Données 17 - Implémentation du type Pile

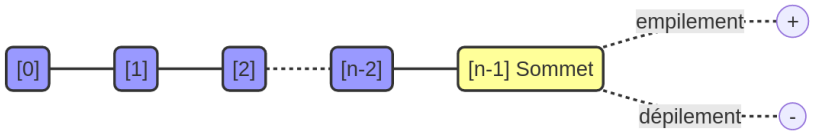
I – Implémentation MUABLE basée sur un tableau dynamique

Cette version est MUABLE, il faut utiliser des appels comme : **`empiler(f, 5)`** ou **`depiler(f)`**

Pour des primitives à un coût CONSTANT, il faut placer le SOMMET au niveau _____.
En effet :

- **`append(elt)`** et **`pop()`** sont à coût _____lors de l'action sur la dernière case.
- **`insert(elt, 0)`** et **`pop(0)`** sont à coût _____lors de l'action sur la première case.
- **`len(t)`** est à coût constant dans l'implémentation du type list de Python.

Une Pile Vide est implémentée comme un tableau vide. Dans tous les cas, la Pile est juste une référence vers le tableau.



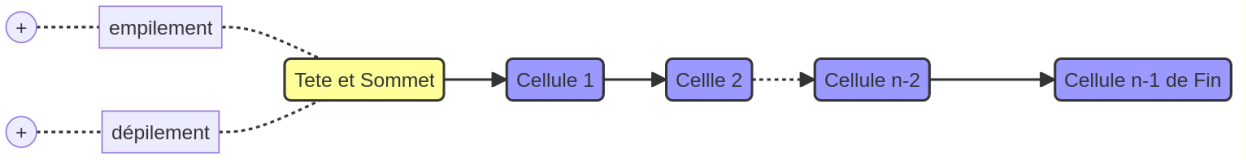

```
01  def nouvelle_pile() -> 'Pile':                . . . . . .  coût _____
02      '''Renvoie une nouvelle Pile vide'''
03      return []
04
05  def est_pile_vide(p:'Pile') -> bool:                  . . . .  coût _____
06      '''Prédicat qui renvoie True si p est une Pile vide'''
07      return p == []
08
09  def empiler(p:'Pile', elt:'Element') -> None:          . . . .  coût _____
10      '''Modifie la pile p en rajouter elt au sommet'''
11      p.append(elt) # rajoute un élément à la fin du tableau
12
13  def depiler(p:'Pile NON VIDE') -> 'Element':                     coût _____
14      '''Supprime le sommet de la Pile p et renvoie l'élément supprimé'''
15      return p.pop() # supprime et renvoie le dernier élément du tableau
16
17  def lire_sommet(p:'Pile NON VIDE') -> 'Element':                 coût _____
18      '''Renvoie la valeur actuellement au sommet de la Pile p NON VIDE'''
19      return p[len(p)-1] # ou même return p[-1]
```

II – Implémentation IMMUABLE basée sur une liste chaînée de tuples

Cette version est IMMUABLE, il faut utiliser des appels comme : **`p = empiler(p, 5)`** ou **`p = depiler(p)`**

Pour avoir des primitives à coût CONSTANT, il faut placer le SOMMET au niveau _____.

Les tuples formant les Cellules ont tous deux éléments : (_____, _____)
Une Pile Vide est implémentée comme un tuple vide, sinon la Pile pointe la cellule au sommet.



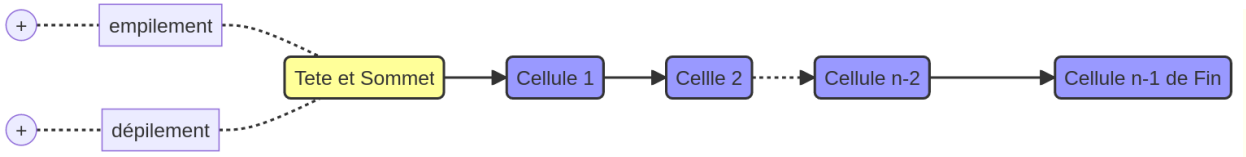

```
1   def nouvelle_pile() -> 'Pile':               . . . . . .  coût _____
2       return ()
3
4   def est_pile_vide(p:'Pile') -> bool:         . . . . . .  coût _____
5       return p == ()
6
7   def empiler(p:'Pile', elt:'Element') -> 'Pile':                  coût _____
8       '''Renvoie une pile basée sur p mais en rajoutant elt au sommet'''
9       return (elt, p) # renvoie un tuple contenant elt et menant ensuite à la pile p reçue
10
11  def depiler(p:'Pile NON VIDE') -> 'Pile':                      coût _____
12      '''Renvoie une pile basée sur p mais en supprimant le sommet'''
13      return p[1] # renvoie un tuple qui est le reste de la pile p
14
15  def lire_sommet(p:'Pile NON VIDE') -> 'Element':               coût _____
16      return p[0] # renvoie la valeur stockée dans la cellule Sommet
```

III – Implémentation MUABLE basée sur une liste chaînée d'objets

Comme cette version est MUABLE, il faut utiliser des appels comme : **`empiler(p, 5)`** ou **`depiler(p)`**

Pour avoir des primitives à un coût CONSTANT, il faut placer le SOMMET au niveau _____.

Les objets formant les Cellules ont tous deux attributs _____.
La Pile elle-même est implémentée comme un objet n'ayant qu'un attribut pointant vers le sommet.
Si la Pile est vide, on pointe la cellule au sommet.



MODULE **AUTONOME** à une liste chaînée créée sur mesure

```python
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

class Pile:
    def __init__(self):
        self.sommet = None

    def est_pile_vide(self:'Pile') -> bool:
        return self.sommet is None

    def empiler(self:'Pile', elt:'Element') -> None:
        nouvelle = Cellule(elt, None) # Etape 1 : création de la cellule
        nouvelle.s = self.sommet      # Etape 2 : on relie la nouvelle cellule à l'"ancien" sommet
        self.sommet = nouvelle        # Etape 3 : modification de la liste

    def depiler(self:'Pile NON VIDE') -> 'Element':
        ancienne_valeur = self.sommet.v    # Etape 1 : on mémorise la valeur au sommet
        self.sommet = self.sommet.s        # Etape 2 : sommet devient le succ. du sommet supprimé
        return ancienne_valeur             # Etape 3 : on renvoie la valeur stockée

    def lire_sommet(self:'Pile NON VIDE') -> 'Element':
        return self.sommet.v

def nouvelle_pile() -> 'Pile':
    return Pile()

def est_pile_vide(p:'Pile') -> bool:
    return p.est_pile_vide()

def empiler(p:'Pile', elt:'Element' ) -> None:
    p.empiler(elt)

def depiler(p:'Pile NON VIDE') -> 'Element':
    return p.depiler()

def lire_sommet(p:'Pile NON VIDE') -> 'Element':
    return p.lire_sommet()
```

MODULE **AVEC DEPENDANCE** à un module de Liste chaînée d'objets

```python
from liste_chainee_tete import Liste

def nouvelle_pile() -> 'Pile':
    '''Renvoie une nouvelle Pile vide'''
    return Liste()

def est_pile_vide(p:'Pile') -> bool:
    '''Prédicat qui renvoie True si p est une Pile vide'''
    return p.est_liste_vide()

def empiler(p:'Pile', elt:'Element') -> None:
    '''Rajoutant elt au sommet de la pile p'''
    p.inserer_tete(elt)

def depiler(p:'Pile NON VIDE') -> 'Element':
    '''Dépile et renvoie le sommet de la pile p'''
    return p.supprimer_tete()

def lire_sommet(p:'Pile NON VIDE') -> 'Element':
    '''Renvoie la valeur actuellement au sommet de la Pile p NON VIDE'''
    return p.premier()
```