Pontifícia Universidade Católica do Rio Grande do Sul
Escola Politécnica

Algoritmos e Estruturas de Dados II - Turma 128
Professor Diego Vrague

Trabalho 1 - Individual

João Gabriel Dourado Cervo
j.dourado@edu.pucrs.br

Maio de 2021

1. Introdução

No presente relatório está a minha solução proposta para o trabalho individual I da disciplina de Algoritmos e Estruturas de Dados II. O objetivo do trabalho reside em desenvolver uma classe chamada *ChainHashMap*, que é um dicionário simples capaz de armazenar pares (*chave*, *valor*) e resolver conflitos através de encadeamento. Tal classe recebe valores do tipo String tanto para as chaves quanto para os valores, a fim de simplificar o projeto. Também foi usado como padrão a linguagem em inglês para nomenclatura das variáveis e nomes de classe.

2. Solução

A fim de resolver a tarefa, a classe *ChainHashMap* foi implementada. Ela estende a classe *AbstractMap* e possui três variáveis privadas, a *capacity*, do tipo inteiro, *size*, do tipo inteiro, e os buckets. No caso do último, se deu a implementação através de um *ArrayList* de *Node*. Tal classe *Node* foi criada como uma classe privada, e dispõe de variáveis para armazenar o *key*, *value,* e uma referência para o próximo *Node*. Deste modo, é feita uma lista encadeada dentro de cada bucket, para caso o mesmo índice seja gerado. A classe *Node* pode ser visualizada na figura 1.

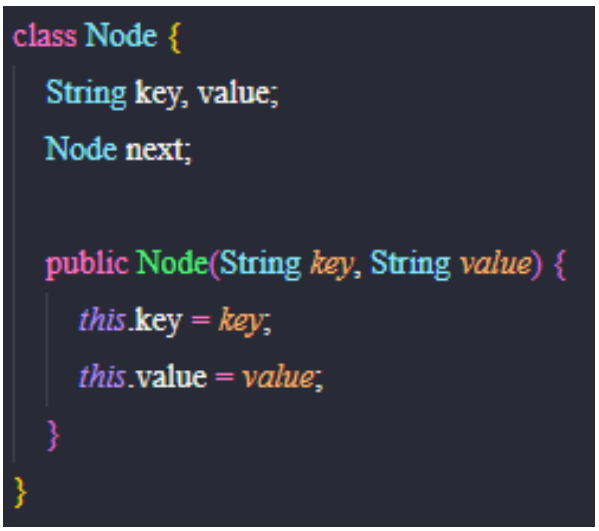

Figura 1: Implementação da classe *Node*

Fonte: Autoria Própria

Em seguida, foi realizada a implementação dos construtores da classe *ChainHashMap.* Um desses recebe uma variável inteira para capacidade, e outro é um construtor padrão caso nada seja passado. Também foi criado um método para inicialização dos *buckets*, assim o código fica mais limpo ao reutilizar funções iguais.

Foram criados alguns métodos privados para auxiliar a utilização interna de algumas funções. Sendo eles, *compress(long hash)*, que faz a compressão do hash para a capacidade dos *buckets*, *getIndex(String key)*, que pega o índice nos *buckets* para determinada *key*, *getHead(String key)*, que pega o primeiro item da lista encadeada em dada key, neste método se tem a utilização do *getIndex()*, descrito anteriormente. No *getIndex()* se dá a utilização do *compress().* Esses três métodos são a base da implementação e utilizados em diversos outros métodos. Eles podem ser vistos na figura 2. Também há um método para pegar o valor de carga, chamado de *getLoadFactor()*.

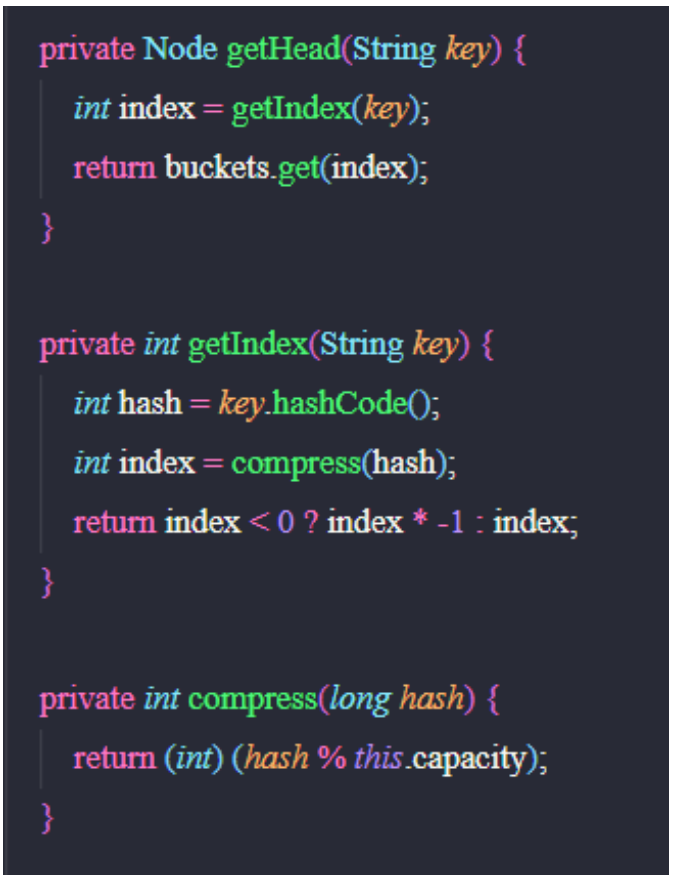

Figura 2: Implementação dos métodos base.

Fonte: Autoria Própria

Em relação às implementações gerais, há um método *get(String key)* que recebe uma key e retorna o valor associado a tal chave, ou nulo, caso nenhum. Um método *put(String key, string value)*, que adiciona um valor a determinada chave e retorna o valor antigo, ou nulo caso não haja nenhum valor associado a tal chave. Também é verificado o fator de carga, e caso esse seja maior que 0.75, chama-se o método *doubleCapacity()*, que dobra a capacidade dos buckets. Por fim, um método *remove(String key)*, que remove um valor associado a chave e o retorna, ou nulo caso não encontre. A fim de manter o relatório o mais sucinto e simples possível, a implementação desses métodos não será mostrada aqui. Mas todas estão documentadas no código, tal como métodos *iterable* para retornar valores e chaves.

2.0.1. Conflitos

Para evitar possíveis conflitos de *hashes* caindo no mesmo índice após a compressão, se deu a utilização de uma lista encadeada nos *buckets*. De tal modo que, caso duas chaves caiam no mesmo índice, possam co-existir em uma lista, tornando possível acessar elas.

3. Casos de Teste

A fim de testar a implementação e analisar o tempo de complexidade, se deu a criação de dois dicionários, um com 1000 *buckets*, e outro com 100 *buckets*. Nestes, foram inseridos um milhão de pares gerados automaticamente. O fator de carga de cada *bucket* pode ser visualizado na tabela 1. O dicionário 1 se refere ao com 1000 *buckets*, e o dicionário 2 ao com 100 *buckets*. Além do fator de carga, se deram testes para busca de elementos, tanto quais estavam no dicionário, e que não estavam. Os resultados desses são debatidos na seção 4.

Casos de Teste	Fator de carga
Dicionário 1	0.390625
Dicionário 2	0.6103515625

Tabela 1: Fator de carga em cada caso de teste.

Fonte: Autoria Própria.

4. Conclusão

Como pode ser observado na tabela 1, o dicionário 1 possui o menor fator de carga entre os dois casos de teste. Isso se dá devido ao primeiro teste ter um tamanho de *buckets* maior no começo, visto que sempre que atinge um fator de carga de 0.75 ele dobra de tamanho, o tamanho final do bucket é bem maior que o observado no dicionário 2, pois as multiplicações são maiores no primeiro.

Em relação ao tempo de busca, ambos possuem um tempo constante, buscando valores de maneira extremamente rápida. Contudo, a real diferença está na quantidade de memória utilizada. O dicionário 1 usa muito mais memória devido a ter alocado um espaço maior de *buckets*.

Como pode ser observado nessa tarefa, dicionários são uma forma muito eficiente de armazenar valores para serem recuperados através de uma chave. Isso se deve devido ao hashing e compressão, buscando valores em uma média de tempo constante *O(1)*. Esse tempo pode ser afetado em um cenário ruim, com todos os hashes sendo mapeados para o mesmo índice, deixando o tempo de busca para *O(n)*, visto que todos os valores estarão armazenados em uma lista encadeada única.