Structure de pile

1 Généralités

Une **pile informatique** (en anglais **stack**) est une structure de données basée sur le principe « le dernier arrivé est le premier sorti » (on caractérise une pile par l'acronyme **LIFO** pour **Last In, First Out**) ; dans cette structure, c'est toujours le dernier élément ajouté à la pile qui sera le premier disponible : on peut se souvenir du fonctionnement d'une pile par l'image de la **pile d'assiettes** : lorsque l'on empile des assiettes, on les récupère de la dernière à la première et donc dans l'ordre inverse.
On verra quel est l'intérêt de cette inversion des éléments dans certains algorithmes.

Remarque : 1 :
Il ne faut pas confondre les deux structures opposées de **pile** et de **file** pour laquelle on remplacera l'image de la pile d'assiettes par la **file d'attente** (chez le boulanger par exemple) : cette fois, le principe d'une file (**queue** en anglais) est « le premier arrivé est le premier sorti » et on caractérise une file par l'acronyme **FIFO** pour **First In, First Out**.

Exemples de piles LIFO :

— La fonction "annuler la frappe" d'un traitement de texte (activée par la combinaison de touches "CTRL + Z", mémorise les modifications apportées au texte dans une pile

— La mémorisation des pages Web visitées dans un navigateur internet

Définition. Une **pile** est une structure de donnée sur laquelle on autorise 4 opérations (appelées primitives de la structure) :

✎ `creer_pile()` : qui crée une pile vide

✎ `empiler(p,x)` : qui empile un élément x à la pile p, i.e. qui met l'élément x au sommet de la pile (fonction "push")

✎ `depiler(p)` : qui supprime de la pile p le dernier élément empilé (i.e. son sommet) et renvoie sa valeur (fonction "pop")

✎ `pile_vide(p)` : qui renvoie le booléen True si la pile p est vide et False sinon

On peut rajouter à ces primitives 2 autres fonctions (qui peuvent s'écrire à partir des précédentes)

✎ `sommet(p)` : qui renvoie le sommet de la pile p sans l'enlever de cette pile

✎ `taille(p)` : qui donne la taille de la pile

Remarque : 2 :
On utilisera dans ce cours les listes Python pour l'implantation de la structure de pile. Dans un autre langage ne disposant pas de la structure de liste, on peut utiliser toute structure de tableau à une dimension.

Exercice Ecrire ces fonctions en Python.

Remarque : 3 :
On a deux possibilités pour concevoir des piles.

✎ Une pile dont la taille maximale est prédéfinie : sur Python on peut créer une liste Python de cette taille maximale, de considérer que tous les éléments dont on aura pas besoin sont affectés de la valeur None. On peut même considérer que le premier terme de cette liste sera la taille de la pile p

✎ Une pile non bornée dont la taille maximale n'est pas prédéfinie : sur Python on peut créer une liste Python de cette taille maximale, de considérer que tous les éléments dont on aura pas besoin sont affectés de la valeur None. On peut même considérer que le premier terme de cette liste sera la taille de la pile p

2 Utilisation des listes comme des piles

La structure de pile implique la définition des deux opérations (qui se traduiront par des fonctions) de bases « **empiler** » et « **dépiler** » ; même si leur traduction anglaise sont *stack* et *unstack*, les termes consacrés en informatique sont « **push** » et « **pop** » :
la fonction **empiler** (**push**) doit permettre d'ajouter un élément (en bout de pile, on dit d'ailleurs « au sommet de la pile ») et la fonction **dépiler** (**pop**) permet de sortir et de récupérer le dernier élément de la pile (i.e. au sommet de la pile).

En fait, sans ajout de fonctionnalité, les méthodes des listes permettent très facilement de les utiliser comme des piles ;
pour **empiler** (i.e. ajouter un élément sur la pile), on utilise la méthode **append()** ;
pour **dépiler** (qui inclut le fait de récupérer l'objet au sommet de la pile), on utilise la méthode **pop()** (sans indicateur de position) :

```
pile = [1, 2, 3]
pile.append(4); pile.append(5)
print(pile)     # reponse [1, 2, 3, 4, 5]
pile.pop()
print(pile)     # reponse [1, 2, 3, 4]
pile.pop();pile.pop()
print(pile)     # reponse [1, 2]
```

Remarque : 4 :
On utilisera évidemment aussi, si besoin, les autres fonctionnalités liées aux listes dont la fonction **len** donnant la longueur d'une liste qui nous donnera donc la **hauteur** de la pile.
Pour implémenter une file, on peut utiliser la classe **collections.deque** qui a été conçue pour fournir des opérations d'ajouts et de retraits rapides aux deux extrémités :

```
from collections import deque
file = deque([1, 2,3])
print(file)             # affiche la file  deque([1, 2, 3])
file.append(4); file.append(5)  # 4 arrive puis 5
file.popleft(); file.popleft()  # 1 sort puis 2
print(file)           # affiche la file   deque([3, 4, 5])
```

3 exercices

Dans ces exercices, on ne pourra utiliser que les fonctions associées aux piles, `creer_pile`, `empiler`, `depiler` et `pile_vide`.

1. Ecrire, en anticipant les différents cas qui peuvent provoquer des erreurs, les fonctions suivantes :
   (a) `taille(p)` : renvoie la taille de la pile p sans modifier la pile. Quelle est la complexité en temps et en espace de cette fonction ?
   (b) `sommet(p)` : renvoie l'élément le plus haut de la pile p sans modifier ladite pile.
   (c) `vide_pile(p)` : vide la pile.
   (d) `intervertir_sommet(p)` : intervertit les deux éléments situés en tête de la pile. Quelle est la complexité en temps et en espace de cette fonction ?
   (e) `inverser(p)` : renvoie une autre pile constituée des éléments de la pile initiale mais dans l'ordre inverse. On s'autorisera à vider la pile de départ.
   In [1] : inverser([1,2,3,4])
   Out[1] : [4,3,2,1]
   (f) `lit_element(p,n)` : renvoie le n-ièeme élément de la pile sans modifier la pile.
   (g) `couper(p,n)` : coupe la pile après le n-ième élément et renvoie la partie haute dans une autre pile. (Par exemple pile([1,2,3,4,5],2), change la pile en [1,2,3] et renvoie une autre pile [4,5]) Quelle est la complexité en temps et en espace de cette fonc- tion
2. Implémenter un test de bon parenthésage `parenthese(chaine)` d'une chaine, indiquant par un message si le nombre et l'ordre des parenthèses ouvrantes et fermantes est respecté. On travaille ici qu'avec les parenthèses usuelles '(' et ')'.
   In [1] : parenthese('((a+b)*8-2)')
   Out[1] : 'la syntaxe est correcte'
   In [2] : parenthese('(a+b))(')
   Out[2] : 'la syntaxe n'est pas correcte'
3. **La notation polonaise inverse**
   La **notation polonaise inverse** (**NPI**) permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses.
   Par exemple l'expression « $1 + 6 \times 5$ » s'écrit « $1\ 6\ 5 \times +$ », l'expression « $(1 + 6) \times 5$ » s'écrit « $1\ 6 + 5 \times$ » et enfin l'expression « $(2 \times (4 + 6))/5$ » s'écrit « $2\ 4\ 6 + \times 5\ /$ ».
   (a) Ecrire en NPI les expressions suivantes :
   $$(1 + 6) \times (5 + 4 \times 3),\ ((1 + 2) \times 3 + 4) \times 5,\ (1 + 6) - (5 + 9 \times 4) \text{ et } 9 \times 5/(4 + 3).$$
   (b) Evaluer les expressions suivantes écrites en NPI :
   $$5\ 10+,\ 2\ 10\ 4 + -,\ 10\ 4 + 2 -\ \text{et}\ 4\ 3 + 3\ 2 + \times.$$
   (c) On cherche maintenant à écrire une fonction permettant d'évaluer une expression donnée en NPI. Pour cela, on suppose que l'expression est donnée sous la forme d'une liste. Par exemple, on représentera l'expression « $10\ 4 + 2-$ » par $[10, 4,' +', 2,' -']$. On effectue alors les actions suivantes :
   ✎ On parcourt la liste
   ✎ Si on rencontre un nombre, on le place dans une pile (vide au début)
   ✎ Si on recontre un opérateur (+, -, *, /), il faut alors dépiler deux éléments de la pile, effectuer l'opération et empiler le résultat.
   ✎ Une fois la liste parcourue, la pile ne contient plus qu'un élément qui est la valeur souhaitée. *On admettra que les expressions entrées en NPI sont correctes.*
   i. Tester cette procédure sur des exemples.

ii. Construire une liste de deux listes **op**

- ✎ **op[0]** valant ['+','-', '*', '/']
- ✎ **op[1]** valant la liste des fonctions correspondantes $[add, sou, fois, sur]$ où par exemple add peut-être définie par

```
def add(x,y):
  return (x + y)
```

iii. Ecrire une fonction **evalNPI** prenant en argument une liste représentant une expression écrite en NPI et renvoyant la valeur de cette expression.

4. **Jeu de cartes**

Importer le module `random` qui contient des commandes générant des nombres aléatoires.
Par exemple l'appel `randint(a,b)` où a et b sont des entiers, renvoie un entier choisi aléatoirement dans $[\![a, b]\!]$

(a) Créer une pile représentant une couleur de carte : $p = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$

(b) Écrire une fonction `couper(pile)` qui coupe une pile en deux piles. La coupure est aléatoire mais les deux piles devront être non vides (lorsque cela est possible). De la pile de départ ne resteront que les premiers termes (le nombre de termes restant est le nombre aléatoire choisi). La seconde pile présente les éléments restants dans l'ordre inverse. Par exemple : In [1] : couper([1,2,11,4,13,6,7])
Out[1] : ([1,2,11] , [7,6,13,4])

(c) Écrire une fonction `melange` qui prend en entrée un couple de deux piles p_1 et p_2, et qui renvoie une pile p formée du mélange suivant :

- ✎ tant qu'aucune pile n'est vide, on dépile alátoirement p_1 ou p_2 et on empile sur p
- ✎ si l'une des deux piles d'entrée est vide, on dépile les éléments restants de l'autre sur p
- ✎ On inverse p

In [1] : melange([1,2,3,4],[5,6,7])
Out[1] : [1,5,2,3,6,4,7]

(d) Tour de magie de Gilbreath
Créer une pile formée de n répétitions de k cartes.
Couper puis mélanger cette pile selon les procédures précédentes. Remarquer que l'on retrouve dans la pile résultat n répétitions successives des k cartes, éventuellement mélangées au sein d'une répétition. Exemple :
In [1] : Gilbreath([1,2,3,1,2,3,1,2,3,1,2,3])
Out[1] : [3,2,1,3,1,2,1,3,2,3,2,1]

(e) **Bataille**

i. Créer un jeu complet de 52 cartes (4 ensembles de 13 cartes). Le mélanger plusieurs fois. Deux joueurs découvrent alternativement le sommet du jeu. Le joueur dont la carte possède la valeur la plus forte marque un point.

ii. Implémenter un jeu de bataille prenant un jeu de cartes mélangées en entrée et retournant les scores des deux joueurs et proclamant le joueur gagnant