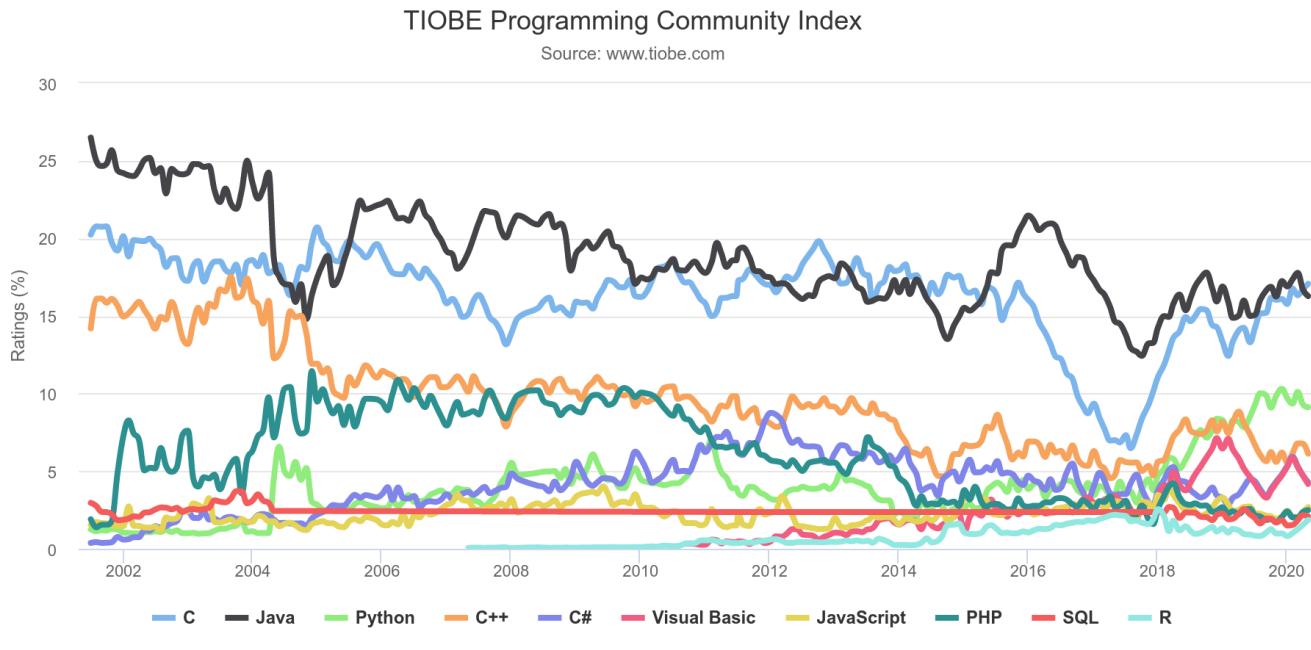


PROGRAMMATION

Pierre Bénard – Xavier Blanc

Arthur Chevalier – David Renault

INTRODUCTION



Ceci n'est pas un cours de programmation.

LE PROGRAMME (1/2)

Notion de programme en tant que donnée. Calculabilité, décidabilité.	Comprendre que tout programme est aussi une donnée. Comprendre que la calculabilité ne dépend pas du langage de programmation utilisé. Montrer, sans formalisme théorique, que le problème de l'arrêt est indécidable.	L'utilisation d'un interpréteur ou d'un compilateur, le téléchargement de logiciel, le fonctionnement des systèmes d'exploitation permettent de comprendre un programme comme donnée d'un autre programme.
Récurivité	Écrire un programme récursif. Analyser le fonctionnement d'un programme récursif.	Des exemples relevant de domaines variés sont à privilégier.
Modularité	Utiliser des API (Application Programming Interface) ou des bibliothèques. Exploiter leur documentation. Créer des modules simples et les documenter.	
Paradigmes de programmation	Distinguer sur des exemples les paradigmes impératif , fonctionnel et objet . Choisir le paradigme de programmation selon le champ d'application d'un programme.	Avec un même langage de programmation, on peut utiliser des paradigmes différents. Dans un même programme, on peut utiliser des paradigmes différents.
Mise au point des programmes. Gestion des bugs.	Dans la pratique de la programmation, savoir répondre aux causes typiques de bugs : problèmes liés au typage, effets de bord non désirés, débordements dans les tableaux, instruction conditionnelle non exhaustive, choix des inégalités, comparaisons et calculs entre flottants, mauvais nommage des variables, etc.	On prolonge le travail entrepris en classe de première sur l'utilisation de la spécification , des assertions, de la documentation des programmes et de la construction de jeux de tests. Les élèves apprennent progressivement à anticiper leurs erreurs.

LE PROGRAMME (2/2)

Structures de données, interface et implémentation.	Spécifier une structure de données par son interface. Distinguer interface et implémentation. Écrire plusieurs implémentations d'une même structure de données.	L'abstraction des structures de données est introduite après plusieurs implémentations d'une structure simple comme la file (avec un tableau ou avec deux piles).
Vocabulaire de la programmation objet : classes, attributs, méthodes, objets.	Écrire la définition d'une classe. Accéder aux attributs et méthodes d'une classe.	On n'aborde pas ici tous les aspects de la programmation objet comme le polymorphisme et l'héritage.
Listes, piles, files : structures linéaires. Dictionnaires, index et clé.	Distinguer des structures par le jeu des méthodes qui les caractérisent. Choisir une structure de données adaptée à la situation à modéliser. Distinguer la recherche d'une valeur dans une liste et dans un dictionnaire.	On distingue les modes FIFO (first in first out) et LIFO (last in first out) des piles et des files.
Arbres : structures hiérarchiques. Arbres binaires : noeuds, racines, feuilles, sous-arbres gauches, sous-arbres droits.	Identifier des situations nécessitant une structure de données arborescente. Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).	On fait le lien avec la rubrique «algorithmique».
Graphes : structures relationnelles. Sommets, arcs, arêtes, graphes orientés ou non orientés.	Modéliser des situations sous forme de graphes. Écrire les implémentations correspondantes d'un graphe : matrice d'adjacence, liste de successeurs/de prédécesseurs. Passer d'une représentation à une autre.	On s'appuie sur des exemples comme le réseau routier, le réseau électrique, Internet, les réseaux sociaux. Le choix de la représentation dépend du traitement qu'on veut mettre en place : on fait le lien avec la rubrique «algorithmique».

PARADIGME DE PROGRAMMATION

- Un **paradigme** est un modèle général pour **décomposer** les problèmes et **composer** des solutions. Il permet de concevoir des systèmes complexes à partir d'éléments simples et composable.
- A chaque paradigme est associé un ensemble de **briques élémentaires** que l'on cherche à composer.
- Les paradigmes sont compris et analysés en fonctions des propriétés de leurs briques élémentaires, qui permettent à leur tour d'identifier des constructions plus complexes.
- En fait, un paradigme **façonne** la forme des solutions apportées à un problème donné.

LISTE DE PARADIGMES (SUR WIKIPEDIA)

Programming paradigms	
<ul style="list-style-type: none">• Action• Agent-oriented• Array-oriented• Automata-based• Concurrent computing<ul style="list-style-type: none">• Relativistic programming• Data-driven• Declarative (contrast: Imperative)<ul style="list-style-type: none">• Functional<ul style="list-style-type: none">• Functional logic• Purely functional• Logic<ul style="list-style-type: none">• Abductive logic• Answer set• Concurrent logic• Functional logic• Inductive logic• Constraint<ul style="list-style-type: none">• Constraint logic<ul style="list-style-type: none">• Concurrent constraint logic• Dataflow<ul style="list-style-type: none">• Flow-based• Reactive• Ontology• Differentiable• Dynamic/scripting• Event-driven• Function-level (contrast: Value-level)<ul style="list-style-type: none">• Point-free style• Concatenative• Generic• Imperative (contrast: Declarative)<ul style="list-style-type: none">• Procedural• Object-oriented• Polymorphic	<ul style="list-style-type: none">• Language-oriented<ul style="list-style-type: none">• Domain-specific• Literate• Natural-language programming• Metaprogramming<ul style="list-style-type: none">• Automatic• Inductive programming• Reflective<ul style="list-style-type: none">• Attribute-oriented• Macro• Template <ul style="list-style-type: none">• Non-structured (contrast: Structured)<ul style="list-style-type: none">• Array• Nondeterministic• Parallel computing<ul style="list-style-type: none">• Process-oriented• Probabilistic• Quantum• Set-theoretic• Stack-based <ul style="list-style-type: none">• Structured (contrast: Non-structured)<ul style="list-style-type: none">• Block-structured<ul style="list-style-type: none">• Structured concurrency• Object-oriented<ul style="list-style-type: none">• Actor-based• Class-based• Concurrent• Prototype-based• By separation of concerns:<ul style="list-style-type: none">• Aspect-oriented• Role-oriented• Subject-oriented• Recursive• Symbolic• Value-level (contrast: Function-level)

https://en.wikipedia.org/wiki/Programming_paradigm

LISTE DE PARADIGMES (BRIQUES)

Paradigme	Brique élémentaire
Impératif	Instruction
Fonctionnel	Expression, Fonction
Objet	Objet, Message
Logique	Formule logique
Parallèle	Algorithmes parallèles
Événementiel	Événement, Cible

EXEMPLE DE PARADIGME : L'IMPÉRATIF

- Brique élémentaire : l'**instruction**
- Idée générale :
 - la machine possède un **état** (mémoire, registres, ...),
 - chaque instruction modifie cet état,
 - les instructions se composent en **séquence**.
- Représentants historiques : Fortran (1954), Algol (1958)
- Exemple emblématique : **C** (1972, dernière norme : 2018)

CAS D'ÉTUDE : LA SUITE DE FIBONACCI

- Un objet mathématique simple à définir :

$$\begin{cases} f_0 = 1 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} & \text{si } n \geq 2 \end{cases}$$

1, 1, 2, 3, 5, 8, 13, 21, 34 ...

- Un programme simple à spécifier :

```
def fibo(n):  
    """Assuming that n is a non-negative integer (i.e n >= 0)  
    Returns the n-th element of the Fibonacci sequence """
```

- Plusieurs exemples tirés de [Rosetta Code](#).

CODE IMPÉRATIF : ASSEMBLEUR

Calcul de la suite de Fibonacci en Assembleur 8080

```
fibncl: mov C, A ; C will store the counter
        dcr C ; decrement because we know f(1) already
        mvi A, 1
        mvi B, 0
loop:   mov D, A
        add B ; A := A + B
        mov B, D
        dcr C
        jnz loop ; jump if not zero
        ret ; return from subroutine
```

https://rosettacode.org/wiki/Fibonacci_sequence#8080_Assembly

CODE IMPÉRATIF : FORTRAN

Calcul de la suite de Fibonacci en Fortran IV (1962)

```
FUNCTION IFIBO(N)
  IF(N) 9,1,2                      ; test N == 0
1  IFN=0
  GOTO 9
2  IF(N-1) 9,3,4                  ; test N == 1
3  IFN=1
  GOTO 9
4  IFNM1=0
  IFN=1
  DO 5 I=2,N                      ; start loop
  IFNM2=IFNM1
  IFNM1=IFN
5  IFN=IFNM1+IFNM2                ; end loop
9  IFIBO=IFN
  END
```

https://rosettacode.org/wiki/Fibonacci_sequence#FORTRAN_IV

CODE IMPÉRATIF : C

Calcul de la suite de Fibonacci en C

```
long long int fibo(int n) {
    int fnow = 0, fnext = 1, tempf;
    while (--n > 0) {
        tempf = fnow + fnext;
        fnow = fnext;
        fnext = tempf;
    }
    return fnext;
}
```

https://rosettacode.org/wiki/Fibonacci_sequence#Iterative_13

CODE IMPÉRATIF : PYTHON

Calcul de la suite de Fibonacci en Python

```
def fibo(n):
    if n <= 1:
        return n
    fibPr = 1
    fib = 1
    for num in range(2, n):
        fibPr, fib = fib, fib + fibPr
    return fib
```

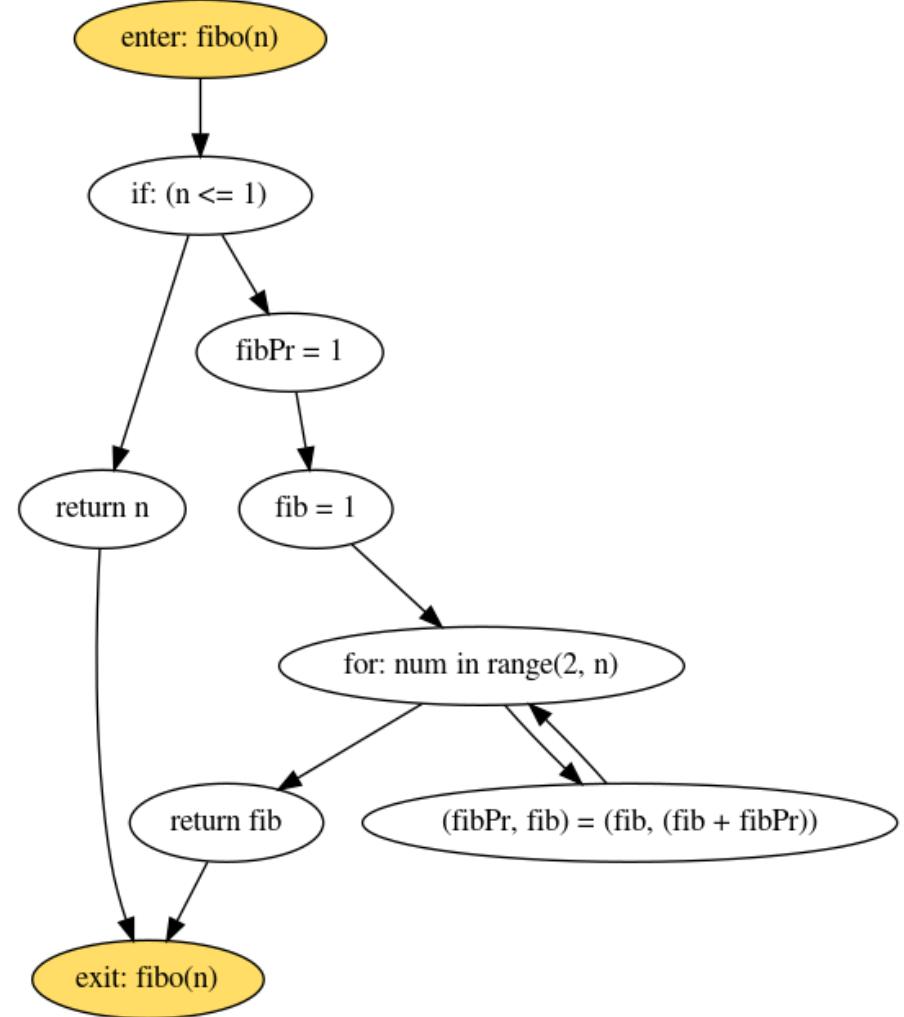
FLOT DE CONTRÔLE

- La programmation structurée encourage l'utilisation de **structures de contrôle** pour organiser le code :
 - branchements (**Python if**),
 - boucles (**Python for, while**),
 - blocs de code (**begin, end**).
- ... et leur arrangement à discrédition dans des fonctions.
- L'idée est d'organiser le **flot de contrôle**, à savoir l'agencement des instructions entre elles.
 - problème des sauts (**goto, jumps**)

GRAPHE DE FLOT DE CONTRÔLE

en Python

```
def fibo(n):
    if n <= 1:
        return n
    fibPr = 1
    fib = 1
    for num in range(2, n):
        fibPr, fib = fib, fib + fibPr
    return fib
```



TRANSITION VERS LE MODULAIRE

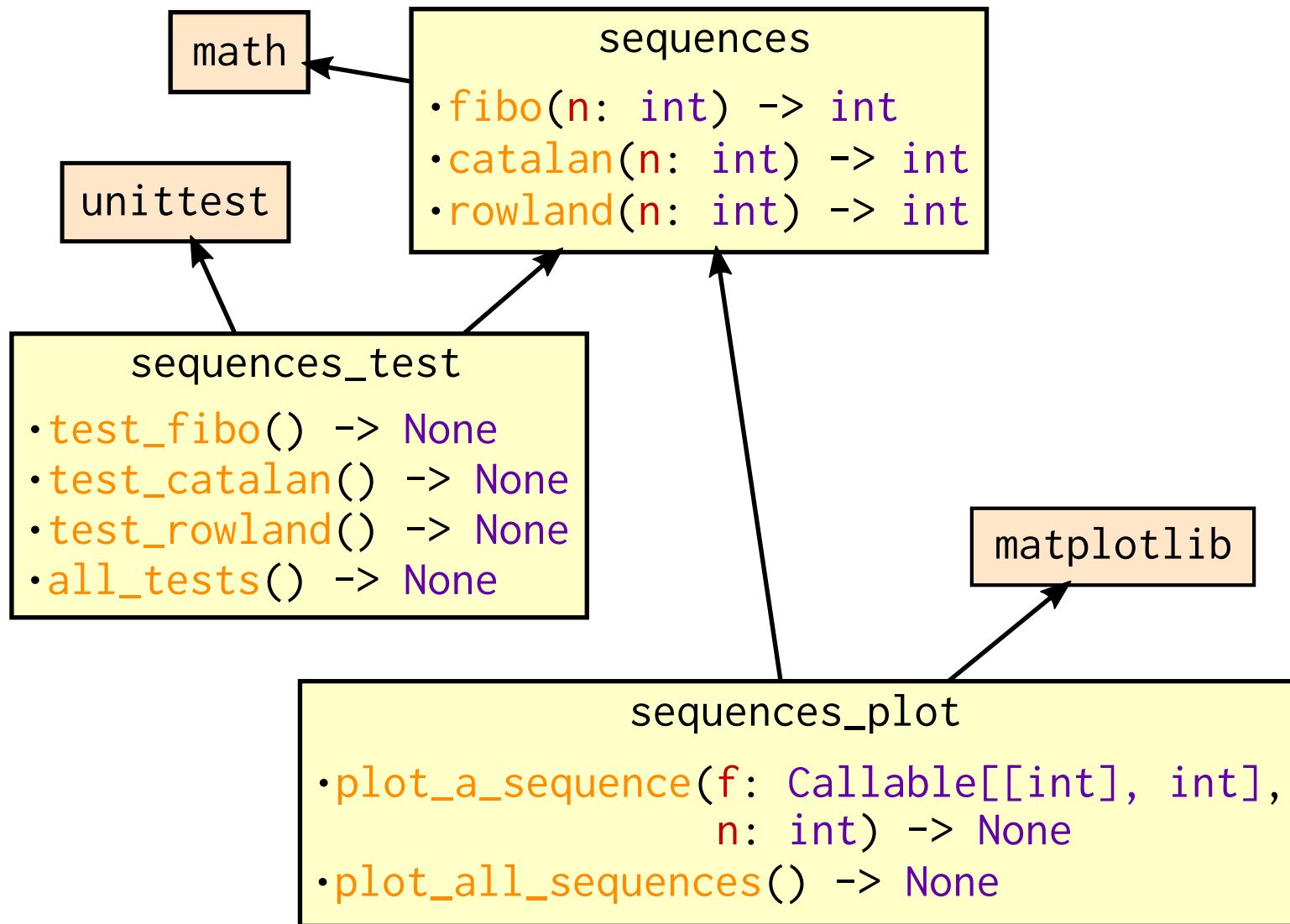
- Pour structurer ces ensembles d'instructions, il est naturel d'arranger le code en ensembles de fonctions.
- Cet arrangement est une première forme d'organisation **hiérarchique** du code.
- La généralisation de cette idée est la notion d'**architecture logicielle** : un ensemble de modèles et techniques pour organiser des composants logiciels de manière efficace.
- Exemple : la **programmation modulaire**.

PROGRAMMATION MODULAIRE

- Brique élémentaire : le **module**
- Idée générale :
 - le code est réparti dans des composants séparés,
 - ces modules sont reliés par des liens de dépendance,
 - chaque module peut être remplacé aisément.
- Qualités : les dépendances entre modules doivent être **faibles**, les dépendances à l'intérieur peuvent être **fortes**
- Représentant historique : CLU (1974)
- Exemple emblématique : **OCaml** (1996, v4.10 : 2020)

EXEMPLE DE DÉCOMPOSITION MODULAIRE

- Partons du code calculant la suite de Fibonacci.
- Ce code peut être **étendu** de diverses manières :
 - en ajoutant le code d'autres suites connues :
catalan, rowland ...
 - en ajoutant des tests de validation pour chaque suite :
test_fibo, test_catalan ...
 - en ajoutant un afficheur générique de suites :
plot_a_sequence, plot_all_sequences ...
- Réfléchissons à une décomposition du code arrangeant ces ajouts selon leur nature.



CODE MODULAIRE : PYTHON

sequences.py

```
import math

def fibo(n: int) -> int:
    fibPr = 0
    fib = 1
    for num in range(1, n+1):
        fibPr, fib = fib, fib+fibPr
    return fibPr

def catalan(n: int) -> int:
    ...
def rowland(n: int) -> int:
    ...
```

sequences_test.py

```
import unittest

def test_fibo() -> None: ...
def test_catalan() -> None: ...
def test_rowland() -> None: ...
```

sequences_plot.py

```
import matplotlib

def plot_a_sequence(f: Callable[[int],
                                 int],
                     n: int) -> None: ...

def plot_all_sequences() -> None: ...
```

Remarque : les indications de type sont optionnelles en **Python** et disponibles uniquement depuis la version 3.5

CODE MODULAIRE : OCAML

sequences.ml

```
module type SEQUENCE = sig
  val fibo : int -> int
  val catalan : int -> int
  val rowland : int -> int
end

module S : SEQUENCE = struct
  let fibo(n) =
    let fibP = ref (0,1) in
    for i = 1 to n do
      let (u,v) = !fibP in
      fibP := (v, u+v)
    done;
    fst !fibP
end
```

sequences_test.ml

```
module type SEQUENCE_TEST = sig
  val test_fibo : unit -> bool
  val test_catalan : unit -> bool
  val test_rowland : unit -> bool
end
```

sequences_plot.ml

```
module type SEQUENCE_PLOT = sig
  val plot_a_sequence : (int -> int)
                        -> int -> unit
  val plot_all_seqs : unit -> unit
end
```

- Distinction est faite entre l'**interface** et l'**implémentation**.

QUELQUES REMARQUES ...

- Chaque module met ainsi à disposition une **interface** :
 - la liste des éléments à l'intérieur (fonctions, valeurs...),
 - le cas échéant une forme de spécification plus précise (types, documentation...).
- Et contient une **implémentation** de ces éléments :
 - le code de chacun des éléments dans le modules,
 - le cas échéant, des éléments internes servant à implémenter les autres.

ALORS, C'EST QUOI UN MODULE ?

- La forme que prennent les modules varie selon les langages :
 - en **Python**, un module s'appelle bien un module,
<https://docs.python.org/fr/3/tutorial/modules.html>
 - en **C**, les modules n'existent pas concrètement, mais sont simulables avec des séparations par fichiers,
 - en **Java**, plusieurs niveaux de modules coexistent, à travers les paquetages et les classes,
 - en **JavaScript**, une technique (obsolète) consiste à encapsuler les modules à l'aide de fonctions.

QUALITÉS DE GÉNIE LOGICIEL

Quels sont les objectifs que l'on vise lorsqu'on écrit du code ?

1. des qualités de **fonctionnalité** : le code produit-il les comportements attendus ? avec quel degré de certitude ?
2. des qualités d'**évolutivité** : peut-on faire évoluer le code dans le temps et ainsi l'ensemble des comportements attendus ?

QUALITÉS DE FONCTIONNALITÉ

CORRECTION

- Qualité du code consistant à vérifier une spécification :
 - le code réalise t'il les calculs demandés ?
 - le code termine t'il ses calculs sans erreur ?
 - le code utilise t'il une quantité de ressources (temps, mémoire) raisonnables ?
- Concepts : vérification, fiabilité, sûreté, validation ...
- Outils : spécifications, types, tests, modèles formels ...

QUALITÉS D'ÉVOLUTIVITÉ

MODULARITÉ

- Qualité de découpage du code en composants distincts ayant des dépendances réduites entre eux
- Concepts : cohésion, couplage

ABSTRACTION

- Qualité d'un composant à n'exposer qu'une interface minimale pour interagir avec d'autres composants
- Concepts : encapsulation, distinction public/privé ...

POUR QUELQUES QUALITÉS DE PLUS...

- Combinées, les deux dernières propriétés permettent d'envisager de nombreuses techniques :
 - construire des composants fortement réutilisables (bibliothèques, frameworks ...),
 - construire des composants fortement indépendants,
 - remplacer des composants par d'autres proposant la même interface (résolution de bugs, améliorations ...).
- Concepts : généricité, réutilisabilité, séparation des responsabilités ...

TYPE ABSTRAIT DE DONNÉES

- Un **type abstrait de données** (aussi appelé TAD) consiste en :
 1. un **ensemble de valeurs** possibles (aussi appelé type)
 2. un **ensemble d'opérations** agissant sur ces valeurs
- Il s'agit d'un exemple classique de composant modulaire.

EXEMPLE DE TAD : LE LIVRE

1. L'ensemble des livres possibles : encyclopédie, bande dessinée, journal, prospectus ...
2. Un ensemble d'opérations pour manipuler les livres :

```
open   : book -> book
        # à partir d'un livre, produit un livre ouvert
close  : book -> book
        # à partir d'un livre, produit un livre fermé
pages  : book -> [number]
        # à partir d'un livre, produit la liste de ses pages
read   : (book * number) -> string
        # à partir d'un livre ouvert et d'un numéro de page
        # produit le texte sur une page
```

- ⇒ Abstraction : les fonctions **open**, **close** ... permettent de manier n'importe quel livre, sans connaître ses particularités.
- ⇒ Modularité : l'implémentation du livre et son utilisation sont clairement séparées.

EXAMPLE DE TAD : sequence EN PYTHON

1. L'ensemble des suites possibles :

`list ([]), [1, 2, 3], ['a', 'b'] ...), tuple, range ...`

2. Un ensemble d'opérations pour manipuler les suites :

```
__contains__ : (list * any) -> bool
    # à partir d'une liste `l` et d'une valeur `x`,
    # produit un booléen disant si `x` est dans `l`
    # Notation : `x in l`
__add__      : (list * list) -> list
    # à partir de deux listes `l1` et `l2`,
    # produit une liste contenant les valeurs de `l1`
    # suivies des valeurs de `l2`
    # Notation : `l1 + l2`
__getitem__  : (list * int) -> any
    # à partir d'une liste `l` et d'un indice `i`,
    # produit le ième élément de `l`
    # Notation : `l[i]`
```

Cf. <https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>

À SUIVRE ...

Les parties suivantes de ce cours explorent plus précisément :

- Deux paradigmes :
 - le paradigme **objet**,
 - et le paradigme **fonctionnel**.
- Trois types abstraits de données :
 - les séquences (**listes, piles, files**),
 - les **arbres**,
 - et les **graphes**.

LE PARADIGME OBJET

- Brique élémentaire : l'**objet** (la **classe**)
- Idée générale :
 - un objet est une entité rassemblant des données et des traitements sur ces données;
 - les objets échangent entre eux à travers des messages;
 - la décomposition en objets/classes structure le code.
- Représentants historiques : **Simula** (1967), **Smalltalk** (1971)
- Exemple emblématique : **Java** (1996, Java SE 14 date de 2020)

LE PARADIGME OBJET

Qu'est ce qu'un objet ?

~~Comment programmer en objet ?~~

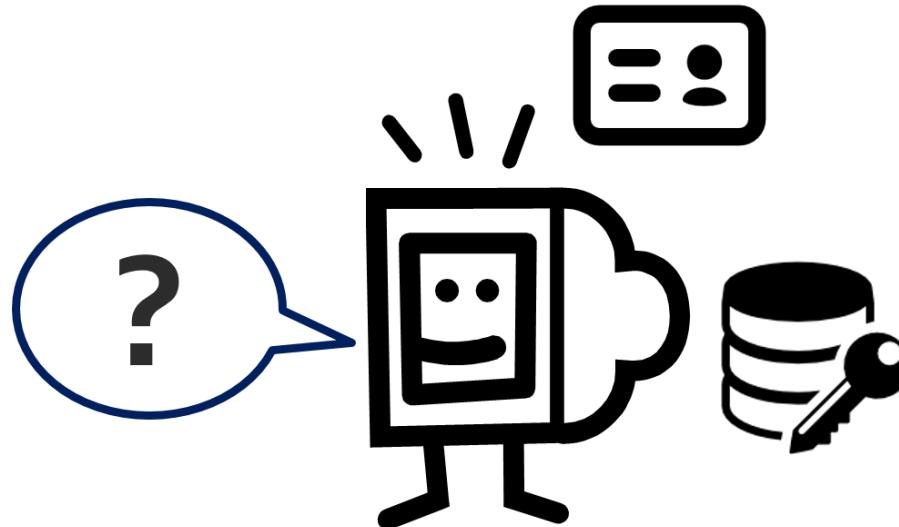
~~Architecture Orientée Objet ?~~

UNE ENTITÉ À PART ENTIÈRE

- Propose **plusieurs traitements**
 - Possède **ses données (son état)**
- => Un objet existe **à l'exécution**

DÉFINITION : OBJET

1. Une **identité** unique
2. Des **données** propres
3. Des **traitements** dont il est responsable

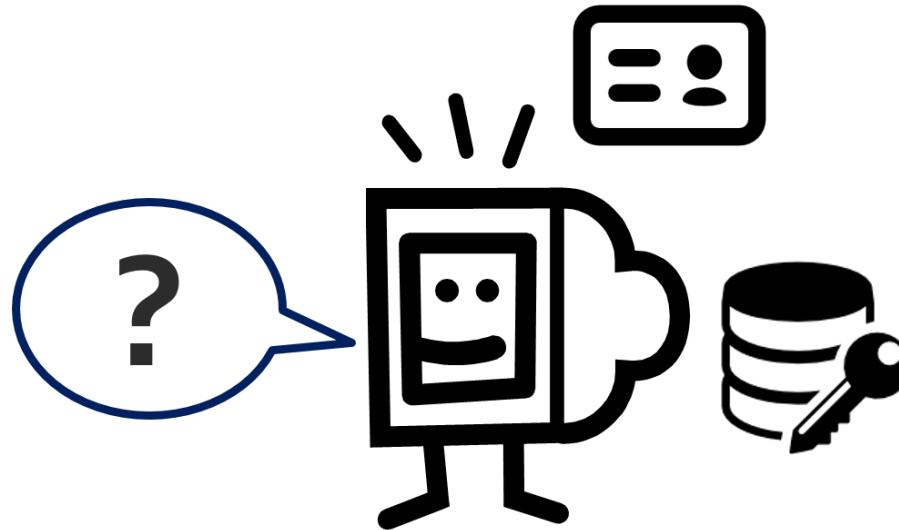


EXEMPLE : LA PILE DE CARACTÈRES

1. Identité : **Pile_1**
2. Données : Une liste de caractères (vide au début)
3. Traitements
 1. Empiler un caractère
 2. Dépiler le dernier caractère
 3. Savoir si la pile est vide

UN OBJET SANS IDENTITÉ ?

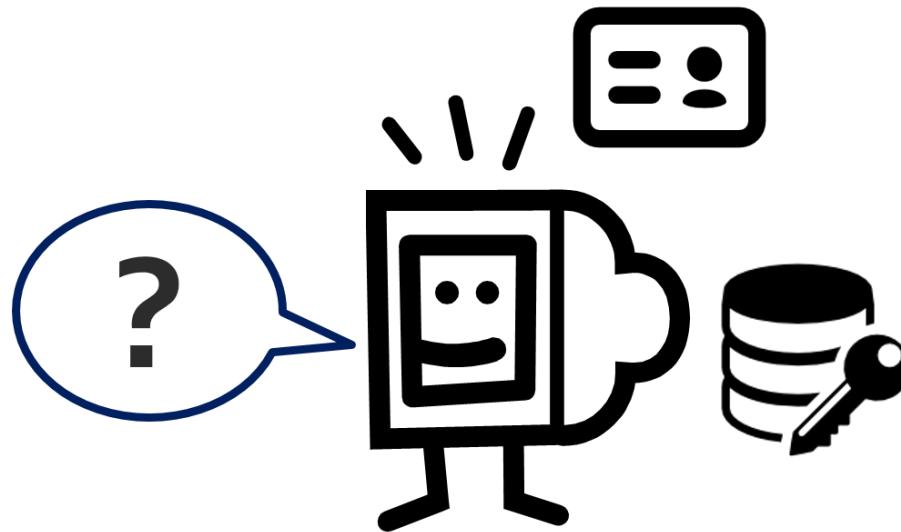
1. Identité
2. Données
3. Traitements



C'est possible, objet **autonome** et **anonyme**

UN OBJET SANS DONNÉES ?

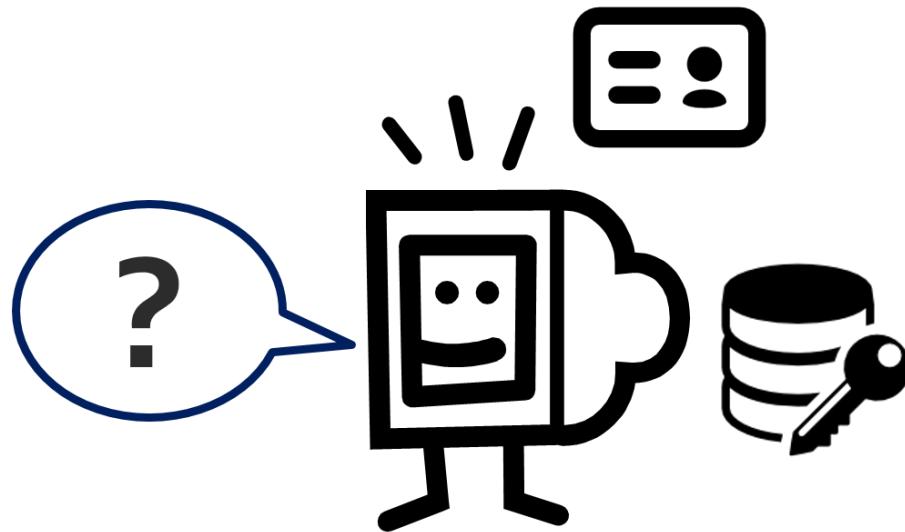
1. Identité
2. ~~Données~~
3. Traitements



C'est possible, objet **stateless** (sans état)

UN OBJET SANS TRAITEMENTS ?

1. Identité
2. Données
3. ~~Traitements~~



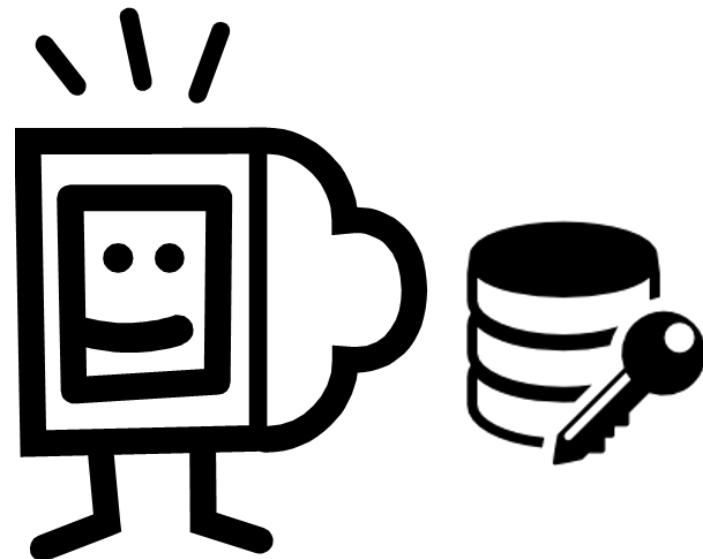
Plus rare (**objet données**), pas très cohérent avec le modèle.

ENCAPSULATION

Un objet **protège ses données**

Un objet **peut laisser d'autres objets lire** ses données

Un objet est **le seul à modifier** ses données

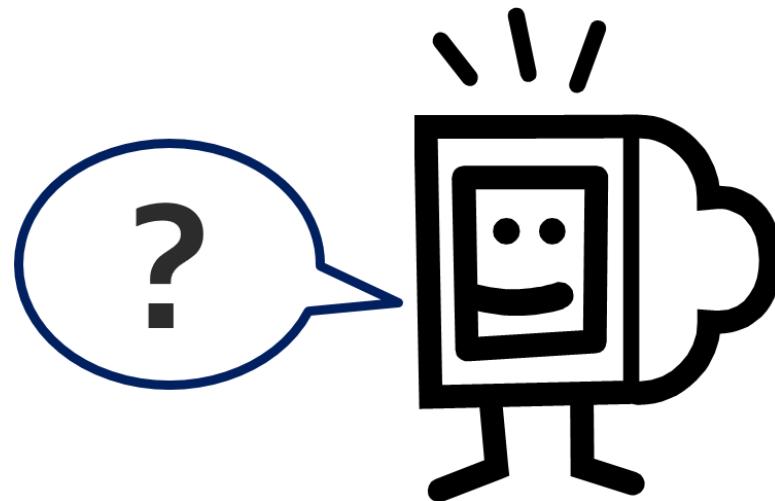


RESPONSABILITÉ

Un objet est **responsable des traitements qu'il propose**

Il a donc toutes les données nécessaires et suffisantes

Il peut utiliser (les traitements) d'autres objets



COHÉRENCE

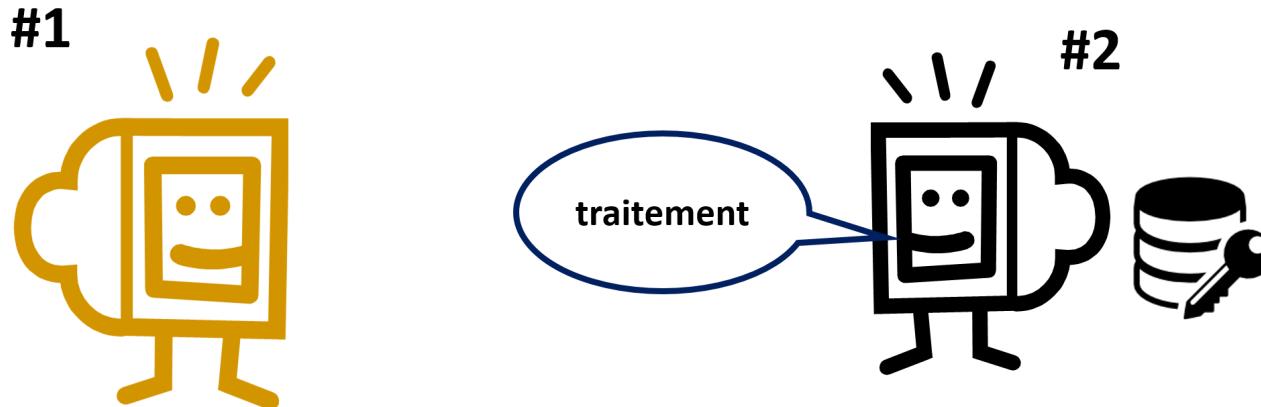
Un objet propose **peu** de traitements (liés entre eux)

Les traitements **partagent** des données

⇒ Couper un objet en deux ne devrait pas avoir de sens

COMMUNICATION

- Les objets communiquent par **échange de messages**
 - Celui qui envoie le message
 - Celui qui reçoit le message



ENVOYER UN MESSAGE

- Connaître l'identifiant du destinataire
- Préciser quel traitement il demande
- Fournir les paramètres nécessaires au traitement

RECEVOIR UN MESSAGE

- Obligation de réponse
- (Envoyeur inconnu)

SYNCHRONE & MONOTHREAD

Un seul object actif

L'objet actif peut envoyer

L'envoyeur n'est plus actif

Le receveur devient actif

L'envoyeur redevient actif avec la réponse

COUPLAGE

Un objet communique avec **peu** d'autres objets

Chaque traitement fait intervenir **peu** d'autres objets

⇒ Limiter les chaînes et interdir les boucles !

CRÉATION D'OBJET

Tout **objet** peut créer des **objets** !

L'objet créateur **connaît l'id** de l'objet créé.

Il peut alors **donner les ids** des objets qu'il a créé

SUPPRESSION

Avec un **garbage collector** (ramasse-miettes) : les objets **inaccessibles** sont supprimés

Sans garbage collector : les objets doivent **se supprimer**

APPLICATION OO

Plusieurs objets qui communiquent

Certains objets communiquent avec l'extérieur

⇒ Une configuration initiale

CONCEPTION OBJET

Quels objets ?

Quelles cohérence ?

Quels couplage ?

Quelle dynamicité ?

LE PARADIGME OBJET

~~Qu'est ce qu'un objet ?~~

Comment programmer en objet ?

~~Architecture Orientée Objet ?~~

PEUT-ON PROGRAMMER DES OBJETS ?

Le programme crée l'objet

Lui donne ses données

Lui donne ses traitements

⇒ Programmer plusieurs objets : programmer des classes !

LA CLASSE : DÉFINITION

Une classe a un **nom** unique dans le programme.

Elle définit les données associées à ses objets: les **propriétés**

Elle définit les traitements des objets: les **méthodes**

⇒ Un objet est instance d'une classe !

PROPRIÉTÉ

Un **nom** unique parmi les propriétés

(Un **type** qui définit la structure des données)

(Une valeur par défaut donnée lors de la création)

MÉTHODE

Une **signature**: nom, paramètres entrée et sortie

Un **corps**: le code du traitement (**this** ou **self** référence l'objet lui-même)

(Des exceptions qui peuvent être levées)

EXAMPLE DE CLASSE

nom: Pile

données: liste de caractères

méthodes: empiler, dépiler

⇒ Une pile: p = **Pile()**

CLASSE ET ENCAPSULATION

(Rappel: l'objet doit encapsuler ses données)

La classe précise les règles d'accès aux champs

public : accès total

private : accès interdit

VALEURS ET GETTER / SETTER

Affectation à la création

Lecture (`get_field`)

Écriture (`set_field`)

COUPLAGE ET CLASSE

Relations entre classes

Durée de vie

Lien avec les traitements

EXAMPLE : UE, ETUDIANT ET EXAMEN

Une UE a deux examens

Un étudiant est inscrit à plusieurs UE

⇒ Calcul de la moyenne pour une UE

COHÉRENCE

Les méthodes **utilisent** les données

Une classe devrait être **insécable**

LE PARADIGME OBJET (OPTIONNEL)

~~Qu'est ce qu'un objet ?~~

~~Comment programmer en objet ?~~

Architecture Orientée Objet ?

OBJECTIFS

Améliorer la maintenance

Améliorer la performance

Améliorer la sécurité

...

REUSE

Réutiliser du code

Coder pour être réutilisé

⇒ Héritage / Interface

HÉRITAGE

Une classe **hérite** d'une autre classe

Relation de **conformité** / **substituabilité** entre les objets

⇒ Les objets instances de la sous-classe ont les données et les traitements définis dans la super-classe.

QUAND HÉRITER ?

Réduire la redondance de son code

Étendre une classe pour la réutiliser

Faire un template pour un autre développeur

MÉTHODE & HÉRITAGE

Surcharge : la sous-classe définit une méthode avec une signature similaire (au moins le même nom)

Polymorphisme : la sous-classe définit une méthode avec la même signature mais pas le même corps.

L'INTERFACE

Définition d'un contrat d'usage

Limite la dépendance de l'utilisateur

⇒ Contrat appelant / appelé sur la signature

QUAND PASSER PAR UNE INTERFACE

Masquer l'implémentation

Inverser la dépendance

LE PARADIGME OBJET

Qu'est ce qu'un objet ?

Comment programmer en objet ?

Architecture Orientée Objet ?

CONCLUSION

Pensez Object (id, données, traitement)

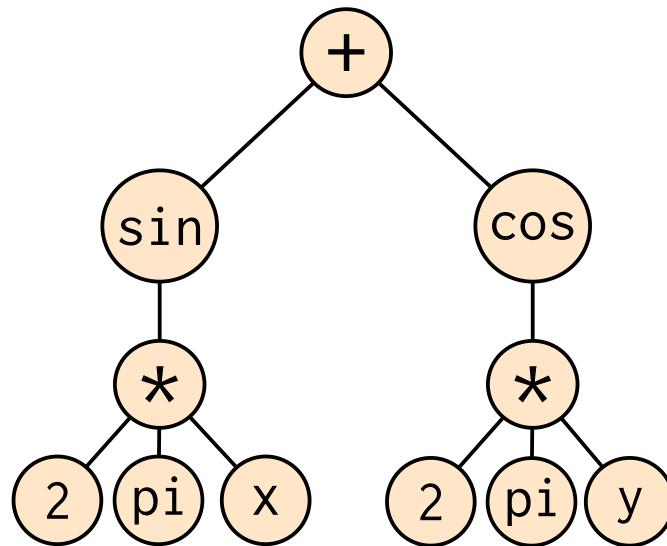
Coder les classes

Améliorer la conception

LE PARADIGME FONCTIONNEL

- Brique élémentaire : l'**expression**
- Idée générale :
 - partir d'un ensemble de valeurs (nombres, listes ...)
 - utiliser un ensemble de moyens de les composer en expressions complexes (opérateurs, fonctions ...)
- Exemple simpliste : **sin(2*pi*x) + cos(2*pi*y)**
- Représentants historiques : Lisp (1958), Scheme (1975)
- Exemple emblématique : **Haskell** (1990, dernière norme: 2010)

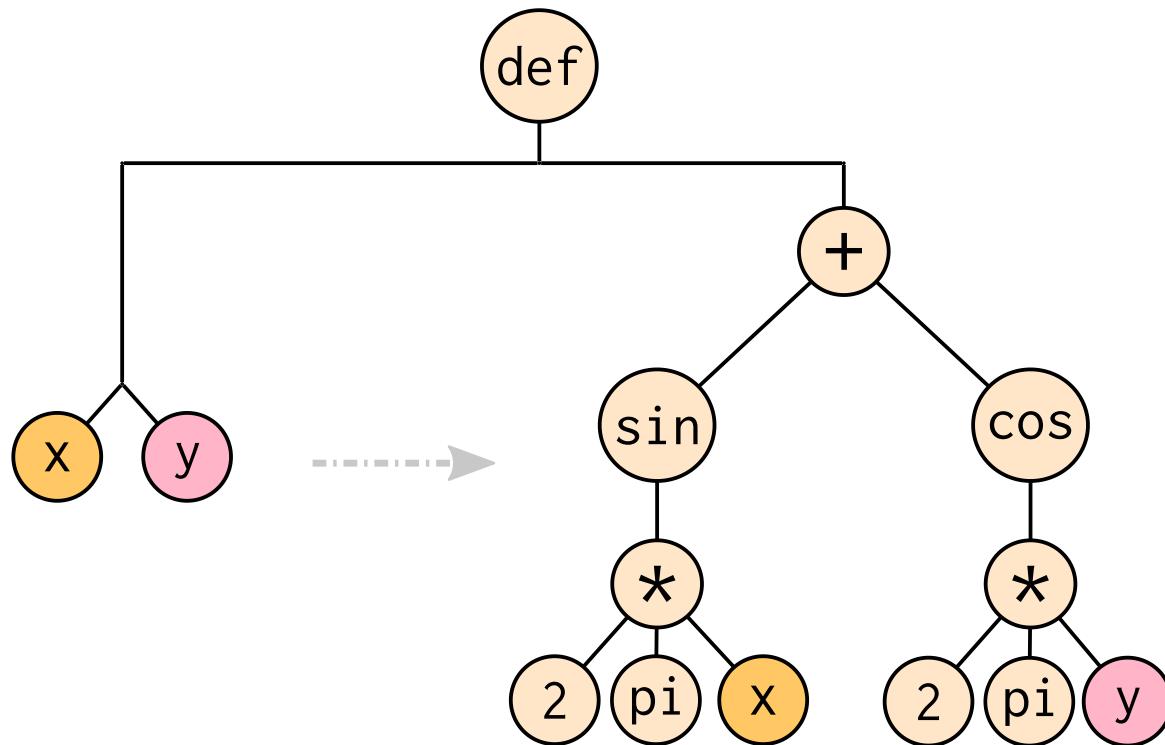
QU'EST-CE QU'UNE EXPRESSION ?

$$\sin(2\pi x) + \cos(2\pi y)$$


- L'expression est la représentation arborescente d'un calcul.
- **Évaluer** une expression permet d'obtenir le résultat du calcul.

QU'EST-CE QU'UNE FONCTION ALORS ?

```
def f(x, y):  
    return sin(2*pi*x) + cos(2*pi*y)
```

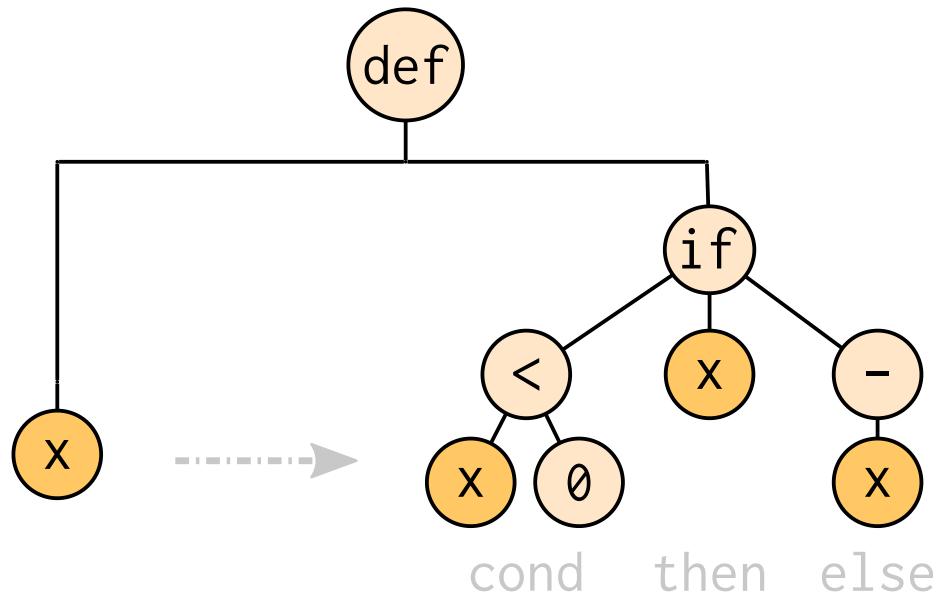


Un moyen de construire des expressions plus complexes.

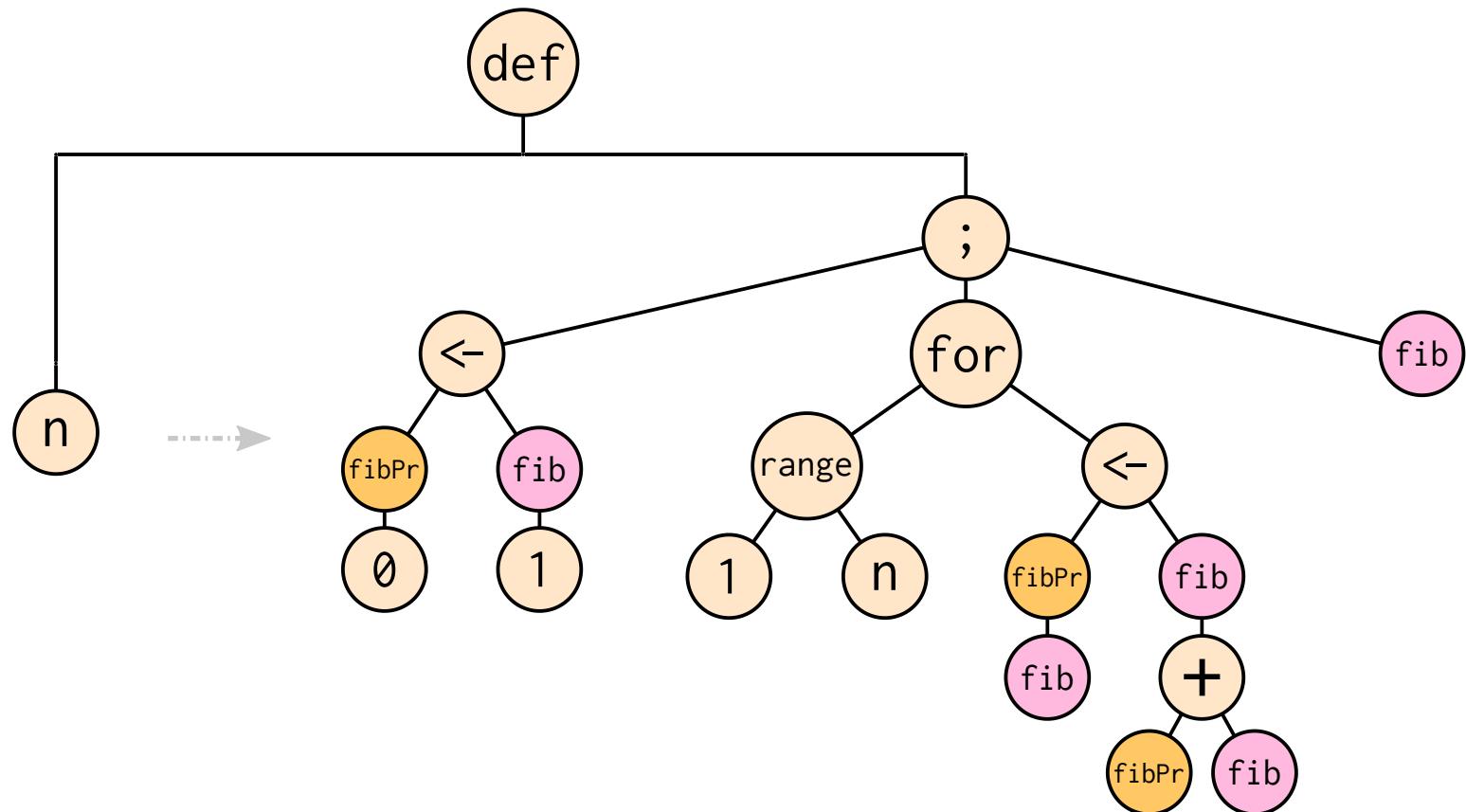
UN IF-THEN-ELSE ?

```
def abs(x):  
    if (x > 0):  
        return x  
    else:  
        return -x
```

```
def abs(x):  
    return (  
        x if (x > 0) else -x  
    )
```



```
def fibo(n):
    fibPr, fib = 0, 1
    for num in range(1, n):
        fibPr, fib = fib, fib + fibPr
    return fib
```



PROPRIÉTÉS

La programmation fonctionnelle s'appuie sur deux principes :

- **Pureté** : le résultat de l'évaluation d'une fonction ne dépend que de la valeur de ses paramètres, pas de facteurs externes;
- **Fonctions de 1ère classe** : les fonctions sont les briques de base pour composer les expressions; elles peuvent apparaître comme paramètres, retours ou données d'un programme.

PURETÉ (1/2)

- Idée : le résultat de l'évaluation d'une fonction ne dépend que de la valeur de ses paramètres, sans aucun facteur externe.
- Revient à considérer les fonctions comme leurs homologues mathématiques : des **applications**.
- Dans l'exemple suivant, le résultat de l'appel à **f()** dépend du moment où on l'appelle :

```
i = 0

def f():
    global i
    i = i + 1
    return i

f() - f() # (1 - 2) or (2 - 1) ?
```

- En **Python** : **-1**
- En **c** : **1** ou **-1**
- En **ocaml** : **1** ou **-1**

PURETÉ (2/2)

- Une fonction "impure" produisant ou dépendant de facteurs externes est dite réaliser des **effets de bords**.
- La présence de **variables** ou de **boucles**, et plus généralement la notion d'**état** (**impératif**) sont propices aux effets de bords.
⇒ Idée : manipuler et transformer des objets **constants**
- Exemples d'effets de bord :
 - lecture/écriture dans un fichier, une base de données
 - dépendance à un générateur aléatoire,
 - dépendance à une mesure externe (heure, lieu ...)
- Il n'est pas toujours simple d'écrire des fonctions pures, il faut parfois transiger.

PURETÉ : UN EXEMPLE (1/2)

- Transformation de code impératif → fonctionnel

Avec effet de bord (imp)

```
cpt = 0          # global state

def count_calls():
    global cpt
    cpt += 1
    print("calls={}".format(cpt))

count_calls() # calls=1
count_calls() # calls=2
```

Sans effet de bord (fonc)

```
def count_calls(cpt):
    new_cpt = cpt + 1
    return new_cpt

cpt1 = 0
cpt2 = count_calls(cpt1)
print("calls={}".format(cpt2)) # calls=1
cpt3 = count_calls(cpt2)
print("calls={}".format(cpt3)) # calls=2
```

- Idée : transformer le **cpt** global en plusieurs intermédiaires.
- Chaque intermédiaire peut être considéré comme constant.

PURETÉ : UN EXEMPLE (2/2)

- Transformation de code impératif → fonctionnel

Avec effet de bord (imp)

```
global_board = Board() # global state

def play(board, color):
    m = board.get_move(color)
    # Modify board 'in place'
    board.moves.append(m)

play(global_board, Color.WHITE)
play(global_board, Color.BLACK)
```

Sans effet de bord (fonc)

```
def play(board, color):
    m = board.get_move(color)
    # Return new independent board
    return Board(moves = \
                 board.moves + [m])

board1 = Board()
board2 = play(board1, Color.WHITE)
board3 = play(board2, Color.BLACK)
```

- La version sans effet de bord permet facilement de :
 - conserver les états intermédiaires,
 - jouer des coups et revenir en arrière,
 - envisager la construction de stratégies ...

CODER SANS EFFETS DE BORDS

- Comment organiser un calcul (non trivial) sans variables ?

```
play(  
    play(  
        play(  
            play(board,  
                  Color.WHITE),  
                  Color.BLACK),  
                  Color.WHITE),  
                  Color.BLACK)
```

- Solution possible : écrire des fonctions récursives.

```
def play_rec(board, n):  
    if (n == 0):  
        return board  
    else:  
        next_board = play(play(board, Color.WHITE), Color.BLACK)  
        return play_rec(next_board, n-1)
```

INTÉRÊTS DE LA PURETÉ

- **Portabilité** de la fonction : indépendance du moment et de lieu de l'appel.
- Facilitation des **tests** : pas de nécessité de préparer un contexte particulier à chaque fois.
- **Parallélisation** possible du code : des appels indépendants peuvent être faits sur des machines différentes.

Il ne s'agit pas d'être dogmatique : on peut mélanger les styles purs et impurs, si on prend soin des effets de bords.

RÉCURSIVITÉ

- Une fonction est **récursive** si son code fait appel à elle-même.

Version impérative

```
def fibo(n):  
    fibPr, fib = 0, 1  
    for num in range(1, n+1):  
        fibPr, fib = fib, fib + fibPr  
    return fibPr
```

Version récursive

```
def fibo(n):  
    if (n <= 1):  
        return 1  
    else:  
        return fibo(n-1) + fibo(n-2)
```

- La version récursive repose sur la définition mathématique :

$$\begin{cases} f_0 = 1 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad \text{si } n \geq 2 \end{cases}$$

RÉCURSIVITÉ : CAS D'APPLICATION

- Quand écrire des fonctions récursives ?
 - pour des calculs mettant en jeu des structures de données récursives (listes, arbres ...)
 - pour des algorithmes du type "diviser pour régner"
- Toute boucle est convertible en appel récursif et vice versa.

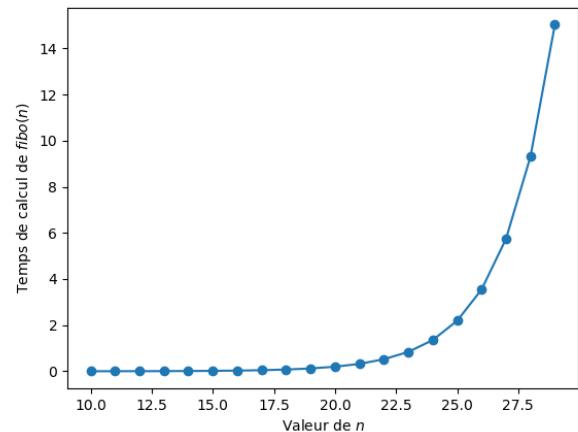
```
def fun_with_loop(n):
    res = 0
    for i in range(n):
        res += i
    return res
```

```
def fun_with_rec(n):
    def f_rec(res, i):
        if (i < n):
            return f_rec(res+i, i+1)
        else:
            return res
    return f_rec(0, 0)
```

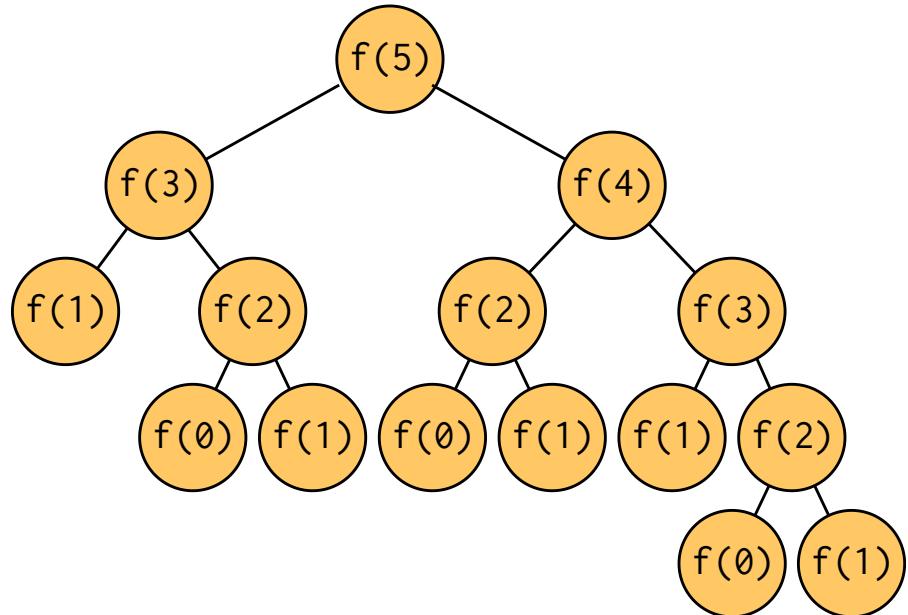
RÉCURSIVITÉ : DÉBORDEMENTS DE PILE

- Problème : les appels récursifs parfois trop nombreux

```
def fibo(n):  
    if (n <= 1):  
        return n  
    else:  
        return fibo(n-1) + fibo(n-2)
```



Arbre des appels pour **fibo(5)**

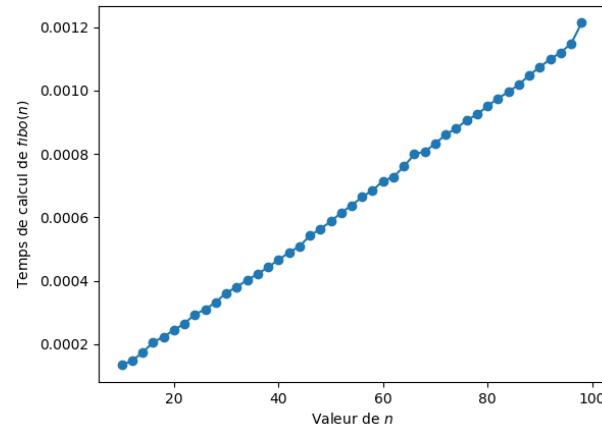


- Nombre d'appels exponentiel en **n** \Rightarrow débordements de pile.

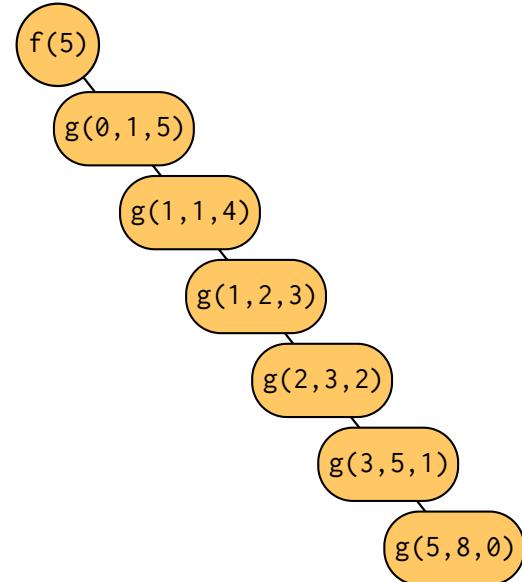
RÉCURSIVITÉ TERMINALE

- En pratique, il s'agit de faire attention en construisant **fibo**:

```
def fibo(n):  
    def fib_rec(a, b, n):  
        if n == 0:  
            return a  
        else:  
            return fib_rec(b, a+b, n-1)  
    return fib_rec(0, 1, n)
```



Arbre des appels pour **fibo(5)**



Version **récursive-terminale**

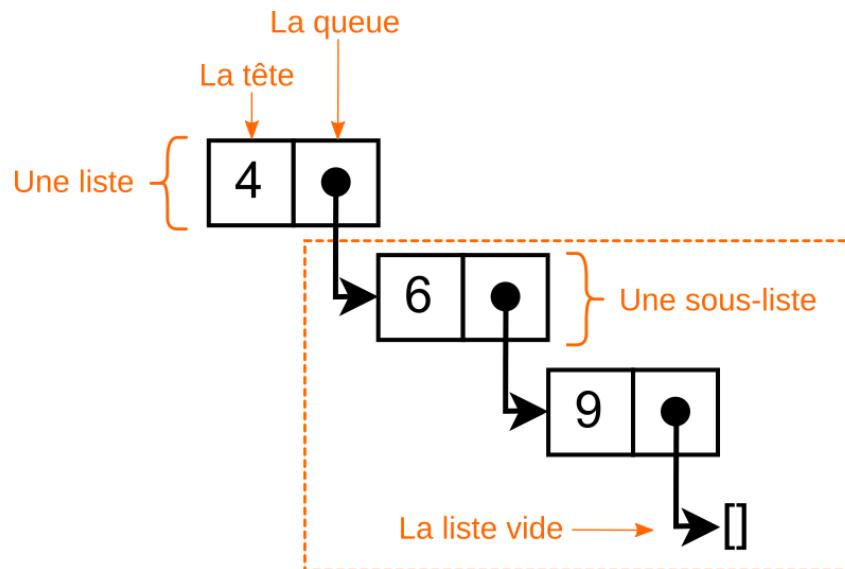
- Nombre d'appels linéaire en **n** ✓
- Pas de duplication des calculs ✓

UNE STRUCTURE RÉCURSIVE : LA LISTE

Les **listes** sont un type de données construit de manière **récursive**.

Une liste peut prendre deux formes différentes :

- soit elle est vide (appelée par la suite **liste vide**),
- soit elle est constituée d'une valeur initiale (la **tête**) et d'une autre liste (la **queue**).



STRUCTURE RÉCURSIVE, ALGOS RÉCURSIFS

- Lorsqu'un type de données est conçu de manière récursive, les algorithmes qui l'utilisent sont naturellement récursifs.

```
def cons(x, l):
    return { "hd": x, "tl": l }

def head(l): return l["hd"]
def tail(l): return l["tl"]

def empty(): return cons(None, None)
def is_empty(l):
    return (head(l) is None) and \
           (tail(l) is None)
```

```
def length(l):
    if is_empty(l):
        return 0
    else:
        return 1 + length(tail(l))
```

```
l = cons(1, cons(2, cons(3, empty())))
length(l)          # -> 3
length(tail(l))   # -> 2
length(empty())    # -> 0
```

- Structure canonique des algorithmes :

```
if is_empty(l):
    # Do something for the empty list
else:
    # Now l is not empty, do something with its head and tail
```

RÉCURSIVITÉ DANS D'AUTRES LANGAGES

En Scheme

```
(define (length lis)
  (cond ((null? lis)
          0) ;; liste vide
        (else
          (+ 1 (length (cdr lis)))))) ;; liste non vide
```

En OCaml

```
let rec length l = match l with
| []      -> 0 (* liste vide *)
| _ :: tl -> 1 + length tl (* liste non vide *)
```

En Haskell

```
length :: [a] -> Integer           -- type de la fonction
length []     = 0                     -- liste vide
length (_:xs) = 1 + length xs       -- liste non vide
```

FONCTIONS DE 1ÈRE CLASSE

- Les fonctions sont les briques de base pour composer les expressions. Elles peuvent apparaître :
 - dans des structures de données,
 - ou comme paramètres et retours d'autres fonctions.
- Une fonction est alors une petite **unité de code**, que l'on peut créer, transmettre et utiliser à la demande.
- Exemple fondamental : les **lambda-expressions**.

```
lambda x: x+1          # <function <lambda> at 0x7>
```

en tant qu'exemple de fonction anonyme.

1ÈRE CLASSE : CRÉATION

- Construire des fonctions en **Python** :

- https://docs.python.org/3/reference/compound_stmts.html#function
- <https://docs.python.org/3/reference/expressions.html#lambda>

Version nommée

```
def func(param1, param2):  
    return param1 - param2  
  
func      # <function func at 0x7>  
func(34,23) # 11
```

Version anonyme

```
lamb = lambda param1, param2: \  
        param1 - param2  
  
lamb    # <function <lambda> at 0x7>  
lamb(34,23) # 11
```

- Caveat **Python** : le corps d'une lambda doit être une expression.
- Les fonctions peuvent alors :
 - être stockées dans des variables (comme **lamb**)
 - apparaître dans des structures : [**func**, **lamb**]

1ÈRE CLASSE : PARAMÈTRE

- Une fonction peut être paramétrée par une autre fonction.
- Exemple : un algorithme de tri paramétrée par un ordre de tri

```
def sort(l, cmp): # Generic Bubble Sort
    n, nl = len(l), list(l)
    for i in range(n):
        for j in range(n - i - 1):
            if cmp(nl[j], nl[j + 1]):
                nl[j], nl[j+1] = nl[j+1], nl[j]
    return nl
```

```
sort(range(10), lambda x,y: x>y)
# -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sort(range(10), lambda x,y: x<y)
# -> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

- Idée : déléguer une partie d'un algorithme à l'appelant.
De telles fonctions sont dites **génériques**.
- Exemples : **filter**, **map**, **min**, **max**, **sorted** (via **key**)

1ÈRE CLASSE : RETOUR

- Une fonction peut renvoyer une autre fonction :
- Exemple : un algorithme de dérivation de fonction

```
def derivate(h, f):  
    return lambda x: \  
        (f(x+h) - f(x)) / h
```

```
c = derivate(0.01, np.sin)  
c(0)      # 0.9999833334166665  
c(np.pi) # -0.9999833334166452
```

- Idée : créer un code paramétré applicable de manière différée
- Exemples :

- spécialisation par application partielle (**partial**)

```
functools.partial(derivate, 0.01)    # derivation operator
```

- contrôle de l'évaluation / évaluation paresseuse

```
timeit.timeit(lambda: fibo(100))
```

INTÉRÊTS DE LA 1ÈRE CLASSE

- Considérer les fonctions comme des valeurs ayant les mêmes possibilités d'utilisation que les entiers, les objets...
- Leur vocation : représenter des calculs paramétrés.
- Exemples d'application :
 - la représentation des données par les fonctions (cf. [characteristic](#)),
 - la parallélisation automatique des calculs (cf. [mapreduce](#)),
 - le contrôle de l'évaluation et la paresse (cf. [laziness](#)).

LISTES, PILES ET FILES

- Structures de données :

- linéaires
- séquentielles

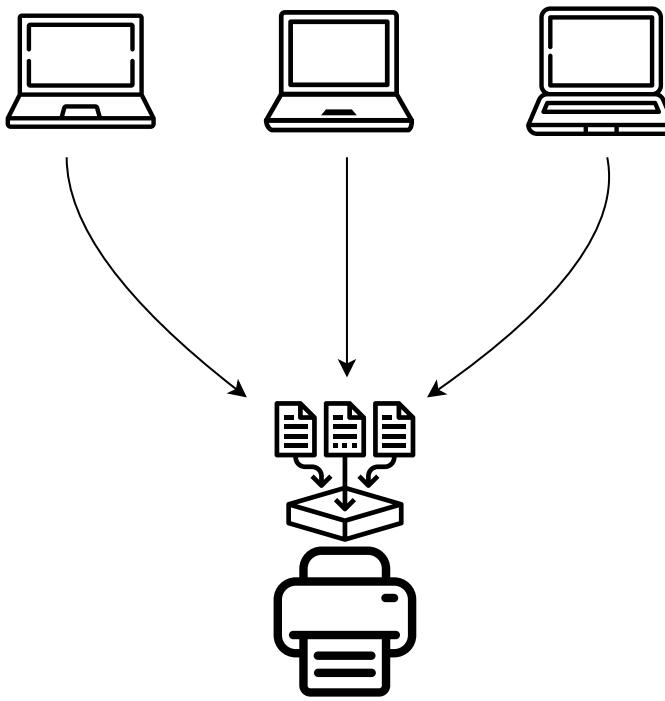
LISTES, PILES ET FILES

- Structures de données :
 - linéaires
 - séquentielles
- TADs **très proches** les uns des autres :
 - Structures **identiques**
 - Différents sur leur **fonctionnement**

LISTES, PILES ET FILES

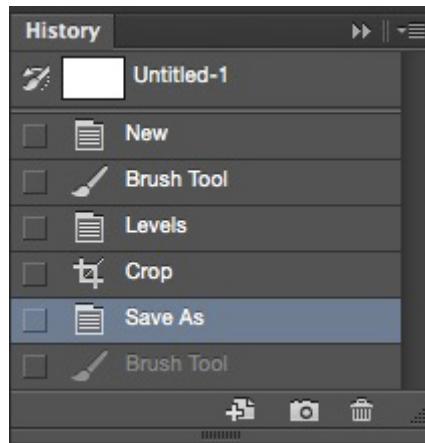
- Structures de données :
 - linéaires
 - séquentielles
- TADs **très proches** les uns des autres :
 - Structures **identiques**
 - Différents sur leur **fonctionnement**
- Largement utilisés dans les **algorithmes** :
 - Tri
 - Ordonnancement
 - Gestion mémoire (appels de fonctions)

EXEMPLE: FILE D'IMPRESSION



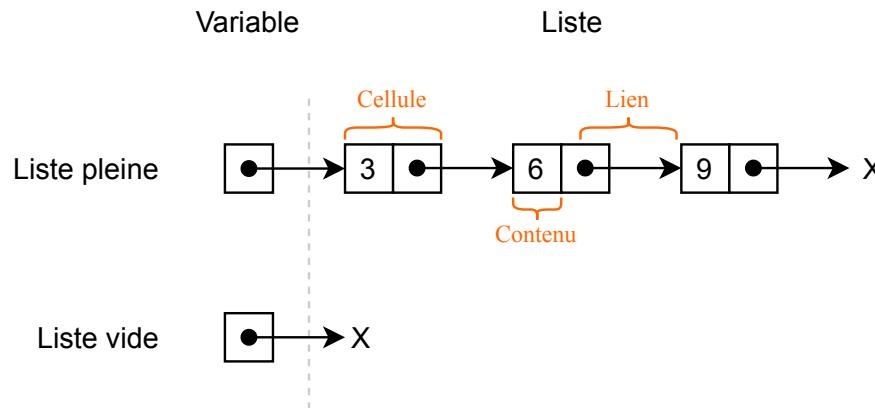
Une imprimante ne pouvant pas imprimer plusieurs documents en même temps, les requêtes doivent être stockées et organisées avant d'être traitées.

EXEMPLE: HISTORIQUE DES ACTIONS



Certaines implémentations d'annulation/rétablissement d'actions utilisent une structure sous forme de pile.

LISTE: TERMINOLOGIE



- Une **liste** est caractérisée par un ensemble de **cellules**
- Chaque **cellule** contient une valeur et un **lien** vers la cellule suivante
- Une **liste** peut être vide

DÉFINITION RÉCURSIVE

- Une **liste** c'est :
 - soit une **liste vide**, notée Δ ;
 - soit un **couple** (e, L) , appelé cellule, avec :
 - e l'étiquette de la cellule (la valeur stockée)
 - L le reste de la liste

DÉFINITION RÉCURSIVE

- Une **liste** c'est :
 - soit une **liste vide**, notée Δ ;
 - soit un **couple** (e, L) , appelé cellule, avec :
 - e l'étiquette de la cellule (la valeur stockée)
 - L le reste de la liste
- Définition **récursive** : une liste est construite à partir d'un élément et d'une sous-liste.

DÉFINITION RÉCURSIVE

- Une **liste** c'est :
 - soit une **liste vide**, notée Δ ;
 - soit un **couple** (e, L) , appelé cellule, avec :
 - e l'étiquette de la cellule (la valeur stockée)
 - L le reste de la liste
- Définition **récursive** : une liste est construite à partir d'un élément et d'une sous-liste.
- La première partie de la définition assure l'arrêt et donc la cohérence de la définition.

TYPE ABSTRAIT Liste

1. Constructeurs :

```
liste_vide : () -> Liste
  # produit la liste vide
cellule : (Etiquette * Liste) -> Liste
  # à partir d'une étiquette e et d'une liste L,
  # produit la liste (e, L)
```

TYPE ABSTRAIT Liste

1. Constructeurs :

```
liste_vide : () -> Liste
    # produit la liste vide
cellule : (Etiquette * Liste) -> Liste
    # à partir d'une étiquette e et d'une liste L,
    # produit la liste (e, L)
```

2. Sélecteurs :

```
valeur : Liste -> Etiquette
    # à partir d'une liste (e, L), produit l'étiquette e
suite : Liste -> Liste
    # à partir d'une liste (e, L), produit la liste L
```

TYPE ABSTRAIT Liste

1. Constructeurs :

```
liste_vide : () -> Liste
    # produit la liste vide
cellule : (Etiquette * Liste) -> Liste
    # à partir d'une étiquette e et d'une liste L,
    # produit la liste (e, L)
```

2. Sélecteurs :

```
valeur : Liste -> Etiquette
    # à partir d'une liste (e, L), produit l'étiquette e
suite : Liste -> Liste
    # à partir d'une liste (e, L), produit la liste L
```

3. Prédicat :

```
est_vide : Liste -> bool
    # à partir d'une liste L, produit un booléen
    # indiquant si L est la liste vide
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

RÉCURSIVE

1. Constructeurs :

```
def liste_vide():
    return []

def cellule(etiquette, liste):
    return [etiquette, liste]
```

RÉCURSIVE

1. Constructeurs :

```
def liste_vide():
    return []

def cellule(etiquette, liste):
    return [etiquette, liste]
```

2. Sélecteurs :

```
def valeur(liste):
    return liste[0]

def suite(liste):
    return liste[1]
```

RÉCURSIVE

1. Constructeurs :

```
def liste_vide():
    return []

def cellule(etiquette, liste):
    return [etiquette, liste]
```

2. Sélecteurs :

```
def valeur(liste):
    return liste[0]

def suite(liste):
    return liste[1]
```

3. Prédicat :

```
def est_vide(liste):
    return liste == liste_vide()
```

RÉCURSIVE

Exemple d'utilisation :

```
1. L1 = cellule(3, liste_vide())
2. print(L1) # [3, []]
3. print(est_vide(suite(L1))) # True
4.
5. L2 = cellule(3, cellule(6, cellule(9, liste_vide())))
6. print(L2) # [3, [6, [9, []]]]
7. print(valeur(suite(suite(L2)))) # 9
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

2. Classe Cellule

→ paradigme objet

CLASSE CELLULE

1. Liste vide représenté par **None**

2. Liste représentée par la classe suivante :

```
class Cellule:  
    liste_vide = None  
  
    def __init__(self, etiquette, liste):  
        self._valeur = etiquette  
        self._suivant = liste  
  
    def valeur(self):  
        return self._valeur  
  
    def suite(self):  
        return self._suivant  
  
    def est_vide(liste):  
        return liste is Cellule.liste_vide
```

CLASSE CELLULE

Exemple d'utilisation :

```
1. L1 = Cellule(3, Cellule.liste_vide)
2. print(L1) # <__main__.Cellule object at 0x...>
3. print(L1.valeur()) # 3
4. print(Cellule.est_vide(L1)) # False
5.
6. c1 = Cellule(9, Cellule.liste_vide)
7. c2 = Cellule(6, c1)
8. L2 = Cellule(3, c2)
9. print(L2.suite().valeur()) # 6
10. print(Cellule.est_vide(L2.suite().suite().suite()))) # True
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

2. Classe Cellule

→ paradigme objet

3. Tableau

→ paradigme impératif

TABLEAU

Liste stockée dans un tableau de taille fixe. Les opérations se font sur des cellules et non plus sur la liste entière :

```
taille_max = 8

def liste_vide():
    return None

liste = [liste_vide()] * taille_max

def cellule(etiquette, numero_cellule):
    liste[numero_cellule] = etiquette

def valeur(numero_cellule):
    return liste[numero_cellule]

def suite(numero_cellule):
    return liste[numero_cellule:]

def est_vide(numero_cellule):
    return liste[numero_cellule] == liste_vide()
```

TABLEAU

Exemple d'utilisation :

```
1. cellule(3, 0)
2.
3. print(liste) # [3, None, None, None, None, None, None, None]
4.
5. cellule(6, 1)
6. cellule(9, 2)
7.
8. print(liste) # [3, 6, 9, None, None, None, None, None]
9. print(valeur(1)) # 6
10. print(est_vide(4)) # True
```

FILES ET PILES

FILES ET PILES

- Les files et piles sont comme des sacs.

FILES ET PILES

- Les files et piles sont comme des sacs.
- Une file est dite **First-In First-Out** (FIFO) :
 - On insère un élément par la queue de la file
 - On prend un élément par le devant de la file

FILES ET PILES

- Les files et piles sont comme des sacs.
- Une file est dite **First-In First-Out** (FIFO) :
 - On insère un élément par la queue de la file
 - On prend un élément par le devant de la file
- Une pile est dite **Last-In First-Out** (LIFO) :
 - On insère un élément sur le dessus de la pile
 - On prend un élément sur le dessus de la pile

FILES ET PILES

FILES ET PILES

- Ces types abstraits de données nous permettent de :
 - Savoir si la structure est **vide**
 - **Récupérer** un élément
 - **Insérer** un élément

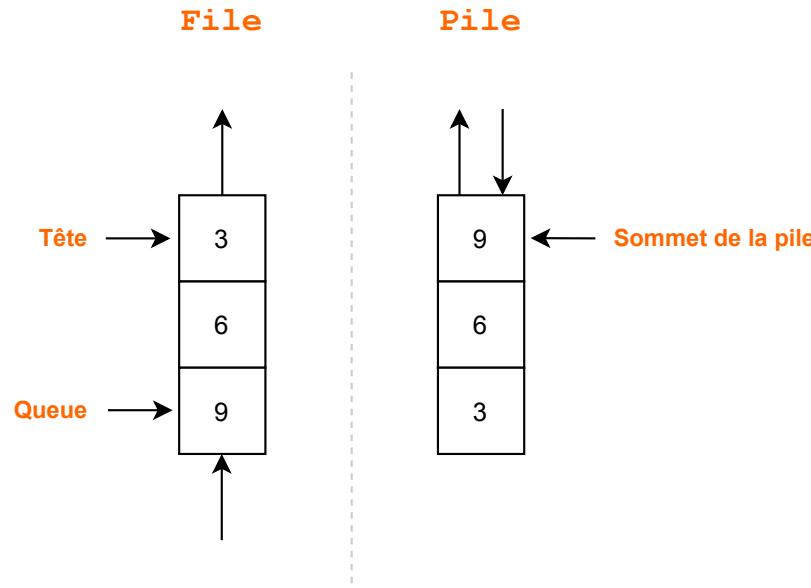
FILES ET PILES

- Ces types abstraits de données nous permettent de :
 - Savoir si la structure est **vide**
 - **Récupérer** un élément
 - **Insérer** un élément
- Dans la majorité des cas :
 - **Empiler/Enfiler** un élément se dit **push**
 - **Dépiler/Défiler** un élément se dit **pop**

FILES ET PILES

- Ces types abstraits de données nous permettent de :
 - Savoir si la structure est **vide**
 - **Récupérer** un élément
 - **Insérer** un élément
- Dans la majorité des cas :
 - **Empiler/Enfiler** un élément se dit **push**
 - **Dépiler/Défiler** un élément se dit **pop**
- **Attention** : Le fait de récupérer un élément l'enlève de la pile/file. Il faudra donc le réinsérer si on souhaite le garder.

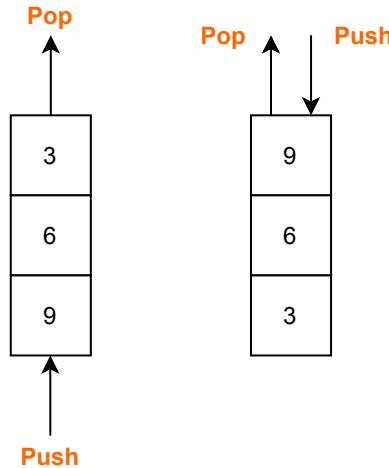
TERMINOLOGIE



- Une file **pop** un élément de la **tête** et **push** un élément dans la **queue**
- Une pile **pop** et **push** les éléments sur le **sommet de la pile** uniquement

EXEMPLE DE FILE ET PILE: Push

On peut représenter une file/pile (nommée **s**) comme suit :

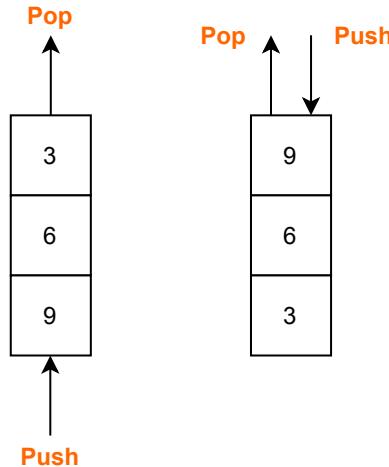


Ces structures sont le résultat du code suivant :

```
1. s.push(3)
2. s.push(6)
3. s.push(9)
```

EXAMPLE DE FILE ET PILE: Pop

En reprenant les structures de la slide précédent e:



L'opération **Pop** produit différents résultats :

1. file.pop() -> 3 # Premier élément inséré (FIFO)
- 2.
3. pile.pop() -> 9 # Dernier élément inséré (LIFO)

FILES

- Les files sont largement utilisées lorsque l'on veut modéliser une file d'attente.
- Par exemple :
 - Les impressions sont gérées par une file.
 - Dans une simulation de guichet, on modélisera généralement l'arrivée des clients par une file.

TYPE ABSTRAIT File

1. Constructeurs :

```
file_vide : () -> File
    # produit la file vide
file : (Valeur * File) -> File
    # produit une file à partir d'une valeur et d'une autre file
```

TYPE ABSTRAIT File

1. Constructeurs :

```
file_vide : () -> File
    # produit la file vide
file : (Valeur * File) -> File
    # produit une file à partir d'une valeur et d'une autre file
```

2. Fonctions :

```
push : (Valeur * File) -> File
    # ajoute une valeur dans la file
pop : File -> (Valeur * File)
    # extrait la valeur de la tête de la file
    # puis renvoi la file sans cette valeur
    # Attention: La file ne doit pas être vide
```

TYPE ABSTRAIT File

1. Constructeurs :

```
file_vide : () -> File
    # produit la file vide
file : (Valeur * File) -> File
    # produit une file à partir d'une valeur et d'une autre file
```

2. Fonctions :

```
push : (Valeur * File) -> File
    # ajoute une valeur dans la file
pop : File -> (Valeur * File)
    # extrait la valeur de la tête de la file
    # puis renvoi la file sans cette valeur
    # Attention: La file ne doit pas être vide
```

3. Prédicat :

```
est_vide : File -> bool
    # indique si la file est vide
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

RÉCURSIVE

1. Constructeurs :

```
def file_vide():
    return []

def file(valeur, file):
    return [valeur, file]
```

RÉCURSIVE

1. Constructeurs :

```
def file_vide():
    return []

def file(valeur, file):
    return [valeur, file]
```

2. Sélecteurs :

```
def push(valeur, file):
    if est_vide(file):
        return [valeur, file_vide()]
    return [file[0], push(valeur, file[1])]

def pop(file):
    return (file[0], file[1])
```

RÉCURSIVE

1. Constructeurs :

```
def file_vide():
    return []

def file(valeur, file):
    return [valeur, file]
```

2. Sélecteurs :

```
def push(valeur, file):
    if est_vide(file):
        return [valeur, file_vide()]
    return [file[0], push(valeur, file[1])]

def pop(file):
    return (file[0], file[1])
```

3. Prédicat :

```
def est_vide(file):
    return file == file_vide()
```

RÉCURSIVE

Exemple d'utilisation :

```
1. F1 = file_vide()
2. F1 = push(3, F1)
3. print(F1) # [3, []]
4.
5. F1 = push(6, F1)
6. print(F1) # [3, [6, []]]
7.
8. resultat = pop(push(9, F1))
9. print(resultat) # (3, [6, [9, []]])
10.
11. # Nous pouvons copier une file en la défilant dans une autre file
11. F2 = file_vide()
12. while not est_vide(F1):
13.     res = pop(F1) # res[0] contient la valeur, res[1] contient la file :
14.     F2 = push(res[0], F2)
15.     F1 = res[1]
16. print(F2) # [6, [9, []]]
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

2. Classe File

→ paradigme objet

CLASSE FILE

La classe file est implémentée à l'aide d'un tableau :

```
class File:  
    file_vide = None  
  
    def __init__(self, valeur, file):  
        self._valeur = valeur  
        self._suite = file  
  
    def push(self, valeur):  
        if File.est_vide(self._suite):  
            self._suite = File(valeur, File.file_vide)  
        else:  
            self._suite.push(valeur)  
        return self  
  
    def pop(self):  
        return (self._valeur, self._suite)  
  
    def est_vide(file):  
        return file is File.file_vide
```

CLASSE FILE

Exemple d'utilisation :

```
1. F1 = File(3, File.file_vide)
3. print(F1.pop()) # (3, None)
4.
5. F2 = File(3, File.file_vide)
6. F2.push(6).push(9)
7. # F2: <-- [3]<-- [6]<-- [9]<--
8.
9. (val, F2) = F2.pop()
10. F3 = File(val, File.file_vide)
11. while not File.est_vide(F2):
12.     (val, F2) = F2.pop()
13.     F3.push(val)
14. # F3: <-- [3]<-- [6]<-- [9]<--
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

2. Classe Cellule

→ paradigme objet

3. Tableau

→ paradigme impératif

TABLEAU

File stockée dans un tableau.

```
def file_vide():
    return []

def file(valeur, file):
    return file + [valeur]

def push(valeur, file):
    return file + [valeur]

def pop(file):
    return (file[0], file[1:])

def est_vide(file):
    return file == file_vide()
```

TABLEAU

Exemple d'utilisation :

```
1. file = file_vide()
2. file = push(3, file)
3. print(pop(file)) # (3, [])
4.
5. file = file_vide()
6. file = push(3, file)
7. file = push(6, file)
8. file = push(9, file)
9. copie = file_vide()
10.
11. while not est_vide(file):
12.     (val, file) = pop(file)
12.     copie = push(val, copie)
13. print(copie) # [3, 6, 9]
```

PILE

- Les piles sont omniprésentes en informatique :
- Par exemple :
 - L'annulation de commande (CTRL-Z) est gérée par une pile.
 - Vérifier si une expression est bien parenthésée nécessite une pile.
 - Évaluer une expression arithmétique peut être fait avec une pile.

TYPE ABSTRAIT Pile

1. Constructeurs :

```
pile_vide : () -> Pile
    # produit la pile vide
pile : (Valeur * Pile) -> Pile
    # produit une pile à partir d'une valeur et d'une autre pile
```

TYPE ABSTRAIT Pile

1. Constructeurs :

```
pile_vide : () -> Pile
    # produit la pile vide
pile : (Valeur * Pile) -> Pile
    # produit une pile à partir d'une valeur et d'une autre pile
```

2. Fonctions :

```
push : (Valeur * Pile) -> Pile
    # ajoute une valeur dans la Pile
pop : Pile -> (Valeur * Pile)
    # extrait la valeur du sommet de la pile
    # puis renvoi la Pile sans cette valeur
    # Attention: La Pile ne doit pas être vide
```

TYPE ABSTRAIT Pile

1. Constructeurs :

```
pile_vide : () -> Pile
    # produit la pile vide
pile : (Valeur * Pile) -> Pile
    # produit une pile à partir d'une valeur et d'une autre pile
```

2. Fonctions :

```
push : (Valeur * Pile) -> Pile
    # ajoute une valeur dans la Pile
pop : Pile -> (Valeur * Pile)
    # extrait la valeur du sommet de la pile
    # puis renvoi la Pile sans cette valeur
    # Attention: La Pile ne doit pas être vide
```

3. Prédicat :

```
est_vide : Pile -> bool
    # indique si la Pile est vide
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

RÉCURSIVE

1. Constructeurs :

```
def pile_vide():
    return []

def pile(valeur, pile):
    return [valeur, pile]
```

RÉCURSIVE

1. Constructeurs :

```
def pile_vide():
    return []

def pile(valeur, pile):
    return [valeur, pile]
```

2. Fonctions :

```
# Nous considérons que le sommet de la Pile est la première case
def push(valeur, pile):
    return [valeur, pile]

def pop(pile):
    return (pile[0], pile[1])
```

RÉCURSIVE

1. Constructeurs :

```
def pile_vide():
    return []

def pile(valeur, pile):
    return [valeur, pile]
```

2. Fonctions :

```
# Nous considérons que le sommet de la Pile est la première case
def push(valeur, pile):
    return [valeur, pile]

def pop(pile):
    return (pile[0], pile[1])
```

3. Prédicat :

```
def est_vide(pile):
    return pile == pile_vide()
```

RÉCURSIVE

Exemple d'utilisation :

```
1. P1 = pile_vide()
2. P1 = push(3, P1)
3. print(P1) # [3, []]
4.
5. P1 = push(6, P1)
6. print(P1) # [6, [3, []]]
7.
8. resultat = pop(push(9, P1))
9. print(resultat) # (9, [6, [3, []]])
10.
11. # Nous pouvons inverser une pile en la dépilant dans une autre pile
11. P2 = pile_vide()
12. while not est_vide(P1):
13.     (val, P1) = pop(P1) # res[0] contient la valeur, res[1] contient la
14.     P2 = push(val, P2)
15. print(P2) # [3, [6, []]]
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

2. Classe Pile

→ paradigme objet

CLASSE PILE

La classe Pile est implémentée de la même façon que la classe File :

```
class Pile:  
    pile_vide = None  
  
    def __init__(self, valeur, pile):  
        self._valeur = valeur  
        self._suite = pile  
  
    def push(self, valeur):  
        return Pile(valeur, self)  
  
    def pop(self):  
        return (self._valeur, self._suite)  
  
    def est_vide(pile):  
        return pile is Pile.pile_vide
```

CLASSE PILE

Exemple d'utilisation :

```
1. P1 = Pile(3, Pile.pile_vide)
2. print(P1.pop()) # (3, None)
3.
4. P2 = Pile(3, Pile.pile_vide)
5. P2.push(6).push(9)
6.
7. P3 = Pile(P2.pop(), Pile.pile_vide)
8. while not Pile.est_vide(P2):
9.     (val, P2) = P2.pop()
9.     P3.push(val)
10. # P3: [3, 6, 9]
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Récursive

→ paradigme fonctionnel

2. Classe Cellule

→ paradigme objet

3. Tableau

→ paradigme impératif

TABLEAU

Pile stockée dans un tableau.

```
def pile_vide():
    return []

def pile(valeur, pile):
    return [valeur] + pile

pile = pile_vide()

def push(valeur, pile):
    return [valeur] + pile

def pop(pile):
    return (pile[0], pile[1:])

def est_vide(pile):
    return pile == pile_vide()
```

TABLEAU

Exemple d'utilisation :

```
1. pile = pile_vide()
2. pile = push(3, pile)
3. print(pop(pile)) # (3, [])
4.
5. pile = pile_vide()
6. pile = push(3, pile)
7. pile = push(6, pile)
8. pile = push(9, pile)
9. print(pile) # [9, 6, 3]
10. rev = pile_vide()
11.
12. while not est_vide(pile):
13.     (val, pile) = pop(pile)
14.     rev = push(val, rev)
15. print(rev) # [3, 6, 9]
```

ARBRES

- Structures de données :
 - hiérarchiques,
 - naturellement récursives.

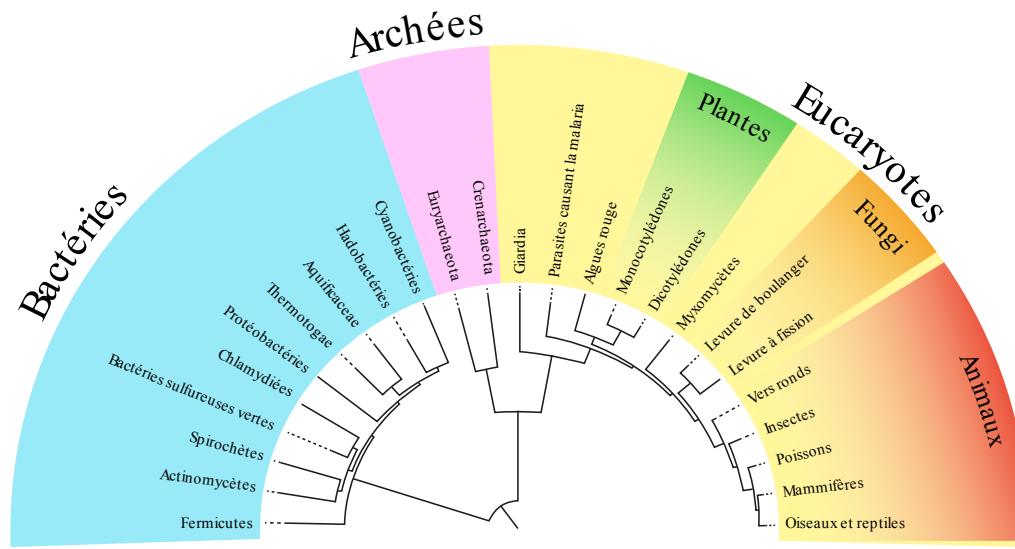
ARBRES

- Structures de données :
 - **hiérarchiques**,
 - **naturellement récursives**.
- Utilisées pour représenter des ensembles de **données** :
 - système de fichiers (répertoires et fichiers),
 - bases de données,
 - documents structurés (HTML, XML)

ARBRES

- Structures de données :
 - **hiérarchiques**,
 - **naturellement récursives**.
- Utilisées pour représenter des ensembles de **données** :
 - système de fichiers (répertoires et fichiers),
 - bases de données,
 - documents structurés (HTML, XML)
- Utilisées dans les **algorithmes** :
 - arbre de décision
 - compression de textes (Huffman)
 - synthèse d'image (*kd-tree*)

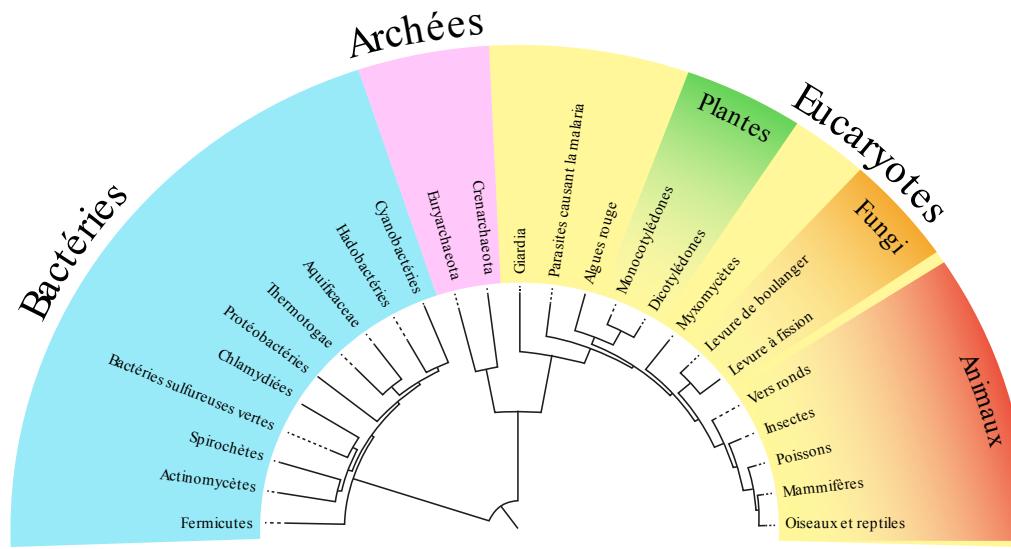
EXEMPLE : ARBRE PHYLOGÉNÉTIQUE



Wikimedia Commons

- Quel lien ?

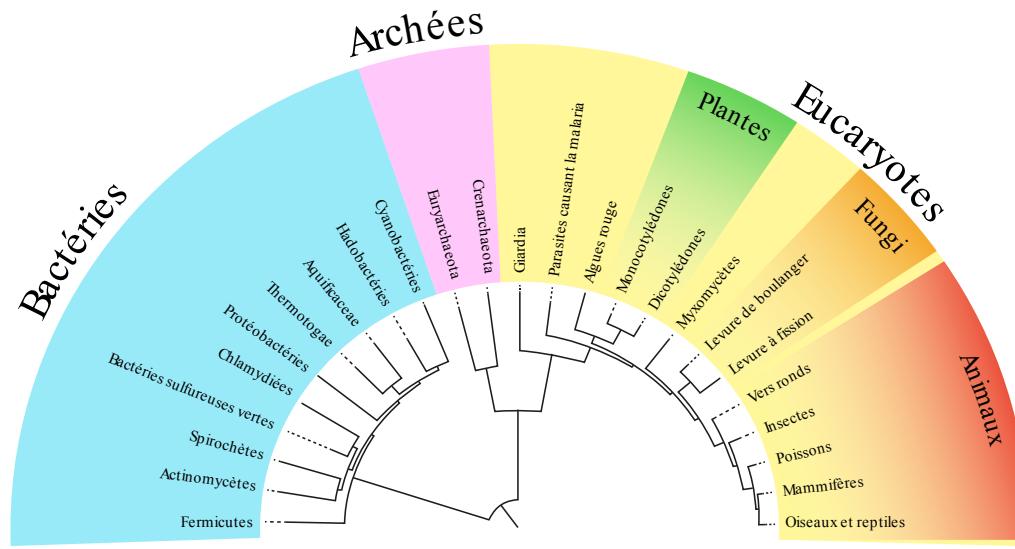
EXEMPLE : ARBRE PHYLOGÉNÉTIQUE



Wikimedia Commons

- **Quel lien ?**
 - du plus générique au plus spécifique

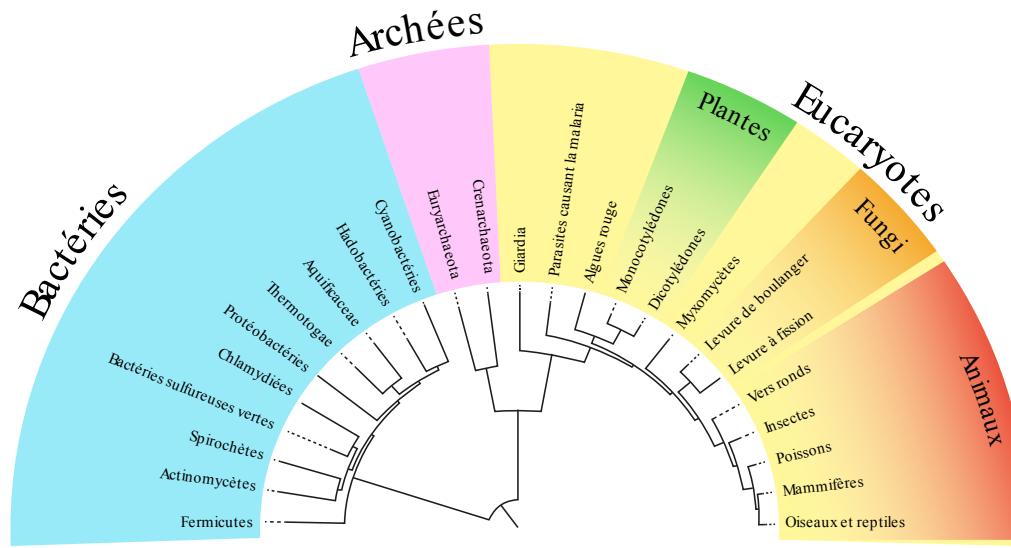
EXEMPLE : ARBRE PHYLOGÉNÉTIQUE



Wikimedia Commons

- **Quel lien ?**
 - du plus générique au plus spécifique
 - du plus ancien au plus récent

EXEMPLE : ARBRE PHYLOGÉNÉTIQUE

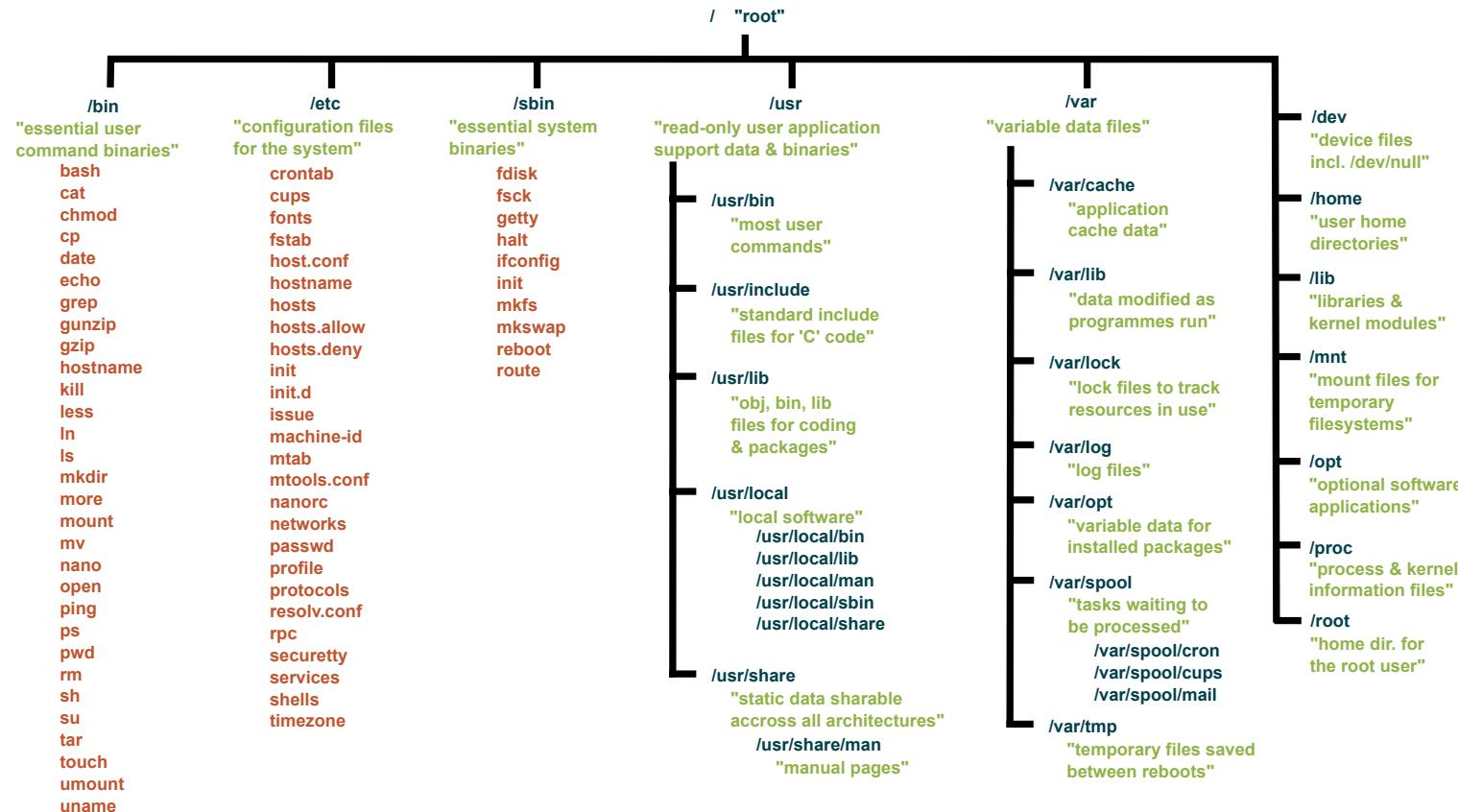


Wikimedia Commons

- **Quel lien ?**

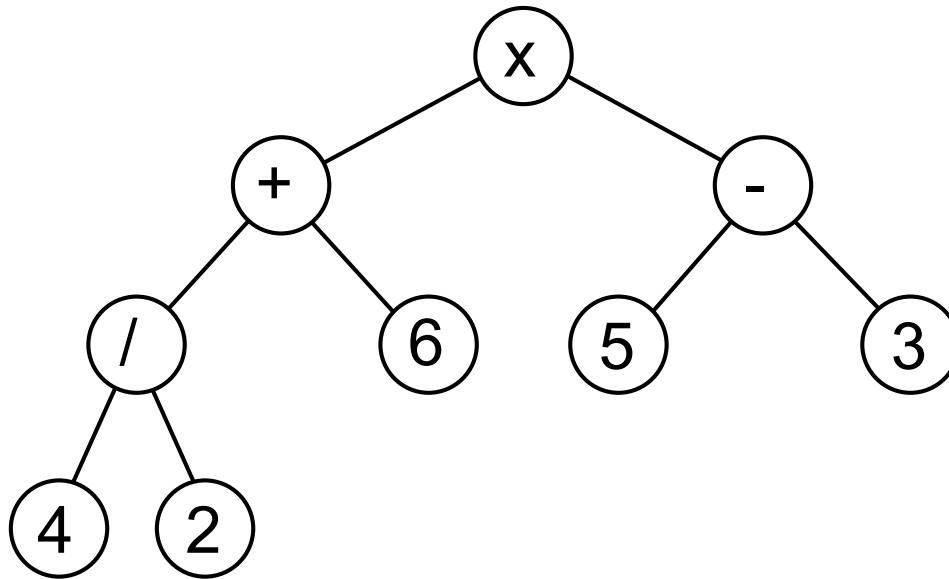
- du plus générique au plus spécifique
- du plus ancien au plus récent
- du plus complexe au plus simple

EXEMPLE : SYSTÈME DE FICHIERS



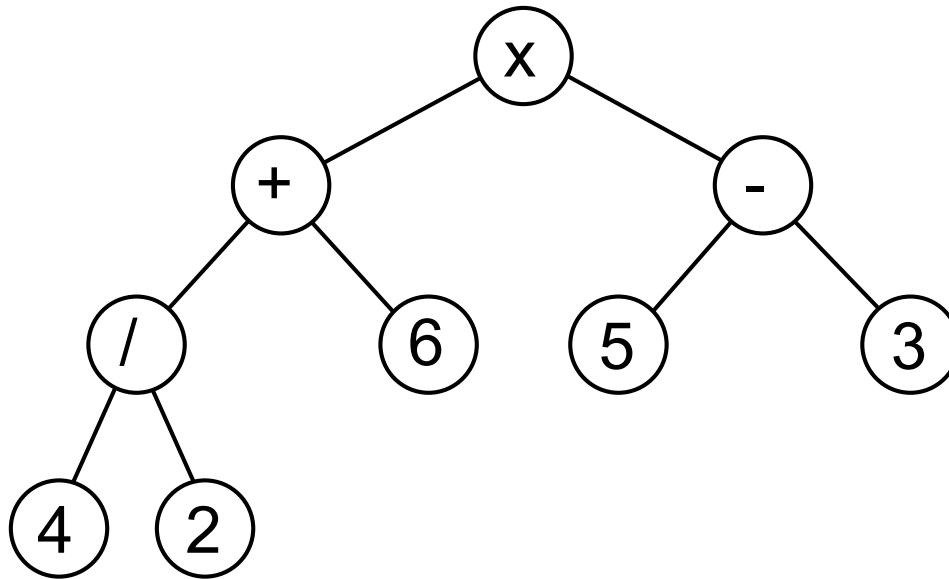
Wikimedia Commons

EXEMPLE : EXPRESSIONS ARITHMÉTIQUES



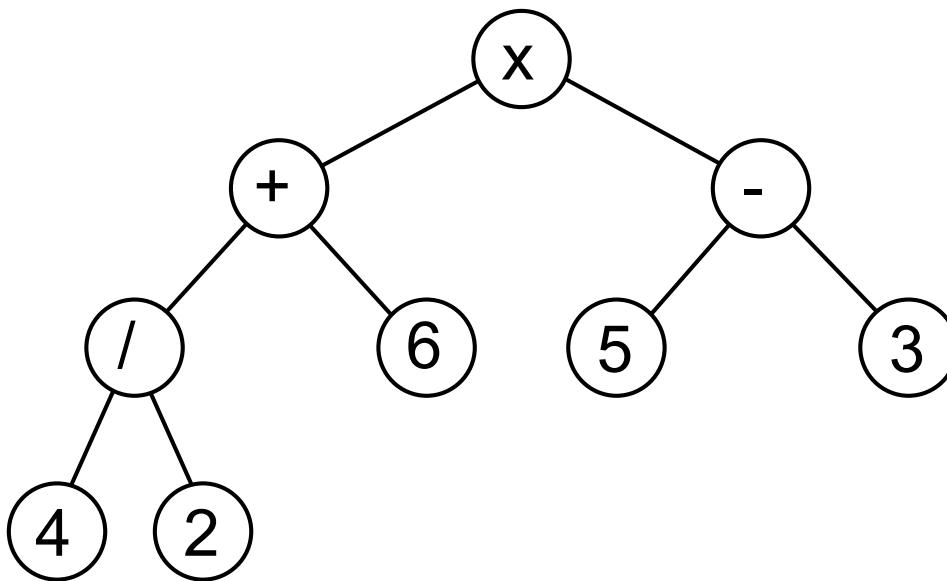
- Quelle expression ?

EXEMPLE : EXPRESSIONS ARITHMÉTIQUES



- Quelle expression ?
 - $4/2+6x5-3$

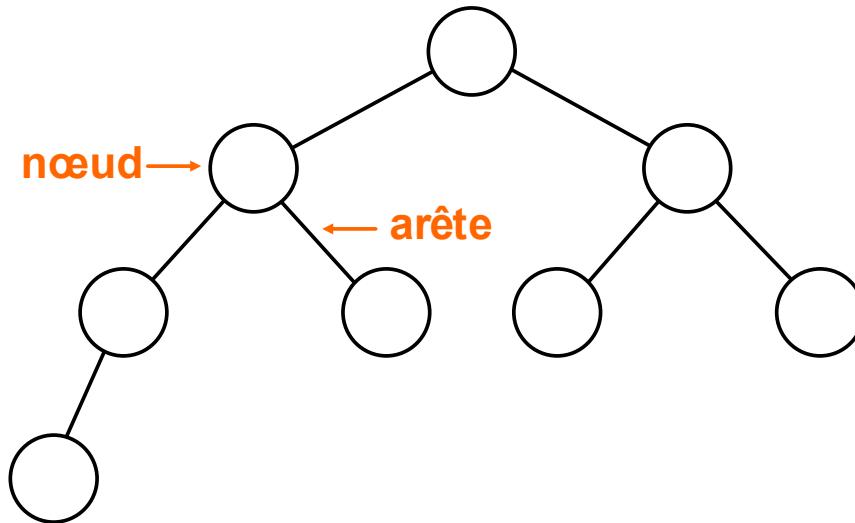
EXEMPLE : EXPRESSIONS ARITHMÉTIQUES



- Quelle expression ?

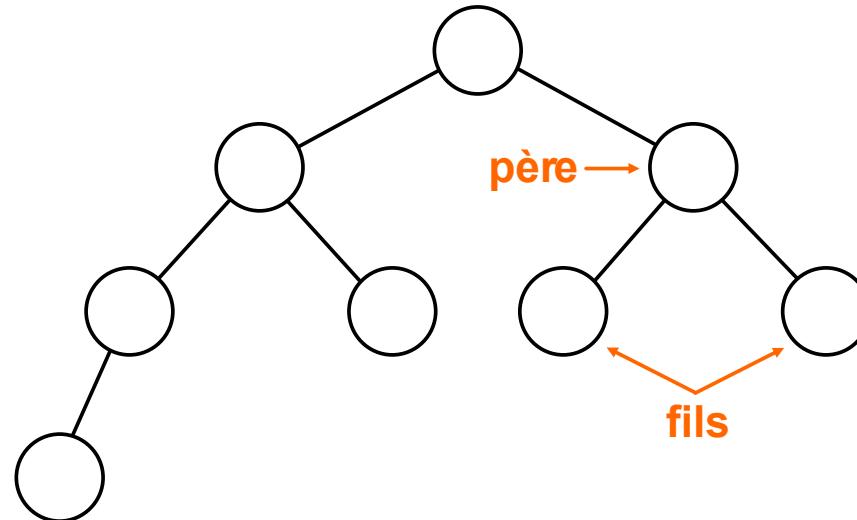
- $4/2+6x5-3$
- $((4/2)+6)x(5-3)$

TERMINOLOGIE



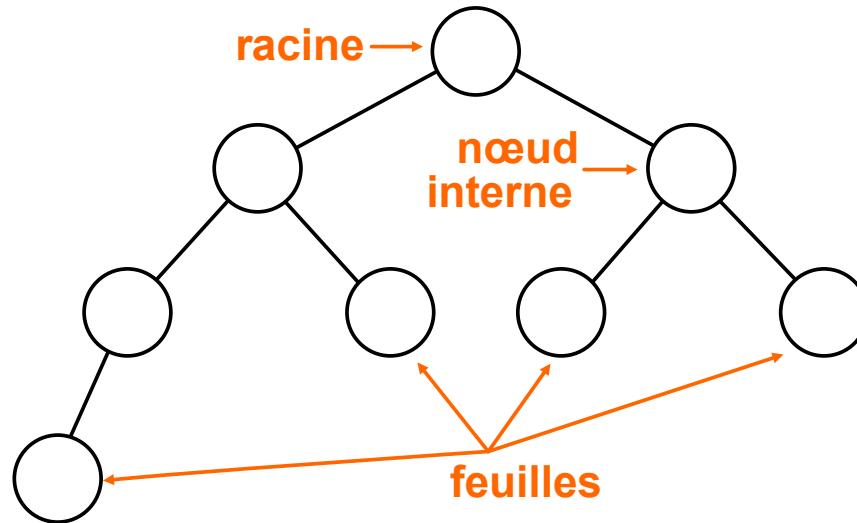
- un **nœud** est caractérisé par une donnée (ou étiquette),
- une **arête** (ou lien) relie deux nœuds.
- l'**arbre vide** n'est pas un nœud.

TERMINOLOGIE



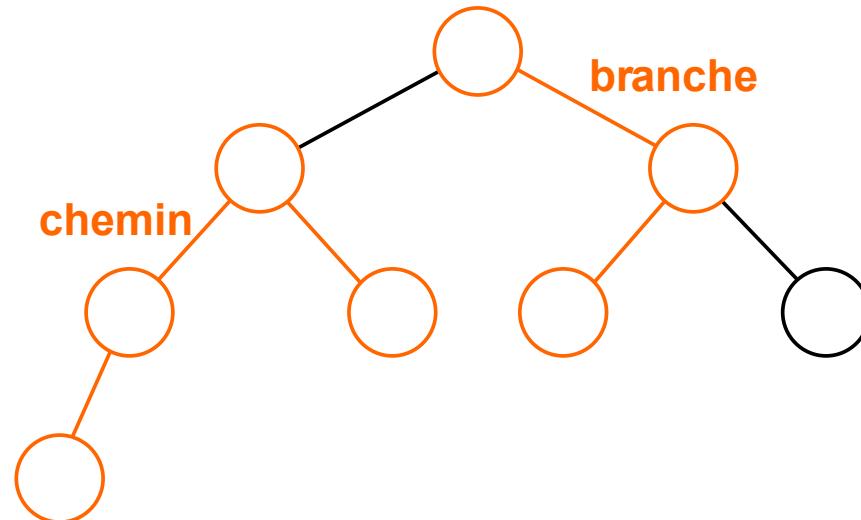
- les **fils** sont l'ensemble des nœuds reliés à un même nœud par des arêtes entrantes.
- le **père** (ou parent) est le nœud relié à ses nœuds fils par une arête sortante.
- un **sous-arbre** est l'ensemble des nœuds et arêtes d'un nœud parent et de ses fils.

TERMINOLOGIE



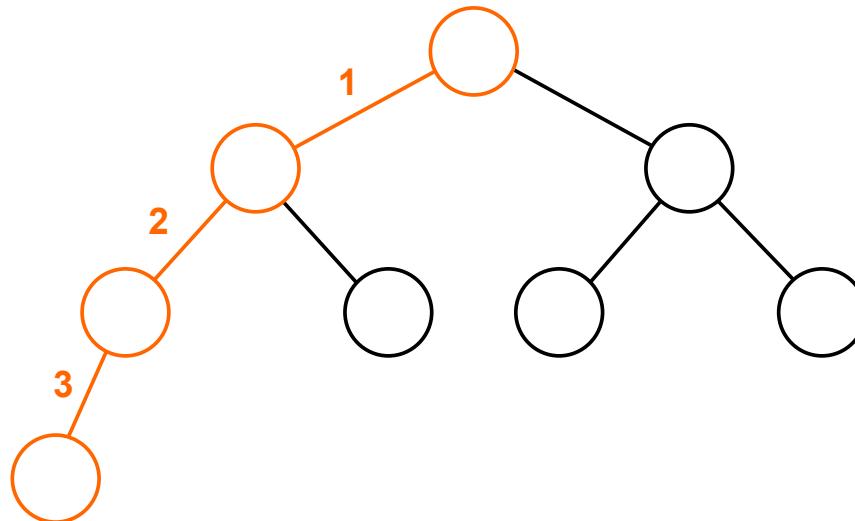
- la **racine** est le seul nœud sans père.
- une **feuille** est un nœud sans fils.
- un **nœud interne** est un nœud qui n'est pas une feuille.

TERMINOLOGIE



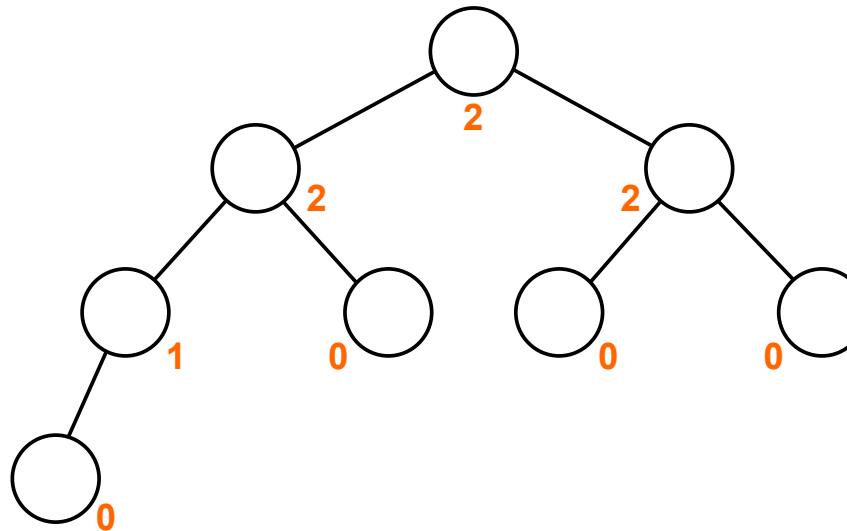
- un **chemin** est une liste de nœuds reliés par des arêtes.
- une **branche** est le chemin le plus court reliant un nœud à la racine.

QUELQUES MESURES SUR LES ARBRES



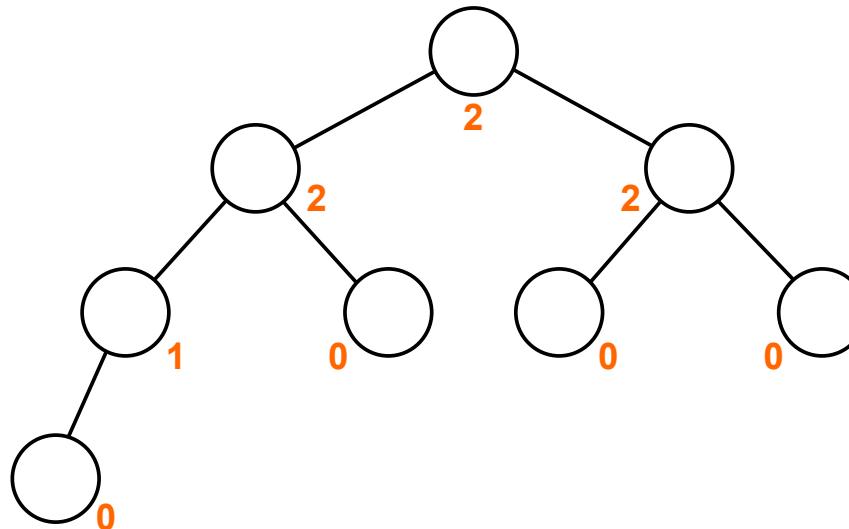
- la **taille** d'un arbre est le nombre de nœuds de l'arbre.
- la **profondeur** d'un nœud est le nombre d'arêtes sur la branche qui le relie à la racine.
- la **hauteur** d'un arbre est la profondeur maximale de l'ensemble des nœuds de l'arbre.

QUELQUES MESURES SUR LES ARBRES



- l'**arité ou degré d'un nœud** est le nombre de fils du nœud.
- l'**arité ou degré d'un arbre** est le nombre maximal de fils des nœuds de l'arbre.

QUELQUES MESURES SUR LES ARBRES



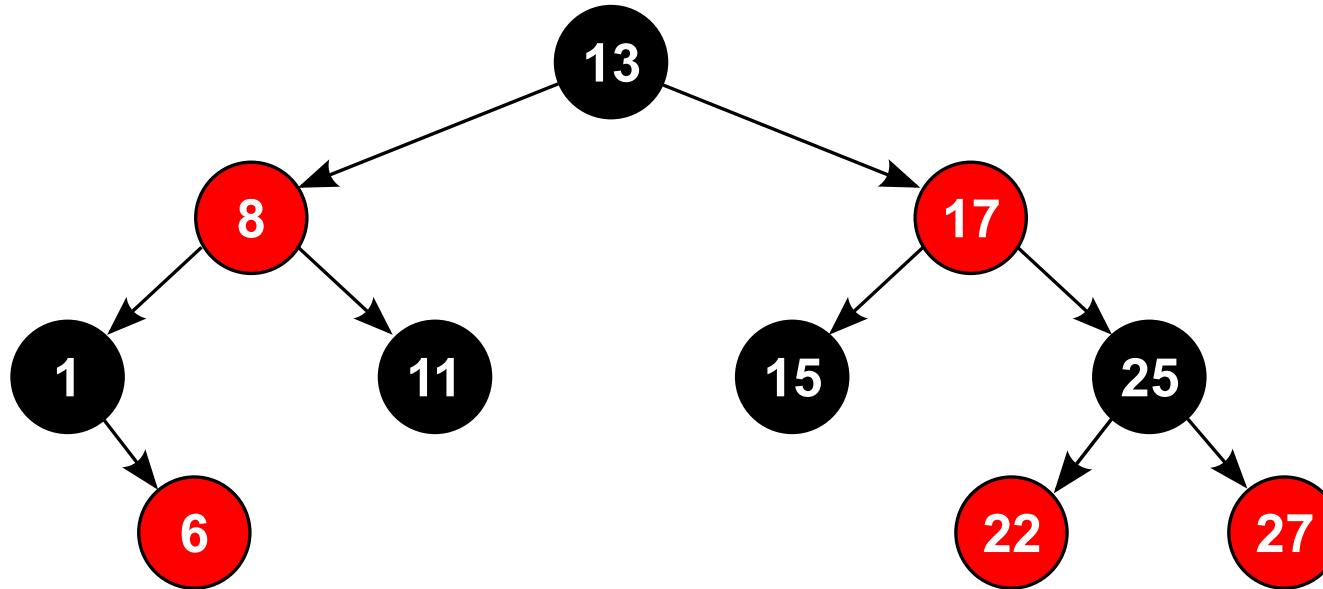
- l'**arité ou degré d'un nœud** est le nombre de fils du nœud.
- l'**arité ou degré d'un arbre** est le nombre maximal de fils des nœuds de l'arbre.

Un **arbre binaire** est donc un arbre d'**arité deux**.

POURQUOI LES ARBRES BINAIRES ?

Structure pour effectuer des recherches rapides,
ou maintenir efficacement des ensembles triés.

Par exemple, **arbre rouge-noir** :

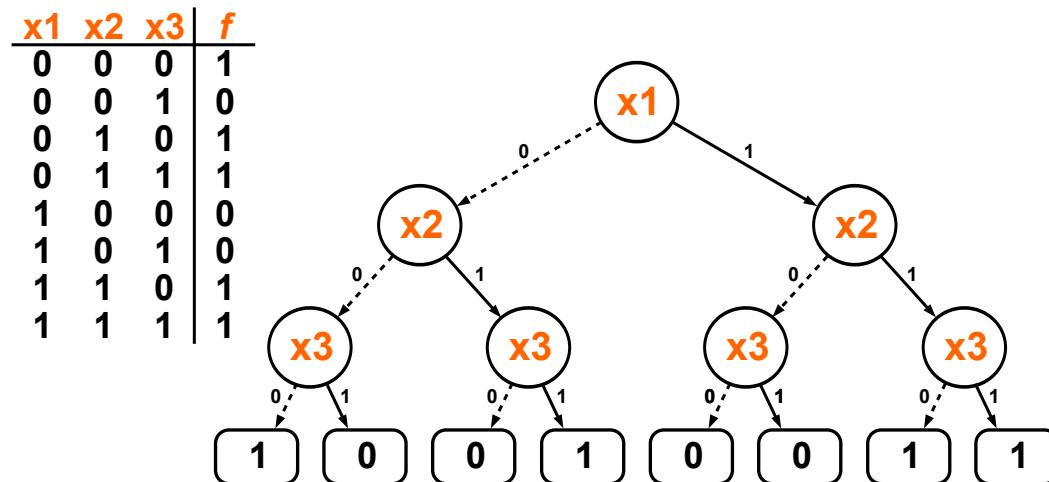


Wikimedia Commons

POURQUOI LES ARBRES BINAIRES ?

Aide à la création de circuits (synthèse logique)
et vérification formelle de programme.

Par exemple, **arbre binaire de décision** :

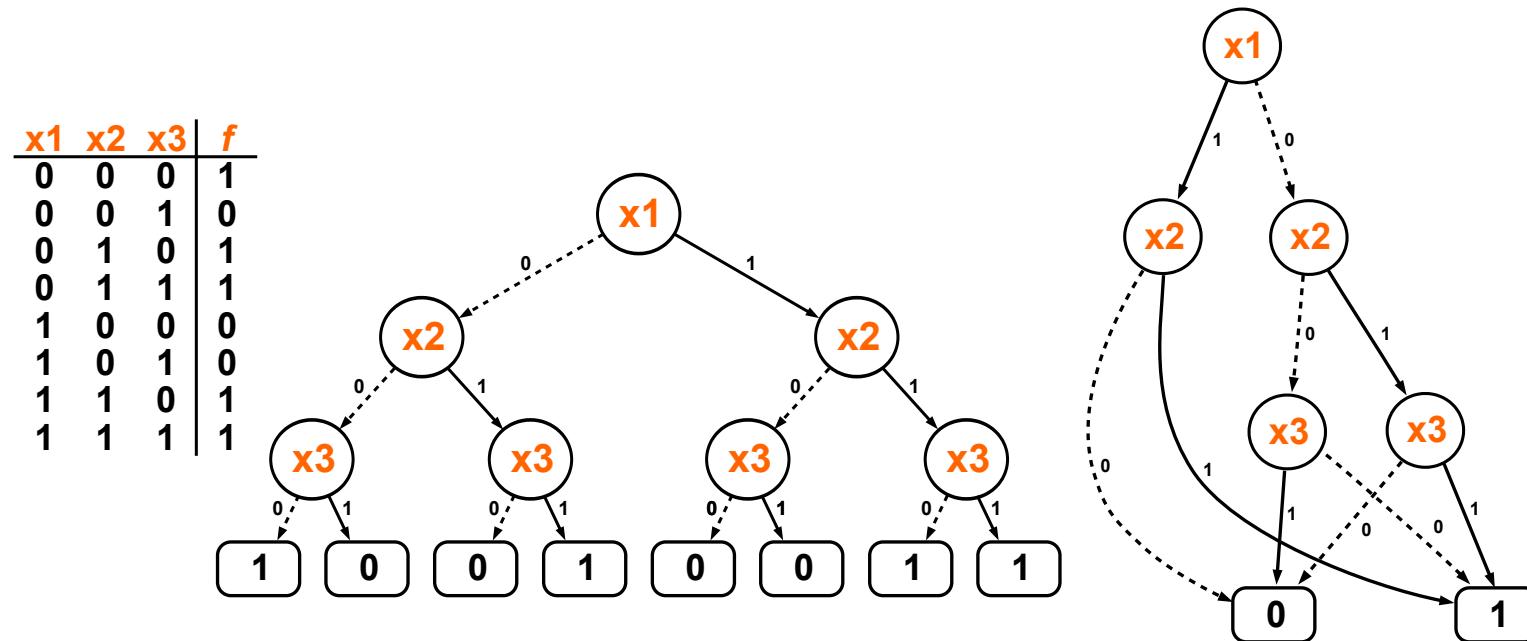


Wikimedia Commons

POURQUOI LES ARBRES BINAIRES ?

Aide à la création de circuits (synthèse logique)
et vérification formelle de programme.

Par exemple, **arbre binaire de décision** :

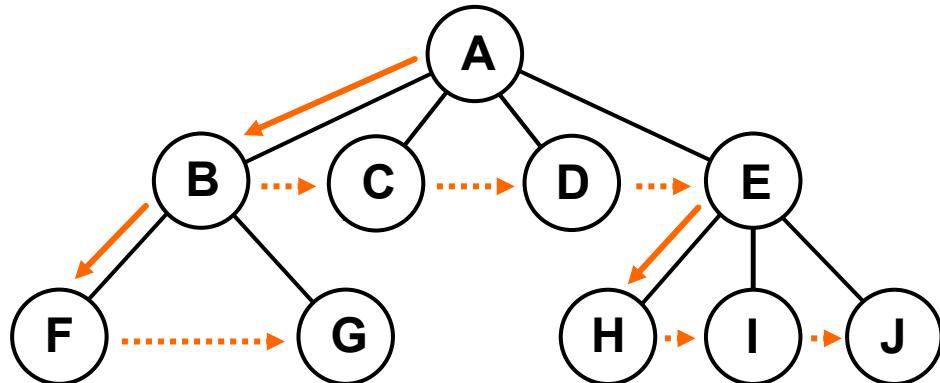


Wikimedia Commons

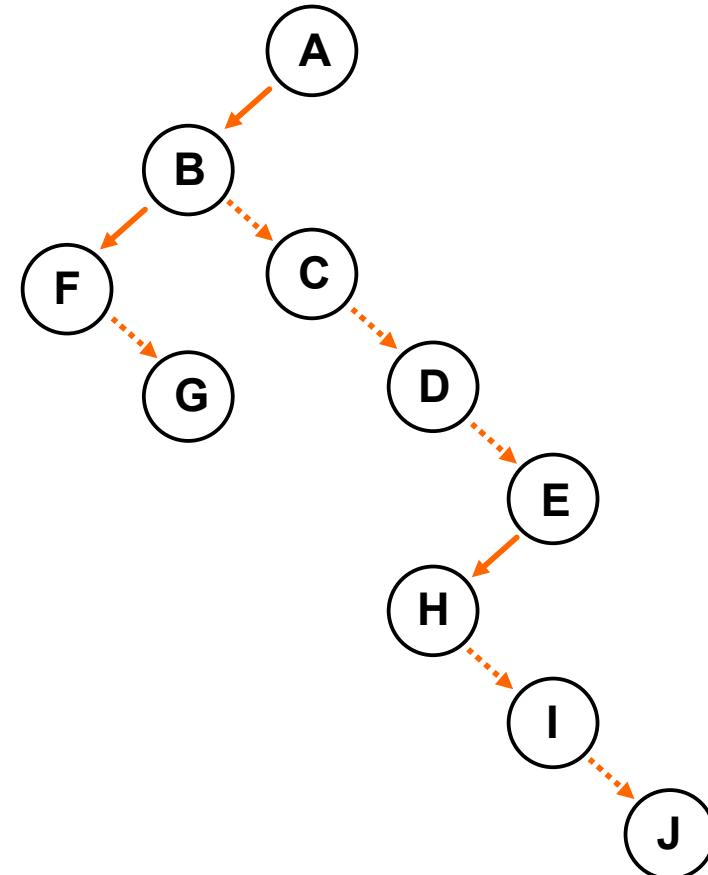
POURQUOI LES ARBRES BINAIRES ?

Permet de représenter un arbre quelconque.

Arbre **n**-aire



Arbre **binaire** équivalent



DÉFINITION RÉCURSIVE

- Un **arbre binaire** c'est :
 - soit un **arbre vide**, noté Δ ;
 - soit un **triplet** (e, g, d) , appelé nœud, avec :
 - e son étiquette,
 - g son sous-arbre binaire **gauche**,
 - d son sous-arbre binaire **droit**.

DÉFINITION RÉCURSIVE

- Un **arbre binaire** c'est :
 - soit un **arbre vide**, noté Δ ;
 - soit un **triplet** (e, g, d) , appelé nœud, avec :
 - e son étiquette,
 - g son sous-arbre binaire **gauche**,
 - d son sous-arbre binaire **droit**.
- Définition **récursive** : un arbre binaire est construit à partir d'un élément et des sous-arbres binaires.

DÉFINITION RÉCURSIVE

- Un **arbre binaire** c'est :
 - soit un **arbre vide**, noté Δ ;
 - soit un **triplet** (e, g, d) , appelé nœud, avec :
 - e son étiquette,
 - g son sous-arbre binaire **gauche**,
 - d son sous-arbre binaire **droit**.
- Définition **récursive** : un arbre binaire est construit à partir d'un élément et des sous-arbres binaires.
- La première partie de la définition assure l'arrêt et donc la cohérence de la définition.

QUELQUES ARBRES BINAIRES

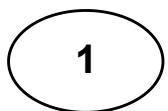
- Dessinez chacun des arbres ci-dessous :
 - $(1, \Delta, \Delta)$
 - $(3, (1, \Delta, (4, (1, \Delta, (5, \Delta, \Delta)), \Delta)), \Delta)$
 - $(3, (1, (1, \Delta, \Delta), \Delta), (4, (5, \Delta, \Delta), (9, \Delta, \Delta)))$
 - $(3, (1, (1, \Delta, \Delta), (5, \Delta, \Delta)), (4, (9, \Delta, \Delta), (2, \Delta, \Delta)))$
- Donnez sa taille et sa hauteur, le nombre de feuilles, le nombre de nœuds à chaque profondeur.

QUELQUES ARBRES BINAIRES

(1, Δ , Δ)

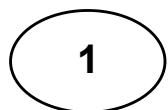
QUELQUES ARBRES BINAIRES

$(1, \Delta, \Delta)$



QUELQUES ARBRES BINAIRES

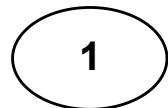
$(1, \Delta, \Delta)$



- taille : 1
- hauteur : 0
- nb feuilles : 1

QUELQUES ARBRES BINAIRES

$(1, \Delta, \Delta)$



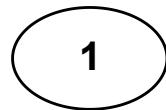
- taille : 1
- hauteur : 0
- nb feuilles : 1

$(3, (1, \Delta, (4, (1, \Delta, (5, \Delta, \Delta)), \Delta)), \Delta)$



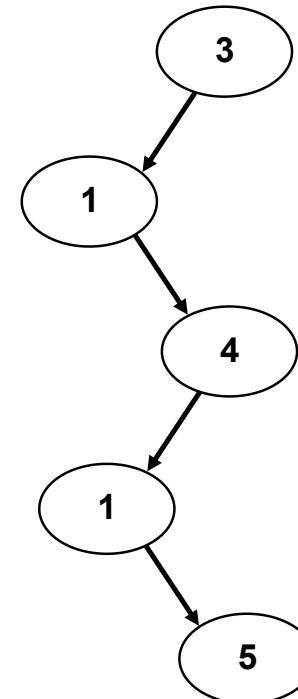
QUELQUES ARBRES BINAIRES

$(1, \Delta, \Delta)$



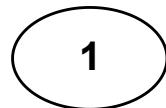
- taille : 1
- hauteur : 0
- nb feuilles : 1

$(3, (1, \Delta, (4, (1, \Delta, (5, \Delta, \Delta)), \Delta)), \Delta)$



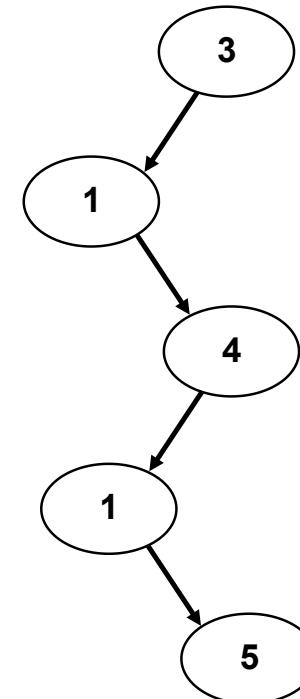
QUELQUES ARBRES BINAIRES

$(1, \Delta, \Delta)$



- taille : 1
- hauteur : 0
- nb feuilles : 1

$(3, (1, \Delta, (4, (1, \Delta, (5, \Delta, \Delta)), \Delta)), \Delta)$



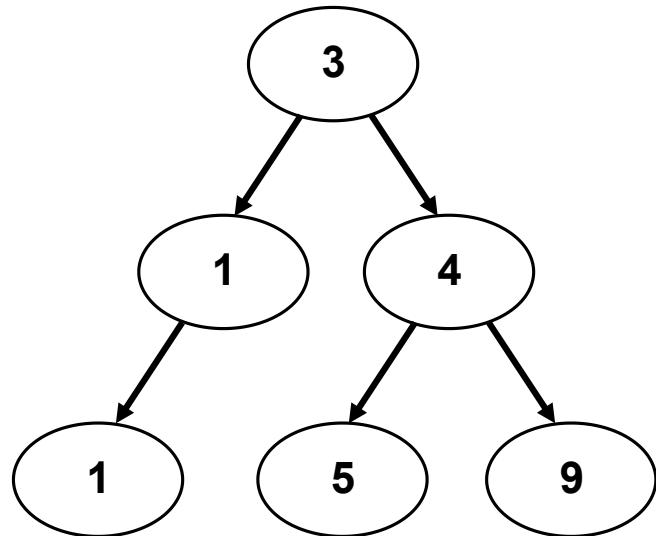
- taille : 5
- hauteur : 4
- nb feuilles : 1

QUELQUES ARBRES BINAIRES

(3, (1, (1, Δ, Δ), Δ), (4, (5, Δ, Δ), (9,
Δ, Δ))))

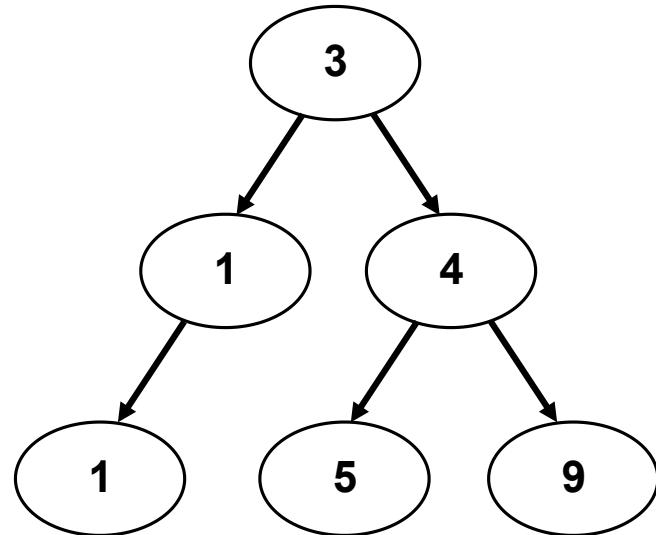
QUELQUES ARBRES BINAIRES

(3, (1, (1, Δ , Δ), Δ), (4, (5, Δ , Δ), (9,
 Δ , Δ)))



QUELQUES ARBRES BINAIRES

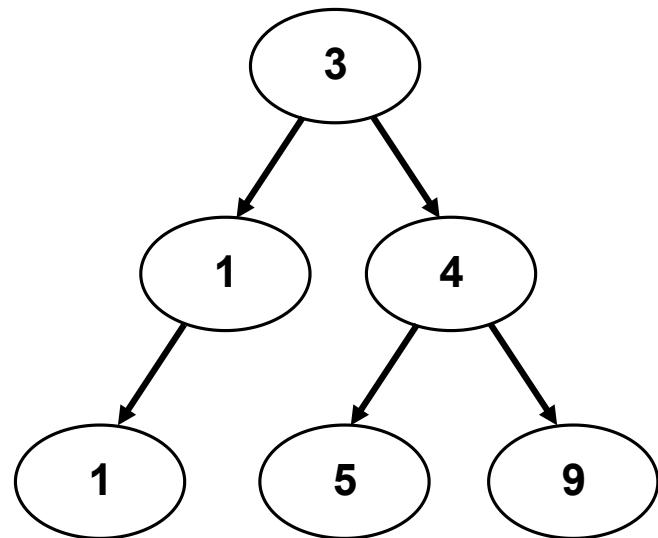
(3, (1, (1, Δ , Δ), Δ), (4, (5, Δ , Δ), (9,
 Δ , Δ)))



- taille : 6
- hauteur : 2
- nb feuilles : 3

QUELQUES ARBRES BINAIRES

(3, (1, (1, Δ, Δ), Δ), (4, (5, Δ, Δ), (9, Δ, Δ)))

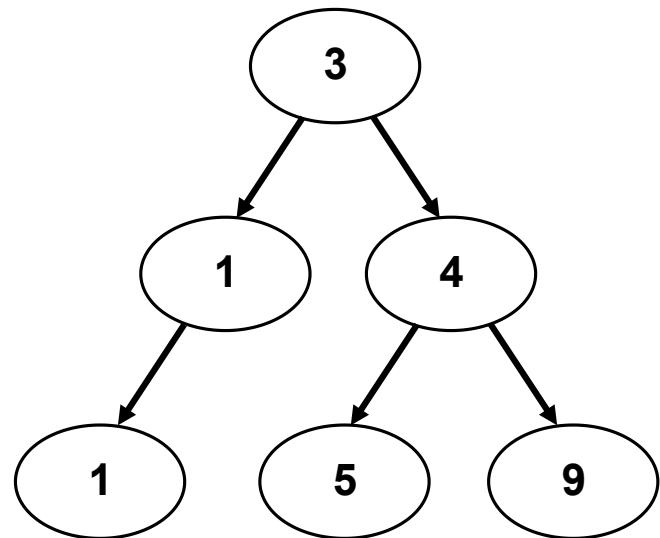


(3, (1, (1, Δ, Δ), (5, Δ, Δ)), (4, (9, Δ, Δ), (2, Δ, Δ)))

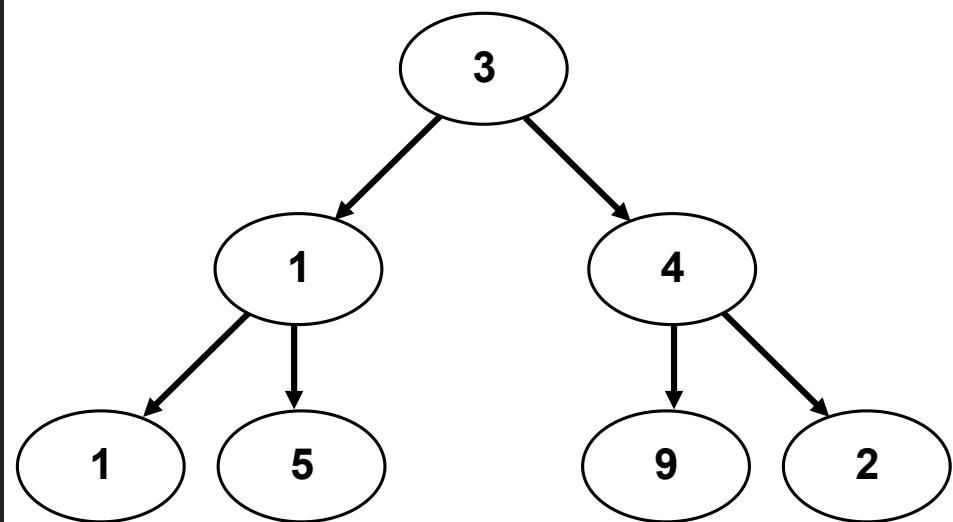
- taille : 6
- hauteur : 2
- nb feuilles : 3

QUELQUES ARBRES BINAIRES

$(3, (1, (1, \Delta, \Delta), \Delta), (4, (5, \Delta, \Delta), (9, \Delta, \Delta)))$



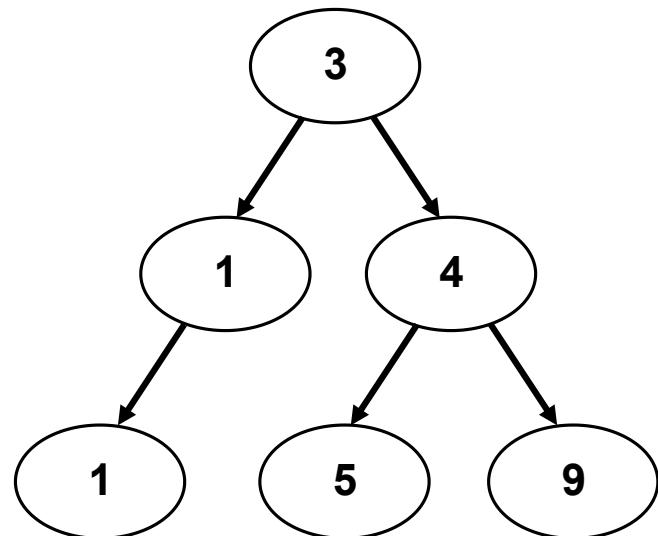
$(3, (1, (1, \Delta, \Delta), (5, \Delta, \Delta)), (4, (9, \Delta, \Delta), (2, \Delta, \Delta)))$



- taille : 6
- hauteur : 2
- nb feuilles : 3

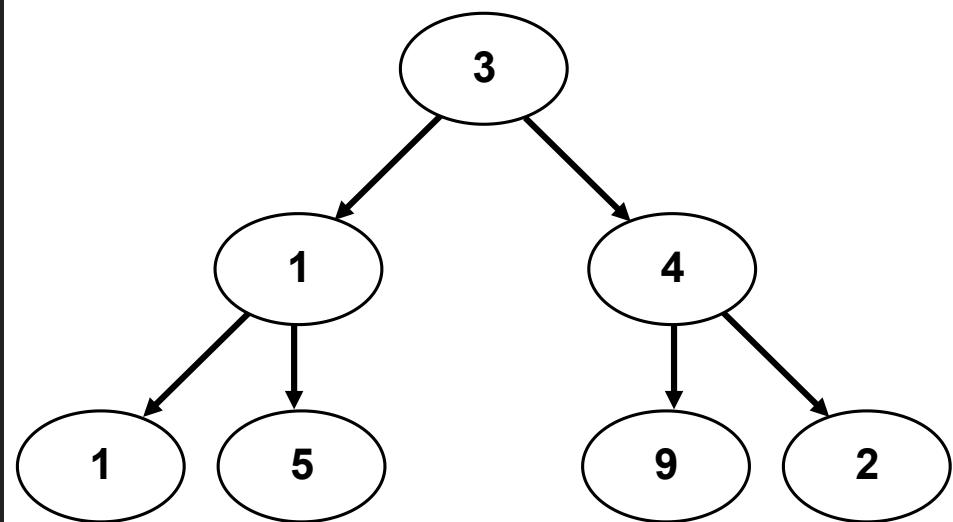
QUELQUES ARBRES BINAIRES

$(3, (1, (1, \Delta, \Delta), \Delta), (4, (5, \Delta, \Delta), (9, \Delta, \Delta)))$



- taille : 6
- hauteur : 2
- nb feuilles : 3

$(3, (1, (1, \Delta, \Delta), (5, \Delta, \Delta)), (4, (9, \Delta, \Delta), (2, \Delta, \Delta)))$



- taille : 7
- hauteur : 2
- nb feuilles : 4

ARBRE BINNAIRE DE HAUTEUR h

Combien de **feuilles** et de **nœuds** comporte-il :

1. Au minimum ?

ARBRE BINAIRE DE HAUTEUR h

Combien de **feuilles** et de **nœuds** comporte-il :

1. Au minimum ?

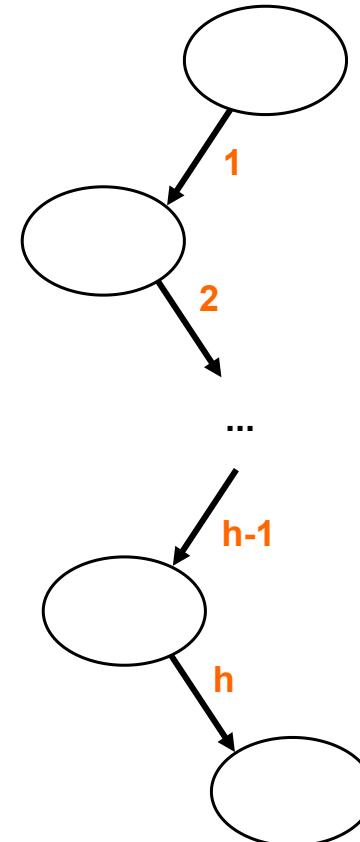
- 1 feuille et $h+1$ nœuds d'arité 1.
- On parle alors d'arbre **filiforme**.

ARBRE BINAIRES DE HAUTEUR h

Combien de **feuilles** et de **nœuds** comporte-il :

1. Au minimum ?

- 1 feuille et $h+1$ nœuds d'arité 1.
- On parle alors d'arbre **filiforme**.



ARBRE BINAIRE DE HAUTEUR h

Combien de **feuilles** et de **nœuds** comporte-il :

1. Au minimum ?

- 1 feuille et $h+1$ nœuds d'arité 1.
- On parle alors d'arbre **filiforme**.

2. Au maximum ?

ARBRE BINAIRE DE HAUTEUR h

Combien de **feuilles** et de **nœuds** comporte-il :

1. Au minimum ?

- 1 feuille et $h+1$ nœuds d'arité 1.
- On parle alors d'arbre **filiforme**.

2. Au maximum ?

- 2^h feuilles et $2^{h+1}-1$ nœuds.
- On parle alors d'arbre **complet**.

ARBRE BINNAIRE DE HAUTEUR h

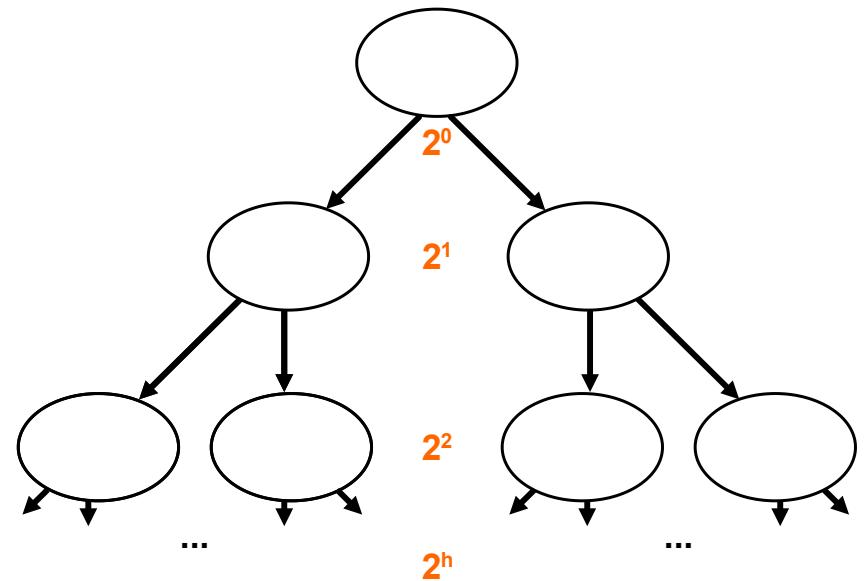
Combien de **feuilles** et de **nœuds** comporte-il :

1. Au minimum ?

- 1 feuille et $h+1$ nœuds d'arité 1.
- On parle alors d'arbre **filiforme**.

2. Au maximum ?

- 2^h feuilles et $2^{h+1}-1$ nœuds.
- On parle alors d'arbre **complet**.



TYPE ABSTRAIT Arbre Binaire

TYPE ABSTRAIT Arbre Binaire

1. Constructeurs :

```
arbre_vide : () -> Arbre binaire
  # produit l'arbre vide
noeud : (Etiquette * Arbre binaire * Arbre binaire) -> Arbre binaire
  # à partir d'un étiquette e et des arbres binaires g et d,
  # produit l'arbre binaire (e, g, d)
```

TYPE ABSTRAIT Arbre Binaire

1. Constructeurs :

```
arbre_vide : () -> Arbre binaire
    # produit l'arbre vide
noeud : (Etiquette * Arbre binaire * Arbre binaire) -> Arbre binaire
    # à partir d'un étiquette e et des arbres binaires g et d,
    # produit l'arbre binaire (e, g, d)
```

2. Sélecteurs :

```
etiquette : Arbre binaire -> Etiquette
    # à partir de l'arbre binaire (e, g, d), produit l'étiquette e
gauche : Arbre binaire -> Arbre binaire
    # à partir de l'arbre binaire (e, g, d), produit l'arbre binaire g
droit : Arbre binaire -> Arbre binaire
    # à partir de l'arbre binaire (e, g, d), produit l'arbre binaire d
```

TYPE ABSTRAIT Arbre Binaire

1. Constructeurs :

```
arbre_vide : () -> Arbre binaire
    # produit l'arbre vide
noeud : (Etiquette * Arbre binaire * Arbre binaire) -> Arbre binaire
    # à partir d'un étiquette e et des arbres binaires g et d,
    # produit l'arbre binaire (e, g, d)
```

2. Sélecteurs :

```
etiquette : Arbre binaire -> Etiquette
    # à partir de l'arbre binaire (e, g, d), produit l'étiquette e
gauche : Arbre binaire -> Arbre binaire
    # à partir de l'arbre binaire (e, g, d), produit l'arbre binaire g
droit : Arbre binaire -> Arbre binaire
    # à partir de l'arbre binaire (e, g, d), produit l'arbre binaire d
```

3. Prédicat :

```
est_vide : Arbre binaire -> bool
    # à partir de l'arbre binaire A, produit un booléen
    # indiquant si A est l'arbre vide
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. **Listes de listes**

→ paradigme fonctionnel

LISTES DE LISTES

1. Constructeurs :

```
def arbre_vide():
    return []

def noeud(etiquette, gauche, droit):
    return [etiquette, gauche, droit]
```

2. Sélecteurs :

```
def etiquette(arbre):
    return arbre[0]

def gauche(arbre):
    return arbre[1]

def droit(arbre):
    return arbre[2]
```

3. Prédicat :

```
def est_vide(arbre):
    return arbre == arbre_vide()
```

LISTES DE LISTES

Exemple d'utilisation :

```
A1 = noeud('r', arbre_vide(), arbre_vide())
print(A1) # ['r', [], []]
print(est_vide(gauche(A1))) # True

a = noeud('a', arbre_vide(), arbre_vide())
b = noeud('b', arbre_vide(), arbre_vide())
A2 = noeud('r', a, b)
print(A2) # ['r', ['a', [], []], ['b', [], []]]
print(etiquette(gauche(A2))) # a

c = noeud('c', arbre_vide(), arbre_vide())
b = noeud('b', c, arbre_vide())
A3 = noeud('r', a, b)
print(A3) # ['r', ['a', [], []], ['b', ['c', [], []], []]]
print(etiquette(gauche(droit(A3)))) # c
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. **Listes de listes**

→ paradigme fonctionnel

2. **Classe Noeud**

→ paradigme objet

CLASSE NOEUD

1. Arbre vide représenté par **None**

2. Nœud représenté par la classe suivante :

```
class Noeud:  
    arbre_vide = None  
  
    def __init__(self, etiquette, gauche, droit):  
        self._etiquette = etiquette  
        self._gauche = gauche  
        self._droit = droit  
  
    def etiquette(self):  
        return self._etiquette  
  
    def gauche(self):  
        return self._gauche  
  
    def droit(self):  
        return self._droit  
  
    def est_vide(arbre):  
        return arbre is Noeud.arbre_vide
```

CLASSE NOEUD

Exemple d'utilisation :

```
A1 = Noeud('r', Noeud.arbre_vide, Noeud.arbre_vide)
print(A1) # <__main__.Noeud object at 0x...>
print(A1.etiquette()) # r
print(Noeud.est_vide(A1.gauche())) # True

a = Noeud('a', Noeud.arbre_vide, Noeud.arbre_vide)
b = Noeud('b', Noeud.arbre_vide, Noeud.arbre_vide)
A2 = Noeud('r', a, b)
print(A2.gauche().etiquette()) # a
print(Noeud.est_vide(a.gauche())) # True

c = Noeud('c', Noeud.arbre_vide, Noeud.arbre_vide)
b = Noeud('b', c, Noeud.arbre_vide)
A3 = Noeud('r', a, b)
print(A3.droit().gauche().etiquette()) # c
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. **Listes de listes**

→ paradigme fonctionnel

2. **Classe Noeud**

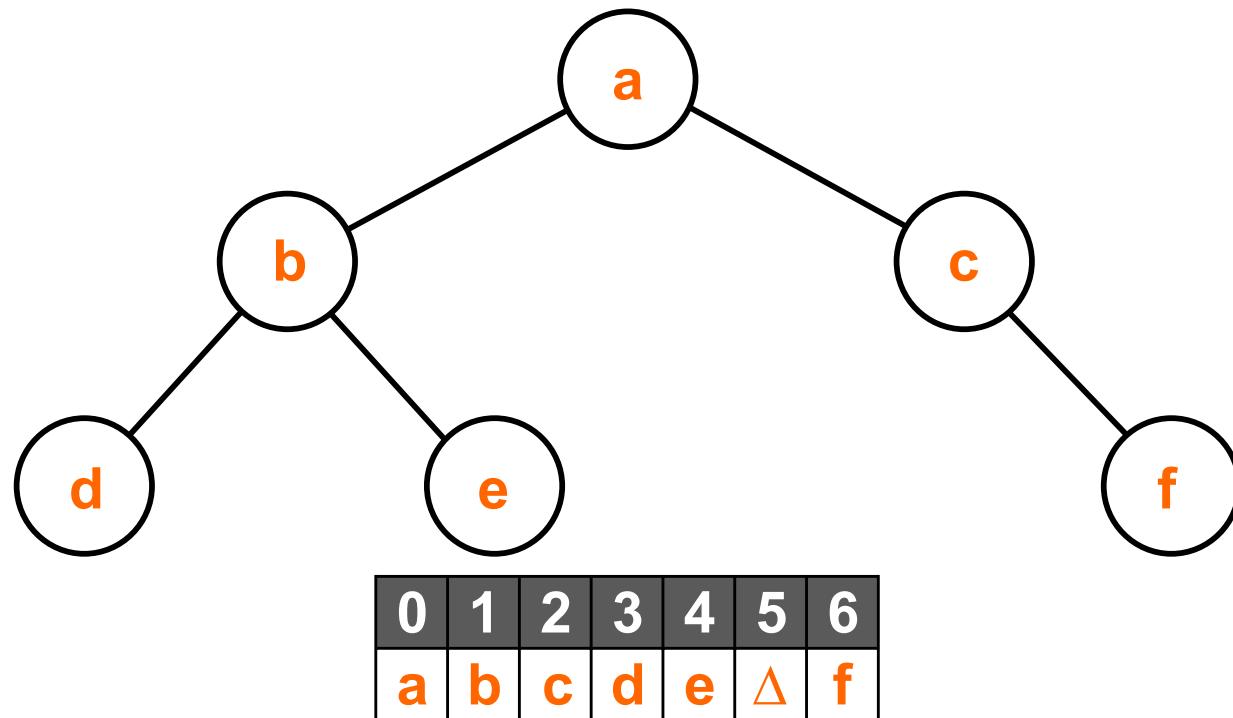
→ paradigme objet

3. **Liste unique** (méthode d'Eytzinger)

→ paradigme impératif

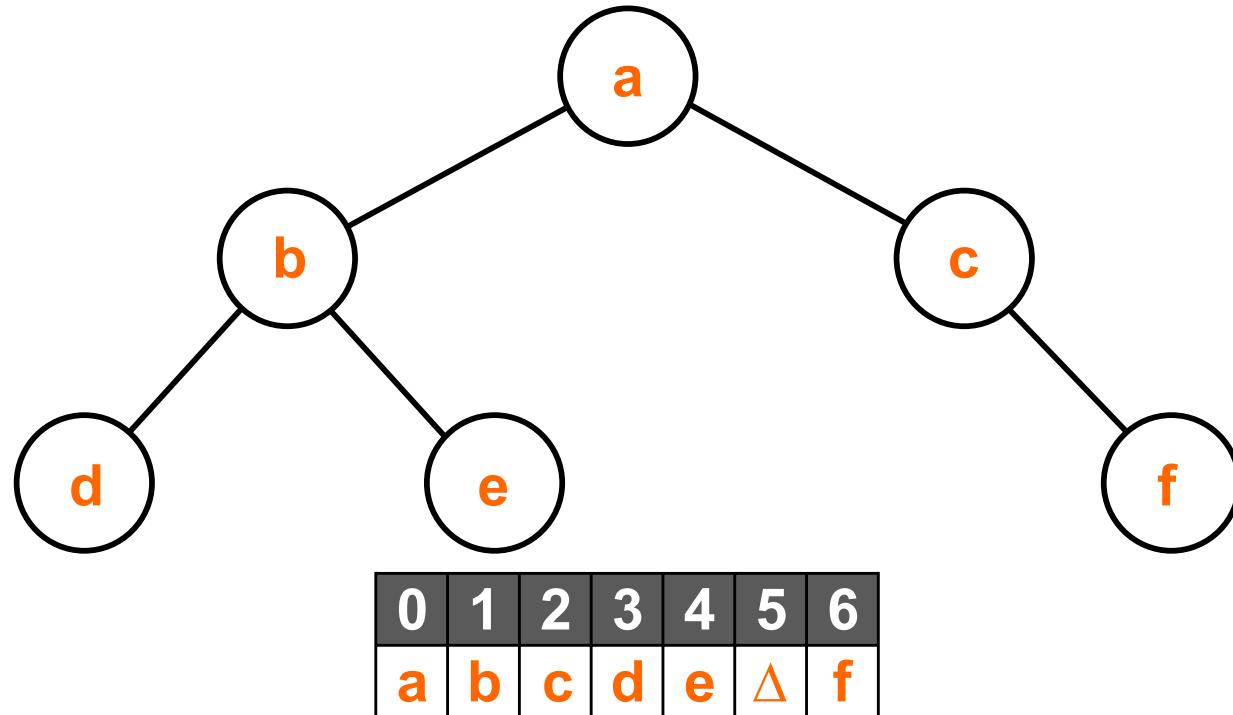
LISTE UNIQUE (MÉTHODE D'EYTZINGER)

Liste dans laquelle les fils gauche et droit d'un nœud i sont rangés respectivement dans les cases $2i+1$ et $2i+2$.



LISTE UNIQUE (MÉTHODE D'EYZINGER)

Liste dans laquelle les fils gauche et droit d'un nœud i sont rangés respectivement dans les cases $2i+1$ et $2i+2$.



Stockage mieux adapté aux **arbres complets**.

LISTE UNIQUE (MÉTHODE D'EYZINGER)

Version simple où la profondeur de l'arbre est fixée *a priori*.

```
profondeur_max = 2

def arbre_vide():
    return None

arbre = [arbre_vide()] * (2**profondeur_max+1)-1)

def noeud(etiquette, i):
    arbre[i] = etiquette

def etiquette(i):
    return arbre[i]

def gauche(i):
    return 2 * i + 1

def droit(i):
    return 2 * i + 2

def est_vide(i):
    return arbre[i] == arbre_vide()
```

LISTE UNIQUE (MÉTHODE D'EYZINGER)

Exemple d'utilisation :

```
noeud('r',0)
print(arbre) # ['r', None, None, None, None, None, None]
print(etiquette(0)) # r
print(est_vide(gauche(0))) # True

noeud('a', gauche(0))
noeud('b', droit(0))
print(arbre) # ['r', 'a', 'b', None, None, None, None]
print(etiquette(gauche(0))) # a
print(etiquette(droit(0))) # b

noeud('c', gauche(droit(0)))
print(arbre) # ['r', 'a', 'b', None, None, 'c', None]
print(etiquette(gauche(droit(0)))) # c
```

GRAPHES

- Structures de données :
 - **relationnelles**,
 - **séquentielles** et **récursives**.

GRAPHES

- Structures de données :
 - **relationnelles**,
 - **séquentielles** et **récursives**.
- Utilisées pour représenter une **relation** entre un **ensemble d'objets homogènes** :
 - réseaux routiers, de machines, sociaux, etc.
 - ordre d'exécution

GRAPHES

- Structures de données :
 - **relationnelles**,
 - **séquentielles** et **récursives**.
- Utilisées pour représenter une **relation** entre un **ensemble d'objets homogènes** :
 - réseaux routiers, de machines, sociaux, etc.
 - ordre d'exécution
- Utilisées dans de nombreux **algorithmes d'optimisation** :
 - ordonnancement de tâches
 - routage réseau
 - jeux

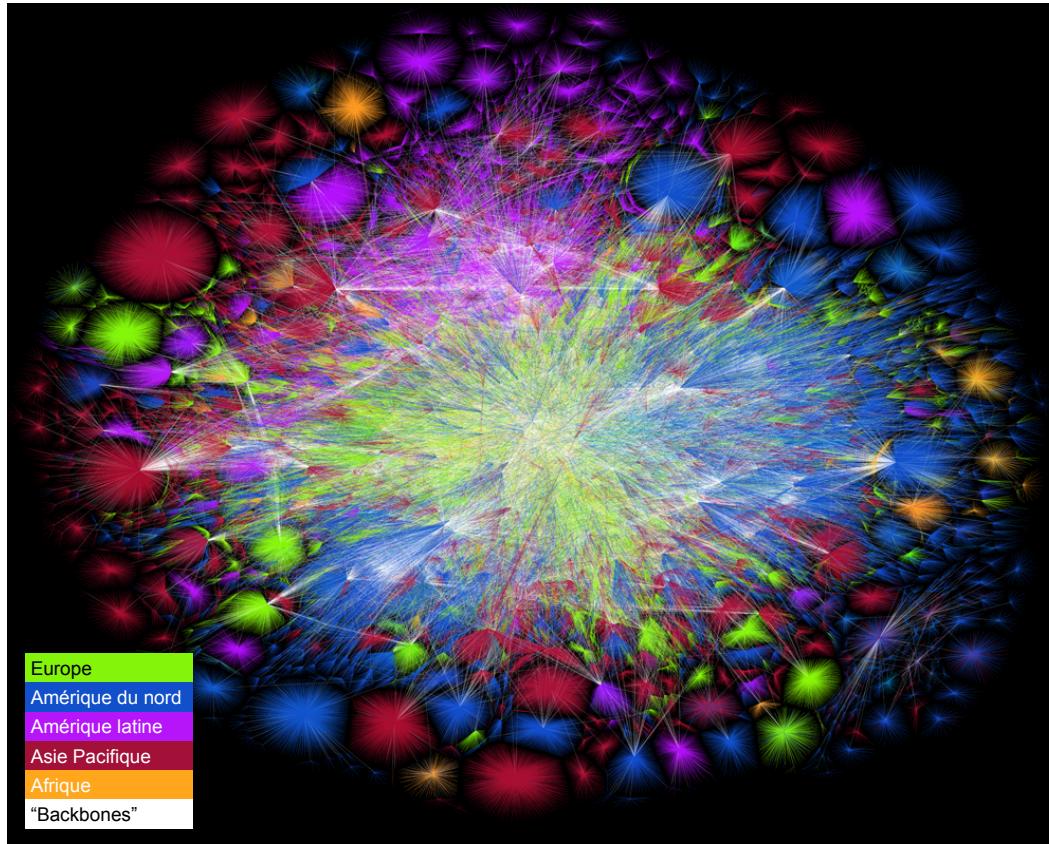
EXEMPLE : REPAS DE FAMILLE



Pixinio

- **Ensemble** : personnes
- **Relation** : cousin

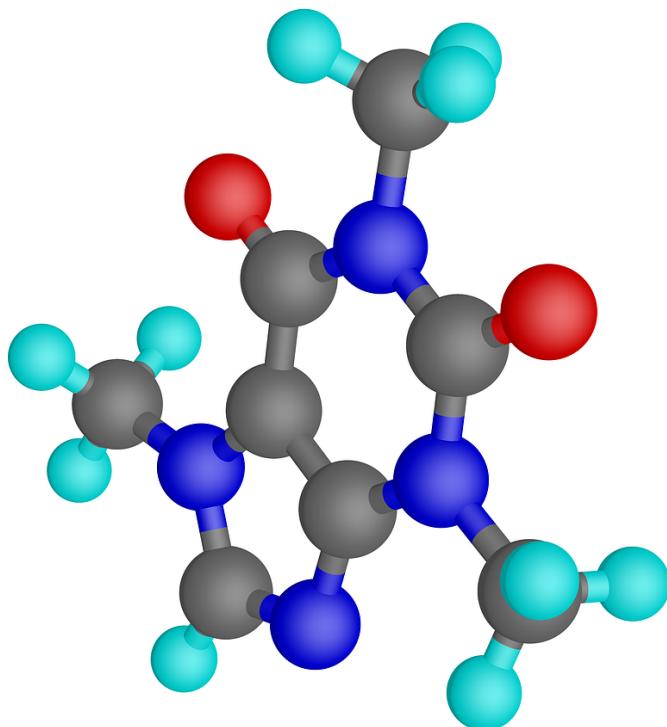
EXEMPLE : INTERNET



The OPTE project

- **Ensemble** : machines
- **Relation** : connexion réseau

EXEMPLE : MOLÉCULE

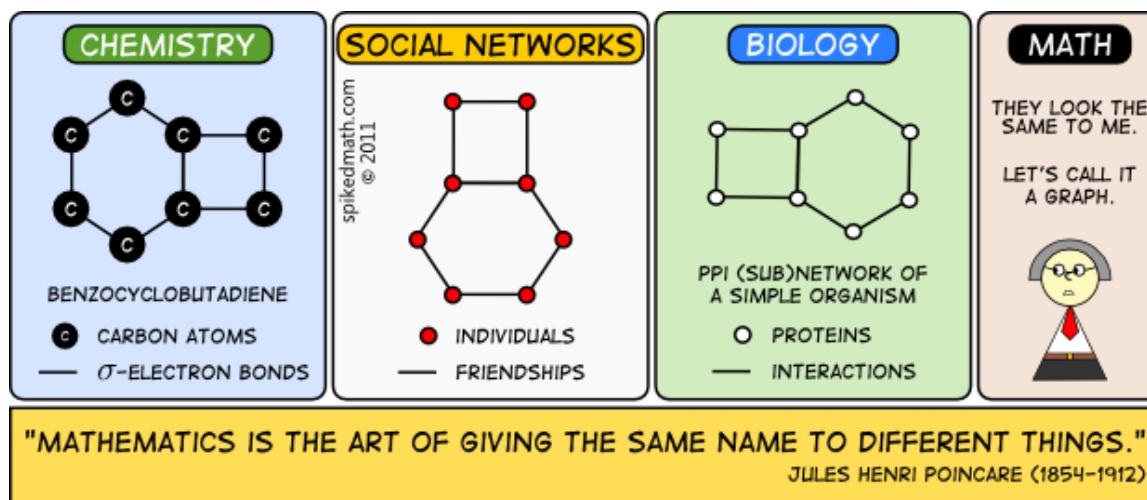


Molécule de caféine

- **Ensemble** : atomes
- **Relation** : liaison covalente

POURQUOI LES GRAPHS ?

Permettent de représenter différentes situations de la vie courante

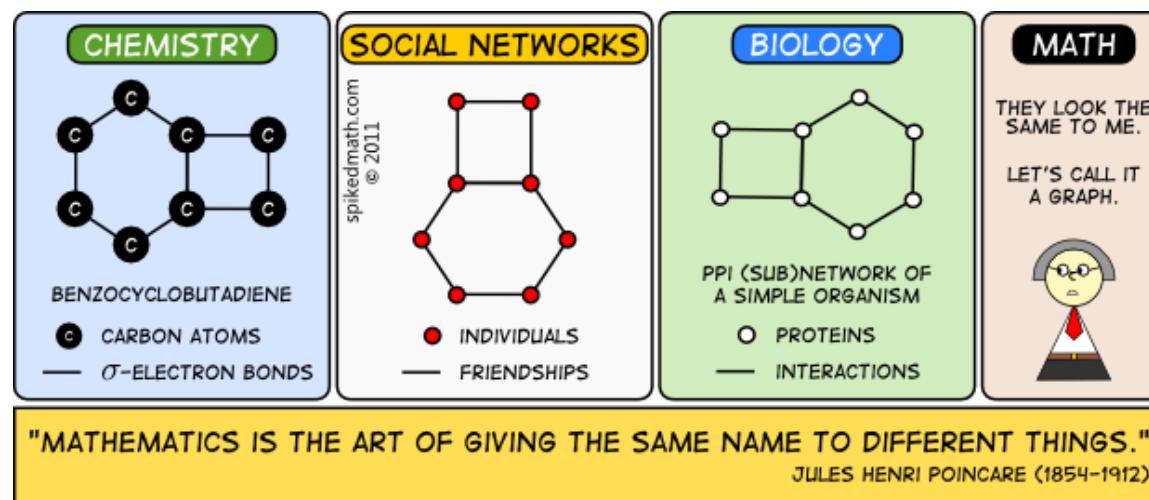


Spiked Math

POURQUOI LES GRAPHS ?

Permettent de représenter différentes situations de la vie courante

- **Problèmes de la vie courante**
 - questions à résoudre sur les graphes

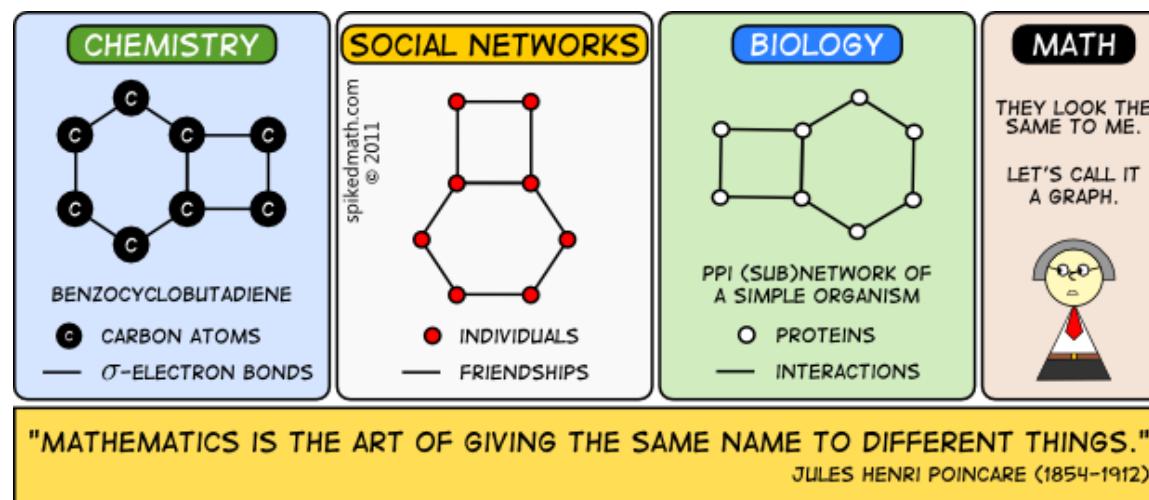


Spiked Math

POURQUOI LES GRAPHS ?

Permettent de représenter différentes situations de la vie courante

- **Problèmes de la vie courante**
 - questions à résoudre sur les graphes
- **Algorithme sur les graphes** (cf. Bloc 5)
 - réponse informatique à des problèmes concrets



Spiked Math

EXEMPLE : EXAMENS DE SPÉCIALITÉ

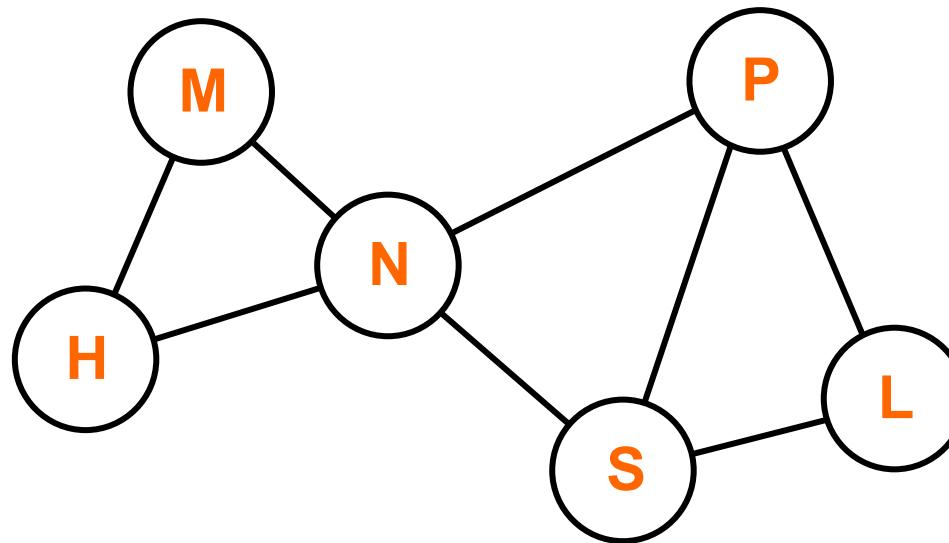
- Les élèves d'une classe de terminal passent leurs examens de spécialité, à choisir parmi : Maths (M), Humanité (H), NSI (N), Physique-Chimie (P), SVT (S) et Langues (L).
- Certains passent M, H, N, d'autres N, P, S et d'autres S, L, P.
- Tous les examens sont d'une durée de 2h.
- **Question :** Quelle est la durée la plus courte pour organiser ces examens ?

EXEMPLE : EXAMENS DE SPÉCIALITÉ

- **Ensemble** : examens {M, H, N, P, S, L}
- **Relation** : incompatibilité entre deux examens
 - si un élève doit passer deux des examens, les épreuves ne peuvent avoir lieu en même temps

EXEMPLE : EXAMENS DE SPÉCIALITÉ

- **Ensemble** : examens {M, H, N, P, S, L}
- **Relation** : incompatibilité entre deux examens
→ si un élèves doit passer deux des examens, les épreuves ne peuvent avoir lieu en même temps

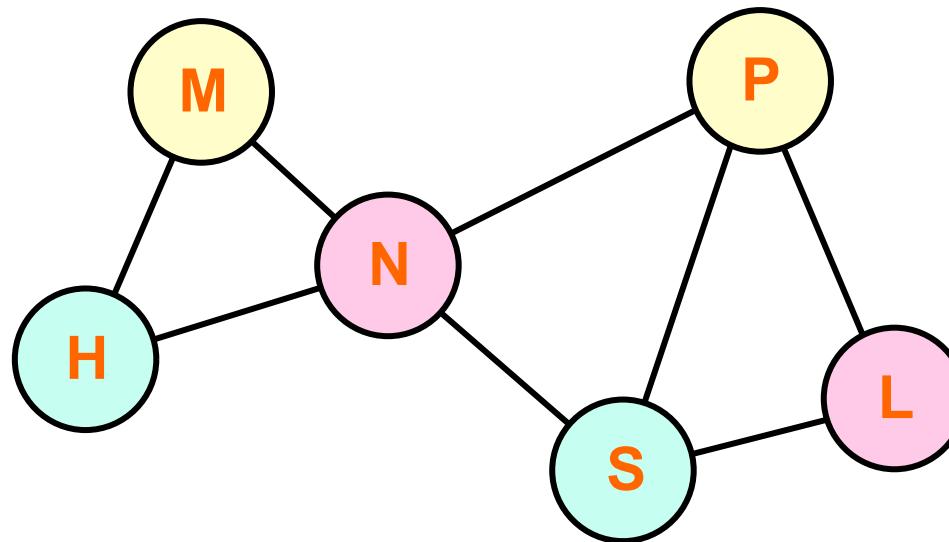


EXEMPLE : EXAMENS DE SPÉCIALITÉ

- On cherche à attribuer à chaque examen un créneau horaire
→ on veut **colorier** les sommets adjacents avec des couleurs différentes en utilisant un **minimum** de couleurs.

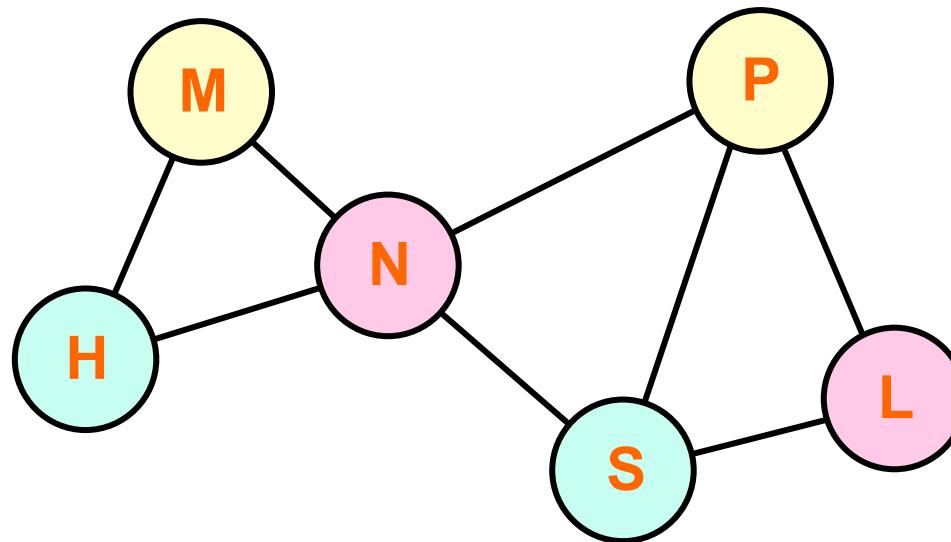
EXEMPLE : EXAMENS DE SPÉCIALITÉ

- On cherche à attribuer à chaque examen un créneau horaire
→ on veut **colorier** les sommets adjacents avec des couleurs différentes en utilisant un **minimum** de couleurs.



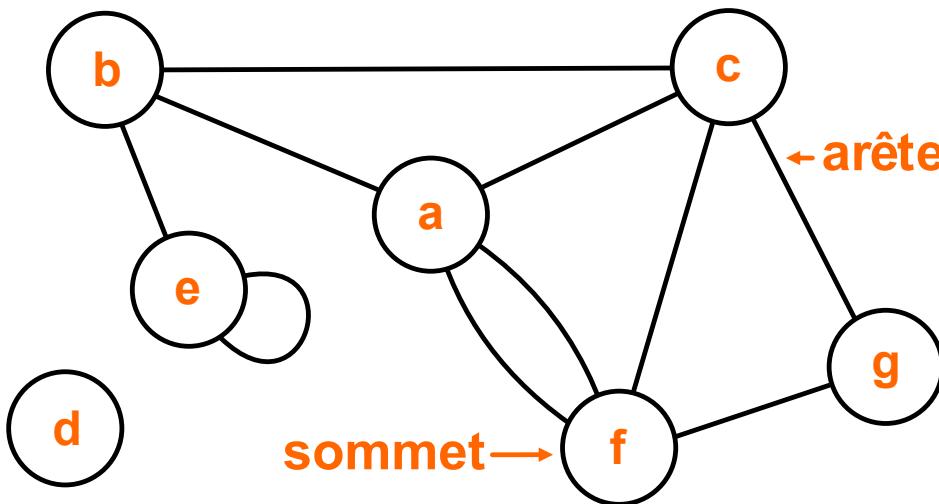
EXEMPLE : EXAMENS DE SPÉCIALITÉ

- On cherche à attribuer à chaque examen un créneau horaire
→ on veut **colorier** les sommets adjacents avec des couleurs différentes en utilisant un **minimum** de couleurs.



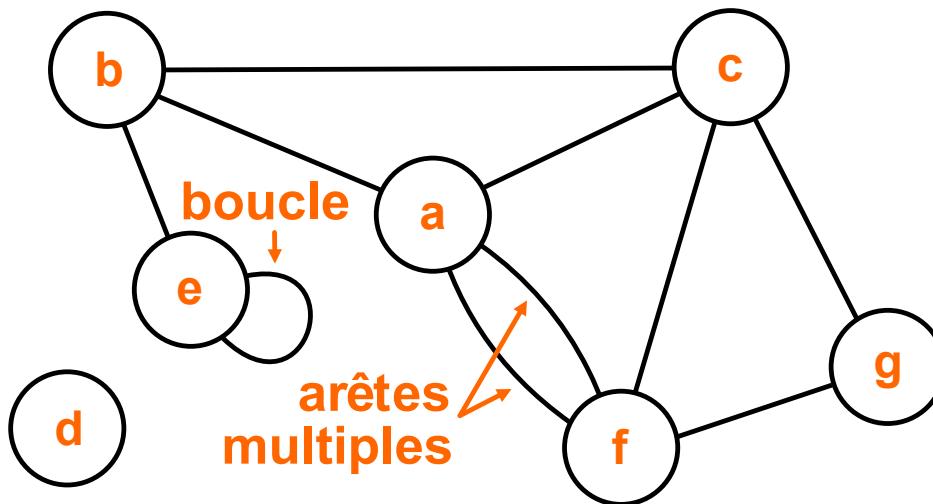
Solution : 3 couleurs → on peut organiser les examens en 3 x 2h.

TERMINOLOGIE



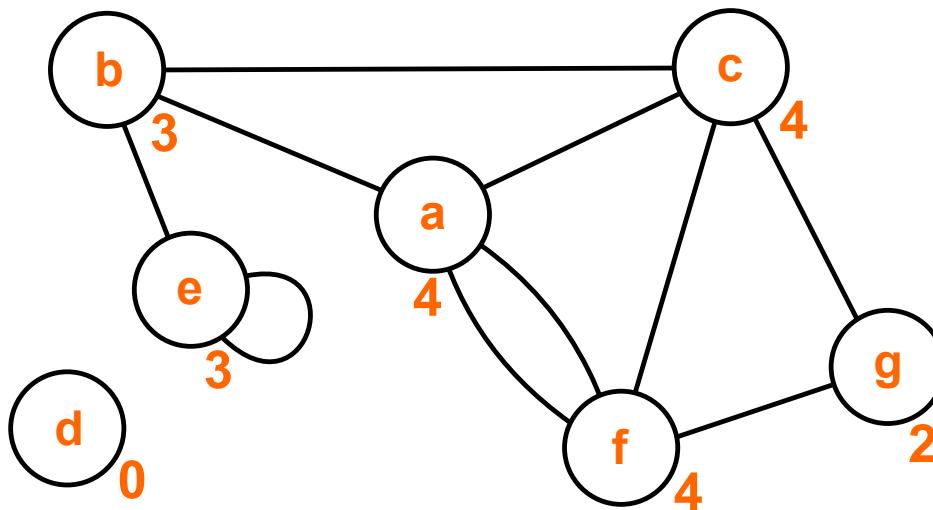
- un **sommet** est caractérisé par une donnée (ou étiquette).
- une **arête** relie deux sommets qui sont alors **voisins**.
- une arête peut porter une **valuation** (un poids, un coût).

TERMINOLOGIE



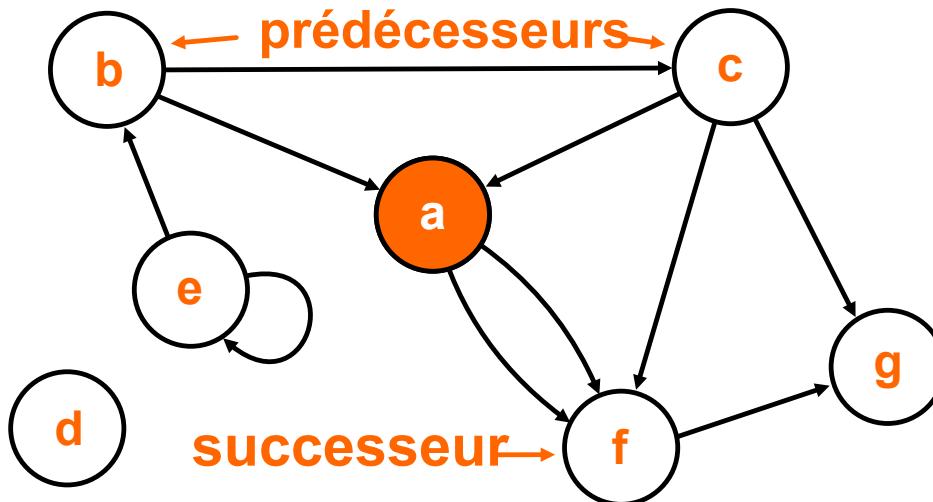
- une **boucle** est une arête reliant un sommet à lui-même.
- des **arêtes multiples** relient les mêmes sommets.

TERMINOLOGIE



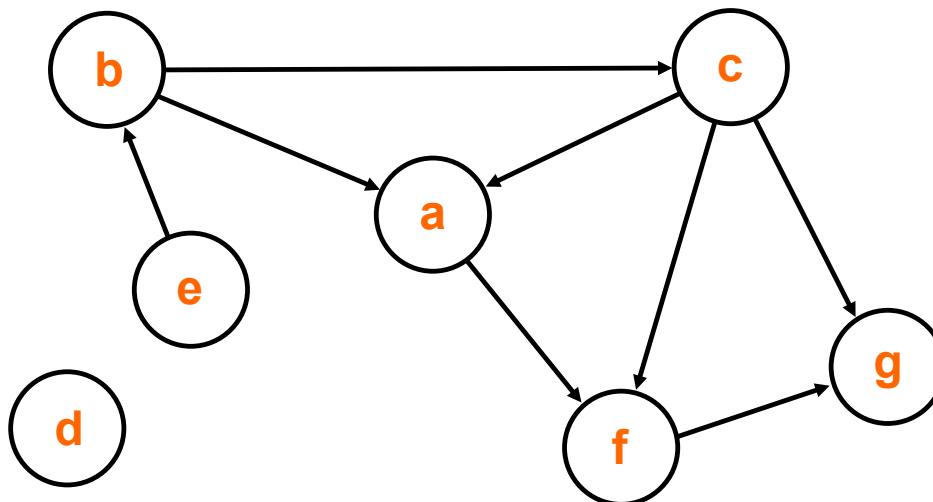
- le **degré** d'un sommet est le nombre d'arêtes qui lui sont incidentes.
- un sommet de degré zéro est dit **isolé**

TERMINOLOGIE



- les arêtes peuvent être **orientées**, on parle alors d'**arcs**.
- auquel cas, on définit l'ensemble des **successeurs** et **prédécesseurs** d'un sommet.

TERMINOLOGIE

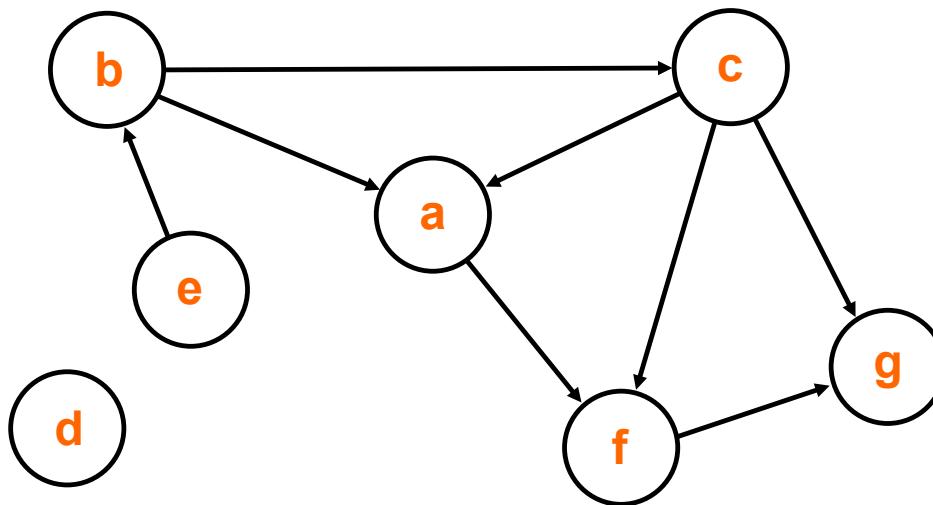


un graphe **simple** ne contient ni boucles, ni arêtes multiples.

DÉFINITION

- Un **graphe simple** est un couple (S, A) formé de :
 - un ensemble $S = \{x_1, x_2, \dots, x_n\}$ de sommets,
 - un ensemble $A = \{a_1, a_2, \dots, a_m\}$ d'arêtes tel que $\forall i, a_i = (x, y) \in S^2 \wedge x \neq y$.
- **Remarque :** pour un graphe orienté, l'arête (x, y) *part* du sommet x et *arrive* au sommet y .

EXEMPLE



- Graphe (S, A) avec :
 - $S = \{a, b, c, d, e, f, g\}$
 - $A = \{(e, b), (b, a), (b, c), (c, a), (c, f), (c, g), (a, f), (f, g)\}$

TYPE ABSTRAIT Graphe simple

1. Constructeur :

```
creer_graphe : liste de Sommets -> Graphe  
    # à partir de la liste des sommets S, produit le graphe (S,Δ)
```

TYPE ABSTRAIT Graphe simple

1. Constructeur :

```
creer_graphe : liste de Sommets -> Graphe  
    # à partir de la liste des sommets S, produit le graphe (S,Δ)
```

2. Opération :

```
ajouter_arete : (Graphe * Sommet * Sommet) -> Graphe  
    # à partir d'un graphe (S,A) et des sommets s1 et s2 appartenant à S,  
    # produit le graphe (S,A ∪ {(s1,s2)})
```

TYPE ABSTRAIT Graphe simple

1. Constructeur :

```
creer_graphe : liste de Sommets -> Graphe  
    # à partir de la liste des sommets S, produit le graphe (S,Δ)
```

2. Opération :

```
ajouter_arete : (Graphe * Sommet * Sommet) -> Graphe  
    # à partir d'un graphe (S,A) et des sommets s1 et s2 appartenant à S,  
    # produit le graphe (S,A ∪ {(s1,s2)})
```

3. Sélecteurs :

```
sommets : Graphe -> liste de Sommets  
    # à partir d'un graphe (S,A), produit la liste de sommets S  
voisins : (Graphe * Sommet) -> liste de Sommets  
    # à partir du graphe (S,A) et du sommet s,  
    # produit la liste des voisins de s
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Matrice d'adjacence

→ paradigme impératif

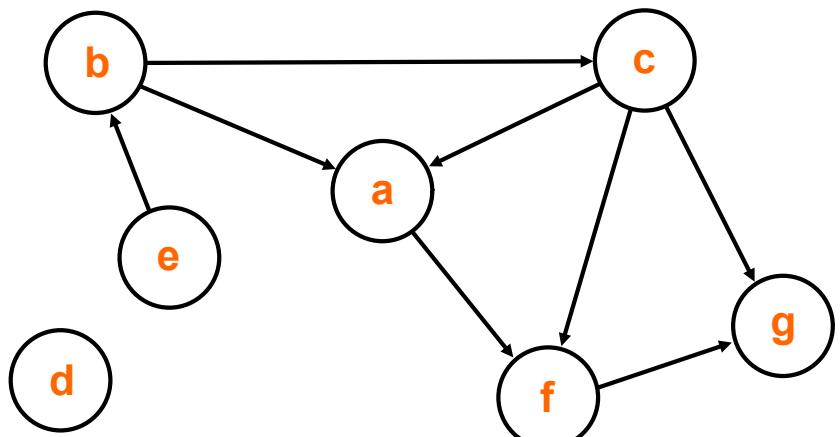
MATRICE D'ADJACENCE

Les arêtes sont représentées par un **tableau 2D** (matrice) :

- **lignes** (d'indice i) : tous les sommets
- **colonnes** (d'indice j) : tous les sommets
- **case(i,j)** :
 - 1 (ou sa valuation) si il existe un arc (une arête) partant du sommet i et arrivant au sommet j,
 - 0 sinon.

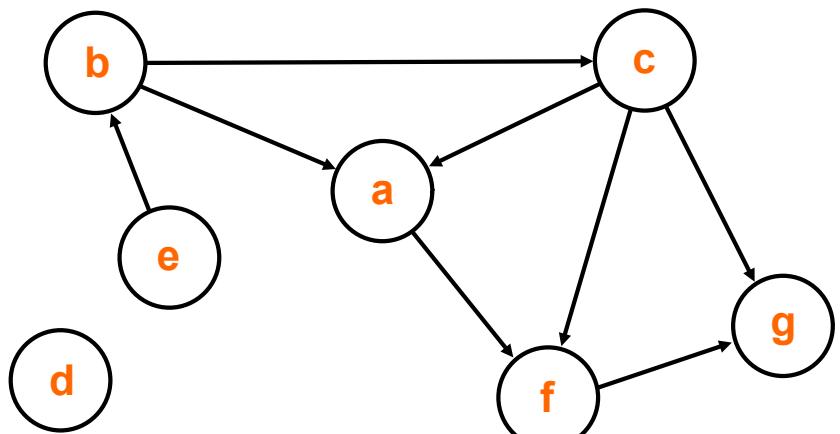
Remarque : dans le cas non-orienté, le tableau est **symétrique**
→ $\text{case}(i, j) = \text{case}(j, i)$

MATRICE D'ADJACENCE



	a	b	c	d	e	f	g
a							
b							
c							
d							
e							
f							
g							

MATRICE D'ADJACENCE



	a	b	c	d	e	f	g
a	0	0	0	0	0	1	0
b	1	0	1	0	0	0	0
c	1	0	0	0	0	1	1
d	0	0	0	0	0	0	0
e	0	1	0	0	0	0	0
f	0	0	0	0	0	0	1
g	0	0	0	0	0	0	0

MATRICE D'ADJACENCE

```
def creer_graphe(sommets):
    dimension = len(sommets)
    adjacence = [[0 for i in range(dimension)] for j in range(dimension)]
    return (sommets,adjacence)

def ajouter_arete(graphe, s1, s2):
    i = graphe[0].index(s1)
    j = graphe[0].index(s2)
    graphe[1][i][j] = 1
    graphe[1][j][i] = 1 # si graphe non-orienté
    return graphe

def sommets(graphe):
    return graphe[0]

def voisins(graphe, s):
    i = sommets(graphe).index(s)
    voisins = []
    for j in range(len(graphe[1][i])):
        if graphe[1][i][j] == 1:
            voisins += [sommets(graphe)[j]]
    return voisins
```

MATRICE D'ADJACENCE

Exemple d'utilisation :

```
G = creer_graphe(['a', 'b', 'c', 'd'])
ajouter_arete(G, 'a', 'b')
ajouter_arete(G, 'a', 'c')
print(G) # ([ 'a', 'b', 'c', 'd' ],
           # [[0, 1, 1, 0],
            # [1, 0, 0, 0],
            # [1, 0, 0, 0],
            # [0, 0, 0, 0]])
print(voisins(G, 'a')) # ['b', 'c']
print(voisins(G, 'c')) # ['a']
print(voisins(G, 'd')) # []
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Matrice d'adjacence

→ paradigme impératif

2. Listes de successeurs (ou de prédecesseurs)

→ paradigme fonctionnel

LISTES DE SUCCESEURS (OU DE PRÉDÉCESSEURS)

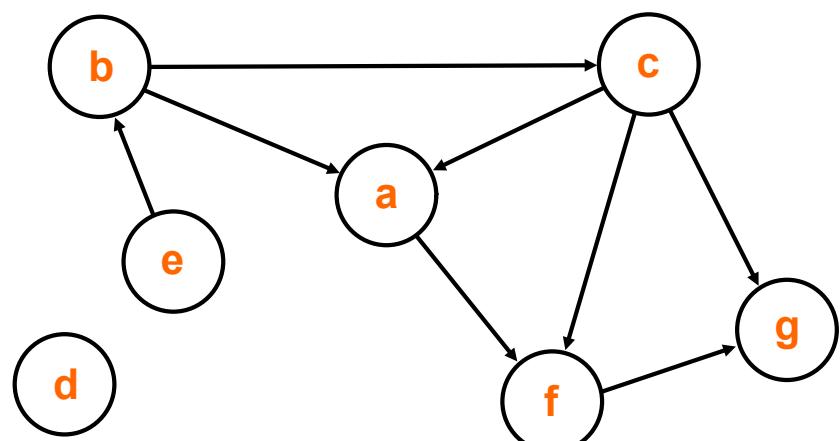
Les arêtes sont représentées par un **dictionnaire** de **listes** :

- **clés** du dictionnaire : tous les sommets
- **liste** associée à une clé : successeurs (ou prédecesseurs) de ce sommet

Remarques :

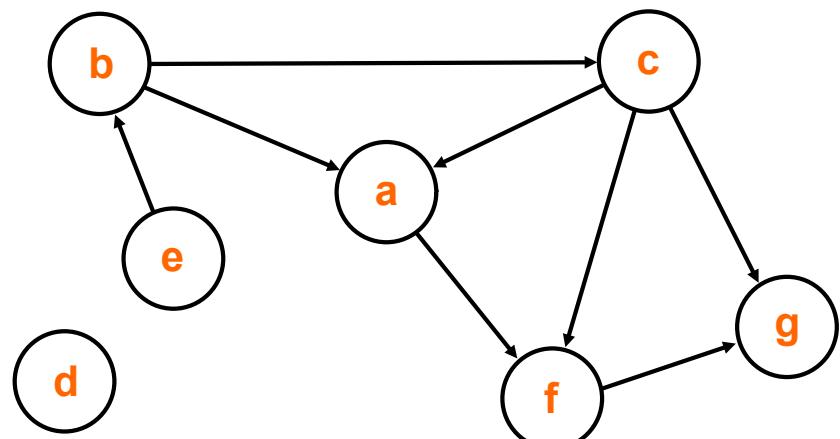
1. dans le cas non-orienté, listes des **voisins** du sommet
2. pour les arcs/arêtes valués, dictionnaire de dictionnaires

LISTES DE SUCCESSEURS



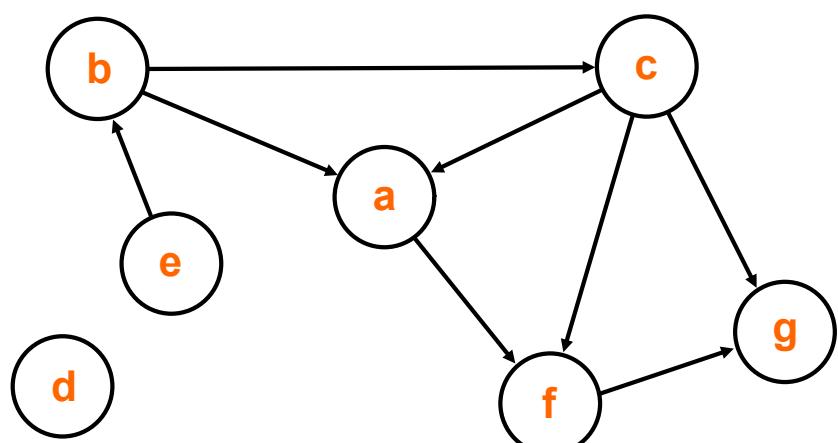
- a :
- b :
- c :
- d :
- e :
- f :
- g :

LISTES DE SUCCESSEURS



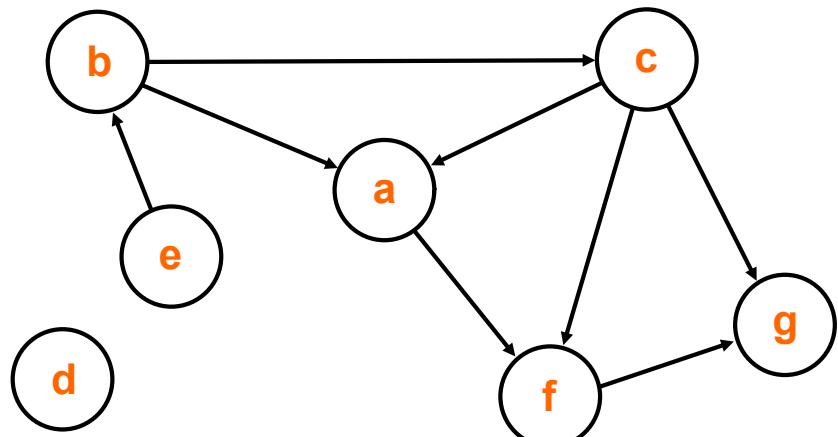
- **a** : [f]
- **b** : [a,c]
- **c** : [a,f,g]
- **d** : []
- **e** : [b]
- **f** : [g]
- **g** : []

LISTES DE PRÉDÉCESSEURS



- a :
- b :
- c :
- d :
- e :
- f :
- g :

LISTES DE PRÉDÉCESSEURS



- **a** : [b,c]
- **b** : [e]
- **c** : [b]
- **d** : []
- **e** : []
- **f** : [a,c]
- **g** : [c,f]

LISTES DE SUCCESSEURS

```
def creer_graphe(sommets):
    return {key: [] for key in sommets}

def ajouter_arete(graphe, s1, s2):
    graphe[s1].append(s2)
    graphe[s2].append(s1) # si graphe non-orienté

def sommets(graphe):
    return list(graphe.keys())

def voisins(graphe, sommet):
    return graphe[sommet]
```

LISTES DE SUCCESSEURS

Exemple d'utilisation :

```
G = creer_graphe(['a', 'b', 'c', 'd'])
ajouter_arete(G, 'a', 'b')
ajouter_arete(G, 'a', 'c')
print(G) # {'a': ['b', 'c'], 'b': ['a'], 'c': ['a'], 'd': []}
print(voisins(G, 'a')) # ['b', 'c']
print(voisins(G, 'c')) # []
```

MISE EN OEUVRE EN PYTHON

Plusieurs implémentations possibles :

1. Matrice d'adjacence

→ paradigme impératif

2. Listes de successeurs (ou de prédecesseurs)

→ paradigme fonctionnel

Remarque : sommets et/ou arêtes peuvent être représentés par des classes (paradigme objet), si nécessaire.

COMPARAISON

	matrice d'adjacence	liste de successeurs
mémoire		
peu d'arêtes		
bcp d'arêtes		
temps de calcul		
x et y voisins ?		
x isolé ?		
nb arêtes ?		

avec n le nombre de sommets et m le nombre d'arêtes.

COMPARAISON

	matrice d'adjacence	liste de successeurs
mémoire	n^2	$n + m$
peu d'arêtes	✗	✓
bcp d'arêtes	✓	✗
temps de calcul		
x et y voisins ?		
x isolé ?		
nb arêtes ?		

avec n le nombre de sommets et m le nombre d'arêtes.

COMPARAISON

	matrice d'adjacence	liste de successeurs
mémoire	n^2	$n + m$
peu d'arêtes	✗	✓
bcp d'arêtes	✓	✗
temps de calcul		
x et y voisins ?	1	degré de x
x isolé ?	n	1
nb arêtes ?	n^2	n ou n^2

avec n le nombre de sommets et m le nombre d'arêtes.