Programmation et IA

Cours 5

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

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

Plan

- arbres
- arbres binaires de recherche
- programmation procédurale ou par objets
- programmation impérative ou fonctionnelle
- arbres binaires équilibrés

[texte des programmes]

Pour apprendre Python:

w3schools: tutoriel et réference

`https://www.w3schools.com/python/default.asp`

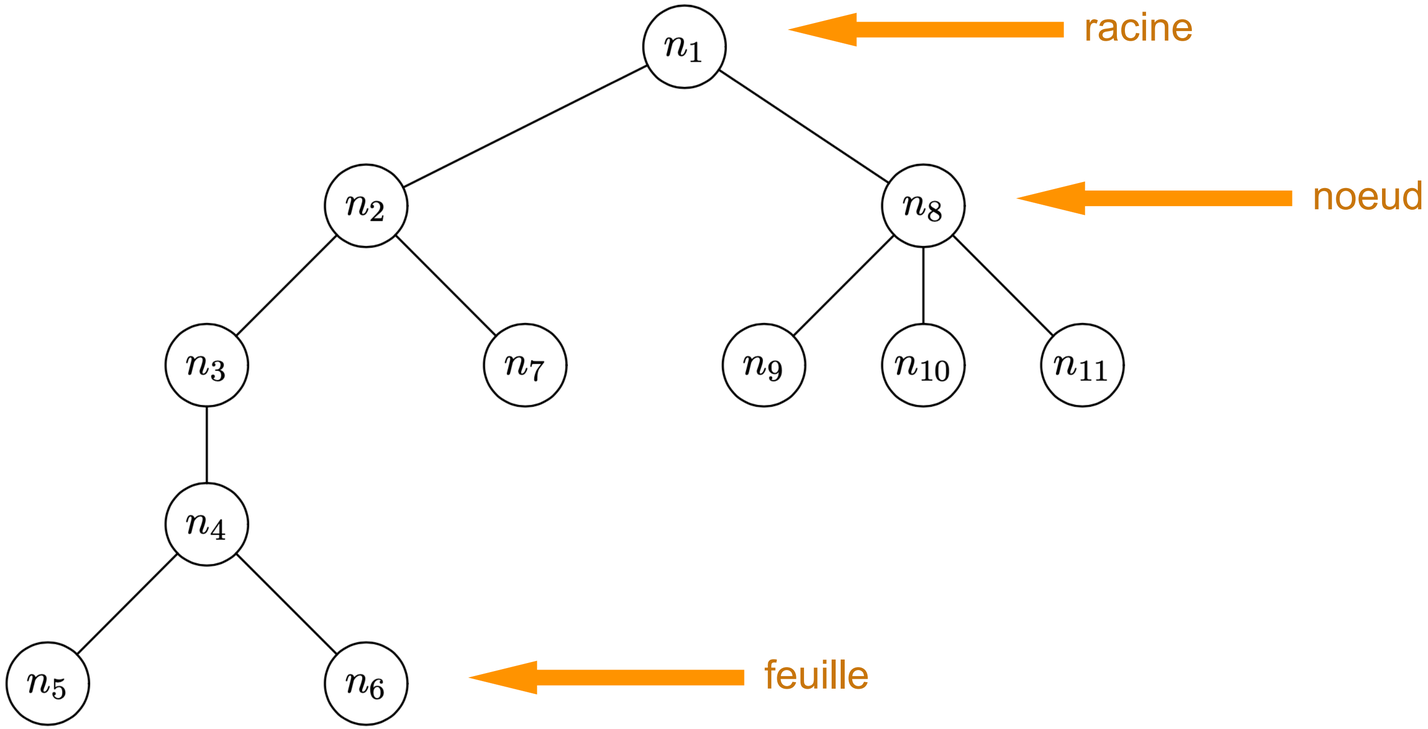
programiz: tutoriel et réference

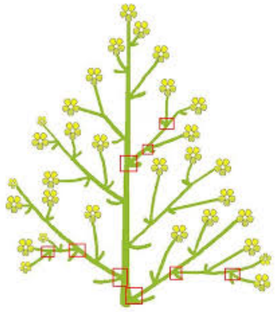
`https://www.programiz.com/python-programming`

les structures de données (2)

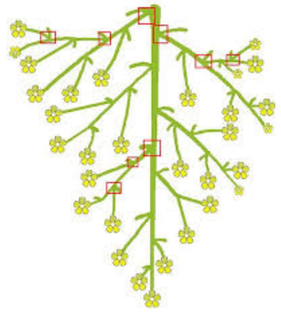
Les arbres en informatique

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





botanique



informatique

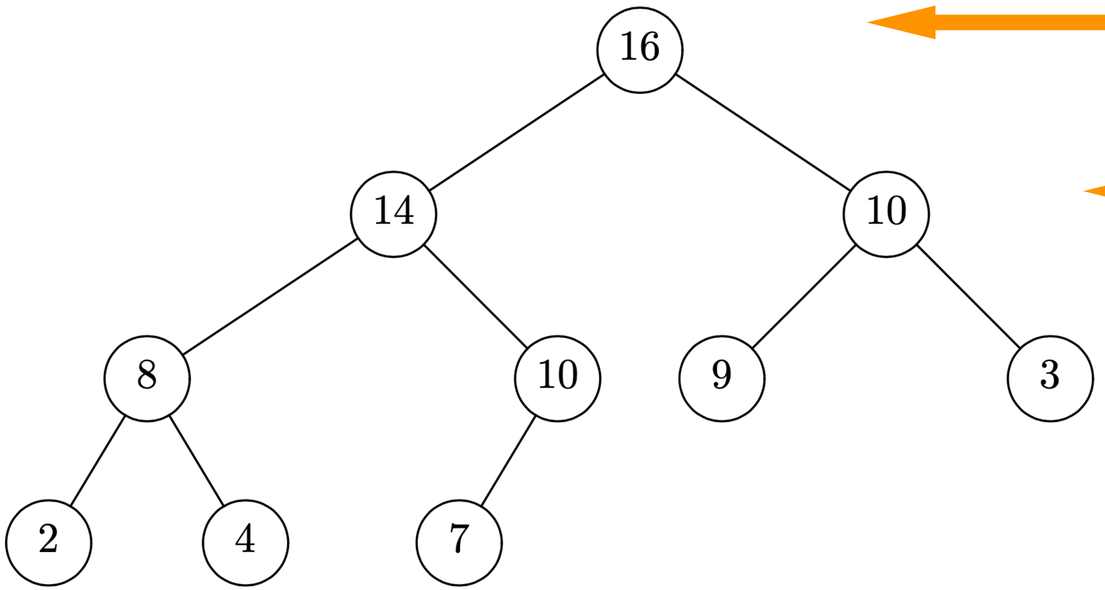
n2 est un **ancêtre** de n4

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

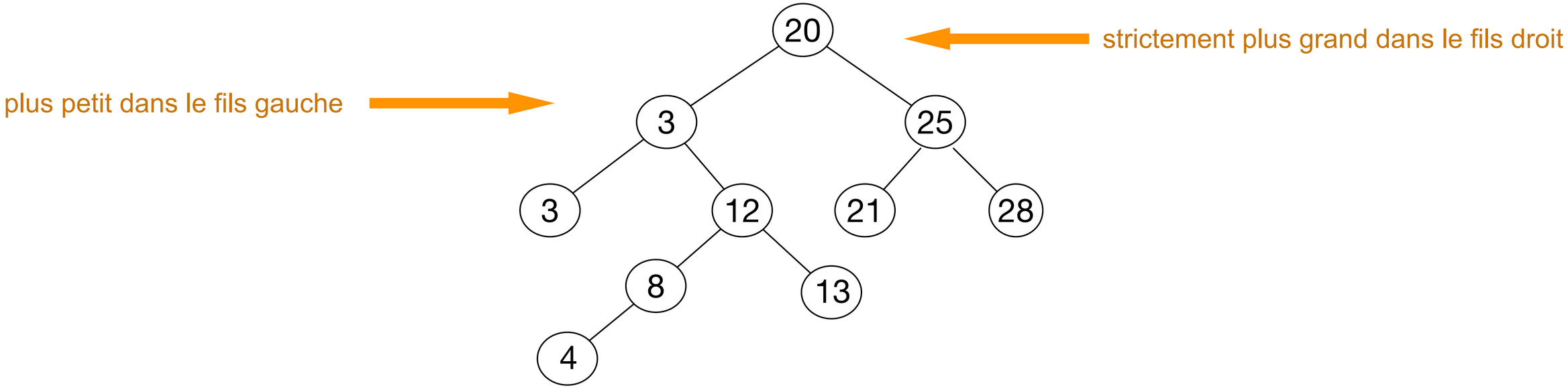
arbre binaire

tout noeud a au plus 2 fils

[arbre représentant une file de priorité]

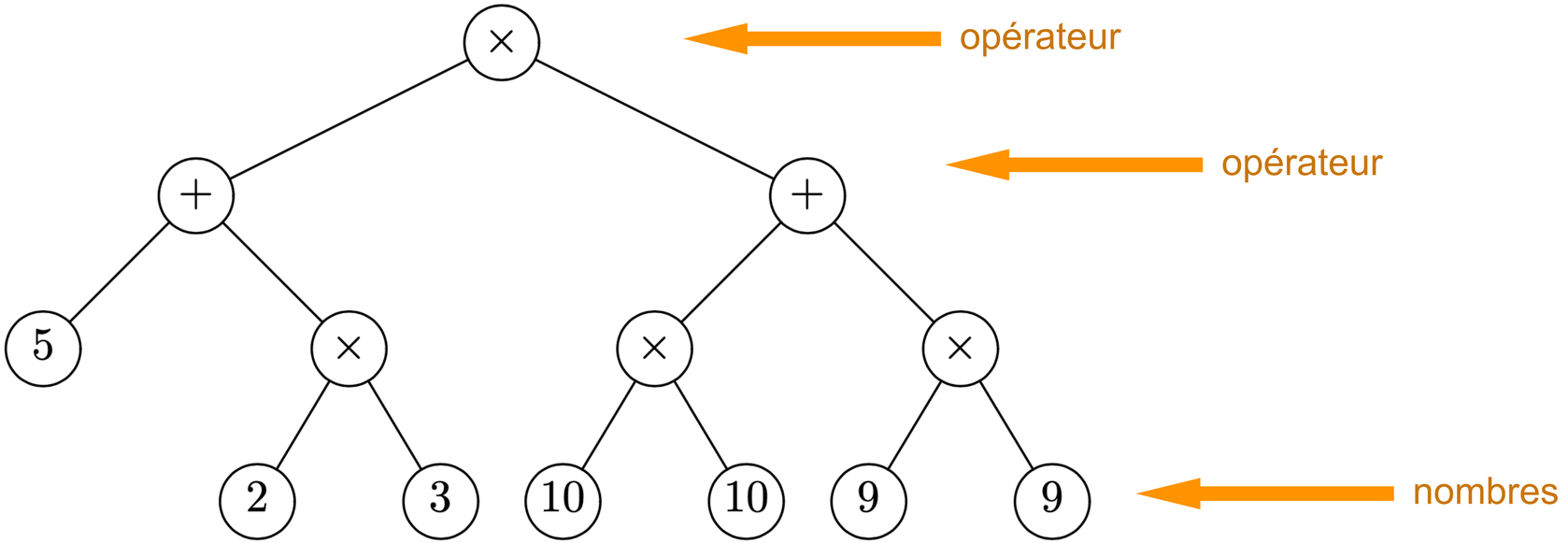
Les arbres en informatique

- arbres binaires de recherche (*BST - binary search tree*)



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]

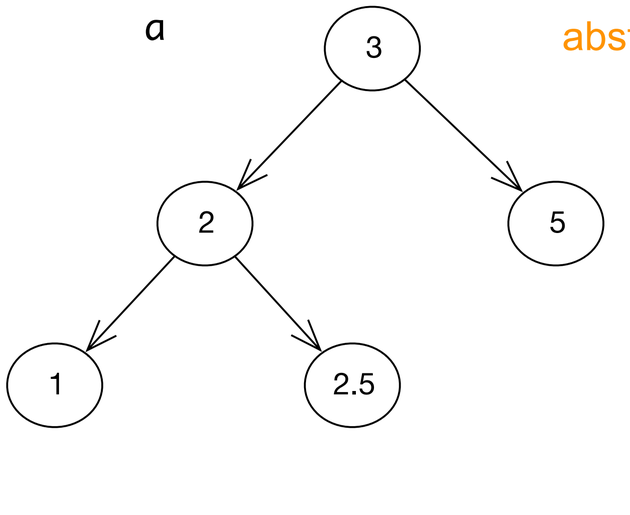
Représentation des arbres

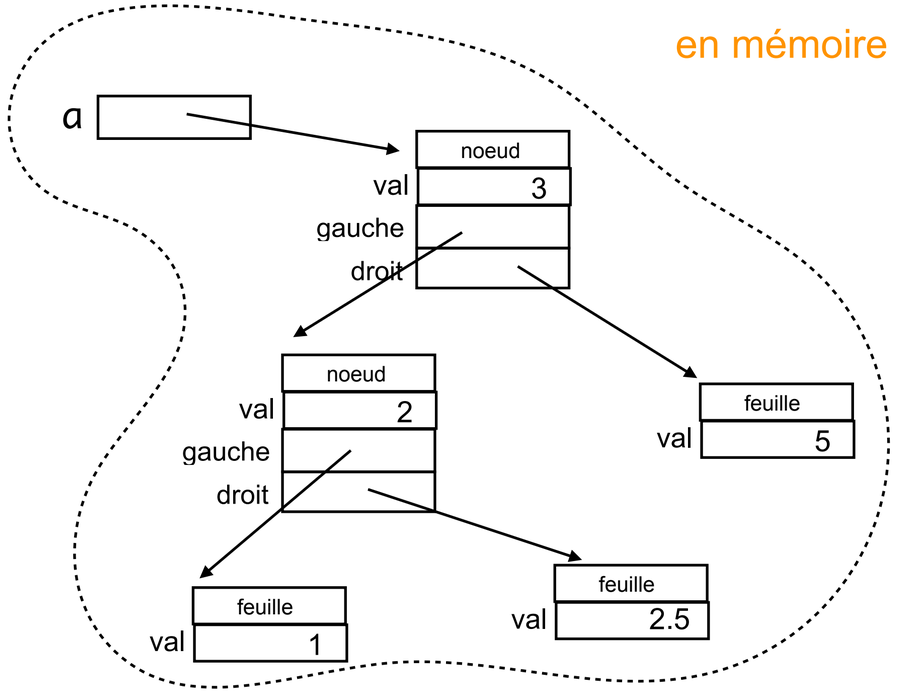
- on définit une classe pour les noeuds et pour les feuilles

```
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))
```





[texte des programmes]

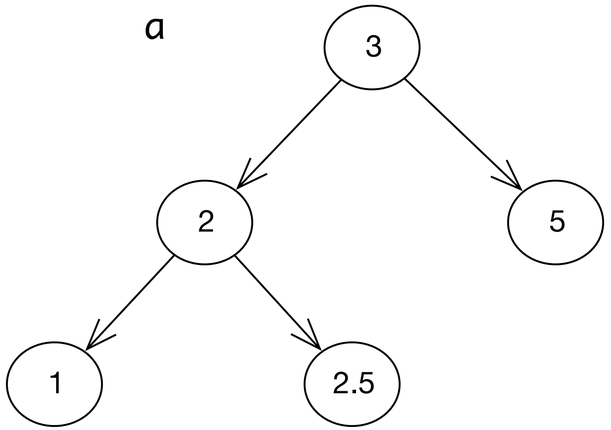
Représentation des arbres

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

```
class Noeud:
    . . .
    def __str__ (self) :
        return f'({self.gauche} <- {self.val} -> {self.droit})'


class Feuille:
    . . .

    def __str__ (self) :
        return f'{self.val}'
```

- on construit et imprime des arbres

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

print (a.droit)
5

print (a.gauche)
(1 <- 2 -> 2.5)
```



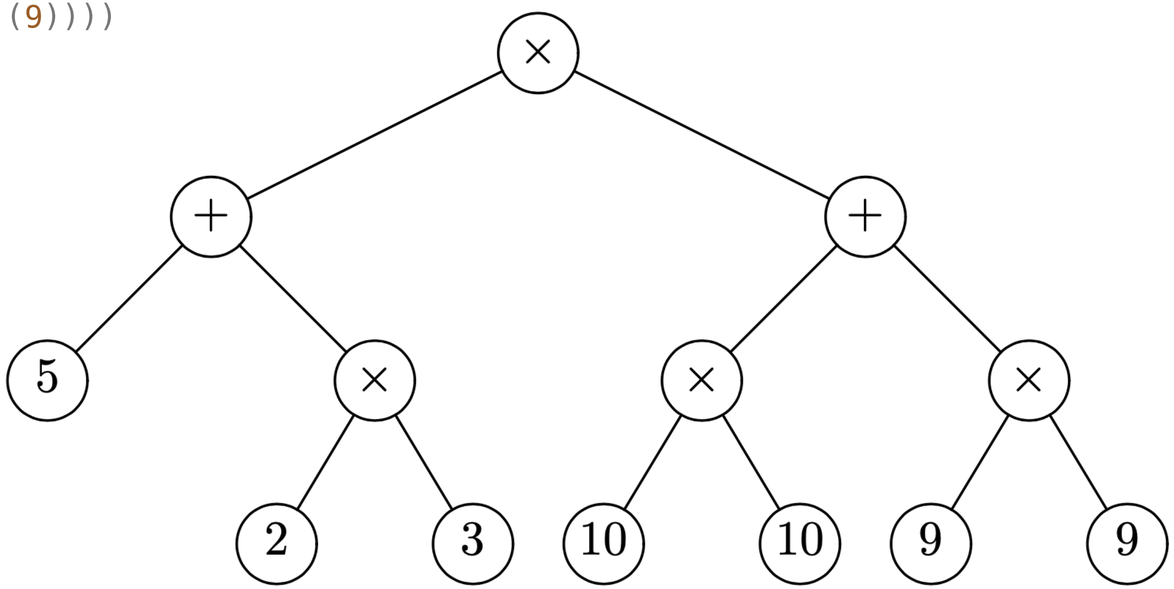
[texte des programmes]

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)
5
print (b.gauche.droit)
(2 <- * -> 3)
```



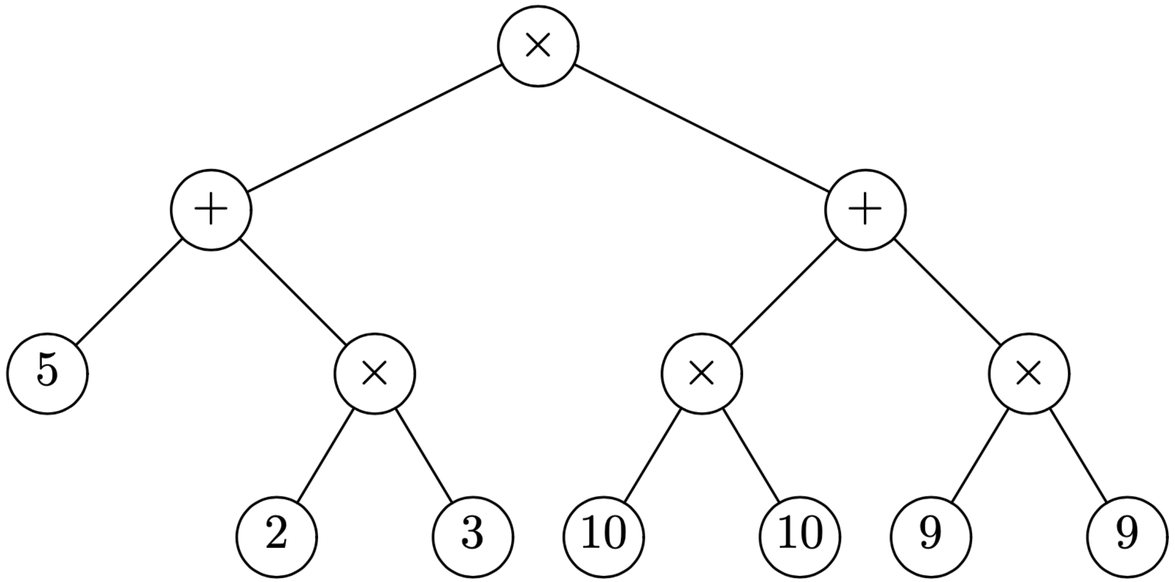
[texte des programmes]

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)
((5 <- + -> (2 <- * -> 3)) <- * -> ((10 <- * -> 10) <- + -> (9 <- * -> 9)))

hauteur (b)
3

taille (b)
13
```

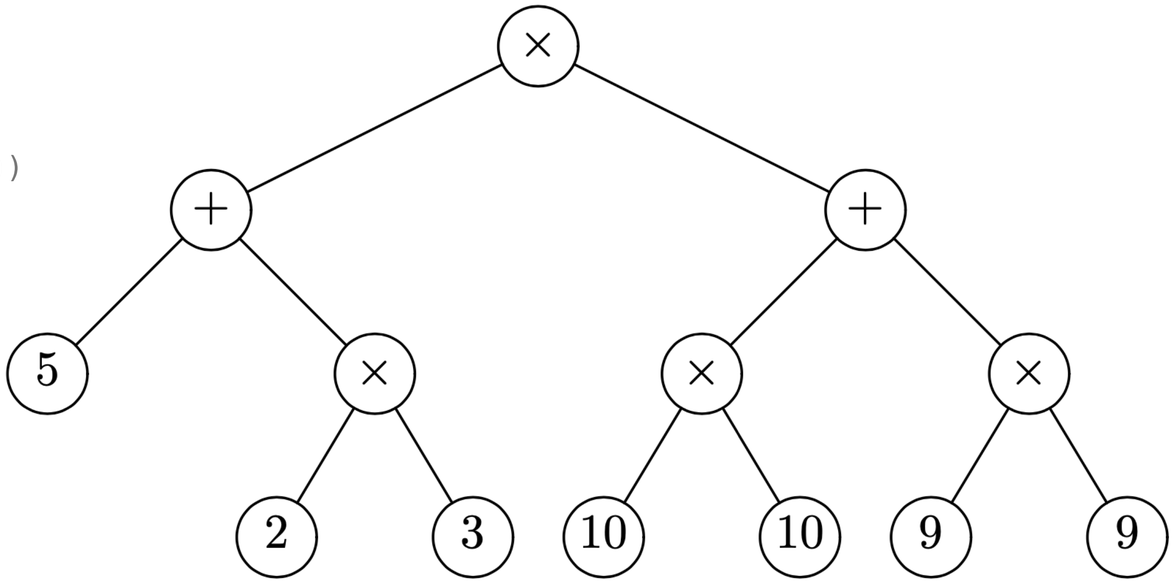
[texte des programmes]

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
```

[texte des programmes]

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

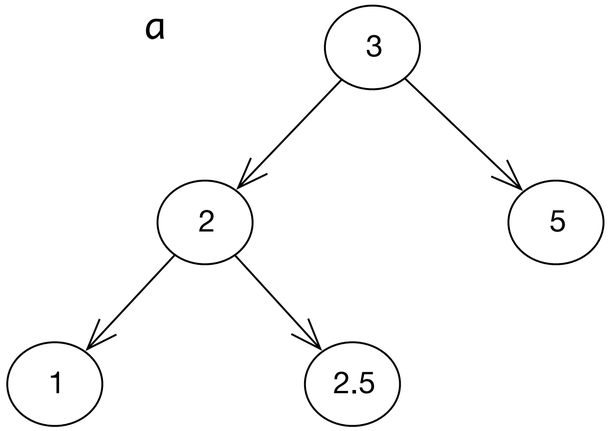
[texte des programmes]

Représentation des arbres (bis)

- parfois on simplifie la représentation des arbres binaires avec un seul type de noeuds

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

```
a = Noeud (3, Noeud (2, Noeud (1, None, None),
                        Noeud (2.5, None, None)),
              Noeud (5, None, None))
```



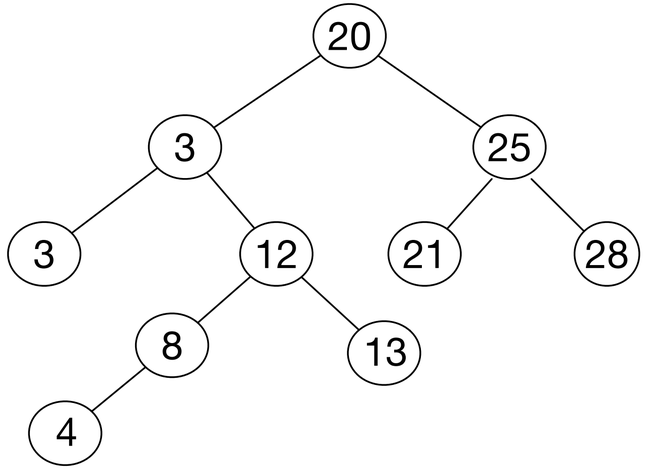
- alors une feuille est un noeud avec 2 fils None

[texte des programmes]

Arbres binaires de recherche

- but: maintenir un ensemble trié **dynamique**
- par exemple, une table de noms et de leurs numéros de téléphone

‘Awa’	‘Claire’	‘Henri’	‘Jeanne’	‘JJ’	‘Louis’	‘Orel’	‘Paul’
0611	0908	1215	0911	4729	1217	0801	0071



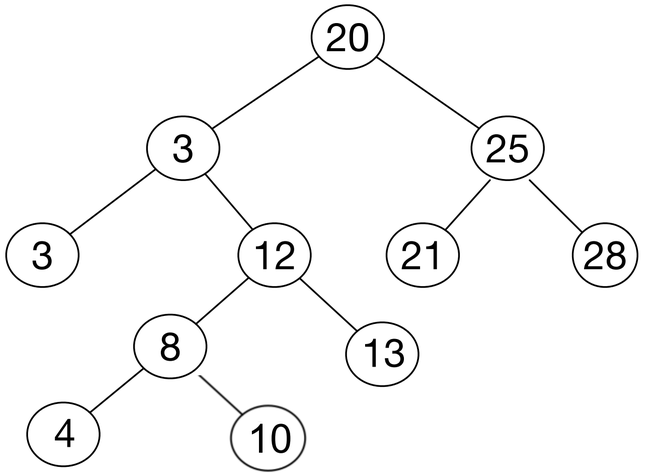
- ou plus simplement une liste ordonnée de nombres

3	3	4	8	12	13	20	21	25	28

- on veut rajouter ou supprimer un élément en gardant l’ordre

3	3	4	8	**10**	12	13	20	21	25	28

- on utilise un arbre binaire de recherche



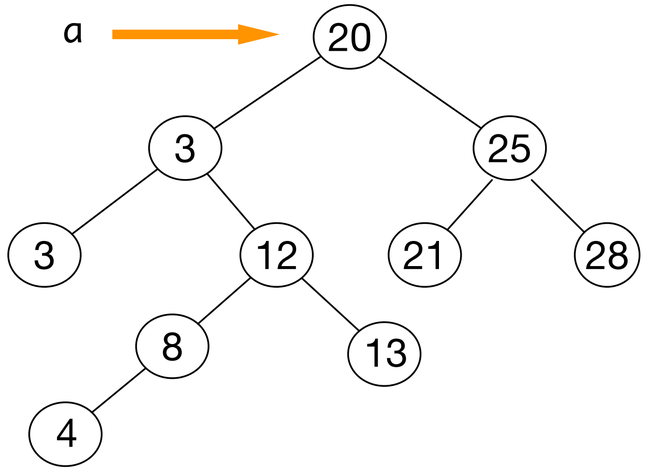
[texte des programmes]

Arbres binaires de recherche

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

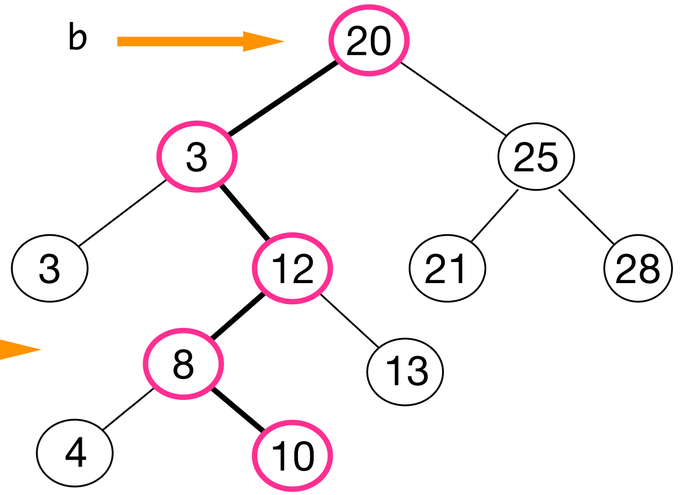
programme plus simple avec un
seul type de noeud

```
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



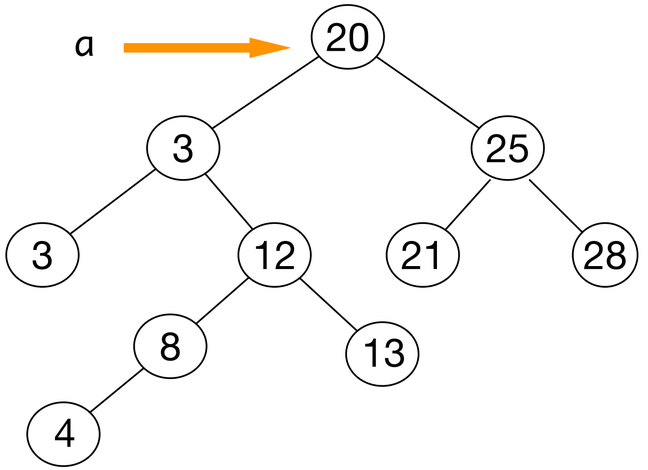
[texte des programmes]

Arbres binaires de recherche

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

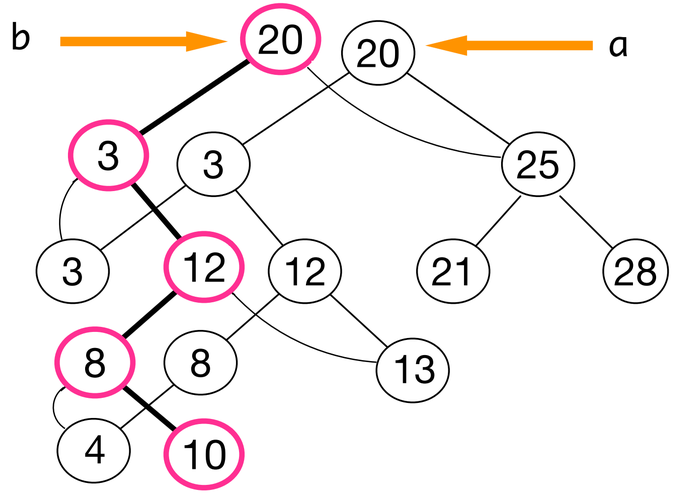
programme plus simple avec un
seul type de noeud

```
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



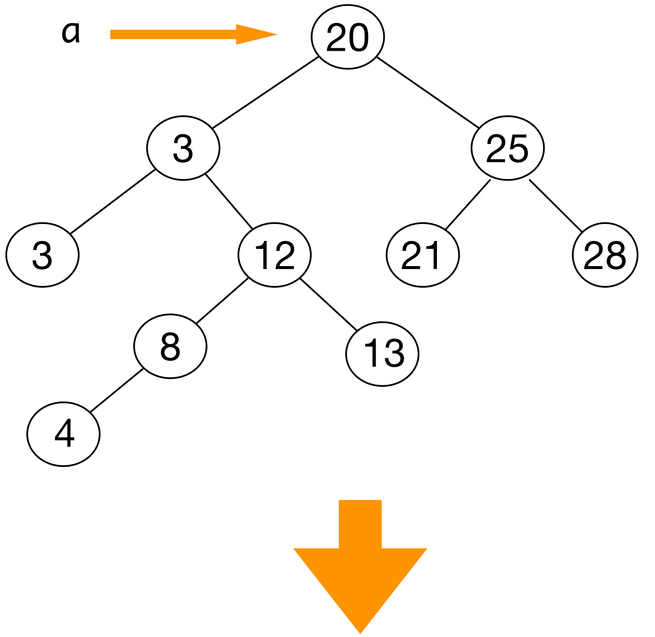
[texte des programmes]

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
```

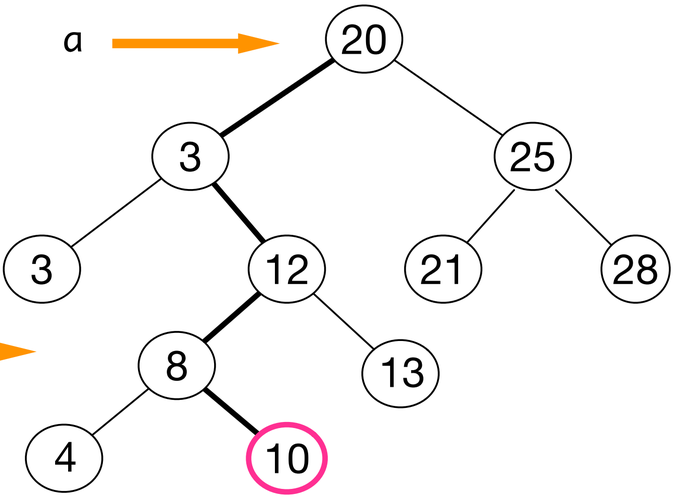


• on modifie l'arbre a [**« effet de bord »**]

DANGER ! DANGER !

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

le fils droit du noeud 8 est modifié



[texte des programmes]

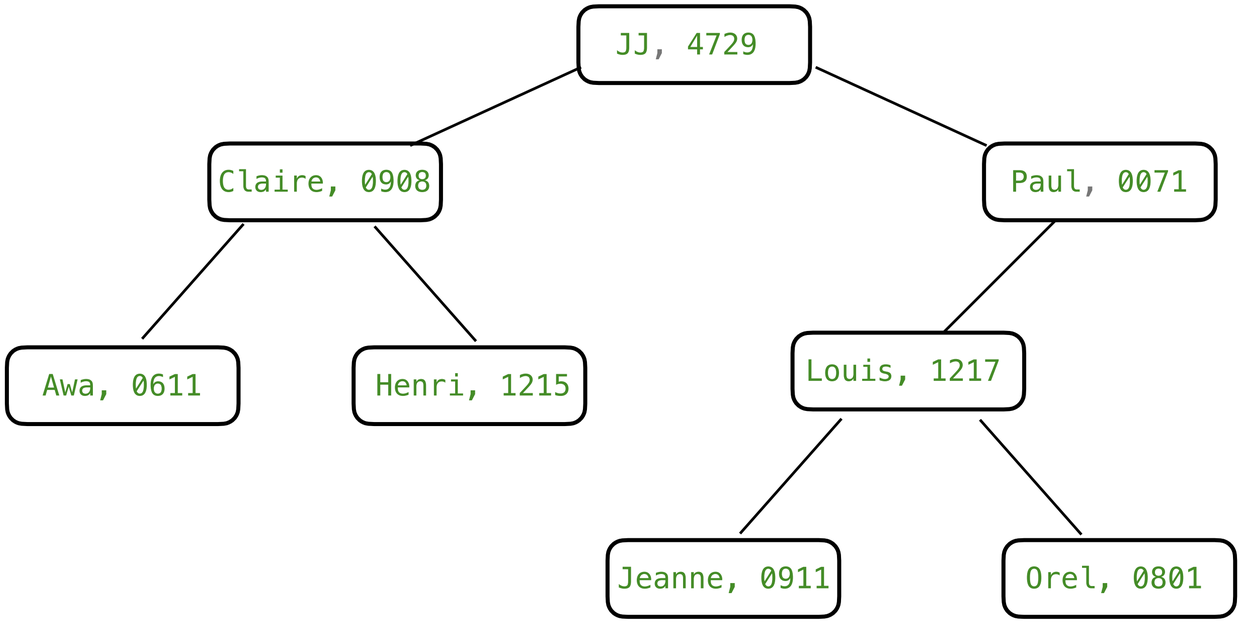
Arbres binaires de recherche

- création de la table

```
a = None
for x in [('JJ', '4729'), ('Paul', '0071'), ('Claire', '0908'), ('Henri', '1215'),
        ('Awa', '0611'), ('Louis', '1217'), ('Jeanne', '0911'), ('Orel', '0801')] :
    a = ajouter (x, a)

print(a)
```

- recherche d'une clé

```
def rechercher (k, a) :
    if a == None :
        raise KeyError (f'Clé {k} non trouvée')
    elif k == a.val[0] :
        return a.val[1]
    elif k < a.val[0] :
        return rechercher (k, a.gauche)
    else :
        return rechercher (k, a.droit)
```

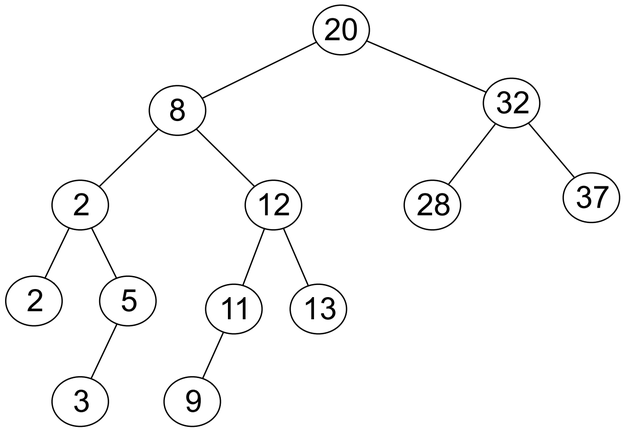



```
print (rechercher ('Claire', a))
print (rechercher ('Orel', a))
```

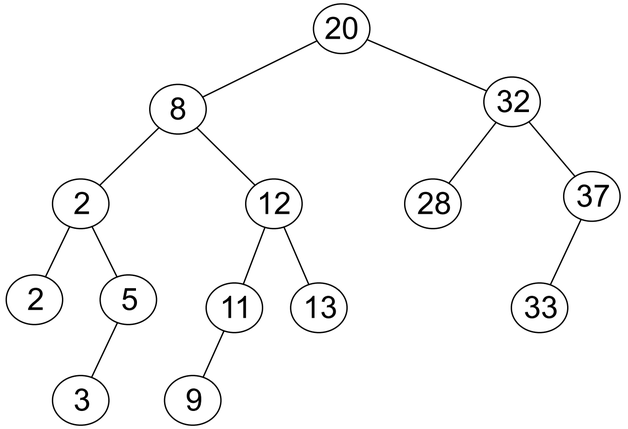
[texte des programmes]

Arbres binaires de recherche

- l'ajout d'un nouveau élément se fait sur une feuille
- la recherche et l'ajout dans un arbre binaire de recherche fait moins de `h` opérations où `h` est la **hauteur** de l'arbre
- la hauteur est `log(n)` pour un arbre de taille `n` si l'arbre binaire est **parfait**
- il faut donc veiller à ce que l'arbre de recherche soit **bien équilibré** pour que la recherche fasse `log(n)` opérations
- pour tout noeud, on veut $-1 \leq \text{hauteur(droit)} - \text{hauteur(gauche)} \leq 1$



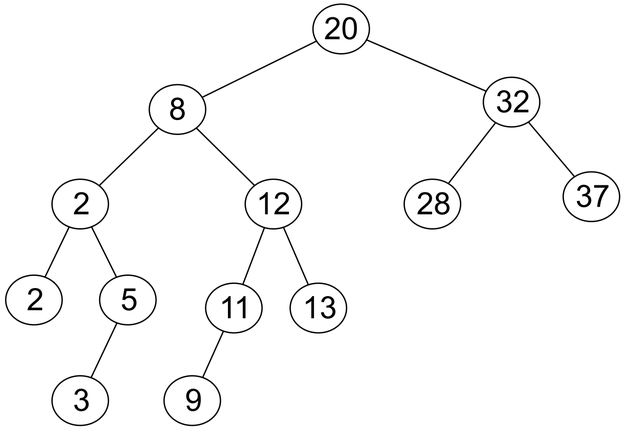
arbre non équlibré



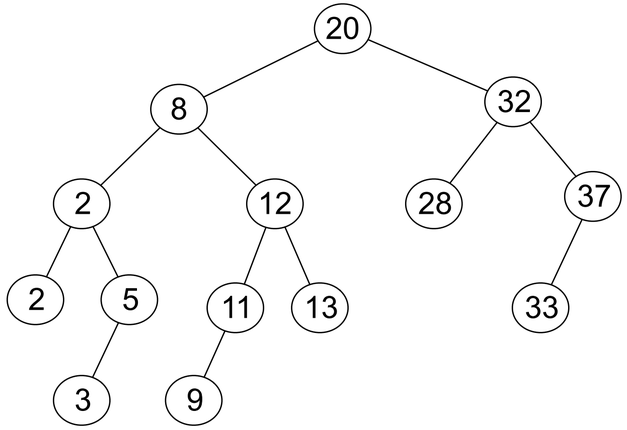
arbre bien équilibré

Arbres binaires de recherche

Exercice: Générer une liste de 20 nombres aléatoires entre 0 et 99 et les ranger dans un arbre binaire de recherche. Tester si cet arbre est bien équilibré. Si non équilibré, recommencer…



arbre non équlibré



arbre bien équilibré

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)

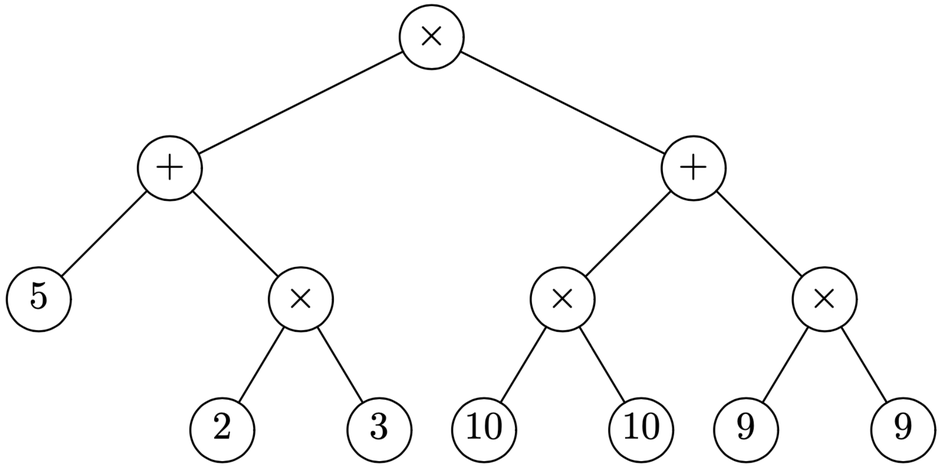
données non modifiables

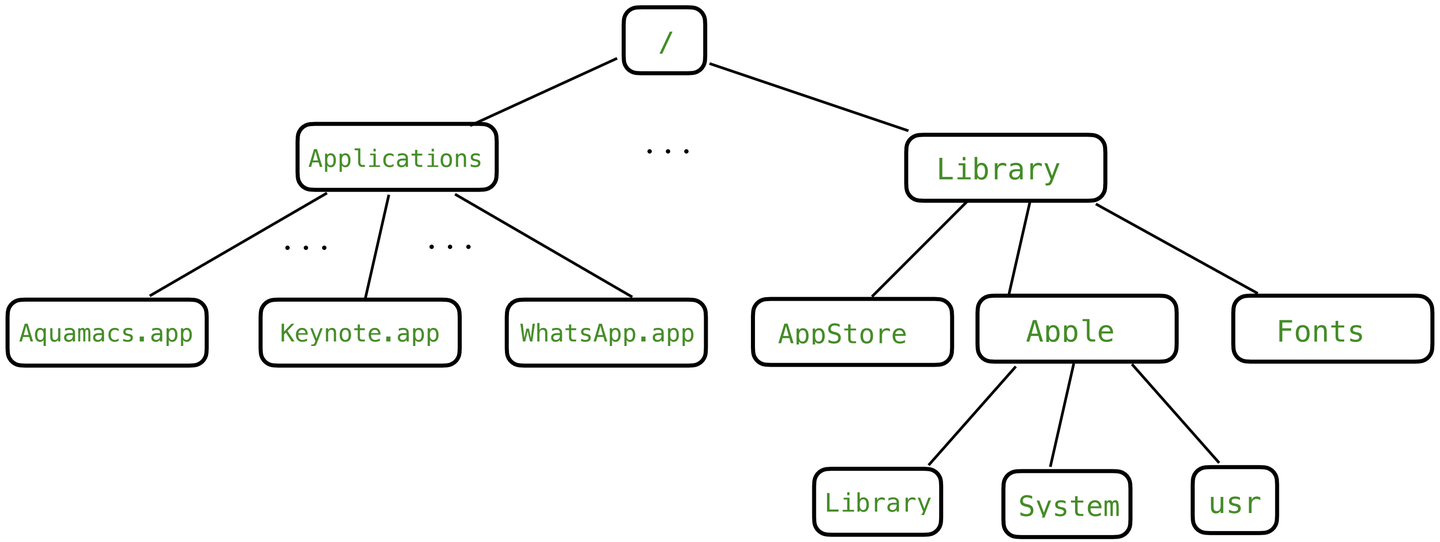
- programmation impérative
  - on fait des effets de bord sur les arbres
  - on modifie donc leur structure
  - on optimise la place mémoire en ne créant pas de nouveaux noeuds
  - danger… danger !!

données modifiables

Au-delà des arbres de recherche

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

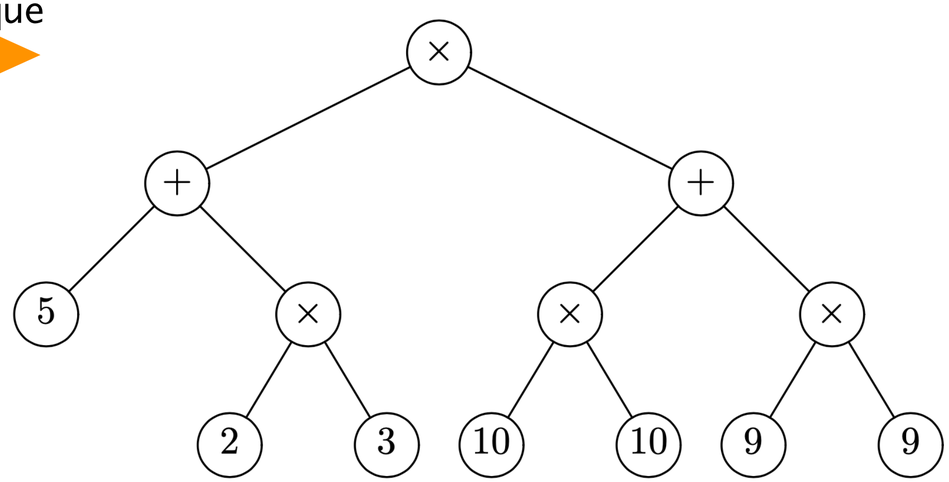
chaîne de caractères

(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

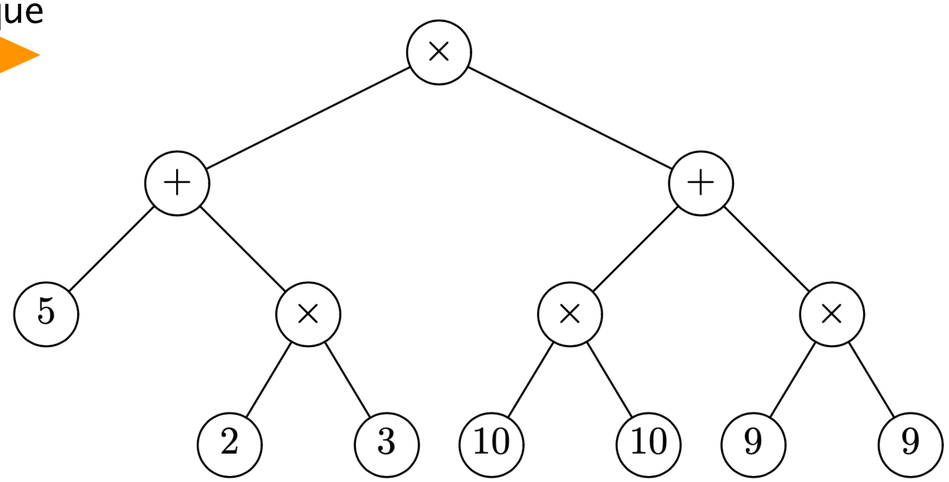
chaîne de caractères

(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}
```

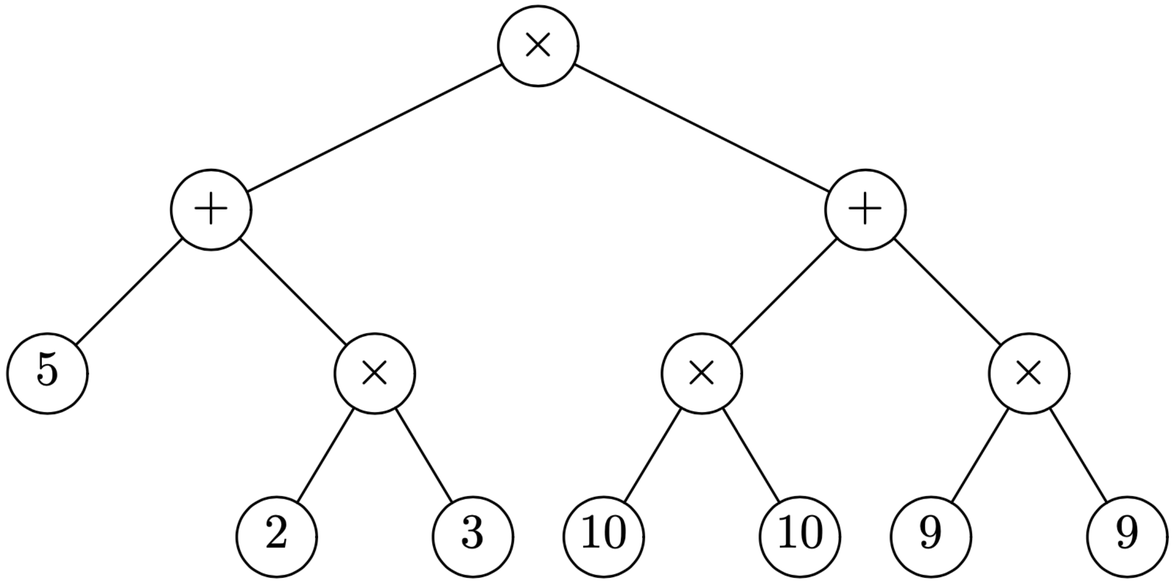
Arbres de syntaxe abstraite (ASA)

- évaluation d'une expression arithmétique

```
BINOP = Noeud
ELT = Feuille

def eval (t, e) :
    if isinstance (t, BINOP) :
        if t.val == '+' :
            return eval(t.gauche, e) + eval(t.droit, e)
        elif t.val == '-' :
            return eval(t.gauche, e) - eval(t.droit, e)
        elif 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 ('BIN_OP impossible')
    elif isinstance (t, ELT) :
        return e[t.val] if isinstance (t.val, str) else t.val
    else :
        raise Exception ('Terme mal formé')

e = {'x' : 4, 'y' : 5, 'z' : 6}
print (b)
print (e)
print (eval (b, e))
```

```
((5 <- + -> (2 <- * -> 3)) <- * -> ((10 <- * -> 10) <- + -> (9 <- * -> 9)))
{'x': 4, 'y': 5, 'z': 6}
1991
```

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



[texte des programmes]

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.

Prochain cours

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