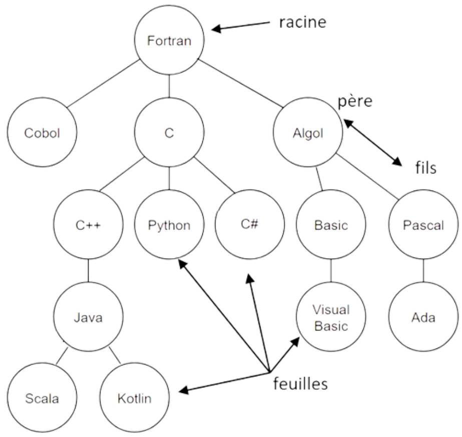
Structures hiérarchiques : arbres

Cours

Un **arbre** est un type abstrait de données constitué d'un ensemble de **nœuds**, reliés entre eux par des **arêtes** et organisés de manière **hiérarchique** :

- Un nœud particulier est la **racine**.
- Chaque nœud peut avoir **aucun, un ou plusieurs fils**. Les nœuds qui n'ont pas de fils sont appelés les **feuilles** de l'arbre, les autres (autre que la racine) sont des nœuds internes. L'**arité** d'un nœud est son nombre de fils.
- Chaque nœud a un **unique père**, à l'exception de la racine qui n'en n'a pas. Les nœuds qui ont le même père sont appelés des **frères**.
- Le **chemin à la racine** d'un nœud est la liste des nœuds qu'il faut parcourir depuis la racine jusqu'au nœud considéré.
- L'**étiquette** est la valeur donnée à chaque nœud (ici les noms de langages informatiques).



Les arbres trouvent de nombreuses applications en informatique, par exemple dans une arborescence de fichiers ou pour des stratégies de jeux.

Cours

- La **taille** d'un arbre est son **nombre de nœuds**.
- La **profondeur** d'un nœud est le **nombre de noeuds, ou niveaux, entre la racine le nœud**. La **profondeur de la racine est donc 1**.
- La **hauteur** d'un arbre est la plus **grande profondeur d'une feuille de l'arbre**. **Un arbre réduit à la racine a une hauteur de 1, un arbre vide a une hauteur de 0 (par convention)**.

⚠️ Il n'existe pas de définition universelle pour la hauteur d'un arbre et la profondeur d'un nœud dans un arbre. Dans certains cas la profondeur est le nombre d'**arêtes** entre la racine le nœud, la **hauteur de l'arbre réduit à la racine est alors de 0 et la hauteur de l'arbre vide est -1**[6].

Interface

Les principales primitives constituant l'interface d'un arbre sont :

- `creer() → arbre` : construire un arbre vide.
- `est_vide() → bool` : vérifier si un arbre est vide ou non.
- `taille() → int` : renvoyer la taille d'un arbre.

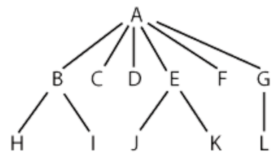
- `hauteur() → int` : renvoyer la hauteur d'un arbre.
- `profondeur(nœud) → int` : renvoyer la profondeur d'un nœud.
- `est_feuille(noeud) → bool` : vérifier si un nœud est une feuille ou pas.
- `branche(noeud) → arbre` : renvoyer un sous-arbre de racine nœud.

Python ne propose pas de façon native l'implémentation des arbres. Ils peuvent être implémentés de plusieurs façons.

Implémentation avec un dictionnaire

Une première implémentation est d'utiliser un dictionnaire dont les clés sont les étiquettes des nœuds et les valeurs des tableaux de fils.

```
arbre = {'A':['B', 'C', 'D', 'E', 'F', 'G'], 'B':['H','I'],
         'C':[], 'D':[], 'E':['J','K'], 'F':[], 'G':['L'],
         'H':[''], 'I':[], 'J':[], 'K':[], 'L':[]
         }
```



Les primitives s'écrivent :

```
def creer():
    return {}

def est_vide(arbre):
    return len(arbre) == 0

def taille(arbre):
    return len(arbre)

def ajouter_noeud(arbre, fils, pere = None):
    if fils not in arbre:
        arbre[fils] = []      # ajoute le noeud fils à l'arbre
        if pere is not None:
            arbre[pere].append(fils)   # et dans la liste des fils du pere


def est_feuille(arbre, noeud):
    """ True si n est une feuille, False sinon"""
    return arbre[noeud] == []       # si noeud n'a pas de fils
```

puis pour créer notre arbre :

```
a = creer()
ajouter_noeud(a, 'A')
for fils in ['B', 'C', 'D', 'E', 'F', 'G']:
    ajouter_noeud(a, fils, 'A')      # les fils de 'A'
for fils in ['H', 'I']:
    ajouter_noeud(a, fils, 'B')      # les fils de 'B'
for fils in ['J', 'K']:
    ajouter_noeud(a, fils, 'E')      # les fils de 'E'
ajouter_noeud(a, 'L', 'G')
```

Il est aussi possible de rajouter quelques primitives de profondeur, hauteur, etc. :

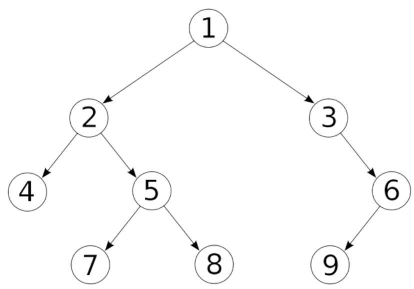
```python
def pere(arbre, noeud):
    """ Renvoie le pere de noeud"""
    for n in arbre:
        if noeud in arbre[n]:       # si noeud est un fils de n
            return n
    return None     # n est la racine, le pere est None



def profondeur(arbre, noeud):
    """ Renvoie la profondeur de noeud"""
    p = 1
    while pere(arbre, noeud) is not None:    # tant qu'on n'est pas à la racine
        p = p + 1           # on ajoute 1 à p
        noeud = pere(arbre, noeud)  # on remplace noeud par son pere
    return p


# ou en récursif
def profondeur_rec(arbre, noeud):
    """ Renvoie la profondeur de n"""
    if pere(arbre, noeud) is None: return 1    # la profondeur de la racine est 0
    return 1 + profondeur_rec(arbre, pere(arbre, noeud))


def hauteur(arbre):
    """ Renvoie la hauteur de l'arbre"""
    p_max = 0
    for n in arbre:
        if profondeur(arbre, n) > p_max:
            p_max = profondeur(arbre, n)
    return p_max
```

Arbre binaire

Cours

Un **arbre binaire** (**AB**) est un cas particuliers d'arbre où **chaque nœud possède au maximum deux fils ordonnés** : un fils **gauche** et/ou un fils **droit**.

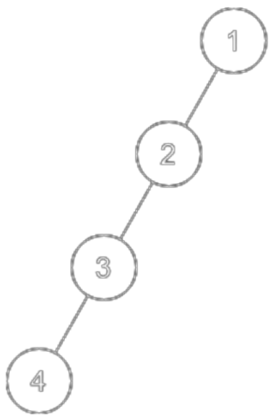
Les fils gauche et droit ne sont pas intervertibles !



Il est possible d'avoir des arbres binaires de même taille mais de « forme » très différente :

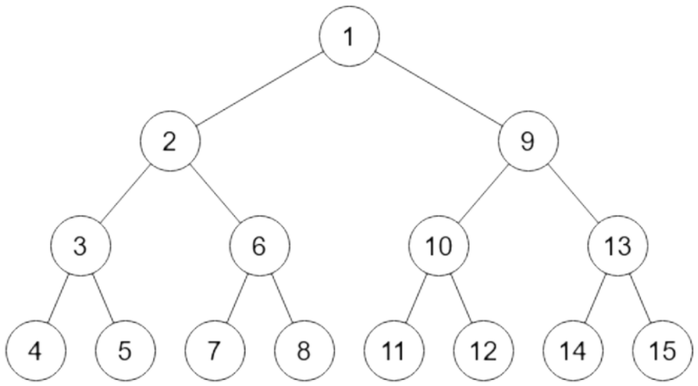
Arbre binaire filiforme (ou dégénéré)

Tous ses nœuds possèdent un unique fils (on parle aussi de peigne).



Arbre binaire parfait

Tous ses nœuds possèdent exactement 2 fils (sauf les feuilles qui en ont zéro !).



Il en résulte certaines propriétés sur la taille n et la hauteur h d'un arbre binaire :

- Un arbre filiforme de taille n a une hauteur h égale à n, c'est la plus grande hauteur possible donc pour tout AB : $h ≤ n $.
- On peut aussi montrer[7] qu'un arbre binaire parfait de hauteur h a une taille n égale à $2^h - 1$, c'est la plus grande taille possible donc pour tout AB : $n \leq 2^h - 1$. On en déduit que $\log_2(n+1) \leq h$ où $\log_2(n+1)$ est le logarithme en base 2 de $n+1$[8].

Cours

La taille d'un arbre binaire quelconque de hauteur est encadrée par :

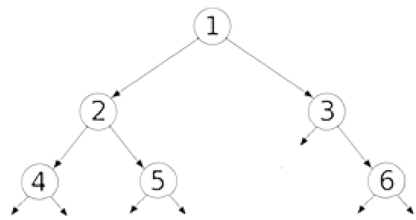
Réciproquement, la hauteur d'un arbre binaire de taille est encadrée par :

Implémentation avec des p-uplets imbriqués

Les arbres binaires ont au plus deux fils, il est donc possible d'utiliser des triplés imbriqués contenant pour chaque nœud : sa valeur, son fils de gauche et son fils de droite (dans cet ordre).

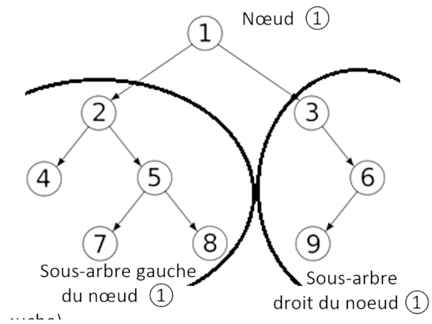
```
>>> n4 = (4, (), ())
>>> n5 = (5, (), ())
```

```
>>> n2 = (2, n4, n5)
>>> n6 = (6, (), ())
>>> n3 = (3, (), n6)
>>> n1 = (1, n2, n3)
>>> n1
(1, (2, (4, (), ()), (5, (), ())), (3, (), (6, (), ())))
```



Implémentation récursive

Notons qu'un arbre binaire peut-être est défini de façon récursive comme étant :

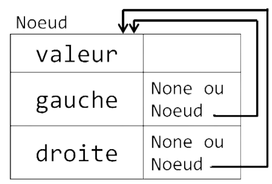


- soit un arbre vide,
- soit composé d'un nœud racine avec une valeur, un sous-arbre gauche et un sous-arbre droit.

Les sous-arbres de gauche et droite sont aussi des arbres (autrement dit des nœuds avec deux sous-arbres, etc…).

Implémentons sur ce modèle un arbre binaire par une classe `Noeud`, récursive, possédant trois attributs :

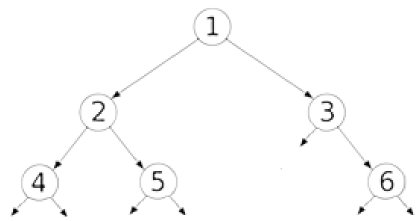
- `valeur` pour l'étiquette du noeud ;
- `gauche`, le sous-arbre gauche, c'est une instance de `Noeud` (ou `None` si le nœud n'a pas de fils gauche) ;
- `droite`, le sous-arbre droit, c'est une instance de `Noeud` (ou `None` si le nœud n'a pas de fils droit).




```python
class Noeud:
    def __init__(self, v, g=None, d=None):
        self.valeur = v
        self.gauche = g   # None ou un Noeud
        self.droite = d   # None ou un Noeud
```

et créons un arbre non vide (un arbre vide est `None`) :

```python
n4 = Noeud(4)
n5 = Noeud(5)
n2 = Noeud(2, n4, n5)
n6 = Noeud(6)
n3 = Noeud(3, d=n6)
a = Noeud(1, n2, n3)
```



ce qui peut aussi s'écrire directement :

```python
a = Noeud(1, Noeud(2, Noeud(4), Noeud(5)), Noeud(3, None, Noeud(6)))
```

Ajoutons une première méthode pour vérifier si un nœud est une feuille :

```python
    def est_feuille(self) -> bool:
        return self.gauche is None and self.droite is None
```

puis la taille et la hauteur de l'arbre :

```
    def taille(self):
        """ Renvoie la taille (le nombre de noeud) de l'arbre"""
        # taille du sous-arbre droit
        if self.droite is None:
            td = 0
        else:
            td = self.droite.taille()

        # taille du sous-arbre gauche
        if self.gauche is None:
            tg = 0
        else:
            tg = self.gauche.taille()

        # 1 (pour le noeud self) + taille à gauche + la taille à droite
        return 1 + td + tg

    def hauteur(self):
        """ Renvoie la hauteur (la plus grande profondeur) de l'arbre,
        Par convention, la taille d'un arbre réduit à la racine est 1, celle de l'arbre vide est
0"""

        # hauteur du sous-arbre droit
        if self.droite is None:
            hd = 0
        else:
            hd = self.droite.hauteur()

        # hauteur du sous-arbre gauche
        if self.gauche is None:
            hg = 0
        else:
            hg = self.gauche.hauteur()

        # 1 (pour le noeud self) + la plus grande taille entre gauche et droite
        return 1 + max(hd , hg)
```

On retrouve cette implémentation avec une classe `Noeud` dans la plupart des exercices de baccalauréat[1], parfois la nommant `arbre`, ou `AB`. Néanmoins on peut lui reprocher de ne pas représenter correctement la définition proposée d'un arbre, puisque les arbres et sous-arbres vides sont représentés par `None`, ce qui n'est pas une instance de cette classe !

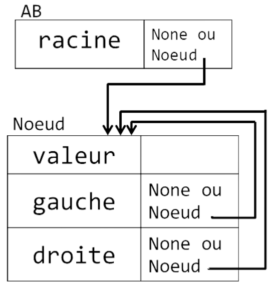
En plus de nous limiter aux arbres non-vides (dit « enracinés ») et d'imposer des manipulations pénibles dans le code pour vérifier si un sous-arbre est `None` ou pas (comme ici `if self.droite is None: ...`), cela engendre beaucoup d'erreurs de programmation (utilisations erronées des méthodes de la classe `Noeud` sur `None`).

⚠️ Pour tenter de remédier à ces défauts, on trouve plusieurs variantes d'implémentation, plus ou moins satisfaisantes.

Une première variante consiste à ajouter une classe d'arbre qui pointe sur `None` quand l'arbre est vide et sur un `Noeud` racine quand l'arbre est enraciné[2] (sur le même modèle des listes chainées utilisant les classes `Cellules` et `ListeChainees`).

> La structure récursive est la classe `Noeud`, pas la classe `AB` !

Ajoutons à notre structure cette classe `AB` avec un attribut racine qui est de type `Noeud` ou `None` pour un arbre vide.




```
class AB:
    def __init__(self, racine=None):
        self.racine = racine     # None ou un Noeud
```

Il est maintenant possible d'implémenter un arbre vide comme un objet de la classe `AB` :

```
arbre = AB()
```

ou un arbre complet :

```
arbre = AB(Noeud(1, Noeud(2, Noeud(4), Noeud(5, Noeud(7), Noeud(8))), Noeud(3, None, Noeud(6,
Noeud(9)))))
```

Ajoutons les primitives d'un arbre binaire :

```
class AB:
    def est_vide(self):
        return self.racine == None

    def hauteur(self):
        if self.racine is None: return 0
        # renvoie la hauteur du nœud racine
        return self.racine.hauteur()

    def taille(self):
        if self.racine is None: return 0
        # renvoie la taille du nœud racine
        return self.racine.taille()
```

Une seconde variante que l'on rencontre parfois[3] consiste à ne garder qu'une seule classe `Noeud` et en représentant un arbre vide avec une instance dont l'attribut `self.valeur` prend la valeur `None`.

```
class Noeud:
    def __init__(self, v=None, g=None, d=None):
        self.valeur = v   # None pour un arbre vide
        self.gauche = g   # None ou un Noeud
        self.droite = d   # None ou un Noeud
```

On peut alors simplement créer un arbre vide :

```
arbre_vide = Noeud()
```

Ici, l'objet `arbre_vide` est représenté par un noeud à part entière. On peut utiliser les méthodes de la classe `Noeud` pour calculer la hauteur et la taille :

```
>>> arbre_vide.taille()
1
```

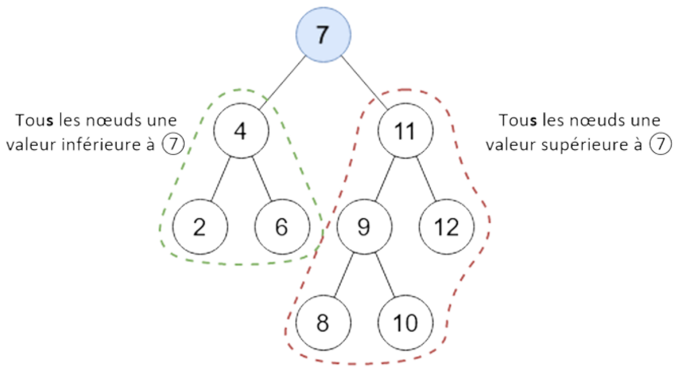
L'arbre vide comporte un noeud, la méthode renvoie une taille 1 au lieu de 0 ! De même `.hauteur()` renvoie 1 au lieu de 0. Il faut donc modifier les deux méthodes en conséquence.

Arbres binaires de recherche

Cours

Un **arbre binaire de recherche** (**ABR**) est un cas particulier d'arbre binaire sans lequel :

- Chaque nœud a une valeur (ou clé) supérieure à celles de tous les nœuds de son sous-arbre gauche.
- Chaque nœud a une valeur inférieure à celles de tous les nœuds de son sous-arbre droit.
- Tous les sous-arbres sont aussi des ABR.



Note : « supérieur » et « inférieur » peuvent être au sens strict ou large en fonction de la définition donnée.

Considèrons l'arbre binaire de recherche précédent qui servira comme support pour illustrer la suite.

Plutôt que de dupliquer la classe `AB` précédente en `ABR` et de la modifier, nous allons créer une sous-classe par héritage[4] et lui ajouter les spécificités d'un ABR.

Inutile de réécrire le constructeur :

```
class ABR(AB):
    pass

a = ABR(Noeud(7, Noeud(4, Noeud(2), Noeud(6)), Noeud(11, Noeud(9, Noeud(8), Noeud(10)),
Noeud(12))))
```

Toutes les méthodes de la classe `AB` fonctionnent par héritage pour un objet de la classe `ABR` :

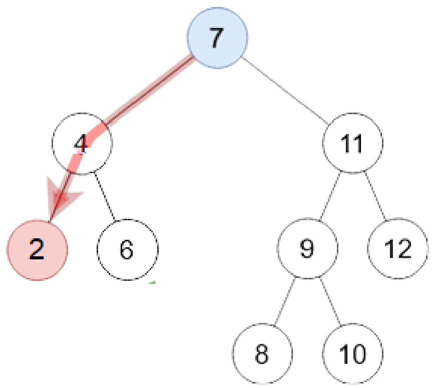
```
>>> a.taille()
9
>>> a.hauteur()
3
```

Ajoutons des méthodes propres aux ABR :

Clés min et max

Pour accéder à la plus petite clé d'un ABR, il suffit de descendre sur les fils à gauche autant que possible. Le dernier nœud visité qui n'a pas de fils gauche porte la plus petite valeur de l'ABR. De même façon, pour trouver la plus grande valeur, il suffit de descendre sur les fils à droite.

La classe `ABR` n'étant pas récursive, il faut définir une **méthode récursive** au niveau de la classe `Noeud` qui descend le plus à gauche[5] :




```
class Noeud:
    def desc_g(self):
```

```
        ''' renvoie la feuille la plus à gauche'''
        if self.gauche is None: return self.valeur
        return self.gauche.desc_gauche()
```

puis renvoyer sa valeur dans la classe ABR pour obtenir le min d'un arbre :

```
 class ABR :
   def min(self):
        """ Renvoie la plus petite valeur de l'arbre """
        if self.racine is None: return None
        return self.racine.desc_g().valeur
```

Vérifier que l'arbre est un ABR (hors programme)

Pour vérifier qu'un arbre est un ABR, il faut vérifier que :

- La clé de chaque nœud est plus grande que le max de son sous-arbre de gauche, et plus petite que le min de son sous-arbre de droite.
- Les sous-arbres de droites et de gauches sont des ABR.

Implémentons cette vérification de façon récursive au niveau de la classe `Nœud` :

```
class Noeud:

    def verif_noeud(self):
        ''' vérifie que le sous-arbre de racine noeud est un ABR'''
        if self.gauche is None: g = True
        else:
            g = self.valeur > self.gauche.desc_d().valeur and self.gauche.verif_noeud()
        if self.droite is None: d = True
        else:
            d = self.valeur < self.droite.desc_g().valeur and self.droite.verif_noeud()
        return g and d
```

Rajoutons une méthode au niveau de la classe `ABR` :

```
    def verif_ABR(self):
        """ Renvoie True si self est bien un ABR """
        if self.racine is None: return True
        return self.racine.verif_noeud()
```

---

1. On trouve des arbres implémentés par une seule classe récursive dans les sujets 21-NSIJ1ME1, 22-NSIJ2ME1, 22-NSIJ1LR1, 23-NSIJ2AS1 23-NSIJ2LI1, 23-NSIJ2LR1, 23-NSIJ2ME1, 23-NSIJ2PO1, 23-sujet_0-b ↩
2. On trouve un arbre implémenté par une classe `Noeud` récursive et une classe `Arbre` dans les sujets 21-NSIJ2ME2 ↩
3. On trouve un arbre implémenté par une classe `Noeud` dont l'attribut `valeur` d'un arbre vide est égal à `None` dans le sujet 21-NSIJ2PO1. Par ailleurs, le sujet https://e-nsi.gitlab.io/pratique/N2/800-arbre_bin/sujet/ montre un exemple de ce type d'implémentation complétement récursif. ↩
4. L'héritage est un des grands principes de la programmation orientée objet (POO) permettant de créer une nouvelle classe à partir d'une classe existante. La sous classe hérite des attributs et des méthodes de la classe mère et en ajoute de nouveaux. ↩
5. On peut aussi définir une fonction récursive directement dans la méthode `min()` de la classe `ABR`.

```
class ABR:
    def min(self):
        """ Renvoie la plus petite valeur de l'arbre """
        if self.racine is None: return None

        def desc_g(n):
            while n.gauche is not None:
                n = desc_g(n.gauche)
            return n

        return desc_g(self.racine).valeur
```

↩

6. Un arbre vide a une hauteur de 0 et un arbre réduit à la racine une hauteur de 1 dans la plupart des sujets de bac (21-Sujet_0, 21-NSIJ2ME1, 21-NSIJ2ME2, 22-NSIJ1AS1, 22-NSIJ1JAN1, 22-NSIJ1PO1, 22-NSIJ2JA1, 22-NSIJ2ME1, 23-NSIJ2G11 et 23-NSIJ2LI1) sauf les sujets 22-NSIJ1NC1, 24-NSI-41 et 24-NSIJ2PO1 où un arbre vide a une hauteur de -1 et un arbre réduit à la racine une hauteur de 0. ↩
7. Par récurrence la taille d'un arbre racine est , et si la taille d'un arbre parfait de hauteur est , pour obtenir la taille de l'arbre parfait de hauteur il faut ajouter nouveaux nœuds, au total on obtient . ↩
8. Le logarithme en base 2, noté est une opération mathématique qui calcule la puissance à laquelle il faut élever le nombre 2 pour obtenir un nombre donné. Par exemple est car . ↩