

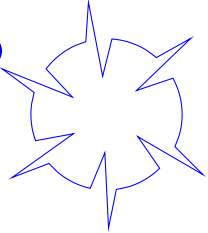
CHAPITRE	<h1 style="font-size: 4em; margin: 0;">4</h1> <h1 style="font-size: 3em; margin: 0;">STRUCTURES LINÉAIRES</h1> 
----------	--

Table des matières

1	Une première situation :	1
2	Structures linéaires :	5
2.1	Les piles :	5
2.2	Les files :	5
2.3	Interface et Implémentation :	5
3	Regards sur les listes :	6
3.1	Listes chaînées :	6
3.2	Complexité des opérations :	8
4	Le coin des exercices :	9

1 Une première situation :



L'écriture en notation polonaise est une façon d'écrire une expression algébrique sans utiliser des parenthèses.

Voici quelques exemples :

<i>Notation classique</i>	<i>Notation polonaise</i>
$(1 + 3) \times 5$	$13 + 5 \times$
$5 + 8 \times 2$	$582 \times +$
$(11 + 2) \times (4 + 3)$	$112 + 43 + \times$

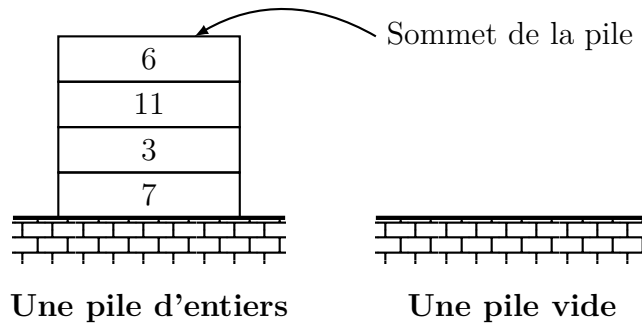
Pour calculer une expression sous cette forme, on utilise une *pile*.



Une pile est une structure linéaire qui se distingue par son mode d'accès.

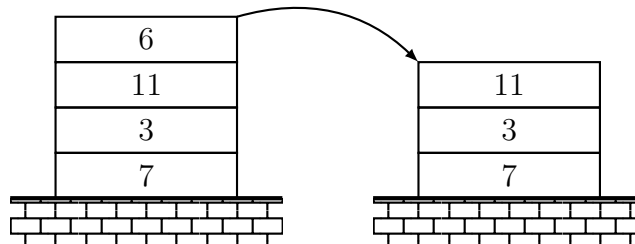
L'accès aux éléments d'une liste se fait par son indice : pour une pile, seul le dernier élément (s'il existe...) est accessible. On parle de mode *FIFO* (First In First Out), pour caractériser cet accès.

Une pile est munie de trois opérations de base : *créer_pile_vide*, *empiler*, *depiler* et *est_vide*.



Pour obtenir la pile d'entiers ci-dessus, on a d'abord à partir d'une pile vide, *empiler* 7 puis 3 puis 11 et enfin 6.

L'action de *depiler* la pile, consiste à enlever (et éventuellement retourner) le sommet de la pile, donc 6 dans ce cas.

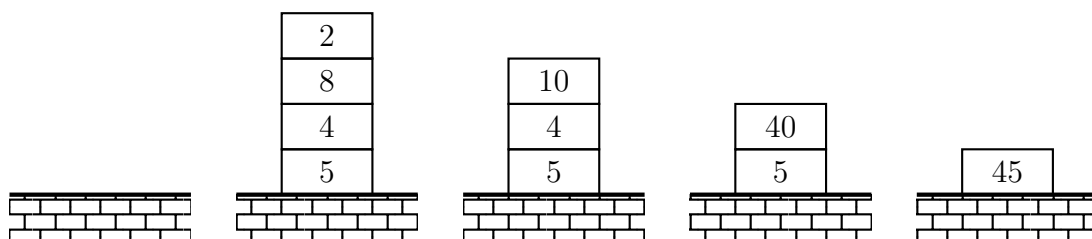


Pour programmer une calculatrice polonaise, nous allons utiliser une pile et suivre scrupuleusement l'algorithme suivant :

```
Données : une expression polonaise et une pile P vide.
Résultat : le résultat
pour chaque élément de l'expression (lue de gauche à droite) faire
  si l'élément est un nombre alors
    | l'empiler dans la pile
  fin
  si l'élément est une opération alors
    | ① dépiler une fois dans une variable a
    | ② dépiler une autre fois dans une variable b
    | ③ calculer l'opération entre a et b
    | ④ empiler le résultat de cette opération.
  fin
fin
retourner Le dernier élément de la pile
```

Algorithme 1 : Calculatrice polonaise

Voici plusieurs états de la pile pour l'expression $5482 + \times +$:



L'algorithme étant compris, nous allons maintenant le *programmer*. Mais avant cela, une question se pose ?



Comment implémenter une pile ?

La structure n'est pas *native* en python contrairement à d'autres langages. Il faut donc la créer et nous allons utiliser nos connaissances sur les listes (Python) pour cela.



Une pile sera une liste Python dont le sommet est le *dernier élément* !

Ce choix est arbitraire et pourrait être débattu.

Voici donc l'*interface* de la pile dans un paradigme fonctionnel :

```
1 def creer_pile_vide():
2     """creating empty stack"""
3     return []
4 def est_vide(p):
5     """empty stack or not"""
6     return p == []
7 def empiler(elt, p):
8     """stack on element"""
9     return p.append(elt)
10 def depiler(p):
11     """unstack an element and return it"""
12     if not est_vide(p):
13         return p.pop()
```



Attention aux effets de bord !

Les fonctions `empiler` et `depiler` créent des *effets de bords* : elles changent l'état de la liste qui est passée en paramètre.

La programmation en orienté objet est aussi très pertinente dans la création de nouveaux objets. On aurait alors :

```

1 class Pile():
2     def __init__(self):
3         self.valeurs = []
4     def empiler(self, valeur):
5         self.valeurs.append(valeur)
6     def depiler(self):
7         return self.valeurs.pop()
8     def est_vide(self):
9         return self.valeurs == []

```

Comprenez une chose :



L'interface des piles ne change pas quel que soit l'implémentation utilisée.

On peut d'ores et déjà *enrichir* la classe `Pile` par la méthode `__str__` qui indique à la méthode `print` comment afficher l'état de la pile dans la console.

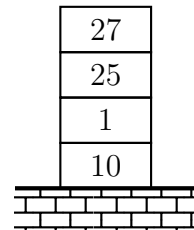
```

1 def __str__(self):
2     ch = ''
3     for x in self.valeurs:
4         ch = "|\\t" + str(x) + "\\t|" + "\\n" + ch
5     ch = "\\n Etat de la pile:\\n" + ch
6     return ch

```

Exercice n°1

- ❶ En utilisant les deux implémentations possibles, créer la pile ci-contre.
- ❷ Faites afficher l'état de la pile.
- ❸ Définissez la méthode `hauteur` qui détermine le nombre d'éléments d'une pile.



Revenons à notre calculatrice polonaise... On considère qu'une expression algébrique sera donnée sous la forme d'un `str`, par exemple : `"5 4 8 2 + * +"`.

Exercice n°2

Créer la fonction `calculatrice_polonaise`, qui à partir d'une expression algébrique, retourne le résultat correspondant.

2 Structures linéaires :

Dans la partie introductive de ce cours, on trouve la notion de pile qui est une *structure linéaire*.



Un type abstrait est une structure qui *stockent* de l'information : la structure est *linéaire* lorsque le stockage est *unidimensionnelle*.

Les *listes* vues en première, sont des structures de données linéaires, les *pires* aussi.

2.1 Les piles :

La pile, comme la liste, permet de stocker des données et d'y accéder : la différence réside dans le *mode d'accès*.



On parle de mode *LIFO* (Last In, First Out, donc, dernier arrivé, premier sorti), c'est-à-dire que le dernier élément ajouté à la structure sera le prochain élément auquel on accèdera.

Les piles ont un *sommet* par lequel on accède aux données, en entrée comme en sortie. C'est avec un *indice* qu'on accède aux éléments d'une liste...

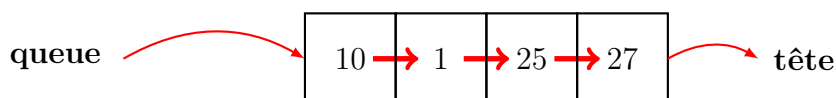
2.2 Les files :

La file est aussi une structure de données linéaires mais son mode d'accès diffère des piles et des listes.



On parle de mode *FIFO* (First in, First out, donc, premier arrivé, premier sorti), c'est-à-dire que le premier élément ayant été ajouté à la structure sera le prochain élément auquel on accèdera.

Les files ont une *tête* pour la sortie et une *queue* pour l'entrée.



2.3 Interface et Implémentation :

Une structure de données permet de gérer un ensemble de données à partir d'*un jeu réduit* de méthodes qui sont les seuls moyens d'accéder à tel ou tel élément, de modifier l'ensemble des données, d'en créer un nouveau, etc...



La description de ce jeu réduit de méthodes ainsi que de leur sémantique s'appelle l'*interface* de la structure de données.

Pour une même interface, on peut proposer *diverses implémentations* de la structure de données, et il est recommandé de pouvoir donner le coût d'exécution de chaque méthode dans le cadre d'une implémentation particulière.



L'utilisateur d'une structure de données n'a besoin de connaître que son interface, et nullement les détails de son implémentation.

Exercice n°3

L'interface minimale des files est `creer_file_vide`, `est_vide`, `enfiler` et `defiler`.

- ❶ Proposer une implémentation fonctionnelle de cette interface.
- ❷ Proposer une implémentation en orienté objet de cette interface.

3 Regards sur les listes :

Vous connaissez la structure *liste* de Python mais...



Python abuse du terme liste qu'il utilise pour ce qui sont des *tableaux dynamiques* munis de méthodes d'accès typiques des listes.

Il s'agit en fait de *tableau* qui permet de stocker des éléments dans des zones contiguës de la *mémoire*. Dans d'autres langages, comme le C par exemple, la déclaration d'un tableau statique est beaucoup plus rigoureuse que celle des listes en Python :

```
1 int tab[20]; /* déclare un tableau de 20 entiers */
```



Lorsqu'un objet de type *list* est créé en Python, l'interpréteur réserve une taille en mémoire proportionnelle à une puissance de 2 nécessaire pour stocker le tableau.

Par exemple, un tableau de 5 éléments nécessite $8(2^3)$ cellules. Une alternative aux tableaux dynamiques : les listes chaînées

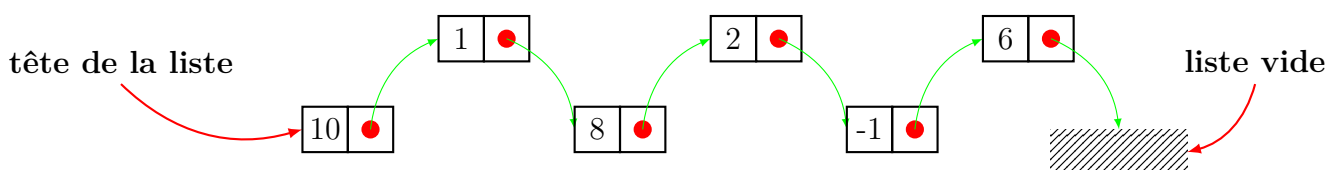
3.1 Listes chaînées :



Une *liste chaînée* est une structure linéaire qui n'a pas de dimension fixée à sa création.

Ses éléments de même type sont éparpillés dans la mémoire et reliés entre eux par des *pointeurs*. Sa dimension peut être modifiée selon la place disponible en mémoire. La liste est accessible uniquement par sa tête de liste c'est-à-dire son premier élément.

Par exemple pour la liste 10,1,8,2,-1,6, on aurait :





Chaque pointeur désigne une nouvelle liste. La construction est récursive!

L'élément de base est la *cellule* qui possède deux caractéristiques : la *valeur* et le *suisvant*. Une implémentation possible en POO donne :

```
1 class Cellule() :
2     """creating cell """
3     def __init__(self, valeur, suivant) :
4         self.valeur = valeur
5         self.suisvant = suivant
```

Une classe avec deux attributs mais attention :



Si *valeur* est une donnée de type *int*, *float* ou *str*, *suisvant* est une cellule, une *instance* de la classe!

Ainsi, l'implémentation de la liste ci-dessus donnerait :

```
1 cel = Cellule(6, None)
```

pour la dernière cellule qui pointe vers une liste vide (*None* ici...) et :

```
1 l = Cellule(10, Cellule(1, Cellule(8, Cellule(2, Cellule(-1, Cellule(6,
→ None))))))
```

pour l'ensemble de la liste.

Enrichissons la classe de méthodes, en particulier celle qui permet l'affichage de la chaîne :

```
1 def __str__(self):
2     if self.suisvant == None :
3         return f"{self.valeur} -> None"
4     else :
5         return f"{self.valeur} -> {str(self.suisvant)}"
```

Exercice n°4

Créer la méthode *longueur* de la classe qui retourne le nombre d'éléments de la liste chaînée.

Exercice n°5

Écrire la méthode *element_n* de la classe qui retourne l'élément situé au rang *n* de la liste.

3.2 Complexité des opérations :

On peut raisonnablement se poser des questions quant aux choix des implémentations possibles.



Quelle différence entre un tableau statique et un liste chaînée ?

La réponse est donnée dans la *complexité* des opérations de bases. Par exemple, quelle est la complexité de l'insertion d'un élément dans un tableau, une liste de n éléments ?

Considérons le tableau de quatre éléments : 200, 3, 175, 8900 que nous souhaitons stocker en mémoire. Contrairement au langage Python où la manipulation est aisée, la déclaration des tableaux dans le langage C donne :

```
1 int tab[4] //déclaration du tableau tab de 4 elements du même type int
```

Chaque entier est codé sur 16 bits(ou 32), soit 2 octets : il faut donc réserver en mémoire 4 cases de 2 octets pour stocker ce tableau. En supposant que la mémoire d'un ordinateur est un long, très long, maillage de cases alors l'*allocation* en mémoire du tableau peut s'illustrer ainsi :

	10			90	
a	200	3	175	8900	
		s		e	
	15				21

Les cases vides modélisent des emplacements mémoires vides ... L'adresse du tableau est A (valeur hexadécimale sur deux octets) et c'est aussi celle de `tab[0]`. Celle de `tab[i]` est alors $A+i*2...$

Naturellement, il est donc interdit de créer un tableau dont la taille dépend de la valeur d'une variable : c'est ce que nous appelons une *allocation dynamique*, et c'est un peu ce que fait *Python* dans la gestion transparente de ses listes !

Pour *insérer* un élément dans ce tableau, par exemple 6 au rang 2, il faut déplacer tous les éléments du tableau en espace contigüe disponible :

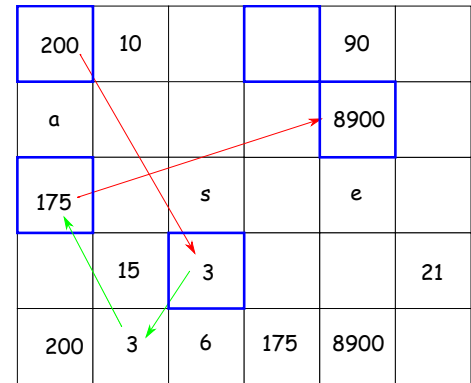
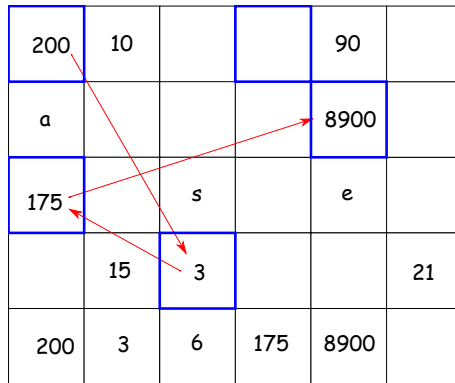
	10			90	
a	200	3	175	8900	
		s		e	
	15				21
	200	3	6	175	8900

En d'autres termes :



La complexité au pire de l'insertion dans un tableau statique de taille n nécessite n opérations.

Et pour les listes chaînées ? On peut déjà positionner les différents éléments n'importe où en mémoire :



L'insertion dans ce cas ne coûte que deux opérations, *quelque soit* la taille de la liste, du tableau.



La complexité au pire de l'insertion dans une liste chaînée taille n est constante.

Exercice n°6

Quelle est au pire, la complexité de l'accès à un élément dans les deux cas ?

4 Le coin des exercices :

Exercice n°7

On dispose des interfaces suivantes sur une Pile :

- ➔ `creer_pile_vide()` qui retourne une pile vide.
- ➔ `empiler(e, P)` empile l'élément e dans la pile P
- ➔ `depiler` retourne le sommet qui est enlevé de la pile P
- ➔ `est_vide(P)` qui retourne un booléen
- ➔ `est_pleine` qui retourne un booléen

- ① Quelles piles obtient-on par les instructions suivantes ?
`p1 = creer_pile_vide() , p2 = creer_pile_vide() empiler(4, p1),
empiler(1, p1),empiler(depiler(p1), p2), depiler(p1),
empiler(1,p2),empiler(7,p1) ?`

② Quelles instructions permettent de créer la pile :

4
5
1
2

③ Quelles instructions permettent de créer la pile **p** à partir des piles **p1** et

	p	p1	p2
	4	9	2
p2 ?	3	3	5
	2	1	4
	1	0	8

④ Créer la primitive **echange(p)** qui échange les deux premiers éléments de la pile **p**(les deux de dessus!) en utilisant seulement les primitives précédentes.

⑤ Créer la primitive **retourne(p)** qui échange les éléments de la pile **p**(le sommet en bas...)

Exercice n°8

Ci-dessous, une implémentation possible d'une liste chaînée en Python avec des listes python.

① Avec cette interface créer la liste 2,4,7,9.

② Qu'obtient-on par l'instruction : `valeur(suite(suite(L)))` où **L** est la liste précédente ?

③ Implémenter une interface `longliste` qui permet de connaître le nombre d'éléments de la liste.

```
1  #---  CONSTRUCTEURS  ---#
2  def liste_vide():
3      return []
4  def cellule(etiquette, liste):
5      return [etiquette, liste]
6  #---  SELECTEURS  ---#
7  def valeur(liste):
8      return liste[0]
9  def suite(liste):
10     return liste[1]
11 #---  PREDICAT  ---#
12 def est_vide(liste):
13     return liste == liste_vide()
```

Exercice n°9

À l'aide des fichiers python ClassFile.py et ClassPile.py ,

- ① Créez la pile $P = [1, 2, 4, 5, 6]$ et la file $F = [a, b, c, d]$ puis affichez-les.
- ② Comment créer une file avec deux Piles ?
- ③ Écrire l'implémentation correspondante en utilisant la classe Pile().

Exercice n°10

Il s'agit de proposer une implémentation récursive du type d'abstrait File dont les interfaces sont :

- ① constructeurs : Créer une file vide et Créer une File à partir d'une valeur et une file
 - ② Sélecteurs : Ajouter(push) une valeur à la file et Extraire(pop) une valeur de la file
 - ③ Prédicat : Teste si la file est vide ou non.
1. L'implémentation est réalisée par des listes Python. Quelle est la tête de la file ?
 2. Créer la file $F = [4, [5, [8, []]]]$.
 3. Ajouter 10 à cette file F
 4. Comment obtenir la tête de la file ? Comment obtenir le reste de la file ?
 5. Comment créer une copie d'une file ?

```
1 def file_vide():
2     return []
3 def file(valeur, file):
4     return [valeur, file]
5 def push(valeur, file):
6     if est_vide(file):
7         return [valeur, file_vide()]
8     else:
9         return [file[0], push(valeur, file[1])]
10 def pop(file):
11     return (file[0], file[1])
12 def est_vide(file):
13     return file == file_vide()
```

Exercice n°11

On peut à l'aide d'une pile, créer une fonction Python qui permet de tester l'appariement de parenthèses dans une expression du type $((4 - 5) + 7) * 4$. Pour cela, on parcourt élément par élément l'expression numérique (considérée comme une chaîne de caractère...) : si on rencontre une parenthèse ouvrante on l'empile dans une Pile et on dépile à chaque parenthèse fermante. Sinon on ne fait rien.

- ① À quelle condition portant sur la pile, l'expression est-elle bien parenthésée ?
- ② Écrire une fonction python qui prend en paramètres une chaîne de caractères représentant une expression numérique et qui renvoie un booléen correspondant au bon parenthésage ou non... On utilisera l'implémentation des piles ci-dessus.