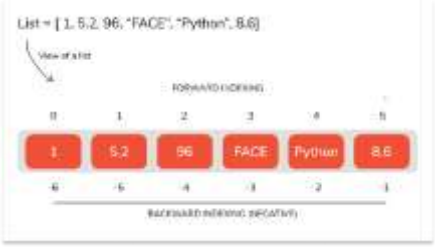
1. Rappels listes en python

Les listes (class list) en python sont des tableaux d'éléments indexés de 0 à n exclu auquel ont peut ajouter et retirer des éléments.




```
list1 = [1,5.2,96,"FACE",8.6]
print(list1)
list1.append(36)
print(list1)
list1.insert(0,45)
print(list1)
list1.pop(0)
print(list1)
```

```
[1, 5.2, 96, 'FACE', 8.6]

[1, 5.2, 96, 'FACE', 8.6, 36]

[45,1,5.2,96,'FACE',8.6,36]

[1, 5.2, 96, 'FACE', 8.6, 36]
```

Les différentes méthodes applicables aux objets list sont :

`list.append`(*x*)
: Ajoute un élément à la fin de la liste. Équivalent à `a[len(a):] = [x]`.

`list.extend`(*iterable*)
: Étend la liste en y ajoutant tous les éléments de l'itérable. Équivalent à `a[len(a):] = iterable`.

`list.insert`(*i*, *x*)
: Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc a.insert(0, x) insère l'élément en tête de la liste et `a.insert(len(a), x)` est équivalent à `a.append(x)`.

`list.remove`(*x*)
: Supprime de la liste le premier élément dont la valeur est égale à x. Une exception ValueError est levée s'il n'existe aucun élément avec cette valeur.

`list.pop`([*i*])
: Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, a.pop() enlève et renvoie le dernier élément de la liste (les crochets autour du i dans la signature de la méthode indiquent que ce paramètre est facultatif et non que vous devez placer des crochets dans votre code !

`list.clear`()
: Supprime tous les éléments de la liste. Équivalent à `del a[:]`.

`list.index`(*x*[, *start*[, *end*]])
: Renvoie la position du premier élément de la liste dont la valeur égale x (en commençant à compter les positions à partir de zéro). Une exception ValueError est levée si aucun élément n'est trouvé.
Les arguments optionnels start et end sont interprétés de la même manière que dans la notation des tranches et sont utilisés pour limiter la recherche à une sous-séquence particulière. L'indice renvoyé est calculé relativement au début de la séquence complète et non relativement à start.

`list.count`(*x*)
: Renvoie le nombre d'éléments ayant la valeur x dans la liste.

`list.sort`(*, *key=None*, *reverse=False*)
: Ordonne les éléments dans la liste (les arguments peuvent personnaliser l'ordonnancement, voir sorted() pour leur explication).

`list.reverse`()
: Inverse l'ordre des éléments dans la liste.

`list.copy`()
: Renvoie une copie superficielle de la liste. Équivalent à `a[:]`.

A retenir list.append(x) , list.insert(0,x) et list.pop(0)

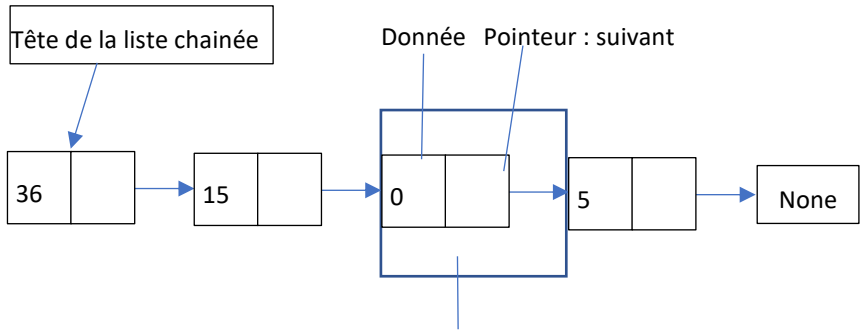
2. Listes chainées

2.1. Définition

Def :

Une liste chainée est une structure de données linéaire. Contrairement aux tableaux (listes en Python) , les éléments d'une liste chainée ne sont pas stockés à un emplacement contigu mais sont liés à l'aide de pointeurs.

Chaque élément de la liste est une cellule ou encore un nœud possédant une valeur et un lien vers la cellule/nœud suivant :



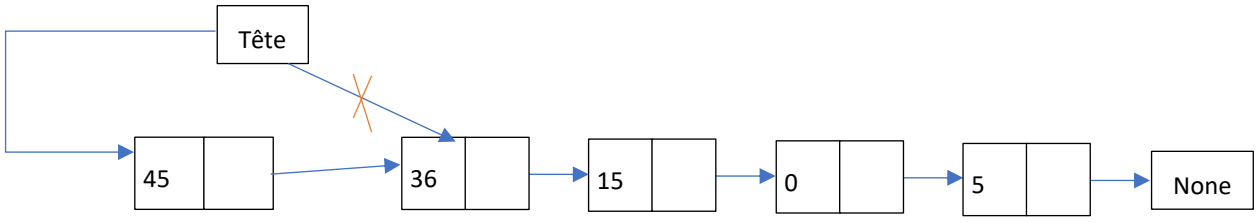
Une Cellule / Nœud / Node

Cellule	`class Cellule :`
	`  def __init__(self,val=None):`
	`  self.val = val`
	`  self.suivant  = None`

Début de liste chainée	`class Liste:`
	`  #liste simplement chainée`
	`  def __init__(self):`
	`  self.tete = None`

2.2. Opérations principales :

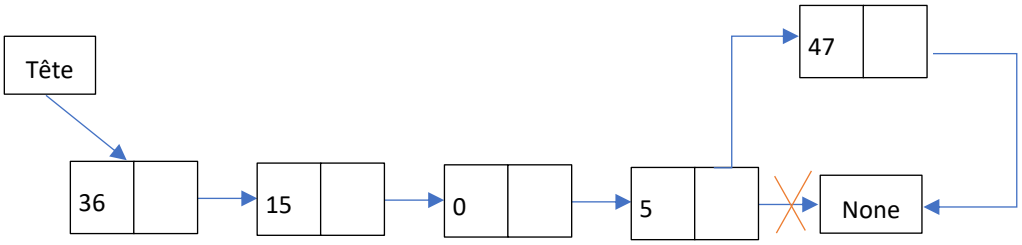
2.2.1. Insérer en tête



Insérer en tête
 	
« PUSH – insert(0,x) »
	`def inserer_en_tete(self,valeur):`
	`  # Créer une nouvelle cellule`
	`  # Y inserer la valeur`
	`  nouvelle_cellule =Cellule(valeur)`
	`  # Mettre la Cellule en tête de la liste`
	`  nouvelle_cellule.suivant =  self.tete`
	`  # Faire pointer la liste sur la cellule`
	`  self.tete = nouvelle_cellule`

Ordre de complexité : $O(1)$

2.2.2. Insérer en queue



Insérer en queue

« append() »

```
def inserer_en_queue(self,valeur):
        # Créer une nouvelle cellule
        # Y inserer la valeur
        nouvelle_cellule = Cellule(valeur)
        # Vérifier si la liste est vide
        # Si c'est le cas faire pointer la tête de liste sur la nouvelle cellule
        if self.tete is None :
            self.tete = nouvelle_cellule
            return
        # Parcourir toute la liste jusqu'à la dernière cellule
        derniere =  self.tete
        while(derniere.suivant):
            derniere = derniere.suivant
        # Ajouter à la dernière cellule de la liste la cellule créée
        derniere.suivant = nouvelle_cellule
```

Ordre de complexité : $O(n)$ (n = taille de la liste)

2.2.3. Est vide

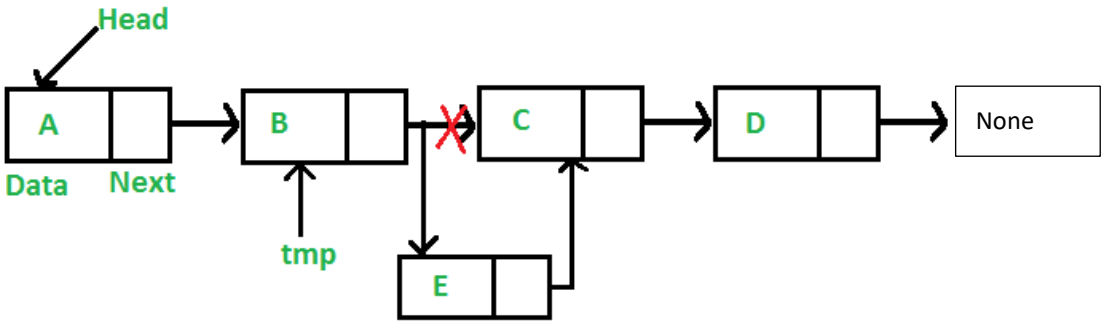
Implémenter une méthode « est-vide » qui renvoie True ou False en fonction de l'existence ou pas de cellule dans la liste chainée.

2.2.4. Taille de la liste chainée

Implémenter une méthode « taille_liste » qui renvoie True ou False en fonction de l'existence ou pas de cellule dans la liste chainée.

2.2.5. Insérer devant la valeur val à la position de la i^{eme} cellule

Compléter le code de la méthode «inserer_apres_position(self,position,valeur) » qui insère une nouvelle cellule dans la liste chainée après la i^{ème} cellule.



```
def inserer_apres_position(self,position,valeur):
        # Vérifier que la position choisie permette l'insertion
        taille =  self.taille_liste()
        if taille < position :
            print("Insertion impossible position supérieure à la taille de la
liste ")
        elif position  == 0 :

        elif taille == position :

        else :
            compteur = 0
            # Parcourir toute la liste jusqu'à la position voulue
            derniere =  self.tete
            while(compteur < position -1 ):
                compteur +=1

            # Créer une nouvelle cellule
            # Y inserer la valeur

            #Créer une cellule temporaire pour garder en mémoire la position d
e la coupure

            #Faire pointer la suite de la coupure  sur la nouvelle cellule
            derniere.suivant = nouvelle_cellule
            #Faire poinyer le suivant de la nouvelle cellule sur la coupure ga
rdée en mémoire
```

3. Mise en évidence du chainage

Pour mettre en évidence le chainage : autrement dit la succession des adresses en mémoire entre deux cellules ajouté le code suivant à la cellule.

```
class Cellule :
    def __init__(self,val=None):
        self.val = val
        self.suivant  = None
    def affiche(self):
        print(f"Valeur{self.val} position :  {hex(id(self))} suivant : {hex(id
(self.suivant))}")
```

Implémenter une nouvelle méthode affichage :

```
def affichage_liste(self):
        temporaire = self.tete
        print("Valeurs de la liste : ")
        while(temporaire):
                                    # affiche les caractéristiques de la cellule
            temporaire = temporaire.suivant
        print("")
```

`Ma_Liste = Liste()`
 	
`Ma_Liste.inserer_en_tete(1)`
`Ma_Liste.inserer_en_tete(2)`
`Ma_Liste.inserer_en_tete(3)`
`Ma_Liste.inserer_en_tete(4)`
`Ma_Liste.inserer_en_queue(5)`
`Ma_Liste.inserer_en_queue(6)`
 	
`Ma_Liste.affichage_liste()`
	Valeurs de la liste :
	Valeur4 position : 0x25008087cf8 suivant : 0x25008087828
	Valeur2 position : 0x25008087828 suivant : 0x250080877f0
	Valeur1 position : 0x250080877f0 suivant : 0x25008087d30
	Valeur5 position : 0x25008087d30 suivant : 0x25008087d68
	Valeur6 position : 0x25008087d68 suivant : 0x7ffb4d004ce0