# Cours 1

# 1. Présentation

## *1.1. But du cours.*

Techniques de programmation

Indépendantes du langage
Améliorant les temps de calcul
Améliorant la conception et la maintenance

## *1.2. Organisation* **(10 cours + 10 TD*)***

Cours
Notions théoriques
TD
Beaucoup de programmes en application immédiate du cours
Faire la conception avant le codage
Choisir un bon environnement pour se concentrer sur la structure des programmes
(.xemacs adapté à java fourni)
Réduire les temps de maintenance et de correction des erreurs.
Réutiliser

## *1.3. Plan*

Tableaux et listes
Piles et files
Arbres
Temps de calculs
Dictionnaires (Hash-codes et arbres binaires)
Notions sur les graphes

## *1.4. Contrôle*

présence et activité en TD
1 contrôle court
1 contrôle long

# 2. Concevoir un programme

## 2.1. Arriver à écrire un bon programme

Processus en boucle :

1. Quelles sont les données sur lesquelles le programme travaille
2. Quels sont les services qu'il doit offrir (service = répondre à une requête d'un utilisateur / d'un autre programme)
3. Quels sont les traitements qu'il doit effectuer pour offrir ces services
4. Comment faut-il organiser et représenter les données pour que les traitements soient possibles (structure abstraite)
5. Comment les traitements s'appellent-ils les uns – les autres / sont-ils appelés par l'interface de requête
6. Comment peut-on construire au mieux (de la façon la plus efficace en temps / en espace) les différents traitements.
7. Implémenter les structures de données dans le langage choisi
8. Implémenter les traitements.

## 2.2. Les critères du bon programme

Efficacité en temps :

Efficacité en espace :

Réutisabilité :

.

→ Pas d' amélioration possible qui soit "raisonnable" (en temps de travail, support matériel,..) par rapport au gain.

## 2.3. Qu'est-ce que l'algorithmique

Il y a des solutions bien connues à 4 (organiser les données) et 6 (construire les traitements)
→ ne pas réinventer la roue, pouvoir adapter ce qui existe.

Pour l'essentiel, les solutions ne dépendent pas du langage
→ Vision algorithmique : décrire l'essentiel, éliminer les parasites.
Structure de donnée abstraite = ensemble de données muni de ses opérations d'accès et de modification.
On ne s'intéresse qu'aux structures de données abstraites (SDA) et à leur manipulation.

En même temps, on sait traduire les SDA dans n'importe quel langage, souvent de plusieurs façons

Quand on a réglé le problème algorithmique, on peut implémenter les structures de données abstraites et les algorithmes dans le langage dont on dispose. (implémenter = donner une traduction dans ce langage)

**Avantages :**

- **Conception de programme plus facile : on sépare en sous problèmes plus simples.**
- **Discussion et déboguage plus facile : on peut analyser et vérifier les algorithmes sans avoir besoin de lire le code – ce que personne ne ferait. Ensuite, on programme à partir d'une structure saine et on ne débogue que l'implémentation.**
- **Portage sur un autre langage, une autre machine, beaucoup plus facile : les structures de données abstraites et les algorithmes ne changent pas – seuls les détails d'implémentation sont à refaire.**

# 3. Structures de base : tableaux, listes, piles et files

Convention : les SDA et leurs opérations sont décrites en pseudo-C ou pseudo-java suivant ce qui est plus commode, sans suivre exactement la syntaxe. En pseudo-C, on note **opération**(arg) quand une opération ne peut pas modifier son argument (il est le même après l'opération qu'avant, quoi qu'on fasse), et opération(@arg) quand elle peut le modifier. Ex :
void ecrAj3(x) ; // ajoute 3 à x et écrit le résultat
void ecrAj3'(@x) ; // ajoute 3 à x et écrit le résultat
x=4 ; ecrAj3(x) ; // x vaut toujours 4
x=4 ; ecrAj3'(x) ; // x vaut 7. On peut aussi écrire ecrAj3'(@x) pour bien insister

## *3.1. Les tableaux*

### 3.1.1. Définition

**Spécification : ensemble de données repérées par un indice**. Les opérations autorisées sont :
donnée **lire**(tableau *tab*, indice *ind*) // lire la donnée d'indice *ind* ; lire à un indice où on n'a pas au préalable écrit est une erreur.
void **écrire**(tableau @*tab*, indice *ind*, donnée *don*) // modifier la donnée d'indice *ind*. en la remplaçant par *don*.

On écrit aussi `don=tab[ind]` pour la première, `tab[ind]=don` pour la seconde. Quelle que soit l'écriture, on n'accède aux données que par leur indice, et on les retrouve à l'indice où on les a placées.

#### 3.1.2. Ce qui change selon les langages

Tous les langages fournissent d'origine des tableaux. C et java n'acceptent que des tableaux élémentaires : la capacité est fixe, décidée à la création, et les indices sont des entiers consécutifs commençant à 0. En PHP, perl, python, les tableaux acceptent n'importe quel indice sans déclaration préalable, et les indices peuvent être des chaînes de caractères.
Ces tableaux (php, etc.) ne sont pas *implémentés* comme des tableaux C ou java. D'autre part, les vecteurs en java ont une taille variable, mais des indices entiers et des opérations qui n'existent pas dans les tableaux.

### *3.2. Les piles*

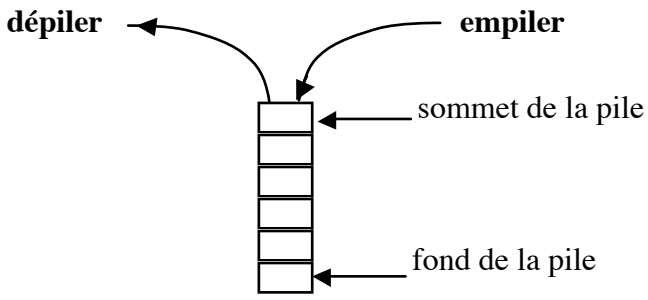
#### 3.2.1. Définition

- Structure de données dynamique : la quantité de données change pendant l'exécution.

**Spécification : ensemble dynamique de données sans adresses internes (on ne peut pas choisir à quoi on accède). On retire les données dans l'ordre inverse de celui dans lequel on les a déposées** (dernier entré, premier sorti ; last in, first out ; lifo)

Opérations autorisées :
void **empiler**(pile @*pi*, donnée *don*) // on peut toujours empiler sans limite
donnée **dépiler**(pile @*pi*) //dépiler une pile vide est une erreur
booléen **estVide**(pile *pi*)

Image :



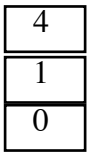

#### 3.2.2. À quoi ça sert ?

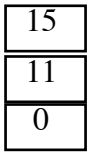
A revenir à l'état antérieur.

**Ex 1: faire la liste des parenthèses qui se correspondent dans un texte (un calcul).**

```
( ( x + ( 3 - y ) ) - ( 2 z . ( x + y ) ) )
0 1       4       8 9   11        15        19 20 21
```

Chaque fois qu'on ouvre une parenthèse, on empile sa position. Chaque fois qu'on ferme une parenthèse, on dépile pour trouver la position de son ouvrante.




Résultat : (4;8), (1;9), (15;19), (11;20), (0;21)

**Ex 2 : traiter un texte codé en pseudo html (version simplifiée : on ne traite que gras et italique)**

On a le html codé avec **<i><\i>** (italique), **<b><\b>** (gras), **<n><\n>** (normal).
A chaque changement de police, il faut envoyer à l'imprimante la police à utiliser (codage imprimante : [[\police=norm]], [[\police=gras]], [[\police=ital]], [[\police=grasItal]].

Texte html :

```
Ce site contient <i> les corrig&eacute;s des <b> prochains
contr&ograve;les d'<n>Algo Avanc&eacute;e<\n> jusqu'en
mars,<\b> et des liens<\i> vers des <b> sites de vacances<\b>
int&eacute;ressants.
```

Il faut imprimer :
Ce site contient *les corrigés des* ***prochains contrôles d***'Algo Avancé***e jusqu'en mars,*** *et des liens* vers des **sites de vacances** intéressants.

Algorithme : on a une variable *police_actuelle*. Quand on a un *début de propriété* (<i>, <b> ou <n>), on calcule la nouvelle police de caractères par modification de l'ancienne, et **on empile l'ancienne**. Quand on a une *fin de propriété*, on **dépile pour revenir à la police précédente**.

Le calcul de la police actuelle avec les états successifs de la pile :

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Pile | | | | | | | | | |
| | | | | grasIta | | | | | |
| | | | italiq. | italiq. | italiq. | | | | |
| | | normal | normal | normal | normal | normal | | normal | |
| police actuelle | normal | italiq. | grasIta | normal | grasIta | italiq. | normal | gras | normal |
| transition | <i> | <b> | <n> | <\n> | <\b> | <\i> | <b> | <\b> | |

le texte envoyé à l'imprimante :

```
[[\police=norm]] Ce site contient [[\police=ital]] les corrigés des
[[\police=grasItal]] prochains contrôles d'[[\police=norm]]Algo Avancée
[[\police=grasItal]] jusqu'en mars, [[\police=ital]] et des liens
[[\police=norm]] vers des [[\police=gras]] sites de vacances[[\police=norm]]
intéressants.
```

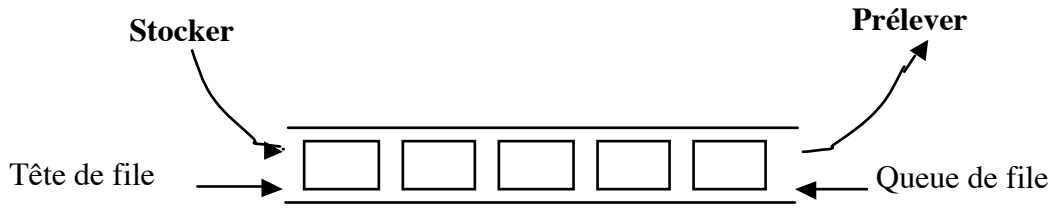
## *3.3. Les files*

### 3.3.1. Définition

**Spécification : ensemble dynamique de données sans adresses internes (on ne peut pas choisir à quoi on accède). On retire les données dans l'ordre dans lequel on les a déposées** (premier entré, premier sorti ; first in, first out ; fifo)

Opérations autorisées :
void **stocker**(file *@fi*, donnée *don*) // on peut toujours empiler sans limite
donnée **prélever**(file *@fi*) //dépiler une pile vide est une erreur
booléen **estVide**(file *fi*)

Image :




### 3.3.2. A quoi ça sert ?

à traiter les données dans l'ordre où on les a reçues, quelle que soit la complexité du traitement de chacune

Ex : traitement des requêtes reçues par un serveur de BD (un serveur Webb – un moteur de recherche – etc) :

- un module de communication lit les requêtes sur le réseau, les stocke dans une file, libère la connexion et se remet à écouter
- un module de traitement prélève les requêtes et les traite (ou les répartit sur des sous-serveurs), puis stocke la réponse.
- un module de communication prélève les réponses et les envoie

## *3.4. Les listes*

### 3.4.1. Définition

**Spécification: une liste stocke un ensemble de données et permet de parcourir cet ensemble du début à la fin dans l'ordre.**

Ont peut expliquer les opérations avec un curseur qui parcourt la liste (quand le curseur change, la liste est modifiée).

Les opérations autorisées sont :

void **rembobiner**(liste @*li*) //place le curseur devant la liste
donnée **voirSuivant**(liste *li*) /* renvoie l'élément qui suit celui sous le curseur (le premier s'il était devant la liste). Quand il n'y a pas de suivant, renvoie l'élément spécial **nil** (ou **null**, ou **NULL**…) */
void **avancer**(liste @*li*) /* place le curseur sur l'élément renvoyé par **voirSuivant** */
void **ajouter**(liste @*li*, donnée *don*) /* insère *don* en position de suivant par rapport au curseur (en tête si le curseur est devant la liste) */
void **supprimer**(liste @*li*) /* Supprime la donnée en position de suivant par rapport au curseur. Supprimer quand le curseur est en fin de liste est une erreur */
booléen **estEnFin**(*liste li*) /* Vrai quand il n'y a pas de suivant */

#### 3.4.2. à quoi ça sert

Stocker des ensembles de données dont le volume change souvent, stocker des données dans l'ordre en ayant la possibilité de mettre à jour quand il y a des changements.

Exemples d'applications:

- **faire l'appel** (10 fois par semaine) : au début, il faut créer une liste alphabétique par groupe. Ensuite, il faut tenir compte des changements de groupes, abandons, nouveaux arrivants…

- **écriture et modification de fichiers sur un disque** : on ne peut pas garantir qu'on trouvera l'espace nécessaire d'un seul tenant, ni qu'on pourra par la suite donner de l'espace supplémentaire à un fichier à la suite de celui qu'il occupe déjà

### *3.5. Les listes à double sens*

#### 3.5.1. Définition

**Spécification: une liste à double sens stocke un ensemble de données et permet de parcourir cet ensemble dans l'ordre vers la fin ou vers le début.**

Les opérations autorisées sont les mêmes que pour les listes **plus** :
void **débobiner**(liste @*li*) //place le curseur après la liste
donnée **voirPrécédent**(liste *li*) /* renvoie l'élément qui précède celui sous le curseur (le dernier s'il est après la liste). Quand il n'y a pas de précédent, renvoie l'élément spécial **nil** (ou **null**, ou **NULL**…) */
donnée **précédent**(liste @*li*) /* renvoie le même élément que **voirPrécédent** après y avoir placé le curseur */

Remarque : en général, on ne suppose pas d'opération ajouterAvant() ou supprimerAvant().

### 3.5.2. Un exemple

- **modification d'un texte** en mémoire : on ne peut pas prédire le volume des données en fin de session, ni les endroits où du texte sera ajouté ou modifié. Il faut pourtant à tout instant disposer de l'ensemble des données dans le bon ordre dans la mémoire de travail. De plus, les déplacement de base du curseur sont (en version minimale) descendre ou remonter d'une ligne.

## 4. Une étape vers l'implémentation : les listes chaînées

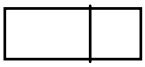
### 4.1.1. Données composées

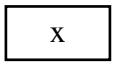
Chaque donnée est stockée à un **emplacement** (qui lui est alloué par sa déclaration ou dynamiquement – voir plus loin)

Tous les emplacements ont une **adresse** grâce à laquelle on peut les retrouver.
L'adresse est une notion abstraite, qui peut être implémentée de plusieurs façons.
L'adresse est une donnée particulière (!!!)

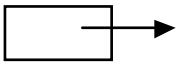
**Notation graphique**

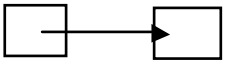

emplacement d'une donnée élémentaire


emplacement d'une donnée composée



cet emplacement contient la donnée x


cet emplacement contient une adresse

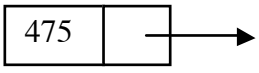

L'emplacement de gauche contient l'adresse de celui de droite

**Convention** : chaque emplacement a un type. S'il convient pour un nombre entier, il ne convient pas pour une chaîne de caractères, une adresse ou un nombre à virgule.

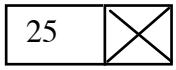
**Notation algorithmique :** pour décrire formellement des opérations sur des données composées, on utilise une notation pointée identique à celles de C, Java, Pascal, etc.

### 4.1.2. Listes

Pour construire des listes, on encapsule chaque donnée de la liste dans une donnée composée appelée **cellule**. Une cellule comporte deux éléments : la donnée de la liste, et l'adresse d'une autre cellule :

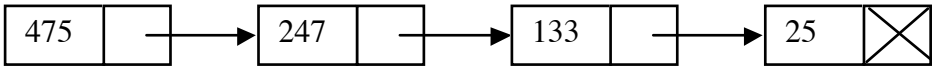



On a un dessin particulier pour signifier que l'emplacement adresse de la cellule est vide (ne contient rien) :

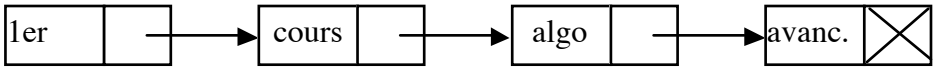


Les cellules permettent de construire des listes : chaque cellule contient l'adresse de la cellule qui la suit, sauf la dernière qui contient l'adresse spéciale **rien**. Pour utiliser la liste, il suffit de connaître sa première cellule (appelée son début, ou sa tête).

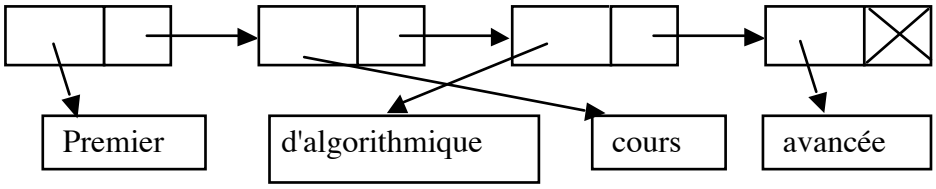
- Une liste de nombres




- Une liste de mots




- Une liste d'adresses




C'est une organisation des données qui a la propriété suivante : les cellules n'ont pas besoin d'être disposées en mémoire dans l'ordre de la liste. On peut donc en permanence ajouter de nouvelles données (au début, au milieu ou à la fin de la liste) sans réorganiser celles qui existent déjà. C'est donc une structure adaptée aux programmes dont les données changent beaucoup pendant l'exécution, mais doivent rester ordonnées.

**Notation** :
Une cellule c comporte 2 données : la donnée de liste et l'adresse du suivant.
Dans les algorithmes, on utilisera pour la première suivant les cas c.donnée, c.nombre, c.mot, etc.

Pour l'adresse, on utilisera c.suivant ; Attention : certains langages comme C et C++ font une différence entre l'adresse (c.suivant) et la cellule à cette adresse (c->suivant). D'autres comme java masquent cette différence en interdisant de manipuler explicitement l'adresse. Dans les algorithmes, j'utiliserai la conception java. Ce qui permet d'écrire x = c.suivant.donnée, c1 = c.suivant.suivant, etc.

## *4.2. Opérations abstraites sur les listes chaînées :*

### 4.2.1. Parcourir

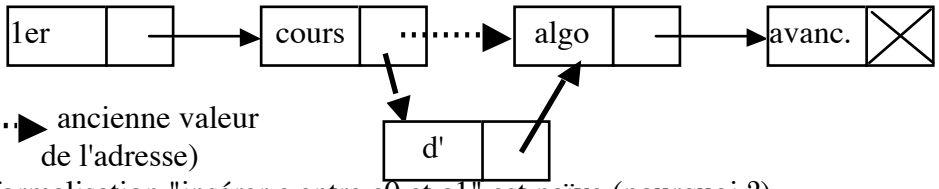
C'est l'opération de base sur les listes : on va du début à la fin de la liste – et pour que ça serve à quelque chose, à chaque cellule on fait une opération qui utilise sa donnée. On a seulement un schéma d'algorithme, parce qu'il y a plusieurs variantes selon le traitement.

Quelques exemples de traitement (ils seront repris dans la partie implémentation) : afficher toutes les données à l'écran, renvoyer la somme ou la moyenne des données, renvoyer la liste des données supérieures à 3.

Algorithme Parcourir
**Entrée :** une cellule *début*
**Sortie** : le type de valeur de retour dépend du traitement
**Variables** : une cellule *courant* plus les variables utiles au traitement
**Début**
courant = début ;
tant que (courant != **rien**) {
TRAITER(courant) ; /* Suivant les cas, TRAITER est un appel de fonction ou le code qui exécute le traitement */
courant = courant.suivant ;
}
/* renvoie éventuellement une valeur */
**fin**

#### 4.2.2. Insérer

on ajoute une cellule entre deux cellules qui auparavant se suivaient




( ⋯▶ ancienne valeur de l'adresse)

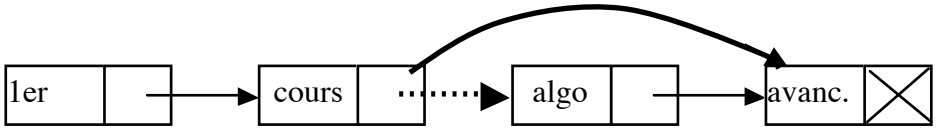
La formalisation "insérer c entre c0 et c1" est naïve (pourquoi ?)

On va modifier l'adresse qui est dans c0, mais on en a besoin pour choisir le suivant de c. Il faut donc faire attention à l'ordre des opérations.

Algorithme Insérer_après
**Entrée :** deux cellules c et c0. L'algorithme insère c après c0.
**Sortie** : pas de valeur de retour
**Variables** : néant
**Début**
c.suivant = c0.suivant ;
c0.suivant = c ;
**fin**

Vérification : si c0 était la fin de liste, comment fonctionne l'algorithme ?

#### 4.2.3. Supprimer



Donner seulement la cellule à supprimer ne suffit pas (pourquoi ?).
On supprime une cellule entre deux autres, mais formaliser ainsi compliquerait inutilement (pourquoi ?)

Algorithme Supprimer_après
**Entrée :** une cellule c0. L'algorithme supprime la cellule qui suit c0.
**Sortie** : pas de valeur de retour /* variante : renvoie la cellule supprimée */
**Variables** : néant
**Début**
c0.suivant = c0.suivant.suivant ;
**fin**