Resolução de Problemas Estruturados em Computação

Mateus M. Batista[1]

[1]Escola Politecnica – Pontifícia Universidade Católica do Paraná (PUCPR)
Rua Imaculada Conceição, 1155 – Bairro Prado Velho – CEP 80215-901

Abstract. *This work corresponds to the computer science course for the discipline of Structured Problem Solving in Computing to professor Andrey Cabral Meira. Corresponding to implementing and analyzing the performance of different hash tables in Java, where we must choose rehashing and chaining.*

Resumo. *Este trabalho corresponde ao curso de informática da disciplina de Solução Estruturada de Problemas em Computação do professor Andrey Cabral Meira. Corresponde à implementação e análise do desempenho de diferentes tabelas hash em Java, onde devemos escolher rehashing e encadeamento.*

1. Informações

O relatório a seguir corresponde a análise de resultados de implementações de tabelas Hash, onde temos as implementações do FNV1, Murmur, multiplicação e resto de divisão

2. Arquivo Main

Método test: Este método é usado para testar o desempenho de uma implementação de tabela hash. Ele recebe os seguintes parâmetros:

h: A tabela hash a ser testada, que é uma instância da classe HashMap. name: Uma string que descreve o nome da função de hash sendo testada. seed: Uma semente para inicializar o gerador de números aleatórios. times: O número de operações de inserção a serem realizadas.

O método cria duas listas de objetos Registro chamadas keys e values, onde cada Registro possui um array de dígitos. Em seguida, ele insere os pares chave-valor na tabela hash e mede o tempo decorrido. O método também imprime informações sobre o teste, incluindo o tempo decorrido, o tamanho da tabela e as colisões.

No método main do programa, temos o seguinte: Gera uma semente aleatória para uso nos testes. Define arrays testSizes e testTimes, que contêm os tamanhos da tabela e o número de operações de inserção a serem testados. Em seguida temos a execução do teste dos 4 tipos de Hash escolhidos.

3. Classe Registro

O arquivo Registro representa uma classe que é usada para armazenar registros de dígitos e calcular o valor numérico com base nesses dígitos.

Atributos:

size: Um atributo que armazena o tamanho do registro, ou seja, o número de dígitos que ele contém. digits: Um array de inteiros que armazena os dígitos do registro.

Construtor: O construtor da classe Registro recebe um parâmetro size que especifica o tamanho do registro e inicializa o array digits com o tamanho fornecido. Isso é usado para criar uma instância de registro com a capacidade de armazenar um número específico de dígitos.

Método setDigits: Este método permite definir os dígitos do registro. Ele recebe um array de inteiros digits como parâmetro e copia esses dígitos para o array digits da instância de Registro. Isso permite definir os dígitos que compõem o registro.

Método getValue: Este método é usado para calcular e retornar o valor numérico representado pelos dígitos do registro. Ele percorre os dígitos do registro e, para cada dígito, multiplica o valor do dígito pela potência de 10 correspondente à posição do dígito e acumula esses valores. Isso resulta no valor numérico total representado pelos dígitos no registro.

4. HashMap

O arquivo HashMap representa uma classe abstrata que utiliza como uma estrutura base para diferentes implementações de tabelas hash.

Atributos: size: Armazena o tamanho da tabela hash. usedPos: Rastreia o número de posições utilizadas na tabela hash.

collisions: Rastreia o número de colisões ocorridas durante as operações.

table: É um array de objetos HashMapPair, que é usado para armazenar os pares chave-valor na tabela hash.

Construtor: O construtor da classe HashMap recebe um parâmetro size e inicializa os atributos correspondentes.

Método abstrato hash: Este é um método abstrato que deve ser implementado pelas subclasses. Ele é responsável por calcular o índice na tabela hash com base no valor (chave) fornecido.

Método rehash: Este método implementa uma estratégia simples de rehashing. Ele recebe um valor e calcula o próximo índice com base no tamanho da tabela.

Método put: Este método é usado para inserir um par chave-valor na tabela hash.

Método get: Este método permite buscar um valor na tabela hash com base na chave fornecida. Ele retorna o valor correspondente se encontrado, caso contrário, retorna null.

Método remove: Este método é usado para remover um par chave-valor da tabela hash com base na chave fornecida. Ele retorna 0 em caso de sucesso e -1 se a chave não for encontrada.

Métodos isEmpty e isFull: Esses métodos verificam se a tabela hash está vazia ou cheia, respectivamente.

Métodos getSize e getCollisions: Esses métodos fornecem o tamanho da tabela hash e o número de colisões registradas, respectivamente.

Método privado getPos: Este método é usado internamente para obter a posição de uma chave na tabela hash. Ele itera sobre as posições da tabela até encontrar a chave

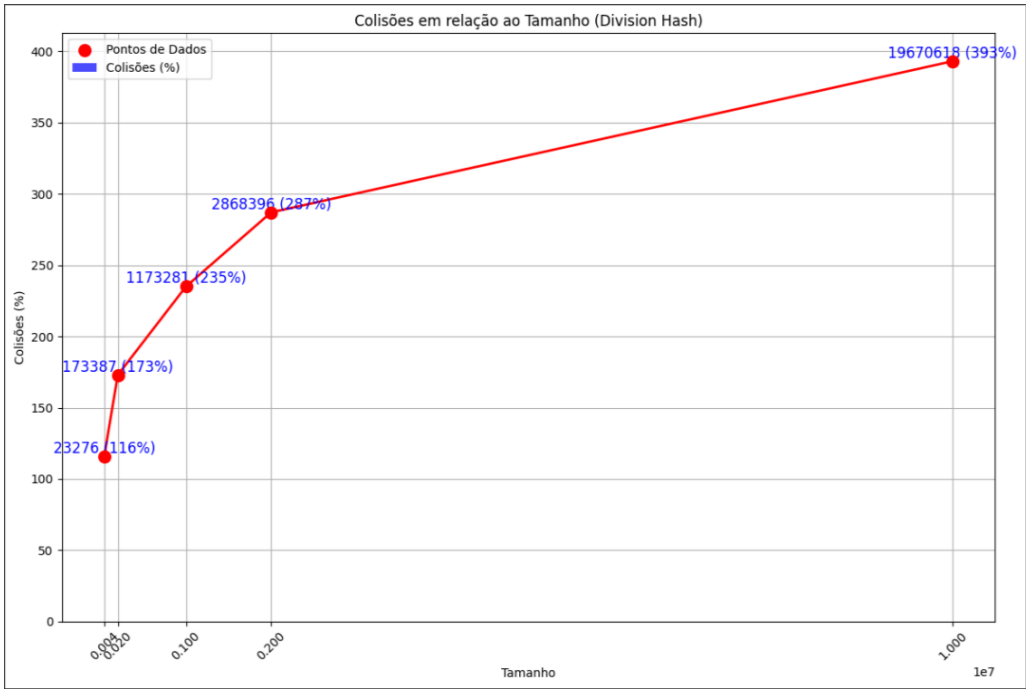

Figure 1. Gráfico resultante operações de Hash de divisão

desejada ou uma posição vazia.

4.1. Hash Divisão

Hash da Divisão:

O ”hash da divisão” é uma técnica de espalhamento muito simples onde o valor de hash é obtido calculando o resto da divisão do valor pela quantidade de slots disponíveis na tabela hash. No caso da implementação apresentada, o valor de hash (value porcentagem getSize()) é calculado usando o operador de módulo porcentagem.

Prós:

É uma técnica muito simples e rápida de se implementar. Funciona bem quando o tamanho da tabela é um número primo.

Contras: Pode resultar em colisões com mais frequência, especialmente se o tamanho da tabela não for um número primo ou se houver padrões nos valores de entrada.

5. Hash Multiplicação

O ”hash de multiplicação” é uma técnica de espalhamento que envolve a multiplicação do valor de entrada por uma constante e, em seguida, a extração da parte fracionária do resultado. O valor final é obtido multiplicando o resultado da multiplicação pela quantidade total de slots na tabela hash e, em seguida, usando a parte inteira do resultado como o valor de hash.

Prós: É uma técnica razoavelmente simples e eficaz para evitar colisões, especialmente quando a constante de multiplicação é bem escolhida. Funciona bem com tamanhos de tabela que são potências de 2.

Multiplicação				
Operações	Tamanhos	Tempo	Colisões	Porcentagem Colisões
20000	40000	16	10424	52%
100000	200000	79	49161	50%
500000	1000000	253	250645	50%
1000000	2000000	701	503371	50%
5000000	10000000	4985	2526908	51%

Figure 2. Gráfico resultante operações de Hash de multiplicação

FNV-1 (Fowler-Noll-Vo):				
Operações	Tamanhos	Tempo	Colisões	Porcentagem Colisões
20000	40000	16	10077	50%
100000	200000	54	49781	50%
500000	1000000	448	252204	50%
1000000	2000000	638	501866	50%
5000000	10000000	3158	2542066	51%

Figure 3. Gráfico resultante operações de Hash FNV1

Contras: A escolha inadequada da constante de multiplicação pode resultar em um pior desempenho. Pode ser menos eficaz em tamanhos de tabela que não são potências de 2

6. FNV-1 (Fowler-Noll-Vo):

O FNV-1 é um algoritmo de hash amplamente utilizado que envolve uma série de operações bitwise e multiplicações para calcular o valor de hash de um dado valor de entrada. A constante FNVPRIME é um número primo específico (16777619) e a constante FNVOFFSETBASIS é um valor inicial. A implementação do FNV-1 pode variar ligeiramente com base no tamanho de dados (8 bits, 16 bits, 32 bits, etc.).

Prós: É uma técnica de hash muito eficaz e amplamente utilizada. Oferece uma distribuição uniforme dos valores de hash. Funciona bem com diferentes tamanhos de tabela.

Contras: A implementação pode ser sensível ao tamanho de dados e à escolha das constantes.

7. MurmurHash3

O MurmurHash3 é uma família de algoritmos de hash que são conhecidos por sua alta qualidade e velocidade. Eles são amplamente utilizados para hash não criptográfico em muitos aplicativos. Existem várias variantes do MurmurHash.

Prós: É um algoritmo de hash rápido e eficaz. Oferece uma distribuição uniforme dos valores de hash. Funciona bem com diferentes tamanhos de tabela. Amplamente utilizado em aplicativos que exigem hash não criptográfico.

MurmurHash3				
Operações	Tamanhos	Tempo	Colisões	Porcentagem Colisões
20000	40000	14	9543	48%
100000	200000	32	50810	51%
500000	1000000	236	251137	50%
1000000	2000000	488	501355	50%
5000000	10000000	2964	2551768	51%

Figure 4. Gráfico resultante operações de Hash Murmur3

Contras: Pode não ser adequado para aplicações de criptografia, pois não é resistente a ataques criptográficos.

8. Conclusão

O método aplicado a Murmur Hash3 foi o mais eficiente. de forma que se manteve constante em relação ao aumento significativo de colisões, seu tempo de execução foi mais rápido que aos demais.