ARBRES BINAIRES : Implémentation

1. Interface

- Dans cette feuille, nous allons implémenter la structure d'arbre binaire en python à l'aide la programmtion objet.
- Voici les opérations de base qui seront définies :
  - Créer un arbre vide
  - Indiquer si un arbre est vide
  - Calculer sa taille
  - Calculer sa hauteur
- Même s'il est possible que des noeuds aient la même valeur, on supposera ici, des raisons de simplification, que les valeurs de tous les noeuds d'un arbre considéré sont distinctes deux à deux.

2. Classes

- Il y a de nombreuses façons de représenter un arbre binaire en python.
- Une façon traditionnelle consiste à représenter chaque noeud par un objet d'une classe appelée `Noeud`. Un objet de cette classe contient trois attributs initialisés par le constructeur :
  - `gauche` pour le sous-arbre gauche.
  - `valeur` pour la valeur contenue dans le noeud.
  - `droite` pour le sous-arbre droit.
- L'arbre vide est représenté par la valeur `None`.
- Comme pour les listes chaînées, on encapsule un arbre binaire dans une classe `AB`. Chaque objet de cette classe contient un unique attribut, `racine` qui est l'arbre qu'il représente.Le constructeur de cette classe initialise cet attribut avec la valeur `None` par défaut (arbre vide).

In [1]:
```
class Noeud:
    '''Classe définissant un noeud d'arbre binaire'''
    def __init__(self,gauche,valeur,droite):
        self.gauche=gauche
        self.valeur=valeur
        self.droite=droite

    #Ex1
    def __str__(self):
        '''Renvoie la valeur du noeud (str)'''
        pass


class AB:
    '''Classe définissant un Arbre Binaire'''
    def __init__(self, racine=None):
        self.racine=racine

    #Ex3
    def est_vide(self):
        '''renvoie True si et seulement si l'arbre est vide'''
        pass

    #Ex4
    def hauteur(self):
        '''renvoie la hauteur de l'arbre'''
        pass

    #Ex5
    def taille(self):
        '''renvoie la taille de l'arbre'''
        pass
```

Exercice 1 :

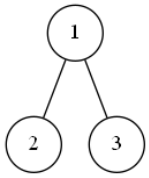
Ecrire la méthode `__str__(self)` de la classe `Noeud` qui affiche la valeur contenue dans un noeud.

In [2]:
```
#Afficher la valeur d'un noeud
n1=Noeud(None,42,None)
print(n1)
```

```
42
```

Exercice 2 :

Trouver deux façons de construire l'arbre ci-contre(on ne demande pas de le dessiner):



In [3]:
```
#méthode 1 :




print(T1.racine)
print(T1.racine.gauche)
print(T1.racine.droite)
```

```
1
2
3
```

In [4]:
```
#méthode 2 :

print(T1.racine)
print(T1.racine.gauche)
print(T1.racine.droite)
```

```
1
2
3
```

3. Méthodes

Dans cette partie , nous allons ajouter quelques fonctionnalités sur les arbres binaires. Ces méthodes sont à ajouter dans le code de définition de la classe `AB` . De par leur définition , les arbres binaires se prêtent bien à la programmation récursive.

Exercice 3 : Arbre vide

Ecrire la méthode `est_vide(self)` qui renvoie `True` si l'arbre est vide , `False` sinon.

In [5]:
```
#tests arbre vide
T1=AB(Noeud(Noeud(None,2,None),1,Noeud(None,3,None)))
T2=AB()
assert(T1.est_vide()==False)
assert(T2.est_vide()==True)
```

Exercice 4 : Calculer la hauteur

Ecrire la méthode `hauteur(self)` qui renvoie la hauteur de l'arbre.

Aide :

- La hauteur d'un arbre vide est 0
- Sinon, la hauteur d'un arbre est égale au maximum de la hauteur du sous-arbre gauche et du sous-arbre droit, auquel on ajoute 1.

In [6]:
```
#tests hauteur
T1=AB(Noeud(Noeud(None,2,None),1,Noeud(None,3,None)))
T2=AB(Noeud(None,1,None))
T3=AB()

assert(T1.hauteur()==2)
assert(T2.hauteur()==1)
assert(T3.hauteur()==0)
```

Exercice 5 : Calculer la taille

Ecrire la méthode `taille(self)` qui renvoie la taille de l'arbre.

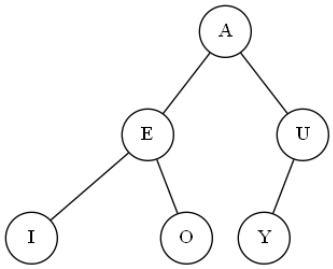
Aide :

- La taille d'un arbre vide est 0
- Sinon, la taille d'un arbre est égale à la somme de la hauteur du sous-arbre gauche et du sous-arbre droit, à laquelle on ajoute 1.

In [7]:
```
#tests hauteur
T1=AB(Noeud(Noeud(None,2,None),1,Noeud(None,3,None)))
T2=AB(Noeud(None,1,None))
T3=AB()

assert(T1.taille()==3)
assert(T2.taille()==1)
assert(T3.taille()==0)
```

4. Parcours

Dans les exercices suivants, on utilisera l'arbre ci-contre, défini par le programme ci-dessous pour tester les fonctions de parcours en profondeur, qui seront définies récursivement.



Exercice 6 :

Ecrire la liste des sommets visités de cet arbre lors d'un parcours en profondeur :

1. préfixe :
2. postfixe :
3. infixe :

In [8]:
```
#construction de l'arbre donné en exemple.
A=Noeud(None,'A',None)
E=Noeud(None,'E',None)
U=Noeud(None,'U',None)
I=Noeud(None,'I',None)
O=Noeud(None,'O',None)
Y=Noeud(None,'Y',None)

A.gauche=E
A.droite=U
E.gauche=I
E.droite=O
U.gauche=Y

arbre=AB(A)
```

Exercice 7 : parcours en profondeur préfixe

Ecrire la fonction `profondeur_prefixe(A)` qui affiche les noeuds de l'arbre `A` parcouru en profondeur préfixe. Aide :

- Si l'arbre est vide, alors on s'arrête
- Sinon :
  - On affiche la racine de l'arbre.
  - On parcourt en profondeur préfixe le sous-arbre gauche de l'arbre.
  - On parcourt en profondeur préfixe le sous-arbre droit de l'arbre.

In [9]:
```
def profondeur_prefixe(A):
    '''Affiche les noeuds de l'arbre A
    parcouru en profondeur préfixe'''

    pass
```

In [10]:
```
profondeur_prefixe(arbre) # A E I O U Y
```

```
A E I O U Y
```

Exercice 8 : parcours en profondeur postfixe

Ecrire la fonction `profondeur_postfixe(A)` qui affiche les noeuds de l'arbre `A` parcouru en profondeur postfixe. Aide :

- Si l'arbre est vide, alors on s'arrête
- Sinon :
  - On parcourt en profondeur postfixe le sous-arbre gauche de l'arbre.
  - On parcourt en profondeur postfixe le sous-arbre droit de l'arbre.
  - On affiche la racine de l'arbre.

In [11]:
```
def profondeur_postfixe(A):
    '''Affiche les noeuds de l'arbre A
    parcouru en profondeur préfixe'''

    pass
```

In [12]:
```
profondeur_postfixe(arbre) # I O E Y U A
```

```
I O E Y U A
```

Exercice 9 : parcours en profondeur infixe

Ecrire la fonction `profondeur_infixe(A)` qui affiche les noeuds de l'arbre `A` parcouru en profondeur infixe. Aide :

- Si l'arbre est vide, alors on s'arrête
- Sinon :
  - On parcourt en profondeur préfixe le sous-arbre gauche de l'arbre.
  - On affiche la racine de l'arbre.
  - On parcourt en profondeur préfixe le sous-arbre droit de l'arbre.

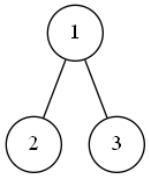
In [13]:
```
def profondeur_infixe(A):
    '''Affiche les noeuds de l'arbre A
    parcouru en profondeur préfixe'''

    pass
```

In [14]:
```
profondeur_infixe(arbre) # I E O A Y U
```

```
I E O A Y U
```

5. Exercices

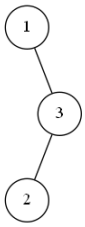
Exercice 10 : Affichage d'un arbre



Il est difficile de visualiser un arbre défini avec les classes et méthodes précédentes. Le but de cet exercice est d'écrire une fonction qui affiche la représentation d'un arbre à l'aide parenthèses ouvrantes et fermantes. L'arbre ci-contre sera représenté ainsi : `((2)1(3))`

PARTIE A :

1. Ecrire la représentation de l'arbre suivant.

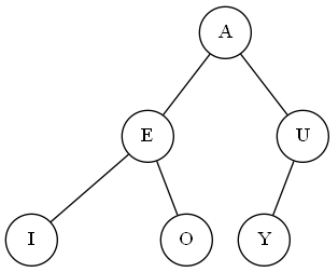


1. Dessiner l'arbre dont la représentation est : `((B(C))A(D))`

1. D'une manière générale, décrire comment retrouver l'arbre à partir de cette écriture aec les parenthèses.

PARTIE B :

Ecriture de la fonction `affiche(arbre)` qui prend en paramètre un arbre et qui affiche sa représentation.Les tests seront effectués sur l'arbre ci-contre :



1. Donner le résultat de la foncction `affiche` sur cet arbre : `(((I)E(O))A((Y)U))`
2. Voici le descriptif de cette fonction :
   - Si l'arbre est vide alors on sort de la fonction.
   - On ouvre une parenthèse.
   - On affiche le sous-arbre gauche.
   - On affiche la valeur de la racine.
   - On affiche le sous-arbre droite.
   - On ferme une parenthèse.

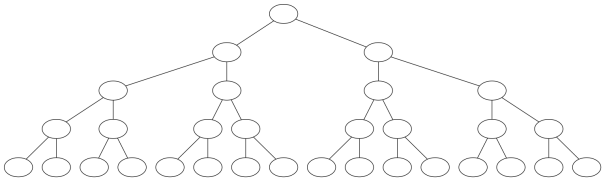
Ecrire ci-dessous la fonction.

In [26]:
```
def affiche(arbre):
    pass

affiche(arbre)
```

```
(((I)E(O))A((Y)U))
```

Exercice 11 : Arbre binaire parfait

Un arbre binaire parfait est un arbre binaire dont chaque noeud(sauf les feuilles) possède exactement deux enfants.Toutes les feuilles sont à la même profondeur.
Voici un arbre binaire parfait de hauteur 5 :



Ecrire la fonction `parfait(h)` qui prend en paramètre la hauteur de l'arbre souhaité et qui renvoie un arbre parfait de hauteur `h` .

Aide :

- Si `h` vaut 0, on construit l'arbre vide.
- Sinon, Il suffit de construire l'arbre dont la racine est `h` , et dont les sous-arbres gauche et droite sont des arbres parfaits de hauteur `h-1` .
- On pourra supposer que la valeur des noeuds est `h` (récursivement)

In [68]:
```
def parfait(h):
    '''Construit un arbre binaire parfait de hauteur h
    h : entier >= 0
    return : un arbre binaire '''

    pass
```

In [69]:
```
#tests arbre parfait
A2=parfait(2)
affiche(A2) # ((1)2(1))
```

```
((1)2(1))
```

In [70]:
```
A4=parfait(4)
affiche(A4) #((((1)2(1))3((1)2(1)))4(((1)2(1))3((1)2(1))))
```

```
((((1)2(1))3((1)2(1)))4(((1)2(1))3((1)2(1))))
```

In [56]:
```
A0=parfait(0)
affiche(A0) #rien !
```