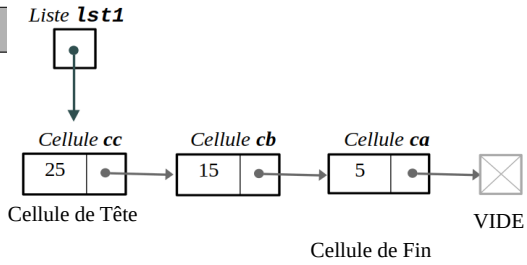
Données 12 - Implémentation d'une LISTE CHAINEE tuple (Tête, Queue)

I – Implémentation en tuple (tête, queue)

On travaille sur l'idée d'une **Liste Chaînée** composée de **Cellules**.



En implémentant :
`Cellule` comme un **tuple** (couple) et
`Liste` comme un **alias vers la Cellule de Tête**, on obtient :
Exemple : `lst1 = cc`

En se ramenant aux types :

```
Successeur = Cellule | ()                Liste = Liste NON VIDE | Liste VIDE
Cellule = ( Elément, Successeur  )       Liste =    Cellule     |     ()
        = ( Elément,  Cellule|() )
```

On voit immédiatement que, dans cette implémentation, il n'y aura pas de différence de traitement entre :
`Liste NON VIDE` et **`Cellule`** et **`Liste`** et **`Successeur`**

Voici les <u>primitives</u> liées à cette Liste Chaînée tuple (tête, queue) :

Programme extérieur ⟺ PRIMITIVES ⟺ Les données de la Liste Chaînée

```
nouvelle_liste_vide() → Liste VIDE
est_liste_vide(lst:Liste) → bool
inserer_tete(lst:Liste, elt:Elément) → Liste NON VIDE
supprimer_tete(lst:Liste NON VIDE) → Liste
---
acces_tete(lst:Liste NON VIDE) → Cellule
contenu(c:Cellule) → Elément
successeur(c:Cellule) → Cellule|VIDE
---
premier(lst:Liste NON VIDE) → Elément
```

IMMUABLE puisque l'insertion et la suppression renvoient une nouvelle liste.

```
1  def nouvelleListe() -> 'Liste NON VIDE':                          # coût constant
2      '''Renvoie une liste vide'''
3      return ()
4
5  def est_liste_vide(lst:'Liste') -> bool:                          # coût constant
6      '''Prédicat qui renvoie True si la liste est vide'''
7      return lst == ()
8
9  def inserer_tete(lst:'Liste', elt:'Element') -> 'Liste NON VIDE':  # coût constant
10     '''Renvoie une nouvelle liste où elt est la Tête menant à lst devenue la queue'''
11     return (elt, lst)
12
13 def supprimer_tete(lst:'Liste NON VIDE') -> 'Liste':               # coût constant
14     '''Renvoie une liste obtenue en supprimant la tête de la liste NON VIDE lst'''
15     return lst[1]
16
17 def acces_tete(lst:'Liste NON VIDE') -> 'Cellule':                 # coût constant
18     '''Renvoie la Cellule de Tête de la liste NON VIDE lst'''
19     return lst
20
21 def contenu(c:'Cellule') -> 'Element':                             # coût constant
22     '''Renvoie le contenu de la cellule'''
23     return c[0]
24
25 def successeur(c:'Cellule') -> 'Cellule|VIDE':                     # coût constant
26     '''Renvoie la cellule qui succède à c, ou VIDE si c est '''
27     return c[1]
28
29 def premier(lst:'Liste NON VIDE') -> 'Element':                    # coût constant
30     '''Renvoie le contenu de la tête de la liste NON VIDE lst'''
31     return contenu(acces_tete(lst))  # voir juste lst[0] si vue comme une primitive
```

II – Création d'autres fonctions

Les **primitives** sont les seules qui **manipulent directement la structure interne.**
Toutes les **autres fonctions manipulent les primitives** et <u>pas</u> la structure interne (Elément, Successeur).

```python
def a_successeur(lst:'Liste') -> bool:                                  # Coût constant
    '''Prédicat qui renvoie True si la Tête de Liste a une Cellule qui lui succède'''
    return not(est_liste_vide(lst) or est_liste_vide(supprimer_tete(lst)))

def longueur(lst:'Liste') -> int:                                       # Coût linéaire, en θ(n)
    '''Renvoie le nombre d'éléments dans la liste lst'''
    if est_liste_vide(lst):                                     # Cas de base de la liste vide
        return 0
    else:                                                       # Cas récursif
        return 1 + longueur(supprimer_tete(lst))

def acces(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Cellule':           # Coût linéaire θ(i) O(n)
    '''Renvoie la Cellule de position i VALIDE dans la liste NON VIDE lst'''
    if i == 0:                                      # Cas de base
        return acces_tete(lst)
    else:                                           # Cas récursif
        return acces(supprimer_tete(lst), i-1)

def ieme(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Element':            # Coût linéaire θ(i) O(n)
    '''Renvoie le contenu de la Cellule de position i VALIDE dans la liste NON VIDE lst'''
    return contenu(acces(lst, i))
    for _ in range(i):                                  # Version ALTERNATIVE sans contenu et acces
        lst = supprimer_tete(lst)
    return premier(lst)

def inserer(lst:'Liste', elt:'Element', i:'int VALIDE') -> 'Liste NON VIDE':   # Lin. θ(i) O(n)
    '''Renvoie une nouvelle liste où on a inséré elt en tant qu'élément de position i'''
    # 1 - Suppression et mémorisation des i éléments de 0 à i-1
    memoire = nouvelle_liste_vide()
    for _ in range(i):                                  # on réalise le bloc i fois
        v = premier(lst)                                # on lit la valeur de Tête
        memoire = inserer_tete(memoire, v)              # on stocke en tête de mémoire
        lst = supprimer_tete(lst)                       # on supprime une tête (lst devient locale)
    # 2 - Rajout du nouvel élément
    lst = inserer_tete(lst, elt)
    # 3 - Remise en place des i éléments précédents
    for _ in range(i):                                  # on réalise le bloc i fois
        v = premier(memoire)                            # on lit la valeur de Tête de mémoire
        lst = inserer_tete(lst, v)                      # on la rajoute en tête de lst
        memoire = supprimer_tete(memoire)               # on supprime cette valeur de la mémoire
    return lst

def supprimer(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Liste':         # Coût linéaire θ(i) O(n)
    '''Renvoie une liste obtenue en supprimant l'élément d'indice i VALIDE dans lst NON VIDE'''
    # 1 - Suppression et mémorisation des éléments de 0 à i-1
    memoire = nouvelle_liste_vide()
    for _ in range(i):                                  # on réalise le bloc i fois
        v = premier(lst)                                # on lit la valeur en Tête
        memoire = inserer_tete(memoire, v)              # on stocke en tête de mémoire
        lst = supprimer_tete(lst)                       # on supprime une tête (lst devient locale)
    # 2 -Suppression de cet élément
    lst = supprimer_tete(lst)
    # 3 - Remise en place des éléments précédents
    for _ in range(i):                                  # on réalise le bloc i fois
        v = premier(memoire)                            # on lit la valeur de Tête de la mémoire
        lst = inserer_tete(lst, v)                      # on rajoute en tête de lst
        memoire = supprimer_tete(memoire)               # on supprime de la mémoire
    return lst

def concatener(gauche:'Liste', droite:'Liste') -> 'Liste':          # Coût linéaire θ(nb_gauche)
    '''Renvoie une nouvelle liste commençant par la tête de gauche vers la fin de droite'''
    memoire = nouvelle_liste_vide()
    while not est_liste_vide(gauche):                   # Etape 1 : on inverse gauche dans memoire
        v = premier(gauche)                             # on lit la valeur en Tête de gauche
        memoire = inserer_tete(memoire, v)              # on stocke en tête de memoire
        gauche = supprimer_tete(gauche)                 # on supprime cette tête dans gauche
    while not est_liste_vide(memoire):                  # Etape 2 : on place memoire dans droite
        v = premier(memoire)                            # on lit la valeur en Tête de memoire
        droite = inserer_tete(droite, v)                # on stocke en tête de droite
        memoire = supprimer_tete(memoire)               # on supprime cette tête dans memoire
    return droite

def rechercher(lst:'Liste', elt:'Element') -> int:                      # Coût linéaire O(n)
    '''Renvoie la position éventuelle de elt dans lst, -1 si non trouvé'''
    position = 0
    while not est_liste_vide(lst):          # Tant que la liste n'est pas vide
        if premier(lst) == elt:             # Si l'élément en tête est l'élément cherché
            return position                 # on répond en fournissant sa position
        else:
            lst = supprimer_tete(lst)       # on supprime la tête
            position = position + 1         # on incrémente pour mémoriser qu'on avance
    return -1
```