Projet 2 POO2 LI3 option Genie Logiciel 2020/2021

CM - TP : M.Diouf

Objectifs ¶

- Decouvrir les librairies pandas, networkx, folium
- Chargement de données csv via des fichiers (Le projet est accompagné de deux fichiers)
- Comprendre la manipulation des structures de données en Python via la POO
- Comprendre les ADT et implementer une File, une Pile
- Comprendre les methodes de parcours de graphe
- Le projet fait quatre parties
- NB : Lisez attentivement le projet, les consignes se trouvent tout a fait à la fin de la notebook

Partie 1

- Decouvrir les librairies pandas, networkx, folium

In [36]:
```
#Chargement des librairies Pandas, Network et folium
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import csv
import folium

%matplotlib inline
'''
En cas de probléme de chargement des librairies utilisez la commande
pip install <nom_librairie>
Conseil : Consultez le site officiel des librairies ci dessus
'''
```

In [37]:
```
# Utilisez la bibliothéque pandas pour lire le fichier transport-nodes.csv
```

Out[37]:

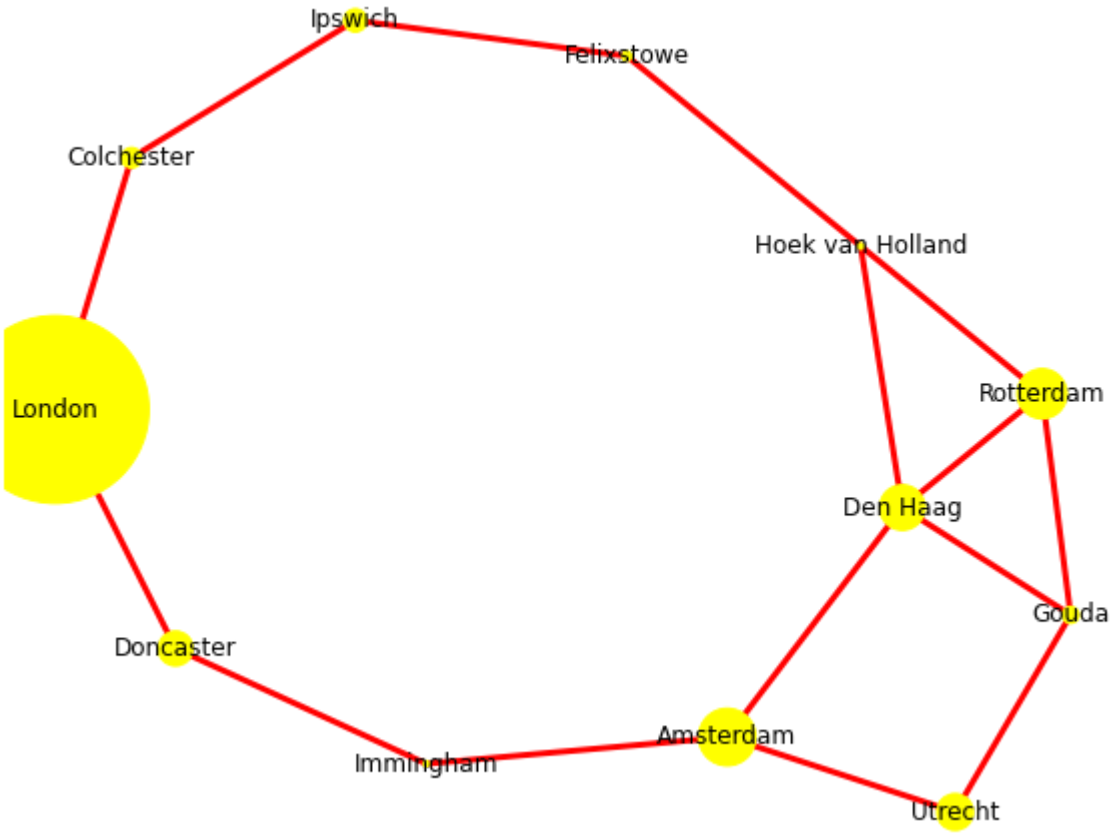
	id	latitude	longitude	population
0	Amsterdam	52.379189	4.899431	821752
1	Utrecht	52.092876	5.104480	334176
2	Den Haag	52.078663	4.288788	514861
3	Immingham	53.612390	-0.222190	9642
4	Doncaster	53.522850	-1.131160	302400
5	Hoek van Holland	51.977500	4.133330	9382
6	Felixstowe	51.963750	1.351100	23689
7	Ipswich	52.059170	1.155450	133384
8	Colchester	51.889210	0.904210	104390
9	London	51.509865	-0.118092	8787892
10	Rotterdam	51.922500	4.479170	623652
11	Gouda	52.016670	4.708330	70939

In [38]:
```
# Utilisez la bibliothéque pandas pour lire le fichier transport-relationship
s.csv
```

Out[38]:

	src	dst	relationship	cost
0	Amsterdam	Utrecht	EROAD	46
1	Amsterdam	Den Haag	EROAD	59
2	Den Haag	Rotterdam	EROAD	26
3	Amsterdam	Immingham	EROAD	369
4	Immingham	Doncaster	EROAD	74
5	Doncaster	London	EROAD	277
6	Hoek van Holland	Den Haag	EROAD	27
7	Felixstowe	Hoek van Holland	EROAD	207
8	Ipswich	Felixstowe	EROAD	22
9	Colchester	Ipswich	EROAD	32
10	London	Colchester	EROAD	106
11	Gouda	Rotterdam	EROAD	25
12	Gouda	Utrecht	EROAD	35
13	Den Haag	Gouda	EROAD	32
14	Hoek van Holland	Rotterdam	EROAD	33

In [39]:
```
# Construisez le graphe et le visualiser avec la fonction from_pandas_datafram
e de networkx
```



Out[39]:
```
{'Amsterdam': {},
 'Utrecht': {},
 'Den Haag': {},
 'Rotterdam': {},
 'Immingham': {},
 'Doncaster': {},
 'London': {},
 'Hoek van Holland': {},
 'Felixstowe': {},
 'Ipswich': {},
 'Colchester': {},
 'Gouda': {}}
```

In [40]:
```
#Ajouter des attributs longitude et latitude aux noeuds avec Networkx en utili
sant:
# - le dictionnaire "node" de networkx qui contient les noeuds
# - le dataframe transportnode defini plus haut

'''
Ajouter des attributs
Inputs:
    Un graphe,
    un dataframe contenant les données,
    nom de l'attribut,
    le nom de la colonne index du dataframe
Pas d'output:
Utilisez la fonction set_node_attributes de networkx

'''
def ajouterAttribut(myGraphe,dfnode, nameAttr, Index):
    pass
```

In [41]:
```
ajouterAttribut(g,transport_nodes,'latitude','id')
ajouterAttribut(g,transport_nodes,'longitude','id')
ajouterAttribut(g,transport_nodes,'population','id')
```

In [42]:
```
dict(g.nodes.data())
```

Out[42]:
```
{'Amsterdam': {'latitude': 52.379189000000004,
  'longitude': 4.899431,
  'population': 821752},
 'Utrecht': {'latitude': 52.092876000000004,
  'longitude': 5.104480000000001,
  'population': 334176},
 'Den Haag': {'latitude': 52.078663,
  'longitude': 4.288787999999999,
  'population': 514861},
 'Rotterdam': {'latitude': 51.9225,
  'longitude': 4.47917,
  'population': 623652},
 'Immingham': {'latitude': 53.612390000000005,
  'longitude': -0.22219,
  'population': 9642},
 'Doncaster': {'latitude': 53.52285,
  'longitude': -1.13116,
  'population': 302400},
 'London': {'latitude': 51.509865000000005,
  'longitude': -0.118092,
  'population': 8787892},
 'Hoek van Holland': {'latitude': 51.9775,
  'longitude': 4.13333,
  'population': 9382},
 'Felixstowe': {'latitude': 51.96375,
  'longitude': 1.3511,
  'population': 23689},
 'Ipswich': {'latitude': 52.05917, 'longitude': 1.15545, 'population': 13338
4},
 'Colchester': {'latitude': 51.88921,
  'longitude': 0.9042100000000001,
  'population': 104390},
 'Gouda': {'latitude': 52.01667, 'longitude': 4.70833, 'population': 70939}}
```

In [45]:
```
#Représentation des noeuds sur une carte avec Folium
#La librairie de visualisation Folium a été chargée en haut
#On met en place d'abord le fond de la carte

for i in g.nodes:
    pass
```

In [47]:
```
basemap
```

Out[47]:

In [48]:
```
#Ajouter des marqueurs pour tous les noeuds du reseau avec folium.Marker
def marker(g):
    for i in g.nodes:
        pass
    return basemap
```

In [49]:
```
marker(g)
```

Out[49]:

In [50]:
```
'''
Cette fonction permet de construire une liste de liste comprenant pour chaque
sous-liste ses coordonnées et celles d'un voisin
Ces points peuvent etre utilisés pour représenter les lignes dans la carte
Input: le graphe
Output: Une liste de couples representant les longitudes et latitudes d'un poi
nt et d'un de ses voisins
'''
def construirePointsImage(myGraphe):
    points = []
    for i in myGraphe.nodes:
        for neighbor in myGraphe.neighbors(i):
            pass
    return points
```

In [51]:
```
coordonneesvoisins = construirePointsImage(g)
print(coordonneesvoisins)
```

```
[[[52.379189000000004, 4.899431], [52.092876000000004, 5.104480000000001]],
[[52.379189000000004, 4.899431], [52.078663, 4.288787999999999]], [[52.379189
000000004, 4.899431], [53.612390000000005, -0.22219]], [[52.092876000000004,
5.104480000000001], [52.379189000000004, 4.899431]], [[52.092876000000004, 5.
104480000000001], [52.01667, 4.70833]], [[52.078663, 4.288787999999999], [52.
379189000000004, 4.899431]], [[52.078663, 4.288787999999999], [51.9225, 4.479
17]], [[52.078663, 4.288787999999999], [51.9775, 4.13333]], [[52.078663, 4.28
8787999999999], [52.01667, 4.70833]], [[51.9225, 4.47917], [52.078663, 4.2887
87999999999]], [[51.9225, 4.47917], [52.01667, 4.70833]], [[51.9225, 4.4791
7], [51.9775, 4.13333]], [[53.612390000000005, -0.22219], [52.37918900000000
4, 4.899431]], [[53.612390000000005, -0.22219], [53.52285, -1.13116]], [[53.5
2285, -1.13116], [53.612390000000005, -0.22219]], [[53.52285, -1.13116], [51.
509865000000005, -0.118092]], [[51.509865000000005, -0.118092], [53.52285, -
1.13116]], [[51.509865000000005, -0.118092], [51.88921, 0.9042100000000001]],
[[51.9775, 4.13333], [52.078663, 4.288787999999999]], [[51.9775, 4.13333], [5
1.96375, 1.3511]], [[51.9775, 4.13333], [51.9225, 4.47917]], [[51.96375, 1.35
11], [51.9775, 4.13333]], [[51.96375, 1.3511], [52.05917, 1.15545]], [[52.059
17, 1.15545], [51.96375, 1.3511]], [[52.05917, 1.15545], [51.88921, 0.9042100
000000001]], [[51.88921, 0.9042100000000001], [52.05917, 1.15545]], [[51.8892
1, 0.9042100000000001], [51.509865000000005, -0.118092]], [[52.01667, 4.7083
3], [51.9225, 4.47917]], [[52.01667, 4.70833], [52.092876000000004, 5.1044800
00000001]], [[52.01667, 4.70833], [52.078663, 4.288787999999999]]]
```

In [52]:
```
#Visualiser une carte du graphe avec ses noeuds et les arcs sous forme lignes
'''
A faire
Permet de visualiser une carte du graphe avec ses noeuds et les arcs sous form
e lignes
Prend en entrée:
- un graphe
- les coordonnées entre chaque point et ses voisins calculées avec la fonction
ci-dessus
- Une location par defaut
- Un paramétrage de folium
Output: la carte
'''

def visualiserFolium(myGraphe, points, locationpardefaut = [52.3791890, 4.8994
31],tiles='Stamen Toner', explored = None ):
    pass
    return basemap
```

In [53]:
```
visualiserFolium(g,coordonneesvoisins)
```

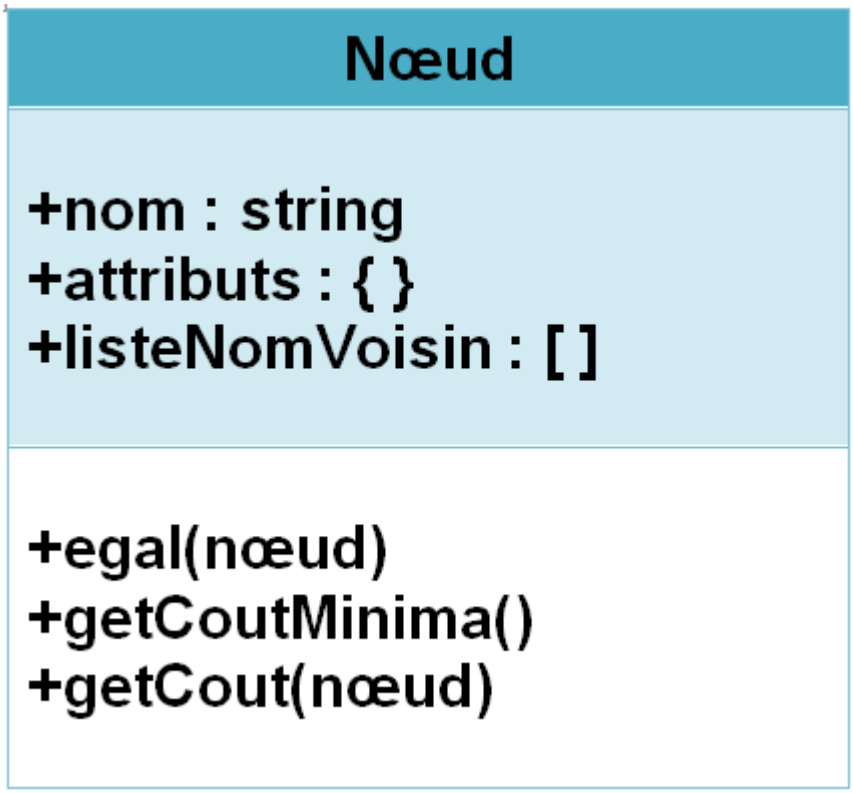
Out[53]:

Partie 2

- Dans cette partie vous allez implémenter des ADT: Noeud, Graphe, File, Pile

Implementer la classe Noeud

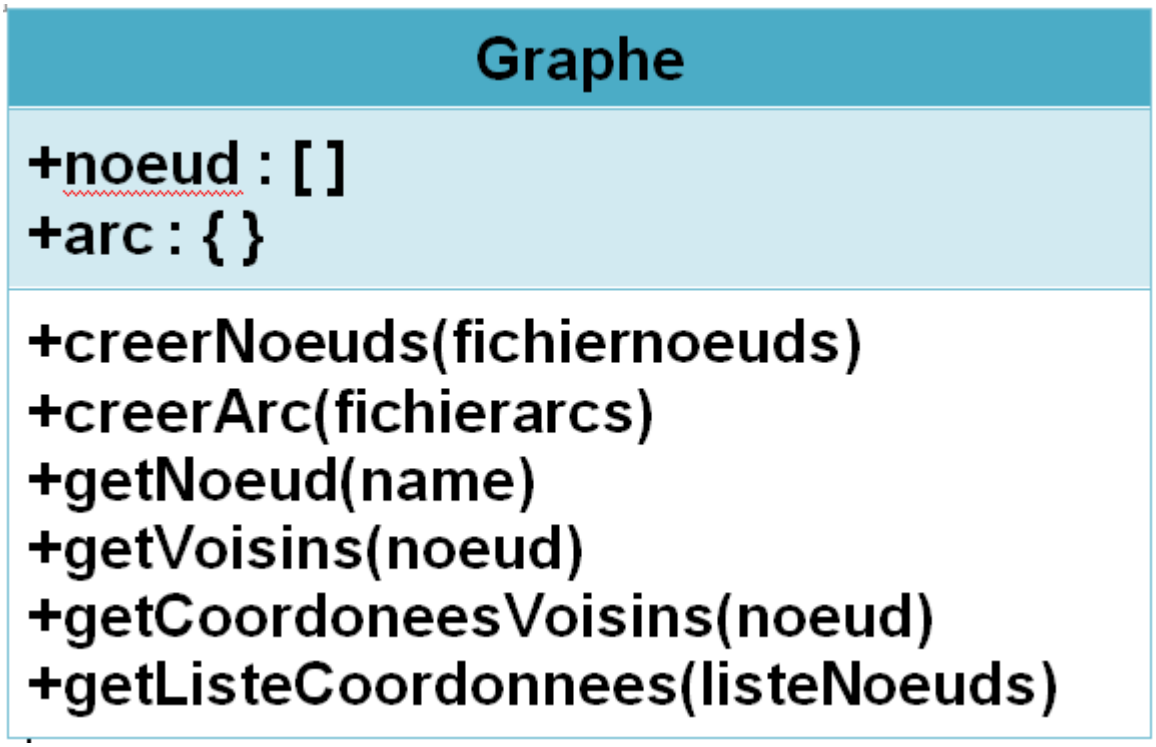
- Un noeud a un nom
- Un noeud a des attributs sous forme de dictionnaire pyton avec comme cle: une liste de coordonnes (latitude, longitude), la taille de la population
- La liste des voisins sera initialisée à la création des arcs dans le graphe
- Vous ajouterez toutes les methodes nécessaires



In [54]:
```
class Noeud:
    def __init__(self,name):
        self.name = name
        self.attributs = {}
        self.listeNomVoisin = []

    def setAttribut(self,key,values):
        pass

    def getAttribut(self,key):
        pass

    def getName(self):
        pass
    '''
    Deux noeuds sont egaux s'ils ont même nom
    '''
    def egal(self, noeud):
        pass

    def getCoutMin(self):
        pass

    def getCout(self,noeud):
        pass
```

Implementer la classe Graphe

- Les noeuds des graphes doivent être initialisé à l'aide du fichier transport-node.csv (vous n'utiliserez plus pandas). Vous initiliserez aussi les attributs
- Les arcs des graphes doivent être initialisé à l'aide du fichier transport-relations.csv (vous n'utiliserez plus pandas). Vous initiliserez aussi les attributs
- On doit pouvoir retrouver les coordonnées d'un noeud grâce à son nom
- On doit avoir la liste des noeuds voisins d'un noeud donné. Le graphe est non orienté donc le voisinage est reciproque


Graphe
+noeud : [ ]
+arc : { }
+creerNoeuds(fichiernoeuds)
+creerArc(fichierarcs)
+getNoeud(name)
+getVoisins(noeud)
+getCoordoneesVoisins(noeud)
+getListeCoordonnees(listeNoeuds)

In [58]:
```
class Graphe:
    '''
    Les noeuds seront mis dans une liste
    Les arcs forment un dictionnaire avec comme clé les noms des noeuds et com
me valeurs une liste de noeud
    '''
    def __init__(self):
        self.noeuds = []
        self.arcs = {}
    '''
    Creer les noeuds avec un fichier csv
    On peut mettre tous les noeuds dans une liste
    On doit attribuer à chaque noeud ses attributs: latitude, longitude, popul
ation
    On initialise le dictionnaire des arcs en creant la cle avec le nom du noe
ud et la valeur avec une liste vide
    N'oubliez pas de gerer les exceptions
    '''
    def creerNoeuds(self,fichiernoeuds):
        pass


    '''
    - Creer les arcs avec un  fichier csv
    - Utilisez un dictionnaire pour les arcs
    - N'oubliez pas que le graphe est non oriente.
    - Pour chaque noeud on mettra des tuples dans la liste de ses voisins: (no
m du voisin, cout du chemin)
    - Gerer les exceptions
    '''
    def creerArc(self,fichierarcs):
        pass
    '''
    Retrouver un noeud à partir de son nom
    '''
    def getNoeud(self,name):
        pass
    '''
    Trouver les noeuds voisins d'un noeud donne
    '''
    def getVoisins(self,noeud):
        pass


    '''
    Recuperer pour un noeud donne les latitudes et longitudes de ses voisins
    Constituer des pairs de listes de coordonnées entre le point et ses voisin
s
    pour une representation sous folium

    '''
    def getCoordonnesVoisins(self, noeud):
        pass

    '''
    Recuperer les coordonnees d'une liste de noeuds pour visualiser sous foliu
```

```
m
    Prend en entrée une liste de nom de noeud
    Retourne une liste de sous-listes à deux elements de coordonnées
    '''
    def getListeCoordonnees(self,listeNoeuds):
        pass

    '''
    Visualiser les noeuds et les arcs sous folium
    Entree: le parametre explored sera utilisé pour les parcours de graphe
    '''


    def visualiserFolium(self, locationpardefaut = [52.3791890, 4.899431],tile
s='Stamen Toner',explored = None ):
        pass
```

In [59]:
```
G = Graphe()
G.creerNoeuds('transport-nodes.csv')
```

Out[59]:
```
['Amsterdam',
 'Utrecht',
 'Den Haag',
 'Immingham',
 'Doncaster',
 'Hoek van Holland',
 'Felixstowe',
 'Ipswich',
 'Colchester',
 'London',
 'Rotterdam',
 'Gouda']
```

In [24]:
```
G = Graphe()
G.creerArc('transport-relationships.csv')
```

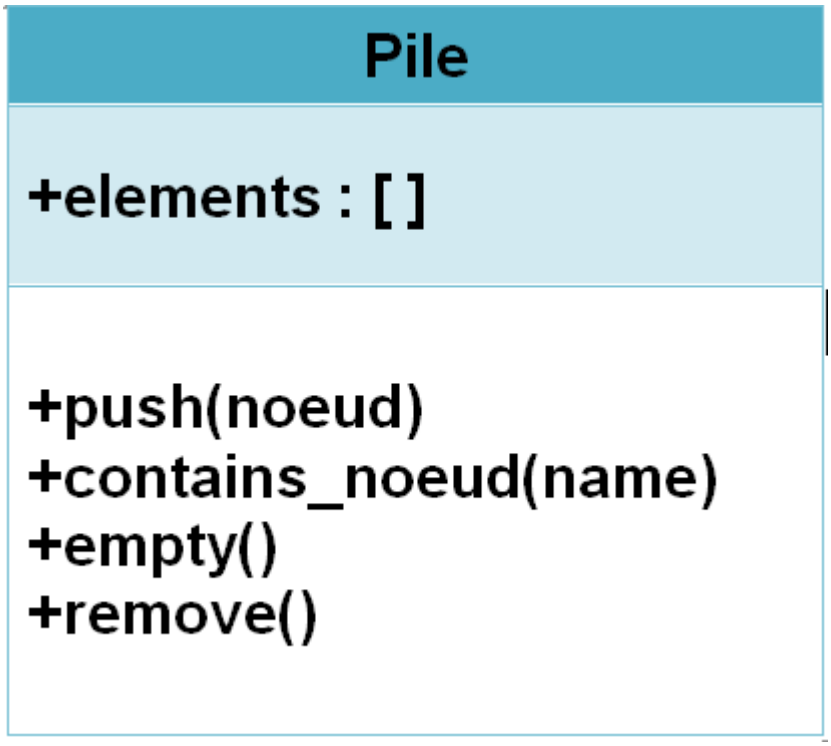
Out[24]:
```
[{'src': 'Amsterdam', 'dst': 'Utrecht', 'relationship': 'EROAD', 'cost': '4
6'},
 {'src': 'Amsterdam',
  'dst': 'Den Haag',
  'relationship': 'EROAD',
  'cost': '59'},
 {'src': 'Den Haag',
  'dst': 'Rotterdam',
  'relationship': 'EROAD',
  'cost': '26'},
 {'src': 'Amsterdam',
  'dst': 'Immingham',
  'relationship': 'EROAD',
  'cost': '369'},
 {'src': 'Immingham',
  'dst': 'Doncaster',
  'relationship': 'EROAD',
  'cost': '74'},
 {'src': 'Doncaster', 'dst': 'London', 'relationship': 'EROAD', 'cost': '27
7'},
 {'src': 'Hoek van Holland',
  'dst': 'Den Haag',
  'relationship': 'EROAD',
  'cost': '27'},
 {'src': 'Felixstowe',
  'dst': 'Hoek van Holland',
  'relationship': 'EROAD',
  'cost': '207'},
 {'src': 'Ipswich',
  'dst': 'Felixstowe',
  'relationship': 'EROAD',
  'cost': '22'},
 {'src': 'Colchester',
  'dst': 'Ipswich',
  'relationship': 'EROAD',
  'cost': '32'},
 {'src': 'London',
  'dst': 'Colchester',
  'relationship': 'EROAD',
  'cost': '106'},
 {'src': 'Gouda', 'dst': 'Rotterdam', 'relationship': 'EROAD', 'cost': '25'},
 {'src': 'Gouda', 'dst': 'Utrecht', 'relationship': 'EROAD', 'cost': '35'},
 {'src': 'Den Haag', 'dst': 'Gouda', 'relationship': 'EROAD', 'cost': '32'},
 {'src': 'Hoek van Holland',
  'dst': 'Rotterdam',
  'relationship': 'EROAD',
  'cost': '33'}]
```

In [25]:
```
G.visualiserFolium()
```

Out[25]:

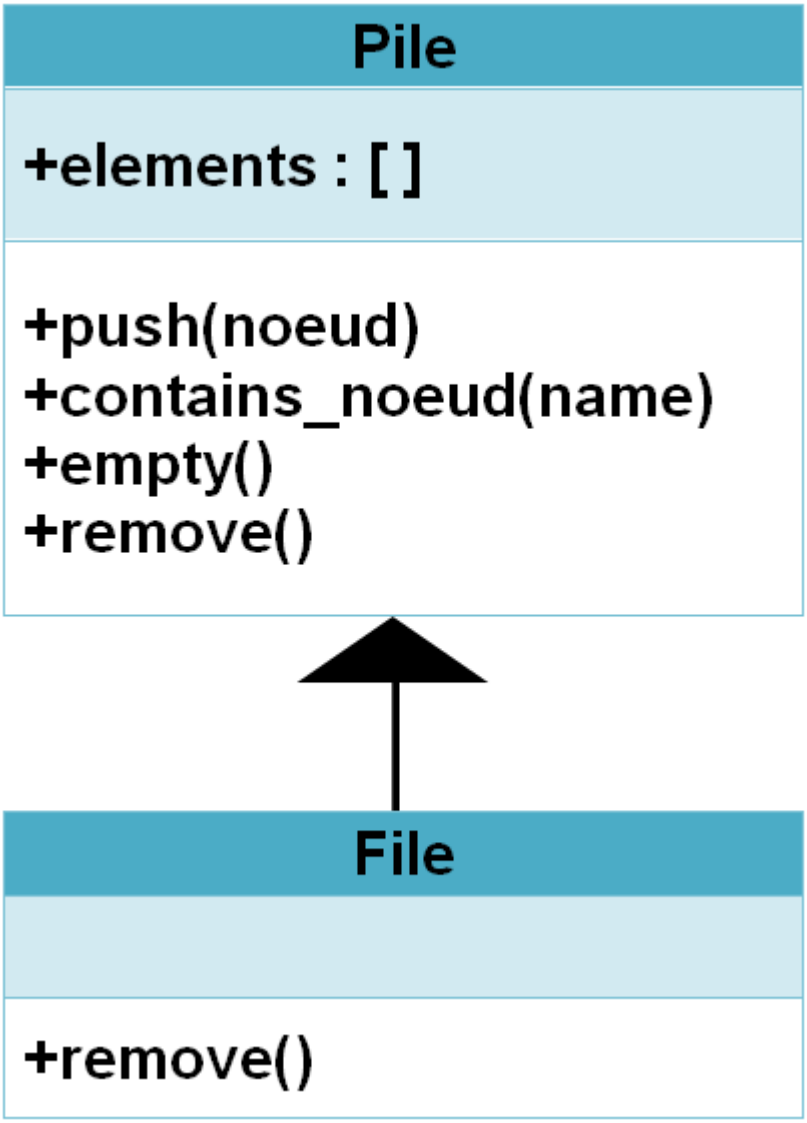
Partie 3

- Implementer les classes File et Pile en utilisant les listes en python



In [26]:
```
class Pile():
    '''
    Classe Pile: voir les definitions ci-dessous
    Dernier arrive premier servi : LIFO
    La classe dispose d'une structure de type list pour ranger les données
    Les consultations, les insertions, les suppressions se font du même cote
    '''
    def __init__(self):
        self.elements = []

    '''
    Insere un objet en tete de la pile
    '''
    def push(self,noeud):
        pass
    '''
    Retourne True si  un noeud est dans la pile
    '''


    def contains_noeud(self, name):
        pass

    '''
    Retourne true si la pile est vide
    '''
    def empty(self):
        pass
    '''
    Retourne et supprime l'element en tete de pile
    Retourne une exception si la pile est vide
    '''
    def remove(self):
        pass

#Test des structures de données Pile et File
f=Pile()
f.push("Mamadou")
f.push("Mansour")
f.push("Dame")
f.push("Khady")
print(f.elements)
f.remove() #L'élément recemmenet ajouté de la liste sera enlevé
print(f.elements)
```

```
['Khady', 'Dame', 'Mansour', 'Mamadou']
['Dame', 'Mansour', 'Mamadou']
```


Pile
+elements : [ ]
+push(noeud)
+contains_noeud(name)
+empty()
+remove()
File
+remove()

In [27]:
```
#Implémentation de la classe File par héritage

class File(Pile):

    '''
    Classe File: voir les definitions ci-dessous
    Premier arrive premier servi : FIFO
    La classe dispose d'une structure de type list pour ranger les données
    Les éléments sont enfilés (insérés) du coté arrière et défilés (retirés) d
u coté avant
    File et Pile peuvent partager certaines methodes donc utilisez l'heritage
 pour definir la classe File.
    Normalement vous ne devez changer l'implementation d'une seule methode
    '''
    def remove(self):
        pass

p=File()
p.push("Mamadou")
p.push("Mansour")
p.push("Dame")
p.push("Khady")
print(p.elements)
p.remove() #Le premier élément de la liste sera enlevé
print(p.elements)
```

```
['Khady', 'Dame', 'Mansour', 'Mamadou']
['Khady', 'Dame', 'Mansour']
```

Partie 4

- Implementer les algorithmes de parcours de graphe: BFS ET DFS
- Algorithme BFS et DFS https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur (https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur) https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur (https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur) Les deux parcours utilisent le même algorithme mais différent suivant la structure de données utilisée comme frontiére. La frontiére est une structure de données qui permet manipuler les noeuds intermediaires. #### Algorithme
- On met le noeud source dans la frontiére
- On cree une structure vide devant contenir les noeuds explorés
- Repeter
  - Si la frontiere est vide pas de solution
  - Prendre un noeud dans la frontiere (idée de suppression)
  - Si le noeud est le noeud destination alors solution
  - Sinon:
    - Mettre le noeud dans l'ensemble des noeuds deja explorés
    - Ajouter les voisins dans la frontiére s'ils ne sont pas dans la frontiere et s'ils ne sont pas deja explores

In [28]:
```
'''
Implémenter le parcours en profondeur non recursif entre deux noeuds
Prend en parametre:
- un graphe
- un noeud source
- un noeud destination
Output:
une liste contenant les noms des noeuds explorés pour aller du noeud source ve
rs le noeud destination
Vous completerez certaines parties du code
'''

def parcoursDFS(myGraphe,noeudSRC,noeudDST):
        """Trouver un parcours DFS entre noeudSRC et noeudDST"""
        pass
```

In [ ]:
```
# Visualisation des arcs en rouge et visualisation des noeuds parcourus pour a
ller du premier au deuxieme noeud en vert
g = Graphe()
g.visualiserFolium(explored=list(parcoursDFS(g,g.noeuds[0],g.noeuds[1])[0]))
```

Consignes

- Tous les scripts Python doivent être postés sur votre dépôt GitHub
- Générez un lien GitHub puis envoyez le à l'adresse suivante :
- projetdesetudiants1@gmail.com avec comme objet (Projet 2 POO2 GL 2021)
- Le projet se fera au plus par groupe de 2 étudiants ;
- Chaque étudiant spécifiera la partie qu'il aura développé dans le contenu de l'email ;
- Vous commenterez vos codes, pour cela utilisez les recommandations de PEP8 Python pour mettre du style dans vos codes ;
- Le travail doit être rendu au plus tard le 23 Novembre 2021 avant 00h00.
- Le plagiat sera reconnu et sera sévérement sanctionné donc travaillez sérieusement et honnétement.

In [ ]:
```
```