Eduardo Eugênio de Souza - 2310822
Luiz Felipe Neves Batista – 2311024

Objetivo

Inserir CPFs em uma tabela *hash* usando uma função que tenha complexidade melhor do que log N.

Estrutura do programa

O programa é composto por 3 funções, além da main: hash_function, busca_no_hash e solve_collision.

A hash_function implementa uma função de hash simples, que utiliza manipulação de bits e também o valor da tentativa atual para dispersar os *hashes* ao longo da tabela.

A busca_no_hash busca um dado CPF em uma tabela hash, usando a hash_function. Simplesmente calculamos o hash referente a um CPF repetidas vezes, aumentando o número de tentativas, até que encontremos no vetor o CPF que estamos buscando.

A solve_collision é chamada quando, após uma tentativa falha de inserção, ocorre uma colisão. Ela se encarrega de chamar repetidamente a hash_function enquanto houver colisões secundárias que impeçam de inserir. Ela também atualiza o vetor collision_count para que ele contenha mais uma colisão para a quantidade de chaves inseridas até o momento.

Solução

Criamos a tabela *hash* como um vetor de inteiros longos de 1249 posições, com o intuito de ter um load factor de 80%. Também temos um vetor que guarda onde ocorreram as colisões e quantas colisões ocorreram até que o valor seja inserido naquele endereço de forma a proporcionar visualização posterior.

A *hash_table* é inicializada com -1 em todas as posições, para indicar um lugar vazio. Primeiramente, abrimos o arquivo. Depois, lemos um CPF e chamamos a hash_function passando 0 tentativas, para tentar inserir o CPF pela primeira vez. Se o lugar do vetor não estiver ocupado, inserimos. Se o lugar do vetor já estiver ocupado, chamamos a função solve_collision, que irá inserir o número na *hash table*, ainda que sejam necessárias várias tentativas. Por fim, sempre que inserimos um CPF, incrementamos a variável que conta a quantidade de chaves inseridas até o momento.

Também exibimos, ao final do código, algumas informações pedidas no enunciado: “Indique o número de colisões e o número de posições vazias.”. Também fizemos um teste da função de busca.

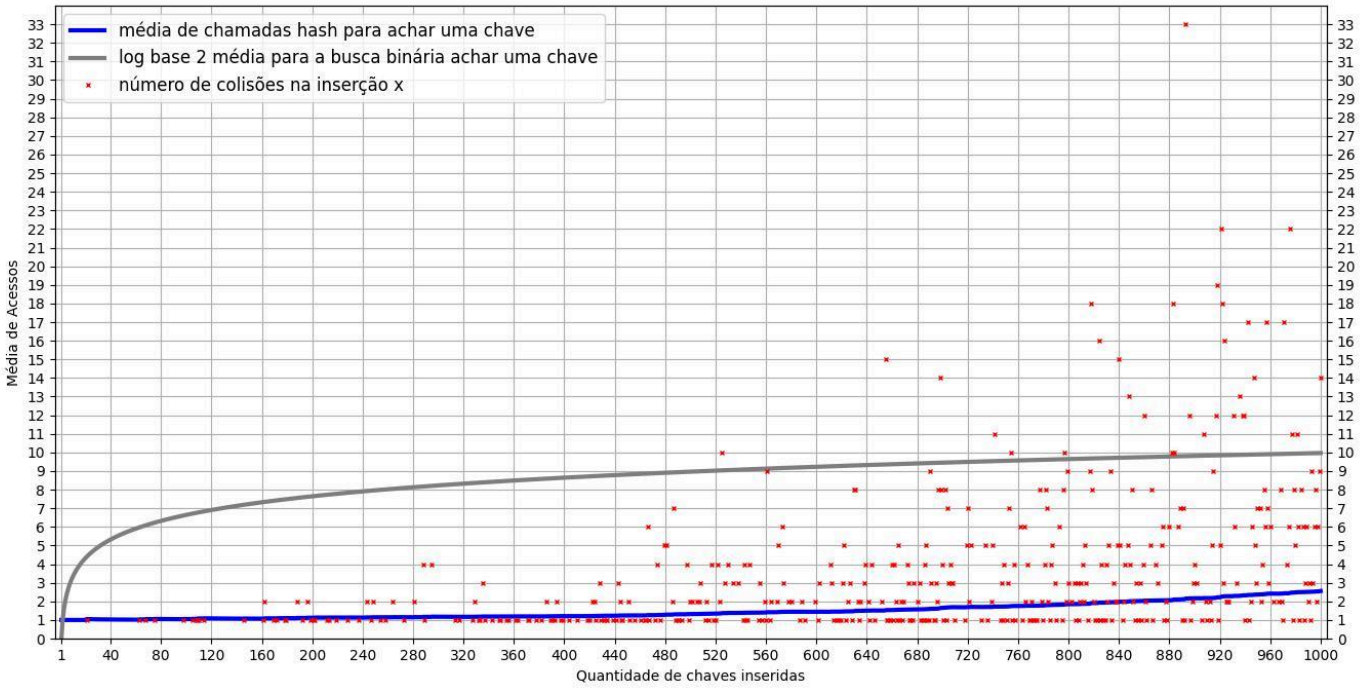
Observações e conclusões

O *output* do programa é o seguinte:

Houve 1554 colisões no total

Sobraram 249 posições vazias no vetor
buscando 69172073900: 795, 69172073900

Percebemos que a função hash tem uma eficiência média muito boa, pois podemos ver no gráfico a seguir, que em média, para achar uma chave com a tabela hash leva muito menos



acessos do que com uma busca binária em outra estrutura. Além disso, pode-se ver neste gráfico exatamente quando foi que a inserção na tabela hash discorreu em um alto número de colisões individualmente, o que seriam os casos onde demoraria mais para se achar a chave nesta tabela hash do que em uma busca binária.

Com base nesses fatos podemos concluir que a tabela hash é uma ótima escolha para se guardar qualquer tipo de dado, pois com ela você tem um grau de liberdade de poder escolher a sua função de hash, o que permite que você consiga adaptar ela aos seus dados. Além disso vale ressaltar que por mais que a curva de média de chamadas à função hash tenha uma inclinação maior do que o log2, ela sempre estará abaixo do log pois quando se forem inserir mais dados, será necessário aumentar o tamanho do vetor que guarda os dados, pois precisa-se tentar manter o load factor, que está em torno de 80%.