Informatique et Programmation

Cours 11

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

Plan

- algorithme glouton (marche du cavalier)
- exploration exhaustive (les 8 reines)
- programmation dynamique (plus longue chaîne commune)

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

Programmation dynamique

- plus longue sous-séquence commune entre 2 chaînes de caractères (commande Unix `diff`)

  [on mémorise les solutions partielles — $m \times n$ opérations]

```
GAUCHE = 1
HAUT = 2
DIAG = 3
```

```
def new_matrix (m, n, v) :
    a = [ [v for _ in range (n) ] for _ in range (m) ]
    return a
```

```
def longueurSSC (u, v) :
  m = len(u); n = len(v)
  lg = new_matrix (m+1, n+1, 0)
  p = new_matrix (m+1, n+1, 0)
  for i in range(1, m+1):
    for j in range(1, n+1):
      if u[i-1] == v[j-1]:
        lg[i][j] = 1 + lg[i-1][j-1]
        p[i][j] = DIAG;
      elif lg[i][j-1] > lg[i-1][j]:
        lg[i][j] = lg[i][j-1]
        p[i][j] = GAUCHE
      else :
        lg[i][j] = lg[i-1][j]
        p[i][j] = HAUT
  return (lg[m][n], p)
```

u = ‘abcadefg’ v = ‘fbcexyg’

lg

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 3 3 3 3
0 1 1 2 3 3 3 3
0 1 1 2 3 3 3 4
```

p

```
0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2
0 2 3 1 1 1 1 1
0 2 2 3 1 1 1 1
0 2 2 2 2 2 2 2
0 2 2 2 2 2 2 2
0 2 2 2 3 1 1 1
0 3 2 2 2 2 2 2
0 2 2 2 2 2 2 3
```

Programmation dynamique

- plus longue sous-séquence commune entre 2 chaînes de caractères (commande Unix `diff`)

  [on mémorise les solutions partielles — $m \times n$ opérations]

```
def ssc (u, v) :
    m = len(u); n = len(v)
    lgp = longueurSSC (u, v)
    lg = lgp[0]; p = lgp[1]
    r = ''; i = m; j = n;
    while lg > 0 :
        if p[i][j] == DIAG :
            r = u[i-1] + r
            i = i - 1; j = j - 1;
            lg = lg - 1
        elif p[i][j] == GAUCHE :
            j = j - 1
        else :
            i = i - 1
    return r


print (ssc ('abcadefg', 'fbcexyg'))
```

u = ‘abcadefg’ v = ‘fbcexyg’

lg

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 3 3 3 3
0 1 1 2 3 3 3 3
0 1 1 2 3 3 3 4
```

p

```
0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2
0 2 3 1 1 1 1 1
0 2 2 3 1 1 1 1
0 2 2 2 2 2 2 2
0 2 2 2 2 2 2 2
0 2 2 2 3 1 1 1
0 3 2 2 2 2 2 2
0 2 2 2 2 2 2 3
```

Programmation dynamique

- calcul de fibonacci

  [on mémorise les calculs intermédiaires]

```
def fib (n) :
    if n == 0 or n == 1 :
        return n
    else :
        return fib (n-1) + fib (n-2)
```

fibonacci

```
>>> fib (10)
55
>>> fib (20)
6765
>>> fib (35)
9227465
```

calcul en temps exponentiel

[en consommant l'espace de la récursion
ici aussi espace linéaire]

```
def fib (n) :
    a = (n+1)*[0]
    a[1] = 1
    for i in range(2, n+1) :
        a[i] = a[i-1] + a[i-2]
    return a[n]
```

fibonacci

```
>>> fib (10)
55
>>> fib (20)
6765
>>> fib (35)
9227465
```

calcul en temps linéaire

[en consommant plus de mémoire]

- plus court chemin dans un graphe [Dijkstra]

  [cf. plus tard]

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.x)
10
print (p1.y)
20
```

nouvel objet de la classe `Point`

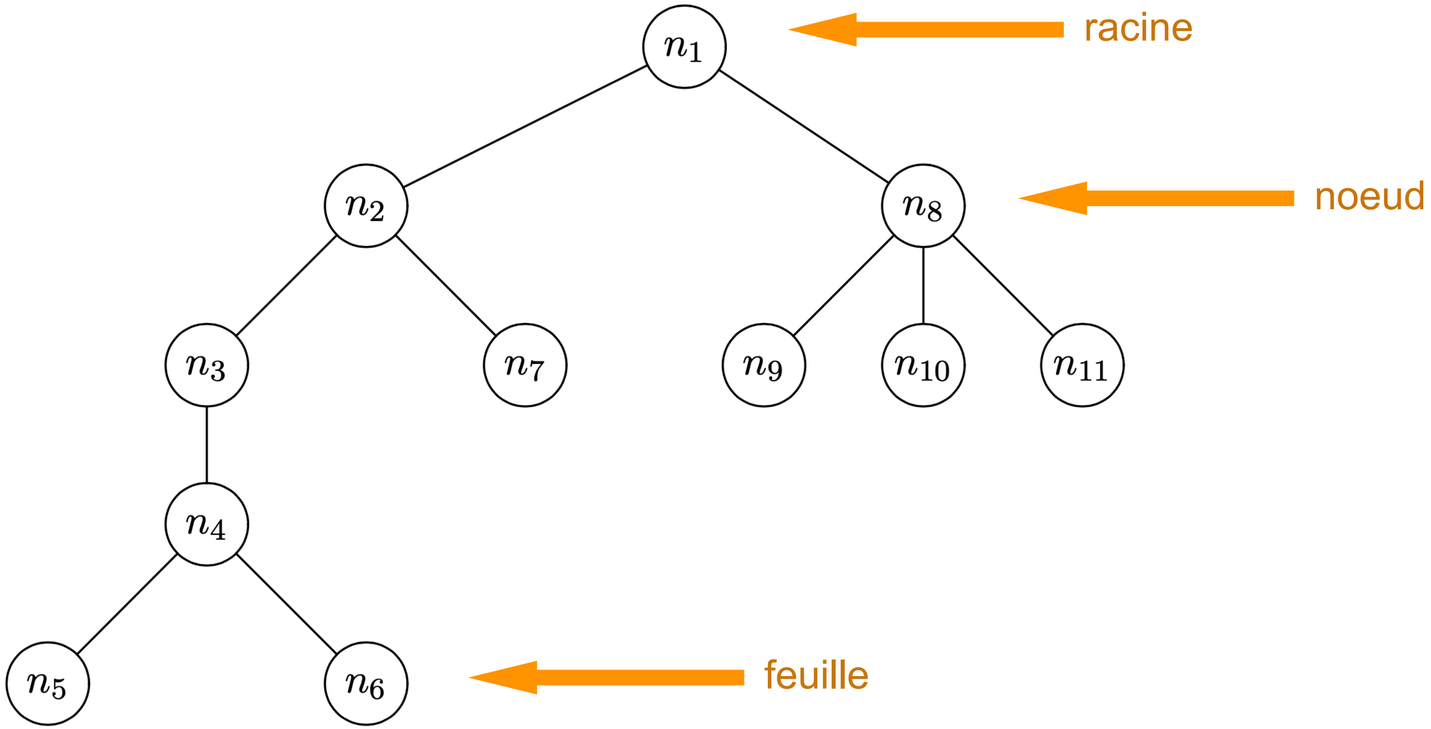
```
print (p1.__str__())
Point(10, 20)

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

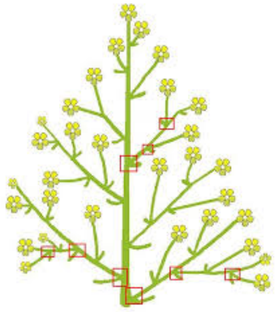
```
p2 = Point (30, 40)
print (p1.__add__ (p2))
Point(40, 60)

print (p1 + p2)
Point(40, 60)
```

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

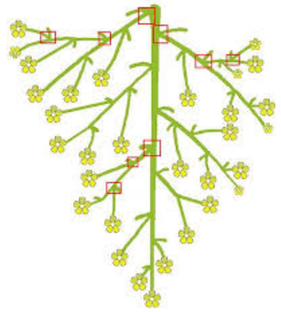
Les arbres en informatique

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





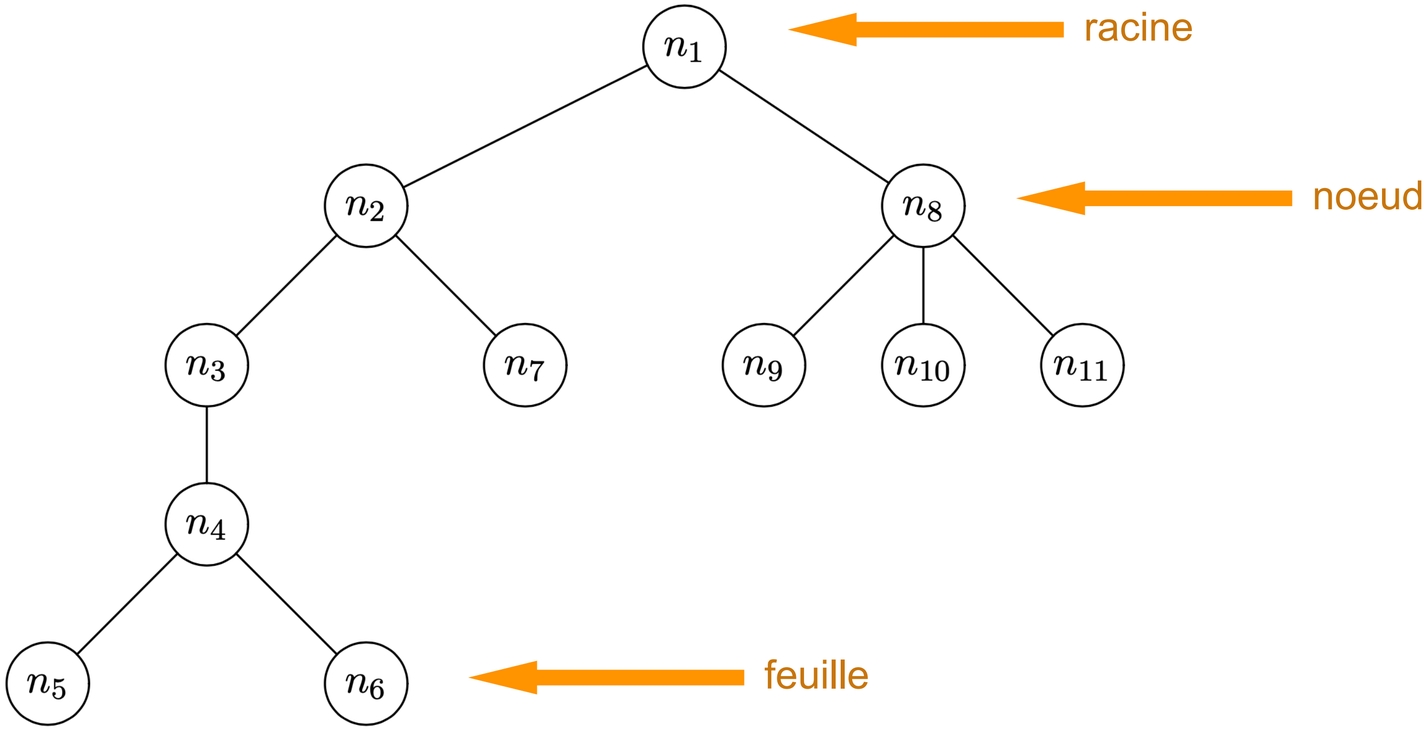
botanique

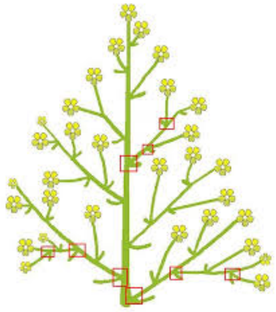


informatique

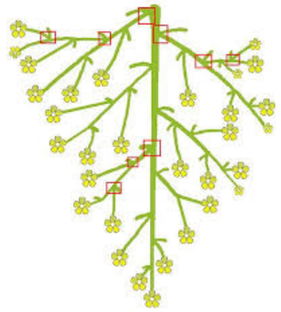
Les arbres en informatique

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





botanique



informatique

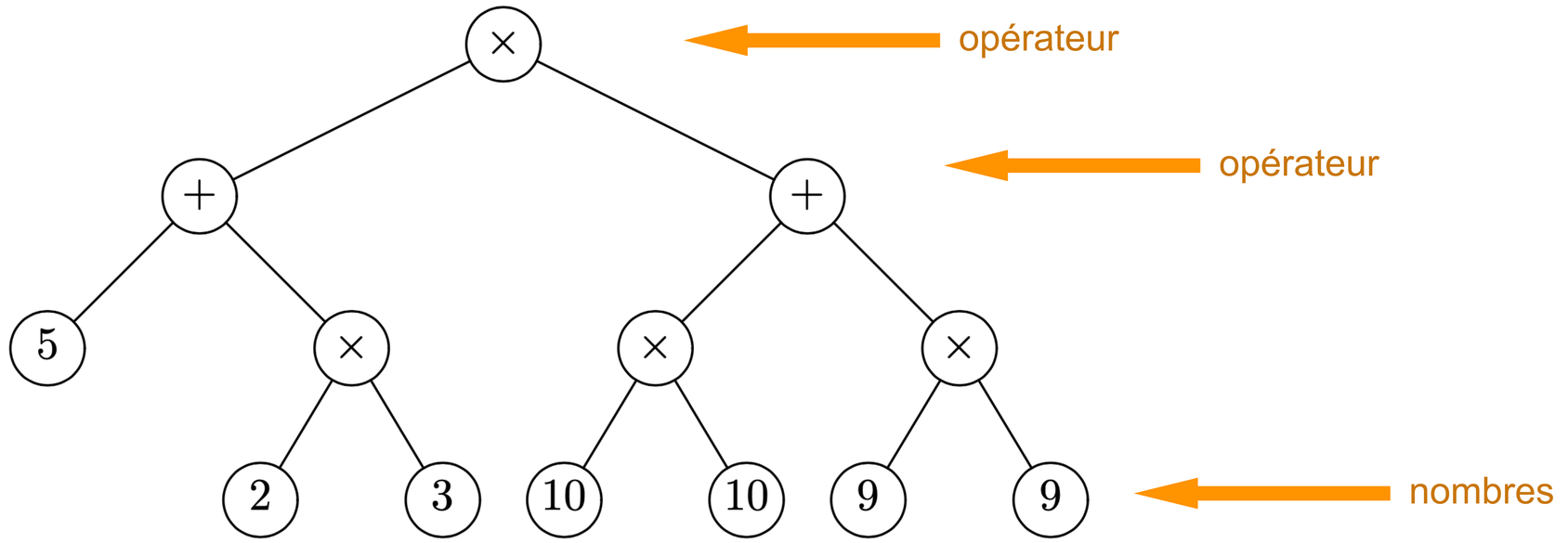
n2 est un **ancêtre** de n4

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]



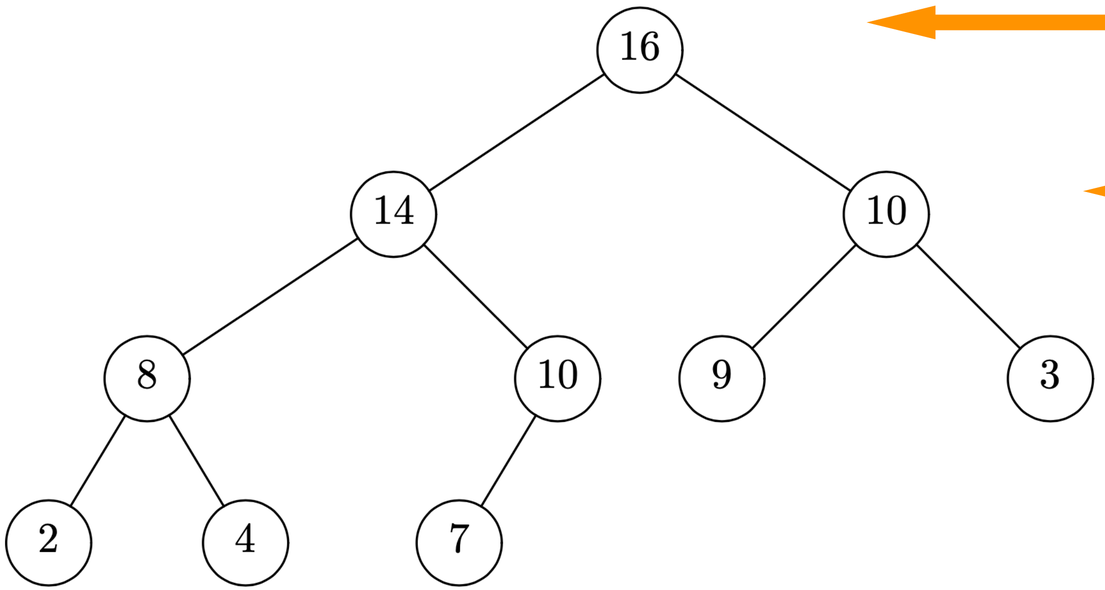
pour représenter une **expression arithmétique**

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

[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]

Représentation des arbres avec classes et objets

- on définit une classe avec des champs et des méthodes

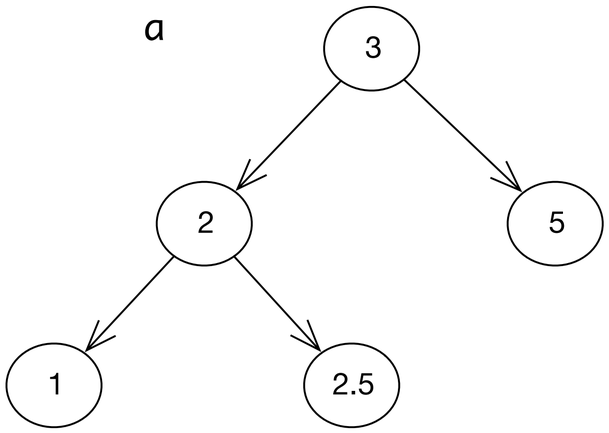
```
class Noeud:
    def __init__ (self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d
```

- 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))
```



les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Arbres (représentation 1)

• on définit une classe pour les **noeuds**

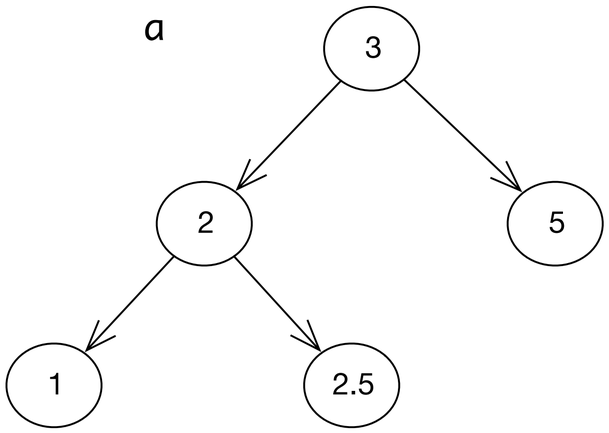
```
class Noeud:
    def __init__ (self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d
```

• 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))
```



les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

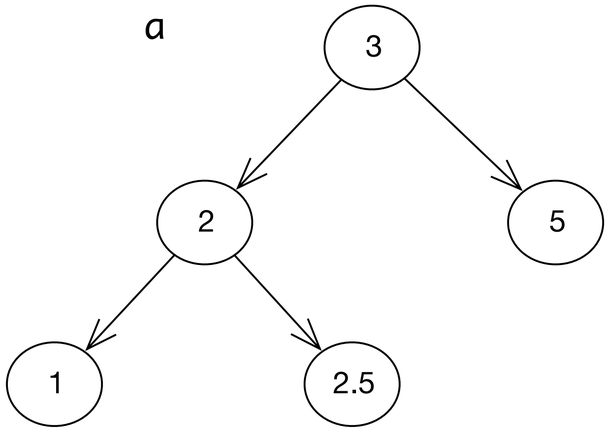
Arbres

- 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))
```

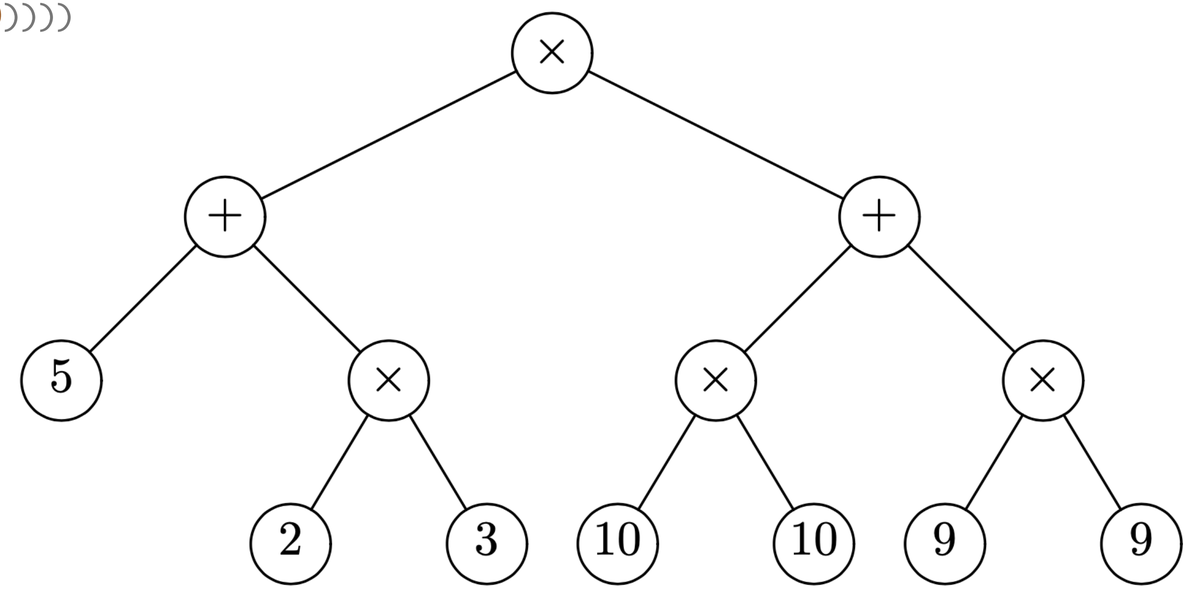


Arbres

- 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))))
```

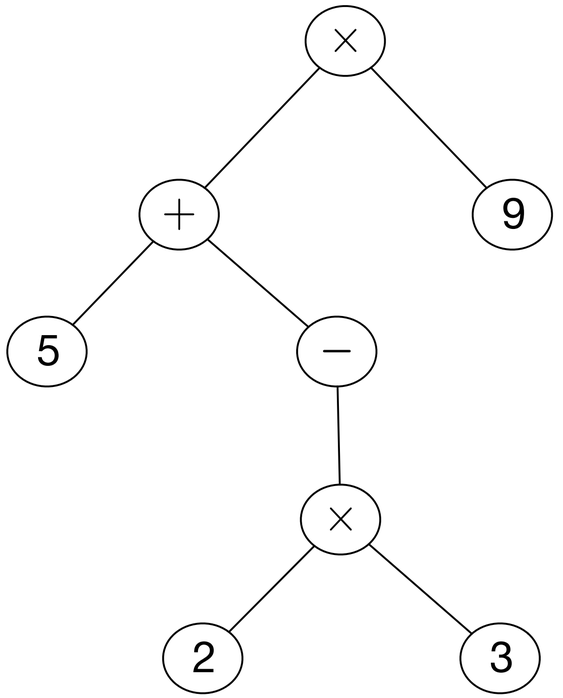
```
print (b.gauche.gauche)
Feuille (5)
print (b.gauche.droit)
Noeud ('*', Feuille (2), Feuille (3))
```



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
```



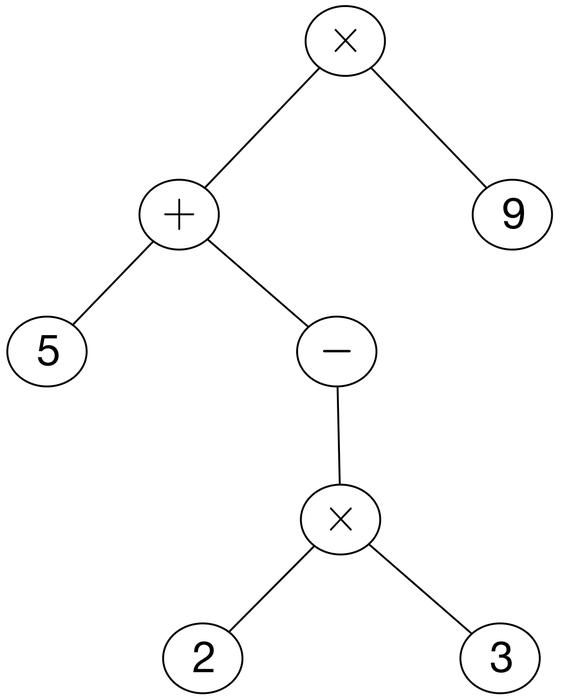
- et on construit l'arbre par:

```
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

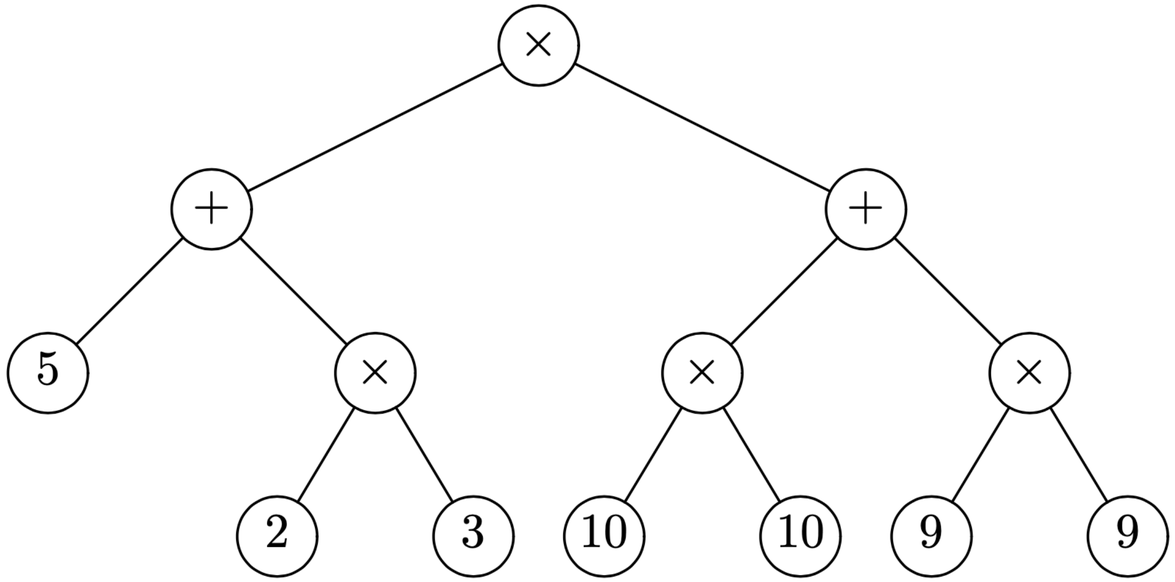
```
c = Noeud ('*', Noeud ('+', Noeud (5, None, None),
                            Noeud ('-', None,
                                        Noeud ('*', Noeud (2, None, None), Noeud (3, None, None)))),
                Noeud (9, None, None))
```

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)

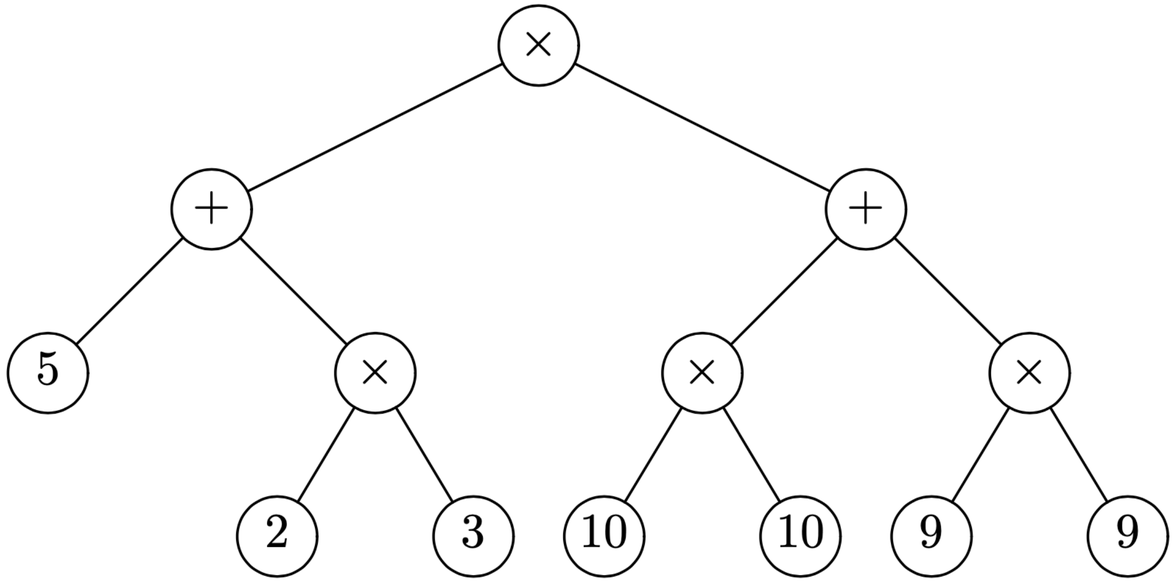
- On parcourt ou calcule sur les arbres avec des méthodes

```
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
```

- et on calcule les hauteur et taille

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

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

par cas sur sous-classes



Arbres

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

  [**tout est fonction (procédure)**]

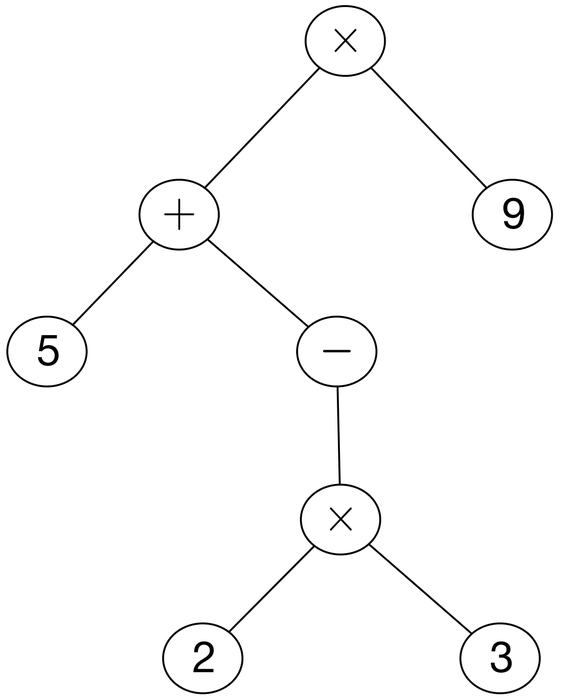
- les méthodes privilégient la programmation par objets

  [**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]
```

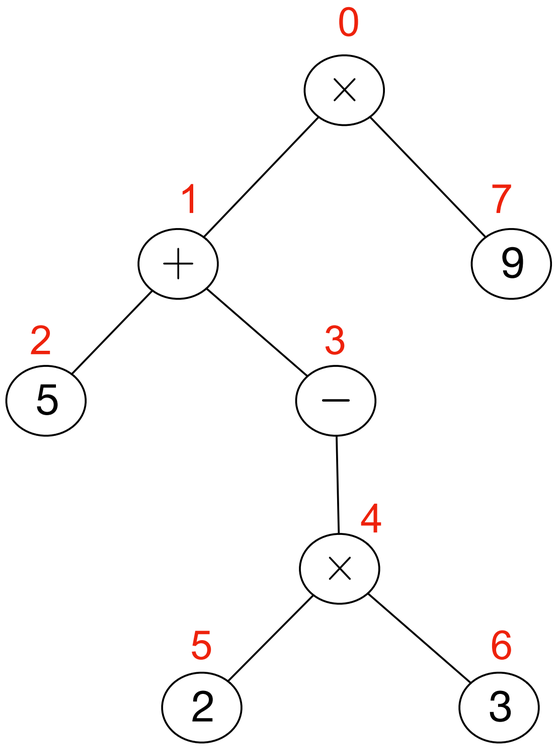
• la représentation plus souple car ne distinguant pas l'arité des noeuds

• 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

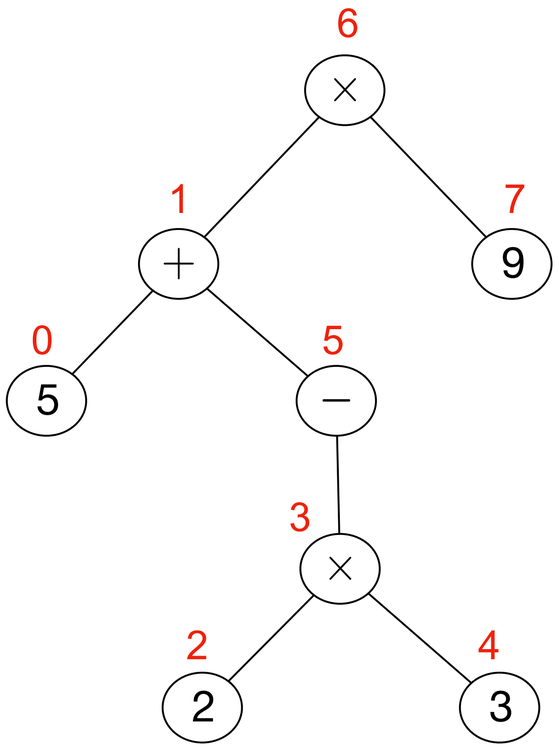
• 3 parcours d’arbre (préfixe, infixe, postfixe)



préfixe

• notation polonaise préfixe

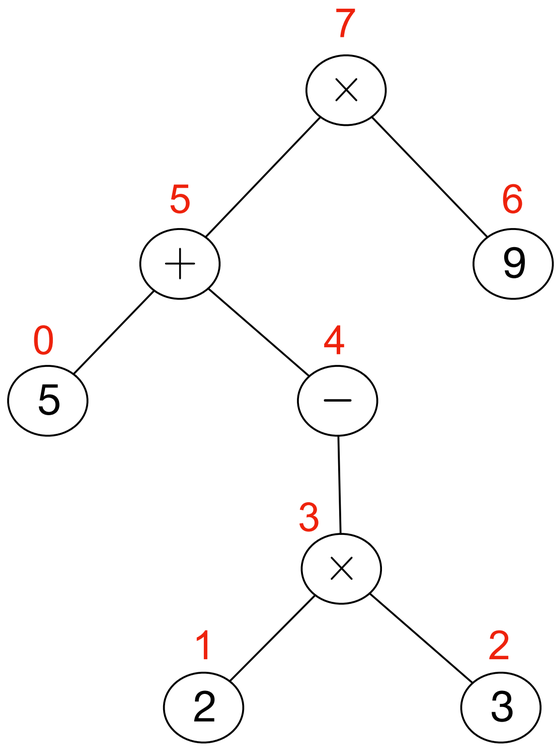
x + 5 - x 2 3 9



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)
```

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 !