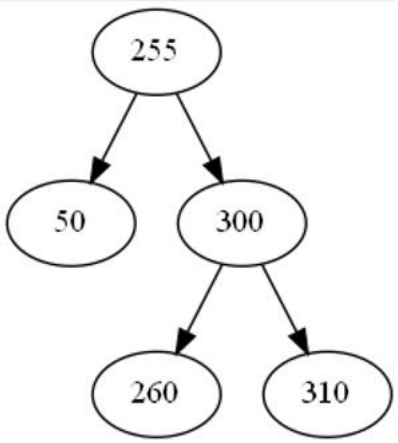
Tp - Arbres binaires de RECHERCHE

1- CREATION DE LA CLASSE ABR EN RECURSIF :



⇨ Ouvrir un nouveau fichier nommé *abrRecursif.py*

⇨ Y copier-coller le code du fichier abr.py disponible sur nsibranly.fr

⇨ Ce code contient une classe *File* et une classe *Abr* incomplète. Cette dernière contient un attribut de classe *liste*, la méthode constructeur *__init__()* et la méthode *view()* qui permet de visualiser l'arbre avec le logiciel *graphviz*. Une instance de cette classe est créée dans la partie programme principale et permet d'obtenir l'arbre donnée ci-contre :

```
if __name__ == '__main__' :
    A = Abr(255,
            Abr(50),
            Abr(300,
                    Abr(260) ,
                    Abr(310)
                    ),
            )
```

a. CREATION DE LA METHODE MIN() :

⇨ Ecrire le code **récursif** de la méthode *min()* qui retourne la valeur minimale de l'arbre.

```
>>> A.min()
50
```

```
def min(self)-> int:
    if self.fg == None : return self.info
    else : return self.fg.min()
```

```
assert A.min() == 50 , "La méthode min() ne donne pas le bon résultat"
```

b. CREATION DE LA METHODE RECHERCHE() :

⇨ Ecrire le code **récursif** de la méthode *recherche()* qui prend en argument une valeur et retourne *True* si la valeur existe, *False* sinon.

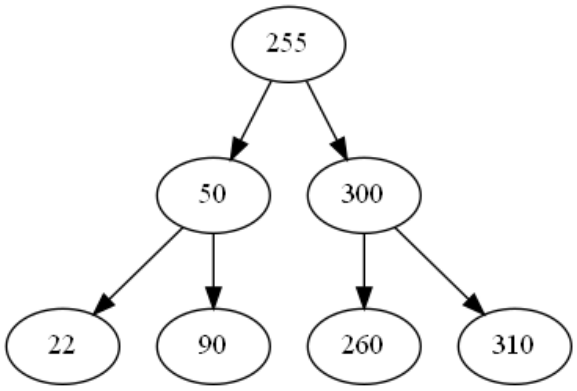
```
>>> A.recherche(260)
True

>>> A.recherche(261)
False
```

```
def recherche(self,val)-> bool:
    Abr.nb = Abr.nb + 1
    if val == self.info : return True
    elif val < self.info :
        if self.fg != None : return self.fg.recherche(val)
        else : return False
    elif val > self.info :
        if self.fd != None : return self.fd.recherche(val)
        else : return False
```

```
assert A.recherche(260) , "La méthode recherche() ne donne pas le bon résultat"
assert not A.recherche(261) , "La méthode recherche() ne donne pas le bon résultat"
```

c. CREATION DE LA METHODE INSERE() :

⇨ Ecrire le code **récursif** de la méthode *insere()* qui prend en argument une valeur et insère dans l'arbre un nouveau nœud dont l'attribut *info* est égal à cette valeur. Si la valeur est déjà présente dans l'arbre, la méthode retourne *False.* Elle retourne *True* dans le cas contraire.




```
>>> A.insere(50)
False

>>> A.insere(90)
True

>>> A.insere(22)
True

>>> A.view()
```

```
for _ in range(50) :
    n = randint(0,500)
    A.insere(n)
```

⇨ Tester la méthode avec l'exécution donnée ci-contre. Vérifier visuellement que l'arbre obtenu est bien un ABR. Essayer avec une insertion plus volumineuse ….

Vérifier qu'avec votre code, vous pouvez insérer dans un arbre initialement vide. Faire en sorte que l'exécution suivante fonctionne :

```
>>> B = Abr()

>>> B.insere(2025)
True

>>> B.view()
```

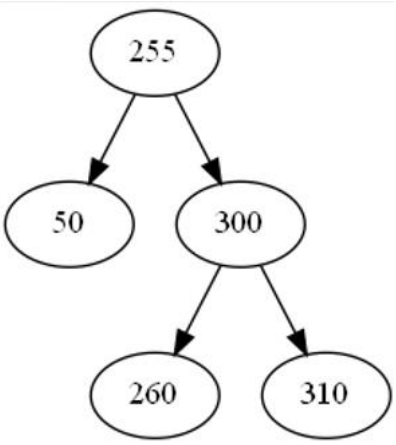
```python
def insere(self,val)-> bool:
    if self.info == None :
        self.info = val
        return True
    if val == self.info : return False
    elif val < self.info :
        if self.fg != None : return self.fg.insere(val)
        else :
            self.fg = Abr(val)
            return True
    elif val > self.info :
        if self.fd != None : return self.fd.insere(val)
        else :
            self.fd = Abr(val)
            return True
```

```python
assert not A.insere(50) , "La méthode recherche() ne donne pas le bon résultat"
assert A.insere(90) , "La méthode recherche() ne donne pas le bon résultat"
```

d. CREATION DE LA METHODE TRI() :

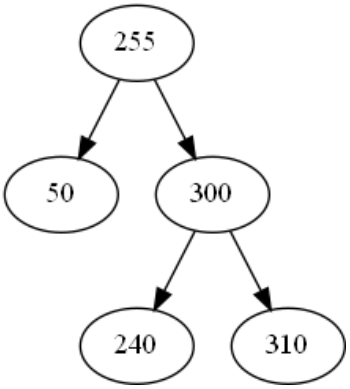
⇨ Ecrire le code **récursif** de la méthode *tri()* qui réalise un parcours infixe de l'arbre en ajoutant les attributs info des nœuds visités dans l'attribut de classe nommé liste.

```
>>> A.liste
[50, 255, 260, 300, 310]
```




```python
def tri(self) -> None :
    '''
        En utilisant un parcours INFIXE, ajoute dans
        la liste __class__.liste les valeurs des noeuds
        dans l'ordre croissant
    '''
    if self.fg != None : self.fg.tri()
    __class__.liste.append(self.info)
    if self.fd != None : self.fd.tri()
```

e. CREATION DE LA METHODE ESTABR() :



⇨ Ecrire le code **itératif** de la méthode *abr()* qui vérifie que les valeurs contenues dans l'attribut de classe nommé *liste* sont bien triées correctement. Cette méthode retourne True si l'arbre est bien un ABR et False dans le cas contraire :

```
>>> A.estABR()
False
```

```
def estABR(self)->bool :
    self.tri()
    l = __class__.liste
    for i in range(1,len(l)) :
        if l[i-1]>=l[i] : return False
    return True
```

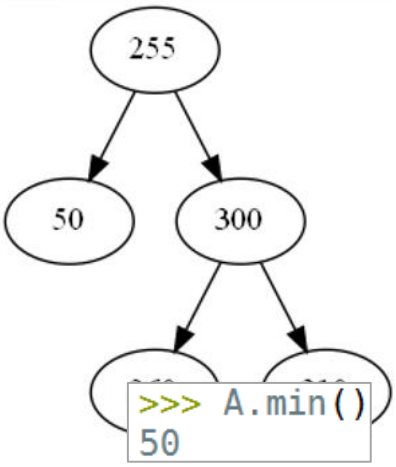
```
assert A.estABR() , "La méthode estABR() ne donne pas le bon résultat"
```

⇨ UPLOADER le fichier *abrRecursif.py* sur nsibranly.fr.

2- CREATION DE LA CLASSE ABR EN ITERATIF :

⇨ Ouvrir un nouveau fichier nommé abrIteratif.py

⇨ Y copier-coller le code du fichier *abr.py* disponible sur nsibranly.fr



a. CREATION DE LA METHODE MIN() :

⇨ Ecrire le code **itératif** de la méthode *min()* qui retourne la valeur minimale de l'arbre. Pour vous aider, on donne ci-dessous, un exemple incomplet de ce code :

```
def min(self)-> int:
    noeud = self
    while noeud.fg != None :
        noeud = noeud.fg
    return noeud.info
```

Info : le symbole `-> int:` placé entre *def min(self)* et : indique que cette fonction va renvoyer un entier. **C'est uniquement indicatif**. Si cette fonction renvoie autre chose qu'un entier, rien ne se produira. Vous utiliserez cette notation dans ce tp pour vous l'approprier. Pour chaque fonction écrite, vous déclarerez un des types suivants : int, str, bool, none pour le retour attendu.

b. CREATION DE LA METHODE RECHERCHE() :

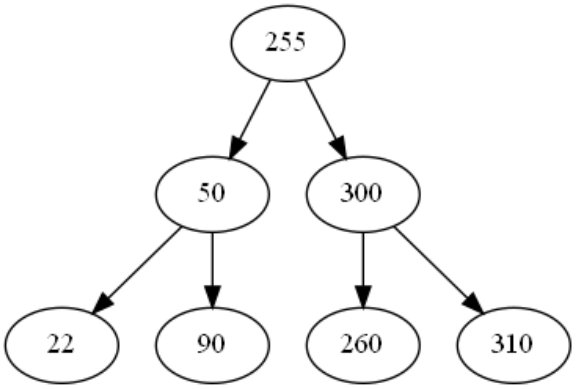
⇨ Ecrire le code **itératif** de la méthode *recherche()* qui prend en argument une valeur et retourne *True* si la valeur existe, *False* sinon.

```
>>> A.recherche(260)
True

>>> A.recherche(261)
False
```

```
def recherche(self,val)-> bool:
    noeud = self
    while val != noeud.info :
        if val < noeud.info :
            if noeud.fg != None : noeud = noeud.fg
            else : return False
        elif val > noeud.info :
            if noeud.fd != None : noeud = noeud.fd
            else : return False
    return True
```

c. CREATION DE LA METHODE INSERE() :

⇨ Ecrire le code **itératif** de la méthode *insere()* qui prend en argument une valeur et insère dans l'arbre un nouveau nœud dont l'attribut *info* est égal à cette valeur. Si la valeur est déjà présente dans l'arbre, la méthode retourne *False.* Elle retourne *True* dans le cas contraire.




```
>>> A.insere(50)
False

>>> A.insere(90)
True

>>> A.insere(22)
True

>>> A.view()
```

```
for _ in range(50) :
    n = randint(0,500)
    A.insere(n)
```

⇨ Tester la méthode avec l'exécution donnée ci-contre. Vérifier visuellement que l'arbre obtenu est bien un ABR.

```
def insere(self,val)-> bool:
    if self.info == None :
        self.info = val
        return True
    noeud = self
    while val != noeud.info :
        if val < noeud.info :
            if noeud.fg != None : noeud = noeud.fg
            else :
                noeud.fg = Abr(val)
                return True
        elif val > noeud.info :
            if noeud.fd != None : noeud = noeud.fd
            else :
                noeud.fd = Abr(val)
                return True
    return False
```

⇨ UPLOADER le fichier *abrIteratif.py* sur nsibranly.fr.

3- UTILISATION DE LA CLASSE ABR AVEC LES MOTS DU DICTIONNAIRE :

⇨ Ouvrir un nouveau fichier nommé *abrDictionnaire.py* dans le même répertoire que celui où se trouve le fichier *abrRecursif.py* .

⇨ Y copier-coller le code du fichier *dictionnaire.py* disponible sur *nsibranly.fr*.

⇨ Télécharger dans ce répertoire de travail le fichier *dictionnaire.txt* aussi disponible sur *nsibranly.fr*.

```
from abrRecursif import graphe, File, Abr
from random import randint

def lectureFichier() :
    l = []
    f = open("dictionnaire.txt","r",encoding='utf8')
    mot = f.readline()
    while mot != "" :
        mot = mot[:-1]
        mot = mot.lower()
        accent = True
        for c in mot :
            if c in "éçùêèàï" : accent = False
        if accent : l.append(mot)
        mot = f.readline()
    f.close()
    return l


# Main
A = Abr()
l = lectureFichier()
i = randint(0,len(l)-1)
mot = l[i]
A.insere(mot)

A.view()
```

On importe les classes du fichiers *abrRecursif.py*

Cette fonction renvoie tous les mots sans accent contenus dans le fichier *dictionnaire.txt* .

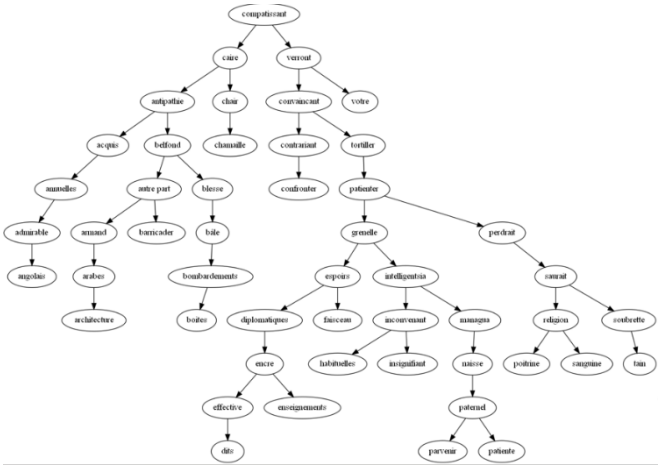
⇨ Dans le fichier *abrRecursif.py* qui est ici importé, placer les exécutions de la partie *programme principal* dans le bloc ***if*** donné ci-contre, afin de ne pas les exécuter si ce fichier est utilisé en tant que bibliothèque.

```
# main
if __name__ == '__main__' :
    A = Abr(255,
            Abr(50),
            Abr(300,
                Abr(240) ,
                Abr(310)
                ),
            )
```

⇨ Exécuter ce code. Normalement il crée un arbre nommé A dont la racine est un mot pris au hasard dans la liste l, donc dans le fichier *dictionnaire.txt* .

⇨ Créer une boucle dans programme principal, qui insère dans cet arbre, 50 mots tirés au sort dans la liste *l* . Exécuter plusieurs fois ce code et observer l'arbre obtenu. Celui-ci se rapproche-t-il plus d'un arbre complet ou d'un arbre peigne ?

```
A = Abr()
l = lectureFichier()
nb = 0
while len(l) > 0 and nb < 50 :
    nb = nb + 1
    i = randint(0,len(l)-1)
    mot = l.pop(i)
    A.insere(mot)
```



Dans la liste l, les mots sont classés par ordre alphabétique. Si on insérait les mots dans cet ordre, l'arbre obtenu aurait une structure peigne parfaite (vous pouvez essayer). Pour éviter d'obtenir un arbre déséquilibré, on opte ici pour une insertion par tirage au sort.

⇨ Enrichir la classe *Abr* du fichier *abrRecursif.py* avec les méthodes *taille()* et *hauteur()* mises au point dans le tp précédent, pour les arbres binaires en général. Ces méthodes sont bien sûr aussi fonctionnelles pour des ABR qui sont avant tout des arbres binaires.

```
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

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)
```

⇨ Ecrire dans le programme principal, une boucle qui permet d'insérer dans l'arbre A, les 16058 mots tirés au sort dans la liste l. Ne pas afficher l'arbre dans ce cas, cela dépasse les capacités du logiciel graphviz.

```
A = Abr()
l = lectureFichier()
while len(l) > 0 :
    i = randint(0,len(l)-1)
    mot = l.pop(i)
    A.insere(mot)
```

⇨ Déterminer la taille de l'arbre :

```
>>> A.taille()
16058
```

Les réponses demandées dans la suite sont à écrire en commentaire dans le fichier *abrDictionnaire.py*

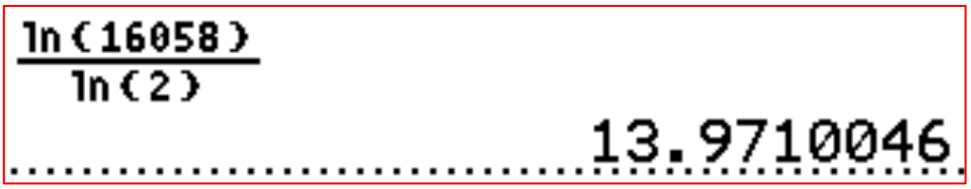
⇨ Déterminer la hauteur de l'arbre en appliquant la méthode *hauteur().* Faire une dizaine d'exécution différentes et calculer la valeur moyenne de cette hauteur.

```
>>> A.hauteur()
32
```

```
>>> A.hauteur()
33
```

```
>>> A.hauteur()
34
```

⇨ Si le remplissage de l'arbre conduisait à un arbre complet, la hauteur de l'arbre serait égale à $log_2(16058)$. Utiliser la calculatrice pour calculer $log_2(16058) = \frac{\ln(16058)}{\ln(2)}$. L'insertion des 16058 mots par tirage au sort permet-elle d'obtenir un arbre complet ?



⇨ Réaliser une recherche des mots 'zone' et 'nsi' :

```
>>> A.recherche('zone')
True

>>> A.recherche('nsi')
False
```

⇨ 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 une liste, d'avoir la même complexité ?
La complexité est en $O(log_2(n))$ si l'arbre est complet, ce qui est à peu près le cas si les insertions se font aléatoirement. Une recherche dichotomique permet également d'obtenir une complexité en $O(log_2(n))$.

⇨ Enrichir la classe *Abr* avec un attribut de classe nommé nb.

Utiliser cet attribut pour créer un compteur d'appel de la méthode *recherche()*.
Donner finalement le nombre d'appels de cette méthode pour les exécutions `A.recherche('nsi')` et `A.recherche('zone')`

```
class Abr :
    liste = []
    nb = 0

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

```
def recherche(self,val)-> bool:
    Abr.nb = Abr.nb + 1
    if val == self.info : return True
    elif val < self.info :
        if self.fg != None : return self.fg.recherche(val)
        else : return False
    elif val > self.info :
        if self.fd != None : return self.fd.recherche(val)
        else : return False
```

```
>>> A.recherche('zone')
True

>>> A.nb
13
```

```
>>> A.recherche('nsi')
False

>>> A.nb
37

>>> A.hauteur()
35
```