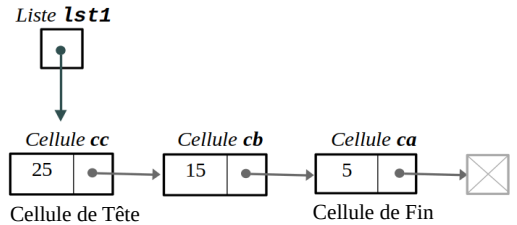


## Données 12 - Implémentation d'une LISTE CHAÎNÉE (Tête, Queue)



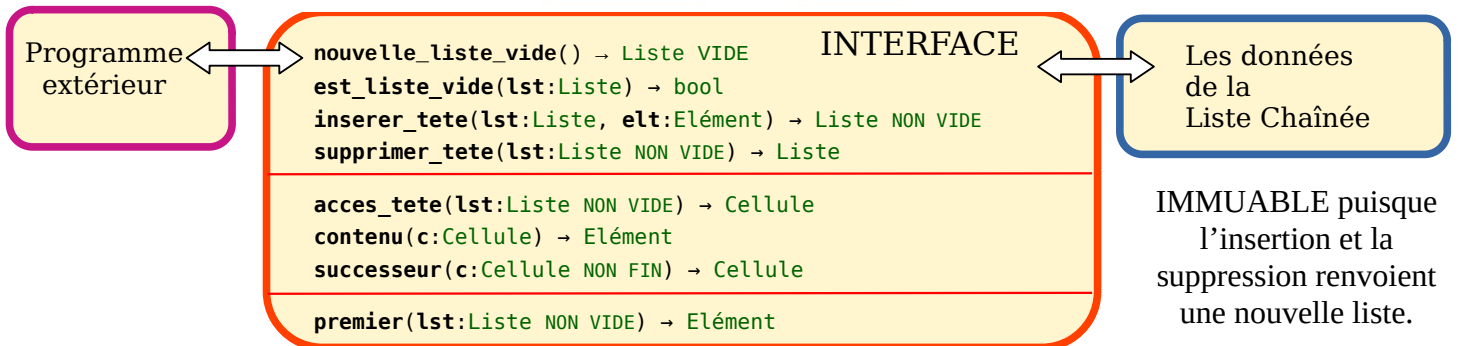
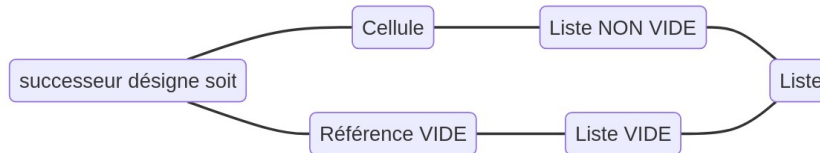
### I - Implémentation en tuple (tête, queue)

On travaille sur l'idée d'une **Liste Chaînée** composée de Cellules.  
 Une **Cellule** est un couple (2-uplet) (**Élément, Successeur**).  
 avec Successeur = Cellule | ()



Une **Liste VIDE** est implémentée comme un **tuple vide ()**  
 Une **Liste NON VIDE** est implémentée comme un alias de la **Cellule** de Tête. **Exemple : lste1 = cc**  
 Une **Liste** est **Liste VIDE|Liste NON VIDE**.

En implémentant **VIDE ≈ Liste VIDE**, il n'y a pas de différence en terme de traitement de données entre :  
**Liste NON VIDE** et **Cellule** (car une Liste NON VIDE contient l'adresse de sa Cellule de tête).  
**Liste** et **Successeur** :



```

1  def nouvelleListe() -> 'Liste NON VIDE':                               # cout constant
2  '''Renvoie une liste vide'''
3  return ()
4
5  def est_liste_vide(lst:'Liste') -> bool:                             # cout 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':   # cout 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':                 # cout 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':                  # cout constant
18 '''Renvoie la Cellule de Tête de la liste NON VIDE lst'''
19 return lst
20
21 def contenu(c:'Cellule') -> 'Element':                              # cout constant
22 '''Renvoie le contenu de la cellule'''
23 return c[0]
24
25 def successeur(c:'Cellule NON FIN') -> 'Cellule':                  # cout constant
26 '''Renvoie la cellule qui succède à c, c NE DOIT PAS être la DERNIERE Cellule de la liste'''
27 return c[1]
28
29 def premier(lst:'Liste NON VIDE') -> 'Element':                     # cout 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 précédentes** sont les seules qui **manipulent directement la structure interne**.

Toutes les **autres fonctions manipulent les primitives** et pas la structure interne (Élément, Successeur).

```

1  def a_successeur(lst:'Liste') -> bool:                                # Coût constant
2  '''Prédicat qui renvoie True si la Tête de Liste a une Cellule qui lui succède'''
3  return not(est_liste_vider(lst) or est_liste_vider(supprimer_tete(lst)))
4
5  def longueur(lst:'Liste') -> int:                                    # Coût linéaire, en  $\theta(n)$ 
6  '''Renvoie le nombre d'éléments dans la liste lst'''
7  if est_liste_vider(lst):                                          # Cas de base de la liste vide
8      return 0
9  else:                                                            # Cas récursif
10     return 1 + longueur(supprimer_tete(lst))
11
12 def acces(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Cellule':      # Coût linéaire  $\theta(i)$   $0(n)$ 
13 '''Renvoie la Cellule de position i VALIDE dans la liste NON VIDE lst'''
14 if i == 0:                                                        # Cas de base
15     return acces_tete(lst)
16 else:                                                            # Cas récursif
17     return acces(supprimer_tete(lst), i-1)
18
19 def ieme(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Element':      # Coût linéaire  $\theta(i)$   $0(n)$ 
20 '''Renvoie le contenu de la Cellule de position i VALIDE dans la liste NON VIDE lst'''
21 return contenu(acces(lst, i))
22 for _ in range(i):                                              # Version ALTERNATIVE sans contenu et acces
23     lst = supprimer_tete(lst)
24 return premier(lst)
25
26 def inserer(lst:'Liste', elt:'Element', i:'int VALIDE') -> 'Liste NON VIDE': # Lin.  $\theta(i)$   $0(n)$ 
27 '''Renvoie une nouvelle liste où on a inséré elt en tant qu'élément de position i'''
28 # 1 - Suppression et mémorisation des i éléments de 0 à i-1
29 memoire = nouvelle_liste_vider()
30 for _ in range(i):                                             # on réalise le bloc i fois
31     v = premier(lst)                                           # on lit la valeur de Tête
32     memoire = inserer_tete(memoire, v)                          # on stocke en tête de mémoire
33     lst = supprimer_tete(lst)                                   # on supprime une tête (lst devient locale)
34 # 2 - Rajout du nouvel élément
35 lst = inserer_tete(lst, elt)
36 # 3 - Remise en place des i éléments précédents
37 for _ in range(i):                                             # on réalise le bloc i fois
38     v = premier(memoire)                                       # on lit la valeur de Tête de mémoire
39     lst = inserer_tete(lst, v)                                  # on la rajoute en tête de lst
40     memoire = supprimer_tete(memoire)                          # on supprime cette valeur de la mémoire
41 return lst
42
43 def supprimer(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Liste':   # Coût linéaire  $\theta(i)$   $0(n)$ 
44 '''Renvoie une liste obtenue en supprimant l'élément d'indice i VALIDE dans lst NON VIDE'''
45 # 1 - Suppression et mémorisation des éléments de 0 à i-1
46 memoire = nouvelle_liste_vider()
47 for _ in range(i):                                             # on réalise le bloc i fois
48     v = premier(lst)                                           # on lit la valeur en Tête
49     memoire = inserer_tete(memoire, v)                          # on stocke en tête de mémoire
50     lst = supprimer_tete(lst)                                   # on supprime une tête (lst devient locale)
51 # 2 -Suppression de cet élément
52 lst = supprimer_tete(lst)
53 # 3 - Remise en place des éléments précédents
54 for _ in range(i):                                             # on réalise le bloc i fois
55     v = premier(memoire)                                       # on lit la valeur de Tête de la mémoire
56     lst = inserer_tete(lst, v)                                  # on rajoute en tête de lst
57     memoire = supprimer_tete(memoire)                          # on supprime de la mémoire
58 return lst
59
60 def concatener(gauche:'Liste', droite:'Liste') -> 'Liste':       # Coût linéaire  $\theta(\text{nb\_gauche})$ 
61 '''Renvoie une nouvelle liste commençant par la tête de gauche vers la fin de droite'''
62 memoire = nouvelle_liste_vider()
63 while not est_liste_vider(gauche):                              # Etape 1 : on inverse gauche dans memoire
64     v = premier(gauche)                                        # on lit la valeur en Tête de gauche
65     memoire = inserer_tete(memoire, v)                        # on stocke en tête de memoire
66     gauche = supprimer_tete(gauche)                          # on supprime cette tête dans gauche
67 while not est_liste_vider(memoire):                            # Etape 2 : on place memoire dans droite
68     v = premier(memoire)                                       # on lit la valeur en Tête de memoire
69     droite = inserer_tete(droite, v)                          # on stocke en tête de droite
70     memoire = supprimer_tete(memoire)                        # on supprime cette tête dans memoire
71 return droite
72
73 def rechercher(lst:'Liste', elt:'Element') -> int:               # Coût linéaire  $0(n)$ 
74 '''Renvoie la position éventuelle de elt dans lst, -1 si non trouvé'''
75 position = 0
76 while not est_liste_vider(lst):                                # Tant que la liste n'est pas vide
77     if premier(lst) == elt:                                    # Si l'élément en tête est l'élément cherché
78         return position                                       # on répond en fournissant sa position
79     else:
80         lst = supprimerTete(lst)                              # on supprime la tête
81         position = position + 1                                # on incrémente pour mémoriser qu'on avance
82 return -1

```