# Données 13 - Implémentation du type abstrait LISTE CONTIGUË

## I – Implémentation immuable basée sur des tableaux statiques

**1.1 RAM** (voir notes *)

**1.2 Principe du tableau** (*)

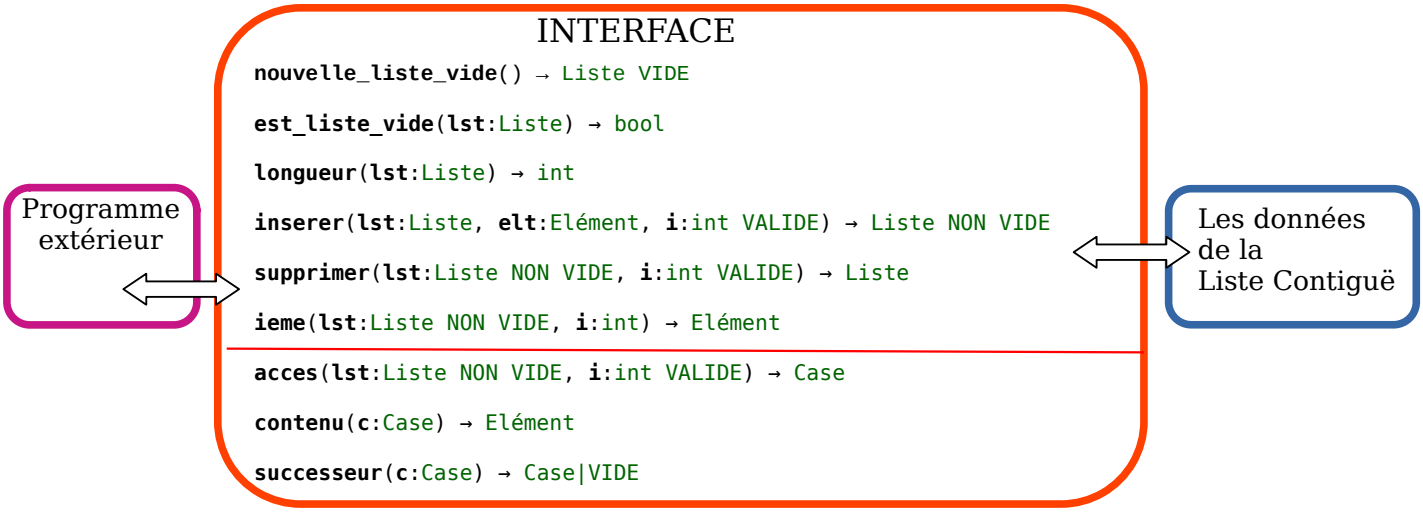
**1.3 Coûts des accès** (*)

### 1.3 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)]
```

### 1.5 Coûts des actions sur les tableaux en Python (*)

### Primitives de la Liste Contiguë (version immuable et basée sur un tableau statique)




```
def nouvelle_liste_vide() -> 'Liste VIDE':                                  # coût constant
    '''Renvoie une liste vide'''
     return []

def est_liste_vide(lst:'Liste') -> bool:                                    # coût constant
    '''Prédicat qui renvoie True si la liste est vide'''
    return lst == []

def longueur(lst:'Liste') -> int:                                           # coût constant
    '''Renvoie le nombre d'éléments stockés dans lst'''
    return len(lst)

def ieme(lst:'Liste NON VIDE', i:int) -> 'Element':                         # coût constant
    '''Renvoie le contenu de la Case de position i VALIDE dans la liste NON VIDE lst'''
    return lst[i]

def inserer(lst:'Liste', elt:'Elément', i:'int VALIDE') -> 'Liste NON VIDE': # linéaire en θ(n)
   '''Renvoie une Liste basée sur lst où elt est en position VALIDE i'''
    nb = len(lst) + 1
    nouveau = [None for _ in range(nb)]   # 1 - nouveau tableau avec une case en plus
    for k in range(0, i):                 # 2 - copie les elts de de 0 à i-1 sans décalage
        nouveau[k] = lst[k]
    for k in range(i, len(lst)):          # 3 - décalage à droite des elts d'indice i et plus
        nouveau[k+1] = lst[k]
    nouveau[i] = elt                      # 4 - elt à la position i
    return nouveau

def supprimer(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Liste':             # linéaire en θ(n)
    '''Renvoie une Liste basée sur lst NON VIDE en supprimant la case i VALIDE'''
    nb = len(lst) - 1
    nouveau = [None for _ in range(nb)]   # 1 - nouveau tableau avec une case en moins
    for k in range(0, i):                 # 2 - copie les elts de 0 à i-1 sans décalage
        nouveau[k] = lst[k]
    for k in range(i+1, len(lst)):        # 3 - décalage à gauche des elts d'indice i+1 et plus
        nouveau[k-1] = lst[k]
    return nouveau
```

```python
def acces(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Case':                    # coût constant
    return (lst, i)                        # on crée un tuple-case (quelle liste ?, quel indice ?)

def contenu(c:'Case') -> 'Elément':                                           # coût constant
    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') -> 'Case|VIDE':                                      # coût constant
    lst = c[0]                        # on récupère la liste contenant notre case
    i = c[1]                          # on récupère l'indice de notre case
    if i == len(lst) - 1:             # si la case est la dernière de la Liste
        return ()                     # on renvoie un Successeur vide
    else:
        return (lst, i+1)             # on crée le tuple-case et on renvoie
```

### 1.6 Gestion des cases (*)

### 1.7 Coûts (*)

### 1.8 Concaténation (*)

## II – Type list de Python

### 2.1 Le type list de Python

La **`list`**-Python est un **tableau <u>dynamique</u> <u>mutable</u>**. **<u>Ce n'est PAS une LISTE (CHAINEE)</u>**.

L'idée 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.

### 2.2 Coûts des opérations

- TYPE MUTABLE car **`list.append(Element) → None`** **`list.pop() → Element`**
- **`list.append()`** : un coût amorti **constant** $\mathcal{O}(1)$ pour l'insertion en fin.
- **`List.pop()`** : un coût **constant** $\mathcal{O}(1)$ pour la suppression en fin.
- **`list.insert(v, i)`** : un **coût linéaire** $\mathcal{O}(n)$ pour l'insertion, le pire des cas étant en i = 0.
- **`list.pop(i)`** : un **coût linéaire** $\mathcal{O}(n)$ pour la suppression, le pire des cas étant en i = 0.

***Au vu des coûts, on voit bien que la list Python n'est donc pas une Liste Chaînée.***

```python
def nouvelle_liste_vide() -> 'Liste VIDE':                          # coût constant
    return []

def est_liste_vide(lst:'Liste') -> bool:                            # coût constant
    return lst == []

def longueur(lst:'Liste') -> int:                                   # coût constant
    return len(lst)

def ieme(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Element':        # coût constant
    return lst[i]

def inserer_fin(lst:'Liste', elt:'Elément') -> None:                # coût constant
    lst.append(elt)

def supprimer_fin(lst:'Liste NON VIDE') -> 'Element':               # coût constant
    return lst.pop()

def inserer(lst:'Liste', elt:'Elément', i:'int VALIDE') -> None:    # coût linéaire en θ(n-i)
    lst.insert(i, elt)

def supprimer(lst:'Liste NON VIDE', i:'int VALIDE') -> 'Elément':   # coût linéaire en θ(n-i)
    return lst.pop(i)
```

### 2.2 Utilisation directe du type list

→ <u>Nouvelle liste :</u> **`t = list()`** ou **`[]`** <u>Est liste vide</u> : **`t == []`**

→ <u>Insérer en fin</u> : **`t.append(5)`** **surtout pas ~~t = t.append(50)~~**

**`t.append(50)`** La liste contient 5 → 50

→ <u>Longueur</u> : **`nb = len(t)`** nb contient alors 2.

→ <u>ieme</u> **t[1]** renvoie 50 sans modifier la list.

→ <u>Supprimer en fin</u> : **`v = t.pop()`** v contient 50 et t devient [5] uniquement.

→ <u>Concaténation</u> : t = tg + td