CC2.0 BY SA – www.infoforall.fr

Données 14 - Implémentation du type Liste Chaînée Mutable (avec un modèle objet)

I – Liste chaînée sous forme d'objets : la base

1.1 – Classe Cellule

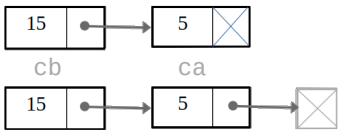
Les **Cellules** sont des instances de **`Cellule`** possédant 2 attributs : **`v`** pour la valeur, et **`s`** pour le successeur.
L'attribut **`s`** est une référence menant soit à **`None`** s'il s'agit d'une Cellule de Fin,
soit à une vraie **`Cellule`**.

```
class Cellule:
    def __init__(self, valeur:'Element', successeur:'Cellule|None'):
        self.v = valeur                        # Valeur ne peut pas être None si liste homogène
        self.s = successeur                    # successeur vaut None si Cellule de Fin

    def contenu(self:'Cellule') -> 'Element':
        return self.v

    def successeur(self:'Cellule NON FIN') -> 'Cellule':
        return self.s

    def a_successeur(self:'Cellule') -> bool:
        return self.s != None                          # ou return self.s is not None

    def est_fin(self:'Cellule') -> bool:
        return self.s == None                          # ou return self.s is None
```

Exemples d'instanciation :

```
ca = Cellule(5, None)
cb = Cellule(15, cc)
```

Voici deux représentations mentales qu'on peut se faire des 2 cellules :



Pour les primitives **`contenu()`** et **`successeur()`**, on peut :

- soit ne pas respecter l'encapsulation et utiliser la manipulation directe des attributs : **`c.v`** et **`c.s`**
- soit respecter l'encapsulation et passer par des méthodes **`contenu()`** et **`successeur()`**.

1.2 - Implémentation cellules-objets muables : la Liste

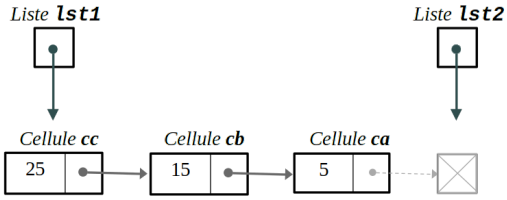
Une instance de **`Liste`** possède un attribut **`tete`** faisant référence :

1. soit à **`None`** s'il s'agit d'une Liste VIDE.
2. soit à la **`Cellule`** **en tête** s'il s'agit d'une Liste NON VIDE.

```
class Liste:
    def __init__(self):
        self.tete = None

    def est_liste_vide(self:'Liste') -> bool:
        return self.tete == None  # ou self.tete is None
```

Exemples d'instanciation :

```
lst1 = Liste()   # Liste vide
lst1.tete = cc   # lst est la liste 25 puis 15 puis 5
lst2 = Liste()   # Liste vide
```



1.3 - Un exemple (exercice de cours)

```
ca = Cellule(___, None)
cb = Cellule(___, ___)
cc = Cellule(___, ___)
lst1 = Liste()
lst1.tete = ___
```

On peut alors facilement accéder aux éléments stockés :

```
>>> lst1.tete.v
___

>>> lst1.tete.s.v
___

>>> lst1.tete.s.s.v
___
```

1.4 - Méthode acces_tete() et acces() (À SAVOIR RECREER)

```
class Liste:
    # ...
    def acces_tete(self:'Liste NON VIDE') -> Cellule:
        '''Renvoie l'adresse de la cellule de Tête dans la liste NON VIDE'''
        return self.tete

    def acces(self:'Liste NON VIDE', i:'int VALIDE') -> Cellule:
        '''Renvoie l'adresse de la cellule i dans la liste NON VIDE'''
        c = self.acces_tete()                                   # récupère la cellule de tête
        for _ in range(i):                                      # Faire i fois
            c = c.s                                             # passe au successeur
        return c
```

Le coût de la méthode **acces_tete()** est _____

Le coût de la méthode **acces()** est _____

1.5 - Méthode longueur() (À SAVOIR RECREER)

Le coût de la méthode **longueur()** est _____

Version itérative

```
class Liste:
    # ...
    def longueur(self:'Liste') -> int:
        '''Renvoie la longueur de la liste'''
        if self.est_liste_vide():
            return 0
        else:
            c = self.tete          _____________________________________
            nb = 1
            while c.s != None:     _____________________________________
                nb = nb + 1
                c = c.s            _____________________________________
            return nb
```

Version récursive : la méthode de liste va faire appel à la méthode récursive **nb_cellules()** de Cellule

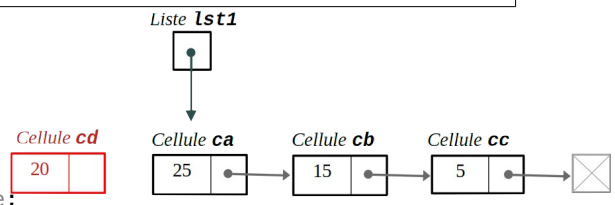
```
class Cellule:
    # ...
    def nb_cellules(self:'Cellule') -> int:
        '''Renvoie le nombre de cellules dans la chaîne à partir de cette cellule'''
        if self.s == None:
            return 1                        # CAS DE BASE _____________________________
        else:
            return 1 + self.s.nb_cellules()  # CAS RECURSIF _____________________________

                                            # _____________________________________
```

```
class Liste:
    # ...
    def longueur(self:'Liste') -> int:
        '''Renvoie la longueur de la liste'''
        if self.est_liste_vide():
            return 0                       # _____________________________________
        else:
            return self.tete.nb_cellules()  # _____________________________________
```

2 - Liste chaînée sous forme d'objets : l'insertion

2.1 - Insérer une cellule en position de TETE dans notre liste chaînée (À SAVOIR RECREER)

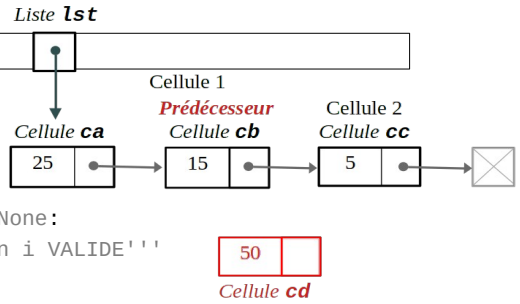
Cellule à localiser : la Tête actuelle.




```
class Liste:
    # ...
    def inserer_tete(self:'Liste', elt:'Element') -> None:
        '''Rajoute une nouvelle cellule contenant elt en tête de liste'''
___     nouvelle = ___________________      # Etape 1 : création de la cellule
___     nouvelle.s = _________________      # Etape 2 : on relie la nouvelle cellule à l'"ancienne"
___     __________ = _________________      # Etape 3 : modification de la liste
```

Le coût de la méthode **inserer_tete()** est _____

2.2 - Insérer une cellule en position i (À SAVOIR RECREER)

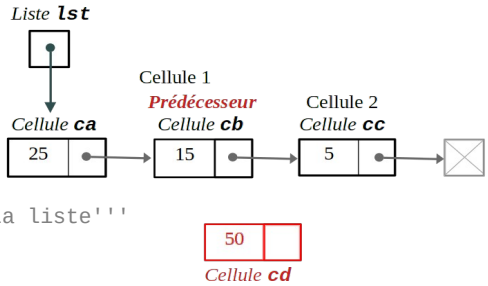


Cellule à localiser : la Cellule d'indice i-1 (si elle existe)

```
class Liste:
    # ...
    def inserer(self:'Liste', elt:'Element', i:'int VALIDE') -> None:
        '''Rajoute une nouvelle cellule contenant elt en position i VALIDE'''
        if i == 0:
            self.inserer_tete(elt)
        else:
            pred = ____________________          # Etape 1 : recherche de la cellule i-1
            nouvelle = Cellule(elt, None)        # Etape 2 : création de la cellule
            ___________ = _______________        # Etape 3 : liaison nouvelle i vers ancienne i
            ___________ = _______________        # Etape 4 : liaison i-1 vers nouvelle i
```

Le coût de la méthode **inserer()** est _____

2.3 - Insérer une cellule derrière la position de Fin (À SAVOIR RECREER)



Cellule à localiser : la **Fin** actuelle.

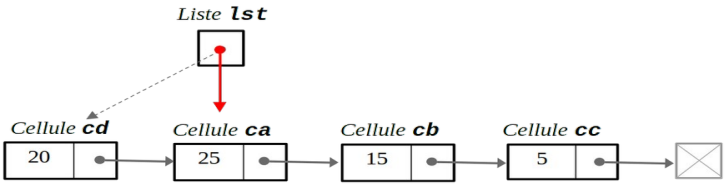
```
class Liste:
    # ...
    def inserer_fin(self:'Liste', elt:'Element') -> None:
        '''Rajoute une nouvelle cellule de Fin contenant elt dans la liste'''
        if self.est_liste_vide():
            self.inserer_tete(elt)
        else:
            pred = ________________                  # Etape 1 : recherche de la cellule de Fin
            while _____________!= None:              # TANT QUE pred n'est pas la fin actuelle
                pred = ______________                # on passe à la cellule suivante
            nouvelle = Cellule(elt, None)            # Etape 2 : création de la cellule
            ______________ = ________________        # Etape 3 : ancienne Fin vers nouvelle
```

Le coût de la méthode **inserer_fin()** est _____

Par contre, on pourrait faire mieux si _____

3 - Liste chaînée sous forme d'objets : la suppression

3.1 - Supprimer la cellule en position de TETE dans notre liste chaînée (À SAVOIR RECREER)

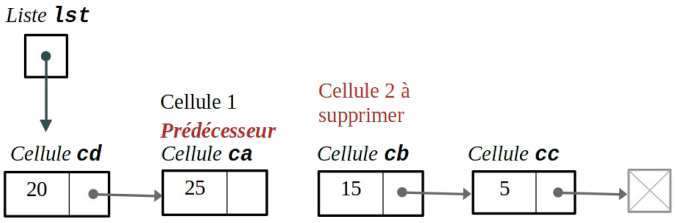


Cellule à localiser : la **Tête**.

Puisque la méthode n'aura rien à renvoyer, on lui fait envoyer la valeur qu'elle vient de supprimer.
VOIR VOTRE COURS et le CODE page 4.

Le coût de la méthode **supprimer_tete()** est _____

3.2 - Supprimer la cellule en position i dans notre liste chaînée (À SAVOIR RECREER)



Cellule à localiser : **la Cellule d'indice i-1** (si elle existe).

Puisque la méthode n'aura rien à renvoyer, on lui fait renvoyer la valeur qu'elle vient de supprimer.
VOIR VOTRE COURS et le CODE page 4.

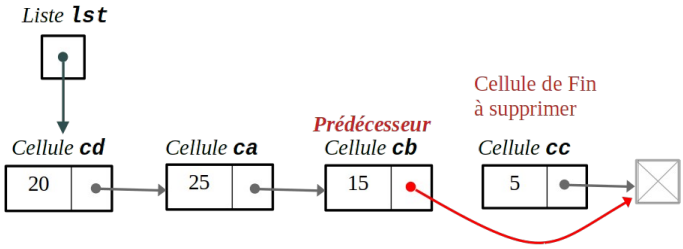
Le coût de la méthode **supprimer()** est

_____

3.3 - Supprimer la cellule en Fin de liste chaînée (À SAVOIR RECREER)

<u>Cellule à localiser</u> : le prédécesseur de la Fin (si elle existe).

Puisque la méthode n'aura rien à renvoyer, nous allons pouvoir en profiter pour lui faire renvoyer la valeur qu'elle vient de supprimer.



VOIR VOTRE COURS et le CODE page 4.

Le coût de la méthode `supprimer_fin()` est _____

Par contre, même en mémoriser la cellule de fin, _____

4 – Des listes chaînées plus compliquées ?

5 – Concaténation fortement mutable ? Bonne idée ou pas ?

Code-source fondamental du MODULE
LISTE CHAINEE MUABLE (tête) [modèle objet]

```
class Cellule:

    def __init__(self, valeur, successeur):
        self.v = valeur
        self.s = successeur

    def nb_cellules(self) :
        if self.s == None:
            return 1
        else:
            return 1 + self.s.nb_cellules()

    def contenu(self) :
        return self.v

    def successeur(self):
        return self.s


class Liste:

    def __init__(self):
        self.tete = None

    def est_liste_vide(self):
        return self.tete == None

    def acces_tete(self):
        return self.tete

    def acces_fin(self):
        c = self.tete
        while c.s != None:
            c = c.s
        return c

    def acces(self, i):
        c = self.tete
        for _ in range(i):
            c = c.s
        return c

    def inserer_tete(self, elt'):
        nouvelle = Cellule(elt, None)
        nouvelle.s = self.tete
        self.tete = nouvelle

    def inserer_fin(self, elt):
        if self.est_liste_vide():
            self.inserer_tete(elt)
        else:
            nouvelle = Cellule(elt, None)
            ancienne_fin = self.acces_fin()
            ancienne_fin.s = nouvelle

    def inserer(self, elt, i):
        if i == 0:
            self.inserer_tete(elt)
        else:
            pred = self.acces(i-1)
            nouvelle = Cellule(elt, None)
            nouvelle.s = pred.s
            pred.s = nouvelle

    def supprimer_tete(self):
        ancienne_valeur = self.tete.v
        self.tete = self.tete.s
        return ancienne_valeur

    def supprimer_fin(self):
        if self.tete.s is None:
            return self.supprimer_tete()
        else :
            c = self.tete
            while c.s.s != None:
                c = c.s
            ancienne_valeur = c.s.v
            c.s = None
            return ancienne_valeur

    def supprimer(self, i):
        if i == 0:
            return self.supprimer_tete()
        else:
            pred = self.acces(i-1)
            ancienne_valeur = pred.s.v
            pred.s = pred.s.s
            return ancienne_valeur

    def longueur(self)
        if self.est_liste_vide():
            return 0
        else:
            return self.tete.nb_cellules()

    def premier(self):
        return self.tete.v

    def dernier(self):
        return self.acces_fin().v

    def ieme(self, i):
        return self.acces(i).v
```