Chapitre 1 : Structures de données avancées

Les Piles ~ Corrigé 1.2

Exercice 1 :

Solution 1 : Mettre en œuvre une pile avec une liste initialisée avec 40 valeurs égales à **None** .

```
1.    #pile.py
2.
3.    NMAX = 40
4.
5.    def creer_pile():
6.        return [None]*NMAX #2
7.
8.    def pile_vide(p):
9.        return p.count(None) == NMAX
10.
11.   def pile_pleine(p):
12.       return p.count(None) == 0
13.
14.   def empiler(p,x):
15.       if not pile_pleine(p):
16.           p[p.index(None)]=x
17.       else:
18.           raise Exception("Pile saturée")
19.
20.   def depiler(p):
21.       assert not pile_vide(p), "Pile vide"
22.       pos =  -1 if pile_pleine(p) else L.index(None)-1
23.       elt = p[pos]
24.       p[pos] = None
25.       return elt
26.
27.   def sommet(p):
28.       assert not pile_vide(p), "Pile vide"
29.       return p[-1] if pile_pleine(p) else p[L.index(None)-1]
30.
31.   def afficher(p):
32.       p1 = copy(p)
33.       while not pile_vide(p1):
34.           print(depiler(p1))
35.
36.   def depilerKelt(p,k):
37.       while not pile_vide(p) and k>0:
38.           depiler(p)
39.           k-=1
40.
41.
42.   def deplacer(p1,p2):
43.       while not pile_vide(p1):
44.           empiler(p2,depiler(p1))
45.
46.
47.
```

```
48.   def inverser_pile(p):
49.       p1,p2 = creer_pile(),creer_pile()
50.       deplacer(p,p1)
51.       deplacer(p1,p2)
52.       deplacer(p2,p)
53.
54.
55.   def depilerJusqua(p,e):
56.       while not est_vide(p) and sommet(p) != e:
57.           depiler(p)
58.
59.   def appartient(p,x):
60.       p1 = p.copy()
61.       depilerJusqua(p1,x)
62.       return False if pile_vide(p1) else True
```

Solution 2 : Mettre en œuvre une pile avec une liste initialement vide.

```
1.    #pile.py
2.
3.    NMAX = 40
4.    from copy import copy,deepcopy
5.
6.    def creer_pile():
7.        return []
8.
9.    def taille(p):
10.       return len(p)
11.
12.   def pile_vide(p):
13.       return taille(p) == 0
14.
15.   def empiler(p,x):
16.       if len(p) < 40:
17.           p.append(x)
18.       else:
19.           #raise : déclencher une erreur
20.           raise Exception ("Erreur: Pile saturée")
21.
22.   def empiler(p, x):
23.       assert taille(p)< NMAX, "Pile pleine"
24.       p.append(x)
25.
26.   def depiler(p):
27.       assert not pile_vide(p), "Erreur : pile vide"
28.       return p.pop() #p.pop(-1)
29.
30.   def sommet(p):
31.       assert not pile_vide(p), "Erreur: Pile vide"
32.       return p[-1]
33.
34.
35.
```

```
#Solution 1
def afficher(p):
    p1 = p.copy()
    while not pile_vide(p1): print(depiler(p1))

#Solution 2
def afficher(p) :
    temp = creer_pile()

    while not est_vide(p):
        elt = depiler(p)
        print(elt)
        empiler(temp,elt)

    while not est_vide(temp):
        empiler(p,depiler(temp))

#Solution 1
def depilerKelt(p,k):
    for i in range(k):
        if taille(p) > 0:
             depiler(p)
        else:
            break

#Solution 2
def depilerKelt(p,k):
    if k>= len(p):
        p = creer_pile() #[]
    else:
        for i in range(k):
            depiler(p)


#solution 3
def depilerKelt(p,k):
    while not pile_vide(p) and k>0:
        depiler(p)
        k-=1

#Solution 1
def inverser_pile(p):
    p1,p2 = creer_pile(),creer_pile()

    while not pile_vide(p):
        empiler(p1,depiler(p))

    while not pile_vide(p1):
        empiler(p2,depiler(p1))

    while not pile_vide(p2):
        empiler(p,depiler(p2))
```

```
92.   #Solution 2
93.   def deplacer(p1,p2):
94.       while not pile_vide(p1):
95.           empiler(p2,depiler(p1))
96.
97.
98.   def inverser_pile(p):
99.       p1,p2 = creer_pile(),creer_pile()
100.      deplacer(p,p1)
101.      deplacer(p1,p2)
102.      deplacer(p2,p)
103.
104.  #Solution 3
105.  def inverser_pile(p):
106.      p1 = deepcopy(p)
107.      p = creer_pile()
108.      while not pile_vide(p1):
109.          empiler(p,depiler(p1))
110.
111.  #Solution 1
112.  def depilerJusqua(p,e):
113.      while not est_vide(p) and sommet(p) != e:
114.          depiler(p)
115.
116.  #Solution 2
117.  def depilerJusqua(p,x):
118.      if appartient(p,x):
119.          while sommet(p) != x:
120.              depiler(p)
121.      else:
122.          while not pile_vide(p):
123.              depiler(p)
124.
125.  #Solution 1
126.  def appartient(p,elt) :
127.      p1 = creer_pile()
128.      while not est_vide(p) and sommet(p) != elt:
129.          empiler(p1,depiler(p))
130.      if est_vide(p):
131.          resultat = False
132.      else:
133.          resultat = True
134.      while not est_vide(p1):
135.          empiler(p,depiler(p1))
136.      return resultat
137.
138.  #Solution 2
139.  def appartient(p,x):
140.      p1 = copy(p)
141.      while not pile_vide(p1) and sommet(p1)!=x:
142.          depiler(p1)
143.      if pile_vide(p1):
144.          return False
145.      return True
146.
147.
```

```
148.  #Solution 2
149.  def appartient(p,x):
150.      p1 = p.copy()
151.      depilerJusqua(p1,x)
152.      return False if pile_vide(p1) else True
```

Exercice 2 :

```
1.   from pile import *
2.
3.   def conversion(nb10):
4.
5.       res = '0b' if nb10 else '0b0
6.
7.       p = creer_pile()
8.       while nb10 != 0:
9.           empiler(p, nb10%2)
10.          nb10 = nb10 // 2
11.
12.      while not est_vide(p):
13.          res += str(depiler(p))
14.
15.      return res
16.
17.  #test
18.  print(conversion(27))
```

Exercice 3 :

Question 1 : Analyser si une expression arithmétique est bien parenthésée.

```
1.    from pile import *
2.
3.    def analyse_parenthese(ch) :
4.        p = creer_pile()
5.        for c in ch :
6.            if c == "(" :
7.                empiler(p,c)
8.            elif c == ")" :
9.                if est_vide(p):
10.                   print("Il manque une ou des parenthèse(s) ouvrante(s)")
11.                   return False
12.               else :
13.                   depiler(p)
14.       if est_vide(p):
15.           print("Placement des parenthèses correct")
16.           return True
17.       else :
18.           print("Il manque une ou des parenthèses fermantes")
19.           return False
20.
21.
22.   #test
```

```
23.    ch="(3+2))*2"
24.    print(ch)
25.    analyse_parenthese(ch)
26.
27.    ch="((3+2)*2"
28.    print(ch)
29.    analyse_parenthese(ch)
30.
31.    ch="(3+2)*2"
32.    print(ch)
33.    analyse_parenthese(ch)
```

Question 2 : Donner les trois représentations d'une expression arithmétique :

Expression 1 :

La représentation **infixée** : **2 + (3 - (5 + 1) × 2)**

La représentation **préfixée** : **+ 2 - 3 × + 5 1 2**

La représentation **postfixée** : **2 3 5 1 + 2 × - +**

Expression 2 :

La représentation infixée : **/ 2 × - 3 + 4 5 6**

La représentation préfixée : **2 / ((3 - (4 + 5)) × 6)**

La représentation postfixée : **2 3 4 5 + - 6 × /**

Question 3 : Calculer la valeur numérique d'une expression arithmétique.

```
1.    from pile import *
2.    import string
3.
4.    def calc_exp_arith(ch):
5.
6.        p = creer_pile()
7.
8.        for i in range(len(ch)):
9.            if ch[i] in string.digits :
10.               empiler(p, int(ch[i]))
11.           else:
12.               n1 = depiler(p)
13.               n2 = depiler(p)
14.               if   ch[i] == '+' : empiler(p, n1 + n2)
15.               elif ch[i] == '-' : empiler(p, n2 - n1)
16.               elif ch[i] == '*' : empiler(p, n1 * n2)
17.               elif ch[i] == '/' : empiler(p, n2 / n1)
18.
19.       return depiler(p)
20.
21.   #test
22.   print(calc_exp_arith("27*5+"))
```

Exercice 4 :

```
1.  #Q1
2.  def init_pile(p, n):
3.      assert type(n)== int and n > 0
4.      p.append(1)
5.      for i in range(n): p.append(0)
6.
7.  #Q2
8.  def pile_vide(p):
9.      return p[0] == 1
10.
11.
12. #Q3
13. def pile_pleine(p):
14.     return p[0] == len(p)
15.
16. #Q4
17. def empiler(p, x):
18.     assert not pile_pleine(p), "Pile pleine"
19.     p[p[0]] = x
20.     p[0] += 1
21.
22. #Q5
23. def depiler(p):
24.     assert not pile_vide(p), "Pile vide"
25.     p[0] -= 1
26.     return p[p[0]]
27.
28. #Q6
29. def copier(p):
30.     p1 = list(); init_pile(p1, len(p)-1)
31.     p2 = list(); init_pile(p2, len(p)-1)
32.
33.     while not pile_vide(p):
34.         empiler(p1,depiler(p))
35.
36.     while not pile_vide(p1):
37.         x = depiler(p1)
38.         empiler(p,x)
39.         empiler(p2,x)
40.
41.     return p2
42.
43. #Q7
44. def echanger(p):
45.     x1, x2 = depiler(p), depiler(p)
46.     empiler(p,x1); empiler(p,x2)
47.
```

```python
#Q8
def enroler(p,n):
    assert not pile_vide(p), "Pile vide"
    assert type(n)== int and n > 0
    assert len(p) > n

    x = depiler(p)

    p1 = list(); init_pile(p1, len(p)-1)
    for i in range(n-1):
        empiler(p1,depiler(p))

    empiler(p,x)
    for i in range(n-1):
        empiler(p,depiler(p1))

#Q9
def superposition(p1, p2):
    n = len(p1) + len(p2) - 2
    p3 = list()
    init_pile(p3, n)


    p21 = copier(p2)
    p_temp = list()
    init_pile(p_temp, len(p21)-1)

    while not pile_vide(p21):
        empiler(p_temp,depiler(p21))

    while not pile_vide(p_temp):
        empiler(p3,depiler(p_temp))



    p11 = copier(p1)
    p_temp = list()
    init_pile(p_temp, len(p11)-1)

    while not pile_vide(p11):
        empiler(p_temp,depiler(p11))

    while not pile_vide(p_temp):
        empiler(p3,depiler(p_temp))

    return p3

```

```python
#Q10
def inserer(x,p):
    #La pile p est supposée triée en ordre croissant
    #le sommet de p est le plus grand element
    assert not pile_pleine(p),"Pile pleine, impossible d'insérer "+str(x)

    if pile_vide(p):
        empiler(p,x)
    elif p[p[0]-1] <= x :
        empiler(p,x)
    else:
        p_temp = list()
        init_pile(p_temp, len(p)-1)

        while not pile_vide(p) and p[p[0]-1] > x:
            sommet = depiler(p)
            empiler(p_temp,sommet)

        empiler(p,x)

        while not pile_vide(p_temp):
            empiler(p,depiler(p_temp))

#Q11
def tri_insertion_en_place(p) :
    #copier p
    p_temp = copier(p)

    #vider p
    while not pile_vide(p):
        depiler(p)

    #trier p par insertion
    while not pile_vide(p_temp):
        sommet = depiler(p_temp)
        inserer (sommet, p)

#test
if __name__ == '__main__':
    n = 5
    p = list()
    init_pile(p, n); print(p)
    print("Pile vide: ", pile_vide(p))

    #Q4 : empiler
    try:
        for i in range(6):
            empiler(p,i)
    except:
        print("Pile saturée")

    print("etat de la pile: ",p)

    #Q4 : depiler
    print("sommet :", depiler(p))
    print("Etat de la pile: ",p)
```

```
151.
152.     #Q5: copier
153.     print("p =", p)
154.     print("copie de p=", copier(p))
155.     print("p =", p)
156.
157.     #Q7: echanger
158.     echanger(p)
159.     print("Après echanger =", p)
160.
161.     #Q8: enroler
162.     enroler(p,3)
163.     print("Après enroler =", p)
164.
165.     #Q9: superposition
166.     p1 = [ 4 , 7 , 8 , 12 , 42 , 10 ]
167.     p2 = [ 6 , 7 , 42 , 8 , 12 , 10 ]
168.     p3 = superposition(p1, p2)
169.     p31=[ 9 , 7 , 42 , 8 , 12 , 10 , 7 , 8 , 12 , 0 , 0 ]
170.     print("superposition : ",p3 == p31)
171.
172.     #Q10: inserer
173.     p = [ 6 , 7 , 8 , 10 , 12 , 42, 0 , 0 , 0 , 0 ]
174.     inserer(9,p);inserer(50,p);inserer(2,p);
175.     p1 = [ 9 , 2 , 7 , 8 , 9 , 10 , 12 , 42 , 50 , 0 ]
176.     print("insertion : ",p==p1)
177.
178.     #Q11: tri_insertion_en_place
179.     p = [ 11 , 4 , 41 , 3 , 4 , 5 , 40 , 21 , 33 , 50 , 30 ]
180.     tri_insertion_en_place(p)
181.     p1 = [ 11 , 3 , 4 , 4 , 5 , 21 , 30 , 33 , 40 , 41 , 50 ]
182.     print("tri_insertion_en_place : ",p==p1)
183.
```