Identificação: Pedro Mesquita Maia – 2312664

Descrição: O objetivo do trabalho foi implementar uma tabela de hash para gerenciar e armazenar números de CPF (Cadastro de Pessoa Física), utilizando funções para inserir novos CPFs e calcular estatísticas sobre o desempenho da tabela. Além disso, foi solicitado o tratamento de colisões e a verificação da ocupação das posições na tabela.

Estrutura: Foram criadas várias funções, além da função principal (main), para realizar as operações necessárias: loadCPFs, hash, hash2, insertCPF e displayStatistics. A função loadCPFs é responsável por ler os CPFs a partir de um arquivo, enquanto hash e hash2 geram valores de hash para os CPFs, permitindo a inserção eficiente na tabela. A função insertCPF insere um CPF na tabela, lidando com colisões, e a função displayStatistics apresenta as estatísticas relacionadas ao número de inserções, colisões e posições vazias.

Solução: Foram declaradas funções para criar a tabela de hash, inserir elementos, e exibir estatísticas de desempenho. A função hash é responsável por gerar um índice de hash para o CPF, enquanto hash2 fornece um valor adicional para resolver colisões utilizando o método de endereçamento aberto. A função insertCPF insere o CPF na tabela, verificando se a posição já está ocupada e, caso positivo, realiza a busca pela próxima posição disponível. A função loadCPFs carrega os CPFs de um arquivo e chama a função de inserção para cada CPF lido.

Após a leitura dos CPFs e a inserção na tabela, a função displayStatistics é utilizada para imprimir as estatísticas das operações realizadas, como o número de colisões e o número de posições ainda vazias na tabela. O uso de uma tabela de hash com um tamanho fixo (TABLE_SIZE = 2003) e a escolha de números primos ajudam a minimizar as colisões e garantir uma distribuição uniforme dos valores de hash.

Avaliação da Complexidade: A complexidade média da operação de inserção em uma tabela de hash é O(1), ou seja, tempo constante, desde que o número de colisões seja baixo e a função hash distribua os dados uniformemente. No entanto, no pior caso, onde todas as entradas podem ser mapeadas para a mesma posição, a complexidade pode se degradar para O(N), onde N é o número de elementos na tabela. Em comparação, a complexidade O(log N) de estruturas de dados como árvores binárias de busca (BST) oferece um desempenho logarítmico, que é eficiente, mas ainda assim mais lento que uma operação de hash em média. Portanto, quando a tabela de hash é bem projetada e gerenciada, ela geralmente resulta em um desempenho melhor que uma BST para operações de inserção e busca.

Observações: Durante o desenvolvimento, percebi que era importante monitorar as colisões e as posições vazias da tabela, o que não havia sido considerado inicialmente. A inclusão da função displayStatistics foi crucial para entender a eficiência do algoritmo. Além disso, o teste com diferentes arquivos de entrada ajudou a perceber a necessidade de ajustes nas funções de hash para garantir uma melhor distribuição dos CPFs. O código foi testado e funcionou corretamente, demonstrando a eficácia da abordagem adotada.
Para compilar, foi utilizado o comando gcc -Wall -o main main.c.

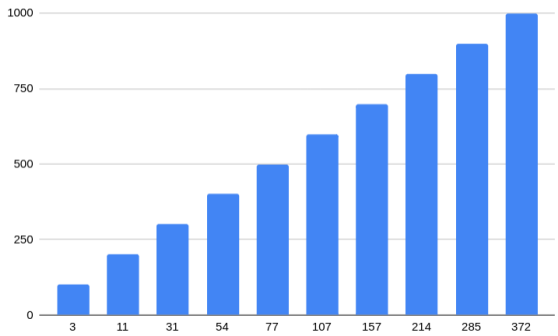

Gráfico: eixo y é a quantidade de chaves inseridas e o eixo x é a quantidade de colisões