Funções e tabelas de hash: implementações e aplicações

Breno Helfstein Moura (Supervisor: José Coelho de Pina Júnior)

Trabalho de Formatura Supervisionado do Bacharelado em Ciência da Computação no Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP)

Por que estudar *funções e tabelas de hash*?

Funções e tabelas de hash são amplamente usadas em ciência da computação e engenharia de software. Podemos imaginar funções de hash como uma função que gera uma assinatura digital de um objeto para identificá-lo futuramente. Por sua vez, tabela de hash é uma estrutura de dados que admite operações de inserção, busca e remoção de pares chave-valor. Um dos ingredientes fundamentais é que, funções de hash devem ser parecidas com funções aleatórias, para que exista um número baixo de colisões e as operações em tabelas de hash sejam realizadas eficientemente.

O que foi feito?

Implementações e análises de funções clássicas de hash [1], de diversas de tabelas de hash e de algumas aplicações utilizando ambas as técnicas. Entre as aplicações estudadas temos a verificação de isomorfismo em árvores e, entre as implementações de tabela de hash, temos *Cuckoo Hashing*.

Como verificar *isomorfismo de árvores*?

Durante o filme *Gênio indomável* (1997), Will Hunting (Matt Damon), faxineiro do MIT, resolve um problema deixado em uma lousa da famosa universidade americana. O problema em questão é encontrar todas as árvores homeomorficamente irredutíveis diferentes, de 10 vértices.

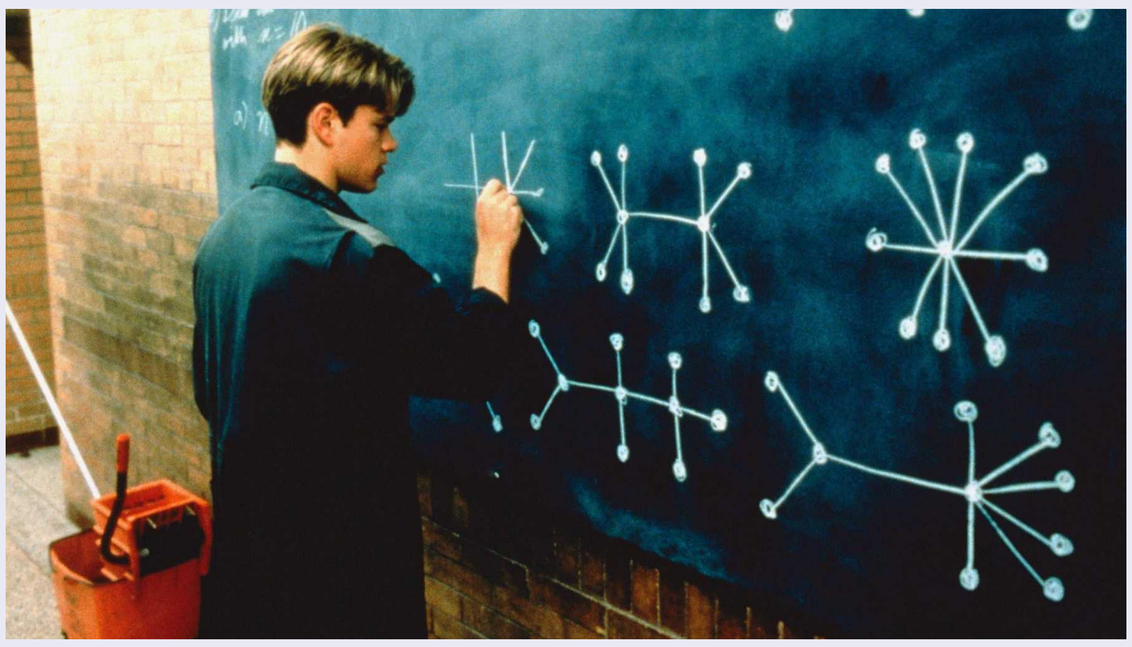

Figura 1: Cena do filme *Gênio Indomável*

Mas afinal o que são árvores diferentes? Duas árvores T_1 e T_2 são iguais, ou isomorfas, se existe uma bijeção dos vértices de T_1 para os vértices de T_2, que preserve as adjacências da árvore. Podemos perceber, que duas árvores são iguais se conseguirmos obter T_1 a partir de T_2 renomeando os nós. Logo, duas árvores são diferentes se elas são não isomorfas.

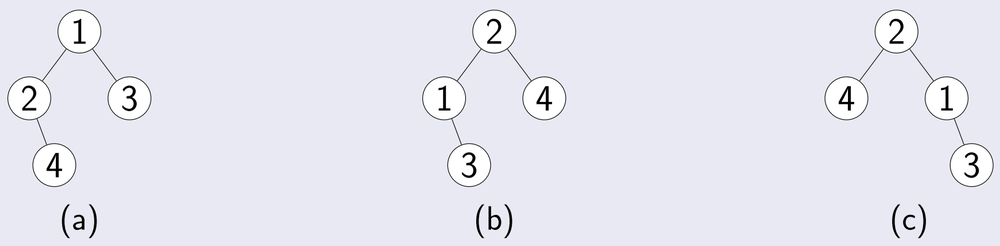

Figura 2: Três árvores isomorfas

Podemos verificar se as árvores dadas por Will são isomorfas utilizando uma função de hash específica para árvores enraizadas [2]. Essa função pode ser calculada de forma recursiva pela seguinte fórmula:

$$h(N) = \begin{cases} x_0 \text{ se N é uma folha da árvore} \\ \Pi_{i=1}^{k}(x_d + h(C_i)) mod\ M \text{ onde } C_i \text{ é um filho de N e d é a altura de N,} \end{cases}$$

onde x é um vetor de números aleatórios maiores que 0 préfixado. Para calcularmos esse hash precisamos primeiro enraizar as árvores dadas por Will. Podemos enraizar pelo centro ou centroid, pois sabemos que toda árvore tem no máximo dois vértices de centro ou centroid [3]. Abaixo podemos ver duas árvores do filme *Gênio Indomável* enraizadas pelos seus centros:

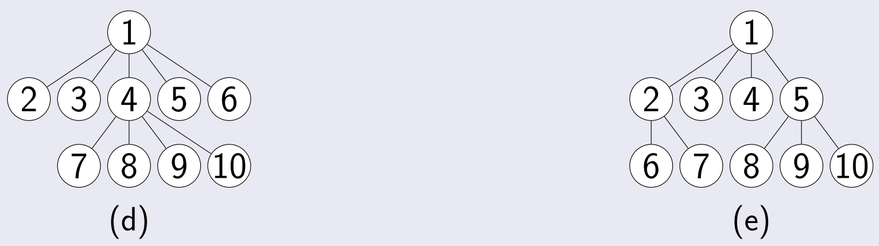

Figura 3: Árvores do filme *Gênio indomável*

Como ambas árvores possuem altura dois, podemos supor o seguinte vetor x de tamanho três: $[13, 1, 9]$. Logo se supormos $M = 103$, o valor de hash da árvore (d) seria 19 enquanto o valor de hash da árvore (e) seria 58, e como são diferentes podemos afirmar com certeza que não são isomorfas (Uau! Parte da resposta de Will está correta). Caso fossem iguais poderia ainda ter ocorrido, com baixa probabilidade, uma colisão. Nesse útlimo caso, poderíamos apenas afirmar que as árvores seriam *provavelmente* isomorfas.

Como são implementadas tabelas de hash?

Tabela de hash é uma estrutura de dados que admite inserção, busca e remoção de pares chave-valor de forma eficiente. Para isso ela se utiliza de funções de hash para indicar as posições na tabela onde um par chave-valor pode ser possivelmente inserido ou encontrado.

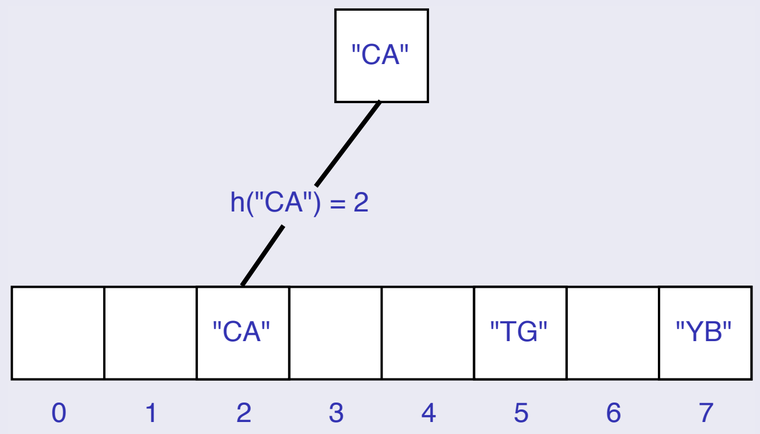

Figura 4: Exemplo de uma tabela de hash onde h é a função de hash.

Contudo, apesar de funções de hash se comportarem idealmente como funções aleatórias, existe uma grande chance de colisão conforme a tabela vai sendo preenchida. Para ilustrar, podemos lembrar do paradoxo do aniversário, que é o fato de que só são necessárias 23 pessoas em uma sala para a chance de duas pessoas terem a mesma data de aniversário ser superior a 50%. Com 57 pessoas a probabilidade é maior que 99%. Podemos perceber que $\frac{23}{365} = 6.3\%$ e que $\frac{57}{365} = 15.6\%$. Logo precisamos aprender a lidar com colisões neste tipo de estrutura de dados.

Para lidar com colisões, exitem diversas técnicas, entre elas temos *Linear Probing*, *Quadratic Probing*, *Robin Hood Hashing*, *Chaining Hashing*, *Cuckoo Hashing*, entre outras. Algumas técnicas se utilizam de mais memória que outras, ou possuem mais garantias ou são mais simples de implementar, tendo diversos prós e contras para cada uma das implementações. *Chaining Hashing* por exemplo, é fácil de ser especificado e possui operações muito bem definidas, o que faz dela a implementação preferida para tabelas de hash padrão de linguagens de programação como C++, Java, Golang, C# e Scala.

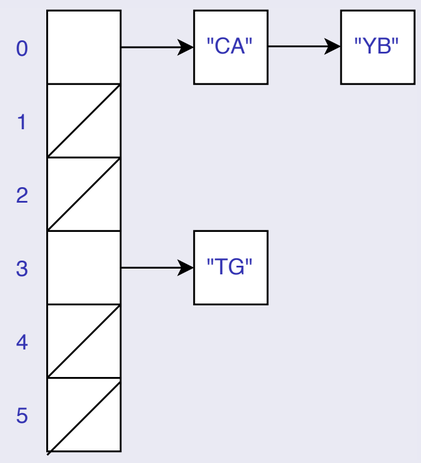

Figura 5: Ilustração de uma tabela de hashing implementada com chaining hashing

Cuckoo Hashing

Cuckoo Hashing é uma das implementações mais exóticas de tabela de hash, contudo é uma implementação muito interessante, pois possui tempo de busca e remoção constante no pior caso, fato que não é verdade para a maioria das outras implementações [4]. Essa implementação de tabela de hash utiliza dois ou mais vetores e duas ou mais funções de hash, e quando existe uma colisão durante a inserção ela move pares chave-valor de um vetor para o outro. O nome Cuckoo vem do passarinho de mesmo nome, que é conhecido por depositar seus ovos em ninhos de outros passarinhos, muitas vezes empurrando os ovos que ali estavam para fora. Para mover os pares chave-valor de um vetor para o outro, o algoritmo primeiro insere o par que já estava tentando inserir na posição colidida e então tenta inserir o par que foi removido em outro vetor. Para a busca e remoção basta olhar em todos os vetores na posição para o qual a função de hash indica. Vale mencionar que a maioria das implementações de *Cuckoo Hashing* se utiliza de dois vetores.

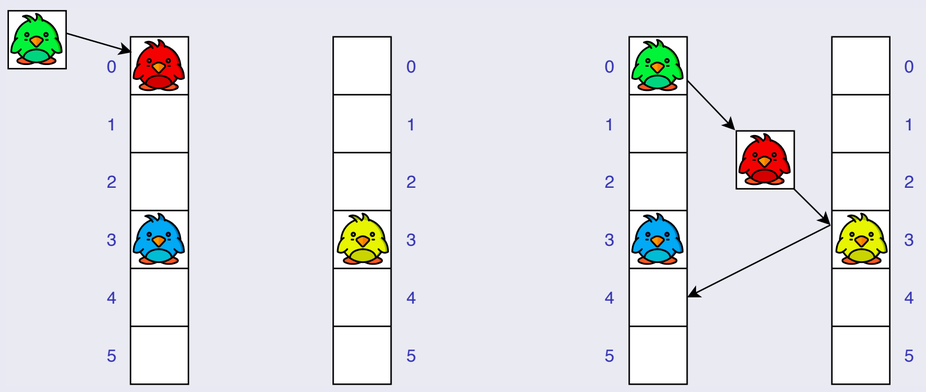

Figura 6: Ilustração da inserção do cuckoo hashing

Mais informações

Este pôster, a monografia e os códigos desenvolvidos estão disponíveis em:
`https://github.com/breno-helf/TCC`

Bibliografia

[1] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[2] rng_58. Hashing and probability of collision, 2017.

[3] Igor Carpanese. An illustrated introduction to centroid decomposition, 2018.

[4] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing, 2001.