III - TP 2 Implémentation par des listes chaînées

1 Listes chaînées : description

1.1 Introduction

On rappelle que l'interface du type `Liste` est la suivante :

Fonction	Description
`creer_vide()`	Renvoie une liste vide
`est_vide(l)`	Renvoie `True` si et seulement si la liste est vide
`tete(l)`	Renvoie le premier élément de la liste l (l'élément dit **de tête**)
`queue(l)`	Renvoie la liste constituée de tous les éléments de l à l'exception du premier
`ajoute(l, e)`	Renvoie la liste constituée de l'élément e, suivi des éléments de l
`affiche(l)`	Affiche la liste des éléments de l sur la sortie standard, séparés par le caractère `-`. La liste vide est représentée par `x`.

On se propose de donner une implémentation de cette interface via un objet nommé *liste chaînée*. En python, on peut représenter les listes de la manière suivante : chaque élément de la liste est associé à un **objet**, que nous appelerons un "maillon" (faisant partie d'une "chaîne" de maillons) qui comporte deux attributs :

- un attribut `valeur` qui stocke la valeur de l'élément considéré ;
- un attribut `suivant` qui permet de déterminer quel est le prochain maillon dans la chaîne.

On représente **un maillon vide** par la valeur spéciale `None`. Ainsi, on pourra instancier un objet `m` de type `Maillon` de trois façons distinctes :

- `m = None` : le maillon est vide ;
- `m = Maillon(v, None)` : le maillon est le seul dans sa chaîne ;
- `m = Maillon(v, m2)` : le prochain maillon de la chaîne est le maillon `m2` (supposé ici non vide).

Une chaîne de maillons ainsi constituée représente bien une liste d'éléments. La classe `Liste` possède donc un seul attribut : `head` une référence vers le premier maillon de la chaîne.

Code python

```python
class Maillon:
    """ Un maillon d'une liste chainée. """
    def __init__(self, v, s):
        """ int, Maillon -> None """
        self.valeur = v
        self.suivant = s

class Liste:
    """ Une liste d'éléments. """
    def __init__(self, m):
        """ Maillon -> None """
        self.head = m
```

À l'aide de l'objet `Maillon`, on peut représenter la liste $l = (3, 8, 42, 1)$. Pour chaque élément e de la liste, on créé un maillon qui contient la valeur de l'élément et le lien vers le maillon suivant.

Code python

```python
maillon4 = Maillon(1, None)
maillon3 = Maillon(42, maillon4)
maillon2 = Maillon(8, maillon3)
maillon1 = Maillon(3, maillon2)

print(maillon2.valeur)
print(maillon2.suivant)
print(maillon2.suivant.valeur)
```

Créer des listes de cette façon est très fastidieux, en raison de l'utilisation de multiples variables. On peut condenser le code de la manière à n'utiliser qu'une seule variable :

Code python

```python
m = Maillon(1, None)
m = Maillon(42, m)
m = Maillon(8, m)
m = Maillon(3, m)
# encore plus court :
# m = Maillon(3, Maillon(8, Maillon(42, Maillon(1, None))))

print(m.valeur)
print(m.suivant)
print(m.suivant.valeur)
```

Résultat

```
3
<__main__.Maillon object at 0x7eb3b8d157c0>
8
```

1.2 Premier exemple

Écrire les instructions python permettant de stocker à l'aide d'objets de type `Maillon`, ou de la valeur spéciale `None` les listes $l_1 = ()$, $l_2 = (-1)$, $l_3 = (5, 6, 9, -1)$, $l_4 = (-5, 9, 13)$.

Code python

```python
# À compléter
# m1 = ...
# m2 = ...
# m3 = ...
# m4 = ...
```

Code python

```python
print(m4.valeur)
print(m4.suivant.valeur)
print(m4.suivant.suivant.valeur)
print(m4.suivant.suivant.suivant.valeur)
```

Résultat

```
-5
9
13
```

Résultat

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
↪ last)
Cell In[1882], line 4
      2 print(m4.suivant.valeur)
      3 print(m4.suivant.suivant.valeur)
----> 4 print(m4.suivant.suivant.suivant.valeur)

AttributeError: 'NoneType' object has no attribute 'valeur'
```

Question 1. Comment pouvez-vous expliquer l'erreur obtenue par l'exécution de la dernière instruction de l'exemple ci-dessus ?

2 Implémentation de l'interface de base des listes

2.1 Créer une liste vide

Écrire une fonction `creer_vide` qui renvoie, **avec les conventions de l'énoncé**, un maillon vide.

Code python

```python
def creer_vide():
    """ () -> Maillon
    Renvoie un maillon vide """
    pass
```

2.2 Tester si un maillon est vide

Écrire une fonction `est_vide` étant donné un maillon `m` renvoie `True` si et seulement si celui-ci est vide.

Code python

```python
def est_vide(m):
    """ Maillon -> bool
    Renvoie True si et seulement si le maillon m est le maillon vide """
    pass
```

Code python

```python
print(est_vide(m1))
print(est_vide(m2))
```

Résultat

```
True
False
```

2.3 Élément de tête

Écrire une fonction `tete` qui étant donné un maillon non vide `m`, renvoie la valeur stockée dans ce maillon.

Code python

```python
def tete(m):
    """ Maillon -> int
    m est non vide
    Renvoie l'attribut valeur du maillon m """
    pass
```

Code python

```python
for m in [m4, m3, m2]:
    print(tete(m), end = ' ')
```

Résultat

```
-5 5 -1
```

2.4 Élément de queue

Écrire une fonction `queue` qui étant donné un maillon non-vide `m`, renvoie le maillon suivant de la chaîne de maillons.

Code python

```python
def queue(m):
    """ Maillon -> Maillon
    m est non vide
    Renvoie le maillon suivant dans la chaine de maillon de tête m. """
    pass
```

Code python

```python
print(queue(maillon1))
print(maillon2)
print(queue(maillon1) is maillon2)
print(queue(maillon1) is Maillon(8, None))
```

Résultat

```
<__main__.Maillon object at 0x7eb3b8b947d0>
<__main__.Maillon object at 0x7eb3b8b947d0>
True
False
```

Question 2. Selon vous, quel est l'effet de l'instruction `a is b` ?

2.5 Ajout d'un maillon

Écrire une fonction `ajoute` qui étant donné un maillon `m` et un élément `e` renvoie le maillon `m'` dont l'attribut valeur est `e` et dont l'attribut `suivant` est `m`.

Code python

```python
def ajoute(m, e):
    """ Maillon, int -> Maillon
    Renvoie le maillon dont l'attribut valeur est e et l'attribut suivant
    ↪ est m """
    pass
```

Code python

```python
m = creer_vide()
m = ajoute(m, 5)
m = ajoute(m, 10)
print(tete(m), tete(queue(m)))
```

Résultat

```
10 5
```

2.6 Affichage d'une liste chaînée

Pour afficher une liste chaînée, il faut **parcourir** tous les maillons la composant. Pour cela, deux approches sont possibles : une méthode récursive et méthode itérative.

2.6.1 Méthode récursive

Si le maillon est vide alors on affiche un symbole indiquant qu'il s'agit du maillon vide. Sinon, on affiche la valeur contenue dans le maillon, puis on affiche les maillons suivants de la chaîne.

Code python

```
1  def affiche_rec(m):
2      """ Maillon -> None
3      Affiche les valeurs de la chaîne dont le premier maillon est m. """
4      if est_vide(m):
5          print("x")
6      else:
7          print(tete(m), end = " - ")
8          affiche_rec(queue(m))
9
10 affiche_rec(m3)
```

Résultat

```
5 - 6 - 9 - -1 - x
```

Question 3. **1.** Que se passe-t-il si on échange les lignes 7 et 8 du code de la fonction `affiche_rec` ?

2. À quoi sert l'argument nommé `end` de la fonction `print` ? Modifier le code de la fonction précédente afin que l'affichage effectué par l'instruction `affiche_rec(m3)` soit :

Résultat

```
5 | 6 | 9 | -1 | x
```

2.6.2 Méthode itérative

On initialise une variable `maillon_courant` qui stocke le maillon courant du parcours de la liste. Tant que le maillon courant n'est pas vide (ce qui veut dire que l'on n'est pas encore au bout de la chaîne), on affiche la valeur du maillon courant puis le maillon courant devient le maillon suivant dans la chaîne. Lorsque l'on atteint la fin de la chaîne, on affiche un symbole indiquant qu'il s'agit du maillon vide.

Code python

```
1  def affiche_i(m):
2      """ Maillon -> None
3      Affiche les valeurs de la chaîne dont le premier maillon est m """
4      maillon_courant = m
5      while not est_vide(maillon_courant):
6          print(tete(maillon_courant), end=' - ')
7          maillon_courant = queue(maillon_courant)
8      print("x")
9
10 affiche_i(m3)
11 affiche = affiche_i
```

On utilisera dans la suite simplement l'instruction `affiche` pour afficher une chaîne de maillons.

3 Parcours d'une liste chaînée

3.1 Longueur d'une chaîne

On appelle longueur d'une chaîne de maillons le nombres de maillons non vide qui la composent. Écrire une fonction **itérative** `longueur` qui étant donné un maillon `m`, détermine la longueur de la chaîne de maillons dont le premier maillon est `m`.

Code python

```python
def longueur(m):
    """ Maillon -> int
    Compte le nombre de maillons présents dans la chaîne dont le premier
    ↪ maillon est m """
    # version itérative
    pass
```

Code python

```python
affiche(m3)
print(longueur(m3))
```

Résultat

```
5 - 6 - 9 - -1 - x
4
```

3.2 Élément d'indice i

On dit qu'un entier i est compatible avec la longueur d'une liste `l` constituée de n éléments si $i < n$. Écrire une fonction **itérative** `element` qui étant donné un maillon `m`, et un entier `i`, supposé compatible avec la longueur de la chaîne, renvoie la valeur stockée dans le maillon d'indice i de la chaine dont le premier maillon est `m`.

Code python

```python
def element(m, i):
    """ Maillon, int -> int
    m est non vide, 0 <= i < longueur(m)
    Renvoie l'élément d'indice i de la chaîne de maillons dont le premier
    ↪ est m """
    pass
```

Code python

```python
for i in range(longueur_i(m3)):
    print(element_i(m3, i), end = ' ')
```

Résultat

```
5 6 9 -1
```

Question 4. Écrire une fonction **récursive** `element_rec` qui étant donné un maillon `m`, et un entier `i`, supposé compatible avec la longueur de la chaîne, renvoie la valeur stockée dans le maillon d'indice i de la chaine dont le premier maillon est `m`.
Remarque. Le cas de base de cette fonction est atteint lorsque i vaut 0.

3.3 Remplacer l'élément d'indice i

Écrire une fonction itérative `remplace` qui étant donné un maillon `m` non vide, un entier `i`, supposé compatible avec la longueur de la chaîne, et un élément `e`, modifie la valeur du maillon d'indice `i` par `e`. Ainsi, après l'exécution de `remplace(m, i, e)` :

- le premier maillon de la chaîne est toujours le maillon `m` ;
- l'attribut `valeur` du maillon d'indice `i` de la chaine a été remplacé par `e` ;
- les autres maillons de la chaîne sont inchangés.

Code python

```python
def remplace(m, i, e):
    """ Maillon, int, int -> None
    m est non vide, 0 <= i < longueur(m)
    Remplace par e la valeur du i-ème maillon de la chaine commençant par
    ↪ m """
    pass
```

Code python

```python
affiche(m3)
remplace(m3, 2, 42)
affiche(m3)
affiche(m4)
remplace(m4, 0, 24)
affiche(m4)
```

Résultat

```
5 - 6 - 9 - -1 - x
5 - 6 - 42 - -1 - x
-5 - 9 - 13 - x
24 - 9 - 13 - x
```

Question 5. **1.** Écrire une fonction récursive `remplace_rnm` (récursive non mutable) qui étant donné un maillon `m` non vide, un entier `i`, supposé compatible avec la longueur de la chaîne, et un élément `e`, renvoie le premier maillon de la chaîne où la valeur du maillon d'indice `i` a été remplacée par `e`, **sans modifier** de maillon.

Code python

```python
affiche(m4)
affiche(remplace_rnm(m4, 0, 54))
affiche(m4)
affiche(m3)
affiche(remplace_rnm(m3, 2, 24))
affiche(m3)
```

Résultat

```
24 - 9 - 13 - x
54 - 9 - 13 - x
24 - 9 - 13 - x
5 - 6 - 42 - -1 - x
5 - 6 - 24 - -1 - x
5 - 6 - 42 - -1 - x
```

2. Écrire une version itérative de la fonction `remplace` qui ne modifie aucun maillon.

3.4 Ajouter un élément à l'indice i

Écrire une fonction itérative `ajoute_position` qui étant donné un maillon `m`, un entier `i`, supposé compatible avec la longueur de la chaîne, et un élément `e`, ajoute un maillon à la chaine de telle sorte que :

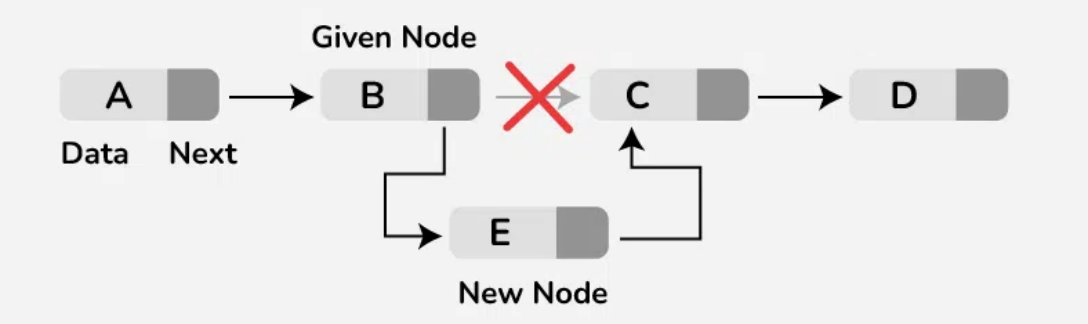
- le premier maillon de la chaîne soit toujours le maillon `m` ;
- le maillon d'indice `i` de la chaîne ait pour valeur `e` ;
- l'ordre d'apparition des valeurs des autres maillons de la chaîne est inchangé.

Code python

```python
def ajoute_position(m, e, i):
    """ Maillon, int, int -> None
    Ajoute l'élément e en i-ième position de la chaine dont le premier
    ↪ maillon est m """
    pass
```

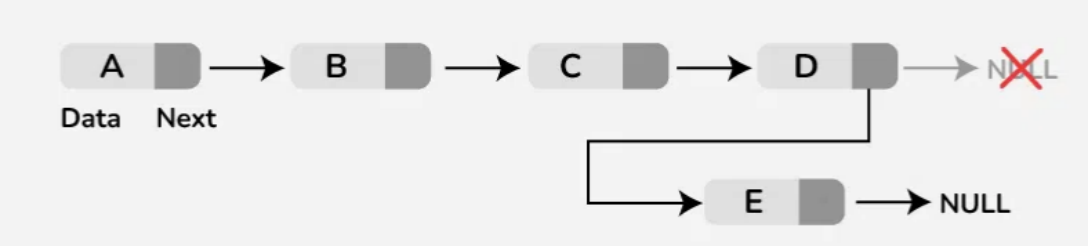
Pour cela, on parcourera tous les maillons de la chaîne jusqu'à ce que le `maillon_courant` ait pour indice `i` (insertion en milieu de liste) ou que le maillon suivant de la chaîne soit vide (insertion en fin de liste).

- Dans le cas où on insère en milieu de liste : on créé alors un `nouveau` maillon que l'on insère entre le `maillon_courant` et le maillon suivant de la chaîne comme sur le schéma ci-dessous.



On s'assure que les attributs `valeur` des maillons ont été correctement modifiés : l'attribut `valeur` du maillon d'indice `i` doit être `e`, l'attribut valeur du `nouveau` maillon doit être l'ancienne valeur du maillon d'indice `i`.

- Dans le cas où on insère en fin de liste (cela veut dire que `i` vaut `longueur(m)`) : il suffit juste de créer un nouveau maillon et de le relier au dernier maillon de la liste.



Code python

```
affiche(m3)
ajoute_position(m3, 0, 123)
affiche(m3)
ajoute_position(m3, longueur(m3), 321)
affiche(m3)
```

Résultat

```
5 - 6 - 42 - -1 - x
123 - 5 - 6 - 42 - -1 - x
123 - 5 - 6 - 42 - -1 - 321 - x
```

Question 6. **1.** Écrire une fonction récursive `ajoute_position_rnm` qui étant donné un maillon `m` non vide, un entier `i`, supposé compatible avec la longueur de la chaîne, et un élément `e`, renvoie le premier maillon de la chaîne où on a inséré à l'indice `i` un maillon de valeur `e` **sans modifier** de maillon.

Code python

```
affiche(m4)
affiche(ajoute_position_rnm(m4, 0, 123))
affiche(m4)
affiche(ajoute_position_rnm(m4, 3, 321))
affiche(m4)
```

Résultat

```
24 - 9 - 13 - x
123 - 24 - 9 - 13 - x
24 - 9 - 13 - x
24 - 9 - 13 - 321 - x
24 - 9 - 13 - x
```

2. Écrire une version itérative de la fonction `ajoute_position` qui ne modifie aucun maillon.