Piles

Une nouvelle structure de données informatiques

20 septembre 2017

Table des matières

1 Introduction

1.1 Structure simple : le tableau

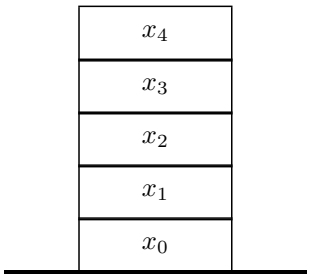
Les tableaux existent dans tous les langages de programmation. C'est la manière la plus simple de structurer des données. Principales caractéristiques :

- la taille du tableau est fixe ;
- les éléments du tableau ont tous le même type (qui peut être tableau) ;
- on peut accéder directement à chaque élément du tableau, à condition de connaître sa place : $T[i]$ retourne le $i^{ème}$ élément du tableau T .

Avantages :
- maîtrise de la place occupée en mémoire ;
- possibilité de faire du calcul matriciel.

1.2 Structure de pile

La structure de **pile** (*stack*) est différente, elle correspond à des données empilées. Le pointeur de la machine (tête de lecture/écriture) est au sommet de la pile.



C'est une structure *LIFO : Last In First Out*.

Principales caractéristiques des piles :
- la hauteur de la pile n'est pas bornée, on peut empiler autant de données qu'on veut ;
- les éléments d'une pile ne sont pas forcément de même type ;
- seul la donnée au sommet de la pile est accessible, pour la lecture ou la suppression ;
- pour accéder aux autres données, il faut d'abord supprimer une par une toutes les données qui sont au-dessus (**dépilage**).

Avantages des piles :
- structure facile à gérer pour la machine, tant que la mémoire n'est pas saturée ;
- les fonctionnalités élémentaires sont instantanées ;
- elles permettent la récursivité.

Inconvénients :
- structure très pauvre ;
- toute manipulation élaborée nécessitera énormément d'empilages et de dépilages.

2 Fonctionnalités des piles

2.1 Définitions

- Une **fonction pure** s'applique à 0, 1 ou plusieurs **arguments**, *ne les modifie pas*, et *retourne un résultat.*
- Une **procédure pure** s'applique ou non à des arguments, *les modifie*, et *ne retourne aucun résultat.*
- Une **fonction-procédure** (ou fonction impure) modifie les données et retourne un résultat.

Python ne fait pas la différence entre fonctions et procédures : elles se déclarent de la même manière. Mais vous devrez comprendre la différence.

2.2 Fonctions et procédures disponibles

Fonctions pures :
- taille : retourne un entier ;
- vacuité : retourne un booléen ;
- sommet : retourne la valeur de la donnée en haut de la pile, sans l'enlever.
- créer : retourne une pile vide (pas d'argument).

Procédures pures :
- empiler un élément sur une pile existante (2 arguments : pile, élément) ;
- dépiler une pile non vide (un argument : pile).

Fonction-procédure :
- dépiler-lire : retire le sommet de la pile et donne sa valeur.

2.3 Réalisation en Python

En Python, les piles n'existent pas, mais sont simulables avec le type *list* qui est en fait un hybride de pile et de tableau.

```
def taille(P):
    return len(P)

def sommet(P):
    assert len(P)!=0
    return P[-1]

def vacuite(P):
    return len(P)==0

def creer():
    return []
```

```
def empiler(P,x):
    P.append(x)

def depiler(P):
    assert len(P)!=0
    P.pop()


def depilerlire(P):
    assert len(P)!=0
    return P.pop()
```

3 Application 1 : analyse du parenthésage d'un texte

3.1 Le problème

On dispose d'un texte, c'est-à-dire d'une chaine de caractères, comportant entre autres les délimiteurs (,) , [,] , < , > etc. On voudrait savoir si ce texte est correctement parenthésé, c'est-à-dire si chaque délimiteur entrant est suivi du sortant correspondant.

Exemples :
- Ceci est (sans doute (et même certainement)) une phrase correctement [parenthésée] .
- Ceci n'est pas (c'est [facile à voir)] une phrase correctement parenthésée.
- simplify(seq(int(sin(a[i]*x)/(x**(2*i+1)+1,x=0..Pi),i=1..10)) l'est-elle ?

3.2 Une solution

Algorithme d'analyse du parenthésage :

— Il s'agit de définir une fonction qui prend en argument un texte, et retourne un booléen : `True` s'il est correctement parenthésé, `False` sinon.
— On crée une pile pour les délimiteurs.
— On parcourt le texte en testant chaque caractère :
  — si c'est un délimiteur entrant, on l'empile ;
  — si c'est un délimiteur sortant, on compare son type à celui du sommet de la pile ; si c'est le même on dépile, sinon on arrête tout car le parenthésage est incorrect ;
  — sinon on ne fait rien.
— Dans le cas où on a pu terminer le texte : le parenthésage est correct ssi la pile est vide.

3.3 Implémentation en Python

```
def parentheses(texte):
    t=str(texte) # conversion en chaine de caractères
    P=creer()
    for c in t:
        if c=='(' or c=='[' or c=='{': empiler(P,c)
        elif c==')':
            if sommet(P)=='(': depiler(P)
            else: return False
        elif c==']':
            if sommet(P)=='[': depiler(P)
            else: return False
        elif c=='}':
            if sommet(P)=='{': depiler(P)
            else: return False
    return vacuite(P) # vrai ssi la pile est vide.
```

4 Application 2 : évaluateur de notation polonaise inversée

4.1 Le problème

L'écriture classique d'une expression algébrique nécessite l'utilisation de parenthèses pour regrouper certaines opérations. La notation polonaise inversée présente l'avantage de rendre le parenthésage inutile. Le principe est que les opérateurs (+, -, *, /) sont placés *après* leurs opérandes (nombres).

Exemples

— 2+3 s'écrit 2 3 + ;
— 2+3*4 s'écrit 2 3 4 * + ;
— (2+3)*4 s'écrit 2 3 + 4 * ;
— (2+3)*(4+5) s'écrit 2 3 + 4 5 + * .

Il s'agit d'écrire un évaluateur de telles expressions.

4.2 Une solution

Algorithme d'évaluation :

- Il s'agit de définir une fonction qui prend en argument une liste contenant des nombres et des opérateurs *transcrits en caractères*, et retourne un nombre, ou `False`.
- On crée une pile qui contiendra les nombres.
- On parcourt la liste en testant chaque élément :
  - si c'est un nombre, on l'empile ;
  - si c'est un opérateur, on dépile et lit 2 fois, on effectue l'opération, et on empile le résultat ;
  - sinon on sort car l'expression est incorrecte.
- Dans le cas où on a pu terminer la liste : il ne devrait rester qu'un nombre dans la pile, qui est le résultat ; sinon l'expression est incorrecte.

4.3 Implémentation en Python

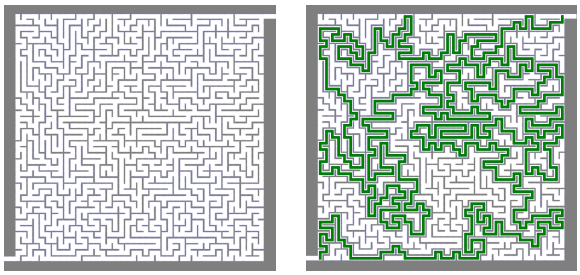
```
def NPI(L):
    P=creer()
    for c in L:
        if type(c)==int or type(c)==float: empiler(P,c)
        elif c in {'+','-','*','/'} and len(P)>=2:
            b=depilerlire(P)
            a=depilerlire(P)
            if c=='+': empiler(P,a+b)
            elif c=='-': empiler(P,a-b)
            elif c=='*': empiler(P,a*b)
            else: empiler(P,a/b)
        else: return False
    if len(P)==1: return depilerlire(P)
    else: return False
```

5 Application 3 : création d'un labyrinthe parfait

5.1 Le problème

Il s'agit de construire un labyrinthe carré, de taille à choisir. Ce labyrinthe devra être **parfait** : pour tout couple de points distincts du carré, il devra exister exactement un chemin qui les joint. Une fois le labyrinthe construit, on doit pouvoir demander la solution.

Exemple



5.2 Une solution

— Le labyrinthe est une tableau carré de taille n, qui sera l'argument de la fonction de construction, et rempli de booléens : `True` si la case a déjà été visitée, `False` sinon.
— Le cheminement dans ce tableau sera une pile, car le principe est qu'on avance (=empilement) quand on le peut, et recule (=dépilement) quand on arrive dans un cul de sac.
— Quand on peut avancer, on choisit de manière équiprobable parmi les cases adjacentes pas encore visitées ; on dessine alors le segment parcouru.
— La pile est vide quand il n'y a plus aucune case disponible, ce qui signifie qu'on les a toutes visitées. Le labyrinthe est alors terminé.

5.3 Implémentation, en Python

À vous de jouer !