MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

Liberté
Égalité
Fraternité

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

IMPLANTATION DES ARBRES BINAIRES DE RECHERCHE À L'AIDE DE LA POO

SOMMAIRE

Prérequis

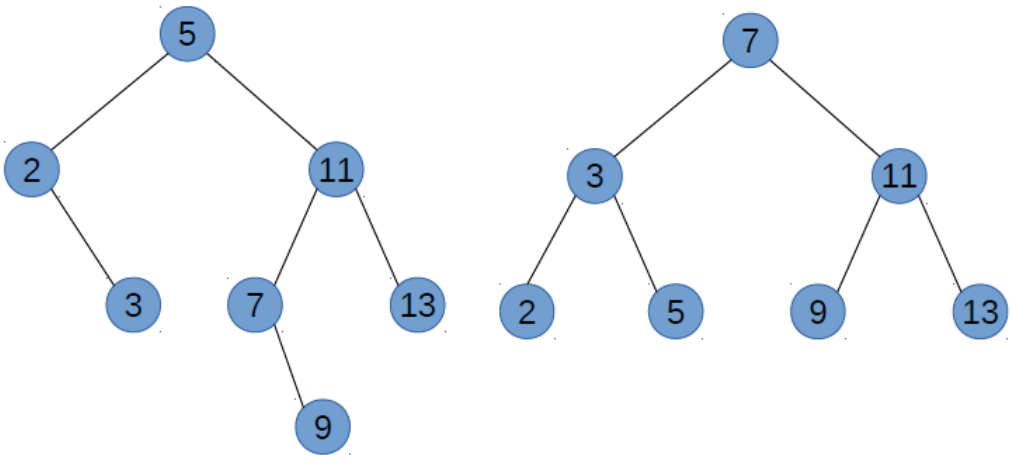
On suppose que la structure d'arbre, la programmation orientée objet et la récursivité ont déjà été abordées.

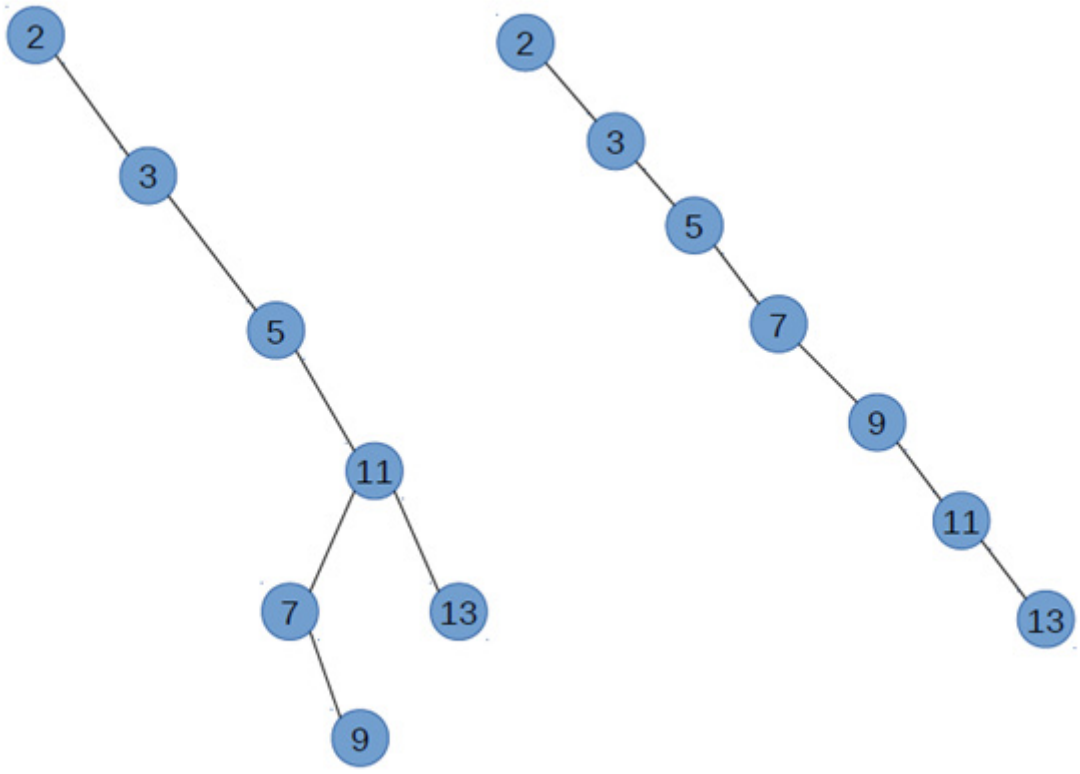
Introduction

Les **arbres binaires de recherche** (ABR) constituent une structure permettant de stocker un ensemble de données ordonnées, potentiellement efficace pour certains algorithmes d'accès, d'insertion ou de suppression de données. Un ABR est un arbre dont les nœuds comportent au plus 2 fils appelés *fils gauche* et *fils droit* et tel que pour tout nœud *n* :

- les valeurs « portées » par les nœuds « à gauche de n » doivent être inférieures à la valeur portée par le nœud n ;
- les valeurs « portées » par les nœuds « à droite de n » doivent être supérieures à la valeur portée par le nœud .

Un même ensemble ordonné peut être implanté par des arbres binaires différents. Prenons par exemple l'ensemble {2, 3, 5, 7, 9, 11, 13}. Il peut être implanté par les différents ABR suivants (liste non exhaustive) :





Ces différentes implantations sont équivalentes en place mémoire, mais l'efficacité en temps de certains algorithmes est par contre différente. Prenons le cas de la recherche d'un élément. La recherche se fait à partir de la racine de l'arbre, avec l'algorithme récursif suivant :

```
recherche (arbre, valeur) :
        si arbre.valeur == valeur :
                renvoyer vrai
        sinon si valeur < arbre.valeur :
                si sous_arbre_gauche existe :
                        renvoyer recherche(sous_arbre_gauche, valeur)
                sinon :
                        renvoyer faux
        sinon :
                si sous_arbre_droit existe :
                        renvoyer recherche(sous_arbre_droit, valeur)
                sinon :
                        renvoyer faux
```

Ainsi, le temps de recherche est directement proportionnel à la profondeur du nœud recherché dans l'arbre.

Si on définit à 1 la hauteur de la racine, le temps moyen de recherche est donc :

- pour le premier arbre : (1 + 2 + 2 + 3 + 3 + 3 + 4) / 7 = 18/7 ;
- pour le deuxième arbre : (1 + 2 + 2 + 3 + 3 + 3 + 3) / 7 = 17/7 ;
- pour le troisième arbre : (1 + 2 + 3 + 4 + 5 + 5 + 6) / 7 = 26/7 ;
- pour le quatrième arbre : (1 + 2 + 3 + 4 + 5 + 6 + 7) / 7 = 28/7.

Ainsi, dans le cas présent, la représentation 4 prend environ 50 % de temps en plus que la solution 1.

On comprend aisément qu'un arbre équilibré (solution 2) donne en moyenne de meilleurs résultats que tout autre arbre, et qu'un arbre dégénéré (solution 4) donne des résultats plus mauvais que toute autre représentation. La notion formelle d'arbre équilibré n'est pas au programme, mais elle peut être abordée intuitivement.

Sur un ABR équilibré, la recherche d'un élément est en moyenne en $\log_2(n)$, comme avec une recherche par dichotomie dans une liste. On peut alors se poser la question de l'intérêt d'utiliser un ABR. La réponse tient dans le temps mis à ajouter / supprimer un élément : dans une liste, on est en 0(n) (complexité linéaire), alors qu'on est en $\log_2(n)$ dans un ABR (complexité logarithmique).

Aussi, lorsqu'on doit stocker une collection d'éléments ordonnés, une liste Python peut être utilisée si la collection évolue peu ou bien si les éléments sont stockés dans l'ordre du tri. Mais si la collection doit être régulièrement modifiée et que les recherches par rapport au critère de tri sont fréquentes, on privilégie les ABR. On peut ainsi utiliser des ABR pour gérer la liste des adhérents à une bibliothèque (la recherche est en général faite sur le nom). Mais cela est peut-être moins adapté à une liste de films où la recherche peut être faite sur de nombreux critères différents (titre, acteurs, type, année de sortie, etc.).

Nous allons maintenant aborder :

- la représentation d'un ABR ;
- la recherche dans un ABR ;
- l'ajout et la suppression d'un élément dans un ABR.

En annexe, nous présentons le mécanisme AVL (Adelson-Veskill et Landis) permettant de maintenir des ABR relativement équilibrés lors de l'ajout et de la suppression d'un élément, mais cela ne fait pas partie du programme de NSI.

Représentation d'un arbre binaire de recherche

Il existe de nombreuses façons de représenter un ABR. On peut par exemple représenter le premier arbre ci-dessus par le tuple suivant : `(5, (2, None, 3), (11, (7, None, 9), 13))`. Cependant, l'implantation par des classes est la plus naturelle en Python. Par ailleurs, le programme précise : « L'exemple des arbres permet d'illustrer la programmation par classe ». C'est donc via des classes que l'implantation d'ABR est présentée dans ce document.

Structure permettant de définir un ABR

La structure d'un ABR se définit assez simplement, à l'aide de 2 classes :

```
class ABR:
    def __init__(self, racine = None):
        self.racine = racine
class Nœud:
    def __init__(self, valeur, noeud_gauche = None, noeud_droit = None):
        self.valeur = valeur
        self.noeud_gauche = noeud_gauche
        self.noeud_droit = noeud_droit
```

Cependant, c'est une structure récursive, et c'est cela qui peut la rendre compliquée à appréhender, d'autant plus que, comme souvent sur les structures récursives, les implantations les plus simples des algorithmes sont souvent des implantations récursives.

Pour construire l'ABR du premier exemple ci-dessus, on peut procéder ainsi :

```
n9 = Noeud(9)
n7 = Noeud(7, None, n9)
n13 = Noeud(13)
n11 = Noeud(11,n7, n13)
n3 = Noeud(3)
n2 = Noeud(2, None, n3)
n5 = Noeud(5, n2, n11)
arbre = ABR(n5)
```

Remarque sur la structure présentée

La classe avec structure récursive est la classe Noeud. La classe Arbre référence le nœud racine de l'arbre si celui-ci n'est pas vide.
On peut donc rajouter à la classe ABR la méthode suivante :

```
    def est_vide(self):
        return self.racine is None
```

On peut noter l'utilisation de « `is None` » et non « `== None` » pour vérifier qu'une variable contient bien la valeur spéciale None afin de passer outre l'éventuelle redéfinition de l'opérateur d'égalité.

Affichage d'un ABR

Visualisation texte

La représentation proposée est la suivante : pour chaque nœud non feuille, on affiche d'abord son sous-arbre gauche décalé de 2 espaces, puis son sous-arbre droit décalé de 2 espaces. Lorsqu'un nœud n'a qu'un fils, le sous-arbre vide est représenté par un « X » afin de savoir l'unique fils est un fils gauche ou un fils droit. Pour les nœuds feuille, on n'affiche pas de X pour les 2 sous-arbres vides.

Le premier arbre donné ci-dessus est affiché ainsi :

```
5
  2
    X
    3
  11
    7
      X
      9
    13
```

Pour calculer la représentation d'un ABR sous la forme d'une chaîne de caractères, on définit un méthode récursive `to_string` dans la classe `Noeud` qui prend en paramètre des espaces dont le nombre est proportionnel à la profondeur du nœud dans l'arbre. L'appel initial de cette méthode est effectué par la méthode `__str__` de la classe `ABR`.

Dans la représentation choisie, on représente les fils vides par des X, mais, dans le cas d'un nœud feuille, on ne représente pas ses fils. On a donc besoin de rajouter les méthodes suivantes :

- classe `ABR` :

```
    def __str__(self):
        if self.est_vide():
            representation = «Arbre Vide !»
        else:
            representation = self.racine.to_string(«»)
        return representation
```

- classe `Noeud` :

```
    def to_string(self, decalage):
        retour = decalage
        retour += str(self.valeur) + «\n»
        if self.est_feuille():
            return retour
        if self.noeud_gauche is None:
            retour += decalage + INDENTATION + «X\n»
        else:
            retour += self.noeud_gauche.to_string(decalage + INDENTATION)
        if self.noeud_droit is None:
            retour += decalage + INDENTATION + «X\n»
        else:
            retour += self.noeud_droit.to_string(decalage + INDENTATION)
        return retour
    def est_feuille(self):
        return self.noeud_gauche is None and self.noeud_droit is None
```

On prend également soin de définir la variable INDENTATION ainsi dans notre fichier python :

```
INDENTATION = «  « # indentation de 2 espaces
```

Visualisation graphique

Une visualisation graphique peut également être obtenue, par exemple en utilisant la bibliothèque *graphviz* si celle-ci a été installée. Voilà comment obtenir une telle représentation :

- au début du fichier Python :

```
from graphviz import Digraph
```

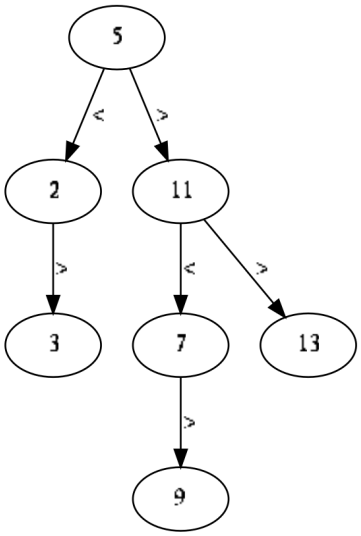
- Dans la classe ABR :

```
    def visu(self):
        assert not(self.est_vide())
        graphe = Digraph()
        self.racine.ajoute_noeud_visu(graphe)
        graphe.render(«arbre», view=True)
```

- dans la classe Nœud :

```
def ajoute_noeud_visu(self, graphe, parent = None, etiquette = None):
    noeud = str(self.valeur)
    graphe.node(nœud)
    if not(parent is None):
        graphe.edge(parent, nœud, label=etiquette)
    if not(self.noeud_gauche is None):
        self.noeud_gauche.ajoute_noeud_visu(graphe, nœud, «<»)
    if not(self.noeud_droit is None):
        self.noeud_droit.ajoute_noeud_visu(graphe, nœud, «>»)
```

On obtient alors, pour le premier arbre donné en exemple, l'affichage suivant :



Recherche dans un arbre binaire de recherche

Implanter récursivement la recherche dans un ABR est relativement simple. L'algorithme ayant déjà été donné dans l'introduction, voici son implantation via l'ajout de 2 méthodes :

- dans la classe `ABR` :

```
def rechercher(self, valeur):
    if self.est_vide():
        trouve = False
    else:
        trouve = self.racine.rechercher(valeur)
    return trouve
```

- dans la classe Nœud :

```
def rechercher(self, valeur):
    if self.valeur == valeur:
        return True
    elif valeur < self.valeur:
        if self.noeud_gauche is None:
            return False
        else:
            return self.noeud_gauche.rechercher(valeur)
    else:
        if self.noeud_droit is None:
            return False
        else:
            return self.noeud_droit.rechercher(valeur)
```

Cette version se contente de renvoyer `True` ou `False` suivant que la donnée est présente dans l'arbre ou pas. Si on souhaite renvoyer la donnée elle-même, il suffit de modifier ces méthodes pour renvoyer `None` à la place de `False`, et `self.valeur` à la place de `True`.

Modification d'un arbre binaire de recherche

Ajout d'un élément

L'ajout d'un élément doit être effectué en respectant la structure ordonnée de l'ABR. Par ailleurs, si l'élément est déjà présent dans l'ABR, on ne l'ajoute pas. On va donc parcourir l'ABR comme pour la recherche, mais si la branche que l'on doit parcourir est vide, c'est qu'on est arrivé à l'endroit où insérer la nouvelle valeur.

On ajoute donc les 2 méthodes suivantes :

- à la classe ABR :

```
    def inserer(self, valeur):
        if self.est_vide():
            self.racine = Nœud(valeur)
        else:
            self.racine.inserer(valeur)
```

- à la classe Nœud :

```
    def inserer(self, valeur):
        if valeur < self.valeur:
            if self.noeud_gauche is None:
                self.noeud_gauche = Noeud(valeur)
            else:
                self.noeud_gauche.inserer(valeur)
        elif valeur > self.valeur:
            if self.noeud_droit is None:
                self.noeud_droit = Noeud(valeur)
            else:
                self.noeud_droit.inserer(valeur)
```

Suppression d'un élément (hors programme)

La suppression d'un élément dans un ABR est plus complexe :

- si l'élément n'est pas présent dans l'arbre, il n'y a rien à faire ;
- si l'élément à supprimer est stocké dans un nœud feuille, cela ne pose pas de problème : il suffit de le supprimer ;
- si l'élément à supprimer est stocké dans un nœud n'ayant qu'un fils, il suffit de le remplacer par ce fils ;
- si l'élément à supprimer est stocké dans un nœud ayant 2 fils, il faut remplacer la valeur à supprimer soit par l'élément le plus grand de son sous-arbre gauche (il est plus grand que tous les autres éléments à gauche par définition et, par construction, plus petit que tous les éléments à droite), soit par l'élément le plus petit de son sous-arbre droit (il est plus petit que tous les autres éléments à droite par définition et, par construction, plus grand que tous les éléments à gauche). C'est cette deuxième solution qui est présentée ici.

Comme on peut le voir, la suppression d'un élément dans un ABR est relativement compliquée. Elle est d'ailleurs hors programme. Voici une implantation du processus, découpée en plusieurs méthodes pour rendre l'ensemble plus compréhensible :

- méthode ajoutée à la classe `ABR` :

```
    def supprimer(self, valeur):
        if self.est_vide():
            return
        else:
            self.racine = self.racine.supprimer(valeur)
```

- méthodes ajoutées à la classe `Nœud` :

```
    def supprimer(self, valeur):
        if valeur < self.valeur:
            self.noeud_gauche = self.noeud_gauche.supprimer(valeur)
            return self
        elif valeur > self.valeur:
            self.noeud_droit = self.noeud_droit.supprimer(valeur)
            return self
        else:
            return self.supprimer_noeud_courant()
    def supprimer_noeud_courant(self):
        if self.est_feuille():
            return None
        elif self.noeud_gauche is None:
            return self.noeud_droit
        elif self.noeud_droit is None:
            return self.noeud_gauche
        else:
            (valeur, nœud) = self.noeud_droit.chercher_et_supprimer_min()
            self.valeur = valeur
            self.noeud_droit = nœud
            return self
    def chercher_et_supprimer_min(self):
        if not(self.noeud_gauche is None):
            (valeur, nœud) = self.noeud_gauche.chercher_et_supprimer_min()
            self.noeud_gauche = nœud
            return (valeur, self)
        else:
            return (self.valeur, None)
```

Commentaires sur les méthodes de la classe `Nœud` :

`supprimer` recherche le nœud à supprimer. Une fois celui-ci trouvé, elle passe la main à la méthode supprimer_noeud_courant ;

`supprimer_noeud_courant` supprime le nœud courant dans les cas simples. Si on tombe sur un cas compliqué (le nœud courant a 2 fils), elle fait appel à la méthode chercher_et_supprimer_min ;

`chercher_et_supprimer_min`, comme son nom l'indique, recherche la plus petite valeur du sous-arbre droit, supprimer le nœud la contenant, et la renvoie.

Stocker et rechercher des données quelconques dans un ABR

Les ABR permettent de stocker et rechercher des données de n'importe quel type du moment que les opérateurs suivants sont disponibles sur le type en question : <, > et ==.

Ainsi, si on définit les 3 méthodes suivantes pour une classe, on peut stocker des instances de cette classe dans un ABR :

- `__eq__` pour l'opérateur ==
- `__lt__` pour l'opérateur <
- `__gt__` pour l'opérateur >

Bien sûr, un ABR ne permet de stocker des données triées que via un seul ordre (éventuellement multicritère).

Ainsi, les instances des 2 classes `Personne1` et `Personne2` ci-dessous sont classées différemment dans un ABR :

```
class Personne1:
    def __init__(self, id, nom, prenom):
        self.id = id
        self.nom = nom
        self.prenom = prenom
    def __str__(self):
        return self.prenom + « « + self.nom + « [« + str(self.id) + «]»
    def __eq__(self, autre):
        return self.id == autre.id
    def __lt__(self, autre):
        return self.id < autre.id
    def __gt__(self, autre):
        return self.id > autre.id
```

```
class Personne2:
    def __init__(self, id, nom, prenom):
        self.id = id
        self.nom = nom
        self.prenom = prenom
    def __str__(self):
        return self.prenom + « « + self.nom + « [« + str(self.id) + «]»
    def __eq__(self, autre):
        return self.nom == autre.nom \
            and self.prenom == autre.prenom \
            and self.id == autre.id
    def __lt__(self, autre):
        return self.nom < autre.nom or \
               (self.nom == autre.nom and self.prenom < autre.prenom) or \
               (self.nom == autre.nom and self.prenom == autre.prenom and
self.id < autre.id)
    def __gt__(self, autre):
        return self.nom > autre.nom or \
               (self.nom == autre.nom and self.prenom > autre.prenom) or \
               (self.nom == autre.nom and self.prenom == autre.prenom and
self.id > autre.id)
```

Une application possible à la suite du cours sur la programmation fonctionnelle est de montrer que l'on peut paramétrer un ABR par des fonctions permettant de définir un ordre sur les données qu'il contient. 2 fonctions sont alors nécessaires : l'égalité et la relation « < ». la relation « > » se déduit des 2 autres.

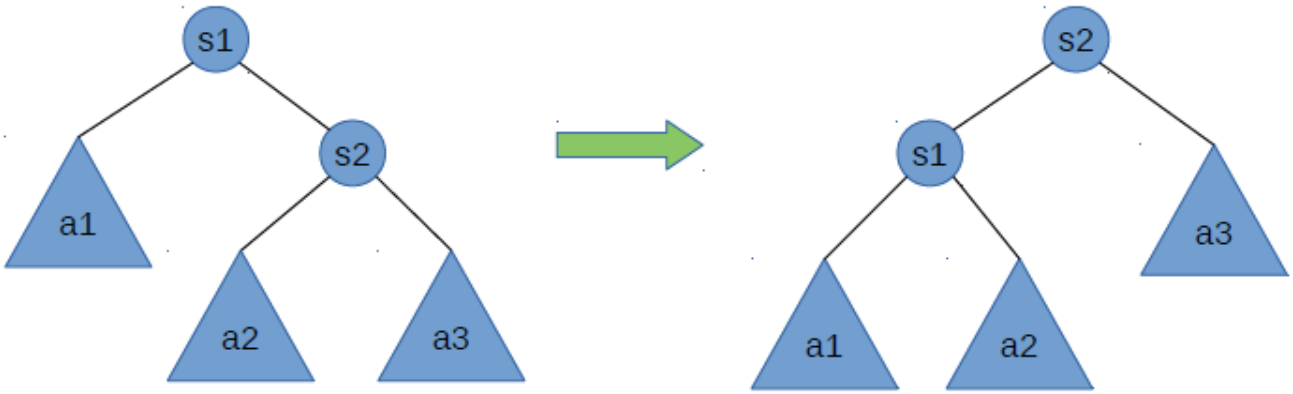
Arbres AVL (hors programme)

Les arbres AVL sont des arbres ABR maintenus *relativement équilibrés* par construction. Dans un arbre AVL, pour tout nœud, les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1 (avec, par convention, le fait que la hauteur d'un arbre vide est -1).

Lors d'une insertion dans un arbre AVL, on rééquilibre l'arbre des feuilles vers la racine grâce à un algorithme reposant sur 2 opérations appelées *rotation gauche* et *rotation droite* préservant les propriétés d'un arbre binaire de recherche.

Les principes de ces opérations sont présentés sur les 2 schémas suivants, sur lesquels les ronds représentent des nœuds et les triangles des sous-arbres :

- rotation gauche sur s1 :



Cette transformation préserve bien les propriétés d'un ABR puisque dans les 2 cas :

- les valeurs des nœuds de a1 sont plus petites que s1 et s2 ;
- les valeurs des nœuds de a3 sont plus grandes que s1 et s2 ;
- les valeurs des nœuds de a2 sont plus grandes que s1 et plus petites que s2 ;
- les arbres a1, a2 et a3 ne sont pas modifiés ;

- rotation droite sur s1 :

L'algorithme permettant de maintenir un arbre AVL relativement équilibré est alors le suivant :

```
Si hauteur(noeud_gauche) - hauteur(noeud_droit) = 2:
       Si hauteur(noeud_gauche.noeud_gauche) > hauteur(noeud_gauche.noeud_
droit) :
              rotation_droite()
       Sinon:
              rotation_gauche(noeud_gauche)
              rotation_droite()
Sinon Si hauteur(noeud_gauche) - hauteur(noeud_droit) = -2 :
       Si hauteur(noeud_droit.noeud_droit) > hauteur(noeud_droit.noeud_
gauche) :
              rotation_gauche()
       Sinon :
              rotation_droite(noeud_droit)
              rotation_gauche()
```

Pour implanter ce mécanisme, nous pouvons rajouter les méthodes suivantes :

- à la classe `ABR` :

```
    def inserer_AVL(self, valeur):
        if self.est_vide():
            self.racine = Noeud(valeur)
        else:
            self.racine = self.racine.inserer_AVL(valeur)
```

• à la classe `Nœud` :

```
    def inserer_AVL(self, valeur):
        if valeur < self.valeur:
            if self.noeud_gauche is None:
                self.noeud_gauche = Noeud(valeur)
                return self
            else:
                self.noeud_gauche = self.noeud_gauche.inserer_AVL(valeur)
                return self.equilibrer()
        elif valeur > self.valeur:
            if self.noeud_droit is None:
                self.noeud_droit = Noeud(valeur)
                return self
            else:
                self.noeud_droit = self.noeud_droit.inserer_AVL(valeur)
                return self.equilibrer()
        else:
            return self

    def equilibrer(self):
        hauteur_gauche = hauteur(self.noeud_gauche)
        hauteur_droit = hauteur(self.noeud_droit)
        if hauteur_gauche - hauteur_droit == 2:
            hauteur_gauche_gauche = hauteur(self.noeud_gauche.noeud_gauche)
            hauteur_gauche_droit = hauteur(self.noeud_gauche.noeud_droit)
            if hauteur_gauche_gauche > hauteur_gauche_droit:
                return self.rotation_droite()
            else:
                self.noeud_gauche = self.noeud_gauche.rotation_gauche()
                return self.rotation_droite()
        elif hauteur_gauche - hauteur_droit == -2:
            hauteur_droit_droit = hauteur(self.noeud_droit.noeud_droit)
            hauteur_droit_gauche = hauteur(self.noeud_droit.noeud_gauche)
            if hauteur_droit_droit > hauteur_droit_gauche:
                return self.rotation_gauche()
            else:
                self.noeud_droit = self.noeud_droit.rotation_droite()
                return self.rotation_gauche()
        else:
            return self

    def rotation_gauche(self):
        s1 = self
        s2 = s1.noeud_droit
        s1.noeud_droit = s2.noeud_gauche
        s2.noeud_gauche = s1
        return s2

    def rotation_droite(self):
        s1 = self
        s2 = s1.noeud_gauche
        s1.noeud_gauche = s2.noeud_droit
        s2.noeud_droit = s1
        return s2

    def hauteur(self):
        hauteur_gauche = hauteur(self.noeud_gauche)
        hauteur_droit = hauteur(self.noeud_droit)
        return 1 + max(hauteur_gauche, hauteur_droit)
```

- et la fonction utilitaire suivante :

```
def hauteur(nœud):
    if noeud is None:
        h = -1
    else:
        h = nœud.hauteur()
    return h
```

La bibliothèque binarytree

La bibliothèque Python *binarytree* (https://pypi.org/project/binarytree/) permet de réaliser aisément des arbres binaires. Les nœuds sont représentés par la classe `Node` qui possède 3 attributs : `val`, `left`, et `right`. La méthode `__str__` permet d'avoir une représentation texte particulièrement lisible. De nombreuses propriétés sont définies pour connaître la hauteur d'un arbre (`height`), la liste des feuilles (`leaves`), pour savoir si l'arbre est équilibré (`is_balanced`), s'il s'agit d'un arbre binaire de recherche (`is_bst`), etc.

Installation avec pip : `pip install binarytree`

Installation avec anaconda : `conda install -c conda-forge binarytree`

Cette bibliothèque propose notamment quelques méthodes de génération d'arbres aléatoires :

- arbre totalement aléatoire : `tree(hauteur_souhaitée)` ;
- arbre binaire de recherche : `bst(hauteur_souhaitée)`.

Passer la valeur `True` au paramètre nommé `is_perfect` permet d'obtenir un arbre complet.

Il est bien sûr possible de convertir un arbre représenté par notre classe ABR et arbre du module binarytree et inversement.

Création d'un `ABR` à partir d'un binarytree représentant déjà un arbre binaire de recherche :

```
def bst_to_noeud(noeud_bst):
    if noeud_bst.left is None:
        noeud_gauche = None
    else:
        noeud_gauche = bst_to_noeud(noeud_bst.left)
    if noeud_bst.right is None:
        noeud_droit = None
    else:
        noeud_droit = bst_to_noeud(noeud_bst.right)
    return Noeud(noeud_bst.val, noeud_gauche, noeud_droit)

def bst_to_arbre(noeud_bst):
    racine = bst_to_noeud(noeud_bst)
    arbre = ABR(racine)
    return arbre
```

Transformation d'un binarytree qui n'est pas forcément arbre binaire de recherche en ABR :

```
def binarytree_to_arbre(noeud_bst):
    arbre = ABR()
    for noeud in noeud_bst:
        arbre.inserer_AVL(noeud.val)
    return arbre
```

Création d'un binarytree à partir d'un ABR :

- méthode à ajouter à la classe `ABR` :

```
def to_binarytree(self):
  if self.est_vide():
    return None
  else:
    return self.racine.to_binarytree()
```

- méthode à ajouter à la classe Nœud :

```
    def to_binarytree(self):
        noeud = Node(self.valeur)
        if not(self.noeud_gauche is None):
            noeud.left = self.noeud_gauche.to_binarytree()
        if not(self.noeud_droit is None):
            noeud.right = self.noeud_droit.to_binarytree()
        return noeud
```

Pour avoir un affichage texte plus joli, on peut alors réécrire la méthode `__str__` de la classe ABR ainsi :

```
    def __str__(self):
        if self.est_vide():
            representation = «Arbre vide»
        else :
            representation = str(self.to_binarytree())
        return representation
```