

Table des matières

Définitions de POO	3
Paradigme	3
Abstraction	3
Polymorphisme	3
Classe	3
Propriété, méthode, objet	4
Encapsulation	4
Délégation	5
Constructeur	5
Héritage	6
Surcharge et redéfinition	6
Unified Modeling Language	8
Types de diagrammes en UML	8
Système	9
Modèle	9
Cycle de vie d'un projet	10
Diagramme de Classes	10
Cardinalités	11
Types de liaisons	12
Comparaison Classe <i>Abstraite / Concète</i>	14
Diagramme de cas d'utilisation	15
Include/Extend	15
Diagramme de Séquences	17
Algorithmie	21
Les arbres	21
Les arbres binaires	21
Arbres binaires de recherche	24
Tri par tas	27
Les tables de hachage	28
Pathfinding	31
La théorie des graphes	31
Dijkstra	33
Floyd-Warshal	34
Autres Algorithmes de Graphes	34

Heuristique.....	35
Théorie des jeux	36
Chiffrement	37
CHIFFREMENT SYMETRIQUE.....	38
CHIFFREMENT ASYMETRIQUE.....	39
RegEx	39

Définitions de POO

Paradigme

- ♥ Façon de penser, réfléchir à la résolution d'un problème, le code Informatique.

ex. POO, programmation procédurale (étape par étape) etc.

Réflexion ⇒ implémentation

Abstraction

- ♥ Concentration sur ce que représente un objet et sur son comportement avant de décider de la façon de l'implémenter.

Polymorphisme

- ♥ La capacité d'une entité (comme une variable, une fonction, ou un objet) à prendre plusieurs formes. En pratique, cela permet à des éléments de code tels que des variables, des fonctions ou des objets, d'agir de manière différente selon le contexte dans lequel ils sont utilisés.
- ♥ **Polymorphisme ad hoc** : Cela implique des méthodes ayant le même nom mais se comportant différemment en fonction des arguments qui leur sont passés. La surcharge de fonctions et d'opérateurs en est un exemple.
- ♥ **Polymorphisme paramétrique** : Cela se produit quand une fonction ou une classe peut travailler avec n'importe quel type de données. Cela est souvent réalisé à l'aide de génériques, où le type exact peut ne pas être spécifié lors de la création de la classe ou de la fonction, mais sera déterminé lors de son utilisation.

Classe

- ♥ Abstraction décrivant un ensemble d'objets potentiellement infini
- ♥ **Instance d'une classe** – objet appartenant à la classe
- ♥ **Conception des classes** - concentration sur les structures de données et algorithmes de chaque classe. Une classe peut admettre plusieurs conceptions.
- ♥ **Signature de la classe** - nom de la fonction et type des paramètres

Classe Interface

- ♥ Une **classe interface** est un type de classe qui ne contient que des déclarations de méthodes sans aucune implémentation, toutes les méthodes sont abstraites. Cela signifie qu'elle spécifie ce qu'une classe doit faire, mais pas comment le faire. Notion de contrat entre l'interface et les classes qui l'implémente. Les méthodes DOIVENT être utilisées par la classe qui utilise ce contrat.

ex. une interface "Véhicule" pourrait déclarer une méthode "démarrer" mais ne fournirait pas de détails sur la façon dont le véhicule démarre. C'est aux classes qui implémentent l'interface "Véhicule" de définir comment elles démarrent.

- ♥ Une interface ne peut contenir ni attribut, ni méthode implémentée. Le terme d'héritage n'est pas utilisé entre classes et interfaces : on dit qu'une classe implémente une interface.

Classe abstraite

- ♥ Une classe qui peut contenir à la fois des méthodes avec des implémentations (comportements définis) et des méthodes sans implémentations (méthodes abstraites). Les classes abstraites ne peuvent pas être instanciées directement, c'est-à-dire que vous ne pouvez pas créer d'objet directement à partir d'une classe abstraite.

Toutefois, pour définir une interface en Python, on peut utiliser les "Abstract Base Classes" (ABC) du module abc. Les ABC permettent de définir des méthodes abstraites qui doivent être implémentées par les sous-classes, offrant ainsi une manière de créer des interfaces.

```
from abc import ABC, abstractmethod

class Teacher(ABC):

    #normal method
    def display(self):
        #method definition
        print('I am a Teacher')

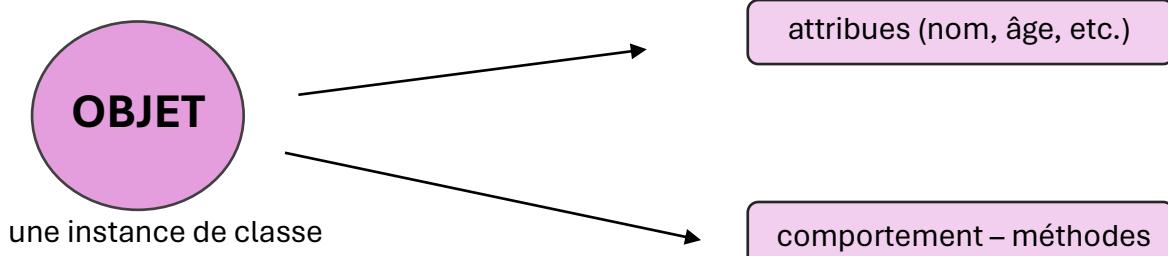
    @abstractmethod
    def abs_method(self):

        #Abs_method definition
        None
```

Propriété, méthode, objet

- ♥ **Propriété** –manière de manipuler un attribut, le niveau d'accès ⇒

- {readOnly}
- {redefines nomAttribut}
- {ordered}
- {unique}
- {nonunique}



- ♥ **Méthodes** – opération d'une classe, changent l'état ou affichent des informations
- ♥ **Méthodes de classe** – fonctions d'une classe particulière
- ♥ **Méthodes statiques** - indépendantes des instances de la classe, elles ne pourront pas manipuler les variables d'instances ou accéder aux méthodes non statiques

Encapsulation

- ♥ **Masquage de l'information** : séparation des parties externes d'un objet accessibles aux autres objets en définissant des **méthodes**
- ♥ **Getter** - méthode d'accès pour obtenir la valeur d'un attribut
- ♥ **Setter** - méthodes d'altération pour modifier la valeur d'un ou plusieurs attributs

```
@property
def name(self) -> str:
    """ GETTER. """
    return self._name

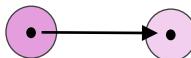
@name.setter
def name(self, value: str) -> None:
    """ SETTER. """
    self._name: str = value
```

- ♥ Permet de définir des **niveaux de visibilité** des éléments de la classe. Ces niveaux de visibilité définissent **les droits d'accès** aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque.
- ♥ Par défaut, les attributs d'une classe sont privés et ses méthodes publiques
 - **public** (+)- les méthodes et attributs sont accessibles aisément par l'utilisateur de l'objet, mais également par les instances de la classe elle-même
 - **_protected** (#)- les méthodes et attributs sont accessibles uniquement depuis les instances de la classe elle-même et ses classes héritées (on préfère ce niveau pour pouvoir communiquer avec les classes héritées)
 - **__private** (-) - les méthodes et attributs sont accessibles uniquement depuis les instances de la classe elle-même

Délégation

- ♥ Principe d'utilisation de méthode d'un autre objet au lieu d'utiliser sa propre méthode

ex. bouger un point pour bouger le cercle



Constructeur

- ♥ Une fonction particulière appelée lors de l'instanciation. Elle permet d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs.
 - La première méthode de la classe
 - Nom de la classe

```
class Line:
    def __init__(self, p1, p2):
        self.line = (p1, p2)
```
- ♥ **Constructeur pas défaut** - ne prend aucun argument. Il est appelé lorsqu'un objet est créé sans fournir de paramètres spécifiques. Si aucun constructeur n'est explicitement défini dans une classe, la plupart des langages de programmation fournissent automatiquement un constructeur par défaut qui initialise l'objet avec des valeurs par défaut.

ex.

```
class Exemple {
    Exemple() {
        // Corps du constructeur par défaut
```

- ♥ **Constructeur par recopie** - est un type de constructeur qui initialise un nouvel objet en utilisant les valeurs d'un autre objet de la même classe. Il prend en argument une référence à un objet de la même classe

ex.

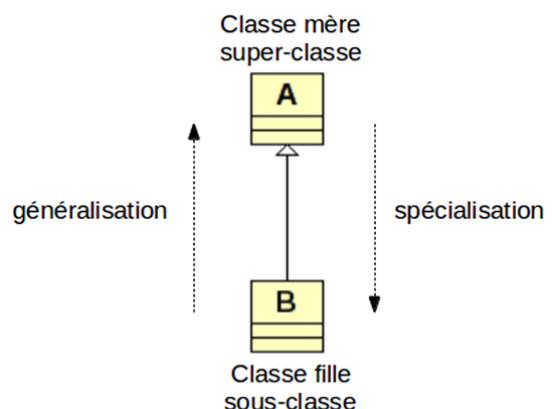
```
class Exemple {
    Exemple(Exemple autreObjet) {
        // Initialisation de cet objet en utilisant 'autreObjet'
```

- ♥ **Constructeur paramétrique** - prend un ou plusieurs arguments et utilise ces arguments pour initialiser l'objet. Cela permet de créer des objets avec des états initiaux spécifiques définis par les valeurs passées en paramètres au moment de la création.

```
ex. class Exemple {
    Exemple(int param1, String param2) {
        // Initialisation de l'objet en utilisant param1 et param2
    }
}
```

Héritage

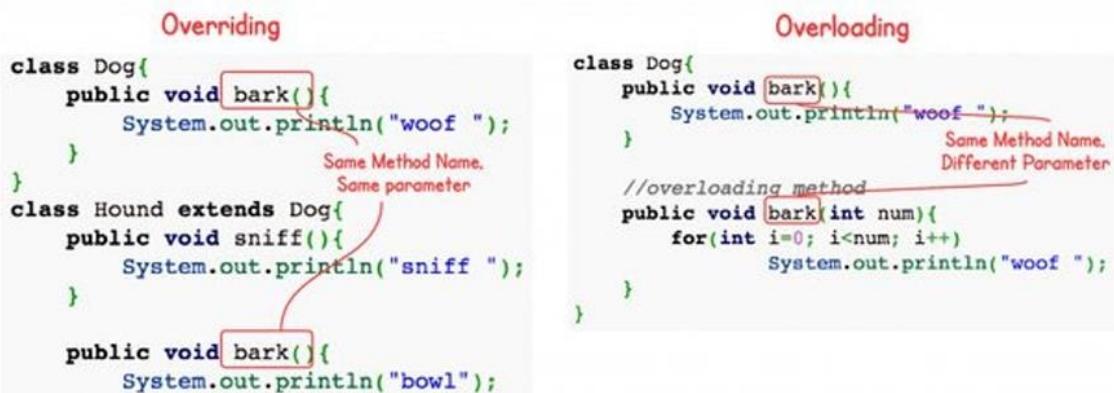
- ♥ Permet à une classe d'hériter des caractéristiques (attributs et méthodes) d'une autre classe.
- ♥ La classe enfant est une spécialisation de la classe parent. Classe fille qui hérite d'une classe mère.
- ♥ **Héritage d'Implémentation:** Une classe enfant hérite directement de la classe parente, y compris les méthodes et attributs avec leur code. La classe enfant peut utiliser ces méthodes telles quelles, les modifier (redéfinition) ou ajouter de nouvelles méthodes et attributs.
- ♥ **Héritage(implémentation) d'Interface:** Une classe s'engage à implémenter les méthodes définies par une interface, mais l'interface ne fournit pas d'implémentation (code). La classe doit fournir le code pour toutes les méthodes déclarées dans l'interface. Cela permet de définir un **contrat** que la classe doit respecter, favorisant ainsi une certaine forme de polymorphisme.



contrat claire

Surcharge et redéfinition

- ♥ **Surcharge, «overloading», ad hoc** - consiste à déclarer, dans une même classe, deux méthodes de même nom mais avec le comportement différent. **changements via les paramètres**
 - Même nom de méthode, Paramètres différents (soit sur le nombre ou le/les type(s))
 - Le type de retour n'est pas pris en compte
 - L'aspect de la «surcharge» est retrouvé lors de la déclaration de plusieurs constructeurs dans une même classe.
- ♥ **La redéfinition «overriding»** - est un concept où une classe enfant fournit une implémentation spécifique pour une méthode qui est déjà définie dans sa classe mère. **changements de code**
 - Le même nom de méthode
 - Même type de retour
 - Même paramètre(nombre et type)



Définition d'un algorithme (et preuves)

Méthode de résolution d'un problème de manière systématique. Même résultats dans les mêmes conditions à chaque fois.

- Pas d'initiative
- Eviter les ambiguïtés
- Langage de programmation (clair)

Preuves...

1. Terminaison

L'algorithme se termine en un temps fini

2. La correction

Le résultat est une solution au problème

3. La complétude

Pour une classe de problème, l'algo donne bien l'ensemble des solutions.

Unified Modeling Language

adopté en 1997 par l'Object Management Group(OMG)

- une manière standardisée de visualiser la conception d'un système à l'aide des schémas simples et compréhensibles par tout le monde

Types de diagrammes en UML

- ♥ **Diagrammes de cas d'utilisation** (Use Case Diagrams) : Ces diagrammes décrivent les fonctionnalités du système du point de vue des utilisateurs. Ils montrent les interactions entre les utilisateurs (acteurs) et les différentes parties du système.
- ♥ **Diagrammes de classes** (Class Diagrams) : Ils représentent la structure statique d'un système en montrant les classes du système, leurs attributs, méthodes et les relations entre ces classes, comme l'héritage, l'association, la composition et l'agrégation.
- ♥ **Diagrammes de séquence** (Sequence Diagrams) : Ces diagrammes montrent comment les objets interagissent dans une certaine séquence de temps, illustrant ainsi le déroulement des processus.
- ♥ **Diagrammes d'objets** (Object Diagrams) : Similaires aux diagrammes de classes, mais focalisés sur les instances d'objets à un moment précis.
- ♥ **Diagrammes d'activité** (Activity Diagrams) : Ils représentent les flux de travail ou les activités d'un processus ou d'une fonctionnalité, en montrant le flux de contrôle d'une activité à l'autre.
- ♥ **Diagrammes de composants** (Component Diagrams) : Ces diagrammes décrivent l'organisation et les dépendances entre les composants logiciels d'un système.
- ♥ **Diagrammes de déploiement** (Deployment Diagrams) : Ils détaillent l'emplacement physique et la distribution des composants logiciels sur l'infrastructure matérielle.
- ♥ **Diagrammes d'états-transition** (State Diagrams) : Ces diagrammes illustrent les états d'un objet au cours de son cycle de vie et les transitions d'état en réponse à des événements.
- ♥ **Diagrammes de packages** (Package Diagrams) : Utilisés pour regrouper des éléments UML similaires, comme des classes ou des activités, en paquets, facilitant ainsi l'organisation et la gestion du modèle.
- ♥ **Diagramme de Timing** (Timing Diagram) : Ce diagramme est une forme spécifique de diagramme de séquence où l'accent est mis sur les contraintes de temps. Il montre l'évolution de l'état ou de la condition d'un objet en fonction du temps. C'est particulièrement utile pour modéliser les systèmes embarqués et les systèmes en temps réel où le timing est crucial.
- ♥ **Diagramme de Communication** (Communication Diagram) : Anciennement connu sous le nom de diagramme de collaboration, ce diagramme illustre les interactions entre les objets en se concentrant sur les messages échangés. Contrairement aux diagrammes de séquence qui montrent l'ordre temporel des messages, les diagrammes de communication mettent en évidence les relations entre les objets qui participent à l'interaction.
- ♥ **Vue d'ensemble des Interactions** (Interaction Overview Diagram) : Ce diagramme est une variante du diagramme d'activité qui montre le flux de contrôle d'une interaction à une autre. Il combine des éléments de diagramme de séquence et de diagramme d'activité pour offrir une vue d'ensemble du flux d'interactions. Cela permet de visualiser les enchaînements complexes d'interactions au sein d'un système.

♥ Structure composite

Aprofondir le mode de fonctionnement des classes, méthodes,...

Aller dans les détails du système.

♥ Best practices

- Utiliser noms clairs et descriptif
- Cohérence dans les notations
- Adaptation du grain selon d'audience

♥ 10 points méthodes UML...

- Commencer par le besoin (clarifier le problème à résoudre.)
- Sélectionner les diagrammes pertinents (Choisir le bon type de diagramme)
- Garder les diagrammes simples (pas surcharger les diagrammes. Doit rester visible)
- Utiliser conventions de nomination consistantes (Adopter les conventions...)
- Préciser les relations (héritages, association, agrégation, composition...)
- Utiliser stéréotypes et commentaire
- Eviter redondances
- Valider avec les parties prenantes (que les diagrammes répondent bien aux besoins)
- Utiliser la dernière version UML, rester à jour.
- Se former et Utiliser les bonnes ressources. (Améliorer ses compétences en se formant)

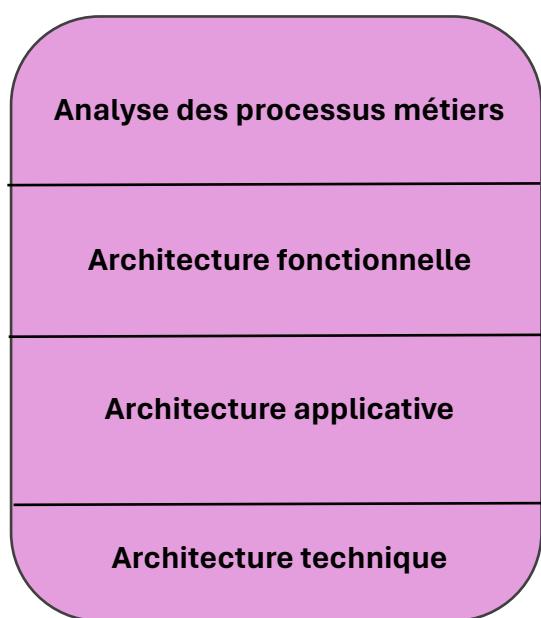
Système

- ♥ Est composé de différentes parties ayant des relations. Le tout est constitué de la sorte pour atteindre un but bien précis. Un système peut par exemple être un logiciel, du matériel, un processus de traitement de l'information, une entreprise, un processus métier, etc.

Modèle

- ♥ Peut représenter un système à n'importe quel niveau d'abstraction, allant de l'architecture à l'implémentation technologique.

Modélisation d'un SI



Cette étape consiste à comprendre et à documenter les processus commerciaux ou organisationnels que le système d'information doit soutenir. ex. diagrammes de processus métiers qui décrivent les activités, les flux de travail et les règles métiers.

Après avoir analysé les processus métiers, l'architecture fonctionnelle est définie. Elle décrit les fonctions de haut niveau que le système doit exécuter et la manière dont ces fonctions interagissent entre elles. ex. diagrammes de cas d'utilisation qui montrent les différents utilisateurs (acteurs) et les actions qu'ils peuvent effectuer avec le système.

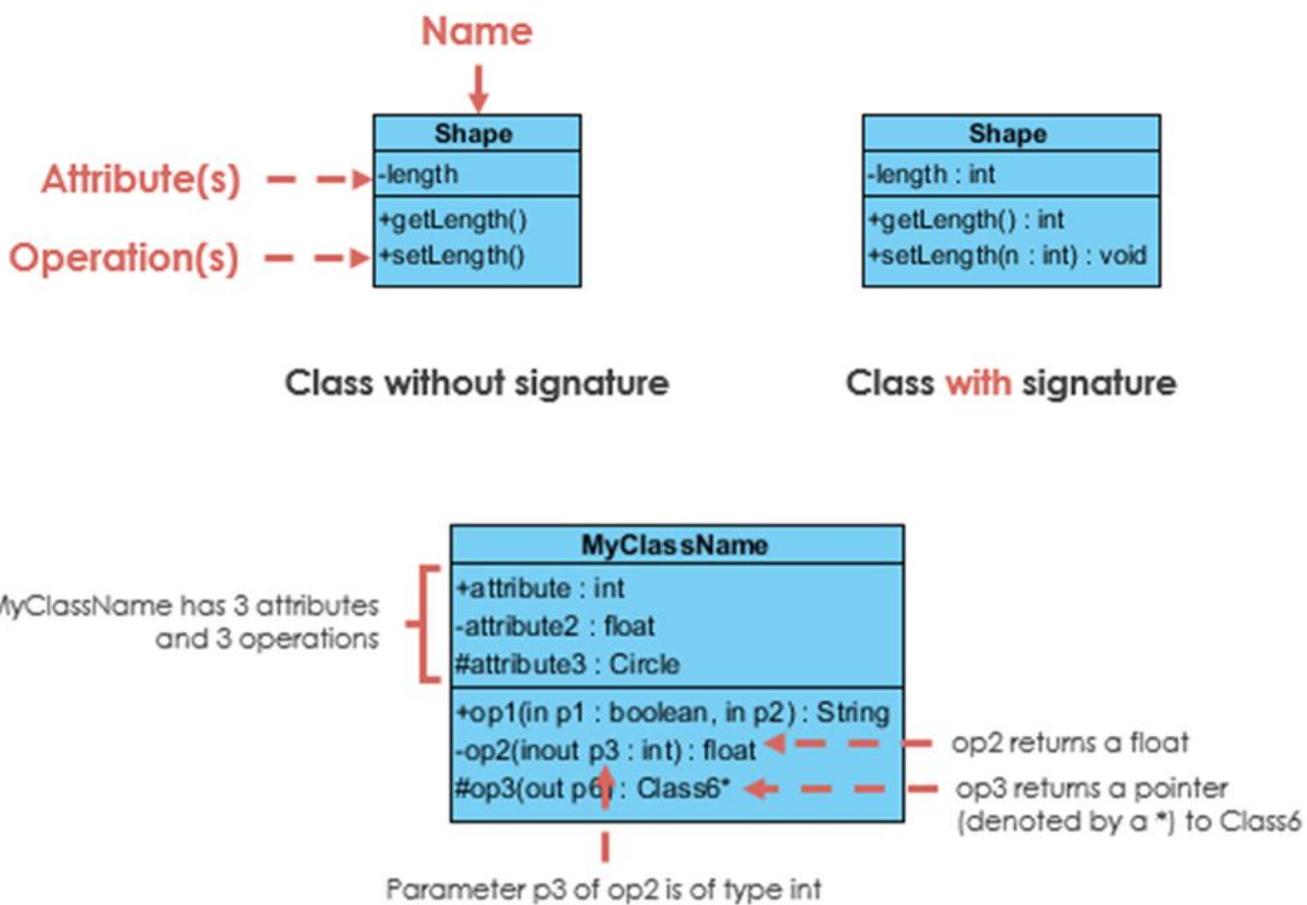
Spécifie les différents composants logiciels ou applications qui seront nécessaires, leurs relations, leur interaction et la manière dont ils s'assemblent pour réaliser les fonctions identifiées dans l'architecture fonctionnelle. ex. diagrammes de composants

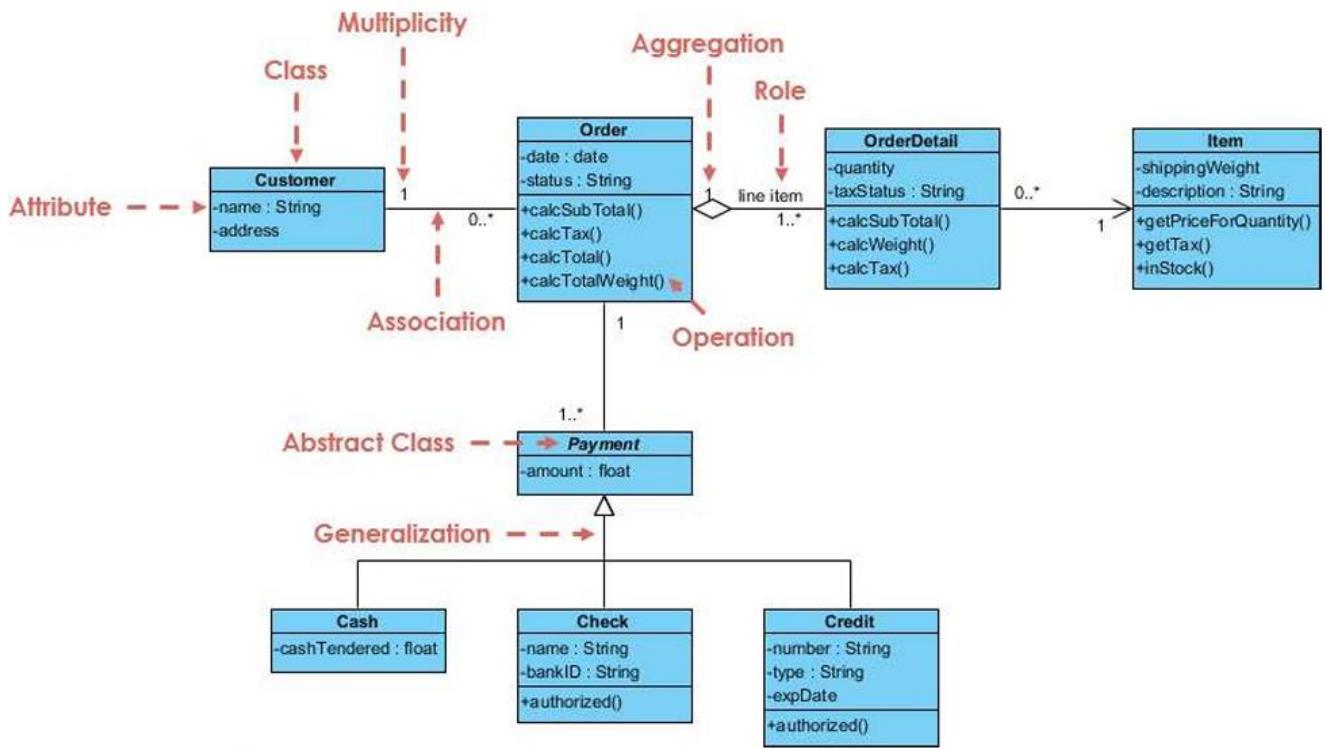
Détaille l'infrastructure nécessaire pour soutenir les applications, y compris le matériel, les réseaux, les serveurs, et d'autres aspects de l'environnement technologique. Ex. Les diagrammes de déploiement

Cycle de vie d'un projet

- ♥ Analyse des besoins et faisabilité
- ♥ Spécifications ou conception générale
- ♥ Conception détaillée
- ♥ Codage (Implémentation ou programmation)
- ♥ Tests unitaires
- ♥ Intégration
- ♥ Qualification (ou recette)
- ♥ Documentation
- ♥ Mise en production
- ♥ Maintenance

Diagramme de Classes



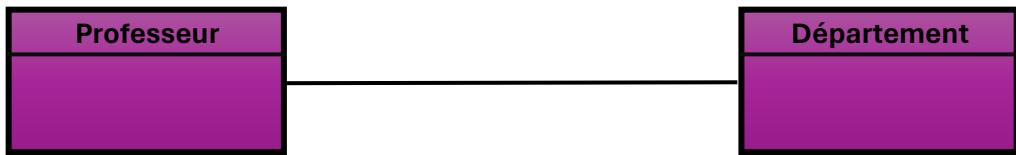


Cardinalités

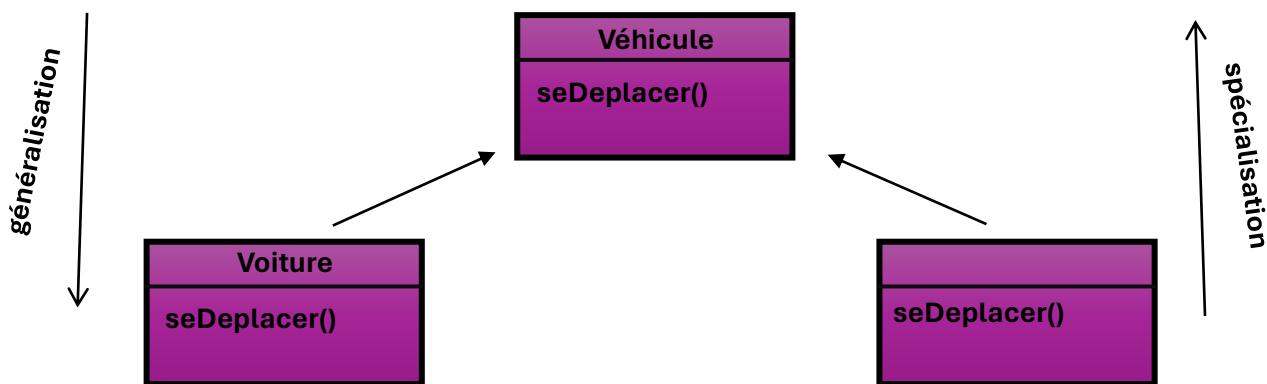
Multiplicité	Définition
1	Un et un seul
0..1	Zéro ou un
n ou *	n (entier naturel)
m..n	De m à n (entiers naturels)
0..*	De zéro à plusieurs
1..*	D'un à plusieurs

Types de liaisons

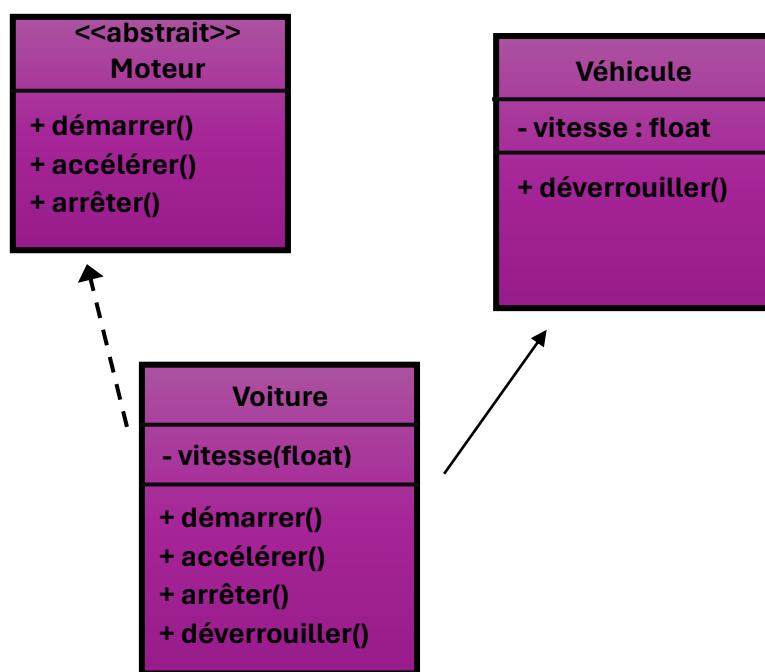
- ♥ **Association:** Une relation entre deux classes qui établit des connexions entre leurs objets. (ils sont indépendants)



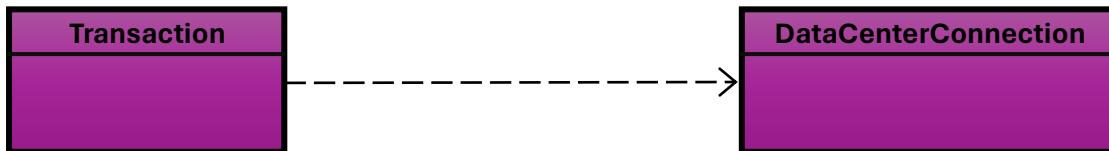
- ♥ **Héritage:** indique qu'une classe (souvent appelée classe enfant ou sous-classe) hérite des propriétés et des méthodes d'une autre classe (classe parent ou superclasse). Cela signifie que la sous-classe est une spécialisation de la superclasse.



- ♥ **Réalisation:** Cela se produit généralement entre une interface et une classe qui implémente les méthodes définies dans l'interface.

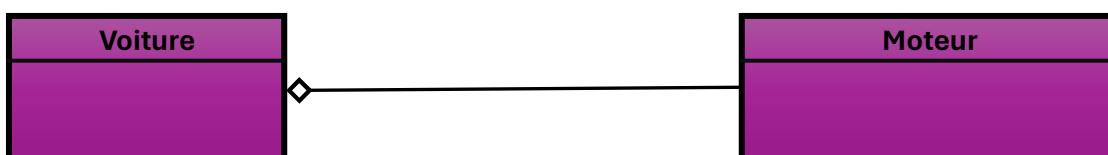


- ♥ **Dépendance:** signifie qu'un changement dans la définition de l'une des classes pourrait affecter l'autre classe. C'est souvent une relation d'utilisation où une classe utilise une autre temporairement ou dans un contexte particulier

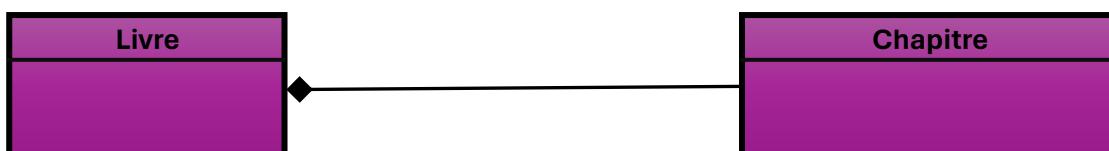


- ♥ **Agrégation:** un type spécial d'association où les objets d'une classe peuvent être assemblés ou configurés ensemble pour créer un objet plus complexe.

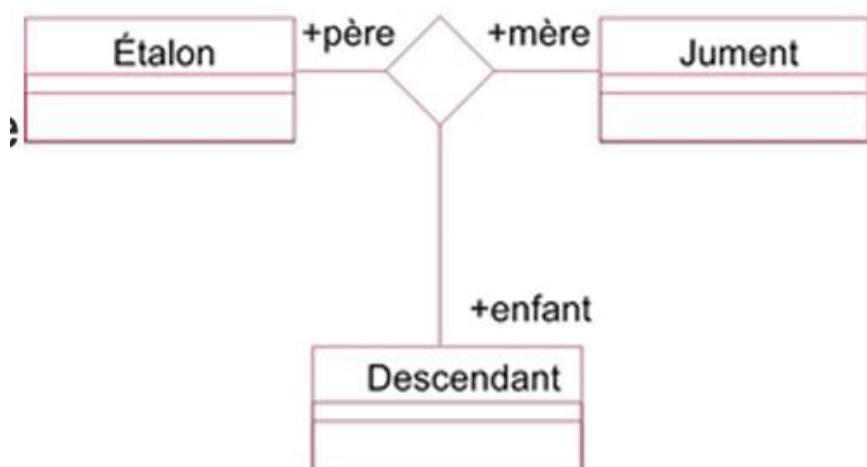
une relation "**tout-partie**" où la partie peut exister indépendamment du tout (moteur peut exister sans voiture)



- ♥ **Composition:** Une ligne solide avec un losange plein représente une composition, qui est aussi une forme spéciale d'association. C'est une relation "tout/partie" forte où la partie ne peut pas exister sans le tout, signifiant que si le tout est détruit, les parties le sont également.



- ♥ **Association ternaire**

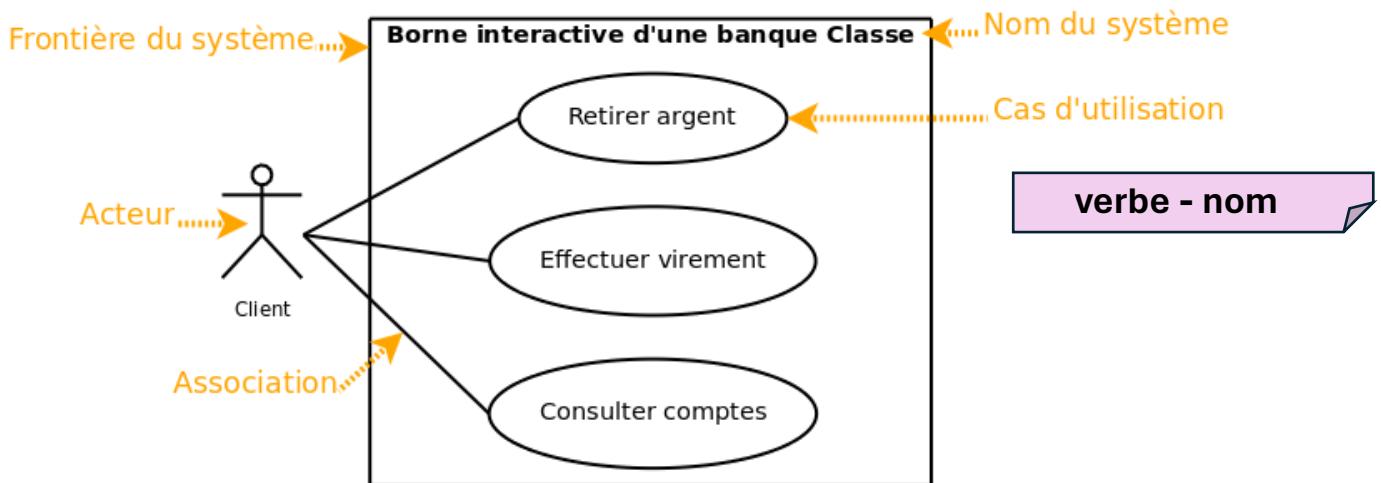


Comparaison Classe *Abstraite* / *Concrète*

Critère	Classe Abstraite	Classe Concrète
Instanciation	Ne peut pas être instanciée directement.	Peut être instanciée pour créer des objets.
Utilisation	Sert de modèle ou de base pour d'autres classes.	Utilisée pour créer des objets spécifiques dans le système.
Méthodes	Peut contenir des méthodes abstraites (sans implémentation) et des méthodes avec implémentation.	Doit fournir une implémentation pour toutes ses méthodes, y compris celles héritées.
Objectif	Définir une interface commune et un comportement partagé pour les classes dérivées.	Implémenter cette interface commune en fournissant des comportements spécifiques.
Héritage	Peut être héritée par d'autres classes abstraites ou concrètes.	Peut hériter d'une classe abstraite mais doit implémenter toutes les méthodes abstraites.
Exemple d'utilisation	Une classe Véhicule avec une méthode abstraite démarrer().	Une classe Voiture qui implémente la méthode démarrer() de Véhicule.

Diagramme de cas d'utilisation

- les fonctionnalités d'un système du point de vue des utilisateurs



- Acteur** - Type stéréotypé qui représente un rôle joué par une personne ou une chose qui interagit avec le système



- Les liaisons** - les lignes reliant les acteurs aux cas d'utilisation indiquent qu'ils sont impliqués ou intéressés par ces fonctionnalités. Peuvent inclure des relations de généralisation entre acteurs ou des relations d'extension et d'inclusion entre cas d'utilisation.
- Cas d'utilisation** - une unité cohérente représentant une fonctionnalité visible de l'extérieur. Il réalise un service de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie. Un cas d'utilisation modélise donc un service rendu par le système, sans imposer le mode de réalisation de ce service.

Include/Extend

- Include** : Un cas d'utilisation "include" est un cas d'utilisation inclus qui est toujours appelé par un autre cas d'utilisation de base. L'utilisation de "include" est appropriée lorsqu'une séquence d'actions est commune à plusieurs cas d'utilisation et doit donc être factorisée pour éviter la redondance. Cela établit une relation de dépendance obligatoire entre les deux cas d'utilisation.

Par exemple, si vous avez un cas d'utilisation "Gérer Commande" et un cas d'utilisation "Valider Paiement", "Valider Paiement" pourrait être inclus dans "Gérer Commande" car chaque fois qu'une commande est gérée, le paiement doit être validé.

- ♥ **Extend** : Un cas d'utilisation "extend" est un cas d'utilisation qui ajoute des comportements supplémentaires ou des conditions optionnelles à un cas d'utilisation de base. Contrairement à "include", l'extension n'est pas toujours exécutée, elle dépend de certaines conditions ou choix réalisés au cours de l'exécution du cas d'utilisation de base. Cela permet d'ajouter des fonctionnalités facultatives sans surcharger le cas d'utilisation de base

Par exemple, considérons un cas d'utilisation de base "Passer Commande". Il pourrait y avoir un cas d'utilisation étendu "Offrir Option Cadeau" qui ajoute l'option d'emballer la commande comme un cadeau. Ce cas d'utilisation "Offrir Option Cadeau" est optionnel et ne s'applique que si le client choisit de l'utiliser.

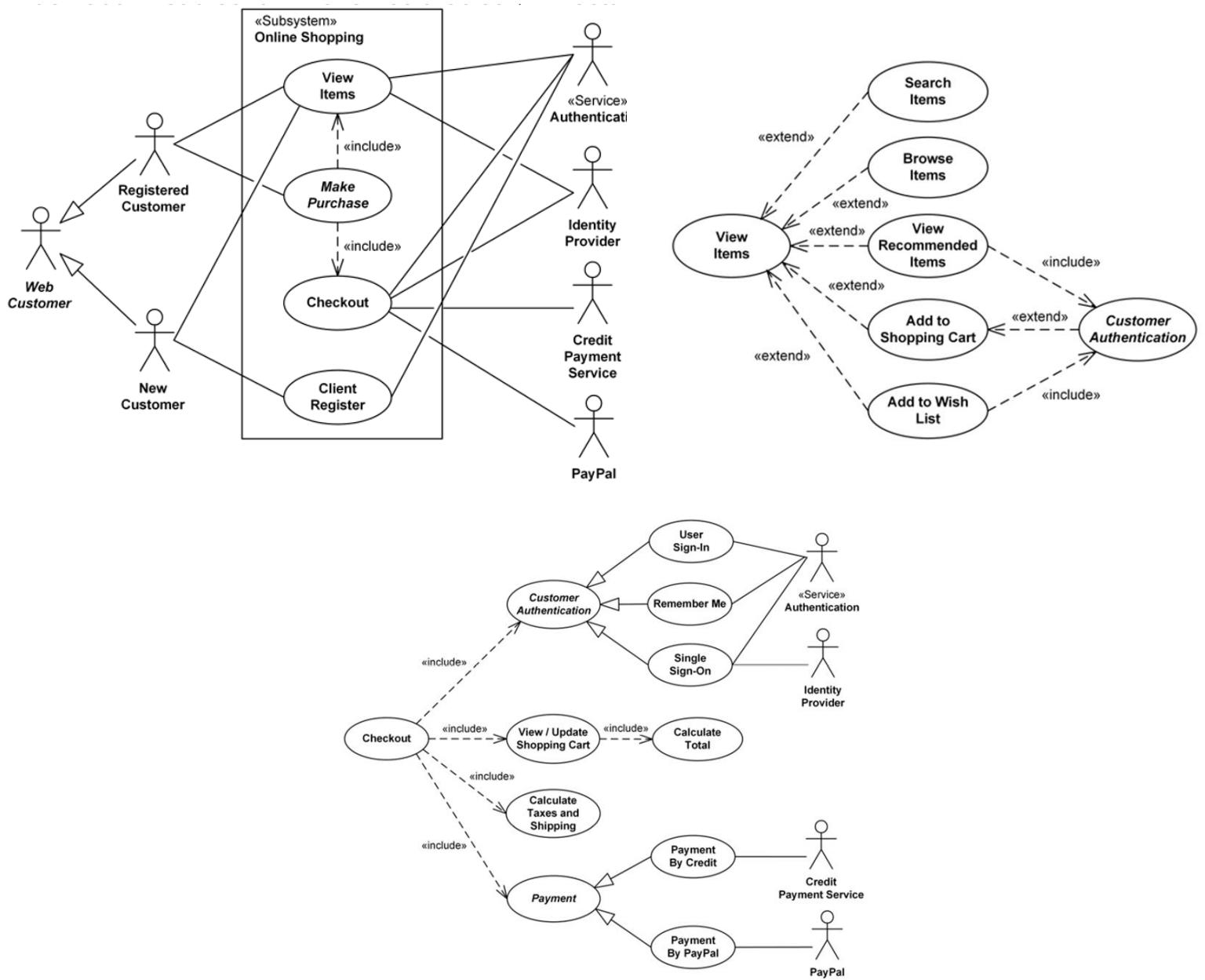
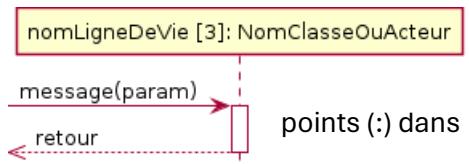


Diagramme de Séquences

Les diagrammes de séquences permettent de décrire **COMMENT** les éléments du système interagissent entre eux et avec les acteurs

- **Éléments** : acteurs, objets, lignes de vie
- ♥ **Une ligne de vie** - représente un participant à une interaction (objet ou acteur).

Une ligne de vie est une instance, donc il y a nécessairement les deux son libellé.



♥ Messages

Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie :

- Ils sont représentés par des flèches
- Ils sont présentés du haut vers le bas le long des lignes de vie, dans un ordre chronologique

Un **message** définit une communication particulière entre des lignes de vie (objets ou acteurs).

Plusieurs **types de messages** existent, dont les plus courants :

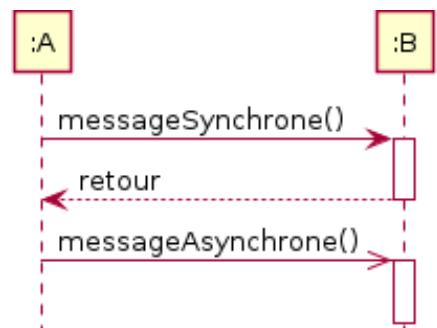
- **L'envoi** d'un signal ;
- L'invocation d'une opération (**appel de méthode**)
- La **création** ou la **destruction** d'un objet.

La réception des messages provoque une période d'activité (rectangle vertical sur la ligne de vie) marquant le traitement du message (spécification d'exécution dans le cas d'un appel de méthode).

♥ Messages synchrones et asynchrones

Un message **synchrone** bloque l'expéditeur jusqu'à la réponse du destinataire. Le flux de contrôle passe de l'émetteur au récepteur.

- Si un objet A envoie un message synchrone à un objet B, A reste bloqué tant que B n'a pas terminé.
- On peut associer aux messages d'appel de méthode un message de retour (en pointillés) marquant la reprise du contrôle par l'objet émetteur du message synchrone.

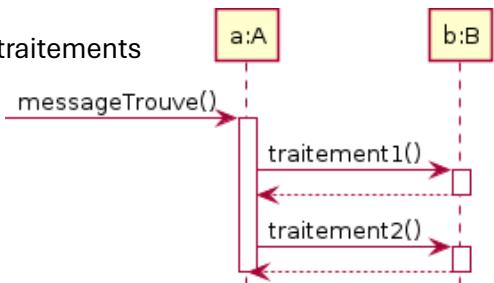


Un message **asynchrone** n'est pas bloquant pour l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré.

♥ Messages trouvés

Les diagrammes de séquences peuvent être employés pour décrire les traitements d'un système résultant de l'envoi d'un message, indépendamment de l'émetteur. Dans ce cas, l'émetteur importe peu et on ne le spécifie pas.

Les messages trouvés peuvent être synchrones ou asynchrones.



♥ Messages perdus

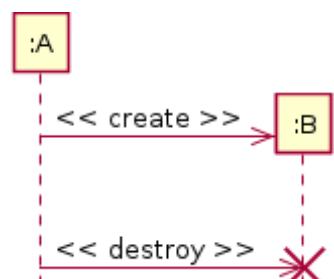
Des messages perdus, on connaît l'émetteur mais pas le récepteur, à l'inverse des messages trouvés.

On utilise souvent des messages de retour perdus pour spécifier le résultat d'un message synchrone trouvé.

♥ Création et destruction d'objets (et de lignes de vie)

Création : message asynchrone stéréotypé << create >> pointant vers le rectangle en tête de la ligne de vie

Destruction : message asynchrone stéréotype << destroy >> précédant une croix sur la ligne de vie



♥ Fragment combiné

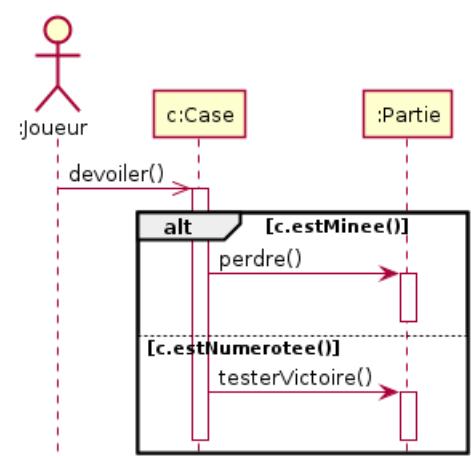
Un fragment combiné permet de décomposer une interaction complexe en fragments suffisamment simples pour être compris.

Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté un rectangle dont le coin supérieur gauche contient un pentagone.

♥ Fragment alt : opérateur conditionnel

Les différentes alternatives sont spécifiées dans des zones délimitées par des pointillés.

- Les conditions sont spécifiées entre crochets dans chaque zone.
- On peut utiliser une clause [else]



Fragment loop : opérateur d'itération

Le fragment `loop` permet de répéter ce qui se trouve en son sein.

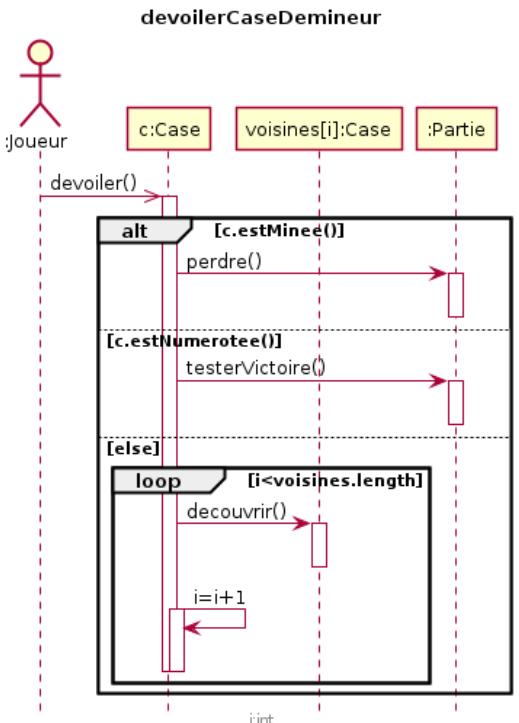
On peut spécifier entre crochets à quelle condition continuer.

Remarques

- Les fragments peuvent s'imbriquer les uns dans les autres

Tous les éléments d'un diagramme doivent être définis. Typiquement, les attributs doivent correspondre :

- soit à des attributs définis dans un diagramme de classes au niveau de la ligne de vie contrôlant le flux d'exécution
- soit à des attributs définis localement au diagramme de séquence (ici, \perp)



Opérateurs de flux de contrôle

opt (*facultatif**): Contient une séquence qui peut ou non se produire. Dans la protection, vous pouvez spécifier la condition sous laquelle elle se produit.

alt: Contient une liste des fragments dans lesquels se trouvent d'autres séquences de messages. Une seule séquence peut se produire à la fois.

loop: Le fragment est répété un certain nombre de fois. Dans la protection, on indique la condition sous laquelle il doit être répété.

break: Si ce fragment est exécuté, le reste de la séquence est abandonné. Vous pouvez utiliser la protection pour indiquer la condition dans laquelle la rupture se produira.

par (*parallel*): Les événements des fragments peuvent être entrelacés.

critical: Utilisé dans un fragment `par` ou `seq`. Indique que les messages de fragment ne doivent pas être entrelacés avec d'autres messages.

seq: Il existe au moins deux fragments d'opérande. Les messages impliquant la même ligne de vie doivent se produire dans l'ordre des fragments. Lorsqu'ils n'impliquent pas les mêmes lignes de vie, les messages des différents fragments peuvent être entrelacés en parallèle.

strict: Il existe au moins deux fragments d'opérande. Les fragments doivent se produire dans l'ordre donné.

Opérateurs d'interprétation de la séquence

consider: Spécifie une liste des messages que ce fragment décrit. D'autres messages peuvent se produire dans le système en cours d'exécution, mais ils ne sont pas significatifs quant aux objectifs de cette description.

ignore: Liste des messages que ce fragment ne décrit pas. Ils peuvent se produire dans le système en cours d'exécution, mais ils ne sont pas significatifs quant aux objectifs de cette description.

assert : Le fragment d'opérande spécifie les seules séquences valides. Généralement utilisé dans un fragment Consider ou Ignore.

neg : La séquence affichée dans ce fragment ne doit pas se produire. Généralement utilisé dans un fragment Consider ou Ignore.

Algorithmie

Un algorithme est une méthode permettant de résoudre un problème de manière systématique

Les arbres

- ♥ Représentation hiérarchique des données
- **Racine** - Nœud initial, se trouve tout au-dessus de l'arbre
- **Nœud** - élément d'un arbre qui comprend une valeur ainsi qu'une référence vers d'autres nœuds
- **Feuilles** - élément en bout d'arbre, il s'agit d'un nœud qui n'a pas de référence vers d'autres nœuds

Profondeur de l'arbre (niveau) – distance entre le nœud et de la racine

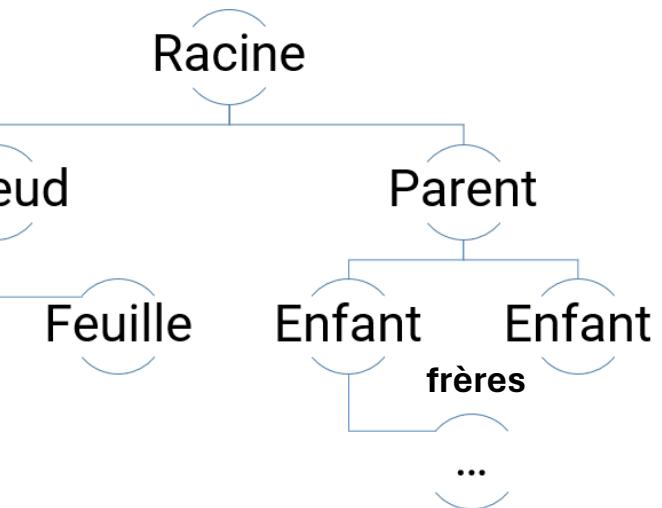
Hauteur – profondeur maximale d'un nœud

A quoi ça sert ?

- ♥ Présentation des fichiers
- ♥ Les cas
- ♥ Décomposer pour rechercher un élément particulier

Les arbres binaires

- ♥ Un parent ne peut avoir en maximum 2 enfants

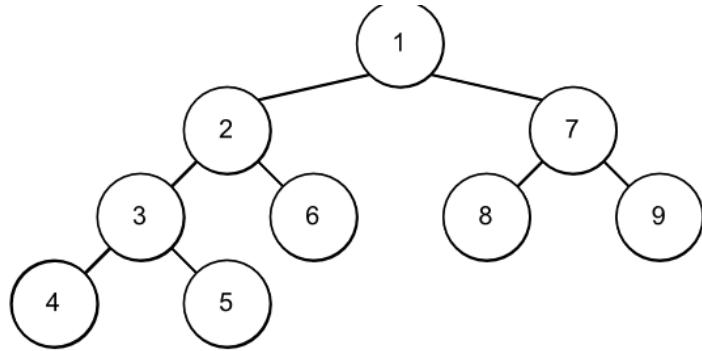


```
class BTree:  
    def __init__(self, key):  
        self.left = None  
        self.right = None  
        self.val = key
```

- ♥ Méthodes de parcours :

- En profondeur :
 - **Pr  fixe** – racine, gauche, droite, parents en premier jusqu'au bas, ensuite r  cursif.
 - **Utilit  ** - cr  er une copie exacte de l'arbre, facilement acc  der aux donn  es de dessus de l'arbre.

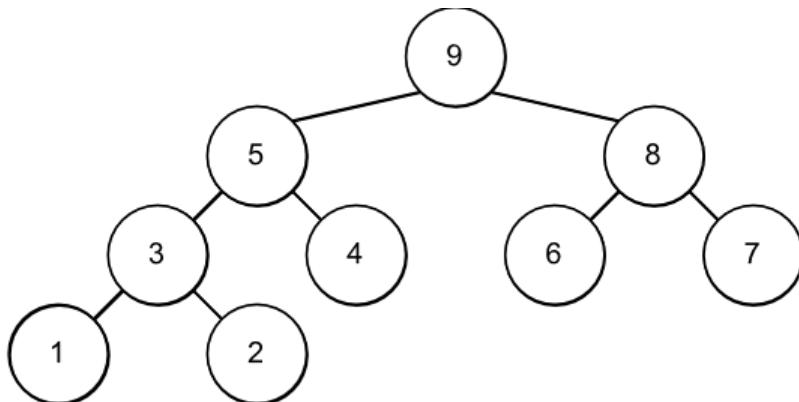
Pr   = avant, donc la racine en premier dans la liste



- **Postfixe** – gauche, droite, racine.
- **Utilité** - pour supprimer ou libérer l'arbre de manière sûre (les enfants avant le parent), et pour certains types de calculs qui nécessitent que les résultats des sous-arbres soient disponibles avant de traiter la racine.

Post = après, donc la racine en dernier dans la liste

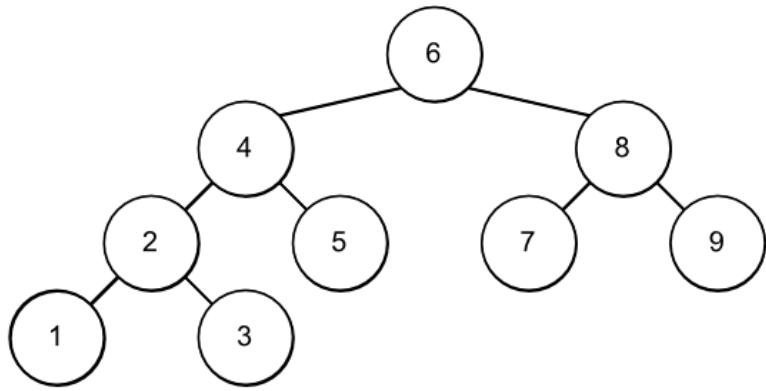
Tous les enfants d'abord, ensuite les parents.



- **infixe** – gauche , racine, droite

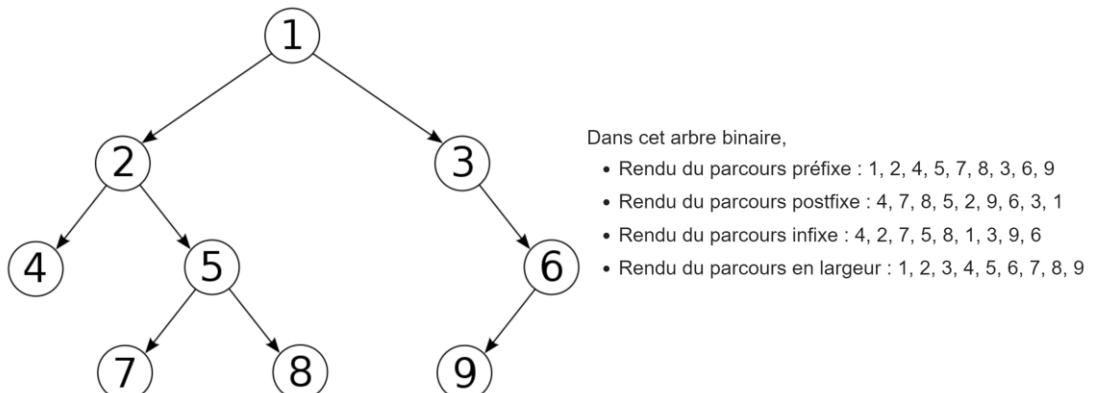
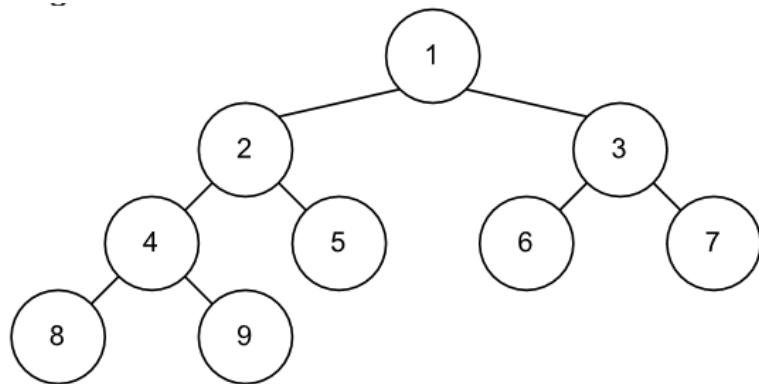
Parcours :

- Est-ce que mon nœud a un fils à gauche ? OUI = descendre (sans ajouter au parcours)
NON = ajouter à la liste, ensuite remonter au parent, ajouter le parent à la liste.
- Est-ce que le dernier enfant parcouru (ou même si il était inexistant, exemple avec un nœud avec un seul fils à droite, à ce moment-là, on compte le parent ensuite on va sur le fils, car il check en premier à gauche, ensuite remonte et va à droite), avait un frère ? (càd un fils à droite, le frère du fils de gauche duquel on vient de remonter)
OUI = on descend sur ce fils et recommence l'algorithme. NON = On remonte au parent et recommencer cette étape.



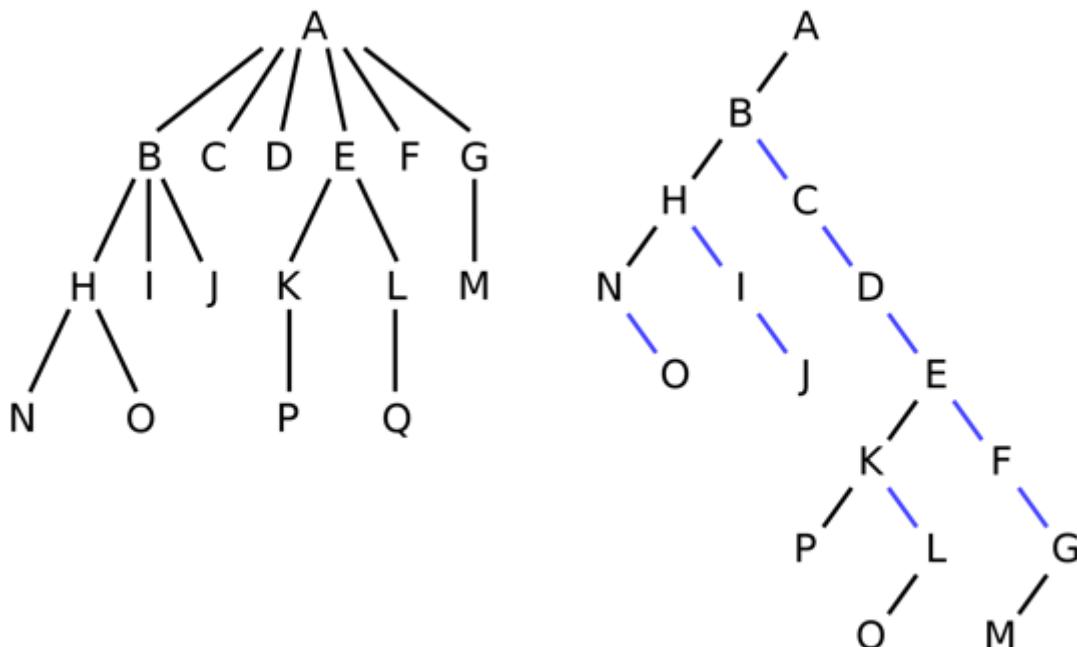
- **En largeur – par niveau**

- Utilité - pour trouver le chemin le plus court sans pondération dans les arbres ou pour des applications nécessitant une exploration uniforme de tous les nœuds à une profondeur donnée avant de passer à la suivante.



♥ Transformer un arbre n-aire en binaire – les frères deviennent les enfants de l'un à l'autre

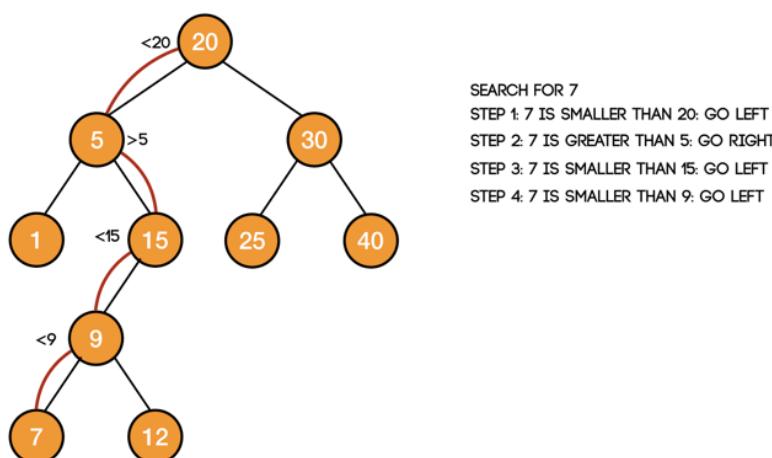
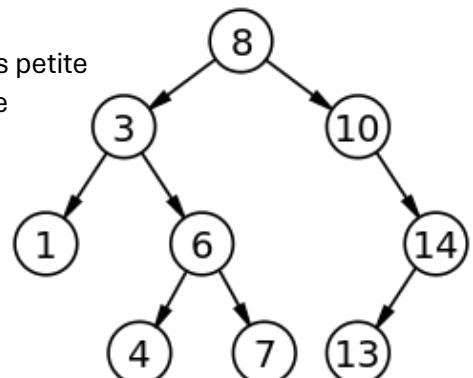
- Tout ce qui est à droite – enfants, à gauche – parents



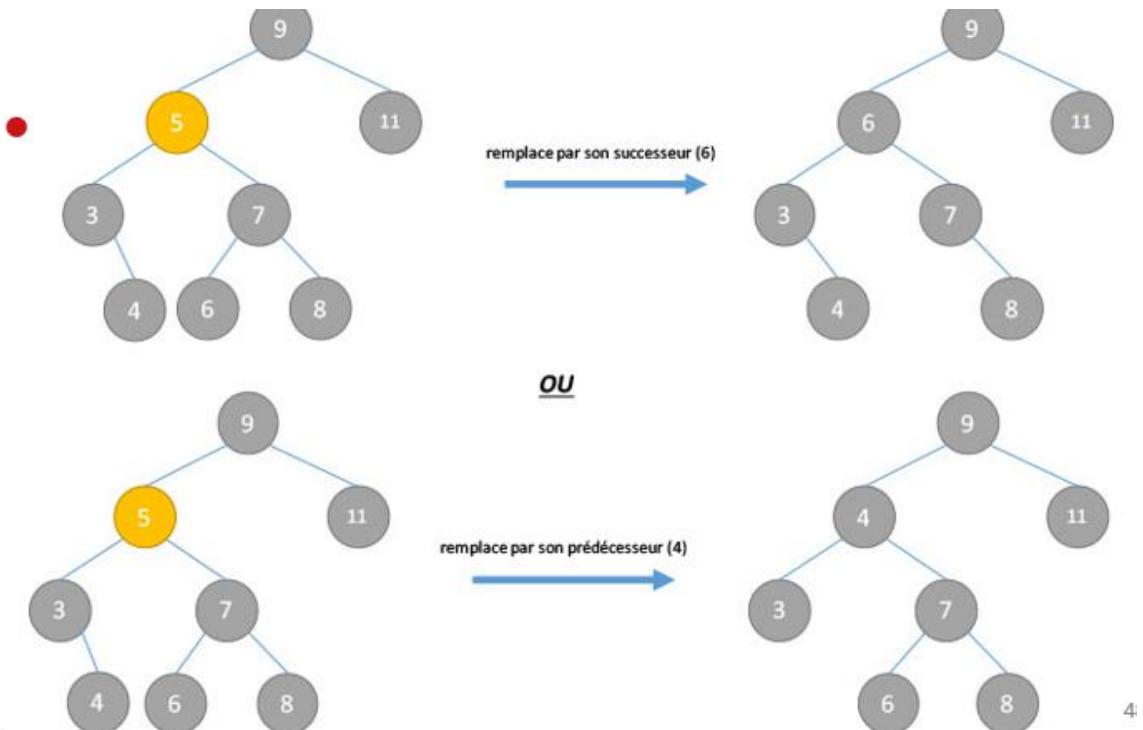
Arbres binaires de recherche

- ♥ Chaque nœud possède une clef (valeur), où n de gauche a une clef plus petite ou égale au nœud considéré, et inversement pour le sous-arbre à droite
- ♥ **Successeur** – le plus petits des plus grands, donc le plus à gauche de sous-arbre de droite
- ♥ **Prédécesseur** – le plus grands des plus petits, donc parcourir le reste de l'arbre et remplacer
- ♥ Actions principales :

- **Rechercher** – smaller \Rightarrow go left \Rightarrow grater ? go right \Rightarrow égale ? c'est bon



- **ajouter** – ce sera toujours une feuille , smaller ? go left \Rightarrow greater ? on ajoute à droite
- **supprimer** :
 - supprimer une feuille
 - enfant ? remplacer par celui de dessous, soit successeur, soit prédécesseur



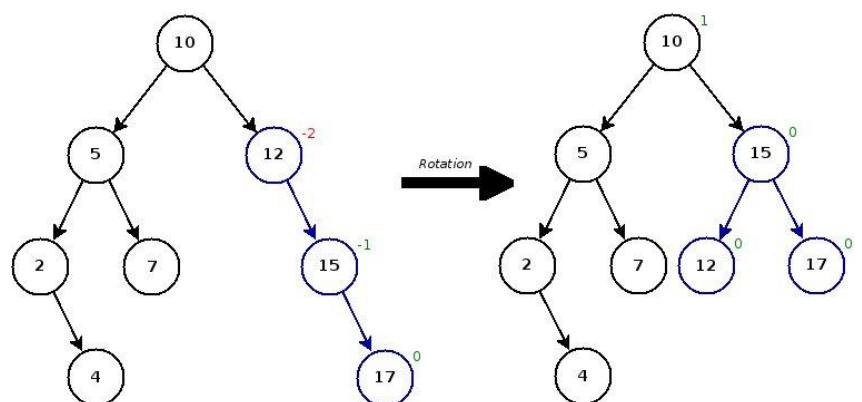
♥ Complexité temporelle

- Pire : $O(n)$
- Autres cas – ça dépend de la structure de l'arbre
 - Si équilibré : $O(\log(n))$
 - Sinon : $O(n)$

♥ Comment équilibrer un arbre binaire de recherche ?

Un arbre binaire est équilibré si tous les chemins de la racine aux feuilles ont la même longueur.

La rotation d'un arbre binaire de recherche permet de changer la structure d'un ABR sans invalider l'ordre des éléments. Une telle rotation consiste en fait à faire remonter un nœud dans l'arbre et à en faire redescendre un autre.



Pseudo code pour calculer la hauteur (H)

```
H(node) :
    H = 0
    R = node.R
    L = node.L
    If R != null and L != null :
        H = MAX( H(L), H(R)) +1
    If R != null and L == null :
        H = H(R) +1
    If R==null and L != null :
        H = H(L) +1
```

Pseudo code pour calculer la balance (B)

```
B(node) :
    R = node.R
    L = node.L
    If R != null and L != null :
        B = H(L) - H(R)
    If R != null and L == null :
        B = -1 - H(R)
    If R==null and L != null :
        B = H(L) + 1
```

si $B > 0$: left heavy

si $B < 0$: right heavy

Example sur l'image:

$B(15) - ?$

1) $R = 17, L = \text{null};$

$$B(15) = -1 - H(17)$$

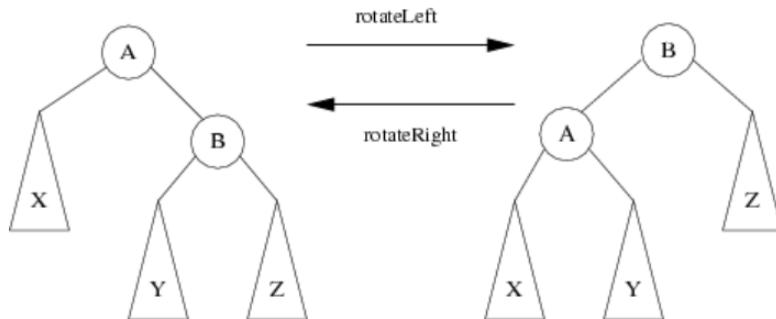
2) $H(17) = ?$

$R, L = \text{null}$

$$H(17) = 0$$

3) $B(15) = -1 - 0 = -1 \rightarrow \text{right heavy}$

Rotating



If the tree on the left is a binary search tree, the keys in subtree X are less than A, the keys in subtree Y are greater than A, and the keys in subtree Z are greater than B.

```
/** Rotates node a to the left, making its right child into its parent.
 * @param a the former parent
 * @return the new parent (formerly a's right child)
 */
Node rotateLeft(Node a) {
    Node b = a.getRight();
    a.setRight(b.getLeft());
    b.setLeft(a);
    return b;
}
```

Rotate Left :

En gros: Déplacer l'enfant de droite d'un Node en tant que parent. Utiliser si la balance de ce Node est < -1 , donc RightHeavy

Rotate Right :

Déplacer l'enfant de gauche d'un Node en tant que parent. Utiliser si la balance de ce Node est > 1 , donc LeftHeavy.

Tri par tas

- ♥ Arbre doit être parfait
- ♥ Respecte la propriété du tas : la clé d'un nœud parent a une plus haute *priorité* que les clés de ses enfants

1. Associer une liste à l'arbre
2. Construire un tas : Transformez votre liste en un tas où chaque parent est plus grand que ses enfants. (**parcours en largeur**)
3. Trier : Répétez les étapes suivantes jusqu'à ce que le tas soit vide :
4. Retirez le plus grand élément (la racine du tas) et déplacez-le à la fin de la liste.
5. Réorganisez le reste du tas pour maintenir sa structure (le nouvel élément en haut doit "descendre" à sa bonne place).
6. Le résultat est une liste triée, du plus petit au plus grand élément.

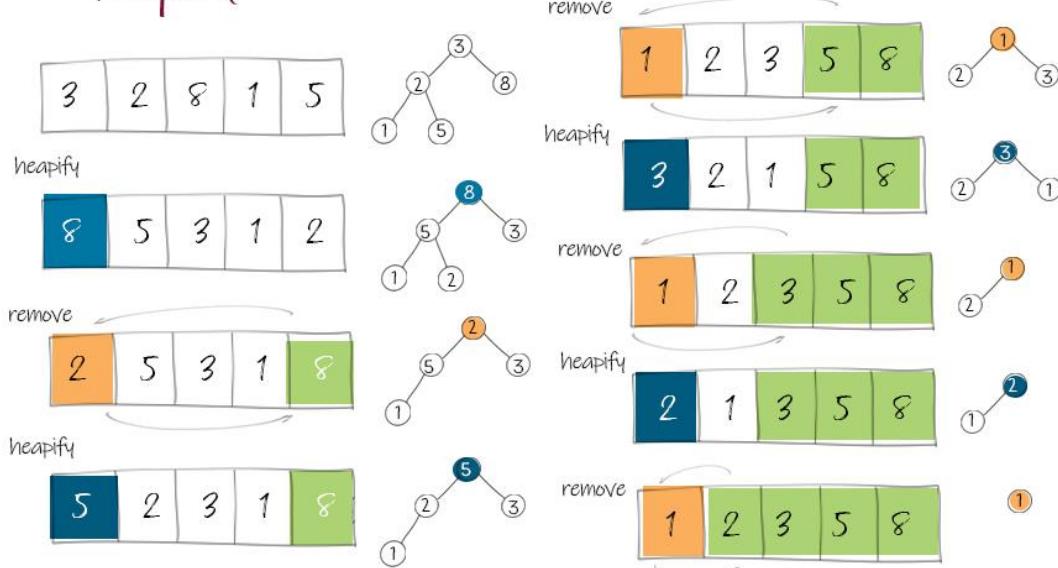
Complexité temporelle

Meilleur	$O(n \log n)$
Pire	$O(n \log n)$
Moyenne	$O(n \log n)$
Complexité spatiale	$O(1)$
Stabilité	Non

https://www.youtube.com/watch?v=mgUiY8CVDhU&ab_channel=AbhilashBiswas

Utilités : os, gui, moteurs de recherche

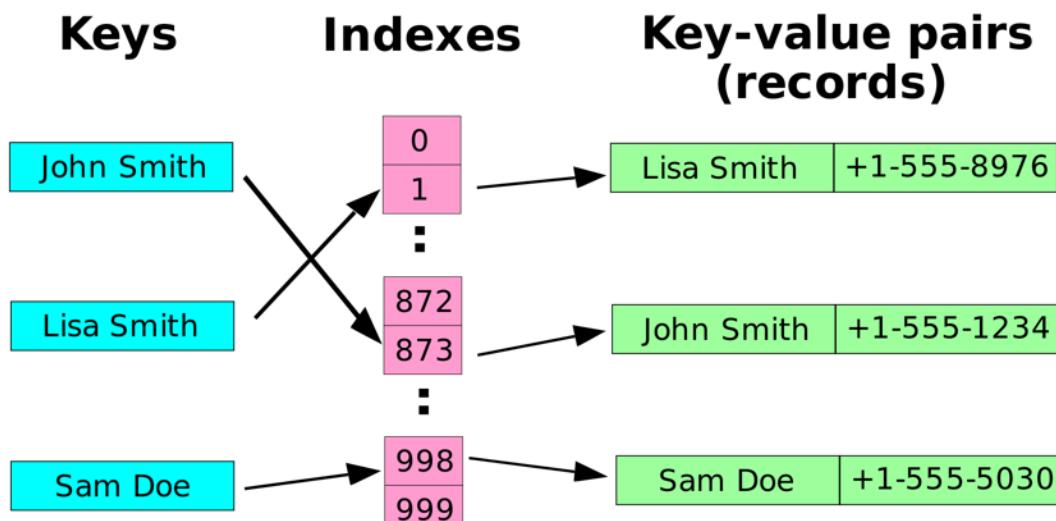
Heapsort



Les tables de hachage

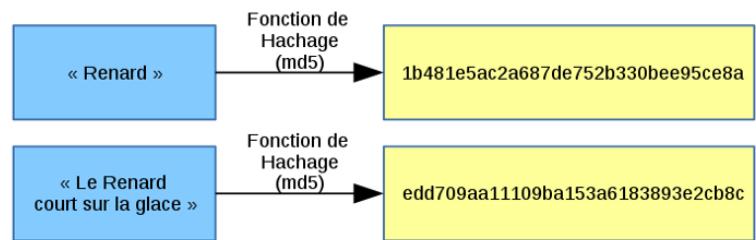
- ♥ Structure des données reposant sur des **tableaux** (donc statique)
- ♥ Algorithme performant
- ♥ Position de l'élément en fonction de l'élément lui-même
- ♥ **Clef** – partie d'un élément qui permet de désigner le contenu de cet élément sans ambiguïté

Ex. Etudiant – Matricule



- ♥ **Fonction de hachage** : Lorsque vous voulez stocker une donnée (comme une valeur associée à une clé), vous utilisez une fonction de hachage qui transforme la clé en un nombre appelé "hash".
- ♥ **Stockage** : Ce hash est utilisé pour déterminer à quel endroit (index) de la table de hachage la valeur sera stockée. Pensez à l'index comme à un casier spécifique où vous rangez vos affaires.
- ♥ **Collisions** : Parfois, deux clés différentes donnent le même hash, ce qu'on appelle une "collision". Pour gérer les collisions, la table de hachage peut stocker une liste à chaque index ou utiliser une autre méthode pour trouver un autre emplacement pour la deuxième valeur.
- ♥ **Recherche** : Quand vous voulez retrouver une valeur, vous utilisez la même fonction de hachage sur la clé, qui vous donne l'index pour trouver rapidement la valeur dans la table.
- ♥ **Avantage principal** : L'avantage des tables de hachage est qu'elles peuvent, dans le meilleur des cas, offrir des temps d'accès, d'insertion et de suppression en temps constant, c'est-à-dire O(1).
- ♥ **Redimensionnement** : Si la table de hachage se remplit trop, elle peut être redimensionnée. Un nouveau tableau plus grand est créé et les valeurs sont re-hachées et réparties dans ce nouveau tableau.
- ♥ **Fonction de hachage** – utilisation d'une fonction de hachage qui va calculer une empreinte unique à partir de données fournies

- Longueur de la signature identique quel que soit la taille des données
- Unidirectionnel
- Impossible de prédire la signature
- 2 données différentes = 2 signatures différentes



Pour éviter des collisions :

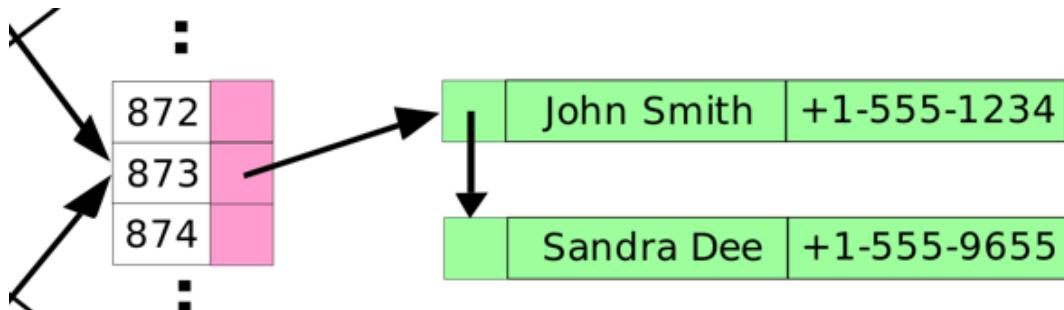
- Utiliser une fonction de hachage à faible taux de collisions
- **Salage** - ajouter un élément à la donnée pour en modifier la signature
 - Augmente la sécurité face à différentes attaques
 - Eviter des doublons

Exemple : Python > 6e3604888c4b4ec08e2837913d012fe2834ffa83

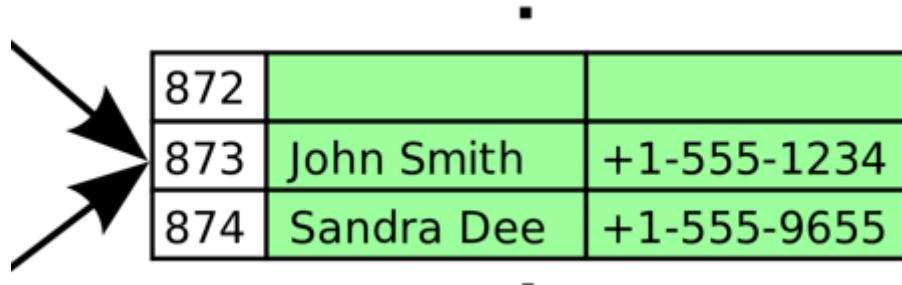
Python_salé > 7d32954c9a36e2cdcce761bf462932b1cb12539

♥ Solutions :

- **par chainage** - stocker dans la même case de mémoire 2 données



- **par adressage ouvert** – stocker juste après car la possibilité d'avoir ce hachage est très faible



♥ Ex de fonction :

- MD5 – bas niveau – signature de 128 bits
- SHA-1 – 160 bits
- SHA-2 – 256 et 512 bits

♥ Complexité temporelle

Algorithm	Average	Worst case
List	$O(n)$	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Complexité temporelle extrêmement intéressante pour chercher, ajouter ou supprimer un élément

♥ Utilité

Base de données, caches, etc

♥ Le SHA-256

Est une fonction de hachage cryptographique faisant partie de la famille SHA-2 (Secure Hash Algorithm 2), utilisée pour créer un résumé quasi unique d'une taille fixe (256 bits, d'où son nom) à partir de données de n'importe quelle taille.

1. **Entrée** : Vous avez des données d'entrée, de taille quelconque.
2. **Hachage** : La fonction SHA-256 traite ces données par blocs de taille fixe et effectue une série de calculs complexes et bien définis pour chaque bloc.
3. **Digest** : À la fin du processus, quelle que soit la taille des données d'entrée, le résultat est un "digest" ou "hash" unique de 256 bits.
4. **Unicité** : Il est pratiquement impossible de trouver deux entrées différentes donnant le même hash (ce qu'on appellerait une collision), ce qui fait de SHA-256 une fonction de hachage sûre pour la cryptographie.

SHA-256 est largement utilisé dans la sécurité informatique et la blockchain pour vérifier l'intégrité des données et créer des empreintes digitales uniques pour les fichiers.

La fonction de hachage SHA-256 utilise une combinaison d'opérations mathématiques et logiques pour transformer les données d'entrée en une empreinte digitale de 256 bits. Voici quelques-unes des opérations clés utilisées :

1. **Addition modulo 2^{32}** : Cela signifie ajouter deux nombres ensemble et ensuite prendre le reste de la division de cette somme par 2^{32} . Cela garantit que les résultats restent dans les limites de 32 bits.
2. **Opérations bit à bit** : Des opérations telles que ET, OU, NON, OU exclusif (XOR), décalages de bits à gauche (shifts) et rotations de bits à gauche sont utilisées pour mélanger et transformer les bits des blocs de données.
3. **Fonctions logiques** : Des fonctions spécifiques comme Ch (choix), Maj (majorité) et Sigma, qui mélange les bits d'une manière non linéaire complexe.
4. **Compression** : Chaque nouveau bloc de données est mélangé avec le résultat précédent, ce qui compresse le message en un résumé final de 256 bits

Pathfinding

- ♥ Utilité...

Trouver des chemins entre deux points A et B. Selon différents critères : Distance/coût/vitesse

- ♥ Différentes approches...

- 1) Intuitif.

Prendre les coordonnées de fin, créer une liste FIFO et ajouter l'élément.

Ensuite, parcourir et pour chaque case vérifier les 4 adjacentes, si c'est un obstacle on retire de la liste, sinon on garde.

Utilisation d'un compteur pour la longueur du trajet.

Enfin on mesure les distances et sélectionne la plus courte.

- 2) Algorithmes de graphes

- A*

- Dijkstra ...

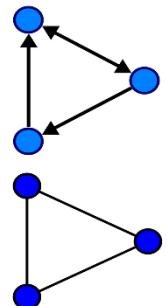
La théorie des graphes

- ♥ Définition...

Graphe est un couple d'ensembles tel que $G = (V, E)$

V = ensemble des sommets

E = ensemble des arêtes (qui sont les liaisons entre deux sommets)



- ♥ Différents types de graphes...

Non-orienté / orienté (dont les arêtes sont dirigées ou non dans une direction donnée)

Non-connexe / connexe (toutes les arêtes sont liées ou non)

- ♥ Structure de donnée abstraite

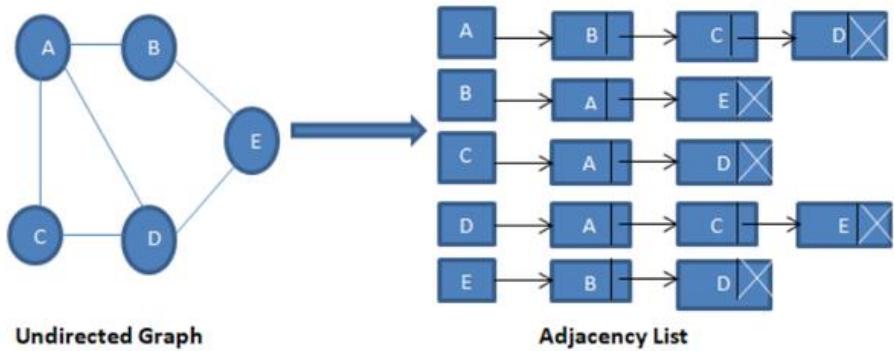
- ♥ Opérations de base

- 1) Adjacents(G, x, y)
- 2) Voisins(G, x)
- 3) Ajouter_Sommet(G, x)
- 4) Supprimer_Sommet(G, x)
- 5) Ajouter_Arete(G, x, y)
- 6) Supprimer_Arete(G, x, y)
- 7) Retourner_Valeur(G, x)
- 8) Fixer_Valeur(G, x, v)

- ♥ Représentation des graphes

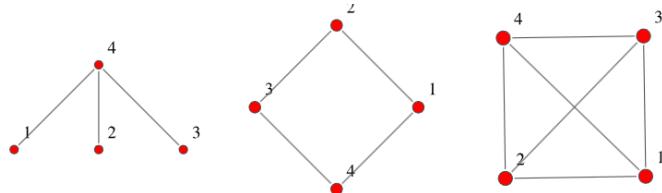
- 1) Liste d'adjacence...

Chaque sommet est un objet qui comprend une liste des sommets adjacents



2) Matrice d'adjacence

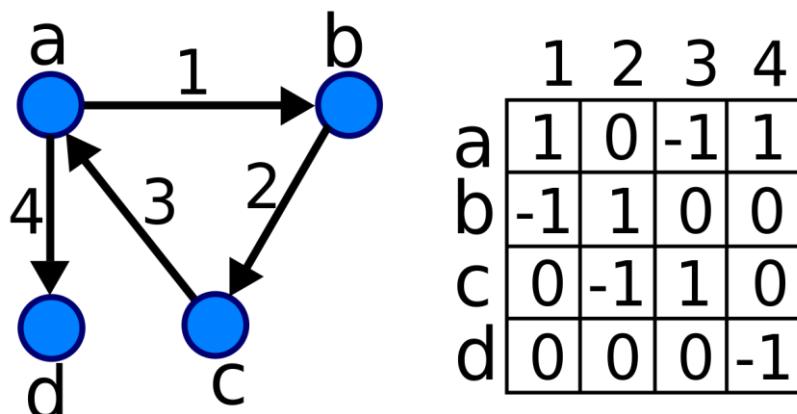
Matrice carrée où les lignes représentent les sommets de départ et les colonnes les sommets d'arrivée



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

3) Matrice d'incidence

Matrice où les lignes représentent les sommets et les colonnes les arêtes



- 1 si le sommet correspondant est l'une des extrémités de l'arête (dans le cas d'un graphe non orienté)
- -1 et 1 pour indiquer le sommet de départ et d'arrivée respectivement (dans le cas d'un graphe orienté)
- 0 si le sommet n'est pas connecté par l'arête.

4) Comparaison...

Comparaison	Liste d'adjacence	Matrice d'adjacence	Matrice d'incidence
Remarques	Lent dans la suppression parce qu'il faut trouver les sommets ou arêtes	Lent dans l'ajonction ou suppression de sommets parce que la matrice doit être reformatée	Lent dans l'ajonction ou suppression de sommets ou d'arêtes parce que la matrice doit être reformatée

♥ Parcours des graphes

1) Parcours en largeur

Utilisation d'une file pour les nœuds voisins

2) Parcours en profondeur

Marquage des sommets visités
Similaire aux labyrinthes

♥ Algorithmes Importants...

1) Shortest path (chemin le plus court entre deux points)

2) Spanning tree (arbre couvrant avec le poids minimal)

3) Min-cost-flow (Manière la plus économique d'utiliser un réseau de transport)

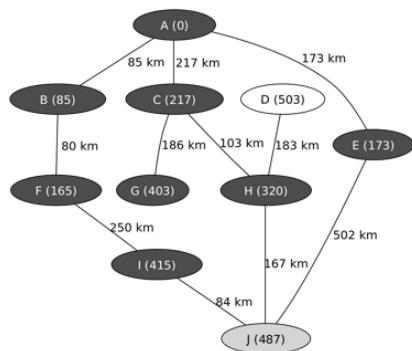
Dijkstra

♥ Graphe pondéré (non-orienté)

♥ Trouver le chemin entre deux sommets avec le poids minimal

♥ Construction de sous-graphes

L'algo « découvre » le graphe, et calcul la distance depuis le point de départ, s'il rencontre une possibilité plus grande que le minimal connu, il change de direction et revient en arrière pour vérifier s'il n'existe pas un chemin plus court. Car la somme des arrêtes les plus courtes sera forcément le chemin le plus court.

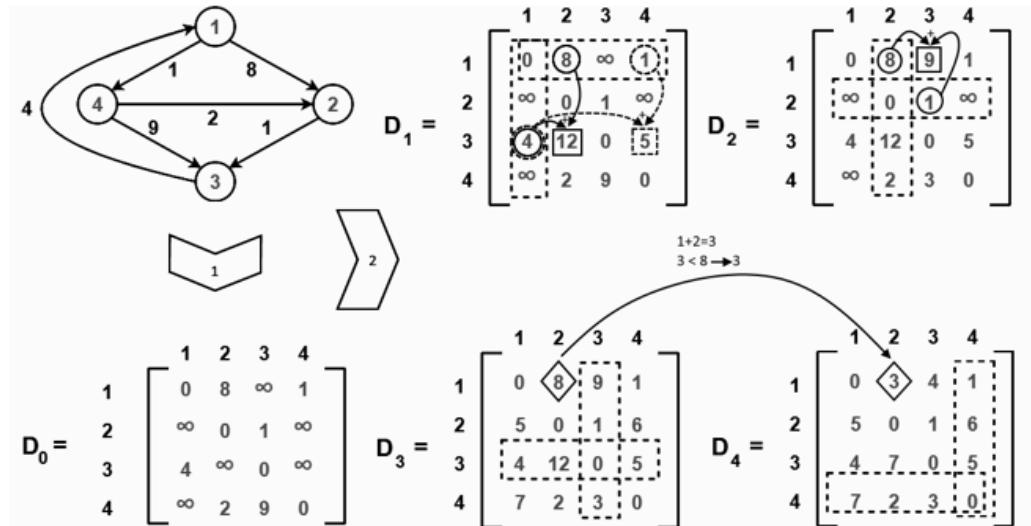


	à A	à B	à C	à D	à E	à F	à G	à H	à I	à J	
étape initiale	0	∞									
A(0)	0	85	217	∞	173	∞	∞	∞	∞	∞	∞
B(85 _A)	-	-	217	∞	173	165	∞	∞	∞	∞	∞
F(165 _B)	-	-	217	∞	173	-	∞	∞	415	∞	
E(173 _A)	-	-	217	∞	-	-	∞	∞	415	675	
C(217 _A)	-	-	-	∞	-	-	403	320	415	675	
H(320 _C)	-	-	-	503	-	-	403	-	415	675 487	
G(403 _C)	-	-	-	503	-	-	-	-	415	487	
I(415 _F)	-	-	-	503	-	-	-	-	-	487	
J(487 _H)	-	-	-	503	-	-	-	-	-	-	
D(503 _H)	-	-	-	-	-	-	-	-	-	-	

Floyd-Warshall

- Distance la plus courte entre toutes les paires de sommets.
Nécessite une représentation en matrice d'adjacence.

♥



24

Autres Algorithmes de Graphes

♥ Bellman-Ford

Autorise les poids négatifs

♥ A*

♥ Algorithme de Kruskal (Spanning tree)

♥ Algorithme de Prim

Similaire à Dijkstra

♥ Algorithme de Boruvka

Utilisation des arêtes et pas des sommets comme base

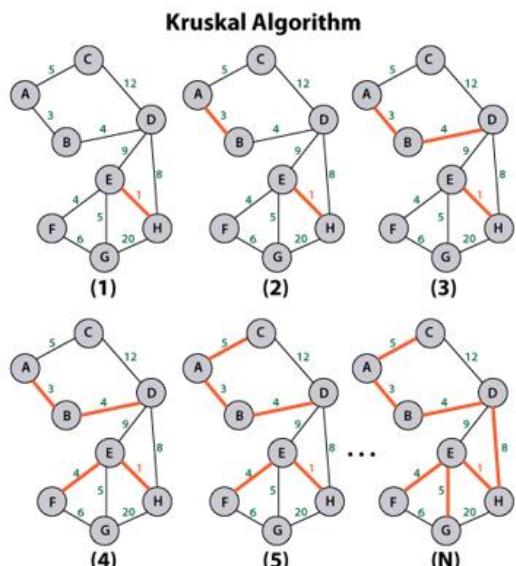
♥ Algorithmes de Min-cost-flow problem...

-Busacker et Gowen

-Ford-Fulkerson (variante du premier)

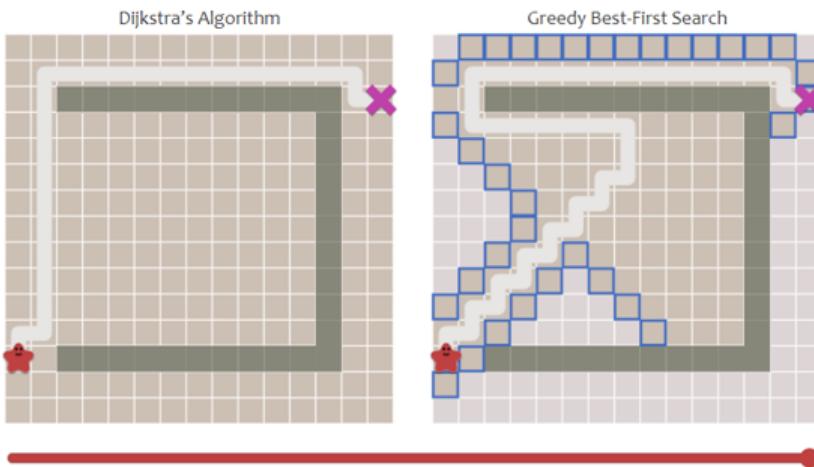
- Edmonds-Karp (flot maximum)

- Goldberg-Tarjan (poussage/réétiquetage)



Heuristique

- ♥ méthode de calcul qui fournit rapidement une solution réalisable, mais pas forcément optimale.
- ♥ Utilisation dans : Graphes/IA/Programmation de jeux



- ♥ **Cirrières d'évaluation :** Qualité du résultat (comparaison avec la solution optimale) / coût (complexité) de l'heuristique / Etendue du domaine d'application
- ♥ **Algorithmes probabilistes...**

1) Algorithme de Monte-Carlo

Temps : déterministe

Résultat : probabilité minime d'incorrection

2) Algorithme de Las Vegas

Temps : aléatoire

Résultat : correct

3) Algorithme d'Atlantic City

Théorie des jeux

♥ Types de jeux...

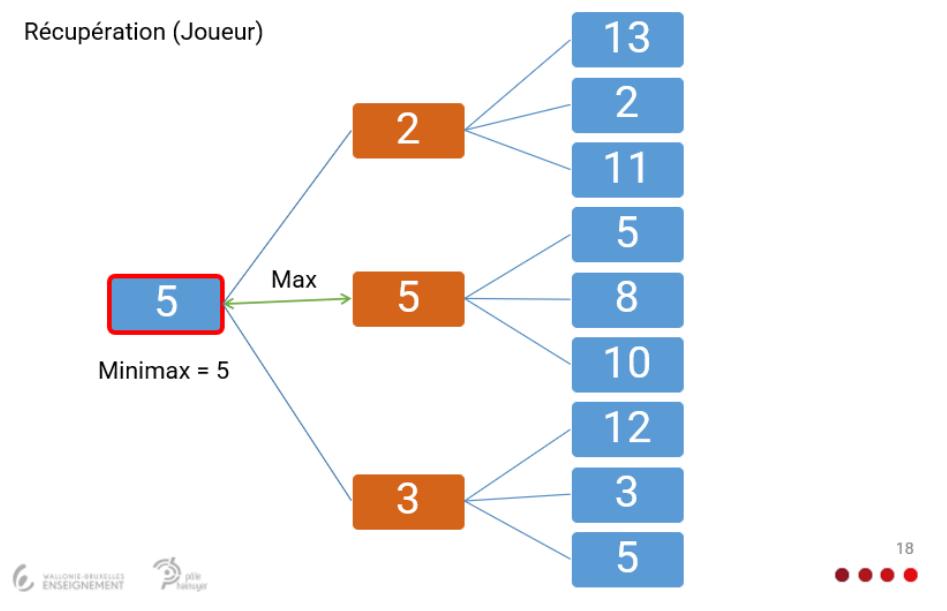
- Jeux à somme nulle (la somme des gains de l'un et des pertes de l'autres est nulle)
- Jeux à somme non-nulles (la somme des gains de l'un et des pertes de l'autres est positives ou négative.)
- (d'office) Jeux impliquant des décisions complexes qui influent sur les gains ou pertes des joueurs, et sur le résultat final.

♥ Algorithmes de recherche adversariale...

1) Minimax

- Pour les jeux à somme nulle
- **Minimiser la perte potentielle maximale**
- Jeux à deux joueurs tour à tour
- Joueur appelé maximisant, opposant appelé minimisant
minimisant car il veut minimiser ses pertes, maximisant car il veut maximiser ses gains
Donc le maximisant va prendre la solution dont le minimisant perd le plus, afin qu'il(maximisant) maximise ses gains

Récupération (Joueur)



2) Élagage αβ

L'élagage Alpha-Bêta est une optimisation de l'algorithme Minimax. Il réduit le nombre de nœuds évalués dans l'arbre de décision, ce qui améliore l'efficacité sans modifier le résultat final.

Concepts Clés

Alpha : La meilleure valeur que le joueur maximisant peut garantir jusqu'à présent.

Bêta : La meilleure valeur que le joueur minimisant peut garantir jusqu'à présent.

Fonctionnement de l'Élagage Alpha-Bêta

Initialisation : Commencez avec Alpha = $-\infty$ et Bêta = $+\infty$.

Propagation :

Lors du parcours de l'arbre, mettez à jour Alpha et Bêta.

Si $\text{Bêta} \leq \text{Alpha}$: Élaguer (ne pas explorer) cette branche car elle ne peut pas influencer la décision finale.

♥ Backtracking...

- Problème de satisfaction de contraintes
- Construire une solution de manière récursive, étape par étape, en retirant les solutions qui ne répondent pas à la contrainte
- ≡ Parcours en profondeur de l'arbre de décision d'un problème

Chiffrement

♥ Objectifs...

- Confidentialité (cacher les données, modifier accessibilité)
- Authenticité (Assurer que la donnée soit la bonne donnée, envoyée par X)
- Intégrité (La donnée arrive complète)

♥ Techniques de base

- 1) Substitution (César...)
Changer un élément par un autre
- 2) Transposition
Swap des éléments.
- 3) Par blocs
Réaliser des opérations spécifiques sur des blocs complets (DES, AES)
- 4) Par flux
Chiffrer séparément les flux RC4

♥ Exemples...

Caesar

Vigenère

Playfair

♥ Chiffrements...

- 1) *Symétrique*
La clef de chiffrement est la même que la clef de déchiffrement
- 2) *Asymétrique*
Pour chiffrer et déchiffrer les clefs sont différentes. Clefs privées et publiques.
En gros, le communiquant chiffre avec notre clef publique et on déchiffre avec la clef privée. A

l'inverse on chiffre les messages des autres personnes avec leurs clef publiques et ils déchiffrent avec leurs clefs privées.

- 3) Sécurisation...
 - Différences HTPP/HTTPS
 - VPN

CHIFFREMENT SYMETRIQUE

♥ Algorithmes

- 1) DES (Data Encryption Standard)
(AJOUTER IMAGE)
- 2) 3DES (Triple DES)
- 3) AES (Advanced Data Encryption Standard)
- 4) Blowfish
- 5) Twofish

♥ Mode de chiffrement

- 1) ECB (Electreonic CodeBook)
Divise les données en blocs de taille fixes.
Vulnérable aux attaques de type dico car des blocs identiques sont chiffrés de la même manière.
- 2) CBC (Cipher Block Chaining)
Bloc de données combiné avec le précédent
Attaque dico plus difficile
- 3) CTR (Counter)
- 4) OFB (Output Feedback)

♥ Problèmes

- 1) Symétrique
 - Echange de clefs
 - Distribution de clefs
 - Rotation de clefsSOLUTION :
 - Chiffrement de clef...
 - PKI, Gestionnaire de clefs centralisé

♥ Vulnérabilités

- 1) Brut force
- 2) Attaque par dictionnaire
 - Utilisation d'une liste de mots courants
- 3) Attaque par analyse différentielle
 - Trouver un patterns ou des vulnérabilités dans le chiffrement en comparant clair/chiffré
- 4) Attaque par force brute en temps réel
 - Intercepte le message chiffré et essaye de déchiffrer en temps réel.

♥ Amélioration de la sécurité

- 1) Longueur de la clef plus grande
- 2) Utilisation de clefs aléatoires
- 3) Changer méthode de chiffrement
- 4) Utilisation de clefs uniques

♥ Utilisation chiffrement symétrique

- 1) Fichier
- 2) Disques
- 3) Bases de données
- 4) Chiffrement de communications
- 5) Courriers électroniques

CHIFFREMENT ASYMETRIQUE...

♥ Fonctionnement

- Deux clefs, une publique une privée
 - Clef publique permet de chiffrer le message, la privée permet de déchiffrer.
- Ex : A->B : A chiffre avec la clef publique de B, B déchiffre avec sa clef privée

♥ Algorithmes...

1) RSA

- Basé sur la factorisation de grands nombres premiers.
- Clef publique (n, e)
- Clef privée (d)
- Fonction de chiffrement :

$$f(\text{message.clair}) = \text{message.clair}^e \pmod{n}$$

- Fonction de déchiffrement :

$$g(\text{message.chiffré}) = \text{message.chiffré}^d \pmod{n}$$

- Génération de la clef :

Choisir deux nombres entiers p et q

Calculer $n = p \times q$

Calculer l'indicatrice d'Euler : $\text{indiEuler}(n) = (p-1)(q-1)$

On choisit un nombre premier $e < \text{indiEuler}(n)$ et premier avec $\text{indiEuler}(n)$

Calculer nombre entier $d = e^{-1} \pmod{\text{indiEuler}}$

- Intérêts ?

Rapide à créer mais très difficile à déchiffrer

- 2) Chiffrement EL Gamal
- 3) Chiffrement de Merkle-Hellman

♥ Attaques :

- Brute force
- Wiener
- Hastad
- Time attack
- Adaptive chose cyphertext attack

RegEx

Expressions régulières - chaîne de caractères qui décrit un ensemble de chaînes de caractères

<https://regex101.com/>

[a-z] : les lettres minuscules

[0-9] : les chiffres

[&é »'(è_çà)=] : les caractères spéciaux (à compléter au besoin)

[^a] : ce qui n'est pas dans mon ensemble

Raccourcis

♥ Caractères spéciaux

. : Correspond à n'importe quel caractère sauf un caractère de nouvelle ligne.

^ : Correspond au début de la chaîne.

\$: Correspond à la fin de la chaîne.

* : Correspond à zéro ou plusieurs occurrences du caractère précédent.

+ : Correspond à une ou plusieurs occurrences du caractère précédent.

? : Correspond à zéro ou une occurrence du caractère précédent.

\ : Échappe un caractère spécial (e.g., \. correspond à un point).

| : OU logique (e.g., a|b correspond à a ou b).

♥ Quantificateurs

{n} : Correspond exactement à n occurrences du caractère précédent.

{n,} : Correspond à au moins n occurrences du caractère précédent.

{n,m} : Correspond entre n et m occurrences du caractère précédent.

♥ Classes de caractères

[] : Classe de caractères (e.g., [abc] correspond à a, b, ou c).

[^] : Classe de caractères négative (e.g., [^abc] correspond à tout sauf a, b, ou c).

\d : Correspond à n'importe quel chiffre (équivalent à [0-9]).

\D : Correspond à n'importe quel caractère qui n'est pas un chiffre.

\w : Correspond à n'importe quel caractère alphanumérique ou underscore (équivalent à [a-zA-Z0-9_]).

\W : Correspond à n'importe quel caractère qui n'est pas alphanumérique.

\s : Correspond à n'importe quel espace blanc (espace, tabulation, etc.).

\S : Correspond à n'importe quel caractère qui n'est pas un espace blanc.

♥ Groupes et références

() : Groupe de capture. Permet de capturer une partie de la chaîne pour une référence ultérieure.

(?:...) : Groupe non capturant. Permet de grouper sans capturer.

\n : Référence arrière au n-ième groupe capturé.

Utilité

Numéro de tel \b \+32 \d{3}\d{2}. ?\d{2}. ?\d{2}

(\+32 [0-9]{3} [0-9]{2} [0-9]{2} [0-9]{2})|(0[0-9]{9})|(\+32[0-9]{3}[0-9]{2}[0-9]{2}[0-9]{2}) 😊

\+\d{2} ?\d{3} ?\d{2} ?\d{2} ?\d{2} 😊

\+\d{2} ?\d{3} ?\d{2} ?\d{2} ?\d{2} ?\d{2} |0 ?\d{3} ?\d{2} ?\d{2} ?\d{2} 😞

Email
[^\s]+@[a-z]+\.[a-z]+
\w+@\w+\.\w+ 😎

Adresse

Password (minimum 10 caractères, majuscule, minuscule, chiffre, caractère spéciale)

^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[\W_])[A-Za-z\d\W_]{10,}\$ 😕

username

date

exemples formats : 06/01/2004 ou 16/11/2004 ou 28/02/1865 ou 10/10/-505000

\b((0[1-9]|12)[0-9]3[01])\V(0[1-9]|1[0-2]))\V-?\d+\b