

Chap 05. Listes chaînées

Livre p 107 – Chap 6 Listes chaînées

1. Une structure abstraite : les listes chaînées

a. Notion de liste chaînée

En python comme dans d'autres langages, il existe des listes ou tableaux (type list) permettant de stocker des valeurs, puis les récupérer grâce à leur indice. La gestion de ces listes est propre à chaque langage. Il s'agit ici, d'étudier un modèle de listes chaînée, indépendant de tout langage de programmation.

- Une **cellule** (ou maillon) : C'est un objet qui contient deux informations, une valeur et un pointeur qui dirige soit vers la cellule suivante soit vers le vide (fin).
- Une **liste chaînée** : C'est un objet qui contient une tête composée d'une cellule (ou vide), et de la succession des cellules liées par leurs pointeurs.

Remarque : il existe un modèle plus élaboré de listes doublement chaînées où chaque maillon contient une troisième information : l'adresse du maillon précédent.

Nous n'étudierons ici que les listes chaînées non cycliques...

b. Opérations élémentaires sur les listes chaînées

Les opérations élémentaires sur les listes chaînées sont :

- La **création** d'une liste chaînée vide.
- La **vérification** qu'une liste chaînée est vide ou non.
- La **lecture** du contenu de la cellule en tête d'une liste chaînée.
- L'**ajout** d'une cellule en tête d'une liste chaînée.
- La **suppression** d'une cellule en tête d'une liste chaînée.
- Le **parcours** des cellules d'une liste chaînée.

c. Opérations élaborées sur les listes chaînées

En utilisant les opérations élémentaires, on peut créer d'autres opérations plus élaborées, par exemple :

- Le calcul de la longueur d'une liste chaînée.
- La lecture du contenu de la $n^{\text{ème}}$ cellule d'une liste chaînée.
- L'insertion d'une cellule à un rang quelconque d'une liste chaînée.
- La suppression d'une cellule à un rang quelconque d'une liste chaînée.
- La concaténation de deux listes chaînées.
- Le renversement d'une liste chaînée.

2. Implémentation en Python

a. Deux nouveaux objets

En python, on peut créer les objets **Cellule** et **Liste** :


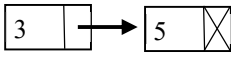
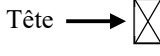
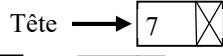
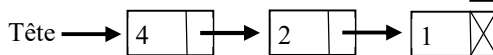
```
class Liste:
    """une liste chaînée"""
    def __init__(self):
        self.tete = None

    def __repr__(self):
        if self.tete is None:
            return "[]"
        else:
            return "[{}].format(self.tete)
```

```
class Cellule:
    """une cellule d'une liste chaînée"""
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s

    def __repr__(self):
        if self.suivante is None:
            return "{}".format(self.valeur)
        else:
            return "{}{}".format(self.valeur, self.suivante)
```

Exemples

- Cellule isolée :  (12)
- Cellules liées :  (3)(5)
- Liste chaînée vide :  []
- Liste chaînée d'une unique cellule :  [(7)]
- Liste chaînée de 3 cellules :  [(4)(2)(1)]

b. Fonctions et méthodes sur ces nouveaux objets

Les opérations élémentaires sur les cellules et les listes chaînées peuvent être définies ainsi :

- **Lecture** de la valeur d'une cellule.
Exemple : cel.valeur
- **Récupération** de la cellule pointée par une cellule.
Exemple : cel.suivante
- **Création** d'une liste chaînée vide.
Exemple : lst = Liste()
- **Vérification** qu'une liste chaînée est vide ou non.
Exemple : lst.est_vide()
- **Récupération** de la cellule en tête d'une liste.
Exemple : lst.tete
- **Ajout** d'une valeur en tête d'une liste chaînée.
Exemple : lst.ajouter_en_tete(val)
- **Suppression** d'une cellule en tête d'une liste chaînée.
Exemple : lst.supprimer_en_tete()

```
class Liste:
    (...)

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

    def ajouter_en_tete(self, x):
        self.tete = Cellule(x, self.tete)

    def supprimer_en_tete(self):
        if not(self.est_vide()):
            self.tete = self.tete.suivante
```

Exemples d'opérations plus élaborées :

- Calcul de la **longueur** d'une liste chaînée.
Exemple : lst.longueur() ou len(lst)
- Lecture du **contenu** de la n-1^{ème} cellule d'une liste.
Exemple : lst.nieme_element(n-1) ou lst[n-1]

Versions récursives sur les cellules de tête

```
class Liste:
    (...)

    def __len__(self):
        return longueur(self.tete)

    def __getitem__(self, n):
        return nieme_element(n, self.tete)

def longueur(cel):
    if cel is None :
        return 0
    else :
        return 1 + longueur(cel.suivante)

def nieme_element(n, cel):
    """NB : numérotation à partir de 0"""
    if cel.suivante is None:
        raise IndexError("indice invalide")
    if n == 0 :
        return cel.valeur
    else :
        return nieme_element(n-1, cel.suivante)
```

Versions itératives sur les listes

```
class Liste:
    (...)

    def __len__(self):
        return longueur(self)

    def __getitem__(self, n):
        return nieme_element(n, self)

def longueur(lst):
    cpt = 0
    cel = lst.tete
    while not(cel is None) :
        cpt += 1
        cel = cel.suivante
    return cpt

def nieme_element(n, lst):
    """NB : numérotation à partir de 0"""
    cel = lst.tete
    for i in range(n) :
        if cel.suivante is None:
            raise IndexError("indice invalide")
        else :
            cel = cel.suivante
    return cel.valeur
```