# **Relatório de Implementação e Análise de Tabelas Hash em Java**

## **1. Introdução**

Este trabalho tem como objetivo implementar e analisar o desempenho de diferentes tabelas hash em Java, utilizando abordagens de rehashing e encadeamento.  
 A proposta visa comparar o desempenho dessas técnicas em relação a tempo de inserção, tempo de busca, número de colisões e distribuição dos elementos, considerando diferentes funções hash e tamanhos de tabelas.

## **3. Configuração**

### **3.1 Tamanhos das Tabelas**

Foram escolhidos três tamanhos diferentes para o tamanho da tabela hash, com uma variação de 10 vezes entre eles:

* **Tamanho 1:** 150.000
* **Tamanho 2:** 1.500.000
* **Tamanho 3:** 15.000.000

**Justificativa da escolha:**

Os três tamanhos de tabela hash foram escolhidos pela conveniência para o teste a fim de que não houvessem colisões excessivas que afetariam o desempenho. Foram testados tamanhos menores mas que ao rodar ocasionava no travamento do programa devido à grande diferença entre o tamanho das tabelas e a quantidade de registros nos conjuntos

**3.2 Funções hash utilizadas**

Foram utilizadas três variações de funções hash, sendo uma para rehashing, e as outras duas utilizadas juntamente com encadeamento e rehashing:

1. **Função 1 (Hash Modular Simples):**  
   * // 1 - Hash modular simples
   * public static int hashModularSimples(int chave**,** int tamanho) {
   * if (chave < 0) chave = -chave**;**
   * return chave % tamanho**;**
   * }
2. **Função 2 (Hash Multiplicativo):**

public static int hashMultiplicativo(int chave**,** int tamanho) {

if (chave < 0) chave = -chave**;**

long chaveLong = chave**;** // para evitar overflow

long produto = (long)(chaveLong \* 6180339887L)**;** // Aprox. Knuth multiplicativo

produto = produto % 10000000000L**;** // mantem parte decimal significativa

return (int)((produto \* tamanho) / 10000000000L)**;**

}

1. **Função 3 (Hash Duplo - para rehashing):**

public static int hashDuplo(int chave**,** int tentativa**,** int tamanho) {

if (chave < 0) chave = -chave**;**

int h1 = chave % tamanho**;**

int h2 = 1 + (chave % (tamanho - 1))**;**

int resultado = h1 + tentativa \* h2**;**

// Garantir índice positivo dentro do tamanho da tabela

resultado = resultado % tamanho**;**

if (resultado < 0) resultado += tamanho**;**

return resultado**;**

}

**Justificativa das funções escolhidas:**

A função de hash modular simples foi escolhida por sua simplicidade, velocidade e fácil implementação, servindo como ponto de referência base para avaliar o desempenho das outras funções. A função de hash multiplicativa foi escolhida por ser matematicamente simples, de baixo custo computacional e conhecida por produzir uma boa dispersão dos dados. Já a função de Duplo Hash foi escolhida por ser comumente utilizada em rehashing e por sua simplicidade de implementação.

### **3.3 Conjuntos de dados**

Foram gerados três conjuntos de dados, utilizando uma seed fixa para garantir a reprodutibilidade dos experimentos. A seed utilizada foi “2025” em long (2025L). Os conjuntos foram:

* **Conjunto 1:** 100.000 registros
* **Conjunto 2:** 1.000.000 registros
* **Conjunto 3:** 10.000.000 registros

Cada registro é um objeto da classe “Registro”, contendo apenas um ID de 9 dígitos.  
 Para isso, os dados foram gerados por meio da utilização da função Random com o número mínimo sendo 100\_000\_000 e o máximo 999\_999\_999, para manter-se dentro do tamanho de 9 dígitos previamente mencionado.

**4. Metodologia**

Cada conjunto de dados foi inserido nas tabelas hash utilizando as duas funções hash principais (Modular simples e Multiplicativa), ocasionalmente sendo utilizada a duplo hash ao haver colisões em tabelas grandes.

Essa escolha foi feita devido ao fato de que ao serem feitos os testes, notou-se que quando tabelas de tamanhos menores eram rodadas com conjuntos muitos grandes, não era possível alocar todos os dados necessários na tabela. Sendo assim, quando isso ocorria, foi utilizado o encadeamento de dados como uma maneira de lidar com essa limitação da tabela e do rehashing.

Durante o processo, foram coletadas as seguintes métricas:

1. Tempo total de inserção em cada tabela x conjunto x função hash;
2. Número total de colisões em cada tabela X conjunto X função hash;
3. Tempo total de busca de todos os elementos em cada tabela X conjunto X função hash;
4. Tamanho das três maiores listas encadeadas geradas em cada tabela X conjunto X função hash;
5. Gap mínimo, máximo e médio entre os elementos das tabelas.

## 

## 

## 

## 

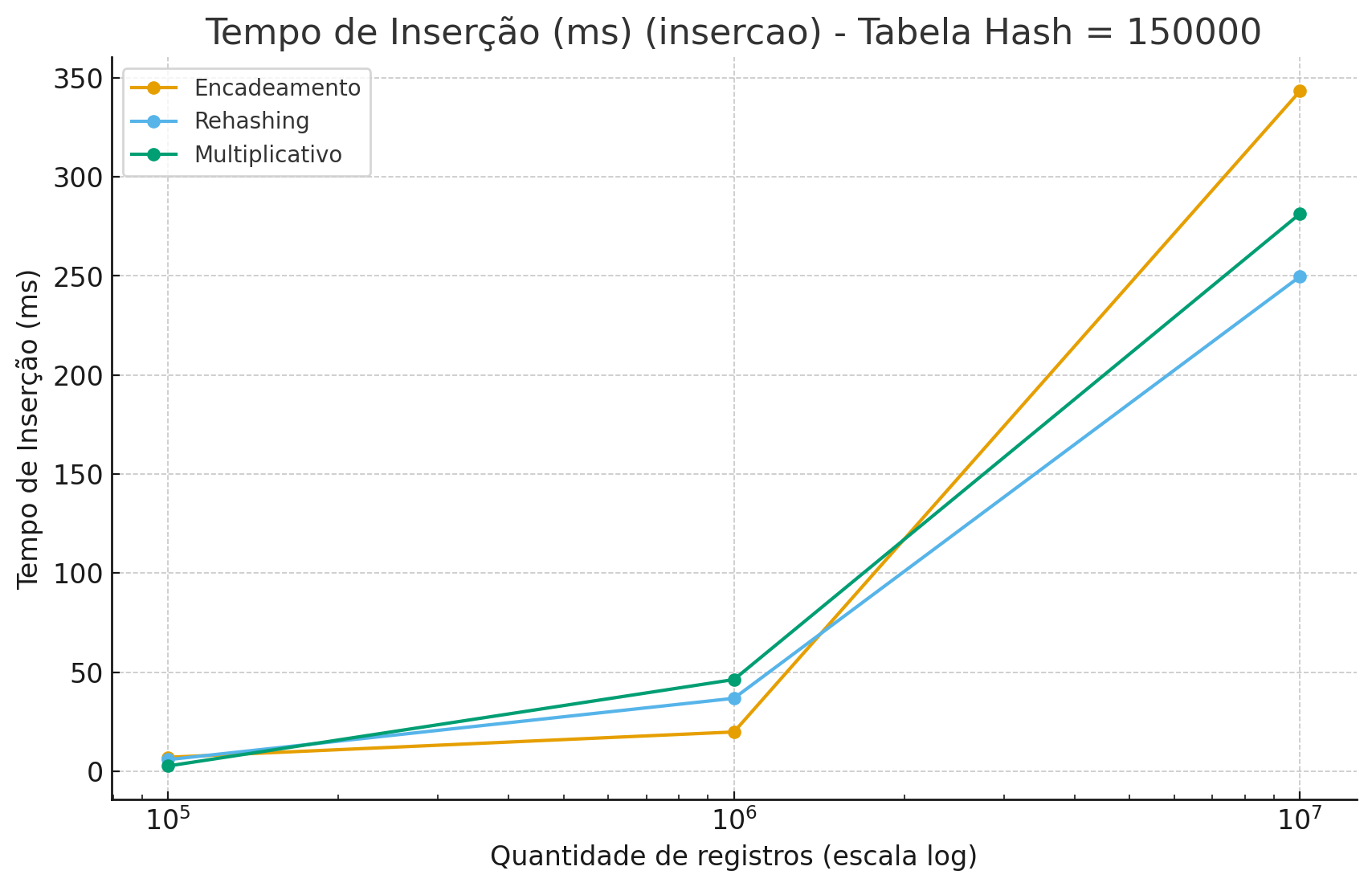
## 

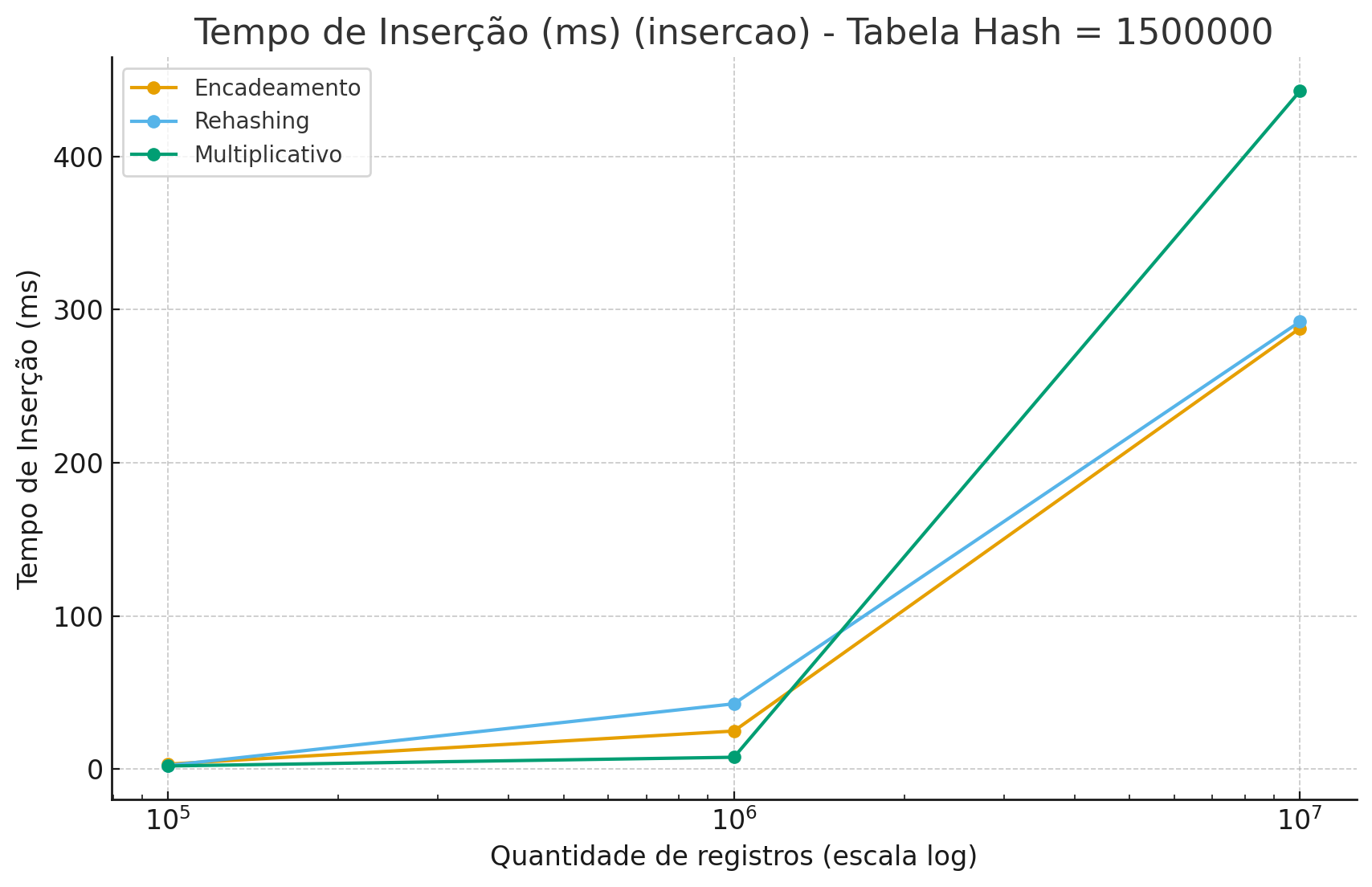
## 

## **5. Resultados**

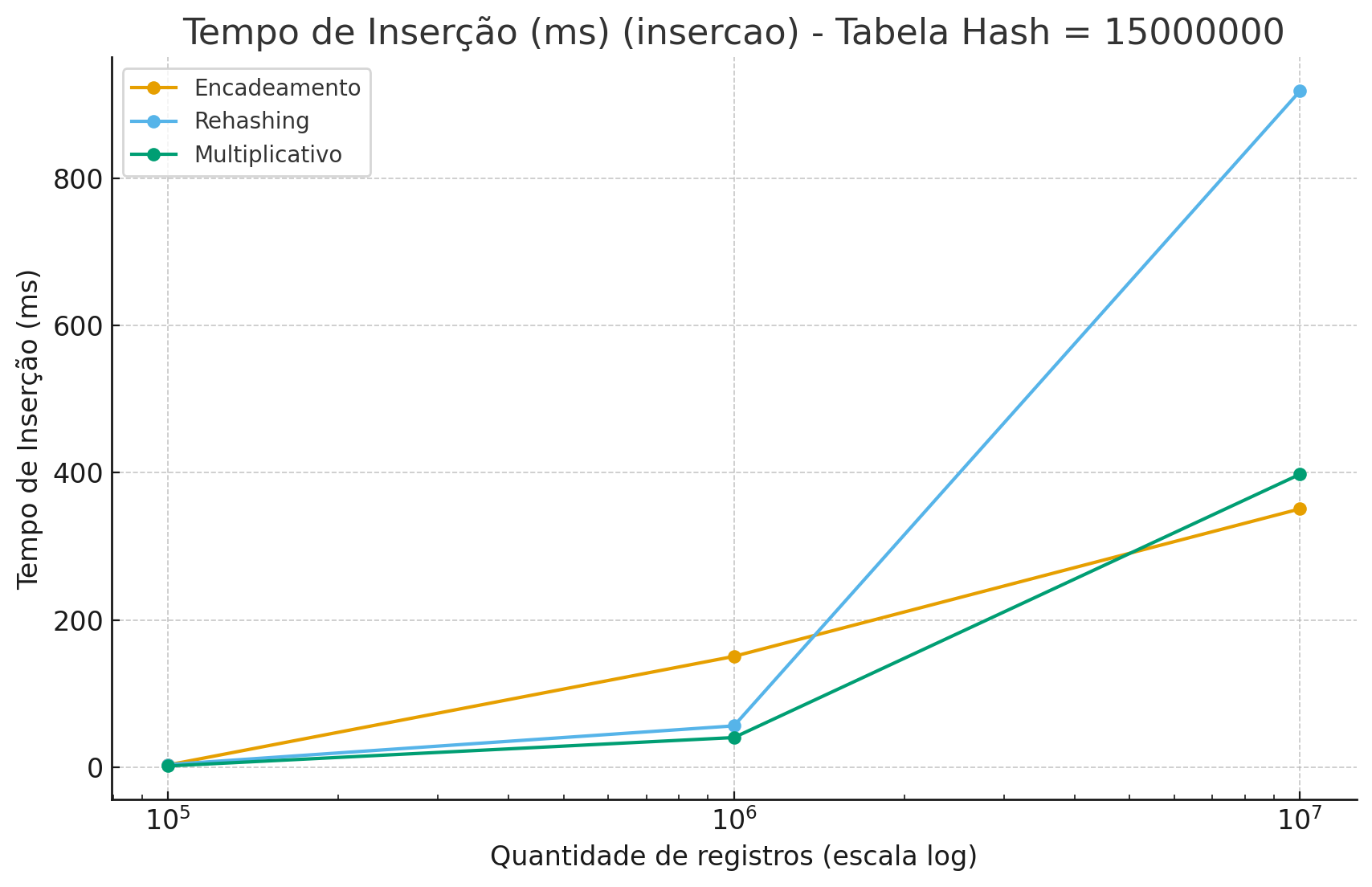
### **5.1 Tempo de Inserção**

| **Função** | **Tamanho Tabela** | **Conjunto 1 (**100.000**)** | **Conjunto 2 (**1.000.000**)** | **Conjunto 3 (**10.000.000**)** |
| --- | --- | --- | --- | --- |
| Hash Modular + Encadeamento | 150.000 | 6.573125 ms | 38.134959 ms | 461.50625 ms |
| Hash Multiplicativo | 150.000 | 8144.06825 m | 28.582917 ms | 216.802667 ms |
| Hash Modular + Hash Duplo | 150.000 | 11.460292 ms | 45.63825 ms | 318.2635 ms |
| Hash Modular + Encadeamento | 1.500.000 | 2.303791 ms | 21.893667 ms | 168.926 ms |
| Hash Multiplicativo | 1.500.000 | 3.382458 ms | 24.157125 ms | 549.204583 ms |
| Hash Modular + Hash Duplo | 1.500.000 | 3.039417 ms | 30.634708 ms | 245.808083 ms |
| Hash Modular + Encadeamento | 15.000.000 | 4.868125 ms | 63.752458 ms | 339.376917 ms |
| Hash Multiplicativo | 15.000.000 | 4.500125 ms | 96.452208 ms | 1883330.734833 ms |
| Hash Modular + Hash Duplo | 15.000.000 | 3.853417 ms | 186.602667 ms | 1271.776875 ms |





### 



### 

### 

### 

### 

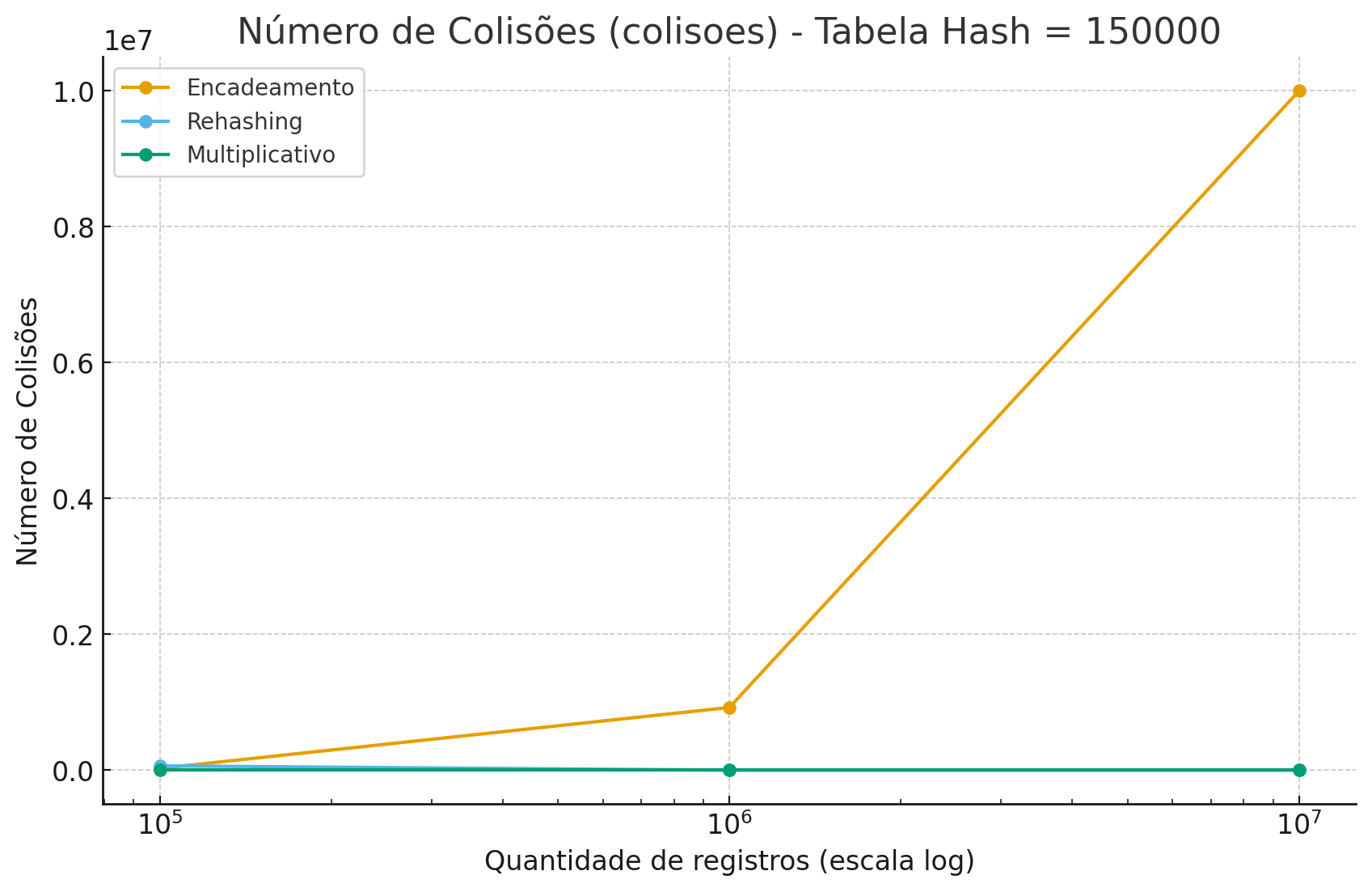
### 

### 

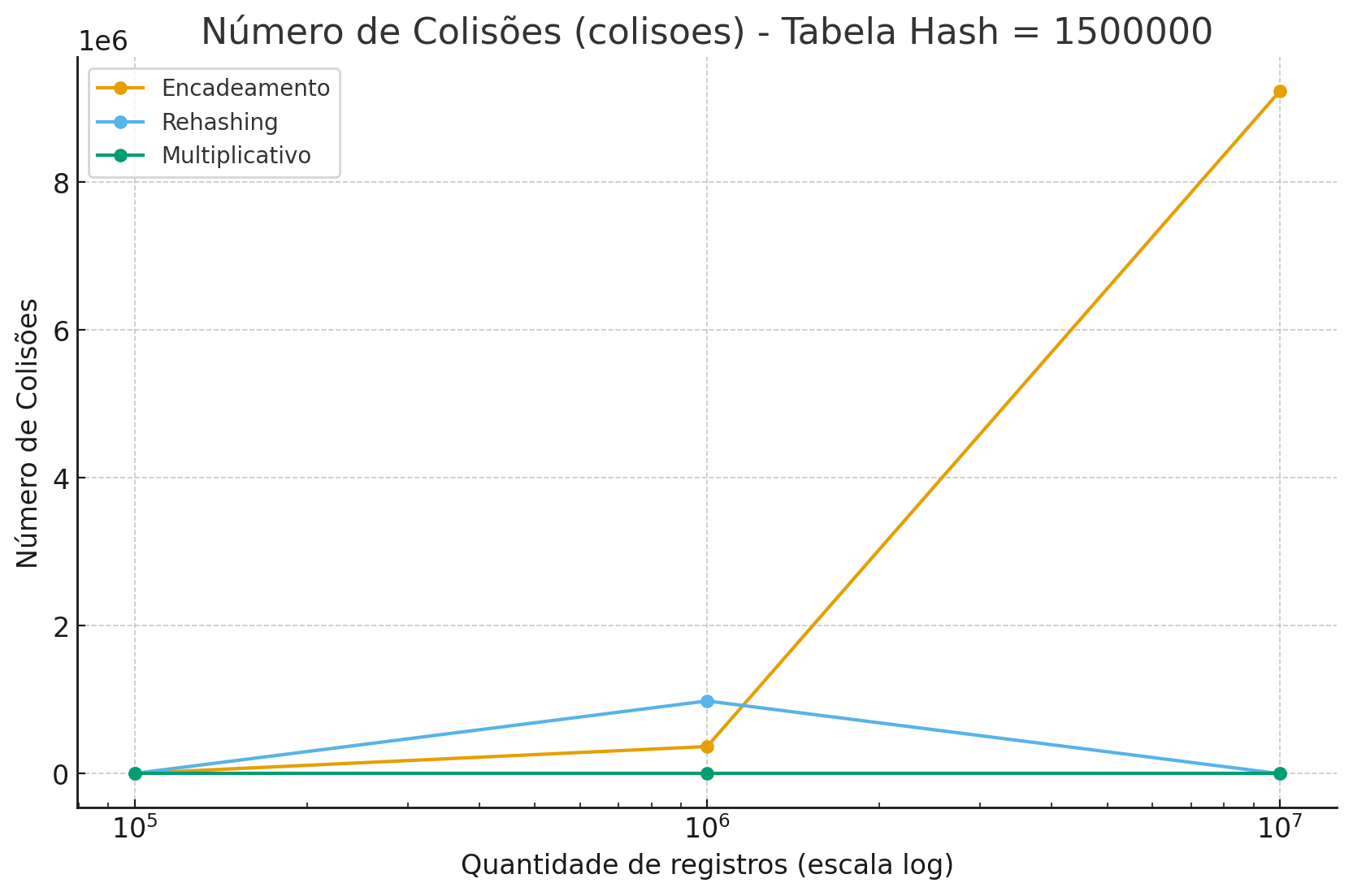
### 

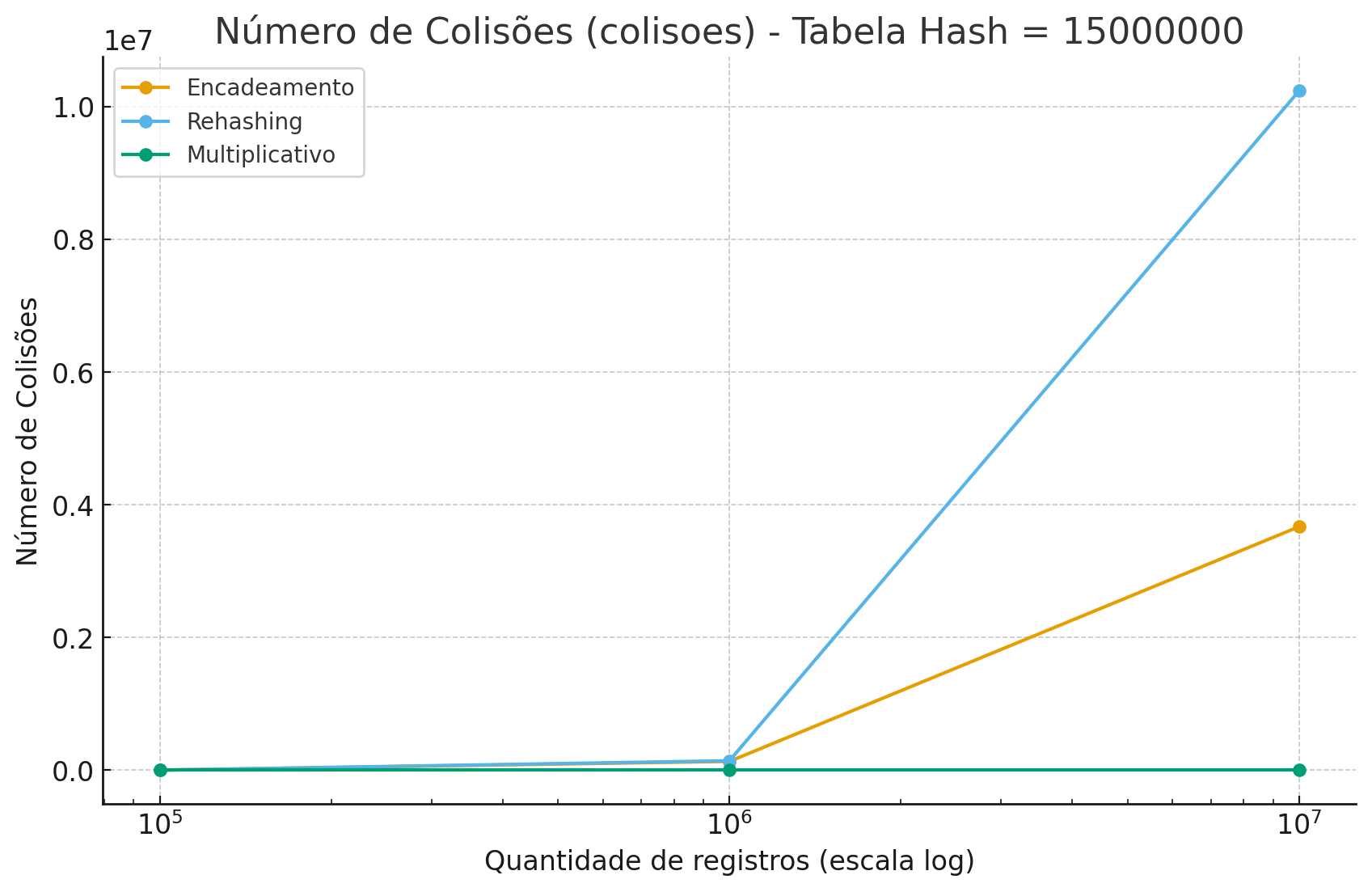
### **5.2 Número de Colisões**

| **Função** | **Tamanho Tabela** | **Conjunto 1 (**100.000**)** | **Conjunto 2 (**1.000.000**)** | **Conjunto 3 (**10.000.000**)** |
| --- | --- | --- | --- | --- |
| Hash Modular + Encadeamento | 150.000 | 26786 | 923406 | 9999808 |
| Hash Multiplicativo | 150.000 | 4042950000 | 0 | 0 |
| Hash Modular + Hash Duplo | 150.000 | 64013 | 0 | 0 |
| Hash Modular + Encadeamento | 1.500.000 | 3198 | 366784 | 9231936 |
| Hash Multiplicativo | 1.500.000 | 4768500000 | 475797000000 | 0 |
| Hash Modular + Hash Duplo | 1.500.000 | 3419 | 983228 | 0 |
| Hash Modular + Encadeamento | 15.000.000 | 343 | 132040 | 3670592 |
| Hash Multiplicativo | 15.000.000 | 5235000000 | 586860000000 | 48257505000000 |
| Hash Modular + Hash Duplo | 15.000.000 | 347 | 142378 | 10240497 |



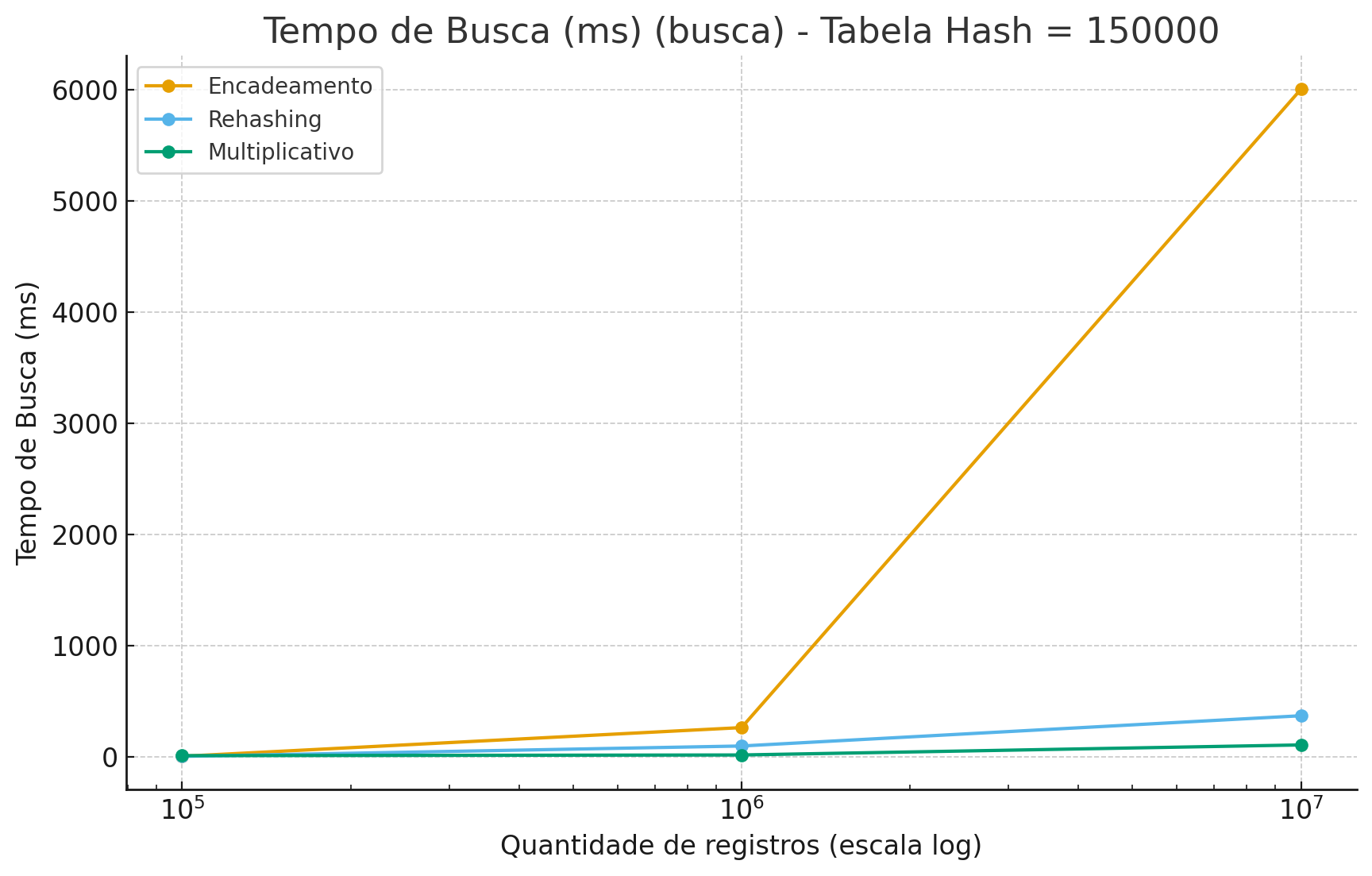
### 



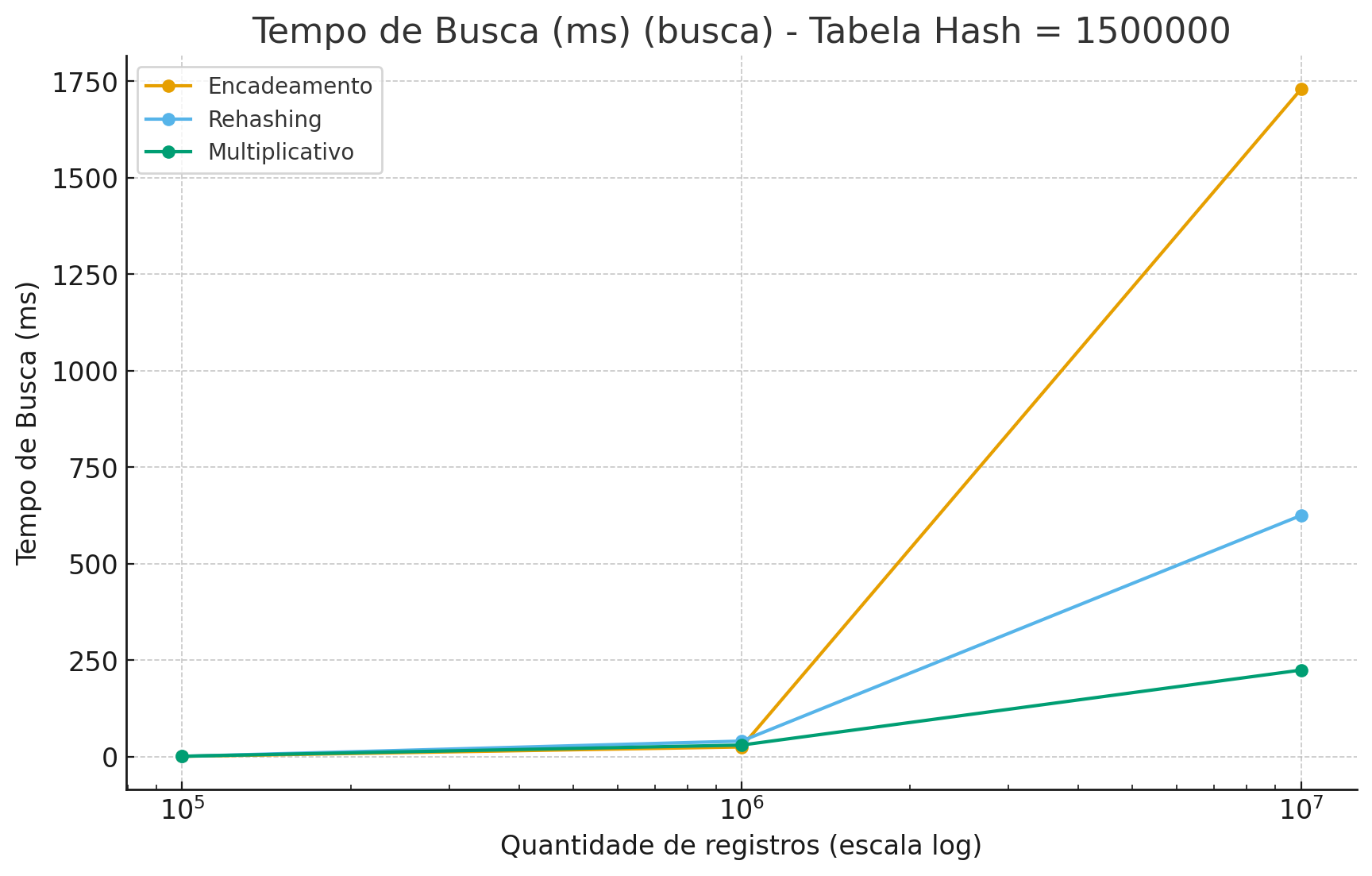
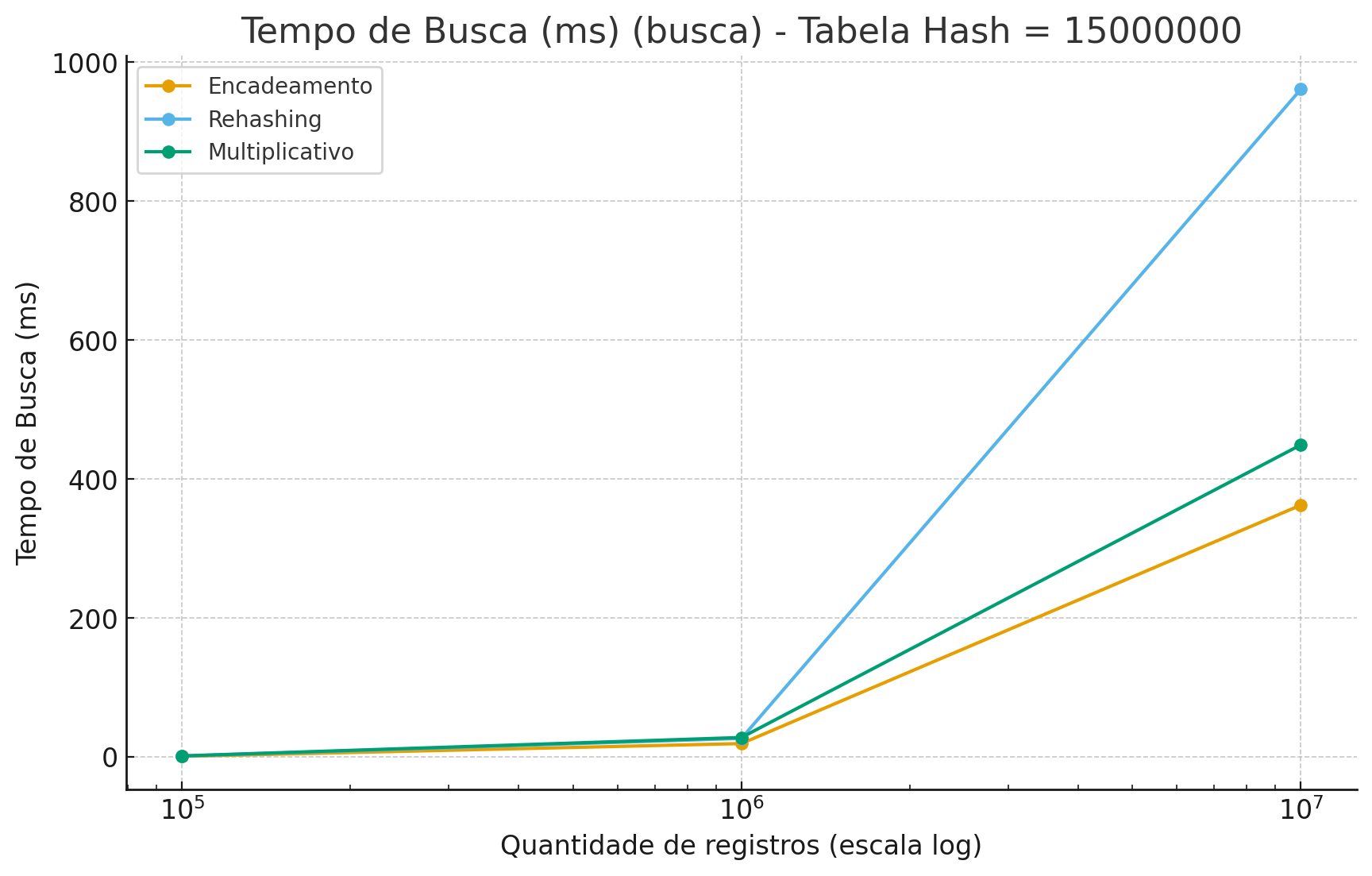


**5.3 Tempo de Busca**

| **Função** | **Tamanho Tabela** | **Conjunto 1 (**100.000**)** | **Conjunto 2 (**1.000.000**)** | **Conjunto 3 (**10.000.000**)** |
| --- | --- | --- | --- | --- |
| Hash Modular + Encadeamento | 150.000 | 5.812625 ms | 90.974542 ms | 13014.001875 ms |
| Hash Multiplicativo | 150.000 | 9.705375 ms | 10.587375 ms | 99.66025 ms |
| Hash Modular + Hash Duplo | 150.000 | 3.235333 ms | 66.382875 ms | 360.946375 ms |
| Hash Modular + Encadeamento | 1.500.000 | 1.001292 ms | 25.36275 ms | 1414.748542 ms |
| Hash Multiplicativo | 1.500.000 | 0.855917 ms | 25.585375 ms | 239.616333 ms |
| Hash Modular + Hash Duplo | 1.500.000 | 0.684583 ms | 46.292166 ms | 652.935084 ms |
| Hash Modular + Encadeamento | 15.000.000 | 1.299666 ms | 29.91925 ms | 420.370334 ms |
| Hash Multiplicativo | 15.000.000 | 1.460584 ms | 28.8215 ms | 423.782375 ms |
| Hash Modular + Hash Duplo | 15.000.000 | 1.740584 ms | 42.173875 ms | 1139.99425 ms |



### 



### 

### 

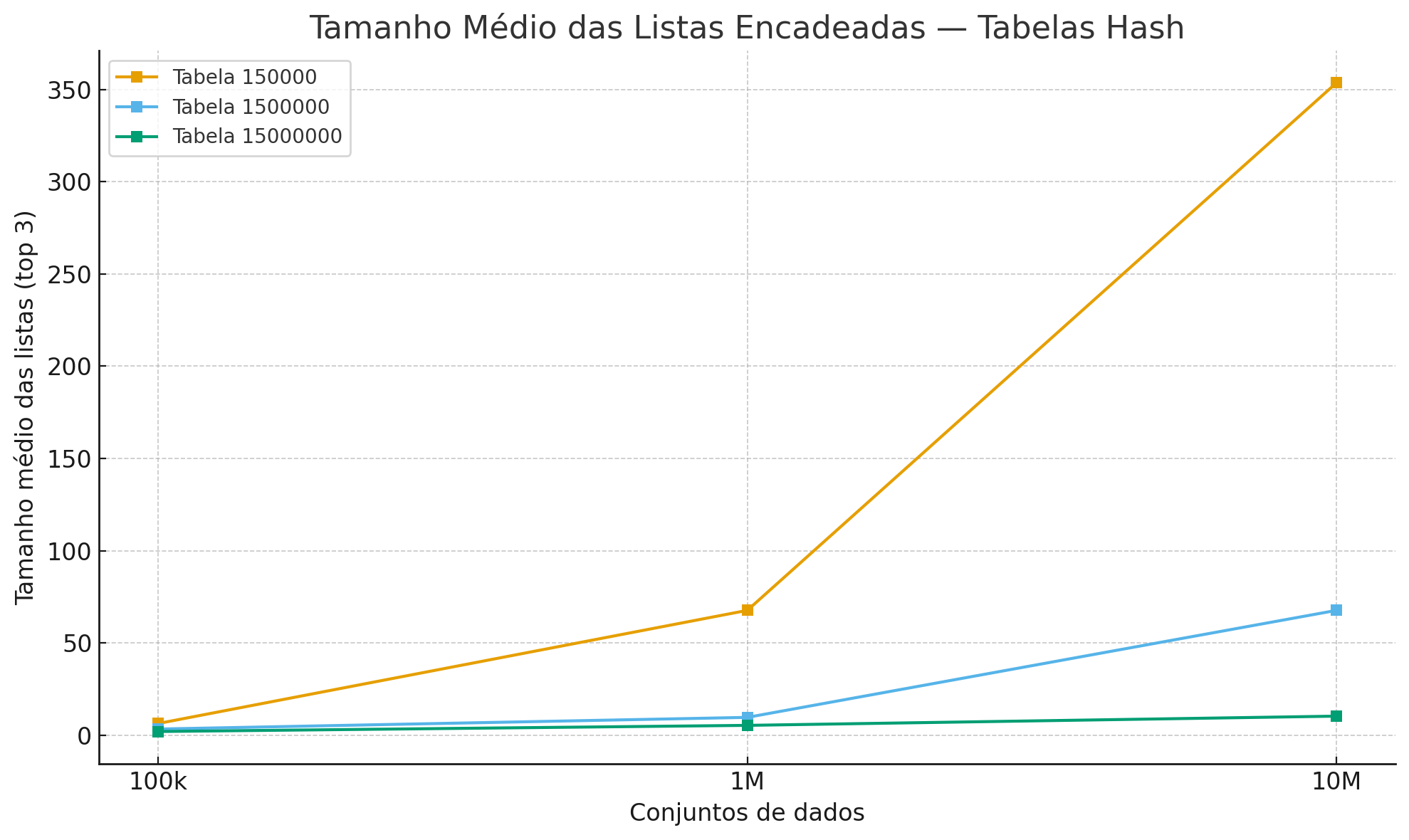
### 

### 

### 

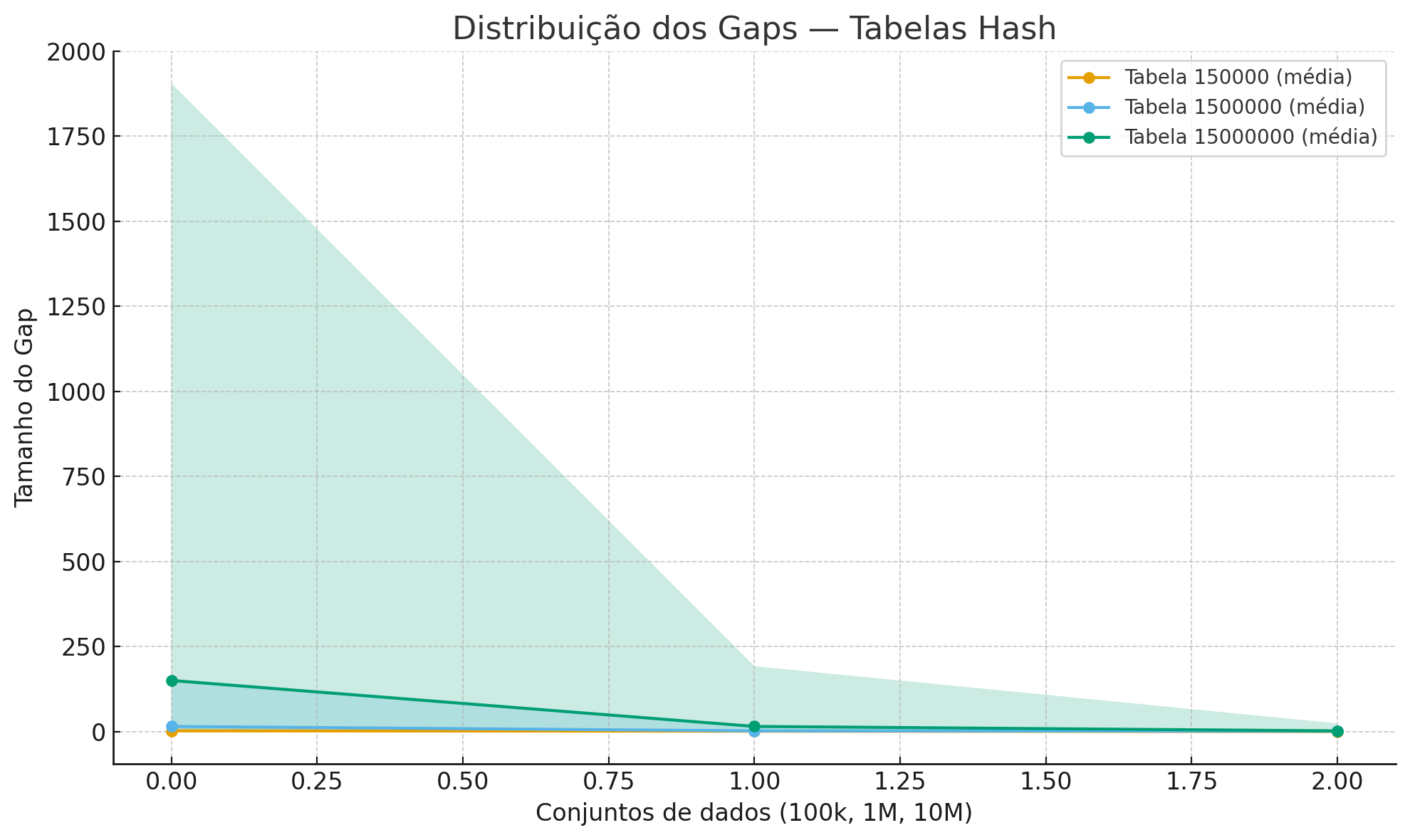
### **5.4 As Três maiores Listas Encadeadas:**

| **Função** | **Tamanho Tabela** | **Conjunto 1 (**100.000**)** | **Conjunto 2 (**1.000.000**)** | **Conjunto 3 (**10.000.000**)** |
| --- | --- | --- | --- | --- |
| Hash Modular + Encadeamento | 150.000 | 7, 6, 6 | 71, 66, 66 | 364, 350, 347 |
| Hash Modular + Encadeamento | 1.500.000 | 4, 3, 3 | 10, 10, 9 | 68, 68, 67 |
| Hash Modular + Encadeamento | 15.000.000 | 2, 2, 2 | 6, 5, 5 | 11, 10, 10 |

****

### **5.5 Gaps:**

| **Tamanho Tabela** | **Conjunto 1 (**100.000**)** | **Conjunto 2 (**1.000.000**)** | **Conjunto 3 (**10.000.000**)** |
| --- | --- | --- | --- |
| 150.000 | Mínimo: 1  Máximo: 18  Médio: 2 | Mínimo: 1  Máximo: 1  Médio: 1 | Mínimo: 0  Máximo: 0  Médio: 0 |
| 1.500.000 | Mínimo: 1  Máximo: 156  Médio: 15 | Mínimo: 1  Máximo: 19  Médio: 2 | Mínimo: 1  Máximo: 1  Médio: 1 |
| 15.000.000 | Mínimo: 1  Máximo: 1905  Médio: 150 | Mínimo: 1  Máximo: 192  Médio: 15 | Mínimo:1  Máximo: 24  Médio: 2 |

****

## 

**6. Conclusão:**

Com base nos resultados obtidos, pode-se perceber que, de modo geral, o Encadeamento apresentou excelente desempenho em cenários de baixa densidade. No entanto, quando o número de registros se aproximou ou superou o tamanho da tabela, essa técnica mostrou grande diminuição de desempenho, com tempos de busca muito altos e listas encadeadas longas.

O Rehashing (Hash Duplo) demonstrou ser o método mais equilibrado e estável, mantendo tempos de busca bons mesmo sob alta carga. Apesar de mais complexo de implementar, o Rehashing apresentou melhor robustez diante do aumento de colisões, especialmente quando o tamanho da tabela era pequeno em relação ao volume de dados.

Outro fator que pode ser observado foi que as tabelas menores tendem à saturação, resultando em gaps iguais a zero e listas longas, enquanto as tabelas maiores apresentaram maior dispersão, porém também com desperdício de espaço. Sendo assim, o melhor desempenho foi alcançado quando o fator de carga era moderado, o que equilibrou ocupação e distribuição.