Performance em tabelas hash em Java
Análises e comparações

Pedro Lunardelli Antunes[1]

[1]Escola Politécnica – Pontifície Universidade Católica do Paraná (PUC-PR)

LINK DO GITHUB: https://github.com/Pow3rfulR2307/Hashing-em-Java

Resumo. *Este documento tem como objetivo servir como uma análise de diferentes algoritmos de hashing em tabelas(vetores) de tamanho e quantidade de valores variáveis. Cada algoritmo foi testado com tabelas de tamanhos 20000, 50000, 75000, 100000 e 200000, para cada grupo de 20000, 100000, 500000, 1000000 e 5000000 de valores gerados aleatóriamente. Foram utilizados três algoritmos de hashing diferentes, com base no módulo, uso parcial da razão áurea e operação de XOR para a escolha das posições das entradas no vetor.*

1. Introdução

A escolha de um algoritmo de hashing é de extrema importância quando se fala em desempenho. Número de colisões entre valores pode prejudicar muito as operações de busca e inserção de valores. Para este trabalho foi utilizado uma estrutura de encadeamento de dados em ocorrências de colisões, cada entrada aponta para a próxima a ser inserida na mesma posição, abaixo é possível encontrar como foi implementado as operações de inserções, note que o loop percorre a quantidade de valores a ser inserido no vetor, não seu tamanho.

```
for(int i = 0; i < quantValores; i++){

    Registro r = new Registro(valor.nextInt(999999999 - 1 + 1) + 1);
    int posicao = hashFunction(posicao:r.valor, tamanho, option: operacao);

    if(tabela[posicao] == null){

        tabela[posicao] = r;
    }else{

        Registro atual = tabela[posicao];

        while (atual.proximo!=null){

            atual = atual.proximo;
            contador+=1;
        }
        atual.proximo = r;
        colisoes +=1;
        if(contador > maiorContador){

            maiorContador = contador;
            maiorPosicao = posicao;
            //System.out.println("Maior posicao: "+maiorPosicao+" com valores: "+contador);
        }

    }
    contador = 0;
}
```

Figure 1. Inserção

A variável ”contador” mantém controle sobre a quantidade de posições percorridas em uma certa posição, esse valor é usado para descobrir qual a entrada com mais colisões no vetor, dessa forma a busca é realizada com base no último valor da posição com mais colisões para cada combinação. As comparações nesse código são com base na verificação da existência de um próximo valor na sequência, abaixo é possível ver essa busca.

```java
public static void procurar(Registro[]tabela, int quantValores, int tamanho, int maiorPosicao){

    long startSearch;
    long finishSearch;
    int comparacoes = 0;

    startSearch = System.nanoTime();

    //System.out.println("procurar com posicao "+maiorPosicao);

    Registro atual = tabela[maiorPosicao];

    while(atual.proximo!=null){

        atual = atual.proximo;
        comparacoes++;
    }

    finishSearch = System.nanoTime() - startSearch;

    System.out.println("Tempo para procurar o ultimo valor da maior posicao em vetor de tamanho "+tamanho+
            " e quantidade de valores "+quantValores+ ": "+(double)finishSearch/1000000+" ms");

    System.out.println("Comparacoes: "+comparacoes);

}
```

Figure 2. Busca pelo último valor da maior posição

2. Divisão por Módulo

O primeiro algoritmo a ser analisado é o do módulo do valor a ser inserido pelo tamanho do vetor. Essa é a operação mais simples de hashing, garante que as posições definidas respeitem os limites do tamanho do vetor já que o tamanho toma um papel importante na decisão de onde o valor será inserido. O algoritmo foi selecionado pela sua simplicidade, sua performance na inserção de poucos valores é mediana, não tão ruim mas também não é uma das melhores. Abaixo é possível observar as operações realizadas com base nesse algoritmo.

Table 1. Hash módulo do valor pelo tamanho do vetor

	A	B	C	D	E	F
1	TAMANHO	QUANTIDADE	TEMPO HASH MÓDULO	COLISÕES	TEMPO DE PROCURA (ms)	COMPARAÇÕES
2		20000	7.761 ms	7362	0.0012	6
3		100000	21.8502 ms	80141	0.0014	16
4	20000	500000	309.0229 ms	480000	0.0017	46
5		1000000	1132.9126 ms	980000	0.0021	78
6		5000000	15876.6911 ms	4980000	0.0073	312
7		20000	1.3431 ms	3546	2.00E-04	5
8		100000	6.3966 ms	56668	2.00E-04	10
9	50000	500000	184.1089 ms	450001	0.002	24
10		1000000	769.4563 ms	950000	8.00E-04	39
11		5000000	9002.8302 ms	4950000	0.0038	144
12		20000	1.1341 ms	2465	1.00E-04	3
13		100000	6.4623 ms	44835	3.00E-04	7
14	75000	500000	146.2655 ms	425075	0.0017	22
15		1000000	647.5596 ms	925000	0.0021	31
16		5000000	6491.1459 ms	4925000	0.004	101
17		20000	1.182 ms	1861	1.00E-04	3
18		100000	5.9308 ms	36821	3.00E-04	7
19	100000	500000	97.3939 ms	400681	0.0015	16
20		1000000	387.3933 ms	900002	0.0022	25
21		5000000	5960.8174 ms	4900000	0.0024	85
22		20000	1.3576 ms	960	2.00E-04	3
23		100000	10.0388 ms	21282	2.00E-04	6
24	200000	500000	66.847 ms	316564	7.00E-04	11
25		1000000	321.4863 ms	801394	6.00E-04	16
26		5000000	3842.5258 ms	4800000	0.002	48
27						

Como é possível perceber seus tempos de inserção são bem medíocres, em alguns testes como na inserção de 20000 valores em um vetor de tamanho 20000 teve o pior resultado. Entretanto, quando analisamos algumas inserções com mais valores percebemos que o tempo de inserção de 5000000 em um vetor de tamanho 20000 é quase 3000 ms mais rápido que o segundo algoritmo mais rápido para essa combinação, a inserção de 5000000 de valores no vetor de 50000 foi cerca de 600 ms mais rápida que à do algoritmo com uso de áurea. Concluí-se que sua implantação, apesar de bem simples, não possui a melhor performance para conjuntos de dados menores, mas é uma boa escolha para grandes conjuntos. Suas colisões estão na média, melhores que o algoritmo de XOR, mas piores do que com o uso da áurea.

3. Uso de arredondamento e Golden Ratio(áurea)

Esse algoritmo funciona da seguinte forma, primeiro é multiplicado o valor a ser inserido pelo valor áurea decrementado 1 (0.618033988749895), depois é extraído a parte fracionária desse valor por meio da subtração da parte inteira, arrendondada para baixo por meio da conversão para ”int”. Então o tamanho do vetor é multiplicado por essa parte fracionária e arrendondado novamente para baixo, converte-se o valor para ”int” e temos a posição no vetor. Apesar de sua complicada implementação, seus resultados se destacam quando o assunto é número de colisões, já que possui menos colisões que o os outros dois algoritmos, o que para uma questão de organização de uma tabela é um fator de extrema importância. Mas como já é possível perceber pela sua complexa implementação, seus tempos de inserção são consideravelmente mais lentos do que o uso do módulo para grandes conjuntos de dados, em uma inserção de 5000000 de valores em um vetor de

tamanho 50000 temos um tempo de aproximadamente 640 ms maior, ocorre algo similiar na inserção de 5000000 de valores em um vetor de tamanho 20000 onde há uma disparidade enorme de aproximadamente 3000 ms com relação ao uso do módulo. Abaixo é possível visualizar esses resultados.

Table 2. Hash com base no produto e arredondamento com áurea

	A	B	C	D	E	F
1	TAMANHO	QUANTIDADE	TEMPO HASH AUREA	COLISÕES	TEMPO DE PROCURA (ms)	COMPARAÇÕES
2		20000	6.8599 ms	7336	6.00E-04	6
3		100000	29.4476 ms	80114	0.0014	18
4	20000	500000	388.1335 ms	480000	0.0037	47
5		1000000	1185.5657 ms	980000	0.002	84
6		5000000	18023.4704 ms	4980000	0.0179	312
7		20000	0.9593 ms	3527	2.00E-04	5
8		100000	5.9985 ms	56754	4.00E-04	9
9	50000	500000	167.0083 ms	450001	0.0017	24
10		1000000	563.1588 ms	950000	0.0011	39
11		5000000	9642.3096 ms	4950000	0.0044	148
12		20000	1.0184 ms	2424	3.00E-04	3
13		100000	6.1736 ms	44573	3.00E-04	8
14	75000	500000	192.0334 ms	425103	0.0024	21
15		1000000	440.1903 ms	925000	0.002	29
16		5000000	7337.1955 ms	4925000	0.0044	103
17		20000	1.1623 ms	1856	2.00E-04	3
18		100000	6.0217 ms	36977	5.00E-04	6
19	100000	500000	100.7856 ms	400643	0.0015	15
20		1000000	447.4609 ms	900009	0.0021	24
21		5000000	5637.7008 ms	4900000	0.0022	81
22		20000	0.9155 ms	982	1.00E-04	3
23		100000	8.1086 ms	21588	8.00E-04	5
24	200000	500000	73.903 ms	316512	0.0014	12
25		1000000	262.4291 ms	801382	0.002	19
26		5000000	3737.9447 ms	4800000	0.0035	50
27						

4. Hash com XOR

Esse algoritmo funciona com base na operação XOR muito usada em criptografia. Aqui sua implementação foi bem simples, movemos os bits do valor de entrada para a direita 16 posições, criando uma variação do valor que queremos inserir, então é realizada a operação XOR entre o valor a ser inserido e sua versão modificada. Após isso é calculado o módulo desse resultado do XOR pelo tamanho do vetor. É possível notar que essa operação é mais complexa do que o uso simples do módulo, é de esperar que os resultados sejam menos eficientes. Quando consideramos quantidades maiores que 100000 isso acaba se tornando verdade, entretando para quantidades menores a operação XOR é extremamente mais eficiente em alguns casos. Na inserção de 20000 valores em um vetor de tamanho 20000, XOR levou cerca de 1 ms enquanto a operação de módulo simples levou cerca de 7 ms, na inserção de 100000 valores no vetor de 20000 enquanto os outros algoritmos levaram cerca de 21 ms e 29 ms, XOR levou apenas 8 ms. Entretanto para grandes valores os resultados não são bons, abaixo estão os resultados.

Table 3. Hash com uso de XOR

	A	B	C	D	E	F
1	TAMANHO	QUANTIDADE	TEMPO HASH XOR	COLISÕES	TEMPO DE PROCURA (ms)	COMPARAÇÕES
2		20000	1.0406 ms	7377	2.00E-04	6
3		100000	8.7473 ms	80127	5.00E-04	15
4	20000	500000	418.702 ms	480000	0.0041	45
5		1000000	2054.5174 ms	980000	0.0018	80
6		5000000	29116.3893 ms	4980000	0.0027	312
7		20000	1.015 ms	3415	1.00E-04	4
8		100000	6.1776 ms	56775	2.00E-04	10
9	50000	500000	202.5712 ms	450001	0.0024	26
10		1000000	758.946 ms	950000	0.0022	39
11		5000000	8917.6784 ms	4950000	0.0035	151
12		20000	1.0612 ms	2471	1.00E-04	3
13		100000	5.8607 ms	44860	2.00E-04	7
14	75000	500000	144.5869 ms	425093	0.0019	19
15		1000000	415.1185 ms	925000	0.0024	32
16		5000000	6950.5727 ms	4925000	0.0031	102
17		20000	1.3692 ms	1931	2.00E-04	3
18		100000	5.1849 ms	36712	2.00E-04	7
19	100000	500000	106.9537 ms	400678	0.0017	17
20		1000000	411.2541 ms	900007	0.0013	25
21		5000000	5291.7806 ms	4900000	0.0042	83
22		20000	1.6859 ms	986	3.00E-04	3
23		100000	7.4807 ms	21291	5.00E-04	5
24	200000	500000	73.8368 ms	316354	0.0012	13
25		1000000	328.8048 ms	801368	0.0018	17
26		5000000	3969.4754 ms	4800000	0.0025	51
27						

Na inserção de 5000000 de valores em um vetor de 20000 é possível ver esse contraste, enquanto os algoritmos de módulo simples e áurea demoraram cerca de 15876 e 18023 ms respectivamente, o algoritmo de XOR levou cerca de 29116 ms para executar a mesma operação. Ao analisar a quantidade de colisões pode se dizer que é um algoritmo com uma distrbuição de valores não muito boa, já que possui a maior média de colisões entre todos os algoritmos testados. XOR acaba sendo o melhor para lidar com poucos valores, mas quando se trabalha com grandes conjuntos de dados uma operação de módulo pelo tamanho talvez seja mais ideal.