Sprint 1

Dynamic Programming

Aplicativo Mobile para Mecânico

Este trabalho foi realizado em conjunto com a entrega de Domain-Driven Design (DDD) Java. Todas as estruturas e algoritmos foram implementados em Java.

Desenvolvido por:

Lucas Garcia - RM554070

Felipe Santana - RM554259

Enzo Barbeli - RM554272

1. Escolha de Estrutura de Dados

Atributos e Estruturas de Dados da classe **Estoque.**

```
//Dynamic Programming
private List<Produto> produtosEmEstoque; // Listar produtos disponiveis
private Map<String, Produto> produtosPorNome; // Busca por nome
private List<Produto> produtosOrdenadosPorPreco; // Cotação de preço
private Queue<Produto> carrinhoDeCompras; // Processo de compra
  You, 2 seconds ago • Uncommitted changes
```

1.1 private List<Produto> produtosEmEstoque:

- **Tipo de Estrutura**: List (mais especificamente, um ArrayList).
- **Complexidade de Ordenação**: O(n log n) ao usar sort() com um Comparator.
- **Complexidade de Busca**: O(1) para acesso por índice.
- **Complexidade de Inserção:** O(1) quando adicionando no final (amortizado) e O(n) para remoção e busca.
- **Justificativa**: A estrutura produtosEmEstoque armazena a lista de produtos disponíveis no estoque. Como frequentemente precisamos percorrer ou atualizar a lista, o ArrayList oferece acesso rápido e é uma escolha apropriada para armazenar produtos em uma lista ordenada, permitindo adicionar, remover e percorrer a coleção.

1.2 Map<String, Produto> produtosPorNome:

- **Tipo de Estrutura**: Map (HashMap). O HashMap é uma estrutura de dados versátil e eficiente para armazenar pares de chave-valor, sendo muito útil em diversas aplicações onde o acesso rápido aos dados é essencial.
- **Complexidade de Busca**: O(1) no caso de HashMap.
- **Complexidade de Inserção**: O(1) no caso de HashMap.
- **Justificativa**: Usamos HashMap para a busca de produtos por nome, pois oferece acesso rápido e eficiente a partir de uma chave, tornando a busca direta e sem necessidade de percorrer todos os elementos.

1.3 List<Produto> produtosOrdenadosPorPreco:

- **Tipo de Estrutura:** ArrayList.
- **Complexidade de Ordenação**: O(n log n) ao usar sort() com um Comparator.
- **Complexidade de Busca**: O(1) para acesso por índice.
- **Complexidade de Inserção**: O(1) quando adicionando no final (amortizado) e O(n) para remoção e busca.
- **Justificativa**: A lista produtosOrdenadosPorPreco facilita o acesso sequencial e a ordenação. Como precisamos exibir a lista em ordem de preço, o uso de sort() ao atualizar a lista é eficiente para organizar os produtos de uma só vez.

1.4 Queue<Produto> carrinhoDeCompras:

- **Tipo de Estrutura:** Queue (Fila). É uma estrutura de dados que segue o princípio FIFO (First-In, First-Out), onde o primeiro elemento a ser inserido é o primeiro a ser removido.
- **Complexidade de Inserção**: O(1) no final da fila.
- **Complexidade de Remoção**: O(1) no início da fila.
- **Justificativa**: Queue permite adicionar e remover produtos em uma sequência FIFO, o que é útil para simular o processo de compra e processamento do carrinho.

2. Análise de Algoritmos

Algoritmos de busca e ordenação da classe **Estoque.**

Métodos

```
public void listarProdutos() {···

private void atualizarProdutosOrdenadosPorPreco() {···

public Produto buscarProdutoPorNome(String nome) {···

public void exibirProdutosOrdenadosPorPreco() {···

public void adicionarAoCarrinho(Produto produto) {···

public void processarCompra() {···

public void enderecoEstoque(){···
```

2.1 Método listarProdutos():

```
public void listarProdutos() {
    You, 37 seconds ago | 1 author (You)
    produtoService.buscarProdutos(new Callback<List<ProdutoJson>>() {
        @Override
        public void onResponse(Call<List<ProdutoJson>> call, Response<List<ProdutoJson>> response) {
            if (response.isSuccessful() && response.body() != null) {
                List<ProdutoJson> listaDeJson = response.body();
                for (ProdutoJson produtoJson : listaDeJson) {
                    Produto produto = new Produto(produtoJson.getId(), produtoJson.getTitle(), produtoJson.getPrice(), produtoJson.getDescription(), produto
                    produtosEmEstoque.add(produto);
                    produtosPorNome.put(produto.getNome(), produto);
                }

                atualizarProdutosOrdenadosPorPreco();

                enderecoEstoque();
                System.out.println(x:"Produtos em Estoque: ");
                for (Produto produto : produtosEmEstoque) {
                    produto.exibirInfo();
                }

            } else {
                System.out.println(x:"Falha ao obter produtos: Resposta sem sucesso.");
            }
        }

        @Override
        public void onFailure(Call<List<ProdutoJson>> call, Throwable t) {
            System.out.println("Erro ao obter produtos: " + t.getMessage());
        }
    });
}
```

Funcionamento e Complexidade

Faz uma chamada ao ProdutoService para buscar os produtos de uma fonte externa (via API) e adicioná-los à lista produtosEmEstoque. Após obter a lista de produtos, exibe o endereço do estoque e as informações de cada produto.

- **Callback onResponse**: Este callback é chamado quando a resposta da API é recebida.
    - **Complexidade de Criação de Objetos**: **O(n)**, onde n é o número de produtos recebidos.
    - **Processamento de Resposta**: Para cada **ProdutoJson** recebido, criamos um novo objeto Produto e o adicionamos a **produtosEmEstoque**. Isso tem

complexidade linear, pois precisamos iterar sobre cada produto retornado pela API.
    - **Exibição de Produtos**: Exibimos cada produto usando **produto.exibirInfo()**, o que também tem complexidade **O(n)**, onde n é o número de produtos na lista **produtosEmEstoque**.

- **Callback onFailure**: Este callback é acionado caso haja um erro ao se conectar à API ou ao receber a resposta. Ele exibe uma mensagem de erro com o detalhe da exceção (t), e sua complexidade é constante O(1), pois não realiza operações sobre a lista.

Justificativa do Uso do Callback

A utilização de um Callback permite que o método listarProdutos seja assíncrono, ou seja, ele não bloqueia a execução do aplicativo enquanto aguarda a resposta da API. Quando os dados de produtos são retornados, o callback onResponse é executado, processando os produtos recebidos e adicionando-os a produtosEmEstoque. Esse design assíncrono torna o aplicativo mais responsivo e permite que o usuário continue utilizando outras funcionalidades enquanto os dados são carregados em segundo plano.

2.2 Método atualizarProdutosOrdenadosPorPreco():

```
private void atualizarProdutosOrdenadosPorPreco() {
    produtosOrdenadosPorPreco = new ArrayList<>(produtosEmEstoque);
    produtosOrdenadosPorPreco.sort(Comparator.comparing(Produto::getPreco));
}
```

Funcionamento e Complexidade

Atualiza a lista de produtos ordenados por preço, criando uma nova lista com os produtos em estoque e ordenando-os pelo preço (**O(n log n)** de complexidade). A lista `**produtosOrdenadosPorPreco**` é ordenada de acordo com o preço dos produtos utilizando `**Comparator.comparing(Produto::getPreco)**`.

- **Comparator.comparing(Produto::getPreco)**: É uma maneira de criar um Comparator (comparador) em Java, que será usado para ordenar uma lista de objetos com base no valor retornado pelo método getPreco() da classe Produto.

- **comparing()**: Método da classe Comparator que cria um comparador a partir de uma função que extrai a chave de comparação de um objeto. Nesse caso, ele vai criar um comparador que usa o valor retornado pelo método getPreco() para comparar dois objetos Produto. Em vez de escrever algo como produto -> produto.getPreco(), você usa Produto::getPreco, o que faz a mesma coisa, mas de forma mais limpa.

Complexidade: O(n log n) para a ordenação dos produtos.

Justificativa: A complexidade do sort() em uma lista é O(n log n), o que torna o método eficiente para ordenar os produtos por preço.

2.3 Método buscarProdutoPorNome(String nome):

```
public Produto buscarProdutoPorNome(String nome) {
    return produtosPorNome.get(nome);
}
```

Funcionamento e Complexidade

Retorna o produto associado ao nome fornecido, utilizando o mapa **produtosPorNome**. A busca no mapa é realizada em **O(1)**, já que a chave (nome) permite acesso direto ao valor (produto).

Complexidade: O(1).

Justificativa: Usando HashMap, a busca por nome é muito rápida e eficiente, com complexidade constante para acessar um produto específico. O método **get()** do Map é usado para buscar um valor associado a uma chave. Neste caso, a chave é o nome do produto, e o valor associado a essa chave é o objeto Produto.

2.4 Método exibirProdutosOrdenadosPorPreco():

```
public void exibirProdutosOrdenadosPorPreco() {
    System.out.println(x:"Produtos ordenados por preço:");
    for (Produto produto : produtosOrdenadosPorPreco) {
        produto.exibirInfo();
    }
}
```

Funcionamento e Complexidade

Exibe todos os produtos ordenados por preço. A exibição segue a ordem crescente de preços, utilizando a lista **produtosOrdenadosPorPreco**.

Complexidade: O(n).

Justificativa: Este método apenas itera sobre a lista ordenada, com complexidade linear, tornando a exibição de produtos ordenados por preço eficiente.

2.5 Método adicionarAoCarrinho(Produto produto):

```java
public void adicionarAoCarrinho(Produto produto) {
    carrinhoDeCompras.add(produto);
    System.out.println("Produto adicionado ao carrinho: " + produto.getNome());
}       You, 30 minutes ago • Uncommitted changes
```

Funcionamento e Complexidade

Adiciona o produto ao carrinho de compras (representado por uma fila **carrinhoDeCompras**). A adição à fila é realizada em **O(1)**, pois as operações de inserção e remoção em uma fila são eficientes.

Complexidade: O(1).

Justificativa: Como a Queue usa LinkedList para gerenciar o carrinho, a inserção no final da fila é direta e de complexidade constante.

2.6 Método processarCompra():

```java
public void processarCompra() {
    System.out.println(x:"Processando compra...");
    while (!carrinhoDeCompras.isEmpty()) {
        Produto produto = carrinhoDeCompras.poll();
        if (produtosEmEstoque.contains(produto)) {
            produtosEmEstoque.remove(produto);
            System.out.println("Comprando: " + produto.getNome());
        }else{
            System.out.println("Produto não encontrado em estoque: " + produto.getNome());
        }       You, 34 minutes ago • Uncommitted changes
    }
}
```

Funcionamento e Complexidade

Processa a compra, removendo os produtos do carrinho e do estoque. Para cada produto no carrinho, é verificado se ele está presente no estoque. Se o produto for encontrado, ele é removido do estoque e comprado. Esse processo envolve a remoção de produtos da lista **produtosEmEstoque** e é realizado em **O(n)** no pior caso, onde **n** é o número de produtos no estoque, já que o método **contains()** da lista percorre os elementos da lista.

Complexidade para Remover do carrinhoDeCompras: O(m), onde m é o número de itens no carrinho, pois cada chamada de poll() remove o primeiro elemento, com complexidade O(1).

Complexidade de contains e remove em produtosEmEstoque: O(n) cada, onde n é o número de produtos no estoque.

Complexidade Geral: O(m * n), pois para cada item no carrinho, fazemos uma verificação e remoção no estoque, o que pode ser otimizado se tivermos uma estrutura de dados que permita acesso direto aos produtos.

Método poll(): remove o primeiro produto da fila, o que simula a ação de processar o primeiro produto da lista de compras do cliente. Em termos de complexidade, a operação de remoção de um item de uma fila (poll()) em uma implementação típica de Queue (como LinkedList) é O(1), ou seja, uma operação constante.

Justificativa: A fila é utilizada porque o processo de compra segue a ordem em que os produtos foram adicionados ao carrinho, ou seja, é um comportamento **First-In-First-Out (FIFO)**. Isso é adequado para o contexto de compra, pois os produtos são comprados na ordem em que o cliente os coloca no carrinho.

Justificativa da Ordenação por Preço

Ordenar por preço é eficiente porque as comparações são feitas entre valores numéricos, que são operações rápidas.

Em comparação com ordenar por **nome** (que envolve comparações lexicográficas) ou **categoria** (que pode ser qualitativa e exigir manipulações adicionais), ordenar por preço tende a ser mais eficiente e prático, especialmente em uma loja de autopeças onde o preço é frequentemente um critério de decisão central.

Além disso, ordenar por preço facilita outras operações, como **comparação de custo**, **filtros de faixa de preço** e até **cálculos de cotação** para os clientes.

3. Estruturas Dinâmicas e Otimização

Aqui estão algumas sugestões de estruturas de dados dinâmicas e soluções de programação dinâmica que podem ser implementadas.

Estrutura de Dados para Otimizar a Busca e Ordenação de Produtos

Estrutura de Dados Sugerida:

Árvore Balanceada (Red-Black Tree / AVL Tree): Para otimizar a busca e ordenação de produtos por preço, podemos substituir a **ArrayList** por uma árvore balanceada, como a **TreeMap** ou **TreeSet** em Java. Essas estruturas mantêm os elementos ordenados e garantem operações de busca, inserção e remoção com complexidade de tempo **O(log n)**.

Justificativa:

- **Busca rápida e ordenação automática**: Diferente de uma lista (ArrayList), que exige uma ordenação explícita (O(n log n)) a cada atualização, a árvore balanceada mantém a lista de produtos sempre ordenada e realiza buscas mais eficientes com O(log n).
- **Eficiência em tempo de execução**: Operações como **adicionar**, **remover** ou **buscar produtos por preço** se tornam muito mais eficientes do que em listas ordenadas, onde seria necessário reordenar a lista a cada modificação.

Complexidade de inserção e busca: **O(log n)**, já que a árvore balanceada mantém os elementos ordenados automaticamente.

Estrutura de Dados para Otimizar o Carrinho de Compras

Estrutura de Dados Sugerida:

Deque (Double-Ended Queue): Em vez de usar uma Queue, podemos usar um Deque (ou seja, LinkedList em Java) para o carrinho de compras. Isso oferece a flexibilidade de adicionar e remover produtos tanto do início quanto do fim da fila.

Justificativa:

- **Operações de remoção rápida de itens**: Com um Deque, você pode facilmente adicionar ou remover produtos do carrinho de compras em ambas as extremidades. Isso pode ser útil para casos onde você precisa adicionar produtos com base em diferentes prioridades ou remover produtos sem ter que percorrer toda a fila.
- **Desempenho melhorado**: Em uma Queue, você só consegue remover elementos da frente da fila, o que pode ser uma limitação. No caso do carrinho, o Deque permite maior flexibilidade.

Complexidade de inserção e remoção: **O(1)** para operações no início ou no final.

Solução de Programação Dinâmica para Otimizar Busca de Produtos Similares

Objetivo da Solução:

É possível implementar uma **solução de programação dinâmica** para otimizar a busca de **produtos similares** com base em atributos como **categoria** ou **faixa de preço**. Uma

abordagem interessante seria o uso de **memoization** para armazenar resultados de buscas anteriores.

Justificativa:

Ao realizar buscas frequentes de produtos semelhantes, como sugerir peças com preços próximos ou produtos dentro de uma categoria específica, cada nova consulta pode ser lenta. Usando programação dinâmica e **memoization**, podemos armazenar os resultados das buscas anteriores para evitar refazer o trabalho. Isso pode reduzir o tempo de busca ao buscar por produtos já consultados anteriormente.

Estrutura de Dados e Implementação:

- Podemos usar um **Map** para armazenar os resultados das buscas anteriores com chave-valor, onde a chave seria um parâmetro de consulta (por exemplo, uma faixa de preço ou categoria) e o valor seria a lista de produtos correspondentes.

Complexidade da primeira busca: **O(n)**, onde n é o número de produtos em estoque.

Complexidade das buscas subsequentes (usando cache): **O(1)**, já que estamos apenas acessando o mapa.