Chap . . . : Listes chaînées

1 Implémentation d'une liste avec un tableau.

Nous avons vu que le tableau était une implémentation possible du type LISTE.

L'idée derrière le tableau est de placer les éléments les uns derrière les autres « en mémoire » de façon à trouver facilement leurs localisations par un simple calcul lié à leur numéro d'index. On parle de zones contiguës en mémoire.

```
>>> L=["a","z","e","r","t","y"]
>>> L[2]
"e"
```

Avantage : Cette implémentation permet la lecture des éléments à coût constant.

Que signifie « coût constant » noté $\mathcal{O}(1)$? . . .

Inconvénients :

— Que ce passe t-il si l'on veut insérer un nouvel élément en index 0 ? *Il va donc falloir d'abord déplacer le contenu de l'index 0 vers l'index 1 puis déplacer le contenu de l'index 1 vers l'index 2 puis . . . et enfin insérer le nouveau contenu dans l'index 0.*

— Pourquoi lors de la création d'un tableau, on est obligé d'allouer à l'avance plus d'éléments que nécessaire ?
Dans l'espace mémoire, le tableau a besoin de zones contiguës. Il faut donc anticiper l'ajout éventuel d'éléments.

Exercice n° 1

1. Soit `L=[1,2,3,4,5,6,7,8,9,10]`. On veut insérer un 0 à l'index 0. Combien d'opérations sont nécessaires pour réaliser cette insertion ?
2. Sachant que mon ordinateur a mis 0,1ms pour insérer un 0 à l'index 0.
   a. Combien de temps va t-il mettre pour insérer un 0 à l'index 0 à la liste `L=[1,2,3,...19,20]` ?
   b. Combien de temps va t-il mettre pour insérer un 0 à l'index 0 à la liste `L=[1,2,3,...999,1000]` ?
3. Quel est le coût d'une insertion dans une liste implémentée ?

Le coût est linéaire noté $\mathcal{O}(n)$, c'est à dire que si la taille de la liste double le temps mis pour insérer un élément double (dans le pire des cas).

Nous allons voir comment parvenir à créer une implémentation permettant l'insertion et la suppression des éléments de manière plus efficace c'est à dire avec avec un coût . . .

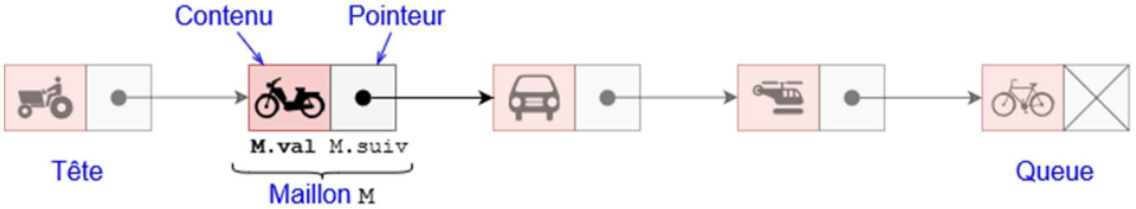
2 Liste chaînée

Une liste chaînée est une liste composée de cellules ou de maillons, comme une chaîne en métal.

Chaque maillon (ou cellule) est associé deux informations :

— Le contenu de la cellule

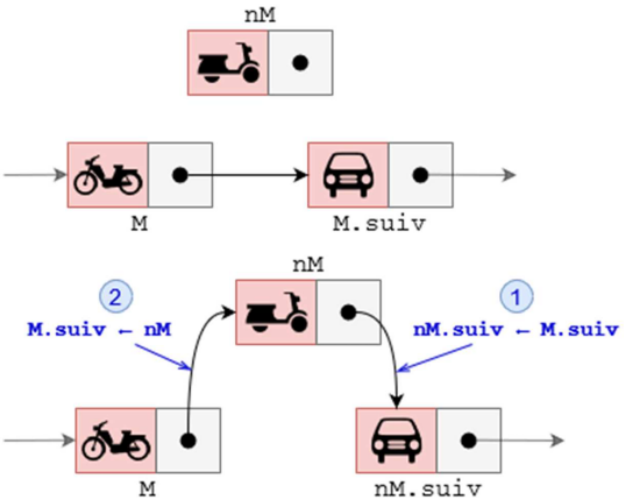
— L'adresse de la prochaine cellule (le successeur ou le pointeur)



Insertion d'un maillon :

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

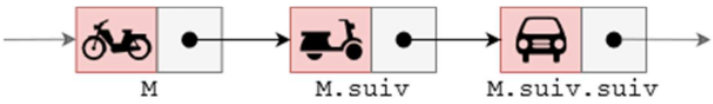
1. Faire pointer le pointeur **nM.suiv** vers **M.suiv**
2. Faire pointer le pointeur **M.suiv** vers **nM**



Suppression d'un maillon :

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

1. Faire pointer le pointeur . . . vers . . .
2. Détruire le maillon . . .



Exercice nº 2

1. Lors d'une insertion ou d'un suppression, le nombre d'opération à réaliser dépend t-il de la taille de la liste chaînées ?
2. Quelle est alors le coût pour l'insertion ou la suppression ?
3. Quel est l'inconvénient majeur d'une liste chaînée ?
   À nombre d'éléments égal, une liste chaînée occupe plus de mémoire qu'un tableau, car elle a besoin de stocker également les pointeurs « suivant ». Pour accéder à un élément d'une liste chaînée, on est obligé de parcourir tous les maillons jusqu'au maillon recherché. Le temps d'accès est donc d'autant plus grand que l'élément recherché est « loin » dans la liste, alors que dans un tableau, tous les éléments peuvent être accédés immédiatement.

3 Implémentation d'un liste chaînée

On peut implémenter un maillon de liste chaînée en Python à l'aide d'une classe **Maillon** :

```
1 class Maillon :
2         def __init__(self,valeur,suivant=None) :
3                 self.valeur = valeur
4                 self.suivant = suivant
```

Son attribut **suivant** est de type **Maillon**, ou bien vaut **None** si le maillon est le dernier de la liste.
Et une liste chaînée s'implémente par une classe **ListeC** :

```
1 class ListeC:
2    def __init__(self):
3       self.tete = None # Liste vide
```

Son attribut **tete** est de type **Maillon**, ou bien vaut **None** si la liste est vide.

```
L = ListeC()
M1 = Maillon()
M2 = Maillon()
M1.suiv = M2
L.tete = M1
```

Remarque : Une liste chaînée est entièrement déterminée par son maillon de tête. Que signifie cette remarque ?

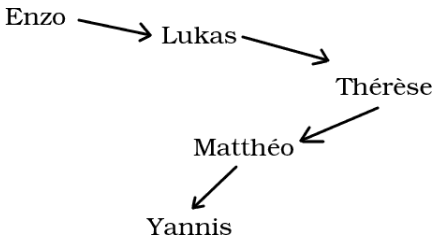
En connaissant le maillon de tête, on connaît la première valeur ainsi que le maillon suivant. Connaissant le deuxième maillon, on connaît sa valeur ainsi que le maillon suivant, . . .

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

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

Exercice n° 3

Instancier une liste chaînée : **classeNSI**, schématisée ci-dessous :



Exercice n° 4

1. Implémenter en Python les méthodes `taille(L)` et `get_dernier_maillon(L)`
2. Implémenter de façon récursive la méthode `tailleRecu(L)` qui renvoie la taille de la liste.

Exercice n° 5

Implémenter en Python la fonction `get_maillon_indice(L,i)`

Exercice n° 6

1. En suivant les schémas des interfaces d'insertion et de suppression de maillon, implémenter en Python les fonctions :
   - `ajouter_debut(L,nM)`
   - `ajouter_fin(L,nM)`
   - `ajouter_apres(L,M,nM)`
   - `supprimer_debut(L)`
   - `supprimer_fin(L)`
   - `supprimer_apres(L,M)`

   (Les méthodes de suppressions doivent retourner le maillon supprimé).
2. Pour chacune des méthodes ci-dessus, déterminer l'ordre de complexité de chacune de ces fonctions.

Exercice n° 7

Tester vos méthodes sur la liste chaînée **classeNSI** en :
- Supprimant Thérèse
- Ajoutant Gauthier à la fin
- Ajoutant Estéban au début

4 Bilan

La liste chaînée est donc une alternative. Elle présente en revanche les inconvénients suivants :

1. À nombre d'éléments égal, une liste chaînée occupe plus de mémoire qu'un tableau, car elle a besoin de stocker également les pointeurs ≪suivant≫.
2. Pour accéder à un élément d'une liste chaînée, on est obligé de parcourir tous les maillons jusqu'au maillon recherché. Le temps d'accès est donc d'autant plus grand que l'élément recherché est ≪loin≫ dans la liste, alors que dans un tableau, tous les éléments peuvent être accédés immédiatement.

Ces inconvénients sont cependant à relativiser puisque :

1. Le coût pour la suppression et l'insertion est constant.
2. Dans l'espace mémoire, le tableau a besoin de zones contiguës. Il faut donc anticiper l'ajout éventuel d'éléments. L'utilisation d'un liste chaînée permet de s'affranchir de cette contrainte.