



FUNÇÕES DE DISPERSÃO PRONTAS

ELABORAR UMA FUNÇÃO DE DISPERSÃO é, em boa proporção, uma arte. Assim é sempre melhor usar uma função de dispersão reconhecidamente boa (i.e., que tenha sido exaustivamente testada e aprovada) do que tentar inventar uma. Há muitos algoritmos de dispersão bons disponíveis que passaram por análise rigorosa e têm excelentes distribuições para quase todos os tipos de chaves. Algumas são extremamente simples. Todas elas quase certamente superam qualquer função que você (ou o próprio autor) pode desenvolver em termos de distribuição uniforme. Muitas delas são executadas bem mais rapidamente do que uma solução amadora.

Frequentemente, encontram-se funções de dispersão que afirmam que são *a melhor para strings* ou *a melhor para inteiros*. Essas funções devem ser evitadas porque, se uma função de dispersão não é boa para todos os tipos de dados, provavelmente, ela representa um algoritmo ruim. Algumas vezes, por outro lado, uma função de dispersão pode ser otimizada para um único tipo por questão de eficiência. É bom aprender a discernir essas nuances, mas uma prática segura é usar apenas funções gerais de dispersão que são reconhecidamente boas.

Como a cuidadosa elaboração de uma função de dispersão não é uma tarefa fácil para muitos programadores, o melhor conselho é usar funções consagradas que já tenham sido exaustivamente estudadas. Este apêndice irá descrever várias boas funções de dispersão, de modo que o programador não sucumba à tentação de escrever um algoritmo *ad hoc* quando uma boa função de dispersão for necessária.

C.1 JOAAT

Uma das funções de dispersão favoritas dos especialistas com relação a rapidez, simplicidade e qualidade é a função **JOAAT** de Robert Jenkins^[1]. A função de dispersão de Jenkins excede as sugestões apresentadas no

[1] A denominação **JOAAT** é derivada das letras iniciais de *Jenkins one-at-a-time*, na qual Jenkins é o autor da função e o restante refere-se ao fato de um byte ser processado a cada iteração.

Capítulo 7 é baseada em análise com respeito a colisões. O que há de ruim sobre a função de Jenkins é que o algoritmo é confuso e usa um grande número de constantes misteriosas difíceis de entender. Jenkins é uma conhecida autoridade no projeto de funções de dispersão para tabela de busca. Uma implementação da função **JOAAT** é apresentada a seguir.

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

    while (*chave) {
        dispersao += *chave++;
        dispersao += (dispersao << 10);
        dispersao ^= (dispersao >> 6);
    }
    dispersao += (dispersao << 3);
    dispersao ^= (dispersao >> 11);
    dispersao += (dispersao << 15);

    return dispersao;
}
```

A função **DispersaoJOAAT()** é uma boa escolha como uma primeira tentativa de função de dispersão, pois ela atinge avalanche (v. **Seção 7.2.1**) rapidamente e funciona muito bem. Por conseguinte essa função deve ser uma das primeiras a ser testada em qualquer implementação de tabela de dispersão.

C.2 DJB e DJB2

A função **DJB** criada por Daniel Julius Bernstein é bastante utilizada na prática com sucesso. Mas, apesar disso, o algoritmo em si não é muito confiável com relação a avalanche. Essa função tem provado ser muito boa para chaves pequenas constituídas por caracteres. Em tal caso, ela pode superar algoritmos que resultam numa distribuição mais aleatória. A função **DispersaoDJB()** abaixo é uma implementação da função **DJB**.

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

    while (*chave) dispersao = 33*dispersao + *chave++;

    return dispersao;
}
```

A função **DJB** deve ser usada com cautela, pois, apesar de ela funcionar muito bem na prática, teoricamente, ela não é razoável. O papel desempenhado pela misteriosa constante 33 e a razão pela qual ela funciona melhor do que outras constantes são questões ainda não respondidas. Sempre teste essa função com amostras de dados para cada uso para assegurar que ela não causa colisões excessivas.

Substituindo-se soma por **XOR** na instrução que efetua a combinação de bytes (i.e., o corpo do laço), obtém-se a função **DJB2** mostrada abaixo. Essa nova função parece não ser tão conhecida ou frequentemente usada quanto a função **DJB** original, mas, aparentemente, resulta numa melhor distribuição de chaves.

```
unsigned int DispersaoDJB2(const char *chave)
{
    unsigned int dispersao = 0;

    while (*chave)
        dispersao = 33*dispersao ^ *chave++;

    return dispersao;
}
```

C.3 SAX

A função de dispersão **SAX**^[2] foi criada como uma função de dispersão para strings, mas ela também funciona para qualquer tipo de dado com a mesma eficiência. O algoritmo seguido por essa função é similar àquele de dispersão rotativa apresentado na **Seção 7.2.3**, mas a função **SAX** usa constantes diferentes para rotação e a operação usada para a mistura é soma. Essa função é muito poderosa e flexível, mas, como a maioria das funções apresentadas aqui, ela não é aprovada em testes para avalanche, mesmo que isso não pareça afetar sua eficiência na prática.

```
unsigned int DispersaoSAX(const char *chave)
{
    unsigned int dispersao = 0;
    while (*chave) dispersao ^= (dispersao << 5) + (dispersao >> 2) + *chave++;
    return dispersao;
}
```

C.4 FNV

A função de dispersão **FNV**, cuja denominação é derivada das iniciais de Fowler, Noll e Vo, em homenagem a seus criadores, é um algoritmo poderoso que segue a mesma linha que as funções de dispersão de Bernstein (v. **Seção C.2**) com constantes cuidadosamente escolhidas. Esse algoritmo tem sido usado em muitas aplicações práticas com excelentes resultados e, por sua simplicidade, a função de dispersão **FNV** deve ser uma das primeiras a ser experimentadas numa implementação de tabela de dispersão.

```
unsigned int DispersaoFNV(const char *chave)
{
    unsigned int dispersao = 2166136261;
    while (*chave) dispersao = (dispersao*16777619) ^ *chave++;
    return dispersao;
}
```

Uma versão mais sofisticada da função **FNV**, denominada **FNV-1A**, pode ser facilmente encontrada na internet.

C.5 JSW

A função de dispersão **JSW**, assim denominada em homenagem a seu criador — Julianne S. Walker — combina dispersão rotativa com dispersão tabular, na qual uma tabela de números aleatórios é utilizada. O algoritmo acessa cada byte da chave de entrada e usa-o como um índice da tabela de números aleatórios. Os bits de uma chave são deslocados à esquerda e à direita, então aplica-se **XOR** com o número aleatório obtido dessa tabela, cujo tamanho deve ser igual ao número de possíveis valores inteiros não negativos de um byte. Por exemplo, se um byte contiver 8 bits (como ocorre normalmente nos dias atuais), a tabela conterá 256 (i.e., 2⁸) números aleatórios. O resultado obtido é uma distribuição uniforme se os números aleatórios utilizados na construção da tabela também forem uniformemente distribuídos.

O programa apresentado a seguir ilustra o uso de uma função de dispersão **JSW**.

```
#include <stdio.h>    /* printf() */
#include <stdlib.h>    /* srand() e rand() */
#include <time.h>      /* time() */
#include <limits.h>    /* UCHAR_MAX */
```

[2] O nome **sax** é derivado das letras iniciais de *Shift*, *Add* e *Xor*, que representam as operações sobre as quais essa função é baseada: deslocamento (*shift*, em inglês), soma (*add*) e disjunção exclusiva sobre bits (**xor**).

```
static void CriaTabela(int tab[], int tam)
{
    int i;
    srand((unsigned)time(NULL)); /* Inicia o gerador de números aleatórios */
    for (i = 0; i < tam; ++i) /* Cria a tabela de valores aleatórios */
        tab[i] = rand();
}

unsigned int DispersaoJSW(const char *chave)
{
    unsigned int dispersao = 16777551; /* Número mágico */
    static int primeiraChamada = 1, tabela[ UCHAR_MAX + 1];

    if (primeiraChamada) {
        CriaTabela(tabela, UCHAR_MAX + 1);
        primeiraChamada = 0;
    }

    while (*chave)
        dispersao = (dispersao << 1 | dispersao >> 31) ^ tabela[*chave++];

    return dispersao;
}

int main(void)
{
    printf( "\n >>> Usando Dispersao DispersaoJSW <<<<\n");
    printf( "\n>>> Dispersao de \"bola\" = %u", DispersaoJSW("bola") );
    printf( "\n>>> Dispersao de \"loba\" = %u\n", DispersaoJSW("loba") );
    printf( "\n>>> Dispersao de \"bola\" = %u", DispersaoJSW("bola") );
    printf( "\n>>> Dispersao de \"loba\" = %u\n", DispersaoJSW("loba") );

    return 0;
}
```

Uma execução desse último programa produz como resultado:

```
>>>> Usando Dispersao DispersaoJSW <<<<
>>> Dispersao de "bola" = 268690024
>>> Dispersao de "loba" = 268643032

>>> Dispersao de "bola" = 268690024
>>> Dispersao de "loba" = 268643032
```

Como a função **DispersaoJSW()** usa uma tabela de números aleatórios que são diferentes a cada execução do programa, obviamente, os valores de dispersão produzidos serão diferentes a cada execução desse programa. Note, entretanto que, apesar de parecer o contrário, essa função é bem determinística no sentido de que ela produz sempre o mesmo valor de dispersão para uma dada chave em cada execução do programa que a utiliza, como você pode conferir no exemplo de execução acima. O problema com essa função é que ela não pode ser utilizada para armazenamento persistente de tabelas de dispersão.

C.6 Breve Avaliação das Funções de Dispersão Prontas

De acordo com o que foi discutido na **Seção 7.2.1**, as principais propriedades que uma função de dispersão devem satisfazer para que seja considerada adequada são:

- ❑ **Apresentar baixo custo computacional.** Todas as funções de dispersão prontas discutidas neste apêndice apresentam custo temporal $\theta(k)$, em que k é o tamanho (i.e., número de caracteres) da chave. Além disso, todas elas, com exceção da função de dispersão **JSW**, apresentam custo espacial $\theta(1)$. A função de dispersão **JSW** tem custo $\theta(|\Sigma|)$, em que $|\Sigma|$ é o tamanho do alfabeto do qual se derivam as chaves, mas, mesmo assim, esse custo é irrelevante, visto que ele é decorrente do uso de uma tabela de números aleatórios que é criada uma única vez para um dado conjunto de chaves.
- ❑ **Ser determinística.** Todas as funções de dispersão discutidas neste apêndice são determinísticas. A única função discutida naquela seção que aparenta não ser determinística é a função **JSW**, pois essa função usa uma tabela de valores aleatórios.
- ❑ **Lidar com normalização de dados.** Nenhuma das funções discutidas neste apêndice trata de normalização de dados. Nem poderia, pois essa é uma propriedade característica de cada conjunto de chaves, que, evidentemente, não é de conhecimento dos criadores dessas funções que, nesse aspecto, são genéricas.
- ❑ **Apresentar avalanche.** O atendimento a essa propriedade não tem grande importância prática para o uso de funções de dispersão na implementação de tabelas de busca. Ou seja, essa propriedade reveste-se de maior importância quando uma função de dispersão é usado em segurança de dados (p. ex., criptografia).
- ❑ **Apresentar boa distribuição.** Uma das afirmações mais categóricas dos autores dessas funções é elas distribuem chaves uniformemente. Essa propriedade, entretanto, precisa ser testada com cada conjunto particular de chaves.

