# Aula 7 – Filas (Queues) e suas Variações

Imagine o final de um longo dia de trabalho. Você está cansado, mas motivado para dar o próximo passo na sua jornada de aprendizado. Você abre a plataforma de streaming para ouvir um podcast e, sem pensar, adiciona três episódios à sua lista de reprodução. O primeiro que você adicionou é o primeiro a tocar. Em seguida, você envia um documento para a impressão e, logo depois, um colega faz o mesmo. Sua impressão, por ter chegado antes, é a primeira a ser executada. Essas situações, tão comuns em nosso dia a dia, são a manifestação de um dos conceitos mais fundamentais da ciência da computação: a **fila**.

Nesta aula, vamos desmistificar essa estrutura de dados poderosa e intuitiva. Ao final destes 45 minutos, você não apenas entenderá o que é uma fila, mas será capaz de analisar sua eficiência, decidir qual a melhor forma de implementá-la para um problema específico e reconhecer sua presença nos sistemas que você usa todos os dias. Vamos mergulhar no princípio que rege a ordem e a justiça no mundo digital: o *First-In, First-Out* (FIFO). Exploraremos como construir filas usando as ferramentas que já conhecemos, como arrays e listas ligadas, e descobriremos variações inteligentes, como as filas circulares e os versáteis *deques*.

Esta jornada conectará o conhecimento que você já adquiriu sobre estruturas de dados lineares com aplicações práticas e de alto impacto. Vamos entender como a escolha correta de uma estrutura de dados como a fila é um pilar para a escrita de código eficiente, impactando desde o desempenho de um sistema de e-commerce até a forma como seu sistema operacional gerencia tarefas. Prepare-se para ver o mundo digital sob uma nova ótica, onde a organização e a ordem são a chave para a eficiência.

### O Primeiro a Chegar é o Primeiro a Sair

# O Princípio FIFO: Entendendo a Essência das Filas

O que uma fila de atendimento bancário, os comandos que você digita no terminal e a gestão de processos em um servidor web têm em comum? Todos operam sob um princípio de justiça e ordem cronológica. A primeira pessoa que chega ao banco é a primeira a ser atendida. O primeiro comando que você envia é o primeiro a ser executado. Essa lógica, conhecida como **FIFO (First-In, First-Out)**, é a alma da estrutura de dados que chamamos de **fila** (*queue*). É uma regra simples, mas que impede que novos itens "furem a fila", garantindo um processamento ordenado e previsível.

Pense na fila como um túnel com uma entrada e uma saída distintas. Os carros (nossos dados) entram por um lado e, inevitavelmente, saem pelo outro na mesma ordem em que entraram. Não há como um carro que acabou de entrar ultrapassar os que já estavam lá. Essa estrutura é definida por duas operações fundamentais que espelham essa ação. A operação de adicionar um novo elemento ao final da fila é chamada de **enqueue** (enfileirar). Já a operação de remover o elemento que está na frente da fila é chamada de **dequeue** (desenfileirar). Simples, elegante e extremamente poderoso.

Construindo a Fila com Arrays

# A Primeira Tentativa: Implementando Filas com Arrays

01

### Estrutura Inicial

Usamos um array para armazenar os elementos e dois índices: **frente** (primeiro elemento) e **traseira** (última posição vaga)

02

### Operação Enqueue

Adicionamos um elemento na posição **traseira** e incrementamos o índice. Complexidade: **O(1)**

03

### Operação Dequeue

Removemos o elemento da posição **frente** e avançamos o índice. Complexidade: **O(1)**

Agora que a ideia de uma fila está clara, como podemos traduzir esse conceito para o código? A primeira ferramenta que geralmente vem à mente é o **array**, uma estrutura que já conhecemos bem. Parece simples: podemos usar um array para armazenar os elementos e dois "ponteiros" (na verdade, índices) para controlar o início e o fim da fila. Um índice, que podemos chamar de frente, apontará para o primeiro elemento, e outro, chamado traseira, para a última posição vaga.

> ⚠️ **O Problema do Desperdício**
>
> Conforme removemos elementos com dequeue, o índice frente se move, deixando espaços vazios e inutilizados no início do array. É como se nossa prateleira ficasse com vários espaços vazios no começo, mas estivesse "cheia" no final, impedindo a adição de novos livros. Chegaria um momento em que traseira atinge o final do array, e o sistema acusa "fila cheia", mesmo que o array esteja quase todo vazio.

Vamos usar uma analogia: imagine uma prateleira de livros com espaço para exatamente 10 exemplares. Quando adicionamos um livro (enqueue), o colocamos no primeiro espaço livre no final (traseira). Quando alguém pega um livro emprestado (dequeue), sempre pega o que está na frente da prateleira (frente). A cada empréstimo, o ponteiro frente avança uma posição. Parece funcionar, certo? Inicialmente, sim. A operação de enqueue tem complexidade $O(1)$, pois apenas adicionamos um elemento e incrementamos um índice. O dequeue também é $O(1)$.

Isso nos leva a uma questão crucial: como podemos reutilizar esses espaços vazios no início do array sem ter que mover todos os elementos (o que seria uma operação custosa de complexidade $O(n)$)? A resposta está em fazer a fila "dar a volta".

### A Solução Elegante das Filas Circulares

# A Fila que se Reinventa: O Poder da Fila Circular

Revisitando o problema do espaço desperdiçado em nossa implementação com array, a solução é tão elegante quanto eficiente. E se, ao invés de uma prateleira linear, tivéssemos uma estante de livros giratória? Quando o ponteiro traseira atinge o último espaço, em vez de parar, ele simplesmente "dá a volta" e continua a partir do primeiro espaço, caso ele esteja livre. Essa ideia de conectar o final ao início transforma nosso array linear em uma **fila circular**.

> **A Mágica do Módulo**
>
> A operação matemática **módulo (%)** faz o ponteiro "pular" do final para o começo
>
> ```
> (indice + 1) % tamanho_do_array
> ```

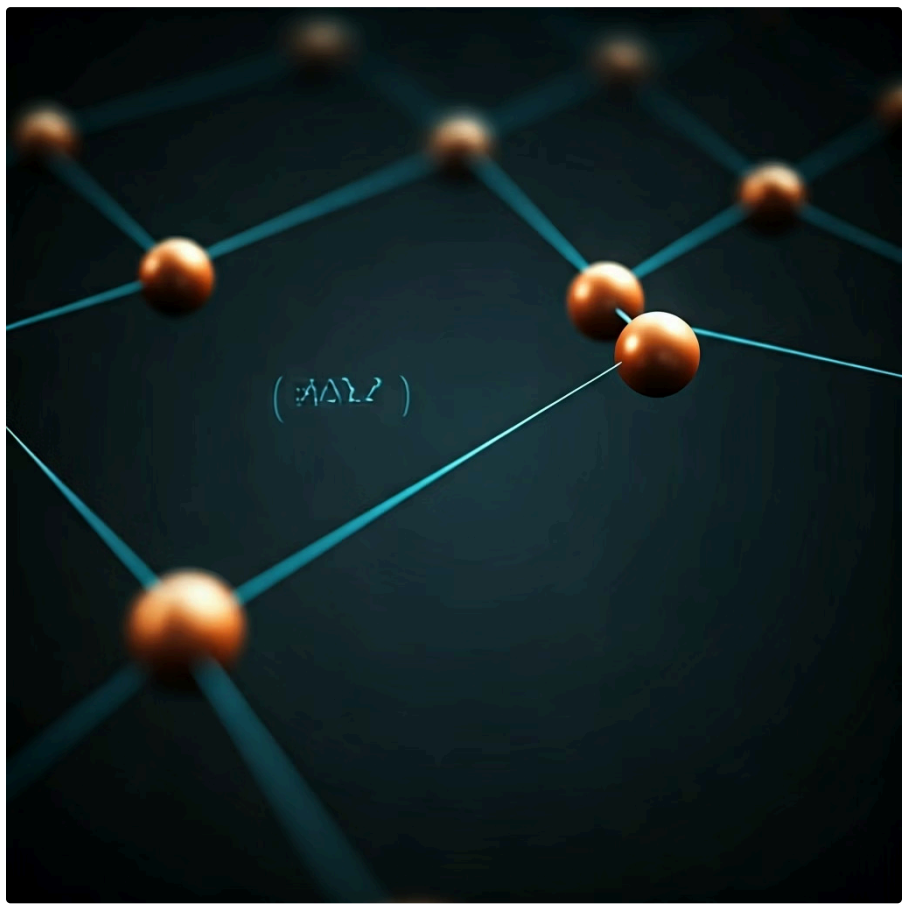
> **Eficiência Mantida**
>
> Complexidade de tempo: $O(1)$ para enqueue e dequeue
>
> Problema de desperdício de memória: **Resolvido!**

A mágica por trás da fila circular é uma operação matemática simples: o **módulo (%)**. Ao calcular a próxima posição dos ponteiros frente e traseira, usamos a fórmula (indice + 1) % tamanho_do_array. Quando indice + 1 for igual ao tamanho_do_array, o resultado do módulo será 0, efetivamente fazendo o ponteiro "pular" do final para o começo. É como um relógio: depois das 11h vem 12h, e depois das 12h, voltamos para 1h, reutilizando os números em um ciclo contínuo.

Com essa abordagem, nossa fila utiliza o espaço de forma muito mais inteligente. Um enqueue pode preencher os espaços no final do array e, quando necessário, continuar a preencher os espaços vazios deixados pelo dequeue no início. Isso mantém a complexidade de tempo de enqueue e dequeue em $O(1)$, mas resolve completamente o problema do desperdício de memória. Essa otimização é fundamental em sistemas de baixo nível, como em *buffers* de sistemas operacionais ou em firmware de dispositivos embarcados, onde cada byte de memória é precioso.

Flexibilidade Máxima com Listas Ligadas

# Quebrando as Correntes: Filas com Listas Ligadas



Arrays, mesmo na sua forma circular, têm uma característica inerente: um **tamanho fixo**. No momento em que criamos o array, precisamos decidir sua capacidade máxima. Mas e se não tivermos a menor ideia de quantos elementos nossa fila precisará armazenar? Pense na fila de processamento de pedidos de um e-commerce durante a Black Friday. O número de pedidos é imprevisível. Definir um tamanho fixo aqui é arriscado: pequeno demais, e o sistema falha; grande demais, e desperdiçamos memória na maior parte do tempo.

É aqui que as **listas ligadas** (*linked lists*), que já exploramos em aulas anteriores, entram em cena como uma alternativa superior em flexibilidade. Uma fila implementada com uma lista ligada não tem um tamanho predefinido. Ela cresce e encolhe dinamicamente, conforme a necessidade. Para isso, só precisamos de dois ponteiros: um para o head (cabeça) da lista, que será a nossa frente, e outro para o tail (cauda), que será a nossa traseira.

### Enqueue

Criamos um novo nó, fazemos o nó tail atual apontar para ele e atualizamos o ponteiro tail

### Dequeue

Pegamos o valor do nó head e atualizamos o ponteiro head para o próximo nó da lista

### Crescimento

A fila pode crescer enquanto houver memória disponível no sistema

A implementação é maravilhosamente intuitiva. Para a operação de enqueue, criamos um novo nó, fazemos o nó tail atual apontar para ele e, em seguida, atualizamos o ponteiro tail para que aponte para este novo nó. Para dequeue, simplesmente pegamos o valor do nó head e atualizamos o ponteiro head para que aponte para o próximo nó da lista. Ambas as operações mantêm a complexidade de tempo em $O(1)$, mas agora sem a preocupação com o limite de capacidade. A fila pode crescer enquanto houver memória disponível no sistema.

Essa flexibilidade é o motivo pelo qual filas baseadas em listas ligadas são frequentemente usadas em aplicações de alto nível, onde a demanda é flutuante e a gestão de memória precisa ser dinâmica.

### A Escolha Certa para Cada Cenário

# A Tomada de Decisão: Array ou Lista Ligada?

Até agora, vimos duas formas eficazes de construir uma fila. Ambas são válidas e possuem complexidade de tempo ideal para as operações fundamentais. Então, como um bom desenvolvedor, como você decide qual usar? A resposta não está em qual é "melhor", mas em qual é **mais adequada** para o problema que você está resolvendo. Essa decisão é um exercício clássico de análise de *trade-offs* (compromissos) entre uso de memória, performance e flexibilidade.

### Array (Circular)

🚄 **Como um trem de passageiros**

- Número fixo de vagões
- Extremamente rápido
- Excelente localidade de cache
- Ideal para capacidade conhecida

### Lista Ligada

🚂 **Como um trem de carga**

- Adiciona/remove vagões dinamicamente
- Sem desperdício de espaço
- Tamanho virtualmente ilimitado
- Ideal para demanda imprevisível

Pense no **array (circular)** como um trem de passageiros com um número fixo de vagões. Ele é extremamente rápido e eficiente para embarcar e desembarcar passageiros, pois todos os vagões estão conectados sequencialmente na memória (o que chamamos de *localidade de cache*). Isso significa que o processador pode acessar os dados mais rapidamente. É a escolha perfeita para cenários onde a capacidade máxima é conhecida ou limitada e a performance bruta é a maior prioridade, como em *buffers* de dados para streaming de vídeo.

A **lista ligada**, por outro lado, é como um trem de carga. Podemos adicionar ou remover vagões (nós) a qualquer momento, conforme a necessidade. Não há desperdício de espaço, e o tamanho é virtualmente ilimitado. O contraponto é que cada vagão (nó) tem um custo extra de memória para o engate (o ponteiro para o próximo nó), e os vagões podem estar espalhados pela memória, o que pode tornar o acesso um pouco mais lento para o processador. É a escolha ideal quando a flexibilidade e a imprevisibilidade do volume de dados são os fatores dominantes.

## Quadro Comparativo: Implementações de Fila

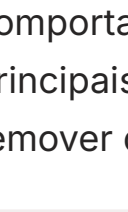
| Característica | Fila com Array (Circular) | Fila com Lista Ligada |
|---|---|---|
| **Gerenciamento de Memória** | Estático (tamanho pré-definido) | Dinâmico (cresce sob demanda) |
| **Complexidade (Enqueue/Dequeue)** | $O(1)$ | $O(1)$ |
| **Uso de Memória** | Sem sobrecarga por elemento | Custo de um ponteiro por elemento |
| **Localidade de Cache** | Excelente (elementos contíguos) | Baixa (elementos podem estar espalhados) |
| **Cenário Ideal** | Capacidade máxima conhecida, alta performance | Capacidade desconhecida, alta flexibilidade |

### Expandindo o Horizonte com Deques

# Quebrando a Regra: Deques, a Fila de Duas Pontas

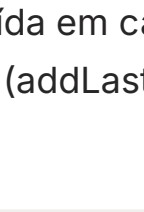
Nossa jornada pelas filas foi guiada pela regra estrita do FIFO. Mas o mundo do desenvolvimento de software está cheio de problemas que exigem um pouco mais de flexibilidade. E se fosse possível, em certas situações, adicionar um item de alta prioridade diretamente no início da fila? Ou talvez processar o último item adicionado imediatamente? Para esses cenários, precisamos de uma estrutura mais versátil, que combine o melhor dos dois mundos.

Essa estrutura é o **Deque (Double-Ended Queue)**, pronunciado "deck". Um deque é uma generalização da fila que permite adicionar e remover elementos de *ambas* as extremidades – da frente e da traseira. Ele se comporta como uma fila suíça: tem uma entrada e uma saída em cada ponta. Isso nos dá quatro operações principais: adicionar no início (addFirst), adicionar no final (addLast), remover do início (removeFirst) e remover do final (removeLast).



**addFirst**

Adiciona elemento no início



**addLast**

Adiciona elemento no final

**removeFirst**

Remove elemento do início

**removeLast**

Remove elemento do final

A analogia aqui pode ser uma pilha de pratos onde você pode adicionar ou remover pratos tanto do topo quanto da base (embora arriscado na vida real!). Um deque pode ser usado para implementar tanto uma fila (usando addLast e removeFirst) quanto uma pilha (usando addLast e removeLast, por exemplo). Sua implementação mais comum é através de uma lista duplamente ligada, que permite a navegação e manipulação em ambas as direções com eficiência $O(1)$. Os deques são incrivelmente úteis em algoritmos mais complexos, como no "sliding window maximum", que busca encontrar o valor máximo em todas as subseções de um array, e na implementação de sistemas de "desfazer" (*undo*) em editores de texto.

Consolidação e Próximos Passos

# Consolidando o Conhecimento: Da Teoria à Prática

Nesta aula, viajamos pelo conceito de organização sequencial. Partimos de uma simples fila de supermercado para entender o princípio fundamental do FIFO. Vimos como podemos construir essa ideia usando arrays, otimizando o uso de espaço com filas circulares, e depois como ganhamos flexibilidade ilimitada com listas ligadas. Por fim, expandimos nossos horizontes com os deques, as filas de duas pontas. A grande lição é que a escolha da estrutura de dados correta é um ato de engenharia: uma análise cuidadosa dos requisitos de memória, velocidade e flexibilidade do seu projeto.

## Em Prática

### Fila Padrão

Ao gerenciar uma sequência de tarefas que devem ser executadas em ordem, uma **fila** é sua primeira escolha.

### Fila Circular

Para um *buffer* de tamanho fixo em um sistema de alto desempenho, uma **fila circular** baseada em array é imbatível.

### Fila com Lista Ligada

Se o número de itens a serem gerenciados é desconhecido e volátil, a segurança e a flexibilidade de uma **fila com lista ligada** evitarão falhas.

### Deque

Quando precisar de uma estrutura que possa adicionar ou remover elementos das duas pontas, como em um histórico de ações, um **deque** é a ferramenta certa.

## Autoavaliação

1. **(Nível Fácil)** Uma aplicação precisa gerenciar o envio de e-mails em um sistema, garantindo que as mensagens sejam enviadas na ordem exata em que foram recebidas. Qual princípio de estrutura de dados é o mais adequado para essa situação? A) LIFO (Last-In, First-Out) B) FIFO (First-In, First-Out) C) Acesso Aleatório D) Hashing
2. **(Nível Médio)** Um desenvolvedor implementou uma fila usando um array padrão (não circular). Após uma série de operações enqueue e dequeue, o ponteiro traseira alcançou o final do array, enquanto o ponteiro frente está no meio. O que acontece se ele tentar enfileirar um novo elemento? A) O elemento será adicionado no início do array. B) O array será redimensionado automaticamente. C) A operação falhará, indicando que a fila está cheia, mesmo havendo espaço livre. D) Todos os elementos existentes serão deslocados para o início do array.
3. **(Banca FCC/2024 - Adaptado)** Ao projetar um sistema de cache para um player de música que precisa armazenar em buffer os próximos segundos de áudio (uma quantidade fixa e conhecida de dados), qual implementação de fila ofereceria a melhor performance em termos de localidade de cache e eficiência de memória, assumindo que as operações de adicionar e remover dados são constantes? A) Fila baseada em lista duplamente ligada. B) Fila baseada em lista simplesmente ligada. C) Fila circular baseada em array. D) Um deque implementado com árvore binária.
4. **(Nível Difícil)** Qual é a principal vantagem de um Deque sobre uma Fila e uma Pilha tradicionais? A) Ele possui complexidade de tempo $O(\log n)$ para inserção. B) Ele consome menos memória que qualquer outra estrutura de dados. C) Ele pode ser usado para implementar tanto uma fila quanto uma pilha eficientemente. D) Ele só pode ser implementado com arrays.
5. **(Discursiva)** Descreva um cenário do mundo real, diferente dos mencionados na aula, onde a utilização de um **Deque** seria mais vantajosa do que uma fila ou uma pilha padrão. Explique o porquê.

### 📋 Gabarito

**1-B, 2-C, 3-C, 4-C**

### 💡 Resposta Sugerida (Discursiva)

Um sistema de histórico de navegação em um browser. O usuário pode tanto avançar (removeLast/addLast da página atual) quanto voltar (removeLast/addFirst para a página anterior) no histórico de abas visitadas. Um deque permite adicionar e remover páginas de ambas as pontas do histórico de forma eficiente.

---

## Próxima Aula

Agora que dominamos como organizar dados que chegam em uma sequência, o próximo desafio é lidar com dados que estão completamente desordenados. Na **Aula 8 – Algoritmos de Ordenação Básicos**, vamos explorar as primeiras técnicas para colocar o caos em ordem, um passo essencial para otimizar buscas e processar informações de forma eficiente.

## Recursos Adicionais

- **VisuAlgo.net**: Para uma visualização interativa e animada de filas, deques e outras estruturas de dados.
- **GeeksforGeeks - "Queue Data Structure"**: Para aprofundar nos conceitos e ver exemplos de código em várias linguagens de programação.

> *NOTA IMPORTANTE*: As informações técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais e a documentação da sua linguagem de programação para verificar implementações e otimizações específicas.