Avalição de desempenho entre diferentes métodos Hash

Tiago Vieira Paulin

Estrutura de Dados – Pontifícia Universidade Católica do Paraná (PUCPR) – Curitiba, Paraná – Brazil

tiago.paulin@pucpr.edu.br

Abstract. This report aims to propose a comparative analysis of different Hash encryption methods in Hash Tables. Through tests that take into account execution time, the number of collisions, and comparisons made, an analysis will be conducted between the folding, division remainder, and multiplication methods to assess their performance in various scenarios, with different table sizes and varying numbers of elements to conclude which method performs better in each case and why.

Resumo. Esse relatório tem como finalidade propor uma análise comparativa entre diferentes métodos de criptografia Hash em Tabelas Hash. Por meio de testes que levarão em conta o tempo de execução, número de colisões e comparações realizadas, será feita uma análise entre os métodos de dobramento, resto de divisão e multiplicação para averiguar o desempenho deles em diferentes cenários, com diferentes tamanhos de tabela e diferentes quantidades de elementos para concluir qual método se sai melhor em qual caso e o motivo.

1. Introdução

Primeiramente no relatório será feita uma uma breve explicação de como foi feita a implementação da tabela hash em Java bem como os métodos de inserção, busca e sobre cada método hash implementado no trabalho, com prints e explicação do código completo. Após isso será feita a análise comparativa, exibindo como foram feitos os testes, os resultados da execução do código e gráficos baseado nos resultados obtidos. Posteriormente será feita a conclusão do trabalho, que tem a finalidade de justificar os resultados obtidos bem como apontar as principais dificuldades enfrentadas na implementação do projeto.

2. Implementação da Tabela Hash

Uma Tabela Hash consiste em uma estrutura de dados que armazena os dados de forma organizada de acordo com um padrão estipulado pelo método de Hashing implementado. Por sua vez, o método Hashing consiste em um cálculo feito baseado no valor da chave que será armazenada na tabela, cálculo esse que resultará em um valor que corresponde a um índice que será a posição que a chave ocupará na tabela.

```java
public class HashTable {
    // definindo atributos
    15 usages
    private Registro[] hashTable;
    4 usages
    private int tamanho;
    3 usages
    private long totalColisoes;
    8 usages
    private long totalComparacoes;
    // metodo contrutor
    3 usages
    public HashTable(int tamanho){
        this.tamanho = tamanho;
        this.hashTable = new Registro[tamanho];
        this.totalColisoes = 0;
        this.totalComparacoes = 0;
    }
```

Figura 1. Atributos e Construtor da Tabela Hash

A implementação em Java da tabela possui os atributos hashTable, que será a tabela que armazenará as chaves inseridas, o atributo tamanho que corresponde ao tamanho da tabela que servirá para auxiliar o cálculo do hash, o atributo totalColisoes que contabiliza todas as colisões que foram identificadas na inserção dos elementos na tabela, o atributo totalComparacoes que contabiliza o total de comparações feitas na busca de elementos na tabela e o atributo algarismos que irá armazenar o total de algarismos que o valor do tamanho possui. No método construtor, todos os atributos da tabela são inicializados, a tabela é criada com o tamanho recebido no método construtor, os atributos totalComparacoes e totalColisões são inicializados como zero pois ainda não houve nenhuma colisão nem comparação no momento em que a tabela é criada.

2.1.Inserção na Tabela Hash

Um dos principais fatores a se levar em conta ao fazer uma inserção em uma tabela Hash são as colisões de chaves. As colisões ocorrem quando se tenta inserir uma chave na tabela e o hash resultante da chave referencia uma posição que já está sendo ocupada por outra chave na tabela, ocorrendo assim a colisão. As colisões em tabelas hash podem ser tratadas de duas formas, fazendo o rehashing da chave ou usando encadeamento, que consiste em usar a lógica de LinkedList para organizar várias chaves em sequência dentro da mesma posição na tabela.

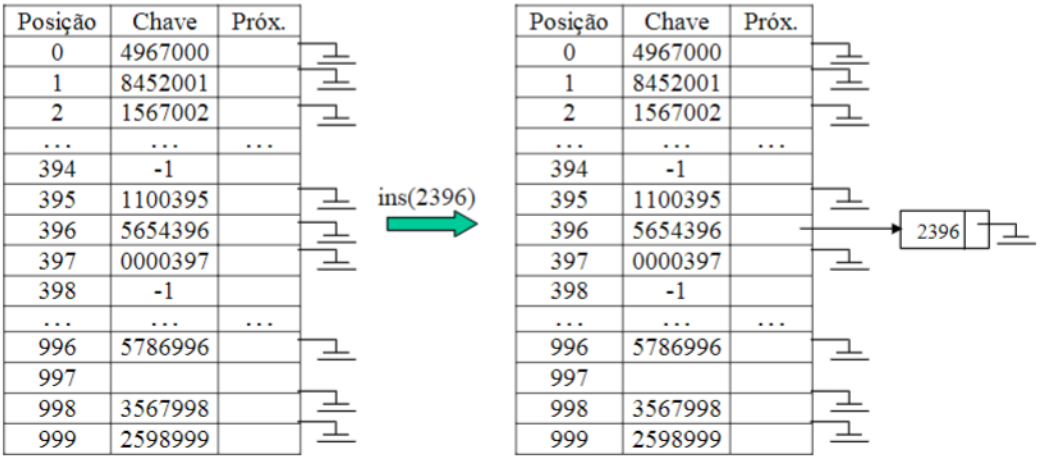


Figura 2. Exemplo de inserção em tabela Hash por encadeamento

Como método de solucionar as colisões nesse trabalho foi escolhido o encadeamento. Para isso, vale ressaltar que a tabela armazena objetos do tipo Registro, a classe registro, por sua vez, possui dois atributos, o atributo key, que corresponde a chave e o atributo next que é um atributo do tipo registro. Por meio desse atributo será possível fazer o encadeamento de vários registros, um em sequência do outro. A classe registro possui como métodos o construtor, que inicializa os atributos como null, os métodos getters e setters dos atributos para que os registros possam ser manipulados na inserção e busca na tabela hash.

```java
public class Registro {
    // definindo atributos
    3 usages
    private Integer key;
    3 usages
    private Registro next;
    // metodo construtor
    1 usage
    public Registro(){
        this.key = null;
        this.next = null;
    }
    // metodos get
    14 usages
    public Integer getKey() { return key; }
    8 usages
    public Registro getNext() { return next; }
    // metodos set
    1 usage
    public void setKey(Integer key) { this.key = key; }
    4 usages
    public void setNext(Registro next) { this.next = next; }
}
```

Figura 3. Implementação da classe Registro

```java
// metodo para inserir na tabela
3 usages
public void inserir(Integer key, int tipoHash){ // recebe a chave e o tipo de hash que será feito na inserção
    Registro registro = new Registro(); // cria um novo registro
    registro.setKey(key); // armazena o valor no registro
    int hash; // atributo que vai armazenar o valor de hash correspondente a chave recebida
    int colisoes = 0; // atributo que vai contabilizar o número de colisões caso a posição hash ja esteja ocupada com um registro
    if(tipoHash == 1){ // se o tipo do hash for 1
        hash = hashResto(key); // o valor do hash será calculado baseado no resto da divisão
    } else if(tipoHash == 2){ // se o tipo do hash for 2
        hash = hashMultiplicacao(key); // o valor do hash será calculado baseado na multiplicação pelo fator A
    } else { // se o tipo do hash for 3
        hash = hashDobramento(key); // o valor do hash será calculado por meio de dobramento
    }
```

Figura 4. Implementação do método inserir

```java
if(this.hashTable[hash] == null){ // após o hash ser calculado será verificado se a posição correspondente na tabela já está ocupada por um registro
   hashTable[hash] = registro; // caso esteja vazia aquela posição irá receber o registro
} else { // caso a posição já esteja ocupada por um registro
    // o tratamento de colisões será feito por meio de encadeamento (LinkedList)
    Registro temp = hashTable[hash]; // armazena o primeiro registro em uma variável auxiliar
    colisoes ++; // contabiliza uma colisão
    if(temp.getKey() >= key){ // caso o valor do registro que ja estava ocupando a posição for menor ou igual o registro que está sendo inserido
        registro.setNext(temp); // o registro que estava ocupando a posição será setado como próximo em relação ao registro que está sendo inserido
        hashTable[hash] = registro; // o novo registro ocupa a posição do registro que estava aqui anteriormente
    } else { // caso o valor do registro que já esteja ocupando a posição seja maior que o valor que está entando
        Registro atual = temp; // cria um registro para percorrer os registros da posição
        while((atual.getNext() != null) && (atual.getNext().getKey() < key)){ // enquanto o proximo nao for nulo e o proximo for maior que a chave
            atual = atual.getNext(); // percorre a linkedlist
            colisoes ++; // contabilisa uma colisão a cada registro que é comparado com a chave
        }
        // após encontrar o registro que ira setar o novo registro como proximo
        if(atual.getNext() == null){ // verifico se já existe algum registro setado como próximo dele
            atual.setNext(registro); // caso não eu seto meu novo registro
        } else { // caso tenha
            registro.setNext(atual.getNext()); // seto como proximo do novo registro o proximo do registro que irá receber o novo
            atual.setNext(registro); // e seto meu novo registro como próximo do no, dessa forma meu registro fica entre os dois registros mantendo a
        }
    }
}
this.totalColisoes += colisoes; // soma as colições registradas na inserção do registro ao total de colisões
// feedback da inserção no terminal (comentado para não atrapalhar análises com grandes quantidades de elementos)
  System.out.println("Código " + key + " inserido na Tabela Hash.");
  System.out.println("Número de colisões registradas: " + colisoes);
```

Figura 5. Continuação da implementação do método inserir

O método inserir vai receber a chave que será armazenada na tabela e o tipo do Hash que será feito que são definidos pelos números 1, 2 e 3, lembrando que também poderia ser feito um método inserir para cada tipo de Hash, mas foi preferível fazer dessa forma pois as chamadas do método são manuais e para fins de testes. Primeiramente é instanciado um objeto da classe Registro e a chave é setada no objeto, depois são criadas dois atributos, o hash que será o índice que o registro irá ocupar na tabela e o atributo colisões que caso haja colisões durante a inserção de um registro, as mesmas serão contabilizadas e ao fim da execução serão somadas ao total de colisões da tabela. Após isso, o atributo hash vai receber o valor retornado pelo método hash escolhido.

Com o valor do hash em mãos se da início a segunda etapa da inserção, primeiramente será verificado se a posição do hash na tabela está sendo ocupada por um registro, caso a posição

apontada pelo hash seja null o registro ocupa aquela posição e a inserção está concluída com um total de zero colisões. Caso haja algum registro na posição apontada pelo hash é contabilizada uma colisão e verifica se o valor da chave que está sendo inserida é menor ou igual a chave que já estava ocupando a posição, se sim, a nova chave ocupa a primeira posição da lista encadeada e o registro é setado como próximo do novo, dessa forma a lista encadeada na posição da tabela será organizada de forma crescente de acordo com as chaves. Caso o valor da chave for maior que o primeiro registro na posição, a lista encadeada será percorrida até o final ou até encontrar um valor maior que a chave inserida, a cada registro percorrido é contabilizada uma colisão, ao fim o novo registro será inserido no final da lista encadeada ou entre dois registros, um menor e outro maior que a chave dele.

2.2. Busca na Tabela Hash

A busca na tabela Hash consiste em encontrar o registro baseado no valor da chave que o método de busca recebe. A busca, assim como a inserção é feita baseada no hash do valor que está sendo procurado, dessa forma otimiza o processo de busca, pois não será necessário passar por todas as posições da tabela, ela vai direto na posição em que a chave estaria e caso necessário percorre a lista encadeada até que o valor seja encontrado e por fim retorna o mesmo

```
// metodo para  buscar chave na tabela hash
1 usage
public Integer buscaHashResto(int key){ // recebe a chave que será buscada na tabela
    int hash = hashResto(key); // calcula o hash da chave
    if(hashTable[hash].getKey() == key){ // se a chave se encontra na primeira posição da tabela
        this.totalComparacoes ++; // contabiliza uma comparação
        return hashTable[hash].getKey(); // retorna o valor do registro
    } else { // caso a chave não se encontre na primeira posição da tabela
        Registro atual = hashTable[hash]; // atributo auxiliar para varrer a lista ligada
        while(atual.getKey() != key){ // enquanto o valor do registro não corresponder com a chave buscada
            atual = atual.getNext(); // percorre a lista ligada
            this.totalComparacoes ++; // contabiliza uma comparação
        }
        return atual.getKey(); // retona o valor do registro assim que for encontrado
    }
}
```

Figura 6. Implementação do método de busca para o Hash de Resto da Divisão

```
1 usage
public Integer buscaHashMultiplicacao(int key){ // recebe a chave que será  buscada na tabela
    int hash = hashMultiplicacao(key); // calcula o hash da chave que está sendo buscada
    if(hashTable[hash].getKey() == key){ // se a chave corresponde ao valor do registro na primeira posição
        this.totalComparacoes ++; // contabiliza uma comparação
        return hashTable[hash].getKey(); // retorna o valor do registro
    } else { // se nao
        Registro atual = hashTable[hash]; // atributo auxiliar para percorrer a linkedlist
        while(atual.getKey() != key){ // enquanto o valor do registro não corresponder a chave buscada
            atual = atual.getNext(); // percorre a lista
            this.totalComparacoes ++; // contabiliza uma comparação
        }
        return atual.getKey(); // retorna o valor do registro assim que encontrar
    }
}
```

Figura 7. Implementação do método de busca para o Hash de Multiplicação

```java
public Integer buscaHashDobramento(int key){ // recebe a chave que será buscada na tabela
    int hash = hashDobramento(key); // calcula o hash da chave
    if(hashTable[hash].getKey() == key){ // se corresponde a primeira posição
        this.totalComparacoes ++; // contabiliza uma comparação
        return hashTable[hash].getKey(); // retorna o valor do registro
    } else { // se nao
        Registro atual = hashTable[hash]; // atributo auxiliar para varrer os registros
        while(atual.getKey() != key){ // enquanto o valor do registro não corresponder a chave buscada
            atual = atual.getNext(); // percorre a lista
            this.totalComparacoes ++; // contabiliza uma comparação
        }
        return atual.getKey(); // retona o valor do registro assim que for encontrado
    }
}
```

Figura 8. Implementação do método de busca para o Hash de Dobramento

Os foi implementado um método de busca para cada tipo de hash, todos da mesma forma com exceção da chamada do método de hashing. A cada Registro que a busca verifica se coincide com a chave procurada, é contabilizada uma comparação.

2.3 Métodos Hash

Exitem várias formas de hashing, as ecolhidas para o trabalho são o hashing de Resto, Multiplicação e hash de Dobramento.

O hash de resto ou hash modular consiste em realizar uma divisão entre o valor da chave que será armazenada no registro e o tamanho da tabela e o resto dessa divisão será o valor do Hash, ou seja, corresponderá a posição do registro na tabela

```java
// hash de resto de divisão
2 usages
public Integer hashResto(Integer key){ // recebe a chave que para calcula o hash
    return key % this.tamanho; // retorna o valor do hash da chave, que corresponde ao resto da divisão pelo tamanho da tabela
}
```

Figura 9. Implementação do método hashResto.

Este método retorna o valor do hash da chave que ele recebe. Por exemplo, ao receber uma chave de valor 123456789 para ser inserido em uma tabela de tamanho 10000, o hash dessa chave ira indicar a posição 6789 da tabela.

O hash de Multiplicação consiste em utilizar de um fator de multiplicação A, onde $0 < A < 1$, o valor de A pode variar de acordo com a aplicação, mas o mais indicado para cálculo de hash e que foi utilizado nesse tabalho é $A = (\sqrt{5} – 1) / 2$ que resulta em um valor aproximado de 0.61803398875. Esse fator de multiplicação é multiplicado pelo valor da chave com o intuito de obter a parte fracionária da operação, após isso a parte inteira é multiplicada pelo tamanho da tabela retornando como hash a parte inteira dessa operação. Por exemplo, na inserção de uma chave com valor 123456789 em uma tabela de tamanho 10000, o hash resultante desse valor ira corresponder a posição 7377 na tabela.

```
3 usages
public Integer hashMultiplicacao(Integer key){ // recebe a chave para calcular o hash
    double a = 0.6180339887; // fator de multiplicação que corresponde ao valor aproximado de raíz de 5 menos 1 dividido por 2
    double value = key * a; // multiplica o chave pelo fator de multiplicação escolhido
    double fracao = value - (int) value; // pega somente a parte fracionária da operação
    return (int) (fracao * this.tamanho); // retorna o hash da chave que corresponde a parte fracionária da operação multiplicada pelo tamanho da tabela
}
```

Figura 10. Implementação do método hashMultiplicação

O Hash de Dobramento, dentre os escolhidos é o de implementação mais complexa e com mais variações de implementação, o método consiste em realizar contantes “dobras” no valor da chave até que ela corresponda a uma posição da tabela. As dobras consistem em pegar um algarismo da chave, somar com outro e concatenar a parte unitária do resuldado dessa soma no valor da chave.

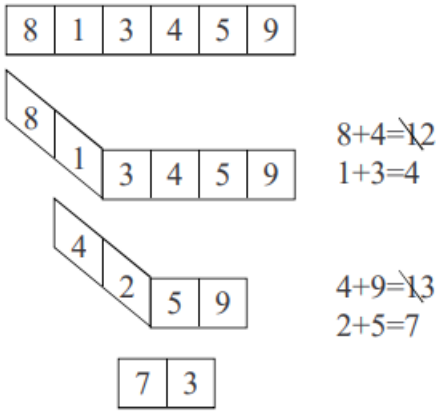


Figura 11. Exemplificando dobramentos de chave

```
public Integer hashDobramento(Integer key){ // recebe a chave para ser calculado o hash
    int chave = key; // atributo de apoio que recebe a chave
    int a = chave % 100000; // atributo a pega os ultimos 5 digitos da chave
    chave = chave / 100000; // atualiza a chave sem os valores que foram para a
    return (a + chave) % this.tamanho; // retorna o hash que é o resto da divisão entre a soma das partes e o tamanho da tabela
}
// metodos get
```

Figura 12. Implementação do método hashDobramento

Na implementação do método, foi escolhido um método mais simples de dobra, onde a chave é dividida em duas partes, os valores são somados e o resto da divisão desses valores pelo tamanho da tabela será o hash da chave. Por exemplo, supondo que a chave de entrada seja 123456789, primeiramente a variável a receberá 56789 e a chave será atualizada para 1234, os valores serão somados e o resto da divisão pelo tamanho da tabela será o hash, que no caso de uma tabela com tamanho 10000, será 8023.

2.4. Métodos Get

Os métodos getters da tabela foram implementados para facilitar o feedback da execução dos testes no terminal.

```
// metodos get
3 usages
public long getTotalComparacoes() { return totalComparacoes; }
3 usages
public long getTotalColisoes() { return totalColisoes; }
no usages
public Registro[] getHashTable() { return hashTable; }
```

Figura 13. Implementação dos métodos Get

3. Execução dos Testes

Os teste de métodos hash funcionarão da seguinte forma, o código será executado e serão escolhidos o tamanho da tabela que será instanciada e o número de elementos que serão inseridos e buscados na tabela.

```
// definindo número de elementos
System.out.println("=== TABELA HASH ===");
System.out.println("Selecione o número de elementos para inserir na tabela hash");
System.out.println("1. 20 mil elementos");
System.out.println("2. 100 mil elementos");
System.out.println("3. 500 mil elementos");
System.out.println("4. 1 milhão de elementos");
System.out.println("5. 5 milhões de elementos");
System.out.println(" ");
System.out.print("Escolha uma das opções: ");
int opcaoElementos = scan.nextInt();
switch (opcaoElementos){
    case 1:
        nElementos = 20000;
        break;
    case 2:
        nElementos = 100000;
        break;
    case 3:
        nElementos = 500000;
        break;
    case 4:
        nElementos = 1000000;
        break;
    case 5:
        nElementos = 5000000;
        break;
    default:
        System.out.println("Opção inválida");
        break;
}
```

Figura 14. Menu de seleção de número de elementos

```
System.out.println("Selecione o tamanho da tabela hash");
System.out.println("1. 10000");
System.out.println("2. 20000");
System.out.println("3. 35000");
System.out.println("4. 50000");
System.out.println("5. 100000");
System.out.println(" ");
System.out.print("Escolha uma das opções: ");
int opcaoTamanho = scan.nextInt();
switch (opcaoTamanho){
    case 1:
        tamanhoTabela = 10000;
        break;
    case 2:
        tamanhoTabela = 20000;
        break;
    case 3:
        tamanhoTabela = 35000;
        break;
    case 4:
        tamanhoTabela = 50000;
        break;
    case 5:
        tamanhoTabela = 100000;
        break;
    default:
        System.out.println("Opção inválida");
        break;
}
scan.close();
```

Figura 15. Menu de seleção de tamanho da tabela

Após o tamanho da tabela e o número de elementos serem definidos será criado um vetor de inteiros com o tamanho correspondente ao número de elementos a serem inseridos e este vetor é populado com números aleatórios de 9 dígitos. A criação desse vetor irá auxiliar nos testes de inserção e busca. Depois do vetor ser criado são instanciados 3 objetos do tipo HashTable, um para cada tipo de Hashing.

```java
// preenchendo vetor com os dados que serão utilizados
vetor = new int[nElementos];
for(int i = 0; i < nElementos; i ++){
    int chave = Integer.parseInt(String.format("%09d", rd.nextInt( bound: 1000000000)));
    vetor[i] = chave;
}
// instanciando as tabelas
HashTable hashTableDivisao = new HashTable(tamanhoTabela);
HashTable hashTableMultiplicacao = new HashTable(tamanhoTabela);
HashTable hashTableDobramento = new HashTable(tamanhoTabela);
```

Figura 16. Criação do vetor e instância dos objetos

Com os objetos criados serão iniciados os testes, primeiramente será cronometrado o tempo em nanossegundos para a inserção de todos os elementos na tabela e todas as colisões que foram registradas durante a inserção dos elementos, após isso os resultados serão mostrados no terminal. Em seguida todos os elementos da tabela serão buscados 5 vezes a fim de obter uma média de tempo para buscar todos os elementos que será também cronometrado em nanossegundos e as comparações realizadas no processo também serão registradas, os resultados serão mostrados no terminal e o mesmo processo será feito para os 3 Hashings.

```java
// inserindo valores na tabela hash
inicioResto = System.nanoTime();
for (int key : vetor){
    hashTableDivisao.inserir(key, tipoHash: 1);
}
fimResto = System.nanoTime();
long insercaoResto = fimResto - inicioResto;
// feedback da inserção no terminal
System.out.println("\n");
System.out.println("=====    HASH DE DIVISÃO    =====");
System.out.println("Tempo em nanossegundos para a inserção: " + insercaoResto + "ns" + " Aproximadamente " + insercaoResto / 1000000000 + "s");
System.out.println("Total de colisões da inserção: " + hashTableDivisao.getTotalColisoes());
System.out.println(" ");
```

Figura 17. Inserção dos elementos pelo hash de divisão

```java
// buscando valores na tabela hash
inicioResto = System.nanoTime();
for(int i = 0; i < 5; i ++){
    for (int key : vetor){
        hashTableDivisao.buscaHashResto(key);
    }
}
fimResto = System.nanoTime();
long buscaResto = (fimResto - inicioResto) / 5;
// feedback da busca no terminal
System.out.println("Média de tempo em nanossegundos para a busca de todos os elementos: " + buscaResto + "ns" + " Aproximadamente " + buscaResto / 1000000000 + "s");
System.out.println("Total de comparações realizadas na busca de todos os elementos: " + hashTableDivisao.getTotalComparacoes() / 5);
```

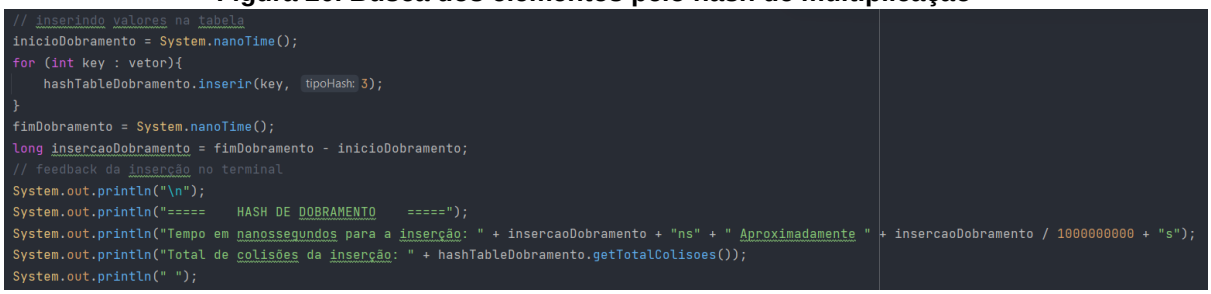
Figura 18. Busca dos elementos pelo hash de divisão

```java
// inserindo valores na tabela hash
inicioMultiplicacao = System.nanoTime();
for (int key : vetor){
    hashTableMultiplicacao.inserir(key, tipoHash: 2);
}
fimMultiplicacao = System.nanoTime();
long insercaoMultiplicacao = fimMultiplicacao - inicioMultiplicacao;
// feedback da inserção no terminal
System.out.println("\n");
System.out.println("=====    HASH DE MULTIPLICAÇÃO    =====");
System.out.println("Tempo em nanossegundos para a inserção: " + insercaoMultiplicacao + "ns" + " Aproximadamente " + insercaoMultiplicacao / 1000000000 + "s");
System.out.println("Total de colisões da inserção: " + hashTableMultiplicacao.getTotalColisoes());
System.out.println(" ");
```

Figura 19. Inserção dos elementos pelo hash de multiplicação

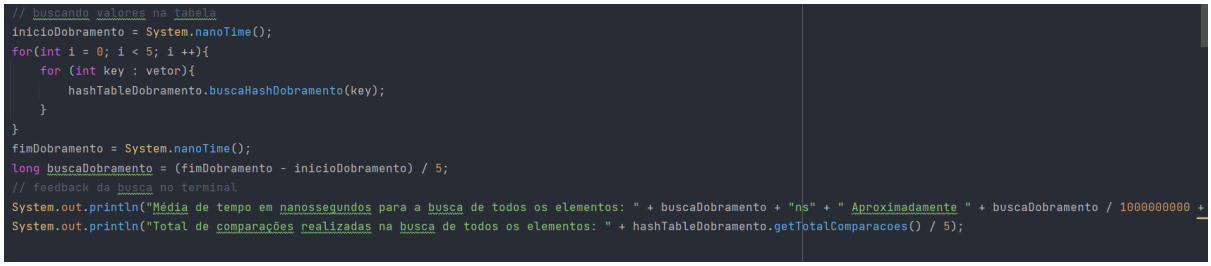
```
// buscando valores na tabela
inicioMultiplicacao = System.nanoTime();
for(int i = 0; i < 5; i ++){
    for (int key : vetor){
        hashTableMultiplicacao.buscaHashMultiplicacao(key);
    }
}
fimMultiplicacao = System.nanoTime();
long buscaMultiplicacao = (fimMultiplicacao - inicioMultiplicacao) / 5;
// feedback da busca no terminal
System.out.println("Média de tempo em nanossegundos para a busca de todos os elementos: " + buscaMultiplicacao+ "ns" + " Aproximadamente " + buscaMultiplicacao / 1000000
System.out.println("Total de comparações realizadas na busca de todos os elementos: " + hashTableMultiplicacao.getTotalComparacoes() / 5);
```

Figura 20. Busca dos elementos pelo hash de multiplicação



```
// inserindo valores na tabela
inicioDobramento = System.nanoTime();
for (int key : vetor){
    hashTableDobramento.inserir(key, tipoHash: 3);
}
fimDobramento = System.nanoTime();
long insercaoDobramento = fimDobramento - inicioDobramento;
// feedback da inserção no terminal
System.out.println("\n");
System.out.println("=====    HASH DE DOBRAMENTO    =====");
System.out.println("Tempo em nanossegundos para a inserção: " + insercaoDobramento + "ns" + " Aproximadamente " + insercaoDobramento / 1000000000 + "s");
System.out.println("Total de colisões da inserção: " + hashTableDobramento.getTotalColisoes());
System.out.println(" ");
```

Figura 21. Inserção dos elementos pelo hash de dobramento



```
// buscando valores na tabela
inicioDobramento = System.nanoTime();
for(int i = 0; i < 5; i ++){
    for (int key : vetor){
        hashTableDobramento.buscaHashDobramento(key);
    }
}
fimDobramento = System.nanoTime();
long buscaDobramento = (fimDobramento - inicioDobramento) / 5;
// feedback da busca no terminal
System.out.println("Média de tempo em nanossegundos para a busca de todos os elementos: " + buscaDobramento + "ns" + " Aproximadamente " + buscaDobramento / 1000000000 +
System.out.println("Total de comparações realizadas na busca de todos os elementos: " + hashTableDobramento.getTotalComparacoes() / 5);
```

Figura 22. Busca dos elementos pelo hash de dobramento

Para exemplificar o funcionamento do teste será executada as etapas utilizando uma tabela hash de tamanho 10.000 com 5 milhões de elementos, os resultados obtidos serão printados e anexados a seguir.

```
=====    HASH DE DIVISÃO    =====
Tempo em nanossegundos para a inserção: 54498995800ns Aproximadamente 54s
Total de colisões da inserção: 625190468

Média de tempo em nanossegundos para a busca de todos os elementos: 88042631680ns Aproximadamente 88s
Total de comparações realizadas na busca de todos os elementos: 1250005410
```

Figura 23. Resultados do Hash de Divisão

```
=====    HASH DE MULTIPLICAÇÃO    =====
Tempo em nanossegundos para a inserção: 34301803000ns Aproximadamente 34s
Total de colisões da inserção: 625208980

Média de tempo em nanossegundos para a busca de todos os elementos: 99363538100ns Aproximadamente 99s
Total de comparações realizadas na busca de todos os elementos: 1249998392
```

Figura 24. Resultados do Hash de Multiplicação

```
=====    HASH DE DOBRAMENTO    =====
Tempo em nanossegundos para a inserção: 38068179900ns Aproximadamente 38s
Total de colisões da inserção: 625184111


Média de tempo em nanossegundos para a busca de todos os elementos: 95550229260ns Aproximadamente 95s
Total de comparações realizadas na busca de todos os elementos: 1249991417
```

Figura 25. Resultados do Hash de Dobramento

Para realizar a análise, como mostrado, teremos grupos de elementos de 20 mil, 100 mil, 500 mil, 1 milhão e 5 milhões de elementos todos esses grupos de elementos serão testados em todos os tamanhos de tabela, que são tabelas de 10mil, 20 mil, 35 mil, 50 mil e 100 mil de tamanho. Cada grupo de elementos terá seus dados coletados em relação a cada tamanho de tabela, os dados coletados serão o número de colisões, comparações, tempo de inserção e média de busca de cada grupo de elementos em cada tamanho de tabela, os dados serão armazenados em tabelas e depois representados por meio de gráficos para a análise de resultados, todas as tabelas e gráficos estarão presentes na seção 4 de resultados.

4. Resultados dos Testes

DIVISÃO INSERÇÃO			
N° ELEMENTOS	TAMANHO	N° COLISÕES	TEMPO
5000000	100000	62863498	8224285300ns (8s)
5000000	50000	125242127	8685417900ns (8s)
5000000	35000	178785763	14208739300ns (14s)
5000000	20000	312653610	26709205500ns (26s)
5000000	10000	625190468	30881784500ns (30s)
1000000	100000	2687225	297027700ns (0s)
1000000	50000	5123859	710650800ns (0s)
1000000	35000	7235764	876463000ns (0s)
1000000	20000	12555556	2661144800ns (2s)
1000000	10000	25010658	3978256400ns (3s)
500000	100000	744022	76608400ns (0s)
500000	50000	1342410	110664600ns (0s)
500000	35000	1860667	185233200ns (0s)
500000	20000	3175593	311507600ns (0s)
500000	10000	6274653	693976600ns (0s)
100000	100000	41385	42791500ns (0s)
100000	50000	72462	10319800ns (0s)
100000	35000	96144	29131400ns (0s)
100000	20000	149073	22056100ns (0s)
100000	10000	269524	18622600ns (0s)
20000	100000	1908	5116200ns (0s)
20000	50000	3686	4649800ns (0s)
20000	35000	5210	4603200ns (0s)
20000	20000	8348	2727600ns (0s)
20000	10000	14514	3782600ns (0s)

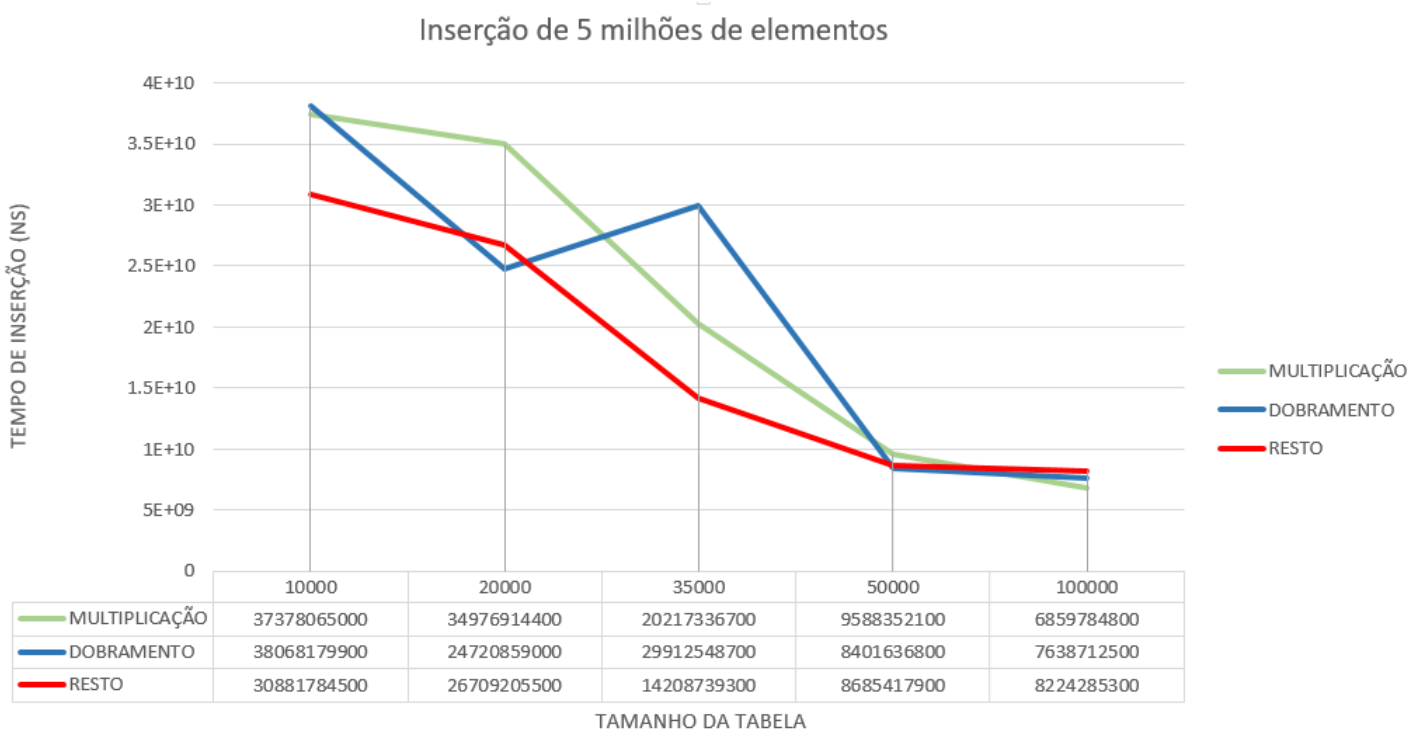
Figura 26. Coleta de Dados na Inserção com Hash de Resto

MULTIPLICAÇÃO INSERÇÃO			
N° ELEMENTOS	TAMANHO	N° COLISÕES	TEMPO
5000000	100000	62849209	6859784800ns (6s)
5000000	50000	125224853	9588352100ns (9s)
5000000	35000	178760556	20217336700ns (20s)
5000000	20000	312669042	34976914400ns (34s)
5000000	10000	625208980	37378065000ns (37s)
1000000	100000	2682318	426464400ns (0s)
1000000	50000	5118251	722166400ns (0s)
1000000	35000	7231657	677778100ns (0s)
1000000	20000	12546720	2065280900ns (2s)
1000000	10000	24998309	3678389400ns (3s)
500000	100000	742672	112651800ns (0s)
500000	50000	1340295	140051000ns (0s)
500000	35000	1859643	332002700ns (0s)
500000	20000	3172391	386268900ns (0s)
500000	10000	6266461	684012900ns (0s)
100000	100000	41423	63513100ns (0s)
100000	50000	72530	38115500ns (0s)
100000	35000	95741	25217400ns (0s)
100000	20000	148460	34987000ns (0s)
100000	10000	267952	16846900ns (0s)
20000	100000	1945	21387800ns (0s)
20000	50000	3735	4715500ns (0s)
20000	35000	5164	5259200ns (0s)
20000	20000	8284	7525200ns (0s)
20000	10000	14617	2968100ns (0s)

Figura 27. Coleta de Dados na Inserção com Hash de Multiplicação

DOBRAMENTO INSERÇÃO			
N° ELEMENTOS	TAMANHO	N° COLISÕES	TEMPO
5000000	100000	62858797	7638712500ns (7s)
5000000	50000	125227146	8401636800ns (8s)
5000000	35000	179619149	29912548700ns (29s)
5000000	20000	312640985	24720859000ns (24s)
5000000	10000	625184111	38068179900ns (38s)
1000000	100000	2685597	416463400ns (0s)
1000000	50000	5121885	1012681300ns (1s)
1000000	35000	7266606	1448431500ns (1s)
1000000	20000	12551595	2187024400ns (2s)
1000000	10000	24996434	4368415700ns (4s)
500000	100000	743255	144493800ns (0s)
500000	50000	1341003	169352900ns (0s)
500000	35000	1866370	238452000ns (0s)
500000	20000	3173010	311044500ns (0s)
500000	10000	6266776	299444500ns (0s)
100000	100000	41371	16954300ns (0s)
100000	50000	72632	12578700ns (0s)
100000	35000	96200	19593100ns (0s)
100000	20000	149324	21265600ns (0s)
100000	10000	269379	31955200ns (0s)
20000	100000	1849	4471000ns (0s)
20000	50000	3579	1053000ns (0s)
20000	35000	5158	1092000ns (0s)
20000	20000	8176	6690000ns (0s)
20000	10000	14447	9419100ns (0s)

Figura 28. Coleta de Dados na Inserção com Hash de Dobramento



	10000	20000	35000	50000	100000
MULTIPLICAÇÃO	37378065000	34976914400	20217336700	9588352100	6859784800
DOBRAMENTO	38068179900	24720859000	29912548700	8401636800	7638712500
RESTO	30881784500	26709205500	14208739300	8685417900	8224285300

Figura 29. Representação gráfica do tempo de inserção com 5 milhões de elementos

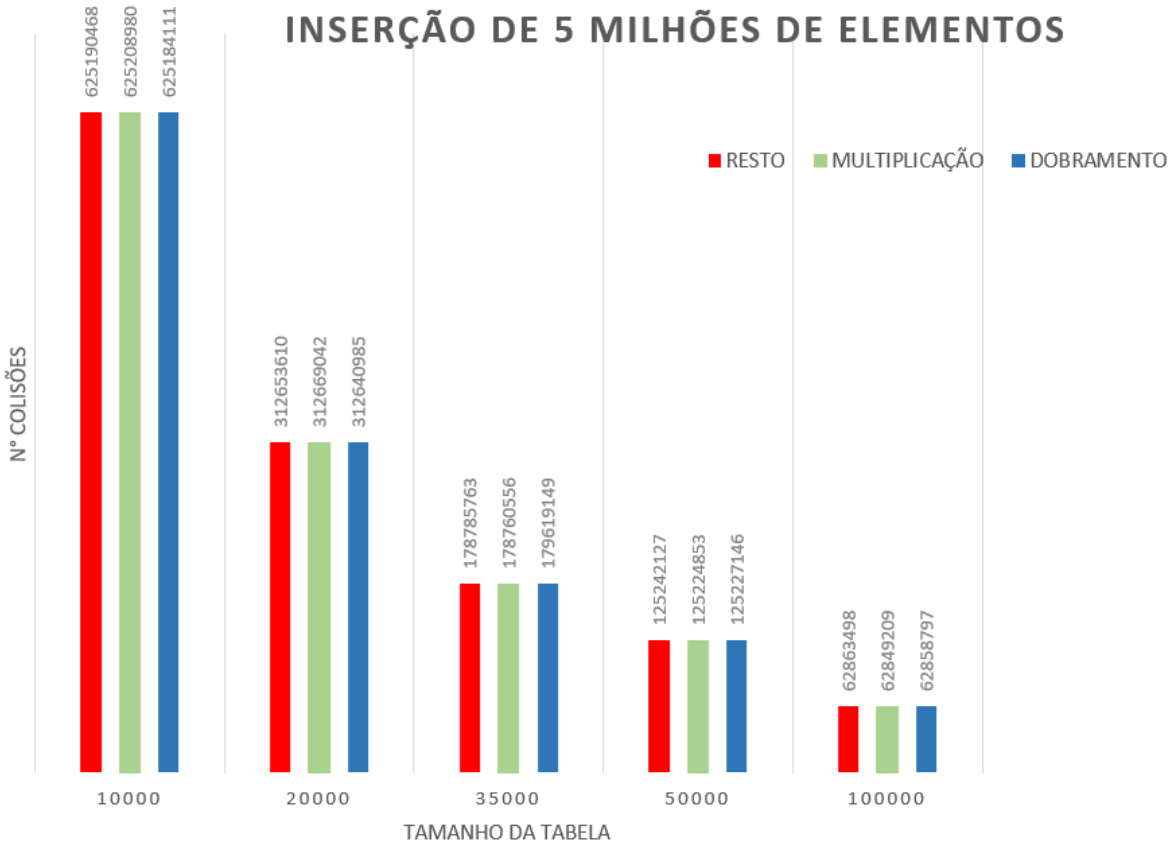
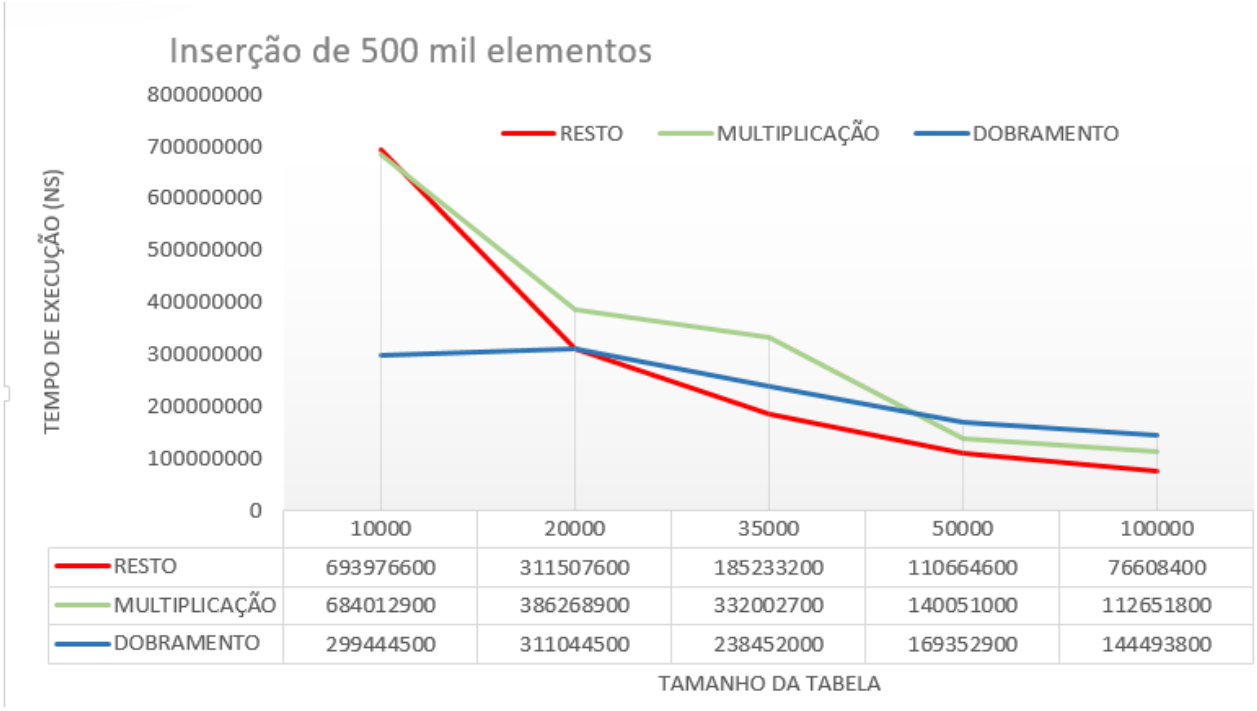


Figura 29. Representação gráfica do número de colisões com 5 milhões de elementos



	10000	20000	35000	50000	100000
RESTO	693976600	311507600	185233200	110664600	76608400
MULTIPLICAÇÃO	684012900	386268900	332002700	140051000	112651800
DOBRAMENTO	299444500	311044500	238452000	169352900	144493800

Figura 30. Representação gráfica do tempo de inserção com 500 mil elementos

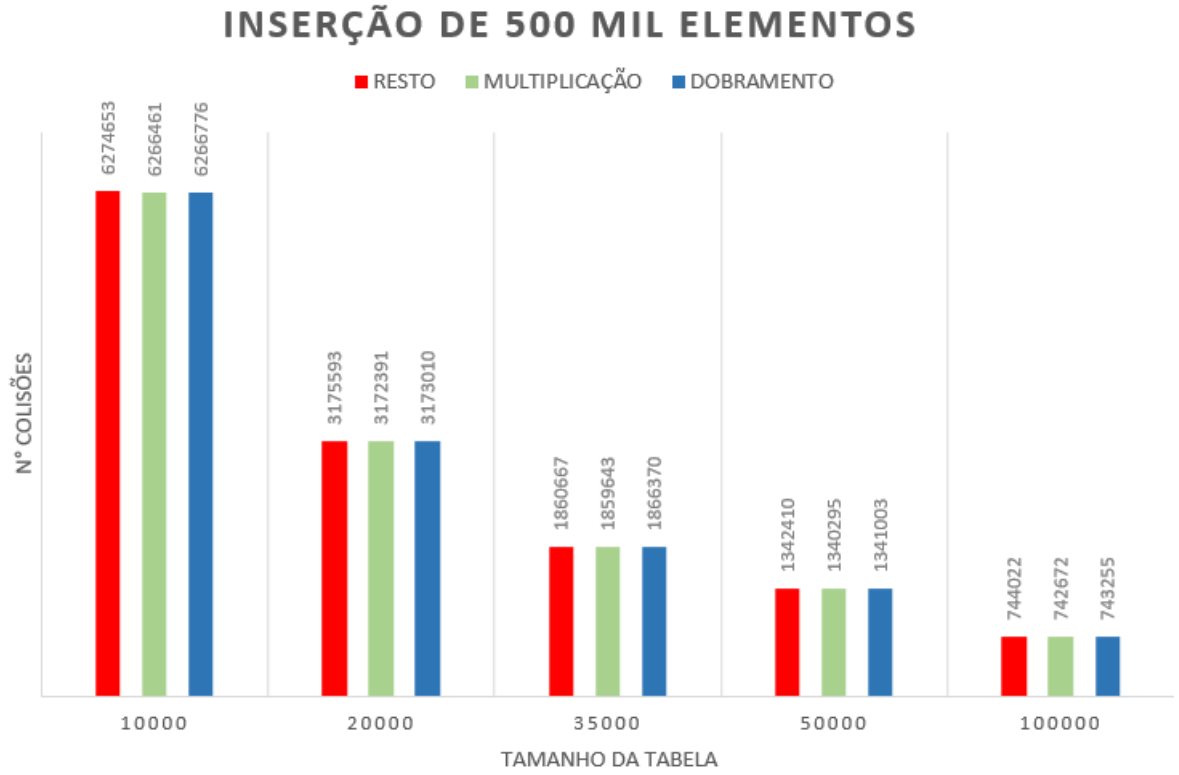


Figura 31. Representação gráfica do número de colisões com 500 mil elementos

	10000	20000	35000	50000	100000
RESTO	3782600	2727600	4603200	4649800	5116200
MULTIPLICAÇÃO	2968100	7525200	5259200	4715500	21387800
DOBRAMENTO	9419100	6690000	1092000	1053000	4471000

Figura 32. Representação gráfica do tempo de inserção com 20 mil elementos

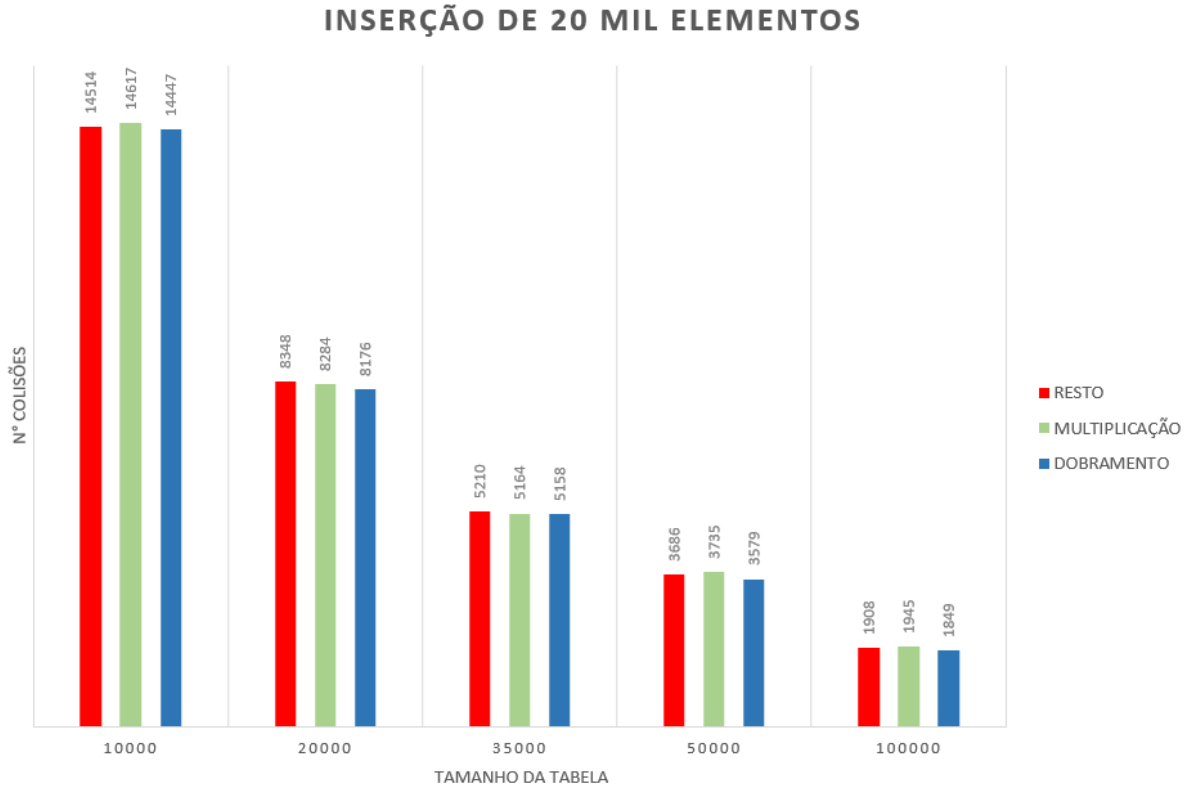


Figura 33. Representação gráfica do número de colisões com 20 mil elementos

DIVISÃO BUSCA			
N° ELEMENTOS	TAMANHO	N° COMPARAÇÕES	TEMPO
5000000	100000	125096305	16279055200ns (16s)
5000000	50000	250049427	14689124780ns (14s)
5000000	35000	357195847	32242049120ns (32s)
5000000	20000	625001140	27554614620ns (27s)
5000000	10000	1250005410	40450888620ns (40s)
1000000	100000	5101411	607086440ns (0s)
1000000	50000	10050857	1616784220ns (1s)
1000000	35000	14322746	2018109300ns (2s)
1000000	20000	25022370	4822818060ns (4s)
1000000	10000	50013218	8724345720ns (8s)
500000	100000	1350037	70806560ns (0s)
500000	50000	2551452	242341580ns (0s)
500000	35000	3607445	369546000ns (0s)
500000	20000	6270657	811296260ns (0s)
500000	10000	12513386	1989707060ns (1s)
100000	100000	113100	11747620ns (0s)
100000	50000	142784	7034000ns (0s)
100000	35000	175831	10101980ns (0s)
100000	20000	269258	13025100ns (0s)
100000	10000	509861	16917040ns (0s)
20000	100000	20116	3912620ns (0s)
20000	50000	20489	2604160ns (0s)
20000	35000	21040	1676440ns (0s)
20000	20000	22617	2584480ns (0s)
20000	10000	28663	1962460ns (0s)

Figura 34. Coleta de Dados na Busca com Hash de Resto

MULTIPLICAÇÃO BUSCA			
N° ELEMENTOS	TAMANHO	N° COMPARAÇÕES	TEMPO
5000000	100000	125072372	13428586480ns (13s)
5000000	50000	250021994	18760140600ns (18s)
5000000	35000	357156408	44773573220ns (44s)
5000000	20000	624999860	74334338080ns (74s)
5000000	10000	1249998392	99363538100ns (99s)
1000000	100000	5098257	449895820ns (0s)
1000000	50000	10050153	1757259660ns (1s)
1000000	35000	14321481	1047116700ns (1s)
1000000	20000	25022015	1113308600ns (1s)
1000000	10000	50012702	2445576020ns (2s)
500000	100000	1346883	115352860ns (0s)
500000	50000	2548488	269683560ns (0s)
500000	35000	3606473	412778320ns (0s)
500000	20000	6268663	978016520ns (0s)
500000	10000	12503567	1877411380ns (0s)
100000	100000	113165	12394360ns (0s)
100000	50000	142823	10280700ns (0s)
100000	35000	175903	22893440ns (0s)
100000	20000	269070	10360980ns (0s)
100000	10000	508749	12631260ns (0s)
20000	100000	20136	8059540ns (0s)
20000	50000	20495	8921300ns (0s)
20000	35000	21022	4581100ns (0s)
20000	20000	22621	5779180ns (0s)
20000	10000	28795	3896840ns (0s)

Figura 35. Coleta de Dados na Busca com Hash de Multiplicação

DOBRAMENTO BUSCA			
N° ELEMENTOS	TAMANHO	N° COMPARAÇÕES	TEMPO
5000000	100000	125092001	18519878000ns (18s)
5000000	50000	250030958	16591120660ns (16s)
5000000	35000	358862441	9968983380ns (9s)
5000000	20000	624979315	48631803200ns (48s)
5000000	10000	1249991417	66557332020ns (66s)
1000000	100000	5099813	605502200ns (0s)
1000000	50000	10049438	1848556040ns (1s)
1000000	35000	14386810	2403277720ns (2s)
1000000	20000	25022688	4478291280ns (4s)
1000000	10000	50001526	8018653960ns (8s)
500000	100000	1349783	103496700ns (0s)
500000	50000	2550142	206561460ns (0s)
500000	35000	3622577	438233540ns (0s)
500000	20000	6271335	769817420ns (0s)
500000	10000	12507862	631696020ns (0s)
100000	100000	113182	7726300ns (0s)
100000	50000	142952	6859500ns (0s)
100000	35000	176385	10814540ns (0s)
100000	20000	269968	19149480ns (0s)
100000	10000	509945	21611240ns (0s)
20000	100000	20134	7139380ns (0s)
20000	50000	20480	4282880ns (0s)
20000	35000	20982	3662300ns (0s)
20000	20000	22555	769360ns (0s)
20000	10000	28515	8297600ns (0s)

Figura 36. Coleta de Dados na Busca com Hash de Dobramento

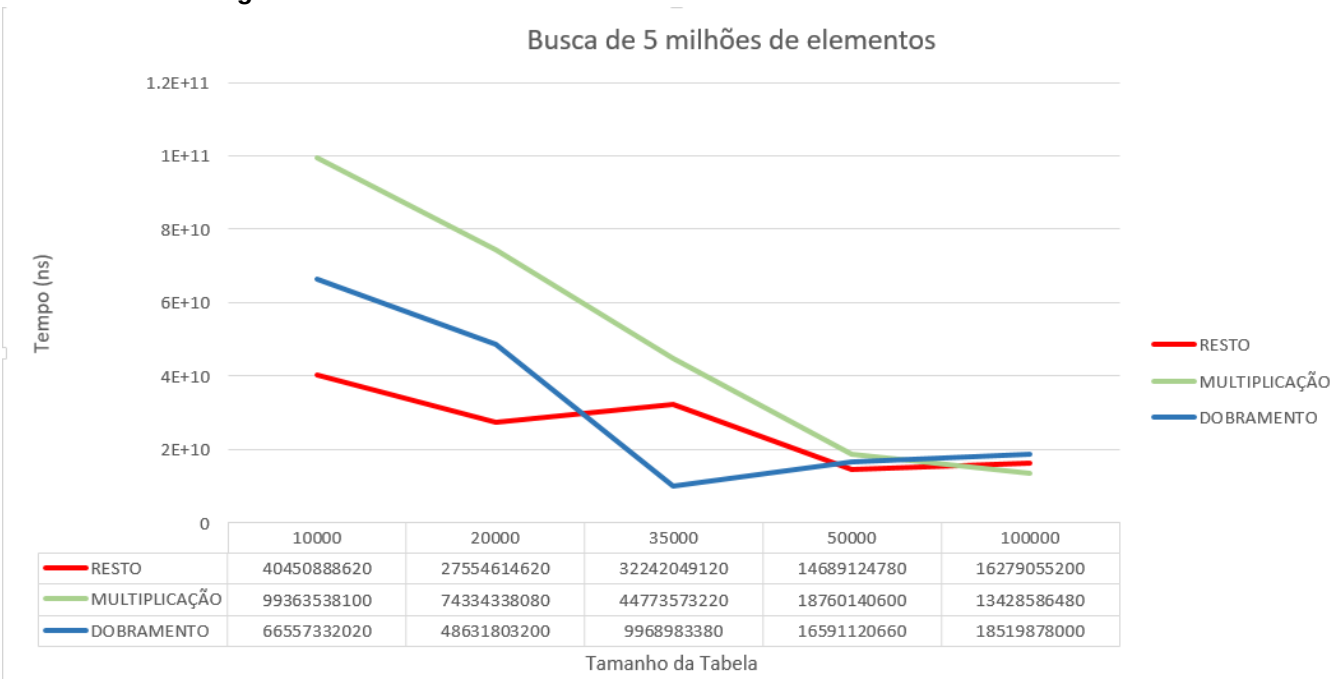


Figura 37. Representação gráfica do tempo de busca de 5 milhões de elementos

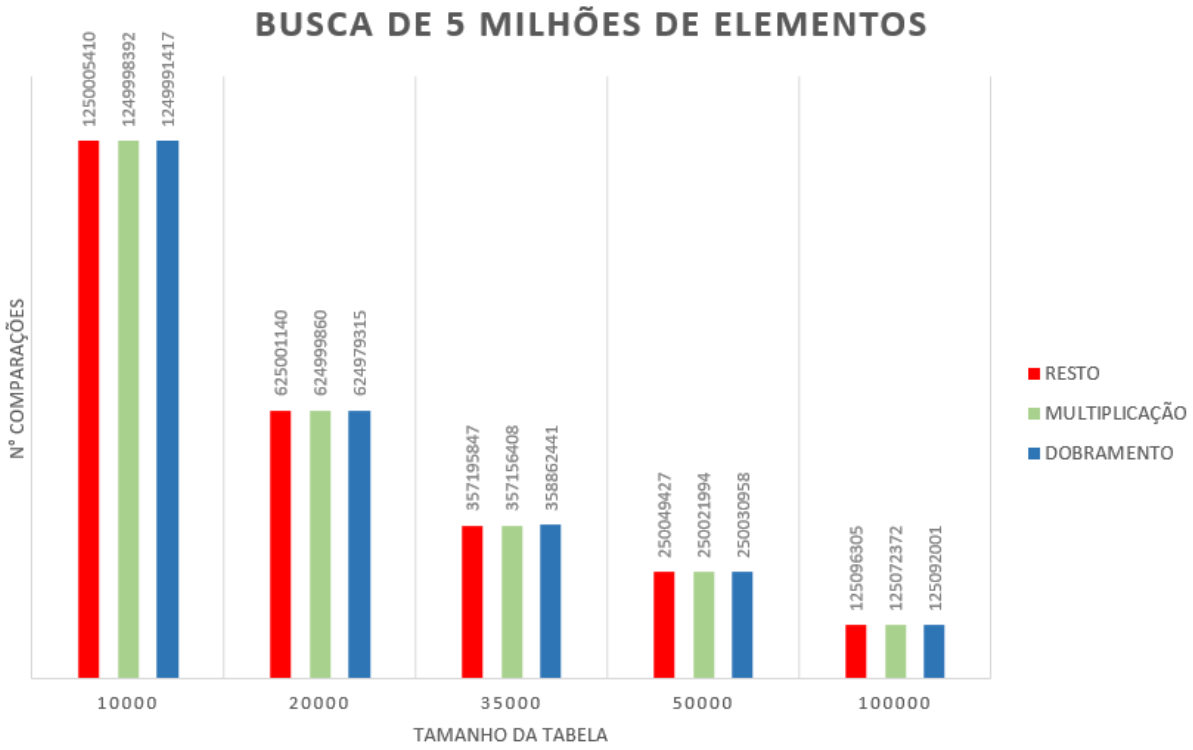
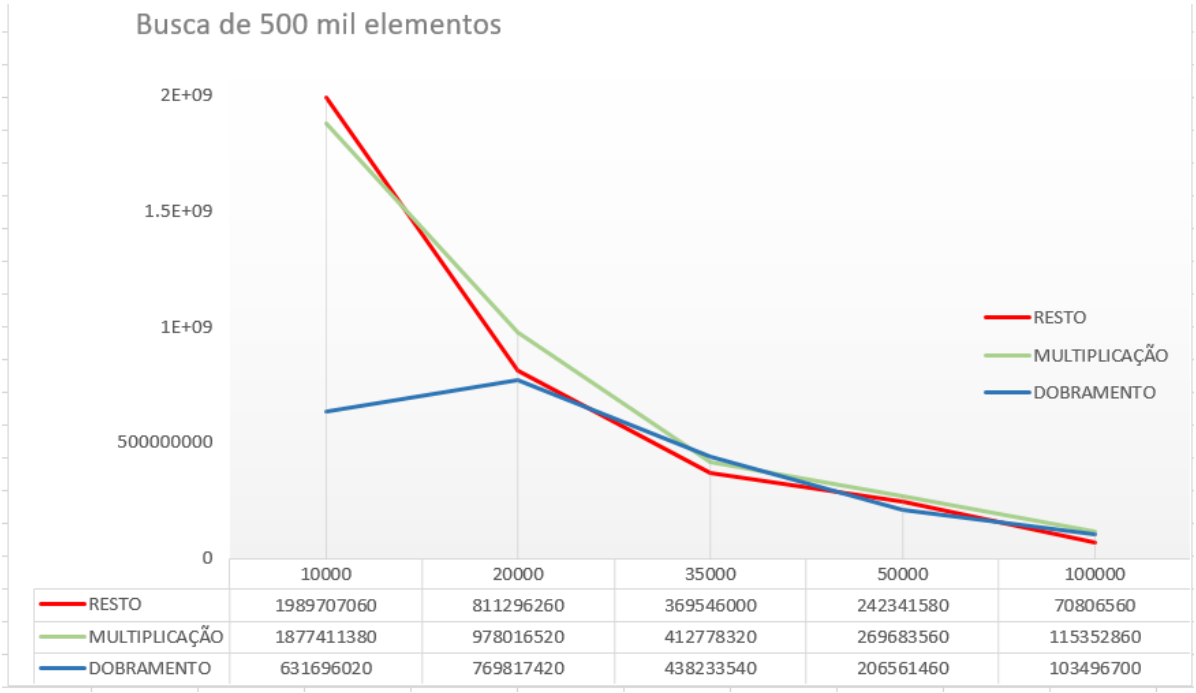


Figura 38. Representação gráfica das comparações de 5 milhões de elementos



	10000	20000	35000	50000	100000
RESTO	1989707060	811296260	369546000	242341580	70806560
MULTIPLICAÇÃO	1877411380	978016520	412778320	269683560	115352860
DOBRAMENTO	631696020	769817420	438233540	206561460	103496700

Figura 39. Representação gráfica do tempo de busca de 500 mil de elementos

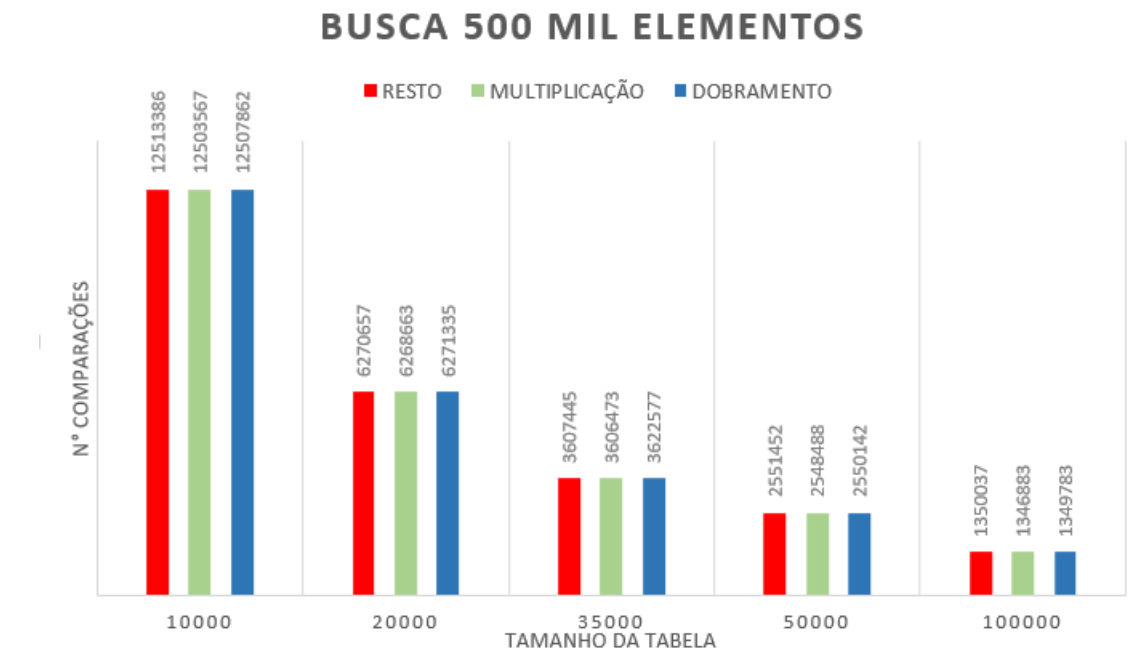
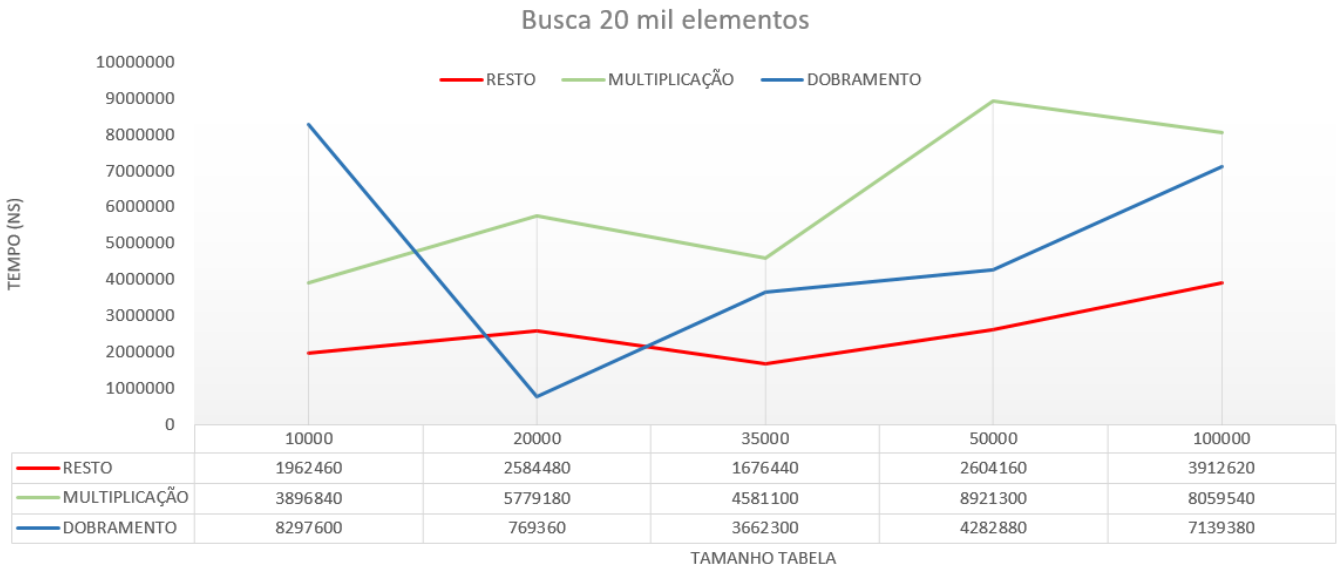


Figura 40. Representação gráfica das comparações de 500 mil de elementos



	10000	20000	35000	50000	100000
RESTO	1962460	2584480	1676440	2604160	3912620
MULTIPLICAÇÃO	3896840	5779180	4581100	8921300	8059540
DOBRAMENTO	8297600	769360	3662300	4282880	7139380

Figura 41. Representação gráfica da busca de 20 mil de elementos

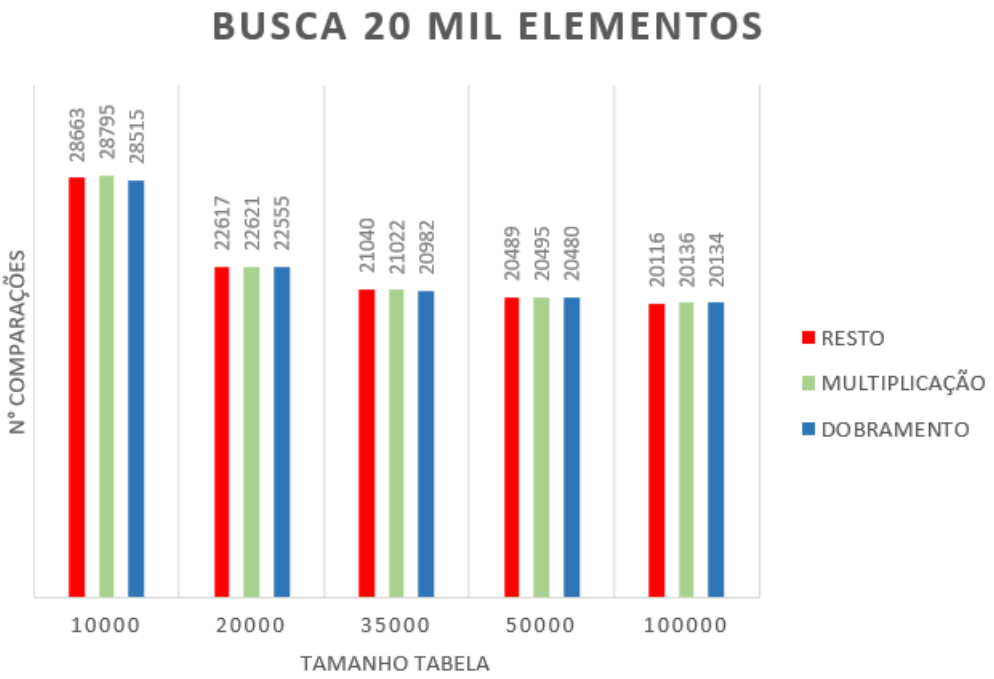


Figura 42. Representação gráfica das comparações de 20 mil de elementos

5. Conclusão

Primeiramente vale ressaltar que todos os resultados, análises e conclusões feitas a respeito dos métodos hash são referentes a implementação descrita no trabalho bem como os casos particulares que foram documentados no relatório. Sendo assim, todas as conclusões feitas são baseadas nesse cenário.

Como documentado nos resultados dos testes, todos os grupos de elementos foram inseridos e buscados em todos os tamanhos de tabela, os dados desses testes foram coletados e armazenados nas tabelas. Porém, para representação gráfica foram escolhidos grupos de elementos com diferentes dimensões, que vão de grandes quantidades, de tamanho intermediário e de pequenas quantidades, que correspondem aos grupos de 5 milhões, 500 mil e 20 mil elementos.

Para o grupos de 5 milhões de elementos, é possível observar no gráfico de inserção que o método hash com o desempenho mais constante é o de resto de divisão, onde seu tempo de execução apresenta uma queda uniforme e coerente na inserção desses valores nas tabelas de 10 a 100 mil elementos, já o método de dobramento apresenta muita distorção no tempo de execução onde seu desempenho até mesmo cai em comparação com tabelas menores, por fim, na inserção o método de multiplicação foi o método que apresentou o aumento mais brusco de desempenho conforme o tamanho da tabela aumentava, sendo o método com melhor desempenho para inserir grandes quantidades de elementos em tabelas maiores. Quanto as colisões, todos os métodos apresentaram valores semelhantes em cada tamanho de tabela, as variações estão coerentes com o tempo de execução, onde o método com melhor desempenho em determinado tamanho de tabela apresenta o menor número de colisões, assim como os piores desempenhos apresentam o maior número de colisões. Da mesma forma, nas análises de busca com 5 milhões de elementos, os resultados são semelhantes, apesar da perca de

desempenho no caso da tabela de 35 mil espaços o hash de resto apresentou resultados mais constantes em relação aos outros. Assim como na inserção, na busca o cenário foi o mesmo, para o hash de multiplicação, onde o desempenho teve um aumento brusco sendo o melhor na tabela de 100 mil elementos.

Para o grupo de 500 mil elementos o método hash com desempenho mais constante é o método de dobra, pois o mesmo apresenta pouca variação de tempo conforme o tamanho da tabela aumenta, sendo o mais rápido na tabela com o tamanho de 10000, em contrapartida, o método de divisão teve o aumento mais brusco de desempenho, sendo o mais lento na inserção da tabela de 10000 e o mais rápido na tabela de 100000. As colisões das inserções estão coerentes com os tempos de execução, onde as execuções mais rápidas possuem menos colisões em relação as mais demoradas. Da mesma forma, nas análises de busca os métodos hash possuem desempenho semelhante ao da inserção.

Já para o grupo de 20 mil elementos o comportamento do gráfico muda, a tendência é a perca de desempenho conforme o tamanho da tabela aumenta, o método de multiplicação ao contrario dos outros cenários teve uma piora constante no desempenho conforme a tabela aumenta. Sendo assim, o mais constante foi o hash de resto, tendo pouca variação de desempenho com o aumento da tabela, o mesmo se aplica para a busca com 20 mil elementos.

Baseado nos testes realizados pode se tirar as seguintes conclusões a respeito dos método de hashing implementados.

O método de Hash de Multiplicação apresenta um desempenho de inserção e busca significativamente melhor que os outros métodos em cenários onde o tamanho da tabela é proporcional ao número de elementos. Isso se prova nos testes pois o método teve os melhores desempenhos em grandes grupos de elementos com tabelas maiores e e pequenas quantidades de elementos com tabelas menores, com exceção de quando o tamanho da tabela é igual o número de elementos, onde todos os métodos apresentam queda de desempenho. Sendo assim o método de multiplicação não é indicado para grandes quantidades de elementos em pequenas tabelas e vice-versa, pois gera muitas colisões na inserção e muitas comparações na busca.

O método Hash de Resto da Divisão se mostrou ser o mais versátil dentre os três, por mais que em muitos casos não apresentou o melhor desempenho, ele apresentou resultados mais constantes e com pouca variação na maioria dos casos tanto de inserção quanto busca. Além disso, os resultados de tempo, colisões e comparações são satisfatórios e coerentes. Porém, vale ressaltar que seu desempenho nos testes de busca foram melhores que nos testes de inserção, se mostrando melhor para consulta.

O método de Hash de Dobramento implementado possui os resultados mais imprevisíveis dos testes, onde o mesmo ganha e perde desempenho constantemente se mostrando excelente em casos específicos e não tão bom em outros. Mas pode-se observar que ele possui uma tendência de ter melhor desempenho com grandes quantidades de dados em tabelas menores e pequenas quantidades de dados em tabelas maiores, tanto na inserção quanto na busca.

6. Referências

Netto, A. J. L. (2007) "Hashing."
Disponível em: http://netto.ufpel.edu.br/lib/exe/fetch.php?media=aed2:hashing.pdf

Página do Professor Aldo von Wangenheim sobre "Estruturas de Dados: Hashing".
Disponível em: https://www.inf.ufsc.br/~aldo.vw/estruturas/Hashing/

Universidade Estadual de Campinas (UNICAMP). "Tabela Hash".
Disponível em: https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node26.html

João Arthur B. M. "Estruturas de Dados - Hashtable".
Disponível em: https://joaoarthurbm.github.io/eda/posts/hashtable/

Backes, P. "Aula 07 - Tabela Hash."
Disponível em: https://www.facom.ufu.br/~backes/gsi011/Aula07-TabelaHash.pdf

Acervo Lima. "Método de Dobragem em Hash."
Disponível em: https://acervolima.com/metodo-de-dobragem-em-hash/

Vídeo no YouTube: "Vídeo sobre Hashing"
Disponível em: https://youtu.be/o0TXB3QPOWY?si=aDcCBs1ZA4S6ibTn

Instituto de Ciências Matemáticas e de Computação (ICMC) - USP. "Aula Hashing."
Disponível em: http://wiki.icmc.usp.br/images/a/af/Aula_hashing.pdf