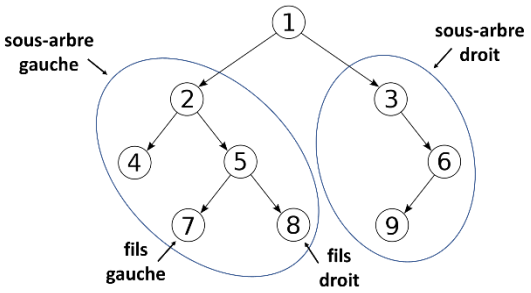
1. Définition

Un **arbre binaire** est un cas particulier d'un arbre.
C'est un arbre dont chaque nœud a **au plus deux fils** (appelé **fils gauche** et **fils droit**).



Il est important de noter que l'on peut voir les arbres comme des structures récursives :

- les fils d'un nœud sont des arbres (sous-arbre gauche et sous-arbre droit), ces arbres sont eux-mêmes constitués d'arbres…

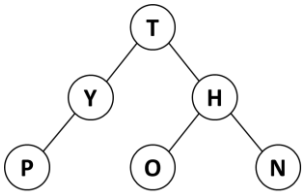
2. Implémentation des arbres binaires

Plusieurs implémentations sont possibles pour représenter un arbre binaire. On peut utiliser la programmation orientée objet, un dictionnaire, des tuples de tuples, des tableaux de tableaux…

3. Implémentation des arbres binaires avec des tuples

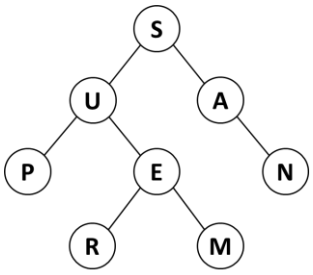
3.1. Représentation

- Un arbre vide est représenté par un tuple vide.
- Chaque nœud est un tuple de trois valeurs contenant dans l'ordre :
  - l'étiquette du nœud ;
  - le sous-arbre gauche (en l'absence de fils, on considère que le fils est None (vide)) ;
  - le sous-arbre droit (en l'absence de fils, on considère que le fils est None (vide)).



Voici la correspondance entre l'arbre ci-dessus et son écriture à base de tuples :
`('T', ('Y', ('P', None, None), None), ('H', ('O', None, None), ('N', None, None)))`

✍ Donner l'expression à base de tuples de l'arbre ci-contre :



✍ Soit l'écriture ci-dessous représentant un arbre, dessiner cet arbre.
`('T', ('A', ('M', None, None), None), ('R', ('I', None, ('X', None, None)), None))`

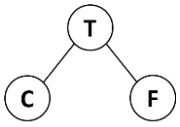
3.2. Implémentation de l'arbre

Soit la fonction **noeud** qui crée un nœud en donnant un nom à ce nœud et en indiquant le fils gauche et le fils droit.

```
def noeud(nom, fils_gauche, fils_droit):
    """crée un noeud pour constituer un arbre binaire"""
    return (nom, fils_gauche, fils_droit)
```

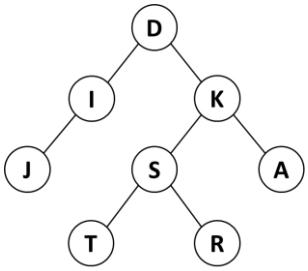
Le script suivant crée, à partir de la fonction **noeud** précédente, l'arbre ci-contre puis affiche son écriture :

```
#création de l'arbre
c = noeud('C', None, None)
f = noeud('F', None, None)
t = noeud('T', c, f)
#affichage de l'arbre à partir de la racine
print(t)
```




```
>>> ('T', ('C', None, None), ('f', None, None))
```

✍ Donner le script qui permet de créer l'arbre ci-contre à partir de la fonction **noeud**, puis l'implémenter afin de vérifier le résultat obtenu.

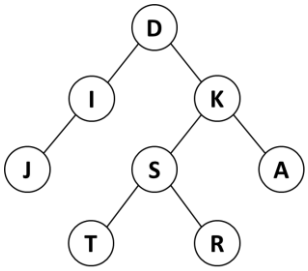


3.3. Feuille et nœud

✍ Implémenter une fonction **est_feuille** qui teste si le nœud est une feuille.

Pour l'arbre ci-contre :

```
>>> est_feuille(d)
False
>>> est_feuille(j)
True
>>> est_feuille(i)
False
```



✍ Implémenter une fonction **est_noeud** qui si le nœud est un nœud et non une feuille.

3.4. Taille de l'arbre

Pour rappel, la **taille** de l'arbre est le **nombre de nœuds** qui le constitue.

✍ Donner la taille de l'arbre précédent.

L'idée est la suivante pour créer une fonction qui calcul la taille d'un arbre

- Si l'arbre est vide (None) alors on retourne 0
- Sinon on retourne 1 + appel récursif à cette fonction sur le fils gauche + appel récursif à cette fonction sur le fils droit.

✍ Implémenter la fonction récursive **taille** qui permet de retourner la taille d'un arbre.

3.5. Hauteur de l'arbre

Pour rappel, la **hauteur** d'un arbre est la **profondeur du nœud le plus profond**.

✍ Toujours pour le même arbre que précédemment, donner sa hauteur.

Pour l'écriture de la fonction hauteur, l'idée est la suivante :

- Si l'arbre est vide la hauteur vaut -1.
- Sinon la hauteur vaut 1 auquel il faut ajouter le maximum entre les hauteurs des sous arbres gauche et droit.
- Ces sous-arbres sont eux-mêmes des arbres dont il faut calculer la hauteur.

Une méthode récursive semble tout à fait adaptée à la situation.

Voici l'algorithme :

```
Fonction hauteur (arbre)
     Si arbre est vide alors
          renvoyer - 1
     Sinon
          h1 ← 1 + hauteur(arbre[fils_gauche])
          h2 ← 1 + hauteur(arbre[fils_droit])
          renvoyer max(h1, h2)
```

✍ Implémenter la fonction récursive **hauteur** qui permet de retourner la hauteur d'un arbre.

4. Implémentation des arbres binaires en paradigme objet

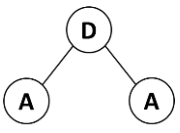
4.1. Implémentation de l'arbre

Soit la classe **noeud** qui crée un nœud en donnant un nom à ce nœud et en indiquant le fils gauche et le fils droit.

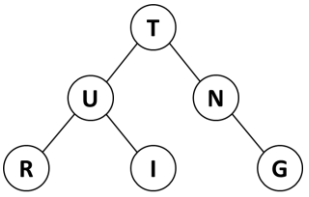
```
class Noeud:
    def __init__(self, nom, fils_gauche, fils_droit):
        self.nom = nom
        self.fg = fils_gauche
        self.fd = fils_droit
```

Le script suivant crée, à partir de la classe **noeud** précédente, l'arbre ci-contre :

```
#création de l'arbre
a1 = Noeud('A', None, None)
a2 = Noeud('A', None, None)
d = Noeud('D', a1, a2)
```



✍ Donner le script qui permet de créer l'arbre ci-contre à partir de la classe **noeud**.



4.2. Affichage de l'arbre

L'affichage de l'arbre peur se faire à l'aide de la fonction affiche ci-dessous :

```
def affiche(arbre):
    if arbre != None:
        return (arbre.get_nom(), affiche(arbre.get_gauche()), affiche(arbre.get_droit()))
```

Cette fonction fait appel aux méthodes :

- **`get_nom(self)`** : retourne le nom du nœud ;
- **`get_gauche(self)`** : retourne le fils gauche du nœud ;
- **`get_droit(self)`** : retourne le fils droit du nœud.

✍ Implémenter et rajouter ces **méthodes** à la classe **`Noeud`** et tester l'affichage de l'arbre précédent.

4.3. Feuille et nœud

✍ Implémenter une **méthode `est_feuille(self)`** qui teste si le nœud est une feuille.

✍ Implémenter une **méthode `est_noeud(self)`** qui teste si le nœud est un nœud et non une feuille.

4.4. Taille de l'arbre

✍ Implémenter une **fonction** qui donne la taille de l'arbre.

4.5. Hauteur de l'arbre

✍ Implémenter une **fonction** qui donne la hauteur de l'arbre.

Source : https://isn-icn-ljm.pagesperso-orange.fr/NSI-TLE/co/section_chapitre3.html