

## La notion de liste (chaînée)

*Tout comme un tableau, une liste est une collection d'éléments. Mais, contrairement aux tableaux, les éléments ne sont pas contigus en mémoire mais « chaînés ». Nous voyons ses avantages et inconvénients par rapport aux tableaux, comment les implémenter en orienté-objet et en non orienté-objet et les opérations de bases sur les listes.*

### Définition

---

Voici quelques définitions glanées dans différents livres sur le sujet. Elles nous indiquent à la fois à quoi cela ressemble et ce que cela peut nous apporter.

- « Une liste est une collection linéaire d'éléments d'information. »  
(Structures de données – Lipschutz – ed Schaum )
- « Une liste est un conteneur séquentiel capable d'insérer et de supprimer des éléments localement de façon constante, c'est-à-dire indépendamment de la taille du conteneur. »  
(Structures de données en Java – Hubbard – ed. Schaum's)
- « Les listes sont des conteneurs destinés aux insertions s'effectuant en temps constant quelle que soit la position dans le conteneur. »  
(La bibliothèque standard STL du C++ - Fontaine – InterEditions)
- « Une liste chaînée est une structure de données dans laquelle les objets sont arrangés linéairement. Toutefois, contrairement au tableau, pour lequel l'ordre linéaire est déterminé par les indices, l'ordre d'une liste chaînée est déterminé par un pointeur dans chaque objet. »  
(Introduction à l'algorithmique – Cormen, Leiserson, Rivest – Dunod )

### Liste vs Liste chaînée

Bien que nous allons, par facilité, parler de "*liste*", il serait plus exact de parler de "*liste chaînée*" ("*linkedlist*" en anglais).

Dans la terminologie moderne, le terme "*liste*" est plutôt utilisé pour dénoter toute collection séquentielle d'éléments (il existe un premier élément, un deuxième, ...). Autrement dit, chaque élément à une position précise dans la collection. En ce sens, un tableau est également une *liste* !

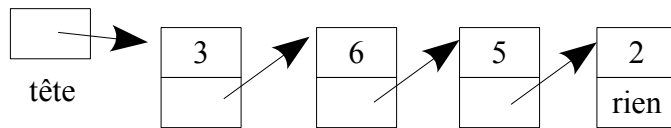
On peut opposer la liste à la structure *d'ensemble* où les éléments ne sont pas positionnés (un élément est ou n'est pas dans un ensemble mais sans que sa position puisse être donnée).

Faites également attention à ne pas confondre avec la notion de tri. On peut parler de la liste contenant les éléments (3, 8 et 2) dans cet ordre (au sens *séquence* et pas ordre de tri)

## Représentation et terminologie

Sous forme compacte, on représente habituellement une liste simplement en indiquant la liste des valeurs entre parenthèse. Par exemple la liste (3,6,5,2).

Visuellement, on la représente plutôt ainsi.

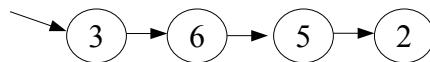


On montre par là que chaque élément de la liste est composé de 2 parties : la valeur proprement dite et un « accès » à son suivant.

De plus, on « connaît » le premier élément de la liste via une zone spéciale la *tête de la liste*.

Le dernier élément n'ayant pas de suivant, on l'indique par « rien » dans la zone réservée à l'accès au suivant. On parle de *fin de liste* ou *queue de la liste*. Dans la littérature vous trouverez parfois, à la place de *rien*, les notations *null*, *nil*,  $\lambda$  ou encore  $\emptyset$ .

Parfois, on rencontre aussi une forme graphique plus compacte



Au contraire du tableau, donc, il n'y a pas d'accès direct à un élément (en donnant son indice) mais il est nécessaire de partir du premier élément et de suivre la « chaîne ». Comment représenter ce lien, cet accès en mémoire ? Cela dépend du type de langage.

En orienté-objet, un élément est représenté par un objet qui possède une référence vers l'objet représentant l'élément suivant.

En non orienté-objet, on utilise la notion de pointeur (vous verrez cela au cours de C/C++). En fin de document, on montre qu'il est même possible de s'en sortir avec un langage qui ne dispose pas de la notion de pointeur (comme Cobol par exemple).

## Rappel : opérations sur les tableaux et complexité

Afin de mieux comprendre ce qu'apporte la liste par rapport aux tableaux, revoyons ce que « coûte » les opérations courantes sur les tableaux.

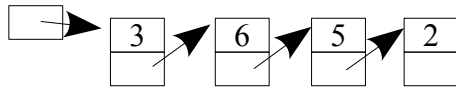
<i>Opération</i>	<i>Complexité</i>
Ajout	O(n)
Suppression	O(n)
Recherche dans tableau non trié	O(n)
Recherche dans tableau trié	O(log <sub>2</sub> n)
Parcours	O(n)

Pour rappel, la notation « O(n) » indique que l'opération prend un temps qui est proportionnel à « n », la taille du tableau. En effet, lors d'une suppression il faut décaler des éléments pour boucher le trou qui est apparu. Il en est de même pour un ajout. Dans le cas d'un tableau trié, l'algorithme de recherche dichotomique permet de rapidement trouver un élément.

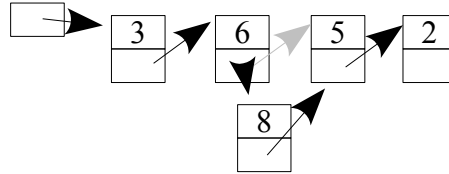
## Ajout d'un élément dans une liste

Dans une liste, il n'est pas nécessaire de décaler des éléments pour en ajouter un. Reprenons la liste définie plus haut et ajoutons l'élément 8 entre le 6 et le 5

Avant :



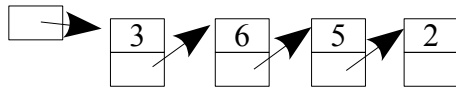
Après :



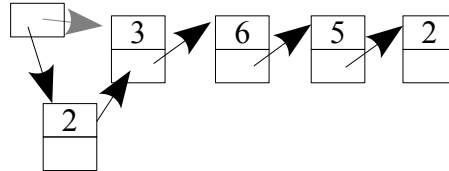
Il aura suffi de modifier des « flèches » (des « liens », des « accès »). L'opération prend donc le même temps quelle que soit la taille de la liste (pour autant qu'on soit déjà positionné à l'endroit de l'ajout). On note cela en disant que la complexité est  $O(1)$ .

Remarquons qu'un ajout en tête de liste est un cas particulier puisqu'on modifie la tête de liste

Avant :



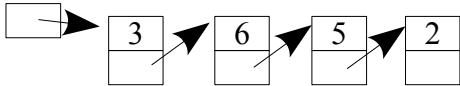
Après :



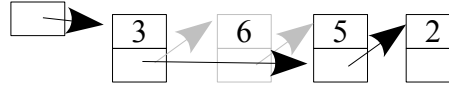
## Suppression d'un élément dans une liste

Il en est de même pour la suppression d'un élément. Illustrons cela en supprimant l'élément 6.

Avant :

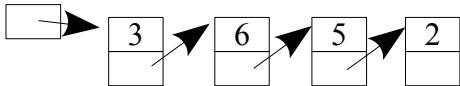


Après :

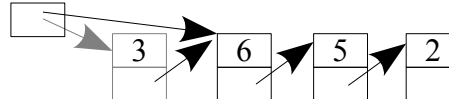


A nouveau l'opération est de complexité  $O(1)$  et le cas de la suppression du premier élément est particulier.

Avant :



Après :



## Recherche d'un élément

Au contraire d'un tableau, le fait de savoir que la liste est triée ne permet pas de développer un algorithme rapide de type « recherche logarithmique ». Tout au plus on peut, comme avec un tableau, s'arrêter lorsqu'on dépasse la position où l'élément recherché *aurait* du se trouver.

## Liste vs Tableau

Synthétisons ce que l'on a vu de la complexité des listes.

<i>Opération</i>	<i>Tableau</i>	<i>Liste</i>
Ajout	O(n)	O(1)
Suppression	O(n)	O(1)
Recherche dans collection non trié	O(n)	O(n)
Recherche dans collection trié	O(log <sub>2</sub> n)	O(n)
Parcours	O(n)	O(n)

Une liste montre tout son intérêt pour les ajouts/suppressions mais se montre moins efficace pour les recherches si l'information est triée.

Le choix d'un tableau ou d'une liste pour représenter une collection dans un problème précis dépend dès lors du rapport entre le nombre d'ajout/suppression et de recherche que l'on compte effectuer.

Remarquons aussi que si les ajouts/suppressions se font toujours en *fin* de tableau (le cas d'une *pile* par exemple), la complexité rejoint celle d'une liste.

## Représentation OO

Voici une description simple d'une liste en Orienté-Objet. Plus loin dans le cours nous aurons l'occasion de la critiquer et de l'améliorer.

```
classe ElémentListe
privé :
    valeur : Objet
    suivant : ElémentListe
public :
    constructeur ElémentListe ( ↓ v : Objet , ↓ s : ElémentListe )
    constructeur ElémentListe ( ↓ v : Objet ) // suivant à null
    méthode valeur ( ) → Objet
    méthode affValeur ( ↓ v : Objet )
    méthode suivant ( ) → ElémentListe // ou rien si pas de suivant
    méthode affSuivant ( ↓ s : ElémentListe )
fin classe

classe Liste
privé :    premier : ElémentListe
public :
    constructeur Liste ( )
    méthode premier ( ) → ElémentListe
    méthode estVide ( ) → booléen
    méthode taille ( ) → entier
    méthode insérerTête ( ↓ v : Objet )
    méthode insérerAprès ( ↑ e : ElémentListe , ↓ v : objet )
    méthode supprimerTête ( ) → Objet // retourne la valeur supprimée
    méthode supprimerAprès ( ↑ e : ElémentListe ) → Objet // idem
fin classe
```

### Remarques sur les notations

- **Objets** : nous respectons les conventions Java qui sont également celles que vous avez utilisées en première année. En C++, par contre, une variable peut désigner directement un objet ou bien être une référence à l'objet (ce dernier cas devant être indiqué explicitement). Soyez donc prudent

lorsque vous traduisez une logique en C++.

- **Paramètres :**

- x ↓ : indique que la valeur initiale du paramètre est nécessaire à l'algorithme pour fonctionner.
- x ↑ : indique que l'algorithme va donner une valeur au paramètre.
- x ⇕ : indique que l'algorithme va utiliser la valeur de départ mais la modifier.
- x On pourra aussi indiquer IN, OUT et IN-OUT en lieu et place des flèches.
- x Si on n'indique rien, il faut comprendre qu'il s'agit d'un paramètre en entrée (IN)

Exemples :

- x Tri d'un tableau : le tableau est en IN-OUT
- x Calcul du maximum d'un tableau : le tableau est en IN et le maximum en OUT (ou en valeur de retour)
- x Swap de 2 entiers : ils sont en IN-OUT

Remarques sur les méthodes et attributs

- Nous avons une méthode insérerTête() mais pas de méthode insérerFin(). Cette dernière serait très lente car il faut d'abord parcourir toute la liste pour arriver au dernier. On pourrait améliorer la situation en ajoutant un troisième attribut, nommé *dernier*, qui donne constamment accès au dernier élément de la liste. L'ajout en fin de liste devient alors trivial mais les autres méthodes s'en trouvent légèrement complexifiées car elle doivent maintenir la validité de cet attribut.
- Dans le même ordre d'idée, on peut envisager l'ajout d'autres attributs pour accélérer certaines opérations. Par exemple, on pourrait garder la taille de la liste. On parle d'attribut « calculable » pour indiquer un attribut qui contient une information qui pourrait être calculée à partir des autres attributs. La pratique montre qu'il ne faut pas abuser de ce type d'attribut.
- Les valeurs contenues dans la liste sont indiquées par le type "Objet". Pour le moment, on, peut lire cela comme "n'importe quel type". Plus tard, nous verrons quelles sont les possibilités OO pour décrire en une seule classe des listes pour tous les types d'éléments.
- A votre avis, pourquoi n'y a t-il pas de méthode supprimer(↓ e: ElémentListe) qui supprime de la liste l'élément donné en paramètre ?

Implémentation de la classe **ElémentListe**

```
constructeur ElémentListe ( ↓ v: Objet ,  
                        ⇕ s : ElémentListe )  
    valeur ← v  
    suivant ← s  
fin constructeur  
  
méthode valeur() → Objet  
    retourner valeur  
fin méthode  
  
méthode affValeur (↓ v: Objet)  
    valeur ← v  
fin méthode
```

```
constructeur ElémentListe ( ↓ v: Objet )  
    valeur ← v  
    suivant ← rien  
fin constructeur  
  
méthode suivant() → ElémentListe  
    retourner suivant  
fin méthode  
  
méthode affSuivant( ↓ s: ElémentListe )  
    suivant ← s  
fin méthode
```

## Implémentation de la classe Liste

- Dans chacune des méthodes ci-dessous, il faudrait vérifier qu'aucun des paramètres de type référence n'a la valeur *rien* auquel cas il faudrait envoyer une exception de type "paramètre invalide". Nous ne l'indiquons pas explicitement.

```
constructeur Liste()
    premier ← rien
fin constructeur

méthode premier() → ElémentListe
    retourner premier
fin méthode

méthode estVide() → booléen
    retourner (premier=rien)
fin méthode

méthode taille() → Entier
    taille : Entier
    elt : ElémentListe
    taille ← 0
    elt ← premier
    tantque elt ≠ rien faire
        taille ← taille + 1
        elt ← elt.suivant()
    fin tantque
    retourner taille
fin méthode

méthode insérerTête (↓ v: Objet)
    elt : ElémentListe
    elt ← nouveau ElémentListe(v, premier)
    premier ← elt
fin méthode

méthode insérerAprès (↑ elt : ElémentListe ; ↓ v : objet )
    eltAInsérer : ElémentListe
    eltAInsérer ← nouveau ElémentListe(v, elt.suivant())
    elt.affSuivant(eltAInsérer)
fin méthode

méthode supprimerTête() → Objet
// Les lignes en italique sont nécessaires si la libération de l'espace inutilisé
// est à charge du programme (comme en C++)
    valeur : Objet
    sauve : ElémentListe
    si premier = rien alors ERREUR fin si
    valeur ← premier.valeur()
    sauve ← premier
    premier ← premier.suivant()
    libérer sauve
    retourner valeur
fin méthode
```

```

méthode supprimerAprès(↑ elt : ÉlémentListe ) → Objet
    valeur : Objet
    sauve : ÉlémentListe
    si elt.suivant()=rien alors ERREUR fin si
    valeur ← elt.suivant().valeur()
    sauve ← elt.suivant()
    elt.affSuivant(sauve.suivant())
    libérer sauve
    retourner valeur
fin méthode

```

## Exercices

1. Commençons simplement et plaçons nous dans le cas d'une liste contenant des valeurs non triées.

Écrivez les méthodes suivantes (dans la classe Liste).

- **méthode** insérer( v : objet )  
qui insère dans la liste la valeur donnée (à l'endroit que vous voulez).
- **méthode** supprimer( v : objet ) → booléen  
qui supprime de la liste la valeur donnée. La valeur retournée est « vrai » si la valeur a pu être supprimée et « faux » si ce n'est pas le cas (parce que la valeur ne s'y trouvait pas). Si la valeur s'y trouve en plusieurs exemplaires, on supprime la première occurrence.
- **méthode** existe( v : objet ) → booléen  
qui indique si la valeur s'y trouve (en un ou plusieurs exemplaires)
- **méthode** rechercher( v : objet ) → ÉlémentListe  
qui recherche dans la liste la valeur donnée et retourne un accès à l'élément de liste qui le contient. Si la valeur ne s'y trouve pas, on retourne *null*. Si la valeur s'y trouve en plusieurs exemplaires, on retourne la première occurrence.  
*A première vue, cette méthode peut paraître redondante avec la précédente puisqu'elle ne fait rien de plus que retourner une information qu'on possède déjà, l'élément recherché. Vous avez raison. En pratique toutefois, la recherche se fait sur une partie de l'information, la clé.*

### Correction

```

méthode insérer( v : objet )
    insérerTête(v)
fin méthode

méthode existe( v : objet ) → booléen
    retourner (rechercher(v) ≠ rien)
fin méthode

méthode rechercher( v : objet ) → ÉlémentListe
    courant : ÉlémentListe
    courant ← premier
    tant que courant ≠ rien et courant.valeur() ≠ v faire
        courant ← courant.suivant()
    fin tant que
    retourner courant
fin méthode

```

```

méthode supprimer( v : objet ) → booléen
    précédent, courant : ÉlémentListe
    trouvé : booléen
    courant ← premier
    précédent ← rien
    tant que courant ≠ rien et courant.valeur() ≠ v faire
        précédent ← courant
        courant ← courant.suivant()
    fin tant que
    trouvé ← ( courant ≠ rien )
    // Attention ! ne pas écrire : trouvé ← ( courant.valeur() = v ). Pourquoi ?
    si trouvé alors
        si courant = premier alors
            supprimerTête()
        sinon
            supprimerAprès(précédent)
        fin si
    fin si
    retourner trouvé
fin méthode

```

2. Identifiez les cas limites pour la suppression et montrer que l'algorithme les traite.
3. Nous vous avons demandé de placer les méthodes dans la classe Liste. Qu'en pensez-vous ?
4. Plaçons nous à présent dans le cas d'une liste **triée sans doublon**. Écrivez les méthodes :

- **méthode** insérer( v : objet ) → booléen  
qui insère dans la liste la valeur donnée. Retourne faux si la valeur y était déjà.
- **méthode** supprimer( v : objet ) → booléen  
qui supprime de la liste la valeur donnée. Retourne « vrai » si la valeur a pu être supprimée.
- **méthode** existe( v : objet ) → booléen qui indique si la valeur si trouve
- **méthode** rechercher( v : objet ) → accès à ÉlémentListe  
qui recherche dans la liste la valeur donnée et retourne un accès à l'élément de liste qui le contient. Si la valeur ne s'y trouve pas, on retourne *null*.  
Conseil : Ces 4 méthodes partagent un code commun. Isolez-le.

## Représentation NON OO (C like)

Par rapport à votre cours de Logique de 1ère année, nous devons introduire de nouvelles notations :

- « accès à » : dans un type indiquera une référence (un pointeur, un accès) vers un élément d'un type donné. Ex : *p* : accès à *Personne* indique que *p* permet d'accéder à une personne. Dans le cadre de ce cours, nous n'aurons que des accès à des structures mais les langages non objets permettent généralement des accès vers des éléments de type quelconque.
- Dans le contexte des références, il faut pouvoir créer dynamiquement l'élément référencé. Cela s'indique par la notation « Allouer(type de l'élément référencé) ». Ex: *p* ← Allouer(Personne)

```

Structure ÉlémentListe
    valeur : type
    suivant : accès à ÉlémentListe
Fin Structure

```

```

Module newElémentListe ( v : Type ; s : accès à ElémentListe )
    → accès à ElémentListe
Module suivant ( e : accès à ElémentListe ) → accès à ElémentListe
Module valeur ( e : accès à ElémentListe ) → type
Module affSuivant ( ↑ e : accès à ElémentListe ; ↓ s : accès à ElémentListe )
Module affValeur ( ↑ e : accès à ElémentListe ; ↓ v : type )

Structure Liste
    premier : accès à ElémentListe
Fin Structure

Module newListe ( ) → accès à Liste
Module premier ( ↓ l : accès à Liste ) → accès à ElémentListe
Module estVide ( ↓ l : accès à Liste ) → booléen
Module taille ( ↓ l : accès à Liste ) → entier
Module insérerTête ( ↑ l : accès à Liste ; ↓ v : type )
Module insérerAprès ( ↑ e : accès à ElémentListe ; ↓ v : type )
Module supprimerTête ( ↑ l : accès à Liste ) → type
Module supprimerAprès ( ↑ e : accès à ElémentListe ) → type

```

### Implémentation

L'implémentation est très proche de celle donnée dans la version OO. Pointons 2 différences.

- Une évidente différence de notation puisque le paramètre principal est donné dans la liste des paramètres et pas mis en avant comme en OO. Ainsi, on écrira "suivant(elt)" et pas "elt.suivant()".
- Le module newElémentListe (qui correspond au constructeur dans la version OO) doit d'abord allouer l'espace avant d'initialiser les attributs.

Tout le reste est strictement identique. Nous en détaillons certains pour l'exemple et vous laissons les autres en exercice.

```

Module newElémentListe ( v : Type ; s : accès à ElémentListe ) → accès à ElémentListe
    elt : accès à ElémentListe
    elt ← Allouer(ElémentListe)
    elt.valeur ← v
    elt.suivant ← s
    retourner elt
fin module

Module insérerTête ( ↑ l : accès à Liste ; ↓ v : type )
    elt : accès à ElémentListe
    elt ← newElémentListe(v, l.premier)
    l.premier ← elt
fin module

Module supprimerAprès ( ↑ elt : accès à ElémentListe ) → type
    valeur : type
    sauve : accès à ElémentListe
    si suivant(elt)=rien alors ERREUR fin si
    valeur ← valeur(suivant(elt))
    sauve ← suivant(elt)
    affSuivant( elt, suivant(sauve))
    libérer sauve
    retourner valeur
fin module

```

## Représentation non OO (sans pointeur)

La plupart des langages non orienté-objet possèdent la notion de pointeur mais il y a des exceptions (Cobol par exemple). Comment implémenter une liste avec un tel langage ? En simulant la liste via un tableau qui contient 2 champs (la valeur de l'élément et l'indice de l'élément suivant).

Exemple : Il faut aussi savoir où commencer, c-à-d connaître l'indice du premier élément (3 dans notre exemple).

1 2 3 4 5

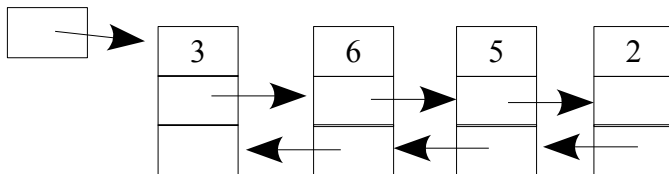
2	6	3	5	
0	4	2	1	

Pour ajouter un élément, il suffit de l'ajouter en fin de tableau et d'adapter les indices sur la deuxième ligne. La suppression est aussi aisée mais crée un « trou » qu'il faut pouvoir gérer (par exemple via une liste des places libres).

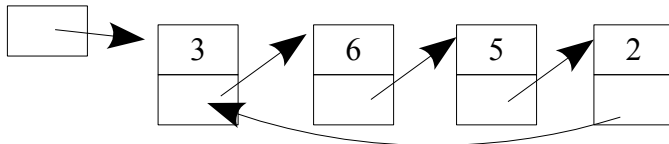
## Autres types de listes

Il existe quelques variantes à la liste que nous venons de voir.

- La liste **bidirectionnelle** : chaque élément possède, en plus d'une référence à l'élément suivant, une référence à l'élément *précédent*. Cela facilite grandement la suppression car il n'est plus nécessaire de connaître l'élément précédent pour pouvoir le faire car on peut le retrouver à partir de l'élément à supprimer. Visuellement, cela donne :



- La liste **circulaire** : le dernier élément ne référence pas « rien » mais le premier élément. Cela donne :



- La liste **bidirectionnelle circulaire** : Une combinaison des 2. Le premier élément a pour élément précédent le dernier de la liste.

## Exercices

1. Récrire les classes OO pour les adapter à ces variantes de listes. Il vous faudra peut-être modifier la signature des méthodes voire même en supprimer et en ajouter.
2. Reprendre l'exercice d'insertion/suppression/recherche dans une liste triée pour ces 3 nouvelles variantes de liste.

## Exercices récapitulatifs

---

### 1. UN TASSEMENT DE LISTE

Soit une liste linéaire d'entiers. Les éléments sont ordonnés en ordre croissant sur les valeurs qui peuvent se répéter. Plusieurs éléments consécutifs peuvent donc avoir la même valeur !

On demande d'écrire l'algorithme qui supprime de cette liste tous les éléments ayant une valeur redondante (déjà existante). La liste résultante ne contiendra donc plus qu'une seule occurrence des valeurs présentes initialement (la première rencontrée). **Il ne peut** y avoir de nouvelles allocations pour résoudre le problème.

Exemple : La liste (5,5,7,7,7,11,13,13,15) deviendra (5,7,11,13,15)

### 2. INTER-CLASSEMENT DE LISTE

Soient L1 et L2 deux listes monodirectionnelles de même type d'éléments. On demande de créer la liste L3 issue de l'inter-classement des éléments de ces deux listes (sans aucune nouvelle demande d'allocation). Les éléments successifs de la liste L3 seront donc le premier élément de L1, le premier élément de L2, le deuxième élément de L1, le deuxième élément de L2 et ainsi de suite. Quand l'inter-classement n'est plus possible (une des deux listes est plus courte), on rattache le reste de la liste plus longue à la fin de L3.

La méthode est à écrire dans la classe Liste (elle reçoit donc UNE liste en paramètre).

### 3. LE GRAND NETTOYAGE

Soit L, une liste linéaire d'entiers. Ces valeurs ne sont pas ordonnées et sont dans tous les cas des entiers compris entre 1 et 10 inclus. On demande d'écrire l'algorithme qui supprime de cette liste toutes les occurrences de la valeur la plus fréquente.

Exemple : Pour la liste (8,7,6,7,7,1,7,1,5,7) on supprime tous les 7.

N.B. : en cas d'*ex-æquo*, c'est la plus grande des valeurs qui est éliminée.

### 4. LES ENSEMBLES

Définissons une classe Ensemble représentant un ensemble d'entiers (au sens mathématique du terme). Voici à quoi elle ressemble. Donnez le code des méthodes.

```
classe Ensemble
    privé : valeurs : liste d'entiers // les valeurs seront ordonnées
    public :
        constructeur Ensemble() // crée un ensemble vide
        méthode ajoute( val : entier ) // ajoute une valeur à l'ensemble
        méthode estDans( val : entier ) → booléen // indique si la valeur est présente
        méthode union( e2 : Ensemble ) → Ensemble // crée un 3ème ensemble union des 2
        méthode intersection( e2 : Ensemble ) → Ensemble // crée un 3ème ensemble intersection des 2
        méthode complément( e2 : Ensemble ) → Ensemble // le résultat contient les éléments de l'ensemble qui ne sont pas dans e2
fin classe
```

## 5. L'ALGORITHME DU SURVIVANT

L'empereur romain Exterminus avait l'habitude d'organiser de nombreuses festivités auxquelles il conviait, paradoxalement, les personnes qu'il n'aimait guère. Il leur offrait moult ripailles ainsi que toutes les fantaisies que la morale interdit de détailler, avant de les inviter à participer à un jeu qui était pour lui, le clou de la journée, et pour eux, le clou de leur cercueil. Voyez plutôt.

Tous devaient venir se répartir uniformément autour d'une grande table ronde, sur un tabouret numéroté. Il y avait autant de tabourets que d'invités et chacun pouvait choisir son tabouret. Les numéros se suivaient comme les nombres naturels à partir de 1 et ce, dans le sens horloger (Notez que ce sens était inconnu à l'époque!).

Ensuite, l'empereur consultait les auspices en éventrant un poulet, ceci afin de compter le nombre de graines que contenait son gésier. Les poulets étaient bien nourris et l'empereur en extrayait toujours au moins une graine. Le nombre de graines allait lui servir de pas de décompte lors de sa ronde démoniaque! Il commençait son décompte derrière l'invité assis sur le tabouret 1 et poursuivait dans le sens horloger en comptant 2 derrière le suivant, puis 3, et ainsi de suite jusqu'à ce que son décompte atteigne le pas initial. Le malheureux derrière lequel il venait de s'arrêter allait être le premier à être envoyé au trépas (avec son tabouret). Il continuait sa ronde en entamant un nouveau décompte à partir du suivant de l'exterminé, et ce, jusqu'à ce qu'il n'en reste plus qu'un. L'heureux survivant était tout heureux de se voir offrir un séjour d'un an dans un camp de vacances à Petitbonum.

Réaliser l'algorithme de simulation de cette extermination. Les paramètres sont le nombre initial d'invités et le pas de décompte. Le résultat à obtenir est la position où il faudrait s'asseoir pour rester le dernier survivant.

## 6. TRI FUSION PAR ECLATEMENT DES MONOTONIES CROISSANTES.

Soit une liste chaînée à valeurs entières. Une monotonie croissante est une suite d'éléments qui se suivent et dont les valeurs sont dans l'ordre croissant. Le but de cette méthode est d'extraire une monotonie, puis une autre, et ainsi de suite, et de fusionner chaque monotonie extraite avec le résultat des fusions précédentes. Après la dernière fusion, la liste obtenue sera ordonnée, et la liste initiale aura donc été transformée en une liste triée. Réalisez l'algorithme.

## 7. POLYNOME

### Rappels

- Un *monôme* est de la forme  $c \cdot x^e$  où  $c$  est le coefficient,  $x$  l'inconnue et  $e$  l'exposant (exemple :  $3x^2$ )

- Un *polynôme* est une somme de monômes (ex:  $3x^4 - x^2 + 2$ )

### Représentation

Convenons de représenter un monôme comme une simple classe avec 2 attributs : le coefficient et l'exposant (l'inconnue sera toujours  $x$ ). Représentons un polynôme comme une liste de monômes triée en décroissant sur l'exposant.

### Problème

On demande d'écrire une méthode permettant d'additionner 2 polynômes et une autre permettant de les soustraire.

### Réflexion

Il est plus que probable que les 2 algorithmes que vous venez d'écrire ne diffèrent qu'en un seul endroit. Dans cette situation, il faut toujours se demander si il n'est pas possible de factoriser le code, c'est-à-dire de le modifier pour n'avoir plus qu'un seul exemplaire du code similaire.

Avez-vous une suggestion ? Est-ce que votre solution tient toujours la route si on ajoute un troisième opérateur binaire (par exemple, une addition modulaire) ? Une solution idéale devrait permettre d'ajouter des cas sans devoir modifier le code existant.

## 8. MATRICE CREUSE.

Une matrice creuse est une matrice où la plupart des éléments sont nuls. On les rencontre beaucoup dans des problèmes physiques impliquant des systèmes linéaires. Représenter une telle matrice par un tableau classique à 2 dimensions n'est efficace ni en terme d'espace mémoire (une matrice 1000x1000 va contenir un million de valeurs presque toutes nulles) ni en terme de temps de calcul (beaucoup de calcul avec des 0). C'est pourquoi on utilise souvent dans ce cas là une maillage de listes (une liste par ligne et une liste par colonne).

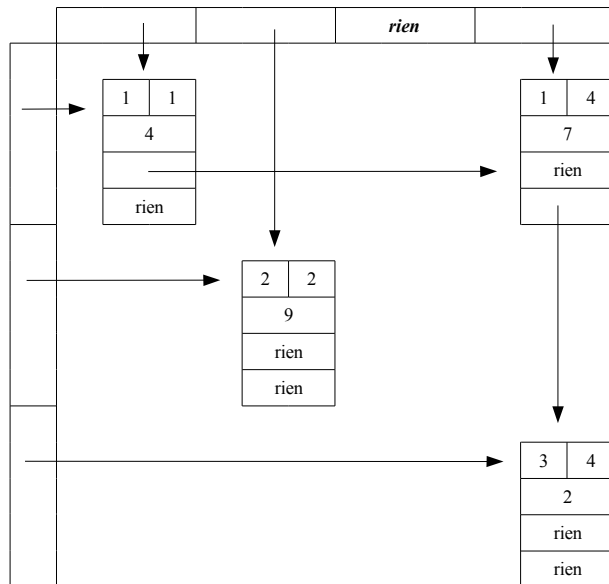
Exemple : La matrice

4	0	0	7
0	9	0	0
0	0	0	2

se représente comme indiqué à droite

- Chaque élément de la liste contient le numéro de ligne et de colonne, la valeur, un lien vers l'élément non nul suivant dans la ligne et un lien vers l'élément non nul suivant dans la colonne.

- On dispose également d'un tableau pour les têtes de lignes et un tableau pour les têtes de colonnes.



On vous demande de :

- Définir les attributs de la classe `ElémentMatriceCreuse` représentant un élément de la matrice.
- Définir les attributs de la classe `MatriceCreuse` ainsi que

```

constructeur MatriceCreuse (nbLignes, nbColonnes)
    // construit une matrice vide
méthode get(ligne,col) → Entier
    // qui donne la valeur dans la matrice en position (ligne,col)
méthode set(ligne,col,valeur)
    // qui met la valeur dans la matrice en position (ligne,col)
    // si la valeur est nulle, il faudra peut-être supprimer un élément !
    // au contraire, si la valeur est non nulle, il faudra peut-être ajouter un élément !
    
```