TP1 - Correction

Vous trouverez ci-dessous la classe Sommet.

```java
import java.util.LinkedList;

public class Sommet {
    private String nom ;
    private LinkedList<Sommet> successeurs ;

    public Sommet(String nom) {
        this.successeurs = new LinkedList<Sommet>() ;
        this.nom = nom ;
    }

    public void ajouterSuccesseur(Sommet s) {
        this.successeurs.add(s) ;
    }

    public void ajouterListeSuccesseurs(Sommet[] listeSommet) {
        for(Sommet s : listeSommet) {
            this.ajouterSuccesseur(s) ;
        }
    }

    public String getNom() {
        return this.nom ;
    }

    public LinkedList<Sommet> getSuccesseurs(){
        return this.successeurs ;
    }

    public void afficher() {
        System.out.println(getNom() + (" : "));
        System.out.print(" Successeurs : ") ;
        for(Sommet s : this.successeurs) {
            System.out.print(s.getNom() + " ");
        }
    }
}
```

La classe Graphe ci dessous permet de crÃ©er un graphe qui a en attribut une liste de sommets.

```java
import java.util.LinkedList;

public class Graphe {
    private LinkedList<Sommet> listeSommet ;
```

```
5
6       public Graphe() {
7           this.listeSommet = new LinkedList<Sommet>() ;
8       }
9
10      public void ajouter(Sommet s) {
11          this.listeSommet.add(s) ;
12      }
13
14      public void ajouterListeSommets(Sommet[] liste) {
15          for(Sommet s : liste) {
16              this.ajouter(s);
17          }
18      }
19
20      public Sommet get(int index) {
21          return this.listeSommet.get(index) ;
22      }
23
24      public Sommet getSommetParNom(String nom) {
25          for(Sommet s : this.listeSommet) {
26              if(s.getNom() == nom) {
27                  return s ;
28              }
29          }
30          return null ;
31      }
32
33      public int getIndex(Sommet s) {
34          for(int i=0; i < this.listeSommet.size(); i++) {
35              if(s.getNom() == liste.get(i).getNom()) {
36                  return i ;
37              }
38          }
39          return -1 ;
40      }
41
42      public LinkedList<Sommet> getSommets(){
43          return this.listeSommet ;
44      }
45
46      public int size() {
47          return this.listeSommet.size();
48      }
49
50      public void afficher() {
51          for(Sommet s : this.listeSommet) {
52              s.afficher();
53              System.out.println();
54          }
55      }
56  }
```

Pour tester ses deux classes, vous pourrez utiliser le code suivant.

```
1   public class test {
2
3       public static void main(String[] args) {
```

```
4           Graphe gr = new Graphe() ;
5
6           Sommet a = new Sommet("A");
7           Sommet b = new Sommet("B");
8           Sommet c = new Sommet("C");
9           Sommet d = new Sommet("D");
10          Sommet e = new Sommet("E");
11          Sommet f = new Sommet("F");
12          Sommet g = new Sommet("G");
13          Sommet h = new Sommet("H");
14          Sommet i = new Sommet("I");
15          Sommet j = new Sommet("J");
16          Sommet k = new Sommet("K");
17          Sommet l = new Sommet("L");
18
19          a.ajouterListeSuccesseurs(new Sommet[] {b , e , i, j});
20          b.ajouterListeSuccesseurs(new Sommet[] {a , c , d});
21          c.ajouterListeSuccesseurs(new Sommet[] {d , f});
22          d.ajouterSuccesseur(c);
23          e.ajouterSuccesseur(b);
24          f.ajouterSuccesseur(h);
25          g.ajouterSuccesseur(f);
26          h.ajouterSuccesseur(g);
27          j.ajouterSuccesseur(k);
28
29          gr.ajouterListeSommets(new Sommet[] {a , b , c , d , e , f , g , h , i , j ,
        k , l});
30
31          gr.afficher();
32      }
33  }
```

TP2 - Correction

```java
import java.util.LinkedList;

public class Sommet {
    private String nom ;
    private LinkedList<Sommet> successeurs ;
    private LinkedList<Sommet> predecesseurs ;
    private boolean marque = false ;

    public Sommet(String nom) {
        this.successeurs = new LinkedList<Sommet>() ;
        this.predecesseurs = new LinkedList<Sommet>();
        this.nom = nom ;
    }

    public void ajouterSuccesseur(Sommet s) {
        this.successeurs.add(s) ;
        s.predecesseurs.add(this) ;
    }

    public void ajouterListeSuccesseurs(Sommet[] listeSommet) {
        for(Sommet s : listeSommet) {
            this.ajouterSuccesseur(s) ;
        }
    }

    public String getNom() {
        return this.nom ;
    }

    public void setMarque(boolean marque) {
        this.marque = marque ;
    }

    public boolean getMatque() {
        return this.marque ;
    }

    public LinkedList<Sommet> getSuccesseurs(){
        return this.successeurs ;
    }

    public LinkedList<Sommet> getPredecesseurs(){
        return this.predecesseurs ;
    }

    public void afficher() {
        //System.out.print(getNom() + (" (") + etat() + (")") + (" : "));
```

```
48          System.out.println(getNom() + (" : "));
49          System.out.print(" Successeurs : ") ;
50          for(Sommet s : this.successeurs) {
51              System.out.print(s.getNom() + " ");
52          }
53          System.out.print(" Prédécesseurs : ") ;
54          for(Sommet s : this.predecesseurs) {
55              System.out.print(s.getNom() + " ");
56          }
57      }
58  }
```

La méthode suivante permet l'exploration du graphe en profondeur à partir du sommet s et de manière itérative.

```
1       public void parcoursProfondeurIteratif(Sommet s) {
2           LinkedList<Sommet> liste = new LinkedList<Sommet>() ;
3           liste.addLast(s);
4           while(liste.size() != 0) {
5               Sommet x = liste.removeLast();
6               x.setMarque(true);
7               System.out.print(x.getNom() + " ");
8               for(Sommet som : x.getSuccesseur()) {
9                   if (!som.getMarque() && !liste.contains(som)) {
10                      liste.addLast(som);
11                  }
12              }
13          }
14      }
```

La méthode suivante permet l'exploration du graphe en largeur à partie du sommet s et de manière itérative.
l'ordre d'exploration des sommets du graphe est donc différente de la méthode précédente.

```
1       public void parcoursLargeurIteratif(Sommet s) {
2           LinkedList<Sommet> liste = new LinkedList<Sommet>() ;
3           liste.addLast(s);
4           s.setMarque(true) ;
5           System.out.print(s.getNom() + " ");
6           while(liste.size() != 0) {
7               Sommet x = liste.removeFirst();
8               for(Sommet som : x.getSuccesseur()) {
9                   if (!som.getMarque()) {
10                      som.setMarque(true);
11                      System.out.print(som.getNom() + " ");
12                      liste.addLast(som);
13                  }
14              }
15          }
16      }
```

Vous trouverez ci dessous une méthodes de la classe Graphe permettant une exploration en profondeur à partir d'un sommet s, de manière récursive. son objectif est de marqué et d'afficher les sommets

visités.
Cette méthode peut être très coûteuse en terme de mémoire et générera une erreur de dépassement de pile ("Stack overflow) si le nombre d'arêtes du graphe est trop important.

```
1     public void parcoursProfondeurRecursif(Sommet s) {
2         if(!s.etat()) {
3             s.setMarque(true);
4             System.out.print(s.getNom() + " ");
5             for(Sommet som : s.getSuccesseur()) {
6                 parcoursProfondeurRecursif(som) ;
7             }
8         }
9     }
```

Vous trouverez ci-dessous une méthode de la classe graphe permettant d'obtenir la composante fortement connexe d'un graphe. Elle est basé sur une exploration en largeur qui parcours les listes de successeurs et stocke les résultats dans une LinkedList<Sommet> appelée "descendants".
Une seconde exploration en largeur parcourant les listes de prédécesseurs permet d'obtenir la liste des ancêtres.
Cette méthode termine par l'affichage des sommets qui sont à la fois dans la liste des descandants et dans celle des ancêtres.

```
1     public void composanteFortementConnexe(Sommet s) {
2         LinkedList<Sommet> descandants = new LinkedList<Sommet>() ;
3         LinkedList<Sommet> ancetres = new LinkedList<Sommet>() ;
4         for(Sommet x : this.listeSommet) {
5             x.setMarque(false);
6         }
7         LinkedList<Sommet> liste = new LinkedList<Sommet>() ;
8         liste.addLast(s);
9         s.setMarque(true) ;
10        descandants.add(s) ;
11        while(liste.size() != 0) {
12            Sommet x = liste.removeFirst();
13            for(Sommet som : x.getSuccesseurs()) {
14                if (!som.getMarque()) {
15                    som.setMarque(true);
16                    descandants.add(som) ;
17                    liste.addLast(som);
18                }
19            }
20        }
21        for(Sommet x : this.listeSommet) {
22            x.marquer(false);
23        }
24        liste.clear(); ;
25        liste.addLast(s);
26        s.setMarque(true) ;
27        ancetres.add(s) ;
28        while(liste.size() != 0) {
29            Sommet x = liste.removeFirst();
30            for(Sommet som : x.getPredecesseurs()) {
31                if (!som.getMarque()) {
32                    som.setMarque(true);
33                    ancetres.add(som) ;
34                    liste.addLast(som);
```

```
35                      }
36                  }
37              }
38              for(Sommet x : descandants) {
39                  if(ancetres.contains(x)) {
40                      System.out.print(x.getNom());
41                  }
42              }
43          }
```

TP3 - Correction

Tout d'abord, on peut modifier la classe Sommet en rajoutant les attribut couleur et Dsat qui donneront la couleur et le niveau de saturation du sommet.

```java
import java.util.LinkedList;

public class Sommet {
    private String nom ;
    private LinkedList<Sommet> successeurs ;
    private LinkedList<Sommet> predecesseurs ;

    private boolean marque = false ;
    private int couleur = -1 ;
    private int dsat = 0 ;

    public Sommet(String nom) {
        this.successeurs = new LinkedList<Sommet>() ;
        this.predecesseurs = new LinkedList<Sommet>();
        this.couts = new LinkedList<Integer>() ;
        this.nom = nom ;
    }

    .....
    .....

    public void setCouleur(int i) {
        this.couleur = i ;
    }

    public void setDsat(int i) {
        this.dsat = i ;
    }

    public int getCouleur() {
        return this.couleur ;
    }

    public int getDsat() {
        return this.dsat ;
    }
}
```

Dans un second temps, on mettra en ?uvre l'algorithme de coloration.
Pour cela, il y aura trois étapes :

- Déterminer le sommet à colorer
- Déterminer la couleur à utiliser
- Colorier le sommet et actualiser les niveau de saturation de ses voisins.

Une première méthode privée va permettre de déterminer le sommet à colorier.

```
1      private Sommet sommetAColorier() {
2          int dsatMax = 0;
3          int degMax = 0 ;
4          Sommet s = null ;
5          for (Sommet som : this.listeSommet) {
6              if(som.getCouleur() == -1 &&
7                      (som.getDsat() > dsatMax
8                      || (som.getDsat() == dsatMax && som.getSuccesseurs().size() >
       degMax)
9                      )) {
10                 s = som ;
11                 dsatMax = s.getDsat() ;
12                 degMax = s.getSuccesseurs().size();
13             }
14         }
15         return s ;
16     }
```

La seconde méthode ci-dessous va permettre de déterminer la couleur à utiliser en parcourant l'ensemble des voisins du sommet à colorier, qui sera donnée en variable d'entrée.
Cette méthode renverra la couleur à utiliser.

```
1      private int rechercheCouleur(Sommet s) {
2          int couleur = 1 ;
3          boolean couleurChangee = false ;
4          boolean couleurTrouvee = false ;
5
6          while (!couleurTrouvee) {
7              for(Sommet som : s.getSuccesseurs()) {
8                  if(som.getCouleur() == couleur) {
9                      couleur += 1 ;
10                     couleurChangee = true ;
11                 }
12             }
13             if (couleurChangee == false) {
14                 couleurTrouvee = true ;
15             }
16             couleurChangee = false ;
17         }
18         return couleur ;
19     }
```

La dernière méthode qui sera publique va colorer le graphe et afficher le résultat.

```
1  public void coloration() {
2          int nbSommetAColorier = this.listeSommet.size() ;
3
```

```
4           Sommet s = null ;
5           int couleur ;
6
7           while (nbSommetAColorier != 0) {
8               s = this.sommetAColorier() ;
9
10              couleur = this.rechercheCouleur(s) ;
11
12              s.setCouleur(couleur);
13              nbSommetAColorier -= 1 ;
14
15              //On actualise Dsat
16              for(Sommet som : s.getSuccesseurs()) {
17                  som.setDsat(som.getDsat() + 1) ;
18              }
19          }
20
21          //affichage de la coloration
22          for(Sommet som : this.listeSommet) {
23              System.out.println(som.getNom() + " : " + som.getCouleur());
24          }
25      }
26
```

Pour le test, vous pourrez utiliser la classe suivante :

```
1  public class test {
2
3      public static void main(String[] args) {
4          Graphe gr = new Graphe() ;
5
6          Sommet a = new Sommet("A");
7          Sommet b = new Sommet("B");
8          Sommet c = new Sommet("C");
9          Sommet d = new Sommet("D");
10         Sommet e = new Sommet("E");
11         Sommet f = new Sommet("F");
12         Sommet g = new Sommet("G");
13         Sommet h = new Sommet("H");
14         Sommet i = new Sommet("I");
15         Sommet j = new Sommet("J");
16         Sommet k = new Sommet("K");
17         Sommet l = new Sommet("L");
18
19         a.ajouterListeSuccesseurs(new Sommet[] {b , e , i, j});
20         b.ajouterListeSuccesseurs(new Sommet[] {a , c , d, e});
21         c.ajouterListeSuccesseurs(new Sommet[] {d , f , b});
22         d.ajouterListeSuccesseurs(new Sommet[] {c , l , b});
23         e.ajouterListeSuccesseurs(new Sommet[] {b , a});
24         f.ajouterListeSuccesseurs(new Sommet[] {h , c , g });
25         g.ajouterListeSuccesseurs(new Sommet[] {f , h });
26         h.ajouterListeSuccesseurs(new Sommet[] {g , f });
27         j.ajouterListeSuccesseurs(new Sommet[] {k,a});
28         k.ajouterSuccesseur(j);
29         i.ajouterSuccesseur(a);
30         l.ajouterSuccesseur(d);
31
32         gr.ajouterListeSommets(new Sommet[] {a , b , c , d , e , f , g , h , i , j ,
```

```
    k , l});
33
34          gr.coloration();
35      }
36  }
```

TP4 - Correction

```
1     private boolean reconnaissanceSimplicial(Sommet s, LinkedList<Sommet> liste) {
2         boolean sommetSimplicial = true ;
3         for(Sommet som1 : s.getSuccesseurs()) {
4             for(Sommet som2 : s.getSuccesseurs()) {
5                 if(som1 != som2 &&
6                         (liste.contains(som1) && liste.contains(som2)) &&
7                         (som1.getSuccesseurs().contains(som2) == false ||
8                                 som2.getSuccesseurs().contains(som1) == false
      )) {
9                     sommetSimplicial = false ;
10                }
11            }
12        }
13        return sommetSimplicial ;
14    }
15
16    public boolean grapheCordal() {
17        LinkedList<Sommet> liste = new LinkedList<Sommet>() ;
18        boolean sommetSimplicialTrouve = true ;
19
20        for(Sommet s : this.listeSommet) {
21            liste.add(s) ;
22            System.out.print(s.getNom()) ;
23        }
24
25        while (sommetSimplicialTrouve == true && liste.size() != 0) {
26            System.out.println() ;
27            sommetSimplicialTrouve = false ;
28            for(Sommet s : liste) {
29                if (this.reconnaissanceSimplicial( s , liste) == true) {
30                    liste.remove(s) ;
31                    sommetSimplicialTrouve = true ;
32                    for(Sommet som : liste) {
33                        System.out.print(som.getNom()) ;
34                    }
35                    break ;
36                }
37            }
38        }
39        return sommetSimplicialTrouve ;
40    }
```

Pour le test, vous pourrez utiliser la classe suivante :

```
1  public class test {
2
```

```
3       public static void main(String[] args) {
4           Graphe gr = new Graphe() ;
5
6           Sommet a = new Sommet("A");
7           Sommet b = new Sommet("B");
8           Sommet c = new Sommet("C");
9           Sommet d = new Sommet("D");
10          Sommet e = new Sommet("E");
11          Sommet f = new Sommet("F");
12          Sommet g = new Sommet("G");
13          Sommet h = new Sommet("H");
14          Sommet i = new Sommet("I");
15          Sommet j = new Sommet("J");
16          Sommet k = new Sommet("K");
17          Sommet l = new Sommet("L");
18
19          a.ajouterListeSuccesseurs(new Sommet[] {b , e , i, j});
20          b.ajouterListeSuccesseurs(new Sommet[] {a , c , d, e});
21          c.ajouterListeSuccesseurs(new Sommet[] {d , f , b});
22          d.ajouterListeSuccesseurs(new Sommet[] {c , l , b});
23          e.ajouterListeSuccesseurs(new Sommet[] {b , a});
24          f.ajouterListeSuccesseurs(new Sommet[] {h , c , g });
25          g.ajouterListeSuccesseurs(new Sommet[] {f , h });
26          h.ajouterListeSuccesseurs(new Sommet[] {g , f });
27          j.ajouterListeSuccesseurs(new Sommet[] {k,a});
28          k.ajouterSuccesseur(j, 0);
29          i.ajouterSuccesseur(a, 0);
30          l.ajouterSuccesseur(d, 0);
31
32          e.ajouterSuccesseur(l, 0);
33          l.ajouterSuccesseur(e, 0);
34
35          gr.ajouterListeSommets(new Sommet[] {a , b , c , d , e , f , g , h , i , j ,
        k , l});
36
37          System.out.println(gr.grapheCordal()) ;
38      }
39  }
```

TP5 - Correction

```java
public class Sommet {
    private String nom ;
    private LinkedList<Sommet> successeurs ;
    private LinkedList<Float> coutSuccesseurs ;
    private LinkedList<Sommet> predecesseurs ;
    private Sommet predecesseurPCC ;
    private float coutPCC ;
    private boolean marque = false ;
    private int couleur = -1 ;
    private int dsat = 0 ;

    public Sommet(String nom) {
        this.successeurs = new LinkedList<Sommet>() ;
        this.predecesseurs = new LinkedList<Sommet>();
        this.coutSuccesseurs = new LinkedList<Float>() ;
        this.nom = nom ;
        this.predecesseurPCC = null ;
    }

    public void ajouterSuccesseur(Sommet s, float c) {
        this.successeurs.add(s) ;
        s.predecesseurs.add(this) ;
        this.coutSuccesseurs.add(c) ;
    }

    public void ajouterListeSuccesseurs(Sommet[] listeSommet, float[] listeCout) {
        if(listeSommet.length == listeCout.length) {
            for(int i = 0 ; i < listeSommet.length ; i++) {
                this.ajouterSuccesseur(listeSommet[i] , listeCout[i]) ;
            }
        }
        else {
            System.out.println("Erreur de dimension entre la liste des sommets et la liste des couts");
        }
    }

    public void setCoutPCC(float c) {
        this.coutPCC = c ;
    }

    public float getCout(Sommet s) {
        for(int i= 0 ; i < this.successeurs.size(); i++) {
            if(s==this.successeurs.get(i)) {
                return this.coutSuccesseurs.get(i) ;
            }
        }
```

```
47          return 0 ;
48      }
49
50      public float getCoutPCC() {
51          return this.coutPCC ;
52      }
53
54      ...
55
56      public void afficher() {
57          System.out.println(getNom() + (" : "));
58          System.out.print(" Successeurs : (") ;
59          for(int i = 0 ; i < this.successeurs.size(); i++) {
60              System.out.print(this.successeurs.get(i).getNom() + " , " + this.
    coutSuccesseurs.get(i) + " ; ");
61
62          }
63          System.out.println(")");
64          System.out.print(" Prédécesseurs : ") ;
65          for(Sommet s : this.predecesseurs) {
66              System.out.print(s.getNom() + " ");
67          }
68      }
69  }
```

Vous trouverez ci-dessous une méthode permettant de déterminer et d'afficher le plus court chemin entre deux sommets s et t, en utilisant l'algorithme de Dijkstra.

```
1   public class Graphe {
2       ...
3
4           public void dijkstra(Sommet s, Sommet t){
5           for(Sommet som : this.listeSommet) {
6               som.setPredecesseurPCC(null);
7               som.setCoutPCC(Float.MAX_VALUE);
8           }
9           LinkedList<Sommet> liste = new LinkedList<Sommet>() ;
10          LinkedList<Sommet> listeSommetDistMin = new LinkedList<Sommet>() ;
11          Sommet sommetDistMin = s;
12
13          liste.add(s) ;
14          s.setPredecesseurPCC(s);
15          s.setCoutPCC(0);
16          while(liste.size() != 0) {
17              sommetDistMin = distMin(liste) ;
18              listeSommetDistMin.add(sommetDistMin) ;
19              for(Sommet sommet : sommetDistMin.getSuccesseurs()) {
20                  relacher(sommetDistMin , sommet, liste) ;
21              }
22              liste.remove(sommetDistMin) ;
23          }
24
25          //Affichage
26          Sommet som = t ;
27          if(listeSommetDistMin.contains(t)) {
28              while(som.getPredecesseurPCC() != som) {
29                  liste.add(som) ;
```

```
30                  som = som.getPredecesseurPCC() ;
31              }
32              liste.add(s) ;
33              while(liste.size() != 0) {
34                  System.out.print(liste.removeLast().getNom());
35              }
36              System.out.print(" : " + t.getCoutPCC());
37          }else {
38              System.out.print("Il n'y a aucun chemin pour aller de " + s.getNom() + "
    à " + t.getNom() );
39          }
40
41      }
42
43      private Sommet distMin(LinkedList<Sommet> liste) {
44          float distMini ;
45          Sommet sommetDistMin = null;
46          distMini = liste.getFirst().getCoutPCC() ;
47          for(Sommet som : liste) {
48              if(som.getCoutPCC() <= distMini) {
49                  distMini = som.getCoutPCC() ;
50                  sommetDistMin = som ;
51              }
52          }
53          return sommetDistMin ;
54      }
55
56      private void relacher(Sommet x , Sommet y , LinkedList<Sommet> liste) {
57          if(y.getCoutPCC() > x.getCoutPCC() + x.getCout(y)) {
58              y.setPredecesseurPCC(x);
59              y.setCoutPCC(x.getCoutPCC() + x.getCout(y)) ;
60              if(liste.contains(y) == false) {
61                  liste.add(y) ;
62              }
63          }
64      }
```

Pour le test, vous pourrez utiliser la classe suivante :

```
1  public class test {
2
3      public static void main(String[] args) {
4          Graphe gr = new Graphe() ;
5
6          Sommet a = new Sommet("A");
7          Sommet b = new Sommet("B");
8          Sommet c = new Sommet("C");
9          Sommet d = new Sommet("D");
10         Sommet e = new Sommet("E");
11         Sommet f = new Sommet("F");
12         Sommet g = new Sommet("G");
13         Sommet h = new Sommet("H");
14         Sommet i = new Sommet("I");
15         Sommet j = new Sommet("J");
16         Sommet k = new Sommet("K");
17         Sommet l = new Sommet("L");
18
19         a.ajouterListeSuccesseurs(new Sommet[] {b , e, i, j}, new float[] {5,3,2,1});
```

```
20          b.ajouterListeSuccesseurs(new Sommet[] {a , c , d} , new float[] {4,6,2});
21          c.ajouterListeSuccesseurs(new Sommet[] {d , f} , new float[] {3 , 4});
22          d.ajouterListeSuccesseurs(new Sommet[] {c , l} , new float[] {2 , 2});
23          e.ajouterSuccesseur(b , 1);
24          f.ajouterSuccesseur(h , 3);
25          g.ajouterSuccesseur(f , 3);
26          h.ajouterSuccesseur(g , 7);
27          j.ajouterSuccesseur(k, 4);
28
29          gr.ajouterListeSommets(new Sommet[] {a , b , c , d , e , f , g , h , i , j ,
        k , l});
30
31          gr.dijkstra(a, f);
32      }
33  }
```