Algorithmes, Programmation, IA

Cours 2

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/algo-prog-ia-25

Plan

- alias, notation compréhensive
- classes et objets
- parcours d'arbres
- fonctions et méthodes (programmation impérative ou orientée-objet)
- programmation fonctionnelle (avec données persistantes)
- programmation impérative
- arbres de syntaxe abstraites
- belle impression

dès maintenant: télécharger Python 3 en `http://www.python.org`

Quelques rappels

- le cours utilise le langage Python et l'environnement Visual Studio Code. (`vscode`)
- et pour la partie IA, la bibliothèque Pytorch

Impressions formattées

- impression

```
def concat_print0 (s1, s2):
    print (s1 + " " + s2)

concat_print0 ("Hello", "World !")
Hello World !


def concat_print1 (s, n, x):
    print ("%s vaut %d ou %.2f" %(s, n, x))

concat_print1 ("Le resultat", 32, 28.5)
Le resultat vaut 32 ou 28.50


def concat_print2 (s, n, x):
    print ("{} vaut {} ou {}".format (s, n, x))

concat_print2 ("Le resultat", 32, 28.5)
Le resultat vaut 32 ou 28.5
```

- voir en `www.programiz.com/python-programming/input-output-import`

Ensembles — Compréhension

- les ensembles sont des listes non ordonnées d'éléments tous distincts

```
print ({10, 2, 3} == {3, 2, 10})
True

print ({10, 2, 3} == {5, 2, 10})
False
```

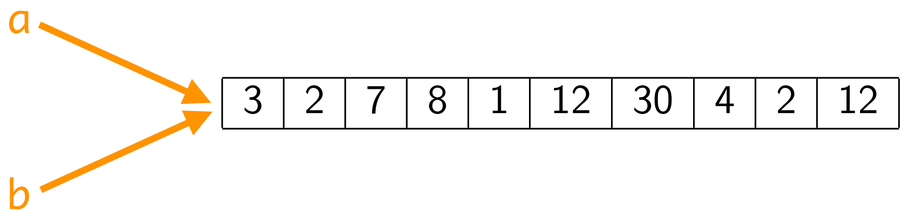
- on peut générer des tableaux, listes, ensembles, dictionnaires avec la notation compréhensive

```
a = [x**2 for x in range(21) if x*2 < 21]
print (a)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

b = {2 * x for x in range (20)}
print (b)
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38}

c = {x : x**2 for x in range (10)}
print (c)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Valeur d'un tableau — Alias

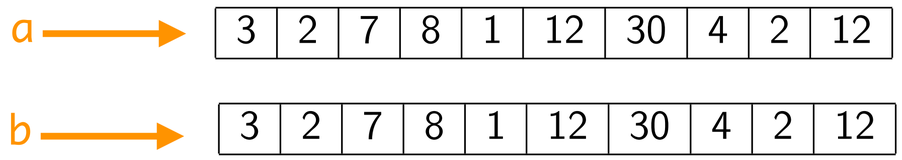
- soient 2 tableaux a et b

```
a = [3, 2, 7, 8, 1, 12, 30, 4. 2. 12]
b = a

a[2] = 888

print (a)
[3, 2, 888, 8, 1, 12, 30, 4. 2. 12]

print (b)
[3, 2, 888, 8, 1, 12, 30, 4. 2. 12]
```



- les variables a et b sont 2 alias d'un même tableau

```
a = [3, 2, 7, 8, 1, 12, 30, 4. 2. 12]
b = [3, 2, 7, 8, 1, 12, 30, 4. 2. 12]

a[2] = 888

print (a)
[3, 2, 888, 8, 1, 12, 30, 4. 2. 12]

print (b)
[3, 2, 7, 8, 1, 12, 30, 4. 2. 12]
```



- les variables a et b sont 2 tableaux distincts

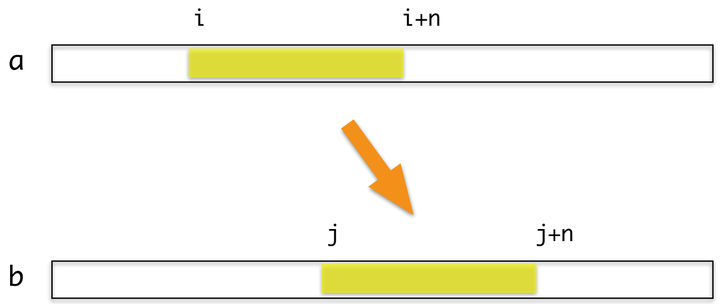
- la valeur de a ou de b est l'adresse mémoire de son premier élément

données modifiables et alias sont sources de bugs

Valeur d'un tableau — Alias

Exercice: le programme suivant est-il correct?

```
def copy (a, b, i, j, n) :
    for k in range (n) :
        b [j + k] = a [i + k]
```



Solution: le programme est correct, même quand les paramètres *a* et *b* sont des alias.

```
def copy (a, b, i, j, n) :
        if i > j :
            for k in range (n) :
                b [ j + k ] = a [i + k]
        else :
            for k in range (n-1, -1, -1) :
                b [ j + k ] = a [ i + k]
```

Classes et objets

• une classe décrit un ensemble d'objets tous de la même forme avec attributs et méthodes

```
class Point:
    def __init__ (self, x, y) :
        self.x = x
        self.y = y

    def __str__ (self) :
        return "(%d, %d)" %(self.x, self.y)

    def __add__ (self, delta) :
        return Point (self.x + delta.x, self.y + delta.y)
```

constructeur d'un nouvel objet

`__str__` est appelé par `print`

`__add__` est appelé par `+`

• objets dans cette classe

```
p1 = Point (10, 20)
print (p1)
(10, 20)

p2 = Point (40, 50)
print (p2)
(40, 50)

p3 = p1 + p2
print (p3)
(50, 70)
```

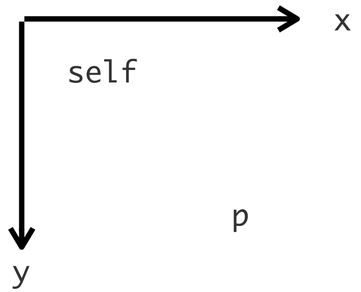
nouvel objet de la classe `Point`

Classes et objets

- les attributs d'un objet ont des valeurs quelconques (par exemple des références à d'autres objets)

```
class Point:
    # comme avant

    def __le__ (self, p) :
        return self.x <= p.x and self.y <= p.y
```

`__le__` est appelé par `<=`



- le constructeur de la classe `Rectangle` utilise des objets `Point`

```
class Rectangle:
    def __init__ (self, p, q) :
        self.haut_gauche = p
        self.bas_droite = q
        if not p <= q :
            raise ValueError

    def __str__(self) :
        return "Rectangle ({}, {})".format (self.haut_gauche, self.bas_droite)
```

on vérifie que q est dans le quadrant inférieur droit

```
r = Rectangle (p1, p3)
Rectangle (10, 20), (40, 50)
print (r)
```

Classes et héritage

- une classe peut être une sous-classe d'une classe plus générale

```
class Carre (Rectangle) :
    def __init__ (self, p, c) :
        super().__init__ (p, p + Point(c, c))
```

on appelle le constructeur de `Rectangle`

- un carré est un rectangle particulier

- le constructeur de `Carre` appelle le constructeur de la super classe `Rectangle`

Exercice écrire la classe `Polygone` qui construit des objets à partir d'une liste de `Point`

Exercice écrire la classe `Triangle`

Exercice écrire les méthodes `perimetre` et `surface`

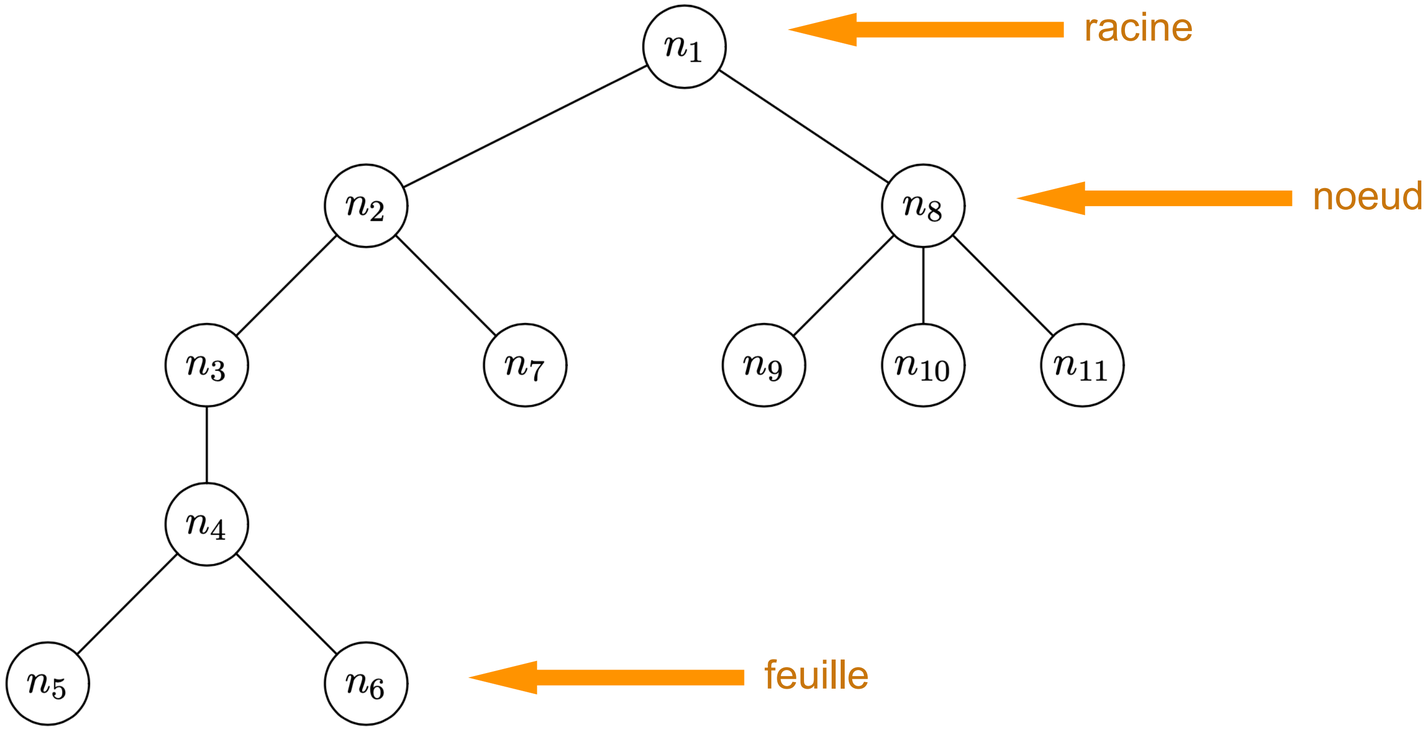
Classes et objets

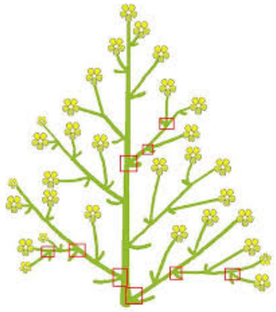
- les classes permettent d'**encapsuler** un ensemble de données (attributs) et de fonctions (méthodes)
- les classes représentent donc une forme de **modularité**
- à l'extérieur, le détail de l'implémentation des objets de cette classe est **opaque**
- on peut donc modifier la représentation sans changer leur interface et les fonctions qui utilisent cette classe
- **attention**: classes et modules sont deux notions différentes en Python
  [les modules sont importés et attachés à la notion de fichier]

les sources sont en `http://jeanjacqueslevy.net/prog-py-22/progs/c7a.py`

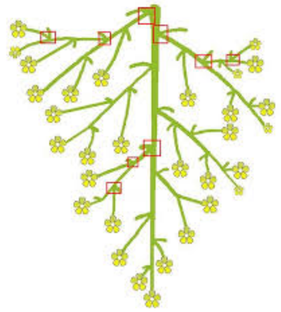
Les arbres en informatique

• les arbres sont une structure de données de base en informatique





botanique



informatique

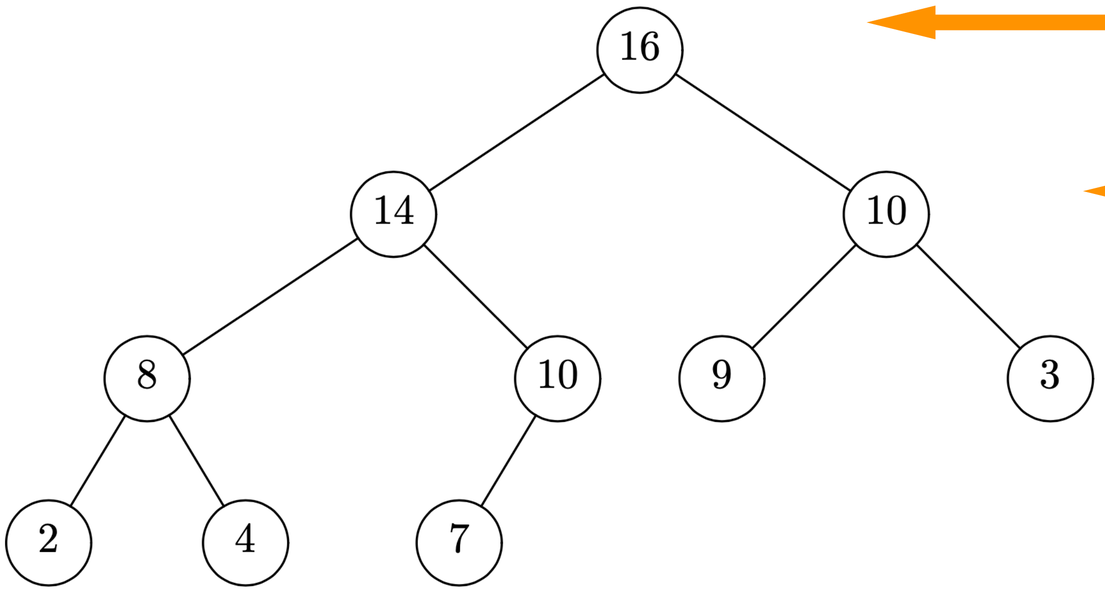
n2 est un **ancêtre** de n4

n3 et n7 sont des **fils** de n2

la **hauteur** d'un arbre est la longueur du plus long chemin de la racine à une feuille

Les arbres en informatique

- les noeuds et feuilles peuvent être étiquetés par des nombres entiers



maximum

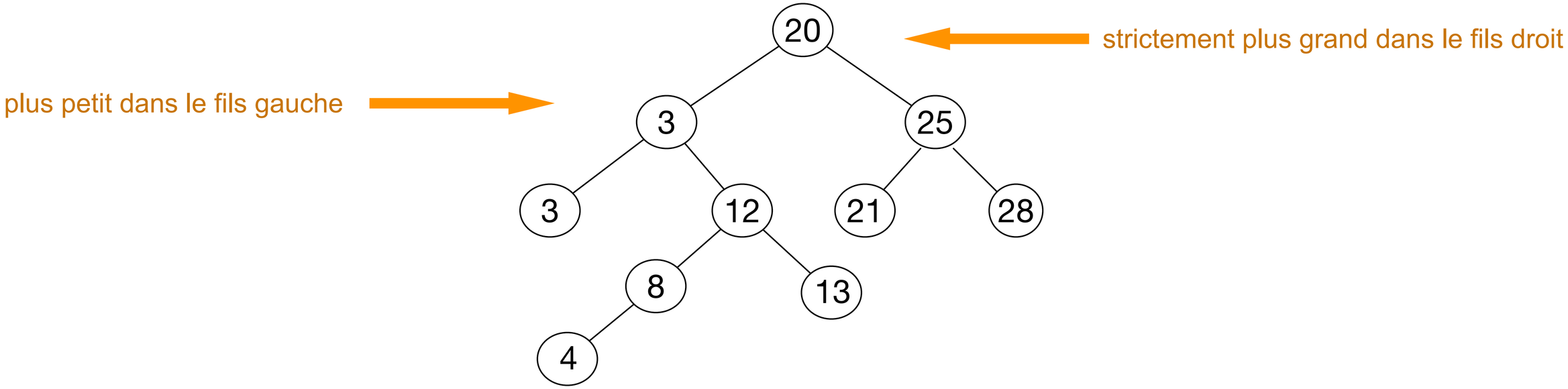
arbre binaire

tout noeud a au plus 2 fils

[arbre représentant une file de priorité]

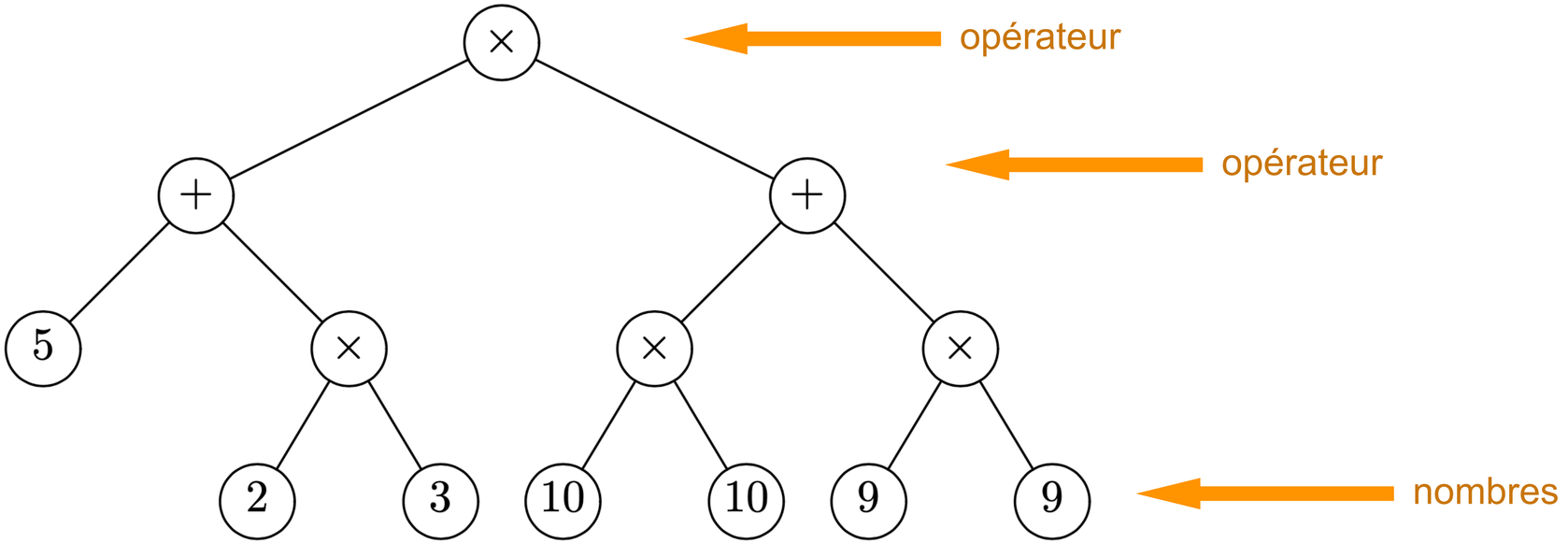
Les arbres en informatique

- arbres binaires de rechercher (*BST*)



Les arbres en informatique

• les noeuds et feuilles peuvent être étiquetés par des valeurs quelconques [ici des chaînes de caractères]



pour représenter une **expression arithmétique**

$$(5 + 2 \times 3) \times (10 \times 10 + 9 \times 9)$$

[plus besoin de parenthèses]

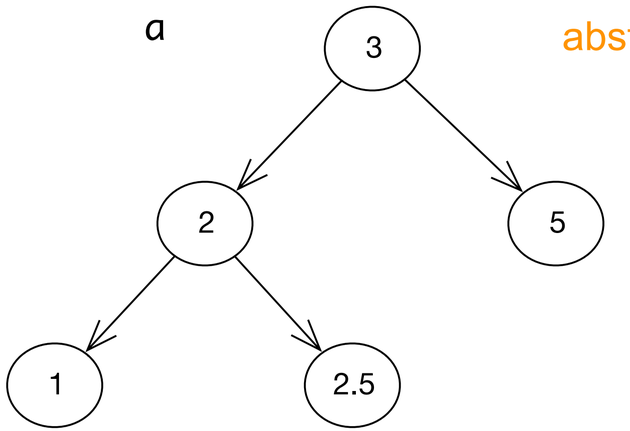
Représentation des arbres

- on définit une classe pour les noeuds et pour les feuilles

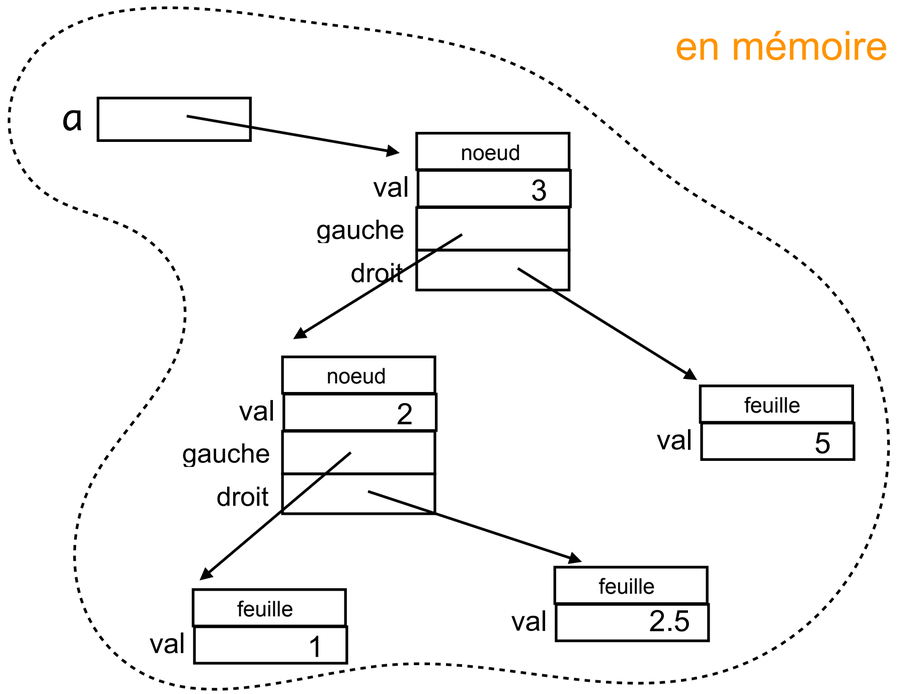
```
class Noeud:
    def __init__ (self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d

class Feuille:
    def __init__ (self, x) :
        self.val = x
```

- et on construit des arbres

```
a = Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))
```



abstraction



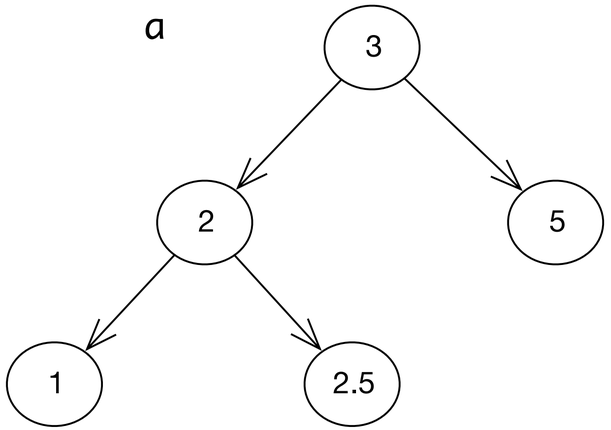
Représentation des arbres

- on définit une méthode pour l'impression des noeuds et des feuilles

```
class Noeud:
    # ...

    def __str__ (self) :
        return "Noeud ({}, {}, {})".format (self.val, self.gauche, self.droit)


class Feuille:
    # ...

    def __str__ (self) :
        return "Feuille ({})".format (self.val)
```

- on construit et imprime des arbres

```
a = Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))
print (a)
Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))

print (a.droit)
Feuille (5)

print (a.gauche)
Noeud (2, Feuille (1), Feuille (2.5))
```

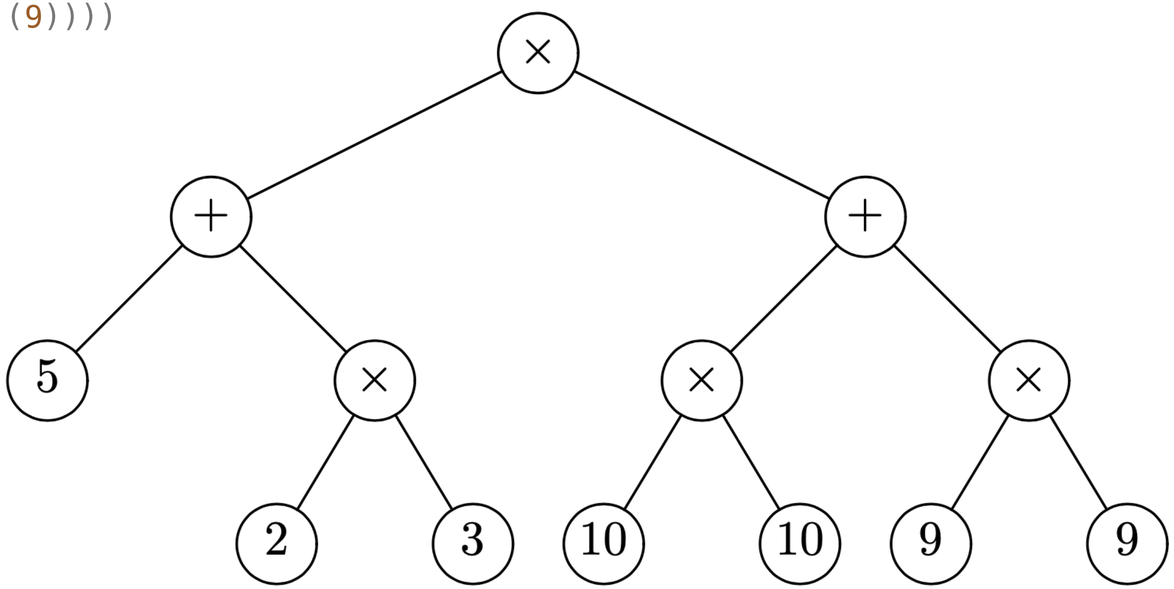


Représentation des arbres

- on construit et imprime des arbres

```
b = Noeud ('*', Noeud ('+', Feuille (5),
                            Noeud ('*', Feuille (2), Feuille (3))),
                Noeud ('+', Noeud ('*', Feuille (10), Feuille (10)),
                            Noeud ('*', Feuille (9), Feuille (9))))
```

```
print (b.gauche.gauche)
Feuille (5)
print (b.gauche.droit)
Noeud ('*', Feuille (2), Feuille (3))
```

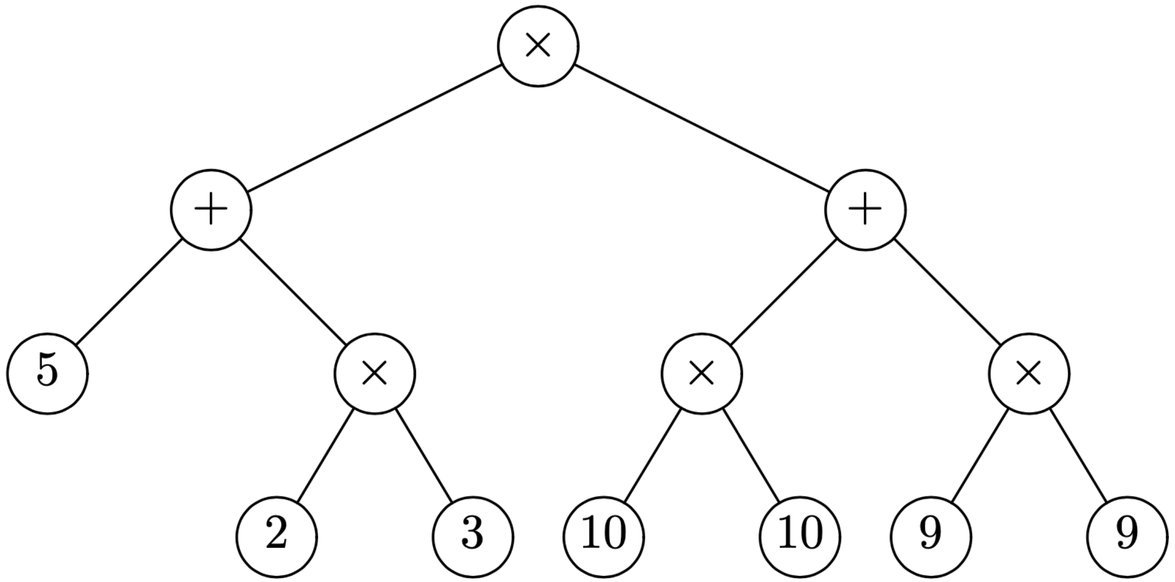


Fonctions sur les arbres

induction structurelle

- On parcourt ou calcule sur les arbres avec des fonctions récursives

```
def hauteur (a) :
    if isinstance (a, Feuille) :
        return 0
    else :
        return 1 + max (hauteur (a.gauche), hauteur (a.droit))

def taille (a) :
    if isinstance (a, Feuille) :
        return 1
    else :
        return 1 + taille (a.gauche) + taille (a.droit)
```



- et on calcule les hauteur et taille

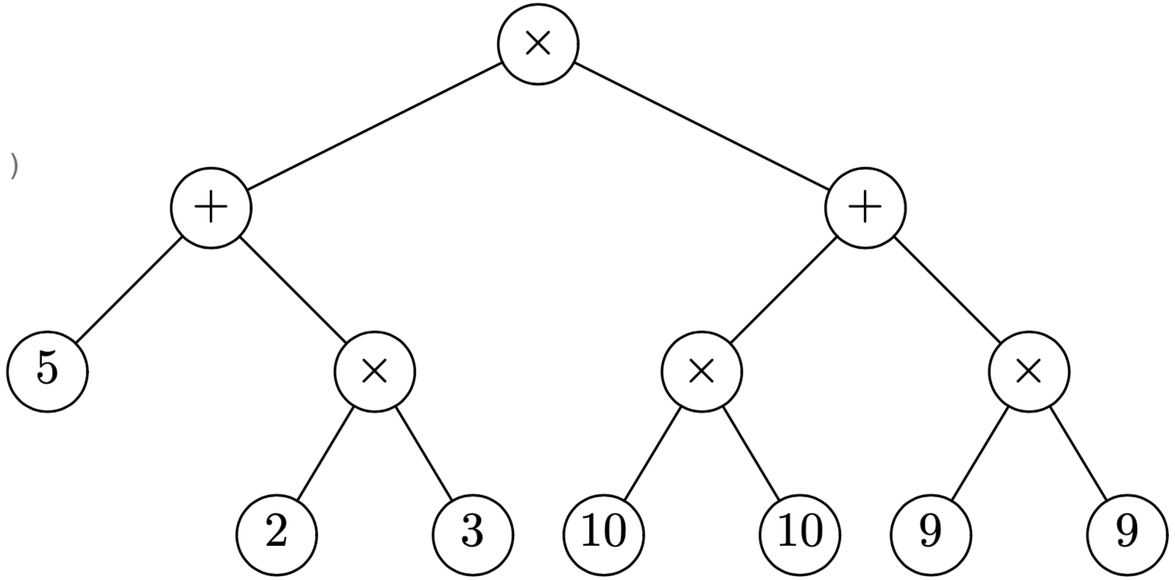
```
print (b)
Noeud (*, Noeud (+, Feuille (5), Noeud (*, Feuille (2), Feuille (3))), Noeud (+, Noeud (*, Feuille (10),
Feuille (10)), Noeud (*, Feuille (9), Feuille (9))))

hauteur (b)
3

taille (b)
13
```

Fonctions sur les arbres

par cas sur sous-classes

- On parcourt ou calcule sur les arbres avec des méthodes

```
class Noeud:
    # comme avant et en plus :
    def hauteur (self) :
        return 1 + max (self.gauche.hauteur(), self.droit.hauteur())

    def taille (self) :
        return 1 + a.gauche.taille() + a.droit.taille()

class Feuille:
    # comme avant et en plus :
    def hauteur (self) :
        return 0

    def taille (self) :
        return 1
```



- et on calcule les hauteur et taille

```
print (b.hauteur())
3

print (b.taille())
13
```

Procédures ou Méthodes

- programmation procédurale
  - on regroupe les opérations à l'intérieur du corps de la fonction
  - on fonctionne par induction structurelle
  - si on modifie la classe, on doit changer toutes les fonctions

contrôle par les procédures

- programmation orientée-objet. (*OO programming*)
  - chaque classe a une méthode spécifique
  - l'objet applique la méthode de sa classe
  - si on modifie la méthode, on doit changer la même méthode dans toutes les classes

contrôle par les données

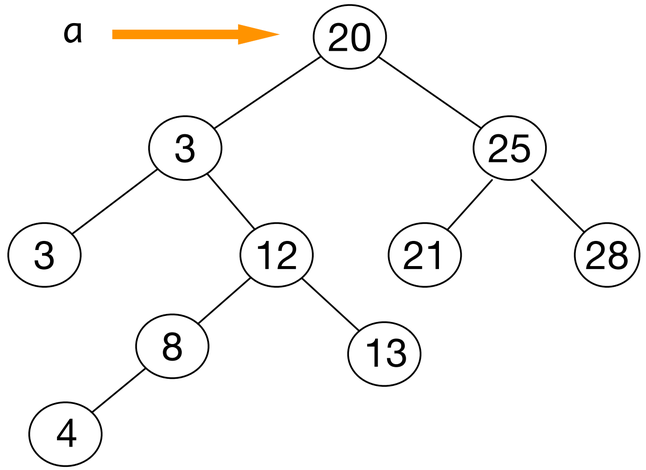
Arbres binaires de recherche

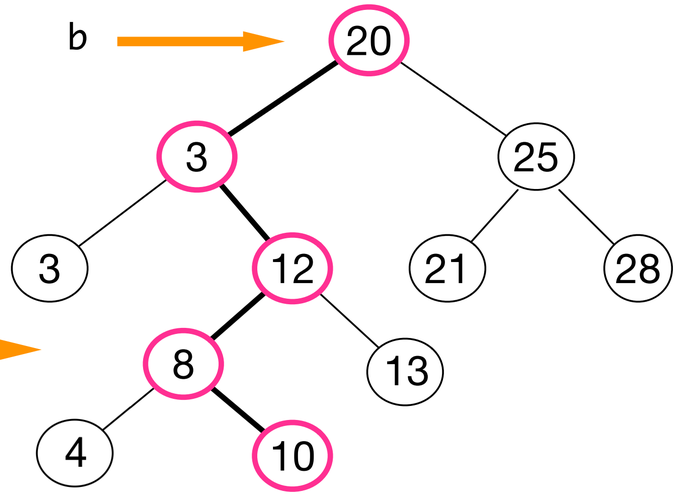
- ajouter une clé (style: **programmation fonctionnelle**)

programme plus simple avec un
seul type de noeud

```
def ajouter (x, a) :
    if a == None :
        return Noeud (x, None, None)
    elif x <= a.val :
        return Noeud (a.val, ajouter (x, a.gauche), a.droit)
    else :
        return Noeud (a.val, a.gauche, ajouter (x, a.droit))
```

```
b = ajouter (10, a)
```





on ne modifie pas l'arbre `a`, les
noeuds **rouges** sont nouveaux

Arbres binaires de recherche

• ajouter une clé (style: **programmation impérative**)

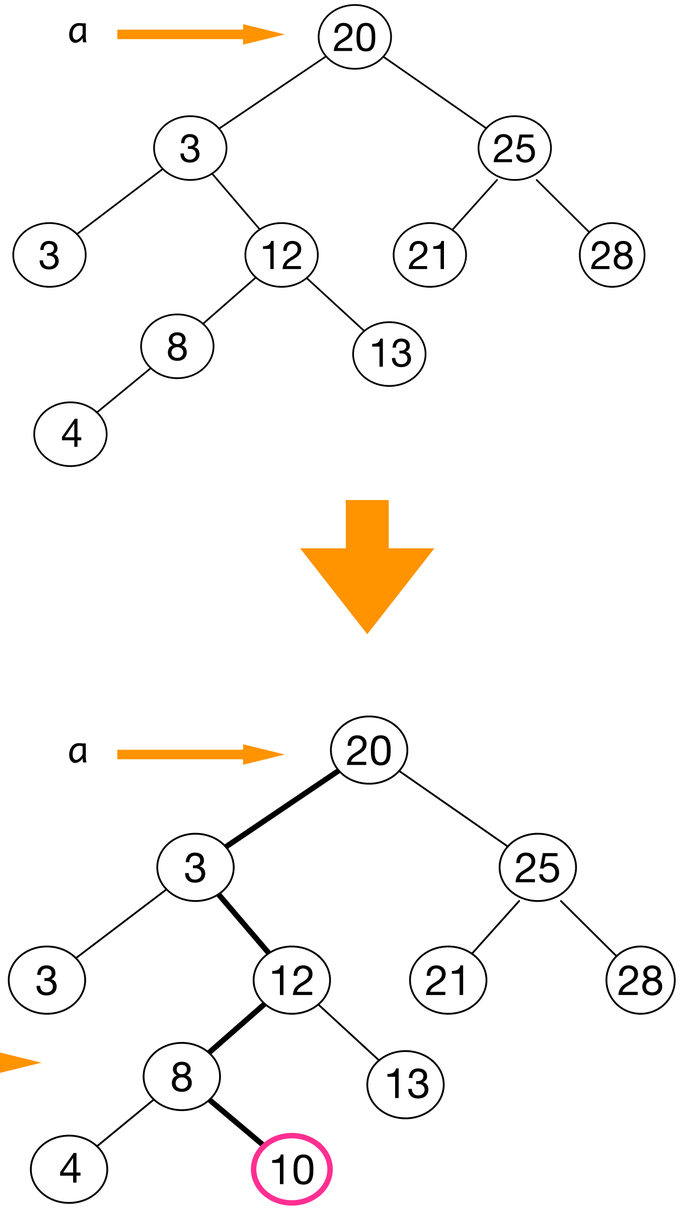
programme ne crée qu'un nouveau noeud

```
def ajouter (x, a) :
    if a == None :
        a = Noeud (x, None, None)
    elif x <= a.val :
        a.gauche = ajouter (x, a.gauche)
    else :
        a.droit = ajouter (x, a.droit)
    return a
```

• on modifie l'arbre a [**« effet de bord »**]

DANGER ! DANGER !

```
b = ajouter (10, a)
```



le fils droit du noeud 8 est modifié

Programmation fonctionnelle ou impérative

- programmation fonctionnelle
  - on ne modifie pas les arbres
  - on rajoute de nouveaux noeuds
  - et on partage les sous-arbres (non modifiés)

données non modifiables

- programmation impérative
  - on fait des effets de bord sur les arbres
  - on modifie donc leur structure
  - on optimise la place mémoire en ne créant pas de nouveaux noeuds
  - danger… danger !!

données modifiables

Au-delà des arbres de recherche

- algorithmes Diviser pour Régner (*divide and conquer*)
- géométrie (*computational geometry*)
- analyse syntaxique

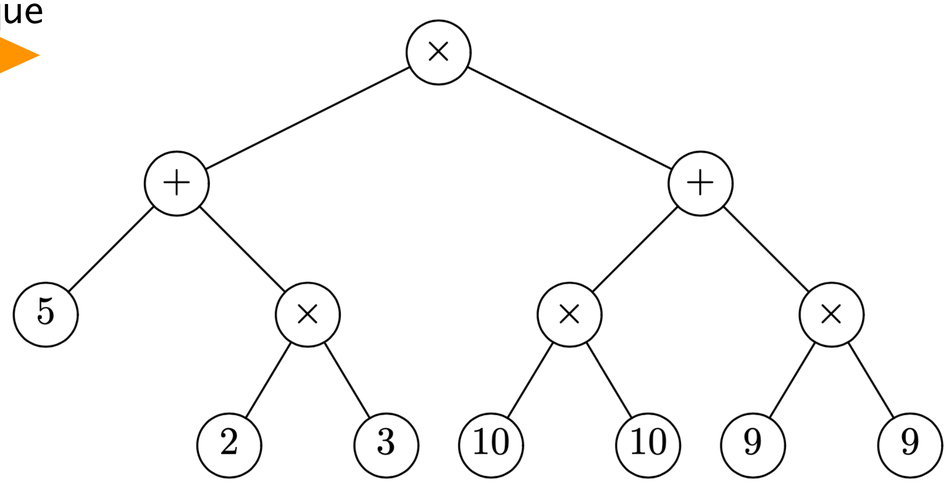
chaîne de caractères

(5 + 2 * 3) * (10 * 10 + 9 * 9)

analyse syntaxique

belle impression

arbre de syntaxe abstraite



- structure arborescente des systèmes de fichiers
- les arbres sont à la base des algorithmes de l'informatique

Arbres de syntaxe abstraite (ASA)

- les ASA représentent des expressions arithmétiques

- analyse syntaxique

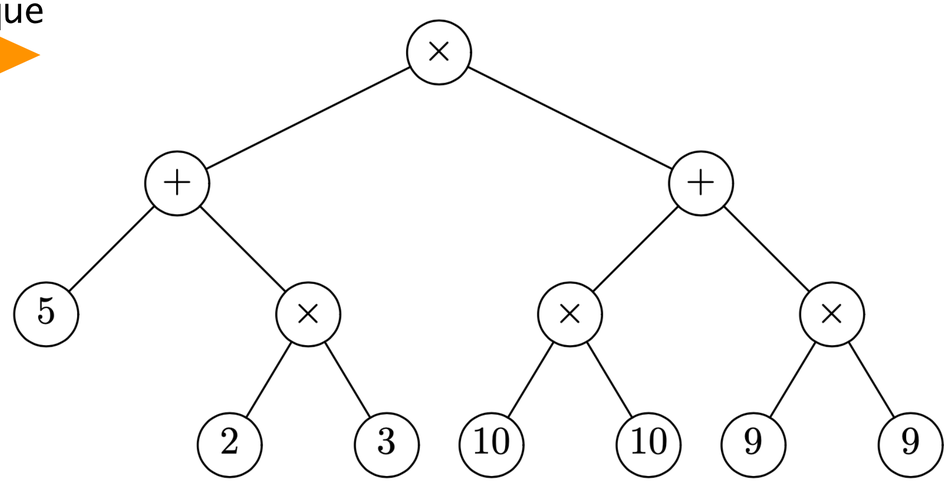
chaîne de caractères

(5 + 2 * 3) * (10 * 10 + 9 * 9)

analyse syntaxique

belle impression

arbre de syntaxe abstraite



Exercice: Imprimer en notation polonaise préfixe

Exercice: Imprimer en notation polonaise postfixe

Exercice: Imprimer en notation infixe sans parenthèses

Exercice: Imprimer en notation infixe avec parenthèses

5 2 3 * + 10 10 * 9 9 * + *

* + 5 * 2 3 + * 10 10 * 9 9

Hint: on tiendra compte de la précédence des opérateurs

```
precedence = {'+': 1, '*': 2, '-': 1, '/':2, '**':3}
```

Question ++: comment privilégier l'assiociation à gauche ou à droite pour les opérateurs de même précédence ?

Arbres de syntaxe abstraite (ASA)

- les ASA représentent des expressions arithmétiques

- analyse syntaxique

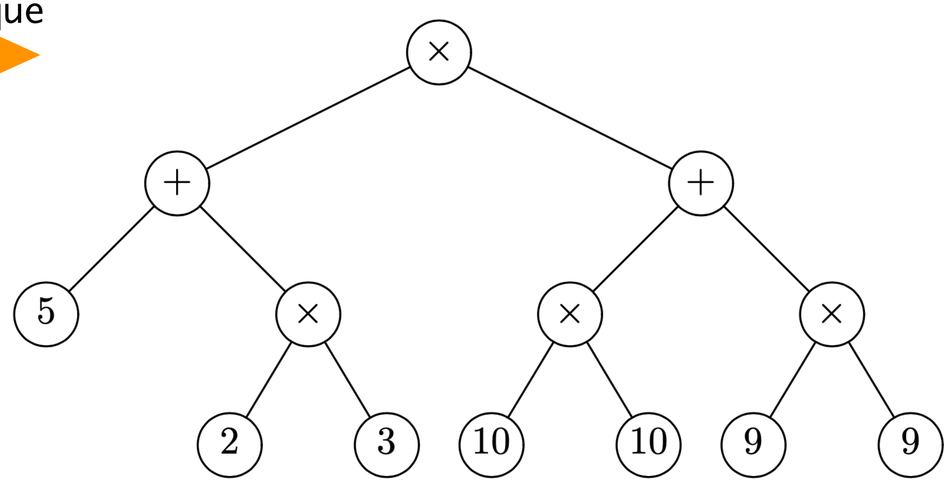
chaîne de caractères

(5 + 2 * 3) * (10 * 10 + 9 * 9)

analyse syntaxique

belle impression

arbre de syntaxe abstraite



Exercice: Evaluer le résultat d'un arbre ASA dans un environnement donné.

Hint: on représentera l'environnement par un dictionnaire associant une valeur à toute variable

```
env = {'x': 1, 'y': -2, 'z': 10}
```

Python ++

- traitement des exceptions
  - `try:` début d'un bloc avec exception possible
  - `except IOError:` récupère l'exception `IOError`
  - `except:` récupère toutes les exceptions
  - `finally:` pour le traitement normal **et** le traitement exceptionnel

```
def lire_lignes (nom) :
    try:
        f = open (nom, 'r')
        return f.read().splitlines()
    except IOError:
        print("Fichier '%s' inexistant." % nom)

lire_lignes('abc')
```

➨ Fichier 'abc' inexistant.

- alias de module
  - `as` déclare un alias pour un module (par exemple si le nom est trop long)

```
import random as r

r.choice (['a', 'b', 'c'])
```

➨ 'a'

Prochain cours

- réviser les classes et objets
- graphes
- parcours de graphe
- arbres de recouvrement
- recherche de chemins