Piles et files

Activité – Terminale NSI

① Une pile d'assiettes, une file d'attente

Exercice 1 *Deux discipline d'attente*

Pour chaque situation décrite, indiquer s'il s'agit d'une **pile** (« dernier arrivé, premier servi », LIFO : *Last In First Out*) ou d'une **file** (« premier arrivé, premier servi », FIFO : *First In First Out*).

1. Une pile d'assiettes propres dans une cantine.
2. La file devant la caisse d'un supermarché.
3. L'historique de navigation d'un navigateur (bouton « retour »).
4. La file d'impression d'une imprimante.
5. Les annulations successives (`Ctrl+Z`) dans un éditeur de texte.
6. Les messages d'une messagerie instantanée à traiter dans l'ordre de réception.

Remarque 1. La distinction **LIFO** ou **FIFO** est ce qui caractérise pile et file. Toutes deux ont les mêmes *opérations* (ajouter et retirer un élément), mais le choix de l'élément retiré n'est pas le même.

② La pile (LIFO)

Définition 1 **(Opérations sur une pile)** Une pile fournit quatre opérations :

— `empiler(p, x)` : ajouter `x` au sommet de la pile ;

— `depiler(p)` : retirer et renvoyer l'élément du sommet ;

— `sommet(p)` : lire l'élément du sommet sans le retirer (parfois appelé *peek*) ;

— `est_vide(p)` : tester si la pile est vide.

Exercice 2 *Tracer une pile*

On part d'une pile vide `p`, et l'on effectue dans l'ordre les opérations :

```
empiler(p, 3)
empiler(p, 7)
empiler(p, 1)
depiler(p)            # renvoie ?
empiler(p, 5)
depiler(p)            # renvoie ?
sommet(p)             # renvoie ?
```

1. Compléter le tableau ci-dessous, indiquant l'état de la pile après chaque opération (sommet à droite). Préciser ce qui est renvoyé par les `depiler` et `sommet`.

Opération	**État de la pile (base → sommet)**	**Valeur renvoyée**
Initial	`[]`	—
`empiler(p, 3)`		
`empiler(p, 7)`		
`empiler(p, 1)`		
`depiler(p)`		
`empiler(p, 5)`		
`depiler(p)`		
`sommet(p)`		

2. Quel est l'élément qui se trouvera au sommet à la fin ?

Exercice 3 *Implémenter une pile en Python*

On utilise une liste Python pour représenter la pile : le sommet est l'élément à l'*indice -1* (la fin de la liste). Les méthodes `append` et `pop` de la liste correspondent alors directement à `empiler` et `depiler`.

1. Compléter les fonctions ci-dessous :

```
def pile_vide():
    return ............

def empiler(p, x):
    p.append(............)

def depiler(p):
    return p.pop(............)

def sommet(p):
    return p[............]

def est_vide(p):
    return ............ == 0
```

2. Pourquoi utiliser `p.pop()` (sans argument) plutôt que `p.pop(0)` ?

③ La file (FIFO)

Définition 2 **(Opérations sur une file)**

— `enfiler(f, x)` : ajouter `x` en queue de file ;

— `defiler(f)` : retirer et renvoyer l'élément de tête ;

— `tete(f)` : lire l'élément de tête sans le retirer ;

— `est_vide(f)` : tester si la file est vide.

Exercice 4 *Tracer une file*

On part d'une file vide `f` et l'on effectue :

```
enfiler(f, 3)
enfiler(f, 7)
enfiler(f, 1)
defiler(f)            # renvoie ?
enfiler(f, 5)
defiler(f)            # renvoie ?
tete(f)               # renvoie ?
```

Compléter le tableau :

Opération	État de la file (tête → queue)	Valeur renvoyée
Initial	`[]`	—
`enfiler(f, 3)`		
`enfiler(f, 7)`		
`enfiler(f, 1)`		
`defiler(f)`		
`enfiler(f, 5)`		
`defiler(f)`		
`tete(f)`		

Exercice 5 *Implémenter une file (version simple)*

Avec une liste Python, on enfile à la fin (`append`) et on défile au début (`pop(0)`) :

```
def file_vide():
    return ............

```

```python
def enfiler(f, x):
    f.append(............)

def defiler(f):
    return f.pop(............)

def tete(f):
    return f[............]

def est_vide(f):
    return ............ == 0
```

1. Compléter les pointillés.
2. Quel est l'inconvénient de `f.pop(0)` sur une grande file ?
3. En Python, on préfère le module `collections.deque` qui propose `popleft()` en temps $O(1)$. Pourquoi est-ce préférable ?

4 Application : vérifier un parenthésage

Exercice 6 *Le « vrai » test des parenthèses*

Une expression est **bien parenthésée** si chaque parenthèse ouvrante est suivie, à un moment, d'une parenthèse fermante correspondante. Par exemple :

— `((1+2)*3)` est bien parenthésée ;

— `((1+2)*3` ne l'est pas (il manque une fermante) ;

— `(1+2))*3(` ne l'est pas non plus.

L'idée avec une pile : on parcourt l'expression caractère par caractère. Pour chaque `(` on *empile* ; pour chaque `)` on *dépile*. À la fin, l'expression est bien parenthésée si et seulement si la pile est vide *et* qu'on n'a jamais essayé de dépiler une pile vide.

1. Compléter la fonction :

```python
def bien_parenthesee(expression):
    p = pile_vide()
    for c in expression:
        if c == "(":
            empiler(............)
        elif c == ")":
            if est_vide(p):
                return ............        # fermante orpheline
            depiler(............)
    return est_vide(............)
```

2. Tester sur `"((1+2)*3)"`, `"((1+2)*3"`, `"(1+2))*3("`.

5 Application : la rumeur (parcours en largeur)

Exercice 7 *Propager une rumeur*

Dans un réseau d'amis, une rumeur se propage de proche en proche. Chaque personne, dès qu'elle entend la rumeur, la transmet à tous ses amis qui ne l'ont pas encore entendue. On souhaite simuler la propagation depuis une personne donnée.

L'idée naturelle est d'utiliser une **file** : à chaque tour, on défile la première personne qui doit transmettre la rumeur, on note ses amis comme « au courant », et on les enfile à leur tour.

1. Compléter le pseudo-code :

```
def propager(amis, depart):
    """amis : dict prénom -> liste de prénoms ; renvoie l'ordre de propagation."""
    f = file_vide()
    enfiler(f, ............)
    deja_au_courant = [............]
    ordre = []
    while not est_vide(f):
        personne = defiler(............)
        ordre.append(............)
        for ami in amis[personne]:
            if ............ not in deja_au_courant:
                deja_au_courant.append(............)
                enfiler(............)
    return ordre
```

2. Sur le réseau ci-dessous, donner l'ordre de propagation depuis Léa. Indiquer à chaque étape l'état de la file et la liste `deja_au_courant`.

```
amis = {
    "Léa":  ["Ana", "Max", "Tom"],
    "Ana":  ["Hugo", "Léa"],
    "Max":  ["Enzo", "Léa"],
    "Tom":  ["Léa", "Sara"],
    "Enzo": ["Max"],
    "Sara": ["Jade", "Tom"],
    "Hugo": ["Ana", "Jade"],
    "Jade": ["Hugo", "Sara"],
}
```

3. Quel est le rapport entre cette propagation et l'algorithme *BFS* (parcours en largeur) sur les graphes ?

6 Ce qu'il faut retenir

Propriété 1 **(Pile vs file)**

— **Pile** (LIFO) : dernier arrivé, premier servi. Opérations : `empiler`, `depiler`, `sommet`, `est_vide`.

— **File** (FIFO) : premier arrivé, premier servi. Opérations : `enfiler`, `defiler`, `tete`, `est_vide`.

Propriété 2 **(Implémentation en Python)** Une liste Python suffit pour une pile : `append` et `pop` (sans argument) opèrent sur le sommet en temps $O(1)$.

Pour une file, `pop(0)` est en $O(n)$: en pratique, on utilise `collections.deque` (`append` + `popleft` en $O(1)$).

Propriété 3 **(Quand utiliser quoi ?)**

— **Pile** : parcours en profondeur, vérification de parenthésage, gestion d'historique (annulation), évaluation d'expressions.

— **File** : parcours en largeur (BFS), gestion d'attente (impressions, requêtes), simulation « premier arrivé, premier servi ».

Remarque 2. Pile et file sont des **types abstraits** : ce qui compte est l'*interface* (les opérations qu'elles offrent), pas l'implémentation. On peut donc choisir une représentation interne efficace (liste, liste chaînée, `deque`...) sans changer le code qui utilise la structure.