Les arbres binaires - Exercices

Exercice 1 :

La classe Noeud étant celle du cours, l'instanciation d'une racine est :

```
Noeud("A",\
      Noeud("B",Noeud("D"),Noeud("E",Noeud("G"),Noeud("H"))),\
      Noeud("C",None,Noeud("F",Noeud("I"),None)))
```

1. Dessiner l'arbre correspondant.
2. Donner la liste des sommets obtenus par un parcours :
   - en profondeur préfixe, infixe, postfixe
   - en largeur

Exercice 2 :

Les classes Noeud et ABR sont celles du cours. L'algorithme récursif suivant permet de modifier la racine de l'ABR en y insérant une valeur à la bonne place.

Insertion(racine,val)

```
si racine est vide alors racine ← feuille de valeur val
sinon
    si val ≤ valeur de la racine alors
        si fils gauche de racine est vide alors
            fils gauche de la racine ← feuille de valeur val
        sinon
            Insertion(fils gauche de la racine,val)
    sinon
        si fils droit de racine est vide alors
            fils droit de la racine ← feuille de valeur val
        sinon
            Insertion(fils droit de la racine,val)
```

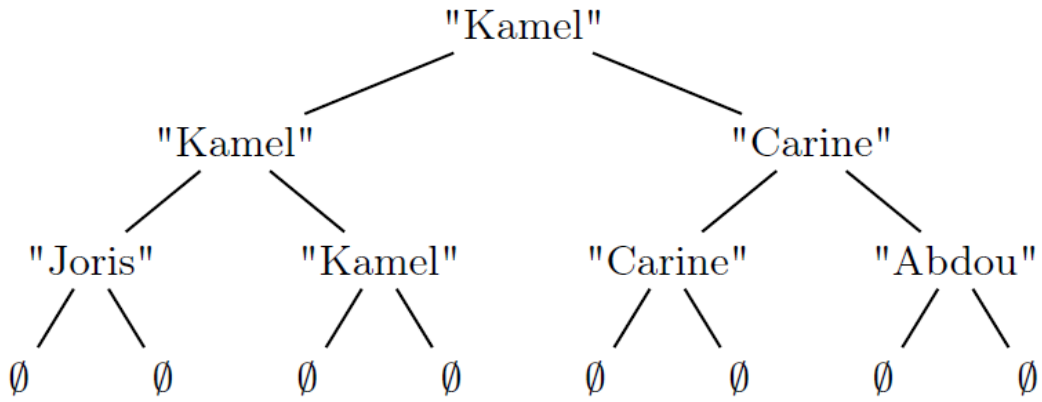
1. Compléter les classes Noeud et ABR avec une méthode récursive permettant d'insérer une valeur dans l'ABR de la classe.
2. Le parcours en profondeur infixe donne les nœuds d'un ABR classés par ordre croissant.
   (a) Ecrire une méthode infixe de la classe Noeud qui retourne la liste des nœuds rencontrés dans l'ordre du parcours infixe.
   (b) Dans le Shell saisir l'instruction help(sorted). Que fait la méthode sorted( ) ?
   (c) Ecrire une méthode est_ABR de la classe ABR qui retourne le booléen True si l'arbre binaire est un ABR et False sinon.
   (d) Créer à l'aide de la méthode insertion un arbre binaire de taille 10 dont les valeurs sont des entiers naturels aléatoires compris entre 0 et 100.
   Vérifier que l'arbre binaire obtenu est effectivement un ABR.
3. Un peigne gauche est un ABR sans fils droit.
   (a) Dessiner l'arbre obtenu en partant d'un ABR vide et en y insérant successivement les valeurs de la liste [5, 4, 3, 2, 1] est un un peigne gauche.
   (b) Ecrire une méthode récursive peigne_gauche qui retourne vrai si un ABR est un peigne gauche et faux sinon.
   (c) Vérifier la cohérence de la méthode avec les ABR suivants obtenus à partir d'un ABR vide puis en y insérer toutes les valeurs successives d'une liste L avec :
      - L=[7,4,3,1]
      - L=[5,5,5]
      - L=[5,4,5]
      - L=[2,5,6,1]

Exercice 3 :

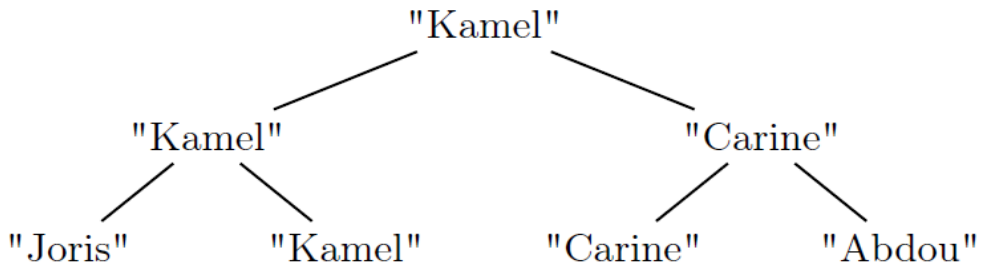
La fédération de badminton souhaite gérer ses compétitions à l'aide d'un logiciel.
Pour ce faire, une structure arbre de compétition a été définie récursivement de la façon suivante : un arbre de compétition est soit l'arbre vide, noté $\emptyset$, soit un triplet composé d'une chaîne de caractères appelée valeur, d'un arbre de compétition appelé sous-arbre gauche et d'un arbre de compétition appelé sous-arbre droit.

Partie I

On représente graphiquement un arbre de compétition de la façon suivante :



Pour alléger la représentation d'un arbre de compétition, on ne notera pas les arbres vides.
L'arbre ci-dessus sera donc représenté par l'arbre A suivant :



Cet arbre se lit de la façon suivante :

- 4 participants se sont affrontés : Joris, Kamel, Carine et Abdou.
  Leurs noms apparaissent en bas de l'arbre, ce sont les valeurs de feuilles de l'arbre.
- Au premier tour, Kamel a battu Joris et Carine a battu Abdou.
- En finale, Kamel a battu Carine, il est donc le vainqueur de la compétition.

Pour s'assurer que chaque finaliste ait joué le même nombre de matchs, un arbre de compétition a toutes ces feuilles à la même hauteur.

1. Quel est la taille et la hauteur de cet arbre ?
2. Quelles sont les feuilles et la racine de cet arbre ?

On implémente un arbre de compétition à l'aide d'une liste :

- vide si l'arbre est vide
- de trois éléments :
  - le premier représente la valeur du nœud,
  - le second représente l'arbre gauche (qui est donc une liste vide ou de trois éléments)
  - le troisième représente l'arbre droit (qui est don une liste vide ou de trois éléments

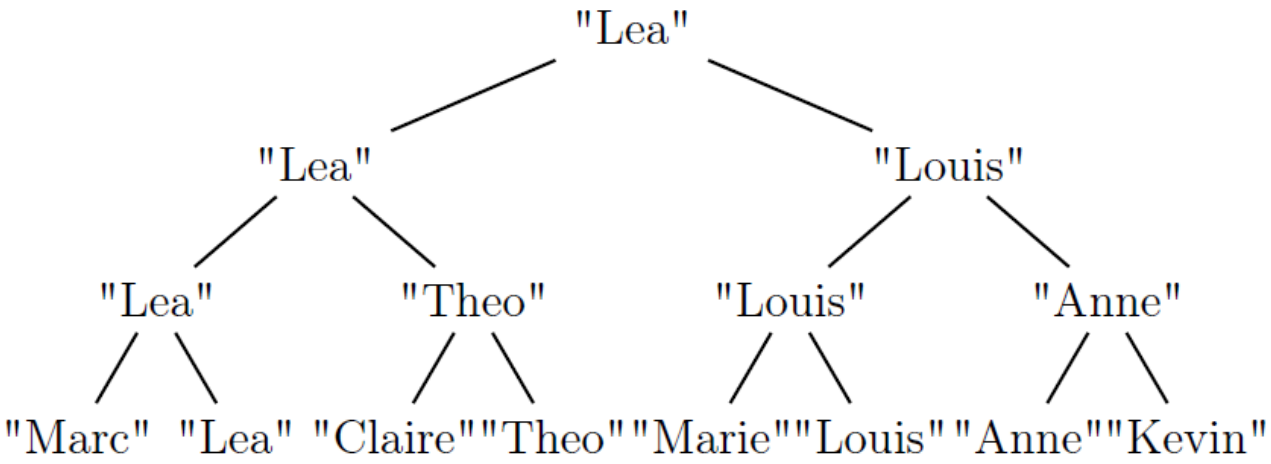
L'arbre A est donc créé par les instructions suivantes :

```
A=[]
A.append("Kamel")
A.append(["Kamel",["Joris",[],[]],["Kamel",[],[]]])
A.append(["Carine",["Carine",[],[]],["Abdou",[],[]]])
```

1. Créer puis vérifier la cohérence des fonctions suivantes qui prennent toutes pour paramètre un arbre :
   - est_vide qui revoie le booléen True si l'arbre est vide est False sinon
   - racine qui renvoie la racine de l'arbre
   - gauche qui renvoie le sous-arbre gauche de l'arbre.
   - droit qui renvoie le sous-arbre droit de l'arbre.
2. Créer puis vérifier la cohérence les fonctions suivantes à l'aide des précédentes et qui prennent toutes pour paramètre un arbre :
   - vainqueur qui retourne le nom du vainqueur du tournoi
   - finale qui retourne la liste des deux finalistes

Partie II

Pour toutes les questions de cette partie, on considère l'arbre B de compétition ci-dessous :



3. Créer les fonctions suivantes puis les valider en utilisant l'arbre B :
   - Une fonction occurrences, de paramètres arbre et nom, qui renvoie le nombre d'occurrences du joueur nom dans l'arbre
   - Une fonction a_gagne, de paramètres arbre et nom, qui renvoie le booléen True si le joueur nom a gagné au moins un match dans la compétition
4. On souhaite programmer une fonction nombre_matchs qui prend pour argument l'arbre de compétition et le nom d'un joueur nom et qui renvoie le nombre de matchs joués par le joueur nom dans la compétition.
   Exemples : nombre_matchs(B,"Lea") vaut 3 et nombre_matchs(B,"Marc") vaut 1.
   a. Expliquer pourquoi la proposition ci-dessous n'est pas satisfaisante :

```
def nombre_matchs(arbre,nom):
    """arbre, str -> int"""
    return occurences(arbre,nom)
```

   b. Proposer une correction de la fonction nombre_matchs puis la créer et la tester.
5. Compléter puis valider la fonction liste_joueurs ci-dessous qui renvoie une liste contenant les participants au tournoi, chaque nom ne devant figurer qu'une seule fois dans la liste.

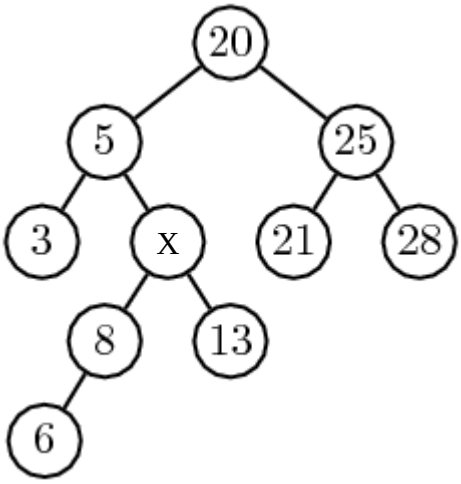
```
def liste_joueurs(arbre):
    if est_vide(arbre):
        return ***************
    elif **************** and ****************:
        return [racine(arbre)]
    else:
        return **************** +liste_joueurs(droit(arbre))
```

<u>Remarque :</u> vous ne devez utiliser que des fonctions existantes.

Exercice 4 :

Partie I

On considère l'arbre binaire de recherche A ci-contre.



1. Quelles sont la taille et la hauteur de cet arbre ?
2. Quelles sont les feuilles ?
3. Quelles sont les valeurs possibles pour X ?
4. Donner la liste des nœuds si l'on fait un parcours en profondeur infixe de l'arbre. Que constate-t-on ?
5. Si l'on souhaite obtenir la liste des nœuds par niveau, c'est à dire 20, 5, 25, 3 , X, … quel type de parcours faut-il faire ?

Partie II

Dans la suite de l'exercice, on prendra X = 12.
On supposera par la suite que les ABR initiaux ne sont jamais vides, c'est à dire qu'il y a toujours initialement au moins une racine.

L'implémentation choisie pour l'ABR est une classe comme illustré dans le programme ci-dessous :

```
#---CLASSES---
class ABR:
    def __init__(self,data):
        self.V=data # valeur d'un noeud
        self.G=None # ABR gauche
        self.D=None # ABR droit

    def insert(self,val): # méthode récursive
        if val<=self.V:
            if self.G==None: # on est au bon endroit pour insérer val
                self.G=ABR(val)
            else:
                self.G.insert(val) # méthode récursive sur l'arbre gauche
                ''' si l'ABR gauche était vide alors self.G serait du type None,
                    or la méthode insert ne peut s'appliquer qu'à un objet de type ABR
                    self.G.insert(val) aurait déclenché une erreur de type
                    'NoneType' object has no attribute 'insert'
                '''
        else:
            if self.D==None: # on est au bon endroit pour insérer val
                self.D=ABR(val)
            else:
                self.D.insert(val) # méthode récursive sur l'arbre droit

#---PROGRAMME PRINCIPAL---
A=ABR(20)
LA=[5,25,3,12,21,28,8,13,6]
for valeur in LA:
    A.insert(valeur)
```

Pour chaque méthode demandée,

- ***vous devez proposer une solution récursive***
- ***vous trouverez une aide dans le fichier exercice4_aide.py***
- ***vous vérifierez la cohérence de votre programmation avec l'ABR A***

1. Ecrire une méthode taille qui renvoie le nombre de nœuds de l'ABR
2. Ecrire une méthode hauteur qui renvoie la hauteur de l'ABR
3. Le parcours de l'ABR en profondeur infixe donne la liste des nœuds classés par ordre croissant. On donne ci-dessous la méthode croissant correspondante :

```
def croissant(self,L=[]): # perso, je préfère le pain au chocolat
    if self.G!=None:
        self.G.croissant(L)
    L.append(self.V)
    if self.D!=None:
        self.D.croissant(L)
    return L
```

   Ecrire une méthode decroissant qui retourne la liste des nœuds classés par ordre décroissant.
4. Ecrire une méthode recherche de paramètre val qui retourne True si l'ABR contient val et False sinon.
5. Le minimum d'un ABR est toujours la feuille la plus à gauche.
   Ecrire une méthode minimum qui détermine le minimum d'un ABR (sans appeler les methodes croissant ou decroissant qui effectuent un parcours total !)
6. Le minimum d'un ABR est toujours la feuille la plus à droite.
   Ecrire une méthode maximum qui détermine le maximum d'un ABR (même contrainte).

Partie III

On dit qu'un ABR est « bien construit » s'il n'existe pas d'ABR de hauteur inférieure qui pourrait contenir tous ses nœuds.

1. Montrer que l'ABR A de la partie II n'est pas « bien construit » en proposant un autre ABR B qui possède les mêmes nœuds mais dont la hauteur est 4.
2. Soit N la taille et H la hauteur d'un ABR. Donner un encadrement de N.
3. Montrer que pour qu'un ABR de hauteur H soit « bien construit », il faut qu'il ait une taille minimum de 2^{H-1} .
4. Ecrire une méthode qui retourne le booléen True si l'ABR est bien construit et False sinon. Vérifier la cohérence de votre méthode avec les ABR A et B.