PILES ET FILES

1. Introduction

- Nous allons aborder deux structures de données abstraites linéaires : Les piles et les files .
- Ces structures permettent de stocker des éléments et fournissent des opérations permettant d'ajouter ou de retirer des éléments un à un.
- Chacune de ces structures a ses règles adaptées à de nombreuses situations en informatique.

2. Pile (LIFO)

- Dans une pile ("stack" en anglais), chaque opération de retrait retire l'élément arrivé le plus récemment.
- On associe cette structure à l'image d'une pile d'assiettes, dans laquelle chaque nouvelle assiette est ajoutée au-dessus des précédentes et où l'assiette retirée est systématiquement celle du sommet.
- Ce principe est nommé "dernier entré, premier sorti", plus couramment LIFO en anglais (Last In First Out).
- Exemples d'utilisation :
  - Bouton de retour en arrière (undo) qui permet d'annuler la(les) dernière(s) modification(s) effectuée(s).
  - Historique d'un navigateur web.
  - Pile des appels d'une fonction récursive.
  - Un processeur utilise une pile pour gérer les instructions à éxécuter.
  - Un algorithme de parcours en profondeur utilise une pile pour mémoriser les nœuds non encore visités.

Interface d'une pile

Voici les principales fonctions souhaitées pour cette interface :

1. Créer une pile, afficher son contenu.
2. Déterminer si une pile est vide.
3. Empiler (push en anglais) un élément, c'est à dire le placer au sommet de la pile.
4. Dépiler(pop en anglais) l'élément placé au sommet de la pile.

D'autres fonctionnalités qui facilitent la manipulation :

1. Lire la valeur située au sommet de la pile (sans la dépiler).
2. Connaître le nombre d'élément de la pile.
3. Vider une pile.

Représentations :

- Dans la pile ci-dessous, on considère que 3 est arrivé en dernier et se trouve au sommet de la pile.

```
|3|
|2|
|1|
|1|
```

Ainsi empiler 5 placera cette valeur au sommet de la pile et donnera la pile :

```
|5|
|3|
|2|
|1|
|1|
```

Implémentation d'une pile

- On utilise ici le paradigme de la programmation objet, mais ce n'est pas la seule façon de faire.
- L'implémentation se fait aisément à l'aide du type `list` de python, en particulier avec les méthodes suivantes :
  - La méthode `append()` qui ajoute un élément en fin de liste.
  - La méthode `pop()` qui supprime le dernier élément d'une liste, en le renvoyant.
  - la méthode `len()` qui renvoie la longueur d'une liste c'est à dire son nombre d'éléments.
- Quelques rappels :
  - L'indice `-1` permet d'accéder au dernier élément d'une liste.
  - `[]` est la liste vide.
- Le type `list` de Python est particulier. Nous verrons ultérieurement une autre façon de faire, qui utilise une structure de données adaptable à d'autres langages de programmation.

Exercice 1 :

Implémenter toutes les fonctionnalités de l'interface (numérotées plus haut) en complétant le code de la classe `Pile` .

In [9]:

```
class Pile:
    #1.
    def __init__(self):
        self.valeurs=[]

    def affiche(self):
        '''Affiche le contenu de la pile'''
        n=len(self.valeurs)
        if n==0:
            print('| |')
        else:
            for i in range(n):
                print('|'+ str(self.valeurs[n-1-i]) + '|')
        print('¯¯¯')


    #2.
    def est_vide(self):
        '''renvoie True si la pile est vide,
        False sinon'''


        pass

    #3.
    def empiler(self,a):
        '''Place l'élément a au sommet de la pile'''


        pass

    #4.
    def depiler(self):
        '''Supprime l'élément placé au sommet de la pile
        A condition qu'elle soit non vide.
        Renvoie l'élément supprimé.'''



        pass

    #5.
    def sommet(self):
        '''Renvoie la valeur du sommet de la pile
        si elle est n'est pas vide
        (sans la retirer)'''



        pass

    #6.
    def longueur(self):
        '''Renvoie le nombre d'élément dans la pile'''


        pass

    #7.
    def vider(self):
        '''Vide la pile'''


        pass
```

```
#Jeu de tests pour les piles ci-dessous
```

In [10]:
```
#1 init
p=Pile()
p.affiche() # la pile est vide par défaut
```

```
| |
```

In [11]:
```
#2 est_vide
p.est_vide() # renvoie True
```

Out[11]: True

In [12]:
```
#3 empiler
p.empiler(1)
p.empiler(1)
p.empiler(2)
p.empiler(3)
p.empiler(5)
p.affiche()  # 5 est au sommet de la pile
p.est_vide() # renvoie False
```

```
|5|
|3|
|2|
|1|
|1|
```

Out[12]: False

In [13]:
```
#4 depiler
p.depiler()
p.depiler()
p.affiche() #2 est au sommet de la pile
```

```
|2|
|1|
|1|
```

In [14]:
```
#5 sommet
p.empiler(5)
p.sommet() # le sommet est 5
```

Out[14]: 5

In [15]:
```
#6 longueur
p.affiche()
p.longueur()
```

```
|5|
|2|
|1|
|1|
```

Out[15]: 4

In [16]:
```
#7 vider
p.vider()
p.affiche()
print(p.est_vide()) # la pile est vide
```

```
| |
```

```
True
```

Exercice 2 :

- En utilisant les méthodes de l'implémentation précédente, compléter le code de la fonction `inverse(chaine)` qui prend en paramètre une chaîne de caractères et qui renvoie la chaîne de caractère inversée.
- Par exemple, `inverse('abcdef')` doit renvoyer `'fedcba'` .

In [13]:
```
def inverse(chaine):
    #on crée une pile vide
    p=...

    #On parcourt la chaîne et on empile les éléments dans p
    for lettre in chaine:
        ...

    #On crée une chaîne de caractère vide
    inv=...

    # Tant que p n'est pas vide, on dépile et on ajoute à la chaîne inversée.
    while p.est_vide()==False:
        inv=...

    return ...
```

In [14]:
```
#tests
assert inverse('abcdef')=='fedcba'
assert inverse('a')=='a'
assert inverse('')==''
```

3. File (FIFO)

- Dans une file ("queue" en anglais), chaque opération de retrait retire l'élément qui avait été ajouté en premier. On associe cette structure à l'image d'une file d'attente, dans laquelle les personnes arrivent à tour de rôle, patientent et sont servies dans leur ordre d'arrivée.
- Ce principe est appelé "premier entré, premier sorti", plus couramment en anglais FIFO (First In First Out).
- En programmation, les files sont utiles pour mettre en attente des informations dans l'ordre dans lequel elles sont arrivées.
- Exemple d'utilisation :
  - Les serveurs d'impression traitent les requêtes dans l'ordre dans lequel elles arrivent et les insèrent dans une file d'attente (dite aussi queue ou spool).
  - Un algorithme de parcours en largeur utilise une file pour mémoriser les nœuds non encore visités.
  - Gestion de mémoires tampons (en anglais « buffers »), par exemple lors de la lecture d'une vidéo.

Interface d'une file

Voici les principales fonctions souhaitées pour cette interface :

1. Créer une file, afficher son contenu.
2. Déterminer si une file est vide.
3. Enfiler (enqueue en anglais) un élément, c'est à dire le placer à la fin de la file.
4. Défiler(dequeue en anglais) l'élément placé au début de la file.

D'autres fonctionnalités qui facilitent la manipulation :

1. Lire la valeur située à l'avant de la file (sans la défiler).
2. Connaître le nombre d'élément de la file.
3. Vider une file.

Représentation :

Dans la file ci-dessous, on considère que 1 est arrivé en premier et se trouve en tête de file.

```
→ 3 | 2 | 1 | 1 →
```

Ainsi, enfiler 5 le placera en queue de file et donnera la file

```
→ 5 | 3 | 2 | 1 | 1 →
```

Implémentations d'une file

Directement avec le type `list` de python 😒

- On pourra ici utiliser la méthode :
  - `insert(i, e)` pour insérer l'élément `e` à l'index `i` .

Exercice 3 :

Implémenter les fonctionnalités de l'interface d'une file numérotées plus haut.

In [15]:
```
class File:
    #1.
    def __init__(self):
        self.valeurs=[]

    def affiche(self):
        '''Affiche le contenu de la file'''
        n=len(self.valeurs)
        print('→', end=' ')
        if n !=0:
            for i in range(n-1):
                print(str(self.valeurs[i]), end=' | ')
            print(str(self.valeurs[-1]), end=' ')
        print('→')


    #2.
    def est_vide(self):
        '''renvoie True si la file est vide,
        False sinon'''


        pass

    #3.
    def enfiler(self,a):
        '''Place l'élément a en queue de file'''


        pass

    #4.
    def defiler(self):
        '''Supprime l'élément placé en tête de  file
        A condition qu'elle soit non vide
        Renvoie l'élément supprimé.'''



        pass

    #5.
    def tete(self):
        '''Renvoie la valeur en début de file
        si elle n'est pas vide
        (sans la defiler)'''



        pass

    #6.
    def longueur(self):
        '''Renvoie le nombre d'élément dans la file'''


        pass

    #7.
    def vider(self):
        '''Vide la file'''


        pass
```

```
#Jeu de tests pour les files
```

In [16]:
```
#1 init
f=File()
f.affiche() #La file f est vide
```

```
→ →
```

In [17]:
```
#2 est_vide
f.est_vide() # renvoie True
```

Out[17]:
```
True
```

In [18]:
```
#3 enfiler
f.enfiler(1)
f.enfiler(1)
f.enfiler(2)
f.enfiler(3)
f.enfiler(5)

f.affiche() # 1 est en tête de file
```

```
→ 5 | 3 | 2 | 1 | 1 →
```

In [19]:
```
#4 defiler
f.defiler()
f.defiler()
f.defiler()
f.affiche() #3 est en tête de file
```

```
→ 5 | 3 →
```

In [20]:
```
#5 tete
f.tete() #renvoie 3
```

Out[20]:
```
3
```

In [21]:
```
#6 longueur
f.affiche()
f.longueur() #renvoie 2
```

```
→ 5 | 3 →
```

Out[21]:
```
2
```

In [23]:
```
#7 vider
f.vider()
f.affiche() #la file est vide
```

```
→ →
```

Remarque :

- Cette solution peut convenir pour des files de taille raisonnable. Mais lorsque le nombre d'éléments est très grand, le coût en temps de l'insertion d'un élément à l'index 0 est important car il faut décaler tous les autres éléments vers la droite.

Avec deux piles 😊

- Une file est dans ce cas caractérisée par deux piles. La première contient les éléments que l'on veut enfiler, la seconde contient les éléments que l'on veut défiler.

Exemple :

On veut créer la file `→ 3 | 2 | 1 →` , enfiler 4 puis défiler l'élément 1.

La file deviendra alors `→ 4 | 3 | 2 →` .

- On crée 2 piles, une pile d'entrée et une pile de sortie.
- Pour enfiler un élément, il suffit d'empiler les éléments dans leur ordre d'arrivée dans la pile d'entrée.
  - Ainsi , on empile les trois premiers éléments `1` , puis `2` puis `3` et ensuite le quatrième `4`
- Lorsque l'on veut défiler l'élément en tête de file(ici `1` , celui qui est arrivé en premier) :
  - On dépile les éléments de la pile d'entrée pour les empiler dans la pile de sortie.
  - Le dernier élément empilé dans la pile de sortie est bien `1`
  - On le dépile de la pile de sortie.

Exercice 4 :

A l'aide de la description précédente et des méthodes de la classe `Pile` , compléter le code de la classe `File` ci-dessous :

In [25]:

```
class File:
    #1
    def __init__(self):
        '''Initialise une file à l'aide de deux piles vides'''
        self.entree=...
        self.sortie=...

    def affiche(self):
        '''Affiche le contenu de la file'''
        n1=len(self.entree.valeurs)
        print('→', end=' ')
        if n1 !=0:
            for i in range(n1):
                print(str(self.entree.valeurs[n1-1-i]), end=' | ')

        n2=len(self.sortie.valeurs)
        if n2 !=0:
            for i in range(n2-1):
                print(str(self.sortie.valeurs[i]), end=' | ')
            print(str(self.sortie.valeurs[-1]), end=' ')
        print('→')

    #2
    def est_vide(self):
        '''Renvoie True si la file est vide,
        False sinon'''
        return ... and ...

    #3
    def enfiler(self,a):
        '''Place l'élément a en tête de file'''
        self.entree. ...

    #4
    def defiler(self):
        '''Si la file n'est pas vide, défile l'élément en tête de file
        et le renvoie'''
        if not self.est_vide():
            if self.sortie.est_vide():
                while not self.entree.est_vide():
                    a=self.entree.depiler()
                    ...
        return ...

    #5
    def tete(self):
        '''Renvoie la valeur de l'élément en tête de file, si la file n'est pas vide.
        Cet élément n'est pas supprimé'''
        if self.entree.est_vide():
            return ...
        else:
            return ...


    #6
    def longueur(self):
        '''Renvoie len nombre d'éléments dans la file'''
        return ...

    #7
    def vider(self):
        '''Vide la file'''
        ...
        ...
```

```
#Jeu de tests pour les files
```

In [32]:
```
#1 init
f=File()
f.affiche() #La file est vide
```

```
→ →
```

In [33]:
```
#2 est_vide
f.est_vide() # renvoie True
```

Out[33]:
```
True
```

In [34]:
```
#3 enfiler
f.enfiler(1)
f.enfiler(2)
f.enfiler(3)
f.enfiler(4)
f.enfiler(5)
f.enfiler(6)
f.affiche() #1 est en tête de file
```

```
→ 6 | 5 | 4 | 3 | 2 | 1 | →
```

In [35]:
```
#4 defiler
f.defiler() #1
f.defiler() #2
f.affiche() #3 est en tête de file
```

```
→ 6 | 5 | 4 | 3 →
```

In [36]:
```
#5 tete
f.tete() # renvoie 3
```

Out[36]:
```
3
```

In [37]:
```
#6 longueur
f.longueur() #4
```

Out[37]:
```
4
```

In [38]:
```
#7 vider
f.vider()
f.affiche() # la file est vide
```

```
→ →
```

Exercice 5 :

Ecrire la fonction `pile_vers_file(pile)` qui prend en paramètre une pile et qui renvoie cette pile transformée en file.

Ainsi la pile ci-dessous,

```
|3|
|2|
|1|
```

deviendra la file `→ 1 | 2 | 3 →`

In [39]:
```
def pile_vers_file(pile):
    file=File()




    return file
```

In [40]:
```
#test pile_vers_file
p=Pile()
p.empiler(1)
p.empiler(2)
p.empiler(3)
p.affiche()

f=pile_vers_file(p)
f.affiche()
```

```
|3|
|2|
|1|
```

```
→ 1 | 2 | 3 | →
```

4. Exercices

Exercice 6 :

On souhaite simuler la distribution d'un jeu de 32 cartes entre 2 joueurs. On suppose que toutes les cartes seront distribuées. On modélise le jeu de 32 cartes par une pile contenant les cartes qui sont mélangées(voir code donné). Pour la distribution, chaque carte au sommet de la pile est donné tour à tour aux deux joueurs pour constituer deux piles. Les deux dernières cartes distribuées sont donc au sommet de chacune des deux piles.

Ecrire les instructions qui permettent de distribuer ainsi un jeu de 32 cartes.

In [41]:
```
from random import *
#jeu de 32 cartes
cartes=[i+j for i in ['7','8','9','10','V','D','R','A'] for j in ["♠","♥","♦","♣"]]
shuffle(cartes)

#le jeu est une pile
jeu=Pile()
jeu.valeurs=cartes
jeu.affiche()
print()

#distribution des cartes (piles)
jeu1=Pile()
jeu2=Pile()




#
```

```
|D♦|
|A♥|
|8♠|
|10♣|
|R♦|
|7♦|
|9♥|
|R♠|
|A♦|
|V♣|
|V♦|
|A♠|
|D♣|
|10♥|
|D♥|
|8♥|
|V♠|
|8♣|
|7♠|
|7♥|
|9♠|
|9♣|
|V♥|
|R♥|
|9♦|
|7♣|
|A♣|
|D♠|
|R♣|
|8♦|
|10♠|
|10♦|
```

In [42]:
```
#tests
print('Jeu 1')
jeu1.affiche()
print()
print('Jeu 2')
jeu2.affiche()
```

```
Jeu 1
|10♠|
|R♣|
|A♣|
|9♦|
|V♥|
|9♠|
|7♠|
|V♠|
|D♥|
|D♣|
|V♦|
|A♦|
|9♥|
|R♦|
|8♠|
|D♦|
```

```
Jeu 2
|10♦|
|8♦|
|D♠|
|7♣|
|R♥|
|9♣|
|7♥|
|8♣|
|8♥|
|10♥|
|A♠|
|V♣|
|R♠|
|7♦|
|10♣|
|A♥|
```

Exercice 7 :

Dans certaines calculatrices, les expressions sont évaluées à l'aide de la notation polonaise inversée (ce nom vient du Mathématicien polonais *Jan Łukasiewicz*).Cette notation ne cécéssite aucune parenthèse ni aucune règle de priorité. Elle place les opérandes avant les opérateurs.

Exemple : En notation polonaise inversée, l'expression `(1+2)*3+4` peut s'écrire `4321+*+` .

- Pour calculer cette expression, on lit de gauche à droite jusqu'à trouver un opérateur, ici $+$.
- On utilise ensuite les deux derniers opérandes trouvés qui sont 2 et 1, puis on effectue l'opération $1+2$.
- On remplace les opérandes et l'opérateur utilisés par le résultat de l'opération, ici l'expression devient alors `433*+` .
- On utilise l'opérateur suivant sur les deux derniers opérandes présents dans l'expression.Ici on effectue donc 3×3.
- On remplace les opérandes et l'opérateur utilisés par le résultat de l'opération, ici l'expression devient alors `49+` .
- On réitère le procédé sur les derniers opérandes, ce qui revient ici à effectuer le calcul $4+9$.
- Le résultat est bien 13.

Partie A : Manipulation

1. Quelle notation sera utilisée pour calculer `(3+4)*6+1` ? **Réponse :**
2. A quelle expression correspond la notation `4132*+*` ? **Réponse :**

Partie B : Implémentation

Pour simplifier , on suppose que :

- L'on utilise uniquement les opérateurs `+` et `*` .
- Les opérandes sont des entiers naturels à un chiffre (entre 0 et 9).
- L'expression à évaluer est une chaîne de caractères.

La valeur d'une expression en notation polonaise inversée peut être calculée à l'aide d'une pile pour stocker les résultats intermédiaires. Pour cela, on parcourt l'expression de gauche à droite et on effectue les actions suivantes :

- Si on voit un nombre , on le place sur la pile.
- Si on voit un opérateur, on récupère les deux nombres au sommet de la pile, on leur applique l'opérateur et on replace le résultat au sommet de la pile.
- A la fin du processus, il reste exactement un nombre sur la pile, qui est le résultat.

Ecrire la fonction `npi(expr)` qui prend en paramètre un chaîne de caractères contenant une expression à évaluer et qui renvoie le résultat de l'évalution de cette expression.

In [43]:
```
def npi(expr):
    #pile pour stocker les résultats intermédiaires
    pile=Pile()
    #variable qui contient le résultat final
    res=0









    return res
```

In [44]:
```
#tests pour la fonction npi
e1='4312+*+'
e2='1643+*+'
e3='4132*+*'
assert(npi(e1)==13)
assert(npi(e2)==43)
assert(npi(e3)==28)
```

Exercice 8 :

- On dit qu'une chaîne de caractères comprenant, entre autre choses, des parenthèses `(` et `)` est bien parenthésée lorsque chaque parenthèse ouvrante est associée à une unique fermante et réciproquement.

1. Dans la chaîne `.(..(.).())` :

   - La parenthèse ouvrante d'indice 1 est associée à la prenthèse fermante d'indice 10.
   - La parenthèse fermante d'indice 6 est associée à la parenthèse ouvrante d'indice 4.
   - Quelle autre association de parenthèses peut-on faire dans cette chaîne ?

     Réponse :

1. On souhaite écrire la fonction `ouvrante_associee(f,chaine)` qui prend en paramètres une chaîne de caractères bien parenthésée et un l'indice d'une parenthèse fermante.Cette fonction renvoie l'indice de la parenthèse ouvrante correspondante :

   - Une parenthèse fermante correspond à la dernière parenthèse à avoir été ouverte.
   - On peut utiliser une pile pour suivre ces associations.
   - Cette pile enregistre les indices des parenthèses ouvertes qui n'on pas encore été associées à des parenthèses fermantes.
   - En parcourant la chaîne jusqu'à l'indice `f` :
     - Si l'on trouve une parenthèse ouvrante, on empile son indice au sommet de la pile.
     - Si l'on trouuve une parenthèse fermante, on dépile l'indice stocké au sommet de la pile, puisqu'il correspond à la parenthèse ouvrante associée.
   - Une fois l'indice `f` atteint, le dernier élément au sommet de la pile est bien celui de la parenthèse ouvrante cherché, on le renvoie.

A l'aide de ces indications, compléter la fonction.

In [45]:
```
def ouvrante_associee(f,chaine):
    '''chaine : chaine de caractères bien parenthésée
       f : indice d'une parenthèse fermante, entier
       return : indice de sa parenthèse ouvrante associée, entier'''
    pile=Pile()
    for i in range(f):
        if chaine[i]==...:
            pile.empiler(...)
        elif chaine[i]==...:
            ...
    return ...
```

In [46]:
```
#tests ouvrante_associée
chaine='.(..(.).())'
print(ouvrante_associee(10, chaine)) #1
print(ouvrante_associee(6, chaine)) #4
print(ouvrante_associee(9, chaine)) #8
```

```
1
4
8
```

Exercice 9 :

Une chaîne de caractère est bien parenthésée si chaque parenthèse fermante est associée à une parenthèse ouvrante. Ecrire la fonction `bonnes_parentheses(chaine)` qui prend en paramètre une chaîne de caractères parenthésée qui renvoie `True` si la chaîne est bien parenthésée et `False` sinon.

Aide : On pourra utiliser une pile pour stocker toutes les parenthèses ouvrantes non encore fermées lors du parcours de la chaîne :

- Lorsque l'on trouve un parenthèse fermante, plusieurs cas peuvent indiquer que la chaîne est mal parenthésée:
  - La pile est vide (une parenthèse fermante est obligatoirement associé à une ouvrante)
  - La valeur que l'on dépile n'est pas une parenthèse ouvrante
- A la fin du parcours, si la pile n'est pas vide, c'est également que la chaîne n'est pas bien parenthésée.

In [47]:
```
def bonnes_parentheses(chaine):
    pile=Pile()
    reponse =True









    return reponse
```

In [48]:
```
#tests bonnes_parenthèses
assert(bonnes_parentheses('()')==True)
assert(bonnes_parentheses('())')==False)
assert(bonnes_parentheses('(..(.))')==True)
```