UniEinstein
CENTRO UNIVERSITÁRIO | COLÉGIO

Filas

Docente: Leandro Lopes Taveira
Coordenador: Pedro Ivo Garcia Nunes

Revisão - Pilhas (Stack)

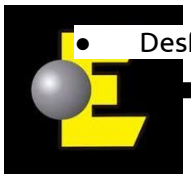
Princípio LIFO

- **L**ast **I**n, **F**irst **O**ut
- O último a entrar é o primeiro a sair.

Operações

- `push()`: Adiciona ao topo
- `pop()`: Remove do topo
- `peek()`: Vê o topo
- `isEmpty()`: Verifica se está vazia

Aplicações



- Desfazer/Refazer, chamadas de função, validação de parênteses.

Introdução a Filas (Queue)

Conceito

- Estrutura de dados linear.
- Princípio **FIFO** (First In, First Out).
    - O primeiro a entrar é o primeiro a sair.

Analogia

- Fila de banco, fila de supermercado, fila de impressão.

Operações Principais da Fila

`enqueue(elemento)`

- Adiciona um elemento ao **final** da fila.

`dequeue()`

- Remove e retorna o elemento do **início** da fila.

`peek()`

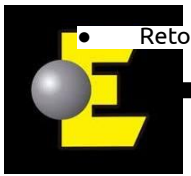
- Retorna o elemento do **início** sem removê-lo.

`isEmpty()`

- Verifica se a fila está **vazia**.

`size()`

- Retorna o **número de elementos** na fila.



Arquitetura do Código

Para uma melhor organização, vamos separar os dados da estrutura:

1. **`PedidoPouso`**: Uma classe que representa os **dados** de um voo.
2. **`No`**: O **elemento estrutural** da nossa lista encadeada. Ele "carrega" um `PedidoPouso`.
3. **`Fila`**: A classe que **gerencia** a fila de `No`s.

Essa separação torna o código mais limpo e reutilizável.

A Classe PedidoPouso

Esta classe serve apenas como um "contêiner" para os dados de cada voo que entra na fila.

```java
public class PedidoPouso {  14 usages
    // Atributos do pedido
    private String numeroVoo;  3 usages
    private String companhiaAerea;  3 usages
    private String tipoAeronave;  3 usages
    private boolean emergencia;  3 usages

    // Método para preencher os dados
    public void preencher(String voo, String companhia,  3 usages
                          String aeronave, boolean isEmergencia) {
        this.numeroVoo = voo;
        this.companhiaAerea = companhia;
        this.tipoAeronave = aeronave;
        this.emergencia = isEmergencia;
    }

    public String getNumeroVoo() { return numeroVoo; }
    public void setNumeroVoo(String numeroVoo) { this.numeroVoo = numeroVoo; }
    public String getCompanhiaAerea() { return companhiaAerea; }
    public void setCompanhiaAerea(String companhiaAerea) { this.companhiaAerea = companhiaAerea; }
    public String getTipoAeronave() { return tipoAeronave; }
    public void setTipoAeronave(String tipoAeronave) { this.tipoAeronave = tipoAeronave; }
    public boolean isEmergencia() { return emergencia; }
    public void setEmergencia(boolean emergencia) { this.emergencia = emergencia; }
}
```

Código - A Classe No

Sua única função é armazenar um PedidoPouso e apontar para o próximo No na fila.

```java
public class No {  1 usage
    private PedidoPouso pedido; // O dado que o nó carrega  2 usages
    public No proximo;          // Ponteiro para o próximo nó  1 usage

    // Método para preencher o nó com um pedido
    public void preencher(PedidoPouso pedido) {  no usages
        this.pedido = pedido;
        this.proximo = null;
    }

    // Método para acessar o pedido dentro do nó
    public PedidoPouso getPedido() {  no usages
        return this.pedido;
    }
}
```

A Classe Fila

A estrutura da classe `Fila` permanece a mesma, com ponteiros para o `inicio` e `fim`. Os nomes dos métodos serão padronizados.

- `inicio`: Aponta para o primeiro `No` da fila.
- `fim`: Aponta para o último `No` da fila.

```
public class Fila {  no usages

    private No head = null;
    private No tail = null;

    /*Your Code*/

}
```

Método enqueue(elemento)

O que ele faz?

Adiciona um `PedidoPouso` ao **final** da fila.

1. Cria um novo `No` para "carregar" o `PedidoPouso`.
2. Se a fila estiver vazia, o novo `No` se torna o `inicio` e o `fim`.
3. Caso contrário, o `No` que era o `fim` passa a apontar para o novo `No`, que se torna o novo `fim`.

```
public void enqueue(PedidoPouso pedido) {/*CODE*/}
```

Código - Método enqueue

```
public void enqueue(PedidoPouso pedido) {  no usages
    No novoNo = new No();
    novoNo.preencher(pedido);
    if (head == null) {
        head = novoNo;
    } else {
        tail.proximo = novoNo;
    }
    tail = novoNo;
}
```

Método dequeue()

O que ele faz?

Remove e retorna o `PedidoPouso` que está no **início** da fila.

1. Verifica se a fila não está vazia.
2. Salva a referência ao `PedidoPouso` do primeiro `No`.
3. Avança o ponteiro `inicio` para o próximo `No` da fila.
4. Se a fila ficar vazia, atualiza também o ponteiro `fim`.

```
public PedidoPouso dequeue() {/*CODE*/}
```

Código - Método dequeue

```
public PedidoPouso dequeue() {  no usages
    if (head == null) {
        return null;
    }
    PedidoPouso pedidoRemovido = head.getPedido();
    head = head.proximo;
    if (head == null) {
        tail = null;
    }
    return pedidoRemovido;
}
```

Método peek()

O que ele faz?

Retorna o `PedidoPouso` do **início** da fila, mas **sem removê-lo**.

- É uma operação de consulta que não altera a estrutura da fila.

```
public PedidoPouso peek() {/*CODE*/}
```

Código - Método peek

```
public PedidoPouso peek() {  no usages
    if (head == null) {
        return null;
    }
    return head.getPedido();
}
```

Método size()

O que ele faz?

Retorna o **número exato de elementos** (pedidos de pouso) que estão atualmente na fila.

- É uma operação de consulta que informa o tamanho atual da fila.
- Útil para monitoramento e relatórios (ex: "Há 5 aeronaves aguardando para pousar").
- Alterações
  a. Adicione um contador na classe Fila, inicie com zero.
  b. faça incremento no *enqueue*, e decremento no *dequeue*
  c. Retorne o valor no size;

```
public int size() {/*CODE*/}
```

Código - Método size()

```java
public int size() {  no usages
    return this.contador;
}
```

```java
public void enqueue(PedidoPouso pedido) {  no usages
    No novoNo = new No();
    novoNo.preencher(pedido);
    if (head == null) {
        head = novoNo;
    } else {
        tail.proximo = novoNo;
    }
    tail = novoNo;
    contador++;
}
```

```java
public PedidoPouso dequeue() {  no usages
    if (head == null) {
        return null;
    }
    PedidoPouso pedidoRemovido = head.getPedido();
    head = head.proximo;
    if (head == null) {
        tail = null;
    }
    contador--;
    return pedidoRemovido;
}
```

Método isEmpty()

O que ele faz?

Verifica se a fila está **vazia**. É a pergunta mais fundamental antes de qualquer operação de remoção ou consulta.

- **Retorna `true` (verdadeiro):** Se não houver nenhum `No` na fila (o `inicio` é `null` ou o `contador` é 0).
- **Retorna `false` (falso):** Se houver um ou mais `No`s na fila.

Código - Método isEmpty()

```java
public boolean isEmpty() {
    if(this.contador == 0){
        return true;
    }
    return false;
}
```

```java
public void enqueue(PedidoPouso pedido) {  no usages
    No novoNo = new No();
    novoNo.preencher(pedido);
    if (isEmpty()) {
        head = novoNo;
    } else {
        tail.proximo = novoNo;
    }
    tail = novoNo;
    contador++;
}
```

```java
public PedidoPouso peek() {  no usages
    if (isEmpty()) {
        return null;
    }
    return head.getPedido();
}
```

```java
public PedidoPouso dequeue() {  no usages
    if (isEmpty()) {
        return null;
    }
    PedidoPouso pedidoRemovido = head.getPedido();
    head = head.proximo;
    if (isEmpty()) {
        tail = null;
    }
    contador--;
    return pedidoRemovido;
}
```

Método showQueue()

O que ele faz?

Ajuda na hora testar o funcionamento de Fila..

- Percorre a Fila e imprime os dados do pedido.
- Pode conter outras informações que queira.

```
public void showQueue(){/*CODE*/}
```

Código - Método showQueue()

```
public void showQueue(){  2 usages
    System.out.println(" --- Inicio Fila --- ");
    No temp = head;
    int count = 1;
    while (temp != null){
        System.out.println(count + " - Voo numero : " + temp.getPedido().getNumeroVoo());
        count++;
        temp = temp.proximo;
    }
    System.out.println(" --- Fim Fila --- ");
}
```

Exercício: Implementar o Método clear()

Objetivo: Adicionar uma nova funcionalidade à classe `Fila`.

Tarefa:

1. Adicione um novo método `public void clear()` à sua classe `Fila`.
2. Este método deve esvaziar a fila completamente, reiniciando os atributos `inicio`, `fim` e `contador` para seus estados iniciais (`null`, `null`, `0`).
3. Na sua classe de simulação, após adicionar alguns voos, chame o método `clear()` e, em seguida, use `isEmpty()` para confirmar que a fila foi limpa.

Exercício: Atendimento Prioritário (Fura-Fila)

Objetivo: Modificar a lógica de enfileiramento para lidar com casos especiais.

Tarefa: Crie um novo método na classe `Fila` chamado

```
public void enqueuePrioritario(PedidoPouso pedido).
```

1. Este método deve receber um `PedidoPouso`.
2. Se o pedido for uma emergência (`emergencia == true`), ele deve ser adicionado no **início** da fila, e não no fim.
3. Se não for uma emergência, ele pode ser adicionado no fim, como o enqueue normal.
4. **Desafio:** Garanta que, se a fila estiver vazia, o `inicio` e o `fim` sejam atualizados corretamente.

Exercício: Relatório de Companhia Aerea

Objetivo: Realizar cálculos sobre os dados contidos na fila.

Tarefa: Adicione um método `public int getNumeroVoosCompanhia(String nomeCompanhia)` à classe `Fila`.

1. Este método deve percorrer a fila e somar o `número de voos da companhia informada`.
2. O método deve retornar o total de voos aguardando para pousar.
3. Na sua aplicação de console, adicione uma opção para exibir este total.

Exercício: Autorizar Pouso por Número do Voo

Objetivo: Criar uma lógica de busca e remoção de um elemento específico.

Tarefa: Implemente o método `public PedidoPouso autorizarPouso(String numeroVoo)` na classe `Fila`.

1. Este método deve procurar por um `PedidoPouso` com o `numeroVoo` especificado.
2. Se o voo for encontrado, ele deve ser **removido** da fila, e o método deve retornar o `PedidoPouso` removido.
3. **Atenção:** Cuidado especial ao remover um elemento que seja o `inicio`, o `fim` ou um elemento no meio da fila. Os ponteiros `proximo` devem ser reajustados corretamente.
4. Se o voo não for encontrado, o método deve retornar `null`.

Exercício: Tempo de Espera Estimado

Objetivo: Aplicar lógica de negócio usando os dados da fila.

Tarefa:

1. Adicione um método `public String getTempoEsperaEstimado(String numeroVoo)` à classe `Fila`.
2. Considere que cada pouso leva 5 minutos.
3. O método deve encontrar a posição do voo na fila (o primeiro está na posição 1, o segundo na 2, etc.).
4. O tempo de espera será `(posição - 1) * 5` minutos.
5. O método deve retornar uma string como "Tempo de espera para o Voo X: 15 minutos." ou "Voo X não encontrado na fila.".

Exercício: Simulador de Múltiplas Pistas de Pouso

Contexto: Um grande aeroporto tem duas pistas, uma para voos normais e outra exclusivamente para emergências.

Tarefa: Crie uma simulação que gerencia duas filas.

- Instancie dois objetos `Fila`: `filaPistaNormal` e `filaPistaEmergencia`.
- O menu principal permitirá ao controlador de tráfego aéreo gerenciar ambas:

1. **`Registrar Chegada de Voo`**: Pede os dados do voo ao usuário. Se o campo `emergencia` for `true`, o `PedidoPouso` é adicionado à `filaPistaEmergencia`. Caso contrário, vai para a `filaPistaNormal`.
2. **`Autorizar Pouso em Pista Normal`**: Executa o dequeue na `filaPistaNormal`.
3. **`Autorizar Pouso em Pista de Emergência`**: Executa o dequeue na `filaPistaEmergencia`.
4. **`Ver Status das Pistas`**: Exibe o `size()` de ambas as filas, mostrando quantos voos aguardam em cada uma.
5. **`Ver Próximos Voos`**: Executa o `peek()` em ambas as filas para mostrar o próximo de cada categoria.
6. **Transferir Voo:** Se a `filaPistaEmergencia estiver vazia, transfere um voo para ela.`
7. **`Sair`**.

Comparativo: Lista, Pilha e Fila

Característica	`List` (Lista)	`Stack` (Pilha)	`Queue` (Fila)
Princípio	Acesso por índice	**LIFO** (Last-In, First-Out)	**FIFO** (First-In, First-Out)
Analogia	Uma lista de compras	Uma pilha de pratos	Uma fila de banco
Operação Principal de Adição	`add(index, elemento)`	`push(elemento)` (Adiciona no topo)	`enqueue(elemento)` (Adiciona no fim)
Operação Principal de Remoção	`remove(index)`	`pop()` (Remove do topo)	`dequeue()` (Remove do início)
Acesso	Em qualquer posição	Apenas no topo	Apenas no início (para ver/remover)
Uso Comum	Armazenar coleções ordenadas	Desfazer/Refazer, chamadas de método	Gerenciar tarefas, filas de atendimento

Análise de Desempenho da Nossa Fila (Big O)

Método	Operação	Notação Big O	Justificativa
`enqueue(elemento)`	Adicionar no fim	**O(1)**	Acesso direto ao `fim` da fila. Não importa o tamanho, a operação é sempre rápida.
`dequeue()`	Remover do início	**O(1)**	Acesso direto ao `inicio` da fila. Apenas um reajuste de ponteiro.
`peek()`	Espiar o início	**O(1)**	Acesso direto ao `inicio` da fila, sem nenhuma alteração.
`isEmpty()`	Verificar se está vazia	**O(1)**	Apenas lemos o valor da variável `contador`.
`size()`	Obter o tamanho	**O(1)**	Apenas lemos o valor da variável `contador`.

Boas Práticas e Dicas

1. **Separe os Dados da Estrutura:** Como fizemos com `PedidoPouso` e `No`, manter os dados em uma classe e a lógica da estrutura em outra torna o código mais limpo, flexível e reutilizável.
2. **Cuidado com a Fila Vazia:** Sempre verifique se a fila está vazia (`isEmpty()`) antes de tentar remover (dequeue) ou espiar (peek) um elemento para evitar erros (`NullPointerException`).
3. **Mantenha um Contador de Tamanho:** Manter uma variável `size` ou `contador` e atualizá-la a cada enqueue/dequeue garante que a verificação de tamanho seja O(1). Evite percorrer a fila para contar os elementos.
4. **No Mundo Real, Use o JCF:** Para projetos profissionais, prefira as implementações do Java Collections Framework (como `LinkedList` ou `ArrayDeque`), que são robustas, testadas e otimizadas. Nossa implementação manual serve para **entender o conceito**, que é o mais importante.

Conclusão e Próximos Passos

que aprendemos hoje?

- O que é uma **Fila** e o seu princípio **FIFO**.
- Como **implementar uma Fila manualmente** em Java usando uma lista encadeada, compreendendo o papel do `inicio`, `fim` e dos ponteiros.
- A importância de **separar responsabilidades** (dados, nós, estrutura).
- A **alta eficiência (O(1))** das operações fundamentais de uma fila bem implementada.
- As diferenças e casos de uso para **Listas, Pilhas e Filas**.

Para onde vamos agora?

Na próxima aula, exploraremos Java Collections Framework (JCF).