TD – Piles et files

Corrigé

Piles

Exercice N°1 – Copie d'une pile

Ecrire une fonction `stack_copy(s)` recevant une pile (`s`) comme argument et renvoyant une copie `s2` de `s`. Attention, la pile `s` doit (bien sûr) être conservée !

Evaluer le coût en mémoire et le nombre d'opérations de la fonction.

Puisque, dans une pile, nous ne pouvons manipuler que le sommet de la pile, nous n'avons pas d'autre choix, pour pouvoir accéder aux éléments successifs de `s`, que de la dépiler dans un premier temps (1ère boucle `for` ci-dessous) …

```
def stack_copy(s):
     s2 = stack_create()
     if len(s) != 0:
          t = stack_create()
          for i in range(len(s)):
               e = stack_peek(s)
               stack_pop(s)
               stack_push(t,e)
          for i in range(len(t)):
               e = stack_peek(t)
               stack_pop(t)
               stack_push(s,e)
               stack_push(s2,e)
     return(s2)
```

Notons n la taille de l'espace mémoire occupé par la pile `s`.

A priori, la fonction `stack_pop` libère progressivement l'espace mémoire occupé par `s`. Mais parallèlement, on construit la pile `t`. Ainsi, l'espace mémoire total requis par les piles `s` et `t` dans la première boucle `for` est constant et égal à n.

Dans la deuxième boucle `for`, on vide la pile `t` mais on construit au fur et à mesure les piles `s` et `s2`. Ainsi, l'occupation mémoire lors de l'exécution de la deuxième boucle passe de n à

$2n$. Bien sûr, si on ne souhaite pas conserver `s`, cette occupation est à nouveau égale à n (on dépile `t` pour construire `s2`).

Pour ce qui est des appels à des fonctions, on se limite aux fonctions `stack_peek`, `stack_pop` et `stack_push`.
Dans la première boucle `for`, on a n appels à chacune des fonction `stack_peek`, `stack_pop` et `stack_push`.
Dans la seconde boucle `for`, on a n appels à chacune des fonction `stack_peek` et `stack_pop` et $2n$ appels à la fonction `stack_push`.

En définitive, on a :

- $2n$ appels à la fonction `stack_peek`.
- $2n$ appels à la fonction `stack_pop`.
- $3n$ appels à la fonction `stack_push`.

Si on ne souhaite pas conserver `s`, on ne reconstruira pas cette pile dans la deuxième boucle `for` et on aura « seulement » `n` appels à la fonction `stack_push` pour un total de $2n$ appels (au lieu de $3n$).

Exercice N°2 – Inversion d'une pile

Ecrire une fonction `stack_reverse` recevant une pile (`s`) comme argument et renvoyant une copie inversée `rs` de `s`. Attention, la pile `s` doit être conservée !

Evaluer le coût en mémoire et le nombre d'opérations de la fonction.

Dans l'écriture de la fonction précédente, on a vu que la première boucle `for` permettait d'obtenir une nouvelle pile, inverse de la pile initiale MAIS en lieu et place de celle-ci. Pour conserver cette pile initiale, il suffit donc, dans un premier temps, de la copier en utilisant la fonction `stack_copy` !

```
def stack_reverse(s):
    rs = stack_create()
    if len(s) != 0:
        sc = stack_copy(s)
        for i in range(len(s)):
            e = stack_peek(sc)
            stack_pop(sc)
            stack_push(rs,e)
    return(rs)
```

Notons encore n la taille de l'espace mémoire occupé par la pile `s`.

Dans l'exercice précédent, on a vu que, en conservant la pile initiale, le besoin en mémoire de la fonction `stack_copy` était de $2n$.

Pour ce qui est des fonctions, on avait :

- $2n$ appels à la fonction `stack_peek`.
- $2n$ appels à la fonction `stack_pop`.
- $3n$ appels à la fonction `stack_push`.

La construction de la pile `rs` n'engendre pas de besoin mémoire supplémentaire.

En revanche, cette construction de `rs` engendre :

- n appels à la fonction `stack_peek`.
- n appels à la fonction `stack_pop`.
- n appels à la fonction `stack_push`.

En définitive, on a au total :

- $3n$ appels à la fonction `stack_peek`.
- $3n$ appels à la fonction `stack_pop`.
- $4n$ appels à la fonction `stack_push`.

En résumé, pour une liste `s` de longueur n :

	Avec conservation de `s`	Sans conservation de `s`
Besoin en mémoire	$2n$	n
Appels `stack_peek`	$3n$	n
Appels `stack_pop`	$3n$	n
Appels `stack_push`	$4n$	n

Sans surprise, ce tableau illustre clairement le fait que c'est la conservation de `s` qui est coûteuse en mémoire et en appels aux fonctions `stack_peek`, `stack_pop` et `stack_push`.

Exercice N°3 – Permutations circulaires (acte 1)

Ecrire une fonction `stack_circperm` qui reçoit en argument une pile `s` et un entier `n` et effectue sur la pile `n` permutations circulaires successives.

Dans cet exercice, c'est la pile `s` elle-même qui sera modifiée.

Exemple avec `n=2` :

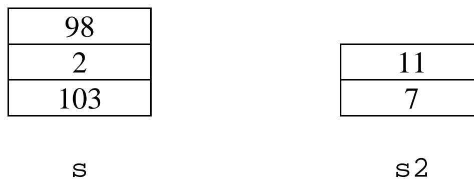
7		98
11		2
98	donnera	103
2		7
103		11

Evaluer le coût en mémoire et le nombre d'opérations de la fonction.

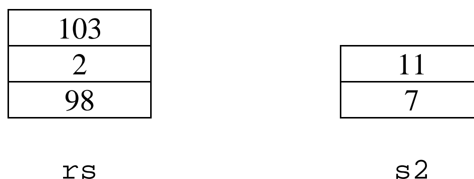
Une première remarque en guise de préambule : on peut supposer que l'utilisateur (ou le programme appelant) fournisse bien pour `n` un entier naturel. Mais il est possible que cet entier soit plus grand que la longueur de la pile `s`. Ainsi, le nombre effectif de permutations circulaires à mettre en œuvre est en fait égal au reste `r` de la division euclidienne de `n` par `len(s)`, soit, en Python : `r=n%len(s)`. Il est clair que si ce reste est nul, il convient de ne rien faire !

Illustrons le principe général de l'algorithme à partir de l'exemple fourni dans l'énoncé.

On commence par construire une pile `s2` contenant les `r` premiers éléments de la pile `s` qui est donc successivement dépilée. On se retrouve ainsi avec les deux piles :



On inverse alors la pile `s` (on note `rs` la pile inversée). On pourrait bien sûr utiliser la fonction de l'exercice 2 mais conserver la pile `s` ne nous est à priori ici d'aucune utilité. La pile `rs` est donc obtenue directement en dépilant la pile `s`. On obtient :



La pile demandée, que nous notons `cps`, est alors directement obtenue en dépilant successivement `s2` puis `rs`.

```
def stack_circperm(s,n):
n = n%len(s)
if n != 0:
     # Construction de s2
     s2 = stack_create()
     for i in range(n):
          x = stack_peek(s)
          stack_pop(s)
          stack_push(s2,x)
     # Construction de rs
     rs = stack_create()
     for i in range(len(s)):
          x = stack_peek(s)
          stack_pop(s)
          stack_push(rs,x)
     # Construction de cps
     cps = stack_create()
     for i in range(len(s2)):
          x = stack_peek(s2)
          stack_pop(s2)
          stack_push(cps,x)
     for i in range(len(rs)):
          x = stack_peek(rs)
          stack_pop(rs)
          stack_push(cps,x)
return cps
```

Soit L la taille de la pile `s` :

- dans la première boucle for, on dépile `s` pour construire `s2` : le besoin en mémoire est toujours égal à L.
- dans la seconde boucle, on inverse la pile `s` pour construire la pile `rs` mais sans en garder de copie : le besoin en mémoire est toujours égal à L.
- enfin, dans les deux dernières boucles, on dépile les listes `s2` et `rs` respectivement pour construire la pile `cps` : une fois encore, le besoin en mémoire est égal à L.

Le besoin en mémoire de la fonction `stack_circperm` est égal à la taille de la liste `s` passée en argument (la variable `n` devrait en toute rigueur être prise en compte mais pour L grand, le besoin en mémoire admet L pour équivalent).

Pour ce qui est des appels à des fonctions, on se limite aux fonctions `stack_peek`, `stack_pop` et `stack_push` :

- dans la première boucle `for`, on a `n` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.
- dans la seconde boucle `for`, on a L`-n` (longueur de la pile `s`) appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.
- dans la troisième boucle `for`, on a `n` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.

- enfin, dans la quatrième boucle `for`, on a `L-n` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.

En définitive, on a 2L appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.

Exercice N°4 – Permutations circulaires (acte 2)

Ecrire une fonction `stack_circperm2` qui reçoit en argument une pile `s` et un entier `k` et effectue une permutation circulaire sur les `k` premiers éléments de la pile.

Dans cet exercice, c'est la pile `s` elle-même qui sera modifiée.

Exemple avec `k=4` :

7		11
11		98
98	donnera	2
2		7
103		103
5		5

Evaluer le coût en mémoire et le nombre d'opérations de la fonction.

Remarquons d'abord que la modification souhaitée n'a de sens que si l'entier `k` est inférieur ou égal à la longueur de la pile `s`.

A partir de là, on peut adopter deux approches :

- On utilise la fonction `stack_circperm` de l'exercice précédent mais en l'appliquant à une certain pile.
- On note que le problème consiste à déplacer le sommet de `s`.

Evidemment, on ne fait quelque chose sur `s` que si les conditions suivantes sont satisfaites :

- `s` est non vide.
- `k` est strictement positif.
- `k` est inférieur ou égal à la taille de `s`.

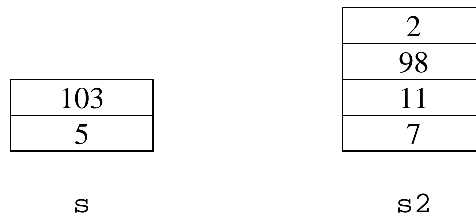
1ère approche : utilisation de `stack_circperm`

L'algorithme peut être sommairement décrit comme suit (les inversions apparaissent du fait que tout dépilement/empilement induit une inversion) :

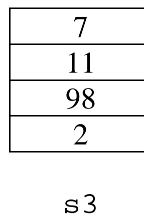
- Dépiler les `k` premiers éléments de `s`. On obtient une deuxième pile `s2`.
- On inverse `s2`. On obtient `s3`.
- Effectuer une permutation circulaire de `s3`. On obtient `cps3`.
- On inverse `cps3`. On obtient `s4`.
- Dépiler `s4`, les éléments étant successivement empilés sur `s`.

A la deuxième et à la quatrième étape, on utilisera la fonction `stack_reverse`.

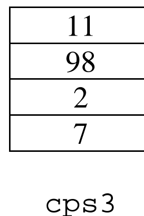
Avec l'exemple de l'énoncé, on obtient :



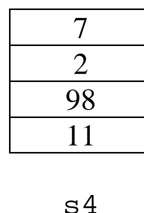
On inverse `s2` :



La permutation circulaire de `s3` donne la pile :



On inverse `cps3` :



On dépile s4 et les éléments sont successivement empilés sur s :

11
98
2
7
103
5

s

D'où la fonction :

```
def stack_circperm2(s,k):
     if len(s) > 0 and k > 0 and k <= len(s):
          # Construction de s2
          s2 = stack_create()
          for i in range(k):
               x = stack_peek(s)
               stack_pop(s)
               stack_push(s2,x)
          # Inversion de s2
          s3 = stack_reverse(s2)
          # Permutation de s3
          cps3 = stack_circperm(s3,1)
          # Inversion de cps3
          s4 = stack_reverse(cps3)
          # Reconstruction de s
          for i in range(k):
               x = stack_peek(s4)
               stack_pop(s4)
               stack_push(s,x)
```

Soit L la taille de la pile s :

- dans la première boucle for, on dépile les k premiers éléments de s pour construire s2 : le besoin en mémoire est toujours égal à L.
- l'inversion de la pile s2 pour construire la pile s3 se fait sans conservation d'une copie : le besoin en mémoire est toujours égal à L.
- la permutation de la pile s3 pour construire la pile cps3 se fait sans conservation d'une copie : d'après l'exercice précédent, le besoin en mémoire est toujours égal à L.
- l'inversion de la pile cps3 pour construire la pile s4 se fait sans conservation d'une copie : le besoin en mémoire est toujours égal à L.
- enfin, dans la dernière boucle, on dépile s4 pour reconstruire la pile s : une fois encore, le besoin en mémoire est égal à L.

Le besoin en mémoire de la fonction `stack_circperm2` est égal à la taille de la liste `s` passée en argument (la variable `k` devrait en toute rigueur être prise en compte mais pour L grand, le besoin en mémoire admet L pour équivalent).

Pour ce qui est des appels à des fonctions, on se limite aux fonctions `stack_peek`, `stack_pop` et `stack_push` :

- dans la première boucle `for`, on a `k` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.
- l'inversion de `s2` requiert `k` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.
- La permutation circulaire de `s3` requiert, d'après l'exercice précédent, `k` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.
- l'inversion de `s4` requiert `k` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.
- enfin, dans la quatrième boucle `for`, on a `k` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.

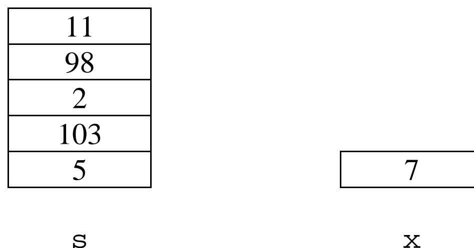
En définitive, on a 6`k` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.

2ème approche : déplacement du sommet de la pile

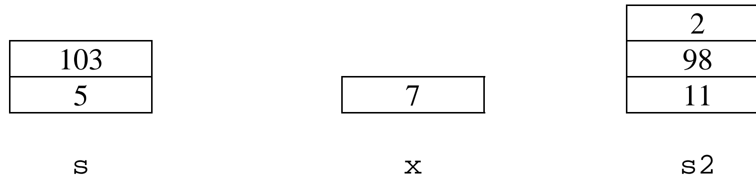
Dans cette approche, on procède comme suit :

- Obtenir le sommet de `s` : `x`.
- Dépiler `k-1` éléments de `s`. On obtient une deuxième pile `s2`.
- Empiler `x` sur `s`.
- Dépiler `s2`, les éléments étant successivement empilés sur `s`.

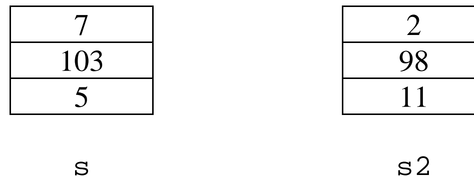
Avec l'exemple de l'énoncé, on obtient :



On dépile alors `k-1=4-1=3` éléments de `s` pour obtenir `s2` :



On empile le sommet `x` sur `s` :



Enfin, on dépile `s2` et les éléments sont successivement empilés sur `s`.

D'où la fonction :

```
def stack_circperm2bis(s,k):
     if len(s)>0 and k>0 and k <= len(s):
          # Obtention du sommet de s
          x = stack_peek(s)
          stack_pop(s)
          # Construction de s2
          s2 = stack_create()
          for i in range(k-1):
               e = stack_peek(s)
               stack_pop(s)
               stack_push(s2,e)
          # On empile x sur s
          stack_push(s,x)
          # Reconstruction de s
          for i in range(k-1):
               e = stack_peek(s2)
               stack_pop(s2)
               stack_push(s,x)
```

Soit L la taille de la pile `s` :

- dans la première boucle for, on dépile les `k-1` premiers éléments de `s` pour construire `s2` : le besoin en mémoire est toujours égal à L. (L-1, qui est la longueur courante de `s` auquel on ajoute la mémorisation du sommet initial de `s` dans la variable `x`).
- dans la deuxième boucle, on dépile `s2` pour reconstruire la pile `s` : une fois encore, le besoin en mémoire est égal à L.

Le besoin en mémoire de la fonction `stack_circperm2` est égal à la taille de la liste `s` passée en argument (la variable `k` devrait en toute rigueur être prise en compte mais pour L grand, le besoin en mémoire admet L pour équivalent).

Pour ce qui est des appels à des fonctions, on se limite aux fonctions `stack_peek`, `stack_pop` et `stack_push` :

- l'obtention du sommet de la pile initiale s requiert 1 appel à chacune des fonctions `stack_peek` et `stack_pop`.
- dans la première boucle `for`, on a `k-1` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.
- placer x sur s requiert 1 appel à la fonction `stack_push`.
- enfin, dans la deuxième boucle `for`, on a `k-1` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.

En définitive, on a `2k-1` appels à chacune des fonctions `stack_peek`, `stack_pop` et `stack_push`.

Cette deuxième approche est donc nettement plus économe en terme d'appels à nos fonctions de base de manipulation de piles.

Exercice N°5 – Expression correctement parenthésée

Dans un logiciel de calcul formel ou, plus généralement dans un éditeur de texte (par exemple utilisé pour écrire des programmes), il y a une gestion dynamique (i.e. « au vol ») du parenthésage : si une parenthèse fermante de trop est ajoutée ou si elle n'est pas de même nature (une accolade au lieu d'un crochet par exemple) que la parenthèse ouvrante associée alors un message d'erreur (visuel, sonore) est envoyé.

Par exemple, les deux expressions suivantes sont erronées :

$$A = \left(4 + i\sqrt{3}\right))^{17}$$

(deux parenthèses fermantes pour une ouvrante.)

$$B = \lim_{n \to +\infty} \left[1 + \frac{3}{n}\right)^{n}$$

(la parenthèse fermante n'est pas de même nature que la « parenthèse » ouvrante.)

Programmer une fonction qui reçoit comme argument une chaîne de caractères, en analyse la correction du parenthésage et renvoie à l'utilisateur un message adapté.

L'analyse de la chaîne de caractères se fera caractère après caractère. On gèrera une pile contenant les parenthèses ouvrantes et on comparera chaque parenthèse fermante au sommet (éventuel) de la pile.

Remarque : dans ce corrigé, le terme « parenthèse » désigne, sans plus de précision, une parenthèse, un crochet ou une accolade.

Le cœur de l'algorithme de traitement de la chaîne de caractère repose sur l'idée simple que toute parenthèse fermante rencontrée doit être associée dans la pile à la parenthèse (qui doit donc exister !) ouvrante de même nature située au sommet de la pile.

Pour chaque caractère de la chaîne passée en argument, il y a trois possibilités :

- Le caractère n'est pas une parenthèse.
  Dans ce cas, on ne fait rien et on traite l'éventuel caractère suivant.
- Le caractère est une parenthèse ouvrante.
  Dans ce cas, la parenthèse ouvrante est placée dans la pile.
- Le caractère est une parenthèse fermante.
  C'est dans ce cas que le traitement est le plus intéressant …
  On doit vérifier que la pile des parenthèses ouvrantes est non vide.
  Par ailleurs, le sommet de la pile doit être une parenthèse ouvrante de même nature que la parenthèse fermante rencontrée.
  Si les deux conditions suivantes sont vérifiées, on dépile.

Puisque l'on souhaite gérer des parenthèses de divers types (parenthèses, crochets et accolades), on va commencer par créer deux fonctions renvoyant `True` lorsque l'on aura affaire à une parenthèse ouvrante, respectivement fermante.

```
def OpenPar(c):
     return c == '(' or c == '[' or c == '{'

def ClosePar(c):
     return c == ')' or c == ']' or c == '}'
```

Puisqu'on aura besoin de comparer une parenthèse fermante à son éventuelle parenthèse ouvrante associée, on a intérêt à utiliser une fonction `OpeningPar(oc)` qui, à partir de la parenthèse ouvrante « `oc` » fournie en argument, renvoie la parenthèse fermante associée.

```
def OpeningPar(oc):
     if oc == ')':
          return '('
     if oc == ']':
          return '['
     if oc == '}':
          return '{'
```

A partir de là, on aura la fonction `CheckPar` suivante :

```
def CheckPar(string):
     OpenParStack = stack_create()
     for i in range(len(string)):
          if OpenPar(string[i]):
               stack_push(OpenParStack,string[i])
          elif ClosePar(string[i]):
               if stack_isempty(OpenParStack)
                         or stack_peek(OpenParStack) !=
                              OpeningPar(string[i]):
                    print('Parenthésage incorrect.')
                    return
               else :
                    stack_pop(OpenParStack)
     if stack_isempty(OpenParStack):
          print('Parenthésage correct.')
          return
     elif
          print('Parenthésage incorrect (nombre de
               parenthèses ouvrantes en excès).')
          return
```

Files

Exercice N°5 – Permutations circulaires

Reprendre les exercices 1 et 3 de la partie « Piles » mais en traitant cette fois le cas d'une file.

Copie

On a immédiatement la fonction :

```
def queue_copy(q):
     q2 = queue_create()
     if len(q) != 0:
          for i in range(len(q)):
               e = queue_out(q)
               queue_in(q2,e)
     return(q2)
```

Permutations circulaires successives

La manipulation avec une file est beaucoup plus aisée qu'avec une pile puisqu'une permutation circulaire consiste simplement à sortir un élément et à le replacer aussitôt dans la file. Pour n permutations successives, il suffit d'effectuer les deux opérations précédentes n fois dans une boucle. D'où la fonction :

```
def queue_circperm(q,n):
n = n%len(q)
cpq = queue_copy(q)
if n != 0:
     for i in range(n):
          x = queue_out(s)
          queue_in(cpq,x)
return cpq
```