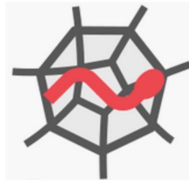
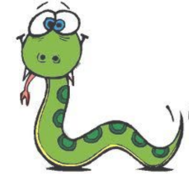
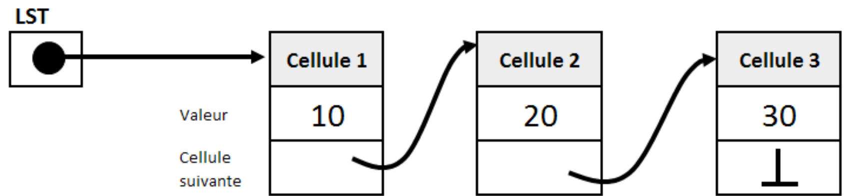
NSI





Structures de données : les listes

Liste, Pile, File

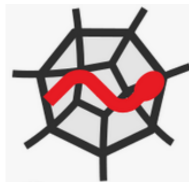
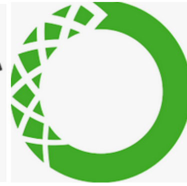
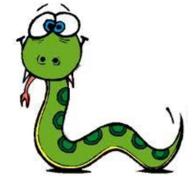


Sommaire

1 Le type abstrait de données 2
1.1 Définition et utilité 2
1.2 Avantages du concept de Type Abstrait de Données 3

2 Un premier type abstrait de données, la liste 3
2.1 Description du type abstrait liste 3
Organisation 3
Opérations disponibles 3
Exemples d'utilisation 4
2.2 La liste une première implémentation 4
Retour sur la liste en Python 4
Implémentation des opérations décrites dans l'interface 6
2.3 Améliorations de l'interface 7
(a) Impression de la liste 7
(b) Lire un élément à une position quelconque de la liste 8
(c) Ajouter un élément à une position quelconque de la liste 8

3 La liste chainée une deuxième implémentation 9
Description de la liste chainée 9
3.1 Avantages de la liste chainée 9
Suppression d'un élément à l'intérieur de la liste 9
Ajout d'un élément à l'intérieur de la liste 9
3.2 Avantages et inconvénients des listes chainées 10
Inconvénients 10
Avantages 10
3.3 Les listes doublement chainées et les listes circulaires 10
Les listes doublement chainées 10
Listes circulaires doublement chainée 10
3.4 Utilisation des listes chainées 11
3.5 Quelques opérations implémentant l'interface de la liste chainée 11
Opérations disponibles 11
3.6 Exemple d'implémentation d'une liste chainée mise en œuvre 11
Implémentation impérative 12
Implémentation en POO 12

1 Le type abstrait de données

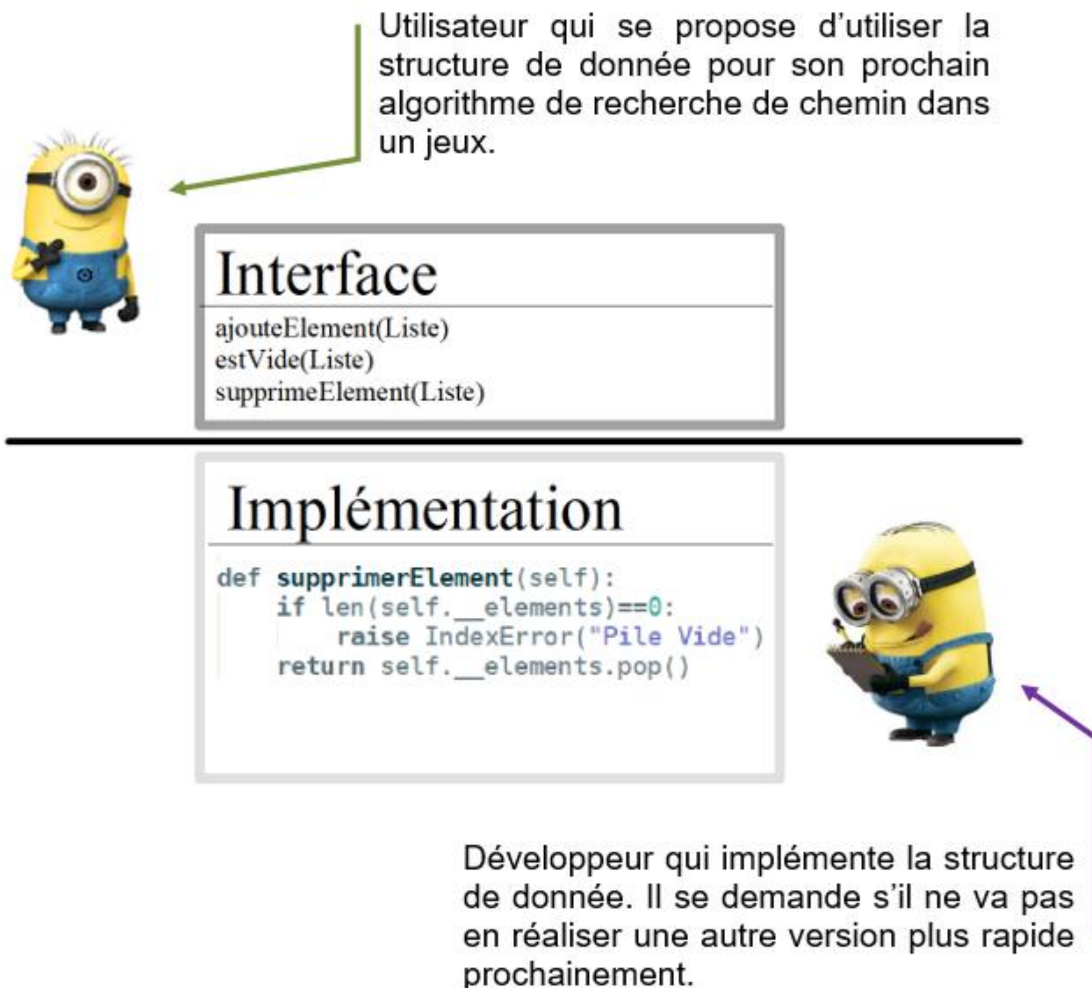
1.1 Définition et utilité

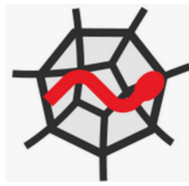
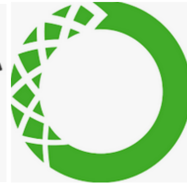
Le type abstrait de données, ou TAD, répond au besoin de définir des structures de données complexes, indépendamment de la manière dont elles sont implantées dans le système de traitement de l'information utilisé.

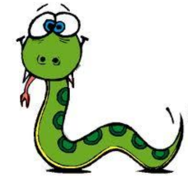
Deux niveaux permettent de définir un TAD : l'interface et l'implémentation

L'interface : permet de décrire la nature de la structure de données, ce qu'elle permet de faire et comment l'utiliser pour résoudre des problèmes d'algorithmique.

L'implémentation : elle réalise effectivement la mise en œuvre du TAD dans un langage de programmation particulier sur un système donné. Cette implémentation doit réaliser tous les attendus exprimés dans l'interface.





1.2 Avantages du concept de Type Abstrait de Données[1]

Le découpage entre l'interface et l'implémentation d'un TAD est très important :

L'utilisateur n'a pas à connaître comment le TAD est implémenté. Il n'a accès qu'à ce que le TAD peut faire et comment faire pour s'en servir : donc uniquement à l'interface. Il reste concentré sur son algorithme sans être 'parasité' par 'oui mais comment je vais réaliser ceci ou bien cela'.

Le développeur réalise le codage du TAD en fonction des spécifications attendues et précisées dans l'interface. Il a en charge la maintenance et l'amélioration de cette implémentation. Comme il ne modifie pas l'interface lors de ses améliorations le comportement attendu du TAD n'est pas modifié, ni la façon de l'utiliser et donc toute modification ou amélioration est transparente pour l'utilisateur final. Plusieurs implémentations sont donc possibles.

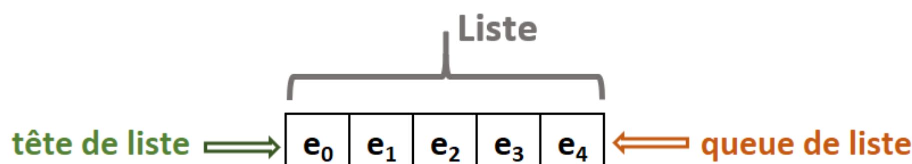
2 Un premier type abstrait de données, la liste

2.1 Description du type abstrait liste

Attention : le TAD Liste ne correspond pas au type *list* de Python. Même si ce type sera utilisé pour l'implémentation car étant assez proche. **! ! !**

Organisation

La liste est une collection d'objets organisée séquentiellement. Les éléments sont placés dans une certaine position dans la structure de données. Pour chacun d'entre eux il y a les éléments situés '*avant*' et les autres situés '*après*'. Il n'y a pas d'autres relations hiérarchiques.



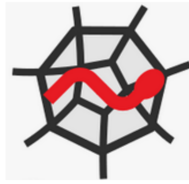
Dans la liste représentée ci-dessous l'élément e_2 est précédé des éléments e_0 et e_1 et suivit des éléments e_3 et e_4.

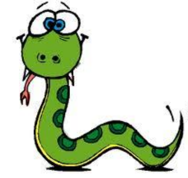
Opérations disponibles

Voilà la liste de quelques opérations disponibles avec une structure de données liste appelée L :

estVide(L) : Renvoi un Booléen à *Vrai* si la liste ne contient aucun élément sinon renvoi la valeur *Faux*.
longueur(L) : Renvoi le nombre d'éléments de la liste L

[1] Ressource utilisée Algorithmique avancée DIU EIL Grenoble Bloc 5, Vincent Danjean

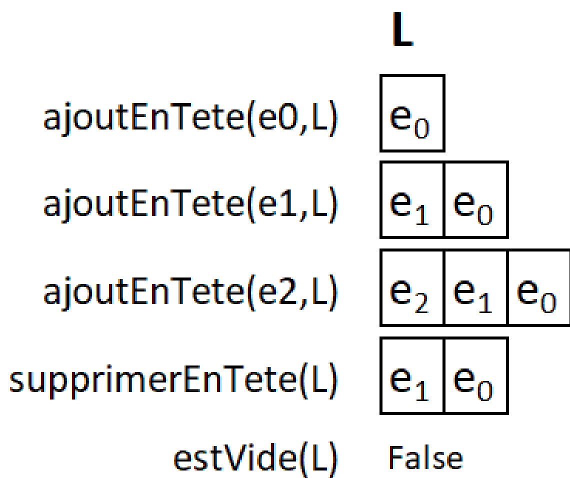




ajoutEnTete(x,L) : Ajoute la valeur x à la liste L.
supprimerEnTete(L) : Renvoi la valeur présente en tête de liste L et la supprime de la liste. Précondition : la liste ne doit pas être vide.
supprimerEnQueue(L) : Renvoi la valeur présente en queue de liste L et la supprime de la liste. Précondition : la liste ne doit pas être vide.

Exemples d'utilisation

Un premier exemple :



Q1. Compléter le tableau résultat de l'exécution des instructions ci-dessous :

	L			
ajoutEnTete(4,L)				
ajoutEnTete(5,L)				
supprimerEnTete(L)				
ajoutEnTete(-3,L)				
ajoutEnTete(6,L)				
supprimerEnQueue(L)				
estVide(L)				

2.2 La liste une première implémentation

Il est temps de réaliser une première implémentation de notre type abstrait liste tel qu'il est défini dans le paragraphe précédent.

Retour sur la liste en Python

Nous utilisons le paradigme de représentation objet. Le type de données Python que nous allons utiliser ici est la liste. Il faut nous rappeler comment ce type de données est implémenté dans ce langage et quelles sont les méthodes disponibles.

Q2. Reprendre dans le document[2] étudié en 1ère le paragraphe sur les listes et remplir le tableau page suivante. NSI_TYPES_CONSTRUITS.pdf

[2] http://sti2dvox.patgue.com/NSI_1ERE/Fichiers_2020/Types_construits/NSI_TYPES_CONSTRUITS.pdf

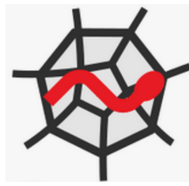
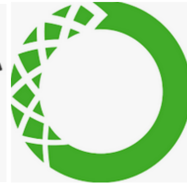
Retour sur les listes Python et quelques méthodes associées

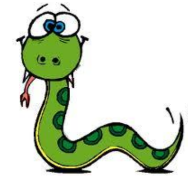
Fonction réalisée	**Code Python**	**Complexité en fonction de n la taille de la liste**
Création d'une liste		
Calcul de la longueur		$O(1)$
Accès à un élément en position index de la liste		$O(1)$
Modification d'un élément en position index de la liste		$O(1)$
Ajout d'un élément x en queue de liste		$O(1)$
Renvoi et enlève le dernier élément inséré dans la liste		$O(1)$
Supprime un élément en position index de la liste		$O(n)$
Insère un élément x à une position index quelconque de la liste		$O(n)$
Renvoi et enlève l'élément en position index dans la liste		$O(n)$
Recherche d'un élément		$O(n)$

Pour mémoire :

- *O* (1) signifie que le temps de calcul est en temps constant donc identique quel que soit la taille de la liste.
- *O* (n) signifie que le temps de calcul est proportionnel à la quantité de données présente dans la liste.

Q3. Expliquez d'où provient, à votre avis, la complexité en O(n) de certaines opérations.



Implémentation des opérations décrites dans l'interface

Création de la classe :

```
class Liste:
    '''
    Implémentation du type abstrait liste.
    P.G 10/10/2020

    '''
    def __init__(self):
        self.__elements = []
```

Ajout des méthodes implémentant l'interface du TAD Liste

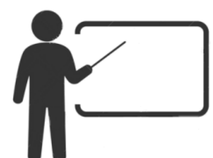
estVide(L)	`def estVide(self):`
	`    if len(self.__elements)==0:`
	`        return True`
	`    else:`
	`        return False`

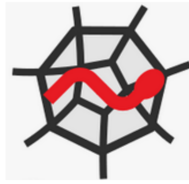
ajoutEnTete(x,L)	`def ajoutEnTete(self,x):` `    self.__elements.append(x)`

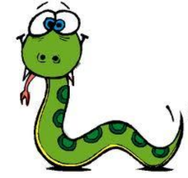
supprimerEnTete(L)	`def supprimerEnTete(self):`
	`    if len(self.__elements)==0:`
	`        raise IndexError("La pile est vide")`
	`    return self.__elements.pop()`

supprimerEnQueue(L)	`def supprimerEnQueue(self):`
	`    if len(self.__elements)==0:`
	`        raise IndexError("La pile est vide")`
	`    return self.__elements.pop(-1)`

Script_TAD_1. Créer un script qui implémente la classe Liste. Puis coder les opérations dans le script pour retrouver le comportement de la liste décrite dans l'exercice Q1.







Q4. Indiquez les numéros de lignes produisant les résultats dans la console comme ci-dessous à partir du script *Main* listé à gauche :

	True
	[10]
	[10,22,-4]
	[10,22,-4,7,28,-12]
	q = -12
	[10,22,-4,7,28]

```
42 ## MAIN
43 ma_liste_1 = Liste()
44 print(ma_liste_1.estVide())
45 ma_liste_1.ajoutEnTete(10)
46 print(ma_liste_1)
47 ma_liste_1.ajoutEnTete(22)
48 ma_liste_1.ajoutEnTete(-4)
49 print(ma_liste_1)
50 ma_liste_1.ajoutEnTete(7)
51 ma_liste_1.ajoutEnTete(28)
52 ma_liste_1.ajoutEnTete(-12)
53 print(ma_liste_1)
54 q = ma_liste_1.supprimerEnQueue()
55 print("q = ",q)
56 print(ma_liste_1)
```

2.3 Améliorations de l'interface

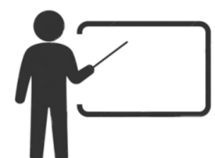
Nos opérations disponibles dans l'interface sont assez limitées. Nous nous proposons d'en ajouter quelques-unes décrites ci-dessous. Malheureusement le développeur pressé n'a pas pris le temps de bien réfléchir aux nouvelles interfaces, il va falloir l'aider et corriger son projet d'amélioration. Les corrections à faire sont indiquées par son manager en souligné rouge.

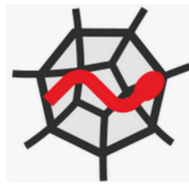
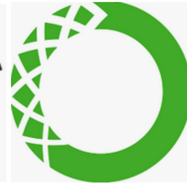
(a) Impression de la liste

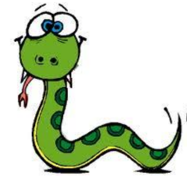
Pour imprimer la liste il faut définir la méthode __str__ . C'est une méthode spéciale, comme __init__, qui doit nous renvoyer une représentation sous forme de chaîne de caractères de la liste. Vous pouvez observer un exemple d'utilisation dans la question précédente. Préconditions aucunes.

```
def __str__(self):
    '''
    Impression du contenu de l'objet liste via la commande print
    '''
```

Script_TAD_2. Compléter le script __str__(self) pour obtenir une belle représentation du contenu de la liste.





(b) Lire un élément à une position quelconque de la liste

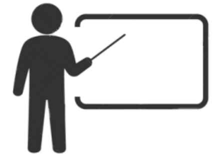
Opération de lecture dans la liste à une position quelconque. L'opération renvoie la valeur lue dans la liste à la position index. L'index varie entre 0 et n-1, n étant le nombre d'éléments.

Précondition aucune.

Q5. Corriger la définition de l'interface en précisant les conditions requises pour la valeur de l'index assurant une réponse correcte à la requête.

```
def getEnPosition(self,index):
    '''
    Retourne la valeur de l'élément en position index
    dans la liste.
    '''
```

Script_TAD_3. Compléter le script.



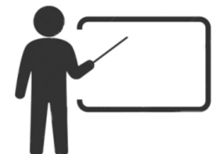
(c) Ajouter un élément à une position quelconque de la liste

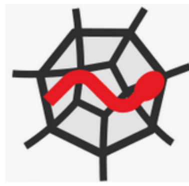
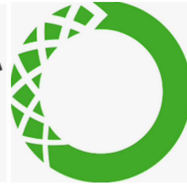
Opération d'écriture d'une valeur x à une position quelconque de la liste.

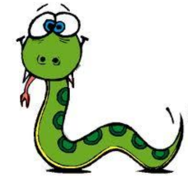
Précondition aucune.

```
def ajoutEnPosition(self,index,x):
    '''
    Ajoute la valeur x en position index
    dans la liste.
    '''
```

Script_TAD_4. Compléter le script en y ajoutant la vérification des préconditions donnant un fonctionnement correct.





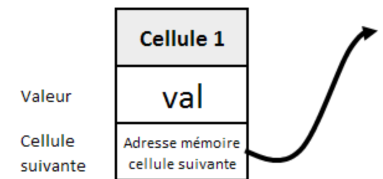
3 La liste chainée une deuxième implémentation

Les éléments des liste Python sont ordonnés et contigus en mémoire. Cela signifie qu'il est difficile d'ajouter, de modifier, ou de supprimer des éléments à une position quelconque du tableau.

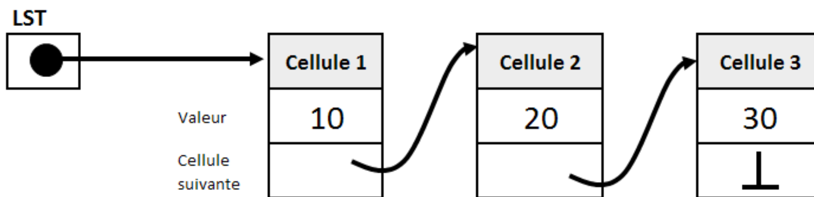
La liste chainée est un TAD qui permet de créer dynamiquement des objets en mémoire. C'est une alternative aux listes vues précédemment.

Description de la liste chainée

Une liste chainée est constituée d'éléments de bases que l'on peut appeler cellule. Chaque cellule contient deux informations, une valeur et l'adresse de la cellule suivante dans la chaine.



Exemple d'une liste chainée de trois cellules

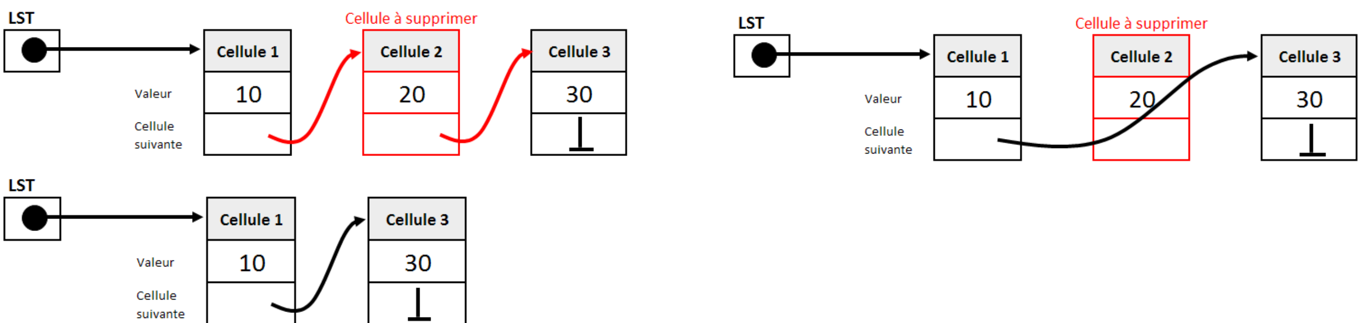


Le symbole ⊥ indique la fin de la liste.

Il se lit 'taquet vers le haut'

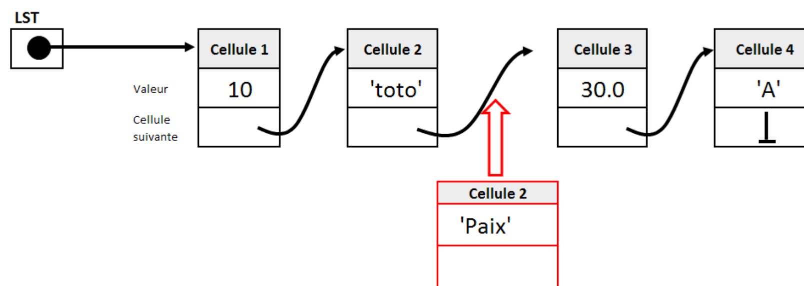
3.1 Avantages de la liste chainée

Suppression d'un élément à l'intérieur de la liste



Ajout d'un élément à l'intérieur de la liste

Q6. Indiquer graphiquement les opérations à effectuer pour insérer une cellule 'Paix' dans la liste suivante :



3.2 Avantages et inconvénients des listes chainées[3]

Inconvénients

- À *nombre d'éléments égal*, une liste chaînée occupe plus de mémoire qu'un tableau, car elle a besoin de stocker également les pointeurs « suivant ».
- Pour accéder à un élément d'une liste chaînée, on est obligé de parcourir tous les maillons jusqu'au maillon recherché. Le *temps d'accès* est donc d'autant plus grand que l'élément recherché est « loin » dans la liste, alors que dans un tableau, tous les éléments peuvent être accédés immédiatement.

Avantages

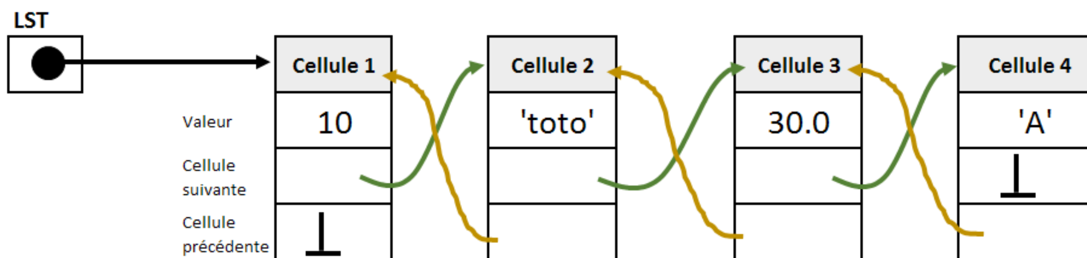
Ces inconvénients sont cependant à relativiser puisque :

- le surcoût en mémoire peut être compensé par le fait qu'on n'est plus obligé d'allouer à l'avance plus d'éléments que nécessaire ;
- le surcoût en temps dépend du type d'utilisation qu'on fera de la liste (cf. l'implémentation de la pile, ci-dessous).

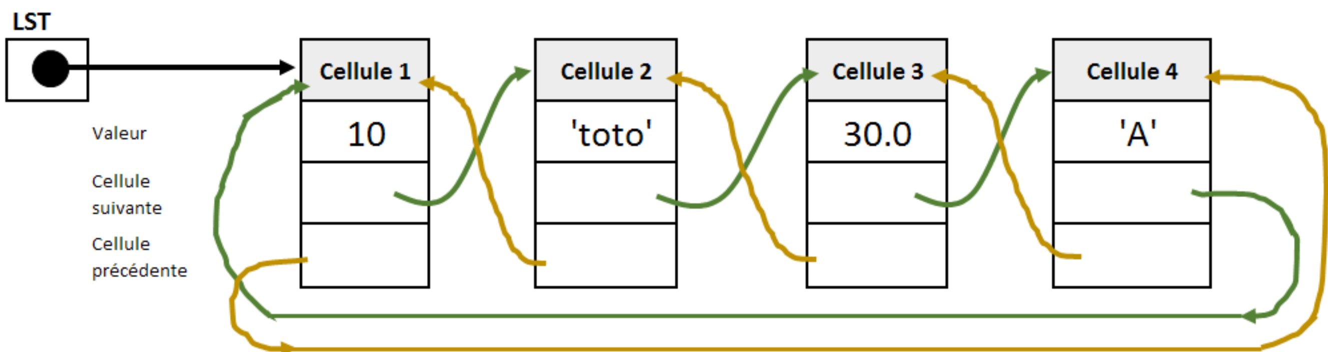
3.3 Les listes doublement chainées et les listes circulaires

Il existe d'autres types de listes chainées à savoir entre autres :

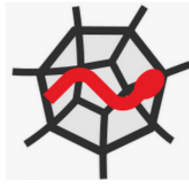
Les listes doublement chainées

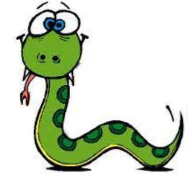


Listes circulaires doublement chainée



[3] https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/algo/cours/sda/liste_chainee.html





3.4 Utilisation des listes chainées

Les listes chainées peuvent être utilisées pour implémenter d'autres types abstraits de données comme les Piles et les Files, objets d'un prochain document de cours.

3.5 Quelques opérations implémentant l'interface de la liste chainée

Opérations disponibles

Voilà la liste de quelques opérations disponibles avec une structure de données liste chainée appelée L, vous les compléterez en exercice :

estVide(L) : Renvoi un Booléen à *Vrai* si la liste ne contient aucun élément sinon renvoi la valeur *Faux*.
longueur(L) : Renvoi le nombre d'éléments de la liste L
ajoutEnTete(x,L) : Ajoute la valeur x au début de la liste L.
ajoutEnFin(x,L) : Ajoute la valeur x à la fin de la liste L.
niemeElement(n,L) : Renvoi le nième élément de la liste, les éléments sont numérotés à partir de 0
Imprime(L) : Réalise l'impression de la liste dans la console

3.6 Exemple d'implémentation d'une liste chainée mise en œuvre

Le maillon de base de nos listes peut être implémenté dans une classe spécifique :

La classe Cellule[4]

```
class Cellule:
    """une cellule d'une liste chaînée"""
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s
```

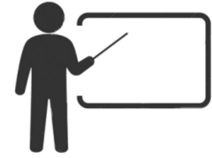
Nous pouvons par exemple utiliser cette classe pour créer la liste chainée de trois cellules vue plus haut de cette manière :

```
lst = Cellule(10,Cellule(20,Cellule(30,None)))
```

[4] L'organisation de l'implémentation de la liste chainée est inspirée du livre NSI Terminale ellipses.

Implémentation impérative

Script_TAD_5. Compléter progressivement le script pour faire fonctionner les fonctions de l'interface.



Exemple de résultat :

```
ENTRAINEMENT SUR LES LISTES
===========================

Création d'une liste de trois éléments
--------------------------------------

Calcul de sa longueur
Algo  récursif :  3
Algo impératif :  3

Impression de la liste
Départ => [Elt: 0 ,Val 10 ] => [Elt: 1 ,Val 20 ] => [Elt: 2 ,Val 30 ] => None

Accès direct à l'élément n°1
Valeur :  20
```

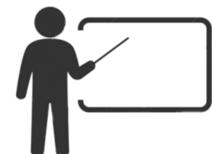
Implémentation en POO[5]

Vous trouverez une implémentation utilisant la POO dans le script Script_TAD_6.py

Voilà un exemple d'utilisation dans la console :

```
>>> (executing lines 1 to 96 of "Script_TAD_6.py")

>>> ma_liste=Liste()

>>> ma_liste.ajoute(10)

>>> ma_liste.ajoute(20)

>>> ma_liste.ajoute(30)

>>> len(ma_liste)
3

>>> ma_liste[0]
30

>>> ma_liste[1]
20

>>> ma_liste[2]
10

>>> ma_liste.estVide()
False
```

Script_TAD_6. Compléter le script en y ajoutant l'opération imprime(L). Puis faites le fonctionner.



[5] D'après le livre NSI Terminale ellipses 24 leçons avec exercices corrigés