Listes chaînées

Implémentation des listes.

Idée d'une chaîne

Pour ne pas avoir à occuper un grand bloc continu en mémoire, on a l'idée d'étaler la liste en petits morceaux qui peuvent se loger à n'importe quelle place disponible de la mémoire. Ces morceaux sont reliés les uns aux autres formant une chaîne de sorte qu'en connaissant la position du premier, on peut parcourir la chaîne de maillon en maillon jusqu'au bout.

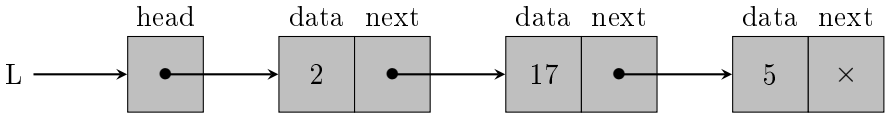
Idée d'un maillon de chaîne

Notre liste est constituée de maillons contenant chacun un item de la liste et chaîné avec le maillon suivant.

Chaque maillon a deux attributs :

- L'item, que nous appellerons `data`.
- Le maillon suivant que nous appellerons `next`.

Le maillon en bout de chaîne aura l'attribut `next` vide.



La liste a besoin de d'accéder au premier maillon de la chaîne. La liste possède donc un attribut `head` correspondant au premier maillon.
Si `head` est vide, c'est que la liste est vide.

Implémentation Python

On choisit un **dictionnaire** `m` pour chaque maillon, les clés sont :

- `m["data"]`, item contenu dans le maillon,
- `m["next"]`, maillon suivant.
  Quand `m["next"] == None`, c'est qu'on est en bout de chaîne.

De la même façon, la liste `L` est un dictionnaire dont l'unique clé est `L["head"]` contenant le premier maillon.
Quand `L["head"] == None`, c'est que la liste est vide.

Exemples de création de la liste L = 2 :17 :5

Créer les éléments d'abord et les relier ensuite.

```
L = {"head":None}
m0 = {"data":2, "next":None}
m1 = {"data":17, "next":None}
m2 = {"data":5, "next":None}
L["head"] = m0
m0["next"] = m1
m1["next"] = m2
```

Construire la liste à l'envers.

```
m2 = {"data":5, "next":None}
m1 = {"data":17, "next":m2}
m0 = {"data":2, "next":m1}
L = {"head":m0}
```

Ou encore à l'endroit mais sans variables supplémentaires.

```
L = {"head":None}
L["head"] = {"data":2, "next":None}
L["head"]["next"] = {"data":17, "next":None}
L["head"]["next"]["next"] = {"data":5, "next":None}
```

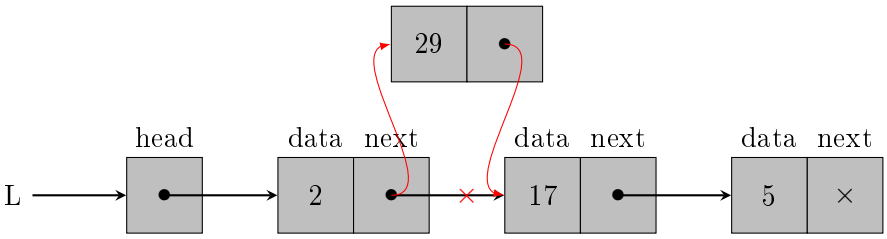
Notion de pointeur

```
m0 = {"data":2, "next":None}
m1 = {"data":17, "next":None}
m0["next"] = m1
```

Dans ce code, `m0` et `m1` ne sont pas les maillons eux-mêmes mais des **références** aux maillons, des adresses mémoires, des **pointeurs**. C'est ce qu'on a représenté par des flèches. En écrivant `m0["next"] = m1`, on n'agrandit donc pas la taille de `m0`, puisque `m1` est une simple adresse, un simple nombre dont la taille en mémoire est connue d'avance et déjà réservée au moment de créer `m0`.

Insérer un élément

Partant de `L = 2:17:5`, on exécute `INSERER(L, 29, 1)`.



Il faut, dans cet ordre :

i. créer le nouveau maillon contenant 29, appelons le `new_m`,
`prec_m = {"data":29, "next":None}`

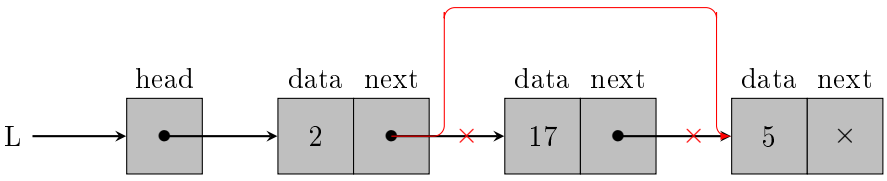
ii. trouver le point d'insertion : soit `prec_m` le maillon juste avant l'insertion, ici celui contenant 2,

iii. brancher `new_m["next"]` sur le maillon suivant, celui contenant 17, c'est à dire `prec_m["next"]` :
`new_m["next"] = prec_m["next"]`

iv. brancher le maillon précédent sur le nouveau, c'est à dire `prec_m["next"] = new_m`.

Supprimer un élément

Partant de `L = 2:17:5`, on exécute `SUPPRIMER(L, 1)`.



Il faut, dans cet ordre :

i. trouver le maillon à supprimer, appelons le `del_m`,
et celui immédiatement précédent, `prec_m`,

ii. brancher le maillon précédent sur le maillon suivant celui à supprimer :
`prec_m["next"] = del_m["next"]`

iii. débrancher la sortie du maillon à supprimer pour éviter tout problème :
`del_m["next"] = None`