DIU Bloc 4, séance du vendredi 12/06

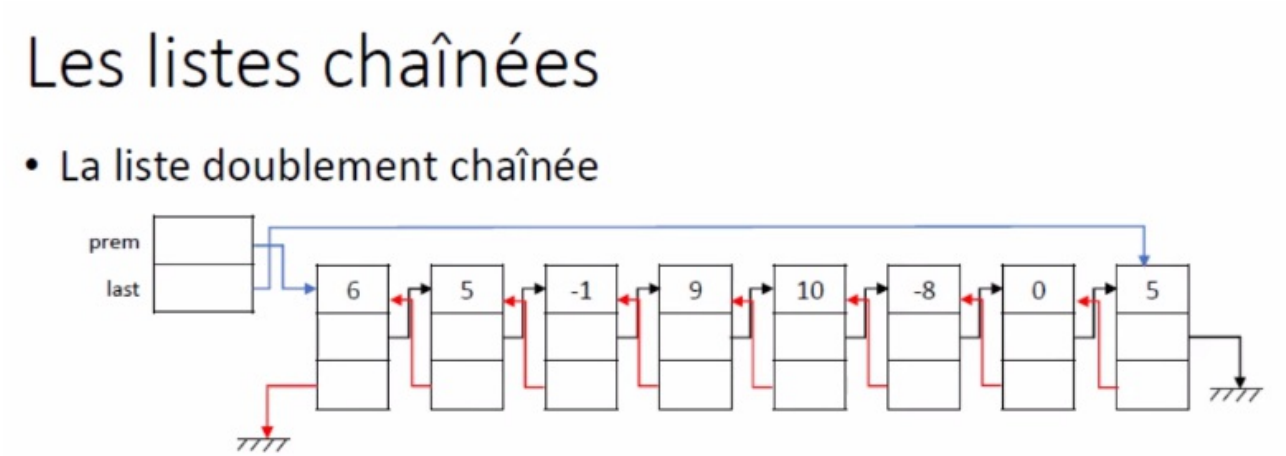
Complexité des structures de données

https://www.bigocheatsheet.com/ (https://www.bigocheatsheet.com/)

{%pdf https://www.bigocheatsheet.com/pdf/big-o-cheatsheet.pdf %}

Liste doublement chainée non circulaire

Construction de la classe




```
# ======================    Cellule
=========================================
class Cellule:
    def __init__(self, info, suivant, precedent):
        self.info = info
        self.suivant = suivant
        self.precedent = precedent


# ======================    Liste
=============================================
class Liste:
    """ Liste doublement chaînée non circulaire"""

    def __init__(self):
        self.premier = None
        self.dernier = None

    """Commentaires
    le fait d'avoir un pointeur vers le premier ou le dernier
    n'est pas lié au caractèr simplement chainé ou doublement
    chainé
```

```
On décide qu'une liste vide est représentée par
self.premier et self.dernier valent None : choix
"""


def __del__(self):
    self.vider()

def estVide(self):
    #on aurait pu rajouter que self.dernier == None
    #mais on n'est pas obligé de le faire
    #peut-on utiliser len ?
    #OK mais il faut surcharger l'opérateur len
    #dans la définition de la classe
    return self.premier == None

def vider(self):
    while self.premier:
        elt = self.premier
        self.premier = elt.suivant
        del elt
        #on supprimer manuellement l'élément
        #car le garbage collector ne pourrait pas supprimer
        #les cellules en doublement chainé
        #car on n'aurait pas supprimé tous les liens
        ##rouge et noir entre les cellules (voir image)
        #on peut utiliser supprimer_tete() dans vider
        """
        while self.premier:
            self.supprimer_tete()
        #pas besoin de self.dernier = None géré dans supprimer_tete
        """
    self.dernier = None



def nbElements(self):
    nb = 0
    elt = self.premier
    while elt:
        nb += 1
        elt = elt.suivant
    return nb

def nbElements2(self):
    return 1 + _nbE(self, self.suivant)

def _nbE(self, c):
    if c is None:
        return 0
    return _nbE(self, c.suivant)
```

```
    def iemeElement(self, indice):
        elt = self.premier
        for i in range(indice):
            elt = elt.suivant
            if elt == None:
                print("Erreur iemeElement : indice trop grand")
        return elt.info

    def modifierIemeElement(self, indice, infoElement):
        elt = self.premier
        for i in range(indice):
            elt = elt.suivant
            if elt == None:
                print("Erreur modifierIemeElement : indice trop grand")
        elt.info = infoElement

    def afficher(self):
        elt = self.premier
        while elt:
            print(elt.info, end=' ')
            elt = elt.suivant
        print()

    def ajouterEnTete(self, infoElement):
        elt = Cellule(infoElement, self.premier, None)
        if self.estVide():
            self.dernier = elt
        else:
            self.premier.precedent = elt
        self.premier = elt

    def supprimerTete(self):
        #ici le garbage collector supprimera bien l'élément
        #car on a bien supprimé tous les liens vers l'objet
        elt = self.premier
        self.premier = elt.suivant
        if self.premier:
            self.premier.precedent = None
        else:
            self.Dernier = None

    def ajouterEnQueue(self, infoElement):
        if self.estVide():
            self.ajouterEnTete(infoElement)
        else:
            elt = Cellule(infoElement, None, self.dernier)
            self.dernier.suivant = elt
            self.dernier = elt

    def rechercherElement(self, infoElement):
        #pourquoi utiliser un booléen
```

```
        #alors qu'on pourrait faire une sortie prématurée
        elt = self.premier
        trouve = False
        pos = 0
        while elt and not trouve:
            if infoElement == elt.info:
                trouve = True
            else:
                elt = elt.suivant
                pos += 1
        if trouve:
            return pos
        else:
            return -1

    def insererElement(self, indice, infoElement):
        #on aime bien réutiliser des fonctions déjà érites
        if self.estVide() or indice == 0:
            self.ajouterEnTete(infoElement)
        elif indice == self.nbElements():
            self.ajouterEnQueue(infoElement)
        else:
            #self.nbElements() en O(n)
            #quand on arrive ici on est donc en O(2n)
            #on aurait pu choisir d'utiliser un attibut self.length
            elt = self.premier
            for _ in range(indice): elt = elt.suivant
            nvelt = Cellule(infoElement, elt, elt.precedent)
            elt.precedent.suivant = nvelt
            elt.precedent = nvelt

    def importerTableau(self, tab):
        self.vider()
        for elt in tab:
            self.ajouterEnQueue(elt)

    def trier(self):
        if not self.premier or not self.premier.suivant:
            return # rien a faire si moins de 2 elements
        derniereTriee = self.premier
        aplacer = self.premier.suivant
        while aplacer: # boucle principale sur les elements a placer
            c = self.premier
            while c != aplacer and c.info < aplacer.info:
                c = c.suivant
            # En sortie du while , on sait que:
            # (1) soit c->info est superieur a aplacer->info : on doit inserer
aplacer entre le precedent de c et c.
            # (2) soit c == aplacer : aplacer est a sa place
            precC = c.precedent
            precAPlacer = aplacer.precedent
            suivAPlacer = aplacer.suivant
```

```
            # Cas 1 :
            if c != aplacer:
                if precC :
                    precC.suivant = aplacer
                else:
                    self.premier = aplacer
                if suivAPlacer == None:
                    self.dernier = precAPlacer
                else:
                    suivAPlacer.precedent = precAPlacer
                # desinsertion aplacer
                precAPlacer.suivant = suivAPlacer
                # insertion [preC,C]
                aplacer.suivant = c
                aplacer.precedent = precC
                c.precedent = aplacer
                # iteration
                derniereTriee.suivant = suivAPlacer

            # Cas 2 :
            else :
                dernieTriee = aplacer

            aplacer = dernieTriee.suivant
```

Remarques diverses

- Différences entre `__str__` et `__repr__` :
    - `__repr__` est appelé lorsqu'on saisit le nom de l'objet dans la console => voir aussi https://docs.python.org/3/library/functions.html#repr (https://docs.python.org/3/library/functions.html#repr) ou https://docs.python.org/3/library/functions.html#repr (https://docs.python.org/3/library/functions.html#repr)

Importer un tableau dans la liste chaînée

Ici il s'agit d'une méthode d'instance

```
def importerTableau(self, tab):
        #on a un accès direct en fin de liste
        #donc ajouterEnQueue en cout constant
        #donc importerTableau en O(n)
        self.vider()
        for elt in tab:
            self.ajouterEnQueue(elt)
```

Remarque : vider est en O(n) (suppression de la tête tant que pas vide) et insérer un élément en queue de liste est en O(1)
(grâce à l'attribut dernier) ce qui est fait n fois donc importer un tableau ainsi est en O(n). Si on avait une liste simplement
chaînée, ajouter en queue serait en O(n) et donc on aurait eu une complexité en $O(n^2)$. Il aurait alors mieux valu parcourir
le tableau dans le sens inverse en faisant des appels à ajouterEnTete (qui est en O(1) grâce à l'attribut premier)

Tri par insertion

```
 def trier(self):
        if not self.premier or not self.premier.suivant:
            return # rien a faire si moins de 2 elements
        derniereTriee = self.premier
        aplacer = self.premier.suivant
        while aplacer: # boucle principale sur les elements a placer
            c = self.premier
            while c != aplacer and c.info < aplacer.info:
                c = c.suivant
            # En sortie du while , on sait que:
            # (1) soit c->info est superieur a aplacer->info : on doit inserer
aplacer entre le precedent de c et c.
            # (2) soit c == aplacer : aplacer est a sa place
            precC = c.precedent
            precAPlacer = aplacer.precedent
            suivAPlacer = aplacer.suivant

            # Cas 1 :
            if c != aplacer:
                if precC :
                    precC.suivant = aplacer
                else:
                    self.premier = aplacer
                if suivAPlacer == None:
                    self.dernier = precAPlacer
                else:
                    suivAPlacer.precedent = precAPlacer
                # desinsertion aplacer
                precAPlacer.suivant = suivAPlacer
                # insertion [preC,C]
                aplacer.suivant = c
                aplacer.precedent = precC
                c.precedent = aplacer
                # iteration
                derniereTriee.suivant = suivAPlacer

            # Cas 2 :
            else :
                derniereTriee = aplacer

            aplacer = derniereTriee.suivant
```

++Remarques :++ :-1:

Si on déplace juste les infos et pas les cellules, cela peut poser problème de changer les valeurs des cellues, si d'autres objets du programme font référence aux cellules de la liste chainée
Le tri par insertion avec une liste chainée est plus performant que le tri par insertion sur un tableau dynamique en Python : même complexité mais en Python il faut parfois réallouer de l'espace en mémoire pour le tableau (en doublant sa taille)
Méthodes d'instance, de classe, statique(on ne sera pas amené à l'utiliser avec les élèves):

- https://realpython.com/instance-class-and-static-methods-demystified/] (https://realpython.com/instance-class-and-static-methods-demystified/) (https://realpython.com/instance-class-and-static-methods-demystified/%5D(https://realpython.com/instance-class-and-static-methods-demystified/))
- https://www.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/c9b29d750f974f7ea06fdf971a40a7f4/9cad56d60cde49f (https://www.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/c9b29d750f974f7ea06fdf971a40a7f4/9cad56d60cde49f
- https://docs.python.org/3/library/functions.html#classmethod (https://docs.python.org/3/library/functions.html#classmethod)

Les identifiants des items dans Tkinter :-1:
Stocké dans un attribut "tag" de l'objet si je me souviens bien. Il faudrait se replonger dans la doc :
https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/tkinter.pdf

```
class NombreComplexe :
    nbComplexesCrees = 0

    def __init__ (self) :
        self.Re = 0
        self.Im = 0
        NombreComplexe.nbComplexeCrees += 1

    def afficheNbComplexe (cls) :
        print(NombreComplexe.nbComplexeCrees)

    afficheNbComplexe = classmethod(afficheNbComplexe)

NombreComplexe.afficheNbComplexe()
# affiche 0
nc1 = NombreComplexe()
NombreComplexe.afficheNbComplexe()
# affiche 1
nc2 = NombreComplexe()
NombreComplexe.afficheNbComplexe()
# affiche 2
# on peut aussi faire : nc2.afficheNbComplexe()
```

Piles et Files

Pile

```
class Pile :
    def __init__(self, L):
        self.liste = L

    def __repr__(self):  # ou __str__
        s = 'pile: '
        for e in self.liste:
            s += str(e) + ' '
        return s

    def empiler(self, e):
        #pourquoi insérer en tête de liste Python ?
        #on peut choisir que le sommet soit le dernier
        #élément de la liste  (cela éviterait de décaler tout vers la droite
dans une insertion)
        self.liste.insert(0, e)

    def depiler(self):
        del self.liste[0]

    def sommet(self):
        return self.liste[0]

    def estVide(self):
        return len(self.liste) == 0

    def hauteur(self):
        return len(self.liste)

    def traiter(self):
        e = self.liste[0]
        del self.liste[0]
        return e
```

++Remarques:++

> La complexité des opérations `empiler`, `depiler`, `sommet` dépend du choix d'implémentation :-1:
> * coût amorti constant avec une structure de liste chainée
> * coût amorti constant avec une liste Python si on choisit que le sommet de Pile est en fin de liste

Le coût de len sur une liste Python est en $O(1)$, car les listes Python ont un attribut longueur. A propos de l'implémentation des listes en Python https://realpython.com/instance-class-and-static-methods-demystified/ (https://realpython.com/instance-class-and-static-methods-demystified/)

Files

```
class File :
    def __init__(self, L):
        self.liste = L

    def __repr__(self):  # ou __str__
        s = 'file: '
        for e in self.liste:
            s += str(e) + ' '
        return s

    def traiter(self):
        e = self.liste[-1]
        del self.liste[-1]
        return e

    def enfiler(self, e):
        self.liste.insert(0, e)

    def estVide(self):
        return len(self.liste) == 0

    def longueur(self):
        return len(self.liste)
```

Inversion d'une file en utilisant une pile

```
def inverserFile (file) :
    pile = Pile()
    while not file.estVide() :
        pile.empiler(file.traiter())
    while not pile.estVide() :
        file.enfiler(pile.traiter())
```

++Remarques++

Exemples de Pile et de File :-1:

```
* Piles :

    * Jeux de cartes avec pioche
    * Les undo : CTRL + Z = > je veux être capable de revenir en arrière =>
quelle structure de données utiliser => une pile
    * vailidité d'un parentésage
    * pour l'allocation sur le tas, on a un modèle de pile (empilement des
appels des fonctions et de la mémoire qu'elles uitilisent)
    * Passer de notation infixe à polonaise, c'est un cas d'usage d'une pile :
y'a une appli sur android : droid48
    * Jeux de société : Le Jungle Speed ne fonctionne qu'avec des Piles, tout
comme la plupart des manches du Dooble
    * parcours en profondeur d'un graphe (parcours d'un labyrinthe , résolution
d'un sudoku ... on peut faire des mini projets avec les piles
    *
<https://runestone.academy/runestone/books/published/pythonds/BasicDS/InfixPref
ixandPostfixExpressions.html>

* Files :
    * files d'attente : impression, événement
    * parcours en largeur
```

Validité du parenthésage d'une expression

```python
def valide (ch) :
    pile = Pile()
    i = 0
    for car in ch :
        if car == '(' or car == '[' :
            pile.empiler(car)
        elif car == ')' :
            if not pile.estVide() :
                sommet = pile.traiter()
            if sommet != '(' :
                return False
            else :
                return False
        elif car == ']' :
            if not pile.estVide() :
                sommet = pile.traiter()
            if sommet != '[' :
                return False
            else :
                return False
            return pile.estVide()
```

++Remarques++

> On n'a besoin de traiter dans la pile que les caractères de parenthésages.
> Attention aux cas où la liste est vide.

Bilan

++Avec les élèves++

Liste chaînée : trop dur Pile et File : accessible Intérêt de la POO : rassembler ensemble autour d'une même entité toutes les opérations de manipulation de cet objet