Exercices Arbres

Exercice 1 : ABR et POO

Cet exercice porte sur les arbres binaires de recherche et la programmation orientée objet.

On rappelle qu'un arbre binaire est composé de nœuds, chacun des nœuds possédant éventuellement un sous-arbre gauche et éventuellement un sous-arbre droit. Un nœud sans sous-arbre est appelé feuille. La taille d'un arbre est le nombre de nœuds qu'il contient ; sa hauteur est le nombre de nœuds du plus long chemin qui joint le nœud racine à l'une des feuilles. Ainsi la hauteur d'un arbre réduit à un nœud, c'est-à-dire la racine, est 1.

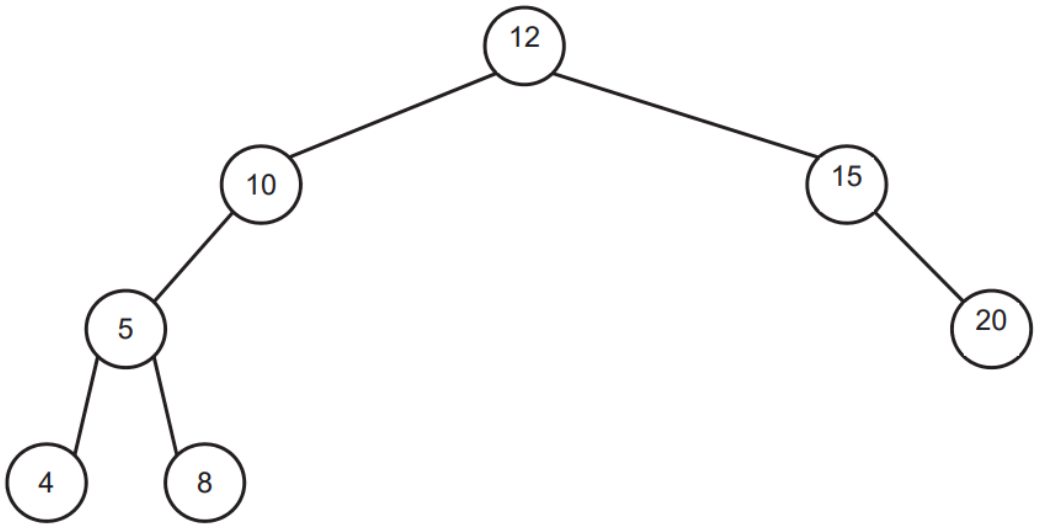
Dans un arbre binaire de recherche, chaque nœud contient une clé, ici un nombre entier, qui est :

- strictement supérieure à toutes les clés des nœuds du sous-arbre gauche ;
- strictement inférieure à toutes les clés des nœuds du sous-arbre droit.

Ainsi les clés de cet arbre sont toutes distinctes.

Un arbre binaire de recherche est dit « bien construit » s'il n'existe pas d'arbre de hauteur inférieure qui pourrait contenir tous ses nœuds.

On considère l'arbre binaire de recherche ci-dessous.



1. **a**. Quelle est la taille de l'arbre ci-dessus ?
   b. Quelle est la hauteur de l'arbre ci-dessus ?

2. Cet arbre binaire de recherche n'est pas « bien construit ». Proposer un arbre binaire de recherche contenant les mêmes clés et dont la hauteur est plus petite que celle de l'arbre initial.

3. Les classes `Noeud` et `Arbre` ci-dessous permettent de mettre en œuvre en Python la structure d'arbre binaire de recherche. La méthode `insere` permet d'insérer récursivement une nouvelle clé.

```
class Noeud :

    def __init__(self, cle):
        self.cle = cle
        self.gauche = None
        self.droit = None

    def insere(self, cle):
        if cle < self.cle :
            if self.gauche == None :
                self.gauche = Noeud(cle)
            else :
                self.gauche.insere(cle)
        elif cle > self.cle :
            if self.droit == None :
                self.droit = Noeud(cle)
            else :
                self.droit.insere(cle)
```

```
class Arbre :

    def __init__(self, cle):
        self.racine = Noeud(cle)

    def insere(self, cle):
        self.racine.insere(cle)
```

Donner la représentation de l'arbre codé par les instructions ci-dessous.

```
a = Arbre(10)
a.insere(20)
a.insere(15)
a.insere(12)
a.insere(8)
a.insere(4)
a.insere(5)
```

4. Pour calculer la hauteur d'un arbre non vide, on a écrit la méthode ci-dessous dans la classe `Noeud`.

```
def hauteur(self):
    if self.gauche == None and self.droit == None:
        return 1
    if self.gauche == None:
        return 1+self.droit.hauteur()
    elif self.droit == None:
        return 1+self.gauche.hauteur()
    else:
        hg = self.gauche.hauteur()
        hd = self.droit.hauteur()
        if hg > hd:
            return hg+1
        else:
            return hd+1
```

Écrire la méthode `hauteur` de la classe `Arbre` qui renvoie la hauteur de l'arbre.

5. Écrire les méthodes `taille` des classes `Noeud` et `Arbre` permettant de calculer la taille d'un arbre.

6. On souhaite écrire une méthode `bien_construit` de la classe `Arbre` qui renvoie la valeur `True` si l'arbre est « bien construit » et `False` sinon.

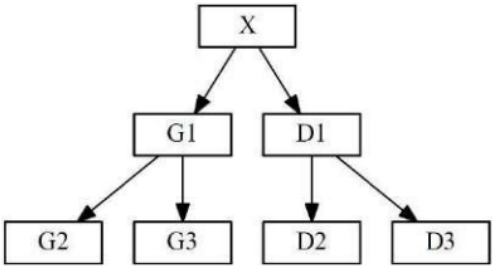
   On rappelle que la taille maximale d'un arbre binaire de recherche de hauteur h est $2^h - 1$.

   a. Quelle est la taille minimale, notée t_min, d'un arbre binaire de recherche « bien construit » de hauteur h ?
   b. Écrire la méthode `bien_construit` demandée.

Exercice 2 : ABR

Notion abordée : les arbres binaires de recherche.

Un arbre binaire est soit vide, soit un nœud qui a une valeur et au plus deux fils (le sous-arbre gauche et le sous-arbre droit).



X est un nœud, sa valeur est X.valeur

G1 est le fils gauche de X, noté X.fils_gauche

D1 est le fils droit de X, noté X.fils_droit

Un arbre binaire de recherche est ordonné de la manière suivante :

Pour **chaque** nœud X,

- les valeurs de tous les nœuds du sous-arbre gauche sont **strictement inférieures** à la valeur du nœud X
- les valeurs de tous les nœuds du sous-arbre droit sont **supérieures ou égales** à la valeur du nœud X

Ainsi, par exemple, toutes les valeurs des nœuds G1, G2 et G3 sont strictement inférieures à la valeur du nœud X et toutes les valeurs des nœuds D1, D2 et D3 sont supérieures ou égales à la valeur du nœud X.

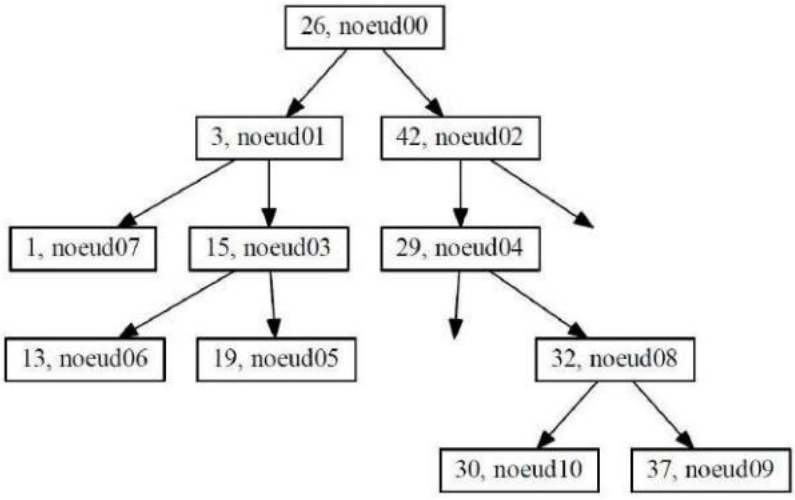
Voici un exemple d'arbre binaire de recherche dans lequel on a stocké dans cet ordre les valeurs :

[26, 3, 42, 15, 29, 19, 13, 1, 32, 37, 30]

L'étiquette d'un nœud indique la valeur du nœud suivie du nom du nœud.

Les nœuds ont été nommés dans l'ordre de leur insertion dans l'arbre ci-dessous.

`'29, noeud04'` signifie que le nœud nommé `noeud04` possède la valeur `29`.



1. On insère la valeur 25 dans l'arbre, dans un nouveau nœud nommé nœud11.
   Recopier l'arbre binaire de recherche étudié et placer la valeur 25 sur cet arbre en coloriant en rouge le chemin parcouru.
   Préciser sous quel nœud la valeur 25 sera insérée et si elle est insérée en fils gauche ou en fils droit, et expliquer toutes les étapes de la décision.

2. **Préciser** toutes les valeurs entières que l'on peut stocker dans le nœud fils gauche du nœud04 (vide pour l'instant), en respectant les règles sur les arbres binaires de recherche ?

3. Voici un algorithme récursif permettant de parcourir et d'afficher les valeurs de l'arbre :

```
Parcours(A)        # A est un arbre binaire de recherche
  Afficher(A.valeur)
  Parcours(A.fils_gauche)
  Parcours(A.fils_droit)
```

   3.a. **Écrire** la liste de toutes les valeurs dans l'ordre où elles seront affichées.
   3.b. **Choisir** le type de parcours d'arbres binaires de recherche réalisé parmi les propositions suivantes : Préfixe, Suffixe ou Infixe

4. En vous inspirant de l'algorithme précédent, écrire un algorithme `Parcours2` permettant de parcourir et d'afficher les valeurs de l'arbre `A` dans l'ordre croissant.