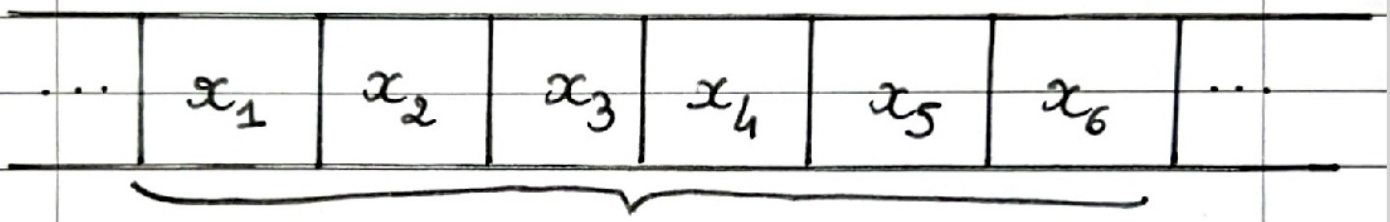
# Structures de données usuelles

## I. Les tableaux

Structure de donnée *primitive*
(elle sert notamment à implémenter d'autres structures)

En mémoire :




en un seul bloc de taille fixée — limites

Que peut-on faire avec un tableau ?

- accéder à un élément ⇝ $O(1)$
- modifier un élément ⇝ $O(1)$

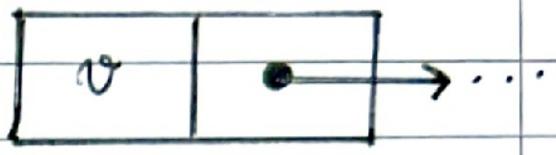
avantage : très rapide

## II Les listes chaînées

Structure "duale" aux tableaux
↪ les limites de l'un sont les avantages de l'autre

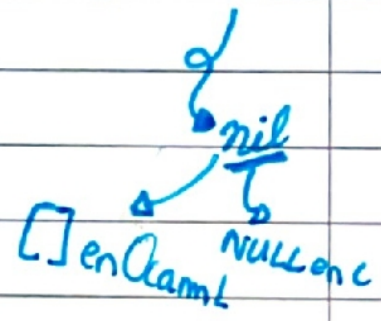
Aussi utilisée pour implémenter d'autres structures.

À la base, il y a un nœud:



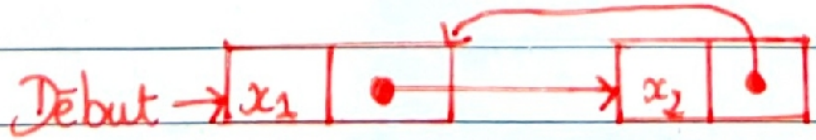

- une valeur v (ou plus)
- un pointeur vers l'élément suivant.

Une liste chaînée est une succession de nœuds.

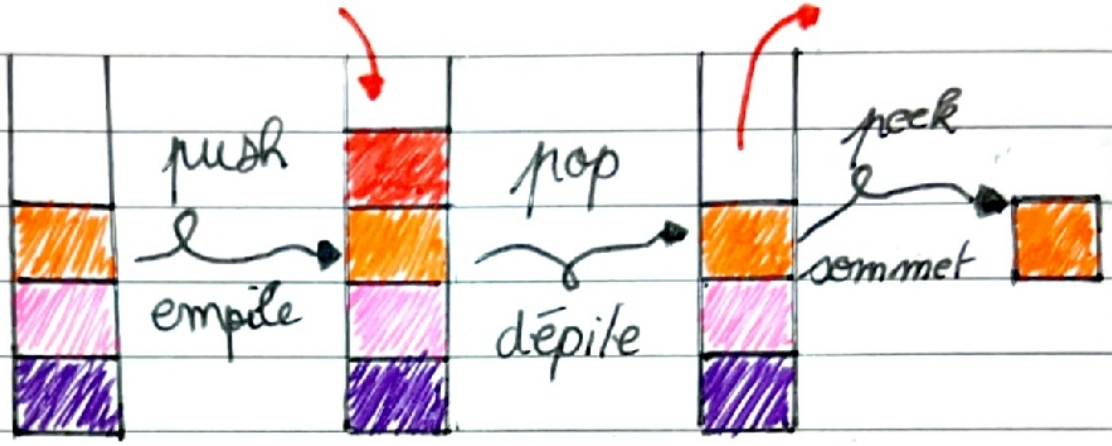

Avantages :
- pas en un seul bloc
- taille variable

Limites :
- accès au i-ème élément en $O(i)$
- de même pour la modification
- ⚠ peut contenir des cycles si mal construite (**boucles ∞**)



# III La pile

Implémentée à l'aide d'un tableau ou d'une liste chaînée.




3 opérations

LIFO
Last in, First out

- push / empiler
  Ex ajoute un élément en haut
- pop / dépiler
  Ex retire l'élément le plus en haut
- peek / sommet
  Ex récupère la valeur de l'élément en haut.

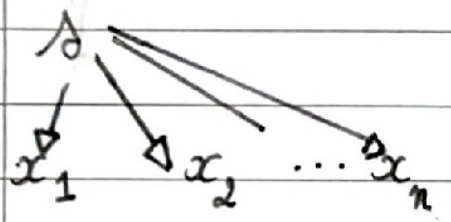
| | TABLEAU | LISTE CHAÎNÉE |
|---|---|---|
| push | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| pop | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| peek | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

↳ si l'on ne dépasse pas la taille maximale

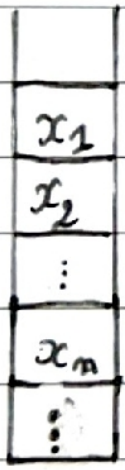
## Quelques utilisations des piles

① pile d'appels (récursif notamment)
↳ Max call stack exceeded
pile d'appels

② parcours de graphes (ou d'arbres) en profondeur (DFS)




Lors du parcours du sommet $s$, on empile $x_n, x_{n-1}, \dots x_1$ (dans cet ordre), puis $s$



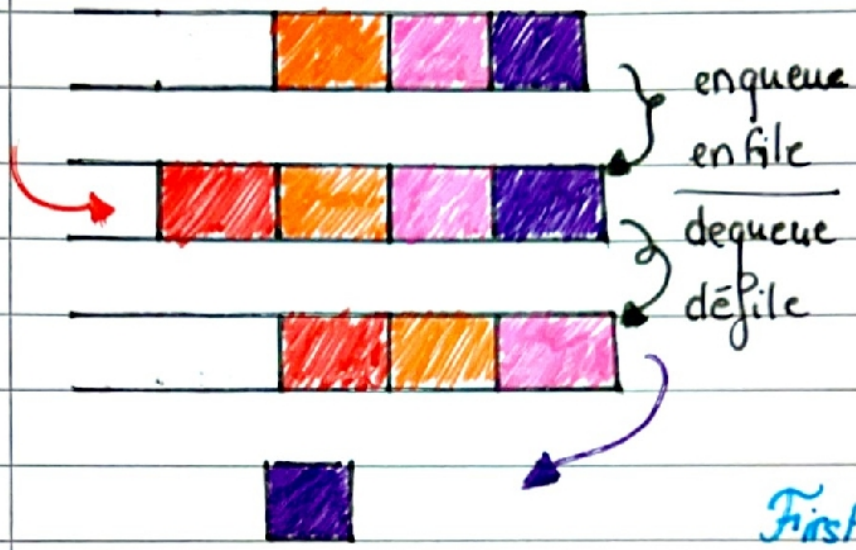

La pile contient l'ordre dans lequel on traite les éléments

### Parcours en profondeur :

- on dépile un élément
- si il a des voisins non traités on les empile.
- on traite le sommet actuel

# IV Les files

Souvent confondues avec les piles, elles sont bien différentes...




FIFO
First in, First out

2 opérations
- enfile / enqueue
  → ajoute un élément à la fin de la file
- défile / dequeue
  → supprime l'élément au début de la File et renvoie cet élément

en stockant le début et la fin de la liste

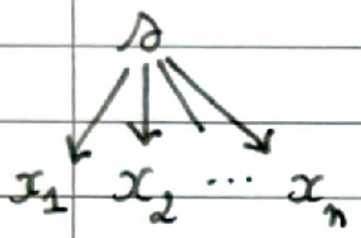
| | Tableau | Liste Chaînée |
|---|---|---|
| enfile | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| défile | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |

si l'on ne dépasse pas la taille maximale

$\mathcal{O}(1)$ avec des buffers (tableaux) circulaires

## Quelques utilisations des files

1. les files de priorités (naïves)
2. lors du parcours d'un graphe (ou d'un arbre) en largeur (BFS)




Lors du parcours du sommet $s$, on enfile les voisins $x_1, \ldots, x_n$ dans cet ordre


la file contient l'ordre dans lequel on traite les éléments

**Parcours en largeur :**

- on défile un élément
- s'il a des voisins non traités, on les enfile
- on traite l'élément actuel

# V Les tables de hachage

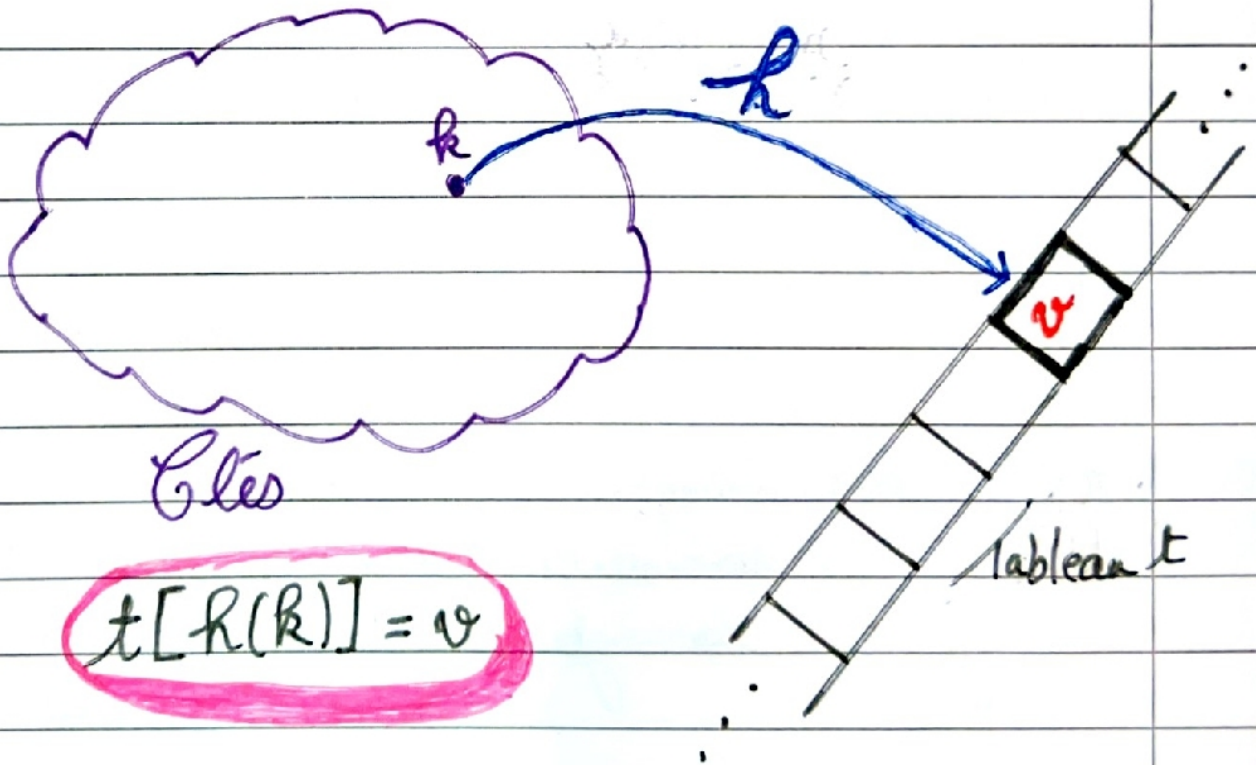
On se donne une fonction

$$h : K \longrightarrow I$$

($K$ : clés ; $I$ : indices, $I \subseteq \mathbb{N}$ et $I$ est fini)

de hachage.

À une clé $k$, on associe l'emplacement $h(k)$ dans un tableau défini préalablement.




⚠ Attention aux collisions (non injectivité de $h$)

$$k \neq k' \text{ et } h(k) = h(k')$$

## 2 opérations

- add / ajouter $O(1)$ en moyenne
  ↳ ajoute une association clé-valeur
- get / récupère $O(1)$ en moyenne
  ↳ récupère la valeur pour une clé

## Implémentations similaires

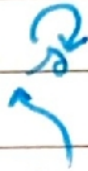
- liste d'associations
- ABR équilibrés

} ↳ pas de collisions ↳ mais moins efficace

## Quelques utilisations des tables de hachages

1. un "dictionnaire" "à la Python"
2. mémoïsation ✓
   ↳ sauvegarder les calculs précédents pour ne pas les refaire

# VI Arbres

sans :  et 

Un arbre est un graphe simple, non-orienté, acyclique et connexe.

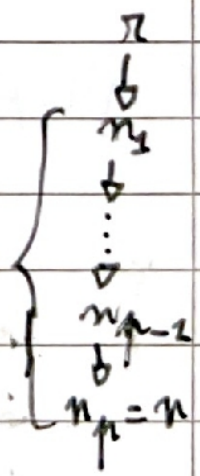
- acyclique → pas de cycles
- connexe → tous les sommets sont accessibles

L'arité d'un arbre, c'est le nombre maximum de fils (directs) pour un noeud donné.

Arbre binaire : arité $\leq 2$
Arbre binaire entier : arité $= 2$
Arbre binaire complet : taille $= 2^k - 1$
La taille d'un arbre, c'est le nombre de noeuds.

La profondeur d'un noeud, c'est le nombre de noeuds entre la racine et le noeud.

$p$ est la profondeur du noeud $n$



La **hauteur** d'un arbre est la profondeur maximale de ses feuilles.

Quelques inégalités sur les arbres binaires

On note :

- $h$ la hauteur $h$,
- la taille $n$,
- le nombre de feuilles $f$.
- le nombre de noeuds internes ou racine $i = n - f$.

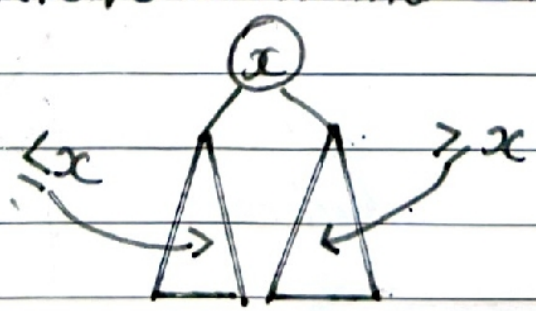
$$(1) \quad f = i + 1$$

$$(2) \quad \lfloor \log_2 n \rfloor \leq h \leq n - 1$$

$$(3) \quad f \leq 2^h$$

## IV. A. Arbres binaires de recherche (ABR)

Propriété fondamentale d'un ABR



Opérations sur un ABR si équilibré

- min / max ⇝ $O(\log n)$
  - toujours à droite / gauche
- recherche x ⇝ $O(\log n)$
  - on compare à la racine et on choisit (gauche / droite)
- insérer x ⇝ $O(\log n)$
  - comme rechercher
- supprimer x ⇝ $O(\log n)$
  - si feuille, ou père d'un seul enfant, c'est simple
  - sinon "on se ramène au cas simple"
    - on trouve min (sous arbre droit) = m
    - on remplace x par m
    - on supprime m du sous-arbre droit

## VI.B. Arbres rouges-noirs

Propriétés Arbre rouge-noir

- feuille ⇝ noire
- racine ⇝ noire
- noeud rouge ⇝ fils noirs
- nb noeuds noirs racine ⇝ feuille est constant : $h_n$(arbre) hauteur noire

## Quelques inégalités

$$h(\text{arbre}) \leq 2\, h_n(\text{arbre})$$
$$n(\text{arbre}) \geq 2^{h_n(\text{arbre})} - 1$$

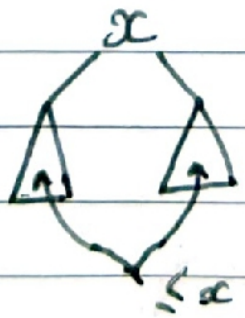
$$h(\text{arn}) \leq 2 \log_2(n(\text{arn}) + 1)$$

L'ajout et la suppression sont des opérations compliquées à décrire simplement sans prendre plusieurs pages...

## II.C Tas (binaires) max

Un tas (binaire) max est

- un arbre binaire
- presque complet (complet partout sauf au dernier niveau)
- ayant la propriété



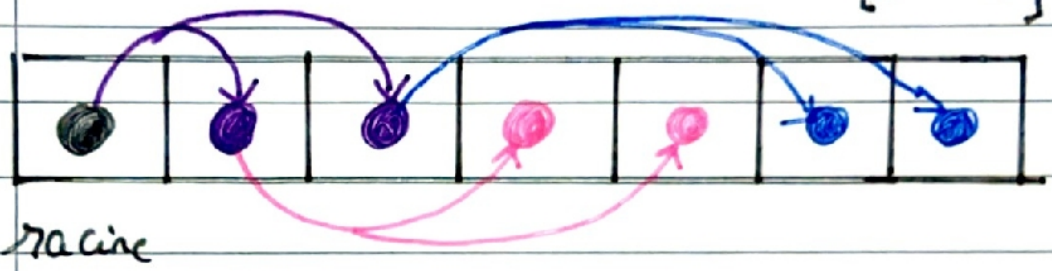

"le max est en racine de chaque sous arbre"

≠ ABR

On peut représenter un tas max dans un tableau $t$ :

- $t[0]$ contient la racine (i.e. le max)
- $t[i]$ a pour fils $\begin{cases} \text{gauche } t[2i+1] \\ \text{droit } t[2i+2] \end{cases}$ (si définis)

Le père de $t[j]$ est $t\left[\left\lfloor \frac{j-1}{2} \right\rfloor\right]$ si $j \neq 0$




Ce tableau correspond à un parcours en largeur du tas.

Insertion en $O(\log n)$
Supprimer max en $O(\log n)$
Valeur du max en $O(1)$
Changer un élément en $O(\log n)$
Tableau en tas max en $O(n)$

Utilisée dans le tri par tas → on ajoute les éléments un par un au tas, on récupère le max $n$ fois.

# VII Union find / Unir & trouver

La structure union find contient les fonctions

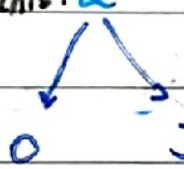
- init_partition qui crée le partitionnement "chacun pour soi"
- find qui trouve un représentant de la classe de l'élément
- union qui fusionne deux classes

On remonte dans l'arbre

On fusionne les arbres

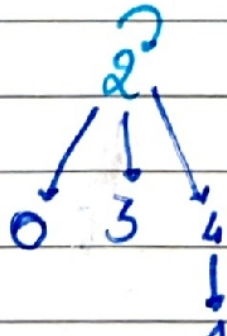
On implémente cette structure par des arbres :

Représentants : 2 4 5




Classe {0, 2, 3}, {1, 4}, {5}.

union (3, 4)




Classe {0, 1, 2, 3, 4}, {5}

## Quelques applications d'Unionfind

1. Algorithme de Kruskal
2. Représentation d'un partitionnement

# VII Les graphes.

Un graphe non orienté $G=(S,A)$ est composé
- de sommets : les éléments de $S$,
- d'arête : les éléments de $A$.

Dans un graphe non orienté, l'arête $x-y$ est représentée par $\{x,y\} \in A$

Un graphe orienté $\vec{G}=(S,\vec{A})$ est composé
- de sommets : les éléments de $S$,
- d'arcs : les éléments de $\vec{A}$.

Dans un graphe orienté, l'arc $x \to y$ est représenté par $(x,y) \in \vec{A}$.

chemin de $x$ à $y$

On note $x \dashrightarrow^* y$ si $x - s_1 - s_2 - \cdots - s_n - y$ dans $G$

$x \dashrightarrow^* y$ si $x \dashrightarrow s_1 \dashrightarrow s_2 \dashrightarrow \cdots \dashrightarrow s_n \dashrightarrow y$ dans $\vec{G}$

$x \dashleftarrow^* y$ si $y \dashrightarrow^* x$ dans $\vec{G}$

Une boucle est de la forme $x \overset{\rightarrow}{\circlearrowright}$ (orienté ou non)

Un cycle est de la forme (orienté ou non)

$$x \dashrightarrow y_1 \dashrightarrow \cdots \dashrightarrow y_n \dashrightarrow x$$

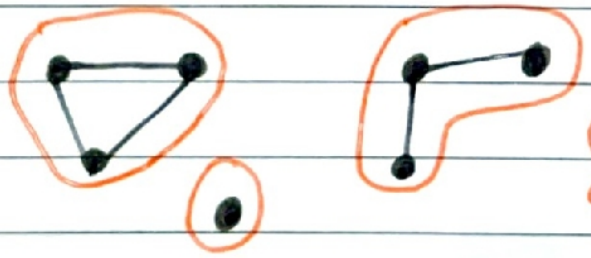
notée $\sim$ dans le cours

On note $\rightleftharpoons$ la relation

$$x \rightleftharpoons y \iff x \dashrightarrow^* y \text{ et } y \dashrightarrow^* x \text{ dans } \vec{G}$$

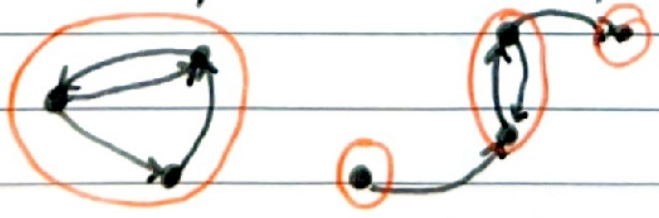
deux chemins différents

Les composantes connexes de $G$ sont les classes d'équivalences de $G$ pour la relation $\dashrightarrow^*$.



Composantes connexes

Les composantes fortement connexes de $\vec{G}$ sont les classes d'équivalences de $\vec{G}$ pour la relation $\rightleftharpoons$



Composantes fortement connexe

Le degré entrant $d_-(s)$ est le nb d'arêtes arrivant en $s$. De même pour le degré sortant $d_+(s)$.

Dans un graphe non orienté, $d_+ = d_-$, on note donc le degré de $s$, $d(s)$.

Formule des degrés (dans $G=(S,A)$ non orienté)

$$\sum_{s \in S} d(s) = 2\,\mathrm{Card}(A)$$

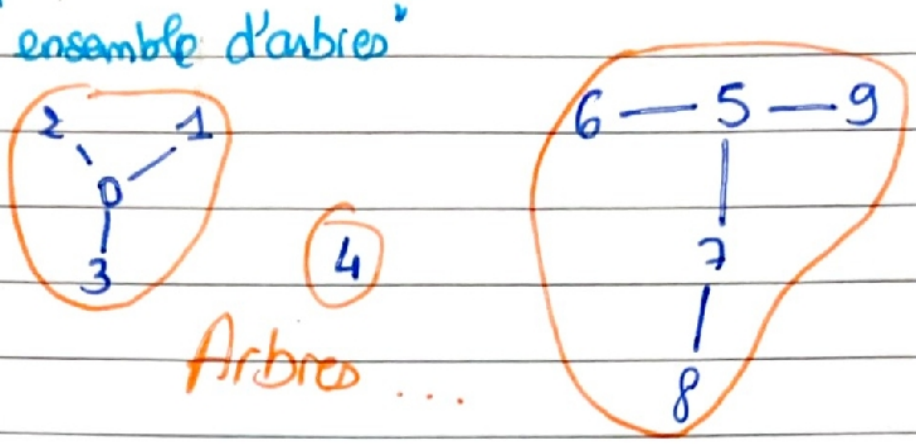
Ça toujours vrai dans $\vec{G}=(S,\vec{A})$ en remplaçant $d$ par $d_+$ ou $d_-$.

Un graphe est complet si, pour tout sommets $x$ et $y$, on ait $\{x,y\} \in A$.

On a toujours $|A| = \mathcal{O}(|S|^2)$

La longueur d'un chemin est le nombre d'arêtes de ce chemin.
La distance entre deux sommets est la longueur minimale d'un chemin $u \rightsquigarrow v$, ou $+\infty$ s'il n'y en a pas.

Une forêt est un graphe acyclique.




Un graphe est pondéré dès lors que l'on dispose de $w : \vec{A} \to \mathbb{R}$, une fonction de pondération.

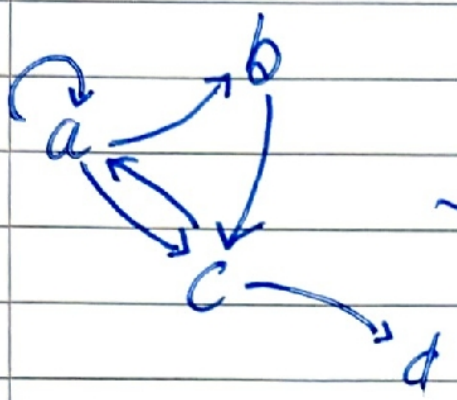
Rq : Dans un graphe non pondéré, on peut définir la pondération $w : a \mapsto 1$.

Rq : D'un graphe non orienté $G = (S, A)$, on peut en déduire un graphe orienté $\vec{G} = (S, \vec{A})$ équivalent en posant

$$\vec{A} = \{ (u,v), (v,u) \mid \{v,u\} \in A \}$$

# Représentation d'un graphe

- par liste d'adjacence




$a \to [a, b, c]$,
$b \to [c]$,
$c \to [a, d]$,
$d \to []$

On peut également représenter un graphe pondéré en ajoutant la pondération des arêtes dans la liste.

- par une matrice d'adjacence

pondérations ici

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{matrix} a \\ b \\ c \\ d \end{matrix}$$

$a \quad b \quad c \quad d$

un graphe est non-orienté ssi A est symétrique

le graphe transposé est représenté par $A^T$.