LES PILES

Informatique PSI

2021-2022

Résumé

Une des problématiques de la programmation est le stockage des données dans des structures adaptées aux algorithmes qui les utilisent.
Une structure de donnée est caractérisée par les opérations qu'elle permet d'effectuer :

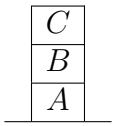
- accéder à un élément ;
- parcourir les éléments ;
- introduire ou retirer un élément.

Jusqu'à présent, les seules structures de données que nous avons vues sont les les listes. Nous présentons ici la structure de *pile* et quelques unes de ses applications.

Table des matières

1 Notion de pile

La structure de pile correspond à l'image d'une pile d'assiettes. On peut poser un élément sur la pile ou prendre le dernier élément posé appelé *sommet* de la pile.
Dans l'illustration suivante on a empilé trois éléments : A, puis B, puis C.



On peut soit ajouter un nouvel élément D au-dessus de C, on dit qu'on *empile* D, soit retirer C du sommet de la pile, on dit qu'on *dépile* C. Ainsi, pour accéder à A, il faut d'abord dépiler C, puis B.
Une pile se caractérise donc par "dernier arrivé, premier sorti" ou encore en anglais "Last In, First Out", en abrégé LIFO.
Les applications des piles sont nombreuses. Parmi elles, on peut citer :

- ✏ les piles d'exécution des fonctions récursives vues dans le chapitre récursivité ;
- ✏ la gestion d'un bouton "retour à la page précédente" dans un navigateur Web ;
- ✏ l'analyse syntaxique et évaluation d'expressions arithmétiques ;
- ✏ le parcourt d'un labyrinthe ;
- ✏ ...

Remarque 1.1. Il existe aussi une structure de *file* où le premier arrivé est le premier sorti, en anglais "First In, First Out", en abrégé FIFO. Cette structure modélise par exemple une file d'attente à un guichet.

2 Réalisation d'une structure de pile

2.1 Interface

L' *interface* d'une structure est l'ensemble des applications qui agissent sur cette structure. Pour réaliser une structure de pile, il faut définir les fonctions suivantes :

- `creer_pile()` : renvoie une pile vide ;
- `est_vide(p)` : indique si la pile `p` est vide.
- `empiler(p,e)` : empile l'élément `e` sur la pile `p` ;
- `depiler(p)` : dépile l'élément situé au sommet de la pile `p`, si celle-ci n'est pas vide, et renvoie cet élément.

On peut définir d'autres fonctions qui ne modifient pas la pile telles que :

- `taille(p)` : renvoie le nombre d'éléments de la pile `p`.
- `sommet(p)` : renvoie la valeur du sommet de la pile `p`, sans le dépiler, si la pile n'est pas vide.
- `afficher(p)` : affiche les éléments de `p` les uns au-dessus de autres. Par exemple, si `p` est représentée par la liste `[4,5,8,1]`, `afficher(p)` affichera :

```
1
8
5
4
```

Exemple 2.1. On veut utiliser une pile qui gère les actions "aller à une nouvelle adresse" et "retour à la page précédente" d'un navigateur Web. On écrit les deux fonctions suivantes associées à ces deux actions :

```
def aller_a(nouvelle_url, url_courante, pile_url):
    empiler(pile_url, url_courante)
    return nouvelle_url
def retour(pile_url):
    nouvelle_url = depiler(pile_url)
    return nouvelle_url
```

Quelles sont, dans l'ordre, les pages consultées par l'utilisateur après l'exécution des opérations suivantes ? Indiquer l'état de la pile `pile_url` après chaque changement d'état.

```
pile_url = creer_pile()
url_courante = "home"
url_courante = aller_a("page 1",url_courante,pile_url)
url_courante = aller_a("page 2",url_courante,pile_url)
url_courante = aller_a("page 3",url_courante,pile_url)
url_courante = retour(pile_url)
url_courante = aller_a("page 4",url_courante,pile_url)
url_courante = retour(pile_url)
url_courante = retour(pile_url)
url_courante = aller_a("page 3",url_courante,pile_url)
url_courante = retour(pile_url)
url_courante = retour(pile_url)
```

2.2 TD : Implémentation de piles avec les listes Python

Un objet de type `list` de Python possède deux méthodes `append` et `pop` qui réalisent les opérations "empiler" et "dépiler". Le sommet de la pile est représenté par le dernier élément de la liste.

```
>>>L=[]
>>>L.append(1)
>>>L.append(2)
>>>L.append(4)
>>>L
[1,2,4]
>>>L.pop()
4
>>>L
[1,2]
>>>L.pop()
2
>>>L
[1]
>>>L.pop()
1
>>>L
[]
```

1. En utilisant les fonctionnalités des listes Python, écrire les fonctions `creer_pile` , `est_vide`, `empiler`, `depiler`, `taille` et `sommet`, caractérisant la structure de pile.
2. Tester ces fonctions sur l'exemple 2.1.

3 Exercices

Dans les exercices suivants, on dispose des fonctions de gestion d'une pile définies dans le TD 2.2 : `creer_pile` , `est_vide`, `empiler`, `depiler`, `taille`, `sommet` et `afficher`, et uniquement de ces fonctions.

1. Dans le programme suivant, `p` désigne une pile.

```
1  q = creer_pile()
2  while not est_vide(p):
3      empiler(q, depiler(p))
```

   (a) Que contiennent les piles `p` et `q` à la fin de l'exécution du programme suivant ?

   (b) Modifier ce programme pour qu'à la fin de l'exécution `q` contiennent les éléments de la pile `p` renversés et que la pile `p` reste inchangée.

2. **Analyse des mots bien parenthésés**

   Voici la définition d'un mot bien parenthésé :

   - le mot vide est bien parenthésé.
   - La concaténation de deux mots bien parenthésés est bien parenthésé.
   - Si le mot `m` est bien parenthésé, alors le mot `'(m)'` est bien parenthésé.

   Ainsi les mots '()' et '()(())' sont bien parenthésés. En revanche, les mots ')(' et et '(()' ne le sont pas.
   Voici un algorithme qui permet de décider si un mot est bien parenthésé. Il affiche de plus pour chaque parenthèse ouvrante, l'indice de la parenthèse fermante correspondante. Par exemple , pour le mot '()(())', il affichera les couples `(0,1),(2,5),(3,4)`.

   (a) On crée une pile vide `p` et une liste vide `lst_indices`.

   (b) On parcourt le mot de gauche à droite
   - si on lit une parenthèse '(', on empile son indice sur `p`.
   - si on lit une parenthèse droite d'indice `j`, alors si `p` n'est pas vide, on dépile l'indice `i` se trouvant au sommet de `p` et on ajoute le couple `(i,j)` à la liste `lst_indices` ; si `p` est vide, le mot n'est pas bien parenthésé.
   - si à la fin du parcours du mot, la pile `p` est vide alors le mot est bien parenthésé, alors on renvoie `lst_indices`, sinon il ne l'est pas.

1. Écrire en Python une fonction `parenthese(e)` traduisant cet algorithme pour un mot `e`. Tester cette fonction avec les mots `'(3+4)*9-(8+(9-7*9))'`, `'(3+4*9-(8+(9-7*9))'`, et `''`.
2. On ne souhaite pas que l'algorithme précédent renvoie la liste des couples d'indices, mais qu'il décide seulement si le mot est bien parenthésé. Modifier l'algorithme en conséquence. La structure de pile est-elle encore nécessaire ?
3. Écrire une fonction `parentheses_multiples` pour qu'elle vérifie si des mots contenant les symboles : `'(','(',')','[',']','{','}'` sont bien parenthésés (on n'affichera pas les couples d'indices comme dans la question 1).
   Tester cette fonction avec les mots `'(4x+3)-5[2(x+y)+5]+6'` et `'(4x+3)-52(x+y)+5]+6'`.

4. **Évaluation d'expressions arithmétiques**
On considères ici les expressions arithmétiques écrites avec les caractères `+`, `*` et les chiffres de `0` à `9` comme par exemple :
`'2+3*4'`, `'4*5+2+5*9+3'`,...
Les autres symboles sont interdits. On suppose dans la suite que l'expression est syntaxiquement correcte : elle commence et se termine par un chiffre et entre chaque chiffre il y a un symbole d'opération.
L'objectif est d'écrire une fonction qui évalue une telle expression en tenant compte des priorités de calculs (la multiplication est prioritaire sur l'addition.

> On utilise une pile d'entiers et on lit la chaine de caractères à évaluer de gauche à droite. À chaque caractère `c` lu :
>
> - Si `c` est un symbole d'opération, on l'empile.
> - Si `c` est un chiffre et que le sommet de la pile est le symbole `'*'`, on le dépile. On dépile ensuite le nombre `x` situé au sommet de la pile et on empile le produit de `x` par l'entier correspondant à `c`.
> - Si `c` est un chiffre et que le sommet de la pile est le symbole `'+'`, on empile l'entier correspondant à `c`.
>
> Une fois la chaine lue, on effectue la somme des éléments de la pile en la dépilant.

   (a) Écrire une fonction `evaluer(exp)` qui prend en argument une chaine de caractères `exp` représentant une expression arithmétique sans erreur de syntaxe et qui renvoie son évaluation.

   (b) Écrire une fonction `verif(exp)` qui vérifie si la chaine de caractères `exp` est syntaxiquement correcte.

5. **Résolution du problème des tours de Hanoï**
On veut utiliser une pile ici pour résoudre le problème des tours de Hanoï décrit dans le chapitre récursivité. En s'inspirant de la solution récursive du problème, compléter la fonction suivante pour qu'elle renvoie la liste des déplacements permettant de résoudre le problème (une solution prendra la forme d'une liste de couples d'entiers).

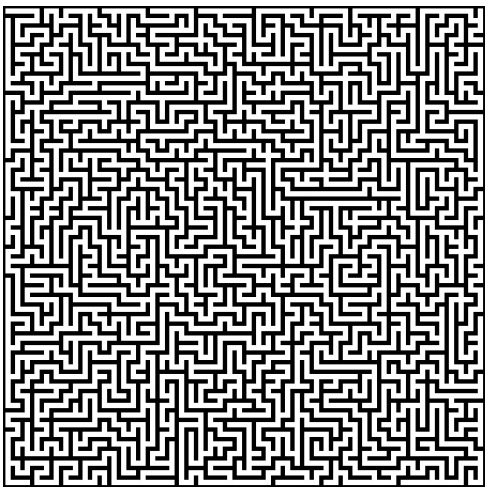
```
1  def hanoi(nb_anneaux,tour1,tour2,tour3):
2      p = creer_pile()
3      sol = []
4      empiler(p,(nb_anneaux,tour1,tour2,tour3))
5      while not est_vide(p):
6          q = depiler(p)
7          n, t1, t2, t3 = q
8          if n == 1 :
9              sol.append(.......)
10         else :
11             empiler(.................)
12             empiler(.................)
13             empiler(.................)
14     return sol
```

On devrait avoir par exemple :

```
>>>hanoi(3,1,2,3)
[(1, 3), (1, 2), (3, 2), (1, 3), (2, 1), (2, 3), (1, 3)]
```

4 TD : Construction d'un labyrinthe parfait

Un *labyrinthe* parfait est un labyrinthe dans lequel il existe un unique chemin entre deux points quelconques. Voici un exemple d'un labyrinthe 50×50.



Il s'agit d'écrire un algorithme générant aléatoirement un labyrinthe parfait de taille donnée $n \times m$. L'algorithme renvoie pour chaque case du labyrinthe la liste de ses cases voisines accessibles.

L'algorithme suivant utilise une pile. Les cases du labyrinthe seront modélisées par un tableau `L` de taille $n \times m$. On supposera que l'entrée du labyrinthe est la case $(0, 0)$ (en haut à gauche) et que sa sortie est la case (n, m) (en bas à droite), mais il est possible faire tout autre choix.

L'idée de l'algorithme est de parcourir aléatoirement toutes les cases du tableau `L` en partant de la case de départ. À chaque fois qu'une case est atteinte on la marque comme visitée. On peut passer d'une case à une de ses cases adjacentes qui n'a pas encore été visitée. Dans ce cas on relie ces deux cases. Si lors du parcours, une case n'a pas de cases adjacentes accessibles, on remonte le chemin jusqu'à trouver une case ayant des cases adjacentes accessibles. On remonte ainsi jusqu'à la première case. On admet que ceci garantit que toutes les cases ont été visitées.

L'algorithme utilise une pile qui mémorise le chemin par lequel on a accédé à la case courante visitée.

1. Créer un tableau `L` de taille $n \times m$ et initialiser chaque case du tableau avec la valeur `Faux` (aucune case n'a encore été visitée).
2. Créer un tableau `A` de taille $n \times m$. Ce tableau indiquera les cases reliées entre elles. Chaque case du tableau comportera la liste des cases adjacente qui lui sont reliées. On initialise chaque case de `A` avec la liste vide.
3. Créer une pile vide `p`.
4. Empiler la case de départ `(0,0)` et la marquer comme visitée : `L[0][0] = Vrai`
5. Tant que la pile n'est pas vide :
   - Dépiler la case `c`.
   - Si `c=(`i`,`j`)` a au moins une case adjacente accessible :
     - choisir aléatoirement une case suivante `s=(`k`,`l`)`
       se trouvant parmi `(`i`,`$j+1$`)`,`(`i`,`$j-1$`)`,`(`$i+1$`,`j`)`,`(`$i-1$`,`j`)`.
     - Marquer `s` comme visitée : `L[`k`][`l`]=vrai`.
     - Ajouter à la liste `A[`i`][`j`]` la case `s`.
     - Empiler `c` puis `s`.

Exercice 4.1.

1. Écrire une fonction `labyrinthe(n,m)` qui prend en entrée la largeur et la hauteur du labyrinthe et qui renvoie le tableau `A` décrit ci-dessus. Pour structurer la fonction, on écrira deux fonctions auxiliaires :
   - `cases_adjacentes(c)` qui prend en argument une case et qui renvoie la liste de ses case adjacentes accessibles.
   - `choix(V)` qui prend en argument une liste de cases adjacentes accessibles et qui renvoie aléatoirement une des ces cases.
2. Pour visualiser le labyrinthe, on utilisera la classe `image_labyrinthe` donnée en annexe. Pour générer une image ppm, il suffira d'écrire les commandes suivantes :

```
>>>A=labyrinthe(100,20)
>>>im=image_labyrinthe(A,20)
>>>im.vers_ppm("essai")
```

5 ANNEXES

```
class image_labyrinthe():
    def __init__(self,A,dim_case):
        self.dim_case =dim_case
        n,m=len(A),len(A[0])
        self.largeur = m*dim_case
        self.hauteur = n*dim_case
        self.__init_image__(self.largeur,self.hauteur)
        for i in range(n):
            for j in range(m):
                c=(i,j)
                for s in A[i][j]:
                    self.dessine_trait(c,s)

    def __init_image__(self,N,M):
        self.__tab__=[[(0,0,0) for i in range(N)] for j in range(M)]

    def dessine_trait(self,c,s):
        d=self.dim_case
        (i,j),(k,l)=c,s
        if  k == i-1 or l == j-1:
            hg =(k*d + d // 4,l*d + d // 4)
            bd =(i*d + 3*d // 4,j*d + 3*d // 4)
        if  k == i+1 or l == j+1:
            bd =(k*d + 3*d // 4,l*d + 3*d // 4)
            hg =(i*d + d // 4,j*d + d // 4)
        for x in range(hg[0],bd[0]+1):
            for y in range(hg[1],bd[1]+1):
                self.__tab__[x][y] = (255,255,255)
    def vers_ppm(self,nom):
        f=open(nom+".ppm",'w')
        N,M=self.largeur,self.hauteur
        f.write("P3\n"+str(N)+" "+str(M)+"\n"+"255\n")
        for i in range(M):
            for j in range(N):
                for k in range(3):
                    f.write(str(self.__tab__[i][j][k])+" ")
                f.write("\n")
        f.close()
```

6 Corrections

6.1 TD 2.2 : Réalisation de piles

1. Implémentation de la structure de pile :

```
def creer_pile():
    return []

def est_vide(p):
    return p == []

def empiler(p,e):
    p.append(e)

def depiler(p):
    assert not est_vide(p)
    return p.pop()

def taille(p):
    return len(p)

def sommet(p):
    assert not est_vide(p)
    return p[-1]
```

2. Pour visualiser le résultat du test, on ajoute des fonctions d'affichage :

```
def aller_a(nouvelle_url, url_courante, pile_url):
    empiler(pile_url, url_courante)
    print(nouvelle_url,pile_url)
    return nouvelle_url
def retour(pile_url):
    nouvelle_url = depiler(pile_url)
    print(nouvelle_url,pile_url)
    return nouvelle_urlRésultat du test :
```

Résultat du test :

```
page 1 ['home']
page 2 ['home', 'page 1']
page 3 ['home', 'page 1', 'page 2']
page 2 ['home', 'page 1']
page 4 ['home', 'page 1', 'page 2']
page 2 ['home', 'page 1']
page 1 ['home']
page 3 ['home', 'page 1']
page 1 ['home']
home []
```

6.2 Exercices 3

1. (a) A la fin de l'exécution du programme, la pile `p` est vide et la pile `q` contient les anciens éléments de `p` renversés.

   (b)

```
q = creer_pile()
r = creer_pile()
while not est_vide(p):
    x = depiler(p)
    empiler(q, x)
    depiler(r, x)
while not est_vide(r):
    empiler(p,depiler(r))
```

2. (a)

```
def parenthese(mot):
    p = creer_pile()
    lst_indices = []
    for j in range(len(mot)):
        if mot[j] == '(':
            empiler(p,j)
        elif mot[j] == ')':
            if est_vide(p):
                return False
            else :
                i = depiler(p)
                lst_indices.append((i,j))
    if est_vide(p):
        return lst_indices
    else :
        return False
```

   (b)

```
def parenthese(mot):
    p = creer_pile()
    for e in mot:
        if e == '(':
            empiler(p,e)
        elif e == ')':
            if est_vide(p):
                return False
            else :
                depiler(p)
    return est_vide(p)
```

   La structure de pile n'est pas nécessaire ici puisqu'on empile toujours le même élément. on peut remplacer la pile par une variable entière égale à la hauteur de la pile.

(c)

```
def parenthese_multiples(mot):
    p = pile()
    for e in mot:
        if e in ['(','{','[']:
            empiler(p,e)
        elif e == ')':
            if est_vide(p) or depiler(p) != '(' :
                return False
        elif e == '}':
            if est_vide(p) or depiler(p) != '{' :
                return False
        elif e == ']':
            if est_vide(p) or depiler(p) != '[' :
                return False
    return est_vide(p)
```

3. **Évaluation d'expressions arithmétiques**

(a)

```
def eval(exp):
    p = creer_pile()
    empiler(p,int(exp[0]))
    for e in exp[1:]:
        x = depiler(p)
        if x == '*':
            n = depiler(p)
            empiler(p,n*int(e))
        elif x == '+':
            empiler(p,'+')
            empiler(p,int(e))
        else :# x est un chiffre
            empiler(p,x)
            empiler(p,e)
    # calcul de la somme des éléments de la pile
    somme = depiler(p)
    while not est_vide(p):
        depiler(p) #on dépile '+'
        somme = somme + depiler(p)
    return somme
```

(b)

```
def verif(exp):
    chiffres = [chr(i) for i in range(ord('0'),ord('9')+1)]
    op = ['+','*']
    if len(exp) == 0:
        return False
    e = exp[0]
    if e not in chiffres :
        return False
    else :
        for x in exp[1:]:
            if x in chiffres and e not in op:
                return False
            elif x in op and e not in chiffres:
                return False
            else :
                e = x
    return x in chiffres
```

4. **Tours de Hanoï**

```
def hanoi(nb_anneaux,tour1,tour2,tour3):
    p = creer_pile()
    sol = []
    empiler(p,(nb_anneaux,tour1,tour2,tour3))
    while not est_vide(p):
        q = depiler(p)
        n, t1, t2, t3 = q
        if n == 1 :
            sol.append((t1,t3))
        else :
            empiler(p,(n-1,t2,t1,t3))
            empiler(p,(1,t1,t2,t3))
            empiler(p,(n-1,t1,t3,t2))
    return sol
```

6.3 TP 4 : Construction d'un labyrinthe parfait

1.

```
def labyrinthe(m,n):
    import random
    L=[[False] * m for i in range(n)]
    A=[[[] for j in range(m)] for i in range(n)]
    p=pile()
    def cases_adjacentes(c):
        (i,j)=c
        V=[]
        if i-1>=0 and L[i-1][j] == False : V.append((i-1,j))
        if j-1>=0 and L[i][j-1] == False : V.append((i,j-1))
        if i+1<=n-1 and L[i+1][j] == False : V.append((i+1,j))
        if j+1<=m-1 and L[i][j+1] == False : V.append((i,j+1))
        return V
    def choix(V):
        return V[random.randint(0,len(V)-1)]
    s=(0,0)
    empiler(p,s)
    L[0][0]=True
    while not est_vide(p):
        c = depiler(p)
        (i,j)=c
        V = cases_adjacentes(c)
        if V != []:
            s= choix(V)
            (k,l)=s
            L[k][l]=True
            A[i][j].append(s)
            empiler(p,c)
            p.empiler(p,s)
    return A
```