Cours d'algorithmique et programmation 2

Anne Parrain

UFR des Sciences
Licence Sciences et Technologie
mentions Mathématiques et Informatique
Semestre 2

Section 1

Les piles

La structure de données Pile (1)

Comment fonctionne la touche *Undo* ou *Annuler la frappe* d'un éditeur ou d'un traitement de texte ?

- Les actions effectuées sont mémorisées en les empilant l'une derrière l'autre,
- au fur et à mesure qu'on annule, on revient en arrière en dépilant les actions effectuées.

La structure nécessaire est très courante en algorithmique. Elle reprend l'idée de la pile d'assiettes

- on les range l'une au-dessus de l'autre après la vaisselle,
- on les enlève dans l'ordre inverse au moment de dresser la table.

Cette structure s'appelle également une **LIFO** : *Last In First Out*.

La structure de données Pile (2)

De quoi a-t-on besoin ?

- d'une structure qui mémorise une collection de données
- d'une structure qui mémorise l'ordre d'ajout des données
- d'une structure qui garantisse l'enlèvement des éléments dans l'ordre inverse de l'ajout

En fait, on veut une sorte de liste dont l'usage serait restreint.

La structure de données Pile (3)

Une structure de données se caractérise par les opérations élémentaires qu'on peut faire sur cette structure.

Dans le cas d'un pile, il s'agit :

- d'un constructeur, qui retourne une pile vide
- d'une opération `empiler` ou `push` qui met un élément au sommet d'une pile
- d'une opération `dépiler` ou `pop` qui enlève l'élément au sommet de la pile et le retourne
- d'une opération `est_vide` ou `empty` qui indique si une pile est vide ou non
- d'une opération `sommet` ou `peek` qui retourne l'élément au sommet de la pile

Remarque : Une pile peut manipuler n'importe quel type d'éléments.

Implémentation d'une classe `Pile`

```
1  class Pile :
2    '''Classe qui modelise la structure de donnees Pile.'''
3    def __init__(self):
4      '''Pile -> Pile
5      construit une pile vide.'''
6    def empiler(self,elt):
7      '''(modif) Pile, Objet -> None
8      empile elt au sommet de la pile.'''
9    def depiler(self):
10     '''(modif) Pile -> Objet
11     enleve l'element au sommet de la pile
12     et le retourne.'''
13   def est_vide(self):
14     '''Pile -> boolean
15     teste si la pile est vide.'''
16   def sommet(self):
17     '''Pile -> Objet
18     retourne l'element au sommet de la pile
19     (sans l'enlever).'''
```

À vous !

Complétez l'implémentation de la classe `Pile`.

- Quel(s) attribut(s) nécessaire(s) ?
- Comment les utiliser ?

Implémentation d'une classe `Pile` (2)

```
1  class Pile :
2      '''Definition d'une pile -
3      structure de donnees LIFO (Last In First Out)
4      '''
5
6      def __init__(self) :
7          '''Pile -> Pile
8          construit une pile vide
9          '''
10         self.__les_elts = list()
11
12
13     def empiler(self,elt) :
14         '''(modif) Pile, Objet -> Rien
15         ajoute un element au sommet de la pile.
16         '''
17         self.__les_elts.append(elt)
```

Implémentation d'une classe `Pile` (3)

```
1     def est_vide(self) :
2         '''Pile -> Boolean
3         teste si la pile est vide. '''
4         return len(self.__les_elts) == 0
5
6     def sommet(self) :
7         '''Pile -> Objet
8         retourne l'element au sommet de la pile.
9         La pile ne doit pas etre vide.
10        '''
11        assert not self.est_vide()
12        return self.__les_elts[-1]
```

`assert` : si la condition n'est pas vérifiée, le programme se termine en erreur !

Implémentation d'une classe `Pile` (4)

```
1     def depiler(self) :
2         '''(modif) Pile -> Objet
3         enleve l'element au sommet de la pile.
4         La pile ne doit pas etre vide.
5         '''
6         assert not self.est_vide()
7         elt = self.__les_elts[-1]
8         del(self.__les_elts[-1])
9         return elt
```

Section 2

Utiliser une pile

Une calculatrice postfixée

La notation postfixée pour les expressions arithmétiques (appelée aussi notation polonaise inverse) est une notation qui place les opérandes d'une expression avant l'opérateur qui les concerne.

```
(3 2 +) est equivalent a (3 + 2)
((12 6 7 8 *) (6 4 -) +) est equivalent a
                          ((12 * 6 * 7 * 8) + (6 - 4))
```

On veut écrire une calculette en python qui évalue de telles expressions :

```
moi@mamachine$ python3 calculette.py
Votre expression postfixee ? (. pour arreter)
((3 12 5 +)(12 6 -)*)
Le resultat est  120
Votre expression postfixee ? (. pour arreter)
((5 10 -) (18 3 /) 251 +)
Le resultat est  252
Votre expression postfixee ? (. pour arreter) .
```

Les parenthèses ne sont pas obligatoires mais aident à la lisibilité.

Évaluer une expression postfixée

On ne considère que le cas d'expressions bien formées, avec uniquement des entiers positifs.

Idée :

- on parcourt l'expression
- on empile les opérandes (les nombres) et les (
- quand on rencontre un opérateur (+, -, *, /) :
  - on dépile les opérandes jusqu'à rencontrer une parenthèse ouvrante
  - en faisant l'opération souhaitée
  - on dépile la (
  - on empile le résultat

Lorsqu'on aura fini de lire l'expression postfixée, le résultat sera au sommet de la pile.

Algorithme pour l'évaluation d'une expression postfixée

```
Entree : expr, une expression postfixee de type str
Sortie : le resultat de l'evaluation

nb <- 0
nb_lu <- Faux
pile est une Pile
pour chaque caractere car de expr faire\
  si car est une ( alors
    pile.empiler('(')
  sinon si car est un + alors
    addition(pile)
  sinon si car est un - alors
    soustraction(pile)
  sinon si car est un * alors
    multiplication(pile)
  sinon si car est un / alors
    division(pile)
  ...
```

Algorithme pour l'évaluation d'une expression postfixée

```
  ...
  sinon si car est un chiffre alors
    nb <- nb*10 + int(car)
    nb_lu <- Vrai
  sinon si car est un separateur et nb_lu alors
    pile.empiler(nb)
    nb <- 0
    nb_lu <- Faux
  # on ne fait rien pour la ')'
fin pour
res <- pile.depiler()
retourner res
```

Algorithme pour l'addition

```
addition(pile) :
    res <- pile.depiler()
    tant que pile.sommet() n'est pas '(' faire
      res <- res + pile.depiler()
    fin tant que
    pile.depiler() # pour la (
    pile.empiler(res)
```

Algorithme pour la soustraction

```
soustraction(pile) :
  nb2 <- pile.depiler()
  nb1 <- pile.depiler()
  pile.depiler() # pour la (
  pile.empiler(nb1-nb2)
```

À vous !

- Spécifiez puis écrivez les fonctions `est_un_chiffre(car)` et `est_un_separateur(car)`
- Spécifiez les différentes fonctions spécifiques pour une opération.
- Spécifiez puis écrivez la fonction principale.
- Écrivez les fonctions spécifiques pour une opération.