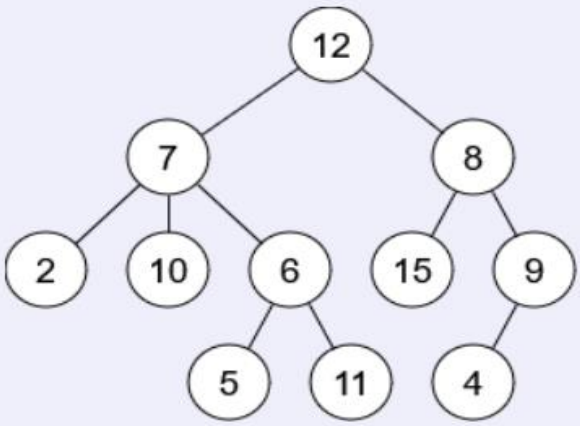
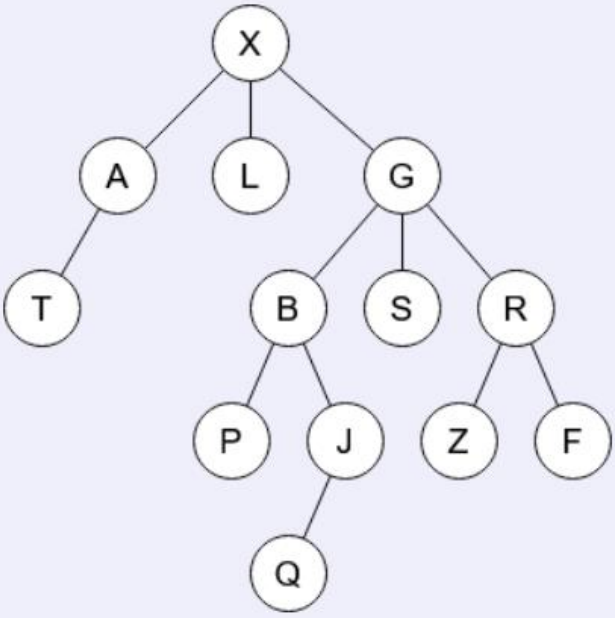
Exercices - Arbres binaires

Exercice 1.: Pour les 2 arbres ci-dessous, donner leur hauteur et taille, le nom des feuilles, le père de ces feuilles.

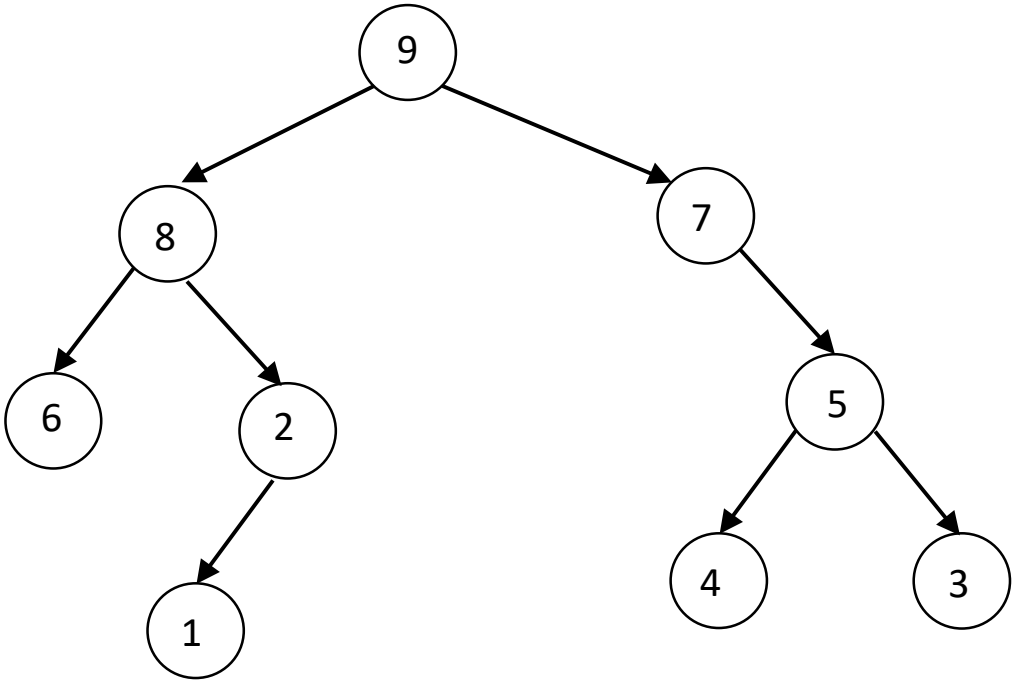
Soit l'arbre suivant :



Soit l'arbre suivant :



Exercice 2.: On donne l'arbre ci-dessous :



1- Donner la hauteur de cet arbre.

2- Pour un arbre de cette hauteur, combien de nœuds peut-on avoir au maximum ?

3- Donner l'ordre des nœuds dans un parcours en ordre (infixe) : fils gauche, nœud, fils droit

4- Donner l'ordre des nœuds dans un parcours en pré-ordre (préfixe) : nœud, fils gauche, fils droit

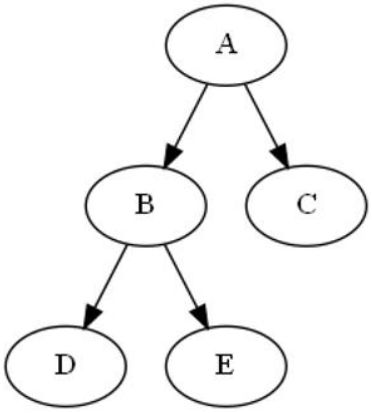
5- Donner l'ordre des nœuds dans un parcours en post-ordre (postfixe) : fils gauche, fils droit, nœud

6- Donner l'ordre des nœuds dans un parcours en largeur.

Exercice 3.: Implémentation d'une classe arbre en python

Le code ci-contre comprend une classe File et le script incomplet d'une classe Arbre.

La partie programme principale permet d'implémenter l'arbre donné ci-dessous :

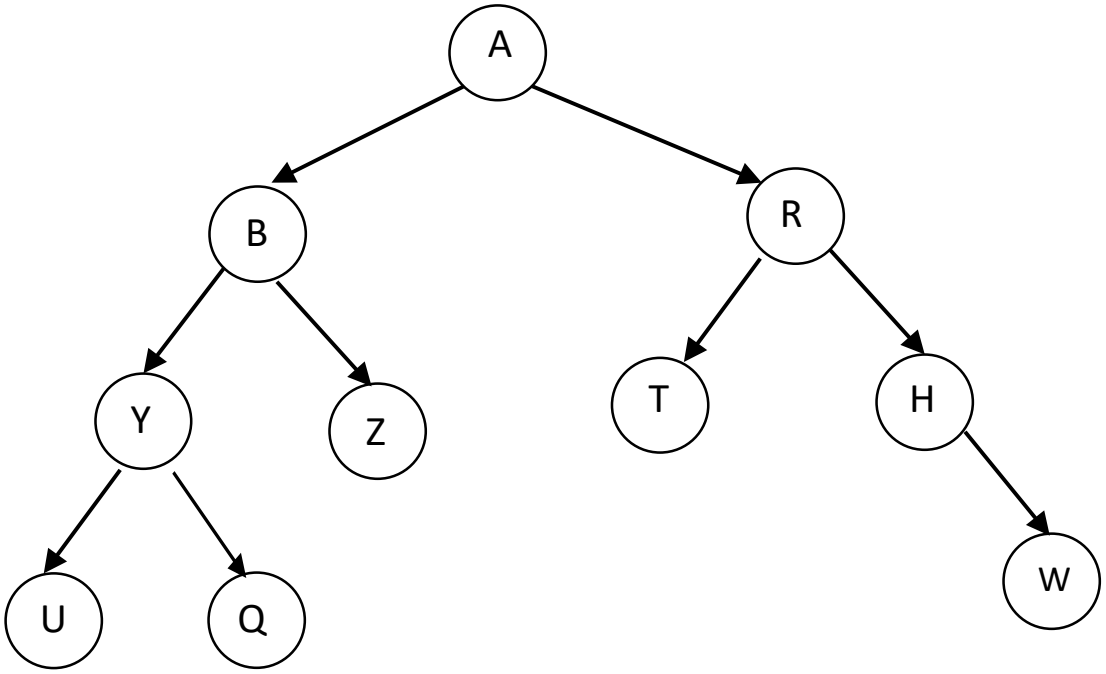


1- Télécharger ce code sur *nsibranly.fr* , le copier dans votre répertoire de travail et le tester.

```
from graphviz import Digraph
graphe = Digraph(format = 'png' , filename = 'arbre')

class File :
    def __init__(self) :
        self.l = []

    def __str__(self) :
        s = "file : "
        for e in self.l : s += str(e) + " "
        return s

    def estVide(self) :
        return self.l == []

    def enfiler(self,e) :
        self.l.append(e)

    def defiler(self) :
        if not self.estVide() :
            return self.l.pop(0)
```

```
class Arbre :
    def __init__(self, info = None , fg = None , fd = None) :
        self.info = str(info)
        self.fg = fg
        self.fd = fd

    def view(self) :
        graphe.clear()
        l = [self]
        n = 0
        while l != [] :
            n = n + 1
            nd = l.pop(0)
            nd.num = str(n)
            graphe.node(nd.num , nd.info)
            if nd.fg != None :
                l.append(nd.fg)
            if nd.fd != None :
                l.append(nd.fd)
        l = [self]
        while l != [] :
            nd = l.pop(0)
            if nd.fg != None :
                graphe.edge(nd.num,nd.fg.num)
                l.append(nd.fg)
            if nd.fd != None :
                graphe.edge(nd.num,nd.fd.num)
                l.append(nd.fd)
        graphe.view()


# Programme principal
A = Arbre('A',
          Arbre('B',
                Arbre('D') ,
                Arbre('E')
                ) ,
          Arbre('C')
          )
A.view()
```

2- Modifier le programme principal pour implémenter l'arbre donné ci-dessous :



3- Donner l'ordre des nœuds dans un parcours en ordre (infixe) : fils gauche, nœud, fils droit.

4- Dans le fichier *nsibranly.fr,* compléter la classe Arbre avec la méthode *parcoursRecursif()* . Retrouver le résultat précédent en exécutant `>>> A.parcoursRecursif()`

```
def parcoursRecursif(self) :
    if self.fg != None : self.fg.parcoursRecursif()
    print(self.info , end=" ")
    if self.fd != None : self.fd.parcoursRecursif()
```

```
>>> A.parcoursRecursif()
U Y Q B Z A T R H W
```

5- Donner l'ordre des nœuds dans un parcours pré-ordre (préfixe) : nœud, fils gauche, fils droit . Corriger la méthode *parcoursRecursif()* pour retrouver le résultat précédent en exécutant `A.parcoursRecursif()`

```
def parcoursRecursif(self) :
    print(self.info , end=" ")
    if self.fg != None : self.fg.parcoursRecursif()
    if self.fd != None : self.fd.parcoursRecursif()
```

```
>>> A.parcoursRecursif()
A B Y U Q Z R T H W
```

6- Donner l'ordre des nœuds dans un parcours en post-ordre (postfixe) : fils gauche, fils droit, nœud. Corriger la méthode *parcoursRecursif()* pour retrouver le résultat précédent en exécutant `A.parcoursRecursif()`

```
def parcoursRecursif(self) :
    if self.fg != None : self.fg.parcoursRecursif()
    if self.fd != None : self.fd.parcoursRecursif()
    print(self.info , end=" ")
```

```
>>> A.parcoursRecursif()
U Q Y Z B T W H R A
```

7- Donner l'ordre des nœuds dans un parcours en largeur. Compléter la classe Arbre avec la méthode *parcoursLargeur()* . Retrouver le résultat précédent en exécutant `>>> A.parcoursLargeur()`

```
def parcoursLargeur(self) :
    f = File()
    f.enfiler(self)
    while not f.estVide() :
        nd = f.defiler()
        print(nd.info , end=" ")
        if nd.fg != None : f.enfiler(nd.fg)
        if nd.fd != None : f.enfiler(nd.fd)
```

```
>>> A.parcoursLargeur()
A B R Y Z T H U Q W
```

8- Quelle est la taille de l'arbre ? Compléter la classe Arbre avec la méthode *taille()* . Renvoie-t-elle le même résultat ?

Il y a 10 nœuds dans cet arbre. La taille est donc $n = 10$

```
>>> A.taille()
10
```

Version simple à comprendre :

```
def taille(self) :
    if self.fg == None and self.fd == None :
        return 1
    if self.fg == None and self.fd != None :
        return 1 + self.fd.taille()
    if self.fg != None and self.fd == None :
        return 1 + self.fg.taille()
    if self.fg != None and self.fd != None :
        return 1 + self.fg.taille() + self.fd.taille()
```

Version plus épurée :

```
def taille(self) :
    if self.fg == None : nbG = 0
    else : nbG = self.fg.taille()
    if self.fd == None : nbD = 0
    else : nbD = self.fd.taille()
    return 1 + nbG + nbD
```

9- Quelle est la hauteur de l'arbre ? Compléter la classe Arbre avec la méthode *hauteur()* . Renvoie-t-elle le bon résultat ?

La hauteur est $h = 3$

```
>>> A.hauteur()
3
```

```
def hauteur(self) :
    if self.fg == None : hG = -1
    else : hG = self.fg.hauteur()
    if self.fd == None : hD = -1
    else : hD = self.fd.hauteur()
    return 1 + max(hG , hD)
```

10- Quel est le nombre de feuilles de l'arbre précédent ? Ecrire une méthode nommée *nb_feuilles()* qui renvoie ce nombre.

```
def nb_feuilles(self) :
    if self.fg == None and self.fd == None :
        return  1
    elif self.fg == None :
        return self.fd.nb_feuilles()
    elif self.fd == None :
        return self.fg.nb_feuilles()
    else :
        return self.fg.nb_feuilles() + self.fd.nb_feuilles()
```

```
>>> A.nb_feuilles()
5
```

11- Ecrire une méthode nommée *est_vide()* qui renvoie *True* si l'arbre n'a pas de fils et *False* dans le cas contraire.

```
def est_vide(self) :
    return self.fg == None and self.fd == None
```

```
>>> A.est_vide()
False

>>> Arbre('T').est_vide()
True
```

12- Ecrire une méthode nommée *trouve()* qui prend en argument une valeur. Si cette valeur est dans l'arbre, la méthode renvoie *True.* Dans le cas contraire, elle revoie *False*. Par exemple, l'exécution de `A.trouve('Y')` renvoie True.

Version récursive :

```
def trouve(self,val) :
    if self.fg == None and self.fd == None :
        return  self.info == val
    elif self.fg == None :
        return self.info == val or self.fd.trouve(val)
    elif self.fd == None :
        return self.info == val or self.fg.trouve(val)
    else :
        return self.info == val or self.fg.trouve(val) or self.fd.trouve(val)
```

Version itérative :

```
def trouve(self,val) :
    f = File()
    f.enfiler(self)
    while not f.estVide() :
        nd = f.defiler()
        if nd.info == val : return True
        if nd.fg != None : f.enfiler(nd.fg)
        if nd.fd != None : f.enfiler(nd.fd)
    return False
```

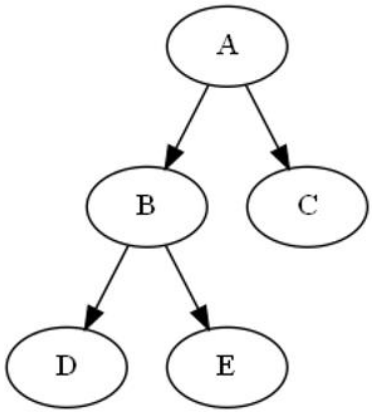
13- Ecrire une méthode nommée *nb()* qui prend en argument une valeur. Cette méthode renvoie le nombre de nœuds qui ont un attribut *info* égal à cette valeur. On aura par exemple l'exécution :

```
>>> A.nb('A')
1
```

Version récursive :

```
def nb(self,val) :
    if self.info == val : n = 1
    else : n = 0
    if self.fg == None and self.fd == None :
        return  n
    elif self.fg == None :
        return n + self.fd.trouve(val)
    elif self.fd == None :
        return n + self.fg.trouve(val)
    else :
        return n + self.fg.trouve(val) + self.fd.trouve(val)
```

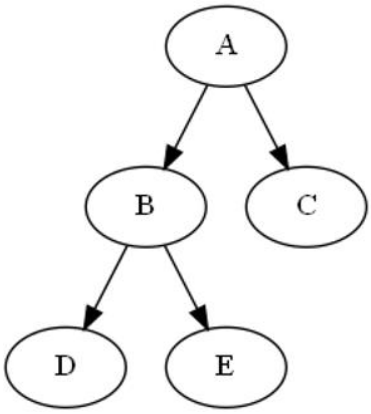
14- Ecrire une version **itérative** d'une méthode nommée *neud()* qui prend en argument une valeur. Cette méthode renvoie l'objet arbre qui a comme attribut info la valeur. On aura par exemple l'exécution suivante avec l'arbre donné ci-contre :




```
>>> A.noeud('B')
<__main__.Arbre object at 0x000001F03AD32910>

>>> A.noeud('B').info
'B'

>>> A.noeud('B').fg.info
'D'
```

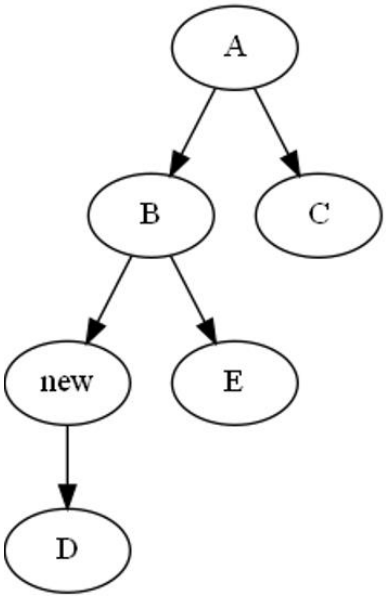
```
def noeud(self,val) :
    f = File()
    f.enfiler(self)
    while not f.estVide() :
        nd = f.defiler()
        if nd.info == val : return nd
        if nd.fg != None : f.enfiler(nd.fg)
        if nd.fd != None : f.enfiler(nd.fd)
    return None
```

15- Ecrire une méthode nommée *insere_gauche()* qui prend en argument 2 strings. Le premier correspond à l'attribut info d'un nœud existant de l'arbre. Le second à celui d'un nouveau nœud que l'on veut insérer à la gauche du nœud existant repéré.
Par exemple, avec l'arbre donné ci-contre, l'exécution de :




```
>>> A.insere_gauche('B','new')

>>> A.view()
```

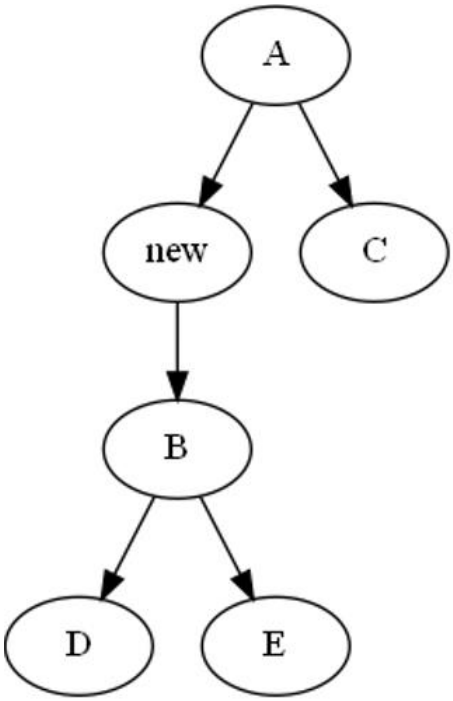
Permet d'obtenir l'arbre suivant :



L'exécution de :

```
>>> A.insere_gauche('A','new')

>>> A.view()
```

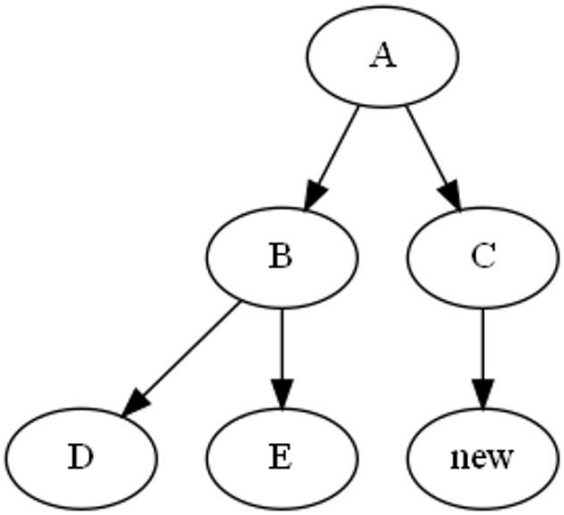
Donnerait :



L'exécution de :

```
>>> A.insere_gauche('C','new')

>>> A.view()
```

donnerait



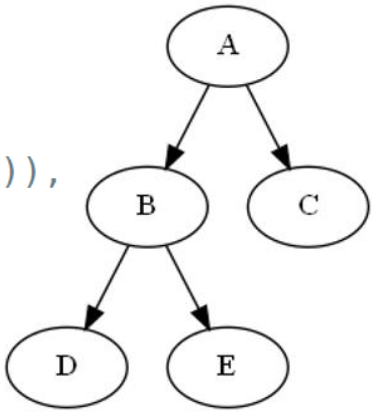
```
def insere_gauche(self,val,info) :
    nd = self.noeud(val)
    new = Arbre(info,nd.fg,None)
    nd.fg = new
```

16- Ecrire une méthode nommée *insere_droite()* qui réalise les mêmes opérations, mais en insérant à droite.

```
def insere_droite(self,val,info) :
    nd = self.noeud(val)
    new = Arbre(info,None,nd.fd)
    nd.fd = new
```

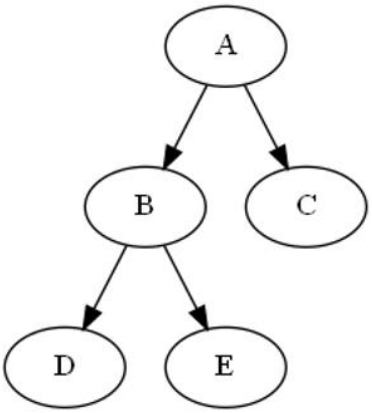
17- Ecrire la méthode __str__() qui permet d'obtenir l'affichage donné ci-dessous, pour l'arbre ci-contre :

```
>>> print(A)
Arbre(A,Arbre(B,Arbre(D,None,None),Arbre(E,None,None)),
Arbre(C,Arbre(new,None,None),None))
```




```
def __str__(self) :
    if self.fg == None : g = "None"
    else : g = self.fg.__str__()
    if self.fd == None : d = "None"
    else : d = self.fd.__str__()
    return(f"Arbre({self.info},{g},{d})")
```

18- Ecrire une version **récursive** d'une méthode nommée profondeur(). Elle permet d'obtenir l'affichage donné ci-dessous, pour l'arbre ci-contre :
Aide : il suffit de faire un parcours récursif postfixe et d'avoir en paramètre la variable p qui est la profondeur, initialisée à 0 au 1er appel :




```
def profondeur(self,p = 0) :
```

```
>>> A.profondeur()
noeud D de profondeur 2
noeud E de profondeur 2
noeud B de profondeur 1
noeud C de profondeur 1
noeud A de profondeur 0
```

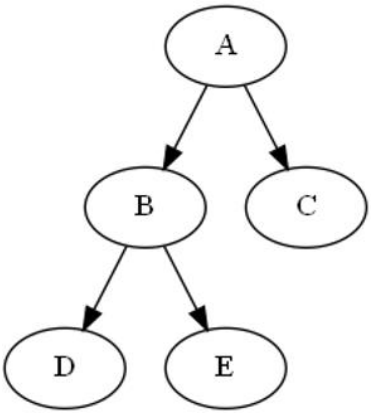
```python
def profondeur(self,p = 0) :
    if self.fg != None : self.fg.profondeur(p+1)
    if self.fd != None : self.fd.profondeur(p+1)
    print(f"noeud {self.info} de profondeur {p}")
```

19- Ecrire une version **récursive** de la méthode nommée *neud()* qui prend en argument une valeur. Cette méthode renvoie l'objet arbre qui a comme attribut info la valeur (comme dans la question 14).
Aide : il suffit de faire un parcours récursif préfixe, infixe ou postfixe et de mémoriser dans un attribut de classe, par exemple *Arbre.obj*, l'objet lorsqu'il est trouvé.

```python
class Arbre :
    obj = None
    def __init__(self, info = None , fg = None , fd = None) :
        self.info = str(info)
        self.fg = fg
        self.fd = fd

    def noeudRec(self,val) :
        if self.info == val : Arbre.obj = self
        if self.fg != None : self.fg.noeudRec(val)
        if self.fd != None : self.fd.noeudRec(val)
```

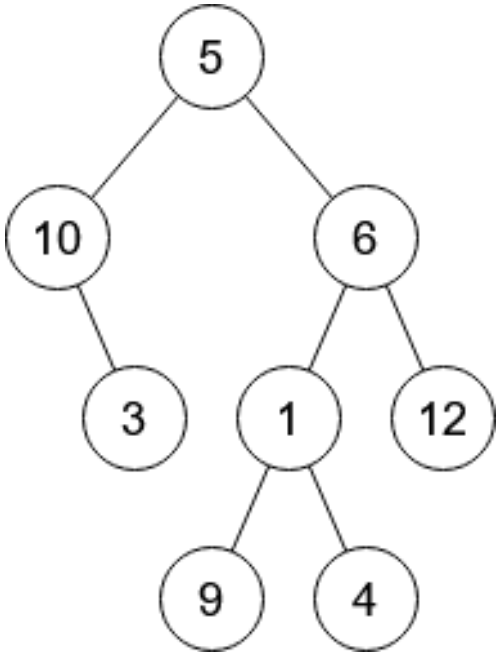
20- Ecrire une version **itérative** d'une méthode nommée profondeur(). Elle permet d'obtenir l'affichage donné ci-dessous, pour l'arbre ci-contre :

```
>>> A.profondeur()
noeud A de profondeur 0
noeud B de profondeur 1
noeud C de profondeur 1
noeud D de profondeur 2
noeud E de profondeur 2
```




```python
def profondeur(self) :
    profondeur = 0
    nbParents = 1
    nbEnfants = 0
    f = File()
    f.enfiler(self)
    while not f.estVide() :
        nd = f.defiler()
        if nbParents == 0  :
            nbParents = nbEnfants
            nbEnfants = 0
            profondeur +=1
        print(f"noeud {nd.info} de profondeur {profondeur}")
        if nd.fg != None :
            f.enfiler(nd.fg)
            nbEnfants += 1
        if nd.fd != None :
            f.enfiler(nd.fd)
            nbEnfants += 1
        nbParents -= 1
```

Exercice 4.: Pour l'arbre ci-contre :



1- Donner la taille, la hauteur, le nombre de feuilles.
2- Donner l'ordre de visite dans un parcours infixe
3- Donner l'ordre de visite dans un parcours préfixe
4- Donner l'ordre de visite dans un parcours postfixe
5- Donner l'ordre de visite dans un parcours en largeur