Ce cours est inspiré de https://glassus.github.io/terminale_nsi/T1_Structures_de_donnees/1.3_Arbres/cours/

Rappel : Arbre = Graphe connexe sans cyle. Une structure arborescente est utile pour représenter des données présentant une certaine hierarchie.

Exercice 1

Arbres binaires, définition

Un **arbre binaire** est un arbre dont chaque nœud possède au plus deux fils.

Rappel : La **hauteur** d'un arbre admet plusieurs définitions. L'une de ces définitions est la **profondeur maximale** de cet arbre. Dans ce cas, un arbre ne contenant qu'un seul noeud a une hauteur de zero, et un arbre nul un hauteur de -1. Mais on peut aussi trouver la définition pour lequel l'arbre nul a une hauteur de 0, et la hauteur 1 pour un seul noeud.

Il est possible pour un arbre binaire de séparer le «dessous» de chaque nœud en deux **sous-arbres** (éventuellement vides) : le sous-arbre gauche et le sous-arbre droit.

Voir le cours sd5 sur les Arbres pour des exemple d'implémentation en Liste imbriquée, dictionnaire, objet, ou liste simple

1.1 Cas des arbres binaires complets

Avec la première defintion de la hauteur **n** vue en cours (profondeur maximale) : La **taille d'un arbre binaire complet** est égale à $2^{h+1} - 1$.

Dans ce cas, le nombre de noeuds, appelée *taille*, est egale à la somme des termes d'une suite géometrique (de raison q = 2). Alors :

$$S = 1 + q + q^2 + q^3 + ... + q^h$$

Soit :

$$S * q = q + q^2 + ...q^{h+1}$$

Si on soustrait ces 2 égalités, on obtient :

$$S = \frac{1-q^{h+1}}{1-q}$$

Il vient :

$$S = 2^{h+1} - 1$$

1.2 Implémentation sans encapsulation

(sans méthode de type mutateur/ accesseur)

```
class Arbre:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
6
7  a = Arbre(4)
8  a.left = Arbre(3)
9  a.right = Arbre(1)
10 a.right.left = Arbre(2)
11 a.right.right = Arbre(7)
12 a.left.left = Arbre(6)
13 a.right.right.left = Arbre(9)
```

Question : Représenter l'arbre défini ci-dessus.

1.3 Algorithme de calcul de la taille d'un arbre binaire

```
1 def taille(arbre):
2     if arbre is None:
3         return 0
4     else:
5         return 1 + taille(arbre.left) + taille(arbre.right)
```

1.4 Algorithme de calcul de la hauteur d'un arbre binaire

```
1 def hauteur(arbre):
2     if arbre is None:
3         return 0
4     else:
5         return 1 + max(hauteur(arbre.left), hauteur(arbre.right))
```

1.5 Calcul du nombre de feuilles d'un arbre binaire

```
1 def nb_feuilles(arbre):
2     if arbre is None:
3         return 0
4     if (arbre.left is None) and (arbre.right is None):
5         return 1
6     return nb_feuilles(arbre.left) +  nb_feuilles(arbre.right)
```

1.6 Recherche dans un arbre binaire

```
1 def recherche(arbre, valeur):
2     if arbre is None:
3         return False
4     if arbre.data ==  valeur:
5         return True
6     return recherche(arbre.left, valeur) or recherche(arbre.right,
    valeur)
```

Exercice 2

Arbres binaires de recherche (ABR)

2.1 Definition

Un arbre binaire de recherche (ABR) est un arbre binaire dont les valeurs des nœuds (valeurs qu'on appelle étiquettes, ou clés) vérifient la propriété suivante :

- l'étiquette d'un nœud est supérieure ou égale à celle de chaque nœud de son sous-arbre gauche.
- l'étiquette d'un nœud est strictement inférieure à celle de chaque nœud de son sous-arbre droit.

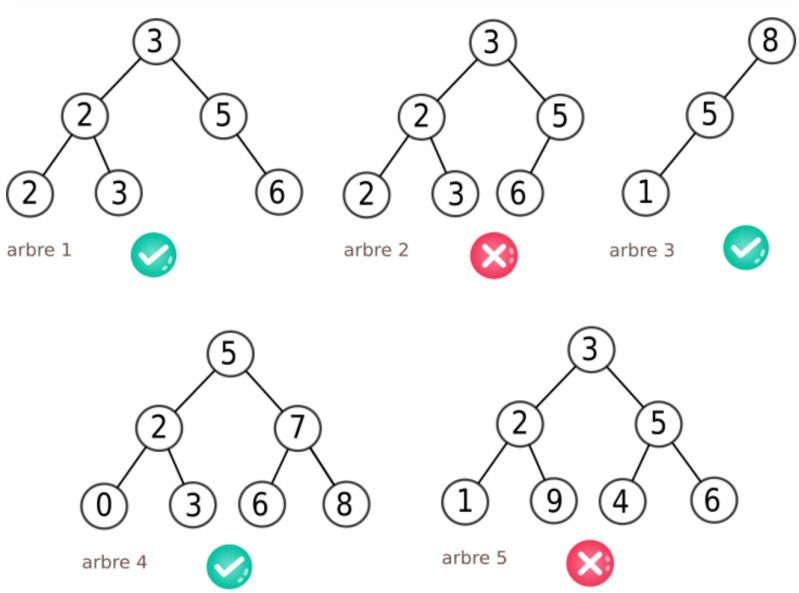

FIGURE 1 – lesquels de ces arbres sont des ABR ?

L'arbre 5 n'est pas un ABR à cause de la feuille 9, qui fait partie du sous-arbre gauche de 3 sans lui être inférieure.

Propriété : on peut aussi définir un ABR comme un arbre dont le parcours infixe est une suite croissante.

2.2 Test de l'état ABR ou non ABR d'un arbre binaire

2.2.1 Méthode utilisant un parcours infixe

On utilise la propriété d'un ABR : Son parcours infixe donne une suite croissante.

On va alors récupérer le parcours infixe dans une liste, et faire un test sur cette liste :

```
1 def infixe(arbre, s = None):
2     if s is None:
3         s = []
4     if arbre is None :
5         return None
6     infixe(arbre.left, s)
7     s.append(arbre.data)
8     infixe(arbre.right, s)
```

```
    return s


def est_ABR(arbre):
    '''renvoie un booléen indiquant si arbre est un ABR'''
    parcours = infixe(arbre)
    return parcours == sorted(parcours) # on regarde si le parcours est
    égal au parcours trié
```

2.2.2 Exercice

Pour chacun des 2 arbres définis ci-dessous, que retournnent chacune des instructions suivantes :

```
>>> est_ABR(a)
>>> est_ABR(b)
```

```
# arbres-tests

#arbre n°1
a = Arbre(5)
a.left = Arbre(2)
a.right = Arbre(7)
a.left.left = Arbre(0)
a.left.right = Arbre(3)
a.right.left = Arbre(6)
a.right.right = Arbre(8)

#arbre n°2
b = Arbre(3)
b.left = Arbre(2)
b.right = Arbre(5)
b.left.left = Arbre(1)
b.left.right = Arbre(9)
b.right.left = Arbre(4)
b.right.right = Arbre(6)
```

2.3 Rechercher une clé dans un ABR de taille n

Pour savoir si une valeur particulière fait partie des clés, on peut parcourir **tous les nœuds** de l'arbre, jusqu'à trouver (ou pas) cette valeur dans l'arbre. Dans le pire des cas, il faut donc faire **n** comparaisons.

Mais si l'arbre est un ABR, le fait que les valeurs soient «rangées» va considérablement améliorer la vitesse de recherche de cette clé, puisque la moitié de l'arbre restant sera écartée après chaque comparaison.

```
def contient_valeur(arbre, valeur):
    if arbre is None :
        return False
    if arbre.data == valeur :
        return True
    if valeur < arbre.data :
        return contient_valeur(arbre.left, valeur)
    else:
```

```
9           return contient_valeur(arbre.right, valeur)
```

Puis on teste :

```
>>> contient_valeur(a, 8)
True
>>> contient_valeur(b, 8)
False
```

2.3.1 Exercice

Représenter directement sur le schéma de ces 2 arbres, **a** et **b**, le parcours réalisé lors de la recherche de la valeur 8.

2.4 Complexité

Après chaque nœud, le nombre de nœuds restant à explorer est divisé par 2. On retrouve là le principe de recherche dichotomique.

S'il faut parcourir tous les étages de l'arbre avant de trouver (ou pas) la clé recherchée, le nombre de nœuds parcourus est donc égal à la hauteur **h** + 1 de l'arbre.

Pour une hauteur h donnée, le nombre de noeuds contenus est egal à $2^{h+1} - 1$.

Le nombre maximal de nœuds à parcourir pour rechercher une clé dans un ABR équilibré de taille n est donc de l'ordre de $log_2(n)$, ce qui est très performant !

Pour arbre contenant 1000 valeurs, 10 étapes suffisent : $log_2(1000) = 10$

Cette complexité logarithmique est un atout essentiel de la structure d'arbre binaire de recherche.

2.5 Insertion dans un ABR

L'insertion d'une clé va se faire au niveau d'une feuille, donc au bas de l'arbre. Dans la version récursive de l'algorithme d'insertion, que nous allons implémenter, il n'est pourtant pas nécessaire de descendre manuellement dans l'arbre jusqu'au bon endroit : il suffit de distinguer dans lequel des deux sous-arbres gauche et droit doit se trouver la future clé, et d'appeler récursivement la fonction d'insertion dans le sous-arbre en question.

Algorithme :

- Si l'arbre est vide, on renvoie un nouvel objet Arbre contenant la clé.
- Sinon, on compare la clé à la valeur du nœud sur lequel on est positionné :
  - Si la clé est inférieure à cette valeur, on va modifier le sous-arbre gauche en le faisant pointer vers ce même sous-arbre une fois que la clé y aura été injectée, par un appel récursif.
  - Si la clé est supérieure, on fait la même chose avec l'arbre de droite.
  - on renvoie le nouvel arbre ainsi créé.

```
def insertion(arbre, cle):
    if arbre is None :
        return Arbre(cle)
    else :
        val = arbre.data
        if cle <= val :
```

```
            arbre.left = insertion(arbre.left, cle)
        else:
            arbre.right = insertion(arbre.right, cle)
        return arbre
```

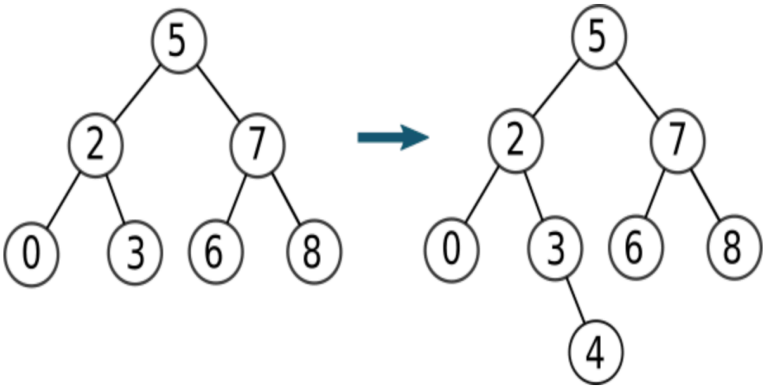

FIGURE 2 – insertion dans un ABR

Exercice 3

Exercice de type bac

Bac 2022 Metropole 2 : Exercice 1

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.

3.1 Question 1

On considère l'arbre binaire représenté ci-dessous :

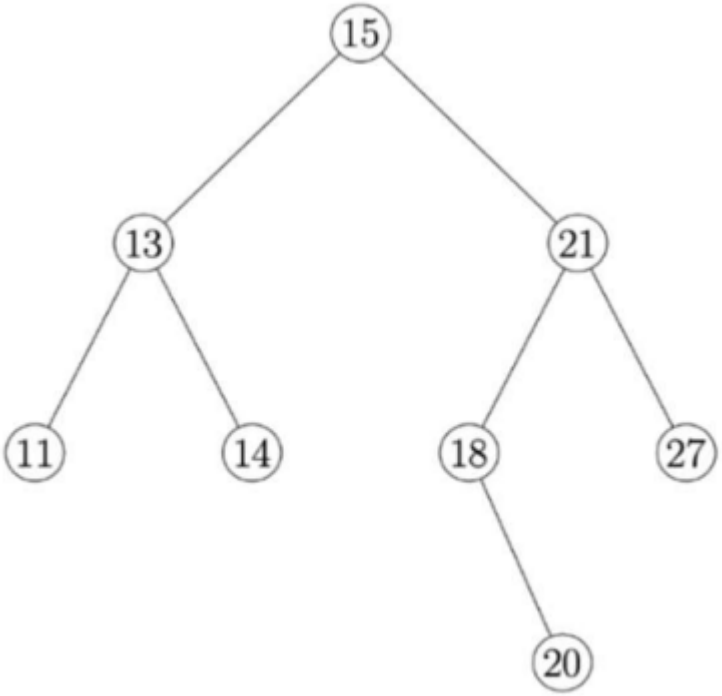

FIGURE 3 – ABR

A. Donner la taille de cet arbre.

B. Donner la hauteur de cet arbre.

C. Représenter sur la copie le sous-arbre droit du nœud de valeur 15.

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

E. 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. Représenter sur la copie ce nouvel arbre.

3.2 Question 2

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
```

A. Parmi les trois instructions (A), (B) et (C) suivantes, écrire sur la copie la lettre correspondant à celle qui construit et stocke dans la variable abr l'arbre représenté ci-contre.

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

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

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

B. Recopier et compléter la ligne 7 du 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 ........................................
8   else:
9     return abr
```

3.3 Question 3

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 :

```
1 def nb_sup(v, abr):
2   if abr is None:
3     return 0
4   else:
5     if abr.valeur >= v:
6       return 1+nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
7     else:
8       return nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
```

A. On exécute l'instruction `nb_sup(16, abr)` dans laquelle `abr` est l'arbre initial de la figure 1. Déterminer le nombre d'appels à la fonction `nb_sup`.

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