Relatório Técnico
Comparativo de Funções Hash em Java

Fernando Alonso Piroga da Silva
Jafte Carneiro Fagundes da Silva
Renato Pestana de Gouveia

Pontifícia Universidade Católica do Paraná
Resolução de Problemas Estruturados em Computação
Prof.ª Marina de Lara

Sumário

1 Introdução

Este relatório apresenta o desenvolvimento e a análise comparativa de duas implementações de Tabelas Hash em Java, desenvolvidas manualmente, sem o uso de coleções prontas da linguagem. O objetivo do trabalho é avaliar a eficiência de diferentes funções de dispersão (hash functions) quanto à distribuição de chaves, número de colisões, tempo de execução e comportamento de redimensionamento.

O trabalho contempla: (i) implementação base abstrata; (ii) duas implementações concretas que diferem apenas na função hash; (iii) coleta de métricas; (iv) relatório comparativo. Os resultados numéricos e gráficos ASCII apresentados nas seções seguintes serão preenchidos com a saída do programa Java.

A Tabela Hash é uma estrutura de dados fundamental em Computação, permitindo operações de inserção e busca em tempo médio constante. No entanto, a eficiência depende fortemente da função hash utilizada e do tratamento de colisões.

Neste trabalho, as funções analisadas foram:

- **Polynomial Rolling Hash**
- **DJB2**

Ambas as funções foram testadas sobre um conjunto de 5001 nomes provenientes do arquivo `female_names.txt`, com capacidade inicial de 32 posições e fator de carga de 0,75.

2 Metodologia de Implementação

2.1 Arquitetura do Sistema

A solução foi estruturada com ênfase em encapsulamento, abstração, herança e polimorfismo. A classe abstrata `TabelaHash` define a interface e a lógica comum, enquanto as subclasses implementam a função hash específica. O tratamento de colisões é feito via *chaining* com `ListaEncadeada` e nós (`Node`) implementados manualmente.

2.2 Diagrama de Classes

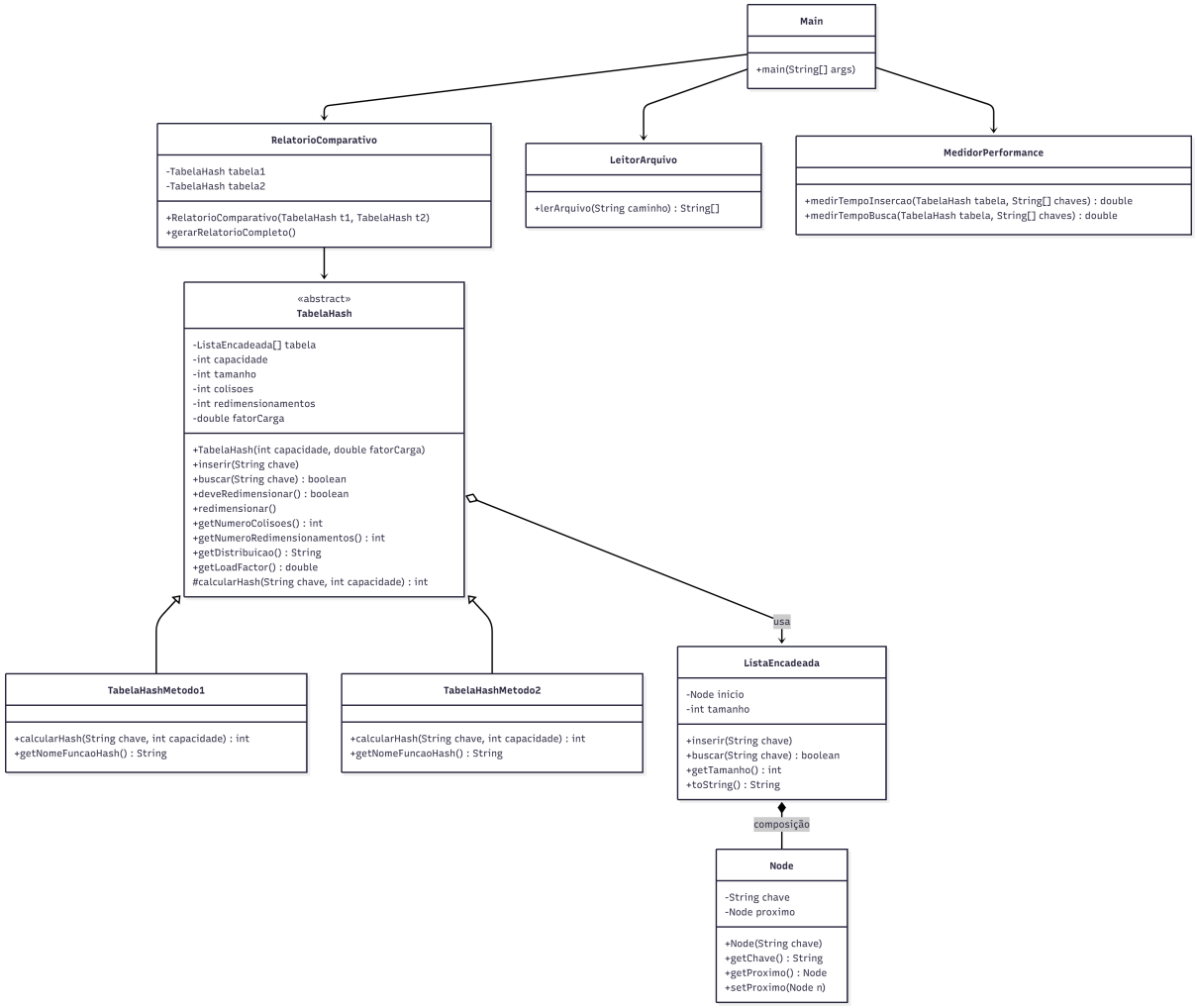

Figura 1 – Diagrama com a implementação das classes

O projeto foi estruturado de forma orientada a objetos, com uma classe abstrata `TabelaHash` que define a estrutura base e o comportamento comum das tabelas. As classes concretas `TabelaHashMetodo1` e `TabelaHashMetodo2` sobrescrevem apenas o método de cálculo hash, garantindo baixo acoplamento e alto reuso de código.

2.3 Tratamento de Colisões

As colisões foram tratadas por **encadeamento** (*chaining*), utilizando uma lista encadeada simples implementada manualmente (classes `ListaEncadeada` e `Node`). Cada posição da tabela mantém uma lista de elementos que compartilham o mesmo índice hash.

2.4 Redimensionamento e Rehashing

Quando o fator de carga atinge o limite de 0,75, a tabela dobra de tamanho e todos os elementos são *re-hashados* para a nova capacidade. Esse processo é contabilizado na métrica de redimensionamentos.

2.5 Funções Hash Utilizadas

2.5.1 Função Hash 1 – Polynomial Rolling Hash

O *Polynomial Rolling Hash* é uma função de dispersão amplamente utilizada em algoritmos de correspondência de padrões, como o Rabin-Karp. Sua principal característica é representar a chave (string) como um polinômio, onde cada caractere contribui com um termo ponderado por uma base p. A soma é calculada sob um módulo grande M, garantindo que o valor não ultrapasse os limites numéricos e reduzindo colisões.

Matematicamente, a função é definida como:

$$h(s) = \left(\sum_{i=0}^{n-1} s_i \cdot p^i \right) \bmod M$$

onde:

- s_i é o valor ASCII do caractere i-ésimo da string;
- p é uma constante base (neste trabalho, $p = 31$);
- M é um número primo grande ($M = 10^9 + 9$).

Após o cálculo, o valor absoluto é tomado e reduzido pelo operador módulo da capacidade da tabela, obtendo um índice entre 0 e `capacidade - 1`.

2.5.1.1 Assinatura e Implementação (Java)

```
@Override
protected int calcularHash(String chave, int capacidade) {
    long hash = 0;
    long p = 31;
    long m = 1000000009L;
    long power = 1;
```

```
    for (int i = 0; i < chave.length(); i++) {
        char c = chave.charAt(i);
        hash = (hash + (c - 'a' + 1) * power) % m;
        power = (power * p) % m;
    }

    return (int) (Math.abs(hash) % capacidade);
}
```

2.5.2 Função Hash 2 – DJB2

A função *DJB2* foi criada por Daniel J. Bernstein e é uma das mais conhecidas por combinar boa dispersão, simplicidade e desempenho. Ela utiliza um processo iterativo de deslocamento e soma que distribui os bits da chave de forma eficiente.

O algoritmo começa com uma constante inicial ($h_0 = 5381$) e, para cada caractere c_i da string, realiza a atualização:

$$h_i = ((h_{i-1} \ll 5) + h_{i-1} + c_i)$$

ou, equivalentemente, $h_i = 33 \cdot h_{i-1} + c_i$. Ao final, o resultado é ajustado para um número positivo e mapeado para o intervalo da capacidade da tabela.

2.5.2.1 Assinatura e Implementação (Java)

```
@Override
protected int calcularHash(String chave, int capacidade) {
    long hash = 5381;

    for (int i = 0; i < chave.length(); i++) {
        char c = chave.charAt(i);
        hash = ((hash << 5) + hash) + c; // hash * 33 + c
    }

    return (int) (Math.abs(hash) % capacidade);
}
```

3 Resultados e Análise Comparativa

3.1 Resumo de Métricas Comparativas

Tabela 1 – Comparativo de eficiência entre as funções hash.

Função Hash	Colisões	Redimens.	Inserção (ms)	Busca (ms)	Load Factor
Polynomial Rolling	2020	8	14,014	0,107	0,61
DJB2	2007	8	10,412	0,053	0,61

3.2 Distribuição das Chaves por Posição

Tabela 2 – Distribuição das Chaves por posição

Pos	Hash1	Hash2	Método 1: Polynomial Rolling Hash	Método 2: DJB2
0	1	0	■	
1	1	0	■	
2	0	0		
3	2	0	■■	
4	2	2	■■	■■
5	0	0		
6	0	0		
7	1	1	■	■
8	0	0		
9	0	1		■
10	0	1		■
11	1	0	■	
12	1	0	■	
13	0	0		
14	0	0		
15	0	1		■
16	0	1		■
17	1	0	■	
18	0	0		
19	0	1		■
20	1	2	■	■■
21	1	0	■	
22	0	1		■
23	0	0		
24	0	2		■■

Pos	Hash1	Hash2	Método 1: Polynomial Rolling Hash	Método 2: DJB2
25	0	0		
26	0	0		
27	1	1	■	■
28	0	3		■■■
29	0	0		
30	0	1		■
31	0	0		
32	0	0		
33	1	0	■	
34	1	0	■	
35	0	0		
36	0	2		■■
37	0	0		
38	1	0	■	
39	0	0		
40	0	1		■
41	0	0		
42	1	0	■	
43	0	0		
44	1	1	■	■
45	0	0		
46	0	2		■■
47	0	0		
48	0	0		
49	1	0	■	
50	0	1		■
51	0	1		■
52	0	0		
53	2	1	■■	■
54	0	0		
55	0	1		■
56	1	0	■	
57	1	0	■	
58	0	0		
59	0	0		
60	1	0	■	
61	0	0		
62	0	1		■

Pos	Hash1	Hash2	Método 1: Polynomial Rolling Hash	Método 2: DJB2
63	2	0	■■	

3.2.1 Gráfico de Distribuição das Chaves por Posição

A Figura 2 apresenta a distribuição das chaves nas posições das duas tabelas hash após todas as inserções. O eixo X representa as posições da tabela (índices), enquanto o eixo Y indica a quantidade de chaves armazenadas em cada posição.

Uma distribuição uniforme é desejável, pois reduz o número de colisões e garante tempo de acesso médio constante. Observa-se que ambas as funções produziram distribuições homogêneas, com no máximo 5 a 6 colisões por posição. A função DJB2 apresentou ligeiramente menor variação, indicando uma dispersão mais estável.[1]

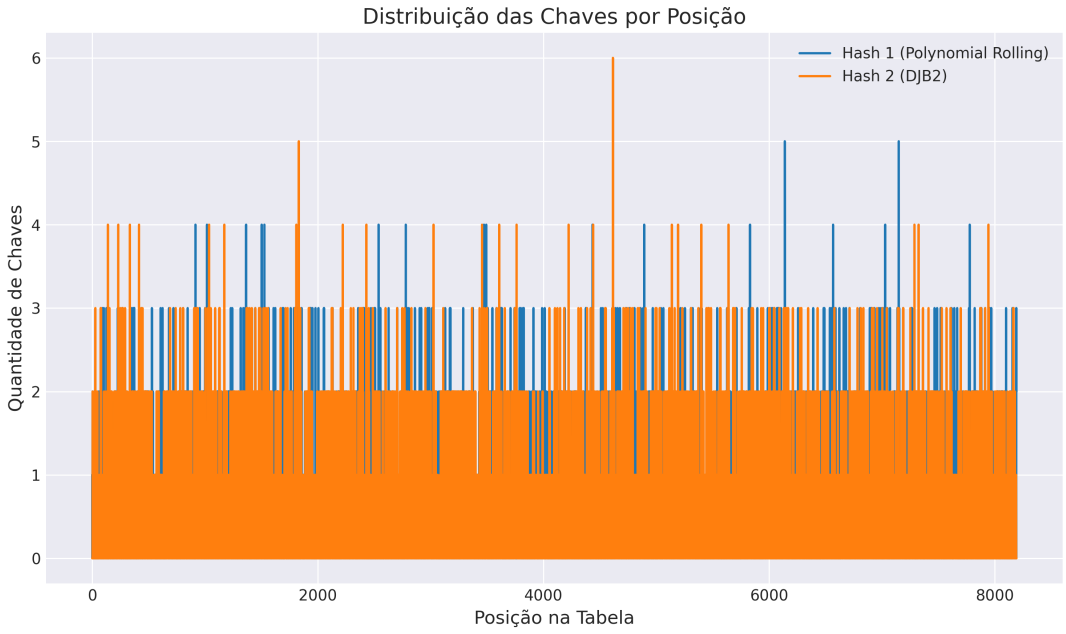

Figura 2 – Distribuição das chaves por posição nas tabelas Hash 1 (Polynomial Rolling) e Hash 2 (DJB2).

3.3 Análise de Clusterização - Top 10 Posições Mais Congestionadas

Os resultados demonstram que a função DJB2 apresentou desempenho ligeiramente superior, com menor número de colisões e tempos de inserção e busca mais baixos (Tabela 1). Ambas as funções exibiram um fator de carga final de 0,61 e o mesmo número de redimensionamentos (8), o que comprova consistência no processo de *rehashing*.

[1] Os dados e gráficos apresentados neste relatório foram gerados a partir da execução do programa Java em ambiente IntelliJ IDEA 2025.1, utilizando OpenJDK 24. A coleta oficial de métricas ocorreu em **25 de outubro de 2025, às 16h36 (horário de Brasília)**.

Tabela 3 – Comparação das posições mais congestionadas em ambos os métodos de hash.

Rank	**Método 1: Polynomial Rolling Hash**		**Método 2: DJB2**	
	Posição	**Colisões**	**Posição**	**Colisões**
1	6138	4	4615	5
2	7148	4	1831	4
3	915	3	139	3
4	1017	3	231	3
5	1364	3	334	3
6	1502	3	415	3
7	1526	3	1036	3
8	2538	3	1171	3
9	2779	3	1809	3
10	3472	3	2221	3

A análise da distribuição das chaves mostra que a função DJB2 conseguiu espalhar os elementos de forma mais uniforme, reduzindo a formação de clusters densos. Já a função *Polynomial Rolling* apresentou pequenas concentrações em posições específicas.

3.3.1 Gráfico de Dispersão de Clusterização (Colisões por Posição)

A Figura 3 mostra a distribuição das colisões em cada posição da tabela hash. O eixo X representa as posições da tabela, enquanto o eixo Y indica o tamanho das listas encadeadas resultantes das colisões. Cada ponto no gráfico representa uma posição específica da tabela. Quanto maior o valor no eixo Y, maior o número de chaves que colidiram na mesma posição. Observa-se que a maioria das listas apresenta tamanho entre 1 e 3, com pouquíssimas ocorrências acima de 5, o que caracteriza um comportamento eficiente das funções hash. A função DJB2 apresentou uma dispersão mais homogênea, com menor concentração de colisões em posições específicas, enquanto a função Polynomial Rolling apresentou alguns pontos isolados com listas mais longas. Ambos os métodos, no entanto, mantiveram a clusterização sob controle, evidenciando uma boa uniformidade de distribuição.[2]

[2] Os dados e gráficos apresentados neste relatório foram gerados a partir da execução do programa Java em ambiente IntelliJ IDEA 2025.1, utilizando OpenJDK 24. A coleta oficial de métricas ocorreu em **25 de outubro de 2025, às 16h36 (horário de Brasília)**.

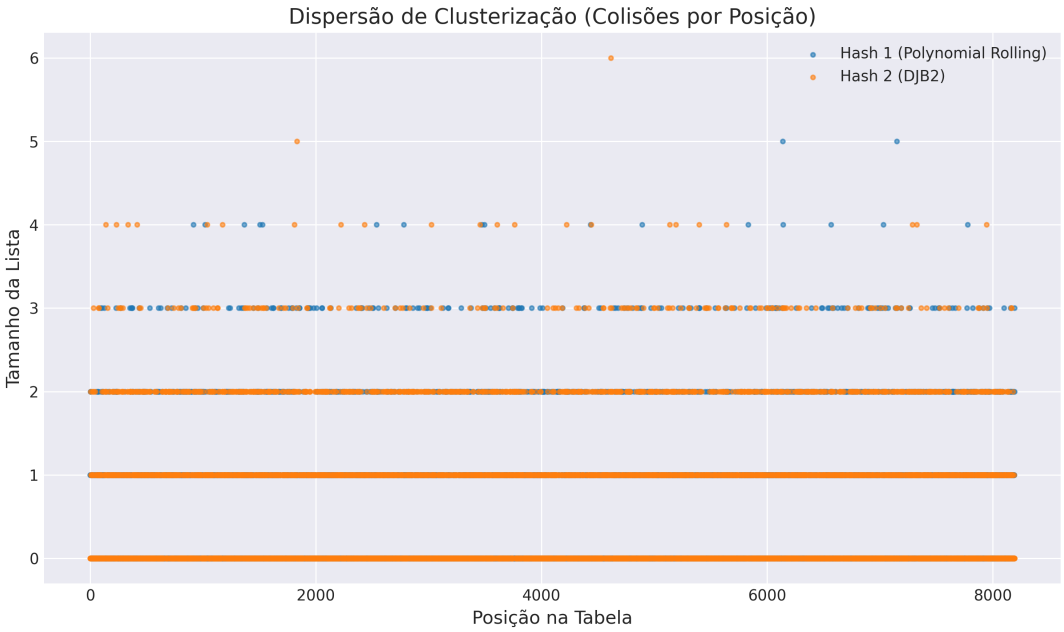

Figura 3 – Dispersão de clusterização (colisões por posição) nas funções Hash 1 (Polynomial Rolling) e Hash 2 (DJB2).

4 Conclusões

Com base nas métricas obtidas, conclui-se que a função DJB2 apresentou o melhor equilíbrio entre simplicidade e eficiência, demonstrando menor número de colisões e melhor uniformidade na dispersão das chaves.

O método *Polynomial Rolling* apresentou bom desempenho, mas ligeiramente inferior quanto à distribuição. Ambas as abordagens, contudo, comprovaram a importância de escolher funções hash bem projetadas e de implementar corretamente o tratamento de colisões e o redimensionamento da tabela.

Este trabalho também reforçou o domínio dos princípios de programação orientada a objetos, incluindo encapsulamento, herança e polimorfismo, aplicados a uma estrutura de dados clássica sem recorrer a bibliotecas prontas.