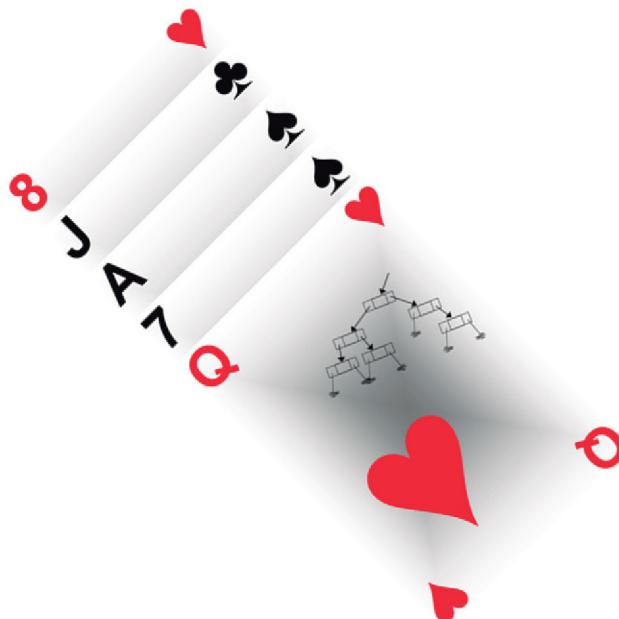


Capítulo 2

Pilhas com Alocação Sequencial e Estática



Seus objetivos neste capítulo

- Entender o que é e para que serve uma estrutura do tipo Pilha.
- Entender o significado de Alocação Sequencial e Alocação Estática de Memória, no contexto do armazenamento temporário de conjuntos de elementos.
- Desenvolver habilidade para implementar uma estrutura do tipo Pilha, como um Tipo Abstrato de Dados (TAD), com Alocação Sequencial e Estática de Memória.
- Desenvolver habilidade para manipular Pilhas através dos Operadores definidos para o TAD Pilha.
- Iniciar o desenvolvimento do seu primeiro jogo.

2.1 O que é uma Pilha?

No contexto deste livro, o termo Pilha diz respeito a uma pilha de pratos, pilha de livros ou pilha de cartas. Ou seja, o sentido é de empilhar uma coisa sobre a outra.

Em uma pilha de pratos, se você tentar tirar um prato do meio da pilha, a pilha poderá desmoronar. Também será bem complicado você inserir um prato no meio da pilha. O mais natural é retirar pratos do topo da pilha e inserir pratos sempre no topo da pilha.

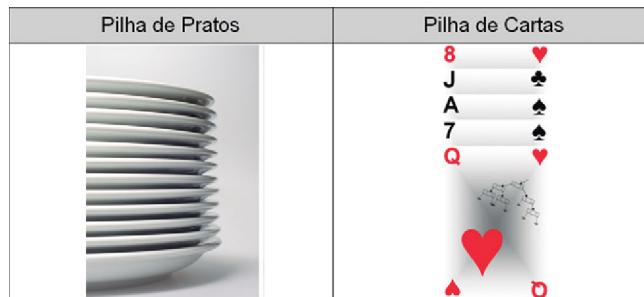


Figura 2.1 Ilustração do conceito de Pilha — sentido de empilhar.

Em essência, é assim que funciona uma Pilha: novos elementos entram sempre no topo; se quisermos retirar um elemento, retiramos sempre o elemento do topo.

Definição: Pilha

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

- Novos elementos entram no conjunto sempre no topo da Pilha.
- O único elemento que pode ser retirado da Pilha em dado momento é o elemento do topo.

Do inglês Stack, L.I.F.O.

Uma Pilha (em inglês, Stack) é uma estrutura que obedece ao critério *Last In, First Out* (L.I.F.O.; Knuth, 1972, p. 236; Langsam, Augenstein e Tenenbaum, 1996, p. 78). Ou seja, o último elemento que entrou no conjunto será o primeiro elemento a sair do conjunto.

Figura 2.2 Definição de Pilha.

Uma Pilha é um conjunto ordenado de elementos. A ordem de entrada determina a posição dos elementos no conjunto. Se temos três elementos em uma Pilha (A, B e C) e se eles entraram na Pilha na sequência A, depois B e então C, sabemos que o elemento C estará no topo da Pilha ([Figura 2.3a](#)).

Na situação da [Figura 2.3a](#), o único elemento que podemos retirar é exatamente o elemento que está no topo da Pilha. Ou seja, o elemento C. Retirando o elemento C, a situação da Pilha fica sendo a da [Figura 2.3b](#). Se, em seguida, quisermos inserir o elemento D, esse elemento D passará a ser o topo da Pilha — [Figura 2.3c](#). E, finalmente, se quisermos inserir o elemento E, esse novo elemento passará a ser o elemento do topo — [Figura 2.3d](#). Se a ordem de inserção dos elementos na Pilha tivesse sido outra, a Pilha resultante também seria outra.

O mesmo raciocínio vale para uma Pilha de Cartas, por exemplo. Considere como situação inicial uma Pilha com três cartas: 8 de copas, J de paus e, no topo da Pilha, A de espadas, como mostra a [Figura 2.4a](#). Ao contrário do que fizemos na pilha de cubos da [Figura 2.3](#), agora o topo está representado na parte de baixo da Pilha de Cartas. Essa

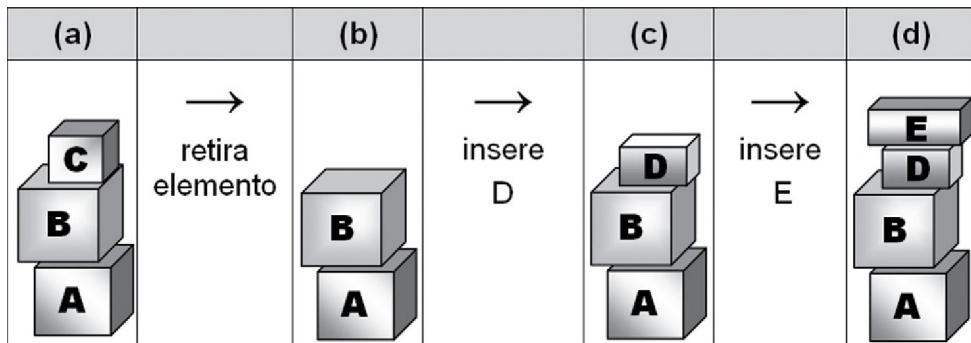


Figura 2.3 Retirando e inserindo elementos em uma Pilha.

representação mostra o valor e o naipe de todas as cartas da Pilha, como se estivéssemos empilhando uma carta sobre a outra com um baralho de papel. É a representação usual das Pilhas de Cartas em jogos do tipo *FreeCell*.

Se quisermos inserir no conjunto a carta Q de copas, essa nova carta precisará ser inserida no topo da Pilha, resultando na situação da [Figura 2.4b](#). Se, em seguida, quisermos retirar uma carta da Pilha, a única carta que podemos retirar é a carta do topo da Pilha — o próprio Q de copas, que acabou de ser inserido. A Pilha então voltará à situação da [Figura 2.4a](#), tendo apenas três elementos, com o A de espadas novamente no topo.

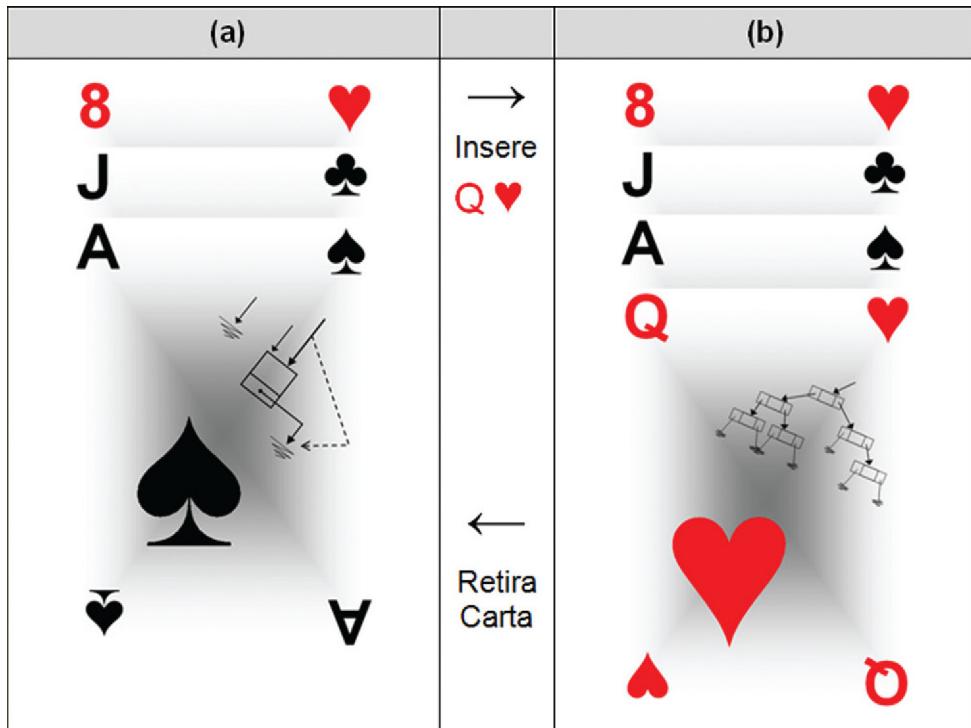


Figura 2.4 Inserindo e retirando cartas em uma Pilha.

2.2 Operações de uma Pilha

Vamos chamar a operação que insere elementos em uma Pilha de Empilha, e a operação que retira elementos de uma Pilha, de Desempilha. A essas duas Operações, vamos acrescentar outras três: operação para criar uma Pilha, operação para testar se a Pilha está vazia e operação para testar se a Pilha está cheia. Na [Figura 2.5](#) temos a especificação das operações Empilha, Desempilha, Cria, Vazia e Cheia, bem como uma descrição de seus parâmetros e funcionamento.

Operações e parâmetros	Funcionamento
Empilha (P, X, DeuCerto)	Empilha o elemento X, passado como parâmetro, na Pilha P, também passada como parâmetro. O parâmetro DeuCerto indica se a execução da operação foi bem-sucedida ou não.
Desempilha (P, X, DeuCerto)	Desempilha (retira o elemento do topo) da Pilha P passada no parâmetro, retornando o valor do elemento que foi desempilhado no parâmetro X. O parâmetro DeuCerto indica se a operação foi bem-sucedida ou não.
Vazia (P)	Verifica se a Pilha P passada como parâmetro está vazia. Uma Pilha vazia é uma Pilha sem nenhum elemento.
Cheia (P)	Verifica se a Pilha P passada como parâmetro está cheia. Uma Pilha cheia é uma Pilha em que não cabe mais nenhum elemento.
Cria (P)	Cria uma Pilha P, iniciando sua situação como vazia.

Figura 2.5 Operações de uma Pilha.

Exercício 2.1 Transfere elementos de uma Pilha para outra

Desenvolva um algoritmo para transferir todos os elementos 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 [Figura 2.5](#). Considere que as Pilhas P1 e P2 possuem elementos do tipo inteiro. Você terá que elaborar esse algoritmo sem saber como o Tipo Abstrato de Dados Pilha é efetivamente implementado. Você encontrará uma solução para este exercício na sequência do texto. Mas, antes de prosseguir a leitura, tente propor uma solução. É assim que desenvolvemos habilidade para manipular Pilhas (e outras estruturas) pelos seus Operadores.

```
TransfereElementos (parâmetros por referência P1, P2 do tipo Pilha);
/* transfere todos os elementos de P1 para P2 */
```

A lógica da solução do Exercício 2.1 é a seguinte: enquanto a Pilha P1 não estiver vazia, retiramos elementos de P1 através da operação Desempilha. Cada valor desempilhado de P1 deve ser empilhado na Pilha P2 através da operação Empilha. Repetimos essa sequência de comandos até que a Pilha P1 se torne vazia. Veja o algoritmo da [Figura 2.6](#).

```

TransfereElementos(parâmetros por referência P1, P2 do tipo Pilha) {
    /* transfere todos os elementos de P1 para P2 */

    Variável ElementoDaPilha do tipo Inteiro;
    Variável DeuCerto do tipo Boolean;

    Enquanto (Vazia(P1) == Falso) Faça {
        Desempilha(P1, ElementoDaPilha, DeuCerto);           // enquanto P1 não for vazia
        Empilha(P2, ElementoDaPilha, DeuCerto); }             // desempilha de P1
                                                               // empilha em P2
    }
}
    
```

Figura 2.6 Algoritmo para transferir elementos de uma Pilha para outra.

Note que ainda não temos a menor ideia de como essas operações do TAD Pilha (Empilha, Desempilha, Vazia, Cheia, Cria) são efetivamente implementadas. Consideramos as Pilhas como caixas-pretas e manipulamos os valores armazenados apenas através dos Operadores (ou “botões da televisão”). É assim que manipulamos estruturas de armazenamento, como Pilhas, de modo abstrato. É assim que utilizamos os Tipos Abstratos de Dados.

Proponha uma solução para os Exercícios 2.2, 2.3 e 2.4 sempre com a mesma estratégia: manipulando os valores armazenados exclusivamente através dos Operadores do TAD Pilha descritos na [Figura 2.5](#).

Exercício 2.2 Mais elementos?

Desenvolva um algoritmo para testar se uma Pilha P1 tem mais elementos do que uma Pilha P2. Considere que as Pilhas P1 e P2 já existem e são passadas como parâmetro. Considere também que as Pilhas P1 e P2 possuem elementos do tipo Inteiro. Você pode criar Pilhas auxiliares, se necessário. Você deve preservar os dados de P1 e P2. Ou seja, ao final da execução dessa operação, P1 e P2 precisam conter exatamente os mesmos elementos que continham no início da operação, e na mesma sequência. Para propor a solução, utilize os Operadores da [Figura 2.5](#).

```

Boolean MaisElementos (parâmetros por referência P1, P2 do tipo Pilha);
/* retorna Verdadeiro se a Pilha P1 tiver mais elementos do que a Pilha P2 */
    
```

Exercício 2.3 Algum elemento igual a X?

Desenvolva um algoritmo para verificar se uma Pilha P possui algum elemento igual a um valor X. P e X são passados como parâmetros. Considere que a Pilha P possui elementos do tipo Inteiro. Para propor a solução, utilize os Operadores da [Figura 2.5](#). Você pode criar Pilhas auxiliares, se necessário.

```

Boolean AlgumElementoIgualX (parâmetro por referência P do tipo Pilha, parâmetro X do tipo inteiro);
/* Verifica se a Pilha P possui algum elemento igual ao valor do parâmetro X */
    
```

Exercício 2.4 As Pilhas são iguais?

Desenvolva um algoritmo para testar se duas Pilhas P1 e P2 são iguais. Duas Pilhas são iguais se possuem os mesmos elementos, exatamente na mesma ordem. Para propor sua solução, você pode utilizar Pilhas auxiliares, se necessário. Considere que as Pilhas P1 e P2 já existem e são passadas como parâmetro. Considere que as

Pilhas P1 e P2 possuem elementos do tipo Inteiro. Para propor a solução, utilize os Operadores da [Figura 2.5](#).

Boolean iguais (parâmetros por referência P1, P2 do tipo Pilha);

```
/* retorna Verdadeiro se a Pilha P1 for igual à Pilha P2, ou seja, se P1 e P2 possuirem exatamente os mesmos elementos, na mesma ordem. Se ambas as Pilhas forem vazias, serão consideradas iguais */
```

No final deste capítulo você encontrará respostas ou sugestões para alguns dos exercícios. Mas, para desenvolver as habilidades pretendidas, proponha suas próprias soluções antes de consultar as respostas sugeridas.

2.3 Alocação de memória para conjuntos de elementos

Alocar memória, em essência, significa reservar uma porção da memória do computador para um uso específico. Alocar memória para um conjunto de elementos, como uma Pilha, envolve reservar espaço de memória para armazenar cada um dos elementos do conjunto.

Alocação Sequencial

Na Alocação Sequencial de Memória para um conjunto de elementos ([Knuth, 1972](#), p. 240; [Pereira, 1996](#), p. 12), todos os elementos do conjunto serão armazenados juntos, em posições adjacentes de memória. Ou seja, os elementos ficam “em sequência”, um ao lado do outro na memória do computador.

Definição: Alocação Sequencial de Memória para um conjunto de elementos

- Os elementos ficam, necessariamente, em posições de memória adjacentes ou “em sequência”.

Figura 2.7 Definição de Alocação Sequencial de Memória para um conjunto de elementos.

A [Figura 2.8](#) mostra uma Pilha com quatro elementos: o elemento A no topo da Pilha e, em seguida, os elementos D, C e B. Na Alocação Sequencial de Memória, a ordem desses quatro elementos é refletida nas posições de memória em que os elementos são armazenados.

Alocação Estática

Na Alocação Estática de Memória para um conjunto de elementos, estabelecemos o tamanho máximo do conjunto previamente, ao elaborar o programa, e reservamos memória para todos os seus elementos. Esse espaço de memória permanecerá reservado durante toda a execução do programa, mesmo que não esteja sendo totalmente utilizado, isto é, mesmo que em determinado momento a quantidade de elementos no conjunto seja menor que o que seu tamanho máximo. Como o tamanho do espaço de memória é definido previamente, ele não poderá crescer ou diminuir ao longo da execução do programa.

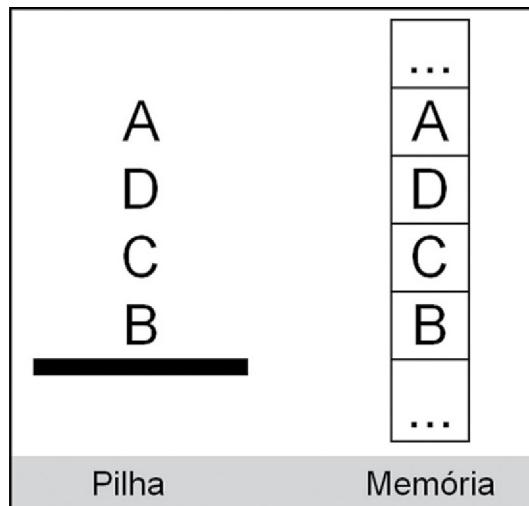


Figura 2.8 Alocação Sequencial — armazenamento em posições de memória adjacentes.

Definição: Alocação Estática de Memória para um conjunto de elementos

- O espaço de memória para todos os elementos que poderão fazer parte do conjunto é dimensionado previamente.
- Esse espaço de memória permanecerá reservado durante toda a execução do programa, mesmo se não estiver sendo efetivamente utilizado.

Figura 2.9 Definição de Alocação Estática de Memória para um conjunto de elementos.

2.5 Implementando uma Pilha com Alocação Sequencial e Estática

Para implementar uma Pilha combinando os conceitos de Alocação Sequencial e Alocação Estática de Memória, ao elaborar o programa precisamos definir o tamanho do espaço de memória a ser reservado e possibilitar que os elementos da Pilha possam ser armazenados em posições adjacentes de memória. Podemos fazer isso através de uma estrutura do tipo vetor.

Na [Figura 2.10](#) temos o esquema de uma Pilha implementada através de um vetor denominado Elementos, que armazena os elementos da Pilha, e de uma variável denominada Topo, que indica o topo da Pilha.

Podemos definir o tipo Pilha como um Registro (uma *Struct* na linguagem C) contendo dois campos: Elementos e Topo. Topo do tipo Inteiro e Elementos do tipo vetor de 1 até Tamanho (constante que indica o tamanho da Pilha) ou, ainda, de zero até Tamanho – 1, conforme o padrão da linguagem C. Os elementos da Pilha podem ser do tipo Char, do tipo Inteiro, do tipo Carta-do-Baralho ou de outro tipo qualquer, conforme necessário.

Para implementar um Tipo Abstrato de Dados (TAD) precisamos de uma estrutura de armazenamento e de Operadores que possam ser aplicados sobre os Dados armazenados. Vamos implementar o TAD Pilha com a estrutura de armazenamento definida na [Figura 2.10](#) e com os Operadores definidos na [Figura 2.5](#): Empilha, Desempilha, Cria, Vazia e Cheia.

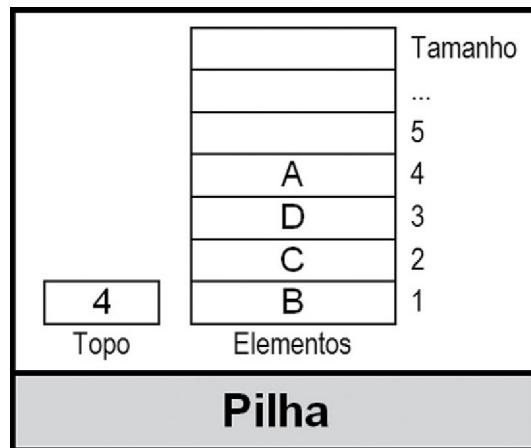


Figura 2.10 Esquema da implementação de uma Pilha com Alocação Sequencial e Estática de Memória.

Exercício 2.5 Operação Empilha

Conforme especificado na [Figura 2.5](#), a operação Empilha recebe como parâmetros a Pilha e o valor do elemento que queremos empilhar. O elemento só não será inserido se a Pilha já estiver cheia. Você encontrará uma possível solução para este exercício na sequência do texto. Mas tente propor uma solução antes de prosseguir a leitura.

Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);
/* Empilha o elemento X na Pilha P. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não. */

A [Figura 2.11](#) ilustra o funcionamento da operação Empilha. A partir da situação da [Figura 2.11 a](#), um elemento de valor 'E' é inserido, resultando na situação da [Figura 2.11 b](#).

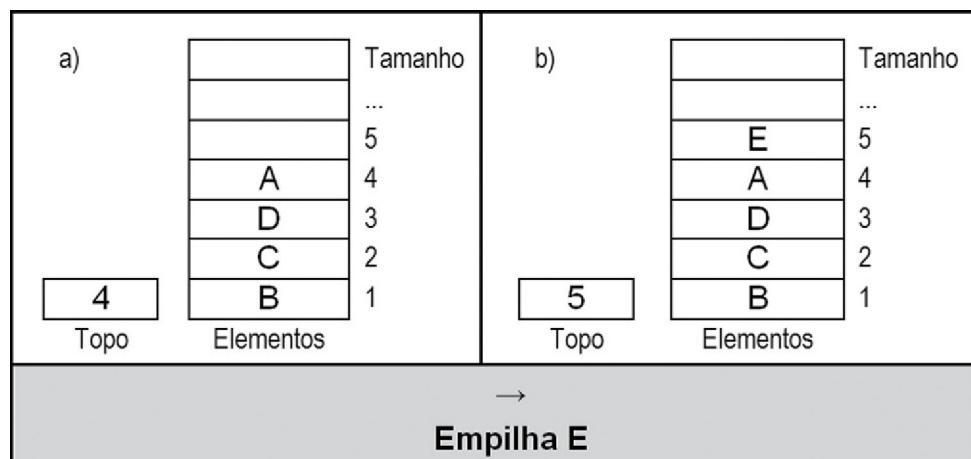


Figura 2.11 Operação Empilha.

Note que o valor da variável Topo foi incrementado, e o elemento 'E' foi inserido no vetor Elementos na posição indicada pelo valor atualizado da variável Topo.

A [Figura 2.12](#) apresenta um algoritmo conceitual para a Operação Empilha. Se a Pilha P estiver cheia, sinalizamos através do parâmetro DeuCerto que o elemento X não foi inserido. Mas, se a Pilha P não estiver cheia, X será inserido da seguinte forma: incrementaremos o Topo da Pilha P ($P.\text{Topo} = P.\text{Topo} + 1$) e armazenamos X no vetor Elementos, na posição indicada pelo Topo da Pilha ($P.\text{Elementos}[P.\text{Topo}] = X$). O elemento X passará a ser o elemento do Topo da Pilha.

```

Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro
por referência DeuCerto do tipo Boolean) {
    /* Empilha o elemento X, passado como parâmetro, na Pilha P também passada como
       parâmetro. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não.*/
    Se (Cheia(P) == Verdadeiro)           // se a Pilha P estiver cheia...
        Então DeuCerto = Falso            // ... não podemos empilhar
    Senão { P.Topo = P.Topo + 1;          // incrementa o Topo da Pilha P
            P.Elementos[ P.Topo ] = X;    // armazena o valor de X no Topo da Pilha
            DeuCerto = Verdadeiro; }      // a operação deu certo
}
    
```

Figura 2.12 Algoritmo conceitual — Empilha.

Exercício 2.6 Operação Desempilha

Conforme especificado na [Figura 2.5](#), a operação Desempilha recebe como parâmetro a Pilha da qual queremos retirar um elemento. Caso a Pilha não estiver vazia, a operação retorna o valor do elemento retirado. A ação da operação Desempilha, na situação da [Figura 2.13a](#), resultaria na situação da [Figura 2.13b](#).

Desempilha(parâmetro por referência P do tipo Pilha, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);
 /* Caso a Pilha P não estiver vazia, retira o elemento do Topo e retorna seu valor no parâmetro X. Se a Pilha P estiver vazia,
 o parâmetro DeuCerto deve retornar Falso */

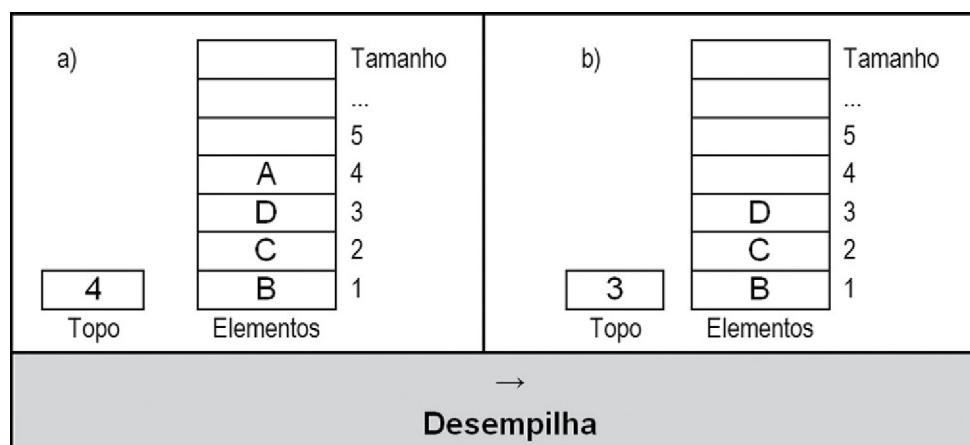


Figura 2.13 Operação Desempilha.

Exercício 2.7 Operação Cria

Conforme especificado na [Figura 2.5](#), a operação Cria recebe como parâmetro a Pilha que deverá ser criada. Criar a Pilha significa inicializar os valores de modo a indicar que a Pilha está vazia, ou seja, sem nenhum elemento.

Cria (parâmetro por referência P do tipo Pilha);

/* Cria a Pilha P, inicializando-a como vazia – sem nenhum elemento. */

Exercício 2.8 Operação Vazia

Conforme especificado na [Figura 2.5](#), a operação Vazia testa se a Pilha passada como parâmetro está vazia (sem elementos), retornando o valor Verdadeiro (Pilha vazia) ou Falso (Pilha não vazia).

Boolean Vazia (parâmetro por referência P do tipo Pilha);

/* Retorna Verdadeiro se a Pilha P estiver vazia – sem nenhum elemento; Falso caso contrário. */

Exercício 2.9 Operação Cheia

Conforme especificado na [Figura 2.5](#), a operação Cheia testa se a Pilha passada como parâmetro está cheia. A Pilha estará cheia se na estrutura de armazenamento não couber mais nenhum elemento.

Boolean Cheia (parâmetro por referência P do tipo Pilha);

/* Retorna Verdadeiro se a Pilha P estiver cheia, ou seja, se na estrutura de armazenamento não couber mais nenhum elemento; Retorna Falso caso a Pilha P não estiver cheia. */

Conforme mostra a [Figura 2.14](#), na operação Cria inicializamos o Topo da Pilha com o valor zero, pois os elementos começam a ser armazenados no vetor Elementos a partir da posição 1. Assim, com o valor de Topo sendo zero, indicamos que a Pilha está vazia.

 Pilha Vazia	 Pilha Cheia
------------------------	------------------------

Figura 2.14 Pilha Vazia e Pilha Cheia.



Note que, em vez de começar em 1, se o vetor começar em zero (esse é o padrão na linguagem C) a situação de Pilha vazia será caracterizada quando o valor do Topo for -1.

A operação Vazia simplesmente testará se o Topo da Pilha está apontando para o valor zero. Se o valor do Topo for zero, a Pilha estará vazia. Se o valor do Topo não estiver em zero, a Pilha não estará vazia. Se, em vez de começar em 1, o vetor começar em zero (padrão na linguagem C) a Pilha estará vazia quando o valor do Topo for -1.

Analogamente, na operação Cheia verificamos se o valor do Topo é igual à constante Tamanho, que indica o Tamanho da Pilha. Se, em vez de começar em 1, o vetor começar em zero (padrão na linguagem C) a situação de Pilha cheia será caracterizada quando o valor do Topo for igual a Tamanho -1.

No final deste capítulo são apresentadas soluções ou sugestões para alguns dos exercícios. Mas, para desenvolver as habilidades pretendidas, proponha suas próprias soluções antes de consultar as respostas e sugestões.

2.6 Abrir ou não a televisão?

Suponha que queiramos desenvolver uma operação para consultar o valor do elemento do topo de uma Pilha. Não queremos retirar o elemento do topo; queremos apenas consultar o seu valor, sem retirá-lo da Pilha. Pense em duas estratégias diferentes para implementar essa operação.

Exercício 2.10 Duas estratégias para elemento do topo

Desenvolva uma operação que retorne o valor do elemento do topo de uma Pilha sem retirar o elemento da Pilha. Proponha duas soluções diferentes para essa operação e compare as soluções, apontando suas vantagens e desvantagens. Proponha e compare as soluções antes de prosseguir a leitura.

Char ElementoDoTopo (parâmetro por referência P do tipo Pilha, parâmetro por referência DeuCerto do tipo Boolean);
/* Retorna o valor do elemento do Topo da Pilha P, caso a Pilha não estiver vazia. Caso a Pilha estiver vazia, DeuCerto retorna o valor Falso */

Podemos implementar a operação Elemento do Topo basicamente de duas maneiras. A primeira maneira é dependente da estrutura de armazenamento, e a segunda maneira é independente da estrutura de armazenamento. Na [Figura 2.15](#) apresentamos duas soluções para a operação Elemento do Topo. A diferença entre as soluções está destacada em negrito. O que muda é o modo de pegar a informação do Topo da Pilha.

Na primeira solução, pegamos o valor do elemento do Topo simplesmente atribuindo a X o valor de P.Elementos[P.Topo]. Nessa primeira solução, fica evidente que a estrutura de armazenamento utilizada é um vetor. Ou seja, é uma solução dependente da estrutura de armazenamento; só funcionará na estrutura de armazenamento em questão — um vetor. Na segunda implementação, pegamos o valor do elemento do Topo através da operação Desempilha. A operação Desempilha retorna o valor do elemento do Topo na variável X, mas retira o elemento da Pilha. Por isso, temos que recolocar o elemento na Pilha através da operação Empilha. Nessa segunda solução, não fica evidente a estrutura de armazenamento utilizada. É, portanto, uma solução independente da estrutura de armazenamento.

```

Char ElementoDoTopo (parâmetro por referência P do tipo Pilha, parâmetro por referência
DeuCerto do tipo Boolean) {
    /* Retorna o valor do elemento do Topo da Pilha P, caso a Pilha não esteja vazia. Caso a
    Pilha esteja vazia, DeuCerto retorna Falso. */
    Variável X do tipo Char; /* X armazenará o valor do elemento do Topo */
    /* primeira solução: abrindo a televisão com uma chave de fenda para aumentar o
    volume */
    Se (Vazia (P)==Verdadeiro)
        Então DeuCerto = Falso
    Senão { X = P.Elementos[ P.Topo ]; /* pega o valor do elemento do Topo */
        DeuCerto = Verdadeiro;
        Retorne X; };

    /* segunda solução: aumentando o volume pelo botão de volume */
    Se (Vazia (P)==Verdadeiro)
        Então DeuCerto = Falso
    Senão { Desempilha ( P, X, DeuCerto); /* pega o valor do elemento do Topo */
        Empilha ( P, X, DeuCerto);
        DeuCerto = Verdadeiro;
        Retorne X; }
}

```

Figura 2.15 Soluções para a operação Elemento do Topo.

Quais as vantagens de uma solução e de outra? A primeira solução funcionará exclusivamente nessa implementação que fizemos da Pilha, com Alocação Sequencial e Estática de Memória. Nos capítulos a seguir implementaremos uma Pilha com outras técnicas, e essa primeira solução não continuará funcionando; terá que ser reescrita. A segunda solução para a operação Elemento do Topo é independente da estrutura de armazenamento, pois manipula a Pilha exclusivamente através dos operadores Vazia, Empilha e Desempilha. Assim, essa segunda implementação continuará funcionando mesmo se adotarmos uma nova estrutura de armazenamento para a Pilha, como faremos nos próximos capítulos. Por ser independente da estrutura de armazenamento, a segunda solução proporciona maior portabilidade de código, conceito que estudamos no Capítulo 1.

Você pode estar pensando: “Para que complicar, se podemos simplesmente consultar o valor do elemento do Topo com um comando `X = P.Elementos[P.Topo]`”? Sim, é uma solução eficiente. Mas é dependente da implementação e não proporciona portabilidade de código.

Pense também nas soluções dos Exercícios 2.1 a 2.4. O Exercício 2.2, por exemplo, verifica se uma Pilha P1 possui mais elementos do que uma Pilha P2. Podemos ter uma solução que simplesmente compara os valores do Topo das Pilhas P1 e P2, ou seja, se P1.Topo for maior do que P2.Topo, a Pilha P1 tem mais elementos do que a Pilha P2. Essa seria uma solução dependente da estrutura de armazenamento; dependente da implementação. O código dessa solução não seria portável. Seria como abrir uma televisão com uma chave de fenda para aumentar o volume.

Uma segunda solução poderia desempilhar todos os elementos de P1 e de P2, colocando os elementos em Pilhas auxiliares e contando os elementos de cada Pilha. Depois os elementos poderiam ser devolvidos às Pilhas originais. Essa segunda solução seria independente da estrutura de armazenamento; independente da implementação. Uma solução portável. Conforme estudamos no Capítulo 1, portabilidade e reusabilidade

de código são características extremamente desejáveis e, no longo prazo, tornam as soluções mais baratas.

Chave de fenda ou botão da televisão? Qual dessas soluções você considera mais interessante?

Operações Primitivas e Não Primitivas

Podemos dizer que, em uma Pilha, as operações Empilha, Desempilha, Cria, Vazia e Cheia são *Operações Primitivas*, pois só podem ser implementadas através de uma solução dependente da estrutura de armazenamento. Já as operações que desenvolvemos nos Exercícios 2.1 a 2.4 e 2.10 (transferir elementos de uma Pilha a outra, verificar se uma Pilha tem mais elementos que outra, verificar se uma Pilha possui um elemento com valor X, verificar se duas Pilhas são iguais e pegar o valor do Elemento do Topo de uma Pilha) são *Operações Não Primitivas*, pois podem ser implementadas a partir de chamadas a Operações Primitivas do TAD Pilha, resultando em uma implementação independente da estrutura de armazenamento.

- Definição: Operações Primitivas e Operações Não Primitivas**
- **Operações Primitivas** de um Tipo Abstrato de Dados são aquelas que só podem ser implementadas através de uma solução dependente da estrutura de armazenamento.
 - **Operações Não Primitivas** de um Tipo Abstrato de Dados são aquelas que podem ser implementadas através de chamadas a Operações Primitivas, possibilitando uma implementação independente da estrutura de armazenamento.

Figura 2.16 Definição de Operações Primitivas e Não Primitivas.

Visando proporcionar portabilidade e reusabilidade de código, menor custo de desenvolvimento e manutenção, a melhor forma de implementar uma Operação Não Primitiva é de modo abstrato, manipulando o TAD em questão exclusivamente pelas Operações Primitivas ou, ainda, pelos “botões da televisão”.

2.7 Projeto FreeCell: Pilha Burra ou Pilha Inteligente?

No *FreeCell* temos oito Pilhas Intermediárias que possuem algumas restrições com relação ao valor dos elementos que serão empilhados. Nessas Pilhas Intermediárias só é possível empilhar cartas em ordem decrescente e em cores alternadas — pretas sobre vermelhas e vermelhas sobre pretas.

Temos também no *FreeCell* um segundo tipo de Pilha, as Pilhas Definitivas, que também possuem suas regras com relação ao valor dos elementos que serão empilhados. Nessas quatro Pilhas Definitivas só é possível empilhar cartas de um mesmo naipe e em ordem crescente.

A característica básica de inserir elementos sempre no Topo é válida tanto para as Pilhas Intermediárias quanto para as Pilhas Definitivas. Mas as regras com relação ao valor dos elementos que serão empilhados são diferentes para Pilhas Intermediárias e Pilhas Definitivas.

A Pilha que definimos e implementamos nas seções anteriores é uma “pilha burra”, pois ela não faz qualquer verificação quanto ao valor do elemento que está sendo empilhado. Como você acha que podemos implementar a “inteligência” das Pilhas Intermediárias e das Pilhas Definitivas? Por “inteligência” queremos dizer a verificação dos valores que estão sendo empilhados, para saber se podem ser inseridos em uma Pilha Intermediária ou em uma Pilha Definitiva, de acordo com as regras de funcionamento do *FreeCell*.

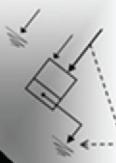
K Q J 10 9 8 7	♣ ♥ ♣ ♦ ♠ ♥ ♠	A 2 3 4 5 6 7	♠ ♠ ♠ ♠ ♠ ♠ ♠
 Pilha Intermediária	 Pilha Definitiva		

Figura 2.17 Pilha Intermediária: ordem decrescente e em cores alternadas. Pilha Definitiva: ordem crescente e cartas do mesmo naipe.

A [Figura 2.18](#) mostra três alternativas para implementar a “inteligência” das Pilhas. Uma primeira alternativa (solução A) seria utilizar uma Pilha Burra e realizar todas as verificações na aplicação. Ou seja, “fora” do TAD Pilha é que iríamos verificar se uma carta pode ou não ser empilhada. Nessa solução A, tanto as Pilhas Intermediárias quanto as Pilhas Definitivas poderiam ser implementadas através do mesmo TAD Pilha Burra.

Uma segunda alternativa (solução B) seria desenvolver dois TADs diferentes, Pilha Intermediária e Pilha Definitiva, cada qual incorporando sua própria inteligência. Nessa solução, a aplicação em si não faria qualquer verificação de valores. Toda a verificação seria feita “dentro” das Pilhas. Se, pelas regras do *FreeCell*, uma carta não puder ser empilhada, a

própria Pilha vai identificar isso e comunicar à aplicação. Note que, na solução B, temos duas implementações distintas de Pilha: a Pilha Intermediária e a Pilha Definitiva.

A solução C é uma variação da solução B. A diferença é que na solução C a Pilha Intermediária e a Pilha Definitiva são implementadas com base em uma Pilha Burra. Em uma linguagem orientada a objeto, como C++, as Pilhas Intermediárias e as Pilhas Definitivas podem “herdar” características da Pilha Burra e agregar suas características próprias e diferenciadas — a “inteligência”. Mesmo sem utilizar o conceito de herança da orientação a objetos, as operações das Pilhas Intermediária e Definitiva podem ser implementadas através de chamadas às operações da Pilha Burra.

As três soluções incluem um módulo que implementa a interface do sistema com o usuário. Incluir um módulo independente para a interface aumenta a portabilidade da solução e facilita ajustes ou mesmo uma troca radical de interface ou de plataforma, sem grandes mudanças nos demais módulos.

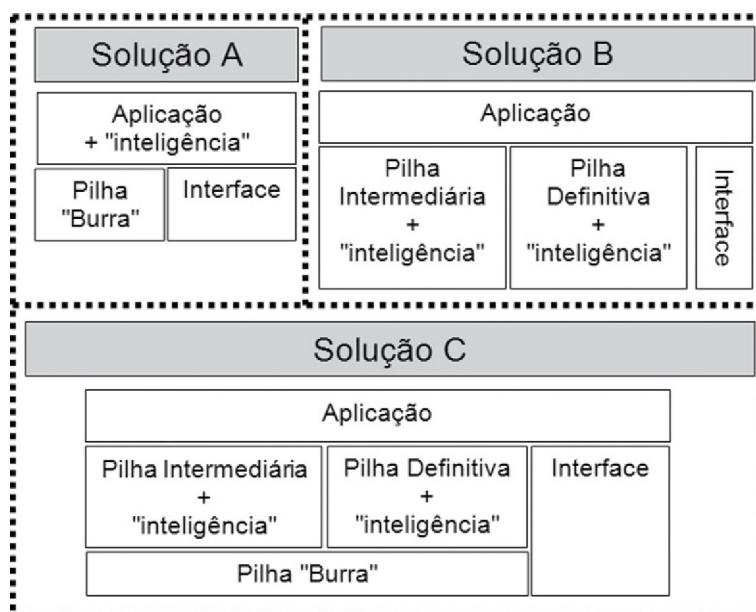


Figura 2.18 Projeto *FreeCell* — soluções alternativas quanto à arquitetura do software.

Qual dessas três soluções — A, B ou C — você considera mais adequada ao seu projeto?

Exercício 2.11 Avançar o Projeto FreeCell — propor uma arquitetura de software

Pilha Burra? Pilha Inteligente? Proponha uma arquitetura de software para o seu Projeto *FreeCell* identificando os principais módulos do sistema e o relacionamento entre eles. Tome como ponto de partida as soluções alternativas propostas na [Figura 2.18](#). Escolha a alternativa que for mais adequada ao seu projeto ou adapte uma das alternativas, conforme necessário. Sugestão para uso acadêmico: desenvolva o Projeto *FreeCell* em grupo. Promova uma discussão, defina a arquitetura do software e divida o trabalho entre os componentes do grupo.

Seja qual for a arquitetura de software que adotar para o seu Projeto *FreeCell*, visando agregar portabilidade e reusabilidade, observe as seguintes recomendações na implementação:

- Para manipular as Pilhas de Cartas, projete e utilize um TAD Pilha (seja uma Pilha Burra, seja uma Pilha Inteligente).
- A aplicação em si e o TAD Pilha devem estar em unidades de software independentes e em arquivos separados; utilize um arquivo exclusivamente para a implementação do TAD Pilha.
- A aplicação (e outros módulos) deve manipular o TAD Pilha exclusivamente através dos seus Operadores Primitivos: Empilha, Desempilha, Vazia, Cria e Cheia. Aumente o volume da televisão apenas pelo botão de volume.
- Inclua no código do TAD Pilha exclusivamente ações pertinentes ao armazenamento e recuperação das informações sobre as Pilhas de Cartas. Faça o possível para não incluir no TAD Pilha ações relativas à interface ou a qualquer outro aspecto que não seja o armazenamento e a recuperação de informações sobre as Pilhas de Cartas.

Exercício 2.12 Implementar uma Pilha Burra

Implemente um TAD Pilha Burra (Pilha sem qualquer restrição aos valores que estão sendo empilhados) em uma linguagem de programação como C ou C++, sem utilizar recursos da orientação a objetos. Defina um tipo estruturado e os Operadores. Os elementos da Pilha devem ser do tipo Inteiro. Implemente a Pilha como uma unidade independente. Em um arquivo separado, faça o programa principal bem simples, para testar o funcionamento da Pilha.

Exercício 2.13 Implementar uma Pilha Burra como uma classe

Implemente um TAD Pilha Burra (Pilha sem qualquer restrição aos valores que estão sendo empilhados) em uma linguagem de programação orientada a objetos, como C++. Implemente a Pilha como uma classe. Os elementos da Pilha devem ser do tipo Inteiro. Implemente a Pilha como uma unidade independente. Em um arquivo separado, faça o programa principal bem simples, para testar o funcionamento da Pilha.

Exercício 2.14 Avançar o Projeto FreeCell — defina e implemente a Carta do Baralho

Defina e implemente a Carta do Baralho, como uma classe ou como um tipo estruturado. Não se preocupe, neste momento, com aspectos da interface. Preocupe-se em definir uma carta com naipe (ouro, paus, copas ou espadas) e valor (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K).

Exercício 2.15 Avançar o Projeto FreeCell — implemente uma Pilha Intermediária Inteligente

Defina e implemente uma Pilha Intermediária “Inteligente”, ou seja, com restrições quanto às cartas que serão empilhadas, conforme as regras do *FreeCell* convencional ou conforme as regras do seu Projeto *FreeCell*. Os elementos da Pilha devem ser do tipo Carta, conforme definido no Exercício 2.14. Para implementar a Pilha Intermediária Inteligente, utilize de algum modo a Pilha Burra implementada no Exercício 2.12 ou no

Exercício 2.13. Faça um programa para testar o funcionamento da Pilha Intermediária Inteligente sem se preocupar com a qualidade da interface.

Exercício 2.16 Avançar o Projeto FreeCell — implemente uma Pilha Definitiva Inteligente

Defina e implemente uma Pilha Definitiva “Inteligente”, ou seja, com restrições quanto às cartas que serão empilhadas, conforme as regras do *FreeCell* convencional. Os elementos da Pilha devem ser do tipo Carta, conforme definido no Exercício 2.14. Para implementar a Pilha Definitiva Inteligente, utilize de algum modo a Pilha Burra implementada no Exercício 2.12 ou no Exercício 2.13. Faça um programa para testar o funcionamento da Pilha Definitiva Inteligente sem se preocupar com a qualidade da interface.

Exercício 2.17 Avançar o Projeto FreeCell — defina as regras, escolha um nome e inicie o desenvolvimento do seu FreeCell

Defina as regras do seu *FreeCell* alterando as regras do *FreeCell* convencional em algum aspecto. Dê personalidade própria ao seu *FreeCell* e escolha para o seu jogo um nome que reflita suas características mais marcantes. Escreva as regras do seu *FreeCell* e coloque em um arquivo-texto, para que possa ser utilizado na documentação do jogo ou em uma opção do jogo que ensine como jogar. Defina a arquitetura do software (Exercício 2.11), adapte as Pilhas desenvolvidas (Exercícios 2.12 a 2.16) e inicie o desenvolvimento do seu jogo. Sugestão para uso acadêmico: desenvolva o Projeto *FreeCell* em grupo. Tome as principais decisões em conjunto e divida o trabalho entre os componentes do grupo, cada qual ficando responsável por parte das atividades.

Exercício 2.18 Avançar o Projeto FreeCell — inicie o projeto da interface



Em paralelo ao desenvolvimento da lógica do *FreeCell*, escolha e estude uma biblioteca gráfica para ajudar a construir uma interface visual e intuitiva para o seu jogo. Você pode começar estudando o **Tutorial de Programação Gráfica** disponível nos Materiais Complementares de *Estruturas de dados com jogos*, mas fique livre para estudar outros materiais ou adotar outras ferramentas gráficas no seu *FreeCell*. Projete a interface do modo mais independente possível dos demais módulos. Se estiver trabalhando em grupo, um dos membros do grupo pode se dedicar a estudar mais intensamente os aspectos referentes à interface e depois ajudar os demais componentes no aprendizado. Você pode optar também por implementar primeiramente uma interface bem simples, textual, para testar bem os demais componentes do jogo e depois substituir por uma interface gráfica. Em algum momento, você precisará mesmo ganhar experiência no desenvolvimento de software utilizando bibliotecas gráficas. Por que não agora?

Comece a desenvolver o seu *FreeCell* agora!

Faça um jogo legal! Faça um jogo que tenha a sua cara! Faça seu jogo ficar atrativo, distribua para os amigos, disponibilize publicamente, faça seu jogo bombar! Aprender a programar pode ser divertido!

Consulte no Banco de Jogos

Adaptações do FreeCell

www.elsevier.com.br/edcomjogos



Consulte nos Materiais Complementares

Tutorial de Programação Gráfica



<http://www.elsevier.com.br/edcomjogos>

Exercícios de fixação

Exercício 2.19 O que é uma Pilha?

Exercício 2.20 Para que serve uma Pilha? Em que tipo de situações uma Pilha pode ser utilizada?

Exercício 2.21 O que significa Alocação Sequencial de Memória para um conjunto de elementos?

Exercício 2.22 O que significa Alocação Estática de Memória para um conjunto de elementos?

Exercício 2.23 Faça o esquema da implementação de uma Pilha com Alocação Sequencial e Estática de Memória, e descreva o seu funcionamento.

Exercício 2.24 Desafio das Torres de Hanoi. O Desafio das Torres de Hanoi é um jogo clássico, inspirado em uma antiga lenda. O jogo conta com três hastes verticais. Uma das hastes contém três discos: embaixo, um primeiro disco de diâmetro maior; em cima desse primeiro disco, há um segundo disco de diâmetro um pouco menor; e em cima desse segundo disco, há um terceiro disco de diâmetro ainda menor, como ilustra a [Figura 2.19](#). O desafio do jogo é passar todos os três discos, nessa mesma sequência, para uma das outras hastes verticais. É possível utilizar as três hastes para auxiliar a movimentação dos discos. Mas existem algumas restrições na movimentação: só é possível movimentar um disco de cada vez; só é possível retirar o disco do topo de uma pilha de discos; só é possível inserir discos no topo de uma pilha de discos; em nenhum momento você pode colocar um disco em cima de outro disco com diâmetro menor. Se não conhece bem o Desafio das Torres de Hanoi, jogue algumas partidas em uma versão on-line (por exemplo, no Games On e no UOL Jogos — links 1 e 2) ou assista a algum vídeo (por exemplo, nos links 3 e 4) para se familiarizar com o jogo. É possível aumentar a quantidade de discos e com isso aumentar o grau de dificuldade. Como você poderia usar uma ou mais Pilhas para implementar um jogo como o Desafio das Torres de Hanoi?

Exercício 2.25 Identificar outras aplicações de Pilhas. Identifique e liste aplicações de Pilhas. Identifique e anote alguns jogos que podem ser implementados com o uso de uma Pilha e identifique também outras aplicações de fora do mundo dos games. Sugestão de uso acadêmico: faça uma discussão em grupo. Ao final, cada grupo apresenta

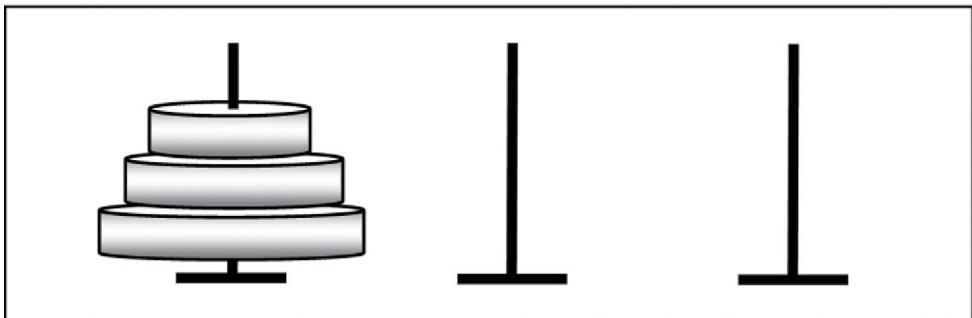


Figura 2.19 Ilustração do Desafio das Torres de Hanoi.

a todos os estudantes as aplicações que identificou. Pode ser uma boa fonte de ideias para novos jogos.

Soluções e sugestões para alguns exercícios

Exercício 2.2 Mais elementos?

```
Boolean MaisElementos (parâmetros por referência P1, P2 do tipo Pilha) {
    /* retorna Verdadeiro se a Pilha P1 tiver mais elementos do que a Pilha P2 */

    /* A questão deve ser resolvida exclusivamente através dos operadores da Pilha - Empilha,
    Desempilha, Vazia, Cria, e Cheia. A lógica é retirar um a um os elementos de P1 e colocar em uma
    Pilha auxiliar PAux1. A cada elemento retirado, incrementamos um contador. Depois retornamos
    todos os elementos de PAux1 para P1. Fazemos o mesmo para P2. Finalmente, verificamos se o
    número de elementos em P1 é maior do que o número de elementos em P2 */

    Variável DeuCerto do tipo Boolean;
    Variável ElementoDaPilha do tipo Inteiro;
    Variáveis ElementosEmP1, ElementosEmP2 do tipo Inteiro;
    Variáveis PAux1, PAux2 do tipo Pilha;

    /* contando o número de elementos em P1: retira todos de P1 e coloca em PAux1 */
    ElementosEmP1 = 0;
    Cria( PAux1 );
    Enquanto (Vazia(P1) == Falso) Faça {
        Desempilha(P1, ElementoDaPilha, DeuCerto); // retira elemento de P1...
        Empilha(PAux1, ElementoDaPilha, DeuCerto); // ... e insere em PAux1
        ElementosEmP1 = ElementosEmP1 + 1; }; // incrementa o contador

    /* retornando todos os elementos de PAux1 para P1 */
    Enquanto (Vazia(PAux1) == Falso) Faça {
        Desempilha(PAux1, ElementoDaPilha, DeuCerto); // retira elemento de PAux1
        Empilha(P1, ElementoDaPilha, DeuCerto); }; // ... e retorna para P1

    /* contando o número de elementos de P2: retira todos de P2 e coloca em PAux2 */
    ElementosEmP2 = 0;
    Cria( PAux2 );
    Enquanto (Vazia(P2) == Falso) Faça {
        Desempilha(P2, ElementoDaPilha, DeuCerto); // retira elemento de P2...
        Empilha(PAux2, ElementoDaPilha, DeuCerto); // ... e insere em PAux1
        ElementosEmP2 = ElementosEmP2 + 1; }; // incrementa o contador

    /* retornando todos os elementos de PAux2 para P2 */
    Enquanto (Vazia(PAux2) == Falso) Faça {
        Desempilha(PAux2, ElementoDaPilha, DeuCerto); // retira elemento de PAux2
        Empilha(P2, ElementoDaPilha, DeuCerto); }; // ... e retorna para P2

    /* o resultado... */
    Se (ElementosEmP1 > ElementosEmP2) // compara os contadores...
    Então Retorne Verdadeiro;
    Senão Retorne Falso;
} /* fim do procedimento */
```



Exercício 2.4 As Pilhas são iguais?

```
Boolean Iguais (parâmetros por referência P1, P2 do tipo Pilha) {  
    /* retorna o valor Verdadeiro se a Pilha P1 for igual à Pilha P2, ou seja, se possuem exatamente os  
    mesmos elementos, na mesma ordem. Se ambas as Pilhas forem vazias, serão consideradas  
    iguais */
```

```
/* Resolução: A questão deve ser resolvida exclusivamente através dos operadores da Pilha -  
Empilha, Desempilha, Vazia, Cria, e Cheia */
```

```
Variáveis DeuCerto1, DeuCerto2, PorEnquantoSãoIguais do tipo Boolean;  
Variáveis ElementoDaPilha1, ElementoDaPilha2 do tipo Inteiro;  
Variáveis PAux1, PAux2 do tipo Pilha;
```

```
PorEnquantoSãoIguais = Verdadeiro; // a princípio, consideramos que P1 e P2 são iguais.
```

```
/* tentando encontrar situações em que as Pilhas P1 e P2 não são iguais... */
```

```
/* enquanto houver elemento em P1 E também em P2... retirar 1 elemento de P1 e 1 elemento de  
P2 */
```

```
Enquanto ((Vazia(P1) == Falso) AND Vazia(P2) == Falso) {  
    Desempilha(P1, ElementoDaPilha1, DeuCerto1);  
    Desempilha(P2, ElementoDaPilha2, DeuCerto2);  
    Se (DeuCerto1 == Verdadeiro)  
        Então Empilha(PAux1, ElementoDaPilha1, DeuCerto1);  
    Se (DeuCerto2 == Verdadeiro)  
        Então Empilha(PAux2, ElementoDaPilha2, DeuCerto2);
```

```
/* Para buscar situações em que as Pilhas P1 e P2 não são iguais... em cada "loop" desse  
comando de repetição, quatro casos podem ocorrer:
```

	DeuCerto1	DeuCerto2
Caso 1- Falso	Falso	Falso
Caso 2- Falso	Falso	Verdadeiro
Caso 3- Verdadeiro	Verdadeiro	Falso
Caso 4- Verdadeiro	Verdadeiro	Verdadeiro

No Caso 1, concluímos que as Pilhas continuam sendo iguais. Nos casos 2 e 3 concluímos que as Pilhas não são iguais (pois o número de elementos é diferente). No caso 4 se o elemento retirado de P1 for diferente do elemento retirado de P2, concluímos que as Pilhas P1 e P2 não são iguais...
*/

```
Se ((DeuCerto1 == Verdadeiro) AND (DeuCerto2 == Falso))  
    Então PorEnquantoSãoIguais = Falso; // caso 2  
Se ((DeuCerto1 == Falso) AND (DeuCerto2 == Verdadeiro))  
    Então PorEnquantoSãoIguais = Falso; // caso 3  
Se ((DeuCerto1 == Verdadeiro) AND (DeuCerto2 == Verdadeiro)) // caso 4  
    Então Se (ElementoDaPilha1 != ElementoDaPilha2)  
        Então PorEnquantoSãoIguais = Falso;  
    } // fim do comando Enquanto */
```

```
/* voltando os elementos de PAux1 para P1 */
```

```
Enquanto (Vazia(PAux1) == Falso) Faça {  
    Desempilha(PAux1, ElementoDaPilha1, DeuCerto1);  
    Empilha(P1, ElementoDaPilha1, DeuCerto1);
```

```
/* voltando os elementos de PAux2 para P2 */
```

```
Enquanto (Vazia(PAux2) == Falso) Faça {  
    Desempilha(PAux2, ElementoDaPilha2, DeuCerto2);  
    Empilha(P2, ElementoDaPilha2, DeuCerto2);
```

```
/* dando o resultado...*/
```

```
Se (PorEnquantoSãoIguais == Verdadeiro)
```

```
Então Retorne Verdadeiro;
```

```
Senão Retorne Falso;
```

```
} /* fim do procedimento */
```

Exercício 2.12 Implementação de uma "Pilha Burra"

```
/* arquivo PilhaSeqEstd.h - implementa o TAD Pilha sem utilizar recursos da orientação a objetos */
#include<conio.h>
#include<stdio.h>
#define Tamanho 3 // Tamanho da Pilha
typedef struct P {
    int Elementos[Tamanho]; // vetor de 0 a Tamanho-1
    int Topo;
} Pilha; // define o Tipo Estruturado Pilha
void Cria(Pilha *p){
    p->Topo = -1; // o vetor começa em 0; logo, inicializamos a pilha em -1
}
bool Vazia(Pilha *p){
    if(p->Topo == -1)
        return true;
    else
        return false;
}
bool Cheia(Pilha *p){
    if(p->Topo == Tamanho-1)
        return true;
    else
        return false;
}
void Empilha(Pilha *p, int x, bool *DeuCerto){
    if(Cheia(p)==1)
        *DeuCerto = false;
    else {
        p->Topo++;
        p->Elementos[p->Topo] = x;
        *DeuCerto = true;
    }
}
void Desempilha(Pilha *p, int *x, bool *DeuCerto){
    if(Vazia(p)==1)
        *DeuCerto = false;
    else {
        *x = p->Elementos[p->Topo];
        p->Topo--;
        *DeuCerto = true;
    }
}
/* arquivo PilhaSeqEstd.cpp - programa para testar o TAD Pilha */
#include "PilhaSeqEstd.h"
void imprime(Pilha *p){ // imprime sem abrir a TV
    int x;
    Pilha PAux;
    bool ok;
    Cria(&PAux);
    while (Vazia(p)==false) {
        Desempilha(p, &x, &ok);
        if(ok)
            printf("%d ", x);
    }
}
```



ELSEVIER

Estruturas de Dados com Jogos

```
if (ok) {
    Empilha(&PAux, x, &ok);
} // if
} // while
printf("\n imprimindo a pilha: ");
while (Vazia(&PAux)==false) {
    Desempilha(&PAux, &x, &ok);
    if (ok){
        printf("%d ", x);
        Empilha(p, x, &ok);
    } // if
} // while
printf(" <- topo");
}

int main(){
    Pilha p;
    Cria(&p);
    bool ok;
    int a;
    printf("\n tentando empilhar 4 elementos em uma Pilha em que só cabem 3");
    Empilha(&p,10,&ok); imprime (&p);
    Empilha(&p,20,&ok); imprime (&p);
    Empilha(&p,30,&ok); imprime (&p);
    Empilha(&p,40,&ok); // tamanho da Pilha é 3; 40 não será empilhado
    imprime (&p);
    printf("\n pressione uma tecla... \n"); getch();
    printf("\n tentando desempilhar 4 elementos de uma Pilha que só tem 3");
    Desempilha(&p,&a,&ok); imprime (&p);
    Desempilha(&p,&a,&ok); imprime (&p);
    Desempilha(&p,&a,&ok); imprime (&p);
    Desempilha(&p,&a,&ok); //Pilha vazia; nao vai conseguir desempilhar
    imprime (&p);
    printf("\n pressione uma tecla... \n"); getch(); return(0);
}
```

Exercício 2.13 Pilha Burra como classe em C++

```
/* arquivo PilhaBurra.h - implementa um TAD PilhaBurra como uma Classe */
#define Tamanho 3
class PilhaBurra {
private:
    int Topo;
    char Elementos[Tamanho];
public:
    PilhaBurra();
    bool Vazia();
    bool Cheia();
    void Empilha(int, bool &);
    void Desempilha(int &, bool &);
};

PilhaBurra::PilhaBurra () {
    Topo = -1;
}

bool PilhaBurra::Vazia () {
    if (Topo == -1)
        return true;
    else return false;
}

bool PilhaBurra::Cheia () {
    if (Topo == (Tamanho-1))
        return true;
    else return false;
}

void PilhaBurra::Empilha( int X, bool &DeuCerto) {
    if (Cheia())
        DeuCerto = false;
    else {
        DeuCerto = true;
        Topo = Topo + 1;
        Elementos[Topo] = X;
    }
}

void PilhaBurra::Desempilha( int &X, bool &DeuCerto) {
    if (Vazia())
        DeuCerto = false;
    else {
        DeuCerto = true;
        X = Elementos[ Topo ];
        Topo = Topo - 1;
    }
}
```



```
/* arquivo PilhaBurra.cpp */
/* programa para testar a TAD Pilha implementada como Classe */

#include<conio.h>
#include<stdio.h>
#include<iostream>
#include "PilhaBurra.h"

using namespace std;

void imprime(PilhaBurra &p){
    int x;
    PilhaBurra PAux;
    bool ok;
    while (p.Vazia()==false) {
        p.Desempilha(x, ok);
        if (ok) {
            PAux.Empilha(x, ok);
        } // if
    } // while
    cout << "imprimindo a pilha: ";
    while (PAux.Vazia()==false) {
        PAux.Desempilha(x, ok);
        if (ok){
            cout << x << " ";
            p.Empilha(x, ok);
        } // if
    } // while
    cout << "--- topo" << endl;
}

int main(){
    PilhaBurra p;
    bool ok;
    char op = 't';
    int valor;
    while (op != 's') {
        cout << "digite: (e)empilhar,(d)desempilar, (s)sair [enter]" << endl;
        cin >> op;
        switch (op) {
            case 'e' : cout << "digite valor INTEIRO para empilhar empilhar [enter]" << endl;
                        cin >> valor; // CUIDADO: DIGITE UM INTEIRO MESMO!!!
                        p.Empilha(valor, ok);
                        if (ok==true) cout << "valor empilhado= " << valor << endl;
                        else cout << "nao conseguiu empilhar" << endl;
                        break;
            case 'd' : p.Desempilha(valor,ok);
                        if (ok==true) cout << "valor desempilhado= " << valor << endl;
                        else cout << "nao conseguiu desempilar" << endl;
                        break;
            default : cout << "saindo..." << endl; op = 's'; break;
        }; // case
        imprime (p);
    } // while
    cout << "pressione uma tecla... " << endl;
    getch();
    return(0);
}
```

Exercício 2.14 Carta do Baralho

Sugestões:

- A Carta-do-Baralho pode ser um tipo estruturado ou uma classe, com dois campos: Naipe (ouros, copas, espadas e paus) e Valor (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K). O naipe pode ser definido como um tipo enumerado ou do tipo inteiro (nesse caso, estabelecendo uma correlação do tipo ouros = 1, copas = 2, e assim por diante). Valor pode ser um tipo enumerado, do tipo inteiro ou do tipo Char.
- Também é possível definir operações sobre cartas — operação para retornar o naipe, retornar o valor, atualizar o naipe, atualizar o valor, entre outras.

Exercício 2.15 e 2.16 Pilha Intermediária Inteligente e Pilha Definitiva Inteligente

Sugestões:

- Defina uma classe PilhaBurra de elementos do tipo Carta-do-Baralho. Então defina uma classe PilhalIntermediária herdando características da classe PilhaBurra; reescreva a operação Empilha, só permitindo empilhar cartas na sequência correta; em uma PilhalIntermediária podemos empilhar qualquer carta se a Pilha estiver vazia ou em ordem decrescente e em cores alternadas, se a Pilha não estiver vazia.
- Faça o mesmo para a classe PilhaDefinitiva; em uma PilhaDefinitiva podemos empilhar somente a carta de valor A se a Pilha estiver vazia ou em ordem crescente e do mesmo naipe, se a Pilha não estiver vazia.
- Na prática você terá três operações Empilha (Pilha Burra, Pilha Definitiva e Pilha Intermediária). O código das demais operações será comum.

Para exemplificar, segue trecho de implementação de uma PilhalInteligente com elementos do tipo inteiro, na qual só é possível empilhar elementos maiores do que o elemento do topo da pilha.

```
class PilhalInteligente : public PilhaBurra{
public:
    void Empilha(int, bool &);           // Empilha condicionalmente
};

void PilhalInteligente::Empilha(int X, bool &DeuCerto){
    /* so empilha se X for maior que o elemento que está no topo da pilha ou se a pilha estiver vazia */
    if (Vazia())
        PilhaBurra::Empilha(X, DeuCerto); // se a pilha estiver vazia, empilha
    else {int Y;
        Desempilha(Y, DeuCerto); // pega o valor do topo - valor em Y
        PilhaBurra::Empilha(Y, DeuCerto); // volta o valor Y para o topo
        if (X > Y)
            PilhaBurra::Empilha(X, DeuCerto); // empilha X apenas se X>Y
        else DeuCerto = false;
    }
}
```

Links

1. Games On: Torre de Hanoi: <http://www.gameson.com.br/Jogos-Online/ClassicoPuzzle/Torre-de-Hanoi.html> (consulta em setembro de 2013).
2. UOL Jogos: Torre de Hanoi: http://jogosonline.uol.com.br/torre-de-hanoi_1877.html (consulta em setembro de 2013).
3. YouTube: Torres de Hanoi: Bezerra: <http://www.youtube.com/watch?v=yrNWFFbcEY> (consulta em setembro de 2013).
4. YouTube: Torres de Hanoi: Neves: http://www.youtube.com/watch?v=3qTe_X1yXEs (consulta em setembro de 2013).

Referências e leitura adicional

Knuth D. *The Art of Computer Programming*. Volume I: Fundamental Algorithms. Reading, MA: Addison-Wesley, 1972.

Langsam Y, Augenstein MJ, Tenenbaum A M. *Data Structures Using C and C++*. 2nd ed. Upper Saddle River. New Jersey: Prentice Hall, 1996.

Pereira SL. *Estruturas de dados fundamentais: conceitos e aplicações*. São Paulo: Érica; 1996.