V - PLANCHE 2 Exercices

Les interfaces des structures de données abstraites `Pile` et `File` sont rappelées ci-dessous.

- Structure de données abstraite: `Pile`

  `creer_pile_vide:` `() → Pile` `creer_pile_vide()` renvoie une pile vide

  `est_vide:Pile → Booléen` `est_vide(pile)` renvoie `True` si pile est vide, `False` sinon

  `empiler:` `Pile, Élément →` `empiler(pile,element)` ajoute `element` à la pile `pile`

  `depiler:` `Pile → Élément` `depiler(pile)` renvoie l'élément au sommet de la pile en le retirant de la pile

- Structure de données abstraite: `File`

  `creer_file_vide:` `() → File` `creer_file_vide()` renvoie une file vide

  `est_vide:File → Booléen` `est_vide(file)` renvoie `True` si file est vide, `False` sinon

  `enfiler:` `File, Élément →` `enfiler(file,element)` ajoute `element` à la file `file`

  `defiler:` `File → Élément` `defiler(file)` renvoie l'élément au sommet de la file en le retirant de la file

On répondra aux questions en utilisant uniquement les fonctions données dans l'interface des type `Pile` et `File`. On dispose de plus de deux fonctions d'affichage `affiche_pile` et `affiche_pile` qui étant donné une pile `p` (resp. une file `f`) en proposent un affichage de la manière suivante :

Code python

```
p = creer_pile_vide()
empiler(p, 1)
empiler(p, 2)
affiche_pile(p)
f = creer_file_vide()
enfiler(f, 1)
enfiler(f, 2)
affiche_file(f)
```

Résultat

```
| 2 | -> sommet
| 1 |
-----

               ---
enfilement -> 2 1 -> défilement
               ---
```

1 **1.** Soit une pile `p` une pile composée des éléments suivants : 12, 14, 8, 7, 19 et 22 (le prochain élément à être dépilé est 22).

a. Écrire le code python permettant d'instancier une variable `p` correspondant à la pile de l'énoncé.

b. Que renvoie `depiler(p)` ? De quels éléments est maintenant composée `p` ?

c. On exécute `empiler(p, 42)`. De quels éléments est maintenant composée `p` ?

d. Écrire une fonction `sommet` qui prend en argument une pile `p` et qui en renvoie l'élément placé au sommet. **À la fin de l'exécution, la pile `p` doit être dans le même état qu'initialement.**

e. Après avoir appliqué `depiler(p)` une fois, quelle est la taille de la pile ?

f. Si on applique `depiler(p)` 5 fois de suite, que renvoie `est_pile_vide(p)` ?

2. Soit une file `f` composée des éléments suivants : 12, 14, 8, 7, 19 et 22 (le prochain élément à être défilé est 22).

a. Écrire le code python permettant d'instancier une variable `f` correspondant à la file de l'énoncé.

b. On effectue `enfiler(f, 42)`. De quels éléments est maintenant composée `f` ?

c. Que renvoie `defiler(f)` ? De quels éléments est maintenant composée `f` ?

d. Écrire une fonction `tete` qui prend en argument une file `f` et qui en renvoie l'élément placé en première position de la file. **À la fin de l'exécution, la file `f` doit être dans le même état qu'initialement.**

e. Si on applique `defiler(f)` 5 fois de suite, que renvoie `est_file_vide(f)` ?

2 Les éléments constituant les piles de cet exercice sont des entiers. On ne se souciera pas de restaurer l'état de la pile après exécution des fonctions demandées.

1. Écrire une fonction `tamis` qui étant donné une pile `p` renvoie deux piles `p_pair` et `p_impair` contenant respectivement les éléments pairs et impairs de la pile `p`.

Code python

```
def tamis(p):
    """ Pile -> Pile, Pile """
    pass
```

2. Écrire une fonction `maximum` qui étant donné une pile `p` non vide renvoie le plus grand élément de la pile.

Code python

```
def maximum(p):
    """ Pile -> int """
    pass
```

3. Écrire une fonction python `retourne` inversant l'ordre des éléments présents dans la pile `p`.

Code python

```
def retourne(p):
    """ Pile -> None"""
    pass
```

3 On dit qu'une chaîne de caractères constituée de parenthèses ouvrantes `(` et fermantes `)` est bien parenthésée lorsque chaque parenthèse ouvrante est associée à une unique fermante, et réciproquement. Par exemple la chaîne de caractères `()()()` est correctement parenthésée tandis que la chaîne de caractères `(()((` ne l'est pas.

1. a. La chaîne de caractères `(())()` est-elle correctement parenthésée ?

b. Donner toutes les chaînes de caractères de taille 6 correctement parenthésées (il y en a cinq).

2. a. Écrire une fonction `est_bien_parenthesee` qui étant donné une chaîne de caractère `chaine` renvoie `True` si et seulement si `chaine` est une expression bien parenthésée. On utilisera pour cela l'algorithme suivant :

- on initialise une pile vide `p`
- pour chaque symbole de la chaîne de caractères :
  - si le symbole est une parenthèse ouvrante, on l'empile dans `p`
  - si le symbole est une parenthèse fermante, on dépile `p`.
- l'expression est correctement parenthésée si et seulement si l'algorithme se termine correctement et la pile finale est vide.

Code python

```python
def est_bien_parenthesee(chaine):
    """ str -> bool """
    pass
```

Code python

```python
print(est_bien_parenthesee("()"))
print(est_bien_parenthesee("("))
```

Résultat

```
True
False
```

b. Appliquer votre algorithme aux deux expressions données en exemple dans l'énoncé. Détailler les états successsifs de la pile utilisée.

3. On applique un algorithme dit "générer et filtrer" afin d'obtenir la liste de toutes les expressions correctement parenthésées. Pour cela, on commence par générer la liste de toutes les expressions de taille `n` fixée constituées uniquement de symboles `(` et `)`. Puis, on filtre les résultats obtenus en ne gardant que les expressions correctement parenthésées.

a. Combien y a-t-il d'expressions de taille 4, constituées uniquement de symboles `(` et `)` ?

b. Soit $n \in \mathbb{N}$. Combien y a-t-il d'expressions de taille n, constituées uniquement de symboles `(` et `)` ?

c. Pour tout nombre $i \in \mathbb{N}$ s'écrivant sur `n` bits, on associe la chaîne de caractère dans laquelle tous les 0 de l'écriture binaire de i ont été remplacés par `'('` et tous les 1 par `')'`.

On admet que la fonction `int2strbin` prend en entrée deux nombres entier `i` et `n` et renvoie la chaîne de caractère correspondant à son écriture en base 2 sur `n` bits. On pourra l'utiliser sans justification.

Code python

```python
print(int2strbin(5, 6))
```

Résultat

```
000101
```

Écrire une fonction `expression` qui prend en entrée un entier `i` et un entier `n` et qui renvoie la chaîne de `n` caractères constituée uniquement de parenthèses correspondante.

Code python

```python
def expression(x, n):
    """ int, int -> str """
    pass
```

Code python

```python
print(expression(5, 6))
```

Résultat

```
((()()
```

d. Écrire une fonction `liste_bien_parenthesee` qui prend en argument un entier `n` et qui renvoie la liste de toutes les expressions de taille `n` correctement parenthésées.

On pourra utiliser les fonctions `est_bien_parenthesee` et `expression` écrite précédemment.

Code python

```python
print(liste_bien_parenthesee(2))
print(liste_bien_parenthesee(4))
```

Résultat

```
['()']
['(())', '()()']
```

4 L'écriture polonaise inverse des expressions arithmétiques place l'opérateur après ses opérandes. Cette notation ne nécessite aucune parenthèse ni aucune règle de priorité. Ainsi l'expresion polonaise inverse décrite par la chaîne de caractères `1 2 3 * + 4 *` désigne l'expression traditionnellement notée $(1+2\times 3)\times 4$. La valeur d'une telle expression peut être calculée facilement en utilisant une pile pour stocker les résultats intermédiaires. Pour cela, on observe un à un les éléments qui constituent l'expression et on effectue les actions suivantes :

- si on voit un nombre, on le place sur la pile;
- si on voit un opérateur binaire, on récupère les deux nombres au sommet de la pile, on leur applique l'opérateur, et on replace le résultat sur la pile.

Si l'expression était bien écrite, il y a bien toujours deux nombres sur la pile lorsque l'on voit un opérateur, et à la fin du processus il reste exactement un nombre sur la pile, qui est le résultat.

1. a. À quel expression écrite en notation polonaise inverse correspond l'expression traditionnellement notée $2+3\times(4+5/6)$?

b. Quel est le résulat de l'expression qui s'écrit en notation polonaise inverse `3 4 - 12 1 + *` ? Détailler les états intermédiaires de la pile de calcul.

2. On représente une expression par une chaîne de caractères. Compléter le code de la fonction `decoupe` qui prend en entrée une chaîne de caractère `e` et qui renvoie la file des éléments qui la constituent (dans leur ordre d'occurrence).

Code python

```
def decoupe(e):
    f = creer_file_vide()
    element_en_lecture = ''
    for c in e:
        if c in {'+', '-', '/', '*'}:
            enfiler(..., ...)
            element_en_lecture = ...
        elif ... and not element_en_lecture == '':
            ...
            ...
        else:
            ...
    return f
```

Code python

```
e = '3 4 - 12 1 + *'
f = decoupe(e)
affiche_file(f)
```

Résultat

```
                --------------------
 enfilement -> "*" "+" 1 12 "-" 4 3 -> défilement
                --------------------
```

3. Écrire le code d'une fonction `evalue` qui étant donné une file `f` dont les éléments sont des entiers et les chaînes `"+" "-" "*" "/"` renvoie le résultat du calcul écrit en notation polonaise inverse correspondant.

On renverra la valeur spéciale `None` si le calcul n'est pas écrit correctement.

Code python

```python
def evalue(f):
    """ File -> int """
    pass
```

Code python

```python
print(evalue(f))
```

Résultat

```
13
```

4. Écrire une fonction `calculatrice_npi` qui étant donné une chaîne de caractères `entree` qui décrit un calcul utilisant la notation polonaise inverse, renvoie le résultat de ce calcul.

Code python

```python
def calculatrice_npi(entree):
    """ str -> int """
    pass
```

Code python

```python
print(calculatrice_npi("1 2 3 * + 4 *"))
```

Résultat

```
28
```

5 On souhaite trier les éléments d'une pile `A`. On propose pour cela l'algorithme suivant :

- on utilise deux piles `B` et `C`, initialement vides ;
- tant que `A` n'est pas vide, on est dans l'un des cas suivants :
  - soit la pile `B` est vide ou bien l'élément au sommet de `A` est plus petit que l'élément au sommet de `B`.

    Dans ce cas on retire l'élément au sommet de `A` pour l'empiler au sommet de `B`. Si la pile `C` n'était pas vide, on en dépile les éléments les uns après les autres et on les empile dans `B`.
  - sinon on dépile l'élément au sommet de `B` et on l'empile dans `C`.
- la pile `B` contient les éléments de `A` triés (le minimum est au sommet).

1. Suivre le déroulé de l'algorithme avec la pile `A = [4, 3, 2, 5, 8, 2, 6, 9, 3]` (le sommet de la pile est 3).

2. À quel algorithme de tri cela vous fait-il penser ?

3. Implémenter cet algorithme en python (vous n'utiliserez que les fonctions de l'interface du type `Pile`).

6 **1.** On suppose dans cette question que `p` est la pile `[8, 5, 2, 4]` (le sommet de la pile est 4).

Quel sera le contenu de la pile `q` après exécution du code ci dessous ?

Code python

```
q = creer_pile_vide()
while not est_pile_vide(p):
    empiler(q, depiler(p))
```

2. On appelle *hauteur* d'une pile le nombre d'élément qui la composent. Écrire une fonction `hauteur_pile` qui prend comme argument une pile `p` et en renvoie sa hauteur. Après l'appel de cette fonction, la pile `p` doit avoir retrouvé son état d'origine.

Code python

```
print(hauteur_pile(q))
affiche_pile(q)
```

Résultat

```
4
| 8 | -> sommet
| 5 |
| 2 |
| 4 |
-----
```

3. Écrire une fonction `max_pile` qui prend pour argument une pile `p` et un entier `i`. Cette fonction renvoie la position `j` de l'élément maximum parmi les `i` premiers éléments au sommet de la pile `p`. Après l'appel de cette fonction, la pile `p` doit avoir retrouvé son état d'origine. La positiion du sommet de la pile est 1.

Code python

```
print(max_pile(q, 2))
affiche_pile(q)
```

Résultat

```
1
| 8 | -> sommet
| 5 |
| 2 |
| 4 |
-----
```

4. Écrire une fonction `retourner` qui prend en argument une pile `p` et un entier `j` et inverse l'ordre des `j` éléments au sommet de la pile `p`.

Remarque. Si on dépile tout `p` dans `q1`, les éléments de `q` apparaissent dans l'ordre inverse que dans `p`. Si on dépile tout `q1` dans `q2`, les éléments apparaissent dans `q2` dans le même ordre que dans `p`. Si on dépile `q2` dans `p`, les éléments de `p` sont dans l'ordre inverse par rapport à la situation initiale.

Code python

```
retourner(q, 3)
affiche_pile(q)
```

Résultat

```
| 2 | -> sommet
| 5 |
| 8 |
| 4 |
-----
```

5. On cherche à trier une pile d'entiers, avec la méthode dite du "tri crêpe". L'idée est de procéder à des retournements successifs de la pile, comme on le ferait avec une spatule.

Le principe est le suivant :

- On recherche le plus grand élément M parmi les n éléments qui la composent.
- On retourne le morceau de la pile compris entre le sommet de la pile et M. Celui-ci se retrouve donc au sommet de la pile.
- On retourne toute la pile. M est donc tout en bas de la pile, il est correctement placé.
- On réitère ce procédé avec $n-1$ éléments au sommet de la pile.

Écrire une fonction `tri_crepe` qui trie une pile `p` suivant cet algorithme.

7 On considère la file `f` suivante :

Résultat

```
                ---------------------------------------
enfilement -> "rouge" "vert" "jaune" "rouge" "jaune" -> défilement
                ---------------------------------------
```

1. Écrire le code python permettant d'instancier une variable `f` de type `File` correspondant à la file `f` de l'énoncé.

2. Quel sera le contenu de la pile `p` et de la file `f` après l'exécution du programme Python suivant?

Code python

```
p = creer_pile_vide()
while not(est_file_vide(f)):
    empiler(p, defiler(f))
```

3. Créer une fonction `taille_file` qui prend en paramètre une file `f` et qui renvoie le nombre d'éléments qu'elle contient. Après appel de cette fonction la file `f` doit avoir retrouvé son état d'origine.

Code python

```
def taille_file():
    """File -> int"""
    pass
```

Ainsi, si `f` est la file de la question 1, alors on doit avoir :

Code python

```

print(taille_file(f))
affiche_file(f)
```

Résultat

```
5
                ---------------------------------------
enfilement -> "rouge" "vert" "jaune" "rouge" "jaune" -> défilement
                ---------------------------------------
```

1. Écrire une fonction `former_pile` qui prend en paramètre une file `f` et qui renvoie une pile `p` contenant les mêmes éléments que la file.

Le premier élément sorti de la file devra se trouver au sommet de la pile; le deuxième élément sorti de la file devra se trouver juste en-dessous du sommet, etc.

Ainsi, si `f` est la file de la question 1, alors l'appel `former_pile(f)` va renvoyer la pile `p` ci-dessous :

Résultat

```
| "jaune" | -> sommet
| "rouge" |
| "jaune" |
|  "vert" |
| "rouge" |
-----------
```

2. Écrire une fonction `nb_elements` qui prend en paramètres une file `f` et un élément `elt` et qui renvoie le nombre de fois où `elt` est présent dans la file `f`. Après appel de cette fonction la file `f` doit avoir retrouvé son état d'origine.

Code python

```python
def nb_elements(f, elt):
    """ File, int -> int """
    pass
```

Ainsi, si `f` est la file de la question 1, alors on doit avoir :

Code python

```python

print(nb_elements(f, "jaune"))
affiche_file(f)
```

Résultat

```
2
                ---------------------------------------
enfilement -> "rouge" "vert" "jaune" "rouge" "jaune" -> défilement
                ---------------------------------------
```

3. Écrire une fonction `verifier_contenu` qui prend en paramètres une file `f` et trois entiers: `nb_rouge`, `nb_vert` et `nb_jaune`. Cette fonction renvoie `True` si `"rouge"` apparaît au plus `nb_rouge` fois dans la file `f` et si `"vert"` apparaît au plus `nb_vert` fois dans la file `f` et si `"jaune"` apparaît au plus `nb_jaune` fois dans la file `f`. Elle renvoie `False` dans tous les autres cas. On pourra utiliser les fonctions précédentes.