Listes chaînées

Structures de données

Objectifs.
Prérequis : programmation orientée objet, notion de référence (pointeur) en Python
À l'issue de ce chapitre, vous saurez :

- distinguer le type abstrait *liste* de ses implémentations concrètes (tableau dynamique, liste chaînée) ;
- implémenter une liste chaînée en Python à l'aide de deux classes (`Noeud` et `ListeChainee`) ;
- réaliser les opérations d'ajout, de suppression et d'accès en tête, en queue et en position quelconque ;
- comparer la complexité de ces opérations avec celles du type `list` de Python.

Introduction

On considère le type abstrait *liste* au sens d'une collection (finie) et ordonnée d'éléments. On munit ce type des opérations suivantes :

- création d'une liste vide ;
- ajout d'un élément en tête / en queue / en position i ;
- accès à l'élément en tête / en queue / au i-ième élément ;
- modification d'un élément en tête / en queue / du i-ième élément ;
- suppression en tête / en queue / en position i ;
- longueur ;
- concaténation de deux listes.

Certaines opérations peuvent modifier une liste. Les listes sont présentées ici dans un contexte de programmation impérative. Ces opérations ne sont pas toutes élémentaires.

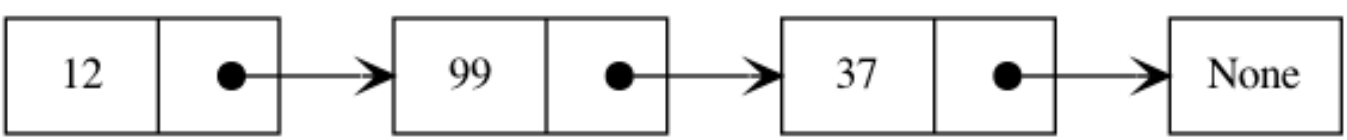
Une implémentation existante de ce type abstrait est le type prédéfini `list` de Python qui utilise des tableaux dynamiques. Un **tableau dynamique** range ses éléments dans des cases voisines de la mémoire, ce qui permet d'accéder directement à la case d'indice i ; quand il est plein, Python le recopie dans un tableau plus grand.

On s'intéresse ici à une implémentation utilisant des listes chaînées.

Une implémentation objet des listes chaînées

L'objectif ici est de réimplanter de manière élémentaire le type *liste* par des listes simplement chaînées en utilisant la programmation orientée objet. Pour cela, on doit choisir de définir un ensemble de méthodes élémentaires, et redéfinir les autres en fonction des méthodes élémentaires choisies.

On choisit une représentation non contiguë des listes, avec des *noeuds* comportant chacun un élément et une référence au suivant. C'est donc une structure de données récursive.



Remarque : ces choix sont différents des listes contiguës - dans un tableau par exemple - où l'accès à un élément peut se faire de manière directe. Avec une liste chainée, il faut parcourir de noeud en noeud.

La classe `Noeud`

L'idée des listes chaînées est d'utiliser des noeuds reliés les uns aux autres. Pour pouvoir constituer une liste, il suffit d'avoir des noeuds comportant chacun un élément et un lien vers le noeud suivant.

On crée un noeud en fournissant un élément et éventuellement une référence vers le suivant.

```
class Noeud:
    def __init__(self, element, suivant=None):
        self.element = element
        self.suivant = suivant

ma_chaine = Noeud(1, Noeud(2, Noeud(3)))

print(f"premier élément : {ma_chaine.element}")
print(f"deuxième élément : {ma_chaine.suivant.element}")
print(f"troisième élément : {ma_chaine.suivant.suivant.element}")
```

On peut ainsi créer une chaine et accéder à ses éléments en consultant l'élément ou l'élément du suivant...

On peut aussi définir une fonction qui calcule la longueur d'une chaîne :

```
def longueur(chaine):
    res = 0
    courant = chaine
    while courant:
        courant = courant.suivant
        res += 1
    return res
```

Activité. Écrire une fonction qui affiche successivement tous les éléments d'une chaîne.

```
def affiche(chaine):
    """Affiche tous les éléments de la chaine"""
    pass

affiche(ma_chaine)
```

Remarque. Si on essaie d'implémenter une des méthodes qui peut renvoyer une liste vide, on s'aperçoit que la classe `Noeud` ne peut pas suffire : en effet rien n'est encore prévu pour représenter une liste vide. Une liste à un élément a un seul noeud qui contient la valeur de l'élement et `None` comme référence au suivant puisqu'il n'y en a pas. Pour une liste sans élément, il faudrait choisir, par exemple la valeur `None` pour la liste, sans aucun noeud.

On a besoin pour cela de définir une nouvelle classe `ListeChainee`.

La classe `ListeChainee`

Opérations

On veut pouvoir implémenter toutes les opérations définies sur le type abstrait *liste*, avec en particulier une méthode pour créer une liste vide, des méthodes pour supprimer un élément, même s'il n'y en a plus qu'un... on doit donc tenir compte de la possibilité d'une liste vide.

Attributs

On choisit de définir un seul attribut `premier`, qui peut être soit une référence vers le premier `Noeud`, soit la valeur particulière `None` pour représenter une liste vide.

Méthodes

Toutes les méthodes sont définies dans la classe `ListeChainee` qui utilise en cas de besoin la classe `Noeud` pour créer des maillons de chaîne.

Implémentation

La méthode d'initialisation `__init__` crée une liste vide.

La fonction `longueur` peut maintenant être définie comme la méthode spéciale `__len__` de la classe.

La méthode `ajout_en_tete` permet d'ajouter un élement en première position.

La méthode `supprime_en_tete` enlève un élément en tête et renvoie cet élément.

La méthode `ieme_element` permet d'accéder à l'élément d'indice `i`, qui doit être compris entre `0` et `len - 1`.

```
class ListeChainee:
    def __init__(self):
        """Initialise une liste vide."""
        self.premier = None

    def __len__(self):
        """Renvoie le nombre d'éléments présents dans la liste."""
        courant = self.premier
        cpt = 0
        while courant:
            courant = courant.suivant
            cpt += 1
        return cpt

    def ajout_en_tete(self, element):
        """Insère element en tête de liste en créant un nouveau noeud."""
        nouveau = Noeud(element)
        nouveau.suivant = self.premier
        self.premier = nouveau

    def supprime_en_tete(self):
        """Supprime l'élément en tête et retourne ce dernier"""
        element = self.premier.element
        self.premier = self.premier.suivant
        return element

```

```
    def ieme_element(self, i):
        """Renvoie le ième élément de la liste."""
        courant = self.premier
        cpt = 0
        while cpt < i:
            courant = courant.suivant
            cpt += 1
        return courant.element
```

Pour les ajouts et suppressions en tête, on a pris soin de bien mettre à jour les références, pour que l'attribut `premier` de la liste désigne bien le premier noeud.

On peut tester ce début d'implémentation en construisant et accédant à une liste.

```
une_chaine = ListeChainee()
une_chaine.ajout_en_tete(4)
une_chaine.ajout_en_tete(2)
une_chaine.ajout_en_tete(6)
une_chaine.supprime_en_tete()
une_chaine.ajout_en_tete(5)
for i in range (len(une_chaine)):
    print(une_chaine.ieme_element(i))
```

Remarque. Le choix de représenter les listes par des listes chainées à partir du premier rend relativement simples les opérations en tête de liste. Accéder au i-ième élément nécessite de parcourir la liste de noeud en noeud.

Pour effectuer un ajout ou suppression en queue - ou ailleurs dans la liste - il faut la parcourir jusqu'à la bonne position en prenant soin de retenir à chaque fois deux noeuds, le précédent et le courant.

Activité. Ajouter dans la classe `ListeChainee` la méthode d'ajout en queue suivante.

```
def ajout_en_queue(self, element):
    """Insère element en queue de liste"""
    if self.premier == None:
        self.ajout_en_tete(element)
    else:
        courant = self.premier
        while courant.suivant != None:
            courant = courant.suivant
        dernier = Noeud(element)
        courant.suivant = dernier
```

S'en inspirer pour écrire une nouvelle méthode :

```
def ajout_ieme_position(self, element, i):
    """Insère element en position i """
```

Activité. Ajouter dans la classe `ListeChainee` la méthode de suppression en queue suivante.

```
def supprime_en_queue(self):
    """Supprime un element en queue de liste"""
    if self.premier.suivant == None:
```

```
        return self.supprime_en_tete()
    else:
        precedent = self.premier
        courant = self.premier.suivant
        while courant.suivant != None:
            precedent = courant
            courant = courant.suivant
        element = courant.element
        precedent.suivant = None
        return(element)
```

S'en inspirer pour écrire une nouvelle méthode :

```
def supprime_ieme_position(self, i):
    """Supprime élément en position i """
```

Activité. Ajouter à la classe `ListeChainee` les méthodes suivantes. On suppose que les positions dans une liste sont comptées à partir de 0.

- méthode `contient(self, element)` qui renvoie `True` si et seulement si `element` apparaît dans la liste ;
- méthode `modifie_ieme_element(self, i, elem)` qui remplace la valeur contenue à la position `i` dans la liste par `elem` ;
- méthode `concatene(self, autre)` qui ajoute à la fin de la liste `self` tous les éléments de la liste `autre`.

Bilan

La classe `ListeChainee` permet d'implémenter complètement le type abstrait *liste* avec toutes les fonctions d'accès et de modification possibles. On approche ainsi en terme de fonctionnalités, le type `list` standard de Python. Il y a cependant des différences importantes en terme de complexité algorithmique des opérations.

Dans la classe `ListeChainee`, seules les opérations en tête de liste sont en temps constant. Toutes les opérations nécessitant un parcours de la liste ont une complexité linéaire en fonction de la taille de la liste.

Cela permet de savoir dans quel contexte privilégier ou limiter l'usage de cette classe.

À retenir.

- Une **liste chaînée** est une structure non contiguë où chaque nœud contient une valeur et une référence vers le suivant.
- Les opérations **en tête** (ajout, suppression) s'effectuent en $O(1)$; les opérations **en position quelconque** nécessitent un parcours en $O(n)$.
- Contrairement au `list` de Python (tableau dynamique, accès direct en $O(1)$), la liste chaînée n'offre pas d'accès indexé rapide, mais excelle pour les insertions et suppressions fréquentes en tête.
- L'implémentation repose sur deux classes : `Noeud` (maillon) et `ListeChainee` (gestion de la liste, y compris le cas vide).
- Les piles et files réutilisent cette structure comme brique de base.