Chapitre 5
Arbres (ABR)

Les arbres sont des structures de données **hiérarchiques** : chaque élément possède un élément parent et, éventuellement, plusieurs éléments enfants. On les rencontre partout en informatique : systèmes de fichiers, arbres généalogiques, pages web (le DOM), expressions arithmétiques, ou encore moteurs de recherche. Dans ce chapitre, nous étudions les **arbres binaires**, puis les **arbres binaires de recherche** (ABR), qui permettent des opérations de recherche et d'insertion très rapides.

I. Généralités sur les arbres

1. Une structure hiérarchique

Un arbre modélise une organisation « parent–enfant » : chaque élément a un seul parent, mais peut avoir plusieurs enfants.

Exemple 5.1.
Un **arbre généalogique** est un arbre : une personne a des parents, qui ont eux-mêmes des parents, et ainsi de suite. La structure est hiérarchique, de génération en génération. ■

Exemple 5.2.
Le **système de fichiers** d'un ordinateur est organisé en arbre : un dossier contient des fichiers et des sous-dossiers, qui contiennent eux-mêmes des fichiers et des sous-dossiers. La racine du système est le dossier qui contient tout le reste. ■

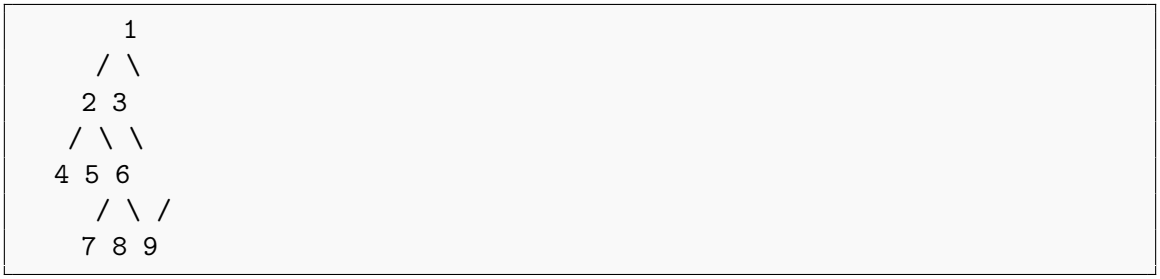
2. Vocabulaire : racine, nœud, feuille, branche

Définition 5.1.
Un **arbre** est un ensemble organisé de **nœuds** dans lequel chaque nœud a un **père**, sauf un nœud que l'on appelle la **racine**. Si un nœud n'a pas d'enfants, on dit que c'est une **feuille**. Les nœuds sont reliés par des **branches** (ou arêtes).

Pour la suite du chapitre, nous utilisons l'arbre de référence ci-dessous, dont les nœuds sont étiquetés par les entiers de 1 à 9.

Exemple 5.3.

Voici l'**arbre de référence** du chapitre : la racine est le nœud 1, ses enfants sont 2 et 3, etc.



■

■ **Exemple 5.4.**
Sur cet arbre, les **feuilles** sont les nœuds 4, 7, 8 et 9 (ils n'ont aucun enfant). Le **père** de 5 est 2, le père de 6 est 3. Les **enfants** de 5 sont 7 et 8 ; le nœud 1 n'a pas de père : c'est la racine. ■

3. Arbres étiquetés

Définition 5.2.
Un arbre dont tous les nœuds sont nommés est dit **étiqueté**. L'étiquette (ou nom du sommet) représente la « valeur » du nœud, c'est-à-dire l'information associée à ce nœud.

■ **Exemple 5.5.**
L'arbre de référence est étiqueté avec les entiers de 1 à 9 : l'étiquette du nœud racine est 1, celle de son fils gauche est 2, etc. ■

4. Arbres binaires

Définition 5.3.
Un **arbre binaire** est un arbre où chaque nœud a **au plus 2 enfants**. Un arbre binaire est une structure de nœuds **récursive** : un nœud a pour attributs :
- une **valeur** ;
- un **sous-arbre gauche** (qui peut être vide) ;
- un **sous-arbre droit** (qui peut être vide).

■ **Exemple 5.6.**
Une **expression arithmétique** comme $(2+3)\times(4-1)$ se représente par un arbre binaire : chaque opérateur binaire ($+$, $-$, $\times$, $/$) a exactement deux enfants, et chaque nombre est une feuille.

```
2 3 4 1
```

■

5. Représentation en Python : la classe Noeud

Un arbre sera représenté en Python par sa racine : un objet de la classe `Noeud`. Chaque nœud possède sa valeur, son sous-arbre gauche et son sous-arbre droit ; un sous-arbre vide est représenté par `None`.

```
class Noeud:
    def __init__(self):
        self.valeur = None
        self.gauche = None
        self.droite = None
```

■ **Exemple 5.7.**

On construit l'arbre de référence en créant les nœuds un par un et en les reliant : le nœud 1 a pour fils gauche le nœud 2 (dont les fils sont 4 et 5) et pour fils droit le nœud 3 (dont le fils droit est 6, lui-même père de 9). ■

```
Arbre = Noeud()
Arbre.valeur = 1
Arbre.gauche = Noeud()
Arbre.gauche.valeur = 2
Arbre.gauche.gauche = Noeud()
Arbre.gauche.gauche.valeur = 4
Arbre.gauche.droite = Noeud()
Arbre.gauche.droite.valeur = 5
Arbre.gauche.droite.gauche = Noeud()
Arbre.gauche.droite.gauche.valeur = 7
Arbre.gauche.droite.droite = Noeud()
Arbre.gauche.droite.droite.valeur = 8
Arbre.droite = Noeud()
Arbre.droite.valeur = 3
Arbre.droite.droite = Noeud()
Arbre.droite.droite.valeur = 6
Arbre.droite.droite.gauche = Noeud()
Arbre.droite.droite.gauche.valeur = 9
```

6. Taille et hauteur d'un arbre

Définition 5.4.
La **taille** d'un arbre est égale au nombre de nœuds de l'arbre.

Définition 5.5.
La **hauteur** (ou profondeur, ou niveau) d'un nœud X est égale au nombre d'arêtes qu'il faut parcourir à partir de la racine pour aller jusqu'au nœud X. Par convention, la hauteur de la racine est 0.

Définition 5.6.
La **hauteur d'un arbre** est égale à la hauteur (profondeur) du nœud le plus profond, c'est-à-dire le nombre d'arêtes du plus long chemin reliant la racine à une feuille.

■ **Exemple 5.8.**
L'arbre de référence a pour **taille** 9 : il contient exactement 9 nœuds. ■

■ **Exemple 5.9.**
La **hauteur** de l'arbre de référence est 3 : les plus longs chemins partent de la racine et arrivent aux feuilles 7, 8 ou 9, par exemple $1 \to 2 \to 5 \to 7$, qui compte 3 arêtes.
■

■ **Exemple 5.10.**
La hauteur du nœud 5 est 2 (chemins $1 \to 2 \to 5$), celle du nœud 7 est 3, celle de la racine est 0. ■

On peut calculer la hauteur d'un arbre par une fonction récursive : un arbre vide a une hauteur nulle, et un arbre non vide a pour hauteur 1 plus la hauteur maximale de ses deux sous-arbres.

```
def hauteur(Arbre):
    if Arbre == None:
        return 0
    else:
        return 1 + max(hauteur(Arbre.gauche), hauteur(Arbre.droite))
```

■ **Exemple 5.11.**
Appliquée à l'arbre de référence, cette fonction renvoie 4 : elle compte le nombre de **niveaux** (la racine compte pour 1), alors que la hauteur en nombre d'arêtes vaut 3. Les deux conventions existent ; l'important est de rester cohérent. ■

De même, la taille se calcule récursivement : la taille d'un arbre vide vaut 0, celle d'un arbre non vide vaut 1 plus les tailles de ses deux sous-arbres.

```
def taille(Arbre):
    if Arbre == None:
        return 0
    else:
        return 1 + taille(Arbre.gauche) + taille(Arbre.droite)
```

■ **Exemple 5.12.**
Appliquée à l'arbre de référence, cette fonction renvoie 9 : $1 + \text{taille}(2) + \text{taille}(3) = 1 + 3 + 2 + \cdots$ ■

Propriété 5.1.
Pour un arbre binaire de taille n et de hauteur h (exprimée en nombre de niveaux, c'est-à-dire le nombre renvoyé par la fonction `hauteur`), on a toujours :
- $n \leqslant 2^h - 1$: l'arbre le plus « rempli » possible est l'arbre parfait ;
- $h \leqslant n$: l'arbre le plus « étiré » possible est l'arbre dégénéré.

On en déduit l'encadrement $\log_2(n+1) \leqslant h \leqslant n$: la hauteur d'un arbre binaire de taille n est comprise entre $\log_2(n+1)$ et n.

7. Arbres complets, parfaits, dégénérés, équilibrés

Définition 5.7.
On distingue quelques formes particulières d'arbres binaires :
- un arbre binaire de recherche est dit **complet** si tous les niveaux de l'arbre sont remplis, sauf éventuellement le dernier, sur lequel les nœuds sont à

gauche ;
— un arbre binaire **parfait** est un arbre complet dont toutes les feuilles sont à la même hauteur (le dernier niveau est complètement occupé) ;
— un arbre binaire est dit **dégénéré** si chacun de ses nœuds a au plus un fils ;
— un arbre binaire est **équilibré** si tous les chemins de la racine aux feuilles ont la même longueur.

■ **Exemple 5.13.**
L'arbre de référence n'est ni complet ni parfait : le nœud 4 n'a aucun enfant alors que le nœud 5 en a deux, et les feuilles ne sont pas toutes à la même hauteur. Il n'est pas non plus dégénéré ni équilibré. ■

■ **Exemple 5.14.**
Voici un arbre **parfait** : toutes les feuilles sont au même niveau, et chaque nœud interne a exactement deux enfants. Sa taille est 7 et sa hauteur (en arêtes) est 2.

■

■ **Exemple 5.15.**
Voici un arbre **dégénéré** : chaque nœud a au plus un fils. L'arbre est réduit à une simple chaîne, comme une liste chaînée.

■

II. Parcours d'un arbre

Il existe diverses façons de parcourir les nœuds d'un arbre. On distingue le **parcours en largeur** et les **parcours en profondeur**.

1. Parcours en largeur

Définition 5.8.
Le **parcours en largeur** parcourt les nœuds **étage par étage**, de haut en bas et, pour chaque étage, de gauche à droite.

■ **Exemple 5.16.**
Le parcours en largeur de l'arbre de référence affiche : `1, 2, 3, 4, 5, 6, 7, 8, 9`. On visite d'abord la racine, puis tous les nœuds du niveau 1, puis tous ceux du niveau 2, etc. ■

Pour implémenter ce parcours, on utilise une **file** (structure FIFO, vue au chapitre 2) : on enfile la racine, puis tant que la file n'est pas vide, on défile un nœud, on l'affiche et on enfile ses deux enfants.

```
class File:
    def __init__(self):
        self.data = []

    def enfiler(self, elt):
        self.data.append(elt)

    def longueur(self):
        return len(self.data)

    def est_vide(self):
        return self.longueur() == 0

    def defiler(self):
        return self.data.pop(0)

    def dernier(self):
        return self.data[-1]

f = File()
f.enfiler(Arbre)
while not f.est_vide():
    s = f.defiler()
    print(s.valeur)
    if s.gauche != None:
        f.enfiler(s.gauche)
    if s.droite != None:
        f.enfiler(s.droite)
```

■ **Exemple 5.17.**
Ce programme affiche les valeurs de l'arbre de référence dans l'ordre du parcours en largeur : `1, 2, 3, 4, 5, 6, 7, 8, 9`. ■

2. Parcours en profondeur : préfixe, infixe, postfixe

Définition 5.9.
Le **parcours en profondeur** explore complètement un sous-arbre avant de commencer l'exploration de l'autre. Il existe trois façons de faire :

- le **parcours préfixe** (ou préordre, noté NGD) : on visite d'abord le nœud, puis son sous-arbre gauche, puis son sous-arbre droit ;
- le **parcours infixe** (ou en ordre, noté GND) : on visite d'abord le sous-arbre gauche, puis le nœud, puis le sous-arbre droit ;
- le **parcours postfixe** (ou postordre, noté GDN) : on visite d'abord le sous-arbre gauche, puis le sous-arbre droit, et enfin le nœud.

Voici le pseudo-code du parcours préfixe :

```
visiterPrefixe(Arbre A) :
    visiter(A)
    Si nonVide(gauche(A))
        visiterPrefixe(gauche(A))
    Si nonVide(droite(A))
        visiterPrefixe(droite(A))
```

■ **Exemple 5.18.**
Le parcours **préfixe** de l'arbre de référence donne : `1, 2, 4, 5, 7, 8, 3, 6, 9`. On affiche 1, puis on parcourt tout le sous-arbre gauche (2, 4, 5, 7, 8), puis tout le sous-arbre droit (3, 6, 9). ■

Voici le pseudo-code du parcours postfixe :

```
visiterPostfixe(Arbre A) :
    Si nonVide(gauche(A))
        visiterPostfixe(gauche(A))
    Si nonVide(droite(A))
        visiterPostfixe(droite(A))
    visiter(A)
```

■ **Exemple 5.19.**
Le parcours **postfixe** de l'arbre de référence donne : `4, 7, 8, 5, 2, 9, 6, 3, 1`. On parcourt le sous-arbre gauche, puis le sous-arbre droit, et on affiche le nœud à la fin. ■

Voici le pseudo-code du parcours infixe :

```
visiterInfixe(Arbre A) :
    Si nonVide(gauche(A))
        visiterInfixe(gauche(A))
    visiter(A)
    Si nonVide(droite(A))
        visiterInfixe(droite(A))
```

■ **Exemple 5.20.**
Le parcours **infixe** de l'arbre de référence donne : `4, 2, 7, 5, 8, 1, 3, 9, 6`. On parcourt le sous-arbre gauche, puis on affiche le nœud, puis on parcourt le sous-arbre droit. ■

Voici l'implémentation Python récursive des trois parcours en profondeur :

```
def parcours_prefixe(Arbre):
    if Arbre != None:
        print(Arbre.valeur)
        parcours_prefixe(Arbre.gauche)
        parcours_prefixe(Arbre.droite)

def parcours_infixe(Arbre):
    if Arbre != None:
        parcours_infixe(Arbre.gauche)
        print(Arbre.valeur)
        parcours_infixe(Arbre.droite)

def parcours_postfixe(Arbre):
    if Arbre != None:
        parcours_postfixe(Arbre.gauche)
        parcours_postfixe(Arbre.droite)
        print(Arbre.valeur)
```

Parcours	Ordre de visite	Résultat sur l'arbre de référence
Préfixe (NGD)	nœud, gauche, droite	1, 2, 4, 5, 7, 8, 3, 6, 9
Infixe (GND)	gauche, nœud, droite	4, 2, 7, 5, 8, 1, 3, 9, 6
Postfixe (GDN)	gauche, droite, nœud	4, 7, 8, 5, 2, 9, 6, 3, 1
Largeur	niveau par niveau	1, 2, 3, 4, 5, 6, 7, 8, 9

III. Les arbres binaires de recherche (ABR)

1. Principe et définition

En informatique, un **arbre binaire de recherche** (ou ABR ; en anglais *binary search tree*, BST) est une structure de données représentant un ensemble ou un tableau associatif dont les clés appartiennent à un ensemble totalement ordonné. Un arbre binaire de recherche permet des opérations rapides pour rechercher une clé, insérer ou supprimer une clé.

Définition 5.10.
Un **arbre binaire de recherche** est un arbre binaire dans lequel chaque nœud possède une **clé**, telle que :
- chaque nœud du **sous-arbre gauche** ait une clé **inférieure ou égale** à celle du nœud considéré ;
- chaque nœud du **sous-arbre droit** possède une clé **supérieure ou égale** à celle-ci.

Selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale. Les nœuds que l'on ajoute deviennent des feuilles de l'arbre.

■ **Exemple 5.21.**
Voici un arbre binaire de recherche dont les clés sont les entiers 0, 9, 10, 11, 12, 13 et 14 : chaque nœud est plus grand que tous les nœuds de son sous-arbre gauche et plus petit que tous les nœuds de son sous-arbre droit.

```
       13
      / \
    11 14
   / \
  9 12
 / \
0 10
```

■

■ **Exemple 5.22.**
Un ABR peut servir d'**annuaire** : chaque nœud associe une clé (par exemple un nom) à une valeur (par exemple un numéro de téléphone). Pour retrouver une entrée, on compare le nom cherché à la racine, puis on descend à gauche ou à droite : à chaque étape, la moitié de l'arbre restant est écartée. ■

On construit cet ABR en Python comme n'importe quel arbre binaire, avec la classe `Noeud` :

```
ABR = Noeud()
ABR.valeur = 13
ABR.gauche = Noeud()
ABR.gauche.valeur = 11
ABR.gauche.gauche = Noeud()
ABR.gauche.gauche.valeur = 9
ABR.gauche.droite = Noeud()
ABR.gauche.droite.valeur = 12
ABR.gauche.gauche.gauche = Noeud()
ABR.gauche.gauche.gauche.valeur = 0
ABR.gauche.gauche.droite = Noeud()
ABR.gauche.gauche.droite.valeur = 10
ABR.droite = Noeud()
ABR.droite.valeur = 14
```

2. Vérifier qu'un arbre est de recherche

La fonction `est_de_recherche` vérifie récursivement qu'un arbre binaire est un arbre de recherche : un arbre vide en est un ; sinon, on vérifie localement que la valeur du nœud est strictement supérieure à celle de son fils gauche (si ce fils existe) et strictement inférieure à celle de son fils droit, puis on vérifie récursivement les deux sous-arbres.

```
class Noeud:
    def __init__(self):
        self.valeur = None
        self.gauche = None
        self.droite = None

def est_de_recherche(Arbre):
    if Arbre == None:
        return True
    if Arbre.gauche != None and Arbre.valeur <= Arbre.gauche.valeur:
        return False
    if Arbre.droite != None and Arbre.valeur >= Arbre.droite.valeur:
        return False
    return est_de_recherche(Arbre.gauche) and est_de_recherche(Arbre.
        droite)
```

■ **Exemple 5.23.**
Appliquée à l'ABR construit plus haut, la fonction `est_de_recherche(ABR)` renvoie `True` : l'arbre est bien de recherche. On vérifie par exemple que $11 > 9$ et $11 < 12$, que $9 > 0$ et $9 < 10$, etc. ■

R La fonction `est_de_recherche` ci-dessus ne vérifie la propriété que **localement** : elle compare chaque nœud à ses deux fils directs, mais ne vérifie pas que *tous* les descendants respectent l'ordre. Par exemple, si le sous-arbre gauche de 10 contient un nœud 12 (plus grand que 10), la fonction renvoie `True` à tort. On supposera dans ce chapitre que les clés sont toutes distinctes ; une version qui vérifie les bornes sur tout le sous-arbre est proposée en exercice de réflexion.

3. Rechercher une clé

La recherche d'une clé dans un ABR est naturelle : on compare la clé cherchée à la valeur du nœud courant ; si elles sont égales, la clé est trouvée ; si la clé est plus petite, on continue dans le sous-arbre gauche ; sinon, dans le sous-arbre droit. Si l'on arrive sur un arbre vide, la clé n'est pas dans l'arbre.

```
def rechercher(Arbre, element):
    if Arbre == None:
        return False
    else:
        x = Arbre.valeur
        if x == element:
            return True
        elif element < x:
            return rechercher(Arbre.gauche, element)
        else:
            return rechercher(Arbre.droite, element)
```

■ **Exemple 5.24.**
`rechercher(ABR, 12)` renvoie `True` : $12 < 13$ donc on va à gauche ; $12 > 11$ donc on va à droite ; $12 = 12$: la clé est trouvée. ■

■ **Exemple 5.25.**
`rechercher(ABR, 1)` renvoie `False` : $1 < 13$ (gauche), $1 < 11$ (gauche), $1 < 9$ (gauche), $1 > 0$ (droite) ; le sous-arbre droit de 0 est vide, la fonction renvoie `False` : la clé 1 n'est pas dans l'arbre. ■

Propriété 5.2.
La recherche dans un arbre binaire de recherche de hauteur h effectue au plus h comparaisons : son coût est en $O(h)$. Si l'ABR est **équilibré**, sa hauteur est en $O(\log_2 n)$ où n est la taille de l'arbre, et la recherche est de coût **logarithmique**. Si l'ABR est dégénéré, la recherche retombe en $O(n)$: c'est le pire cas.

4. Insérer une clé

L'insertion d'une clé suit le même principe que la recherche : on descend dans l'arbre en comparant la clé à insérer aux valeurs rencontrées, jusqu'à trouver la place d'une nouvelle feuille. La version ci-dessous est dite **fonctionnelle** : elle ne modifie pas l'arbre donné, elle renvoie un *nouvel* arbre dans lequel la clé a été insérée. Elle recopie les nœuds du chemin parcouru et réutilise (par référence) les sous-arbres non modifiés.

```
def inserer(elt, Arbre):
    if Arbre == None:
        rep = Noeud()
        rep.valeur = elt
        return rep
    x = Arbre.valeur
    if elt < x:
        rep = Noeud()
        rep.valeur = x
        rep.gauche = inserer(elt, Arbre.gauche)
        rep.droite = Arbre.droite
        return rep
    else:
        rep = Noeud()
        rep.valeur = x
        rep.gauche = Arbre.gauche
        rep.droite = inserer(elt, Arbre.droite)
        return rep
```

■ **Exemple 5.26.**
On insère la clé 8 dans l'ABR : $8 < 13$ (gauche), $8 < 11$ (gauche), $8 < 9$ (gauche), $8 > 0$ (droite) ; le sous-arbre droit de 0 est vide, on crée une nouvelle feuille de valeur 8. On obtient l'arbre `ABR2` suivant : ■

```
       13
      / \
    11 14
   / \
  9 12
 / \
0 10
 \
  8
```

■ **Exemple 5.27.**
Le parcours en largeur de `ABR2` (réalisé avec la classe `File` vue au paragraphe II.1) affiche : `13, 11, 14, 9, 12, 0, 10, 8`. On remarque que l'arbre `ABR` d'origine n'a pas été modifié : `inserer` a renvoyé un nouvel arbre. ■

R Les clés insérées deviennent toujours des **feuilles** de l'arbre. La forme de l'ABR dépend donc de **l'ordre d'arrivée** des clés : si l'on insère des clés déjà triées, on obtient un arbre dégénéré (une simple chaîne), et la recherche retombe en $O(n)$.

5. Propriété remarquable : le parcours infixe

Propriété 5.3.
Le **parcours infixe** d'un arbre binaire de recherche affiche les clés dans l'**ordre croissant**.

■ **Exemple 5.28.**
Le parcours infixe de l'ABR du début de la section affiche : `0, 9, 10, 11, 12, 13, 14` : les clés sont bien triées dans l'ordre croissant. ■

IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien distinguer, pour chaque arbre, le vocabulaire (racine, feuille, père, enfant), les mesures (taille, hauteur) et les parcours (largeur, préfixe, infixe, postfixe).

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : vocabulaire des arbres, lecture de code court, prédiction d'affichage, complétion de code. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 5.1.

Dire si les affirmations suivantes sont vraies ou fausses, en justifiant brièvement.

1. Dans un arbre, la racine est le seul nœud qui n'a pas de père.
2. Une feuille est un nœud qui a exactement un enfant.
3. Dans un arbre binaire, chaque nœud a au plus deux enfants.
4. La taille d'un arbre est le nombre d'arêtes de l'arbre.
5. Dans un arbre binaire de recherche, tous les nœuds du sous-arbre gauche d'un nœud ont une clé inférieure ou égale à celle de ce nœud.
6. Une file fonctionne selon le principe FIFO (*first in, first out*) : le premier élément enfilé est le premier défiler. Rappeler pourquoi ce principe est utile pour le parcours en largeur.

■

■ Exercice 5.2.

On considère l'arbre de référence du cours (représenté ci-dessous).

1. Donner le numéro de la racine et celui des feuilles.
2. Quel est le père de 5 ? de 6 ? de 1 ?
3. Quels sont les enfants de 4 ? de 5 ? de 1 ?
4. Donner la taille et la hauteur (en nombre d'arêtes) de cet arbre.

■

■ **Exercice 5.3.**
Que va afficher le programme suivant (on suppose que la classe `Noeud` du cours est définie) ?

```
arbre = Noeud()
arbre.valeur = 4
arbre.gauche = Noeud()
arbre.gauche.valeur = 2
arbre.droite = Noeud()
arbre.droite.valeur = 6

def mystere(a):
    if a != None:
        mystere(a.gauche)
        print(a.valeur)
        mystere(a.droite)

mystere(arbre)
```

Quel parcours du chapitre réalise la fonction `mystere` ? ■

■ **Exercice 5.4.**
Rappel de la programmation orientée objet (vue au chapitre 0) : compléter la classe `Noeud` pour qu'un nœud possède une valeur, un sous-arbre gauche et un sous-arbre droit.

```
class Noeud:
    def __init__(self):
        self.valeur = ...
        self.gauche = ...
        self.droite = ...
```

■

■ **Exercice 5.5.**
Parmi les arbres suivants, lesquels sont des arbres binaires de recherche ? Justifier.

```
       7 10 5
     / \ / \ / \
    3 9 5 15 8 2
   / \ \ / \
  1 5 11 2 12
```

■

Exercices d'application

Les exercices d'application demandent d'écrire des fonctions complètes sur les arbres : mesures, parcours, recherche et insertion dans un ABR.

■ **Exercice 5.6.**

On souhaite calculer la taille d'un arbre binaire.

1. Écrire une fonction récursive `taille(Arbre)` qui renvoie le nombre de nœuds d'un arbre binaire. On précisera le cas d'arrêt (l'arbre vide) et l'appel récursif.
2. Tester sur l'arbre de référence du cours (taille attendue : 9) puis sur l'ABR construit au paragraphe III.1 (taille attendue : 7).

```
def taille(Arbre):
    ...
```

■

■ **Exercice 5.7.**

On souhaite calculer la hauteur d'un arbre binaire.

1. Écrire une fonction récursive `hauteur(Arbre)` qui renvoie la hauteur d'un arbre binaire. On précisera le cas d'arrêt et l'appel récursif.
2. Quelle valeur renvoie-t-elle pour l'arbre de référence du cours ? pour un arbre réduit à une seule feuille ? pour l'arbre vide ?
3. Quelle est la hauteur (en nombre d'arêtes) de l'arbre de référence ? Expliquer l'écart éventuel avec la valeur renvoyée par la fonction.

■

■ **Exercice 5.8.**

On souhaite implémenter les parcours en profondeur de l'arbre de référence du cours.

1. Écrire une fonction récursive `parcours_prefixe(Arbre)` qui affiche les valeurs dans l'ordre préfixe. On doit obtenir : `1, 2, 4, 5, 7, 8, 3, 6, 9`.
2. Écrire une fonction récursive `parcours_infixe(Arbre)`. On doit obtenir : `4, 2, 7, 5, 8, 1, 3, 9, 6`.
3. Écrire une fonction récursive `parcours_postfixe(Arbre)`. On doit obtenir : `4, 7, 8, 5, 2, 9, 6, 3, 1`.
4. Expliquer avec vos mots ce que fait chacun des trois parcours.

■

■ **Exercice 5.9.**

On souhaite implémenter le parcours en largeur d'un arbre binaire à l'aide d'une file.

1. Recopier la classe `File` du cours (structure FIFO) :

```
class File:
    def __init__(self):
        self.data = []

    def enfiler(self, elt):
        self.data.append(elt)

    def longueur(self):
        return len(self.data)

    def est_vide(self):
        return self.longueur() == 0

    def defiler(self):
        return self.data.pop(0)
```

2. Écrire une fonction `parcours_largeur(Arbre)` qui affiche les valeurs de l'arbre dans l'ordre du parcours en largeur.
3. Tester sur l'arbre de référence : on doit obtenir `1, 2, 3, 4, 5, 6, 7, 8, 9`.

■

■ **Exercice 5.10.**

On considère la fonction suivante :

```
def inserer(elt, Arbre=None):
    if Arbre == None:
        rep = Noeud()
        rep.valeur = elt
        return rep
    x = Arbre.valeur
    if elt < x:
        rep = Noeud()
        rep.valeur = x
        rep.gauche = inserer(elt, Arbre.gauche)
        rep.droite = Arbre.droite
        return rep
    else:
        rep = Noeud()
        rep.valeur = x
        rep.gauche = Arbre.gauche
        rep.droite = inserer(elt, Arbre.droite)
        return rep
```

1. Expliquer ce que fait cette fonction, et pourquoi.
2. Pourquoi recopie-t-elle les nœuds du chemin parcouru au lieu de modifier l'arbre donné ? Que devient l'arbre d'origine après l'appel ?
3. Que renvoie l'appel `inserer(8)` sans second argument ?

■

■ **Exercice 5.11.**
On considère la fonction suivante :

```
def rechercher(Arbre, element):
    if Arbre == None:
        return False
    else:
        x = Arbre.valeur
        if x == element:
            return True
        elif element < x:
            return rechercher(Arbre.gauche, element)
        else:
            return rechercher(Arbre.droite, element)
```

1. Expliquer comment fonctionne cette fonction.
2. Proposer un autre nom pour cette fonction, plus explicite.
3. Tester cette fonction pour plusieurs valeurs de `element` sur l'ABR du cours : 12 (attendu `True`), 1 (attendu `False`), 14 (attendu `True`), 7 (attendu `False`).
4. Combien de comparaisons faut-il pour rechercher 12 ? pour rechercher 14 ?

■

■ **Exercice 5.12.**
On construit un ABR en insérant successivement les clés de la liste `[7, 3, 9, 1, 5]` dans un arbre vide, avec la fonction `inserer` du cours.

1. Représenter l'arbre obtenu après les cinq insertions.
2. Donner l'ordre d'affichage du parcours en largeur de cet arbre.
3. Donner l'ordre d'affichage du parcours infixe de cet arbre. Que constate-t-on ?

■

Exercices de réflexion

Les exercices de réflexion demandent de concevoir des fonctions plus complexes, d'analyser des cas limites et de discuter des propriétés des ABR.

■ **Exercice 5.13.**
On rappelle qu'un ABR doit vérifier la propriété sur *tous* les descendants : tous les nœuds du sous-arbre gauche d'un nœud ont une clé inférieure, tous ceux du sous-arbre droit une clé supérieure. La fonction `est_de_recherche` du cours ne vérifie que les fils directs. On considère l'arbre suivant :

```
      10
     / \
```

```
  5 15
 / \
2 12
```

1. Cet arbre est-il un ABR ? Justifier (on s'intéressera au nœud 12).
2. Que renvoie la fonction `est_de_recherche` du cours sur cet arbre ? Expliquer pourquoi.
3. Écrire une fonction `est_de_recherche_correct(Arbre, min, max)` qui vérifie qu'un arbre est un ABR en s'assurant que chaque nœud a une valeur strictement comprise entre `min` et `max`. On pourra utiliser `None` pour représenter une borne absente, et l'appeler avec `est_de_recherche_correct(arbre, None, None)`.

■

■ **Exercice 5.14.**

On insère successivement les clés `[1, 2, 3, 4, 5]` dans un ABR vide avec la fonction `inserer`.

1. Représenter l'arbre obtenu. De quel type d'arbre s'agit-il (complet, parfait, dégénéré, équilibré) ?
2. Combien de comparaisons faut-il pour rechercher la clé 5 dans cet arbre ?
3. Représenter un ABR équilibré contenant les mêmes clés 1, 2, 3, 4, 5. Combien de comparaisons faut-il alors pour rechercher la clé 5 ?
4. Conclure sur l'influence de l'ordre d'insertion sur le coût de la recherche.

■

■ **Exercice 5.15.**

La fonction `inserer` du cours est **fonctionnelle** : elle renvoie un nouvel arbre et ne modifie pas l'arbre donné.

1. Écrire une version `inserer_mutation(elt, Arbre)` qui, cette fois, **modifie** l'arbre sur place : si l'arbre est vide, on le remplace par une nouvelle feuille ; sinon, on insère récursivement dans le sous-arbre gauche ou droit, sans créer de nouveau nœud pour le nœud courant.
2. Quel est l'intérêt de la version fonctionnelle du cours ? (On pourra s'intéresser à ce qui se passe si l'on veut garder l'ancienne version de l'arbre après une insertion.)

■

■ **Exercice 5.16.**

On souhaite compter les feuilles d'un arbre binaire.

1. Écrire une fonction récursive `nb_feuilles(Arbre)` qui renvoie le nombre de feuilles d'un arbre binaire. On précisera les cas d'arrêt.
2. Tester sur l'arbre de référence du cours (attendu : 4 feuilles, les nœuds 4, 7, 8 et 9) puis sur l'ABR du cours (attendu : 4 feuilles, les nœuds 0, 10, 12 et 14).

```
def nb_feuilles(Arbre):
    ...
```

■

■ **Exercice 5.17.**

On rappelle la propriété du cours : le parcours infixe d'un ABR affiche les clés dans l'ordre croissant.

1. Vérifier cette propriété sur l'ABR construit dans l'exercice 5.12 (insertions de `[7, 3, 9, 1, 5]`).
2. Expliquer, en vous appuyant sur la définition de l'ABR, pourquoi cette propriété est toujours vraie.
3. Application : on dispose d'un ABR contenant 1 000 000 de clés, parfaitement équilibré. Estimer le nombre de comparaisons nécessaires pour y rechercher une clé. Même question pour un ABR dégénéré.

■

■ **Exercice 5.18.**

Défi. On souhaite **supprimer** une clé dans un ABR. La suppression est plus délicate que l'insertion : trois cas se présentent selon que le nœud à supprimer est une feuille, a un seul enfant, ou a deux enfants (dans ce dernier cas, on le remplace par le plus petit nœud de son sous-arbre droit).

1. Traiter le cas d'une feuille : proposer une fonction qui renvoie l'arbre sans cette feuille.
2. Traiter le cas d'un nœud ayant un seul enfant.
3. **Question de recherche.** Traiter le cas d'un nœud ayant deux enfants : chercher comment remplacer le nœud supprimé par son « successeur » (le plus petit nœud du sous-arbre droit), puis écrire la fonction complète `supprimer(elt, Arbre)`.

■