TP N°2 : Les Piles

Avant de commencer : Créez un dossier *TP_2* dans *« D :\votre_groupe\votre_nom_prenom »* dans lequel vous placerez tous vos scripts que vous créerez durant ce TP.

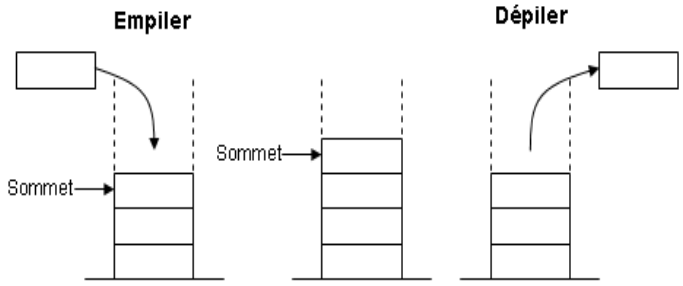
1 Définition

Une **pile** (**stack** pour les anglo-saxons) est une structure de données qui met en oeuvre le principe : **dernier entré, premier sorti** ou **LIFO** (**L**ast - **I**n - **F**irst - **O**ut).

L'image qu'on peut en donner est celle d'une pile d'assiettes : la dernière assiette placée dans la pile est au sommet de celle-ci. Ce sera bien sûr aussi la première assiette que l'on enlèvera de cette pile.
Cette structure permet de "stocker" provisoirement des éléments, en attendant de les utiliser plus tard.

Les seules opérations dont on a besoin avec cette structure sont INSERER et SUPPRIMER qu'on appelle ici EMPILER (**push** en anglais) et DEPILER (**pop** en anglais).

Une pile permet donc d'arranger et de récupérer les données les plus récentes.



Exemples de cas d'utilisations d'une pile :

- Gestion de l'historique dans un navigateur web : Les deux boutons « page suivante » et « page précédente» de votre navigateur, chacun utilise une pile. Les différentes pages visitées sont stockées dans une pile et l'on dépile lorsque l'on clique sur le bouton de retour en arrière.
- L'évaluation des expressions arithmétiques postfixées (notation polonaise inversée) utilise une pile.
- Exécution d'une fonction récursive. Chaque appel récursif est stocké dans la pile d'exécution jusqu'à atteindre un cas terminal, puis les fonctions sont dépilées.

Il y a beaucoup de façons de concevoir une pile : listes, tableaux,.... La plus simple consiste à utiliser un conteneur de type **list**.

2 Mise en oeuvre d'une pile avec une liste

Programmer le module `pile.py` contenant les fonctions suivantes :

1. `creer_pile()` : crée et retourne une pile vide non bornée (dont la taille maximale n'est pas définie, donc on peut y empiler autant d'éléments que l'on veut, dans la limite de la mémoire de l'ordinateur).
2. `pile_vide(p)` : retourne `True` si la pile p est vide et `False` sinon.
3. `sommet(p)` : renvoie l'élément du sommet de la pile p, sans le supprimer.
4. `taille(p)` : renvoie la taille de la pile p.
5. `empiler(p,x)` : ajoute l'élément x au sommet de la pile p.
6. `depiler(p)` : supprime le sommet de la pile p et le retourne.
   Que se passe-t-il quand on essaye de dépiler une pile vide?

3 Exercices

Exercice 1: conversion de base 10 en base 2

Écrire une fonction `conversion(n)` qui retourne la représentation en base 2 du nombre n représenté en base 10 à l'aide de piles.
Indication : La fonction python `bin(10)` retourne `'0b1010'`.

Exercice 2: Évaluation d'une expression arithmétique

Une des tâches fondamentales d'un interpréteur comme celui de Python (ou d'un compilateur dans un autre langage de programmation) est d'attribuer une valeur à une expression arithmétique. Une expression arithmétique est formé des 10 chiffres «0,1,...,9», des signes «+,−,×,/» et des parenthèses ouvrante «(» et fermante «)».
Par exemple : (3 + (4 × 5))/2 est une expression arithmétique dont la valeur est 11,5.

En général, l'utilisateur tape cette expression à l'aide de son clavier, ce qui génère une chaîne de caractères ch = "(3 + (4 ∗ 5))/2". Le problème est de traduire cette chaîne en un nombre, tout en gérant les priorités des opérations et les parenthèses : l'interpréteur doit se livrer pour cela à une **analyse syntaxique**.

Q 1. **Vérification des parenthèses dans une expression**
Écrire une fonction `verif_parenthese(expression)` qui reçoit une expression mathématique sous la forme d'une chaîne de caractères et renvoie **False** si l'expression n'est pas bien parenthésée et la liste des couples d'indices des paires de parenthèses dans le cas contraire.
Par exemple pour '((2+5)*4(+1*3' la fonction renverra **False** et pour '((2+5)*4+1)*3' elle renverra **[(0,10),(1,5)]**.
Indication : empiler les indices des parenthèses ouvrantes dans une pile.

Une même expression arithmétique a au moins trois représentations naturelles :

- La représentation **infixée** : C'est celle qu'on utilise tout le temps. Exemple : (3 + 2) × 5
- La représentation **préfixée** : où on place tous les opérateurs avant les opérandes.
  Exemples : 2 + 5 ⟶ +25 ; (3 + 2) × 5 ⟶ × + 325 ; 3 + (2 × 5) ⟶ +3 × 25
- On peut également placer les opérateurs après les opérandes : c'est la représentation **postfixée**. Dans ce cas 2 + 5 s'écrirait : 25+ et si on partait de (3 + 2) × 5 on aurait : 32 + 5×. Aucun besoin de parenthèses là non plus.

Q 2. Donner les deux autres formes des expressions suivantes et indiquer à chaque fois leurs valeurs :
a) 2 + (3 − (5 + 1) × 2) ; b) 2345 + −6 × /

L'évaluation d'une expression **postfixée** peut se faire de façon très simple, en une seule lecture de la gauche vers la droite à l'aide d'une pile qui stocke les résultats intermédiaires.

Sur l'exemple suivant, le caractère « ► » a été rajouté pour indiquer la limite de la partie déjà traitée de l'expression.
Partons de la chaîne postfixée : ch = "234 + −5 × "

► 2 3 4 + - 5 ×	pile P : []
2 ► 3 4 + - 5 ×	pile P : [2]
2 3 ► 4 + - 5 ×	pile P : [2,3]
2 3 4 ► + - 5 ×	pile P : [2,3,4]
2 3 4 + ► - 5 ×	pile P : [2,7]
2 3 4 + - ► 5 ×	pile P : [−5]
2 3 4 + - 5 ► ×	pile P : [−5,5]
2 3 4 + - 5 × ►	pile P : [−25] ⟶ **Résultat** =-25

Q 3. Écrire une fonction `calc_exp_arith(ch)` qui prend en paramètre une expression arithmétique (chaîne de caractères en notation postfixée) et qui renvoie sa valeur numérique.
Indication : L'expression contiendra uniquement des entiers positifs (pas de nombre négatifs, ni de nombres à virgule) et des signes des opérations séparés par des espaces.

Exercice 3: Permutations circulaires

Écrire une fonction `permut_circ(p,n)` qui reçoit en argument une pile `p` et un entier `n` et effectue sur la pile `n` permutations circulaires successives. Dans cet exercice, c'est la pile `p` elle-même qui sera modifiée.
Par exemple avec $n = 2$:
`p = [103, 2, 98, 11, 7]` ⟶ `p = [11, 7, 103, 2, 98]`

Exercice 4: Somme des éléments d'une pile

Écrire une fonction récursive `somme(p)` qui prend en paramètre une pile `p`, contenant éventuellement des sous-piles, de profondeur quelconques, dont tous les composants élémentaires sont des nombres, et calcule la somme de tous ses éléments.
Par exemple : `somme([ [[1,2], [3,4,5]], [6], [7,8], [9,[10]], 11])` ⟶ `66`