UNIVERSIDADE FEDERAL DO PIAUÍ - CAMPUS SHNB
SISTEMAS DE INFORMAÇÃO
ALGORITMOS E PROGRAMAÇÃO II
Ana Caroline de Almeida Alves
Mateus Pinto Garcia

Trabalho Prático

1 Introdução

As funções de Hashing são utilizadas para melhorar a recuperação de registros de dados, para a validação de dados ("somas de verificação") e para criptografia. No caso de consultas, o código *hash* é usado como um índice em uma tabela *hash* que contém um ponteiro para o registro de dados.

A tabela *hash* ou tabela de dispersão é uma estrutura de dados utilizada para a dispersão de dados, constituindo-se numa abordagem comum para o problema de armazenamento. Esta estrutura é bastante utilizada para armazenar dados de grande volume, como arquivo de dados. Seu objetivo é, a partir de uma chave simples, realizar uma busca rápida e obter o valor desejado. Tal busca se dá a partir de registros, onde, cada registro possui um campo especial chamado chave. A posição onde o registro será posicionado pode ser calculada com uma função de dispersão, nomeada Função *hash*.

Quando uma tabela *hash* está alocada alguns espaços contém registros válidos e outros estão vazios. A chave deve, de alguma forma ser convertida em um índice de vetor, para que a inserção de um novo registro seja bem sucedida. O índice é chamado de valor *hash* da chave. Após o cálculo o valor *hash* obtido deve ser utilizados para a localização do novo registro.

Em tabelas *hash* pode ocorrer o problema de calcular o mesmo índice para duas chaves diferentes. Tal problema é conhecido como colisões. Por conta disso, a função deve ser projetada para evitar ao máximo a ocorrência de colisões. Por mais bem projetada que seja a função de dispersão, sempre haverá colisões. A estrutura de dispersão utiliza mecanismos para tratar as colisões, que dependem de características da tabela usada. Assim, os mecanismos mais comuns para tratamento de colisões são: endereçamento aberto e encadeamento. Neste trabalho as colisões serão tratadas usando o método de endereçamento aberto, ou seja, tentar inserir a chave na tabela na primeira posição livre com a seguinte função de dispersão:

$$\textbf{(Hash(key) + j 2 + 23} * \textbf{j) mod 101, para j = 1, ..., 19.}$$

Desse modo, partir de 20 ou mais colisões, será assumido que uma operação de inserção não pode ser executada.

Além de inseridos, os registros também podem ser removidos, porém o local não pode ser visto como uma célula vazia, pois, pode atrapalhar nas busca. Diante disso o local deve ser marcado, de modo que na busca seja possível ter conhecimento que havia um registro no determinado local.

Diante do exposto, será desenvolvido um algoritmo que calcule o resultado do espalhamento de dados numa tabela de 101 espalhamentos. Assim, serão adicionadas chaves que serão no formato strings com comprimentos de no máximo 15 letras.

2 Solução Proposta

Ao executar as operações mencionadas abaixo foi definido o procedimento int Hash(string key). Para o desenvolvimento do trabalho, a implementação da hash possuirá as seguintes operações:

- Inserir uma nova chave na tabela hash: Ao ser inserida uma chave foi definido o procedimento int Hash(string key), onde foi realizado um cálculo para a definição um local pré determinado para o armazenamento da chave;
- Buscar o índice do elemento definido pela chave: A melhor maneira de implementar a busca, é desenvolver a tabela de forma que cada elemento seja adicionado numa posição pré-determinada. Tal posição é obtida aplicando-se ao elemento uma função (função de hash) que devolve a sua posição na tabela. Então para buscar, deve-se verificar se o elemento realmente está nesta posição.
- Excluir uma chave da tabela: Marcando a posição na tabela como vazia, foram ignoradas chaves inexistentes na tabela.

Um problema bastante corriqueiro em tabelas hash, são elementos com o valor da função hash iguais. Tal problema é conhecido como colisões. Para solucionar o problema das colisões foi colocado o elemento na primeira posição livre seguinte, além de considerar a tabela circular (o elemento seguinte ao último a[n-1] é o primeiro a[0]). Isso foi aplicado na inserção de novos elementos e na busca. No pior caso, no qual a lista está com N elementos ocupados, será necessário percorrer todos os elementos antes encontrar o elemento ou conclui-se que ele não está na tabela. O algoritmo do tratamento de colisões possui complexidade O(N), pelo fato de precisar percorrer toda a tabela para esse tratamento.

O desenvolvimento da tarefa proposta se deu na linguagem de programação Python. Esta linguagem é de alto nível, interpretada por script. Python possui propósito geral de alto nível, multi paradigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedural.

Quanto a modularização, o algoritmo está dividido nas seguintes funções:

- **def Inicializa (numeroDeSlots):** nesta função é criada a tabela com base no parâmetro dos slots da tabela;
- **def Hash(Chave):** Soma o ASCII dos caracteres da String;
- **def MakeHash(Chave):** Nesta função é calculado o hash com base na função anterior(def **Hash(Chave)**);
- **def ArquivoSaida(Saida):** Nesta função é criado um arquivo com o número de chaves na tabela após inserções e remoções [primeira linha] índice: key [ordenado por índices];
- **def Pesquisar(Chave):** Na referida função foi desenvolvido um algoritmo de busca onde o índice do elemento é definido pela chave;
- **def Remover(Chave):** Esta função trata da remoção de um uma chave com base na sua existência;
- **def Inserir(Chave):** Nesta função foi realizado um cálculo para a definir um local pré determinado para a inserção da chave;
- **def TrataColisoes(Chave):** A referida função realiza o tratamento de colisões com base no método de endereçamento aberto, ou seja, tentar inserir a chave na tabela na primeira posição livre. Caso não seja possível inserir, então é recalculado a hash e é tentado novamente.

3 Análise de Desempenho

A presente seção trata da análise de desempenho do trabalho. Serão discutidos o melhor, médio e pior caso de execução do programa proposto. A complexidade de tempo de um algoritmo é comumente expressada usando a notação big O, que exclui coeficientes e termos de baixa ordem.

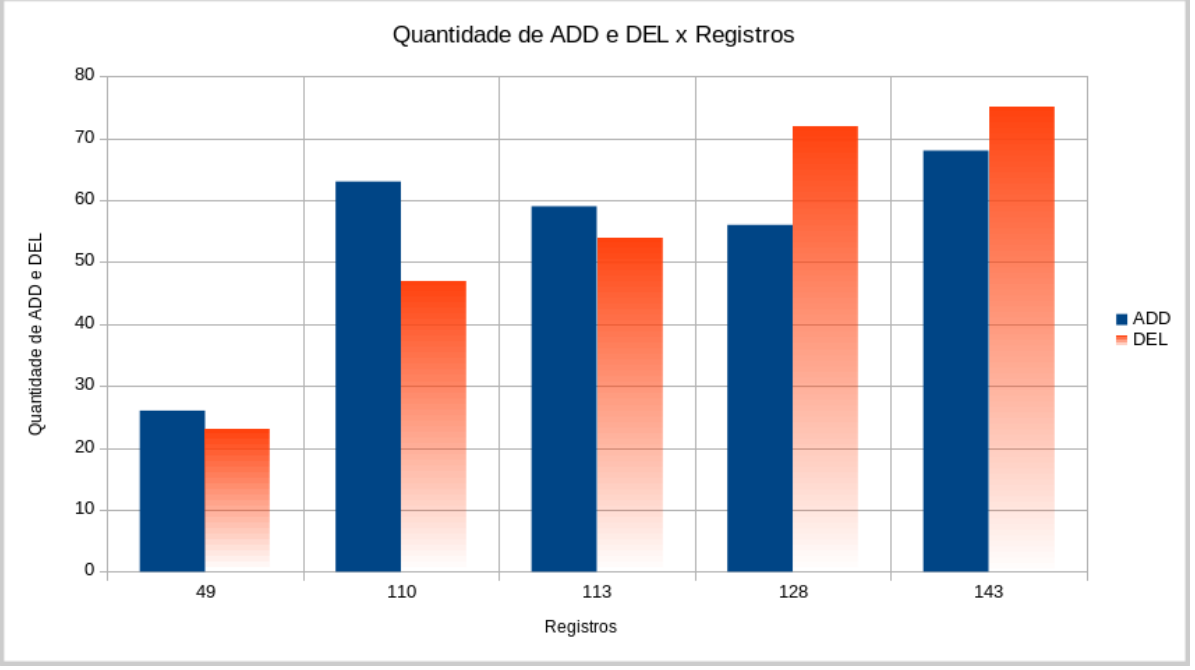
Considerando que através de uma função h(n) uma determinada chave é transformada em um endereço da tabela, a complexidade média por operação da *hash* é O(1). No pior caso sua complexidade é O(n).

A grande vantagem na utilização da tabela *hash* está no desempenho, enquanto a busca linear tem complexidade temporal O(n) e a busca binária tem complexidade O(logN), o tempo de busca na tabela *hash* é praticamente independente do número de chaves armazenadas na tabela,

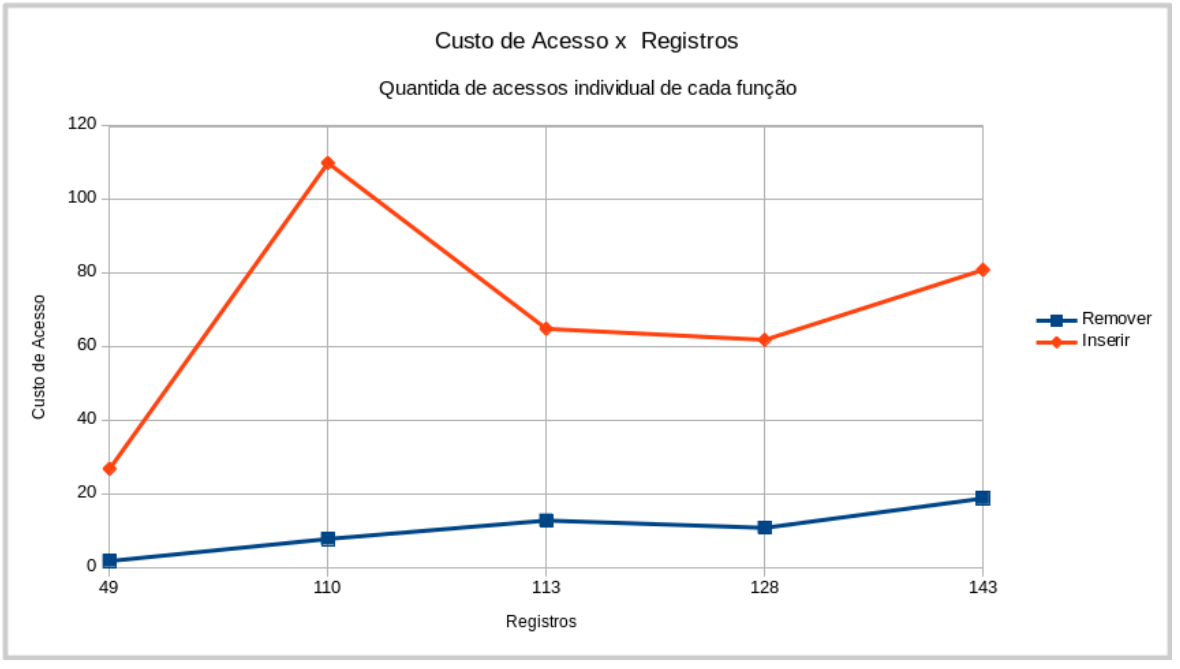
ou seja, tem complexidade temporal O(1) . Aplicando a função *hash* no momento de armazenar e no momento de buscar a chave, a busca pode se restringir diretamente àquela posição da tabela gerada pela função.

Nos gráficos abaixo temos alguns dos resultados após os testes de entrada de diferentes tamanhos de registros para uma tabela *hash* de tamanho 101 (101 posições)

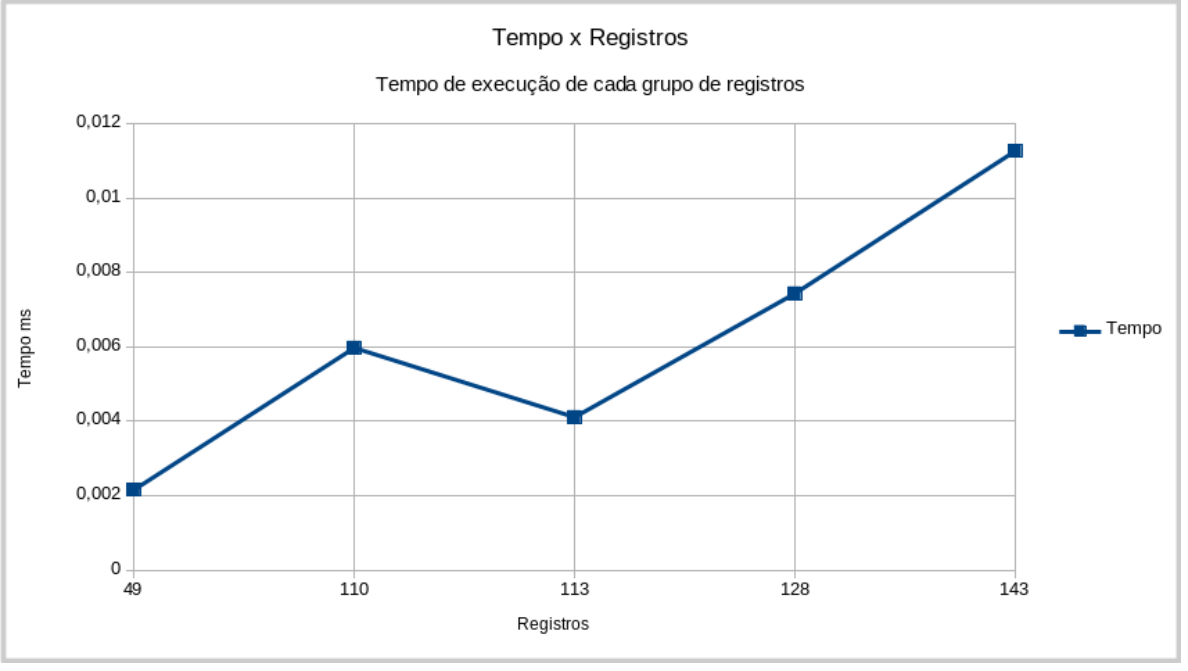
Neste gráfico temos os resultados da quantidade de registros adicionados e deletados em relação à quantidade de registros do arquivo de entrada.



Foi testado o tempo do custo de acesso em relação à quantidade de registros. O teste foi feito tanto na função de remoção quanto na função de inserir. Os testes foram feitos a partir de registros entre 49 e 143. O resultado se deu conforme a tabela abaixo.

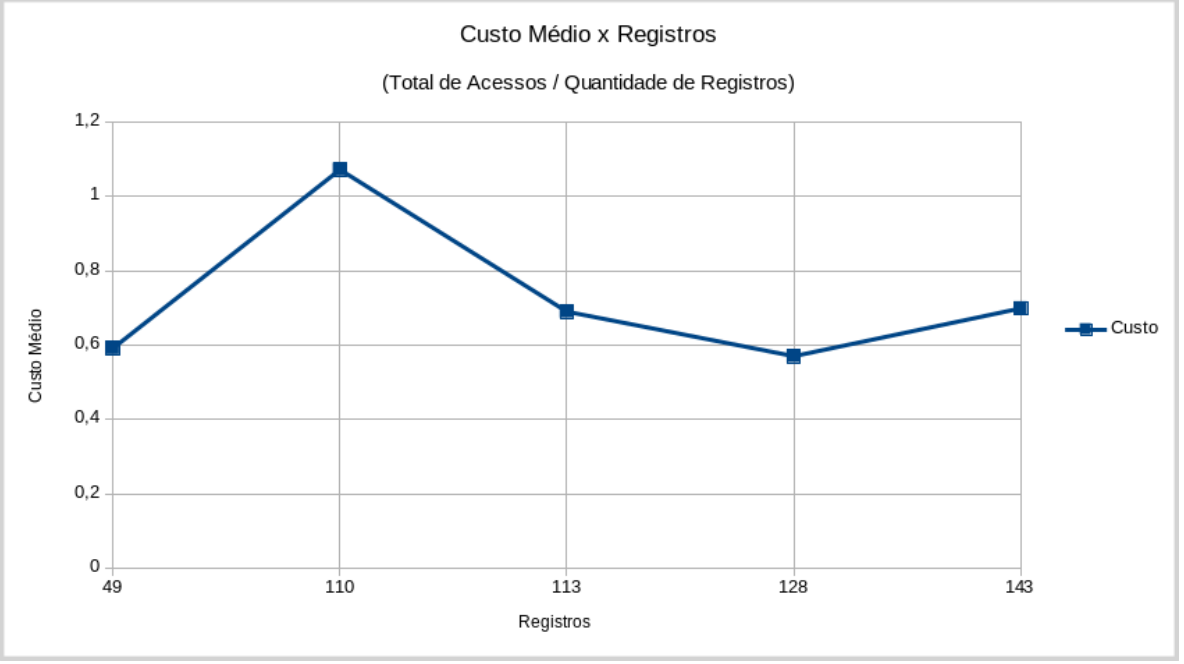


Também foi feito o teste do custo médio do tempo de execução em relação à quantidade de registros, onde o tempo foi dado em segundos, e a quantidade de registros foi entre 49 e 143.



No custo médio foi verificado a quantidade de vezes que foi necessário acessar a tabela *hash*, ou seja efetuar uma busca, seja para inserir ou mesmo remover. O calculo se deu por:

CustoTotal = Acessos de Remover(Chave) + Acessos de Inserir(Chave) / Quantidade de registros.



4 Conclusão

Por meio do desenvolvimento deste trabalho pudemos assimilar o funcionamento da estrutura de dados *hash*, no qual possui como objetivo espalhar dados. Esta estrutura é utilizada para armazenar dados de grande volume, como arquivo de dado, sendo muito eficiente, pois a posição de onde o registro será posicionado pode ser facilmente calculado com uma função de dispersão, nomeada Função *hash*.