Informatique et Programmation

Appendice 4

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py-22

Plan

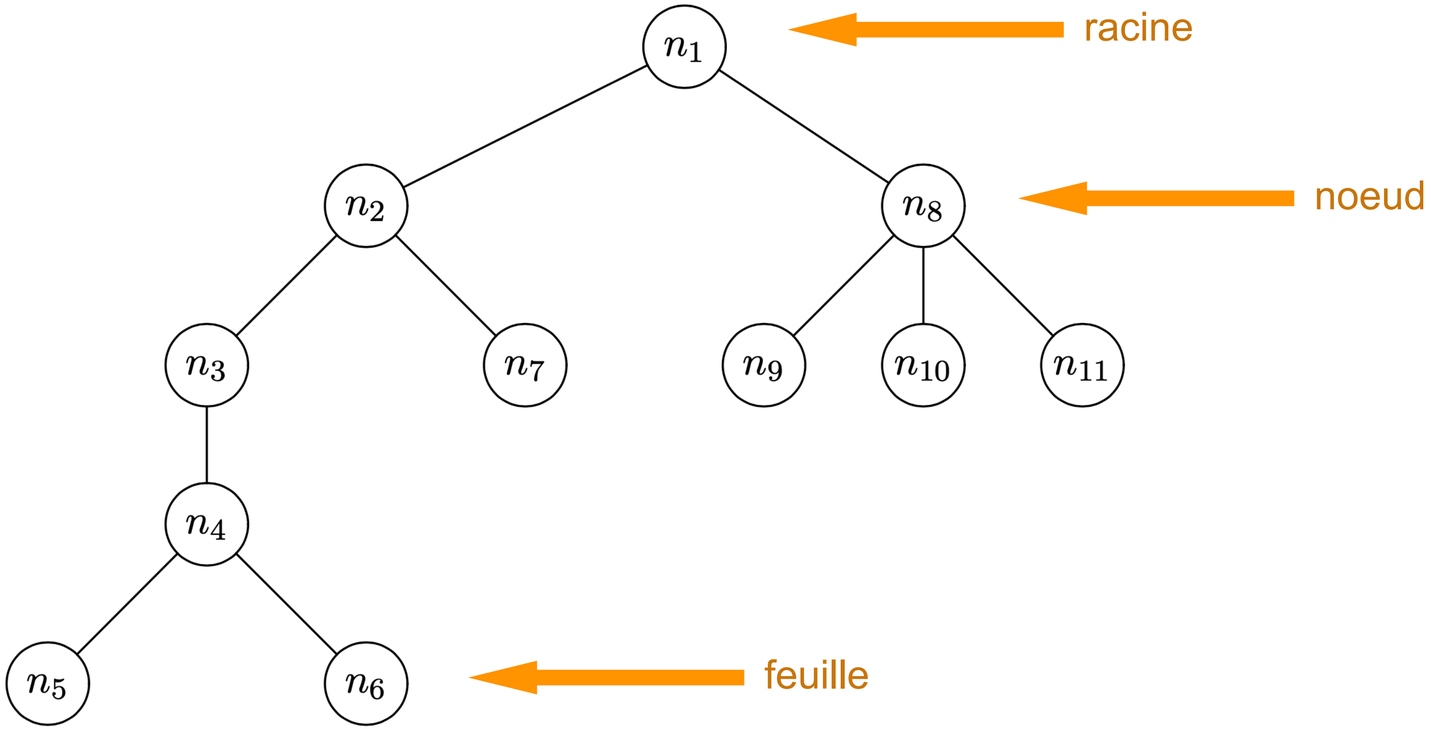
- arbres en informatique
- arbre binaire de recherche
- arbre de syntaxe abstraite
- rudiments d'analyse syntaxique

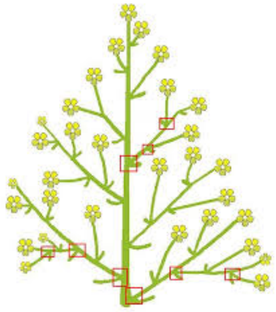
dès maintenant: télécharger Python 3 en `http://www.python.org`

un cours Python en `http://www.w3schools.com/python/default.asp`

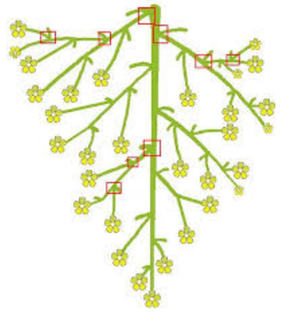
Les arbres en informatique

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





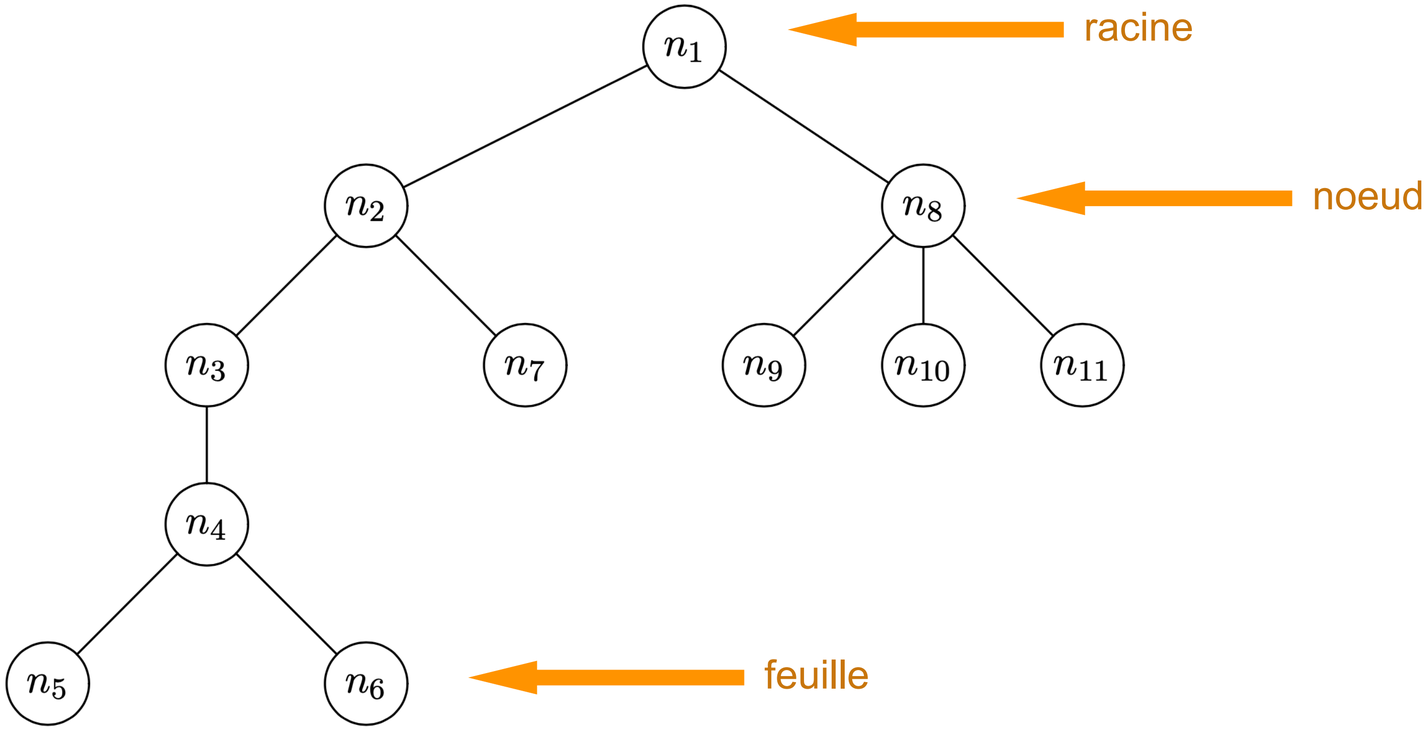
botanique

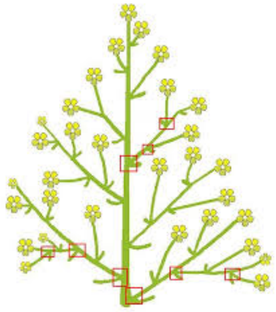


informatique

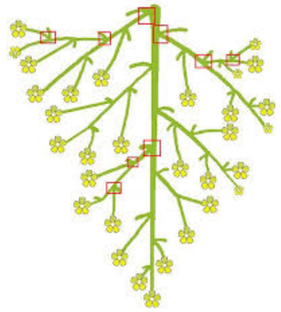
Les arbres en informatique

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





botanique



informatique

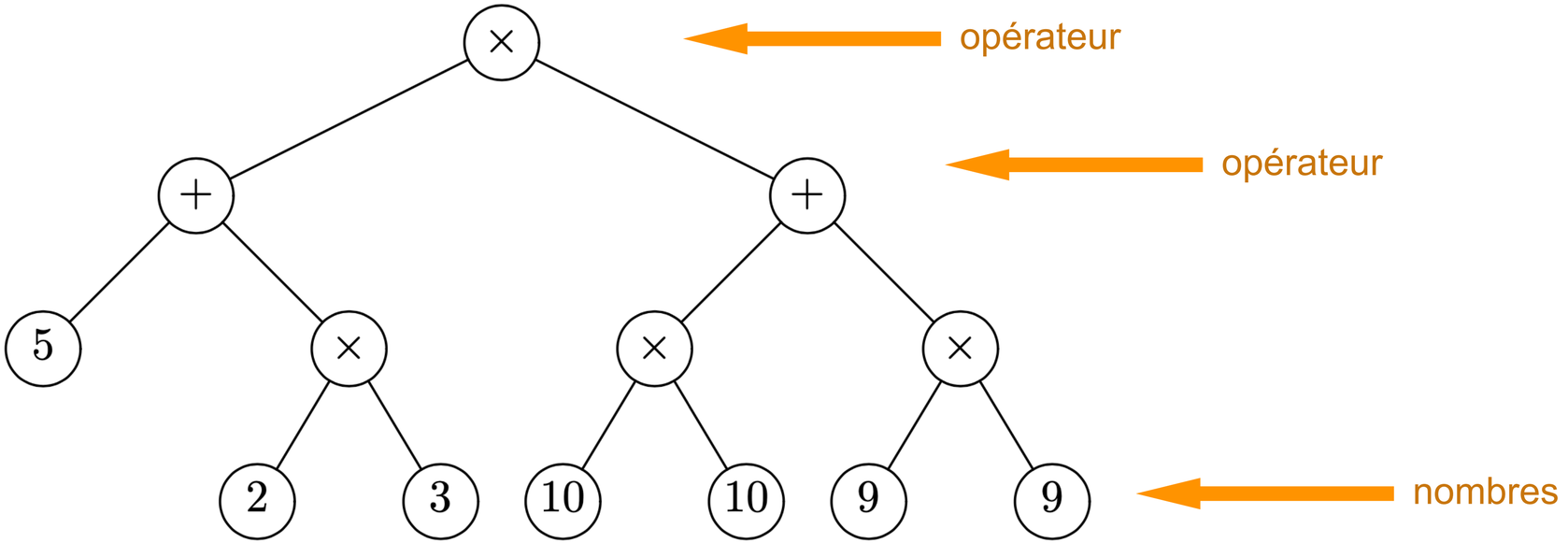
n2 est un **ancêtre** de n4

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 valeurs quelconques [ici des chaînes de caractères]



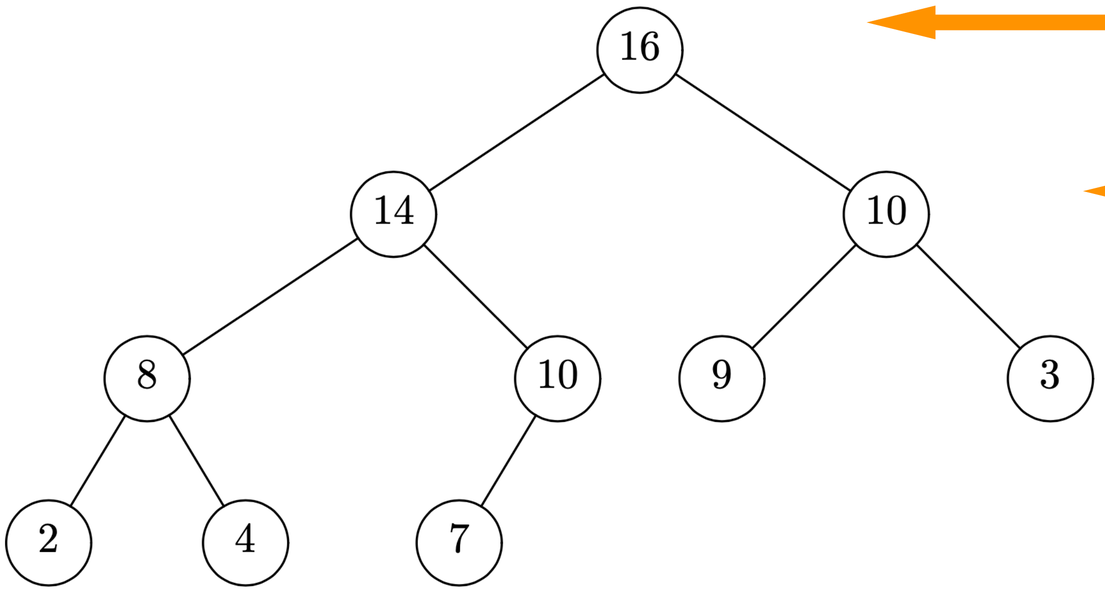
pour représenter une **expression arithmétique**

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

[plus besoin de parenthèses]

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

[ici un ancêtre a une valeur plus élevée que ses descendants]

Arbres (représentation 1)

- on définit une classe avec des champs et des méthodes

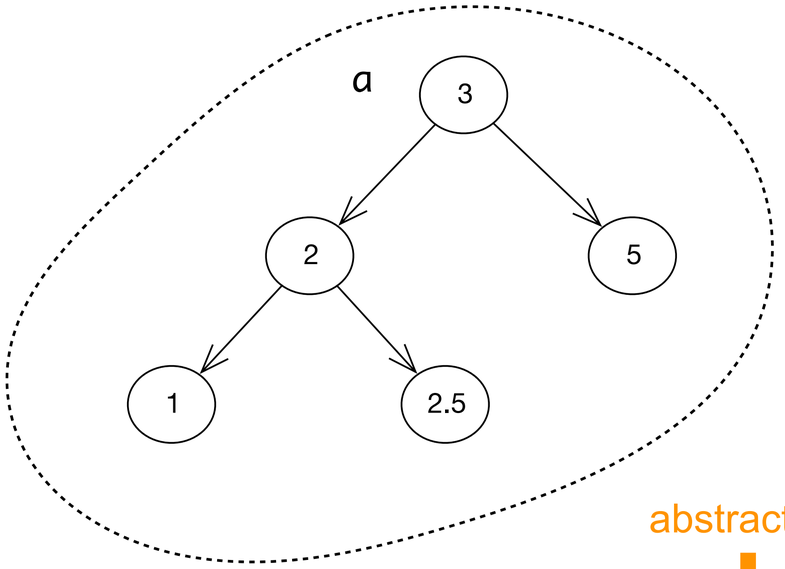
```
class Noeud:
    def __init__ (self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d
```

- on définit une classe pour les feuilles

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

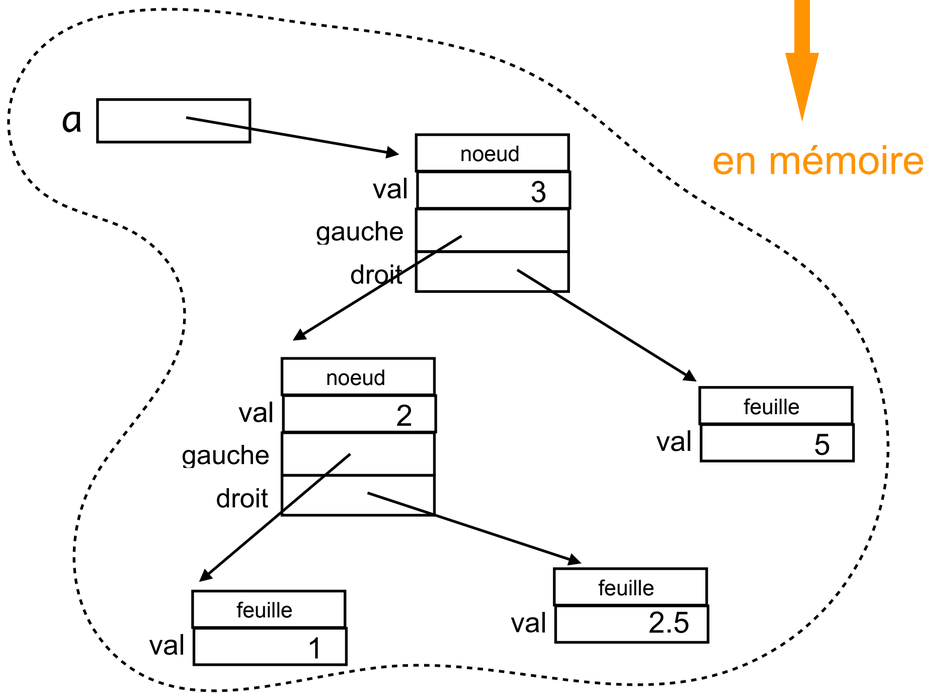
- et on construit des arbres

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



abstraction

en mémoire



Arbres

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

```
class Noeud:
    # comme avant

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


class Feuille:
    # comme avant

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

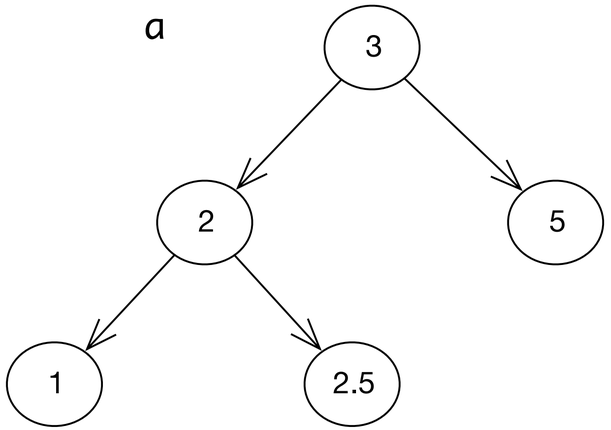
- on construit et imprime des arbres

```
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))
```

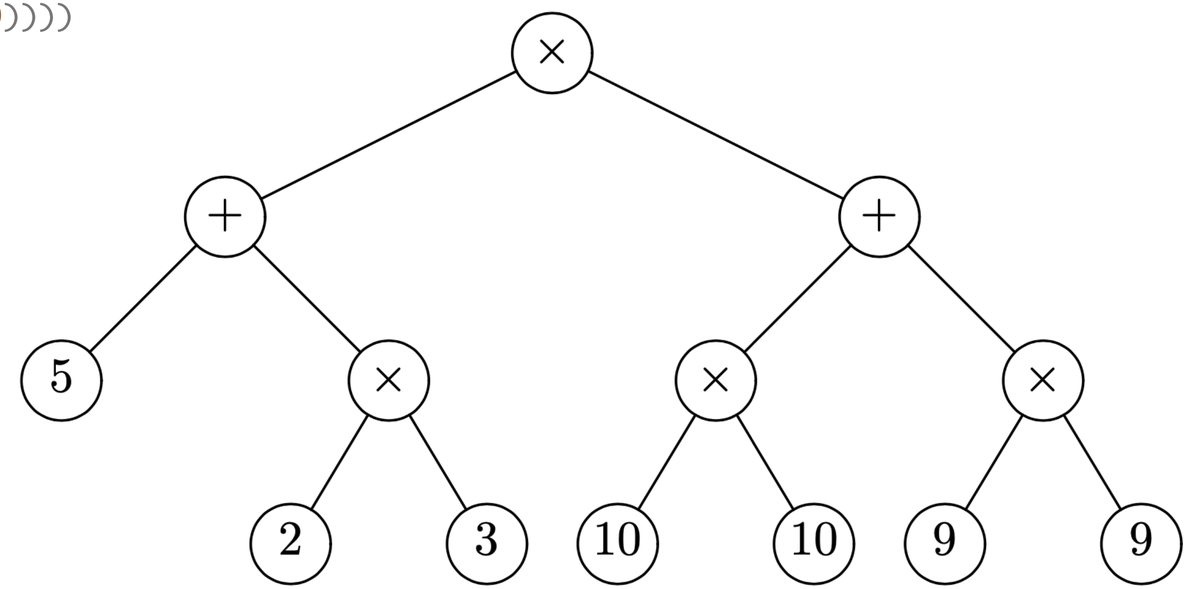


Arbres

- 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))
```



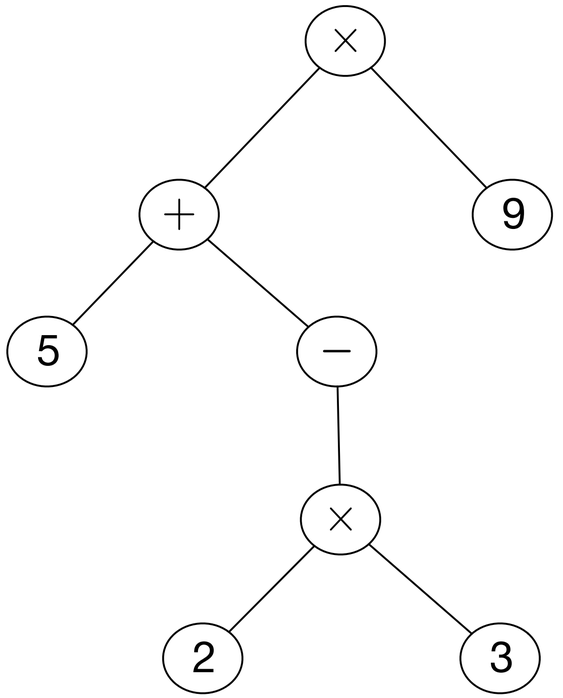
Arbres (représentation 2)

- On peut distinguer les noeuds binaires et les noeuds unaires

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

class Noeud_Un:
    def __init__ (self, x, a) :
        self.val = x
        self.fils = a

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



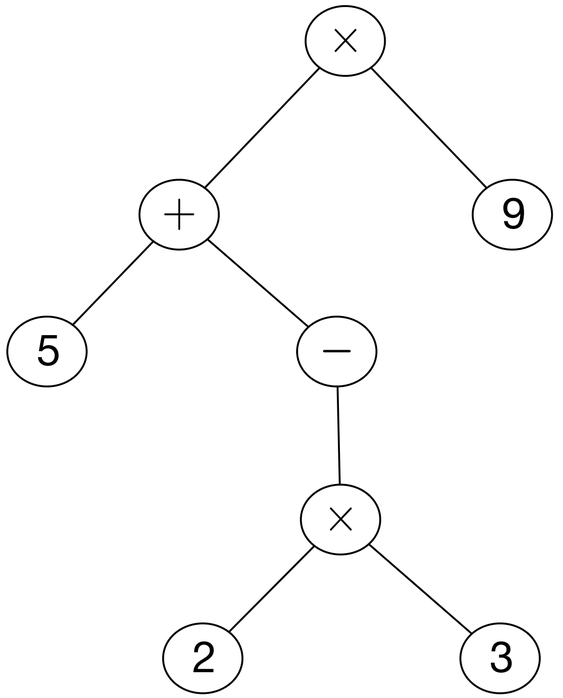
- et on construit l'arbre par:

```
d = Noeud_Bi ('*',
              Noeud_Bi ('+', Feuille (5),
                             Noeud_Un ('-',
                                       Noeud_Bi ('*', Feuille (2), Feuille (3)))),
              Feuille (9))
```

Arbres (représentation 3)

- None représente l'arbre vide (0 neuds, 0 feuilles)

```
c = Noeud ('*', Noeud ('+', Feuille (5),
                            Noeud ('-', Noeud ('*', Feuille (2), Feuille (3)),
                                   None)),
                Feuille (9))
```



- ou encore ici :

```
c = Noeud ('*', Noeud ('+', Feuille (5),
                            Noeud ('-', None,
                                        Noeud ('*', Feuille (2), Feuille (3)))),
                Feuille (9))
```

- ou encore en identifiant feuilles et noeuds sans fils

```
c = Noeud ('*', Noeud ('+', Noeud (5, None, None),
                            Noeud ('-', None,
                                        Noeud ('*', Noeud (2, None, None), Noeud (3, None, None)))),
                Noeud (9, None, None))
```

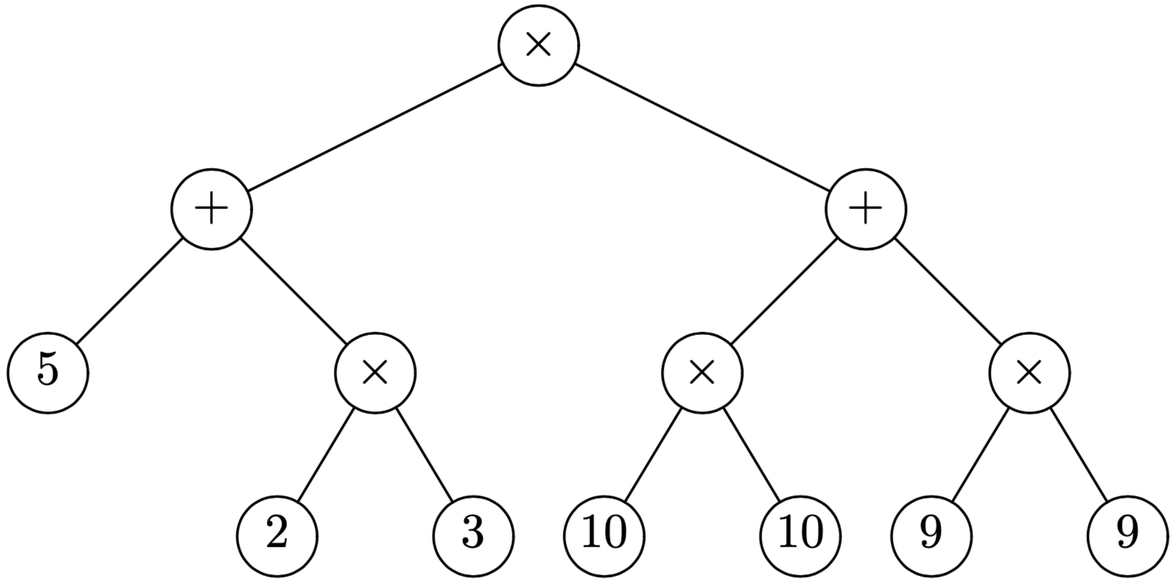
Arbres (et fonctions récursives)

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))))

print (hauteur (b))
3

print (taille (b))
13
```

Arbres (et méthodes)

- On parcourt ou calcule sur les arbres avec des méthodes

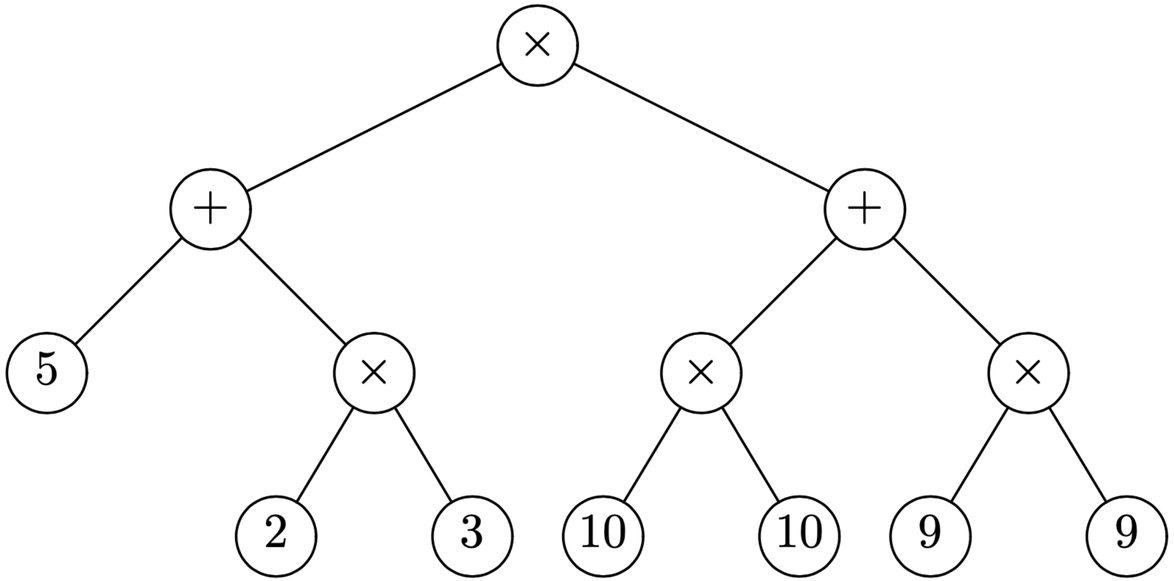
```
class Noeud:
    # comme avant et en plus :
    def hauteur (self) :
        return 1 + max (self.gauche.hauteur(), self.droit.hauteur())

    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
```

par cas sur sous-classes



- et on calcule les hauteur et taille

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

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

Arbres

- choisir entre fonctions récursives et méthodes est affaire de goût
- les fonctions récursives favorisent la programmation procédurale

  [**tout est fonction (procédure)**]

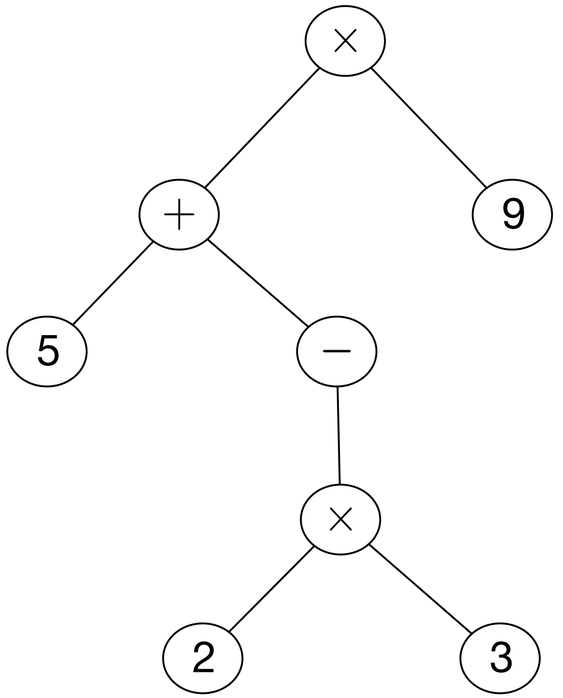
- les méthodes privilégient la programmation par objets

  [**tout est piloté par les données**]

Arbres (représentation 4)

• une arborescence est une représentation par lien arrière

```
arbre = [ ('x', None),
          ('+', 0),
          ('5', 1),
          ('-', 1),
          ('x', 3),
          ('2', 4),
          ('3', 4),
          ('9', 4)]
```



• ou encore

```
val = ['x', '+', '5', '-', 'x', '2', '3', '9']
pere = [None, 0, 1, 1, 3, 4, 4, 0]
```

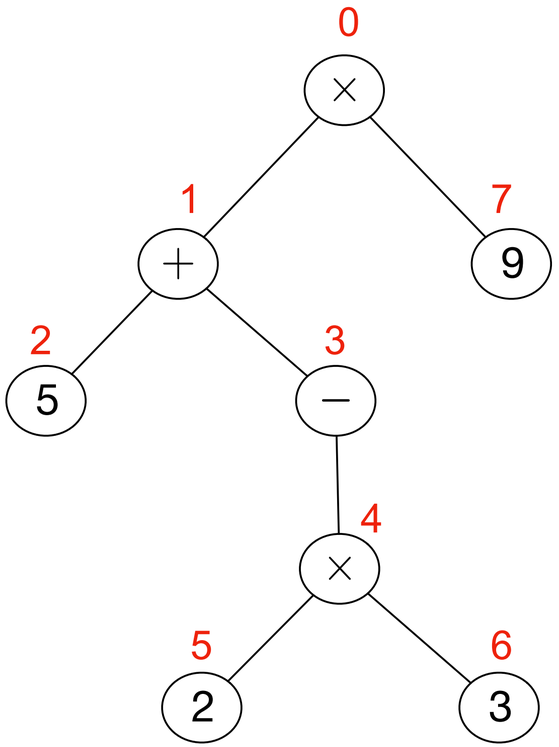
• la représentation plus souple car ne distinguant pas l'arité des noeuds

• mais elle ne permet pas un simple parcours d'arbre

Exercice Passer de la représentation 1 à la représentation 4 et réciproquement.

Parcours d’arbre

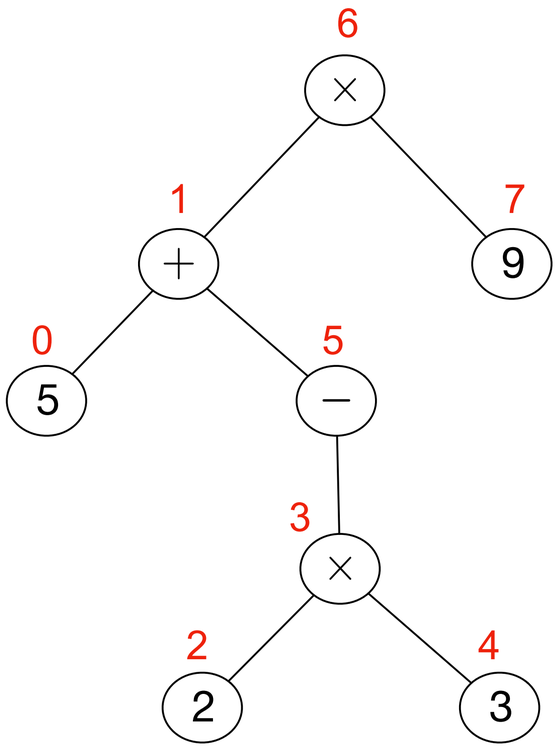
• 3 parcours d’arbre (préfixe, infixe, postfixe)



préfixe

• notation polonaise préfixe

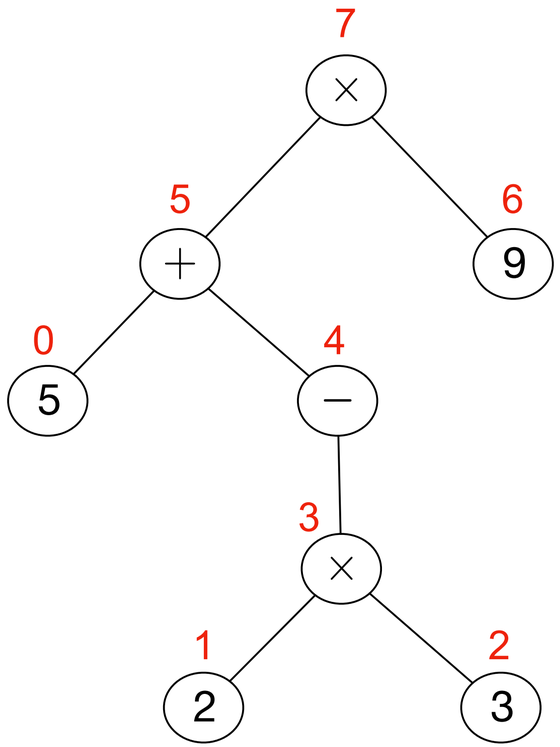
x + 5 - x 2 3 9



infixe

• notation arithmétique

(5 + - (2 x 3)) x 9



postfixe

• notation polonaise postfixe

5 2 3 x - + 9 x

Parcours d'arbre

- générer les notations préfixe, postfixe et infixe

```
def polprefix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return a.val + ' ' + polprefix (a.fils)
    else :
        return a.val \
              + ' ' + polprefix (a.gauche) \
              + ' ' + polprefix (a.droit)
```

```
def polpostfix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return polpostfix (a.fils) + ' ' + a.val
    else :
        return polpostfix (a.gauche) \
             + ' ' + polpostfix (a.droit) \
             + ' ' + a.val
```

```
def notinfixe (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return '(' + a.val + ' ' + notinfixe (a.fils) + ')'
    else :
        return '(' + notinfixe (a.gauche) \
             + ' ' + a.val \
             + ' ' + notinfixe (a.droit) + ')'
```

trop de parenthèses

[on verra plus tard pour les enlever]

- notation polonaise préfixe

  x + 5 - x 2 3 9

- notation infixe

  (5 + - (2 x 3)) x 9

- notation polonaise postfixe

  5 2 3 x - + 9 x

Arbres (représentation 5)

- Arbres n-aires avec nombre arbitraire de fils

```
class Noeud:
    def __init__ (self, x, l) :
        self.val = x
        self.fils = l
    #
    def __str__ (self) :
        r = ''
        for a in self.fils :
            r = r + ', ' + str(a)
        return "Noeud ({})".format (r[2:])

class Feuille:
    def __init__ (self, x) :
        self.val = x
    #
    def __str__ (self) :
        return "Feuille ({})".format (self.val)

a = Noeud (10,
        [Noeud (12, [Feuille (3)]),
                Feuille (4), Feuille (5)])
print (a)
```

```
def __str__ (self) :
    r = ', '.join(map(str, self.fils))
    return "Noeud ({})".format (r)
```

Arbres binaires de recherche

- recherche en table organisée en **arbre binaire**

- chaque paire (clé, valeur) est stockée dans les noeuds et feuilles

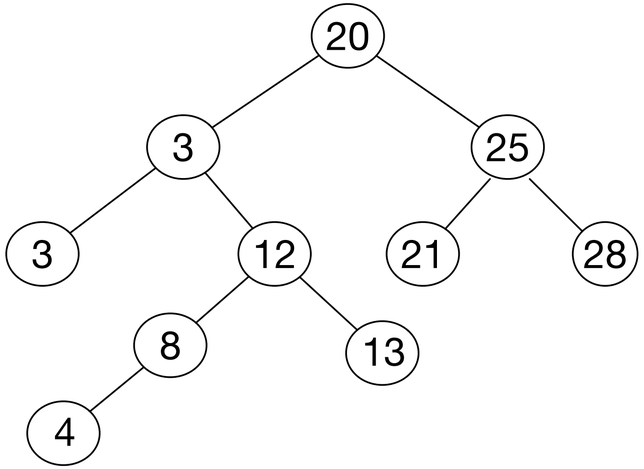
  [**on simplifie ici en ne considérant que les clés**]

- les clés sont stockées dans **l'ordre préfixe**:

  la clé d'un noeud est plus grande que les clés de son fils gauche

  la clé d'un noeud est plus petite que les clés de son fils droit

  [ici, on met les clés égales vers la gauche]




```
def rechercher (x, a) :
    if a == None :
        return False
    elif isinstance (a, Feuille) :
        return x == a.val
    elif x == a.val :
        return True
    elif x < a.val :
        return rechercher (x, a.gauche)
    else :
        return rechercher (x, a.droit)
```

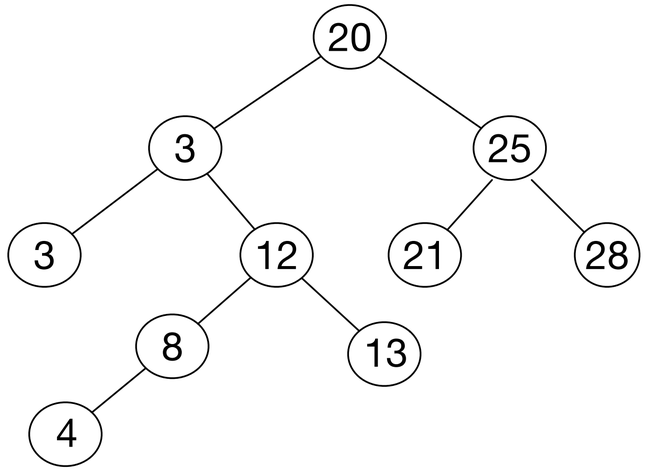
Arbres binaires de recherche

- les clés sont stockées dans l'ordre préfixe:

  la clé d'un noeud est plus grande que les clés de son fils gauche

  la clé d'un noeud est plus petite que les clés de son fils droit

  [ici, on met les clés égales vers la gauche]




```
def rechercher (x, a) :
    if a == None :
        return False
    elif x == a.val :
        return True
    elif x < a.val :
        return rechercher (x, a.gauche)
    else :
        return rechercher (x, a.droit)
```

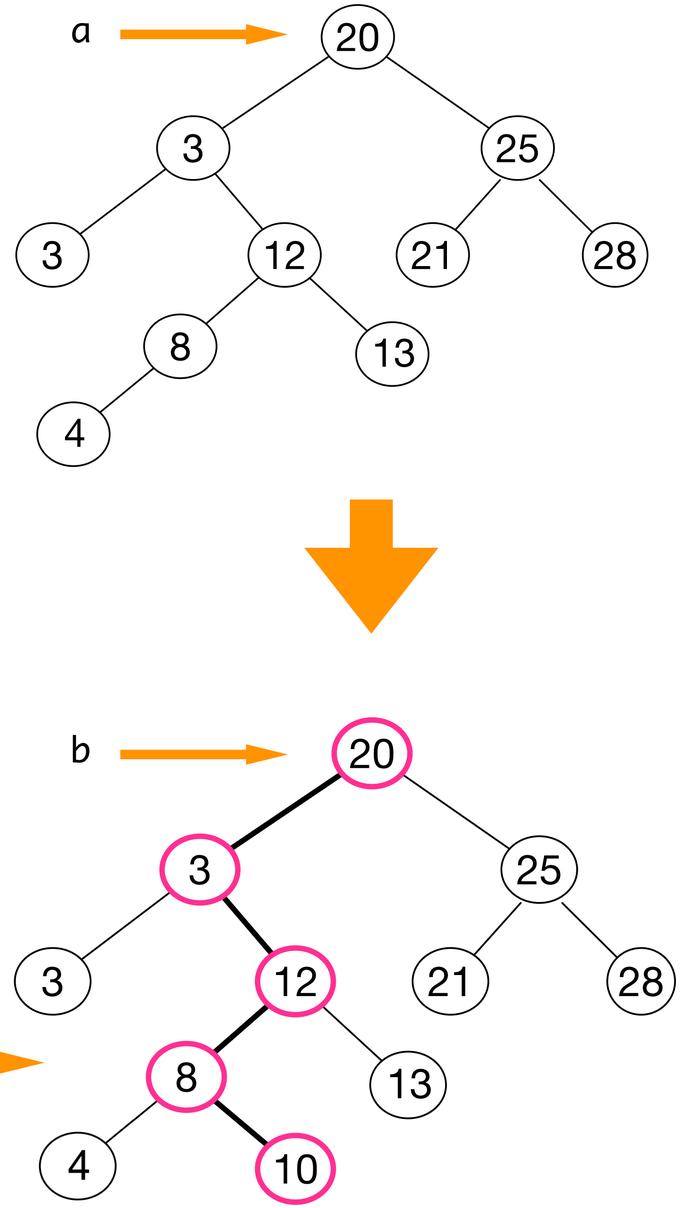
programme plus simple avec un seul type de noeud

Arbres binaires de recherche

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

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

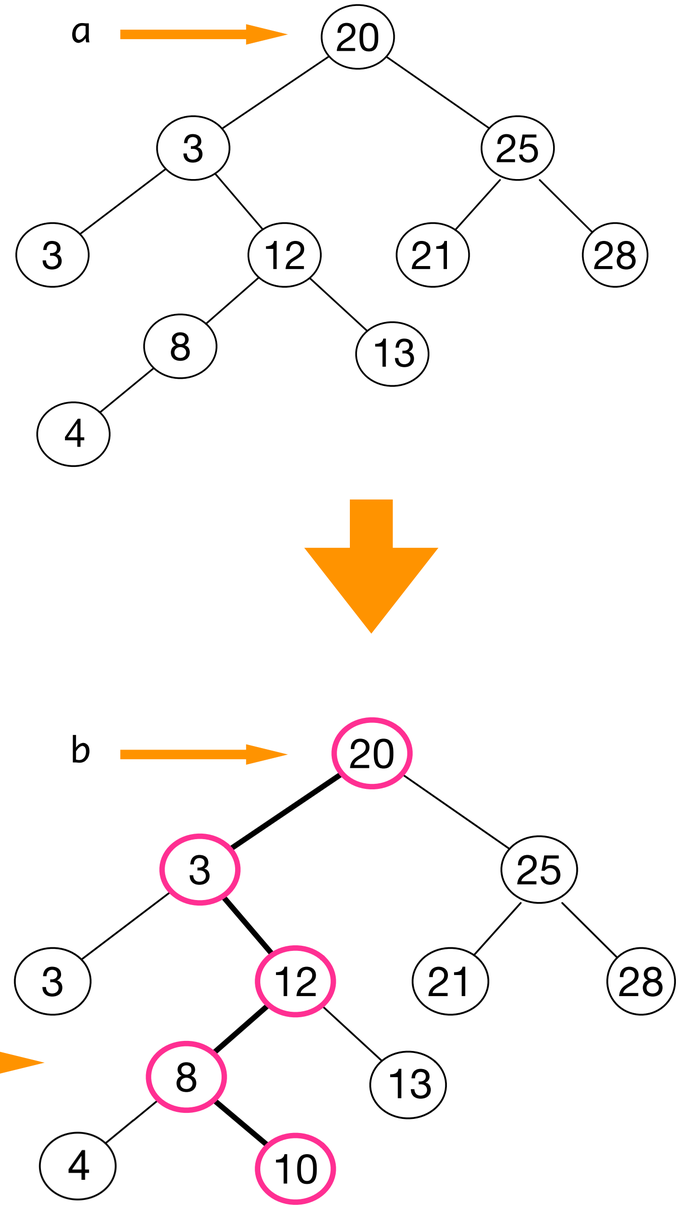
Arbres binaires de recherche

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

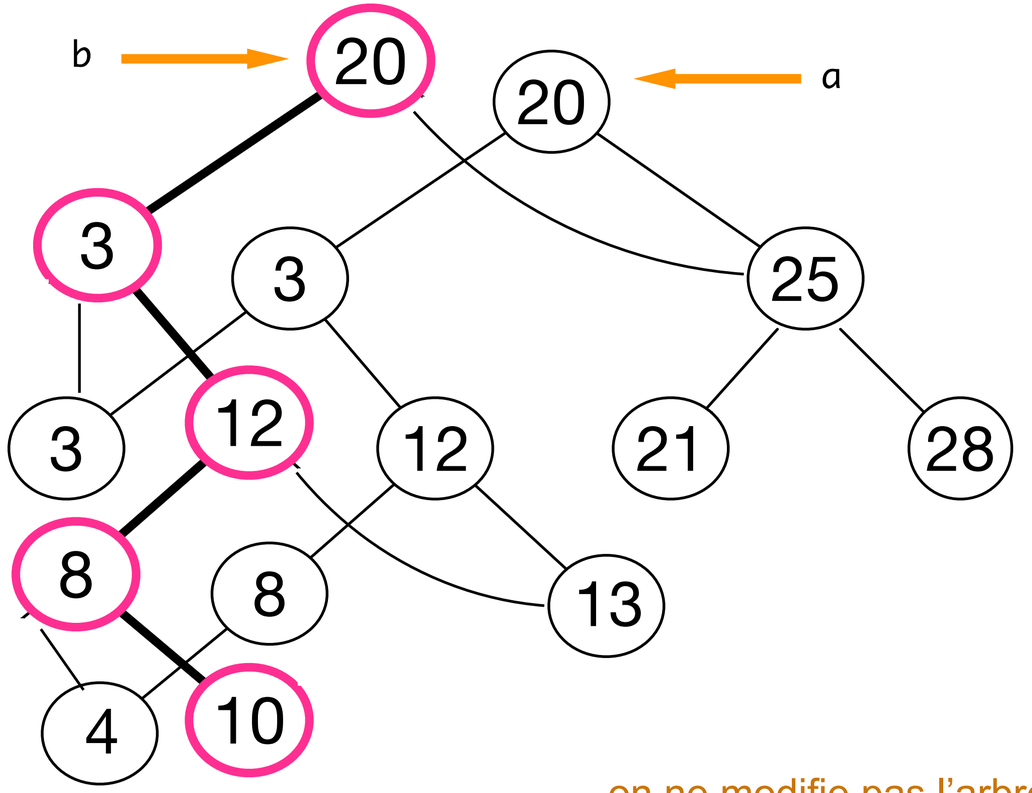
```
b = ajouter (10, a)
```



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

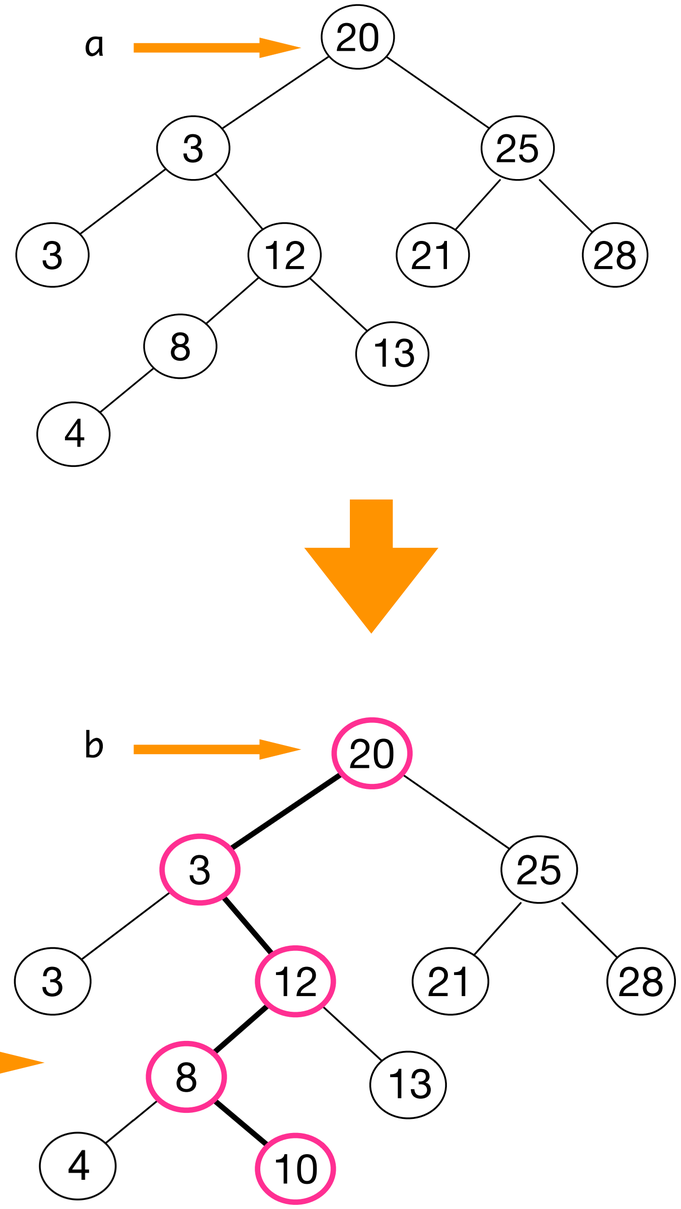
Arbres binaires de recherche

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




```
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**)

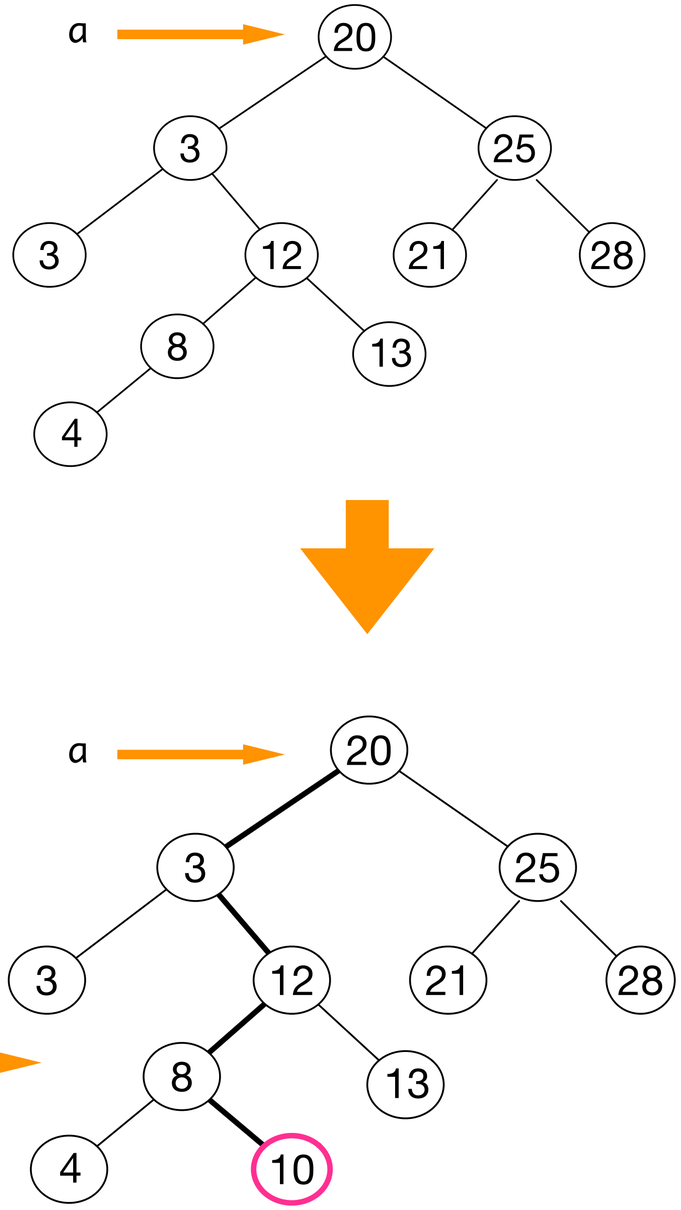
programme ne crée qu'un nouveau noeud

```
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

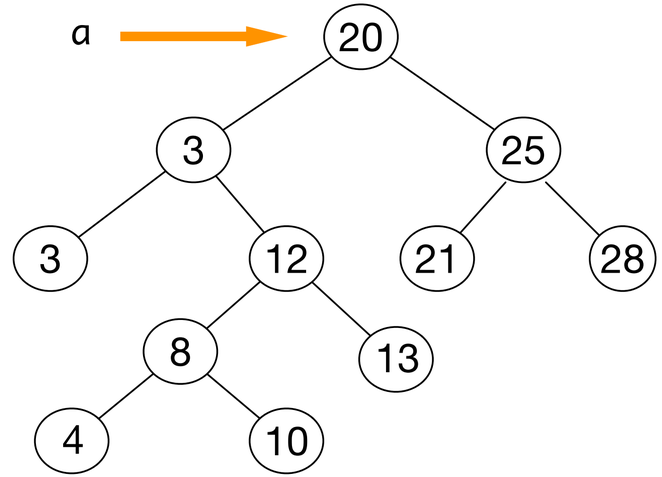
Arbres binaires de recherche

- supprimer une clé

  on la supprime simplement si la clé est dans une feuille

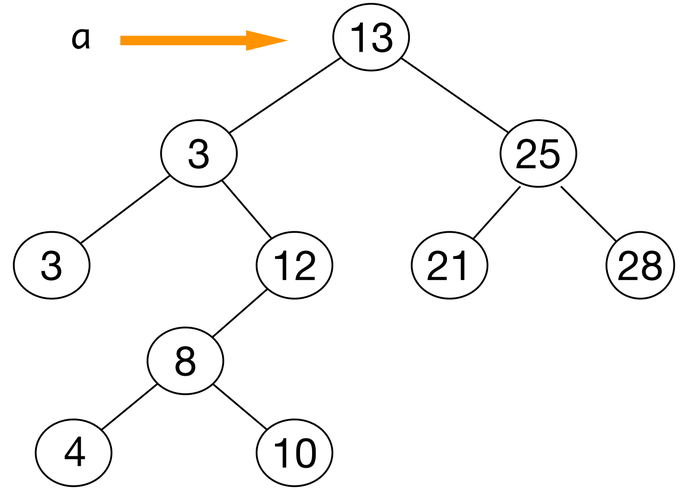
  sinon on la remplace par la plus grande dans le sous-arbre de gauche ou la plus petite dans le sous-arbre de droite

le programme est plus compliqué



Exercice écrire la fonction `supprimer (x, a)`

```
a = supprimer (20, a)
```



Arbres de syntaxe abstraite

- représentation d'expressions arithmétiques (ou plus généralement de programmes)

```
class ASA :
    def __init__ (self, val, a1, a2) :
        self.value = val
        self.gauche = a1
        self.droite = a2

    def __str__ (self) :
        return '({}, {}, {})'.format \
           (self.value, self.gauche, self.droite)
```

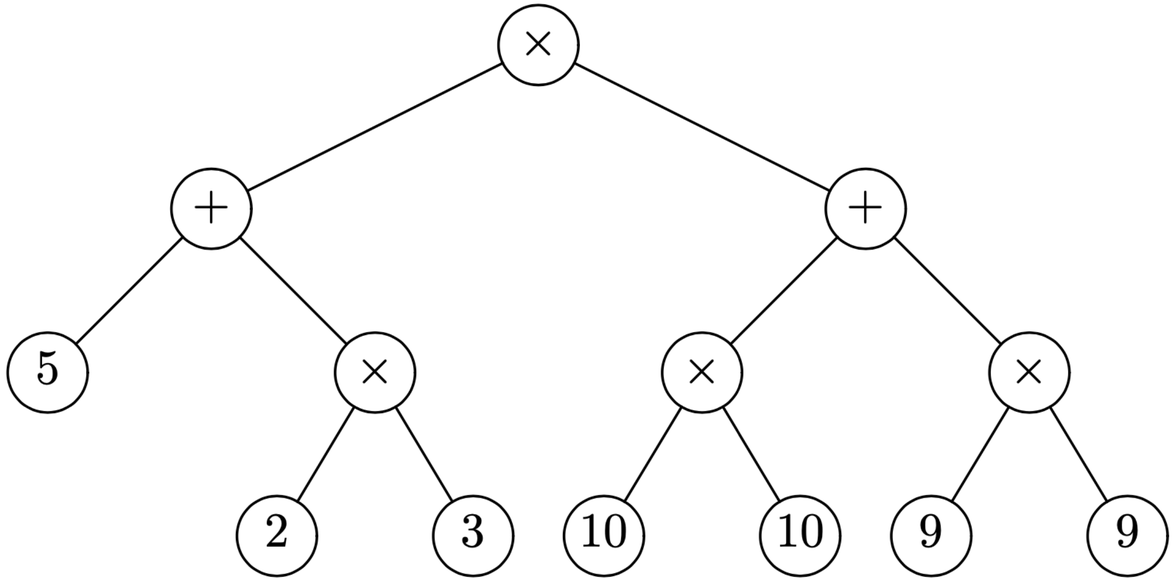
```
Op_bi = ASA

class Op_un (ASA) :
    def __init__ (val, a1) :
        super().__init__ (val, a1, None)

    def __str__ (self) :
        return '({}, {})'.format (self.value, self.gauche)


class CVar (ASA) :
    def __init__ (val) :
        super().__init__ (val, None, None)

    def __str__ (self) :
        return '{}'.format (self.value)
```




```
a = Op_bi ('*', Op_bi ('+', CVar(5),
                       Op_bi ('*', CVar (2), CVar (3))),
              Op_bi ('+', Op_bi ('*', CVar (10), CVar (10)),
                   Op_bi ('*', CVar (9), CVar(9))))
```

Arbres - Belle impression

- on peut réduire le nombre de parenthèses si on connait la précédence des opérateurs

- en mathématiques, ‘*’ a une plus forte précédence que ‘+’

$$3 + 4 \times 5 \quad \equiv \quad 3 + (4 \times 5)$$

$$(11 + 3) * 4 \quad \not\equiv \quad 11 + 3 * 4$$

- on peut donc faire le dictionnaire suivant de précédences: `preds = {'+': 0, '*': 2, '-': 3}`

- la fonction d’impression met des parenthèses si la précédence est inférieure à la précédence du contenant

Arbres - Belle impression

- on peut réduire le nombre de parenthèses si on connait la précédence des opérateurs

```
preds = {‘+': 0, '*': 2, ‘-unaire': 3}


def str (a, p) :
    if isinstance (a, CVar) :
        return '{}'.format (a.val)
    else :
        q = preds[a.val]
        if p > q :
            return '({} {} {})'.format (str (a.gauche, q),
              a.val, str(a.droite, q))
        else :
            return '{} {} {}'.format (str (a.gauche, q),
              a.val, str(a.droite, q))
```

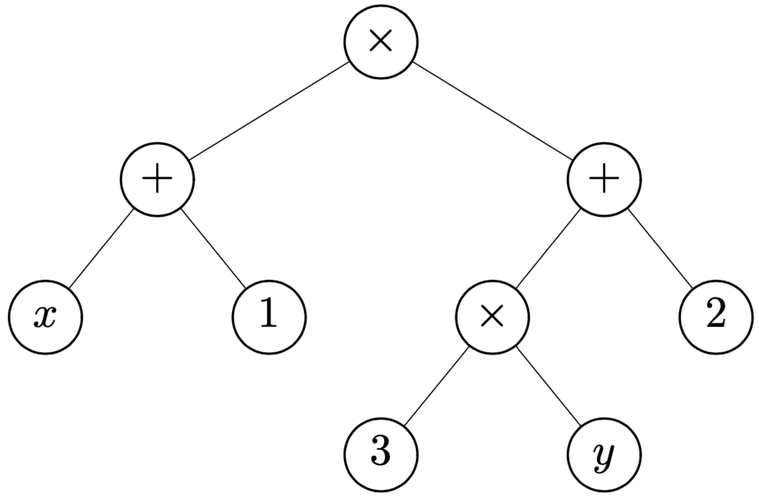
Arbres de syntaxe abstraite

- passer d'une chaîne de caractères à un arbre (syntaxe astraite) est plus difficile

$(x + 1) * (3 * y + 2)$

Analyse syntaxique

Belle impression

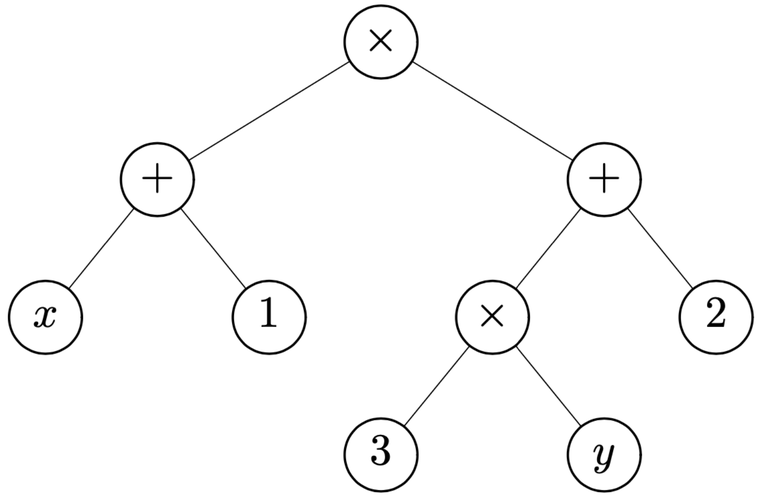


Arbres de syntaxe abstraite

- on peut évaluer sa valeur en donnant une valeur aux variables *x* et *y*

- on définit l'environnement par le dictionnaire:

```
e = {'x' : 20, 'y' : -20}
```




```
def eval (t, e) :
    if isinstance (t, CVar) :
        if isinstance (t.val, int) :
            return t.val
        else :
            return e[t.val]
    elif isinstance (t, Op_un) :
        if t.val == '-unaire' :
            return - eval (t.gauche, e)
        else:
            raise Exception
    elif isinstance (t, Op_bi) :
        if t.val == '+' :
            return eval (t.gauche, e) + eval (t.droit, e)
        elif t.val =='*' :
            return eval (t.gauche, e) * eval (t.droit, e)
    else :
        raise Exception
```

```
t = Op_bi ('*', Op_bi ('+', CVar ('x'), CVar (1)),
           Op_bi ('+', Op_bi ('*', CVar (3), CVar ('y')),
                       CVar (2)))


print (eval (t, e))
```

à faire

- retour sur les objets et les arbres
- analyses lexicale et syntaxique
- modularité et programmation objet
- programmation graphique
- algorithmes géométriques
- calculs flottants et méthodes numériques
- programmation de plusieurs fils de calcul
- assertions et logique des programmes
- introduction à l'informatique théorique
- etc

vive l’informatique

et

la programmation !