CC2.0 BY SA – www.infoforall.fr

Données 14 - Implémentation du type LISTE CHAINEE via un objet

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

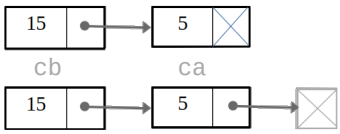
1.1 - Implémentation cellules-objets muables : les Cellules

Les **Cellules** sont des instances de **`Cellule`** possédant 2 attributs : **`v`** pour la valeur, et **`s`** pour le successeur.
L'attribut **`s`** est alors une référence menant : → soit à **`None`** s'il s'agit de la Cellule de Fin.
→ soit à une **`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 not self.s is None   # ou return self.s is not None

    def est_fin(self:'Cellule') -> bool:
        return self.s is None      # ou même return not self.a_successeur()
```

Exemples d'instanciation :

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



Ci-contre, deux représentations mentales qu'on peut se faire d'une cellule contenant 15 menant à une cellule menant à 5.

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

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

1.2 - Implémentation cellules-objets muables : la Liste

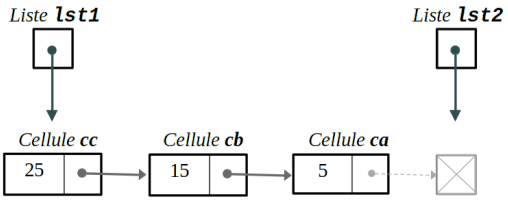
La **`Liste`** est une instance d'une classe **`Liste`** possédant un attribut **`tete`** faisant référence :

1. soit à **`None`** pour indiquer qu'il s'agit d'une liste vide.
2. soit à la **`Cellule` en tête** de liste.

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

    def est_liste_vide(self:'Liste') -> bool:
        return 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 - Implémentation cellules-objets muables : un exemple

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

4       def acces(self:'Liste NON VIDE', i:'int VALIDE') -> Cellule:
5           '''Renvoie l'adresse de la cellule i dans la liste NON VIDE'''
6           c = self.acces_tete()  # récupère la cellule de tête
7           for _ in range(i):     # Faire i fois
8               c = c.s            # passe au successeur
9           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:
         # ...
1        def longueur(self:'Liste') -> int:
2            '''Renvoie la longueur de la liste'''
3            if self.est_liste_vide():
4                return 0
5            else:
6                c = self.tete  _____
7                nb = 1
8                while c.s is not None:  _____
9                    nb = nb + 1
10                   c = c.s  _____
11               return nb
```

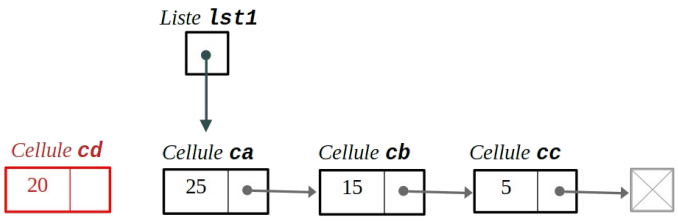
Version récursive : la méthode de liste va faire appel à la méthode récursive de Cellule

```
    class Cellule:
        # ...
1       def nb_cellules(self:'Cellule') -> int:
2           '''Renvoie le nombre de cellules dans la chaîne à partir de cette cellule'''
3           if self.s is None:
4               return 1                          # _____
5           else:
6               return 1 + self.s.nb_cellules()   # _____
                                                  # _____

    class Liste:
        # ...
7       def longueur(self:'Liste') -> int:
8           '''Renvoie la longueur de la liste'''
9           if self.est_liste_vide():
10              return 0                          # _____
11          else:
12              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.

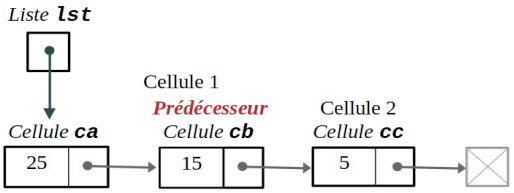


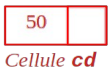
```
    class Liste:
        # ...
1       def inserer_tete(self:'Liste', elt:'Element') -> None:
2           '''Rajoute une nouvelle cellule contenant elt en tête de liste'''
3           nouvelle = Cellule(elt, None)   # Etape 1 : création de la cellule
4           nouvelle.s = __________         #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

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

2.2 - Insérer une cellule en position i dans notre liste chaînée (À SAVOIR RECREER)

Cellule à localiser : la Cellule d'indice i-1 (si elle existe) pour faire de son successeur la nouvelle Cellule i.



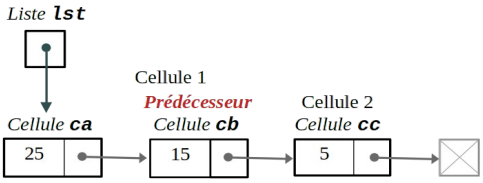


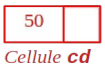

```
    class Liste:
        # ...
1       def inserer(self:'Liste', elt:'Element', i:'int VALIDE') -> None:
2           '''Rajoute une nouvelle cellule contenant elt en position i VALIDE de lst NON VIDE'''
3           if i == 0:
4               self.inserer_tete(elt)
5           else:
6               pred = ____________________     # Etape 1 : recherche de la cellule i-1
7               nouvelle = Cellule(elt, None)   # Etape 2 : création de la cellule
8               ___________ = _____________     # Etape 3 : liasion nouvelle i vers ancienne i
9               ___________ = _____________     # 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 pour faire de son successeur la nouvelle Cellule.






```
    class Liste:
        # ...
1       def inserer_fin(self:'Liste', elt:'Element') -> None:
2           '''Rajoute une nouvelle cellule de Fin contenant elt dans la liste'''
3           if self.est_liste_vide():
4               self.inserer_tete(elt)
5           else:
6               pred = _________________                # Etape 1 : recherche de la cellule de Fin
7               while _____________ is not None:        # TANT QUE pred n'est pas la fin actuelle
8                   pred = _____________                # on passe à la cellule suivante
9               nouvelle = Cellule(elt, None)           # Etape 2 : création de la cellule
10              _____________ = _____________           # Etape 3 : liaison ancienne Fin vers nouvelle
```

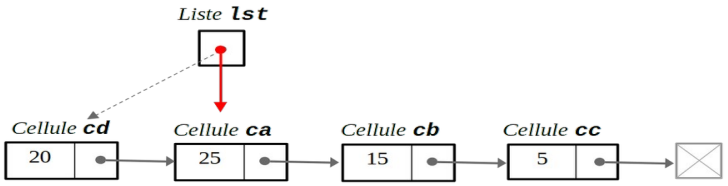
Le coût de la méthode **inserer_fin()** est _____

Par contre,

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,** pour trouver son successeur et en faire la nouvelle Tête.



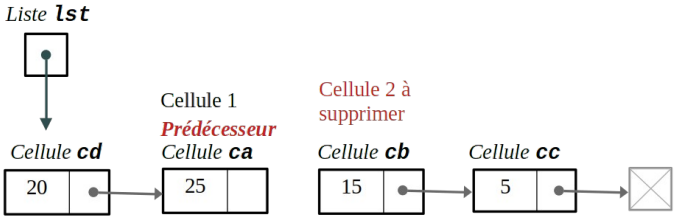
Puisque la méthode n'aura rien à renvoyer, nous allons pouvoir en profiter pour lui faire renvoyer la valeur qu'elle vient de supprimer.

VOIR LA METHODE SUR VOTRE COURS.

Le coût de la méthode **`suppimer_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).



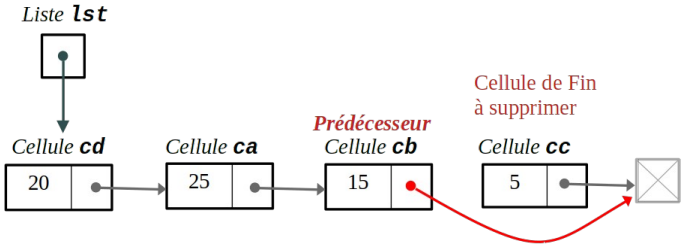
Puisque la méthode n'aura rien à renvoyer, nous allons pouvoir en profiter pour lui faire renvoyer la valeur qu'elle vient de supprimer.

VOIR LA METHODE SUR VOTRE COURS.

Le coût de la méthode **`suppimer()`** `est` _____

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

Cellule à localiser : **le prédecesseur 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 LA METHODE SUR VOTRE COURS.
Le coût de la méthode **`suppimer_fin()`** `est` _____
Par contre,

4 – Des listes chaînées plus compliquées ?

VOIR VOTRE COURS

5 – Concaténation fortement muable ? Bonne idée ou pas ?

VOIR VOTRE COURS.

Données 14 – Code-source fondamental du MODULE
LISTE CHAINEE BASEE SUR UN OBJET STOCKANT LA CELLULE DE TETE

```
class Cellule:
    '''Cellule (valeur, successeur)'''

    def __init__(self, valeur:'Element', successeur:'Cellule|None'):
        self.v = valeur
        self.s = successeur

    def nb_cellules(self:'Cellule') -> int:
        '''Renvoie le nombre de cellules dans la chaîne à partir de cette cellule'''
        if self.s is None:
            return 1
        else:
            return 1 + self.s.nb_cellules()

    def contenu(self:'Cellule') -> 'Element':
        '''Renvoie le contenu de la Cellule'''
        return self.v

    def successeur(self:'Cellule NON FIN') -> 'Cellule':
        '''Renvoie la Cellule qui succède à une cellule NON FIN'''
        return self.s



class Liste:
    '''Liste chaînée MUABLE Tete'''

    def __init__(self):
        self.tete = None

    def est_liste_vide(self:'Liste') -> bool:
        '''Prédicat qui renvoie True si la liste est vide'''
        return self.tete is None

    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_fin(self:'Liste NON VIDE') -> Cellule:
        '''Renvoie l'adresse de la cellule de Fin dans la liste NON VIDE'''
        c = self.tete
        while c.s is not None:
            c = c.s
        return c

    def acces(self:'Liste NON VIDE', i:'int VALIDE') -> Cellule:
        '''Renvoie l'adresse de la cellule i dans la liste NON VIDE'''
        c = self.tete
        for _ in range(i):
            c = c.s
        return c

    def inserer_tete(self:'Liste', elt:'Element') -> None:
        '''Rajoute une nouvelle cellule contenant elt en tête de liste'''
        nouvelle = Cellule(elt, None)
        nouvelle.s = self.tete
        self.tete = nouvelle

    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:
        nouvelle = Cellule(elt, None)
        ancienne_fin = self.acces_fin()
        ancienne_fin.s = nouvelle

def inserer(self:'Liste', elt:'Element', i:'int VALIDE') -> None:
    """Rajoute une cellule contenant elt en position i VALIDE de la liste"""
    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:'Liste NON VIDE') -> 'Elt':
    """Supprime la tête et renvoie son contenu"""
    memoire = self.tete.v
    self.tete = self.tete.s
    return memoire

def supprimer_fin(self:'Liste NON VIDE') -> 'Element':
    '''Supprime la cellule de fin d'une liste NON VIDE et renvoie l'élément'''

    if self.tete.s is None:
        return self.supprimer_tete()
    else :
        c = self.tete
        while c.s.s is not None:
            c = c.s
        ancienne_valeur = c.s.v
        c.s = None
        return ancienne_valeur

def supprimer(self:'Liste NON VIDE', i:'int VALIDE') -> 'Element':
    """Supprime la cellule i VALIDE et renvoie l'élément qui y était stocké"""
    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:'Liste') -> int:
    '''Renvoie la longueur de la liste'''
    if self.est_liste_vide():
        return 0
    else:
        return self.tete.nb_cellules()

def premier(self:'Liste NON VIDE') -> 'Element':
    return self.tete.v

def dernier(self:'Liste NON VIDE') -> 'Element':
    return self.acces_fin().v

def ieme(self:'Liste NON VIDE', i:'int VALIDE') -> 'Element':
    return self.acces(i).v
```