Universidade Federal do ABC

Gabriel Zolla Juarez
RA: 11201721446

FUNÇÕES DE ESPALHAMENTO (HASH)

Repositório: https://github.com/gabezolla/Tabela-Hash-AED2

Santo André
2020

1. INTRODUÇÃO

As estruturas de dados são mecanismos importantes em quaisquer aplicações relacionadas à Computação, uma vez que elas têm a função de organizar e buscar os dados de forma otimizada dependendo do que a aplicação demanda, isto é, visando processamento mais rápido, menor uso de memória, entre outros. Para tal, existem estruturas de ordenação, como o QuickSort, amplamente utilizado na computação, bem como estruturas de organização como árvores, pilhas e filas. Neste meio, incluir-se-á o objeto de estudo do relatório em questão, que serão as tabelas Hash, denominadas também como funções de espalhamento.

As estruturas de dados são classificadas a partir da quantidade de operações necessárias para realizar determinada tarefa, seja ela a busca, ordenação, inserção e remoção dos dados. Para fazê-lo, é comumente feita a representação por meio de uma convenção, a letra O seguida do número de operações em parênteses. Uma busca feita em O(n), por exemplo, será necessária n operações, sendo n o número de elementos do vetor ou tabela.

As tabelas Hash surgiram justamente para resolver o problema de busca, já que um método mais rápido de busca do que os convencionais era buscado pelos estudiosos da teoria da computação. Tendo em vista que os métodos vigentes de busca sequencial, que operava em O(n), a busca binária em O(log2n), e as árvores balanceadas, por sua vez, em O(logn), ainda eram custosos, as funções de espalhamento com o seu acesso feito, em média, por O(1), isto é, em tempo constante na média, com o pior caso sendo O(n), tornou a implementação de estruturas como dicionários (frequentemente usados em linguagens como Python) muito menos custosos e mais rápidos.

2. DESENVOLVIMENTO

A implementação mais simplória das funções de espalhamento consistem em um array de *m-1* posições que armazenam elementos cujas chaves são determinadas segundo equações específicas, pois podem ser implementadas a partir do método da multiplicação, da divisão, que serão detalhados posteriormente, e outros.

Com isso, as tuplas de chave e valor vão sendo armazenadas no array. No entanto, tendo em vista que as posições do array são definidas através de equações, duas chaves distintas podem gerar o mesmo valor, e deveriam estar, portanto, em uma mesma posição no array. Esse processo é denominado de colisão, e, para solucioná-lo de forma a não excluir chaves de forma desnecessária, é necessário compreender uma outra estrutura de dados: a lista ligada. Estas funcionam sucintamente da seguinte forma: cada variável possui um endereço de memória e um valor específico para si. A lista ligada consiste em cada variável “apontar” para o endereço da próxima, formando uma lista dinâmica. Portanto,

quando ocorrer colisões na tabela hash, basta que a variável que será adicionada na posição em questão seja deslocada até o fim da lista e que a última variável aponte para o endereço de memória da mesma, como mostra a imagem a seguir:

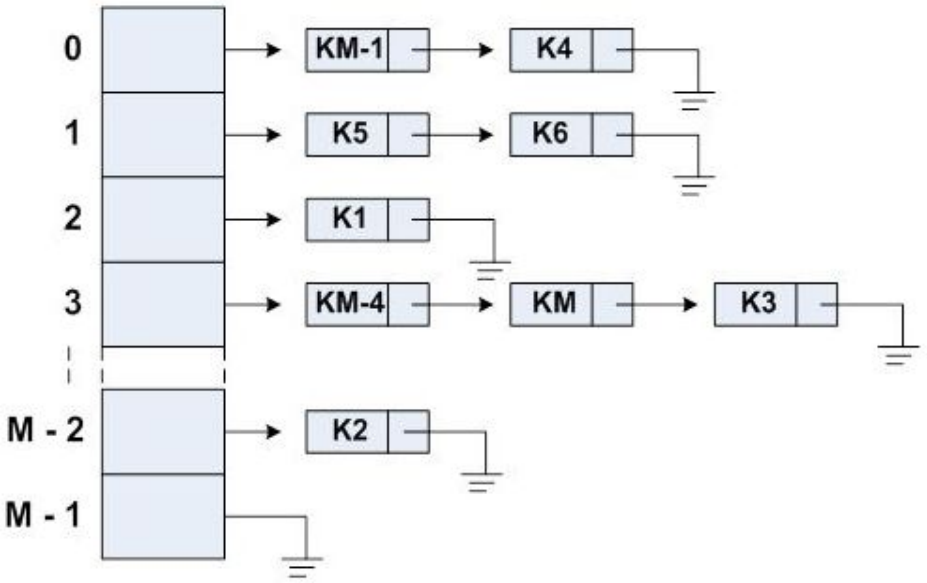

Figura 1: Tabela hash com encadeamento.

Quanto ao processo de determinação das posições que serão armazenadas no array, como dito anteriormente, existem dois processos: o método da divisão e o da multiplicação. O primeiro baseia-se na fórmula:

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

Sendo h(k) a chave hash, k a chave da tupla em questão, e m um valor arbitrário que corresponde ao tamanho do array e, dependendo da escolha do mesmo, existe melhor ou pior otimização. A operação %, por sua vez, é o módulo, isto é, o resto da divisão. Os valores de m mais viáveis são números primos distantes de potências de dois, cujo motivo não é tão trivial e sua abordagem não é de interesse para a compreensão do tema. Com isso, há uma melhor distribuição de chaves hash, otimizando, em média, o processo. Vale ressaltar que é ideal que o valor de m seja sempre positivo, de forma a evitar valores de chave negativos. O método da multiplicação funciona de forma semelhante, porém, com algumas ressalvas:

$$h(k) = m.(kA\%1)$$

Sendo a constante A uma constante arbitrária entre 0 e 1, que será fundamental para a melhor implementação do processo, sendo alguns valores melhores do que outros, contudo, aquele que segue a razão áurea é interessante, como A= 0,6180339887...

3. METODOLOGIA E RESULTADOS

Para estudar as funções de espalhamento, serão implementados ambos os métodos de chaveamento hash: tanto o de multiplicação quanto o de divisão, bem como a estrutura completa de uma tabela hash por meio da linguagem Java, sendo necessário, principalmente, o constante uso de sua orientação a objeto, com classes e atributos.

A arquitetura da aplicação é simples: as tuplas da tabela Hash baseiam-se na classe Tuple, que consiste em um objeto que contém três parâmetros: a chave (key), o valor (value), e a variável next, que servirá para apontar para o endereço de memória do próximo valor, estabelecendo a lista ligada. As funções hashMapMultiply e hashMapDivision, por sua vez, implementam os dois métodos de multiplicação e divisão, respectivamente, bem como contêm o array que armazenará as tuplas que serão recebidas. O App.java, por fim, possui a aplicação propriamente dita, e envolve um trecho de código gerador de números aleatórios (a partir das funções da biblioteca Math, .random() e .round()), para gerar chaves inteiras aleatórias e assim estudarmos o comportamento da função, bem como um gerador de Strings para fazer o papel do valor na tupla. Existe, também, uma interatividade com o usuário, caso ele queira construir a sua tabela Hash, se localiza no código comentado dentro do laço *for*. Por fim, há um método de busca de chaves na tabela a partir de entradas do usuário, também comentado a princípio, tendo em vista que não será necessário para o estudo em questão.

A partir das estruturas implementadas, serão gerados dados e gráficos de cenários variados para estudar as funções de espalhamento, bem como averiguar como os valores das constantes arbitrárias citadas acima são cruciais na otimização da estrutura.

O repositório em questão pode ser encontrado no seguinte link: https://github.com/gabezolla/Tabela-Hash-AED2. Para utilizar a aplicação, basta colocar na pasta o arquivo desejado e utilizar as classes Tuple e hashMap da forma descrita anteriormente, ou fazer a importação do pacote manualmente por meio do *import* do Java.

3.1. Método da Divisão

Para analisar os casos referentes ao método da divisão, é necessário estudar os diferentes valores da constante m. Como visto anteriormente, os valores mais adequados são números primos cujo valor distam de potências de dois.

Primeiramente, observar-se-á o caso de m=12, com valores de chave variando de 1 a 100. Para fazê-lo, utilizamos a função que gera números aleatórios, explicada anteriormente, e estudamos os casos de colisão. A partir do Python e da biblioteca matplotlib, fora possível traçar o histograma das ocorrências de colisões em cada posição do array.

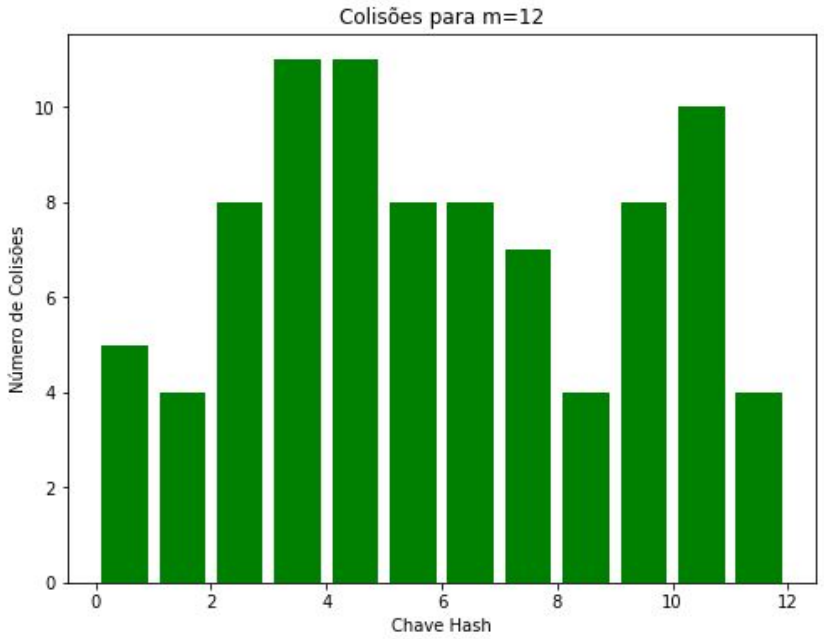

Gráfico 1: Histograma de colisões para m=12.

Analisando o histograma em questão que representa um certo padrão apresentado por esse valor de m, fora possível identificar que na chave 3 principalmente, ocorreu grande número de colisões, enquanto em outras como as chaves 0, 1 e 12 as colisões foram menores, demonstrando um certo desbalanceamento que prejudica o funcionamento médio da aplicação. O número de colisões total, por sua vez, fora de 88, enquanto as colisões referentes ao valor 3 foram de 11.

Agora, analisar-se-á o histograma quando m=11, um número primo, para o mesmo número de chave que o experimento anterior.

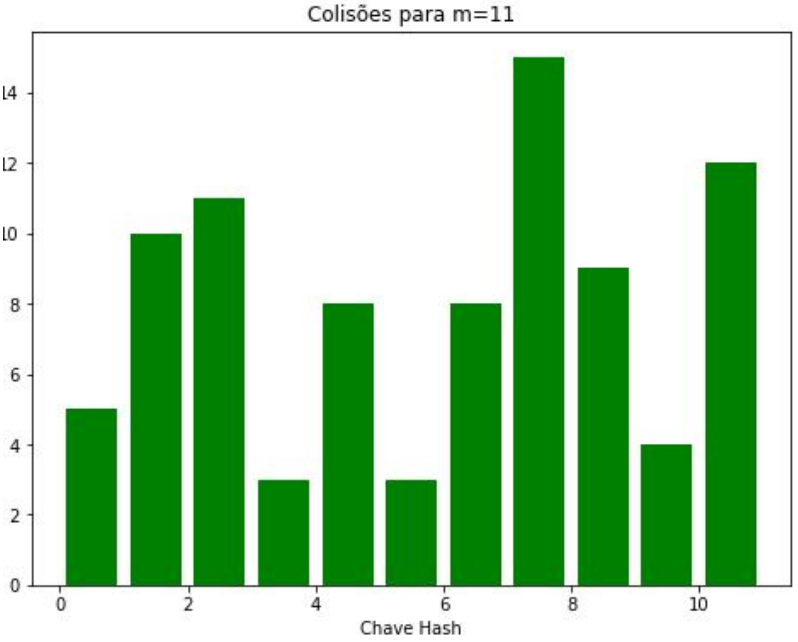

Gráfico 2: Histograma de colisões para m=11.

Após algumas repetições, o padrão que se estabelece é o acima, que diferentemente do primeiro, encontra-se mais balanceado, apesar de um grande número de colisões no valor de chave 8, que seria o pior caso da estrutura, os demais teriam um tempo de acesso médio menor. Na chave 3, por sua vez, ocorreu um padrão menor, mais mediano, diferentemente do caso anterior.

Todavia, nos casos acima cuja quantidade de chaves possíveis são relativamente pequenas, a importância da escolha adequada da constante não é tão clara. Para averiguar mais precisamente esse fato, observar-se-á um caso com chaves cujo valor possível giram em torno de 1 a 10000, com m=97.

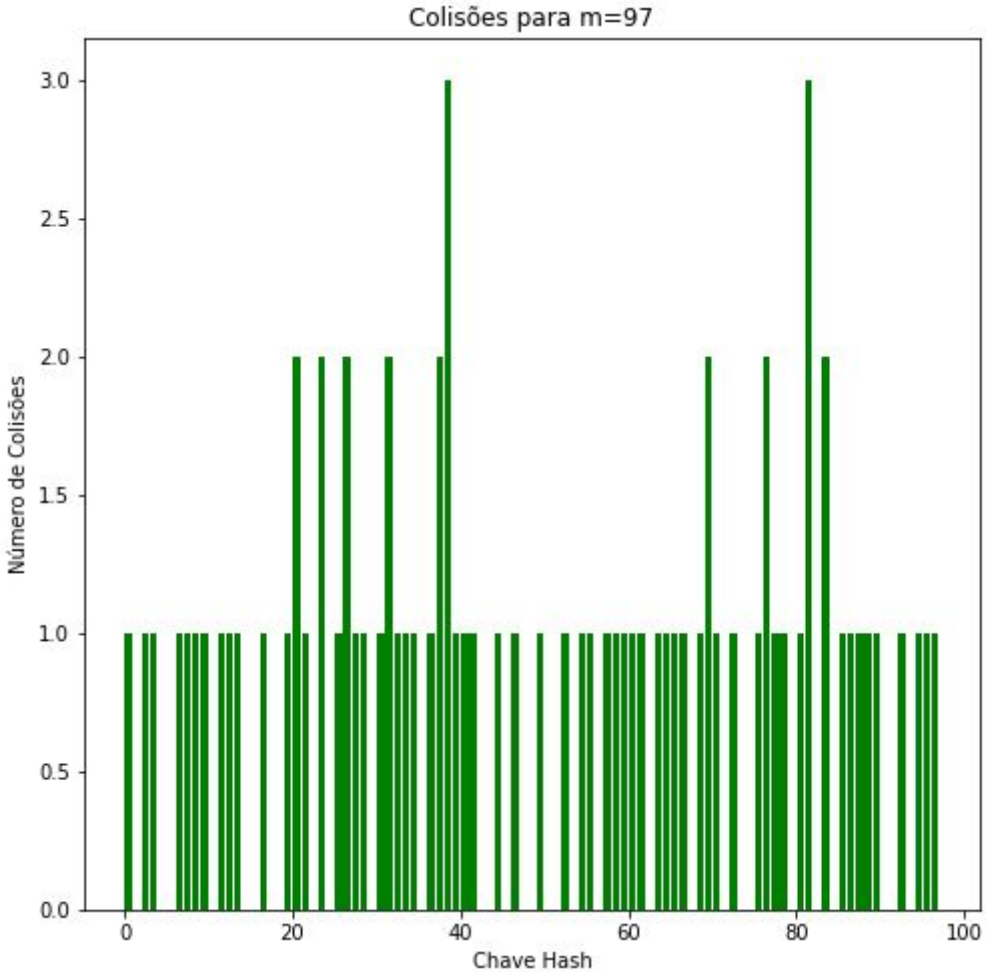

Gráfico 3: Histograma de colisões para m=97 e chaves em torno de 1 a 10000.

Percebe-se, nesse caso, que o número de colisões correspondente a 1 é maioria no histograma, o que reitera o tempo de acesso médio de O(1) dito anteriormente O total de colisões fora de 35, muito inferior aos casos anteriores. Aumentando o número de casos, pode-se estudar o comportamento da função de espalhamento para casos muito maiores, como com a inserção de 10000 chaves:

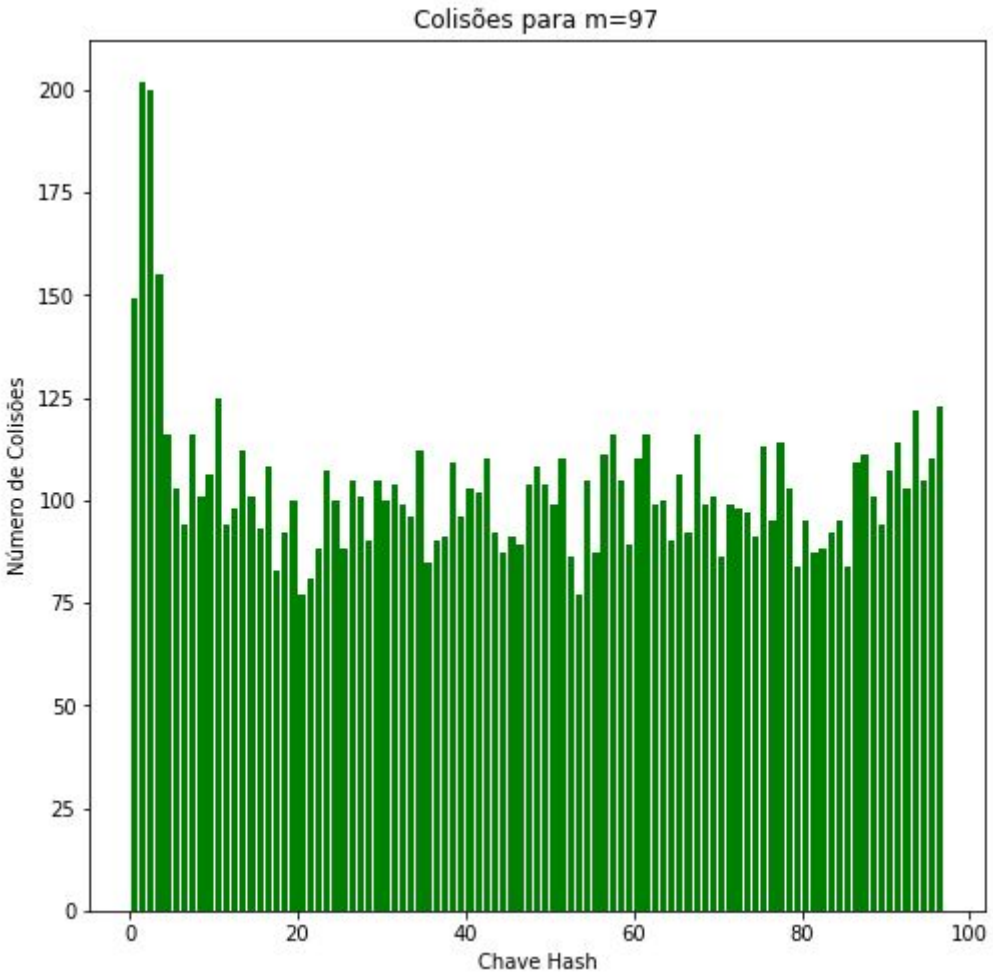

Gráfico 4: Histograma de colisões para m=97 com 10000 entradas.

Reitera-se, portanto, embora um número maior de colisões nas chaves iniciais, que se deve basicamente pela geração randômica de chaves, o comportamento médio de chaves com valores de colisões fica próximo de 100, mesmo com um espaço amostral de 9903 colisões, o que demonstra o tempo de acesso médio constante para boa parte dos casos.

3.2. Método da Multiplicação

Para o método da multiplicação, por sua vez, serão analisados dois casos de constante A para o mesmo valor de m: um caso de A = 0,62 arbitrário, e outro valor de A = 0,61803398875, correspondente à razão áurea, ambos com m=200. Utilizar-se-á um espaço amostral de 10000 chaves, tal como o último exemplo anterior, para analisar o comportamento das funções.

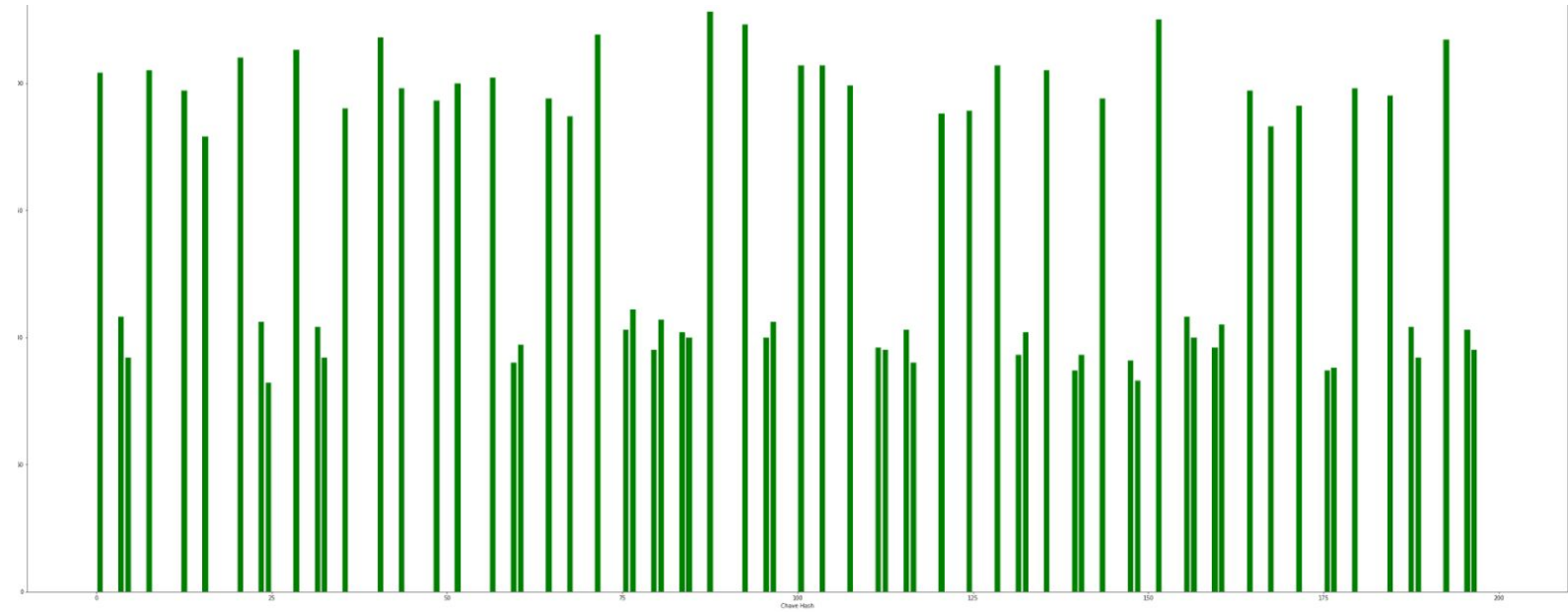

Gráfico 5: Histograma de colisões para m=200 e A = 0,62

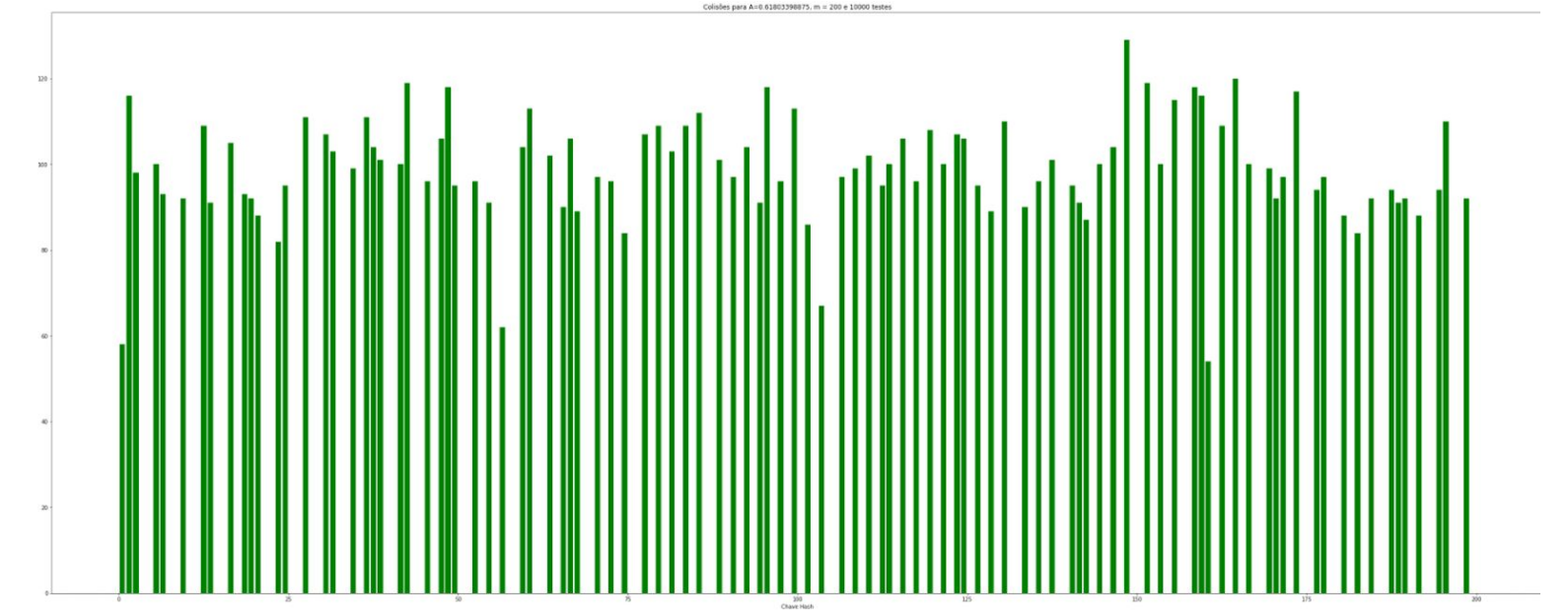

Gráfico 6: Histograma de colisões para m=200 e A = 0,61803398875.

Comparando ambos os gráficos, para um número de colisões muito próximo (9932 para o primeiro e 9899 para o segundo), apesar de um número ligeiramente maior de colisões, nota-se um histograma muito melhor distribuído quando a constante A corresponde à razão áurea, reduzindo a razão número de colisão por chave, o que diminui o tempo médio de acesso perante ao histograma cuja constante não obedece tal razão, demonstrando a importância da escolha da constante para determinada aplicação. Vale ressaltar que o pico de número de colisões (eixo y) por chave no primeiro gráfico passa os 200, enquanto no segundo fica em torno de 120. Os gráficos serão mostrados de forma ampliada no ***Apêndice.***

4. CONCLUSÃO

As tabelas Hash são importantes estruturas para organizar dados e facilitar busca e acesso às tuplas que ali estão inseridas. Para tal, é necessário uma implementação adequada, com escolha de constantes ideais dependendo do seu método de cálculo de chave hash. Vale ressaltar que a aplicação aqui desenvolvida é simples e uma implementação mais robusta e quiçá com outros métodos de cálculo da chave hash que não sejam os métodos abordados no relatório em questão podem tornar o tempo de acesso bem como o processamento muito inferiores. Contudo, percebe-se que o tempo de acesso das funções de espalhamento, correspondentes à O(1), isto é, tempo constante, são realidade e imensamente funcionais.

5. APÊNDICE

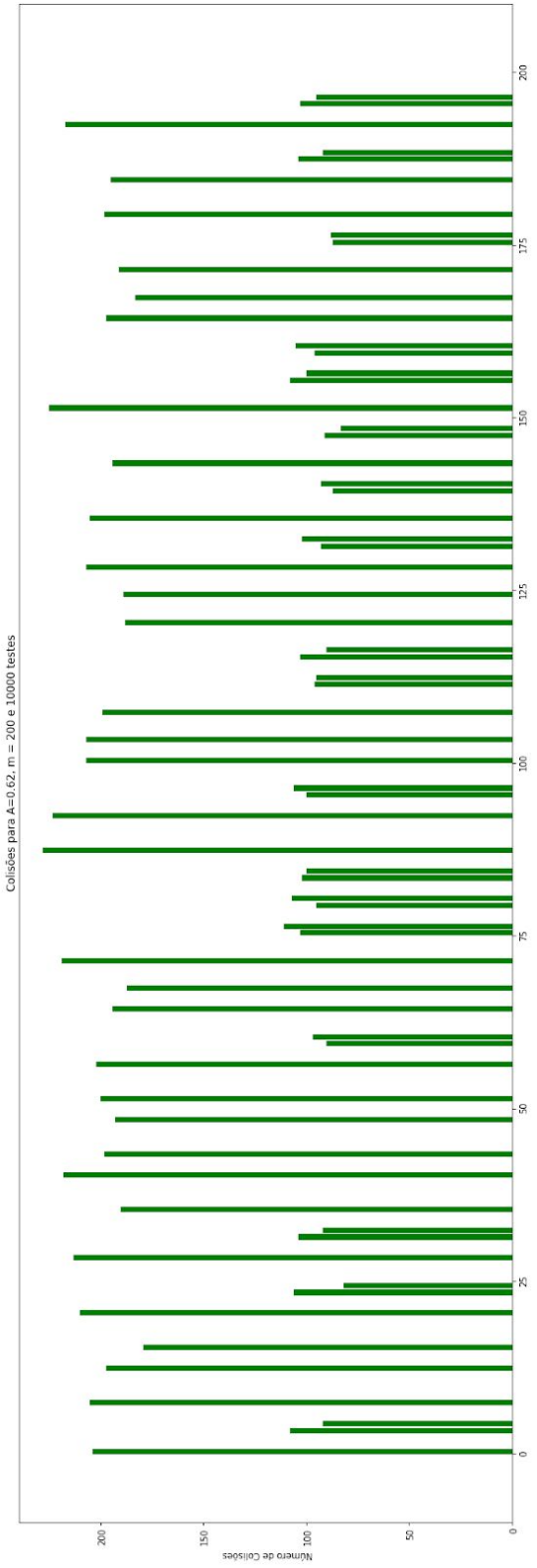

Gráfico 5: Ampliado

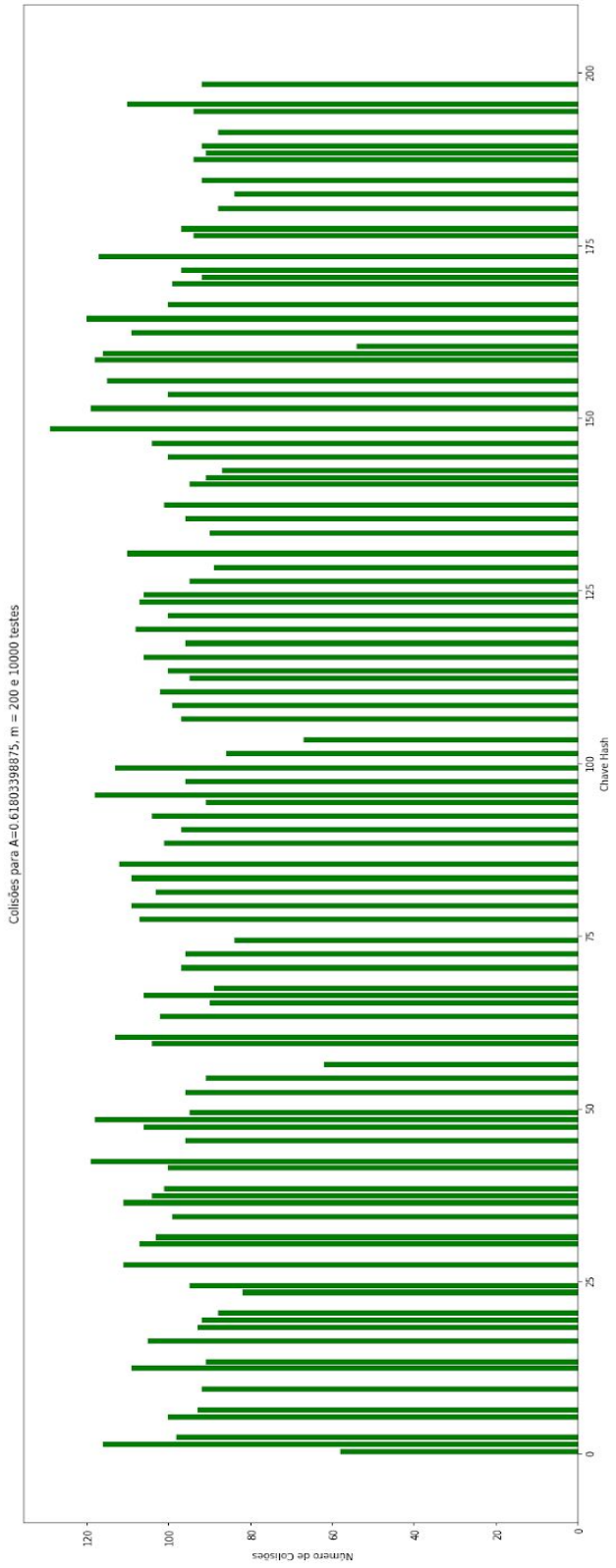

Gráfico 6: Ampliado