I Listes chaînées

Nous avons vu différentes structures de données dont les tableaux et les dictionnaires.

◇ les tableaux sont de taille ..... mais peuvent être agrandis quitte à copier tous les éléments dans un ..... L'accès mémoire étant très ..... en temps, les solutions ont tendance à doubler la taille des tableaux à chaque fois (*cf.* exo n° 9 POO).

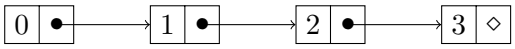
◇ les dictionnaires sont assez complexes (*cf.* 1ère : ils requièrent une 'bonne' fonction de .....) mais restent semblables à un grand tableau de taille fixe, qui est d'autant plus difficile à agrandir qu'il faudrait trouver une nouvelle .....

Ne serait-il pas possible de prévoir une structure de données qui soit intrinsèquement dynamique ?

1 Définition

Définition 1

Une liste chaînée est une structure composée d'objets contenant une valeur et reliés à l'objet suivant.

expl1 : 

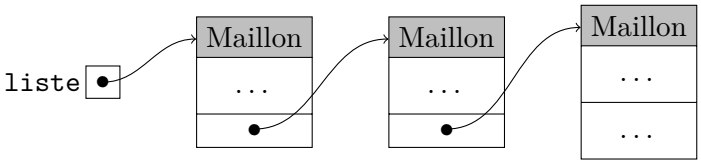
L'avantage est qu'ainsi l'ajout ou la suppression d'un objet (un maillon) n'implique plus une copie de l'ensemble de la structure mais seulement de ..... maillons.

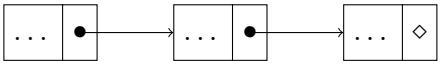
Il est naturel d'implémenter une telle structure au travers d'une classe `Maillon`

```
class Maillon :
    def __init__(self, ...
        ...
        ...

# et pour définir la liste chaînée expl1, on écrit
Maillon(1, Maillon(2, Maillon(3, Maillon(4, None))))
```

Maillon
valeur (object)
suivant ('Maillon')

Rq : de manière plus rigoureuse, la commande `liste = Maillon(21, Maillon(-3, Maillon(7, None)))`

définit la chaîne suivante 

correspondante à la chaîne 

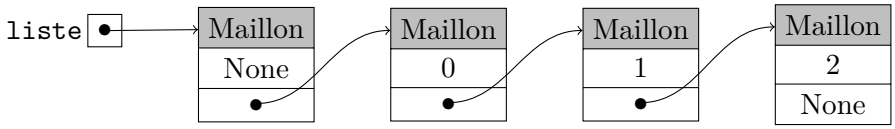
? Comment représenter la liste vide ?

Si notre variable contenant la liste prend la valeur `None`, on ne pourra plus utiliser..... (telle que `len()`) et on devra continuellement faire .....

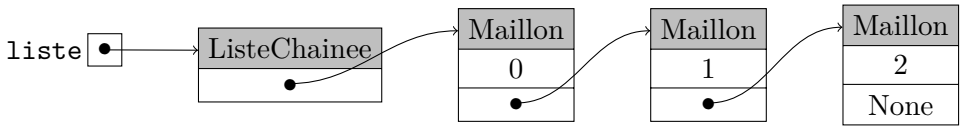
Pour palier cela on peut imaginer :

◇ que le premier maillon comporte l'attribut.....

On a défini une ..... qui sera toujours notre premier maillon de liste. Cela permet de ne pas devoir traiter à part le cas de la liste vide. La variable `liste` sera donc du type .....



◇ créer une classe `ListeChainee` englobante la classe `maillon` (on dit qu'on ..... la classe `maillon`). Il n'y a alors pas de sentinelle et on traite le cas de la liste vide à part. La variable `liste` sera donc du type .....



Les méthodes de la classe `ListeChainee` se définissent alors

◇ si la classe Maillon ne contient aucune méthode et est considérée comme une simple structure, en manipulant directement les objets de type `Maillon`

◇ si la classe Maillon définit un ensemble de méthodes, à partir des méthodes de `Maillon` avec quelques pré-traitements si nécessaire.

```
class Maillon : # structure maillon sans méthode
    def __init__(self, val : object, nxt : 'Maillon') -> 'Maillon' :
        self.valeur = val
        self.suivant= nxt

class ListeChainee :
    def __init__(self) -> 'ListeChainee':
        self.maillon = None # contiendra par la suite une instance Maillon
```

2 Analyse de la liste

Contrairement au tableau qui possède une taille fixe et dont la connaissance de la taille est ..... (le coût est donc en .....), on peut se demander comment obtenir la longueur d'une liste chaînée.

Il faut parcourir toute la liste jusqu'à obtenir un `maillon` avec ..... Le coût est donc ..... (en ..... où n est la taille de la liste).

L'accès au i^è élément de la liste se fait de manière similaire en comptant le nombre d'instances visitées et en prenant garde de ne pas dépasser le dernier maillon de la liste.

```
class ListeChainee :
    def __init__(self) -> 'ListeChainee':
        self.maillon = None

    def __len__(self) -> ......... :
        i = 0
        maillActu = ...
        while ............... is not None :
            i ...
            maillActu = ...
        return ...

    def __getitem__(...
        if ...
            raise IndexError( "getitem : l'indice doit être positif et strictement inférieur
            ↪ à la longueur de liste" )
        j = 0
        maillActu = ...
        while j < i :
            j ...
            maillActu = ...
        return ...
```

Rq : la solution proposée de `__getitem__(i : int) -> object` parcourt toujours toute la liste puisque compare la valeur de i à la longueur de la liste (ainsi le coût est linéaire ($\mathcal{O}(n)$ avec n longueur de la liste).

Pour ne pas avoir à toujours parcourir toute la liste, on peut proposer l'implémentation suivante mais elle ajoute à chaque tour de boucle une comparaison supplémentaire.

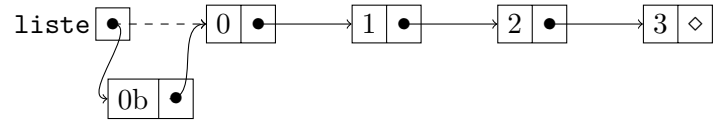
```
class ListeChainee:
    ...
    def __getitem__(...
```

```
        if ...
            raise IndexError( "getitem : l'indice doit être positif" )
        j = 0
        maillActu = self.maillon # None ou de type Maillon
        while j < i and maillActu is not ..... :
            j ...
            maillActu = ...
        if maillActu is None : # la longueur de liste a été dépassée
            raise IndexError( "getitem : ...
        return ...
```

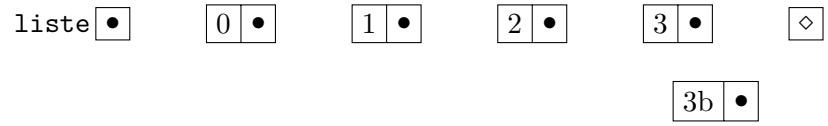
3 Ajout d'un objet et concaténation de listes

On peut vouloir ajouter un élément

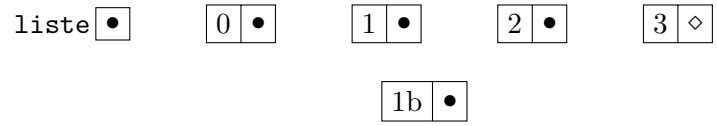
◇ en début de liste



◇ en fin de liste



◇ à n'importe quel endroit



On s'aperçoit qu'à chaque fois le même processus est effectué : `maillon.suivant` prend la valeur du nouveau maillon et le nouveau `maillon.suivant` prend la valeur de l'ancien `maillon.suivant`.

⚠ Comme nous le verrons pour la concaténation, modifier des objets en place ouvre de grands risques. Pour limiter ces risques, on préfèrera souvent recréer un nouvel objet (ce qui n'est pas le cas ici).

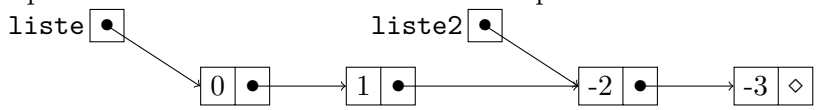
Pour ajouter une nouvelle valeur à l'indice i de la liste chaînée on peut donc proposer :

```
class ListeChainee:
    ...
    def ajout_debut(self, val : object) -> None :
        suiv = ...
        self.maillon = ...

    def ajout_i(self, val : object, i : int) -> None : # 0 <= i <= len(self)
        if ...
            self.....
        else :
            j = 1 # pour mettre val à l'indice i
            maillActu = ...
            while j < i : # maillActu.suivant is not None -> inutile
                j ...
                maillActu = ...
            suiv = ...
            maillActu.suivant = ...
```

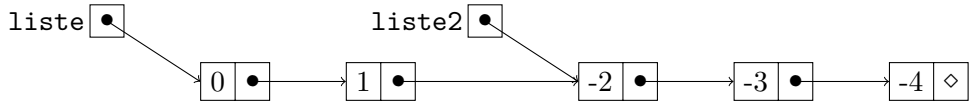
Concaténer deux listes revient à ajouter à la fin de la 1er liste le premier maillon de la 2n. Ainsi,

```
class ListeChainee:
    ...
    def concatener(self, liste2 : 'ListeChainee') -> None :
        if self.maillon is None :
            self.maillon = ...
        else :
            maillActu = ...
            while maillActu.suivant is not None :
                maillActu = ...
            maillActu.suivant = ...
```

⚠ modifier des listes en place est dangereux car cela peut entraîner des effets de bords indésirables.
expl2 : on considère deux listes suivantes que l'on concatène ensuite.



Si maintenant on ajoute -4 à la fin de la `liste2`, la `liste1` est ..... par effet de bords (!).

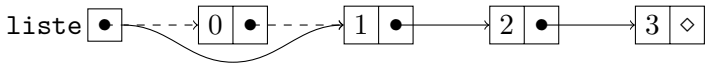


On prévilègera donc des modifications de listes qui renvoient des ..... (pas de modification en place).

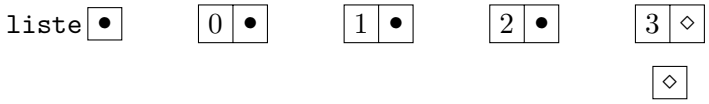
4 Suppression d'un objet

On peut vouloir supprimer un élément

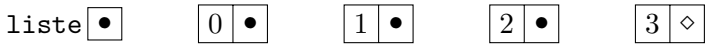
◇ en début de liste



◇ en fin de liste



◇ à n'importe quel endroit



Pour supprimer un maillon à l'indice i de la liste chaînée on peut donc proposer :

```
class ListeChainee :
    ...
    def supprime_debut(self) -> None :
        if ......... is not None :
            ...

    def supprime_i(self, i : int) -> None : # 0 <= i < len(self)
        if i==0 :
            ...
        else :
            j = 1 # pour retirer le maillon i
            maillPrec = ...
            while j < i : # maillPrec.suivant is not None -> inutile
                j ...
                maillPrec = ...
            maillPrec.suivant = ...
```

5 Autres opérations

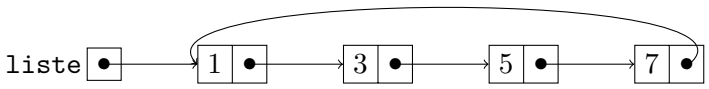
De multiples opérations sont possibles, parmis lesquelles l'inversion en place d'une liste chaînée et la copie.

```
class ListeChainee :
    ...
    def inversion(self) -> None : # <-> reverse()
        newLC = ...
        maillActu = ...
        while ...
            ...
            maillActu = ...
        self.maillon = ...

    def copie(self) -> 'ListeChainee' :
        newLC = ...
        i = 0
        maillActu = ...
        while ...
            ...
            i ...    # i == len(newLC)
            maillActu = ...
        return ...
```

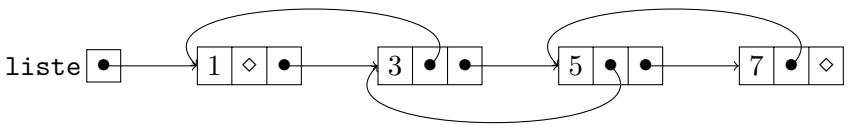
⚠ Les listes précédentes sont des listes simplement chaînées. Elles sont peu pratique si l'on veut, par exemple, remonter la liste à partir de la fin.

Ainsi on peut palier à certains problèmes en proposant d'autres listes (*cf.* exos) :

◇ des listes ..... (ou circulaires) :



◇ des listes ..... chaînées :



Pour conclure, nous utilisons depuis un an la notion de tableau.
En `python`, un tableau est du type ..... Il s'agit en fait d'une implémentation de liste chaînée avec les méthodes `__getitem__()` et `__setitem__()` qui permettent respectivement ..... à la i^ème valeur (*e.g.* `liste[i]`) et ..... de la i^ème valeur (*e.g.* `liste[i]=2`).
De plus, pour agrandir un tableau nous faisions la concaténation de tableau (*e.g.* .....).

L'ajout d'une valeur en fin de liste consiste à ..... Cette dernière action se fait avec notre classe ..... et sous `python` via `liste.append(val)`.

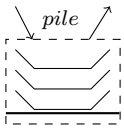
Tandis que la suppression du dernier maillon se fait avec notre classe ..... et sous `python` via `liste.pop()`.

II Piles et files

1 les piles : LIFO

Définition 2

Une pile est une structure dans laquelle le dernier élément arrivé est également le ..... à pouvoir en sortir (*Last In First Out*)



Rq : la taille d'une pile est en théorie infinie.

On retrouve les piles dès lors que ce qui nous importe est la dernière action faite, dans la récursivité ou bien au niveau de certains CPU.

On implémente les structures de pile avec généralement six méthodes :

◇ le constructeur qui renverra .....

◇ `empiler(val : object) -> None`, qui .....

◇ `depiler() -> object`, qui retire l'élément au sommet de la pile et le renvoie

◇ `taille() -> int`, qui précise ..... contenus dans la pile

◇ `est_vide() -> bool`, qui précise si la pile est vide ou non

◇ `sommet() -> object`, qui renvoie l'élément au sommet .....

On peut facilement simuler une pile avec des listes chaînées ou des tableaux. Ici, une solution avec notre classe `ListeChainee` :

```
class Pile:
    def __init__(self) -> ...
        self.pile = ...

    def empiler(self, val : object) -> None :
        ...

    def depiler(self) -> object : # sans programmation défensive
        ...

    def taille(self) -> int :
        ...

    def est_vide(self) -> bool :
        ...

    def sommet(self) -> object : # sans programmation défensive
        ...
```

ou bien avec les listes `python` :

```
class Pile:
    def __init__(self) -> 'Pile' :
        self.pile = []

    def empiler(self, val : object) -> None :
        ...

    def depiler(self) -> object : # sans programmation défensive
        ...

    def taille(self) -> int :
```

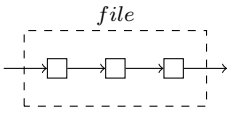
```
        ...

    def est_vide(self) -> bool :
        ...

    def sommet(self) -> object : # sans programmation défensive
        ...
```

Rq : les deux solutions ci-dessous ne sont pas optimales pour obtenir la longueur d'une pile. Nous verrons en exercices une possibilité pour obtenir la longueur de la pile avec un coût constant ($\mathcal{O}(1)$).

2 les files : FIFO

Définition 3

Une file est une structure dans laquelle le premier élément arrivé est également le ..... élément à pouvoir en sortir (*Fist In First Out*)



Rq : la taille d'une file est en théorie infinie.

On retrouve les files dès lors que ce qui nous importe est l'ordre des actions faites, dans des appels concurrents ou dans certains CPU.

On implémente les structures de file avec généralement cinq méthodes :

- le constructeur qui renverra .....
- `enfiler(val : object) -> None`, qui.....
- `defiler() -> object`, qui retire.....
- `taille() -> int`, qui précise le nombre d'éléments contenus dans la file
- `est_vide() -> bool`, qui précise si la file est .....

Rq : on peut facilement simuler une file à l'aide de piles, de listes chaînées ou de tableaux. Ici une implémentation avec notre classe `ListeChainee` où l'on retient la taille de la file :

```
class File:
    def __init__(self) -> 'File' :
        self.file = ...
        ...

    def enfiler(self, val : object) -> None :
        ...
        ...

    def defiler(self) -> object : # sans programmation defensive
        self.taille ...
        derMaillon = ...
        self.file ...
        return ...

    def taille(self) -> int :
        return ...

    def est_vide(self) -> bool :
        return ...
```