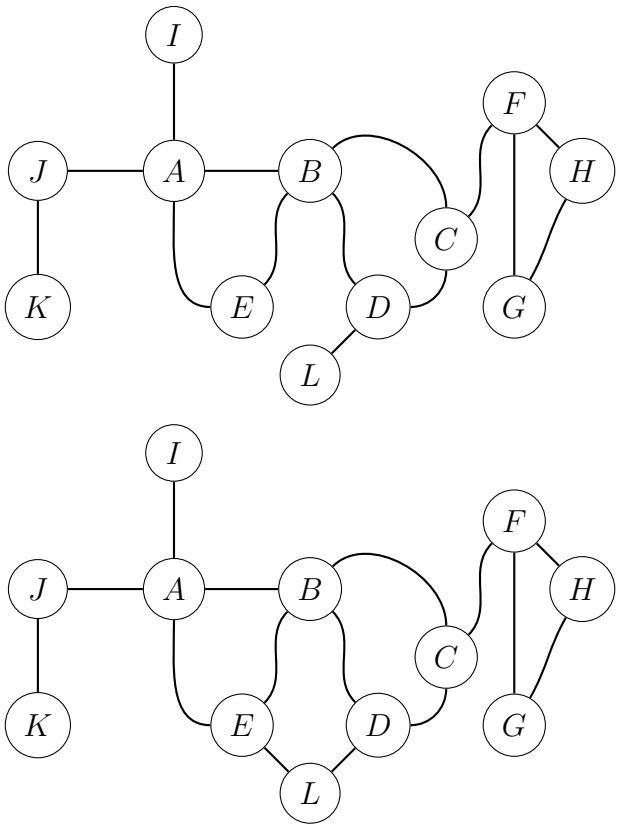
- La première méthode sera privé et devra permettre de déterminer si le Sommet s, donné en paramètre, est simplicial dans le sous graphe associé à une liste de sommets qui lui sera donné elle aussi en paramètre.
- La seconde méthode, déterminera si un graphe est cordal ou non et affichera le résultat.

l'algorithme général est le suivant :

Algorithme 2 : Reconnaissance d'un graphe triangulé de Fulkerson et Gross.

$G' \leftarrow G$;
tant que *il existe un sommet simplicial dans* G' **faire**
  retirer un sommet simplicial de G'
fin
si $G' == \emptyset$ **alors**
  G est triangulé
fin

Vous testerez votre programme sur les deux graphes suivants



TP5 - Graphe Dijkstra

L'objectif de ce TP est créer des méthodes permettant de déterminer le plus court chemin, dans un graphe orienté, en utilisant l'algorithme de Dijkstra.
Pour cela, on va créer de nouveaux attributs de la classe sommet qui permettront de stocker le coût du plus court chemin d'un sommet s au sommet objet de la classe ainsi que son prédécesseur dans ce plus court chemin.
Implémenter aussi les méthodes permettant d'accéder à ces nouveaux attributs ainsi que de les modifier.
On implémentera ensuite une méthode de la classe graphe permettant de déterminer le plus court chemin entre deux sommets selon l'algorithme de Dijkstra ci-dessous.
Dans cet algorithme, L représente la liste des sommets x pour lesquels on peut encore à priori diminuer la valeur de $d(x)$ et D représente la liste des distances minimales de s aux autres sommets de $\widehat{\Gamma}^+(s)$
On rajoute le sommet x de L ayant la plus petite distance au sommet initial s au tableau D des distances minimales, puis on relâche les successeurs de s n'appartenant pas à D.
On supprime ensuite le sommet x de L et on recommence.

Algorithme 3 : Algorithme de Dijkstra.

$L(s) \leftarrow 0$;
tant que $L \neq \emptyset$ **faire**
  trouver un sommet x de L tel que $d(x)$ soit minimale;
  $D(x) \leftarrow d(x)$;
  pour chaque $y \in \Gamma^+(x)$ *n'appartenant pas à* D **faire**
    relâcher (x,y)
  fin
  $L \leftarrow L - \{x\}$
fin

L'algorithme de Dijkstra utilise la procédure relâcher ci-dessous.
L'idée générale est de stocker les sommets y pour lesquels on peut encore diminuer leur distance $d(y)$ dans la liste L, puis de diminuer progressivement les valeurs de $d(y)$ en examinant chaque arc (x, y) : si $d(y) > d(x) + c(x, y)$ alors on peut améliorer $d(y)$ en passant par x...

Vous testerez votre programme sur le graphe suivant

Algorithme 4 : Procédure : relâcher(x, y)**.**

si $d(y) > d(x) + c(x, y)$ **alors**
  $d(y) \leftarrow d(x) + c(x, y)$;
  $\pi(y) \leftarrow x$;
fin