Apontamentos de Estruturas de Dados

Hashing

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Dezembro de 2022

Índice

16. *Hashing*

16.1. Introdução

Num mundo ideal, recuperar um valor de um mapa (*map*) seria feito instantaneamente através da chave do valor. Esse é o objetivo do *hash*, que usa uma função de *hash* para transformar uma chave num índice de *array*, fornecendo assim acesso instantâneo ao valor armazenado num *array* que contém os pares chave-valor no mapa. Esse *array* é denominado de tabela de *hash*.

Função de *hash*: Uma função que transforma uma chave num valor no intervalo de índices de uma tabela de *hash*.

Tabela de *hash*: um *array* que contém pares de associações chave-valor cujas localizações são determinadas pela função de *hash*.

16.2. O problema de *hash*

Se um conjunto de pares chave-valor for pequeno e pudermos alocar um *array* grande o suficiente para contê-los todos, podemos encontrar uma função de *hash* que transforma as chaves em posições exclusivas numa tabela de *hash*. Por exemplo, em algumas linguagens de programação antigas os identificadores consistiam numa letra maiúscula possivelmente seguida por um dígito. Suponha que essas seriam as nossas chaves. Existem 286 identificadores e não é muito difícil encontrar uma função que mapeie cada chave dessa forma para um valor único no intervalo 0 . .285. No entanto, geralmente o conjunto de chaves é muito grande para fazer uma tabela para armazenar todas as possibilidades.

Uma tabela menor que armazene as chaves com um grande intervalo de valores inevitavelmente exigirá que a função transforme várias chaves para a mesma posição na tabela. Quando duas ou mais chaves são mapeadas para a mesma posição na tabela através de uma função de *hash* temos uma colisão. Os mecanismos para lidar com estas situações são chamados de tratamento de resolução de colisões.

Colisão: O evento que ocorre quando duas ou mais chaves são transformadas para a mesma posição da tabela de *hash*.

Quão sério é o problema da colisão? Afinal, se tivermos uma tabela bastante grande e uma função de *hash* que distribua as chaves uniformemente pela tabela as colisões podem ser raras. Na verdade, porém, as colisões ocorrem com uma frequência surpreendente. Para percebermos porquê vamos considerar o problema do aniversário que é um famoso problema da teoria da probabilidade: qual é a hipótese de que pelo menos duas pessoas num grupo de k pessoas tenham o mesmo aniversário? O problema acaba por ser expressado da seguinte forma p =1-(365!/k!/365k).

A tabela apresentada de seguida lista alguns valores para esta expressão. Surpreendentemente, num grupo de apenas 23 pessoas existe a probabilidade que duas dessas pessoas tenham o mesmo dia de aniversário!

Se imaginarmos que uma tabela de *hash* tem 365 posições e que essas probabilidades são as probabilidades de que uma função de *hash* transforme dois valores para a mesma localização (colisão), podemos ver que é quase certo que iremos ter uma colisão quando a tabela tiver 100 valores e muito provavelmente terá uma colisão com apenas cerca de 40 valores na tabela. 40 é apenas cerca de 11% de 365, então vemos que as colisões são muito prováveis. O tratamento de colisões são, portanto, uma parte essencial para fazer o *hash* funcionar na prática.

k	*p*
5	0.027
10	0.117
15	0.253
20	0.411
22	0.476
23	0.507
25	0.569
30	0.706
40	0.891
50	0.970
60	0.994
100	0.9999997

Uma implementação de tabelas de *hash* requer, portanto, duas coisas:

- Uma função de *hash* para transformar chaves em posições de tabelas de *hash*, idealmente uma que reduza a probabilidade de colisões;
- Uma estratégia para o tratamento de colisões para lidar com as colisões que estão prestes a ocorrer.

16.3. Funções de *hash*

Uma função de *hash* deve transformar uma chave num inteiro no intervalo 0 . . .t, onde t é o tamanho da tabela de *hash*. Uma função de *hash* deve distribuir as chaves na tabela o mais uniformemente possível para minimizar as colisões. Embora muitas funções de *hash* tenham sido propostas e investigadas, as melhores funções de *hash* usam o método de divisão, que para chaves numéricas é o seguinte:

$$hash(k) = k \bmod t$$

Esta função é simples, rápida e distribui as chaves uniformemente na tabela. Funciona melhor quando t é um número primo não próximo de uma potência de dois. Por esse motivo, os tamanhos das tabelas de *hash* devem sempre ser escolhidos para serem um número primo não próximo a uma potência de dois.

Para chaves não numéricas geralmente há uma forma bastante simples de converter o valor num número e usar o método de divisão. Por exemplo, o pseudocódigo apresentado de seguida ilustra uma possível forma de implementação de uma função de *hash* para uma *string*.

```
1. def hash_function(string, table_size)
2.       result = 0
3.       string.each_byte do |byte|
4.         result = (result * 151 + byte) % table_size
5.       end
6.       return result
7. end
```

Assim, fazer funções de *hash* não é muito complicado. Uma boa regra prática é usar números primos sempre que uma constante for necessária e testar a função num conjunto representativo de chaves para garantir que são distribuídas uniformemente pela tabela de *hash*.

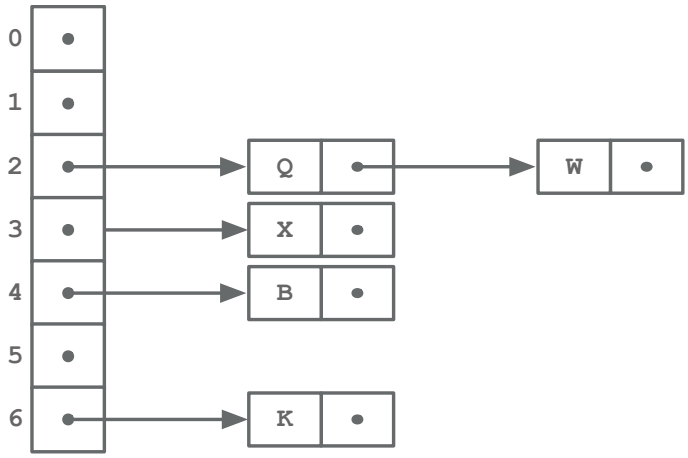
16.4. Tratamento de Colisões

Existem dois tipos principais de estratégias para o tratamento e resolução de colisões com muitas variações: listas e dispersão aberta. Em cada uma destas abordagens um valor importante a ser considerado é o fator de carga, λ=n/t, onde n é o número de elementos na tabela de *hash* e t é o tamanho da tabela.

16.4.1. Listas

No caso das listas os elementos cujas chaves colidem são formados numa lista ligada ou cadeia cuja "cabeça" está no array da tabela de *hash*. A figura apresentada de seguida mostra uma tabela de *hash* com colisões resolvidas através de listas. Para simplificar, apenas as

chaves são listadas, e não os elementos que as acompanham (ou, se preferir, a chave e o valor são os mesmos).



Neste exemplo, a tabela tem sete posições. Duas chaves, q e w, colidem na posição dois e são colocadas numa lista ligada cuja cabeça está na posição dois. As chaves x, b e k são o hash para os locais três, quatro e seis, respectivamente. Este exemplo usa um *array* de cabeças da lista, mas também poderia ter sido usado um *array* com os nós de lista com algum valor especial no campo de dados para indicar quando um nó não é usado.

Ao usar listas, o comprimento médio da cadeia é λ. Se a cadeia não for ordenada, as pesquisas bem-sucedidas irão exigir cerca de 1+ λ/2 comparações em média e as pesquisas malsucedidas λ comparações em média. Se a cadeia for ordenada, tanto as pesquisas bem-sucedidas como as malsucedidas levam cerca de 1+ λ/2 comparações, mas as inserções demoram mais. No pior caso, que ocorre quando todas as chaves são mapeadas para um único local de tabela e a chave de pesquisa está no final da lista ligada ou mesmo quando não está na lista, as pesquisas exigem n comparações. Este é no entanto um caso extremamente improvável.

Estruturas ligadas mais complexas (como árvores de pesquisa binária), geralmente não melhoram muito o desempenho, principalmente se λ for mantido relativamente pequeno. Como regra geral, λ deve ser mantido menor que cerca de 10. No entanto o desempenho só degrada gradualmente à medida que o número de itens na tabela cresce, portanto, as tabelas

de *hash* que usam encadeamento para resolver colisões podem ter um bom desempenho para um grande intervalo de valores.

16.4.2. Dispersão aberta

Para o tratamento e resolução de colisão através de dispersão aberta, os registos com chaves em colisão são armazenados noutras posições livres na tabela de *hash* encontradas através de uma pesquisa linear. Estratégias diferentes para a dispersão aberta normalmente usam diferentes algoritmos para a pesquisa por posições livres e respetiva sequência para a alocação das colisões. Em todos os casos, no entanto, a tabela só pode conter tantos itens quantos forem as posições, ou seja, $n \leq t$, e λ não pode exceder um; esta restrição não está presente para a lista.

Estudos teóricos acerca da pesquisa sequencial aleatória determinam níveis ideais de desempenho para a dispersão aberta. O desempenho diminui drasticamente à medida que o fator de carga se aproxima de 1. Por exemplo, com um fator de carga de 0,9, o número de comparações para uma pesquisa bem-sucedida é de cerca de 2,6 e para uma pesquisa malsucedida é de 20. Esquemas de dispersão aberta reais não funcionam tão bem quanto isso, então os fatores de carga geralmente devem ser mantidos abaixo de cerca de 0,75.

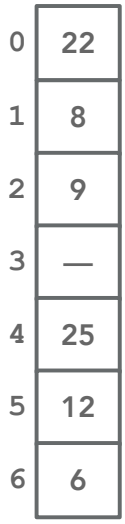
Outro ponto a ser compreendido acerca da dispersão aberta é que quando ocorre uma colisão, o algoritmo prossegue através da sequência de teste até que (a) a chave desejada seja encontrada, (b) é encontrada uma posição livre, ou (c) toda a tabela é percorrida. Mas isso só funciona quando um marcador é colocado nas posições onde um elemento foi excluído para indicar que a posição pode não estar vazia, portanto, o algoritmo deve prosseguir com a pesquisa sequencial. Numa tabela dinâmica irão existir muitos marcadores e poucas posições vazias, portanto, o algoritmo irá precisar de seguir longas pesquisas sequenciais, especialmente para pesquisas malsucedidas mesmo quando o fator de carga for baixo.

A **pesquisa linear** usa uma sequência de pesquisa que começa com o índice da tabela de hash e incrementa-o por um valor constante do módulo do tamanho da tabela. Se o tamanho e o incremento da tabela forem primos, todas posições da tabela irão aparecer

na sequência da pesquisa. O desempenho da pesquisa linear diminui drasticamente quando os fatores de carga excedem 0,8. A pesquisa linear também está sujeita ao agrupamento primário, que ocorre quando aglomerados de locais preenchidos se acumulam em torno de um local onde ocorre uma colisão pela primeira vez. O agrupamento primário aumenta as chances de colisões e degrada bastante o desempenho.

O ***hash* duplo** funciona ao gerar um incremento a ser usado na sequência de teste com recurso a uma segunda função de *hash* na chave. A segunda função de *hash* deve gerar valores bem diferentes da primeira para que as duas chaves que colidem sejam mapeadas para valores diferentes pela segunda função de *hash* tornando as sequências de sonda para as duas chaves diferentes. O *hash* duplo elimina o agrupamento primário. A segunda função *hash* deve sempre gerar um número que seja primo do tamanho da tabela. Isso é fácil se o tamanho da tabela for um número primo. O *hash* duplo funciona tão bem que o seu desempenho aproxima-se de uma sequência de sonda verdadeiramente aleatória. É, portanto, o método de escolha para gerar sequências de pesquisa.

A figura seguinte apresenta um exemplo de dispersão aberta com *hash* duplo. Como anteriormente, o exemplo usa chaves apenas para simplificar e não associações chave-valor. A principal função de *hash* é f (x) = x mod 7, e a função hash usada para gerar uma constante para a sequência da pesquisa é g(x) = (x mod 5)+1. Os valores 8, 12, 9, 6, 25 e 22 são agrupados na tabela.



As primeiras cinco chaves não colidem. Mas 22 mod 7 é 1, então 22 colide com 8. A constante de teste para o hash duplo é (22 mod 5)+1 = 3. Adicionamos 3 ao local 1, onde ocorre a colisão, para obter o local 4. Mas 25 já está neste local, então adicionamos 3 novamente para obter o local 0 (contornamos o início do array usando o tamanho da tabela: (4+3) mod 7 = 0). O local 0 não está ocupado, então é onde 22 é colocado.

Observe que algum tipo de valor especial deve ser colocado nos locais desocupados - neste exemplo, usamos um traço. Um valor diferente deve ser usado quando um valor é removido da tabela para indicar que o local está livre mas que não estava antes para que as pesquisas continuem além desse valor quando for encontrado durante uma sequência de pesquisa.

Observamos que ao usar dispersão aberta para resolver colisões o desempenho diminui consideravelmente à medida que o fator de carga se aproxima de 1. Mecanismos de *hash* que usam dispersão aberta devem ter uma maneira de expandir a tabela, diminuindo assim o fator de carga e melhorando o desempenho. Uma nova tabela maior pode ser criada e preenchida percorrendo a tabela antiga e inserindo todos os registros na nova tabela. Observe que isso envolve o *hash* de todas as chaves novamente porque a função de *hash* geralmente irá usar o tamanho da tabela que agora mudou. Consequentemente, esta é uma operação muito cara.

Alguns esquemas de expansão da tabela funcionam de forma incremental, mantendo a tabela antiga e fazendo todas as inserções na nova tabela, todas as exclusões da tabela antiga e

talvez movendo os registros gradualmente da tabela antiga para a nova no decorrer de outras operações. Eventualmente, a tabela antiga fica vazia e pode ser descartada.

16.5. Notas Finais

Hashing usa uma função de *hash* para transformar uma chave numa posição da tabela de *hash* fornecendo assim acesso quase instantâneo aos valores através das suas chaves. Infelizmente, é inevitável que mais que uma chave sejam *hash* para cada posição da tabela, causando uma colisão e necessitando alguma forma de armazenar mais de um valor associado a uma única posição da tabela.

As duas principais abordagens para resolução de colisões são as listas e a dispersão aberta. As listas usam listas vinculadas de pares chave-valor que começam em posições da tabela de hash. A dispersão aberta usa sequências de sondagem para pesquisar na tabela uma posição livre para armazenar uma associação chave-valor e, posteriormente, encontrá-la novamente. A dispersão aberta é uma técnica muito robusta e tem bom desempenho para uma ampla variedade de fatores de carga mas requer espaço extra para os links nos nós da lista. A dispersão aberta usa o espaço com eficiência, mas o seu desempenho diminui rapidamente à medida que o fator de carga se aproxima de um e expandir a tabela é muito caro.

Não importa como o *hash* seja implementado, no entanto, o desempenho médio para inserções, remoções e pesquisas é O(1). O pior desempenho é O(n) para resolução de colisões ligadas no entanto isso ocorre apenas no caso muito improvável de que as chaves sejam geradas para apenas alguns locais da tabela. O desempenho do pior caso para a dispersão aberta é uma função do fator de carga que fica muito grande quando λ está próximo de 1, mas se λ é mantido abaixo de cerca de 0,8, W(n) é menor que 10.

16.6. Exercícios Propostos

Exercício 1

O que acontece quando duas ou mais chaves são mapeadas para o mesmo local numa tabela de *hash*?

Exercício 2

O que é um fator de carga? Podemos exceder o fator de carga?

Exercício 3

O que é uma sequência de sonda? O que é melhor: sondagem linear ou *hash* duplo?

Exercício 4

Porquê que o exemplo de dispersão aberta e *hash* duplo no exemplo apresentado usa a função de *hash g(x) = (x mod 5)+*1 em vez *de g(x) = x mod 5* para gerar sequências de sonda?

Exercício 5

Vamos supor que uma tabela de *hash* tem 11 posições e a função de *hash* do método de divisão simples *f (x) = x mod 11* é usada para mapear chaves para a tabela. Calcule os locais onde as seguintes chaves seriam armazenadas: 0, 12, 42, 18, 6, 22, 8, 105, 97. Alguma dessas chaves colide?

Exercício 6

Demonstre através de um pequeno problema a utilização de listas para o tratamento de colisões.

Exercício 7

Na sua opinião as tabelas de *hash* têm algum problema?