Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
Bacharelado em Tecnologia da Informação
Estrutura de Dados Básica

Tabela Hash

Autor: Yuri Alessandro Martins

16 de maio de 2016
Natal/RN

Sumário

1 Introdução

Uma tabela de dispersão, ou **Hash Table**, é uma TAD que funciona como um *dicionário*, contendo uma **chave** e uma **informação** para cada objeto armazenado. Cada chave é mapeada em um determinado ponto da tabela.

Toda tabela possui um **Universo de Chaves**, que nada mais é do que todas as chaves possíveis de serem utilizadas. Obviamente não iremos criar uma tabela com o tamanho do Universo de Chaves, até porque a quantidade que será realmente usada deve ser bem menor[1]. Então nossa tabela pode ser um vetor de tamanho n, e as chaves poderão ser armazenadas entre as posições 0 e n-1.

A determinação de em qual posição cada chave irá ficar é feita pela técnica de hash (2). A ideia é que cada chave consiga ocupar uma posição única dentro da tabela, o que claramente podemos constatar que é muito dificil em termos de grande volume de dados a serem armazenados.

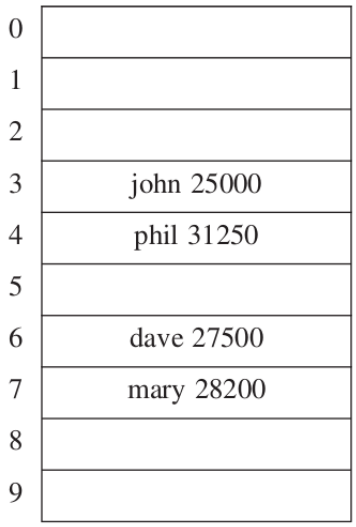

Figura 1: Exemplo de organização de uma tabela de dispersão para "RG" e nome de uma pessoa. Imagem retirada de *Data Structure and Algorithms in C++* [1]

[1]"Deve ser bem menor" no sentido de que provavelmente não iremos utilizar tantas chaves ao ponto de utilizarmos todo nosso Universo de Chaves

2 Função de Hashing

Também tida como **função de espelhamento**, a função hash é a parte mais importante dessa técnica. Caso não seja bem escolhida, a implementação pode ter um péssimo desempenho.

Ela é a responsável por encontrar, dada uma chave, uma posição de armazenamento adequada no nosso vetor (tabela). É importante que a função seja capaz de **retornar uma mesma chave para uma mesma entrada, sempre** (um exemplo de função hash é a *modular*, que determina a posição como $chave \ \% \ n - 1$).

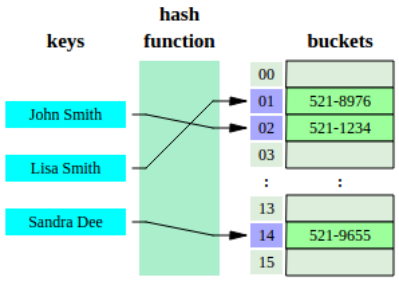

Figura 2: Rápida demonstração de como função de hash funcionaria. Imagem retirada de *Wikipedia* [2]

```
int hash( int key, int tableSz ){
        int hashval;
        hashval = key % (tablesz-1);
        return hashval;
}
```

Listing 1: Uma rápida demonstração de como seria uma função de hash em C/C++

Agora, suponha que, usando a função descrita acima, iremos adicionar a chave 4 para uma tabela com tamanho 1000. Pela nossa função, a posição onde essa chave se encaixaria era a posição 4, dado que $4 \ \% \ 999 = 4$. Agora, suponha que após varias inserções de várias outras chaves (10, 1000, 2, 321321, 14243, ...) queremos adicionar a chave 1003. Pela nossa função, ela ocuparia a mesma posição que o elemento 4, pois $1003 \ \% \ 999 = 4$. Quando isso ocorre, dizemos que houve uma **colisão**

Uma função de hash perfeita, portanto, seria uma capaz de que para qualquer duas chaves diferentes, seriam geradas duas posições sempre diferentes, e que nunca houvessem colisões (nosso exemplo de *modular* não é um bom exemplo disso). Como não podemos evitar as colisões em 100%,

2.1 Resolvendo colisões

Podemos tentar resolver as colisões de várias maneiras. Uma maneira trivial, por exemplo, seria buscar, a partir da "posição original" da chave, o próximo espaço vazio do vetor e colocar a informação lá. Isso é nem um pouco prático, pois assim a busca ficará ruim, já que a informação não estará na posição de sua determinada chave.

Uma outra maneira pode ser relacionar a lista encadeada com uma outra TAD, a **lista encadeada**.

3 Relacionando com outras TADs

Também conhecido como **separate chaining**, essa técnica permite utilizarmos a TAD **lista encadeada** para tentar resolver o problema das colisões (tendo em vista que quanto menos colisão, mais eficiente é nossa implementação). Dessa forma, basicamente, ao invés de armazenamos diretamente a informação em uma posição do nosso vetor (tabela), armazenamos uma lista encadeada.

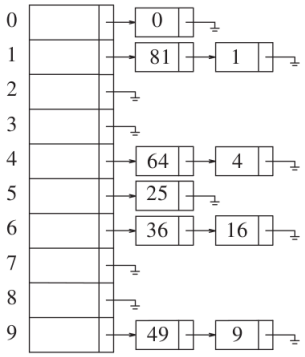

Figura 3: Exemplo de organização de uma tabela de dispersão utilizando lista encadeada. Imagem retirada de *Data Structure and Algorithms in C++* [1]

Assim, quando a colisão ocorre, podemos criar um novo nó e ir "armazenando várias chaves em uma mesma posição". Note que todas as chaves congruentes estarão na mesma lista encadeada, que poderá ser percorrida linearmente para achar a informação de uma chave.

4 SHA-1 [3]

SHA-1 é uma função de dispersão (função hash) criada pela Agência Nacional de Segurança (**NSA**) dos Estados Unidos. Esse algoritmo é amplamente utilizado na área de criptografia, sendo publicado em 1995 (sucessor do SHA, publicado em 1995, que não foi muito adotado).

No caso dessa aplicação, como ela foi criada para o âmbito da criptografia, uma vez que a chave passa pela função de hash (ou seja, é criptografada), o valor gerado nunca poderá voltar a ser a chave original. Isso é conhecido como "Algoritmos de criptografia *One Way*". Ele é usado numa grande variedade de aplicações e protocolos de segurança, como por exemplo TLS, SSL, SSH, PGP, etc.

Como funciona como uma Tabela Hash, ele também possui a propriedade de sempre retornar uma mesmo valor hash[2] para uma mesma chave de entrada.

5 Conclusão

Ao utilizarmos a TAD de **tabela hash**, estamos armazenando uma coleção de **objetos** com **chaves associadas** a cada um (Imagem 1). Diferentemente de outras TADs, o local onde esse objeto será armazenado dentro da nossa tabela (basicamente, um vetor de tamanho n) é determinado por meio de uma **função de hash** (Imagem 2), que deverá (i) as chaves uniformemente pelo vetor (tabela), (ii) garantir que duas chaves iguais irão armazenar uma informação numa mesma posição e (iii) evitar o máximo que possível as **colisões**, que ocorre quando duas chaves diferente acabam tendo que ocupar a mesma posição na tabela.

Vimos, portanto, que uma maneira de acabar com as colisões é associando a TAD de tabela hash com a tad de **lista encadeada**, fazendo como que várias chaves sejam agrupadas congruentemente numa mesma posição da tabela (Imagem 3).

A TAD de Hash Table tem a vantagem de permitir que a **busca de elementos** seja feita de maneira bastante **rápida**, pois para recuperar a informação tudo que precisamos fazer é passar a chave pela função de hash e dessa forma teremos a posição em que a informação se encontra, não sendo necessário percorrer todo o vetor até encontrar a informação, como num array. Todavia, isso faz com que a complexidade do programa esteja diretamente ligada a sua função de hash (quanto melhor ela for, melhor seu programa será).

[2]Basicamente, o que a gente tratava como "posição" onde a chave seria inserida.

Durante as várias inserções, a tabela vai perdendo sua ordem relativa, ficando "desorganizada". Isso faz com que não seja possível criar funções de "sucessor", "antecessor", entre outros, o que pode ser considerado uma desvantagem dessa TAD.

Referências

[1] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998.

[2] Wikipedia. Hash table — wikipedia, the free encyclopedia, 2016. [Online; accessed 16-May-2016].

[3] Wikipedia. Sha-1 — wikipedia, the free encyclopedia, 2016. [Online; accessed 16-May-2016].