UniEinstein
CENTRO UNIVERSITÁRIO | COLÉGIO

Sobre o Trabalho

Docente: Leandro Lopes Taveira
Coordenador: Pedro Ivo Garcia Nunes

gemini.google.com/app/569c0a5c0e9df38f

Implemente um fila encadeada em Java

Com certeza! A implementação de uma fila encadeada em Java é uma ótima forma de entender as estruturas de dados dinâmicas. Uma fila segue a filosofia **FIFO (First-In, First-Out)**, ou seja, o primeiro elemento que entra é o primeiro a sair. A abordagem encadeada é ideal para esse tipo de estrutura, pois permite que a fila cresça e encolha dinamicamente, sem a limitação de um tamanho fixo, como em um array.

A implementação que vou mostrar utiliza classes genéricas (`<T>`), o que a torna mais flexível. Ela pode armazenar qualquer tipo de objeto (inteiros, strings, objetos personalizados, etc.).

---

1. A Classe `No` (ou Node)

Primeiro, precisamos de uma classe para representar cada elemento da nossa fila. Cada nó precisa ter dois componentes:

- O **dado** que ele armazena (o elemento da fila).

Java

```java
public class No<T> {

    private T dado;
    private No<T> proximo;

    public No(T dado) {
        this.dado = dado;
        this.proximo = null; // Inicialment
    }

    // Métodos Getters e Setters
    public T getDado() {
        return dado;
    }
```

Java

```java
public class FilaEncadeada<T> {

    private No<T> inicio; // Referência para o primeiro elemento da fila
    private No<T> fim;    // Referência para o último elemento da fila
    private int tamanho;

    public FilaEncadeada() {
        this.inicio = null;
        this.fim = null;
        this.tamanho = 0;
    }
```

Descrição no documento do trabalho

Conceitos e Estruturas de Dados a Serem Aplicados

Este trabalho exige a aplicação dos seguintes conceitos e estruturas de dados:

- **Programação Orientada a Objetos (POO):** Classes, objetos, encapsulamento (modificadores de acesso `public`/`private`), construtores, métodos.
- **Filas (Queue):** Utilização da implementação manual de uma fila para a fila comum e de prioridade.
- **Pilhas (Stack):** Utilização da implementação manual de uma pilha para o histórico de atendimentos.

Descrição no documento do trabalho

Objetivos de Aprendizagem

Ao final deste trabalho, os alunos deverão ser capazes de:

- Aplicar os princípios da Programação Orientada a Objetos (classes, objetos, agregação) na modelagem de entidades do mundo real.
- Implementar e manipular estruturas de dados lineares como Filas (Queue) e Pilhas (Stack) de forma eficiente.
- Compreender e aplicar o conceito de filas de prioridade.
- Desenvolver algoritmos para gerenciar o fluxo de atendimento e calcular estatísticas básicas.
- Organizar o código de forma modular, legível e bem documentada.
- Trabalhar em equipe, dividindo tarefas e integrando módulos de código.

UniEinstein
CENTRO UNIVERSITÁRIO | COLÉGIO

Java Collections Framework (JCF)

Docente: Leandro Lopes Taveira
Coordenador: Pedro Ivo Garcia Nunes

Revisão

Filas (Queue)

- **O que é?** Uma estrutura de dados linear que segue o princípio **FIFO** (**F**irst **I**n, **F**irst **O**ut).
- **Analogia:** Uma fila do dia a dia, onde a primeira pessoa a entrar é a primeira a ser atendida.

Operações Essenciais

- `enqueue()`**:** 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.

Sobre JCF

O que é?

- Conjunto de **interfaces e classes** para representar e manipular coleções de objetos.
- **Benefícios:** Reusabilidade, eficiência, interoperabilidade.

Principais Interfaces

- `Collection` (base)
- `List` (coleção ordenada, permite duplicatas)
- `Set` (coleção de elementos únicos)
- `Map` (pares chave-valor)
- `Queue` (filas)

Interface List

Características

- Coleção **ordenada** de elementos.
- Permite **duplicatas**.
- Acesso por **índice**.

Implementações Comuns

- **`ArrayList`:**
  - Baseado em **array redimensionável**.
  - **Acesso aleatório:** O(1).
  - **Inserção/Remoção no meio:** O(n).
- **`LinkedList`:**
  - Baseado em **lista duplamente encadeada**.
  - **Acesso aleatório:** O(n).
  - **Inserção/Remoção no início/fim:** O(1).

Exemplo - ArrayList e LinkedList

```
List<String> nomesArrayList = new ArrayList<>();
nomesArrayList.add("Joaozinho");
nomesArrayList.add("Mariazinha");
nomesArrayList.add( index: 0,  element: "Manoelzinho"); // Adiciona na posição 0
System.out.println("ArrayList: " + nomesArrayList);

List<String> nomesLinkedList = new LinkedList<>();
nomesLinkedList.add("Pedrinho");
nomesLinkedList.add("Rosinha");
nomesLinkedList.remove( index: 0); // Remove da posição 0
System.out.println("LinkedList: " + nomesLinkedList);
```

Exercício - Comparando o Desempenho

O objetivo deste exercício é demonstrar a diferença de desempenho entre `ArrayList` e `LinkedList` na prática.

Escreva um programa que:

1. Crie um `ArrayList` de `Integers` e adicione 100.000 elementos em uma ordem crescente (de 0 a 99.999).
2. Crie um `LinkedList` de `Integers` e adicione os mesmos 100.000 elementos.
3. Meça o tempo (em milissegundos ou nanossegundos) que leva para **adicionar um novo elemento no início** de cada lista, 10.000 vezes consecutivas.
4. Imprima o tempo total gasto por cada tipo de lista e compare os resultados.

Dica: Utilize `System.currentTimeMillis()` ou `System.nanoTime()` para medir o tempo de execução de cada operação. Explique nos comentários do seu código o porquê da diferença de desempenho observada.

Exercício - Gerenciador de Tarefas

Crie uma classe chamada `TaskManager` com um `ArrayList` de `Strings` para armazenar uma lista de tarefas. Sua classe deve ter os seguintes métodos:

- `adicionarTarefa(String tarefa)`: Adiciona uma nova tarefa ao final da lista.
- `removerTarefa(int indice)`: Remove a tarefa no índice especificado.
- `listarTarefas()`: Imprime todas as tarefas da lista, mostrando o índice e o nome de cada uma.
- `marcarComoConcluida(int indice)`: Atualiza a tarefa no índice especificado adicionando o sufixo " (Concluída)".

No método `main`, instancie a classe `TaskManager` e utilize os métodos para demonstrar a funcionalidade.

Comparativo ArrayList vs LinkedList

Operação	ArrayList	LinkedList
Acesso por índice	O(1)	O(n)
Inserção no início	O(n)	O(1)
Inserção no fim	O(1)	O(1)
Remoção no início	O(n)	O(1)
Remoção no fim	O(1)	O(1)

Quando usar?

- `ArrayList`: Muitas buscas por índice, poucas inserções/remoções no meio.
- `LinkedList`: Muitas inserções/remoções no início/fim, poucas buscas por índice.

Pilha com JCF

Classe `java.util.Stack`

- Implementa uma pilha LIFO.
- **Métodos:** `push()`, `pop()`, `peek()`, `empty()`, `search()`.
- **Considerada legada:** Estende `Vector` (sincronizada, mais lenta).

Alternativa Preferida (não usaremos)

- Usar a interface `Deque` (Double Ended Queue) com a implementação `ArrayDeque` ou `LinkedList`.
- `ArrayDeque` é mais eficiente para pilhas e filas.

Exemplo Pilha com JCF

```
Stack<String> pilha = new Stack<>();
pilha.push( item: "Item 1"); // Adiciona no início
pilha.push( item: "Item 2");
System.out.println(pilha.pop()); // Remove do início
```

```
Deque<String> pilha = new ArrayDeque<>();
pilha.push( e: "Item 1"); // Adiciona no início
pilha.push( e: "Item 2");
System.out.println(pilha.pop()); // Remove do início
```

Exercício - Invertendo uma String com Stack

O objetivo deste exercício é praticar o conceito **LIFO (Last-In, First-Out)** usando a classe `java.util.Stack`.

Escreva um método estático `inverterString(String texto)` que receba uma string como parâmetro e retorne a mesma string, mas com os caracteres em ordem inversa. O método deve utilizar um `Stack<Character>` para armazenar os caracteres do texto original e depois reconstruir a string invertida.

Exemplo:

- **Entrada:** `"Estrutura"`
- **Saída:** `"aruturtS"`

Exercício - Validador de Parênteses com ArrayDeque

A classe `Stack` é considerada legada. Para este exercício, você usará a alternativa preferida, `ArrayDeque`, que é mais eficiente.

Crie um método chamado `isBalanced(String expressao)` que receba uma string contendo parênteses `()` e chaves `{}`. O método deve retornar `true` se a expressão tiver parênteses e chaves balanceados e `false` caso contrário.

Use um `ArrayDeque` para simular uma pilha. Para cada caractere da string, se for um parêntese ou chave de abertura, adicione-o à pilha. Se for um de fechamento, remova o último elemento da pilha e verifique se ele corresponde.

Exemplos:

- `"{()}"` -> `true`
- `"{())"` -> `false`
- `"()"` -> `true`
- `"({[]})"` -> `true`

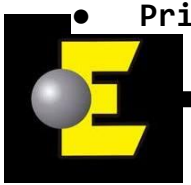
Fila com JCF

Interface `java.util.Queue`

- Define o comportamento de uma fila FIFO.
- **Métodos:**
  - **Inserção:** `add()` (lança exceção), `offer()` (retorna false)
  - **Remoção:** `remove()` (lança exceção), `poll()` (retorna null)
  - **Consulta:** `element()` (lança exceção), `peek()` (retorna null)

Implementações Comuns

- **`LinkedList`:** Implementa `Queue`, bom para filas gerais.
- **`ArrayDeque`:** Implementa `Deque`, pode ser usado como fila ou pilha.
- **`PriorityQueue`:** Fila onde elementos são processados por prioridade.



Exemplo de Fila com JCF

```
Queue<String> fila = new LinkedList<>();
fila.offer( e: "Cliente 1");
fila.offer( e: "Cliente 2");
System.out.println(fila.poll()); // Remove do início
```

Exercício - Fila de Atendimento em um Banco

Simule um sistema de fila de atendimento em um banco. Utilize um `LinkedList` para implementar a fila.

Sua tarefa é criar um programa que:

1. Crie uma fila de `Strings` para representar os clientes.
2. Adicione 5 clientes à fila (`Maria`, `João`, `Ana`, `Pedro`, `Carlos`).
3. Utilize um laço de repetição para remover cada cliente da fila (simulando o atendimento) e imprima a mensagem "Atendendo cliente: [nome do cliente]".
4. Ao final, imprima o status da fila para confirmar que está vazia.

Exercício - Fila de Atendimento em um Banco Triagem

Simule um sistema de fila de atendimento em um banco com triagem por tipo de serviço.

1. Crie uma classe `Cliente` com os atributos `nome` (String) e `servico` (String), onde `servico` pode ser "Saque" ou "Depósito".
2. No método `main`, crie **duas filas** separadas usando `LinkedList`: uma para clientes que precisam de **"Saque"** e outra para clientes que precisam de **"Depósito"**.
3. Crie uma lista inicial de 6 clientes (um array por exemplo) com serviços mistos
4. Percorra a lista e adicione cada um à fila correta, simulando a triagem.
5. Em seguida, simule o atendimento:
    - Primeiro, atenda todos os clientes da fila de **Saque**.
    - Depois, atenda todos os clientes da fila de **Depósito**.
6. Imprima mensagens claras a cada etapa (triagem, início de atendimento de cada fila, etc.).

Dicionários (HashMap)

Problema

Como armazenar e acessar dados quando **não há índice numérico sequencial**?

Exemplos

- **Dicionário de palavras:** palavra → definição
- **Lista telefônica:** nome → telefone
- **Cadastro de alunos:** matrícula → dados do aluno
- **Estoque de produtos:** código → produto

Necessidade

Acesso **rápido** por uma **chave** específica.

Conceito de Dicionário (Map)

Definição

- Estrutura de dados que armazena **pares chave-valor**.
- **Chave:** Única, usada para identificar o valor.
- **Valor:** Dado associado à chave.

Analogia

- **Dicionário de papel:** palavra (chave) → definição (valor)
- **Lista telefônica:** nome (chave) → telefone (valor)

Diferença de List

- **List:** Acesso por **índice numérico** (0, 1, 2, ...)
- **Map:** Acesso por **chave** (qualquer tipo)

Operações Básicas de Dicionário

`put(chave, valor)`

- Insere ou atualiza um par chave-valor.

`get(chave)`

- Retorna o valor associado à chave.

`remove(chave)`

- Remove o par chave-valor.

`containsKey(chave)`

- Verifica se a chave existe.

`size()` e `isEmpty()`

- Tamanho e verificação de vazio.

Interface Map em Java

Definição

- Representa um **mapeamento de chaves para valores**.
- Não permite chaves duplicadas.

Métodos Principais

```java
Map<String, String> mapa = new HashMap<>();
mapa.put("chave", "valor");         // Inserir/Atualizar
String valor = mapa.get("chave");   // Buscar
mapa.remove( key: "chave");              // Remover
boolean existe = mapa.containsKey("chave"); // Verificar existência
int tamanho = mapa.size();          // Tamanho
boolean vazio = mapa.isEmpty();     // Verificar se vazio
```

Classe HashMap

Características

- Implementação de `Map` baseada em **tabela hash**.
- **Não garante ordem** dos elementos.
- Permite **uma chave `null`** e **múltiplos valores `null`**.
- **Complexidade:** O(1) em média para operações básicas.

Quando Usar

- Quando você precisa de **acesso rápido** por chave.
- Quando a **ordem não importa**.

Exemplo Básico - HashMap

```java
Map<String, String> capitais = new HashMap<>();
// Inserir dados
capitais.put("Brasil", "Brasília");
capitais.put("França", "Paris");
capitais.put("Japão", "Tóquio");
// Buscar dados
System.out.println("Capital do Brasil: " + capitais.get("Brasil"));
// Verificar existência
if (capitais.containsKey("França")) {
    System.out.println("França está no mapa!");
}
// Remover dados
capitais.remove( key: "Japão");
// Tamanho
System.out.println("Tamanho: " + capitais.size());
```

Iterando sobre HashMap

```java
Map<String, String> capitais = new HashMap<>();
// Inserir dados
capitais.put("Brasil", "Brasília");
capitais.put("França", "Paris");
capitais.put("Japão", "Tóquio");

for (String pais : capitais.keySet()) {
    System.out.println("País: " + pais);
}
for (String capital : capitais.values()) {
    System.out.println("Capital: " + capital);
}
for (Map.Entry<String, String> entrada : capitais.entrySet()) {
    System.out.println(entrada.getKey() + " -> " + entrada.getValue());
}
```

Exercício 1 e 2

1 - Gerenciador de Contatos: Crie um `HashMap` onde a chave é o nome de uma pessoa (`String`) e o valor é o número de telefone (`String`). Adicione 5 contatos, busque o número de telefone de um deles e, por fim, remova um contato. Imprima o mapa antes e depois das operações para visualizar as mudanças.

2 - Contador de Frequência de Palavras: Escreva um programa que leia uma frase e utilize um `HashMap` para contar a frequência de cada palavra. O mapa deve ter a palavra (`String`) como chave e a contagem (`Integer`) como valor. Ao final, itere sobre o mapa e imprima cada palavra com sua respectiva frequência.

Exercício 3 e 4

3* - Catálogo de Produtos: Crie uma classe `Produto` com os atributos `id` (`int`), `nome` (`String`) e `preco` (`double`). Em seguida, crie um `HashMap<Integer, Produto>`. Adicione 3 produtos ao mapa, use o `id` como chave. Crie um método para buscar um produto pelo seu ID e imprimir seus detalhes.

4 - Inversor de Chave-Valor: Crie um `HashMap<String, Integer>` com 5 entradas. Em seguida, crie um novo `HashMap<Integer, String>` que contenha os mesmos dados, mas com as chaves e valores invertidos. Itere sobre o primeiro mapa para popular o segundo e imprima o resultado.

Exercício 5 e 6

5* - `Map` + `List`: Agrupador de Pessoas por Cidade: Crie uma classe `Pessoa` com os atributos `nome` e `cidade`. Crie um `ArrayList<Pessoa>` e adicione 10 pessoas, com algumas cidades se repetindo. Em seguida, crie um `HashMap<String, List<Pessoa>>` para agrupar as pessoas por cidade. Onde a chave é a cidade (`String`) e o valor é uma `List<Pessoa>`. Itere sobre o `ArrayList` original e adicione cada pessoa à `List` correta dentro do mapa.

6 - `Map` + `Queue`: Processador de Tarefas por Prioridade: Simule um sistema de processamento de tarefas em que cada tipo de tarefa tem sua própria fila. Crie um `HashMap<String, PriorityQueue<String>>`. As chaves do mapa devem ser os tipos de tarefa (`"Desenvolvimento"`, `"Marketing"`, `"Financeiro"`) e os valores devem ser uma `PriorityQueue` para as tarefas. Adicione tarefas a cada fila e, em seguida, processe uma tarefa por vez de cada tipo, sempre removendo a que tem a maior prioridade.

Exercício 6 e 8

7* - `Map` + `Stack`: Editor de Código com Histórico por Arquivo: Crie um programa que simula um editor de código com um sistema de "desfazer". Utilize um `HashMap<String, Stack<String>>` onde a chave é o nome do arquivo (`"main.java"`, `"index.html"`) e o valor é uma `Stack` que armazena as versões do conteúdo do arquivo. Implemente os métodos `abrirArquivo(String nome)`, `salvarVersao(String nome, String conteudo)` (que adiciona à pilha) e `desfazer(String nome)` (que remove da pilha).

8 - `Map` + `List`: Filtragem Eficiente de Dados: Imagine que você tem um `ArrayList` de todos os IDs de usuários de uma plataforma (que pode conter milhões de IDs). Você recebe um `HashMap` contendo os IDs e nomes dos usuários que estão ativos. Escreva um programa que gere um novo `ArrayList` contendo apenas os IDs dos usuários que estão na lista de todos os usuários **e** também no mapa de usuários ativos. Compare o desempenho de usar `map.containsKey(id)` (O(1)) vs. `list.contains(id)` (O(n)) para realizar essa filtragem.

Considerações finais

O que aprendemos hoje?

- A interface `Map` é ideal para armazenar dados em pares de **chave-valor**, uma alternativa poderosa quando o acesso por índice numérico não é a melhor solução.
- A classe `HashMap` é a implementação mais comum de `Map`, baseada em **tabelas hash**, que oferece acesso e manipulação de dados em tempo **O(1) em média**.
- A performance do `HashMap` depende da função hash e de como ele lida com **colisões**.
- Lembre-se: use `HashMap` quando a **velocidade** de busca for a prioridade e a **ordem** dos elementos não for importante.
- As estruturas de dados que vimos (List, Queue, Stack e Map) resolvem problemas diferentes. Saber quando usar cada uma é a chave para uma programação eficiente.