> Version pdf

Type abstrait Liste (Bac 🎯)

Type abstrait Liste

> ✏️ **"Point de cours 3"**
>
> Le **type abstrait Liste** permet de créer une séquence d'éléments ordonnés par leur position dans la liste.
>
> - Un liste peut être vide.
> - Si la liste est non vide, contrairement aux types abstraits tableau ou dictionnaire, un seul élément de la liste est accessible directement, c'est la **tête** de liste
> - La **queue** de liste permet d'accéder aux éléments suivants. La queue de liste est une liste donc on accède au deuxième élément de la liste (s'il existe) en prenant la tête de la queue de liste. De même pour accéder au troisième élément on prend la tête de la queue de la queue de liste ...
>
> L'accès aux éléments successifs de la liste n'est donc pas direct mais nécessite une répétition d'opérations identiques, appelée **parcours de liste**. On dit qu'une liste est une **structure linéaire**.
>
> Une interface minimale du **type abstrait Liste** est la suivante :

Opération	Signature	Description
creer_liste	creer_liste()	Renvoie une liste vide
liste_vide	liste_vide(lis)	Renvoie un booléen indiquant si la liste `lis` est vide
inserer	inserer(lis, elt)	Insère `elt` en tête de la liste `lis` , renvoie une nouvelle liste ou modifie la liste en place
tete	tete(lis)	Renvoie l'élément en tête de liste
queue	queue(lis)	Renvoie la liste privée de l'élément en tête liste

"Liste homogène"

Dans tout le chapitre on se restreint à manipuler uniquement des listes dont tous les éléments sont de même type.

"Méthode 2"

Le parcours d'une **liste** est une répétition d'opérations `tete` pour lire la valeur de l'élément accessible en tête de liste et `queue` pour passer à l'élément suivant.

Cette répétition peut s'effectuer de façon *itérative* avec une boucle ou *récursive*.

"Exemple 2"

On peut écrire une fonction déterminant la longueur d'une liste de façon itérative ou récursive. Retenez bien ce modèle, qu'on peut décliner pour tous les parcours de liste.

"Parcours de liste itératif"

```
def longueur(lis):
    lon = 0
    courant = lis  # élément en tête
    while not liste_vide(courant):
        # si besoin traitement sur tete(courant)
        lon = lon + 1
        courant = queue(courant)
    return lon
```

"Parcours de liste récursif"

```
def longueur_rec(lis):
    if liste_vide(lis):
        return 0
    return 1 + longueur_rec(queue(lis))
```

Liste chaînée

"Point de cours 4"

On représente en général une **liste** comme une chaîne de **cellules** reliées par des flèches, on parle de **liste chaînée**.

Chaque cellule est un couple qui porte deux informations :

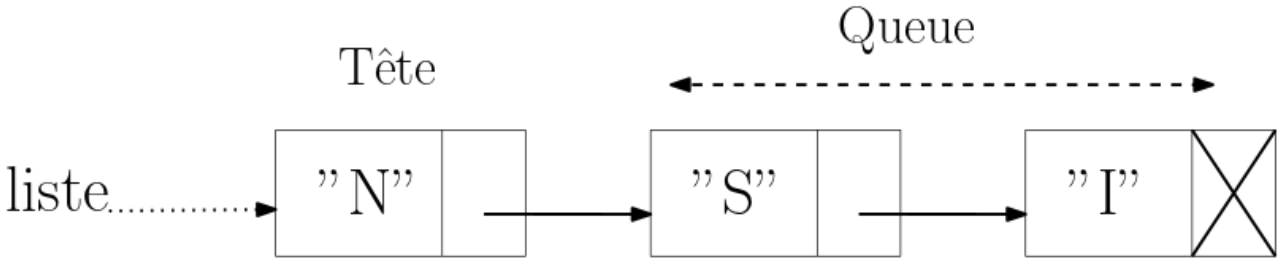
- une valeur
- un lien vers la cellule suivante

On peut définir le **type abstrait Liste** récursivement. Une liste est :

- soit vide
- soit une cellule portant une valeur appelée la **tête** de la liste, et un lien vers une autre liste appelée **queue** de la liste.

"Exemple 3"

On a représenté ci-dessous une liste avec une chaîne de trois cellules contenant les caractères "N", "S" et "I". Notez que l'élément en tête de liste est le dernier inséré.




```
liste = creer_liste()
inserer(liste, "I")
inserer(liste, "S")
inserer(liste, "N")
```

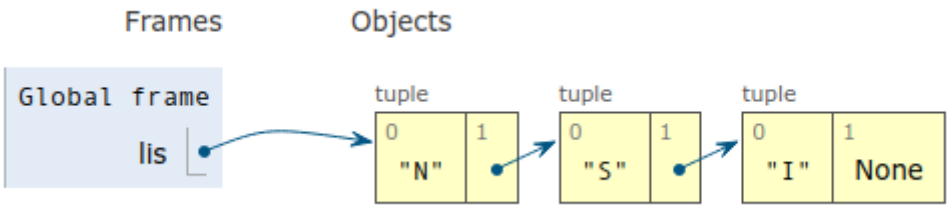
"Méthode 3"

Pour implémenter une **liste chaînée**, il faut une structure de données permettant de représenter une **cellule**. Voici deux solutions :

"Implémentation avec tuples"

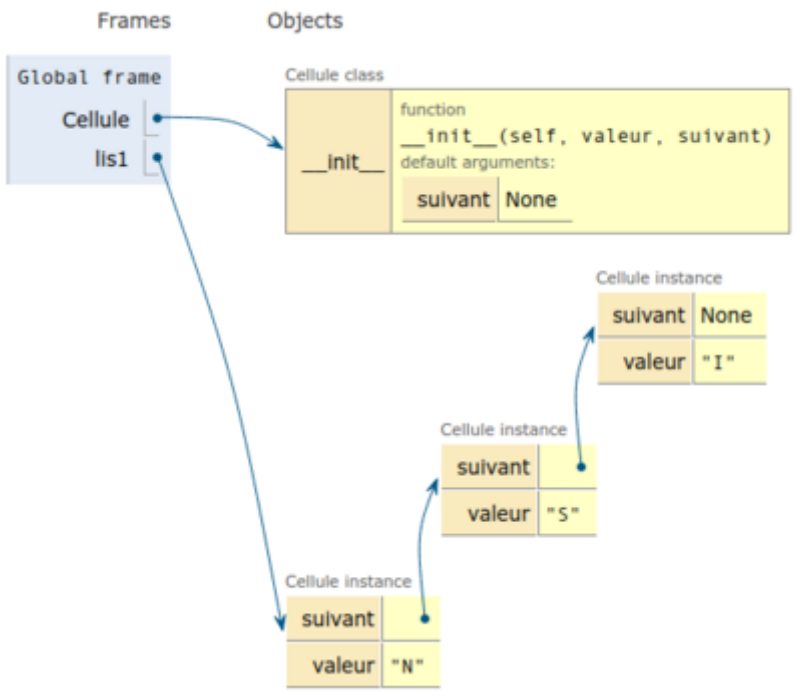
Une **cellule** est représentée par un `tuple`, la première composante porte la valeur et la seconde un lien vers une autre cellule. La dernière cellule porte un lien vers un objet représentant une cellule vide, par exemple `None` ou `()`. On implémente alors la liste chaînée de l'exemple 2 par `lis = ("N", ("S", ("I", None)))`.

```
lis = ("N", ("S", ("I", None)))
```



"Implémentation avec une classe `Cellule`"

Une **cellule** est représentée par un objet de la classe `Cellule`, qui possède deux attributs : `valeur` portant la valeur et `suivant` portant le lien vers la cellule suivante dans la liste chaînée. On implémente alors la liste chaînée de l'exemple 3 par :




```
class Cellule:

    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

lis1 = Cellule("N", Cellule("S", Cellule("I", None)))
```

"Complexité temporelle"

Pour les deux implémentation de liste chaînée avec `tuple` ou classe `Cellule`, on a les mêmes complexités temporelles par rapport au nombre n d'éléments dans la liste. Le seul élément accessible directement est la *tête*, en temps constant. Le coût d'un parcours complet de la liste en suivant le chaînage est *linéaire*, proportionnel à la taille de la liste.

Opération	Type complexité	Notation de Landau
`inserer`, `tete` ou `queue`	constante	$O(1)$
parcours de toute la liste	linéaire	$O(n)$

Liste chaînée mutable

"Point de cours 5"

- Une structure de données est **immuable** si on ne peut la modifier une fois qu'elle est construite. Évidemment on peut toujours accéder aux données en lecture
- Une structure de données est **mutable** si on peut la modifier une fois qu'elle est construite.
- Cette distinction entre types **immuable** et **mutable** n'a de sens que pour des types structurés, les types simples comme les entiers, les flottants, les booléens sont **immuables** (1 reste égal à 1 !).

"Exemple 4 types natifs mutables et immuables en Python"

En Python, pour représenter une séquence de données on peut utiliser le type `list` des tableaux dynamiques qui est **mutable** ou le type `tuple` qui est **immuable**.

```
>>> tup = (14, 10)
>>> tup[0]
14
>>> tup[0] = 11
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tad = [14, 10]
>>> tad[0]
14
>>> tad[0] = 11
```

Quel est l'intérêt d'un type **immuable** ? Dans certains cas on a besoin de fixer certaines valeurs par exemple lorsqu'elles servent de clefs dans un dictionnaire implémenté par une table de hachage. Une fonction de hachage calcule à partir de la clef l'index de la case du tableau où est stockée la valeur donc la clef ne doit pas être modifiée pour que son image par la fonction de hachage reste inchangée. Ainsi un type **mutable** ne peut servir de clef dans un dictionnaire.

```
>>> dico = dict()
>>> dico[tup] = 1
>>> dico
{(14, 10): 1}
>>> dico[tad] = 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Si dans l'implémentation d'une liste chaînée, une *cellule* implémentée par un *tuple*, la liste chaînée est donc **immuable**.

"Exemple 6 listes chaînées immuables avec des tuples"

Avec l'implémentation du type abstrait Liste par des cellules implémentées à l'aide de `tuple`, les opérations `queue` et `inserer` renvoient une nouvelle liste. Le type `tuple` étant **immuable** en Python, ces listes chaînées sont immuables.

On a également présenté une implémentation à l'aide d'une classe `Cellule` qui devrait donc permettre de construire des listes chaînées **mutables**, il nous manquait juste la possibilité de représenter une liste vide.

"Méthode 4 : listes chaînées mutables"

Pour construire des listes chaînées mutables, il est naturel d'utiliser le *paradigme objet (POO)*. On reprend l'implémentation avec une classe `Cellule` mais on intègre cette fois toutes les opérations du type abstrait Liste comme méthode d'une classe `Liste`. Cette classe possède un seul attribut `tete_liste` qui pointe soit vers `None` (liste vide) ou vers la première cellule de la liste chaînée.

```python
class Cellule:

    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

class Liste:

    def __init__(self):
        self.tete_liste = None

    def liste_vide(self):
        return self.tete_liste is None

    def inserer(self, valeur):
        self.tete_liste = Cellule(valeur, self.tete_liste)

    def tete(self):
        assert not self.liste_vide()
        return self.tete_liste.valeur

    def queue(self):
        assert not self.liste_vide()
        liste_queue = Liste()
        liste_queue.tete_liste = self.tete_liste.suivant
        return liste_queue

    def modifier_tete(self, valeur):
        assert not self.liste_vide()
        self.tete_liste.valeur = valeur
```

"Effets de bord"

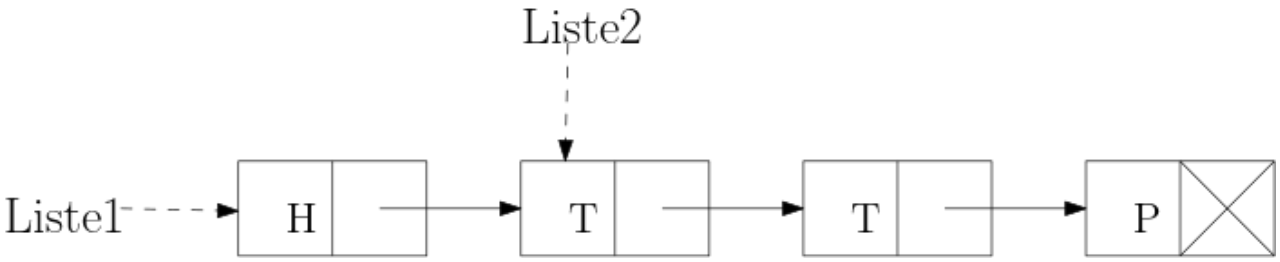
Les *listes chaînées mutables* peuvent être modifiées après leur construction. Si deux listes chaînées mutables partagent les mêmes cellules, modifier une liste modifie l'autre par **effet de bord**, ce qui n'est pas toujours souhaité.

"Exemple 5"

On crée une liste mutable `liste1` puis on associe la queue de `liste1` à une variable `liste2`. Si on modifie la valeur de la cellule en tête de `liste2` c'est aussi la cellule suivante de la tête de `liste1` qui est donc modifiée par **effet de bord**.

```
>>> liste1 = Liste()
>>> liste1.inserer('P')
>>> liste1.inserer('T')
>>> liste1.inserer('T')
>>> liste1.inserer('H')
>>> print(liste1)
(H, (T, (T, (P, ()))))
>>> liste2 = liste1.queue()
>>> print(liste2)
(T, (T, (P, ())))
>>> liste2.modifier_tete('F')
>>> >>> print(liste2)
(T, (T, (P, ())))
>>> print(liste1)
(H, (F, (T, (P, ()))))
```

Avant la modification de `liste2` :



Après la modification de `liste2` :