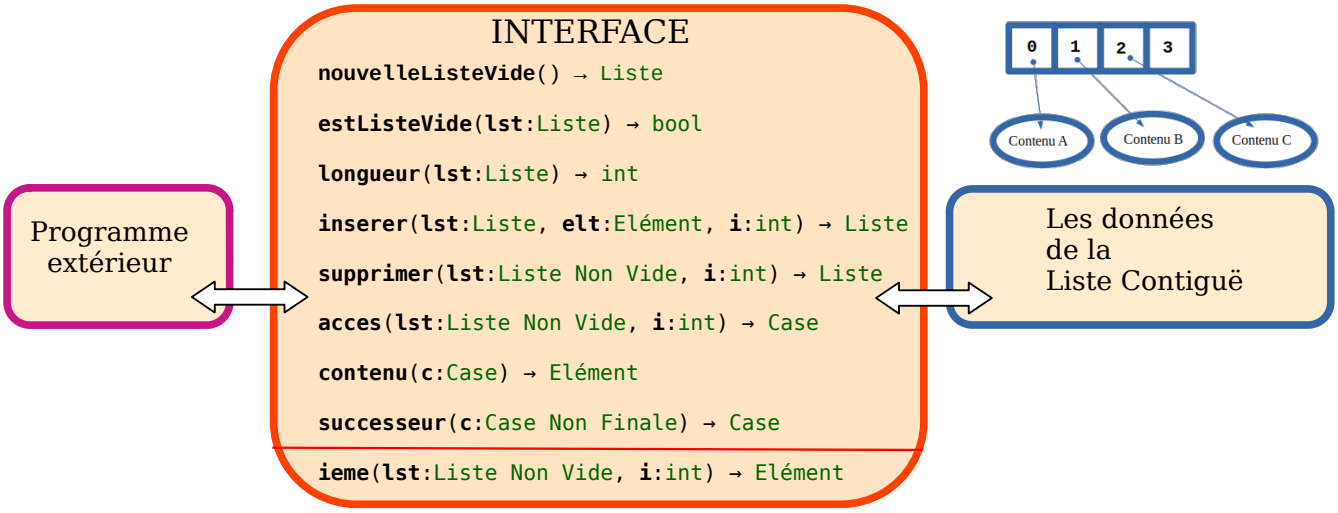
Implémentation du type abstrait LISTE CONTIGUE *(Données 13)*

I – Implémentation immuable basée sur des tableaux statiques

1.1 Création par compréhension

```
t = [0 for x in range(5)]
t1 = [0 for x in range(len(t)+1)]
t2 = [0 for x in range(len(t)-1)]
```

1.2 Primitives de la Liste Contiguë (version immuable)




```
01  def nouvelleListe() -> 'Liste':                                    # coût constant
02      '''Renvoie une liste vide'''
03       return []
04
05  def estListeVide(lst:'Liste') -> bool:                             # coût constant
06      '''Prédicat qui renvoie True si la liste est vide'''
07      return lst == []
08
09  def longueur(lst:'Liste') -> int:                                  # coût constant
10      '''Renvoie le nombre d'éléments stockés dans lst'''
11      return len(lst)
12
13  def ieme(lst:'Liste Non Vide', i:int) -> 'Element':                # coût constant
14      '''Renvoie le contenu de la Case de position i VALIDE dans la liste NON VIDE lst'''
15      return lst[i]
16
17  def inserer(lst:'Liste', elt:'Elément', i:int) -> 'Liste':         # coût linéaire en θ(n)
18     '''Renvoie une Liste basée sur lst où elt est en position VALIDE i'''
19      nb = len(lst) + 1
20      nouveau = [None for _ in range(nb)]   # 1 - nouveau tableau avec une case en plus
21      for k in range(0, i):                 # 2 - copie les elts de de 0 à i-1 sans décalage
22          nouveau[k] = lst[k]
23      for k in range(i, len(lst)):          # 3 - décalage à droite des elts d'indice i et plus
24          nouveau[k+1] = lst[k]
25      nouveau[i] = elt                      # 4 - elt à la position i
26      return nouveau
27
28  def supprimer(lst:'Liste Non Vide', i:int) -> 'Liste':             # coût linéaire en θ(n)
29      '''Renvoie une Liste basée sur lst NON VIDE en supprimant la case i VALIDE'''
30      nb = len(lst) - 1
31      nouveau = [None for _ in range(nb)]   # 1 - nouveau tableau avec une case en moins
32      for k in range(0, i):                 # 2 - copie les eltss de 0 à i-1 sans décalage
33          nouveau[k] = lst[k]
34      for k in range(i+1, len(lst)):        # 3 - décalage à gauche des elts d'indice i+1 et plus
35          nouveau[k-1] = lst[k]
36      return nouveau
37
38  def acces(lst:'Liste Non Vide', i:int) -> 'Case':                  # coût constant
39      '''Renvoie la référence de la case d'indice i VALIDE dans la liste lst NON VIDE'''
40      return (lst, i)                       # on crée un tuple-case (quelle liste ?, quel indice ?)
```

```python

def contenu(c:'Case') -> 'Elément':                                    # coût constant
    '''Renvoie le contenu de la case valide fournie'''
    lst = c[0]                      # on récupère la liste contenant notre case
    i = c[1]                        # on récupère l'indice de notre case
    return lst[i]                   # on renvoie l'élément stocké à cet endroit

def successeur(c:'Case Non Finale') -> 'Case':                         # coût constant
    '''Renvoie le case qui succède à la case fournie (qui ne doit pas être la dernière)'''
    lst = c[0]                      # on récupère la liste contenant notre case
    i = c[1]                        # on récupère l'indice de notre case
    return (lst, i+1)               # on crée un tuple-case correspond à la case suivante

def precedent(c:'Case Non Tete') -> 'Case':                            # coût constant
    '''Renvoie le case qui précéde la case fournie (qui ne doit pas être la première)'''
    lst = c[0]                      # on récupère la liste contenant notre case
    i = c[1]                        # on récupère l'indice de notre case
    return (lst, i-1)               # on crée un tuple-case correspond à la case précédente
```

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 (Elément, Successeur). MAIS parfois, on peut se rendre compte après étude qu'il vaut mieux créer une nouvelle primitive, plus efficace.

```python
def insererTete(lst:'Liste', elt:'Elément') -> 'Liste':                # coût linéaire en θ(n)
    '''Renvoie une Liste basée sur lst où elt est en Tête'''
    return inserer(lst, elt, 0)

def supprimerTete(lst:'Liste Non Vide') -> 'Liste':                    # coût linéaire en θ(n)
    '''Renvoie une Liste basée sur lst NON VIDE en supprimant la Tête'''
    return supprimer(lst, 0)

def accesTete(lst:'Liste Non Vide') -> 'Case':                         # coût constant
    '''Renvoie la référence de la case de tête dans lst NON VIDE'''
    return acces(lst, 0)

def insererFin(lst:'Liste', elt:'Elément') -> 'Liste':                 # coût linéaire en θ(n)
    '''Renvoie une Liste basée sur lst où elt est en Fin'''
    return inserer(lst, elt, longueur(lst))

def supprimerFin(lst:'Liste') -> 'Liste':                              # coût linéaire en θ(n)
   '''Renvoie une Liste basée sur lst NON VIDE en supprimant la fin'''
    return supprimer(lst, longueur(lst) - 1)

def accesFin(lst:'Liste') -> 'Case':                                   # Coût constant
    '''Renvoie la référence de la case de fin dans lst NON VIDE'''
   return acces(lst, longueur(lst) - 1)

def premier(lst:'Liste Non Vide') -> 'Element':                        # Coût constant
    '''Renvoie la valeur de tête de lst NON VIDE'''
    return ieme(lst, 0)

def dernier(lst:'Liste Non Vide') -> 'Element':                        # Coût constant
   '''Renvoie la valeur de fin de lst NON VIDE'''
   return ieme(lst, longueur(lst) - 1)

def concatener(gauche:'Liste', droite:'Liste') -> 'Liste':             # Coût quadratique !
    '''Renvoie une nouvelle liste commençant par la tête de gauche vers la fin de droite'''
    nouveau = nouvelleListe()           # 1 - nouvelle liste pour la concaténation
    ng = longueur(gauche)
    for i in range(ng):                 # 2 - on copie les éléments de gauche
        nouveau = inserer(nouveau, ieme(gauche, i), i)
    nd = longueur(droite)
    for i in range(nd):                 # 3 - on copie les éléments de droite
        nouveau = inserer(nouveau, ieme(droite, i), ng + i)
    return nouveau

def concatener_version_primitive(gauche:'Liste', droite:'Liste') -> 'Liste':   # Coût linéaire
    '''Renvoie une nouvelle liste commençant par la tête de gauche vers la fin de droite'''
    nb = len(gauche) + len(droite)
    nouveau = [None for _ in range(nb)]  # 1 - nouveau tableau avec autant de cases que g et d
    for i in range(0, len(gauche)):      # 2 - on copie les éléments de gauche
        nouveau[i] = gauche[i]
    ng = len(gauche)                     # on récupère le nb d'éléments à gauche
    for i in range(0, len(droite)):      # 3 - on copie les éléments de droite
        nouveau[ng+i] = droite[i]
    return nouveau
```

Bref, des performances pas folles...

III – Type list de Python

La structure de données natives list est basée sur un **tableau dynamique muable**.

L'idée générale est d'avoir un tableau statique plus grand que nécessaire de façon à pouvoir « rajouter » ou « supprimer » à coût constant des éléments en (fausse) fin de tableau.

Si on résume :

- **`append()`** : un coût amorti constant pour l'insertion en fin.
- **`pop()`** : un coût **constant** pour la suppression en fin.
- **`insert(v, i)`** : un **coût linéaire** pour l'insertion, **le pire des cas étant en i = 0**.
- **`pop(i)`** : un **coût linéaire** pour la suppression, **le pire des cas étant en i = 0**.

La list Python n'est donc pas une Liste Chaînée.

ATTENTION : **`append() renvoie None et pop() renvoie la valeur supprimée. TYPE MUABLE.`**

```
01  def nouvelleListe() -> 'Liste':                                    # coût constant
02      '''Renvoie une liste vide'''
03      return []
04
05  def estListeVide(lst:'Liste') -> bool:                             # coût constant
06      '''Prédicat qui renvoie True si la liste est vide'''
07      return lst == []
08
09  def longueur(lst:'Liste') -> int:                                  # coût constant
10      '''Renvoie le nombre de cases utilisées dans le tableau'''
11      return len(lst)
12
13  def ieme(lst:'Liste Non Vide', i:int) -> 'Element':                # coût constant
14      '''Renvoie la valeur stockée à l'indice VALIDE voulu de lst NON VIDE'''
15      return lst[i]
16
17  def insererFin(lst:'Liste', elt:'Elément') -> None:                # coût constant
18      '''Modifie en place la liste en insérant l'élément en fin de liste'''
19      lst.append(elt)
20
21  def supprimerFin(lst:'Liste Non Vide') -> 'Element':               # coût constant
22      '''Renvoie l'élément de fin supprimé dans lst NON VIDE'''
23      return lst.pop()
24
25  def inserer(lst:'Liste', elt:'Elément', i:int) -> None:            # coût linéaire en θ(n-i)
26      '''Modifie en place la liste en insérant l'élément à l'indice fourni'''
27      lst.insert(i, elt)
28
29  def supprimer(lst:'Liste Non Vide', i:int) -> 'Elément':           # coût linéaire en θ(n-i)
30      '''Renvoie l'élément supprimé à l'indice VALIDE dans lst NON VIDE'''
31      return lst.pop(i)
32
33  def concatener(gauche:'Liste', droite:'Liste') -> 'Liste':         # coût linéaire en θ(ng+nd)
34      nouveau = nouvelleListe()
35      for i in range(longueur(gauche)):
36          nouveau.insererFin(ieme(gauche, i))
37      for i in range(longueur(droite)):
38          nouveau.insererFin(ieme(droite, i))
39      return nouveau
40
41  def concatener_primitive(gauche:'Liste', droite:'Liste') -> 'Liste':  # coût linéaire en θ(ng+nd)
42      nouveau = []
43      for i in range(len(gauche)):
44          nouveau.append(gauche[i])
45      for i in range(len(droite)):
46          nouveau.append(droite[i])
    return nouveau
```

Comme on le voit, inutile d'utiliser des fonctions d'interface qui vont exactement ce que font Python sur le type **`list`**.

D'ailleurs, la concaténation est très bien réalisée par **`lst1 + lst2`**,

Pour savoir comment, il est temps de passer à la programmation orientée objet.

BILAN :

1) Créer une **liste contiguë immuable** basée sur des tableaux statiques est assez peu performant.

2) Créer une **liste contiguë muable** basée sur le type **`list`** de Python est efficace si vous devez surtout rajouter et supprimer en Fin de Liste. Créer des fonctions d'interface pour manipuler **`list`** est inutile dans ce cas. Une seule ligne par fonction, autant la taper quand on en a besoin.

Quels sont les opérations primitives à coût constant avec list ?

→ Nouvelle Liste
```
t = list()
t = []
```

→ Insérer en fin :
```
t.append(5)
t.append(50)
```
La liste contient alors 5 → 50

→ Longueur :
```
nb = len(t)
```
nb contient alors 2.

Et surtout pas ~~`t = t.append(50)`~~ car **append()** renvoie **None**.

→ ieme **t[1]** renvoie 50 sans modifier la list.

→ Supprimer en fin :
```
v = t.pop()
```
`v contient 50 et t ne contient 5 uniquement.`

→ Est liste vide :
```
t == []
```