# ALG. Listes chaînées

## I/ Structure d'un tableau

La structure d'un tableau permet de stocker des **séquences** d'éléments : les **éléments** sont **contigus** et **ordonnés** en mémoire, un numéro d'ordre appelé **indice** permet de repérer chacun des éléments du tableau. Elle n'est pas adaptée à toutes les opérations que l'on pourrait vouloir effectuer.
Les tableaux de Python permettent par exemple **d'insérer** ou de **supprimer efficacement** les éléments en **fin de tableau** à l'aide des méthodes *append()* et *pop()* mais se prêtent mal à l'insertion ou suppression d'éléments à une autre position car insérer un élément demande donc de déplacer tous les éléments qui le suivent pour lui laisser une place ! On utilise en Python la méthode *insert(indice, valeur)* pour se faire mais elle n'est pas efficace.

Exemple : quelques opérations sur un tableau en Python

```
tab = [3,6,8,-1,"a",7.4]
tab.append("coucou") # Ajout de "coucou" en fin de tableau
print(tab)
tab.pop() # Suppression du dernier élément ("coucou" ici)
print(tab)
tab.insert(0,10)
print(tab)
```

```
[3, 6, 8, -1, 'a', 7.4, 'coucou']
[3, 6, 8, -1, 'a', 7.4]
[10, 3, 6, 8, -1, 'a', 7.4]
```

Cette méthode *insert(0, 10)* est très coûteuse puisqu'il faut :

- Ajouter un élément en fin de tableau.
- Déplacer tous les éléments vers la « droite ».
- Ecrire « 10 » à la première place du tableau.

Cela revient à effectuer le programme suivant :

```
# Programme équivalent à insert(0,10)
tab = [3,6,8,-1,"a",7.4]
tab.append(None)
for i in range(len(tab)-1,0,-1) :# On part de la fin
    tab[i] = tab[i-1]
tab[0] = 10
print(tab)
```

```
[10, 3, 6, 8, -1, 'a', 7.4]
```

Le coût est donc ici **linéaire** avec le parcours du tableau par la boucle *for*. Supprimer un élément du tableau reviendrait au même (sauf le dernier bien sûr).

A noter également que dans certains langages comme le C, les tableaux doivent contenir des éléments de même type et ne sont pas redimensionnables directement (il faut recréer explicitement un tableau).

**La liste chaînée** permet d'apporter une meilleure solution à l'insertion et suppression d'éléments. Cette structure de données servira aussi de fondement d'autres structures comme les **piles** et les **files** qui seront également étudiées.

# II/ Structures de données

## 1/ Généralités

En informatique, une *variable* a un nom et correspond à de l'information placée en mémoire. On y accède grâce à son adresse.
Le *type de donnée* (ou *type*) est contenu dans cette information : une valeur contenant à un ensemble précis (nombre entier, flottant, booléen etc.) et l'ensemble des opérations autorisées avec cette valeur.
En langage Python, des types de données simples ou construits sont déjà implémentés : entiers, booléens etc. sont des *types simples* et listes, dictionnaires etc. sont des *types construits*. Il n'est pas nécessaire de savoir comment ces types sont représentés dans la machine pour pouvoir les utiliser.

Afin d'organiser et traiter des données, les manipuler avec des algorithmes, et donc obtenir un traitement informatique efficace, on peut définir une **structure de données**. Celle-ci peut regrouper des objets de types différents et permet de stocker les données de manière particulière.
Un exemple que l'on peut citer est l'organisation sous forme de **classe**.

Exemple : une structure de données en Python (*classe*) et en langage C (*structure*).

*En Python*

```python
class Temps :
    def __init__(self) :
        self.heures = 0
        self.minutes = 0
        self.secondes = 0
```

*En langage C*

```c
struct temps {
    unsigned heures;
    unsigned minutes;
    double secondes;
};
```

Remarque : dans les deux cas, les valeurs sont initialisées à zéro. On peut évidemment modifier les valeurs.

## 2/ Caractéristiques

Une structure de données a plusieurs caractéristiques.

- **Structure linéaire** ou **non linéaire** : si les éléments sont ordonnés, s'il on y accède séquentiellement, on parle de structure linéaire. Un tableau ou une classe en sont des exemples. Sinon il n'y a pas d'ordre mais simplement des relations entre éléments comme dans des arbres ou graphes.
- **Structure homogène** ou **hétérogène (non homogène)** : si les éléments sont du même *type* (un tableau en langage C l'est forcément) ou pas comme dans une classe. Une structure homogène est plus efficace à parcourir que son homologue hétérogène.
- **Structure statique** ou **dynamique** : si la taille est fixée et ne peut plus être modifiée ou si elle est peut varier.

Voici un tableau récapitulatif :

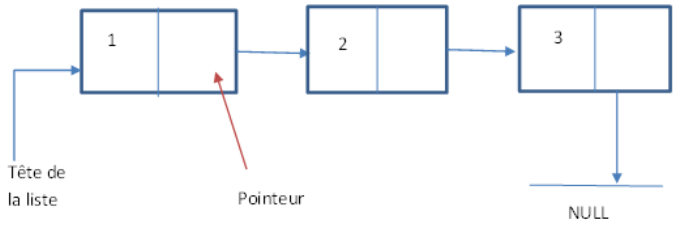
| *Structures élémentaires* | *Structures complexes* | *Complexité* |
|---|---|---|
| Entier | Tableau | Linéaire |
| Flottant | Pile et File | Linéaire |
| Booléen | Classe (structure en langage C) | Linéaire |
| Caractère | Liste chaînée et variantes | Linéaire |
| | Arbre | Non linéaire |
| | Graphe | Non linéaire |

Comment choisir une structure de données ? Le choix dépend du type des informations utilisées, de la manière de stocker une donnée et du type d'algorithme utilisé.
Par exemple, si l'on a besoin d'accéder rapidement à un élément dans un ensemble ordonné, un tableau indexé est une bonne idée. A l'inverse, si l'on parcourt fréquemment une structure en entier, avec des ajouts / suppressions fréquents d'éléments, une liste chaînée est une bonne solution.

# III/ Structure de liste chaînée

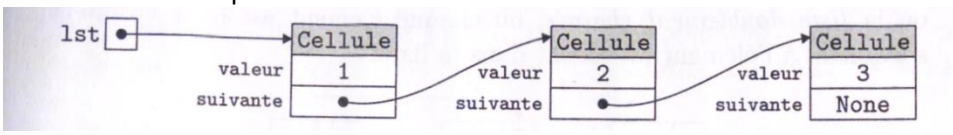
## 1/ Définition

Une liste chaînée permet avant tout de représenter une **liste**, c'est-à-dire une **séquence finie de valeurs**. Comme son nom le suggère, cette structure est caractérisée par le fait que les éléments sont liés entre eux par une « chaîne » : chaque élément connait l'adresse mémoire de son suivant via un *pointeur* et une valeur NULL (*None* pour Python) indique la terminaison de la liste.
Voici une schéma explicatif :




Remarque : contrairement à un tableau, **les éléments ne sont pas contigus en mémoire**, ce qui facilitera grandement l'ajout ou suppression d'éléments dans cette structure.

On souhaite créer une liste chaînée (super intéressante 😊 ) contenant les valeurs 1 ; 2 et 3.
Voici un schéma explicatif :




Voici un programme en Python :

```
# Définit un élément d'une liste chaînée
class Cellule :
    def __init__(self,val,suiv) :
        self.valeur = val
        self.suivante = suiv

# Création d'une liste chaînée 1,2,3
lst = Cellule(1, Cellule(2, Cellule(3,None)))

# Adresse mémoire de la première cellule
print(lst)

# Affichage des valeurs de la liste
# Tant que l'on n'est pas en fin de liste
while lst is not None :
    print(lst.valeur,end = " ")
    lst = lst.suivante
```

```
<__main__.Cellule object at 0x00000233DA2D4790>
1 2 3
```

La variable *liste* qui représente la liste chaînée est construite à partir d'appels imbriqués de constructeurs de la classe *Cellule* : une construction dite par **récursivité** sera plus adaptée (cette notion sera vue ultérieurement).
On remarquera l'affichage de l'adresse mémoire de la première cellule avec l'instruction *print(liste)*.

## 2/ Longueur d'une liste chaînée

Cette méthode renvoie le nombre d'élément de la liste. Elle est de zéro si la liste est vide.

Voici un programme en Python :

```
# Définit un élément d'une liste chaînée
class Cellule :
    def __init__(self,val,suiv) :
        self.valeur = val
        self.suivante = suiv

    def longueur(self) :
        long = 0
        liste = self # Liste de départ
        # Tant que la liste n'est pas vide
        while liste is not None :
            long += 1
            liste = liste.suivante

        return long

# Création d'une liste chaînée 1,2,3
lst = Cellule(1, Cellule(2, Cellule(3,None))) # Longueur de 3
# Longueur de 5
lst2 = Cellule(1, Cellule(2, Cellule(3, Cellule(5, Cellule (6,None)))))
print(f"la longueur est",lst.longueur())
print(f"la longueur est",lst2.longueur())
```

```
la longueur est 3
la longueur est 5
```

## 3/ Renvoi d'un élément d'une liste chaînée

Contrairement à un tableau, on ne peut pas accéder directement à un élément d'une liste chaînée. Cela est normal car la notion d'indice n'existe pas dans cette structure linéaire.
Si l'on veut par exemple le 4ème élément, il faut parcourir la liste jusqu'à l'élément souhaité.

Voici un programme permettant d'accéder à un élément précis d'une liste chaînée (méthode *get(self, n)*):

```
# Définit un élément d'une liste chaînée
class Cellule :
    def __init__(self,val,suiv) :
        self.valeur = val
        self.suivante = suiv

    def longueur(self) :
        long = 0
        liste = self # Liste de départ
        # Tant que la liste n'est pas vide
        while liste is not None :
            long += 1
            liste = liste.suivante

        return long
```

```
# Renvoie le nième élément
# Premier élément = 0
def get(self, n) :
    # On contrôle la validité de n
    if n < 0 or n >= self.longueur() :
        raise IndexError("Erreur d'indice")

    liste = self
    num = 0
    valeur = None
    # Parcours séquentiel de la liste
    while liste is not None :
        valeur = liste.valeur
        # Si on arrive au nième élément, on quitte la boucle
        if num == n :
            break
        num += 1
        liste = liste.suivante

    return valeur
```

```
# Création d'une liste chaînée 1,2,"a","x",6
lst2 = Cellule(1, Cellule(2, Cellule("a", Cellule("x", Cellule (6,None)))))
print(lst2.get(0)) # Attendu "1"
print(lst2.get(3)) # Attendu "x"
print(lst2.get(5)) # Erreur
```

```
1
x

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-70d2600a90cb> in <module>
     41 print(lst2.get(0)) # Attendu "1"
     42 print(lst2.get(3)) # Attendu "x"
---> 43 print(lst2.get(5)) # Erreur

<ipython-input-3-70d2600a90cb> in get(self, n)
     20         # On contrôle la validité de n
     21         if n < 0 or n >= self.longueur() :
---> 22             raise IndexError("Erreur d'indice")
     23
     24         liste = self

IndexError: Erreur d'indice
```

Le fait de devoir parcourir la liste pour trouver l'élément indique que la **complexité** de sa recherche est **linéaire**.

Remarque : il s'agit ici d'une structure non homogène, ce qui ne pose pas de problèmes en particulier.

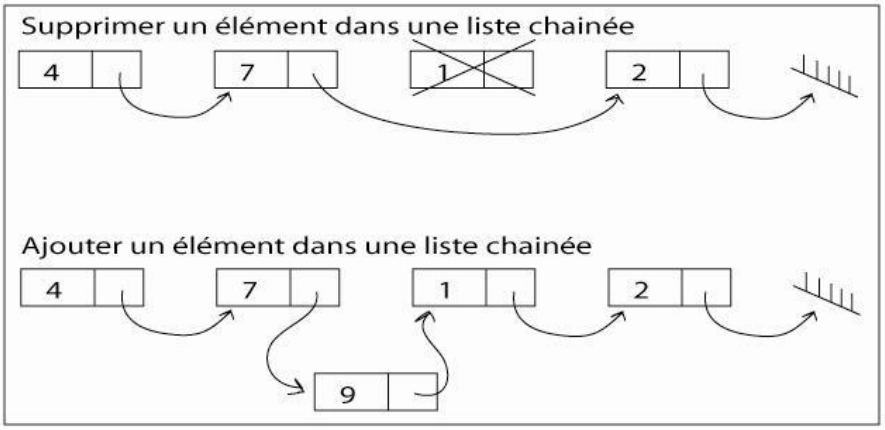
## 4/ Ajout d'un élément dans une liste chaînée

Etant donné que les éléments ne sont pas contigus en mémoire dans une liste chaînée, il est relativement aisé d'y ajouter (ou supprimer) un élément (contrairement dans un tableau) à la position souhaitée.

Il y a trois types d'ajouts :

- **L'ajout en tête de liste** : le nouvel élément pointera simplement sur l'ancienne liste.
- **L'ajout en fin de liste** : le dernier élément de l'ancienne liste pointera sur le nouvel élément qui lui-même pointera sur NULL.
- **L'ajout en milieu de liste**, décrit ci-dessous.

Voici un schéma explicatif :




Dans cet exemple, ajouter un « 9 » en troisième position revient à faire pointer le « 7 » sur le « 9 » et le « 9 » sur le « 1 » sans avoir à déplacer d'éléments ni recréer de liste chaînée.
On comprend aisément que cette structure linéaire convient tout à fait à l'ajout / suppression de tableau.

Voici un programme en Python permet l'ajout d'un élément dans une liste chaînée :

```python
class Cellule :
    def __init__(self,val,suiv) :
        self.valeur = val
        self.suivante = suiv

    def length(self) :
        long = 0
        liste = self # Liste de départ
        # Tant que la liste n'est pas vide
        while liste is not None :
            long += 1
            liste = liste.suivante

        return long

    # Affichage des valeurs de la liste
    def __str__(self) : # Permet de surcharger "print(
        liste = self
        affichage = ""
        # Tant que l'on n'est pas en fin de liste
        while liste is not None :
            affichage += str(liste.valeur)
            # Pour éviter le ";" à la fin de la liste
            if liste.suivante is not None :
                affichage += " ; "

            liste = liste.suivante
        # Affiche les éléments séparés par un " ; "
        return affichage
```

```python
# Ajout en début de liste et renvoi
def add_front(self,val) :
    nouv_cel = Cellule(val,self)
    return nouv_cel


# Ajout en fin de liste et renvoi
def add_back(self,val) :
    liste = self
    # Parcours de la liste pour trouver le dernier élément
    # Tant que l'on n'est pas en fin de liste
    while liste.suivante is not None :
        liste = liste.suivante

    # Création et ajout du dernier élément
    liste.suivante = Cellule(val,None)

    return self
```

```python
# On tient compte du fait qu'il y aura un élément de plus !
def add(self,ind,val) : # Ajout de 'val' à l'indice 'ind'
    # Contrôle de la validité de la position
    if ind < 0 or ind > self.length() :
        raise IndexError("Erreur d'indice")
    # Cas particuliers
    if ind == 0 :
        return self.add_front(val)

    if ind == self.length() :
        return self.add_back(val)

    # Récupération du nième élément (liste)
    liste = self
    num = 0
    valeur = None
    # Parcours séquentiel de la liste
    while liste is not None :
        # Si on arrive au n - 1 ième élément, on quitte la boucle
        if num == ind - 1 : # Pour récupérer l'élément précédent
            break
        num += 1
        liste = liste.suivante

    # Le nouveau élément pointe vers l'élément suivant
    nouv_cel = Cellule(val,liste.suivante)
    # L'élément suivant de liste est bien nouv_cel
    liste.suivante = nouv_cel

    return self
```

```python
# Création d'une liste chaînée : 1 ; 2 ; a ; x ; 6
lst2 = Cellule(1, Cellule(2, Cellule("a", Cellule("x", Cellule (6,None)))))
lst2 = lst2.add_front("t")
print(lst2)  # Attendu : t ; 1 ; 2 ; a ; x ; 6
lst2 = lst2.add_back(7)
print(lst2)  # Attendu : t ; 1 ; 2 ; a ; x ; 6 ; 7
lst2 = lst2.add(4,"v")
print(lst2)  # Attendu : t ; 1 ; 2 ; a ; v ; x ; 6 ; 7
```

```
t ; 1 ; 2 ; a ; x ; 6
t ; 1 ; 2 ; a ; x ; 6 ; 7
t ; 1 ; 2 ; a ; v ; x ; 6 ; 7
```

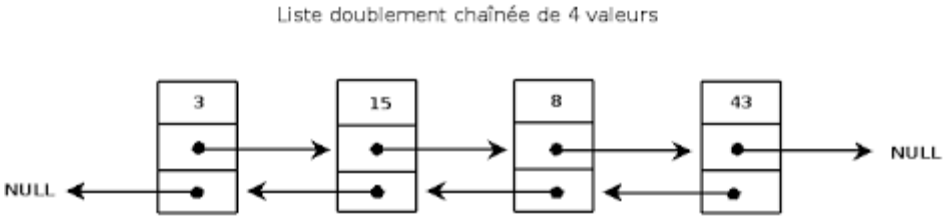
La méthode *__str__(self)* permet de redéfinir la fonction habituelle *print()* : on appelle cela une **surcharge** de fonction, très utilisée en POO (on étudiera ceci dans le chapitre POO avancée).

Remarque : dans la méthode *add(self, ind, val)*, pour contrôler la validité de l'indice passé en paramètre, on accepte bien le nombre d'éléments de la liste chaînée : ceci est logique puisqu'elle aura un élément de plus.

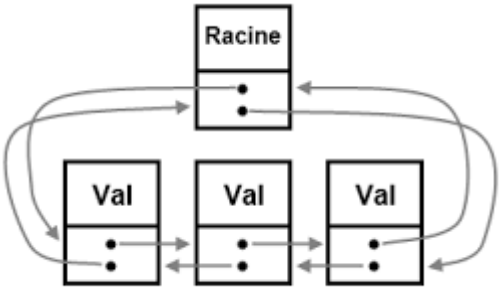
## 5/ Variantes de listes chaînées

Il existe de nombreuses variantes de structure linéaire de listes chaînées, on peut citer par exemple :

- La liste **doublement chaînée** : chaque élément est lié à son suivant mais aussi à son prédécesseur. Cette version est très utilisée en pratique car elle permet aisément de remonter la liste « à l'envers ».




- La liste **doublement chaînée cyclique** : en plus du cas précédent, le dernier élément de la liste est relié au premier (*« Racine »* dans l'exemple ci-dessous).




**A savoir : une liste chaînée est une structure de donnée linéaire, homogène ou pas, qui représente une séquence finie d'éléments. Chaque élément est contenu dans une cellule qui possède une valeur et un pointeur vers la cellule suivante. La dernière valeur est NULL (None en Python).**
**Les opérations s'effectuent sous forme de parcours qui suivent ces liaisons.**