LISTES CHAINEES 1/2

- En python, ce que l'on désigne par le terme liste est en réalité ce que l'on appelle en algorithmique un tableau, c'est à dire une suite d'éléments contigus et ordonnés en mémoire. Ces éléments sont stockés en mémoire les uns à la suite des autres.

Exemple :

```
In [1]: T=[1,1,2,3,5]
        print(id(T[0]))
        print(id(T[1]))
        print(id(T[2]))
        print(id(T[3]))
        print(id(T[4]))
```

```
140722133210784
140722133210784
140722133210816
140722133210848
140722133210912
```

- Les tableaux permettent efficacement d'insérer ou de supprimer un élément à la fin du tableau, grâce aux méthodes `append()` et `pop()`.
- En revanche, ils le sont beaucoup moins lorsqu'il s'agit d'insérer ou de supprimer un élément à une autre position que la dernière.

Exercice 1:

On considère le tableau ci-dessous :

$$| 1 | 1 | 3 | 5 | 8 |$$

On souhaite insérer la valeur 0 en première position et obtenir le tableau suivant:

$$| 0 | 1 | 1 | 3 | 5 | 8 |$$

- On commence par agrandir le tableau, en ajoutant un nouvel emplacement à la fin :

$$| 1 | 1 | 3 | 5 | 8 | \quad |$$

- Il faut ensuite décaler tous les élements du tableau initial vers la droite en commençant par le dernier :

$$| 1 | 1 | 1 | 3 | 5 | 8 |$$

- Enfin, on écrit la valeur souhaitée dans la première case du tableau :

$$| 0 | 1 | 1 | 3 | 5 | 8 |$$

Questions :

Considérons maintenant un tableau qui comporte 1 millions d'éléments.

1. Combien de déplacements seront nécéssaires pour insérer une valeur en première position ?
2. Et si l'on souhaite supprimer le premier élément ?

Réponse :

1.
2.

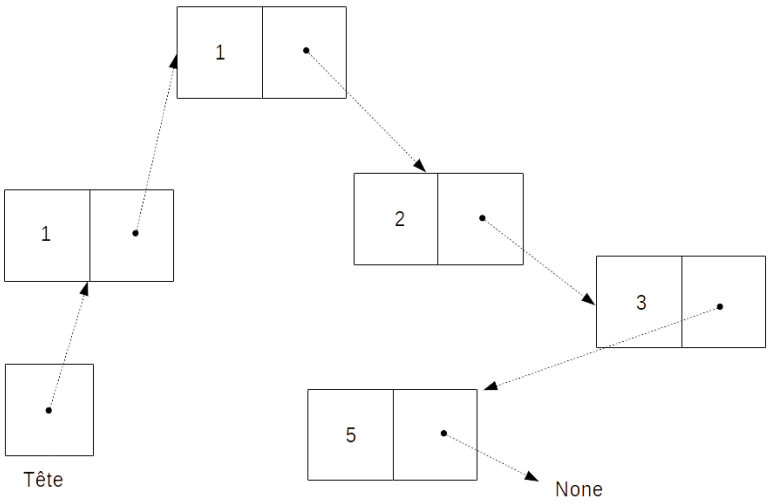
Remarques :

- Dans cette feuille, nous allons étudier une structure de données qui apporte une meilleure solution aux problèmes de l'insertion et de la suppression : la liste chaînée.
- Nous verrons aussi plus tard qu'elle permet de construire d'autres structures de données.

1. Définition

Principes

Voici ci-contre une représentation de la liste chaînée $1, 1, 2, 3, 5$



- Une liste chaînée permet de représenter une séquence finie d'éléments.
- Comme le nom le suggère, ces éléments sont liés entre eux comme le sont les maillons d'une chaîne, nous appellerons ces maillons des cellules.
- Chaque cellule contient deux informations :
  - La valeur de l'élément.
  - L'adresse mémoire de la cellule contenant l'élément suivant.
- La dernière cellule contient la valeur du dernier élément et également l'information spéciale `None` qui indique qu'il n'y a pas de suivant.
- L'adresse de la première cellule est appelée tête de la liste.

Remarques :

- Avec cette définition, il n'y a pas d'accès direct aux valeurs des éléments.On avance dans la liste à partir de la tête, jusqu'à l'élément souhaité.
- Cette représentation occupe un espace mémoire important puisqu'il faut stocker pour chaque cellule, une valeur et une adresse.
- Elle est néanmoins performante en temps d'éxécution pour certaines opérations.

Encapsulation dans un objet

Nous allons utiliser le paradigme de la programmation objet pour implémenter ce concept en python et définir deux classes : la classe `Cell` qui définit une cellule et la classe `Lc` qui définit une liste chaînée, classe pour laquelle nous ajouterons ensuite des méthodes pour effectuer des opérations usuelles sur les listes.

Exercice 2 : Définition d'une cellule

La classe `Cell` contient deux attributs initialisés par la méthode constructeur :

- `valeur` : Contient la valeur de la cellule définie
- `suivant` : Contient l'adresse mémoire de la cellule suivante, par défaut la valeur `None`

Ecrire la méthode `__str(self)__` qui doit permettre d'afficher la valeur de la cellule.

In [2]:
```
class Cell:
    '''Cellule d'une liste chainee'''
    def __init__(self,valeur,suivant=None):
        self.valeur=valeur
        self.suivant=suivant




c1=Cell(1)
print(c1)
```

```
1
```

Construction d'une première liste

Voici alors une première façon de construire la liste chaînée $1, 1, 2, 3, 5$ à l'aide de la classe `Cell` :

In [3]:
```
L=Cell(1,Cell(1,Cell(2,Cell(3,Cell(5,None)))))
```

- La variable `L` contient l'adresse mémoire de l'objet contenant la valeur 1 qui lui même contient l'adresse de l'autre objet contenant 1 qui lui même contient l'adresse de l'objet contenant 2 qui lui même contient l'adresse de l'objet contenant 3 qui lui même contient l'adresse du dernier objet contenant la valeur 5 et l'attribut `None` .
- Il s'agit d'une définition récursive de la notion de liste, une liste est ici une cellule.
- Cette implémentation est cependant incomplète, car il n'est pour l'instant pas possible d'afficher une version plus lisible de cette liste :

In [4]:
```
print(L)
```

```
1
```

Exercice 3 : Définition d'une liste

Nous allons construire une classe `Lc` (liste chaînée) qui va permettre de compléter l'implémentation. Cette classe contient un attribut `tete` initialisé par le constructeur avec la valeur par défaut `None` . Cet attribut est simplement le lien vers l'adresse de la première cellule.

In [6]:
```
class Lc:
    '''Liste chaînée'''
    def __init__(self, t=None):
        '''tete : lien vers la premiere cellule'''
        self.tete=t
```

1. Compléter la construction de la liste 1,1,2,3,5 ci-dessous en utilisant les classes `Cell` et `Lc` .

In [7]:
```
c1=Cell(1)
c2=Cell(1)
c1.suivant=c2




#
```

1. Afficher :
   - La valeur de la tête.
   - La valeur du deuxième élément à partir de la tête.
   - La valeur du dernier élément à partir de la tête.

In [13]:
```
#valeur de la tête


#valeur du 2nd élément


#valeur du dernier élément
```

```
1
1
5
```

Exercice 4: Compléter la méthode `__str(self)__` de la classe `LC` qui doit permettre d'afficher sous forme lisible la liste considérée .

Ainsi , si `L` est la liste définie plus haut, `print(L)` renvoie `<1→ 1→ 2→ 3→ 5→ ∅` .

Le caractère `<` indique la cellule de tête, le caractère `→` indique le lien vers la cellule suivante et le caractère `∅` indique `None`

Si la liste est vide, alors la chaîne `∅` est renvoyée.

In [120]:
```
class Lc:
    '''Liste chaînée'''
    def __init__(self, t=None):
        '''tete : lien vers la premiere cellule'''
        self.tete=t

    def __str__(self):
        '''renvoie une forme lisible de Lc'''
        if self.tete is None:
            return ...
        else:
            cellule=self.tete
            valeurs='<'+ str(cellule.valeur)
            while cellule.suivant is not None:
                cellule=...
                valeurs=...       + '→ ' + str(cellule.valeur)

            return ...
```

In [121]:
```
#tests
L=Lc(Cell(1,Cell(1,Cell(2,Cell(3,Cell(5,None))))))
print(L)

L1=Lc(Cell(1,None))
print(L1)

print(Lc())
```

```
<1→ 1→ 2→ 3→ 5→ ∅
<1→ ∅
∅
```

Exercice 5:

Compléter la fonction `listeN(n)` qui renvoie une liste chaînée contenant les n premiers entiers de 1 à n. Si $n \leq 0$, on renvoie une liste vide.

In [122]:
```
def listeN(n):
    '''liste des n premiers entiers de 1 à n
    parametre : n entier >0
    return : liste chaînée
    '''
    L=Lc()
    if n <=0 :
        ...
    else:
        cellule=Cell(1)
        L.tete=...

    for i in range(...,...):
        cellule.suivant=Cell(i)
        cellule=...

    return L
```

In [123]:
```
#Tests
print(listeN(0))
print(listeN(1))
print(listeN(10))
```

```
∅
<1→ ∅
<1→ 2→ 3→ 4→ 5→ 6→ 7→ 8→ 9→ 10→ ∅
```

2. Méthodes

Dans cette partie, nous allons compléter au fur et à mesure la classe `Lc` ci-dessous avec des méthodes permettant de réaliser des opérations que l'on fait habituellement avec les tableaux python.

In [124]:

```
class Cell:
    '''Cellule d'une liste chainee'''
    def __init__(self,valeur,suivant=None):
        self.valeur=valeur
        self.suivant=suivant

    def __str__(self):
        return str(self.valeur)

class Lc:
    '''Liste chaînée'''
    def __init__(self, t=None):
        '''tete : lien vers la premiere cellule'''
        self.tete=t

    def __str__(self):
        '''renvoie une forme lisible de Lc'''
        if self.tete is None:
            return 'ø'
        else:
            cellule=self.tete
            valeurs='<'+ str(cellule.valeur)
            while cellule.suivant is not None:
                cellule=cellule.suivant
                valeurs=valeurs + '→ ' + str(cellule.valeur)

            return valeurs + ' →ø'

    #Ex 6
    def est_vide(self):
        '''renvoie True si la liste est vide
        False sinon'''




        pass


    #Ex 7
    def __len__(self):
        '''renvoie la longueur de la liste'''
        n=0







        return n

    #Ex 8
    def __getitem__(self, index):
        '''renvoie l'élement  d'index donné,
           numéroté à partir de 0'''









        return

    #Ex 9
    def inserer(self,x,index):
        '''insere l'élément x dans la liste
        à l'index donné, numéroté à partir de 0'''
        i=0
        cellule=self.tete
```

```
        if cellule is None:
            self.tete=...

        elif index == 0:
            self.tete=...

        else :
            while ... :
                cellule=cellule.suivant
                i=i+1

            nouv_cellule=...
            ...     =nouv_cellule

    #Ex 10
    def supprimer(self,index):
        ''' Supprime l'élément d'index donné
        numéroté  partir de 0, de la liste'''









        pass
```

Exercice 6 : Liste vide

La méthode `est_vide(self)` doit renvoyer `True` si la liste est vide (c'est à dire si l'attribut `tete` à la valeur `None`) et `False` sinon.

In [125]:
```
#Ex6 : Tests liste vide
L=Lc(Cell(1,Cell(1,Cell(2,Cell(3,Cell(5,None))))))
print(L, L.est_vide())

L1=Lc()
print(L1, L1.est_vide())
```

```
<1→ 1→ 2→ 3→ 5 →ø False
ø True
```

Exercice 7 : Longueur d'une liste

Ecrire La méthode `__len(self)__` qui renvoie la longueur de la liste c'est à dire son nombre de cellules. Il s'agit de parcourir la liste de la première cellule à la dernière, en suivant les liens qui relient les cellules entre elles. On peut réaliser ce parcours au choix avec une fonction récursive ou avec une boucle. Dans cet exercice, nous allons utiliser une boucle.

In [127]:
```
#Ex7 : Tests longueur de la liste

L=Lc(Cell(1,Cell(1,Cell(2,Cell(3,Cell(5,None))))))
print(L, len(L))

L1=Lc()
print(L1, len(L1))
```

```
<1→ 1→ 2→ 3→ 5 →ø 5
ø 0
```

Remarques :

- Les instructions `len(L)` et `L.__len__()` sont équivalentes(cela fonctionne par ailleurs avec les tableaux python)
- Complexité du calcul du nombre d'éléments de la liste :
  - Quelque soit le nombre de cellules, il faut les parcourir toutes.
  - La complexité du calcul est donc proportionnelle au nombre n de cellules, en $\mathcal{O}(n)$
  - Pour 1000 cellules, il faudra donc effectuer 1000 tests (`while`), 1000 additions (`n+1`), et 2000 affectations (`=`), soit 4000 opérations élémentaires.

Exercice 8 : Accès aux éléments

Ecrire la méthode `__getitem__(self,index)` . Cette méthode renvoie la valeur contenue dans la cellule d'index donnée, compté à partir de 0, index de la première cellule. Comme pour l'exercice 6, on peut écrire cette méthode de façon récursive ou avec une boucle, nous allons le faire avec une boucle.L'idée est d'avancer de cellule en cellule jusqu'à l'indice voulu et renvoyer alors la valeur de la cellule.Lorsque la liste est vide ou que l'index demandé est supérieur à la longueur de la liste, le message `index error` sera renvoyé.

Remarques :

- Les instructions `L[i]` et `L.__getitem__(i)` sont équivalentes(cela fonctionne par ailleurs avec les tableaux python)
- Complexité du calcul de l'accès à un élément :
  - Cela dépend de la valeur de `index` :
    - Dans certains cas, il faut autant de passages dans la boucle `while` que de cellules à parcourir jusqu'à l'index demandé.
    - Dans le pire des cas, il faut parcourir toute la liste (par exemple lorsque l'index est supérieur à égal à celui de la dernière cellule)

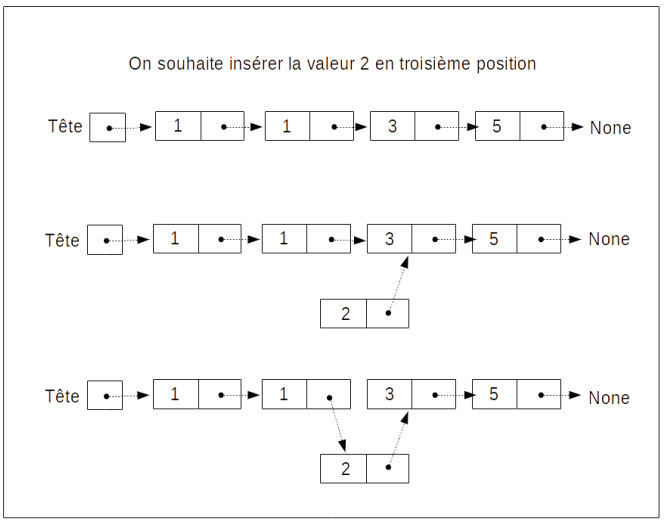
In [128]:
```
#Ex8 : Tests accès aux éléments

L=Lc(Cell(1,Cell(1,Cell(2,Cell(3,Cell(5,None))))))
print(L, L[0], L[5])

L1=Lc()
print(L1, L1[0])
```

```
<1→ 1→ 2→ 3→ 5 →ø 1 index error
ø index error
```

Exercice 9 : Insertion

Le but de cet exercice est d'écrire la méthode `inserer(self, x, index)` qui insère l'élément `x` à l'index donné en paramètre numéroté à partir de 0.

- On envisagera d'abord les cas particuliers ou :
  - La liste est vide.
  - `index` est égal à 0 (insertion en début de liste).
- Cas général(voir exemple ci-contre):
  - On avance dans la liste jusqu'à la cellule numéro `index-1`
  - On crée une nouvelle cellule de valeur `x` et liée à la cellule numéro `index`
  - On lie la cellule numéro `index-1` à la nouvelle cellule
- Bonus : Si l'index est absurde, renvoyer `index error` (ajouter des tests adéquats)



In [129]:
```
#Ex9 : Tests insertion d'élément

#insérer dans une liste vide
L1=Lc()
print(L1)
L1.inserer(1,0)
print(L1)
```

```
ø
<1 →ø
```

In [130]:
```
#génération de la liste 1,1,3,5
L=Lc(Cell(1,Cell(1,Cell(3,Cell(5,None)))))
print(L)

#insérer au début de la liste
L.inserer(0,0)
print(L)

#inserer dans la liste
L.inserer(2,3)
print(L)

#insérer à la fin de la liste
L.inserer(8,len(L))
print(L)
```

```
<1→ 1→ 3→ 5 →ø
<0→ 1→ 1→ 3→ 5 →ø
<0→ 1→ 1→ 2→ 3→ 5 →ø
<0→ 1→ 1→ 2→ 3→ 5→ 8 →ø
```

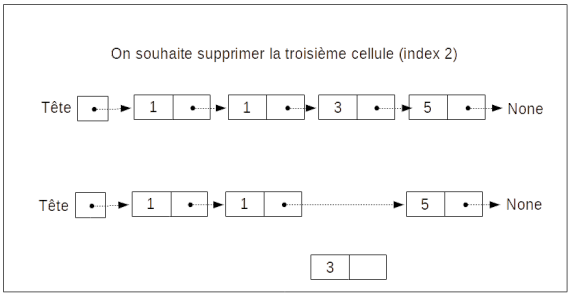
Remarques :

- On voit ici l'éfficacité de l'insertion dans une liste chaînée en début de liste. Il est inutile de décaler des éléments comme on le ferait pour un tableau, il suffit de créer une cellule à placer en tête et la lier à la cellule qui était précédemment en tête.
- Dans ce cas la complexité de calcul est en temps constant(en $\mathcal{O}(1)$) quelque soit la longueur de la liste !

Exercice 10 : Suppression

Le but de cet exercice est d'écrire la méthode `supprimer(self,index)` qui supprime l'élément `x` à l'index donné en paramètre numéroté à partir de 0.

- On envisagera d'abord le cas particulier ou `index` est égal à 0 (le premier élément est supprimé) :
- Cas général(voir exemple ci-contre):
  - On avance dans la liste jusqu'à la cellule numéro `index-1`
  - On lie la cellule numéro `index-1` à la cellule numéro `index+1`
- Bonus : Si l'index est absurde, renvoyer `index error` (ajouter des tests adéquats).



In [131]:
```
#Ex10 : Tests suppression d'élément

#génération de la liste 1,1,3,5
L=Lc(Cell(1,Cell(1,Cell(1,Cell(2,Cell(3,Cell(5,None)))))))
print(L)
```

```
<1→ 1→ 1→ 2→ 3→ 5 →ø
```

In [132]:
```
#supprimer au début de la liste
L.supprimer(0)
print(L)
```

```
<1→ 1→ 2→ 3→ 5 →ø
```

In [133]:
```
#supprimer dans la liste
L.supprimer(2)
print(L)
L.supprimer(2)
print(L)
L.supprimer(2)
print(L)
```

```
<1→ 1→ 3→ 5 →ø
<1→ 1→ 5 →ø
<1→ 1 →ø
```

In [134]:
```
#supprimer à la fin de la liste
L.supprimer(len(L)-1)
print(L)

#supprimer le seul élément de la liste
L.supprimer(0)
print(L)
```

```
<1 →ø
ø
```

Remarques :

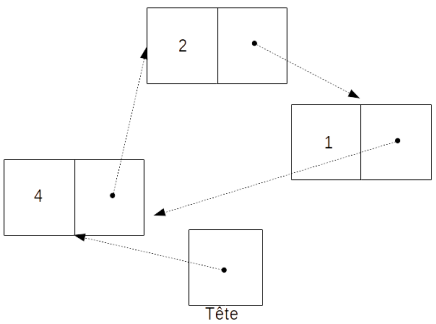
- Encore une fois la suppression dans une liste chaînée en début de liste est efficace. Il est inutile de décaler des éléments comme on le ferait pour un tableau, il suffit de bien choisir la cellule à placer en tête de liste.
- Dans ce cas la complexité de calcul est en temps constant(en $\mathcal{O}(1)$) quelque soit la longueur de la liste !

3. Autres types de listes chaînées

Il existe de nombreuses variantes de la structure de liste chaînée, dont celles présentées ci-dessous :

Listes cycliques

- Dans cette structure , le dernier élément est lié au premier.
- L'utilisation de ce type de liste requiert des précautions pour éviter des parcours infinis, par exemple, lors d'une recherche vaine d'élément.



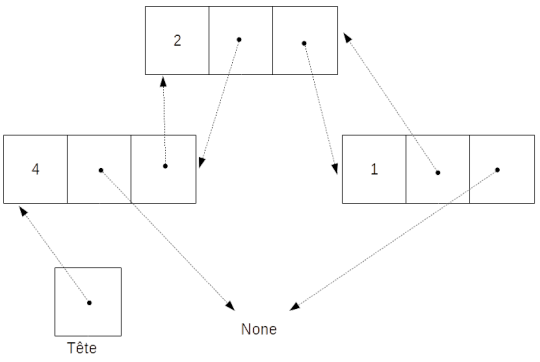
Exercice 11 :

1. Compléter les instructions situées après le code des classes pour construire la liste cyclique $4, 2, 1$.
2. Modifier la méthode `__str(self)__` pour permettre l'affichage d'un cycle (ici `<4→ 2→ 1 →<`)

In [135]:
```
#liste chaînée cyclique

class Cell:
    '''Cellule d'une liste chainee'''
    def __init__(self,valeur,suivant=None):
        self.valeur=valeur
        self.suivant=suivant

    def __str__(self):
        return str(self.valeur)

class Lcc:
    '''Liste chaînée cyclique'''
    def __init__(self, t=None):
        '''tete : lien vers la premiere cellule'''
        self.tete=t

    def __str__(self):
        '''renvoie une forme lisible de Lc'''
        if self.tete is None:
            return 'ø'
        else:
            cellule=self.tete
            valeurs='<'+ str(cellule.valeur)
            while cellule.suivant is not None:
                cellule=cellule.suivant
                valeurs=valeurs + '→ ' + str(cellule.valeur)

            return valeurs + ' →ø'


#1.
n1=Cell(4)
n2=Cell(2)
n3=Cell(1)
```

Listes doublement chaînées

- Dans cette structure, chaque élément est lié à l'élément suivant et à l'élément précédent (s'ils existent).
- L'espace mémoire occupé est encore plus important puisque chaque cellule contient une valeur et deux liens vers d'autres cellules.
- Cette structure facilite néanmoins l'accès aux éléments.



Exercice 12 :

Modifier les classes ci-dessous pour que le code s'éxécute sans erreurs.

In [94]:
```python
#liste chaînée cyclique

class Cell2:
    '''Cellule d'une liste doublement chaînee'''
    def __init__(self,valeur,suivant=None):
        self.valeur=valeur
        self.suivant=suivant


    def __str__(self):
        return str(self.valeur)

class Ldc:
    '''Liste doublement chaînée'''
    def __init__(self, tete=None):
        '''tete : lien vers la premiere cellule'''
        self.tete=tete


n1=Cell2(4)
n2=Cell2(2)
n3=Cell2(1)

L=Ldc(n1)

print(n1.precedent) # None
print(n2.suivant) # 1
print(n2.precedent) # 4
print(n3.suivant) # None
print(L.tete.suivant) # 2
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-94-54059edbe00e> in <module>
     23 L=Ldc(n1)
     24
---> 25 print(n1.precedent) # None
     26 print(n2.suivant) # 1
     27 print(n2.precedent) # 4

AttributeError: 'Cell2' object has no attribute 'precedent'
```

Liste doublement chaînée cyclique

- C'est une combinaison des deux structures précédentes.
- La première cellule est liée à la dernière et chaque cellule est liée à la précédente et à la suivante.