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`

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

- 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	L	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

A implementação de uma pilha num vetor de inteiros ficaria:

```
int pilha[MAX]; /* pilha[0] ... pilha[MAX-1] */
int topo;       /* indica elemento de cima da pilha */

/* inicia pilha */
void inicia_pilha () {
  topo = -1;
}

/* empilha novo elemento */
int empilha(int x) {
  if (topo == MAX-1) return -1; /* não há mais espaço */
  topo++;
  pilha[topo] = x;
  return 0;
}

/* desempilha novo elemento */
int desempilha(int *x) {
  if (topo < 0) return -1; /* pilha vazia */
  *x = pilha[topo];
  topo--;
  return 0;
}
```

Usamos acima uma pilha de inteiros (**`int`**), mas poderíamos usar as mesmas funções com uma pilha de elementos de qualquer um dos tipos básicos (**`char`**, **`float`**, **`double`**, etc.). Não haveria mudança mesmo se a pilha fosse de elementos do tipo **`struct`**. Veja abaixo:

```
/* cada um dos elementos da pilha */
struct elemento {
 ...
}
struct elemento pilha[MAX];
int topo;

/* inicia pilha */
void inicia_pilha () {
  topo = -1;
}

/* empilha novo elemento */
```

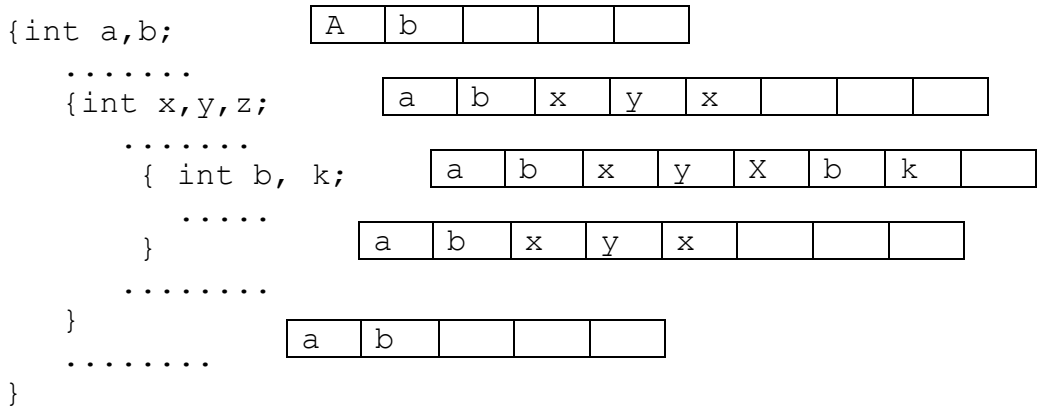
```
int empilha(struct elemento x) {
  if (topo == MAX-1) return -1; /* não há mais espaço */
  topo++;
  pilha[topo] = x;
  return 0;
}

/* desempilha novo elemento */
int desempilha(struct elemento *x) {
  if (topo < 0) return -1; /* pilha vazia */
  *x = pilha[topo];
  topo--;
  return 0;
}
```

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 na saída do bloco o espaço ocupado deve ser liberado para dar lugar a novas variáveis quando a execução entrar em novos blocos.



b) A tabela de símbolos de um compilador, onde ficam as variáveis declaradas tem uma estrutura de pilha. Note que variáveis podem ter o mesmo nome em blocos diferentes e usa-se sempre a que foi declarada por último. Assim a busca na tabela de símbolos deve ser feita sempre do fim para o começo. Além disso, quando termina um bloco onde há variáveis locais, estas desaparecem da tabela de símbolos.

c) 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).

d) 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.

e) Um caso especial 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 inventada 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) Podemos usar também os operadores unários. No caso do exemplo 7, usamos os operadores – (unário) e √ (raiz quadrada). No caso dos operadores unários – e + que possuem o mesmo símbolo dos binários correspondentes tem que se fazer a distinção usando outro símbolo para que não haja dúvida sobre a operação.
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 por que tem mais prioridade ou porque está entre parêntesis.

Algoritmo para transformar uma expressão para a notação pós-fixa

Varrendo a expressão norma 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) {
    pegue próximo elemento p;
    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.

```
int prioridade (char x) {
   switch (x) {
        case '+': return 1;
        case '-': return 1;
        case '*': return 2;
        case '/': return 2;
        case '^': return 3;
        case '(': return 4;   /* caso particular */
        case ')': return 5;   /* caso particular */
        default : return 0;  /* é operando */
   }
```

}

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 C

Nos exemplos acima, usamos os operadores mais usuais, mas todos os operadores em C tem a sua prioridade associada. Os operadores unários (+ e −), a atribuição (=), os operadores lógicos (`&&` e `||`), etc. seguem a mesma regra de cálculo com a sua respectiva prioridade.
A tabela abaixo mostra os operadores em C com a respectiva prioridade e o mesmo algoritmo acima pode ser usado para traduzir qualquer expressão ou comando em C.

Prioridade	Operador	Descrição	Direção
2	`() [] . -> ++ --`	Pós	Esquerda-Direita
	`++ -- ~ ! sizeof new delete`	Unários e pré	
3	`* &`	Ponteiros	Direita-Esquerda
	`+ -`	Unários	
4	`(type)`	type casting	Direita-Esquerda
5	`.* ->*`	Ponteiros	Esquerda-Direita
6	`* / %`	Multiplicativos	Esquerda-Direita
7	`+ -`	Aditivos	Esquerda-Direita
8	`<< >>`	Shift	Esquerda-Direita
9	`< > <= >=`	Relacional	Esquerda-Direita
10	`== !=`	Igualdade	Esquerda-Direita
11	`&`	Bitwise AND	Esquerda-Direita
12	`^`	Bitwise XOR	Esquerda-Direita
13	`|`	Bitwise OR	Esquerda-Direita
14	`&&`	Lógico AND	Esquerda-Direita
15	`||`	Lógico OR	Esquerda-Direita
16	`?:`	Condicioanl	Esquerda-Direita
17	`= *= /= %= += -= >>= <<= &= ^= |=`	Atribuição	Direita-Esquerda
18	`,`	Vírgula	Esquerda-Direita

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.
O ideal seria usar símbolos diferentes para operadores unários e binários.

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 operados tem 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:

```
abcde+*+*
```

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)**

Observe que agora estamos usando também operadores unários: **-** e **√.** Apenas para ilustrar vamos usar um símbolo _ em vez de – para o operador 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