capítulo 9

Filas

> *A fila anda.*
>
> Autor desconhecido

Quem nunca se deparou com essa frase, principalmente depois de uma desilusão amorosa? Sua função básica é nos deixar com a ideia de que algo se foi e outra coisa está chegando. Neste capítulo, veremos claramente que a fila anda, e as informações também.

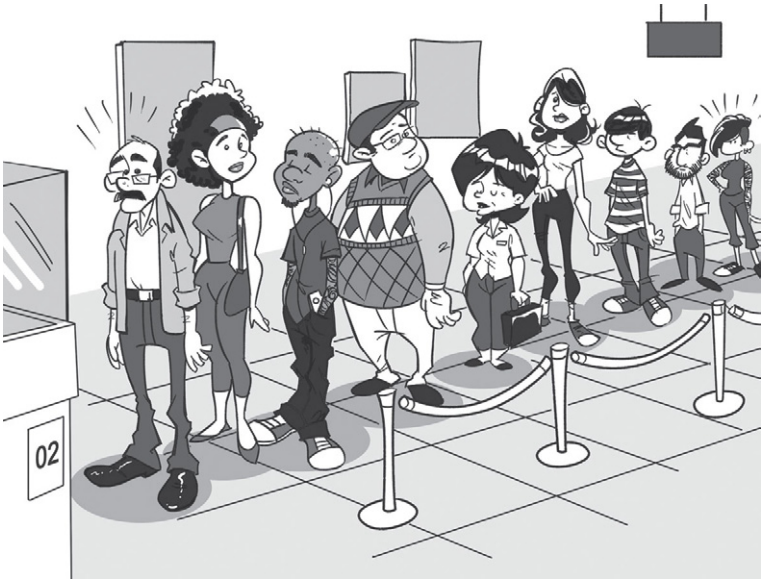
OBJETIVOS DO CAPÍTULO

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

- identificar situações e maneiras de utilizar filas;
- simular situações do mundo real em um ambiente virtual;
- entender como vários recursos computacionais utilizam-se das filas para controlar a chegada e o consumo de dados.

Para começar

Como sabemos, hoje a população mundial ultrapassa os 7,2 bilhões de habitantes e, se não mantivéssemos um mínimo de respeito, ética e ordem, nossa convivência não seria possível, muito menos teríamos a sociedade que conhecemos, com suas virtudes e problemas.



Nesse cenário populacional e social que naturalmente vivenciamos, temos inúmeras regras e costumes que nos fazem conviver com harmonia. Certamente, uma das principais regras é sermos justos uns com os outros. Imagine alguém sendo injusto com você. É agradável? Imagino que não.

Imagine que você necessite de um serviço ou recurso que não esteja numericamente à disposição de todos ao mesmo tempo, mas tenha tempo de utilização limitado, como um caixa de banco, por exemplo. Naturalmente, esperamos o término da utilização por outra pessoa para dar início à nossa. Agora pondere: por que você é o próximo a utilizar esse serviço? Não seria a vez de outra pessoa? Pois então: existe uma ordem predefinida, que todos nós sabemos bem como funciona, ou seja, quando não há uma regra determinada para a utilização de um serviço utilizamos a ordem de chegada. Mas por que temos essa regra? Sim, foi isso mesmo que você pensou: para sermos justos uns com os outros. Você teria um sentimento de injustiça se outras pessoas, que chegaram depois de você, tivessem acesso ao serviço antes.

Essa regra força as pessoas a se enfileirar para ter acesso ao serviço, pois é intrínseco entender que quem chegou antes terá acesso ao serviço primeiro e, obviamente, quem chegou depois terá acesso depois, portanto, não seria justo que acessasse o serviço antes de você, ou seria? É claro que não.

Gostaríamos de exemplificar utilizando fatos corriqueiros com que, provavelmente, a grande maioria de nós se depara. Será que você poderia nos ajudar? Temos certeza de que sim, vamos lá!

Imagine que você e seus colegas de sala de aula saíram de uma maravilhosa e empolgante aula de Estrutura de Dados e estão com muita fome – afinal, é hora do almoço! Entretanto, como o professor se empolgou em responder às dúvidas geradas pelos exercícios dados no final da aula, já é um pouco mais tarde do que de costume e parece que todas as outras turmas resolveram chegar minutos antes de vocês à praça de alimentação. Resultado: todos os restaurantes estão cheios. E então, qual é a sua primeira reação? Imaginamos que seria filtrar os restaurantes de sua preferência; depois, analisar o tamanho da fila, mesmo porque isso lhe mostraria quantas pessoas seriam atendidas antes de você, já que, em filas, o primeiro a chegar é o primeiro a ser atendido, correto?

Atenção:

Nas filas, o primeiro a chegar é o primeiro a ser atendido. No mundo virtual não há como ter penetras, a não ser que você mesmo implemente tal funcionalidade.

Assim como na vida real, podemos ter filas com diferentes prioridades de atendimento, assim como temos filas para idosos e deficientes.

Já temos todos os conhecimentos a respeito de como funciona uma fila. Agora precisamos fazer com que o conceito do mundo real seja traduzido para o mundo virtual. É isso o que faremos neste capítulo.

Papo técnico:

Temos vários recursos computacionais que são compartilhados, como processador e *sockets*, e usamos os conceitos de fila para controlar sua utilização.

Conhecendo a teoria para programar

Acabamos de perceber que temos todos os conceitos necessários para entender uma fila, mas, para traduzi-la de forma adequada para o mundo virtual, precisamos identificar detalhes pequenos porém de extrema importância para uma perfeita implementação, como o início e o fim da fila, além de perceber que, no mundo real, quem entra na fila é um ser humano, ao passo que no mundo virtual dizemos, genericamente, que "colocamos informações na fila".

Papo técnico:

A "informação colocada na fila" pode ser qualquer tipo primitivo disponível ou abstrato criado no seu programa.

Precisamos controlar adequadamente as extremidades das filas, pois será nelas que faremos as manipulações. Imagine que você está chegando à fila do restaurante para comer. Onde você se posiciona? No fim da fila, certo? De outro lado, quando você estiver na iminência de ser atendido, estará no início da fila.

Conceito:

Toda informação que chega à fila é adicionada no FIM; toda informação a ser consumida pelo recurso é retirada do INÍCIO da fila. Lembre-se: você SEMPRE entra no fim da fila.

A Figura 9.1a ilustra uma fila com três elementos, A, B e C, em que o elemento A é o início da fila e o elemento C é o fim. Nesse cenário, caso venhamos a tirar um elemento da fila, de acordo com o que já discutimos, o que estiver no início é o que deverá ser removido, resultando na Figura 9.1b.

E se solicitássemos a você que incluísse um novo elemento, onde você o colocaria? Isso mesmo: você o colocaria no fim da fila, por isso apresentamos a sua solução na Figura 9.1c. Neste momento, deve estar bem claro que *B* seria o próximo elemento a ser removido, depois *C* e, posteriormente, *D*.

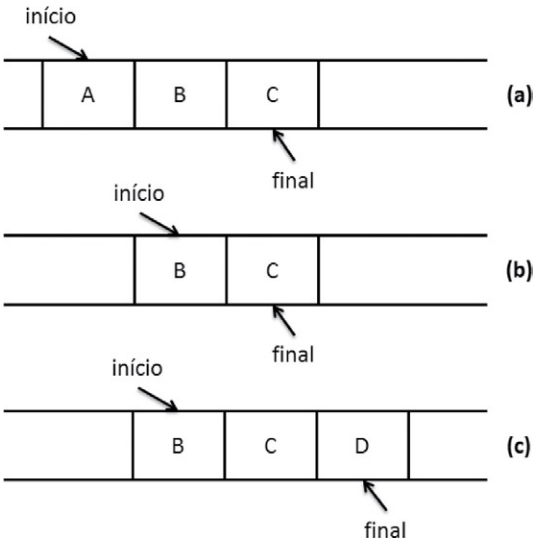


FIGURA 9.1: Uma fila.

Papo técnico:

As filas são comumente chamadas de *filas fifo* (*first-in, first-out* – o primeiro a entrar é o primeiro a sair).

Agora falemos um pouco sobre como esses conceitos de *fila* podem ser usados em programação. Depois de ter estudado, em capítulos anteriores, vários conceitos e técnicas interessantes de guardar e manipular informação, como você imagina que possamos implementar uma fila? Existem duas formas principais. Será que você imaginou as duas? Esperamos que sim!

Se os conceitos de *lista ligada* apresentados nos capítulos anteriores ainda estiverem frescos em sua mente, provavelmente você imaginou implementar uma fila usando os nós da lista ligada. Isso mesmo, essa é uma das formas. A outra é mais simples e exige um conceito menos avançado de programação, que é a utilização de vetor.

Independentemente de como a implementação será feita, para manipular uma fila teremos duas funcionalidades características, que, posteriormente, poderão ser utilizadas com a implementação com vetor e a lista ligada.

Retomando o exemplo da fila do restaurante da praça de alimentação, que funcionalidades você enxerga na manipulação dessa fila? É exatamente isso que você pensou**:** *entrar na fila* e *sair da fila*.

Para entrar na fila, você deve se posicionar imediatamente atrás da última pessoa que está nela e, nesse instante, se tornará o último da fila, posição que será procurada pelo próximo a entrar no fim da fila. De outro lado, se você for o primeiro da fila, quando o caixa for liberado e chamar o próximo cliente, você sairá da fila e será atendido. Portanto, a pessoa que estiver atrás de você se tornará, naquele momento, o primeiro da fila, sendo a próxima a ser chamada (veja o exemplo dessa dinâmica na Figura 9.2). Isso também acontecerá em programação. Como é que você pensa em fazer isso? Pense um pouco antes de continuar lendo este texto.

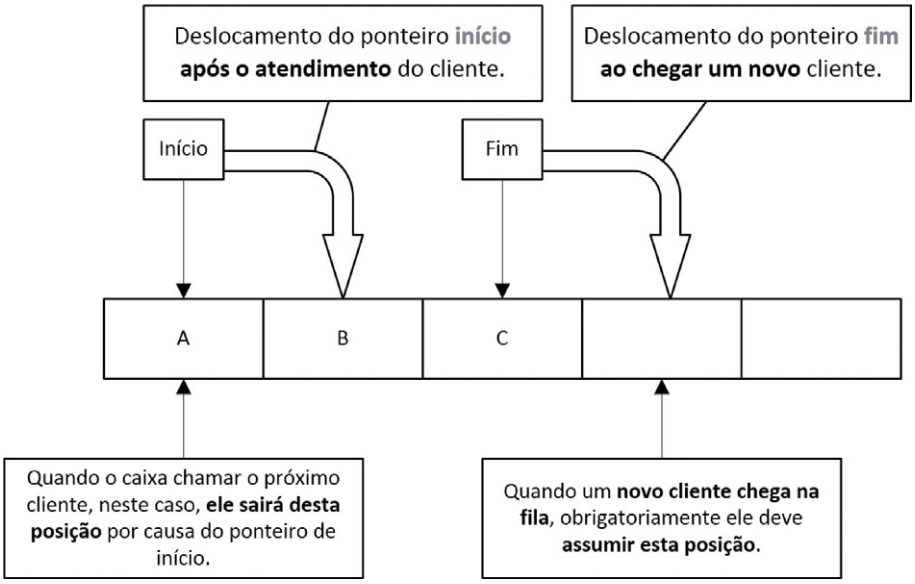


FIGURA 9.2: Exemplificação gráfica da entrada e da saída de um cliente em uma fila.

Levando em consideração o que foi descrito no parágrafo anterior, fica claro que seria conveniente elaborar duas funções para manipular a pilha: uma para inserir nova informação e outra para consumir uma informação já presente nela.Chamaremos essas funções de *inserir* e *consumir*, respectivamente.

Nessa implementação usaremos um vetor como armazenamento das informações em forma de fila. Para isso, precisamos ter alguns cuidados e definições:

- estipular um tamanho para o vetor (quantidade máxima de informações guardadas nele);
- verificar se não chegamos ao final do vetor, ou seja, se não o *estouramos*;
- ter duas variáveis *inicio* e *fim* para controlar as posições do vetor onde estão essas posições;

- a posição que a variável *inicio* aponta tem a informação para ser consumida, assim como a variável *fim* aponta para a última posição que tem informação armazenada.

Para essa implementação, o que você acha de definir como será a nossa fila? Então, vamos lá!

Primeiramente, é necessário definir o tamanho da fila – no caso deste exemplo, a quantidade de informações que podem passar pela fila não significa, necessariamente, que seja o tamanho máximo que ela pode assumir durante a execução. Ela poderá ter esse tamanho única e exclusivamente se não houver consumo de informação; quando chegarem as informações nessa quantidade, a fila ficará cheia.

```
#define TAMANHO 100
```

Com essa definição podemos criar uma estrutura de dados abstrata, que representará nossa fila. Nela teremos a fila, que nada mais é que um vetor de números inteiros e duas variáveis inteiras, tendo como valor armazenado o índice do vetor que representa a posição onde estão o início e o fim da fila.

Depois dessa explicação, será que você conseguiria montar sua própria estrutura? Temos certeza de que sim. Só para que você confira e nós possamos continuar a explicação, faremos a definição da nossa sugestão de estrutura nas próximas linhas. Além disso, daremos, na Figura 9.3, uma representação gráfica dessa estrutura. Observe:

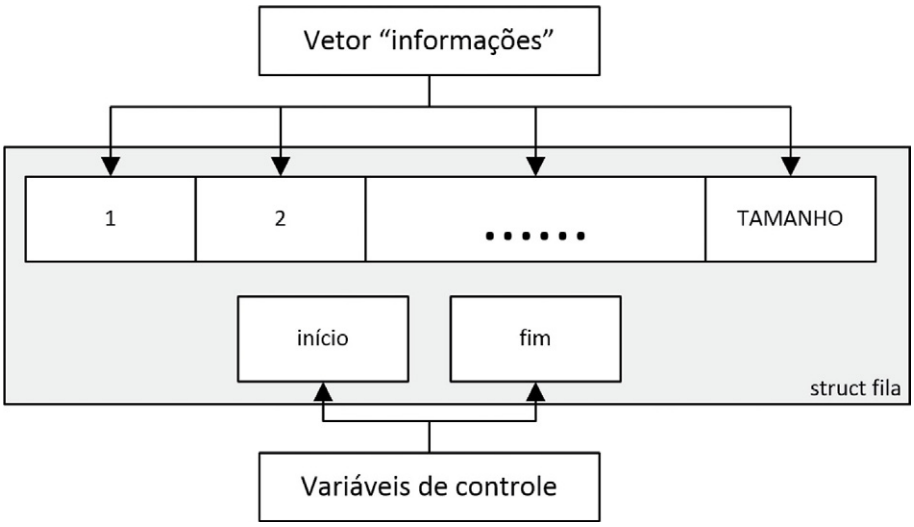


FIGURA 9.3: Exemplificação gráfica da estrutura definida para a fila.

```
#define TAMANHO 100 //definição de uma constante para o tamanho do vetor
struct fila {                          // criando uma estrutura
       int informações[TAMANHO];   // vetor que controla a fila
       int inicio, fim;        // variáveis que controlam as posições de início e fim da fila dentro do vetor
};
```

Agora que temos nossa fila definida, o que podemos fazer? Vamos elaborar as funções *inserir* e *consumir* uma informação? Então, vamos lá. Vamos começar com a função *inserir*.

Vale ressaltar que a função para inserir um novo elemento recebe o endereço de memória onde se encontra a fila definida dentro da função que a chamou, nesse caso, provavelmente a função principal; o outro parâmetro é o valor com que faremos a inserção dentro da fila.

```
void inserir(struct fila *f, int valor) {
```

Para *inserir* adequadamente nova informação na fila, temos de verificar se existe a próxima posição que usaremos com uma informação nova. Vale lembrar que a variável que guarda a posição final está com a posição do vetor onde tivemos a última inserção de valor, ou seja, a próxima posição do vetor é que deve existir para receber a próxima informação. Lembre-se do exemplo do restaurante: ao chegar à fila, você procura a última pessoa e se acomoda imediatamente atrás dela; portanto, no vetor, a variável guarda a última posição preenchida, e a próxima informação vem na posição posterior, que deve existir, por isso a verificação. Observe, na Figura 9.4, o posicionamento do ponteiro *fim*, que está na última posição possível, para inserir nova informação. Essa verificação é uma simples condicional. Observe:

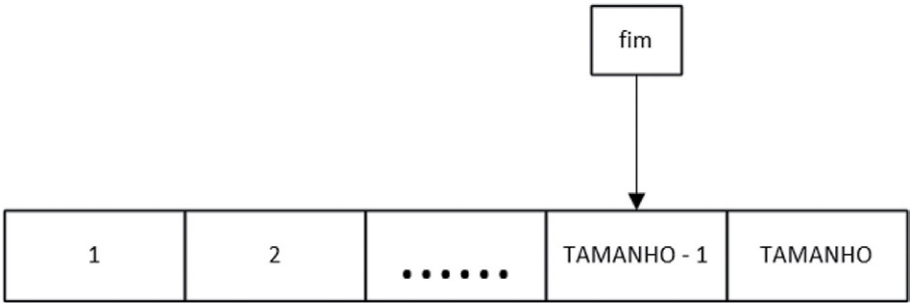


FIGURA 9.4: Posição da última possibilidade de inserir novo elemento.

```
if((f->fim + 1) < TAMANHO) {
```

Para fazer a inserção da informação, devemos apenas atualizar a posição que a receberá e, simplesmente, atribuí-la. Isso é feito com as próximas duas linhas de código e demonstrada na Figura 9.5:

```
        f->fim++;  // atualiza a posição final para inserir um novo valor
        f->informações[f->fim] = valor;  // insere o valor na posição
```

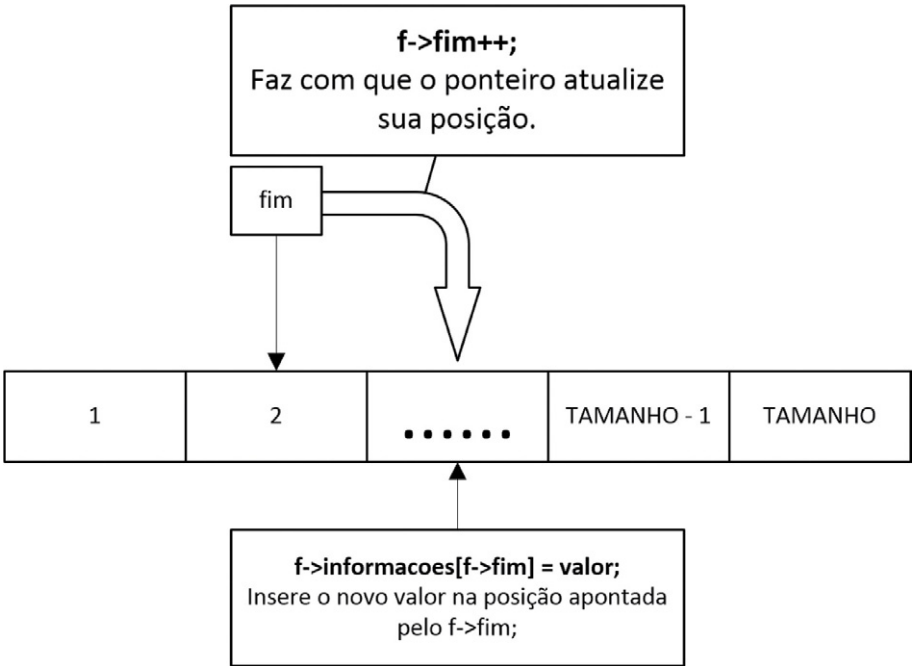


FIGURA 9.5: Dinâmica de uma inserção na fila.

A seguir, apresentamos a sugestão completa do código para essa função.

Código 9.1

```
// Função para inserir um novo elemento na fila
void inserir(struct fila *f, int valor) { // recebe o endereço da fila e o valor que será colocado
        if((f->fim + 1) < TAMANHO) // verifica se não chegamos ao final da fila
        {
                f->fim++; // atualiza a posição final para inserir um novo valor
                f->informações[f->fim] = valor; // insere o valor na posição
                printf(">>> Inserido >>>: %d\n\n",valor); // imprimi na tela para controle
        }
        else
                printf("Estouro de fila\n\n"); // avisa que houve estouro de fila
}
```

O passo seguinte é apresentar o código proposto para a função *consumir*. Vale ressaltar que ela é um pouco mais complexa, por causa de suas verificações, porém, repare que o conceito principal é o mesmo que já discutimos amplamente nos parágrafos anteriores.

Assim como na função *inserir*, essa função também recebe o endereço onde está nossa fila. Você se lembra porque isso precisa ser feito? Obviamente sim! Você acaba de pensar que é para que essa função tenha total acesso à fila criada na função que chamou essa função – nesse caso, ter acesso à informação que está no *início* da fila, bem como consultar e atualizar a variável que controla essa posição inicial. O cabeçalho da função fica assim:

```
int consumir(struct fila *f) { // recebe o endereço da fila
```

Após criar uma variável local para receber o valor que foi consumido da fila, é necessário fazer a primeira verificação, para identificar se ainda temos posição inicial válida, ou seja, se a posição inicial ainda está abaixo da última posição. Por isso, uma comparação entre a posição inicial sendo

menor que o valor da última posição. Veja abaixo nossa sugestão de código, e depois representado visualmente, na Figura 9.6:

```
int valor; // cria uma variável local para receber o valor consumido
if(f->inicio < TAMANHO) { // verifica se não chegamos ao final da fila
```

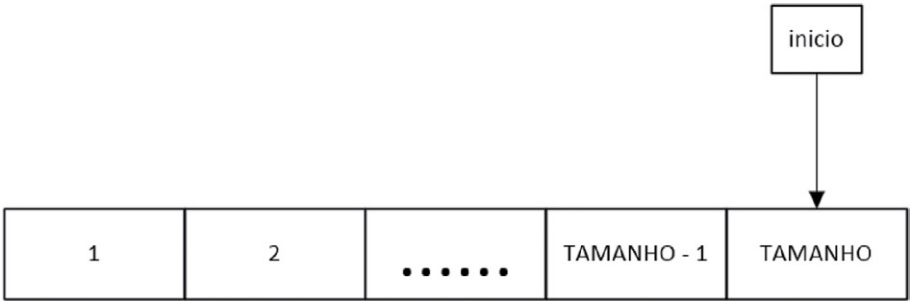


FIGURA 9.6: Quando o ponteiro *inicio* aponta para a última posição.

Após saber que temos uma posição de início de fila adequado, é importante verificar se temos alguma informação na fila, ou seja, se estivermos com o início da fila na posição inicial e a informação que tivermos for -1, valor que indica que o vetor apenas foi inicializado (veremos essa inicialização na função principal), isso caracterizará que não temos nenhuma informação na fila, portanto, não há o que consumir. Fizemos a verificação com a condicional a seguir, e demonstramos esse cenário visualmente na Figura 9.7:

```
if((f->inicio == 0) && (f->informações[f->inicio] == -1)) {
```

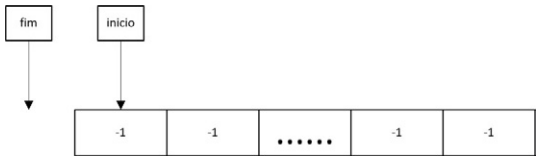


FIGURA 9.7: Cenário em que não houve inserções na fila.

Cair no *else* dessa condição significa que estamos em uma posição que pode ser a inicial, mas tem algo a ser consumido, portanto, é hora de fazer o que estávamos realmente querendo nessa função. Isso acontecerá em apenas duas linhas de código, pois basta guardar a informação que está nessa posição e atualizar a variável que guarda a posição onde está o início da fila. A Figura 9.8 ilustra esses passos, e nossa sugestão de código é apresentada abaixo:

```
valor = f->informacoes[f->inicio]; // obtém o valor da posição e atribui na variável local

f->inicio++;        // atualiza a posição onde se encontra a posição inicial
```

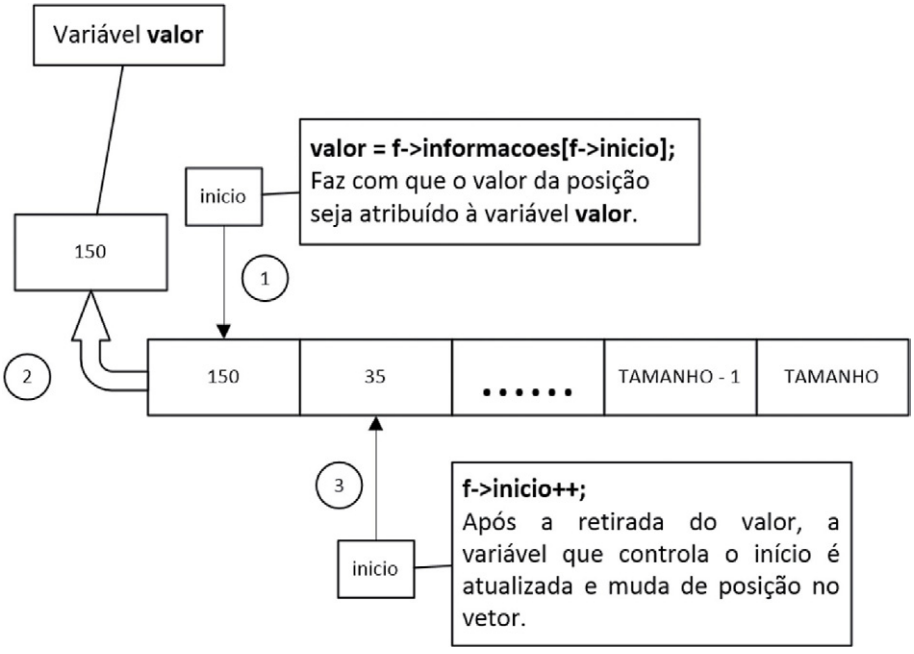


FIGURA 9.8: Sequência de retirada de um valor da fila.

Veja a sugestão desse código completo, com outros tratamentos e impressões de mensagens de erros encontrados:

Código 9.2

```
// Função para retirar/consumir um elemento da fila, se existir
int consumir(struct fila *f) // recebe o endereço da fila
{
	int valor; // cria uma variável local para receber o valor consumido
	if(f->inicio < TAMANHO) // verifica se não chegamos ao final da fila
	{
		if((f->inicio == 0) && (f->informações[f->inicio] == -1))
		{
			printf("Tentativa de consumir o primeiro elemento que ainda nao
existe. \n"); // Avisa que houve um problema
			valor = -1; // retorna um valor indicando problema
		}
		 else
		{
			valor = f->informações[f->inicio]; // obtém o valor da posição e atribui na
                                                // variável local
			f->inicio++; // atualiza a posição onde se encontra a posição inicial
			printf("<<< Consumido <<<: %d\n",valor); // imprimi na tela para controle
		}
	}
	else
	{
		printf("Tentativa de consumir uma informacao inesistente. \n\n"); // Avisa
		// que houve um problema
		valor = -1; // retorna um valor indicando problema
	}
	return valor; // retornando valor consumido ou erro
}
```

Antes de entrar na função principal, o que você acha de fazermos uma função que nos auxilie a ver como a fila está? Estamos falando de fazer uma função de impressão da fila, onde ela pode nos mostrar quais informações (valores) estão na fila naquele momento. Acreditamos que isso possa facilitar nosso aprendizado.

Para imprimir o que está na fila, temos que percorrer a fila desde a posição inicial até a posição final, imprimindo os valores de cada posição

intermediária a elas (veja a Figura 9.9). Para fazer isso, necessitamos de um comando de repetição; nesse caso, preferimos utilizar o comando *for*, mas você pode usar qualquer comando de repetição que conheça. Veja como esse comando de repetição ficou na nossa sugestão:

```
for(int i = f.inicio; i <= f.fim; i++){
```

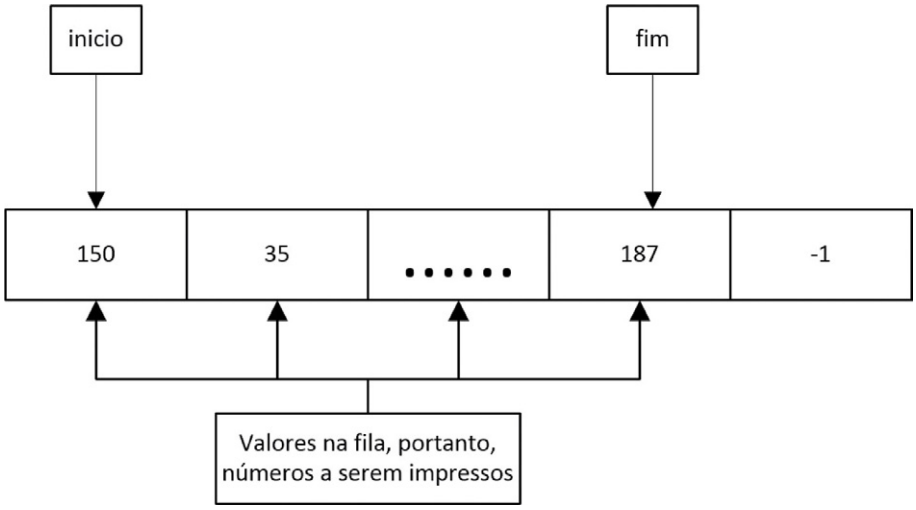


FIGURA 9.9: Cenário típico de impressão dos elementos que estão em uma fila, considerando que há elementos na fila.

Dentro desse comando, para melhorar nossa visualização nos exemplos, fizemos impressões diferenciadas para as posições inicial e final. Por isso, dentro do *loop* temos um comando *if*, *elseif* e um *else*. O primeiro *if* é para saber que estamos na posição inicial da fila; o comando *elseif* deve ser acionado apenas quando estivermos na posição final da fila; as outras posições são impressas no *else* final, portanto, esse trecho do código fica assim:

```
if(i == f.inicio) printf("inicio[%d],",f.informações[i]);
else if(i == f.fim) printf(" fim[%d]",f.informações[i]);
else printf(" %d,",f.informações[i]);
```

Papo técnico:

Lembre-se de que, dentro de um comando condicional, como o *if*, caso tenhamos apenas uma linha de comando, não necessitamos dos "{" e "}" para delimitar o escopo. Isso também se aplica aos comandos de repetição, entre outros.

Antes de terminar essa função, é necessário identificar quando a fila está vazia e imprimir um aviso a esse respeito, pois nem sempre teremos uma impressão válida. Como podemos identificar que temos uma fila vazia? Pense um pouco sobre o posicionamento das posições inicial e final dentro do vetor. Qual é sua principal característica? Se você pensou que a posição inicial deve vir sempre antes da posição final, está no caminho certo. Nesse caso, verificaremos apenas se a posição inicial está à frente da posição final, por isso, nossa condicional ficou como apresentado a seguir. A Figura 9.10 representa graficamente esse cenário.

```
if(f.inicio == (f.fim + 1))
```

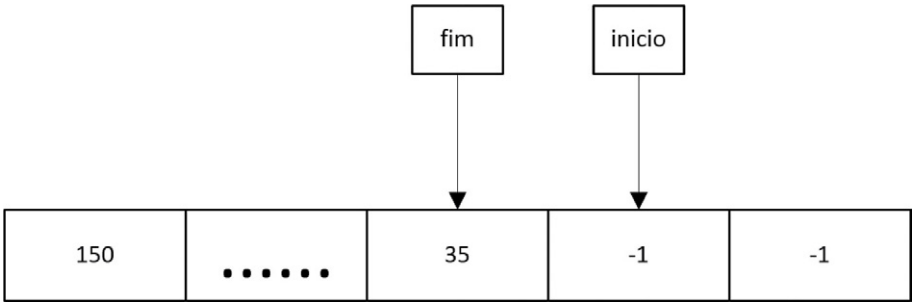


FIGURA 9.10: Com os ponteiros nessa posição, caracterizamos a fila vazia.

Apresento a seguir uma sugestão completa para essa função.

Código 9.3

```
// Função que imprimi apenas os elementos existentes na fila
void imprimir(struct fila f)
{
	printf("\n\n---------- Fila atual ----------\n");
	// Loop para imprimir na tela todos os elementos atuais da fila, por isso a variável de interação i
	// é inicializada com o valor da posição inicial da fila até a posição final.
	for(int i = f.inicio; i <= f.fim; i++)
	{
		if(i == f.inicio) printf("inicio[%d],",f.informações[i]);
		else if(i == f.fim) printf(" fim[%d]",f.informações[i]);
		else printf(" %d,",f.informações[i]);
	}
	if(f.inicio == (f.fim + 1))
		printf("Fila vazia!");
	printf("\n----------\n\n");
}
```

Após definir as funções *inserção* na fila, consumo de uma informação da fila e também uma função que nos ajudará a visualizar nossa fila, gostaríamos de propor uma função principal que faça uso de todas elas. Mas não a use sem propósito exemplifique as formas de usá-la e faça alguns testes, ou seja, tente fazer algumas operações que não serão possíveis. Foi com esses dois objetivos que elaboramos essa função.

A primeira parte dessa função principal é a declaração de nossa fila e sua inicialização. Isso consiste em duas partes principais: a declaração de uma variável do tipo *struct* fila, que, no nosso caso, chamaremos simplesmente de *f*. Essa declaração fica assim:

```
struct fila f;
```

Papo técnico:

Lembre-se de que, ao se declarar uma variável, é alocada uma região de memória na *heap* suficientemente grande para conter todas as informações daquele tipo abstrato (nesse caso, não é um tipo primitivo). Portanto, ao declarar uma variável local *f*, já temos uma região de memória alocada para guardar nossa fila.

Após essa declaração, a segunda etapa é a inicialização dos elementos da nossa fila. Para a posição inicial, atribuiremos a posição zero, pois é a primeira posição em que colocaremos uma informação e, posteriormente, a obteremos. Já para a posição final, como temos um algoritmo que atualiza essa variável antes de atribuir o valor na posição que ela indica, temos de inicializá-la com -1, pois, após incrementada, pela primeira vez ela ficará com zero e será a posição inicial.

Já para o vetor que representa a fila, sugerimos que se faça uma inicialização com valores que teoricamente não serão usados. Nesse exemplo, assumimos que o valor -1 nunca será usado na fila como uma informação válida, por isso, o atribuímos em todas as posições.

Essas duas etapas estão contempladas nas linhas de código abaixo e representadas graficamente na Figura 9.11:

```
struct fila f; // declaração da fila
f.inicio = 0;
f.fim = -1;
for(int i = 0; i < TAMANHO; i++)
      f.informações[i] = -1;
```

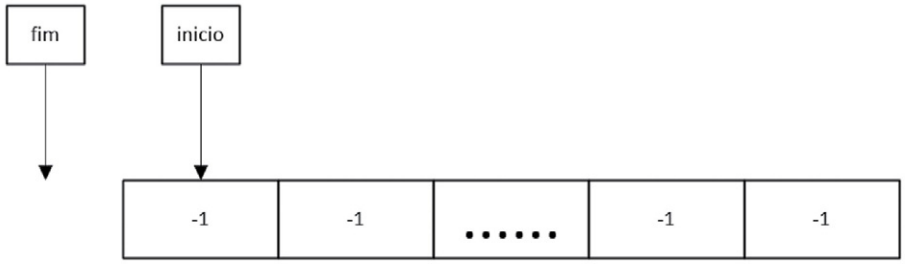


FIGURA 9.11: A fila, inicialmente, fica dessa maneira.

Depois da fase de declarações e inicializações, a fila está pronta para ser usada, portanto, já podemos manipulá-la. Como temos dois objetivos claros nesse exemplo – exemplificar o uso das funções de fila e testar alguns

comportamentos –, já iniciamos fazendo uso da função de consumir um valor sem sequer ter inserido algo. Nesse caso, como estamos usando a função *consumir* e ela retorna um valor consumido, necessitamos sempre atribuir o retorno dela em uma variável do tipo inteira, que é a nossa variável *valor*. Lembre-se de que essa função recebe o endereço de memória onde a fila está, pois necessita manipulá-la. Por isso passamos & *f* como parâmetro, ou seja, enviamos o endereço de memória onde a variável local *f* está alocada, e assim permitimos que a função faça o que for necessário com essa região de memória. Logo após o consumo de uma informação, fazemos a chamada para a função de impressão para verificar como se encontra a fila depois daquela passagem. O trecho de código dessa passagem é:

```
// Primeiro teste, o que acontece ao consumir algo de uma fila vazia?
valor = consumir(&f);
imprimir(f);
```

Será que você consegue identificar o que será mostrado na tela após essas duas linhas de código? Faça uma verificação nas funções envolvidas para ter uma ideia melhor e tente escrever essa saída em uma folha de seu caderno. Fazer esse teste é a melhor forma de entender o funcionamento da pilha.

Logo após esse primeiro teste, é feito uso da outra função que criamos, que é a de inserir um valor na fila. Essa função também recebe o endereço de memória da fila, pois, da mesma forma que a função de consumo, manipula as informações da fila e necessita acessar a memória diretamente. Como ela não retorna nenhuma informação, basta chamá-la passando o & *f* (endereço da variável *f*), ficando assim sua chamada:

```
inserir(&f,10);
```

Apresentamos a seguir nossa sugestão completa para essa função principal. O que você acha de passar esse código linha a linha e escrever em uma folha de papel o que ele imprimirá na tela? Faça isso com calma e você verá que isso será bastante útil para o aprendizado total dos conceitos aqui apresentados. Boa sorte!

O código da função principal é:

Código 9.4

```
void main() {
        int valor = -1; // variavel que receberá a informação retirada da fila
        struct fila f; // declaração da fila
        f.inicio = 0; // inicializando a posição inicial como sendo a zero, ou seja, a posição
        f.fim = -1; // inicializando a posição final como sendo menos um, por que ao inserir um novo elemento essa
                    // posição é incrementada e se tornará a posição zero

        // Loop de inicialização de todas as posições da fila
        for(int i = 0; i < TAMANHO; i++)
                f.informações[i] = -1;

        // Primeiro teste, o que acontece ao consumir algo de uma fila vazia?
        valor = consumir(&f);
        imprimir(f);

        //Inserindo 3 valores na fila
        inserir(&f,10);
        inserir(&f,20);
        inserir(&f,30);
        imprimir(f);

        // Consumindo dois valores
        valor = consumir(&f);
        valor = consumir(&f);
        imprimir(f);

        // Por fim, inserindo mais dois valores e consumindo um
        inserir(&f,40);
        inserir(&f,50);
        valor = consumir(&f);
        imprimir(f);
}
```

Imaginando que você tenha passado esse código linha a linha, apresentando e escrevendo no papel o que era impresso, apresentamos na Figura 9.12, para simples conferência, o que esse código resultou em nossos testes:

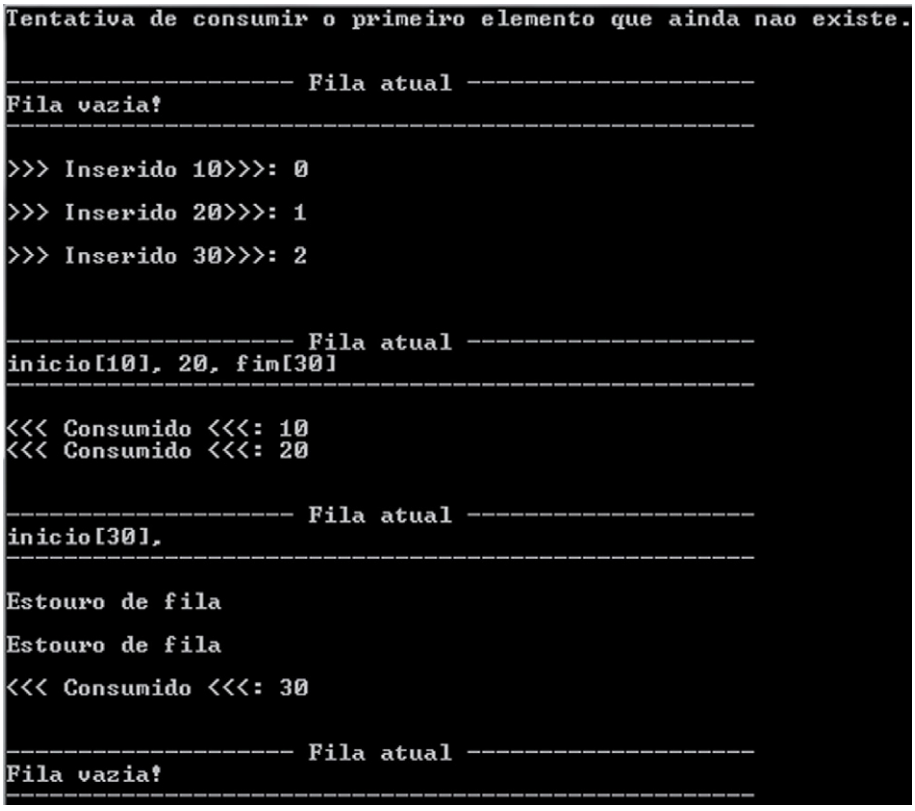


FIGURA 9.12: Saída do programa exemplo no *prompt* de comando.

Nesse momento fechamos a discussão e exemplificação inicial a respeito de uma fila e sua implementação em vetor de forma contínua.

Para que possamos aprofundar um pouco o tema, gostaríamos de induzir você a ponderar a respeito de limitações no que foi apresentado até aqui. Quais foram as limitações que você percebeu nessa maneira de simular uma fila em programação? Pense alguns minutos a respeito. Temos certeza de que você reparará em vários fatores.

Muito bem. Provavelmente, o principal ponto que fica evidente é a limitação na quantidade de informações dentro da fila criada e manipulada. A utilização de vetor para essa função (controlar as posições da fila) traz esse problema intrinsecamente, mas podemos pensar em duas maneiras de tentar escapar dele.

A primeira visão que podemos ter é de que, apesar de o vetor estar limitando a quantidade de informações que serão vinculadas, nesse momento, temos um problema na forma como fizemos a implementação, pois, ao atualizar a posição de *inicio*, não usamos mais aquela posição do vetor. A Figura 9.13 procura demonstrar e evidenciar as posições que estão desperdiçadas, o que pode ser encarado como problema, pois temos um pedaço de memória reservado e não usado – um desperdício, certo?

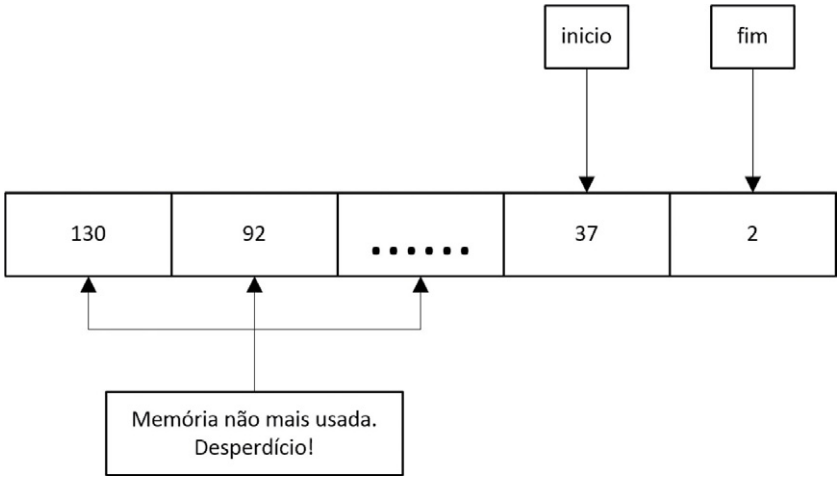


FIGURA 9.13: Posições já usadas desperdiçando espaço de memória.

Nós concordamos. Por isso, sugerimos a você que pense em uma implementação de vetor circular, ou seja, mesmo que o *final* da fila chegue ao final do vetor, se a posição *inicial* não estiver na posição zero do vetor, mesmo que o *final* da fila chegue ao final do vetor, se o *inicio* da fila não estiver nas posições iniciais do vetor, podemos deslocar o *final* da fila pelas posições em que o *início* da fila já tenha passado, fazendo com que o vetor fique circular. A Figura 9.14 caracteriza essa circularidade dos ponteiros.

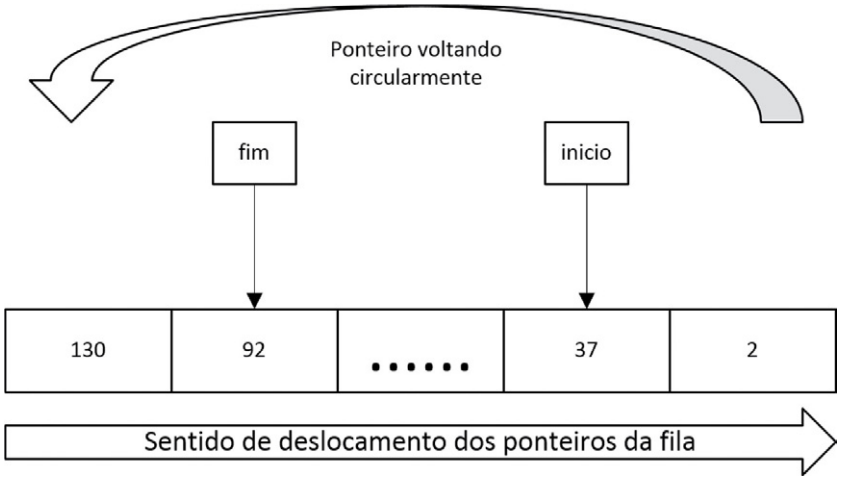


FIGURA 9.14: Vetor usado circularmente.

Na proposta apresentada, deve-se ter cuidado para que a posição *final* não ultrapasse a posição *inicial*, o que nos faz identificar outra limitação: o tamanho do vetor, nesse caso, é o tamanho máximo dessa fila.

Assim, para não haver limitação do tamanho da fila, uma ótima solução seria implementar essas mesmas funções levando em consideração o que foi aprendido e usado no capítulo 7 que fala sobre lista ligada. O que você acha de elaborar uma versão com essa nova possibilidade? Achamos que seria um exercício interessante para praticar e entender os conceitos aqui apresentados.

Vamos programar

Acabamos de ver a implementação de várias funções em linguagem C, mas, para que possamos ampliar nossos conhecimentos e horizontes, apresentamos as mesmas funções com a sintaxe nas linguagens JAVA e PYTHON. Aproveite para aprendê-las também.

Java

Código 9.1

```
private static void inserir (Queue<Integer> fila, int valor)
{
        try
        {
            fila.add(valor);
        }
        catch (Exception e)
        {
            System.out.println("Estouro de fila. Erro: "+e.toString());
        }
}
```

Código 9.2

```
private static int consumir (Queue<Integer> fila)
{
        try
        {
            return fila.poll();
        }
        catch (Exception e)
        {
            System.out.println("Tentativa de consumir uma informacao inesistente. Erro:
"+e.toString());
        }
        return -1;
}
```

Código 9.3

```
private static void imprimir (Queue<Integer> fila)
{
        System.out.println(fila);
}
```

Código 9.4

```
public static void main (String[] args)
{
        Queue<Integer> fila = new LinkedList<Integer>();
        int valor;
        // Primeiro teste, o que acontece ao consumir algo de uma fila vazia?
        valor = consumir(fila);
        System.out.println(fila);

        //Inserindo 3 valores na fila
        inserir(fila, 10);
        inserir(fila, 20);
        inserir(fila, 30);
        imprimir(fila);

        // Consumindo dois valores
        valor = consumir(fila);
        valor = consumir(fila);
        imprimir(fila);

        // Por fim, inserindo mais dois valores e consumindo um
        inserir(fila, 40);
        inserir(fila, 50);
        valor = consumir(fila);
        imprimir(fila);
}
```

Python

Código 9.1

```
def inserir(self, valor):
    if self.fim + 1 < TAMANHO:   #verifica se nao chegamos ao final da fila
      self.fim+=1   #atualiza a posicao final para inserir um novo valor
      self.informacoes[self.fim] = valor   #insere o valor na posicao
      print "Inserido : %d" % (valor)   #imprimi na tela para controle
    else:
      print "Estouro de fila"   #avisa que houve estouro de fila
```

Código 9.2

```
#Funcao para retirar/consumir um elemento da fila, se existir
  def consumir(self):
    valor=0   #cria uma variavel local para receber o valor consumido
    if self.inicio < TAMANHO:   #verifica se não chegamos ao final da fila
      if self.inicio == 0 and self.informacoes[self.inicio] == -1:
        print "Tentativa de consumir o primeiro elemento que ainda nao existe"
        valor = -1   #Avisa que houve um problema
      else:
        valor = self.informacoes[self.inicio]   #obtem o valor da posicao e atribui na variavel local
        self.inicio+=1   #atualiza a posicao onde se encontra a posicao inicial
        print "Consumido %d" % (valor)   #imprimi na tela para controle
    else:
      print "Tentativa de consumir uma informacao inexistente"
      valor = -1   #retorna um valor indicando problema
    return valor   #retornando valor consumido ou erro
```

Código 9.3

```
#Funcao que imprimi apenas os elementos existentes na fila
  def imprimir(self):
    print "Fila Atual"
    i =0
    while i<=self.fim: # Loop para imprimir na tela todos os elementos atuais da fila, por isso a variavel de
                       # interação i é inicializada com o valor da posição inicial da fila até a posição final
      if i == self.inicio:
        print "inicio[%d]" %(self.informacoes[i])
      elif i==self.fim:
        print "fim[%d]" %(self.informacoes[i])
      else:
        print self.informacoes[i]
      i+=1
    if self.inicio == self.fim +1:
      print "fila vazia"
    print "fim"
```

Código 9.4

```
#DEFINICAO DA ESTRUTURA
class Fila:
       def __init__(self):
              self.informacoes = [-1]*TAMANHO
              self.inicio = 0
              self.fim = -1

       .
       .
       .
       #Funções 9.1, 9.2 e 9.3 seriam colocadas aqui.
       .
       .
       .

       fila.consumir()
       fila.imprimir()

       fila.inserir(num)
       fila.inserir(num)
       fila.inserir(num)
       fila.imprimir()

       fila.consumir()
       fila.consumir()
       fila.imprimir()

       fila.inserir(num)
       fila.inserir(num)
       fila.imprimir()
```

Para fixar

1. Implemente seu próprio programa usando qualquer uma das propostas apresentadas neste capítulo.
2. Em seguida, crie uma rotina para a fila do restaurante na praça de alimentação, mas leve em consideração que ele tem apenas um atendente no caixa. Depois de ser atendido no caixa, o cliente passa a outra fila, para receber seu pedido. Nessa segunda fila, temos três atendentes.

Para saber mais

Alguns de vocês podem questionar a validade de uma estrutura de dados como a fila, mas ela é muito mais utilizada do que se pode imaginar.

Geralmente, em *sistemas operacionais* ou mesmo em *redes de computadores*, os recursos compartilhados têm políticas de acesso baseadas em filas, ou seja, os primeiros a chegar são os primeiros a ter os recursos disponibilizados para seu uso. Sem dúvida, existem recursos com várias filas, cada uma delas com prioridades diferentes. Mesmo assim, o compartilhamento do recurso está baseado em filas.

Um exemplo simples e corriqueiro dessa utilização é o *servidor de impressão* de seu laboratório de informática. Vários alunos podem mandar seus arquivos para serem impressos ao mesmo tempo, mas nem por isso os textos são impressos de forma misturada. Cada um dos arquivos é impresso de uma vez, sem ter as partes misturadas. Isso se deve ao enfileiramento dos arquivos que chegam ao servidor e são consumidos pela impressora.

Outro recurso, hoje muito importante, que se utiliza do enfileiramento das informações que chegam é o processador, em que o sistema operacional deve gerenciar uma boa quantidade de processos aptos a serem processados. Dentre as várias formas, a fila é uma das mais usadas.

Em redes de computadores, as filas são muito usadas em algoritmos de comunicação de dados, principalmente para receber, processar e enviar pacotes que trafegam pelos equipamentos de núcleo de rede. A fila de entrada dos servidores é extremamente explorada para fazer ataques a esses recursos. Como a fila, por melhor implementada que seja, esbarra na limitação de memória do servidor, os atacantes sempre tentam explorar seu congestionamento total para fazer com que o servidor pare de responder e inicie o processo de negar as novas requisições feitas, levando o cliente a

obter a informação de que o servidor está indisponível naquele momento, ou mesmo inutilizando um importante nó do núcleo da rede, provavelmente congestionando em cascata os demais nós da rede.

Por este último exemplo, podemos perceber que as filas são bastante importantes. Imagine se nossas redes ficassem vulneráveis dessa forma e tivéssemos um colapso total na internet, a ponto de não podermos acessar momentaneamente nossas redes sociais. Como seria isso?

Existe uma interessante discussão sobre *fila de prioridade* no Capítulo 4 do livro *Estrutura de dados usando C* (Tenenbaum, 2004).

Navegar é preciso

Para se aprofundar no tema, saber um pouco sobre a famosa "teoria das filas" é muito importante. Para isso, acesse a discussão presente em:

http://www.eventhelix.com/RealtimeMantra/CongestionControl/queueing_theory.htm#.UhAnJ3-YdD0.

Os simuladores presentes em http://queueing-systems.ens-lyon.fr/ dão exemplos de vários tipos de fila.

Exercícios

1. Tente simular uma fila de banco, nesse caso, apenas uma fila de clientes e dois caixas de atendimento.
2. Acrescente à simulação anterior uma fila prioritária.
3. Adicione também um caixa de atendimento. Ele deve ser o único caixa a atender à fila prioritária.
4. Por fim, acrescente as impressões de tempo de espera de cada cliente. Considere que chega um cliente a cada segundo, e que cada um dos caixas atende a um cliente a cada dois minutos.
5. Crie uma estrutura cliente em que se tenha o número de chegada dele e o tempo que ele estima usar no caixa (o cliente deve saber o tempo que vai demorar no caixa). Simule um atendimento com quatro caixas e uma única fila, que receba com frequência clientes com diferentes tempos de atendimento (você pode escolher como fará essa atribuição, porém, ela tem que ser aleatória).
6. Elabore um programa que simule um caixa que demora sempre o mesmo tempo para atender a um cliente. O problema é que a frequência de chegada de novos clientes pode variar e fazer com que a fila cresça. Implemente essa fila com um limite de 10 clientes e que tenha vetor circular.

Glossário

FIFO: *First In, First Out* – o primeiro que entra é o primeiro a sair.

FILO: *First In, Last Out* – o primeiro que entra é o último a sair.

Referências bibliográficas

PIVA, D. et al. *Algoritmos e programação de computadores*. Rio de Janeiro: Campus, 2012.

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

O QUE VEM DEPOIS

Você já conhece várias maneiras de fazer busca, ordenação e listas ligadas, além das filas. No Capítulo 10 será a vez de aprender um pouco sobre uma nova estrutura de dados chamada *pilha*.