Informatique et Programmation

Cours 8

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

Plan

- classes et objets
- implémentation des arbres
- arbre binaire de recherche
- ajout dans un arbre binaire de recherche
- arbres équilibrés AVL
- arbres 2-3-4
- arbres rouge-noir

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

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 "Point (%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

Classes et objets

- files d'attente sous forme de classe et objets avec **attributs** et **méthodes**

```
class FIFO :
    def __init__(self) :
        self.contents = []

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

    def ajouter (self, x) :
        self.contents.append (x)

    def retirer (self) :
        if len (self.contents) == 0 :
            raise ValueError
        res = self.contents[0]
        del self.contents[0]
        return res
```

- objets dans cette classe

```
f = FIFO()
f.ajouter ('JJ')
f.ajouter ('Paul')
f.ajouter ('Claire')
f.ajouter ('Henri')
print (f)
for i in range (10) :
    r = f.retirer()
    print (r, '..', f)
```

- représentation plus modulaire des filles d'attente

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

Classes et objets

- files de priorité sous forme de classe et objets avec **attributs** et **méthodes**

```
class PriorityQueue :
    def __init__(self) :
        self.contents = []

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

    def ajouter (self, x) :
        self.contents.append (x)

    def retirer (self) :
        if len (self.contents) == 0 :
            raise ValueError
        i = index_of_max (self.contents)
        res = self.contents[i]
        del self.contents[i]
        return res
```

```
f = PriorityQueue()
f.ajouter ((10,'JJ'))
f.ajouter ((4,'Paul'))
f.ajouter ((7, 'Claire'))
f.ajouter ((11, 'Henri'))
f.ajouter ((1, 'Jeanne'))
f.ajouter ((9, 'Louis'))
f.ajouter ((20, 'Awa'))
f.ajouter ((18, 'Orel'))

print (f)
for i in range (10) :
    r = f.retirer()
    print (r, '..', f)
```

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

Classes et objets

- files de priorité sous forme de classe et objets avec **attributs** et **méthodes**

```
class PriorityQueue :
    def __init__(self) :
        self.contents = []

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

    def ajouter (self, x) :
        a = self.contents; a.append(x)
        n = len (a); i = n - 1
        while i > 0 and a[(i-1) // 2] < x :
            a[i] = a[(i-1) // 2]
            i = (i-1) // 2
        a[i] = x
```

```
def retirer (self) :
        a = self.contents; n = len (a)
        try:
            res = a[0]
            v = a[0] = a[n-1]
            i = 0
            while 2*i + 1 < n-1 :
                j = 2*i + 1
                if j + 1 < n-1 :
                    if a[j+1] > a[j] :
                        j = j + 1
                if v >= a[j] :
                    break
                a[i] = a[j]; i = j
            a[i] = v
            del a[n-1]
            return res
        except Exception:
            print ('File vide')
```

- représentation plus abstraite des filles d'attente

- l'utilisation extérieure est identique

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

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`

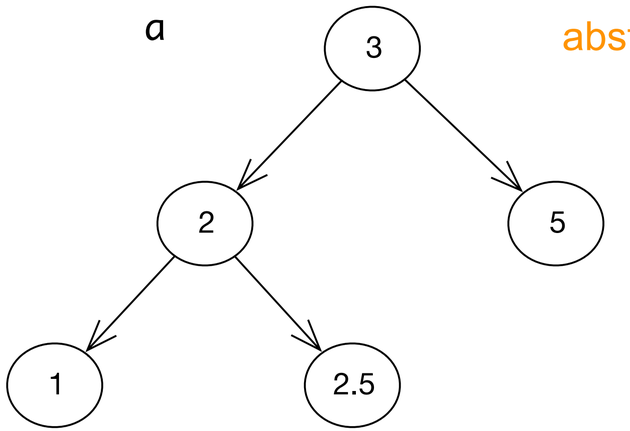
Représentation des arbres

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

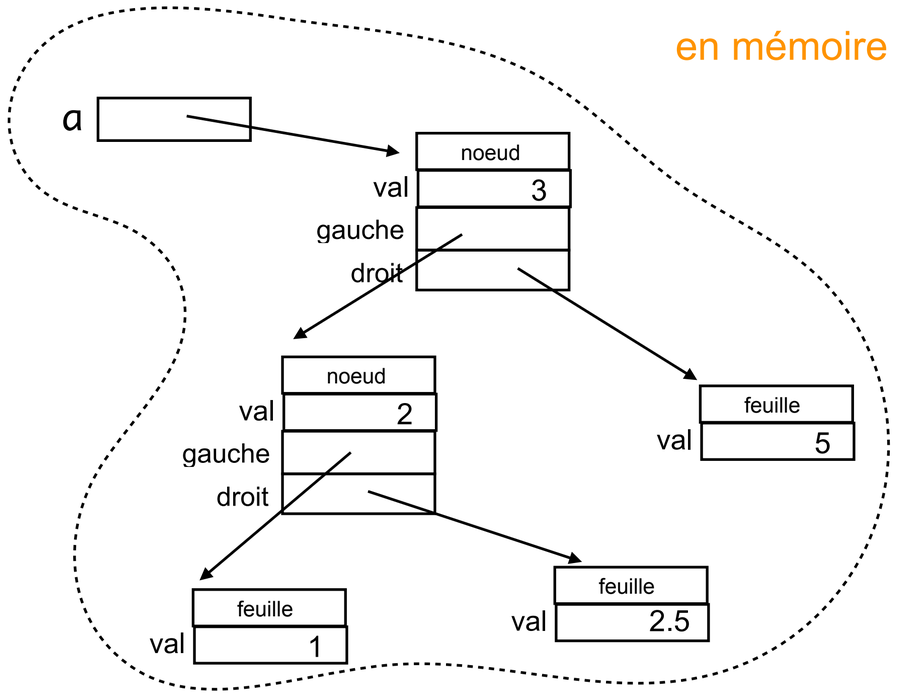
```
class Noeud:
    def __init__ (self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d

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



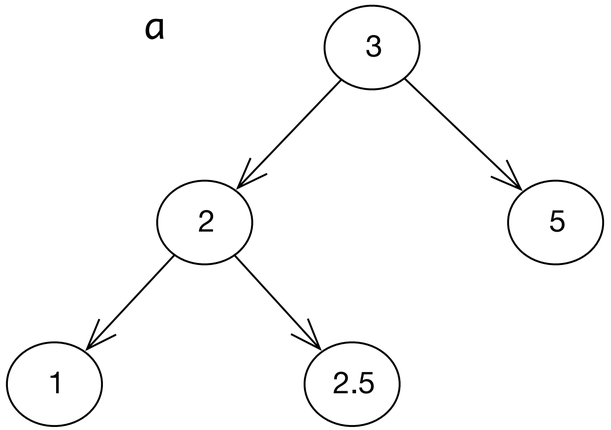
Représentation des arbres

- 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))
```

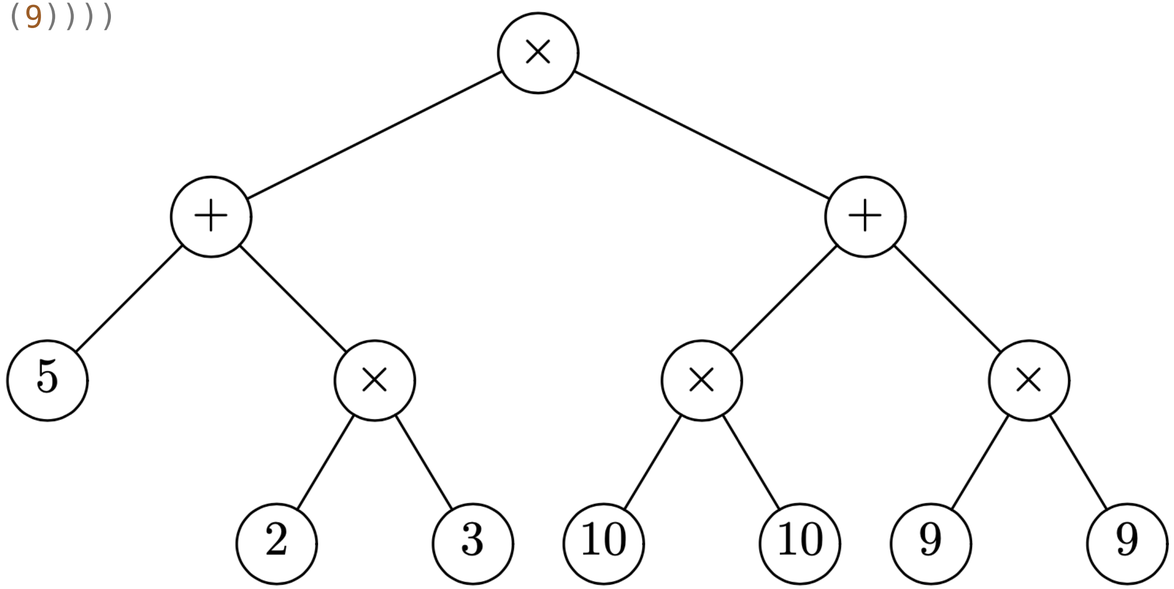


Représentation des arbres

- 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))))
```

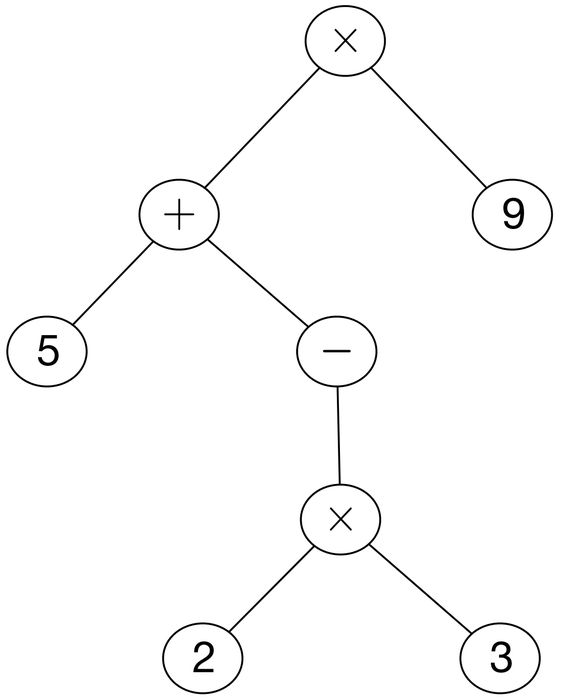
```
print (b.gauche.gauche)
Feuille (5)
print (b.gauche.droit)
Noeud ('*', Feuille (2), Feuille (3))
```



Représentation des arbres

- Autre représentation avec l'arbre vide (0 neuds, 0 feuilles) ici implémenté par None

```
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))
```

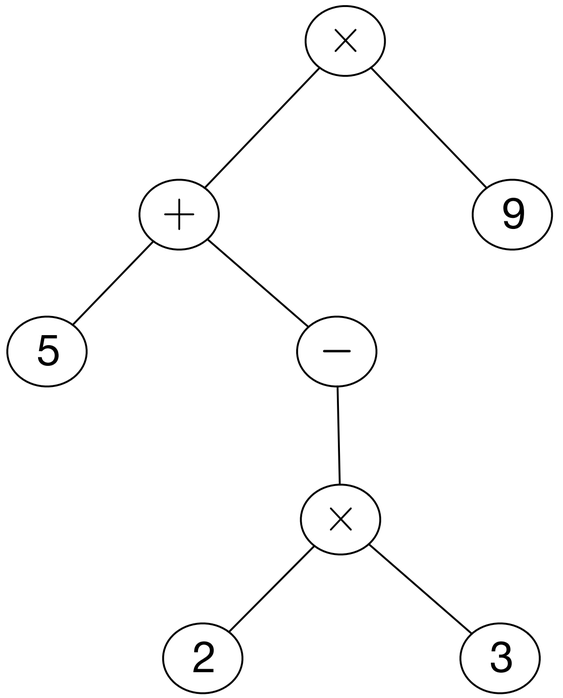
- ou encore en identifiant feuilles et noeuds sans fils

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

Représentation des arbres

- dans un autre représentaiton, 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:

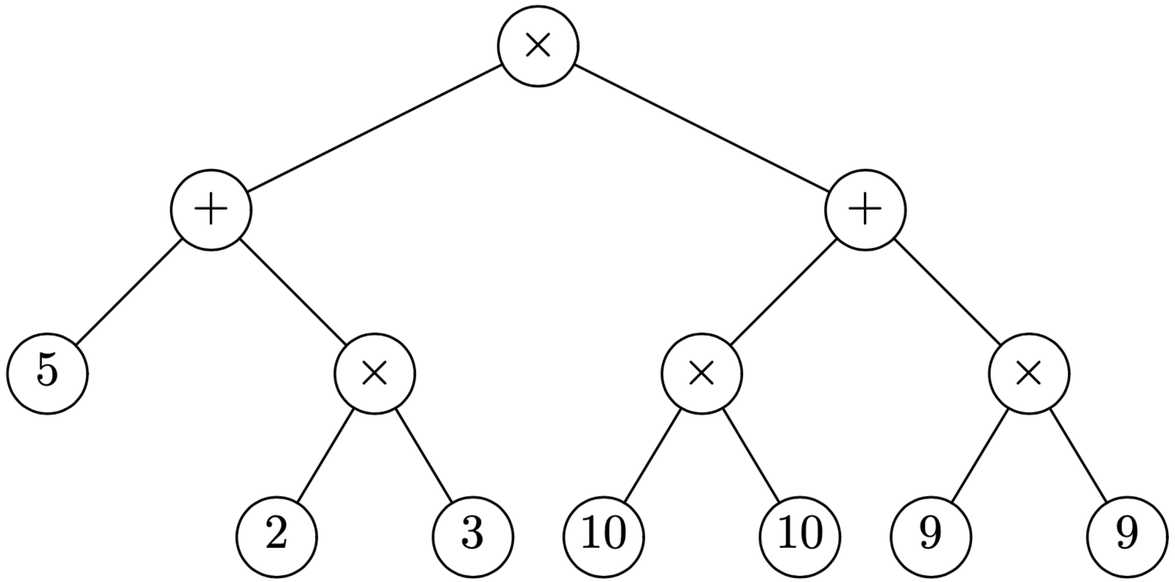
```
d = Noeud_Bi ('*',
              Noeud_Bi ('+', Feuille (5),
                             Noeud_Un ('-',
                                       Noeud_Bi ('*', Feuille (2), Feuille (3)))),
              Feuille (9))
```

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

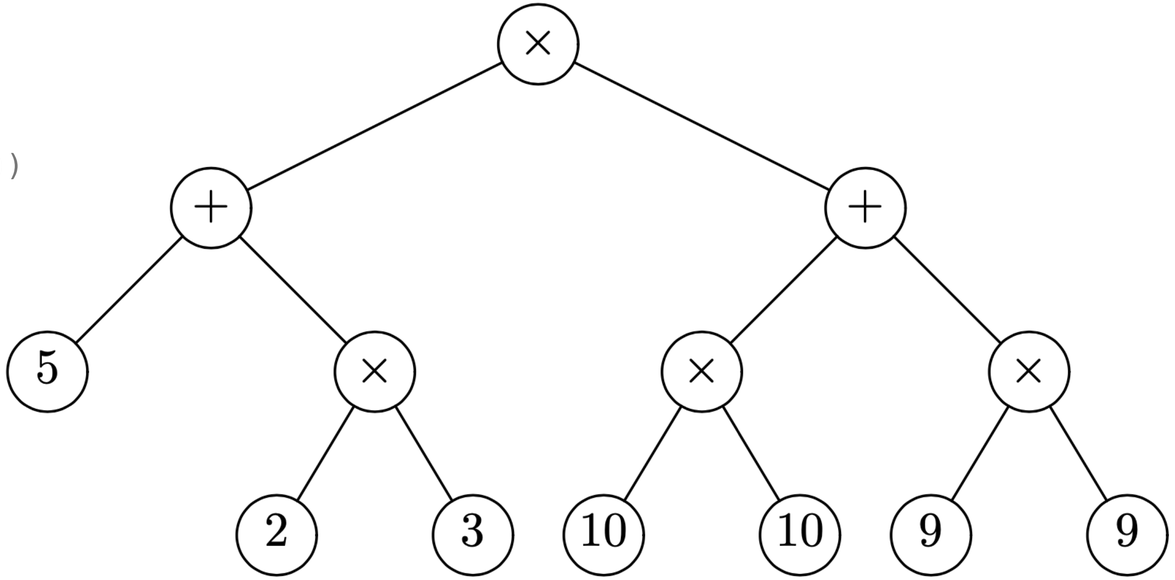
```
print (b)
Noeud (*, Noeud (+, Feuille (5), Noeud (*, Feuille (2), Feuille (3))), Noeud (+, Noeud (*, Feuille (10),
Feuille (10)), Noeud (*, Feuille (9), Feuille (9))))

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

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

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

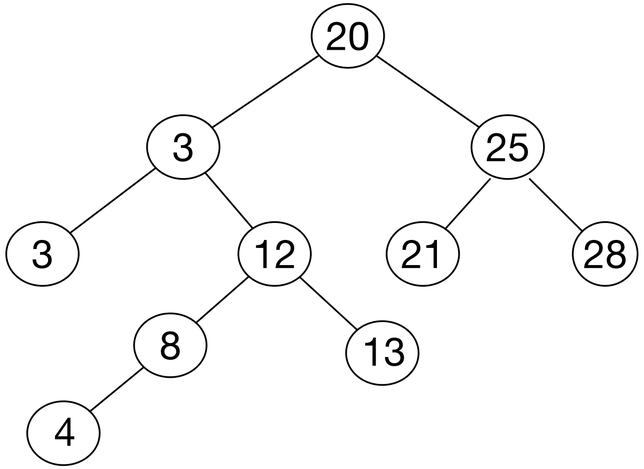
  [**on simplifie ici en ne considérant que les clés**]

- 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

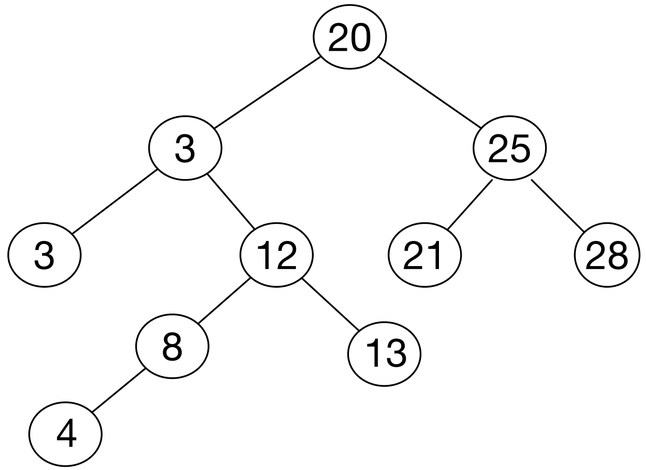
  [ici, on met les clés égales vers la gauche]




```
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

- recherche en table organisée en **arbre binaire** (*OO programming*)




```
class Noeud:
    # comme avant et en plus :
    def rechercher (self, x) :
       if self.val == x :
           return True
       elif x < a.val :
           return self.gauche.rechercher (x)
       else :
           return self.droit.rechercher(x)


class Feuille:
    # comme avant et en plus :
    def rechercher (self, x) :
        return self.val == x
```

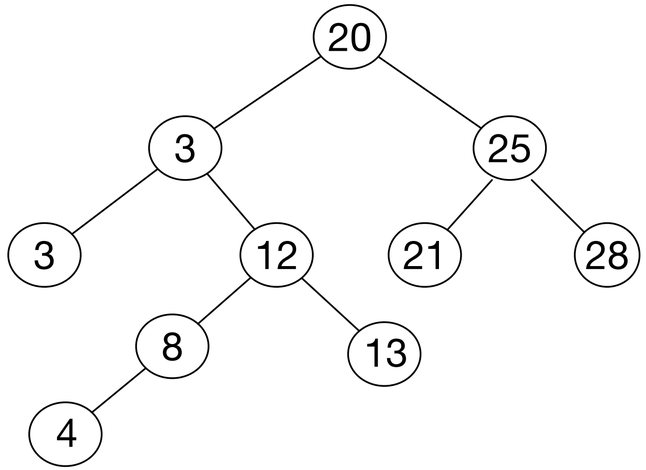
Arbres binaires de recherche

- 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]



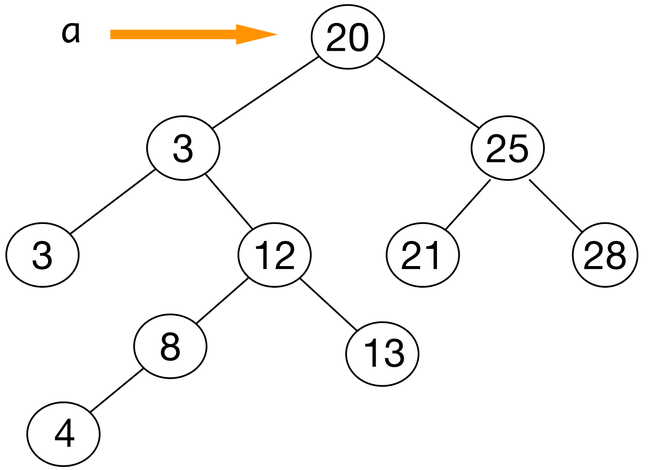

```
def rechercher (x, a) :
    if a == None :
        return False
    elif x == a.val :
        return True
    elif x < a.val :
        return rechercher (x, a.gauche)
    else :
        return rechercher (x, a.droit)
```

programme plus simple avec un seul type de noeud

Arbres binaires de recherche

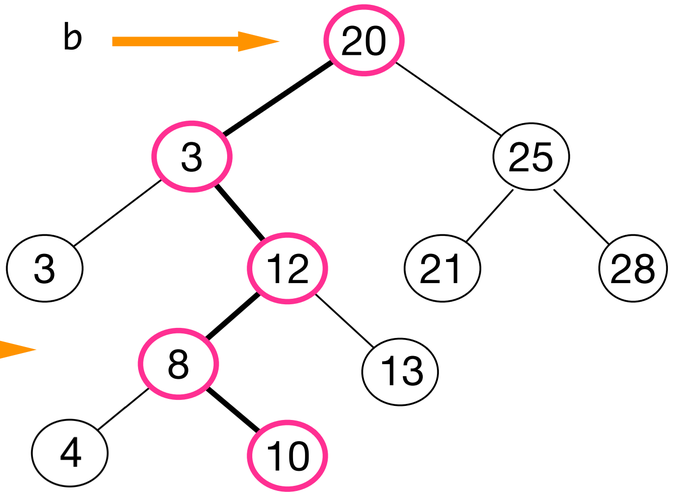
- ajouter une clé (style: **programmation fonctionnelle**)

```
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



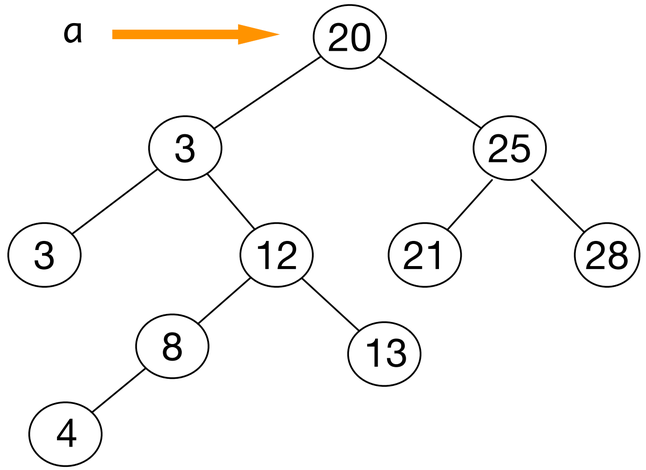
Arbres binaires de recherche

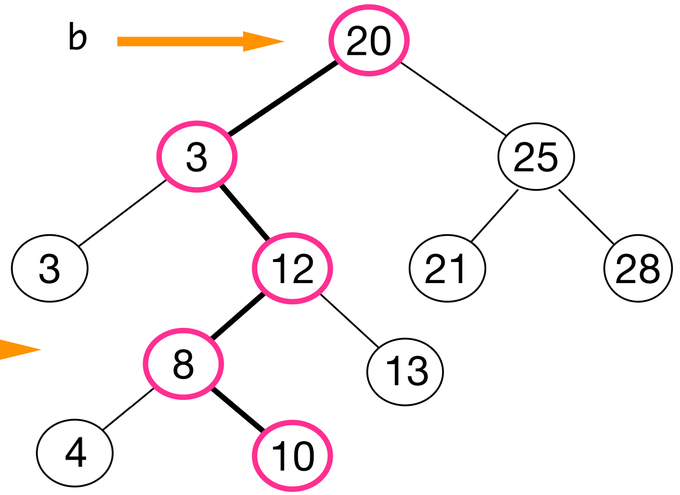
- ajouter une clé (style: **programmation fonctionnelle**)

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 impérative**)

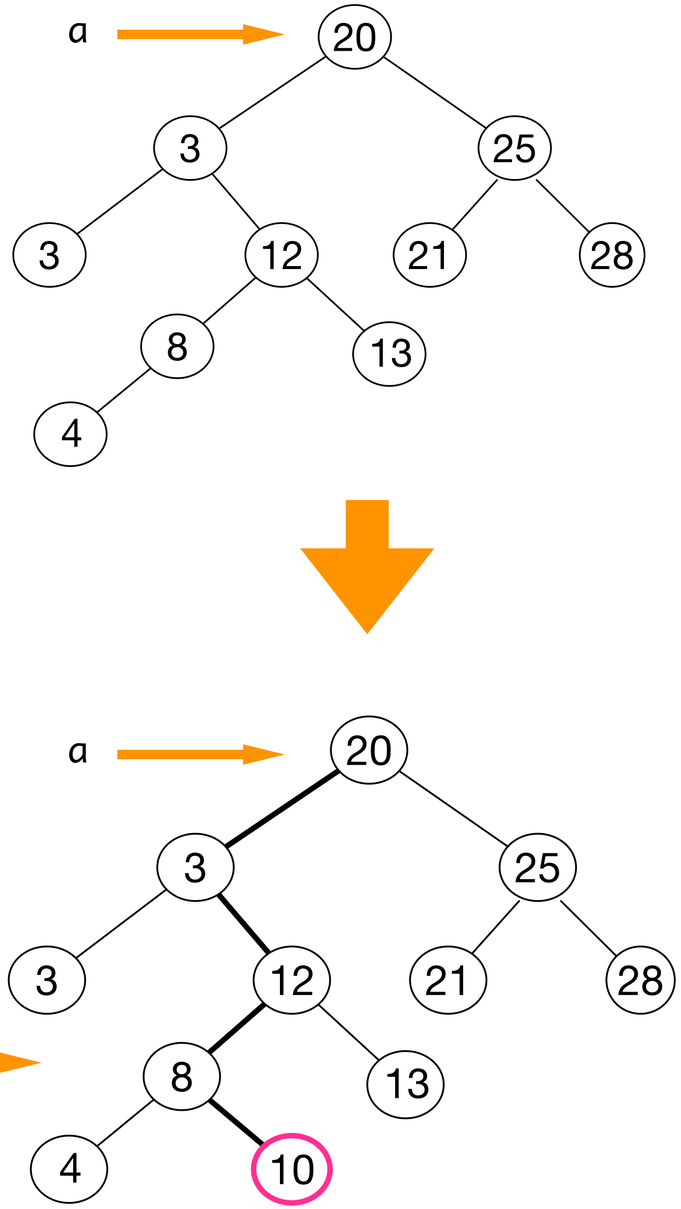
programme ne crée qu'un nouveau noeud

```
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

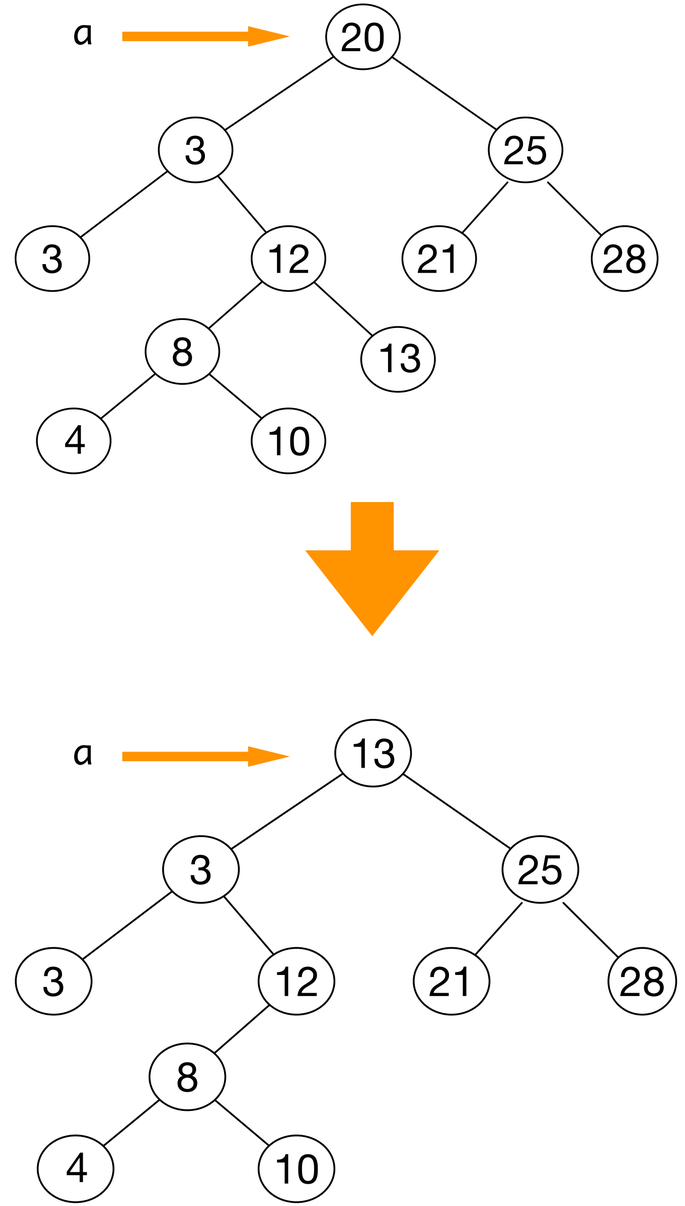
Arbres binaires de recherche

- 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)`

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

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)

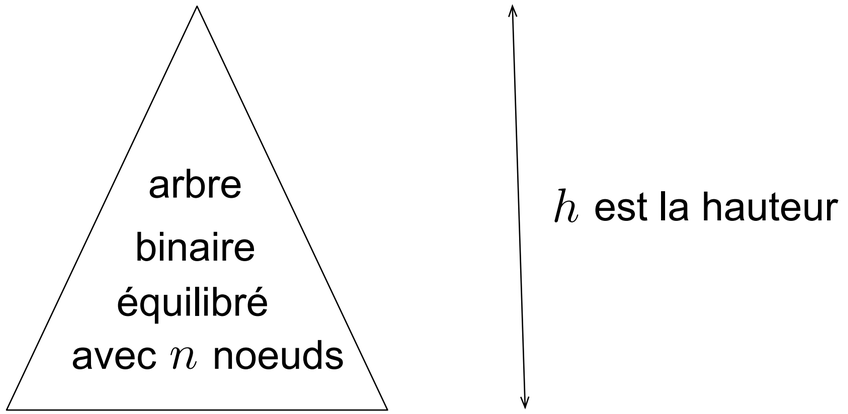
données non modifiables

- 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

Arbres binaires de recherche

- l'ajout d'une clé 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
- comment faire des arbres bien équilibrés ?



$$h \simeq \log n$$

$$n \simeq 2^h$$

Arbres de recherche équilibrés (AVL)

[Adelson-Velsky & Landis, 1962]

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

• où le champ h est la hauteur de l'arbre

```
a.hauteur == hauteur (a)
```

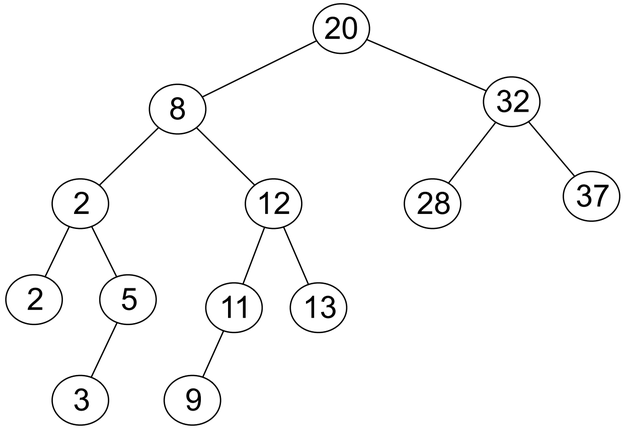
• on s'intéresse à la différence de hauteur entre fils gauche et droit

```
def getH (a) :
    return 0 if a == None else a.hauteur

def getBal (a) :
    return getH (a.droit) - getH (a.gauche)
```

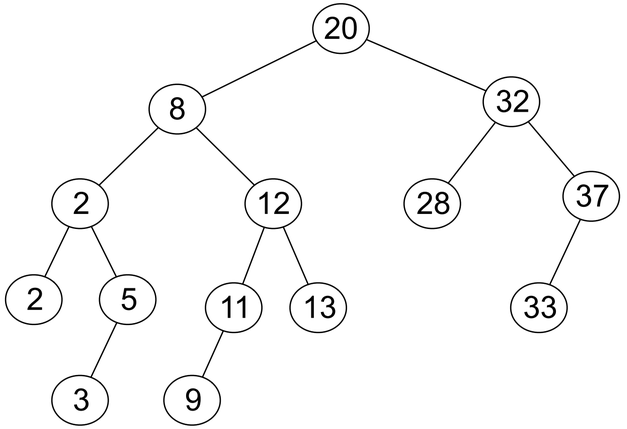
• les arbres AVL équilibrent les hauteurs des fils gauche et droit à une unité près

```
-1 <= getBal(a) <= 1
```

Arbres de recherche équilibrés (AVL)

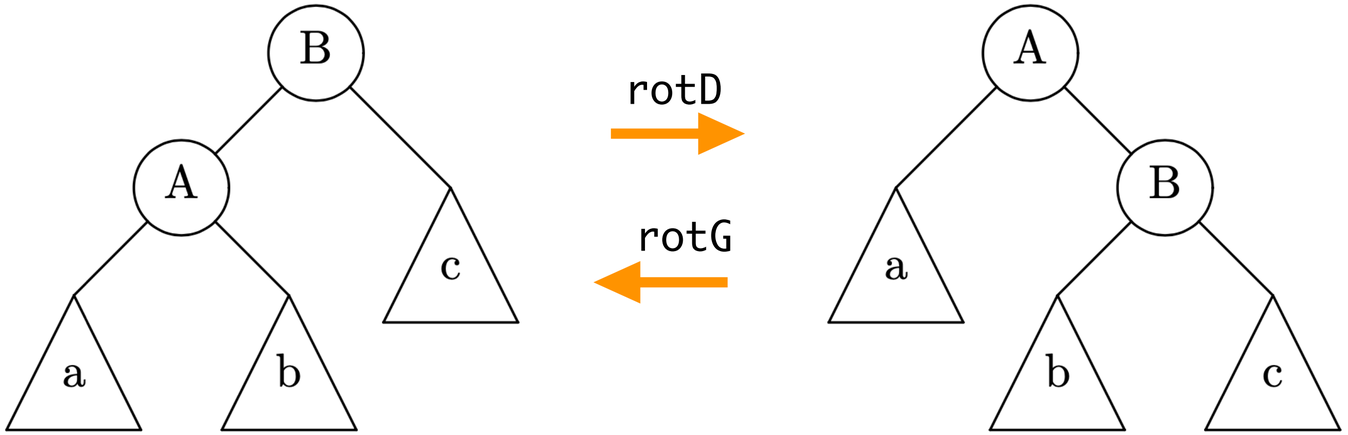


AVL ✗



AVL ✓

Arbres de recherche équilibrés (AVL)



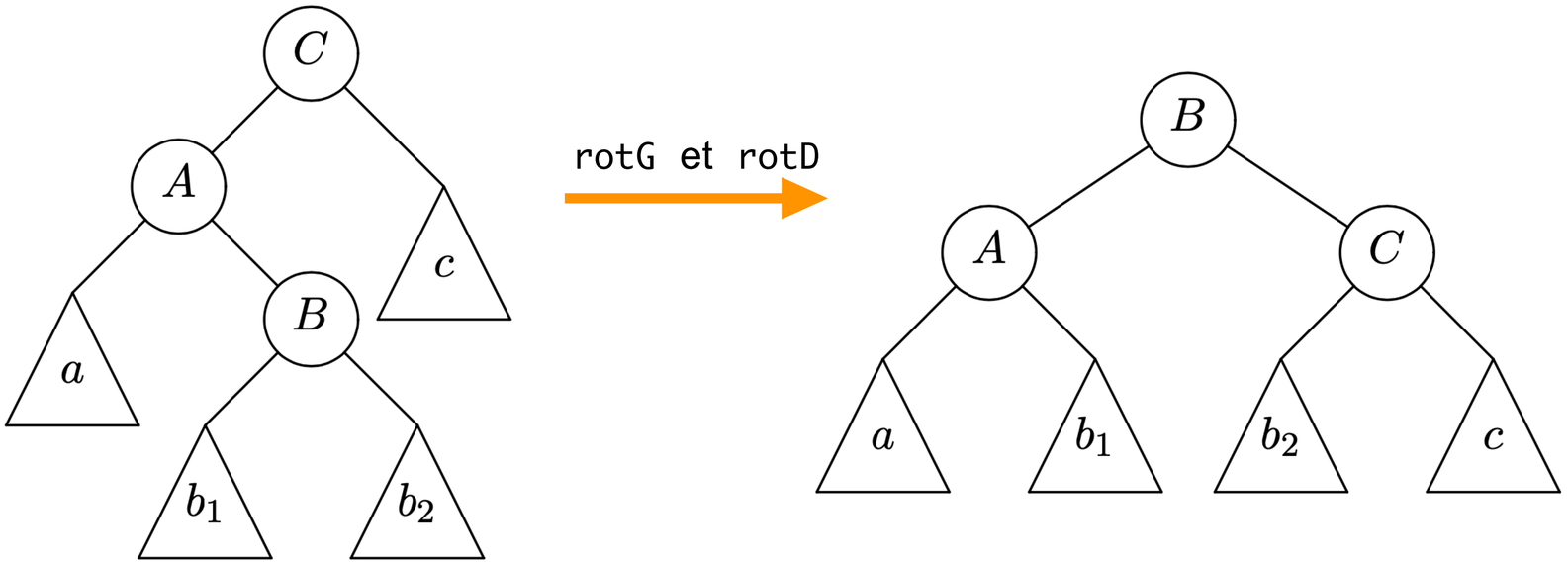

```
def rotD (b) :
    a = b.gauche
    b.gauche = a.droit
    a.droit = b
    return a
```

```
def rotG (a) :
    b = a.droit
    a.droit = b.gauche
    b.gauche = a
    return b
```

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

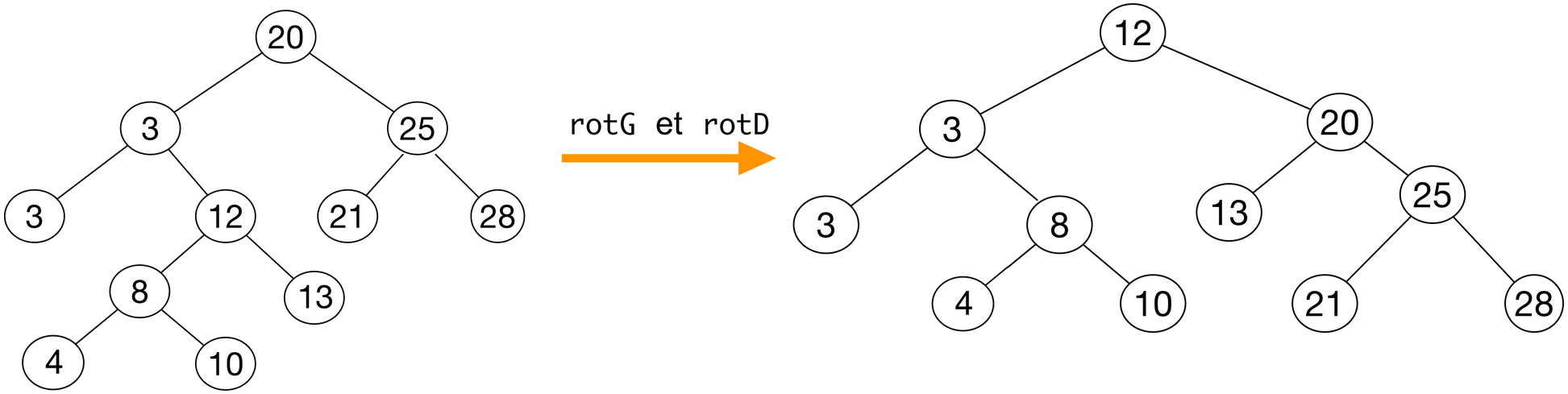
Arbres de recherche équilibrés (AVL)

- double rotation (gauche puis droite)



Arbres de recherche équilibrés (AVL)

- exemple de double rotation

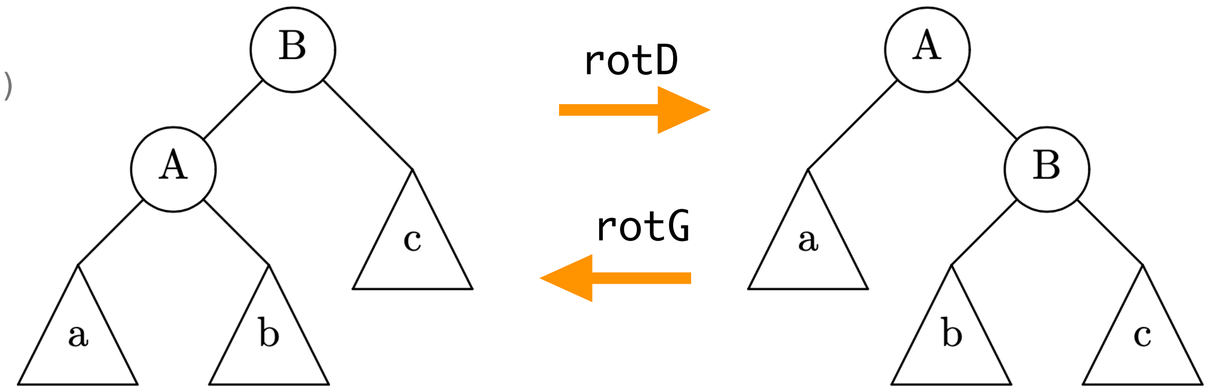


Arbres de recherche équilibrés (AVL)

- rotations dans un arbre AVL

```
def rotD (b) :
    a = b.gauche
    b.gauche = a.droit
    a.droit = b
    b.hauteur =  1 + max (getH (b.gauche), getH (b.droit))
    a.hauteur = 1 + max (getH (a.gauche), b.hauteur)
    return a
```

```
def rotG (a)
    b = a.droit
    a.droit = b.gauche
    b.gauche = a
    a.hauteur =  1 + max (getH (a.gauche), getH(a.droit))
    b.hauteur = 1 + max (a.hauteur, getH (b.droit))
    return a
```



Arbres de recherche équilibrés (AVL)

- ajouter une clé à un arbre AVL

```
def ajouter (x, a) :
    if a == None :
       return Noeud (x, 1, None, None)
    elif x <= a.val :
       a.gauche = ajouter (x, a.gauche)
       a.hauteur = 1 + max (a.gauche.hauteur, getH (a.droit))
    else :
       a.droit = ajouter (x, a.droit)
       a.hauteur = 1 + max (getH (a.gauche), a.droit.hauteur)
    bal = getBal (a)
    if bal < -1 :
       if getBal (a.gauche) >= 0 :
            a.gauche = rotG (a.gauche)
       a = rotD (a)
    elif bal > 1 :
       if getBal (a.droit) <= 0 :
            a.droit = rotD (a.droit)
       a = rotG (a)
    return a
```

Exercice écrire la fonction `supprimer (x, a)` pour enlever une clé d'un arbre AVL

- ces fonctions sont bien compliquées

Arbres de recherche équilibrés

- on peut rendre **plus flexible** la loi d'équilibre des arbres AVL

- arbres 2-3

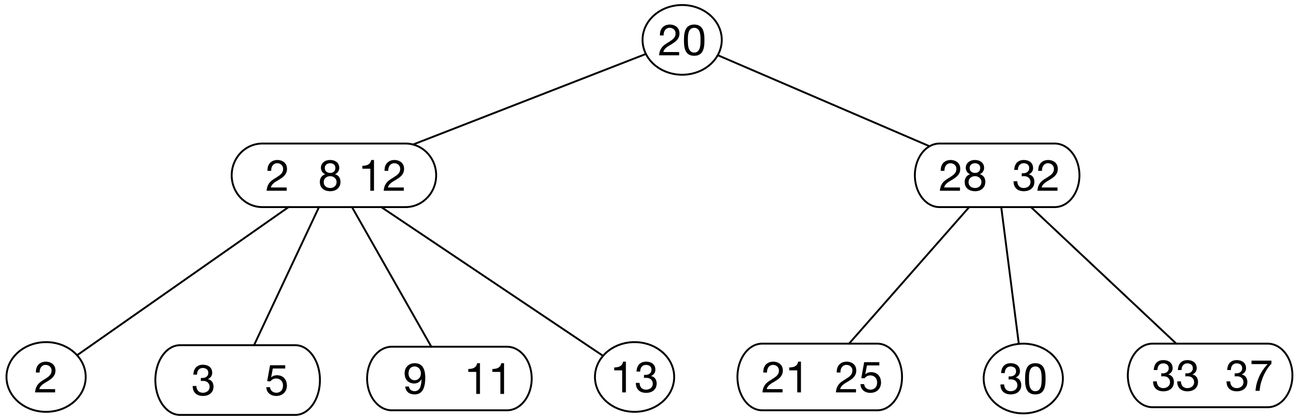
- arbres 2-3-4 ou plus généralement arbres-B *(B-trees*)
  [le système de fichiers de MacOs utilise des B-trees pour accéder à leur adresses disque]

- arbres bicolores rouge-noir

Arbres de recherche équilibrés (2-3-4)

[Bayer & McCreight, 1970]

- les noeuds peuvent contenir 1, 2 ou 3 clés et donc 2, 3 ou 4 fils

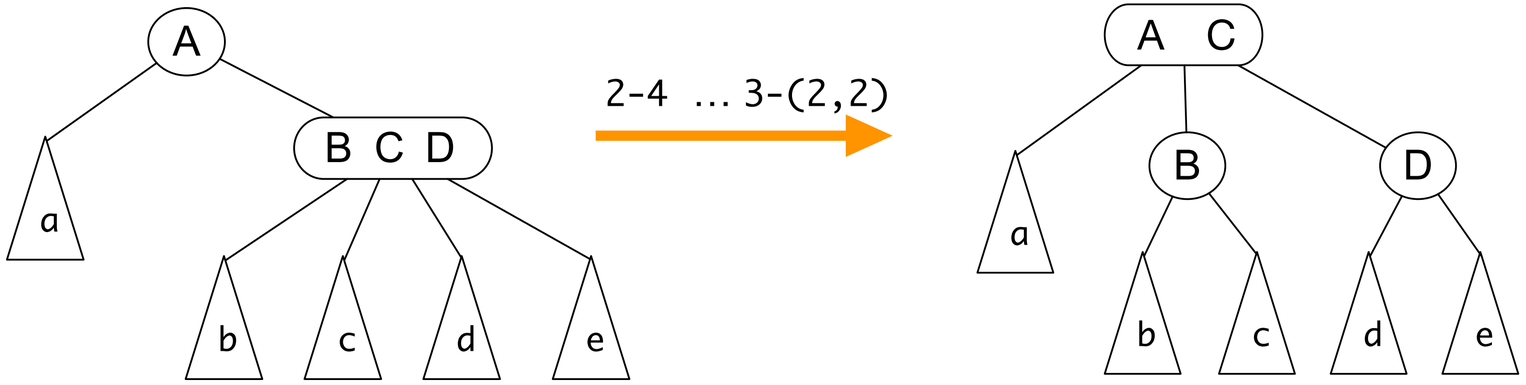


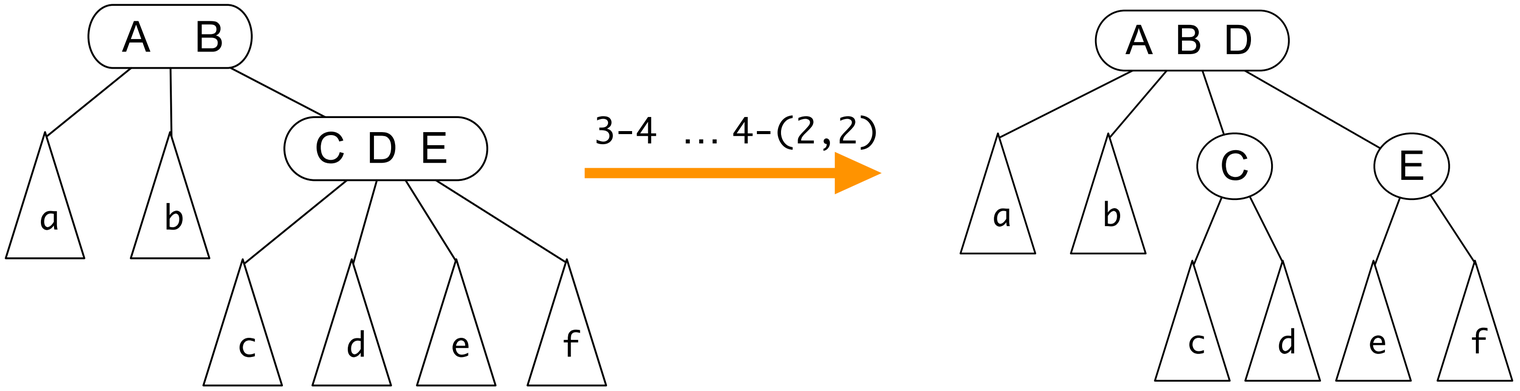
- on peut insérer une nouvelle clé dans tout noeud non quaternaire

- si impossible, on éclate le noeud quaternaire

Arbres de recherche équilibrés (2-3-4)

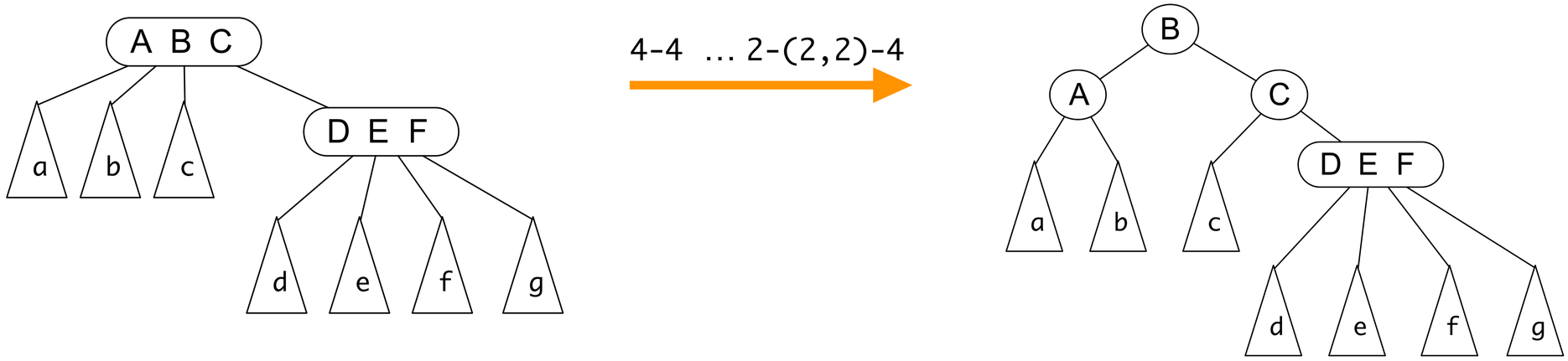
- on éclate les noeuds 4 sans augmenter la hauteur





Arbres de recherche équilibrés (2-3-4)

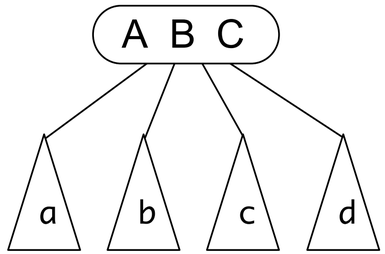
- on éclate un noeud 4 racine en augmentant la hauteur de 1 sans déséquilibrer les sous-arbres

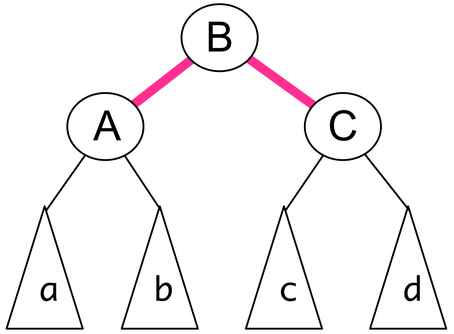


Arbres bicolores (rouge-noir)

[Guibas & Sedgewick, 1978]

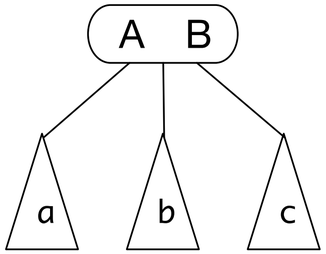
• on peut implémenter les arbres 2-3-4 par des arbres binaires bicolores

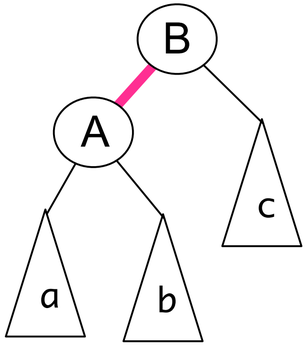




arbres 2-3-4

arbres bicolores

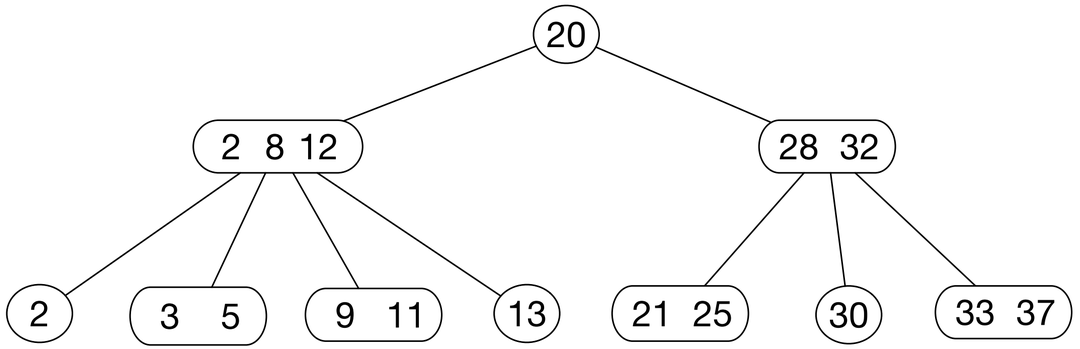


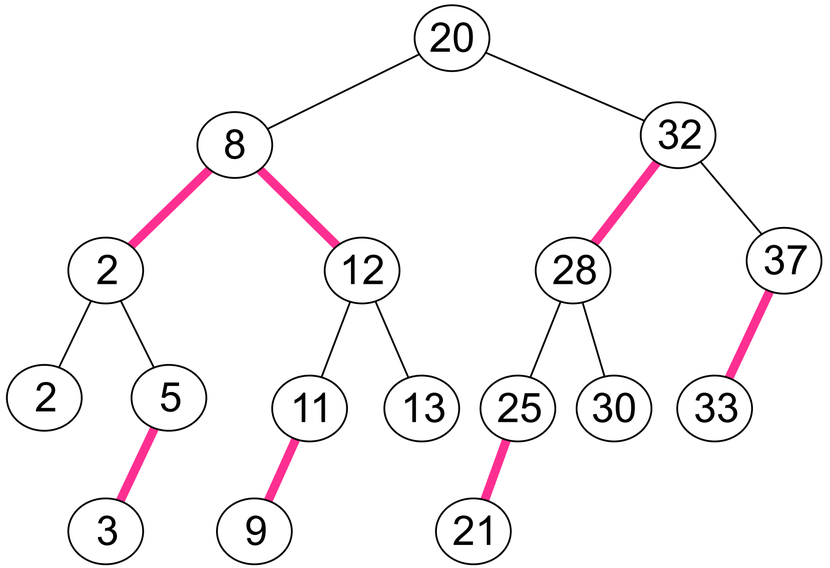


[on choisit
arbitrairement
vers la gauche]

Arbres bicolores (rouge-noir)

• on peut implémenter les arbres 2-3-4 par des arbres binaires bicolores





arbres 2-3-4 ⟷ arbres bicolores

• dans un arbre rouge-noir, le nombre de branches noires sur tout chemin d'un noeud à ses feuilles est le même

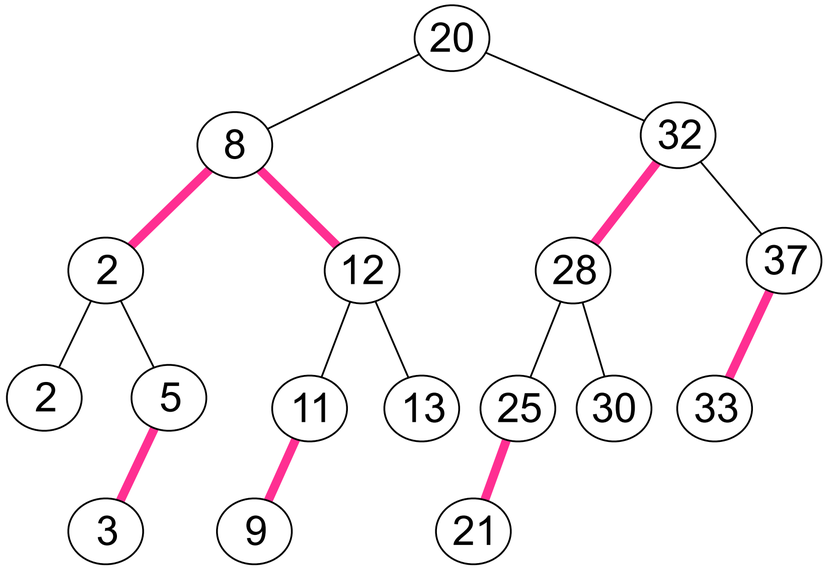
[tout noeud est équilibré par rapport à sa hauteur noire]

Arbres bicolores (rouge-noir)

• la couleur d'un noeud est la couleur de la branche qui la relie à son père

• le champ `couleur` est un simple booléen

```
ROUGE = True
NOIR = False

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



• dans un arbre rouge-noir, le nombre de branches noires sur tout chemin d'un noeud à ses feuilles est le même

[tout noeud est équilibré par rapport à sa hauteur noire]

Arbres bicolores (rouge-noir)

```
def est_rouge (a) :
    return a != None and a.couleur == ROUGE
```

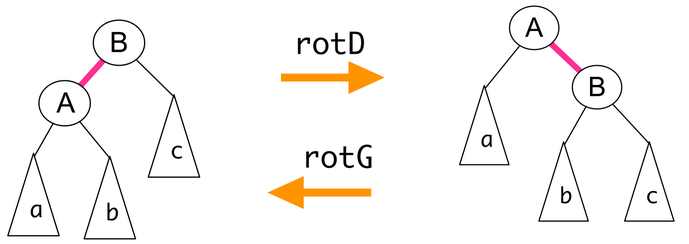
- le programme complet pour les arbres rouge-noir

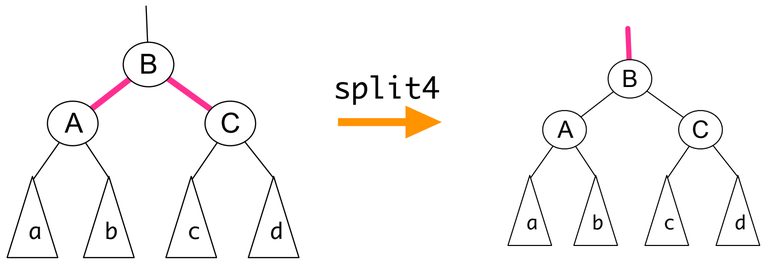
```
def rotD (b) :
    a = b.gauche
    b.gauche = a.droit
    a.droit = b
    a.couleur = b.couleur
    a.droit.couleur = ROUGE
    return a
```

```
def rotG (a) :
    b = a.droit
    a.droit = b.gauche
    b.gauche = a
    b.couleur = a.couleur
    b.gauche.couleur = ROUGE
    return b
```

```
def split4 (b) :
    b.couleur = not b.couleur
    b.gauche.couleur = not b.gauche.couleur
    b.droit.couleur = not b.droit.couleur
    return b
```

```
def ajouter (x, a) :
    if a == None :
      return Noeud (x, ROUGE, None, None);
    if est_rouge (a.gauche) and est_rouge (a.droit) :
      split4 (a)
    if x <= a.val :
      a.gauche = ajouter (x, a.gauche)
    else :
      a.droit = ajouter (x, a.droit)
    if est_rouge (a.droit) :
      a = rotG (a)
    if est_rouge (a.gauche) and est_rouge (a.gauche.gauche) :
      a = rotD (a)
    return a
```





Arbres bicolores (rouge-noir)

- on rajoute une méthode pour l'impression

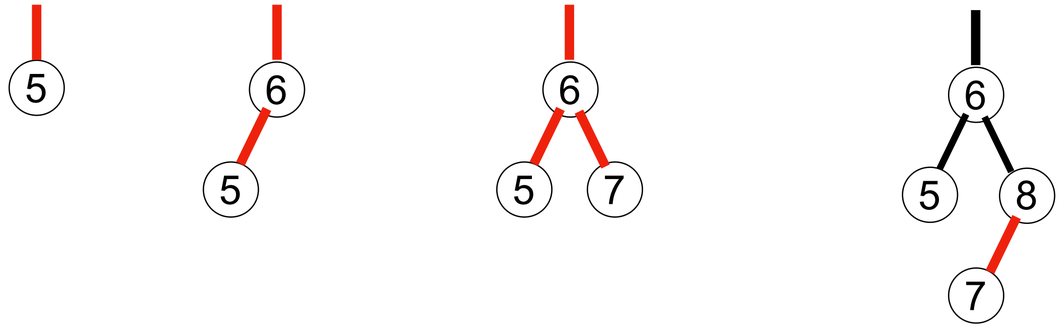
```
class Noeud:
    def __init__ (self, x, c, g, d) :
        self.val = x
        self.couleur = c
        self.gauche = g
        self.droit = d

    def __str__ (self) :
        return "{} ({}, {}, {})".format ("ROUGE" if self.couleur == ROUGE else "NOIR",
                                         self.val, self.gauche, self.droit)
```

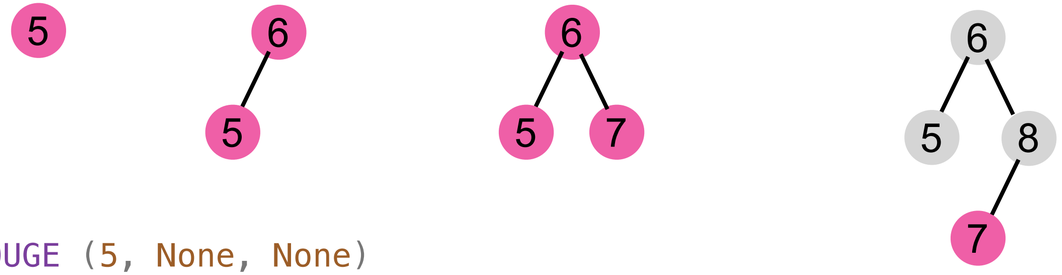
- exemple d'exécution

```
a = None
for i in [5, 6, 7, 8] :
  a = ajouter (i, a)
  print (a)
```

```
ROUGE (5, None, None)
ROUGE (6, ROUGE (5, None, None), None)
ROUGE (6, ROUGE (5, None, None), ROUGE (7, None, None))
NOIR (6, NOIR (5, None, None), NOIR (8, ROUGE (7, None, None), None))
```

Arbres bicolores (rouge-noir)



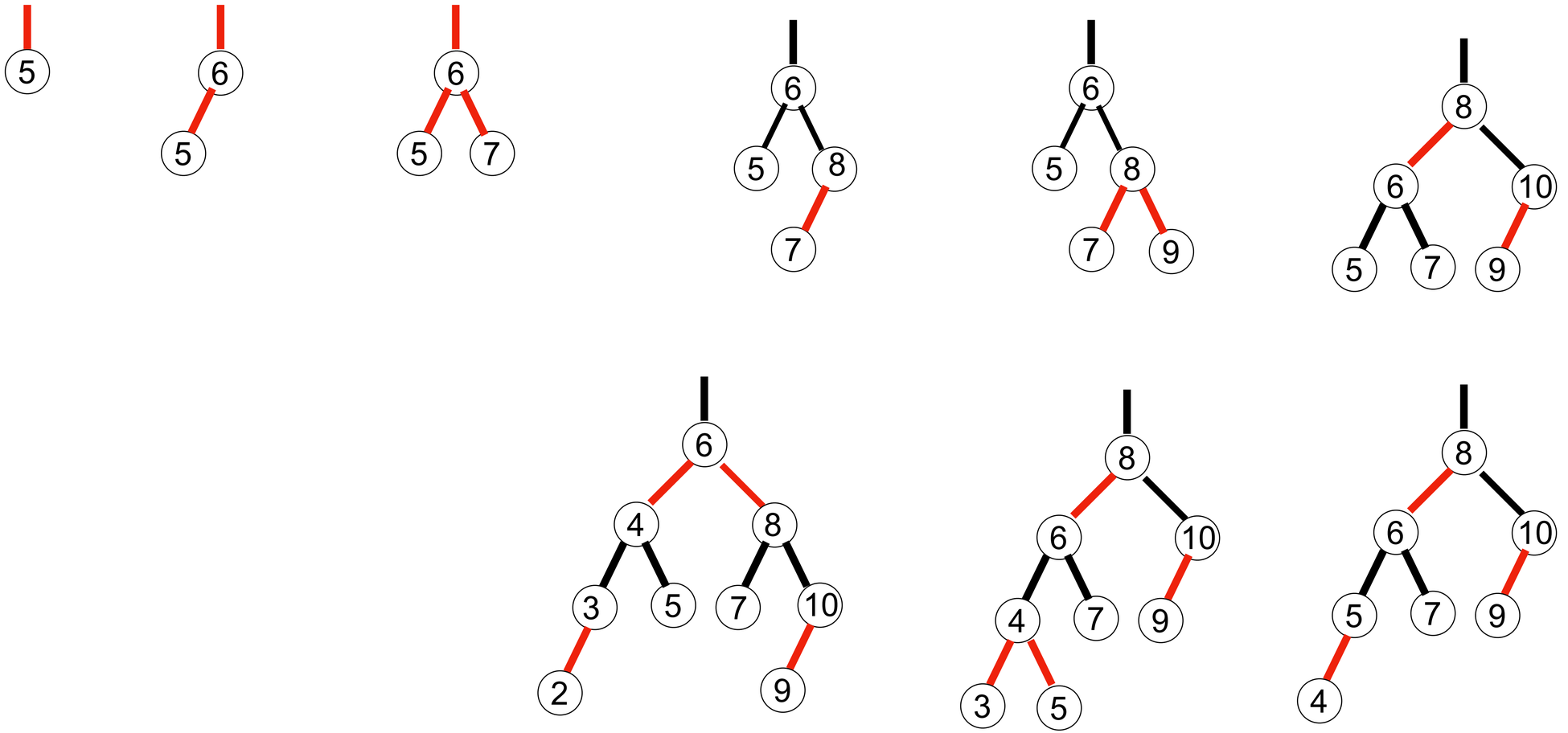


- exemple d'exécution

```
a = None
for i in [5, 6, 7, 8] :
  a = ajouter (i, a)
  print (a)
```

```
ROUGE (5, None, None)
ROUGE (6, ROUGE (5, None, None), None)
ROUGE (6, ROUGE (5, None, None), ROUGE (7, None, None))
NOIR (6, NOIR (5, None, None), NOIR (8, ROUGE (7, None, None), None))
```

Arbres bicolores (rouge-noir)



Arbres bicolores (rouge-noir)

- la programmation précédente est impérative (avec effets de bord sur les arbres)

- on peut aussi la faire en programmation fonctionnelle (sans effets de bord sur les arbres)
  [les arbres sont alors persistents]

- en Haskell (langage fonctionnel), c.f. `https://abhiroop.github.io/Haskell-Red-Black-Tree/`

- en Ocaml (langage semi-fonctionnel), c.f. le cours de `Xavier Leroy` (collège de France)

Exercice Programmer les arbres rouge-noir en style fonctionnel

Exercice (plus difficile) Ecrire la fonction `supprimer` dans les arbres rouge-noir

Exercice Dessiner les arbres binaires (rouge-noir) avec le module `graphics.py`

Au-delà des arbres de recherche

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

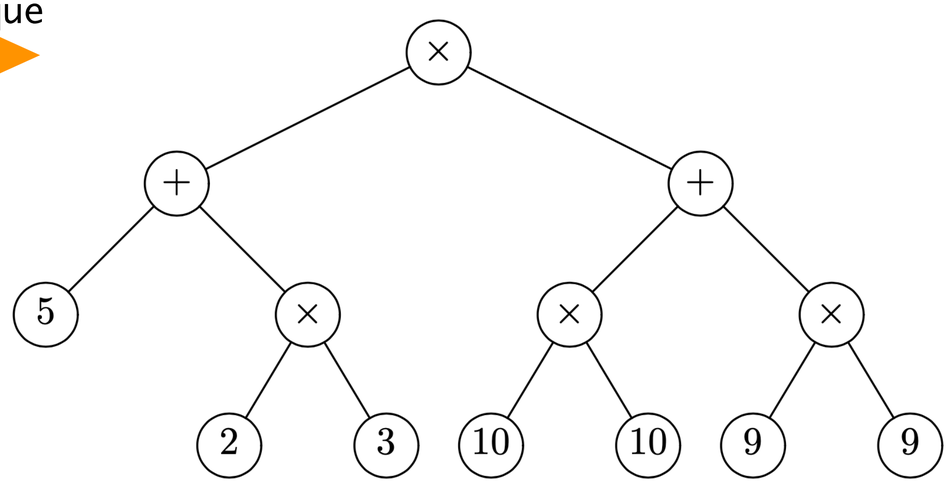
chaîne de caractères

(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

Prochainement

- programmes de base sur les graphes
- exploration et *backtracking*