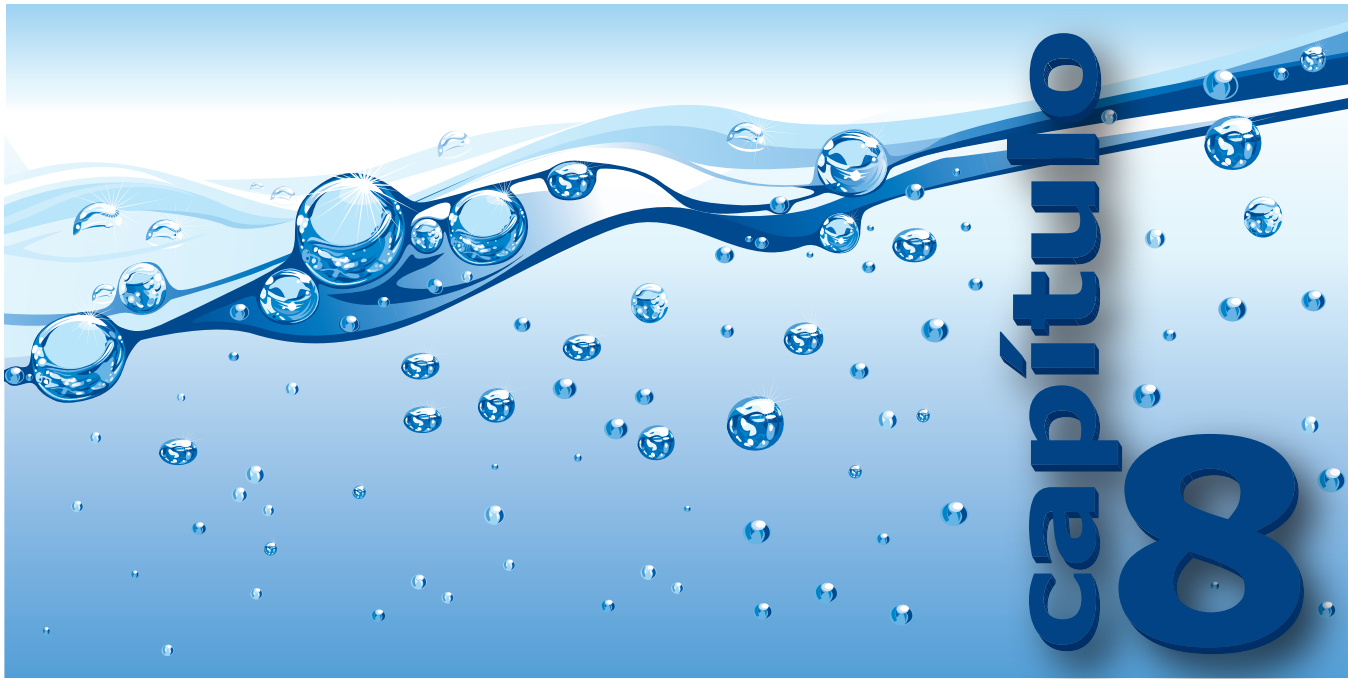


Pilhas e Filas

objetivos

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Pilha
  - ❒ Fila
  - ❒ Estrutura LIFO
  - ❒ Estrutura FIFO
  - ❒ Empilhamento
  - ❒ Desempilhamento
  - ❒ Pilha vazia
  - ❒ Fila vazia
  - ❒ Fila linear
  - ❒ Fila circular
  - ❒ Contêiner
  - ❒ Dicionário
  - ❒ Enfileiramento (acréscimo)
  - ❒ Desenfileiramento (retirada)
  - ❒ Meta-algoritmo
  - ❒ Palíndromo
  - ❒ Desfazimento (*undo*)
  - ❒ Refazimento (*redo*)
- Descrever as operações definidas sobre os TADs pilha e fila
- Apresentar exemplos de aplicações práticas de pilhas e filas
- Contrastar os conceitos de lista, pilha e fila
- Analisar os custos temporais de operações sobre pilhas e filas
- Identificar quando é apropriado o uso de pilha ou fila na solução de um determinado problema
- Explicar o funcionamento de uma fila circular
- Apresentar as vantagens de uma fila circular com relação a uma fila linear
- Explicar o papel desempenhado por resto de divisão em implementação de filas circular
- Transformar recursão em iteração usando pilha
- Explicar o papel desempenhado por uma pilha em implementações de operações de desfazimento/refazimento

DICIONÁRIOS SÃO ESTRUTURAS DE DADOS que permitem acesso a seus componentes de acordo com seus conteúdos. Assim as listas apresentadas no **Capítulo 7** constituem exemplos de dicionários. Por outro lado, **contêineres** são estruturas de dados que permitem armazenamento e recuperação de dados independentemente de conteúdo. Este capítulo lida com duas categorias de contêineres: pilhas e filas.

8.1 Pilhas

8.1.1 Conceito

Uma **pilha** é um contêiner no qual qualquer inserção ou retirada de elementos é feita numa de suas extremidades, denominada **topo** da pilha. A denominação dessa abstração é derivada de sua analogia com uma pilha de objetos, como documentos impressos numa mesa de escritório, por exemplo. Como mostra a **Figura 8–1**, os documentos são sempre colocados no topo da pilha e retirados do topo da pilha, i.e. o último documento que foi colocado é o primeiro a ser retirado. Pilhas são frequentemente denominadas estruturas **LIFO**, que é uma expressão derivada da língua inglesa: *Last In, First Out* (i.e., o último a entrar é o primeiro a sair).

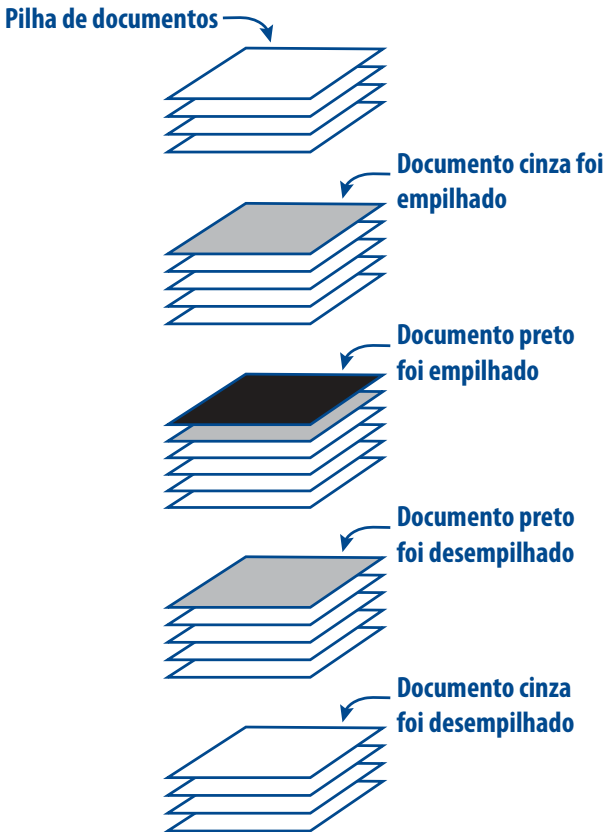


FIGURA 8–1: PILHA DE DOCUMENTOS

A **Figura 8–2** mostra, esquematicamente, a representação de uma pilha de caracteres. Nessa figura, se fosse requerida a retirada de um elemento (**desempilhamento**), o elemento removido seria `D` e o topo da pilha passaria a apontar para o elemento `C` (v. **Figura 8–3**). Por outro lado, se um elemento `E` precisasse ser inserido na pilha (**empilhamento**), ele seria colocado sobre o elemento `D` e o topo passaria a apontar para o elemento `E` (v. **Figura 8–4**). Assim o topo da pilha é movido para cima ou para baixo, conforme se acrescente (**empilhe**) ou remova (**desempilhe**) elementos da pilha.

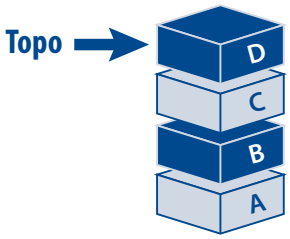


FIGURA 8–2: PILHA DE CARACTERES

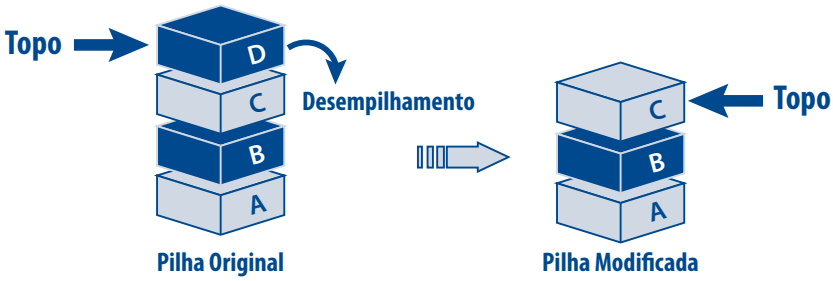


FIGURA 8–3: OPERAÇÃO DE DESEMPILHAMENTO

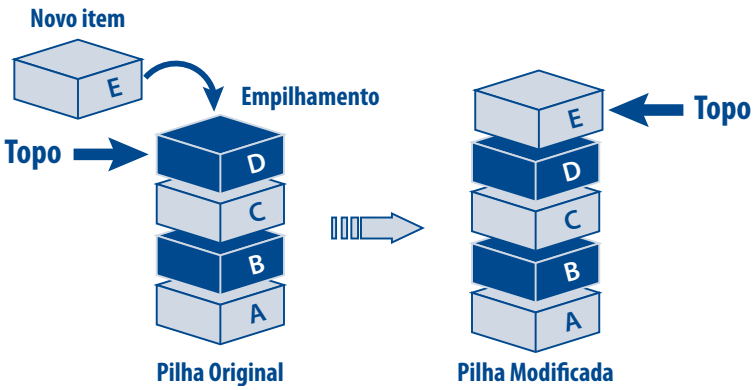


FIGURA 8–4: OPERAÇÃO DE EMPILHAMENTO

O conceito de pilha é largamente utilizado em programação. Um exemplo de sua utilização é no controle de execução de programas, que foi apresentado em detalhes na **Seção 4.3** e explorado em boa parte do **Capítulo 4**. Outras aplicações de pilhas são:

- Balanceamento (**casamento**) de símbolos (v. **Seção 8.5.2**)
- Avaliação de expressões aritméticas (v. **Seção 11.6**)
- Transformação de recursão em iteração (v. **Seção 8.4**)
- Implementação de desfazer/refazer (v. **Seção 8.5.4**)
- Casamento de etiquetas (*tags*, em inglês) em HTML e XML

As operações definidas sobre o tipo abstrato pilha são:

[1] Criação da pilha

[2] Empilhamento, i.e., acréscimo de um novo elemento acima do topo pilha

[3] Desempilhamento, i.e., retirada do elemento do topo da pilha

[4] Consulta ao elemento do topo da pilha sem desempilhá-lo

[5] Checagem de pilha vazia

8.1.2 Implementação

Uma das formas mais simples e elegantes de se representar uma pilha é por meio de uma estrutura com dois campos:

1. Um array unidimensional para conter os elementos da pilha (campo **itens**) e
2. Um indicador para o topo da pilha (campo **topo**).

Assim o tipo definido como:

```
typedef struct {
            tItemPilha itens[MAX_ELEMENTOS];
            int        topo;
        } tPilhaIdx;
```

poderia ser utilizado na criação de pilhas com itens do tipo **tItemPilha**, com um número máximo de elementos determinado pela constante simbólica **MAX_ELEMENTOS**.

Para criar uma pilha do tipo **tPilhaIdx**, deve-se definir uma variável desse tipo e, para completar a criação da pilha, deve-se chamar a função **CriaPilhaIdx()** definida como:

```
void CriaPilhaIdx(tPilhaIdx *p)
{
   p->topo = -1;
}
```

Na função acima, a atribuição do valor inicial **-1** ao topo da pilha é decorrente do fato de o primeiro elemento da pilha ocupar a posição **0** no array.

As operações [4] e [5] são tão resumidas que, aparentemente, não precisariam ser colocadas em funções isoladas. Quer dizer, para verificar se a pilha **p** está vazia, bastaria testar a condição: **p.topo == -1** e, sendo a pilha não vazia, poder-se-ia consultar seu elemento do topo por meio da expressão: **p.item[p.topo]**. No entanto, o uso dessas instruções em um programa que utiliza pilha torna-o dependente de implementação. Com efeito, uma atribuição como **x = p.item[p.topo]** feita num programa-cliente assume que a pilha (abstração) é implementada como uma estrutura tendo um array unidimensional como um de seus campos. Se, posteriormente, desejar-se modificar a implementação de pilha (p. ex., com o uso de listas encadeadas, como será visto no **Capítulo 10**) todas as instruções semelhantes a essa feitas no programa-cliente necessitarão ser modificadas. Mas, se essas instruções forem confinadas em funções [aliás, como já foi feito em **CriaPilhaIdx()**], quando forem necessárias algumas modificações na implementação de pilha, essas mudanças ficarão restritas aos corpos das funções e não afetarão suas chamadas. Sugere-se, então, como implementação das operações [4] e [5], as funções definidas abaixo.

```
int PilhaIdxVazia(const tPilhaIdx *p)
{
   return p->topo == -1;
}
```

```
tItemPilha ElementoTopoIdx(const tPilhaIdx *p)
{
   ASSEGURA(!PilhaIdxVazia(p), "Erro: Pilha vazia.");
   return p->itens[p->topo];
}
```

A macro **ASSEGURA** invocada no corpo da função **ElementoTopoIdx()** foi definida na **Seção 7.4**.

As operações [2] e [3] são representadas, respectivamente, pelas funções **EmpilhaIdx()** e **DesempilhaIdx()** exibidas abaixo.

```
void EmpilhaIdx(tPilhaIdx *p, const tItemPilha *item)
{
   ASSEGURA(!PilhaIdxCheia(p), "Erro: Pilha cheia.");
   p->itens[++p->topo] = *item;
}
```

```
tItemPilha DesempilhaIdx(tPilhaIdx *p)
{
   ASSEGURA(!PilhaIdxVazia(p), "Erro: Pilha vazia.");
   return p->itens[p->topo--];
}
```

O uso das funções acima, além de tornar o conceito de pilha independente de sua implementação, torna um programa-cliente muito mais compreensível. Por exemplo, a expressão `PilhaIdxVazia(p)` é muito mais significativa do que `p.topo == -1`. Observe ainda que a condição de pilha cheia (`p->topo < MAX_ELEMENTOS`) não é uma exceção da abstração pilha, mas sim do array unidimensional no qual seus elementos são armazenados. Nas funções anteriores, o fato de a constante `MAX_ELEMENTOS` e uma variável indicando o topo da pilha não fazerem parte das listas de parâmetros formais também contribui para a independência de implementação da abstração pilha. De uma forma geral, confinar detalhes de implementação de uma abstração de dados em funções constitui-se uma boa norma de programação, na medida em que torna o programa que utiliza a abstração mais compreensível e fácil de modificar.

Para completar a implementação de pilha, resta apresentar a função `PilhaIdxCheia()`.Essa função faz parte da implementação de pilha, mas não faz parte da abstração pilha. Por isso, ela deve ser definida como `static`.

```
static int PilhaIdxCheia(const tPilhaIdx *p)
{
   return p->topo >= MAX_ELEMENTOS - 1;
}
```

8.1.3 Análise

Uma vez que nenhuma operação sobre pilha requer movimentação de elementos do array que armazena os itens da pilha, como ocorre com listas (v. **Seção 7.6.1**), todas as operações sobre pilhas têm custos temporal e espacial $\theta(1)$. Para facilidade de referência, a **Tabela 8–1** apresenta os custos temporais das operações sobre pilhas.

Operação	Custo Temporal
Criação	$\theta(1)$
Empilhamento	$\theta(1)$
Desempilhamento	$\theta(1)$
Consulta ao elemento do topo	$\theta(1)$
Verificação de pilha vazia	$\theta(1)$

Tabela 8–1: Custos Temporais de Operações sobre Pilhas

8.2 Filas Lineares

8.2.1 Conceito

Numa **fila**, todos os acréscimos de elementos são efetuados na extremidade final (**fundo**) da fila e todas as retiradas de elementos são efetuadas na extremidade inicial (**frente**) da fila. Isto significa que o primeiro elemento a ser retirado de uma fila é o primeiro que foi colocado nela. Por isso, frequentemente, a fila é denominada de

estrutura **FIFO**, expressão derivada do inglês *First In, First Out* (traduzindo: o primeiro a entrar é o primeiro a sair). O nome *fila* é derivado da analogia entre essa estrutura de dados e uma fila de pessoas (civilizadas).

Esquematicamente, pode-se representar uma fila de caracteres conforme ilustrado na **Figura 8–5**.

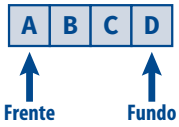


Figura 8–5: Fila de Caracteres

Filas aparecem com muita frequência na solução de problemas computacionais. Um uso bastante comum de filas é no compartilhamento de tempo de processamento por vários programas em execução simultânea. Nesse caso, as solicitações de uso do processador são enfileiradas e processadas uma após a outra de acordo com a ordem de chegada. Outras aplicações de filas são: tarefas de impressão, *call center* e simulações (v. **Seção 8.5.5**).

As operações requeridas sobre as filas são semelhantes àquelas definidas para as pilhas, mas implementadas de modo diferente e menos trivial. O conjunto de operações sobre filas deve incluir:

[1] **Criação** da fila

[2] **Acréscimo** de um novo elemento no fundo da fila

[3] **Retirada** do elemento da frente da fila

[4] Consulta ao **elemento da frente** da fila, sem retirá-lo

[5] Checagem de **fila vazia**

A **Figura 8–6** mostra a fila da **Figura 8–5** após uma operação de **desenfileiramento**, enquanto a **Figura 8–7** apresenta essa mesma fila após uma operação de **enfileiramento**.

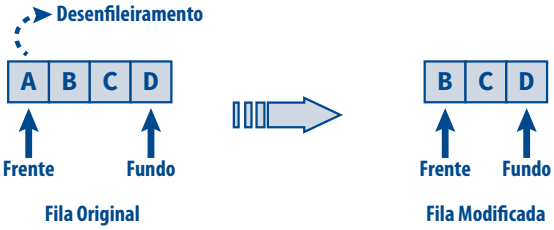


Figura 8–6: Operação de Desenfileiramento numa Fila de Caracteres

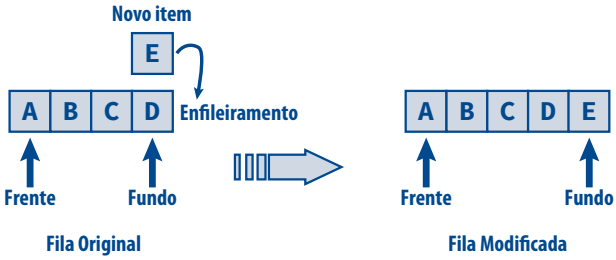


Figura 8–7: Operação de Enfileiramento numa Fila de Caracteres

A denominação *filas lineares*, que intitula a presente seção, diz respeito a um modo específico de implementação que será aqui explorado. A razão para tal denominação será esclarecida na **Seção 8.3**.

8.2.2 Implementação

Novamente, na implementação de filas, será considerado o uso de um tipo de estrutura tendo como um dos seus campos um array unidimensional para conter os elementos de uma fila. Nesse tipo de estrutura, serão

necessários dois campos adicionais para indicarem a frente e o fundo de uma fila. Assim a seguinte definição de tipo será usada:

```
typedef struct {
            tItemFila itens[MAX_ELEMENTOS];
            int       frente, fundo;
        } tFilaIdx;
```

A convenção a ser adotada aqui é que o campo **frente** indica sempre uma posição do array anterior à posição real do elemento que se encontra na frente da fila, enquanto o campo **fundo** sempre indica a posição do elemento no fundo da fila. Desse modo, **frente** é igual a **fundo** se e somente se não há nenhum elemento na fila; i.e., se a fila está vazia. Deve-se salientar que essas considerações são apenas uma convenção e que outras convenções poderiam, da mesma forma, ter sido adotadas sem que o conceito abstrato de fila fosse alterado.

Levando em consideração essa convenção, as funções `CriaFilaIdx()`, que inicia uma fila, e `FilaIdxVazia()`, que verifica se uma fila está vazia, podem ser definidas como:

```
void CriaFilaIdx(tFilaIdx *f)
{
   f->frente = f->fundo = -1;
}

int FilaIdxVazia(const tFilaIdx *f)
{
   return f->frente == f->fundo;
}
```

A **Figura 8–8** ilustra várias operações de acréscimo e retirada de elementos de uma fila `f` do tipo `tFilaIdx` definido acima.

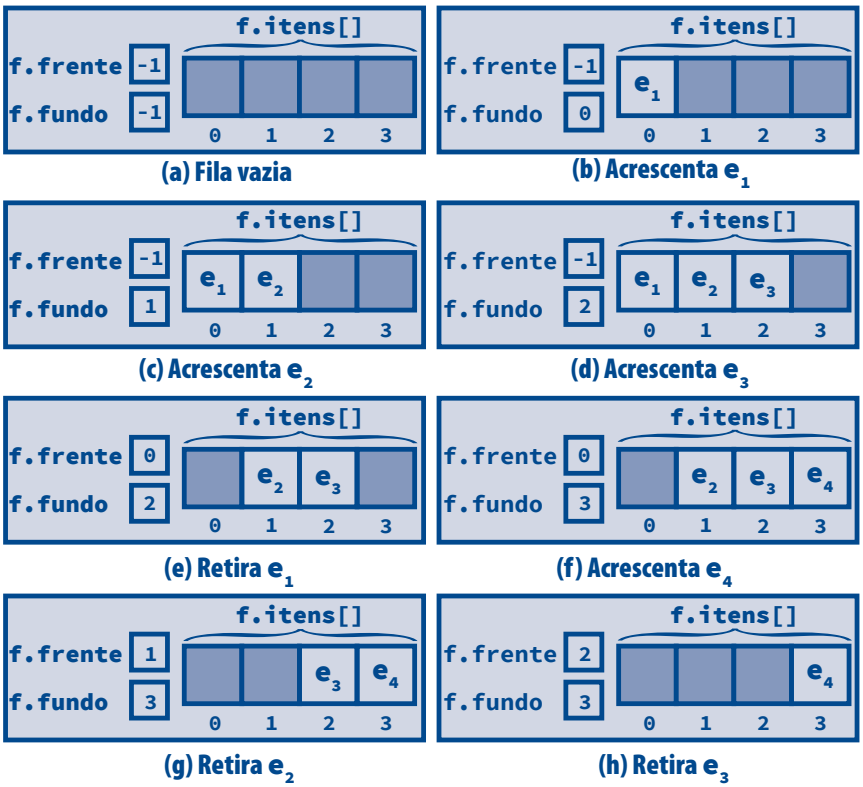


FIGURA 8–8: OPERAÇÕES DE ACRÉSCIMO E RETIRADA NUMA FILA LINEAR

A função `ElementoFrenteIdx()`, apresentada abaixo, permite a consulta ao elemento que se encontra na frente de uma fila sem removê-lo da fila.

```
tItemFila ElementoFrenteIdx(const tFilaIdx *f)
{
   ASSEGURA(!FilaIdxVazia(f), "Erro: Fila vazia");

   return f->itens[f->frente + 1];
}
```

As funções `EnfileiraIdx()` e `DesenfileiraIdx()` implementam as operações de acréscimo e retirada de elementos de uma fila.

```
void EnfileiraIdx(const tItemFila *item, tFilaIdx *f)
{
   ASSEGURA(!FilaIdxCheia(f), "Erro: Fila cheia.");

   ++f->fundo;
   f->itens[f->fundo] = *item;
}
tItemFila DesenfileiraIdx(tFilaIdx *f)
{
   ASSEGURA(!FilaIdxVazia(f), "Erro: Fila vazia.");

   ++f->frente;

   return f->itens[f->frente];
}
```

Agora, examine novamente a **Figura 8–8** e note que o fato de o fundo da fila ser igual ao maior índice do array que armazena seus elementos não implica necessariamente que haja `MAX_ELEMENTOS` elementos na fila e que, portanto, a fila esteja cheia. Em particular, esse fato ocorre na **Figura 8–8 (h)**, quando só há um único elemento na fila. Como o apontador da frente da fila é incrementado a cada retirada de elemento, a fila só deve ser considerada cheia quando as seguintes condições forem satisfeitas ao mesmo tempo:

[1] O fundo da fila é igual a `MAX_ELEMENTOS - 1`

[2] A frente da fila é igual a `-1`

É importante salientar que essas condições que indicam se uma fila está cheia estão intrinsecamente associadas à convenção adotada para indicar a frente e o fundo de uma fila.

Diante do que foi exposto acima, faz-se necessária uma função que detecte a existência de espaço vazio no início do array e promova o afastamento da fila para o início do array para permitir novos acréscimos na fila. Essa função deve ainda informar quando a fila estiver realmente cheia. A função `FilaIdxCheia()`, apresentada a seguir, leva em consideração o que foi exposto.

```
static int FilaIdxCheia(tFilaIdx *f)
{
   int i, deslocamento;

   if (f->fundo < MAX_ELEMENTOS - 1)
      return 0; /* Há espaço no final do array */

   if (f->frente == -1 && f->fundo == MAX_ELEMENTOS - 1)
      return 1; /* Não há mais espaço no array */

      /* Neste ponto, sabe-se que há espaço, mas é necessário mover os  */
      /* elementos da fila para o início do array. O deslocamento a ser */
      /* feito em cada elemento é dado por:                             */
   deslocamento = MAX_ELEMENTOS - (f->fundo - f->frente);
```

```
      /* O laço for a seguir desloca cada elemento da fila para o início do array */
   for (i = f->frente + 1; i <= f->fundo; ++i)
      f->itens[i - deslocamento] = f->itens[i];

      /* A frente e o fundo da fila também precisam ser deslocados */
   f->frente -= deslocamento;
   f->fundo -= deslocamento;

      /* A fila foi deslocada e agora podem-se acrescentar novos itens */
   return 0;
}
```

Note que a função `FilaIdxCheia()` é definida usando **static** porque essa operação é necessária apenas como parte da implementação e não é parte do conceito abstrato de fila.

8.2.3 Análise

A única operação sobre fila que não tem $\theta(1)$ como custo temporal é a operação de acréscimo que, em seu pior caso, tem custo temporal $\theta(n)$ devido ao afastamento de itens. Para facilidade de referência, a **Tabela 8–2** apresenta os custos das operações sobre filas.

Operação	Custo Temporal
Criação	$\theta(1)$
Acréscimo	❒ $\theta(1)$ no melhor caso ❒ $\theta(n)$ no pior caso
Retirada	$\theta(1)$
Consulta ao elemento da frente	$\theta(1)$
Verificação de fila vazia	$\theta(1)$

Tabela 8–2: Custos Temporais de Operações sobre Filas Lineares

8.3 Filas Circulares

Uma forma de evitar o deslocamento de uma fila para o início do array que armazena seus elementos para ceder espaço para acréscimo de elementos é por meio de uma **fila circular**. Nesse caso, o array é visto como circular ao invés de linear e é mais conveniente especificar seu número de elementos como sendo `MAX_ELEMENTOS + 1` (mais adiante, você verá por qual razão) e iniciar o armazenamento de elementos a partir do índice `1` do array (isto é apenas convenção).

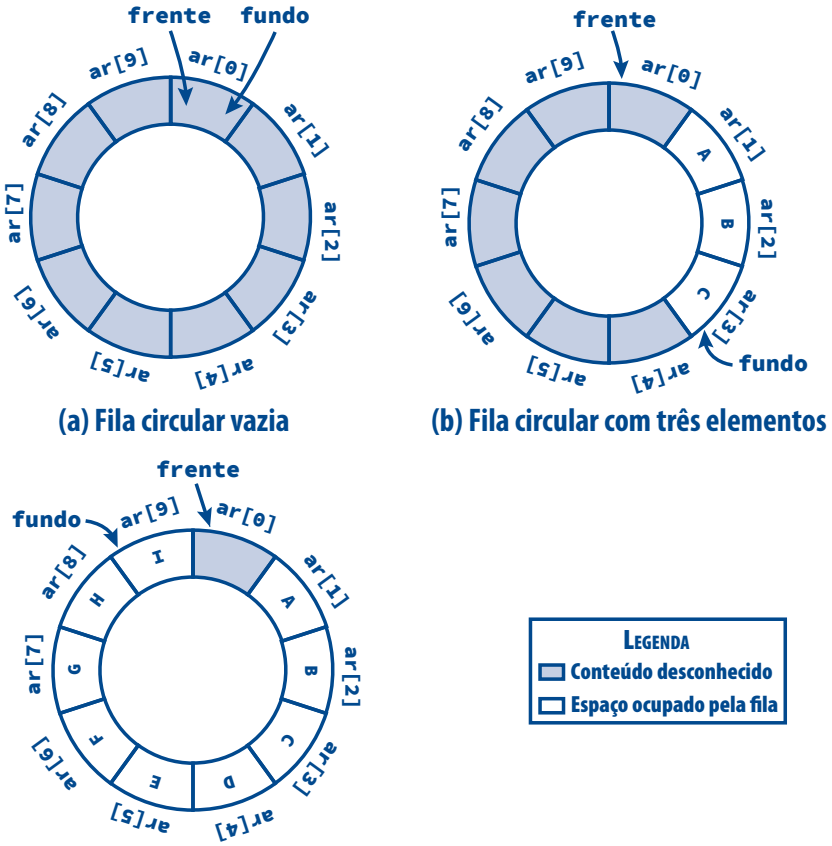
A **Figura 8–9** ilustra três filas circulares de caracteres. Note que uma fila circular é considerada cheia [v. **Figura 8–9 (c)**] mesmo que haja um espaço sobressalente no array. Mais adiante, você entenderá a razão para esse fato.

A definição de tipo que permite a criação de filas circulares sofre uma pequena alteração com relação àquela vista na **Seção 8.2**:

```
typedef struct {
           tItemFila itens[MAX_ELEMENTOS + 1];
           int       frente, fundo;
        } tFilaC;
```

A **Figura 8–10** ilustra várias operações de acréscimo e retirada de elementos de uma fila circular `fc` do tipo `tFilaC`. Nessa figura, o array que armazena os itens da fila possui quatro elementos (i.e., `MAX_ELEMENTOS = 3`).

Compare essa figura com a **Figura 8–8** e observe que as operações efetuadas sobre as filas `f` e `fc` são as mesmas, mas os elementos dessas filas são dispostos nos respectivos arrays de maneiras diferentes.

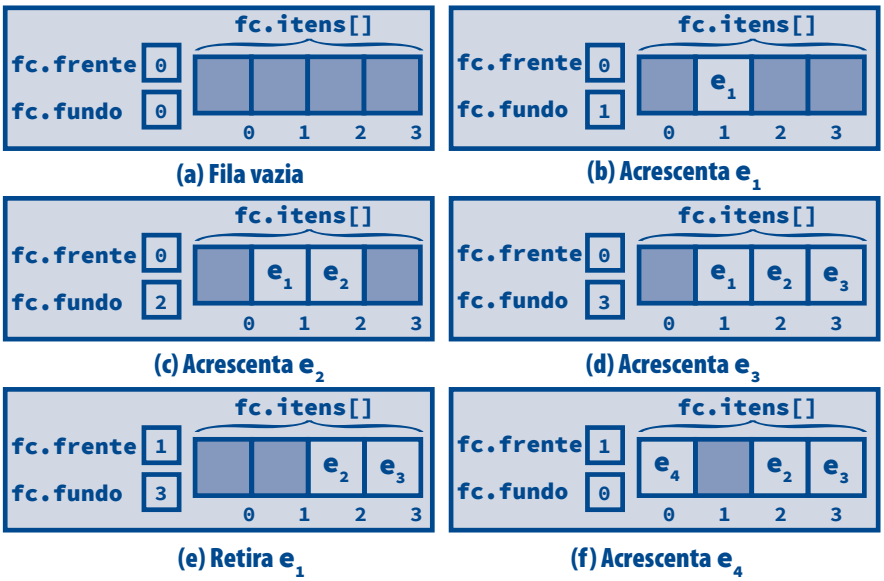


(a) Fila circular vazia

(b) Fila circular com três elementos

(c) Fila circular completa

Figura 8–9: Filas Circulares de Caracteres



(a) Fila vazia

(b) Acrescenta e_1

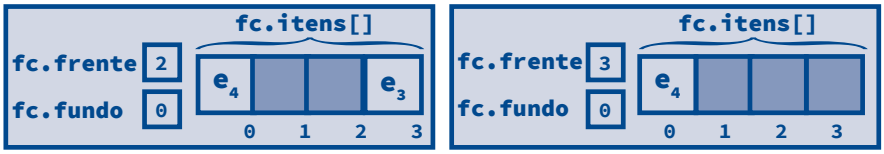
(c) Acrescenta e_2

(d) Acrescenta e_3

(e) Retira e_1

(f) Acrescenta e_4

Figura 8–10: Operações de Acréscimo e Retirada numa Fila Circular



(g) Retira e_2 (h) Retira e_3

FIGURA 8–10 (CONT.): OPERAÇÕES DE ACRÉSCIMO E RETIRADA NUMA FILA CIRCULAR

Note que, numa fila circular, quando o fundo é igual ao maior índice do array que acomoda os elementos da fila (que, neste caso, é `MAX_ELEMENTOS`), o próximo elemento é acrescentado na posição `0`, se essa posição estiver vazia, como mostra a **Figura 8–10 (f)**.

Usando a mesma convenção vista na **Seção 8.2**, o campo `frente` apontará para uma posição anterior ao primeiro elemento da fila. Novamente, os campos `frente` e `fundo` serão iguais se e somente se a fila estiver vazia.

Agora, observe novamente a situação ilustrada na **Figura 8–9 (c)**. Nesse caso, há um espaço sobressalente no array que armazena os elementos da fila, mas se for acrescentado mais um elemento a essa fila, ter-se-á que os campos `frente` e `fundo` serão iguais, o que, de acordo com a convenção assumida, indica que a fila está vazia. Ocorre, porém, que nesse caso se chega a uma contradição porque, de fato, o array que armazena os elementos não comporta mais nenhum novo elemento (i.e., a fila está cheia). Portanto, para que seja possível distinguir entre uma fila cheia e uma fila vazia, o número máximo de elementos permitidos numa fila circular implementada num array com `N` posições é `N - 1`. Consequentemente, para a fila comportar `MAX_ELEMENTOS` é necessário que esse array seja dimensionado com `MAX_ELEMENTOS + 1`, pois uma dessas posições é desperdiçada.

Uma maneira de permitir o uso de todos os elementos do array seria acrescentar um campo denominado, por exemplo, `nElementos`, que mantivesse atualizado o número de elementos na fila. Assim a fila `f` estaria cheia apenas quando o campo `nElementos` fosse igual ao número de elementos do array. No entanto, isto levaria a um acréscimo no número de instruções nas funções que implementam operações sobre filas e aumentaria seus tempos de execução, o que provavelmente não compensaria o ganho de uma única posição na fila. Logo essa abordagem não será adotada aqui.

A suposição de circularidade provoca pequenas mudanças nas funções `EnfileiraIdx()` e `DesenfileiraIdx()` apresentadas na **Seção 8.2**. Quer dizer, para acrescentar um elemento, será necessário mover o campo `fundo` uma posição *circularmente*. Ou seja, a instrução da função `EnfileiraIdx()` apresentada na **Seção 8.2**:

```
++f->fundo;
```

deve ser reescrita como:

```
if (f->fundo == MAX_ELEMENTOS)
   f->fundo = 0;
else
   ++f->fundo;
```

Ora, mas essa última instrução **if** pode ser escrita de modo mais sucinto como:

```
(f->fundo == MAX_ELEMENTOS) ? (f->fundo = 0) : (++f->fundo);
```

Ou, ainda melhor, como:

```
f->fundo = (f->fundo + 1) % (MAX_ELEMENTOS + 1);
```

levando em consideração a definição de resto de divisão inteira (representada por **%**).

Tente entender por que essas duas últimas instruções são equivalentes:

Se `f->fundo` é igual a `MAX_ELEMENTOS`, então `(f->fundo + 1) % (MAX_ELEMENTOS + 1)` é igual a `0`. Por outro lado, se `f->fundo` é diferente de `MAX_ELEMENTOS`, `f->fundo` é menor do que `MAX_ELEMENTOS`. Logo `(f->fundo + 1) % (MAX_ELEMENTOS + 1)` é igual a `f->fundo + 1`.

De modo semelhante, quando é feita uma retirada, é necessário mover `f->frente` uma posição circularmente, o que pode ser realizado utilizando o operador `%` da seguinte forma:

```
f->frente = (f->frente + 1) % (MAX_ELEMENTOS + 1)
```

As funções completas para acréscimo e retirada de elementos de uma fila circular são apresentadas a seguir.

```
void AcrescentaC(tFilaC *f, const tItemFila *item)
{
   ASSEGURA(!FilaCheiaC(f), "Erro: Fila cheia.");
   f->fundo = (f->fundo + 1) % (MAX_ELEMENTOS + 1);
   f->itens[f->fundo] = *item;
}
tItemFila RetiraC(tFilaC *f)
{
   ASSEGURA(!FilaVaziaC(f), "Erro: Fila vazia.");
   f->frente = (f->frente + 1) % (MAX_ELEMENTOS + 1);
   return f->itens[f->frente];
}
```

Finalmente, a função `FilaCheiaC()` pode ser implementada como mostrado a seguir:

```
static int FilaCheiaC(const tFilaC *f)
{
   return f->frente == (f->fundo + 1) % (MAX_ELEMENTOS + 1);
}
```

A função que verifica se uma fila circular está vazia é implementada de modo semelhante à função `FilaIdxVazia()` definida na **Seção 8.2** e a função que retorna o elemento que se encontrada na frente de uma fila circular é semelhante à função `ElementoFrenteIdx()` definida nessa mesma seção.

Em filas circulares, nenhuma operação depende do tamanho da fila. Portanto todas elas têm custos temporal e espacial $\theta(1)$. A **Tabela 8–3** resume esses resultados para facilidade de referência.

Operação	Custo Temporal
Criação	$\theta(1)$
Acréscimo	$\theta(1)$
Retirada	$\theta(1)$
Consulta ao elemento da frente	$\theta(1)$
Verificação de fila vazia	$\theta(1)$

Tabela 8–3: Custos Temporais de Operações sobre Filas Circulares

8.4 Transformando Recursão em Iteração Usando Pilha

Converter funções recursivas em iterativas é sempre uma boa medida a ser seguida quando se deseja obter um programa eficiente, conforme foi amplamente discutido no **Capítulo 4**. Em particular, a **Seção 4.4** mostrou como se pode facilmente transformar recursão de cauda em iteração (i.e., laços de repetição). Essa seção também mostrou que, se um bom compilador for utilizado, ele é capaz de transformar recursão de cauda em iteração automaticamente.

O estudo do **Capítulo 4** revela ainda que existe uma íntima relação entre recursão e uso de pilha, pois quando ocorre uma chamada recursiva, o sistema cria um novo registro de ativação e o empilha e, quando ocorre o retorno dessa chamada, o registro de ativação correspondente é desempilhado. Esses fatos sugerem que uma função recursiva pode ser substituída por outra função iterativa que faça uso adequado de pilha. É isso que esta seção se propõe a investigar.

Em primeiro lugar, pode-se afirmar categoricamente que qualquer algoritmo recursivo pode ser transformado num algoritmo iterativo funcionalmente equivalente e a prova dessa afirmação é muito simples. Em linguagem de máquina, para a qual qualquer programa, em última instância, será traduzido não existe recursão. Na prática, entretanto, nem sempre é fácil obter um algoritmo iterativo a partir de um algoritmo recursivo já existente.

Quando se tem uma função recursiva que não apresenta recursão de cauda, sua transformação numa função iterativa muitas vezes requer o uso de pilha, que pode não ser tão trivial. Nesse caso, empilhamento e desempilhamento que são efetuados automaticamente pelo sistema no caso de recursão (v. **Seção 4.3**), passam a ser responsabilidade do programador no caso de iteração.

A substituição de uma função recursiva por uma função iterativa funcionalmente equivalente deve seguir o procedimento descrito na **Figura 8–11**. Esse procedimento trata-se de um **meta-algoritmo**, visto que ele visa orientar a construção de algoritmos iterativos a partir de algoritmos recursivos previamente conhecidos.

META-ALGORITMO CONVERTEALGORITMOSRECURSIVOS

ENTRADA: Um algoritmo recursivo

SAÍDA: Um algoritmo iterativo funcionalmente equivalente

1. Utilize uma pilha *p* como variável local da função iterativa; os elementos dessa pilha devem ser estruturas, de modo que cada parâmetro da função seja representado por um campo desse tipo de estrutura
2. Substitua cada chamada recursiva da função recursiva por um empilhamento na pilha *p*
3. Substitua cada retorno da função recursiva por um desempilhamento na pilha *p*

FIGURA 8–11: META-ALGORITMO DE CONVERSÃO DE ALGORITMOS RECURSIVOS

Esse meta-algoritmo não é nenhuma varinha de condão, pois ele não informa exatamente como uma função recursiva pode ser transformada em função iterativa já que seus passos são vagos e imprecisos. Quer dizer, esse meta-algoritmo deve ser considerado como guia para transformar funções recursivas em funções iterativas, de modo que cada caso deve ser considerado à parte, com suas próprias peculiaridades.

Como exemplo de execução do meta-algoritmo descrito acima, considere novamente o Problema das torres de Hanói resolvido por meio de recursão na **Seção 4.8.1**. Para facilitar o entendimento, a função `TorresDeHanoi()` apresentada naquela seção será renomeada como `TorresRecursivas()`, ao passo que a função que a ser derivada aqui será denominada `TorresIterativas()`. Para facilidade de referência, a função `TorresRecursivas()` é reproduzida a seguir, com suas instruções de interesse numeradas de **[1]** a **[4]**.s

```
void TorresRecursivas( int nDiscos, int hasteOrigem,
                       int hasteDestino, int hasteAuxiliar )
{
   if (nDiscos == 1){
         /* Passo 1 do algoritmo: escreve e encerra */
[1]    printf( "Mova o disco 1 da haste %c para a haste %c\n",
               hasteOrigem, hasteDestino );
       return;
   }

      /* Passo 2 do algoritmo: move os n - 1 discos */
      /* de A para B usando C como auxiliar         */
[2]   TorresRecursivas( nDiscos - 1, hasteOrigem,
                        hasteAuxiliar, hasteDestino );

      /* Passo 3 do algoritmo: move último disco de A para C */
[3]   printf( "Mova o disco %d da haste %c para a haste %c\n",
              nDiscos, hasteOrigem, hasteDestino );

      /* Passo 4 do algoritmo: move os n-1 discos */
      /* de B para C usando A como auxiliar       */
[4]  TorresRecursivas(nDiscos - 1, hasteAuxiliar, hasteDestino, hasteOrigem);
}
```

Cada elemento armazenado na pilha gerenciada pela função `TorresIterativas()`, a ser apresentada adiante, é do tipo `tEstado` definido como:

```
typedef struct {
          char origem, destino, aux;
          int n;
       } tEstado;
```

Note que cada campo do tipo `tEstado` representa um parâmetro tanto da função recursiva, apresentada acima, quanto da função iterativa, apresentada a seguir.

```
void TorresIterativas( int n, int hasteOrigem, int hasteDestino, int hasteAuxiliar )
{
   tEstado    estado; /* Armazena um estado do problema */
   tPilhaIdx  pilha; /* Pilha de elementos do tipo tEstado */
                     /* que armazena estados do problema   */

   CriaPilhaIdx(&pilha);

      /* Armazena o estado inicial do problema, que são os  */
      /* valores dos parâmetros reais recebidos pela função */
   estado.n = n;
   estado.origem = hasteOrigem;
   estado.destino = hasteDestino;
   estado.aux = hasteAuxiliar;

      /* Transforma recursão em iteração substituindo chamadas recursivas em */
      /* empilhamentos e retornos de chamadas recursivas em desempilhamentos */
   while (1) {
         /* Substitui a primera chamada recursiva */
      while (estado.n > 0) {
         EmpilhaIdx(&estado, &pilha);
         Troca(&estado.destino, &estado.aux);
         estado.n--;
      }
```

```
        /* Se a pilha ficou vazia, o problema está resolvido */
      if (PilhaIdxVazia(&pilha))
         break;
      else
           /* Desempilhamento corresponde a um retorno de chamada recursiva */
         estado = DesempilhaIdx(&pilha);

         /* O último estado desempilhado representa um   */
         /* movimento de discos que será exibido na tela */
      printf( "Mova o disco %d da haste %c para a haste %c\n",
              estado.n, estado.origem, estado.destino );

         /* Substitui a segunda chamada recursiva */
      Troca(&estado.aux, &estado.origem);

      estado.n--;
   }
}
```

A seguir, a função `TorresIterativas()` será escrutinada e, para facilitar o entendimento, suponha que ela seja chamada como:

```
TorresIterativas(3, 'A', 'C', 'B');
```

Suponha ainda que o tipo `tPilhaIdx`, utilizado pela função `TorresIterativas()`, implementa pilhas em arrays com 20 elementos do tipo `tEstado` discutido acima.

A **Figura 8–12** mostra a situação das variáveis locais da função `TorresIterativas()` logo antes da execução do seu laço **while** externo.

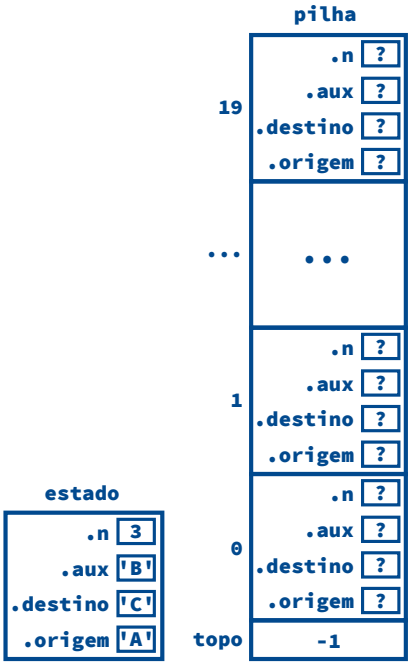


FIGURA 8–12: TORRES DE HANÓI: VERSÃO ITERATIVA 1

A **Figura 8–13** mostra a situação das variáveis locais da função `TorresIterativas()` logo após a primeira execução do laço **while** interno dessa função. É importante não confundir a pilha apresentada nessas figuras com as pilhas de execução apresentadas em exemplos do **Capítulo 4**.

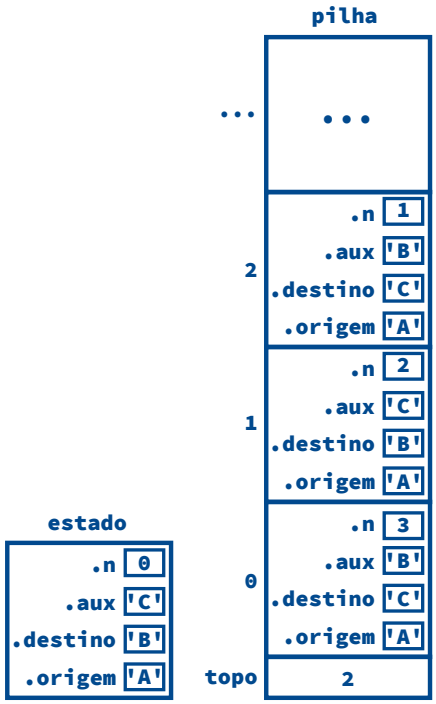


FIGURA 8–13: TORRES DE HANÓI: VERSÃO ITERATIVA 2

Os três empilhamentos mostrados na **Figura 8–13**, correspondem à primeira chamada recursiva da função `TorresRecursivas()` (instrução [**2**]). Após esses empilhamentos, ocorre um desempilhamento, corresponde a um retorno de chamada recursiva da função `TorresRecursivas()`. O elemento desempilhado representa o primeiro movimento exibido na tela por meio de uma chamada de **printf**(), que é:

```
Mova o disco 1 da haste A para a haste C
```

Neste instante, o conteúdo das variáveis locais da função sob discussão é aquele ilustrado na **Figura 8–14**.

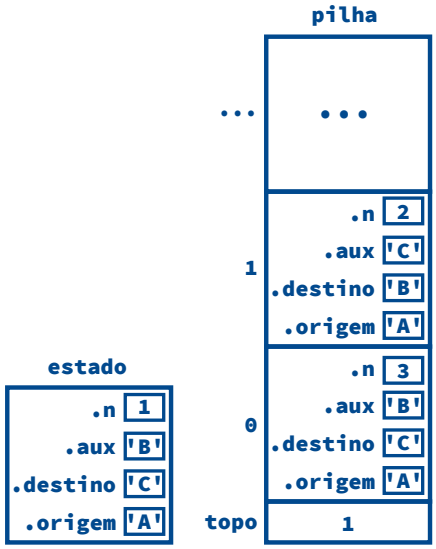


FIGURA 8–14: TORRES DE HANÓI: VERSÃO ITERATIVA 3

As duas próximas instruções da função `TorresIterativas()` que serão executadas são:

```
Troca(&estado.aux, &estado.origem);
estado.n--;
```

A primeira dessas instruções é uma chamada de função que troca os valores de duas hastes. Essas duas instruções combinadas com o laço **while** interno substituem a segunda chamada recursiva da função `TorresRecursivas()` (instrução [**4**]). Logo após a execução dessas duas instruções, a situação das variáveis locais da função sob discussão é aquela ilustrada na **Figura 8–15**.

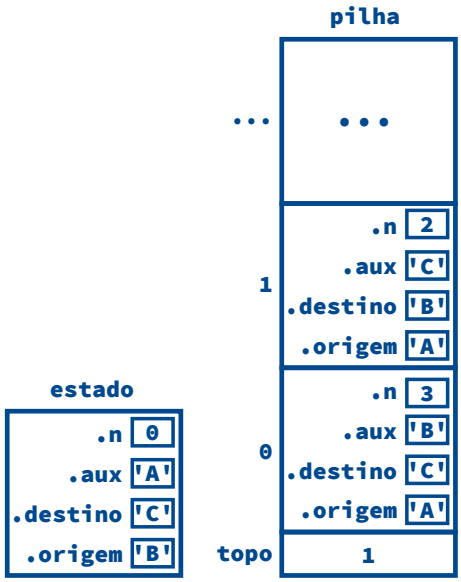


FIGURA 8–15: TORRES DE HANÓI: VERSÃO ITERATIVA 4

Como, neste instante, o valor de `estado.n` é igual a zero, o laço **while** interno não é executado, de modo que, desta vez, não ocorre empilhamento. Assim a parte **else** da instrução **if-else** é executada, o que resulta num novo desempilhamento que acarreta na exibição na tela de:

```
Mova o disco 2 da haste A para a haste B
```

Agora, a situação das variáveis locais da função `TorresIterativas()` é aquela ilustrada na **Figura 8–16**.

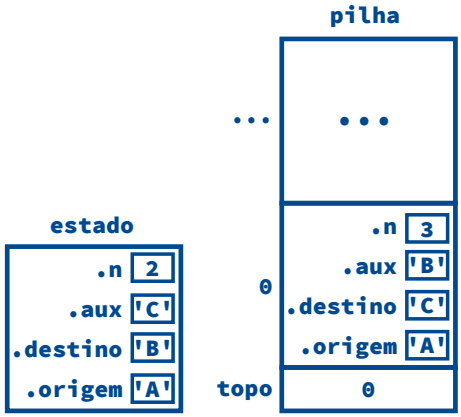


FIGURA 8–16: TORRES DE HANÓI: VERSÃO ITERATIVA 5

Em seguida, são executadas as duas últimas instruções do corpo do laço **while** externo, fazendo com que os conteúdos das variáveis locais da função `TorresIterativas()` sejam aqueles mostrados na **Figura 8–17**.

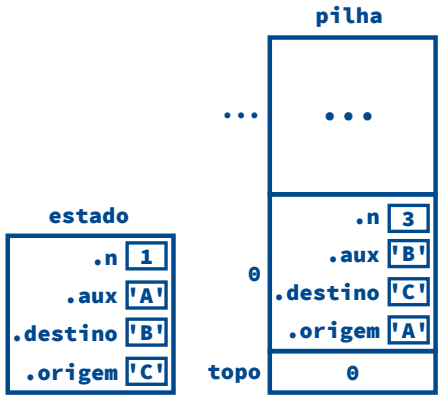


Figura 8–17: Torres de Hanói: Versão Iterativa 6

Agora, o laço **while** interno efetua mais um empilhamento e o conteúdo das variáveis locais sob escrutínio passa a ser aquele mostrado na **Figura 8–18**.

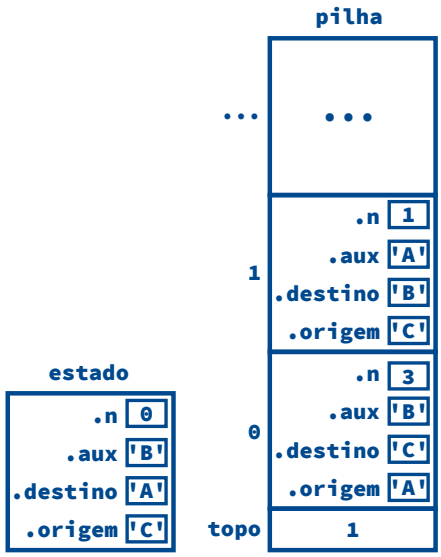


Figura 8–18: Torres de Hanói: Versão Iterativa 7

A partir daqui, a história se repetirá até que a pilha esteja vazia e, assim, a instrução **break** seja executada na instrução **if-else** da função `TorresIterativas()`. Quando isso ocorrer, o problema estará resolvido.

8.5 Exemplos de Programação

8.5.1 Invertendo Entradas 2

Problema: Escreva um programa que inverte a sequência de caracteres que o usuário digitar no meio de entrada padrão.

Solução: Esse problema já foi resolvido na **Seção 4.8.6** utilizando recursão. Aqui, será apresentada uma solução iterativa que utiliza pilha explicitamente.

```
int main (void)
{
   tPilhaIdx pilha;
   int       c;
```

```
   CriaPilhaIdx(&pilha);  /* Inicia a pilha */

   printf("\n>>> Digite uma sequencia de caracteres\n>>> para o programa inverter: ");

   while ((c = getchar()) != '\n')
      EmpilhaIdx(c, &pilha);

   while (!PilhaIdxVazia(pilha))
      putchar( DesempilhaIdx(&pilha) );

   return 0;
}
```

A função **main()** apresentada acima segue a seguinte linha de raciocínio:

[1] Cada caractere digitado pelo usuário é empilhado numa pilha de elementos do tipo **char**. Essa operação continua até que o usuário introduza o caractere `'\n'`, que representa `[ENTER]` ou `[RETURN]` no teclado e encerra qualquer entrada *normal* de dados.

[2] Após a leitura de `'\n'`, a função desempilha todos os caracteres que se encontram na pilha e exibe-os na tela um a um, produzindo, assim, o resultado desejado.

O custo temporal dessa operação ora implementada recursiva ora iterativamente é a mesma: $\theta(n)$, em que n é o número de caracteres lidos. O custo espacial também é o mesmo em ambos os casos: $\theta(n)$. Em termos de legibilidade, a versão recursiva parece ser a favorita. Afinal, você não precisa conhecer o conceito de pilha para implementá-la.

8.5.2 Casamento de Parênteses, Colchetes e Chaves

Preâmbulo: Um problema comum em programação é o casamento de símbolos que ocorrem aos pares: um de **abertura** e outro de **fechamento**. Por exemplo, normalmente, parênteses, colchetes e chaves aparecem emparelhados. Uma condição necessária para que haja casamento entre esses símbolos é que o número de símbolos de abertura seja igual ao número de símbolos de fechamento do mesmo tipo. Mas, essa condição não é suficiente para que ocorra o casamento. Por exemplo, os símbolos na **Figura 8–19** não casam, mesmo que o número de parênteses e colchetes de abertura seja igual ao número de parênteses e colchetes de fechamento:

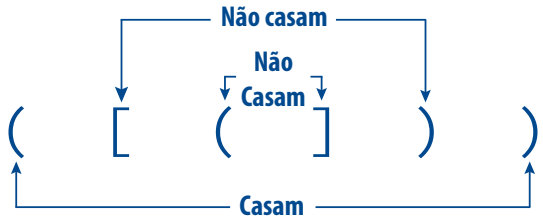


Figura 8–19: Parênteses e Colchetes que Não Casam

Pode-se fazer com que os símbolos que não casam na **Figura 8–19** passem a casar alterando suas posições, como mostra a **Figura 8–20**.

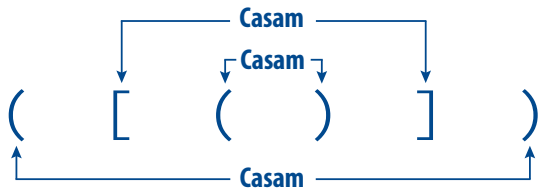


Figura 8–20: Parênteses e Colchetes que Casam

Problema: Escreva um programa que verifique se um string contendo parênteses, colchetes ou chaves apresenta má formação; i.e., o programa deve checar se todos esses símbolos casam.

Solução: A função **main**(), apresentada a seguir, resolve o problema proposto.

```
int main(void)
{
   tPilhaIdx  pilha;
   char       string[MAX], *p = string;
   int        posErro = 1, /* Posição de um possível erro */
              parentese;

   CriaPilhaIdx(&pilha);  /* Inicia pilha */

   printf( "\n>>> Digite uma sequencia de caracteres contendo"
           "\n>>> '()', '[]' ou '{}' para o programa verificar"
           "\n>>> se eles estao balanceados (max = %d caracteres):\n", MAX - 1 );
   LeString(p, MAX);

      /* Examina o string introduzido pelo usuário */
   for (; *p; ++p) {
         /* Aqui só interessam caracteres de abertura */
         /* (e.g., '(') ou fechamento (e.g., ')')     */
      if (EhAbertura(*p)) {
         EmpilhaIdx((tItemPilha) *p, &pilha); /* Caractere de abertura é empilhado */
      } else if (EhFechamento(*p)) {
            /* Caractere de fechamento causa desempilhamento e comparação */
         if (!PilhaIdxVazia(pilha)) {
            parentese = DesempilhaIdx(&pilha);

               /* Verifica se ocorre casamento entre o caractere */
               /* corrente e o caractere desempilhado            */
            if (!Casam(*p, parentese)) {
                  /* Não houve casamento Mostra onde ocorreu o erro */
               MostraErro(posErro);
               return 1;
            }
         } else {
               /* A pilha está vazia; portanto, não há casamento */
            MostraErro(posErro);
            return 1;
         }
      }

         /* Ainda não foi encontrado erro. Então, passa adiante. */
      ++posErro;
   }

   while (!PilhaIdxVazia(pilha))
      printf( "\nNao foi encontrado casamento para %c\n", DesempilhaIdx(&pilha) );

   return 0;
}
```

A função **main**() definida acima usa uma pilha de elementos do tipo **char** semelhante àquela definida na **Seção 8.1**. Essa função lê um string chamando a função `LeString()`, definida na **Seção 3.12.1**, e utiliza as seguintes funções auxiliares:

- `EhAbertura()` — verifica se um caractere é `'('`, `'['` ou `'{'`
- `EhFechamento()` — verifica se um caractere é `')'`, `']'` ou `'}'`

- `Casam()` — verifica se dois caracteres casam; i.e., se um é `'('` e o outro é `')'` etc
- `EscreveEspacos()` — escreve na tela o número de espaços em branco determinado por seu parâmetro; essa função é chamada por `MostraErro()` descrita a seguir
- `MostraErro()` — apresenta na tela um erro de casamento (examinar o exemplo de execução do programa ajuda a entender o que essa função faz)

Essas funções serão exibidas a seguir.

```
char *EhAbertura(int c)
{
   static const char *const abre = "([{";

   return strchr(abre, c);
}

char *EhFechamento(int c)
{
   static const char *const fecha = ")]}";

   return strchr(fecha, c);
}

int Casam(int fecha, int abre)
{
   if (fecha == ')' && abre == '(')
      return 1;
   else if (fecha == ']' && abre == '[')
      return 1;
   else if (fecha == '}' && abre == '{')
      return 1;

   return 0;
}

void EscreveEspacos(int n)
{
   for (int i = 1; i < n; ++i)
      putchar(' ');
}

void MostraErro(int posicao)
{
   EscreveEspacos(posicao);
   printf("^\n");
   EscreveEspacos(posicao);
   printf("|\n");
   EscreveEspacos(posicao);
   printf("--- Nao casa\n");
}
```

Exemplo de execução do programa:

```
>>> Digite uma sequencia de caracteres contendo
>>> '()', '[]' ou '{}' para o programa verificar
>>> se eles estao balanceados (max = 19 caracteres):
a ( b [ (c]))
          ^
          |
          --- Nao casa
```

8.5.3 Testando Palíndromos

Preâmbulo: Em linguajar cotidiano, um **palíndromo** é uma palavra ou frase que pode ser lida tanto em sentido direto quanto ao contrário (i.e., de trás para a frente). Tipicamente, num palíndromo, não se levam em consideração acentos, pontuações, hifens ou espaços entre palavras. Um dos exemplos de palíndromo mais simples da língua portuguesa é arara. Por outro lado, em programação, um palíndromo é um string que representa a mesma sequência de caracteres quando examinado tanto do início para o final quanto do final para o início. Em linguagem natural, tipicamente, a definição de palíndromo não leva em consideração acentuação, sinais de pontuação ou espaços em branco. Assim, por exemplo, em língua portuguesa, a frase *Socorram-me, subi no ônibus em Marrocos* é considerada um palíndromo, mas o mesmo não ocorre em programação.

Problema: Escreva um programa em C que utilize uma pilha de caracteres para determinar se um string é palíndromo (do ponto de vista de programação, é claro).

Solução: A ideia a ser implementada consiste em dividir conceitualmente o string em duas metades. À medida que os caracteres da primeira metade são acessados sequencialmente, eles são empilhados. Por outro lado, enquanto cada caractere da segunda metade é acessado, ele é comparado ao respectivo caractere desempilhado. Se todos os caracteres assim comparados casarem, o string sob escrutínio é um palíndromo; caso contrário, não se está lidando com um palíndromo. É importante notar que, se o comprimento do string for ímpar, haverá um caractere separando as duas metades aludidas. Nesse caso, esse caractere não tem nenhuma influência na verificação.

A função **main**(), definida a seguir, resolve o problema proposto usando pilha para verificar se os caracteres digitados pelo usuário constituem um palíndromo.

```
int main(void)
{
   tPilhaIdx pilha;
   char      string[TAM_ARRAY];
   int       tamString, metade, i;

   CriaPilhaIdx(&pilha);  /* Inicia pilha */

   printf( "\n>>> Digite uma sequencia de caracteres para"
           "\n>>> o programa verificar se e' palindromo"
           "\n>>> (max = %d caracteres): ", TAM_ARRAY - 1 );
   LeString(string, TAM_ARRAY);

   tamString = strlen(string);
   metade = tamString/2;

   for (i = 0; i < metade; ++i)
      EmpilhaIdx(string[i], &pilha);

   if (tamString%2)
      i = metade + 1; /* Tamanho do string é ímpar */
   else
      i = metade; /* Tamanho do string é par */

   while (!PilhaIdxVazia(pilha)) {
      if ( string[i] != DesempilhaIdx(&pilha) ) {
         printf("\n>>> \"%s\" NAO e' palindromo\n", string);
         return 0;
      }
      ++i;
   }
```

```
    printf("\n>>> \"%s\" e' um palindromo\n", string);
    return 0;
}
```

Exemplo de execução do programa:

```
>>> Digite uma sequencia de caracteres para
>>> o programa verificar se e' palindromo
>>> (max = 30 caracteres): omississimo

>>> "omississimo" e' um palindromo
```

Observação: O objetivo desse exemplo é apenas demonstrar o uso de pilhas. Quer dizer, existem maneiras mais eficientes de verificar se um string é palíndromo. Em particular, em termos de espaço, a abordagem utilizada aqui tem custo $\theta(n)$, em que n é o comprimento do string examinado. Outras abordagens não usam espaço adicional.

8.5.4 Fazendo, Desfazendo e Refazendo

Preâmbulo: Um dos princípios básicos de interação humano-computador (**HCI**, utilizando sigla comum em inglês) reza que o usuário deve ter oportunidade de **desfazer** toda ação que *pode* ser desfeita[1]. Por exemplo, qualquer editor de texto razoável permite que você desfaça a última ação que você executou. Quando esse princípio começou a ser implementado, os programas permitiam apenas que a última ação fosse desfeita, mas, hoje em dia, é comum que um programa permita que o usuário desfaça múltiplas ações. Essas ações são desfeitas na ordem inversa na qual elas foram executadas. Por exemplo, se, ao usar um editor de texto, você recortar e colar uma palavra, a primeira ação que pode ser desfeita é colar, que foi a última ação a ser executada. Por outro lado, quando uma ação é desfeita, deve-se permitir que ela seja **refeita**. Neste contexto, refazer uma ação significa desfazer o desfazimento de uma ação. Novamente, a ordem de refazimento (*redo*, em inglês) segue o mesmo raciocínio empregado em desfazimento (*undo*, em inglês).

Problema: Reescreva o programa da **Seção 7.6.1** (aquele da lista de compras) de modo que ele permita que o usuário desfaça e refaça o que pode ser desfeito e refeito, como qualquer programa *decente*.

Solução: Para implementar *undo*/*redo*, são necessários três ingredientes fundamentais:

1. Um tipo de estrutura que armazene todas as informações necessárias para realizar uma operação de desfazimento ou refazimento.
2. Uma pilha para implementação de desfazimento. Os elementos dessa pilha são do tipo de estrutura descrita como ingrediente **1**.
3. Uma pilha para implementação de refazimento. Essa pilha armazena conteúdos semelhantes àqueles da pilha de desfazimento [ingrediente **2**].

Relembrando o exemplo apresentado na **Seção 7.6.1**, as opções disponíveis no programa consistiam em operações básicas sobre uma lista de compras:

- ❒ Acrescenta um item na lista
- ❒ Insere um item na lista
- ❒ Remove um item da lista
- ❒ Modifica um item da lista

[1] O usuário deve alertado quando estiver prestes a executar uma ação que não pode ser desfeita, mas esse é outro princípio de interação que não será explorado aqui.

Como você deve estar desconfiado, todas essas operações podem ser desfeitas e refeitas. Mas, o que significa precisamente desfazer cada uma dessas ações? A **Tabela 8–4** responde essa questão.

Ação	Desfazimento
Acréscimo de um item na lista	Remoção do mesmo item da lista
Inserção de um item na lista	Idem
Remoção de um item da lista	Inserção do mesmo item na lista
Modificação de um item da lista	Modificação do mesmo item da lista

Tabela 8–4: Fazimentos e Desfazimentos numa Lista

Examinando-se detidamente a **Tabela 8–4**, conclui-se que uma estrutura que ofereça subsídios para o desfazimento de qualquer ação enumerada nessa tabela precisa armazenar apenas duas informações sobre o item que sofreu a referida ação: seu conteúdo e sua posição na lista no instante em que a ação foi efetuada. É importante ainda que tal estrutura informe qual é o tipo de ação a qual ela se refere.

Utilizando-se uma análise semelhante àquela discutida acima, chega-se à conclusão que o mesmo tipo de estrutura utilizada para desfazimento pode ser usada para refazimento. Portanto as definições de tipo a seguir podem ser usadas para representar essas estruturas.

```
   /* Tipo de ação */
typedef enum {ACRESCENTA, INSERE, REMOVE, MODIFICA} tAcao;

   /* Tipo de estrutura que armazena informações sobre desfazimento ou refazimento */
typedef struct {
           tAcao       acao;
           tItemCompra item;
           int         posicao;
        } tDesfazRefaz;
```

Na definição do tipo `tDesfazRefaz`, `tItemCompra` é o tipo de cada elemento da lista gerenciada pelo programa (i.e., `tItemCompra` é o tipo dos strings que a lista armazena).

Definindo-se o tipo `tItemPilha` no arquivo de cabeçalho do tipo `tPilhaIdx` da **Seção 8.1** como:

```
  /* Tipo de um elemento de pilha */
typedef tDesfazRefaz tItemPilha;
```

podem-se usar as mesmas funções definidas naquela seção no presente programa sem nenhuma alteração.

A seguir, será apresentada a função **main()** que gerencia uma lista de compras permitindo desfazer e refazer operações. Espera-se que os comentários inseridos nessa função sejam suficientes para o seu entendimento. Em caso de dúvidas, o leitor é instado a examinar o exemplo de execução (v. **página 351**) do programa do qual essa função faz parte.

```
int main(void)
{
   int           opcao, /* Uma opção escolhida pelo usuário */
                 pos, /* Uma posição na lista */
                 nElementos; /* Número de elementos da lista */
   tListaCompras lista; /* A lista de compras */
   tItemCompra   umItem; /* Um item da lista de compras */
   char         *removido; /* Um item removido da lista */
   tPilhaIdx     pDesfaz, /* Pilha que armazena informações sobre como desfazer */
                 pRefaz; /* Pilha que armazena informações sobre como refazer */
```

```
tDesfazRefaz  desfazRefaz; /* Um item da pilha que contém informações */
                           /* sobre como desfazer ou refazer          */
char         *opcoes; /* Aponta para o string contendo as   */
                      /* opções que o usuário pode escolher */

IniciaListaIdx(&lista); /* Inicia a lista de compras  */
CriaPilhaIdx(&pDesfaz); /* Inicia a pilha de desfazimento */
CriaPilhaIdx(&pRefaz);  /* Inicia a pilha de refazimento  */

   /* Apresenta o programa ao usuário e explica seu funcionamento */
printf( "\n\t>>> Este programa gerencia uma lista de"
        "\n\t>>> compras e permite desfazer e refazer\n" );

while (1) {
   ApresentaLista(&lista); /* Apresenta a lista */

      /* Alerta o usuário quando a lista está cheia */
   if (EstaCheiaListaIdx(&lista))
      printf("\n*** A lista esta' cheia ***\n");

      /* Apresenta o menu e obtém as opções disponíveis */
   opcoes = ApresentaMenu(&lista, &pDesfaz, &pRefaz);

      /* Lê a opção do usuário (o prompt já foi apresentado ) */
   opcao = LeOpcao(opcoes);

      /* É melhor tratar a opção de encerramento aqui */
   if (opcao == 'E' || opcao == 'e')
      break; /* Usuário quer encerrar o programa */

      /* Processa as demais opções */
   switch (opcao) {
      case 'A': /* Acrescenta */
      case 'a':
         ASSEGURA(!EstaCheiaListaIdx(&lista), "Lista cheia");

            /* Acrescenta um novo intem na lista */
         printf("\nDigite o nome do item >>> ");
         LeString( (char *) umItem, MAX_ITEM );
         AcrescentaListaIdx(&lista, umItem);

            /* Preenche a estrutura com informações sobre desfazimento */
         desfazRefaz.acao = ACRESCENTA;
         strcpy((char *) desfazRefaz.item, (char *) umItem);
         desfazRefaz.posicao = ComprimentoListaIdx(&lista) - 1;

            /* EmpilhaIdx as informações sobre desfazimento */
         EmpilhaIdx(&desfazRefaz, &pDesfaz);

            /* Quando uma ação é executada deixa de haver opção de      */
            /* refazimento. Portanto esvazia-se a pilha de refazimento. */
         EsvaziaPilha(&pRefaz);
         break;
      case 'I': /* Insere */
      case 'i':
         ASSEGURA(!EstaCheiaListaIdx(&lista), "Lista cheia");

            /*** Insere um elemento na lista ***/
         nElementos = ComprimentoListaIdx(&lista);

         if (nElementos > 1) {
            printf( "\nInforme a posicao entre 1 e %d >>> ", nElementos );
            pos = LeNaturalPositivo();
```

```
            } else {
               pos = 1;
            }

            if (pos > nElementos) {
               printf( "\n>>> A posicao deveria ser menor "
                       "do que %d. Nao havera' insercao.\n", nElementos + 1 );
            } else {
                  /* A correção a seguir é necessária porque usuário normal conta */
                  /* a partir de 1, mas programador de C conta a partir de zero   */
               --pos;

               printf("\nDigite o nome do item >>> ");
               LeString( (char *) umItem, MAX_ITEM );

               InsereListaIdx(&lista, umItem, pos);

                  /* Preenche a estrutura com informações sobre desfazimento */
               desfazRefaz.acao = INSERE;
               strcpy((char *) desfazRefaz.item, (char *) umItem);
               desfazRefaz.posicao = pos;

                  /* EmpilhaIdx as informações sobre desfazimento */
               EmpilhaIdx(&desfazRefaz, &pDesfaz);

                  /* Quando uma ação é executada deixa de haver opção de      */
                  /* refazimento. Portanto esvazia-se a pilha de refazimento. */
               EsvaziaPilha(&pRefaz);
            }
            break;
         case 'R': /* Remove */
         case 'r':
            ASSEGURA(!EstaVaziaListaIdx(&lista), "Lista vazia");

               /*** Remove um elemento da lista ***/

            nElementos = ComprimentoListaIdx(&lista);
            printf( "\nInforme a posicao entre 1 e %d >>> ", nElementos );
            pos = LeNaturalPositivo();

            if (pos > nElementos) {
               printf( "\n>>> A posicao deveria ser menor "
                       "do que %d. Nao havera' remocao.\n", nElementos + 1 );
            } else {
                  /* A correção a seguir é necessária porque usuário ormal conta */
                  /* na partir de 1, mas programador de conta a partir de zero   */
               --pos;

               removido = RemoveListaIdx(&lista, pos);

               printf( "\n>>> \"%s\" foi removido da lista\n", removido );

                  /* Preenche a estrutura com informações sobre desfazimento */
               desfazRefaz.acao = REMOVE;
               strcpy( (char *) desfazRefaz.item, (char *) removido );
               desfazRefaz.posicao = pos;

                  /* EmpilhaIdx as informações sobre desfazimento */
               EmpilhaIdx(&desfazRefaz, &pDesfaz);

               EsvaziaPilha(&pRefaz); /* Esvazia a pilha de refazimento */
            }
            break;
```

```
        case 'M': /* Modifica */
        case 'm':
           ASSEGURA(!EstaVaziaListaIdx(&lista), "Lista vazia");

              /* Modifica um elemento da lista */

           nElementos = ComprimentoListaIdx(&lista);
           printf( "\nInforme a posicao entre 1 e %d >>> ", nElementos );
           pos = LeNaturalPositivo();

           if (pos > nElementos) {
              printf( "\n>>> A posicao deveria ser menor "
                      "do que %d. Nao havera' modificacao.\n", nElementos + 1 );
           } else {
                 /* A correção a seguir é necessária porque usuário normal conta */
                 /* a partir de 1, mas programador de C conta a partir de zero   */
              --pos;

              printf( "\nDigite o novo nome do item: %s >>> ",
                      ConsultaElemento(&lista, pos) );
              LeString( (char *) umItem, MAX_ITEM );

              AlteraValor(&lista, umItem, pos);

                 /* Preenche a estrutura com informações sobre desfazimento */
              desfazRefaz.acao = MODIFICA;
              strcpy( (char *) desfazRefaz.item, (char *) umItem );
              desfazRefaz.posicao = pos;

                 /* EmpilhaIdx as informações sobre desfazimento */
              EmpilhaIdx(&desfazRefaz, &pDesfaz);

                 /* Esvazia a pilha de refazimento */
              EsvaziaPilha(&pRefaz);
           }
           break;
        case 'D': /* Desfaz */
        case 'd':
           ASSEGURA(!PilhaIdxVazia(&pDesfaz), "Pilha vazia");

           Desfaz(&lista, &pDesfaz, &pRefaz);
           break;
        case 'F': /* Refaz */
        case 'f':
           ASSEGURA(!PilhaIdxVazia(&pRefaz), "Pilha vazia");

           Refaz(&lista, &pRefaz, &pDesfaz);
           break;
        default:
           printf("\a\n\t>>> Opcao invalida");
           break;
     }
  }

  printf( "\n\t>>> Obrigado por usar este programa. Boas compras!\n" );

  return 0;
}
```

A função `ApresentaMenu()`, exibida a seguir, é uma versão incrementada da função de idêntico nome apresentada na **Seção 7.6.1**. Essa nova versão continua apresentando um menu sensível ao contexto, mas, agora, ela considera as opções de desfazimento e refazimento. Para tal, essa função recebe dois parâmetros adicionais

que representam as pilhas contendo informações sobre essas novas opções. Em caso de dúvidas, sugere-se que o leitor estude novamente a referida seção.

```
char *ApresentaMenu(const tLista *lista, const tPilhaIdx *desfaz, const tPilhaIdx *refaz)
{
   tDesfazRefaz *desfazer;
   static char   opcoes[MAX_OPCOES + 1];

      /* O array opcoes[] armazenará um string cujos caracteres representam  */
      /* as opções que o usuário poderá escolher. Aqui, ele recebe as opções */
      /* de encerramento que são sempre disponíveis. Isso não pode ser feito */
      /* como iniciação pois o array tem duração fixa                        */
   strcpy(opcoes, "Ee");

      /* Apresenta o cabeçalho do menu */
   printf( "\n\n\t*** Opcoes ***" );

      /*************************************************************/
      /* Apresenta o menu de acordo com o estado da lista e das pilhas */
      /*************************************************************/

   if (!EstaCheiaListaIdx(lista)) {
         /* Se a lista não está cheia, o usuário pode acrescentar mais itens */
      printf( "\n\t[A]crescenta" );
      strcat(opcoes, "Aa");

      if (ComprimentoListaIdx(lista) > 1) {
            /* Se a lista tiver mais de um elemento, */
            /* o usuário pode inserir um elemento    */
         printf( "\n\t[I]nsere" );
         strcat(opcoes, "Ii");
      }
   }

     /* Se a lista não estiver vazia, o usuário pode remover e modificar um elemento */
   if (!EstaVaziaListaIdx(lista)) {
      printf( "\n\t[R]emove" );
      printf( "\n\t[M]odifica" );

      strcat(opcoes, "RrMm");
   }

      /* Se a pilha de desfaz não estiver vazia, */
      /* o usuário pode desfazer a última ação   */
   if (!PilhaIdxVazia(desfaz)) {
      printf( "\n\t[D]esfaz " );
      strcat(opcoes, "Dd");
         /* Obtém o elemento do topo da pilha 'desfaz' */
      desfazer = ElementoTopoIdx(desfaz);

         /* Especifica a ação que pode ser desfeita */
      switch (desfazer->acao) {
         case ACRESCENTA:
            printf("acrescenta item");
            break;
         case INSERE:
            printf("insere item");
            break;
         case REMOVE:
            printf("remove item");
            break;
```

```
         case MODIFICA:
            printf("modifica item");
            break;
         default: /* O programa não deve chegar até aqui */
            ASSEGURA(0, "Erro que nao deveria ocorrer");
            break;
      }
   }

      /* Se a pilha de refaz não estiver vazia, o usuário pode refazer a última ação */
   if (!PilhaIdxVazia(refaz)) {
      printf( "\n\tRe[F]az " );
      strcat(opcoes, "Ff");

         /* Obtém o elemento do topo da pilha 'refaz' */
      desfazer = ElementoTopoIdx(refaz);

         /* Especifica a ação que pode ser desfeita */
      switch (desfazer->acao) {
         case ACRESCENTA:
            printf("acrescenta item");
            break;
         case INSERE:
            printf("insere item");
            break;
         case REMOVE:
            printf("remove item");
            break;
         case MODIFICA:
            printf("modifica item");
            break;
         default: /* O programa não deve chegar até aqui */
            ASSEGURA(0, "Erro que nao deveria ocorrer");
            break;
      }
   }

      /* Apresenta a última opção e o prompt */
   printf( "\n\t[E]ncerra o programa\n\nEscolha sua opcao >>> " );

   return opcoes;
}
```

A função `Desfaz()`, definida abaixo, implementa o desfazimento de uma ação efetuada sobre a lista gerenciada pelo programa. Essa função possui três parâmetros:

- `lista` (saída) — a lista sobre a qual a ação será desfeita
- `desfaz` (entrada e saída) — pilha contendo informações sobre como desfazer
- `refaz` (saída) — pilha contendo informações sobre como refazer

```
void Desfaz(tLista *lista, tPilhaIdx *desfaz, tPilhaIdx *refaz)
{
   tDesfazRefaz desfazer;

      /* Se a pilha 'desfaz' estiver vazia, esta função não deve ser chamada */
   ASSEGURA(!PilhaIdxVazia(desfaz), "Nao ha o que desfazer");

      /* DesempilhaIdx as informações sobre a operação a ser desfeita */
   desfazer = DesempilhaIdx(desfaz);
```

```
      /* EmpilhaIdx as informações sobre a operação a ser desfeita na pilha 'refaz' */
   EmpilhaIdx(&desfazer, refaz);

      /* Desfaz a última operação */
   switch (desfazer.acao) {
      case ACRESCENTA: /* Remove o elemento que havia   */
      case INSERE:     /* sido acrescentado ou inserido */
         RemoveListaIdx(lista, desfazer.posicao);
         break;
      case REMOVE:
            /* Insere o elemento que havia sido removido */
         InsereListaIdx( lista, desfazer.item, desfazer.posicao );
         break;
      case MODIFICA:
            /* Retorna o valor anterior do item */
         AlteraValor(lista, desfazer.item, desfazer.posicao);
         break;
      default: /* O programa não deve chegar até aqui */
         ASSEGURA(0, "Erro que nao deveria ocorrer");
         break;
   }
}
```

É importante notar que a função `Desfaz()` inclui como parâmetro não apenas a pilha que armazena informações sobre desfazimento, o que é natural, como também a pilha que contém informações sobre refazimento, o que não aparenta fazer sentido. Ocorre, porém, que essa última pilha se faz necessária porque essa função empilha as informações sobre o que ela desfaz na pilha de refazimento.

O raciocínio que norteia a função `Refaz()` definida a seguir é semelhante àquele discutido para a função `Desfaz()` apresentada acima.

```
void Refaz(tLista *lista, tPilhaIdx *refaz, tPilhaIdx *desfaz)
{
   tDesfazRefaz refazer;

      /* Se a pilha 'refaz' estiver vazia, esta função não deve ser chamada */
   ASSEGURA(!PilhaIdxVazia(refaz), "Nao ha o que refazer");

      /* DesempilhaIdx as informações sobre a operação a ser refeita */
   refazer = DesempilhaIdx(refaz);

      /* EmpilhaIdx as informações sobre a operação a ser refeita na pilha 'desfaz' */
   EmpilhaIdx(&refazer, desfaz);

      /* Refaz a última operação desfeita */
   switch (refazer.acao) {
      case ACRESCENTA:
         AcrescentaListaIdx(lista, refazer.item);
         break;
      case INSERE:
         InsereListaIdx(lista, refazer.item, refazer.posicao);
         break;
      case REMOVE:
         (void) RemoveListaIdx(lista, refazer.posicao);
         break;
      case MODIFICA:
         AlteraValor(lista, refazer.item, refazer.posicao);
         break;
      default: /* O programa não deve chegar até aqui */
```

```
            ASSEGURA(0, "Erro que nao deveria ocorrer");
         break;
      }
}
```

A função **EsvaziaPilha()**, que será definida a seguir, esvazia uma pilha de qualquer natureza, visto que sua implementação independe do tipo de conteúdo armazenado na pilha que ela recebe como parâmetro. Essa função é chamada pela função **main()** exposta acima para esvaziar pilhas de refazimento quando essa operação deixa de fazer sentido. Por exemplo, em programas mais simples que implementam *undo*/*redo*, essa última opção só fica disponível quando o usuário desfaz alguma ação, mas isso não ocorre quando ele *faz* uma ação.

```
void EsvaziaPilha(tPilhaIdx *p)
{
   while (!PilhaIdxVazia(p))
      (void) DesempilhaIdx(p);
}
```

Exemplo de execução do programa:

```
       >>> Este programa gerencia uma lista de
        >>> compras e permite desfazer e refazer

        *** Lista vazia ***

        *** Opcoes ***
        [A]crescenta
        [E]ncerra o programa

Escolha sua opcao >>> a

Digite o nome do item >>> Arroz

        *** Lista ***

            1. Arroz
        *** Opcoes ***
        [A]crescenta
        [R]emove
        [M]odifica
        [D]esfaz
        [E]ncerra o programa

Escolha sua opcao >>> a

Digite o nome do item >>> Acucar

        *** Lista ***

            1. Arroz
            2. Acucar

        *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [D]esfaz
        [E]ncerra o programa
Escolha sua opcao >>> i

Informe a posicao entre 1 e 2 >>> 2
```

```
Digite o nome do item >>> Feijao
        *** Lista ***

            1. Arroz
            2. Feijao
            3. Acucar

        *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [D]esfaz
        [E]ncerra o programa

Escolha sua opcao >>> d
        *** Lista ***

            1. Arroz
            2. Acucar

        *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [D]esfaz
        Re[F]az
        [E]ncerra o programa

Escolha sua opcao >>> f
        *** Lista ***

            1. Arroz
            2. Feijao
            3. Acucar

        *** Opcoes ***
        [A]crescenta
        [I]nsere
        [R]emove
        [M]odifica
        [D]esfaz
        [E]ncerra o programa

Escolha sua opcao >>> e
```

8.5.5 Simulação de Fila de Banco

Problema: Escreva um programa em C que simula o funcionamento de uma fila de banco (ou qualquer outra fila civilizada na qual as pessoas sejam atendidas *democraticamente*).

Solução: Cada cliente a ser armazenado na fila será representado pelo instante no qual ele é enfileirado, de modo que a simulação será implementada da seguinte maneira:

1. O programa solicita ao usuário um valor a ser utilizado como tempo de duração da simulação. Por simplicidade, esse valor será considerado inteiro.
2. O laço principal do programa executa as seguintes operações:
   2.1 Exibe na tela o estado da fila. Essa exibição usa asteriscos para representar os clientes ora na fila.

2.2 Se a fila não estiver vazia, remove-se o cliente que se encontra na frente da fila e calcula-se o tempo de espera desse cliente, que é dado pelo instante corrente menos o instante no qual ele foi enfileirado. Então, acrescenta-se o tempo de espera do cliente ao tempo total de espera e atualiza-se o valor do maior tempo de espera

2.3 Sorteia-se o número de clientes que acabam de chegar ao banco e coloca-os na fila. O valor sorteado varia de zero a três (esses valores são arbitrários).

2.4 O laço encerra quando se esgota o tempo introduzido pelo usuário.

3. O restante do programa é dedicado à apresentação dos resultados da simulação e é relativamente trivial, como se verá a seguir.

É importante notar que o banco sendo simulado não atende clientes que se encontrem na fila após o encerramento do expediente; i.e., após o intervalo de tempo estipulado pelo usuário para a simulação.

A função **main**(), definida a seguir, realiza a simulação discutida acima.

```
int main(void)
{
   tFilaIdx fila; /* Fila de clientes contendo o instante */
                  /* em que cada cliente chegou na fila   */
   int      duracao, /* Duração da simulação (em minutos) */
            minuto,          /* Minuto corrente */
            instanteChegada, /* Instante em que o cliente entrou na fila */
            tempoEspera, /* Tempo de espera de um cliente na fila */
            totalAtendidos = 0, /* Total clientes atendidos */
            esperaTotal = 0, /* Tempo de espera total dos clientes */
            maiorEspera = 0,   /* Tempo de espera mais longo */
            clientesChegando = 0, /* Número de clientes que  */
                                  /*acabam de chegar na fila */
            naoAtendidos = 0; /* Número de clientes que não foram atendidos */
                              /* porque a simulação acabou antes            */

   srand(time(NULL)); /* Inicia o gerador de números aleatórios */

      /* Lê o tempo de duração da simulação */
   printf("\n>>> Digite a duracao da simulacao (inteiro): ");
   duracao = LeInteiro();

   if (duracao <= 0) {
      printf("\nValor de duracao invalido. Bye.\n");
      return 1;
   }

   CriaFila(&fila); /* Inicia a fila */

      /* Executa a simulação durante o tempo estipulado */
   for (minuto = 1; minuto <= duracao; ++minuto) {
         /* Exibe o estado da fila */
      ExibeFila(fila);

       /* Se a fila não estiver vazia, atende mais um cliente; i.e., remove-o da fila */
      if (!FilaIdxVazia(fila)) {
            /* Cada cliente é representado por seu instante de chegada */
         instanteChegada = DesenfileiraIdx(&fila);

         totalAtendidos++; /* Mais um cliente foi atendido */

            /* Calcula o tempo de espera deste cliente */
         tempoEspera = minuto - instanteChegada;
```

```
            /* Acrescenta o tempo de espera deste cliente ao tempo total de espera */
         esperaTotal = esperaTotal + tempoEspera;

            /* Atualiza o valor do maior tempo de espera */
         if (maiorEspera < tempoEspera)
            maiorEspera = tempoEspera;
      }

         /* Sorteia o número de clientes que acabam de chegar ao banco */
      clientesChegando = rand()%(MAX_CLIENTES-MIN_CLIENTES+1) + IN_CLIENTES;

         /* Coloca os clientes na fila */
      for ( int j = 0; j < clientesChegando ; j++ )
         EnfileiraIdx(minuto, &fila);
   }

   ExibeFila(fila); /* Exibe o estado final da fila */

      /* Apresenta o resultado da simulação */
   if (!totalAtendidos) {
      printf("\n\t>>> Nenhum cliente foi atendido\n");
   } else {
      printf( "\n\t>>> Numero de clientes atendidos: %d", totalAtendidos );
      printf( "\n\t>>> Maior tempo de espera: %d min", maiorEspera );
      printf( "\n\t>>> Tempo medio de espera: %3.2f min",
              (double) esperaTotal/totalAtendidos );
   }

      /* Conta o número de clientes que não foram atendidos */
   while (!FilaIdxVazia(fila)) {
      (void) DesenfileiraIdx(&fila);
      ++naoAtendidos;
   }

      /* Apresenta o número de clientes que não foram atendidos */
   if (naoAtendidos)
      printf( "\n\t>>> Numero de clientes nao atendidos: %d\n", naoAtendidos );
   else
      printf("\n\t>>> Todos os clientes foram atendidos\n");

   return 0;
}
```

A função `ExibeFila()`, definida a seguir, é chamada por **main()** para apresentar um diagrama na tela mostrando o estado de uma fila (v. exemplo de execução adiante).

```
void ExibeFila(tFilaIdx f)
{
   int        nElementos = NItensFilaIdx(f);
   static int primeiraChamada = 1;

   if (primeiraChamada) {
      primeiraChamada = 0;
      printf("\n\t  >>> Fila <<<\n\n");
   }

   if (!nElementos) {
      printf("\tNinguem na fila\n");
      return;
   }

   putchar('\t');
```

```
   for (int i = 0; i < nElementos; ++i)
      putchar('*');

   putchar('\n');
}
```

É interessante notar que a função `ExibeFila()` chama a função `NItensFilaIdx()`, que calcula o número de elementos de uma fila. O que há de interessante nesse fato aparentemente tão banal é que a função `NItensFilaIdx()` não implementa nenhuma das operações sobre filas discutidas na **Seção 8.2**. Pior, essas operações não permitem que essa função seja implementada de modo trivial e eficiente (embora essa implementação seja possível). Numa situação como essas, a solução ideal consiste em implementar a função em questão considerando-a como uma das operações disponíveis para o tipo. Neste caso específico, essa função pode ser implementada de modo trivial como mostrado abaixo.

```
int NItensFilaIdx(tFilaIdx f)
{
   return f.fundo - f.frente;
}
```

Exemplo de execução do programa:

```
>>> Digite a duracao da simulacao (inteiro): 15

          >>> Fila <<<

        Ninguem na fila
        ***
        *****
        ****
        *****
        ******
        ******
        *****
        ******
        *******
        *******
        ********
        *******
        ******
        ********
        **********

        >>> Numero de clientes atendidos: 14
        >>> Maior tempo de espera: 6 min
        >>> Tempo medio de espera: 4.36 min
        >>> Numero de clientes nao atendidos: 10
```

8.6 Exercícios de Revisão

Pilhas (Seção 8.1)

1. No contexto de estruturas de dados, defina (a) dicionário e (b) contêiner.
2. (a) O que significa *LIFO*? (b) Por que pilhas são chamadas *estruturas LIFO*?
3. Apresente três exemplos de usos práticos de pilhas.
4. Por que a estrutura de dados pilha é classificada como contêiner?
5. Defina as operações de empilhamento e de desempilhamento.
6. Apresente duas diferenças entre pilhas e listas.

7. Que operações sobre pilhas têm correlação para listas?

8. (a) Apresente operações sobre listas que não têm correlação para pilhas. (b) Apresente operações sobre pilhas que não têm correlação para listas.

9. Como pilhas são utilizadas na implementação de chamadas de funções?

10. Considerando que p é uma pilha e x é um elemento, uma pilha pode ser definida abstratamente por meio das seguintes regras:

Δ é uma pilha vazia	**(1)**
EmpilhaIdx(p, x) é uma pilha	**(2)**
DesempilhaIdx(EmpilhaIdx(p, x)) = p	**(3)**
Topo(EmpilhaIdx(p, x)) = x	**(4)**

Utilizando essas regras, tem-se, por exemplo, que:

Topo(DesempilhaIdx(EmpilhaIdx(EmpilhaIdx(Δ*, x), y)))* = *Topo(EmpilhaIdx(*Δ*, x))* = x

No exemplo acima, a primeira igualdade é obtida aplicando-se a regra **(3)** com p = *EmpilhaIdx(*Δ*, x)*, enquanto a segunda igualdade é decorrência da regra **(4)** com p = Δ. Baseado nas regras **(1)**, **(2)**, **(3)** e **(4)** da definição acima, verifique quais das seguintes expressões são corretas:

(a) *EmpilhaIdx(DesempilhaIdx(EmpilhaIdx(*Δ*, x)), y)* é uma pilha

(b) *Topo(EmpilhaIdx(DesempilhaIdx(EmpilhaIdx(*Δ*, x)), y))* = y

(c) *DesempilhaIdx(DesempilhaIdx(EmpilhaIdx(*Δ*, x)))* = x

11. Quais são os custos temporais das operações sobre pilha?

12. Seja `tPilhaCar` um tipo de pilha de elementos do tipo **char**. Escreva um trecho de programa em C para exibir na tela todo o conteúdo de uma pilha desse tipo, mantendo-a intacta.

13. Suponha a existência do tipo `tPilhaIdx` que define pilhas cujos elementos são do tipo `tItemPilha`. Suponha ainda a existência do seguinte trecho de programa:

```
tPilhaIdx  p;
tItemPilha item;
...
CriaPilhaIdx(&p);
```

Utilize as funções `PilhaIdxVazia()`, `ElementoTopoIdx()`, `EmpilhaIdx()` e `DesempilhaIdx()` definidas para o tipo `tPilhaIdx` para apresentar instruções em C que implementem as seguintes operações:

(a) Atribuir a `item` o valor do segundo elemento a partir do topo da pilha, deixando a pilha sem seus dois elementos mais próximos ao topo.

(b) Atribuir a `item` o valor do segundo elemento a partir do topo da pilha, deixando a pilha inalterada.

(c) Dado um inteiro positivo `n`, atribuir a `item` o enésimo elemento a partir do topo da pilha, deixando a pilha sem seus `n` elementos mais próximos ao topo.

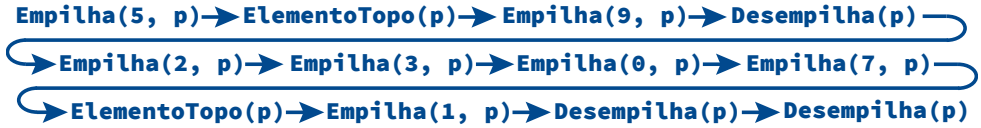
(d) Dado um inteiro positivo `n`, atribuir a `item` o enésimo elemento a partir do topo da pilha, deixando a pilha inalterada.

(e) Atribuir a `item` o valor do último elemento a partir do topo da pilha, deixando a pilha vazia.

(f) Atribuir a `item` o valor do último elemento a partir do topo da pilha, deixando a pilha inalterada.

14. Suponha que uma pilha implementada num array em C e inicialmente vazia tenha passado por 12 operações de empilhamento, cinco operações de consulta ao elemento do topo e três operações de desempilhamento. Em que índice do array se encontra o indicador de topo dessa pilha?

15. Suponha que uma pilha `p` seja do tipo `tPilhaIdx` definido na **Seção 8.1** e que o tipo de elemento `tItemPilha` seja **int**. Quais serão os valores desempilhados quando a pilha `p` passa pela seguinte sequência de operações:



16. Dentre as aplicações a seguir, quais delas são apropriadas para o uso da estrutura de dados pilha.

(a) Avaliação de expressões aritméticas
(b) Simulação de atendimento de um sistema de atendimento ao consumidor (SAC)
(c) Desafazer e refazer uma operação
(d) Processamento de dados na ordem inversa na qual eles são obtidos
(e) Processamento de dados na mesma ordem na qual eles são obtidos
(f) Manutenção de histórico de sites visitados por um navegador de internet
(g) Gerenciamento de atendimento de pacientes num consultório médico
(h) Substituição do uso de recursão por iteração
(i) Gerenciamento de uso de processador a cargo de um sistema operacional

Filas Lineares (Seção 8.2)

17. O que é uma fila?

18. Apresente duas diferenças entre filas e listas.

19. Quais são as operações que devem fazer parte do repertório de uma estrutura de dados fila?

20. Que operações sobre filas têm correlação para listas?

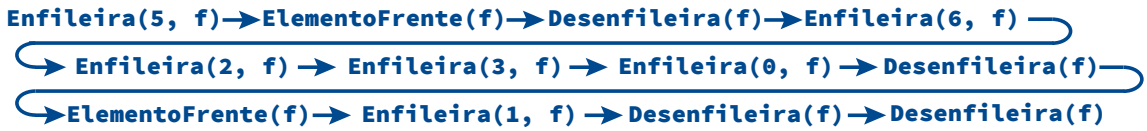
21. Apresente três exemplos de usos práticos de filas.

22. (a) Qual é significado de *FIFO*? (b) Por que filas são denominadas *estruturas FIFO*?

23. Por que a estrutura de dados fila é classificada como contêiner?

24. Qual é o custo temporal das operações sobre filas lineares?

25. Suponha que uma fila `f` seja do tipo `tFilaIdx` definido na **Seção 8.2** e que o tipo de elemento `tItemFila` seja **int**. Quais serão os valores removidos da fila quando a ela passa pela seguinte sequência de operações:



26. Suponha que uma fila linear implementada num array em C e inicialmente vazia tenha passado por 12 operações de enfileiramento, cinco operações de consulta ao elemento do frente e três operações de desenfileiramento. (a) Em que índice do array se encontra o indicador de frente dessa fila? (b) Em que índice do array se encontra o indicador de fundo dessa fila?

27. Dentre as aplicações a seguir, quais delas são apropriadas para o uso da estrutura de dados fila.

(a) Avaliação de expressões aritméticas
(b) Simulação de atendimento de um sistema de atendimento ao consumidor (SAC)
(c) Desafazer e refazer uma operação realizada por um programa
(d) Processamento de dados na ordem inversa na qual os dados são obtidos
(e) Processamento de dados na mesma ordem na qual eles são obtidos
(f) Histórico de sites visitados por um navegador de internet

(g) Gerenciamento de atendimento de pacientes num consultório médico
(h) Substituição do uso de recursão por iteração
(i) Gerenciamento do uso de processador por parte de um sistema operacional

Filas Circulares (Seção 8.3)

28. (a) O que é uma fila circular? (b) Explique o funcionamento de uma fila circular. (c) Que vantagens oferece uma fila circular com relação a uma fila linear?

29. Fila circular refere-se a abstração ou implementação? Explique sua resposta.

30. Por que se define o tamanho do array que armazena uma fila circular como `MAX_ELEMENTOS + 1` e não simplesmente `MAX_ELEMENTOS`?

31. Suponha que uma fila circular implementada num array em C com 10 elementos e inicialmente vazia tenha passado por oito operações de enfileiramento, cinco operações de consulta ao elemento do frente e três operações de desenfileiramento. (a) Em que índice do array se encontra o indicador de frente dessa fila? (b) Em que índice do array se encontra o indicador de fundo dessa fila?

32. Suponha que você tenha uma implementação de fila linear utilizando array. Se você desejar obter uma fila circular, que operações precisarão ser alteradas?

33. Explique o uso do operador **%** na implementação de filas circulares.

34. Em que diferem os custos temporais das operações sobre filas circulares dos respectivos custos de operações sobre filas lineares?

35. (a) Por que a implementação de fila circular apresentada neste capítulo não utiliza todos os elementos do array no qual os itens da fila são armazenados? (b) Qual é a alternativa para essa abordagem e por que ela é desfavorável?

Transformando Recursão em Iteração Usando Pilha (Seção 8.4)

36. É verdade que qualquer função recursiva pode ser transformada numa função iterativa? Justifique sua resposta.

37. O que é um meta-algoritmo?

38. Descreva o meta-algoritmo de transformação de recursão em iteração.

39. Que estrutura de dados é provavelmente utilizada na transformação de uma função recursiva em função iterativa equivalente? Justifique sua resposta.

Exemplos de Programação (Seção 8.5)

40. Qual é a diferença entre os conceitos de palíndromo em linguagem cotidiana e em programação?

41. Qual das duas versões de função que inverte strings introduzidos pelo usuário via teclado deve ser favorecida: a versão recursiva (apresentada na **Seção 4.8.6**) ou a versão iterativa (apresentada na **Seção 8.5.1**)? Justifique sua resposta.

42. (a) O que é HCI? (b) O que significa *desfazimento* no contexto de HCI? (c) O que significa *refazimento* em HCI?

43. (a) O que é *undo*/*redo*? (b) Que informações um programa que pretende implementar *undo*/*redo* precisa guardar? (c) Como *undo*/*redo* pode ser implementado?

44. (a) Qual é o papel desempenhado por pilhas na implementação de *undo*/*redo*? (b) Por que são necessárias duas pilhas (e não apenas uma) para implementar *undo*/*redo*?

45. Por que pilhas contendo informações sobre refazimento são esvaziadas pelo programa da **Seção 8.5.4**?

46. Virtualmente, qualquer editor de texto moderno permite desfazer ou refazer ações que podem ser desfeitas ou refeitas. Que informações um editor de texto precisa armazenar para permiti-lo desfazer ou refazer as seguintes ações:

(a) Recortar
(b) Colar
(c) Apagar uma palavra
(d) Substituir

47. (a) Explique como a função `NItensFilaIdx()` apresentada no exemplo da seção **Seção 8.5.5** poderia ser implementada por um programa-cliente do tipo `tFilaIdx` implementado na **Seção 8.2**. (b) Como essa mesma função seria implementada por um cliente do tipo `tFilaC` implementado na **Seção 8.3**?

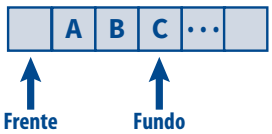
8.7 Exercícios de Programação

EP8.1 Escreva funções que implementam duas pilhas num único array. Uma tal pilha só deve ser considerada cheia quando não houver mais espaço para nenhum elemento no array.

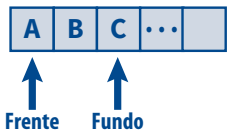
EP8.2 Escreva um programa dirigido por menu que utilize cinco filas do tipo `tFilaIdx` definido na **Seção 8.2** e ofereça as opções de acrescentar ou retirar elementos (um de cada vez) de uma das cinco filas à escolha do usuário.

EP8.3 Utilize uma variável do tipo **int** para distinguir entre fila circular vazia e fila circular cheia e reescreva as funções `AcrescentaC()` e `RetiraC()` apresentadas na **Seção 8.3**, de forma que todos os elementos do array no qual a fila é implementada sejam utilizados.

EP8.4 Na implementação de fila apresentada na **Seção 8.2**, convencionou-se que o campo `frente` indicaria uma posição anterior ao elemento da frente da fila, enquanto o campo `fundo` indicaria posição do elemento do fundo da fila. Esquematicamente, essa situação é apresentada na figura abaixo:



No entanto, outra convenção poderia ter sido adotada. Considere a convenção na qual `frente` indica a posição do elemento da frente da fila e `fundo` indica a posição do elemento do fundo da fila, ou, esquematicamente:



Utilize a convenção mostrada na última figura para implementar uma fila linear em um array unidimensional com `MAX` elementos do tipo **int**, em que `MAX` é uma constante simbólica previamente definida.

(a) Como uma fila desse tipo é iniciada?
(b) Qual é a expressão condicional que denota fila vazia?
(c) Escreva uma função em C que implemente a operação de acréscimo de um elemento fila. A fila só deve ser considerada cheia se o array estiver repleto.
(d) Escreva uma função em C que implemente a operação de retirada de um elemento da fila.

EP8.5 Uma fila contendo valores do tipo **int** pode ser implementada a partir da seguinte definição de tipo:

```
typedef int tFila[N + 2];
```

Sendo `f` uma variável do tipo `tFila` acima, tem-se que o elemento `f[0]` é utilizado para representar a frente da fila, `f[1]` é utilizado para representar o fundo da fila e os demais elementos do array (`f[2]`, ..., `f[N + 1]`) são utilizados para armazenar os elementos da fila. Adotando essa representação para filas, escreva funções em C para:

(a) Inicializar uma fila
(b) Verificar se uma fila está vazia
(c) Acrescentar um elemento numa fila
(d) Retirar um elemento de uma fila

EP8.6 Uma **fila dupla** é um tipo de fila que admite acréscimos e remoções em suas duas extremidades. Represente uma dupla fila em um array unidimensional e escreva, em C, as funções `AcrescentaEsquerda()`, `AcrescentaDireita()`, `RetiraEsquerda()` e `RetiraDireita()`, que acrescentam elementos à esquerda e à direita, e retiram elementos à esquerda e à direita, respectivamente.

EP8.7 HTML é uma linguagem utilizada na construção de páginas na Web que podem ser interpretadas por navegadores. Todo documento HTML possui etiquetas (*tags*, em inglês), que são palavras-chave entre parênteses angulares (i.e., `<` e `>`) e que constituem os comandos de formatação da linguagem. Por exemplo, `<p>` é a etiqueta de abertura que define um parágrafo e `</p>` é a etiqueta correspondente de fechamento. Alguns comandos de formatação são compostos, como, por exemplo, `<a href="http://www.ulysseso.com/ed1/index.htm">site</a>`. Escreva um programa em C que lê um arquivo HTML e verifica se as etiquetas casam. Para facilitar, suponha que o arquivo não contém comandos compostos. [**Sugestão:** Utilize como modelo o exemplo apresentado na **Seção 8.5.2**, mas diferentemente do que ocorre naquele exemplo, agora, a pilha deve armazenar strings em vez de caracteres.]