TD 4 : Piles – Tris (hors-programme) en récursif

1 Piles

1.1 Implémentation d'une pile avec Python

Exercice 1. Implémenter les opérations élémentaires sur les piles

On rappelle le modèle choisi, à savoir qu'une pile sera représentée par une liste, le sommet de la pile correspondant au dernier élément de la liste.

- `creer_pile()` : renvoie une pile vide
- `est_vide(p)` : renvoie `True` si la pile `p` est vide, `False` sinon
- `empiler(p, v)` : ajoute la valeur `v` au sommet de la pile `p` (en anglais : fonction « push »)
- `depiler(p)` : supprime le sommet d'une pile `p` non vide (en anglais : fonction « pop »)

On pourra définir en complément une fonction qui affiche une pile, en colonne, avec son sommet qui sera affiché en premier.

Note : cette fonction sera « hors modèle » car on y traitera la pile comme la liste qu'elle est ; on s'autorisera d'autres manipulations que celles permises par les quatre fonctions du modèle.

Pour les exercices suivant sur les piles, la contrainte est de n'utiliser, pour créer, modifier ou manipuler une pile que les quatre fonctions précédentes.

Exercice 2. <exercice donné en classe>

On considère un jeu de n cartes, représenté par une pile dont les éléments sont les entiers de 1 à n.

Q2.1 Écrire une fonction `créer_jeu(n)` créant un jeu de n cartes (ordonné), sous la forme d'une pile (la carte la plus grande étant située à la base de la pile).

Q2.2. Écrire une fonction `retourne(p)` qui prend en argument une pile `p` et renvoie une pile composée des mêmes éléments mais dans l'ordre inverse.

Note : la pile donnée en argument sera vide à la fin de l'exécution de la fonction.

Q2.3. [Pour cette question on supposera que la pile comporte un nombre pair d'éléments, connu] Écrire une fonction `coupe(p, n)` qui prend en argument une pile, `p`, de hauteur paire, égale à `n`, et son nombre d'éléments, `n`, et renvoie deux nouvelles piles composées l'une de la moitié supérieure de la pile p, l'autre de la moitié inférieure de p (sans changer l'ordre des cartes).

Q2.4. Écrire une fonction `melange(p1, p2)` qui prend en argument deux piles, **supposées de même hauteur**, et qui renvoie une nouvelle pile dans laquelle on alterne les éléments de la pile 1 et de la pile 2. On demande que le sommet de la pile 1 se retrouve au sommet de la pile renvoyée.

Q2.5. Pour valider vos implémentations des fonctions précédentes, vérifiez, pour $p = 2, 3, 4, 5$, qu'un jeu de 2^p cartes, après p coupes suivies chacune d'un mélange, revient à son ordre initial.

Exercice 3. Implémentation de la structure de file

Le type abstrait de données « file d'attente » est une structure linéaire de stockage munie des primitives suivantes :

- `creer_file()` : revoyant une file vide ;
- `est_file_vide(f)` : renvoyant « Vrai » si la file `f` est vide, « Faux » sinon ;
- `ajouter(x, f)` : ajoute le « client » `x` à la fin de la file `f` ;
- `suivant(f)` : renvoie le client en tête de la file non vide `f` et le supprime de `f`.

Q3.1. Proposer une implémentation en Python où la file est stockée dans une liste. Évaluer la complexité des fonctions `ajouter` et `suivant`.

Q3.2. Pour obtenir une complexité en temps constant (ou temps constant amorti) des deux fonctions `ajouter` et `suivant`, on se propose de stocker une file à l'aide deux piles :

- la première contient le début de la file, le premier client de la file se trouvant au sommet ;
- la seconde contient la fin de la file, le dernier client entré se trouvant au sommet.

Donner une implémentation de la structure de données en Python selon ce modèle. Concrètement, la file `f` pourra être définie comme une liste de deux piles, `[p0, p1]`. Expliquer l'intérêt de ce modèle du point de vue de la complexité. Outre les primitives ci-dessus, on programmera l'affichage de la file.

2 Problèmes sur les piles

Exercice 4.

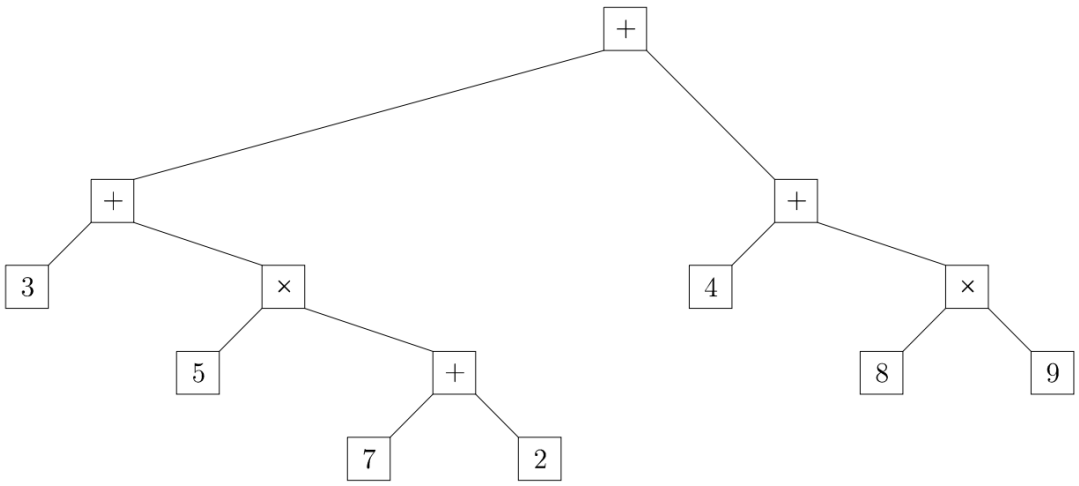
1) Représentation d'une expression arithmétique

Pour simplifier, nous nous limiterons ici à des enchaînements d'additions et de multiplications de deux nombres entiers. Considérons par exemple l'expression

$$E = (3 + (5 \times (7 + 2))) + (4 + (8 \times 9)) .$$

Cette écriture, dite *écriture infixée totalement parenthésée*, est allégée — dans l'usage courant — de quelques parenthèses, compte tenu de la convention de priorité de la multiplication sur l'addition et aussi du fait que l'on accepte les sommes de plusieurs nombres. Mais, en l'absence de priorités et si l'on se limite à des opérateurs *binaires*, les parenthèses ci-dessus sont nécessaires pour définir l'ordre des opérations.

On peut aussi représenter une telle expression par un arbre, chaque sous-arbre correspondant à une sous-expression entre parenthèses. L'expression E serait représentée par l'arbre suivant :



Une autre représentation courante est l'écriture *postfixée* (également appelée *notation polonaise inversée*, RPN en anglais, en hommage à son "inventeur" Jan Lukasiewicz, logicien et philosophe polonais). Dans cette écriture, les opérandes sont écrits l'un après l'autre, suivis de l'opérateur (l'écriture infixée $a + b$ devient $a\ b+$). Partant de la représentation par un arbre, on écrit (récursivement !) l'expression postfixée associée au sous-arbre gauche, suivie de l'expression postfixée associée au sous-arbre droit, suivie de l'opérateur à la racine de l'arbre (cela pour un opérateur *binaire*, *i.e.* à deux arguments…) ; on parle de *parcours postfixe* de l'arbre.
Pour l'arbre précédent, cela donne

$$3 \quad 5 \quad 7 \quad 2 \ + \ \times \ + \ 4 \quad 8 \quad 9 \ \times \ + \ +$$

2) Évaluation à l'aide d'une pile

Par sa structure même, l'écriture postfixée d'une expression permet une évaluation très simple à l'aide d'une pile (d'où son intérêt par rapport à l'écriture infixée avec parenthèses, qui nécessite une analyse syntaxique…).

L'algorithme est le suivant : partant d'une pile vide,on lit successivement les "termes" de l'expression postfixée (de gauche à droite) et

- si l'on trouve un nombre on l'empile
- si l'on trouve un opérateur unaire f (par exemple le *moins unaire* ou une fonction d'une variable), on remplace le sommet x de la pile par $f(x)$
- si l'on trouve un opérateur binaire ω (par exemple une opération comme $(x, y) \mapsto x+y$), on remplace les deux valeurs x, y au sommet de la pile par $\omega(x, y)$.

Cela suppose que les identificateurs des opérateurs unaires et binaires soient distincts (attention au symbole $-$!) et l'on peut facilement généraliser à des opérateurs d'arité supérieure à 2…

Dans ce qui suit, pour simplifier l'implémentation, on se limitera à des nombres entiers naturels et à l'opérateur unaire de passage à l'opposé (noté « - ») et aux opérateurs binaires d'addition et de multiplication (notés « + » et « * »).
On pourra tester si une chaîne représente un entier à l'aide de la méthode `.isdigit()`.

3. Pour évaluer une expression arithmétique postfixée à l'aide d'une pile, on se limite pour simplifier à des additions et multiplications sur des entiers.

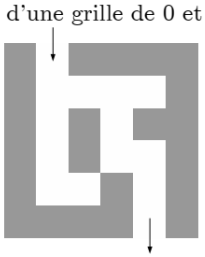
Écrire une unique fonction qui contrôle et évalue (à l'aide d'une pile !) une expression arithmétique postfixée fournie sous forme d'une chaîne de caractères. Ladite chaîne contiendra les différents symboles (nombres et opérateurs), séparés par des espaces.

Pour extraire les symboles, on peut parcourir la chaîne à la recherche des espaces, ou bien utiliser la méthode `split` de Python, qui — appliquée à une chaîne — renvoie la liste des constituants de ladite chaîne, constituants délimités par un ou des séparateurs que l'on peut préciser parmi les paramètres (cf. l'aide de Python). Mais les espaces font partie des séparateurs par défaut, ainsi `'3 4 +'.split()` renvoie `['3','4','+']`.

Pour convertir en nombre une chaîne représentant un nombre, il suffit de lui appliquer le *transtypage* : `int('7')` renvoie le nombre 7...

Exercice 5.

On s'intéresse au problème de la recherche d'une sortie dans un labyrinthe. On considérera des labyrinthes bâtis à partir d'une grille de 0 et de 1 comme ci-dessous :



0	1	0	0	0	0
0	1	1	1	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	1	0
0	0	0	0	1	0

Un labyrinthe est une grille de taille $n \times n$ où :

- les 0 représentent les murs et les 1 représentent les couloirs du labyrinthe ;
- l'entrée se fait par la case $(0, 1)$ et la sortie par la case $(n-1, n-2)$;
- le labyrinthe est entouré de murs, à l'exception de ces deux cases.

L'objectif est d'écrire un programme qui détermine s'il existe un chemin de l'entrée vers la sortie en se déplaçant vers le haut, le bas, la gauche ou la droite (mais pas en diagonale).

On pourra stocker le tableau `laby` sous forme d'une liste de listes, l'accès à la case `(i,j)` se faisant alors par `laby[i][j]`.

a) Écrire une fonction `casesVoisines(laby,i,j)` prenant en arguments un tableau `laby` et deux entiers `i` et `j`, et renvoyant la liste des cases adjacentes à la case `(i,j)` qui ne sont pas des murs.

Dans l'exemple ci-dessus, `casesVoisines(laby,1,1)` doit renvoyer `[(0,1),(1,2),(2,1)]`.

Attention à gérer les cases du bord !

b) *Parcours du labyrinthe*

On va explorer tous les couloirs de proche en proche jusqu'à trouver la sortie. Pour cela :

- on représente les cases par des couples d'indices (i, j) ;
- on stocke les cases à explorer dans une pile `p` ;
- on marque les cases visitées.

Algorithme :

- **Initialisation :** aucune case visitée, pile vide
- **Première étape :** on marque la case d'entrée, on la met dans la pile
- **Itération :** tant que la pile n'est pas vide, on dépile une case de la pile `p`, et l'on traite chacune des cases adjacentes qui n'est pas un mur
  - si elle est déjà marquée, on ne fait rien ;
  - si elle n'est pas marquée, on la marque et on l'empile dans `p`.
- **Fin :** les cases marquées sont exactement les cases accessibles depuis l'entrée.

Implémenter la fonction `parcours(laby)` prenant en argument le tableau `laby` représentant le labyrinthe, renvoyant un booléen indiquant s'il existe un chemin de l'entrée jusqu'à la sortie et plaçant 2 à la place de 1 dans les cases de `laby` accessibles depuis l'entrée.

Évaluer la complexité de cette fonction.

N.B. Pour afficher graphiquement le labyrinthe associé au tableau `laby`, on peut utiliser la commande `matshow` de `matplotlib`. On peut même choisir les couleurs associées aux valeurs entières présentes dans `laby` en définissant une *palette de couleurs* personnalisée grâce à la fonction `ListedColormap` du module `colors` de `matplotlib`. Par exemple :

```
plt.matshow(laby, cmap = colors.ListedColormap(['black', 'white', 'blue']))
```

Application du parcours de labyrinthe au problème de percolation en milieu aléatoire

La *percolation* est l'écoulement d'un liquide à travers un milieu poreux (eau à travers de la mouture de café...). L'objectif est de décider si l'eau peut traverser un milieu donné.

- **Modélisation :** on considère un milieu où les pores sont disposés aléatoirement. On le modélise par une grille dont chaque case est aléatoirement pleine ou vide. L'eau s'écoule en passant dans les cases vides.
- **Réduction au problème du labyrinthe :** étant donné un tableau rempli de 0 (cases pleines) et de 1 (cases vides) modélisant un milieu poreux `m`, on veut obtenir une instance `laby` du problème du labyrinthe telle que la fonction `parcours` appliquée à `laby` réponde au problème de la percolation pour `m`. Pour cela, il suffit de :
  - **encadrer le milieu :** le tour du milieu sera rempli de 0 ;
  - **ajouter une entrée :** la ligne d'indice 0 est bien remplie de 0 sauf un 1 sur la deuxième case ;
  - **ajouter une ligne d'écoulement à l'entrée :** la ligne d'indice 1 est remplie de 1 sauf les deux extrémités ;
  - **ajouter une ligne d'écoulement à la sortie :** la ligne d'indice -2 est remplie de 1 sauf les deux extrémités ;
  - **ajouter une sortie :** la ligne d'indice -1 est bien remplie de 0 sauf l'avant-dernière case.

a) *Construction du milieu poreux* : écrire une fonction `creerMilieu(n,p)` prenant en arguments un entier `n` et un flottant `p` de $[0,1]$, et renvoyant un tableau `milieuPoreux` de dimension `n`×`n` représentant le milieu poreux. Ce tableau sera rempli comme suit :
  - les deux premières lignes, les deux dernières lignes, la première et la dernière colonne seront comme expliqué précédemment ;
  - les autres cases seront chacunes vides avec probabilité p (on utilisera le test `random.random() < p` qui est vrai avec probabilité p).

b) *Affichage de l'écoulement* : écrire une fonction `afficherEcoulement(n,p)` qui crée un milieu en appelant `creerMilieu(n,p)` et qui, ensuite, affiche en noir les cases pleines, en bleu les cases vides où l'eau s'écoule et en blanc les autres.

c) *Estimation de la probabilité d'écoulement* : écrire une fonction `estimerProba(n)` réalisant une estimation numérique de la probabilité P que l'eau traverse le milieu en fonction de la probabilité p que chaque case soit vide, en faisant plusieurs itérations pour chaque valeur.

d) *Seuil critique* : on observe un effet de seuil, en-dessous d'une certaine valeur critique p_c, l'eau ne traverse presque jamais. Cet effet de seuil est d'autant plus marqué que n est grand. Écrire une fonction `valCritique(n,eps)` prenant en argument la taille n du milieu et un flottant $\varepsilon > 0$ et calculant une valeur approchée de p_c à ε près à l'aide d'un algorithme dichotomique.

e) *Illustration du seuil* : Tester la fonction `afficherEcoulement(n,p)` avec $n = 200$ et successivement $p = 0.95 * p_c$, $p = p_c$ et $p = 1.05 * p_c$.