Piles prof

September 19, 2020

1 Définition d'une pile

Cette partie s'interesse à une structure de donnée linéaire Last In First Out (LIFO), les piles. Pour se représenter une pile on peut s'appuyer sur l'image d'une pile d'objet empilé les uns sur les autres.

[1]:
```
from IPython.display import Image
Image("img/pile_assiette.jpg")
```

[1]:

Par exemple l'image de cette pile d'assiette constitue une pile.

Cette pile a pour propriété d'être une structure de donnée à une dimension, tous les objets sont rangés suivant le même axe vertical. On parle ainsi de structure de donnée linéaire.

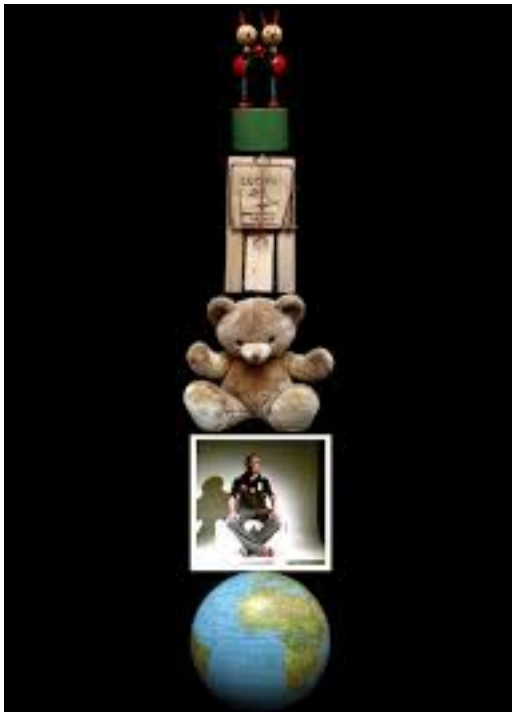
Et on remarque que l'on a accès uniquement à l'assiette en haut de la pile, toutes les autres assiettes sont coincés dans la pile.

Enfin l'assiette à laquelle on a accès est la dernière assiette posée sur la pile, il s'agit d'un ordre "dernier arrivé, premier sortie" ou Last In, First Out (LIFO).

Ceci définit la strucutre de donnée de type pile pour tout les objets possibles

[2]:
```
from IPython.display import Image
Image("img/pile_quelconque.jpg")
```

[2]:



de même en python on peut stocké dans une pile toute forme d'objet, par contre on doit garder la même structure de la base de donnée linéaire LIFO.

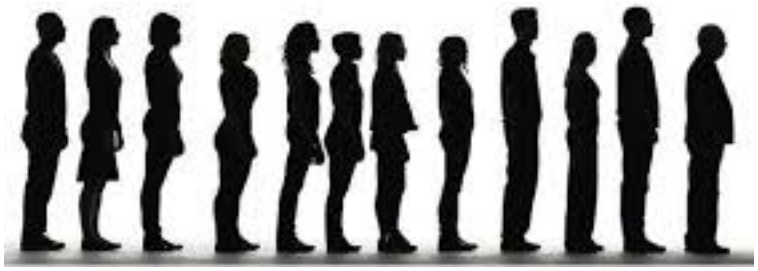
[3]:
```
from IPython.display import Image
Image("img/pile_python.jpg")
```

[3]:

D'autre structure de donnée linéaire existe, notamment des structures dites de file où comme dans une file d'attente c'est le premier arrivé qui est le premier sortie.

[4]:
```
from IPython.display import Image
Image("img/file.jpg")
```

[4]:



Nous rencontrons ce type de structure de donné dans différentes situations. De manière très régulière avec les fonctions “reculer d’une page” et “avancer d’une page” dans un navigateur ou avec les fonction “annuler” et “rétablir” d’un logiciel. De manière indirecte lors de l’appel de fonction récursive que l’on étudiera par la suite.

- Justifier pourquoi les fonctions “reculer d’une page” et “avancer d’une page” constituent bien une pile.

Quels sont les avantages et inconvénients de la pile ?
avantages : On veut stocker des éléments dont le nombre est variable et/ou inconnu à l’avance On peut ou on doit se contenter de n’avoir accès qu’au dernier élément stocké
inconvénients: si on veut pouvoir accéder à n’importe quel élément, il vaut mieux utiliser un tableau

- Dans quel cas faut il utiliser une pile, un tableau, une file, . . .
  - Un répertoire téléphonique-
  - L’historique des actions effectuées dans un logiciel
  - Compatibiliser les pièces ramassées et dépensées par un personnage
  - Ranger des dossiers à traiter

pour un répertoire téléphonique on utilise plutôt un tableau car on veut pouvoir accéder à n’importe quel élément, n’importe quand
pour l’historique des actions une pile est particulièrement adaptée, on accède à la dernière action effectuée, puis la précédente et ainsi de suite
pour compter les pièces ramassés et dépensés on ne se soucie pas de l’ordre dans lequel les pièces ont été ramassées, seul un nombre suffit, de même pour les pièces dépensées
pour ranger des dossiers une structure de pile implique que le dernier dossier serait toujours traité en premier, donc certains dossiers au fond de la pile ne seront jamais traité, une file serait plus adaptée.

2 Réalisation concrète d'une pile

Nous allons maintenant étudier les fonctions disponibles sur les piles.

En gardant la même image que pour une pile d'objet on peut :

créer une pile vide, on fait de la place sur une étagère pour y empiler des objets

ajouter un élément, on empile un objet sur la pile existante, cette fonction est communément appelée "push"

retirer le premier élément de la pile si la pile n'est pas vide et le renvoyer, on dépile l'objet en haut de la pile, cette fonction est communément appelée "pop"

lire le sommet de la pile, identifier l'objet en haut de la pile sans le dépiler

tester si la pile est vide, identifier s'il y a au moins un objet dans la pile

ces fonctions de base sont les seules disponibles pour manipuler ces structures de données

2.1 Cas d'une pile de taille fixée

Commençons par écrire les fonctions de base pour une pile dont la taille maximale est fixée à sa création, on parle de pile bornée ou de capacité finie.

- écrire une fonction creer_pile(c) qui prend en argument la capacité c de la pile et qui retourne une pile vide de taille fixée et dont le premier élément indique la taille actuelle de la pile, ici 0. on pourra utiliser une liste de taille fixée c+1 et dont les éléments vides d'indices 1 à c sont remplis par None et dont l'élément d'indice 0 indique la taille de la liste.

[5]:
```
def creer_pile(c):
    p = (c + 1) * [None]
    p[0] = 0
    return p
```

- utiliser votre fonction pour créer une pile de taille 10 puis utiliser la fonction print() pour l'afficher

[6]:
```
ma_premiere_piles = creer_pile(10)

print(ma_premiere_piles)
```

```
[0, None, None, None, None, None, None, None, None, None, None]
```

- écrire une fonction empiler(p, v) qui prend en argument une pile p et un élément v, vérifie que la liste n'est pas pleine, actualise la taille de la pile, et empile l'élément v au sommet de la pile donc dans le premier espace vide disponible de la liste

[7]:
```
def empiler(p, v):
    n = p[0]
    assert n < len(p)-1
    n = n + 1
    p[0] = n
    p[n] = v
```

- empiler dans l'ordre les éléments suivant dans votre pile 3, 1, 'chat', [1, 2, 3] et afficher la pile obtenue

[8]:
```
empiler(ma_premiere_piles,1)

empiler(ma_premiere_piles,'chat')

empiler(ma_premiere_piles,[1,2,3])

print(ma_premiere_piles)
```

```
[3, 1, 'chat', [1, 2, 3], None, None, None, None, None, None, None]
```

- écrire une fonction depiler(p) qui prend en argument une pile p, vérifie que la pile n'est pas vide, actualise la taille de pile, retire l'élément au sommet de la pile, et le retourne

[9]:
```
def depiler(p):
    n = p[0]
    assert n > 0
    p[0] = n - 1
    a_retourner = p[n]
    p[n] = None
    return a_retourner
```

- utiliser votre fonction pour dépiler un élément de votre pile et afficher l'élément retiré et la pile sans cet élément

[10]:
```
element = depiler(ma_premiere_piles)

print(element)

print(ma_premiere_piles)
```

```
[1, 2, 3]
[2, 1, 'chat', None, None, None, None, None, None, None, None]
```

- écrire une fonction taille(p) qui prend en argument une pile p, et retourne la taille de la pile p

[11]:
```
def taille(p):
    return p[0]
```

- utiliser votre fonction pour connaitre la taille actuelle de votre pile

[12]:
```
print(taille(ma_premiere_piles))
```

```
2
```

- écrire une fonction est_vide(p) qui prend en argument une pile p, et retourne un booléen VRAI si la pile est vide et FAUX si la pile n'est pas vide

[13]:
```
def est_vide(p):
    return taille(p) == 0
```

- tester si votre pile est vide

[14]:
```
print(est_vide(ma_premiere_piles))
```

```
False
```

- écrire une fonction sommet(p) qui prend en argument une pile p, vérifie si la pile est vide, et retourne l'élément au sommet de la pile sans le dépiler

[15]:
```
def sommet(p):
    assert taille(p) > 0
    return p[p[0]]
```

- utiliser cette fonction pour afficher le sommet de votre pile

[16]:
```
print(sommet(ma_premiere_piles))
```

```
chat
```

2.2 Cas de pile non bornées

Toutes les fonctions que nous avons définie implique de connaître à l'avance la capacité maximale de la pile dont nous aurons besoin, ce qui limite les possibilités d'utilisation de cette pile. L'objet liste [..., ..., ...] sous python possède en réalité toutes les méthodes pour réaliser les fonctions d'une pile sans prédéfinir sa capacité.

- sachant que la commande [] définit une liste vide, réécrire la fonction creer_pile et tester cette fonction

[17]:
```
def creer_pile():
    return []

une_pile_non_bornee = creer_pile()

print(une_pile_non_bornee)
```

```
[]
```

- sachant que la méthode p.append(v) ajoute l'élément v à la fin de la liste p, réécrire et tester la fonction empiler(p,v)

[18]:
```
def empiler(p, v):
    p.append(v)
```

```
empiler(une_pile_non_bornee,'chat')

print(une_pile_non_bornee)
```

```
['chat']
```

- sachant que la fonction len(p) retourne le nombre d'élément dans une liste, réécrire et tester la fonction taille(p) et est_vide(p)

[19]:
```
def taille(p):
    return len(p)

print(taille(une_pile_non_bornee))

def est_vide(p):
    return taille(p) == 0

print(est_vide(une_pile_non_bornee))
```

```
1
False
```

- sachant que la méthode p.pop() retire et retourne le dernier élément de la liste p, réécrire et tester la fonction depiler(p)

[20]:
```
def depiler(p):
    assert len(p) > 0
    return p.pop()

print(depiler(une_pile_non_bornee))

print(une_pile_non_bornee)
```

```
chat
[]
```

- sachant que l'indice -1 d'une liste est l'indice du dernier élément d'une liste, réécrire et tester la fonction sommet(p)

[21]:
```
def sommet(p):
    assert len(p) > 0
    return p[-1]

empiler(une_pile_non_bornee,'chat')

print(sommet(une_pile_non_bornee))
```

```
chat
```

3 Quelques fonctions manipulant des piles

En se basant uniquement sur les fonctions disponibles pour les piles nous allons programmer quelques fonctions qui manipulent les piles.

- écrire une fonction renverser(p) qui renverse une pile donnée

[22]:
```
def renverser(p):
    p1 = creer_pile()
    p2 = creer_pile()
    while not(est_vide(p)):
        empiler(p1,depiler(p))
    while not(est_vide(p1)):
        empiler(p2,depiler(p1))
    while not(est_vide(p2)):
        empiler(p,depiler(p2))
```

- renverser la pile [3, 1, 'chat', [1, 2, 3]] en [[1, 2, 3], 'chat', 1, 3]

[23]:
```
ma_pile = creer_pile()
empiler(ma_pile,3)
empiler(ma_pile,1)
empiler(ma_pile,'chat')
empiler(ma_pile,[1, 2, 3])
print(ma_pile)
renverser(ma_pile)
print(ma_pile)
```

```
[3, 1, 'chat', [1, 2, 3]]
[[1, 2, 3], 'chat', 1, 3]
```

- écrire une procédure affichage(p) d'affichage d'une pile qui affiche les éléments d'un pile en partant du fond jusqu'au sommet

[24]:
```
def affichage(p):
    renverser(p)
    while not(est_vide(p)):
        print(depiler(p))
```

- tester la procédure d'affichage sur votre pile

[25]:
```
affichage(ma_pile)
```

```
[1, 2, 3]
chat
1
3
```

- écrire une fonction superposer(p1,p2) qui superpose la pile p2 au-dessus de la pile p1

[26]:
```
def superposer(p1,p2):
    renverser(p2)
    while not(est_vide(p2)):
        empiler(p1,depiler(p2))
```

- emplier la pile [‘chat’, [1, 2, 3]] au-dessus de la pile[3, 1]

[27]:
```
pile_initiale = creer_pile()
empiler(pile_initiale,3)
empiler(pile_initiale,1)
print(pile_initiale)
pile_a_ajouter = creer_pile()
empiler(pile_a_ajouter,'chat')
empiler(pile_a_ajouter,[1,2,3])
print(pile_a_ajouter)
superposer(pile_initiale,pile_a_ajouter)
print(pile_initiale)
```

```
[3, 1]
['chat', [1, 2, 3]]
[3, 1, 'chat', [1, 2, 3]]
```

4 Exercice: Traitement d’une expression parenthésée

Etant donnée une chaîne de caractères ne contenant que des caractères ‘(’ et ‘)’, déterminer s’il s’agit d’un mot bien parenthésé. Un mot bien parenthésé est soit le mot vide, soit la concaténation de deux mots bien parenthésés, soit un mot bien parenthésé mis entre parenthèses. Ainsi, les trois mots ‘’,’()()’ et ‘(())()’ sont bien parenthésés. À l’inverse, les mots ‘(()’, ‘())’ ou encore ‘)(’ ne le sont pas. On se propose de plus d’indiquer, pour chaque parenthèse ouvrante, la position de la parenthèse fermante correspondante. Ainsi, pour le mot ‘(())()’, on donnera les couples d’indices (0, 3), (1, 2) et (4, 5).

[28]:
```
def parentheses(s):
    p = creer_pile()          # on créé une pile p
    for i in range(len(s)):   # on parcours tout les caractères du 'mot' de la␣
↪gauche vers la droite
        if s[i] == '(':       # si on rencontre une parenthèse ouvrante, on␣
↪vient d'ouvrir une parenthèse
            empiler(p, i)     # alors on note dans la liste l'indice de la␣
↪parenthèse ouvrante
        else:                 # sinon c'est que c'est une parenthèse fermante
            if est_vide(p):   # si la pile est vide ça veut dire qu'on n'a pas␣
↪ouvert de parenthèse
                return False # donc le mot est mal parenthésé
            j = depiler(p)    # si la pile n'est pas vide, on récupère l'indice␣
↪de la dernière parenthèse ouverte
```

```
            print((j, i))    # et on affiche les indices des parenthèse␣
↪ouvrante/fermante correspondante
    return est_vide(p)       # une fois le mot parcourus, on vérifie qu'il ne␣
↪reste pas de parenthèse ouverte
```

[29]:
```
parentheses('()(()())')
```

```
(0, 1)
(3, 4)
(5, 6)
(2, 7)
```

[29]: True

[30]:
```
parentheses('()(()()))(')
```

```
(0, 1)
(3, 4)
(5, 6)
(2, 7)
```

[30]: False

[31]:
```
parentheses('())(()())')
```

```
(0, 1)
```

[31]: False

5 Exercice: Tour de magie de Gilbreath.

Écrire une fonction couper qui prend une pile et la coupe en enlevant de son sommet un certain nombre d'éléments (tiré au hasard) qui sont renvoyés dans une seconde pile. Exemple : si la pile initiale est [1, 2, 3, 4, 5] et si le nombre d'éléments retirés vaut 2, alors la pile ne contient plus que [1, 2, 3] et la pile renvoyée contient [5, 4].

[32]:
```
from random import *

def couper(p):
    p_prime = creer_pile()
    taille_de_p = taille(p)
    nombre_element_retire = randint(0,taille_de_p)
    for i in range(nombre_element_retire):
        element = depiler(p)
        empiler(p_prime,element)
    return p_prime
```

[33]:
```
pile_initiale = creer_pile()

for i in range(5):
    empiler(pile_initiale,i+1)

print(pile_initiale)

pile_renvoyée = couper(pile_initiale)

print(pile_renvoyée)

print(pile_initiale)
```

```
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
[]
```

Écrire une fonction melange qui prend en arguments deux piles et qui mélange leurs éléments 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. Exemple : un mélange possible des piles [1, 2, 3] et [5, 4] est [3, 2, 4, 1, 5]. Note : à l'issue du mélange, les deux piles de départ sont donc vides.

[34]:
```
def melange(p1,p2):
    p_melange = creer_pile()
    while not(est_vide(p1)):
        if est_vide(p2):
            empiler(p_melange,depiler(p1))
        else :
            if randint(1,2) == 1 :
                empiler(p_melange,depiler(p1))
            else :
                empiler(p_melange,depiler(p2))

    while not(est_vide(p2)):
        empiler(p_melange,depiler(p2))

    return p_melange
```

[35]:
```
p1 = [1,2,3]
p2 = [5,4]
print(melange(p1,p2))
```

```
[3, 2, 4, 1, 5]
```

Construire un paquet de cartes en empilant n fois les mêmes k cartes (par exemple, pour un paquet de 32 cartes, on empile n = 16 paquets de paires rouge/noir). Couper alors le paquet avec la fonction couper ci-dessus, puis mélanger les deux paquets obtenus à l'aide de la fonction

melange. On observe alors que le paquet final contient toujours n blocs des mêmes k cartes (même si ces dernières peuvent apparaître dans un ordre différent au sein de chaque bloc). Sur l'exemple des 16 paquets rouge/noir, on obtient toujours 16 paquets rouge/noir ou noir/rouge.

[36]:
```
paquet_carte = 16*['rouge','noir']
#print('paquet initial =',paquet_carte)
paquet_coupe = couper(paquet_carte)
#print('paquet coupé =',paquet_coupe)
#print('paquet coupé =',paquet_carte)
paquet_final = melange(paquet_carte,paquet_coupe)
print('paquet mélangé =',paquet_final)
```

```
paquet mélangé = ['rouge', 'noir', 'noir', 'rouge', 'rouge', 'noir', 'rouge',
'noir', 'rouge', 'noir', 'rouge', 'noir', 'rouge', 'noir', 'noir', 'rouge',
'rouge', 'noir', 'noir', 'rouge', 'rouge', 'noir', 'noir', 'rouge', 'rouge',
'noir', 'rouge', 'noir', 'rouge', 'noir', 'rouge', 'noir']
```

[37]:
```
paquet_carte = 4*['roi','dame','cavalier','valet']
#print('paquet initial =',paquet_carte)
paquet_coupe = couper(paquet_carte)
#print('paquet coupé =',paquet_coupe)
#print('paquet coupé =',paquet_carte)
paquet_final = melange(paquet_carte,paquet_coupe)
print('paquet mélangé =',paquet_final)
```

```
paquet mélangé = ['cavalier', 'dame', 'valet', 'roi', 'roi', 'dame', 'cavalier',
'valet', 'roi', 'dame', 'cavalier', 'valet', 'roi', 'dame', 'cavalier', 'valet']
```