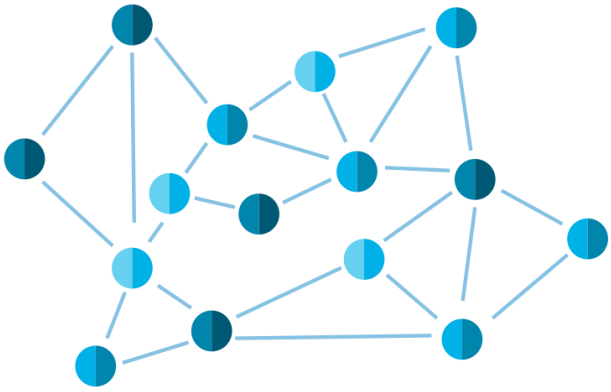
**Université Mohammed V de Rabat**
**École Normale Supérieure de Rabat**
**Département d'Informatique**

# Structures de données



**Réalisé par : Pr. Abdelali ELMOUNADI**

**Année universitaire 2023 - 2024**

# Table des matières

## Introduction

La programmation procédurale est un paradigme de programmation informatique où les fonctions et les procédures jouent un rôle prépondérant. En effet, le déroulement d'un programme écrit dans un langage procédural sera gouverné par l'ensemble des fonctions qui constituent celui-ci. En termes de traitement des données, à savoir leur stockage, leur manipulation ainsi que leur extraction, les compilateurs offrent généralement des types de données prédéfinis qui sont dédiés à cet effet. Ces types de données peuvent être primitifs ou complexes, selon le degré d'évolution du langage en question. Ce qui est sûr, c'est que ces derniers seront principalement destinés aux calculs numériques ou au traitement des chaines de caractères, voire à la simplification de certains traitements liés au système. Cependant, les programmeurs ont parfois besoin de représenter des notions de la vie courante. Se restreindre alors aux types prédéfinis ne sera pas d'une grande utilité dans un tel cas. Ainsi, les programmeurs se retrouvent forcés de faire appel à un mécanisme particulier qui est intimement lié à la programmation procédurale : celui des structures de données. Il existe trois niveaux de description pour une structure de données. Ces niveaux peuvent aller de l'aspect externe et général jusqu'à la spécification précise d'une réalisation particulière. La spécification fonctionnelle correspond à la présentation de la structure à l'aide d'exemples tirés du domaine informatique ou de la vie courante comme cité plus haut. L'étude de ces exemples permet de mettre en évidence les propriétés générales et les opérations caractéristiques de ladite structure. À ce niveau, les mécanismes internes de la matérialisation de la structure ne sont pas fournis. La spécification logique correspond à la description logique des données et des algorithmes de traitement. À ce niveau, les objets structurés sont décomposés en des objets plus élémentaires afin de pouvoir synthétiser les opérations portant sur la structure, celles-ci seront alors exprimées sous forme d'algorithmes. Nous obtenons ainsi une idée sur les mécanismes et la structure interne, mais la question de l'optimisation restera soulevée, car celle-ci dépend du caractère de la réalisation physique. Enfin, la spécification physique quant à elle correspond à un mode particulier d'implémentation au niveau de la mémoire et au codage des procédures de traitement. Le choix d'une représentation interne particulière sera généralement déterminant dans l'efficacité des programmes utilisant la structure de données.

# 1 Définition

## 1.1 Qu'est-ce qu'une structure de données ?

Linus Torvalds a dit: *"Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "*

D'un point de vue physique, une structure de données constitue une organisation au niveau de la mémoire qui permet de traiter, de stocker et d'extraire des données utiles à un programme. Il s'agit d'un ensemble organisé d'informations reliées logiquement qui peuvent être traitées collectivement ou de manière individuelle.

Le meilleur exemple d'une structure de données est le tableau monodimensionnel ou Vecteur, constitué de plusieurs éléments du même type. Il est possible d'effectuer des opérations sur chacun des éléments du vecteur pris individuellement, mais il existe aussi un moyen d'effectuer des opérations globales qui porterait sur l'ensemble du tableau en le considérant comme un seul et unique objet.

Une structure de données est une mise en œuvre concrète d'un type abstrait. En informatique, un type abstrait (en anglais, Abstract Data Type ou ADT) est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations pouvant être effectuées sur celle-ci. De manière générale, un ADT correspond à un cahier des charges qu'une structure de données doit ensuite respecter et mettre en œuvre.

Les structures de données sont essentielles pour la gestion efficace d'un grand nombre de données (Exemple : Historiquement, les informations contenues dans des fichiers, ou plus tard, celles provenant de bases de données), mais il est important d'utiliser des structures de données appropriées au contexte de leur utilisation. Une structure de données est caractérisée par l'arrangement de ses composantes, mais surtout par son mode d'utilisation. Ainsi, deux structures ayant les mêmes composantes et les mêmes arrangements (Exemple : les piles et les files d'attente) peuvent être malgré tout différentes vu que leur philosophie de traitement est fondamentalement différente. Le choix d'une structure est alors crucial, car une structure inadaptée pourrait impacter négativement la vitesse d'exécution. Quelques facteurs à prendre en compte lors du choix d'une structure de données incluraient dans ce cas le type d'informations à stocker ou à traiter, l'emplacement des données existantes, le mode d'accès aux données et la taille mémoire à réserver pour ces données.

## 1.2 Classification des structures de données

Il existe 2 grandes familles de structures de données :

- Les structures de données primitives (Types primitifs ou atomiques) ;
- Les structures de données non primitives (Types complexes).

Par ailleurs, les structures de données peuvent être organisées selon plusieurs axes :

- **Linéaire ou non linéaire** : Cette caractéristique décrit si les éléments de ladite structure respecte un ordre déterminé ou non (Exemple : Les tableaux sont considérés comme des structures linéaires).
- **Homogène ou non-homogène** : Cette caractéristique indique si tous les éléments de ladite structure sont du même type ou non.
- **Statique ou dynamique** : Cette caractéristique décrit la manière dont la mémoire dédiée aux éléments de ladite structure est allouée. Si la structure est statique, cela signifie que l'emplacement mémoire lui sera attribué au moment de la compilation et que ses éléments seront stockés sur la pile. Cependant, si la structure est dynamique, cela signifie que la mémoire lui sera allouée à la volée (pendant l'exécution) et que ces éléments seront stockés sur le tas.

La figure ci-dessous reprend la classification des structures de données selon l'axe de la linéarité :

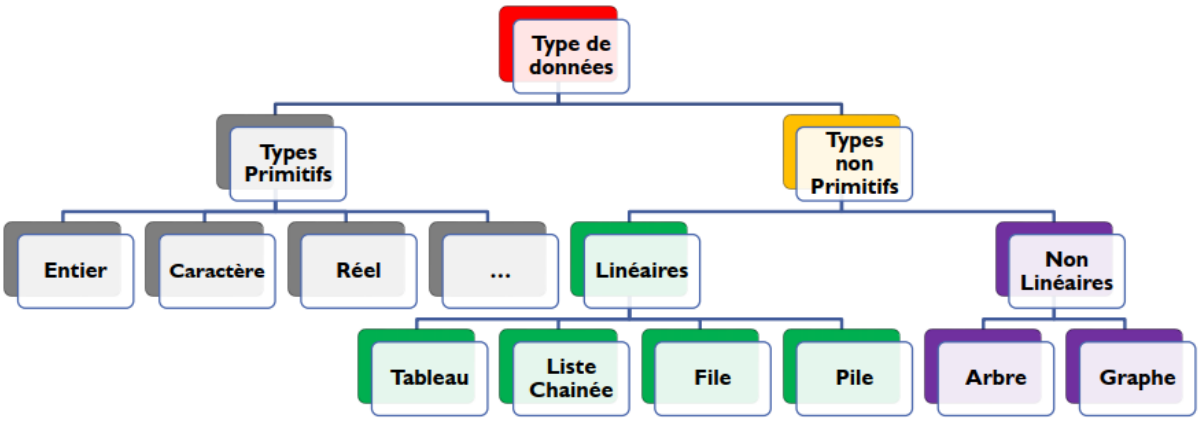



Fig. 1 Classification des structures de données suivant l'axe de la linéarité.

Enfin, les structures de données vont permettre au programmeur de s'acquitter des tâches relatives à la gestion des données d'un programme, de leur représentation et de leur organisation, mais il sera nécessaire d'utiliser un certain nombre de méthode d'accès (primitives d'accès) pour exploiter ces structures de données, d'où la fameuse formule : **Algorithms + Data Structures = Programs**.

## 2 Structures de données statiques homogènes

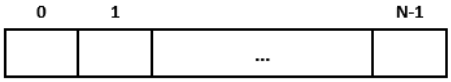
Une structure de données statiques homogène est un ensemble de données du même type reliées logiquement et dont la taille est fixe. Si tous les éléments de cette structure sont du même type, alors dans ce cas elle est dite homogène. Dans le cas contraire, elle est dite structure statique hétérogène.

## 2.1 Tableaux monodimensionnels

### 2.1.1 Définition

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contigües. Il s'agit de la représentation informatique de la notion de Vecteur en mathématiques.

Lorsqu'un tableau déclaré contient N éléments, ces derniers seront accessibles par un ensemble d'indices bornés entre 0 et N-1. Lorsqu'un tableau contient N éléments, il s'agit alors d'un tableau de taille N.




En langage C, La déclaration d'un tableau à une dimension se fait de la manière suivante :

**type nom-du-tableau[nombre-éléments];**

Où « nombre-éléments » est une expression constante entière positive. Par exemple, **int tab[10];** est une déclaration qui indique que tab est un tableau de 10 éléments de type int. Cette déclaration alloue donc en mémoire un espace de 10 x 4 octets consécutifs. Pour plus de clarté, il est recommandé de donner un nom à la constante nombre-éléments par la directive ***#define***. Par exemple :

***#define nombre-éléments 10***

Par ailleurs, il est possible d'initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

**type nom-du-tableau[N] = {valeur-1, valeur-2,...,valeur-N};**

Exemple :

```
#define N 4

int tab[N] = {1, 2, 3, 4};
```

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront initialisés à 0.

Toujours dans le cadre du langage C, un autre moyen peut être utilisé pour déclarer un tableau tout en ciblant les cases que nous souhaitons initialiser :

**type nom-du-tableau[N] = { [indice-1] = valeur-1, [indice-2] = valeur-2… };**

Mais il faut attention à ne pas dépasser l'indice N-1.

De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale.

Exemple :

```
char tab[10] = "exemple";

char tab[10] = {'e', 'x', 'e', 'm', 'p', 'l', 'e'};
```

Notons que le compilateur complète toute chaîne de caractères avec un caractère nul '\0'. Par conséquent, les exemples de code suivants auront pratiquement le même effet :

```
char tab[] = "exemple";

char tab[] = "exemple\0";

char tab[] = {'e', 'x', 'e', 'm', 'p', 'l', 'e', '\0'};
```

Lors de l'initialisation d'un tableau, il est possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation.

Exemple :

```
int tab[] = {1, 2, 3, 4};

char tab[] = "exemple";
```

### 2.1.2 Opérations sur les tableaux monodimensionnels

Il est possible d'effectuer un certain nombre d'opérations sur les tableaux, parmi ces opérations, nous retrouvons les opérations de tri.

L'opération de tri consiste à ordonner les éléments d'un tableau selon un (ou plusieurs) critère(s) afin de faciliter le traitement des données contenues dans le tableau en question, notamment la recherche et l'accès à ses éléments.

#### a) Le Tri par Bulles

Aussi appelé Tri par Propagation, ce tri consiste à parcourir le tableau en permutant toute paire d'éléments consécutifs (tab[i], tab[i+1]) non ordonnée. Après le premier parcours, le plus grand élément se retrouve dans la dernière case du tableau, puis la même procédure s'applique sur les éléments restants de manière itérative. Le nom "Tri par Bulles" est inspiré du mouvement des bulles qui ont tendance à se déplacer vers le haut dans un liquide. Notons aussi que le Tri par Bulles est une variante du Tri par Sélection que nous venons de voir plus haut.

Le tri par bulles a l'avantage d'être facile à comprendre, néanmoins, il reste inefficace dans le cas des tableaux de grandes dimensions, car il a une complexité temporelle moyenne de $O(n^2)$, où « n » est évidemment la taille du tableau.

- **Algorithme du Tri par Bulles :**

```
Début
Pour i de 0 à n-1
      Pour j de 0 à n-1-i
         Si tab[j] > tab[j+1] alors
            Permuter tab[j] et tab[j+1]
         Fin Si
      Fin Pour
   Fin Pour
```

### b) Le Tri par Insertion

Afin d'insérer tab[i], on utilise une variable intermédiaire pour conserver sa valeur qu'on compare successivement à chaque élément tab[i-1], tab[i-2], … qu'on déplace vers la droite tant que sa valeur est supérieure à celle de la variable temporaire. On affecte alors à l'emplacement laissé libre dans le tableau par ce décalage la valeur stockée temporairement.

- **Algorithme du Tri par Insertion :**

```
Début
 Pour i allant de 1 à n - 1 en incrémentant de 1 faire
          tmp := tab[i]
          j := i
          Tant que j > 0 et tab[j - 1] > tmp faire
                    tab[j] := tab[j - 1]
                    j := j - 1
          Fin tant que
           tab[j] := tmp
 Fin pour
Fin
```

Le tri par insertion a une complexité temporelle moyenne de $O(n^2)$, mais il est généralement plus efficace que le tri par bulles pour des tableaux de taille restreinte, en particulier lorsque le tableau est partiellement trié, vu qu'il effectue moins de comparaisons et d'échanges.

### c) Le Tri par Sélection

Également appelé Tri par Extraction, ce tri consiste à trouver dans le tableau l'indice de l'élément le plus petit, i.e. l'entier min tel que : tab[i] >= tab[min] pour tout indice i. Une fois cet indice localisé, les éléments tab[i] et tab[min] sont échangés. Cette opération est répétée jusqu'à la fin du tableau.

- **Algorithme du Tri par Sélection :**

```
Début
Pour i allant de 1 à n - 1
        min := i
        Pour j de i + 1 à N
                Si tab[j] < tab[min] alors
                        min := j
                Fin Si
        Fin Pour
        Permuter tab[j] et tab[min]
Fin Pour
Fin
```

Tout comme les deux algorithmes de tri précédent, le tri par sélection a une complexité temporelle moyenne de $O(n^2)$.

### d) Le Quick Sort

Le principe de base est de diviser pour mieux régner. On procède alors à une partition du tableau en 2 zones autour d'un pivot : les valeurs inférieures ou égales au pivot, et les valeurs supérieures ou égales au pivot. Le pivot est un élément choisi au hasard dans le tableau dont on affecte la valeur à une variable. Si on parvient à mettre les éléments plus petits au début du tableau et les éléments plus grands en queue du tableau, alors on pourra placer le pivot à la bonne place entre les 2 zones. L'opération est répétée récursivement sur chacune des zones créées jusqu'à ce que la zone soit réduite à un ensemble atomique d'un seul élément.

```
Procédure TriRapide(tab[] : tableau, début : entier, fin : entier)
   Si début < fin alors
      pivot := partitionner(tab, début, fin)
      TriRapide(tab, début, pivot - 1) % Tri du sous-tableau de gauche %
      TriRapide(tab, pivot + 1, fin)   % Tri du sous-tableau de droite %
Fin Procédure

Fonction Partitionner(tab, début, fin)
   pivot := tab[fin]   % Choisir le dernier élément comme pivot %
   i := début - 1      % Indice de l'élément plus petit %
   pour j de début à fin - 1
      si tab[j] <= pivot alors
         i := i + 1
         Permuter tab[i] et tab[j]
      fin si
   fin pour
   Permuter tab[i + 1] et tab[fin] % Placer le pivot à la bonne position %
   Retourner i + 1 % Retourner l'indice du pivot %
Fin Fonction
```

Le tri rapide ou « quick sort » a une complexité temporelle quasi-linéaire de O(n x log(n)) dans le cas moyen, mais peut atteindre une complexité de $O(n^2)$ dans les cas extrêmes, si par exemple, le pivot est choisi comme le plus petit ou le plus grand élément. Par ailleurs, en pratique, cet algorithme reste considéré comme le plus efficace et souvent plus rapide que d'autres algorithmes de tri pour de grandes quantités de données.

### e) Le Tri par Fusion

Il s'agit d'un autre exemple de tri qui applique le principe de diviser pour mieux régner. En effet, étant données deux collections d'éléments triées, de longueurs respectives m et n, il serait opportun d'obtenir une 3ème suite d'éléments triés de longueur m+n, par interclassement (ou fusion) des deux précédentes.

```
Procédure fusion(tab[] : tableau, tmp[] : tableau, début : entier, mil : entier, fin : entier)
        i := début
        j := mil + 1
        pour k allant de début à fin en incrémentant de 1 faire
                si j > fin ou (i <= mil et tab[i] < tab[j]) alors
                        tmp[k] := tab[i]
                        i := i + 1
                sinon
                        tmp[k] := tab[j]
                        j := j + 1
                fin si
        fin pour
        pour k allant de début à fin en incrémentant de 1 faire
                tab[k] := tmp[k]
        fin pour
Fin Procédure

Procédure TriFusion(tab[] : tableau d'entiers, tmp[] : tableau d'entiers, début : entier, fin : entier)
        si début < fin alors
                entier milieu := (début + fin)/2
                TriFusion(tab, tmp, début, milieu)
                TriFusion(tab, tmp, milieu + 1, fin)
                fusion(tab, tmp, début, milieu, fin)
        fin si
Fin Procédure

Appel de TriFusion(tab, tmp, 0, N-1) % tmp est une duplication de tab %
```

Le tri fusion a une complexité temporelle moyenne de O(n x log(n)), ce qui en fait un algorithme de tri très efficace, même dans les cas de grandes quantités de données.

## 2.2 Tableaux multidimensionnels

Il existe un cas particulier des tableaux multidimensionnels où ces tableaux sont à deux dimensions. Ce cas s'appelle les matrices. En Mathématiques, Une matrice constitue un tableau de nombres à *m* lignes et *n* colonnes, où *m* et *n* sont appelés *les dimensions de la matrice*. On dit que la matrice est de dimension *n* x *m*. une matrice est dite **carrée** lorsque *m* = *n*.

Par exemple :

$\begin{bmatrix} 1 & 3 & 7 \\ 8 & 5 & 6 \\ 2 & 9 & 4 \end{bmatrix}$ est une matrice de dimension *3 x 3*. Par la même occasion, il s'agit d'une matrice carrée.

*À* partir de cela, nous pouvons effectuer toutes les opérations admises sur les matrices (le calcul matriciel), à savoir l'addition ou la soustraction, la multiplication, le calcul du déterminant (pour les matrices carrées), etc.

Par ailleurs, un tableau à 2 dimensions sera alors interprété comme un tableau unidimensionnel de taille *L* dont chaque composante est un tableau unidimensionnel de taille *C*. ici, *L* représente le nombre de lignes du tableau alors que *C* représente le nombre de colonnes du tableau. Un tableau tel que celui-ci va alors contenir L x *C* éléments. En langage *C*, un tableau à 2 dimensions est déclaré de la manière suivante :

**type nom-du-tableau[nombre-lignes][nombre-colonnes];**

Où *nombre-lignes* et *nombre-colonnes* sont 2 expressions constantes entières positives. Par exemple, la déclaration int tab[10][10]; indique que tab est un tableau de 10 x 10 (soit 100) éléments de type int. Cette déclaration alloue donc en mémoire un espace de 10 x 10 x 4 octets consécutifs.

Exemple de l'initialisation d'un tableau à 2 dimensions :

**int tab[2][3] = { {1, 2, 3}, {4, 6, 8} };**

*À* noter que si le compilateur peut déduire la taille d'un tableau à 1 dimension à partir des données d'initialisation, il ne peut cependant pas déduire le nombre de colonnes dans un tableau à 2 dimensions. *À* cet effet, la déclaration suivante :

**int tab[][] = { {1, 2, 3}, {4, 6, 8}, {3, 6, 9} };**

doit être remplacée par :

**int tab[][3] = { {1, 2, 3}, {4, 6, 8}, {3, 6, 9} };**

afin d'éviter une erreur de compilation.

# 3 Structures de données dynamiques

Dans la pratique, très souvent, on veut représenter des objets soit dont on ne connaît pas à priori la taille, soit dont la taille est variable selon les cas ou au cours du traitement. On est alors amené à utiliser des structures qui peuvent évoluer, pour bien s'adapter à ces objets. Ce sont les structures de données dynamiques. Exemple : On doit lire sur un fichier d'entrée une suite de nombres sur lesquels on effectuera un traitement ultérieur. On ne connaît pas la quantité des nombres à lire et il n'est pas question de les compter avant. Quelles sont les structures de données possibles pour représenter l'ensemble de ces nombres ?

Le premier choix possible est de prendre un tableau "surdimensionné" (la taille d'un tableau est limitée par la taille d'un segment) pouvant accueillir l'ensemble des données au risque évident qu'il soit trop petit, ou vraiment trop grand, ce qui induit une perte énorme de places mémoires.

Une solution meilleure consiste à utiliser une structure dynamique qui s'agrandit au fur et à mesure de la lecture des nombres. Une structure dynamique consiste à allouer de l'espace mémoire dans le Tas au fur et à mesure de l'exécution du programme, ce qui induit une optimisation de l'espace mémoire alloué à la structure.

Contrairement aux structures de données statiques, dont la représentation en mémoire est contiguë, la représentation physique des structures dynamiques est dispersée en mémoire. Donc comment créer des liens entre les différentes informations ? L'espace réservé à chaque élément d'une structure dynamique est alloué dans le Tas, et par conséquent on est amené à utiliser des pointeurs pour accéder aux différents éléments.

## 3.1 Les listes chainées

Une liste chaînée est une suite de données liées entre elles. Le lien est réalisé par des pointeurs.

Chaque élément de la liste est un enregistrement dans l'un des champs est un pointeur qui pointe vers l'élément suivant.

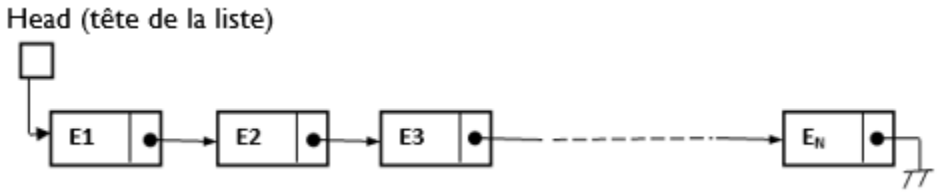



Fig. 2 Représentation de la notion de liste chainée.

Une liste chaînée est caractérisée par la donnée d'un pointeur qui pointe sur le premier élément de la liste. Elle peut être déclarée de la manière suivante :

```
Elément : ENREGISTREMENT
        Valeur % Ce champ pourra éventuellement être de n'importe quel type %
        Suivant : ^Elément
Fin ENREGISTREMENT

Liste : ENREGISTREMENT
        Tête : ^Elément
        Taille : Entier % Champ facultatif %
Fin ENREGISTREMENT
```

Les opérations possibles sur une liste chainée sont les suivantes :

- **L'initialisation d'une liste.**
- **La vérification si une liste est vide.**
- **Le calcul de la taille d'une liste.**
- **L'ajout d'un élément dans la liste :** L'opération de l'ajout peut être réalisée de différentes manières :
  - **L'ajout d'un élément en début de liste.**

```
Fonction insertion_debut(L : liste, nouveau : ^élément)
        nouveau.suivant := L.tête
        L.tête := nouveau
        L.taille := L.taille + 1
Fin Fonction
```

  - **L'ajout d'un élément en fin de liste.**

```
Fonction insertion_fin(L : liste, nouveau : ^element)
        Si L.tête = NIL alors
                L.tête := nouveau
        Sinon
                t := L.tête
                Tant que T.suivant <> nil faire
                        t := T.suivant
                fin tant que
                t.suivant := nouveau
        Fin Si
        nouveau.suivant = NIL
        L.taille := L.taille + 1
Fin Fonction
```

- **L'ajout d'un élément au milieu de la liste (ou plus précisément à un emplacement donné de la liste).**

```
Fonction insertion_A(L : liste, nouveau : ^element, position : entier)
        Si position < 1 ou position > L.taille alors
                écrire ''position incorrecte''
        Sinon
                t := L.tete
                pour i allant de 1 à position - 2 faire
                        t := t.suivant
                fin pour
                nouveau.suivant := t.suivant
                t.suivant := nouveau
        Fin Si
        L.taille := L.taille + 1
Fin Fonction
```

- **La suppression d'un élément de la liste :** Tout comme l'opération d'ajout, l'opération de suppression peut se faire de différentes manières :

  - **La suppression d'un élément à partir du début de la liste.**

```
Fonction supprimer_debut(L : liste)
        Si L.taille = 0 alors
                écrire ''la liste est vide''
        Sinon
                T := L.tête
                L.tête := L.tête.suivant
                L.taille := L.taille - 1
        Fin Si
Fin Fonction
```

- **La suppression d'un élément à partir de la fin de la liste.**

```
Fonction supprimer_fin(L : liste)
        Si L.taille <= 1 alors
                supprimer_debut(L) %faire appel à la fonction définie plus haut%
        Sinon
                T := L.tête
                tant que t.suivant.suivant <> nil faire
                        t := t.suivant
                Fin Tq
                t.suivant = nil
                L.taille := L.taille - 1

        Fin Si
Fin Fonction
```

- **La suppression d'un élément à partir du milieu de la liste (ou plus précisément à partir d'un emplacement donné de la liste).**

```
Fonction supprimer_A(L : liste, position : entier)
        Si position < 1 ou position >L.taille alors
                écrire ''position incorrecte''
        Sinon
                t := L.tête
                Pour i allant de 1 à position - 2 faire
                        t := t.suivant
                Fin Pour
                t.suivant := t.suivant.suivant
                L.taille := L.taille - 1
        Fin Si
Fin Fonction
```

- **La mise à jour d'un élément dans la liste** : Il faudra alors accéder à l'élément souhaité avant d'apporter une modification sur ses champs.

## 3.2 Les files d'attente

Une file d'attente est un cas particulier des listes chainées. En effet, il s'agit d'une structure de données qui respecte le principe FIFO (First In First Out). Du coup, les opérations que l'on peut avoir (en plus des fonctions utilitaires du calcul de la taille etc.) sur une file d'attente vont se résumer dans :

- **L'ajout d'un élément en fin de liste (enqueue).**
- **La suppression d'un élément en tête de liste (dequeue).**

## 3.3 Les piles

Tout comme les files d'attente, les Piles sont également des cas particuliers des listes chainées. En revanche, contrairement aux files d'attente, les piles vont plutôt respecter le principe LIFO (Last In First Out). Par conséquent, les opérations qui seront autorisées sur ce genre de structure de données sont les suivantes (en plus des fonctions utilitaires) :

- **L'ajout d'un élément en tête de liste (push).**
- **La suppression d'un élément en tête de liste (pop).**

## 3.4 Les graphes

La théorie des graphes est une branche mathématique qui stipule que l'on peut formuler des problèmes mathématiques sous forme de graphe. Elle s'intéresse plutôt à des problèmes d'optimisation, mais les graphes sont désormais utilisés dans une vaste panoplie de domaines (télécommunication, réseau sociaux, modélisation informatique, etc.).

En mathématiques, un graphe est un ensemble de nœuds reliés par des arcs.

**G = (V, E)**
***Avec V : l'ensemble des sommets***
***Et E l'ensemble des arcs***

L'ordre d'un graphe est le nombre de ses sommets : $\boldsymbol{n = \sum S_i \; avec \; S_i \in V}$

La relation entre deux sommets liés par un arc constitue une relation d'adjacence. D'une manière plus formelle, un sommet $S_i$ est adjacent au sommet $S_j$ s'il existe une arête (un arc) entre $S_i$ est $S_j$.

L'ensemble des sommets adjacents au sommet Si est défini par :

$$\boldsymbol{adj(S_i) = \{S_j \; avec \; (S_i, S_j) \in E \; ou \; (S_j, S_i) \in E\}}$$

En informatique, Un graphe est une structure de données non linéaire qui se constitue des deux éléments suivants :

- Un ensemble fini de sommets également appelés **nœuds**.
- Un ensemble fini de paires ordonnées de la forme (u, v) appelées **arêtes** (ou **arcs**). La paire de la forme (u, v) indique qu'il existe une arête entre le nœud u et le nœud v. Les arêtes peuvent dans certains cas avoir un poids lorsqu'il s'agit d'un graphe pondéré.

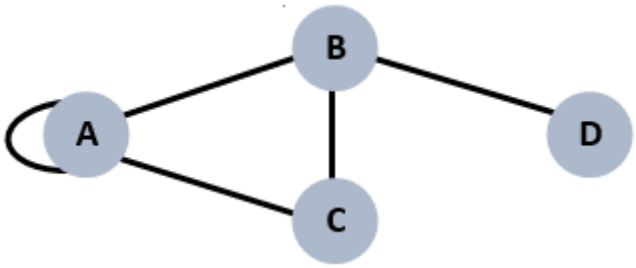



Fig. 3. Exemple d'un graphe non orienté.

Lorsque nous voulons représenter un graphe au niveau de la mémoire, il existe plusieurs méthodes pour le faire. Dans ce cours, nous retenons les deux méthodes suivantes :

- Représentation par matrice d'adjacence.
- Représentation par listes d'adjacence.

**Représentation par matrice d'adjacence :**

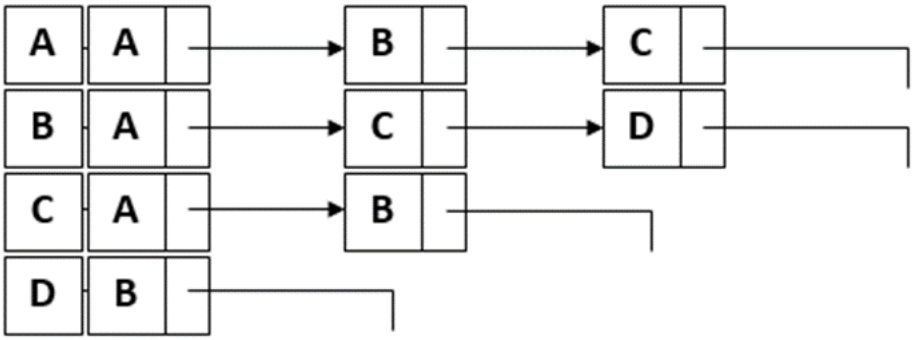
La matrice d'adjacence est un tableau 2D de taille V x V où V est le nombre de sommets dans un graphe. Soit le tableau 2D adj[][], adj[i][j] = 1 indique qu'il existe une arête entre le sommet i et le sommet j. La matrice d'adjacence pour un graphe non dirigé est toujours symétrique. La matrice d'adjacence est également utilisée pour représenter les graphes pondérés. Si adj[i][j] = w, alors il existe une arête entre le sommet i et le sommet j avec un poids w.

La matrice d'adjacence pour le graphe de l'exemple ci-dessus est la suivante :

| | A | B | C | D |
|---|---|---|---|---|
| A | 1 | 1 | 1 | 0 |
| B | 1 | 0 | 1 | 1 |
| C | 1 | 1 | 0 | 0 |
| D | 0 | 1 | 0 | 0 |

**Représentation par listes d'adjacence :**

Dans cette représentation, un tableau de listes est utilisé. La taille du tableau est égale au nombre de sommets. Soit un tableau de listes tab[]. L'entrée tab[i] va alors représenter la liste des sommets adjacents au $i^{ème}$ sommet. Ci-après la représentation des listes d'adjacence du graphe ci-dessus :




## 3.5 Les arbres

Les arbres sont des structures de données non linéaires. Ils constituent un cas particulier de graphes. Aussi, ils ont une nature hiérarchique et acyclique, et sont traités de manière récursive. Chaque nœud appartenant à un niveau pointe sur un ensemble de nœud d'un niveau inférieur, la relation qui relie ces nœuds est alors une relation ***père - fils.***

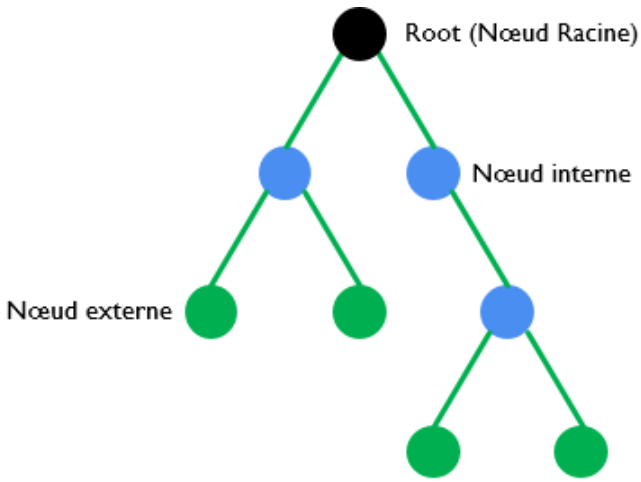



**Fig. 4 Représentation de la notion d'arbre.**

Les éléments qui constituent un arbre sont considérés comme des nœuds ; on parle de nœud interne lorsque celui-ci dispose d'un ou de plusieurs nœuds fils. En revanche, on parle de nœud externe ou de « feuille » lorsque celui-ci se trouve à l'extrémité de l'arborescence et qu'il ne dispose pas de nœuds fils. Chaque arbre est constitué d'au moins un nœud, ce nœud est alors considéré comme un nœud ***racine***. Étant donné que chaque nœud dispose généralement d'un nœud parent, le nœud racine quant à lui, échappe à cette règle.

Il existe un cas particulier des arbres : les arbres binaires. Ces derniers ont la spécificité d'avoir des nœuds qui dispose *d'au plus 2 nœuds fils (ou 2 sous arbres)*. Il s'agit d'un bon exemple qui facilite la compréhension des notions par rapport aux arbres en relation avec les structures de données. C'est celui qui est traité dans le cadre de ce cours.

```
Nœud : ENREGISTREMENT
        Valeur % Ce champ pourra éventuellement être de n'importe quel type %
        Droit : ^nœud % Représente le nœud fils droit %
        Gauche : ^nœud % Représente le nœud fils gauche %

Fin ENREGISTREMENT

Arbre : ENREGISTREMENT
        Racine : ^nœud
        Taille : Entier % Champ facultatif %
Fin ENREGISTREMENT
```

**Propriétés d'un Arbre :**

- Un arbre est caractérisé avant tout par une taille. La taille d'un arbre représente le nombre de nœuds contenus dans celui-ci.
- La profondeur d'un nœud x dans un arbre est sa distance par rapport à la racine. La profondeur de la racine est donc 0.
- La hauteur d'un arbre est représentée par la valeur maximale des profondeurs de ses nœuds par rapport à la racine (par convention nous ajoutons un + 1).

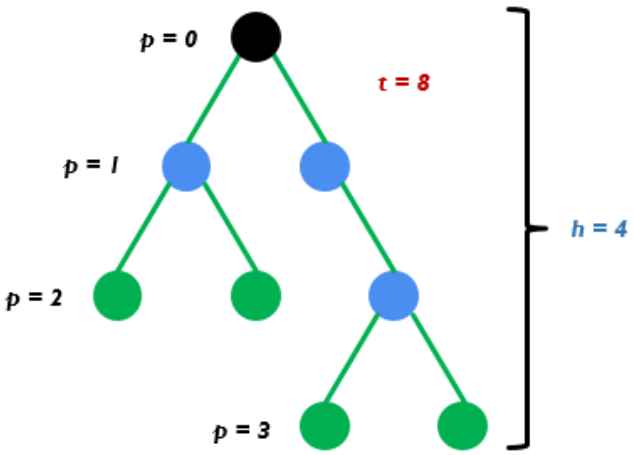


Fig. 5 Exemple d'un arbre de taille 8 et de hauteur 4.

- Un arbre peut être parcouru de deux manières différentes : en utilisant un algorithme de parcours en Profondeur **(Depth-First Search)**, ou bien de parcours en Largeur (Breadth-First Search). À noter que ces deux algorithmes restent appliqués aux graphes de manière générale.