TD13. Listes doublement chaînées

Bases de la Programmation Impérative – Chapitre 3 : references
version du 7 juin 2024

L'objectif de ce TD est de créer une liste doublement chaînée et de réfléchir à l'implémentation et à la complexité temporelle de différentes opérations sur ce type de listes chaînées.

Exercice 1 : listes doublement chaînées

On travaille donc aujourd'hui sur les listes doublement chaînées. Dans une liste doublement chaînée, chaque cellule contient deux références : une vers la cellule **suivante** et une vers la cellule **précédente**.

L'image ci dessous est la liste doublement chaînée `3 --> 2 --> 1` dessinée par le module `traceur` à l'aide de l'option `deeply=False` pour simplifier le dessin. Cette option permet de ne pas dessiner les instances "primitives" (entiers, flottants, chaînes de caractères et `None`) comme des objets à part entière mais de les imbriquer dans les instances les contenant. Ceci **n'est pas la réalité** de ce qu'il se passe dans l'interpréteur. Si vous n'êtes pas encore à l'aise avec les dessins conformes à la réalité que nous avons vus jusqu'à présent, **il est déconseillé d'utiliser** l'option `deeply=False`.

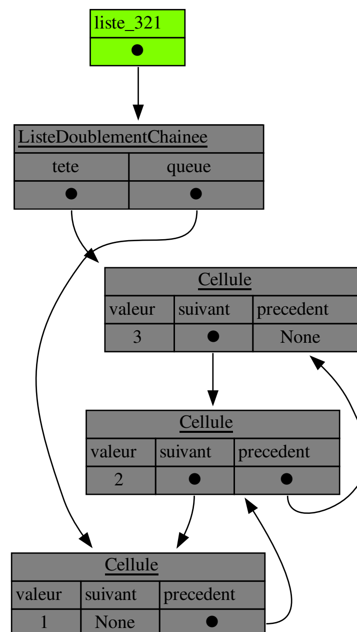

Fig. 1 : liste doublement chaînée

Comme l'illustre l'image ci-dessus, le type `ListeDoublementChainee` est le même que le type `ListeSimplementChainee` manipulé jusque là, c'est à dire avec une référence vers la tête de liste et une autre vers la queue.

Le type `Cellule` devient par contre le suivant :

```
class Cellule:
    """Une cellule d'une liste doublement chaînée.

    Contient une référence vers une valeur, une référence
    vers la cellule suivante et une référence vers la cellule
    précédente.
    """

    def __init__(self, valeur, precedent, suivant):
        self.valeur = valeur
        self.suivant = suivant
        self.precedent = precedent
```

```
13
14     def __str__(self):
15         """Pour pouvoir afficher des cellules facilement."""
16         return str(self.valeur)
```

Question 1

Écrire les fonctions d'ajout d'éléments, de prototypes
`ajoute_en_tete(liste_chainee, valeur)` et `ajoute_en_queue(liste_chainee, valeur)`
Dans ce TD, ces fonctions seront les deux seules créant des cellules. Quelle est la complexité temporelle de chacune de ces deux fonctions ?

Question 2

Programmer un générateur renvoyant un itérateur sur toutes les cellules de la liste de la tête à la queue. Il devra être résistant aux modifications sur la cellule courante.

Question 3

Programmer un générateur renvoyant un itérateur sur toutes les cellules de la liste de la queue à la tête. Il devra être résistant aux modifications sur la cellule courante. Serait-ce aussi simple avec une liste simplement chaînée ?

Question 4

En utilisant `filter`, `next`, et le générateur sur les cellules, programmer une fonction renvoyant la première cellule contenant la valeur donnée ou `None` si la valeur n'est pas présente dans la liste.
Le prototype de cette fonction est `recherche(liste_chainee, valeur)`.
La documentation de `next` et de `filter` est donnée ci-dessous.

```
1 filter(function or None, iterable) --> filter object
2 Return an iterator yielding those items of iterable for which function(item)
3 is true. If function is None, return the items that are true.
4
5 next(iterator[, default])
6 Return the next item from the iterator. If default is given and the iterator
7 is exhausted, it is returned instead of raising StopIteration.
```

Question 5

Programmer une fonction : `enleve_cellule(liste_chainee, cellule)` enlevant la cellule donnée, qui **fait** partie de la liste. Quelle est sa complexité temporelle ?

Question 6

En supposant que notre liste soit de longueur ≥ 2 et contienne des entiers, proposer une fonction renvoyant un des couples d'entiers les plus proches de la liste, de prototype : `recupere_entiers_proches(liste_chainee)`
Si le cœur nous en dit, on pourra utiliser `itertools.combinations` dont la documentation est donnée ci-dessous.
Quelle est la complexité temporelle de la fonction `recupere_entiers_proches` ?

```
combinations(iterable, r) --> combinations object
Return successive r-length combinations of elements in the iterable.
combinations(range(4), 3) --> (0,1,2), (0,1,3), (0,2,3), (1,2,3)
```

Question 7

Écrire une fonction `supprime_doublons(liste_chainee)`, éliminant tous les doublons de la liste.

Exercice 2 : j'entrelace (pour aller plus loin)

Question 1

Écrire une fonction `entrelace(liste_chainee_1, liste_chainee_2)` qui entrelace les deux listes chaînées dans `liste_chainee_1`. Par exemple, entrelace `0 --> 2 --> 4 --> 6 --> 8 --> 10 --> 12` et `1 --> 3 --> 5 --> 7 --> 9` donne `0 --> 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> 9 --> 10 --> 12`. On supposera qu'aucune liste n'est vide. En sortie, `liste_chainee_2` doit être vide.