Informatique et Programmation

Cours 7

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

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

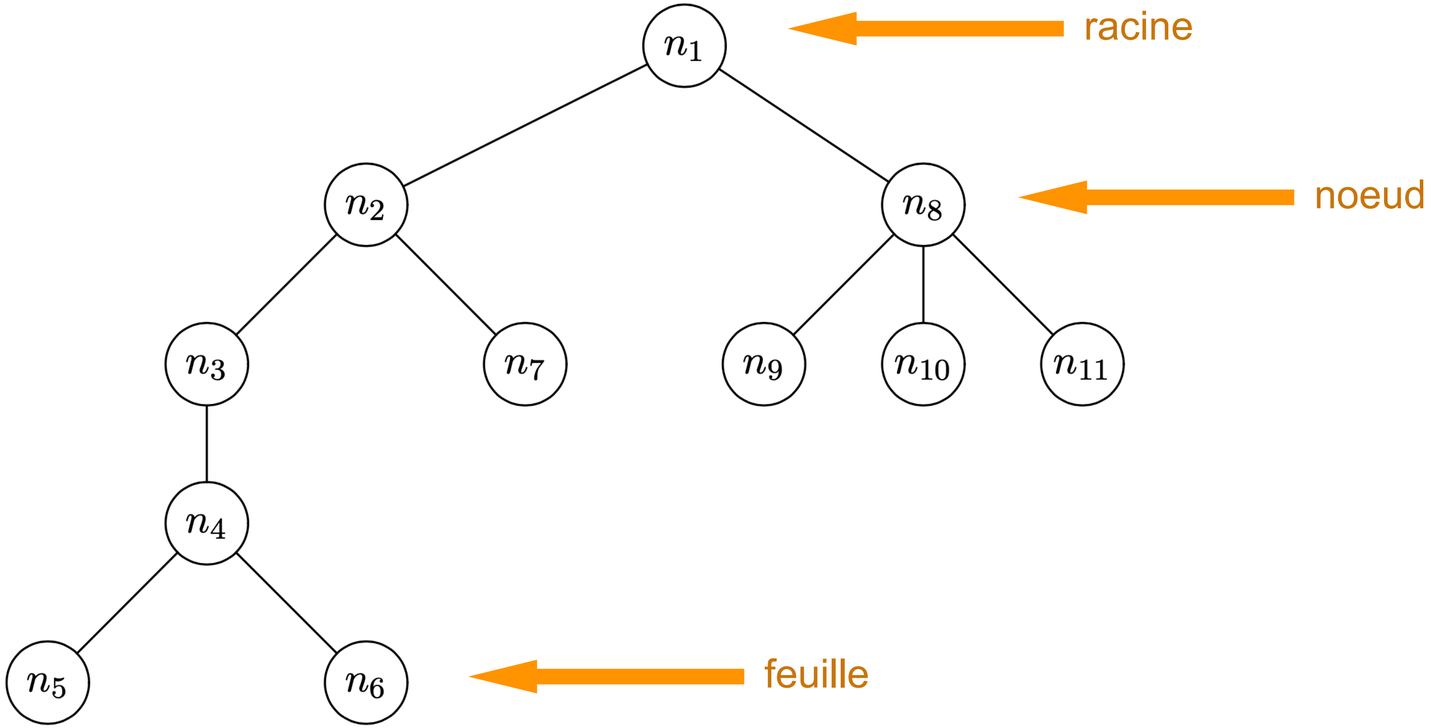
Plan

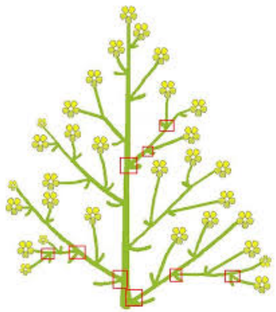
- arbres
- files de priorité
- classes
- objets

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

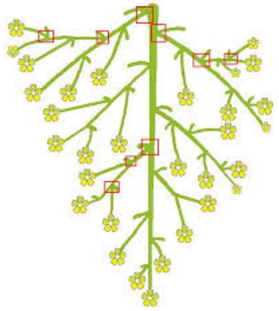
Les arbres en informatique

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





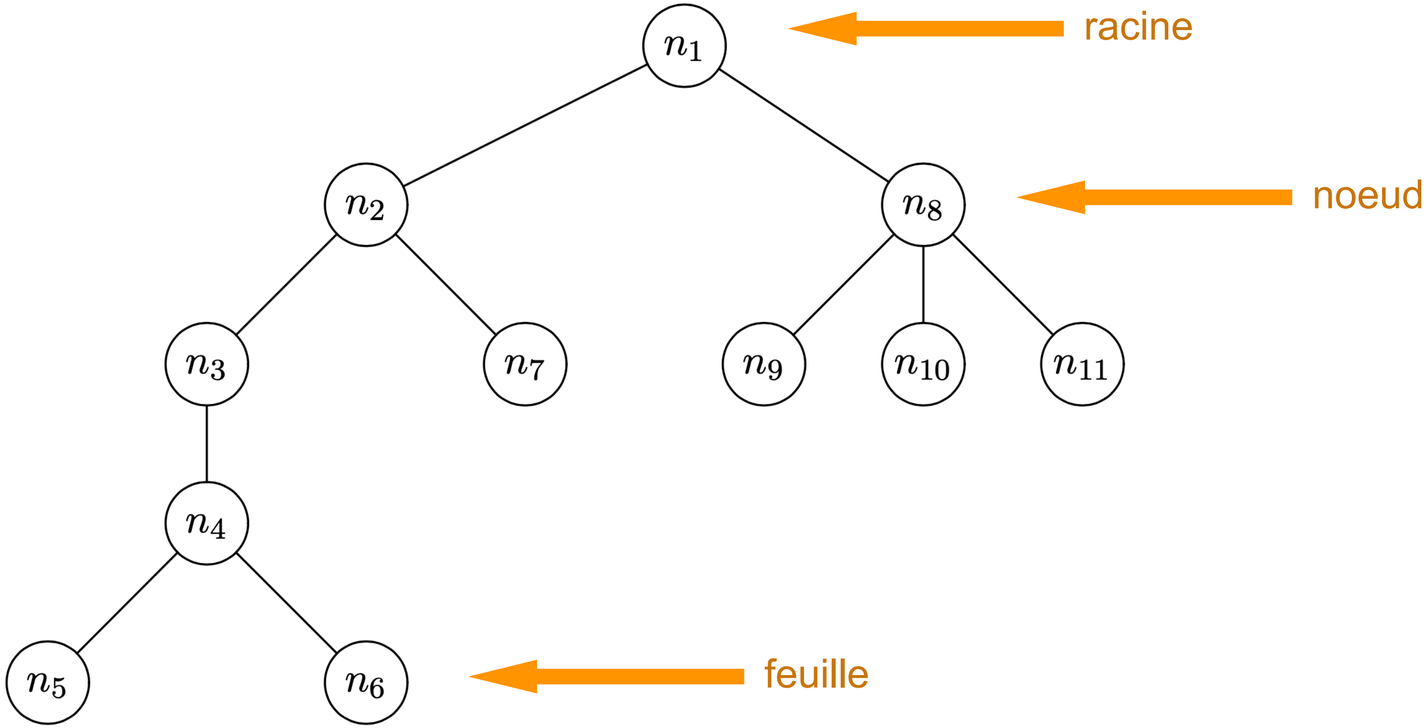
botanique

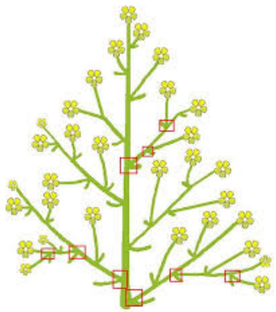


informatique

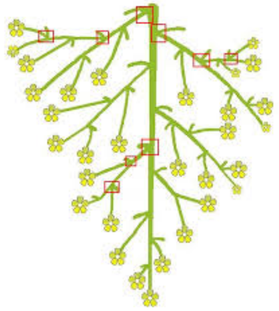
Les arbres en informatique

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





botanique



informatique

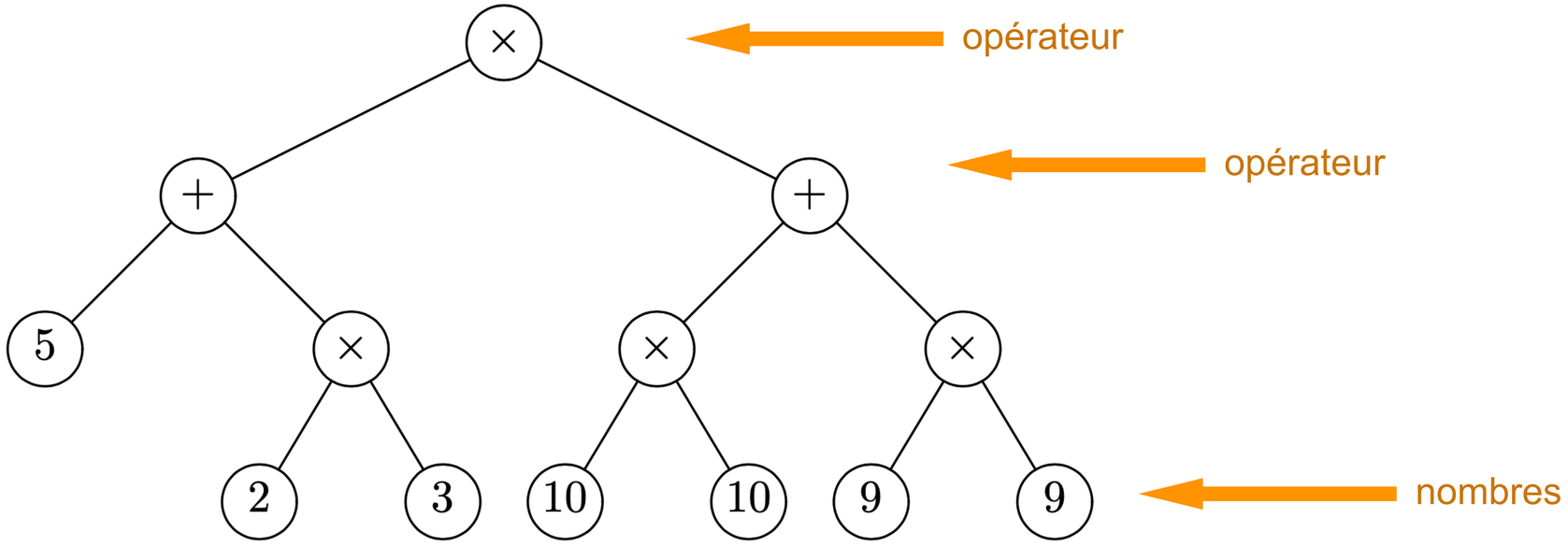
n2 est un **ancêtre** de n4

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]



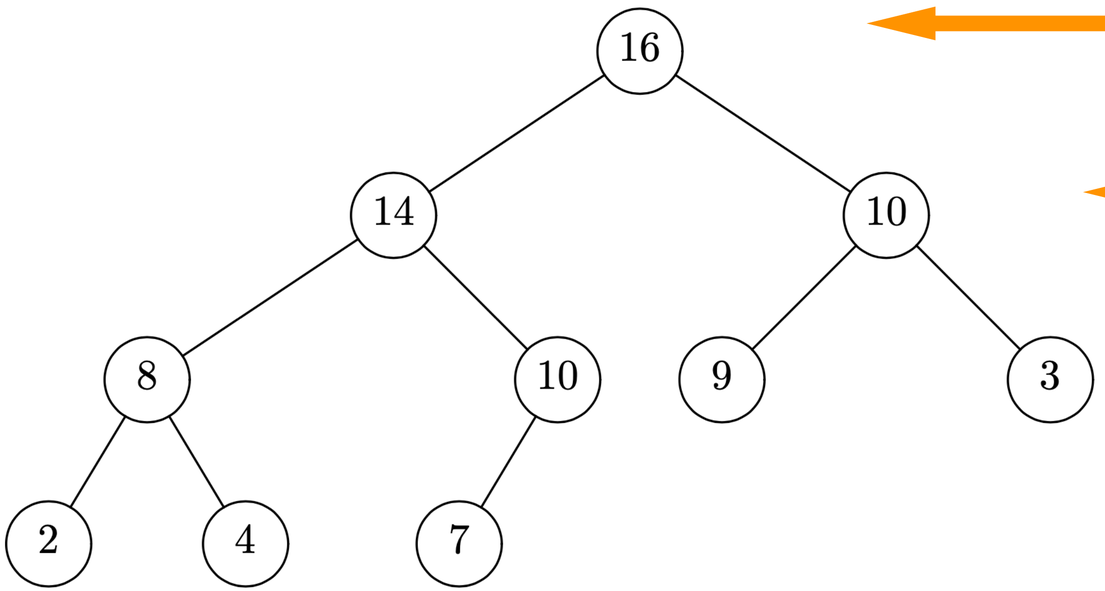
pour représenter une **expression arithmétique**

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

[plus besoin de parenthèses]

Les arbres en informatique

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



maximum

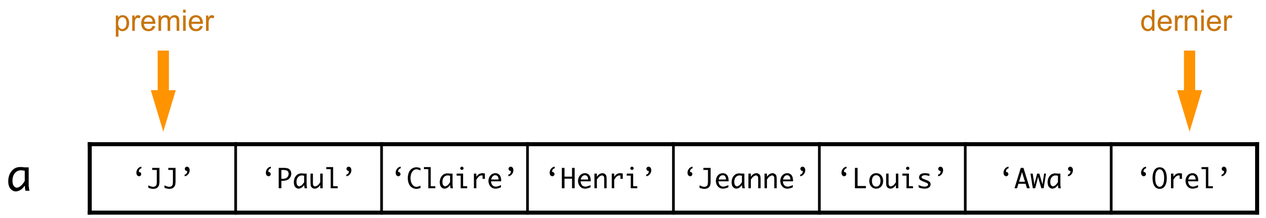
arbre binaire

tout noeud a au plus 2 fils

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

Files d’attente

- une simple file d’attente (*FIFO — First In First Out*)




```
def ajouter_file (x, a) :
    a.append(x)
```

```
def enlever_file () :
    try:
      del a[0]
    except Exception:
      print ('erreur')
```

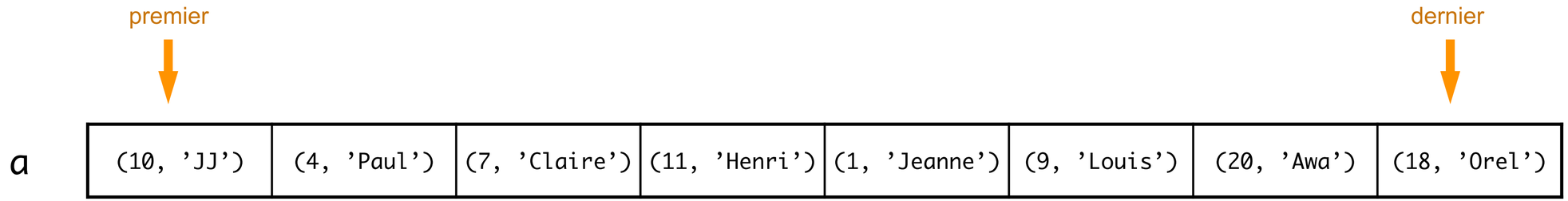
```
def nouvelle_file () :
    return [ ]

a = nouvelle_file ()
```

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

Files d’attente

- une file d’attente avec priorité [le plus prioritaire passe en premier]




```
def ajouter_file (x, a) :
    a.append(x)
```

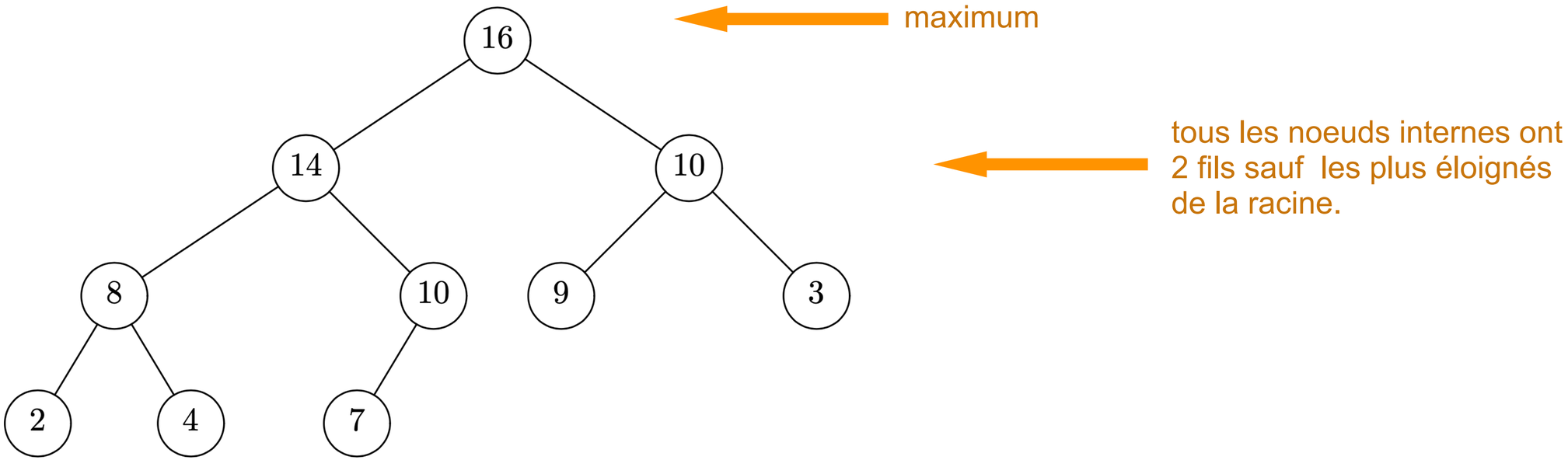
```
def enlever_file () :
    try:
      i = index_max_of(a)
      del a[i]
    except Exception:
      print ('erreur')
```

```
def nouvelle_file () :
    return [ ]

a = nouvelle_file ()
```

Files de priorité

• implémentation efficace [pour simplifier, on ne considère que les priorités]

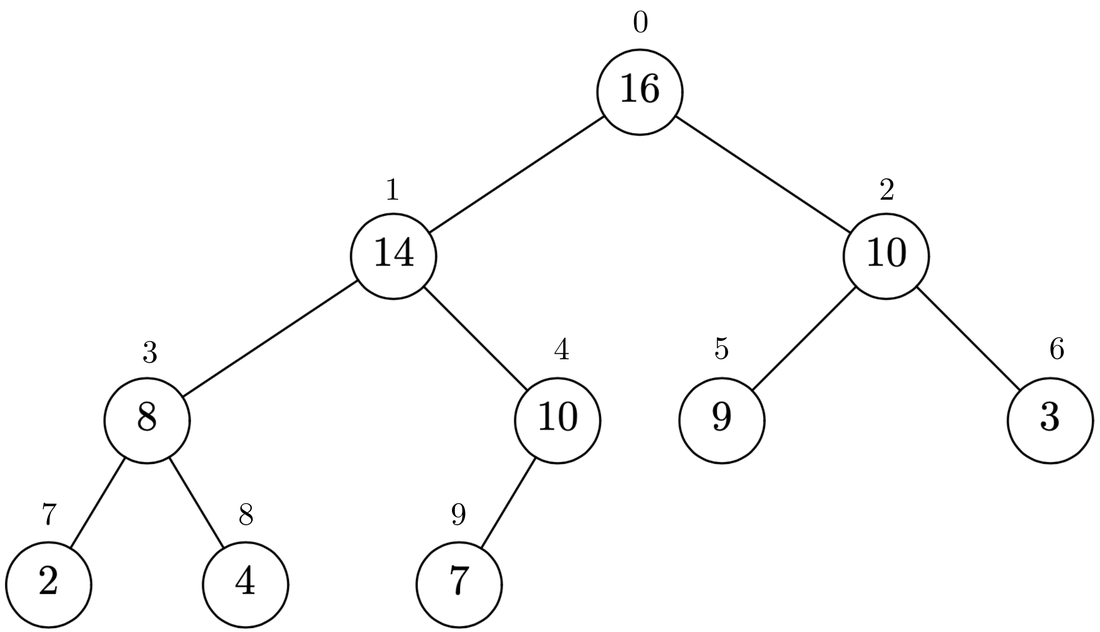


pour représenter une **file de priorité**, on utilise un arbre binaire presque parfait

[un ancêtre a une valeur plus élevée qu'un descendant]

Files de priorité

- on veut gérer une file d'attente où chacun a une priorité [le plus prioritaire passe en premier]



fils_gauche$(i) = 2i + 1$

fils_droit$(i) = 2i + 2$

père$(i) = \lfloor (i-1)/2 \rfloor$

- on utilise un **tas** (*heap*) c'est-à-dire un tableau indicé par les numéros figurant au dessus de chaque noeud

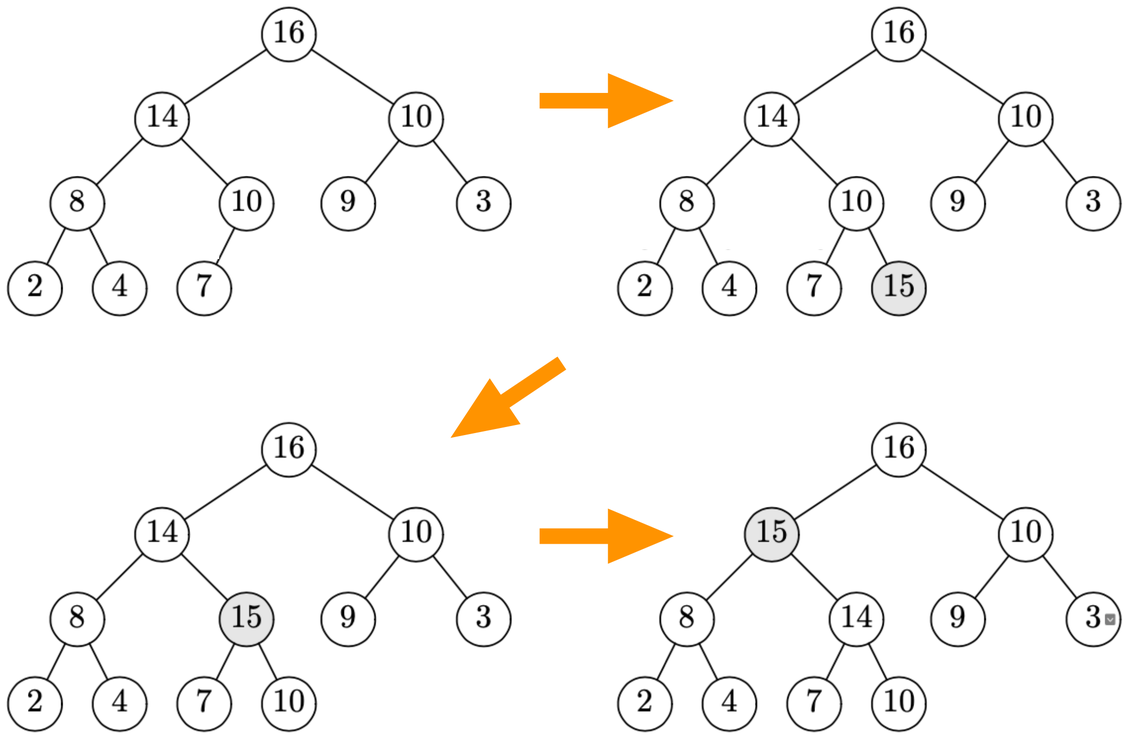
i	0	1	2	3	4	5	6	7	8	9
$a[i]$	16	14	10	8	10	9	3	2	4	7

Files de priorité

- créer une file vide

```
def nouvelle_file () :
    return []

a = nouvelle_file ()
```



- ajouter un élément à la file

```
def ajouter_file (x, a) :
    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
```

- si `n` est la longueur de la file, l'ajout d'un élément ne fait pas plus que `log(n)` opérations

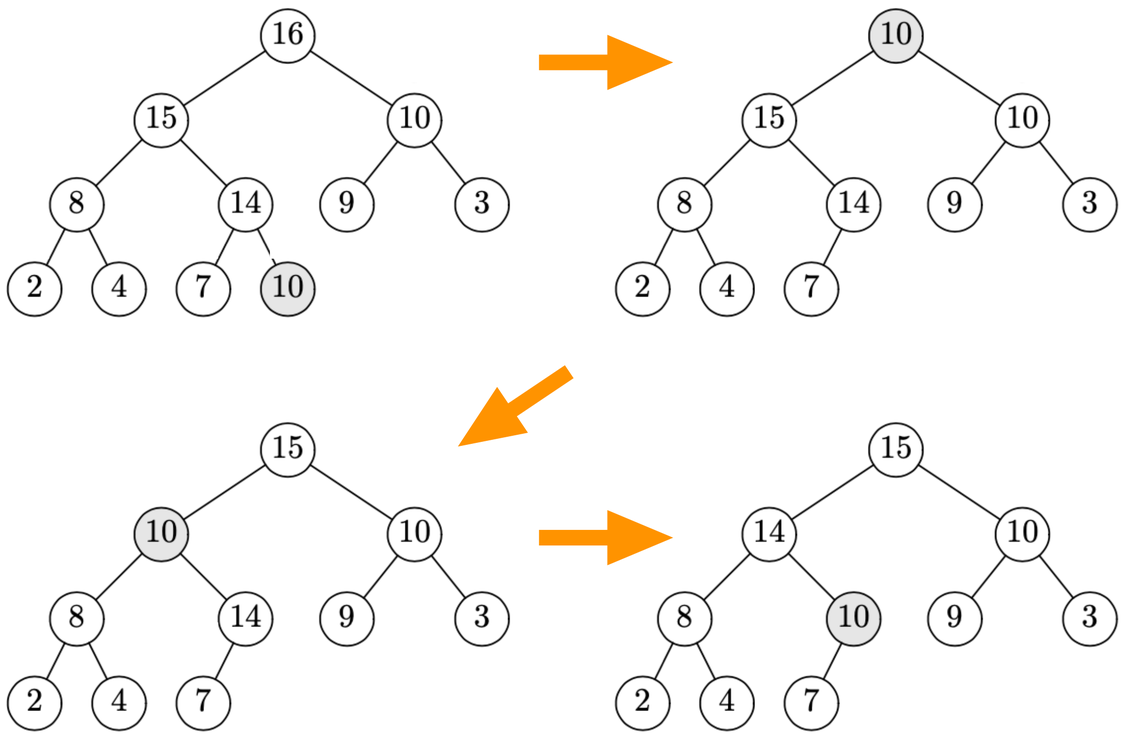
```
def max_de_file (a) :
    return a[0]
```

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

Files de priorité

Exercice expliquer la fonction `enlever_file (a)` qui retire l'élément maximum de la file *a*

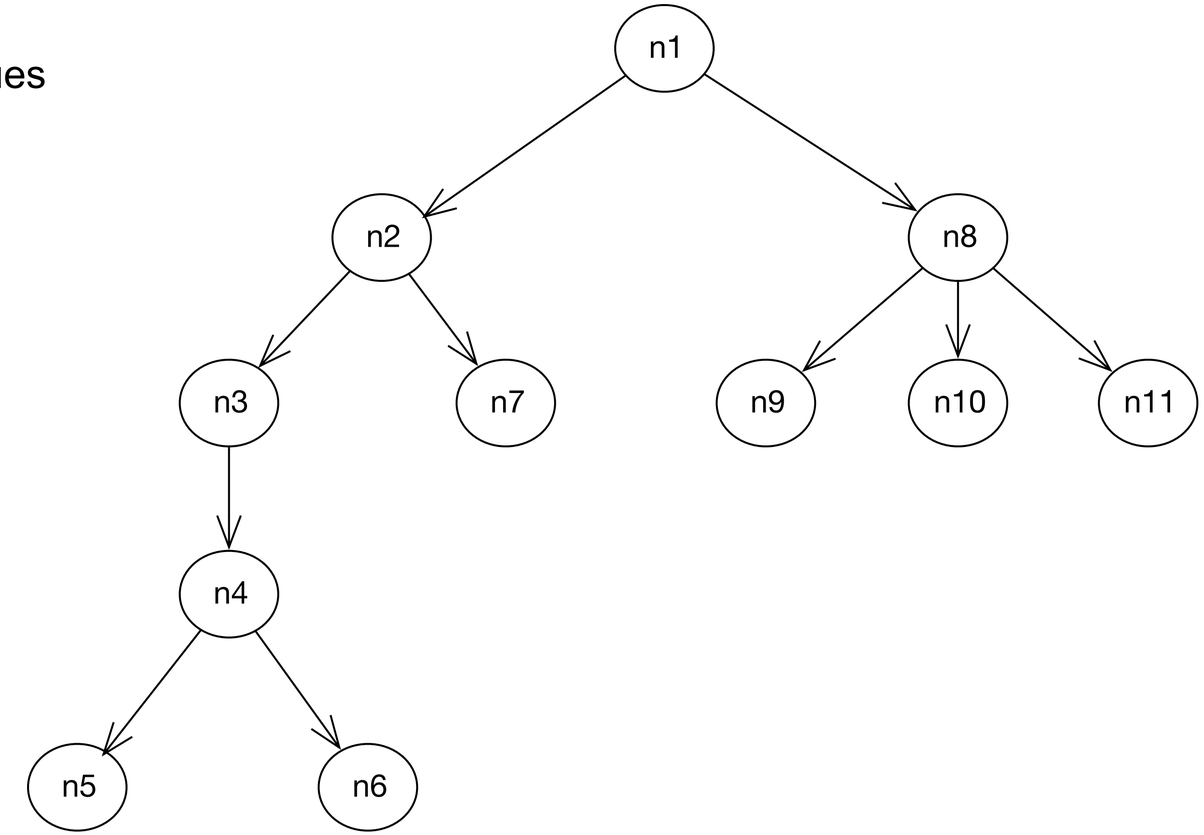
```
def enlever_file (a) :
    n = len (a)
    try:
        v = a[0] = a[n-1]
        i = 0
        while 2*i + 1 < n-1 :
            j = 2*i + 1
            if j + 1 < n-1 :
                if a[j+1] > a[j] :
                    j = j + 1
            if v >= a[j] :
                break
            a[i] = a[j]; i = j
        a[i] = v
        del a[n-1]
    except Exception:
        print ('erreur')
```



Exercice (*Heapsort*) écrire la fonction `trier_par_tas (a)` qui trie le tableau *a* en `nlog(n)` opérations à l'aide de tas

Les arbres en informatique

- les files de priorité utilisent des arbres binaires quasi parfaits, qu'on peut représenter par des tas
- comment représenter un arbre quelconque ?
- on utilise des structures de données dynamiques
- en Python, ce sont des **objets**
- chaque noeud est un objet



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)
(10, 20)

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

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

nouvel objet de la classe `Point`

Classes et objets

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

```
class Point:
    # comme avant

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

`__le__` est appelé par `<=`

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

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

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

on vérifie que q est dans le quadrant inférieur droit

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

Classes et héritage

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

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

on appelle le constructeur de `Rectangle`

- un carré est un rectangle particulier

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

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

Exercice écrire la classe `Triangle`

Exercice écrire les méthodes `perimetre` et `surface`

Conclusion

VU:

- arbres
- files
- classes et objets

TODO list

- arbres et objets
- arbres de recherche