

XVII – Structures de pile et de file

Introduction

Le stockage des données est un enjeu essentiel en informatique. Il faut les ranger de manière adéquate pour les traiter efficacement et y accéder facilement. L'objet informatique qui permet de stocker des données en mémoire s'appelle une **structure de données**. Elle est caractérisée par les opérations permises et le coût de celles-ci.

Nous connaissons pour l'instant les **listes**, les **tableaux** et les **dictionnaires**. Par exemple, on peut obtenir le nombre d'éléments, accéder à l'un d'eux, en enlever certains, en rajouter d'autres ou encore les parcourir un à un.

Dans ce chapitre, nous allons aborder deux nouvelles structures fréquemment utilisées en algorithmique : les **piles** et les **files**.

1/ Mise en œuvre d'une structure de pile

1.1) Définitions

Définition *On dit que des données sont organisées en **pile** lorsque l'on ne peut accéder qu'au dernier élément ajouté, appelé le **sommet** de la pile.*

Cela correspond bien à l'image d'une pile d'assiettes posées sur une table. Les seules opérations possibles pour éviter de les casser sont alors :

- enlever l'assiette au sommet de la pile ;
- ajouter une assiette au sommet de la pile.

Un peu de vocabulaire

- ❶ On parle de structure **dernier arrivé, premier sorti** (ou LIFO pour **last in, first out**).
- ❷ L'ajout d'un élément au sommet de la pile s'appelle **l'empilage**.
- ❸ La suppression d'un élément au sommet de la pile s'appelle **le dépilage**.

Exemple : lors de l'exécution d'une fonction récursive, les appels successifs sont empilés jusqu'à ce que l'on arrive à un **cas terminal**, puis dépilés successivement.

1.2) Opérations sur les piles

Avant de passer à la réalisation d'une structure de pile, il faut définir l'ensemble des opérations qui la caractérise. Les trois principales sont réalisées par les fonctions suivantes :

- `creer_pile()` renvoie une nouvelle pile initialement vide ;
- `depiler(p)` dépile et renvoie le sommet de la pile `p` ;
- `empiler(p,v)` empile la valeur `v` sur la pile `p`.

Les deux dernières fonctions modifient le contenu de la pile qui est passée en argument, contrairement aux fonctions suivantes :

- `taille(p)` renvoie la longueur de la pile `p` ;
- `est_vide(p)` indique si la pile `p` est vide ;
- `sommet(p)` renvoie le sommet de la pile `p` sans l'en enlever.

1.3) Implémentation d'une pile de capacité infinie

On peut implémenter cette structure de données en utilisant des listes ainsi que les fonctions `append()` et `pop()`, qui permettent d'ajouter ou de retrancher un élément en temps constant à une liste. Il faudra juste faire attention au cas où la liste est vide.

Exemple : voici comment créer les six fonctions ci-dessus en Python.

<pre>def creer_pile(): return [] def empiler(p,v): p.append(v) def taille(p): return len(p) def est_vide(p): return taille(p)==0</pre>	<pre>def depiler(p): if not est_vide(p): return p.pop() def sommet(p): if not est_vide(p): s=depiler(p) empiler(p,s) return s</pre>
---	--

1.4) Complexité des opérations

Les listes sont stockées dans un espace mémoire, qui se remplit lorsque l'on ajoute des éléments via la commande `append()`. Quand la place manque, Python crée une liste deux fois plus longue pour y recopier les éléments de la première (temps linéaire).

De même, si l'on enchaîne des `pop()` et que la liste n'occupe plus que le quart de la capacité allouée, on réduit sa taille de moitié en temps linéaire.

Sinon, les ajouts et extractions se font en temps constant.

Au final, une suite de n `append()` ou n `pop()` se fait en temps en $\mathcal{O}(n)$, et l'on dit que chaque opération a une **complexité amortie** constante. Pour éviter de rentrer dans les détails, on dira que toutes les opérations sur les piles de capacité infinie se font en temps constant. C'est aussi le cas de la fonction `len`, que l'on se permet d'utiliser pour accéder à la taille.

2/ Applications

2.1) Tri d'une liste de nombres

On dispose d'une pile p de nombres que l'on désire ordonner. Pour cela, on va utiliser deux piles p_1 et p_2 qui vérifient à chaque étape les propriétés suivantes :

- ❶ les éléments de p_1 sont empilés dans l'ordre décroissant ;
- ❷ ceux de p_2 sont empilés dans l'ordre croissant ;
- ❸ le sommet de p_1 est strictement supérieur au sommet de p_2 .

L'idée consiste à essayer de répartir les éléments de p entre ces deux piles en respectant les trois propriétés ci-dessus. Lorsque la pile p est vide, il suffit alors de dépiler les éléments de p_1 et de les empiler sur p_2 pour obtenir une pile triée dans l'ordre croissant. Pour obtenir une pile triée dans l'ordre décroissant, il suffira simplement de dépiler p_2 sur p_1 .

Exemple : faire ceci à la main avec $p = [5, 9, 3, 6, 4, 8, 1, 7] \dots$ en évitant de lire la suite !

L'exemple ci-dessus permet de trouver la démarche à adopter.

- On dépile d'abord p deux fois et on place le plus grand élément sur p_1 , l'autre sur p_2 .
- Ensuite, tant que la pile p n'est pas vide, on compare son sommet s aux sommets s_1 et s_2 des piles p_1 et p_2 .
 - Si s est égal à s_1 ou s_2 , on l'empile sur la pile correspondante. Ceci permet d'éviter un blocage.
 - Si $s_2 < s < s_1$, on peut empiler s sur n'importe laquelle des deux piles. On prendra la plus petite des deux.
 - Si $s > s_1$, on dépile p_1 sur p_2 jusqu'à ce que son sommet devienne supérieur ou égal à s ou qu'elle soit vide, puis on empile s sur p_1 .
 - Si $s < s_2$, on procède de manière analogue en dépilant p_2 sur p_1 .
- Lorsque la pile p est vide, il ne reste plus qu'à dépiler p_1 sur p_2 , ou l'inverse.

Remarque : la programmation de cet algorithme sera faite en exercice.

2.2) Parenthésage d'une expression

Le problème est le suivant : étant donnée une expression mathématique écrite sous forme d'une chaîne de caractères s , on veut déterminer si elle est bien parenthésée¹. Ceci signifie qu'elle contient autant de parenthèses ouvrantes et fermantes, qui de plus sont bien placées : quand on parcourt la chaîne de gauche à droite, à tout moment, on a rencontré au moins autant de parenthèses ouvrantes que de parenthèses fermantes.

Exemples :

- Les expressions "(Ba(ba)r)" et "(Co)(nan)" sont bien parenthésées.
- Les expressions ")Céleste(" et "(Cro)m)" ne le sont pas.

1. c'est un problème douloureux auquel vos mésaventures Pythoniques ont dû vous sensibiliser !

Tant qu'à faire, on veut aussi déterminer les indices des parenthèses fautives éventuelles. Voici comment procéder :

- On crée une pile vide *p*.
- On parcourt ensuite la chaîne *s* à l'aide d'une boucle *for*.
 - Si l'on trouve une parenthèse ouvrante, on empile son indice dans *p*.
 - Si l'on trouve une parenthèse fermante, on dépile *p* si elle n'est pas vide. Sinon, *s* est mal parenthésée et l'on affiche l'indice de cette parenthèse fermante en trop.
- À la fin de la boucle, la pile *p* doit être vide. Si ce n'est pas le cas, *s* est mal parenthésée et la pile *p* contient les indices des parenthèses ouvrantes en trop.

Exemple : voici une fonction `parentheses(s)` qui renvoie `True` si *s* est bien parenthésée et `False` sinon, en affichant dans ce cas les indices des parenthèses en trop.

```
def parentheses(s):
    p=creer_pile()

    for i in range(len(s)):
        if s[i]=='(':
            empiler(p,i)
        if s[i]==')':
            if est_vide(p):
                print("Parenthèse fermante en trop d'indice",i)
                return False
            else:
                depiler(p)

    if est_vide(p):
        return True
    else:
        for i in p:
            print("Parenthèse ouvrante en trop d'indice",i)
        return False
```

2.3) Évaluation d'une expression en notation polonaise inverse (NPI)

Dans cette notation, les opérateurs sont placés après le(s) opérande(s). Par exemple :

Notation standard	2+3	log(3)	(2+3)*5
NPI	2 3 +	3 log	2 3 + 5 *

On fera la distinction entre les *opérateurs unaires*, qui prennent un seul argument comme les fonctions, et les *opérateurs binaires* qui en prennent deux comme les opérations algébriques de base. Avec cette notation, les parenthèses deviennent inutiles puisque les opérations sont effectuées dans l'ordre où elles apparaissent. Certaines calculatrices fonctionnent encore selon ce principe.

On va représenter les expressions en NPI à l'aide de listes contenant des nombres (ou des noms de variables) et des chaînes de caractères pour les opérateurs. Pour les évaluer, on utilise une pile initialement vide et l'on parcourt les éléments de la liste de gauche à droite :

- Si l'élément est un nombre ou une variable, on l'empile.
- Si c'est un opérateur unaire, on dépile le sommet de la pile, on lui applique l'opérateur et on en empile le résultat.
- Si c'est un opérateur binaire, on dépile les deux premiers éléments de la pile, on leur applique l'opérateur et on empile le résultat.

A l'issue de la boucle, la pile contient uniquement la valeur de l'expression.

Exemple : voici comment on peut programmer ceci. Les opérateurs étant des chaînes de caractères, il faudra créer une chaîne contenant l'expression à évaluer, puis lui appliquer la fonction eval.

```
def eval_npi(L):
    p=creer_pile()
    for e in L:
        if type(e)==str:
            if e in ('+', '-', '*', '**', '/', '//', '%'):
                b=depiler(p)
                a=depiler(p)
                res=eval(str(a)+e+str(b))
                empiler(p, res)
            else:
                a=depiler(p)
                res=eval(e+'('+str(a)+')')
                empiler(p, res)
        else:
            empiler(p, e)
    return depiler(p)
```

3/ La structure de file

3.1) Définition

Définition On dit que des données sont organisées en **file** lorsque l'on ne peut accéder qu'au plus ancien élément ajouté, appelé la **tête** de la file.

Cela correspond bien au principe d'une file d'attente à un guichet :

- les nouveaux arrivants viennent s'ajouter en queue de file ;
- dès que le guichet est disponible, la personne en tête de file y va.

On utilise les files pour stocker des données qui doivent être traitées selon leur ordre d'arrivée.

Un peu de vocabulaire

- ❶ On parle de structure **premier arrivé, premier sorti** (ou FIFO pour **first in, first out**).
- ❷ L'ajout d'un élément en queue de file s'appelle **l'enfilage**.
- ❸ La suppression de l'élément en tête de file s'appelle **le défilage**.

3.2) Opérations sur les files

On retrouve les mêmes opérations de base que sur les piles :

- `creer_file()` renvoie une nouvelle file initialement vide ;
- `defiler(f)` défile et renvoie la tête de la file `f` ;
- `enfiler(f,v)` enfile la valeur `v` à la fin de la file `f` .

Les deux dernières fonctions modifient le contenu de la file qui est passée en argument, contrairement aux fonctions suivantes :

- `taille(f)` renvoie la longueur de la file `f` ;
- `est_vide(f)` indique si la file `f` est vide ;
- `tete(f)` renvoie la tête de la file `f` sans l'en enlever.

3.3) Implémentation d'une file de capacité infinie

On ne peut pas utiliser les listes, car l'extraction d'un élément à gauche est faite en temps linéaire. On va se servir des objets de type `deque`, qui sont des **double ended queues** ou **listes doublement chaînées**. On les crée avec la fonction `deque()` de la bibliothèque `collections`. Ils permettent de gérer les piles et les files, car ils disposent de méthodes pour ajouter ou retirer en temps constant des éléments aux deux extrémités.

<code>deque()</code>	crée une file vide.
<code>deque(L)</code>	crée une file à partir des éléments de la liste <code>L</code> .
<code>len(d)</code>	renvoie la longueur de la deque <code>d</code> .
<code>d.append(x)</code> / <code>d.appendleft(x)</code>	ajoute <code>x</code> à la fin / au début de <code>d</code> .
<code>d.pop()</code> / <code>d.popleft()</code>	supprime et renvoie la queue / la tête de <code>d</code> .

Remarque : on peut encore accéder aux éléments d'une deque grâce à leur indices.

Il suffit d'adapter les codes vus dans la section 2 pour écrire en Python les six fonctions `creer_file()`, `defiler(f)`, `enfiler(f,v)`, `taille(f)`, `est_vide(f)` et `tete(f)`.

Exercices

Exercice 1.

On suppose que l'on dispose d'une structure de pile avec les opérations fondamentales bien implémentées. Donner l'état de la pile p à l'issue de chacune des opérations suivantes :

- $p = \text{creer\_pile}()$
- $\text{empiler}(p, A)$
- $\text{empiler}(p, B)$
- $\text{empiler}(p, CF)$
- $\text{depiler}(p)$

Exercice 2.

Écrire une fonction $\text{echangisme}(p)$ qui échange les deux éléments situés au sommet d'une pile p de taille au moins égale à 2.

Exercice 3.

Écrire une fonction $\text{virer\_troisieme}(p)$ qui dépile et renvoie le troisième élément d'une pile p de taille au moins égale à 3, en laissant les autres éléments à leur place.

Exercice 4.

Écrire une fonction $\text{lecture}(n, p)$ qui lit le n^{e} élément d'une pile p s'il existe.

On s'assurera que la pile n'est pas modifiée en sortie.

Exercice 5.

Écrire une fonction $\text{tri}(p)$ qui trie les nombres de la pile p dans l'ordre croissant.

Indication : on utilisera l'algorithme vu en cours.

Exercice 6.

Programmer les fonctions sommet et taille uniquement à l'aide des instructions empiler , depiler et est_vide . Ainsi, elles seront indépendantes de la réalisation de pile choisie.

Exercice 7.

Adapter la fonction $\text{parentheses}(s)$ afin qu'elle s'occupe également des crochets et des accolades. Attention, le programme est un peu plus subtil : il faut s'assurer que le symbole fermant rencontré est du même type que le dernier symbole ouvrant rencontré. Du coup, de nouveaux messages d'erreur vont apparaître !

Indication : on va utiliser une seconde pile q dans laquelle on stocke le symbole fermant attendu dès que l'on rencontre un symbole ouvrant. De plus, on poursuivra l'examen de la chaîne même après la détection d'une erreur.

Exercice 8.

Écrire une fonction `destituer(p)` qui prend le sommet d'une pile non vide `p` et le place au fond de la pile, en conservant l'ordre des autres éléments.

Exercice 9.

Écrire une fonction `vice_et_versa(p)` qui prend une pile `p` en argument et renvoie une autre pile constituée des mêmes éléments placés dans l'ordre inverse.

Voici maintenant trois fonctions utiles quand on cherche à simuler une partie de cartes

Exercice 10.

Écrire une fonction `couper(p)` qui prend une pile `p` et la coupe en enlevant de son sommet un nombre aléatoire d'éléments, qui sont renvoyés dans une seconde pile.

Exemple : si la pile initiale est [1, 2, 3, 4, 5] et que l'on tire deux éléments, elle ne contient plus que [1, 2, 3] et l'on renvoie la pile [4, 5].

Exercice 11.

Modifier la fonction précédente pour écrire une fonction `couper_grouper(p)` qui coupe la pile `p` puis regroupe les paquets obtenus en plaçant le paquet enlevé en-dessous.

Exemple : si la pile initiale est [1, 2, 3, 4, 5] et que l'on tire les deux éléments du dessus en coupant, on renvoie la pile [4, 5, 1, 2, 3].

Exercice 12.

Écrire une fonction `melanger(p1,p2)` qui prend en argument deux piles et qui mélange leurs éléments dans une troisième pile de la façon suivante : tant qu'au moins l'une des deux piles de départ n'est pas vide, on dépile aléatoirement le sommet de l'une d'entre elles et on l'empile sur la pile résultat.

Exemple : un mélange possible de $p1=[1, 2, 3]$ et $p2=[5, 4]$ est $p=[3, 2, 4, 1, 5]$.

On finit avec quelques exercices sur les files

Exercice 13.

Écrire une fonction `copie(f)` qui renvoie une copie d'une file `f`.

Exercice 14.

Écrire une fonction `permutation(f,n)` qui effectue `n` permutations circulaires dans le sens direct sur les éléments d'une file `f`.

Exercice 15.

Écrire une fonction `inversion(f)` qui inverse l'ordre des éléments de la file `f`.
Il pourra être intéressant d'utiliser une pile pour les calculs intermédiaires.