Algoritmos e Estruturas de Dados

2ª Série

(Problema)

Análise de ocorrências de k-mers

N. 50548 Nome: Gonçalo Amigo
N. 50518 Nome: Rita Monteiro
N. 50500 Nome: Humberto Carvalho

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2023/2024

13/05/2024

Índice

1. Introdução

O projeto em mãos, tem como objectivo o desenvolvimento duma classe que implementa o tipo de dados `HashMap` (Tabela de Distribuição) na linguagem *Kotlin,* que armazene no âmbito do enunciado, numa `HashTable` todas instâncias dos K-mers, dum certo ficheiro.

Neste relatório serão descritas todas as implementações necessárias para o desenvolvimento do tipo abstrato desenvolvido, `AEDHashMap` de modo a obter os resultados pretendidos.

2. Número de ocorrências de k-mers

2.1 Análise do problema

O problema em análise, como já foi referido anteriormente, baseia-se na contagem de ocorrências dos *K-mers* num ficheiro de texto.

Para tal implementação é necessário ter em conta alguns aspetos como, qual a estrutura de dados mais adequada em termos de eficiência para a resolução do problema em mãos.

A estrutura escolhida foi uma Tabela de Dispersão, por inúmeras razões que serão descritas na secção 2.2.

De modo a implementar esta estrutura, foi necessário a criação dum tipo de dados abstratos que segue o contrato descrito na interface `MutableMap`, isto é, define um conjunto de métodos/comportamentos e propriedades a implementar pela classe que representará o ADT do `HashMap`. Desta forma, de acordo a interface implementa os métodos necessários para a resolução do problema, sendo esses métodos:

- *HashNode<K, V>()*
- *hashFunction()*
- *operator get()*
- *Getter* referente às keys do HashMap
- *Iterator* (e suas implementações)
- *put()*
- *contains()*
- *needToExpand()*
- *expand()*

2.1.1 HashNode<K, V>()

A classe HashNode é uma das partes essenciais da implementação da estrutura de dados HashMap. Ela representa um nó individual dentro do HashMap, armazenando uma chave (*key*), um valor (*value*) e uma referência ao próximo nó em caso de colisões.

Tem como propósito fornecer uma estrutura para armazenar as informações no HashMap. Em que cada instância da classe representa um par chave-valor e pode ser armazenada em um dos índices do HashMap.

➔ Estrutura de Dados:
- key → Armazena a chave associada ao nó.
- value → Armazena o valor associado à chave.
- next → Referência ao próximo nó em caso de colisões.

A classe `HashNode` implementa a interface `MutableMap.MutableEntry<K, V>`, que consequentemente a torna compatível com a estrutura de dados HashMap. Esta interface gera métodos para acessar e modificar a chave e o valor armazenados no nó:

➔ `setValue(newValue: V): V`
- Este método permite modificar o valor associado ao nó e retorna o valor anterior.

Tendo em consideração o seu desempenho, a classe `HashNode` é otimizada, prometendo um acesso eficiente às chaves e valores armazenados nos nós. A utilização de listas encadeadas para resolver colisões contribui para manter um desempenho consistente mesmo em situações de alta carga.

Em resumo, a classe `HashNode` desempenha um papel essencial na implementação da estrutura de dados HashMap, fornecendo uma representação eficiente e flexível para armazenar os elementos individuais. Já a sua estrutura e o seu comportamento são projetados para garantir um desempenho robusto e escalável em uma variedade de cenários de uso.

2.1.2 Função `hashFunction()`

Para a implementação dum `HashMap`, existe um aspeto importante a ter em conta, que são as colisões quando o utilizador quer adicionar um elemento à estrutura de dados.

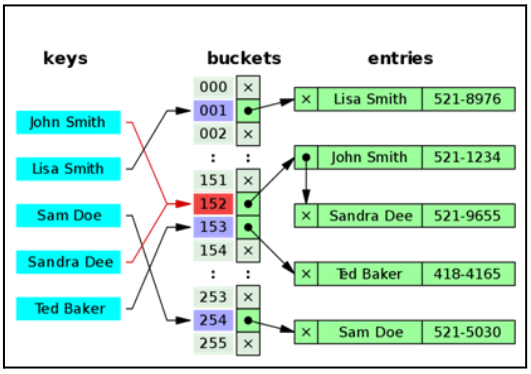

Figura 1 - Colisões num `HashMap` baseado em encadeamento externo

Quando o utilizador quer adicionar algum elemento ao `HashMap`, tem que ter um índice ao qual o inserir, pois as inserções não são feitas a começar na primeira posição e a acabar na última, as mesmas são feitas através dum cálculo que na maior parte dos casos atribui a cada par de valores um índice único, mas caso isso não aconteça, no contexto presente do problema, será adicionado a uma lista ligada os elementos que colidiram, como demonstrado na Figura 1.

Logo esta função de dispersão é essencial tanto para adicionar, como para pesquisar sobre a tabela de dispersão.

2.1.3 Função `operator get()`

A função get é um método essencial na implementação da estrutura de dados `HashMap`. Ela é responsável por recuperar o valor associado a uma chave específica dentro do `HashMap`.

Tem como propósito fornecer uma maneira eficiente de acessar os valores armazenados no `HashMap`, fornecendo uma chave específica. Isto permite que os usuários obtenham rapidamente o valor correspondente a essa chave, independentemente do tamanho do `HashMap`.

➔ A função `get` opera da seguinte forma:

- Calcula o índice no array interno do `HashMap` para a chave fornecida usando uma função de dispersão (hash function).
- Acessa o compartimento correspondente no array interno.
- Se houver um nó presente no compartimento, verifica se a chave corresponde à chave procurada.
- Se houver correspondência, retorna o valor associado a essa chave.
- Se não houver correspondência ou o compartimento estiver vazio, retorna null.

Na implementação do `HashMap`, a função get é chamada quando um usuário deseja recuperar o valor associado a uma determinada chave. Isto é útil para consultar os valores armazenados no `HashMap` sem precisar percorrer todos os elementos.

A função `get` é otimizada para ter um desempenho eficiente, garantindo acesso rápido aos valores associados às chaves. A utilização de uma função de dispersão eficiente e a estruturação dos compartimentos do `HashMap` ajudam a minimizar o tempo de busca, mesmo em `HashMaps` grandes.

Em resumo, a função get desempenha um papel crucial na acessibilidade e eficiência da estrutura de dados `HashMap`. Devido à sua implementação permite recuperar rapidamente os valores associados às chaves, contribuindo para o desempenho geral e a utilidade do HashMap em uma variedade de aplicativos.

2.1.4 `Getter referente às keys do HashMap`

A implementação desta função tem como principal objetivo auxiliar na implementação do problema de forma a facilitar a obtenção das chaves que não são nulas no `HashMap` original.

Logo é percorrido o `HashMap` original, e adicionado a um novo `HashMap` os conjuntos de chaves e valores associados.

2.1.5 `Iterator e as suas implementações`

De modo a implementar a estrutura dum iterador é necessário implementar a interface `Iterator`, isto porque a interface Iterator irá permitir a iteração sobre os "nós" definidos em `MutableEntry<Key, Value>`. Inicialmente é criada uma inner class `MyIterator` de modo a que seja possível aceder aos dados da classe externa contrariamente a uma classe normal. O iterator permite iterar sobre os vários "nós" que estão dentro do `HashMap`. Nesta classe existem dois métodos que dão auxílio à iteração sobre a tabela de dispersão, sendo esses a função `hasNext()` que verifica a existência de um nó seguinte, consoante a existência ou não, retorna um valor booleano associado, e a função `next()` que invoca o método `hasNext()` e caso retorne verdadeiro, irá retornar esse nó (node.next), caso retorne falso, será lançada uma exceção que referencia a falta de elemento seguinte.

2.1.6 Função `put()`

Este método tem como principal objetivo adicionar elementos à tabela de dispersão. Para adicionar elementos é necessário de acordo o valor retornado da função de dispersão e da função `contains()`, que indica se esse certa key já existe na tabela, se caso o elemento associado à chave seja null, significa que não há nenhum elemento na posição a analisar, logo é adicionado à cabeça da lista ligada o elemento, caso o índice já tenha elementos, é também adicionado à cabeça o conjunto <Chave, Valor>, como é evidente no exemplo representado na Figura 2 e 3.

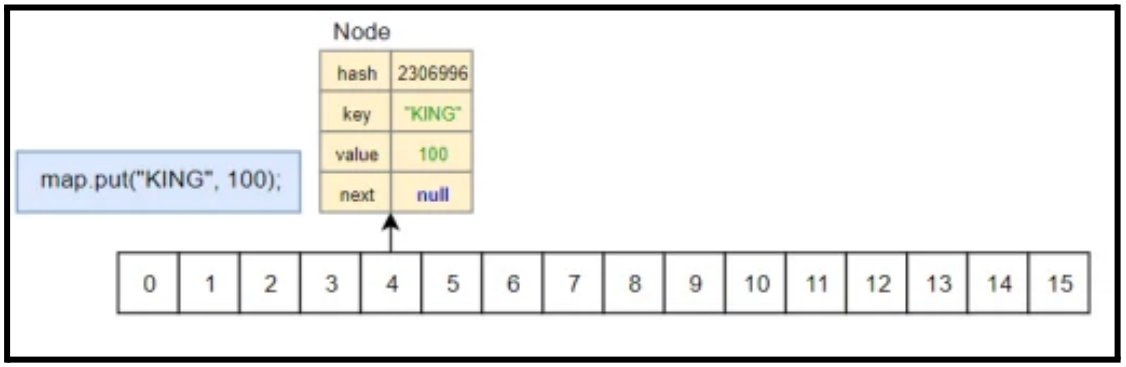

Figura 2 - Exemplo de adicionar elemento a uma Tabela de Dispersão caso não haja elementos nessa posição

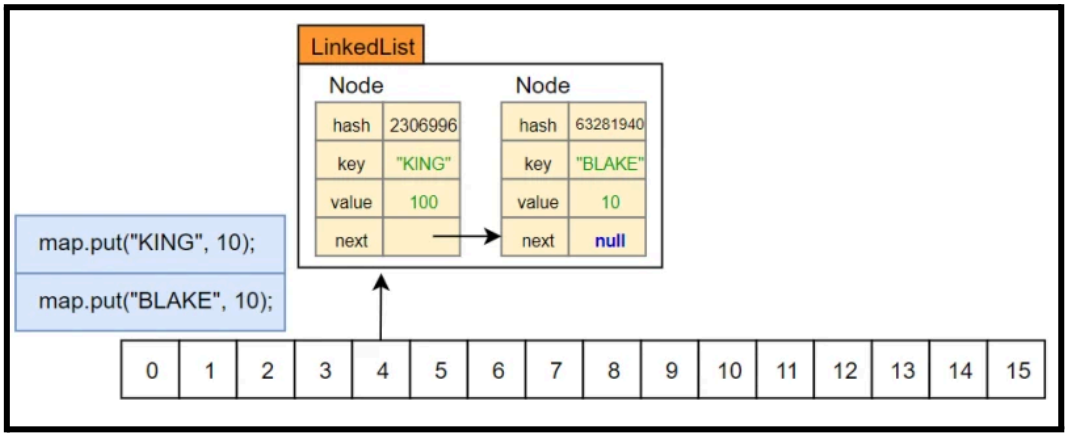

Figura 3 - Exemplo de adicionar elemento a uma Tabela de Dispersão caso já haja elementos nessa posição

2.1.7 Função `contains()`

A função contains é um método fundamental na implementação da estrutura de dados `HashMap`. Ela é responsável por verificar se uma determinada chave existe no `HashMap`.

O propósito da função contains é fornecer uma maneira eficiente de determinar se uma chave específica está presente no `HashMap`. Com isso, permite que os usuários verifiquem rapidamente a existência de uma chave antes de realizar operações como adição ou substituição de valores.

➔ A função `contains` opera da seguinte forma:

- Calcula o índice no array interno do `HashMap` para a chave fornecida usando uma função de dispersão (`hash function`).
- Acessa o compartimento correspondente no array interno.
- Se houver um nó presente no compartimento, verifica se a chave corresponde à chave procurada.
- Se houver correspondência, retorna o próprio nó que contém a chave procurada.
- Se não houver correspondência ou o compartimento estiver vazio, retorna null.

Na implementação do `HashMap`, a função contains é chamada quando um usuário deseja verificar se uma chave específica está presente no `HashMap`. Isso é útil para evitar duplicatas de chaves e garantir a integridade dos dados armazenados.

A função contains é otimizada para desempenho, garantindo uma verificação rápida da existência de uma chave no `HashMap`. A utilização de uma função de dispersão eficiente e a estruturação dos compartimentos do `HashMap` contribuem para minimizar o tempo necessário para determinar a presença de uma chave.

Em resumo, a função contains desempenha um papel crucial na verificação da existência de chaves dentro da estrutura de dados `HashMap`. Devido à sua implementação eficiente permite uma verificação rápida e confiável da presença de chaves, contribuindo para a robustez e utilidade geral do `HashMap` em várias aplicações.

2.1.8 Função `needToExpand()` e `expand()`

A função *`needToExpand()`* é evocada cada vez que é necessário ajustar o tamanho da `HashTable` para o dobro, visto que ao adicionar um elemento ao `HashMap`, o número de elementos multiplicado pelo fator de carga é igual ou superior à dimensão da tabela (como é referido no enunciado do projeto), sendo essa verificação implementada no método `needToExpand()`, assim desta forma possível garantir a integridade e linearidade do `HashMap`.

Caso seja a condição se verifique, é chamada a função `expand()` que cria uma tabela totalmente nova, inicializada com valores a null com o dobro do tamanho da tabela original. Após a criação dessa tabela é necessário povoá-la com os elementos já presentes na tabela original, por isso foi iterado sobre a tabela original de modo a verificar se o elemento corrente é diferente de `null`, a lista ligada associada a esse nó do `HashMap` deve ser passada para a nova tabela com um novo índice dado pela

função de dispersão. No final a nova tabela mantém os mesmos dados definidos anteriormente mas com o dobro do tamanho.

2.2 Estruturas de Dados Utilizadas

2.2.1 Estrutura de dados - `HashMap`

Como já foi referido na secção 2.1, a estrutura escolhida para a resolução do problema foi uma Tabela de Distribuição, visto que o tempo de execução ao adicionar elementos (tendo em conta que a implementação foi feita com base em encadeamento externo), é constante e tanto a procura na mesma como a remoção de algum elemento é proporcional à dimensão da lista. Por estas razões, esta estrutura será a mais acertada e eficiente.

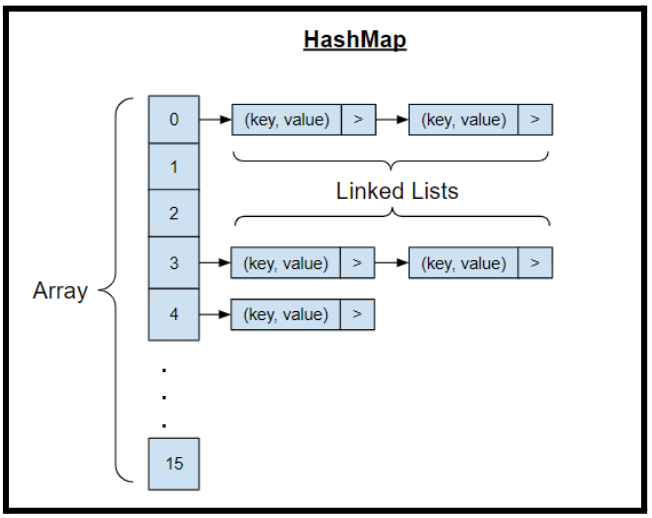

Figura 4 - Estrutura `HashMap` baseada em encadeamento externo

Um `HashMap` é uma estrutura que armazena valores de acordo um par entre uma chave e um valor associado a essa mesma chave (<K, V>), tendo uma posição desta tabela, de acordo uma função de dispersão, que calcula de acordo o cálculo matemático representado na Tabela 1.

Função de dispersão
$key \% dimTable$

Tabela 1- Fórmula usada para o cálculo do índice de cada posição

De acordo com o índice obtido são executadas diferentes operações de modo a adicionar, remover ou pesquisar elementos na `HashTable`.

O `HashMap` presente na Figura 4 é um exemplo de uma implementação deste tipo de dados, tem um array com elementos ou a null ou com listas ligadas (caso haja colisões adiciona elementos à lista).

2.2.2 Estrutura de dados - Linked List

De modo a implementar o `HashMap` baseado em encadeamento externo, foi necessário a implementação de listas ligadas.

Listas Ligadas `(Linked Lists)`, é uma estrutura baseada em listas mas com um conceito diferente, em vez de se iterar normalmente sobre uma lista de acordo com o seu índice, no contexto duma lista ligada apenas se consegue aceder a um certo elemento de acordo a referência para o próximo nó, sendo esta ligada numa direção.

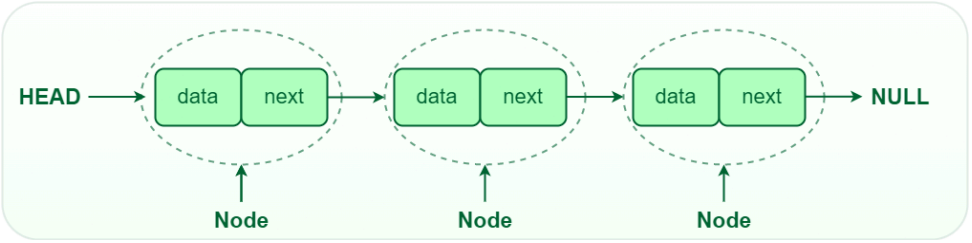

Figura 5 - Estrutura Linked List

Na Figura 5, está representado um exemplo de uma lista ligada, sendo o primeiro elemento da lista designado por `head`, que inicialmente é `null`.

Cada elemento de uma lista ligada é do tipo `Node`, que é caracterizado por uma propriedade `data`, que contém os dados associados àquele `Node` e a propriedade `next`, que contém a referência associada ao próximo `Node` que é o próximo elemento desta lista.

2.3 Algoritmos e análise da complexidade

2.3.1 Implementação de um algoritmo de contagem de ocorrência dos K-mers

De modo a implementar o problema proposto no projeto tendo a estrutura `HashMap` de auxílio é necessário ter em conta alguns pontos importantes, tais como:

- ➢ A leitura do ficheiro
- ➢ Adicionar ao `HashMap` as strings necessárias de modo a fazer a contabilização das ocorrências
- ➢ Iterar sobre o `HashMap` com o getter das keys, de modo a retornar num ficheiro de output os seus valores

Para a leitura do ficheiro foi utilizada a biblioteca `java.io.File` da linguagem Java, passando todo o conteúdo do ficheiro para uma `string`. Através dessa string com o conteúdo do ficheiro e o tamanho desejado para a contabilização dos *K-mers* (variável k), é iterado, a cada intervalo de zero a $k - 1$, sobre essa mesma string de modo a obter `substrings` que equivalem aos *K-mers* a serem contabilizados.

Caso a substring já se encontre no `HashMap`, o seu *value* será incrementado por 1, caso contrário, será inserida pela primeira vez com o valor de ocorrência igual a 1 (pois ocorreu uma vez até ao momento de análise da string).

Após construir o `HashMap`, é feita uma iteração sobre o mesmo de modo a apresentar todos os valores das keys e values (que no caso equivalem às diferentes substrings e às diferentes ocorrências) que não sejam `null` (caso não tenham preenchido todos os elementos do `HashMap`, haverá valores a `null`).

A apresentação dos resultados será escrita num ficheiro de output seguindo o formato: *key* - *value.*
Esta implementação foi aplicada para a função que utiliza a implementação do Kotlin `HashMap` e a implementação feita pelo grupo de trabalho.

3. Avaliação Experimental

De modo a observar a eficiência e funcionalidade das implementações do `HashMap` desenvolvido comparado com o `HashMap` do Kotlin, é necessário ter em conta os tempos de execução de cada estrutura.

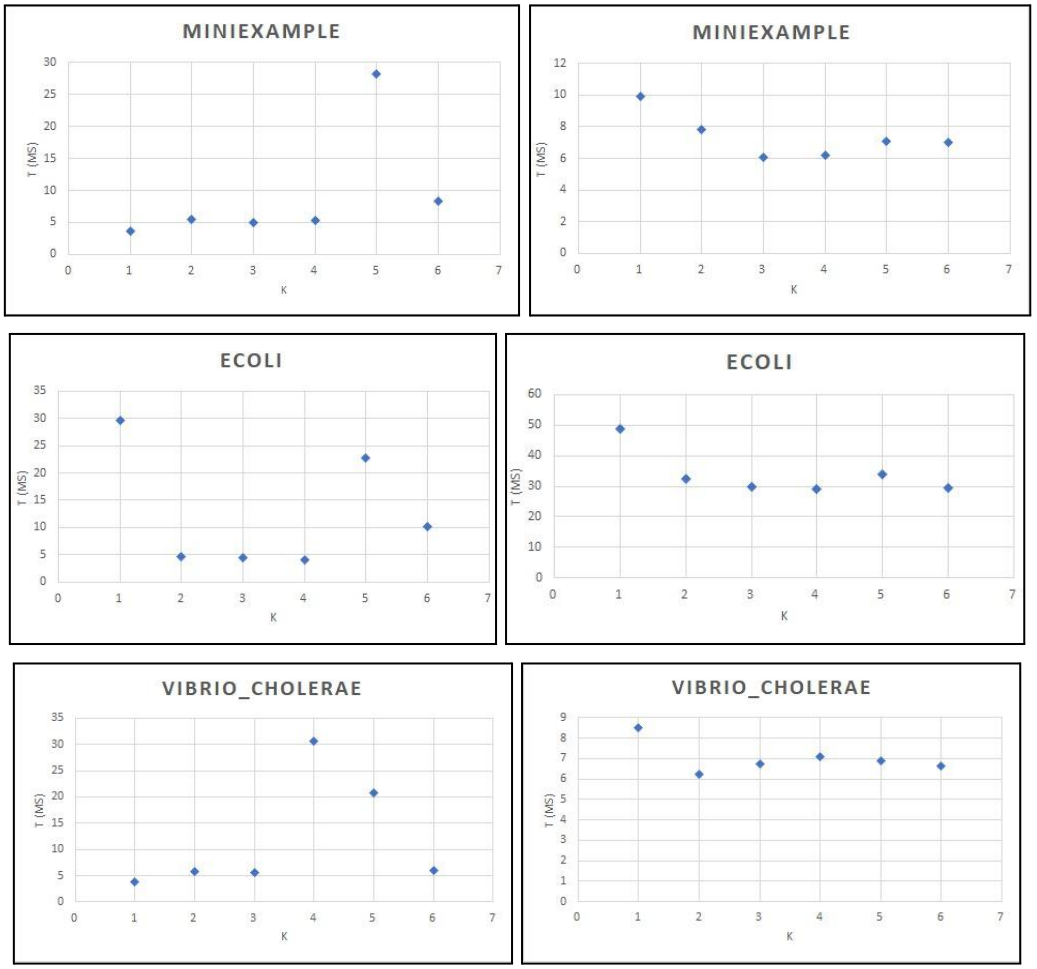

Tabela 2 - Tempos de execução dos ficheiros de teste para diferentes valores de k (`implementedHash()` e `kotlinHash()`)

Para a execução dos ficheiros presentes, os tempos de execução estão apresentados na Tabela 2 (referente ao `HashMap` implementado no projeto e ao `HashMap` do Kotlin respectivamente). É evidente que comparando o HashMap do Kotlin com aquele que foi implementado, o HashMap do Kotlin apresenta tempos mais eficientes independentemente do tamanho do ficheiro, pois este está otimizado no seu máximo de modo a ter uma melhor performance. O implementedHash() apesar de não apresentar os mesmos valores, para certos casos tem uma tendência mais positiva que o do kotlinHash(). Resumindo, para o problema em mãos o implementedHash() é específico

ao projeto, em certos casos é mais eficiente que o do Kotlin mas no geral, o kotlinHash() demonstra uma tendência constante, o que não acontece no implementedHash().

4. Conclusões

Em suma, no âmbito do projeto proposto foi possível compreender conceitos programáticos bastantes importantes, bem como diferentes estruturas que proporcionam a fácil implementação dos métodos necessários, sendo essas estruturas as Tabelas de Dispersão (`HashMap`) e as Listas Ligadas (`Linked Lists`).

De um modo geral, face ao primeiro projeto igualmente, este projeto foi desafiador uma vez que foi o primeiro contacto com este tipo de estruturas, mesmo assim é possível perceber a importância das mesmas.

5. Referências

Figura 1 - Colisões num HashMap baseado em encadeamento externo
https://levelup.gitconnected.com/java-hashmap-explained-a601c48ddc44

Figura 2 - Exemplo de adicionar elemento a uma Tabela de Dispersão caso não haja elementos nessa posição
https://javarush.com/pt/groups/posts/pt.2496.anlise-detalhada-da-classe-hashmap

Figura 3 - Exemplo de adicionar elemento a uma Tabela de Dispersão caso já haja elementos nessa posição
https://javarush.com/pt/groups/posts/pt.2496.anlise-detalhada-da-classe-hashmap

Figura 4 - Estrutura HashMap baseada em encadeamento externo
https://javarevisited.blogspot.com/2022/12/how-to-update-value-for-key-hashmap.html

Figura 5 - Estrutura Linked List
https://www.geeksforgeeks.org/what-is-linked-list/