Listes chaînées

Activité – Terminale NSI

① Un train de wagons

Exercice 1 *Une autre façon d'organiser des données*

Imaginons un train. Chaque *wagon* contient une *cargaison* et est attaché au *wagon suivant* (ou à rien, s'il s'agit du dernier wagon). On accède au train par sa *locomotive*.

1. Combien d'éléments un wagon contient-il en propre ? (En dehors de sa cargaison.)
2. Pour parcourir tous les wagons d'un train, quelle stratégie naturelle adopte-t-on ?
3. Pour ajouter un wagon en **tête** du train (juste après la locomotive), que faut-il modifier ? Et pour ajouter un wagon en **queue** ?
4. Comment représenter un train *vide* (sans aucun wagon) ?

Remarque 1. Une **liste chaînée** est exactement cela : une suite de *cellules*, chaque cellule contenant une valeur et un lien vers la cellule suivante. Une liste vide est représentée par `None`. C'est une structure **récursive** : le « suivant » d'une cellule est lui-même une liste chaînée (éventuellement vide).

② La classe `Cellule`

Définition 1 **(Représentation d'une liste chaînée)** Une cellule contient deux attributs : la `valeur` qu'elle stocke, et la référence vers la cellule `suivante` (ou `None` s'il n'y en a pas).

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

Une liste chaînée est représentée par sa *première cellule* (la « tête »). Une liste vide est `None`.

Exercice 2 *Construire une liste chaînée à la main*

On veut représenter la liste chaînée $[3 \to 7 \to 1]$.

1. Compléter le code Python qui construit cette liste :

```
c3 = Cellule(1, ............)
c2 = Cellule(7, ............)
tete = Cellule(3, ............)
```

2. Quelle est la valeur stockée dans `tete.valeur` ? Dans `tete.suivant.valeur` ? Dans `tete.suivant.suivant.valeur` ? Dans `tete.suivant.suivant.suivant` ?
3. Comment représenter la liste vide ?

③ Parcourir une liste chaînée

Exercice 3 *Longueur, deux versions*

1. **Version itérative.** Compléter la fonction `longueur_iter` qui renvoie le nombre de cellules de la liste, en parcourant les cellules avec une boucle `while` :

```
def longueur_iter(tete):
    n = 0
    courant = .............
    while courant is not None:
        n = n + 1
        courant = .............
    return n
```

2. **Version récursive.** Compléter la fonction `longueur_rec` :

```
def longueur_rec(tete):
    if tete is None:
        return .............
    return ............. + longueur_rec(.............)
```

3. Tester les deux versions sur la liste $[3 \to 7 \to 1]$ et sur la liste vide.

Exercice 4 *Recherche d'une valeur*

Compléter la fonction `appartient(tete, x)` qui renvoie `True` si `x` apparaît dans la liste chaînée, `False` sinon.

```
def appartient(tete, x):
    courant = .............
    while courant is not None:
        if courant.valeur == .............:
            return True
        courant = .............
    return False
```

Tester avec la liste $[3 \to 7 \to 1]$ et les valeurs 7, 4, 3.

④ Ajout et suppression

Exercice 5 *Ajouter en tête, supprimer en tête*

Ce sont les deux opérations les plus rapides sur une liste chaînée : $O(1)$.

1. Compléter la fonction `ajouter_en_tete(tete, x)` qui renvoie une nouvelle tête, avec `x` ajouté en tête :

```python
def ajouter_en_tete(tete, x):
    nouvelle_cellule = Cellule(....., .....)
    return nouvelle_cellule
```

2. Compléter la fonction `supprimer_en_tete(tete)` qui renvoie la liste sans son premier élément :

```python
def supprimer_en_tete(tete):
    assert tete is not None, "Liste vide"
    return .....
```

3. Pourquoi ces deux opérations sont-elles en temps constant ? Pourquoi l'ajout *en queue* est-il en $O(n)$?

Exercice 6 *Conversion liste chaînée ↔ liste Python*

1. Compléter la fonction `vers_liste(tete)` qui transforme une liste chaînée en liste Python :

```python
def vers_liste(tete):
    resultat = []
    courant = tete
    while courant is not None:
        resultat.append(.....)
        courant = .....
    return resultat
```

2. Compléter la fonction `depuis_liste(L)` qui crée une liste chaînée à partir d'une liste Python :

```python
def depuis_liste(L):
    tete = .....
    for x in reversed(L):
        tete = ajouter_en_tete(tete, x)
    return tete
```

3. Pourquoi parcourt-on `L` *à l'envers* ?

5 Liste chaînée ou liste Python ?

Exercice 7 *Coûts comparés*

Pour chaque opération, indiquer le coût (en O) sur une liste Python et sur une liste chaînée. On note n la taille de la collection.

Opération	Liste Python	Liste chaînée
Accès à l'élément d'indice i	$O(1)$	.....
Ajout en tête	.....	.....
Ajout en queue	.....	.....
Suppression en tête	.....	.....
Recherche d'une valeur	$O(n)$	.....

1. Compléter le tableau.

2. En pratique, dans quels cas une liste chaînée est-elle préférable à une liste Python ? Et l'inverse ?

⑥ Ce qu'il faut retenir

Propriété 1 **(Définition récursive)** Une liste chaînée est *soit* la liste vide (`None`), *soit* une cellule composée d'une valeur et d'une référence vers une autre liste chaînée. Cette définition récursive se traduit naturellement par une classe `Cellule` avec deux attributs (`valeur` et `suivant`).

Propriété 2 **(Schéma de parcours)** Un parcours typique commence par `courant = tete`, puis avance avec `courant = courant.suivant` tant que `courant is not None`. Toutes les fonctions classiques (longueur, recherche, conversion) suivent ce schéma.

Propriété 3 **(Coûts caractéristiques)** Sur une liste chaînée :

— ajout/suppression *en tête* : $O(1)$;

— ajout/suppression *en queue* ou en milieu : $O(n)$;

— accès indexé : $O(n)$ (pas d'accès direct).

Remarque 2. La liste chaînée est la structure de données idéale pour implémenter une **pile** (LIFO) : les opérations `empiler` et `depiler` se font toutes deux en tête, donc en $O(1)$. Pour une **file**, il suffit d'avoir aussi un pointeur vers la queue : c'est la *liste doublement chaînée*, qui sert de base à `collections.deque` en Python.