Introduction aux graphes

L2 informatique – C. Renaud

christophe.renaud@univ-littoral.fr

www-lisic-univ-littoral.fr/~renaud

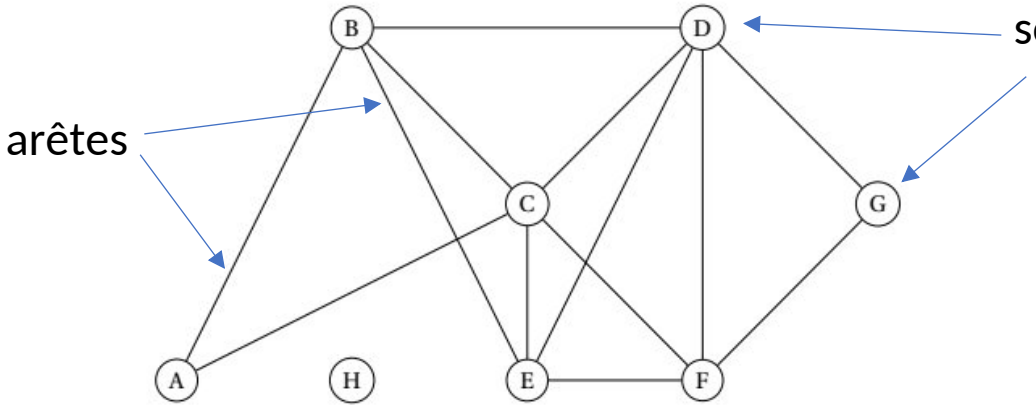
Version 1.7.0 du 11/01/2026

Introduction

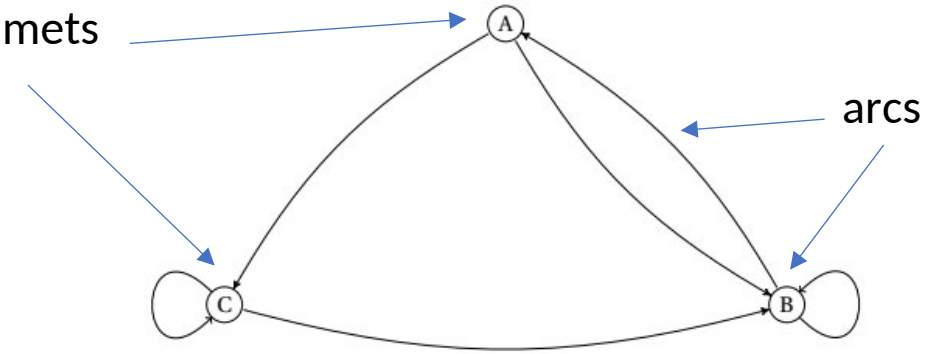
- Objectifs
  - Connaissances de base sur les graphes
  - Compréhension de l'algorithmique dédiée
  - Mise en pratique au travers du langage C
- Evaluation
  - Interrogations de cours
  - Notes de TPs
  - Un examen sur machine
- Documentation
  - www-lisic.univ-littoral.fr/~renaud

Quelques définitions

- **Définition 1** : Un **graphe** est un ensemble de points, appelés **sommets**, pouvant être reliés entre eux par des **arêtes**. Il peut être :
  - **non orienté** : les arêtes ne possèdent pas de sens de parcours;
  - **orienté** : les arêtes possèdent un sens de parcours, représenté sur chacune des arêtes par une flèche. Les arêtes portent alors le nom d'**arcs**.



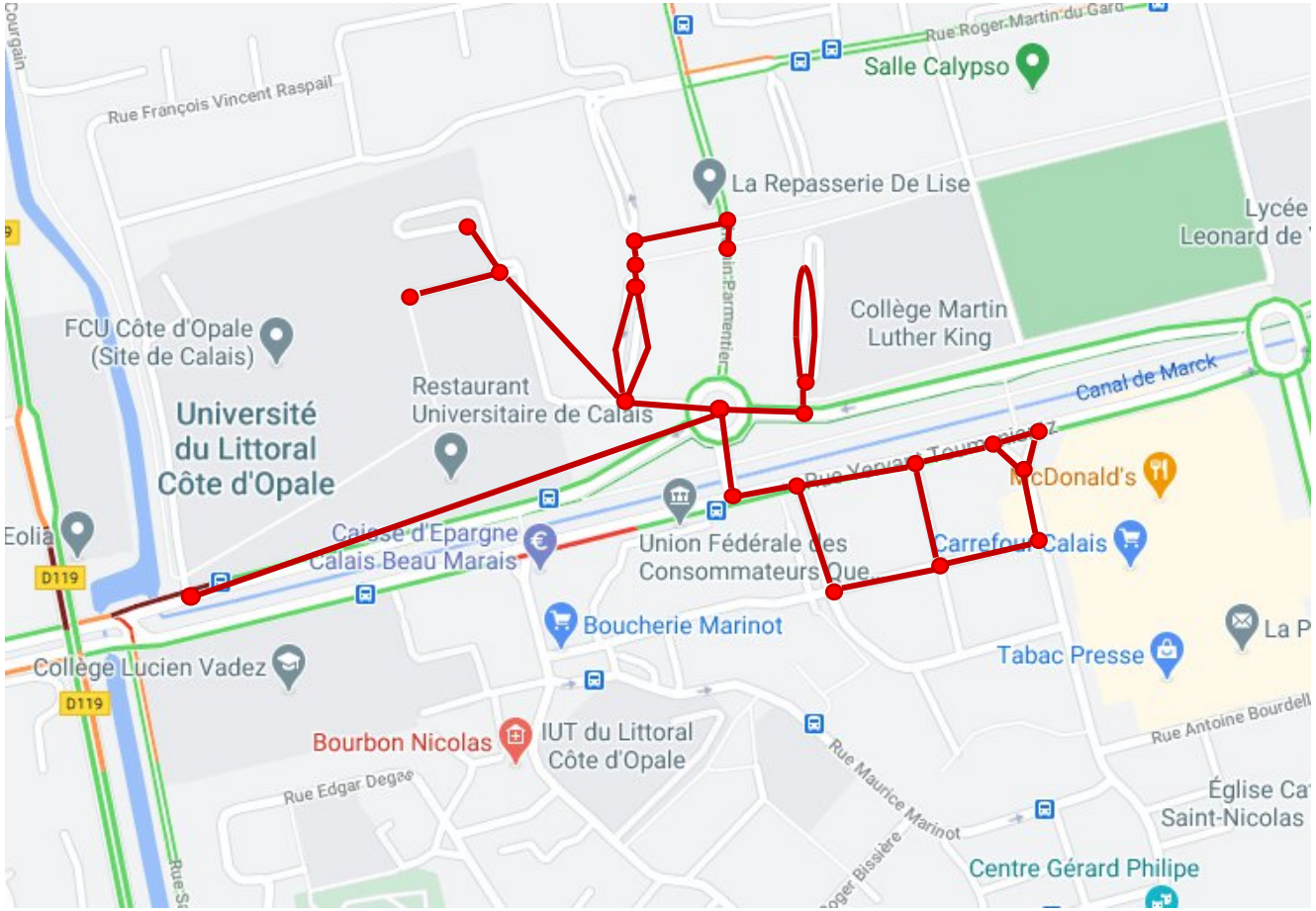
Un graphe non orienté
de sommets S={A,B,C,D,E,F,G,H}



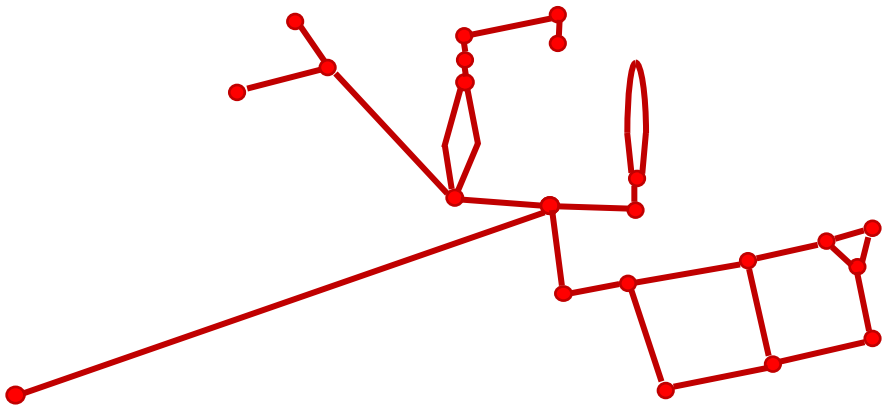
Un graphe orienté
de sommets S={A,B,C}

Quelques définitions

- Exemples : carte routière



Une arête = une voie de cheminement



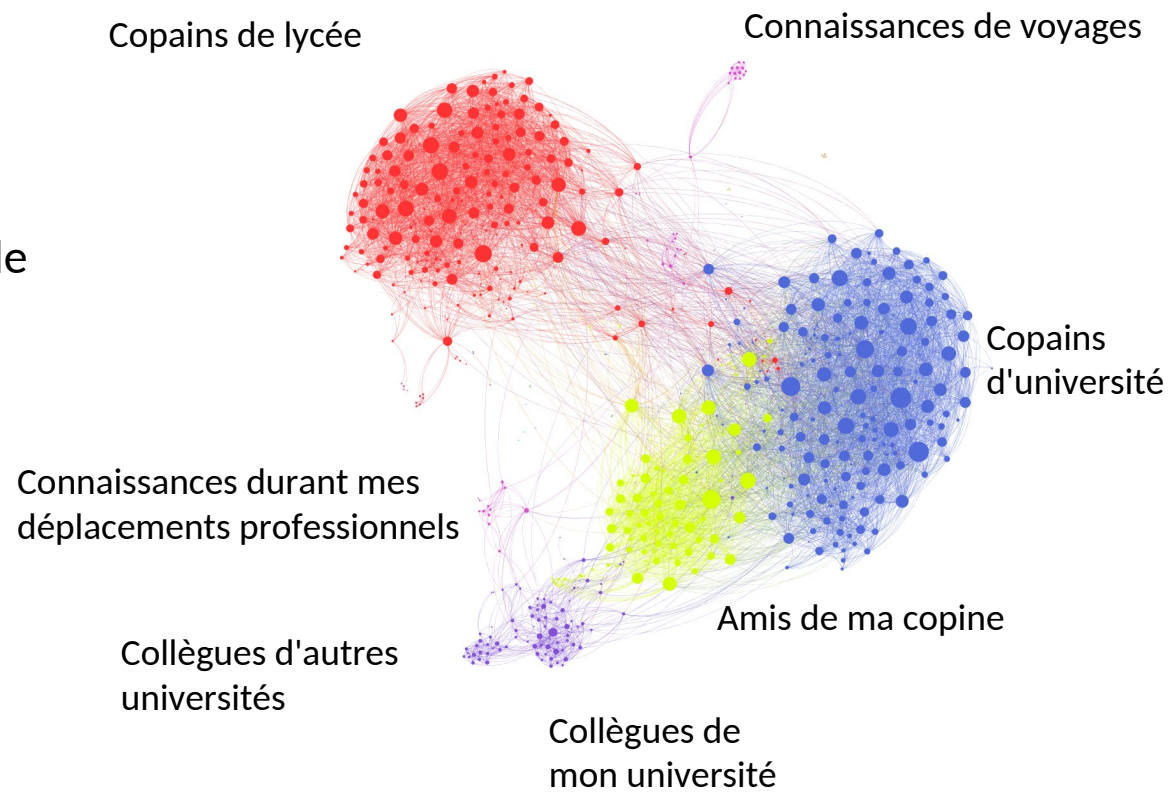
un sommet = une intersection

Quelques définitions

- Exemples : réseau social

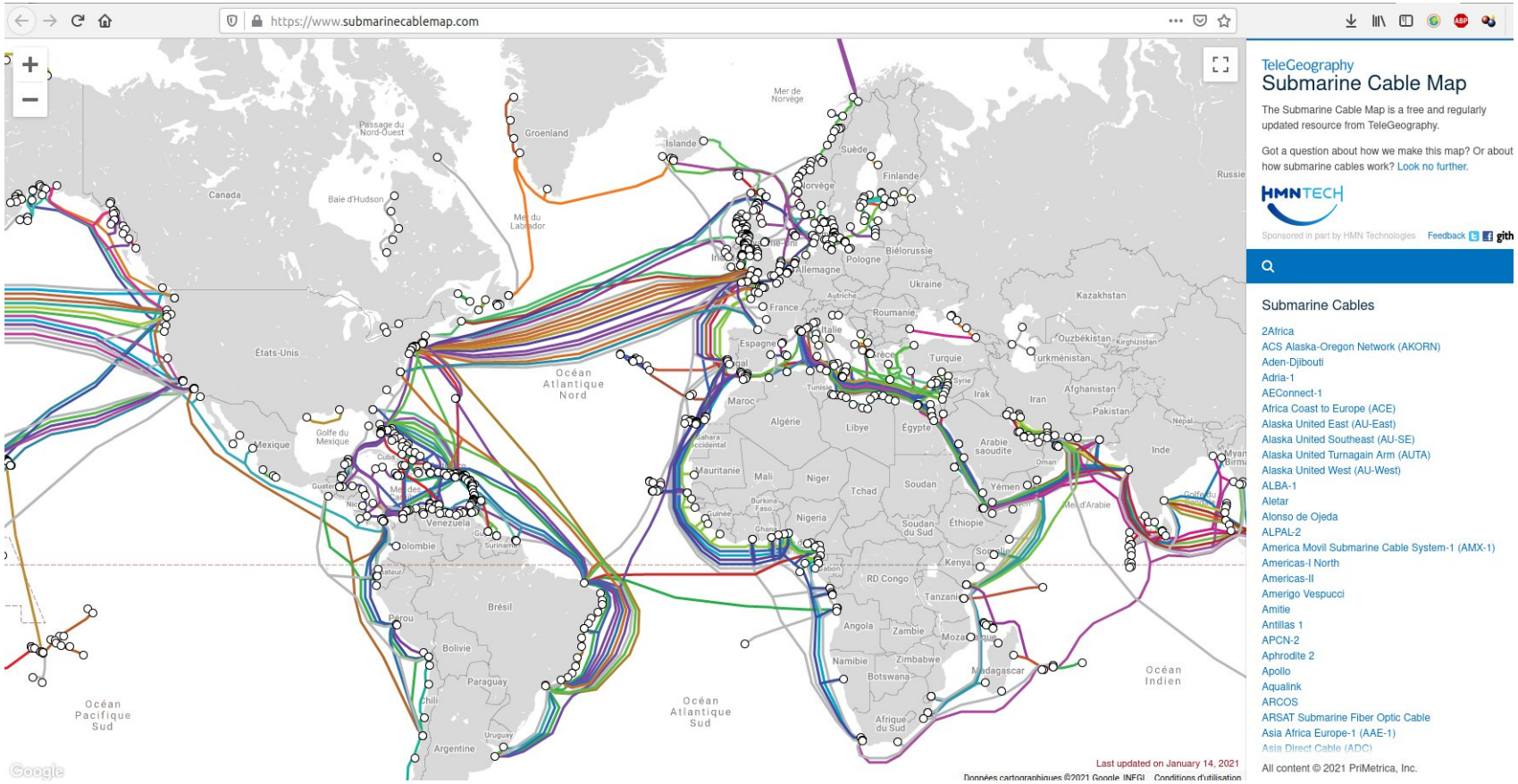
Source : https://griffsgraphs.wordpress.com/2012/07/02/a-facebook-network/

Un sommet = une personne
Une arête = une relation sociale
Une couleur = type de relation



Quelques définitions

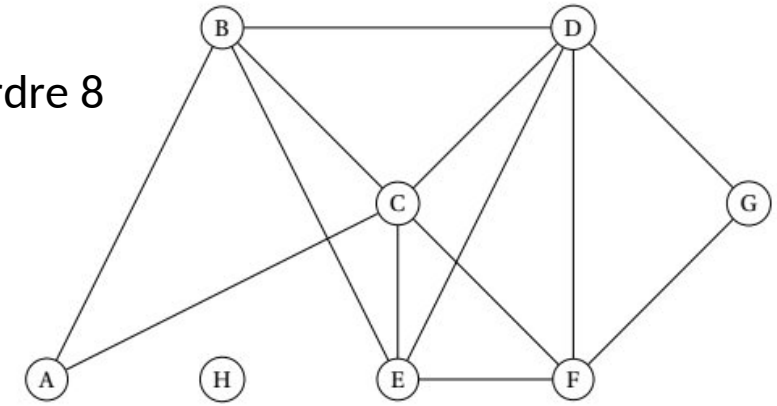
- Exemples : cables internet sous-marins



Quelques définitions

- **Définition 2 :** L'**ordre** d'un graphe est le nombre de sommets qu'il possède.

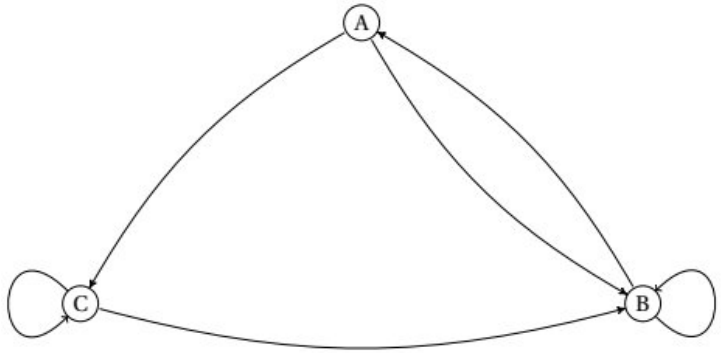
Graphe d'ordre 8



Degré de A = 2
Degré de B = 4
Degré de C = 5
Degré de D = 5
Degré de E = 4
Degré de F = 4
Degré de G = 2
Degré de H = 0

- **Définition 3 :** Le **degré** d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité

Graphe d'ordre 3

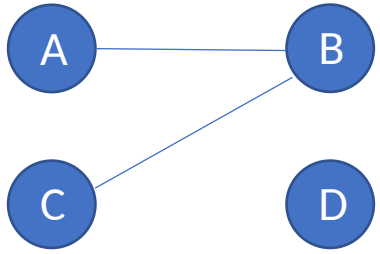


Dans un graphe orienté, on compte pour 1 les arêtes entrantes et pour 1 les arêtes sortantes. Une boucle compte donc pour 2.

Degré de A = 3
Degré de B = 5
Degré de C = 4

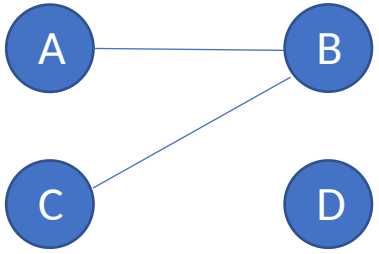
Quelques définitions

- **Définition 4** : Deux sommets distincts sont dits **adjacents** s'ils sont reliés par une arête.

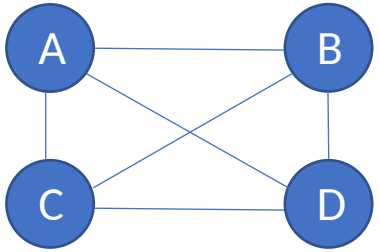


A et B sont adjacents
B et C sont adjacents

- **Définition 5** : Un graphe est dit **simple** si aucun sommet ne possède de boucle et si deux sommets distincts sont reliés par au plus une arête.

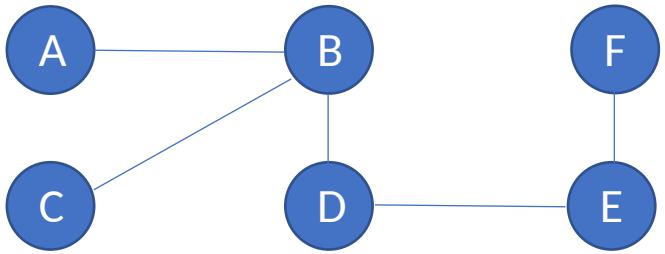


- **Définition 6** : Un graphe d'ordre n >= 1 est dit **complet** si tous ses sommets sont deux à deux adjacents.

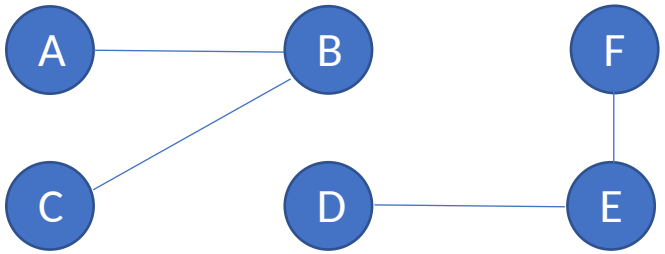


Quelques définitions

•**Définition 7** : Un graphe est dit **connexe** si il est possible, à partir de n'importe quel sommet, de rejoindre tous les autres sommets en suivant les arêtes.

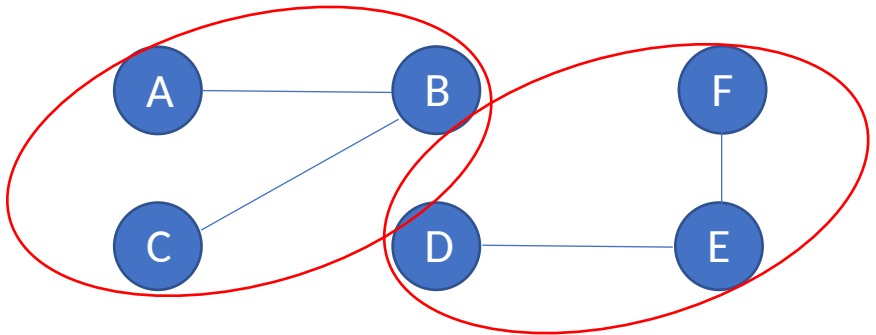


Graphe connexe



Graphe non connexe

•**Définition 8** : Un graphe non connexe se décompose en **composantes connexes**.

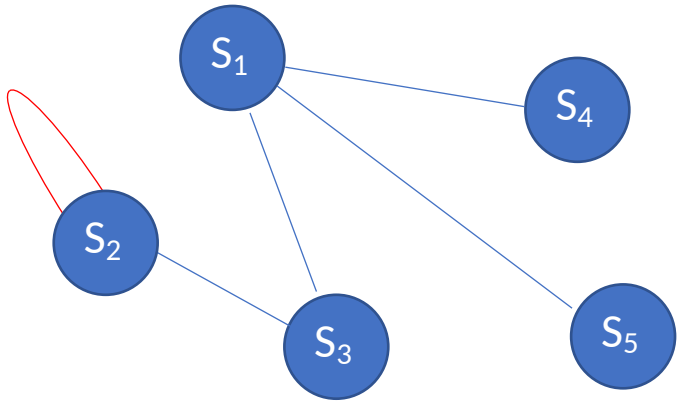


ici deux composantes connexes

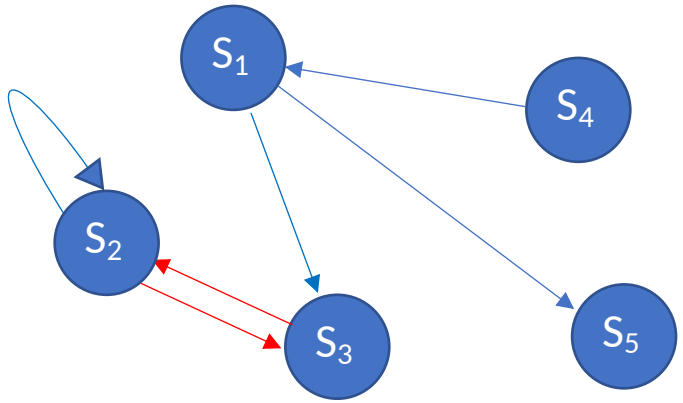
Matrice d'adjacence

Matrice d'adjacence (1)

- À tout graphe d'ordre n, de sommets S_1, S_2, …, S_n, on peut associer une matrice carrée $M=(m_{ij})$ d'ordre n, où le coefficient m_{ij} est égal au nombre d'arêtes reliant le sommet s_i au sommet s_j (en respectant le sens de parcours dans le cas d'un graphe orienté)



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



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

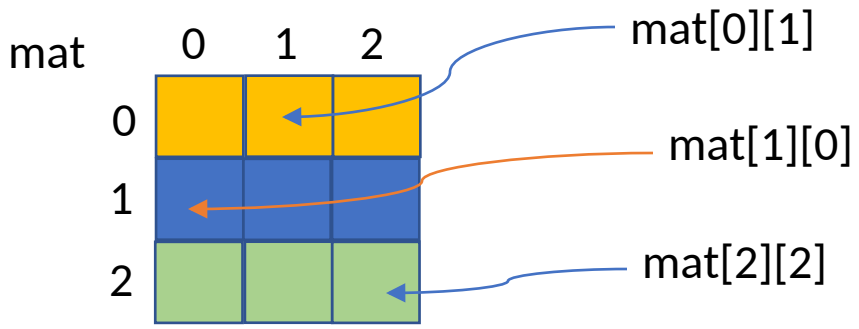
Remarque : La matrice d'adjacence d'un graphe non orienté est symétrique.

Matrice d'adjacence (2)

- Représentation générique en C
  - Dimension connue

```
// exemple : ordre 3

int mat[3][3];
```



Accès à une case mat_{ij} : mat[i][j]

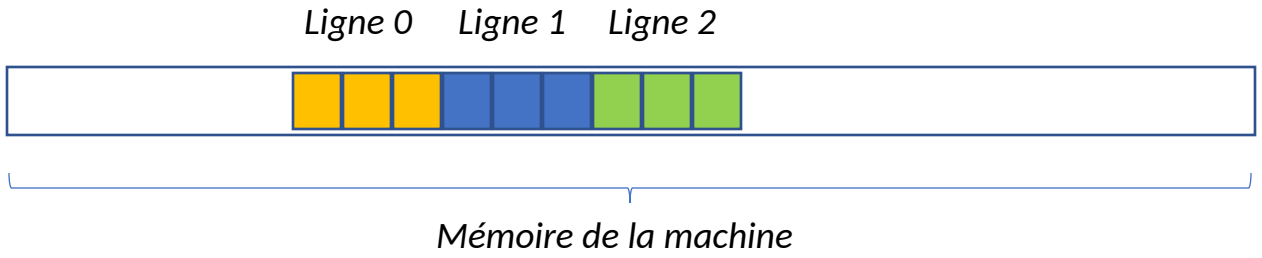
Indice de ligne

Indice de colonne

Indices dans {0, 1, …, ordre-1}

Remarque :

- les lignes sont stockées les unes après les autres en mémoire



Matrice d'adjacence (2)

- Représentation générique en C
  - Dimension inconnue

On doit stocker l'ordre et allouer dynamiquement les cases de la matrice

Accès à une case mat_{ij} : mat.coef[i *mat.ordre + j]

(j^{eme} case de la ligne i ; Début de la i^{eme} ligne)

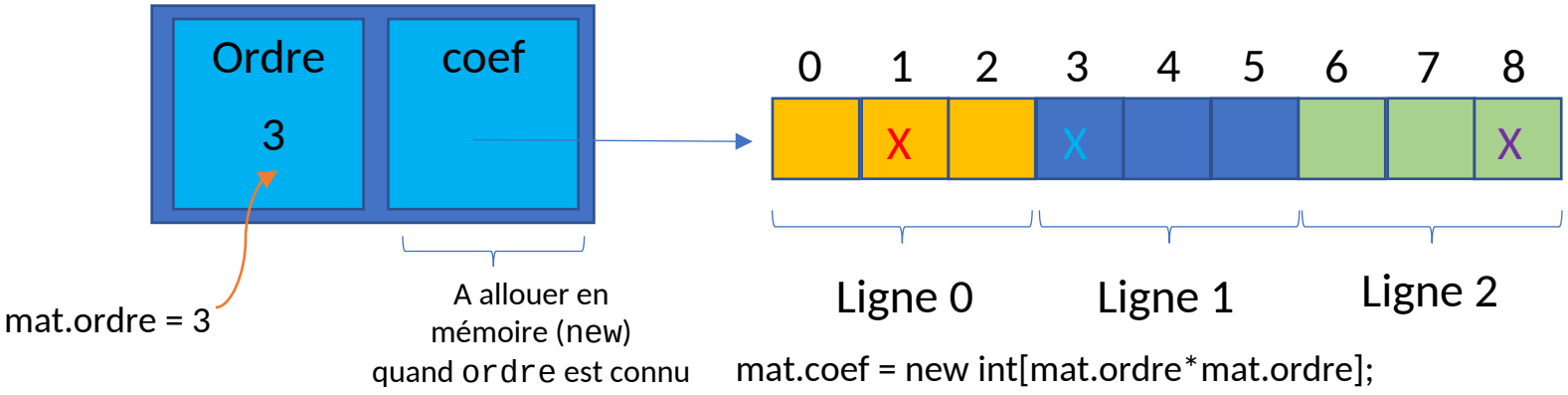
```
struct MatriceAdjacence {
  int ordre; // nombre de sommets du graphe
  int *coef; // les (ordre x ordre) coefficients
};
```

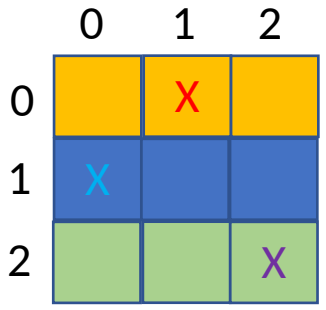
mat_{01} : mat.coef[0 * 3 + 1] = mat.coef[1]

mat_{10} : mat.coef[1 * 3 + 0] = mat.coef[3]

mat_{22} : mat.coef[2 * 3 + 2] = mat.coef[8]

MatriceAdjacence mat;



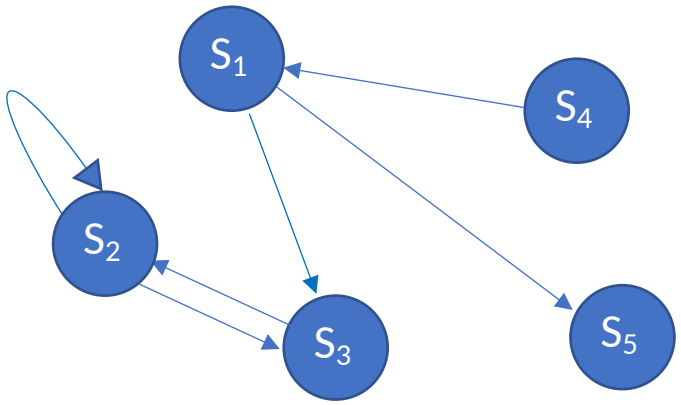


Matrice d'adjacence compacte

Matrice d'adjacence compacte (1)

- Retour sur la représentation matricielle

Avantage : simplicité



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

Inconvénient : coût mémoire

Matrice de taille N x N

1 octet par coefficient

→ N^2 octets

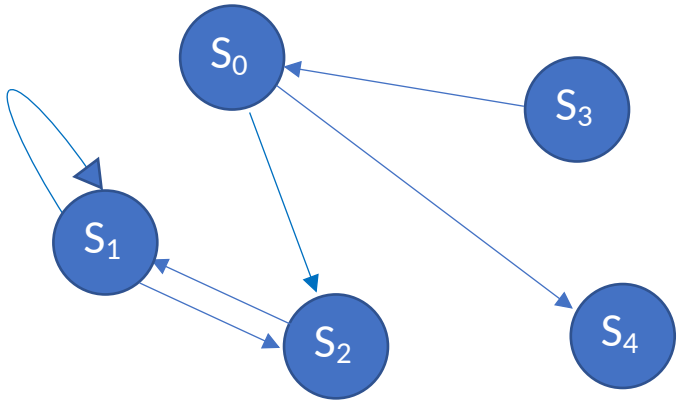
N	Taille mémoire
1000	1 Mo
10000	100 Mo
36000	1.2 Go

Nombre de communes en France
(cf guidage GPS)

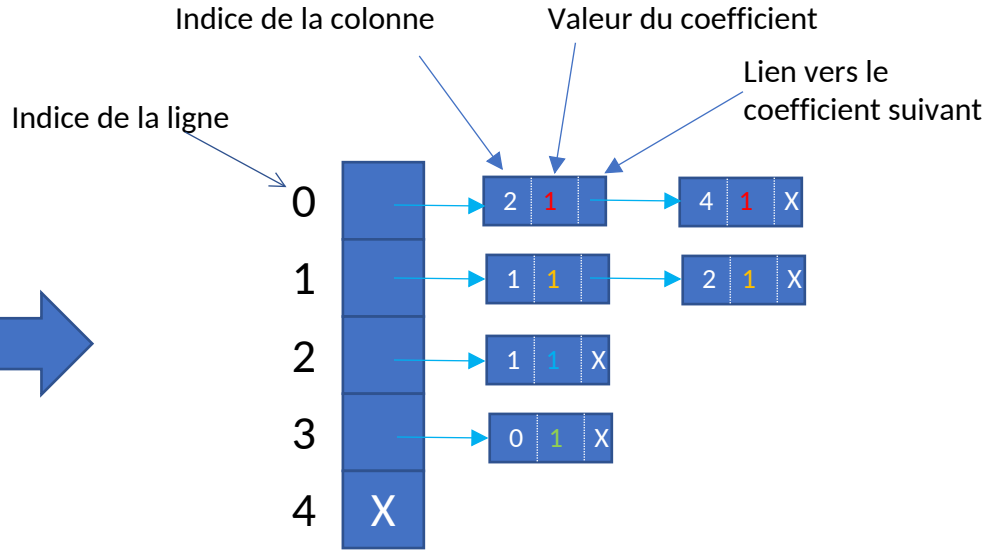
Nombreux 0

Matrice d'adjacence compacte (2)

- Représentation compacte
  - **Matrice creuse**
  - On ne stocke que les coefficients non nuls



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



Une liste chaînée des coefficients non nuls pour chaque ligne

Matrice d'adjacence compacte (3)

- Rappels sur les pointeurs en C

TYPE `*var;`

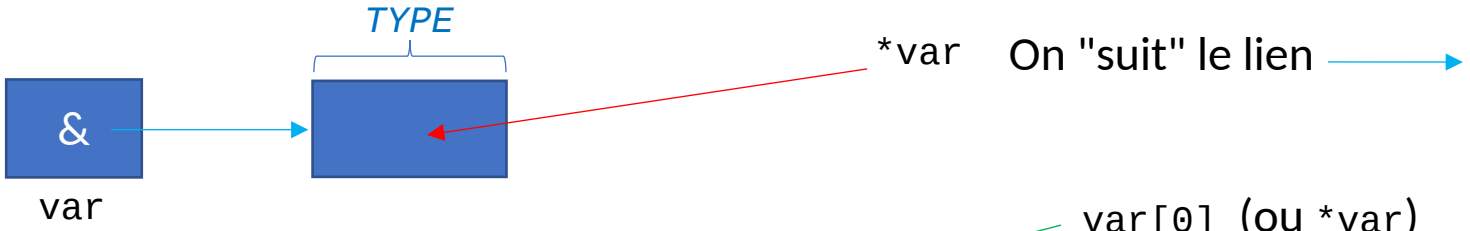
les adresses sont typées

`var` est un pointeur vers une zone mémoire → Contiendra l'adresse du début de la zone

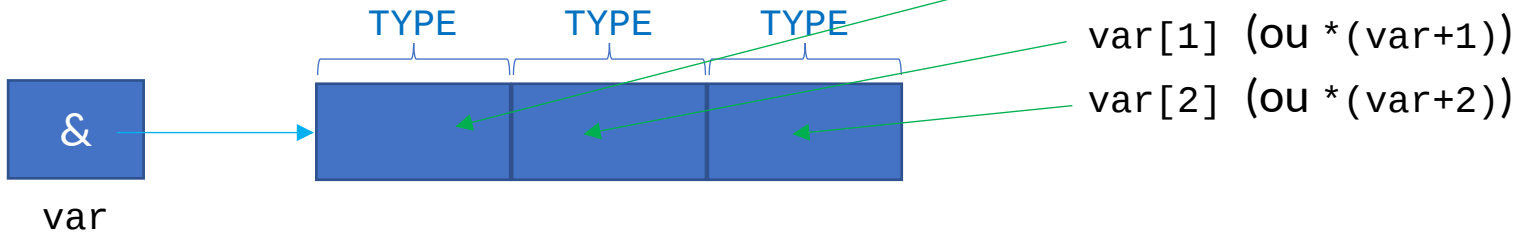
Cette zone mémoire devra contenir un (ou plusieurs) élément(s) de type *TYPE*.

Elle doit être allouée dynamiquement avec l'opérateur `new`

var = new *TYPE*;

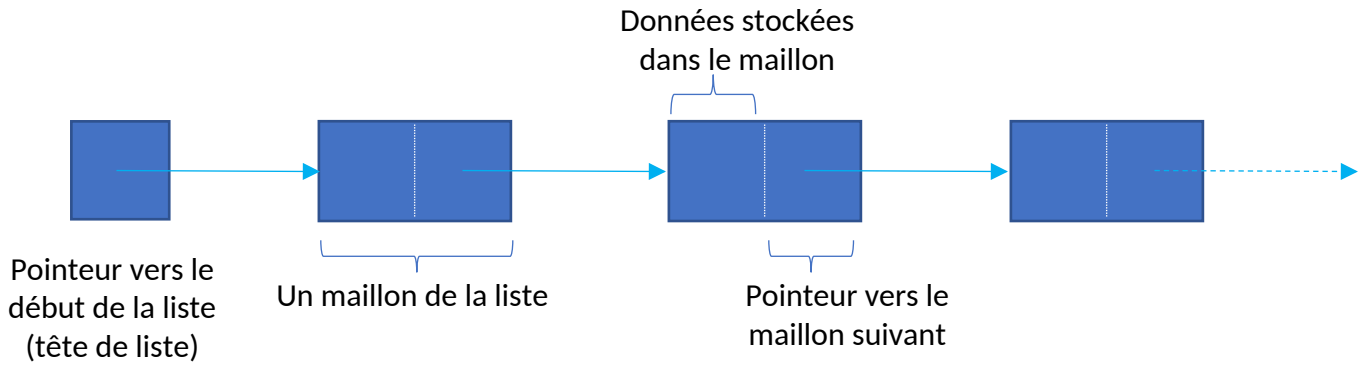


var = new *TYPE*[3];



Matrice d'adjacence compacte (4)

- Rappel sur les listes chaînées
  - Structure dynamique permettant de représenter des listes
  - Longueur inconnue *a priori* (donc tableaux inutilisables)




```
// maillon d'une liste chaînée

struct Maillon {
  Type donnees;   // données stockées dans le maillon
  Maillon *suiv;  // pointeur vers l'élément suivant
} ;
```

Matrice d'adjacence compacte (5)

- Rappel sur les listes chaînées

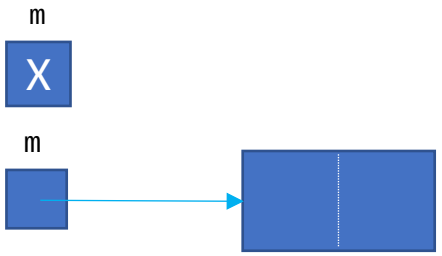
Initialisation d'une nouvelle liste :

```
Maillon *teteListe = nullptr;
```

Création d'un nouveau maillon :

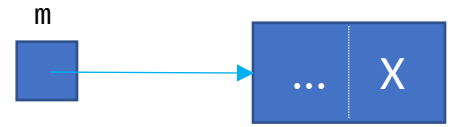
```
Maillon *m;

m = new Maillon;
```



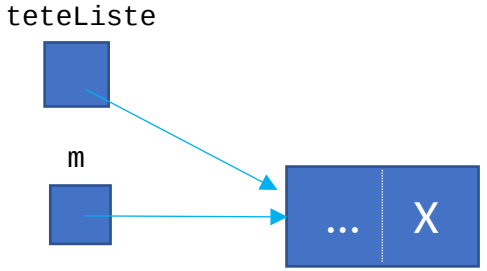
Initialisation du nouveau maillon :

```
m->donnees = …;
m->suiv = nullptr;
```



Ajout du premier maillon à la liste :

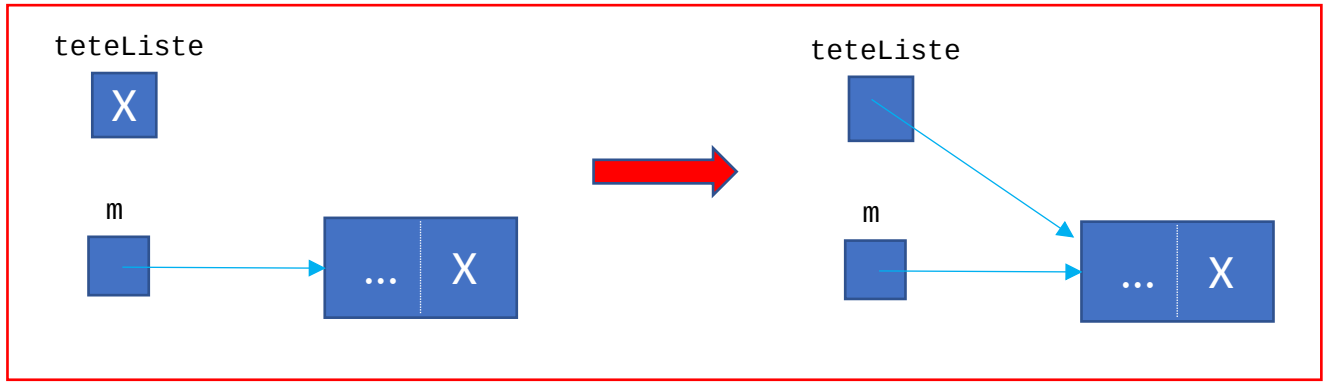
```
teteListe = m;
```

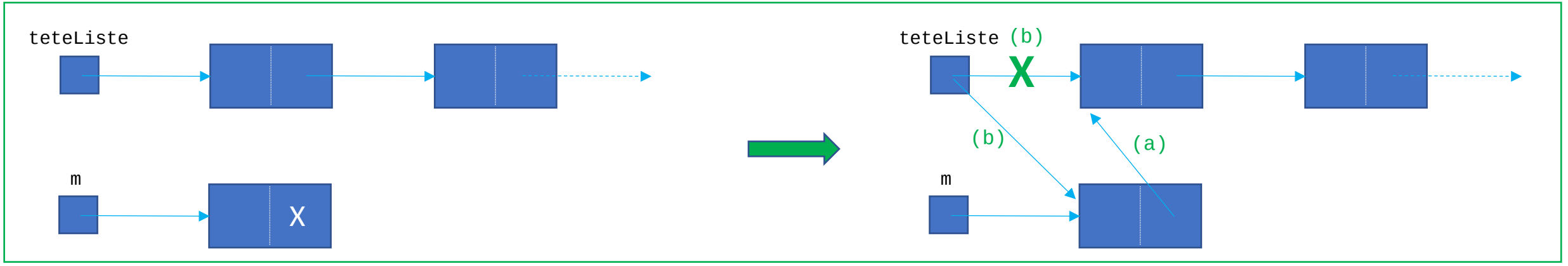


Matrice d'adjacence compacte (6)

- Rappel sur les listes chaînées
  - Insertion en tête de liste

```
if(teteListe == nullptr){ // la liste est vide
  teteListe = m;
} else { // la liste n'est pas vide
  m->suiv = teteListe; (a)
  teteListe = m; (b)
}
```

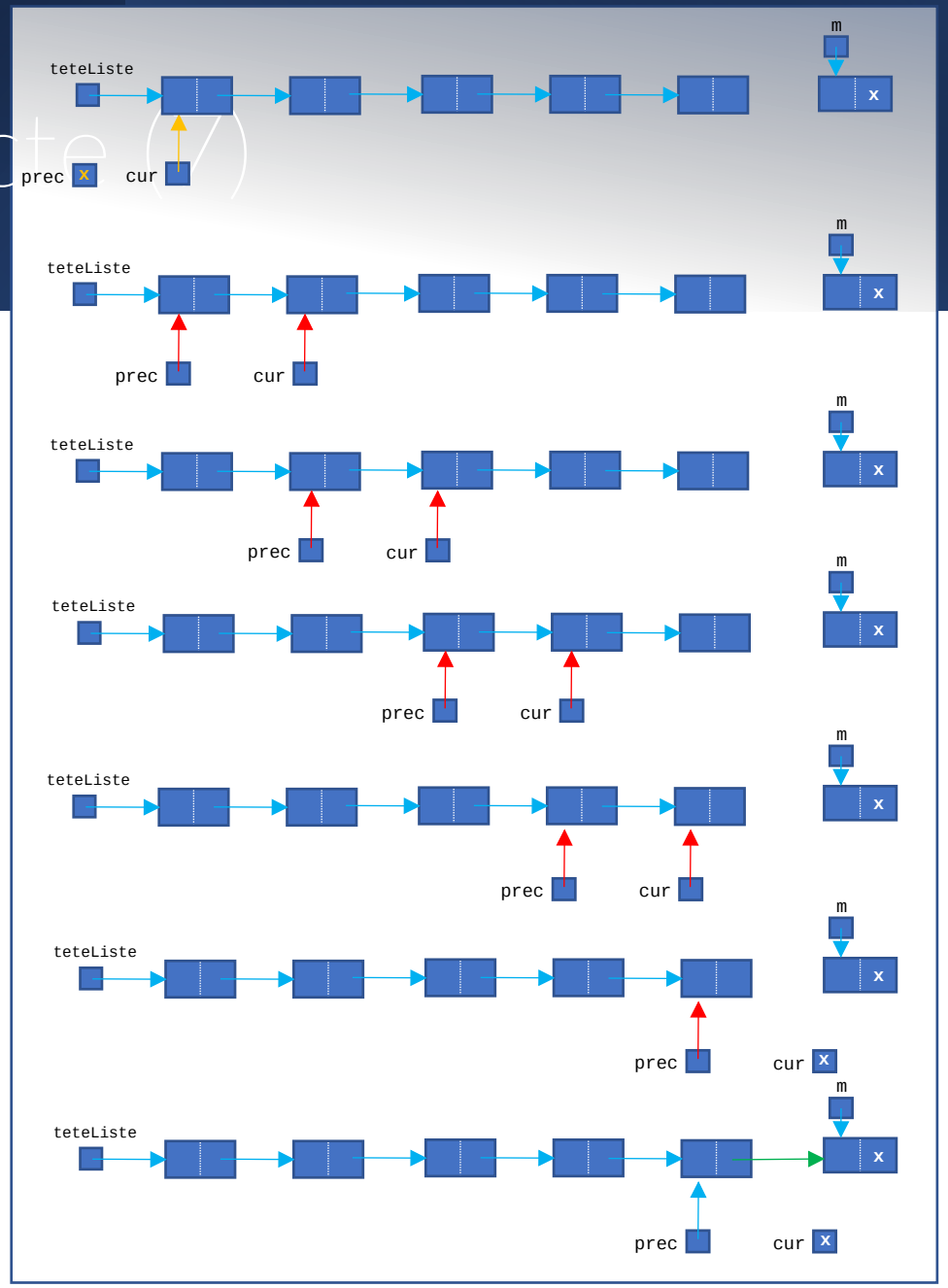




Matrice d'adjacence compacte (7)

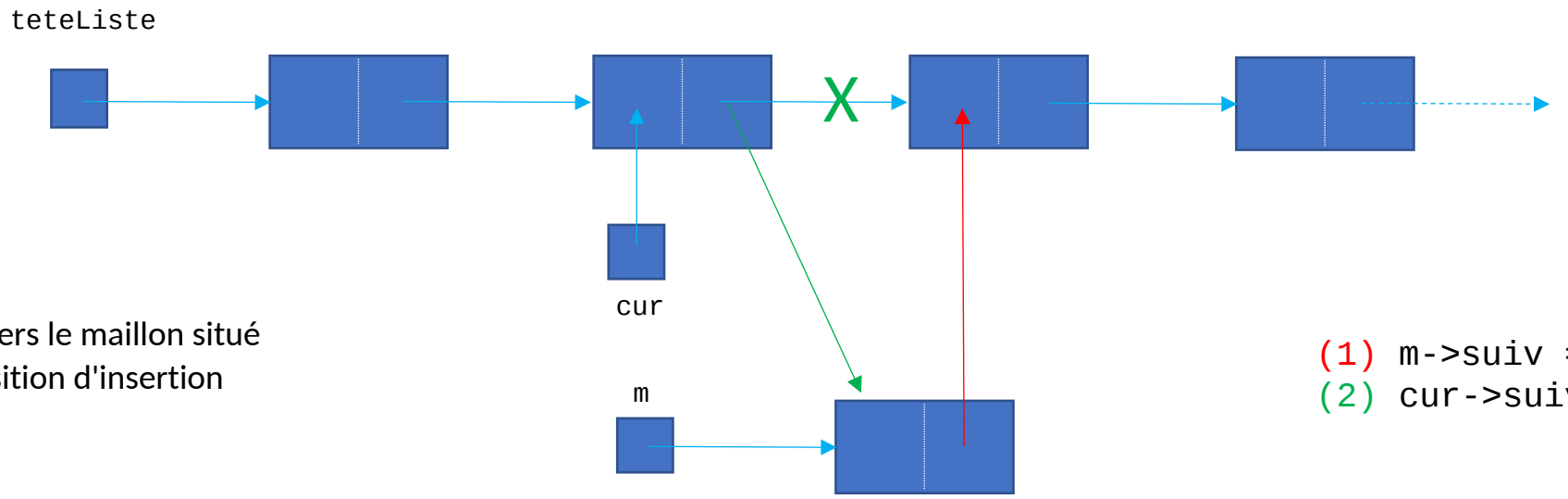
- Rappel sur les listes chaînées
  - Ajout en fin de liste

```
if(teteListe == nullptr){ // la liste est vide
  teteListe = m;
} else { // la liste n'est pas vide
  Maillon *cur = teteListe; //pointeur de parcours
  Maillon *prec = nullptr; // pointeur vers le maillon précédant cur
  while(cur!=nullptr){// il y a encore des maillons derrière cur
    prec = cur;
    cur = cur->suiv;
  }// while
  // cur pointe sur le dernier maillon
  prec->suiv = m;
}
```



Matrice d'adjacence compacte (8)

- Rappel sur les listes chaînées
  - Insérer entre deux maillons (1)

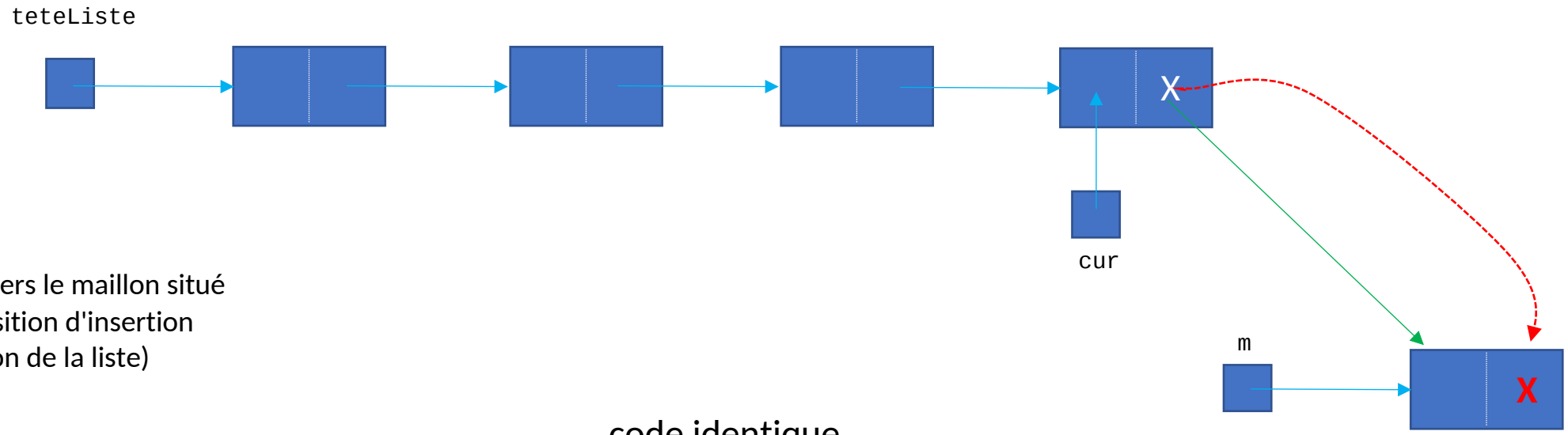


`cur` = pointeur vers le maillon situé juste avant la position d'insertion

```
(1) m->suiv = cur->suiv;
(2) cur->suiv = m;
```

Matrice d'adjacence compacte (9)

- Rappel sur les listes chaînées
  - Insérer entre deux maillons (2)



`cur` = pointeur vers le maillon situé juste avant la position d'insertion (ici dernier maillon de la liste)

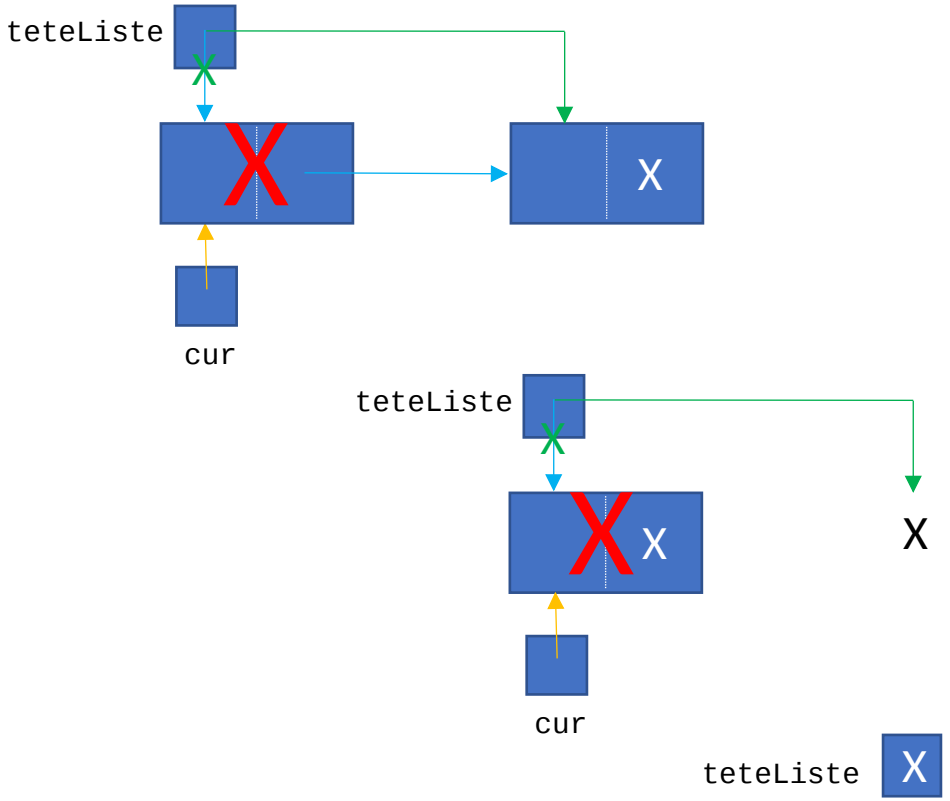
code identique

```
(1) m->suiv = cur->suiv;
(2) cur->suiv = m;
```

Matrice d'adjacence compacte (10)

- Rappel sur les listes chaînées
  - Effacer une liste

```
Maillon *cur;
...
while(teteListe != nullptr){
  cur = teteListe;
  teteListe = teteListe->suiv;
  delete cur;
}
```



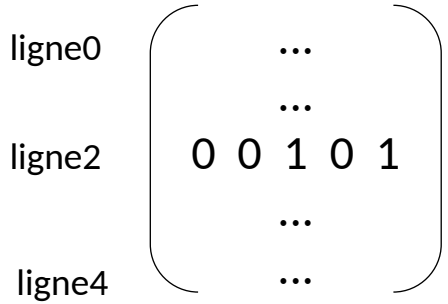
Matrice d'adjacence compacte (11)

- Structures de données pour la représentation compacte

Structure de données pour une liste chaînée

```
// maillon d'une liste chaînée

struct Maillon {
  int col;   // numéro de la colonne à laquelle correspond le coefficient
  int coef;    // coefficient de la matrice
  Maillon *suiv;  // élément suivant non nul sur la ligne
} ;
```



Matrice d'adjacence compacte (13)

- Structures de données pour la représentation compacte

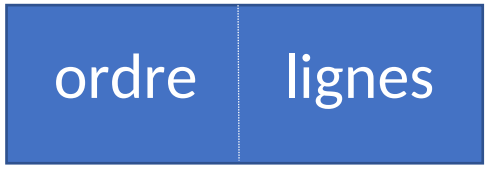
Structure de données pour une matrice d'adjacence

```
// représentation compacte d'une matrice d'adjacence

struct MatriceAdjacence {
  int ordre; // nombre de sommets du graphe
  Maillon** lignes; // tableau à allouer de taille "ordre",
                    // représentant les lignes de la matrice
};
```

Chaque case est une pointeur vers la liste chainée contenant les valeurs non nulles de la matrice.

Tableau
(1 case par ligne de la matrice)



MatriceAdjacence mat;

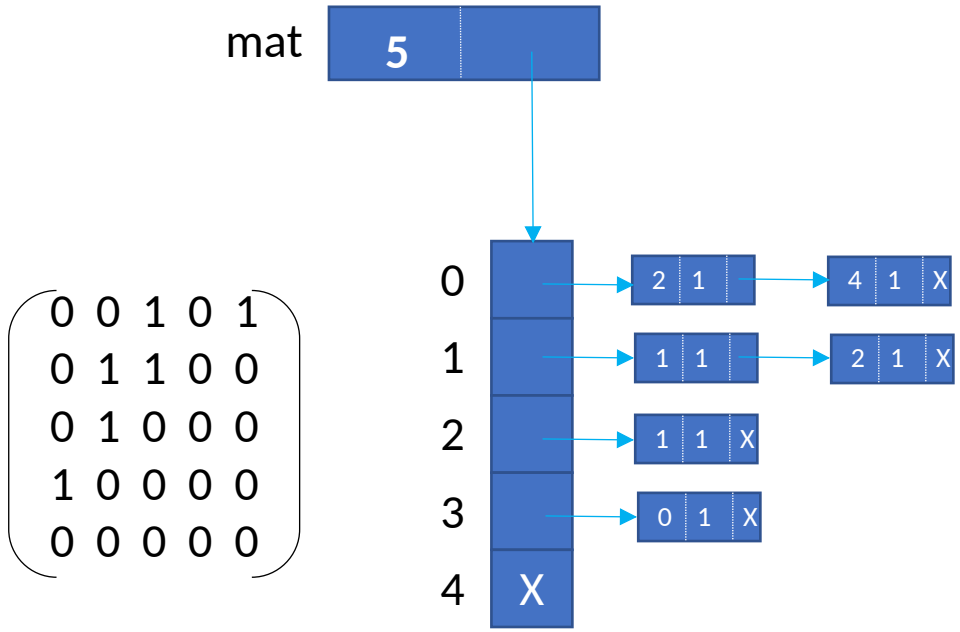
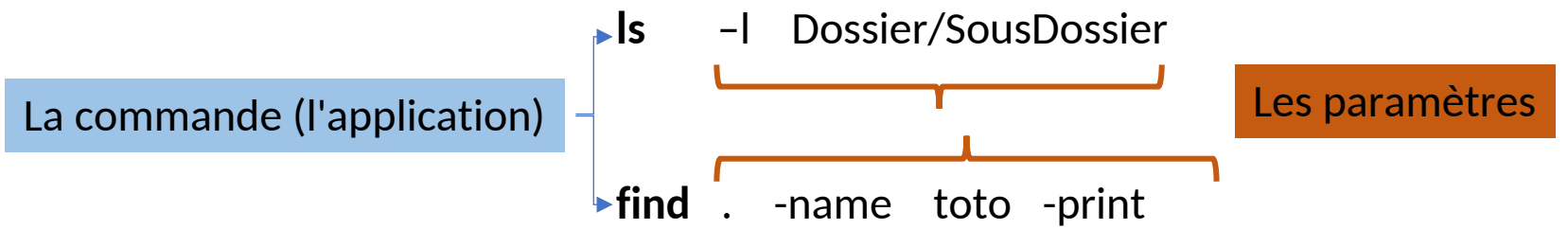


Tableau de têtes de liste chaînée
(1 par ligne)

Préparation aux TPs

Passage de paramètres dans le main (1)

- Objectif : fournir des données à l'application sur la ligne de commande
- Exemples :



argc = 3

argv = | "ls" | "–l" | "Dossier/SousDossier" |

argc = 5

argv = | "find" | "." | "-name" | "toto" | "-print" |

- Généralisable à toute application

```
int main(int argc, char *argv[])
```

Nombre de "mots" sur la ligne de commandes

Liste des "mots"

Passage de paramètres dans le main (2)

- Exemple :
  - Affichage de la liste des paramètres

exple.cpp

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){

  for(int i=0; i<argc; i++)
    cout << "->" << argv[i] << endl;

  return 1;
}
```

Compilation

```
g++  -std=c++11  exple .cpp  -o  exple

./exple  toto titi
-> ./exple
-> toto
-> titi
```

Exécution

- argc >=1
- argv[0] est le nom de la commande

Lecture dans un fichier texte (1)

- Outils
  - Utilisation de flots (stream) dédiés au fichiers
  - Définis dans la bibliothèque <fstream>
  - Type ifstream (input file stream)
  - Utilise le même opérateur que pour la lecture clavier (cin) : >>
- Étapes d'utilisation
  1. Ouvrir le fichier en lecture
  2. Vérification de l'ouverture
  3. Lectures
  4. Fermeture du fichier

Lecture dans un fichier texte (2)

void *flot* .open(nom_fichier,
mode d'ouverture)

Modes définis par :
- ios::out // ouverture en écriture
- ios::in // ouverture en lecture

- Refermer le fichier connecté au flot après utilisation
- Après fermeture, la variable flot peut être réutilisée pour un autre fichier

void *flot*.close()

```
#include <fstream>
...
ifstream fichier;// flot d'entrée

// ouverture du fichier en mode lecture
fichier.open ("test.txt", ios::in);

// test d'ouverture du fichier
if(fichier.is_open()==false){
    cout << "erreur d'ouverture "<< endl;
 return;
}

// lecture d'un entier dans le fichier
int vEntiere
fichier >> vEntiere;

//lecture d'un réel dans le fichier
float vReelle;
fichier >> vReelle;

// fermeture du fichier
fichier.close();
```

Toujours tester l'ouverture avant d'utiliser le flot la première fois

bool *flot*.is_open()

Lecture avec l'opérateur >>
(comme pour cin) :
- L'opérateur convertit les données lues dans le type attendu en partie droite

Exercices

- Relire et afficher le contenu d'un fichier contenant N entiers, la valeur N étant le premier entier présent dans le fichier.
- Compléter le programme pour que les entiers soient stockés dans une liste chaînée en ordre inverse de leur apparition dans le fichier.
- Compléter le programme pour afficher le contenu de la liste chaînée.
- Compléter le programme pour effacer le contenu de la liste chaînée.

```
5
67 45
89 12
4
```