Sistema de Análise Comparativa de Tabelas Hash

Documentação Técnica do Sistema

Projeto PJBL - Estruturas de Dados

Sumário

1 Introdução **3**
1.1 Objetivos do Sistema . . . 3

2 Funções Hash Implementadas **3**
2.1 Function Hash por Divisão (Division Hash) . . . 3
2.2 Função Hash por Multiplicação (Multiplication Hash) . . . 3
2.3 Função Hash do Meio do Quadrado (Mid-Square Hash) . . . 4
2.4 Critérios de Seleção . . . 4

3 Técnicas de Tratamento de Colisões **4**
3.1 Encadeamento (Chaining) . . . 5
3.2 Rehashing Linear (Linear Probing) . . . 5
3.3 Rehashing Quadrático (Quadratic Probing) . . . 6
3.4 Análise Comparativa das Técnicas . . . 6

4 Arquitetura do Sistema **6**
4.1 Estrutura de Pacotes . . . 7

5 Componentes do Sistema **7**
5.1 Camada de Modelo (Pacote `model`) . . . 7
5.1.1 Classe `Registro` . . . 7
5.2 Camada de Funções Hash (Pacote `hash`) . . . 7
5.2.1 Interface `FuncaoHash` . . . 7
5.2.2 Classe `HashDivisao` . . . 7
5.2.3 Classe `HashMultiplicacao` . . . 7
5.2.4 Classe `HashMeioQuadrado` . . . 7
5.3 Camada de Tabelas Hash (Pacote `hashtable`) . . . 8
5.3.1 Interface `TabelaHash` . . . 8
5.3.2 Classe `No` . . . 8
5.3.3 Classe `TabelaHashEncadeamento` . . . 8
5.3.4 Classe `TabelaHashLinear` . . . 8
5.3.5 Classe `TabelaHashQuadratica` . . . 8
5.4 Camada de Métricas (Pacote `metrics`) . . . 8
5.4.1 Classe `ColetorMetricas` . . . 8
5.5 Camada de Geração de Dados (Pacote `generator`) . . . 8

5.5.1 Classe `GeradorDados` 8
5.6 Coordenador Principal 9
5.6.1 Classe `Main` 9

6 Diagrama de Componentes UML **9**

7 Funcionalidades Principais **10**
7.1 Análise Comparativa 10
7.2 Geração de Datasets 10
7.3 Coleta de Métricas 10
7.4 Geração de Relatórios 10

8 Fluxo de Execução **10**

9 Geração de Datasets Sintéticos **11**
9.1 Tipo de Dados Gerados 11
9.2 Características da Geração 11
9.2.1 Reprodutibilidade 11
9.2.2 Aleatoriedade Controlada 11
9.3 Carregamento e Utilização dos Datasets 11
9.3.1 Processo de Carregamento CSV 11
9.3.2 Estratégia de Reutilização 12
9.3.3 Integração com os Testes 12

10 Resultados - Graficos da execução **12**

11 Gráficos da execução **12**
11.1 Encadeamento (Chaining) 12
11.2 Rehashing Linear 12
11.3 Rehashing Quadratico 12

1 Introdução

O Sistema de Análise Comparativa de Tabelas Hash é uma aplicação desenvolvida em Java que implementa e analisa o desempenho de diferentes técnicas de tratamento de colisões em tabelas hash. O sistema permite uma comparação sistemática entre três funções hash distintas e três métodos de resolução de colisões, fornecendo métricas detalhadas de desempenho para análise acadêmica.

1.1 Objetivos do Sistema

- Implementar três técnicas diferentes de tratamento de colisões em tabelas hash
- Comparar o desempenho de três funções hash distintas
- Gerar datasets sintéticos para testes controlados
- Coletar métricas detalhadas de desempenho
- Produzir relatórios estruturados em formato CSV para análise posterior

2 Funções Hash Implementadas

As funções hash são algoritmos fundamentais que determinam como os elementos são distribuídos nas posições da tabela. A escolha da função hash impacta diretamente na distribuição dos dados e, consequentemente, no número de colisões. Este sistema implementa três funções hash clássicas, cada uma com características distintas:

2.1 Function Hash por Divisão (Division Hash)

A função hash por divisão é uma das mais simples e amplamente utilizadas. Ela utiliza o operador módulo para mapear uma chave k para uma posição na tabela de tamanho m:

$$h(k) = k \bmod m$$

Características:

- Implementação simples e eficiente computacionalmente
- Desempenho altamente dependente da escolha do tamanho da tabela m
- Pode apresentar padrões de distribuição ruins se m não for adequadamente escolhido

2.2 Função Hash por Multiplicação (Multiplication Hash)

A função hash por multiplicação utiliza a aritmética fracionária para criar uma distribuição mais uniforme dos elementos. A fórmula é definida como:

$$h(k) = \lfloor m \times (k \times A \bmod 1) \rfloor$$

onde A é uma constante no intervalo $0 < A < 1$. Knuth sugere o uso de $A = \frac{\sqrt{5}-1}{2} \approx 0.618$ (inverso da razão áurea).

Características:

- Menos sensível ao tamanho da tabela comparada à divisão
- Oferece boa distribuição uniforme independentemente de m
- Ligeiramente mais custosa computacionalmente que a divisão

2.3 Função Hash do Meio do Quadrado (Mid-Square Hash)

A função hash do meio do quadrado eleva a chave ao quadrado e extrai dígitos do meio do resultado. O processo envolve:

1. Elevar a chave k ao quadrado: k^2
2. Extrair os dígitos centrais do resultado
3. Aplicar módulo para ajustar ao tamanho da tabela: middle_digits(k^2) mod m

Características:

- Pode produzir boa distribuição quando os dados têm características apropriadas
- Sensível aos padrões dos dados de entrada
- O número de dígitos extraídos deve ser balanceado com o tamanho da tabela
- Historicamente utilizada, mas menos comum em implementações modernas

2.4 Critérios de Seleção

A escolha dessas três funções hash permite uma análise comparativa abrangente:

- **Simplicidade vs. Sofisticação**: Da divisão simples à multiplicação matematicamente fundamentada
- **Dependência do Tamanho**: Comparação entre funções sensíveis e insensíveis ao tamanho da tabela
- **Distribuição**: Análise de diferentes padrões de distribuição dos dados
- **Desempenho Computacional**: Comparação do custo de cálculo entre as funções

3 Técnicas de Tratamento de Colisões

As colisões ocorrem quando duas ou mais chaves são mapeadas para a mesma posição na tabela hash. O tratamento eficiente dessas colisões é fundamental para manter o desempenho da estrutura de dados. Este sistema implementa três técnicas clássicas de resolução de colisões, cada uma com características e comportamentos distintos:

3.1 Encadeamento (Chaining)

O encadeamento é uma técnica de tratamento de colisões que utiliza estruturas de dados auxiliares para armazenar múltiplos elementos na mesma posição da tabela. Quando uma colisão ocorre, o novo elemento é adicionado a uma lista encadeada associada àquela posição.

Funcionamento:

- Cada posição da tabela contém um ponteiro para o início de uma lista encadeada
- Quando há colisão, o elemento é inserido na lista correspondente
- A busca percorre a lista na posição indicada pela função hash
- A remoção localiza o elemento na lista e o remove

Características:

- **Simplicidade**: Implementação direta e intuitiva
- **Flexibilidade**: Suporta qualquer fator de carga sem degradação catastrófica
- **Memória**: Requer espaço adicional para ponteiros das listas
- **Robustez**: Funciona bem mesmo com distribuições não uniformes

3.2 Rehashing Linear (Linear Probing)

O Rehashing linear é uma técnica de endereçamento aberto onde, em caso de colisão, o sistema procura sequencialmente pela próxima posição disponível na tabela. A busca é feita de forma linear, incrementando a posição uma unidade por vez.

Funcionamento:

- Calcula-se a posição inicial usando a função hash: $h(k)$
- Se a posição estiver ocupada, tenta-se a próxima: $(h(k) + 1) \bmod m$
- Continua-se até encontrar uma posição livre: $(h(k) + i) \bmod m$, onde $i = 0, 1, 2, ...$
- Para busca, segue-se o mesmo padrão até encontrar o elemento ou uma posição vazia

Características:

- **Simplicidade**: Implementação muito simples e eficiente
- **Localidade de Referência**: Boa performance de cache devido ao acesso sequencial
- **Clustering Primário**: Formação de blocos contíguos de elementos ocupados
- **Degradação**: Performance deteriora significativamente com alto fator de carga
- **Memória**: Não requer espaço adicional além da tabela principal

Clustering Primário: O principal problema da sondagem linear é o clustering primário, onde elementos tendem a se aglomerar em blocos contíguos, aumentando o tempo médio de busca e inserção.

3.3 Rehashing Quadrático (Quadratic Probing)

O Rehashing quadrático é uma variação do endereçamento aberto que utiliza incrementos quadráticos para encontrar posições livres. Esta técnica reduz o problema de clustering primário presente na sondagem linear.

Funcionamento:

- Calcula-se a posição inicial: $h(k)$
- Em caso de colisão, tenta-se posições com incrementos quadráticos: $(h(k) + i^2) \bmod m$
- A sequência de tentativas é: $h(k), h(k) + 1^2, h(k) + 2^2, h(k) + 3^2, ...$

Características:

- **Redução de Clustering**: Minimiza o clustering primário comparado à sondagem linear
- **Distribuição**: Melhor espalhamento dos elementos na tabela
- **Clustering Secundário**: Pode apresentar clustering secundário (mesmo padrão de sondagem para chaves com mesmo hash)
- **Complexidade**: Ligeiramente mais complexa que a sondagem linear
- **Requisitos**: Funciona melhor com tamanhos de tabela primos

3.4 Análise Comparativa das Técnicas

A escolha dessas três técnicas permite uma análise das vantagens e desvantagens em estruturas hash:

- **Simplicidade de Implementação**: Linear < Quadrática < Encadeamento
- **Uso de Memória**: Encadeamento > Linear = Quadrática
- **Resistência a Clustering**: Encadeamento > Quadrática > Linear
- **Desempenho com Alto Fator de Carga**: Encadeamento > Quadrática > Linear
- **Localidade de Cache**: Linear > Quadrática > Encadeamento

4 Arquitetura do Sistema

O sistema foi desenvolvido seguindo princípios de orientação a objetos, com separação clara de responsabilidades entre diferentes camadas e componentes. A arquitetura é modular, permitindo fácil extensão e manutenção.

4.1 Estrutura de Pacotes

O sistema está organizado em seis pacotes principais:

- **Pacote Principal**: Contém a classe `Main` que coordena a execução
- **model**: Define as estruturas de dados do domínio
- **hash**: Implementa as diferentes funções hash
- **hashtable**: Implementa as diferentes técnicas de tabelas hash
- **metrics**: Coleta e gerencia métricas de desempenho
- **generator**: Gera datasets sintéticos para testes

5 Componentes do Sistema

5.1 Camada de Modelo (Pacote `model`)

5.1.1 Classe `Registro`

Representa um registro de dados no sistema, contendo um código de 9 dígitos. Esta classe encapsula as informações básicas de um elemento a ser armazenado na tabela hash, garantindo formatação consistente e implementando métodos essenciais como `equals()`, `hashCode()` e `toString()`.

5.2 Camada de Funções Hash (Pacote `hash`)

5.2.1 Interface `FuncaoHash`

Define o contrato para todas as funções hash implementadas no sistema, garantindo uniformidade na interface e permitindo polimorfismo durante a execução dos testes.

5.2.2 Classe `HashDivisao`

Implementa a função hash de divisão, utilizando o operador módulo para distribuir os elementos. Esta é a função hash mais simples e amplamente utilizada, calculada como $h(k) = k \bmod m$.

5.2.3 Classe `HashMultiplicacao`

Implementa a função hash de multiplicação, que utiliza a constante da razão áurea para uma distribuição mais uniforme. A função é calculada como $h(k) = \lfloor m \times (k \times A \bmod 1) \rfloor$, onde $A \approx 0.618$.

5.2.4 Classe `HashMeioQuadrado`

Implementa a função hash do meio do quadrado, que eleva a chave ao quadrado e extrai dígitos do meio do resultado. Esta técnica pode proporcionar boa distribuição quando aplicada adequadamente.

5.3 Camada de Tabelas Hash (Pacote `hashtable`)

5.3.1 Interface `TabelaHash`

Define o contrato para todas as implementações de tabela hash, garantindo métodos uniformes para inserção, busca e análise estrutural.

5.3.2 Classe `No`

Representa um nó na estrutura de dados, utilizado pelas implementações que necessitam de estruturas auxiliares como listas encadeadas.

5.3.3 Classe `TabelaHashEncadeamento`

Implementa o tratamento de colisões por encadeamento, utilizando listas encadeadas para armazenar múltiplos elementos na mesma posição da tabela. Esta técnica é eficiente para altas taxas de ocupação e oferece desempenho previsível.

5.3.4 Classe `TabelaHashLinear`

Implementa o tratamento de colisões por sondagem linear, procurando sequencialmente pela próxima posição disponível. Embora simples, esta técnica pode sofrer com o problema de clustering primário.

5.3.5 Classe `TabelaHashQuadratica`

Implementa o tratamento de colisões por sondagem quadrática, utilizando incrementos quadráticos para encontrar posições livres. Esta abordagem reduz o clustering primário comparado à sondagem linear.

5.4 Camada de Métricas (Pacote `metrics`)

5.4.1 Classe `ColetorMetricas`

Responsável por coletar, processar e armazenar todas as métricas de desempenho do sistema. Coleta dados sobre colisões, tempos de execução, distribuição de elementos e gaps na tabela, fornecendo uma visão abrangente do comportamento das diferentes configurações.

5.5 Camada de Geração de Dados (Pacote `generator`)

5.5.1 Classe `GeradorDados`

Gera datasets sintéticos para testes controlados e reproduzíveis. Permite a criação de conjuntos de dados com características específicas e salva os resultados em formato CSV para uso posterior.

5.6 Coordenador Principal

5.6.1 Classe `Main`

Atua como o ponto de entrada do sistema e coordenador principal da execução. Gerencia o fluxo de execução, coordena os testes entre diferentes configurações e produz os relatórios finais de análise.

6 Diagrama de Componentes UML

O diagrama a seguir ilustra a arquitetura do sistema e as relações entre os diferentes componentes:

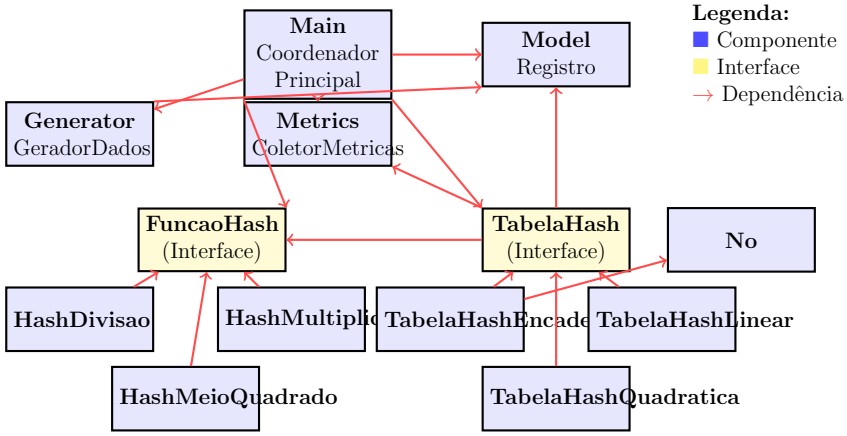

Figura 1: Diagrama de Componentes UML do Sistema de Análise de Tabelas Hash

7 Funcionalidades Principais

7.1 Análise Comparativa

O sistema executa testes sistemáticos comparando:

- Três técnicas de tratamento de colisões
- Três funções hash diferentes
- Três tamanhos de tabela (1.000, 10.000 e 100.000 elementos)

7.2 Geração de Datasets

Capacidade de gerar datasets sintéticos com características controladas, permitindo testes reproduzíveis e análises estatísticas consistentes.

7.3 Coleta de Métricas

Sistema abrangente de coleta de métricas incluindo:

- Número total e médio de colisões
- Tempos de inserção e busca
- Análise das três maiores cadeias (para encadeamento)
- Análise de gaps mínimo, máximo e médio (para sondagem)

7.4 Geração de Relatórios

Produção automática de relatórios estruturados em formato CSV, facilitando análises posteriores e visualizações gráficas.

8 Fluxo de Execução

O sistema segue um fluxo bem definido de execução:

1. **Inicialização**: A classe `Main` processa os parâmetros de entrada e configura o ambiente de execução
2. **Geração/Carregamento de Dataset**: O sistema utiliza `GeradorDados` para criar ou carregar um dataset existente
3. **Configuração de Testes**: Para cada combinação de tamanho de tabela e função hash, o sistema prepara os testes
4. **Execução dos Testes**: Cada técnica de tratamento de colisões é testada com inserção e busca de elementos
5. **Coleta de Métricas**: `ColetorMetricas` registra todas as métricas de desempenho
6. **Geração de Relatórios**: Os resultados são consolidados e salvos em formato CSV

9 Geração de Datasets Sintéticos

Para permitir análises controladas e reproduzíveis do desempenho das tabelas hash, o sistema implementa um gerador de datasets sintéticos que produz conjuntos de dados padronizados para os testes.

9.1 Tipo de Dados Gerados

O sistema gera datasets compostos por objetos do tipo `Registro` contendo códigos numéricos aleatórios. Especificamente:

- **Objetos Registro**: Cada elemento do dataset é um objeto da classe `Registro`
- **Códigos Numéricos**: Números inteiros de 9 dígitos formatados como strings
- **Distribuição Aleatória**: Valores gerados pseudo-aleatoriamente usando um gerador congruencial linear
- **Formato Padronizado**: Todos os códigos são formatados com zeros à esquerda para manter consistência
- **Faixa de Valores**: Números no intervalo de 000000000 a 999999999
- **Encapsulamento**: Cada código é encapsulado em um objeto `Registro` com métodos apropriados

9.2 Características da Geração

9.2.1 Reprodutibilidade

O sistema utiliza seeds determinísticas baseadas no nome do dataset, garantindo que o mesmo conjunto de dados seja gerado consistentemente para análises repetíveis. Isso permite:

9.2.2 Aleatoriedade Controlada

Os números são gerados usando um algoritmo pseudo-aleatório que produz:

- **Distribuição Uniforme**: Probabilidade igual para qualquer valor no intervalo
- **Independência**: Cada valor gerado é independente dos anteriores
- **Período Longo**: Sequência de repetição suficientemente longa para os testes

9.3 Carregamento e Utilização dos Datasets

9.3.1 Processo de Carregamento CSV

O sistema implementa um mecanismo eficiente para carregar datasets previamente salvos em formato CSV:

- **Verificação de Existência**: O sistema primeiro verifica se o arquivo CSV já existe no diretório de dados

- **Leitura Sequencial**: Se o arquivo existir, ele é lido linha por linha usando `BufferedReader`
- **Parsing de Dados**: Cada linha (exceto o cabeçalho) é convertida em um objeto `Registro`
- **Construção do Array**: Os objetos são armazenados em um array para uso nos testes
- **Validação**: O sistema verifica a integridade dos dados carregados

9.3.2 Estratégia de Reutilização

O sistema adota uma estratégia inteligente para otimizar o uso de datasets:

- **Cache de Arquivos**: Se um dataset com o nome especificado já existe, ele é reutilizado
- **Geração sob Demanda**: Caso o arquivo não exista, um novo dataset é gerado automaticamente
- **Persistência**: Novos datasets são salvos em CSV para reutilização futura
- **Consistência**: O tamanho do dataset carregado substitui o parâmetro inicial se diferente

9.3.3 Integração com os Testes

Uma vez carregado, o dataset é utilizado uniformemente em todos os testes:

- **Distribuição Equitativa**: O mesmo conjunto de objetos `Registro` é usado para todas as configurações
- **Ordem Consistente**: A sequência de inserção é mantida constante entre diferentes testes
- **Isolamento de Testes**: Cada configuração de tabela hash recebe uma cópia independente dos dados
- **Métricas Comparáveis**: A uniformidade dos dados garante comparações válidas entre técnicas

10 Resultados - Graficos da execução

O **Task Control Block (TCB)** é a estrutura de dados central de um RTOS, usada para armazenar todas as informações de gerenciamento de uma tarefa.

11 Gráficos da execução

11.1 Encadeamento (Chaining)

11.2 Rehashing Linear

11.3 Rehashing Quadratico

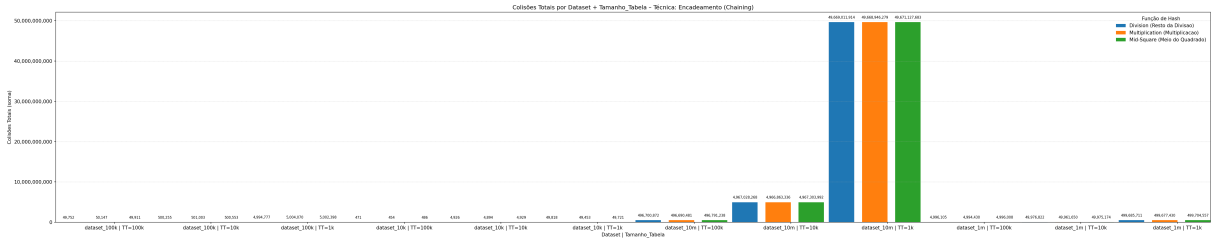

Figura 2: Grafico da quantidade de colisões utilizando Encadeamento

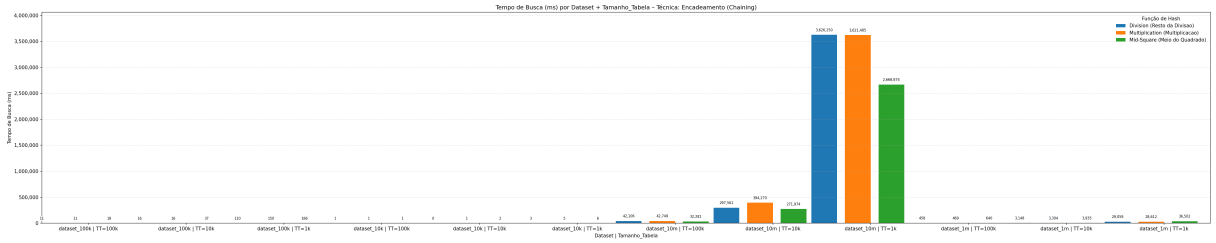

Figura 3: Grafico de tempo de busca (ms) utilizando Encadeamento

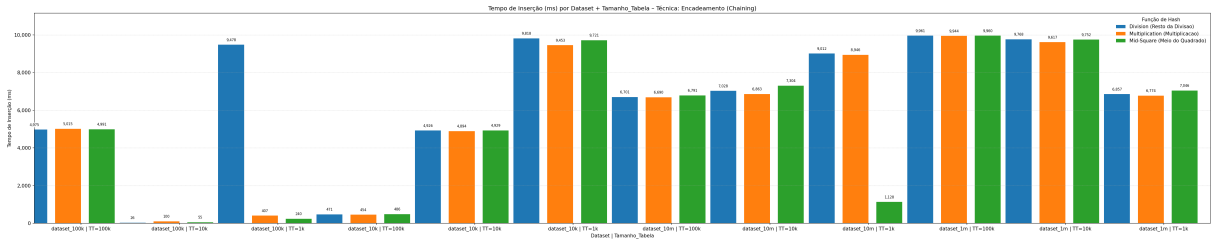

Figura 4: Grafico de tempo de inserções utilizando Encadeamento

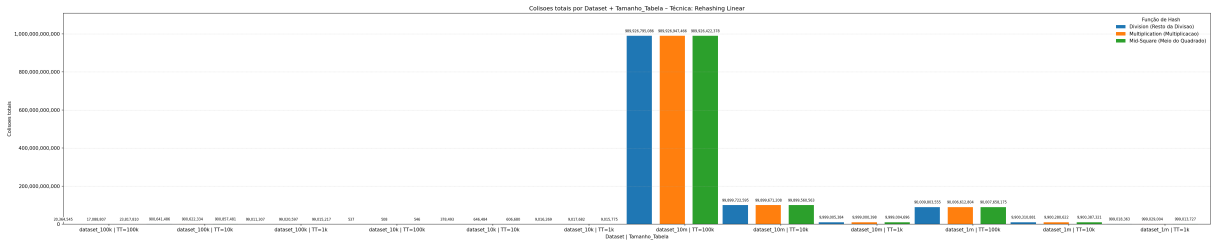

Figura 5: Grafico da quantidade de colisões utilizando Rehashing Linear

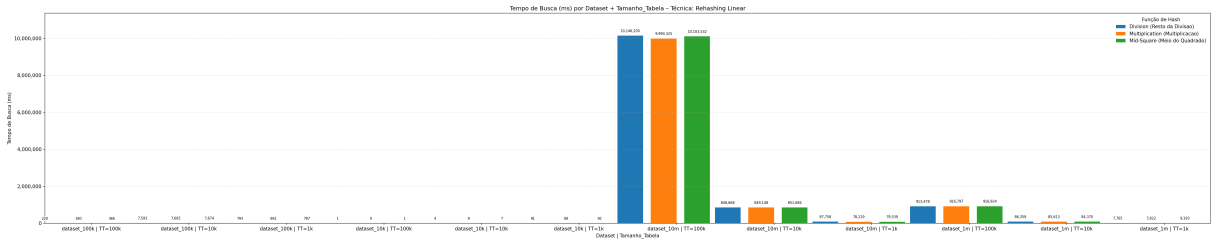

Figura 6: Grafico de tempo de busca (ms) utilizando Rehashing Linear

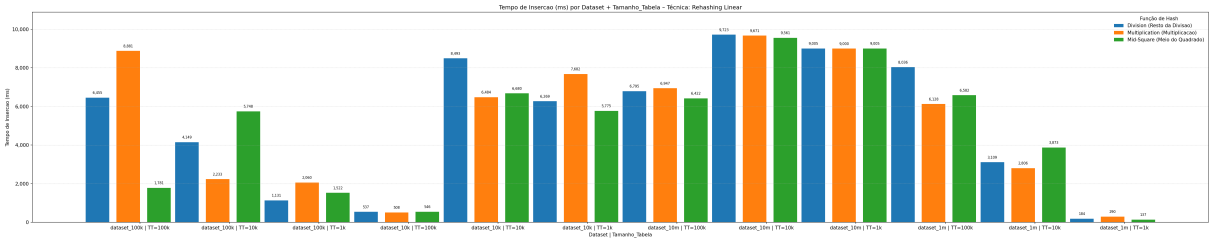

Figura 7: Grafico de tempo de inserções (ms) utilizando Rehashing Linear

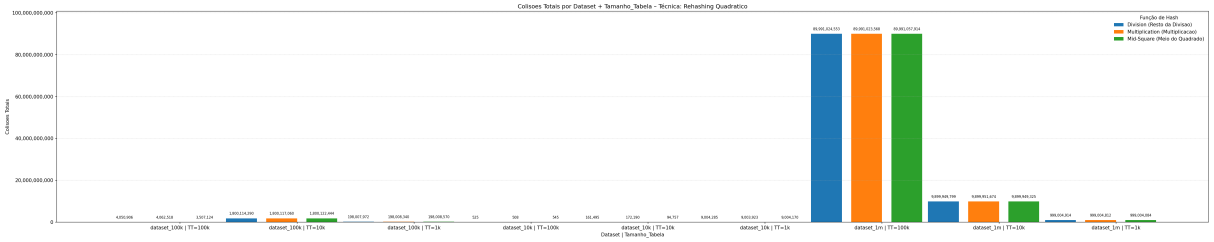

Figura 8: Grafico da quantidade de colisões utilizando Rehashing Quadratico

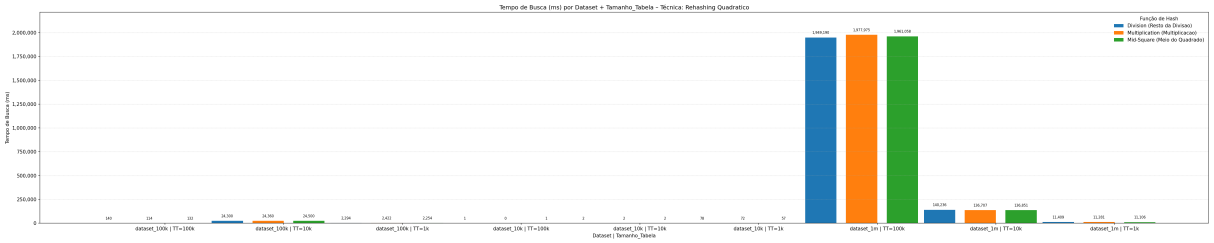

Figura 9: Grafico de tempo de busca (ms) utilizando Rehashing Quadratico

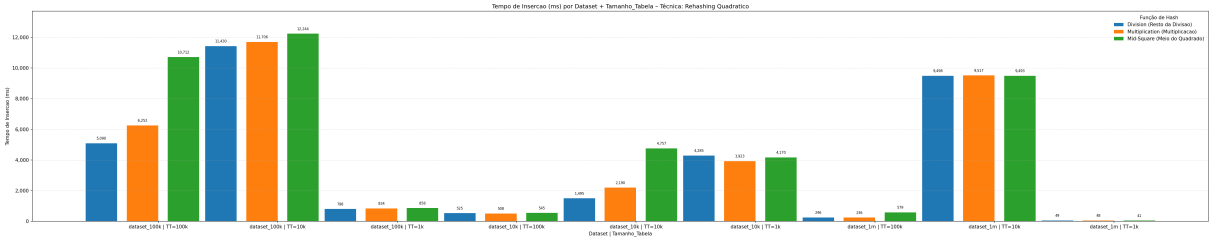

Figura 10: Grafico de tempo de inserções (ms) utilizando Rehashing Quadratico

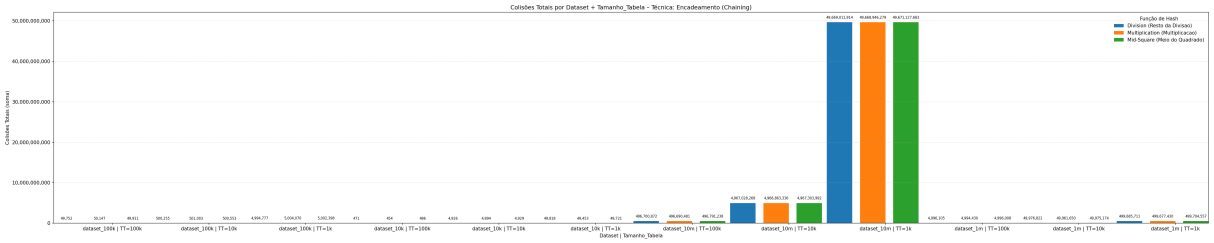

Figura 11: Gráfico da quantidade de colisões utilizando Encadeamento

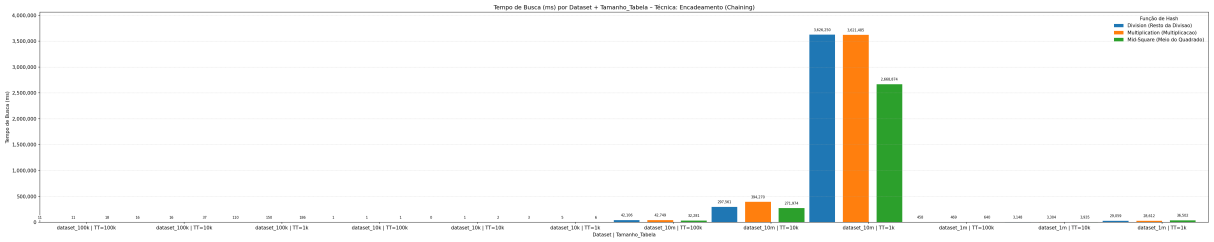

Figura 12: Gráfico de tempo de busca (ms) utilizando Encadeamento

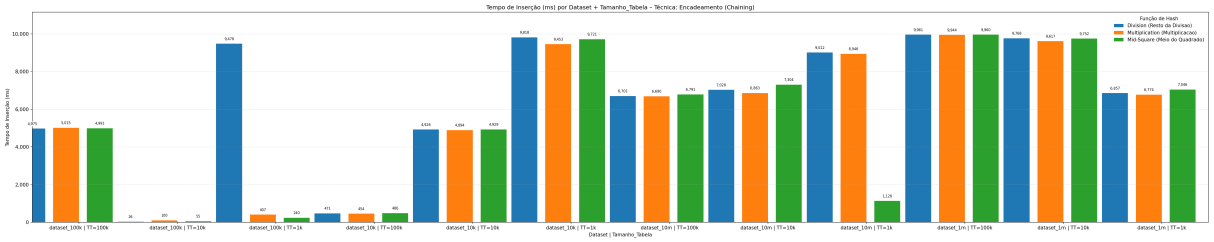

Figura 13: Gráfico de tempo de inserções utilizando Encadeamento

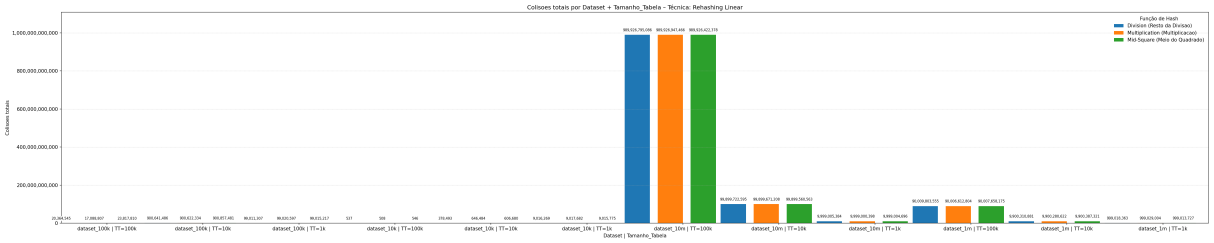

Figura 14: Gráfico da quantidade de colisões utilizando Rehashing Linear

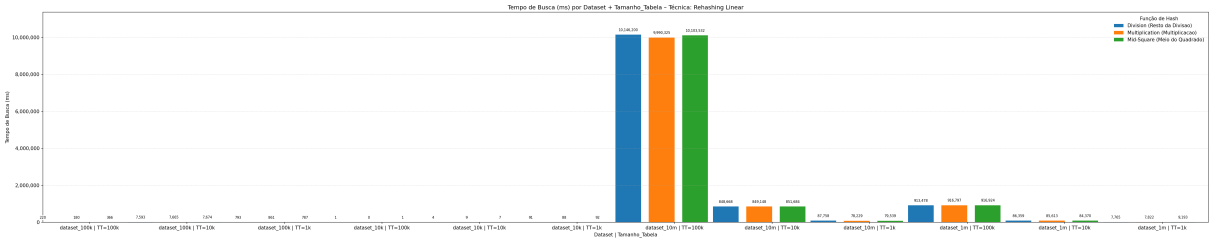

Figura 15: Gráfico de tempo de busca (ms) utilizando Rehashing Linear

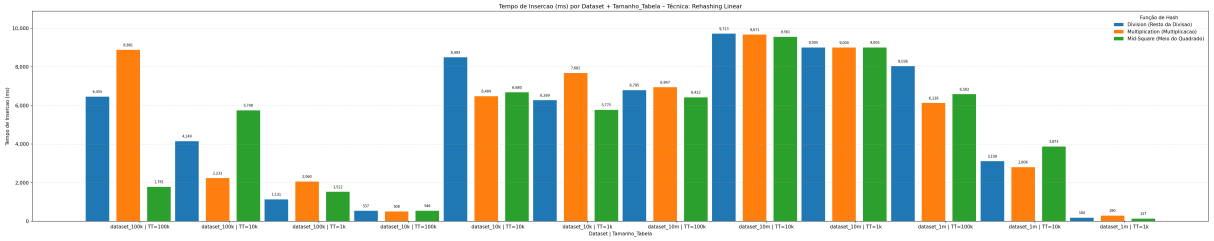

Figura 16: Gráfico de tempo de inserções (ms) utilizando Rehashing Linear

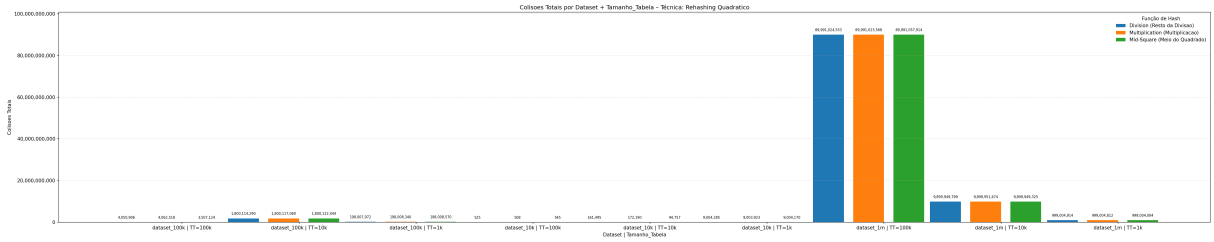

Figura 17: Gráfico da quantidade de colisões utilizando Rehashing Quadratico

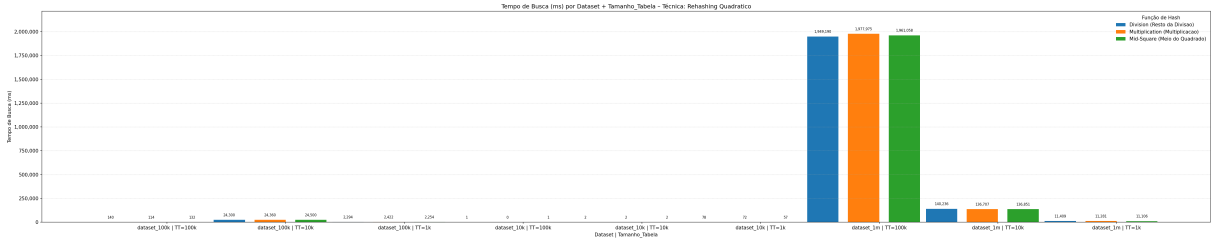

Figura 18: Gráfico de tempo de busca (ms) utilizando Rehashing Quadratico

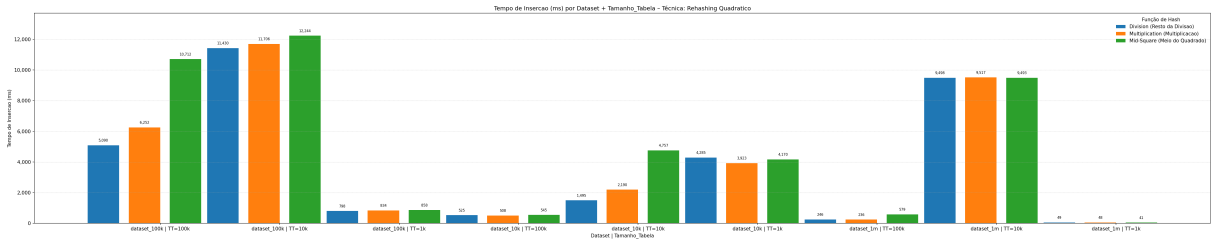

Figura 19: Gráfico de tempo de inserções (ms) utilizando Rehashing Quadratico