Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais – Campus Barbacena
Núcleo de Informática
Curso Superior de Tecnologia em Sistemas para Internet

Trabalho Prático de Estruturas de Dados II

Ramon Giovane Dias Rosa

Trabalho prático ministrado no REI que consiste na simulação de um processo eleitoral. O objetivo do trabalho é realizar operações semelhantes a tal processo utilizando estruturas de dados rápidas, que processem dezenas, centenas de milhares de instruções em milissegundos.

Barbacena - MG, Setembro de 2020.

1. Introdução

A tabela hash

Os estudos sobre **estruturas de dados** não são novidade. Já há muito, cientistas e pesquisadores têm apresentado soluções para o processamento e armazenamento de grandes porções de dados que naturalmente despenderiam de alto custo computacional. Uma dessas estruturas, que visa o armazenamento e acesso de dados de maneira inteligente e veloz, e também alvo deste trabalho, é a **tabela hash**.

A estratégia da tabela hash consiste em dispor os dados em um vetor, associando chaves únicas aos elementos que serão inseridos nessa estrutura, de forma que possam ser rapidamente acessados.

Para isso, é necessário um processo chamado **hashing**, em que utiliza-se parte de um dado a ser inserido como uma chave única transformando-a em uma posição do vetor. A **função de hash** ou **função de espalhamento** é o código responsável por essa transformação, e, idealmente, deve definir posições bem esparsas, evitando o maior número de colisões possível. Uma **colisão** acontece quando dois elementos são mapeados para uma mesma posição na tabela.

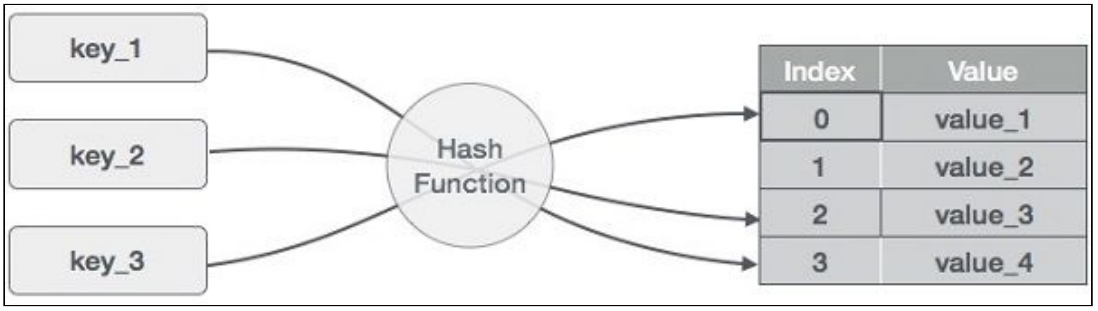

Figura 1-1: Mapeando de uma chave única à uma posição da tabela, por meio da função de hash. Disponível em: https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm

Tendo em vista que a função de hash deve ser consistida de um código eficiente, atingir posições exclusivas para cada elemento, eliminando totalmente colisões indesejadas, é uma tarefa muito difícil, se não for impossível. Para esse problema, algumas soluções de resolução de colisão podem ser aplicadas, dependendo de como o algoritmo da tabela for implementado. Neste trabalho, apresentaremos um software que utiliza e analisa a performance das seguintes estratégias:

- **Resolução de colisão por hash duplo:** consiste em utilizar uma segunda função de hash para encontrar outra posição vaga na tabela, uma vez que uma colisão ocorra. Para essa estratégia, é importante levar em conta o tamanho da estrutura e criar uma segunda função de espalhamento que não

cause *loops* eternos na busca por um espaço vazio, nem que ultrapasse os limites da tabela. *(Neste trabalho será também tratado como Hash Duplo).*

- **Resolução de colisão com lista encadeada:** nessa estratégia, a tabela deve abrigar uma estrutura de dados, lista encadeada, em cada uma de suas posições. Quando se deseja inserir um item em determinada posição da tabela, isto é feito transformando-o em um nodo da lista contida naquela posição do vetor. As colisões que acontecerem no futuro serão tratadas como nós subsequentes aos item que nela já se encontra. *(Neste trabalho será também tratado como Hash Lista).*

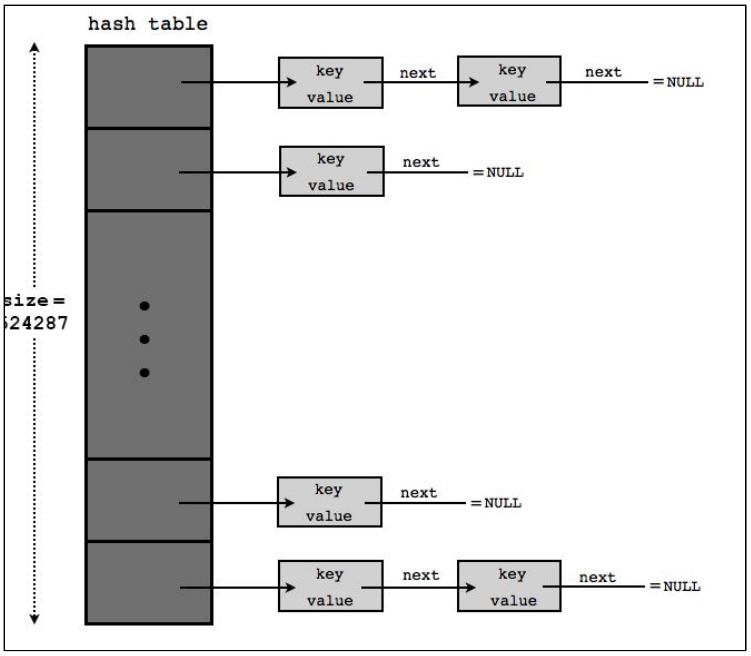

Figura 1-2: Tabela hash com lista encadeada.
Disponível em: https://www.clear.rice.edu/comp321/html/laboratories/lab06/

- **Resolução de colisão com árvore binária de pesquisa:** consiste na mesma idéia da resolução com listas, com o diferencial de agora usarmos árvores binárias de pesquisa nas posições da tabela. A vantagem é que as árvores binárias possuem uma performance melhor para a inserir, consultar e remover itens, na faixa de **O(log n)**, por isso teriam uma resposta mais rápida na resolução de colisões, embora tenham uma implementação ligeiramente menos trivial do que aquela utilizando listas. *(Neste trabalho será também tratado como Hash Abp).*

Sistema Eleitoral

O objetivo deste trabalho é a medição da performance da tabela hash usando as três estratégias de resolução de colisão supracitadas, através de um **sistema de eleições** simplificado. Escrito em **linguagem C**, o software deve receber uma entrada de ações advindas de um arquivo de texto, que podem ser:

- **Votar**: inserir um voto de um eleitor em um candidato.
    - Sintaxe: `[número da ação] [categoria do candidato] [título de eleitor] [número do candidato]`
    - Exemplo: `0 1 abcd123 500`

- **Remover**: remover os votos de um eleitor.
    - Sintaxe: `[número da ação] [título de eleitor]`
    - Exemplo: `1 abcd123`

- **Apurar**: exibir um ranking com as contagens de votos de cada candidato de uma categoria específica.
    - Sintaxe: `[número da ação] [categoria dos candidatos] [tamanho do ranking]`
    - Exemplo: `2 1 100`

Todas as ações realizadas pelo programa devem ser reportadas em um arquivo de texto de saída, e devem somente ser feitas seguindo determinadas regras:

- Um eleitor só pode votar em um candidato de cada categoria.
- Os votos de um eleitor são podem ser retirados se ele não houver votos registrados.
- Nem todos eleitores votarão em todas as categorias e nem todos os candidatos terão votos.
- Duas são as categorias de candidatos: 0 para prefeito, 1 para vereador.

O cabeçalho do arquivo de entrada, na primeira linha, definirá qual tipo de estrutura de dados será utilizada:

1. Tabela Hash com Resolução de Colisão de Colisão por Hash Duplo
2. Tabela Hash com Resolução de Colisão com Árvore Binária de Pesquisa
3. Tabela Hash com Resolução de Colisão com Lista Encadeada

Na linha subsequente, define-se quantos serão os candidatos para vereador, prefeito e o número de eleitores, informação crucial para a criação das tabelas que guardarão os votos e os eleitores no sistema eleitoral. Exemplo do cabeçalho:

```
1
27 10 2561
```

Da terceira linha em diante, as operações do sistema começarão a serem analisadas e processadas. O programa finalizará quando for lida ação de número 3, até então não descrita, comando para finalizar a leitura e terminar a execução.

2. Implementação

Leitura dos dados

O programa deve ser executado via linha de comando, e recebe dois parâmetros, o arquivo de entrada e de saída. Cada linha do arquivo de entrada é lida utilizando a função da *stdlib* `fgets()`. Os parâmetros de cada ação são extraídos utilizando `strtok()`. Valores numéricos são convertidos de `char*` para `int`, usando `atoi()` . Quando a entrada atinge a ação de número 3, a execução é finalizada.

Criação das estruturas

Lido o cabeçalho, as estruturas serão então criadas. Para a resolução do problema, são criadas três estruturas:

- Uma tabela hash com resolução de colisão definida pelo usuário, para armazenar os eleitores e seus votos para prefeito e vereador.

```
typedef struct TChave {
    char tituloEleitor[TITULO_ELEITOR];

} TChave;

typedef struct TStatusVotos {
    bool votouPrefeito;
    bool votouVereador;

} TStatusVotos;

typedef struct {
    int prefeito;
    int vereador;
    TChave id;
    TStatusVotos statusVotos;
    TCondicao condicao;

} TElemento;
```

Figura 2-1: `cedula.h` que será o TAD utilizado pela tabela hash de eleitores.

- Duas tabelas hash com resolução de colisão por hash duplo para armazenar a os números de vereadores e prefeitos já inseridos e seus números de votos.

```
typedef struct{
    int numeroCandidato;
    unsigned int numeroVotos;
    TStatusCandidato status;
    TCondicao condicao;
}TCandidato;
```

Figura 2-2: `candidato.h` que será o TAD utilizado pela tabela hash de eleitores e prefeitos

Cada uma das tabelas criadas, além de um ponteiro para o TAD, guardará os atributos: **tamanho** (capacidade máxima da tabela), **número de elementos** e **número de colisões**.
Na etapa de criação um ponto crucial é o tamanho da tabela. Ele deve ser baseado no número de itens a serem inseridos na tabela, mas por convenção, recomenda-se que seja **1.5 vezes maior** que o número de elementos a serem inseridos e para que haja um boa performance, que o tamanho seja um **número primo**. Isto é assim feito, e as consequências disso serão entendidas adiante.

Gerando uma chave única
A chave única é necessária para geração de um índice na tabela através da função de hash. Para isso, utiliza-se neste trabalho, o número do candidato como chave única nas tabelas de candidatos e o título de eleitor na tabela de eleitores. Porém, neste último caso, sendo o título um valor do tipo `char*`, uma conversão para inteiro é necessária antes que o hashing seja realizado. Dessa forma, converte-se uma *string C* em `unsigned int` por meio do seguinte algoritmo:

```c
//Transforma um TChave em um valor numerico.
unsigned int transforma_chave(TChave ch) {
    unsigned int i, soma = 0;
    for (i = 0; ch.tituloEleitor[i] != '\0'; i++) {
        //Através de experimentos notou-se que o somatório de i a um número
        //de casas decimais maiores
        //pode reduzir o número de colisões drasticamente.
        soma += ch.tituloEleitor[i] * (i + 1000);
    }
    return soma;
}
```

Figura 2-3: Transformação de um título de eleitor em um número inteiro.

Função de hash
A função de espalhamento é uma simples transformação de uma chave única em um índice da tabela, usando uma fórmula simples:

```
H(chave) = chave % tamanho da tabela
```

O uso da operação de módulo, garante que nunca seja gerado um índice maior que o tamanho da tabela. O fato do tamanho da tabela ser um número primo evita que valores iguais sejam gerados com mais frequência, diminuindo assim o número de colisões.

Função de incremento
A função de incremento é apenas necessária quando se utiliza Hash Duplo. Quando uma colisão ocorre, a partir desta fórmula, pesquisa-se uma nova posição na tabela:

```
H2(chave) = NUMERO_PRIMO - (chave % NUMERO_PRIMO)
```

Sendo `NUMERO_PRIMO` um valor constante primo, menor que o tamanho da tabela.
Desta forma, temos uma função de incremento que respeita duas regras essenciais: nunca retornar 0, e ser diferente da função de hash.

A partir dela, quando `H(chave)` resulta em colisão, procura-se uma nova posição fazendo: `posição = H(chave) + H2(chave) % tamanho da tabela`.

Inserindo um voto

Quando uma ação de inserção é lida, os seguintes passos são realizados:

- **Checagem**: verifica-se se os parâmetros advindos do arquivo de entrada são íntegros.
- **Já existe um voto do eleitor?**: ao identificar o título de eleitor da ação, checa-se se o mesmo já existe no sistema.
    - Pesquisa usando **Hash Abp** e **Hash Lista**: depois de realizado o hashing através da chave em que se deseja procurar, acessa-se a estrutura de dados associada a posição obtida. Percorre-se a estrutura e retorna o elemento desejado se existir.

    - Pesquisa usando **Hash Duplo**: realiza-se o hashing obtendo `H(chave)`.
        - Se o índice associado seja a de uma **posição vazia**, significa que o valor não existe na tabela, pois sequer fora inserido.
        - Se o índice associado seja a de uma posição de um elemento que fora **removido**, deve-se utilizar o incremento (descrito na seção **Função de incremento**) e repetir o teste, até que todas as posições sejam testadas.
        - Se o índice associado for a do um elemento ao qual se busca e este elemento não fora previamente removido, significa que a busca obteve **sucesso** e o item é então retornado.
        - Para a averiguação de uma posição do *array*, se está preenchida, vazia ou removida, utiliza-se a seguinte enumeração:

```
#ifndef CONDICAO_H
#define CONDICAO_H
enum Condicao { VAZIO = 1, CHEIO = 2, DELETADO = 3 };

typedef int TCondicao;

#endif
```

Figura 2-4: representação do estado de um elemento na tabela, definida em `condicao.h`

- **Computa o novo voto se possível:** se o eleitor existe, verifica-se se ele já votou para o candidato a quem pretende votar agora.
    - Se já havia registrado voto para outro candidato, acrescenta a nova escolha no item retornado pela pesquisa anterior e acrescenta um voto no candidato escolhido. (Importante frisar que aqui estamos tratando de duas estruturas

diferentes, a que contém o voto (cédula) com as escolhas de um eleitor e a que contém o candidato e número de votos recebidos).
    - Se nunca havia votado, insere-o desta vez e acrescenta um voto no candidato escolhida.
    - Caso já tenha votado para todas as categorias, seu novo voto não é computado.

Removendo um voto

- Verifica se o eleitor existe no sistema, em caso positivo:
    - Decrementa um voto para o candidato de cada categoria se os votos existirem nas tabelas de candidatos.
    - Exclui a cédula do eleitor da tabela de eleitores.

Exibindo o ranking

Para a exibição do ranking, a partir do tipo de candidato escolhido, copia-se os elementos da tabela hash original que não estão com condição de vazio ou removido, para um vetor auxiliar, local e estático, com o tamanho sendo o número de elementos presente na tabela. Esta é uma operação de natureza linear (**O(n)**), mas é necessária para evitar falhas de segmentação.
O próximo passo é a ordenação do ranking, para isso utiliza-se a *stdlib* novamente, com a função `qsort()`. O resultado é então imprimido no arquivo de saída.

3. Experimentos

Foram realizados testes usando arquivos de entrada:

- **de cinco mil ações**, para 27 prefeitos, 10 vereadores e 2561 eleitores.
- **de quinhentas mil ações**, para 118 vereadores, 55 prefeitos e 25080 eleitores.

Dessa forma, podemos analisar o desempenho das estrutura em instâncias pequenas e grandes.
Para o arquivo de entrada de cinco mil linhas, o gráfico na **Figura 3-1** resume a performance das três estruturas, contendo o tempo de execução e o número de colisões de cada.
Podemos notar que o **Hash Lista** e o **Hash Abp** obtiveram o mesmo **número de colisões**, já que a estratégia de resolução dessas tabelas é quase a mesma, mudando apenas a natureza das estruturas auxiliares. É também notório que o número de colisões do **Hash Duplo** é muito maior que as demais, justamente pela necessidade de realizar incrementos constantemente, até que uma posição vazia seja encontrada.
Para medir os **tempos de execução**, cada uma das estruturas foi testada cinco vezes para um mesmo arquivo e uma média aritmética fora feita a partir desses dados. Podemos ver que nesse caso, todos as três estratégias tiveram um desempenho satisfatório e muito semelhante.
Embora o resultado obtido seja perto do esperado - como veremos adiante -, entradas pequenas de dados são geralmente insuficientes para medir a velocidade desses algoritmos, pois nos processadores atuais, computar porções de dados dessa magnitude

soa como uma tarefa trivial, além de que as estatísticas obtidas a partir dessas execuções de baixa instância podem ser severamente prejudicadas por oscilações da CPU.

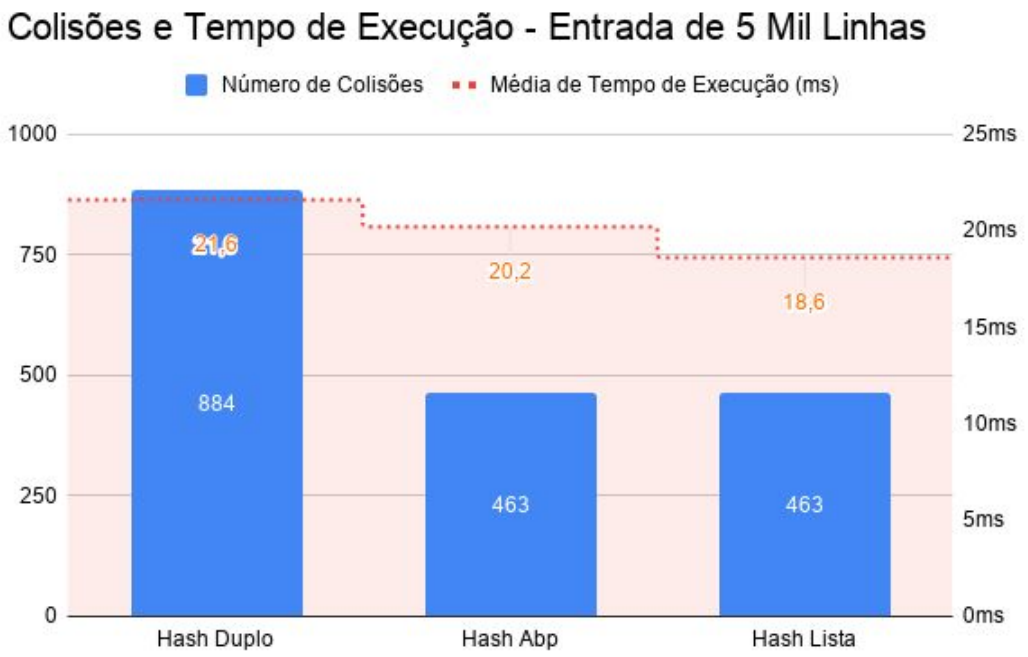

Figura 3-1: medindo o tempo de execução em milissegundos e o número de colisões das estruturas para uma entrada de dados pequena.

Figura 3-2: medindo o tempo de execução em milissegundos e o número de colisões das estruturas para uma entrada de dados grande.

Já usando um arquivo de entrada com quinhentas mil ações, podemos notar uma séria discrepância, tanto em tempo de execução como em número de colisões, ao comparar o **Hash Duplo** com os demais métodos. As colisões ocorrem em demasia pela natureza da solução do problema, quando a tabela está cheia, cada pesquisa, inserção e remoção da cai para uma performance quase linear. Podemos ver que o também desastroso tempo de execução é inerente a isso.
A performance dos **Hash Abp** e **Hash Lista** é muito semelhante, mas é importante e ao mesmo tempo surpreendente notar que a solução usando lista encadeada, que apesar de utilizar um modo linear para suas operações, se saiu melhor que aquela utilizando árvore binária de pesquisa. A explicação mais plausível para isso é que como as estruturas auxiliares abrigam apenas itens frutos de colisões, os benefícios de uma árvore na busca não são tão sentidos, enquanto que por outro lado, as remoções e inserções são mais refletidas negativamente. Ou seja, nota-se que para instâncias pequenas, como no caso de uma posição da tabela hash, listas podem ser mais eficientes que árvores.

4. Conclusão

Em resumo, **a solução utilizando listas encadeadas** se mostrou a mais eficiente das opções e associado ao fato de ter uma implementação bem mais **simples** que as demais, termina este trabalho como a **melhor** das alternativas.
Enfim, o desenvolvimento do trabalho fora concluído com sucesso, algumas dificuldades foram encontradas, principalmente com relação às funções de hash e na checagem de colisões, na escolha de um algoritmo que minimizasse essas incidências, que evitasse *loops* e que tivesse um tempo de execução satisfatório.
No mais, poderia contar com aperfeiçoamentos, como a realocação da tabela no caso do hash do duplo que pode descartar votos quando a capacidade máximo é atingida.

5. Referências utilizadas no desenvolvimento do trabalho

- C++ Reference - https://en.cppreference.com/w/
- Double Hashing - Geeks for Geeks - https://www.geeksforgeeks.org/double-hashing/
- C Program to Implement Hash Tables with Double Hashing - Sanfoundry - https://www.sanfoundry.com/c-program-implement-hash-tables-with-double-hashing/
- Data Structure and Algorithms - Hash Table - Tutorials Point - https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm