TP 8 – Piles
Année 2024-2025

Ce TP permet de créer des structures de données de type pile, de les implémenter à partir de listes chaînées simples (exercices 1 et 2) et d'écrire des algorithmes dans le cadre d'applications qui les utilisent. L'exercice 3 proposer d'implémenter les piles à partir de tableaux.

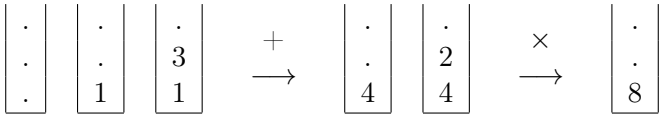
Exercice 1 – Implémentation du type abstrait Pile par une liste chaînée simple

Une pile est une séquence d'éléments dans laquelle les insertions et les suppressions se font à une seule extrémité. Les piles sont aussi appelées LIFO (Last-In-First-Out), c'est-à-dire dernier entré premier sorti.

1. Complétez la spécification et l'implémentation de la classe `Pile_List` (fichier `pile_list.py`).
2. Implémentez l'ensemble des méthodes définies dans la spécification `Pile_List` à partir d'une liste chaînée.
3. Testez dans le fichier `pile_list_tests.py` l'ensemble des méthodes implémentées en créant plusieurs exemples de piles.

Exercice 2 – Évaluation d'expressions arithmétiques à partir d'une pile

On se propose de simuler l'évaluation d'expressions arithmétiques. On s'appuie sur une notation postfixée, dite parfois "polonaise" inverse, utilisée par certaines calculatrices. On doit d'abord saisir les opérandes (les nombres), puis l'opérateur. Par exemple 2.3 (enter) 3.1 + provoque l'affichage de la valeur réelle 5.4 (2.3 + 3.1).
Dans cet exercice, les expressions seront composées d'entiers et des opérateurs `+`, `-`, `*` et `/`.
Donnez les résultats des expressions suivantes : `2 3 +, 1 3 + 2 *, 4 3 * 2 5 + +`
L'intérêt de cette notation est quelle permet d'éviter les parenthèses. Si on compare cette notation à la notation traditionnelle infixe, on remarque la nécessité dans la notation infixe (opérateurs entre les opérandes) de prendre en compte les priorités entre opérateurs :
1 + 2 * 3 : calculer d'abord la multiplication (seconde opération) puis l'addition (première opération),
1 * 2 + 3 : calculer d'abord la première opération puis la seconde,
alors que pour la notation postfixe, on ne tient compte que de l'ordre des opérations, par exemple :
1 2 3 * + (=7) et 1 2 * 3 + (=5)
Vous utiliserez une structure de pile d'entiers pour empiler les opérandes qui seront dépilés lorsque l'on doit leur appliquer une opération.
Exemple : Pour l'expression `(1, 3, +, 2, *)`, les états de la pile aux différentes étapes sont :



1. Implémentez la classe `EvalExp` et la méthode `evaluate(self, exp)` permettant d'évaluer une expression arithmétique `exp`. Les expressions arithmétiques pourront être entrées sous la forme d'une séquence de chaînes de caractères (en notation postfixée). Vous considérerez dans un premier temps que les expressions ne traitent que des chiffres, et qu'il n'y a pas de caractère séparateur dans la chaîne.

2. Testez l'évaluation d'expressions arithmétiques données en notation postfixée.

3. Pour étendre l'évaluation de l'expression arithmétique à des nombres, il est nécessaire de gérer les séparateurs entre nombres dans l'expression. Pour prendre en compte cela, vous transformerez l'expression `exp` dans laquelle les nombres sont séparés par des " ", sous la forme d'une liste de chaînes de caractères à partir de la méthode `split` :

   ```
   Par exemple : si     exp ="2 10 * 10 2 / - 3 +"
   self.exp.split(" ") retourne : ['2', '10', '*', '10', '2', '/', '-', '3', '+']
   ```

Exercice 3 – Exercice supplémentaire pour s'entraîner

On considère dans cet exercice une pile implémentée à partir d'un tableau de taille MAX.

1. Ré-implémentez la classe `PileTab` et toutes les méthodes de l'exercice 1. Vous testerez cette pile en supposant que l'on ne dépasse jamais la taille MAX du tableau.

2. Modifiez certaines des méthodes précédentes pour gérer la politique d'accroissement de la taille de la pile dans le cas où la pile est pleine et l'on souhaite empiler de nouveaux éléments.