indice 0 du tableau

Un tableau est ici traitée comme un objet statique. La position de l'indice 0 est toujours fixe.

sera organisé comme ci-dessous dans la RAM de l'ordinateur.

...	...	10	6	3	5	12		...	...
...	...	...	...	...	...	...	...	...	...

Que se passe-t-il si on insère 42 en première position avec `lst.insert(0, 42)` ? Python utilise des tableaux dynamiques dont la taille peut varier. Il va donc déplacer tous les éléments vers la droite afin de faire la place pour ajouter 42 en tête.

Python commence par ajouter un élément `None` en fin de liste avec `append`.

...	...	10	6	3	5	12	None	...	...
...	...	...	...	...	...	...	...	...	...

Puis, il décale tous les éléments vers la droite en faisant une **boucle** de taille n.

...	...	10	10	6	3	5	12	...	...
...	...	...	...	...	...	...	...	...	...

Finalement, il écrit 42 en position 0.

Une méthode plus facile aurait été de redéfinir l'indice 0 du tableau. Mais pour faire cela, la structure de liste Python n'est pas adaptée!

...	...	42	10	6	3	5	12	...	...
...	...	...	...	...	...	...	...	...	...

Rappelez vous la complexité !

Le nombre d'opérations réalisé est proportionnel au nombre d'éléments du tableau. Avec un tableau d'un million d'éléments, on pourra ainsi instantanément ajouter un élément en fin de tableau mais on devra faire un million de décalages successifs pour insérer (ou retirer) un élément en tête.

Conclusion : Pas très efficace mais peut-on faire mieux :

Liste Python		**Deque Python**	
Operation	**Average Case**	**Operation**	**Average Case**
Copy	O(n)	Copy	O(n)
Append last	O(1)	append last	O(1)
Append first	O(n)	append first	O(1)
Pop last	O(1)	pop last	O(1)
Pop / Insert first	O(n)	pop first	O(1)
extend	O(k)	extend	O(k)
extend beginning	O(n)	extend beginning	O(k)
Delete Item	O(n)	remove	O(n)
Accès à un élément	O(1)	Accès à un élément	O(1)
Source : https://wiki.python.org/moin/TimeComplexity			

Oui… Si on utilise ce deque, on peut faire mieux pour un certain nombre d'opérations. Étudions ce qui peut nous permettre d'accélérer certaines opérations.

La liste chainée est une des implémentation possible pour la liste. Une liste devient alors dynamique et chaque élément peut être stocké n'importe où dans la mémoire. L'autre alternative est le tableau statique où on bloque un grand nombre de cases mémoire

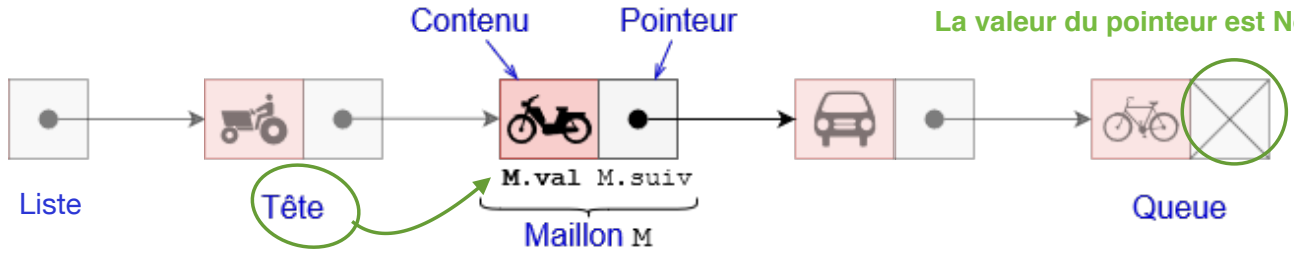
2) Définition du type abstrait : interface

Un des types fondamentaux en informatique théorique est la **liste**.
Une liste chaînée est une structure de données composées d'une séquence d'éléments de liste.
Chaque enregistrement d'une liste chaînée est souvent appelé **élément**, **nœud** ou **maillon**.

La tête d'une liste est son premier nœud. La queue d'une liste peut se référer soit au **reste de la liste** après la tête, soit au **dernier nœud** de la liste.

Chaque élément (ou maillon) **`M`** de la liste est composé :

- d'un contenu **`M.valeur`** (de n'importe quel type),
- d'un pointeur **`M.suivant`** pointant vers l'élément suivant de la séquence.



La valeur du pointeur est None.

On peut dire que M est le suivant de Tete : tete.suivant est le maillon M.

Le dernier élément de la liste possède un pointeur **`M.suivant`** vide : 

La valeur du maillon suivant la tete est accessible par tete.suivant.valeur.

> Définition : Une liste chainée L est un ensemble de maillons. Elle entièrement définie par son maillon de tête L.tete, c'est à dire l'adresse de son premier maillon.

Il n'existe pas de normalisation pour les **primitives**[1] de manipulation de listes chainées. Les plus fréquentes sont :

- `est_vide(L)` : renvoie vrai si la liste est vide
- `taille(L)` : renvoie le nombre d'éléments de la liste
- `get_dernier_maillon(L)` : renvoie le dernier élément de la liste
- `get_maillon_indice(L, i)` : renvoie le maillon d'indice i
- `ajouter_debut(L, d)` : ajoute un élément au début de la liste
- `ajouter_fin(L, d)` : ajout un maillon à la fin de la liste
- `inserer_apres(L, i, M)` : insert un maillon à l'indice i
- `supprimer_apres(L, M)` : supprime le maillon suivant le maillon M

À cela, on peut ajouter d'autres opérations appelées **auxiliaires** (non essentielles) :

- `afficher(L)` : parcourir et lire toute la liste.
- `ajouter(L1, L2)` : concaténer deux listes.

Ce type de données est formateur car fondamental en informatique. Toutefois, nous étudierons juste les opérations primitives.

[1] Les primitives sont les opérations proposées par le concepteur à l'utilisateur. Par exemple, en Python, le concepteur propose une fonction **len** pour mesurer la taille d'une séquence.

Exemple d'utilisation :

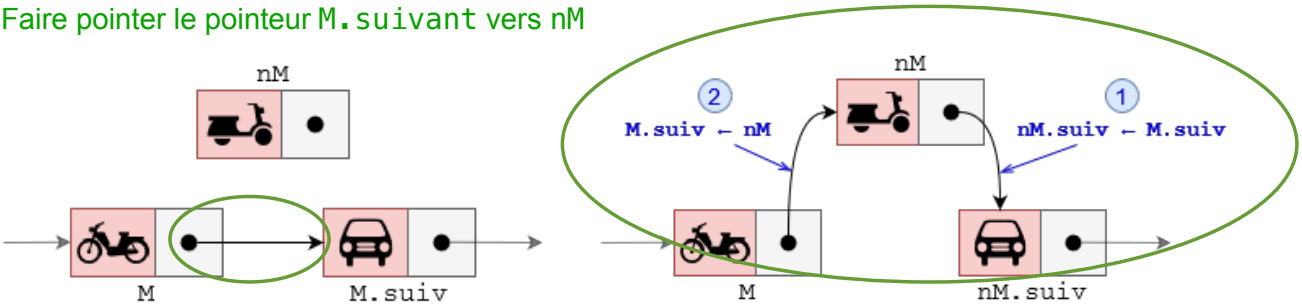
Le suivant de la vespa nM est la voiture.
La voiture est repérée comme étant le suivant de la moto M : nM.suivant = M.suivant

a. Insertion d'un maillon

Pour insérer un maillon nM après un maillon M, il faut :

① Faire pointer le pointeur `nM.suivant` vers `M.suivant`

② Faire pointer le pointeur `M.suivant` vers nM



Surtout ne pas perdre ce lien à l'étape 1 !! Sinon, la voiture sera perdue dans la mémoire.

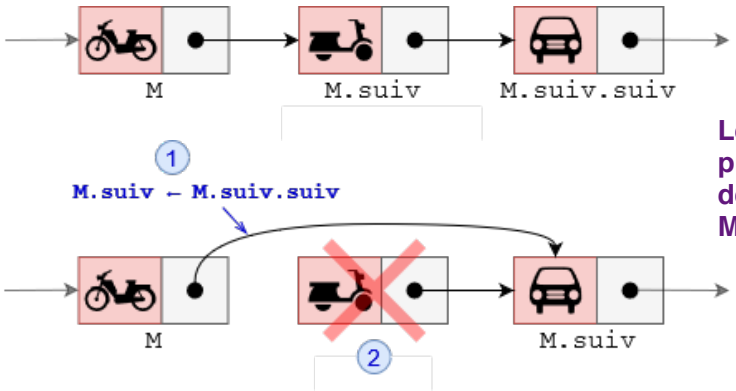
Expliquer pourquoi l'insertion en tête va plus vite dans le cas d'une liste chaînée que de la cas d'un tableau Python.

b. Suppression d'un maillon

Pour supprimer le maillon suivant un maillon M, il faut :

① Faire pointer le pointeur `M.suivant` vers `M.suivant.suivant`

② Détruire (effacer de la mémoire) le maillon `M.suivant` (…si besoin[2])



Le suivant du maillon M est à présent le suivant du suivant de M (la voiture).
M.suivant = M.suivant.suivant

A l'inverse, ici on veut perdre la vespa : rien ne va pointer vers ce maillon et il sera perdu (détruit).

Propriété :
Il est donc couteux en temps d'accéder à un élément donné dans une liste chainée. Il faut en effet lire tous les éléments jusqu'à l'élément recherché.

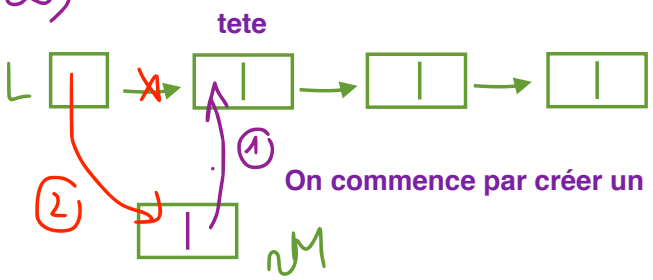
Exercice : **voire page suivante.**

Graphiquement, donnez :

a. une méthode permettant de rajouter un élément en tête de liste chainée.
b. une méthode permettant de déterminer la taille d'une liste chainée.
c. une méthode permettant de rajouter un élément à la fin de la liste chainée.

[2] Remarque : certains langages comme Python disposent d'un dispositif de ramasse-miettes. Le simple fait de ne plus être référencé détruit l'objet Maillon. Dans d'autres langages, il faut explicitement détruire l'objet : par conséquent, il important d'en garder une référence avant l'étape 1.

a)



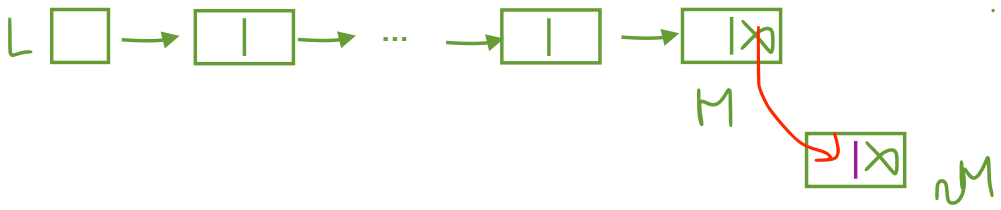
On commence par créer un pointeur vers L.tete

On fait pointer la liste vers le nouveau maillon nM. Cela supprime l'ancien lien car un pointeur ne peut pas pointer vers deux maillons.

nM.suivant = L.tete
L.tete = nM

b) Cela revient à compter le nombre de pointeurs. On fait +1 tant qu'un pointeur de Maillon ne pointe pas vers None.

c) Supposons que l'on connaisse quel est le dernier maillon :



On alloue juste le pointe du dernier vers nM. nM devient le suivant de M : M.suivant = nM

Si on ne connait pas M, c'est un peu moins efficace !! Il faut passer toute la liste en revue en prenant le suivant du suivant du suivant …. jusqu'à trouver le dernier (celui qui pointe vers None).

4) Implémentation en Python

En Python, la liste chaînée s'implémente par une classe `ListeC` dont le seul attribut est la tête :

```
class ListeC:
    def __init__(self):
        self.tete = None
```

Dans ce cours, le maillon de la liste chaînée est implémentée à l'aide d'une classe Maillon:

```
class Maillon:
   def __init__(self):
      self.valeur = None
      self.suivant = None # Pas de maillon suivant
```

Son attribut suivant est de type Maillon, ou bien vaut None si le maillon est le dernier de la liste. Pour l'instant, on n'attribue aucune valeur à un Maillon et on ne le relie à aucun autre Maillon. Cela sera réalisé grâce à des méthodes dans la classe ListeC.

Exercice :
On considère la liste d'opérations ci-dessous :

```
L = ListeC()
M1, M2 = Maillon(), Maillon()
M2.suivant = M1
M1.valeur = 'canard'
M2.valeur = 'cygne'
L.tete = M2
```

a) Dessinez la liste chainée obtenue.
b) Quelle instruction devez-vous utiliser pour afficher "cygne" ? pour afficher le Maillon M1 ? pour afficher "canard" ?
c) Quelles instructions devez-vous écrire pour ajouter un troisième maillon dont la valeur est "poule d'eau" à votre liste chainée L ?
d) Finalement, vous souhaitez supprimer le 'canard'. Comment faire ?

Exemple de fonction primitive :
On implémente la fonction `est_vide(L)` simplement de cette manière :

```
def est_vide(L):
   return L.tete is None
```

❐ — Exercice 1 —

- Implémenter en Python les fonctions taille(L) et get_dernier_maillon(L).
- Préciser la complexité de vos algorithmes.
- Modifiez vos fonctions afin de les intégrer dans la classe ListeC. On renommera taille en __len__ .

❐ — Exercice **2** —

- Implémenter en Python la primitive `get_maillon_indice(L, i)`.
- Préciser la complexité de cette fonction.
- Modifiez cette fonction afin de l'intégrer dans la classe ListeC. On renommera `get_maillon_indice` en __get_item__ .

❐ — Exercice **3** —

- Écrivez une méthode __str__(self) permettant de représenter l'ensemble d'une liste chainée sous la forme :

  L->[M0 | cygne] -> [M1 | canard] -> [M2 | poule d'eau] -> None

- En suivant les schémas des interfaces d'insertion et de suppression de maillon, implémentez en Python les fonctions `ajouter_debut(L, nM)`, `ajouter_fin(L, nM)` et `ajouter_apres(L, M, nM)`, puis `supprimer_debut(L)`, `supprimer_fin(L)` et `supprimer_apres(L, M)`. Les deux dernières fonctions doivent également renvoyer le maillon supprimé comme le ferait pop.Préciser la complexité de ces fonctions.

- Intégrez la fonction ajouter_debut et supprimer_debut à la classe ListeC. Appelez-les appendleft et popleft. Faites de même avec supprimer_fin et appelez-le pop.

❐ — Exercice **4** —

- Ré-écrivez de manière récursive les fonctions `taille(L)` et `get_dernier_maillon(L, i)`. On les appellera tailleR et get_dernier_maillonR.

II. Structures linéaires dérivées : piles et files

La nature des données ne fait pas tout. Il faut aussi s'intéresser à la manière dont on voudra les traiter : un serveur empilant des assiettes n'aura sans doute pas la même stratégie que le palettiseur avec des denrées périssables dans un supermarché.

À quelle position faire entrer nos données dans notre structure ? À quel moment devront-elles en sortir ? Lorsque cette problématique d'entrée/sortie intervient, la structure «classique» de liste doit être modifiée : c'est ainsi qu'a été introduite les structures de piles (aussi appelées **liste LIFO**) et files (**liste FIFO**).