Les listes chaînées

Résumé

Les listes Python que l'on connaît (les `list`) stockent leurs éléments côte à côte en mémoire, comme des cases numérotées. C'est pratique pour accéder directement au k-ième élément, mais insérer ou supprimer un élément au milieu oblige à décaler tous les suivants. La **liste chaînée** adopte une approche différente : chaque élément (appelé maillon) contient sa valeur et un lien vers le maillon suivant, comme les wagons d'un train reliés par des attelages. Insérer ou supprimer un maillon ne demande que de modifier un lien, sans toucher au reste. Comprendre cette structure, c'est comprendre que le choix de la bonne représentation des données dépend des opérations qu'on veut effectuer.

Prérequis : Programmation orientée objet, La récursivité.

① Mémo

Définition 1 (Principe d'une liste chaînée)

Une liste chaînée est une suite de **maillons** (ou nœuds). Chaque maillon contient :

- une **valeur** (la donnée) ;
- une **référence** vers le maillon suivant (`None` pour le dernier).

On accède à la liste par sa **tête** (premier maillon). Il n'y a pas d'accès direct par indice : pour atteindre le k-ième élément, il faut parcourir les k premiers maillons.

Propriété 1 (Comparaison avec les tableaux (listes Python))

Accès par indice Tableau : $O(1)$. Liste chaînée : $O(n)$.
Insertion en tête Tableau : $O(n)$ (décalage). Liste chaînée : $O(1)$.
Insertion en fin Tableau : $O(1)$ amorti. Liste chaînée : $O(n)$ (parcours).
Suppression en tête Tableau : $O(n)$. Liste chaînée : $O(1)$.
Mémoire Tableau : contiguë. Liste chaînée : maillons dispersés en mémoire.

Implémentation en Python

```python
class Maillon:
    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

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

La liste vide correspond à `tete = None`.

⚠ Pièges fréquents

- Oublier de mettre à jour la tête après une insertion ou suppression en tête.
- Parcourir la liste sans vérifier que le maillon courant n'est pas `None`.
- Perdre la référence vers un maillon lors d'une insertion (toujours chaîner le nouveau maillon *avant* de modifier les pointeurs existants).

⚠ Erreurs classiques

Code erroné	Code correct	Explication
Insérer un maillon en modifiant `tete` *avant* de chaîner	`nouveau.suivant = self.tete` `self.tete = nouveau`	Si l'on écrit `self.tete = nouveau` d'abord, l'ancien premier maillon est perdu : le chaînage est rompu.
`courant.suivant.valeur` sans vérifier `None`	`if courant.suivant is not None:`	Accéder à `.valeur` ou `.suivant` sur `None` provoque un `AttributeError`. Toujours vérifier avant de déréférencer.
Supprimer en tête sans mettre à jour `self.tete`	`self.tete = self.tete.suivant`	Si l'on supprime le premier maillon sans modifier `tete`, la liste pointe toujours vers l'ancien maillon : la suppression est ineffective.
`courant.suivant = courant` (référence circulaire)	`courant.suivant = nouveau` (le bon maillon)	Faire pointer un maillon vers lui-même crée une boucle infinie : tout parcours avec `while courant is not None` ne s'arrête jamais.

② Exemples

Programme 1 · Classe ListeChainee complète

```
class Maillon:
    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

class ListeChainee:
    def __init__(self):
        self.tete = None

    def est_vide(self):
        return self.tete is None

    def inserer_en_tete(self, valeur):
        nouveau = Maillon(valeur, self.tete)
        self.tete = nouveau

    def supprimer_en_tete(self):
        if self.est_vide():
            raise IndexError("Liste vide")
        val = self.tete.valeur
        self.tete = self.tete.suivant
        return val

    def longueur(self):
        compteur = 0
        courant = self.tete
        while courant is not None:
            compteur += 1
            courant = courant.suivant
        return compteur

    def rechercher(self, valeur):
        courant = self.tete
        while courant is not None:
            if courant.valeur == valeur:
                return True
            courant = courant.suivant
        return False

    def __str__(self):
        elements = []
        courant = self.tete
        while courant is not None:
            elements.append(str(courant.valeur))
            courant = courant.suivant
        return " -> ".join(elements) + " -> None"
```

```python

# Utilisation
L = ListeChainee()
L.inserer_en_tete(3)
L.inserer_en_tete(7)
L.inserer_en_tete(1)
print(L)              # 1 -> 7 -> 3 -> None
print(L.longueur())   # 3
print(L.rechercher(7))  # True
L.supprimer_en_tete()
print(L)              # 7 -> 3 -> None
```

③ Exercices

Exercice 1 *Opérations de base* Ajouter à la classe `ListeChainee` :

1. Une méthode `ajouter_en_fin(valeur)` qui insère un élément à la fin de la liste ;
2. Une méthode `ieme(i)` qui renvoie la valeur du i-ième maillon (en partant de 0) ;
3. Une méthode `supprimer(valeur)` qui supprime la première occurrence d'une valeur donnée.

Exercice 2 *Fonctions récursives sur les listes chaînées* En considérant une liste chaînée représentée simplement par des maillons (sans la classe enveloppe), écrire des fonctions récursives :

1. `longueur_rec(m)` qui renvoie la longueur à partir du maillon `m`.
2. `contient_rec(m, x)` qui teste si la valeur `x` est dans la liste.
3. `renverser_rec(m)` qui renverse la liste chaînée et renvoie la nouvelle tête.

On considère que `None` représente la liste vide.

Exercice 3 *Pile avec une liste chaînée* Implémenter une pile en utilisant une liste chaînée au lieu d'une liste Python. Les opérations `empiler`, `depiler`, `sommet` et `est_vide` doivent toutes être en O(1).

Solutions des exercices

Corrigé de l'exercice 1.

```
def ajouter_en_fin(self, valeur):
    nouveau = Maillon(valeur)
    if self.est_vide():
        self.tete = nouveau
    else:
        courant = self.tete
        while courant.suivant is not None:
            courant = courant.suivant
        courant.suivant = nouveau

def ieme(self, i):
    courant = self.tete
    for _ in range(i):
        if courant is None:
            raise IndexError("Indice hors limites")
        courant = courant.suivant
    if courant is None:
        raise IndexError("Indice hors limites")
    return courant.valeur

def supprimer(self, valeur):
    if self.est_vide():
        return
    if self.tete.valeur == valeur:
        self.tete = self.tete.suivant
        return
    courant = self.tete
    while courant.suivant is not None:
        if courant.suivant.valeur == valeur:
            courant.suivant = courant.suivant.suivant
            return
        courant = courant.suivant
```

Complexité : `ajouter_en_fin` : $O(n)$. `ieme` : $O(i)$. `supprimer` : $O(n)$ dans le pire cas.

Point clé de `supprimer` : on traite la suppression en tête séparément car il faut modifier `self.tete`. Pour les autres cas, on cherche le maillon *précédent* celui à supprimer.

Corrigé de l'exercice 2.

```python
def longueur_rec(m):
    if m is None:
        return 0
    return 1 + longueur_rec(m.suivant)

def contient_rec(m, x):
    if m is None:
        return False
    if m.valeur == x:
        return True
    return contient_rec(m.suivant, x)

def renverser_rec(m):
    if m is None or m.suivant is None:
        return m
    reste_inverse = renverser_rec(m.suivant)
    m.suivant.suivant = m
    m.suivant = None
    return reste_inverse
```

Remarque sur `renverser_rec` : cette version modifie la liste d'origine (en place). Pour créer une nouvelle liste sans modifier l'ancienne :

```python
def renverser_copie(m, acc=None):
    if m is None:
        return acc
    return renverser_copie(m.suivant, Maillon(m.valeur, acc))
```

Corrigé de l'exercice 3.

```python
class PileChainee:
    def __init__(self):
        self._tete = None

    def est_vide(self):
        return self._tete is None

    def empiler(self, x):
        self._tete = Maillon(x, self._tete)

    def depiler(self):
        if self.est_vide():
            raise IndexError("Pile vide")
        val = self._tete.valeur
        self._tete = self._tete.suivant
        return val

    def sommet(self):
```

```
        if self.est_vide():
            raise IndexError("Pile vide")
        return self._tete.valeur
```

Principe : le sommet de la pile est la tête de la liste chaînée. Empiler revient à insérer en tête (O(1)), dépiler revient à supprimer en tête (O(1)).

Chaque opération est en O(1) strict, sans coût ponctuel de redimensionnement ; en pratique, la `list` Python reste toutefois plus rapide.