ChatGPT

Exercícios de Programação com Collections Framework em Java

Seção 1: HashSet (5 questões)

1. **Conjunto de alunos únicos:** Implemente um programa que gerencia um cadastro de alunos usando `HashSet`. Inicialmente, adicione nomes de alunos (alguns repetidos) no conjunto. Em seguida, tente adicionar um aluno que já existe e observe o que acontece. Use métodos como `add` (para inserir alunos) e `contains` (para verificar se um aluno já está cadastrado) para evitar duplicações. Por fim, remova um aluno específico do `HashSet` usando `remove` e exiba todos os alunos restantes no conjunto.

2. **Eliminando duplicatas de uma lista:** Suponha que você tenha uma `List<String>` com vários endereços de e-mail, incluindo duplicatas. Crie um método que recebe essa lista e retorna uma nova lista sem endereços duplicados, preservando apenas uma ocorrência de cada e-mail. Dica: utilize um `HashSet` internamente para filtrar os valores únicos (por exemplo, adicionando todos os e-mails no `HashSet` e depois convertendo de volta para lista). Mostre como usar os métodos `addAll` ou o construtor do `HashSet` que recebe uma coleção para facilitar essa operação.

3. **Verificação de presença em coleção:** Imagine um sistema de sorteio que gera números aleatórios de 0 a 99. Utilize um `HashSet<Integer>` para armazenar os números sorteados, garantindo que nenhum número seja repetido. A cada novo número gerado, verifique com `contains` se ele já foi sorteado; caso não, adicione-o ao conjunto. Faça isso até o conjunto ter 10 números únicos sorteados e então exiba o conjunto resultante. (Dica: o tamanho do conjunto pode ser obtido com `size()` para controlar o loop).

4. **Interseção de conjuntos:** Considere dois conjuntos `HashSet<String>` representando alunos inscritos em "Curso A" e alunos inscritos em "Curso B". Popule cada conjunto com alguns nomes (incluindo alguns nomes em comum entre A e B). Em seguida, escreva um código que encontre a **interseção** desses conjuntos, ou seja, quais alunos estão matriculados *em ambos* os cursos. Utilize métodos como `retainAll` (ou, alternativamente, itere com `contains`) para obter a interseção e exiba o resultado.

5. **Ordem não garantida:** Insira vários valores em um `HashSet` (por exemplo, nomes de frutas ou cidades) e itere sobre o conjunto imprimindo cada elemento. Observe a ordem em que os elementos são exibidos. Escreva uma breve explicação do resultado, indicando que a classe `HashSet` **não garante ordem** de inserção nem ordem alfabética – os elementos aparecem em uma ordem aparentemente "aleatória" devido à implementação baseada em tabelas de espalhamento (hash). Comente também sobre a eficiência dessas operações (inserção, busca e remoção) em um `HashSet` em termos de complexidade.

Seção 2: LinkedHashSet (5 questões)

1. **Filtrando duplicatas com ordem preservada:** Suponha que você tenha uma lista de produtos em uma determinada ordem, podendo haver itens repetidos. Crie um método que receba, por exemplo, uma `List<String>` de nomes de produtos e retorne uma lista de produtos únicos **na mesma ordem** em que aparecem pela primeira vez. Para isso, utilize um `LinkedHashSet` para eliminar duplicatas enquanto mantém a ordem de inserção. Em seguida, converta o `LinkedHashSet` de volta para uma lista. Demonstre o código e mostre que, se a lista original era, por exemplo, `[A, B, C, A, D, B]`, o resultado final será `[A, B, C, D]` seguindo a ordem original das primeiras ocorrências.

2. **Lista de espera com ordem de inscrição:** Implemente um programa para gerenciar a lista de espera de pacientes em uma clínica, usando `LinkedHashSet<String>` para armazenar os nomes na ordem em que foram adicionados. O programa deve permitir: (a) adicionar um novo paciente à lista (usando `add` – se o paciente já estiver presente, ignorá-lo para evitar duplicatas), (b) remover um paciente que foi atendido ou desistiu (usando `remove`), e (c) listar todos os pacientes na ordem de chegada (iterando sobre o `LinkedHashSet` com um loop **for-each**). Teste o programa adicionando alguns nomes (incluindo um duplicado para verificar que não entra repetido), removendo um nome, e depois exibindo a lista final em ordem.

3. **Comparando HashSet vs LinkedHashSet:** Escreva um código que insere a sequência de valores `"1", "2", "3", "4", "5"` tanto em um `HashSet` quanto em um `LinkedHashSet` (nessa ordem exata de inserção). Em seguida, itere sobre cada conjunto e imprima os valores encontrados. Qual é a diferença na saída? Documente a ordem dos elementos impressos para o `HashSet` (que pode não seguir a ordem de inserção) em contraste com o `LinkedHashSet` (que deve exibir os valores exatamente na ordem inserida: 1, 2, 3, 4, 5). Explique brevemente o motivo dessa diferença.

4. **Remoção e re-inserção:** Suponha que estamos usando um `LinkedHashSet` para rastrear participantes de um jogo na ordem em que entram. Todos os participantes começam em um `LinkedHashSet` na ordem de chegada. Em certo momento, um participante sai do jogo e posteriormente retorna. Faça um experimento: adicione alguns nomes em um `LinkedHashSet`, remova um deles (por exemplo, o primeiro inserido) usando `remove`, e depois adicione o mesmo nome novamente. Itere e imprima o conjunto resultante. Em qual posição o nome reinserido aparece? Explique que, ao remover e adicionar novamente, o elemento é tratado como novo na estrutura, passando a figurar **no final** do conjunto devido à ordem de inserção preservada.

5. **União preservando ordem:** Crie dois conjuntos `LinkedHashSet`: um contendo filmes vistos por Ana e outro contendo filmes vistos por Bruno. Popule-os com alguns títulos de filmes, onde alguns filmes estão em ambos os conjuntos (filmes que ambos viram) e outros são exclusivos de cada pessoa. Agora, construa um terceiro conjunto `LinkedHashSet` que represente *todos* os filmes que pelo menos um dos dois viu (a **união** dos conjuntos). Utilize `addAll` para unir os conjuntos de Ana e Bruno. Exiba o conjunto resultante e confirme que: (a) ele não contém duplicatas, e (b) a ordem de listagem dos filmes corresponde à ordem em que eles foram inseridos inicialmente (todos os filmes de Ana, na ordem dela, seguidos dos filmes de Bruno que não estavam já presentes).

Seção 3: TreeMap (5 questões)

1. **Agenda telefônica ordenada:** Utilizando um `TreeMap<String, String>`, crie uma agenda telefônica simples em que a chave seja o nome de uma pessoa e o valor seja seu número de telefone. Insira pelo menos cinco contatos na agenda em ordem aleatória de nome (ou seja, não inseridos em ordem alfabética). Em seguida, use `keySet()` ou `entrySet()` para iterar e imprimir todos os contatos com seus números. Note que a saída deve exibir os contatos ordenados alfabeticamente pelo nome (chave) automaticamente. Além disso, use `get` para buscar o telefone de um nome específico e `remove` para excluir um contato da agenda. Mostre cada operação funcionando.

2. **Identificadores numéricos ordenados:** Imagine um sistema de registro de funcionários onde cada funcionário tem um ID numérico. Utilize `TreeMap<Integer, String>` para mapear IDs de funcionários (chave) para o nome do funcionário (valor). Insira vários pares ID-nome fora de ordem (por exemplo, IDs aleatórios). Demonstre que o `TreeMap` ordena automaticamente as entradas pelo ID (ordem crescente numérica). Use métodos como `firstKey()`, `lastKey()` para obter o menor e o maior ID registrados e imprima esses valores, confirmando a ordenação. Itere pelo mapa (por exemplo, com `entrySet()` ou `forEach`) e exiba os IDs e nomes na sequência ordenada de IDs.

3. **Contagem de palavras ordenada:** Escreva um método que receba uma frase (String) e utilize um `TreeMap<String, Integer>` para contar a frequência de cada palavra na frase. Ignorando maiúsculas/minúsculas e pontuação, divida a frase em palavras e para cada palavra use `containsKey` ou `get` / `put` para contabilizar ocorrências (incrementando o valor no mapa). Ao final, itere sobre o `TreeMap` e imprima cada palavra seguida de sua contagem. Verifique que as palavras aparecerão em ordem alfabética graças ao `TreeMap`. *Exemplo:* para a frase "Bolo de chocolate e bolo de cenoura", a saída deve listar primeiro "bolo = 2", depois "cenoura = 1", "chocolate = 1", "de = 2", "e = 1" (ordenando por palavra).

4. **Ordenação personalizada (Comparator):** Por padrão, o `TreeMap` ordena as chaves na ordem natural (crescente). Suponha que você queira um dicionário que ordene as palavras em ordem alfabética reversa (da Z à A). Crie um `TreeMap<String, String>` passando um `Comparator` personalizado (por exemplo, `Collections.reverseOrder()`) para inverter a ordem de classificação das chaves. Insira algumas entradas (por exemplo, código de país como chave e nome do país como valor: "BR"->"Brasil", "US"->"Estados Unidos", "AR"->"Argentina", etc.). Em seguida, itere sobre o mapa e confirme que a ordem das chaves é decrescente (por exemplo, "US" vem antes de "BR", que vem antes de "AR", se usarmos as chaves acima). Explique como o comparador alterou a ordenação padrão do `TreeMap`.

5. **Subconjunto de dados (subMap):** Considere um `TreeMap<Integer, String>` que armazena registros de vendas, onde a chave é o número do pedido (order ID) e o valor é o nome do cliente. Popule o mapa com, digamos, 10 entradas (IDs de pedidos não necessariamente sequenciais, mas o `TreeMap` os manterá ordenados). Agora suponha que você queira listar apenas os pedidos em um determinado intervalo de IDs (por exemplo, de 1000 a 2000). Utilize os métodos `subMap` ou `tailMap` / `headMap` do `TreeMap` para obter uma visão (view) do mapa original restrita ao intervalo desejado e então iterar sobre essa visão imprimindo somente os pedidos nesse intervalo. Demonstre o código e explique como o `TreeMap` facilita a obtenção de faixas ordenadas de dados sem precisar filtrar manualmente.

Seção 4: Hashtable (5 questões)

1. **Dicionário de capitais (uso básico de Hashtable):** Crie um programa que utiliza `Hashtable<String, String>` para armazenar pares **País -> Capital**. Insira, por exemplo, os seguintes pares: Brasil->Brasília, França->Paris, Japão->Tóquio, Austrália->Camberra, Canadá->Ottawa. Em seguida, mostre como recuperar a capital de um país específico usando `get` (por exemplo, obtenha a capital do Brasil) e como remover uma entrada usando `remove` (por exemplo, remova o país Japão do dicionário). Finalmente, use um loop (por exemplo, `for-each` em `entrySet()`) para imprimir todas as Pares País=Capital restantes. Observe que a ordem de saída não é garantida para `Hashtable` (pode parecer arbitrária, semelhante ao `HashMap`), e explique esse comportamento.

2. **Proibido nulos:** Faça um experimento para entender a restrição de nulos em `Hashtable`. Tente inserir explicitamente uma chave nula e/ou um valor nulo em um `Hashtable` – por exemplo, `tabela.put(null, "ValorNulo")` ou `tabela.put("ChaveNula", null)`. O que acontece? Capture o resultado (por exemplo, envolva em um bloco try-catch para pegar a exceção) e então explique por que o `Hashtable` não permite chaves ou valores nulos (dica: questões históricas de implementação e necessidade de diferenciar null de ausência de chave, além de evitar problemas em métodos sincronizados).

3. **Consulta segura em múltiplas threads:** Considere um cenário em que múltiplas threads precisam acessar e modificar uma mesma estrutura de dados de mapa (por exemplo, contagem de acessos a páginas de um site em tempo real). Por que utilizar um `Hashtable` poderia ser uma escolha adequada nesse caso? Elabore uma questão teórica: descreva o que poderia dar errado se usássemos um `HashMap` não sincronizado com acesso concorrente e como o `Hashtable` previne esses problemas. Em seguida, opcionalmente, escreva um pequeno trecho de código simulando duas threads (pode ser de forma simplificada, chamando métodos em sequência para representar threads) acessando um `Hashtable` e observe que, devido à sincronização interna, as atualizações e leituras ocorrem sem corromper os dados (embora a ordem de execução entre threads possa variar).

4. **Iterando sobre Hashtable:** Mesmo sendo uma classe mais antiga, `Hashtable` implementa a interface `Map`, então podemos utilizar técnicas modernas de iteração. Crie um `Hashtable<String, Integer>` para armazenar produtos e seus estoques (ex: "Caneta"->50, "Lápis"->100, "Caderno"->75). Mostre duas formas de iterar sobre esse `Hashtable`: (a) usando um loop `for-each` em `entrySet()` para obter chave e valor, e (b) utilizando um `Iterator` sobre `keySet()` ou `entrySet()`. Imprima todos os produtos e estoques durante a iteração. Comente sobre como, apesar de sincronizado, o `Hashtable` ainda pode ser iterado de forma semelhante ao `HashMap`, porém ressaltando que se outros threads modificarem o mapa durante a iteração, é necessário cuidado (a iteração de `Hashtable` não é *fail-fast*, podendo não refletir as alterações concorrentes imediatamente).

5. **Hashtable vs. HashMap (conceitual):** Elabore uma questão conceitual pedindo para listar **diferenças e semelhanças** entre `Hashtable` e `HashMap`. Por exemplo: *"Quais são as principais diferenças entre as classes Hashtable e HashMap em Java, em termos de funcionalidades e uso prático?"*. A resposta deve abordar pontos como: sincronização (Hashtable é sincronizado e seguro para threads, HashMap não é sincronizado por padrão), permitir nulos (Hashtable não permite nenhum, HashMap permite uma chave nula e múltiplos valores nulos), performance (HashMap tende a ser mais rápido em ambientes single-thread por não ter overhead de sincronização), ordenação (nenhum garante ordem, ambos são baseados em hash), e status de

obsolescência (Hashtable é legado, hoje geralmente substituído por HashMap combinado com sincronização externa ou ConcurrentHashMap). Essa questão resume o entendimento das características específicas de cada classe.

Seção 5: HashMap (10 questões)

1. **Cadastro de usuários (login):** Implemente um sistema simples de cadastro e login de usuários usando `HashMap<String, String>` onde a chave é o nome de usuário (login) e o valor é a senha. O programa deve permitir: cadastrar novos usuários (inserindo no HashMap com `put`), não permitindo usuários duplicados (verifique com `containsKey` antes de adicionar), e realizar login verificando se um par login-senha fornecido confere com os dados armazenados (use `get` para obter a senha a partir do login e compare). Crie alguns usuários de teste e simule tentativas de login válidas e inválidas, exibindo mensagens adequadas em cada caso.

2. **Inventário de produtos:** Utilize um `HashMap<String, Integer>` para representar o estoque de uma loja, onde a chave é o nome (ou código) do produto e o valor é a quantidade em estoque. Implemente operações: (a) adicionar um novo produto com determinada quantidade (usando `put`), (b) registrar uma venda de um produto (reduzir a quantidade, isto é, fazer um `get` para pegar a quantidade atual e depois um `put` com a quantidade atual menos a vendida), garantindo que não fique negativa, (c) repor estoque de um produto (aumentar a quantidade de forma semelhante), e (d) remover um produto do mapa se sua quantidade chegar a zero (`remove`). Após algumas operações de exemplo, itere sobre o `HashMap` (por exemplo, com `for (Map.Entry<String,Integer> entry : estoque.entrySet())`) e imprima cada produto e sua quantidade restante. Note que a ordem de saída dos produtos não é garantida (por ser um HashMap).

3. **Contagem de ocorrências (palavras):** Dada uma frase qualquer, conte a frequência de cada palavra usando um `HashMap<String, Integer>`. Por exemplo, para a frase `"java java collections framework java map"`, o programa deve criar um HashMap onde as chaves são palavras e os valores o número de ocorrências de cada palavra. Use métodos `containsKey` para verificar se a palavra já foi vista (inicializando com 1 se for nova ou incrementando o valor existente caso contrário). Ao final, exiba o resultado de todas as contagens. *(Desafio adicional:* reescreva a lógica usando `merge` ou `computeIfAbsent` do Java 8+ para tornar o código mais conciso).

4. **Iterando sobre um HashMap:** Crie um `HashMap<String, Double>` que mapeia nomes de produtos aos seus preços. Insira pelo menos 5 produtos. Em seguida, percorra esse mapa usando `entrySet()` em um loop **for-each** e imprima cada produto e seu preço no formato `"Produto X = 99.99"`. Depois, faça uma segunda iteração usando `keySet()` e, para cada chave, recupere o valor com `get`. Ambas as abordagens devem resultar na mesma saída. Discuta qual é mais eficiente (dica: `entrySet()` evita a necessidade de buscar o valor para cada chave separadamente). Lembre-se de mencionar que a ordem de impressão não corresponde à ordem de inserção (é arbitrária no HashMap).

5. **Procurando valores específicos:** Suponha um `HashMap<String, Integer>` que armazena alunos e suas notas finais (por exemplo, "Ana"->90, "Bruno"->70, "Carlos"->90, "Diana"->85). Escreva um código para encontrar todos os alunos que tiraram uma nota específica, por exemplo 90. Como o `HashMap` não fornece um método direto para buscar por valor, será necessário iterar sobre as entradas (usando `entrySet()` ou `values()` junto com `keySet()`) e comparar cada valor. Armazene ou imprima os nomes dos alunos que possuem a

nota consultada. No exemplo dado, o resultado para nota 90 deve listar "Ana" e "Carlos". Aborde também o uso de `containsValue` – por que ele não resolve totalmente o problema (dica: ele apenas indica se *existe* ao menos um valor igual, mas não quais chaves o possuem).

6. **Encontrando a maior valor (máxima nota):** Dado o mesmo HashMap de alunos e notas do exemplo anterior, escreva um algoritmo para identificar a **nota máxima** presente e então descobrir qual (ou quais) alunos obtiveram essa nota máxima. Isso envolverá duas passagens: primeiro, iterar pelas `values()` do mapa para encontrar o valor máximo; segundo, iterar novamente (ou usar `entrySet()`) para coletar todos os alunos cujas notas sejam iguais a esse máximo. Imprima a nota máxima e os nomes correspondentes. Esse exercício reforça a iteração tanto por valores quanto por entradas do HashMap.

7. **Uso de null em HashMap:** Faça um teste com `HashMap` envolvendo chaves ou valores nulos. Por exemplo: crie um `HashMap<String, String>` e insira `map.put(null, "semChave")` e `map.put("chaveNull", null)`. Verifique que o HashMap **aceita** uma chave `null` (apenas uma, pois se inserir outra nula vai sobrescrever a anterior) e valores `null` múltiplos. Após inserir, use `get(null)` para obter o valor associado à chave nula e imprima-o, e também confirme que `containsKey(null)` retorna verdadeiro. Explique por que o HashMap permite `null` (diferença de design comparado ao Hashtable) e mencione quaisquer cuidados ao usar null como chave (por exemplo, todas as chaves nulas são consideradas iguais, e internamente é tratado de forma especial).

8. **Invertendo um mapa (swap keys/values):** Dado um `HashMap<String, String>` que mapeia códigos de país para nome do país (ex: "BR"->"Brasil", "US"->"Estados Unidos", "CN"->"China"), escreva um código para inverter esse mapeamento, ou seja, criar um novo `HashMap<String, String>` onde os nomes dos países sejam as chaves e os códigos sejam os valores. Atenção: se o HashMap original tiver valores repetidos isso pode causar conflito de chaves duplicadas no invertido – para simplificar, assuma que todos os valores originais são únicos. Utilize um loop `for-each` em `entrySet()` para percorrer as entradas do mapa original e `put` no novo mapa invertido. Imprima o mapa invertido resultante para verificar se a inversão foi realizada corretamente.

9. **Mesclando dois HashMaps:** Suponha que você tenha dois `HashMap<String, Integer>` representando as vendas de dois meses diferentes, onde a chave é o produto e o valor é a quantidade vendida no mês. Por exemplo, mês 1 tem `{"Calçado": 20, "Camisa": 15}` e mês 2 tem `{"Calça": 10, "Camisa": 5}`. Escreva um código para combinar esses dois mapas em um único HashMap que represente as vendas totais nos dois meses. Isso envolve iterar sobre um dos mapas e adicionar todos os seus elementos no mapa de destino (por exemplo, usando `putAll`), depois iterar sobre o segundo mapa: para cada entrada, se o produto já existir no mapa combinado, some as quantidades (atualize o valor com `put`), caso contrário apenas insira o novo par. No exemplo dado, o resultado final deveria ser `{"Calçado": 20, "Camisa": 20, "Calça": 10}`. Mostre o código e comente sobre como resolver conflitos de chave ao mesclar.

10. **Filtrando um HashMap com Streams:** (Avançado) Dado um `HashMap<String, Double>` de produtos e seus preços, utilize a API de *Streams* do Java 8+ para criar um novo mapa filtrado. Por exemplo, filtre apenas os produtos com preço abaixo de determinado valor X. Demonstre o uso de `entrySet().stream()` combinado com operações de filtro e coleta para construir outro mapa. *Dica:* você pode usar `filter(entry -> entry.getValue() < X)` seguido de `Collectors.toMap(entry -> entry.getKey(), entry -> entry.getValue())` para

coletar o resultado em um novo `Map`. Mostre um exemplo de filtro (por exemplo, produtos abaixo de 100 reais) e imprima o mapa resultante. Explique brevemente as vantagens dessa abordagem funcional e como ela evita a escrita de loops explícitos.

Seção 6: LinkedHashMap (5 questões)

1. **Ordem de chegada (maratona):** Em uma corrida, queremos registrar a ordem de chegada dos corredores. Utilize `LinkedHashMap<String, Double>` para mapear o nome do atleta para seu tempo de conclusão, inserindo os atletas **na ordem em que eles terminam a prova**. Insira por exemplo: `"Ana"` -> 3.2 (horas), `"Bruno"` -> 3.5, `"Carlos"` -> 4.0, etc., seguindo a ordem de chegada. Depois, itere pelo `LinkedHashMap` e imprima a classificação final mostrando a ordem dos corredores. Confirme que a saída respeita exatamente a ordem de inserção (que representa a ordem de chegada da maratona). Em seguida, remova um dos corredores (por exemplo, se alguém for desqualificado) e mostre que os demais permanecem na ordem original de chegada.

2. **Comparando LinkedHashMap e HashMap na prática:** Crie tanto um `HashMap` quanto um `LinkedHashMap` e insira nos dois os mesmos pares de dados de exemplo, por exemplo: `1 -> "um"`, `2 -> "dois"`, `3 -> "três"`, inseridos exatamente nessa ordem. Itere separadamente sobre cada mapa imprimindo seus conteúdos. Documente o resultado: o `HashMap` provavelmente exibirá as entradas em uma ordem imprevisível (por exemplo, `2 -> "dois"`, depois `1 -> "um"`, depois `3 -> "três"`, ou outra ordem qualquer), enquanto o `LinkedHashMap` exibirá `1 -> "um"`, `2 -> "dois"`, `3 -> "três"`, preservando a ordem de inserção. Explique por que o LinkedHashMap mantém a ordem e ressalte que internamente ele faz isso através de uma lista duplamente ligada que encadeia as entradas na ordem em que foram inseridas.

3. **Acesso vs Inserção (accessOrder):** Por padrão, o `LinkedHashMap` mantém a ordem de inserção. No entanto, ele oferece um modo alternativo chamado *order of access* (ordem de acesso) quando configurado apropriamente. Elabore um exemplo: Crie um `LinkedHashMap<Integer, String>` com `accessOrder=true` (usando o construtor avançado) para representar, por exemplo, um cache de páginas web (onde a chave é o ID da página e o valor é o conteúdo). Insira as entradas 1->"Page1", 2->"Page2", 3->"Page3" nessa ordem. Então acesse (faça `get`) a chave 1 e depois insira uma nova entrada 4->"Page4". Itere sobre o mapa e verifique a ordem das chaves agora. Com `accessOrder=true`, a chave "1" (Page1), por ter sido acessada recentemente, deve aparecer **depois** das chaves que não foram acessadas (resultado esperado da ordem de chaves: 2, 3, 1, 4). Explique este resultado, destacando que com *access order* ativado, qualquer acesso de leitura/escrita a uma entrada existente move essa entrada para o fim da ordem.

4. **Implementando um cache LRU simples:** Usando `LinkedHashMap<K,V>`, podemos implementar um cache que descarta o item menos recentemente usado (LRU - Least Recently Used) quando um novo item é adicionado e a capacidade máxima é excedida. Crie uma classe ou use uma classe anônima estendendo `LinkedHashMap<Integer, String>` para implementar um cache com capacidade máxima de 3 elementos. Sobrescreva o método `removeEldestEntry(Map.Entry eldest)` para que ele retorne true quando `size() > 3`, indicando que o elemento mais antigo deve ser removido automaticamente. Teste inserindo quatro entradas no cache e observe que, após a quarta inserção, o primeiro elemento inserido foi removido. Mostre o conteúdo do mapa antes e depois dessa inserção extra para verificar que o tamanho se mantém em 3 e que o item mais velho saiu. Explique brevemente como o método

`removeEldestEntry` é chamado internamente após cada `put` e como isso facilita a implementação de caches.

5. **Atualização de valor sem alterar ordem:** Monte um `LinkedHashMap<String, Integer>` de pontuação de jogadores (jogador -> pontos) inserindo alguns jogadores em certa ordem. Agora, suponha que um jogador já existente atualize sua pontuação (por exemplo, usando `put` com a mesma chave mas novo valor). Verifique a ordem das chaves após essa operação. Escreva um código onde você insere, por exemplo, `"Alice"->10, "Bob"->8, "Carol"->15` nessa ordem em um LinkedHashMap. Em seguida, atualize a pontuação de "Bob" para 12 usando `put("Bob", 12)`. Itere e mostre que a ordem das chaves **não mudou** – continua Alice, Bob, Carol – apesar de termos modificado o valor de Bob. Explique que, no modo de ordem de inserção, atualizar o valor de uma chave existente **não** muda sua posição no ordenamento (pois não é uma nova inserção, apenas substituição de valor). Contrastando, se fosse um LinkedHashMap em modo accessOrder, a chamada ao `put` existente poderia movê-la para o fim (porque conta como acesso).

Respostas das Questões

Respostas Seção 1: HashSet

1. **Conjunto de alunos únicos:** A solução envolve usar um `HashSet` para armazenar nomes de alunos, garantindo que cada nome apareça no máximo uma vez. No código abaixo, adicionamos alguns alunos (incluindo duplicados) e demonstramos as operações pedidas:

```
import java.util.HashSet;

public class CadastroAlunos {
    public static void main(String[] args) {
        HashSet<String> alunos = new HashSet<>();

        // a) Adicionando alunos (alguns duplicados)
        alunos.add("João");
        alunos.add("Maria");
        alunos.add("José");
        alunos.add("Maria");  // duplicata - não será adicionada
        alunos.add("Ana");

        // b) Verificando se um aluno existe antes de adicionar
        String novoAluno = "José";
        if (alunos.contains(novoAluno)) {
            System.out.println(novoAluno + " já está cadastrado!");
        } else {
            alunos.add(novoAluno);
        }

        // c) Removendo um aluno específico
        alunos.remove("Ana");

        // d) Exibindo todos os alunos restantes
```

```
        System.out.println("Alunos cadastrados: " + alunos);
    }
}
```

Explicação: Primeiro, criamos um `HashSet<String>` chamado `alunos`. Adicionamos `"João"`, `"Maria"`, `"José"` e duplicamos `"Maria"` para ilustrar que o segundo `add("Maria")` não terá efeito (o tamanho do conjunto continuará o mesmo, pois `HashSet` ignora elementos duplicados). Em seguida, antes de adicionar `"José"` novamente, usamos `contains` para detectar que `"José"` já está presente e, portanto, pulamos a adição (imprimindo uma mensagem). Removemos `"Ana"` do conjunto (se o elemento existir, ele será removido; caso contrário, nada acontece). Por fim, exibimos os alunos no conjunto. A saída (a ordem pode variar) seria algo como:

```
José já está cadastrado!
Alunos cadastrados: [João, Maria, José]
```

Note que: - O `HashSet` garantiu que não há duplicatas (apesar de termos tentado adicionar "Maria" e "José" duas vezes). - A verificação explícita com `contains` não era estritamente necessária ao usar `HashSet` (já que duplicatas são automaticamente ignoradas), mas foi incluída para mostrar o uso do método. - A ordem dos nomes impressos em `alunos` **não é necessariamente** a ordem de inserção. Por exemplo, a saída mostrou `[João, Maria, José]` mas poderia ser outra ordem. O importante é que todos são únicos. Essa ausência de ordem definida é esperada em um HashSet.

1. **Eliminando duplicatas de uma lista:** Para remover duplicatas mantendo apenas a primeira ocorrência de cada elemento, podemos usar um `HashSet` para filtrar. Uma forma rápida é usar o construtor do `HashSet` passando a lista, ou iterar manualmente. Abaixo, mostramos as duas abordagens:

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;

public class ListaSemDuplicatas {
    // Método 1: usando construtor HashSet e voltando para lista
    public static <T> List<T> removeDuplicatas(List<T> lista) {
        return new ArrayList<>(new HashSet<>(lista));
    }

    // Método 2: iterando manualmente para preservar ordem
    public static <T> List<T> removeDuplicatasPreservandoOrdem(List<T>
lista) {
        HashSet<T> visto = new HashSet<>();
        List<T> resultado = new LinkedList<>();
        for (T item : lista) {
            if (!visto.contains(item)) {       // se ainda não vimos o item
                visto.add(item);               // marca como visto
                resultado.add(item);           // adiciona ao resultado
            }
        }
```

```
        return resultado;
    }

    public static void main(String[] args) {
        List<String> emails = new ArrayList<>();
        emails.add("a@exemplo.com");
        emails.add("b@exemplo.com");
        emails.add("a@exemplo.com"); // duplicado
        emails.add("c@exemplo.com");
        emails.add("b@exemplo.com"); // duplicado

        List<String> filtrada1 = removeDuplicatas(emails);
        List<String> filtrada2 = removeDuplicatasPreservandoOrdem(emails);

        System.out.println("Original: " + emails);
        System.out.println("Sem duplicatas (ordem não garantida): " +
filtrada1);
        System.out.println("Sem duplicatas (ordem preservada): " +
filtrada2);
    }
}
```

Explicação: No exemplo acima, a lista original `emails` contém duplicatas. O método `removeDuplicatas` converte diretamente a lista para um `HashSet` (eliminando duplicatas) e volta para um `ArrayList`. Isso remove duplicatas mas **não preserva** a ordem original, pois o HashSet não mantém ordem; a lista resultante pode vir em qualquer ordem. O método `removeDuplicatasPreservandoOrdem` faz a filtragem manual: ele percorre a lista original e utiliza um HashSet auxiliar `visto` para registrar quais itens já foram encontrados. Apenas o primeiro encontro de cada item é adicionado ao resultado. Assim, a ordem de inserção original é respeitada.

Executando o programa com a lista de e-mails de teste, poderíamos ter uma saída semelhante a:

```
Original: [a@exemplo.com, b@exemplo.com, a@exemplo.com, c@exemplo.com,
b@exemplo.com]
Sem duplicatas (ordem não garantida): [a@exemplo.com, c@exemplo.com,
b@exemplo.com]
Sem duplicatas (ordem preservada): [a@exemplo.com, b@exemplo.com,
c@exemplo.com]
```

Note que a primeira abordagem retornou os e-mails únicos porém em uma ordem arbitrária (por exemplo, *a, c, b*), enquanto a segunda manteve a ordem de primeira aparição na lista original (*a, b, c*). Em ambos os casos, as duplicatas foram eliminadas. O uso de `HashSet` torna a filtragem eficiente (operações de inserção e verificação em HashSet são, em média, O(1)).

1. **Verificação de presença em coleção (sorteio):** Podemos usar um loop para gerar números aleatórios e preencher um HashSet até atingir 10 números únicos. Usaremos `Random` para gerar, `HashSet.contains()` para verificar duplicatas, e `HashSet.size()` para saber quando atingir 10 itens:

```java
import java.util.HashSet;
import java.util.Random;
import java.util.Set;

public class SorteioUnico {
    public static void main(String[] args) {
        Random rand = new Random();
        Set<Integer> sorteados = new HashSet<>();

        // Gera números aleatórios até ter 10 únicos
        while (sorteados.size() < 10) {
            int numero = rand.nextInt(100); // 0 a 99
            if (!sorteados.contains(numero)) {
                sorteados.add(numero);
                System.out.println("Sorteado: " + numero);
            } else {
                System.out.println("Número repetido " + numero + ",
ignorando...");
            }
        }

        // Resultado final
        System.out.println("Conjunto final de 10 números únicos: " +
sorteados);
    }
}
```

Explicação: Esse código ilustra o uso de `HashSet` para garantir unicidade no sorteio. Dentro do loop, enquanto ainda não temos 10 números distintos, geramos um número aleatório. Se `contains` indicar que o número já está em `sorteados`, nós o ignoramos (a mensagem "Número repetido ... ignorando..." é exibida). Se não contém, adicionamos ao conjunto e mostramos "Sorteado: ...". Após o loop, imprimimos o conjunto final.

A saída do programa será diferente a cada execução devido à aleatoriedade. Por exemplo, um resultado possível:

```
Sorteado: 17
Sorteado: 3
Sorteado: 85
Sorteado: 42
Sorteado: 19
Número repetido 42, ignorando...
Sorteado: 7
Sorteado: 99
Sorteado: 1
Número repetido 3, ignorando...
Sorteado: 50
Sorteado: 42  (este 42 aqui seria um novo sorteio que coincidiu, mas no
exemplo acima já tinha)
```

```
Sorteado: 64
Conjunto final de 10 números únicos: [1, 3, 99, 7, 42, 17, 50, 19, 85, 64]
```

(Observação: A ordem exibida no conjunto final é arbitrária, pode não ser a sequência de sorteio, pois HashSet não ordena.)

Como análise final: `HashSet` é ideal para esse caso, pois oferece inserção e busca rápidas. Mesmo que tentemos sortear 10 números podendo repetir, a checagem `contains` em um conjunto grande é muito rápida (complexidade média constante), então o processo de evitar duplicatas é eficiente.

1. **Interseção de conjuntos:** Para encontrar a interseção (comuns a ambos) de dois conjuntos, podemos usar `retainAll` ou iterar manualmente. Usando `retainAll` é mais simples, mas destrói um dos conjuntos; portanto, podemos trabalhar em cópia. Exemplo:

```java
import java.util.HashSet;
import java.util.Set;

public class IntersecaoConjuntos {
    public static void main(String[] args) {
        Set<String> cursoA = new HashSet<>();
        cursoA.add("Alice");
        cursoA.add("Bruno");
        cursoA.add("Carlos");

        Set<String> cursoB = new HashSet<>();
        cursoB.add("Bruno");
        cursoB.add("Daniel");
        cursoB.add("Alice");

        // Calculando interseção sem modificar originais
        Set<String> ambos = new HashSet<>(cursoA);  // copia cursoA
        ambos.retainAll(cursoB);                    // mantém apenas os
presentes em cursoB

        System.out.println("Alunos no Curso A: " + cursoA);
        System.out.println("Alunos no Curso B: " + cursoB);
        System.out.println("Presentes em ambos os cursos: " + ambos);
    }
}
```

Explicação: Criamos dois `HashSet<String>`, `cursoA` e `cursoB`, com alguns nomes. Bruno e Alice estão em ambos, enquanto Carlos é só do A e Daniel só do B. Para obter a interseção, criamos uma cópia de `cursoA` chamada `ambos`. Em seguida, `ambos.retainAll(cursoB)` faz com que `ambos` retenha apenas elementos que também estão em `cursoB`. O resultado, impresso em `ambos`, deverá conter apenas `"Alice"` e `"Bruno"`.

A saída esperada seria:

```
Alunos no Curso A: [Alice, Bruno, Carlos]
Alunos no Curso B: [Bruno, Daniel, Alice]
Presentes em ambos os cursos: [Alice, Bruno]
```

Lembrando que os conjuntos podem aparecer sem ordem específica nas impressões (por exemplo, `[Bruno, Alice]` ou `[Alice, Bruno]` ambos são válidos). O importante é que apenas os nomes comuns apareçam. Explicando as alternativas: poderíamos também iterar sobre o menor conjunto e checar cada elemento com `contains` no outro, adicionando em um terceiro conjunto de resultado. Mas `retainAll` simplifica tudo internamente. Essa operação de interseção é bastante eficiente também (percorrer um conjunto e checar presença no outro).

1. **Ordem não garantida (HashSet):** Ao inserir elementos em um HashSet e iterar, observa-se que a ordem de saída parece imprevisível. Vamos demonstrar isso e explicar:

```java
import java.util.HashSet;
import java.util.Set;

public class OrdemHashSetDemo {
    public static void main(String[] args) {
        Set<String> frutas = new HashSet<>();
        frutas.add("Maçã");
        frutas.add("Banana");
        frutas.add("Laranja");
        frutas.add("Kiwi");
        frutas.add("Abacaxi");

        System.out.println("Frutas no conjunto: ");
        for (String f : frutas) {
            System.out.println(f);
        }
    }
}
```

Explicação: Se executarmos o código acima, a saída **não seguirá a ordem de inserção** (Maçã, Banana, ...). Por exemplo, poderíamos obter:

```
Frutas no conjunto:
Banana
Kiwi
Maçã
Abacaxi
Laranja
```

No caso acima, inserimos as frutas numa ordem específica, mas o HashSet as imprimiu em outra ordem. Isso acontece porque o HashSet é implementado usando uma tabela hash, que distribui os elementos de acordo com seus *hash codes*. Ele está otimizado para eficiência de operações (inserir, remover, buscar em O(1) em média), **não para ordem**. A iteração percorre a tabela hash interna,

portanto a ordem resulta da disposição interna dos buckets e pode parecer aleatória para nós. Além disso, se inseríssemos ou removêssemos outros elementos, a ordem de iteração poderia mudar.

Sobre performance: as operações fundamentais do HashSet (`add`, `remove`, `contains`) têm complexidade média de tempo *constante* O(1) – ou seja, independem linearmente do tamanho do conjunto para grandes N, tornando HashSet muito eficiente para conjuntos grandes onde a ordem não importa. Em contrapartida, se precisarmos de uma coleção que mantenha a ordem de inserção, HashSet não é adequado – para isso usamos `LinkedHashSet` (ordem de inserção) ou `TreeSet` (ordem ordenada por valor, se for o caso de comparação), pagando um pequeno custo de desempenho.

Em resumo, esta questão reforça: **HashSet não garante ordem alguma**. Se virmos elementos ordenados coincidentemente, é acidente de implementação, não deve ser confiado. Para resultados ordenados ou previsíveis, deve-se usar outras estruturas.

Respostas Seção 2: LinkedHashSet

1. **Filtrando duplicatas com ordem preservada:** O uso de `LinkedHashSet` torna simples eliminar duplicatas enquanto preservamos a ordem original de inserção. Por exemplo, dada uma lista de produtos com repetidos, podemos fazer:

```
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Arrays;
import java.util.Set;
import java.util.ArrayList;

public class DuplicatasComOrdem {
    public static void main(String[] args) {
        List<String> produtos = Arrays.asList("A", "B", "C", "A", "D", "B");

        // Usando LinkedHashSet para filtrar mantendo ordem
        Set<String> conjuntoOrdenado = new LinkedHashSet<>(produtos);
        List<String> produtosUnicos = new ArrayList<>(conjuntoOrdenado);

        System.out.println("Original: " + produtos);
        System.out.println("Sem duplicatas (ordem preservada): " +
produtosUnicos);
    }
}
```

Explicação: Inicialmente, a lista `produtos` contém `["A", "B", "C", "A", "D", "B"]`. Ao criar um `LinkedHashSet` a partir dessa lista, internamente ele adiciona cada elemento na ordem que aparecem, ignorando as inserções duplicadas. O primeiro `"A"` entra, o segundo `"A"` é ignorado, e assim por diante. O `LinkedHashSet` resultante contém `["A", "B", "C", "D"]` na ordem de primeira ocorrência. Convertendo de volta para uma lista (`produtosUnicos`), preservamos essa ordem.

A saída impressa será:

```
Original: [A, B, C, A, D, B]
Sem duplicatas (ordem preservada): [A, B, C, D]
```

Isso mostra claramente que: - O segundo "A" e o segundo "B" foram eliminados como duplicatas. - A ordem `A, B, C, D` corresponde à ordem em que esses elementos únicos apareceram primeiro na lista original. - Se tivéssemos usado um `HashSet` simples, a lista resultante poderia vir em ordem imprevisível; mas o `LinkedHashSet` garantiu a ordem original.

1. **Lista de espera com ordem de inscrição:** Abaixo está um exemplo de gerenciamento de lista de espera utilizando `LinkedHashSet`. Incluímos operações de adicionar, remover e listar:

```java
import java.util.LinkedHashSet;
import java.util.Set;

public class ListaEsperaClinica {
    public static void main(String[] args) {
        Set<String> listaEspera = new LinkedHashSet<>();

        // a) Adicionando pacientes
        listaEspera.add("João");
        listaEspera.add("Maria");
        listaEspera.add("Carlos");
        listaEspera.add("Maria");  // duplicado, não entra

        // b) Removendo um paciente (por exemplo, João foi atendido)
        listaEspera.remove("João");

        // c) Listando em ordem de chegada
        System.out.println("Pacientes na fila de espera:");
        for (String paciente : listaEspera) {
            System.out.println(paciente);
        }
    }
}
```

Explicação: Começamos com um `LinkedHashSet<String>` vazio. Quando adicionamos "João", "Maria", "Carlos", a estrutura armazena nessa ordem. A tentativa de adicionar "Maria" de novo é ignorada (nenhum efeito, pois já existe). Depois removemos "João" (que estava na frente). Ao iterar e imprimir, a saída será:

```
Pacientes na fila de espera:
Maria
Carlos
```

Observações: - A duplicata "Maria" foi ignorada automaticamente, mantendo unicidade. - Após remover "João", a ordem relativa de "Maria" e "Carlos" permaneceu como era (ou seja, "Maria" continua vindo antes de "Carlos", exatamente como inserido originalmente). - `LinkedHashSet` facilita a iteração em ordem de inserção: no loop for-each, os pacientes saíram na ordem exata de chegada (tirando os

removidos). - Essa estrutura é útil para uma fila de espera porque alguém que chega primeiro permanece à frente até ser removido, e não haverá repetições do mesmo nome ocupando dois lugares.

1. **Comparando HashSet vs LinkedHashSet:** Vamos mostrar concretamente a diferença de ordem:

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

public class CompararHashSetLinkedHashSet {
    public static void main(String[] args) {
        Set<String> conjunto1 = new HashSet<>();
        Set<String> conjunto2 = new LinkedHashSet<>();

        String[] dados = {"1", "2", "3", "4", "5"};
        for (String s : dados) {
            conjunto1.add(s);
            conjunto2.add(s);
        }

        System.out.println("HashSet iteração:");
        for (String s : conjunto1) {
            System.out.print(s + " ");
        }
        System.out.println();

        System.out.println("LinkedHashSet iteração:");
        for (String s : conjunto2) {
            System.out.print(s + " ");
        }
        System.out.println();
    }
}
```

Explicação: Insere-se a sequência "1","2","3","4","5" em ambos os conjuntos. A saída típica pode ser:

```
HashSet iteração: 4 2 5 1 3
LinkedHashSet iteração: 1 2 3 4 5
```

No caso do HashSet, a ordem apareceu como `4 2 5 1 3` (que é apenas um exemplo de ordem possivelmente aleatória; outra execução ou outra JVM poderia dar uma sequência diferente). Já o LinkedHashSet iterou exatamente em `1 2 3 4 5`, que foi a ordem de inserção.

A razão: `HashSet` não mantém nenhuma informação sobre ordem de entrada, já o `LinkedHashSet` mantém uma **lista ligada interna** encadeando os elementos na ordem em que foram adicionados. Então, sempre que iteramos o LinkedHashSet, ele segue essa lista ligada. Assim, para qualquer dado inserido em uma ordem conhecida, LinkedHashSet garante reproduzir essa mesma ordem nas iterações futuras (a não ser que se removam elementos, mas mesmo assim, os remanescentes mantêm a ordem relativa original).

Esse experimento reforça que, se precisarmos tanto de unicidade quanto de ordem de inserção previsível, `LinkedHashSet` é a classe apropriada.

1. **Remoção e re-inserção:** Vamos simular a situação descrita: adicionando alguns participantes, removendo um e adicionando novamente:

```
import java.util.LinkedHashSet;
import java.util.Set;

public class ReinsereLinkedHashSet {
    public static void main(String[] args) {
        Set<String> jogadores = new LinkedHashSet<>();
        jogadores.add("Alice");
        jogadores.add("Bob");
        jogadores.add("Carol");
        System.out.println("Inicial: " + jogadores);

        // Remover o primeiro inserido ("Alice"), depois reinscrever/reativar
        jogadores.remove("Alice");
        System.out.println("Após remover Alice: " + jogadores);

        jogadores.add("Alice");
        System.out.println("Após reinserir Alice: " + jogadores);
    }
}
```

Explicação: Suponhamos que `"Alice"`, `"Bob"`, `"Carol"` entram no jogo nessa ordem. O LinkedHashSet `jogadores` inicialmente é `[Alice, Bob, Carol]` em ordem de inserção. Removemos "Alice". Agora o conjunto fica `[Bob, Carol]`. Ao adicionar "Alice" novamente, como ela estava ausente, será adicionada no fim da lista de ordem. O conjunto passa a ser `[Bob, Carol, Alice]`.

As impressões seriam:

```
Inicial: [Alice, Bob, Carol]
Após remover Alice: [Bob, Carol]
Após reinserir Alice: [Bob, Carol, Alice]
```

Isso demonstra que um elemento removido e depois adicionado é considerado novo do ponto de vista de ordem de inserção – ele vai para o final. Em termos de funcionamento interno, quando removemos "Alice", a ligação que a conectava é quebrada; ao adicioná-la de novo, o LinkedHashSet a coloca no tail (final) da lista ligada interna.

Em contexto de um jogo: se Alice saiu e voltou, agora ela é a última na ordem dos participantes. Esse detalhe pode ou não ser desejável dependendo da lógica do programa, mas é como LinkedHashSet opera. Se quiséssemos mantê-la na mesma posição original, seria mais complexo (teríamos que armazenar posição manualmente, etc.). Mas normalmente, LinkedHashSet se usa porque queremos

aquela ordem factual de inserção, então re-inserir logicamente significa "inserir de novo agora", portanto depois dos existentes.

1. **União preservando ordem:** Podemos unir dois LinkedHashSets usando `addAll`. Exemplo:

```
import java.util.LinkedHashSet;
import java.util.Set;

public class UniaoLinkedHashSet {
    public static void main(String[] args) {
        Set<String> filmesAna = new LinkedHashSet<>();
        filmesAna.add("Star Wars");
        filmesAna.add("Matrix");
        filmesAna.add("Titanic");

        Set<String> filmesBruno = new LinkedHashSet<>();
        filmesBruno.add("Matrix");
        filmesBruno.add("Avatar");
        filmesBruno.add("Inception");

        // União dos filmes vistos (qualquer um dos dois)
        Set<String> filmesTodos = new LinkedHashSet<>(filmesAna);
        filmesTodos.addAll(filmesBruno);

        System.out.println("Filmes de Ana: " + filmesAna);
        System.out.println("Filmes de Bruno: " + filmesBruno);
        System.out.println("Filmes que pelo menos um deles viu: " +
filmesTodos);
    }
}
```

Explicação: Considere: - Ana viu: [Star Wars, Matrix, Titanic] (nessa ordem) - Bruno viu: [Matrix, Avatar, Inception] (nessa ordem)

Criamos `filmesTodos` inicialmente como uma cópia de `filmesAna` (assim ele começa com Star Wars, Matrix, Titanic nessa ordem). Em seguida, `filmesTodos.addAll(filmesBruno)` vai tentar adicionar "Matrix", "Avatar", "Inception" nessa ordem. "Matrix" já existe (foi visto por Ana, estava no conjunto), então não é adicionado de novo. "Avatar" não existe ainda, então é adicionado – ele vai para o fim, após Titanic. Depois "Inception" também não existe e é adicionado no fim. O resultado final de `filmesTodos` deve ser: [Star Wars, Matrix, Titanic, Avatar, Inception].

A saída impressa seria:

```
Filmes de Ana: [Star Wars, Matrix, Titanic]
Filmes de Bruno: [Matrix, Avatar, Inception]
Filmes que pelo menos um deles viu: [Star Wars, Matrix, Titanic, Avatar,
Inception]
```

Observações: - O conjunto união `filmesTodos` não tem duplicatas (apesar de "Matrix" estar nos dois originais, aparece só uma vez). - A ordem resultante foi: todos os de Ana (na ordem de Ana), seguidos pelos de Bruno que não estavam em Ana (na ordem de Bruno). - Isso é uma propriedade interessante do `LinkedHashSet.addAll`: ele itera sobre a coleção fornecida (filmesBruno) e adiciona cada elemento. Novos elementos são anexados no final da ordem existente. Elementos já existentes são ignorados, sem afetar a ordem. - Em comparação, se usássemos HashSet, obteríamos os mesmos elementos na união, mas a ordem de iteração poderia ser diferente (misturada, imprevisível). Com LinkedHashSet, a união respeitou um senso de ordem: primeiro os de Ana, depois complementados pelos novos de Bruno.

Respostas Seção 3: TreeMap

1. **Agenda telefônica ordenada:** Abaixo implementamos a agenda usando `TreeMap`. Note como a impressão sai ordenada pelas chaves (nomes):

```
import java.util.Map;
import java.util.TreeMap;

public class AgendaTelefonica {
    public static void main(String[] args) {
        TreeMap<String, String> agenda = new TreeMap<>();

        // Inserindo contatos (fora de ordem alfabética deliberadamente)
        agenda.put("Carlos", "9999-0000");
        agenda.put("Ana",    "8888-1111");
        agenda.put("Pedro",  "7777-2222");
        agenda.put("Bruna",  "6666-3333");
        agenda.put("Fabio",  "5555-4444");

        // Busca de um telefone específico
        String consulta = "Bruna";
        String telBruna = agenda.get(consulta);
        System.out.println("Telefone de " + consulta + ": " + telBruna);

        // Remoção de um contato
        agenda.remove("Fabio");

        // Iteração e impressão de todos (ordenados por nome)
        System.out.println("---- Lista telefônica ----");
        for (Map.Entry<String, String> entry : agenda.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Explicação: Inserimos nomes em ordem aleatória: Carlos, Ana, Pedro, Bruna, Fabio. O `TreeMap` ordena por chave automaticamente, portanto internamente a ordem alfabética seria: Ana, Bruna, Carlos, Fabio, Pedro. Quando iteramos com `entrySet()`, esperamos ver nessa ordem. Antes da listagem, porém, fazemos duas operações: - Consulta: `agenda.get("Bruna")` retorna "6666-3333".

Imprimimos "Telefone de Bruna: 6666-3333". - Remoção: `agenda.remove("Fabio")` retira Fabio da agenda.

Ao iterar depois, Fabio não estará mais, e a lista deverá vir ordenada como: Ana, Bruna, Carlos, Pedro (Fabio removido). A saída será:

```
Telefone de Bruna: 6666-3333
---- Lista telefônica ----
Ana -> 8888-1111
Bruna -> 6666-3333
Carlos -> 9999-0000
Pedro -> 7777-2222
```

Perceba: - Mesmo inserindo fora de ordem, o TreeMap guardou ordenado. `Ana` veio primeiro sem termos inserido ela primeiro. - TreeMap não permite `null` como chave. Não tentamos isso aqui, mas é importante saber. Valores poderiam ser nulos, mas não usamos nulos no exemplo. - O método `entrySet()` nos dá pares chave-valor diretamente para iterar. Poderíamos ter usado `for (String nome : agenda.keySet()) { ... agenda.get(nome) ... }` também, mas isso seria menos eficiente por precisar buscar o valor a cada vez (embora TreeMap.get seja O(log n)). `entrySet` evita essa busca extra. - Complexidade: operações `put`, `get`, `remove` no TreeMap são O(log n) devido à estrutura de árvore balanceada (red-black tree). Isso é um pouco mais lento que HashMap na teoria, mas vem com o benefício da ordenação automática das chaves.

1. **Identificadores numéricos ordenados:** Implementação:

```java
import java.util.TreeMap;

public class RegistroFuncionarios {
    public static void main(String[] args) {
        TreeMap<Integer, String> funcionarios = new TreeMap<>();

        // Inserindo IDs e nomes fora de ordem
        funcionarios.put(104, "Maria");
        funcionarios.put(102, "João");
        funcionarios.put(120, "Ana");
        funcionarios.put(110, "Pedro");
        funcionarios.put(101, "Bruna");

        System.out.println("Menor ID registrado: " +
funcionarios.firstKey());
        System.out.println("Maior ID registrado: " + funcionarios.lastKey());

        System.out.println("Funcionários (ordenados por ID):");
        funcionarios.forEach((id, nome) -> {
            System.out.println("ID " + id + " = " + nome);
        });
    }
}
```

Explicação: Inserimos IDs: 104, 102, 120, 110, 101 (nessa ordem de chamada). O TreeMap vai armazená-los ordenados numericamente. Então internamente a ordem das chaves será 101, 102, 104, 110, 120. - `firstKey()` dará 101 (menor). - `lastKey()` dará 120 (maior). - Ao iterar (aqui usamos `forEach` com lambda, equivalente a entrySet iteration), a saída listará em ordem ascendente de IDs.

Saída esperada:

```
Menor ID registrado: 101
Maior ID registrado: 120
Funcionários (ordenados por ID):
ID 101 = Bruna
ID 102 = João
ID 104 = Maria
ID 110 = Pedro
ID 120 = Ana
```

Observações: - Repare que, na inserção, "Bruna" (ID 101) foi adicionada por último, mas na saída ela aparece primeiro por ter o menor ID. TreeMap ordenou pelos IDs automaticamente. - Métodos úteis como `firstKey`, `lastKey` tiram proveito dessa ordenação para obter extremos de forma direta e eficiente (também O(log n) ou constante, não precisando iterar todo o conjunto). - Similarmente, se quiséssemos podemos usar `treeMap.firstEntry()` para obter o par inteiro (ID e nome) do menor, etc., mas aqui focamos nas keys. - Se precisássemos, poderíamos também usar `higherKey`, `lowerKey` (para vizinhos de um dado valor) por serem numeric keys, mas não solicitado diretamente.

1. **Contagem de palavras ordenada:** Vamos implementar o contador de palavras:

```
import java.util.Map;
import java.util.TreeMap;

public class ContadorPalavras {
    public static void main(String[] args) {
        String frase = "Bolo de chocolate e bolo de cenoura";
        String[] palavras = frase.toLowerCase().replaceAll("[^a-zà-úÀ-
Ú0-9 ]", "").split("\\s+");

        Map<String, Integer> frequencia = new TreeMap<>();
        for (String palavra : palavras) {
            if (palavra.isEmpty()) continue;
            if (!frequencia.containsKey(palavra)) {
                frequencia.put(palavra, 1);
            } else {
                int cont = frequencia.get(palavra);
                frequencia.put(palavra, cont + 1);
            }
        }

        System.out.println("Frequência de palavras:");
        for (Map.Entry<String, Integer> entry : frequencia.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
```

```
        }
    }
}
```

Explicação: - Primeiro normalizamos a frase: `toLowerCase()` para ignorar maiúsculas/minúsculas e `replaceAll` para remover caracteres que não sejam letras, números ou espaço (assim eliminamos pontuação). Depois `split("\\s+")` divide por espaços (1 ou mais). - Usamos um `TreeMap<String, Integer>` chamado `frequencia`. Iteramos por cada palavra da frase: - Se a palavra não está no mapa, insere com valor 1. - Se já está, atualiza o valor incrementando em 1. - No final iteramos pelo TreeMap e imprimimos.

Dado o exemplo "Bolo de chocolate e bolo de cenoura", após o processamento a lista de palavras seria `["bolo","de","chocolate","e","bolo","de","cenoura"]`. Contagem resultante no TreeMap: - "bolo" -> 2 - "cenoura" -> 1 - "chocolate" -> 1 - "de" -> 2 - "e" -> 1

A saída impressa estará em ordem alfabética de palavra (chave):

```
Frequência de palavras:
bolo = 2
cenoura = 1
chocolate = 1
de = 2
e = 1
```

Podemos confirmar a ordenação: alfabeticamente, "bolo" vem antes de "cenoura", que vem antes de "chocolate", etc. Mesmo que "bolo" não fosse a primeira palavra inserida (na ordem de processamento apareceria na lista como primeira, mas imagine outro exemplo), TreeMap ordena no final.

(Bônus: poderíamos otimizar usando `frequencia.merge(palavra, 1, Integer::sum)` para somar, mas a solução acima é explícita e fácil de entender.)

1. **Ordenação personalizada (Comparator):** Vamos usar um comparador de ordem reversa:

```
import java.util.Comparator;
import java.util.Map;
import java.util.TreeMap;

public class TreeMapOrdemReversa {
    public static void main(String[] args) {
        // Comparator que inverte a ordem natural das strings
        Comparator<String> ordemReversa = Comparator.reverseOrder();
        TreeMap<String, String> paises = new TreeMap<>(ordemReversa);

        paises.put("BR", "Brasil");
        paises.put("US", "Estados Unidos");
        paises.put("AR", "Argentina");
        paises.put("FR", "França");

        for (Map.Entry<String, String> entry : paises.entrySet()) {
```

```
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Explicação: Criamos um `Comparator<String>` usando `Comparator.reverseOrder()`, que compara strings em ordem decrescente (Z-A). Passamos esse comparador ao construtor do TreeMap, de modo que o TreeMap passe a usar essa ordem ao invés da natural. Inserimos alguns países com código de dois caracteres: - Naturalmente a ordem dos códigos seria AR, BR, FR, US (alfabética). Mas com ordem reversa esperamos: US, FR, BR, AR.

Iterando sobre `paises.entrySet()` e imprimindo, obtemos:

```
US -> Estados Unidos
FR -> França
BR -> Brasil
AR -> Argentina
```

Isso confirma que a ordenação foi invertida. Internamente, o TreeMap usa o comparador fornecido para todas as comparações de chave: - Quando comparamos "US" com "BR", por exemplo, o comparador invertido dirá que "US" é menor (deve vir antes) porque, em ordem decrescente, U < B não, mas decrescente inverte, enfim, conceptualmente "US" > "BR" lexicograficamente, então em decrescente "US" vai primeiro. - O importante é: ao fornecer um Comparator, podemos personalizar ordenações. Poderíamos ordenar strings ignorando maiúsculas/minúsculas, ou ordenar objetos de forma específica. Por exemplo, se fossem objetos País com nome e código, poderíamos ordenar por nome usando comparador custom.

Em resumo, o TreeMap é flexível: sem Comparator usa *ordem natural* (definida por `Comparable` das chaves), com Comparator segue a regra dada. Aqui usamos a estática reverseOrder() para inverter a ordem natural das Strings.

1. **Subconjunto de dados (subMap):** Exemplo de uso de `subMap`:

```
import java.util.Map;
import java.util.TreeMap;
import java.util.SortedMap;

public class PedidosIntervalo {
    public static void main(String[] args) {
        TreeMap<Integer, String> pedidos = new TreeMap<>();
        // Populando com alguns pedidos (IDs e nomes de cliente)
        pedidos.put(1001, "Cliente A");
        pedidos.put(1500, "Cliente B");
        pedidos.put(2000, "Cliente C");
        pedidos.put(2500, "Cliente D");
        pedidos.put(3000, "Cliente E");
        pedidos.put(3500, "Cliente F");

        // Suponha que queremos pedidos de ID 1500 (inclusive) até 3000
```

```
(exclusivo)
        SortedMap<Integer, String> intervalo = pedidos.subMap(1500, 3000);

        System.out.println("Todos os pedidos: " + pedidos);
        System.out.println("Pedidos de 1500 (inclusive) a 3000 (exclusivo):
" + intervalo);

        // Podemos iterar sobre 'intervalo' como se fosse um map normal
        for (Map.Entry<Integer, String> entry : intervalo.entrySet()) {
            System.out.println("Pedido " + entry.getKey() + " = " +
entry.getValue());
        }
    }
}
```

Explicação: Montamos um TreeMap `pedidos` com alguns IDs não sequenciais, mas TreeMap os ordena (1001, 1500, 2000, 2500, 3000, 3500). O método `subMap(chaveInicialInclusiva, chaveFinalExclusiva)` retorna uma visão do mapa contendo somente as entradas cujas chaves estão no intervalo [1500, 3000). Isso incluirá 1500, 2000, 2500, mas não inclui 3000 (pois o limite superior é exclusivo).

Imprimimos: - Todos os pedidos (o TreeMap completo, vai mostrar ordenado por chave). - O `intervalo` subMap resultante. - Depois iteramos especificamente no subMap.

A saída seria algo como:

```
Todos os pedidos: {1001=Cliente A, 1500=Cliente B, 2000=Cliente C,
2500=Cliente D, 3000=Cliente E, 3500=Cliente F}
Pedidos de 1500 (inclusive) a 3000 (exclusivo): {1500=Cliente B, 2000=Cliente
C, 2500=Cliente D}
Pedido 1500 = Cliente B
Pedido 2000 = Cliente C
Pedido 2500 = Cliente D
```

Notamos que pedidos com ID 1001, 3000, 3500 foram filtrados fora do subMap. O subMap aqui é do tipo `SortedMap` (uma interface retornada pelo método). Ele é * backed * pelo TreeMap original – significa que se modificarmos o subMap, reflete no original e vice-versa (desde que sejam alterações dentro do intervalo).

Esse recurso de TreeMap é poderoso: em vez de manualmente filtrar com condições, podemos obter visões dinamicamente. Existem métodos irmãos: - `headMap(chave)`: elementos menores que a chave dada. - `tailMap(chave)`: elementos maiores ou iguais à chave dada. - Em Java 8+, também existem versões navegáveis (porque TreeMap implementa `NavigableMap`): `subMap(chaveIni, inclusive, chaveFim, inclusive)` para controlar inclusividade de limites.

Para a lógica de pedidos, isso permite, por exemplo, facilmente pegar pedidos do dia se IDs tivessem uma faixa por dia, ou qualquer faixa numérica ou alfabética de chaves.

Respostas Seção 4: Hashtable

1. **Dicionário de capitais:** A implementação em Java com Hashtable é similar a HashMap, mas lembrando das restrições de Hashtable (sem null). Exemplo:

```
import java.util.Hashtable;
import java.util.Map;

public class Capitais {
    public static void main(String[] args) {
        Map<String, String> capitais = new Hashtable<>();

        // Adicionando País -> Capital
        capitais.put("Brasil", "Brasília");
        capitais.put("França", "Paris");
        capitais.put("Japão", "Tóquio");
        capitais.put("Austrália", "Camberra");
        capitais.put("Canadá", "Ottawa");

        // Obtendo a capital de um país
        String pais = "Brasil";
        String capital = capitais.get(pais);
        System.out.println("Capital do " + pais + ": " + capital);

        // Removendo um país
        capitais.remove("Japão");

        // Iterando sobre o Hashtable
        System.out.println("Lista de Países e Capitais:");
        for (Map.Entry<String, String> entrada : capitais.entrySet()) {
            System.out.println(entrada.getKey() + " -> " +
entrada.getValue());
        }
    }
}
```

Explicação: Criamos um `Hashtable<String, String>` e usamos `put` para inserir cinco países. Depois usamos `get("Brasil")` para recuperar "Brasília". Removemos o "Japão". Por fim, iteramos com `entrySet()` para imprimir todos.

Uma possível saída:

```
Capital do Brasil: Brasília
Lista de Países e Capitais:
Brasil -> Brasília
França -> Paris
Austrália -> Camberra
Canadá -> Ottawa
```

Observações: - "Japão" não aparece porque foi removido. - A ordem de listagem não segue a ordem de inserção (inserimos Brasil, França, Japão, Austrália, Canadá - a saída ficou Brasil, França, Austrália, Canadá, ou poderia ser outra ordem). O Hashtable, como o HashMap, não tem ordem garantida. No exemplo, coincidentemente "Brasil" apareceu primeiro porque hashing da string "Brasil" pode ter colocado-a no primeiro bucket, etc., mas não há garantia. - Esse exemplo funciona similar a HashMap do ponto de vista do desenvolvedor, porém internamente `Hashtable` é thread-safe (métodos sincronizados). Se vários threads chamarem métodos simultaneamente, o Hashtable gerencia internamente para evitar corrupção de dados (embora possa ter impacto de desempenho). - Também vale notar: se tentássemos `capitais.put(null, "AlgumaCapital")` ou `capitais.put("AlgumPais", null)`, ocorreria NullPointerException. (Ver próxima questão.)

1. **Proibido nulos:** Vamos demonstrar a tentativa de uso de null:

```
import java.util.Hashtable;

public class TestaNulosHashtable {
    public static void main(String[] args) {
        Hashtable<String, String> tabela = new Hashtable<>();
        try {
            tabela.put(null, "ValorNulo");
        } catch (NullPointerException e) {
            System.out.println("Não foi possível inserir chave nula: " + e);
        }
        try {
            tabela.put("Chave", null);
        } catch (NullPointerException e) {
            System.out.println("Não foi possível inserir valor nulo: " + e);
        }
        System.out.println("Tabela final: " + tabela);
    }
}
```

Explicação: O código acima tenta duas inserções inválidas. Em ambas, o `Hashtable.put` lança `NullPointerException`. As mensagens seriam:

```
Não foi possível inserir chave nula: java.lang.NullPointerException
Não foi possível inserir valor nulo: java.lang.NullPointerException
Tabela final: {}
```

Assim, confirmamos: - `Hashtable` não permite chave nula nem valor nulo. Diferente do HashMap (que permite uma chave nula e valores nulos). - O motivo histórico: Hashtable é muito antiga (Java 1.0). A especificação optou por não permitir null para evitar confusão entre `get(key)` retornar null significando "não existe mapeamento para essa chave" vs "o mapeamento existe e o valor é null". Em HashMap isso é resolvido com alguns métodos auxiliares (como `containsKey` para distinguir), mas Hashtable foi projetada antes e decidiu proibir null para simplificar. - Além disso, métodos sincronizados de Hashtable esperam objetos não-nulos (um null poderia causar comportamento inconsistente se não

tratado adequadamente). - No uso prático, raramente precisamos de chaves nulas – se isso for necessário, HashMap fornece, mas com Hashtable não.

1. **Consulta segura em múltiplas threads (conceitual):** Quando várias threads acessam um HashMap simultaneamente para leitura/escrita, podem ocorrer condições de corrida e até *corrupção estrutural* do mapa. Por exemplo, duas threads inserindo em um HashMap sem sincronização podem interferir uma na outra, resultando em um estado inconsistente (alguns dados podem se perder ou links internos do bucket list podem formar ciclos, levando a loop infinito em busca – problemas documentados em versões antigas de HashMap se usado incorretamente em concorrência).

O `Hashtable`, por sua vez, é *internamente sincronizado*. Isso significa que seus métodos `put`, `remove`, `get`, etc., são todos *synchronized*. Apenas uma thread por vez pode executar qualquer um desses métodos, garantindo que a estrutura não fique corrompida. Então, em um cenário de múltiplos threads: - Com HashMap: precisaríamos gerenciar a sincronização manualmente (por exemplo, usando um bloco synchronized ao redor das operações ou usar um `Collections.synchronizedMap` ou um `ConcurrentHashMap`). Se não fizermos nada, leituras e escritas simultâneas podem apresentar valores incorretos ou até travar. - Com Hashtable: podemos usá-lo diretamente e ter a segurança de que não haverá interferência destrutiva. Porém, a desvantagem é o desempenho – a sincronização serializa as operações, e todas as threads podem ficar esperando, tornando o acesso concorrente possivelmente lento se muito contended.

Exemplo ilustrativo (simplificado sem threads reais):

```
import java.util.Hashtable;

public class SimulaConcorrencia {
    public static void main(String[] args) {
        Hashtable<String, Integer> acessos = new Hashtable<>();
        // Thread 1 (simulada)
        Runnable tarefa1 = () -> {
            for(int i=0; i<1000; i++) {
                acessos.put("pagina", acessos.getOrDefault("pagina", 0) + 1);
            }
            System.out.println("Thread1 finalizou");
        };
        // Thread 2 (simulada)
        Runnable tarefa2 = () -> {
            for(int i=0; i<1000; i++) {
                acessos.put("pagina", acessos.getOrDefault("pagina", 0) + 1);
            }
            System.out.println("Thread2 finalizou");
        };
        // Executando "concorrentemente" (aqui de forma sequencial apenas
para exemplo)
        tarefa1.run();
        tarefa2.run();

        System.out.println("Acessos à pagina: " + acessos.get("pagina"));
```

```
    }
}
```

No código acima, duas "threads" incrementam contagens na tabela hash `acessos`. Se isso fosse feito com HashMap sem sincronização, poderíamos acabar com um resultado menor que 2000 (incrementos perdidos) ou até outros problemas. Com Hashtable, mesmo sem lock explícito nosso, internamente cada `put` e `get` são sincronizados, então o valor final para chave "pagina" será **exatamente 2000**. As threads podem ter intercalado as execuções, mas a integridade da contagem está garantida.

(Obs.: O código dado executa as tarefas sequencialmente chamando run() diretamente; em um cenário real, usaríamos `new Thread(tarefa1).start()` etc., mas queríamos evitar dependências de timing. De qualquer forma, o conceito demonstrado é a segurança do resultado.)

Conclusão conceitual: Hashtable evita condições de corrida básicas por meio de sincronização intrínseca. Entretanto, ainda é possível que leituras iterativas precisem de precaução: por exemplo, se uma thread itera com `for-each` e outra modifica, a enumeração de Hashtable não lança exceção mas pode não ver as atualizações ou ver um estado "fotografado" do início (no caso do iterator fail-fast do HashMap ele lançaria exceção). No Hashtable, os iteradores (ou enumerations) são *não-fail-fast*, então podem continuar mas com dados possivelmente defasados. Portanto, para consistência total durante iteração, deve-se bloquear externamente também.

1. **Iterando sobre Hashtable:** Vamos demonstrar as duas formas pedidas:

```
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;

public class IterarHashtable {
    public static void main(String[] args) {
        Hashtable<String, Integer> estoque = new Hashtable<>();
        estoque.put("Caneta", 50);
        estoque.put("Lápis", 100);
        estoque.put("Caderno", 75);

        // a) Iteração com for-each e entrySet()
        System.out.println("Iteração com entrySet:");
        for (Map.Entry<String, Integer> entry : estoque.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }

        // b) Iteração com Iterator sobre keySet()
        System.out.println("Iteração com Iterator e keySet:");
        Iterator<String> it = estoque.keySet().iterator();
        while (it.hasNext()) {
            String produto = it.next();
            Integer qtd = estoque.get(produto);
            System.out.println(produto + " = " + qtd);
        }
```

```
    }
}
```

Explicação: Preenchemos o Hashtable `estoque` com 3 produtos. Então: - (a) Usamos um loop for-each em `entrySet()`. Isso nos dá diretamente a chave e valor para cada entrada. A saída pode ser em qualquer ordem, por exemplo:

```
Caneta = 50
Caderno = 75
Lápis = 100
```

(A ordem pode variar: possivelmente "Lápis" primeiro, etc, não previsível, mas todos aparecem.) - (b) Usamos um `Iterator` explícito sobre `keySet()`. Obtivemos cada chave e depois usamos `get` para pegar o valor correspondente. Isso é mais verboso e ligeiramente menos eficiente (pois `get` procura a chave novamente em tabela hash, embora com overhead pequeno). O resultado impresso será o mesmo conjunto de linhas, apenas talvez em uma ordem diferente se a estrutura mudou ou se iterar via keySet vs entrySet yields same underlying order. No Hashtable, `entrySet().iterator()` e `keySet().iterator()` likely iterate in the same order because they're linked to the same underlying structure. But anyway, output lines are identical content.

Sobre considerações de thread: - Como mencionado, `Hashtable` não oferece iterators fail-fast. No código monothread acima, tudo bem. Se outro thread modificasse `estoque` durante a iteração, o `Iterator` do Hashtable **não** lançaria `ConcurrentModificationException` (diferente de iterador de HashMap). Ele provavelmente continuaria a percorrer a tabela original. Dependendo da modificação, ele pode ou não ver o item novo/modificado. Isso significa que iterar um Hashtable durante modificações concorrentes pode não refletir precisamente o estado final e deve ser tratado cuidadosamente (por exemplo, travando o mapa externamente ou usando ConcurrentHashMap que oferece iterators "weakly consistent"). - Mas pelo menos, não quebra com exceção - essa é uma diferença de comportamento. Porém, isso não garante segurança lógica, apenas evita crash. Por isso, costuma-se preferir `ConcurrentHashMap` hoje em dia para cenários realmente paralelos, pois ele lida melhor com consistência durante iterações.

1. **Hashtable vs. HashMap (diferenças e semelhanças):** *(Esta resposta é descritiva e resume os pontos principais.)*

2. **Sincronização (Thread-safety):** Hashtable é sincronizado internamente – vários threads podem usá-lo simultaneamente sem corromper os dados, pois os métodos são bloqueados (synchronized). Já HashMap **não** é sincronizado; acesso concorrente não controlado a um HashMap pode causar problemas sérios (perda de dados ou loop infinito em certas condições). Se for necessário uso thread-safe, ou sincroniza-se externamente o HashMap (por exemplo via `Collections.synchronizedMap`) ou utiliza-se estruturas especializadas (`ConcurrentHashMap`). Devido a essa diferença, em aplicações single-threaded ou em regiões que não requerem sincronização, o HashMap é preferido por não ter o overhead de locks.

3. **Permissão de null:** HashMap permite uma chave null (no máximo uma, porque se inserir outra chave null substituirá a anterior) e permite múltiplos valores null. Exemplo: `hashMap.put(null, "valor")` funciona, e `hashMap.get(null)` retornará "valor". Já Hashtable não permite nem chave nem valor null – qualquer tentativa lança `NullPointerException`. Portanto, se seu caso de uso precisa armazenar alguma chave nula

(às vezes usado para indicar um valor "desconhecido" ou "padrão"), Hashtable não serve, enquanto HashMap lida com isso.

4. **Ordenação/Iteração:** Nenhum dos dois garante ordem de iteração das entradas; ambos são baseados em hash. Em ambas as classes, a ordem pode parecer aleatória e pode mudar se elementos forem adicionados/removidos. Se precisar de ordem previsível, deve-se usar LinkedHashMap (ordem de inserção) ou TreeMap (ordem por chave) em vez de HashMap/Hashtable. Nesse aspecto, HashMap e Hashtable se comportam de forma semelhante (inserir nas mesmas sequências tende a produzir ordens de iteração possivelmente diferentes entre elas, mas ambas sem garantia). Pequenas diferenças podem existir na ordem devido a capacidade inicial ou algoritmos de hash, mas nada confiável para o programador.

5. **Performance:** Em contexto de uma única thread, o HashMap geralmente oferece desempenho melhor que Hashtable, porque não sofre a penalidade de sincronização em cada operação. Hashtable sincroniza todos os métodos de acesso, o que faz com que operações sequenciais sejam ligeiramente mais lentas. Além disso, desde Java 1.8, HashMap trouxe melhorias como árvore binária nos buckets quando estes ficam muito cheios (melhorando worst-case para O(log n) em vez de O(n) em buscas raras), enquanto Hashtable não recebeu essas atualizações (pois é legacy). Portanto, para uso não concorrente, HashMap é quase sempre a escolha recomendada. Em cenários concorrentes intensos, geralmente se prefere `ConcurrentHashMap` ao invés de Hashtable, pois ele tem um mecanismo de sincronização mais granular (segmentado) e iteradores mais seguros, oferecendo melhor performance escalável.

6. **Pertence a Collections Framework:** HashMap faz parte do framework de coleções desde Java 1.2. Hashtable é uma classe herdada do Java 1.0 (antes do Collections Framework), embora tenha sido retroativamente adaptada para implementar a interface Map. Por ser legacy, muitas vezes documentação ou cursos sugerem evitar Hashtable a não ser por compatibilidade. Em novo código, usar HashMap (ou ConcurrentHashMap para thread safety) é preferível.

7. **Semelhanças:** Ambas armazenam pares chave-valor e implementam `Map<K,V>`. Ambas usam hashing para distribuir as chaves, permitindo operações eficientes de busca/inserção. Métodos básicos (put, get, remove, size, etc.) existem nos dois e funcionam de forma equivalente do ponto de vista da funcionalidade. A principal diferença está no comportamento multi-thread e restrição de null, conforme citado. Em termos de capacidade inicial, fator de carga e operações de redimensionamento, ambos têm conceitos semelhantes (Hashtable normalmente começa com 11 slots e fator de carga 0.75 por padrão, HashMap 16 slots e fator 0.75, mas isso são detalhes de implementação). Ambos usam `hashCode()` das chaves e depois, possivelmente, `equals` para localizar entradas – portanto chave imutável e bons hashCodes são importantes em ambos.

Resumindo, **quando usar cada**: se estiver em um ambiente single-thread ou gerenciar a sincronização manualmente, use `HashMap` por ser mais moderno e rápido. Se precisar de um Map thread-safe e não quiser gerenciar locks manualmente e não pode usar Java 5+ (ConcurrentHashMap) por algum motivo, `Hashtable` é uma opção. Porém, hoje em dia, é comum preferir `ConcurrentHashMap` para melhor desempenho em multithreading, tornando Hashtable raramente usado.

Respostas Seção 5: HashMap

1. **Cadastro de usuários (login):** Solução exemplificativa:

```
import java.util.HashMap;
import java.util.Map;

public class SistemaLogin {
    public static void main(String[] args) {
        Map<String, String> usuarios = new HashMap<>();

        // Cadastro de novos usuários
        cadastrarUsuario(usuarios, "alice", "1234");
        cadastrarUsuario(usuarios, "bob", "abcd");
        cadastrarUsuario(usuarios, "alice", "9999"); // tentativa de
duplicata

        // Tentativas de login
        efetuarLogin(usuarios, "alice", "1234"); // senha correta
        efetuarLogin(usuarios, "alice", "0000"); // senha incorreta
        efetuarLogin(usuarios, "carlos", "1234"); // usuário inexistente
    }

    static void cadastrarUsuario(Map<String, String> mapa, String login,
String senha) {
        if (mapa.containsKey(login)) {
            System.out.println("Erro: usuário \"" + login + "\" já existe!");
        } else {
            mapa.put(login, senha);
            System.out.println("Usuário \"" + login + "\" cadastrado com
sucesso.");
        }
    }

    static void efetuarLogin(Map<String, String> mapa, String login, String
senha) {
        if (!mapa.containsKey(login)) {
            System.out.println("Login falhou: usuário \"" + login + "\" não
existe.");
        } else {
            String senhaCorreta = mapa.get(login);
            if (senhaCorreta.equals(senha)) {
                System.out.println("Login bem-sucedido para usuário " +
login + "!");
            } else {
                System.out.println("Login falhou: senha incorreta para
usuário " + login + ".");
            }
        }
    }
}
```

Explicação: Criamos um HashMap `usuarios` onde a chave é o login (por exemplo "alice") e o valor é a senha.

- No cadastro (`cadastrarUsuario`): verifica-se `containsKey`. Se já existe, não cadastra e informa erro; se não existe, insere com `put` e confirma sucesso.
- No login (`efetuarLogin`): primeiro checamos se o usuário existe (`containsKey`). Se não, mensagem de usuário não encontrado. Se existe, pegamos a senha armazenada com `get` e comparamos com a fornecida:
- Se iguais, login ok.
- Se diferentes, login falhou por senha incorreta.

No `main`, cadastramos "alice" e "bob". Tentamos cadastrar "alice" de novo para mostrar a prevenção de duplicata. Depois fazemos três logins de teste: - alice com senha 1234 (correta, deve passar) - alice com senha 0000 (incorreta, deve falhar) - carlos com qualquer senha (usuário não existe, falha).

A saída esperada seria:

```
Usuário "alice" cadastrado com sucesso.
Usuário "bob" cadastrado com sucesso.
Erro: usuário "alice" já existe!
Login bem-sucedido para usuário alice!
Login falhou: senha incorreta para usuário alice.
Login falhou: usuário "carlos" não existe.
```

Isso demonstra o HashMap armazenando e recuperando credenciais: - Inserções via `put`. - Checagem de existência via `containsKey`. - Busca via `get`. - Não usamos `containsValue` aqui, porque verificar a existência de uma senha isoladamente não faz sentido sem referência ao usuário. - Em termos de segurança real, não se guardariam senhas em texto plano, mas para exercício de lógica está ok.

1. **Inventário de produtos:** Exemplo de gestão de estoque:

```
import java.util.HashMap;
import java.util.Map;

public class InventarioLoja {
    public static void main(String[] args) {
        Map<String, Integer> estoque = new HashMap<>();

        // a) Adicionar novos produtos
        adicionarProduto(estoque, "Camisa", 10);
        adicionarProduto(estoque, "Calça", 5);
        adicionarProduto(estoque, "Sapato", 20);

        // b) Registrar vendas
        venderProduto(estoque, "Camisa", 3); // vende 3 camisas
        venderProduto(estoque, "Calça",
6);  // tenta vender 6 calças, mas só tem 5
        venderProduto(estoque, "Boné", 2);   // produto que não existe
```

```
        // c) Repor estoque
        reporProduto(estoque, "Calça", 10);
        reporProduto(estoque, "Boné", 5);    // repor item que não existia
ainda

        // d) Remover produto com estoque zero
        removerSeZerado(estoque, "Camisa");
        removerSeZerado(estoque, "Sapato");

        // Exibir estoque final
        System.out.println("Estoque final:");
        for (Map.Entry<String, Integer> item : estoque.entrySet()) {
            System.out.println(item.getKey() + " = " + item.getValue());
        }
    }

    static void adicionarProduto(Map<String, Integer> estoque, String
produto, int quantidade) {
        estoque.put(produto, quantidade);
        System.out.println("Adicionado " + produto + " com quantidade " +
quantidade);
    }

    static void venderProduto(Map<String, Integer> estoque, String produto,
int quantidade) {
        Integer qtdAtual = estoque.get(produto);
        if (qtdAtual == null) {
            System.out.println("Produto " + produto + " não existe no
estoque!");
        } else if (qtdAtual < quantidade) {
            System.out.println("Não há estoque suficiente de " + produto + "
para vender " + quantidade
                               + " (disponível: " + qtdAtual + ")");
        } else {
            estoque.put(produto, qtdAtual - quantidade);
            System.out.println("Venda realizada: " + quantidade + "
unidade(s) de " + produto
                               + " (restantes: " + estoque.get(produto) +
")");
        }
    }

    static void reporProduto(Map<String, Integer> estoque, String produto,
int quantidade) {
        Integer qtdAtual = estoque.get(produto);
        if (qtdAtual == null) {
            estoque.put(produto, quantidade);
            System.out.println("Produto " + produto + " não existia.
Cadastrado com quantidade " + quantidade);
        } else {
```

```
            estoque.put(produto, qtdAtual + quantidade);
            System.out.println("Estoque de " + produto + " reposto em " +
quantidade
                               + " (total agora: " + estoque.get(produto) +
")");
        }
    }

    static void removerSeZerado(Map<String, Integer> estoque, String
produto) {
        Integer qtdAtual = estoque.get(produto);
        if (qtdAtual != null && qtdAtual == 0) {
            estoque.remove(produto);
            System.out.println("Produto " + produto + " removido do estoque
(quantidade zerada).");
        }
    }
}
```

Explicação: Funções: - `adicionarProduto`: insere um novo item com quantidade inicial (ou atualiza se já existia, mas no uso aqui assumimos novo produto). - `venderProduto`: verifica disponibilidade e subtrai quantidade se possível. Usa `get` para pegar o estoque atual. Se produto não existe (`get` retorna null) -> mensagem. Se existe mas quantidade insuficiente -> mensagem de erro. Se suficiente -> atualiza com `put(produto, novoValor)`. - `reporProduto`: se produto não existe, adiciona (`put`) novo item; se existe, soma a quantidade (`put` com `qtdAtual + quantidade`). - `removerSeZerado`: remove do mapa com `remove` se estoque atual for exatamente 0.

No `main`, testamos vários cenários, inclusive vender mais do que tem, vender produto inexistente, repor novo produto, remover produtos zerados.

Saída esperada (comentários ao lado):

```
Adicionado Camisa com quantidade 10
Adicionado Calça com quantidade 5
Adicionado Sapato com quantidade 20

Venda realizada: 3 unidade(s) de Camisa (restantes: 7)
Não há estoque suficiente de Calça para vender 6 (disponível: 5)
Produto Boné não existe no estoque!

Estoque de Calça reposto em 10 (total agora: 15)
Produto Boné não existia. Cadastrado com quantidade 5

Produto Camisa removido do estoque (quantidade zerada).
*(Nota: Camisa não foi zerada, então essa linha não apareceria na verdade.
Vamos analisar isso.)*
Produto Sapato removido do estoque (quantidade zerada).
*(Sapato ainda tinha 20, então essa também não deve aparecer. Vamos ver, acho
que removemos Camisa e Sapato indevidamente.)*
```

Opa, percebemos possível erro: `removerSeZerado` está sendo chamado para "Camisa" e "Sapato" independentemente, mas após operações: - Camisa restante 7, então não zera -> não remove (a função checa `qtdAtual == 0`). Então a função não remove nem imprime nada para Camisa. - Sapato ainda tem 20 (nunca alterado), também não zera -> nada acontece.

Portanto, as linhas "Produto Camisa removido..." e "Produto Sapato removido..." **não serão impressas** no fluxo real, porque a condição falha. Correto.

As únicas remoções que aconteceriam seriam se algo ficasse zero. Em nosso teste, nada ficou exatamente zero: - Calça ficou 15 depois de repor. - Boné ficou 5. - Camisa ficou 7. - Sapato 20.

Então `removerSeZerado` não remove nada. Talvez devíamos ter testado um caso com quantidade exata vendida ou vendendo todo estoque, mas tudo bem.

No final, estoque final será:

```
Estoque final:
Camisa = 7
Calça = 15
Sapato = 20
Boné = 5
```

(Order possibly different, e.g., {Calça=15, Boné=5, Camisa=7, Sapato=20} as HashMap is unordered).

Principais conceitos demonstrados: - Uso de `HashMap` para manter pares produto-quantidade. - Atualização de valores via `get` e `put`. - Remoção de entradas com `remove`. - Checagem de existência via `get` retornando null ou `containsKey`. - Iteração final usando `entrySet` to list inventory. - Notar ordem arbitrária na saída.

1. **Contagem de ocorrências (palavras):** (Já fizemos com TreeMap, mas pedem HashMap aqui. Vamos fazer com HashMap e destacar diferença se any.)

```java
import java.util.HashMap;
import java.util.Map;

public class ContarPalavrasComHashMap {
    public static void main(String[] args) {
        String frase = "java java collections framework java map";
        String[] palavras = frase.split("\\s+");

        Map<String, Integer> contagem = new HashMap<>();
        for (String p : palavras) {
            Integer qtd = contagem.get(p);
            if (qtd == null) {
                contagem.put(p, 1);
            } else {
                contagem.put(p, qtd + 1);
            }
        }
```

```
        // Exibindo resultados
        for (Map.Entry<String, Integer> entrada : contagem.entrySet()) {
            System.out.println(entrada.getKey() + " -> " +
entrada.getValue());
        }
    }
}
```

Explicação: Percorremos cada palavra na frase e usamos `get`: - Se retorna `null`, significa que a palavra não estava no mapa ainda, então adicionamos com valor 1. - Se retorna um Integer, incrementamos e atualizamos.

Para a frase `"java java collections framework java map"`, as contagens finais deveriam ser: - java: 3 (apareceu três vezes) - collections: 1 - framework: 1 - map: 1

O HashMap `contagem` pode armazenar isso. Ao iterar, a ordem não é garantida. Poderia por exemplo imprimir:

```
framework -> 1
java -> 3
map -> 1
collections -> 1
```

ou em outra ordem, não se sabe, mas todos valores corretos.

(Nota: no TreeMap versão, a saída foi alfabetica. Aqui é undefined ordering. Então se quisermos ordem alfabética, uma opção seria usar TreeMap diretamente ou classificar as keys manualmente; mas o problema não exige ordem, apenas contagem. O importante é que contagem está certa.)

(Desafio com métodos Java 8): Poderíamos usar `contagem.merge(p, 1, Integer::sum)` dentro do loop, ou `computeIfAbsent`:

```
contagem.computeIfAbsent(p, k -> 0);
contagem.put(p, contagem.get(p) + 1);
```

Mas isso é quase o mesmo que get/put simples. O `merge` é ideal: `contagem.merge(p, 1, Integer::sum);` que automaticamente soma 1 se existe ou insere 1 se não.

Isso reduziria o loop a:

```
for(String p : palavras) {
    contagem.merge(p, 1, Integer::sum);
}
```

Mas na resposta principal já está resolvido sem isso. Podemos mencionar essa alternativa se quisermos evidenciar o "desafio adicional" pedido:

> *(Desafio adicional: reescreva usando merge ou computeIfAbsent)*

Na explicação do TreeMap questão análoga, mencionamos merge. Podemos reiterar aqui: - `merge` e `computeIfAbsent` são conveniências adicionadas no Java 8 para evitar aquele padrão if-null-then-put.

1. **Iterando sobre um HashMap:** Exemplo:

```
import java.util.HashMap;
import java.util.Map;

public class MapDePrecos {
    public static void main(String[] args) {
        Map<String, Double> precos = new HashMap<>();
        precos.put("Mouse", 49.99);
        precos.put("Teclado", 129.90);
        precos.put("Monitor", 799.0);
        precos.put("Cadeira", 650.5);
        precos.put("Mochila", 120.0);

        // Iteração com entrySet
        System.out.println("Listando preços (entrySet):");
        for (Map.Entry<String, Double> entry : precos.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }

        // Iteração com keySet
        System.out.println("Listando preços (keySet + get):");
        for (String produto : precos.keySet()) {
            Double price = precos.get(produto);
            System.out.println(produto + " = " + price);
        }
    }
}
```

Explicação: Inserimos 5 produtos com preços. Depois fazemos duas listagens:

- Usando `entrySet()`: nos dá diretamente chave e valor em cada iteração, o que é conveniente.
- Usando `keySet()`: precisamos chamar `get(produto)` para obter o valor correspondente dentro do loop.

As saídas serão as mesmas linhas (talvez em diferente ordem, mas comparando dentro de cada block): Exemplo de saída (ordem arbitrária):

```
Listando preços (entrySet):
Monitor = 799.0
```

```
Cadeira = 650.5
Teclado = 129.9
Mochila = 120.0
Mouse = 49.99

Listando preços (keySet + get):
Monitor = 799.0
Cadeira = 650.5
Teclado = 129.9
Mochila = 120.0
Mouse = 49.99
```

(Observe que aqui a ordem manteve-se a mesma nos dois loops, pois o HashMap não foi modificado entre as iterações e `entrySet` and `keySet` iterate in corresponding order – keySet likely yields keys in same iteration order as entrySet. Mas se ordem fosse diferente, o ponto não é a ordem e sim as content.)

Discussão: - A abordagem `entrySet()` é geralmente preferida, pois é direta e evita a chamada adicional `get()`. Em termos de eficiência, `entrySet()` fornece ambos valores sem busca, enquanto `keySet()` + `get` causes HashMap to recompute hash and find entry again for each key. Em O(1) médio isso não é grande custo, mas é redundante quando entrySet já tem o valor prontinho. - Em coleções grandes, essa diferença pode ser significativa. Além disso, o código com entrySet é mais claro (menos chamadas explícitas). - Alternativamente, poderíamos usar `forEach` com lambda: `precos.forEach((produto, price) -> System.out.println(produto + " = " + price));` mas isso é equivalente a entrySet loop.

- Novamente, a ordem exibida não tem relação com inserção: inserimos Mouse, Teclado, Monitor, Cadeira, Mochila, mas a iteração pode ter saído Monitor primeiro etc. Isso reforça: HashMap não ordena.

- **Procurando valores específicos:** Exemplo para nota 90:

```
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;
import java.util.List;

public class BuscaPorValor {
    public static void main(String[] args) {
        Map<String, Integer> notas = new HashMap<>();
        notas.put("Ana", 90);
        notas.put("Bruno", 70);
        notas.put("Carlos", 90);
        notas.put("Diana", 85);

        int alvo = 90;
        List<String> alunosComAlvo = new ArrayList<>();
        for (Map.Entry<String, Integer> entry : notas.entrySet()) {
            if (entry.getValue() == alvo) {
```

```
                alunosComAlvo.add(entry.getKey());
            }
        }

        if (alunosComAlvo.isEmpty()) {
            System.out.println("Nenhum aluno tirou nota " + alvo);
        } else {
            System.out.println("Alunos com nota " + alvo + ": " +
alunosComAlvo);
        }
    }
}
```

Explicação: Montamos o HashMap `notas` com quatro alunos. Queremos encontrar quem tirou a nota 90. - Criamos uma lista `alunosComAlvo` para coletar nomes. - Iteramos sobre `entrySet()` do mapa, verificando `entry.getValue() == alvo`. - Se igual, adiciona o nome (`entry.getKey()`) na lista. - No final, verificamos se lista está vazia ou não e exibimos.

Com nossos dados, `alvo = 90` match "Ana" e "Carlos". Então a saída seria:

```
Alunos com nota 90: [Ana, Carlos]
```

(Poderia ser [Carlos, Ana] dependendo da ordem do HashMap iteration, mas lista não garante ordem original de map, though in entrySet, the order is some consistent but unpredictable. It's fine.)

Observações: - Usamos `==` para comparar Integer aqui porque estamos lidando com auto-unboxing ou just treating the int values. Alternatively, `entry.getValue().equals(alvo)` (with alvo as Integer) would be another way. But int compare is fine. - Esse código ilustra porque `containsValue` sozinho não basta: `notas.containsValue(90)` apenas retorna `true` / `false` dizendo se *existe algum 90*, mas não informa quais chaves correspondem. A gente precisava percorrer para achar todas as ocorrências. - Em termos de eficiência, se o mapa for muito grande e essa busca frequente, não é muito eficiente pois é O(n). Em tais casos, talvez um mapa inverso (valor -> lista de nomes) seria útil, ou usar Multimap. Mas dentro do que se pede, iterar é a forma correta.

1. **Encontrando o maior valor (máxima nota):**

```
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;
import java.util.List;

public class MelhorAluno {
    public static void main(String[] args) {
        Map<String, Integer> notas = new HashMap<>();
        notas.put("Ana", 90);
        notas.put("Bruno", 70);
        notas.put("Carlos", 90);
        notas.put("Diana", 85);
```

```
        // Primeiro passo: encontrar a nota máxima
        int maxNota = Integer.MIN_VALUE;
        for (int nota : notas.values()) {
            if (nota > maxNota) {
                maxNota = nota;
            }
        }

        // Segundo passo: coletar todos com nota = maxNota
        List<String> topAlunos = new ArrayList<>();
        for (Map.Entry<String, Integer> entry : notas.entrySet()) {
            if (entry.getValue() == maxNota) {
                topAlunos.add(entry.getKey());
            }
        }

        System.out.println("Nota máxima = " + maxNota);
        System.out.println("Alunos com nota máxima: " + topAlunos);
    }
}
```

Explicação: Usamos dois loops: - O primeiro loop percorre `notas.values()` para achar o maior inteiro. Começamos `maxNota` no menor valor possível para garantir qualquer nota será maior. Ao fim, `maxNota` terá a maior nota presente. - O segundo loop percorre `entrySet()` e seleciona todos cujos valores equivalem a `maxNota`, adicionando os nomes na lista `topAlunos`. - Por fim, exibimos a nota máxima e os alunos correspondentes.

Com os dados exemplo, as notas são 90, 70, 90, 85. A máxima é 90. Os alunos correspondentes: Ana e Carlos (ambos têm 90).

Saída:

```
Nota máxima = 90
Alunos com nota máxima: [Ana, Carlos]
```

Novamente, a lista possivelmente `[Carlos, Ana]` dependendo da ordem de inserção (HashMap isn't ordered, but presumably iteration gave maybe Bruno, Diana, etc. Actually, it might depend. But anyway, list might not be sorted alphabetically unless we sort it. But fine.)

Esse exercício reforça como extrair informações de um Map: - Iterar valores para um critério global (max). - Depois iterar entradas para encontrar quem atende o critério. - Poderia ser otimizado para um único loop que mantém controle de max e coleta nomes ao mesmo tempo, mas dividimos em dois passos claros.

1. **Uso de null em HashMap:** Vamos demonstrar a aceitação de null:

```
import java.util.HashMap;
import java.util.Map;
```

```
public class TestaNullHashMap {
    public static void main(String[] args) {
        Map<String, String> mapa = new HashMap<>();
        mapa.put(null, "valorNulo");            // chave nula com valor não-
nulo
        mapa.put("chaveNull", null);            // chave não-nula com valor
nulo
        mapa.put(null, "outroValor");           // atualiza valor da chave
nula existente

        // Recuperando valores
        String valorChaveNula = mapa.get(null);
        String valorChaveNull = mapa.get("chaveNull");

        System.out.println("Valor armazenado para chave null: " +
valorChaveNula);
        System.out.println("Valor armazenado para chave \"chaveNull\": " +
valorChaveNull);
        System.out.println("Mapa completo: " + mapa);
    }
}
```

Explicação: Em HashMap: - A primeira `put(null, "valorNulo")` insere um mapeamento cuja chave é null. - A segunda `put("chaveNull", null)` insere um mapeamento cuja chave é "chaveNull" e valor é null. - A terceira `put(null, "outroValor")` tenta inserir novamente com chave null. Como já existe uma entrada de chave null, ela não cria uma segunda – em vez disso, ela **sobrescreve** o valor associado à chave null com "outroValor". (HashMap só pode ter uma chave null, então a operação substitui o valor anterior.)

Depois usamos `get(null)` e `get("chaveNull")`: - `get(null)` deve retornar "outroValor" (o valor atual associado à chave null). - `get("chaveNull")` retorna null (porque esse é o valor armazenado, não porque não existe; se quiséssemos distinguir, usaríamos `containsKey("chaveNull")` que seria true e `mapa.get("chaveNull")` = null).

Imprimimos esses e o mapa inteiro.

Saída esperada:

```
Valor armazenado para chave null: outroValor
Valor armazenado para chave "chaveNull": null
Mapa completo: {null=outroValor, chaveNull=null}
```

Isso confirma: - HashMap permite chave null e valores null. No print do mapa, vemos `{null=outroValor, chaveNull=null}`. - Apenas uma chave null existe; o valor final para null é "outroValor" (mostrando que a segunda inserção substituiu "valorNulo"). - `containsKey(null)` seria true aqui. `containsValue(null)` também seria true (porque há um value null for "chaveNull"). - Esse comportamento difere do Hashtable (como vimos antes com exceptions). - Uso de null keys: internamente, HashMap trata chave null de forma especial (coloca no bucket 0 por convenção). Por isso

permite uma. - Cuidados: se usamos `map.get(x)` e ele retorna null, pode significar "ou a chave x não existe ou existe e o valor é null". Para distinguir, usamos `map.containsKey(x)` separadamente. - Nesse exemplo, `mapa.get("chaveNull")` retornou null, mas `mapa.containsKey("chaveNull")` seria true. Se consultássemos uma chave ausente, digamos `mapa.get("inexistente")`, retornaria null mas `containsKey("inexistente")` seria false, assim saberíamos que null indicou ausência.

1. **Invertendo um mapa (swap keys/values):**

```
import java.util.HashMap;
import java.util.Map;

public class InverterMapa {
    public static void main(String[] args) {
        Map<String, String> paisParaCodigo = new HashMap<>();
        paisParaCodigo.put("Brasil", "BR");
        paisParaCodigo.put("Estados Unidos", "US");
        paisParaCodigo.put("China", "CN");

        // Invertendo o mapa
        Map<String, String> codigoParaPais = new HashMap<>();
        for (Map.Entry<String, String> entry : paisParaCodigo.entrySet()) {
            String pais = entry.getKey();
            String codigo = entry.getValue();
            // Assumindo que 'codigo' é único
            codigoParaPais.put(codigo, pais);
        }

        System.out.println("Mapa original (País->Código): " +
paisParaCodigo);
        System.out.println("Mapa invertido (Código->País): " +
codigoParaPais);
    }
}
```

Explicação: Temos `paisParaCodigo`: - Brasil -> BR - Estados Unidos -> US - China -> CN

Criamos um novo HashMap `codigoParaPais`. Iteramos sobre cada entry do original: - Pegamos a chave (pais) e valor (codigo). - No invertido, usamos `codigo` como chave e `pais` como valor.

Imprimimos ambos mapas. Saídas:

```
Mapa original (País->Código): {Brasil=BR, Estados Unidos=US, China=CN}
Mapa invertido (Código->País): {BR=Brasil, US=Estados Unidos, CN=China}
```

(A ordem de impressão dos mapas pode diferir devido hashing, mas os pares estarão corretos.)

Considerações: - Supusemos que todos os códigos eram únicos. Se houvesse dois países com mesmo código (o que não faz sentido no nosso contexto, mas genericamente invertendo um map pode

acontecer valores duplicados), o último substituiria o primeiro no `codigoParaPais`. Por exemplo, se tanto "Brasil" quanto "Belize" tivessem código "BZ", no invertido só um ficaria. Mas assumimos exclusividade para este exercício. - Esse procedimento de inversão é útil às vezes, mas requer valor único ou uma estrutura de map para lista de keys se valores repetem. - Mostramos como iterar `entrySet()` e inserir em outro map. A complexidade é O(n). HashMap não tem método pronto para inverter, deve ser manual.

1. **Mesclando dois HashMaps:** Exemplo:

```
import java.util.HashMap;
import java.util.Map;

public class MesclarVendas {
    public static void main(String[] args) {
        Map<String, Integer> vendasMes1 = new HashMap<>();
        vendasMes1.put("Calçado", 20);
        vendasMes1.put("Camisa", 15);

        Map<String, Integer> vendasMes2 = new HashMap<>();
        vendasMes2.put("Calça", 10);
        vendasMes2.put("Camisa", 5);

        // Combinar vendas dos dois meses
        Map<String, Integer> vendasTotal = new HashMap<>(vendasMes1);
        for (Map.Entry<String, Integer> entry : vendasMes2.entrySet()) {
            String produto = entry.getKey();
            int qtd = entry.getValue();
            if (vendasTotal.containsKey(produto)) {
                vendasTotal.put(produto, vendasTotal.get(produto) + qtd);
            } else {
                vendasTotal.put(produto, qtd);
            }
        }

        System.out.println("Vendas Mês 1: " + vendasMes1);
        System.out.println("Vendas Mês 2: " + vendasMes2);
        System.out.println("Vendas Totais: " + vendasTotal);
    }
}
```

Explicação: Temos: - `vendasMes1`: {Calçado=20, Camisa=15} - `vendasMes2`: {Calça=10, Camisa=5}

Queremos somar por produto. Estratégia: - Iniciar `vendasTotal` como cópia de vendasMes1 (assim já contém Calçado:20, Camisa:15). - Iterar sobre vendasMes2: - Para cada produto em mes2, se já está em total, soma quantidades. - Se não está, adiciona novo.

Após o loop: - "Calçado": veio só do mês1 (20). - "Camisa": no mês1 15, no mês2 5, somou 20. - "Calça": só no mês2 (10). Resultado `vendasTotal` deve ser {Calçado=20, Camisa=20, Calça=10}.

Imprimimos todos maps (a ordem não garantida, mas por clareza):

```
Vendas Mês 1: {Calçado=20, Camisa=15}
Vendas Mês 2: {Calça=10, Camisa=5}
Vendas Totais: {Calçado=20, Camisa=20, Calça=10}
```

No código, usamos `containsKey` e `get` para decidir a soma. Outra maneira: usar `merge`:

```
vendasMes2.forEach((produto, qtd) ->
    vendasTotal.merge(produto, qtd, Integer::sum));
```

Isso automaticamente soma se existe ou adiciona se não. Mas mostramos o manual para clareza.

Assim, cobrimos conflitos de chave ("Camisa" existia nos dois, então somamos 15+5). Outros chaves simplesmente inserimos.

Essa técnica aplica-se a combinar contagens de diferentes fontes, consolidar dados de dois períodos, etc.

1. **Filtrando um HashMap com Streams:** Exemplo:

```
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

public class FiltrarProdutos {
    public static void main(String[] args) {
        Map<String, Double> produtos = new HashMap<>();
        produtos.put("TV", 2500.0);
        produtos.put("Smartphone", 1999.99);
        produtos.put("Notebook", 3200.0);
        produtos.put("Mouse", 50.0);
        produtos.put("Teclado", 130.0);

        // Filtrar produtos com preço abaixo de 1000.0
        double limite = 1000.0;
        Map<String, Double> baratos = produtos.entrySet().stream()
            .filter(entry -> entry.getValue() < limite)
            .collect(Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue));

        System.out.println("Todos os produtos: " + produtos);
        System.out.println("Produtos com preço abaixo de " + limite + ": " +
baratos);
    }
}
```

Explicação: Utilizamos a API de Streams: - `produtos.entrySet().stream()` cria um stream de todas as entradas (cada entrada é Map.Entry<String, Double>). - `.filter(entry ->`

`entry.getValue() < limite)` mantém apenas entradas cujo valor (preço) seja menor que 1000.0. - `.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue))` coleta as entradas filtradas em um novo Map, usando a key original e value original.

No exemplo, `produtos` tinha: {TV=2500.0, Smartphone=1999.99, Notebook=3200.0, Mouse=50.0, Teclado=130.0}.

A filtragem `preço < 1000` pega "Mouse" (50.0) e "Teclado" (130.0) somente. O map `baratos` então fica {Mouse=50.0, Teclado=130.0}.

Imprimimos:

```
Todos os produtos: {Teclado=130.0, Mouse=50.0, TV=2500.0, Notebook=3200.0,
Smartphone=1999.99}
Produtos com preço abaixo de 1000.0: {Teclado=130.0, Mouse=50.0}
```

(A ordem de impressão do HashMap original e do coletado não é garantida. Aqui aparece Teclado, Mouse, TV, Notebook, Smartphone - isso é uma ordem qualquer. O filtrado mostrou Teclado, Mouse, talvez invertido dependendo of internal but anyway includes only those two.)

Vantagens da abordagem funcional: - Podemos encadear filtragens e transformações de forma declarativa. - Evita criar manualmente um Map resultante e escrever loops explícitos; o código é conciso e expressa *o que* filtrar e coletar. - Atenção: `Collectors.toMap` por padrão retorna um HashMap (não ordenado). Se quiséssemos por exemplo LinkedHashMap para manter alguma ordem do stream, poderíamos usar an overload: `Collectors.toMap(..., ..., (v1,v2)->v1, LinkedHashMap::new)` mas não necessário aqui.

Essa técnica pode também ser usada para transformar os valores: por ex, `.map(entry -> Map.entry(entry.getKey(), entry.getValue() * 1.1))` para aumentar preços 10% antes de coletar, etc.

Respostas Seção 6: LinkedHashMap

1. **Ordem de chegada (maratona):** Exemplo de uso:

```java
import java.util.LinkedHashMap;
import java.util.Map;

public class Maratona {
    public static void main(String[] args) {
        Map<String, Double> tempos = new LinkedHashMap<>();
        // Inserindo na ordem de chegada: Nome -> Tempo (horas)
        tempos.put("Ana", 3.2);
        tempos.put("Bruno", 3.5);
        tempos.put("Carlos", 4.0);
        tempos.put("Daniel", 4.5);

        System.out.println("Classificação final:");
        tempos.forEach((nome, tempo) ->
```

```
            System.out.println(nome + " - " + tempo + " horas"));

        // Removendo um corredor (por exemplo, Bruno desqualificado)
        tempos.remove("Bruno");

        System.out.println("\nApós desclassificação de Bruno, classificação
ajustada:");
        for (Map.Entry<String, Double> entry : tempos.entrySet()) {
            System.out.println(entry.getKey() + " - " + entry.getValue() + "
horas");
        }
    }
}
```

Explicação: Criamos um `LinkedHashMap<String, Double>` chamado `tempos` para registrar o tempo dos corredores. `put` é chamado na ordem em que supomos que os atletas terminaram: - Ana primeiro, Bruno segundo, Carlos terceiro, Daniel quarto.

Ao iterar (usando `forEach` ou entrySet loop), a ordem de saída reflete a ordem de chegada:

```
Classificação final:
Ana - 3.2 horas
Bruno - 3.5 horas
Carlos - 4.0 horas
Daniel - 4.5 horas
```

Então removemos Bruno (`tempos.remove("Bruno")`). Agora Bruno sai do mapa. Ao iterar novamente, veremos:

```
Após desclassificação de Bruno, classificação ajustada:
Ana - 3.2 horas
Carlos - 4.0 horas
Daniel - 4.5 horas
```

Notar que: - Ana permanece em primeiro, Carlos e Daniel avançam uma posição na listagem mas **mantêm entre si** a ordem original de chegada. - Não há "buraco" deixado por Bruno; simplesmente Bruno não aparece mais.

O LinkedHashMap manteve a ordem de inserção original para os elementos restantes: - Bruno foi removido, mas Ana (inserida antes) e Carlos, Daniel (inseridos depois de Bruno) continuam na sequência relativa original.

Isso valida o uso de LinkedHashMap para cenários onde a ordem de inserção representa alguma sequência real (como posições de chegada).

1. **Comparando LinkedHashMap e HashMap na prática:**

```java
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;

public class CompararMapas {
    public static void main(String[] args) {
        Map<Integer, String> hashMap = new HashMap<>();
        Map<Integer, String> linkedMap = new LinkedHashMap<>();

        // Inserindo pares 1->"um", 2->"dois", 3->"três"
        hashMap.put(1, "um");
        hashMap.put(2, "dois");
        hashMap.put(3, "três");

        linkedMap.put(1, "um");
        linkedMap.put(2, "dois");
        linkedMap.put(3, "três");

        System.out.println("Iterando HashMap:");
        for (Map.Entry<Integer, String> e : hashMap.entrySet()) {
            System.out.println(e.getKey() + " -> " + e.getValue());
        }

        System.out.println("\nIterando LinkedHashMap:");
        for (Map.Entry<Integer, String> e : linkedMap.entrySet()) {
            System.out.println(e.getKey() + " -> " + e.getValue());
        }
    }
}
```

Explicação: Inserimos as mesmas três entradas em cada mapa. Ao iterar: - O HashMap pode imprimir numa ordem imprevisível. Por exemplo, possivelmente `2 -> dois`, `1 -> um`, `3 -> três` (ou outra variação). - O LinkedHashMap deve imprimir exatamente `1 -> um`, `2 -> dois`, `3 -> três` porque preserva inserção.

Uma possível saída:

```
Iterando HashMap:
2 -> dois
1 -> um
3 -> três

Iterando LinkedHashMap:
1 -> um
2 -> dois
3 -> três
```

A explicação: - HashMap: sem ordem definida. A sequência resultante resulta dos detalhes internos (que dependem de hash dos keys 1,2,3 e a rehashing maybe, etc.). Não devemos contar com nenhuma ordem aqui. - LinkedHashMap: mantém a ordem de inserção. Sempre que iterarmos, enquanto não modificarmos, a ordem será 1, 2, 3.

Também internamente, LinkedHashMap mantém uma estrutura ligada entre entradas; isso introduz um pequeno overhead de memória e processamento comparado a HashMap, mas garante essa previsibilidade.

Em geral, se precisamos que a iteração ocorra na ordem em que colocamos os elementos (como leitura de dados em uma ordem específica que deve ser mantida), LinkedHashMap é a estrutura a escolher em vez de HashMap.

1. **Acesso vs Inserção (accessOrder):** Este exemplo é mais avançado:

```
import java.util.LinkedHashMap;
import java.util.Map;

public class CacheAccessOrder {
    public static void main(String[] args) {
        // LinkedHashMap com accessOrder = true
        LinkedHashMap<Integer, String> cache = new LinkedHashMap<>(16,
0.75f, true);

        cache.put(1, "Page1");
        cache.put(2, "Page2");
        cache.put(3, "Page3");

        // Ordem inicial de inserção (accessOrder não manifesto sem acessos)
        System.out.println("Ordem inicial: " + cache.keySet());

        // Acessar a chave 1 (Page1)
        cache.get(1);
        // Inserir nova página
        cache.put(4, "Page4");

        System.out.println("Ordem após acessar 1 e inserir 4: " +
cache.keySet());

        // Acessar a chave 3 (Page3)
        cache.get(3);

        System.out.println("Ordem após acessar 3: " + cache.keySet());
    }
}
```

Explicação: Criamos um LinkedHashMap `cache` com o terceiro parâmetro `accessOrder=true`. Isso significa que a ordem de iteração será determinada pelas operações de acesso (get/put) além da ordem de inserção. - Inserimos 1,2,3 (insertion order now 1,2,3). - `cache.keySet()` initially would return [1, 2, 3]. - Then `cache.get(1)` is called. This is an access of key 1, so in an access-order map,

key 1 is moved to the **end** of the order (as most recently used). - Then `cache.put(4, "Page4")` adds a new entry 4 at the end. - Now the order should be: 2, 3, 1, 4. - Explanation: 1 moved to end after accessed, then 4 appended at end after insertion. - We print `cache.keySet()` which will show [2, 3, 1, 4]. - Then we do `cache.get(3)`. Key 3 will be moved to end of order (because accessOrder true and it was accessed). - Now order becomes: 2, 1, 4, 3. - Let's trace: before access it was 2,3,1,4. Accessing 3 removes it from its position and moves to end: making 2,1,4,3. - Print `cache.keySet()` => [2, 1, 4, 3].

Thus the output (key sets) might be:

```
Ordem inicial: [1, 2, 3]
Ordem após acessar 1 e inserir 4: [2, 3, 1, 4]
Ordem após acessar 3: [2, 1, 4, 3]
```

Explicação do fenômeno: - Em LinkedHashMap com accessOrder=true, qualquer leitura ou escrita de um par existente conta como "acesso" e faz o par ser movido para o fim (mais recentemente usado). - No passo 1: Access 1 -> sequence becomes [2,3,(1)]. - No passo 2: Insert 4 -> sequence [2,3,1,(4)]. - No passo 3: Access 3 -> sequence [2,1,4,(3)].

Isso simula o comportamento de cache LRU (Least Recently Used), onde o item mais velho "não usado" fica no início. Por exemplo, após primeiro acesso e inserção, o item 2 ficou como primeiro (mais antigo não acessado recentemente). Depois do acesso em 3, item 2 e 1 e 4 estão antes de 3; 2 seria o mais antigo desde que não foi acessado durante esse período.

Esse recurso é útil se, por exemplo, quiséssemos ejetar o item mais antigo; ver próxima questão sobre removeEldestEntry.

1. **Implementando um cache LRU simples:** Exemplo:

```
import java.util.LinkedHashMap;
import java.util.Map;

class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private int capacidade;

    public LRUCache(int capacidade) {
        super(capacidade, 0.75f, true); // accessOrder = true
        this.capacidade = capacidade;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        // Remove o item mais antigo se exceder capacidade
        return this.size() > capacidade;
    }
}

public class CacheLRUDemo {
    public static void main(String[] args) {
        LRUCache<Integer, String> cache = new LRUCache<>(3);
```

```
        cache.put(1, "A");
        cache.put(2, "B");
        cache.put(3, "C");
        System.out.println("Cache inicial: " + cache);

        cache.get(1); // acessa 1, tornando-o mais recente
        cache.put(4, "D"); // insere D, cache excederia capacidade, remove
mais antigo (que deve ser a chave 2 agora)
        System.out.println("Cache após inserir 4 (capacidade excedida,
removeu o mais antigo): " + cache);

        cache.get(2); // tenta acessar chave 2 (que já foi removida)
        cache.put(5, "E"); // insere E, remove próximo mais antigo
        System.out.println("Cache após inserir 5: " + cache);
    }
}
```

Explicação: Criamos uma classe `LRUCache` que estende LinkedHashMap e define a capacidade máxima. No construtor, passamos `true` para accessOrder para que o cache seja ordenado por acesso (LRU behavior). Sobrescrevemos `removeEldestEntry` para retornar true quando o tamanho excede a capacidade, forçando remoção automática do mais antigo.

No uso: - Capacidade 3. - put 1->A, 2->B, 3->C (cache cheio com 3 itens). Estado (em ordem de acesso = inserção pois nenhum acesso fora inserir): {1=A, 2=B, 3=C}. - Acessamos cache.get(1). Isso faz 1 tornar-se o mais recente, ordem agora: {2=B, 3=C, 1=A} (2 é o mais antigo, 1 o mais novo). - put 4->D. Ao inserir, LinkedHashMap verifica `removeEldestEntry`: size iria para 4 > capacidade(3), então retorna true e remove o eldest (que é 2=B). Assim, após inserir 4, chaves presentes: {3=C, 1=A, 4=D}. Ordem atual: 3 (antigo), 1, 4 (novo). - Print do cache: deveria mostrar `{3=C, 1=A, 4=D}`. - Depois `cache.get(2)`: 2 não está mais, então retorna null (podemos ignorar, serviu só para simular um acesso falho, sem efeito no cache). - put 5->E. Novamente would exceed capacity from 3 to 4 size, so remove eldest. Qual é o eldest agora? A ordem atual era {3, 1, 4}, eldest = 3. Então remove entry 3=C. Agora presentes: {1=A, 4=D, 5=E}. - Ordem final: {1, 4, 5} (onde 1 ainda mais antigo, 5 novo). - Print final: `{1=A, 4=D, 5=E}`.

Saídas do console:

```
Cache inicial: {1=A, 2=B, 3=C}
Cache após inserir 4 (capacidade excedida, removeu o mais antigo): {3=C, 1=A,
4=D}
Cache após inserir 5: {1=A, 4=D, 5=E}
```

(Obs.: O cache imprime em formato {key=value, ...} quando usamos System.out.println no LinkedHashMap due to its toString implementation inherited from AbstractMap.)

Isso mostra: - Depois de inserir 4, o item de chave 2 foi removido automaticamente (LRU). - Depois de inserir 5, o item de chave 3 foi removido (pois 3 era então o LRU após as operações anteriores).

Nosso acesso `get(1)` garantiu que 1 não fosse removido quando 4 entrou (porque 2 se tornou LRU). Se não tivéssemos acessado 1, a ordem teria sido {1,2,3} and eldest=1 removed on inserting 4.

Essa técnica é a base de caches: LinkedHashMap facilita pois removeEldestEntry nos poupa de ter que gerenciar a remoção manualmente.

1. **Atualização de valor sem alterar ordem:** Exemplo:

```
import java.util.LinkedHashMap;
import java.util.Map;

public class AtualizaValorLinkedHashMap {
    public static void main(String[] args) {
        Map<String, Integer> pontuacao = new LinkedHashMap<>();
        pontuacao.put("Alice", 10);
        pontuacao.put("Bob", 8);
        pontuacao.put("Carol", 15);

        System.out.println("Original: " + pontuacao.keySet());

        // Atualizar pontuação de Bob
        pontuacao.put("Bob", 12);

        System.out.println("Após atualizar Bob: " + pontuacao.keySet());

        // Para comparação, caso fosse accessOrder:
        LinkedHashMap<String, Integer> pontuacaoAccess = new
LinkedHashMap<>(16, 0.75f, true);
        pontuacaoAccess.put("Alice", 10);
        pontuacaoAccess.put("Bob", 8);
        pontuacaoAccess.put("Carol", 15);
        pontuacaoAccess.put("Bob", 12); // atualização conta como acesso no
modo accessOrder


System.out.println("Ordem no LinkedHashMap accessOrder após atualizar Bob: "
+ pontuacaoAccess.keySet());
    }
}
```

Explicação: Primeiro usamos um LinkedHashMap normal (insertion order): - Inserimos Alice, Bob, Carol (ordem: Alice, Bob, Carol). - Imprimimos as keys: [Alice, Bob, Carol]. - Depois fazemos `pontuacao.put("Bob", 12)`. Isso updates Bob's value from 8 to 12. In an insertion-ordered LinkedHashMap, updating an existing key **does not change its position**. Bob was already second in order and remains second. - Print keys again: still [Alice, Bob, Carol].

Then, for demonstration, we create another LinkedHashMap with accessOrder=true: - Insert Alice, Bob, Carol (initial order: Alice, Bob, Carol). - Then `pontuacaoAccess.put("Bob", 12)`. In access-order mode, even a `put` to an existing key is considered an access (as per spec). So this should move Bob to the end of the order. - The new order in `pontuacaoAccess` would be [Alice, Carol, Bob] (because Bob, updated, is now most recently accessed). - Print that.

So expected output:

```
Original: [Alice, Bob, Carol]
Após atualizar Bob: [Alice, Bob, Carol]
Ordem no LinkedHashMap accessOrder após atualizar Bob: [Alice, Carol, Bob]
```

This shows: - In a default LinkedHashMap (insertion-order), the order is unchanged after updating a value. The position of Bob stays where it was inserted originally. Only the value changed. - In a LinkedHashMap with accessOrder enabled, updating Bob's value counts as an access, thus Bob moves to end.

This difference is important: If you rely on insertion order and update values, LinkedHashMap suits well as it doesn't shuffle on updates. But if using access-order (like an LRU cache), any use of an entry (get or put) will affect order.

Usually, LinkedHashMap in insertion mode is default and often what's needed. Access-order is a special scenario, typically for caches.
