Exercice 1. : Le code python incomplet donné ci-contre, permet d'implémenter des ABR.

Le fichier *trace_arbre.py* qui contient les classes Graph, Pile et File est à télécharger sur le bloc ABR de *nsibranly.fr* . Il est contenu dans le fichier : *trace_arbre.zip* .

Dans cet exercice, vous nommerez le fichier contenant le code à compléter : *tpAbrIt.py*

```
from trace_arbre import Graph , Pile , File
from random import randint

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

    def insEltIte(self,val):
        '''
            Insère dans l'arbre, version itérative
        '''
        nd = self
        while True :
            if val == nd.info : return False
            elif val < nd.info :
                if nd.fg == None :
                    nd.fg = Arbre(val)
                    return True
                else :
```

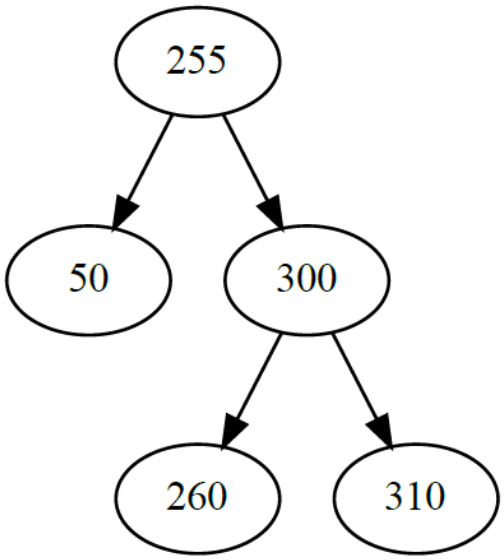
1- ***Insertion d'un nœud*** :

L'exécution des lignes suivantes :

```
# Main -----------------
A = Arbre(255)
print(A.insEltIte(50))
print(A.insEltIte(300))
print(A.insEltIte(310))
print(A.insEltIte(260))
print(A.insEltIte(260))
A.trace_graphique()
```

Donne le résultat ci-dessous dans la console et ci-contre avec Graphviz :

```
>>> (executing file "tpAbrIt.py")
True
True
True
True
False
```

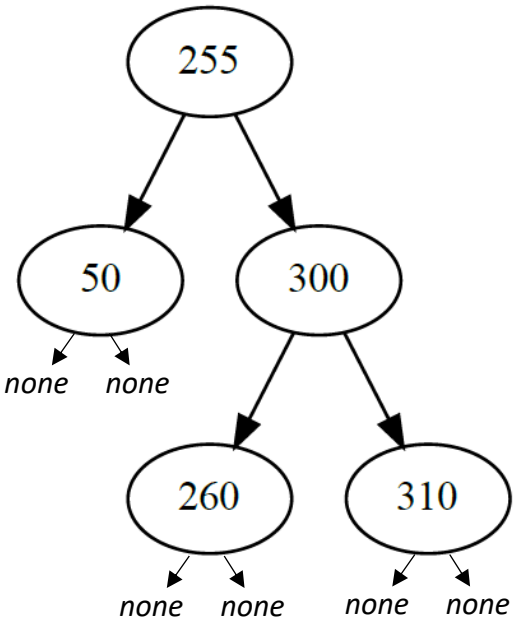


⇨ Compléter le code itératif de la méthode *insEltIte()* .

On exécute à présent la ligne : `A.insEltIte(270)`

⇨ Compléter ci-dessous le tableau qui donne la valeur des variables utilisées dans la méthode *insEltIte()* :

val	nd.info	nd.fg	nd.fd
270	255	50	300



2- ***Recherche d'une valeur*** :

Soit la méthode incomplète ci-contre :

```
def rechEltIte(self,val):
    '''
        Recherche dans l'arbre, version itérative
    '''
    nd = self
    while True :
        if val == nd.info :
            return True
        elif val < nd.info :
```

L'exécution des lignes suivantes :

```
# Main ------------------
A = Arbre(255)
A.insEltIte(50)
A.insEltIte(300)
A.insEltIte(310)
A.insEltIte(260)

print(A.rechEltIte(260))
print(A.rechEltIte(2023))
```

donne le résultat ci-dessous dans la console :

```
>>> (executing file "tpAbrIt.py")
True
False
```

⇨ Compléter le code itératifde la méthode *rechEltIte()*.

3- ***Recherche du maximum contenue dans un ABR*** :

Soit la méthode incomplète ci-contre :

```
def rechMaxIte(self):
    '''
        Recherche max, version itérative
    '''
    nd = self
    while True :
```

L'exécution des lignes suivantes :

```
# Main ----------------
A = Arbre(255)
A.insEltIte(50)
A.insEltIte(300)
A.insEltIte(310)
A.insEltIte(260)

print(A.rechMaxIte())
```

donne le résultat ci-dessous dans la console :

```
>>> (executing file "tpAbrIt.py")
310
```

⇨ Compléter le code itératif de la méthode *rechMaxIte()*.

Exercice 2. : On répond ici aux mêmes questions que celles de l'exercice précédent. Par contre ici les **codes sont de type récursifs**. Le fichier contenant le code sera nommé ici *tpAbrRec.py*

```
from trace_arbre import Graph , Pile , File
from random import randint

class Arbre(Graph) :

    def __init__(self, info , fg = None , fd = None) :
        self.info = info
        self.fg = fg
        self.fd = fd

    def insEltRec(self,val)-> bool:
        '''
            Insère dans l'arbre, version récursive
        '''
        if val == self.info : return False
        elif val < self.info :
            if self.fg == None :
                self.fg = Arbre(val)
                return True
```

1- ***Insertion d'un nœud*** :

L'exécution des lignes suivantes :

donne le résultat ci-dessous dans la console :

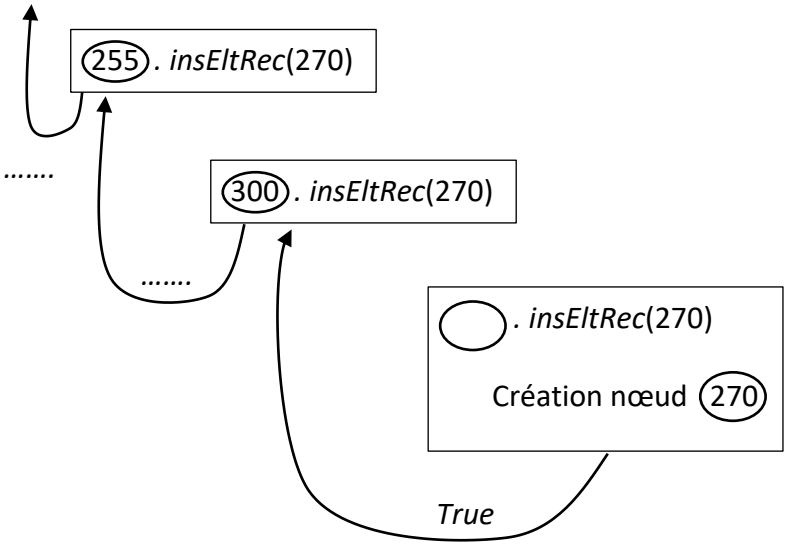
```
# Main ----------------
A = Arbre(255)
print(A.insEltRec(50))
print(A.insEltRec(300))
print(A.insEltRec(310))
print(A.insEltRec(260))

A.trace_graphique()
```

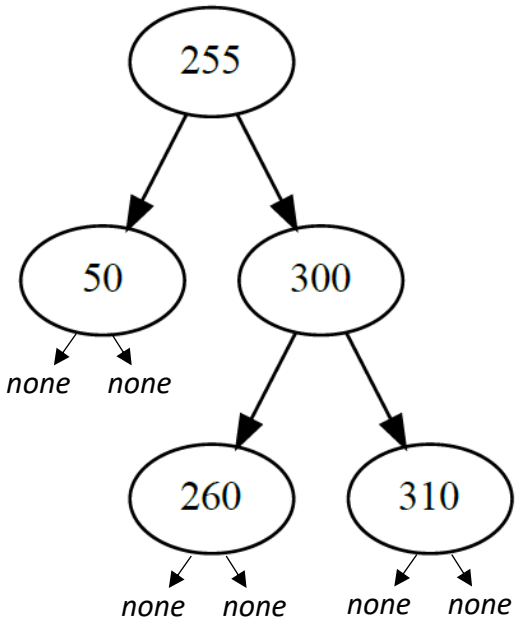
```
>>> (executing file "tpAbrRec.py")
True
True
True
True
```

⇨ Compléter le code récursif de la méthode *insEltRec()*.

On exécute à présent la ligne : `A.insEltRec(270)`

⇨ Compléter ci-dessous le diagramme ci-dessous qui précise les appels de la méthode insEltRec() :





2- ***Recherche d'une valeur*** :

⇨ Soit la méthode incomplète ci-contre :

```
def rechEltRec(self,val)-> bool:
    '''
        Recherche dans l'arbre, version récursive
    '''
    if val == self.info : return True
    elif val < self.info :
```

L'exécution des lignes suivantes :

```
# Main --------------------
A = Arbre(255)
A.insEltRec(50)
A.insEltRec(300)
A.insEltRec(310)
A.insEltRec(260)
A.insEltRec(270)

print(A.rechEltRec(260))
print(A.rechEltRec(2023))
```

donne le résultat ci-dessous dans la console :

```
>>> (executing file "tpAbrRec.py")
True
False
```

⇨ Compléter le code de la méthode *rechEltRec()*.

3- ***Recherche du maximum contenue dans un ABR*** :

Soit la méthode incomplète

```
def rechMaxRec(self):
    '''
        Recherche max, version récursive
    '''
```

L'exécution des lignes suivantes :

```
# Main ----------------
A = Arbre(255)
A.insEltRec(50)
A.insEltRec(300)
A.insEltRec(310)
A.insEltRec(260)
A.insEltRec(270)

print(A.rechMaxRec())
```

donne le résultat ci-dessous dans la console :

```
>>> (executing file "tpAbrRec.py")
310
```

⇨ Compléter le code de la méthode *rechMaxRec()*.

Exercice 3. : On utilise à présent le code précédent pour stocker 16154 mots de la langue française dans un ABR.

⇨ Télécharger sur le bloc ABR de *nsibranly.fr* le fichier *tdABR.zip* . Ce dossier compressé contient les fichiers *dictionnaire.txt* dans lequel se trouve la plupart des mots de la langue française et *dictionnaire.py* . Décompresser ces fichiers dans le répertoire dans lequel se trouve déjà les fichiers *tpAbrIt.py , tpAbrRec.py* et *trace_arbre.py* mis au point dans l'exercice précédent.

⇨ Ouvrir le fichier *dictionnaire.py*. Le code qui s'y trouve est donné ci-dessous :

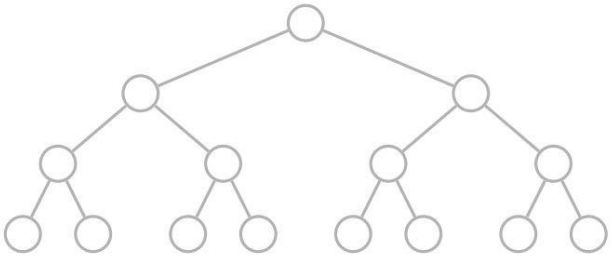
```
from tpAbrRec import Arbre
from random import randint

def accents(mot) :
    for c in mot :
        if c in "éçùêèà" : return True
    return False

def lectureFichier() :
    l = []
    f = open("dictionnaire.txt","r",encoding='utf8')
    mot = f.readline()
    while mot != "" :
        mot = mot[:-1]
        mot = mot.lower()
        if not accents(mot) : l.append(mot)
        mot = f.readline()
    f.close()
    return l

def remplissageABR(N = "all") :
    if N == "all" : N = len(liste)
    mot = liste[randint(0,len(liste)-1)]
    dic = Arbre(mot)
    for i in range(N) :
        mot = liste[randint(0,len(liste)-1)]
        dic.insEltRec(mot)
    if N <= 500 : dic.trace_graphique()
    return dic

def rechercheListe(liste,mot) :
    i = 0
    while liste[i] != mot and i < len(liste):
        i += 1
    if i == len(liste) : return False
    else : return True , i

#Main
liste = lectureFichier()
dic = remplissageABR(10)
```

⇨ Analyser ce code et l'exécuter en tant que fichier principal (click droit sur l'onglet du fichier et

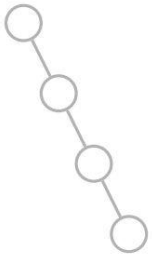
et ⭐ (Dé)sélectionner comme fichier principal (MAIN) + Exécuter / Démarrer le script principal)

1- Dans la console, afficher le contenu de la liste (*print(liste)*) .

2- En exécutant plusieurs fois ce fichier *dictionnaire.py* qui crée pour l'instant un arbre de 10 mots, peut-on affirmer que sa forme est plus proche de celle d'un arbre complet ou de celle d'un arbre peigne ? (**répondre sur document réponse**)



Arbre complet



Peigne

3- Réaliser plusieurs exécutions en stockant à présent 500 mots dans l'ABR. L'arbre obtenu est-il la plupart du temps proche d'un arbre complet ?

```
#Main
liste = lectureFichier()
dic = remplissageABR(500)
```

```
#Main
liste = lectureFichier()
dic = remplissageABR("all")
print(dic.rechEltRec("zone"))
print(dic.rechEltRec("nsi"))
```

4- Réaliser à présent, une exécution en stockant la totalité des 16154 mots contenus dans la liste et rechercher la présence des mots « zone » et « nsi ».

5- Dans le fichier *tpAbrRec.py*, ajouter une variable de classe nommée nb qui compte le nombre d'appels de la méthode récursive rechEltRec() :

```
class Arbre(Graph) :
    nb = 0
    def __init__(self, info , fg = None , fd = None) :
        self.info = info
        self.fg = fg
        self.fd = fd
    def rechEltRec(self,val)-> bool:
        '''
            Recherche dans l'arbre, version récursive
        '''
        __class__.nb += 1
        if val == self.info : return True
        elif val < self.info :
            if self.fg == None : return False
            else : return self.fg.rechEltRec(val)
        elif val > self.info :
            if self.fd == None : return False
            else : return self.fd.rechEltRec(val)
```

```
def raz(self) :
    __class__.nb = 0
```

Et dans le fichier *dictionnaire.py* , exécuter les lignes suivantes :

```
#Main
liste = lectureFichier()
dic = remplissageABR("all")
dic.raz()
print(dic.rechEltRec("nsi"),dic.nb)
dic.raz()
print(dic.rechEltRec("zone"),dic.nb)
```

⇨ Combien d'appels récursifs a-t-on en moyenne pour retrouver le mot « *zone* » et combien pour retrouver le mot « *nsi* » ?

6- Utiliser la fonction *rechercheListe()* pour comparer les nombres d'appels précédents avec le nombre d'itérations en effectuant une recherche naïve :

```
#Main
liste = lectureFichier()
dic = remplissageABR("all")
dic.raz()
print(dic.rechEltRec("nsi"),dic.nb)
dic.raz()
print(dic.rechEltRec("zone"),dic.nb)
print(rechercheListe(liste,"zone"))
print(rechercheListe(liste,"nsi"))
```

7- Si le remplissage de l'arbre conduisait à un arbre complet, la hauteur de l'arbre serai' égale à $log_2(16154)$. Utiliser la calculatrice pour calculer $log_2(16154) = \frac{\ln(16154)}{\ln(2)}$.

8- Un algorithme de recherche naïf dans une liste a une complexité en $O(n)$. D'après-vous quelle est la complexité du code de recherche dans un ABR ? Connaissez-vous une technique algorithmique qui permet sur un liste, d'avoir la même complexité ?

Exercice 4. : L'objectif de cet exercice est d'étoffer la classe *Arbre* en écrivant une méthode qui puisse vérifier si les données stockées dans l'arbre respectent bien le principe de stockage des ABR (fils gauche < fils droit). On travaille ici dans le fichier *tpAbrRec.py* mis au point précédemment. La méthode la plus simple consiste à copier tous les éléments de l'arbre dans une liste, en réalisant **un parcours infixe**. Si l'arbre respecte bien le principe des ABR, les valeurs de cette liste sont rangées dans l'ordre croissant.

1- Compléter le code de la méthode *parcoursInfixe()* ci-contre :

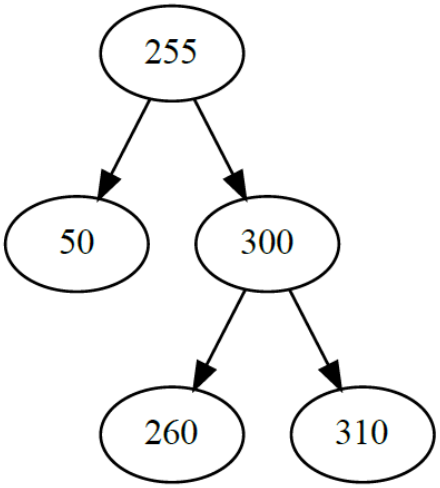
```
def parcoursInfixe(self) :
    '''
        En utilisant un parcours INFIXE, ajoute dans
        la liste __class__.liste les valeurs des noeuds
        dans l'ordre croissant
    '''
```

L'exécution des lignes suivantes :

```
#Main
A = Arbre(255)
A.insEltRec(50)
A.insEltRec(300)
A.insEltRec(310)
A.insEltRec(260)

A.parcoursInfixe()
print(A.liste)
A.trace_graphique()
```

donne dans la console et par Graphviz :

```
>>> (executing file "tpAbrRec.py")
[50, 255, 260, 270, 300, 310]
```



2- Compléter le code de la méthode *estABR()* ci-contre :

```
def estABR(self) :
    self.parcoursInfixe()
    l = __class__.liste
    print(l)
    for i in range(1,len(l)) :
```

L'exécution des lignes suivantes :

```
#Main
A = Arbre(255)
A.insEltRec(50)
A.insEltRec(300)
A.insEltRec(310)
A.insEltRec(260)

print(A.estABR())
```

Donnera dans la console :

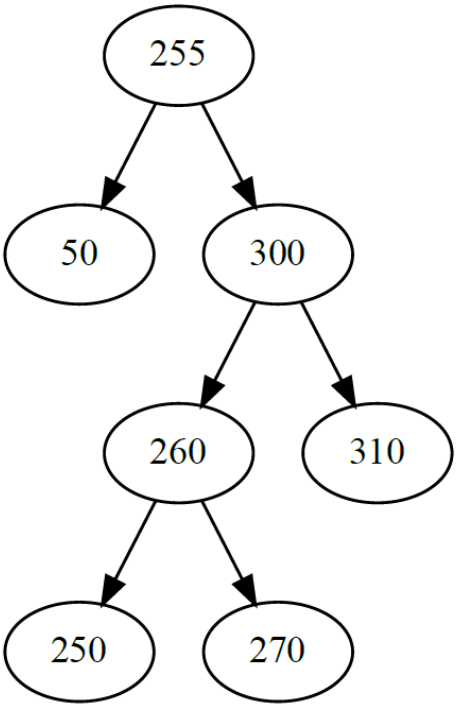
```
>>> (executing file "tpAbrRec.py")
[50, 255, 260, 300, 310]
True
```

Tandis que l'exécution de ces lignes :

```
#Main
A = Arbre(255)
A.insEltRec(50)
A.insEltRec(300)
A.insEltRec(310)
A.insEltRec(260)
A.insEltRec(270)
A.fd.fg.fg = Arbre(250)
print(A.estABR())

A.trace_graphique()
```

Donnera dans la console par Graphviz :

```
>>> (executing file "tpAbrRec.py")
[50, 255, 250, 260, 270, 300, 310]
False
```



Exercice 5. : On propose enfin une dernière méthode qui permet de vérifier si un arbre respecte bien le principe de rangement des ABR. On utilise ici entièrement une méthode récursive.

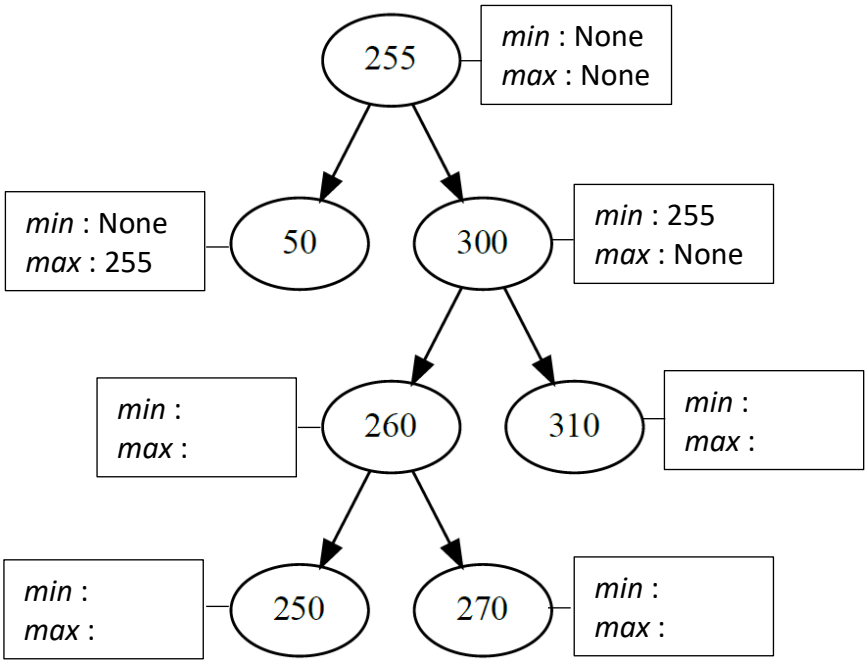
On ajoute dans le constructeur de la classe Arbre les attributs min et max qui indiqueront les valeurs minimale et maximale que peuvent prendre un nœud pour respecter le principe de rangement des ABR.

```
def __init__(self, info , fg = None , fd = None) :
    self.info = info
    self.fg = fg
    self.fd = fd
    self.min = None
    self.max = None
```

La méthode estAbrRec() retourne *True* si l'arbre est un ABR, elle retourne *False* sinon. Le code ci-dessous est à télécharger (fichier *estAbrRec.py* sur le bloc ABR de *nsibranly.fr*)

```
def estAbrRec(self):
    '''
        Vérifie si l'arbre est ABR
    '''
    if self.fg != None :
        self.fg.max = self.info
        self.fg.min = self.min
        if self.fg.max != None and self.fg.info > self.fg.max : return False
        if self.fg.min != None and self.fg.info < self.fg.min : return False
        if not self.fg.estAbrRec() : return False
    if self.fd != None :
        self.fd.min = self.info
        self.fd.max = self.max
        if self.fd.min != None and self.fd.info < self.fd.min : return False
        if self.fd.max != None and self.fd.info > self.fd.max : return False
        if not self.fd.estAbrRec()  : return False
    return True
```

⇨ Faire fonctionner cet algorithme à la main sur l'arbre ci-contre et écrire à coté de chacun des nœuds les valeurs des attributs *min* et *max* qui sont définies durant l'exécution de cette méthode.



Rendu du tp : Uploader les fichiers *tpAbrIt.py* , *tpAbrRec.py* et *dictionnaire.py* avec le code *tp15*

Nom :

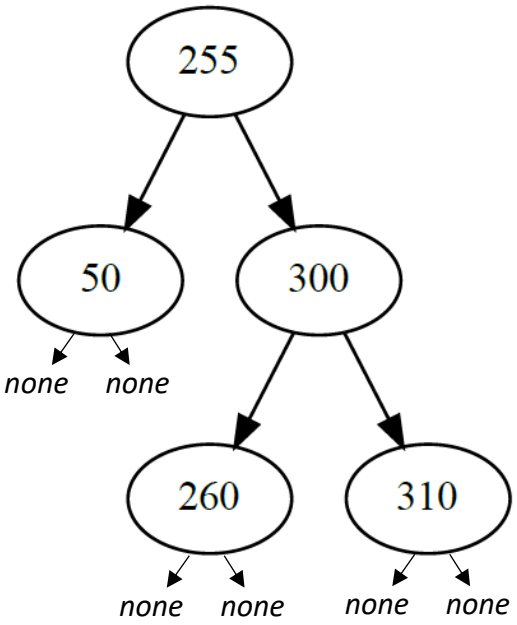
Dossier réponse :

Exercice 1 :

On exécute à présent la ligne : `A.insEltIte(270)`

⇨ Compléter ci-dessous le tableau qui donne la valeur des variables utilisées dans la méthode *insEltIte()* :

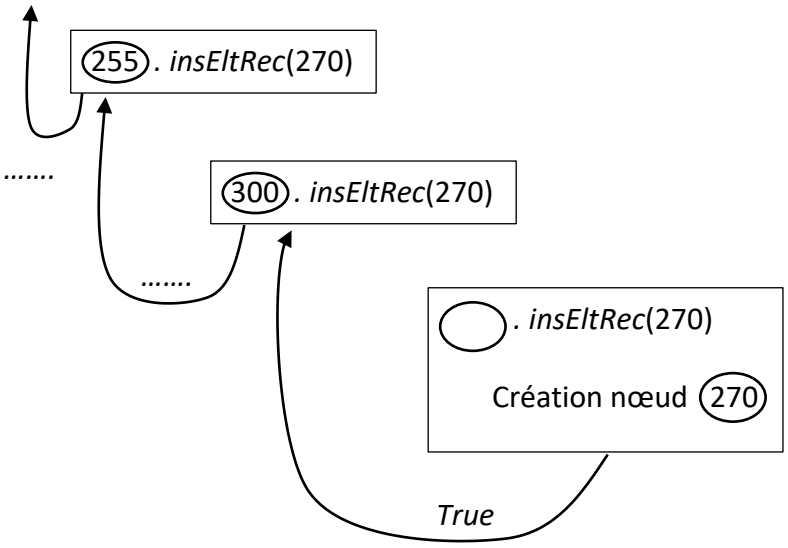
val	nd.info	nd.fg	nd.fd
270	255	50	300

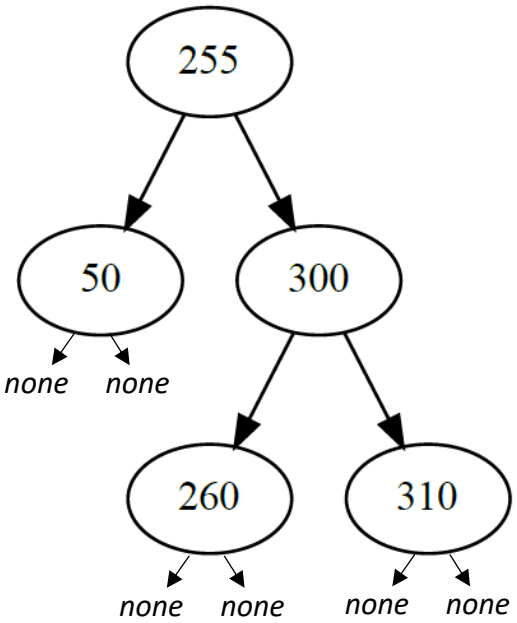


Exercice 2 :

On exécute à présent la ligne :

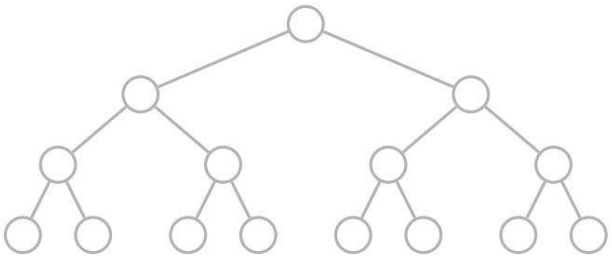
⇨ Compléter ci-dessous le diagramme ci-dessous qui précise les appels de la méthode insEltRec() :



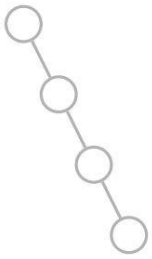


Exercice 4 :

2- En exécutant plusieurs fois ce fichier *dictionnaire.py* qui crée pour l'instant un arbre de 10 mots, peut-on affirmer que sa forme est plus proche de celle d'un arbre complet ou de celle d'un arbre peigne ?



Arbre complet



Peigne

3- Réaliser plusieurs exécutions en stockant à présent 500 mots dans l'ABR. L'arbre obtenu est-il la plupart du temps proche d'un arbre complet ?

```
#Main
liste = lectureFichier()
dic = remplissageABR(500)
```

⇨ Combien d'appels récursifs a-t-on en moyenne pour retrouver le mot « *zone* » et combien pour retrouver le mot « *nsi* » ?

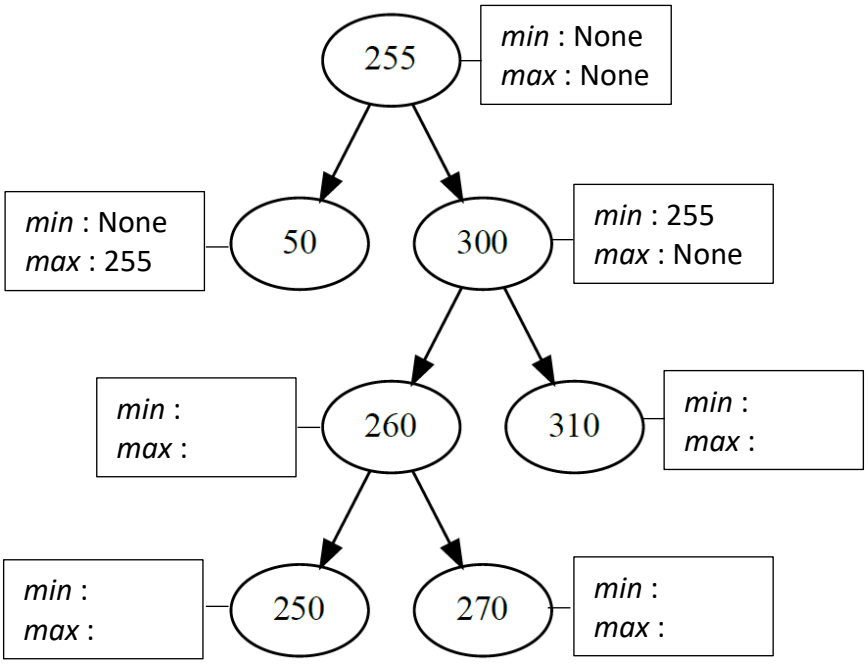
6- Utiliser la fonction *rechercheListe()* pour comparer les nombres d'appels précédents avec le nombre d'itérations en effectuant une recherche naïve :

```
#Main
liste = lectureFichier()
dic = remplissageABR("all")
dic.raz()
print(dic.rechEltRec("nsi"),dic.nb)
dic.raz()
print(dic.rechEltRec("zone"),dic.nb)
print(rechercheListe(liste,"zone"))
print(rechercheListe(liste,"nsi"))
```

7- Si le remplissage de l'arbre conduisait à un arbre complet, la hauteur de l'arbre serai' égale à $log_2(16154)$. Utiliser la calculatrice pour calculer $log_2(16154) = \frac{\ln(16154)}{\ln(2)}$.

8- Un algorithme de recherche naïf dans une liste a une complexité en $O(n)$. D'après-vous quelle est la complexité du code de recherche dans un ABR ? Connaissez-vous une technique algorithmique qui permet sur un liste, d'avoir la même complexité ?

Exercice 5 :


255
min : None
max : None
50
min : None
max : 255
300
min : 255
max : None
260
min :
max :
310
min :
max :
250
min :
max :
270
min :
max :