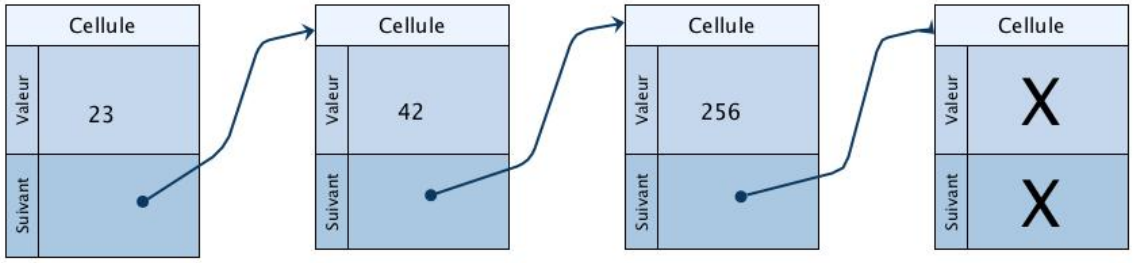
Remarque préliminaire : Jupyter est parfois capricieux pour le téléchargement des images. Si les images n'apparaissent pas dans le notebook, chargez les dans le même dossier que le notebook. Les adresses se trouvent en double cliquant dans les cellules de texte (là où il y a précisé "image", c'est qu'il y a une image normalement...). Puis changez le code comme ceci : `![Image : listes](http://www.maths-info-lycee.fr/images/arbre1.jpg)` devient `![Image : listes](imagearbre1.jpg)` ou même `![](imagearbre1.jpg)`

Structures de données linéaires : les listes

Arbre unaire (ou dégénéré, ou filiforme)

Un arbre unaire est un arbre dans lequel un noeud comporte au plus un unique sous-arbre. Prenons un tel arbre et basculons-le sur le côté, de telle manière que la racine soit à gauche, et l'unique feuille à droite : on obtient alors ce que l'on appelle une liste en informatique générale (et non sécifiquement en Python, cf. ci-après).



Les listes

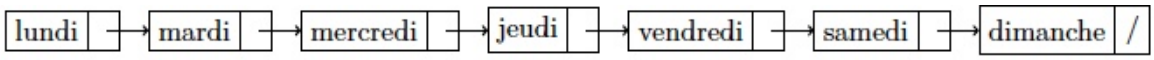
Une première *remarque* fondamentale : on ne parle pas du type `list` vu en Python en première. Le type `list` de Python est en fait un tableau dynamique.

Une liste en informatique est une suite d'éléments. Cette suite est finie, et peut être vide. Chaque élément de la suite est repérée par son indice : la liste est ordonnée par l'indice (et non par la valeur de l'élément).

Deuxième *remarque* : une liste informatique n'est pas non plus ce qu'on appelle une liste dans le langage courant. Quand on a une liste de courses, on ne suit pas l'ordre de la liste pour faire ses courses. Et on ne met pas deux fois le même ingrédient (sauf étourderie). Dans un style plus littéraire, vous pouvez lire les notes de chevet (枕草子, Makura no sōshi) de Sei Shōnagon, écrites vers 990. Ce sont des listes poétiques : "Choses dont on néglige souvent la fin", "Choses que l'on méprise", "Choses qui font battre le cœur", "Fleurs des arbres", "Cascades"...

Exemple à compléter : On donne la liste `L1` des jours de la semaine :

```
L1 = [lundi , mardi, mercredi , jeudi , vendredi , samedi , dimanche]
```



L1 a pour tête ... *à compléter* ... et est suivie de la liste L2 = ... *à compléter*...
L2 a pour tête ... *à compléter* ... et est suivie de la liste L3 = ... *à compléter* ...
Donner la dernière étape de cette décomposition:

Appeler éventuellement le professeur pour vérification

Utilisations des listes chaînées

- piles
- allocation mémoire sur un disque dur : les blocs libres sont stockés dans une liste chaînée
- opérations arithmétiques sur des grands entiers
- page suivante/précédente sur un navigateur : on utilise une liste doublement chaînée
- idem pour un logiciel de visualisation d'images, ou d'écoute de musique

Interface d'une liste

Une défintion simple du type `Liste` utilise les primitives suivantes (ce sont entre autre les méthodes de la classe, mais pas forcément) :

- construction d'une liste vide : `creerListe()`
- test de vacuité d'une liste : `estVide(liste)`
- Ajouter un élément en tête de la liste : `cons(élément, liste)` . C'est en fait le constructeur "historique" du type `liste` .
- Renvoyer le premier élément de la liste sans le supprimer : `donnee(liste)` . Renvoie la "tête" de liste
- Renvoyer la liste suivante, éventuellement vide, obtenue à partir de la liste initiale en supprimant son premier élément. : `suivant(liste)` . Renvoie la "queue" de la liste.

Une liste `L` peut s'écrire `L = cons(donnee(L) , suivant(L))` . On remarque que cette définition est récursive ; `suivant(L)` pouvant être une liste, de la même manière que les descendants d'un noeud dans un arbre peuvent être des arbres.

Exercices (sur papier ou à compléter dans la cellule) :

On utilisera uniquement les fonctions primitives définies ci-dessus

1. Créer la liste de vos quatre films préférés. les films doivent être dans l'ordre de vos préférences. Essayez d'écrire une seule ligne.
2. On donne une liste L1. Ecrire un algorithme en langage naturel renvoyant la liste L2, qui est dans l'ordre inverse de L1.

Une première implémentation

En terminale NSI, on travaillera essentiellement sur les listes chaînées. Les listes chainées sont composées de maillons, et sont définies récursivement. Un maillon est composé d'un élément de tête, et... d'un autre maillon, qui contient les éléments suivants, éléments de

queue. En fin de liste le maillon suivant est `None`. Les maillons sont aussi appelés cellules, notamment en anglais (`cell`).

On utilisera les conventions suivantes (qui ne sont pas universelles, d'autres versions peuvent exister) :

- la liste vide est le maillon de tête `None` et de queue `None` ;
- une liste de longueur 1 est composée d'un unique maillon de tête différente de `None`, et de queue `None` ;
- il est impossible d'avoir un maillon de tête `None` et de queue différente de `None`.

Les attributs de la classe `Cellule` sont :

- `tete` : l'élément de tête de la liste (éventuellement `None`).
- `queue` : la liste composant la deuxième partie de `Cell` (éventuellement `None`)

Correspondance entre liste et arbre filiforme

- `maillon` ou `cellule` = `arbreU` (comme arbre unaire, avec donc un unique sous-arbre)
- `tete` = `noeud` ; la valeur du noeud dans un arbre filiforme
- `queue` = `sous-arbre` ; ici il n'y a pas `gauche` ni `droite` puisqu'il n'y a qu'un sous-arbre

Implémentation des listes

Compléter le code en rajoutant les primitives `longueurListe` qui, comme son nom l'indique, renvoie la longueur de la liste ; et `listeElements`, qui, comme son nom l'indique moins clairement, renvoie un tableau dynamique Python (type `list`) comportant les éléments de la liste, dans l'ordre où la tête de la liste a pour indice 0 dans le tableau dynamique.

Entrée [ ]:

```
class Cellule :
    def __init__(self, tete = None, queue = None) :
        # Admirez le  joli booléen dans l'assertion
        assert (queue is None) or (tete is not None), 'constructi
        # On peut aussi rajouter une assertion pour vérifier que
        # est soit un maillon, soit None
        self.tete = tete
        self.queue = queue

    def est_vide(self):
        return self.tete is None # and self.queue is None est inu

    def donnee(self):
        # Méthode classique, mais qui n'apporte rien de plus que
        assert not(self.est_vide()) , 'Listevide'
        return self.tete

    def suivant(self):
        # Méthode classique, mais qui n'apporte rien de plus que
        assert not(self.est_vide()) , 'Listevide'
        return self.queue

    def longueur_liste_rec(self):
        # Algortihme à coder :
        # si la liste est vide on renvoie 0, sinon on renvoie 1 +

        return

    def longueur_liste_iter(self):
        # Algorithme :
        # Après initialisation de la longueur à 0, tant qu'il res
        # ajoute 1 et on remplace self par sa queue
        long = 0
        return long

    def liste_elements_rec(self) :
        # On reprend l'algorithme récursif du calcul de la longue
        # en cas de liste vide on renvoie [], sinon on renvoie la
        # tête à laquelle on ajoute la liste des éléments de la q
        # Remarque : les listes s'aditionnent [1] + [2, 3] = [1,
        # Remarque : on peut aussi programmer avec la méthode app
        return

    def liste_elements_iter(self) :
        # Algorithme : on se base sur le calcul de la longueur en
        return

    def appartient_iter(self, element):
        return

    def appartient_rec(self, element) :
        return

    def suppression_elt_iter(self, element) :
        if not self.appartient_iter(element) :
```

```
            # On écrit une fonction préliminaire qui teste si l'é
            return False
        else :
            # Il faut garder en mémoire le liens  de la cellule p
            # pour pouvoir "couper et recoller" ce lien vers la c
            # On passe de :
            #               précédente -> courante -> suivante
            # à :
            #               précédente -> suivante
            return True

    def suppression_elt_rec(self, element) :
    # On écrit une fonction préliminaire qui teste si l'élément e
        if not self.appartient_rec(self, element) :
            return False
        else :
            self.suppression_elt_rec_2(self, element)
            return True

    def suppression_elt_rec_2(self, element) :
        # partie récursive proprement dite
        # Reprendre la méthode itérative
        return None

    def __repr__(self):
        if self.tete is None :
            return '()'
        else:
            # la 1ère possibilité met l'aspect récursif en avant
            # la 2ème possibilité mes l'aspect chaîné en avant
            return '(' + str(self.donnee()) + repr(self.suivant()
            # return  str(self.donnee) + '->' + repr(self.suivant

def cons(tete, queue) :
    # Ici la primitive n'est pas une méthode mais une fonction. D
    # peut constater la totale inutilité, puisqu'elle tient sur u
    return Cellule(tete, queue)



nil = Cellule(None, None)       # notation "nil" historique
print("liste nil : ",print(nil)," de longueur : ",nil.longueur_li

print("la liste nil a pour tête ", nil.tete , " et pour queue ",n

liste = Cellule(4,nil)
for i in range(3,-1,-1):
    liste = Cellule(i,liste)
print("liste :" , liste," de tête ",liste.donnee()," et de queue
# Donner l'instruction pour trouver 2, l'écrire à la place de Non
print("Pour trouver 2, on a utilisé l'instruction ... " , None)
print("la longueur de la liste est : ",liste.longueur_liste_rec()
"""
print("Conversion en tableau dynamique (itératif) :", liste.liste
print("Conversion en tableau dynamique :(récursif)", liste.liste_
print ("3 est dans ",liste," : ",liste.appartient_iter(3), liste.
```

```
print ("7 est dans ",liste," : ",liste.appartient_iter(7), liste.
"""

"""
# Suppression d'un élément : penser à tester les cas "limite"
liste.supprimer_elt_rec(3)
liste.supprimer_elt_rec(4)
liste.supprimer_elt_rec(0)
print(liste)

liste = Cellule(4,nil)
for i in range(3,-1,-1):
    liste = Cellule(i,liste)

liste.supprimer_elt_iter(3)
liste.supprimer_elt_iter(4)
liste.supprimer_elt_iter(0)
print(liste)
"""

"""
# Egalité de listes
liste = Cellule(4,nil)
for i in range(3,-1,-1):
    liste = Cellule(i,liste)
print(listeb , "est égale à ",liste," : ", liste.est_egale(listeb
listeb = cons(4,listeb)
print(listeb , "est égale à ",liste," : ", liste.est_egale(listeb
"""
print()
```

Exercices

1. Donner la complexité dans le pire des cas des méthodes `est_vide()` , `longueur()` , `liste_elements()` , aussi bien pour les méthodes récursives qu'itératives.
2. Ecrire une méthode `appartient(element)` qui renvoie `None` si l'élément n'appartient pas à la liste, et son indice sinon. Complément pour ceux qui vont vite : en déduire une méthode `suppr_element(element)` qui supprime la première occurence d'un élément donné dans une liste.

Remarque : Quand on passe de l'itératif au récursif pour l'implémentation des méthodes `longueur_liste()` et `liste_elements()` , cela change-t-il l'usage de la classe ?

Réponse :

Questions complémentaires éventuelles

3. Ecrire une méthode `est_egale(liste2)` qui teste si la liste2 est égale à la liste appelant la méthode.
4. Ecrire une méthode qui inverse la liste. En version un peu plus facile, on peut se contenter de renvoyer la liste inversée, plutôt que de modifier l'objet.
5. Ecrire une méthode `derniere(liste)` qui renvoie la dernière cellule de la liste. En déduire une méthode `concatener(liste2)` qui concatène la liste2 en fin de la liste

appelant la méthode.

Des primitives différentes pour le type liste

Le type `Liste` n'est pas fixé dans le marbre. On peut proposer des primitives plus nombreuses et plus riches.

On propose ici les fonctions primitives sur les listes suivantes :

- construction d'une liste. La liste peut être vide, ou bien on peut la construire à partir d'un élément de tête et d'une autre liste. On appelle cette fonction : `creerListe(e = Aucun , liste = Aucun)`
- test de vacuité d'une liste : `est_vide(liste)`
- Obtention de la longueur de la liste : `longueur(liste)`
- Accéder au k-*ième* élément de la liste : `lire(liste , k)`
- Supprimer le k-*ième* élément de la liste : `supprimer(liste , k)` . Cette méthode renvoie une nouvelle liste.
- Insérer un élément en k-*ième* position dans la liste : `inserer(liste , k)` . Cette méthode renvoie une nouvelle liste.
- *Les trois primitives précédentes seront implémentées sur une méthode* `get_maillon(liste , k)` Cette méthode est donnée ci-dessous en itératif. C'est un bon exercice que de la programmer également en récursif, et de voir que le fonctionnement des primitives n'en change pas pour autant

Exercice : programmer les méthodes `lire` et `insérer` , dans l'implémentation suivante du type `liste` .
Et pour ceux qui vont vite : programmer `supprimer` ,

Entrée [ ]:

```
class Cellule :
    def __init__(self, tete = None, queue = None) :
        # Admirez le  joli booléen dans l'assertion
        assert (queue is None) or (tete is not None), 'constructio
        # On peut aussi rajouter une assertion pour vérifier que q
        # est soit un maillon, soit None
        self.tete = tete
        self.queue = queue

    def estVide(self):
        return self.tete is None # and self.queue is None est inut

    def donnee(self):
        assert not(self.estVide()) , 'Listevide'
        return self.tete

    def suivant(self):
        assert not(self.estVide()) , 'Listevide'
        return self.queue


    def longueur_liste(self):
        long = 0
        while not self.estVide():
            long = long + 1
            self = self.suivant()
        return long


    def get_maillon(self, i):
        # Version itérative
        if i >= self.longueur_liste() :
            raise IndexError('Index trop grand')
        else :
            while i > 0:
                i = i - 1
                self = self.suivant()
            return self

    def get_maillon_rec(self, i):

            return

    def lire(self , i) :
            return

    def inserer(self , i, element) :
        return

    def supprimer(self , i) :
        return

    def __repr__(self):
        if self.tete is None :
            return '()'
```

```
        else:
            # la 1ère possibilité met l'aspect récursif en avant
            # la 2ème possibilité mes l'aspect chaîné en avant
            return '(' + str(self.donnee()) + repr(self.suivant())
            # return  str(self.donnee) + '->' + repr(self.suivant)

maliste = Cellule()
print(maliste, maliste.estVide(), maliste.longueur_liste())
for i in range(5 , -1 , -1) :
    maliste = Cellule("film"+str(i),maliste)
    print("affichage : ",maliste, "de longueur ",maliste.longueur_

# TESTS nombreux et multiples ! Il en manque d'importants d'ailleu
print("lecture des éléments d'indices 1 et 5 :",maliste.lire(1),ma
print()
print("Insertion et suppression en itératif")
i = 5
print("get maillon d'indice ",i,)
print(maliste.get_maillon(i))
maliste = maliste.inserer(i , "film3b")
print("affichage insertion : ",maliste, "de longueur ",maliste.lon
maliste = maliste.supprimer(i)
print("affichage suppression : ",maliste, "de longueur ",maliste.l
maliste = maliste.supprimer(0)
print("affichage suppression indice 0: ",maliste, "de longueur ",m
maliste = maliste.supprimer(maliste.longueurListe() - 1)
print("affichage suppression dernier indice : ",maliste, "de longu
liste_quasi_vide = Cellule("rien", Cellule())
print(liste_quasi_vide, "longueur : ", liste_quasi_vide.longueur_l
liste_quasi_vide = liste_quasi_vide.supprimer(0)
print("affichage suppression indice 0: ",liste_quasi_vide, "de lon
```

Une troisième implémentation pour le type Liste

On donne ci-dessous une implémentation à base de tableaux dynamiques (le fameux type `list` de Python). La tête de la liste sera l'élément d'indice 0, la queue toute la suite.
Comme vous pouvez le constater ci-dessous, on ne construit pas de classe : les primitives sont traduites en fonctions.

Entrée [ ]:

```
def liste_vide() :
    return []

def est_vide(liste) :
    return liste == liste_vide()

def cons(element, liste) :
    liste.insert(element, 0)
    return liste

def tete(liste) :
    assert not(est_vide(liste)), 'liste vide'
    return liste[0]

def queue(liste):
    assert not(est_vide(liste)), 'liste vide'
    return liste[1:]

def supprimer(liste, i) :
    liste.pop(i)
    #return liste : pas indispensable car il ya effet de bord : la

def inserer(liste , i, element) :
    liste.insert(element, i)
    #return liste : idem ci dessus

ma_liste_2 = [0]
print(ma_liste_2, "est vide : ", est_vide(ma_liste_2))
print("tete : ",tete(ma_liste_2), "queue : ", queue(ma_liste_2))
ma_liste_2 = [0, 1, 2, 3, 4, 5]
print(ma_liste_2, "est vide : ", est_vide(ma_liste_2))
print("tete : ",tete(ma_liste_2), "queue : ", queue(ma_liste_2))
inserer(ma_liste_2, 33, 4)
supprimer(ma_liste_2, 5)
print(ma_liste_2)
```

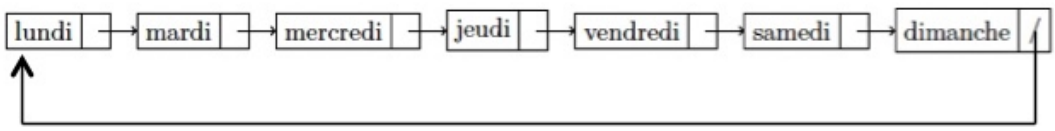
Autres versions des listes

On peut créer d'autres versions des listes:

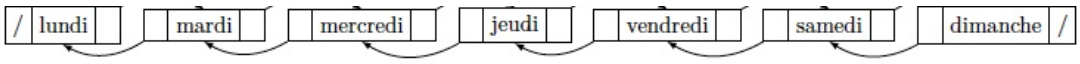
- Listes basées sur des tableaux. On perd l'intérêt des listes, qui est d'insérer facilement un élément

0	1	2	3	4	5	6
lundi	mardi	mercredi	jeudi	vendredi	samedi	dimanche

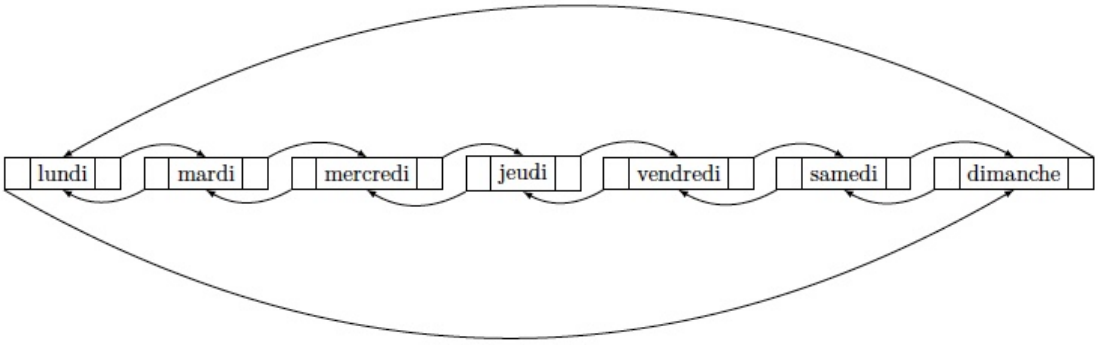
- Listes circulaires. Permet de boucler en fin de liste sur le premier élément



- Listes doublement chaînées. Permet de connaître non seulement l'élément suivant dans la liste, mais aussi le précédent



- Listes doublement chaînées circulaires



Rappel sur une remarque importante : le type abstrait `Liste` n'est pas le type `list` de Python. Les listes de Python sont basées sur des tableaux, et mélangent des accès de type fonctions (`del(ma_liste[3])`), des accès de type objet (`ma_liste.append('truc')`), et des accès plus étranges (`machin in ma_liste` , `ma_liste[3:6]`)

Un exercice complémentaire

Problème de Flavius Josèphe (https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_Jos%C3%A8phe). Flavius Josèphe est un historiographe romain juif du 1er siècle, dont l'oeuvre historique est sujette à caution. Il a donné la première version du problème suivant : "41 soldats juifs, cernés par des soldats romains, décident de former un cercle. Un premier soldat est choisi au hasard et est exécuté, le troisième à partir de sa gauche (ou droite) est ensuite exécuté. Tant qu'il y a des soldats, la sélection continue. Le but est de trouver à quel endroit doit se tenir un soldat pour être le dernier. Josèphe, peu enthousiaste à l'idée de mourir, parvint à trouver l'endroit où se tenir. Quel est-il ?"
Variante : 42 soldats juifs, deux survivants, et les romains en tuent un sur trois.

Exercice :

Résoudre le problème de Josèphe avec une liste chaînée "normale".
Il y a deux sous-problèmes :

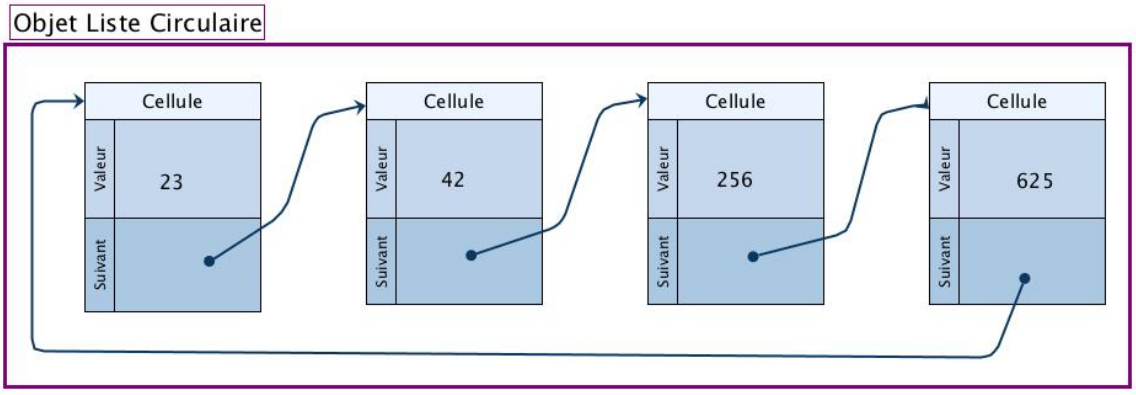
- la création de la liste des soldats
- la recherche du ou des survivants

Entrée [ ]:

```
def josephe(n, k, s):
    """
    Résoud le problème de Josèphe. Les soldats sont numérotés de 1
    @param n : entier >= 1, nombre initial de soldats
    @param k : entier >= 2, saut entre deux meurtres de soldats
    @param s: entier >=0, nombre de soldats survivants
    @return survivor : liste d'entiers, numéros des soplдats survi
    """
    # Création de la liste des soldats : on suppose qu'initialemen
    survivor = CelluleL()
    #for i in range(...) :
        # ajouter les soldats

    #Recherche du survivant en supprimant petit à petit les noeuds
    # while ... :
        # tuer les pauvres soldats et se déplacer dans la liste

    return survivor

#tests à compléter (ça ne risque pas de fonctionner avec les "?")
print("un soldat sur deux")
for i in range(1, 42):
    print("Survivant pour ", i, "soldats :",josephe(i,2,1).???)
print()
tset = josephe(41, 3, 2)
print("2 survivants pour 41 soldats, avec 1 sur 3  :", tset.????,
tset = josephe(1234,7, 10)
print("10 survivants pour 1234 soldats, avec 1 sur 7  :", tset)
```

La liste circulaire

Une liste chaînée circulaire est une liste chaînée dans laquelle le dernier élément n'est pas la liste vide, mais le premier élément de la liste. les listes chaînées circulaires sont notamment utilisées pour représenter des files. Cette structure de données est particulièrement adaptée à la résolution du problème de Josèphe. Mais elle est aussi utilisée par exemple pour gérer le partage du processeur (CPU) entre différents programmes (différents processus).



Une implantation de cette structure est proposée ci-dessous. Elle possède deux classes, `Noeud` et `ListeCirc`. La classe `Noeud` est celle de la liste chaînée non circulaire,

l'attribut `longueur` en moins.
Les attributs de `Noeud` sont:

- `donnée` : le contenu du noeud
- `suivant` : le noeud suivant

La classe `ListeCirc` comporte au moins deux noeuds en général (mais pas à la création, cf. ci-après) : tête et queue.
Plus précisément, les attributs à la création sont :

- `tête` : `Noeud(None par défaut, ou données de la tête)`
- `queue` : égale à `tête` lors de la création de la liste circulaire
- `tête.suivant : queue` . Le noeud suivant la tête est la queue
- `queue.suivant : tête` . Le noeud suivant la queue est la tête On peut remarquer :
- qu'une liste circulaire vide comporte un seul noeud de donnée `None` , qui point sur lui-même
- qu'une liste circulaire ne comportant qu'une seule donnée comporte également un seul noeud
- que dans ces deux cas le noeud pointe sur lui-même : `tête = queue` .

Remarques / questions :

- On aurait pu proposer une implémentation sans objet `ListeCirc` , et de même on aurait pu proposer un objet `ListeChainee` , qui aurait contenu les cellules de la liste chaînée non circulaire. On voit que les possibilités d'implémentations sont multiples.
- Utiliser les mêmes noms de primitives permet d'écrire des programmes fonctionnant de manière identique avec les deux structures de données. Ce qui peut être très pratique.
- La liste vide est composée d'une seule cellule, de donnée `None` , pointant sur elle-même. Lors du calcul de la longueur, de l'insertion ou de la suppression d'un élément, on est obligé de différencier ce cas. Le code est plus complexe que pour la liste chaînée non circulaire.
- Pourquoi utilise-t-on ici deux classes, Noeud et ListeCirc ?
- Pourquoi ne reprend-on pas directement le calcul de la longueur comme dans le cas de la liste chaînée ?

Exercice :

Compléter le code de la classe `ListeCirc` .
Résoudre le problème de Josèphe avec une liste chaînée circulaire.
Comme précédemment il y a deux sous-problèmes :

- la création de la liste des soldats
- la recherche du ou des survivants

Entrée [ ]:

```
class Noeud:
    def __init__(self, donnee, suivant = None):
        self.donnee = donnee
        self.suivant = suivant

    def __repr__(self):
        if self.donnee == None:
            return ""
        else:
            return str(self.donnee)

class ListeCirc:
    def __init__(self, donnee_tete = None):
        self.tete = Noeud(donnee_tete)
        self.queue = self.tete
        self.tete.suivant = self.queue
        self.queue.suivant = self.tete

    def est_vide(self):
        return self.tete.donnee is None

    def longueur(self):
        # A programmer
        lg = None
        return lg

    def supprimer_courant(self, precedent, courant):
        # Suppression du noeud courant connaissant le précédent
        if self.tete == self.queue :    # cas particulier : un seu
            self.tete.donnee = None
        elif courant == self.tete :    # cas particulier : suppres
            self.tete = self.tete.suivant
            self.queue.suivant = self.tete
        elif courant == self.queue :   # cas particulier : suppres
            self.queue = precedent
            self.queue.suivant = self.tete
        else:                          # cas général
            precedent.suivant = courant.suivant

    def ajout_fin(self, donnee):
        # Ajoute un noeud en fin de la liste circulaire
        if self.tete.donnee is None:     # on remplit d'abord la t
            self.tete.donnee = donnee
        else:                            # sinon on crée un nouvea
            nouveauNoeud = Noeud(donnee)
            self.queue.suivant = nouveauNoeud # On ajoute le noeud
            self.queue = nouveauNoeud         # il devient la nouv
            self.queue.suivant = self.tete    # et pointe sur la t

    def __repr__(self):
        if self.tete.donnee is None :
            return 'Liste vide'
        else:
            chaine = str(self.tete.donnee) + "->"
            courant = self.tete
```

```
            while courant.suivant != self.tete and courant.suivant
                courant = courant.suivant
                chaine = chaine + str(courant.donnee) + "->"
            chaine = chaine + "tête"
            return  chaine

```

Entrée [ ]:

```
def josephe(n, k, s):
    """
    Résoud le problème de Josèphe. Les soldats sont numérotés de 1
    @param n : entier >= 1, nombre initial de soldats
    @param k : entier >= 2, saut entre deux meurtres de soldats
    @param s: entier >=0, nombre de soldats survivants
    @return survivor : liste d'entiers, numéros des soplдats survi
    """
    survivor = CelluleL()

    return survivor

#tests à compléter (ça ne risque pas de fonctionner avec les "?")
print("un soldat sur deux")
for i in range(1, 42):
    print("Survivant pour ", i, "soldats :",josephe(i,2,1).???)
print()
tset = josephe(41, 3, 2)
print("2 survivants pour 41 soldats, avec 1 sur 3  :", tset.????,
tset = josephe(1234,7, 10)
print("10 survivants pour 1234 soldats, avec 1 sur 7  :", tset)
```

Sources :

- cours de Clémentine Nebut, Université de Montpellier II
- Wikipedia
- Types de Données et Algorithmes, C. Froidevaux, MC Gaudel, M Soria
- Eléments d'Algorithmique, D. Beauquier, J. Berstel, Ph. Chrétienne
- Document d'accompagnement éduscol Terminale NSI

---

frederic.mandon @ ac-montpellier.fr, Lycée Jean Jaurès - Saint Clément de Rivière - France (2015-2019)