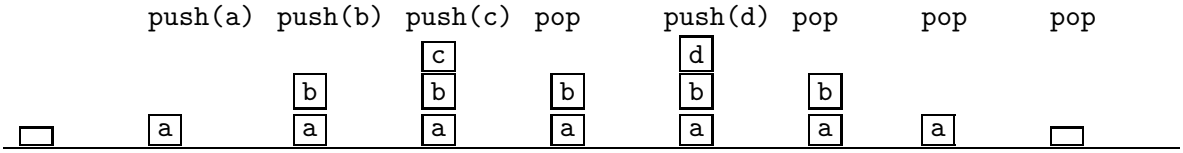
INFORMATIQUE

TD n°5

1 Introduction aux piles

Une pile est une structure de donnée alternative aux listes, mieux adapté 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)` (ajout 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`) ;
- 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]` », d'utiliser la longueur d'une pile « `len(P)` », ainsi que d'effectuer la comparaison de deux pile « `P1==P2` » — sauf pour tester si une pile est vide.

Néanmoins on pourra utiliser ces fonctions sur les listes et les chaînes 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).

2 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 par `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]`

3 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)` et `pcsi(1+2)` $\rightarrow$ `(abac[bc g2]*123)` et `{pcsi(1+2)}` $\rightarrow$ `(abac[bc{pcsi(1+2)}g2]*123)`

Q6 Écrire une fonction `typeDelimiteur(c,L)`, où `c` est un caractère et `L` est une chaîne de caractères distincts, qui renvoie `i` si `L[i]` est égal à `c` et qui renvoie `"non"` s'il n'y a aucun `i` ainsi.

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 caractère 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.

4 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 : $1\ 3\ \underline{2\ 4\ +}\ *\ + \rightarrow 1\ \mathbf{3}\ \underline{\mathbf{6}}\ *\ + \rightarrow \underline{1\ \mathbf{18}\ +} \rightarrow \underline{19}$
(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 caractère `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 $(4*(5+2))*3=84$.

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 consécutifs 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.

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

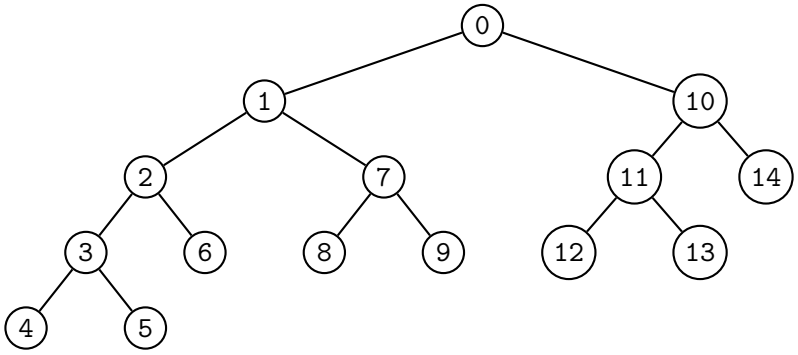
On pourra ré-utiliser les manipulations de base de la partie 2 (sans confondre fonctions et procédures).

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 haut en bas ou de 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.

Q15 Un arbre est 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(sommet,A,racine)`, où `A` est la liste des arêtes, chacune représenté 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(9,A,0)`→ `[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 fichier avant d'exécuter).

INFORMATIQUE

Corrigé du TD n°5

Q1

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

Q2

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

Q3

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

Q4

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

Q5

```
def fusion(P1,P2):
    P=[]
    while P1!=[] and P2!=[]:
        x1=P1.pop()
        x2=P2.pop()
        if x1>x2:
            P.append(x1)
            P2.append(x2)
        else:
            P.append(x2)
```

```
            P2.append(x1)
    while P1!=[]:
        P.append(P1.pop())
    while P2!=[]:
        P.append(P2.pop())
    inverser(P)
    return P
```

Q6

```
def typeDelimiteur(c,L):
    for i in range(len(L)):
        if L[i]==c:
            return i
    return "non"
```

Q7 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 cas suivants : un délimiteur fermant est rencontré mais n'est pas du même type de le sommet de la pile (ou que la pile est vide) ; la pile n'est pas vide à la fin.

```
def bien_parenthesee(s):
    ouvrants="([{"
    fermants=")]}"
    P=[] #initialisation de la pile
    for c in list(s):
        i=typeDelimiteur(c,ouvrants)
        if i!="non": #empilage des délimiteurs ouvrants
            P.append(i)
        else:
            j=typeDelimiteur(c,fermants)
            if j!="non" and P==[]: #délimiteur fermant sans délimiteur
    ouvrant
                return False
            elif j!="non": #delimteur fermant avec pile non vide
                i=P.pop()
                if i!=j:
                    return False
        # ne fait rien si le caractere c n'est pas un delimiteur
    if P==[]:
        return True
    return False
```

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

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

Q10

```
def evalue1(s):
    P=[]
    for c in list(s): #on parcourt la chaine de caracteres s
        if c=='+':
            P.append(P.pop()+P.pop())
        elif c=='*':
            P.append(P.pop()*P.pop())
```

```
96          else: #alors c est un caracteres de 0 a 9
97              P.append(int(c))
98      return P.pop()
```

Q11

```
101 def evalue2(s):
102     P=[]
103     i=0
104     l=len(s)
105     while i<l: #on parcourt la chaine de caracteres s avec l'indice i
106         p=1   #nombre de caracteres de l'element analyse
107         if s[i]=='+':
108             P.append(P.pop()+P.pop())
109         elif s[i]=='*':
110             P.append(P.pop()*P.pop())
111         elif s[i] in ['0','1','2','3','4','5','6','7','8','9']:
112             #alors s[i] est un caractere de 0 a 9 qui fait partie d'un entier
113             #recherche du nombre de chiffres de l'entier
114             while s[i+p] in ['0','1','2','3','4','5','6','7','8','9']:
115                 p=p+1
116             P.append(int(s[i:i+p]))
117         i=i+p
118     return P.pop()
119
120 def evalue3(s):
121     P=[]
122     for c in list(s.split()): #on parcourt la chaine s splittee
123         if c=='+':
124             P.append(P.pop()+P.pop())
125         elif c=='*':
126             P.append(P.pop()*P.pop())
127         else: #alors c represente un entier
128             P.append(int(c))
129     return P.pop()
```

Q12

```
132 def pair(P):
133     R=[]
134     while P!=[]:
135         v=P.pop()
136         if v%2==0:
137             R.append(v)
138     return inverser_pile(R)
```

Q13

```
141 def palindrome(P):
142     Q=inverser_pile(copier_pile(P))
143     while Q!=[]:  # test de l'égalité des piles P et Q
144         if Q.pop()!=P.pop():
145             return False
146     return True
```

Q14

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

Q15

```python
def chemin(sommet, A, racine): #renvoie False si aucun chemin n'existe
    for arete in A:
        d,f=arete
        if d==racine and f==sommet:
            return arete
        if f==sommet:
            ch=chemin(d,A,racine) #test en deux temps pour minimiser le nombre d'appels récursifs
            if chemin!=False:
                return ch+[sommet]
    return False

#version avec pile
def chemin(sommet, A,racine):
    pile=[[racine]]
    while pile!=[]:
        ch=pile.pop()
        for arete in A:
            d,f=arete
            if ch[-1]== sommet: #ch est vu comme une liste, pas comme une pile
                return ch
            elif ch[-1]==d:
                pile.append(ch+[f])
```