Arbres 3/3 : Il est temps d'allumer un ordinateur

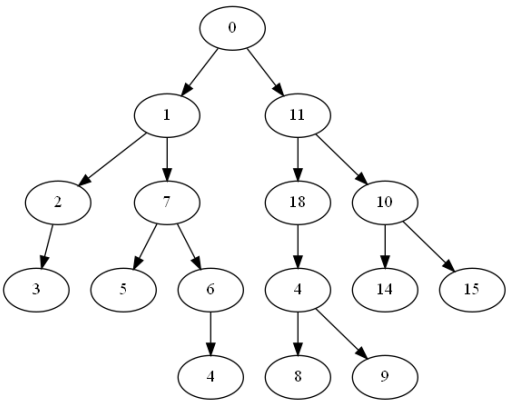
- Pour pouvoir concevoir et afficher des arbres, vous aurez besoin :
  - du module `graphviz` : permet de concevoir des arbres, nécéssite d'abord l'installation sur votre machine du logiciel 'graphviz' (https://graphviz.org/download/)
  - de la classe `Arbre` : c'est un code (que nous n'avez pas à comprendre) qui contient des classes et des méthodes simplifiant la conception d'arbres.
- Rappel pour installer un module sous python :
  - `pip install nom_du_module`
  - Si cela ne fonctionne pas : `python -m pip install nom_du_module`

In [ ]:
```
#Module nécéssaires
from graphviz import *

#classe Arbre
class Arbre :
    """ Classe pour représenter des arbres """

    def __init__(self, label, enfants=None) :
        """
        Crée un arbre à partir d'un label qui ne DOIT PAS être None.
        Exemples :
        a = Arbre('A')
        c = Arbre('C', enfants = [a, Arbre('B')])
        """
        assert(label!=None)
        self.label = label
        self.enfants = enfants or []

    # ======================================================================
    # Une fonctionnalité facile à comprendre  ============================
    # ======================================================================

    def ajoute(self,*a) :
        """
        Ajoute un ou plusieurs arbres a comme enfant(s) de la racine
        (en fin de liste)
        """
        self.enfants.extend(list(a))


    # ======================================================================
    # Redéfinir ce qu'affiche le print
    # ======================================================================

    def __str__(self) :
        """ Renvoie une représentation lisible de l'objet sous
        forme de chaîne de caractères
        """
        affiche=[]
        for noeud in self.enfants:
            affiche.append(str(noeud))
        affiche=",".join(affiche)
        if affiche!="" :
            return ""+str(self.label)+"--["+affiche+"]"
        else :
            return str(self.label)

    # ======================================================================
    # Fonctionnalités d'affichage avec Graphviz                ==========
    # qu'il est inutile de regarder                            ==========
    # ======================================================================

    def _innerdot(self) :
        stri=""
        for a in self.enfants :
            stri+="{0} -> {1} [label=\"\"];\n".format(id(self),id(a))
        col="white"
        infos=str(self.label)
        attr="fillcolor=\"#888888ff\""
        stri+='\"{label}\"  [label=\"{infos}\", style = filled, peripheries = 1, \
                fillcolor = {col}, color = black];\n'.format(label=id(self),infos=infos,col=col)
        for a in self.enfants :
            stri+=a._innerdot()
        return stri

    def dot(self,printinfos=True) :
        """ Crée une chaîne de caractère contenant une
        description de l'arbre pour le programme dot (graphviz)
        """
        return "digraph g {\n"+self._innerdot()+"}\n"
```

In [ ]:
```
#Exemple d'utilisation

a1 = Arbre(1)
b = Arbre(2)
c = Arbre(3)
d = Arbre(4)
a1.ajoute(b, c)
b.ajoute(d, Arbre(5))

#affichage de l'arbre
code_dot = a1.dot()
Source(code_dot)
```

Exercice 1 :

Ecrire le code qui permet de générer l'arbre ci-contre.



Exercice 2 :

On veut écrire une fonction récursive `genere_arbre(hauteur, max_enfants, n_max)` qui génère des arbres de façon aléatoire.Elle prend en paramètres:

- `hauteur` : hauteur de l'arbre généré, de type entier positif.
- `max_enfants` : nombre maximal d'enfants de chaque sous-arbre de l'arbre, de type entier positif.
- `n_max` : valeur maximale des étiquettes de l'arbre , de type entier positif.

Ainsi, l'appel de `genere_arbre(3, 3, 16)` doit générer un arbre de hauteur 3, dont chaque sous arbre comporte un nombre aléatoire d'enfants compris entre 0 et 3 et dont chaque étiquette est un nombre aléatoire compris entre 0 et 16.

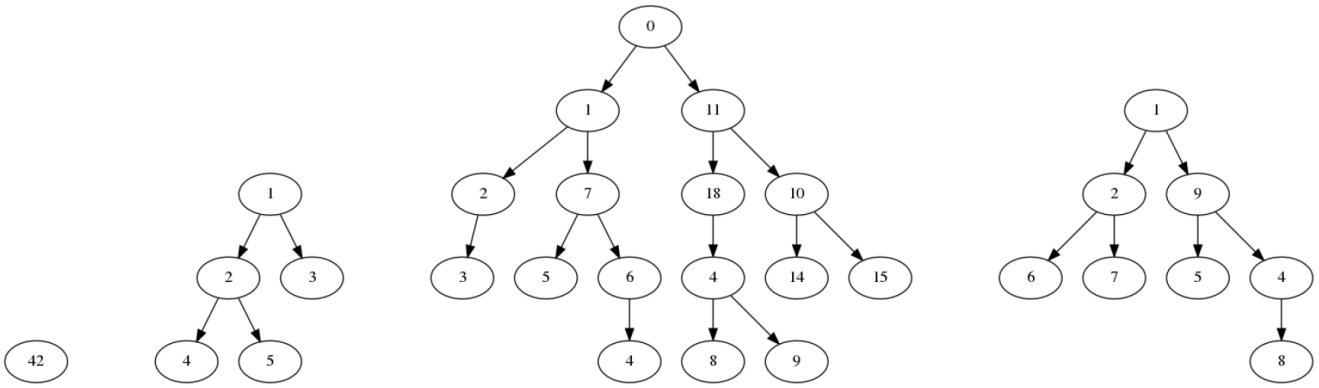
Voici le descriptif de cette fonction:

- On crée un arbre sans enfants avec une étiquette aléatoire.
- Le cas de base est celui ou `hauteur` vaut 1. Dans ce cas, on renvoie cet arbre.
- Le cas récursif genere des sous-arbres de hauteur `hauteur-1` pour chacun des enfants de l'arbre.

In [ ]:
```
from random import *

def genere_arbre(hauteur, max_enfants, n_max):
    tree=Arbre(randint(1,n_max))
    #code à compléter









    return tree


#Exemple :
H=3 #hauteur de l'arbre
E=3 #nombre max d'enfants
N=16 #valeur max étiquette
TREE=genere_arbre(H,E,N)

#affichage de l'arbre
code_dot = TREE.dot()
Source(code_dot)
```

Exercice 3 :

On veut écrire la fonction récursive `taille(arbre)` qui prend en paramètre un arbre et qui calcule sa taille, c'est à dire son nombre d'étiquettes. Dans cet exercice, on considère les arbres `a0` , `a1` , et `a5` respectivement affichés ci-dessous:



__1.__ Indiquer ce que doit renvoyer: * `taille(a0)` : * `taille(a1)` : * `taille(a5)` :

2. Ecrire le code de cette fonction.On pourra utiliserla fonction `est_une_feuille(arbre)` .

In [ ]:
```
def est_une_feuille(arbre):
    """
    Prend un arbre en parametre
    renvoie True si  cet arbre est une feuille (s'il n'a pas d'enfants)
    et False sinon
    """
    return arbre.enfants == []

def taille(arbre):









    pass


#Les arbres a1 et a5 sont déjà créées plus haut
a0=Arbre(42)

print (taille(a0), taille(a1),taille(a5))
```

Exercice 4:

On veut écrire la fonction récursive `contient(etiquette,arbre)` qui prend en paramètres une étiquette et un arbre et qui renvoie `True` si l'étiquette est présente et `False` sinon Dans cet exercice, on considère les arbres `a0` , `a1` , et `a5` précédents.

1. Indiquer ce que doit renvoyer:

- `contient(42,a0)` :
- `contient(42,a1)` :
- `contient(18,a5)` :

2. Ecrire le code de cette fonction.On pourra utiliser la fonction `est_une_feuille(arbre)` .

In [ ]:
```
def contient(etiquette,arbre):









    pass

print(contient(42,a0), contient(42,a1),contient(18,a5))
```