Capítulo 5

Tabelas Hash

As tabelas Hash são um método eficiente de armazenar um número pequeno, n, de números inteiros de uma grande faixa $U = \{0, \ldots, 2^w - 1\}$. O termo *tabela hash* inclui uma ampla gama de estruturas de dados. A primeira parte deste capítulo concentra-se em duas das implementações mais comuns de tabelas de hash: hashing com encadeamento e sondagem linear.

Muitas vezes, as tabelas hash armazenam tipos de dados que não são inteiros. Nesse caso, um *código hash* inteiro está associado a cada item de dados e é usado na tabela hash. A segunda parte deste capítulo discute como esses códigos de hash são gerados.

Alguns dos métodos utilizados neste capítulo exigem escolhas aleatórias de números inteiros em algum intervalo específico. Nos exemplos de código, alguns desses números inteiros "aleatórios" são constantes codificadas. Essas constantes foram obtidas usando bits aleatórios gerados pelo ruído atmosférico.

5.1 ChainedHashTable: Uma Tabela de Dispersão por Encadeamento

Uma estrutura de dados ChainedHashTable usa *dispersão por encadeamento* para armazenar dados como um array, t, de listas. Um número inteiro, n, acompanha o número total de itens em todas as listas (veja Figura 5.1):

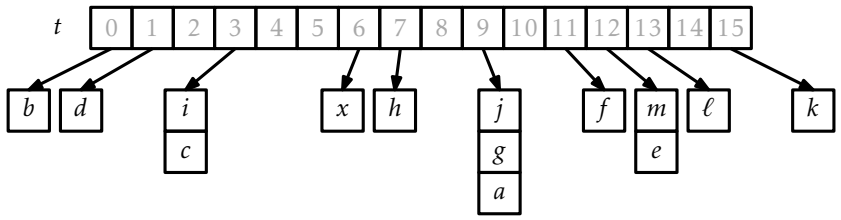

Figura 5.1: Um exemplo de ChainedHashTable com $n = 14$ e length$(t) = 16$. Nesse exemplo hash$(x) = 6$

```
initialize()
    d ← 1
    t ← alloc_table(2^d)
    z ← random_odd_int()
    n ← 0
```

O *valor hash* de um item de dados x, indicado por hash(x), é um valor na faixa $\{0,\ldots,\text{length}(t)-1\}$. Todos os itens com valor de hash i são armazenados na lista em $t[i]$. Para garantir que as listas não sejam grandes, mantemos o invariante

$$n \leq \text{length}(t)$$

assim, a média de números armazenados na lista será $n/\text{length}(t) \leq 1$.

Para adicionar um elemento, x, à tabela de hash, primeiro verificamos se o comprimento de t precisa ser aumentado e, se assim for, crescemos t. Com isso resolvido, vamos decodificar x por meio de uma função hash para obtermos um número inteiro, i, no intervalo $\{0,\ldots,\text{length}(t)-1\}$ e anexamos x à lista $t[i]$:

```
add(x)
    if find(x) ≠ nil then return false
    if n + 1 > length(t) then resize()
    t[hash(x)].append(x)
    n ← n + 1
    return true
```

Crescer a tabela, se necessário, envolve duplicar o comprimento de t e reinserir todos os elementos na nova tabela. Esta estratégia é exatamente a mesma que a utilizada na implementação da ArrayStack e o mesmo resultado se aplica: O custo de crescimento é apenas constante quando amortizado em uma sequência de inserções (veja Lema 2.1 na página 35).

Além de crescer, o único outro trabalho feito ao adicionar um novo valor x a uma ChainedHashTable envolve anexar x à lista $t[\text{hash}(x)]$. Para qualquer uma das implementações de lista descritas nos Capítulos 2 ou 3, isso leva um tempo constante.

Para remover um elemento, x, da tabela hash, iteramos sobre a lista $t[\text{hash}(x)]$ até encontrar x para, então, removê-lo:

```
remove(x)
    ℓ ← t[hash(x)]
    for y in ℓ do
        if y = x then
            ℓ.remove_value(y)
            n ← n − 1
            if 3 · n < length(t) then resize()
            return y
    return nil
```

Isso leva $O(n_{\text{hash}(x)})$ tempo, onde n_i indica o comprimento da lista armazenada em $t[i]$.

Procurar o elemento x em uma tabela de hash é semelhante. Realizamos uma pesquisa linear na lista $t[\text{hash}(x)]$:

```
find(x)
    for y in t[hash(x)] do
        if y = x then
            return y
    return nil
```

Novamente, isso leva tempo proporcional ao comprimento da lista

$t[\text{hash}(x)]$.

O desempenho de uma tabela hash depende criticamente da escolha da função hash. Uma boa função de hash irá espalhar os elementos uniformemente entre as listas length(t), de modo que o tamanho esperado da lista $t[\text{hash}(x)]$ será $O(n/\text{length}(t)) = O(1)$. Por outro lado, uma má função de hash espalhará todos os valores (incluindo x) para a mesma localização da tabela, caso em que o tamanho da lista $t[\text{hash}(x)]$ será n. Na próxima seção, descrevemos uma boa função de hash.

5.1.1 Hash Multiplicativo

O hashing multiplicativo é um método eficiente de gerar valores de hash com base na aritmética modular (discutido em Seção 2.3) e na divisão de números inteiros. Ele usa o operador div, que calcula a parte inteira de um quociente, ao descartar o restante. Formalmente, para qualquer número inteiro $a \geq 0$ e $b \geq 1$, $a \operatorname{div} b = \lfloor a/b \rfloor$.

No hash multiplicativo, usamos uma tabela hash de tamanho 2^d para um número inteiro d qualquer (chamado *dimensão*). A função hash de um inteiro $x \in \{0,\ldots,2^w-1\}$ é

$$\text{hash}(x) = ((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d} \quad .$$

Aqui, z é um inteiro *ímpar* escolhido aleatoriamente em $\{1,\ldots,2^w-1\}$. Esta função de hash pode ser realizada de forma muito eficiente observando que, por padrão, as operações em números inteiros já são feitas em modulo 2^w onde w é o número de bits em um número inteiro. [1] (Ver Figura 5.2.) Além disso, a divisão inteira por 2^{w-d} é equivalente a descartar os $w-d$ bits mais à direita em uma representação binária (que é implementado deslocando os bits da direita por $w-d$ usando o operador $\gg$).

```
hash(x)
    return ((z · hash(x)) mod 2^w) ≫ (w − d)
```

[1] Isso é verdadeiro para a maioria das linguagens de programação, incluindo C, C#, C++ e Java. Exceções notáveis são Python e Ruby, no qual o resultado de uma operação de números inteiros de comprimento w-bit fixo, é atualizado para uma representação de comprimento variável.

2^w (4294967296)	10000000000000000000000000000000
z (4102541685)	11110100100001111101000101110101
x (42)	00000000000000000000000000101010
$z \cdot x$	10100000011110010010000101110100110010
$(z \cdot x) \bmod 2^w$	00011110010010000101110100110010
$((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d}$	00011110

Figura 5.2: A operação de um hash multiplicativo com $w = 32$ e $d = 8$.

O lema a seguir, cuja prova é adiada até mais tarde nesta seção, mostra que o hash multiplicativo faz um bom trabalho para evitar colisões:

Lema 5.1. *Seja x e y dois valores em $\{0, \ldots, 2^w - 1\}$ com $x \neq y$. Então* $\Pr\{\text{hash}(x) = \text{hash}(y)\} \leq 2/2^d$.

Com o Lema 5.1, o desempenho de remove(x) e find(x) são fáceis de analisar:

Lema 5.2. *Para qualquer valor de dados x, o comprimento esperado da lista $t[\text{hash}(x)]$ é no máximo $n_x + 2$, onde n_x é o número de ocorrências de x na tabela hash.*

Demonstração. Tomando S pelo (multi-)conjunto de elementos armazenados na tabela hash que não são iguais a x. Para um elemento $y \in S$, define-se a variável indicadora

$$I_y = \begin{cases} 1 & \text{se hash}(x) = \text{hash}(y) \\ 0 & \text{caso contrário} \end{cases}$$

E observe que, pelo Lema 5.1, $\mathrm{E}[I_y] \leq 2/2^d = 2/\text{length}(t)$. O comprimento esperado da lista $t[\text{hash}(x)]$ é dado por:

$$\begin{aligned} \mathrm{E}\left[t[\text{hash}(x)].\text{size}()\right] &= \mathrm{E}\left[n_x + \sum_{y \in S} I_y\right] \\ &= n_x + \sum_{y \in S} \mathrm{E}[I_y] \\ &\leq n_x + \sum_{y \in S} 2/\text{length}(t) \end{aligned}$$

$$\begin{aligned} &\leq n_x + \sum_{y \in S} 2/n \\ &\leq n_x + (n - n_x)2/n \\ &\leq n_x + 2 \ , \end{aligned}$$

conforme exigido. □

Agora, queremos provar Lema 5.1, mas primeiro precisamos de um resultado da teoria dos números. Na seguinte prova, usamos a notação $(b_r, \ldots, b_0)_2$ para denotar $\sum_{i=0}^{r} b_i 2^i$, onde cada b_i é um pouco, seja 0 ou 1. Em outras palavras, $(b_r, \ldots, b_0)_2$ é o inteiro cuja representação binária é dada por $b_r, \ldots, b_0$. Usamos $\star$ para denotar um bit de valor desconhecido.

Lema 5.3. *Seja S o conjunto de inteiros ímpares em $\{1, \ldots, 2^w - 1\}$; Seja q e i dois elementos em S. Então há exatamente um valor $z \in S$, de modo que $zq \bmod 2^w = i$.*

Demonstração. Uma vez que o número de escolhas para z e i é o mesmo, basta provar que existe *no máximo* um valor $z \in S$ que satisfaça $zq \bmod 2^w = i$.

Suponhamos, por uma questão de contradição, que existem dois desses valores z e z', com $z > z'$. Então

$$zq \bmod 2^w = z'q \bmod 2^w = i$$

Portanto

$$(z - z')q \bmod 2^w = 0$$

Mas isso significa que

$$(z - z')q = k2^w \tag{5.1}$$

para qualquer inteiro k. Pensando em termos de números binários, temos que

$$(z - z')q = k \cdot (1, \underbrace{0, \ldots, 0}_{w})_2 \ ,$$

de modo que os w bits de fuga na representação binária de $(z - z')q$ são todos os 0's.

Além disso, $k \neq 0$, desde $q \neq 0$ e $z - z' \neq 0$. Uma vez que q é impar, não possui 0's na sua representação binária.

$$q = (\star, \ldots, \star, 1)_2 \ .$$

Desde $|z - z'| < 2^w$, $z - z'$ tem menos do que w 0's seguidos na sua representação binária:

$$z - z' = (\star, \ldots, \star, 1, \underbrace{0, \ldots, 0}_{<w})_2 \ .$$

Portanto, o produto $(z - z')q$ tem menos do que w 0's na sua representação binária:

$$(z - z')q = (\star, \cdots, \star, 1, \underbrace{0, \ldots, 0}_{<w})_2 \ .$$

Dessa forma, $(z - z')q$ não pode satisfazer (5.1), produzindo uma contradição e completando a prova. □

A utilidade do Lema 5.3 vem da seguinte observação: Se z for escolhido uniformemente aleatoriamente de S, então zt é distribuído uniformemente em S. Na seguinte prova, ajuda a pensar na representação binária de z, que consiste em $w - 1$ bits aleatórios seguido por um 1.

Prova do Lema 5.1. Primeiro, observamos que a condição $\text{hash}(x) = \text{hash}(y)$ é equivalente à declaração "a ordem mais alta d bits de $zx \bmod 2^w$ e os d bits de ordem superior $zy \bmod 2^w$ são os mesmos." Uma condição necessária dessa afirmação é que os bits d de ordem superior na representação binária de $z(x - y)\ bmod 2^w$ são todos 0 ou todos 1. Isso é,

$$z(x - y) \bmod 2^w = (\underbrace{0, \ldots, 0}_{d}, \underbrace{\star, \ldots, \star}_{w-d})_2 \tag{5.2}$$

quando $zx \bmod 2^w > zy \bmod 2^w$ ou

$$z(x - y) \bmod 2^w = (\underbrace{1, \ldots, 1}_{d}, \underbrace{\star, \ldots, \star}_{w-d})_2 \ . \tag{5.3}$$

quando $zx \bmod 2^w < zy \bmod 2^w$. Portanto, apenas temos que limitar a probabilidade de que $z(x - y) \bmod 2^w$ pareça (5.2) ou (5.3).

Seja q um inteiro ímpar exclusivo tal que $(x-y) \bmod 2^w = q2^r$ para algum inteiro $r \geq 0$. Pela Lema 5.3, a representação binária de $zq \bmod 2^w$ tem $w-1$ bits aleatórios, seguido por um 1:

$$zq \bmod 2^w = (\underbrace{b_{w-1}, \ldots, b_1}_{w-1}, 1)_2$$

Portanto, a representação binária de $z(x-y) \bmod 2^w = zq2^r \bmod 2^w$ tem $w-r-1$ bits aleatórios, seguido de um 1, seguido de r 0's:

$$z(x-y) \bmod 2^w = zq2^r \bmod 2^w = (\underbrace{b_{w-r-1}, \ldots, b_1}_{w-r-1}, 1, \underbrace{0, 0, \ldots, 0}_{r})_2$$

Agora podemos finalizar a prova: se $r > w-d$, então os d bits de ordem superior de $z(x-y) \bmod 2^w$ contêm tanto 0's quanto 1's, então a probabilidade de que $z(x-y) \bmod 2^w$ pareça (5.2) ou (5.3) é 0 . Se $r = w-d$, então a probabilidade de se parecer com (5.2) é 0, mas a probabilidade de se parecer com (5.3) é $1/2^{d-1} = 2/2^d$ (uma vez que devemos ter $b_1, \ldots, b_{d-1} = 1, \ldots, 1$). Se $r < w-d$, então devemos ter $b_{w-r-1}, \ldots, b_{w-r-d} = 0, \ldots, 0$ ou $b_{w-r-1}, \ldots, b_{w-r-d} = 1, \ldots, 1$. A probabilidade de cada um desses casos é $1/2^d$ e eles são mutuamente exclusivos, então a probabilidade de um desses casos é $2/2^d$. Isso completa a prova. □

5.1.2 Resumo

O seguinte teorema resume o desempenho de uma estrutura de dados ChainedHashTable:

Teorema 5.1. *Uma ChainedHashTable implementa a interface USet. Ignorando o custo das chamadas para* grow(), *a ChainedHashTable suporta as operações* add(x), remove(x) *e* find(x) *em $O(1)$ tempo esperado por operação.*

Além disso, começando com uma ChainedHashTable vazia, qualquer sequência de operações de m add(x) *e* remove(x) *resultará num total de $O(m)$ tempo gasto durante todas as chamadas para* grow().

5.2 LinearHashTable: Sondagem Linear

A estrutura de dados ChainedHashTable usa uma matriz de listas, onde a i-ésima lista armazena todos os elementos x, tal que hash(x) = i. Uma alternativa, chamada *endereçamento aberto* é armazenar os elementos diretamente em um array, t, com cada local do array em t armazenando no máximo um valor. Essa abordagem é tomada pela LinearHashTable descrita nesta seção. Em alguns lugares, esta estrutura de dados é descrita como *endereçamento aberto com sondagem linear*.

A principal ideia por trás de uma LinearHashTable é que gostaríamos, de preferência, de armazenar o elemento x com o valor hash $i \leftarrow \text{hash}(x)$ na localização da tabela $t[i]$. Se não pudermos fazer isso (porque algum elemento já está armazenado), então tentamos armazená-lo na localização $t[(i + 1) \bmod \text{length}(t)]$; Se isso não for possível, tentemos $t[(i + 2) \bmod \text{length}(t)]$, e assim por diante, até encontrar um lugar para x.

Há três tipos de registros armazenados em t:

1. dados: valores reais no USet que estamos representando;
2. valores *nil*: em locais de matriz onde nenhum dado foi armazenado; e
3. valores *del*: em locais de matriz onde os dados foram armazenados uma vez, mas que já foram excluídos.

Além do contador, n, que acompanha o número de elementos na LinearHashTable, um contador, q, acompanha o número de elementos dos Tipos 1 e 3. Isso é, q é igual a n mais o número de *del* valores em t. Para que isso funcione de forma eficiente, precisamos que t seja consideravelmente maior do que q, de modo que existam muitos valores *nil* em t. As operações na LinearHashTable mantêm, portanto, o invariante que $\text{length}(t) \geq 2q$.

Para resumir, um LinearHashTable contém uma matriz, t, que armazena elementos de dados e números inteiros n e q que acompanham o número de elementos de dados e valores não *nil* de t, respectivamente. Como muitas funções de hash funcionam apenas para tamanhos de tabela que são uma potência de 2, também mantemos um inteiro d e asseguramos o invariante $\text{length}(t) = 2^d$.

```
initialize()
    del ← object()

initialize()
    d ← 1
    t ← new_array(2^d)
    q ← 0
    n ← 0
```

A operação find(x) em LinearHashTable é simples. Nós começamos na entrada do array $t[i]$ onde $i = \text{hash}(x)$ e pesquisamos as entradas $t[i]$, $t[(i+1) \bmod \text{length}(t)]$, $t[(i+2) \bmod \text{length}(t)]$, e assim por diante, até encontrarmos um índice i' tal que, também, $t[i'] \leftarrow x$, ou $t[i'] \leftarrow nil$. No primeiro caso, nós retornamos $t[i']$. No último caso, concluímos que x não está contido na tabela hash e retorna *nil*.

```
find(x)
    i ← hash(x)
    while t[i] ≠ nil do
        if t[i] ≠ del and x = t[i] then
            return t[i]
        i ← (i + 1) mod length(t)
```

A operação add(x) também é bastante fácil de implementar. Depois de verificar que x ainda não está armazenado na tabela (usando find(x)), buscamos $t[i]$, $t[(i+1) \bmod \text{length}(t)]$, $t[(i+2) \bmod \text{length}(t)]$, e assim por diante, até encontrar *nil* ou *del* e armazenar x nessa localização, incrementar n e q, se apropriado.

```
add(x)
    if find(x) ≠ nil then return false
    if 2 · (q + 1) > length(t) then resize()
    i ← hash(x)
    while t[i] ≠ nil and t[i] ≠ del do
```

```
        i ← (i + 1) mod length(t)
    if t[i] = nil then q ← q + 1
    n ← n + 1
    t[i] ← x
    return true
```

Agora, a implementação da operação *remove*(x) deve ser óbvia. Nós buscamos $t[i]$, $t[(i+1) \bmod \text{length}(t)]$, $t[(i+2) \bmod \text{length}(t)]$, e assim por diante até encontrar um índice i' tal que $t[i'] \leftarrow x$ ou $t[i'] \leftarrow nil$. No primeiro caso, definimos $t[i'] \leftarrow del$ e retornamos *true*. No último caso, concluímos que x não foi armazenado na tabela (e, portanto, não pode ser excluído) e retornar *false*.

```
remove(x)
    i ← hash(x)
    while t[i] ≠ nil do
        y ← t[i]
        if y ≠ del and x = y then
            t[i] ← del
            n ← n − 1
            if 8 · n < length(t) then resize()
            return y
        i ← (i + 1) mod length(t)
    return nil
```

A correção dos métodos find(x), add(x) e remove(x) é fácil de verificar, contudo ela se apoia no uso de valores *del*. Observe que nenhuma dessas operações nunca estabeleceu uma entrada não-*nil* para *nil*. Portanto, quando alcançamos um índice i' tal que $t[i'] \leftarrow nil$, esta é uma prova de que o elemento x, que estamos procurando, não está armazenado na tabela; $t[i']$ sempre foi *nil*, então não há nenhuma razão para que uma operação anterior add(x) tenha prosseguido além do índice i'.

O método resize() é chamado por add(x) quando o número de entradas não-*nil* excede length(t)/2 ou remove(x) quando o número de entradas de dados é inferior a length(t)/8. O método resize() funciona como

os métodos resize() de outras estruturas de dados baseadas em array. Encontramos o menor inteiro não negativo d tal que $2^d \geq 3n$. Realocamos o array t para que ela tenha tamanho 2^d e, em seguida, inserimos todos os elementos da versão anterior de t na nova cópia redimensionada de t. Ao fazer isso, restabelecemos q igual a n, pois o recém-alocado t não contém valores *del*.

```
resize()
    d ← 1
    while (2^d < 3 · n) do d ← d + 1
    told ← t
    t ← new_array(2^d)
    q ← n
    for x in told do
        if x ≠ nil and x ≠ del then
            i ← hash(x)
            while t[i] ≠ nil do
                i ← (i + 1) mod length(t)
            t[i] ← x
```

5.2.1 Análise da Sondagem Linear

Observe que cada operação, add(x), remove(x) ou find(x), termina assim que (ou antes que) descubra a primeira entrada *nil* em t. A intuição por trás da análise de sondagem linear é que, uma vez que pelo menos metade dos elementos em t são iguais a *nil*, uma operação não demorará muito para ser concluída, pois será muito rápido encontrar uma entrada *nil*. No entanto, não devemos confiar muito nessa intuição, porque isso nos levaria à conclusão (incorreta) de que o número esperado de locais em t examinado por uma operação é no máximo de 2.

Para o restante desta seção, assumiremos que todos os valores de hash são distribuídos de forma independente e uniforme em $\{0, \ldots, \text{length}(t) - 1\}$. Esta não é uma suposição realista, mas permitirá que analisemos a sondagem linear. Mais tarde, nesta seção, descreveremos um método, chamado de hash de tabulação, que produz uma função de hash que é

"suficientemente boa"para sondagem linear. Também assumiremos que todos os índices nas posições de t são tomados no módulo length(t), de modo que $t[i]$ é realmente uma abreviatura para $t[i \bmod \text{length}(t)]$.

Dizemos que um *percurso de tamanho k que começa em i* ocorre quando todas as entradas da tabela $t[i], t[i+1], \ldots, t[i+k-1]$ não são *nil* e $t[i-1] = t[i+k] = nil$. O número de elementos não-*nil* de t é exatamente q e o método $add(x)$ garante que, em todos os momentos, $q \leq \text{length}(t)/2$. Existem q elementos $x_1, \ldots, x_q$ que foram inseridos em t desde a última operação resize(). Por nossa suposição, cada um deles tem um valor de hash, $hash(x_j)$, Isso é uniforme e independente do resto. Com esta configuração, podemos provar o lema principal necessário para analisar a sondagem linear.

Lema 5.4. *Corrija um valor $i \in \{0, \ldots, \text{length}(t)-1\}$. Então, a probabilidade de que um período de k começado em i seja $O(c^k)$ para alguma constante $0 < c < 1$.*

Demonstração. Se um período de k começar em i, então existem exatamente k elementos x_j such that $hash(x_j) \in \{i, \ldots, i+k-1\}$. A probabilidade de isso ocorrer é exatamente

$$p_k = \binom{q}{k}\left(\frac{k}{\text{length}(t)}\right)^k \left(\frac{\text{length}(t)-k}{\text{length}(t)}\right)^{q-k} ,$$

uma vez que, para cada escolha de k elementos, esses k elementos devem se dispersar para um dos k locais e o restante $q-k$ elementos devem se dispersar para os outros length(t) $-k$ locais da tabela. [2]

Na derivação a seguir, vamos trapacear um pouco e substituir $r!$ por $(r/e)^r$. A Aproximação de Stirling (Seção 1.3.2) mostra que isso é apenas um fator de $O(\sqrt{r})$ da verdade. Isso é feito apenas para tornar a derivação mais simples; Exercício 5.4 pede ao leitor para refazer o cálculo de forma mais rigorosa usando a Aproximação de Stirling na sua totalidade.

O valor de p_k é máximo quando length(t) é mínimo, e a estrutura de

[2] Note que p_k é maior do que a probabilidade de que um percurso k comece em i, uma vez que a definição de p_k não inclui o requisito $t[i-1] = t[i+k] = nil$.

dados mantém a invariante de $\text{length}(t) \geq 2q$, então

$$\begin{aligned}
p_k &\leq \binom{q}{k}\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&= \left(\frac{q!}{(q-k)!k!}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&\approx \left(\frac{q^q}{(q-k)^{q-k}k^k}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \qquad \text{[Aproximação de Stirling]} \\
&= \left(\frac{q^k q^{q-k}}{(q-k)^{q-k}k^k}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&= \left(\frac{qk}{2qk}\right)^k\left(\frac{q(2q-k)}{2q(q-k)}\right)^{q-k} \\
&= \left(\frac{1}{2}\right)^k\left(\frac{(2q-k)}{2(q-k)}\right)^{q-k} \\
&= \left(\frac{1}{2}\right)^k\left(1+\frac{k}{2(q-k)}\right)^{q-k} \\
&\leq \left(\frac{\sqrt{e}}{2}\right)^k \quad .
\end{aligned}$$

(Na última etapa, usamos a desigualdade $(1+1/x)^x \leq e$, que é válida para todos os $x > 0$.) Como $\sqrt{e}/2 < 0.824360636 < 1$, isso completa a prova. □

Usando Lema 5.4 para provar limites superiores no tempo de execução esperado de find(x), add(x) e remove(x) agora é bastante sucinto. Considere o caso mais simples, onde executamos find(x) para algum valor x que nunca tenha sido armazenado no LinearHashTable. Nesse caso, $i = \text{hash}(x)$ é um valor aleatório em $\{0,\ldots,\text{length}(t)-1\}$ independentemente do conteúdo de t. Se i for parte de uma extensão de k, então o tempo necessário para executar a operação find(x) é no máximo $O(1+k)$. Assim, o tempo de execução esperado pode ser delimitado por

$$O\left(1+\left(\frac{1}{\text{length}(t)}\right)\sum_{i=1}^{\text{length}(t)}\sum_{k=0}^{\infty} k\Pr\{i \text{ é parte de uma série } k\}\right) .$$

Observe que cada rodada de comprimento k contribui para a soma interna k vezes para uma contribuição total de k^2, então a soma acima pode

ser reescrita como

$$
\begin{aligned}
&O\left(1+\left(\frac{1}{\text{length}(t)}\right)\sum_{i=1}^{\text{length}(t)}\sum_{k=0}^{\infty}k^2\Pr\{i \text{ começa uma série } k\}\right)\\
&\leq O\left(1+\left(\frac{1}{\text{length}(t)}\right)\sum_{i=1}^{\text{length}(t)}\sum_{k=0}^{\infty}k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty}k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty}k^2\cdot O(c^k)\right)\\
&= O(1)\ .
\end{aligned}
$$

O último passo nesta derivação vem do fato de que $\sum_{k=0}^{\infty}k^2\cdot O(c^k)$ é uma série exponencialmente decrescente. [3] Portanto, concluimos que o tempo de execução esperado da operação find(x) para um valor x que não está contido em LinearHashTable é $O(1)$.

Se ignorarmos o custo da operação resize(), a análise acima nos dá tudo o que precisamos para analisar o custo das operações em um LinearHashTable.

Em primeiro lugar, a análise de find(x) fornecida acima aplica-se à operação add(x) quando x não está contido na tabela. Para analisar a operação find(x) quando x estiver contida na tabela, precisamos apenas observar que isso é o mesmo que o custo da operação add(x) que adicionou x na tabela anterior. Finalmente, o custo de uma operação remove(x) é igual ao custo de uma operação find(x).

Em resumo, se ignorarmos o custo das chamadas para resize(), todas as operações em LinearHashTable são executadas em $O(1)$ tempo esperado. A contabilização do custo do redimensionamento pode ser feita usando o mesmo tipo de análise amortizada realizada para a estrutura de dados ArrayStack em Seção 2.1.

[3] Na terminologia de muitos textos de cálculo, esta soma passa o teste de razão: existe um inteiro positivo k_0 tal que, para todos $k \geq k_0$, $\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$.

5.2.2 Resumo

O seguinte teorema resume o desempenho da estrutura de dados LinearHashTable:

Teorema 5.2. *A LinearHashTable implementa a interface USet. Ignorando o custo das chamadas para* resize(), *LinearHashTable suporta as operações* add(x), remove(x) *e* find(x) *em* $O(1)$ *tempo esperado por operação.*

Além disso, começando uma LinearHashTable vazia, qualquer sequência de m add(x) *e* remove(x) *operações resulta em um total de* $O(m)$ *tempo gasto durante todas as chamadas para* resize().

5.2.3 Hashing por Tabulação

Ao analisar a estrutura LinearHashTable, fizemos uma suposição muito forte: que para qualquer conjunto de elementos, $\{x_1,\ldots,x_n\}$, os valores de hash $hash(x_1),\ldots,hash(x_n)$ são distribuídos de forma independente e uniforme sobre o conjunto $\{0,\ldots,\text{length}(t)-1\}$. Uma maneira de conseguir isso é armazenar em um array gigante, tab, de comprimento 2^w, onde cada entrada é um inteiro de w-bit aleatório, independente de todas as outras entradas. Desta forma, podemos implementar hash(x) extraindo um inteiro de d-bit de $tab[x.\text{hash\_code}()]$:

```
ideal_hash(x)
    return tab[x.hash_code() ≫ w − d]
```

Aqui, $\gg$, é o operador *deslocamento de bits para a direita*, então

$$x.\text{hash\_code}() \gg w - d$$

extrai os d bits mais significativos de x's de w-bit códigos hash.

Infelizmente, armazenar uma matriz de tamanho 2^w é proibitivo em termos de uso de memória. A abordagem usada pela *hashing por tabulação* é, em vez disso, tratar números w-bit como sendo compostos de w/r inteiros, cada um com apenas r bits. Desta forma, o hashing de tabulação só precisa de w/r arrays cada um do comprimento 2^r. Todas as entradas nesses arrays são w-bit inteiros aleatoriamente independentes. Para obter

o valor de hash(x), divide-se os números inteiros x.hash_code() em w/r r-bit e usamos estes como índices nesses arrays. Em seguida, combinamos todos esses valores com o operador exclusivo de bit a bit para obter hash(x). O código a seguir mostra como isso funciona quando $w = 32$ e $r = 4$:

```
hash(x)
    h ← hash_code(x)
    return (tab[0][h ∧ ff16]
            ⊕ tab[1][(h ≫ 8) ∧ ff16]
            ⊕ tab[2][(h ≫ 16) ∧ ff16]
            ⊕ tab[3][(h ≫ 24) ∧ ff16]) ≫ (w − d)
```

Nesse caso, *tab* é um array bidimensional com quatro colunas e $2^{32/4} = 256$ linhas. Quantidades como ff_{16}, usadas acima, são *números hexadecimais* cujos dígitos têm 16 valores possíveis 0–9, que têm seu significado usual e a-f, que denotam 10–15. O número $\mathrm{ff}_{16} = 15 \cdot 16 + 15 = 255$. O símbolo $\wedge$ é o operador *bitwise AND*, então, o código $h \gg 8 \wedge \mathrm{ff}_{16}$ extrai bits com índices de 8 a 15 de h.

Pode-se verificar facilmente que, para qualquer x, hash(x) é uniformemente distribuído em $\{0, \ldots, 2^d - 1\}$. Com um pouco de trabalho, pode-se verificar se qualquer par de valores possui valores de hash independentes. Isso implica que o hash de tabulação poderia ser usado em lugar de hashing multiplicativo para a implementação ChainedHashTable.

No entanto, não é verdade que qualquer conjunto de n valores distintos forneça um conjunto de valores de hash independentes n. Contudo, quando o hashing de tabulação é usado, o limite de Teorema 5.2 ainda é válido. Referências para isso são fornecidas no final deste capítulo.

5.3 Hash Codes

As tabelas de hash discutidas na seção anterior são usadas para associar dados com chaves inteiras consistindo de w bits. Em muitos casos, temos chaves que não são inteiros. Podem ser strings, objetos, arrays ou outras

estruturas compostas. Para usar tabelas de hash para esses tipos de dados, devemos mapear esses tipos de dados para os códigos de hash com w-bits. Os mapeamentos de código Hash devem ter as seguintes propriedades:

1. Se x e y forem iguais, então x.hash_code() e y.hash_code() são iguais.
2. Se x e y não forem iguais, então a probabilidade de que x.hash_code() = y.hash_code() sejam iguais deve ser pequena (perto de $1/2^w$).

A primeira propriedade garante que, se armazenarmos x em uma tabela hash e, mais tarde, procuremos um valor y igual a x, então encontraremos x — como deveríamos. A segunda propriedade minimiza a perda de converter nossos objetos em números inteiros. Ele garante que os objetos desiguais geralmente tenham códigos de hash diferentes e, portanto, provavelmente serão armazenados em locais diferentes em nossa tabela de hash.

5.3.1 Códigos Hash para Tipos Primitivos de Dados

Pequenos tipos de dados primitivos, como *char*, *byte*, *int* e *float*, geralmente, são fáceis de encontrar códigos de hash. Esses tipos de dados sempre têm uma representação binária e essa representação binária geralmente consiste em w ou menos bits. Nesses casos, tratamos esses bits como a representação de um número inteiro na faixa $\{0,\ldots,2^w-1\}$. Se dois valores são diferentes, eles obtêm códigos hash diferentes. Se eles são o mesmo, eles obtêm o mesmo código hash.

Alguns tipos de dados primitivos são compostos por mais de w bits, geralmente cw bits para algum inteiro constante c. (Os tipos *long* e *double* do Java são exemplos disso com $c = 2$.) Esses tipos de dados podem ser tratados como objetos compostos feitos de c partes, conforme descrito na próxima seção.

5.3.2 Códigos Hash para Objetos Compostos

Para um objeto composto, queremos criar um código hash combinando os códigos hash individuais das partes constituintes do objeto. Isso não é tão fácil quanto parece. Embora se possa encontrar muitos hacks para

isso (por exemplo, combinando os códigos de hash com operações bitwise ou-exclusivo), muitos desses hacks terminam por serem frustrantes (ver exercícios 5.7–5.9). No entanto, se alguém estiver disposto a fazer aritmética com $2w$ bits de precisão, existem métodos simples e robustos disponíveis. Suponha que possamos ter um objeto composto por várias partes $P_0,\ldots,P_{r-1}$ cujos códigos hash são $x_0,\ldots,x_{r-1}$. Então, podemos escolher w-bits inteiros aleatórios e mutuamente independentes $z_0,\ldots,z_{r-1}$ e um $2w$-bit inteiro impar z e, então, calcular um código hash para nosso objeto com

$$h(x_0,\ldots,x_{r-1}) = \left(\left(z\sum_{i=0}^{r-1} z_i x_i\right) \bmod 2^{2w}\right) \operatorname{div} 2^w \quad .$$

Observe que este código hash tem um passo final (multiplicando por z e dividindo por 2^w) que usa a função de hash multiplicativa de Seção 5.1.1 para tirar o $2w$-bit resultado intermediário e reduzi-lo para um resultado final de w-bit. Aqui está um exemplo desse método aplicado a um objeto composto simples com três partes x_0, x_1, and x_2:

```
hash_code()
  z ← [2058cc50_16, cb19137e_16, 2cb6b6fd_16]
  zz ← bea0107e5067d19d_16
  h ← [x_0.hash_code(), x_1.hash_code(), x_2.hash_code()]
  return (((z[0]·h[0] + z[1]·h[1] + z[2]·h[2])·zz) mod 2^2·w)) ≫ w
```

O seguinte teorema mostra que, além de ser direto para implementar, esse método provavelmente é bom:

Teorema 5.3. *Sejam $x_0,\ldots,x_{r-1}$ e $y_0,\ldots,y_{r-1}$ sequencias de w bit inteiros em $\{0,\ldots,2^w-1\}$ e assumindo $x_i \neq y_i$ para pelo menos um índice $i \in \{0,\ldots,r-1\}$. Então*

$$\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq 3/2^w \quad .$$

Demonstração. Primeiro, ignoraremos o passo de hashing multiplicativo final e veremos como essa etapa contribui mais tarde. Definir:

$$h'(x_0,\ldots,x_{r-1}) = \left(\sum_{j=0}^{r-1} z_j x_j\right) \bmod 2^{2w} \quad .$$

Suponhamos que $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$. Podemos reescrever isso como:

$$z_i(x_i - y_i) \bmod 2^{2w} = t \tag{5.4}$$

onde

$$t = \left(\sum_{j=0}^{i-1} z_j(y_j - x_j) + \sum_{j=i+1}^{r-1} z_j(y_j - x_j) \right) \bmod 2^{2w}$$

Se assumirmos, sem perda de generalidade, que $x_i > y_i$, então (5.4) se torna

$$z_i(x_i - y_i) = t \ , \tag{5.5}$$

já que cada z_i e $(x_i - y_i)$ é ao menos $2^w - 1$, então o produto deles é ao menos $2^{2w} - 2^{w+1} + 1 < 2^{2w} - 1$. Pressuposto, $x_i - y_i \neq 0$, então (5.5) tem ao menos uma solução em z_i. Portanto, uma vez que z_i e t são independentes $z_0,\ldots,z_{r-1}$ são mutuamente independentes), a probabilidade de selecionar $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$ seja no máximo $1/2^w$.

O passo final da função hash é aplicar o hashing multiplicativo para reduzir o resultado intermediário $h'(x_0,\ldots,x_{r-1})$ de $2w$-bit para resultado final $h(x_0,\ldots,x_{r-1})$ de w-bit. Pelo Teorema 5.3, se $h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1})$, então $\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq 2/2^w$.

Resumindo,

$$\begin{aligned}
&\Pr\left\{ \begin{array}{l} h(x_0,\ldots,x_{r-1}) \\ \quad = h(y_0,\ldots,y_{r-1}) \end{array} \right\} \\
&= \Pr\left\{ \begin{array}{l} h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1}) \text{ or} \\ h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1}) \\ \quad \text{and } zh'(x_0,\ldots,x_{r-1}) \operatorname{div} 2^w = zh'(y_0,\ldots,y_{r-1}) \operatorname{div} 2^w \end{array} \right\} \\
&\leq 1/2^w + 2/2^w = 3/2^w \ .
\end{aligned}$$

□

5.3.3 Códigos Hash para Arrays e Strings

O método da seção anterior funciona bem para objetos que possuem um número fixo, constante, de componentes. No entanto, ele se destrói quando queremos usá-lo com objetos que possuem uma quantidade variável de componentes, uma vez que requer um número aleatório z_i de w-bit para cada componente. Poderíamos usar uma sequência pseudo-randômica

para gerar tantos z_i's quanto precisássemos, mas então os z_i não seriam mutuamente independentes e, assim, torna-se-ia difícil provar que os números de pseudo-randômicos não interagem bem com a função hash que estamos usando. Em particular, os valores de t e z_i na prova de Teorema 5.3 não são mais independentes.

Uma abordagem mais rigorosa é basear nossos códigos de hash em polinômios sobre os campos principais; Estes são apenas polinômios regulares que são avaliados em um número primo, p. Este método baseia-se no seguinte teorema, que diz que os polinômios sobre os campos principais se comportam quase como os polinômios usuais:

Teorema 5.4. *Seja p um número primo, e seja $f(z) = x_0 z^0 + x_1 z^1 + \cdots + x_{r-1} z^{r-1}$ uma expressão polinomial, não trivial, com coeficientes $x_i \in \{0, \ldots, p-1\}$. Então a equação $f(z) \bmod p = 0$ tem no mínimo $r-1$ soluções para $z \in \{0, \ldots, p-1\}$.*

Para usar o Teorema 5.4, nós "hasheamos"uma sequência de inteiros $x_0, \ldots, x_{r-1}$ com cada $x_i \in \{0, \ldots, p-2\}$ usando um inteiro aleatório $z \in \{0, \ldots, p-1\}$ via fórmula

$$h(x_0, \ldots, x_{r-1}) = \left(x_0 z^0 + \cdots + x_{r-1} z^{r-1} + (p-1) z^r \right) \bmod p \ .$$

Observe o termo extra $(p-1)z^r$ no final da fórmula. Isso ajuda a pensar em $(p-1)$ como o último elemento, x_r, na sequência x_r. Observe, também, que este elemento difere de qualquer outro elemento na sequência (cada um dos quais está no conjunto $\{0, \ldots, p-2\}$). Podemos pensar em $p-1$ como um marcador de fim de sequência.

O seguinte teorema, que considera o caso de duas sequências do mesmo comprimento, mostra que esta função hash dá um bom retorno para a pequena quantidade de aleatorização necessária para escolher z:

Teorema 5.5. *Seja $p > 2^w + 1$ um primo, seja $x_0, \ldots, x_{r-1}$ e $y_0, \ldots, y_{r-1}$ sequências de inteiros de w-bit em $\{0, \ldots, 2^w - 1\}$, e assumindo-se $x_i \neq y_i$ para pelo menos um índice $i \in \{0, \ldots, r-1\}$. Então*

$$\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq (r-1)/p \ .$$

Demonstração. A equação $h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})$ pode ser reescrita como

$$\left((x_0 - y_0) z^0 + \cdots + (x_{r-1} - y_{r-1}) z^{r-1} \right) \bmod p = 0. \tag{5.6}$$

Sendo $x_i \neq y_i$, um polinômio não trivial. Portanto, pelo Teorema 5.4, tem no máximo $r-1$ soluções em z. A probabilidade de escolher z para ser uma dessas soluções é, portanto, no máximo $(r-1)/p$. □

Note-se que esta função de hash também trata do caso em que duas sequências têm comprimentos diferentes, mesmo quando uma das sequências é um prefixo do outro. Isso ocorre porque esta função efetivamente colmeia a sequência infinita

$$x_0, \ldots, x_{r-1}, p-1, 0, 0, \ldots \ .$$

Isso garante que, se tivermos duas sequências de comprimento r e r' com $r > r'$, essas duas sequências diferem no índice $i = r$. Nesse caso, (5.6) se torna

$$\left(\sum_{i=0}^{i=r'-1} (x_i - y_i) z^i + (x_{r'} - p + 1) z^{r'} + \sum_{i=r'+1}^{i=r-1} x_i z^i + (p-1) z^r \right) \bmod p = 0 \ ,$$

que, pelo Teorema 5.4, tem no máximo r soluções em z. Isto combinado com Teorema 5.5 é suficiente para comprovar o seguinte teorema mais geral:

Teorema 5.6. *Seja $p > 2^w + 1$ um primo, seja $x_0, \ldots, x_{r-1}$ e $y_0, \ldots, y_{r'-1}$ sequências distintas de inteiros de w-bit em $\{0, \ldots, 2^w - 1\}$. Então*

$$\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq \max\{r, r'\}/p \ .$$

O código de exemplo a seguir mostra como esta função hash é aplicada a um objeto que contém um array, x, de valores:

```
hash_code()
    p ← 2^32 − 5    # this is a prime number
    z ← 64b6055a_16 # 32 bits from random.org
    z_2 ← 5067d19d_16 # random odd 32 bit number
    s ← 0
    zi ← 1
    for i in 0, 1, 2, ..., length(x) − 1 do
        # reduce to 31 bits
```

$xi \leftarrow ((x[i].\text{hash\_code}() \cdot z_2) \bmod 2^{32}) \gg 1$
$s \leftarrow (s + zi \cdot xi) \bmod p$
$zi \leftarrow (zi \cdot z) \bmod p$
$s \leftarrow (s + zi \cdot (p-1)) \bmod p$
return $s \bmod 2^{32}$

O código anterior sacrifica alguma probabilidade de colisão para conveniência de implementação. Em particular, aplica a função hash multiplicativa de Seção 5.1.1, com $d = 31$ para reduzir $x[i]$.hash_code() para um valor de 31 bits. Isto é para que as adições e multiplicações que são feitas modulo o principal $p = 2^{32} - 5$ podem ser realizadas usando aritmética não assinada de 63 bits. Assim, a probabilidade de duas sequências diferentes, cujo maior comprimento é r, tendo o mesmo código de hash no máximo

$$2/2^{31} + r/(2^{32} - 5)$$

em vez de $r/(2^{32} - 5)$ especificado em Teorema 5.6.

5.4 Discussões e Exercícios

As tabelas Hash e os códigos hash representam um campo de pesquisa enorme e ativo que é apenas abordado neste capítulo. A Bibliografia online sobre Hashing [10] contém cerca de 2000 entradas.

Existe uma variedade de implementações de tabela hash diferentes. A descrita na Seção 5.1 é conhecida como *hashing com encadeamento* (cada entrada do array contém uma cadeia (List) de elementos). Hashing com encadeamento remonta a um memorando interno da IBM criado por H. P. Luhn e datado de janeiro de 1953. Este memorando também parece ser uma das primeiras referências às linked lists.

Uma alternativa ao hashing com encadeamento é a usada pelos esquemas *open address*, onde todos os dados são armazenados diretamente em um array. Esses esquemas incluem a estrutura LinearHashTable de Seção 5.2. Essa idéia também foi proposta, independentemente, por um grupo da IBM na década de 1950. Os esquemas de endereçamento aberto devem lidar com o problema de *resolução de colisão*: index collision re-

solutiono caso em que dois valores hash para o mesmo local da matriz. Existem estratégias diferentes para resolução de colisão; Estes fornecem garantias de desempenho diferentes e muitas vezes exigem funções de hash mais sofisticadas do que as descritas aqui.

Outra categoria de implementações de tabela de hash são os chamados métodos de *hashing perfeito*. Estes são métodos em que as operações find(x) levam $O(1)$ tempo no pior caso. Para conjuntos de dados estáticos, isso pode ser conseguido encontrando *funções de hash perfeitas* para os dados; Essas são funções que mapeiam cada peça de dados para um local de matriz exclusivo. Para os dados que mudam ao longo do tempo, os métodos de hash perfeitos incluem *tabelas de hash de dois níveis FKS* [31, 24] e *cuckoo hashing* [55].

As funções de hash apresentadas neste capítulo estão provavelmente entre os métodos mais práticos atualmente conhecidos que podem comprovadamente funcionar bem para qualquer conjunto de dados. Outros métodos provavelmente bons datam do trabalho pioneiro de Carter e Wegman, que introduziram a noção de *hash universal* e descreveram várias funções de hash para diferentes cenários [14]. Hashing por tabulação, descrito em Seção 5.2.3, é graças a Carter e Wegman [14], mas sua análise, quando aplicada a sondagem linear (e vários outros esquemas de tabela de hash), é graças a Pătraşcu e Thorup [58].

A ideia do *hash multiplicativo* é muito antiga e parece ser parte do folclore hashing [48, Section 6.4]. No entanto, a idéia de escolher o multiplicador z para ser um número aleatório *impar* e a análise em Seção 5.1.1 é devida a Dietzfelbinger *et al.* [23]. Esta versão do hashing multiplicativo é uma das mais simples, mas a probabilidade de colisão de $2/2^d$ é um fator de dois maiores do que o que se poderia esperar com uma função aleatória de $2^w \to 2^d$. O método *multiplica-soma hashing* usa a função

$$h(x) = ((zx + b) \bmod 2^{2w}) \operatorname{div} 2^{2w-d}$$

onde z e b são escolhidos aleatoriamente de $\{0, \ldots, 2^{2w} - 1\}$. O hashing Multiply-add tem uma probabilidade de colisão de apenas $1/2^d$[21], mas requer aritmética de precisão $2w$-bit.

Há uma série de métodos para obter códigos hash de sequências de comprimento fixo de w-bit inteiros. Um método particularmente rápido

[11] é a função

$$h(x_0,\ldots,x_{r-1}) = \left(\sum_{i=0}^{r/2-1}((x_{2i}+a_{2i}) \bmod 2^w)((x_{2i+1}+a_{2i+1}) \bmod 2^w)\right) \bmod 2^{2w}$$

onde r é igual e $a_0,\ldots,a_{r-1}$ são escolhidos aleatoriamente de $\{0,\ldots,2^w\}$. Isso produz um código hash de $2w$-bit com probabilidade de colisão $1/2^w$. Isso pode ser reduzido para um código hash de w-bit usando o hashing multiplicativo (ou multiplicativo). Esse método é rápido porque requer apenas $r/2$ $2w$-bit multiplicações , enquanto o método descrito em Seção 5.3.2 requer r multiplicações. (As operações mod ocorrem implicitamente usando a aritmética w e $2w$-bit para as adições e multiplicações, respectivamente.)

O método de Seção 5.3.3 de usar polinômios sobre campos primos para matrizes de comprimento variável de hash e strings é devido a Dietzfelbinger *et al.* [22]. Devido ao uso do operador mod que depende de uma instrução de máquina dispendiosa, infelizmente não é muito rápido. Algumas variantes deste método escolhem o primo p para ser um da forma $2^w - 1$, caso em que o operador mod pode ser substituído por adição (+) e operações bitwise-E ($\wedge$) [47, Seção 3.6]. Outra opção é aplicar um dos métodos rápidos para sequências de caracteres de comprimento fixo para blocos de comprimento c para alguma constante $c > 1$ e, em seguida, aplicar o método de campo primário à sequência resultante de $\lceil r/c \rceil$ códigos hash.

Exercício 5.1. Uma determinada universidade atribui cada um dos números de seus alunos a primeira vez que se inscreveram para qualquer curso. Esses números são números inteiros sequenciais que começaram em 0 há muitos anos e agora estão em milhões. Suponhamos que tenhamos uma classe de alunos do primeiro ano e queremos atribuir-lhes códigos hash com base nos números de seus alunos. Faz mais sentido usar os dois primeiros dígitos ou os dois últimos dígitos do número do estudante? Justifique sua resposta.

Exercício 5.2. Considere o esquema de hashing na Seção 5.1.1, e suponha $n = 2^d$ e $d \leq w/2$.

1. Mostre que, para qualquer escolha do multiplicador, z, existe n valores que possuem o mesmo código de hash. (Sugestão: isso é fácil

e não exige nenhuma teoria de números).

2. Dado o multiplicador, z, descreva n valores de modo que todos tenham o mesmo código de hash. (Sugestão: isto é mais difícil e requer uma teoria básica de números).

Exercício 5.3. Prove que o limite $2/2^d$ no Lema 5.1 seja o melhor limite possível mostrando que, se $x = 2^{w-d-2}$ e $y = 3x$, entãoPr{hash(x) = hash(y)} = $2/2^d$. (Observe as representações binárias de zx e $z3x$, e use o fato de que $z3x = zx + 2zx$.)

Exercício 5.4. Prove novamente o Lema 5.4 usando a versão completa da Aproximação de Stirling dada em Seção 1.3.2.

Exercício 5.5. Considere a seguinte versão simplificada do código para adicionar um elemento x a um LinearHashTable, que simplesmente armazena x na primeira entrada *nil* do array encontrada. Explique por que isso pode ser muito lento, dando um exemplo de uma sequência de $O(n)$ add(x), remove(x) e find(x) operações que levariam na ordem de n^2 tempo de executar.

```
add_slow(x)
  if 2·(q+1) > length(t) then resize()
  i ← hash(x)
  while t[i] ≠ nil do
    if t[1] ≠ del and x = t[i] then return false
    i ← (i+1) mod len(t[i])
  t[i] ← x
  n ← n+1
  q ← q+1
  return true
```

Exercício 5.6. Versões iniciais do método Java hash_code() para a classe String trabalhada ao não usar todos os caracteres encontrados em strings longos. Por exemplo, para uma sequência de dezesseis caracteres, o código hash foi computado usando apenas os oito caracteres de índices pares. Explique por que esta foi uma idéia muito ruim dando um exemplo de grande conjunto de strings que todos têm o mesmo código hash.

Exercício 5.7. Suponha que você tenha um objeto composto por dois números w-bit, x e y. Mostre porque $x \oplus y$ não faz um bom código hash para o seu objeto. Dê um exemplo de um grande conjunto de objetos que todos teriam código hash 0.

Exercício 5.8. Suponha que você tenha um objeto composto por dois números w-bit, x e y. Mostre porque $x + y$ não faz um bom código hash para o seu objeto. Dê um exemplo de um grande conjunto de objetos que todos teriam o mesmo código hash.

Exercício 5.9. Suponha que você tenha um objeto composto por dois números w-bit, x e y. Suponha que o código hash para seu objeto seja definido por alguma função determinística $h(x, y)$ que produz um inteiro w-bit inteiro. Prove que exista um grande conjunto de objetos que tenham o mesmo código hash.

Exercício 5.10. Seja $p = 2^w - 1$ para algum inteiro positivo w. Explique porque, para um inteiro positivo x

$$(x \bmod 2^w) + (x \operatorname{div} 2^w) \equiv x \bmod (2^w - 1) \ .$$

(Isso nos dá um algoritmo $x \bmod (2^w - 1)$ para repetidamente "settar" até $x \leq 2^w - 1$.)

Exercício 5.11. Encontre algumas implementações de tabelas hash comumente usadas, tais como as implementações ou HashTable ou LinearHashTable neste livro e crie uma programa que armazena inteiros nesta estrutura de dados de modo que haja números inteiros, x, de modo que find(x) demore tempo linear. Ou seja, encontre um conjunto de n inteiros para os quais existem cn elementos que dispersem para o mesmo local da tabela.

Dependendo de quão boa seja a implementação, você poderá fazer isso apenas inspecionando o código para a implementação, ou talvez seja necessário que escreva algum código que faça inserções e pesquisas de teste, medindo quanto tempo demora para adicionar e encontrar valores específicos. (Isso pode ser, e tem sido usado, para lançar ataques de negação de serviço em servidores web [17].)