Structures de données

Listes, piles et files

Objectifs pédagogiques :

- ✓ Types abstraits : liste, pile et file
- ✓ Distinguer les modes FIFO (first in first out) et LIFO (last in first out) des piles et des files.

Les listes, les piles (*stack* en anglais) et les files (*queue* en anglais) sont des structures abstraites de données fondamentales en informatique. Elles diffèrent par les conditions d'ajout et d'accès aux éléments qui les constituent.

1. Notions de liste, pile et de file

1.1. Les listes

Une liste est une structure abstraite de données permettant de regrouper des données sous une forme séquentielle. Elle est constituée d'éléments d'un même type, chacun possédant un rang. Une liste est évolutive : on peut ajouter ou supprimer n'importe lequel de ses éléments. Une liste L est composée de 2 parties :

- sa tête (souvent noté *car*), qui correspond au dernier élément ajouté à la liste ;
- sa queue (souvent noté *cdr*) qui correspond au reste de la liste ;

Le langage de programmation Lisp (inventé par John McCarthy en 1958) a été l'un des premiers langages de programmation à introduire cette notion de liste (Lisp signifie "list processing"). Voici les opérations qui peuvent être effectuées sur une liste :

Actions	Instruction
Créer une liste L vide	`L = vide()`
Tester si la liste L est vide	`estVide(L)`
Ajouter un élément x en tête de la liste L	`ajouteEnTete(x,L)`
Supprimer la tête x d'une liste L et renvoyer cette tête x	`supprEnTete(L)`
Compter le nombre d'éléments dans une liste L	`compte(L)`
Créer une nouvelle liste L1 à partir d'un élément x et d'une liste existante L	`L1 = cons(x,L)`

Q1. Voici une série d'instructions (les instructions ci-dessous s'enchaînent), expliquez ce qui se passe à chacune des étapes :

```
L=vide()
estVide(L)
ajoutEnTete(3,L)
estVide(L)
ajoutEnTete(5,L)
ajoutEnTete(8,L)
t = supprEnTete(L)
L1 = vide()
L2 = cons(8, cons(5, cons(3, L1)))
```

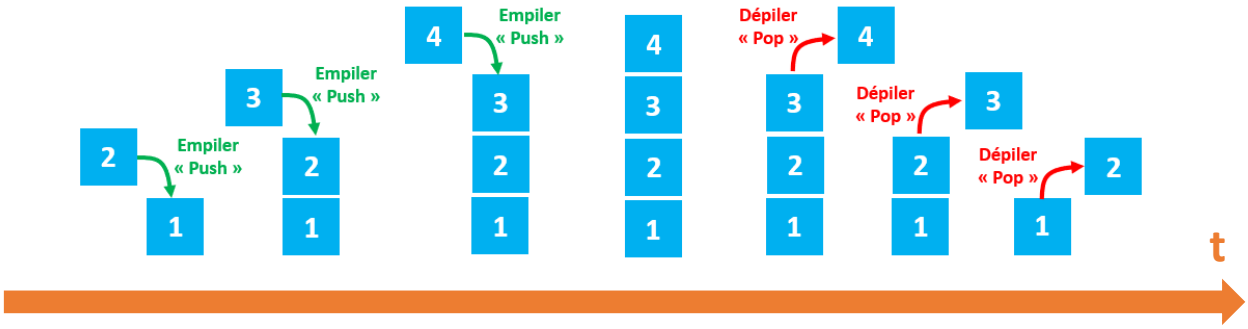
Q2. Voici une série d'instructions (les instructions ci-dessous s'enchaînent), expliquez ce qui se passe à chacune des étapes :

```
L = vide()
ajoutEnTete(10,L)
ajoutEnTete(9,L)
ajoutEnTete(7,L)
L1 = vide()
L2 = cons(5, cons(4, cons(3, cons (2, cons(1, cons(0,L1))))))
```

Remarque : les types abstraits de données piles et files, comme nous allons le voir ci-après, sont des listes possédant des restrictions au niveau de l'ajout ou de la suppression de leurs éléments. En effet ces actions ne pourront être réalisées que selon certaines modalités aux extrémités de ces deux types de structures abstraites de données.

1.2. Les piles (stacks)

Les **piles** sont fondées sur le principe du « **dernier arrivé, premier sorti** » : elles sont dites de type **LIFO** (**Last In, First Out**). C'est le principe même de la pile d'assiette : c'est la dernière assiette posée sur la pile d'assiettes sales qui sera la première lavée.



L'insertion d'un élément dans la pile est appelée « **Empiler** » et la suppression d'un élément de la pile est appelée « **Dépiler** ». Dans la pile, nous gardons toujours trace du dernier élément présent dans la liste avec un pointeur appelé top.

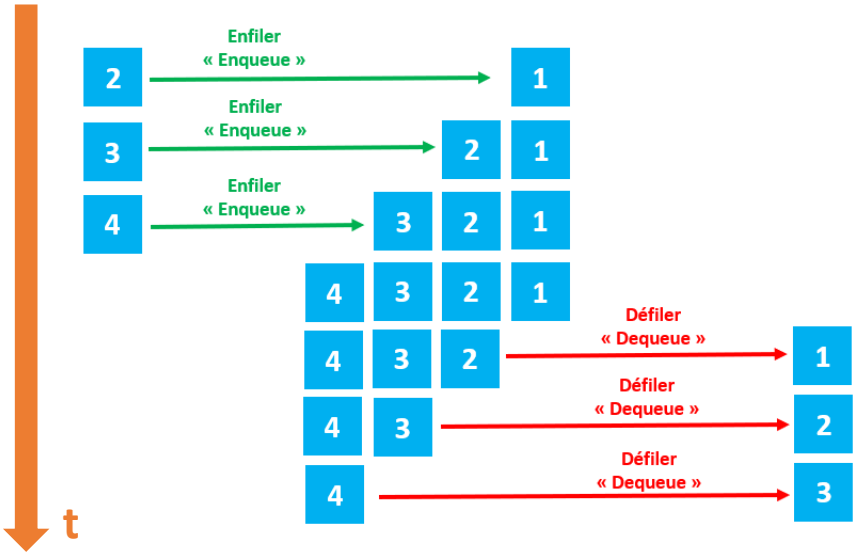
De nombreuses applications s'appuient sur l'utilisation d'une pile. En voici quelques-unes :

- dans un navigateur web, une pile sert à mémoriser les pages web visitées ; l'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant sur le bouton « Afficher la page précédente »
- l'évaluation des expressions mathématiques en notation post-fixée (ou polonaise inverse) utilise une pile ;
- la fonction « Annuler la frappe » (Undo en anglais) d'un traitement de texte mémorise les modification apportées au texte dans une pile ;
- la récursivité (une fonction qui fait appel à elle-même) utilise également une pile ;
- etc …

Remarque : en informatique, un **dépassement de pile** ou **débordement de pile** (en anglais, ***stack overflow***) est un bug causé par un processus qui, lors de l'écriture dans une pile, écrit à l'extérieur de l'espace alloué à la pile, écrasant ainsi des informations nécessaires au processus. **Stack Overflow** est aussi connu comme étant un site web proposant des questions et réponses sous la forme d'un forum d'entre-aide sur un large choix de thèmes concernant la programmation informatique. Il y a de forte chance que vous trouviez la réponse à un problème informatique même ardu sur Stack Overflow !

1.3. Les files (queues)

Les **files** sont fondées sur le principe du « **premier arrivé, premier sorti** » : elles sont dites de type **FIFO** (**First In, First Out**). C'est le principe de la file d'attente devant un guichet.



L'insertion d'un élément dans une file s'appelle une opération de mise en file « **Enfiler** » et la suppression d'un élément s'appelle une opération de retrait de la file « **Défiler** ». Dans la file, nous maintenons toujours deux pointeurs, l'un pointant sur l'élément qui a été inséré en premier et qui est toujours présent dans la liste avec le pointeur en avant et l'autre pointant sur l'élément inséré en dernier avec le pointeur arrière.

En général, on utilise une file pour mémoriser temporairement des transactions qui doivent attendre pour être traitées. Voici quelques exemples d'applications :

- les serveurs d'impression, qui doivent traiter des requêtes dans l'ordre dans lequel elles arrivent, et les insère dans une file d'attente (ou une queue) ;
- les requêtes entre machines sur un réseau ;
- certains moteurs multi-tâches, dans un système d'exploitation, qui doivent accorder du temps machine à chaque tâche, sans en privilégier une plus qu'une autre ;
- un algorithme de parcours en largeur utilise une file pour mémoriser les nœuds visités ;
- on utilise des files pour créer toutes sortes de mémoires tampons (buffers en anglais) ;
- etc…

1.4. Piles vs files : synthèse

Pile	File
Les objets sont insérés et supprimés à 1 seule extrémité	Les objets sont insérés et retirés aux 2 extrémités.
Dans les piles, un seul pointeur est utilisé. Il pointe vers le haut de la pile.	Dans les files, deux pointeurs différents sont utilisés pour les extrémités; le tète et la fin.
Dans les piles, le dernier objet inséré est le premier à sortir.	Dans les files, l'objet inséré en premier est le premier qui sera supprimé.
Les piles suivent l'ordre Last In First Out (**LIFO**)	Les files suivent l'ordre First In First Out (**FIFO**)
Les opérations de pile s'appellent « **Empiler** » et « **Dépiler** ».	Les opérations de file sont appelées « **Enfiler** » et « **Défiler** ».
Les piles sont visualisées sous forme de collections verticales.	Les files sont visualisées sous forme de collections horizontales.

Q3. Expliquer la différence fondamentale entre une pile et une file.
Q4. Que désignent les acronymes FIFO et LIFO ?
Q5. Donner quelques exemples d'application des piles et des files.
Q6. En quoi différent les listes des piles et des files ?

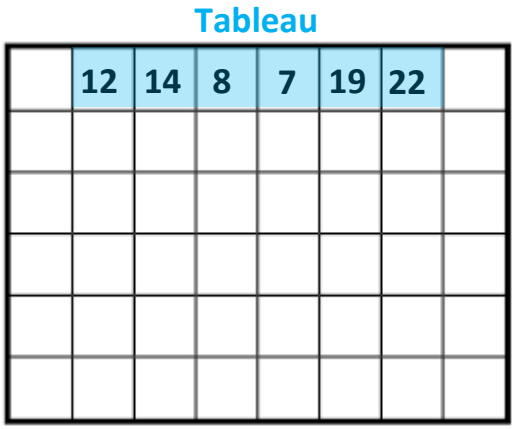
2. Types abstraits et représentation réelle des données

Les listes, les piles et les files sont des types mathématiques abstraits de représentation des données. Pour implémenter ces types abstraits de données d'une manière concrète dans la mémoire RAM d'une machine la plupart des langages de programmation utilisent 2 grandes familles de structures :

- les tableaux ;
- les listes chaînées.

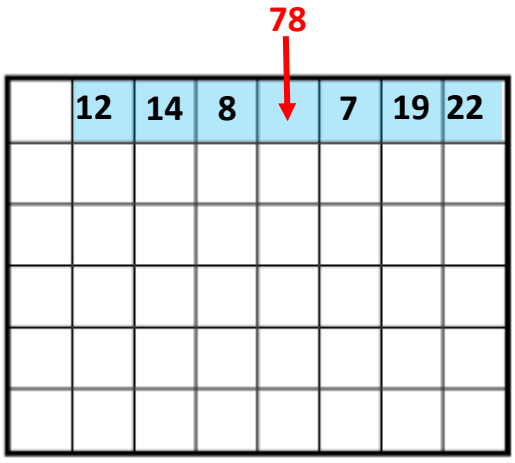
2.1. Les tableaux

Un tableau est une suite contiguë de cases mémoires (les adresses des cases mémoires se suivent). Le système réserve (alloue) une plage d'adresses mémoires afin d'y stocker la valeur des éléments d'une pile ou d'une file par exemple.



RAM : représentation des cases mémoires (tableau statique)

La taille d'un tableau est fixe : une fois que l'on a défini le nombre d'éléments que le tableau peut accueillir, il n'est pas possible modifier sa taille. Si l'on veut insérer une nouvelle donnée, on doit créer un nouveau tableau plus grand et déplacer les éléments du premier tableau vers le second tout en ajoutant la donnée au bon endroit !
Dans certains langages de programmation, on trouve une version "évoluée" des tableaux : les tableaux dynamiques. Les tableaux dynamiques ont une taille qui peut varier. Il est donc relativement simple d'insérer des éléments dans le tableau. Ce type de tableaux permet d'implémenter facilement le type abstrait liste (de même pour les piles et les files).

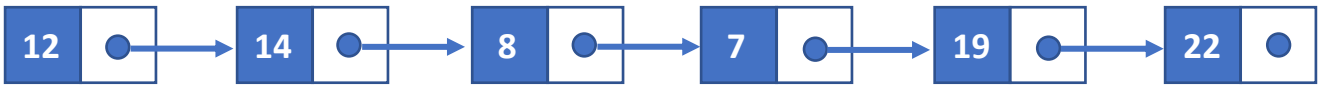


RAM : représentation des cases mémoires (tableau dynamique)

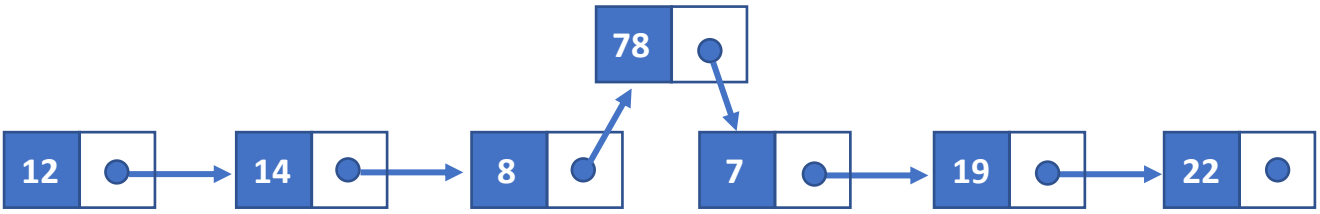
Remarque : les « listes Python » sont des tableaux dynamiques. Attention à ne pas confondre les «listes Python » avec le type abstrait « liste » défini au § 1.1. ci-dessus : ce sont de "faux amis" !

2.2. Les listes chaînées

Dans une liste chaînée, à chaque élément de la liste on associe 2 cases mémoires : la première case contient l'élément et la deuxième contient l'adresse mémoire de l'élément suivant.



Avec ce type de structure, il est aisé d'insérer un nouvel élément :



Q7. A quoi servent les tableaux et les listes chaînées ?
Q8. Quel est la différence entre un tableau statique et un tableau dynamique ?
Q9. Quel est le principal avantage d'une liste chaînée.

3. Pile : implémentation en Python

3.1. Une pile est une structure abstraite de données

Une pile est une structure de données abstraite sur laquelle on va pouvoir réaliser un nombre restreint d'opérations autorisées. Si l'on reprend l'idée "donnée = assiette", une pile est semblable à une pile d'assiettes et l'on précise les opérations permises :

- on peut empiler une assiette (ajouter une assiette en haut de pile) ;
- on peut dépiler une assiette (enlever l'assiette en haut de pile) ;
- on peut savoir si la pile est vide ou non ;
- on peut connaitre quelle assiette est au sommet ;
- on peut connaître le nombre d'assiettes dans la pile.

Voici résumé sous la forme d'un tableau les opérations que l'on peut réaliser sur un **objet** de type **Pile** :

Actions sur la pile	Méthodes de la classe *Pile*
La pile est-elle vide ?	`estVide()`
Empiler un nouvel élément sur la pile	`empiler(élément)`
Dépiler un élément de la pile	`depiler()`
Lire la valeur au sommet de la pile	`lireSommet()`
Afficher le nombre d'éléments présents dans la pile	`taille()`

En programmation orientée objet (POO), on instancie un objet *ma_pile* vide appartenant à la classe *Pile* de la manière suivante : *ma_pile = Pile()*
Une fois l'objet instancié on peut lui appliquer les méthodes de la classe à laquelle il appartient de la manière suivante : *nom_objet.methode_classe()*

Q10. Indiquer quelles seront les instructions dans l'ordre chronologique permettant de créer la pile 12, 14, 8, 7, 19 et 22, le sommet de la pile étant 22 ?
Q11. Quelle instruction affichera le sommet de la pile ?
Q12. Quelle instruction donnera le nombre d'éléments de la pile ?
Q13. On souhaite insérer l'élément 20 entre les éléments 8 et 7 en conservant tous les autres éléments de la pile : comment doit-on procéder ?
Q14. Quelle instruction supprimera l'élément correspondant au sommet de la pile ?
Q15. Comment supprimer tous les éléments qui restent dans la pile et comment vérifier qu'à la fin la pile est vide ?

Remarque : on ne peut pas dans une pile, comme dans une liste, prendre une assiette à n'importe quel rang dans la pile (on risquerait de tout faire tomber et de casser toutes les assiettes), ni ajouter une assiette n'importe où dans la pile.

3.2. Implémentation d'une pile en Python avec une liste

Pour implémenter une pile en python, on peut utiliser le type liste. Et pour interdire d'autres opérations que celles signalées dans le tableau ci-dessus, on peut définir une **classe** *Pile* dont les méthodes correspondront aux opérations autorisées.

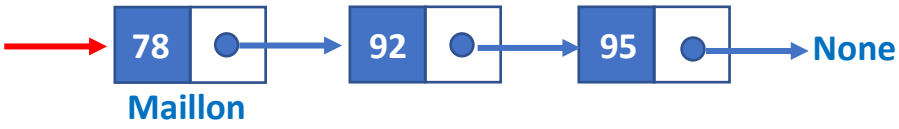
Entrée [1]:
```
class Pile:

    def __init__(self):
        self.valeurs = []

    def empiler(self, valeur):
        self.valeurs.append(valeur)

    def depiler(self):
        if self.valeurs:
            return self.valeurs.pop()

    def estVide(self):
        return self.valeurs == []

    def taille(self):
        return len(self.valeurs)

    def lireSommet(self):
        return self.valeurs[-1]


    def __str__(self):
        ch = ''
        for x in self.valeurs:
            ch = "|\t" + str(x) + "\t|" + "\n" + ch
        ch = "\nEtat de la pile:\n" + ch
        return ch



p = Pile()
p.empiler(9)
p.empiler(2)
p.empiler(5)
p.empiler(4)
print(p)

p.depiler()
print(p)

p.empiler(7)
print(p)

print(p.lireSommet())
print(p.taille())
```

```
Etat de la pile:
|	4	|
|	5	|
|	2	|
|	9	|


Etat de la pile:
|	5	|
|	2	|
|	9	|


Etat de la pile:
|	7	|
|	5	|
|	2	|
|	9	|

7
4
```

Q16. Quel est le constructeur de la classe *Pile* ?
Q17. Quelles sont les méthodes de la classe *Pile* ? Quelles sont les méthodes spéciales ?
Q18. Tester le programme afin d'effectuer quelques opérations sur des piles.

3.3. Implémentation d'une pile en Python avec une liste chaînée

Chaque « boîte » ou « maillon » contient une donnée, la flèche à droite de la boîte est le lien sur la donnée suivante. A noter que le dernier maillon ne pointe sur rien (on utilisera ***None*** en python). La première flèche à gauche (en rouge) peut être associée dans le code python ci-dessous à la variable *p* (la variable *p* de type *Pile* "**pointe**" sur le premier maillon et permet de le récupérer avec la syntaxe *p.lireSommet()*).



Le programme suivant permet d'implémenter une pile en utilisant une liste chaînée. Le programme repose sur une classe *Maillon* et une classe *Pile*.

Entrée [2]:
```
class Maillon:

    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant



class Pile:

    def __init__(self):
        self.taille = 0 # nombre d'éléments dans la pile
        self.sommet = None


    def empiler(self, valeur):
        self.sommet = Maillon(valeur, self.sommet)
        self.taille += 1

    def depiler(self):
        if self.taille > 0:
            valeur = self.sommet.valeur
            self.sommet = self.sommet.suivant
            self.taille -= 1
            return valeur

    def estVide(self):
        return self.taille == 0


    def lireSommet(self):
        return self.sommet.valeur


    def __str__(self):
        ch = "\nEtat de la pile:\n"
        sommet = self.sommet
        while sommet != None:
            ch += "|\t" + str(sommet.valeur) + "\t|" + "\n"
            sommet = sommet.suivant

        return ch
```

Q19. En utilisant les 2 classes précédentes, créer un programme permettant de créer la pile initiale sur laquelle on réalisera les actions affichées ci-dessous :

```
Etat de la pile:
|       4       |
|       5       |
|       2       |
|       9       |

4
4

Etat de la pile:
|       2       |
|       9       |

2
2

Etat de la pile:
|       78      |
|       2       |
|       9       |

78
3
```

Pile initiale

3.4. La notation polonaise inversée

Lorsque l'on écrit usuellement des expressions algébriques, les parenthèses sont indispensables. Elles permettent par exemple de distinguer les expressions 1 + 2 × 3 et (1 + 2) × 3.

Avec la notation préfixée (appelée aussi notation polonaise, en référence au mathématicien polonais, Jan Lukasiewic), les parenthèses ne sont plus nécessaires :

- 1 + 2 * 3 sera noté + 1 * 2 3
- (1 + 2) * 3 sera noté * + 1 2 3

Dérivée de la notation polonaise utilisée pour la première fois en 1924 par le mathématicien polonais Jan Łukasiewicz, la NPI (Notation Plonaise Inversée) a été inventée par le philosophe et informaticien australien Charles Leonard Hamblin au milieu des années 1950, pour permettre les calculs sans faire référence à une quelconque adresse mémoire. À la fin des années 1960, elle a été diffusée dans le public comme interface utilisateur avec les calculatrices de bureau de Hewlett-Packard (HP-9100), puis avec la calculatrice scientifique HP-35 en 1972.

■ **Principe :**

Il est par exemple possible de stocker une expression en notation polonaise inversée dans une liste. Par exemple, l'expression "+ * - / 10 2 4 3 6" est stockée dans la liste **['+', '*', '-', '/', 10, 2, 4, 3, 6]**.

Pour évaluer cette expression, on utilisera une pile. On parcourt la liste de la fin vers le début :

- si l'élément est un nombre, on l'empile.
- si l'élément est un opérateur, on dépile ses deux opérandes et on calcule, puis on empile le résultat de ce calcul.

Le résultat de l'expression est l'unique élément restant dans la pile.

Q20. Soit l'opération arithmétique : 2 x (2 + 4). Comment sera-t-elle codée en notation polonaise inversée ?
Q21. Représenter sous la forme d'une pile l'exécution du calcul précédent.
Q22. En utilisant le principe de pile, évaluer l'expression "+ * - / 10 2 4 3 6" exprimée en notation polonaise inversée. On représentera sous la forme d'un schéma l'évolution de la pile permettant de réaliser le calcul.

Q23. Vérifier avec le programme ci-dessous le résultat obtenu à la question Q22.

Entrée [4]:

```
class Maillon:

    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant



class Pile:

    def __init__(self):
        self.taille = 0 # nombre d'assiettes dans la pile
        self.sommet = None


    def empiler(self, valeur):
        self.sommet = Maillon(valeur, self.sommet)
        self.taille += 1

    def depiler(self):
        if self.taille > 0:
            valeur = self.sommet.valeur
            self.sommet = self.sommet.suivant
            self.taille -= 1
            return valeur

    def estVide(self):
        return self.taille == 0

    def __str__(self):
        ch = "\nEtat de la pile:\n"
        sommet = self.sommet
        while sommet != None:
            ch += "|\t" + str(sommet.valeur) + "\t|" + "\n"
            sommet = sommet.suivant

        return ch


def prefixe(expression):
    pile = Pile()
    for c in reversed(expression):
        if isinstance(c, int):
            pile.empiler(c)
            print(pile)
        else:
            a = pile.depiler()
            b = pile.depiler()
            pile.empiler(eval(str(a) + c + str(b)))
            print(pile)
    return pile.depiler()


r = prefixe(['+', '*', '-', '/', 10, 2, 4, 3, 6])
print(r)
```

4. File : implémentation en Python

4.1. Une file est une structure abstraite de données

Une file est une structure abstraite de données. Si l'on reprend l'idée de la file d'attente les opérations permises sont le suivantes :

- on peut enfiler un élément (une personne arrive en queue de file)
- on peut défiler un élément (la personne en début de file sort de la file).
- on peut savoir si la file est vide ou non.
- on peut afficher la longueur de la file (nombre de personnes la composant)

Voici résumé sous la forme d'un tableau les opérations que l'on peut réaliser sur un **objet** de type **File** :

Actions sur la file	Méthodes de la classe *File*
Enfiler un nouvel élément	`enfiler(élément)`
Défiler	`defiler()`
La file est-elle vide ?	`estVide()`
Afficher le nombre d'éléments présents dans la file	`longeur()`

En programmation orientée objet (POO), on instancie un objet *ma_file* vide appartenant à la classe *File* de la manière suivante : *ma_file = File()*
Une fois l'objet instancié on peut lui appliquer les méthodes de la classe à laquelle il appartient de la manière suivante : *nom_objet.methode_classe()*

Q24. Indiquer quelles seront les instructions dans l'ordre chronologique permettant de créer la file 12, 14, 8, 7, 19 et 22, l'élément de tête de la file étant 22 ?
Q25. Si on appliquer les instructions suivantes à la file précédent, indiquer la composition de la file après l'exécution de celles-ci.
defiler()
defiler()
enfiler(78)

4.2. Implémentation d'une file en Python avec une liste

Pour implémenter une file en python, on peut utiliser le type liste. Et pour interdire d'autres opérations que celles signalées dans le tableau ci-dessus, on peut définir une **classe** *File* dont les méthodes correspondront aux opérations autorisées.

Entrée [5]:
```
class File:

    def __init__(self):
        self.valeurs = []

    def enfiler(self, valeur):
        self.valeurs.append(valeur)

    def defiler(self):
        if self.valeurs:
            return self.valeurs.pop(0)

    def estVide(self):
        return self.valeurs == []

    @property
    def longueur(self):
        return len(self.valeurs)


    def __str__(self):
        ch = "\nEtat de la file:\n"
        for x in self.valeurs:
            ch +=  str(x) + " "
        return ch
```

```
Entrée [6]: q = File()
            q.enfiler(75)
            q.enfiler(78)
            q.enfiler(92)
            q.enfiler(93)
            q.enfiler(95)

            print("La file est-elle vide: ", q.estVide())
            print("Longueur de la file: ", q.longueur)
            print(q)
            print("\n")


            q.defiler()
            q.defiler()
            q.enfiler(75)
            q.enfiler(78)
            print("La file est-elle vide: ", q.estVide())
            print("Longueur de la file: ", q.longueur)
            print(q)
```

```
La file est-elle vide:  False
Longueur de la file:  5

Etat de la file:
75 78 92 93 95


La file est-elle vide:  False
Longueur de la file:  5

Etat de la file:
92 93 95 75 78
```

Q26. Identifier la tête et la queue de la première file. Le programme a-t-il le comportement attendu ?

4.3. Implémentation d'une file en Python avec une liste doublement chaînée

La structure de liste doublement chaînée est bien adaptée pour une implémentation de file. Chaque « boîte » ou « maillon » contient une donnée, la flèche à droite de la boîte est le lien sur la donnée suivante, la flèche à gauche le lien sur la donnée précédente. Le dernier maillon a un suivant "vide", le premier maillon a un prédécesseur vide (on utilisera **None** en python). La flèche en rouge à gauche peut être associée dans le code python ci-dessous à la variable *q* (la variable *q* de type *File* "**pointe**" sur le premier maillon et permet de le récupérer avec la syntaxe *q.debut*).

Le programme ci-après permet d'implémenter une file en utilisant une liste doublement chaînée. Le programme repose sur une classe *Maillon* et une classe *File*.

Entrée [7]:

```
class Maillon:

    def __init__(self, valeur, precedent=None, suivant=None):
        self.valeur = valeur
        self.precedent = precedent
        self.suivant = suivant



class File:

    def __init__(self):
        self.longueur = 0
        self.debut = None
        self.fin = None

    def enfiler(self, valeur):
        if self.longueur == 0:
            self.debut = self.fin = Maillon(valeur)
        else:
            self.fin = Maillon(valeur, self.fin)
            self.fin.precedent.suivant = self.fin
        self.longueur += 1


    def defiler(self):
        if self.longueur > 0:
            valeur = self.debut.valeur
            if self.longueur > 1:
                self.debut = self.debut.suivant
                self.debut.precedent = None
            else:
                self.debut = self.fin = None
            self.longueur -= 1
        return valeur


    def estVide(self):
        return self.longueur == 0




    def __str__(self):
        ch = "\nEtat de la file:\n"
        maillon = self.debut
        while maillon != None:
            ch +=  str(maillon.valeur) + " "
            maillon = maillon.suivant
        return ch
```

Q27. Vérifier que le programme précédent redonne bien les mêmes résultats que ceux du § 4.2.