

Unidade 3

Pilhas, Alocação Seqüencial e Estática

O que é um *stack overflow*?

Nossos Objetivos nesta Unidade:

- Entender o que é e para que serve uma estrutura do tipo Pilha, o que é alocação seqüencial de memória, e o que é alocação estática de memória, no contexto do armazenamento de conjuntos de elementos;
- Desenvolver a habilidade para implementar uma estrutura de armazenamento do tipo Pilha, como um Tipo Abstrato de Dados, com alocação seqüencial e estática de memória;
- Desenvolver a habilidade para manipular Pilhas através dos operadores definidos para o Tipo Abstrato de Dados Pilha.

O Que É uma Pilha?

A Pilha da qual estamos falando não é uma pilha no sentido de bateria para o controle remoto. Estamos falando de uma pilha de pratos, pilha de potes, pilha de livros, pilha de cartas. Ou seja, o sentido é de empilhar coisas (Figura 3.1).





Pilha de Potes	Pilha de Livros	Pilha de Pratos	Pilha de Cartas
			
Imagens: Yew Tree Gallery (potes), iStockPhoto (livros), Darren Maurer (pratos), e Barbosa, Miyoshi, e Gomes (cartas)			

Figura 3.1 Ilustração do conceito de Pilha - sentido empilhar

Em uma pilha de pratos, não é possível retirar um prato que não esteja no topo da pilha, se não a pilha vai

desmoronar. Também não é possível inserir um prato fora do topo. Essencialmente, esse é o funcionamento de uma pilha.

Definição: Pilha

Pilha é uma estrutura para armazenar um conjunto de elementos, que funciona da seguinte forma:

- Novos elementos entram no conjunto, exclusivamente, no topo da pilha;
- O único elemento que posso retirar da pilha em um dado momento, é o elemento do topo.

Do Inglês: *Stack, LIFO*

Uma Pilha (em Inglês: *Stack*) é uma estrutura que obedece o critério L.I.F.O.: *Last In, First Out*. Ou seja, o último elemento que entrou no conjunto será o primeiro a sair.

Uma Pilha é um conjunto ordenado de elementos, ou seja, a ordem dos elementos no conjunto é importante. Se eu tenho três elementos em uma pilha, A, B e C, e se eles entraram na pilha nessa ordem, o elemento que estará no topo da pilha será o elemento C. E se eu quiser retirar um elemento nesse momento, o único elemento que poderei retirar da pilha será exatamente o elemento C (Figura 3.2).

C
B
A



Figura 3.2 Pilha com três elementos A, B e C; C está no topo da pilha

Considerando ainda o exemplo da Figura 3.2, se quisermos inserir um elemento D na pilha nesse momento, este elemento passaria a ser o elemento do topo da pilha, conforme mostra a Figura 3.3.

D
C
B
A



Figura 3.3 Pilha com quatro elementos A, B, C e D; D está no topo da pilha

E se a ordem de inserção dos elementos na pilha tivesse sido outra, por exemplo, B, C, D, A, a pilha resultante seria outra (Figura 3.4).

A
D



Figura 3.4 Pilha com quatro elementos, ordem de inserção: B, C, D A

Simulação 3.1: Operações de uma Pilha

Teste o funcionamento das operações Empilha e Desempilha, interagindo com a [Simulação 3.1: Operações de uma Pilha](#).

Para Que serve uma Pilha? O Exemplo da Chamada de Subprogramas

Ao elaborar um programa de computador você já se deparou com um erro de execução chamado *stack overflow*? Sabe o que significa *stack overflow*? Sabe como ocorre? É um erro bastante comum. Se ainda não aconteceu com você, provavelmente ainda vai acontecer.

Um computador está executando um trecho de programa A, e durante a execução de A encontra o comando *Call B*. Ou seja, uma chamada a um subprograma B. O computador precisa parar de executar A, executar B, e retornar ao programa A no ponto onde parou. Como o computador faz para lembrar a posição exata para onde deve retornar? Essa questão pode se complicar se tivermos várias chamadas sucessivas.

Considere o exemplo da Tabela 3.1. Temos um programa principal, A, e 3 subprogramas: B, C e D. Ao iniciarmos a execução de A, na linha 1 temos o comando *Print A*, que imprime a letra A. Depois temos o comando *Call C*, ou seja, uma chamada ao subprograma C. Nesse ponto temos que interromper a execução de A e iniciar a execução do subprograma C. Ao finalizar a execução do subprograma C (comando *Return*, subprograma C linha 3), precisamos retomar a execução de A na linha 3, porque as linhas 1 e 2 de A já foram executadas.

Programa A (principal)	Subprograma B	Subprograma C	Subprograma D
1 print A 2 call C 3 call B 4 call D 5 return	1 call C 2 print B 3 call D 4 call C 5 return	1 print C 2 call D 3 return	1 print D 2 return
Print A = imprime a letra A; call C = chama o subprograma C; return = fim do programa ou subprograma			

Tabela 3.1 Programa A e subprogramas B, C e D

A propósito, tente executar, de cabeça, o programa A, naturalmente incluindo as chamadas sucessivas aos subprogramas. Anote a seqüência de impressão das letras. Você é um bom computador? Tem um bom processador? Tem uma boa memória? Quais letras serão impressas, e em qual seqüência? Como você chegou a esse resultado?

Pilha de Execução

Para resolver esse problema o computador usa uma Pilha de Execução. Essa pilha funciona assim:

- A cada comando *Call*, ou seja, a cada chamada de subprograma, o computador empilha (operação de

inserir um elemento na pilha) a posição do programa para a qual a execução deverá retornar depois de executar o subprograma que está sendo chamado;

- A cada comando *Return*, ou seja, a cada vez que a execução de um subprograma chega ao fim, o computador desempilha (operação que retira um elemento da pilha) a posição do programa para a qual a execução deverá retornar, e passa a executar a partir desse endereço.

A chamada e execução do programa A, e chamadas sucessivas aos subprogramas B, C e D, conforme apresentado na Tabela 3.1, resultará em uma Pilha de Execução. A Tabela 3.2 apresenta essa Pilha de Execução, começando em uma situação 1, passando para a situação 2, situação 3, e assim por diante, até a situação final (20).

	A3	C3 A3	A3		A4	B2 A4	C3 B2 A4	B2 A4	A4
situação 1:vazia	situação 2	situação 3	situação 4	situação 5:vazia	situação 6	situação 7	situação 8	situação 9	situação 10
B4 A4	A4	B5 A4	C3 B5 A4	B5 A4	A4		A5		
situação 11	situação 12	situação 13	situação 14	situação 15	situação 16	situação 17:vazia	situação 18	situação 19:vazia	situação 20:fim

Tabela 3.2 Sequência de situações da pilha de execução

Vamos executar o Programa A?

Inicialmente a pilha está vazia. Executamos a linha A1, o que **imprime a letra A**. Passamos a executar a linha A2 – comando *Call C*. Nesse ponto temos que empilhar a posição do programa para a qual deveremos retornar (posição A3), e passar a executar o subprograma C. A pilha de execução passa a situação 2.

Passamos a executar o subprograma C. Na linha C1 temos *Print C*, e então **imprimimos a letra C**. Depois temos na linha C2 o comando *Call D*. Nesse ponto empilhamos o endereço de retorno, C3, e passamos a executar o subprograma D. A pilha de execução passa para a situação 3.

Iniciamos a execução do subprograma D. Linha D1: *Print D (Imprime D)*; linha D2: *Return*. Quando chegamos a um comando *Return*, desempilhamos da pilha de execução, e passamos a executar a partir desse endereço. O endereço desempilhado será sempre o elemento do topo da pilha. No nosso caso, será o elemento C3. Passamos a executar a partir de C3, e a pilha passa para a situação 4.

Executando a partir de C3, temos um novo comando *Return*. Desempilhamos novamente, agora o endereço A3, e a pilha passa para a situação 5 (vazia novamente). No endereço A3 temos o comando *Call B*. Empilhamos A4, e passamos a executar o subprograma B. A pilha de execução passa para a situação 6. Já no subprograma B temos *Call C*. Empilhamos B2 (situação 7) e passamos a executar C.

Simulação 3.2: Pilha de Execução

Está difícil entender essa execução? Veja se fica mais fácil através da interação com a [Simulação 3.2: Pilha de Execução](#).

Em C temos *Print C (imprime C)* e *Call D* (situação 8). Em D, *Print D (imprime D)* e *Return* (situação 9). Passamos a executar em C3 que é outro comando *Return* (situação 10).

Passamos a executar em B2, que é um comando *Print B (imprime B)*, e a seguir temos *Call D* (empilha B4, passa para a situação 11). Em D *Print D (imprime D)*, *Return*, desempilha (situação 12) e vamos para B4. Em B4 temos *Call C* (empilha B5, situação 13). Em C1 *Print C (imprime C)* e *Call D* (empilha C3, situação 14). Em D, *Print D (imprime D)*, e *Return* (desempilha C3, situação 15), vamos para C3, que é outro *Return* (desempilha B5, situação 16). B5 é outro *Return*, desempilha A4, situação 17, pilha vazia. Em A4 temos *Call D* (empilha A5, situação 18). Executando D, temos *Print D (imprime D)*, e *Return* (desempilha A5, situação 19, pilha vazia). Executando A5 temos outro comando *Return*, só que desta vez com a pilha vazia. Não temos como desempilhar de uma pilha vazia. Temos então o final da execução.

As letras impressas, na seqüência, foram: **A, C, D, C, D, B, D, C, D, D**. Você havia conseguido chegar a esse resultado de cabeça?

E o tal *stack overflow*, como ocorre? Aguarde um pouco mais e descobrirá.

Para que serve uma pilha então? Note, nesse exemplo, que a pilha de execução serviu para armazenar elementos (endereços de retorno) em uma ordem específica. Que ordem? *Last In, First Out*, ou seja, o último elemento que entrou, será o primeiro elemento a sair do conjunto.

Funcionalidade de uma Pilha. Tipo Abstrato de Dados Pilha

A partir do exemplo das chamadas de subprogramas, é possível identificar o uso de duas operações: empilha e desempilha. A essas duas operações, vamos acrescentar outras duas: operação para criar uma pilha, e operação para testar se uma pilha está vazia. Vamos definir também os principais parâmetros que devem ser considerados nessas operações (Tabela 3.3).

Operações e Parâmetros	Funcionamento
Pilha.Empilha(QualPilha, QualElemento, DeuCerto?)	Empilha o elemento passado como parâmetro QualElemento, na pilha passada no parâmetro QualPilha. O parâmetro DeuCerto? indica se a operação foi bem sucedida ou não.
Pilha.Desempilha(QualPilha, QualElemento, DeuCerto?)	Desempilha (retira o elemento do topo) da pilha passada no parâmetro QualPilha, retornando o valor do elemento que foi desempilhado no parâmetro QualElemento. O parâmetro DeuCerto? indica se a operação foi bem sucedida ou não.
Pilha.Vazia?(QualPilha)	Verifica se a pilha passada como parâmetro (QualPilha) está ou não vazia (vazia = sem nenhum elemento).
Pilha.Cria(QualPilha)	Cria uma pilha, iniciando sua situação como vazia.

Tabela 3.4 Funcionalidade de uma pilha

Lembra do conceito de Tipos Abstratos de Dados? Lembra para que serve esse conceito, na prática? Serve para ser utilizado na fase de Projeto de um programa. Quando queremos armazenar um conjunto de elementos em um programa, primeiramente definimos os dados que serão armazenados e os operadores que podem ser aplicados para manipulação desses dados. A partir daí, nos demais módulos do programa, vamos considerar esse conjunto de dados a serem armazenados dentro de uma caixa preta que só pode ser manipulada através dos operadores definidos. Esses operadores são como os botões para aumentar o volume e para mudar os canais da TV, lembra dessa analogia?

Exercício 3.1 Transferir cartas de uma Pilha P1 para uma Pilha P2

Estamos na fase de Projeto do programa FreeCell. E no FreeCell temos uma pilha de cartas. A melhor maneira de considerar uma pilha de cartas, na fase de Projeto, é como um Tipo Abstrato de Dados. Quais dados devem ser armazenados? Cartas (como na Figura 3.5). Quais operações podemos aplicar na manipulação desses dados? Os operadores de uma pilha, segundo definimos na Tabela 3.4: cria, vazia?, empilha e desempilha.

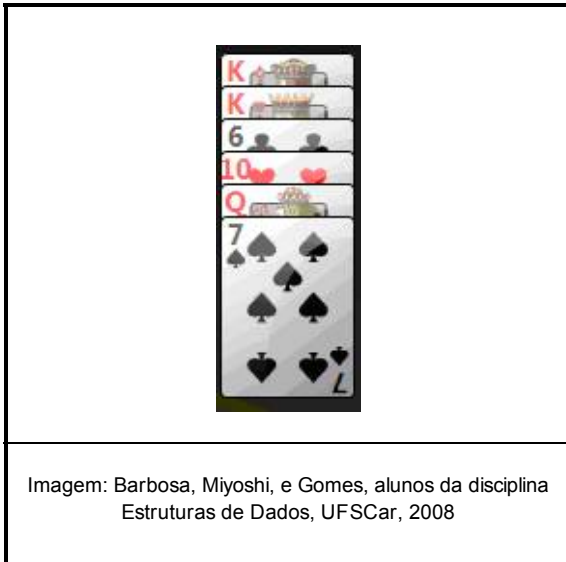


Imagem: Barbosa, Miyoshi, e Gomes, alunos da disciplina Estruturas de Dados, UFSCar, 2008

Figura 3.5 Uma pilha de cartas

Elabore agora um algoritmo que transfere cartas de uma Pilha P1 para uma Pilha P2. Considere que as pilhas P1 e P2 já existem, ou seja, não precisam ser criadas. Para elaborar esse algoritmo, use os operadores definidos na Tabela 3.4. Você terá que elaborar esse algoritmo sem ter a menor idéia de como o Tipo Abstrato de Dados Pilha é efetivamente implementado.

Você poderá ver uma solução para esse exercício logo abaixo, mas não consulte essa solução ainda. Tente elaborar você mesmo a solução. É assim que desenvolvemos nossa capacidade de solucionar problemas.

```
Algoritmo Transfere-Cartas-de-P1-para-P2 ( Parâmetros por referência P1 e P2 do tipo Pilha )  
/* transfere cartas da Pilha P1 para a Pilha P2, já existentes */
```

```
fim do Algoritmo
```

Conseguiu propor uma solução? Muito bom se conseguiu! Se não conseguiu, consulte a solução abaixo, e procure prestar bem a atenção em como o exercício foi resolvido.

```
Algoritmo Transfere-Cartas-de-P1-para-P2 ( Parâmetros por referência P1 e P2 do tipo Pilha )
```

```
Algoritmo Transfere-Cartas-de-P1-para-P2 ( Parametros por referencia P1 e P2 do tipo Pilha )
```

```
/* transfere cartas da Pilha P1 para a Pilha P2, já existentes */
```

```
Variável auxiliar Carta do tipo Elemento-da-Pilha
```

```
Variável auxiliar DeuCerto? tipo boolean
```

```
Enquanto Pilha.Vazia( P1 ) = falso Faça
```

```
    Pilha.Desempilha( P1, Carta, DeuCerto? )
```

```
    Pilha.Emilha( P2, Carta, DeuCerto? )
```

```
fim do Algoritmo
```

A tradução do algoritmo para o português: enquanto a pilha P1 não estiver vazia, retiramos um elemento da pilha P1 através da operação Pilha.Desempilha. O valor desempilhado de P1 retornará no parâmetro Carta. A seguir empilhamos esse mesmo valor na pilha P2, acionando a operação Pilha.Emilha. Repetimos isso, através do comando Enquanto, até que a pilha P1 se torne vazia.

Note que não temos ainda a menor ideia de como as operações do TAD Pilha são efetivamente implementadas. Mas, manipulando a Pilha exclusivamente através das operações, realmente não é preciso saber detalhes de implementação. Consideramos a pilha de cartas como uma caixa preta, a qual manipulamos apenas através dos operadores (ou botões da TV). É assim que projetamos programas utilizando o conceito de Tipos Abstratos de Dados. Achou interessante?

Alterar os dados de um cadastro sem ser por uma operação definida na especificação do TAD é possível? É possível, mas fazendo isso estamos abrindo a TV com uma chave de fenda para aumentar o volume. Manipular uma pilha de cartas sem ser por operações precisamente definidas na especificação do TAD Pilha é possível? É possível também. Estaremos aumentando o volume abrindo a TV com uma chave de fenda, nesse caso também.

Vamos praticar um pouco mais a manipulação de uma pilha apenas através dos seus operadores? Proponha uma solução. Se estiver difícil, forme um grupo de estudos, discuta com o grupo. Do grupo, provavelmente, sairão excelentes soluções. Se ainda restarem dúvidas, discuta com o seu tutor, professor e demais colegas em um fórum criado exatamente para isso.

Exercícios

Exercício 3.2 Desenvolva um algoritmo para testar se uma pilha P1 tem mais elementos que uma pilha P2. Considere que P1 e P2 já existem.

Exercício 3.3 Desenvolva um algoritmo para inverter a posição dos elementos de uma pilha P. Você pode criar pilhas auxiliares, se necessário. Mas o resultado precisa ser dado na pilha P.

Exercício 3.4 Desenvolva um algoritmo para testar se duas pilhas P1 e P2 são iguais. Duas pilhas são iguais se possuem os mesmos elementos, na mesma ordem. Voe pode utilizar pilhas auxiliares também, se necessário.

Alocação de Memória para Conjuntos de Elementos

O contexto da disciplina Estruturas de Dados é o armazenamento temporário de conjuntos de elementos, em um programa. Vamos então apresentar os conceitos de alocação seqüencial de memória, e alocação

estática de memória, sempre no contexto do armazenamento temporário de conjuntos de elementos.

estática de memória, sempre no contexto do armazenamento temporário de conjuntos de elementos.

Definição: Alocação Sequencial de Memória (para um Conjunto de Elementos)

- Os elementos ficam, necessariamente, em seqüência (ou seja, um ao lado do outro, em posições adjacentes) na memória.

Definição: Alocação Estática de Memória (para um Conjunto de Elementos)

- Todo o espaço de memória a ser utilizado (para armazenar os elementos) é reservado (alocado) no início da execução do programa ou módulo (e não no decorrer da execução);
- Esse espaço de memória permanece reservado durante toda a execução do programa ou módulo, independente de estar sendo efetivamente utilizado ou não.

Se queremos implementar uma pilha com alocação sequencial de memória, seus elementos precisam estar em seqüência na memória, ou seja, em posições adjacentes (Figura 3.6).

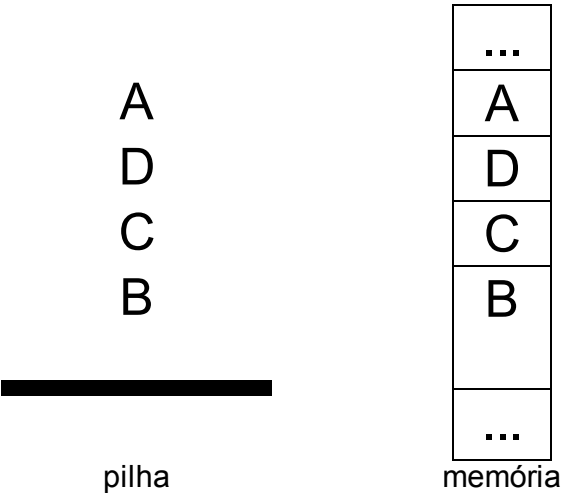


Figura 3.6 Pilha com quatro elementos e seu armazenamento em posições adjacentes de memória: alocação sequencial.

Implementação de Pilha com Alocação Sequencial e Estática

Olhando para a Figura 3.6, em especial a representação do armazenamento dos elementos da pilha na memória, como vocês acham que podemos implementar uma pilha com alocação sequencial? Que tipo de variável composta podemos utilizar?

Para implementar uma pilha com alocação sequencial, utilizamos um vetor para armazenar os elementos da pilha em posições adjacentes de memória. Na Figura 3.7 temos o esquema de uma pilha implementada através de um vetor denominado *Itens*, e de um indicador para o topo da pilha, denominado *Topo*.



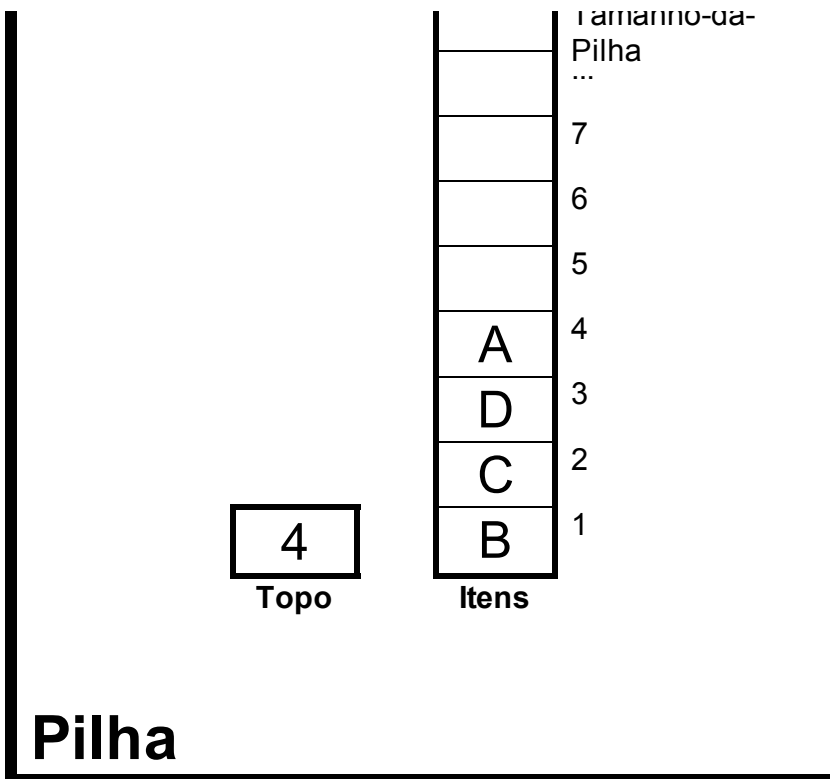


Figura 3.7 Esquema da implantação de uma pilha com alocação seqüencial e estática de memória.

O tipo Pilha pode ser implementado, portanto, através de um Registro (*Struct* na linguagem C, *Record* na linguagem Pascal) com dois campos: Itens e Topo. Topo do tipo inteiro, e Itens do tipo vetor de 1 até Tamanho-da-Pilha, de elementos do tipo *Char*. Naturalmente, ao invés do tipo *Char*, podemos declarar uma pilha de elementos do tipo Carta, por exemplo, e de outros elementos, conforme necessário.

Exercício 3.5 Implementação da Operação Empilha

A operação Empilha recebe como parâmetro o valor de um elemento a ser inserido na pilha (no topo da pilha). O parâmetro DeuCerto? deve indicar se o elemento foi inserido com sucesso, ou não.

Tente propor uma solução a esse exercício. Você encontrará uma solução logo abaixo, mas não olhe a solução. Procure realmente fazer, e depois olhar a solução. É assim que se aprende.

Empilha(parâmetro por referência P do tipo Pilha, parâmetro Elemento do tipo Char, parâmetro por referência DeuCerto? do tipo Boolean)

/* Empilha o elemento Elemento, passado como parâmetro, na pilha P também passada como parâmetro. O parâmetro DeuCerto? deve indicar se a operação foi bem sucedida ou não. */

Empilha - Possível Solução

Empilha(parâmetro por referência P do tipo Pilha, parâmetro Elemento do tipo Char, parâmetro por referência DeuCerto? do tipo Boolean)

/* Empilha o elemento Elemento, passado como parâmetro, na pilha P também passada como parâmetro. O parâmetro DeuCerto? deve indicar se a operação foi bem sucedida ou não. */

Se P.Topo = Tamanho-da-Pilha

Então DeuCerto = Falso

Senão P.Topo = P.Topo + 1

P.Itens[P.Topo] = Elemento

DeuCerto = Verdadeiro

Traduzindo o algoritmo para o português: Se ainda houver espaço na pilha para incluir mais um elemento, então vamos inseri-lo. Para isso, incrementamos o Topo da pilha ($P.Topo = P.Topo + 1$), e inserimos o Elemento no vetor Itens, na posição indicada pelo Topo ($P.Itens[P.Topo] = Elemento$). Nesse caso a operação deu certo ($DeuCerto = Verdadeiro$).

Simulação 3.3: Empilha

Está difícil entender o algoritmo Empilha? Veja se fica mais fácil através da interação com a [Simulação 3.3: Empilha](#).

E o Stack Overflow ?

A operação empilha vai falhar se não houver espaço no vetor para inserir o elemento. Se não couberem mais elementos na pilha, tentamos inserir e não conseguimos. Nesse caso o parâmetro DeuCerto deve retornar falso. Se não tivéssemos esse teste, poderíamos inserir elementos além da capacidade da nossa pilha, ocasionar um 'estouro de pilha', ou ainda *Stack Overflow*. É isso que significa *Stack Overflow*!

Faça agora os exercícios abaixo, implementando as operações Desempilha, Vazia? e Cria. Proponha uma solução. Se estiver difícil, forme um grupo de estudos, discuta com o grupo. Do grupo, provavelmente, sairão excelentes soluções. Se ainda restarem dúvidas, discuta com o seu tutor, professor e demais colegas em um fórum criado exatamente para isso.

Exercício 3.6 Desempilha

Desempilha(parâmetro por referência P do tipo Pilha, parâmetro por referência Elemento do tipo Char, parâmetro por referência DeuCerto? do tipo Boolean)

/* Desempilha (retira o elemento do topo) da pilha P, passada no parâmetro, retornando o valor do elemento que foi desempilhado no parâmetro Elemento. O parâmetro DeuCerto? deve indicar se a operação foi bem sucedida ou não. A operação só não será bem sucedida se tentarmos retirar um elemento de uma pilha vazia. */

Exercício 3.7 Vazia?

Vazia?(parâmetro por referência P do tipo Pilha)

/* Verifica se a pilha passada como parâmetro (P) está ou não vazia (vazia = sem nenhum elemento). */

Exercício 3.8 Cria

Cria(parâmetro por referência P do tipo Pilha)

/* Cria uma pilha, iniciando sua situação como vazia. */

Na operação Cria devemos inicializar o topo da pilha com o valor zero. Veja na Figura 3.7 que os Itens começam a ser armazenados no vetor Itens a partir do índice 1. A operação Vazia? simplesmente irá testar se o topo da pilha está apontando para o valor zero. Se estiver em zero, a pilha estará vazia. Se não estiver em zero, a pilha não estará vazia. Note que se o vetor começar em zero, e não em 1 (esse é o padrão na linguagem C), o valor do topo para pilha vazia deve ser alterado para -1.

Na operação Desempilha, devemos pegar o valor do elemento que está no topo, e retornar na variável Elemento. Depois disso devemos decrementar (diminuir em 1) o topo da pilha.

No final desta unidade temos a implementação completa de uma pilha com alocação seqüencial e estática em notação algorítmica, na linguagem C, na linguagem C++, e na linguagem Pascal. Não se esqueça: tente propor sua própria solução antes de consultar essas soluções no final da unidade.

Operação Elemento-do-Topo

A operação Elemento-do-Topo retorna o valor do elemento que está no topo da pilha, mas sem retirá-lo do topo. Como você implementaria essa operação?

Exercício 3.9 Elemento-do-Topo

Elemento-do-Topo (parâmetro por referência P do tipo Pilha, parâmetro por referência Elemento do tipo Char, parâmetro por referência DeuCerto? do tipo Boolean)

/* Retorna o valor do elemento do topo da pilha P, no parâmetro Elemento, mas não retira esse elemento da pilha. O parâmetro DeuCerto? deve indicar se a operação foi bem sucedida ou não. A operação só não será bem sucedida se tentarmos obter o valor do elemento do topo quando a pilha estiver vazia. */

Podemos implementar essa operação, basicamente, de duas maneiras. A primeira maneira é dependente da implementação, e a segunda maneira é independente da implementação. Na maneira dependente da implementação, simplesmente consultamos o valor do elemento `P.Itens[P.Topo]`. Essa implementação da operação funcionará para essa implementação que fizemos da pilha, com alocação seqüencial e estática. Mas não continuará funcionando se mudarmos a implementação da pilha. Nas Unidades 6 e 7 do Livro Virtual implementaremos uma pilha com outra técnica de implementação. Se implementarmos a operação Elemento-Do-Topo dessa forma dependente da implementação (consultando `P.Itens[P.Topo]`), teremos que reescrever essa operação para que funcione com a implementação de pilha que faremos nas Unidades 6 e 7. Na segunda forma de implementar essa operação Elemento-do-Topo, podemos acionar a operação desempilha, pegar o valor do elemento, e logo em seguida acionar a operação empilha, retornando o elemento à pilha, conforme a solução abaixo.

Elemento-do-Topo (parâmetro por referência P do tipo Pilha, parâmetro por referência Elemento do tipo Char, parâmetro por referência DeuCerto? do tipo Boolean)

/* Retorna o valor do elemento do topo da pilha P, no parâmetro Elemento, mas não retira esse elemento da pilha. O parâmetro DeuCerto? deve indicar se a operação foi bem sucedida ou não. A operação só não será bem sucedida se tentarmos obter o valor do elemento do topo quando a pilha estiver vazia. */

Desempilha (P, Elemento, DeuCerto?)
Se DeuCerto?
Então Empilha (P, Elemento, DeuCerto?)

‘Mas professor, para que complicar, se podemos simplesmente consultar o valor do elemento do topo (`Elemento = P.Itens[P.Topo]`)’? Se fizermos isso, estaremos com uma solução dependente da implementação. Pode ser uma solução eficiente e aparentemente mais simples, mas tem um sério defeito: ela é dependente da implementação, e não funcionará para a implementação de pilha que vamos fazer nas unidades 6 e 7. Mas se implementarmos usando as operações Desempilha e Empilha, conforme a solução mostrada acima, funcionará tanto para essa implementação de Pilha seqüencial e estática, quanto para a implementação que faremos nas Unidades 6 e 7. Ou seja, uma solução independente da implementação, como a mostrada na solução acima, é legal porque proporciona portabilidade e reusabilidade de código.

Vamos pensar também nas soluções dos exercícios 3.1, 3.2, 3.3 e 3.4. O exercício 3.2, por exemplo, verifica se uma pilha P1 possui mais elementos do que uma pilha P2. Podemos ter uma solução que simplesmente compara o valor dos topos das pilhas P1 e P2 (Se `P1.Topo > P2.Topo...` então...). Uma segunda solução, poderia desempilhar todos os elementos de cada pilha, colocando os elementos em pilhas auxiliares, contando quantos elementos tem em cada pilha. Depois os elementos poderiam ser devolvidos às pilhas originais. Qual dessas soluções você considera mais interessante?

A primeira solução é dependente da implementação. Ela abre a TV com uma chave de fenda para aumentar o volume. A segunda solução é independente da implementação. Quando propusemos essas soluções aos exercícios 3.1 a 3.4, nem sabíamos ainda como a pilha seria implementada. Não nos preocupamos com os detalhes da implementação da pilha. Aumentamos o volume da pilha pelo botão de volume, e não com a chave de fenda. E conseguimos, assim, uma solução independente da implementação, que proporciona maior portabilidade e reusabilidade de código. Essa é uma grande vantagem da utilização do conceito de Tipos Abstratos de Dados. Lembra dessas vantagens ([Unidade 2](#))?

O mesmo raciocínio vale para os exercícios 3.1, 3.3 e 3.4.

Operações Primitivas Versus Não Primitivas

Operações Primitivas Versus Não Primitivas

As operações não primitivas de um Tipo Abstrato de Dados são aquelas que podem ser implementadas através do acionamento das operações primitivas. As operações Elemento-do-Topo, Mais-Elementos, Iguais? e Transfere-Elementos são operações não primitivas. E a melhor forma de implementar uma operação não primitiva, visando proporcionar portabilidade e reusabilidade, é através do acionamento de operações primitivas (no nosso caso: Empilha, Desempilha, Vazia? e Cria).

Exercícios de Fixação

1. O que é, e para que serve uma pilha?
2. O que significa [alocação seqüencial de memória](#) para um conjunto de elementos?
3. O que significa [alocação estática de memória](#) para um conjunto de elementos?
4. Faça o [esquema de uma implementação seqüencial e estática de uma pilha](#), e descreva seu funcionamento.
5. Desenvolva uma operação para inverter a posição dos elementos de uma pilha P.
6. Desenvolva uma operação para testar se uma pilha P1 tem mais elementos que uma pilha P2.
7. Desenvolva uma operação para testar se duas pilhas P1 e P2 são iguais.
8. Desenvolva uma operação para transferir elementos de uma pilha P1 para uma pilha P2.
9. Desenvolva uma operação que retorna o valor do elemento do topo de uma pilha, sem desempilhar.

Exercício de Preparação para o Trabalho 1 (FreeCell)

10. A pilha que implementamos nesta Unidade é uma pilha “burra”, ou seja, não verifica o valor do elemento que está sendo empilhado. No exemplo do FreeCell, temos pilhas “inteligentes”, que não permitem ao jogador inserir um elemento qualquer na pilha. Procurando analisar possíveis soluções para o trabalho 1, FreeCell, procure adaptar a implementação de “pilha burra” para uma “pilha inteligente”, segundo as necessidades do FreeCell.

Pensamento do Dia

A primeira solução é dependente da implementação. Ela abre a TV com uma chave de fenda para aumentar o volume. A segunda solução é independente da implementação. Com ela, não nos preocupamos com os detalhes da implementação; aumentamos o volume da TV pelo botão de volume, e não com a chave de fenda. E conseguimos, assim, uma solução que proporciona maior portabilidade e reusabilidade de código. Essa é a grande vantagem da utilização do conceito de Tipos Abstratos de Dados.

Referências (Imagens)

- DARREN MAURER [HTTP://MINIATUREMASTERPIECES.BLOGSPOT.COM/2007_08_01_ARCHIVE.HTML](http://miniaturemasterpieces.blogspot.com/2007_08_01_archive.html)
- Federal FreeCell – desenvolvido por Carlos Eduardo Barbosa, Ronaldo Akio Miyoshi, e Marco Diniz Garcia Gomes, alunos da disciplina Estruturas de Dados, UFSCar, 2008
- IStockPhoto: http://portuguesbrasileiro.istockphoto.com/file_closeup/object/5194365_pile_of_books.php?id=5194365&SearchLang=PT_BR
- Yew Tree Gallery (http://www.yewtreegallery.com/past_06_may-july.php)

Solução de exercícios de aula – implementação do TAD Pilha

```

const StackSize      = 100;

type StackElement    = char;
   Pilha              = record
                           Topo  : integer;
                           Itens : array[1..StackSize] of StackElement;
                       end;

var P : Pilha;          { aloca memória para a pilha P, alocação estática}

procedure Create( var P : Pilha );
begin
    P.Topo := 0;
end;

function IsEmpty( var P : Pilha );
begin
    if P.Topo = 0
    then IsEmpty := true
    else IsEmpty := false;
end;

Procedure Push( var P : Pilha; X : StackElement );
begin
    P.Topo := P.Topo + 1;
    P.Itens[ P.Topo ] := X;
end;

procedure Pop( var P : Pilha; var X : StackElement; var Erro : boolean );
begin
    if IsEmpty( P )
    then Erro := true
    else begin
        Erro := false;
        X := P.Itens[ P.Topo ];
        P.Topo := P.Topo - 1;
    end;
end;

```

```

#define StackSize 100

struct Pilha {
    int Topo;
    char Itens[ StackSize ];
};

struct Pilha P;          { aloca memória para a pilha P, alocação estática }

void pop(struct stack *ps, int *px, int *erro) {
    if ( empty(ps) ) {
        *erro = TRUE;
        return;
    }
    *Erro = FALSE;
    *px = ps -> itens[ ps -> top- ];

    return; }

```

```

void create( struct stack *ps) {
    ps -> top = -1;
    return;
}

int empty(struct stack *ps) {
    if (ps->top == -1)
        return(TRUE);
    else return(FALSE);
}

void push(struct stack *ps, int X) {
    ps -> itens[ ++ (ps -> top) ] = X;
    return;
}

```

Notação C++

```
#define StackSize 100
```

```
Class Pilha {
```

```
    private:
```

```
        int Topo;
        char Itens[ StackSize ];
```

```
    public:
```

```
        pilha ( ) {
            topo=-1 }

```

```
        int isempty ( ) {
            if (topo == -1)
                return 1;
            else return 0; }

```

```
        void push( int x, int &erro) {
            if (topo==StackSize-1)
                erro = 1;
            else {
                erro = 0;
                topo++;
                Itens[topo] = x; } }

```

```
        void pop( int &x, int &erro) {
            if (topo == -1 )
                erro = 1;
            else {
                erro = 0;
                x = Itens[ topo ];
                topo--; } }

```

```
Pilha P;                                { aloca memória para a pilha P, alocação estática }
```

Notação algorítmica...

```
declare...
```

constante StackSize = 100

tipo Pilha = registro
 Topo do tipo inteiro
 Itens do tipo array[1..StackSize] do tipo char
 Fim do registro

variável P do tipo Pilha { aloca memória para a pilha P, alocação estática }

rotina Create(referência P do tipo pilha)
 P.Topo = 0

função IsEmpty(referência P do tipo Pilha) resultado da função tipo boolean
 se P.Topo == 0
 então resultado = true
 senão resultado = false

rotina Push(x do tipo char, referência P do tipo Pilha)
 P.Topo = P.Topo + 1
 P.Itens[P.Topo] = X

rotina Pop(referência P tipo Pilha, referência X tipo char, referência Erro tipo boolean)
 se IsEmpty(P)
 então Erro = true
 senão Erro = false
 X = P.Itens[P.Topo]
 P.Topo = P.Topo - 1