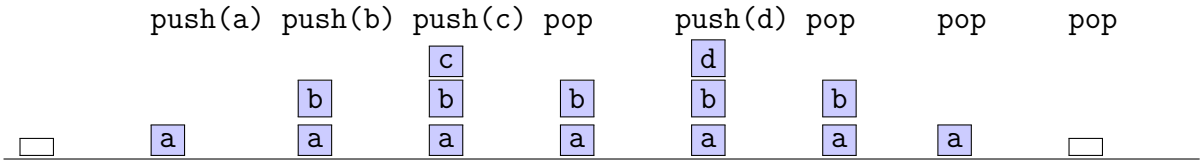
Face aux piles !

I Introduction aux piles

Une pile est une structure de donnée alternative aux listes, mieux adaptée que les listes pour traiter certains types de problèmes. Une pile est caractérisée par le fait que, comme pour une pile d'assiettes :

- la taille maximale d'une pile `P` que l'on forme n'a pas à être spécifiée dès le départ ;
- les éléments sont ordonnés, et seul le sommet de la pile est accessible :
  - soit pour ajouter un élément `v` supplémentaire à la pile `P`
    — opération d'« empilage » (« push » en anglais),
  - soit pour enlever `v` le dernier élément de la pile `P` (et accéder à sa valeur)
    — opération de « dépilage » (« pop » en anglais).



En `Python` les listes peuvent être manipulées comme des piles à l'aide des opérations suivantes (qui effectuent automatiquement le redimensionnement de la liste) :

- méthode `append` : `P.append(v)` (empilage de l'élément `v` au sommet de la pile `P`) ;
- méthode `pop` : `v= P.pop()` (dépilage et affectation de l'élément dépilé dans la variable `v`) ;

⇝ provoque une erreur si `P` est vide ;

- création d'une pile : `P=[]` (alors on teste si une pile est vide par : `P==[]`).

Dans ce qui suit on n'utilisera que les manipulations de piles listées ci-dessus ; en particulier, on s'interdira d'accéder à l'élément d'indice `i` d'une pile `P` avec « `P[i]` » ou « `for x in P` », d'utiliser la longueur d'une pile « `len(P)` », ainsi que de concaténer des piles ou d'effectuer la comparaison de deux piles « `P1==P2` » — **sauf pour tester si une pile est vide**.
Néanmoins on pourra utiliser ces fonctions sur les listes qui ne représentent pas des piles.

On rappelle qu'une procédure est une fonction qui ne renvoie pas de résultat (mais elle peut modifier les paramètres donnés en entrée).

Rappel : sauvegardez votre code au fur et à mesure (et surtout avant d'exécuter une boucle `while`).

II Quelques manipulations de base

Q1 Écrire une procédure `intervertit(P)` qui intervertit les deux derniers éléments de la pile `P` (supposée de taille au moins égale à deux).

Q2 Écrire une fonction `inverser_pile(P)` qui renvoie la pile obtenue avec les éléments de `P` rangés dans l'ordre inverse (la pile `P` est laissée vide)

Q3 Écrire une fonction `copier_pile(P)` qui renvoie une copie de la pile `P` (tout en laissant la pile `P` inchangée). *On n'utilisera pas* `copy.deepcopy` *ni* `Q=P[:]`.

Q4 Écrire une procédure `empiler_pile(P1,P2)` qui modifie la pile `P1` en lui ajoutant tous les éléments de `P2`, de la base au sommet (la pile `P2` est laissée vide).

Q5 Écrire une fonction `fusion(P1,P2)` où `P1` et `P2` sont deux piles formées de nombres (globalement) distincts qui sont empilés dans l'ordre croissant. Cette fonction renvoie la pile obtenue par réunion de ces deux piles, où les nombres sont empilés dans l'ordre croissant.
Par exemple : `fusion([1,3,8],[2,4,6,9])` → `[1,2,3,4,6,8,9]`

III Expressions bien parenthésées

Exemple d'expression bien parenthésée : `([3+2]*5) + (2+(3*{5+2}))`

Exemples d'expressions mal parenthésées : `(3+2]` et `(2*(3+4(]`.

Plus généralement, on appelle délimiteurs ouvrants les caractères `"("`, `"{"`, `"["` et délimiteurs fermants les caractères `")"`, `"}"`, `"]"`.
Une expression est bien parenthésée si elle vérifie (récursivement) les règles suivantes : elle ne contient aucun délimiteur, ou bien elle s'obtient à partir d'une expression bien parenthésée en insérant à un endroit donné une expression bien parenthésée à laquelle on ajoute, ou non au début un délimiteur ouvrant et à la fin un délimiteur fermant de même type. Sinon elle est mal parenthésée. *Par exemple :*

`(abac[bcg2]*123)` / `pcsi(1+2)` → `(abac[bc g2]*123)` / `{pcsi(1+2)}` → `(abac[bc{pcsi(1+2)}g2]*123)`

Q6 On note L1 la liste des délimiteurs ouvrants et L2 la liste des délimiteurs fermants associés, écrits dans le même ordre.
Écrire une fonction auxiliaire `paireDelimiteurs(c1,L1,c2,L2)`, qui renvoie `True` quand `c1` est un caractère ouvrant de `L1` et `c2` est un caractère fermant de `L2` et `c1` est associé à `c2` et `False` dans tous les autres cas.
Par exemple : `paireDelimiteurs("{","({[","}",")}]")`→`True`
`paireDelimiteurs("{","({[",")","}]")`→`False`

Q7 Écrire une fonction `bien_parenthesee(s)` qui revoie `True` si la chaîne de caractères `s` est bien parenthésée, et qui renvoie `False` sinon. Pour cela :

- on parcourra cette chaîne de caractères en empilant les délimiteurs ouvrants (ou leur type) et à chaque délimiteur fermant rencontré, on dépilera le délimiteur ouvrant correspondant, s'il est de même type ;
- on se demandera, lors de ce processus, ce qui indique que l'expression est bien ou mal parenthésée.

IV Notation polonaise inversée (ou notation postfixée)

La notation polonaise inversée permet de représenter une expression algébrique sans utiliser de parenthèses. Les expressions y sont représentées par des suites de valeurs et d'opérateurs où les opérateurs sont situés juste après les nombres sur lesquels ils agissent.
Par exemple : $a * b$ est représenté par la suite $a\ b\ *$.
$a * (b + c)$ est représenté par la suite $a\ b\ c\ +\ *$.
$a * b + c$ est représenté par la suite $a\ b\ *\ c\ +$ (le produit est prioritaire sur la somme).

Inversement, pour lire une expression plus complexe, on lit l'expression de gauche à droite et lorsqu'on rencontre une suite $a\ b\ T$, où a et b sont des valeurs et T est une opération, on peut la remplacer par le résultat de aTb. Et l'on recommence sur la nouvelle suite ainsi obtenue.
Par exemple : $1\ \ 3\ \ \underline{2\ \ 4\ \ +}\ \ *\ \ +\ \to\ 1\ \ \underline{3\ \ \mathbf{6}\ \ *}\ \ +\ \to\ \underline{1\ \ \mathbf{18}\ \ +}\ \to\ \mathbf{19}$
(en notation algébrique habituelle, on a calculé $1 + 3 * (2 + 4)$).

Q8 Traduire en notation polonaise inversée le calcul suivant : $(32 + 7) * 6 + (17 + 4) * 5 * 2$

Q9 Traduire en notation algébrique habituelle le calcul suivant : $4\ 5\ 2\ *\ 7\ +\ *$

Dans la suite, on veut écrire une fonction qui donne le résultat d'un calcul écrit en notation polonaise inversée dans une chaîne de caractères `s`. Pour simplifier, on supposera que le calcul ne fait intervenir que des entiers naturels et les deux opérations $+$ et $*$.

Q10 Écrire une fonction `evalue1(s)` qui répond au problème posé lorsque l'expression `s` ne contient que **des entiers à un seul chiffre**.
Pour cela, on empilera les entiers qui apparaissent dans cette chaîne de caractères jusqu'à rencontrer une opération ; on dépilera alors les deux derniers entiers rencontrés pour empiler à leur place le résultat de l'opération. Et ainsi de suite, jusqu'à la fin de la chaîne.
Par exemple, `evalue1("452+*3*")` devra renvoyer `84` (car $84 = (4 * (5 + 2)) * 3$).

Q11 Écrire la fonction `evalue2(s)` dans le cas général où les entiers sont formés d'un ou plusieurs chiffres. On supposera cette fois-ci que les éléments constitutifs de `s` sont séparés les uns des autres au moyen d'un espace `" "`.
Par exemple, la chaîne `"12 32 *"` correspond à $12 * 32$, tandis que `"123 2 *"` correspond à $123 * 2$.
Remarque : la méthode `split` (tester sur une chaîne) permettrait de simplifier la fonction précédente.

V D'autres questions pour ceux qui ont déjà tout fini

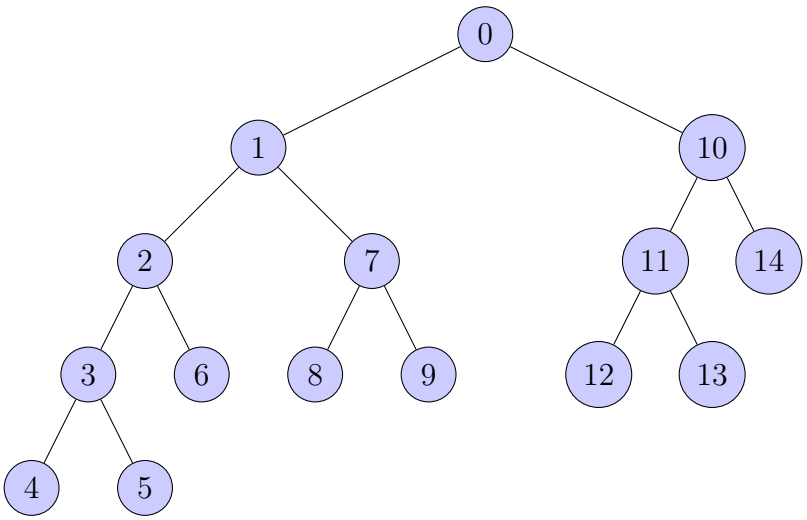
On pourra ré-utiliser les manipulations de base de la partie 2 (sans confondre fonctions et procédures).

Q12 Soit `P` une pile d'entiers naturels. Écrire une fonction `pair(P)` qui renvoie une pile formée des éléments de `P` dont la valeur est paire, placés dans le même ordre que dans `P`.

Q13 La pile `P` est un palindrome si elle contient les mêmes éléments quand on la lit de haut en bas ou quand on la lit de bas en haut. Écrire une fonction `palindrome(P)` qui renvoie `True` si la pile `P` est un palindrome et `False` sinon.
Par exemple : `palindrome([1,3,8,3,1])` → `True`
`palindrome([1,3,8,2,3,1])` → `False`

Q14 Écrire une fonction `melanger(P1, P2)` qui mélange les éléments de deux piles `P1` et `P2` dans une troisième pile de la façon suivante : tant qu'une pile (au moins) n'est pas vide, on retire aléatoirement un élément au sommet d'une des deux piles et on l'empile sur la pile résultat ; à la fin on empile la pile qui n'est pas encore vide avec la pile résultat.

Q15 Un arbre est un graphe (i.e. un objet formé d'un ensemble de sommets et d'arêtes qui relient ces sommets) tel qu'il existe un sommet particulier, appelé la racine, et que tout autre sommet soit relié à la racine par un unique chemin (i.e. une suite d'arêtes contiguës — sans retour en arrière). *Par exemple (racine* 0*) :*



Écrire une fonction `chemin(A,racine,sommet)`, où `A` est la liste des arêtes, chacune représentée par une liste `[d,f]` où `d,f` sont des sommets. Cette fonction renvoie le chemin qui va de la racine au sommet `sommet` sous la forme de la liste des sommets parcourus.
Par exemple (avec l'arbre représenté) : `chemin(A,0,9)`→ `[0,1,7,9]`, où `A` est
`[[0,1],[0,10],[1,2],[1,7],[7,8],[7,9],[2,3],[2,6],[3,4],[3,5],[10,11],[10,14],[11,12],[11,13]]`
On donnera une première version récursive (en faisant attention de minimiser le nombre d'appels récursifs), puis une seconde version, itérative qui construit une pile de chemins. Au début, cette pile ne contient que le chemin `[racine]` ; puis le chemin du sommet de la pile est remplacé par tous ses prolongements possibles (i.e. avec un sommet de plus), ou bien supprimé s'il ne peut pas être prolongé ; et ainsi de suite jusqu'à obtenir un chemin qui aboutit au sommet `sommet` (on pourra utiliser une boucle `while` infinie, interrompue par un `return` — bien sauvegarder son programme avant d'exécuter).

Corrigé

Q1

```
def intervertit(P):
    dernier = P.pop()
    avant_dernier = P.pop()
    P.append(dernier)
    P.append(avant_dernier)
    #pas de return car c'est une procedure

L = [1, 3, 5, 7, 9, 11, 13]; intervertit(L)
assert L == [1, 3, 5, 7, 9, 13, 11]
```

Q2

```
def inverser_pile(P):
    Q = []
    while P != []:
        Q.append(P.pop())
    return Q #N.B. : à la fin, la pile P est vide

L = [1, 3, 5, 7, 9, 11, 13]
assert inverser_pile(L) == [13, 11, 9, 7, 5, 3, 1]
assert L == []
```

Q3

```
def copier_pile(P):
    Q = inverser_pile(P) #on inverse la pile P ; à la fin, la pile P est
    vide
    R = []
    while Q != []:
        dernier = Q.pop()
        P.append(dernier) #on reconstitue la pile P...
        R.append(dernier) #... et on effectue une copie en même temps
    return R

L = [1, 3, 5, 7, 9, 11, 13]
assert copier_pile(L) == [1, 3, 5, 7, 9, 11, 13]
assert L == [1, 3, 5, 7, 9, 11, 13]
```

Q4

```
def empiler_pile(P1, P2):
    Q = inverser_pile(P2) #à la fin, la pile P2 est vide
    while Q != []:
        P1.append(Q.pop())
    #pas de return car c'est une procedure

P1 = [1, 3, 5, 7, 9, 11, 13]
P2 = [0, 2, 4, 6, 8]
empiler_pile(P1, P2)
assert P1 == [1, 3, 5, 7, 9, 11, 13, 0, 2, 4, 6, 8]
assert P2 == []
```

Q5

```
from numpy import inf

def fusion(P1,P2):
    P = []
    P1.append(inf) #ajout de sentinelle à P1
    P2.append(inf) #ajout de sentinelle à P2
    P1R = inverser_pile(P1) #à la fin, la pile P1 est vide
    P2R = inverser_pile(P2) #à la fin, la pile P2 est vide
    while P1R != [] and P2R != []: #avec les sentinelles, aucune liste n'est vide
        e1 = P1R.pop()
        e2 = P2R.pop()
        if e1 <= e2:
            P.append(e1)
            P2R.append(e2)
        else:
            P.append(e2)
            P1R.append(e1)
    P.pop()
    return P

P1 = [1, 3, 5, 7, 9, 11, 13]
P2 = [0, 2, 4, 6, 8]
assert fusion(P1,P2) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13]
```

Q6

```
def paireDelimiteurs(c1,L1,c2,L2):
    for i in range(len(L1)):
        if L1[i] == c1 and L2[i] == c2:
            return True
    return False

assert paireDelimiteurs("{","({[","}",")}]") == True
assert paireDelimiteurs("{","({[",")",")}]") == False
assert paireDelimiteurs("!","({[",")",")}]") == False
```

Q7

```
def bien_parenthesee(s):
    ouvrants = "({["
    fermants = ")}]"
    delimiteurs = [] #pile des délimiteurs ouvrants

    for i in range(len(s)):
        if s[i] in ouvrants:
            delimiteurs.append(s[i]) #empilement délimiteurs ouvrants
        elif s[i] in fermants:
            if delimiteurs == []:
                return False  #si on a un délimiteur fermant et aucun ouvrant en pile: erreur
            else:
                c = delimiteurs.pop()
                if not paireDelimiteurs(c,ouvrants,s[i],fermants):
                    return False #si on a un délimiteur fermant non associé à l'ouvrant correspondant: erreur
    if delimiteurs != []:
        return False #si il reste des ouvrants alors qu'il n'y a plus de fermant : erreur

    return True
```

L'expression est bien parenthésée si tout délimiteur ouvrant empilé est ensuite dépilé par une délimiteur fermant du même type.
Donc ce n'est pas le cas dans les situations suivantes :

- Un délimiteur fermant ne correspond à aucun délimiteur ouvrant car la pile des délimiteurs ouvrants est vide.
- Un délimiteur fermant est rencontré mais n'est pas du même type que celui du sommet de la pile des délimiteurs ouvrants.
- La pile des délimiteurs ouvrants n'est pas vide à la fin càd. qu'il y a au moins un délimiteur ouvrant sans délimiteur fermant associé.

Q8 On obtient : `6 32 7 + * 17 4 + 5 * 2 * +`

Q9 On obtient : `((5 * 2) + 7 ) * 4`

Q10

```
120 def evalue1(s):
121     pile = []
122     for c in s:
123         if c == '+':
124             pile.append(pile.pop() + pile.pop())
125         elif c == '*':
126             pile.append(pile.pop() * pile.pop())
127         elif c.isdigit(): #ou else si c est forcément un chiffre
128             pile.append(int(c))
129     return pile.pop()
130 
131 assert evalue1("452+*3*") == 84
```

Q11

```
134 def pousse_pile(pile, d):
135     if d != '': # cas des espaces avant ou après + et *
136         pile.append(int(d))
137 
138 def evalue2(s):
139     pile = []
140     d = ''
141     for c in s:
142         if not c.isdigit(): # c n'est pas un chiffre
143             pousse_pile(pile, d); d = ''
144             if c == '+':
145                 pile.append(pile.pop() + pile.pop())
146             elif c == '*':
147                 pile.append(pile.pop() * pile.pop())
148         else : # c est forcément un chiffre
149             d += c # on reconstitue le nombre
150     return pile.pop()
151 
152 
153 assert evalue2("12 32*") == 384
154 assert evalue2("6 32 7 + * 17 4 + 5 * 2 * +") == 444
```

```python
# la version simplifiée avec split
def evalue3(s) :
    pile = []
    for c in list(s.split()) : #on parcourt la chaine s splittée
        if c == '+' :
            pile.append(pile.pop() + pile.pop())
        elif c == '*' :
            pile.append(pile.pop() * pile.pop())
        else : #alors c représente un entier
            pile.append(int(c))
    return pile.pop()

assert evalue3("6 32 7 + * 17 4 + 5 * 2 * +") == 444
```

Q12

```python
def pair(P):
    Q = inverser_pile(P)
    R = []
    while Q != []:
        x = Q.pop()
        if x%2 == 0:
            R.append(x)
    return R

P = [1, 2, 3, 4, 6]
assert pair(P) == [2, 4, 6]
```

Q13

```python
def palindrome(P):
    Q = inverser_pile(copier_pile(P))
    while Q != []:
        if Q.pop() != P.pop():
            return False
    return True

P = [1, 3, 8, 3, 1]; assert palindrome(P) == True
P = [1, 3, 8, 3, 8]; assert palindrome(P) == False
```

Q14

```python
from random import randint

def melanger(P1, P2):
    P = []
    while P1 != [] and P2 != []:
        i = randint(1,2)
        if i == 1:
            P.append(P1.pop())
        else:
            P.append(P2.pop())
    empiler_pile(P,P1)
    empiler_pile(P,P2)
    return P

P1 = [1, 3, 5, 7, 9, 11, 13]
P2 = [0, 2, 4, 6, 8]
print(melanger(P1, P2))
```

Q15

```
# version récursive
def recherche_sommet(A, sommet):
    # en O(A**2) dans le pire des cas, mais on peut faire une version dichotomique en O(A*log(A))
    for a in A:
        if a[1] == sommet:
            return a[0]

def chemin(A,racine,sommet):
    '''chemin de racine à sommet. Quand racine == sommet, on renvoie racine.
       Préconditions : racine est la racine de l'arbre et sommet l'une de ses feuilles.'''
    if racine == sommet:
        return [racine]
    else: # la boucle for de recherche est dans la fonction recherche_sommet
        return chemin(A,racine,recherche_sommet(A, sommet)) + [sommet]

A = [[0,1],[0,10],[1,2],[1,7],[7,8],[7,9],[2,3],[2,6],[3,4],[3,5],
     [10,11],[10,14],[11,12],[11,13]]
assert chemin(A,0,9) == [0, 1, 7, 9]
assert chemin(A,0,0) == [0]
```

```
## autre version récursive
def chemin(A, racine, sommet):
    '''chemin de racine à sommet. Quand racine == sommet, on renvoie racine.
       Préconditions : racine est la racine de l'arbre et sommet l'une de ses feuilles.'''
    if racine == sommet:
        return [racine]
    for arete in A: # on cherche le sommet passé comme élément à droite d'une arête, ce qui est toujours le cas (cf. précondition)
        d, f = arete
        if arete == [racine, sommet]: # condition d'arrêt toujours atteinte
            return arete
        if f == sommet:
            ch = chemin(A, racine, d)
            return ch + [sommet]

A = [[0,1],[0,10],[1,2],[1,7],[7,8],[7,9],[2,3],[2,6],[3,4],[3,5],
     [10,11],[10,14],[11,12],[11,13]]
assert chemin(A,0,9) == [0, 1, 7, 9]
assert chemin(A,0,0) == [0]
assert chemin(A,0,1) == [0, 1]
```

```
257 ## version itérative
258 def cheminIt(A,racine,sommet):
259     ''' On part de la racine et on construit tous les chemins possibles
        jusqu'à atteindre le sommet; on empile les chemins possible et on les
        dépile quand ils n'aboutissent pas à sommet'''
260     pile = [[racine]]
261     while pile != []:
262         ch = pile.pop()
263         for arete in A:
264             d, f = arete
265             if ch[-1] == sommet:
266                 return ch
267             elif ch[-1] == d:
268                 pile.append(ch + [f])
269
270 A = [[0,1],[0,10],[1,2],[1,7],[7,8],[7,9],[2,3],[2,6],[3,4],[3,5],
271      [10,11],[10,14],[11,12],[11,13]]
272 assert cheminIt(A,0,9) == [0, 1, 7, 9]
273 assert cheminIt(A,0,0) == [0]
274 assert cheminIt(A,0,1) == [0, 1]
```