

Capítulo 11

Tabelas hash

Muitas aplicações exigem um conjunto dinâmico que admite apenas as operações de dicionário **INSERT**, **SEARCH** e **DELETE**. Por exemplo, um compilador para uma linguagem de computador mantém uma tabela de símbolos, na qual as chaves de elementos são cadeias de caracteres arbitrários que correspondem a identificadores na linguagem. Uma tabela hash é uma estrutura de dados eficiente para implementar dicionários. Embora a busca por um elemento em uma tabela hash possa demorar tanto quanto procurar por um elemento em uma lista ligada – o tempo $\Theta(n)$ no pior caso – na prática, o hash funciona extremamente bem. Sob hipóteses razoáveis, o tempo esperado para a busca por um elemento em uma tabela hash é $O(1)$.

Uma tabela hash é uma generalização da noção mais simples de um arranjo comum. O endereçamento direto em um arranjo comum faz uso eficiente de nossa habilidade para examinar uma posição arbitrária em um arranjo no tempo $O(1)$. A Seção 11.1 discute com mais detalhes o endereçamento direto. O endereçamento direto é aplicável quando temos condições de alocar um arranjo que tem uma única posição para cada chave possível.

Quando o número de chaves realmente armazenadas é pequeno em relação ao número total de chaves possíveis, as tabelas hash se tornam uma alternativa eficiente para se endereçar diretamente um arranjo, pois em geral uma tabela hash utiliza um arranjo de tamanho proporcional ao número de chaves realmente armazenadas. Em vez de usar a chave diretamente como um índice de arranjo, o índice de arranjo é *computado* a partir da chave. A Seção 11.2 apresenta as principais idéias, concentrando-se no “encadeamento” como um meio para tratar “colisões” no qual mais de uma chave é mapeada para o mesmo índice de arranjo. A Seção 11.3 descreve como os índices de arranjos podem ser computados a partir de chaves com o uso de funções hash. São apresentadas e analisadas diversas variações sobre o tema básico. A Seção 11.4 examina o “endereçamento aberto”, outro modo de lidar com colisões. A conclusão é que o hash é uma técnica extremamente eficaz e prática: as operações básicas de dicionário exigem apenas o tempo $O(1)$ em média. A Seção 11.5 explica como o “hash perfeito” pode dar suporte a pesquisas no tempo de *pior caso* $O(1)$, quando o conjunto de chaves que está sendo armazenado é estático (isto é, quando o conjunto de chaves nunca se altera uma vez armazenado).

11.1 Tabelas de endereço direto

O endereçamento direto é uma técnica simples que funciona bem quando o universo U de chaves é razoavelmente pequeno. Suponha que uma aplicação necessite de um conjunto dinâmico no qual cada elemento tenha uma chave definida a partir do universo $U = \{0, 1, \dots, m - 1\}$, onde m não é muito grande. Iremos supor que não há dois elementos com a mesma chave.

Para representar o conjunto dinâmico, usamos um arranjo ou uma **tabela de endereço direto** $T[0 \dots m - 1]$, na qual cada abertura (slot), ou **posição**, corresponde a uma chave no universo U . A Figura 11.1 ilustra a abordagem; a posição k aponta para um elemento no conjunto com chave k . Se o conjunto não contém nenhum elemento com chave k , então $T[k] = \text{NIL}$.

A implementação das operações de dicionário é trivial.

DIRECT-ADDRESS-SEARCH(T, k)

return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[\text{chave}[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)

$T[\text{chave}[x]] \leftarrow \text{NIL}$

Cada uma dessas operações é rápida: é necessário apenas o tempo $O(1)$.

Para algumas aplicações, os elementos no conjunto dinâmico podem ser armazenados na própria tabela de endereço direto. Ou seja, em vez de armazenar a chave e os dados satélite de um elemento em um objeto externo à tabela de endereço direto, com um ponteiro de uma posição na tabela até o objeto, podemos armazenar o objeto na própria posição, e portanto economizar espaço. Além disso, freqüentemente é desnecessário armazenar o campo de chave do objeto pois, se temos o índice de um objeto na tabela, temos sua chave. Entretanto, se as chaves não forem armazenadas, devemos ter algum modo de saber se a posição está vazia.

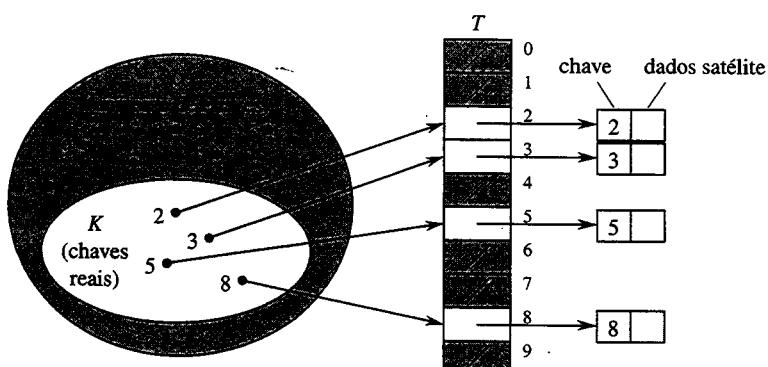


FIGURA 11.1 Implementação de um conjunto dinâmico por uma tabela de endereço direto T . Cada chave no universo $U = \{0, 1, \dots, 9\}$ corresponde a um índice na tabela. O conjunto $K = \{2, 3, 5, 8\}$ de chaves reais determina as posições na tabela que contêm ponteiros para elementos. As outras posições, fortemente sombreadas, contêm NIL.

Exercícios

11.1-1

Considere um conjunto dinâmico S que é representado por uma tabela de endereço direto T de comprimento m . Descreva um procedimento que encontre o elemento máximo de S . Qual é o desempenho de seu procedimento no pior caso?

11.1-2

Um **vetor de bits** é simplesmente um arranjo de bits (0s e 1s). Um vetor de bits de comprimento m ocupa muito menos espaço que um arranjo de m ponteiros. Descreva como usar um vetor de bits para representar um conjunto dinâmico de elementos distintos sem dados satélite. As operações de dicionário devem ser executadas em tempo $O(1)$.

11.1-3

Sugira um modo de implementar uma tabela de endereço direto na qual as chaves de elementos armazenados não precisem ser distintas, e os elementos possam ter dados satélite. Todas as três operações de dicionário (INSERT, DELETE e SEARCH) devem ser executadas no tempo $O(1)$. (Não se esqueça de que DELETE toma como argumento um ponteiro para um objeto a ser eliminado, e não uma chave.)

11.1-4 *

Desejamos implementar um dicionário usando endereçamento direto sobre um arranjo enorme. No início, as entradas do arranjo podem conter lixo e a inicialização do arranjo inteiro é impraticável devido a seu tamanho. Descreva um esquema para implementar um dicionário de endereço direto sobre um arranjo enorme. Cada objeto armazenado deve utilizar o espaço $O(1)$; as operações SEARCH, INSERT e DELETE devem demorar o tempo $O(1)$ cada uma; e a inicialização da estrutura de dados deve demorar o tempo $O(1)$. (Sugestão: Use uma pilha adicional, cujo tamanho é o número de chaves realmente armazenadas no dicionário, a fim de ajudar a determinar se uma dada entrada no arranjo enorme é válida ou não.)

11.2 Tabelas hash

A dificuldade com o endereçamento direto é óbvia: se o universo U é grande, o armazenamento de uma tabela T de tamanho $|U|$ pode ser impraticável, ou mesmo impossível, em virtude da memória disponível em um computador típico. Além disso, o conjunto K de chaves *realmente armazenadas* pode ser tão pequeno em relação a U que a maior parte do espaço alocado para T seria desperdiçada.

Quando o conjunto K de chaves armazenadas em um dicionário é muito menor que o universo U de todas as chaves possíveis, uma tabela hash exige muito menos espaço de armazenamento que uma tabela de endereço direto. Especificamente, os requisitos de armazenamento podem ser reduzidos a $\Theta(|K|)$, embora a pesquisa de um elemento na tabela hash ainda exija apenas o tempo $O(1)$. (A única desvantagem é que esse limite corresponde ao *tempo médio*, enquanto no caso do endereçamento direto ele se mantém válido para o *tempo do pior caso*.) Com o endereçamento direto, um elemento com a chave k é armazenado na posição k . No caso do hash, esse elemento é armazenado na posição $b(k)$; ou seja, uma **função hash** b é utilizada para calcular a posição a partir da chave k . Aqui, b mapeia o universo U de chaves nas posições de uma **tabela hash** $T[0 .. m - 1]$:

$$b : U \rightarrow \{0, 1, \dots, m - 1\}.$$

Dizemos que um elemento com a chave k **efetua o hash** para a posição $b(k)$; dizemos também que $b(k)$ é o **valor hash** da chave k . A Figura 11.2 ilustra a idéia básica. A finalidade da função hash é reduzir o intervalo de índices de arranjos que precisam ser tratados. Em vez de $|U|$ valores, precisamos manipular apenas m valores. Os requisitos de armazenamento são reduzidos de modo correspondente.

O detalhe fundamental dessa boa idéia é que duas chaves podem ter o hash na mesma posição. Chamamos essa situação de **colisão**. Felizmente, existem técnicas eficientes para resolver o conflito criado por colisões.

É claro que a solução ideal seria evitar por completo as colisões. Poderíamos tentar alcançar essa meta escolhendo uma função hash adequada b . Uma idéia é fazer b parecer “aleatória”, evitando assim as colisões, ou pelo menos minimizando seu número. A expressão “efetuar o hash”, que evoca imagens de misturas e recortes aleatórios, capta o espírito dessa abordagem. (É claro que uma função hash b deve ser determinística, no sentido de que uma dada entrada k sempre deve produzir a mesma saída $b(k)$.) Porém, tendo em vista que $|U| > m$, devem existir duas cha-

ves que tenham o mesmo valor hash; assim, é impossível evitar totalmente as colisões. Desse modo, embora uma função hash bem projetada e de aparência “aleatória” possa minimizar o número de colisões, ainda precisaremos de um método para resolver as colisões que ocorrerem.

O restante desta seção apresenta a técnica mais simples para resolução de colisões, chamada encadeamento. A Seção 11.4 introduz um método alternativo para solucionar colisões, denominado endereçamento aberto.

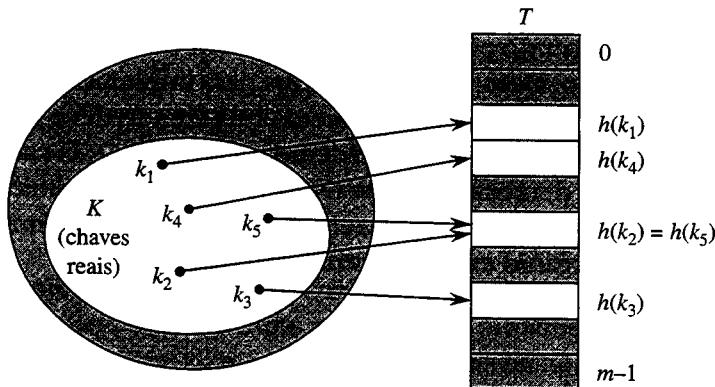


FIGURA 11.2 O uso de uma função hash h para mapear chaves como posições de uma tabela hash. As chaves k_2 e k_5 são mapeadas para a mesma posição, e assim elas colidem

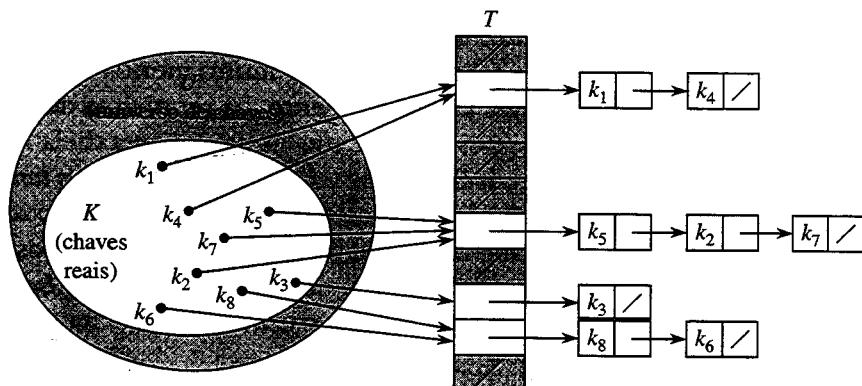


FIGURA 11.3 Resolução de colisões por encadeamento. Cada posição $T[j]$ da tabela hash contém uma lista ligada de todas as chaves cujo valor hash é j . Por exemplo, $b(k_1) = b(k_4)$ e $b(k_5) = b(k_2) = b(k_7)$

Resolução de colisões por encadeamento

No **encadeamento**, colocamos todos os elementos que efetuam hash para a mesma posição em uma lista ligada, como mostra a Figura 11.3. A posição j contém um ponteiro para o início da lista de todos os elementos armazenados que efetuam hash para j ; se não houver nenhum desses elementos, a posição j conterá NIL.

As operações de dicionário sobre uma tabela hash T são fáceis de implementar quando as colisões são resolvidas por encadeamento.

CHAINED-HASH-INSERT(T, x)

insere x no início da lista $T[b(chave[x])]$

CHAINED-HASH-SEARCH(T, k)

procura por um elemento com a chave k na lista $T[b(k)]$

CHAINED-HASH-DELETE(T, x)
elimina x da lista $T[b(chave[x])]$

O tempo de execução no pior caso para a inserção é $O(1)$. No caso da pesquisa, o tempo de execução no pior caso é proporcional ao comprimento da lista; analisaremos esse assunto com mais detalhes em seguida. A eliminação de um elemento x pode ser alcançada no tempo $O(1)$ se as listas forem duplamente ligadas. (Se as listas são simplesmente ligadas, primeiro devemos encontrar x na lista $T[b(chave[x])]$, de forma que o *próximo* vínculo do predecessor de x possa ser definido de modo apropriado para unir x ; nesse caso, a eliminação e a pesquisa terão essencialmente o mesmo tempo de execução.)

Análise do hash com encadeamento

Qual é a qualidade da execução do hash com encadeamento? Em particular, quanto tempo ele leva para procurar por um elemento com uma determinada chave?

Dada uma tabela hash T com m posições que armazena n elementos, definimos o **fator de carga** α para T como n/m , isto é, o número médio de elementos armazenados em uma cadeia. Nossa análise será em termos de α , que pode ser menor que, igual a ou maior que 1.

O comportamento no pior caso do hash com encadeamento é terrível: todas as n chaves executam o hash na mesma posição, criando uma lista de comprimento n . Portanto, o tempo no pior caso para a pesquisa é (n) mais o tempo necessário para calcular a função hash – não melhor do que seria se usássemos uma lista ligada para todos os elementos. É evidente que as tabelas hash não são usadas por seu desempenho no pior caso. (O hash perfeito, descrito na Seção 11.5, oferece porém um bom desempenho no pior caso quando o conjunto de chaves é estático.)

O desempenho médio do hash depende de como a função hash b distribui o conjunto de chaves a serem armazenadas entre as m posições, em média. A Seção 11.3 discute essas questões, mas, por enquanto, devemos supor que qualquer elemento dado tem igual probabilidade de efetuar o hash para qualquer das m posições, independentemente de onde qualquer outro elemento tenha efetuado o hash. Chamamos essa suposição de hipótese de **hash uniforme simples**.

Para $j = 0, 1, \dots, m - 1$, vamos denotar o comprimento da lista $T[j]$ por n_j , de forma que

$$n = n_0 + n_1 + \dots + n_{m-1}, \quad (11.1)$$

e o valor médio de n_j é $E[n_j] = \alpha = n/m$.

Supomos que o valor hash $b(k)$ pode ser calculado no tempo $O(1)$, de modo que o tempo necessário para procurar por um elemento com chave k depende linearmente do comprimento $n_{b(k)}$ da lista $T[b(k)]$. Deixando de lado o tempo $O(1)$ necessário para calcular a função hash e obter acesso à posição $b(k)$, vamos considerar o número esperado de elementos examinados pelo algoritmo de pesquisa, ou seja, o número de elementos na lista $T[b(k)]$ que são examinados para ver se suas chaves são iguais a k . Consideraremos dois casos. No primeiro, a pesquisa não é bem-sucedida: nenhum elemento na tabela tem a chave k . No segundo caso, a pesquisa tem sucesso, encontrando um elemento com chave k .

Teorema 11.1

Em uma tabela hash na qual as colisões são resolvidas por encadeamento, uma pesquisa malsucedida demora o tempo esperado $\Theta(1 + \alpha)$, sob a hipótese de hash uniforme simples.

Prova Sob a hipótese de hash uniforme simples, qualquer chave k ainda não armazenada na tabela tem igual probabilidade de efetuar o hash para qualquer das m posições. O tempo médio para pesquisa sem sucesso para uma chave k é, portanto, o tempo esperado para pesquisar até o

fim da lista $T[b(k)]$, que tem o comprimento esperado $E[n_{b(k)}] = \alpha$. Assim, o número esperado de elementos examinados em uma pesquisa malsucedida é α , e o tempo total necessário (incluindo o tempo para se calcular $b(k)$) é $\Theta(1 + \alpha)$. ■

A situação para uma pesquisa bem-sucedida é ligeiramente diferente, pois cada lista não tem igual probabilidade de ser pesquisada. Em vez disso, a probabilidade de uma lista ser pesquisada é proporcional ao número de elementos que ela contém. Todavia, o tempo de pesquisa esperado ainda é $\Theta(1 + \alpha)$.

Teorema 11.2

Em uma tabela hash na qual as colisões são resolvidas por encadeamento, uma pesquisa bem-sucedida demora o tempo $\Theta(1 + \alpha)$, na média, sob a hipótese de hash uniforme simples.

Prova Vamos supor que o elemento que está sendo pesquisado tem igual probabilidade de ser qualquer dos n elementos armazenados na tabela. O número de elementos examinados durante uma pesquisa bem-sucedida para um elemento x é uma unidade maior que o número de elementos que aparecem antes de x na lista de x . Os elementos antes de x foram todos inseridos após x ser inserido, porque novos elementos são colocados no início da lista. Para encontrar o número esperado de elementos examinados, tomamos a média, sobre os n elementos x na tabela, de 1 mais o número esperado de elementos adicionados à lista de x depois que x foi adicionado à lista. Seja x_i o i -ésimo elemento inserido na tabela, para $i = 1, 2, \dots, n$, e seja $k_i = chave[x_i]$. Para chaves k_i e k_j , definimos a variável indicadora aleatória $X_{ij} = I\{b(k_i) = b(k_j)\}$. Sob a hipótese de hash uniforme simples, temos $\Pr\{b(k_i) = b(k_j)\}$ e então, pelo Lema 5.1, $E[X_{ij}] = 1/m$. Desse modo, o número esperado de elementos examinados em uma pesquisa bem-sucedida é

$$\begin{aligned}
 E & \left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij} \right) \right] \\
 &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}] \right) \text{ (por linearidade de expectativa)} \\
 &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m} \right) \\
 &= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) \\
 &= 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i \right) \\
 &= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right) \text{ (pela equação (A.1))} \\
 &= 1 + \frac{n-1}{2m} \\
 &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
 \end{aligned}$$

Desse modo, o tempo total exigido para uma pesquisa bem-sucedida (incluindo o tempo para calcular a função hash) é $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

O que significa essa análise? Se o número de posições da tabela hash é no mínimo proporcional ao número de elementos na tabela, temos $n = O(m)$ e, consequentemente, $\alpha = n/m = O(m)/m = O(1)$. Assim, a pesquisa demora um tempo constante em média. Tendo em vista que a

inserção demora o tempo $O(1)$ no pior caso e a eliminação demora o tempo $O(1)$ no pior caso quando as listas são duplamente ligadas, todas as operações de dicionário podem ser admitidas no tempo $O(1)$ em média.

Exercícios

11.2-1

Vamos supor que utilizamos uma função hash b para efetuar o hash de n chaves distintas em um arranjo T de comprimento m . Supondo-se hash uniforme simples, qual é o número esperado de colisões? Mais precisamente, qual é a cardinalidade esperada de $\{ \{k, l\} : k \neq l \text{ e } b(k) = b(l) \}$?

11.2-2

Demonstre a inserção das chaves 5, 28, 19, 15, 20, 33, 12, 17, 10 em uma tabela hash com colisões resolvidas por encadeamento. Seja a tabela com 9 posições, e seja a função hash $b(k) = k \bmod 9$.

11.2-3

O professor Marley apresenta a hipótese de que é possível obter ganhos substanciais de desempenho se modificarmos o esquema de encadeamento de tal modo que cada lista seja mantida em seqüência ordenada. Como a modificação do professor afeta o tempo de execução para pesquisas bem-sucedidas, pesquisas sem sucesso, inserções e eliminações?

11.2-4

Sugira como o armazenamento para elementos pode ser alocado e desalocado dentro da própria tabela hash, através da vinculação de todas as posições não utilizadas em uma lista livre. Suponha que uma posição possa armazenar um sinalizador e um elemento mais um ponteiro ou dois ponteiros. Todas as operações de dicionário e de lista livre devem ser executadas no tempo esperado $O(1)$. A lista livre precisa ser duplamente ligada, ou seria suficiente uma lista livre simplesmente ligada?

11.2-5

Mostre que, se $|U| > nm$, existe um subconjunto de U de tamanho n consistindo em chaves que efetuam o hash todas para a mesma posição, de tal forma que o tempo de pesquisa do pior caso para o hash com encadeamento é (n) .

11.3 Funções hash

Nesta seção, discutiremos algumas questões relacionadas ao projeto de funções hash de boa qualidade, e depois apresentaremos três esquemas para sua criação. Dois dos esquemas, o hash por divisão e o hash por multiplicação, são heurísticos por natureza, enquanto o terceiro esquema, o hash universal, utiliza a aleatoriedade para oferecer um desempenho que se pode demonstrar ser bom.

O que faz uma função hash de boa qualidade?

Uma função hash de boa qualidade satisfaz (aproximadamente) à hipótese do hash uniforme simples: cada chave tem igual probabilidade de efetuar o hash para qualquer das m posições, não importando a posição para onde foi feito o hash de qualquer outra chave. Infelizmente, em geral não é possível verificar essa condição, pois é raro conhecer a distribuição de probabilidades segundo a qual as chaves são obtidas, e as chaves não podem ser obtidas de forma independente.

Ocasionalmente, conhecemos a distribuição. Por exemplo, suponha que as chaves sejam conhecidas como números reais aleatórios k , independente e uniformemente distribuídos no intervalo $0 \leq k < 1$. Nesse caso, podemos mostrar que a função hash

$$h(k) = \lfloor km \rfloor$$

satisfaz à condição de hash uniforme simples.

Na prática, podem ser usadas técnicas heurísticas para criar uma função hash que provavelmente terá bom desempenho. Informações qualitativas sobre a distribuição de chaves podem ser úteis nesse processo de projeto. Por exemplo, considere a tabela de símbolos de um compilador, na qual as chaves são cadeias de caracteres arbitrários que representam identificadores em um programa. É comum que símbolos intimamente relacionados, como `pt` e `pts`, ocorram no mesmo programa. Uma função hash de boa qualidade minimizaria a chance de que tais variações efetuassem hash para a mesma posição.

Uma boa abordagem é derivar a tabela hash de um modo supostamente independente de quaisquer padrões que pudesse existir nos dados. Por exemplo, o “método de divisão” (discutido na Seção 11.3.1) calcula o valor hash como o resto quando a chave é dividida por um número primo especificado. Esse método com freqüência fornece bons resultados, supondo-se que o número primo escolhido não esteja relacionado a quaisquer padrões na distribuição de chaves.

Finalmente, observamos que algumas aplicações das funções hash poderiam exigir propriedades mais fortes que aquelas oferecidas pelo hash uniforme simples. Por exemplo, poderíamos desejar chaves que fossem mais “próximas” em algum sentido, a fim de formar valores hash bastante afastados entre si. (Essa propriedade é especialmente desejável quando estamos usando a sondagem linear, definida na Seção 11.4.) O hash universal, descrito na Seção 11.3.3, com freqüência fornece as propriedades desejadas.

Interpretação de chaves como números naturais

A maior parte das funções hash supõe que o universo de chaves é o conjunto $\mathbb{N} = \{0, 1, 2, \dots\}$ de números naturais. Desse modo, se as chaves não são números naturais, deve-se encontrar um modo de interpretá-las como números naturais. Por exemplo, uma cadeia de caracteres pode ser interpretada como um inteiro expresso em uma notação de raiz apropriada. Assim, o identificador `pt` poderia ser interpretado como o par de inteiros decimais $(112, 116)$, pois $p = 112$ e $t = 116$ no conjunto de caracteres ASCII; então, expresso como um inteiro de raiz 128, `pt` se torna $(112 \cdot 128) + 116 = 14452$. Em geral, é uma tarefa objetiva em qualquer aplicação dada elaborar algum método simples como esse para interpretar cada chave como um número natural (possivelmente grande). No texto a seguir, supomos que as chaves são números naturais.

11.3.1 O método de divisão

No **método de divisão** para criação de funções hash, mapeamos uma chave k para uma de m posições, tomando o resto de k dividido por m . Ou seja, a função hash é

$$h(k) = k \bmod m .$$

Por exemplo, se a tabela hash tem tamanho $m = 12$ e a chave é $k = 100$, então $h(k) = 4$. Pelo fato de só exigir uma única operação de divisão, o hash por divisão é bastante rápido.

Quando se utiliza o método de divisão, em geral se evitam certos valores de m . Por exemplo, m não deve ser uma potência de 2 pois, se $m = 2^p$, então $h(k)$ será somente o grupo de p bits de mais baixa ordem de k . A menos que seja conhecido *a priori* que a distribuição de probabilidades sobre chaves torna todos os padrões de p bits de mais baixa ordem igualmente prováveis, é melhor fazer a função hash depender de todos os bits da chave. Como o Exercício 11.3-3 lhe pede para mostrar, a escolha de $m = 2^p - 1$ quando k é uma cadeia de caracteres interpretada em raiz 2^p pode ser uma escolha ruim, porque a permutação de caracteres de k não altera seu valor hash.

Um primo não muito próximo a uma potência exata de 2 freqüentemente é uma boa escolha para m . Por exemplo, suponha que desejamos alocar uma tabela hash, com colisões resolvidas por encadeamento, para conter aproximadamente $n = 2000$ cadeias de caracteres, onde um caractere tem 8 bits. Não nos importamos de examinar uma média de 3 elementos em uma pesquisa malsucedida, e assim alocamos uma tabela hash de tamanho $m = 701$. O número 701 foi escondido porque é um primo próximo a $2000/3$, mas não próximo a qualquer potência de 2. Tratando cada chave k como um inteiro, nossa função hash seria

$$b(k) = k \bmod 701.$$

11.3.2 O método de multiplicação

O **método de multiplicação** para criação de funções hash opera em duas etapas. Primeiro, multiplicamos a chave k por uma constante A no intervalo $0 < A < 1$ e extraímos a parte fracionária de kA . Em seguida, multiplicamos esse valor por m e tomamos o piso do resultado. Em suma, a função hash é

$$b(k) = \lfloor m(kA \bmod 1) \rfloor,$$

onde “ $kA \bmod 1$ ” significa a parte fracionária de kA , ou seja, $kA - \lfloor kA \rfloor$.

Uma vantagem do método de multiplicação é que o valor de m não é crítico. Em geral, nós escolhemos de modo a ser uma potência de 2 ($m = 2^p$ para algum inteiro p) pois podemos implementar facilmente a função na maioria dos computadores como a seguir. Suponha que o tamanho da palavra da máquina seja w bits e que k se encaixe em uma única palavra. Restringimos A a ser uma fração da forma $s/2^w$, onde s é um inteiro no intervalo $0 < s < 2^w$. Consultando a Figura 11.4, primeiro multiplicamos k pelo inteiro de w bits $s = A \cdot 2^w$. O resultado é um valor de $2w$ bits $r_1 2^w + r_0$, onde r_1 é a palavra de alta ordem do produto e r_0 é a palavra de baixa ordem do produto. O valor hash de p bits desejado consiste nos p bits mais significativos de r_0 .

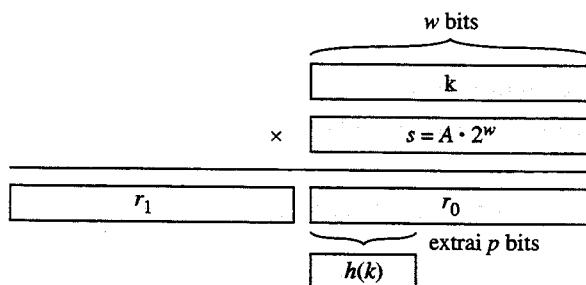


FIGURA 11.4 O método de multiplicação hash. A representação de w bits da chave k é multiplicada pelo valor de w bits $s = A \cdot 2^w$, onde $0 < A < 1$ é uma constante adequada. Os p bits de mais alta ordem da metade inferior de w bits do produto formam o valor hash desejado $b(k)$

Embora esse método funcione com qualquer valor da constante A , ele funciona melhor com alguns valores que com outros. A escolha ótima depende das características dos dados sobre os quais está sendo feito o hash. Knuth [185] sugere que

$$A \approx (\sqrt{5} - 1)/2 = 0,6180339887... \quad (11.2)$$

deverá funcionar razoavelmente bem.

Como exemplo, suponha que temos $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ e $w = 32$. Adap-tando a sugestão de Knuth, escolhemos A como a fração da forma $s/2^{32}$ mais próxima a $(\sqrt{5} - 1)/2$, de forma que $A = 2654435769/2^{32}$. Então, $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, e assim $r_1 = 76300$ e $r_0 = 17612864$. Os 14 bits mais significativos de r_0 formam o va-lor $b(k) = 67$.

★ 11.3.3 Hash universal

Se um adversário malicioso escolher as chaves que deverão sofrer o hash de alguma função hash fixa, então ele poderá escolher n chaves que façam todos o hash para a mesma posição, resultan-do em um tempo médio de recuperação igual a $\Theta(n)$. Qualquer função hash fixa é vulnerável a essa espécie terrível de comportamento do pior caso; a única maneira eficaz de melhorar a situa-ção é escolher a função hash *aleatoriamente*, de um modo que seja *independente* das chaves que na realidade estão sendo armazenadas. Essa abordagem, chamada **hash universal**, pode resultar em um desempenho provavelmente bom na média, não importando as chaves que são escolhidas pelo adversário.

A idéia principal por trás do hash universal é selecionar a função hash ao acaso a partir de uma classe de funções cuidadosamente projetada no início a execução. Como no caso de quick-sort, a aleatoriedade garante que nenhuma entrada isolada evocará sempre o comportamento do pior caso. Em virtude da aleatoriedade, o algoritmo poderá se comportar de modo diferente em cada execução, ainda que para a mesma entrada, garantindo um bom desempenho no caso médio para qualquer entrada. Retornando ao exemplo da tabela de símbolos de um compila-dor, verificamos que agora a escolha de identificadores pelo programador não pode provocar um desempenho pobre consistente no hash. O desempenho pobre ocorre apenas quando o compilador escolhe uma função hash aleatória que faz o conjunto de identificadores efetuar o hash de modo ruim, mas a probabilidade de ocorrer essa situação é pequena, e é a mesma para qualquer conjunto de identificadores de tamanho idêntico.

Seja \mathcal{H} uma coleção finita de funções hash que mapeiam um dado universo U de chaves no intervalo $\{0, 1, \dots, m - 1\}$. Essa coleção é dita ***universal*** se, para cada par de chaves distintas $k, l \in U$, o número de funções hash \mathcal{H} para as quais $b(k) = b(l)$ é no máximo \mathcal{H} . Em outras palavras, com uma função hash escolhida aleatoriamente a partir de \mathcal{H} , a chance de uma colisão entre cha-ves distintas k e l não é maior que a chance $1/m$ de uma colisão se $b(k)$ e $b(l)$ fossem escolhidas aleatória e independentemente a partir do conjunto $\{0, 1, \dots, m - 1\}$.

O teorema a seguir mostra que uma classe universal de funções hash oferece bom comporta-mento no caso médio. Lembre-se de que n_i denota o comprimento da lista $T[i]$.

Teorema 11.3

Suponha que uma função hash b seja escolhida a partir de uma coleção universal de funções hash e seja usada para efetuar o hash de n chaves em uma tabela T de tamanho m , usando o enca-deamento para resolver colisões. Se a chave k não está na tabela, então o comprimento esperado $E[n_{b(k)}]$ da lista para a qual a chave k efetua o hash é no máximo α . Se a chave k está na tabela, en-tão o comprimento esperado $E[n_{b(k)}]$ da lista que contém a chave k é no máximo $1 + \alpha$.

Prova Notamos que as expectativas aqui estão acima da escolha da função hash e não depen-dem de quaisquer hipóteses sobre a distribuição das chaves. Para cada par k e l de chaves distin-tas, defina a variável indicadora aleatória $X_{kl} = I\{b(k) = b(l)\}$. Tendo em vista que, por defini-ção, um único par de chaves colide com probabilidade no máximo $1/m$, temos $\Pr\{b(k) = b(l)\} \leq 1/m$ e, assim, o Lema 5.1 implica que $E[X_{kl}] \leq 1/m$.

Em seguida definimos, para cada chave k , a variável aleatória Y_k que é igual ao número de chaves diferentes de k que efetuam o hash para a mesma posição que k , de forma que

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}.$$

Portanto, temos

$$\begin{aligned}
 E[Y_k] &= E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\
 &= \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}] \text{ (por linearidade de expectativa)} \\
 &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}.
 \end{aligned}$$

O restante da prova depende do fato da chave k estar ou não na tabela T .

- Se $k \notin T$, então $n_{b(k)} = Y_k$ e $|\{l : l \in T \text{ e } l \neq k\}| = n$. Desse modo $E[n_{b(k)}] = E[Y_k] \leq n/m = \alpha$.
- Se $k \in T$ então, como a chave k aparece na lista $T[b(k)]$ e a contagem Y_k não inclui a chave k , temos $n_{b(k)} = Y_k + 1$ e $|\{l : l \in T \text{ e } l \neq k\}| = n - 1$. Assim, $E[n_{b(k)}] = E[Y_k] \leq (n - 1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$.

O corolário a seguir nos diz que o hash universal fornece a compensação desejada: agora é impossível um adversário escolher uma seqüência de operações que force o tempo de execução do pior caso. Tornando habilmente aleatória a escolha da função hash em tempo de execução, garantimos que toda seqüência de operações pode ser tratada com um bom tempo de execução esperado.

Corolário 11.4

Usando-se o hash universal e a resolução de colisões pelo encadeamento em uma tabela com m posições, demora o tempo esperado $\Theta(n)$ tratar qualquer seqüência de n operações INSERT, SEARCH e DELETE contendo $O(m)$ operações INSERT.

Prova Como o número de inserções é $O(m)$, temos $n = O(m)$, e assim $\alpha = O(1)$. As operações INSERT e DELETE demoram um tempo constante e, pelo Teorema 11.3, o tempo esperado para cada operação SEARCH é $O(1)$. Desse modo, por linearidade de expectativa, o tempo esperado para a seqüência de operações inteira é $O(n)$.

O projeto de uma classe universal de funções hash

É bastante fácil projetar uma classe universal de funções hash, como um pouco de teoria dos números nos ajudará a demonstrar. Talvez você queira primeiro consultar o Capítulo 31, se estiver pouco familiarizado com a teoria dos números.

Vamos começar escolhendo um número primo p grande o bastante para que toda chave k possível esteja no intervalo 0 a $p - 1$, inclusive. Seja Z_p^* o conjunto $\{0, 1, \dots, p - 1\}$, e seja Z_p o conjunto $\{1, 2, \dots, p - 1\}$. Tendo em vista que p é primo, podemos resolver equações de módulo p com os métodos dados no Capítulo 31. Como supomos que o tamanho do universo de chaves é maior que o número de posições na tabela hash, temos $p > m$.

Agora definimos a função hash $b_{a,b}$ para qualquer $a \in Z_p^*$ e qualquer $b \in Z_p$, usando uma transformação linear seguida por reduções de módulo p , e depois de módulo m :

$$b_{a,b}(k) = ((ak + b) \bmod p) \bmod m. \quad (11.3)$$

Por exemplo, com $p = 17$ e $m = 6$, temos $b_{3,4}(8) = 5$. A família de todas essas funções hash é

$$\mathcal{H}_{p,m} = \{b_{a,b} : a \in Z_p^* \text{ e } b \in Z_p\}. \quad (11.4) \quad |_{189}$$

Cada função hash $b_{a,b}$ mapeia \mathbf{Z}_p para \mathbf{Z}_m . Essa classe de funções hash tem a interessante propriedade de que o tamanho m do intervalo de saída é arbitrário – não necessariamente primo –, uma característica que usaremos na Seção 11.5. Tendo em vista que existem $p-1$ escolhas para a e que há p escolhas para b , existem $p(p-1)$ funções hash em $\mathcal{H}_{p,m}$.

Teorema 11.5

A classe $\mathcal{H}_{p,m}$ de funções hash definida pelas equações (11.3) e (11.4) é universal.

Prova Considere duas chaves distintas k e l de \mathbf{Z}_p , de forma que $k \neq l$. Para uma dada função hash $b_{a,b}$ fazemos

$$\begin{aligned} r &= (ak + b) \bmod p, \\ s &= (al + b) \bmod p. \end{aligned}$$

Primeiro observamos que $r \neq s$. Por quê? Observe que

$$r - s \equiv a(k - l) \pmod{p}.$$

Segue-se que $r \neq s$ porque p é primo e tanto a quanto $(k-l)$ são diferentes de zero em módulo p , e assim seu produto também deve ser diferente de zero em módulo p , pelo Teorema 31.6. Então, durante a computação de qualquer $b_{a,b}$ em $\mathcal{H}_{p,m}$, entradas distintas k e l mapeiam para valores distintos r e s em módulo p ; ainda não há nenhuma colisão no “nível de mod p ”. Além disso, cada uma das $p(p-1)$ escolhas possíveis para o par (a, b) com $a \neq 0$ produz um par resultante (r, s) diferente com $r \neq s$, pois podemos resolver para a e b dado r e s :

$$\begin{aligned} a &= ((r - s)((k - l)^{-1} \bmod p)) \bmod p, \\ b &= (r - ak) \bmod p, \end{aligned}$$

onde $((k - l)^{-1} \bmod p)$ denota o inverso multiplicativo único em módulo p , de $k - l$. Como existem apenas $p(p-1)$ pares (r, s) possíveis com $r \neq s$, há uma correspondência de um para um entre pares (a, b) com $a \neq 0$ e pares (r, s) com $r \neq s$. Desse modo, para qualquer par de entradas k e l dado, se escolhermos (a, b) uniformemente ao acaso de $\mathbf{Z}_p^* \times \mathbf{Z}_p$, o par resultante (r, s) terá igual probabilidade de ser qualquer par de valores distintos em módulo p .

Então, segue-se que a probabilidade de que chaves distintas k e l colidam é igual à probabilidade de que $r = s$ quando r e s são escolhidos ao acaso como valores distintos em módulo p . Para um dado valor de r , dos $p-1$ valores restantes possíveis para s , o número de valores s tais que $s \neq r$ e $s \equiv r$ é no máximo

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p+m-1)/m) - 1 \quad (\text{pela desigualdade (3.7)}) \\ &= (p-1)/m. \end{aligned}$$

A probabilidade de s colidir com r quando reduzido em módulo m é no máximo $((p-1)/m)/(p-1) = 1/m$.

Assim, para qualquer par de valores distintos $k, l \in \mathbf{Z}_p$,

$$\Pr\{b_{a,b}(k) = b_{a,b}(l)\} \leq 1/m,$$

de forma que $\mathcal{H}_{p,m}$ é realmente universal.

Exercícios

11.3-1

Suponha que desejamos pesquisar uma lista ligada de comprimento n , onde cada elemento contém um chave k juntamente com um valor hash $b(k)$. Cada chave é uma cadeia de caracteres longa. Como poderíamos tirar proveito dos valores hash ao procurar na lista por um elemento com uma chave específica?

11.3-2

Vamos supor que uma cadeia de r caracteres sofra hash em m posições, tratando-a como um número de raiz 128, e depois usando o método de divisão. O número m é facilmente representado como uma palavra de computador de 32 bits, mas a cadeia de r caracteres, tratada como um número de raiz 128, ocupa muitas palavras. Como podemos aplicar o método de divisão para calcular o valor hash da cadeia de caracteres sem usar mais de um número constante de palavras de espaço de armazenamento fora da própria cadeia?

11.3-3

Considere uma versão do método de divisão, na qual $b(k) = k \bmod m$, onde $m = 2^p - 1$ e k é uma cadeia de caracteres interpretada na raiz 2^p . Mostre que, se a cadeia x puder ser derivada da cadeia y pela permutação de seus caracteres, então x e y terão hash para o mesmo valor. Forneça um exemplo de uma aplicação na qual essa propriedade seria indesejável em uma função hash.

11.3-4

Considere uma tabela hash de tamanho $m = 1000$ e a função hash correspondente $b(k)$ igual a $\lfloor m(kA \bmod 1) \rfloor$ para $A = (\sqrt{5} - 1)/2$. Calcule as localizações para as quais as chaves 61, 62, 63, 64 e 65 estão mapeadas.

11.3-5 *

Defina uma família \mathcal{H} de funções hash a partir de um conjunto finito U para um conjunto finito B como ϵ -universal se, para todos os pares de elementos distintos k e l em U ,

$$\Pr \{b(k) = b(l)\} \leq \epsilon$$

onde a probabilidade é obtida sobre a definição da função hash b ao acaso a partir da família \mathcal{H} . Mostre que uma família ϵ -universal de funções hash deve ter

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

11.3-6 *

Seja U o conjunto de n -tuplas de valores obtidos a partir de \mathbf{Z}_p , e seja $B = \mathbf{Z}_p$, onde p é primo. Defina a função hash $b_b : U \rightarrow B$ para $b \in \mathbf{Z}_p$ sobre n -tupla de entrada $\langle a_0, a_1, \dots, a_{n-1} \rangle$ a partir de U como

$$b_b(\langle a_0, a_1, \dots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j$$

e seja $\mathcal{H} = \{b_b : b \in \mathbf{Z}_p\}$. Demonstre que \mathcal{H} é $((n-1)/p)$ -universal, de acordo com a definição de ϵ -universal no Exercício 11.3-5. (Sugestão: Ver Exercício 31.4-4.)

11.4 Endereçamento aberto

No **endereçamento aberto**, todos os elementos estão armazenados na própria tabela hash. Ou seja, cada entrada da tabela contém um elemento do conjunto dinâmico ou NIL. Ao procurar por um elemento, examinamos sistematicamente as posições da tabela até encontrarmos o elemento desejado, ou até ficar claro que o elemento não está na tabela. Não existe nenhuma lista e nenhum elemento armazenado fora da tabela, como há no encadeamento. Desse modo, no endereçamento aberto, a tabela hash pode “ficar cheia”, de tal forma que não podem ser feitas inserções adicionais; o fator de carga α nunca pode exceder 1.

É claro que poderíamos armazenar as listas ligadas para encadeamento no interior da tabela hash, nas posições da tabela hash não utilizadas de outro modo (ver Exercício 11.2-4), mas a vantagem do endereçamento aberto é que ele evita por completo o uso de ponteiros. Em lugar de seguir ponteiros, *calculamos* a seqüência de posições a serem examinadas. A memória extra liberada por não se armazenarem ponteiros fornece à tabela hash um número maior de posições para a mesma quantidade de memória, gerando potencialmente menor número de colisões e recuperação mais rápida.

Para executar a inserção usando o endereçamento aberto, examinamos sucessivamente, ou *sondamos*, a tabela hash até encontrarmos uma posição vazia na qual seja possível inserir a chave. Em lugar de ser fixada na ordem 0, 1, ..., $m - 1$ (o que exige o tempo de pesquisa $\Theta(n)$), a seqüência de posições demonstrada *depende da chave que está sendo inserida*. Para determinar quais serão as posições sondadas, estendemos a função hash com o objetivo de incluir o número de sondagens (a partir de 0) como uma segunda entrada. Desse modo, a função hash se torna

$$b : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

Com o endereçamento aberto, exigimos que, para toda chave k , a *seqüência de sondagem* $\langle b(k, 0), b(k, 1), \dots, b(k, m - 1) \rangle$

seja uma permutação de $\langle 0, 1, \dots, m - 1 \rangle$, de modo que toda posição da tabela hash seja eventualmente considerada uma posição para uma nova chave, à medida que a tabela é preenchida. No pseudocódigo a seguir, supomos que os elementos na tabela hash T são chaves sem informações satélite; a chave k é idêntica ao elemento que contém a chave k . Cada posição contém uma chave ou NIL (se a posição é vazia).

```
HASH-INSERT( $T, k$ )
1  $i \leftarrow 0$ 
2 repeat  $j \leftarrow b(k, i)$ 
3   if  $T[j] = \text{NIL}$ 
4     then  $T[j] \leftarrow k$ 
5     return  $j$ 
6   else  $i \leftarrow i + 1$ 
7 until  $i = m$ 
8 error “hash table overflow”
```

O algoritmo para procurar pela chave k efetua a sondagem da mesma seqüência de posições que o algoritmo de inserção examinado quando a chave k foi inserida. Portanto, a pesquisa pode terminar (sem sucesso) ao encontrar uma posição vazia, pois k teria sido inserido ali e não mais adiante em sua seqüência de sondagem. (Esse argumento pressupõe que as chaves não são eliminadas da tabela hash.) O procedimento HASH-SEARCH toma como entrada uma tabela hash T e um chave k , retornando j se a posição j contendo a chave k é encontrada, ou retornando NIL se a chave k não está presente na tabela T .

```

HASH-SEARCH( $T, k$ )
1  $i \leftarrow 0$ 
2 repeat  $j \leftarrow b(k, i)$ 
3   if  $T[j] = k$ 
4     then return  $j$ 
5    $i \leftarrow i + 1$ 
6 until  $T[j] = \text{NIL}$  or  $i = m$ 
7 return  $\text{NIL}$ 

```

A eliminação de uma tabela hash de endereço aberto é difícil. Quando eliminamos uma chave da posição i , não podemos simplesmente assinalar essa posição como vazia, armazenando NIL em seu interior. Fazer isso poderia tornar impossível recuperar qualquer chave k , se durante a inserção dessa chave tivéssemos sondado a posição i e encontrado essa posição ocupada. Uma solução é assinalar a posição armazenando nela o valor especial DELETED em lugar de NIL. Então, modificariam o procedimento HASH-INSERT para tratar tal posição como se ela estivesse vazia, de modo que uma nova chave possa ser inserida. Nenhuma modificação de HASH-SEARCH é necessária, pois ele passará sobre valores DELETED enquanto estiver pesquisando. Entretanto, quando usamos o valor especial DELETED, os tempos de pesquisa não são mais dependentes do fator de carga α . Por essa razão, o encadeamento é mais comumente selecionado como uma técnica de resolução de colisões quando surge a necessidade de eliminar chaves.

Em nossa análise, fazemos a suposição de **hash uniforme**: supomos que cada chave considerada tem igual probabilidade de ter qualquer das $m!$ permutações de $\langle 0, 1, \dots, m - 1 \rangle$ como sua sequência de sondagem. O hash uniforme generaliza a noção de hash uniforme simples definida anteriormente para a situação na qual a função hash produz não apenas um número único, mas uma sequência de sondagem inteira. Contudo, o verdadeiro hash uniforme é difícil de implementar e, na prática, são usadas aproximações adequadas (como o hash duplo, definido a seguir).

Três técnicas são usadas comumente para calcular as sequências de sondagem exigidas para o endereçamento aberto: sondagem linear, sondagem quadrática e hash duplo. Todas essas técnicas garantem que $\langle b(k, 0), b(k, 1), \dots, b(k, m - 1) \rangle$ é uma permutação de $\langle 0, 1, \dots, m - 1 \rangle$ para cada chave k . Porém, nenhuma dessas técnicas implementa a hipótese de hash uniforme, pois nenhuma delas é capaz de gerar mais de m^2 sequências de sondagem diferentes (em vez das $m!$ que o hash uniforme exige). O hash duplo tem o maior número de sequências de sondagem e, como se poderia esperar, parece proporcionar os melhores resultados.

Sondagem linear

Dada uma função hash comum $b': U \rightarrow \{0, 1, \dots, m - 1\}$, à qual nos referimos como uma **função hash auxiliar**, o método de **sondagem linear** usa a função hash

$$b(k, i) = (b'(k) + i) \bmod m$$

para $i = 0, 1, \dots, m - 1$. Dada a chave k , a primeira posição sondada é $T[b'(k)]$, isto é, a posição dada pela função hash auxiliar. Em seguida, sondamos a posição $T[b'(k) + 1]$ e assim por diante até a posição $T[m - 1]$. Depois, voltamos às posições $T[0], T[1], \dots$, até finalmente sondarmos a posição $T[b'(k) - 1]$. Tendo em vista que a posição inicial de sondagem determina toda a sequência de sondagem, só existem m sequências de sondagem distintas.

A sondagem linear é fácil de implementar, mas sofre de um problema conhecido como **agrupamento primário**. Longas sequências de posições ocupadas são construídas, aumentando o tempo médio de pesquisa. Surgem agrupamentos, pois uma posição vazia precedida por i posições completas é preenchida em seguida com probabilidade $(i + 1)/m$. Sequências de posições ocupadas tendem a ficar mais longas, e o tempo médio de pesquisa aumenta.

Sondagem quadrática

A **sondagem quadrática** utiliza uma função hash da forma

$$b(k, i) = (b'(k) + c_1i + c_2i^2) \bmod m , \quad (11.5)$$

onde b' é uma função hash auxiliar, c_1 e $c_2 \neq 0$ são constantes auxiliares e $i = 0, 1, \dots, m - 1$. A posição inicial sondada é $T[b'(k)]$; posições posteriores sondadas são deslocadas por quantidades que dependem de forma quadrática do número da sondagem i . Esse método funciona muito melhor que a sondagem linear mas, para fazer pleno uso da tabela hash, os valores de c_1 , c_2 e m são limitados. O Problema 11-3 mostra um modo de selecionar esses parâmetros. Além disso, se duas chaves têm a mesma posição de sondagem inicial, então suas seqüências de sondagem são iguais, pois $b(k_1, 0) = b(k_2, 0)$ implica $b(k_1, i) = b(k_2, i)$. Essa propriedade conduz a uma forma mais interessante de agrupamento, chamada **agrupamento secundário**. Como na sondagem linear, a sondagem inicial determina a seqüência inteira; assim, apenas m seqüências de sondagem distintas são utilizadas.

Hash duplo

O hash duplo é um dos melhores métodos disponíveis para endereçamento aberto, porque as permutações produzidas têm muitas características de permutações escolhidas aleatoriamente. O **hash duplo** usa uma função hash da forma

$$b(k, i) = (b_1(k) + ib_2(k)) \bmod m ,$$

onde b_1 e b_2 são funções hash auxiliares. A posição inicial sondada é $T[b_1(k)]$; posições de sondagem sucessivas são deslocadas a partir de posições anteriores pela quantidade $b_2(k)$, módulo m . Desse modo, diferente do caso de sondagem linear ou quadrática, aqui a seqüência de sondagem depende da chave k de duas maneiras, pois a posição de sondagem inicial, o deslocamento, ou ambos, podem variar. A Figura 11.5 oferece um exemplo de inserção por hash duplo.

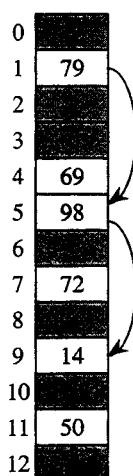


FIGURA 11.5 Inserção por hash duplo. Aqui temos uma tabela hash de tamanho 13 com $b_1(k) = k \bmod 13$ e $b_2(k) = 1 + (k \bmod 11)$. Como $14 \equiv 1 \bmod 13$ e $14 \equiv 3 \bmod 11$, a chave 14 será inserida na posição vazia 9, depois que as posições 1 e 5 tiverem sido examinadas e se descobrir que elas já estão ocupadas

O valor $b_2(k)$ e o tamanho m da tabela hash devem ser primos entre si para que a tabela hash inteira possa ser pesquisada. (Ver Exercício 11.4-3.) Uma forma conveniente de assegurar essa condição é permitir que m seja uma potência de 2 e projetar b_2 de modo que ele sempre produza um número ímpar. Outra maneira é permitir que m seja primo e projetar b_2 de forma que ele sempre retorne um inteiro positivo menor que m . Por exemplo, poderíamos escolher m primo e fazer

$$\begin{aligned} b_1(k) &= k \bmod m, \\ b_2(k) &= 1 + (k \bmod m'), \end{aligned}$$

onde m' é escolhido com um valor ligeiramente menor que m (digamos, $m - 1$). Por exemplo, se $k = 123456$, $m = 701$ e $m' = 700$, temos $b_1(k) = 80$ e $b_2(k) = 257$; assim a primeira sondagem ocorre na posição 80, e depois cada 257-ésima posição (módulo m) é examinada até a chave ser encontrada ou todas as posições serem examinadas.

O hash duplo representa um aperfeiçoamento em relação à sondagem linear ou quadrática, pelo fato de serem usadas $\Theta(m^2)$ seqüências de sondagem, em lugar de $\Theta(m)$, pois cada par $(b_1(k), b_2(k))$ gera uma seqüência de sondagem distinta. Como resultado, o desempenho do hash duplo parece ser muito próximo do desempenho do esquema “ideal” de hash uniforme.

Análise do hash de endereço aberto

Nossa análise de endereçamento aberto, como nossa análise de encadeamento, é expressa em termos do fator de carga $\alpha = n/m$ da tabela hash, à medida que n e m tendem a infinito. É claro que, no caso do endereçamento aberto, temos no máximo um elemento por posição, e portanto $n \leq m$, o que implica $\alpha \leq 1$.

Supomos que seja usado o hash uniforme. Nesse esquema idealizado, a seqüência de sondagem $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$ usada para inserir ou procurar por cada chave k tem igual probabilidade de ser qualquer permutação de $\langle 0, 1, \dots, m - 1 \rangle$. É evidente que uma dada chave tem uma seqüência de sondagem fixa única associada a ela; o que queremos dizer nesse caso é que, considerando a distribuição de probabilidades no espaço de chaves e a operação da função hash sobre as chaves, cada seqüência de sondagem possível é igualmente provável.

Agora, analisamos o número esperado de sondagens para hash com endereçamento aberto, sob a hipótese de hash uniforme, começando com uma análise do número de sondagens realizadas em uma pesquisa malsucedida.

Teorema 11.6

Dada uma tabela hash de endereço aberto com fator de carga $\alpha = n/m < 1$, o número esperado de sondagens em uma pesquisa malsucedida é no máximo $1/(1-\alpha)$, supondo-se o hash uniforme.

Prova Em uma pesquisa malsucedida, toda sondagem exceto a última obtém acesso a uma posição ocupada que não contém a chave desejada, e a última posição sondada está vazia. Vamos definir a variável aleatória X como o número de sondagens feitas em uma pesquisa malsucedida, e vamos também definir o evento A_i , para $i = 1, 2, \dots$, como o evento em que existe uma i -ésima sondagem e ela é para uma posição ocupada. Então, o evento $\{X \geq i\}$ é a interseção dos eventos $A_1 \cap A_2 \cap \dots \cap A_{i-1}$. Limitaremos $\{X \geq i\}$ limitando $\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$. Pelo Exercício C.2-6,

$$\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \dots \Pr\{A_{i-1} | A_1 \cap A_2 \cap \dots \cap A_{i-2}\}.$$

Tendo em vista que existem n elementos e m posições, $\Pr\{A_1\} = n/m$. Para $j > 1$, a probabilidade de existir uma j -ésima sondagem e ela ser para uma posição ocupada, dado que as primeiras $j-1$ sondagens foram para posições ocupadas, é $(n-j+1)/(m-j+1)$. Essa probabilidade se

segue porque estariamos encontrando um dos $(n - (j - 1))$ elementos restantes em uma das $(m - (j - 1))$ posições não examinadas e, pela hipótese de hash uniforme, a probabilidade é a razão entre essas quantidades. Observando que $n < m$ implica $(n - j)/(m - j) \leq n/m$ para todo j tal que $0 \leq j < m$, temos para todo i tal que $1 \leq i \leq m$,

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}.\end{aligned}$$

Agora, usamos a equação (C.24) para limitar o número esperado de sondagens:

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

$$\leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}.$$

O limite anterior $1 + \alpha + \alpha^2 + \alpha^3 + \dots$ tem uma interpretação intuitiva: uma sondagem é sempre realizada. Com probabilidade aproximadamente igual a α , a primeira sondagem encontra uma posição ocupada de forma que é necessária uma segunda sondagem. Com probabilidade aproximadamente igual a α^2 , as duas primeiras posições estão ocupadas, de forma que é necessária uma terceira sondagem e assim por diante.

Se α é uma constante, o Teorema 11.6 prevê que uma pesquisa malsucedida é executada no tempo $O(1)$. Por exemplo, se a tabela hash estiver cheia até a metade, o número médio de sondagens em uma pesquisa malsucedida será $1/(1 - 0,5) = 2$. Se ela estiver 90% cheia, o número de sondagens será no máximo $1/(1 - 0,9) = 10$.

O Teorema 11.6 nos dá quase imediatamente o desempenho do procedimento HASH-INSERT.

Corolário 11.7

A inserção de um elemento em uma tabela hash de endereço aberto com fator de carga α exige no máximo $1/(1 - \alpha)$ sondagens em média, supondo-se o hash uniforme.

Prova Um elemento é inserido apenas se existe espaço na tabela, e portanto $\alpha < 1$. A inserção de uma chave requer uma pesquisa malsucedida seguida pela colocação da chave na primeira posição vazia encontrada. Desse modo, o número esperado de sondagens é no máximo $1/(1 - \alpha)$. ■

O cálculo do número esperado de sondagens para uma pesquisa bem-sucedida exige um pouco mais de trabalho.

Teorema 11.8

Dada uma tabela hash de endereço aberto com fator de carga $\alpha < 1$, o número esperado de sondagens em uma pesquisa bem-sucedida é no máximo

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

supondo-se o hash uniforme e considerando-se que cada chave na tabela tem igual probabilidade de ser pesquisada.

Prova Uma pesquisa de uma chave k segue a mesma seqüência de sondagem que foi seguida quando foi inserido o elemento com chave k . De acordo com o Corolário 11.7, se k foi a $(i+1)$ -ésima chave inserida na tabela hash, o número esperado de sondagens efetuadas em uma procura de k é no máximo $1/(1-i/m) = m/(m-i)$. O cálculo da média sobre todas as n chaves na tabela hash nos dá o número médio de sondagens em uma pesquisa bem-sucedida:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i}$$

$$\frac{1}{\alpha} (H_m - H_{m-n}),$$

onde $H_i = \sum_{j=1}^i 1/j$ é o i -ésimo número harmônico (conforme foi definido na equação (A.7)). Usando a técnica de limitar um somatório por uma integral, descrita na Seção A.2, obtemos

$$\begin{aligned} \frac{1}{\alpha} (H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k} \\ &\leq \frac{1}{\alpha} \int_{m-n}^m (1/x) dx \quad (\text{pela desigualdade (A.12)}) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

para um limite sobre o número esperado de sondagens em uma pesquisa bem-sucedida. ■

Se a tabela hash estiver cheia até a metade, o número esperado de sondagens em uma pesquisa bem-sucedida será menor que 1,387. Se a tabela hash estiver 90% cheia, o número esperado de sondagens será menor que 2,559.

Exercícios

11.4-1

Considere a inserção das chaves 10, 22, 31, 4, 15, 28, 17, 88, 59 em uma tabela hash de comprimento $m = 11$ usando o endereçamento aberto com a função hash primário $b'(k) = k \bmod m$. Ilustre o resultado da inserção dessas chaves com o uso da sondagem linear, empregando a sondagem quadrática com $c_1 = 1$ e $c_2 = 3$, e com a utilização do hash duplo com $b_2(k) = 1 + (k \bmod (m-1))$.

11.4-2

Escreva pseudocódigo para HASH-DELETE da forma descrita no texto e modifique HASH-INSERT para manipular o valor especial DELETED.

11.4-3 *

Suponha que utilizamos o hash duplo para resolver colisões; isto é, usamos a função hash $b(k, i) = (b_1(k) + ib_2(k)) \text{ mod } m$. Mostre que, se m e $b_2(k)$ têm máximo divisor comum $d \geq 1$ para alguma chave k , então uma pesquisa malsucedida para a chave k examina $(1/d)$ -ésimo da tabela hash antes de retornar à posição $b_1(k)$. Desse modo, quando $d = 1$, de forma que m e $b_2(k)$ são primos entre si, a pesquisa pode examinar a tabela hash inteira. (Sugestão: Consulte o Capítulo 31.)

11.4-4

Considere uma tabela hash de endereço aberto com hash uniforme. Forneça limites superiores sobre o número esperado de sondagens em uma pesquisa malsucedida e sobre o número esperado de sondagens em uma pesquisa bem-sucedida quando o fator de carga é $3/4$ e quando ele é $7/8$.

11.4-5 *

Considere uma tabela hash de endereço aberto com um fator de carga α . Encontre o valor α diferente de zero para o qual o número esperado de sondagens em uma pesquisa malsucedida é igual a duas vezes o número esperado de sondagens em uma pesquisa bem-sucedida. Use os limites superiores dados pelos Teoremas 11.6 e 11.8 para esses números esperados de sondagens.

★ 11.5 Hash perfeito

Embora o hash seja usado com maior freqüência por seu excelente desempenho esperado, ele pode ser usado para obter um excelente desempenho *no pior caso*, quando o conjunto de chaves é *estático*: uma vez que as chaves estão armazenadas na tabela, o conjunto de chaves nunca se altera. Algumas aplicações têm naturalmente conjuntos estáticos de chaves: considere o conjunto de palavras reservadas em uma linguagem de programação, ou o conjunto de nomes de arquivos em um CD-ROM. Chamamos uma técnica de hash de **hash perfeito** se o número de acessos de memória exigidos no pior caso para executar uma pesquisa é $O(1)$.

A idéia básica para criar um esquema de hash perfeito é simples. Usamos um esquema de hash de dois níveis com hash universal em cada nível. A Figura 11.6 ilustra o enfoque.

O primeiro nível é essencialmente o mesmo do hash com encadeamento: as n chaves são submetidas ao hash para m posições, com o uso de uma função hash b cuidadosamente selecionada de uma família de funções hash universal.

Porém, em vez de fazer uma lista das chaves que têm hash para a posição j , usamos uma pequena **tabela hash secundário** S_j com uma função hash associada b_j . Escolhendo as funções hash b_j cuidadosamente, podemos garantir que não haverá colisões no nível secundário.

Contudo, para garantir que não haverá nenhuma colisão no nível secundário, precisaremos fazer o tamanho m_j da tabela hash S_j ser o quadrado do número n_j de chaves com hash para a posição j . Embora pareça provável que a existência de tal dependência quadrática de m_j em relação a n_j torne excessivos os requisitos globais de armazenamento, mostraremos que, escolhendo-se bem a função hash de primeiro nível, o espaço total esperado a ser utilizado ainda será $O(n)$.

Usamos funções hash escolhidas a partir das classes universais de funções hash da Seção 1.3.3. A função hash de primeiro nível é escolhida a partir da classe $\mathcal{H}_{p, m}$ onde, como na Seção 1.3.3, p é um número primo maior que qualquer valor de chave. As chaves que efetuam o hash para a posição j têm um novo hash para uma tabela hash secundário S_j de tamanho m_j , com a utilização de uma função hash b_j escolhida a partir da classe \mathcal{H}_{p, m_j} .¹

¹ Quando $n_j = m_j = 1$, não precisamos realmente de uma função hash para a posição j ; quando escolhemos uma função hash $b_{a, b}(k) = ((ak + b) \text{ mod } p) \text{ mod } m_j$ para tal posição, usamos simplesmente $a = b = 0$.

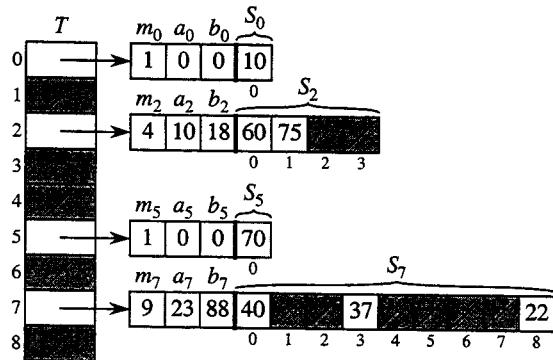


FIGURA 11.6 O uso do hash perfeito para armazenar o conjunto $K = \{10, 22, 37, 40, 60, 70, 75\}$. A função hash exterior é $b(k) = ((ak + b) \bmod p) \bmod m$, onde $a = 3$, $b = 42$, $p = 101$ e $m = 9$. Por exemplo, $b(75) = 2$, então a chave 75 efetua o hash para a posição 2 da tabela T . Uma tabela hash secundário S_j armazena todas as chaves que efetuam o hash para a posição j . O tamanho da tabela hash S_j é m_j , e a função hash associada é $b_j(k) = ((ak + b_j) \bmod p) \bmod m_j$. Tendo em vista que $b_2(75) = 1$, a chave 75 é armazenada na posição 1 da tabela hash secundário S_2 . Não existe nenhuma colisão em quaisquer das tabelas hash secundário, e assim a pesquisa demora um tempo constante no pior caso

Prosseguiremos em duas etapas. Primeiro, vamos determinar como assegurar que as tabelas secundárias não têm nenhuma colisão. Em segundo lugar, mostraremos que a quantidade esperada de memória global utilizada – para a tabela hash primário e todas as tabelas hash secundário – é $O(n)$.

Teorema 11.9

Se armazenarmos n chaves em uma tabela hash de tamanho $m = n^2$ usando uma função hash b escolhida ao acaso de uma classe universal de funções hash, então a probabilidade de haver quaisquer colisões é menor que $\frac{1}{2}$.

Prova Há $\binom{n}{2}$ pares de chaves que podem colidir; cada par colide com probabilidade $1/m$ se b é escolhida ao acaso a partir de uma família universal \mathcal{H} de funções hash. Seja X uma variável aleatória que conta o número de colisões. Quando $m = n^2$, o número esperado de colisões é

$$E[X] = \binom{n}{2} \cdot \frac{1}{n^2}$$

$$= \frac{n^2 - n}{2} \cdot \frac{1}{n^2}$$

$$< 1/2.$$

(Observe que essa análise é semelhante à análise do paradoxo do aniversário na Seção 5.4.1.) A aplicação da desigualdade de Markov (C.29), $\Pr\{X \geq t\} \leq E[X]/t$, com $t = 1$ completa a prova. ■

Na situação descrita no Teorema 11.9, $m = n^2$, segue-se que uma função hash b escolhida ao acaso a partir de \mathcal{H} tem maior probabilidade de não ter *nenhuma* colisão. Dado o conjunto K de n chaves para hash (lembre-se de que K é estático), é fácil encontrar então uma função hash b livre de colisões com algumas experiências aleatórias.

Contudo, quando n é grande, uma tabela hash de tamanho $m = n^2$ é excessiva. Assim, adotamos a abordagem de hash de dois níveis e usamos o enfoque do Teorema 11.9 apenas para o hash das entradas em cada posição. Uma função hash b exterior, ou de primeiro nível, é usada para efetuar o hash das chaves em $m = n$ posições. Então, se n chaves têm hash para a posição j ,

é usada uma tabela hash secundário S_j de tamanho $m_j = n_j^2$ para proporcionar uma pesquisa de tempo constante livres de colisões.

Agora, vamos tratar da questão de assegurar que a memória global usada é $O(n)$. Como o tamanho m_j da j -ésima tabela hash secundário cresce de forma quadrática com o número n_j de chaves armazenadas, existe o risco de que a quantidade global de armazenamento possa ser excessiva.

Se o tamanho da tabela de primeiro nível é $m = n$, então a quantidade de memória usada é $O(n)$ para a tabela hash primário, para o armazenamento dos tamanhos m_j das tabelas hash secundário e para o armazenamento dos parâmetros a_j e b_j que definem as funções hash secundário b_j obtidas a partir da classe \mathcal{H}_{p, m_j} da Seção 11.3.3 (exceto quando $n_j = 1$ e usamos $a = b = 0$). O teorema a seguir e um corolário fornecem um limite sobre os tamanhos combinados esperados de todas as tabelas hash secundário. Um segundo corolário limita a probabilidade de que o tamanho combinado de todas as tabelas hash secundário seja superlinear.

Teorema 11.10

Se armazenarmos n chaves em uma tabela hash de tamanho $m = n$ usando uma função hash h escolhida ao acaso a partir de uma classe universal de funções hash, então

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n,$$

onde n_j é o número de chaves que efetuam o hash para a posição j .

Prova Começamos com a identidade a seguir, válida para qualquer inteiro não negativo a :

$$a^2 = a + 2\binom{a}{2}. \quad (11.6)$$

Temos

$$\begin{aligned} E\left[\sum_{j=0}^{m-1} n_j^2\right] &= E\left[\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(pela equação (11.6))} \\ &= E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(por linearidade de expectativa)} \\ &= E[n] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(pela equação (11.1))} \\ &= n + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(pois } n \text{ não é uma variável aleatória).} \end{aligned}$$

Para avaliar o somatório $\sum_{j=0}^{m-1} \binom{n_j}{2}$, observamos que ele é simplesmente o número total de colisões. Pelas propriedades de hash universal, o valor esperado desse somatório é no máximo

$$\binom{n}{2} \frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2},$$

200 | pois $m = n$. Desse modo,

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + 2 \frac{n-1}{2}$$

$$= 2n - 1$$

$$< 2n .$$

Corolário 11.11

Se armazenarmos n chaves em uma tabela hash de tamanho $m = n$ usando uma função hash b escolhida ao acaso a partir de uma classe universal de funções hash e definirmos o tamanho de cada tabela hash secundário como $m_j = n_j^2$ para $j = 0, 1, \dots, m - 1$, então a quantidade esperada de armazenamento necessário para todas tabelas hash secundário em um esquema de hash perfeito é menor que $2n$.

Prova Tendo em vista que $m_j = n_j^2$ para $j = 0, 1, \dots, m - 1$, o Teorema 11.10 nos dá

$$E\left[\sum_{j=0}^{m-1} m_j\right] = E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n , \quad (11.7)$$

o que completa a prova.

Corolário 11.12

Se armazenarmos n chaves em uma tabela hash de tamanho $m = n$, usando uma função hash b escolhida ao acaso a partir de uma classe universal de funções hash, e definirmos o tamanho de cada tabela hash secundário como $m_j = n_j^2$ para $j = 0, 1, \dots, m - 1$, então a probabilidade de que o armazenamento total usado para tabelas hash secundário exceda $4n$ é menor que $1/2$.

Prova Aplicamos mais uma vez a desigualdade de Markov (C.29), $\Pr\{X \geq t\} \leq E[X]/t$, dessa vez à desigualdade (11.7), com $X = \sum_{j=0}^{m-1} m_j$ e $t = 4n$:

$$\Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} \leq \frac{E\left[\sum_{j=0}^{m-1} m_j\right]}{4n}$$

$$< \frac{2n}{4n}$$

$$= 1/2 .$$

Do Corolário 11.12, vemos que testar algumas funções hash escolhidas ao acaso a partir da família universal produzirá rapidamente uma função que utilizará uma quantidade razoável de espaço de armazenamento.

Exercícios

11.5-1 *

Suponha que inserimos n chaves em uma tabela hash de tamanho m usando o endereçamento aberto e o hash uniforme. Seja $p(n, m)$ a probabilidade de não ocorrer nenhuma colisão. Mostre que $p(n, m) \leq e^{-n(n-1)/2m}$. (Sugestão: Consulte a equação (3.11).) Demonstre que, quando n excede \sqrt{m} , a probabilidade de evitar colisões cai rapidamente a zero.

Problemas

11-1 Limite de sondagem mais longo para o hash

Uma tabela hash de tamanho m é usada para armazenar n itens, com $n \leq m/2$. O endereçamento aberto é usado para resolução de colisões.

- Supondo hash uniforme, mostre que, para $i = 1, 2, \dots, n$, a probabilidade de a i -ésima inserção exigir estritamente mais de k sondagens é no máximo 2^{-k} .
- Mostre que, para $i = 1, 2, \dots, n$, a probabilidade de a i -ésima inserção exigir mais de $2 \lg n$ sondagens é no máximo $1/n^2$.

Seja a variável aleatória X_i que denota o número de sondagens exigidas pela i -ésima inserção. Você mostrou na parte (b) que $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Seja a variável aleatória $X = \max_{1 \leq i \leq n} X_i$ que denota o número máximo de sondagens exigidas por quaisquer das n inserções.

- Mostre que $\Pr\{X > 2 \lg n\} \leq 1/n$.
- Mostre que o comprimento esperado $E[X]$ da mais longa seqüência de sondagens é $O(\lg n)$.

11-2 Limite do tamanho da posição para encadeamento

Suponha que temos uma tabela hash com n posições, com colisões resolvidas por encadeamento, e suponha também que n chaves sejam inseridas na tabela. Cada chave tem igual probabilidade de sofrer hash para cada posição. Seja M o número máximo de chaves em qualquer posição após todas as chaves terem sido inseridas. Sua missão é provar um limite superior $O(\lg n / \lg \lg n)$ sobre $E[M]$, o valor esperado de M .

- Mostre que a probabilidade Q_k de ocorrer o hash de k chaves para uma determinada posição é dada por

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

- Seja P_k a probabilidade de que $M = k$, ou seja, a probabilidade de que a posição contendo a maioria das chaves contenha k chaves. Mostre que $P_k \leq nQ_k$.
- Use a aproximação de Stirling, equação (3.17), para mostrar que $Q_k < e^k / k^k$.
- Mostre que existe uma constante $c > 1$ tal que $Q_{k_0} < 1/n^3$ para $k_0 = c \lg n / \lg \lg n$. Conclua que $P_k < 1/n^2$ para $k \geq k_0 = c \lg n / \lg \lg n$.
- Mostre que

$$E[M] \leq \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Conclua que $E[M] = O(\lg n / \lg \lg n)$.

11-3 Sondagem quadrática

Suponha que recebemos uma chave k para pesquisar uma tabela hash com posições $0, 1, \dots, m-1$, e suponha que temos uma função hash b mapeando o espaço de chaves no conjunto $\{0, 1, \dots, m-1\}$. O esquema de pesquisa é dado a seguir.

- 202 | 1. Calcular o valor $i \leftarrow b(k)$ e definir $j \leftarrow 0$.

2. Efetuar a sondagem na posição i em busca da chave desejada k . Se a encontrar, ou se essa posição estiver vazia, encerrar a pesquisa.
3. Definir $j \leftarrow (j + 1) \bmod m$ e $i \leftarrow (i + j) \bmod m$, e retornar à Etapa 2. Suponha que m seja uma potência de 2.
 - a. Mostre que esse esquema é uma instância do esquema geral de “sondagem quadrática”, exibindo as constantes c_1 e c_2 apropriadas para a equação (11.5).
 - b. Prove que esse algoritmo examina cada posição da tabela no pior caso.

11-4 Hash de k -universal e autenticação

Seja $\mathcal{H} = \{h\}$ uma classe de funções hash na qual cada h mapeia o universo U de chaves para $\{0, 1, \dots, m - 1\}$. Dizemos que \mathcal{H} é **k -universal** se, para toda seqüência fixa de k chaves distintas $\langle x^{(1)}, x^{(2)}, \dots, x^{(k)} \rangle$ e para qualquer h escolhido ao acaso a partir de \mathcal{H} , a seqüência $\langle h(x^{(1)}), h(x^{(2)}), \dots, h(x^{(k)}) \rangle$ tem igual probabilidade de ser qualquer uma das m^k seqüências de comprimento k com elementos estabelecidos a partir de $\{0, 1, \dots, m - 1\}$.

- a. Mostre que, se \mathcal{H} é 2-universal, então ela é universal.
 - b. Seja U o conjunto de n -tuplas de valores obtidos a partir de \mathbb{Z}_p , e seja $B = \mathbb{Z}_p$, onde p é primo. Para qualquer n -tupla $a = \langle a_0, a_1, \dots, a_{n-1} \rangle$ de valores de \mathbb{Z}_p e para qualquer $b \in \mathbb{Z}_p$, defina a função hash $b_{a,b} : U \rightarrow B$ sobre uma n -tupla de entrada $x = \langle x_0, x_1, \dots, x_{n-1} \rangle$ de U como
- $$b_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$
- e seja \mathcal{H} . Mostre que \mathcal{H} é 2-universal.
- c. Suponha que Alice e Bob concordam secretamente sobre uma função hash $b_{a,b}$ de uma família 2-universal \mathcal{H} de funções hash. Mais tarde, Alice envia pela Internet uma mensagem m a Bob, na qual $m \in U$. Ela autentica essa mensagem para Bob enviando também uma marca de autenticação $t = b_{a,b}(m)$, e Bob verifica se o par (m, t) que ele recebe satisfaz a $t = b_{a,b}(m)$. Suponha que um adversário intercepte (m, t) em trânsito e tente iludir Bob substituindo o par por um par diferente (m', t') . Mostre que a probabilidade de o adversário ter sucesso na tentativa de fazer Bob aceitar (m', t') é no máximo $1/p$, independente de quanta capacidade de computação o adversário tenha.

Notas do capítulo

Knuth [185] e Gonnet [126] são excelentes guias de referência para a análise de algoritmos de hash. Knuth credita a H. P. Luhn (1953) a criação de tabelas hash, juntamente com o método de encadeamento para resolução de colisões. Aproximadamente na mesma época, G. M. Amdahl apresentou a idéia de endereçamento aberto.

Carter e Wegman introduziram a noção de classes universais de funções hash em 1979 [52].

Fredman, Komlós e Szemerédi [96] desenvolveram o esquema de hash perfeito para conjuntos estáticos apresentado na Seção 11.5. Uma extensão de seu método para conjuntos dinâmicos, tratamento de inserções e eliminações em tempo esperado amortizado $O(1)$, foi apresentada por Dietzfelbinger *et al.* [73].