Informatique et Programmation

Cours 6

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

Plan

- recherche en table par interpolation
- vérificateur de français
- hachage
- arbres
- files de priorité
- classes
- objets

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

Recherche par interpolation

- on a une connaissance de la distribution des clés (par exemple uniforme)
- au lieu de comparer à la clé du milieu, on peut comparer à une clé plus proche de la distribution

```
def recherche_interpolation (x, a) :
    g = 0; d = len(a) - 1
    lo = a[g]; hi = a[d]
    while g <= d  and lo <= x <= hi :
        if hi == lo:
           m = g
        else :
           m = (g * (hi - x) + d * (x - lo) ) // (hi - lo)
        if x == a[m] :
            return True
        elif x < a[m] :
            d = m - 1; hi = a[d]
        else :
            g = m + 1; lo = a[g]
    return False
```

- pour une distribution uniforme, cela donne de l'ordre de `log(log(n))` opérations

`4.16` **opérations pour table de** `239210` **entrées !!**

Lecture de fichier

- pour lire un fichier, on l'ouvre en mode lecture seule ‘r’ (read)

  et on le lit en séparant les lignes

```
def lire_lignes (nom) :
    f = open (nom, 'r')
    return f.read().splitlines()
```

- dans le répertoire courant, il y a un dictionnaire de français `french-iso de 239210 mots`

```
>>> a = lire_lignes ('french-iso')
>>> len(a)
239210

>>> for i in range (10) :
...     print (a[i])
...
a
abaca
abacule
abaissa
abaissable
abaissai
abaissaient
abaissais
abaissait
abaissant
```

Recherche d’un mot français

```
def val (x) :
   s = 0
   for i in range (6) :
      c = ord (x[i]) - ord (‘a’) if i < len (x) else 0
      s = 32 * s + c
   return s
```

- val (x) est une fonction croissante qui transforme le mot x en entier

- ord(c) retourne le code du caractère c

```
>>> a = lire_lignes (‘french-iso’)
>>> tri_Fusion (a)
>>> recherche_dict ('jouer', a)
True
>>> recherche_dict ('jouera', a)
True
>>> recherche_dict ('jouerra', a)
False
```

- on applique la recherche par interpolation

```
def recherche_dict (x, a) :
    g = 0; d = len(a) - 1
    v = val(x); lo = val(a[g]); hi = val(a[d])
    while g <= d and lo <= v <= hi :
        if hi == lo:
           m = g
        else :
           m = (g * (hi - v) + d * (v - lo) ) // (hi - lo)
        if x == a[m] :
            return True
        elif x < a[m] :
            d = m - 1; hi = val(a[d])
        else :
            g = m + 1; lo = val(a[g])
    return False
```

Recherche d’un mot français

• et on peut faire un vérificateur de français

```
a = lire_lignes (‘french-iso’)
tri_Fusion (a)

b = lire_mots_fichier (‘texte.txt’)
for w in b:
   if not recherche_dict (w, a):
      print (w)
```

```
def lire_mots_fichier (nom) :
    f = open (nom, 'r')
    return f.read.split()
```

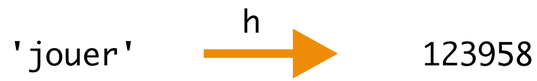
• le vérificateur n’est pas parfait

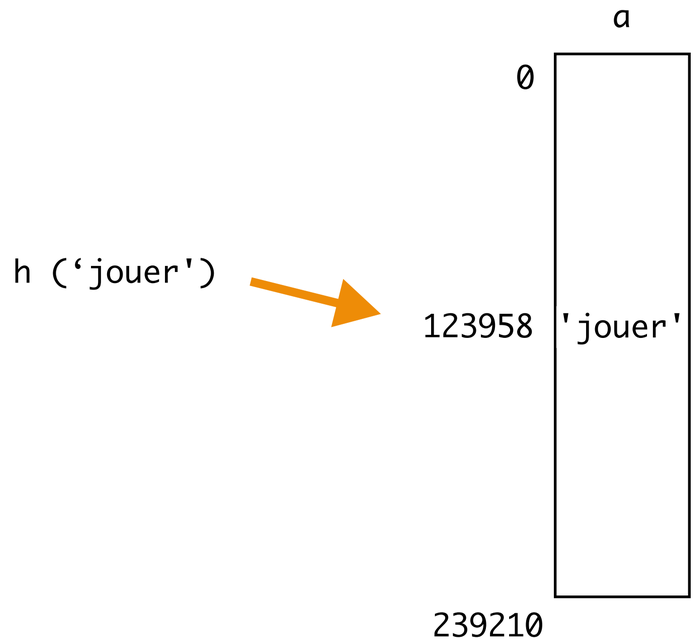
-- le dictionnaire `french-iso` ne contient pas les noms propres

-- la méthode `split()` ne tient pas compte des ponctuations

Hachage

- une fonction de hachage donne l'entrée dans la table en fonction de la clé





- comment réaliser une telle fonction ?

Hachage

- plein de fonctions de hachage possibles
- `h (x, M)` calcule le hash d'un mot `x` en tenant compte du code de chacun de ses caractères
- `ord(c)` retourne le code du caractère `c`
- le résultat est rendu modulo `M` premier pour assurer une meilleure répartition des valeurs de `h`

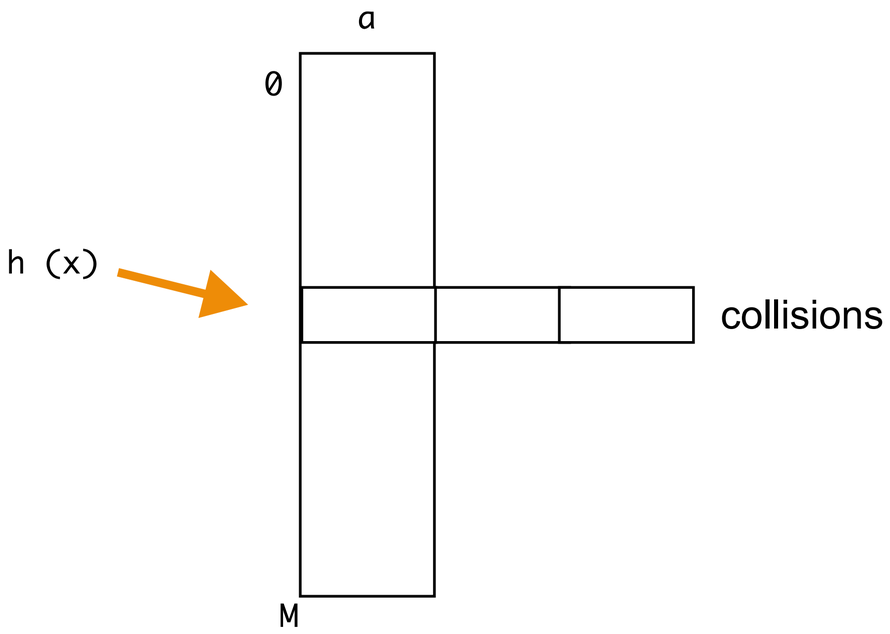
```
def h (x, M) :
   s = 0
   for c in x :
      s = (256 * s + ord(c)) % M
   return s
```

Hachage

• plusieurs clés peuvent avoir la même valeur de h: c'est une collision

• on réorganise la table en créant des classes d'équivalences (autour des collisions)

```
def recherche (x, a) :
    i = h (x, 61)
    for p in a[i] :
        if p == x :
            return True
    return False
```



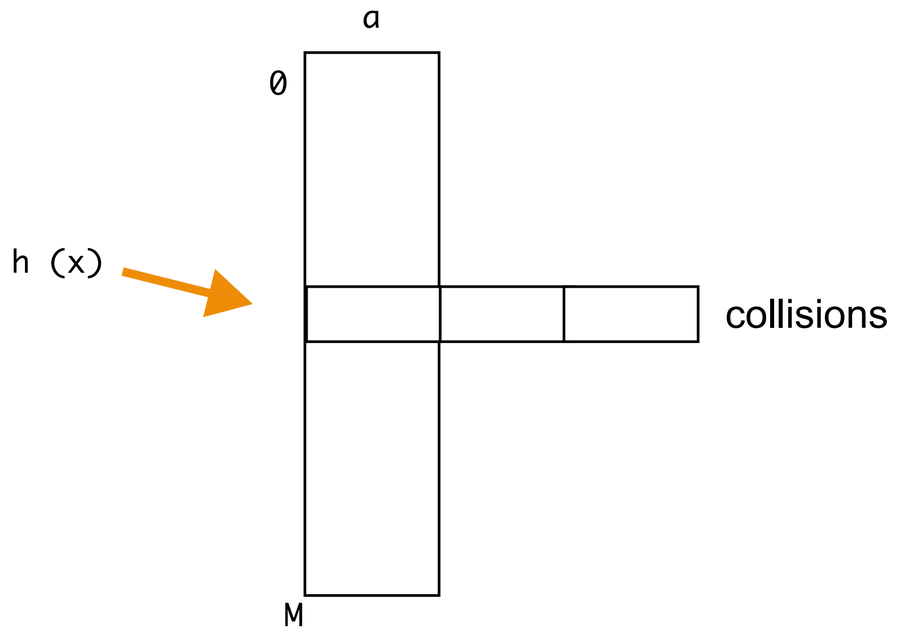
• ne pas confondre 2 types de fonctions de hachage:

- pour les tables de hachage (*hash tables*) (on cherche des valeurs de hachage dans un petit ensemble)
- pour la cryptographie: md5, sha1, sha-512 (on veut distinguer les valeurs avec de longues empreintes)

Hachage

- pour insérer un nouvel élément dans une table

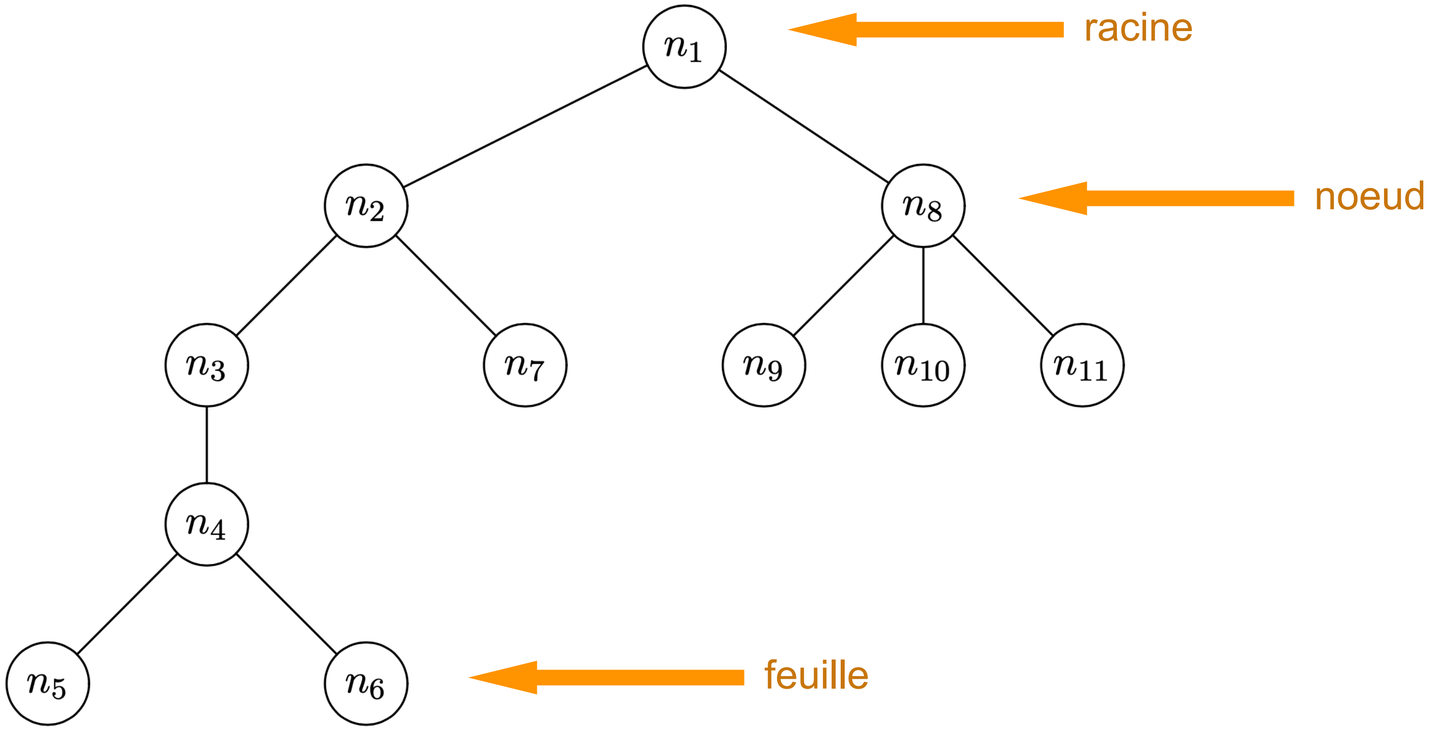
```
def h (x, M) :
    s = 0
    for c in x :
        s = (256 * s + ord(c)) % M
    return s


def nouvelle_table (n) :
     return [[ ] for i in range(n)]


a = nouvelle_table (61)

def insertion (x, a) :
     i = h (x, 61)
     if not x in a[i] :
       a[i].append (x)
```

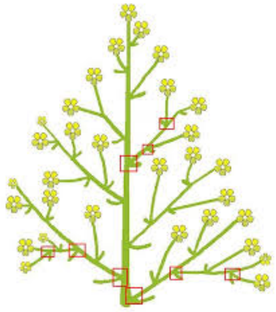


Exercice insérer les 30 premiers mots du fichier `texte.txt`

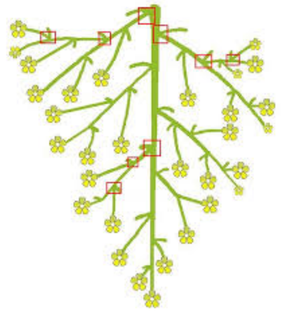
Les arbres en informatique

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





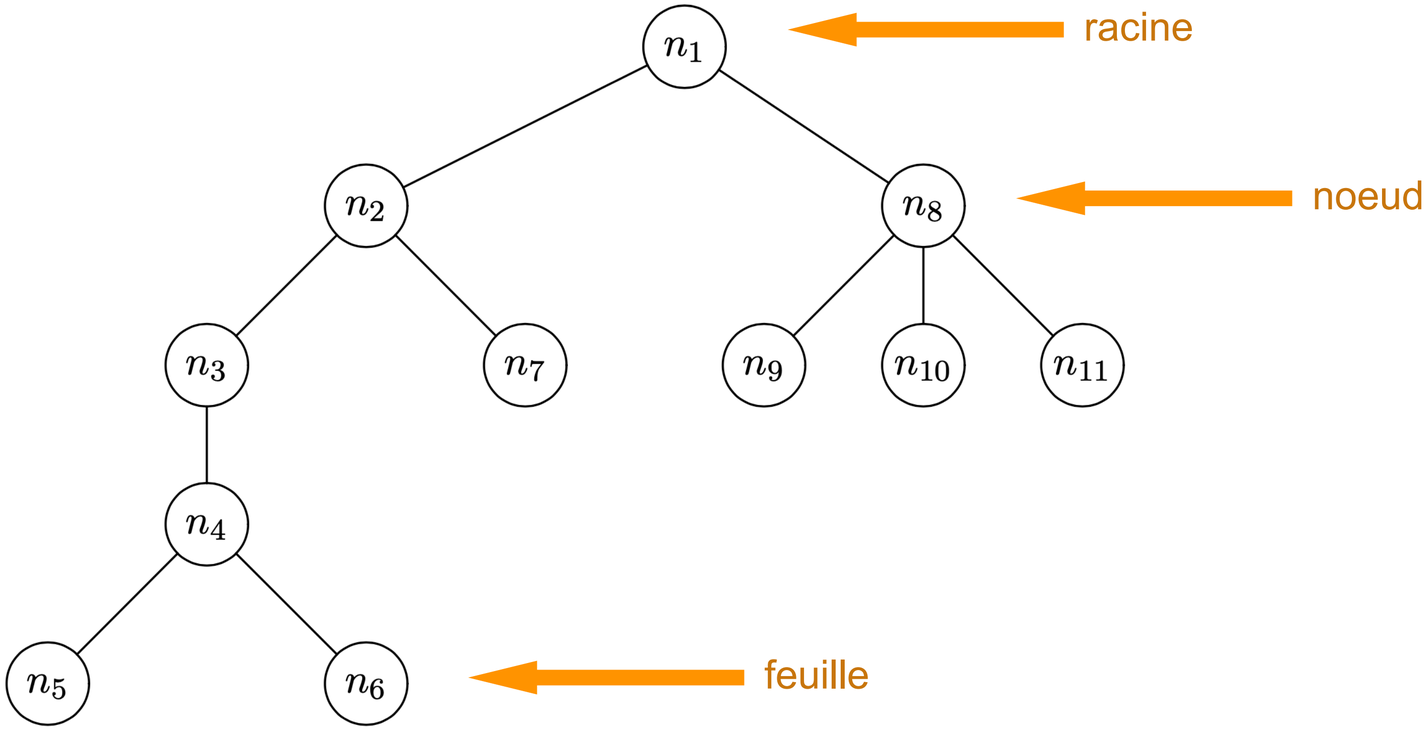
botanique

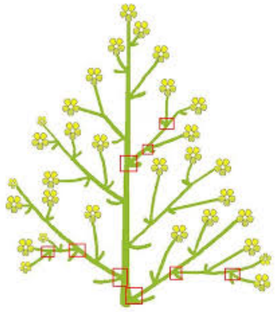


informatique

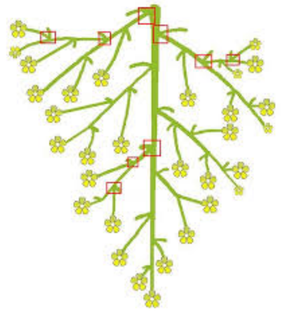
Les arbres en informatique

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





botanique



informatique

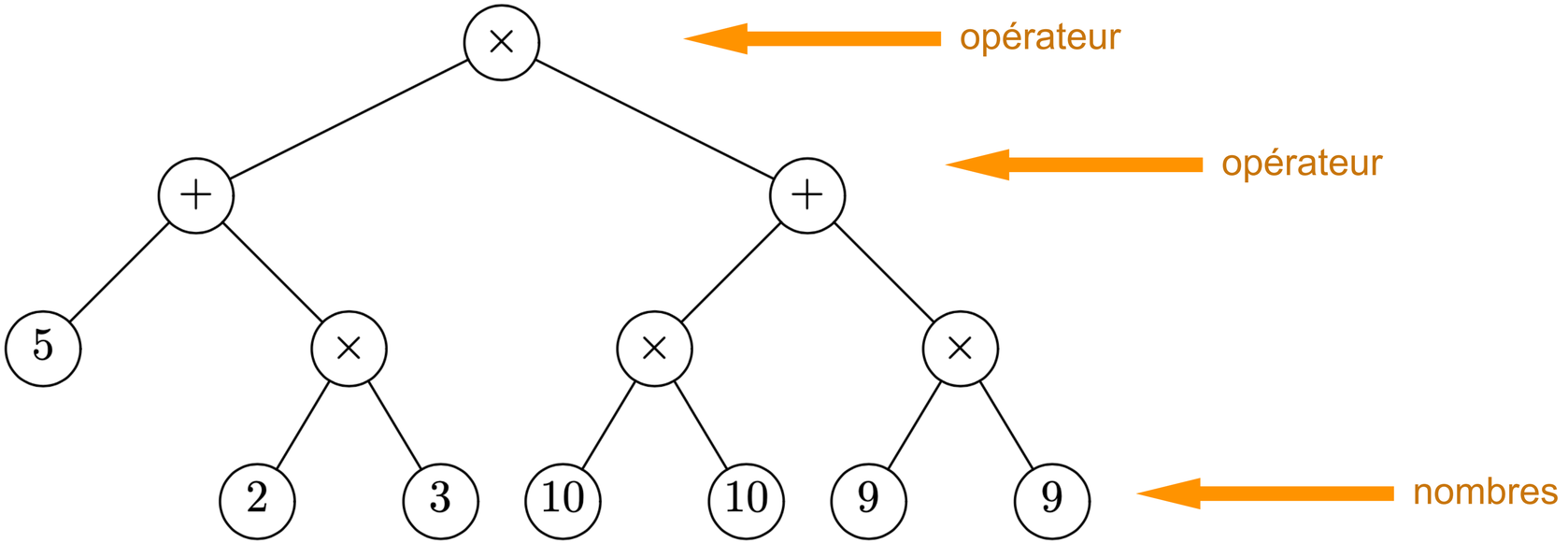
n2 est un **ancêtre** de n4

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]



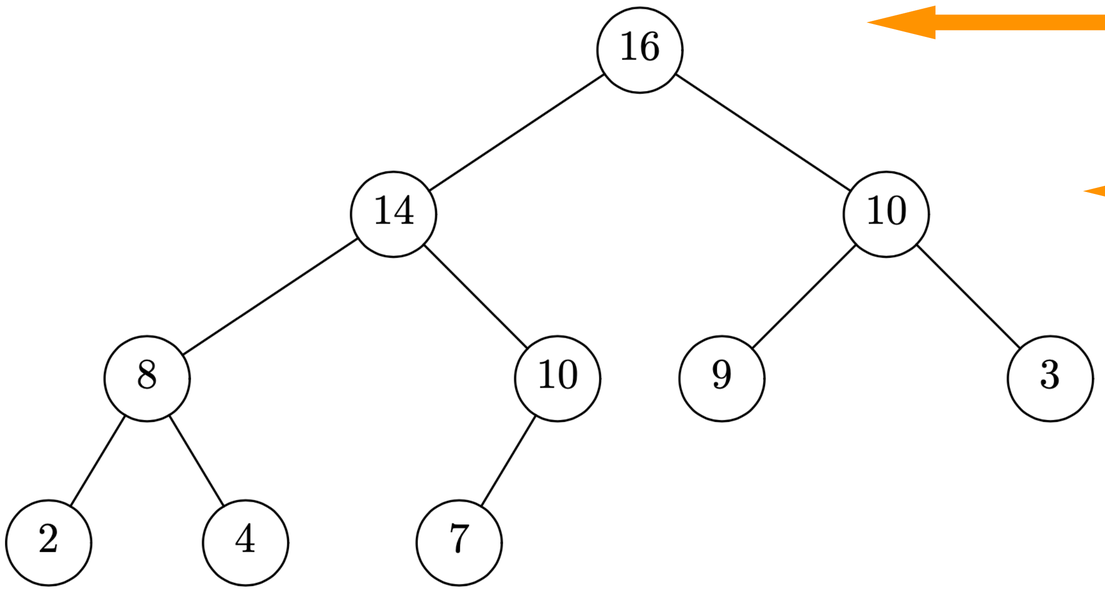
pour représenter une **expression arithmétique**

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

[plus besoin de parenthèses]

Les arbres en informatique

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



maximum

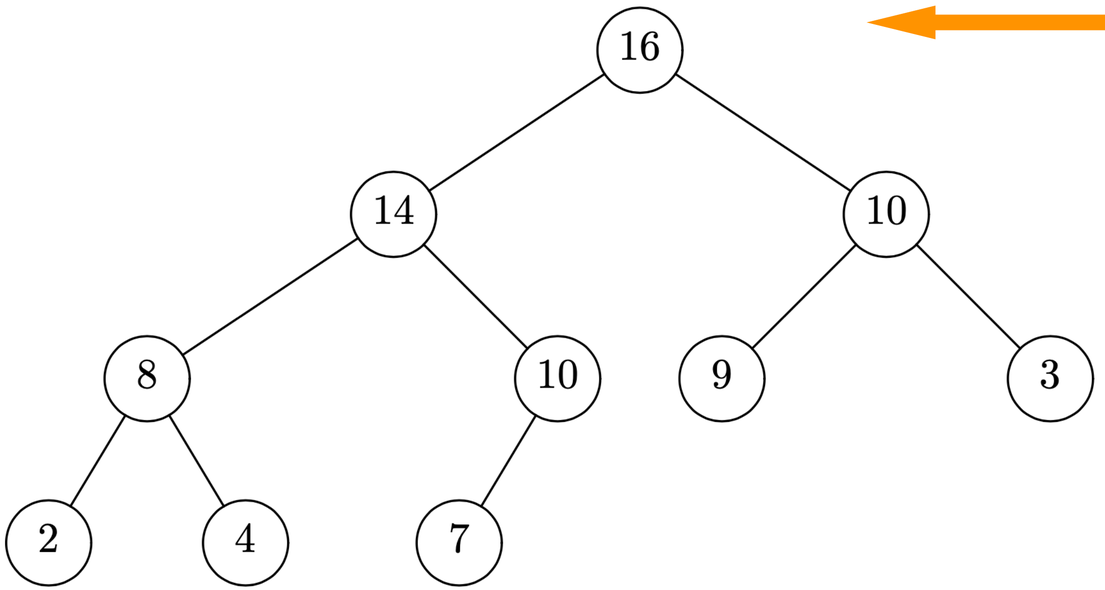
arbre binaire

tout noeud a au plus 2 fils

[ici un ancêtre a une valeur plus élevée qu'un de ses descendants]

Files de priorité

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



maximum

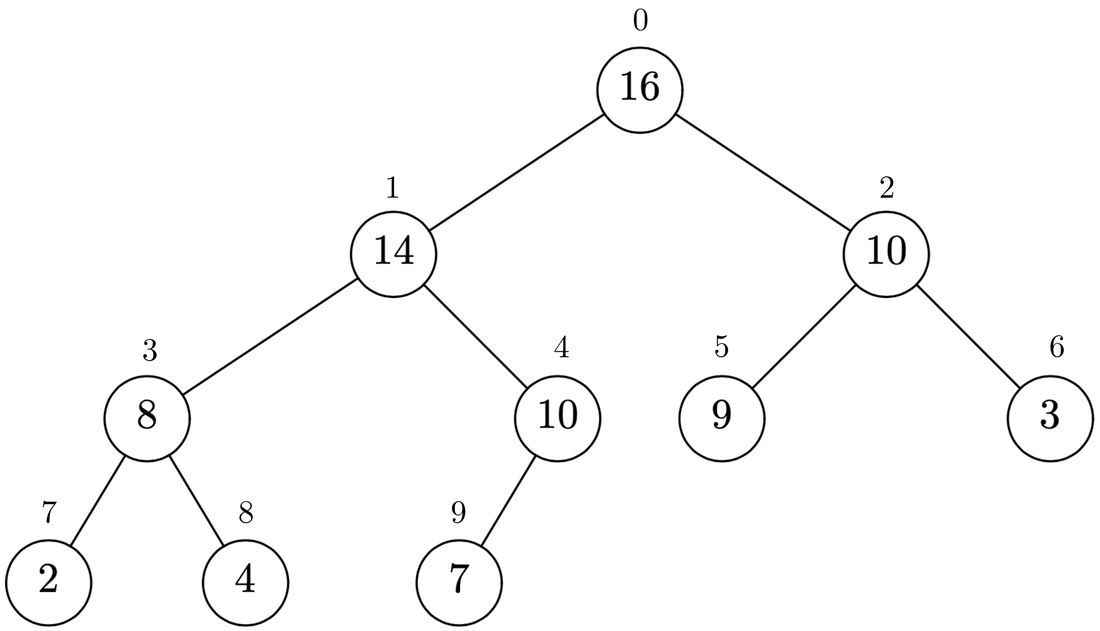
tous les noeuds internes ont 2 fils sauf les plus éloignés de la racine.

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$(i) = \lfloor (i-1)/2 \rfloor$

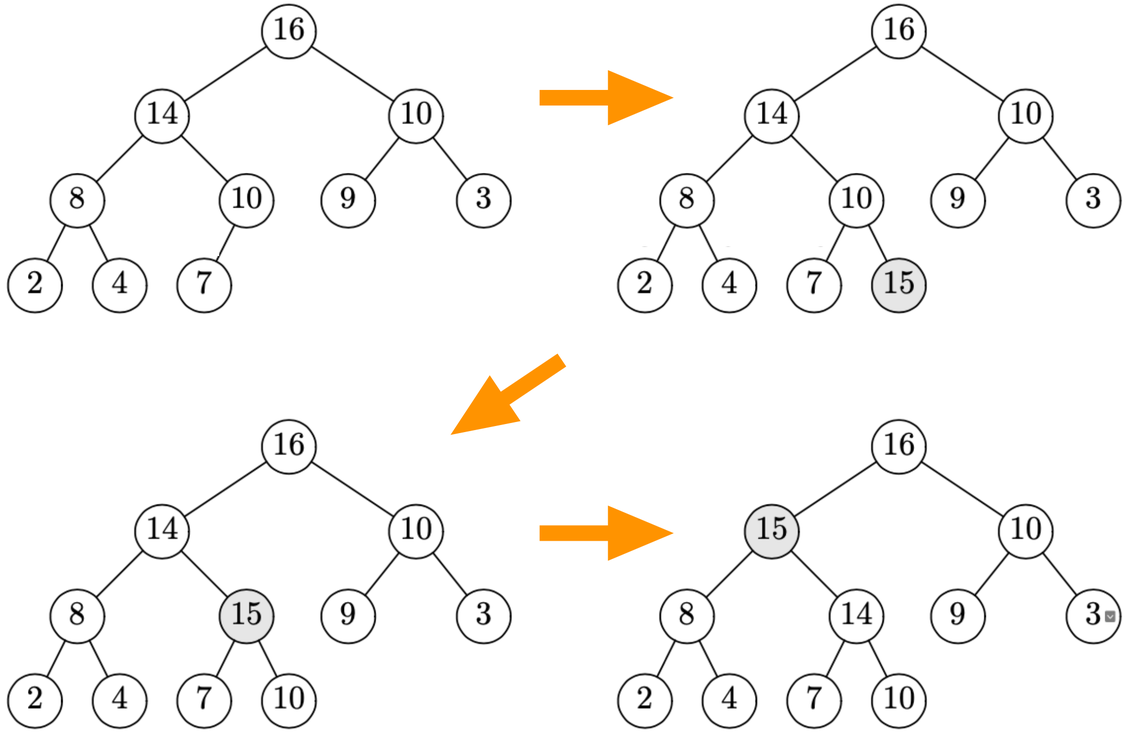
- on utilise un **tas** (*heap*) c'est-à-dire un tableau indicé par les numéros figurant au dessus de chaque noeud

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
```

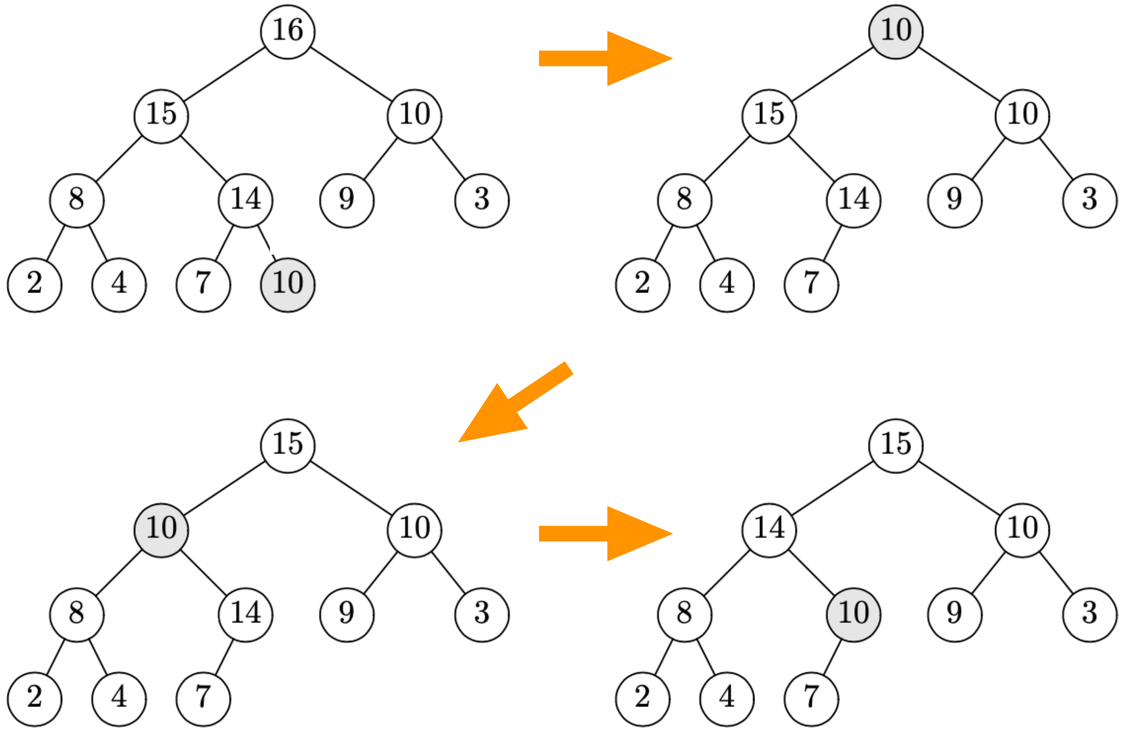
• si `n` est la longueur de la file, l'ajout d'un élément ne fait pas plus que `log(n)` opérations

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

Files de priorité

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

```
def supprimer_file (a) :
    n = len (a)
    v = a[0] = a[n-1]
    del 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
```

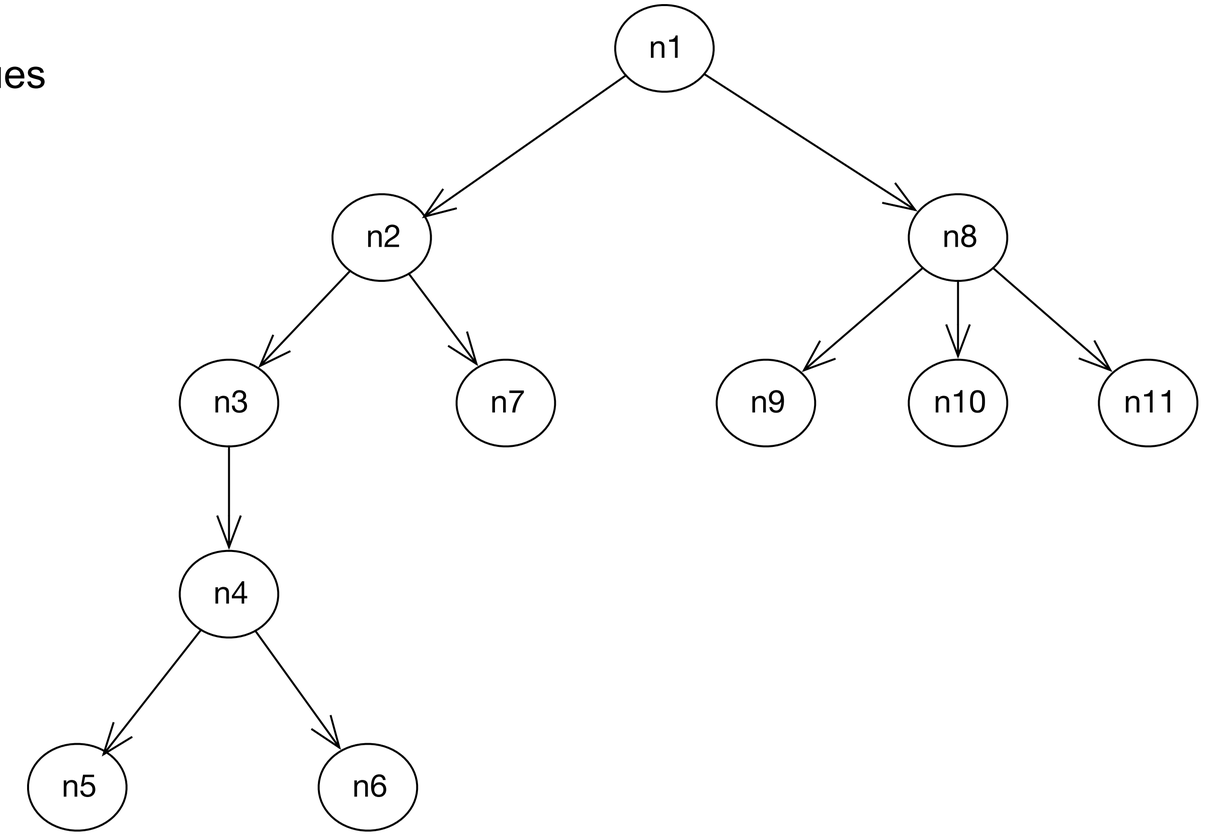


Exercice (*Heapsort*) écrire la fonction `trier_par_tas (a)` qui trie le tableau *a* en `nlog(n)` opérations à l'aide de tas

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 "(%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 `+`

```
>>> p1 = Point (10, 20)
```

nouvel objet de la classe `Point`

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

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

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

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)
>>> print (r)
((10, 20), (50, 70))
```

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`