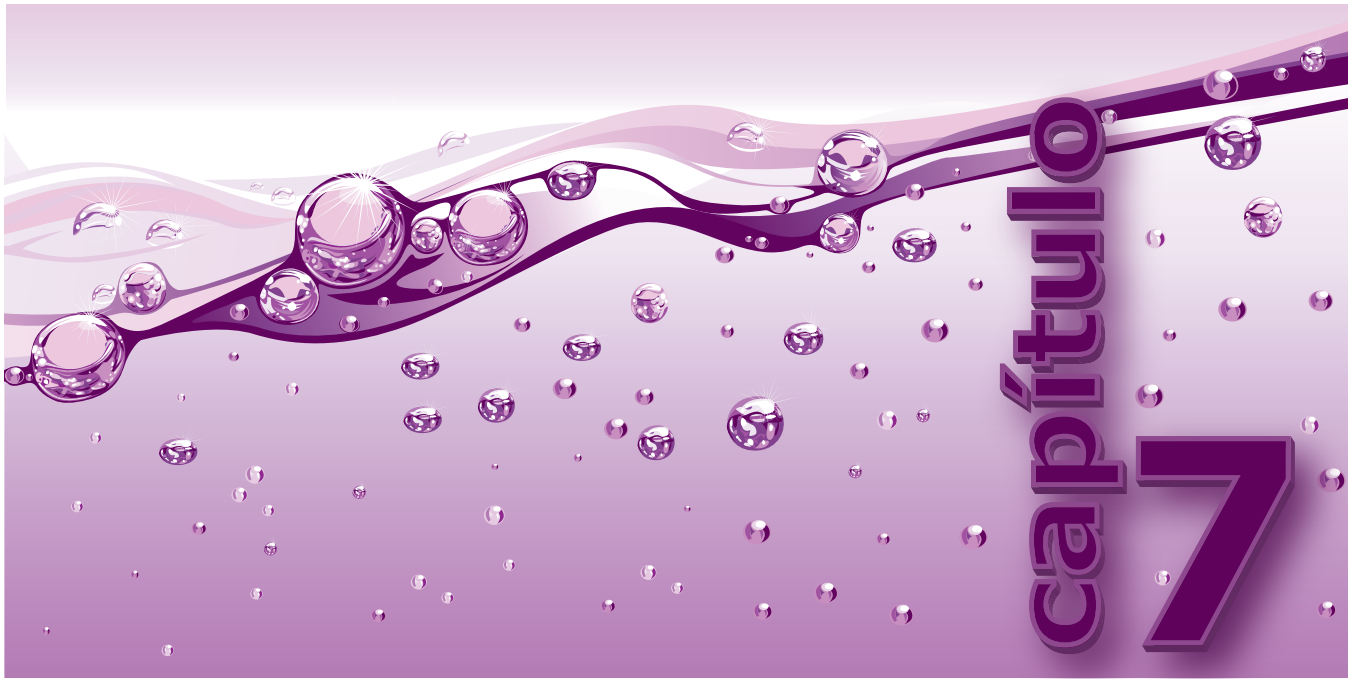

Dispersão em Memória Principal

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Dispersão
  - ❒ Colisão
  - ❒ Coletor
  - ❒ Sondagem
  - ❒ Dispersão cuco
  - ❒ Função de dispersão
  - ❒ Valor de dispersão
  - ❒ Tabela de dispersão
  - ❒ Método de Horner
  - ❒ Função de sondagem
  - ❒ Dispersão com encadeamento e com endereçamento aberto
  - ❒ Agrupamento
  - ❒ Filtro de Bloom
- Expressar limitações de tabelas de dispersão
- Descrever pelo menos três métodos de dispersão
- Discutir as propriedades de uma boa função de dispersão
- Explicar o uso de número primo em dispersão
- Explicar por que o uso de divisão modular é imprescindível em tabelas de dispersão
- Efetuar redimensionamento de dispersão
- Testar uma função de dispersão
- Implementar tabela de busca com dispersão cuco, encadeamento e endereçamento aberto

objetivos

ESTE CAPÍTULO LIDA com uma das mais importantes técnicas de implementação de tabelas de busca. Essa técnica é denominada **dispersão** (*hashing*, em inglês) e consiste basicamente em associar cada chave de uma tabela de busca a um índice dela. Por meio dessa técnica, espera-se obter custo temporal $\theta(1)$ para operações básicas de busca, inserção ou remoção. Uma tabela de busca que utiliza essa abordagem é denominada **tabela de dispersão**. Na prática, apesar de tabelas de dispersão serem uma das estruturas de dados mais fáceis de programar, esse objetivo não é fácil de ser obtido devido à possibilidade de ocorrer colisões, o que acontece quando duas ou mais chaves são associadas a um mesmo índice. A **Seção 7.1** apresentará os principais conceitos e a terminologia usados em dispersão.

Os problemas centrais de dispersão que serão explorados neste capítulo são:

- **Criação de funções de dispersão** que distribuam as chaves de modo uniforme e minimizem o número de colisões. Esse tópico será explorado na **Seção 7.2**.
- **Resolução de colisões**. Há vários métodos para lidar com colisões, que dependem de como uma tabela de dispersão é organizada. Este capítulo discute as duas técnicas mais simples e comuns de organização de tabelas de dispersão: encadeamento (v. **Seção 7.3**) e endereçamento aberto (v. **Seção 7.4**). A técnica de dispersão cuco, que é relativamente recente e pouco explorada, também será investigada (v. **Seção 7.6**).

Aspectos gerais de dispersão e o uso de dispersão para implementação de tabelas de busca em memória principal serão examinados no presente capítulo. O uso de dispersão em memória secundária será discutido no **Capítulo 8**. Sugere-se ao leitor que examine o **Apêndice B** antes de prosseguir, pois ele é importante para o completo entendimento dos tópicos abordados nesses dois capítulos.

7.1 Conceitos, Terminologia e Aplicações

7.1.1 Funções e Tabelas de Dispersão

Em geral, dispersão é uma técnica usada para acessar chaves numa tabela de busca em tempo (idealmente) constante, processando cada chave para identificar sua posição na tabela. Teoricamente, esse objetivo não é impossível. Considere como exemplo uma empresa que possui um número fixo de *100* empregados identificados por um número de identificação entre *0* e *99*. Suponha ainda que, se um empregado dessa empresa deixar o emprego por alguma razão, outro seja contratado e receba o mesmo número de identificação do antigo empregado. Então, nessa situação idealizada, se os registros dos empregados forem armazenados numa tabela indexada de *0* a *99*, pode-se acessar diretamente qualquer registro de empregado utilizando-se o número de registro de cada empregado como índice da referida tabela. De fato, nesse caso, a função de dispersão é a função identidade que associa cada chave (i.e., o referido número de identificação) a um índice da tabela de dispersão. Essa função é injetora, já que existe uma correspondência um a um entre as chaves e os índices da tabela.

Função de dispersão é uma função matemática usada para mapear chaves em índices de uma tabela de busca indexada (v. **Capítulo 3**). Nesse contexto, o resultado da aplicação de uma função de dispersão sobre uma chave é denominado **valor de dispersão**. Uma chave pode ser de qualquer tipo, mas seu valor de dispersão é sempre um inteiro não negativo (porque índices não podem ser negativos). Um **algoritmo** (ou **método**) **de dispersão** descreve os passos que uma função de dispersão deve seguir para calcular o valor de dispersão de uma chave.

Neste contexto, cada espaço de uma tabela de dispersão é frequentemente denominado **coletor** (*bucket*, em inglês). Um coletor de uma tabela de dispersão pode, por exemplo, ser representado por uma lista encadeada (v. **Seção 7.3**) ou por um bloco num meio de armazenamento externo (v. **Capítulo 8**).

Uma tabela de dispersão **diretamente endereçável** é aquela para a qual existe uma função de dispersão que garante que duas chaves nunca resultem no mesmo valor de dispersão. Essa é a situação ideal, mas raramente é possível na prática. A tabela de dispersão apresentada como exemplo no primeiro parágrafo desta seção é diretamente endereçável.

Na prática, essa perfeita relação entre chaves e posições numa tabela de dispersão não é fácil de se obter ou manter. Considere uma pequena alteração no referido exemplo. Ou seja, suponha que a mesma empresa decida usar um número de identificação com seis dígitos de modo que esse número possa ser qualquer valor entre *0* e *999999*. Nesse caso, é impraticável usar uma tabela com *1.000.000* de elementos, dos quais apenas *100* são realmente necessários para assegurar que cada chave que representa um empregado seja armazenada numa posição fácil de ser obtida (i.e., usando a função identidade).

Numa situação como essa, a solução mais simples é usar uma função de dispersão que associe cada chave ao resto da divisão dela pelo tamanho da tabela, como mostra a **Figura 7–1**, que ilustra a atuação de uma função de dispersão cujo domínio é um conjunto de chaves inteiras positivas e que obtém o valor de dispersão de cada chave calculando o resto da divisão dela por *100*. Ou seja, se *c* for a chave, seu valor de dispersão é calculado como[1]: *f(c) = c mod 100*.

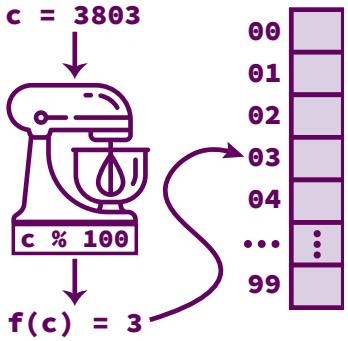

Figura 7–1: Uso de Função de Dispersão para Determinar Posição em Tabela

A **Figura 7–2 (a)** mostra uma tabela de dispersão que armazena as chaves *21804*, *33300*, *50001*, *31702* e *65606*, que foram acrescentadas nessa ordem usando a função de dispersão descrita no último parágrafo.

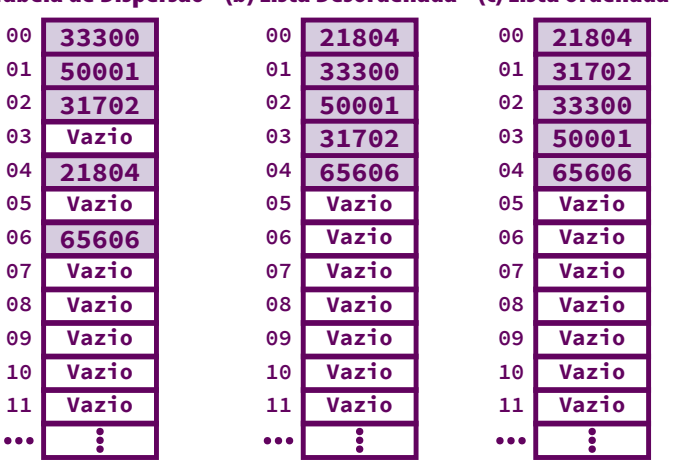

Figura 7–2: Comparação entre Tabela de Dispersão e Tabela Indexada

[1] Nessa fórmula, *mod* denota a operação resto da divisão em matemática. Em C, essa operação é representada pelo símbolo **%**.

Note na **Figura 7–2 (a)** que a função não preenche os espaços da tabela de dispersão sequencialmente, como ocorre com a tabela de busca representada como lista não ordenada mostrada na **Figura 7–2 (b)**, ou na tabela ordenada ilustrada na **Figura 7–2 (c)**. Note que, como não foram ainda inseridas quaisquer chaves que produzem os valores de dispersão *3* e *5*, essas posições da tabela de dispersão são consideradas vazias.

7.1.2 Colisões e suas Resoluções

A maioria das funções de dispersão usada na prática mapeia várias chaves num mesmo coletor de uma tabela de dispersão. Uma **colisão** ocorre quando duas ou mais chaves são mapeadas numa mesma posição da tabela de dispersão. Por exemplo, usando a função de dispersão descrita no exemplo da **Figura 7–1**, as chaves *02166* e *92266* são ambas mapeadas na mesma posição *66* da tabela de dispersão.

Colisões são perfeitamente naturais para uma função de dispersão e todas as implementações de tabelas de dispersão lidam com essa situação de alguma maneira. Uma boa função de dispersão é aquela que minimiza colisões dispersando as chaves uniformemente por toda a tabela de dispersão. Fala-se em *minimizar colisões*, porque é muito difícil evitá-las completamente.

A abordagem a ser utilizada quando ocorrem colisões numa tabela de dispersão depende de como os coletores são organizados na tabela. Existem duas abordagens básicas para lidar com colisões numa tabela de busca:

1. Tabela de dispersão com **encadeamento**. Nessa abordagem, que será discutida na **Seção 7.3**, os coletores são listas encadeadas. Portanto lidar com colisões aqui significa simplesmente armazenar todas as chaves que colidem numa mesma lista encadeada.
2. Numa tabela de dispersão com **endereçamento aberto** ou com **sondagem** (v. **Seção 7.4**), os coletores são os próprios elementos da tabela, como mostra a **Figura 7–2**. Essa abordagem admite vários esquemas de resolução de colisão denominados **sondagens**. Uma sondagem é semelhante a uma busca numa tabela de busca comum, mas ela também inclui o exame de coletores para verificar se eles estão vazios. Os tipos de sondagem mais comuns são:
   - **2.1** **Sondagem linear** é uma forma de resolução de colisões por meio de uma visita sequencial aos coletores de uma tabela de dispersão a partir da posição retornada pela função de dispersão utilizada.
   - **2.2** **Sondagem quadrática** consiste em usar uma fórmula de resolução de colisões, tal como $(d \pm i^2) \bmod m$, sendo d o valor de dispersão de uma chave, i o número de vezes que a fórmula é aplicada e m o tamanho da tabela de dispersão.
   - **2.3** **Sondagem por dispersão dupla** consiste em resolver colisões por meio de uma função de dispersão secundária.

7.1.3 Fator de Carga

O **fator de carga** α de uma tabela de dispersão é definido como:

$$\alpha = n/m$$

em que n é o número de chaves da tabela e $m \neq 0$ é o número de coletores nos quais as chaves podem ser armazenadas.

O fator de carga de uma tabela de dispersão indica o número médio de chaves que se espera encontrar num coletor, supondo a existência de dispersão uniforme. Esse conceito é de suma importância na análise de desempenho de tabelas de dispersão.

7.1.4 Aplicações de Dispersão

Existem inúmeras aplicações de tabelas de dispersão. Por exemplo, compiladores usam tabelas de dispersão numa estrutura de dados conhecida como **tabela de símbolos** para acessar identificadores definidos num

programa-fonte. Tabelas de dispersão são ideais para esse problema, pois identificadores são tipicamente curtos, de modo que os cálculos de seus valores de dispersão podem ser efetuados rapidamente. Além disso, o armazenamento desses identificadores não precisa ser mantido em nenhuma ordem.

Outro uso de tabelas de dispersão é em verificação ortográfica. Nesse caso, calculam-se valores de dispersão para todas as palavras de uma lista de palavras (dicionário), de maneira que a ortografia de uma palavra num texto possa ser verificada com custo temporal constante. Novamente, tabelas de dispersão são convenientes nesse caso, porque a ordem das palavras na tabela não é importante.

Tabelas de dispersão também são frequentemente usadas para implementar arquivos de cache, como, por exemplo, aqueles usados em navegadores da internet.

7.2 Funções de Dispersão

Uma implementação de tabela de dispersão é apenas tão boa quanto a função de dispersão utilizada em conjunto com ela, pois uma má escolha de função de dispersão para uma tabela pode resultar em baixa eficiência em operações nessa tabela. Portanto escolher uma função de dispersão adequada é a primeira decisão que deve ser tomada quando se implementa uma tabela de dispersão. Embora a ideia de dispersão tenha sido concebida na década de 1950, a elaboração de boas funções de dispersão é ainda um tópico de intensa pesquisa.

Desenvolver uma boa função de dispersão é difícil. De fato, se você tiver que escolher uma função de dispersão antes que as chaves sejam conhecidas, é provavelmente impossível evitar o cenário do pior caso em que todas as chaves terminam no mesmo coletor e o custo temporal de busca torna-se $\theta(n)$.

Muitas vezes, uma função de dispersão fraca é usada porque ela não é imprescindível numa situação específica. Uma função de dispersão ruim pode tornar um algoritmo de busca mais lento, mas, desde que a tabela de dispersão lide corretamente com os casos de colisão, é possível que um mau desempenho não seja notado. Em muitos casos em que o número de chaves n não é muito grande, o custo de pior caso $\theta(n)$ pode não ser percebido.

Frequentemente, chaves que seguem um determinado padrão resultam em valores de dispersão que também seguem algum padrão. Uma função ideal de dispersão distribui igualmente as chaves entre todos os coletores (não importando como as chaves possam estar relacionadas entre si), de maneira que qualquer padrão entre chaves que possa levar a uma distribuição desigual seja corrigido.

Um modo de minimizar colisões é usar uma tabela de dispersão com um número maior de coletores do que é realmente necessário para acomodar as chaves esperadas, de modo a aumentar o intervalo de valores de dispersão. Portanto escolher o tamanho de uma tabela de dispersão envolve um conflito de escolha entre espaço versus tempo. Quanto maior for o intervalo de índices da tabela, menor será a possibilidade de duas chaves colidirem. Contudo alocar um array que contém um grande número de coletores vazios desperdiça espaço.

Uma função de dispersão mal projetada pode afetar consideravelmente o desempenho da tabela de dispersão que a utiliza. Assim, qualquer que seja a implementação de tabela de dispersão, é recomendável testar a função de dispersão utilizada quando a eficiência das operações for realmente importante. O **Apêndice C** apresenta uma breve análise das funções de dispersão prontas mais comumente encontradas na internet.

7.2.1 Propriedades Desejáveis

Um algoritmo de dispersão é selecionado com base no fato de ele ser melhor do que outros. O problema é definir critérios que possam ser usados para decidir quando um algoritmo de dispersão é melhor do que outro. Dois critérios essenciais são o grau com que o algoritmo distribui uniformemente chaves sobre o intervalo de valores e a rapidez com que ele é executado, mas outros critérios relevantes que uma função de dispersão deve satisfazer também serão discutidos nesta seção.

Simplicidade

Simplicidade significa que uma boa função de dispersão não deve desperdiçar espaço em memória e deve ser fácil de entender e depurar (para facilitar a vida do programador).

Uma função exótica que produza valores de dispersão para todas as chaves conhecidas pode não funcionar se o domínio das chaves for alterado posteriormente. Tal função pode fazer com que o programador responsável por ela desperdice muito tempo tentando adaptá-la para um novo conjunto de chaves.

Eficiência

Eficiência é uma medida da rapidez com que uma função de dispersão produz valores de dispersão para as chaves de uma tabela. Quando as operações com tabelas de dispersão são analisadas, geralmente, assume-se que funções de dispersão apresentam custo temporal $\theta(1)$. É por isso que se diz que busca numa tabela de dispersão tem, em média, custo temporal $\theta(1)$. Como a maior vantagem de uma tabela de dispersão é a rapidez com que são efetuadas operações de busca, inserção e remoção, se uma função de dispersão for lenta, esse desempenho será degradado. Uma função de dispersão que envolve muitas multiplicações e divisões, por exemplo, pode ser ineficiente.

Uniformidade

Uniformidade é uma medida de como uma função de dispersão distribui bem os valores de dispersão das chaves de uma tabela de dispersão. Uma função de dispersão ideal distribui um conjunto de chaves num intervalo $[0..m - 1]$ de maneira uniforme e independente, de modo que todos os valores de dispersão sejam equiprováveis. Ou seja, uma função de dispersão que distribui os valores de dispersão de maneira uniforme minimiza colisões e preenche a tabela de dispersão mais igualmente. Portanto essa é a principal qualidade que se deve esperar de uma boa função de dispersão.

Idealmente, essa propriedade deve ser independente do tamanho da tabela de dispersão e vice-versa. Isso assegura que, independentemente de tamanho m da tabela e do número n de chaves, o número esperado de chaves por coletor será aproximadamente o mesmo: $\theta(n/m)$. Ou seja, as chaves serão distribuídas igualmente entre os coletores da tabela.

Determinismo

Uma função de dispersão deve ser **determinística**, o que significa que, para uma dada chave, ela deve sempre gerar o mesmo valor de dispersão. Esse requisito exclui, por exemplo, funções de dispersão que usam geradores de números pseudoaleatórios que dependem do instante em que são executadas [p. ex., em C, não se deve usar **srand()** e **rand()** para obter valores de dispersão]. Caso contrário, após serem gerados, esses valores de dispersão precisam ser armazenados de modo persistente (i.e., em arquivo).

Independência

Uma boa função de dispersão deve ser independente das chaves sobre os quais ela é aplicada. Ou seja, ela deve ser eficaz para qualquer tipo de chave e esse requisito exclui peremptoriamente o uso de funções de dispersão talhadas para um conjunto específico de chaves. O programador pode, obviamente, elaborar sua própria função de dispersão para chaves específicas, mas talvez ela deixe de funcionar tão logo essas chaves sejam alteradas.

Avalanche

Uma função de dispersão apresenta **avalanche** quando dois valores de dispersão são substancialmente diferentes, mesmo se as chaves que lhes deram origem diferirem em apenas um único bit. Esse efeito ajuda na distribuição dos valores de dispersão porque chaves muito próximas entre si não terão os mesmos valores de dispersão.

7.2.2 Tipos de Chaves

Uma função de dispersão não deve depender do tipo de chave sobre o qual ela atua, mas isso não constitui um problema porque qualquer chave pode ser vista como um array de bytes (i.e., um array de elementos do tipo **char** em C). De qualquer modo, funções de dispersão mais simples podem ser criadas levando em consideração o tipo específico das chaves sob consideração, como será visto a seguir.

Chaves Inteiras

A técnica mais simples comumente utilizada para calcular valores de dispersão de chaves inteiras não negativas é denominada **dispersão modular**. Uma função de dispersão modular simplesmente obtém como valor de dispersão de uma chave o resto da divisão dela pelo tamanho da tabela de busca (v. *método de divisão modular* na **página 366**). Contudo essa ideia só é razoável se as chaves não apresentarem nenhuma propriedade indesejável, como será visto na **Seção 7.2.3**.

Chaves Reais

Quando as chaves são números reais, uma maneira fácil de calcular valores de dispersão consiste em combinar as partes inteira e fracionária de cada chave usando alguma função de dispersão. Em C, a função **modf()** (`#include <math.h>`) pode ser usada para separar as partes inteira e fracionária de um valor do tipo **double**, como mostra o exemplo a seguir:

```
unsigned DispersaoReal(double chave, int tam)
{
   unsigned precisao = 1000000;
   double   pInteira,
            pReal = modf(chave, &pInteira);

   return ((unsigned) pInteira + (unsigned) (pReal*precisao))%tam;
}
```

Strings

Se as chaves forem strings, pode-se usar a mesma técnica que será descrita na **Seção 9.6** para implementação da digital de Rabin do algoritmo de casamento de strings de Karp e Rabin.

Chaves Compostas

Se as chaves forem constituídas por registros contendo campos inteiros, esses campos podem ser combinados de modo semelhante ao que ocorre com strings. Por exemplo, se o tipo `tData` for definido como:

```
typedef struct {
           int dia;
           int mes;
           int ano;
        } tData;
```

o valor de dispersão de uma chave do tipo `tData` pode ser calculado como:

```
unsigned DispersaoData(const tData *c, int tam)
{
   unsigned b = 31; /* Base */

   return (((c->dia*b + c->mes)%tam)*b + c->ano)%tam;
}
```

É muito importante que, no caso sob discussão, os campos da chave sejam levados em consideração individualmente, em vez de em conjunto, pois estruturas podem conter preenchimentos (v. questão **23** na **página 410**).

7.2.3 Métodos de Cálculo de Valores de Dispersão

Esta seção apresenta métodos (ou algoritmos) clássicos de cálculo de valores de dispersão e alguns deles não são comumente utilizados. O leitor deve estar atento para as observações sobre cada método, pois nem sempre um método apresentado aqui é adequado.

Aditivo

Geralmente, qualquer algoritmo de dispersão que conta essencialmente com uma operação comutativa terá uma distribuição ruim, pois, como será visto na **Seção 9.6**, essa função de dispersão fracassa ao lidar com permutações diferentes dos bytes de uma mesma chave.

Polinomial

O método de **dispersão polinomial** consiste em considerar informações posicionais dos componentes de uma chave e usar o método de Horner para calcular os valores de dispersão. Esse método é usado pelo algoritmo de Karp e Rabin que será discutido na **Seção 9.6**.

Tabular

Quando cada chave pode ser interpretada como uma sequência de componentes $c = c_0\, c_1 \ldots c_{d-1}$, para um dado valor $d > 0$, pode-se usar uma função de dispersão que requer uma simples consulta a uma tabela. Tal função é conhecida como **função de dispersão tabular**.

Supondo que cada c_i $(0 \leq i < d)$ encontra-se no intervalo $[0 .. k - 1]$, são criadas d tabelas, T_0, T_1, ..., T_{d-1}, cada uma de tamanho k, de modo que cada $T_i[j]$ é um número aleatório escolhido no intervalo $[0 .. n - 1]$. Então calcula-se um valor de dispersão de c, $f(c)$, como:

$$f(c) = T_0[c_0] \oplus T_1[c_1] \oplus \ldots \oplus T_{d-1}[c_{d-1}]$$

Nessa fórmula, o símbolo $\oplus$ representa **disjunção exclusiva**, que é popularmente conhecida como *xor*. Em C, a operação *xor* é representada pelo operador ^ (v. **Apêndice B**).

Como os valores dos elementos das tabelas aludidas são escolhidos aleatoriamente, os valores de dispersão produzidos por uma função de dispersão tabular são razoavelmente aleatórios. Por exemplo, pode-se mostrar que ela causa colisão de duas chaves distintas com probabilidade $1/n$, que é o que se obteria usando uma função perfeitamente aleatória. Além disso, essa função usa apenas operações *xor* que, tipicamente, são executadas muito eficientemente.

Um exemplo de função de dispersão que usa a abordagem tabular é a função **JSW**, apresentada no **Apêndice C**.

Divisão Modular

O método de dispersão mais vulgar (literalmente) é aquele que usa apenas **divisão modular** (ou **resto de divisão**) para calcular valores de dispersão. O formato geral desse tipo de função é:

$$f(c) = c \bmod m$$

sendo c uma chave inteira e m o tamanho da tabela de dispersão. Ou seja, usando dispersão modular, o valor de dispersão de cada chave c é calculado como o resto da divisão de c por m (em C, isso significa `c % m`).

O tamanho da tabela (m) precisa ser cuidadosamente escolhido, e é sempre uma boa ideia assegurar que ele seja um número primo. Se m não for primo, pode ser que nem todos os bits que constituem uma chave contribuam para seu valor de dispersão, fazendo com que os valores de dispersão obtidos não sejam distribuídos uniformemente. Por exemplo, se as chaves forem valores inteiros representados na base 10 e o valor de m for 10^x, então apenas os x dígitos menos significativos da chave serão levados em conta[2]. Usando o mesmo racio-

[2] Para tornar o exemplo mais palpável, suponha que $m = 100$. Nesse caso, as chaves 2045, 1145 e quaisquer outras chaves que terminem em 45 terão o mesmo valor de dispersão. Em geral, para esse valor de m, quaisquer chaves que tenham a mesma centena colidirão.

cínio, valores de m que sejam potências de *2* também devem ser evitados, pois, se $m = 2^x$, $f(c)$ resulta nos x bits de menor ordem da chave c. Usualmente, escolhe-se m como um número primo que não seja próximo de uma potência de *2* (p. ex., *7* é primo, mas não é uma boa escolha, pois *7* é próximo de *8*, que é uma potência de *2*).

A vantagem de uma função de dispersão que usa divisão modular é sua simplicidade. Quer dizer, quando as chaves são valores inteiros aleatórios, dispersão modular não apenas é simples de calcular como também distribui as chaves igualmente. Frequentemente, entretanto, é necessário usar uma função de dispersão mais bem elaborada para obter uma boa distribuição de valores de dispersão. Tipicamente, o método de dispersão modular é utilizado em combinação com algum outro método para assegurar que os valores de dispersão se encontram dentro dos limites dos índices da tabela de dispersão.

Multiplicativo

O **método de multiplicação** para obtenção de valores de dispersão considera que as chaves são numéricas e consiste dos seguintes passos:

1. Multiplica-se a chave c por uma constante real a, tal que $0 < a < 1$. Tipicamente, usa-se o seguinte valor para a:
$$a = \frac{\sqrt{5}-1}{2} = 0.618$$
2. Extrai-se a parte fracionária do resultado obtido no passo anterior.
3. Multiplica-se o valor obtido no último passo pelo tamanho m da tabela de dispersão e considera-se o piso do resultado obtido.

De acordo com a descrição acima, a função de dispersão é dada por:

$$f(c) = \lfloor m \cdot modf(c \cdot a) \rfloor$$

Nessa fórmula, $modf(r)$ resulta na parte fracionária de r.

Supondo, por exemplo, que uma tabela de dispersão tenha *2000* coletores (m), o valor de dispersão da chave *6341* (c) será obtido como:

$$\begin{aligned} f(6341) &= \lfloor 2000 \cdot modf(6341 \cdot 0{,}618) \rfloor \\ &= \lfloor 2000 \cdot modf(3918{,}738) \rfloor \\ &= \lfloor 2000 \cdot 0{,}738 \rfloor = 1476 \end{aligned}$$

A função `DispersaoMultiplicacao()` a seguir mostra uma implementação em C do método de multiplicação.

```
unsigned DispersaoMultiplicacao(unsigned chave, unsigned tamTabela)
{
   double a = (sqrt(5) - 1)/2,
          pInteira,
          pFracionaria = modf(chave*a, &pInteira);

   return (unsigned) floor(tamTabela*pFracionaria);
}
```

A desvantagem desse método de dispersão é que ele é adequado apenas para chaves inteiras.

Disjunção Exclusiva (XOR)

O **método de dispersão xor** mistura repetidamente os bytes de uma chave para produzir um valor de dispersão aparentemente aleatório. A função `DispersaoXOR()` ilustra o uso desse método no cálculo de valores de dispersão de chaves que são strings.

```
unsigned int DispersaoXOR(const char *chave)
{
   unsigned int dispersao = *chave;

   while (*chave)
      dispersao ^= *chave++;

   return dispersao;
}
```

Infelizmente, esse algoritmo é simples demais para funcionar apropriadamente com muitos tipos de chaves. O **estado interno** do valor de dispersão, representado pela variável `dispersao` na função `DispersaoXOR()`, não é misturado suficientemente para que se obtenha avalanche (v. **Seção 7.2.1**). Além disso, uma única operação *xor* não é efetiva para misturar bem o estado interno. Assim a distribuição resultante, embora seja melhor do que os métodos aditivo e multiplicativo de dispersão, não é muito bom.

Rotativo

O método de **dispersão rotativa** é semelhante ao método de dispersão *xor*, exceto que, em vez de simplesmente misturar cada byte da chave com o estado interno, ele efetua uma mistura do estado interno antes de combiná-lo com cada byte da chave. Essa mistura extra é suficiente para dotar a dispersão rotativa com uma distribuição de chaves bem melhor do que aquela obtida com os métodos anteriores. Muitas vezes, a dispersão rotativa é suficiente e pode ser considerada aceitável.

A função `DispersaoRotativa()` a seguir apresenta um exemplo de implementação de função de dispersão que usa o método rotativo.

```
unsigned DispersaoRotativa(const char *chave)
{
   unsigned int dispersao = 0;

   while (*chave)
       dispersao = (dispersao << 4) ^ (dispersao >> 28) ^ (*chave++);

   return dispersao;
}
```

Dispersão por Mistura

Mistura é um método de dispersão que divide uma chave em várias partes e concatena ou aplica disjunção exclusiva (*xor*) a algumas das partes para formar o valor de dispersão esperado. Em outras palavras, esse método de dispersão envolve separar a chave em várias partes e concatenar ou aplicar *xor* a algumas dessas partes para obter o valor de dispersão. O **Apêndice B** apresenta um exemplo de criação de uma função de dispersão por mistura.

7.2.4 Anatomia de uma Função de Dispersão Bem Elaborada

Uma função de dispersão bem elaborada é composta de quatro partes principais, como mostra o diagrama da **Figura 7–3**.

O valor de dispersão, representado por d_i na **Figura 7–3**, é o valor calculado no instante i pela função de dispersão f, que recebe um valor inicial d_0, e é sucessivamente modificado por cada símbolo de entrada x_i. O símbolo de entrada x_i é a parcela da chave de entrada que alimenta a função no instante i e faz com que o valor de dispersão resulte em d_i.

Uma boa função de dispersão deve exibir confusão e difusão. **Difusão** é observada quando um único bit alterado na entrada afeta muitos bits na saída. Mais precisamente, quando um bit de uma chave é alterado, seu valor de dispersão muda substancialmente. **Confusão** ocorre quando um bit de saída depende de vários bits de

entrada. Em outras palavras, cada bit de um valor de dispersão depende de vários bits da respectiva chave. Se uma função de dispersão exibir ambas as propriedades, ela será robusta. Confusão e difusão[3] tornam o valor de dispersão resultante suficientemente imprevisível.

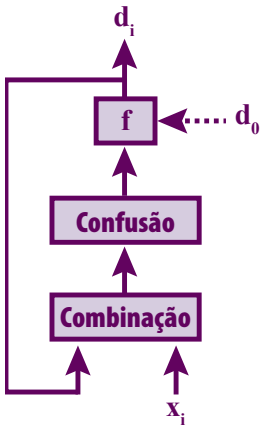

FIGURA 7–3: ANATOMIA DE UMA FUNÇÃO DE DISPERSÃO

O passo de **combinação** na **Figura 7–3** combina o valor de dispersão d_{i-1} com o próximo símbolo de entrada x_i. Essa operação pode ser tão simples quanto somar esse símbolo ao valor corrente de dispersão, mas é preferível que seja uma operação cujo resultado seja muito mais difícil de prever. O passo de **confusão** mistura o valor de dispersão resultante do passo de combinação e é necessário para atenuar qualquer ineficiência que o passo de combinação possa exibir. Se houvesse um passo de combinação perfeito, o passo de confusão seria desnecessário, mas, raramente, esse é o caso.

A escolha do valor inicial de dispersão d_0 não é crítico, pois, se a função de dispersão exibir passos de combinação e confusão suficientemente bons, o efeito do valor inicial será pequeno. Se, por outro lado, a função de dispersão for realmente ruim (p. ex., uma simples soma dos símbolos de entrada), esse valor inicial não irá ajudar em nada. Frequentemente, usa-se zero como valor de d_0.

Um exemplo de função de dispersão que usa o esquema descrito acima é a função **JOAAT** (v. **Apêndice C**):

```
unsigned int DispersaoJOAAT(const char *chave)
{
   unsigned int dispersao = 0; /* Valor inicial */

      /* Combinação */
   while (*chave) {
      dispersao += *chave++;
      dispersao += (dispersao << 10);
      dispersao ^= (dispersao >> 6);
   }

     /* Confusão */
   dispersao += (dispersao << 3);
   dispersao ^= (dispersao >> 11);
   dispersao += (dispersao << 15);

   return dispersao;
}
```

[3] Do ponto de vista da teoria de informação de Shannon — v. **Bibliografia**, difusão significa que, se um único bit da chave for alterado, aproximadamente a metade dos bits do valor de dispersão deve ser alterada e vice-versa (i.e., se um bit do valor de dispersão for alterado, aproximadamente a metade dos bits da chave deve ser alterada). De acordo com a teoria de Shannon, confusão significa que cada bit do valor de dispersão deve depender de várias partes da chave.

7.2.5 Testando uma Função de Dispersão

Em geral, criar uma função de dispersão realmente boa é difícil e, em muitos casos, talvez você precise encontrar uma função pronta que exiba boas propriedades e tenha sido bem testada. O **Apêndice C** apresenta várias boas funções de dispersão prontas dentre as quais você poderá escolher aquela que for mais adequada para uma dada situação.

O teste mais importante para uma função de dispersão é aquele que verifica se ela exibe uma distribuição uniforme para uma amostra das chaves esperadas. Nenhuma função de dispersão é a melhor para qualquer conjunto possível de chaves, o que justifica a oferta de várias funções prontas no referido apêndice. Algumas vezes, numa situação específica, uma boa função de dispersão funcionará melhor do que as demais, e essa melhor função deve ser usada. Outras vezes, todas elas apresentarão o mesmo desempenho e você pode escolher qualquer uma delas. O ponto central é que uma função de dispersão (pronta ou personalizada) não deve jamais ser usada sem ser testada.

Criar um teste para distribuição uniforme é uma tarefa simples. É preciso apenas usar uma amostra das chaves esperadas e inseri-las numa tabela de dispersão encadeada (v. **Seção 7.3**) utilizando a função de dispersão em tela. Então calculando-se os comprimentos das listas, pode-se determinar quantas colisões ocorreram (v. **Seção 7.8.1**).

7.2.6 Recomendações Práticas

Na prática, construir uma função de dispersão envolve mais tentativa e erro com uma pitada de teoria do que o acompanhamento de um procedimento bem definido. Tipicamente, para obter uma distribuição uniforme desejável de valores de dispersão, a parte crítica da elaboração de uma função de dispersão consiste em manipular constantes e operadores para verificar quais deles funcionam melhor. Por exemplo, apesar de boas funções de dispersão serem difíceis de obter e requererem cuidadosa análise, em muitos casos, muitas funções de dispersão prontas famosas seguem uma abordagem relativamente simples que consiste no seguinte:

1. Atribua um valor arbitrário inicial à variável que armazenará o valor de dispersão:

```
unsigned int x = ALGUM_VALOR;
```

2. Para cada byte *b* da chave e para um dado operador `OP` (tipicamente de baixo nível) efetue a seguinte atribuição:

```
x = x OP b;
```

Não se pode afirmar com certeza que a elaboração de qualquer função de dispersão de natureza pragmática segue sempre essa metodologia, mas existem evidências que sugerem que seja assim.

Apesar de não existir nenhum procedimento efetivo a ser seguido que garanta a obtenção de uma boa função de dispersão para cada situação, existem recomendações práticas que, se seguidas, poderão, pelo menos, garantir que ela não seja considerada de má qualidade. Essas recomendações mínimas serão enumeradas a seguir.

- **Use todos os componentes de cada chave**. Cada parte relevante de uma chave deve contribuir para o valor de dispersão. Quanto mais porções da chave contribuírem para seu valor de dispersão, mais provável será que as chaves sejam dispersas igualmente em todo o intervalo de índices de uma tabela de dispersão.
- **Não use dados irrelevantes**. Informações redundantes ou que não são relevantes não devem contribuir para o valor de dispersão. Considere, por exemplo, CPF como chave. Nesse caso, todos os dígitos que constituem a parte principal de um CPF devem ser levados em consideração. Entretanto o mesmo não é recomendável para dígitos de verificação, visto que eles já são obtidos desses dígitos principais e, portanto, não acrescentam qualquer informação útil e são considerados redundantes. Se os *números* de CPF incluírem separadores (p. ex., – ou /), eles também não devem ser levados em conta, já que todas as chaves incluem esses separadores e também são considerados redundantes.

7.3 Dispersão com Encadeamento

7.3.1 Conceitos

Numa tabela de **dispersão com encadeamento,** uma lista encadeada é associada a cada índice da tabela. Nesse esquema de armazenamento de coletores, cada chave cujo valor de dispersão resulta no mesmo índice é inserida na lista encadeada associada a esse índice. Na prática, uma tabela de dispersão com encadeamento consiste num array de listas encadeadas. Cada uma dessas listas encadeadas forma um coletor no qual são colocadas todas as chaves que possuem um mesmo valor de dispersão.

Nessa abordagem de organização, se as colisões resultarem em listas encadeadas muito longas, o custo temporal das operações básicas poderá ser $\theta(n)$ no pior caso. Por exemplo, imagine uma empresa cujos registros de empregados são identificados com chaves de seis dígitos. Então é criada uma tabela de dispersão contendo 100 listas encadeadas para armazenar as chaves dos 500 empregados da empresa. Nesse exemplo, a expectativa é obter uma média de cinco chaves por lista encadeada utilizando-se a função de dispersão *c mod 100*.

Essa função de dispersão usa apenas os dois últimos dígitos do número de identificação como valor de dispersão. Assim, a tabela de dispersão planejada é aquela mostrada na **Figura 7–4 (a)**, ao passo que aquela realmente obtida usando esse esquema de dispersão é apresentada na **Figura 7–4 (b)**.

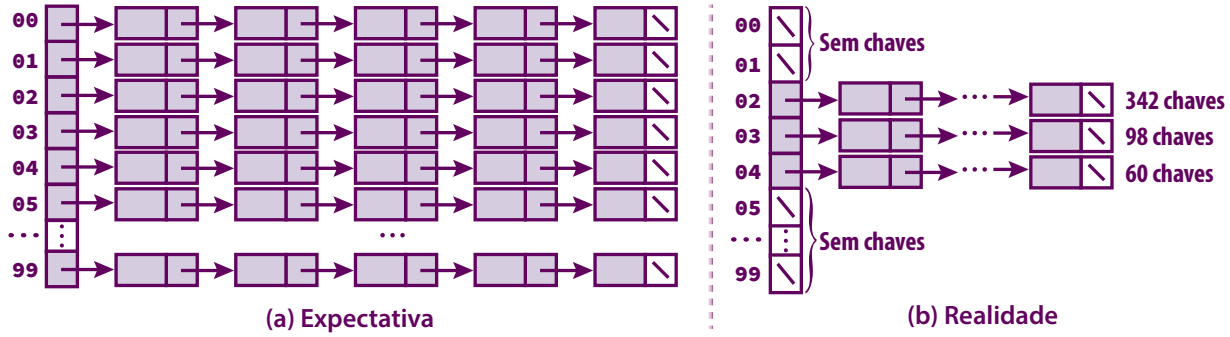

Figura 7–4: Dispersão com Encadeamento: Expectativa e Realidade

Uma aplicação comum de tabelas de dispersão encadeadas é na implementação de tabelas de símbolos (v. **Seção 7.1.4**), porque um compilador não pode prever quantos identificadores um programa pode ter.

Fator de Carga

Em dispersão com encadeamento, é normal ter m ou mais chaves numa tabela com m coletores. Assim o fator de carga (v. **Seção 7.1.3**) pode ser igual a *1* ou maior do que *1*. Por exemplo, se uma tabela tiver $m = 1500$ coletores e $n = 3000$ chaves, então $\alpha = 2$. Nesse exemplo, espera-se encontrar em média duas chaves em cada coletor.

Em dispersão com encadeamento, o fator de carga pode ser maior do que *1* sem afetar muito o desempenho das operações. Isso torna tabelas de dispersão com encadeamento especialmente indicadas quando é difícil prever quantas chaves serão armazenadas nelas.

Em dispersão com encadeamento, é recomendável usar um tamanho da tabela quase tão grande quanto o número de chaves esperadas (i.e., usando $\alpha \approx 1$). Quando o fator de carga ultrapassa *1*, aconselha-se redimensionar o tamanho da tabela. Esse redimensionamento é discutido na **Seção 7.5**. É também uma boa ideia ter um número primo como tamanho da tabela para assegurar uma boa distribuição de valores de dispersão.

Busca

Busca numa tabela de dispersão com encadeamento é simples e segue o algoritmo delineado na **Figura 7–5**.

Algoritmo BuscaEmDispersãoComEncadeamento
Entrada: Uma tabela de dispersão com encadeamento e uma chave de busca
Saída: O valor associado à primeira chave da tabela que casa com a chave de busca ou um valor indicando que a chave não foi encontrada
1. Aplique a função de dispersão sobre a chave de busca, obtendo assim o índice da lista encadeada na qual a chave deve estar armazenada
2. Efetue uma busca sequencial na lista encadeada que se situa no índice da tabela de dispersão correspondente ao valor de dispersão obtido (v. **Capítulo 3**)

Figura 7–5: Algoritmo de Busca em Dispersão com Encadeamento

Inserção

Se a tabela de dispersão permite chaves duplicadas, o algoritmo de inserção de uma chave consiste simplesmente em seguir os passos mostrados na **Figura 7–6**.

Algoritmo InsereEmDispersãoComEncadeamento
Entrada: O conteúdo de um novo elemento
Entrada/Saída: Uma tabela de dispersão com encadeamento
Saída: Um valor informando o sucesso da operação
1. Aplique a função de dispersão sobre a chave.
2. Utilize o valor de dispersão para localizar a lista encadeada da qual a chave fará parte
3. Insira a chave no início da lista encadeada (coletor) correspondente

Figura 7–6: Algoritmo de Inserção em Dispersão com Encadeamento

Se a tabela de dispersão não admite chaves duplicadas, o primeiro passo do algoritmo de inserção deve ser uma operação de busca para verificar se a chave a ser inserida já faz parte da tabela. Se a chave ainda não estiver presente na tabela, ela será inserida utilizando-se os **Passos 2** e **3**. Caso contrário, o procedimento de inserção é encerrado. A inserção é feita no início da lista encadeada porque é, obviamente, mais simples e eficiente.

Remoção

Em dispersão com encadeamento, o algoritmo de remoção segue os passos da **Figura 7–7**.

Algoritmo RemoveEmDispersãoComEncadeamento
Entrada: A chave a ser removida
Entrada/Saída: Uma tabela de dispersão com encadeamento
Saída: Um valor informando o sucesso da operação
1. Aplique a função de dispersão à chave a ser removida
2. Utilize o valor de dispersão obtido para localizar lista encadeada da qual a chave deve fazer parte
3. Remova a chave usando o algoritmo comum de remoção em lista encadeada

Figura 7–7: Algoritmo de Remoção em Dispersão com Encadeamento

7.3.2 Implementação

Conceitualmente, dispersão com encadeamento é mais simples do que outros esquemas de resolução de colisões. Contudo o código é mais longo porque ele deve incluir uma implementação para listas encadeadas.

Definições de Tipos

As seguintes definições de tipos são usadas na construção de uma tabela de dispersão com encadeamento:

```
typedef char tCEP[TAM_CEP + 1]; /* Tipo da chave */

   /* Tipo do conteúdo armazenado em cada nó das listas (coletores) */
typedef struct {
           tCEP chave;  /* Chave de um registro          */
           int  indice; /* Índice do registro no arquivo */
        } tCEP_Ind;

   /* Tipo de nó e tipo de ponteiro para nó de uma */
   /* lista encadeada que representa um coletor    */
typedef struct rotNoLSE {
           tCEP_Ind         conteudo;
           struct rotNoLSE *proximo;
        } tNoListaSE, *tListaSE;

   /* A tabela de dispersão será um array de listas encadeadas */
typedef tListaSE *tTabelaDE;
```

As operações de busca, inserção e remoção usam uma função de dispersão com o seguinte protótipo:

```
unsigned FDispersao(const char *chave);
```

Para permitir que a função de dispersão usada pelo programa seja escolhida durante a execução do programa, é necessário que as funções que executam as operações de busca, inserção e remoção recebam como parâmetro um ponteiro para a função de dispersão a ser usada. O tipo definido a seguir é usado para especificação desse parâmetro:

```
   /* Tipo de um ponteiro para uma função de dispersão */
typedef unsigned (*tFDispersao)(const char *);
```

Iniciação da Tabela

A função `CriaTabelaDE()` a seguir cria e inicializa os elementos de uma tabela de dispersão com encadeamento. O único parâmetro dessa função é o número de coletores da tabela de dispersão e ela retorna o endereço da tabela de dispersão criada.

```
tTabelaDE CriaTabelaDE(int nElementos)
{
   tTabelaDE tabela;
   int       i;

      /* Aloca o array que representa a tabela de dispersão */
   tabela = calloc(nElementos, sizeof(tListaSE));

      /* Certifica que houve alocação */
   ASSEGURA(tabela, "Impossivel alocar tabela de dispersao");

   for (i = 0; i < nElementos; ++i) /* Inicializa as listas encadeadas */
      IniciaListaSE(tabela + i);

   return tabela;
}
```

A função `CriaTabelaDE()` chama `IniciaListaSE()` para iniciar cada lista encadeada da tabela de dispersão. Essa última função foi discutida no **Capítulo 10** do **Volume 1**.

Busca

A função `BuscaDE()` executa uma busca simples numa tabela de dispersão com encadeamento e seus parâmetros são:

- `tabela` (entrada) — a tabela de dispersão
- `tamTabela` (entrada) — tamanho da tabela de dispersão
- `chave` (entrada) — a chave de busca
- `fDispersao` (entrada) — ponteiro para a função de dispersão a ser usada

Essa função retorna o índice do registro no arquivo de registros (i.e., o valor da chave), se a chave for encontrada, ou `-1`, em caso contrário.

```
int BuscaDE(tTabelaDE tabela, int tamTabela, tCEP chave, tFDispersao fDispersao)
{
   int posColetor;

      /* Encontra a respectiva lista encadeada */
   posColetor = fDispersao(chave)%tamTabela;

      /* A lista deve ter sido encontrada */
   ASSEGURA( posColetor >= 0 && posColetor < tamTabela,
             "ERRO: Impossivel localizar lista" );

      /* A função BuscaListaSE() completa o serviço */
   return BuscaListaSE(tabela + posColetor, chave);
}
```

A função `BuscaDE()` chama `BuscaListaSE()` para efetuar uma busca na lista encadeada na qual a chave pode ser encontrada. Essa última função foi discutida no **Capítulo 10** do **Volume 1**.

Inserção

A função `InsereDE()` insere uma nova chave numa tabela de dispersão com encadeamento. Os parâmetros dessa função são:

- `tabela` (entrada) — a tabela de dispersão
- `tamTabela` (entrada) — tamanho da tabela de dispersão (i.e., o número de coletores)
- `conteudo` (entrada) — a chave de busca e seu respectivo índice
- `fDispersao` (entrada) — ponteiro para a função de dispersão a ser usada

```
void InsereDE( tTabelaDE tabela, int tamTabela,
               const tCEP_Ind *conteudo, tFDispersao fDispersao )
{
   int posColetor;

      /* Encontra a lista encadeada que deverá conter a chave */
   posColetor = fDispersao(conteudo->chave)%tamTabela;

      /* Garante que o índice é válido */
   ASSEGURA( posColetor >= 0 && posColetor < tamTabela,
             "ERRO: Impossivel localizar um lista" );

      /* A função InsereListaSE() completa o serviço */
   return InsereListaSE(&tabela[posColetor], conteudo );
}
```

A função `InsereDE()` chama `InsereListaSE()` para inserir um novo nó na devida lista encadeada da tabela. Essa última função foi discutida no **Capítulo 10** do **Volume 1**.

Note que a função `InsereDE()` não verifica se a chave que será inserida já existe na tabela, de maneira que ela permite chaves duplicadas.

Remoção

A função `RemoveDE()`, apresentada abaixo, remove uma chave de uma tabela de dispersão com encadeamento e seus parâmetros são:

- `tabela` (entrada) — a tabela de dispersão
- `tamTabela` (entrada) — tamanho da tabela de dispersão (i.e., número de coletores)
- `chave` (entrada) — a chave de busca
- `fDispersao` (entrada) — ponteiro para a função de dispersão a ser usada

Essa função retorna `1`, se a remoção foi bem-sucedida, ou `0`, em caso contrário.

```
int RemoveDE(tTabelaDE tabela, int tamTabela, tCEP chave, tFDispersao fDispersao)
{
   int posColetor;

      /* Encontra a lista encadeada que contém a chave */
   posColetor = fDispersao(chave)%tamTabela;

      /* Garante que o índice é válido */
   ASSEGURA( posColetor > 0 && posColetor < tamTabela,
             "ERRO: Impossivel localizar lista" );

      /* A função RemoveListaSE() completa o serviço */
   return RemoveListaSE(tabela +posColetor, chave);
}
```

A função `RemoveDE()` chama `RemoveListaSE()` para remover um nó na devida lista encadeada da tabela. Essa última função foi discutida no **Capítulo 10** do **Volume 1**.

7.3.3 Análise

Teorema 7.1: Uma busca bem-sucedida numa tabela de dispersão com encadeamento com fator de carga α requer que cerca de $1 + \alpha/2$ nós sejam visitados.

Prova: A lista que está sendo examinada contém o nó que armazena a chave procurada mais zero ou mais nós adicionais. O número esperado desses *nós adicionais* numa tabela com n chaves e m listas é $(n - 1)/m = \alpha - 1/m$, que é essencialmente α, uma vez que se presume que o valor de m é bem grande. Em média, metade desses *nós adicionais* são examinados. Assim, combinando-se esses nós com aquele que contém a chave, obtém-se que cerca de $1 + \alpha/2$ nós são visitados. ■

O **Teorema 7.1** mostra que o tamanho da tabela não é realmente importante, mas o fator de carga é deveras relevante. Além disso, esse resultado é válido quer as listas estejam ordenadas ou não.

Teorema 7.2: Numa busca malsucedida numa tabela de dispersão com encadeamento com fator de carga α, se as listas forem desordenadas, no pior caso, o número de nós visitados é $1 + \alpha$.

Prova: Nesse caso, todas as chaves devem ser acessadas, de modo que o número de nós visitados é $1 + \alpha$. ■

Teorema 7.3: Numa busca malsucedida numa tabela de dispersão com encadeamento com fator de carga α, se as listas forem ordenadas, no pior caso, o número de nós visitados é $1 + \alpha/2$.

Prova: Quando as listas estão ordenadas, apenas a metade das chaves deve ser examinada, de modo que, numa busca malsucedida, o custo temporal é o mesmo que aquele para uma busca bem-sucedida (v. **Teorema 7.1**). ■

Corolário 7.1: O número médio de nós visitados durante uma inserção numa tabela de dispersão com encadeamento é $1 + \alpha/2$ se as listas forem ordenadas.

Prova: Se as listas forem ordenadas, assim como ocorre com uma busca malsucedida, o número de nós visitados é, em média, $1 + \alpha/2$. ■

Teorema 7.4: O custo temporal de inserção numa tabela de dispersão com encadeamento é $\theta(1)$ se as listas não forem ordenadas.

Prova: Se as listas não forem ordenadas, uma inserção é sempre imediata, pois uma chave pode ser inserida no início da lista, de sorte que nenhuma comparação de chaves é necessária. ■

A inserção de chaves no início de cada lista encadeada não apenas é conveniente como também favorece localidade de referência (v. **Seção 1.5**), já que, frequentemente, chaves recentemente inseridas são mais prováveis de ser acessadas em breve.

Teorema 7.5: Em qualquer caso, o custo temporal de remoção numa tabela de dispersão com encadeamento é igual ao custo de busca pela chave a ser removida.

Prova: Quando a chave a ser removida não se encontra na tabela, a prova é trivial. Uma vez que ela tenha sido encontrada, o que resta a ser feito é alterar alguns ponteiros e liberar o nó que contém essa chave. Todas essas últimas operações têm custo $\theta(1)$. ■

Uma vantagem de dispersão com encadeamento é que a tabela de dispersão não é limitada a um número fixo de elementos, enquanto a maior desvantagem é que o desempenho das operações de busca e remoção é degradado se existirem muitas chaves em cada lista; i.e., se houver excesso de colisões.

Em dispersão com encadeamento, é típico usar um fator de carga igual a $1{,}0$ (i.e., o número de chaves é igual ao tamanho da tabela). Fatores de carga menores não melhoram o desempenho de modo significativo, mas o custo temporal de todas as operações aumenta linearmente com o fator de carga, de maneira que usar um fator de carga maior do que 2 não é geralmente uma boa ideia.

7.4 Dispersão com Endereçamento Aberto

7.4.1 Conceitos

Numa tabela de dispersão com **endereçamento aberto** (ou com **sondagem**), todas as chaves residem no próprio array com o qual a tabela é implementada; i.e., os coletores são elementos desse array. Como todas as chaves residem nesse array, o fator de carga é sempre menor do que ou igual a 1. Ou seja, uma tabela de dispersão com endereçamento aberto não pode conter mais chaves do que o número de elementos alocados para si, como ocorre com tabela de dispersão com encadeamento. Geralmente, o fator de carga deve ser mantido abaixo de $\alpha = 0{,}5$ para uma tabela de dispersão com endereçamento aberto.

Resolução de colisões em dispersão com endereçamento aberto é efetuada examinando-se posições da tabela a partir da posição na qual ocorre a colisão até que se encontre a chave de busca, uma posição vazia ou todas as possíveis posições tenham sido examinadas. Nesses dois últimos casos, conclui-se que a chave não está presente na tabela. Esse processo de resolução de colisões é denominado **sondagem**. As posições de uma tabela de dispersão acessadas durante uma operação de busca, inserção ou remoção formam uma **sequência de sondagem** e a distância entre duas posições consecutivas nessa sequência é denominada **passo de sondagem**.

Para sondar as posições da tabela depois que ocorre uma colisão, utiliza-se uma função de dispersão que depende não apenas da chave de busca, como também de um **índice de sondagem** (v. adiante).

Para ser possível determinar se um coletor da tabela de dispersão está vazio, pode-se iniciar cada elemento da tabela de modo que ele contenha um valor especial. Esse valor deve ser sintaticamente válido, mas semanticamente inválido. Por exemplo, se todas as chaves são valores inteiros não negativos, pode-se usar -1 como valor de chave de cada coletor disponível. Assim para verificar se um coletor está livre, compara-se o valor de sua chave com -1.

Para resolução de colisões em dispersão com endereçamento aberto, são empregadas duas funções de dispersão. A primeira função de dispersão, denominada **função de dispersão primária**, é aplicada numa chave para obtenção de seu valor de dispersão. Ou seja, uma função de dispersão primária é simplesmente uma função de dispersão comum (como aquelas usadas em dispersão com encadeamento, por exemplo). Quando a aplicação de uma função primária resulta em colisão, o valor de dispersão que ela produz é usado como entrada para uma outra função chamada **função de dispersão secundária** (ou **função de sondagem**). Além de usar esse valor como entrada, essa segunda função também usa um índice de sondagem, que é um valor inteiro $i > 0$.

7.4.2 Resoluções de Colisões (Sondagens)

Sondagem Linear

Em **sondagem linear**, examinam-se posições sucessivas circularmente na tabela usando-se uma função de dispersão, que é definida como:

$$f(c, i) = (f'(c) + i) \bmod m$$

em que i é o índice de sondagem e é tal que $0 \leq i < m$ e f' é uma função de dispersão primária e f é uma função de dispersão secundária. Por exemplo, usando-se $f'(c) = c \bmod m$ e $m = 1000$, os possíveis valores de dispersão da chave $c = 2998$ serão:

$$f(2998, 0) = (2998 \bmod 1000 + 0) \bmod 1000 = 998 \qquad \textit{[Primeira sondagem]}$$

$$f(2998, 1) = (2998 \bmod 1000 + 1) \bmod 1000 = 999 \qquad \textit{[Segunda sondagem]}$$

$$f(2998, 2) = (2998 \bmod 1000 + 2) \bmod 1000 = 0 \qquad \textit{[Terceira sondagem]}$$

$$\vdots \qquad \textit{[etc]}$$

Esse resultado significa que a chave $c = 2998$ será primeiro procurada na posição *998*; se ela não for encontrada nessa posição e a posição não estiver vazia, ela será procurada na posição *999*, depois na posição *0* e assim por diante.

A **Figura 7–8** mostra um exemplo de operação de busca malsucedida numa tabela de dispersão que usa sondagem linear para resolver colisões. Nesse exemplo, a função (primária) de dispersão é $f'(c) = c \bmod 13$ e a chave de busca é *17*.

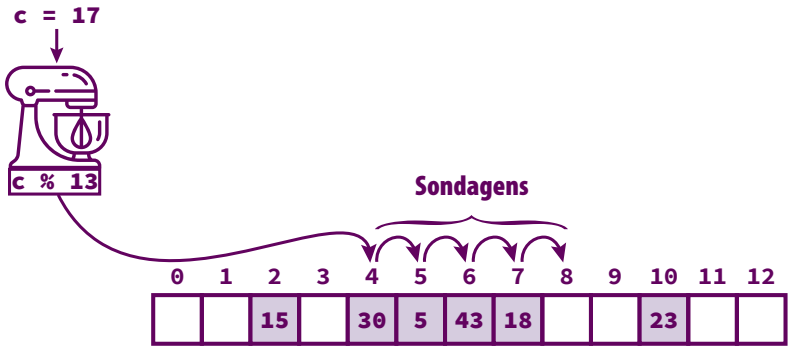

FIGURA 7–8: BUSCA EM TABELA DE DISPERSÃO COM SONDAGEM LINEAR

Sondagens são aplicadas em qualquer operação de busca, inserção ou remoção em tabelas de dispersão com endereçamento aberto. Considerando que a chave é primária, em qualquer uma dessas operações, uma sequência de sondagens para quando a chave procurada é encontrada ou quando o índice de sondagem i é igual a m, que é o tamanho da tabela. No caso de inserção, uma sequência de sondagem também para quando se encontra um coletor vazio no qual será efetuada a inserção.

Supondo que uma tabela de dispersão seja suficientemente grande, um espaço livre para inserção pode sempre ser encontrado, mas o tempo gasto em sondagem pode ser muito elevado. Ainda pior, mesmo se a tabela estiver pouco ocupada, sequências de coletores ocupados se formarão. Esse efeito, conhecido como **agrupamento primário**, significa que qualquer chave que esteja alojada num desses agrupamentos irá requerer uma longa sequência de sondagem antes que ela seja acrescentada ao mesmo agrupamento.

Agrupamento primário faz com que chaves tornem-se desigualmente distribuídas na tabela de dispersão, com muitas chaves se agrupando em torno de uma única posição de dispersão. Em geral, quanto maior for o fator de carga, mais agrupamentos ocorrerão. Entretanto agrupamentos podem ser formados mesmo quando o fator de carga não é grande. Partes da tabela de dispersão podem consistir de grandes agrupamentos, enquanto outras são vagamente habitadas. Agrupamentos prejudicam o desempenho, pois eles acarretam num número excessivo de sondagens.

A **Figura 7–9** mostra exemplos de inserção numa tabela de busca com sondagem linear. Nesse exemplo, a função de dispersão é: $f(c, i) = ((c \bmod 11) + i) \bmod 11$.

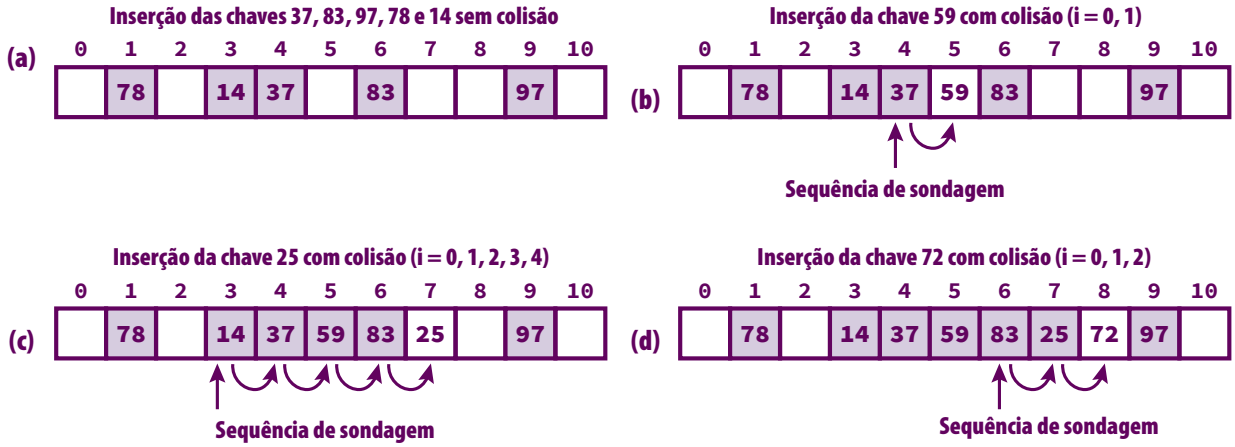

Figura 7–9: Inserções em Tabela de Dispersão com Sondagem Linear

A principal vantagem do método de sondagem linear é sua simplicidade e o fato de não haver restrição com relação ao tamanho da tabela para assegurar que todas as chaves serão eventualmente examinadas.

Sondagem Quadrática

Numa sondagem linear, se um valor da primeira sondagem for x, as sondagens subsequentes seriam $x + 1$, $x + 2$, $x + 3$ e assim por diante. Ou seja, o passo de sondagem é constante. Existe outra abordagem de resolução de colisões utilizada com endereçamento aberto, conhecida como **sondagem quadrática**. Usando-se sondagem quadrática, as sondagens nesse caso seriam $x + 1$, $x + 4$, $x + 9$, $x + 16$, $x + 25$ e assim por diante. Numa sondagem quadrática, a função de resolução de colisão é quadrática e comumente é escolhida como:

$$f(c, i) = (f'(c) + i^2) \bmod m$$

A **Figura 7–10** apresenta exemplos de inserção numa tabela de dispersão com encadeamento aberto que usa sondagem quadrática para resolver colisões. Nesses exemplos, a função de dispersão é: $f(c, i) = ((c \bmod 11) + i^2) \bmod 11$. As chaves usadas nesse exemplo são as mesmas usadas no exemplo de sondagem linear na **Figura 7–9**. Além disso, essas chaves são inseridas na mesma ordem nas duas figuras.

Sondagem quadrática é uma tentativa de impedir a formação de agrupamentos. A ideia é sondar coletores mais largamente separados, em vez daquelas adjacentes à posição inicial de dispersão na qual ocorre colisão. De fato, sondagens quadráticas reduzem a ocorrência de agrupamentos, mas não examinam necessariamente cada coletor da tabela.

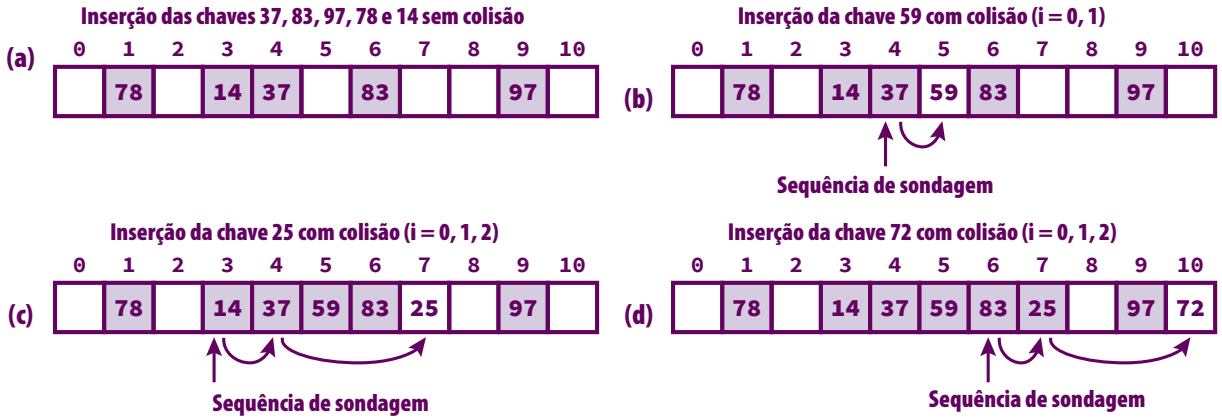

FIGURA 7–10: INSERÇÕES EM TABELA DE DISPERSÃO COM SONDAGEM QUADRÁTICA

Se o tamanho da tabela não for um número primo, o número de posições alternativas pode ser severamente reduzido. Em particular, se m for uma potência de 2, relativamente poucos coletores da tabela são examinados. Por exemplo, se o tamanho da tabela for 16, as únicas posições alternativas estarão a distâncias 1, 4 ou 9 entre si. Quando o tamanho da tabela é um número primo, é mais provável que uma sequência de sondagem eventualmente examine cada coletor.

Sondagens quadráticas eliminam o problema de agrupamento primário. Contudo elas sofrem de um problema de agrupamento diferente e mais sutil denominado **agrupamento secundário**. Isso ocorre porque todas as chaves cujos valores de dispersão resultam num determinado índice seguem a mesma sequência de sondagem. Agrupamentos secundários constituem uma deficiência amena, pois resultados experimentais sugerem que eles causam menos do que meia sondagem adicional para cada operação de busca.

Quando se usa sondagem linear, não é recomendável deixar que a tabela de dispersão se torne quase completa porque a eficiência das operações de busca, inserção e remoção pode ser severamente prejudicada. Usando-se sondagem quadrática, a situação é ainda mais drástica, pois não há garantia que se encontre um espaço vazio quando mais da metade da tabela estiver completa ou mesmo antes disso, se o tamanho da tabela não for um número primo. Mas pode-se mostrar que, quando se usa sondagem quadrática e o tamanho da tabela é um número primo, uma nova chave pode sempre ser inserida se pelo menos metade da tabela estiver vazia.

Sondagem com Dispersão Dupla

Para evitar agrupamentos primários ou secundários, pode-se usar a abordagem de **dispersão dupla**. Sondagens com dispersão dupla utilizam duas funções primárias de dispersão, f_1 e f_2, na composição da função de sondagem, que é definida como:

$$f(c, i) = (f_1(c) + i \cdot f_2(c)) \bmod m$$

em que $0 \leq i < m$, e $f_1(c)$ e $f_2(c)$ são funções de dispersão comuns.

Em dispersão dupla, o tamanho do passo de sondagem depende do valor da chave, e é obtido por meio da função f_2. Por exemplo, se os valores resultantes das aplicações de f_1 e f_2 forem, respectivamente, x e p, então os passos de sondagem serão x, $x + p$, $x + 2p$, $x + 3p$, $x + 4p$ e assim por diante, sendo que p depende da chave, mas permanece constante durante uma sondagem.

Tipicamente, usa-se $f_1(c) = c \bmod m$ e $f_2(c) = 1 + c \bmod m'$, sendo que m' é ligeiramente menor do que m (p. ex., $m' = m - 1$ ou $m' = m - 2$). Mas existem abordagens alternativas para escolha das funções f_1 e f_2, sendo que duas dessas abordagens são:

[1] m é uma potência de 2 e f_2 sempre retorna um valor ímpar

[2] m é um número primo e f_2 sempre retorna um valor positivo menor do que m

Além de satisfazer esses requisitos, a função f_2 não deve ser igual a f_1 nem deve resultar em zero.

A **Figura 7–11** mostra um exemplo de dispersão dupla com função de dispersão dada por:

$$f(c, i) = (c \bmod 11 + i \cdot (1 + c \bmod 9)) \bmod 11$$

Diferentemente do que ocorre com sondagem linear e quadrática, sequências de sondagem em tabelas com endereçamento aberto com dispersão dupla dependem das chaves, de modo que chaves que colidem podem ter diferentes sequências de sondagem. Esse fato minimiza os efeitos de agrupamentos.

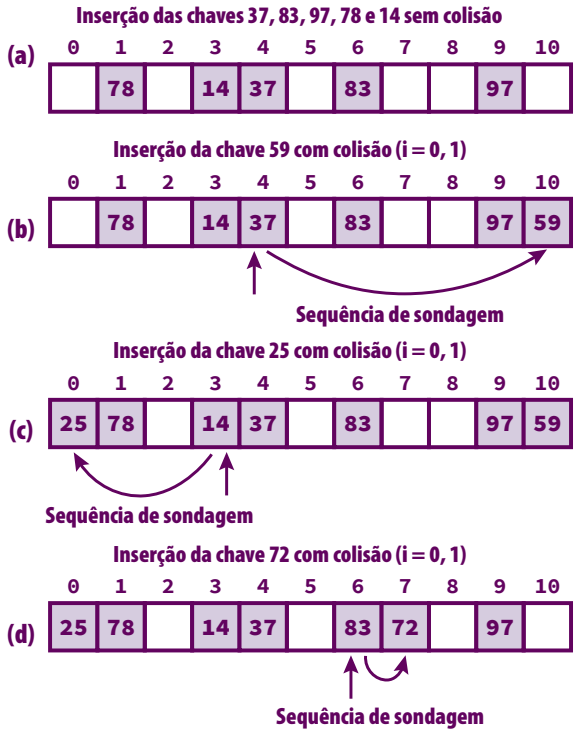

Figura 7–11: Inserções em Tabela de Dispersão com Dispersão Dupla

7.4.3 Operações Básicas

Busca

Uma operação de busca numa tabela de dispersão com endereçamento aberto segue o algoritmo delineado na **Figura 7–12**.

Algoritmo BuscaEmDispersãoComEndereçamentoAberto

Entrada: Uma tabela de dispersão com endereçamento aberto de tamanho m e uma chave de busca c

Saída: O valor associado à chave, se ela for encontrada, ou um valor informando o fracasso da operação

1. Para cada índice i de uma tabela de dispersão com endereçamento aberto, aplique a função de dispersão $f(c, i)$, obtendo, assim, a posição p em que a chave pode ser encontrada
   - **1.1** Se o espaço na posição p da tabela estiver vazio, encerre informando o fracasso da busca
   - **1.2** Se a chave for encontrada na posição p, retorne o valor associado à chave

Figura 7–12: Algoritmo de Busca em Dispersão com Endereçamento Aberto

Inserção

Uma operação de inserção numa tabela de dispersão com endereçamento aberto segue o algoritmo apresentado na **Figura 7–13**.

ALGORITMO INSEREEMDISPERSÃOCOMENDEREÇAMENTOABERTO

ENTRADA: Conteúdo de um novo elemento cuja chave é *c*

ENTRADA/SAÍDA: Tabela de dispersão com endereçamento aberto de tamanho *m*

SAÍDA: Um valor informando o sucesso da operação

1. Para cada índice *i* de uma tabela de dispersão com endereçamento aberto de tamanho *m*, aplique a função *f(c, i)*, obtendo, assim, a posição *p* em que a chave deve ser inserida.
   - **1.1** Se o espaço na posição *p* da tabela estiver vazio ou marcado como removido, armazene a chave e seu valor associado nessa posição
   - **1.2** Se a chave for encontrada na posição *p* e ela for considerada primária, retorne informando o fracasso da operação
2. Se o final da tabela for atingido sem que se tenha encontrado espaço para inserção, retorne informando o fracasso da operação

FIGURA 7–13: ALGORITMO DE INSERÇÃO EM DISPERSÃO COM ENDEREÇAMENTO ABERTO

Remoção

Uma remoção comum não pode ser executada numa tabela de dispersão com sondagem, porque o coletor que contém a chave removida pode fazer parte de uma sequência de sondagem. Por exemplo, se a chave *83* na **Figura 7–9 (d)** for removida (i.e., declarada como vazia), uma operação subsequente de busca não encontrará mais a chave *72*.

Numa operação de remoção, deve-se levar em consideração que a chave a ser removida pode ser necessária para localizar outra chave na tabela. Uma solução simples para amenizar essa dificuldade é utilizar um campo do elemento armazenado na tabela para marcá-lo como removido, de modo que uma sequência de sondagem não seja quebrada. Inserções subsequentes podem ocupar os lugares de chaves marcadas como removidas. A desvantagem desse método é que se forem acumuladas muitas chaves marcadas como removidas, a busca pode tornar-se demasiadamente lenta. Nesse caso, a melhor solução é reconstruir a tabela de dispersão, como mostra a **Seção 7.5**.

Uma operação de remoção numa tabela de dispersão com endereçamento aberto segue o algoritmo da **Figura 7–14**.

ALGORITMO REMOVEEMDISPERSÃOCOMENDEREÇAMENTOABERTO

ENTRADA: A chave do elemento a ser removido

ENTRADA/SAÍDA: Uma tabela de dispersão com endereçamento aberto

SAÍDA: Um valor informando o sucesso ou fracasso da operação

1. Efetue uma busca pela chave a ser removida de acordo com o algoritmo de busca apresentado na **Figura 7–12**
2. Se a chave for encontrada, marque como removido o coletor que ela ocupa e retorne informando o sucesso da operação
3. Se a chave não for encontrada, retorne informando o fracasso da operação

FIGURA 7–14: ALGORITMO DE REMOÇÃO EM DISPERSÃO COM ENDEREÇAMENTO ABERTO

7.4.4 Implementação

Aqui, será apresentada uma implementação de tabela de dispersão com endereçamento aberto que usa ponteiros para funções para permitir o uso de qualquer uma das abordagens de resolução de colisão discutidas acima.

Definições de Tipos

Os seguintes tipos são utilizados na criação de uma tabela de dispersão com endereçamento aberto, além dos tipos `tCEP` e `tCEP_Ind` definidos na **Seção 7.3.2**:

```
   /* Tipo usado para indicar o status de um elemento da tabela */
typedef enum {VAZIO, OCUPADO, REMOVIDO} tStatusDEA;

   /* Tipo dos elementos da tabela de busca */
typedef struct {
          tCEP_Ind   chaveEIndice;
          tStatusDEA  status;
       } tColetorDEA, *tTabelaDEA;
```

Quando uma chave é removida de uma tabela de dispersão com endereçamento aberto, não se pode simplesmente marcar seu coletor com `VAZIO`, pois isso poderá interromper prematuramente uma sequência de sondagem. Nesse caso, o tipo `tStatusDEA` usa uma constante, denominada `REMOVIDO`, para indicar tal situação. Assim, o fato de o status de um coletor ser `REMOVIDO`, indica que ele está livre neste momento, mas já esteve ocupado antes.

As funções de dispersão que serão utilizadas em dispersão com endereçamento aberto possuem um parâmetro a mais do que aquelas usadas em dispersão com encadeamento. Esse parâmetro corresponde ao índice de sondagem. Portanto o tipo de ponteiro para tais funções usado nesta implementação é definido como:

```
   /* Tipo de um ponteiro para uma função de dispersão com endereçamento aberto */
typedef unsigned (*tFDispersaoDEA) (tCEP, unsigned, unsigned);
```

Iniciação da Tabela

A função `CriaTabelaDEA()` a seguir é usada para alocar e iniciar os elementos de uma tabela a ser usada com dispersão com endereçamento aberto. O único parâmetro dessa função é o número de posições da tabela de dispersão e ela retorna o endereço da tabela de dispersão criada.

```
tTabelaDEA CriaTabelaDEA(int nElementos)
{
   tTabelaDEA  tabela;
   int         i;

   tabela = calloc(nElementos, sizeof(tColetorDEA));
   ASSEGURA( tabela, "Impossivel alocar a tabela de dispersao" );

   for (i = 0; i < nElementos; ++i)
      tabela[i].status = VAZIO;/* Todos os elementos estão inicialmente desocupados */

   return tabela;
}
```

Busca

A função `BuscaDEA()` apresentada a seguir executa uma operação de busca numa tabela de dispersão com endereçamento aberto. Os parâmetros dessa função são:

- `tabela` (entrada) — a tabela de dispersão
- `tamanhoTab` (entrada) — tamanho da tabela de dispersão
- `chave` (entrada) — a chave de busca
- `fDispersao` (entrada) — ponteiro para a função de dispersão a ser usada

O retorno dessa função é o índice do registro no arquivo de registros, se a chave for encontrada, ou `-1`, em caso contrário.

```
int BuscaDEA( tTabelaDEA tabela, int tamanhoTab,
              tCEP chave, tFDispersaoDEA fDispersao )
{
   int i, pos;

      /* Faz sucessivas sondagens até encontrar a chave, */
      /* uma posição vazia ou a última sondagem          */
   for (i = 0; i < tamanhoTab; ++i) {
      pos = fDispersao(chave, i, tamanhoTab);
      ASSEGURA(pos < tamanhoTab, "Dispersao invalida recebido por BuscaDEA()");

      if (tabela[pos].status == VAZIO)
         return -1; /* A chave não foi encontrada */

      if (tabela[pos].status == OCUPADO &&
          !memcmp( tabela[pos].chaveEIndice.chave, chave, sizeof(tCEP) ))
         return tabela[pos].chaveEIndice.indice; /* Chave encontrada */
   }
   return -1;  /* A chave não foi encontrada */
}
```

Note que a função `BuscaDEA()` pode ser usada com sondagem linear, quadrática ou com dispersão dupla. O que muda nesses tipos de sondagens é a função passada como último parâmetro para a função `BuscaDEA()`.

Inserção

A função `InsereDEA()` apresentada a seguir faz inserção numa tabela de dispersão com endereçamento aberto. Os parâmetros dela são:

- `tabela` (entrada) — a tabela de dispersão
- `tamanhoTab` (entrada) — tamanho da tabela de dispersão
- `chaveEIndice` (entrada) — a chave de busca e seu respectivo valor
- `fDispersao` (entrada) — ponteiro para a função de dispersão a ser usada

O retorno dessa função é `1`, se houver inserção, ou `0`, em caso contrário.

```
int InsereDEA( tTabelaDEA tabela, int tamanhoTab,
               tCEP_Ind chaveEIndice, tFDispersaoDEA fDispersao )
{
   int i, pos, posInsercao = -1;

      /* Faz sucessivas sondagens até encontrar a chave */
      /* ou um local onde se pode fazer a inserção.     */
   for (i = 0; i < tamanhoTab; ++i) {
      pos = fDispersao(chaveEIndice.chave, i, tamanhoTab);
      ASSEGURA( pos < tamanhoTab, "Valor invalido recebido por InsereDEA()" );

      if ( tabela[pos].status == REMOVIDO && posInsercao < 0 )
            /* Se a chave não for encontrada numa nova   */
            /* sondagem, esta será a posição de inserção */
         posInsercao = pos;
```

```
      if (tabela[pos].status == VAZIO && posInsercao < 0) {
         posInsercao = pos; /* A chave não foi encontrada */
         break; /* A chave será inserida fora do laço */
      }

      if ( tabela[pos].status == OCUPADO &&
           !memcmp(tabela[pos].chaveEIndice.chave, chaveEIndice.chave, sizeof(tCEP)))
         return 0; /* Chave encontrada */
   }

      /* Insere chave e índice na posição determinada dentro do laço for */
   tabela[posInsercao].chaveEIndice = chaveEIndice;
   tabela[posInsercao].status = OCUPADO;

   return 1;
}
```

A função `InsereDEA()` pode ser usada com sondagem linear, quadrática ou com dispersão dupla [v. discussão ao final da apresentação da função `BuscaDEA()`].

Remoção

A função `RemoveDEA()` apresentada abaixo remove uma chave de uma tabela de dispersão com endereçamento aberto. A mesma função pode ser usada com sondagem linear, quadrática ou com dispersão dupla [v. discussão ao final da apresentação da função `BuscaDEA()`]. Os parâmetros da função `RemoveDEA()` são:

- `tabela` (entrada) — a tabela de dispersão
- `tamanhoTab` (entrada) — tamanho da tabela de dispersão
- `chave` (entrada) — a chave de busca
- `fDispersao` (entrada) — ponteiro para a função de dispersão a ser usada

O retorno dessa função é `1`, se houver remoção, ou `0`, em caso contrário.

```
int RemoveDEA(tTabelaDEA tabela, int tamanhoTab, tCEP chave, tFDispersaoDEA fDispersao)
{
   int i, pos;

      /* Faz sucessivas sondagens até encontrar a chave, */
      /* uma posição vazia ou atingir a última sondagem. */
   for (i = 0; i < tamanhoTab; ++i) {
      pos = fDispersao(chave, i, tamanhoTab);
      ASSEGURA(pos < tamanhoTab, "Valor dispersao invalido em RemoveDEA()");

      if (tabela[pos].status == VAZIO)
         return 0; /* A chave não foi encontrada */

      if (tabela[pos].status == OCUPADO &&
          !memcmp( tabela[pos].chaveEIndice.chave, chave,sizeof(tCEP))) {
         tabela[pos].status = REMOVIDO; /* Marca como removido */

         return 1; /* Remoção foi OK */
      }
   }

   return 0;  /* A chave não foi encontrada */
}
```

7.4.5 Análise

As demonstrações das várias afirmações que serão apresentadas nesta seção não são triviais e podem ser encontradas no livro de Knuth (Volume 3, 1997 — v. **Bibliografia**).

O custo temporal de acesso a uma chave depende dos tamanhos das sondagens requeridas. Cada coletor acessado durante uma sondagem acrescenta uma unidade ao custo temporal de busca por um coletor vago (para inserção) ou por um coletor ocupado. Durante um acesso, um coletor deve ser testado para verificar se ele está vazio e, no caso de busca ou remoção, se ele contém a chave desejada. Assim o custo temporal de uma operação de busca, inserção ou remoção é proporcional ao tamanho da sondagem. Como o tamanho médio de sondagem (e, portanto, o custo temporal médio de acesso) depende do fator de carga, à medida que o fator de carga aumenta, os tamanhos de sondagem tornam-se maiores.

Em geral, buscas malsucedidas levam mais tempo do que buscas bem-sucedidas. Durante uma sequência de sondagem, o algoritmo pode parar assim que ele encontra a chave desejada, que é, em média, metade da sequência de sondagem. Por outro lado, ele deve ir até o fim de tal sequência antes de estar seguro que não pode encontrar uma chave.

Custos Temporais em Sondagem Linear

Pode ser mostrado que, quando se usa sondagem linear, o número esperado de sondagens (i.e., o tamanho de sondagem T_s) em inserções e buscas malsucedidas em função do fator de carga (α) é aproximadamente:

$$T_s = (1 + 1 / (1 - \alpha))/2$$

Para buscas bem-sucedidas, o resultado equivalente é:

$$T_s = (1 + 1 / (1 - \alpha)^2)/2$$

Em média, buscas bem-sucedidas devem levar menos tempo do que buscas malsucedidas. Assumindo-se que ocorre dispersão uniforme (i.e., quando não ocorrem agrupamentos), o número esperado de sondagens quando uma busca é malsucedida é:

$$1/(1 - \alpha)$$

em que α é o fator de carga da tabela de dispersão. Por exemplo, o número de posições que se espera sondar numa tabela de dispersão com metade de suas posições ocupadas (i.e., com $\alpha = 0,5$) é $1/(1 - 0,5) = 2$.

Por outro lado, o número médio de sondagens numa busca bem-sucedida é dado por:

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

Esses resultados são melhores do que os resultados correspondentes quando ocorrem agrupamentos. Por exemplo, se $\alpha = 0,75$, então $8,5$ sondagens são esperadas numa inserção com sondagem linear. Por outro lado, se $\alpha = 0,90$, 50 sondagens são esperadas numa inserção com sondagem linear, o que é ruim em termos de eficiência. Caso não ocorram agrupamentos, os resultados serão, respectivamente, 4 e 10 sondagens. A conclusão que se pode tirar dessas fórmulas é que sondagem linear não é uma boa abordagem quando $\alpha > 0,5$.

Com um fator de carga de $0,5$, uma busca bem-sucedida consome $1,5$ comparações e uma busca malsucedida consome $2,5$ comparações. Com um fator de carga igual a $2/3$, esses valores são, respectivamente, $2,0$ e $5,0$. Com fatores de carga maiores, esses valores também serão bem maiores.

Custos Temporais em Sondagens Quadrática e com Dispersão Dupla

Sondagens quadrática e com dispersão dupla compartilham seus desempenhos, que indicam uma modesta superioridade sobre sondagem linear. Para uma busca bem-sucedida, o número de sondagens é dado por:

$$[-\log_2 (1 - \alpha)]/\alpha$$

Por outro lado, para uma busca malsucedida, o número de sondagens é expresso como:

$$1/(1 - \alpha)$$

Com um fator de carga de $0,5$, buscas bem-sucedidas e malsucedidas requerem uma média de duas sondagens. Usando um fator de carga igual a $0,8$, os respectivos números de sondagens são $2,9$ e 5. Assim fatores de carga

maiores do que aqueles recomendados para sondagem linear podem ser tolerados com sondagem quadrática e com dispersão dupla.

Custo Espacial

Como foi visto, o fator de carga deve ser mantido preferencialmente abaixo de *0,5*. Por outro lado, quanto menor for o fator de carga, maior será a memória necessária para armazenar uma certa quantidade de chaves. O fator de carga ótimo numa situação particular depende de uma disputa entre gasto de memória, que diminui com menores fatores de carga α, e rapidez, que aumenta com maiores valores de α.

Comparando Métodos de Sondagem

Resumindo o que foi exposto nesta seção, as seguintes recomendações devem ser seguidas quando se lida com tabela de dispersão com endereçamento aberto:

- O fator de carga máximo em dispersão com endereçamento aberto deve ser em torno de *0,5*. Para dispersão dupla com esse fator de carga, as buscas terão um tamanho de sondagem médio igual a *2*.
- O custo temporal de uma operação de busca tende a infinito à medida que o fator de carga da tabela na qual a busca é efetuada se aproxima de *1,0* em dispersão com endereçamento aberto.

Normalmente, dispersão dupla é a melhor opção para dispersão com endereçamento aberto. A principal vantagem de dispersão dupla é que ela produz uma boa distribuição de chaves por toda uma tabela de dispersão. Como desvantagem tem-se que o tamanho da tabela sofre restrições de modo a assegurar que todas as posições na tabela sejam visitadas numa sequência de sondagens antes que algum coletor seja visitado duas vezes.

Quando há abundância de memória, é mais simples usar uma tabela de dispersão maior com sondagem linear do que usar dispersão dupla. A justificativa é que, uma vez que a tabela tenha cerca de *50%* de ocupação, não há muito benefício que se possa obter usando dispersão dupla.

A **Tabela 7–1** apresenta uma comparação dos métodos de resolução de colisão usados em dispersão com endereçamento aberto.

Sondagem Linear	Sondagem Quadrática	Sondagem com Dispersão Dupla
Melhor localidade de referência	Fácil de implementar	Uso mais eficiente de espaço
Usa poucas sondagens	Não explora todas as posições da tabela	Implementação é mais complicada
Acarreta agrupamento primário	Acarreta agrupamento secundário	Usa menos sondagens, mas leva mais tempo
Passo de sondagem é fixo	Passo de sondagem aumenta de acordo com o índice	Passo de sondagem varia de acordo com o valor de dispersão

Tabela 7–1: Dispersão com Endereçamento Aberto: Comparação de Métodos

7.5 Redimensionamento de Dispersão

7.5.1 Conceito

Quando o fator de carga excede algum valor (tipicamente, *2* para dispersão com encadeamento e *0,75* para endereçamento aberto), aloca-se uma nova tabela, usualmente, dobrando o tamanho da antiga tabela, e realocam-se todas as chaves na nova tabela. Essa operação inteira é chamada **redimensionamento de dispersão**.

Uma operação de redimensionamento de dispersão é claramente uma operação muito onerosa, com custo temporal $\theta(n)$, em que n é o número de chaves no instante em que a tabela é aumentada. Mesmo assim essa operação não afeta tanto a eficiência global de uma tabela de dispersão, porque ela acontece muito raramente.

Suponha, por exemplo, que as chaves *5*, *13*, *15*, *27*, *30* e *45* sejam inseridas numa tabela de dispersão de tamanho (m) igual a *7* com sondagem linear, e a função de dispersão seja $f(c) = c \bmod 7$. A tabela de dispersão resultante é ilustrada na **Figura 7–15 (a)**. Após essas inserções, o fator de carga (α) dessa tabela será igual a *0,86* (i.e., *6/7*). Ou seja, o preenchimento dessa tabela é cerca de *86%*. Assim uma nova tabela é criada. O tamanho dessa tabela é *17*, porque esse é o primeiro número primo que é duas vezes tão grande quanto o tamanho da tabela antiga (que é *7*). A nova função de dispersão é $f'(c) = c \bmod 17$. Então as chaves *5*, *13*, *15*, *27*, *30* e *45* da antiga tabela são acessadas sequencialmente na antiga tabela e inseridas na nova tabela usando-se a função de dispersão f'. A tabela resultante dessa operação é mostrada na **Figura 7–15 (b)**.

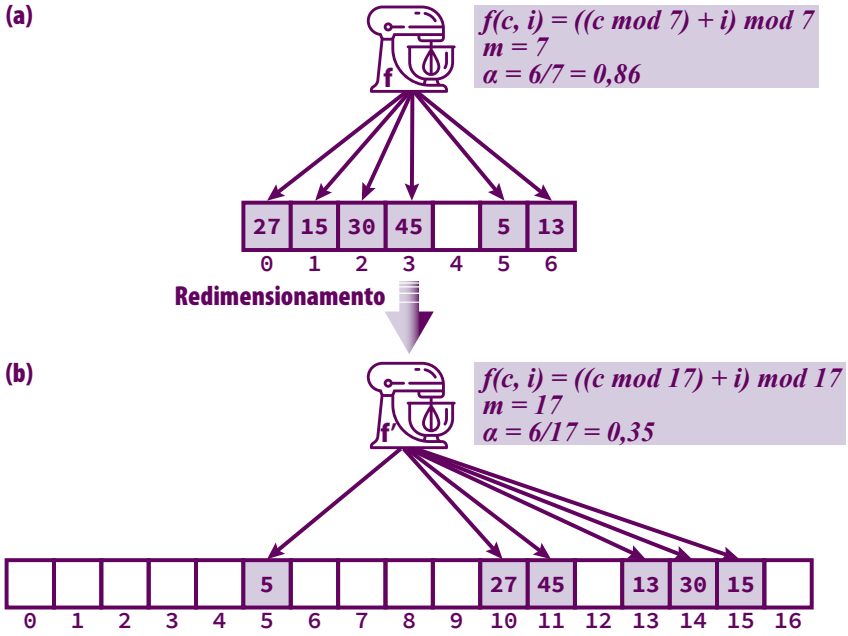

Figura 7–15: Redimensionamento de Dispersão

A decisão sobre quando redimensionar depende da abordagem usada para resolução de colisões e existem diversas alternativas. Por exemplo, com sondagem quadrática, uma alternativa é redimensionar tão logo a tabela esteja completa pela metade, enquanto uma alternativa extrema é redimensionar apenas quando uma inserção fracassar. Uma terceira abordagem é redimensionar quando o fator de carga da tabela atinge um certo valor. Como a eficiência é penalizada à medida que o fator de carga aumenta, a terceira estratégia certamente é a melhor.

7.5.2 Implementação

A função `RedimensionaTabDEA()` apresentada abaixo redimensiona uma tabela de dispersão com endereçamento aberto. Essa função retorna o endereço da nova tabela resultante do redimensionamento e seus parâmetros são:

- `tabela` (entrada) — a tabela de dispersão atual
- `tamanhoTab` (entrada) — tamanho da tabela de dispersão atual
- `fDispersao` (entrada) — ponteiro para a função de dispersão a ser usada

```
static tTabelaDEA RedimensionaTabDEA( tTabelaDEA tabela, int tamanhoTab,
                                      tFDispersaoDEA fDispersao )
{
```

```
   tTabelaDEA tabelaNova;
   int        i, tamanhoNovo = 2*tamanhoTab;

   tabelaNova = calloc(tamanhoNovo, sizeof(tColetorDEA));
   ASSEGURA(tabelaNova, "Impossivel redimensionar a tabela");

      /* Inicia os coletores */
   for (i = 0; i < tamanhoNovo; ++i)
         /* Todos os coletores da nova tabela estão inicialmente desocupados */
      tabela[i].status = VAZIO;

      /* Transfere as chaves da tabela antiga para a tabela nova */
   for (i = 0; i < tamanhoTab; ++i)       if (tabela[i].status == OCUPADO)
         ASSEGURA( InsereDEA( tabelaNova, tamanhoNovo,
                              tabela[i].chaveEIndice, fDispersao ),
                   "Erro de insercao de redimensionamento");

   free(tabela); /* Libera o espaço ocupado pela tabela antiga */

   return tabelaNova;
}
```

Note que a função `RedimensionaTabDEA()` não pode usar **realloc()**, pois, se esse fosse o caso, as chaves armazenadas na antiga tabela seriam copiadas com o mesmo valor de dispersão para a tabela nova. Observe ainda que o redimensionamento efetuado por essa função sempre dobra o tamanho atual da tabela.

7.5.3 Análise

Quando uma tabela de dispersão se torna demasiadamente ocupada, o custo temporal das operações básicas fica elevado e inserções podem fracassar para dispersão com endereçamento aberto e resolução de colisões quadrática. A solução é construir outra tabela que seja cerca de duas vezes maior e com uma nova função de dispersão associada, calcular um novo valor de dispersão para cada chave e inseri-la na nova tabela.

Se o tamanho da tabela for duplicado cada vez que um redimensionamento for necessário, pode-se mostrar, usando um raciocínio similar àquele empregado na **Seção 5.3**, que o custo temporal amortizado de uma operação de inserção numa tabela de dispersão é $\theta(1)$, mesmo quando ocorre redimensionamento. O problema é que, quando uma tabela é duplicada, seu tamanho passa a ser múltiplo de dois, o que não é recomendável.

7.6 Dispersão Cuco

7.6.1 Conceitos

No esquema de **dispersão cuco**, são usadas duas tabelas, T_1 e T_2, cada uma das quais com tamanho m, em que m é maior do que n, o número de chaves a ser armazenadas. Comumente, usa-se $m \geq 2 \cdot n$. Além disso, são usadas duas funções de dispersão: f_1, associada à tabela T_1, e f_2, associada à tabela T_2. Para qualquer chave c, existem dois locais nos quais ela pode ser armazenada, que são $T_1[f_1(c)]$ e $T_2[f_2(c)]$. A **Figura 7–16** ilustra esse tipo de dispersão, que usa um esquema de endereçamento aberto (v. **Seção 7.4**). Observe nessa figura que as chaves *22* e *77* colidem na posição *06* da tabela T_2, mas isso não constitui um problema, pois a chave *22* pode ser armazenada normalmente na posição alternativa *01* da tabela T_1.

É curioso assinalar que a denominação *dispersão cuco* é inspirada no estranho hábito de pássaros da espécie **cuco comum**, cujas fêmeas põem seus ovos em ninhos de outros pássaros para que seus filhotes sejam incubados e criados por esses pais involuntários. Quando esses filhotes nascem, eles jogam para fora do ninho os ovos ou outros filhotes que lá se encontrem. Ou seja, usando um eufemismo, o filhote bastardo *desaloja* seus irmãos (involuntários) de criação. Analogamente, inserir uma nova chave numa tabela de dispersão cuco pode **desalojar**

uma chave mais antiga para uma diferente posição na tabela. Pássaros dessa espécie habitam principalmente alguns países da Europa, Ásia e África.

Quando ocorre uma colisão numa operação de inserção em dispersão cuco, a chave que se encontra na posição de colisão é desalojada e a nova chave é inserida em seu lugar. Isso faz com que a chave desalojada vá para sua posição alternativa na outra tabela e lá seja inserida, o que pode causar a repetição desse processo se houver colisão com outra chave. Eventualmente, encontra-se um espaço vazio e encerra-se ou repete-se um desalojamento que já havia ocorrido, o que indica um ciclo sem fim. Se tal ciclo for encontrado, encerra-se o processo de inserção e redimensionam-se as tabelas de dispersão (com novas funções de dispersão, obviamente).

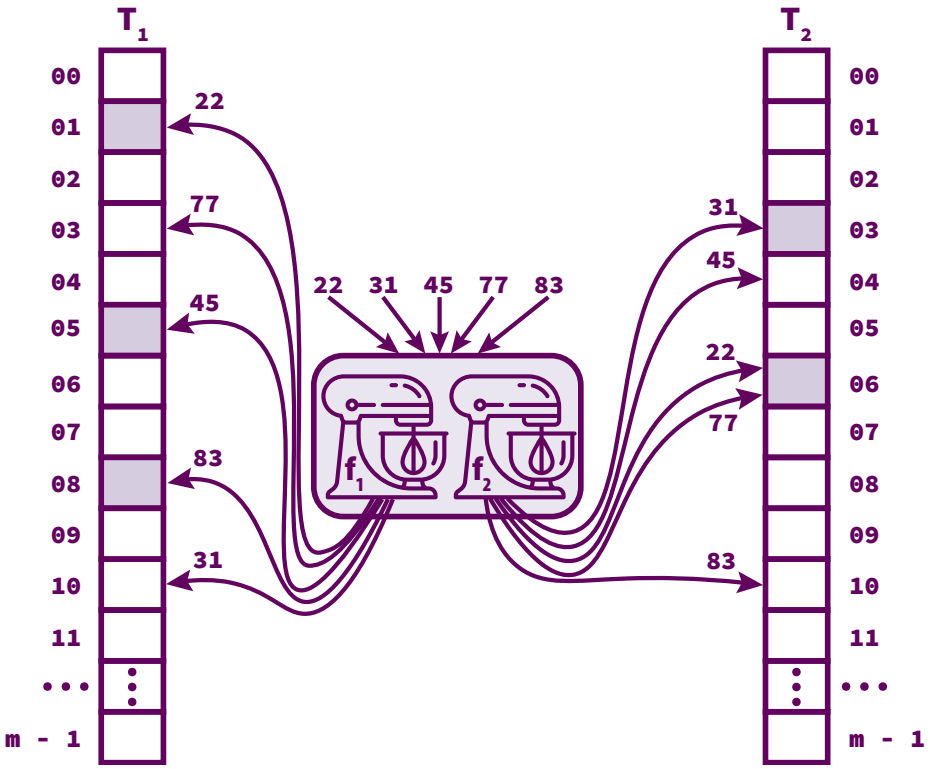

Figura 7–16: Dispersão Cuco

Mais formalmente, para inserir uma nova chave x, a primeira das duas possíveis posições é testada, i.e., $f_1(x)$ em T_1 e $f_2(x)$ em T_2. Se uma dessas posições estiver disponível, a chave é armazenada nessa posição. Caso ambas as posições já estejam ocupadas, então algumas mudanças de posição serão necessárias. Nesse caso, uma das funções de dispersão e sua tabela associada são escolhidas, como, por exemplo, a função f_1 e a tabela T_1. Suponha que y seja uma chave residente na tabela T_1 que colide com uma nova chave x a ser inserida nessa mesma tabela; isto é, essas chaves são tais que $f_1(x) = f_1(y)$. Então o processo de inserção removerá a chave y de sua posição e armazenará a nova chave x nessa posição. Em seguida, o processo tentará inserir y na tabela T_2, pois como essa chave foi removida de sua posição em T_1, ela não poderá mais ser armazenada nessa última tabela. Assim a única possibilidade para y é sua posição alternativa em T_2 na posição determinada por $f_2(y)$. Se esse espaço estiver vazio, a chave y será armazenada nele e o processo de inserção é encerrado. Se já houver uma chave ocupando esse espaço, essa chave será desalojada e a história se repete. Esse processo continua até que uma chave desalojada seja armazenada num espaço vazio. Mas existe a possibilidade de esse processo tornar-se um ciclo sem fim e, por essa razão, o número de desalojamentos deve ser limitado por uma constante. Quando o número de desalojamentos atinge o valor dessa constante, as tabelas são redimensionadas.

Considere o exemplo mostrado na **Figura 7–17**. Nela estão representadas as duas tabelas de uma dispersão cuco bem como os valores de dispersão obtidos quando se aplicam as funções de dispersão f_1 e f_2 às chaves A, B, C, D, E e F, que devem ser inseridas nessas tabelas. A função f_1 está associada à tabela 1, ao passo que a função f_2 está associada à tabela 2. Mais precisamente, os valores de dispersão obtidos com a aplicação da função f_1 são índices da tabela 1 e os valores de dispersão obtidos com a aplicação da função f_2 são índices da tabela 2.

Tabela 1	
Índice	Chave
0	
1	
2	
3	
4	

Tabela 2	
Índice	Chave
0	
1	
2	
3	
4	

c	$f_1(c)$	$f_2(c)$
A	0	2
B	0	0
C	1	4
D	1	0
E	3	2
F	3	4

Figura 7–17: Condição Inicial de uma Tabela de Dispersão Cuco

A inserção da primeira chave (nesse caso, A) é sempre fácil, pois as duas tabelas estão vazias. O valor de dispersão $f_1(A)$ indica que a chave A deve ser inserida na posição 0 da tabela 1, o que ocorre normalmente, como mostra a **Figura 7–18**.

Tabela 1	
Índice	Chave
0	A
1	
2	
3	
4	

Tabela 2	
Índice	Chave
0	
1	
2	
3	
4	

c	$f_1(c)$	$f_2(c)$
A	0	2
B	0	0
C	1	4
D	1	0
E	3	2
F	3	4

Figura 7–18: Inserção sem Desalojamento em Tabela de Dispersão Cuco

Na inserção da chave B, nota-se que $f_1(B)$ resulta em 0, indicando que essa chave deve ser inserida na posição 0 da tabela 1. Ocorre, porém, que essa posição já está ocupada pela chave A. Então a chave A é movida para sua posição alternativa na tabela 2, que é determinada pelo valor de dispersão $f_2(A)$. O resultado é mostrado na **Figura 7–19**, que também mostra a inserção subsequente de C. Essa última inserção ocorre normalmente, visto que a posição 1 [resultante de $f_1(C)$] da tabela 1 encontra-se vazia.

Tabela 1	
Índice	Chave
0	B
1	C
2	
3	
4	

Tabela 2	
Índice	Chave
0	
1	
2	A
3	
4	

c	$f_1(c)$	$f_2(c)$
A	0	2
B	0	0
C	1	4
D	1	0
E	3	2
F	3	4

Figura 7–19: Dispersão Cuco: Inserção com e sem Desalojamento 1

A inserção da chave *D*, mostrada na **Figura 7–20**, novamente, requer o desalojamento de uma chave. Ou seja, a posição de *D* na tabela 1, dada por $f_1(D) = 1$, já está ocupada, o que requer que a chave *C*, que ocupa a referida posição, seja movida para sua posição alternativa na tabela 2, que é dada por $f_2(C) = 4$. Na mesma **Figura 7–20**, aparece a inserção da chave *E*, que ocorre normalmente, pois sua posição na tabela 1, dada por $f_1(E) = 3$, encontra-se vazia.

Tabela 1	
Índice	Chave
0	B
1	D
2	
3	E
4	

Tabela 2	
Índice	Chave
0	
1	
2	A
3	
4	C

c	$f_1(c)$	$f_2(c)$
A	0	2
B	0	0
C	1	4
D	1	0
E	3	2
F	3	4

Figura 7–20: Dispersão Cuco: Inserção com e sem Desalojamento 2

A inserção da chave *F* é a mais complicada e segue os seguintes passos:

1. A posição de *F* na tabela 1, dada por $f_1(F) = 3$, já está ocupada pela chave *E*, o que faz com que se tente mover *E* para a tabela 2.
2. A posição alternativa de *E* na tabela 2, dada por $f_2(E) = 2$, já está ocupada pela chave *A*, o que faz com que se tente mover *A* para a tabela 1.
3. A posição da chave *A* na tabela 1, dada por $f_1(A) = 0$, já está ocupada pela chave *B*. Logo essa última chave deve ser movida para sua posição alternativa na tabela *2*.
4. Finalmente, a posição da chave *B* na tabela 2, dada por $f_2(B) = 0$, está vazia e essa chave pode ser armazenada nessa posição. Este último passo encerra a inserção da chave *F* e o resultado é mostrado na **Figura 7–21**.

Tabela 1	
Índice	Chave
0	A
1	D
2	
3	F
4	

Tabela 2	
Índice	Chave
0	B
1	
2	E
3	
4	C

c	$f_1(c)$	$f_2(c)$
A	0	2
B	0	0
C	1	4
D	1	0
E	3	2
F	3	4

Figura 7–21: Dispersão Cuco: Inserção com Múltiplos Desalojamentos

Considerando o mesmo exemplo acima, suponha que houvesse uma chave *G* com valores de dispersão dados por $f_1(G) = 1$ e $f_2(G) = 2$. Nesse caso, a tentativa de inserção de G numa das tabelas resultaria na seguinte sequência de eventos:

1. A inserção de *G* na tabela 1 desaloja *D*
2. A inserção de *D* na tabela 2 desaloja *B*
3. A inserção de *B* na tabela 1 desaloja *A*
4. A inserção de *A* na tabela 1 desaloja *E*
5. A inserção de *E* na tabela 2 desaloja *F*
6. A inserção de *F* na tabela 1 desaloja *C*
7. A inserção de *C* na tabela 1 desaloja *G*
8. A inserção de *G* na tabela 2 desaloja *A*

⋮

Portanto a tentativa de inserção da chave *G* resulta num ciclo sem fim.

Aparentemente, o problema que surge com a tentativa de inserção da chave *G* é decorrente do fato de o fator de carga de cada tabela ser elevado, pois a inserção de *G* resultaria em $\alpha = 0,7$. Mas esse não é o caso aqui. Quer dizer, com os valores de dispersão utilizados, esse ciclo sem fim ocorreria mesmo que cada tabela tivesse milhares de posições disponíveis. Assim a solução para o referido problema consiste em considerar novas funções de dispersão com a respectiva reconstrução das tabelas de dispersão.

Felizmente, estudos revelam que a probabilidade de ocorrência de ciclo sem fim como aquele apresentado no último exemplo é muito pequena quando o fator de carga se situa abaixo de *0,5*. Além disso, é improvável que o número esperado de desalojamentos durante uma operação de inserção seja maior do que $\theta(\log n)$. Assim pode-se redimensionar as tabelas após a detecção de um certo número de desalojamentos. Por outro lado, se o fator de carga for maior do que *0,5*, a probabilidade de ocorrência de um ciclo sem fim se torna bastante elevada.

7.6.2 Operações Básicas

Busca

Uma operação de busca numa tabela de dispersão cuco segue o algoritmo delineado na **Figura 7–22**.

ALGORITMO BUSCAEMDISPERSÃOCUCO

ENTRADA: Uma tabela de dispersão cuco e uma chave de busca *c*

SAÍDA: O valor associado à chave, se ela for encontrada, ou um valor informando o fracasso da operação

1. Obtenha o valor de dispersão d_1 de *c* usando a função de dispersão associada à primeira tabela
2. Se a chave na posição d_1 da primeira tabela for igual a *c*, retorne o valor associado a essa chave
3. Obtenha o valor de dispersão d_2 de *c* usando a função de dispersão associada à segunda tabela
4. Se a chave na posição d_2 da segunda tabela for igual a *c*, retorne o valor associado a essa chave
5. Retorne um valor informando que a busca não obteve sucesso

FIGURA 7–22: ALGORITMO DE BUSCA EM DISPERSÃO CUCO

Inserção

Uma operação de inserção numa tabela de dispersão cuco segue o algoritmo apresentado na **Figura 7–23**.

ALGORITMO INSEREEMDISPERSÃOCUCO

ENTRADA: O conteúdo de um novo elemento com chave *c*

ENTRADA/SAÍDA: Uma tabela de dispersão cuco

SAÍDA: Um valor informando o sucesso da operação

1. Se a chave já existe na tabela, encerre informando o fracasso da operação (a chave é primária)
2. Calcule o número limite de desalojamentos
3. Enquanto o número máximo de desalojamentos não for atingido, faça:
   - **3.1** Obtenha o valor de dispersão d_1 da chave do elemento a ser inserido aplicando-lhe a função de dispersão associada à primeira tabela
   - **3.2** Se a posição d_1 da primeira tabela estiver vazia, efetue a inserção e retorne indicando o sucesso da operação

CONTINUA

FIGURA 7–23: ALGORITMO DE INSERÇÃO EM DISPERSÃO CUCO

ALGORITMO INSEREEMDISPERSÃOCUCO (CONTINUAÇÃO)

3.3 Caso contrário, faça o seguinte:
- **3.3.1** Guarde o elemento que se encontra na posição d_1
- **3.3.2** Armazene o novo elemento na posição d_1
- **3.3.3** Faça com que o elemento a ser inserido passe a ser aquele que foi guardado

3.4 Obtenha o valor de dispersão d_2 da chave do elemento a ser inserido aplicando-lhe a função de dispersão associada à segunda tabela

3.5 Se a posição d_2 da segunda tabela estiver vazio, efetue a inserção e retorne indicando o sucesso da operação

3.6 Caso contrário, faça o seguinte:
- **3.6.1** Guarde o elemento que se encontra na posição d_2
- **3.6.2** Armazene o novo elemento na posição d_2
- **3.6.3** Faça com que o elemento a ser inserido passe a ser aquele que foi guardado

4. Redimensione a tabela usando o algoritmo **REDIMENSIONATABELACUCO**

FIGURA 7–23 (CONT.): ALGORITMO DE INSERÇÃO EM DISPERSÃO CUCO

O algoritmo **REDIMENSIONATABELACUCO** invocado por **INSEREEMDISPERSÃOCUCO** segue os passos apresentados na **Figura 7–24**.

ALGORITMO REDIMENSIONATABELACUCO

ENTRADA: O conteúdo de um novo elemento

ENTRADA/SAÍDA: Uma tabela de dispersão cuco

SAÍDA: Um valor informando o sucesso da operação

1. Crie uma nova tabela de dispersão cuco com tamanho maior do que o tamanho daquela recebida como entrada
2. Insira as chaves da antiga tabela na nova tabela
3. Libere o espaço ocupado pela antiga tabela
4. Insira o novo elemento na nova tabela

FIGURA 7–24: ALGORITMO DE REDIMENSIONAMENTO DE TABELA DE DISPERSÃO CUCO

Remoção

Uma operação de remoção numa tabela de dispersão cuco segue o algoritmo apresentado na **Figura 7–25**.

ALGORITMO REMOVEEMDISPERSÃOCUCO

ENTRADA: A chave c do elemento a ser removido

ENTRADA/SAÍDA: Uma tabela de dispersão cuco

SAÍDA: Um valor informando o sucesso da operação

1. Obtenha o valor de dispersão d_1 de c usando a função de dispersão associada à primeira tabela
2. Se a chave na posição d_1 da primeira tabela for igual a c, remova o elemento que se encontra nessa posição e encerre informando o sucesso da operação

CONTINUA

FIGURA 7–25: ALGORITMO DE REMOÇÃO EM DISPERSÃO CUCO

Algoritmo RemoveEmDispersãoCuco (Continuação)

3. Se a chave na posição d_2 da segunda tabela for igual a c, remova o elemento que se encontra nessa posição e encerre informando o sucesso da operação
4. Retorne um valor informando que a remoção não obteve sucesso

Figura 7–25 (Cont.): Algoritmo de Remoção em Dispersão Cuco

7.6.3 Implementação

Definições de Tipos

Além dos tipos `tCEP` e `CEP_Ind` definidos na **Seção 7.3.2**, os seguintes tipos serão usados na implementação de dispersão cuco a ser apresentada aqui:

```
   /* Tipo usado para indicar o status de um elemento da tabela */
typedef enum {VAZIO, OCUPADO} tStatusCuco;

   /* Tipo de um ponteiro para uma função de dispersão cuco */
typedef unsigned int (*tFDispersaoCuco) (tCEP, int);

   /* Tipo dos elementos de uma tabela de dispersão cuco */
typedef struct {
           tCEP_Ind    chaveEIndice;
           tStatusCuco status;
        } tColetorCuco;

   /* Tipo de uma tabela de dispersão cuco */
typedef struct {
           tColetorCuco   *tab1; /* Tabela 1 */
           tColetorCuco   *tab2; /* Tabela 2 */
           tFDispersaoCuco fD1;  /* Função de dispersão 1  */
           tFDispersaoCuco fD2;  /* Função de dispersão 2  */
           int             tam;  /* Tamanho de cada tabela */
           int             nChaves; /* Número de chaves    */
        } tTabelaCuco;
```

Criação

A função `CriaTabelaCuco()` cria e inicializa uma tabela de dispersão cuco e seus parâmetros são:

- `tabelas` (saída) — a tabela cuco criada e iniciada
- `nElementos` (entrada) — número de posições da tabela de dispersão
- `fD1`, `fD2` (entrada) — endereços das funções de dispersão utilizadas

```
void CriaTabelaCuco( tTabelaCuco *tabelas, int nElementos,
                     tFDispersaoCuco fD1, tFDispersaoCuco fD2 )
{
   int i;

      /* Aloca espaço para as duas tabelas */
   tabelas->tab1 = calloc(nElementos, sizeof(tColetorCuco));
   tabelas->tab2 = calloc(nElementos, sizeof(tColetorCuco));
   ASSEGURA( tabelas->tab1 && tabelas->tab2,
             "\nImpossivel alocar a tabela de dispersão\n" );

      /* Inicia as duas tabelas */
   for (i = 0; i < nElementos; ++i) {
         /* Todos os elementos estão inicialmente desocupados */
      tabelas->tab1[i].status = VAZIO;
```

```
      tabelas->tab2[i].status = VAZIO;
   }

      /* Inicia as funções de dispersão */
   tabelas->fD1 = fD1;
   tabelas->fD2 = fD2;

   tabelas->nChaves = 0; /* Inicia o número de chaves nas tabelas */
}
```

Busca

A função `BuscaCuco()`, apresentada a seguir, executa uma busca simples numa tabela de dispersão cuco. Essa função retorna o índice do registro no arquivo de registros, se a chave for encontrada, ou `-1`, em caso contrário, e seus parâmetros são:

- `tabela` (entrada) — a tabela de dispersão
- `chave` (entrada) — a chave de busca

```
int BuscaCuco( tTabelaCuco tabela, tCEP chave )
{
   int pos; /* Essa variável não é estritamente necessária, mas */
            /* ela facilita a escrita das expressões seguintes  */

   pos = tabela.fD1(chave, tabela.tam);

      /* Verifica se a chave se encontra na primeira tabela */
   if ( tabela.tab1[pos].status == OCUPADO &&
        !strcmp(tabela.tab1[pos].chaveEIndice.chave, chave) )
      return tabela.tab1[pos].chaveEIndice.indice; /* Chave encontrada */

   /* A chave ainda não foi encontrada. Verifica */
   /* se ela se encontra na segunda tabela.      */

   pos = tabela.fD2(chave, tabela.tam);

   if ( tabela.tab2[pos].status == OCUPADO &&
        !strcmp(tabela.tab2[pos].chaveEIndice.chave, chave) )
      return tabela.tab2[pos].chaveEIndice.indice; /* Chave encontrada */

   return -1; /* A chave não foi encontrada em nenhuma das tabelas */
}
```

Inserção

A função `InsereCuco()`, apresentada a seguir, insere uma chave de uma tabela de dispersão cuco. Essa função retorna `1`, se a chave for inserida, ou `0`, em caso contrário, e seus parâmetros são:

- `tabela` (entrada/saída) — a tabela de dispersão
- `chave` (entrada) — a chave de busca

```
int InsereCuco(tTabelaCuco *tabela, tCEP_Ind chaveEIndice)
{
   int          i,
                pos,
                maxDesalojamentos;
   tCEP_Ind     aux;

      /* Verifica se a chave já existe na tabela */
   if (BuscaCuco(*tabela, chaveEIndice.chave) >= 0)
      return 0; /* Chave já existe na tabela */
```

```
   /* Calcula o valor limite de desalojamentos. O valor   */
   /* nlog n é sugerido pelos criadores da dispersão cuco */
maxDesalojamentos = tabela->nChaves*log2(tabela->nChaves);

   /* Tenta efetuar a inserção numa das tabelas */
for (i = 0; i < maxDesalojamentos; ++i) {
   /*                                 */
   /* Tenta inserir na primeira tabela */

   pos = tabela->fD1(chaveEIndice.chave, tabela->tam);

   if (tabela->tab1[pos].status == VAZIO) {
      /*                                                          */
      /* Foi encontrado um espaço vazio na primeira tabela. Efetua-se a */
      /* inserção e retorna-se  indicando o sucesso da operação         */
      /*                                                          */

         /* Insere o par chave/índice */
      tabela->tab1[pos].chaveEIndice = chaveEIndice;

      tabela->tab1[pos].status = OCUPADO; /* Esta posição passa a ser ocupada */

      ++tabela->nChaves; /* O número de chaves aumentou */

      return 1;
   } else {
      /*                                                                */
      /* Não foi encontrado um espaço vazio. É preciso desalojar a chave que */
      /* se encontra nessa posição e tentar inseri-la na segunda tabela      */
      /*                                                                */

      aux = tabela->tab1[pos].chaveEIndice; /* Guarda chave a ser desalojada */
         /* Armazena a nova chave no lugar da chave desalojada */
      tabela->tab1[pos].chaveEIndice = chaveEIndice;
         /* Tentar-se-a inserir a chave desalojada na segunda tabela */
      chaveEIndice = aux;
   }

   /*                                 */
   /* Tenta inserir na segunda tabela */

   pos = tabela->fD2(chaveEIndice.chave, tabela->tam);

   if (tabela->tab2[pos].status == VAZIO) {
      /*                                                       */
      /* Foi encontrado um espaço vazio na segunda tabela. Efetua-se */
      /* a inserção e retorna-se indicando o sucesso da operação     */

         /* Insere o par chave/índice */
      tabela->tab2[pos].chaveEIndice = chaveEIndice;

         /* Esta posição passa a ser ocupada */
      tabela->tab2[pos].status = OCUPADO;

      ++tabela->nChaves; /* O número de chaves aumentou */

      return 1;
   } else {
      /*                                                                */
      /* Não foi encontrado um espaço vazio. É preciso desalojar a chave que */
      /* se encontra nessa posição e tentar inseri-la na primeira tabela     */

      aux = tabela->tab2[pos].chaveEIndice; /* Guarda a chave a ser desalojada */
         /* Armazena a nova chave no lugar da chave desalojada */
```

```
         tabela->tab2[pos].chaveEIndice = chaveEIndice;

            /* Tentar-se-á inserir a chave desalojada na primeira tabela */
         chaveEIndice = aux;
      }
   }

   /*                                                                  */
   /* Se o laço terminou sem que houvesse retorno, considera-se que    */
   /* o processo entrou em repetição  sem fim (de acordo com o critério */
   /* especificado). Portanto, a tabela será reconstruída e a chave    */
   /* restante será inserida nessa nova tabela.                        */

   RedimensionaCuco(tabela, chaveEIndice);

   return 1;
}
```

Note que a função `InsereCuco()` tem uma condição para detectar a ocorrência de ciclo sem fim na sequência de inserções. Existem várias maneiras de expressar precisamente essa condição. Pode-se, por exemplo, contar o número de iterações do laço de repetição e considerar que existe um ciclo se esse número ultrapassar um certo valor como, por exemplo, $n \cdot \log n$ (esse valor é sugerido pelos criadores da dispersão cuco). Essa função faz uso da função `RedimensionaCuco()`, que redimensiona as tabelas de dispersão quando se percebe, de acordo com um critério previamente especificado, que ocorreu um ciclo sem fim ao se tentar inserir uma chave.

A função `RedimensionaCuco()`, vista abaixo, redimensiona uma tabela de dispersão cuco e seus parâmetros são:

- `tabela` (entrada/saída) — a tabela de dispersão
- `chaveEIndice` (entrada) — o par chave/índice que será inserido após o redimensionamento da tabela

```
static void RedimensionaCuco(tTabelaCuco *tabela, tCEP_Ind chaveEIndice)
{
   tTabelaCuco novaTabela; /* A nova tabela */
   int         i,
               teste, /* Resultado de uma inserção */
               novoTamanho; /* Tamanho da nova tabela */

      /* O novo tamanho será 50% maior do que o antigo valor */
   novoTamanho = 1.5*tabela->tam;

      /* Cria uma nova tabela de dispersão cuco */
   CriaTabelaCuco(&novaTabela, novoTamanho, tabela->fD1, tabela->fD2);

      /* Insere as chaves da antiga tabela na nova tabela */
   for (i = 0; i < tabela->tam; ++i) {
         /* Verifica se a posição corrente da primeira tabela contém uma chave */
      if (tabela->tab1[i].status == OCUPADO) {
            /* Existe uma chave nesta posição */
         teste = InsereCuco( &novaTabela, tabela->tab1[i].chaveEIndice );
            /* Esta inserção não pode falhar */
         ASSEGURA( teste, "Falha de insercao em redimensionamento" );
      }

         /* Verifica se a posição corrente da segunda tabela contém uma chave */
      if (tabela->tab2[i].status == OCUPADO) {
            /* Existe uma chave nesta posição */
         teste = InsereCuco( &novaTabela, tabela->tab2[i].chaveEIndice );
            /* Esta inserção não pode falhar */
         ASSEGURA( teste, "Falha de insercao em redimensionamento" );
      }
   }
```

```
      /* Neste ponto, todas as chaves da antiga tabela foram     */
      /* inseridas, de modo que a tabela antiga pode ser extinta */
   DestroiTabelaCuco(tabela);

   *tabela = novaTabela; /* Substitui a tabela antiga com a nova tabela */

    teste = InsereCuco(tabela, chaveEIndice); /* Insere a chave que restou */

      /* Esta inserção não pode falhar */
    ASSEGURA(teste, "Falha de insercao em redimensionamento");
}
```

A função `DestroiTabelaCuco()` chamada por `RedimensionaCuco()` libera o espaço ocupado por uma tabela de dispersão cuco e é relativamente fácil de implementar.

Remoção

A função `RemoveCuco()`, apresentada a seguir, remove uma chave de uma tabela de dispersão cuco. Essa função retorna `1`, se a chave for encontrada e removida, ou `0`, em caso contrário, e seus parâmetros são:

- `tabela` (entrada/saída) — a tabela de dispersão
- `chave` (entrada) — a chave de busca

```
int RemoveCuco(tTabelaCuco *tabela, tCEP chave)
{
   int pos; /* Essa variável não é estritamente necessária, mas */
            /* ela facilita a escrita das expressões seguintes  */

   pos = tabela->fD1(chave, tabela->tam);

      /* Tenta efetuar a remoção na primeira tabela */
   if ( tabela->tab1[pos].status == OCUPADO &&
        !strcmp(tabela->tab1[pos].chaveEIndice.chave, chave) ) {
      tabela->tab1[pos].status = VAZIO; /* Chave encontrada */

      --tabela->nChaves;

      return 1;
   }

   /*** A chave ainda não foi removida. Tenta removê-la da segunda tabela. ***/

   pos = tabela->fD2(chave, tabela->tam);

   if ( tabela->tab2[pos].status == OCUPADO &&
        !strcmp(tabela->tab2[pos].chaveEIndice.chave, chave) ) {
      tabela->tab2[pos].status = VAZIO; /* Chave encontrada */

      --tabela->nChaves;

      return 1;
   }

   return 0;  /* A chave não foi encontrada */
}
```

7.6.4 Análise

Dispersão cuco é relativamente pouco usada. Ela requer tabelas de dispersão com preenchimento muito esparso para que possa obter um bom tempo em operações de inserção. De fato, é preciso ter cerca de *50%* de cada tabela vazia para obter bom desempenho. Assim dispersão cuco ora é rápida, mas ocupa muito espaço, ora é lenta e usa espaço eficientemente. Ou seja, as duas coisas nunca ocorrem ao mesmo tempo. Outros algoritmos

são eficientes em termos de tempo e espaço, embora eles sejam piores do que dispersão cuco quando apenas tempo ou espaço é levado em consideração.

De acordo com os inventores da dispersão cuco, esse esquema de dispersão é extremamente sensível à escolha das funções de dispersão usadas. Segundo eles, funções de dispersão padrão por eles testadas não funcionaram satisfatoriamente com esse esquema de dispersão. Além disso, o custo temporal, que se espera que seja $\theta(1)$ quando o fator de carga é menor do que $0,5$, é severamente afetado quando esse fator de carga se aproxima de $0,5$ ou é maior do que esse valor.

É relativamente fácil mostrar que o custo temporal amortizado de inserção numa tabela de dispersão cuco é $\theta(1)$ (v. artigo de Pagh e Rodler enumerado na **Bibliografia**). Portanto uma tabela de dispersão cuco tem custo temporal $\theta(1)$ no pior caso para buscas e remoções e custo temporal amortizado $\theta(1)$ para inserções.

Uma desvantagem de dispersão cuco é que à medida que o fator de carga aumenta, o mesmo ocorre com o número médio de operações de desalojamentos. Isso retarda inserções consideravelmente. Assim é aconselhável manter o fator de carga abaixo de 0,5 no caso de dispersão cuco com duas funções de dispersão. Outra desvantagem de dispersão cuco é que, mesmo que o número de comparações entre chaves seja limitado, elas são sempre feitas em tabelas diferentes, o que prejudica a localidade de referência do processo (v. **Capítulo 1**). Com endereçamento aberto básico (v. **Seção 7.4**), a principal vantagem é que as chaves se encontram em posições próximas de memória, o que favorece a localidade de referência. Essa vantagem é perdida com dispersão cuco, pois ela troca boa localidade de referência por um menor número de comparações.

A **Tabela 7–2** resume vantagens e desvantagens de dispersão cuco.

Vantagens	Desvantagens
Busca e remoção apresentam custo temporal $\theta(1)$	**Inserção é muito sensível à escolha das funções de dispersão utilizadas**
Inserção tem custo temporal amortizado $\theta(1)$, desde que $\alpha < 0,5$	**Custo espacial $\theta(n)$**

Tabela 7–2: Vantagens e Desvantagens de Dispersão Cuco

Existem esquemas alternativos baseados na dispersão cuco discutida aqui. Por exemplo, uma alternativa é usar mais de duas funções de dispersão (e mais de duas tabelas). Explorar essas alternativas está além do escopo deste livro.

7.7 Análise de Técnicas de Dispersão

Tabelas de dispersão apresentam como principal vantagem a rapidez com que efetuam busca, inserção e remoção. Quando ocorrem poucas colisões, o custo temporal dessas operações é $\theta(1)$. Por outro lado, a principal desvantagem do uso de dispersão é que, no pior caso, essas operações têm custo temporal linear, o que equipara tabelas de dispersão a tabelas indexadas com busca sequencial. Além disso, tabelas de dispersão não facilitam o acesso a chaves em ordem, como ocorre, por exemplo, com árvores de busca. Usando-se tabelas de dispersão também não é fácil encontrar a maior ou a menor chave nem efetuar buscas de intervalo.

Árvores binárias de busca são capazes de efetuar esses tipos de busca além de busca convencional com custo temporal $\theta(\log n)$, que muitas vezes não é tão superior ao custo $\theta(1)$ atribuído a operações básicas em tabelas de dispersão. Por outro lado, enquanto o pior caso de busca em tabela de dispersão é tipicamente resultante de erro de implementação, o mau desempenho pode facilmente afligir árvores binárias de busca que não são balanceadas. Ademais, implementações de árvores binárias de busca balanceadas são complicadas, o que as tornam susceptíveis a erros de programação. Portanto, quando são desejadas apenas buscas exatas ou quando se

suspeita que os dados podem resultar em árvores não balanceadas, tabela de dispersão é uma boa escolha como estrutura de dados.

Quando se usa uma tabela de dispersão, é importante escolher cuidadosamente a função de dispersão e dar atenção especial ao fator de carga, pois, caso contrário, corre-se o risco de obter o custo temporal de pior caso.

Em dispersão com encadeamento, o fator de carga deve ser próximo de *1*, embora a eficiência não seja tão afetada se esse não for o caso, a não ser que o fator de carga se torne grande demais. Em dispersão com endereçamento aberto, o fator de carga não deve ser maior do que *0,5*. Se for usada sondagem linear, o desempenho degenera-se rapidamente à medida que o fator de carga se aproxima de *1*. Redimensionamento deve ser implementado para aumentar ou diminuir de tamanho da tabela de modo a manter o fator de carga dentro de limites aceitáveis.

Em dispersão com endereçamento aberto, dispersão dupla é a melhor abordagem de resolução de colisões, seguida de perto por sondagem quadrática. Mas, se existe muita memória disponível e espera-se que haja muitas inserções depois de a tabela ter sido criada, sondagem linear é mais simples de implementar. Nesse último caso, se o fator de carga for mantido abaixo de *0,5*, a eficiência não será afetada de modo significativo.

Se o número de chaves que serão inseridas numa tabela de dispersão for desconhecido quando ela for criada, é aconselhável usar dispersão com encadeamento, pois o acréscimo de dados que não foram previstos não causará grandes danos à eficiência. O aumento do fator de carga em dispersão com encadeamento causa maior perda em eficiência do que em dispersão com endereçamento aberto, mas, em dispersão com encadeamento, essa queda de eficiência é apenas linear.

Em dispersão cuco, busca requer verificação de apenas dois coletores da tabela de dispersão, de maneira que, no pior caso, o custo temporal dessa operação é $\theta(1)$. Remoções também podem ser efetuadas com custo temporal constante no pior caso tempo. Esses custos contrastam com aqueles obtidos no pior caso para muitos outros algoritmos usados com tabelas de dispersão. Contudo o processo de inserção em dispersão cuco pode fracassar, entrando em repetição infinita ou encontrando uma cadeia mais longa do que um limite prefixado. Nesse caso, a tabela de dispersão precisa ser reconstruída usando novas funções de dispersão, de sorte que o custo temporal de inserção no pior caso é $\theta(n)$.

7.8 Exemplos de Programação

7.8.1 Testando Funções de Dispersão Prontas

Problema: Escreva um programa que teste algumas funções de dispersão prontas (v. **Apêndice C**) para verificar se cada uma delas apresenta boa distribuição de valores de dispersão. O banco de dados a ser usado nos testes é o de CEPs (v. **Seção 3.2**).

Solução: A função **main()** do programa é apresentada abaixo.

```
int main(void)
{
   tLista      lista; /* Lista de chaves */
   const char *opcoes[] = { "Testa Dispersao de Jenkins",
                            "Testa Dispersao DJB",
                            "Testa Dispersao DJB2",
                            "Testa Dispersao SAX",
                            "Testa Dispersao FNV",
                            "Testa Dispersao JSW",
                            "Testa Todas as Dispersoes",
                            "Encerra o programa"
                          };
   int         op, /* Opção escolhida pelo usuário */
```

```
            nOpcoes = sizeof(opcoes)/sizeof(opcoes[0]);
FILE       *stream;

MedidaDeTempo();
printf("\nCriando lista...");

IniciaLista(&lista); /* Inicia a lista que armazena as chaves */

   /* Lê o conteúdo do arquivo de dados e armazena as chaves na lista */
LeArquivo(NOME_ARQUIVO_BIN, &lista);

printf("\n...lista criada com sucesso\n");
MedidaDeTempo();

   /* O laço a seguir encerra quando o usuário */
   /* escolher a opção de encerramento         */
while (1) {
   ApresentaMenu(opcoes, nOpcoes);

   op = LeOpcao("12345678");

      /* Verifica se o usuário quer encerrar o programa */
   if (op == '8') { /* Encerra o programa */
      DestroiLista(&lista);
      break; /* Saída do laço */
   }

      /* Processa as demais opções */
   switch (op) {
      case '1': /* Dispersão de Jenkins */
         printf("\n\t>>> Dispersao de Jenkins <<<\n");
         TestaDispersao(&lista, DispersaoJenkins, stdout);
         break;

      case '2': /* Dispersão DJB */
         printf("\n\t>>> Dispersao DJB <<<\n");
         TestaDispersao(&lista, DispersaoDJB, stdout);
         break;

      case '3': /* Dispersão DJB2 */
         printf("\n\t>>> Dispersao DJB2 <<<\n");
         TestaDispersao(&lista, DispersaoDJB2, stdout);
         break;

      case '4': /* Dispersão SAX */
         printf("\n\t>>> Dispersao SAX <<<\n");
         TestaDispersao(&lista, DispersaoSAX, stdout);
         break;

      case '5': /* Dispersão FNV */
         printf("\n\t>>> Dispersao FNV <<<\n");
         TestaDispersao(&lista, DispersaoFNV, stdout);
         break;

      case '6': /* Dispersão JSW */
         printf("\n\t>>> Dispersao JSW <<<\n");
         TestaDispersao(&lista, DispersaoJSW, stdout);
         break;

      case '7': /* Testa todas as funções */
         stream = fopen(NOME_ARQUIVO_RES, "w");
         ASSEGURA(stream, "Arquivo nao foi aberto");
```

```
            fprintf( stream, "***** Testes de Funcoes de "
                    "Dispersao Prontas *****\n" );

            fprintf(stream, "\n\t>>> Dispersao de Jenkins <<<\n");
            TestaDispersao(&lista, DispersaoJenkins, stream);

            fprintf(stream, "\n\t>>> Dispersao DJB <<<\n");
            TestaDispersao(&lista, DispersaoDJB, stream);

            fprintf(stream, "\n\t>>> Dispersao DJB2 <<<\n");
            TestaDispersao(&lista, DispersaoDJB2, stream);

            fprintf(stream, "\n\t>>> Dispersao SAX <<<\n");
            TestaDispersao(&lista, DispersaoSAX, stream);

            fprintf(stream, "\n\t>>> Dispersao FNV <<<\n");
            TestaDispersao(&lista, DispersaoFNV, stream);

            fprintf(stream, "\n\t>>> Dispersao JSW <<<\n");
            TestaDispersao(&lista, DispersaoJSW, stream);

            fclose(stream);

          printf("\n\t>>> Resultado escrito no arquivo "%s\" <<<\n", NOME_ARQUIVO_RES);
            break;

         default: /* O programa não deve chegar até aqui */
            printf("\nEste programa contem um erro\n");
            return 1;
      }
   }

   printf( "\n\t>>> Obrigado por usar este programa\n\n"); /* Despede-se do usuário */

   return 0;
}
```

A função `MedidaDeTempo()` chamada pela função **main()** acima foi definida na **Seção 1.7.1**.

Essa função **main()** começa lendo o conteúdo do arquivo de dados e armazenando suas chaves numa lista indexada. Então essa função leva a cabo uma interação dirigida por menu clássica. Um exemplo dessa interação é apresentado abaixo:

```
Escolha uma das opcoes a seguir:
        [1] Testa Dispersao de Jenkins
        [2] Testa Dispersao DJB
        [3] Testa Dispersao DJB2
        [4] Testa Dispersao SAX
        [5] Testa Dispersao FNV
        [6] Testa Dispersao JSW
        [7] Testa Todas as Dispersoes
        [8] Encerra o programa

        >>> 3

        >>> Dispersao DJB2 <<<

>>> Tamanho da tabela: 673580
>>> Posicoes nao ocupadas: 245936
>>> Numero total de colisoes: 423737
>>> Maximo de colisoes numa posicao: 9
>>> Desperdicio de memoria: 36.51%
```

O teste de cada função de dispersão é efetuado pela função `TestaDispersao()`, que tem como parâmetros:

- `lista` (entrada) — a lista de chaves que será utilizada no teste
- `funcao` (entrada) — ponteiro para a função de dispersão que será testada
- `stream` (entrada) — stream de texto associado ao meio de saída no qual o resultado será escrito

```
void TestaDispersao( tLista *lista, tFDispersao funcao, FILE *stream )
{
   int        i,
              tam; /* Número de chaves na lista */
   unsigned   dispersao; /* Um valor de dispersão */
   tElemento  elemento;  /* Um elemento da lista  */
   int       *colisoes; /* Ponteiro para um array que conta o número de colisões */

   tam = Comprimento(lista);

      /* Aloca o array que contará o número de colisões */
   colisoes = calloc(tam, sizeof(int));
   ASSEGURA(colisoes, "Impossivel alocar array");

   for (i = 0; i < tam; ++i) {
      elemento = ObtemElemento(lista, i); /* Obtém um elemento da lista */

         /* Calcula o valor de dispersão da chave */
      dispersao = funcao(elemento.chave)%tam;

        /* Conta quantas vezes esse valor de dispersão foi obtido */
      ++colisoes[dispersao];
   }

     /* Analisa e apresenta o resultado do teste */
   AnalisaColisoes(colisoes, tam, stream);

   free(colisoes); /* Esse array não é mais necessário */
}
```

A função `AnalisaColisoes()` escreve num stream de texto o resultado de uma análise da lista de colisões obtida pela função `TestaDispersao()`. Os parâmetros da função `AnalisaColisoes()` são:

- `colisoes[]` (entrada/saída) — array que representa a lista de colisões
- `tam` (entrada) — tamanho da lista de colisões
- `stream` (entrada) — stream de texto associado ao meio de saída no qual o resultado será escrito

```
void AnalisaColisoes(int colisoes[], int tam, FILE *stream)
{
   int i,
       vazio = 0,
       nColisoes = 0;

      /* Ordena a lista de colisões em ordem decrescente de número de colisões */
   qsort(colisoes, tam, sizeof(colisoes[0]), ComparaColisoes);

   for (i = 0; i < tam; ++i) {
         /* Calcula o número de posições da tabela que não foram usadas */
      if (!colisoes[i])
         ++vazio;

         /* Calcula o número de colisões */
      if (colisoes[i] > 1)
         nColisoes += colisoes[i];
   }
```

```
      /* Escreve os resultados no stream */
   fprintf( stream, "\n>>> Tamanho da tabela: %d", tam );
   fprintf( stream, "\n>>> Posicoes nao ocupadas: %d", vazio );
   fprintf( stream, "\n>>> Numero total de colisoes: %d", nColisoes );
   fprintf( stream, "\n>>> Maximo de colisoes numa posicao: %d", colisoes[0] );
   fprintf( stream, "\n>>> Desperdicio de memoria: %4.2f%%\n",
            100*((double) vazio/tam) );
}
```

O leitor encontrará o programa completo no site dedicado a este livro na internet.

7.8.2 O Filtro de Bloom

Preâmbulo: O **filtro de Bloom**, criado por Burton Howard Bloom em 1970 (v. **Bibliografia**), é uma estrutura de dados probabilística que usa espaço de modo bem eficiente. Ele usa o conceito de dispersão para memorizar dados que lhe tenham sido apresentados sem precisar armazená-los e, assim, é usado para testar se um elemento é um membro de um conjunto (**consulta de pertinência**). Casamentos falso-positivos são possíveis, mas casamentos falso-negativos não o são (v. adiante), de modo que um filtro de Bloom tem uma taxa de recordação de 100%. Em outras palavras, uma consulta retorna *possivelmente está no conjunto* ou *definitivamente não está no conjunto*. Num filtro de Bloom original, elementos podem ser acrescentados ao conjunto, mas não podem ser removidos. Quanto maior o número de elementos acrescentados ao conjunto, maior será a probabilidade de falso-positivos.

Um **erro falso-positivo**, ou apenas **falso-positivo** ou um **alarme falso**, é um resultado que indica que uma dada condição foi satisfeita, quando ela realmente não foi satisfeita. Isto é, um efeito positivo foi assumido erroneamente. Por exemplo, em Engenharia de Software, um programa sendo testado é considerado reprovado num teste quando, de fato, ele funciona corretamente. Isso pode ocorrer quando o engenheiro responsável pelos testes registra, por engano ou negligência, que encontrou um bug numa função que funciona perfeitamente bem.

Um **erro falso-negativo**, ou apenas **falso-negativo**, ocorre quando o resultado de um teste indica que uma condição não foi satisfeita, quando realmente ela foi bem-sucedida. Isto é, erroneamente nenhum defeito foi encontrado. Considerando o mesmo exemplo de teste de programas, isso ocorre quando um programa é considerado aprovado num teste quando, na realidade, ele contém pelo menos um bug que passou despercebido. Falso-negativos são mais perigosos do que falso-positivos porque eles podem causar sérios danos (p. ex., quando um programa antivírus não identifica um vírus).

A criação de um filtro de Bloom começa com um array de bits iniciados com zero. Para registrar a ocorrência de um valor simplesmente avaliam-se k funções de dispersão diferentes e consideram-se os k valores resultantes como índices do array e liga-se (i.e., atribui-se *1* a) cada um dos k elementos do array. Esse processo é repetido para cada item que seja encontrado. A probabilidade de falso-positivo num filtro de Bloom pode ser controlada variando-se o tamanho da tabela e o número de funções de dispersão utilizadas. Isto é, o valor de dispersão não é único para um dado item de dados e não se pode inverter uma função de dispersão para obter valores.

Um filtro de Bloom vazio é um array de m bits, todos eles iniciados com *0*. Deve existir também k funções de dispersão diferentes, cada uma das quais associa algum elemento do conjunto a uma das m posições do array com uma distribuição aleatória uniforme. Tipicamente, k é uma constante muito menor do que m, que é proporcional ao número de elementos a ser acrescentados. As escolhas de k e da constante de proporcionalidade m são determinadas pela taxa de falsos positivos

pretendida para o filtro. Para acrescentar um elemento, alimenta-se cada uma das *k* funções de dispersão com o valor desse elemento de modo a obter *k* posições no array de bits. Então ligam-se os bits em todas essas posições (i.e., atribui-se *1* ao conteúdo de cada uma dessas posições).

Para verificar se um item se encontra no filtro, as *k* funções de dispersão são aplicadas e examinam-se quais elementos do array estão ligados. Se qualquer um deles é zero pode-se ter 100% de certeza que o item nunca foi encontrado antes. Entretanto, mesmo que todos os bits sejam iguais a *1*, não se pode concluir que o item tenha sido encontrado antes porque todos os bits poderiam ter sido ligados pelas *k* funções de dispersão aplicadas a vários outros itens. Tudo que se pode concluir é que é *provável* que se tenham encontrado os itens antes.

Note que é impossível remover um item de um filtro de Bloom original, pois não se pode desligar um bit que pertença a um item porque ele também pode ter sido ligado por outro item. Quer dizer, um elemento é associado a *k* bits e, embora atribuir *0* a qualquer um desses *k* bits seja suficiente para remover um elemento, isso também resulta na remoção de qualquer outro elemento que esteja associado a esse mesmo bit. Uma vez que não há nenhuma maneira de determinar se qualquer outro elemento tenha sido acrescentado que afeta os bits de um elemento a ser removido, desligar (i.e., atribuir *0* a um bit) qualquer dos bits introduziria uma possibilidade para falso-negativos.

Se boa parte de um array de bits está vazio (i.e., se a maioria dos seus bits está desligada) e as *k* funções de dispersão são independentes entre si, então a probabilidade de um resultado falso-positivo (i.e., concluir que item foi encontrando antes quando realmente ele não foi) é baixa. Por exemplo, se há apenas *k* bits ligados (i.e., iguais a *1*) pode-se concluir que a probabilidade de um resultado falso-positivo é muito próxima de zero, visto que a única possibilidade de erro é que se tenha introduzido um item que produza os mesmos *k* valores de dispersão, o que é improvável desde que as funções sejam independentes.

À medida que o array de bits é preenchido a probabilidade de um falso-positivo gradualmente aumenta. Obviamente, quando o array de bits está completo (i.e., com todos os bits ligados) cada item consultado é tido como sendo encontrado antes. Assim pode-se trocar espaço por precisão assim como tempo. Um filtro de Bloom pode também trocar precisão por espaço. Se você pensa que para armazenar um string de *n* bytes requer *n* bytes, então num filtro de Bloom ele consome apenas *k* bits e *k* comparações, mas existe a possibilidade de falso-positivos. À medida que *k* aumenta o armazenamento necessário aumenta assim como o número de comparações, mas a possibilidade de falso-positivo decresce.

A **Figura 7–26 (a)** mostra um filtro de Bloom recém-iniciado, enquanto a **Figura 7–26 (b)** apresenta esse mesmo filtro após a inserção do string s_1, com as funções de dispersão f_1, f_2, f_3 e f_4 que resultam, respectivamente, em *2*, *3*, *6* e *9*.

Embora arriscando falso-positivos, filtros de Bloom têm uma grande vantagem em termos de uso de espaço com relação a outras estruturas de dados usadas para representar conjuntos, tal como árvores binárias de busca autoajustáveis, tabelas de dispersão ou simples arrays ou listas encadeadas. Muitas dessas estruturas requerem armazenamento dos dados, enquanto filtros de Bloom não os armazenam.

Diferente de uma tabela de dispersão padrão, um filtro de Bloom de tamanho fixo pode representar um conjunto com um número arbitrariamente grande de elementos; acrescentar um elemento nunca fracassa devido ao preenchimento da estrutura de dados. Entretanto a taxa de falsos positivos aumenta à medida que elementos são acrescentados até que todos os bits no filtro sejam iguais a 1, nesse ponto qualquer consulta apresenta um resultado positivo.

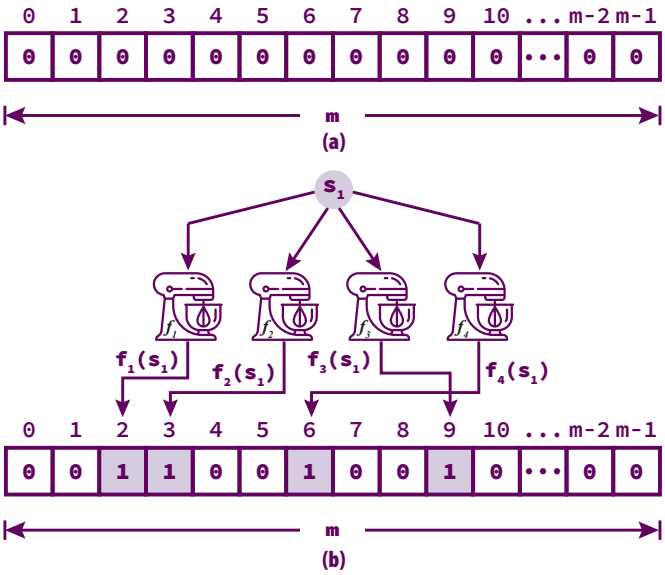

FIGURA 7–26: FILTRO DE BLOOM

Problema: Implemente um filtro de Bloom.

Solução: As seguintes definições de tipos serão usadas na implementação do filtro de Bloom:

```
   /* Tipo de ponteiro para uma função de dispersão */
typedef unsigned int (*tFDispersao)(const char *);

   /* Tipo de nó de lista encadeada que armazena */
   /* ponteiros para funções de dispersão        */
typedef struct rotNoListaBloom {
           tFDispersao             fDispersao;
           struct rotNoListaBloom *proximo;
        } tNoListaBloom, *tListaBloom;

   /* Tipo de estrutura que representa um filtro de Bloom */
typedef struct {
           tListaBloom listaDispersao;
           void       *bits;
           size_t      tamanho;
        } tFiltroBloom, *tFiltroBloomPtr;
```

A função `CriaFiltro()` a seguir cria um novo filtro de Bloom e seu único parâmetro é o tamanho do filtro em bytes. O retorno dessa função é o endereço do filtro criado.

```
tFiltroBloomPtr CriaFiltro(size_t tamanho)
{
   tFiltroBloomPtr filtroPtr = calloc(1, sizeof(tFiltroBloom));

      /* Garante que houve alocação */
   ASSEGURA(filtroPtr, "Alocação filtro fracassou");

   filtroPtr->tamanho = tamanho;
   filtroPtr->bits = calloc(1, tamanho);
   filtroPtr->listaDispersao = NULL;

      /* Garante que houve alocação do array de bits */
   ASSEGURA(filtroPtr->bits, "Alocação do array fracassou");

   return filtroPtr;
}
```

Na implementação completa do programa que você encontra no site dedicado ao livro na internet existe uma função, denominada `DestroiFiltro()`, que libera o espaço ocupado por um filtro de Bloom. A implementação dessa função é relativamente trivial.

A função `AcrescentaFuncaoAFiltro()` a seguir acrescenta uma função de dispersão ao filtro de Bloom. Os parâmetros da função `AcrescentaFuncaoAFiltro()` são:

- `filtro` (entrada/saída) — ponteiro para o filtro de Bloom
- `fDispersao` (entrada) — função de dispersão que será acrescentada

```
void AcrescentaFuncaoAFiltro(tFiltroBloomPtr filtro, tFDispersao fDispersao)
{
   tListaBloom pNo = malloc(sizeof(tListaBloom *));

   ASSEGURA(pNo, "Impossivel alocar no de lista encadeada");

      /* Armazena no novo nó o endereço da função recebido como parâmetro */
   pNo->fDispersao = fDispersao;

      /* O novo nó apontará para o início corrente da lista */
   pNo->proximo = filtro->listaDispersao;

      /* O início da lista passa a apontar para o novo nó */
   filtro->listaDispersao = pNo;
}
```

A função `AcrescentaItemAFiltro()` acrescenta um item a um filtro de Bloom e seus parâmetros são:

- `filtro` (entrada/saída) — ponteiro para o filtro de Bloom
- `item` (entrada) — item que será acrescentado

```
void AcrescentaItemAFiltro(tFiltroBloomPtr filtro, const void *item)
{
   tListaBloom    pNo; /* Apontará para um nó da lista de endereços de funções */
   unsigned char *byte; /* Apontará para cada byte do    */
                        /* array que representa o filtro */
   unsigned int   dispersao, /* Armazenará um valor de dispersão */
                  nBits; /* Número de bits do filtro */

   nBits = CHAR_BIT*filtro->tamanho; /* Calcula o número de bits do filtro */

      /* Faz 'byte' apontar para o primeiro byte  */
      /* do array que constitui o filtro de Bloom */
   byte = filtro->bits;

      /* Faz 'pNo' apontar para o primeiro nó da lista que */
      /* contém os endereços das funções de dispersão      */
   pNo = filtro->listaDispersao;

      /* Aplica cada função de dispersão cujo endereço se encontra na   */
      /* lista e liga o bit correspondente ao valor de dispersão obtido */
   while (pNo) {
         /* Obtém o valor de dispersão aplicando a função */
         /* cujo endereço se encontra no nó corrente      */
      dispersao = pNo->fDispersao(item);

         /* Assegura que o valor de dispersão obtido é */
         /* menor do que o número de bits do filtro    */
      dispersao %= nBits;
```

```
         /* Liga o bit correspondente ao valor de dispersão */
      byte[dispersao/CHAR_BIT] |= 1 << dispersao%CHAR_BIT;

      pNo = pNo->proximo; /* Passa para o próximo nó da lista */
   }
}
```

Para entender bem a instrução da função `AcrescentaFuncao()` que liga o bit desejado, consulte o **Apêndice B**.

A função `ItemEstaEmFiltro()` verifica se um item se encontra num filtro de Bloom. Os parâmetros dessa função são:

- `filtro` (entrada/saída) — ponteiro para o filtro de Bloom
- `item` (entrada) — item que será acrescentado

A função `ItemEstaEmFiltro()` retorna `0`, se o item nunca foi acrescentado ao filtro, ou `1`, se o item foi provavelmente acrescentado ao filtro.

```
int ItemEstaEmFiltro(tFiltroBloomPtr filtro, const void *item)
{
   tListaBloom    pNo; /* Apontará para um nó da lista de endereços de funções */
   unsigned char *byte; /* Apontará para cada byte do array que representa o filtro */
   unsigned int   dispersao, /* Armazenará um valor de dispersão */
                  nBits; /* Número de bits do filtro */

   nBits = CHAR_BIT*filtro->tamanho; /* Calcula o número de bits do filtro */

      /* Faz 'byte' apontar para o primeiro byte  */
      /* do array que constitui o filtro de Bloom */
   byte = filtro->bits;

      /* Faz 'pNo' apontar para o primeiro nó da lista que */
      /* contém os endereços das funções de dispersão      */
   pNo = filtro->listaDispersao;

      /* Aplica cada função de dispersão cujo endereço se encontra na lista e */
      /* verifica se o bit correspondente ao valor de dispersão obtido está   */
      /* ligado. O laço encerra quando é encontrado algum bit desligado ou    */
      /* quando todas as funções tiverem sido aplicadas sem que isso aconteça */
   while (pNo) {
         /* Obtém o valor de dispersão aplicando a função */
         /* cujo endereço se encontra no nó corrente      */
      dispersao = pNo->fDispersao(item);

         /* Assegura que o valor de dispersão obtido é */
         /* menor do que o número de bits do filtro    */
      dispersao %= nBits;

         /* Verifica se o bit correspondente ao valor de dispersão está ligado */
      if ( !(byte[dispersao/CHAR_BIT] & (1 << dispersao%CHAR_BIT)) )
         return 0; /* Item não se encontra no filtro */

      pNo = pNo->proximo;
   }
   return 1; /* Talvez o item faça parte do filtro */
}
```

Consulte o **Apêndice B** para entender a instrução da função `ItemEstaEmFiltro()` que verifica se um bit está ligado. O complemento do programa que implementa o filtro Bloom encontra-se no site dedicado ao livro na internet.

7.9 Exercícios de Revisão

Observação: Algumas questões propostas nesta seção requerem conhecimento prévio de programação de baixo nível em C. O **Apêndice B** apresenta o conhecimento mínimo necessário sobre esse assunto que permite o leitor responder essas questões.

Conceitos, Terminologia e Aplicações (Seção 7.1)

1. Descreva os seguintes conceitos:
   (a) Dispersão
   (b) Função de dispersão
   (c) Valor de dispersão
   (d) Tabela de dispersão
   (e) Coletor

2. Quais são os problemas centrais de dispersão?

3. Apresente exemplos de aplicação da técnica básica de dispersão.

4. (a) O que é uma tabela de dispersão diretamente endereçável? (b) Apresente um exemplo de tabela de dispersão diretamente endereçável.

5. (a) O que é colisão? (b) Cite duas estratégias comumente utilizadas para lidar com colisão.

6. Por que não é fácil encontrar a maior (ou menor) chave armazenada numa tabela de dispersão?

7. Por que não é fácil encontrar o piso (ou teto) de uma chave armazenada numa tabela de dispersão?

8. Por que uma tabela de dispersão não é uma boa escolha como tabela de busca para um programa que permite consultas de intervalo?

Funções de Dispersão (Seção 7.2)

9. Quais propriedades deve possuir uma boa função de dispersão?

10. (a) Descreva e apresente um exemplo de função de dispersão que utilize o método de divisão modular. (b) Por que divisão modular é usada com tanta frequência no cálculo de valores de dispersão?

11. O que é dispersão uniforme?

12. (a) Por que o método aditivo para funções de dispersão não é conveniente? (b) Qual tipo de chave deve ser usado com esse método?

13. Descreva e apresente um exemplo de função de dispersão que utilize o método de multiplicação.

14. O que é um método de dispersão genérico?

15. Descreva o método tabular para criação de funções de dispersão.

16. Descreva o método de dispersão *xor* para criação de funções de dispersão.

17. Em que difere o método de dispersão rotativa do método de dispersão *xor*?

18. (a) Descreva o método de dispersão por mistura. (b) Explique a construção da função de dispersão por mistura apresentada no **Apêndice B**.

19. Suponha que `x` seja um valor negativo na expressão `x % y`. (a) Qual é o valor dessa expressão em C se o compilador usado segue o padrão C99 ou o padrão C11? (b) E se o compilador segue um padrão ISO anterior a C99?

20. (a) Como uma função de dispersão deve ser testada? (b) Por que é importante testar uma função de dispersão?

21. (a) Por que todos os componentes relevantes de uma chave devem ser levados em consideração quando se cria uma função de dispersão? (b) Por que componentes irrelevantes ou redundantes de uma chave não devem ser levados em conta quando se cria uma função de dispersão?

22. Qual é a importância do método de Horner em dispersão?

23. Quando o programa a seguir é executado, ele apresenta dois valores de dispersão diferentes para duas estruturas com o mesmo conteúdo. (a) Explique por que isso acontece. (b) Mostre como corrigir esse problema.

```
#include <stdio.h>

typedef struct {
           char a;
           int  b;
        } tEstrutura;

unsigned G(tEstrutura *c)
{
   unsigned  i, dispersao = 0, m = 541,
             tam = sizeof(tEstrutura);
   char     *p = (char *) c;

   for (i = 0; i < tam; ++i)
      dispersao += *p++ % m;

   return dispersao;
}

int main(void)
{
   tEstrutura e1 = {'a', 10}, e2 = {'a', 10};

   printf("\nDispersao de e1: %d", G(&e1));
   printf("\nDispersao de e2: %d\n", G(&e2));

   return 0;
}
```

24. Qual é o problema com a seguinte função de dispersão?

```
unsigned F(const char *c)
{
   return 10;
}
```

Observação: As questões a seguir (até a questão **44**) requerem conhecimento de programação de baixo nível em C (consulte o **Apêndice B**).

25. O que é manipulação de bits?

26. (a) Qual é o propósito do operador de complemento? (b) Quais são os tipos de operandos que podem ser utilizados com esse operador?

27. (a) Descreva os três operadores lógicos binários que atuam sobre bits. (b) Quais são os tipos de operandos que podem ser utilizados com esses operadores?

28. Supondo que `x`, `y` e `z` são variáveis de algum tipo inteiro, a expressão a seguir realiza uma operação bem comum em programação, mas de modo mais eficiente do que por meio de métodos convencionais. Qual é essa operação?

```
z = y + ((x - y) & -(x < y));
```

29. Suponha que `x`, `y` e `z` sejam variáveis do tipo **unsigned char** e que `n` seja uma constante tal que `1` ≤ `n` ≤ **CHAR_BIT**, em que **CHAR_BIT** é número de bits utilizados para representar um valor do tipo **char**. Quais serão os efeitos colaterais sobre `x`, `y` e `z` resultantes das seguintes operações:

(a) `x |= (1 << n)`
(b) `y &= ~(1 << n)`
(c) `z ^= (1 << n)`

30. (a) O que é uma operação de mascaramento? (b) Qual é o propósito de cada operando numa operação desse tipo?

31. (a) O que é uma máscara numa operação de mascaramento? (b) Como uma máscara é escolhida?

32. Calcule o valor de `~9430₁₀` e escreva o resultado em hexadecimal. (Note que `9430₁₀` é um número em formato decimal.)

33. Escreva uma operação de mascaramento independente de implementação para copiar os seis bits mais à direita de um valor do tipo **int** para uma outra variável desse tipo, com cada bit restante mais à esquerda sendo preenchido com `1`.

34. (a) Descreva, utilizando diagramas, uma operação de mascaramento na qual uma porção de um dado padrão de bits é copiada enquanto os bits restantes são todos igualados a zero. (b) Qual operação sobre bits é utilizada nessa operação? (c) Como a máscara é selecionada?

35. (a) Descreva uma operação de mascaramento na qual uma porção de um dado padrão de bits é copiada enquanto os bits restantes são todos igualados a *1*. (b) Qual operação sobre bits é utilizada nessa operação? (c) Como a máscara é selecionada? (d) Compare este exercício com o exercício anterior.

36. (a) Descreva uma operação de mascaramento na qual uma porção de um dado padrão de bits é copiada enquanto os bits restantes são todos invertidos. (b) Qual operação sobre bits é utilizada nessa operação? (c) Como a máscara é selecionada? (d) Compare este exercício com os dois últimos exercícios.

37. Em que situações o uso do operador complemento de um é recomendável em operações de mascaramento?

38. (a) Como um determinado bit de uma variável inteira pode passar de `0` a `1` ou vice-versa? (b) Qual operador lógico sobre bits é utilizado para esse propósito?

39. (a) Descreva as duas operações de deslocamento sobre bits. (b) Qual é o papel de cada operando? (c) Quais requisitos esses operandos devem satisfazer?

40. Compiladores de C nem sempre tratam operações de deslocamento à direita da mesma maneira. Em que situações as operações de deslocamento à direita não são portáveis?

41. (a) Qual seria o nome mais apropriado para a função `F()` a seguir? (b) Qual seria o nome mais apropriado para a variável local `retorno` desta função?

```
unsigned F(int x)
{
   unsigned retorno = 0;

   while (x) {
      if (x & 0x1)
         retorno++;
      x >>= 1;
   }

   return retorno;
}
```

42. Demonstre que as seguintes propriedades da operação $\oplus$ (*xor*) são válidas:

(a) Para qualquer inteiro x, $x \oplus x = 0$

(b) Para quaisquer inteiros x e y, $(x \oplus y) \oplus y = x$

(c) Para quaisquer inteiros x e y, $x \oplus y = y \oplus x$

43. O que as funções `Misterio1()` e `Misterio2()`, definidas a seguir, calculam?

```
unsigned Misterio1(unsigned a, unsigned b)
{
   unsigned x = 0,
            y = 0,
            z = ~0;

   for (z = ~0; z; z >>= 1) {
      y <<= 1;
      y |= (a^b^x) & 1;
      x = ((a | b) & x | a & b) & 1;
      a >>= 1;
      b >>= 1;
   }

   for (z = ~0, x = ~z; z; z >>= 1) {
      x <<= 1;
      x |= y & 1;
      y >>= 1;
   }

   return x;
}

unsigned Misterio2(unsigned a, unsigned b)
{
   unsigned resultado;

   for (resultado = 0; a; b <<= 1, a >>= 1)
      if (a&1)
         resultado = Misterio1(resultado, b);

   return resultado;
}
```

44. Por que o programa a seguir é executado indefinidamente?

```
#include <stdio.h>

int main(void)
{
   int  i;
   char c;

   for (i = 0x80; i != 0; i = i >> 1)
      printf("i = %x (%d)\n", i, i);

   for (c = 0x80; c != 0; c = c >> 1)
      printf("c =  %x (%d)\n", c, c);

   return 0;
}
```

Dispersão com Encadeamento (Seção 7.3)

45. O que é dispersão com encadeamento?

46. Quando ocorre uma colisão em dispersão com encadeamento, como ela é resolvida?

47. A ordem de inserção de chaves numa tabela de dispersão com encadeamento influencia o posicionamento delas na tabela? Explique seu raciocínio.

48. Descreva cada um dos seguintes algoritmos para tabelas de dispersão com encadeamento:

(a) Algoritmo de busca

(b) Algoritmo de inserção
(c) Algoritmo de remoção

49. (a) O que é fator de carga? (b) Qual é o fator de carga recomendável para dispersão com encadeamento?

50. Na implementação de tabelas de dispersão encadeadas apresentada neste capítulo, o valor real de dispersão usado para acessar a tabela é o valor de dispersão módulo o tamanho da tabela. Por que é assim?

51. Supondo uma distribuição uniforme de chaves, se houver n chaves numa tabela de dispersão com encadeamento e uma média de M chaves em cada lista, quanto tempo leva em média para encontrar uma determinada chave?

52. Em dispersão com encadeamento, o que é mais rápida: uma busca bem-sucedida ou uma busca malsucedida? Explique seu raciocínio.

53. (a) Em dispersão com encadeamento, é possível que o fator de carga seja menor do que *1*? (b) É possível que ele seja maior do que *1*?

54. Mostre como são inseridas as chaves *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J* e *K* numa tabela de dispersão com encadeamento contendo $m = 7$ listas e cuja função de dispersão seja $f(c) = 13 \cdot c \bmod m$. Suponha que o valor de *A* seja *1*, o valor de *B* seja *2* e assim por diante.

55. Apresente uma representação gráfica da tabela de dispersão com encadeamento resultante da inserção das chaves *13*, *41*, *12*, *79*, *21*, *89*, *11*, *37*, *33*, *19*, *16*, *44* e *3*, supondo que a função de dispersão é $f(c) = (2 \cdot c + 5) \bmod 13$ e que a tabela contém *13* coletores (listas).

56. Dadas as chaves *3461*, *1733*, *4163*, *2189*, *2324*, *9359* e *1999* e a função de dispersão $f(c) = c \bmod 5$, apresente graficamente a tabela de dispersão com encadeamento que contém essas chaves supondo que $m = 5$, em que m é o número de listas da tabela.

57. (a) Qual é o custo temporal no pior caso de busca numa tabela de dispersão com encadeamento? (b) Como se assegura que esse caso não irá ocorrer?

Dispersão com Endereçamento Aberto (Seção 7.4)

58. O que é dispersão com endereçamento aberto?

59. Descreva os seguintes conceitos:
(a) Sondagem
(b) Sequência de sondagem
(c) Passo de sondagem
(d) Índice de sondagem

60. Qual é a diferença entre busca e sondagem?

61. Por que não existem sondagens em operações básicas com tabelas de dispersão com encadeamento?

62. A ordem de inserção de chaves numa tabela de dispersão com endereçamento aberto influencia o posicionamento delas na tabela? Explique seu raciocínio.

63. (a) O que é uma função de dispersão primária? (b) O que é uma função de sondagem ou função de resolução de colisões?

64. (a) Qual é a vantagem do uso de dispersão com endereçamento aberto com relação a dispersão com encadeamento? (b) Qual é a desvantagem do uso de dispersão com endereçamento aberto com relação a dispersão com encadeamento?

65. (a) O que é agrupamento primário? (b) Por que agrupamento primário deve ser evitado?

66. Descreva as seguintes técnicas de resolução de colisões:
(a) Sondagem linear

(b) Sondagem quadrática

(c) Sondagem com dispersão dupla

67. (a) O que determina o tamanho do passo de sondagem na sondagem quadrática? (b) O que determina o tamanho do passo de sondagem na sondagem dupla?

68. Qual é a desvantagem de sondagem quadrática com relação a sondagem dupla?

69. Por que o tamanho da tabela deve ser um número primo quando se usa sondagem dupla?

70. O que é agrupamento secundário?

71. (a) Por que quando se usa sondagem quadrática o tamanho da tabela de dispersão deve ser um número primo? (b) Por que esse requisito não é necessário quando se usa sondagem linear?

72. (a) Qual é a desvantagem de sondagem quadrática com relação a dispersão dupla? (b) Por que se utiliza sondagem quadrática se ela é pior do que dispersão dupla?

73. Quais são os critérios que deve satisfazer uma função de dispersão secundária usada em dispersão dupla?

74. Por que uma função de dispersão secundária usada em dispersão dupla não pode resultar em zero?

75. Quais são as principais desvantagens de todas as técnicas de resolução de colisões usadas em endereçamento aberto?

Para os **Exercícios 76–79**, utilize as seguintes chaves:

34	*52*	*76*	*92*	*118*	*135*	*140*	*151*
166	*244*	*371*	*386*	*444*	*555*	*660*	*777*

76. Mostre como são armazenadas essas chaves numa tabela de dispersão com *20* elementos, usando o método de divisão modular de dispersão e o método de sondagem linear para resolução de colisões.

77. Mostre como são armazenadas essas chaves numa tabela de dispersão com *20* elementos, usando dispersão dupla como método de resolução de colisão. Use *c mod m* como função de dispersão e *(c + 3) mod m* como função de dispersão secundária.

78. Mostre como essas chaves são armazenadas numa tabela de dispersão com encadeamento que usa a função de dispersão *c mod 10* para determinar em qual das dez listas encadeadas a chave deve ser armazenada.

79. Preencha a seguinte tabela mostrando o número de comparações necessárias para encontrar cada chave usando os esquemas de dispersão dos **Exercícios 76–78**.

Chave	Exercício 76	Exercício 77	Exercícios 78
34			
444			
555			
777			
244			
76			

80. Mostre como são inseridas as chaves *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J* e *K* numa tabela de dispersão com endereçamento aberto e sondagem linear com $m = 19$ e cuja função de dispersão seja $f(c) = 13 \cdot c \bmod m$. Suponha que o valor de *A* seja *1*, o valor de *B* seja *2* e assim por diante.

81. Dadas as chaves *3461*, *1733*, *4163*, *2189*, *2324*, *9359* e *1999* e a função de dispersão $f(c) = c \bmod 10$, apresente a tabela de dispersão com endereçamento aberto supondo que $m = 10$ e que o método de resolução de colisões é:

(a) Sondagem linear

(b) Sondagem quadrática

(c) Dispersão dupla com $f_2(c) = 7 - (c \bmod 7)$

82. Classifique cada uma das seguintes afirmações como verdadeira (**V**) ou falsa (**F**):

(a) Aumentar o tamanho de uma tabela de dispersão sempre reduz o número de colisões.

(b) Quando se usa dispersão com endereçamento aberto, não é preciso preocupar-se com resolução de colisão.

(c) Quando se usa dispersão com encadeamento, não é preciso preocupar-se com resolução de colisão.

(d) O objetivo de um esquema de dispersão bem-sucedido é busca com custo temporal $\theta(1)$.

(e) A eficiência de dispersão com encadeamento degrada rapidamente à medida que o fator de carga se aproxima de *1*.

83. Uma tabela de dispersão com endereçamento aberto pode apresentar um fator de carga maior do que *1*? Explique.

84. Apresente uma representação gráfica da tabela de dispersão com endereçamento aberto resultante da inserção das chaves *13*, *41*, *12*, *79*, *21*, *89*, *11*, *37*, *33*, *19*, *16*, *44* e *3*, supondo que a função de dispersão é $f(c) = (2 \cdot c + 5) \bmod 13$ e que as colisões são resolvidas por sondagem linear numa tabela com *13* elementos.

85. Resolva o **Exercício 84** supondo que as colisões são resolvidas por meio de sondagem quadrática.

86. Resolva o **Exercício 84** supondo que as colisões são resolvidas por meio de dispersão dupla com a função de dispersão secundária sendo $f'(c) = 7 - (c \bmod 7)$.

87. Por que na implementação de uma função de redimensionamento de dispersão em C não se deve usar **realloc()**?

88. Qual é a vantagem de usar ponteiros para funções como parâmetros de funções que implementam operações de busca, inserção e remoção?

89. (a) Qual é o papel desempenhado pelo tipo enumeração `tStatusDEA` na implementação de tabela de dispersão com endereçamento aberto apresentada na **Seção 7.4.4**? (b) Por que esse tipo de enumeração não é usado na implementação de tabelas de dispersão com encadeamento?

90. (a) Qual é o desempenho no pior caso de busca por uma chave numa tabela de dispersão com endereçamento aberto? (b) Como assegurar que esse caso não irá ocorrer?

91. Considerando uma tabela de dispersão com endereçamento aberto com m coletores, quantas sequências de sondagem existem quando a sondagem é: (a) linear, (b) quadrática e (c) com dispersão dupla?

92. (a) Qual método de sondagem apresenta a melhor localidade de referência? (b) Qual método de sondagem apresenta a pior localidade de referência?

Redimensionamento de Dispersão (Seção 7.5)

93. O que é redimensionamento de dispersão?

94. Apresente uma análise amortizada formal usando o método de potencial para mostrar que o custo amortizado de uma inserção numa tabela de dispersão com endereçamento aberto é $\theta(1)$, mesmo considerando redimensionamento.

Dispersão Cuco (Seção 7.6)

95. (a) O que é dispersão cuco? (b) Por que dispersão cuco recebe essa denominação?

96. Descreva o mecanismo de inserção usado por uma tabela de dispersão cuco.

97. Em que situação pode ocorrer um ciclo infinito durante uma inserção numa tabela de dispersão cuco?

98. Quando as tabelas usadas em dispersão cuco precisam ser redimensionadas?

99. Por que, numa operação de remoção numa tabela de dispersão cuco, o elemento removido não precisa ser marcado como *removido*?

100. Qual é o custo temporal de cada uma das seguintes operações em tabelas de dispersão cuco?
(a) Busca
(b) Inserção
(c) Remoção

101. Compare dispersão cuco com outros esquemas de dispersão apresentados neste capítulo em termos de vantagens e desvantagens.

Análise de Técnicas de Dispersão (Seção 7.7)

102. Quais são as vantagens e desvantagens das abordagens de resolução de colisões discutidas neste capítulo?

103. Por que tabelas de dispersão são boas para acesso direto, mas não o são para acesso sequencial?

104. (a) Qual é o desempenho no pior caso de busca por uma chave numa tabela de dispersão encadeada? (b) Como se assegura que esse caso não irá ocorrer?

105. (a) Em geral, qual é a principal vantagem do uso de dispersão com relação a outros métodos de implementação de tabelas de busca? (b) Quando essa aparente vantagem não se concretiza?

106. Quais são as desvantagens do uso de dispersão com relação a outras abordagens de implementação de tabelas de busca em memória principal apresentadas neste livro?

107. Suponha que você tenha uma função de dispersão que se tenha assegurado que produz uma dispersão uniforme para o conjunto esperado de chaves. Como você escolheria a implementação mais adequada para sua tabela de dispersão? Justifique sua escolha.

108. Quando o número de chaves que serão inseridas numa tabela de dispersão é desconhecido, qual é a melhor abordagem a ser adotada para a tabela? Justifique sua resposta.

109. O que é mais rápido em dispersão com encadeamento: uma busca bem-sucedida ou uma busca malsucedida? Explique sua resposta.

110. Compare implementação de tabela de busca usando dispersão com implementação de tabela de busca usando árvore binária.

Exemplos de Programação (Seção 7.8)

111. Como se testa uma função de dispersão para verificar se ela produz uma distribuição uniforme?

112. (a) O que é filtro de Bloom? (b) Descreva seu funcionamento.

113. O que é uma consulta de pertinência?

114. Por que remoção não é permitida num filtro de Bloom convencional?

115. (a) O que é falso-positivo? (b) O que é falso-negativo?

116. O que é pior para um programa antivírus: um erro falso-positivo ou um erro falso-negativo?

117. Num exame médico, o que é pior: um erro falso-positivo ou um erro falso-negativo?

118. Você tenta entrar num banco e a porta giratória o impede devido ao fato de você portar um chaveiro no bolso. Esse alarme é falso-positivo ou falso-negativo?

119. (a) Como se liga um bit? (b) Como se verifica se um bit está ligado?

7.10 Exercícios de Programação

Observação: Alguns exercícios de programação propostos nesta seção requerem conhecimento prévio de programação de baixo nível em C. O **Apêndice B** apresenta o conhecimento mínimo necessário sobre o assunto.

EP7.1 Generalize a função `RepresentacaoBinaria()` apresentada no **Apêndice B**, de modo que a nova versão possa exibir a representação binária de um parâmetro de qualquer tipo. Essa nova versão da função `RepresentacaoBinaria()` deve ter como protótipo:

```
void  RepresentacaoBinaria2(const void *ptrValor, size_t tamanhoDoValor)
```

EP7.2 (a) Escreva uma função, denominada `InverteBits()`, que inverte os bits de um parâmetro do tipo **unsigned int**. (b) Escreva um programa que recebe como entrada um valor do tipo **unsigned int**, converte-o utilizando a função `InverteBits()` e apresenta o valor original recebido como entrada e o valor alterado pela função `InverteBits()` em formato binário.

EP7.3 Na implementação de dispersão com encadeamento, assumiu-se que cada elemento da tabela de busca era um ponteiro para uma lista encadeada, o que desperdiça espaço no caso em que uma lista armazena apenas uma chave. Apresente uma implementação alternativa de dispersão com encadeamento na qual não ocorra esse tipo de desperdício.

EP7.4 Implemente um algoritmo de detecção de ciclo no esquema de dispersão cuco sem ser por meio de contagem de repetições, como foi efetuado na **Seção 7.6.3**.

EP7.5 Escreva um programa que calcula o número de colisões numa longa sequência aleatória de inserções usando sondagem linear, sondagem quadrática e dispersão dupla.

EP7.6 Numa variante de dispersão cuco, a tabela de dispersão é dividida em duas tabelas menores de tamanhos iguais e cada função de dispersão provê um índice para uma dessas duas metades. Implemente essa variante de dispersão cuco.