



Structures de données et algorithmes avancés

Organisation

- **Equipe pédagogique :**

Grégory Morel / gregory.morel@cpe.fr / Bureau B127A (Responsable du cours)

Alexandre Saidi / alexandre.saidi@ec-lyon.fr

- **Evaluation :**

Examen sur table : 80 %

Contrôle continu : 20 % (TP, QCM, Interrogation écrite, participation...)

Compte pour 50 % du module « MSPELCOMSC » (avec *Mise en œuvre d'un système à microprocesseur* »)



Pourquoi étudier les algorithmes ?

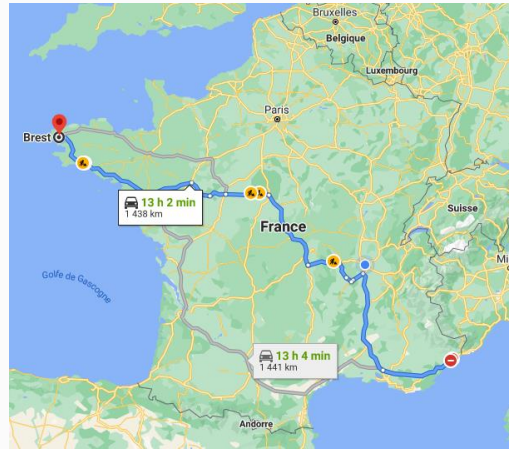
Les algorithmes sont partout et ont une portée universelle

- **Internet** : moteurs de recherche, routage de paquets IP, partage de fichiers...
- **Informatique** : topographie des circuits, systèmes de fichiers, compilateurs...
- **Infographie** : films, jeux vidéo, réalité virtuelle...
- **Sécurité** : smartphone, e-commerce, machines de vote, blockchain...
- **Multimédia** : lecteurs CD, DVD, MP3, JPG, h264, HDTV...
- **Transport** : planification de vols, calculs d'itinéraires...
- **Physique** : simulation à N-corps, simulation de collision de particules...
- **Biologie** : séquençage du génome humain, repliement de protéines...
- **Finance** : trading, bitcoin...



Pourquoi étudier les algorithmes ?

Pour résoudre des problèmes qu'on ne sait pas traiter autrement



« Remplacent » les modèles mathématiques dans la recherche scientifique

$$\begin{aligned}
 E &= mc^2 \\
 F &= ma & F &= \frac{Gm_1m_2}{r^2} \\
 \left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \Psi(r) &= E \Psi(r)
 \end{aligned}$$

Science au 20^è siècle

```

for (double t = 0.0; true; t = t + dt)
  for (int i = 0; i < N; i++)
  {
    bodies[i].resetForce();
    for (int j = 0; j < N; j++)
      if (i != j)
        bodies[i].addForce(bodies[j]);
  }

```

Science au 21^è siècle

« Les algorithmes : une langue commune pour la nature, les hommes et les ordinateurs »

(Avi Wigderson, Prix Abel 2021)



Pourquoi étudier les algorithmes ?

Pour le fun et le profit



Pourquoi étudier les algorithmes ?



Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

-- Linus Torvalds

Objectifs du cours

- **Ecrire du code optimisé et évolutif**

Les connaissances sur les différentes structures de données et algorithmes vous permettent de déterminer la structure de données et l'algorithme optimal à choisir dans diverses conditions

- **Utilisation efficace du temps et de la mémoire**

Avoir des connaissances sur les structures de données et les algorithmes vous aidera à écrire des codes qui s'exécutent plus rapidement et nécessitent moins de mémoire

- **Meilleures opportunités d'emploi**

Des questions sur les structures de données et les algorithmes sont posées systématiquement lors des entretiens d'embauche, en particulier dans les grandes sociétés (GAFAM...)



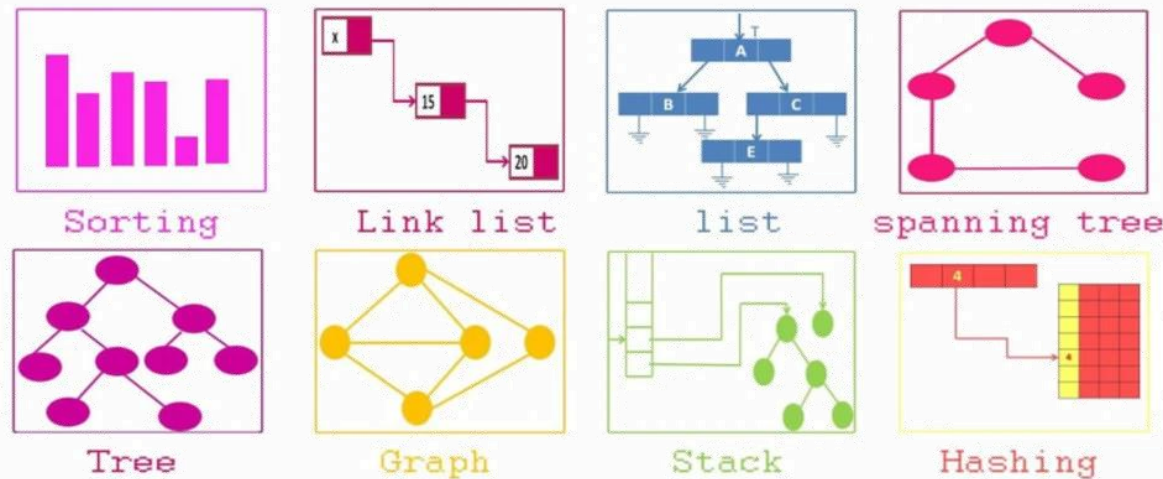
1. Structures de données usuelles (listes chaînées, piles, files, tables de hachage)
2. Introduction à la complexité algorithmique
3. Récursivité / Diviser pour régner / Tris
4. Arbres
5. Théorie des graphes : Bases et parcours
6. Théorie des graphes : Problèmes d'optimisation / Algorithmes génétiques
7. Introduction à la programmation linéaire



Structures de données

Une **structure de données** est une manière particulière d'**organiser des données** de manière à ce qu'elles puissent être **utilisées efficacement**, pour un contexte précis

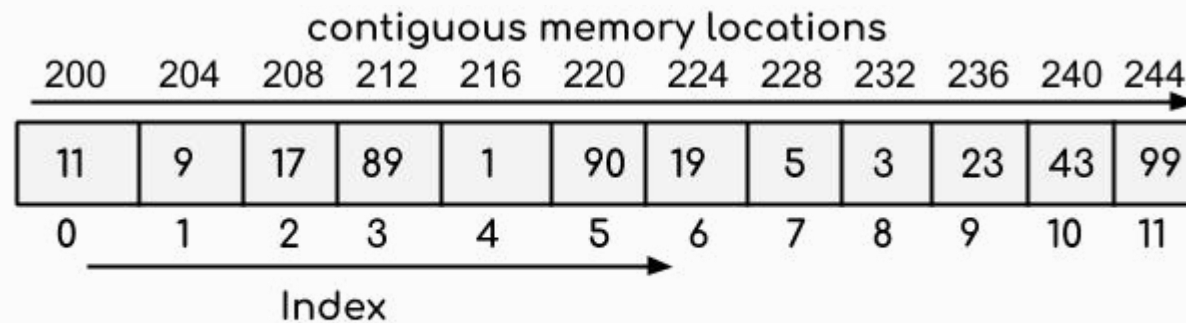
Plus précisément, une structure de données décrit **comment les données sont reliées entre elles**, et les **fonctions / opérations** qui peuvent être appliquées sur les données



By . . . navin.kumar@oprephotography.com



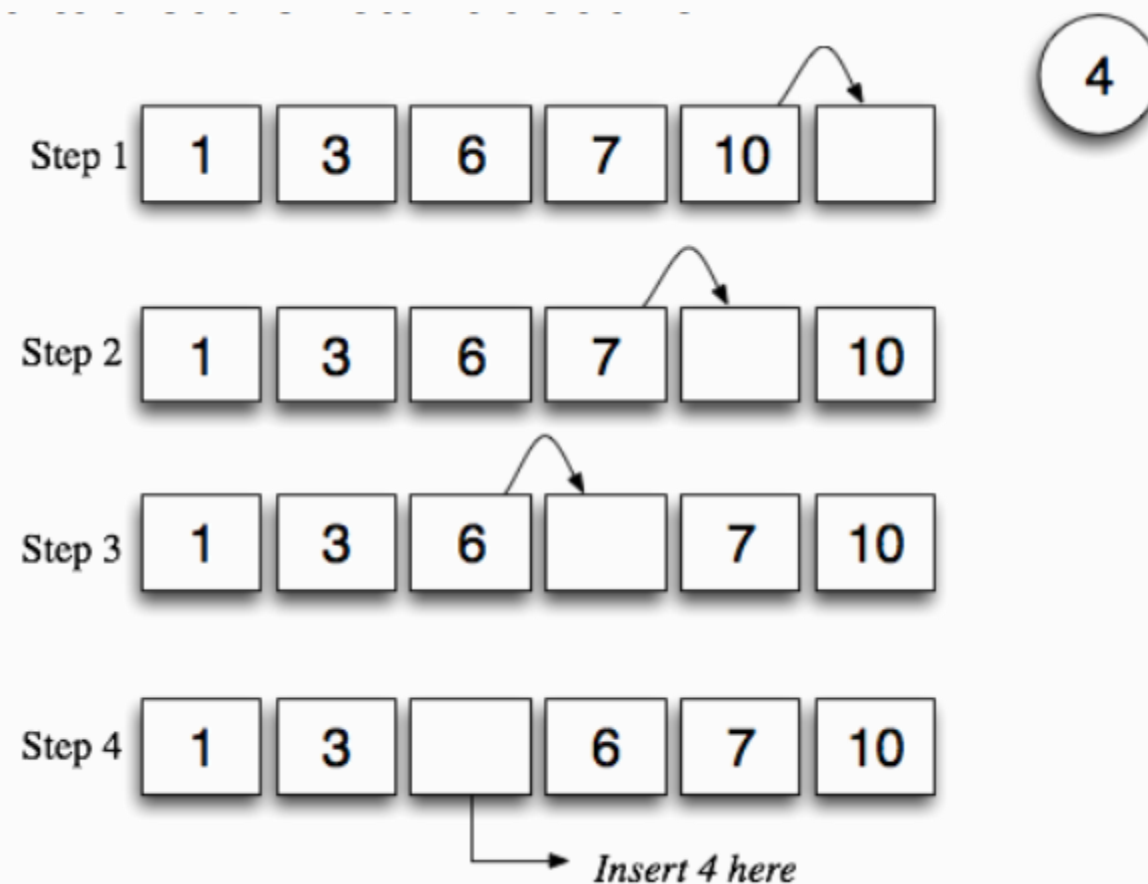
Structure de données la plus simple

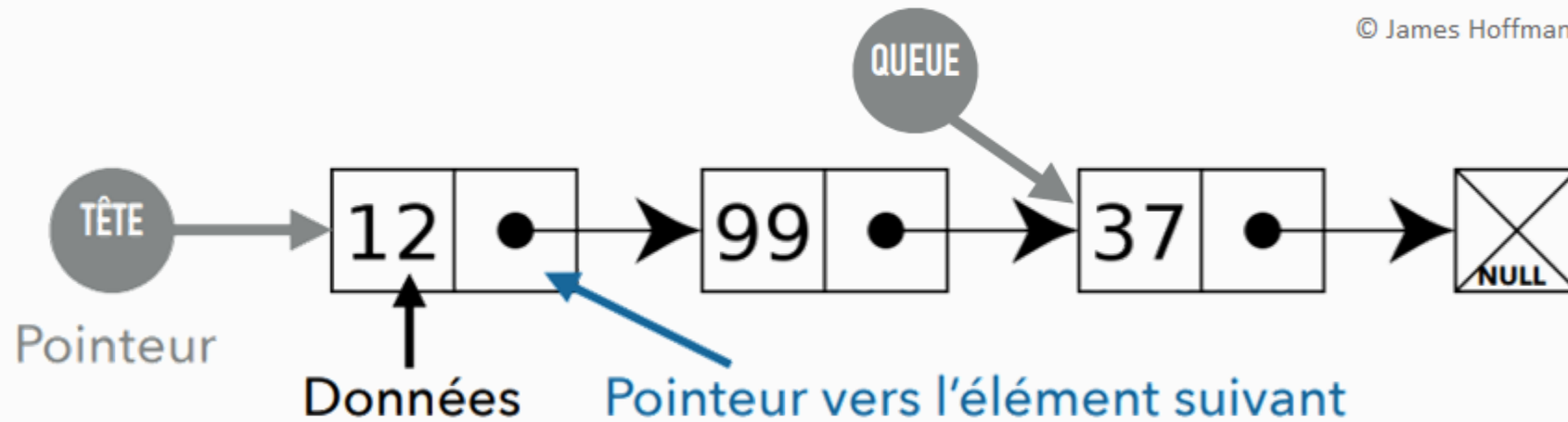


- ✓ données contiguës en mémoire ⇒ accès direct à une case (arithmétique de pointeurs)
- ✗ pour insérer / supprimer un élément, il faut décaler tous les suivants !
- ✗ difficile à redimensionner (il y a peut-être des données juste après le tableau)
 - il faut créer un nouveau tableau plus grand et recopier toutes les valeurs du premier tableau
 - ou allouer de l'espace supplémentaire *au cas où*
- ✗ pas ou mal adapté pour certaines représentations (réseau social, arborescence de fichiers...)



Exemple : insertion d'une valeur dans un tableau trié

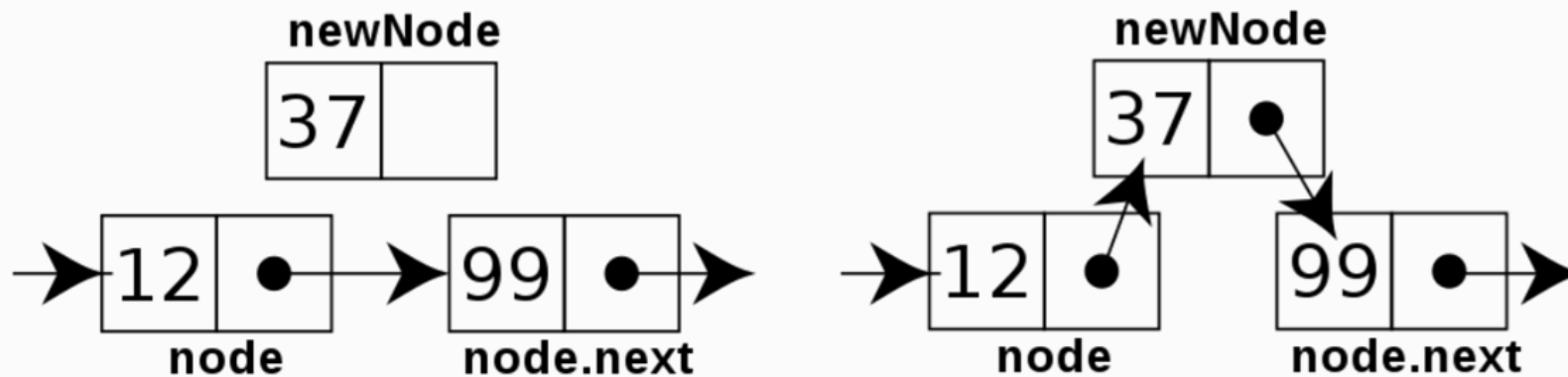




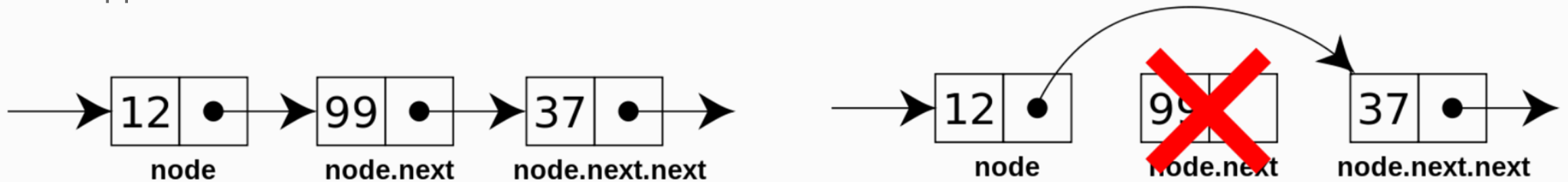
Structure de données *dynamique*

- ✓ conçue pour facilement *insérer* / *supprimer* / *redimensionner*
- ✗ mais difficile *d'accéder* à un élément particulier (il faut parcourir la liste jusqu'à le trouver)
- ✗ difficile à parcourir en *sens inverse*
- ✗ consomme *plus de mémoire* qu'un tableau (à cause des pointeurs)
- ⚠ En cas de perte du pointeur *tête* ⇒ liste perdue ⇒ garbage collector... ou fuite mémoire !

L'insertion est efficace : seulement deux pointeurs à mettre à jour

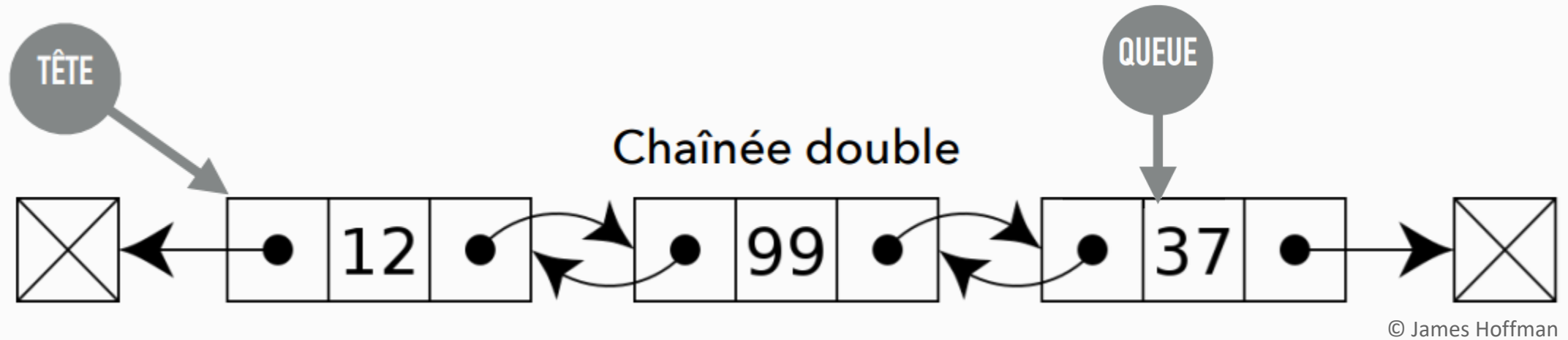


La suppression aussi :



⚠ Le type *List* de Python correspond à un *tableau*, **pas** une liste chaînée

14 Liste doublement chaînée



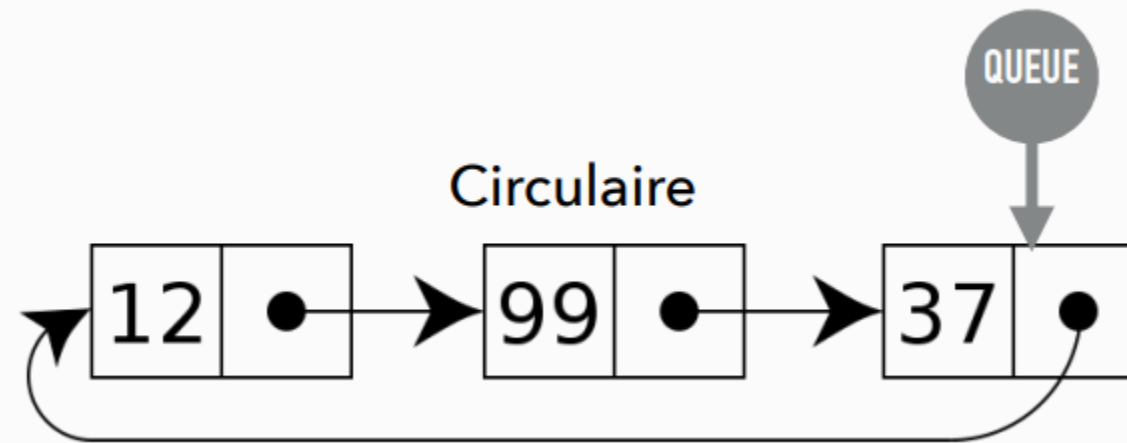
Utilisations :

- Gestion de **playlists**
- Gestion d'un mécanisme « **Annuler / Refaire** » dans un programme
- ...

⚠ Consomme encore (un peu) plus de mémoire !



Liste chaînée circulaire



✓ Parcours en **boucle** plus facile à gérer (pas besoin de regarder constamment si c'est le dernier nœud)

💡 On garde généralement une référence sur le **dernier** élément :

- ajout en fin de liste immédiat
- on retrouve immédiatement la tête de liste

File (*Queue*) : structure de données *linéaire* (tableau, liste) où :

- la lecture / récupération d'une valeur se fait systématiquement au début
- l'insertion d'une nouvelle valeur se fait systématiquement à la fin

💡 On parle de structure FIFO (**F**irst **I**n, **F**irst **O**ut)

Applications :

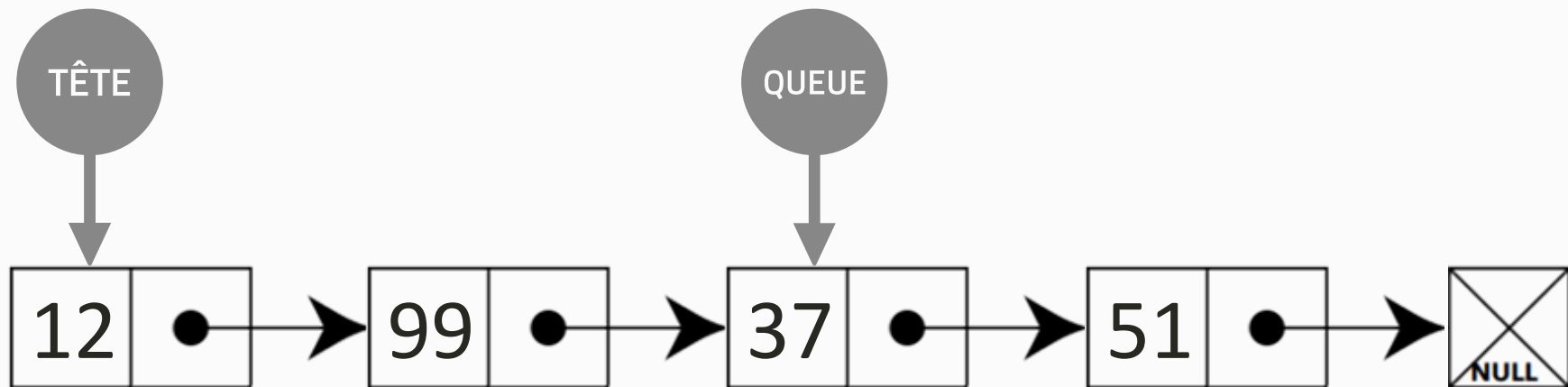
- Toute situation où des données doivent être mémorisées en attendant leur traitement
- File d'impression dans les imprimantes
- En gestion des stocks (on veut faire sortir d'abord les pièces les plus anciennes)
- ...



Implémentation : généralement avec une liste chaînée

Récupération du premier élément

Insertion



Méthodes / Fonctions qu'on rencontre traditionnellement avec une file :

- **enqueue** ajoute un élément dans la file
- **peek** lit la première valeur de la file, **sans la retirer**
- **dequeue** lit la première valeur de la file, **et la retire**
- **size** nombre d'éléments dans la file
- **isEmpty** indique si la file est vide

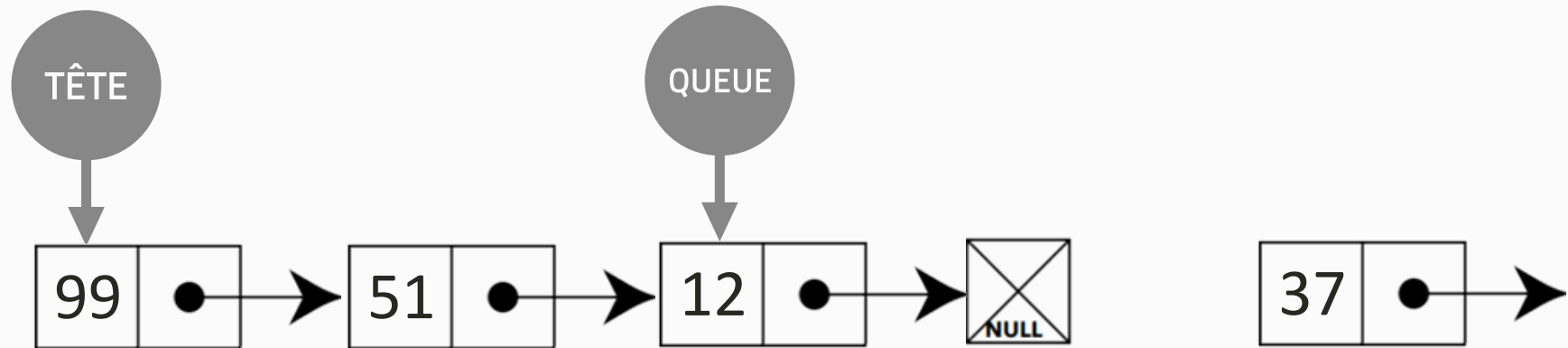


Variante : **file de priorité** (*Priority queue*)

⇒ on peut spécifier où insérer un nouvel élément

⇒ ou (plus flexible) chaque élément est associé à une valeur de priorité

Exemple (chaque valeur correspond à la priorité) :



Applications :

- Ordonnanceurs de systèmes d'exploitations / de processeurs
- Algorithme de Dijkstra (cf. Cours 6)

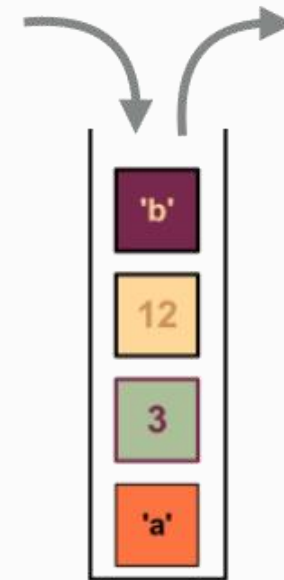
Pile (*Stack*) : structure de données *linéaire* (tableau, liste) où :

- La lecture / récupération **et l'insertion** d'une valeur se font systématiquement au début

💡 On parle de structure LIFO (**L**ast **I**n, **F**irst **O**ut)

Applications :

- Annuler / Refaire
- Récursivité
- Historique dans un navigateur
- Vérification de code (ouverture / fermeture des balises / accolades)



© James Hoffman

Méthodes / Fonctions qu'on rencontre traditionnellement avec une pile :

- **push** ajoute un élément dans la pile
- **peek** lit la première valeur (sommet) de la pile, **sans la retirer**
- **pop** lit la première valeur (sommet) de la pile, **et la retire**
- **size** nombre d'éléments dans la pile
- **isEmpty** indique si la pile est vide



Tables de hachage

Structure de données de type *tableau associatif*:

chaque *valeur* (ou donnée) d'un tableau ou d'une liste est associée à une *clé* permettant de la retrouver *rapidement* (càd *sans parcourir* le tableau ou la liste)

Exemple : annuaire téléphonique

Key-value pairs (records)

Lisa Smith	+1-555-8976
John Smith	+1-555-1234
Sam Doe	+1-555-5030

Applications :

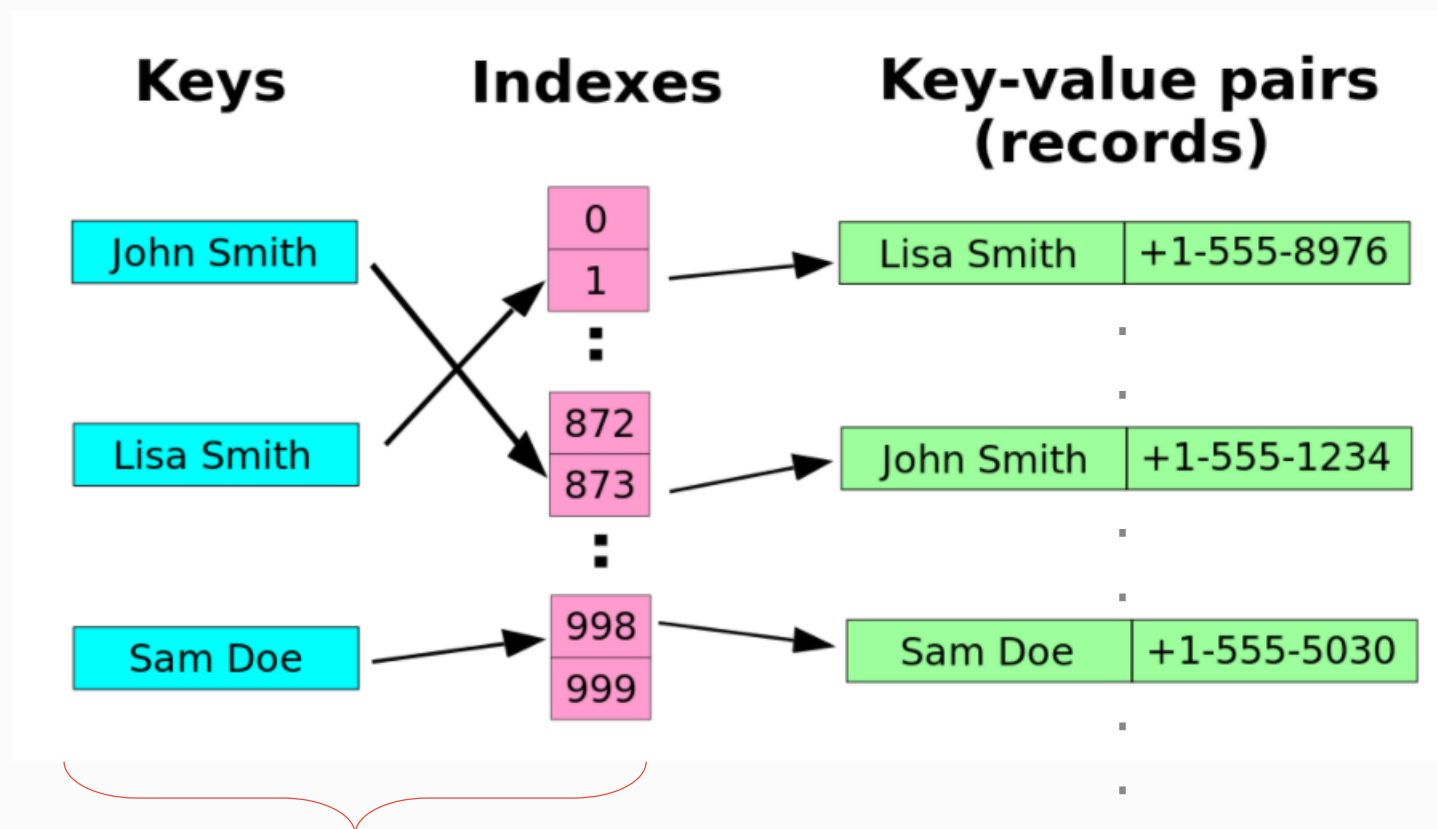
- Toute situation où on veut retrouver rapidement une valeur non indexée par un entier
- Annuaire / Bases de données / Recherche de fichier dans Google Drive...
- Détection de plagiat



Tables de hachage

Fonction de hachage : produit un nombre entier à partir de la clé (son *empreinte*)

Ce nombre sert à identifier la **position** de la donnée dans la table



Fonction de hachage

Tables de hachage

Exemples de fonctions de hachage simples :

- Dans une table de 26 cases, si la clé est un mot, on peut prendre l'initiale puis son rang dans l'alphabet :
 $\text{hash26}(\text{"clavier"}) = 3$ $\text{hash26}(\text{"USB"}) = 21$
- Dans une table de 200 cases, on peut aussi calculer la somme des caractères ASCII du mot puis prendre le reste modulo 200 :
 $\text{hash200}(\text{"clavier"}) = 142$ $\text{hash200}(\text{"USB"}) = 34$

- Images :



Tables de hachage

Problème des fonctions de hachages précédentes : des *collisions* sont possibles !

- $\text{hash26}(\text{"clavier"}) = 3$ mais $\text{hash26}(\text{"carte mère"}) = 3$ aussi !!
- hash200 : si notre liste de clés contient plus de 200 éléments, on aura forcément deux clés ayant la même valeur de hachage !
- Deux images même très différentes peuvent conduire à la même empreinte

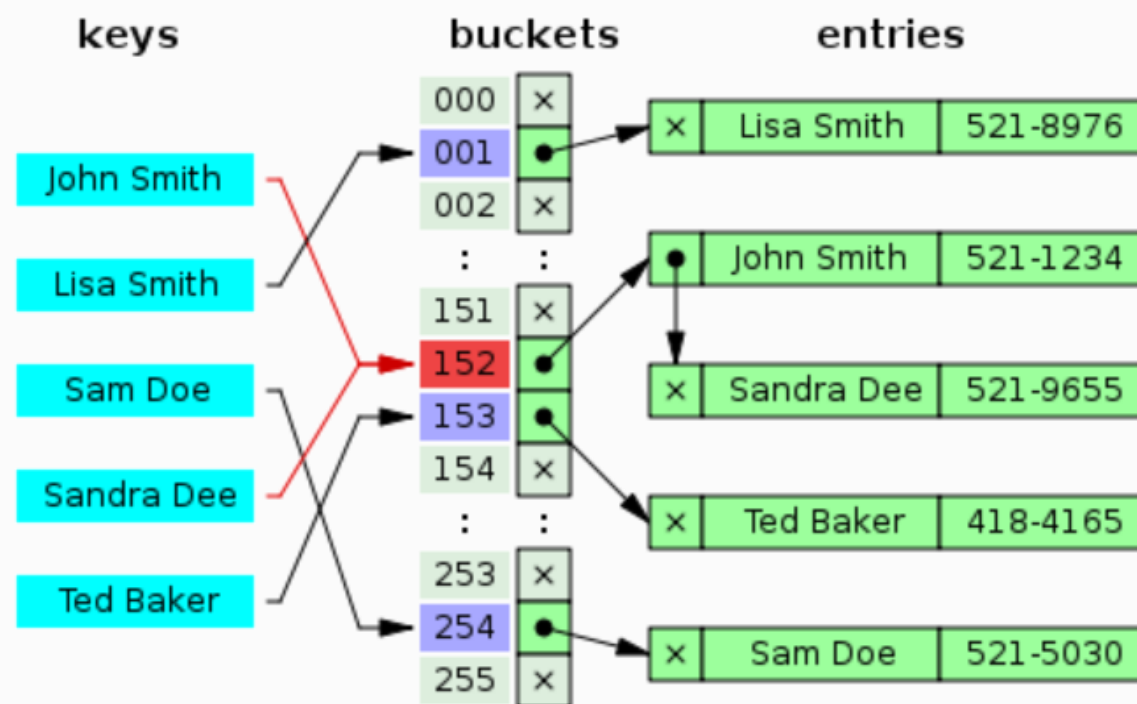


Tables de hachage

💡 Comment résoudre les collisions ?

- *Chaînage* : les données qui ont même valeur de hachage sont stockées dans une liste chaînée

- ✓ simple à implémenter
- ✓ recherche d'une clé facile
- ✓ évite de devoir redimensionner la table en cas de saturation
- ✗ dans le pire cas, la fonction de hachage renvoie toujours la même valeur \Rightarrow la table devient une liste chaînée, et la recherche prend beaucoup plus de temps



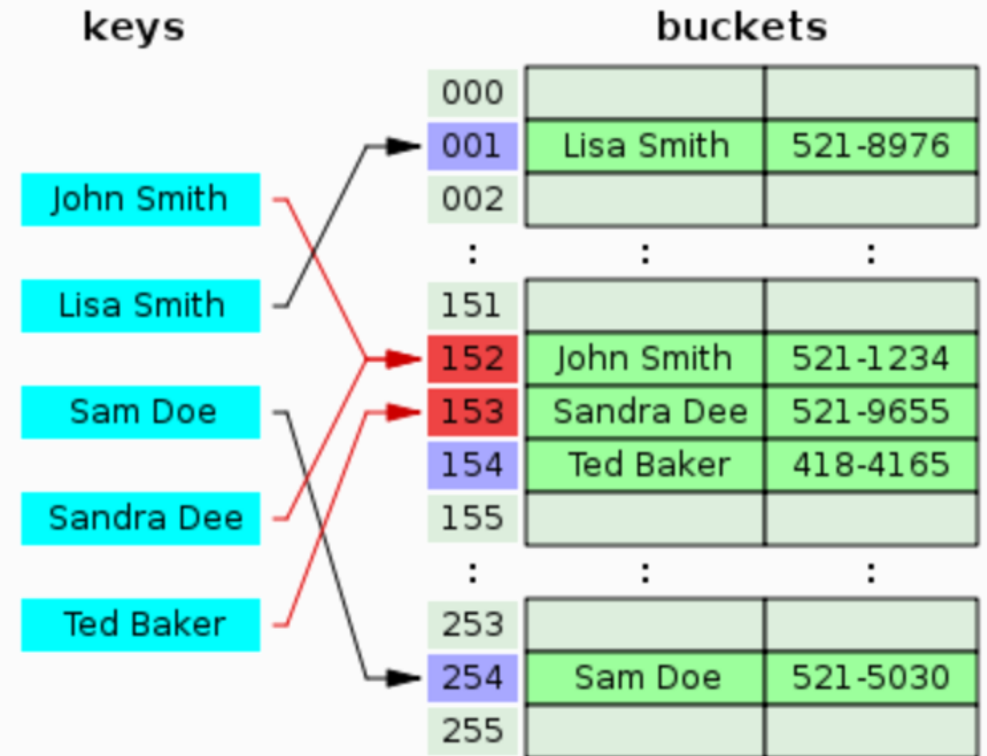
Tables de hachage

💡 Comment résoudre les collisions ?

- **Adressage ouvert** : en cas de collision, on stocke la donnée ailleurs
 - **sondage linéaire** : on parcourt le tableau jusqu'à la prochaine case vide
 - **sondage quadratique** : on parcourt le tableau en doublant d'indice en indice
 - **double hachage** : on applique un second hachage

✓ simple à implémenter ; recherche d'une clé facile

✗ conduit rapidement à une saturation de la table, et la recherche peut demander jusqu'à n opérations



Résolution de collisions par sondage linéaire



Tables de hachage

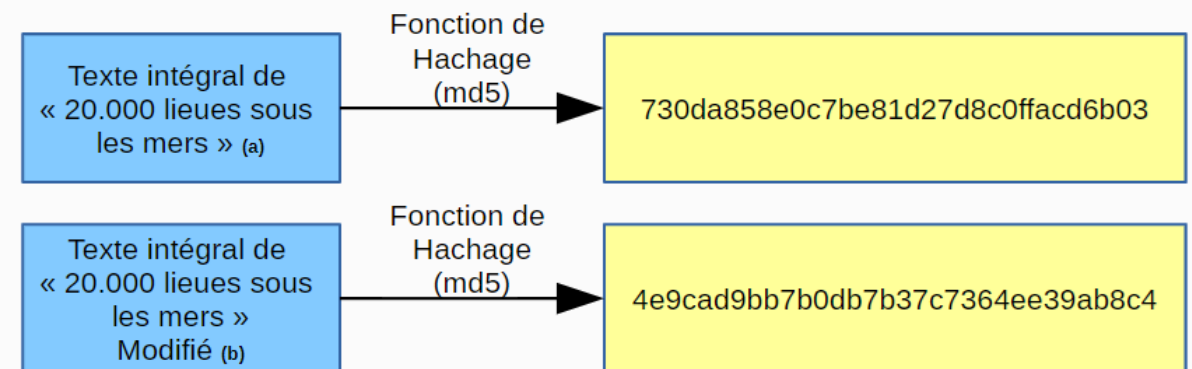
Une *bonne* fonction de hachage doit :

- être *rapide* à calculer
- utiliser *toutes les données* de la clé pour calculer la valeur de hachage
- les valeurs de hachage doivent être *uniformément distribuées*
- générer des valeurs complètement différentes pour des clés presque identiques

Exemples :

MD5 pour l'intégrité de fichiers téléchargés

SHA-256 pour la sécurité de la blockchain Bitcoin



Seule différence : 10e caractère de la 1000e ligne a été remplacé par le caractère '*'

