APÊNDICE B
FUNÇÕES DE DISPERSÃO PRONTAS

Elaborar uma função de dispersão é 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, então é provavelmente um algoritmo ruim em geral. Algumas vezes, por outro lado, a função de dispersão pode ser otimizada para um único tipo por questão de eficiência. É bom aprender a discernir as duas, mas uma prática segura é apenas usar funções gerais de dispersão que são reconhecidamente boas.

Uma vez que cuidadosa elaboração de funções de dispersão não é algo que muitos programadores não sabem como fazer, o melhor conselho é usar funções consagradas por que já foram exaustivamente estudadas. Este apêndice irá descrever várias boas funções de dispersão de modo que se pode evitar a tentação de escrever um algoritmo *ad hoc* quando for necessário.

B.1 Função de Dispersão JOAAT (One-at-a-Time)

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 **Capítulo 8** e é baseada na melhor 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);
```

[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.

```
      dispersao ^= (dispersao >> 6);
   }

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

   return dispersao;
}
```

A função `DispersaoJOAAT()` é uma boa escolha como um primeiro teste de função de dispersão, pois ela atinge avalanche rapidamente e funciona muito bem. Por consequência essa função deve ser uma das primeiras a ser testada em qualquer implementação de tabela de busca.

B.2 Funções de Dispersão de Bernstein (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 mostrado 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. Quer dizer, ela funciona muito bem na prática, por uma razão aparentemente desconhecida, mas, em teoria, 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.

Uma pequena alteração na função **DJB**, substituindo soma por *xor* na instrução que efetua a combinação, dá origem à função **DJB2** mostrada adiante. Essa nova função não parece ser tão conhecida ou frequentemente usada quanto a função **DJB** original, mas o novo algoritmo tipicamente resulta numa melhor distribuição de chaves.

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

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

   return dispersao;
}
```

B.3 Função de Dispersão SAX (Shift–Add–Xor)

A função de dispersão **SAX**[2] foi criada como uma função de dispersão para strings, mas ela também funciona para qualquer tipo de dados com eficiência similar. O algoritmo seguido por essa função é similar àquele de dispersão rotativa apresentado na **Seção 8.2.3**, mas a função **SAX** usa constantes diferentes para a 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;
}
```

B.4 Função de Dispersão de Fowler, Noll e Vo (FNV)

A função de dispersão **FNV**, cuja denominação é derivada das inicias 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 com constantes cuidadosamente escolhidas. Esse algoritmo tem sido usado em muitas aplicações 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 busca.

```
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.

B.5 Função de Dispersão de Julienne Walker (JSW)

A função de dispersão **JSW**, assim denominada em homenagem a seu criador, 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 da referida tabela. O tamanho

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

dessa tabela deve ser igual ao número de possíveis valores inteiros não negativos de um byte. Por exemplo, se um byte contém *8* bits, a tabela conterá *256* (i.e., 2^8) 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        */

static void CriaTabela(int tab[], int tam)
{
   int i;

      /* Inicia o gerador de números aleatórios */
   srand((unsigned)time(NULL));

      /* Cria a tabela de valores aleatórios */
   for (i = 0; i < tam; ++i) {
      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.

B.6 Breve Avaliação das Funções de Dispersão Prontas

De acordo com o que foi discutido na **Seção 8.2.1**, as 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 $O(k)$, 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 $O(1)$. A função de dispersão JSW tem custo $O(|\sum|)$, em que $|\sum|$ 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. Obviamente, todas 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. A importância maior dessa propriedade é em segurança???
- Apresentar boa distribuição. Essa é uma das afirmações mais categóricas dos autores dessas funções, mas que precisa ser testada com cada conjunto de chaves particular.