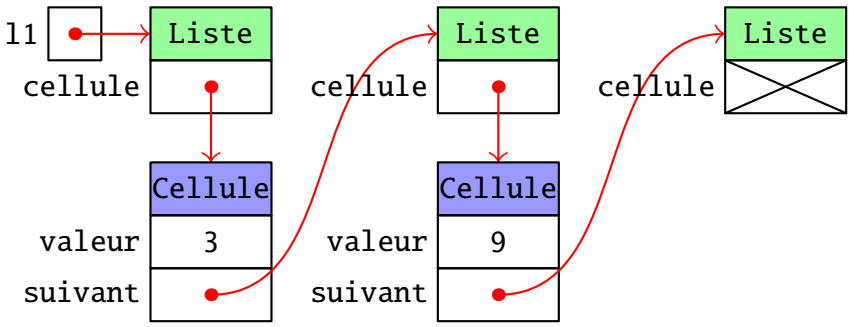
Python – Listes chaînées en POO

Avec des classes

Nous allons définir deux classes : une pour les cellules et une autre pour les listes.

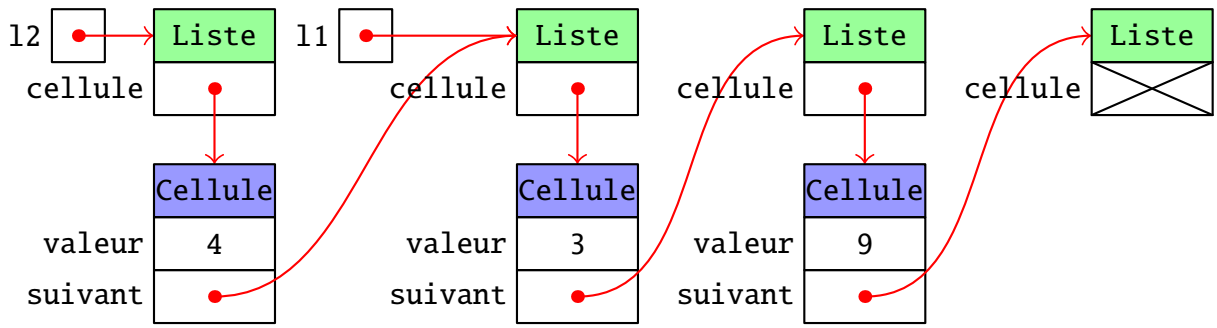
```
class Cellule:
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class Liste:
    def __init__(self, cellule):
        self.cellule = cellule

    def est_vide(self):
        return self.cellule is None

nil = Liste(None)
```

La liste nulle sera toujours notée `nil`. Le test pour la liste vide se fait en regardant si la cellule est nulle ou pas. On pourrait utiliser `self.cellule == None`, mais comme la notion d'égalité peut être redéfinie dans la classe `Cellule`, tester `is None` garantit que c'est bien l'égalité de pointeur qui est réalisée. En effet, l'objet `None` pointe toujours vers la même adresse. Voici une représentation d'une liste :



Exercice 1 : Écrire une fonction `cons(val, liste)` qui renvoie une nouvelle liste correspondant à l'ajout de `val` en tête de la liste.

```
>>> l1 = cons(3, cons(9, nil))
>>> l2 = cons(4, l1)
```



Exercice 2 : Rajouter les méthodes suivantes dans la classe `Liste` :

Méthode	Description
`tete(self)`	Renvoie la valeur du premier maillon de **`self`**, qui ne doit pas être vide.
`queue(self)`	Renvoie la liste sur laquelle pointe le premier maillon de **`self`**, qui ne doit pas être vide.

Lorsque la liste est vide alors qu'elle ne devrait pas l'être, lever l'exception `IndexError` et le message `'liste vide'`.

Exercice 3 : Rajouter la méthode **`__str__(self)`** qui renvoie un texte représentant de la liste. Il est plus simple d'écrire cette méthode en utilisant la récursivité. Vous pouvez utiliser **`str`**`(self.tete())` pour obtenir le texte correspondant à la tête de la liste. Vous pouvez aussi utiliser **`str`**`(self.queue())` pour l'appel récursif.

```
>>> str(l2)
'4 -> 3 -> 9 -> nil'
```

Exercice 4 : Rajouter les méthodes suivantes dans la classe `Liste` :

Méthode	Description
`__len__(self)`	Renvoie le nombre d'éléments de `liste`. La longueur de la liste vide est de 0. Cela permet d'écrire **`len`**`(liste)`.
`__contains__(self, v)`	Renvoie un booléen qui indique si **`self`** contient `v`. Cela permet d'écrire `v` **`in`** `liste`.
`__getitem__(self, i)`	Renvoie la valeur de l'élément d'indice `i` de `liste` s'il existe et sinon lève une exception `IndexError` et le message `'indice non trouvé'`. Cela permet d'écrire `liste[i]`.

Pour les appels récursifs, vous pouvez utiliser **`len`**`(self.queue())`, `v` **`in`** **`self`**`.queue()` ou **`self`**`.queue()[...]`.

```
>>> len(l2)
3
>>> len(cons(4, cons(2, nil)))
2
>>> 4 in l2
True
>>> 7 in l2
False
>>> 1 in nil
False
>>> l2[0]
4
>>> l2[1]
3
>>> l2[2]
9
>>> cons(7, cons(3, cons(8, nil)))[1]
3
```

S'il n'y a pas de méthode `__contains__`, la méthode `__getitem__` qui est utilisée pour `v` **`in`** `liste`.

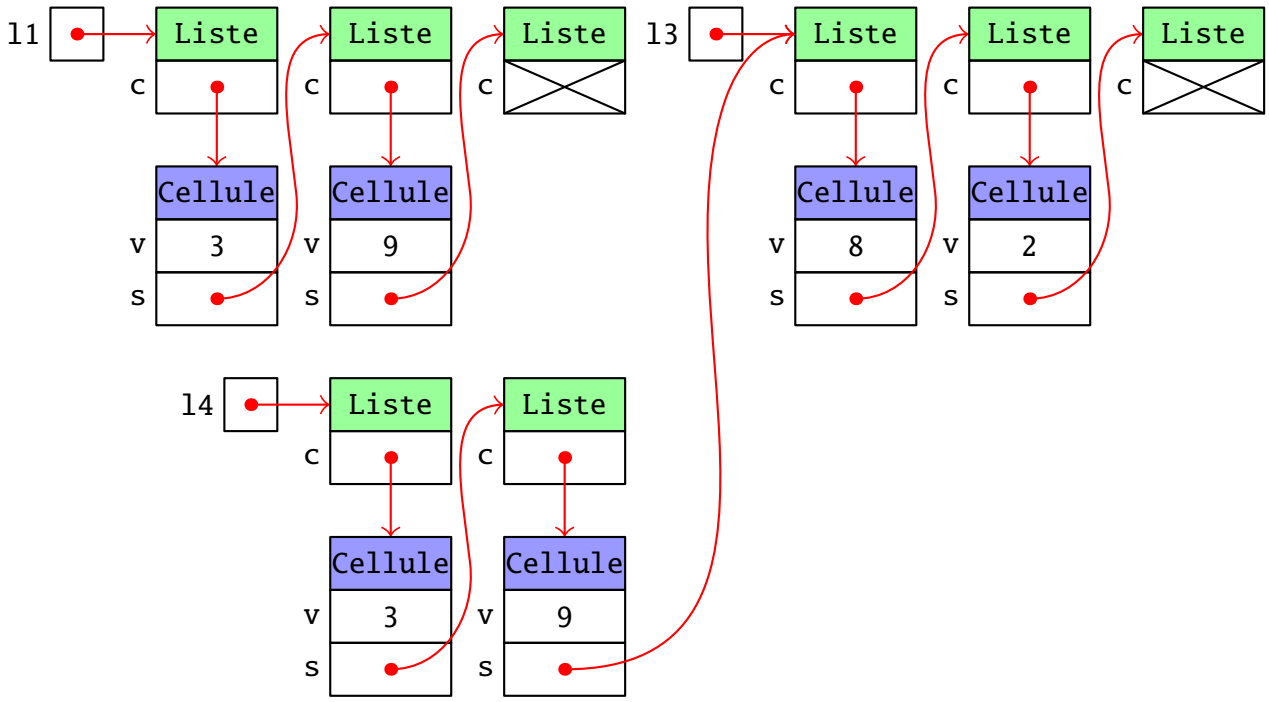
Exercice 5 : Rajouter `print("contains", self)` au début de la méthode `__contains__` et `print("getiteem", self)` au début de `__getitem__`. Tester l'affichage obtenu lors de l'exécution de "`2 in l2`" en commentant ou non la définition de `__contains__`.

Exercice 6 : Écrire une fonction récursive `concatene(liste1, liste2)` qui renvoie une nouvelle liste correspondant à la concaténation de `liste1` et `liste2`. Toutes les maillons de `liste1` sont copiés, mais ceux de `liste2` sont les originaux. La liste `liste1` ne doit pas être modifiée.

```
>>> print(concatene(l2, l2))
4 -> 3 -> 9 -> 4 -> 3 -> 9 -> nil
>>> print(l2)
4 -> 3 -> 9 -> nil
```

Voici une représentation du résultat de cette suite d'instructions :

```
>>> l3 = cons(8, cons(2, nil))
>>> l4 = concatene(l1, l3)
>>> l4
3 -> 9 -> 8 -> 2 -> nil
```



En rajoutant la méthode suivante dans `Liste`, il est possible d'utiliser la notation `liste1 + liste2`.

```
class Liste:
    ...

    def __add__(self, other):
        return concatene(self, other)
```

```
>>> print(l2 + l2)
4 -> 3 -> 9 -> 4 -> 3 -> 9 -> nil
```

Exercice 7 : Écrire une fonction récursive `concat_envers(liste1, liste2)` qui renvoie une nouvelle liste correspondant à la concaténation de la liste `liste1` à l'envers et de `liste2` à l'endroit. Comme pour `concatene`, les maillons de `liste1` sont copiés et pas ceux de `liste2`. Vous ne devez pas utiliser `concatene`.

```
>>> l3 = cons(8, cons(2, nil))
>>> print(concatene(l1, l3))
9 -> 3 -> 8 -> 2 -> nil
>>> print(concatene(cons(1, cons(2, cons(3, nil))), cons(4, cons(5, nil))))
3 -> 2 -> 1 -> 4 -> 5 -> nil
```

Exercice 8 : Écrire une méthode `reverse(self)` pour la classe `Liste` qui renvoie une nouvelle liste correspondant à la liste en sens inverse. Vous pouvez utiliser `concat_envers`.

Rendre la classe itérable

Avec l'ajout de la méthode `__getitem__`, il est possible d'écrire "`for v in liste`". Lors de l'exécution, `v` va prendre pour valeur `liste[0]`, `liste[1]`, ... jusqu'à ce que l'exception `IndexError` soit levée. Sauf que cela veut dire qu'à chaque fois on parcours la liste du départ jusqu'à l'élément d'indice `i`. Le coût du parcours est donc quadratique au lieu d'être linéaire. Pour régler ce problème, on peut créer un **itérateur** qui est un objet qui va parcourir la liste au fur et à mesure. L'objet peut être l'itérateur lui-même, mais dans notre cas, c'est plus simple de rajouter une nouvelle classe pour cela :

```
class Liste:
    ...

def __iter__(self):
        return ListeIter(self)

class ListeIter:
    def __init__(self, liste):
        self.liste = liste

    def __next__(self):
        ...
```

Lors de l'exécution de "`for v in liste`", un nouvel itérateur va être créé, par l'appel de `liste.__iter__()`. Ensuite, des appels à la méthode `__next()__` vont renvoyer les valeurs de `liste`. Lorsqu'il n'y aura plus d'éléments, l'itérateur lèvera l'exception `StopIteration`.

Exercice 9 : Compléter la méthode `__next__` de la classe `ListeIter` pour qu'elle renvoie la tête de l'attribut `liste` de l'itérateur et avance d'un cran dans cette liste. Si la liste est vide, il faut lever l'exception `StopIteration`.

Exercice 10 : Rajouter un `print` au début de `__next__`. Comparer l'affichage en commentant ou pas la méthode `__iter__`.

Pas si immuable que ça

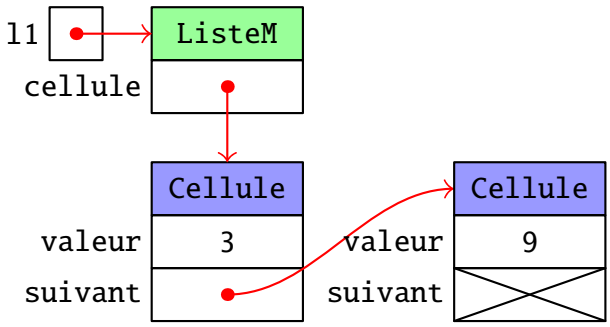
Nous allons maintenant définir une nouvelle classe `ListeM` qui donnera accès à des objets pouvant être modifiés. L'idée, c'est que l'attribut `suivant` des cellules ne pointera plus sur une liste mais directement sur une autre cellule. Pour ajouter un élément à la liste, il faut juste rajouter une nouvelle cellule au début et la liste pointera sur cette cellule.

```
class ListeM():
    def __init__(self):
        self.cellule = None

    def est_vide(self):
        return self.cellule is None

    def tete(self):
        if self.est_vide():
            raise IndexError('Liste vide')
        return self.cellule.valeur

    def queue(self):
        if self.est_vide():
            raise IndexError('Liste vide')
        l = ListeM()
        l.cellule = self.cellule.suivant
        return l

    def ajouter_tete(self, v):
        self.cellule = Cellule(v, self.cellule)
```

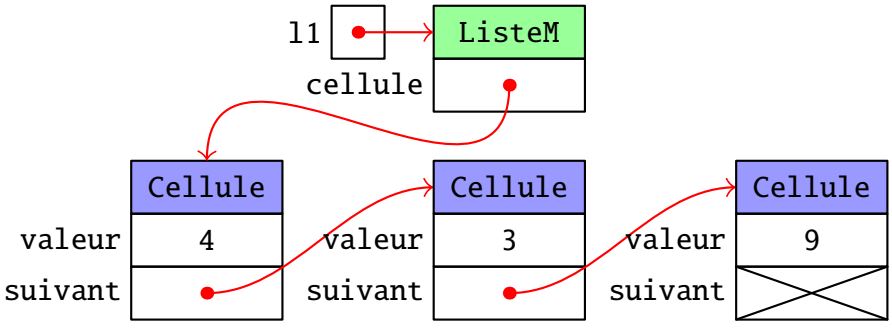
Pour que la méthode queue renvoie bien une liste, on commence par en créer une nouvelle et ensuite on la fait pointer sur la cellule suivante.
Afin de créer une liste, on part de la liste vide et on rajoute les éléments en tête.

```
>>> l1 = ListeM()
>>> l1.ajouter_tete(9)
>>> l1.ajouter_tete(3)
```

On obtient alors la liste suivante :




```
>>> l1.ajouter_tete(4)
```

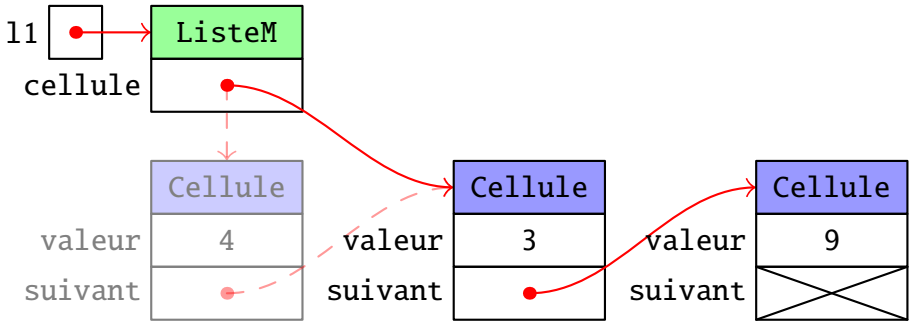


Exercice 11 : Rajouter les méthodes suivantes à `ListeM` :

Méthode	Description
`__str__(self)`	Renvoie un texte représentant de la liste.
`__len__(self)`	Renvoie le nombre d'éléments de `self`. La longueur de la liste vide est de 0.
`__contains__(self, v)`	Renvoie un booléen qui indique si `self` contient `v`.
`__getitem__(self, i)`	Renvoie la valeur de l'élément d'indice `i` de `self` s'il existe et sinon lève une exception `IndexError` et le message `'indice non trouvé'`
`copy(self)`	Renvoie une nouvelle liste qui est égale à `self` mais qui est totalement indépendante.

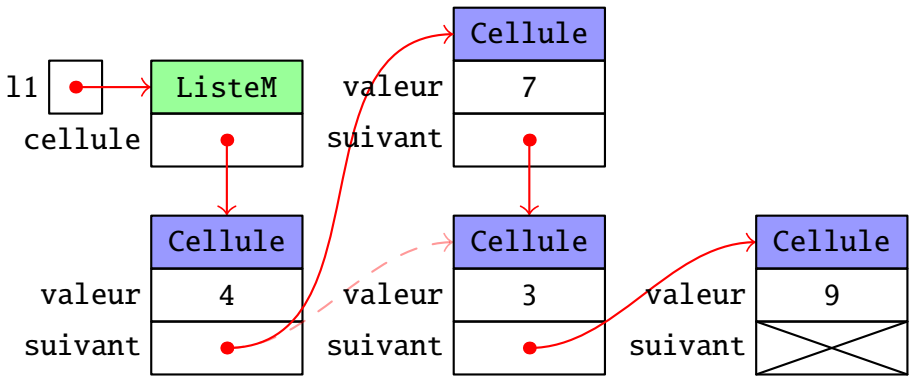
Exercice 12 : Écrire une méthode `enlever_tete(self)` qui renvoie la valeur de la tête et l'enlève de la liste ou lève l'exception `IndexError` et le message `'liste vide'`.

```
>>> l1.enlever_tete()
```



Exercice 13 : Écrire une méthode `ajouter_apres(self, v, i)` qui ajoute la valeur `v` après l'élément de d'indice `i`. Une exception est levée s'il n'y a pas d'élément d'indice `i`.

```
>>> l1.ajouter_apres(7, 0)
```



Exercice 14 : Écrire une méthode `enlever(self, i)` qui renvoie la valeur de l'élément de l'indice `i` et l'enlève de la liste. Une exception est levée s'il n'y a pas d'élément d'indice `i`. Attention, cette méthode est un peu plus dure à écrire. Il faut distinguer le cas où on veut enlever l'élément d'indice 0, le cas où on veut enlever le cas d'indice 1 et les autres cas.

Pour aller plus loin

Exercice 15 : Rajouter les classes `ListeD` et `CelluleD` qui permettent de construire des listes doublement chaînées. Les cellules ont les attributs `valeur`, `precedent` et `suivant`. Au lieu de `tete` et `queue`, on pourra utiliser `valeur`, `suivant` et `precedent`. Vous pouvez rajouter les méthodes de base, comme :

Méthode	Description
`est_vide(self)`	Renvoie un booléen indiquant si la liste est vide.
`__str__(self)`	Renvoie un texte représentant de la liste.
`__len__(self)`	Renvoie le nombre d'éléments de `self`. La longueur de la liste vide est de 0.
`__contains__(self, v)`	Renvoie un booléen qui indique si `v` est dans `self`.
`__getitem__(self, i)`	Renvoie la valeur de l'élément d'indice `i` de `self` s'il existe et sinon lève une exception `IndexError` et le message `'indice non trouvé'`
`copy(self)`	Renvoie une nouvelle liste qui est égale à `self` mais qui est totalement indépendante.

Exercice 16 : Rajouter la méthode `ajouter_apres(self, i, v)` qui rajoute `v` après l'élément d'indice `i`. Si l'élément n'existe pas, une exception est levée.

Exercice 17 : Rajouter la méthode `ajouter_avant(self, i, v)` qui rajoute `v` avant l'élément d'indice `i`. Si `i` vaut 0, il faut penser à modifier la cellule de la liste. Si l'élément n'existe pas, une exception est levée.

Exercice 18 : Rajouter la méthode `enlever(self, i)` qui renvoie la valeur de l'élément en position `i` et l'enlève de la liste. S'il n'existe pas, une exception est levée.