Structures de données usuelles

Stage Liesse

François Pessaux

`https://perso.ensta-paris.fr/~pessaux/teaching/liesse`

Laboratoire U2IS
ENSTA Paris
2021

ENSTA

IP PARIS

Bienvenue à toutes et tous. . .

- Le sujet :

  étudier les structures de données usuelles

- Le présentateur :
  - Enseignant-chercheur en informatique à l'ENSTA depuis +9 ans.
  - Responsable de l'informatique en 1ère année.
  - Cours en 1A et 2A.
  - Responsable de parcours de 3A.
- Le plan :
  - Tableaux
  - Piles
  - Files
  - Listes chaînées
  - Arbres
  - Files de priorité
  - Graphes

Recherche en table

« Pourquoi cette question ? »

- Problème : retrouver une information à partir d'une clef (info discriminante) qui lui est associée.
- Exemples :
  - Dictionnaire : mot $\mapsto$ définition.
  - Annuaire : (nom + prénom) $\mapsto$ (adresse + téléphone).
  - Courbe temporelle : date $\mapsto$ mesure.
- Opérations souhaitées :
  - Insertion.
  - Recherche.
  - Suppression.

Choix de la structure de données à utiliser ?

Paires « *clef-valeur* »

- But : gérer un ensemble d'éléments (`clef`, `valeur`).
- La `clef` donne accès à l'information `valeur`.
- Les clefs représentent un ensemble : m éléments indexés de 0 à $m-1$.
  - Clefs entières sur 16 bits : $2^{16} = 65536$, d'où clefs de 0 à 65535.
  - Téléphone, 10 chiffres décimaux : $10^{10}, \approx 2^{33}$ clefs.
  - Noms sur 8 lettres : $26^8 \approx 2^{37}$ clefs.
- Les clefs doivent être idéalement discriminantes : 1 clef $\rightarrow$ 1 valeur.
- Problème : généralement, nombre de clefs possibles >> nombre de valeurs à enregistrer.

Plusieurs solutions

- Pas de solution unique.
- Compromis entre :
  - Complexité spatiale (coût mémoire du stockage).
  - Complexité temporelle des opérations (insertion, recherche, suppression).
- Cette notion de « table » est générale :
  - stockage en mémoire ou sur disque → ± même problème.

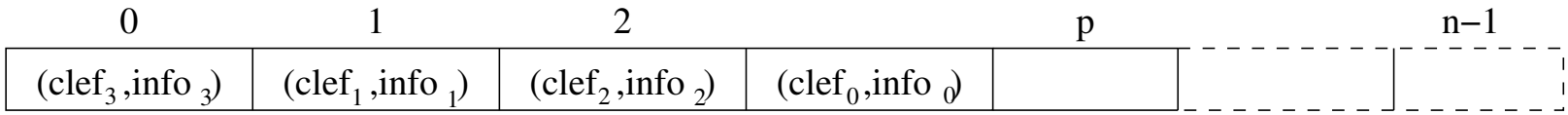
L'adressage direct

- Les résultats sportifs : 3ème du classement → 3ème «*case* ».

- Utilisation d'un tableau de taille m (nombre de clefs possibles).
- Info liée à la clef i à la «*case* » d'indice i : pas de stockage de la clef.
- ⇒ Complexité spatiale en $\theta(m)$, ou $\theta(m \times \text{taille info})$.
- **Mais** : si clef = mot de 10 lettres, $m = 2^{47}$! ! !
- ⇒ Même avec 1 seul bit d'info → 16 To de stockage ! ! !
- ⇒ Irréalisable en pratique dans la majorité des cas.

Recherche séquentielle (1)

- Utilisation d'un tableau de taille *n* (nombre d'infos).
- Infos stockées dans l'ordre « *d'arrivée* ».
- Mémorisation d'un indice *p* dénotant la 1ère « *case* » libre.



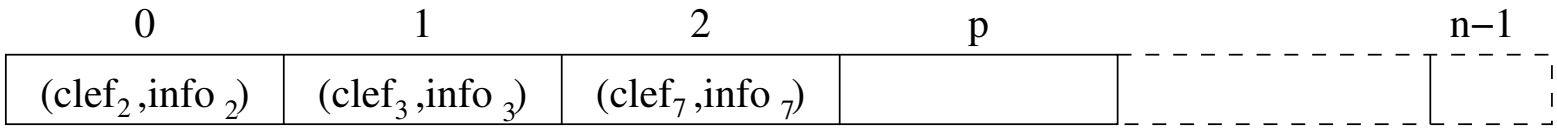
- Complexité spatiale en $\theta(n)$ (complexité minimale).
- **Insertion** : `t[p]` $\leftarrow$ (`clef`, `info`)
  - $\Rightarrow$ Complexité en $\theta(1)$.
  - Si recherche préalable pour éviter les doublons : ajout à la complexité.

Recherche séquentielle (2)

- Recherche trop lente.
- Utilisable si l'on insère des éléments mais qu'on les lit rarement.
- Exemple proche : les journaux (« *logs* »).
- Si les comparaisons sont complexes, devient excessivement coûteux :
  - Comparer des entiers : trivial, en $\theta(1)$.
  - Comparer 2 chaînes de caractères, $l = min(|s_1|, |s_2|) \Rightarrow \theta(l)$.
  - Comparer sur des clefs étant elles-mêmes des tableaux... ☹
  - Ou pire ...

Recherche dichotomique (1)

- La recherche dans le dictionnaire.
- Nécessite un tableau de taille *n* dont les éléments sont triés sur les clefs.
- Dans le dictionnaire : tri des mots par ordre lexicographique.



- Processus de recherche dichotomique :
  - Accès au milieu du tableau.
  - Si clef trouvée = clef recherchée → trouvé.
  - Si clef trouvée < clef recherchée → recherche dichotomique dans la partie « *droite* » du tableau.
  - Si clef trouvée > clef recherchée → recherche dichotomique dans la partie « *gauche* » du tableau.

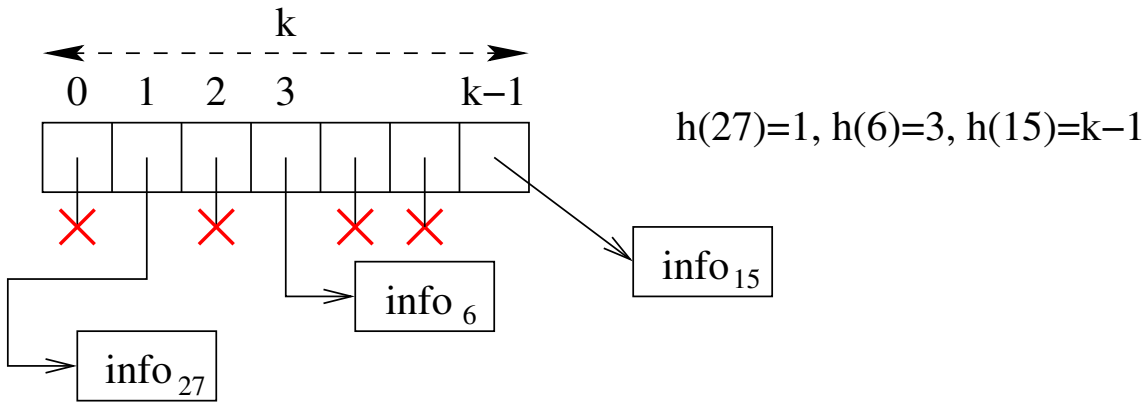
Recherche dichotomique (2)

- **Recherche** : parcours dichotomique jusqu'à trouver la clef.
  - Complexité en $\theta(log\ n)$.
- **Insertion** : recherche puis décalage.
  - Complexité en $\theta(n)$.
- **Suppression** : recherche puis décalage.
  - Complexité en $\theta(n)$.

- Bonne complexité de recherche.
- **Mais** insertion (et suppression) trop chère(s).
- Insertion : plus rentable d'insérer tout puis de trier tout le tableau d'un coup.

Table de hachage, dictionnaire (« *hastable* »)

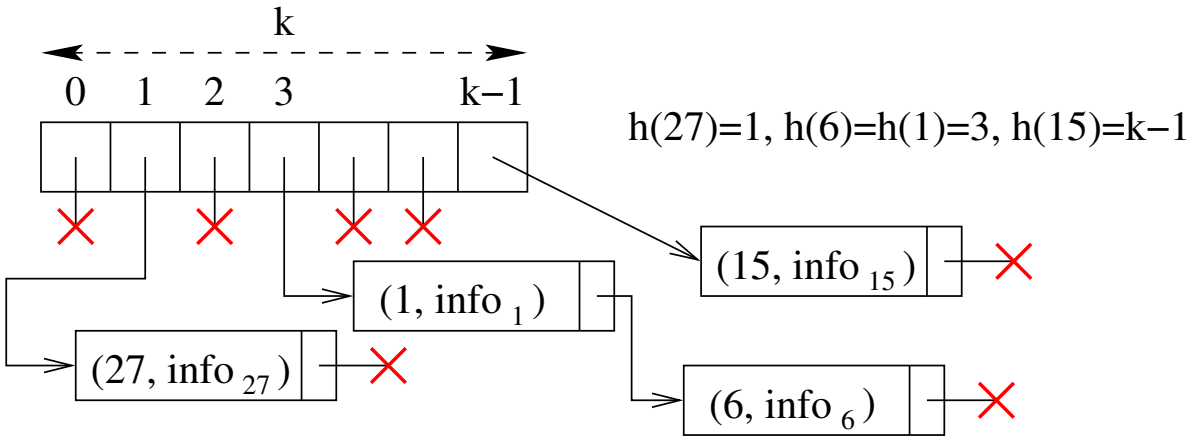
- Réduire l'espace des clefs possibles, m, à un espace plus petit, k des clefs réellement utiles).
- Fonction de hachage $h : [0; m-1] \to [0; k-1]$ avec $k < m$.
- On utilise un tableau de taille k.
- On stocke chaque information dans le tableau à l'indice $h(clef)$.
  - Chaque case contient l'information dont le hachage de la clef vaut l'indice de cette case.



- Mais que se passe-t-il si 2 clefs ont le même « hash » ?
  - Collision.

Table de hachage et collisions

- Collision : plusieurs infos avec le même hachage de clef.
- Au lieu de mettre 1 info dans la table, on met une « *liste* » d'infos.
- Toutes les infos d'une liste ont le même hachage de leur clef.



- Autre solution : adressage ouvert → utiliser la 1ère case contiguë libre
  - ⚠ : si suppression, besoin de mémoriser les éléments « mal logés » ou parcourir la table.

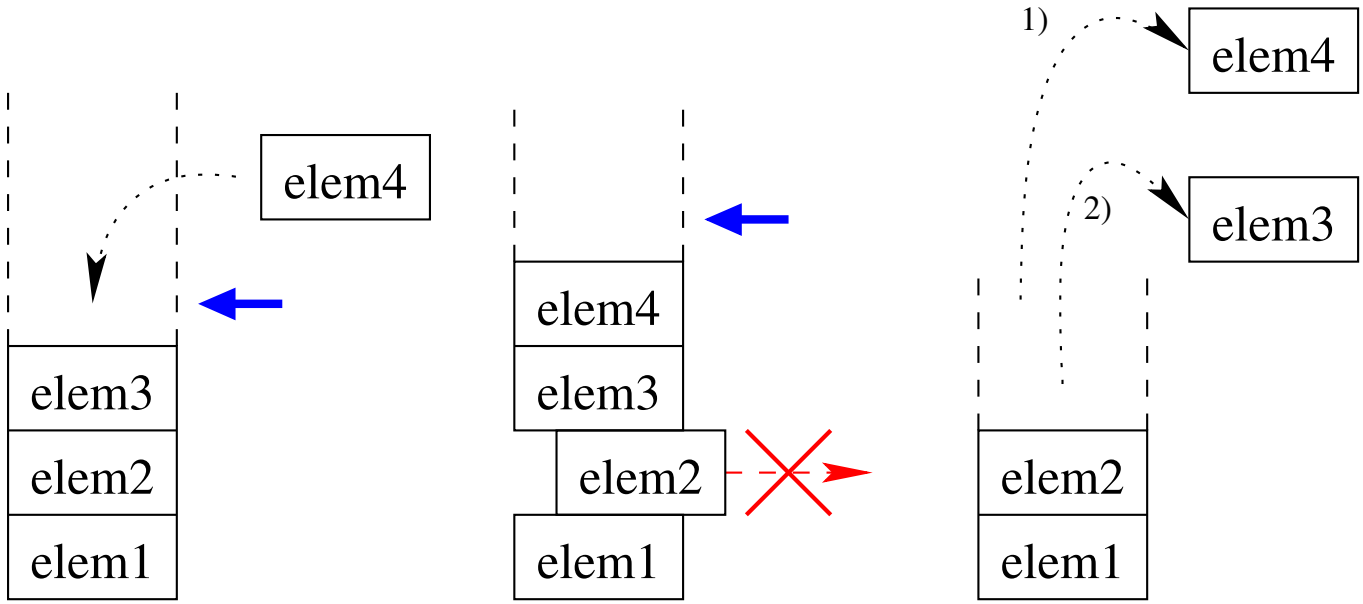
Complexité des tables de hachage

- Les complexités dépendent du facteur de remplissage ρ de la table.
- ρ = nombre éléments dans la table / taille de la table.
- Complexités (en moyenne) :
  - spatiale : $\theta(k+n)$ (taille de la table + taille des données),
  - recherche : $\theta(\rho)$,
  - insertion : $\theta(1)$ (chaînage), $\theta(\rho)$ (adressage ouvert),
  - suppression : $\theta(\rho)$.
- Nécessité d'une une bonne fonction de hachage (non biaisée sur les entrées).
- Nécessité de connaître n à l'avance pour choisir $k \approx n$.

Piles

« Quand utiliser une pile ? » (« stack »)

- Comme dans la vie courante : stocker des choses comme une pile d'assiettes.
- ⇒ Dernier posé au sommet → premier retiré.
- En Anglais : LIFO (« **L***ast* **I***n* **F***irst* **O***ut* »).
- On peut consulter les éléments sous le sommet mais pas les retirer tant que ceux au-dessus sont dans la pile.



Application

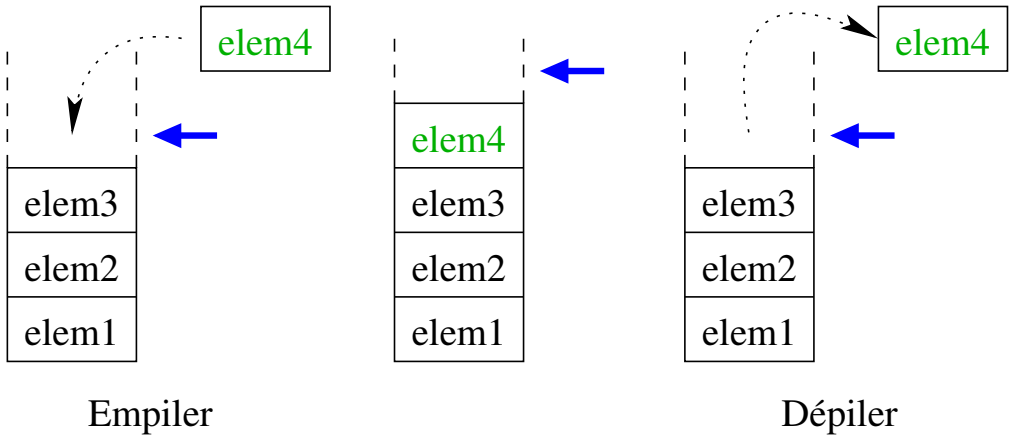
- Garder trace de traitements non complétés (« backtracking »).
- Garder trace d'un historique de traitements.

- Base de nombreuses machines virtuelles (Java par exemple).
  - Notation « *RPN* » : notation postfixe.
  - Opérateurs après les opérandes.
  - Comme les anciennes calculatrices HP.
  - Opérandes sur la pile, opérateur opérant sur les éléments de la pile.

- . . . Même structure que la pile d'exécution d'un programme. . .

Opérations sur une pile

- **Empiler** (« *push* ») : Ajouter un élément au sommet.
  - `push` : (elem × stack) → stack
- **Dépiler** (« *pop* ») : Retirer l'élément du sommet.
  - `pop` : stack → (elem × stack)
- **Consulter** (« *peek* ») : Accéder à un élément sans le retirer.
  - `peek` : (stack × int) → elem
- Le « *pointeur de pile* » indique la prochaine place libre.
- Le « *pointeur de pile* » varie au cours des opérations (sauf `peek`).
- On peut tester si la pile est vide.



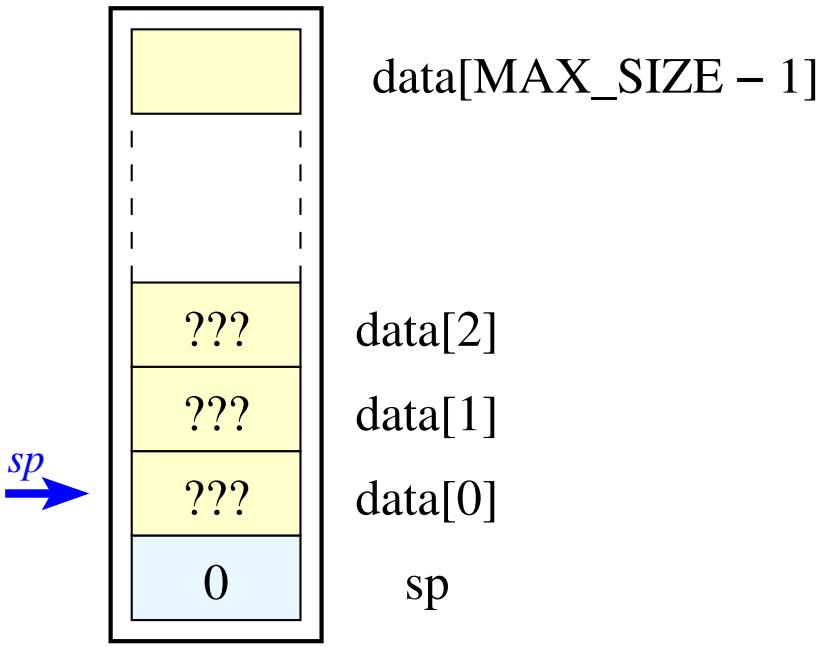
Pile : implantation avec un tableau

- Ici, pile d'entiers avec une taille maximale de 64.

```
class Stack :
  def __init__ (self) :
    self.max_size = 64        # Taille de la pile.
    self.sp = 0               # Pointeur de pile.
    self.storage = [0] * self.max_size  # Mémoire de pile.
```

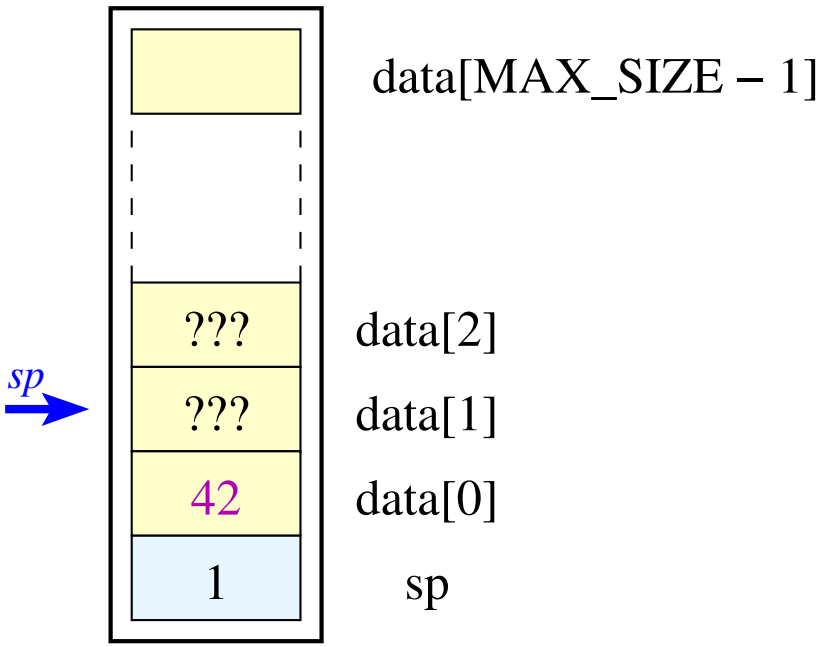
- Pile ≡ simple tableau.
- « *Pointeur* » de pile = entier positif :
  - Représente l'indice de la prochaine « *case* » libre.
  - ⇒ Représente le nombre d'éléments présents dans la pile.
- On ajoute et on retire toujours « *par le haut* ».

Pile : fonctionnement avec un tableau (1)



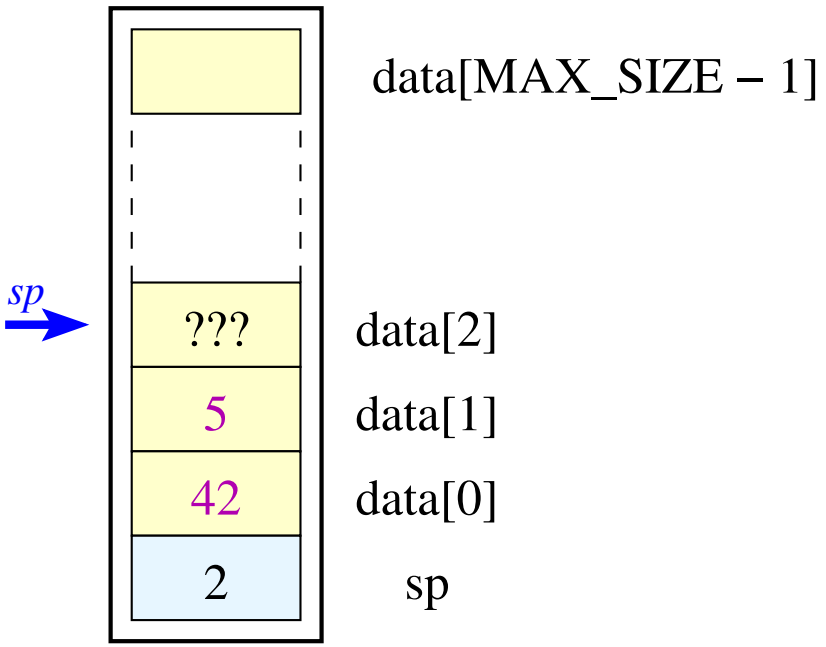
- Init : Pile vide, sp = 0.

Pile : fonctionnement avec un tableau (2)



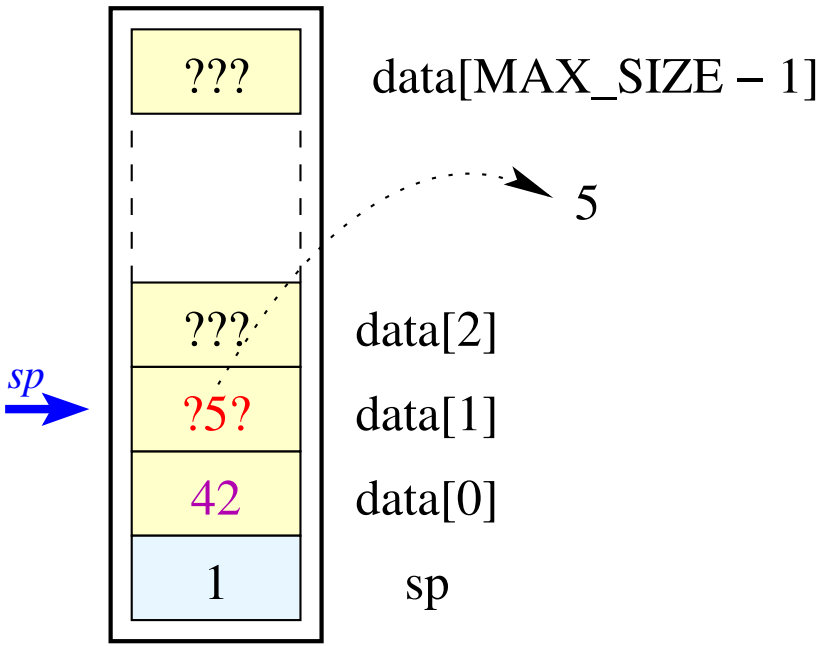
- push 42 :
  - Vérifier que sp < MAX_SIZE
  - 42 mis à l'indice sp ⇒ 0
  - Incrémenter sp : sp ← sp + 1 ⇒ 1

Pile : fonctionnement avec un tableau (3)



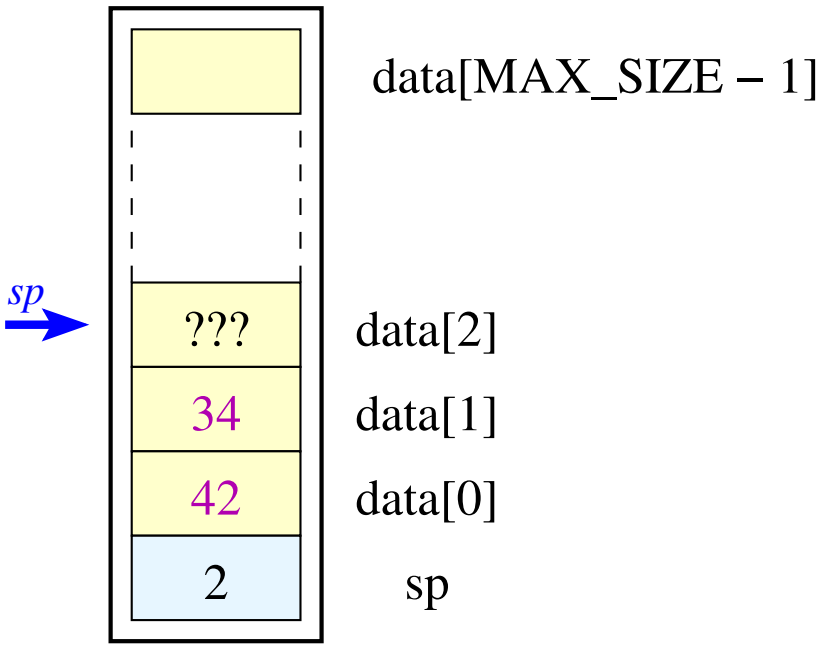
- push 5 :
  - Vérifier que sp < MAX_SIZE
  - 5 mis à l'indice sp ⇒ 1
  - Incrémenter sp : sp ← sp + 1 ⇒ 2

Pile : fonctionnement avec un tableau (4)



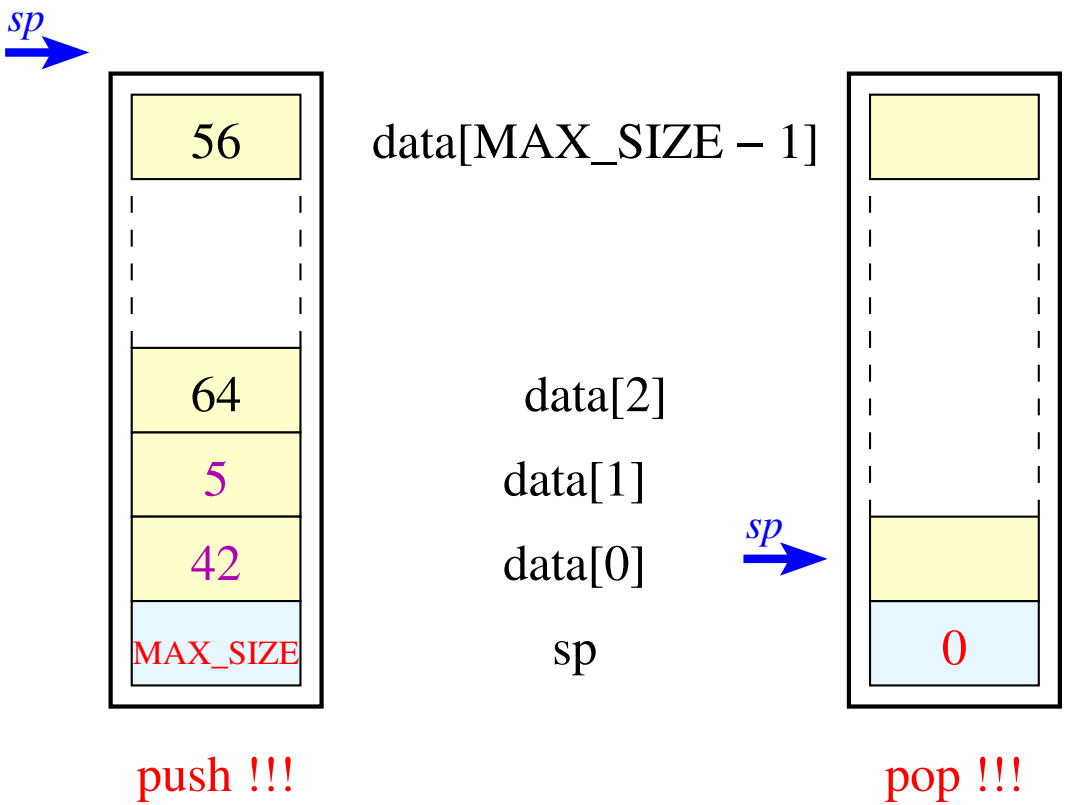
- pop :
  - Vérifier que `sp` > 0
  - Décrémenter `sp` : `sp` ← `sp` - 1 ⇒ 1
  - Retourner la valeur à l'indice `sp` (sommet de la pile)
- La valeur reste dans la pile mais n'est plus à utiliser.
- Elle sera écrasée au prochain `push`.

Pile : fonctionnement avec un tableau (5)



- push 34 :
  - Vérifier que sp < MAX_SIZE
  - 5 mis à l'indice sp ⇒ 1
  - Incrémenter sp : sp ← sp + 1 ⇒ 2
- Ancienne valeur ?5? écrasée par 34.

Pile : fonctionnement avec un tableau (6)



- push dans une pile pleine :
  - sp = MAX_SIZE.
  - ⇒ erreur.
- pop sur une pile vide :
  - sp = 0.
  - ⇒ erreur.

Pile : implantation en Python

```
class EmptyStack (Exception) : pass
class FullStack (Exception) : pass

class Stack : ...

  def pop (self) :
    if self.sp == 0 : raise (EmptyStack)
    self.sp = self.sp -1
    return self.storage[self.sp]

  def push (self, val) :
    if self.sp == self.max_size : raise (FullStack)
    self.storage[self.sp] = val
    self.sp = self.sp + 1
```

Pile : implantation en Python avec taille dynamique

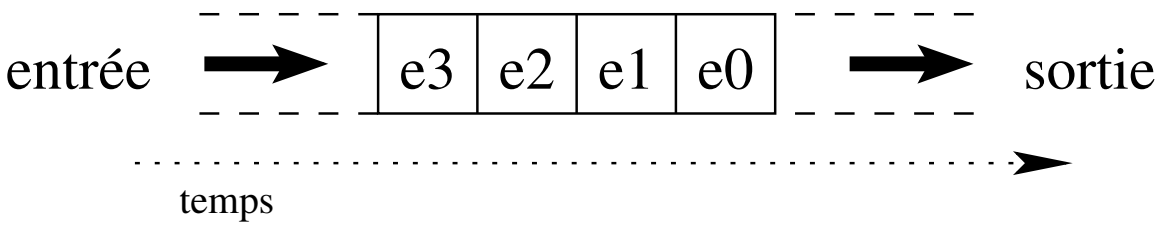
- Taille maximale n'est plus une constante
- Seul `push` change : en cas de pile pleine, on agrandit le tableau.

```
def push_dyn (self, val) :
  if self.sp == self.max_size :
    ndata = [0] * (self.max_size * 2)
    for i in range (0, self.max_size) :
      ndata[i] = self.storage[i]
    self.max_size = self.max_size * 2
    self.storage = ndata
  self.storage[self.sp] = val
  self.sp = self.sp + 1
```

Files

« Quand utiliser une file ? »

- Stocker des choses à traiter dans l'ordre d'arrivée (≈ file d'attente).
- ⇒ Premier inséré en queue → premier retiré en tête.
- En Anglais : FIFO (« **F***irst* **I***n* **F***irst* **O***ut* »).
- On peut consulter les éléments dans la file mais pas les retirer tant que ceux de devant sont dans la file.

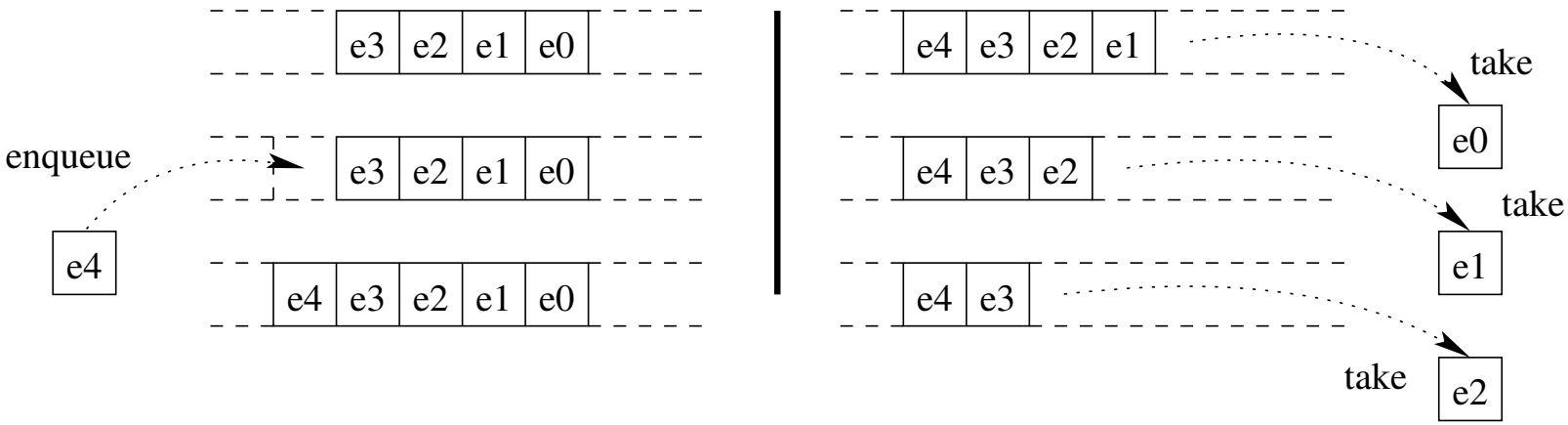


Applications

- Stocker des « transactions » à effectuer.
  - Mémoire tampon (« *buffer* »).
- Ordonnancer (très naïvement) les processus dans un OS multitâche.
- Parcours « en largeur » d'arbres ou de graphes (c.f. plus tard).
- Services client/serveur (WEB).

Opérations sur une file

- **Ajouter** (« *enqueue* ») : Ajouter un élément en queue.
  - `enqueue` : (elem × file) → file
- **Retirer** (« *take* ») : Retirer l'élément de tête.
  - `take` : file → (elem × file)
- **Consulter** (« *peek* ») : Accéder à un élément sans le retirer.
  - `peek` : (file × int) → elem
- On peut tester si la file est vide.

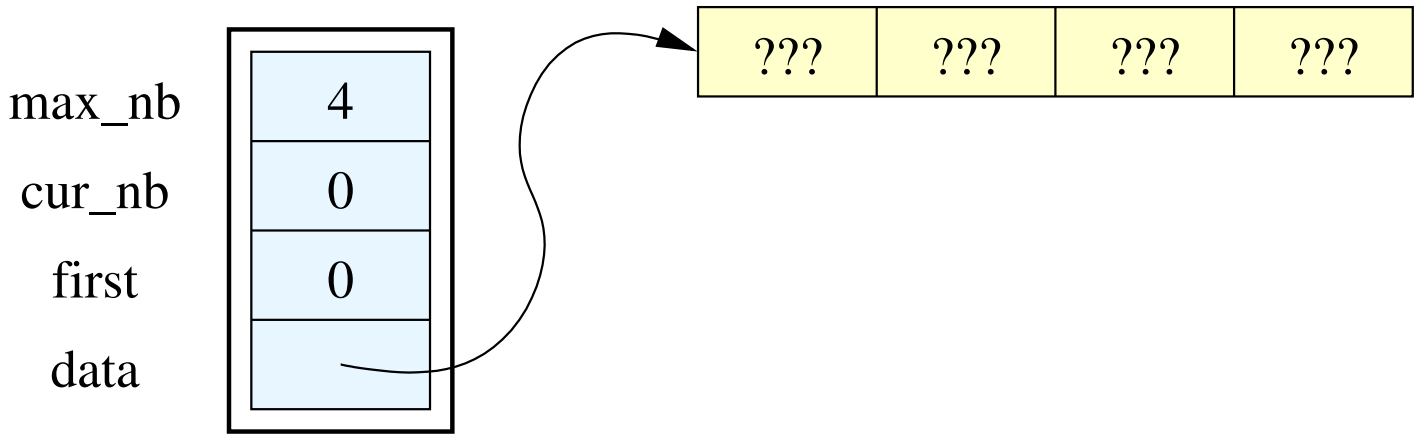


File : implémentation avec un tableau

- Utilisation d'un tableau avec taille bornée.
  - Comme les piles : on peut utiliser un tableau dynamique.
- Ajout des éléments en fin du tableau.
- Retrait des éléments en début de tableau.
- **Mais** on ne veut pas décaler tout le tableau (impossible en $\theta(1)$).
  - $\Rightarrow$ Utilisation du tableau de manière cyclique.
  - Arrivé au bout du tableau, on recommence au début.
- Besoin de garder l'indice du 1er élément (le + ancien encore vivant).

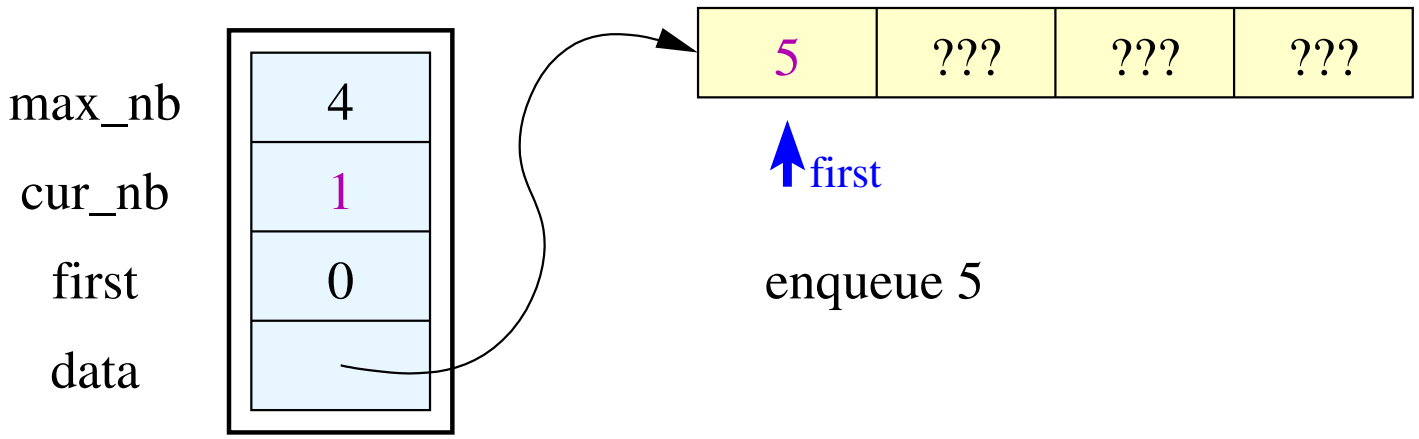
```
class Queue :
  def __init__(self) :
    self.max_nb = 6          # Nombre maximal d'éléments.
    self.cur_nb = 0          # Nombre actuel d'éléments.
    self.first = 0           # Indice du premier élément.
    self.storage = [0] * self.max_nb    # Stockage de la pile.
```

File : fonctionnement avec un tableau (1)



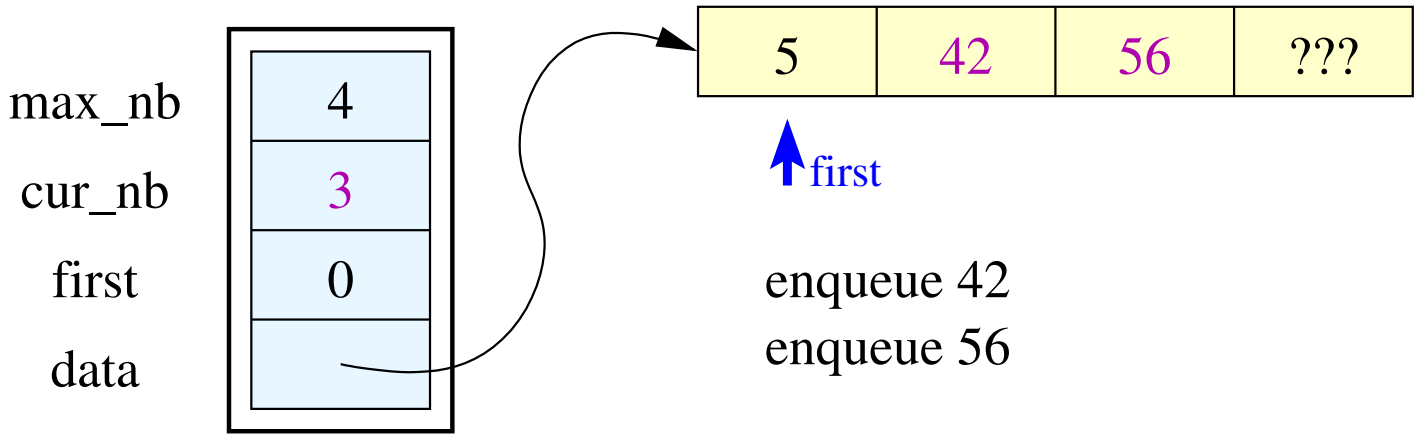
- Taille de la file : 4.
- Initialisation : file vide, i.e. aucune donnée.
- $\Rightarrow$ `cur_nb`, `first` $= 0$.

File : fonctionnement avec un tableau (2)



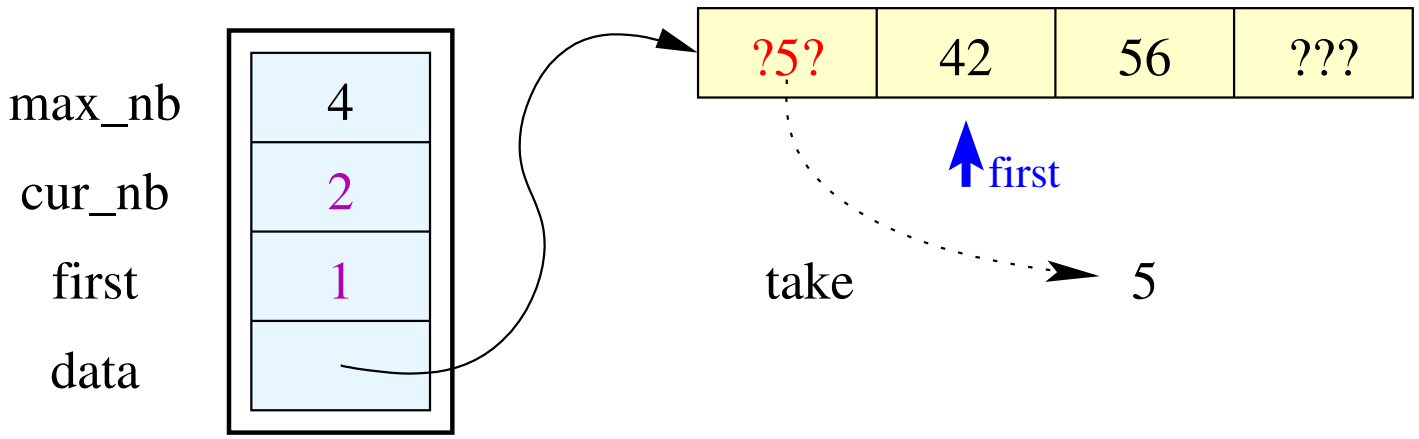
- enqueue 5 :
  - Vérifier que la file n'est pas pleine.
  - Ajouter au *« premier indice libre »*.
  - Incrémenter `nb_cur` : `nb_cur` $\leftarrow$ `nb_cur` $+ 1 \quad \Rightarrow 1$

File : fonctionnement avec un tableau (3)



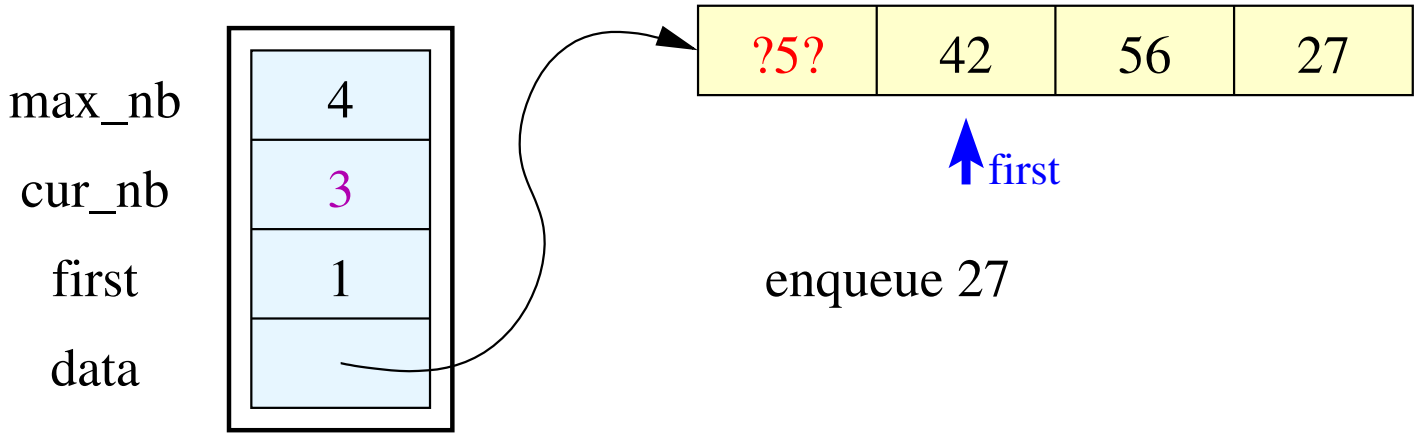
- enqueue 42, enqueue 56 :
  - Vérifier que la file n'est pas pleine.
  - Ajouter au *« premier indice libre »*.
  - Incrémenter `nb_cur` : `nb_cur` ← `nb_cur` + 1 ⇒ 3
- *« Premier indice libre »* : *first* + *cur_nb* . . . tant que ≱ taille max.

File : fonctionnement avec un tableau (4)



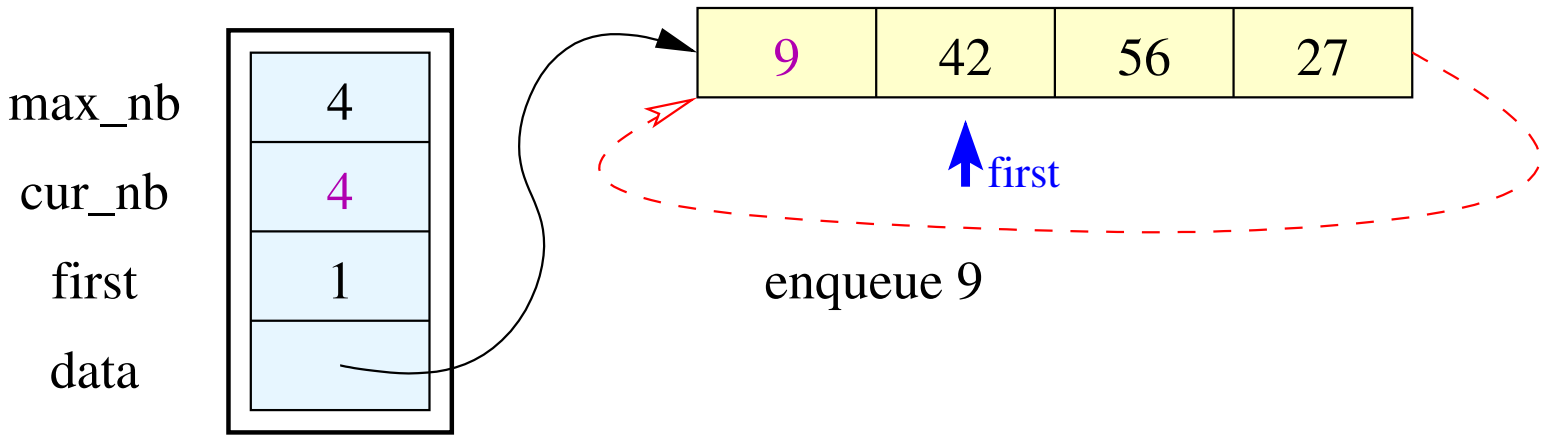
- `take` :
  - Vérifier que la file n'est pas vide.
  - Récupérer la donnée à l'indice `first`.
  - Incrémenter `first` : `first` ← `first` + 1 ⇒ 1 . . . tant que ≱ taille max.
  - Décrémenter `nb_cur` : `nb_cur` ← `nb_cur` - 1 ⇒ 2
- La « *case* » redevient libre mais n'est pas effacée.

File : fonctionnement avec un tableau (4)



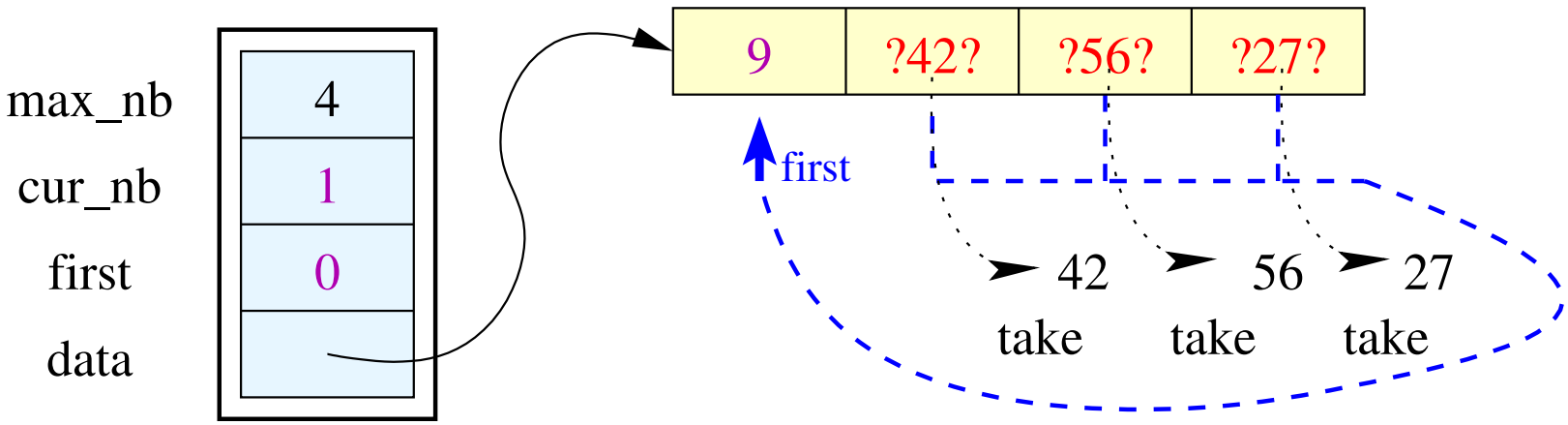
- enqueue 27

File : fonctionnement avec un tableau (5)



- enqueue 9
- Arrivé en bout de tableau la file n'est pas forcément pleine.
- Dépend de s'il y a de la place en début du tableau.
  - ⇒ On fait un cycle et réinsère en début.
  - « *Première case libre* » : $(first + cur\_nb)$ *mod* max_nb

File : fonctionnement avec un tableau (6)



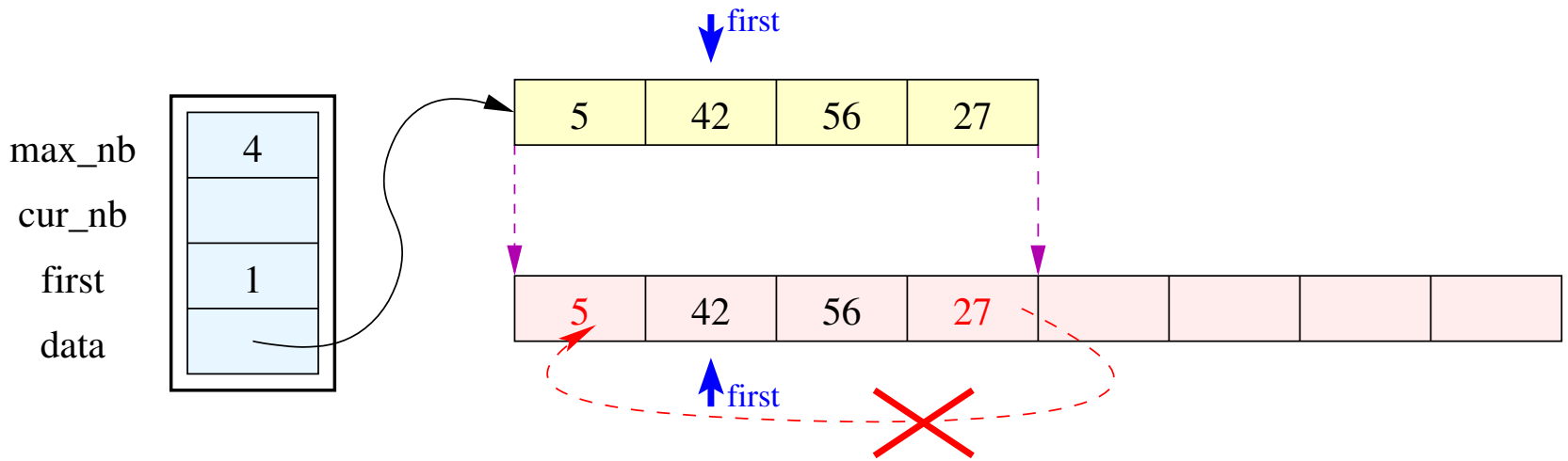
- `take` ; `take` ; `take`
- Arrivé en bout de tableau la file n'est pas forcément vide.
- Dépend de s'il y a des données en début du tableau.
  - ⇒ On fait un cycle et extrait en début.
  - `first` ← (first + 1) mod max_nb
- Situation identique à celle après la 1ère insertion.

File : implantation en Python

```
class EmptyQueue (Exception) : pass
class FullQueue (Exception) : pass

class Queue : ...

  def take (self) :
    if self.cur_nb == 0 : raise (EmptyQueue)
    res = self.storage[self.first]
    self.first = (self.first + 1) % self.max_nb
    self.cur_nb = self.cur_nb - 1
    return res

  def enqueue (self, val) :
    if self.cur_nb == self.max_nb : raise (FullQueue)
    self.storage[(self.first + self.cur_nb) % self.max_nb] = val
    self.cur_nb = self.cur_nb + 1
```

Utiliser l'aspect dynamique pour éviter la file pleine (1)

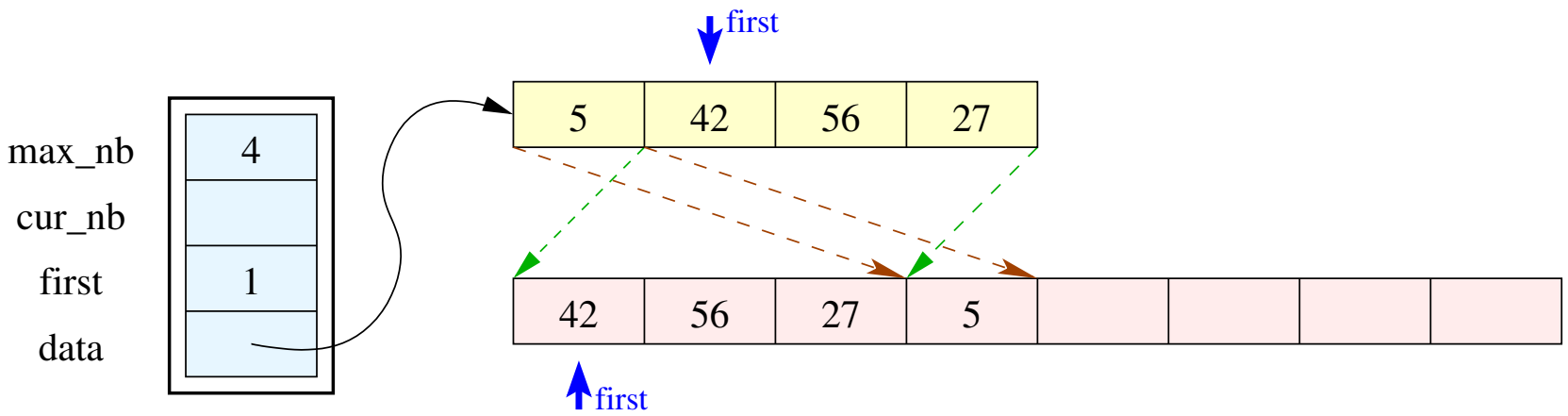
- Traitement à intercaler en cas de file pleine (cur_nb == max_nb).
- Plus compliqué que pour les piles.



- 5 ne suit pas 27 puisque espace libre après 27 !

Utiliser l'aspect dynamique pour éviter la file pleine (2)

- Idée :
  - Recopier depuis *first* → fin du tableau en début de nouveau tableau.
  - Recopier depuis 0 → *first* – 1 à la suite du nouveau tableau.
  - Remettre `first` au début du nouveau tableau (→ 0).



Utiliser l'aspect dynamique pour éviter la file pleine (3)

```
def enqueue (self, val) :
  if self.cur_nb == self.max_nb :
    n_storage = [0] * (self.max_nb * 2)
    d = 0
    for i in range (self.first, self.max_nb) :
      n_storage[d] = self.storage[i]
      d = d + 1
    for i in range (self.first, (self.max_nb - self.first)) :
      n_storage[d] = self.storage[i]
      d = d + 1
    self.storage = n_storage
    self.first = 0
    self.max_nb = self.max_nb * 2
  self.storage[(self.first + self.cur_nb) % self.max_nb] = val
  self.cur_nb = self.cur_nb + 1
```

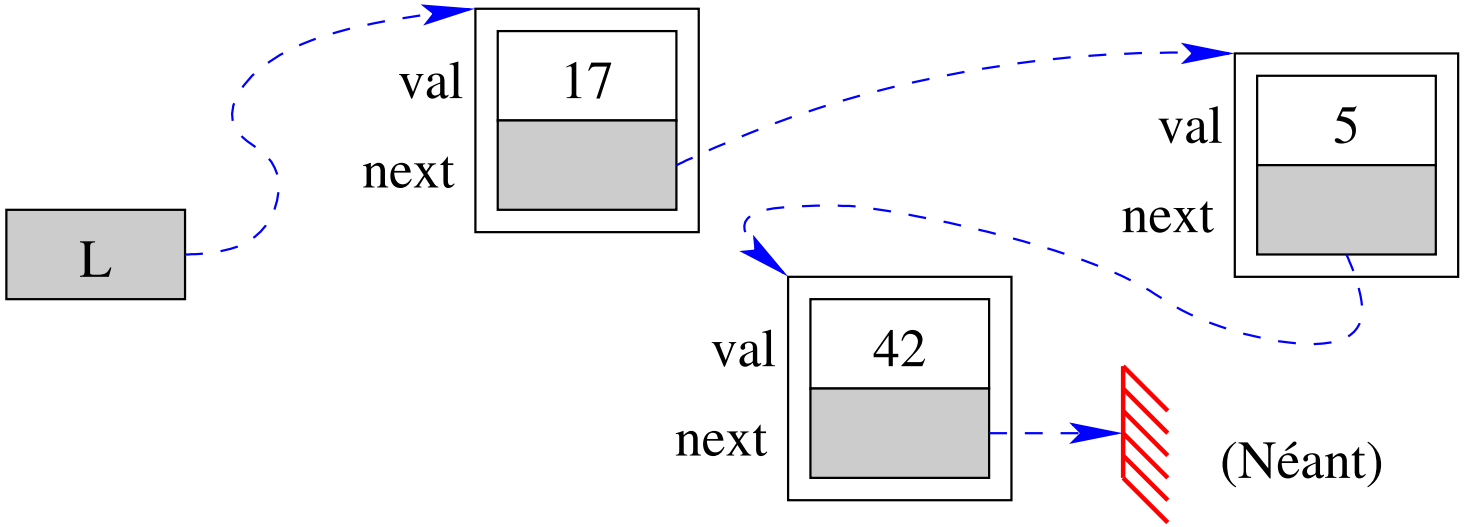
Listes chaînées

Structure de liste

- Une liste c'est :
  - une liste vide
  - ou un élément suivi d'une liste.
- ⇒ Type inductif : list ::= `Nil` | (elem × list)
- En `C`, pas de types inductifs ⇒ encodage à base de pointeurs.
- En Python, pas de pointeurs explicites ⇒ autres encodages.
- Idée de base :
  - On forme une « *chaîne* ».
  - Chaque élément est un maillon de la chaîne.
  - On passe d'un élément au suivant (précédent) en suivant un « pointeur ».
- On ne sait pas efficacement accéder au $i^{\text{ème}}$ élément.
- On « *tient* » une liste par son premier élément.
- Le champ `next` du dernier élément « pointe » vers « une valeur nulle ».

Organisation mémoire

- En mémoire, éléments dispersés ⇒ ≠ tableau où éléments contigus.
- Impossible d'accéder directement au $i^{\text{ème}}$ élément.
- Nécessité de suivre toute la chaîne (parcourir la liste).
- ⇒ Toujours garder accès à la tête de la liste.



Intérêts de listes (1)

- Liste [1 ; 5 ; 6]
  - tête (« *head* ») = 1 de type *elem*.
  - queue (« *tail* ») = [5 ; 6] de type *elem list*.
- Nombreuses opérations en temps constant ($\Theta(1)$).
  - Insérer un élément au début.
  - Supprimer le premier élément (la « *tête* »).
  - Avancer à l'élément suivant.
  - Insérer un élément après un élément dont on connaît l'adresse.
  - Supprimer l'élément suivant un élément dont on connaît l'adresse.

Intérêts de listes (2)

- D'autres opérations pour pas cher, juste 1 ou 2 changements de pointeurs :
  - Concaténer 2 listes (« *append* »).
  - Construire 2 listes partageant la même « *queue* ».
  - Échanger les « *queues* » de 2 listes...
- Structure facilement re-dimensionnable (extension, réduction).
- Accès « direct » plus cher que pour les tableaux...
- Mais dimensionnement dynamique plus facile !

- Langages avec gestion explicite de la mémoire :
  - subtilités (parfois sordides) d'implantation.

Une implantation en Python : structure

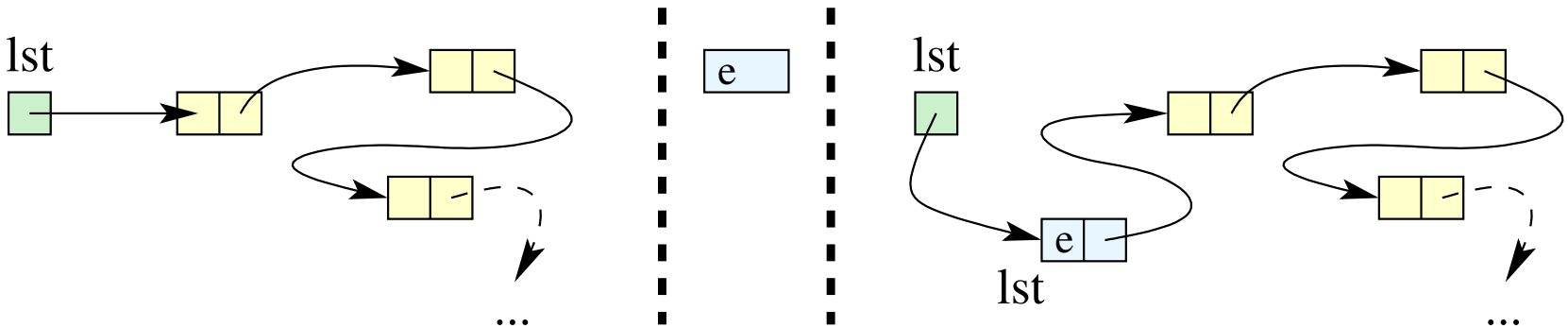
```
class Cell :
  def __init__ (self, val, next) :
    self.val = val          # Valeur mémorisée.
    self.next = next        # Cellule suivante.

# Type des listes simplement chaînées d'entiers.
class List :
  def __init__ (self) :
    self.cells = None       # Cellules de la liste.
```

- Liste : chaîne de cellules.
- Cellules chaînées entre elles.
- Liste vide : chaîne vide de cellules (utilisation de la valeur `None`).

Une implantation en Python : `insert_head`

- `insert_head` : (`elem` × `list`) → `list`
- Choix de modification en place : (`elem` × `list`) → ()

```
class List : ...

  def insert_head (self, val) :
    self.cells = Cell (val, self.cells)
```



Une implantation en Python : impression

- Écriture itérative :

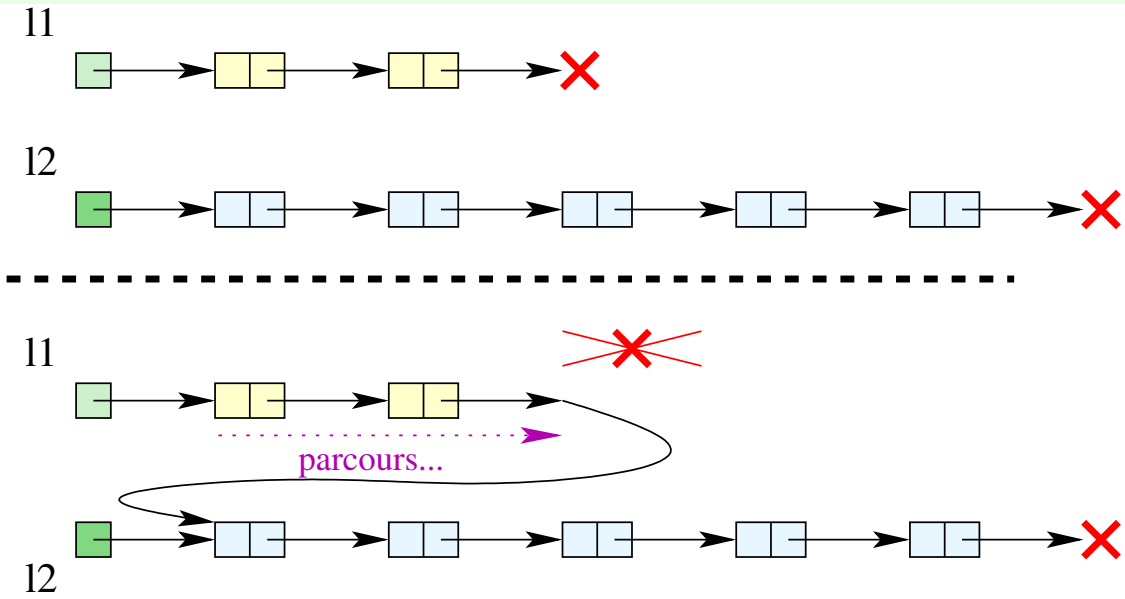
```
def print (self) :
  print ("[", end='')
  tmp = self.cells
  while tmp != None :
    print (" ", tmp.val, sep='', end='') ;
    tmp = tmp.next
  print (" ]")
```

- Écriture récursive (plus coûteuse en pile si compilé naïvement) :

```
def rec_print (self) :
  def print_cells (cell) :
    if cell != None :
      print (" ", cell.val, sep='', end='')
      print_cells (cell.next)
  print ("[", end='')
  print_cells (self.cells)
  print (" ]")
```

Une implantation Python : append

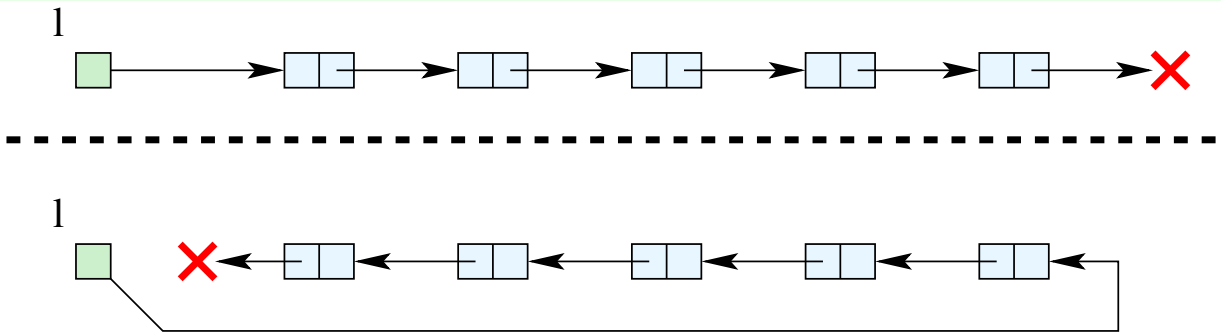
```
def append (self, l2) :
    if self.cells == None : return l2
    tmp = self.cells
    while tmp.next != None : tmp = tmp.next
    tmp.next = l2.cells
    return self
```



Une implantation Python : rev

- Renversement d'une liste en place :

```
def rev (self) :
    curr = self.cells
    prev = None
    while curr != None :
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
    self.cells = prev
```



Une autre implantation en Python : structure

```
empty = None

def insert_head (val, lst) : return (val, lst)

def head (lst) :
  if lst == empty : raise EmptyList
  return lst[0]

def tail (lst) :
  if lst == empty : raise EmptyList
  return lst[1]
```

- Liste : imbrication de couples.
- Liste vide : utilisation de la valeur `None`.
- Modification en place plus possible (car couples non mutables).
- Liste `[ 1 ; 5 ; 6 ]` $\Rightarrow$ (1, (5, (6, None)))

Une autre implantation Python : append

- En exploitant la structure de couple :

```
def append (l1, l2) :
  if l1 == empty : return l2
  return (l1[0], append (l1[1], l2))
```

- Ou (plus joliment) en exploitant les fonctions dédiées :

```
def append_bis (l1, l2) :
  if l1 == empty : return l2
  return insert_head (head (l1), append_bis (tail (l1), l2))
```

- Et l'affichage (en itératif) :

```
def print_list (lst) :
  print ("[", end='')
  while lst != empty :
    print (" ", head (lst), sep='', end='')
    lst = tail (lst)
  print (" ]")
```

Une implantation Python : rev

- Utilisation d'une fonction locale : cache l'accumulateur.

```
def rev (lst) :
  def local_rev (l, accu) :
    if l == empty : return accu
    return local_rev (tail (l), (head (l), accu))
  return local_rev (lst, empty)
```

Variations sur les listes

- Beaucoup de variantes :
  - Doublement chaînées (pointeur `prev`) pour reculer.
  - Listes circulaires : dernier élément pointe sur le premier.
  - Listes circulaires doublement chaînées ...
- La liste peut être plus que le pointeur vers son premier élément :
  - Peut contenir le nombre d'éléments.
  - Peut contenir le pointeur vers le dernier élément.
  - ...

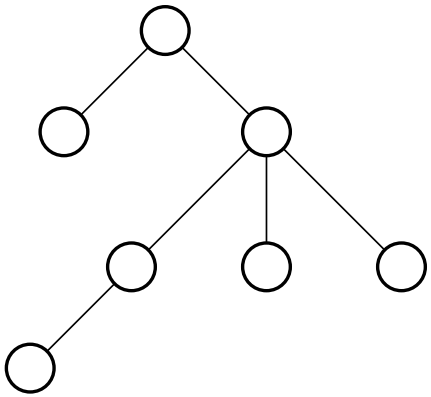
```
class List :
  def __init__ (self) :
    self.nb_elems = 0
    self.cells = None
    ...
```

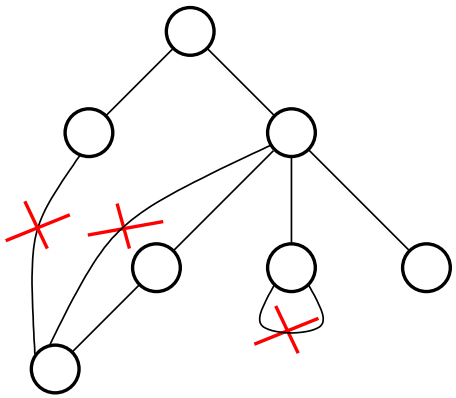
```
class List :
  def __init__ (self) :
    self.head = None
    self.last = None
    ...
```

- En fait, listes = cas «*simple* » d'arbres.
- « Tableaux / listes » natifs de `Python` : pas de vraies listes chaînées.

Les arbres

Arbre ?

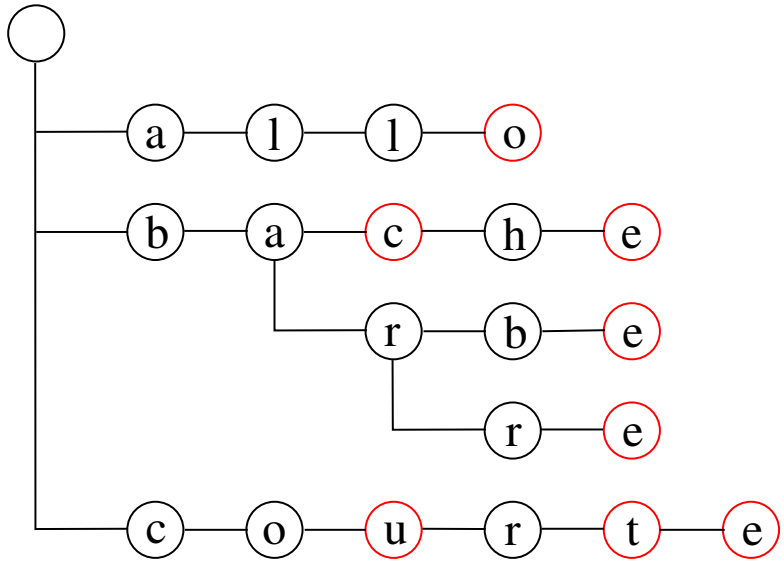




- Structure arborescente.
- Des nœuds, des arcs entre les nœuds.
- Pas de « *boucle* ».
- Pas de « *raccourci* ».

Exemple : le dictionnaire

- Permet le partage de préfixes communs.
- Puisque mots préfixes d'autres, nécessité de marquer les fins de mots.
- Exemple : { allo, bac, bache, barbe, barre, cou, court, courte }.



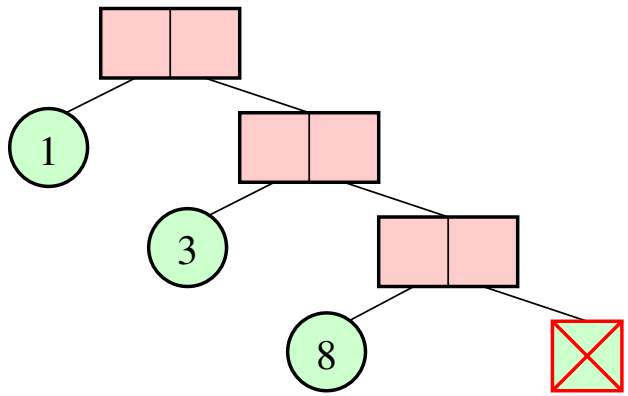
Exemple : le système de fichiers (simple)

```
/
|-- bin
|   |-- alsaunmute
|   |-- arch
|   |-- awk
|   `-- zcat
|-- boot
|   |-- config-2.6.40.6-0.fc15.i686
|   |-- grub
|   |   |-- device.map
|   |   |-- e2fs_stage1_5
|   |   |-- ufs2_stage1_5
|   |   |-- vstafs_stage1_5
|   |   `-- xfs_stage1_5
|   |-- initramfs-2.6.40.6-0.fc15.i686.img
|   `-- vmlinuz-2.6.43.2-6.fc15.i686
|-- cgroup
|-- dev
|   |-- autofs
```

- Racine : /
- Noeuds internes : répertoires.
- Feuilles : fichiers.
- (On ne considère pas les « raccourcis » – ou « liens »).

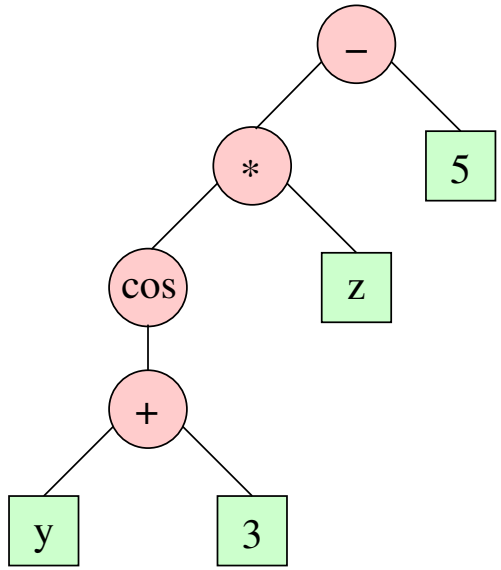
Exemple : les listes

- [1;3;8]
- Type inductif :
  - Liste vide.
  - Élément suivi d'une liste.
- Longueur de la liste : hauteur de l'arbre.



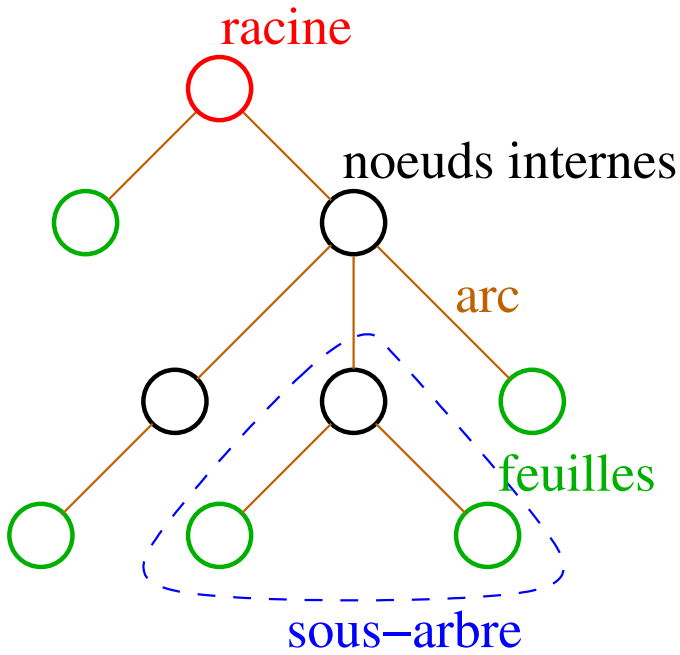
Exemple : les arbres de syntaxe

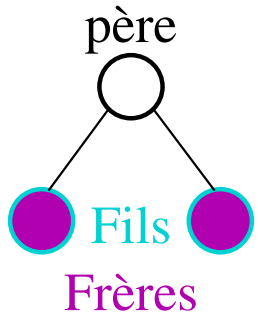
- Expressions arithmétiques :
  - Constantes.
  - Opérateurs binaires entre expressions arithmétiques.
  - Appel de fonction avec des expressions arithmétiques en arguments.
  - Variables.
- Exemple : $cos(y+3) \times z - 5$
- Permet de calculer sur les expressions :
  - évaluation,
  - dérivation,
  - simplification,
  - compilation...



Vocabulaire (1)

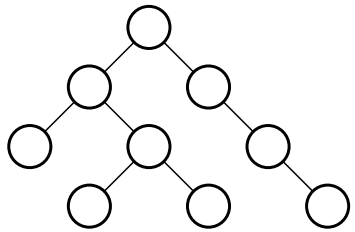
- Nœud (« *node* ») :
  - Racine (« *root* »)
  - Nœud interne
  - Feuille (« *leaf* »)
- Arc (« *edge* »)
- Sous-arbre (« *subtree* »)
- Parenté :
  - Père (« *parent* »)
  - Fils (« *child* »)
  - Frère (« *sibling* »)



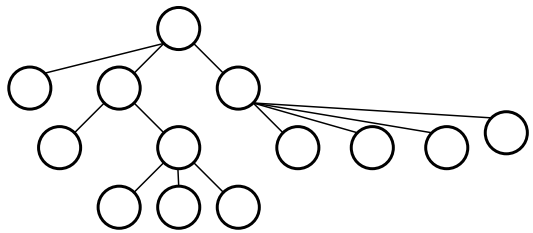


Vocabulaire (2)

- Degré d'un nœud : nombre de fils.
- Arité d'un arbre : nombre max de fils.
  - Certains nœuds peuvent en avoir moins.
- Arbre binaire : arité 2.

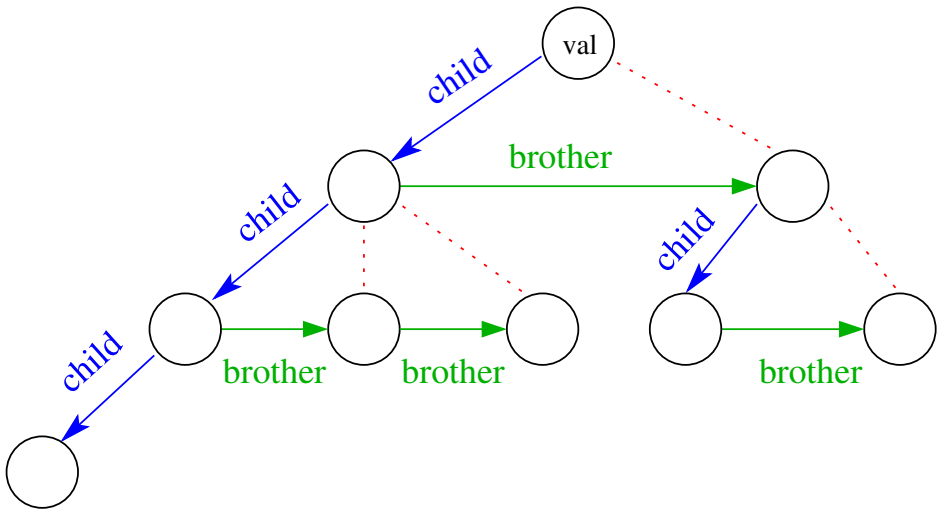


- Sinon, arbre n-aire.



Implantation par « pointeurs » : *Left-Child Right-Sibling*

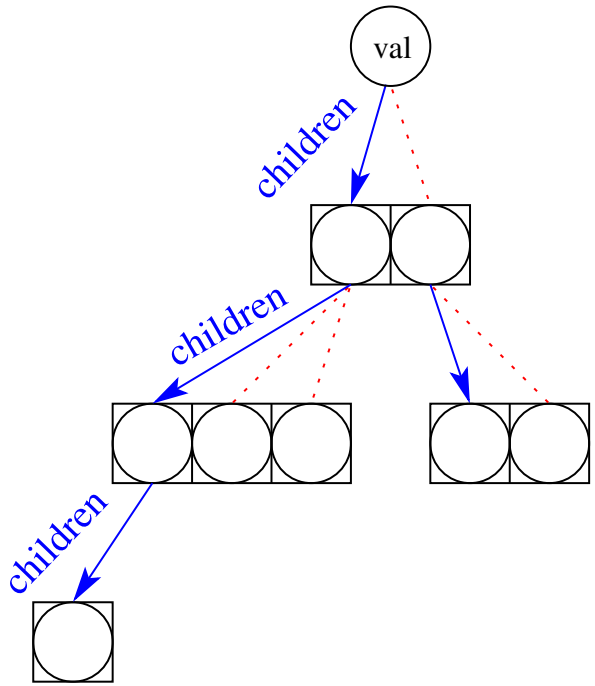
- Cas général : nombre quelconque de fils :
  - Liste chaînée de tous les fils (frères).
  - Le parent a un « pointeur » sur son 1er fils.




```
class Node :
  def __init__ (self) :
    self.val = ""
    self.child = None
    self.sibbling = None

def add_node (val, children) :
  n = Node ()
  n.val = val
  nb_children = len (children)
  if nb_children == 0 :
    n.child = None
  else :
    n.child = children[0]
    i = 1
    while i < nb_children :
      children[i - 1].sibbling =
            children[i]
      i = i + 1
  return n
```

Implantation par « pointeurs » : tableau de fils

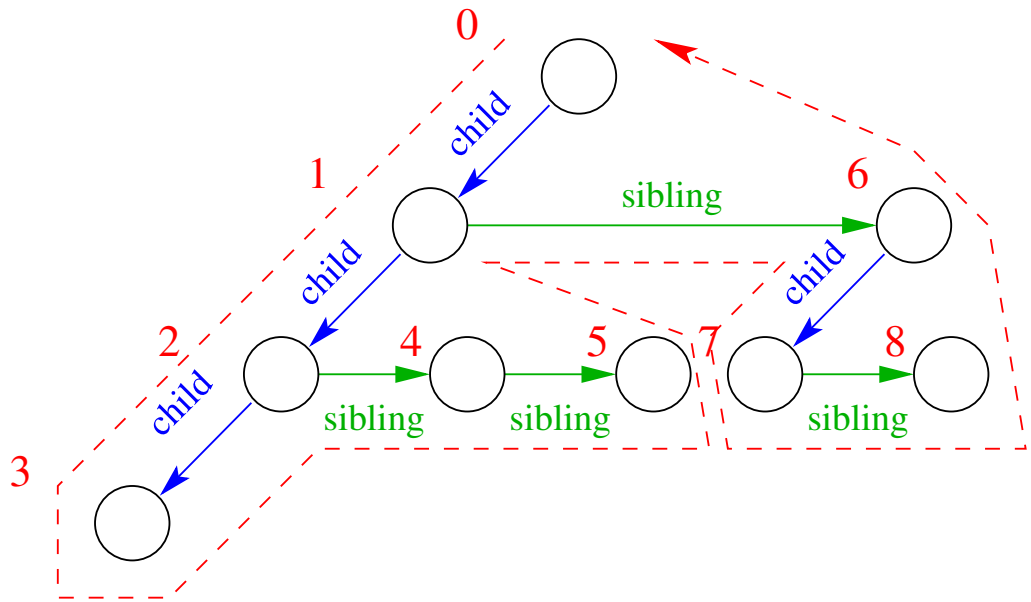
- Cas général : nombre quelconque de fils :
  - Le parent a un tableau de tous ses fils.




```
class Node :
  def __init__ (self) :
    self.val = ""
    self.children = []

def add_node (val, children) :
  n = Node ()
  n.val = val
  n.children = children
  return n
```

Parcours en profondeur

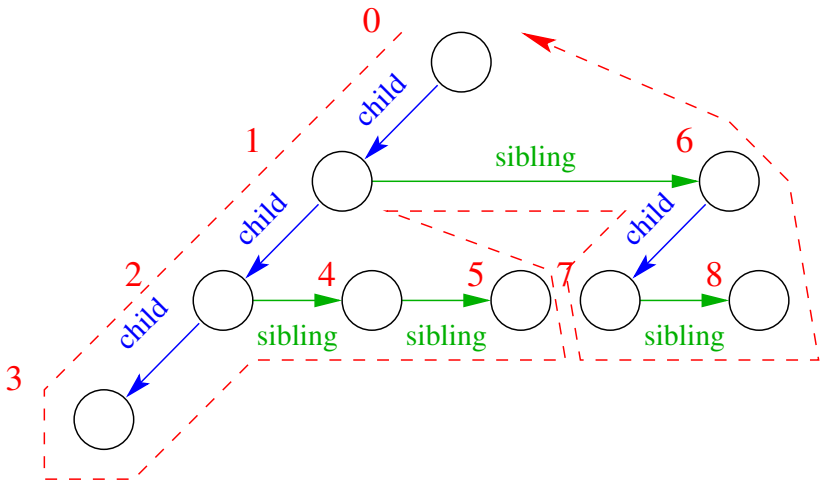
- En Anglais : « *Depth First Search* » (DFS).
- Descente au plus profond en commençant par le fils gauche (ou droit).
- Parcours récursif (exemple parcours gauche-droite) :
  - À chaque nœud, on applique la descente sur le fils le plus à gauche.
  - Une fois le sous-arbre gauche exploré, on explore les fils à droite.
  - Une fois les sous-arbres droites explorés, on remonte et applique le parcours.
  - Fin : fin de l'exploration du sous-arbre le plus à droite.

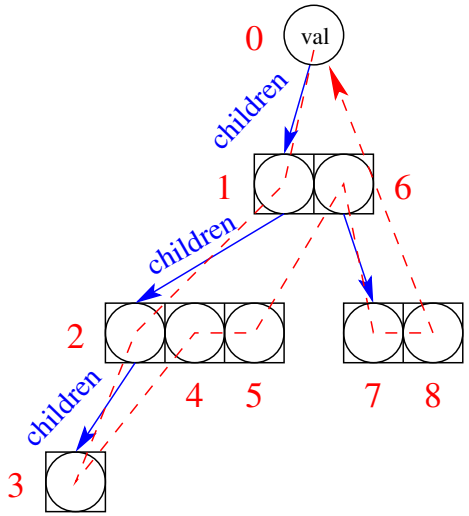


Parcours en profondeur : implantation en Python

```
class Node : ...

  def dfs (self) :
    print (self.val)
    if self.child != None :
      self.child.dfs ()
    s = self.sibbling
    while s != None :
      s.dfs ()
      s = s.sibbling
```

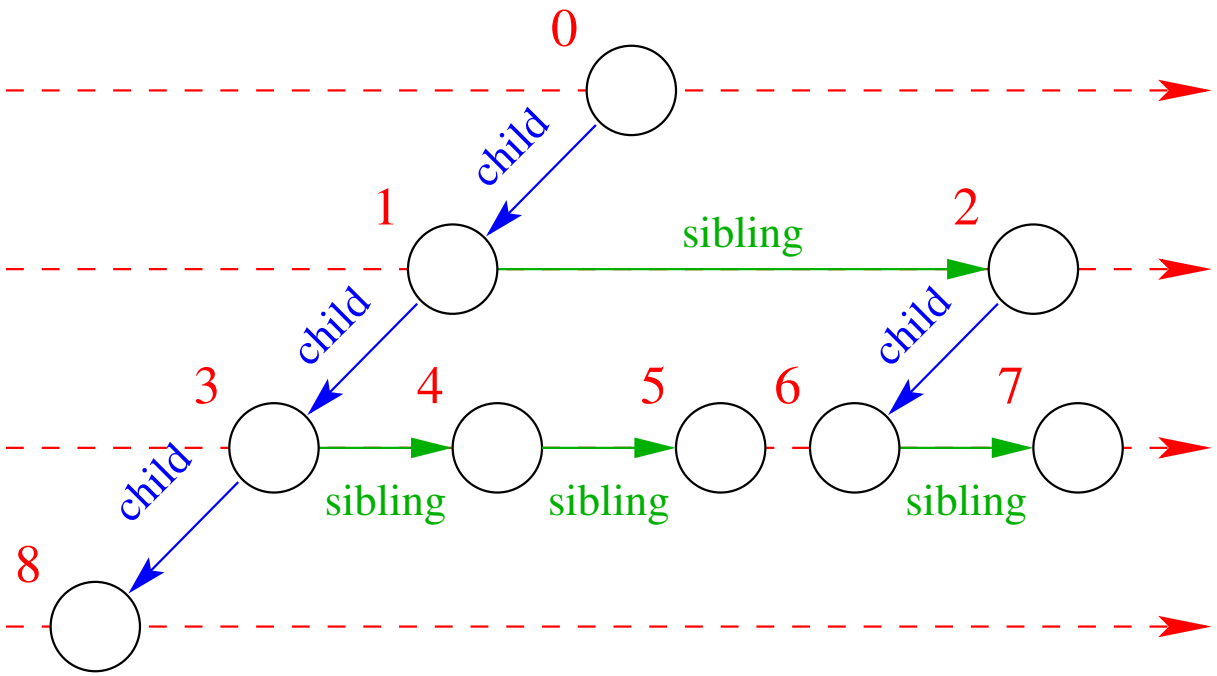
```
class Node : ...

  def dfs (self) :
    print (self.val)
    for c in self.children :
      c.dfs ()
```





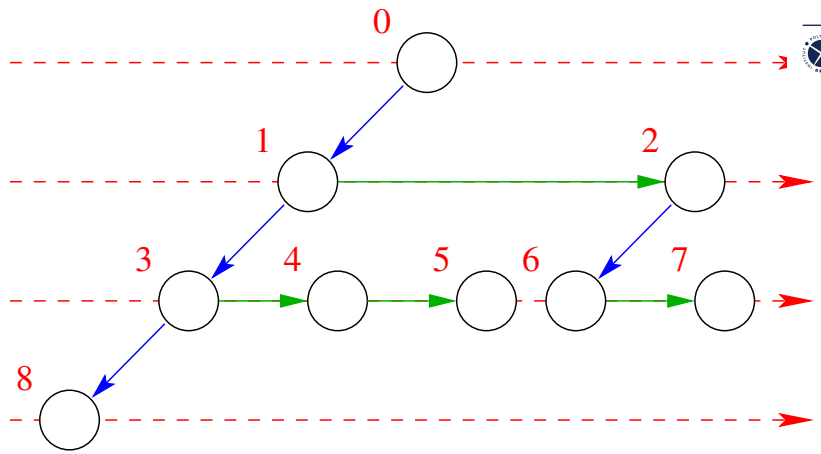
Parcours en largeur

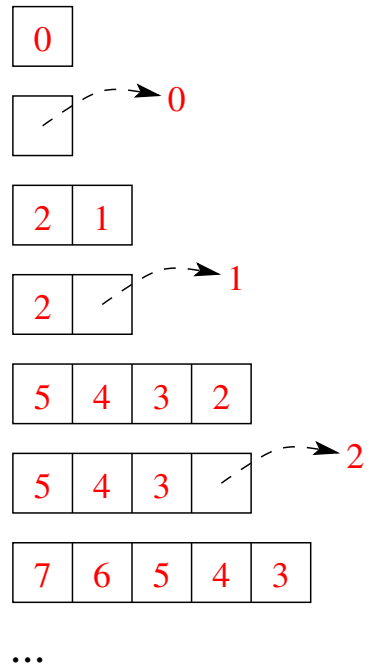
- En Anglais : « *Breadth First Search* » (BFS).
- Parcours « *par niveaux* » (croissants).
- On visite d'abord tous les nœuds de même profondeur.
- Utilisation d'une file :
  - On extrait un nœud, on le traite, on insère tous ses fils.



Parcours en largeur : implantation en Python

```
class Queue : ...
  def take (self) : ...
  def enqueue (self, val) : ...
  def is_empty (self) : ...

class Node : ...
  def bfs (self) :
    q = Queue ()
    q.enqueue (self)
    while not (q.is_empty ()) :
      curr_n = q.take ()
      print (curr_n.val)
      s = curr_n.sibling
      while s != None :
        q.enqueue (s)
        s = s.sibling
```

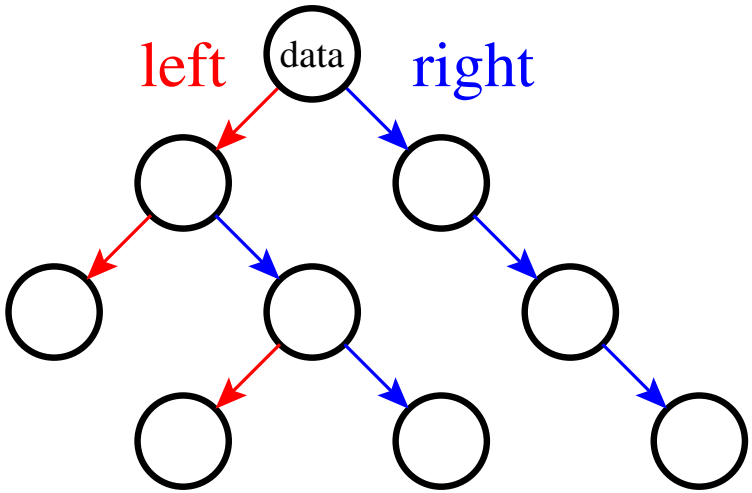




Arbres binaires

Arbres binaires

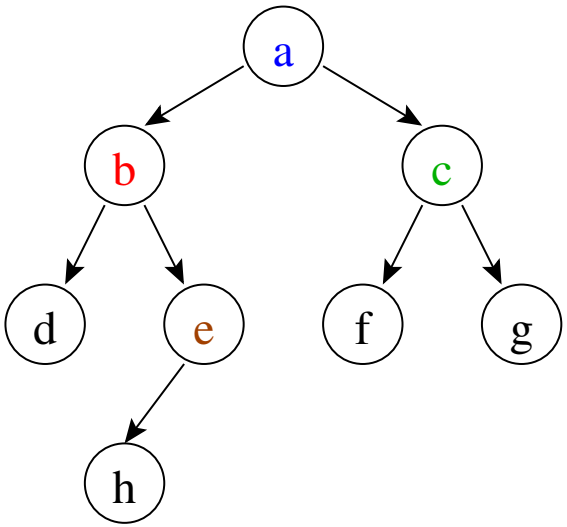
- Un nœud a au plus 2 fils :
  - 1 fils «*droit* »
  - 1 fils «*gauche* »
- Simples mais nombreuses applications :
  - Codage de Huffman.
  - Calcul de la complexité optimale du tri.
  - Recherche sur un ensemble ordonné.
  - Satisfiabilité de formules booléennes (Binary Decision Diagram) . . .

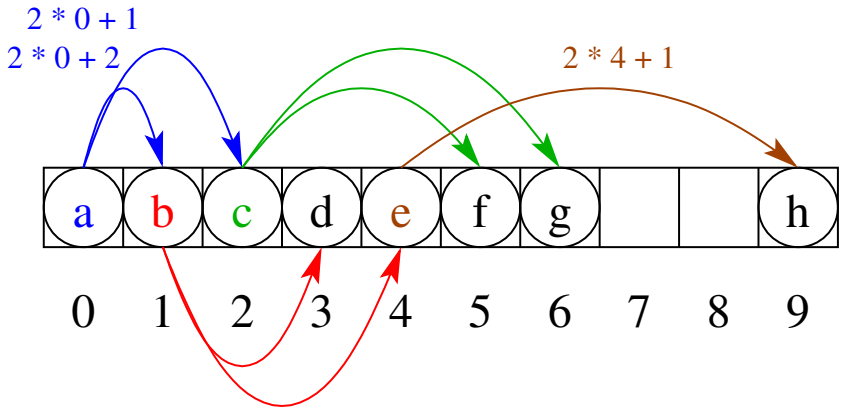



```
class Node :
  def __init__ (self, val, left,
      right) :
    self.val = val
    self.left = left
    self.right = right
```

Implémentation par tableau

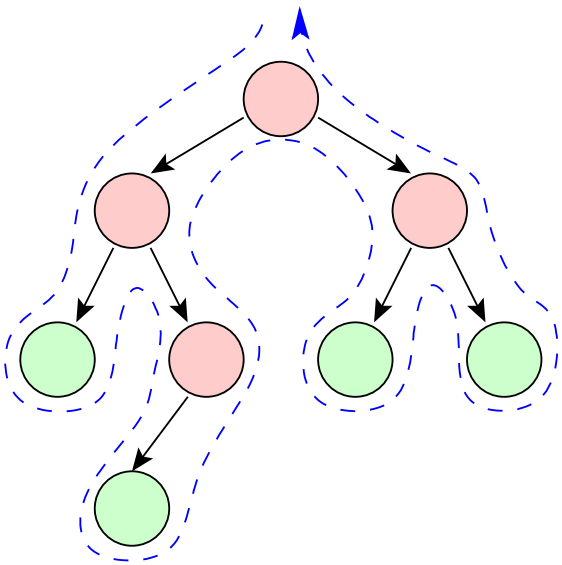
- Un nœud a un indice i.
- Ses fils sont aux indices $2i+1$ et $2i+2$.
- Son père est à l'indice $\lfloor((i-1)/2)\rfloor$.
- ⇒ Pas besoin de pointeurs.
- ⇒ Si arbre binaire complet, pas de perte de place.





Arbres binaires : parcours en profondeur

- Descente au plus profond en commençant par le fils gauche (ou droit).



```
class Node : ...
  def dfs (self) :
    if self.left != None :
      self.left.dfs ()
    if self.right != None :
      self.right.dfs ()
```

Ou comme une fonction et non une méthode :

```
def dfs (n) :
  if n != None :
    dfs (n.left)
    dfs (n.right)
```

- Ici, on a fait que parcourir, aucun traitement...

Arbres binaires : parcours en profondeur préfixe

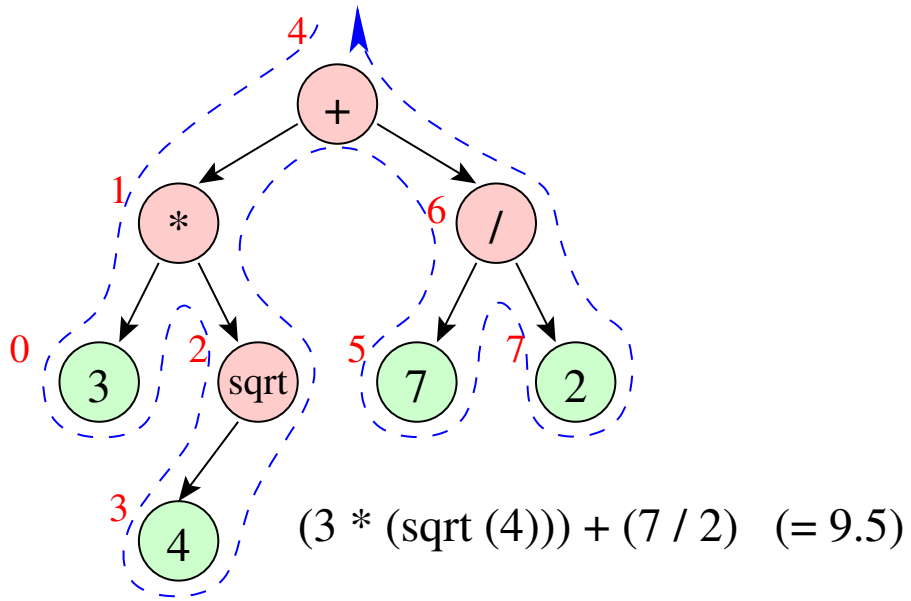
- 3 endroits pour insérer du traitement du nœud courant.
  - Au début.

```
def prefix_traverse (n) :
  if n != None :
    do_something (n.val)
    prefix_traverse (n.left)
    prefix_traverse (n.right)
```

Arbres binaires : parcours en profondeur infixe

- 3 endroits pour insérer du traitement du nœud courant.
  - Au milieu.
  - Si `do_something` est l'impression : correspond à l'écriture *« normale »* d'une expression arithmétique.

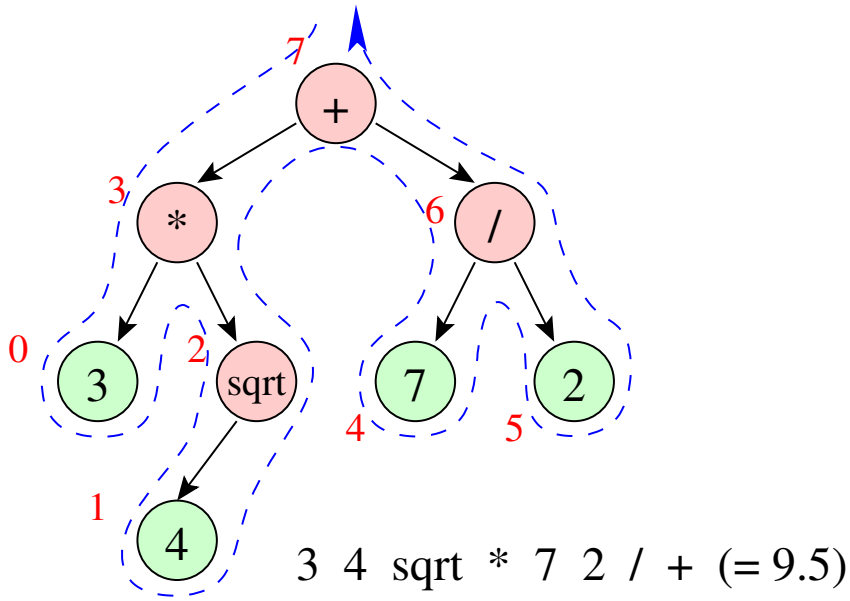
```
def infix_traverse (n) :
  if n != None :
    infix_traverse (n.left)
    do_something (n.val)
    infix_traverse (n.right)
```



Arbres binaires : parcours en profondeur postfixe

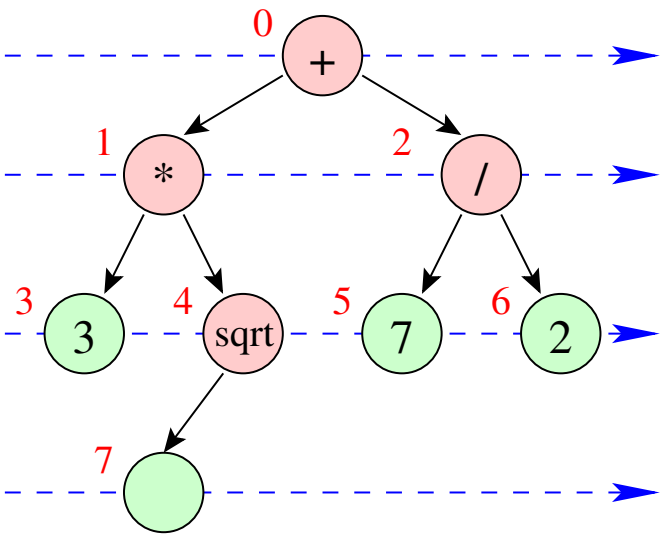
- 3 endroits pour insérer du traitement du nœud courant.
  - À la fin.
  - Si `do_something` est l'impression : correspond à l'écriture RPN (polonaise inverse).

```
def postfix_traverse (n) :
  if n != None :
    postfix_traverse (n.left)
    postfix_traverse (n.right)
    do_something (n.val)
```



Arbres binaires : parcours en largeur

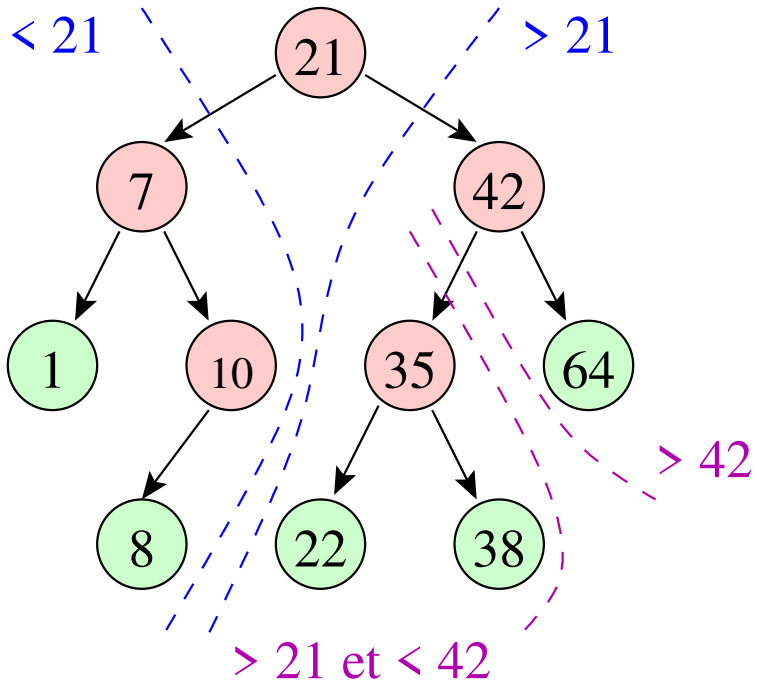
- On visite d'abord tous les nœuds de même profondeur.
- Utilisation d'une file :
  - On extrait un nœud, on le traite, on insère ses 2 fils.




```
class Queue : ...
  def take (self) : ...
  def enqueue (self, val) : ...
  def is_empty (self) : ...

class Node : ...
  def bfs (self) :
    q = Queue ()
    q.enqueue (self)
    while not (q.is_empty ()) :
      curr_n = q.take ()
      do_something (curr_n.val)
      if curr_n.left != None :
        enqueue (curr_n.left)
      if curr_n.right != None :
        enqueue (curr_n.right)
```

Arbres binaires de recherche

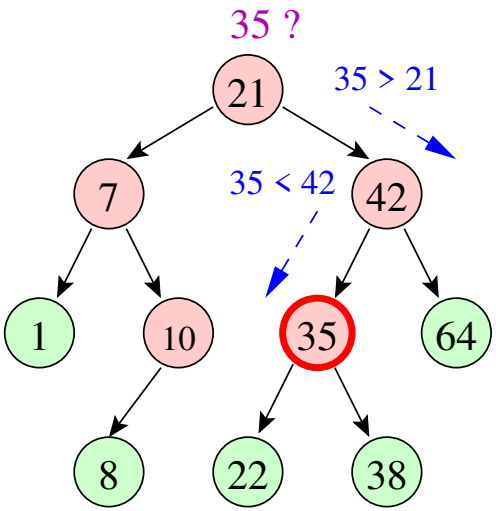
- Structure d'arbre binaire telle que :
  - Chaque nœud contient une clef.
  - La clef d'un nœud est $>$ à celle de son fils gauche.
    - Inductivement $>$ à celles du sous-arbre gauche.
  - La clef d'un nœud est $<$ à celle de son fils droit.
    - Inductivement $<$ à celles du sous-arbre droit.

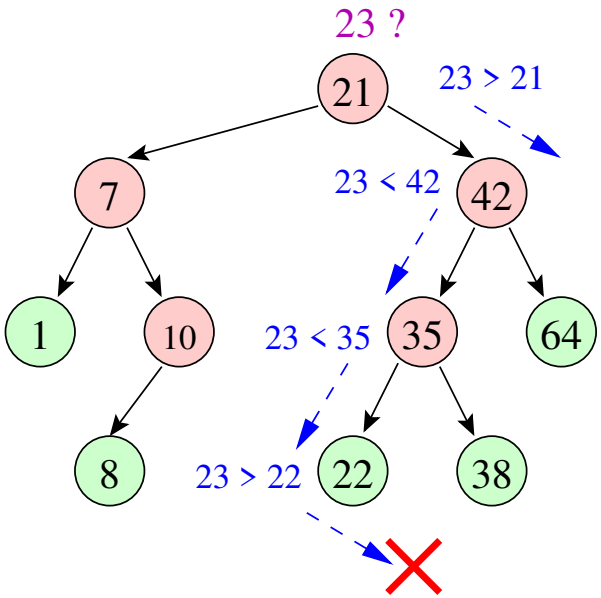


Arbres binaires de recherche : intérêts

- Stockage en $\theta(n)$.
- Recherche en $\theta(h)$ (h hauteur de l'arbre).
- Insertion en $\theta(h)$.
- Suppression en $\theta(h)$.
- En moyenne $h = \theta(log(n))$: $\rightarrow$ bonne solution au problème de recherche en table.
- Dans le pire cas $h = n$: $\rightarrow$ intérêt d'avoir des arbres équilibrés.
- Par rapport aux tables de hachage :
  - Pas besoin de connaître n à l'avance.
  - Pas de mémoire gaspillée.
  - Les ABR équilibrés sont une solution efficace dans le pire cas.

Arbres binaires de recherche : recherche

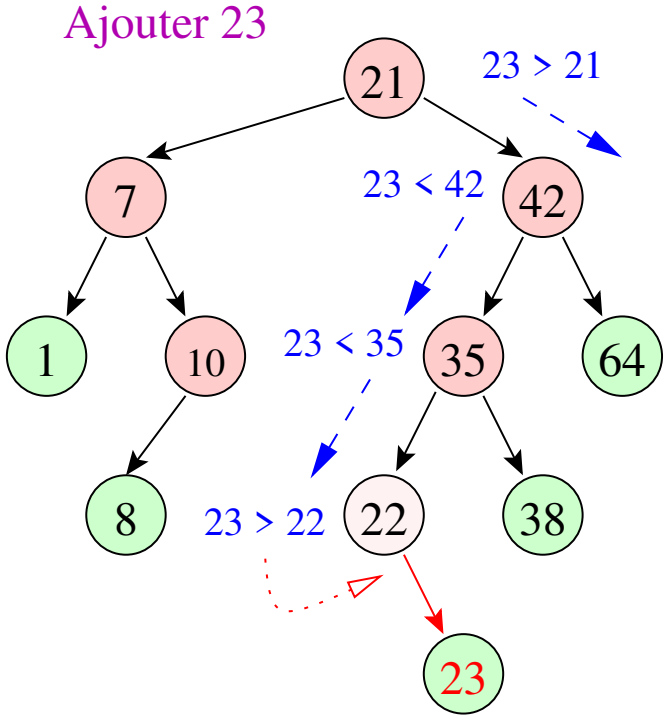





```
def find (n, val) :
  if n == None : return False
  if n.val == val : return True
  if val < n.val : return find (n.
      left, val)
  return find (n.right, val)


def find_iter (n, val) :
  while n != None :
    if n.val == val : return True
    if (val < n.val) : n = n.left
    else : n = n.right
  return False
```

Arbres binaires de recherche : insertion (version naïve)

- Rechercher la position du nœud dans l'arbre.
- L'ajouter comme fil du dernier nœud rencontré.



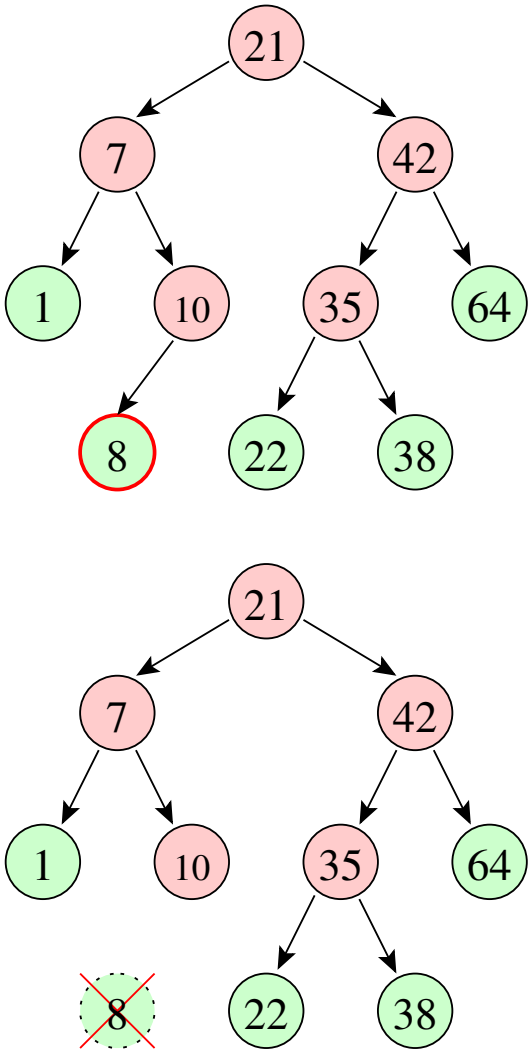

```
def insert (n, val) :
  if n == None :
    nn = Node (val, None, None)
    return nn
  if val < n.val :
    n.left = insert (n.left, val)
  elif val > n.val :
    n.right = insert (n.right, val)
  return n
```

Arbres binaires de recherche : suppression

- On commence par rechercher le nœud à supprimer dans l'arbre.
- 3 cas dont 2 simples :
  - Suppression d'une feuille (→ facile).
  - Suppression d'un nœud avec 1 seul fils (→ facile).
  - Suppression d'un nœud avec 2 fils (→ plus compliqué).

```
def remove_node (n, val) :
  if n == None : return None
  if n.val == val : return __remove_root (n)
  if (val < n.val) : n.left = remove_node (n.left, val)
  else : n.right = remove_node (n.right, val)
  return n
```

ABR, suppression : 1 feuille

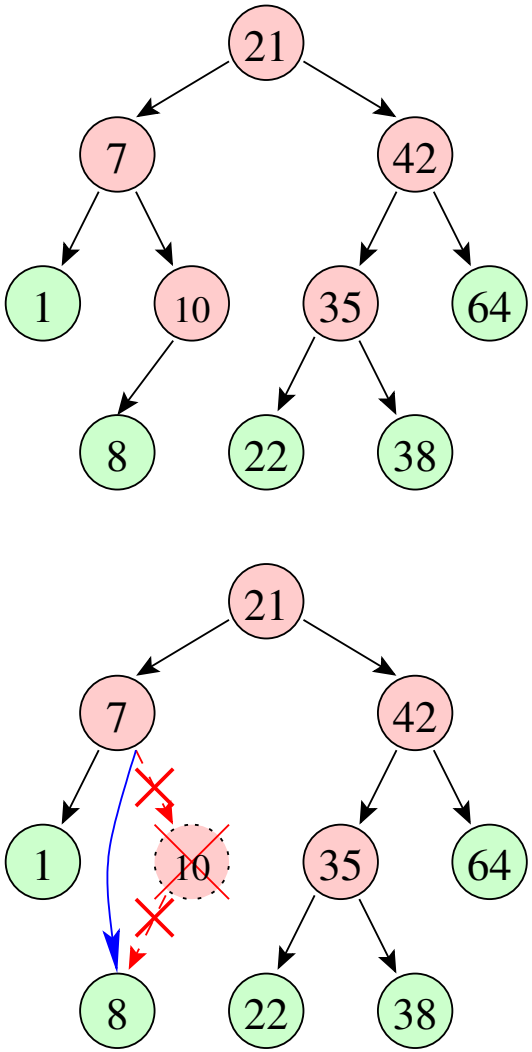


Suppression du nœud 8 . . .

- Mettre le fils droit du nœud 10 à `None`.

```
def __remove_root(n):
    if n.left == None : return n.right
    if n.right == None : return n.left
    child = __max_node_left_subtree(n.left)
    n.val = child.val
    n.left = remove_node(n.left, child.val)
    return n
```

ABR, suppression : 1 seul fils

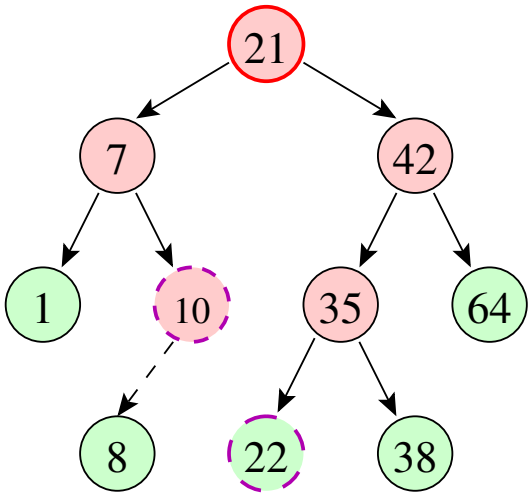


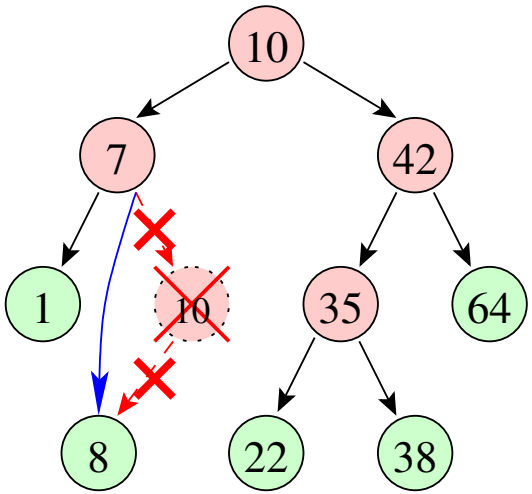
Suppression du nœud 10 . . .

- Faire pointer son père (7) vers son fils (8).

```
def __remove_root(n):
    if n.left == None : return n.right
    if n.right == None : return n.left
    child = __max_node_left_subtree(n.left)
    n.val = child.val
    n.left = remove_node(n.left, child.val)
    return n
```

ABR, suppression : 2 fils





Suppression du nœud 21 . . .

- Remplacement par le min du sous-arbre droit ou le max du sous-arbre gauche.
  - Successeur ou prédécesseur le plus proche.
- Choisir un candidat.
- On recopie sa clef et ses données pour remplacer le nœud 21.
- Ensuite, supprimer le nœud 10
  - 10 étant « le max », il n'a pas de fils droit,
  - ⇒ tombe dans les 2 cas précédents.

```
def __remove_root(n):
    if n.left == None: return n.right
    if n.right == None: return n.left
    child = __max_node_left_subtree(n.left)
    n.val = child.val
    n.left = remove_node(n.left, child.val)
    return n
```

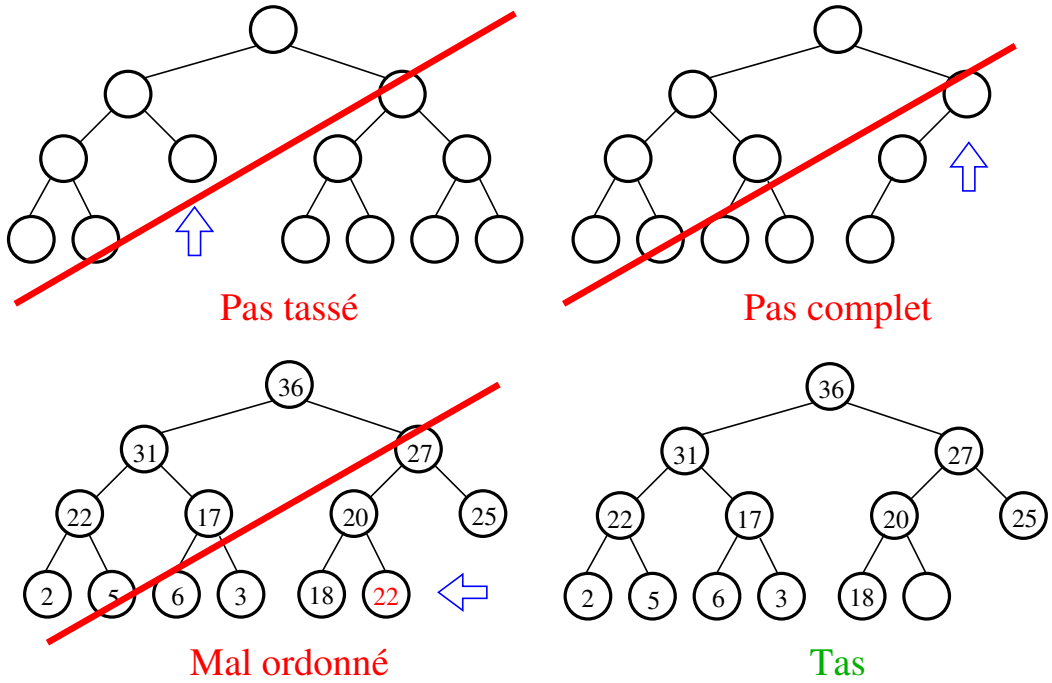
Pour la culture . . . d'arbres. . .

- Nombreuses autres formes d'arbres :
  - Enracinés ou libres (avec ou sans racine).
  - AVL trees.
  - Red-Black trees.
  - Patricia trees (radix trees).
  - Splay trees.
  - BSP trees
  - Quadtrees, octrees.
  - Arbres de syntaxe abstraite.
  - Etc. . .

Les files de priorité

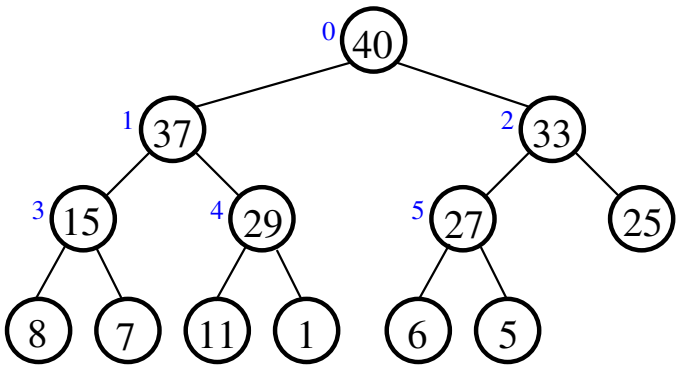
Arbres binaires et tas

- Arbre binaire « tassé » si :
  - tous les niveaux (sauf éventuellement le dernier) sont complets
  - et le dernier niveau est rempli à gauche.
- Tas : arbre binaire « tassé »
  - dont les nœuds sont étiquetés par des clefs,
  - tout noeud possède une clef supérieure ou égale aux clefs de ses fils (i.e. des nœuds de ses sous-arbres gauche et droit)



Représentation d'un tas

- Tableau : comme vu pour les arbres binaires.
- Fils gauche et droit aux indices $2i+1$ et $2i+2$.
- Père à l'indice $\lfloor((i-1)/2)\rfloor$.
- Tableau sans « trous ».

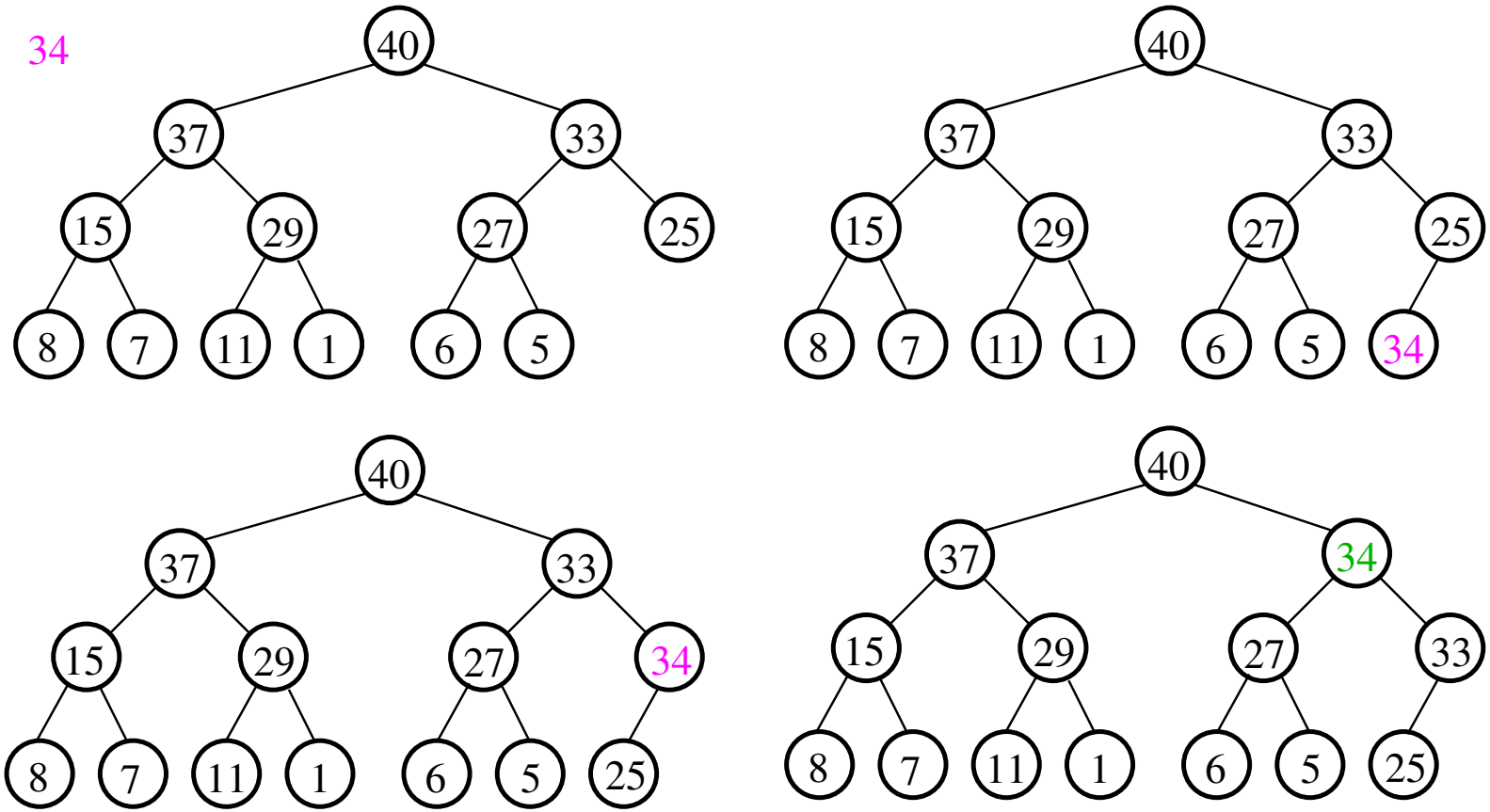


40	37	33	15	29	27	25	8	7	11	1	6	5		
0	1	2	3	4	5									14

```
class Prio :
  def __init__ (self) :
    self.max_size = 64
    self.curr_size = 0
    self.storage = [0] *
        self.max_size
```

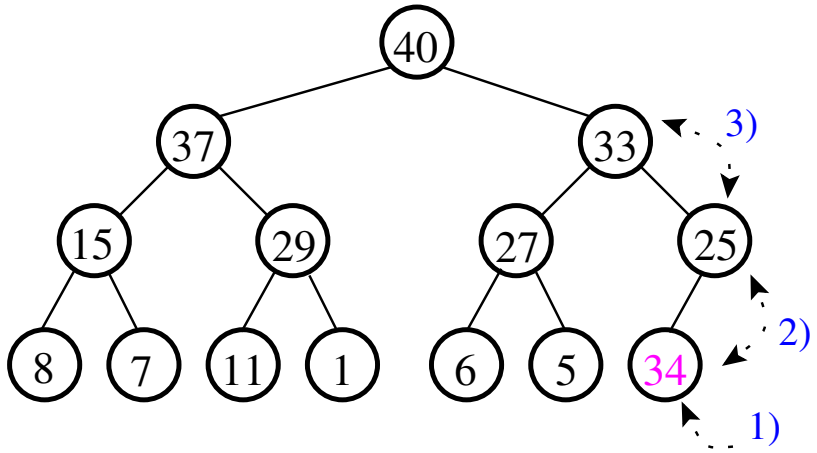
Insertion : première solution (1)

- Placer le noeud à la dernière position libre du tableau.
- Tant que parent plus petit, remonter en permutant avec le parent.



Insertion : première solution (2)

- Placer le noeud à la dernière position libre du tableau.
- Tant que parent plus petit, remonter en permutant avec le parent.

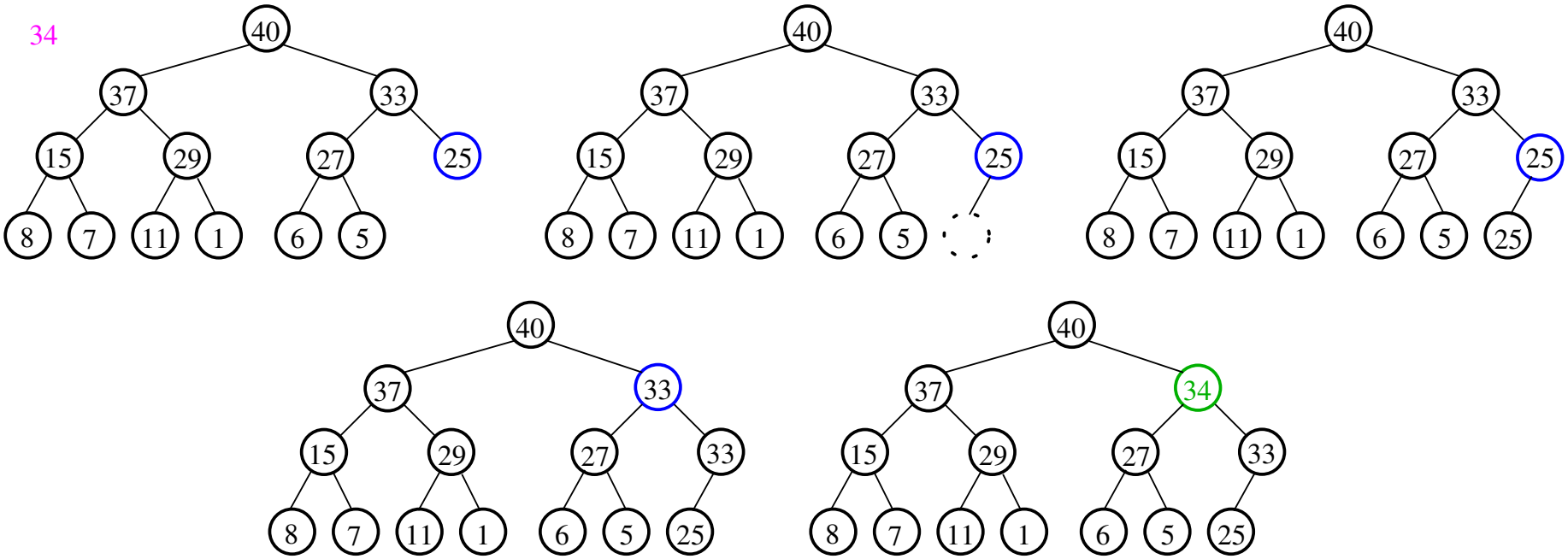



```
class Prio : ...

  def insert (self, val) :
    if self.curr_size == self.max_size : raise (FullQueue)
    storage = self.storage
    storage[self.curr_size] = val
    i = self.curr_size
    while i > 0 and storage[parent_index (i)] <= storage[i] :
      tmp = storage[i]
      storage[i] = storage[parent_index (i)]
      storage[parent_index (i)] = tmp
      i = parent_index (i)
    self.curr_size = self.curr_size + 1
```

Insertion : autre solution (1)

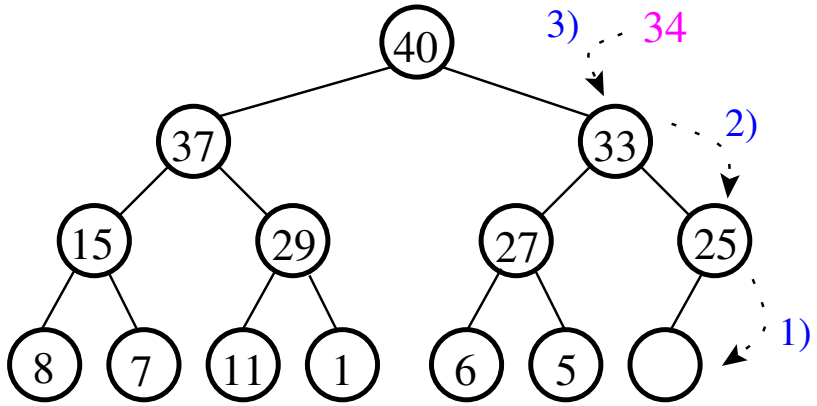
- Partir de la dernière case occupée du tableau.
- En remontant, faire redescendre les parents ≤ la priorité à insérer.
- Quand on trouve un parent >, insérer la valeur au nœud courant.

34



Insertion : autre solution (2)

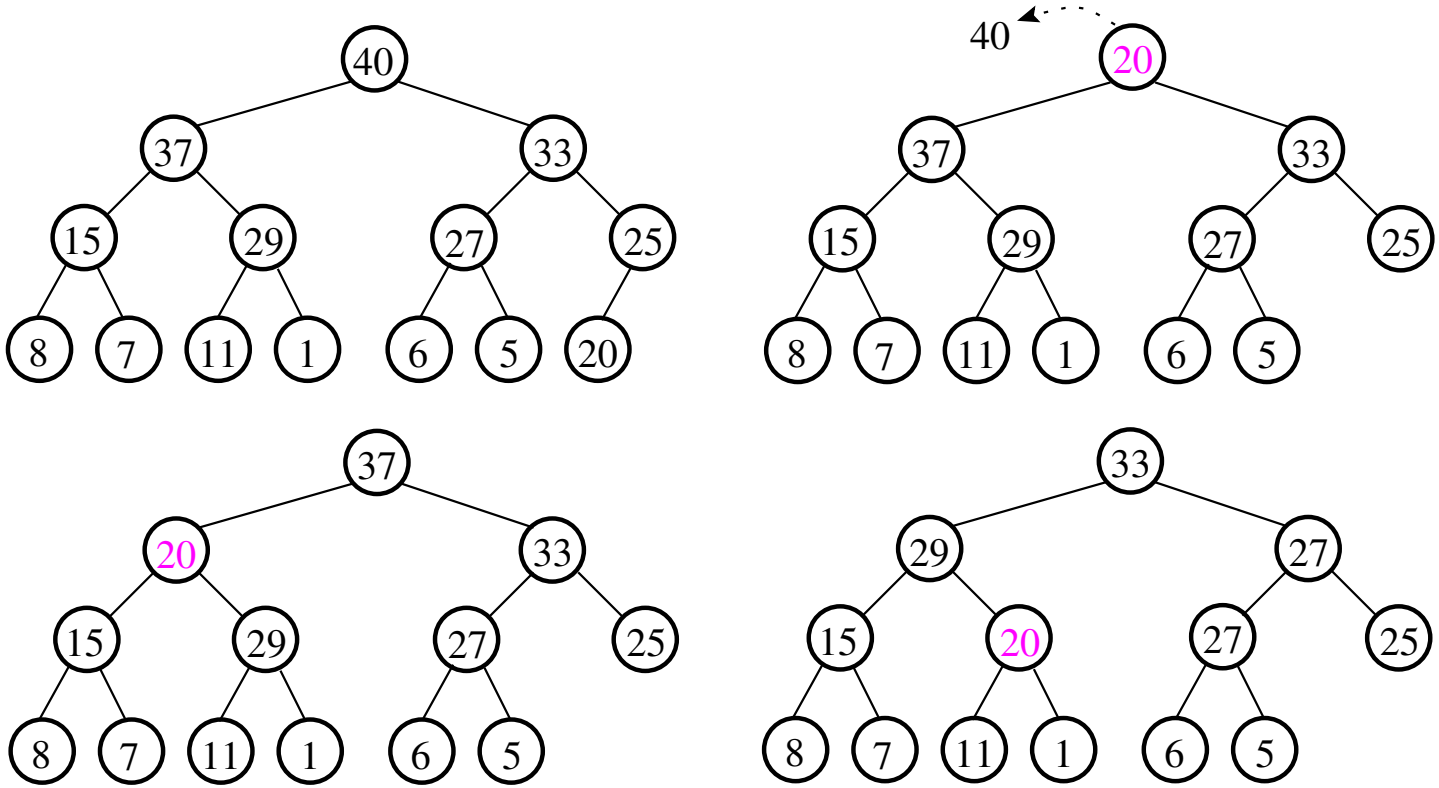
- Partir de la dernière case occupée du tableau.
- En remontant, faire redescendre les parents ≤ la priorité à insérer.
- Quand on trouve un parent >, insérer la valeur au nœud courant.




```
class Prio : ...

  def insert (self, val) :
    if self.curr_size == self.max_size : raise (FullQueue)
    i = self.curr_size
    self.curr_size = self.curr_size + 1
    storage = self.storage
    while i > 0 and storage[parent_index (i)] <= val :
      storage[i] = storage[parent_index (i)]
      i = parent_index (i)
    storage[i] = val
```

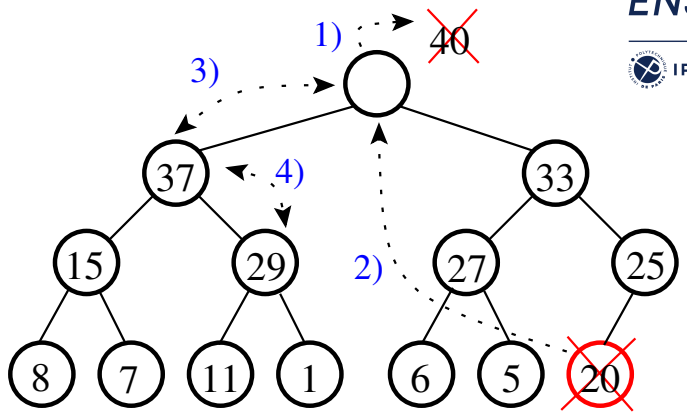
Extraction (2)

- Copier le dernier nœud dans la racine.
- Faire redescendre la valeur de la racine en permutant avec le plus grand des fils.
- Arrêter quand le plus grand des fils est ≤ priorité insérée.



Extraction (2)

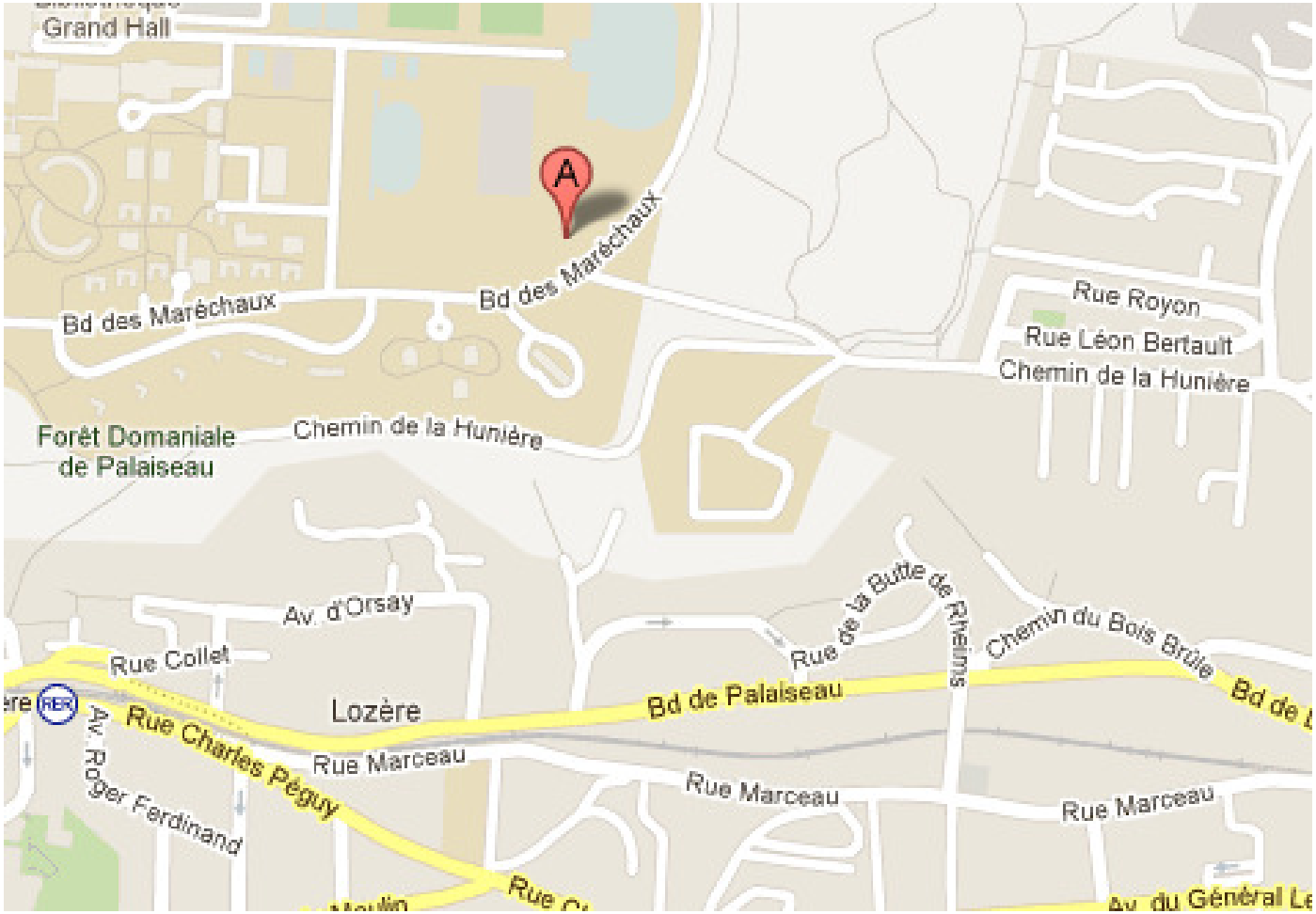
- Copier le dernier nœud dans la racine.
- Faire redescendre la valeur de la racine en permutant avec le plus grand des fils.
- Arrêter quand le plus grand des fils est ≤ priorité insérée.



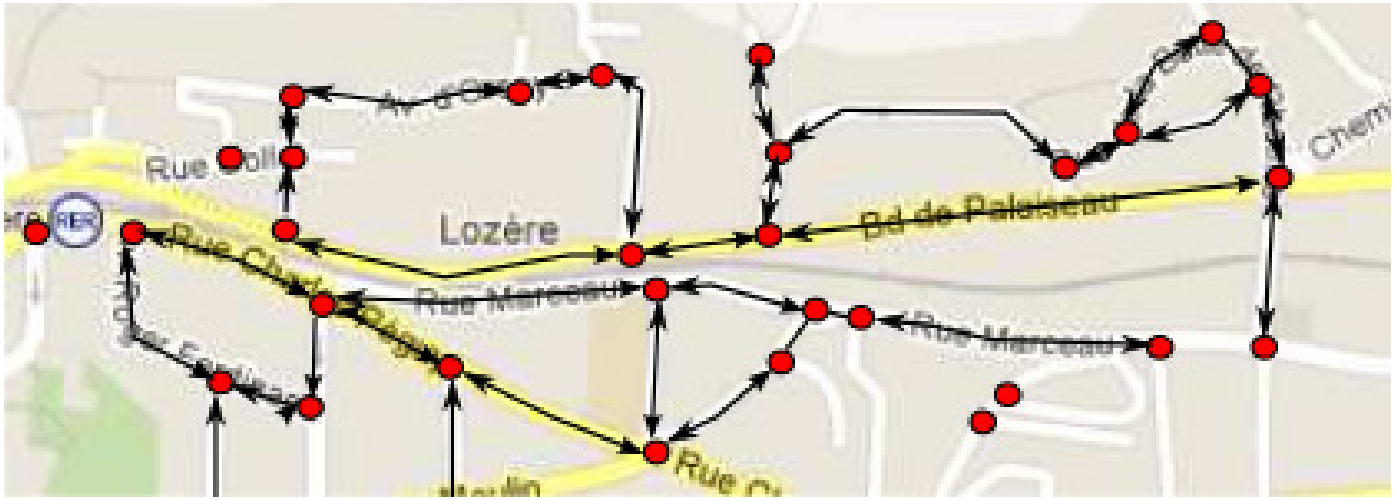

```
class Prio : ...

  def extract (self) :
    if self.curr_size == 0 : raise (EmptyQueue)
    storage = self.storage
    ret_val = storage[0]
    self.curr_size = self.curr_size - 1
    v = storage[self.curr_size]
    storage[0] = v
    i = 0
    while left_child_index (i) < self.curr_size :
      j = left_child_index (i)
      if j + 1 < self.curr_size and storage[j + 1] > storage[j] : j = j + 1
      if v >= storage[j] : break
      storage[i] = storage[j]
      i = j
    storage[i] = v
    return ret_val
```

Les graphes

Motivations : exemple du GPS (1)



Motivations : exemple du GPS (2)



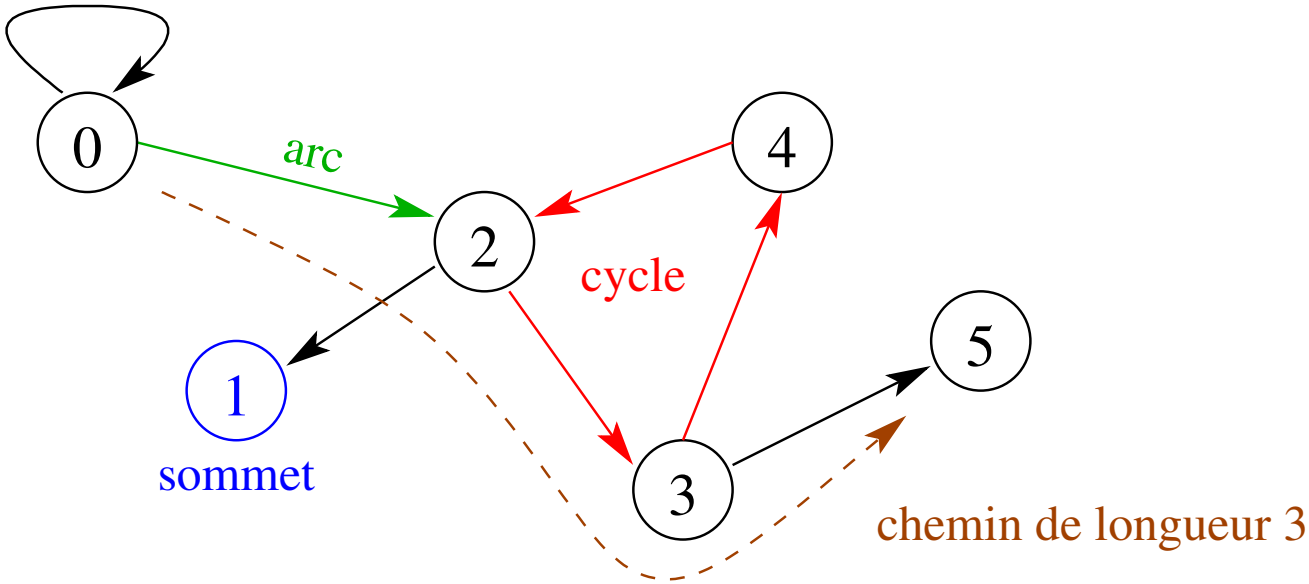
- Intersection = sommet.
- Routes = arcs entre les sommets.
- Routes : peuvent avoir des caractéristiques
  - Une orientation (double-sens, sens unique).
  - Des poids (vitesse, distance, charge. . .)
- Plan = graphe orienté pondéré.
- But : trouver / « optimiser » un trajet ⇒ analyser le graphe.

Quelques utilisations

- Réseaux de communication (routes, trains, métros, télécoms . . .).
- Planification de tâches (dépendance / antériorité).
- Compilation (flot de contrôle de programme).
- Synthèse de types (« *union-find* » & « *path-compression* »).
- Physique (chaînes de Markov).
- Automatique (automates).
- Biologie (réassemblage de sections de génome).
- Etc.

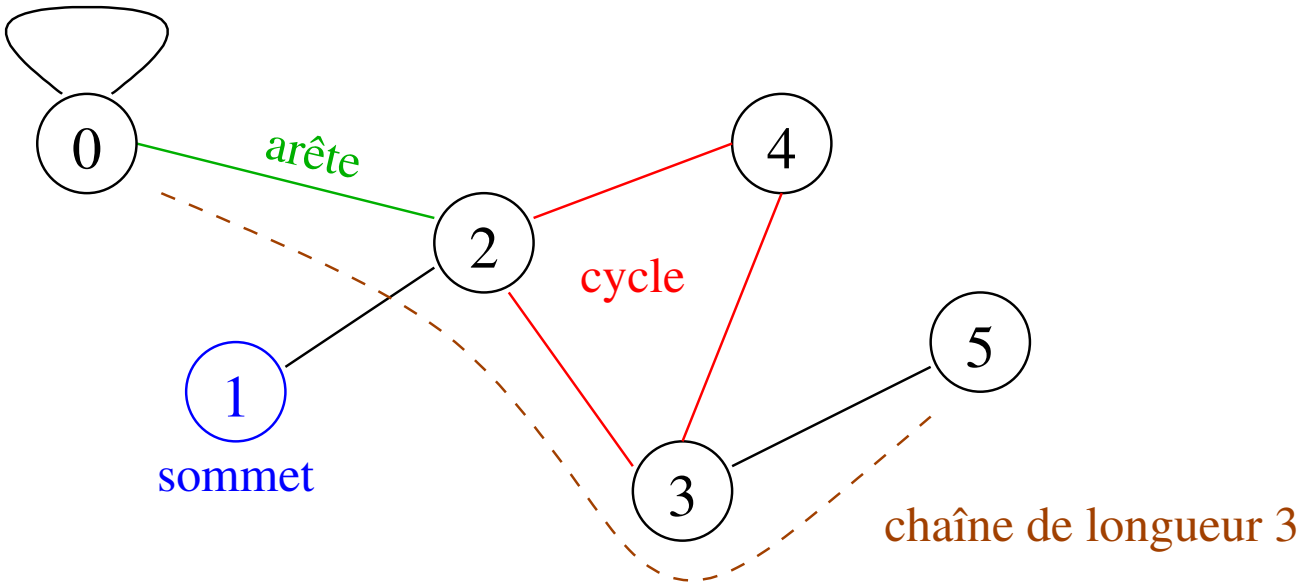
Définitions : graphe orienté

- Graphe orienté : couple (S, A)
  - S : ensemble de sommets (« *vertex/vertices* »).
  - A : ensemble d'arcs (« *edge* »), sous-ensemble de $S \times S$.
- Chemin : suite d'arcs consécutifs (« *path* »).
- Longueur d'un chemin : nombre d'arcs sur ce chemin.
- Chemin simple : chemin où aucun arc n'est parcouru plusieurs fois.
- Cycle : chemin qui boucle.



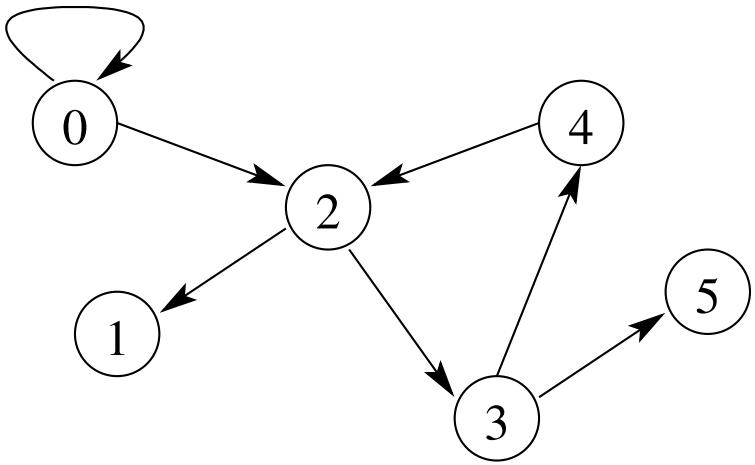
Définitions : graphe non-orienté

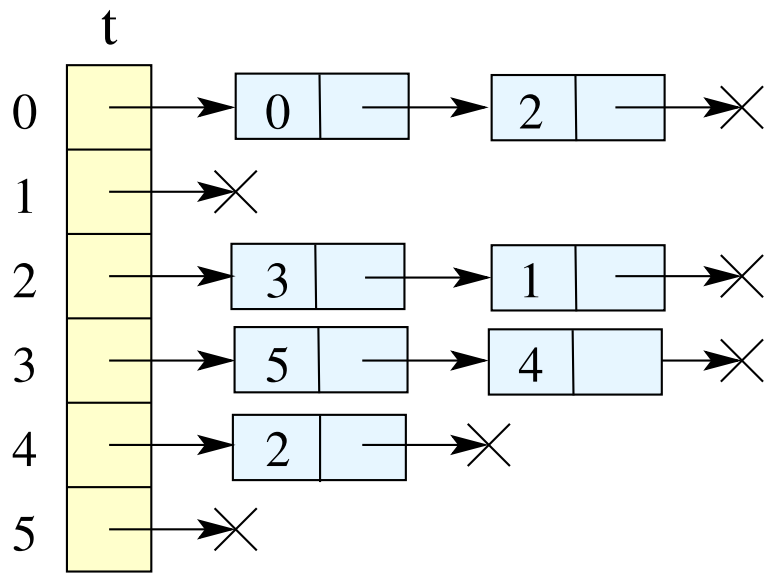
- Graphe non-orienté : comme un graphe orienté . . . sans orientation sur les « *liens* » entre sommets.
- Les « *arcs* » sont souvent appelés arêtes à la place.
- Les « *chemins* » sont souvent appelés chaînes à la place.



Représentation en machine : liste de successeurs (1)

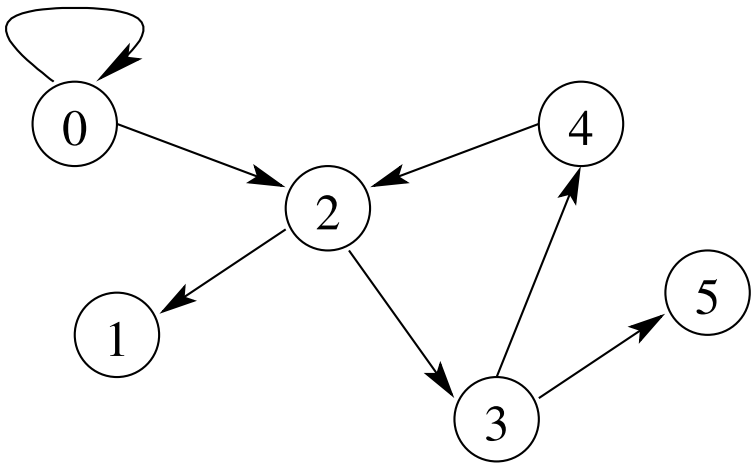
- Méthodes par chaînage et manipulation de pointeurs.
- Plusieurs méthodes selon les besoins.
- Par exemple :
  - Tableau `t` des successeurs de chaque sommet.
  - `t[i]` pointe la liste chaînée (ou tableau) des successeurs du sommet i.
  - Accès direct à un sommet via `t`.
  - Complexité spatiale optimale en $\theta(|S|+|A|)$.

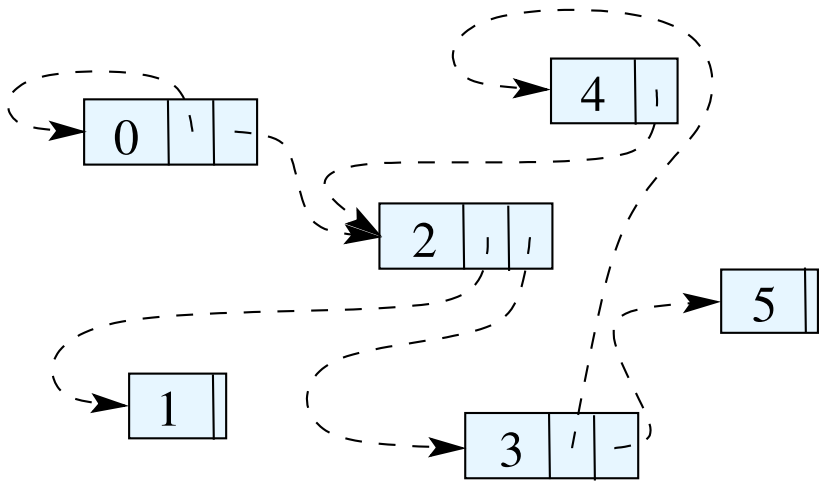




Représentation en machine : liste de successeurs (2)

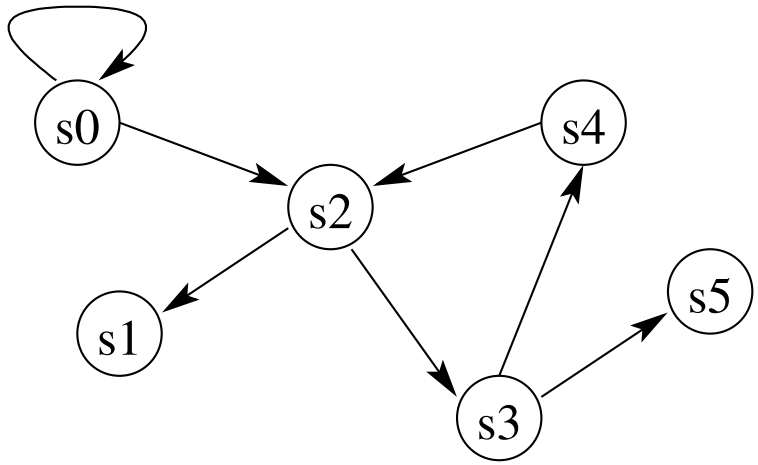
- On peut aussi partager physiquement les sommets.
  - Chaque sommet est représenté 1 et 1 seule fois.
  - Chaque sommet a une liste dont les éléments pointent sur ses successeurs.
- ⇒ Représentation très similaire au «*dessin* ».





Représentation en machine : dictionnaire Python

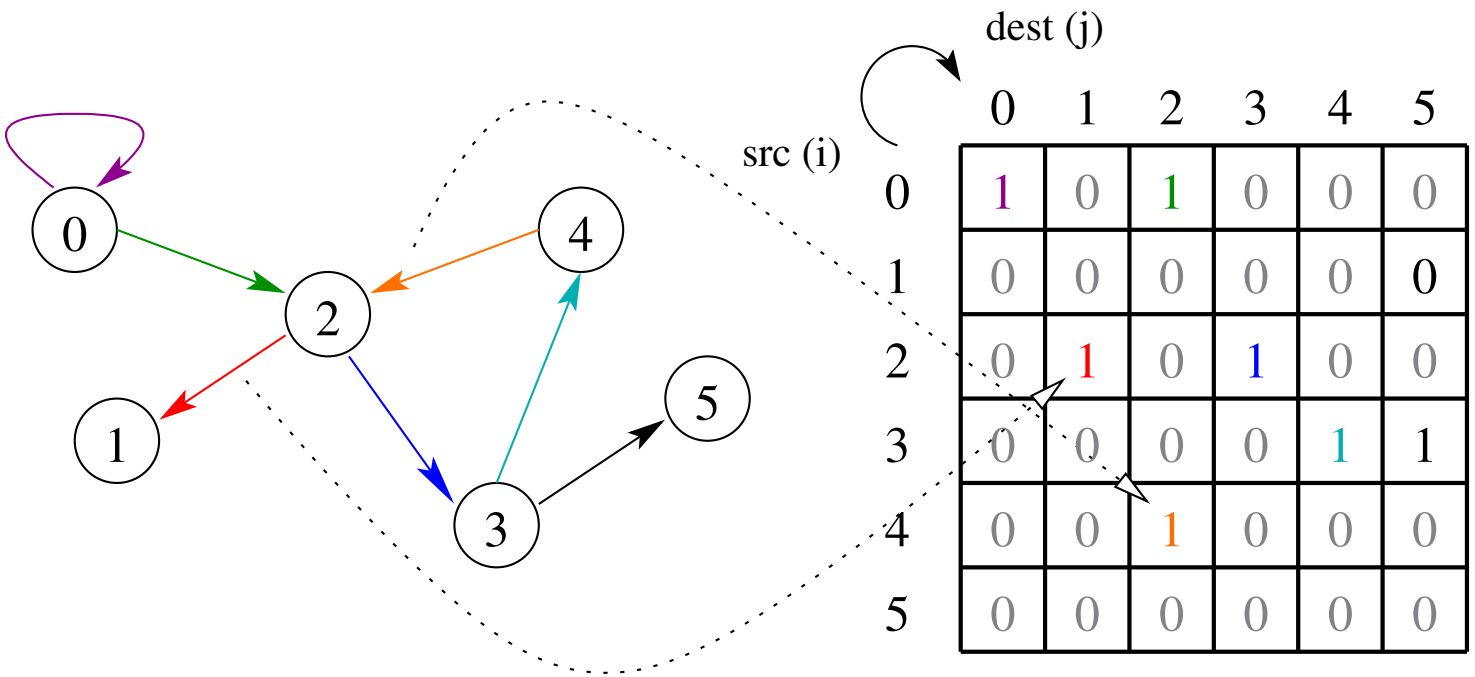
- Permet des sommets qui ne sont pas indicés par des entiers.
- Sommets associés à leur « liste » de successeurs.



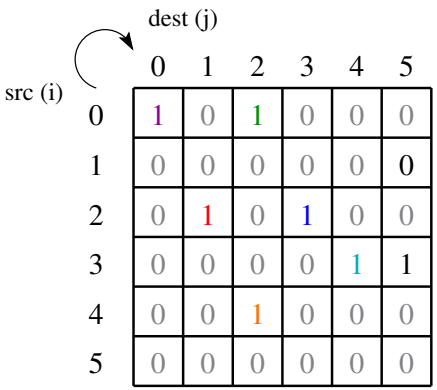

```
g = { 's0' : ['s0', 's2'], 's1' : [], 's2' : ['s1,', 's3'], \
      's3' : ['s4', 's5'], 's4' : ['s2'], 's5' : [] }

g = {}
g['s0'] = ['s0', 's2']
g['s1'] = []
...
```

Représentation en machine : matrice d'adjacence (1)

- Ensemble S des sommets indexés de 0 à $n-1$.
- Arcs $A \subset S \times S$.
- Matrice M de taille $n \times n$ telle que :
  - $M_{ij} = 1$ s'il existe un arc $i \to j$ (i.e. $(i,j) \in A$).
  - $M_{ij} = 0$ sinon.
- Complexité spatiale : $\theta(n^2)$.



Représentation en machine : matrice d'adjacence (2)



- Tableau à 2 dimensions

```
graph = [[]] * NB_VERTICES
for i in range (0, NB_VERTICES) :
  graph[i] = [False] * NB_VERTICES
```

- Tableau à 1 dimension
  - Linéarisation des accès

    graph[i][j] → graph[i * NB_VERTICES + j]

```
graph = [False] * (NB_VERTICES*NB_VERTICES)
```

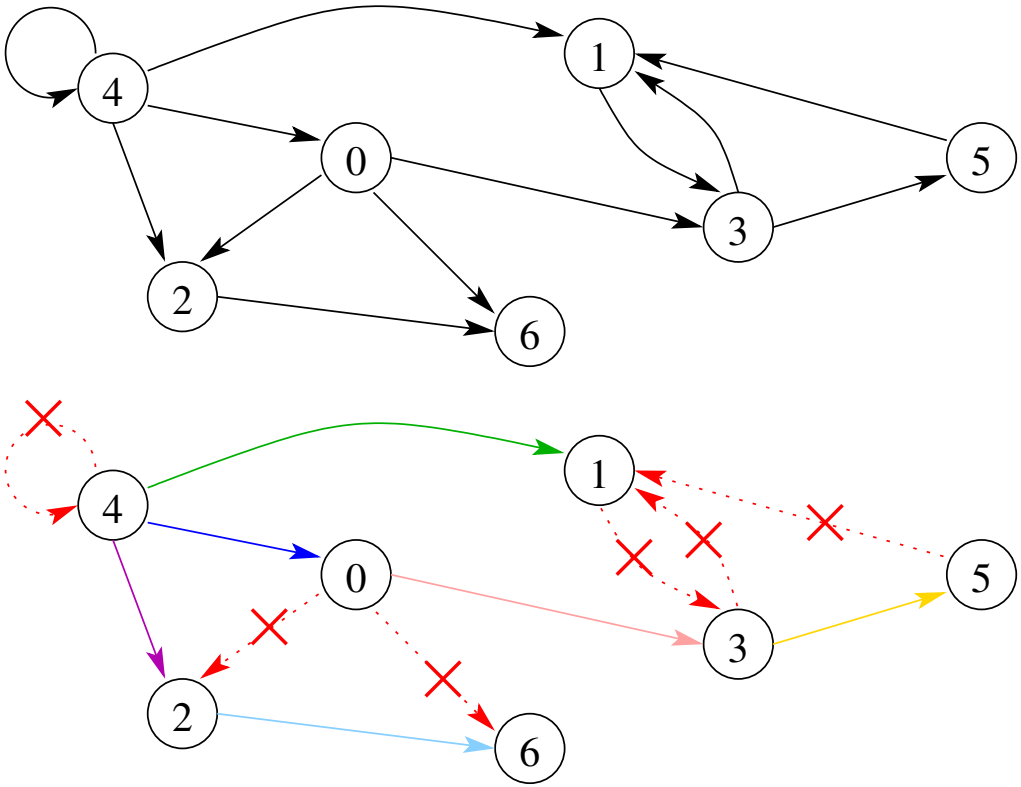
Parcours de graphe : *« pourquoi parcourir ? »*

- Un graphe ne sert pas à stocker des données.
- On construit un graphe à partir de données.
- On analyse le graphe pour déduire des propriétés sur les données.
- Parcours = manière d'analyser un graphe.
- Permet d'extraire de nombreuses informations :
  - Existence de chemin(s).
  - Plus court(s) chemin(s).
  - Composantes (fortement pour graphe orienté) connexes (→ accessibilité).
  - Tri topologique (→ dépendances).
  - Recouvrements (→ sous-graphes, arbres).
  - Etc.

Recouvrement de graphe par un arbre

- Un parcours sert parfois à produire un recouvrement par un arbre.
- Arbre ⇒ chaque sommet a au plus 1 parent.
- Puisque c'est un arbre . . . pas de cycle.
- En quelque sorte, c'est le graphe « amputé » de certains arcs.
- But : « *simplifier* » un graphe
  - Garder seulement une information « *intéressante* ».

Exemple de recouvrement



Noeud	Père
0	4
1	4
2	4
3	0
4	–
5	3
6	2

- Représentation par un tableau de pères.
- On cherchera des recouvrement «*intéressants*».
- Celui de l'exemple ? Fait au hasard . . .

Parcours en largeur

- Équivalent du parcours par niveaux des arbres :
  - On part d'un sommet,
  - on visite tous les voisins,
  - on visite tous les voisins des voisins...

⇒ Sommets de distance d découverts avant ceux de distance $d+1$.

- Problèmes :
  - Ne pas boucler en visitant les cycles.
  - Ne pas visiter plusieurs fois le même sommet (en passant par des « raccourcis »).

⇒ Marquage des sommets lors du parcours :

  - « *Blanc* » : non visité (ou « *non marqué* »).
  - (« *Gris* » : en cours de visite, si besoin.)
  - « *Noir* » : déjà visité (ou « *marqué* »).

Parcours en largeur : algorithme général

- Utilisation d'une file F :
  - contient initialement 1 seul sommet (racine).
- Complexité en $O(|A|)$ pour une représentation par listes.

```
Soit r la racine ;
Marquer r ;
Enfiler r dans la file F ;
Tant que F n'est pas vide {
  s = élément en tête de F ;
  Traiter (s) ;
  Pour chaque voisin v du sommet s {
    Si v n'est pas marqué alors {
      Marquer v ;
      Enfiler v dans la file F ;
    }
  }
}
```

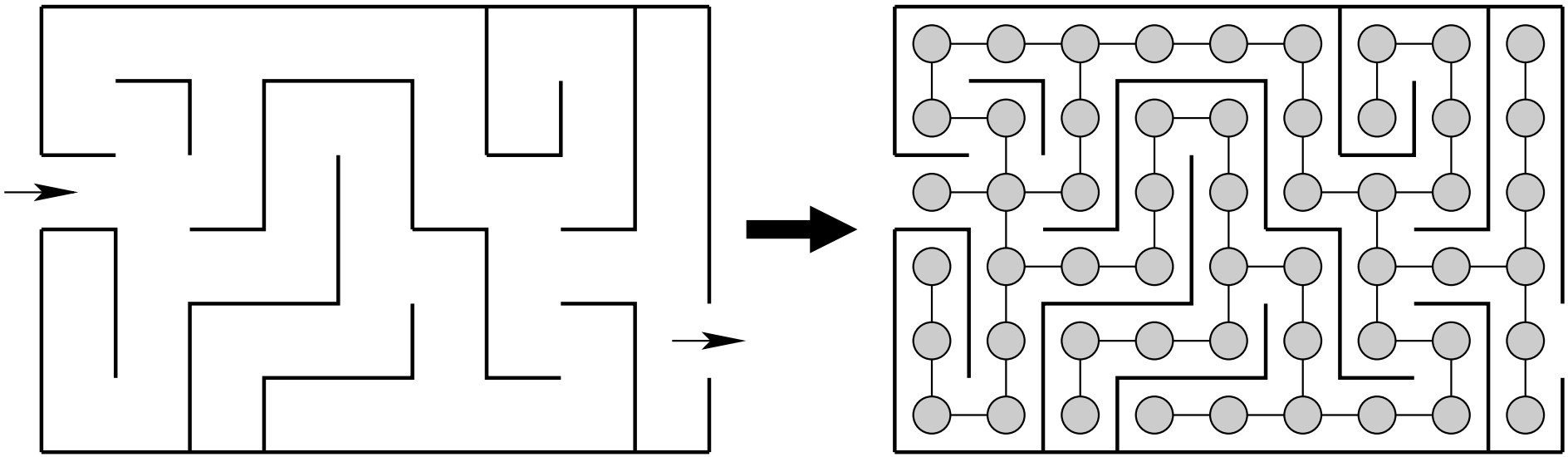
Parcours en largeur : implantation

```
def bfs (g, root) :
  seen = [root]
  q = [root]
  while q != [] :
    n = q.pop (0)
    print ("Processing node", n)
    for s in g[n] :
      if not (s in seen) :
        seen.append (s)
        q.append (s)
```

- Utilisation d'une « fausse » file.
- « Tableaux » à la `Python` avec :
  - re-dimensionnement,
  - ajout en queue,
  - retrait en tête.
- ⇒ Cache les difficultés d'implantation des structures sous-jacentes.
- ⇒ Pas forcément idéal pour l'enseignement.

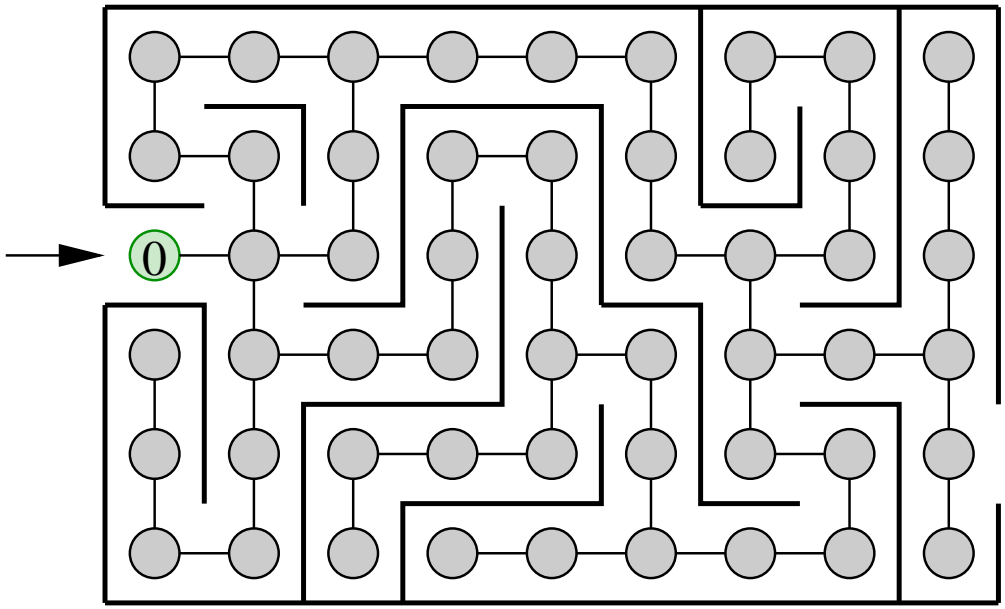
Parcours en largeur : application à un labyrinthe (1)

- But : trouver la sortie dans un labyrinthe...
- ... par le plus court chemin.
- ⇒ Créer un graphe à partir du labyrinthe.



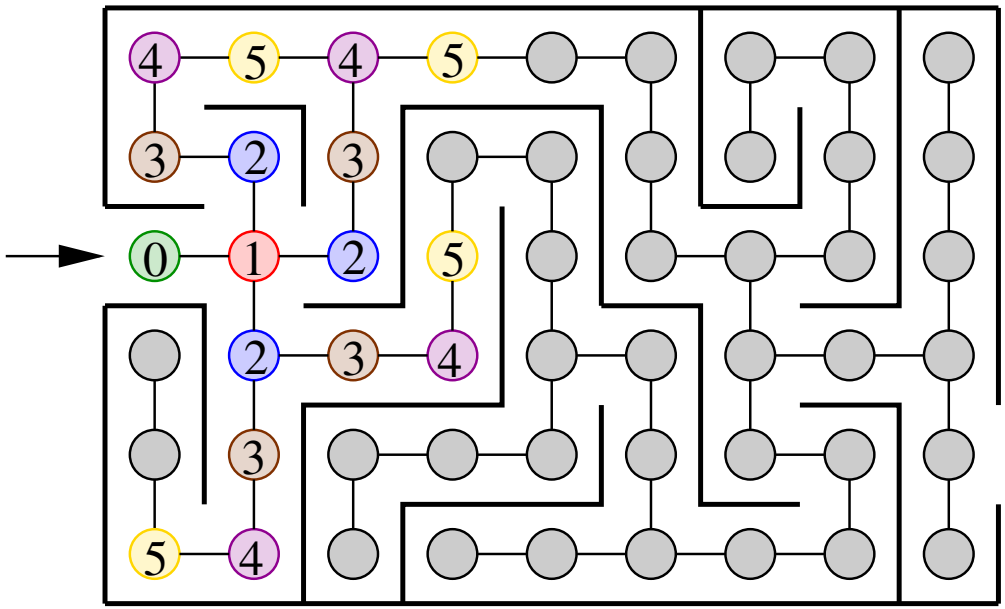
Parcours en largeur : application à un labyrinthe (2)

- On commence le parcours par l'entrée (sommet vert).



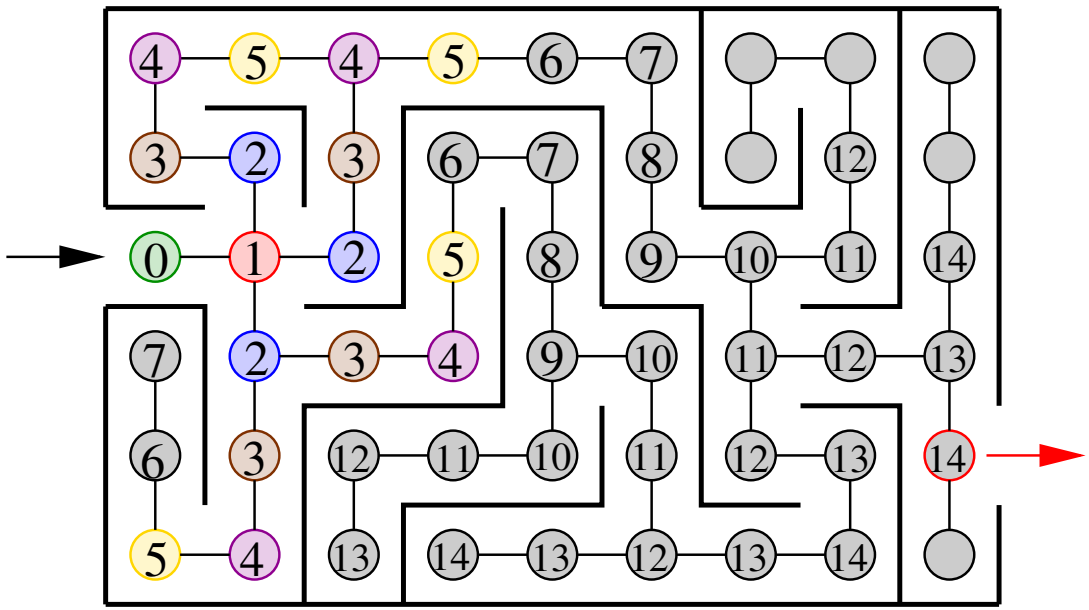
Parcours en largeur : application à un labyrinthe (3)

- On commence le parcours par l'entrée (sommet vert).
- On descend en largeur en mémorisant le père de chaque sommet visité.



Parcours en largeur : application à un labyrinthe (3)

- On commence le parcours par l'entrée (sommet vert).
- On descend en largeur en mémorisant le père de chaque sommet visité.
- Arrivé à la sortie, on retrace le chemin en remontant les pères.



Parcours en profondeur : algorithme général

- Équivalent du parcours en profondeur des arbres :
  - On descend au plus profond en premier.
  - Algorithme naturellement récursif.
- Mêmes problèmes que pour le parcours en largeur ⇒ marquage.
- Complexité en $O(|A|)$ pour une représentation par listes.

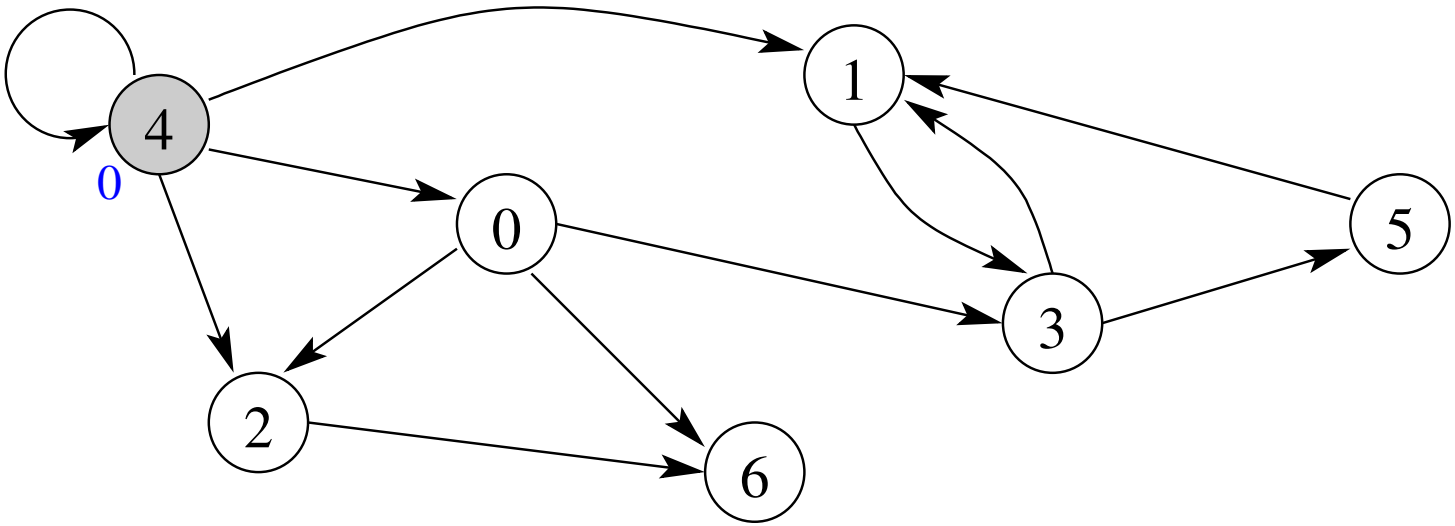
```
DFS (sommet s) {
  Marquer s
  (Pré-traiter s) # Selon le besoin.
  Pour chaque voisin non marqué v du sommet s
    DFS (v)
  (Post-traiter s) # Selon le besoin.
  }
}
```

Parcours en profondeur : mémorisation des dates de visite

- On peut conserver une variable « *temps* » t . . .
- . . . et enregistrer les dates de début / fin (*beg*[] / *end*[]) de visite.

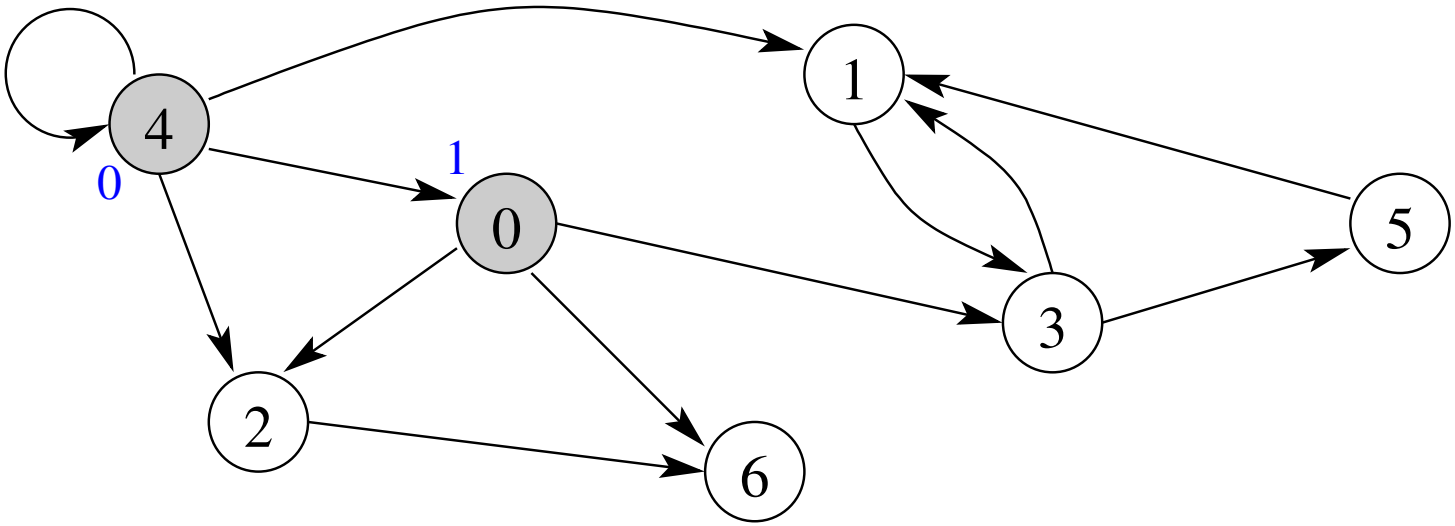
```
DFS (sommet s) {
   Marquer s
   beg[s] ← t     # Mémoriser date début.
   t ← t + 1
   Pour chaque voisin non marqué v du sommet s {
      par[v] ← s  # Mémoriser la parenté.
      DFS (v)
   }
   end[s] ← t     # Mémoriser date fin.
   t ← t + 1
}
```

Parcours en profondeur : exemple (1)



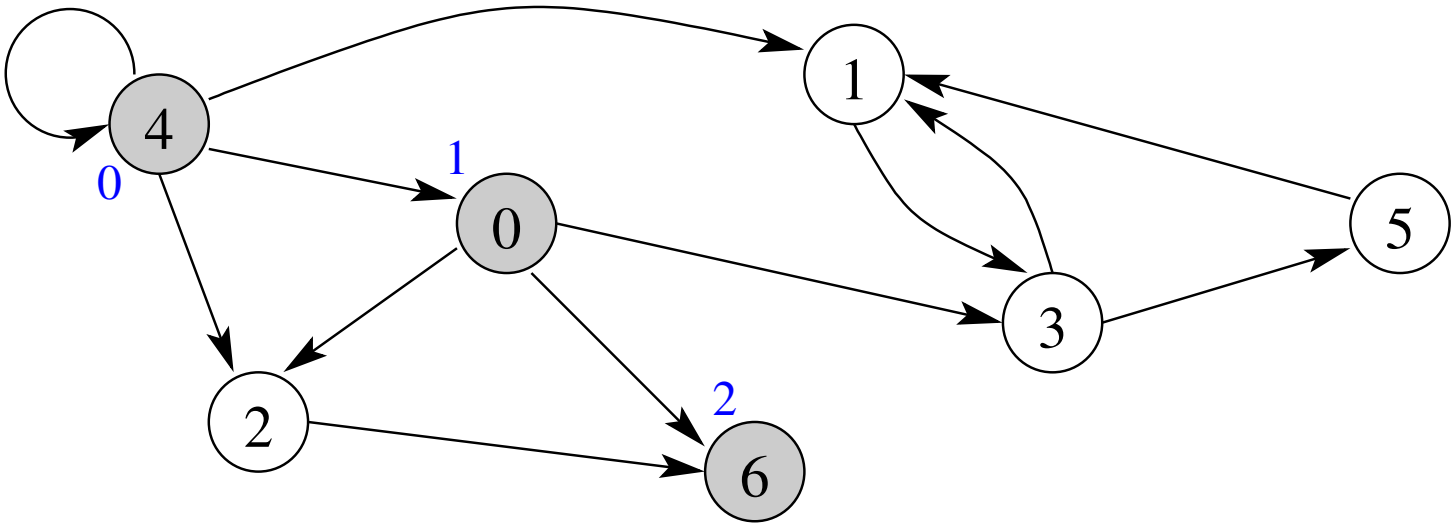
- On commence par le sommet 4, pas encore marqué, on le marque.
- On inspecte son premier voisin . . . lui-même.
- Déjà marqué, donc on ne visite pas dans cette direction.

Parcours en profondeur : exemple (2)



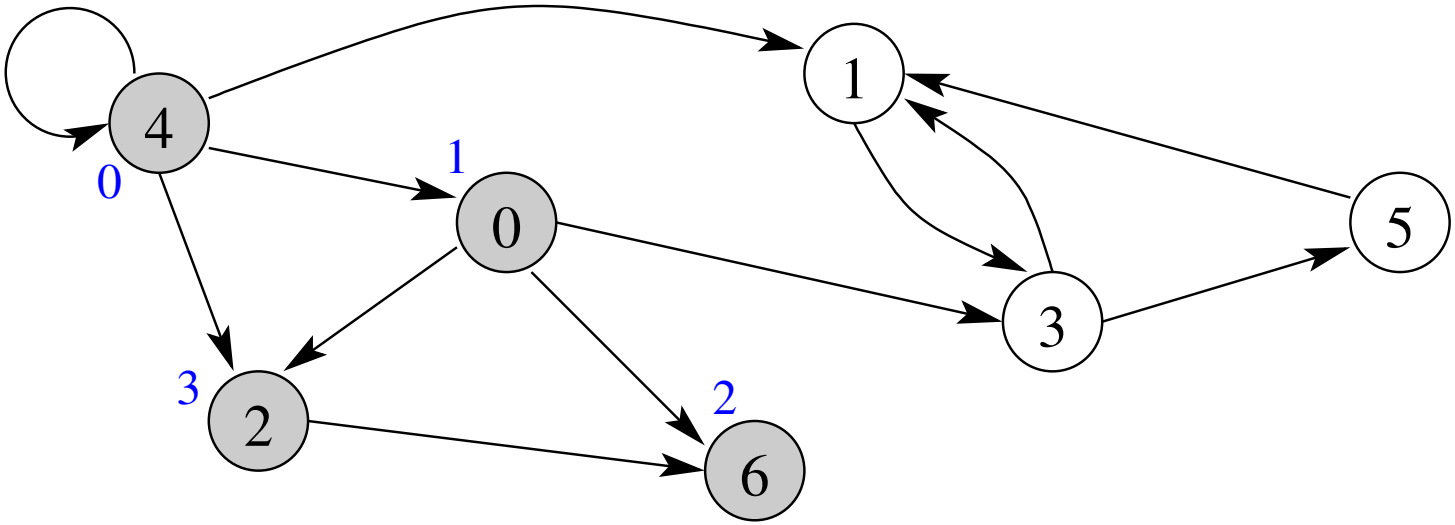
- On inspecte un autre voisin de 4 : par exemple 0.
- Il n'est pas encore marqué, donc on va le visiter.

Parcours en profondeur : exemple (3)



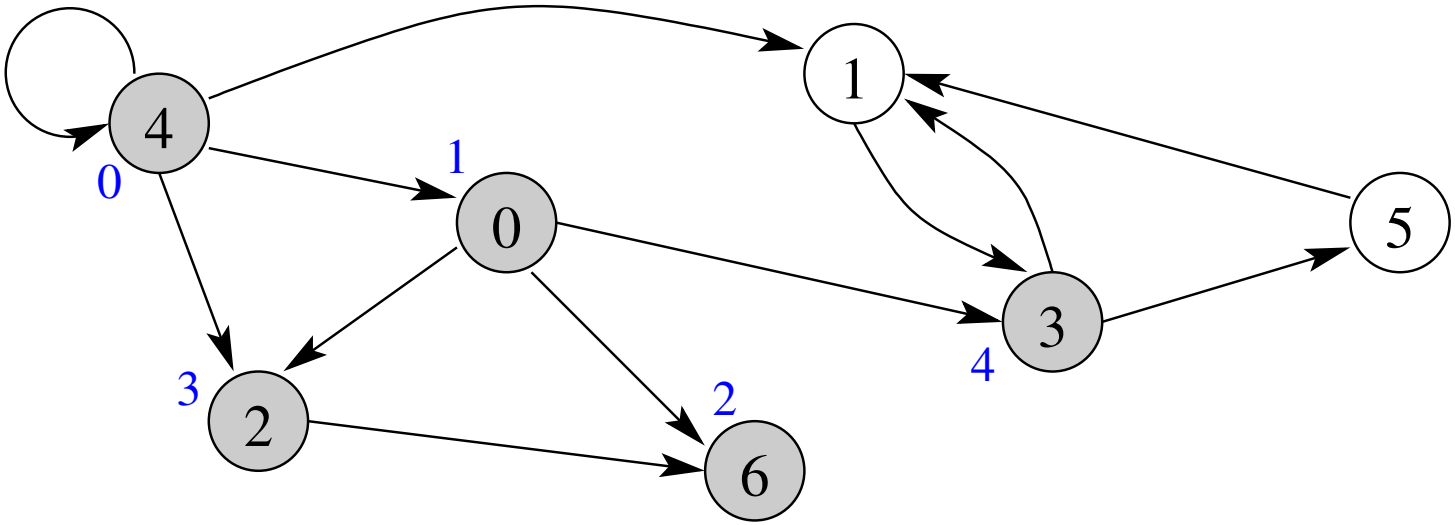
- On inspecte un voisin de 0 : par exemple 6.
- Il n'est pas encore marqué, donc on va le visiter.

Parcours en profondeur : exemple (4)



- Le sommet 6 n'a pas de voisin ⇒ descente terminée.
- On va inspecter un autre voisin de 0 : par exemple 2.
- Pas encore marqué donc on le visite.
- Puis le voisin de 2 : 6. Déjà marqué ⇒ descente terminée.

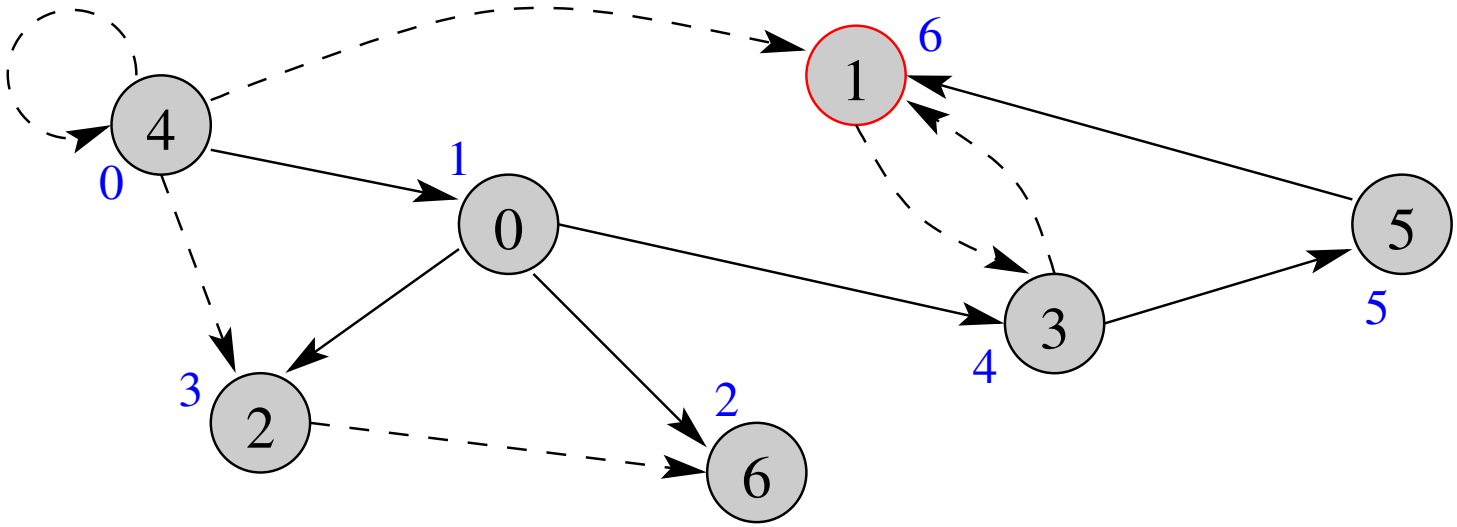
Parcours en profondeur : exemple (5)



- On remonte et va inspecter le dernier voisin de 0 : 3.
- Pas marqué donc on le visite.
- Etc.

Parcours en profondeur : exemple (6)

- À la fin on obtient un recouvrement du graphe.
- En rouge, le dernier sommet visité.
- En pointillé les arcs non suivis lors de la visite.
- Dans un graphe orienté sans cycle :
  - Les fins de visite forment un ordre.
  - S'il existe un chemin de u à v alors $end[u] > end[v]$.

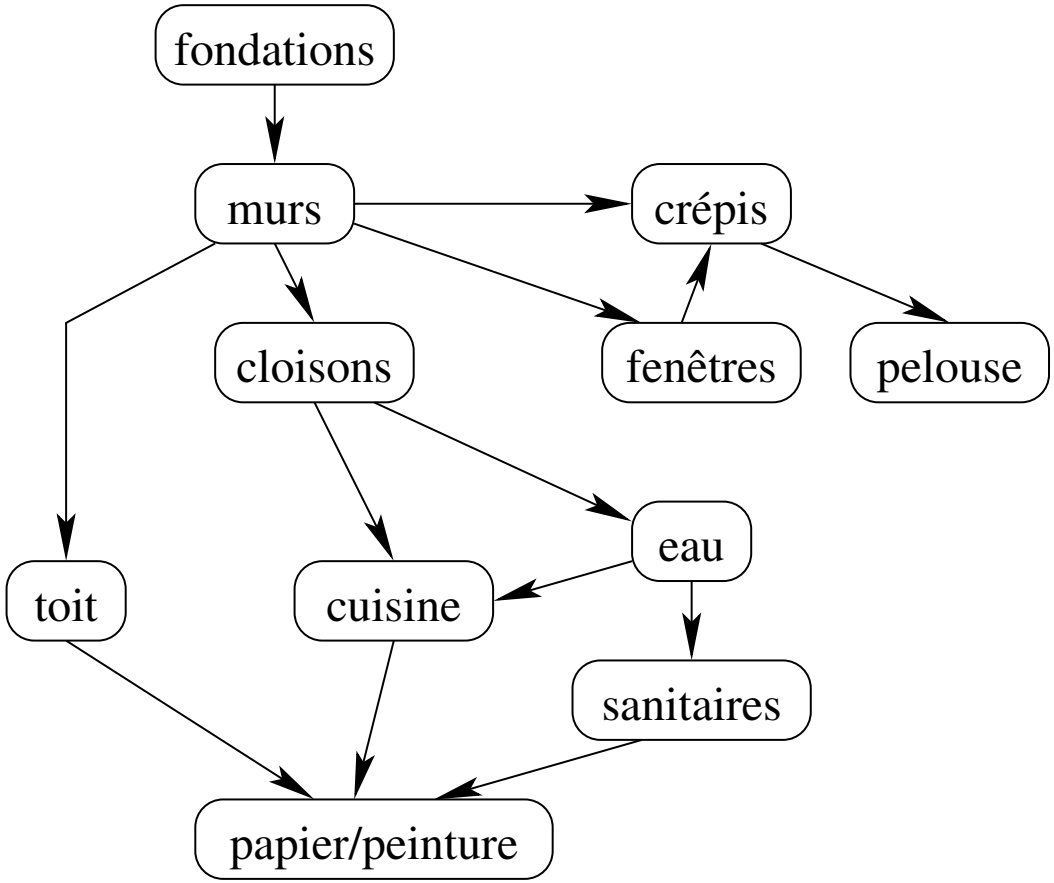


Application du parcours en profondeur : tri topologique

- Soit un ensemble de tâches à réaliser selon des contraintes d'ordre.
- Certaines doivent être réalisées avant d'autres.
- Exemples :
  - Gestionnaire de compilation (`make`).
  - Chaîne de montage en usine.
  - Gestion de projets, construction de bâtiment.
- Construire un graphe de dépendances.
- Le parcourir en profondeur en notant les dates de fin.
- Retourner les sommets en ordre décroissant des dates de fin.
- S'il y a un cycle ⇒ impossible de planifier !

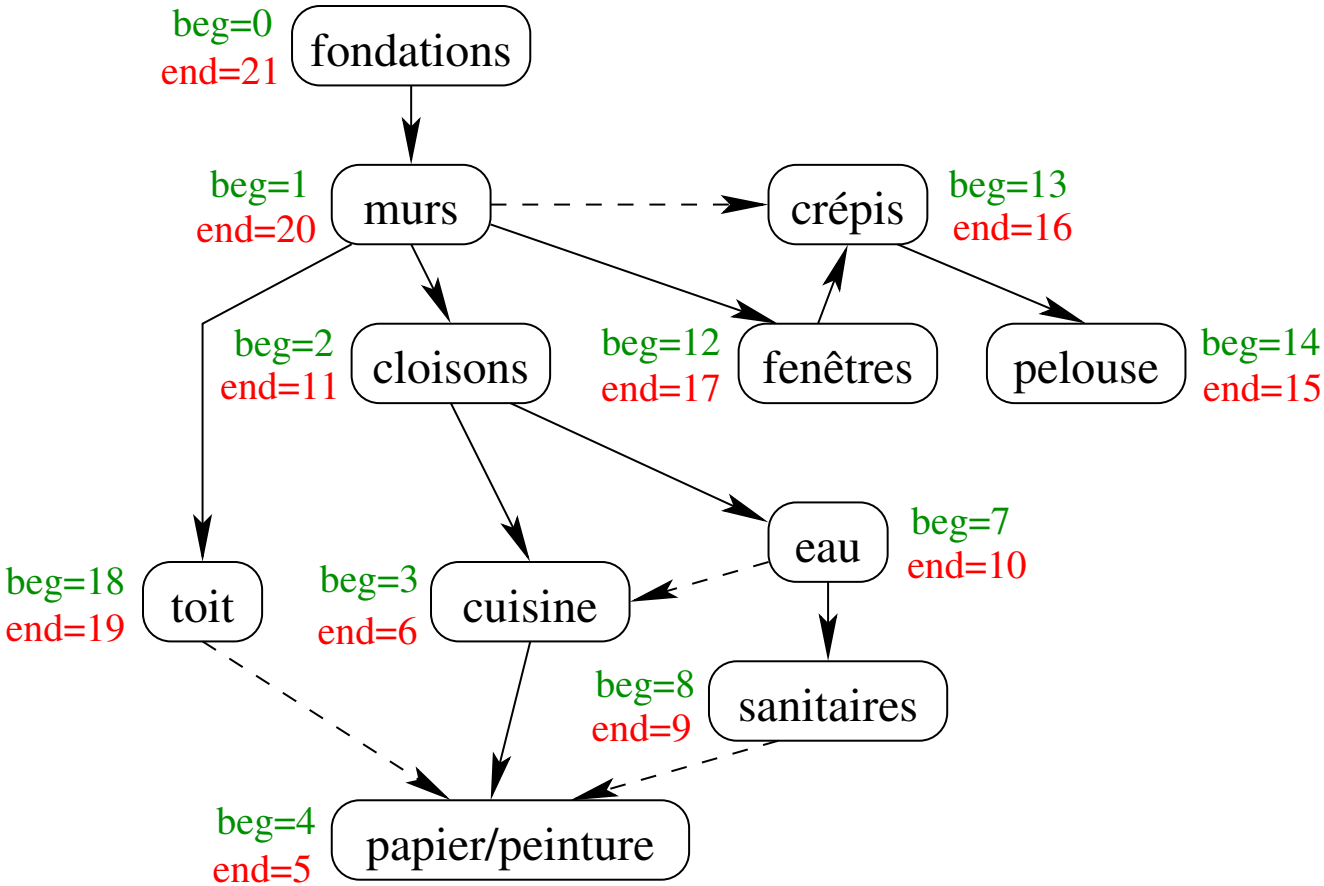
Tri topologique et ordonnancement (1)

- Construire une maison...
- Ici, arc $x \rightarrow y$: « faire x puis y ».



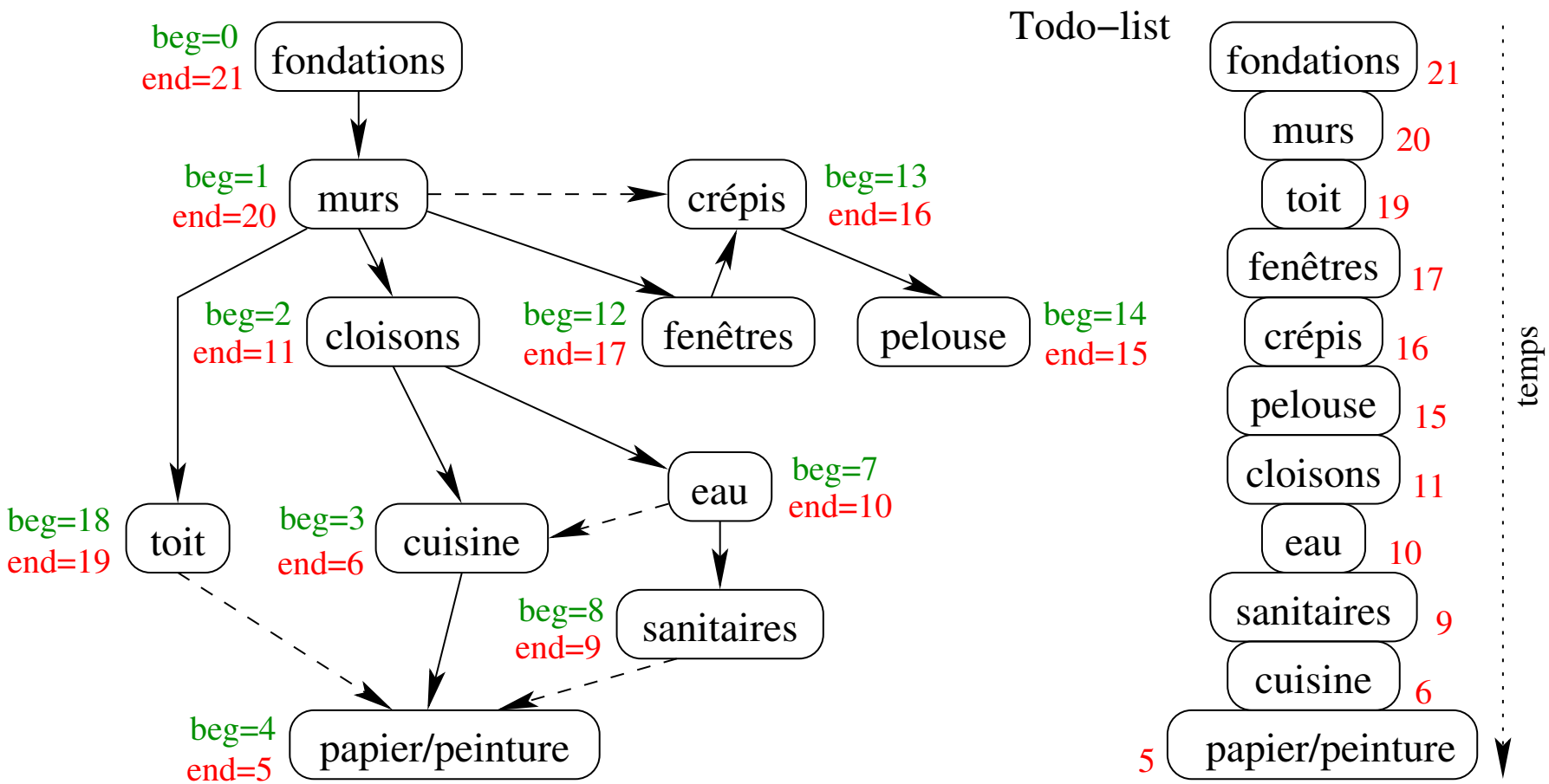
Tri topologique et ordonnancement (2)

- Parcours en profondeur...



Tri topologique et ordonnancement (3)

- Tâches à effectuer en ordre décroissant des dates de fin de visite.



Tri topologique et ordonnancement : en Python

```
g = {
  'fondations' : ['murs'], \
  'murs' : ['toit', 'cloisons', 'crépis', 'fenêtres'], ... }

def topo (graph, root) :
  date = 0   # Date courante (strictement croissante).
  seen = []  # Tableau des noeuds déjà visités.
  begin_dates, end_dates = {}, {}  # Dates de début et fin de visite.

  def __dfs (g, n) :
    nonlocal date, seen, begin_dates, end_dates
    seen.append (n)        # Marquer le noeud courant.
    begin_dates[n] = date
    date = date + 1
    for s in g[n] :        # Explore chaque voisin non déjà marqué.
      if not (s in seen) :
        __dfs (g, s)
    end_dates[n] = date
    date = date + 1

  __dfs (graph, root)
  return end_dates

end_dates = topo (g, 'fondations')
print (sorted (end_dates.items (), key = lambda x: x[1], reverse=True))
```

Matrice d'adjacence : intéressante malgré tout...

- Rappel : matrice $n \times n \Rightarrow$ complexité spatiale importante.
- Possibilité de «*compression* » si matrice binaire $\Rightarrow$ 1 bit par «*case* ».
- Mais souvent, quand même trop lourd en espace mémoire.

  On a vus des algos linéaires avec une représentation par liste de successeurs ! Pourquoi revenir sur une matrice ?

- Elle permet de traiter plusieurs problèmes à la fois :
  - Calculer tous les chemins de longueur l d'un coup.
  - Trouver tous les plus courts chemins d'un coup.
  - Calculer une fermeture transitive.
  - Matrice $\Rightarrow$ toute l'artillerie d'algèbre linéaire...

Calcul du nombre de chemins

- Soit M la matrice $n \times n$ d'adjacence d'un graphe.
- Le nombre de chemins de longueur k reliant i à j est $M^k_{i,j}$.

- Preuve par récurrence sur l'exposant k :
  - Pour $k = 1$, immédiat puisque matrice initiale binaire, (des 1 et 0) et un arc du graphe est un chemin de longueur 1.
  - Pour $k > 1$ on a $M^k = M^{k-1} \times M$.
  - Donc pour $k > 1$, par définition du produit matriciel,

$$M^k_{i,j} = \sum_{l=1}^{n} (M^{k-1}_{i,l} \times M_{l,j})$$

  - Par hypothèse de récurrence, $M^{k-1}_{i,l}$ est le nombre de chemins de longueur $k-1$ de i à l.
  - $M_{l,j}$ est égal à 1 si $l \mapsto j$ est un arc de G et à 0 sinon.
  - Donc, $M^{k-1}_{i,l} \times M_{l,j}$ est le nombre de chemins de i à j de longueur k dont le dernier arc est (l,j) (0 si pas d'arc, $M^{k-1}_{i,l}$ sinon).
  - Donc la sommation $\sum_{l=1}^{n}$ donne donc toutes les possibilités de longueur k de i à j quelque soit le sommet de départ l du dernier arc.

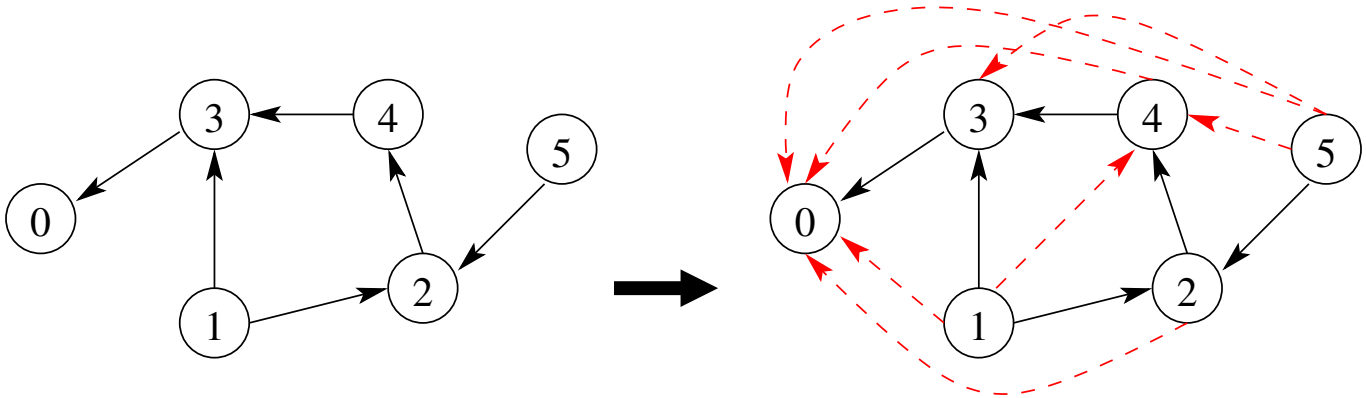
Existence de chemins

- Dans un graphe à n sommets, si un chemin existe entre i et j il est au plus de longueur n.
- On pose $M^+ = M + M^2 + M^3 + ... + M^n$.
- M^+ est le nombre de chemins de longueurs 1 à n reliant i à j.
- Si $M^+_{i,j} = 0$, il n'y a pas de chemin.
- Il existe un chemin de i a j si et seulement si $M^+_{i,j} \neq 0$.

```
exists_path (i, j, mat) {
  prod_mat ← mat
  sum_mat ← mat
  k = 1
  tant que sum_mat[i][j] == 0 et k < n {
    prod_mat ← prod_mat × mat
    sum_mat ← sum_mat + prod_mat
    k = k + 1
  }
  retourner (sum_mat[i][j] != 0)
}
```

Fermeture transitive d'un graphe $G = (S, A)$

- Relation transitive minimale contenant la relation (S, A).
- Graphe G^* tel que chaque chemin de G est représenté par un arc.



- Permet de répondre aux questions d'existence de chemins entre 2 sommets quelconques.
- Naïvement on peut calculer sa matrice M^* à partir de M^+ :
  - $M^*_{i,j} = 0$ si $M^+_{i,j} = 0$.
  - $M^*{}_{i,j} = 1$ sinon.
  - Complexité en $\Theta(n^4)$ ☹

Fermeture transitive plus efficace

- Algorithme de Roy-Warshall.
- Justification non triviale (itération d'une opération d'ajout d'arcs).
- Complexité en $\theta(n^3)$.
- Utilisation de simples opérations booléennes.

```
pour k = 0 à n exclu {
  pour i = 0 à n exclu {
    pour j = 0 à n exclu {
      M[i][j] ← M[i][j] ou (M[i][k] et M[k][j])
    }
  }
}
```

Avez-vous des questions ?