TP - Classement de villes par ABR

Objectifs:

> Utiliser un arbre binaire de recherche (ABR) pour classer des villes selon différents critères.

Position	Commune	Département	
# 1	Paris	Paris	2145906 hab
# 2	Marseille	Bouches-du-Rhône	870 321 hab
#3	Lyon	Rhône	522 228 hab
# 4	Toulouse	Haute-Garonne	498 003 hab

1 Travail à faire

1. Récupérer le fichier villes.csv qui contient la liste des 200 plus grandes villes de France dans un ordre aléatoire.

Chaque ligne de ce fichier contient les 5 informations suivantes :

- ▶ Nom de la ville
- ⊳ Numéro du département
- ▶ Nombre d'habitants
- ⊳ Superficie (en km²)
- ▶ Rang au niveau national (critère : nombre d'habitants)

Voici à quoi ressemblent les deux premières lignes de ce fichier :

```
Saint-Priest, 69, 40944, 29.7, 155
Versailles, 78, 85761, 26.2, 46
```

2. Créer une classe Ville telle que ci-après :

```
class Ville :

def __init__(self,liste) :
    ''' Le constructeur'''

self.nom = liste[0]

self.departement = int(liste[1])

self.population = int(liste[2])

self.superficie = float(liste[3])

self.rang = int(liste[4])

# creation d'un objet:
liste = ['Nice', '6', '343123','71.9','5']

ville=Ville(liste)
```

- 3. Vérifier l'accès aux différents attributs.
- 4. Écrire une méthode getRang(self), qui retourne le rang de la ville.
- 5. Écrire une méthode getSuperficie(self), qui retourne la superficie de la ville.
- 6. Écrire une méthode afficherVille(self), qui affiche les 5 données de la ville.
- 7. Écrire une méthode afficherNom(self), qui affiche le nom de la ville.

8. Utiliser la classe Noeud ci-dessous vue au chapitre précédent sur les ABR:

```
class Noeud:
       '''Structure de donnees de type ABR'''
3
       def __init__(self, liste, left=None, right=None):
           self.ville = Ville(liste)
           self.left = left
           self.right = right
       def __str__(self):
           return str(self.ville.afficherVille())
10
       def estFeuille(self):
           if not self.left and not self.right:
               return True
14
           else:
               return False
```

9. Rajouter la méthode inserer(self, liste):

```
def inserer(self, liste):
    '''Methode pour inserer une ville dans l'ABR'''

ville = Ville(liste)
    if ville.getRang() < self.ville.getRang() :
        if self.left is None:
            self.left = Noeud(liste)
    else:
        self.left.inserer(liste)

else :
    if self.right is None:
        self.right = Noeud(liste)
    else :
        self.right is None:
        self.right is None:
        self.right = Noeud(liste)</pre>
```

L'arbre est alors " $rang\acute{e}$ " suivant le rang de la ville.

10. Rajouter dans votre programme principal les lignes de code ci-dessous :

```
import csv
liste_villes=[]

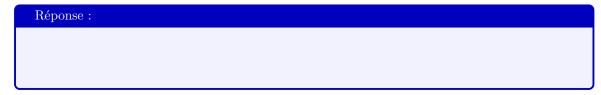
with open("villes.csv",'r',encoding='utf-8') as f:
lecteur=csv.reader(f,delimiter=',')
for ligne in lecteur:
liste_villes.append(ligne)
```

Rappel: il faut que le fichier villes.csv soit dans le même dossier que votre fichier Python.

11. Vérifier que l'on a bien créé une liste liste_villes contenant les lignes du fichier villes.csv.

Questions:

⊳ Pour obtenir un arbre équilibré, il faut choisir la racine avec un rang de 100. Pourquoi?



▷ En quoi est-ce important pour une recherche dans l'arbre?

```
Réponse :
```

12. Construire l'arbre avec le code ci-dessous :

```
liste = ["Maisons-Alfort", '94', '51091', '5.4', '100']
noeud = Noeud(liste)
print(noeud)

for el in liste_villes:
    ville = Ville(el)
noeud.inserer(el)
```

- 13. Récupérer la méthode (ou la fonction) parcours_infixe() créée dans un chapitre précédent et l'adapter pour faire afficher les villes par ordre croissant de leur rang.
- 14. Modifier ce qu'il faut pour faire afficher les villes par ordre croissant de leur superficie.
- 15. En utilisant cette nouvelle construction, écrivez une méthode rechercheMax(self) qui renvoie la ville ayant la plus grande superficie.
- 16. Écrire une méthode rechercher(self,rang), qui retourne la ville dont le rang est rang.