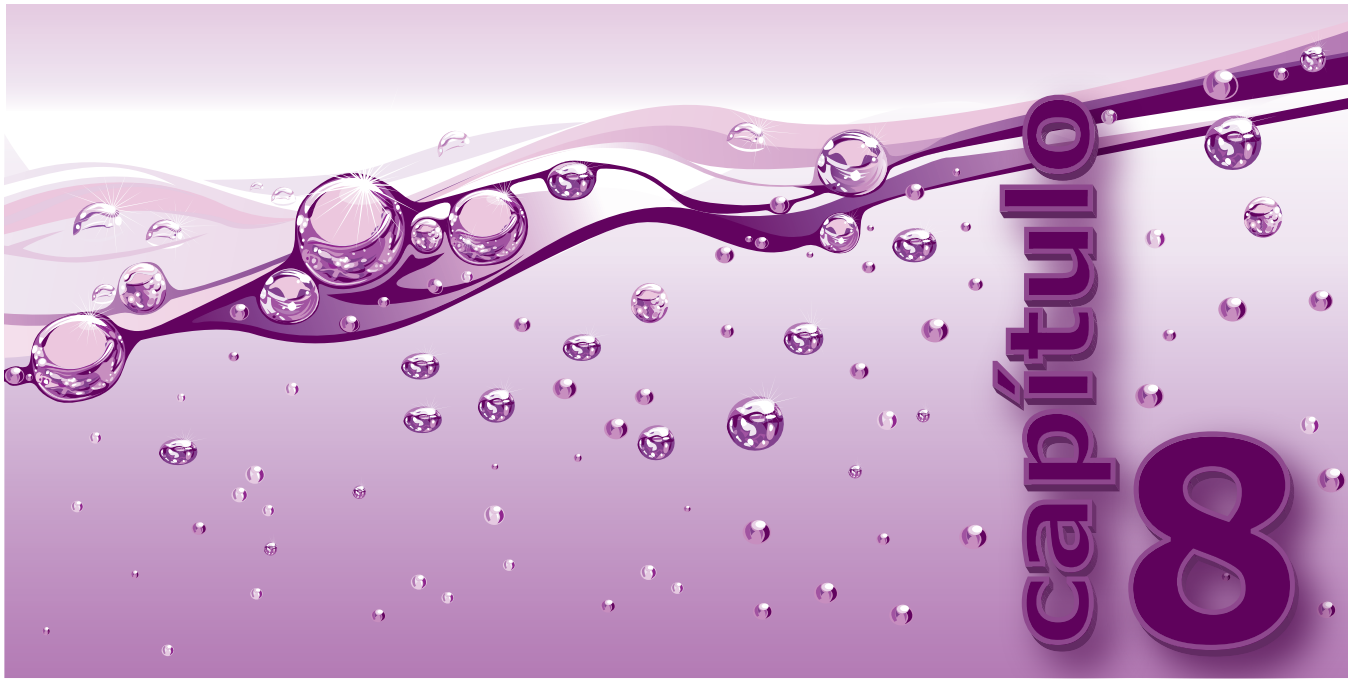


Dispersão em Memória Secundária

objetivos

Após estudar este capítulo, você deverá ser capaz de:

- Descrever os seguintes conceitos:
  - ❒ Coletor primário
  - ❒ Coletor excedente
  - ❒ Coletor camarada
  - ❒ Dispersão estática
  - ❒ Dispersão extensível
  - ❒ Profundidade global
  - ❒ Profundidade local
  - ❒ Diretório de dispersão extensível
- Especificar e implementar operações de busca, inserção e remoção em tabela de dispersão estática
- Comparar dispersão em memória principal com dispersão em memória secundária
- Discutir as limitações das técnicas de dispersão em memória secundária
- Extrair bits mais significativos ou menos significativos de uma chave de busca
- Mostrar como são efetuadas operações de busca, inserção e remoção em tabela de dispersão extensível
- Contrastar técnicas de dispersão em memória secundária com o uso de árvores da família B

DO MESMO MODO que não é conveniente o uso de listas indexadas ou encadeadas para armazenar tabelas de busca em memória secundária, também não é conveniente usar tabelas de dispersão com encadeamento ou com endereçamento aberto nesse meio de armazenamento.

Este capítulo discute dois métodos de dispersão em memória secundária. Na **Seção 8.1**, dispersão estática será discutida, enquanto o conceito e a implementação de dispersão extensível serão explorados na **Seção 8.2**.

8.1 Dispersão Estática

8.1.1 Motivação e Conceitos

Esta seção discute a maneira mais simples de implementar tabelas de dispersão em memória secundária, que é por meio da técnica conhecida como **dispersão estática**.

No contexto de dispersão em memória secundária, **coletor** (*bucket*, em inglês) é uma unidade de armazenamento externo capaz de conter um ou mais registros de uma tabela de busca. Pelas mesmas razões expostas no **Capítulo 6**, idealmente, o tamanho de um coletor deve ser igual ou um pouco menor do que o tamanho de um bloco no meio de armazenamento utilizado. Como todos os coletores têm o mesmo tamanho, eles podem ser indexados como se fossem elementos de um array.

Uma tabela de busca que usa a técnica de dispersão estática consiste em dois tipos de coletores:

- **Coletor primário**. O número de coletores primários deve ser decidido a priori e é considerado o **tamanho da tabela**. Esses coletores são acessados diretamente por intermédio de um índice resultante da aplicação de uma função de dispersão sobre uma chave.
- **Coletor excedente**. Um coletor excedente é usado como complemento de um coletor primário que se encontra repleto e pode ser acessado apenas por meio do coletor primário associado a ele.

A **Figura 8–1** ilustra uma tabela de dispersão estática com coletores primários e excedentes.

Idealmente, uma tabela de dispersão estática contém apenas coletores primários, pois, nesse caso, uma operação de busca requer apenas uma leitura em memória secundária e uma operação de inserção ou de remoção requer uma leitura e uma escrita nesse meio de armazenamento. Essa situação ideal, contudo, pode ser difícil de obter na prática.

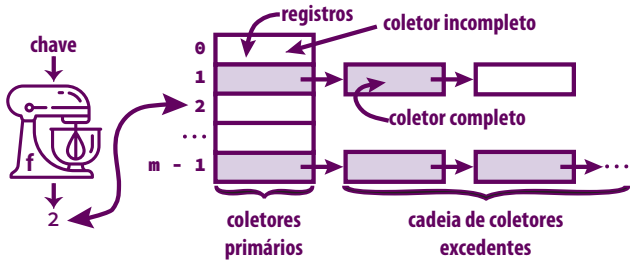


FIGURA 8–1: DISPERSÃO ESTÁTICA EM MEMÓRIA SECUNDÁRIA

8.1.2 Operações Básicas

Inserção

Para inserir um registro numa tabela de dispersão estática, aplica-se uma função de dispersão sobre a chave do registro (como é usual) e obtém-se o índice do coletor primário no qual o registro deverá ser inserido. Se houver espaço nesse coletor, o registro será inserido normalmente nele.

O problema é que o coletor primário que deveria acomodar um novo registro nem sempre tem espaço disponível. Nesse caso, deve-se alocar um coletor excedente e fazer com que o coletor primário aponte para o novo coletor excedente. Então o registro é inserido no referido coletor.

Obviamente, um coletor excedente também pode se tornar repleto, de maneira que vários coletores excedentes podem ser associados a um coletor primário, formando uma **cadeia de coletores excedentes** que se assemelha a uma lista encadeada (v. **Figura 8–1**).

O algoritmo de inserção em tabela de dispersão estática é apresentado na **Figura 8–2**.

ALGORITMO INSEREEMTABELADEDISPERSÃOESTÁTICA

ENTRADA: Um novo registro

ENTRADA/SAÍDA: Uma tabela de dispersão estática

1. Aplique a função de dispersão e obtenha o índice do coletor primário no qual o registro poderá ser inserido
2. Leia o coletor primário no arquivo e torne-o o coletor corrente
3. Enquanto o coletor corrente estiver repleto, faça:
   - 3.1 Se o coletor não possuir coletor excedente:
     - 3.1.1 Crie um novo coletor excedente e torne-o o coletor corrente
     - 3.1.2 Encerre o laço
   - 3.2 Caso contrário, torne-o o coletor corrente
4. Acrescente o novo registro ao final do coletor corrente
5. Atualize o número de registros do coletor corrente

FIGURA 8–2: ALGORITMO DE INSERÇÃO EM TABELA DE DISPERSÃO ESTÁTICA

Busca

Para efetuar uma busca numa tabela de dispersão estática, aplica-se a função de dispersão sobre a chave de busca, o que resulta no índice do coletor primário que poderá conter a chave. Então esse coletor é lido e efetua-se uma busca sequencial em memória principal. Se essa busca não for bem-sucedida e houver coletores excedentes, eles serão lidos um a um e a busca prossegue até que a chave seja encontrada ou não haja mais possibilidade de encontrá-la. A **Figura 8–3** mostra o algoritmo de busca em tabela de dispersão estática.

ALGORITMO BUSCAEMTABELADEDISPERSÃOESTÁTICA

ENTRADA: Uma chave de busca e uma tabela de dispersão estática

SAÍDA: O registro associado à chave de busca, se ele for encontrado, ou um valor indicando o fracasso da operação, em caso contrário

1. Aplicando a função de dispersão, obtenha o índice do coletor primário no qual o registro poderá ser encontrado e torne-o índice corrente
2. Enquanto o índice corrente for válido, faça:
   - 2.1 Leia o coletor associado ao índice corrente
   - 2.2 Efetue uma busca sequencial no coletor lido usando a chave de busca
   - 2.3 Se a chave foi encontrada, retorne o registro associado a ela
   - 2.4 Atribua ao índice corrente o índice do próximo coletor excedente
3. Retorne um valor informando que a chave de busca não foi encontrada

FIGURA 8–3: ALGORITMO DE BUSCA EM TABELA DE DISPERSÃO ESTÁTICA

Remoção

Se a chave do registro a ser removido for encontrada, conforme foi descrito acima, efetua-se, em memória principal, a remoção do registro do coletor em que ele se encontra e, em seguida, o coletor é reescrito no arquivo. Essa remoção em memória principal segue o algoritmo padrão de remoção em lista indexada discutido no **Volume 1**.

Se um coletor excedente ficar vazio após a remoção de um registro, operações posteriores de busca, inserção e remoção podem ser aceleradas se o referido coletor for removido de sua cadeia. Essa **remoção completa** é similar à remoção de um nó de uma lista simplesmente encadeada, mas não é tipicamente implementada, visto que não se espera que essa situação ocorra com frequência. Assim, normalmente, implementa-se apenas a **remoção simples** que foi descrita no parágrafo anterior.

O algoritmo de remoção em tabela de dispersão estática é apresentado na **Figura 8–4**.

ALGORITMO REMOVEEMTABELADEDISPERSÃOESTÁTICA

ENTRADA: A chave do registro a ser removido

ENTRADA/SAÍDA: Uma tabela de dispersão estática

SAÍDA: Um valor informando se a operação foi bem-sucedida

1. Encontre o coletor que contém o registro com a chave de entrada usando um algoritmo semelhante a **BUSCAEMTABELADEDISPERSÃOESTÁTICA**
2. Se o registro a ser removido não foi encontrado, retorne um valor informando que a operação foi malsucedida
3. Remova o registro que contém a chave de entrada do coletor que o contém
4. Atualize o número de registros do referido coletor
5. Escreva o coletor atualizado no arquivo

FIGURA 8–4: ALGORITMO DE REMOÇÃO EM TABELA DE DISPERSÃO ESTÁTICA

8.1.3 Implementação

Definições de Tipos e Constantes

O seguinte tipo de estrutura pode ser usado para implementar uma tabela de busca com dispersão estática em memória secundária. O tipo `tRegistroMEC` que aparece nesta definição é definido no **Apêndice A**.

```
typedef struct {
          tTipoDeColetorDEst tipo; /* Tipo de coletor */
          tRegistroMEC       registros[M]; /* Array de registros  */
          int                nRegistros;   /* Número de registros num coletor */
          int                proximo; /* Próximo coletor excedente */
       } tColetorDEst;
```

Nessa definição de tipo, `M` é uma constante simbólica que representa o número máximo de registros armazenados num coletor. O tipo `tTipoDeColetorDEst` é um tipo de enumeração usada para classificar um coletor como primário ou excedente e é definido como:

```
typedef enum {PRIMARIO, EXCEDENTE} tTipoDeColetorDEst;
```

Idealmente, o tamanho de um coletor deve ser o mais próximo possível do tamanho de um bloco em memória secundária, de modo que o cálculo do número máximo de registros num coletor é similar àquele apresentado na **Seção 6.3.1** para dimensionamento de grau para árvores multidirecionais. Para efetuar esse cálculo, suponha que:

- *TT* é o tamanho de uma variável do tipo `tTipoDeColetor`.
- *TB* é o tamanho de um bloco lido ou escrito num arquivo (v. **Seção 1.5.3**).

- *TR* é o tamanho de um registro armazenado num coletor; esse valor pode ser calculado usando-se o operador **sizeof** como `sizeof(tRegistroMEC)`, sendo que `tRegistroMEC` é o tipo do registro armazenado na tabela (v. **Apêndice A**).
- *TI* é a largura do tipo inteiro usado para representar o número de registros num coletor e a posição do próximo coletor excedente.
- *M* é o número máximo de registros num coletor, que se deseja determinar.

Assim pode-se calcular o tamanho (*TC*) de um coletor da seguinte maneira:

$$TC = TT + M \cdot TR + 2 \cdot TI$$

Como, idealmente, deve-se ter $TC \leq TB$, obtém-se:

$$M \leq (TB - 2 \cdot TI - TT)/TR$$

Idealmente, deve-se escolher o tamanho de um coletor de acordo com o tamanho de cada bloco lido/escrito no meio de armazenamento externo no qual o coletor se encontra armazenado. Portanto, sem levar em consideração um possível preenchimento de estrutura (v. **Seção 6.3.1**), basta considerar o tamanho (*M*) de um coletor como:

$$M = (TB - 2 \cdot TI - TT)/TR - 1$$

As definições de constantes a seguir refletem o que foi discutido acima.

```
#define TB 4096 /* Tamanho do bloco lido ou escrito */
#define TT (int) sizeof(tTipoDeColetorDEst) /* Tamanho de tTipoDeColetorDEst */
#define TR (int) sizeof(tRegistroMEC) /* Tamanho de um registro */
#define TI (int) sizeof(int) /* Tamanho de um campo do tipo int */

   /* Cálculo do número máximo de registros num coletor */
#define M (((TB - 2*TI - TT)/TR) - 1)
```

Funções Auxiliares

As funções a seguir são usadas como auxiliares na implementação de dispersão estática.

A função `NovoColetorDEst()` cria um novo coletor num arquivo e seus parâmetros são o stream associado ao arquivo que contém os coletores e o tipo do coletor que será criado. Essa função retorna o índice do coletor criado.

```
static int NovoColetorDEst(FILE *stream, tTipoDeColetorDEst tipo)
{
   tColetorDEst coletor;
   int          indice;

   coletor.nRegistros = 0;
   coletor.proximo = POSICAO_NULA;
   coletor.tipo = tipo;

      /* Move o apontador de posição do arquivo para seu final */
   MoveApontador(stream, 0, SEEK_END);

   indice = ftell(stream)/sizeof(coletor); /* Obtém o índice do coletor */

   fwrite(&coletor, sizeof(coletor), 1, stream); /* Escreve o coletor no arquivo */

   return indice;
}
```

A função `LeColetorDEst()` lê um coletor em arquivo e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo contendo os coletores no qual será feita a leitura
- `pos` (entrada) — posição no arquivo na qual será feita a leitura
- `coletor` (saída) — ponteiro para o coletor que conterá o resultado da leitura

```
static void LeColetorDEst(FILE *stream, int pos, tColetorDEst *coletor)
{
      /* Tenta mover o apontador de arquivo para o local */
      /* de leitura; se não conseguir, aborta o programa */
   MoveApontador(stream, sizeof(tColetorDEst)*pos, SEEK_SET);

   fread(coletor, sizeof(tColetorDEst), 1, stream); /* Efetua a leitura */

      /* Certifica-se que não houve erro de leitura */
   ASSEGURA(!ferror(stream),  "Erro de leitura em LeColetorDEst()");
}
```

A função `EscreveColetorDEst()` escreve um coletor em arquivo e tem como parâmetros:

- `stream` (entrada) — stream associado ao arquivo contendo os coletores no qual será feita a escrita
- `pos` (entrada) — posição no arquivo no qual será feita a escrita
- `*coletor` (entrada) — coletor que será escrito no arquivo

```
static void EscreveColetorDEst(FILE *stream, int pos, const tColetorDEst *coletor)
{
      /* Tenta mover o apontador de arquivo para o local */
      /* de escrita; se não conseguir, aborta o programa */
   MoveApontador(stream, sizeof(tColetorDEst)*pos, SEEK_SET);

   fwrite(coletor, sizeof(tColetorDEst), 1, stream); /* Efetua a escrita */

      /* Certifica-se que não houve erro */
   ASSEGURA(!ferror(stream), "Erro de escrita em EscreveColetorDEst()");
}
```

Iniciação

A função `IniciaTabDEst()` inicia uma tabela de busca com dispersão estática e seus parâmetros são:

- `streamCol` (entrada) — stream associado ao arquivo que contém os coletores
- `nRegistros` (entrada) — número de registros iniciais da tabela
- `maxRegs` (entrada) — número máximo de registros num coletor

```
int IniciaTabDEst( FILE *streamCol, int nRegistros, int maxRegs )
{
   int i,
       nColetores;

   nColetores = nRegistros/maxRegs; /* Obtém o número de coletores */

      /* Aloca os coletores primários no arquivo */
   for (i = 0; i < nColetores; ++i)
      (void) NovoColetorDEst(streamCol, PRIMARIO);

   return nColetores;
}
```

A função `IniciaTabEstatica()` retorna o número de coletores primários da tabela.

Busca

A função `BuscaDEst()` efetua uma busca numa tabela de dispersão estática e tem como parâmetros:

- `stream` (entrada) — stream associado ao arquivo que contém os coletores
- `nColetores` (entrada) — número de coletores primários da tabela
- `chave` (entrada) — a chave de busca
- `reg` (saída) — o registro encontrado

Essa função retorna o endereço do registro, se ele for encontrado, ou **NULL**, em caso contrário.

```
tRegistroMEC *BuscaDEst(FILE *stream, int nColetores, tChave chave, tRegistroMEC *reg)
{
   int          iColetor, i;
   tColetorDEst coletor;

      /* Obtém o índice do coletor primário que pode conter o registro */
   iColetor = FDispersao(chave, nColetores);

      /* Efetua uma busca no coletor primário e em seus coletores excedentes */
   while (iColetor != POSICAO_NULA) {
         /* Lê um coletor que pode conter o registro */
      LeColetorDEst(stream, iColetor, &coletor);

         /* Efetua uma busca sequencial pelo registro em memória principal */
      for (i = 0; i < coletor.nRegistros; i++)
            /* Verifica se a chave foi encontrada */
         if (chave == ObtemChave(&coletor.registros[i])) {
            *reg = coletor.registros[i]; /* Chave encontrada */
            return reg; /* Serviço completo */
         }

      iColetor = coletor.proximo; /* Passa para o próximo coletor excedente */
   }

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

Inserção

A função `InsereDEst()` insere um registro numa tabela de busca com dispersão estática e seus parâmetros são:

- `nColetores` (entrada) — número de coletores primários da tabela na qual será efetuada a inserção
- `registro` (entrada) — o registro que será inserido
- `stream` (entrada) — stream associado ao arquivo que contém os coletores

```
void InsereDEst(int nColetores, const tRegistroMEC *registro, FILE *stream)
{
   int          iColetor;
   tChave       chave;
   tColetorDEst coletor;

   chave = ObtemChave(registro); /* Obtém a chave do registro */

   iColetor = FDispersao(chave, nColetores); /* Obtém o índice do coletor */

      /* Lê o coletor primário em arquivo */
   LeColetorDEst(stream, iColetor, &coletor);

      /* Encontra o coletor no qual será efetuada a inserção */
   while (coletor.nRegistros == M) {
         /* O coletor está repleto. Portanto passa-se para o    */
         /* próximo coletor excedente. Se o próximo coletor     */
         /* excedente ainda não existe, cria-se um novo coletor */
      if (coletor.proximo == POSICAO_NULA) {
            /* Cria um novo coletor excedente */
         coletor.proximo = NovoColetorDEst(stream, EXCEDENTE);

            /* O coletor foi alterado e precisa ser reescrito */
         EscreveColetorDEst(stream, iColetor, &coletor);
      }
```

```
         /* Guarda a posição do próximo coletor a ser lido */
      iColetor = coletor.proximo;

         /* Lê um coletor excedente em arquivo */
      LeColetorDEst(stream, coletor.proximo, &coletor);
   }

      /* Acrescenta o novo registro ao final do coletor */
   coletor.registros[coletor.nRegistros] = *registro;

   coletor.nRegistros++; /* O número de registros no coletor aumentou */

      /* O coletor foi alterado e é preciso reescrevê-lo no arquivo */
   EscreveColetorDEst(stream, iColetor, &coletor);
}
```

Remoção

A função `RemoveDEst()` remove um registro de uma tabela de dispersão estática e tem como parâmetros:

- `nColetores` (entrada/saída) — número de coletores primários da tabela de dispersão
- `chave` (entrada) — a chave do registro que será removido
- `stream` (entrada) — stream associado ao arquivo que contém os coletores

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

```
int RemoveDEst(int nColetores, tChave chave, FILE *stream)
{
   int          i, j, iColetor, encontrada = 0;
   tColetorDEst coletor;

      /* Obtém o índice do coletor que pode conter o registro */
   iColetor = FDispersao(chave, nColetores);

      /* Efetua uma busca no coletor primário e em seus coletores excedentes */
   while (iColetor != POSICAO_NULA) {
         /* Lê o coletor que pode conter o registro */
      LeColetorDEst(stream, iColetor, &coletor);

         /* Efetua uma busca sequencial pelo registro em memória principal */
      for (i = 0; i < coletor.nRegistros && !encontrada; i++)
            /* Verifica se a chave foi encontrada */
         if (chave == ObtemChave(&coletor.registros[i]))
            encontrada = 1;

         /* Se a chave foi encontrada, encerra a busca */
      if (encontrada)
         break;

      iColetor = coletor.proximo; /* Passa para o próximo coletor excedente */
   }

      /* Verifica se a chave foi encontrada */
   if (!encontrada)
      return 0;  /* A chave não foi encontrada */

      /* Corrige o valor de i que foi incrementado e deixou */
      /* de ser o índice do elemento a ser removido         */
   --i;

      /* O registro a ser removido encontra-se na posição i do */
      /* coletor corrente. É necessário mover uma posição para */
      /* trás todos os registros que se encontram à sua frente */
   for (j = i; j < coletor.nRegistros - 1; ++j)
```

```
      coletor.registros[j] = coletor.registros[j + 1];
   --coletor.nRegistros; /* O número de registros no coletor diminuiu */
      /* O coletor foi alterado e é preciso reescrevê-lo no arquivo */
   EscreveColetorDEst(stream, iColetor, &coletor);
   return 1;  /* Remoção bem-sucedida */
}
```

A função `RemoveEstatica()` implementa apenas a remoção simples descrita na **Seção 8.1.2**. A implementação de uma função que efetua remoção completa é deixada como exercício para o leitor.

8.1.4 Análise

O desempenho das operações com tabelas de busca com dispersão estática pode ser severamente prejudicado devido à ocorrência de longas cadeias de coletores excedentes. Essas cadeias podem ser ocasionadas por dois fatores principais: (1) número insuficiente de coletores primários e (2) excesso de colisões de chaves.

Se o número de registros crescer muito além do previsto, esses coletores excedentes poderão prejudicar consideravelmente o desempenho das operações de busca, inserção e remoção. Por outro lado, se o número de registros ficar muito abaixo do previsto, haverá desperdício de espaço de armazenamento. A única maneira eficiente de resolver o problema decorrente do uso de coletores excedentes é redimensionar a tabela com um número maior de coletores primários.

O redimensionamento de tabela e o uso de uma nova função de dispersão podem aliviar consideravelmente esses problemas, mas aí cria-se um novo obstáculo, pois essas soluções são bastante onerosas em se tratando de tabela de busca implementada em memória secundária. Por exemplo, um redimensionamento de tabela, como aquele descrito na **Seção 7.5**, poderia tornar um programa inutilizável por um período inaceitável se ela fosse implementada em memória secundária.

8.2 Dispersão Extensível

A abordagem que será discutida na presente seção para implementação de tabelas de dispersão mostra como o número de coletores e a função de dispersão de uma tabela de busca implementada em memória secundária podem ser alterados dinamicamente. Essa abordagem, denominada **dispersão extensível**, foi desenvolvida por Fagin, Nievergelt, Pippenger e Strong em 1979 (v. **Bibliografia**). A descrição a ser apresentada aqui é mais simples e diverge consideravelmente da versão original do ponto de vista conceitual, mas o resultado prático é o mesmo.

8.2.1 Conceitos

Tabela de dispersão extensível é uma tabela de busca baseada em memória secundária que usa uma função de dispersão e um diretório para localizar o coletor no qual um registro pode ser encontrado ou inserido. Diferentemente do que ocorre com dispersão estática (v. **Seção 8.1**), dispersão extensível raramente usa coletores excedentes.

O **diretório** (ou **índice**) de uma tabela de dispersão extensível é um array, idealmente, mantido em memória principal durante as operações, com as seguintes características:

1. Cada índice do array está associado a um valor obtido por meio da aplicação de uma função de dispersão sobre uma chave.
2. O conteúdo de cada elemento do array representa uma referência para um coletor armazenado em memória secundária, sendo que dois ou mais desses elementos podem fazer referência a um mesmo coletor.

O diretório de uma tabela de busca com dispersão extensível é dinamicamente ajustado para refletir alterações no número de registros da tabela. A principal característica de dispersão extensível é essa organização de diretório, que é uma **tabela extensível**, visto que seu tamanho pode aumentar ou diminuir de acordo com a demanda.

Quando uma função de dispersão é aplicada a uma determinada chave, o valor resultante indica uma posição no diretório e não no arquivo que contém os registros. Desse modo, o arquivo não requer nenhuma reorganização quando registros são acrescentados ou removidos dele, porque essas alterações são indicadas no diretório.

Para obter o efeito desejado considera-se o valor resultante da aplicação da função de dispersão como um string de bits do qual apenas os g bits mais à esquerda podem ser usados. O valor de g é denominado **profundidade global** ou **profundidade do diretório**. Precisamente, se g representa o número de bits usados para definir cada índice do diretório, o número de índices (i.e., o tamanho) desse diretório é 2^g. Como exemplo, suponha que a função de dispersão gera sequências de 5 bits. Se uma dessas sequências é *01011* e a profundidade global é *2*, os dois bits mais à esquerda (i.e., *01*) constituem uma posição no diretório contendo a referência para um coletor no qual a chave pode ser encontrada ou inserida.

Cada coletor possui uma **profundidade local** l, que é o número de bits iniciais que todas as chaves desse coletor têm em comum. Quer dizer, os bits mais à esquerda são os mesmos para todas as chaves no coletor. O valor de l depende de cada coletor, sendo que $l \leq g$. A profundidade local de um diretório permite não apenas decidir se o diretório será duplicado como também o número de referências para o coletor no diretório. Precisamente, o número de elementos de um diretório de profundidade global g de tabela de dispersão extensível que fazem referência a um coletor de profundidade local l é 2^{g-l}.

A **Figura 8–5** resume a terminologia empregada em dispersão extensível. Nessa figura, a profundidade global é igual a *2* e o tamanho do diretório é *4* (i.e., 2^2). Os coletores são rotulados como t_{00}, t_{01} e t_1, em que os subscritos indicam os bits iniciais que são comuns a todas as chaves armazenadas no respectivo coletor. Por exemplo, os valores de dispersão de todas as chaves armazenadas no coletor t_{00} iniciam com os bits *00*, enquanto os valores de dispersão das chaves armazenadas no coletor t_1 têm apenas um bit inicial em comum, que é *1*. Na **Figura 8–5**, a capacidade de cada coletor é de três registros[1], o que significa que os coletores t_{00} e t_1 estão completos, o que não ocorre com o coletor t_{01}. Nessa figura, a profundidade local de cada coletor aparece no topo dele.

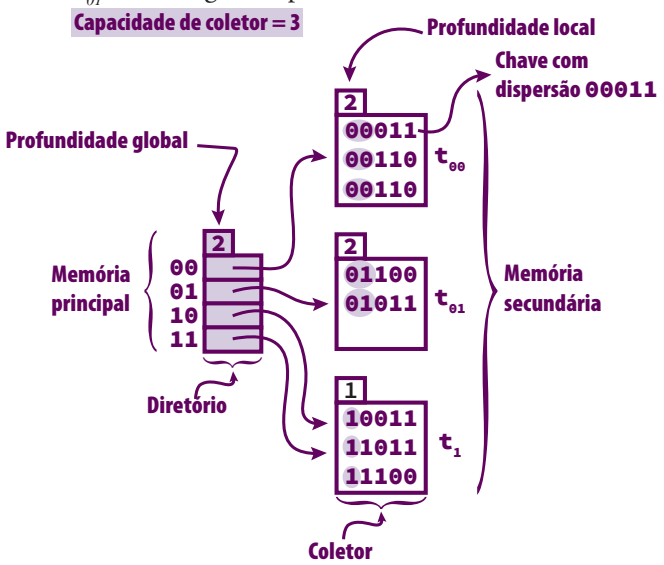


Figura 8–5: Terminologia Usada em Dispersão Extensível

8.2.2 Inserção

Supondo que g seja a profundidade global de uma tabela de busca com dispersão extensível, uma operação de inserção começa obtendo os g bits iniciais da chave a ser inserida. Esses bits representam o índice do elemento

[1] Para simplificar a discussão, fazem-se referências apenas a chaves nos coletores. Mas coletores armazenam registros, e não apenas as chaves desses registros.

do diretório que armazena a posição em arquivo do coletor que armazenará a nova chave. De posse dessa posição, o referido coletor é lido e, se ele tiver espaço disponível, efetua-se a devida inserção e escreve-se o coletor de volta no arquivo que o contém. Se o referido coletor estiver completo, ele precisará ser dividido antes que a inserção ocorra.

A profundidade local de um coletor determina se a duplicação do diretório será necessária após ele ser dividido. Suponha que a profundidade global de um diretório e a profundidade local de um coletor sejam, respectivamente, g e l. Então, quando esse coletor requer divisão, podem acontecer duas situações:

1. $l < g$. Quando a profundidade local é menor do que a profundidade global, dividir o coletor requer apenas alterar metade das referências para esse coletor, de modo que elas apontem para o coletor recentemente criado. Nesse caso, o coletor que será dividido possui mais de uma referência para si no diretório, de sorte que o diretório não precisará ser duplicado. Ou seja, basta que metade das referências para o coletor dividido seja redirecionada para o novo coletor.
2. $l = g$. Nesse caso, o coletor a ser dividido possui apenas uma referência para si no diretório, de modo que o diretório precisará ser duplicado (talvez mais de uma vez), para que sejam criadas referências para os coletores resultantes da divisão. Cada coletor, com possível exceção daquele dividido e do novo coletor, terá um número maior de referências para si. O número de referências repetidas em cada coletor depende do número de vezes que o diretório for duplicado (v. abaixo).

Em qualquer caso, as chaves presentes no antigo coletor e a nova chave são divididas de acordo com o valor do bit de ordem $l + 1$. Quer dizer, se uma chave apresentar *0* como valor desse bit, ela permanece no antigo coletor; se o valor desse bit for *1*, por outro lado, ela será armazenada no novo coletor. Acontece, porém, que, após essa divisão de chaves, pode acontecer que todas elas terminem num mesmo coletor, de modo que o processo precisa ser repetido. Felizmente, isso só ocorre quando as chaves em questão apresentam o mesmo valor de dispersão devido ao fato de elas serem iguais ou porque a função de dispersão não foi convenientemente escolhida. Quando todas as chaves sob consideração são iguais, duplicações múltiplas do diretório não resolverão o problema, que só poderá ser solucionado com o uso de coletores excedentes, como ocorre com dispersão estática (v. **Seção 8.2.5**).

Considerando a **Figura 8–5**, suponha que se deseja inserir uma chave cujo valor de dispersão seja *11001*. Levando em conta esse valor de dispersão, conclui-se que ela deve ser inserida no coletor t_1, pois a profundidade global é *2* e, no índice do diretório correspondente aos dois primeiros bits do valor de dispersão (*11*), emana um ponteiro para esse coletor. Ocorre, porém, que esse coletor se encontra repleto [v. **Figura 8–6 (a)**], já que se supõe que a capacidade de cada coletor corresponde a três chaves. Assim o coletor t_1 é dividido em dois coletores: t_{10}, contendo a chave com valor de dispersão *10011*, e t_{11}, contendo as chaves com valores de dispersão *11011*, *11100* e *11001* [v. **Figura 8–6 (b)**]. Cada um desses dois novos coletores possui profundidade local igual a *2*.

Nesse último exemplo, não é necessário criar os dois novos coletores (t_{10} e t_{11}) e liberar o antigo coletor (t_1). Quer dizer, o coletor já existente precisa apenas ter sua profundidade alterada de *1* para *2*, de maneira que ele passa a ser o coletor t_{10}. Por sua vez, o coletor t_{11} precisa realmente ser criado. Para esse último coletor, são copiadas as chaves do antigo coletor t_1 que têm valores de dispersão que começam com *11* mais a nova chave com valor de dispersão *11001*. Além disso, é necessária apenas uma alteração no diretório: o ponteiro que emanava da posição *11* do diretório e apontava para o coletor t_1 passa a apontar para o coletor t_{11}.

A **Figura 8–6** ilustra o caso mais simples de divisão de coletor, pois ele não requer reestruturação de diretório. A situação é mais complicada quando a capacidade de um coletor é excedida e sua profundidade local é igual à profundidade do diretório. Por exemplo, considere o caso em que uma chave com valor de dispersão igual a *00001* deve ser inserida na tabela da **Figura 8–7 (a)** e é despachada usando-se a posição *00* (seus dois primeiros bits) do diretório para o coletor t_{00}. Nesse caso, ocorre uma divisão, mas o diretório não tem elemento

disponível para fazer referência para o novo coletor resultante dessa divisão. Consequentemente, o tamanho do diretório é duplicado, de modo que sua profundidade passa a ser igual a *3*, t_{00} passa a ser t_{000} e o novo coletor é t_{001}. Todas as chaves de t_{00} são divididas entre t_{000} e t_{001}: aquelas cujos valores de dispersão começam com *000* ficam no coletor t_{000} e as chaves restantes, com prefixo *001*, são alocadas em t_{001}. Além disso, todos os elementos do novo diretório são devidamente alterados, como mostra a **Figura 8–7 (b)**.

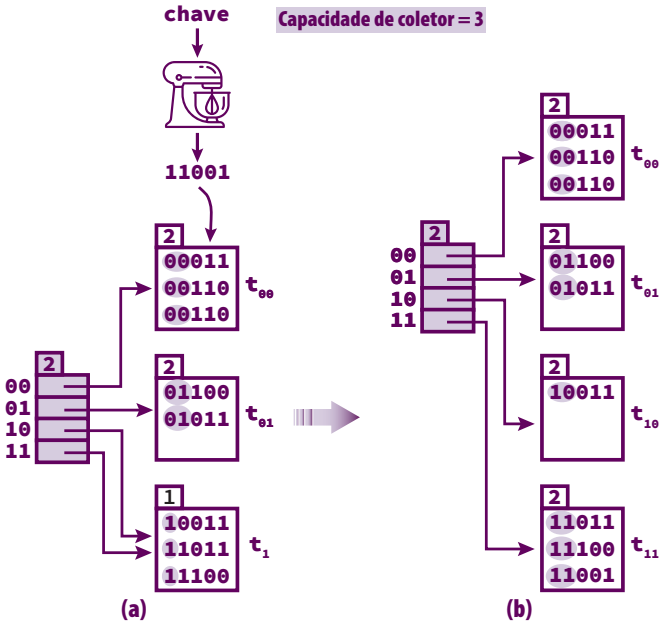


Figura 8–6: Dispersão Extensível: Divisão de Coletor sem Duplicação de Diretório

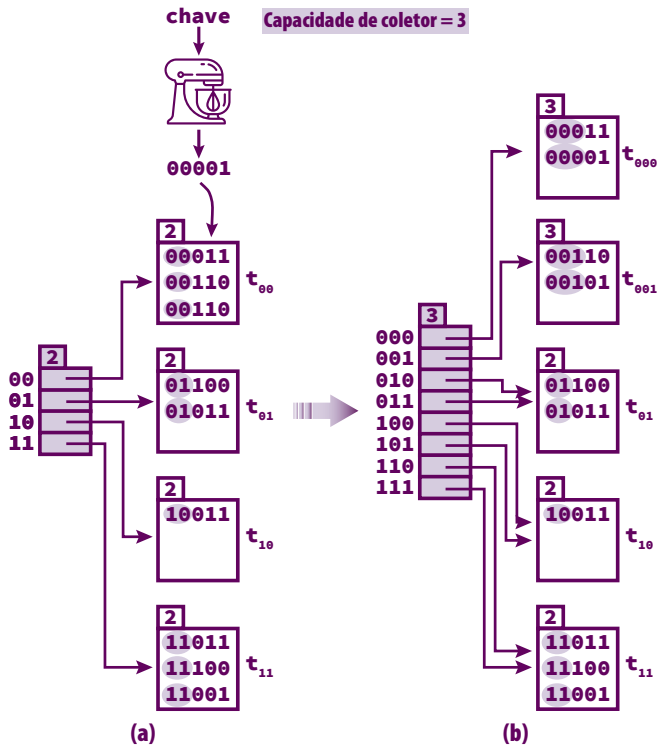


Figura 8–7: Dispersão Extensível: Divisão de Coletor com Duplicação de Diretório

O algoritmo de inserção em tabela de dispersão extensível é apresentado na **Figura 8–8**.

ALGORITMO INSEREEMTABELADEDISPERSÃOEXTENSÍVEL

ENTRADA: Um novo registro
ENTRADA/SAÍDA: Uma tabela de dispersão extensível

1. Obtenha o índice do coletor primário no qual o registro poderá ser inserido usando a chave de busca e a profundidade global da tabela
2. Leia o coletor primário no arquivo
3. Insira, de maneira ordenada, o novo registro no coletor
4. Atualize o número de registros no coletor
5. Se o coletor ficou repleto, divida-o usando o algoritmo **DIVIDECOLETORDEDISPERSÃOEXTENSÍVEL**
6. Escreva o coletor que foi alterado no arquivo

FIGURA 8–8: ALGORITMO DE INSERÇÃO EM TABELA DE DISPERSÃO EXTENSÍVEL

A **Figura 8–9** exibe o algoritmo **DIVIDECOLETORDEDISPERSÃOEXTENSÍVEL** invocado pelo algoritmo de inserção da **Figura 8–8**.

ALGORITMO DIVIDECOLETORDEDISPERSÃOEXTENSÍVEL

ENTRADA: O índice do coletor que será dividido no arquivo de coletores
ENTRADA/SAÍDA: Uma tabela de dispersão extensível e o coletor que será dividido

1. Crie um novo coletor e obtenha seu índice em arquivo
2. Divida os elementos do antigo coletor com o novo coletor
3. Se o coletor não pode ser dividido:
   - **3.1** Insira o novo registro no novo coletor
   - **3.2** Torne o novo coletor um coletor excedente
4. Escreva o novo coletor no arquivo
5. Insira uma referência para o novo coletor no diretório usando o algoritmo **INSEREEMDIRETÓRIODEDISPERSÃOEXTENSÍVEL**

FIGURA 8–9: ALGORITMO DE DIVISÃO DE COLETOR EM TABELA DE DISPERSÃO EXTENSÍVEL

Os detalhes de execução do **Passo 2** do algoritmo **DIVIDECOLETORDEDISPERSÃOEXTENSÍVEL** foram descritos anteriormente nesta seção e o uso de coletores excedentes em dispersão extensível (**Passo 3.2**) será explorado na **Seção 8.2.5**. A **Figura 8–10** mostra o algoritmo **INSEREEMDIRETÓRIODEDISPERSÃOEXTENSÍVEL** invocado pelo algoritmo **DIVIDECOLETORDEDISPERSÃOEXTENSÍVEL**.

ALGORITMO INSEREEMDIRETÓRIODEDISPERSÃOEXTENSÍVEL

ENTRADA: O coletor que será inserido e seu índice no arquivo de coletores
ENTRADA/SAÍDA: Uma tabela de dispersão extensível

1. Atribua a *pl* a profundidade local do coletor
2. Enquanto a profundidade global do diretório for menor do que *pl*, faça
   - **2.1** Crie um novo diretório com o dobro do tamanho do diretório atual
   - **2.2** Copie as referências do antigo diretório para a metade final do novo diretório
   - **2.3** Atualize a profundidade global e o tamanho do novo diretório
3. Utilizando os bits menos significativos coincidentes no coletor, corrija as referências do diretório para o novo coletor

FIGURA 8–10: ALGORITMO DE INSERÇÃO EM DIRETÓRIO DE DISPERSÃO EXTENSÍVEL

8.2.3 Busca

Supondo que *g* seja a profundidade global de uma tabela de busca com dispersão extensível, uma operação de busca em dispersão extensível começa obtendo os *g* bits iniciais da chave de busca, que representam o índice do elemento do diretório que armazena a posição em arquivo do coletor que pode conter o registro procurado. De posse dessa posição, o referido coletor é lido e efetua-se uma busca sequencial em memória principal.

Operações de busca tornam-se mais complicadas se forem usados coletores excedentes (v. **Seção 8.2.5**). Nesse caso, usa-se a abordagem discutida na **Seção 8.1**.

O algoritmo de busca em tabela de dispersão extensível é apresentado na **Figura 8–11**.

ALGORITMO BUSCAEMTABELADEDISPERSÃOEXTENSÍVEL

ENTRADA: Uma tabela de dispersão extensível e uma chave de busca

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

1. Obtenha o índice do coletor que contém o registro
2. Leia o coletor primário no arquivo e torne-o o coletor corrente
3. Enquanto não houver retorno, faça:
   - 3.1 Efetue uma busca sequencial pelo registro no coletor corrente usando a chave de busca
   - 3.2 Se a chave de busca foi encontrada, retorne o registro associado a ela
   - 3.3 Se houver coletor excedente associado ao coletor corrente, leia o coletor excedente no arquivo e torne-o o coletor corrente
   - 3.4 Caso contrário, retorne um valor informando o fracasso da operação

FIGURA 8–11: ALGORITMO DE BUSCA EM TABELA DE DISPERSÃO EXTENSÍVEL

8.2.4 Remoção

Às vezes, quando chaves são removidas de uma tabela de dispersão extensível, é possível recombinar coletores adjacentes, e essa recombinação de coletores pode também permitir uma redução de tamanho do diretório. Quer dizer, a remoção de uma chave pode fazer com que o coletor que a continha torne-se vazio. Nesse caso, talvez, ele possa ser combinado com outro coletor. Coletores que são candidatos a união são chamados **coletores camaradas**.

Coletores camaradas são aqueles cujas referências no diretório diferem apenas em seu último bit. Quando ocorre uma união de coletores, o tamanho do diretório pode ser reduzido à metade e, consequentemente, sua profundidade global também é reduzida, fazendo com que todas as profundidades locais sejam estritamente menores do que a profundidade global. Formalmente, para que dois coletores *X* e *Y* sejam camaradas, eles devem satisfazer os seguintes critérios:

- ❒ As profundidades locais de *X* e de *Y* devem ser iguais à profundidade global do diretório
- ❒ Se o índice do endereço de *X* for $b_0\, b_1 \ldots 0$, o índice do endereço de seu camarada *Y* deve ser $b_0\, b_1 \ldots 1$ ou vice-versa
- ❒ Juntas, as chaves contidas em *X* e *Y* cabem num único coletor

Se dois coletores forem unidos, talvez seja possível reduzir o tamanho do diretório. Isso pode acontecer se cada par de elementos do diretório indexados por $b_0\, b_1 \ldots 0$ e $b_0\, b_1 \ldots 1$ apontar para um coletor comum. Se esse for o caso, o bit final desses índices do diretório não está sendo usado para diferenciar coletores, de modo que um dos constituintes de cada par pode ser removido e o tamanho do diretório pode ser reduzido pela metade.

Considere como exemplo a tabela ilustrada na **Figura 8–12 (a)** e suponha que se remova uma chave do coletor *X*. O endereço de *X* encontra-se no índice *100* do diretório, de modo que seu camarada é o coletor *Y* cujo endereço encontra-se no índice *101*. Se o número combinado de chaves dos coletores *X* e *Y* couber num único coletor, esses coletores podem ser unidos num único coletor, que é o coletor *Y* na **Figura 8–12 (b)**. Cada par de elementos no diretório ilustrado nessa última figura faz referência a um coletor comum. Isso significa que o tamanho desse diretório pode ser reduzido de *3* bits para *2* bits. Fazendo-se isso, obtém-se como resultado o diretório com *4* elementos mostrado na **Figura 8–12 (c)**.

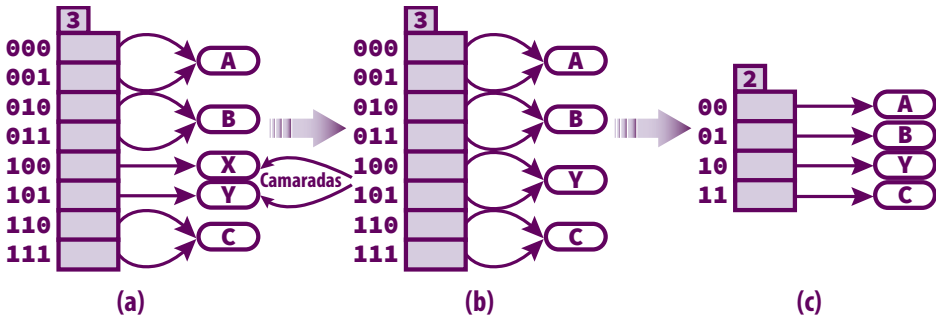


Figura 8–12: Redução de Diretório em Dispersão Extensível

Combinar coletores após uma operação de remoção é relativamente fácil e de baixo custo. Por sua vez, a redução de tamanho do diretório acarretada por essa operação pode ser onerosa quando o tamanho do diretório é muito grande, de maneira que reduzir o tamanho do diretório só é recomendável quando o número de coletores é substancialmente reduzido com relação ao tamanho do diretório. Outro revés decorrente da combinação de coletores é que um coletor vazio sem referência para si continuará a ocupar espaço no arquivo mesmo que não possa mais ser acessado, a não ser que seja adotado um esquema de gerenciamento de coletores vazios, o que complicaria ainda mais a implementação de uma tabela de busca com dispersão extensível.

Os exemplos de inserção apresentados acima também podem servir como exemplos de remoção se forem examinados de trás para a frente. Isto é, se você começar com a tabela resultante de uma inserção e, então, remover o elemento inserido, o resultado deverá ser a tabela original antes da inserção.

O algoritmo de remoção em tabela de dispersão extensível segue os passos vistos na **Figura 8–13**.

Algoritmo RemoveEmTabelaDeDispersãoExtensível

Entrada: A chave do registro a ser removido

Entrada/Saída: Uma tabela de dispersão extensível

Saída: Um valor informando se a operação foi bem-sucedida

1. Encontre o coletor que contém o registro com a chave de entrada usando um algoritmo semelhante a **BuscaEmTabelaDeDispersãoExtensível**
2. Se o registro a ser removido não foi encontrado, retorne um valor informando que a operação foi malsucedida
3. Remova o registro que contém a chave de entrada do coletor que o contém
4. Atualize o número de registros do referido coletor

Figura 8–13: Algoritmo de Remoção em Tabela de Dispersão Extensível

O algoritmo da **Figura 8–13** não leva em consideração uma possível redução de diretório em virtude de uma operação de remoção.

8.2.5 Uso de Coletores Excedentes

Na prática, tipicamente, em vez dos bits mais significativos usados para indexação de diretórios como foi visto na **Seção 8.2.2**, usam-se os bits menos significativos, pois isso facilita a implementação. Por exemplo, desse modo, pode-se duplicar um diretório simplesmente copiando-o (v. **Seção 8.2.6**). Essa opção será adotada no exemplo a ser apresentado nesta seção.

Nesta seção, será apresentado um exemplo completo de construção de uma tabela de dispersão fictícia. Para ilustrar esse exemplo, será usado o conjunto de registros mostrado na **Tabela 8–1**. Cada registro dessa tabela consiste em apenas dois campos: *nome* e *idade*, mostrados com fundo colorido nessa tabela. A chave do registro é o campo *nome* e a função de dispersão usa uma digital de Rabin[2] para obter os dígitos menos significativos necessários para indexar o diretório. Apesar de esse exemplo ser irrealista, o leitor poderá com ele dirimir dúvidas referentes aos conceitos expostos acima.

Nº	Nome	Idade	Valor de Dispersão
1	Jose	21	00001001
2	Joaquim	19	01010101
3	Manoel	31	00011000
4	Jose	18	00001001
5	Maria	22	00110111
6	Mario	25	01000101
7	Isabel	25	00011100
8	Jose	20	00001001

Tabela 8–1: Registros do Exemplo de Dispersão Extensível

No exemplo a ser desenvolvido aqui, assume-se que um coletor tem capacidade para conter apenas dois registros. A **Figura 8–14** ilustra o estado inicial da tabela de busca com dispersão extensível (i.e., logo após ela ter sido criada).

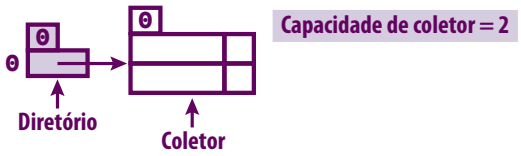


Figura 8–14: Exemplo de Dispersão Extensível: Estado Inicial

A **Figura 8–15** mostra o estado da tabela de busca após as inserções dos dois primeiros registros da **Tabela 8–1**. Essas inserções ocorrem sem divisão de coletor.

Figura 8–15: Exemplo de Dispersão Extensível: Primeiras Duas Inserções

A **Figura 8–16** mostra o estado da tabela de busca após a inserção do terceiro registro. Essa operação é um pouco mais complexa do que a anterior porque requer a criação de um novo coletor e a duplicação do diretório. Note, nessa figura, que os coletores passam a ter profundidades locais iguais a *1*, e o mesmo ocorre com a profundidade global do diretório. Portanto as chaves nesses coletores passam a ter em comum o último bit à direita (i.e., o

[2] Digital de Rabin é um método polinomial de cálculo de valores de dispersão que será explorado na **Seção 9.6**.

bit menos significativo). Mais precisamente, os dois registros do coletor inferior na figura apresentam *1* como último bit de suas chaves (v. **Tabela 8–1**), enquanto a chave do único registro no coletor superior da figura é *0*.

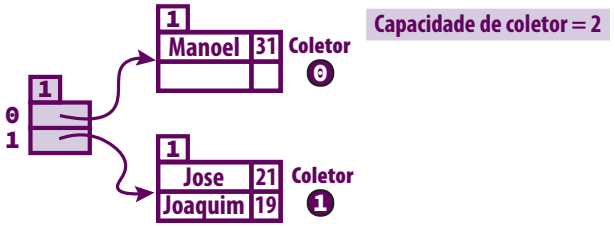


Figura 8–16: Exemplo de Dispersão Extensível: Terceira Inserção

A inserção do quarto registro da **Tabela 8–1** é complicada pelo fato de essa operação requerer duas divisões de coletores e duas duplicações do diretório. Na primeira tentativa de inserção da chave desse registro, verifica-se que ele deveria ser inserido no coletor ❶ da **Figura 8–16**. Ocorre, porém, que esse coletor encontra-se repleto, o que requer a primeira divisão de coletor. Mas a divisão desse coletor não é produtiva, pois os dois bits menos significativos das chaves dos três registros em questão não podem ser usados para distingui-los. Ou seja, os dois bits menos significativos dos registros `1`, `2` e `4` da **Tabela 8–1** são os mesmos. Esse impasse leva à situação intermediária mostrada na **Figura 8–17**.

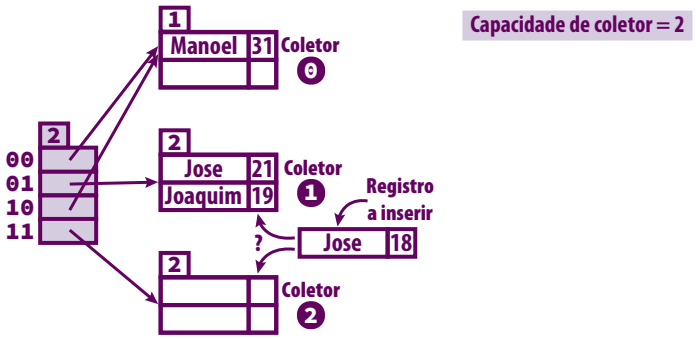


Figura 8–17: Exemplo de Dispersão Extensível com Duplicação Dobrada 1

Neste ponto, duas questões podem estar intrigando o leitor:

1. Por que o novo registro não é inserido no novo coletor ❷ na **Figura 8–17**? A resposta para essa questão é simples. Basta notar que o elemento de índice $\texttt{11}_2$ aponta para o novo coletor, de modo que nesse coletor devem residir registros cujas chaves tenham `11` como seus dois bits menos significativos, o que não é o caso do novo registro.
2. Se o novo coletor não armazena nenhum registro, por que ele precisa existir? A resposta a essa questão é um pouco sutil e está relacionada ao fato de todo elemento do diretório dever estar associado a algum coletor e não há nenhum outro coletor com o qual o elemento de índice $\texttt{11}_2$ possa estar associado.

A **Figura 8–18** mostra o estado da tabela de busca após a criação de um novo coletor seguida de uma nova duplicação de diretório.

A **Figura 8–19** ilustra o estado da tabela de busca após as inserções dos registros `5`, `6` e `7` da **Tabela 8–1**. Essas inserções são efetuadas normalmente sem que haja necessidade de criação de novo coletor ou duplicação de diretório.

A inserção do registro número 8 da **Tabela 8–1**, ilustrada na **Figura 8–20**, pode ser mais intrigante, pois ela faz com que o algoritmo de inserção descrito na **Seção 8.2.2** deixe de funcionar. Ou seja, ele entra em repetição infinita porque não é capaz de distinguir as chaves dos registros que ora se encontram no coletor ❶ da chave do registro a ser inserido para qualquer que seja o número de bits usados para tentar fazer essa distinção. A razão

para esse imbróglio é óbvia: simplesmente, as três chaves em questão são iguais e supõe-se que um coletor tem capacidade para apenas dois registros. Essa situação é a grande fraqueza do esquema de dispersão extensível e sua única solução é a adoção do uso de coletores excedentes, assim como ocorre com dispersão estática discutida na **Seção 8.1**, como mostra a **Figura 8–20**.

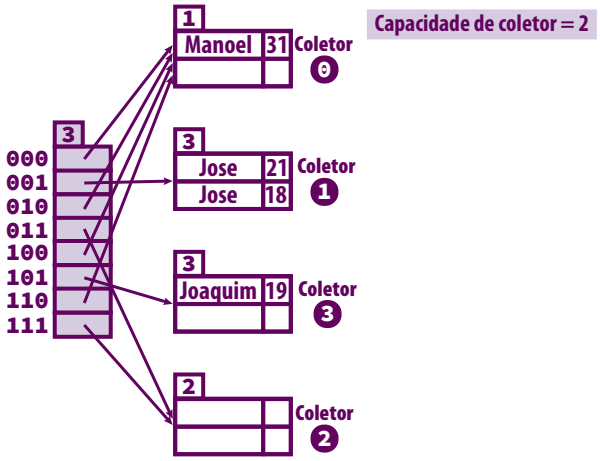


FIGURA 8–18: EXEMPLO DE DISPERSÃO EXTENSÍVEL COM DUPLICAÇÃO DOBRADA 2

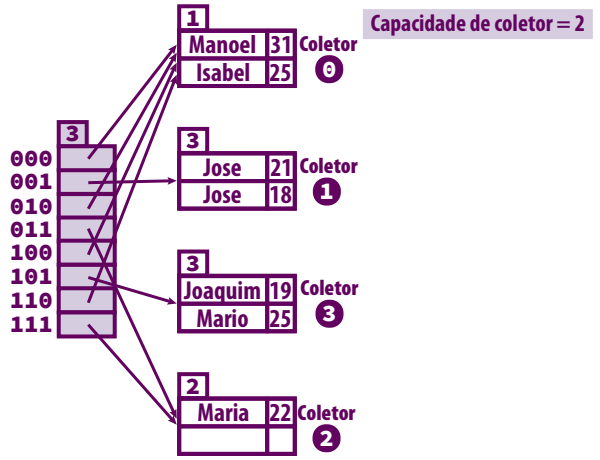


FIGURA 8–19: EXEMPLO DE DISPERSÃO EXTENSÍVEL: INSERÇÃO NORMAL DE TRÊS REGISTROS

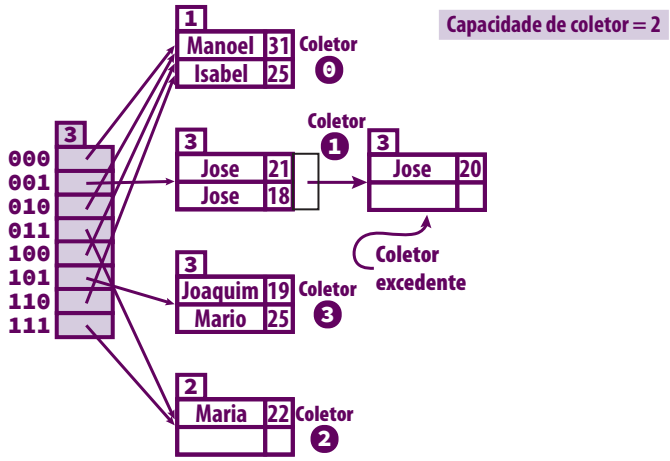


FIGURA 8–20: EXEMPLO DE DISPERSÃO EXTENSÍVEL COM USO DE COLETOR EXCEDENTE

8.2.6 Implementação

A implementação a ser apresentada aqui usa os bits menos significativos dos valores de dispersão das chaves conforme foi visto no exemplo da **Seção 8.2.5**, em vez de usar bits mais significativos desses valores, como aparece nas ilustrações apresentadas antes da referida seção. De fato, a implementação da operação de busca independe dessa escolha, mas a implementação da operação de inserção depende substancialmente dela. Se a remoção simples (i.e., sem compressão de diretório) for implementada, ela também não depende dessa escolha, mas, por outro lado, remoção completa com compressão de diretório sofre tal influência.

Definições de Tipos e Constantes

As seguintes definições de tipos serão usadas na implementação de dispersão extensível a ser desenvolvida aqui. O tipo `tRegistroMEC` que aparece na primeira destas definições de tipo é definido no **Apêndice A**.

```
   /* Tipo de uma variável que armazena um coletor */
typedef struct {
          tRegistroMEC registros[M]; /* Array de registros  */
          int          nRegistros;   /* Número de registros num coletor */
          int          profLocal;    /* Profundidade local  */
       } tColetorDExt;

   /* Tipo de uma variável que representa uma  */
   /* tabela de busca com dispersão extensível */
typedef struct {
          long *dir;        /* Diretório            */
          int   profGlobal; /* Profundidade global  */
          int   tamDir;     /* Tamanho do diretório */
          int   nReg;       /* Número de registros  */
       } tTabelaDExt;
```

O valor da constante `M`, que determina o número máximo de registros num coletor, é calculado de modo semelhante ao cálculo efetuado para determinação do número máximo de registros armazenados num coletor de dispersão estática (v. **Seção 8.1.3**). Assim o dimensionamento do número de elementos armazenados num coletor (i.e., da constante `M`) é obtido por intermédio das seguintes definições de macros:

```
#define TB 4096 /* Tamanho do bloco lido ou escrito  */
#define TR (int) sizeof(tRegistroMEC) /* Tamanho de um registro   */
                                      /* armazenado nos coletores */
#define TI (int) sizeof(int) /* Tamanho de um campo do tipo int */

   /* Cálculo do número máximo de registros num coletor */
#define M (((TB - (2*TI))/TR) - 1)
```

Iniciação

A função `IniciaTabelaDExt()` inicia uma tabela de busca com dispersão extensível e seus parâmetros são:

- `tabela` (entrada) — a tabela que será iniciada
- `stream` (entrada) — stream associado ao arquivo que contém os coletores

```
void IniciaTabelaDExt(tTabelaDExt *tabela, FILE *stream)
{
      /* A tabela é iniciada com um tamanho de diretório */
      /* igual a 1 que aponta para um coletor vazio      */

    tabela->profGlobal = 0;
    tabela->nReg = 0;
    tabela->tamDir = 1;
```

```
    tabela->dir = malloc(tabela->tamDir*sizeof(int));
    tabela->dir[0] = NovoColetorDExt(stream);
}
```

A função `NovoColetorDExt()`, chamada por `IniciaTabela()` e definida a seguir, cria um novo coletor. O único parâmetro dessa função é o stream associado ao arquivo que contém os coletores.

```
static int NovoColetorDExt(FILE *stream)
{
   tColetorDExt coletor;
   int          indice;

   coletor.nRegistros = 0;
   coletor.profLocal = 0;

      /* Move o apontador de posição do arquivo para seu final */
   MoveApontador(stream, 0, SEEK_END);

   indice = ftell(stream)/sizeof(coletor); /* Obtém o índice do coletor */

   fwrite(&coletor, sizeof(coletor), 1, stream); /* Escreve o coletor no arquivo */

   return indice;
}
```

As funções `MoveApontador()` e `ObtemApontador()` chamadas por `NovoColetor()` foram discutidas na **Seção 2.11.2**.

Busca

A função `BuscaDExt()` efetua uma busca numa tabela de busca com dispersão extensível e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo que contém os coletores
- `tab` (entrada) — a tabela de busca
- `chave` (entrada) — a chave de busca

Essa função retorna o índice do registro no arquivo de registros, se ele for encontrado, ou `-1`, em caso contrário. É importante observar que os elementos são ordenados por chave em cada coletor, de modo que busca binária poderia ter sido utilizada.

```
tRegistroMEC *BuscaDExt( FILE *stream, const tTabelaDExt *tab, tChave chave,
                         tRegistroMEC *reg )
{
   int      iColetor, i;
   tColetorDExt coletor;

      /* Obtém índice do coletor que contém o índice do registro */
   iColetor = tab->dir[ObtemBitsDeChave(chave, tab->profGlobal)];

      /* Lê o coletor que contém o índice do registro */
   LeColetorDExt(stream, iColetor, &coletor);

      /* Efetua uma busca sequencial pelo */
      /* registro em memória principal    */
   for (i = 0; i < coletor.nRegistros; i++)
         /* Verifica se a chave foi encontrada */
      if (chave == ObtemChave(coletor.registros + i)) {
         *reg = coletor.registros[i]; /* Chave encontrada */
         return reg;
      }
   return NULL; /* A chave não foi encontrada */
}
```

A função `LeColetorDExt()` chamada por `BuscaExt()` é semelhante à função `LeColetorDEst()` apresentada na **Seção 8.1.3**.

A função `ObtemBitsDeChave()`, chamada por `BuscaExt()` e definida abaixo, resulta nos bits menos significativos de uma chave e seus parâmetros são:

- `chave` (entrada) — a chave da qual os bits serão obtidos
- `bits` (entrada) — o número de bits que serão extraídos da chave

Essa função retorna um valor de dispersão inteiro que representa esses bits e o **Apêndice B** explica detalhadamente seu funcionamento.

```
static int ObtemBitsDeChave(tChave chave, int bits)
{
   int mascara = (1 << bits) - 1;

   return chave & mascara;
}
```

A função `ObtemBitsDeChave()` supõe que as chaves são bem distribuídas, de sorte que ela usa a própria chave (em vez de um valor de dispersão dela). Se for desejado usar uma função de dispersão adicional, a instrução de retorno da função `ObtemBitsDeChave()` deve ser substituída por:

```
return FuncaoDeDispersao(chave) & mascara;
```

sendo `FuncaoDeDispersao()` a função de dispersão utilizada.

Inserção

Para inserir um novo registro, primeiro, emprega-se o mesmo procedimento usado para busca, a fim de determinar em qual coletor o registro deve ser inserido. Todavia coletores têm tamanhos fixos, de modo que pode ser que não haja espaço no referido coletor para conter mais um registro. Para lidar com esse problema, deve-se criar um novo coletor e fazer com que um elemento do diretório faça referência a esse coletor. Além disso, pode ser necessário mover alguns registros do antigo coletor para o novo coletor.

A função `InsereDExt()` insere um novo registro numa tabela de busca com dispersão extensível e seus parâmetros são:

- `tab` (entrada/saída) — a tabela na qual será efetuada a inserção
- `item` (entrada) — o elemento que será inserido
- `stream` (entrada) — stream associado ao arquivo que contém os coletores

```
void InsereDExt( tTabelaDExt *tab, const tRegistroMEC *item, FILE *stream )
{
   int          i, j, iColetor;
   tChave       chave;
   tColetorDExt coletor;

   chave = ObtemChave(item); /* Obtém a chave do item */

      /* Obtém o índice do coletor */
   iColetor = tab->dir[ObtemBitsDeChave(chave, tab->profGlobal)];

   LeColetorDExt(stream, iColetor, &coletor); /* Lê o coletor em arquivo */

      /* O número de registros num coletor deve ser menor do que M */
   ASSEGURA( coletor.nRegistros < M, "Excesso de registros em coletor" );
      /* Como os registros são ordenado, é preciso encontrar */
      /* a posição de inserção do novo item no coletor       */
```

```
   for (i = 0; i < coletor.nRegistros; i++)
      if ( chave < ObtemChave(coletor.registros + i) )
         break;

      /* Abre espaço no coletor para o novo item */
   for (j = coletor.nRegistros; j > i; j--)
      coletor.registros[j] = coletor.registros[j - 1];

   coletor.registros[i] = *item; /* Insere o novo item no coletor */

      /* O número de registros no coletor aumentou */
   coletor.nRegistros++;

   ++tab->nReg; /* O número de registros na tabela também aumentou */

      /* Verifica se o coletor ficou repleto */
   if (coletor.nRegistros == M)
         /* O coletor ficou repleto e é preciso dividi-lo */
      DivideColetorDExt(tab, &coletor, iColetor, stream);
   else
         /* O coletor foi alterado e é necessário reescrevê-lo no arquivo */
      EscreveColetorDExt(stream, iColetor, &coletor);
}
```

Observe que, para facilitar a implementação, o número máximo de elementos em qualquer coletor é `M - 1`, de modo que sempre há espaço para inserção.

A função `EscreveColetorDExt()` chamada por `InsereExt()` é semelhante à função `EscreveColetorDEst()` apresentada na **Seção 8.1.3**.

A função `DivideColetorDExt()`, que será apresentada a seguir, divide um coletor de uma tabela de busca com dispersão extensível e tem como parâmetros:

- `tab` (entrada/saída) — a tabela de busca que contém o coletor que será dividido
- `coletor` (entrada) — o coletor que será dividido
- `iColetor` (entrada) — índice do coletor que será dividido no arquivo de coletores
- `stream` (entrada) — stream associado ao arquivo contendo os coletores

```
static void DivideColetorDExt( tTabelaDExt *tab, tColetorDExt *coletor,
                               int iColetor, FILE *stream )
{
   int          i, iNovo, chave, bits;
   tColetorDExt novo;

      /* Cria um novo coletor e obtém seu índice em arquivo */
   iNovo = NovoColetorDExt(stream);

   LeColetorDExt(stream, iNovo, &novo); /* Lê o novo coletor em arquivo */

      /* Divide registros do antigo coletor com o novo coletor */
   while (coletor->nRegistros == M) {
         /* Zera os registros do antigo e do novo coletor */
      coletor->nRegistros = 0;
      novo.nRegistros = 0;

         /* Distribui registros entre o antigo e o novo coletor */
      for (i = 0; i < M; i++) {
         /* Para cada item do antigo coletor, verifica-se se o bit de ordem       */
         /* profLocal + 1 da chave é 0 ou 1. Se ele for 0, o respectivo item fica */
         /* no antigo coletor. Caso contrário, esse item vai para o novo coletor  */

            /* Obtém a chave do item corrente */
```

```
         chave = ObtemChave(coletor->registros + i);

            /* Obtém profLocal + 1 bits menos significativos */
         bits = ObtemBitsDeChave(chave, coletor->profLocal + 1);

            /* Obtém o valor do bit mais significativo    */
            /* dessa sequência de bits e testa esse valor */
         if (ValorBit(bits, coletor->profLocal) == 0)
               /* Este item fica no antigo coletor */
            coletor->registros[coletor->nRegistros++] = coletor->registros[i];
         else
               /* Este item vai para o novo coletor */
            novo.registros[novo.nRegistros++] = coletor->registros[i];
      }

         /* O novo e o antigo coletor terão a mesma profundidade local, */
         /* que é a profundidade local do antigo coletor mais um        */
      novo.profLocal = ++coletor->profLocal;

         /* Se não ficou nenhum item no coletor antigo, é preciso  */
         /* repetir o processo usando uma profundidade local maior */
      if (novo.nRegistros == M)
            /* Copia os registros do novo coletor de volta para o antigo coletor */
         *coletor = novo;
      else if (novo.nRegistros == 0) {
            /* Verifica se o antigo coletor pode ser dividido. Se ele não pode */
            /* ser dividido, o novo registro deve ser inserido no novo coletor */
            /* que deverá se  tornar um coletor excedente. Como esta implemen- */
            /* tação não lida com esse caso, o programa será abortado se o     */
            /* coletor não puder ser dividido.                                 */
         ASSEGURA( PodeDividirColetorDExt(coletor),
                   "Erro: O coletor nao pode ser dividido\n" );

            /* Insere uma referência para o novo coletor vazio no diretório */
         InsereColetorVazioDExt(tab, coletor, iColetor, iNovo);

            /* Escreve o coletor vazio no arquivo */
         EscreveColetorDExt(stream, iNovo, &novo);

            /* Cria um novo coletor e o lê em arquivo */
         iNovo = NovoColetorDExt(stream);
         LeColetorDExt(stream, iNovo, &novo);
      }
   }

      /* Escreve os dois coletores no arquivo */
   EscreveColetorDExt(stream, iColetor, coletor);
   EscreveColetorDExt(stream, iNovo, &novo);

      /* Insere uma referência para o novo coletor no diretório */
   InsereEmDiretorioDExt(tab, &novo, iNovo);
}
```

A função `DivideColetorDExt()` cria um novo coletor e, então, examina o bit de ordem *k* (contando a partir do bit menos significativo) da chave de cada registro. Se esse bit for *0*, o registro fica no coletor antigo; se esse bit for *1*, o registro é deslocado para o novo coletor. O valor $k + 1$ é atribuído ao campo que representa a profundidade local de cada um desses dois coletores depois da divisão. Todavia pode acontecer que todas as chaves tenham o mesmo valor para o bit de ordem *k*, o que ainda deixa um desses coletores repletos. Se esse for o caso, prossegue-se usando o próximo bit até que se tenha pelo menos um registro em cada coletor. O processo deve

eventualmente terminar, a não ser que se tenham *M* ou mais chaves iguais ou com o mesmo valor de dispersão (v. abaixo). Ao final dessa divisão, insere-se uma referência para o novo coletor no diretório.

Quando *M* ou mais de registros têm chaves duplicadas, o algoritmo descrito na **Seção 8.2.2** entra em repetição infinita tentando fazer distinção entre as chaves. Quer dizer, esse algoritmo deixa de funcionar completamente se houver mais chaves iguais do que cabem num coletor. Assim é necessário acrescentar um teste à função `DivideColetorDExt()` para protegê-la contra a ocorrência desse problema. É para isso que a função `PodeDividirColetorDExt()` é chamada.

A função `PodeDividirColetorDExt()` verifica se um coletor pode ser dividido. Ou seja, essa função testa se todas as chaves de um coletor são iguais e, quando esse é o caso, ela retorna `0`, informando que o coletor recebido como parâmetro não pode ser dividido, de acordo com o que o foi exposto no último parágrafo. Quando o coletor pode ser dividido, essa função retorna `1`.

```
static int PodeDividirColetorDExt(const tColetorDExt *coletor)
{
   tChave chave;
   int    i;

      /* Esta função deve ser chamada apenas quando o coletor está repleto */
   ASSEGURA( coletor->nRegistros == M,
             "PodeDividirColetorDExt() nao deveria ter sido chamada" );

   chave = ObtemChave(coletor->registros); /* Obtém a chave do primeiro registro */

   for (i = 1; i < coletor->nRegistros; ++i)
      if ( chave != ObtemChave(coletor->registros + i) )
         return 1; /* Coletor pode ser dividido */

      /* As chaves deste coletor são todas iguais e, */
      /* portanto, ele não pode ser dividido         */
   return 0;
}
```

Quando se cria um novo coletor, é necessário inserir uma referência para ele no diretório. O caso mais simples é aquele no qual, antes da inserção, o diretório tem exatamente duas referências para o coletor que será dividido. Nesse caso, é necessário apenas fazer com que uma dessas referências aponte para o novo coletor. Se profundidade local do novo coletor for maior do que profundidade global do diretório, deve-se duplicar o tamanho do diretório para que ele possa acomodar o novo coletor e atualizam-se as referências do diretório para refletir essa duplicação.

A função `InsereEmDiretorioDExt()` duplica, quando é necessário, o diretório de uma de tabela de busca com dispersão extensível e acrescenta a ele uma referência para um novo coletor. Os parâmetros dessa função são:

- `tab` (entrada/saída) — a tabela de busca
- `pColetor` (entrada) — endereço do coletor que será inserido
- `iColetor` (entrada) — índice do coletor no arquivo de coletores

```
static void InsereEmDiretorioDExt( tTabelaDExt *tab, const tColetorDExt *pColetor,
                                   int iColetor )
{
   int    i, bits, pLocal;
   tChave chave;

      /* Obtém a chave do primeiro item do coletor */
   chave = ObtemChave(pColetor->registros);

   pLocal = pColetor->profLocal; /* Obtém a profundidade local do coletor */
```

```
      /* Duplica o tamanho do diretório enquanto sua profundidade */
      /* global for menor do que a profundidade local do coletor  */
   while (tab->profGlobal < pLocal) {
         /* Duplica o tamanho do diretório */
      tab->dir = realloc(tab->dir, 2*tab->tamDir*sizeof(long));
      ASSEGURA(tab->dir, "Impossivel alocar novo diretorio");

         /* Copia as referências do antigo diretório */
         /* para a metade final do novo diretório    */
      memcpy(tab->dir + tab->tamDir, tab->dir, tab->tamDir*sizeof(int));

         /* Atualiza a profundidade global e o tamanho do diretório */
      ++tab->profGlobal;
      tab->tamDir = 2*tab->tamDir;
   }
   /***                                                                ***/
   /* Se a profundidade local do novo coletor for menor do que a profundidade */
   /* global, será preciso atualizar mais de uma referência do diretório      */
   /***                                                                ***/

      /* Obtém os bits menos significativos coincidentes no coletor */
   bits = ObtemBitsDeChave(chave, pLocal);

      /* Corrige as referências do diretório para o novo coletor */
   for (i = 0; i < tab->tamDir; ++i)
      if (ObtemLSBs(i, pLocal) == bits)
         tab->dir[i] = iColetor;
}
```

A função `DivideColetorDExt()` chama `InsereColetorVazioDExt()` para inserir uma referência para um coletor vazio no diretório de uma tabela de busca. Essa última função é semelhante à função `InsereEmDiretorioDExt()` e sua necessidade é esclarecida na **Seção 8.2.5**. Os parâmetros da função `InsereColetorVazioDExt()` são:

- `tab` (entrada/saída) — a tabela de busca
- `pColetor` (entrada) — endereço do coletor que deu origem ao coletor vazio
- `iColetor` (entrada) — índice do coletor que deu origem ao coletor vazio no arquivo de coletores
- `iVazio` (entrada) — índice do coletor vazio no arquivo de coletores

```
static void InsereColetorVazioDExt( tTabelaDExt *tab, const tColetorDExt *pColetor,
                                    int iColetor, int iVazio )
{
   int    i, bits, pLocal;
   tChave chave;

   pLocal = pColetor->profLocal; /* Obtém a profundidade local do coletor */

      /* Duplica o tamanho do diretório enquanto sua profundidade global */
      /* for menor do que a profundidade local do coletor                */
   while (tab->profGlobal < pLocal) {
         /* Duplica o tamanho do diretório */
      tab->dir = realloc(tab->dir, 2*tab->tamDir*sizeof(int));
      ASSEGURA(tab->dir, "Impossivel alocar novo diretorio");

         /* Copia as referências do antigo diretório */
         /* para a metade final do novo diretório    */
      memcpy(tab->dir + tab->tamDir, tab->dir, tab->tamDir*sizeof(int));

         /* Atualiza a profundidade global e o tamanho do diretório */
      ++tab->profGlobal;
```

```
      tab->tamDir = 2*tab->tamDir;
   }

      /* Obtém a chave do primeiro item do coletor que deu origem ao coletor vazio */
   chave = ObtemChave(pColetor->registros);

      /* Obtém os bits menos significativos coincidentes no */
      /* coletor que deu origem ao coletor vazio            */
   bits = ObtemBitsDeChave(chave, pLocal);

      /* Corrige as referências para o coletor vazio */
   for (i = 0; i < tab->tamDir; ++i)
      if (tab->dir[i] == iColetor && ObtemLSBs(i, pLocal) != bits)
         tab->dir[i] = iVazio;
}
```

Remoção

A função `RemoveExt()` remove um registro de uma tabela de dispersão extensível e tem como parâmetros:

- `tab` (entrada/saída) — a tabela de dispersão
- `chave` (entrada) — a chave do registro que será removido
- `stream` (entrada) — stream associado ao arquivo que contém os coletores

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

```
int RemoveDExt(tTabelaDExt *tab, tChave chave, FILE *stream)
{
   int          i, j, iColetor;
   tColetorDExt coletor;

      /* Obtém o índice do coletor */
   iColetor = tab->dir[ObtemBitsDeChave(chave, tab->profGlobal)];

   LeColetorDExt(stream, iColetor, &coletor); /* Lê o coletor em arquivo */

      /* Efetua uma busca sequencial pelo item em memória principal */
   for (i = 0; i < coletor.nRegistros; i++)
         /* Verifica se a chave foi encontrada */
      if (chave == ObtemChave(coletor.registros + i))
         break;

      /* Verifica se a chave foi encontrada */
   if (i >= coletor.nRegistros)
      return 0;  /* A chave não foi encontrada */

      /* O item a ser removido se encontra na posição i. É necessário mover uma */
      /* posição para trás todos os registros que se encontram à sua frente.    */
   for (j = i; j < coletor.nRegistros - 1; ++j)
      coletor.registros[j] = coletor.registros[j + 1];

   --coletor.nRegistros; /* O número de registros no coletor diminuiu */

   --tab->nReg; /* O número de registros na tabela também diminuiu */

      /* Verifica se o coletor ficou vazio */
   if (coletor.nRegistros == 0)
         /* O coletor ficou vazio. Talvez o tamanho do diretório possa ser       */
         /* reduzido, mas é possível que, mesmo assim, essa operação não valha a */
         /* pena.  Substituir a instrução vazia a seguir pela implementação de   */
         /* uma operação que reduza o tamanho do diretório fica como exercício.  */
      ;
```

```
      /* O coletor foi alterado e é preciso reescrevê-lo no arquivo */
   EscreveColetorDExt(stream, iColetor, &coletor);

   return 1;  /* Remoção bem-sucedida */
}
```

8.2.7 Análise

Vantagens

Uma propriedade interessante de qualquer tabela de dispersão extensível é que sua configuração depende apenas dos valores das chaves dos registros que a constituem, e não depende da ordem na qual esses registros são inseridos na tabela.

A principal vantagem de dispersão extensível é que o desempenho de suas operações não é degradado quando o número de registros na tabela cresce muito, como ocorre com dispersão estática. Isso ocorre porque o uso de dispersão extensível evita uma reorganização do arquivo que contém os coletores quando a capacidade do diretório é excedida. Nessa situação, apenas o diretório é afetado. Ou seja, apenas os valores de dispersão das chaves envolvidas nesse processo são recalculados. Como, frequentemente, o diretório é mantido em memória principal, o custo de expansão e atualização dele é muito baixo.

Desvantagens

O fato de dispersão extensível raramente usar coletores excedentes (v. **Seção 8.2.5**) pode aumentar de modo significativo o tamanho do diretório, visto que uma única operação de inserção pode fazer com que o diretório seja duplicado várias vezes.

Mesmo quando um coletor pode ser dividido, o diretório pode se tornar desnecessariamente imenso se as chaves apresentarem um número excessivo de bits iniciais coincidentes usados na indexação. Além disso, o tamanho de um diretório não cresce proporcionalmente, pois ele é duplicado quando um coletor com profundidade local igual à profundidade do diretório é dividido, o que pode fazer com que haja muitos elementos redundantes no diretório. Ademais, quando a tabela possui um grande número de coletores pequenos, o tamanho do diretório pode se tornar tão grande que pode ser necessário mantê-lo em memória secundária, o que irá requerer um acesso adicional a esse meio de armazenamento.

Árvores B+, que exibem custos logaritmos para operações de entrada e saída (v. **Seção 6.5.7**), são quase tão eficientes quanto tabelas de dispersão extensível, que têm custo $\theta(1)$ para as mesmas operações. Além do mais, árvores B+ permitem outros tipos de busca, que não são facilmente executadas com tabelas de dispersão extensível. Ou seja, dispersão extensível dá suporte apenas a busca exata. Buscas de intervalo, por exemplo, que podem ser facilmente implementadas com árvores B+, não são facilmente implementadas com dispersão extensível. Por isso, a maioria dos sistemas comerciais de gerenciamento de bancos de dados usam apenas árvores B+.

Na implementação de dispersão extensível apresentada na **Seção 8.2.6** foram utilizadas representações binárias das próprias chaves como índices do diretório, em vez de valores de dispersão dessas chaves, mas isso nem sempre é conveniente. Muitos conjuntos de chaves não exibem distribuição uniforme, de sorte que, se elas forem usadas diretamente, o diretório pode tornar-se bem maior do que deveria ser. Por outro lado, uma boa função de dispersão produz valores de dispersão uniformemente distribuídos, o que garante que alguns coletores não sejam favorecidos em detrimento de outros durante operações de inserção.

Custo Espacial

Pode-se mostrar que, usando-se coletores que contenham no máximo m registros, uma tabela de dispersão extensível contendo n registros requer cerca de $1{,}44 \cdot n/m$ coletores e o tamanho esperado do diretório é cerca de:

$$\frac{3,92}{m} n^{(1+\frac{1}{m})}$$

Assim a taxa de crescimento do tamanho do diretório com relação a n é mais acentuada do que uma taxa linear, principalmente para valores pequenos de m. Entretanto, para valores típicos de n e m, $n^{1/m}$ é bem próximo de 1, de forma que se pode esperar que, na prática, o diretório tenha cerca de $4 \cdot n/m$ elementos.

Com relação à eficiência de uso de espaço, o artigo original sobre dispersão extensível sugere que a utilização média de espaço é 69% do espaço total alocado para todos os coletores. Isso é comparável a árvores B, que têm taxa de utilização de espaço próxima de 67% (v. **Seção 6.4.8**). À medida que coletores são preenchidos, a utilização aproxima-se de 90%, mas, a partir desse ponto, eles são divididos e a taxa de utilização cai para cerca de 50%.

8.3 Avaliação de Dispersão em Memória Secundária

Uma aparente desvantagem de dispersão extensível em relação à dispersão estática é que ela requer acesso indireto. Ou seja, antes de acessar um coletor, é necessário acessar o diretório antes. Entretanto, como, na maioria das vezes, o diretório é mantido em memória principal, esse acréscimo no custo pode ser considerado desprezível.

Se o uso de coletores excedentes não puder ser evitado ou substancialmente aliviado em dispersão extensível, é mais aconselhável usar dispersão estática, pois, pelo menos, a implementação de dispersão estática é bem mais simples. Mas é bom lembrar que o uso de dispersão estática pode requerer múltiplos acessos à memória secundária se uma dada operação precisar acessar coletores excedentes.

Em comparação com árvores da família B (i.e., árvores B, B+, etc.) discutidas no **Capítulo 6**, as tabelas de dispersão apresentadas neste capítulo podem até ser mais rápidas e ocupar menos espaço, mas apresentam como desvantagens:

- ❒ Não permitem encontrar o registro com a menor ou maior chave (v. **Seção 6.8.2**)
- ❒ Não permitem busca de intervalo (v. **Seção 6.8.6**)
- ❒ Não facilitam busca pelo piso ou teto de uma chave
- ❒ Não admitem bulkloading (v. **Seção 12.6**)

8.4 Exemplos de Programação

8.4.1 Coletores Excedentes em Tabela de Dispersão Estática

Problema: Escreva uma função que calcula o número de coletores excedentes numa tabela de dispersão estática considerando a definição de tipos apresentado na **Seção 8.1.3**.

Solução: A função `NColetoresExcedentesDEst()` calcula e retorna o número de coletores excedentes numa tabela de dispersão estática e seus parâmetros são:

- ■ `nColetores` (entrada) — número de coletores primários
- ■ `stream` (entrada) — stream associado ao arquivo que contém os coletores

```
int NColetoresExcedentesDEst(int nColetores, FILE *stream)
{
   int          i, n = 0;
   tColetorDEst coletor;

      /* Para cada coletor primário, conta quantos coletores excedentes ele possui */
   for (i = 0; i < nColetores; ++i) {
      LeColetorDEst(stream, i, &coletor);

         /* Conta os coletores excedentes do coletor primário corrente  */
```

```
      while (coletor.proximo != POSICAO_NULA) {
          ++n;
          LeColetorDEst(stream, coletor.proximo, &coletor);
      }
   }

   return n;
}
```

8.4.2 Maior Profundidade Local em Tabela de Dispersão Extensível

Problema: Escreva uma função que calcula a maior profundidade local encontrada numa tabela de dispersão extensível.

Solução: A função `ProfLocalMaxDExt()` calcula e retorna a profundidade local máxima. Seu único parâmetro é o stream associado ao arquivo que contém os coletores.

```
int ProfLocalMaxDExt(FILE *stream)
{
   int          max;
   tColetorDExt umColetor;

   rewind(stream); /* Assegura que a leitura começa no início do arquivo */

      /* Lê um coletor */
   fread(&umColetor, sizeof(umColetor), 1, stream);

      /* Verifica se ocorreu erro de leitura */
   ASSEGURA( !ferror(stream), "Erro de leitura em ProfLocalMaxDExt" );

   max = umColetor.profLocal;

   while (1) {
      fread(&umColetor, sizeof(umColetor), 1, stream); /* Lê um coletor */

         /* Se ocorreu erro de leitura ou o final  */
         /* do arquivo foi atingido encerra o laço */
      if (feof(stream) || ferror(stream))
         break;

      if (umColetor.profLocal > max)
         max = umColetor.profLocal;
   }

      /* Verifica se ocorreu erro de leitura */
   ASSEGURA( !ferror(stream), "Erro de leitura em ProfLocalMaxDExt" );

   return max;
}
```

8.5 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.

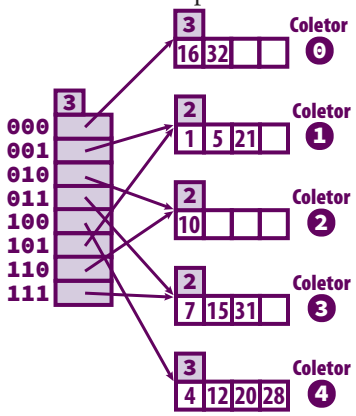
Dispersão Estática (Seção 8.1)

1. Descreva a técnica de dispersão estática.
2. (a) O que é um coletor primário em dispersão estática? (b) O que é um coletor excedente nessa técnica de dispersão?

3. Descreva as operações de busca, inserção e remoção numa tabela de dispersão estática.
4. O que prejudica o desempenho de operações com tabelas de busca implementadas com dispersão estática?
5. (a) Quais são as causas de ocorrências de coletores excedentes numa tabela de dispersão estática? (b) O que pode ser feito para reduzir essas ocorrências?
6. (a) Quais são as semelhanças entre tabela de dispersão estática e tabela de dispersão com encadeamento (discutida no **Capítulo 7**)? (b) Quais são as diferenças entre tabela de dispersão estática e tabela de dispersão com encadeamento?
7. (a) O que é remoção simples numa tabela de dispersão estática? (b) O que é remoção completa numa tabela de dispersão estática?
8. Por que redimensionamento de uma tabela de dispersão estática não é viável, como ocorre com dispersão com endereçamento aberto visto no **Capítulo 7**?

Dispersão Extensível (Seção 8.2)

9. O que é dispersão extensível?
10. Em que situações práticas utiliza-se dispersão extensível?
11. Descreva os seguintes conceitos usados em dispersão extensível:
(a) Profundidade global
(b) Profundidade local
(c) Diretório
12. (a) Para que serve a profundidade local de um coletor? (b) Qual é a relação entre a profundidade local de um coletor e a profundidade global do diretório numa tabela de dispersão extensível?
13. Sejam g a profundidade global de um diretório de uma tabela de dispersão extensível e l a profundidade local de um coletor da mesma tabela. Quantos elementos do referido diretório apontam para esse coletor?
14. Por que o diretório usado em dispersão extensível é denominado *tabela extensível*?
15. Apresente graficamente o resultado de inserção das chaves em formato binário *10111101*, *00000010*, *10011011*, *10111110*, *01111111*, *01010001*, *10010110*, *00001011*, *11001111*, *10011110*, *11011011*, *00101011*, *01100001*, *11110000*, *01101111* numa tabela de dispersão extensível inicialmente vazia supondo que os bits menos significativos das chaves sejam usados na indexação do diretório. Suponha que a capacidade de um coletor é de quatro registros.
16. Repita o exercício **15** supondo que os bits mais significativos das chaves sejam usados na indexação do diretório.
17. Considere a tabela de dispersão extensível ilustrada na figura a seguir, na qual os valores de dispersão das chaves dos registros aparecem no interior dos coletores. Suponha que a indexação seja efetuada considerando os bits menos significativos dos valores de dispersão.



(a) Apresente a tabela resultante da inserção de um registro cujo valor de dispersão de sua chave é *44*.
(b) Apresente a tabela resultante de inserções na tabela original dos registros cujos valores de dispersão de suas chaves são *25* e *77*.
(c) Apresente a tabela resultante da remoção do registro cujo valor de dispersão de sua chave é *10*.

18. (a) Como são obtidos os bits menos significativos de uma chave inteira? (b) Como são obtidos os bits mais significativos de uma chave inteira?

19. Por que obter os bits mais significativos de um valor inteiro é mais complicado do que obter bits mais significativos do mesmo valor?

20. (a) Qual é a vantagem decorrente do uso de bits mais significativos em detrimento a bits menos significativos na indexação do diretório de uma tabela de dispersão extensível? (b) Qual é a vantagem decorrente do uso de bits menos significativos em vez de bits mais significativos nessa indexação?

21. (a) Por que coletores excedentes são às vezes necessários em dispersão extensível? (b) Quando o uso de coletores excedentes se faz necessário em dispersão extensível?

22. Se coletores excedentes são usados em dispersão extensível, qual é a vantagem desse tipo de organização de tabela de busca em relação a dispersão estática?

23. (a) Quando o algoritmo de inserção para dispersão extensível descrito na **Seção 8.2.2** entra em repetição infinita? (b) O que pode ser feito para evitar esse problema?

24. Explique a inserção de coletores vazios aparentemente redundantes em tabelas de dispersão extensível.

25. Qual é a consequência do uso de chaves duplicadas em extensão extensível?

26. Explique o funcionamento do mecanismo de duplicação de diretório em extensão extensível.

27. O que são coletores camaradas?

28. (a) Quando a remoção de uma chave de uma tabela de dispersão extensível permite que coletores sejam recombinados? (b) Quando a remoção de uma chave de uma tabela de dispersão extensível permite que um diretório seja comprimido?

29. Quando uma redução de tamanho de diretório de uma tabela de dispersão extensível é recomendável? Explique sua resposta.

30. (a) Suponha que uma inserção cause duplicação do diretório de uma tabela de dispersão extensível. Quantos coletores terão exatamente uma referência para si nesse diretório? (b) Se, após uma remoção subsequente de registro, um desses coletores ficar vazio, o referido diretório pode ser comprimido?

31. Quais pré-requisitos devem ser satisfeitos para que uma tabela de dispersão extensível garanta que apenas um acesso ao meio de armazenamento externo seja necessário para operações de busca, inserção e remoção?

32. Supondo que os bits menos significativos das chaves sejam usadas na indexação do diretório de uma tabela de dispersão extensível, quando ele for duplicado, quais serão os elementos desse diretório que precisarão ser alterados?

Avaliação de Dispersão em Memória Secundária (Seção 8.3)

33. (a) Uma implementação de tabela de dispersão com encadeamento é conveniente para memória secundária? (b) Uma implementação de tabela de dispersão com endereçamento aberto é conveniente para memória secundária?

34. Quais são as vantagens e desvantagens do uso de dispersão extensível com relação a dispersão estática?

35. Quais são as vantagens e desvantagens do uso de dispersão extensível em comparação com árvores B+?

Exemplos de Programação (Seção 8.4)

36. Qual é o custo de transferência da função `NColetoresVaziosDEst()` em termos do número de registros armazenados na tabela de dispersão?

37. Descreva o funcionamento da função `ProfLocalMaxDExt()` que calcula a profundidade local máxima encontrada numa tabela de dispersão extensível.

8.6 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.

EP8.1 Escreva uma função que calcula o tamanho da maior cadeia de coletores excedentes de uma tabela de dispersão estática considerando a definição de tipos apresentado na **Seção 8.1.3**.

EP8.2 A implementação de dispersão extensível apresentada na **Seção 8.2.6** não usa coletores excedentes, de modo que um programa-cliente que usa tal implementação é abortado quando um coletor não pode ser dividido. Elabore uma nova implementação baseada naquela apresentada na referida seção que considere o uso de coletores excedentes.

EP8.3 A implementação apresentada na **Seção 8.2.6** usa chave externa. Elabore uma nova implementação baseada naquela apresentada na referida seção que use chave interna e compare sua implementação com aquela apresentada na referida seção.

EP8.4 Escreva uma função que calcula o número de registros armazenados numa tabela de dispersão extensível.

EP8.5 Escreva uma função que calcula o número total de coletores de uma tabela de dispersão extensível.

EP8.6 Escreva uma função que calcula o número total de coletores vazios de uma tabela de dispersão extensível.

EP8.7 Escreva uma função que calcula a menor profundidade local encontrada numa tabela de dispersão extensível.

EP8.8 Escreva uma função que calcula o número de coletores primários vazios numa tabela de dispersão estática considerando a definição de tipos apresentado na **Seção 8.1.3**.