TD 22 : Piles et files

Opérations souhaitées sur les piles

Les opérations que l'on souhaite pour les piles dans ce TP sont :

- Une fonction `pile_vide()` qui renvoie une nouvelle pile, vide.
- Une fonction `pile_est_vide(p)` qui indique si la pile passée en argument est vide ou pas. Cette fonction doit renvoyer un booléen (`True` ou `False`).
- Une fonction `empiler(p, x)` qui rajoute un élément `x` dans la pile `p`.
- Une fonction `depiler(p)` qui supprime et renvoie une occurence de l'élément de `p` **le plus récemment ajouté** (parmi ceux encore présents).

Opérations souhaitées sur les files

Les opérations que l'on souhaite pour les piles dans ce TP sont :

- Une fonction `file_vide()` qui renvoie une nouvelle file, vide.
- Une fonction `file_est_vide(f)` qui indique si la file passée en argument est vide ou pas. Cette fonction doit renvoyer un booléen (`True` ou `False`).
- Une fonction `enfiler(f, x)` qui rajoute un élément `x` dans la file `f`.
- Une fonction `defiler(f)` qui supprime et renvoie une occurence de l'élément de `f` **le plus anciennement ajouté** (parmi ceux qui y sont encore).

Rappel : Les *Double Ended QUEues*

La fonction qui permet de construire des nouvelles files à deux bouts peut être importée avec la ligne python suivante :

```
from collections import deque
```

Une fois importée elle peut être utilisée pour créer une nouvelle file à deux bouts avec par exemple

```
d = deque()
```

L'objet `d` créé dispose alors des méthodes suivantes :

- `d.append(x)` rajoute un élément à droite
- `d.appendleft(x)` rajoute un élément à gauche
- `d.pop()` supprime et renvoie (une occurrence de) l'élément le plus à droite.
- `d.popleft()` supprime et renvoie (une occurrence de) l'élément le plus à gauche.

Exercice 1 (Implantation des piles et des files)

1. Implantez la structure de pile en utilisant des listes python. Dit autrement, écrivez les fonctions `pile_vide`, `pile_est_vide`, `empiler` et `depiler`, en utilisant des listes python pour représenter vos piles. Toutes ces fonctions doivent avoir une complexité en $\mathcal{O}(1)$.
2. Implantez la structure de file en utilisant des *Double Ended QUEues*. Dit autrement, écrivez les fonctions `file_vide`, `file_est_vide`, `enfiler` et `defiler` en utilisant des files à deux bouts pour représenter vos files.

Dans la suite du TP, on vous demande de n'interagir avec vos files et vos piles qu'à travers les fonctions que vous avez définies ici ; comme si vous ignoriez quel type a été utilisé pour les implanter.

Exercice 2 (Mots bien parenthésés)

On dit d'une chaîne de caractères `u` que c'est un *mot bien parenthésé* ou un *mot de Dyck* si elle vérifie l'une des trois conditions suivantes :

— soit `u` est le mot vide `''`
— soit `u` peut être obtenu comme le résultat d'une concaténation `'(' + v + ')'` où `v` est un mot bien parenthésé
— soit `u` peut être obtenu comme le résultat d'une concaténation `v1 + v2` où `v1` et `v2` sont des mots bien parenthésés.

Nous avons vu pendant le DS 5 du 14 janvier qu'une façon de vérifier si un mot est bien parenthésé est de le parcourir de gauche à droite en comptant le nombre de parenthèses ouvrantes et fermantes et de

— vérifier au fur et à mesure qu'il y a eu au moins autant de parenthèses ouvrantes que fermantes
— vérifier à la fin qu'il y a eu autant de parenthèses fermantes qu'ouvrantes

Ici on se propose de faire cette même vérification, mais **en procédant autrement**, en utilisant une pile :

— Quand on croise une parenthèse ouvrante, on l'empile
— Quand on croise une parenthèse fermante, on essaie de dépiler la parenthèse ouvrante correspondante (si on n'y arrive pas parce que la pile est vide, ça veut dire qu'on essaie de fermer une parenthèse qui n'a pas été ouverte).
— Une fois le parcours fini, il faut vérifier que la pile est vide (il ne reste pas de parenthèse ouvrante).

1. Écrire une fonction `bien_parenthese` qui prend en entrée une chaîne de caractères et renvoie `True` si c'est un mot bien parenthésé, et `False` sinon. **Votre fonction utilisera une pile**.
2. Écrire une fonction `portees_parentheses` qui prend en entrée un mot bien parenthésé et renvoie une liste de paires `(indice_debut, indice_fin)` indiquant où commence et où finit chaque parenthèse. Par exemple `portees_parentheses('(()())')` peut renvoyer `[(0, 5), (1, 2), (3, 4)]`. On n'exige pas d'ordre particulier entre les différentes paires de la sortie.
3. On souhaite désormais admettre plusieurs types de parenthèses : (), [], {}.
   — Qu'est-ce qui doit changer dans la définition de mot bien parenthésé ?
   — Qu'est-ce qui doit changer dans la fonction `bien_parenthese` ?

Exercice 3 (Écritures d'une expression algébrique)

Vous connaissez plusieurs opérateurs arithmétiques binaires, tels que l'addition $+$, la soustraction $-$ et la multiplication $\times$. Vous êtes habitué·es à écrire ces opérateurs **entre** les arguments auxquels vous les appliquez. Par exemple, pou additionner 18 et 24 vous écrivez $18+24$. C'est ce qu'on appelle la notation *infixe*. Un des désavantages de la notation infixe est qu'**on peut avoir besoin de parenthèses** pour indiquer sans ambigüité quel est le calcul réalisé. Par exemple, si vous voulez écrire $(1+2)\times(3-(4\times5))$ vous ne pouvez vous contentez d'écrire $1+2\times3-4\times5$ car cela pourrait tout aussi bien vouloir dire $1+((2\times3)-(4\times5))$, qui n'est pas ce que vous voulez.
Une autre façon de noter ces mêmes opérateurs est de placer l'opérateur **avant** ses opérandes. C'est ce qu'on appelle la notation *préfixe*. Par exemple, pour additionner 18 et 22, la notation préfixe correspondante est $+\ 18\ 22$. **La notation préfixe n'a pas besoin de parenthèses**. Par exemple, le calcul que l'on souhaitait faire plus haut s'écrit

$$\times\ +\ 1\ 2\ -\ 3\ \times\ 4\ 5$$

sans qu'il y aie la moindre ambigüité.
On peut aussi écrire les opérateurs après leurs opérandes. C'est alors la notation *suffixe*, aussi appelée notation *postfixe*. Par exemple :

$$1\ 2\ +\ 3\ 4\ 5\ \times\ -\ \times$$

La notation préfixe s'appelle également *notation polonaise*, en honneur au polonais Jan Łukasiewicz qui la proposa en 1924. La notation suffixe s'appelle également *notation polonaise inversée*.

1. À la main, écrire l'expression algébrique `(5-7)*3` en notation préfixe puis suffixe.
2. Écrire une fonction `eval_suffixe(expr)` qui évalue une expression en notation suffixe (aide ci-dessous). L'expression sera contenue dans une liste : par exemple l'expression $1\ 2\ +\ 3\ 4\ 5\ \times\ -\ \times$ sera représentée par la liste python `[1, 2, '+', 3, 4, 5, '*', '-', '*']`

Évaluation d'une expression en notation suffixe avec une pile :
- lire la liste de gauche à droite, et pour chaque élément :
- si c'est un nombre, on l'empile
- si c'est un opérateur, on dépile ses deux opérandes de la pile, on réalise l'opération et on empile le résultat
- À la fin, la pile ne contient plus qu'une donnée : le résultat.

Pour tester si un élément `x` est un nombre, on utilisera `isinstance(x, (int,float,complex))`.

Exercice 4 (Bonus : Comparaison expérimentale de la complexité)

1. Vérifier expérimentalement quelle est la complexité des opérations que vous avez défini sur vos files. (Comme pendant le TP 12 du 30 novembre).
2. Comparer cette complexité à celle d'implémenter les files avec des listes python.

Exercice 5 (Bonus : tri par file de priorité)

Dans cet exercice on considère **une nouvelle structure de données**, les *files de priorité*, décrites par les opérations suivantes :

- Une fonction `file_vide()` qui renvoie une nouvelle file, vide.
- Une fonction `file_est_vide(f)` qui indique si la file passée en argument est vide ou pas.
- Une fonction `enfiler(f, x, prio)` qui rajoute un élément `x` dans la pile `f` **en lui associant la priorité** `prio`.
- Une fonction `defiler(p)` qui supprime et renvoie une occurence de l'élément de `f` **avec la plus faible priorité** (parmi ceux qui y apparaissent encore).

1. Implantez des files de priorité. On ne se souciera pas pour l'instant de la complexité des opérations.
2. Codez une fonction `tri_file_prio(l)` qui prend en entrée une liste `l` et renvoie en sortie une permutation triée de `l`. Votre fonction se reposera sur une file de priorité.
3. En utilisant les résultats théoriques sur les tris par comparaison, en déduire quelque chose sur les meilleures complexités que vous pouvez atteindre pour vos opérations `enfiler` et `defiler`.
4. Quelle est la complexité de vos opérations ?