■ ■ ■

Les *piles* et *files* sont des structures de données informatiques. Dans ce T.P., nous verrons leurs caractéristiques, une implémentation, ainsi que quelques applications.

Partie I : Définitions & Applications

Les *piles* sont des structures appelées L.I.F.O. (Last In First Out). Elles permettent par exemple de modéliser la pile d'assiettes propres dans un self-service, la pile des copies à corriger, la pile d'exécution d'un programme,... Par définition, une pile est un objet qui contient une collection ordonnée d'éléments. Sur cet objet, on ne peut effectuer que les opérations suivantes (appelées *primitives*) :

- Créer une pile vide ;
- Tester si la pile est vide ;
- Ajouter (*push*) un élément sur le dessus de la pile ;
- Retirer (*pop*) et lire l'élément sur le dessus de la pile.

En Python, l'utilisation des listes en tant que piles est prévue à l'aide des primitives `pop`, `append` et du test d'égalité à la liste vide :

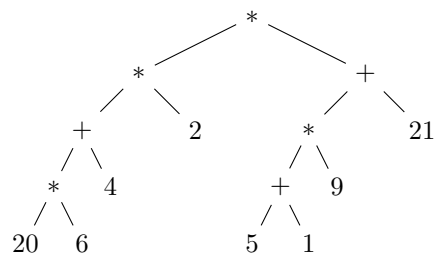
```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack.pop()
7
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Les piles peuvent être utilisées pour évaluer des expressions arithmétiques. Plus précisément, une *expression algébrique bien parenthésée* est une suite constituée d'entiers et des symboles $+$, $-$, $*$, $($, $)$ syntaxiquement correcte. Par exemple, $(2+3)*6$ est une expression algébrique mais $(2+)3*-6$ ne l'est pas. Nous allons voir comment un ordinateur peut traiter une telle expression pour renvoyer son évaluation. À chaque expression algébrique, correspondent une écriture arborescente et une écriture postfixe.

L'écriture préfixe, ou *notation polonaise*, a été introduite par le mathématicien J. LUKASIEWICZ en 1920. Elle consiste à écrire les opérateurs avant les opérandes. L'écriture *postfixe* (ou *notation polonaise inverse*) associée à une expression algébrique consiste à indiquer en premier les opérandes et ensuite l'opérateur. Par exemple, l'expression algébrique dont l'écriture linéaire est

$$[((20*6)+4)*2]*[((5+1)*9)+21]\,,$$

peut être écrite sous forme arborescente ou postfixée :



20 6 * 4 + 2 * 5 1 + 9 * 21 + *

On se convaincra que cette dernière notation se passe de parenthésage !

Dans un ordinateur, les expressions algébriques arrivent sous forme d'une pile au niveau de l'unité de calcul. Lors de l'évaluation d'une expression algébrique postfixe `exp`, on utilise une pile auxiliaire `aux` et on effectue récursivement les opérations suivantes :

- Si `exp` est vide, on retourne la valeur dépilée de `aux` ;
- Si on dépile un entier de `exp`, on l'empile dans `aux` ;
- Si on dépile un opérateur de `exp`, on effectue cette opération avec les deux premiers éléments de `aux` et on stocke le résultat dans `aux`.

1. Expérimenter cet algorithme sur l'expression donnée en exemple.

Les *files* sont des structures appelées F.I.F.O. (First In First Out). Elles permettent par exemple de modéliser la file des étudiants devant un self, la file des voitures en construction sur une chaîne,... Une file est un objet qui contient une collection ordonnée d'éléments sur lesquelles on peut effectuer les opérations suivantes (appelées *primitives*) :

- Créer une file vide ;
- Tester si la file est vide ;
- Ajouter un élément en queue de file ;
- Retirer et lire l'élément en tête de la file.

En Python, il est possible d'utiliser les listes comme files en utilisant le module `deque` et les primitives `append` et `popleft` :

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Partie II : Piles

Pour les questions suivantes, les seules opérations autorisées sont la création d'une pile vide, et les opérations `append`, `pop` et d'égalité à la liste vide.

2. Écrire une fonction `swap` qui échange les deux premiers éléments du haut de la pile. Cette fonction renverra `False` si la pile ne contient pas assez d'éléments.

3. On considère une pile qui contient uniquement des couples `(a, b)` où `a` ne peut prendre que deux valeurs : 0 (disons des assiettes rouges) et des 1 (disons des assiettes bleues). Écrire une fonction `ordonne` qui, en n'utilisant que des piles, renvoie la pile où les 0 sont au-dessus et les 1 en-dessous. L'ordre relatif des assiettes de même couleur devra être identique, après tri, à l'ordre initial.
Par exemple, la pile `[(1,'zebre'), (0, 'acacia'), (1,'chat'), (0, 'ginko')]` sera modifiée en `[(1,'zebre'), (1,'chat'), (0,'acacia'), (0,'ginko')]`.

4. Expressions algébriques. En Python, les éléments de l'écriture postfixée d'une expression algébrique seront stockés dans une pile dont les éléments seront sont soit des entiers, soit les chaînes de caractères `'+'` ou `'*'`. L'opérateur de soustraction `'-'` pourra être traité comme la succession d'un changement de signe puis d'une addition.

a) Donner les écritures arborescentes et postfixées de l'expression

$$[[(2+3)*(6+5)]+[3+(2*3)]]\,.$$

b) Écrire une fonction récursive `evaluer_aux` qui évalue les expressions algébriques postfixes. Cette expression prendra deux arguments : une pile `exp` représentant une expression algébrique postfixe, et une pile `aux` auxiliaire de telle sorte que l'appel `evaluer_aux(exp,[])` renvoie l'évaluation de l'expression `exp`. En déduire une fonction `evaluer` qui prend en argument une expression algébrique postfixe et renvoie son évaluation.
On pourra utiliser `isinstance(<var>, int)` *qui renvoie* `True` *si* `<var>` *est de type* `int` *et* `False` *sinon.*

c) Réécrire une fonction `evaluer_iter` qui a la même spécification que la fonction `evaluer` mais utilise la programmation itérative.

Partie III : Files

5. Les deux questions suivantes sont indépendantes.

a) Écrire une fonction `copie` qui, étant donnée une file, renvoie une copie de cette file, sans la modifier.

b) Écrire une fonction `permutations` qui, étant donnée une file `queue` et un entier `n`, effectue `n` permutations circulaires successives sur la file `queue`.
Par exemple, si `queue = [1, 2, 3, 4]`, l'appel `permutations(queue, 2)` modifiera `queue` de telle sorte qu'elle contienne `[3, 4, 1, 2]`

c) Reprendre la question précédente avec des piles à la place des files.

6. Les files peuvent être utilisées pour parcourir les arbres en largeur. La notion d'arbre est définie récursivement comme étant soit l'arbre vide, soit un nœud et des sous-arbres. Nous avons vu précédemment des arbres (dits *binaires*) dont chaque nœud possède au plus deux sous-arbres. Vous pourrez visualiser un arbre comme étant un arbre généalogique.
Parcourir un arbre en largeur consiste à lister la valeur des nœuds génération par génération. Par exemple, pour l'arbre représentant l'expression algébrique, on obtient

`*, *, + ,+ , 2, *, 21, *, 4, +, 9, 20, 6, 5, 1.`

Étant donné un arbre généalogique, ce parcours consiste à lister les individus par génération.

a) Proposer une implémentation des arbres en Python.

b) En utilisant votre implémentation des arbres, proposer un algorithme de parcours en largeur n'utilisant que des files.