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

Dans cet exercice, 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 (nœuds sans sous-arbres). On convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1 et la hauteur de l'arbre vide vaut 0.

On considère l'arbre binaire représenté ci-dessous :

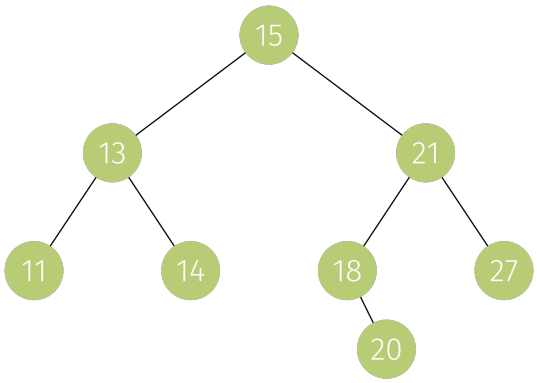
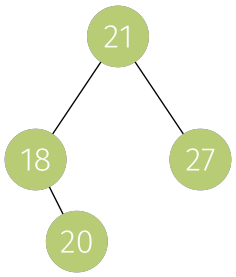


Figure 1

Donner la taille et la hauteur de cet arbre.

Réponse

> La taille est 8 et la hauteur 4.

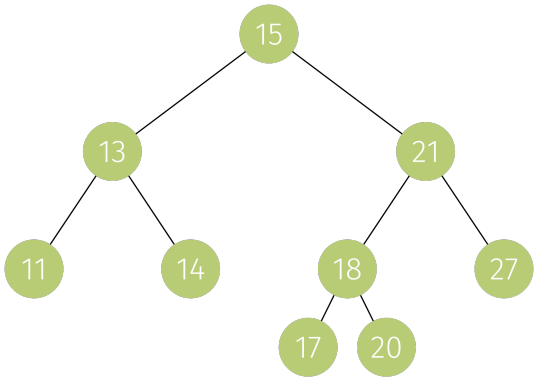


1. Représenter ci-contre le sous-arbre droit du nœud de valeur 15.

2. Justifier que l'arbre de la figure 1 est un arbre binaire de recherche.

Réponse

> C'est un arbre binaire de recherche car pour chaque nœud *n*, le sous-arbre gauche (respectivement droit) est composé de nœuds de valeurs inférieures (respectivement supérieures) ou égales à celle de *n*.

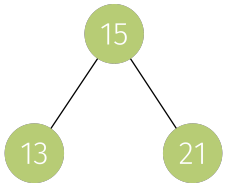


On insère la valeur 17 dans l'arbre de la figure 1 de telle sorte que 17 soit une nouvelle feuille de l'arbre et que le nouvel arbre obtenu soit encore un arbre binaire de recherche.

3. Représenter ci-contre ce nouvel arbre.

On considère la classe `Noeud` définie de la façon suivante en Python :

```
class Noeud:
    def __init__(self, g, v, d):
        self.gauche = g
        self.valeur = v
        self.droit = d
```



4. Parmi les trois instructions suivantes, entourer celle qui construit et stocke dans la variable abr l'arbre représenté ci-contre.

```
abr = Noeud(Noeud(Noeud(None, 13, None), 15, None), 21, None)
abr = Noeud(None, 13, Noeud(Noeud(None, 15, None), 21, None))
abr = Noeud(Noeud(None, 13, None), 15, Noeud(None, 21, None))
```

5. Compléter le code de la fonction `ins` ci-dessous qui prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et qui renvoie l'arbre obtenu suite à l'insertion de la valeur `v` dans l'arbre `abr`.
Les lignes 8 et 9 permettent de ne pas insérer la valeur `v` si celle-ci est déjà présente dans `abr`.

```
1 def ins(v, abr):
2   if abr is None:
3       return Noeud(None, v, None)
4   if v > abr.valeur:
5       return Noeud(abr.gauche, abr.valeur, ins(v, abr.droit))
6   elif v < abr.valeur:
7       return Noeud(ins(v,abr.gauche), abr.valeur, abr.droit)
8   else:
9       return abr
```

La fonction `nb_sup` prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et renvoie le nombre de valeurs supérieures ou égales à la valeur `v` dans l'arbre `abr`.

Le code de cette fonction `nb_sup` est donné ci-dessous :

```
def nb_sup(v, abr):
    if abr is None:
        return 0
    else:
        if abr.valeur >= v:
            return 1 + nb_sup(v, abr.gauche) + nb_sup(v, abr.droit)
        else:
            return nb_sup(v, abr.gauche) + nb_sup(v, abr.droit)
```

On exécute l'instruction `nb_sup(16, abr)` dans laquelle `abr` est l'arbre initial de la figure 1.

6. Déterminer le nombre d'appels à la fonction `nb_sup` lors de cette exécution.

L'arbre passé en paramètre étant un arbre binaire de recherche, on peut améliorer la fonction `nb_sup` précédente afin de réduire ce nombre d'appels.

7. Écrire sur la copie le code modifié de cette fonction.