capítulo 10

Pilhas

“*Os últimos serão os primeiros.*”

Bíblia (Mateus 20:1-16 / Mateus 19:23-30 / Marcos 10:23-31 / Lucas 13:23-30)

Todos nós já ouvimos essa frase, mas o que muitos não sabem é que nela está contida a ideia básica da estrutura de dados que veremos neste capítulo. Vamos perceber que as últimas informações guardadas serão as primeiras a serem utilizadas e, melhor, que essa característica nos ajudará a resolver alguns problemas computacionais.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender o funcionamento das pilhas;
- identificar situações e formas de utilizar pilhas;
- entender como os recursos computacionais utilizam-se das filas para solucionar problemas.

Para começar

A pilha é uma estrutura de dados muito frequente em nosso dia a dia, por isso seu entendimento é fácil e faz com que, de certa forma, sua implementação seja bastante lógica e intuitiva. Afinal, depois de entender um algoritmo em detalhes, a implementação é "fichinha", não é mesmo?

Para um bom entendimento, vamos começar ilustrando um cenário que pode ocorrer em algum momento de nossas vidas, principalmente na daqueles que já se mudaram de residência pelo menos uma vez. Alguns devem estar pensando: "Nossa, nunca me mudei! E agora?". Calma! Aqueles que nunca passaram por isso podem ficar tranquilos, pois a situação é facilmente imaginável. Certamente, algo bem parecido já ocorreu quando você decidiu arrumar algum cômodo do lugar onde mora.

Para entender como funciona a dinâmica de uma pilha, imagine a mudança de uma residência para outra. Considere que você esteja de mudança e que seus inúmeros livros sairão de uma estante acanhada no canto de seu quarto e irão para um lugar especial em sua nova biblioteca pessoal – afinal, bons livros, como este, devem ter um local de destaque em sua nova residência.

Imagine que, dias antes da mudança, a empresa contratada para transportar seus pertences lhe forneceu várias caixas de papelão, em três tamanhos diferentes. Os diferentes tamanhos visam a acomodar adequadamente

as roupas, os utensílios e, entre outras coisas, seus estimados livros. Ao analisar os tamanhos disponíveis, você constata que apenas um deles serve para esse transporte e mais: o tamanho de caixa disponível permite apenas que os livros sejam acondicionados uns sobre os outros, ou seja, que não é possível acomodar um livro ao lado de outros.

Papo técnico

Nessa analogia, a caixa representa um espaço de memória, e os livros são as informações que desejamos guardar.

Nesse cenário, o que fica claro para nós é que, dentro da caixa, um livro ficará em cima do outro, independentemente de seu tamanho, certo? Obviamente, esse é um cenário hipotético, mas não está muito fora da realidade.

Como ocorre esse acondicionamento dos livros nas caixas fornecidas? Quais passos você seguiria? Sem dúvida, o primeiro passo seria escolher uma das prateleiras da estante e decidir qual sequência de livros deslocar para a caixa. O segundo seria o processo de retirar, um a um, os livros da estante e colocá-los na caixa. O primeiro livro será colocado no fundo da caixa. Depois dele, os demais serão acomodados uns em cima dos outros, ou seja, no topo da pilha de livros que está sendo formada.

O processo de guardar livros nas caixas é hipoteticamente fixo, ou seja, um livro é selecionado na estante e sempre acomodado no topo da pilha de livros formada dentro da caixa. Dentro da caixa, sempre que um livro entra, vira o topo atual e permanece nesse posto até que outro livro seja acomodado em cima dele. Esse processo ocorrerá enquanto a caixa tiver espaço para acomodar mais livros. Quando a pilha de livros ocupar o espaço total da caixa, é melhor fechá-la e deixá-la pronta para a mudança.

Logo após a mudança, já em sua maravilhosa e novíssima biblioteca, as caixas com seus estimados livros estão disponíveis, aguardando para serem esvaziadas e, assim, deixar sua biblioteca recheada com seus exemplares.

Depois de escolher uma das caixas e abri-la, você começa a retirar um livro após o outro – antes de guardá-los nas novíssimas estantes, é necessário limpá-los. Por isso, o procedimento de desmontar essas caixas é retirar do topo da pilha de livros um volume, limpá-lo e acomodá-lo no novo local. Esse procedimento será feito para cada um dos livros que estão dentro da caixa. Portanto, um livro é retirado do topo da pilha, limpo e, por fim, acomodado na prateleira da estante.

Atenção

Repare que, na pilha de livros que se formou dentro da caixa, o último livro colocado foi o primeiro a ser retirado.

Conhecendo a teoria para programar

No exemplo hipotético, vimos que os livros foram acomodados no topo da pilha que existia dentro da caixa e, quando estavam sendo acomodados na nova biblioteca, foram retirados do topo da pilha de que faziam parte.

Depois de entender essa dinâmica toda, devemos pensar em como implementá-la. Mas, antes disso, gostaríamos de analisar algumas partes. Em nosso exemplo, a caixa era o artefato que fazia nossos livros serem armazenados. Para nós, em programação, a caixa faz o papel de memória, onde armazenamos informações (no nosso exemplo, os livros). Portanto, no exemplo temos caixas armazenando livros, ao passo que em programação temos memória armazenando quaisquer informações que desejarmos.

A informação que será armazenada na memória pode ser de qualquer tipo primitivo, mas também é aceito qualquer tipo abstrato desenvolvido em seu programa.

Antes de iniciar as implementações de funções importantes, gostaríamos de discutir o tamanho de nossa pilha. Em nossos exemplos usaremos um vetor como alocação de memória para trabalhar a pilha. Um vetor, ao ser declarado, deve ter tamanho definido, que será o tamanho máximo da

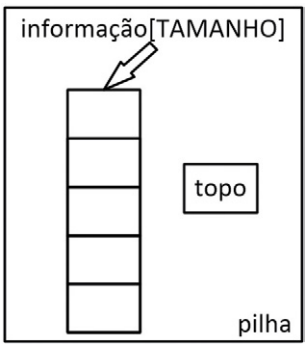

FIGURA 10.1: Pilha representada graficamente.

nossa pilha, ou seja, a quantidade máxima de elementos que caberão ali. A Figura 10.1 mostra como ficará a nossa pilha.

Nesse caso, faremos uso da declaração de uma constante que conterá esse valor. O código ficará assim:

```
#define TAMANHO 4
```

Além do tamanho, o vetor também deve ser um tipo. No nosso caso, nós o criaremos com um tipo primitivo, inteiro, mas nossa pilha será definida em um tipo abstrato, contendo uma variável que controlará a posição do topo da pilha e o vetor de inteiros que servirá de pilha em nossos exemplos. O código apresentado a seguir demonstra a definição dessa estrutura:

```
struct pilha
{
      int topo;
      int informacao[TAMANHO];
};
```

A variável que controla o topo da pilha sempre indicará a posição dessa pilha que receberá a informação seguinte. Dessa forma, se iniciarmos seu valor com 0 (zero), obviamente ela nunca será menor do que isso, porém, na outra extremidade, quando a pilha estiver cheia, terá valor igual ao tamanho máximo. Calma. Como assim terá o valor igual ao tamanho máximo? (No nosso caso, 4.) Sim, isso mesmo. Lembre-se de que se trata de um vetor de quatro posições, portanto, as posições existentes iniciam em 0 (zero) e terminam em 3 (três). Agora, fica bem claro que, quando a variável controladora do topo estiver com valor igual ao tamanho máximo, isso significa que nossa pilha está cheia e que não podemos receber mais valores.

Levando em conta que não é possível apenas inserir valores, isto é, que também é possível retirá-los, é importante salientar que essa variável indica uma posição que não tem um valor a ser considerado. Portanto, para obter a informação que se encontra no topo, basta atualizar essa variável, decrementando seu valor para chegar a uma posição com valor e, aí sim, retirá-lo.

Na Figura 10.2, vemos como exemplo a evolução de uma pilha. Representamos uma pilha de quatro posições. A Figura 10.2(a) demonstra como essa

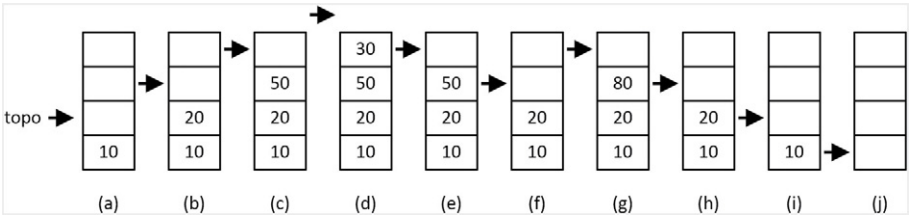

FIGURA 10.2: Evolução do topo da pilha durante ações de *push* e *pop* na pilha.

pilha ficaria com uma informação armazenada. Repare que temos o valor 10 (dez) na primeira posição da pilha (index zero do vetor) e o ponteiro para o topo encontra-se em uma posição sem informação (index número um do vetor), pois o próximo elemento, se existir, entrará nessa posição. As Figuras 10.2(b, c, d) ilustram a evolução da pilha ao receber três informações em sequência, que a deixam cheia. Na posição caracterizada na Figura 10.2(d), temos a informação 30 completando a pilha, e nossa variável de indicação de topo está apontando para uma posição que não existe.

Fazer a variável de topo armazenar uma posição que não existe é errado? O que você acha? Lembre-se de que, se você usar de fato essa posição do vetor, ela não existirá, portanto, você acessará uma região de memória inválida para seu vetor. De outro lado, apenas ter a informação guardada para indicar que a pilha está cheia, sem acessar nenhuma posição, é possível e extremamente válido.

Bem, já vimos uma característica importante de nossa pilha, ou seja, quando ela está cheia – Figura 10.2(d). Agora, analisaremos as Figuras 10.2(e, f). Repare que o topo está descendo – na verdade, decrementando – e alguns valores estão sendo retirados do topo. Essas duas figuras nos mostram que, para retirar uma informação do topo da pilha, devemos primeiramente decrementar a variável indicativa de posição de topo, guardar a informação que está naquela posição e, depois, apagá-la, colocando um valor inválido ou zerando.

A Figura 10.2(g) mostra o caso de termos um novo elemento sendo colocado na pilha, e as Figuras 10.2(h) e (i) mostram retiradas simples de informações da pilha. Já a Figura 10.2(j) demonstra a última retirada possível dessa sequência, pois deixa a pilha vazia. Repare que a posição de topo está na posição zero do vetor, o que caracteriza essa situação.

Espero que esse exemplo tenha deixado bastante claro os principais passos que devemos seguir para inserir uma informação em uma pilha ou retirá-la dela.

A partir deste momento, não falaremos mais em *inserir* e *retirar*; falaremos de como essas operações são conhecidas no mundo técnico, ou seja, para dizer que inserimos algo na pilha, falaremos em *push*; e para dizer que retiramos algo da pilha, usaremos o termo *pop*.

Função *push*: inserindo novo elemento na pilha

Continuando nosso código, gostaríamos de discutir a função *push*. Nela, temos a necessidade de inserir nova informação dentro da nossa pilha. Para tal, devemos receber a pilha propriamente dita, pois, se formos adicionar uma informação, devemos saber onde, concorda? Além disso, precisamos saber o que será adicionado.

Imagine que seu professor solicite a você que se levante e escreva algo. A primeira coisa que você perguntará é: "Onde vou escrever?" Ao receber

a resposta dele, de que será na lousa, você logo formula mentalmente a segunda pergunta: "E o que eu vou escrever ali?". Suponha que o professor solicite que você escreva quantos títulos de campeonato brasileiro de futebol seu time de coração já conquistou. Ótimo, já temos local para escrever e também conteúdo. Agora ficou fácil, não é? Basta encaminhar-se até o local indicado e escrever o que lhe foi solicitado.

Pois bem, com a função *push* ocorre a mesma coisa. Ela receberá uma pilha já definida por quem está chamando essa função (a lousa do nosso exemplo) e também a informação que será inserida no topo (quantidade de títulos conquistados por seu time de coração no campeonato brasileiro de futebol). Caso seja possível, o cabeçalho dessa função fica assim:

```
void push(struct pilha *p, int info)
```

Após receber a pilha e a informação que deve ser colocada nela, é necessário verificar se essa pilha não está cheia. Se estiver, não há o que fazer, a não ser dar uma mensagem informativa; caso contrário, isto é, se a pilha puder receber essa informação, devemos realizar o procedimento de inserção.

Para verificar, cria-se um simples código condicional questionando se o topo é menor que o tamanho máximo da pilha. assim, o condicional ficaria dessa forma:

```
if(p->topo < TAM)
```

Sendo o topo menor que o tamanho máximo, devemos fazer a inserção da informação dentro da pilha, seguindo os passos já explicados e exemplificados na Figura 10.3, ou seja, colocar o valor na posição sinalizada pelo topo (ver Figura 10.3(a)) e atualizar o topo (ver Figura 10.3(b)), incrementando-o e deixando-o pronto para a próxima inserção de valores.

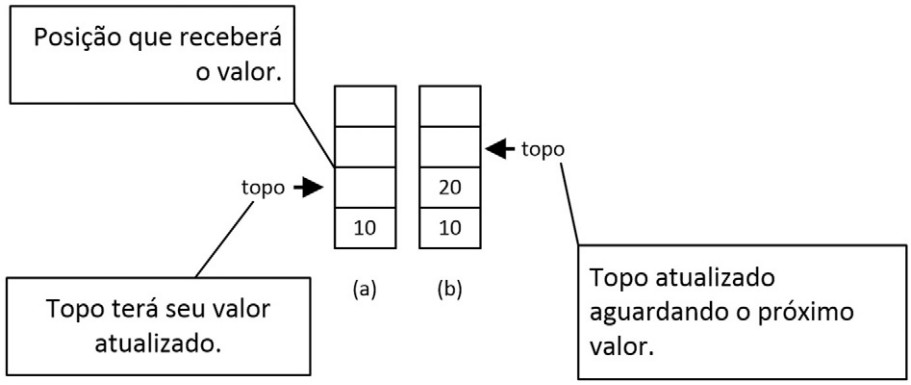

FIGURA 10.3: Ilustração dos passos para inserção de um novo elemento na pilha.

As duas linhas de código que fazem esse procedimento são as seguintes:

```
p->informacao[p->topo] = info;
p->topo++;
```

Se o topo indicar que a pilha está cheia, resta apenas enviar uma mensagem na tela; assim, o código ficaria:

```
else
    printf("Pilha cheia, impossivel INSERIR elemento!\n\n");
```

A função completa do *push* ficaria assim:

Código 10.1

```
void push(struct pilha *p, int info)
{
        //Verifica se há espaço na pilha para armazenar nova informação
        if(p->topo < TAM)
        {
                printf(">>>>> INSERIR >>> : %d\n", info);
                p->informacao[p->topo] = info; //a variável "topo" aponta para a posição onde a nova
                // informação deve ser inserida, portanto, devemos fazer a atribuição.
                p->topo++;  //atualiza a variável que controla o "topo" para que se possa receber a próxima informação
        }
        else
                printf("Pilha cheia, impossivel INSERIR elemento!\n\n");
}
```

Função pop: retirando elemento na pilha

Agora é a vez de discutirmos e apresentarmos a função *pop*. Ela exerce a funcionalidade de retirar uma informação do topo da pilha, como já discutimos. Você se lembra disso, né? Certamente que sim.

A função *pop* deve retornar uma informação retirada do topo da pilha. Por isso, ela tem o tipo primitivo *int* no seu cabeçalho, indicando que retornará um número inteiro. Já em seus parâmetros, ele recebe

um endereço de pilha, pois também fará manipulações nela (mudando o topo e retirando um elemento). Considerando isso, nosso cabeçalho ficará assim:

```
int pop(struct pilha *p)
```

Nessa função, ao contrário da função *push*, precisamos verificar se estão solicitando a remoção de um elemento de uma pilha vazia. Para fazer isso, basta verificar se o topo não está na posição zero da pilha. Essa verificação é feita por uma simples condicional, como:

```
if(p->topo > 0)
```

Se houver elemento(s) a ser(em) retirado(s), deve-se realizar o procedimento de retirada, ou seja, atualizar o topo decrementando-o, guardando o valor que está na posição indicada em uma variável, atribuindo um valor que não representa nada dentro da pilha e retornando dessa função com o valor que foi retirado do topo. A Figura 10.4 mostra os passos descritos. Na Figura 10.4(a) temos a pilha com o elemento que será retirado do topo (número 20), o posicionamento do ponteiro de

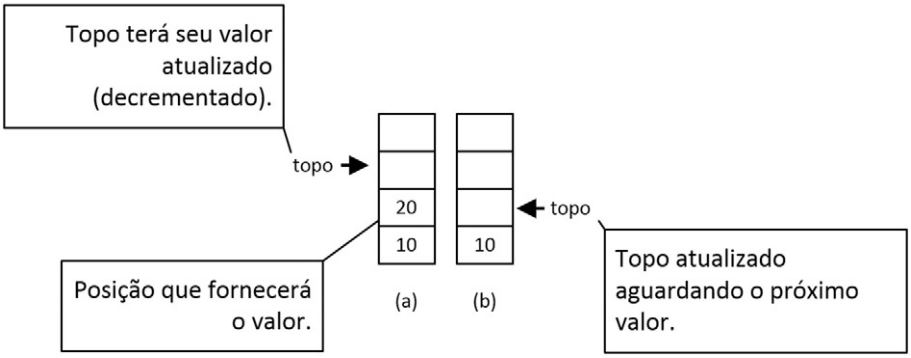

FIGURA 10.4: Passos para a remoção de um elemento da pilha.

topo; a Figura 10.4(b) mostra a situação dessa pilha após a retirada do elemento.

```
p->topo--;
valor = p->informacao[p->topo];
p->informacao[p->topo] = -1;
return valor;
```

Se a condicional indicar que não há mais elementos a serem retirados, imprimiremos uma mensagem ao usuário avisando-o disso.

Assim, o código ficaria:

```
else
      printf("Pilha vazia, impossivel RETIRAR elemento!\n\n");
```

Caso o código chegue até esse ponto, isso significa que não houve retorno de um elemento válido a quem chamou essa função, portanto, é melhor retornar um valor inválido.

Após essas explicações dos trechos de códigos, apresentamos a função proposta completa:

Código 10.2

```
int pop(struct pilha *p)
{
      int valor; //variável que guardará a informação que será retirada da pilha
      //Verifica se existe alguma informação a ser retirada na pilha
      if(p->topo > 0)
      {
            p->topo--; //atualiza o topo para a posição onde temos a última informação inserida
            valor = p->informacao[p->topo]; //obtem a informação
            p->informacao[p->topo] = -1; //atualiza a posição com a informação que não representa nada
            printf("<< RETIRAR <<<<< : %d\n", valor);
            return valor; //retorna o valor para quem chamou a função pop
      }
      else
            printf("Pilha vazia, impossivel RETIRAR elemento!\n\n");
      return -1;
}
```

Função principal: usando *push* e *pop*

Antes de chegar à função principal, para explicar melhor nossos testes, gostaríamos de apresentar uma função que visa única e exclusivamente a mostrar, no terminal de execução desse programa, os elementos que estão dentro dessa pilha. É preciso deixar bem claro que essa função não existe quando se trata de pilha, pois dentro das regras citadas o único elemento visível dessa estrutura de dados é seu elemento caracterizado como topo.

Para essa função, usaremos basicamente a liberdade que nos é dada pelo vetor para sair da posição zero e, avançando posição a posição, chegar até uma delas imprimindo seus elementos. Nesse caso, deixaremos a posição zero visando chegar àquela que está sinalizada como a posição topo.

A função de impressão sugerida é:

Código 10.3

```
int i = 0, valor_retirado;
struct pilha pi; //declaração da variável que guardará a nossa pilha
```

Agora, vamos para a função principal, em que utilizaremos todas as funções apresentadas até aqui. É nessa função que vamos criar nossa pilha e também a variável auxiliar que guardará a informação retirada dessa pilha, quando for solicitada. Para tais declarações, sugiro as seguintes linhas de código:

```
void imprimir(struct pilha p)
{
	printf("********************** PILHA **********************\n\n");
	//repetição entre as posições que contém informações para que sejam impressas na tela
	for(int i = 0; i < p.topo; i++)
		printf(" %d",p.informacao[i]);
	printf("\n\n");
	printf("*****************************************************\n\n");
}
```

Após a criação da pilha e das variáveis auxiliares, é necessário inicializá-la. Certo, mas inicializar a pilha em que sentido? Você se lembra de que tínhamos uma variável interna que controlava a posição do vetor que representava o topo da pilha? Pois então. Essa variável deve ser inicializada com o valor zero. Você recorda por que justamente o valor zero? É porque essa posição receberá a primeira informação a entrar na pilha.

Aliás, e o vetor que representa a pilha, não precisa ser inicializado? Na verdade, isso não é uma regra, mas é uma boa prática. Nesse caso, um valor que representará um valor inválido dentro desse exemplo é -1, portanto, ele fará parte de todas as posições inicialmente.

Para essas inicializações necessárias à nossa pilha, sugerimos as seguintes linhas de código:

```
pi.topo = 0;
for(i = 0; i < TAM; i++)
       pi.informacao[i] = -1;
```

Após essas partes iniciais e importantes da nossa função principal, sugerimos que se faça uma sequência de inserções na pilha, chegando ao ponto de inserir um elemento a mais que ela suporte para testar o comportamento da função *push*. Para facilitar o entendimento do nosso teste, propomos fazer a sequência de inserções na pilha sugeridas na Figura 10.2. O código ficaria assim:

```
//sequência de inserções de valores na pilha
push(&pi,10);
imprimir(pi);
push(&pi,20);
imprimir(pi);
push(&pi,50);
imprimir(pi);
push(&pi,30);
imprimir(pi);
//esta inserção é um teste, pois passa do tamanho da pilha
push(&pi,40);
imprimir(pi);
```

Para a função *push*, temos que passar como primeiro parâmetro o endereço de memória onde se encontra a nossa pilha, pois faremos a manipulação de seu conteúdo e gostaríamos que essa pilha fosse atualizada para todos os que a usarão. Por isso, o primeiro parâmetro é o &*pi*, que envia para a função *push* o endereço da pilha declarada e chamada, nessa função principal, simplesmente de *pi.*

Agora que fizemos as inserções necessárias, sugiro passar para uma sequência que retira da pilha as informações nela contidas. Assim como a função *push*, a função *pop* manipula a mesma pilha, portanto, deve receber seu endereço. Aproveitando a ideia já utilizada na função *push*, propomos também uma retirada a mais da pilha para testar o comportamento da função *pop*. Repare que essa função retorna um valor toda vez que é chamada, por isso colocamos a variável valor_retirado para receber a atribuição do valor que retorna dessa chamada de função. Nesse caso, tal variável não é utilizada para nada, mas poderia ser empregada por alguma lógica específica que usasse as pilhas como parte de sua possível solução.

A sequência que sugiro é:

```
//sequência de retiradas dos valores da pilha
valor_retirado = pop(&pi);
imprimir(pi);
valor_retirado = pop(&pi);
imprimir(pi);
valor_retirado = pop(&pi);
imprimir(pi);
valor_retirado = pop(&pi);
imprimir(pi);
//este valor não deve existir
valor_retirado = pop(&pi);
imprimir(pi);
```

Portanto, o código completo dessa função principal é:

Código 10.4

```
int _tmain(int argc, _TCHAR* argv[])
{
	int i = 0, valor_retirado;
	struct pilha pi; //declaração da variável que guardará a nossa pilha

	//Inicializando a pilha recém criada.
	//A variável "topo" guarda a posição que receberá a informação e depois será atualizada novamente
	pi.topo = 0;
	//Inicializando as posições do vetor que funcionará como pilha
	for(i = 0; i < TAM; i++)
		pi.informacao[i] = -1;

	//sequência de inserções de valores na pilha
	push(&pi,10);
	imprimir(pi);
	push(&pi,20);
	imprimir(pi);
	push(&pi,50);
	imprimir(pi);
	push(&pi,30);
	imprimir(pi);
	//esta inserção é um teste, pois passa do tamanho da pilha
	push(&pi,40);
	imprimir(pi);

	//sequência de retiradas dos valores da pilha
	valor_retirado = pop(&pi);
	imprimir(pi);
	valor_retirado = pop(&pi);
	imprimir(pi);
	valor_retirado = pop(&pi);
	imprimir(pi);
	valor_retirado = pop(&pi);
	imprimir(pi);
	//este valor não deve existir
	valor_retirado = pop(&pi);
	imprimir(pi);

	return 0;
}
```

Apresentaremos, a seguir, o resultado da execução desse código. Observe que as funções *push* e a *pop* se comportaram como esperado, pois emitiram frases informando que não foi possível inserir e retirar elementos, e, em nenhum dos casos, o programa perdeu dados, muito menos teve um comportamento inadequado. Observe a saída do programa proposto (ver Figura 10.5)

O trecho de código que exerce esses passos é:

```
>>>>> INSERIR >>> : 10
********************* PILHA *********************
 10
***********************************************

>>>>> INSERIR >>> : 20
********************* PILHA *********************
 10 20
***********************************************

>>>>> INSERIR >>> : 50
********************* PILHA *********************
 10 20 50
***********************************************

>>>>> INSERIR >>> : 30
********************* PILHA *********************
 10 20 50 30
***********************************************

Pilha cheia, impossivel INSERIR elemento!
********************* PILHA *********************
 10 20 50 30
***********************************************

<< RETIRAR <<<<< : 30
********************* PILHA *********************
 10 20 50
***********************************************

<< RETIRAR <<<<< : 50
********************* PILHA *********************
 10 20
***********************************************

<< RETIRAR <<<<< : 20
********************* PILHA *********************
 10
***********************************************

<< RETIRAR <<<<< : 10
********************* PILHA *********************


***********************************************

Pilha vazia, impossivel RETIRAR elemento!
********************* PILHA *********************


***********************************************
```

FIGURA 10.5: Resultado da execução do código teste.

Pilha na prática: rádio? Não, expressão matemática mesmo!

As pilhas são usadas como parte da solução de um relevante tópico em ciência da computação, que é a análise de expressões matemáticas, principalmente ao pensar que essa análise pode fazer parte de um compilador de linguagem, aumentando ainda mais sua importância.

As expressões são definidas de três maneiras: *infixo*, *prefixo* e *posfixo*. Os prefixos *in, pre* e *pos* referem-se, única e exclusivamente, ao posicionamento dos operadores matemáticos em relação a seus dois operandos. Veja os exemplos a seguir:

- infixa: X + Y + Z
- posfixa: X Y Z + +
- prefixa: + + X Y Z

A expressão, da forma como estamos acostumados a ler e usar, é representada pela forma infixa, como X + Y. Como descrito no parágrafo anterior, os prefixos referem-se ao *posicionamento dos operadores em relação aos operandos*, portanto, se essa mesma expressão do exemplo fosse escrita na forma *prefixa*, a expressão ficaria + X Y; de outro lado, na forma *posfixa*, a expressão ficaria X Y +.

Até aqui tudo muito simples, certo? Pois é. Mas como em computação nada é tão simples assim, vamos adicionar algo para pensar em como resolver. Outro exemplo no formato *infixo* seria X + Y * Z. Nós temos a nítida impressão de que devemos resolver primeiro Y * Z para depois somar o resultado ao *X*, mas como fazer o computador saber o mesmo que nós? Podemos dizer, então, que sabemos que as operações de multiplicação e divisão têm *precedência* sobre as operações de soma e subtração. Nesse exemplo, a forma *posfixa* resultaria em X Y Z * +, pois, dessa forma, o Y e o Z seriam multiplicados e o resultado seria somado com X, obtendo, assim, a resposta pretendida.

Conversão de infixo para posfixo

Para que você possa entender como essa transformação acontece e o que a pilha tem a ver com tudo isso, apresentamos um algoritmo para fazer exatamente essa conversão, usando a estrutura de dados que acabamos de ver, ou seja, a pilha.

Antes de iniciar, devemos supor que exista uma função chamada precd(op1,op2), que retorna um *tipo booleano*, sendo *TRUE* se op1 tiver precedência sobre op2, e *FALSE* se ocorrer o contrário. Por exemplo, precd("*"," + ") retornará *TRUE*, e precd(" + ","*") retornará *FALSE*; afinal, a soma não é uma operação precedente da multiplicação.

Perfeito. Agora já sabemos identificar as precedências, o que é muito importante para esse algoritmo. Nesse momento, faz-se necessário explicar que, para analisar uma expressão, precisamos percorrê-la caractere por caractere e, ao fazer isso, usar os caracteres encontrados como fonte de dados para um vetor de caracteres e uma pilha. É nesse vetor de caracteres que vamos encontrar o final de nossa expressão em formato *posfixo*; já a pilha é

uma estrutura de dados auxiliar para que possamos organizar corretamente os operadores e também os parênteses (quando eles existirem nas expressões analisadas).

Nesse momento, usaremos uma expressão sem os parênteses, tal como a expressão inicialmente apresentada: X + Y * Z. A Tabela 10.1 representa, passo a passo, como vamos manipular caractere por caractere e onde os colocaremos para encontrar no final a nossa expressão *posfixa*.

TABELA 10.1: Passo a passo para transformar uma expressão infixa em posfixa (neste caso, sem parênteses e com precedência de operador já resolvida)

	Símbolo	String	Pilha
1	X	X	
2	+	X	+
3	Y	X Y	+
4	*	X Y	+ *
5	Z	X Y Z	+ *
6		X Y Z *	+
7		**X Y Z * +**	

Observe que, ao ler um símbolo da expressão, caso ele seja um operando (linhas 1, 3 e 5 da Tabela 10.1), é imediatamente encaminhado para a *string*, que formará a expressão *posfixa* final (linha 7). Contudo, se for um operador, esse símbolo é colocado na pilha (linhas 2 e 4), ou seja, o operador é empilhado usando a função *push* de pilha. Ao final, o que está na pilha é desempilhado usando o *pop* e jogado na *string* de saída, finalizando a expressão posfixa.

Fácil né? Foi mesmo, mas vamos complicar um pouco. Nada de mais, mas vai requerer um pouco mais de atenção. Propomos que usemos a seguinte expressão: X /(Y - Z). Qual será a expressão *posfixa* dessa infixa? Será que você conseguiria

rascunhar sozinho a tabela que vamos apresentar a seguir? Acredito que sim, mas antes devemos lhe explicar uma nova regra, pois nessa expressão temos parênteses, e isso muda algumas coisas.

Assim que você encontrar um "abre parêntese", basta empilhá-lo como se fosse um operando; porém, ao encontrar um "fecha parêntese" terá que desempilhar até achar um "abre parêntese" (isto é, desempilhar o "abre parêntese" que está empilhado).

Agora tente fazer essa tabela.

Apresentamos a Tabela 10.2 como uma sugestão de solução para a conversão de infixo para *posfixo* da expressão X / (Y – Z).

Só para terminar essa breve introdução sobre a utilização das pilhas em análise de expressões, por que não atribuímos valores para os operandos? Por exemplo, **X = 10, Y = 5 e Z = 3**. A expressão X Y Z - / será percorrida caractere por caractere, da esquerda para a direita, até encontrar um operador; quando isso ocorre, são pegos os dois operandos imediatamente antes desse operador, e é feita a operação matemática envolvida. O primeiro operando é o sinal de subtração – estamos falando de X **Y Z** - /, nesse caso, 5 – 3, resultando no valor 2, e a expressão é

TABELA 10.2: Conversão de infixo para posfixo da expressão X / (Y – Z)

	Símbolo	String	Pilha
1	X	X	
2	/	X	/
3	(	X	/ (
4	Y	X Y	/ (
5	–	X Y	/ (–
6	Z	X Y Z	/ (–
7	)	X Y Z –	/
8		**X Y Z – /**	

Observe que o "(" e "–" foram desempilhados. O operador "–" foi para a String, o parêntese foi apenas consumido.

atualizada: X 2 /, agora é só realizar essa operação, ou seja, 10 / 2, portanto, o resultado dessa expressão com esses valores hipotéticos seria igual a 5. Pois bem, atribua valores a X Y e Z e brinque um pouco com essa expressão.

Vamos programar

Vimos vários conceitos que permeiam a pilha como estrutura de dados. O que você acha de olharmos esses mesmos conceitos, só que aplicados a outras linguagens bastante importantes? Divirta-se com os códigos em Java e Python.

Java

Código 10.1

```
private static void push(Stack st, int info)
{
   try
   {
       st.push(info);
   }
   catch (Exception e)
   {
       System.out.println("Pilha cheia, impossivel INSERIR elemento! Erro: "+e.toString() );
   }
}
```

Código 10.2

```
private static int pop (Stack st)
{
   try
   {
       return (int) st.pop();
   }
   catch (Exception e)
   {
       System.out.println("Pilha vazia, impossivel RETIRAR elemento! Erro: "+e.toString() );
   }
   return -1;
}
```

Código 10.3

```
private static void imprimir(Stack st)
{
   System.out.println("Stack = "+st);
}
```

Código 10.4

```
public static void main(String[] args)
{
        Stack stack = new Stack();

        //sequência de inserções de valores na pilha
        push(stack,10);
        imprimir(stack);
        push(stack,20);
        imprimir(stack);
        push(stack,50);
        imprimir(stack);
        push(stack,30);
        imprimir(stack);

        //sequência de retiradas dos valores da pilha
        pop(stack);
        imprimir(stack);
        pop(stack);
        imprimir(stack);
        pop(stack);
        imprimir(stack);
        pop(stack);
        imprimir(stack);
        pop(stack);
        imprimir(stack);
}
```

Python

Código 10.1

```
def push(self, info):
    if self.topo < TAM:
      print "Inserir : %d" % (info)
      self.informacao[self.topo] = info # a variavel "topo" aponta para a posicao onde a nova informacao deve ser
                                        # inserida, portanto, devemos fazer a atribuicao.
      self.topo+=1 #atualiza a variavel que controla o "topo" para que se possa receber a proxima informacao
    else:
      print "Pilha cheia"
```

Código 10.2

```
def pop(self):
    if self.topo > 0:   #Verifica se existe alguma informacao a ser retirada na pilha
      self.topo-=1   #atualiza o topo para a posicao onde temos a ultima informacao inserida
      valor = self.informacao[self.topo]   #obtem a informacao
      self.informacao[self.topo]=-1   #atualiza a posicao com a informacao que nao representa nada
      print "Retirar : %d" % (valor)
      return valor   #retorna o valor para quem chamou a funcao pop
    else:
      print "Pilha vazia"
    return -1
```

Código 10.3

```
def imprimir(self):
    print "Pilha"
    #repeticao entre as posicoes que contem informacoes para que sejam impressas na tela
    i = 0
    while i < self.topo:
      print "%d" % (self.informacao[i])
      i+=1
```

Código 10.4

```
#DEFINICAO DA ESTRUTURA
class Pilha:
       def __init__(self):
              self.informacao = [-1]*TAM
              self.topo = 0
       .
       .
       .
       #funções: push, pop e imprimir aparecem aqui
       .
       .
       .
       pilha = Pilha()
       //sequência de inserções de valores na pilha
       push();
       imprimir();
       push(20);
       imprimir();
       push(50);
       imprimir();
       push(30);
       imprimir();

       //sequência de retiradas dos valores da pilha
       pop();
       imprimir();
       pop();
       imprimir();
       pop();
       imprimir();
       pop();
       imprimir();
       pop();
       imprimir();
```

Para fixar

1. Faça um teste de mesa nas funções *push* e *pop*, levando em consideração a sequência abaixo. Ao final, escreva o que fica na fila:
   a. entrada do número 2;
   b. entrada do número 8;

c. entrada do número 6;
d. retirar um número;
e. entrada do número 1;
f. retirar um número.

2. Construa um programa que auxilie na criação dos adesivos que são colocados em carros especiais, tal como ambulâncias e carros de bombeiros, para serem olhados pelo retrovisor; por exemplo: AMBULANCIA ←→ AICNALUBMA
3. Utilizando-se de pilhas, faça um procedimento para saber se existe o mesmo número de entradas dos caracteres *c* e *d*.

Para saber mais

As pilhas são primordiais em vários mecanismos computacionais. Um desses usos foi tema de nossa discussão nos parágrafos anteriores, isto é, sua utilização em compiladores para que possamos interpretar expressões matemáticas e fazer nossos programas funcionarem.

As pilhas também são amplamente utilizadas e importantes no *sistema operacional* de nossas máquinas. Com certeza, usamos e criamos *softwares*, que são modulados para que o sistema operacional possa controlar os módulos que estão chamando outros módulos. Ele simplesmente empilha essas funções, procedimentos ou métodos (dependendo do paradigma de programação utilizado pelo programador). Lembrar como as pilhas funcionam torna mais fácil entender por que o sistema operacional se utiliza delas para se organizar nesse momento: ao empilhar a chamada feita, quando ela termina, basta desempilhá-la e passar a usar o próximo topo.

Caso queira aprofundar o estudo das diversas formas de usar as pilhas para manipular expressões matemáticas, leia o Capítulo 2 do livro *Estrutura de dados usando C* (Tenenbaum, 2004).

Navegar é preciso

No link http://www.frontiernet.net/~prof_tcarr/StackMachine/Stack.html existe um simulador que mostra graficamente o que acontece com as funções *push* e *pop*. Embora nele existam outras funcionalidades, é muito interessante ver o funcionamento dessas duas funções.

Exercícios

1. Transforme as expressões a seguir em forma posfixa.

a. ((X + Y) * Z) + w

b. (X + (Y – Z)) * W

c. X * (Y + Z – W)

d. (X * Y) / (Z / W)

e. (X * Y) * (Z * W)

2. Transforme as expressões a seguir em forma infixa.

a. X Y –

b. X Y Z + -

c. X Y + Z *

d. X Y – Z + G H I - + +

3. Implemente a função chamada de *precd*, introduzida e explicada neste capítulo.

Glossário

Operadores: símbolos matemáticos que representam soma, subtração, multiplicação e outros.

Operandos: caracteres que simbolizam as variáveis que podem assumir valores.

PUSH: palavra usada pelos programadores para indicar que se deseja fazer uma inserção na pilha.

POP: outra palavra muito usada pelos programadores para indicar que se deseja retirar uma informação da pilha.

Referência bibliográfica

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.

O QUE VEM DEPOIS

Das estruturas de dados apresentadas até o momento, chegou a vez de aprender a usar uma nova forma de pensar e armazenar informação. Nos próximos capítulos deste livro serão apresentadas as Árvores (Capítulo 11), Árvores n-árias (Capítulo 12), Árvores Balanceadas (Capítulo 13), além dos grafos (Capítulo 14). Bons estudos!