Lycée René CASSIN

NSI T*erminale*

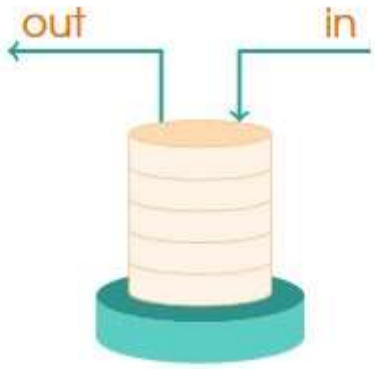
Structure de données - Les piles

Une structure de données est une organisation logique des données permettant de simplifier ou d'accélérer leur traitement.

Introduction

En informatique, une pile (en anglais stack) est une structure de données fondée sur le principe «dernier arrivé, premier sorti» (ou LIFO pour Last In, First Out), ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

Le fonctionnement est donc celui d'une pile d'assiettes: on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.



Voici quelques exemples d'usage courant d'une pile:

- Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton «Afficher la page précédente».
- L'évaluation des expressions mathématiques en notation post-fixée (ou polonaise inverse) utilise une pile.
- La fonction «Annuler la frappe» (en anglais «Undo») d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

Pour implémenter une structure de pile, on a besoin d'un nombre réduit d'opérations de bases :

- «empiler»: ajoute un élément sur la pile. Terme anglais correspondant: « Push ».
- «dépiler»: enlève un élément de la pile et le renvoie. En anglais: « Pop »
- «vide»: renvoie vrai si la pile est vide, faux sinon
- «remplissage»: renvoie le nombre d'éléments dans la pile.

La structure de pile est un concept abstrait. Comment la réaliser dans la pratique ? Voici plusieurs implémentations possibles. L'idée principale étant que les fonctions de bases pourront être utilisées indépendamment de l'implémentation choisie.

Nous utiliserons une simple liste pour représenter la pile.
Il se trouve que les méthodes append et pop sur les listes jouent déjà le rôle de push et pop sur les piles.
Voici les fonctions de base:

```
def pile():
    #retourne une liste vide return []

# vide
def vide(p):
    #renvoie True si la pile est vide et False sinon
    return p==[]

# empiler
def empiler(p,x):
    # Ajoute l'élément x à la pile p

    p.append(x)

# dépiler
def depiler(p):
    # dépile et renvoie l'élément au sommet de la pile p
    assert not vide(p), "Pile vide"
    return p.pop()
```

À faire1:

Tester les instructions suivantes :

```
p=Pile()
for i in range(5):
    p.empiler(2*i)
print(p.L)
a=p.depiler()
print(a)
print(p.L)
print(p.vide())
```

Exercice1 :

Réaliser les fonctions taille(p) et sommet(p) qui retourne respectivement la taille de la liste et le sommet de la liste (sans le supprimer)

```
def taille(p):                          |   def sommet(p):
                                        |
                                        |
                                        |
                                        |
                                        |
-                                       |
```

Implémentation - Méthode 2

Nous allons créer une classe Pile pour implémenter cette structure.

Voici la classe :

```
class Pile: ''' classe Pile création d'une instance Pile avec une
        liste
        ''' def __init__(self):
                "Initialisation d'une pile vide" self.L=[]

        def vide(self):
                "teste si la pile est vide" return self.L==[]

        def depiler(self): "dépile"
                assert not self.vide(),"Pile vide" return self.L.pop()

        def empiler(self,x):
                "empile" self.L.append(x)
```

À FAIRE 2:

Tester les instructions suivantes:

```
p=Pile()
for i in range(5):
    p.empiler(2*i)
print(p.L)
a=p.depiler()
print(a)
print(p.L)
print(p.vide())
```

EXERCICE 2 :

Réaliser les méthodes `taille(self)` et `sommet(self)` qui retournent respectivement la taille de la liste et le sommet de la liste (sans le supprimer)

```
def taille(self):                    |   def sommet(self):
                                     |
                                     |
                                     |
                                     |
                                     |
-                                    |
```

Un premier exemple - Contrôle du parenthésage d'une expression

Il s'agit d'écrire une fonction qui contrôle si une expression mathématique, donnée sous forme d'une chaîne de caractères, est bien parenthésée, c'est- à-dire s'il y a autant de parenthèses ouvrantes que de fermantes, et qu'elles sont bien placées.
Par exemple :
(..(..)..) est bien parenthésée
(...(..(..)...) ne l'est pas

L'algorithme:

On crée une pile
On parcourt l'expression de gauche à droite.
À chaque fois que l'on rencontre une parenthèse ouvrante "(" on l'empile
Si on rencontre une parenthèse fermante ") " et que la pile n'est pas vide on dépile (sinon on retourne faux)
À la fin la pile doit être vide...

À faire3:

Écrire en pseudo code l'algorithme

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Exercice3 :

En utilisant l'une des structures pile réalisées plus haut, écrire une fonction verification(expr) qui vérifie si une expression mathématique passée en paramètre est correctement parenthésée

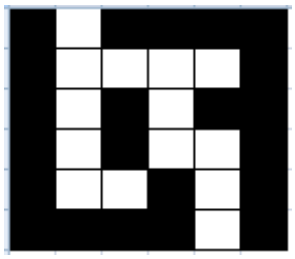
Exercice4 :

Faire en sorte que le programme tienne compte également des [

Prolongement possible : On pourrait imaginer pour un projet de développer davantage cette vérification jusqu'à proposer de montrer l'erreur...voire d'en proposer une correction...

On représentera le labyrinthe à l'aide d'un tableau (liste de listes).

Le labyrinthe (noir pour les murs...) (0 pour les murs...)



```
laby=  [[0,1,0,0,0,0],
        [0,1,1,1,1,0],
        [0,1,0,1,0,0],
        [0,1,0,1,1,0],
        [0,1,1,0,1,0],
        [0,0,0,0,1,0]]
```

- On repère une case par ses coordonnées (*i,j*), on y accède dans le tableau par laby[i][j]
- L'entrée se fait par la case (0,1) et la sortie par la case (5,4)

? Question1:

Compléter le code pour obtenir le nombre de lignes et le nombre de colonnes de ce tableau :

```
lignes=len(          )
colonnes=len(          )
```

Nous aurons besoin de créer une copie de ce tableau pour pouvoir travailler dessus sans altérer le tableau laby. Nous utiliserons une bibliothèque pour cela.

À faire4:

Tester ce code

```
from copy import deepcopy

laby=[[0,1,0,0,0,0],
      [0,1,1,1,1,0],
      [0,1,0,1,0,0],
      [0,1,0,1,1,0],
      [0,1,1,0,1,0],
      [0,0,0,0,1,0]]

T=deepcopy(laby)
T[3][2]='hello'

for ligne in laby:
    print(ligne)
print("----------------")
for ligne in T:
    print(ligne)
```

Objectif : L'objectif est d'écrire un programme qui détermine s'il existe un chemin de l'entrée vers la sortie en se déplaçant vers le haut, le bas, la gauche ou la droite (mais pas en diagonale).
Étape 1:

À faire5:

Voici une fonction qui prend en paramètre un tableau T et un tuple v

```
def voisins(T,v): V=[]
    i,j=v[0],v[1] for a in
    (-1,1):
        if 0<=i+a<lignes:
            if T[i+a][j]==1:
                V.append((i+a,j)) if
        0<=j+a<colonnes:
            if T[i][j+a]==1:
                V.append((i,j+a))
    return V
```

Question2:

Que retourne cette fonction?

.....
.....
.....
.....
.....
.....

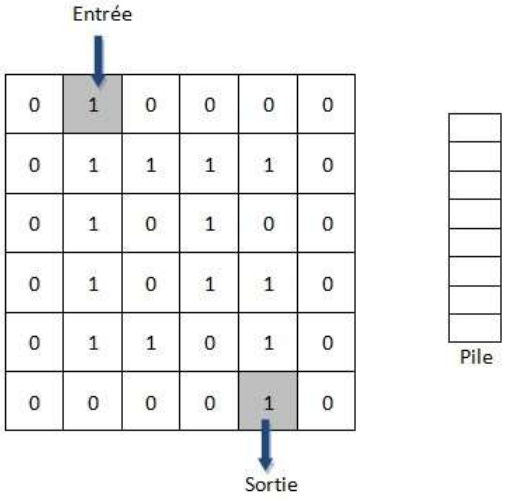
Question3:

Expliquer l'affichage provoqué par cette instruction: print(voisins(laby,(0,1))

.....
.....
.....
.....
.....
.....

Spécifier la fonction

Étape 2: parcours du labyrinthe

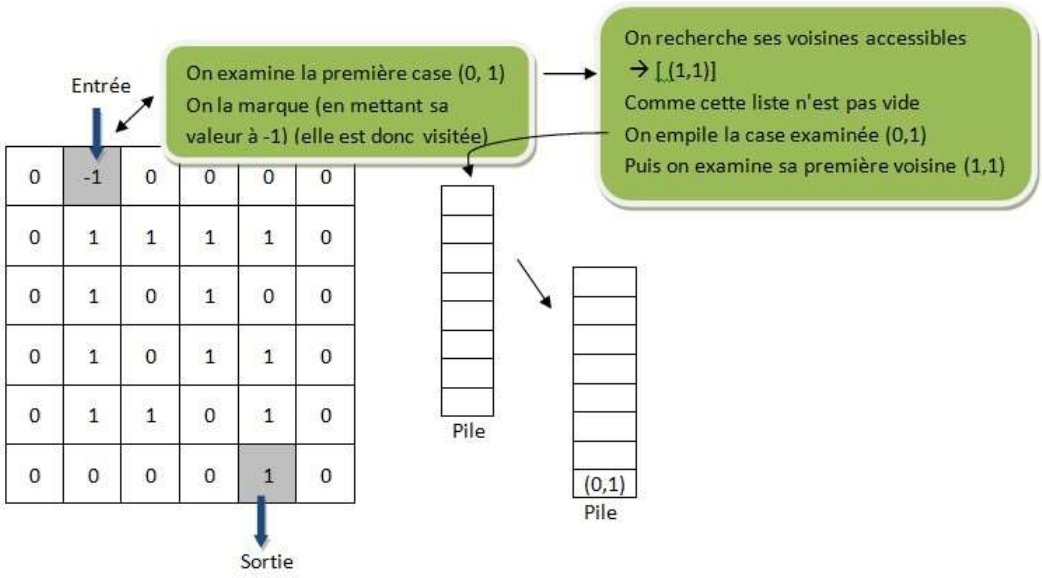


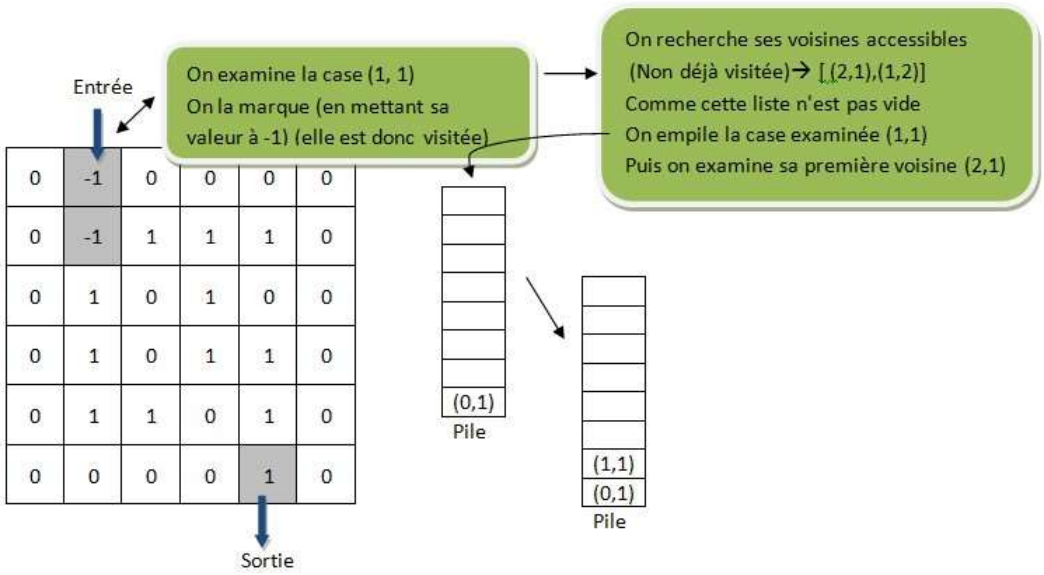
L'idée est de parcourir le labyrinthe depuis l'entrée, en utilisant une pile pour stocker le chemin, pour pouvoir dépiler lorsque le chemin n'aboutit pas et redémarrer sur une autre voie..

Un schéma sera sans doute plus efficace qu'un long discours...

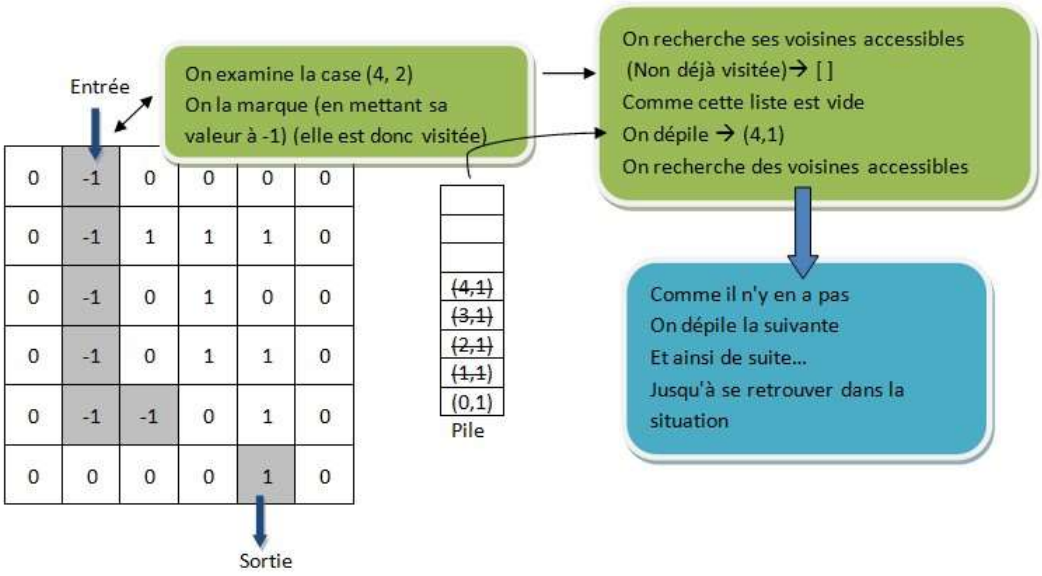
On dispose d'une copie de notre tableau et d'une pile vide.

Le processus...

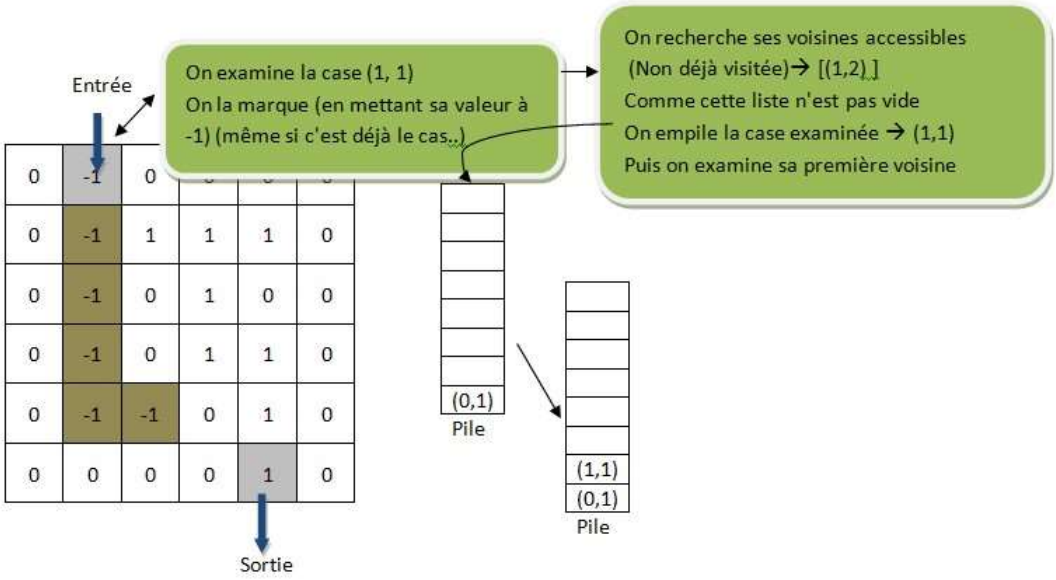




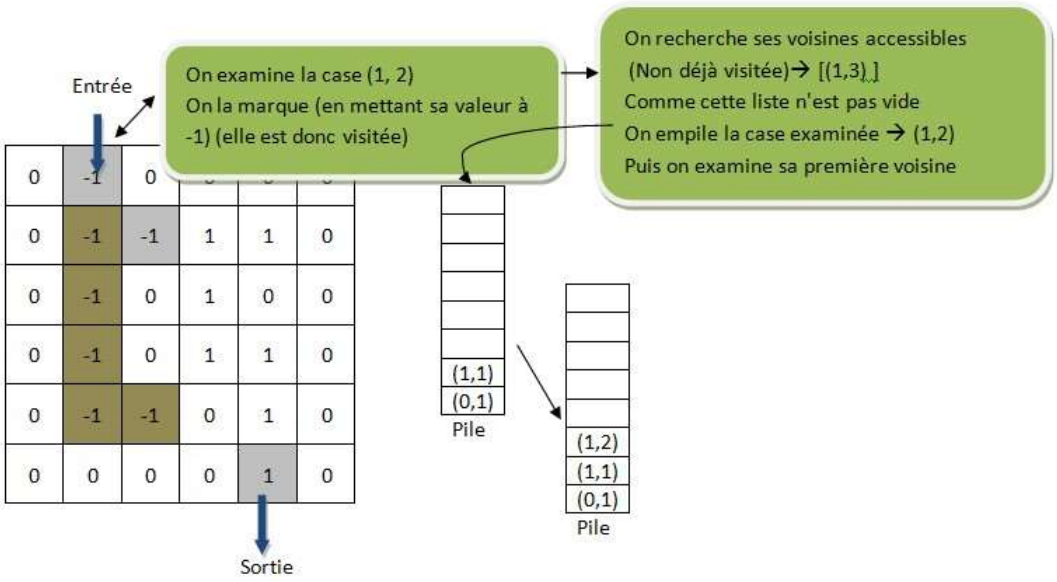
Et ainsi de suite jusqu'à ce que l'on tombe sur une impasse…



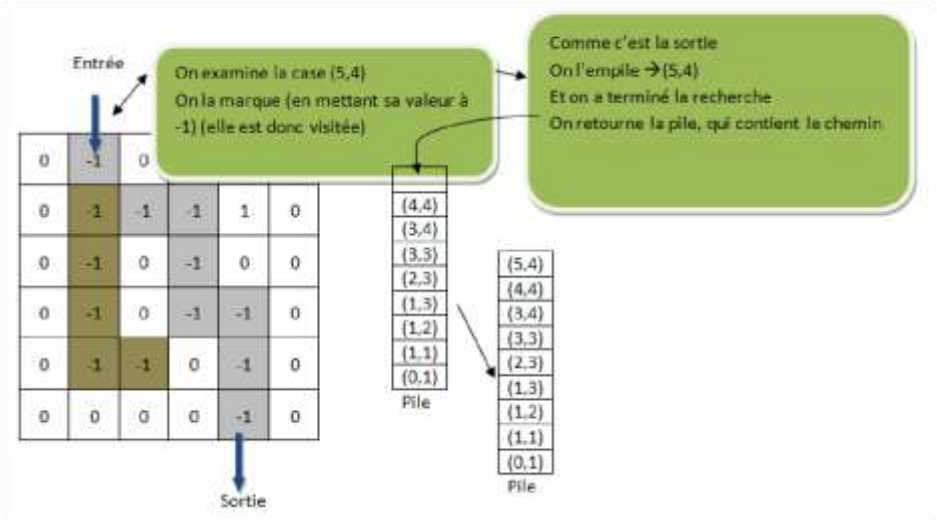
On examine la dernière case dépilée...



On poursuit l'exploration...



Et ainsi de suite jusqu'à la sortie...



L'algorithme :

À FAIRE 6:

Compléter l'algorithme :

```
fonction parcours( laby , entree, sortie)
    T <-- une copie de ........
    p <-- Pile()
    v <-- entree
    on met à .... la valeur de la case.... dans ...
    recherche <--True
    tant que recherche est vrai
        vois <-- voisins (...,...)
        si la liste ... est vide
            si la pile vide --> renvoyer False
            sinon v <--.......................
        sinon
            on ........ p  avec ....
            v <-- le 1er ...................
            on met à .... la valeur de la case.... dans ...
            si v = ......
                On ........ p  avec......
                recherche <--.......
    return ....
```

QUESTION 4:

Si au cours de l'exécution la pile se trouve vide, que cela signifie-t-il?

.....

.....

.....

Réaliser le programme (mettre en évidence le chemin) et tester le avec d'autres labyrinthes

Prolongement possible : Pour un projet on pourrait imaginer un programme qui génère un labyrinthe et qui le résout...