**Apostila**

**Filas – implementação com Arranjo**

Sumário

[**Introdução** 1](#_Toc7443734)

[**Filas – conceitos básicos** 2](#_Toc7443735)

[**Filas – exemplos e aplicações** 2](#_Toc7443736)

[**Fila – implementação com *arranjo circular*** 3](#_Toc7443737)

[Detalhando a estrutura da Fila 6](#_Toc7443738)

[Criando as operações *sucessor*( ) e *antecessor*( ) 7](#_Toc7443739)

[Detalhando as operações *isEmpty*( ) e *isFull*( ) 8](#_Toc7443740)

[Detalhando a operação *enqueue*() 8](#_Toc7443741)

[Detalhando a operação *dequeue*() 9](#_Toc7443742)

[**Conclusão** 10](#_Toc7443743)

# **Introdução**

Nos estudos sobre pilhas, foi aprendido o conceito de *estruturas com restrição de acesso*, que nada mais são do que estruturas onde os acessos (inserções e retiradas) aos elementos ocorrem de forma restrita e imutável, visando a atender a uma política ou a um modelo de negócio.

Uma das estruturas com restrição de acesso de maior interesse pela Ciência da Computação é a **fila**, por implementar uma política de acesso **FIFO** – **F**irst **I**n, **F**irst **O**ut, modelo natural para representar situações ProdutorXConsumidor, em que a capacidade de produzir seja eventualmente maior que a de consumir. Neste caso, o excedente é organizado numa estrutura em que o primeiro a ser produzido será o primeiro a ser consumido.

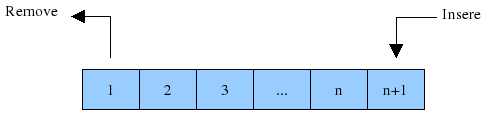
Nesta apostila será visto em mais detalhes a estrutura Fila e suas diversas aplicações práticas.

# **Filas – conceitos básicos**



*FILA* ou *QUEUE* é um caso particular de *DEQUE* em que as retiradas são realizadas somente em uma extremidade, denominada *início*, ou *frente*, ou *front*, em inglês, enquanto que as inserções ou entradas ocorrem obrigatoriamente no lado oposto, denominado *fim*, ou *traseira*, ou em inglês, *rear*. Filas implementam uma politica de acesso denominada **FIFO**, que é um acrônimo da expressão em inglês **F**irst **I**n, **F**irst **O**ut, ou seja, o primeiro que chega é o primeiro a ser atendido. Esta política de acesso é também eventualmente chamada por alguns autores de **LILO** – **L**ast **I**n, **L**ast **O**ut.

Do ponto de vista *estrutural*, uma *Fila* é similar a uma *Lista*: elementos são organizados em uma estrutura linear, unidimensional. A diferença está no *comportamento*. A *Lista* não tem restrições para inserções e retiradas, que podem ocorrer em qualquer ponto da lista. Já a *Fila* apresenta *restrição de acesso*, isto é, toda inserção é realizada de um lado enquanto que toda remoção ocorre sempre e obrigatoriamente do lado oposto, conforme ilustrado na figura abaixo.



Basicamente, as operações de manipulação de elementos numa fila são duas:

1. *Enfileira* ou *Enqueue*: insere um elemento no final da fila.
2. *Desenfileira* ou *Dequeue*: remove um elemento do início da fila.

# **Filas – exemplos e aplicações**

Como já foi citado, a politica de acesso FIFO implementada por filas é solução natural para situações onde a capacidade de produzir insumos é, ao menos momentaneamente, maior do que a capacidade de consumir estes insumos. Se enquandram neste cenário:

1. Atendimento a clientes num guichê de banco ou de um supermercado.
2. Pessoas adquirindo bilhetes para o cinema ou para o concerto de música.
3. Carros passando por uma ponte estreita.
4. Pessoas embarcando em um avião ou trem.
5. Aviões em procedimento de pouso ou decolagem em um aeroporto.

No ambiente computacional também ocorre várias aplicações envolvendo processos requisitando recursos compartilhados, onde pode haver contenção destes recursos:

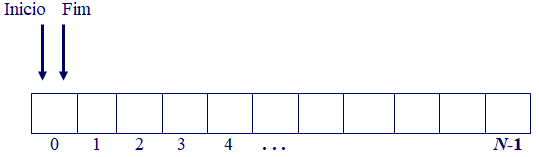
1. Pacotes de dados sendo transmitidos entre *hubs* numa rede de computadores.
2. Jobs de impressão chegando num *pool* de impressoras.
3. Caracteres sendo digitados pelo teclado do computador.
4. *Buffer* de gravação de dados em disco ou em quaisquer mídias.
5. Transações de banco de dados envolvendo dados compartilhados

Note que em todos estes exemplos, e em muitos outros não citados, a parte consumidora pode até ficar ociosa em algum momento, mas em outro momento, os insumos podem chegar aleatoriamente próximos, superando a capacidade de consumo e exigindo a organização da fila para um correto atendimento.

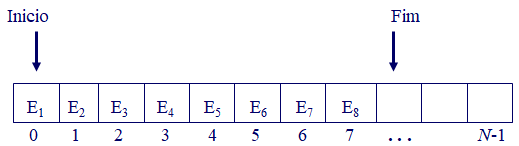
# **Fila – implementação com *arranjo circular***

Visando aprofundar o conhecimento da fila, será construída um TAD que abstrai uma fila utilizando como estrutura de armazenamento um arranjo circular. ***Arranjo Circular***? Isso mesmo. A seguir será demonstrado o que é um arranjo circular e a a razão de adotá-lo na implementação de uma lista.

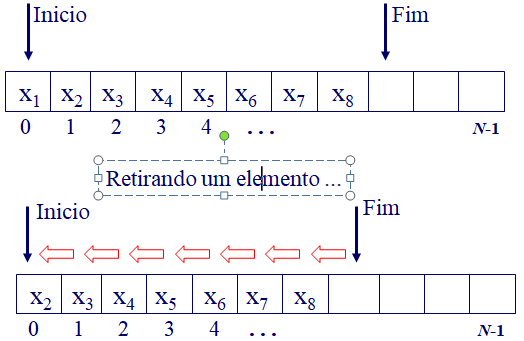
Assim como a Pilha, uma Fila é, estruturalmente igual a uma Lista. A fila deve ter um vetor para armazenar seus elementos. O que se acrescenta aqui é uma marca para o ***inicio*** e outra para o ***fim*** da fila. Estas marcas simplificam os algoritmos das operações de enfileirar (*enqueue*) e desenfileirar (*dequeue*). A figura abaixo ilustra a estrutura de uma fila vazia, onde as marcas de início e fim se posicionam no elemento de índice 0 do arranjo.



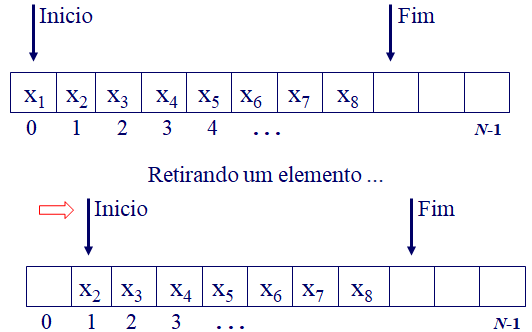
À medida em que elementos são inseridos, a marca de **fim** da fila vai caminhando no sentido do final do arranjo, como representado abaixo.



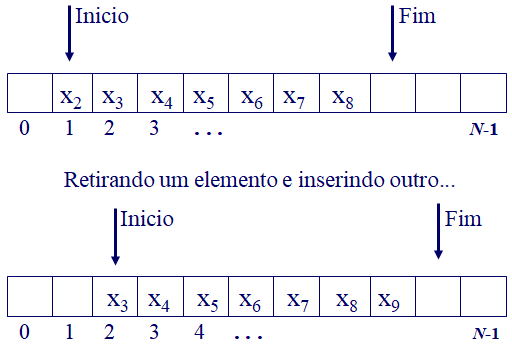
Deste modo, a operação de inserção é simples e de custo computacional constante. O inconveniente desta abordagem é o alto custo da operação de remoção, que exige mover todos os elementos em direção ao início da fila, como ilustrado a seguir:

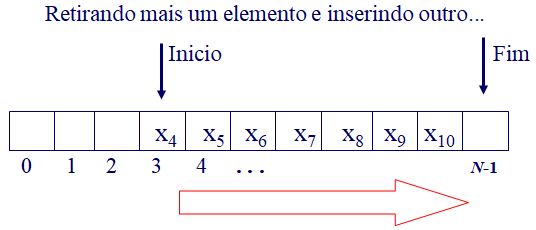


Para evitar esse inconveniente e tornar o custo computacional da remoção também constante, a solução é, na remoção, simplemente mover a marca de início da fila uma posição à direita, conforme ilustrado abaixo:



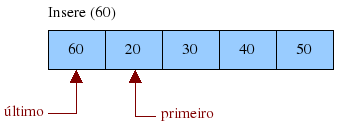
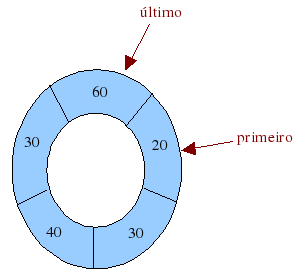
Deste modo, sucessivas inserções e remoções tem o seguinte comportamento:





A imagem acima ilustra que este comportamento dinâmico da Fila tende a deixar elementos vazios no inicio do arranjo, enquanto que caminha-se rapidamente para uma situação de *overflow* por falta de elementos disponíveis no final do arranjo.

Uma solução elegante para aproveitar os elementos vazios do início do arranjo é adotar a abordagem de ***arranjo circular***. Nesta abordagem, é como se o primeiro elemento fosse o subsequente do útimo elemento do arranjo, daí o nome, arranjo circular, pois é como se o arranjo se tornasse um círculo:

Há duas formas de tornar um arranjo circular:

1. Por meio de testes lógicos (comando **if**): sempre para definir um sucessor ou antecessor de um elemento do arranjo com N elementos, testar:

Cálculo do Sucessor:

if(i == N-1)

sucessor = 0;

else sucessor = i+1;

Cálculo do Antecessor:

if(i == 0)

antecessor = N-1;

else antecessor = i-1;

1. Por meio da **aritmética modular**, utiliza-se o módulo ou resto da divisão inteira (é mais elegante pois elimina o teste lógico):

Cálculo do Sucessor:

sucessor = (i+1)% N;

Cálculo do Antecessor:

antecessor = (i–1+N)% N;

Nesta abordagem, a definição de Fila Vazia, já feita antes, é:

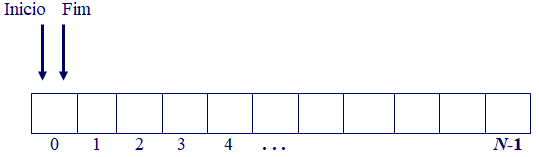
**Fila Vazia:** quando **Fim == Inicio**

E a definição de Fila Cheia então é, quando o Fim é o antecessor do Inicio, ou seja:

**Fila Cheia:** quando **(Fim+1)%N == Inicio**

Com as definições e premissas acima, está tudo pronto para iniciar a implementação d *TAD Fila*. Para tanto, será necessário definir a estrutura de dados e as operações da Fila.

## Detalhando a estrutura da Fila



Conforme a abstração da fila explicitada na figura acima, sua estrutura deve conter:

* Um vetor para armazenar os elementos da fila.
* Uma marca de início da fila, do tipo int.
* Uma marca de fim da fila, do tipo int.

Traduzindo a estrutura acima para a linguagem computacional, tem-se:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| const MAX = 1000  Dados: **Registro**  ***// campos da estrutura de dados da Lista***  **FimRegistro**  Fila: **Registro**  dados: vetor [MAX] de Dados  inicio, fim : inteiro  **FimRegistro** | **#define** MAX 1000  **typedef struct {**  ***// campos da estrutura de dados da Lista***  **} Dados;**  **typedef struct {**  **Dados** dados[MAX];  **int** inicio, fim;  **} Fila;** |

## Criando as operações *sucessor*( ) e *antecessor*( )

Estas operações são auxiliares para a implementação do arranjo circular. Ou seja, serão usadas apenas internamente no TAD Fila e quem for usar o TAD nem precisa saber de suas existências.

A operação *sucessor* tem a assinatura **int** sucessor(**int** pos) calcula e retorna o sucessor de uma posição informada como parâmetro, usando a fórmula: (pos+1)% N;

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** sucessor (pos: inteiro): inteiro  **Inicio**  Retorne ((pos+1) % MAX)  **FimFuncao** | **int** sucessor (int pos)  **{**  **r**eturn (++pos % MAX);  **}** |

A operação *antecessor* tem a assinatura **int ante**cessor(**int** pos) calcula e retorna o antecessor de uma posição informada como parâmetro, usando a fórmula: (pos-1+N)% N;

Esta operação fica indicada mas não será implementada por não ser utilizada no presente estudo.

## Detalhando as operações *isEmpty*( ) e *isFull*( )

A operação *isEmpty* tem a assinatura **int isEmpty**(Fila \* fila) e retorna o resultado do teste lógico fila.inicio == fila.fim;

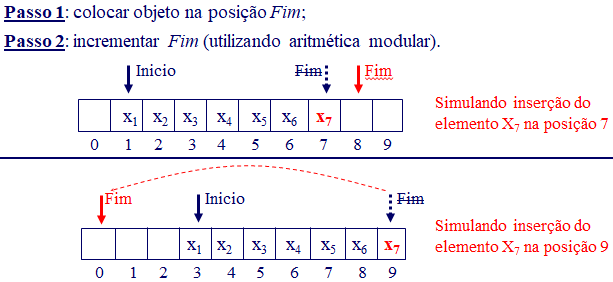
|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** isEmpty (var fila: Fila): logico  **Inicio**  Retorne (fila.inicio = fila.fim)  **FimFuncao** | **int** isEmpty (Fila \* fila)  **{**  **r**eturn (fila->início == fila->fim);  **}** |

A operação *isFull* tem a assinatura **int isFull(Fila \* fila)** e retorna o resultado do teste lógico sucessor(fila.fim)== fila.inicio;

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** isFull (var fila: Fila): logico  **Inicio**  Retorne (sucessor(fila.fim) = fila.inicio)  **FimFuncao** | **int** isFull (Fila \* fila)  **{**  **r**eturn (sucessor(fila->fim) == fila->inicio);  **}** |

## Detalhando a operação *enqueue*()

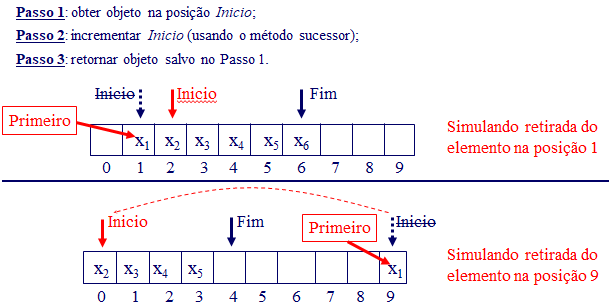
A operação *enqueue* tem a assinatura **void enqueue**(Dados d, Fila \*fila, int \* codErro). Recebe um elemento de dados *x*, como parâmetro de entrada, e o insere no final da *fila*. Antes testa *overflow* da fila. A imagem a seguir ilustra e descreve o algoritmo de inserção:



|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** enqueue (x: Dados, **var** fila: Fila,  **var** codErro; inteiro)  **Inicio**  **Se** ( isFull (fila) ) **Entao**  codErro = 999;*// Fila Cheia*  **R**etorne;  **FimSe**  fila.dados[fila.fim] 🡨 x;  fila.fim 🡨 sucessor (fila.fim);  **FimFuncao** | **void** enqueue (Dados x, Fila \* fila, int \*codErro)  **{**  **if** ( isFull (fila) ) {  \*codErro = 999;*// código para Fila Cheia*  **r**eturn;  }  fila->dados[fila->fim] = x;  fila->fim = sucessor (fila->fim);  **}** |

## Detalhando a operação *dequeue*()

A,operação *dequeue* tem a assinatura **void dequeue**(Dados \*x, Fila \*fila, int \*codErro). Retira e devolve o elemento do início da fila. Antes testa *underflow* da Fila. A imagem abaixo ilustra e descreve o algoritmo de remoção:



|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** dequeue(**var** x: Dados, **var** fila: Fila,  **var** codErro; inteiro)  **Inicio**  **Se** ( isEmpty (fila) ) **Entao**  codErro = 999;*// Fila Cheia*  **R**etorne;  **FimSe**  x 🡨fila.dados[fila.inicio];  fila.inicio 🡨 sucessor (fila.inicio);  **FimFuncao** | **void** dequeue (Dados\* x, Fila \* fila, int \*codErro)  **{**  **if** ( isEmpty (fila) ) {  \*codErro = 999;*// código para Fila Vazia*  **r**eturn;  }  \*x = fila->dados[fila->inicio];  fila->inicio = sucessor (fila->inicio);  **}** |

# **Conclusão**

Nesta apostila estudou-se o conceito de Fila. Inicialmente foram apresentados os conceitos básicos de uma fila e diversos exemplos e situações práticas em que se aplica a política FIFO como solução. Em seguida foi apresentada uma alternativa de implementação de fila, usando a abordagem de arranjo circular.