

CHAPITRE 3

Listes et listes chaînées

1 Listes : interface

Définition 1. Une **liste** est une structure de donnée permettant de regrouper un nombre fini de données de manière séquentielle.

Exemple 1. Par exemple $(0, 1, \pi, 1, -6, 1, \text{"Bonjour"})$ est une liste. De manière générale on préfère que tous les éléments d'une liste soient du même type. Par exemple tous les éléments de la liste $(3, 5, 9)$ sont des entiers.

Interface. On donne ci-dessous la liste des opérations que doit supporter un objet de type liste :

Fonctionnalité	Description
<code>creer_vide()</code>	Renvoie une liste vide.
<code>est_vide(l)</code>	Teste si la liste <code>l</code> est vide.
<code>ajoute(l, e)</code>	Ajoute au début de la liste <code>l</code> l'élément <code>e</code> .
<code>tete(l)</code>	Renvoie le premier élément de la liste <code>l</code> .
<code>queue(l)</code>	Renvoie la liste constituée de tous les éléments de <code>l</code> sauf le premier.
<code>element(l, i)</code>	Renvoie le <code>i</code> -ième élément de la liste <code>l</code>

2 Implémentation via des listes chaînées

2.1 Définition et exemple d'utilisation

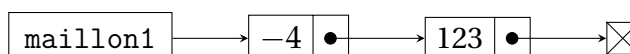
Dans cette implémentation du type liste, les éléments sont chaînés entre eux : chaque élément de la liste est stocké dans un objet `Maillon`, où il y est accompagné de l'adresse mémoire de l'élément suivant dans la liste.

On décide de représenter un maillon vide par l'élément `None`.

Code

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

Schéma. Avec cette implémentation, chaque élément de la liste est chaîné au suivant.



Exemple

```
maillon1 = Maillon(-4, Maillon(123, None))
```

Cela correspond à la liste $(-4, 123)$.

Une liste est représentée par son maillon de tête. On peut alors implémenter les fonctions suivantes de l'interface.

```

def creer_vide() -> Maillon: return None
def est_vide(m: Maillon) -> bool: return m is None
def ajoute(m: Maillon, e: int) -> Maillon: return Maillon(e, m)
def tete(m: Maillon) -> int: return m.valeur
def queue(m: Maillon) -> Maillon: return m.suivant

```

Parcours récursif. Pour écrire une fonction f qui prend en argument une liste l et résout le problème \mathcal{P} :

- On traite le cas de base selon le contexte du problème :
- Dans le cas général :
 1. On obtient récursivement une solution partielle en appliquant la fonction f résolvant le problème à la queue de la liste.
 2. On utilise la solution partielle et la tête de la liste pour répondre au problème général.



2.2 Objet mutable

Création d'un nouvel objet. Le **constructeur** `Maillon` construit un **nouveau** maillon, indépendant des autres maillons déjà créés.

Modification d'un objet. Lorsque `m` est un maillon, on accède à ses attributs via `m.valeur` et `m.suivant`. Si l'on affecte une nouvelle valeur à ces attributs, on **modifie en place** le maillon `m`.

Code Python

```
m1 = Maillon(1, None)
m2 = Maillon(2, m1)
m1 = Maillon(1, m2)
```

Code Python

```
m1 = Maillon(1, None)
m2 = Maillon(2, m1)
m1.suivant = m2
```

Pour le moment l'interface donnée ne permet pas de modifier un objet de type `Maillon`. On peut l'enrichir de deux nouvelles fonctions qui permettent respectivement de modifier l'attribut `valeur` et l'attribut `suivant` d'un objet de type `Maillon` :

Code Python

```
def set_valeur(m:Maillon, e:int) -> NoneType: m.valeur = e
def set_suivant(m:Maillon, m2:Maillon) -> NoneType : m.suivant = m2
```

Exemple 4. Écrire une fonction `ajoute_fin` qui étant donné une liste `l` et un entier `e` et qui renvoie la liste composée des éléments de `l` à laquelle on a ajouté l'élément `e`.

La liste initiale **ne sera pas modifiée**.

.....

.....

.....

.....

.....

.....

.....

Exemple 5. Écrire une fonction `ajoute_fin` qui étant donné une liste `l` et un entier `e` et qui renvoie la liste composée des éléments de `l` à laquelle on a ajouté l'élément `e`.

La liste initiale **pourra être modifiée**.

.....

.....

.....

.....

.....

.....


.....

2.3 Encapsulation dans un objet

La classe Liste ci-dessous implémente le type abstrait de données "listes non mutables", définit par l'interface de la section 1.

Fichier data_structures.py

```
1 class Liste:
2     def __init__(self, m = None):    # Par défaut la liste est vide
3         """ Initialise une liste dont le premier maillon est m. """
4         self.head = m
5
6     def queue(self):
7         """ Renvoie la queue de la liste """
8         return Liste(self.head.suivant) # version non mutable
9
10    def tete(self):
11        """ Renvoie la valeur en tête de liste """
12        return self.head.valeur
13
14    def est_vide(self):
15        """ Détermine si la liste est vide """
16        return self.head is None
17
18    def est_singleton(self):
19        """ Détermine si la liste est constituée d'un seul élément """
20        return not self.est_vide() and self.queue().est_vide()
21
22    def ajoute(self, e: int):
23        """ Ajoute en tête de liste l'élément e. """
24        new = Maillon(e, self.head)
25        return Liste(nouveau) # version non mutable
26
27    def affiche(self):
28        """ Affiche la liste. """
29        if self.est_vide():
30            print("x")
31        else:
32            print(f"{self.tete()}", end = " | ")
33            self.queue().affiche()
34
35 l = Liste()
36 l.ajoute(42)
37 l.ajoute((3))
38 l.ajoute((1))
39 l.affiche()
```

 Résultat

1 | 3 | 42 | x

Il est possible d'importer toutes les fonctionnalités de l'interface avec l'instruction :

```
from data_structures import Liste
```