Tp - Parcours des graphes

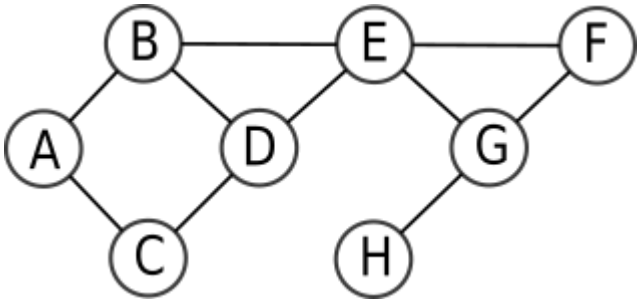
L'objectif de ce travail est de compléter la classe *Gliste* qui implémente un graphe non orienté et non pondéré, en créant des méthodes qui permettent :

- de réaliser un parcours en largeur avec une méthode itérative utilisant une File,
- de déterminer le chemin le plus court entre 2 nœuds,
- de réaliser un parcours en largeur avec une méthode itérative utilisant une Pile,
- de réaliser un parcours en largeur avec une méthode récursive

⇨ Télécharger le fichier *parcoursGraphe.zip* proposé sur *nsibranly.fr*. Décompresser et copier les fichiers qu'il contient :

- *grapheListe.py* qui contient la classe *Gliste* permettant d'implémenter graphe non orienté et non pondéré,
- *pileFile.py* qui contient les classes *Pile* et *File* qui proposent l'implémentation d'une Pile et d'une File en P.O.O.

⇨ Exécuter le programme principal de *grapheListe.py* . Il permet de modéliser le graphe donné ci-contre :



1- PARCOURS EN LARGEUR :

La méthode *parcoursFile()* permet de réaliser un parcours itératif du graphe, en stockant les voisins des nœuds visités dans une File. Le script incomplet de cette méthode est donné ci-contre. Elle a comme paramètre un nœud de départ et elle renvoie la liste *listeVisite* contenant le parcours.

```
def parcoursFile(self,noeudDepart):
    listeVisite = []
    f = File()
```

⇨ Compléter le script de cette méthode.

⇨ Vérifier avec les exemples tests suivants :

```
>>> g.parcoursFile('A')
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

```
>>> g.parcoursFile('E')
['E', 'B', 'D', 'F', 'G', 'A', 'C', 'H']
```

```
def parcoursFile(self,noeudDepart):
    listeVisite = []
    f = File()
    f.enfiler(noeudDepart)
    while not f.estVide() :
        nd = f.defiler()
        if nd not in listeVisite :
            listeVisite.append(nd)
            for voisin in self.dic[nd] :
                if voisin not in listeVisite :
                    f.enfiler(voisin)
    return listeVisite
```

2- RECHERCHE DU CHEMIN LE PLUS COURT :

La méthode *rechercheChemin()* a comme paramètre un nœud de départ et un nœud d'arrivée. Elle renvoie la liste *chemin* contenant les nœuds à parcourir pour aller du nœud de départ au nœud d'arrivée avec un minimum de saut.

On en donne ci-contre le script incomplet.

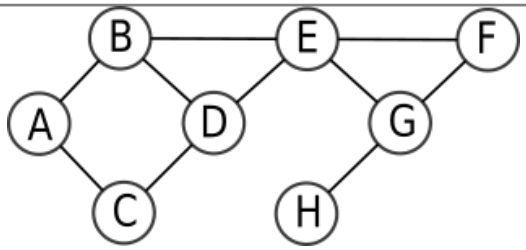
⇨ Compléter le script de cette méthode.

```
def rechercheChemin(self,noeudDepart, noeudArrive):
    listeVisite = []
    chemin = [noeudArrive]
    parents = {}
    f = File()
    Partie à compléter …. Reprendre le script de parcoursFile() dans lequel il faut insérer le remplissage du dictionnaire parents …..
    nd = noeudArrive
    while nd != noeudDepart :
        nd = parents[nd]
        chemin = [nd] + chemin
    return chemin
```

⇨ Vérifier avec les exemples tests suivants :

```
>>> g.rechercheChemin('F','A')
['F', 'E', 'B', 'A']
```

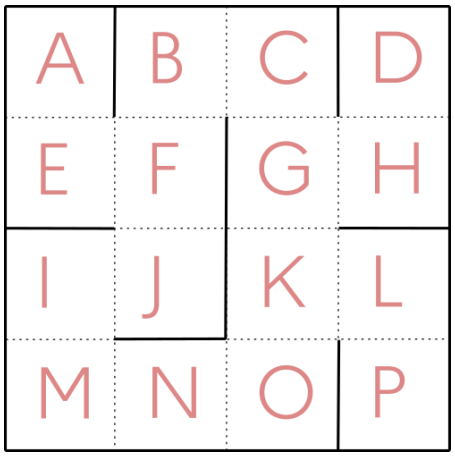
```
>>> g.rechercheChemin('B','H')
['B', 'E', 'G', 'H']
```

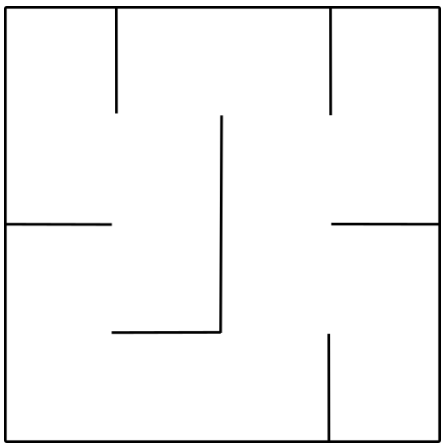



```
def rechercheChemin(self,noeudDepart, noeudArrive):
    listeVisite = []
    chemin = [noeudArrive]
    parents = {}
    f = File()
    f.enfiler(noeudDepart)
    trouve = False
    while f.estVide() == False and trouve == False:
        nd = f.defiler()
        if nd not in listeVisite :
            listeVisite.append(nd)
            for voisin in self.dic[nd] :
                if voisin not in listeVisite :
                    f.enfiler(voisin)
                    if voisin not in parents :
                        parents[voisin] = nd
                if voisin == noeudArrive : trouve = True
    if trouve == False : return None
    nd = noeudArrive
    while nd != noeudDepart :
        nd = parents[nd]
        chemin = [nd] + chemin
    return chemin
```

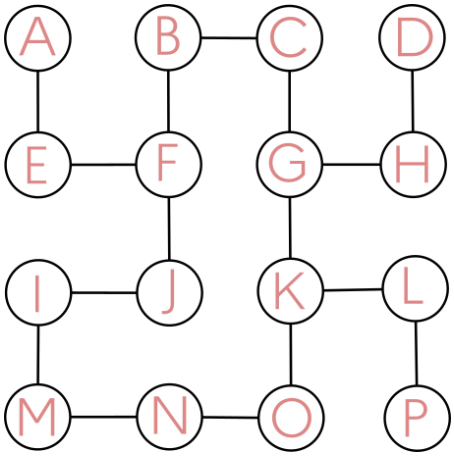
3- APPLICATION AU CAS D'UN LABYRINTHE :

On se propose d'appliquer la méthode *rechercheChemin()* au cas du labyrinthe ci-contre. Pour y voir plus clair, on affecte ci-dessous, une lettre à chaque case.





Cette affectation nous permet de pouvoir modéliser ce labyrinthe par le graphe donné ci-contre :



⇨ Créer une instance de la classe *Gliste* qui implémente ce labyrinthe. Cette instance sera nommée labyrinthe.

```
labyrinthe = Gliste()
labyrinthe.ajout_arete('A', 'E')
```

⇨ Donner en commentaire dans le fichier *grapheParcours.py* , le plus court chemin :

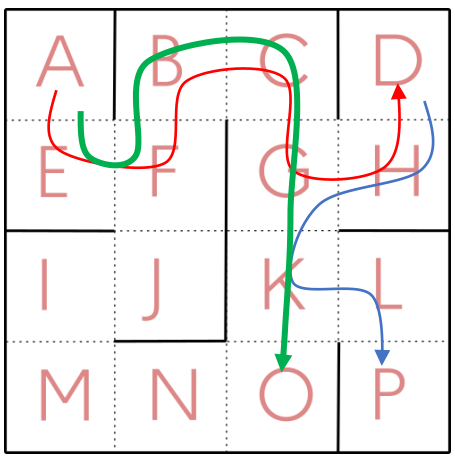
- pour aller de A à D
- pour aller de D à P
- pour aller de A à O

```
# Main
labyrinthe = Gliste()
labyrinthe.ajout_arete('A', 'E')
labyrinthe.ajout_arete('E', 'F')
labyrinthe.ajout_arete('F', 'B')
labyrinthe.ajout_arete('B', 'C')
labyrinthe.ajout_arete('C', 'G')
labyrinthe.ajout_arete('G', 'H')
labyrinthe.ajout_arete('H', 'D')
labyrinthe.ajout_arete('G', 'K')
labyrinthe.ajout_arete('F', 'J')
labyrinthe.ajout_arete('J', 'I')
labyrinthe.ajout_arete('I', 'M')
labyrinthe.ajout_arete('M', 'N')
labyrinthe.ajout_arete('N', 'O')
labyrinthe.ajout_arete('O', 'K')
labyrinthe.ajout_arete('K', 'L')
labyrinthe.ajout_arete('L', 'P')
```

```
>>> labyrinthe.rechercheChemin('A','D')
['A', 'E', 'F', 'B', 'C', 'G', 'H', 'D']

>>> labyrinthe.rechercheChemin('D','P')
['D', 'H', 'G', 'K', 'L', 'P']

>>> labyrinthe.rechercheChemin('A','O')
['A', 'E', 'F', 'B', 'C', 'G', 'K', 'O']
```



4- PARCOURS EN PROFONDEUR :

La méthode *parcoursPile()* permet de réaliser un parcours itératif du graphe, en stockant les voisins des nœuds visités dans une Pile. Elle a comme paramètre un nœud de départ et elle renvoie la liste *listeVisite* contenant le parcours.

⇨ Ecrire le script de cette méthode.

⇨ Vérifier avec les exemples tests suivants :

```
>>> g.parcoursPile('A')
['A', 'C', 'D', 'E', 'G', 'H', 'F', 'B']
```

```
>>> g.parcoursPile('E')
['E', 'G', 'H', 'F', 'D', 'C', 'A', 'B']
```

```
def parcoursPile(self,noeudDepart):
    listeVisite = []
    p = Pile()
    p.empiler(noeudDepart)
    while not p.estVide() :
        nd = p.depiler()
        if nd not in listeVisite :
            listeVisite.append(nd)
            for voisin in self.dic[nd] :
                if voisin not in listeVisite :
                    p.empiler(voisin)
    return listeVisite
```

5- PARCOURS RECURSIF :

La méthode *parcoursRecursif()* permet de réaliser un parcours récursif du graphe. Elle a comme paramètre un nœud de départ et une liste *listeVisite* contenant le parcours qui se construit au fil des appels récursifs.

⇨ Ecrire le script de cette méthode.

⇨ Vérifier avec les exemples tests suivants :

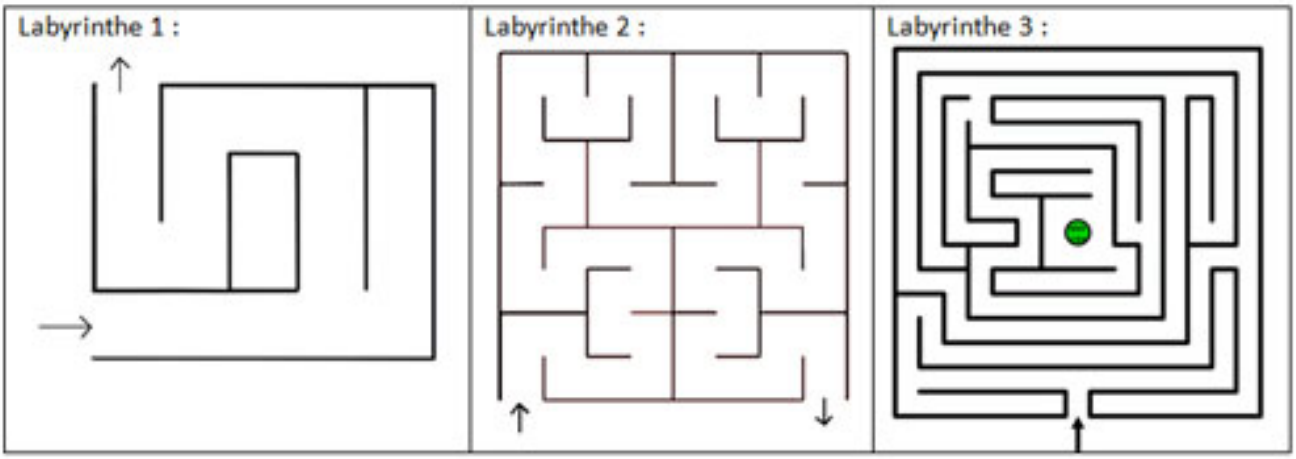
```
>>> g.parcoursRecursif('A',[])
['A', 'B', 'D', 'C', 'E', 'F', 'G', 'H']
```

```
>>> g.parcoursRecursif('E',[])
['E', 'B', 'A', 'C', 'D', 'F', 'G', 'H']
```

```
def parcoursRecursif(self,nd,listeVisite):
    listeVisite.append(nd)
    for voisin in self.dic[nd] :
        if voisin not in listeVisite :
            self.parcoursRecursif(voisin,listeVisite)
    return listeVisite
```

6- APPLICATION AU CAS D'UN LABYRINTHE :

On se propose d'appliquer la méthode *rechercheChemin()* au cas des labyrinthes donnés ci-dessous.



⇨ Créer une instance de la classe *Gliste* qui implémente **l'un** de ces 3 labyrinthes. Cette instance sera nommée *labyrinthe1* ou *labyrinthe2* ou *labyrinthe3*.

⇨ Déterminer le chemin le plus court entre entrée et sortie. Regroupées dans un fichier nommé *monLabyrinthe.docx* toutes les étapes (croquis, copies d'écran) qui ont permis de résoudre ce problème.

⇨ Uploader sur nsibranly.fr les fichiers *grapheParcours.py* et *monLabyrinthe.docx.*