Alocação sequencial - Pilhas

Pilhas

A estrutura de dados Pilha é bastante intuitiva. A analogia é uma pilha de pratos. Se quisermos usar uma pilha de pratos com a máxima segurança, devemos inserir um novo prato “no topo” da pilha e retirar um novo prato “do topo” da pilha.

Por isso dizemos que uma pilha é caracterizada pelas seguintes operações:
O último a entrar é o primeiro a sair ou
O primeiro a entrar é o último a sair

Os nomes usados em inglês são:
`LIFO – Last In First Out` ou
`FILO – First In Last out`

Existem outros exemplos em programas que utilizam uma pilha. Editores de texto, planilhas, browsers, etc. tem sempre um ícone que restaura o texto antes da última modificação. As modificações são armazenadas numa pilha para garantir a sequência de retorno ou avanço. Nesses casos, a pilha é usada para manter a sequência de alterações.

Considere o exemplo abaixo que mostra a evolução de uma pilha de letras.

- Uma letra significa empilhe a letra;
- Um ponto significa desempilhe uma letra;

Operação	Retirado	Pilha
E		E
X		EX
E		EXE
.	E	EX
.	X	E
M		EM
P		EMP
.	P	EM
.	M	E
.	E	
L		L
O		LO
.	O	L
D		LD
E		LDE
P		LDEP
I		LDEPI
.	I	LDEP
.	P	LDE
.	E	LD
.	D	L

.	L	
.		Erro – pilha vazia
L		L
H		LH
A		LHA

Operações sobre uma pilha P

Sobre uma pilha podemos então usar várias operações. As mais comuns são:

Empilhar elemento x – push(P, x) – do verbo push down - empurrar para baixo
Desempilhar elemento – pop(P) – do verbo pop up – surgir, aparecer

Outras operações auxiliares:

Iniciar uma pilha – init(P)
Consultar o valor do elemento do topo – top(P)
Retornar True se a pilha está vazia – is_empty(P)
Retornar o tamanho atual da pilha – len(P)

Em especial as operações pop(P) e top(P) devem avisar se a pilha está vazia.

Implementação de pilha como uma lista

Uma pilha pode ser facilmente implementada usando uma lista em Python. O topo da pilha é o último elemento e a base da pilha o primeiro. Os métodos append() e pop() podem ser usados para empilhar e desempilhar elementos. A operação len(P) é a própria função do Python.

```
# Inicia uma pilha – vazia
def init(P):
    P = []

# Empilha novo elemento
def push(P, x):
    P.append(x)

# Retorna o elemento do topo da pilha mas não desempilha
def top(P):
    if is_empty(P): return None
    return P[-1]

# Remove elemento do topo da pilha
def pop(P):
    if is_empty(P): return None
    return P.pop()

# retorna True se a pilha está vazia
def is_empty(P):
    return len(P) == 0
```

Note que podemos empilhar qualquer tipo de objeto (int, bool, string, tupla, lista, set, etc.). Inclusive misturando tipos na própria pilha. Com as funções acima não estamos restritos a pilhas de um tipo de elementos. Considere o programa abaixo:

```
# Empilha e imprime o estado atual da pilha
def Empilha(p, z):
    push(p, z)
    print(p)

# Desempilha e imprime o estado atual da pilha
def Desempilha(p):
    x = pop(p)
    if x is None: print("Vazia")
    else: print(p)

# inicia a pilha
mp = []
init(mp)

# operações
Empilha(mp, 1)
Empilha(mp, "tipo de elemento")
Empilha(mp, (5, 4, 3))
Empilha(mp, True)
print(top(mp))
Desempilha(mp)
Desempilha(mp)
Desempilha(mp)
Desempilha(mp)
Desempilha(mp)
print(top(mp))
```

Será impresso:

```
[1]
[1, 'tipo de elemento']
[1, 'tipo de elemento', (5, 4, 3)]
[1, 'tipo de elemento', (5, 4, 3), True]
True
[1, 'tipo de elemento', (5, 4, 3)]
[1, 'tipo de elemento']
[]
Vazia
None
```

Limitando o tamanho - outra forma de implementar uma pilha por uma lista

Definir a lista com um número máximo de elementos e limitar o tamanho da pilha é mais eficiente do que inserir cada elemento com append(). Podemos redefinir as funções acima dessa maneira. Um

parâmetro a mais (MaxTamPilha) nas funções init() e push(). A push() deve devolver False se a pilha já estiver cheia.

P5.1) Reescrever as funções acima com essa nova definição.

Para verificar se os parêntesis dentro de uma expressão estão balanceados (cada abre tem o seu fecha correspondente), podemos usar uma pilha. A cada abre empilhamos e a cada fecha desempilhamos. Nem precisa usar uma pilha. Basta um contador mais elaborado.

P5.2) Dada uma expressão, verificar se os parêntesis estão balanceados.

P5.3) Supondo agora que a expressão possa conter também `[]` e `{}` além de `()`. Exemplo:
`{a * [b + c * (d – e) * {a – b}]}`

Implementação de pilha como um tipo de dado abstrato (ADT)

A pilha pode ser implementada como um tipo abstrato de dados.
O caso de pilha vazia é certamente um caso de erro, pois há forma de verificar essa condição antes de usar as funções pop() e top(). Por isso, optamos por indicar uma exceção quando isso ocorre. A função __len__ assim definida para usar o mesmo nome para um novo objeto.

```
class PilhaLista:

    '''Pilha como uma lista.'''

    # Construtor da classe PilhaLista
    def __init__(self):
        self._pilha = [] # lista que conterá a pilha

    # retorna o tamanho da pilha
    def __len__ (self):
        return len(self._pilha)

    # retorna True se pilha vazia
    def is_empty(self):
        return len(self._pilha) == 0

    # empilha novo elemento e
    def push(self, e):
        self._pilha.append(e)

    # retorna o elemento do topo da pilha sem retirá-lo
    # exceção se pilha vazia
    def top(self):
        if self.is_empty( ):
            raise Empty("Pilha vazia")
        return self._pilha[-1]

    # desempilha elemento
```

```
    # excessão se pilha vazia
    def pop(self):
        if self.is_empty( ):
            raise Empty("Pilha vazia")
        return self._pilha.pop( )

# testes
P = PilhaLista()
P.push(1)
P.push("tipo de elemento")
P.push((5, 4, 3))
P.push(True)
print("tamanho da pilha = ", len(P))
print("elemento do topo da pilha = ", P.top())
print(P.pop())
print(P.pop())
print(P.pop())
print(P.pop())
# os comandos abaixo geram uma excessão
x = P.pop()
x = P.top()
```

Será impresso:

```
tamanho da pilha =  4
elemento do topo da pilha =  True
True
(5, 4, 3)
tipo de elemento
1
Traceback (most recent call last):
  File "D:\Marcilio\Python - fontes - Marcilio\ClassePilhaLista.py",
line 52, in <module>
    x = P.pop()
  File "D:\Marcilio\Python - fontes - Marcilio\ClassePilhaLista.py",
line 36, in pop
    raise Empty("Pilha vazia")
Empty: Pilha vazia
```

P5.4) Escreva também dentro do ADT PilhaLista, a função __str__(self), para permitir a impressão de uma pilha no formato vertical. Por exemplo, no exemplo acima, o comando print(P) deveria imprimir:

```
3 - True
2 - (5, 4, 3)
1 - tipo de elemento
0 - 1
```

Aplicações

A estrutura de pilha é uma estrutura fundamental em muitas aplicações da computação. Citamos alguns:

a) Na execução de um programa, quando se entra num novo bloco, podem existir novas variáveis. Estas novas variáveis são alocadas numa pilha de variáveis, pois podem existir variáveis com os mesmos nomes. A busca sobre qual variável está referenciada é feita do topo para a base da pilha.

```
def f1():
    a = 1                       # pilha = a, z, a
    b = 1.0                     # pilha = a, z, a, b
    def f2():
        a = 2                   # pilha = a, z, a, b, a
        b = 2.0                 # pilha = a, z, a, b, a, b
        c = 2.0                 # pilha = a, z, a, b, a, b, c
        print(a, b, c, z)
    f2()
    print(a, b, z)              # pilha = a, z, a, b

a = 0                           # pilha = a
z = -1                          # pilha = a, z
f1()
print(a, z)
```

b) O próprio algoritmo de análise sintática (aquele algoritmo que verifica se a sintaxe do programa está correta) usa uma pilha sintática, para guardar o contexto do programa sendo analisado (para entender melhor isso seria necessário conhecer melhor esses algoritmos, o que não é objeto deste curso).

c) Suponha que durante a execução de um programa, uma função f, chama uma função g, que chama uma função h. Obviamente, depois da execução de h, deve-se voltar para g e depois de g, voltar para f. Os endereços de retorno após a chamada de uma função são colocados numa pilha de execução, para que se volte para o lugar certo.

d) Um caso especial de chamada de funções é quando usamos funções recursivas. Como a função chamada é sempre a mesma, o controle dos endereços de retorno e dos parâmetros de cada chamada, é feito pela pilha de execução. Os parâmetros de cada chamada são também empilhados.

Notação pós-fixa para expressões

A notação pós-fixa para expressões, também chamada de notação polonesa, pois foi proposta pelo matemático polonês Jan Łukasiewicz (1878 - 1956) e usada pela primeira vez em computação pelo

cientista de computação Charles Hamblin em 1957, apresenta uma série de facilidades em relação a notação tradicional. Nela os operadores aparecem após os operandos e não entre os operandos.

Exemplos:

	Notação usual (in-fixa)	Notação polonesa (pós-fixa)
1	`a+b`	`ab+`
2	`a+b+c`	`ab+c+ ou abc++`
3	`a+b*c`	`abc*+`
4	`(a+b)*c`	`ab+c*`
5	`c*(a+b)`	`cab+*`
6	`b**2-4*a*c`	`b2^4a*c*- ou b2^4ac**-`
7	`(-b+√(b**2-4*a*c))/(2*a)`	`b_b2^4ac**-√+2a*/`

Algumas observações sobre a notação pós-fixa:

a) Essa notação elimina a necessidade de parêntesis.

b) Pode existir mais de uma solução (exemplos 2 e 6) devido a mesma prioridade dos operandos envolvidos. Nos 2 exemplos acima, é melhor considerar a primeira solução, pois a convenção usual é que quando temos operadores de mesma prioridade (no exemplo 2 **++** e no exemplo 6 `**`) as operações são feitas da esquerda para a direita.

c) Os operandos aparecem na mesma ordem em que se encontram na expressão original.

d) Operadores diferentes que usam o mesmo símbolo podem causar confusão na notação pós-fixa. é o caso dos operadores unários – e + que podem ser interpretados como a subtração e adição normais. É também o caso da exponenciação ** que pode ser interpretada como multiplicações sucessivas. Não é o caso da √ (raiz quadrada) que possui símbolo próprio. Assim é conveniente, ao traduzir para a notação pós-fixa, usar outro símbolo para distinguir os operadores. Por isso, nos exemplos acima, o – unário foi traduzido por _ (underline) e a exponenciação ** traduzida por ^.

e) Para se calcular o valor da expressão em notação pós-fixa, varre-se a expressão da esquerda para a direita simplesmente. Não é necessário verificar qual a operação que se faz primeiro devido a sua prioridade ou a existência de parêntesis.

Uma decorrência dessa última observação, é que se houver uma maneira fácil de traduzir uma expressão para a sua correspondente forma pós-fixa, o cálculo da mesma será feito de forma direta, simplesmente percorrendo a expressão da esquerda para a direita e fazendo as operações à medida que aparecem. É o que veremos a seguir.

Algoritmo para transformar uma expressão para a notação pós-fixa

Varrendo a expressão normal da esquerda para a direita, o algoritmo deve levar em conta a prioridade dos operadores antes de colocá-los na expressão pós-fixa. Assim, antes de colocar um operador na expressão pós-fixa, é necessário saber se o próximo operador é menos prioritário que ele.

Isso sugere usar uma pilha para os operadores. Quando chega um operador mais prioritário que o do topo da pilha, empilha-se este também. Mas se for menos prioritário, o do topo tem que ir para a expressão pós-fixa e dar lugar para este que acabou de chegar.

Os parêntesis são um caso especial. Quando aparece um fecha, deve-se colocar na pós-fixa todos os operadores até o abre correspondente.

```
while (expressão não chegou ao fim):
    p = próximo elemento da expressão (operador ou operando)
    if (p é operando): coloque na pós-fixa
    if (p é operador):
      tire da pilha e coloque na pós-fixa todos os operadores
      com prioridade maior ou igual a p, na mesma ordem de
      retirada da pilha
      empilhe p;
    if (p é abre parêntesis): empilha p;
    if (p é fecha parêntesis):
      desempilhe os operadores até o primeiro abre e coloque
      na pós-fixa na mesma ordem de retirada da pilha;

desempilhe todos os operadores que ainda estão na pilha e coloque na
pós-fixa na mesma ordem de retirada da pilha
```

Os operadores devem então estar organizados por sua prioridade.
A função abaixo define a prioridade dos operadores binários +, -, * , /, e ^ .
Para facilitar o algoritmo damos também prioridade ao abre, fecha e a operando, embora tenham tratamento especial no algoritmo.

```
def prioridade(x):
    if x == '+': return 1
    elif x == '-': return 1
    elif x == '*': return 2
    elif x == '/': return 2
    elif x == '^': return 3
    elif x == '(': return 4 # caso particular
    elif x == ')': return 5 # caso particular
    else: return 0          # não é operador
```

Comportamento da pilha em alguns exemplos

Abaixo, exemplos do comportamento da pilha para algumas expressões:

a+b

	Pilha	Pós-fixa
a		a
+	+	a
b	+	ab
		ab+

a+b+c

	Pilha	Pós-fixa
a		a
+	+	a
b	+	ab
+	+	ab+
c	+	ab+c
		ab+c+

a*b+c

	Pilha	Pós-fixa
a		a
*	*	a
b	*	ab
+	+	ab*
c	+	ab*c
		ab*c+

a+b*c

	Pilha	Pós-fixa
a		a
+	+	a
b	+	ab
*	+*	ab
c	+*	abc
		abc*+

a*(b+c)

	Pilha	Pós-fixa
a		a
*	*	a
(	*(	a
b	*(	ab
+	*(+	ab
c	*(+	abc
)	*	abc+
		abc+*

(a+b)*c

	Pilha	Pós-fixa
(	(	
a		a
+	(+	a
b	(+	ab
)		ab+
*	*	ab+
c	*	ab+c
		ab+c*

a*(b+(c*(d+e)))

	Pilha	Pós-fixa
a		a
*	*	a
(	*(	a
b	*(	ab
+	*(+	ab
(	*(+(	ab
c	*(+(	abc
*	*(+(*	abc
(	*(+(*(	abc
d	*(+(*(	abcd
+	*(+(*(+	abcd
e	*(+(*(+	abcde
)	*(+(*	abcde+
)	*(+	abcde+*
)	*	abcde+*+
		abcde+*+*

a/(b*c)*d

	Pilha	Pós-fixa
a		a
/	/	a
(	/(	a
b	/(	ab
*	/(*	ab
c	/(*	abc
)	/	abc*
*	*	abc*/
d	*	abc*/d
		abc*/d*

A prioridade dos operadores em Python

Nos exemplos acima, usamos os operadores mais usuais. Porém, todos os operadores em Python tem uma prioridade associada. Os operadores unários (+ e −), os operadores lógicos (and, or e not), etc. seguem a mesma regra com a sua respectiva prioridade.
A tabela abaixo mostra os operadores em Python com a respectiva prioridade e o mesmo algoritmo acima pode ser usado para traduzir qualquer expressão ou comando.

Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call

`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not x`	Boolean NOT
`And`	Boolean AND
`Or`	Boolean OR
`lambda`	Lambda expression

Como exemplos, vamos traduzir para a notação pós-fixa as expressões abaixo:

```
a > b or c < d
a b > c d < or

a == b or c + d >= e * f and g
a b == c d + e f * >= g and or

a > b or b < c and c == d
a b > b c < c d == and or

b + c * d != x and not y > 0
b c d * + x != y 0 > not and
```

Operadores unários e operadores binários

Na notação usual os operadores + e – (adição e subtração) possuem o mesmo símbolo quando unários ou binários, embora tenham significado diferente. Essa diferença é resolvida pelos compiladores quando analisam o contexto da expressão.

Exemplo – Instruções da máquina virtual do Python

Considere o seguinte programa:

```
def teste(a, b, c):
    x = a + b * c
    y = a * (b + c)
    z = a * (b + c * (x + y))
    return x, y, z

print(teste(1, 2, 3))
```

Abaixo o código gerado para a interpretação da máquina virtual do Python no ambiente IDLE para a função teste(a, b, c). Ao lado uma explicação sobre cada instrução.

A máquina virtual do Python utiliza uma pilha para a execução do programa.

```
>>> import dis
>>> dis.dis(teste)
  2           0 LOAD_FAST                0 (a) - empilha a
              2 LOAD_FAST                1 (b) - empilha b
              4 LOAD_FAST                2 (c) - empilha c
              6 BINARY_MULTIPLY             - multiplique
              8 BINARY_ADD                  - some
             10 STORE_FAST               3 (x) - armazene em x

  3          12 LOAD_FAST                0 (a) - empilha a
             14 LOAD_FAST                1 (b) - empilha b
             16 LOAD_FAST                2 (c) - empilha c
             18 BINARY_ADD                 - some
             20 BINARY_MULTIPLY            - multiplique
             22 STORE_FAST               4 (y) - armazene em y

  4          24 LOAD_FAST                0 (a) - empilha a
             26 LOAD_FAST                1 (b) - empilha a
             28 LOAD_FAST                2 (c) - empilha a
             30 LOAD_FAST                3 (x) - empilha x
             32 LOAD_FAST                4 (y) - empilha y
             34 BINARY_ADD                 - some
             36 BINARY_MULTIPLY            - multiplique
             38 BINARY_ADD                 - some
             40 BINARY_MULTIPLY            - multiplique
             42 STORE_FAST               5 (z) - armazene em z
  5          44 LOAD_FAST                3 (x) - empilha x
             46 LOAD_FAST                4 (y) - empilha y
             48 LOAD_FAST                5 (z) - empilha z
             50 BUILD_TUPLE              3     - transforme em tupla
             52 RETURN_VALUE               - retorne tupla
>>>
```

Algoritmo para o cálculo do valor de uma expressão em notação pós-fixa

A vantagem da expressão na forma pós-fixa é que para se calcular o seu valor o algoritmo é bem simples. Basta varrê-la da esquerda para a direita e efetuar as operações com os dois últimos operandos, ou o último no caso de operadores unários. Para tanto, os operandos têm que ser empilhados à medida que aparecem, pois, a operação será aplicada aos dois últimos (se for operação binária) ou apenas ao último se for uma operação unária.

```
while (expressão pós-fixa não chegou ao fim):
    pega próximo elemento p;
    if (p é operando) empilha p;
    if (p é operador unário) {
      faz a operação com o elemento do topo da pilha;
    if (p é operador binário) {
      faz a operação com os 2 elementos do topo da pilha;
      neste caso a pilha diminui de 1 elemento;
```

O resultado da expressão estará no topo da pilha, que ao final se reduz a um único elemento.
Observe que no algoritmo de tradução, a pilha era de operadores enquanto que no algoritmo de cálculo a pilha é de operandos.

Exemplo - Considere a expressão aritmética abaixo:

```
a * (b + c * (d + e))
```

Que na notação pós-fixa ficaria:

```
a b c d e + * + *
```

Suponha que `a=1,b=2,c=3,d=4,e=5`.
Vamos calcular o valor da expressão usando a sua forma pós-fixa. Veja o cálculo abaixo e a evolução da pilha a medida que cada elemento é considerado. Acompanhe o cálculo usando o algoritmo acima:

A	b	c	d	e	+	*	+	*
				5				
			4	4	9			
		3	3	3	3	27		
	2	2	2	2	2	2	29	
1	1	1	1	1	1	1	1	29

Façamos o mesmo com a expressão: **(-b + √(b * b – 4 * a * c)) / (2 * a)**

Vamos usar o símbolo _ para indicar o menos unário.

Que fica com pós-fixa: **b _ b b * 4 a c * * - √ + 2 a * /**

Suponha que `a=3, b=-4, c=1`.

b	**_**	**b**	**b**	*****	**4**	**a**	**c**	*****	*****	**-**	**√**	**+**	**2**	**a**	*****	**/**

							1									
						3	3	3								
			-4		4	4	4	4	12					3		
		-4	-4	16	16	16	16	16	16	4	2		2	2	6	
-4	4	4	4	4	4	4	4	4	4	4	4	6	6	6	6	1